Instituto Politécnico
de Viana do Castelo

PLATAFORMA DE ANIMAÇÃO E TRANSIÇÃO DE UM AVATAR 3D PARA LINGUA GESTUAL PORTUGUESA

AN ANIMATION AND TRANSITION PLATFORM FOR A 3D AVATAR IN PORTUGUESE SIGN LANGUAGE

Masters Thesis

Duarte Nuno Carvalho Dias

Escola Superior de Tecnologia e Gestão

Instituto Politécnico
de Viana do Castelo

Duarte Nuno Carvalho Dias

PLATAFORMA DE ANIMAÇÃO E TRANSIÇÃO DE UM AVATAR 3D PARA LINGUA GESTUAL PORTUGUESA

AN ANIMATION AND TRANSITION PLATFORM FOR A 3D AVATAR IN PORTUGUESE SIGN LANGUAGE

Course name:

Mestrado em Engenharia Informática

Trabalho efetuado sob a supervisão de:

Professor Doutor Pedro Miguel Teixeira Faria

Professor Doutor Luís Miguel Cabrita Romero

17 Maio 2024

This work is part of the project entitled "IVLinG: Interprete Virtual de Língua Gestual," funded by the European Regional Development Fund (FEDER) through the Compete2020 program, project code POCI-01-0247-FEDER-068605.

Abstract

Portuguese Sign Language (LGP) is one of Portugal's three official languages, alongside Portuguese and Mirandese, and is the primary means of communication for the country's deaf community. The problem, despite LGP's official status, is that only around 30,000 people speak it as their mother tongue. This figure contrasts sharply with Portugal's population of approximately 11 million. LGP users represent only around 1% of the total population. Furthermore, LGP has a different structure and grammar to Portuguese. When trying to read and understand texts written in Portuguese correctly, this linguistic divergence constitutes a significant barrier for most deaf people. Although numerous technologies facilitate seamless translation between various spoken languages, the availability of translators to bridge the gap between a spoken language and a sign language remains considerably limited. As a result, communication with users and non-users of LGP becomes extremely difficult.

In this context, the work carried out as part of the IVLinG - Interprete Virtual de Língua Gestual project, described here, had as its main objective the design and development of an animation component capable of animating gestures in LGP. To this end, studies were carried out on translation systems in Portuguese Sign Language, and various methods were developed and used with other languages. The outcome of this endeavour led to the creation of an animation platform that leverages the user's machine's hardware capabilities through an animation system that receives text input, translates it to an LGP gesture, and animates said gesture, using a 3D Avatar directly on the user's machine.

Keywords: Deaf, Assistive Technology, Portuguese Sign Language, Sign Language Animation, Avatar, Animation, 3D Model, Glosses, Unity, Blender, UPBGE;

Resumo

A Língua Gestual Portuguesa (LGP) é uma das três línguas oficiais de Portugal, a par do português e do mirandês, e é o principal meio de comunicação da comunidade surda do país. O problema, apesar do estatuto oficial da LGP, é que apenas cerca de 30.000 pessoas a falam como língua materna. Este número contrasta fortemente com a população portuguesa de aproximadamente 11 milhões de habitantes. Ou seja, os utilizadores de LGP constituem apenas cerca de 1% de toda a população.

Para além disso, a LGP tem uma estrutura e uma gramática diferentes da portuguesa. Ao tentar ler e compreender corretamente textos escritos em português, esta divergência linguística constitui uma barreira significativa para a maioria das pessoas surdas. Embora numerosas tecnologias facilitem a tradução sem problemas entre várias línguas faladas, a disponibilidade de tradutores para fazer a ponte entre uma língua falada e uma língua gestual continua a ser consideravelmente limitada. Consequentemente, a comunicação com utilizadores e não utilizadores de LGP torna-se extremamente difícil.

Neste contexto, o trabalho realizado no âmbito do projeto IVLinG - Virtual Sign Language Interpreter, aqui descrito, teve como principal objetivo a conceção e desenvolvimento de uma componente de animação capaz de animar gestos em LGP. Para o efeito, foram realizados estudos sobre sistemas de tradução em Língua Gestual Portuguesa e foram desenvolvidos e utilizados vários métodos com outras línguas.

O resultado deste trabalho conduziu à criação de uma plataforma de animação que tira partido das capacidades de hardware da máquina do utilizador através de um sistema de animação que recebe texto, traduz em gesto LGP e anima esse gesto, usando um Avatar 3D, diretamente na máquina do utilizador.

Palavras-chave: Surdo, Tecnologia Assistiva, Língua Gestual Portuguesa, Animação de Língua Gestual, Avatar, Animação, Modelos 3D, Glosses, Unity, Blender, UPBGE;

Acknowledgments

Firstly, I would like to express my sincere gratitude to Professor Luís Romero and Professor Pedro Faria, who have served as my advisors over the past years, for their priceless advice, support, and criticism.

A special thanks go out to all of the instructors I had for both my bachelor's and master's degrees for all of their guidance and instruction over this drawn-out process, and I would especially like to thank Ângelo Costa and Vânia Ferreira for their invaluable assistance with Portuguese Sign Language. I would especially like to thank Beatriz Miranda, Bruno Ribeiro, David Verde, Tânia Silva, and Vasco Alves, my friends and coworkers, for their support, assistance, and happy memories. In addition, I want to thank all my friends for helping and inspiring me during this process.

Finally, I want to thank my family for their constant encouragement and belief in me.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AR Augmented Reality

BGE Blender Game Engine

DSR Design Science Research

IK Inverse Kinematics

ISL Indian Sign Language

LGP Portuguese Sign Language

LP Portuguese Language

MoCap Motion Capture

TSL Thai Sign Language

UPBGE Uchronia Project Blender Game Engine

VR Virtual Reality

WebGL Web Graphics Library

XNA Xbox/DirectX New generation Architecture

Chapter 1

Introduction

1.1 Context

One of Portugal's three official national languages is Portuguese Sign Language, alongside Portuguese Language and Mirandese, and the deaf community primarily uses it to enhance communication. According to data from the Portuguese Deaf Association, Portugal is home to approximately 30,000 members of this community [1]. However, when we consider their more comprehensive network of connections, such as their relatives, teachers, and coworkers, the total number of Portuguese Sign Language users rises to an estimated 110,000. Surprisingly, this figure only accounts for 1% of the entire population of the country, according to data from the 2021 Census. This glaringly low percentage highlights the enormous difficulties deaf people encounter when attempting to interact with the general population. Given the limited public understanding of sign language, these obstacles are frequently so overwhelming that effective communication seems almost impossible. Despite this, nearly everyone may already have access to a promising solution. Why not use these tools to bridge the gap between spoken language and sign language, much like smartphones, which have become commonplace for seamless translations between Portuguese spoken language and other languages?

-

1.2 Objectives

The main objective of this work was to study and explore various techniques for the creation of a transition in the animations used by a 3D Avatar for translating Portuguese into LGP, to identify multiple solutions for realistic animation of the avatar and to create an animation module that can be integrated into an application designed to improve communication difficulties between deaf or hard of hearing people and non-deaf people by presenting signs in LGP through an animated 3D avatar. To achieve this goal, the following objectives must be met:

- **Research and comparison of existing tools:** To choose the approach(es), it will be necessary to research to comprehend the state of the art of the tools and techniques used in sign language avatar animation;
- **Development of an animation transition system:** After researching the existing tools and techniques, develop one or more systems of animation;
- **Testing of the developed systems:** After creating a system, it will be tested extensively by the target audience, in this case Portuguese Sign Language (LGP) native speakers;
- **Evaluation and discussion of the results of testing both systems:** After the tests are completed, a careful analysis of the results is conducted and discussed to evaluate the animation system developed.

1.3 Research Methodology

The Design Science Research methodology was used to carry out the work described here. This methodology is based on creating and evaluating artefacts to solve problems, involving well-defined processes such as designing and evaluating the artefacts produced, producing scientific contributions, and the presentation of the results obtained. The authors of [2], following a review of various published works on DSR, identified recurring points in all of them to suggest a DSR approach for developing and evaluating research in information systems. These points are as follows.

- **Problem identification and motivation:** Describe the specific issue, which in this case is the communication barrier that deaf individuals encounter when trying to interact with non-deaf people and the need for a solution in a society where many people have smartphones;
- **Define the objectives for a solution:**

  The primary goal of this solution is to create a platform where a user may enter a message that will be animated and understandable to an LGP speaker. This platform should be easily accessible by anyone and provide realistic animations of LGP gestures;
- **Design and development:** Has the name implies, design the architecture of the animation platform and develop it;
- **Demonstration:** Demonstrate how the animation platform functions and how deaf individuals can use it to enhance their quality of life;
- **Evaluation:** Show the developed platform to several LGP and deaf speakers to get their opinions on the solutions developed;
- **Communication:** Share the problem and its importance, the solution created for it, its utility, and its effectiveness. This can be achieved by publishing articles in conferences or journals.

1.4 Organization

The organization of the work is briefly discussed in this sub-chapter, along with each of the chapter's contents. The following eighth chapters comprise this piece of writing:

- **1- Introduction:** This chapter discusses the background, motivations, and goals for why it is crucial to complete this work. Additionally, the method of development and organizational structure are presented;
- **2- Theoretical Framework:** The second chapter discusses the concepts around the primary technology used in the development of the animation and transition system, as well as core concepts about Portuguese Sign Language;

- **3- Related Work:** The third chapter focuses on the presentation of work already carried out related to sign language avatars and animation systems;
- **4- System architecture and development:** The fourth chapter presents the general architecture of the entire system, exploring its various components. In addition, a description is given of the system implementations built using the architectures mentioned above, as well as the development decisions and processes involved;
- **5- Results analyses:** This chapter analyzes the results obtained regarding testing the implemented animation transition component. These results focus on both the quality of the translations produced by the element and how efficient the component is regarding the whole system;
- **6- Conclusions and future work:** This chapter presents a summary of the work carried out, discussing results achieved and contributions produced, suggesting possible continuations of this work;
- **7- References:** This chapter provides a list of sources consulted during the project creation described in this document;
- **8- Appendices:** In this chapter, there are documents complementary or supportive to the development of this work, as well as articles published in international conferences and code developed.

Chapter 2

Concepts and Technology

This chapter gives some theoretical background to help readers better comprehend the features of LGP and its distinctions from other languages. Followed by an explanation of some software that will be used in the project, motion capture, and concepts of 3D animation

2.1 Portuguese Sign Language

Portuguese Sign Language, one of Portugal's three official national languages, is primarily employed by the deaf community to improve communication. Around 30,000 members of this community reside in Portugal, according to data from the Portuguese Deaf Association [1]. However, the total number of Portuguese Sign Language users rises to an estimated 110,000 when we consider their more comprehensive network of connections, including their relatives, teachers, and coworkers[3]. Surprisingly, according to information from the 2021 Census, this number only represents 1% of the nation's total population. This conspicuously low percentage emphasizes the enormous challenges that deaf people face when attempting to interact with the general population.

2.1.1 Characteristics of Portuguese Sign Language

Portuguese Sign Language, often referred to just as LGP, like other sign languages, possesses some key characteristics, also known as "*queremas*" [4] [5]:

- **Hand and Body Configuration:** It represents the position the dominant hand,

or each hand individually, takes while making the gesture. The Portuguese manual alphabet and numbering are used as the basis for LGP. Figure 2.1 shows an example of hand configuration. Body posture can be used to specify referents (subjects or objects) and can also be used as a reference point;

Figure 2.1: Hand configuration for the numbers 1, 2 and 3, respectively.

- **Hand Location:** Position where the configured hand makes the gesture. There are two ways to use this position. The first is where the gesture is initially created in an imaginary space encompassing the front of the torso and the head, with no physical contact between the hand and the body. The second involves making contact with your own body, including your ears, neck, and shoulders, as can be seen in figure 2.2;

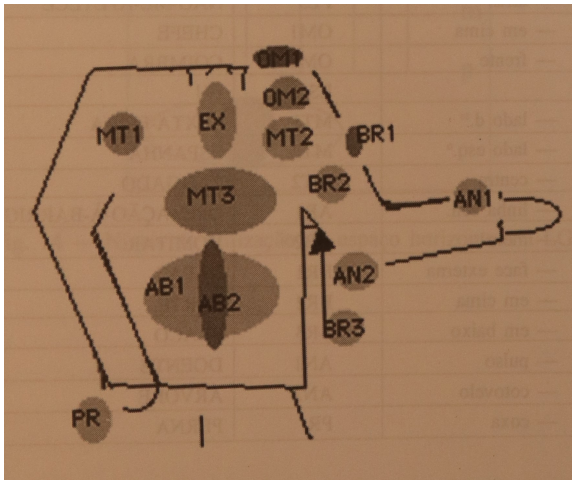

Figure 2.2: Areas of articulations of gestures in LGP.

- **Hand Orientation:** The direction the palm faces while performing the motion. The inversion of orientation can convey the concept of opposition, contradiction, or number agreement;

- **Movement:** Once set up, the hands can move through their joints, but some gestures don't require movement. A single movement alone or a combination of several

movements can produce the gesture. Fast, slow, long, short, smooth, and tense movements can all be used to distinguish the action of a movement;

- **Facial Expression:** Facial expression is crucial because it gives what is being represented emotion. It is possible to tell through fluid expression whether something is a question, exclamation, denial, affirmation, admiration, or fear, among other things. Some gestures are only differentiated by facial expression;

2.1.2 Glosses

A sign language sentence is represented in written text by glosses [4] [5] [6]. Each term in the gloss is a different gesture in the sentence. Typically, the gloss is written in all capital letters and uses the "+" symbol to represent a separation of gestures, the "-" symbol to represent the separation of letters in a word and or numerical digits respectively, and the "/" and "//" symbols to represent a short and long pause respectively. For example, the phrase "O Duarte comeu frango." in glosses is "FRANGO+D-U-A-R-T-E+COMER".

2.1.3 Grammatical and Structural Features of Portuguese Sign Language

Considering there are few resources on the grammar and structure of LGP, the characteristics listed below are primarily the product of analyzing some works done on LGP, such as [4] [5] [6], as well as the results of meetings with several interpreters and LGP users.

- **Sentence Structure:** An LGP sentence follows the OSV (Object-Subject-Verb) structure. This is not the case in Portuguese, where the proper form is SVO (Subject-Verb-Object). Some claim that LGP uses SOV (Subject-Object-Verb). Nevertheless, LGP users can understand SOV and OSV. Most of the time, non-manual expressions, typically facial ones, convey whether a sentence is a question by tilting the head slightly or a negative statement by shaking the head side to side. On the other hand, in certain situations, a question is indicated with question words(e.g., "when," "who," "how many"), and the negative is determined by adding a sign for "NO" at the end;

- **Names:** As far as personal names are concerned, most LGP users have a sign name that they choose to use to identify themselves to other LGP speakers. There is no relationship between the sign name and the written name because each person prefers their sign name, and it is impossible to translate the names. Nonetheless, when addressing someone without knowing their sign name or when they don't have a sign name, one uses their written name, which can also be done in terms of translation.

- **Verbs:** In LGP, verbs are not conjugated. Instead, each verb has a single sign, usually written in the infinitive. LGP users can use facial expressions, time adverbs (such as yesterday, tomorrow), or even signs like "past," "was," and "soon," which denote the past negative, past-positive, and future, respectively, to indicate when the action of the verb was set in LGP;

- **Genders:** Unlike Portuguese, which has two grammatical genders, masculine and feminine, LGP lacks a gender the majority of the time. The sign for "woman" is added before the actual sign, acting as a prefix, to define it as feminine when it is necessary to distinguish between genders. Users typically omit the prefix when a sign is marked as masculine, making the sign automatically masculine. This indicates that the prefix "woman" allows for differentiation between a noun's two genders in certain situations;

- **Iconicity:** Several signs in sign languages attempt to mimic real-world objects, making them apparent to non-sign language users as well. They refer to these signs as "iconic." In a virtual world, referential signs, such as pointing at someone or something, are more challenging to depict. Lastly, some signs are arbitrary and unrelated to the actual object;

- **Absence of signs:** When a Portuguese word does not have an equivalent sign in LGP, it is necessary to substitute it with a synonym. The word in question must be spelt out in LGP if no synonyms are found. It is important to remember that LGP users do not know every sign or word, just like speakers of other languages do. Notably, several medical terms have a sign associated with them, but most of the

deaf community is unaware of them.

2.2 Character representation and animation

2.2.1 Avatar

An avatar is a graphical or 3D model that, for the most times, serves as the personal virtual embodiment of a real-world user in a digital or virtual environment. These environments can include video games, virtual reality simulations, online social platforms, or other computer-generated worlds. Avatars are used to interact with virtual surroundings and with other users or entities within that environment.

To create an avatar, a humanoid mesh with the proper topology for animation had to be made, and it had to be linked to the skeleton to move, as shown in Figure 2.3. A mesh is a fundamental component representing three-dimensional objects or surfaces in a digital environment. It is a collection of vertices (points in 3D space), edges (lines connecting vertices), and faces (polygons formed by connecting edges) that define the shape and structure of an object in a 3D space. A skeleton, the component used by the algorithms that generate the character's poses and movements, should ideally have an Inverse Kinematics tree chain starting at the spine and ending at the hands. This means that when one bone moves, all the bones that are connected to it also move in unison. For instance, the elbow and shoulder bones also move when raising the hand. Additionally, each finger must have its own Inverse Kinematics chain to allow for precise positioning.

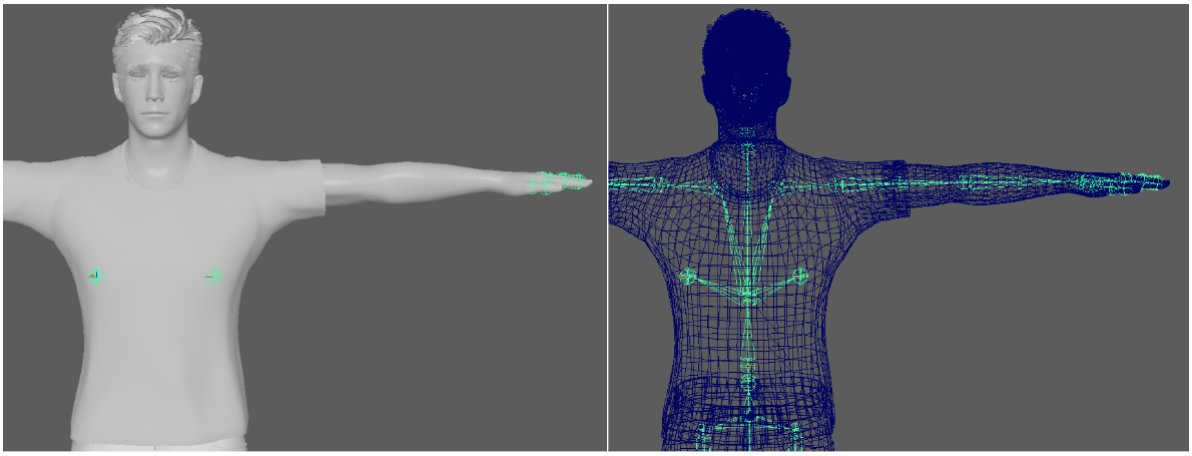

Figure 2.3: Example of a Rigged Avatar.

An essential part of animating glosses is facial expression. This cannot be controlled by bones like the rest of the body. A technique called Blend Shapes [7] is used for facial expressions. Blend shapes, also referred to as shape keys or morph targets, is a method used to produce natural and lifelike deformations and animations of 3D models, especially of humanoid avatars and objects. To make a blend shape work, a base 3D mesh has to represent the neutral or default state of an object or character, in this case, the face mesh. Then, a set of target shapes or poses you want the base mesh to morph into defines the blend shapes. Each target shape is a different mesh configuration in which vertices have been added, moved, or subtracted to produce a particular result, as shown in Figure 2.4. These target shapes can represent various facial expressions, speech animation phonemes, or any other deformations you desire. By interpolating(blending) between several predefined target shapes, blend shapes let you change the shape or expression of a base 3D mesh.

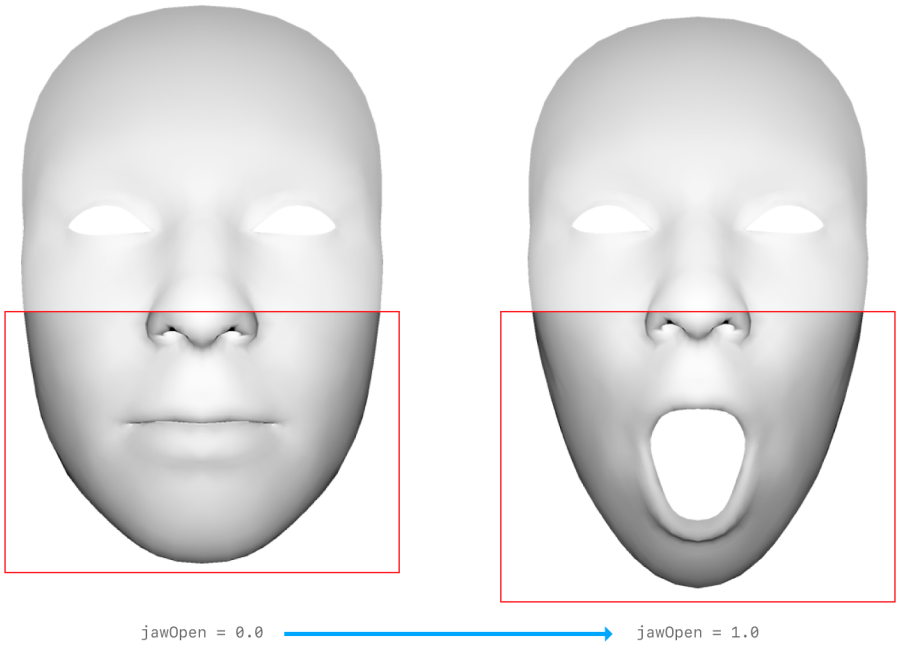

Figure 2.4: Example of a Blendshape.

2.2.2 Animation

An animation is, by default, composed of 24 frames per second [8]. This means 24 images are required to show a video for one second. To make a 3d animation, one has to

manipulate each desired skeleton bone and set a Keyframe.

Keyframes [9] are particular frames in a sequence where an animator modifies an object's property or parameter. These keyframes usually mark the beginning and conclusion of an animated movement or change, leaving the generation of the intermediate Keyframes to be calculated by the software the animator is using, be it a 3D modelling software or game development software. An animator could, for instance, start a humanoid avatar in a running pose in one frame, then move it a little to the left and alter its arm and leg positions in the following frame, as seen in Figure 2.5. The animator went through this procedure 24 more times to create an animation that lasted one second.

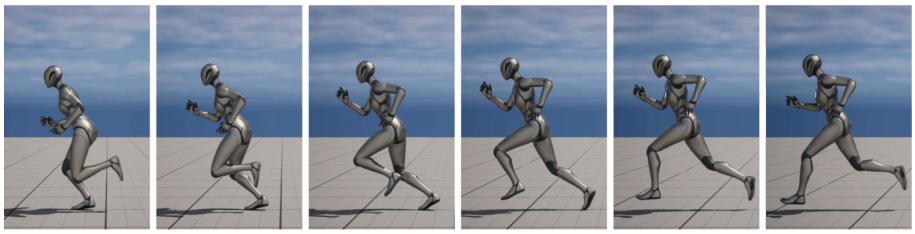

Figure 2.5: Example of a Keyframe Animation.

2.3 Modelling

This section briefly explains the software used in our project for modelling, animation, and 3D concepts. Blender [10] and Autodesk Maya [11] were chosen due to our familiarity with these free tools, in the case of Autodesk Maya, a free 3-year subscription for students, and their alignment with the project requirements. Although both these software can create a realistic humanoid character, developing it would take time. The software Character Creator 4 [12] and Iclone [13] were used to create an avatar to speed up the project's development. These software were chosen because they were made available for this project.

In terms of file format to store all the animations and avatars produced, there was the choice between the FBX(Filmbox) [14] format and the glTF(Graphics Library Transmis-

sion Format) [15] format. Both these formats can store 3D models and animations but have different purposes. FBX is a more comprehensive format suitable for complex scenes and animations. At the same time, glTF is designed for efficiency and simplicity, making it ideal for real-time web-based and VR/AR(Virtual Reality and Augmented Reality, respectively) applications. The FBX format was chosen for storing the animations because it is more comprehensive and suitable for complex animations of sign language gestures.

2.3.1 Blender

Blender is a free and open-source 3D computer graphics software application widely used in academic settings for teaching, learning, and research in various computer graphics, animation, modelling, game development, and visual arts fields. In Figure 2.6, we can see the process of animation.

Developed and maintained by the Blender Foundation, Blender has gained recognition for its versatility, robust feature set, and accessibility, making it a valuable tool for educators, students, and researchers. Furthermore, due to its open-source nature, Blender boasts a community filled with artists, developers, and enthusiasts who are very welcoming. Blender fosters a culture of collaboration and continuous improvement, making it a vibrant hub for creativity and innovation.

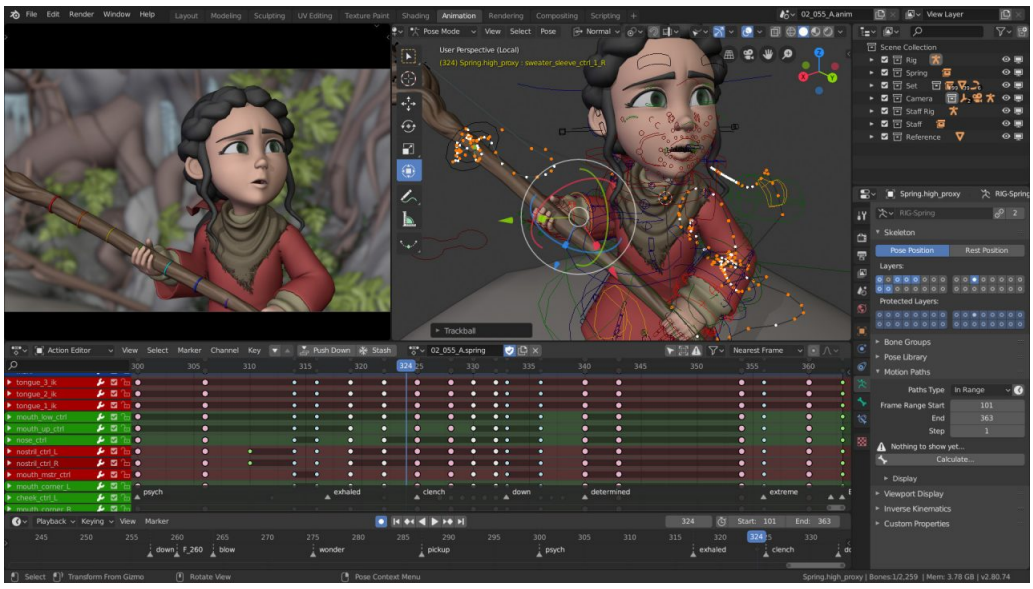

Figure 2.6: Blender 4.0 example scene.

2.3.2 Maya

Autodesk Maya, often called Maya, is a comprehensive and industry-standard 3D computer graphics software application with a prominent place in computer animation, visual effects, motion capture, and 3D modelling. Widely adopted by professionals in the entertainment, film, gaming, and design industries, Maya is also a valuable tool within academic settings, where it serves as a powerful and versatile platform for teaching, learning, and research.

2.3.3 Character Creator 4

Character Creator 4 is a commercial 3D character creation software developed by Reallusion. It provides users with powerful tools for designing and customising 3D characters. The software supports detailed customisation of character faces, bodies, clothing and accessories, allowing users to create unique and realistic characters for various applications. In addition to character customisation, Character Creator 4 provides a content library with various pre-made characters, clothing, accessories and animations. This library can be a valuable resource for users looking to speed up their character creation process. One of Character Creator 4's outstanding features is its compatibility with other 3D software and game engines. Users can export their characters to popular platforms and integrate them seamlessly into their projects.

2.3.4 Iclone

iClone is a commercial real-time 3D animation software developed by Reallusion. It offers real-time animation capabilities, allowing users to see instant results. The software includes tools for creating 3D characters, allowing users to customise faces, bodies, clothing and accessories. Motion capture technology is supported, allowing real-life movements to be applied to 3D characters. iClone's animation tools are aimed at both beginners and experienced animators, offering features such as keyframing, motion clips and much more. The software is helpful for virtual production, allowing filmmakers and content creators to visualise scenes and characters in real-time during production. iClone provides rendering capabilities to create high-quality images and videos, with customisation options

for lighting, shadows and visual elements.

2.4 Motion Capture

Motion capture suits, also called mocap suits or motion tracking suits, are specialized wearable suits that capture and digitize a person's movements and actions in real-time. Numerous industries use these suits extensively, including video games, sports analysis, biomechanics research, and virtual reality(VR) applications. Motion capture suits are essential for recording physical movements and converting them into digital data for various artistic and scientific uses. In this case, to record realistic animations of Portuguese Sign Language (LGP) signs. A cheaper alternative to using these suits is to use a Microsoft Kinnect [16], which can record a person's movement. However, stand alone, it can't capture hand gestures, which is necessary for sign language gestures.

For this work, two Motion Capture systems were used to capture the signs performed by an LGP speaker, shown in Figure 2.7: the Rokoko Studio Pro from Rokoko Studio [17] and the Perception Neuron Studio from Neuron Mocap [18].

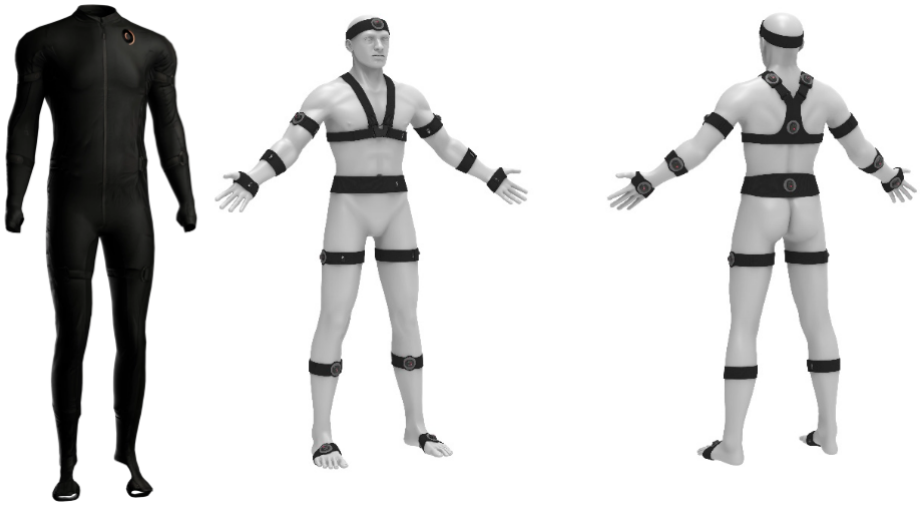

Figure 2.7: Rokoko Studio Pro on hand on the left / Perception Neuron Studio is on the right.

2.4.1 Rokoko Studio Pro

Rokoko Studio Pro uses a suit with sensors positioned strategically on the body to transmit data on the user's position and rotation to a 3D avatar in the corresponding software. With the help of an Apple iPhone, this system can also perform facial tracking, photographing the user's face and converting the image into one of ARkit's 52 standard facial blend shapes, which are in a way, a skeleton for the face. By deforming the mesh in various ways, blend shapes give the impression that one shape naturally transforms into another. This works by creating a copy of the object, which is then reconfigured into a different shape. For example, you can animate a character's face using blend shapes, changing it from neutral to smiling.

This system has a "plug and play" design that makes it very easy to use, but it can be somewhat impractical because the suit comes in several sizes. If the person wearing it doesn't have the correct size, the suit creates several inconsistencies in the data it captures.

2.4.2 Perception Neuron Studio

The Perception Neuron Studio system trades off simplicity and facial tracking for adaptability. By using straps to secure the sensors to the vital areas of the body instead of a full-body tracksuit, it has a one-size-fits-all "suit," but these sensors need some setup and calibrations to function correctly. Even though the sensors in this system are highly susceptible to magnetic interference from nearby devices like computers, phones, and routers, the Perception Neuron Studio can produce incredibly lifelike animation data in a concise amount of time, enabling the recording of motion data for hundreds of signs in a single day with only a few minor skeleton adjustments needed due to the sensors slowly moving out of place due to the straps.

2.5 Game Engines

This section briefly explains the game engines used in our project. Although there are many free game engines, for example, Unreal Engine [19], known for its incredible visual graphics, and Godot [20], ultimately, we selected Unity [21] since we are experienced with these technologies, and they meet the project requirements. This work also developed a

system utilizing UPBGE [22], a game engine based on Blender, to compare it to the work produced in Unity.

2.5.1 Unity

The Unity development engine, often called Unity, represents a versatile and highly regarded multi-platform game engine widely used in academia and the professional game and application development industry. It provides a comprehensive and integrated framework for creating, designing, and implementing interactive digital content on various platforms, which can be seen in Figure 2.8, including, but not limited to, desktop computers, mobile devices, games consoles, augmented reality (AR) and virtual reality (VR) platforms.

Unity is known for its user-friendly interface and vast collection of tools that allow programmers, researchers, and educators to quickly create interactive 2D and 3D applications, simulations, and games. It has a strong graphics rendering engine, physics simulation, and a critical resource pipeline for managing multimedia resources such as textures, 3D models, audio files, and animations.

In addition, Unity is a flexible option for implementing customized features and integrating third-party technologies because it supports C# scripts and has many plugins and extensions that extend its capabilities.

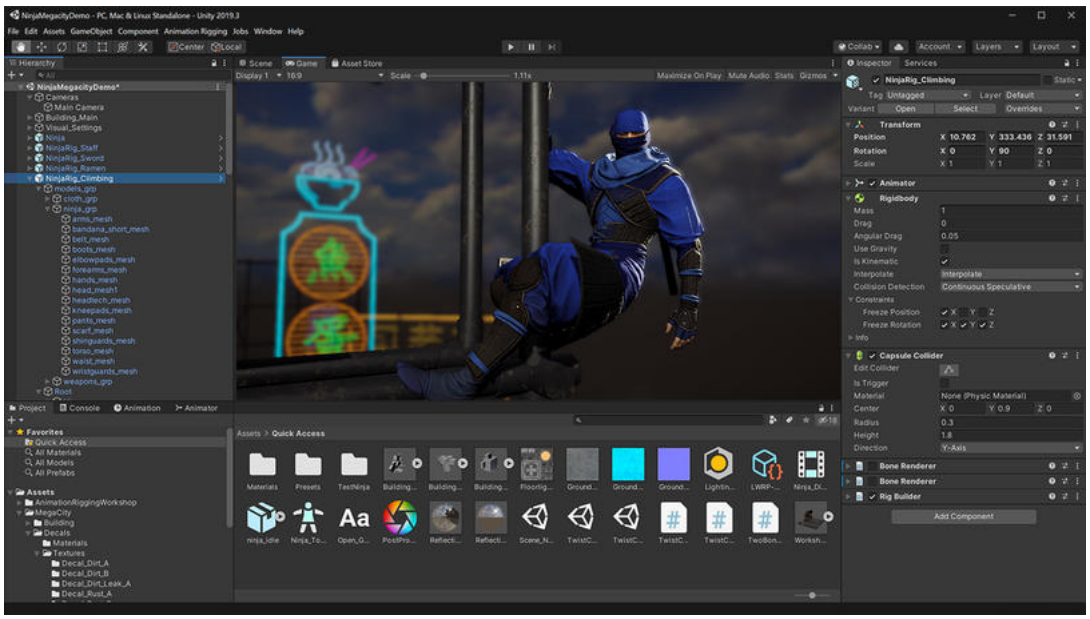

Figure 2.8: Unity 3D example Scene.

2.5.2 UPBGE

"Uchronia Project Blender Game Engine", or UPBGE for short, is a fork of the Blender Game Engine (BGE) [23]. The goal of the Uchronia Project is to advance and carry out Blender's game engine development, which became independent from the Blender Foundation when the original Blender Game Engine was removed from Blender version 2.80 and is run primarily by a community of developers. The development and maintenance of UPBGE involved contributions from volunteers interested in extending the capabilities of the Blender Game Engine.

2.6 Conclusion

This chapter offers a thorough theoretical framework to improve the reader's understanding of LGP by clarifying its unique characteristics and outlining essential differences from other language forms. Investigating the theoretical foundations promotes a deep comprehension of LGP and creates a background for the ensuing real-world implementations in technology and animation. The chapter then explores the technological aspects crucial to the future project, explaining the importance of foundational concepts in 3D animation, including the intricacies of keyframes and the manipulation of 3D models, the utilization of motion capture suits, and software programs such as Blender, Maya, and Unity. These platforms, renowned for their 3D modelling and animation skills, will be essential in converting academic knowledge into valuable applications.

Chapter 3

Related Work

In this chapter, a systematic literature review is carried out to find research on sign language animation, its various forms, and the user's presentation of these animated signs. After explaining and analyzing each of the studies chosen, a comparison is made to determine the approaches taken and the results obtained.

3.1 Systematic Review

Since there are so many sign languages, it is not easy to create a universal system, and, as a result, most of the solutions implemented to date focus on just one sign language. When translating spoken language into sign language, you need a system that allows a deaf person to see and understand a sequence of glosses without reading the pre-translated sentence. Before that, it is necessary to translate a written sentence (in a spoken language) into glosses since the grammar and structure of spoken and sign languages are different. As translation requires the use of a "person" capable of executing a translated movement, the use of 3D avatars is the most popular solution. Most systems for translating spoken language into sign language, including the animation and transition of each sign, use 3D manipulation tools such as Blender and Autodesk Maya and 3D application generation platforms such as Unity.

Utilizing the Preferred Reporting Items for Systematic Reviews and Meta-analyses (PRISMA) methodology [24], a literature analysis was conducted on the existing research that centres on the animation of sign language. To find research publications,

two databases were searched: the Institute of Electrical and Electronics Engineers (IEEE) and Scopus. The search terms "sign language," "3D," and "Animation" were combined and used in searches of both of the databases. The inclusion criteria included the previously listed keyword combinations and being published after 2000, as most technologies developed before that year are either too old, deprecated, or unsupported. After 95 articles from Scopus and 121 from IEEE were located, 26 were duplicates, resulting in a total of 190 needed to be screened.

These papers were filtered based on the title, keywords, and abstract. The exclusion criteria included literature reviews, incorrect language (neither English nor Portuguese), incorrect setting (focusing solely on the translation of sign language to gloss and ignoring gloss display), and irrelevant content.

Finally, 59 articles underwent full-text screening and were determined eligible or rejected based on specific criteria. These included being unable to access the entire text, the evaluated result not being pertinent to our study, the lack of original data, and minimal to nonexistent information on the animation component. This led to the screening of 16 publications, some of which were associated with the same project. Consequently, 14 distinct investigations were identified from the collection of selected articles. The entire process is shown in the Figure 3.1.

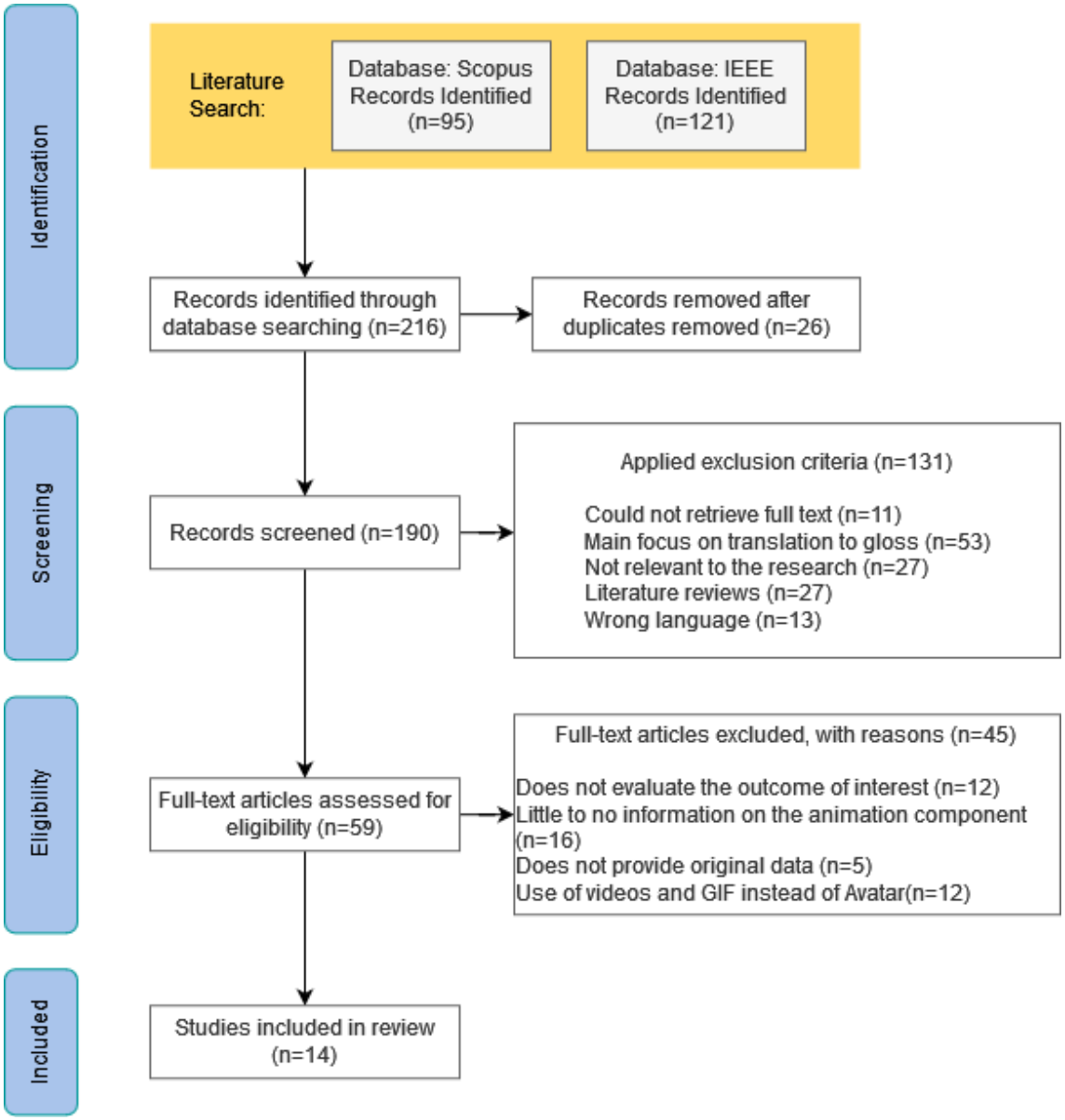

Figure 3.1: Search Process for Study Selection.

3.2 Related Studies

Santos [25] [26] proposes a system for translating the Portuguese language into LGP using Microsoft Kinect named PE2LGP (From European Portuguese to Portuguese Sign Language). The proposed system initially used Blender to create the Avatar to animate the glosses and obtain the corresponding animations. The whole system was transferred to Unity since it is a platform for developing games, has more tools for development, and, more importantly, implements a state machine. A state machine is a behaviour model that can define, as the name implies, the state of an Avatar by defining a start or stop position, an "idle," and a state for each animation, with conditions to be active. It, therefore, allows the definition of how many animations will be animated by the avatar and makes it possible to associate a transition between them through the various states. This software change was made because Blender, despite having the ability to do transitions between different animations, doesn´t have a feature to create a program that runs on any platform. However, it can produce an animation as a sequence of animations associated with each term of the LGP. In Figure 3.2, we can see the Unity application that allows a user to create an LGP animation by modifying the hand configuration and utilizing key points in the arms that will be recorded using the Microsoft Kinect.

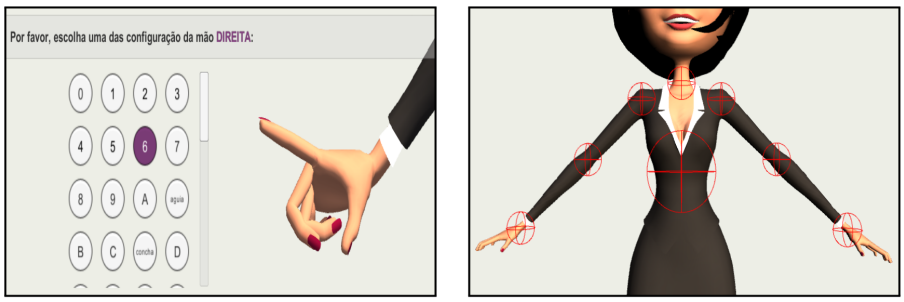

Figure 3.2: PE2LGP Hand selection on the left and Movement points on the right.

Escudeiro [27] presents a system similar to the previous one called Virtual Sign. However, instead of using Unity's state machine to handle the animations, they used Blender's capabilities to create a single 3D model, with all the gesture animations and the respective transitions between them, within a timeline by utilizing Blender's NLA track. Then, Unity runs the animations at the indicated track in the order the translation requires. This system was compiled and exported as a WebGL [28] application implemented in several

Portuguese websites to provide a live translation of a selected text, as shown in Figure 3.3.

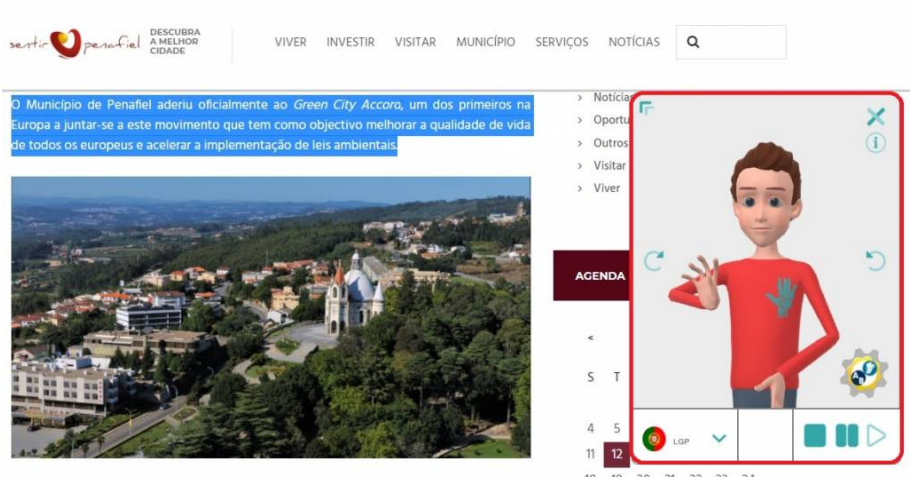

Figure 3.3: Example of virtual sign WebGL application on a website.

When comparing the two systems, the former takes an average of 11.8 minutes per gloss, while the latter, more automated, takes an average of 2.4 minutes. However, when testing, both systems proved not to be very effective since most evaluators did not recognize the gloss or answered incorrectly to the visualization of the animations. These tests demonstrate that improvements are needed in creating LGP signs before they are used in a final product by the deaf community, such as using better systems and taking more caution and time for each animation.

Real, Santamaria, and Olmedo [29] present an online sign language representation system of Ecuadorian Sign Language(ESL) that seeks to extract the text from the "Closed Captions," that is, the subtitles automatically generated in specific contents on the Internet, namely videos, using algorithms to concatenate, process and animate the subtitles, with a 3D Avatar. The transitions of this Avatar's various animations are frail because the transition between all its animations requires the Avatar to return to this neutral position with the arms resting by the avatar's side. This makes it so that there is a pause between each animated gesture. This pause also forces the gesture animations to have a higher speed to keep the translation at the rhythm of the subtitles. Because of all those weak points, it is difficult for deaf people to interpret what the avatar is gesturing. Regarding alternatives to the tools mentioned above, some systems are created for users who want

to develop animations without the knowledge of 3D.

An example is in the article by Heloir, Alexis, Michael, and Kipp [30] in which they present an "Engine" for researchers and developers who do not have extensive knowledge of tools and applications for manipulating 3D content called EMBR (Embodied Agents Behavior Realizer) which has its control programming language called EMBRScript. With this application, it is possible to produce multi-modal animations through a script-like language programmatically. The user can define how he wants the animation to run from start to finish, which animations run simultaneously, merge animations of several avatars, and create transitions between animations using EMBRScript code.

This system is used in Kipp, Heloir, Nguyen [31] as the base algorithm of the German-to-German Sign Language translation system (Deutsche Geb¨ardensprache, DGS). This paper proposes a cyclic gloss animation system created using EMBRScript and a database of individual gloss animations. Depending on the text entered, the system communicates with a database and produces a video with the sequence of glosses together and animated by an Avatar. Then, they compare with videos of how the sentence should be gestured to compare results. These tests concluded that the animation system is a good starting point for animating glosses, although its limitations prevent it from being the norm. One limitation is that the context of what is being translated dramatically influences how the gesture is performed and can cause the translation to be misinterpreted. Another is that the EMBR system cannot process facial animation, causing the Avatar to have no expression, keeping it neutral. Having a neutral face can impact what the translation means because deaf people need to read a person's expression to understand the context and or meaning of what is being said in the conversation.

This same facial animation problem was found in the article by Zijl Barker [32] presenting a solution for realistic facial animation of an avatar in South African Sign Language(SASL) using a NURBS mesh. A mesh based on NURBS (non-uniform rational B-splines) consists of a mathematical representation of several points connected by curves instead of having a mesh based on polygons. This solution allows manipulating the various points in a facial mesh more efficiently for a system to process. It has a beneficial side effect in which the face produces wrinkles while expressing itself, giving the end user a more realistic and perceptible touch, as seen in Figure 3.4. They utilized 3D Studio Max[33] to

produce the facial animation, resulting in a VRML[34] file, which is now a very outdated format. Still, Blender and Unity are prepared to deal with NURBS meshes. Having a face with realistic expressions is necessary for gesture animation, as the meaning of signs can change depending on the person's facial expression. An example is given in this article, where the gestures of "weight" and "maybe" are differentiated only by the person's facial expression.

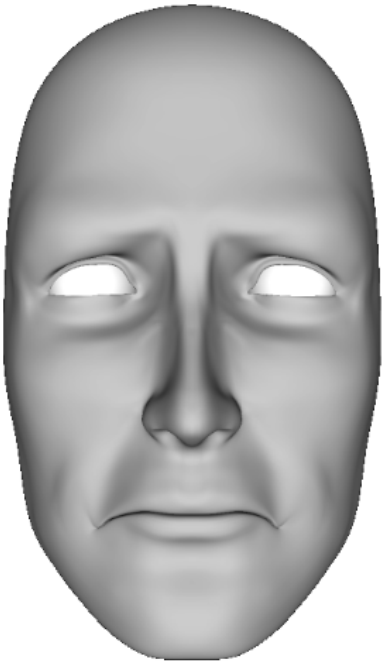

Figure 3: 'sad'

Figure 3.4: NURBS face representing a sad emotional state.

The article by Ying He, Alifu Kuerban, QingLing Yu, and QiNa Xie [35] discusses the current challenges in TV news development, mainly focusing on issues faced by deaf and hard-of-hearing Chinese Sign Language speakers. It emphasizes that the small size of news hosts on the screen makes it difficult for viewers to discern their gestures. The diversity of sign languages, including natural and grammatical variants, poses comprehension challenges. To address these issues, the article proposes using 3D animation software with virtual human models created using Autodesk's Maya software and increasing their screen proportion. To animate the virtual humans, they used keyframe technology [36] to

enhance animation efficiency, requiring only two keyframes for mathematical interpolation to generate intermediate frames. With this, they could create a comprehensive library of sign language animations. Subtitle acquisition involves processing video frames through binarization and statistical analysis of horizontal and vertical edge intensity matrices to extract subtitle coordinates. Using WPF (Windows Presentation Foundation) [37] based on the C# language to process the subtitles and visualize the interface and invoking Python script in C# to extract the subtitles, the sign language animation is reproduced in a Unity instance, and communication between the subtitle processing module and the sign language animation module is achieved through both, as can be seen in the Figure 3.5.

Figure 3.5: Interface diagram of video sign language translation system.

An article by Punchimudiyanse and Meegame [38] presents a comprehensive 3D avatar and animation system designed for Sinhala Sign Language (SSL), capable of defining and animating sign gestures without the need for video sequencing or motion capture hardware. The animation system presented focuses on the upper body of the avatar, featuring 29 bones per arm. It uses the Blender game engine, utilizing the "Always Sensor" in BGE with a Python controller to develop animation scripts executed every 1/60 seconds. The signing avatar's bone structure is animated independently to represent sign gestures. To animate a gesture, this system utilizes text files to store state, coordinates, and animation playlists persistently of every bone and two techniques simultaneously to animate. The

first involves the animation algorithm to determine the movement of each bone toward its final position using either a specified number of frames and the second is an incremental value for each bone that gives an animator control over the speed of the animations. This technique provides total control to the animator to move specific bones at lower speeds than some other bones. Additionally, inverse kinematic features are turned off here to prevent unintended bone movement, and the algorithm provided in the text is used to change the properties/coordinates of each bone. In Figure 3.6, the Avatar moves from one pose to another using only scripting.

Figure 3.6: Animating two static posture words in a sentence in BGE.

The article by Jing Wang, Yanfeng Sun, and Lichun Wang [39] discusses the importance of sign animation effectiveness and reliability, particularly when employing a 3D model alongside extensive sign language data. To address potential challenges associated with prolonged rendering times for each frame, the authors propose a sequence rendering method utilizing OpenGL ES(OpenGL Embedded System) [40]. This approach leverages a double buffers-based fast rendering algorithm designed for mobile devices, as shown in Figure 3.7. In the sequence rendering method, the current frame is displayed on the screen within the foreground buffer. In contrast, simultaneously, the next frame is rendered in the background graphics buffer, which is invisible to the user. When it's time to display the next frame, the foreground and background buffers are swapped, ensuring continuous sign animation. OpenGL ES simplifies the definition of a virtual human geometric model.

Instead of using a set of glVertex, the authors use the VertexArray and glDrawArrays functions to define a polygon. The process of rendering a set of triangles involves declaring an array for vertex coordinates, establishing a pointer with glVertexPointer and using glDrawArrays to generate the triangles simultaneously.

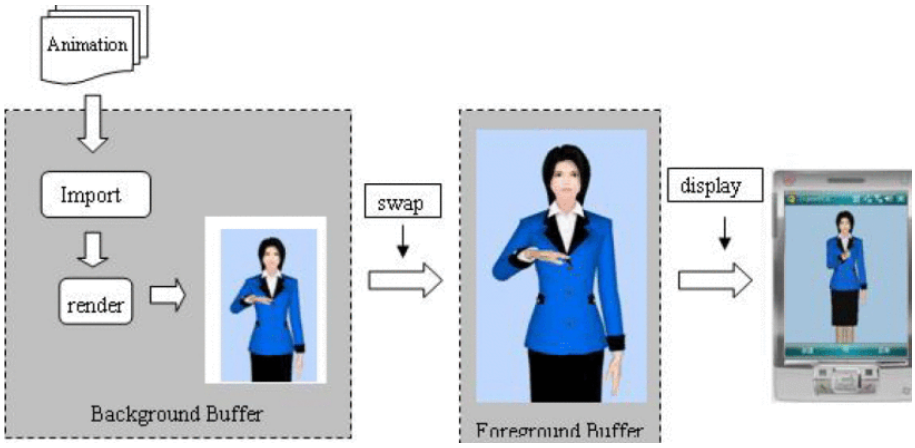

Figure 3.7: Diagram of the Double buffers-based fast rendering.

In the article [41], the authors aim to address a gap in the adoption of solutions and techniques for sign language communications of several European countries to the Arab world by contributing to the transfer of assistive technology to deaf people. They propose the development of a prototype of a virtual translator capable of converting Arabic text into an animated sequence of Arabic sign symbols in real time. They achieved this by testing several different tools, which include Vcommunicator [42], and in the end, decided that eSign [43] is the best suited to do a translator system for Arabic Sign Language. The two main reasons this system, shown in Figure 3.8, was chosen were the eSign editor making it simple to create several animations, either via clicking a few graphical buttons or by directly entering a HamNoSys [44] notation string, which can also be easily exported, and its capability to manipulate facial expressions, a crucial factor in enhancing the realism of sign language. Editing facial expressions involves modifying elements such as eye gaze direction, eyebrow movement, nose, mouth, and cheeks.

Figure 3.8: Interface for translating and signing Arabic text.

The article by Paramat Ittisarn and Nunnapad Toadithep [45] aims to create a system for editing and displaying Thai Sign Language (TSL), facilitating communication between the general population and the Deaf community by utilizing XNA (Xbox/DirectX New generation Architecture) [46], which is a tool for developing games in Xbox and Window operating system developed by Microsoft, that can be seen in Figure 3.9. To animate a translation, they use XML (eXtensible Markup Language) [47] to save arm, hand, and finger motion and connect it with Thai sign language words, meaning that by just scanning the inserted text, each word will, in theory, have an associated XML file with the animation data for the gesture that will be played for the user.

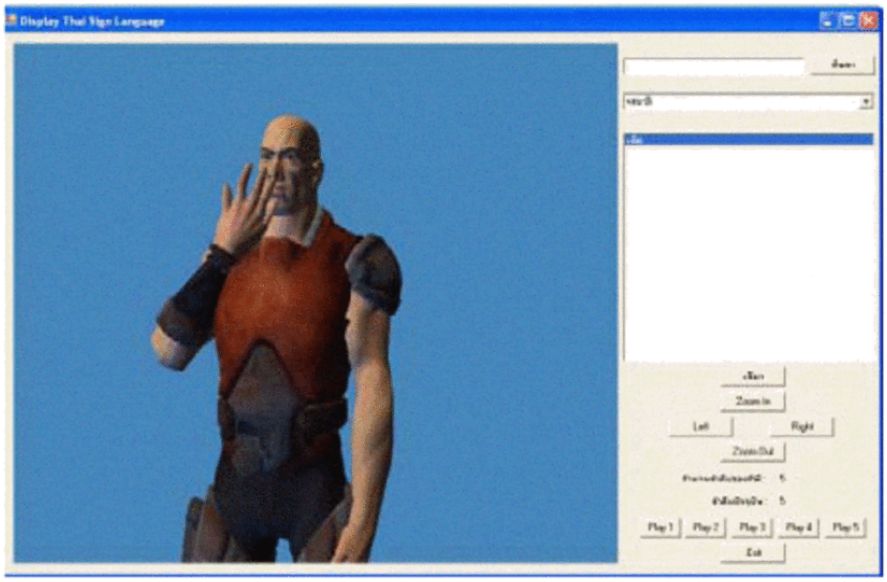

Figure 3.9: System displaying Thai Sign Language word for “spicy”.

The authors in [48] recognize that the Indian deaf and mute community predominantly uses Indian Sign Language (ISL) for communication, employing hand gestures, facial expressions, and body postures and that most people do not have a way to interact with them. To solve this, they propose a system that can be implemented in an Android [49] phone that utilizes a 3D Avatar to translate spoken language to Sign language, which Figure 3.10 illustrates. To create this system, they utilized Blender to make the 3D Avatar and used a Microsoft Kinect to record the animation data for the gestures. Both the avatar and recorder gestures were then integrated into a Unity application where they were organized in the animator hierarchy with specified parameters and lastly, an integration of Google Speech-to-Text service [50], usually available native in Android, to convert spoken input into parseable strings. When Unity receives a text string, it is passed as input to an ISL parser. The ISL parser separates all the words in order as prescribed by the ISL rules. If any word isn’t present in the library, that word is broken down into individual letters, and each letter is processed independently. The Unity package was converted to an APK file for Android devices, facilitating widespread distribution on the Google Play store.

Figure 3.10: Final User interface.

The article [51] introduces an extension to traditional sign language production using an animated avatar based on a vocabulary of animation clips and a novel multi-modal and multichannel sign description representation (MMS). The MMS augments glosses with parameters describing alterations of vocabulary signs on a geometrical level, independently from language, compensating for limitations in previous approaches. Current sign language representation, like the previously mentioned EMBRScript, lacks inflexion, and the research in this article proposes a data-driven approach to transform motion capture data associated with glosses by automatically computed inflexion parameters derived from text conversion. The authors aim to address the challenge of determining the list of inflexion parameters related to a gloss, consider existing manual annotation schemes, and propose a solution to enhance the process. With Blender's Python scripting flexibility and an already rigged avatar, they could produce the animation for the word "Nicht"(the word "no" in german), as illustrated in the figure 3.11. Their final goal is to create an exten-

sion for Blender to help users who use an annotation scheme to adjust the inflexion values visually.

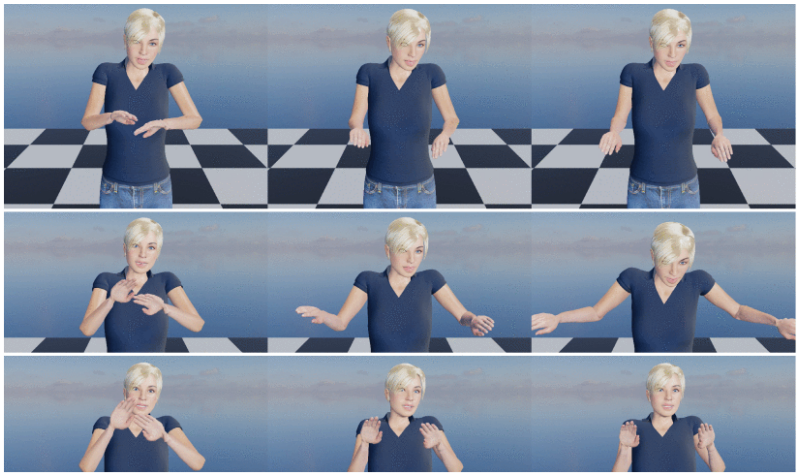

Figure 3.11: Sign NICHT animated.

In [52], the authors describe that in Colombia, there is a recognized need for equal educational opportunities for the deaf community, given their low educational levels, challenges in accessing social and employment opportunities, and lack of such tools hinder interaction and inclusion. The authors propose a mobile application focused on translating written Spanish into Colombian Sign Language called VLSCApp to address this. While just a prototype, as shown in Figure 3.12, this application aims to conduct tests and experiments with deaf and hearing users to try and develop a user interface that both parties can easily understand. The article discusses two options for designing and incorporating 3D content: DAZ3D [53] and POSER [54]. The POSER platform was chosen because it allows for more efficient and resource-friendly animations, contributing to an intuitive and easy-to-use program. In this early application stage, the 3d Avatar can only use the Colombian Sign Language alphabet with plans to create an expansive library of animations.

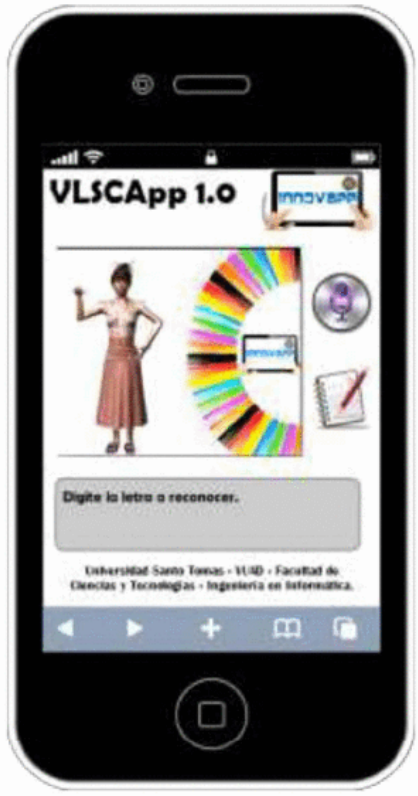

Figure 3.12: VLSCApp System Interface.

The article by Yosra Bouzid and Mohamed Jemni [55] discusses the challenges deaf learners face in acquiring literacy skills and the importance of sign language as a bridge for their language development. Currently, there is a lack of practical tools to represent SL in written form, and SignWriting [56] is one notable attempt. However, it requires training and has limitations in fully capturing the three-dimensional nature of signs. The article introduces a new tool called tuniSigner, designed to interpret SignWriting transcriptions using a 3D virtual signer. The system takes SignWriting notation in XML format. It generates corresponding avatar animations utilizing Websign [57], a Web application that transforms user text input into a real-time rendered animation and stores it in a dictionary after being assessed by an expert who supervises the system. This shows that the tuniSigner can interpret any other sign language and is not restricted to the local sign language, as seen in Figure 3.13, where the system interprets the gesture "Me" in American Sign Language.

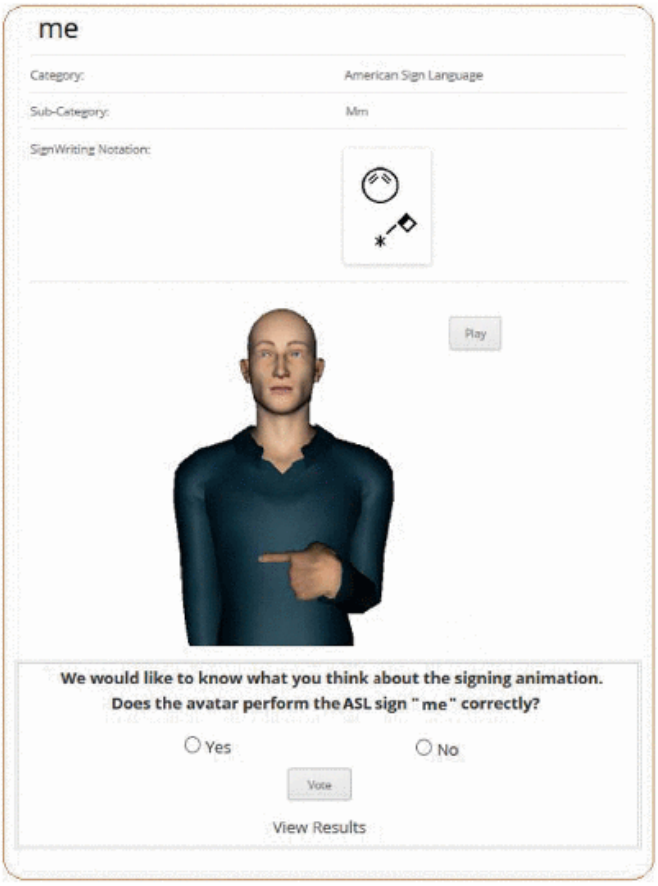

Figure 3.13: The signing animation is equivalent to the SignWriting notation of the ASL sign “me.”

3.3 Discussion

After analysing the studies, we can determine that technological advancements have significantly impacted sign language over the years, with a notable emphasis on using game engines like Unity and Blender for animating sign language. These platforms offer tools for creating realistic and interactive sign language applications, enhancing deaf users' quality of life and learning experience for non-deaf people. The literature review reveals a dual approach in this domain: the adoption of established game engines for their advanced animation capabilities and the development of specialized ”in-house” software to address the specific needs of sign language animation, such as nuanced gestures and expressions and the lack of accessible technology. Most studies here created animations for the body by hand, while some utilized additional tools, like Microsoft Kinect, focused on their local sign language with no clear way to expand their application to other sign languages and

are being developed for a desktop platform. While creating these technologies for Desktops might seem logical due to their processing capabilities, most deaf people might not have easy access to them daily. However, in recent times, it is far more common for everyone to own a smartphone with suitable hardware capable of rendering and animating realistic avatars.

System	**SLs**	**Platform**	**Animation Aquisition**	**Animation Engine**
PE2LGP [25] [26]	LGP	Desktop	Kinnect + Blender	Unity
Virtual sign [27]	LGP	Desktop/Web	Blender	Unity
[29]	ESL	Desktop	Blender	BGE
EMBR[30] [31]	DGS	Desktop	EMBR	EMBR
SASL-MT [32]	SASL	Desktop	3D Studio Max	3D Studio Max
[35]	CSL	Desktop/Broadcast	Maya	Unity
[38]	SSL	Desktop	Blender	BGE
[39]	CSL	Mobile	OpenGL	OpenGL
[41]	ASL	Desktop	eSign	eSign
[45]	TSL	Desktop	XNA	XNA
[48]	ISL	Mobile/Web	Kinnect+Blender	Unity
[51]	DGL	Desktop	MMS	BGE
VLSCApp [52]	CSL	Mobile	POSER	POSER
tuniSigner [55]	SL	Desktop	Websign	tuniSigner

Table 3.1: Table 1 Results of the Key Concepts

3.4 Conclusion

In this chapter, we studied works developed to animate several Sign languages. Initially, we studied the animation system developed in LGP, which was created utilizing Blender Game Engine (BGE) and later moved to Unity. Afterwards, we explored several other solutions developed for different sign languages, most involving using the same game engines previously mentioned, including XNA. Several works tried to develop technologies to animate sign language in several forms, all of which had some success. However, a mixed reception by their target deaf community caused the work to be still in development. However, the constant improvement of technology, the update of game engines, and support for several platforms still make for a solid base to develop an application that animates Sign language.

Chapter 4

System Architecture and Development

This chapter presents the system's general architecture with the various components that make it up. Initially, an overview of the architecture of the complete system will be given, followed by the architecture of the animation module of the Portuguese Sign Language glosses used in the entire system with Unity and a new architecture of the animation module developed in Blender. After that, the implementations, the numerous components, and the thought processes for both animation transition systems created during this work will be presented. It will begin by showing the computational characteristics of the machine used to create this work, both in terms of the hardware of the machine and all of the Software used for development. Then, a description of how the animations used in this system were collected using several Mocap suits and how they were processed and "cleaned "using Autodesk Maya software to be used by the final avatar of the application, followed by the creation of said avatar. Afterwards, the suggested Unity transition components will be implemented in detail, outlining the strategies employed and factors considered during development. This will be followed by describing the Blender transition component's development, difficulties encountered, and decisions made. The communication between the final Unity instance in the user's application and the Translation API is not part of the scope of this work, being only mentioned when necessary.

4.1 Framework

The work presented here was developed under the Research and Development project IVLinG: "Virtual Sign Language Interpreter" [1], whose aim is to design and create a platform to help communicate between deaf and non-deaf via smartphones, specifically in medically oriented scenarios. This project was developed by First Solutions, IPVC and CCG/ZGDV Institute. Therefore, the project is structured in different sub-components to provide translations between Portuguese and Portuguese Sign Language Glosses and vice-versa using an app on smartphone devices or web browsers, as shown in figure 4.1. The inverse translation, handled by CCG/ZGDV Institute, focuses on capturing the representation of gestures through the smartphone camera. Using image recognition techniques, the represented terms are extracted through this capture, which will then be converted to the respective Portuguese text. The work here focuses on the transition and animation component of the Portuguese to Portuguese Sign Language system developed by IPVC for the project. Finally, First Solutions developed a mobile and web application that integrated both translation modules.

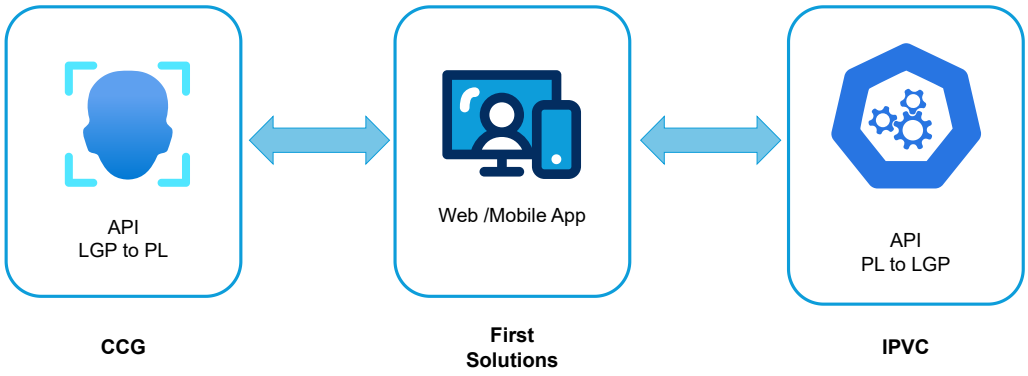

Figure 4.1: Overall Framework of the whole Project.

[1] $IVLinG : Virtual Sign Language Interpreter, Project website : https://tech.ipvc.pt/projeto.php?id_projeto = 184$

4.2 Architecture

4.2.1 Complete Portuguese to LGP System architecture

The overall system architecture of the Portuguese to LGP system, shown in Figure 4.2, encompasses several main components: the translation API, as illustrated in [58] [59] [60], the animation database, and the animation module. The central idea behind this design is to build a system that receives text or audio from the user who is speaking. An extra step of converting audio to text using a third-party speech-to-text technology is necessary when working with audio. Once the utterance has been analysed into text, a translation API transforms it into LGP glosses. Subsequently, the viewer must visualise the animated avatar containing these specific glosses, ideally stored in a database. However, there are various ways of presenting the avatar to the user so that the final stage will dictate the system's behaviour and workflow.

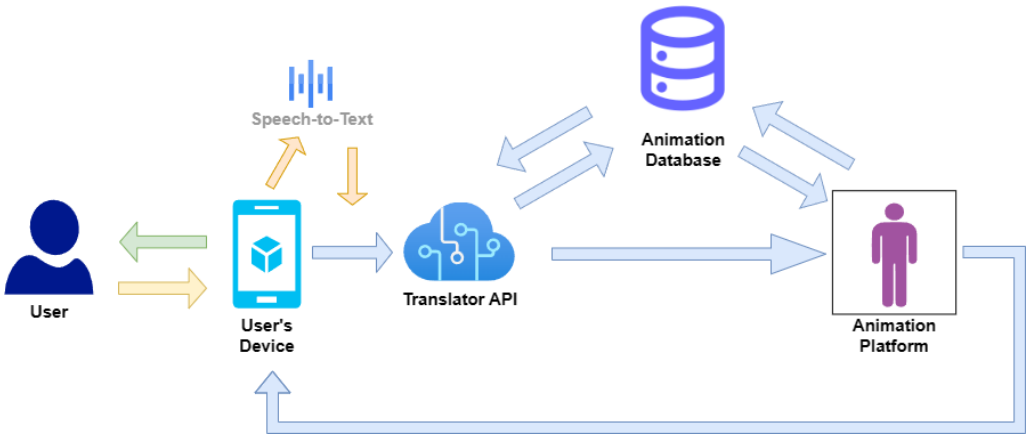

Figure 4.2: Overall Framework of the Architecture of the Portuguese to LGP system.

A live streaming option was first considered when the first architectures for the complete system were being discussed. The idea was to launch an avatar instance on a server whenever two users initiated an online video call using the IVLinG application. Users could participate in a "three-way" video call by watching this avatar instance broadcast live. The speaking user types or speaks to record the audio message they want to send to the translation API, which takes care of the translation and communication with the speech-to-text service in the case of audio files. The glossary translations would be sent to the server with the avatar, which would be animated and broadcast live to users. At this

point, it was decided that Unity would be the platform for configuring the avatar because it is a game engine that can handle many humanoid avatars to animate simultaneously and can be compiled in various formats, opening up a wide range of options.

Many problems were encountered when trying to test the viability of this option. Most of these problems were related to the connection between the application and the server hosting the avatar. Leaving aside the live-stream option, a proposed alternative was to have Unity directly in the application, using the user's device to process the animations needed for a translation, with only the translation API to transform text or audio into LGP glosses on a server. After some time developing the implementation of Unity directly in the application, this proved to be a more straightforward approach to the project's central idea. The final complete system, represented in Figure 4.3, is as follows.

Upon receiving a text and or audio file that is passed through a speech-to-text API, the Unity instance, located in the app, will send the message to a server that houses a translation API that will translate the message from text to glosses and send them, in the correct order, to an algorithm back in Unity. With the glosses received, Unity then calls a database, where all the animations needed for Portuguese Sign Language (LGP) are stored and downloads only the necessary ones for the translation. Once all animations are downloaded, Unity will animate the avatar with the collected animations. All animations will be stored as long as needed within the Unity instance, and when downloading new animations, the old ones will be deleted.

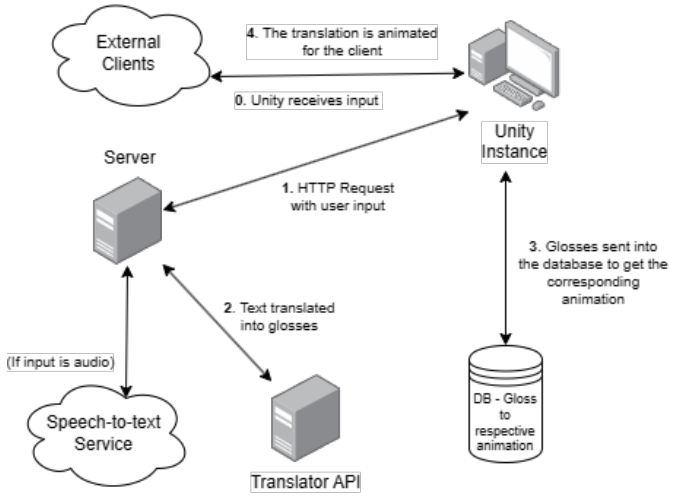

Figure 4.3: System Architecture for translation from Portuguese to Portuguese Sign Language.

4.2.2 Unity animation system architecture

This architecture, depicted in Figure 4.4, focuses on the animation and transition component within the overarching system of the Complete Portuguese to LGP System architecture. This component is vital in orchestrating seamless and lifelike transitions between each sign, or gloss, specific to LGP. Its primary objective is to enhance the visual appeal, clarity, and overall fluidity of the avatar's animations, thereby delivering a more engaging and understandable user experience.

Making the most of the user's pre-existing Unity instance on their smartphone or web browser is essential for this architecture. By taking advantage of the Unity instance's resources, the animation and transitions component can ensure that animation generation occurs locally, minimizing latency and improving the system's responsiveness.

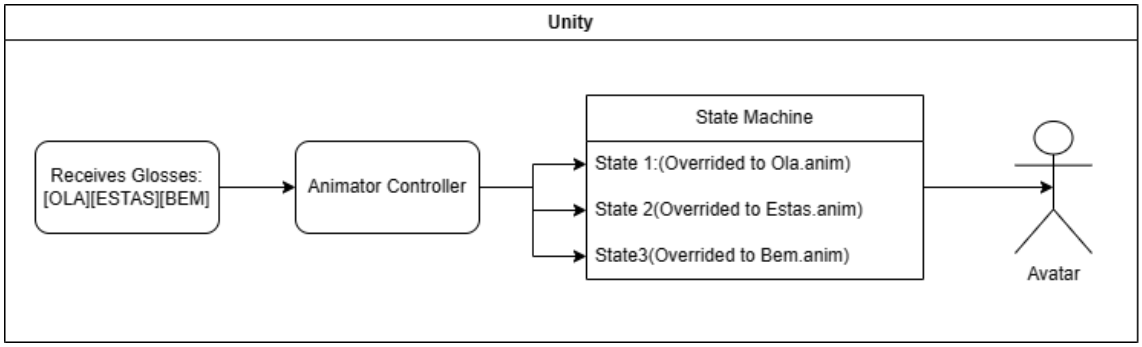

Figure 4.4: Unity System architecture for animating Portuguese Sign Language.

Improving the animations' aesthetic appeal, readability, and general fluidity is the main goal to provide a more exciting and understandable user experience. A key aspect employed for achieving smooth and natural transitions is the integration of Unity's State Machine. With this feature, developers can define different animation states for an object and seamlessly switch between them. Within this particular system, Unity starts animating a series of glosses after downloading all the required animations. After the download, the animations are arranged by an algorithm inside the State Machine. Then, a trigger for each occupied state is created and activated in the current order, which starts a series of animations that eventually result in the avatar "speaking" through LGP signs. This elaborate orchestration highlights the Unity State Machine's technical prowess and the intricacy and sophistication required to create an expressive and cohesive conversation

using animated LGP signs.

4.2.3 Blender animation system architecture

Similar to the previous system, when receiving a text from a user, the Unity instance will send the message to a server that houses a translation API if the receiving message is an audio file that has been easily converted to text first by using a speech-to-text API. The translation API efficiently transforms the message into glosses, arranging them in the correct order before forwarding them back to Unity. While in the previous system, Unity was responsible for collecting, ordering, and playing the animations, requiring an external database to house all of the animations, in this system, the storage, collection, ordering, and creation of the final animations are passed to Blender.

In this context, Blender creates a scene that includes all required animations and the skeleton of the avatar in the Unity instance that is already configured. When Blender receives the necessary glosses, as shown in Figure 4.5, an algorithm causes Blender to search the scene and find and arrange the appropriate glosses in a timeline. This complex procedure is made more accessible by utilizing a unique Blender feature called the NLA (Non-Linear Animation) editor, which can manipulate individual animations within a timeline for a particular object, or in this case, the avatar's skeleton. Blender completes the process by exporting the resulting animation of the glosses incorporated into the timeline. When the export is complete, Unity is notified that the animation file is available for download.

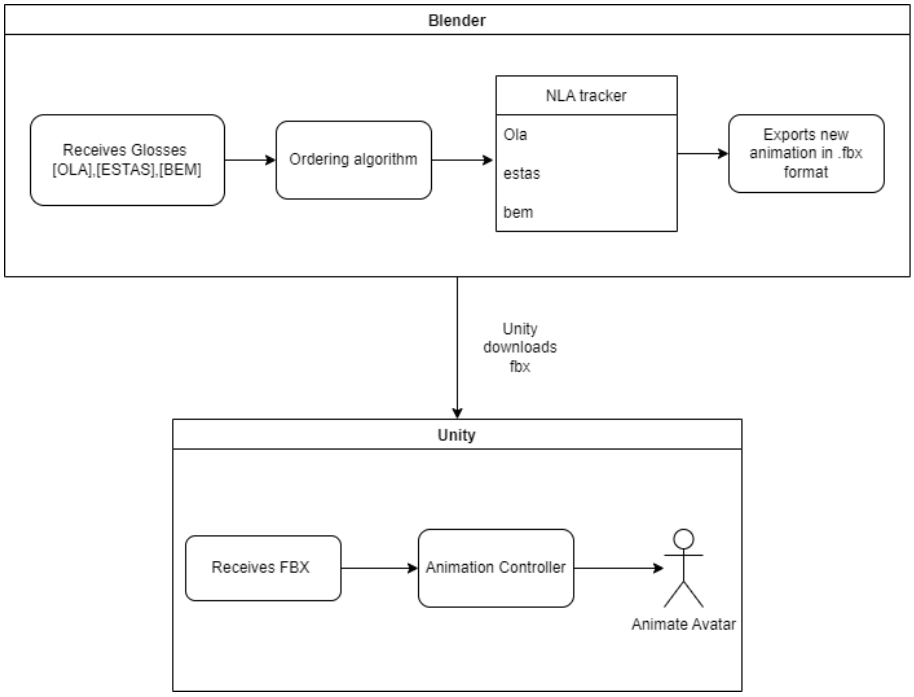

Figure 4.5: Blender Animation architecturePortuguese Sign Language.

By the time this work was done, there was no possible way to have a Blender instance running in a Web browser and smartphone, so the options for using it were limited. Still, two possible architectures, shown in Figure 4.6 and 4.7, can be implemented with Blender in the overall system.

The first is to have a dedicated server just running an instance of Blender that is open and will receive requests. After Unity receives the translated glosses, it will call the Blender server with those same glosses. Blender will accept the call and begin to order, in an empty NLA track of the skeleton, all the necessary animations requested by Unity and blend them into one single animation. Then, it will export that final animation as an FBX file and send it to Unity, which will play the complete animation to the user. While straightforward, this system would require multiple servers with different applications to achieve a final animation, so gaining several weak points in which the system could fail.

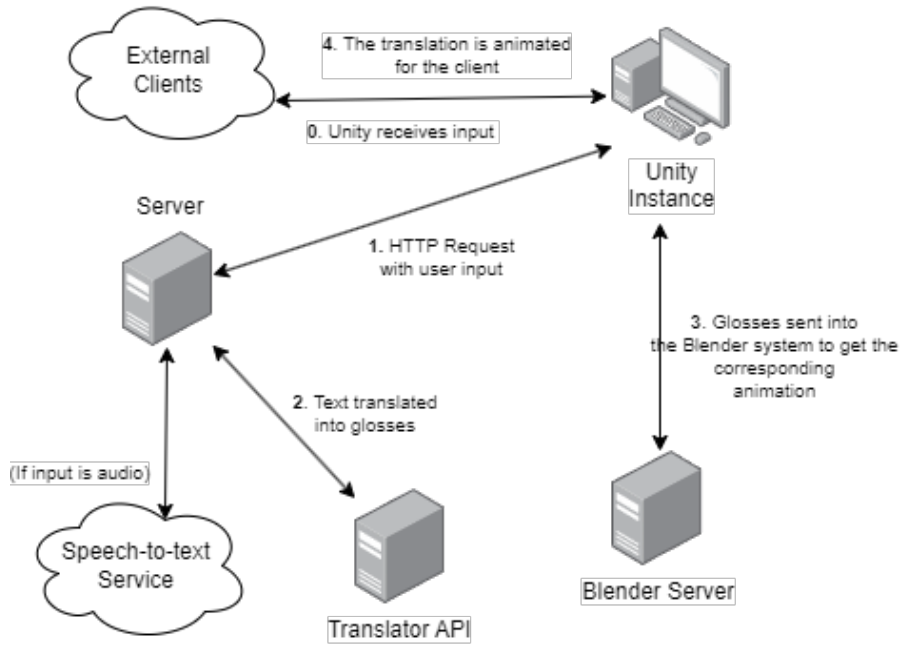

Figure 4.6: First possible Blender System for animating Portuguese Sign Language.

The second possible architecture is to have the Blender instance on the same server as the Translation API. With this, when the translation API receives the message for translation, instead of sending the translated glosses back to Unity, it feeds them to the algorithm in Blender that will take the gloss's names and produce the final animation of the translation. After Blender completes and creates the file with the final animation, the translation API will signal Unity, where the animation is for Unity to download and play to the final user.

For this work, the second architecture was chosen to develop the Blender transition system since it is easier for the translation API to communicate directly with Blender and vice versa than having a separate server in the overall architecture, with the Blender system and having it communicate with Unity separately.

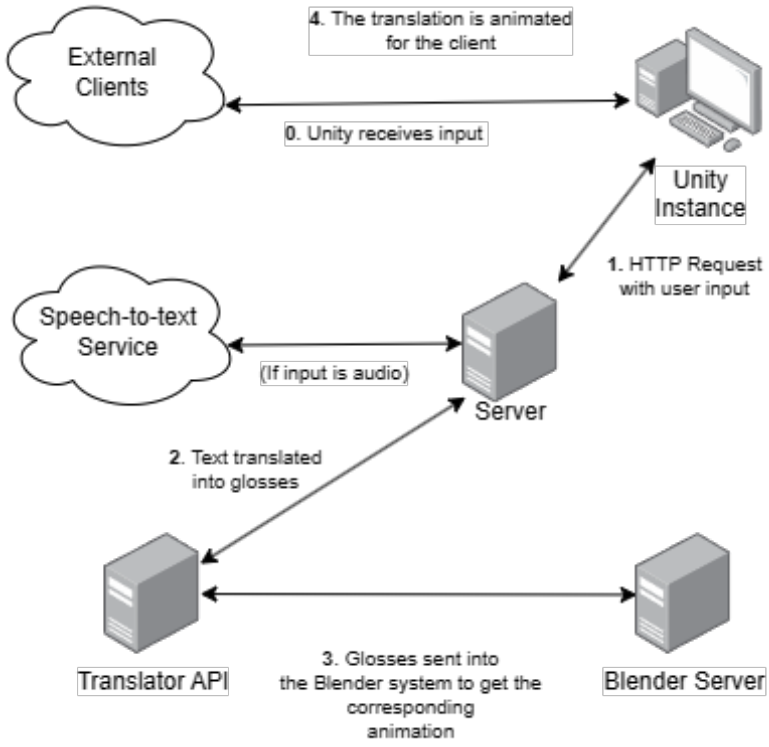

Figure 4.7: Second possible Blender System for animating Portuguese Sign Language.

4.3 Computational Characteristics

Every component developed in this work for the Unity system was created using Unity software version 2020.2.28f1 with the programming language C# 8. to systematically model the behaviour of game objects or characters and Python 3.10 programming language. In terms of hardware, the machine used to develop this work has an AMD Ryzen 7 5800H processor with Radeon Graphics, NVIDIA GeForce graphics card RTX 3060 12GB, 16 Gbs DDR4 3200 MHz, and 500Gbs SSD storage. All the implementations presented here follow the architectures shown in the previous chapter.

4.4 Tools and Technologies used

Unity State Machine

A state machine is a programming design pattern used in Unity to systematically model the behaviour of game objects or characters. Figure 4.8 shows a visual example of a simple state machine. It is a crucial idea in game development and is particularly

helpful for managing intricate behaviours and animations. A state machine features are as follows:

- **States:** A "state" represents a specific behaviour or condition where an object or character can be at any given time. These states can include actions such as "Idle", "Walking", "Running", and "Jumping". Each state defines how the object or character behaves and interacts with the game world while in that state;
- **Transitions:** Transitions are the connections between states in a state machine. They specify the conditions that trigger a transition from one state to another. For example, a character may transition from the "Idle" state to the "Walking" state when the user presses a movement key and the transition to running when a user presses another key. Transitions often depend on variables, user input, animations, or other criteria;
- **Animations:** Each state in a Unity state machine is associated with animations or animation clips that control the visual representation of the object or character during that state. For instance, the "Walking" state might use a walking animation, while the "Jumping" state uses a jumping animation;
- **Parameters:** Parameters are variables that can be used to control transitions between states. Standard parameters include Booleans (true/false), floats (numeric values), and triggers (instantaneous events). For example, you might have a Boolean parameter "IsWalking" to determine if the character should transition from "Idle" to "Walking";
- **Layers:** In complex state machines, you can use layers to manage different animations and behaviours. For example, you might have a base layer for locomotion (walking, running) and an upper body layer for attacks or interactions.

4.4.1 Blender Non-Linear Animation Editor

Blender has a dedicated workspace called the NLA Editor for non-linear animation editing. In character and object animation, it arranges, combines, and manipulates animations in a timeline. The NLA Editor's main attributes and capabilities are as follows:

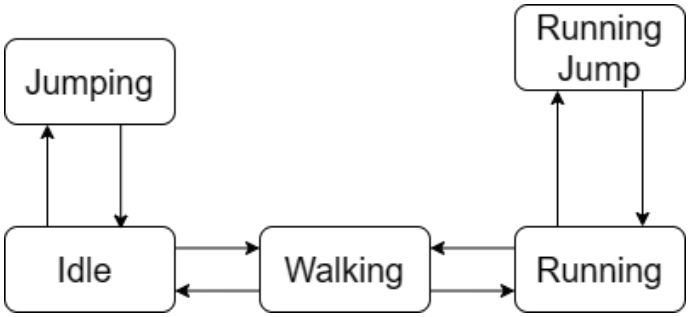

Figure 4.8: Example of a State Machine.

- **Action Strips:** The NLA Editor refers to animations as "action strips." Each action strip is associated with a particular animation or action that can be used on a character or object. These strips can be stacked and arranged to control the timing and blending of multiple animations;
- **Layering:** The NLA Editor enables users to layer different action strips on each other, giving some animations more priority than others. This makes it possible for animations to blend seamlessly, creating complex character animations by combining different movements like walking, running, and jumping with a hand waving and a head shake.
- **Blending and Transitions:** Action strips can be blended, allowing smooth transitions between animations. It is possible to manually control the blending mode, weight of the animation, and timing of these transitions to achieve a precise and realistic result;
- **Repeat and Offset:** Without having to duplicate keyframes manually, action strips can be repeated or offset in time to create cyclical animations like walking or breathing;
- **Masking:** The parts of an object or character affected by an animation action can be controlled using masks. This is particularly useful for isolating specific body parts during complex movements;
- **Synchronization:** The NLA Editor allows you to synchronize animations across multiple objects or characters, ensuring that actions coincide or with precise timing;

- **Simplification:** The NLA Editor makes it simpler to manage and edit long sequences by organizing and simplifying complex animations;

4.4.2 C#

C#(pronounced "C sharp") is a versatile language known for its ease of use, strong typing, and the support of a large developer community. It's often chosen for its robust development tools and integration with the Microsoft ecosystem, making it popular for various software development projects. It is the primary programming language of Unity.

4.4.3 Python

Python is a high-level, adaptable, and simple-to-read programming language renowned for these qualities. Numerous applications, including web development, data analysis, 3D object manipulation, and others, use it extensively. Both novice and seasoned developers like Python because of its clear syntax and sizable standard library. This programming language is commonly used to develop 3D modelling and animation software like Maya and is the primary programming language of Uchronia Project Blender Game Engine (UPBGE).

4.5 Acquisition of animations

To animate glosses, it is necessary to create the respective animations first. One could use a model in any 3D software and start to produce an animation for every gloss by hand. Still, the process would take a long time to achieve a considerable dictionary of glosses, and it would most likely produce animations with an automated look. A solution for this problem is to use a motion capture suit, which consists of a suit that can record and replicate the movements of a natural person in a 3D avatar in the software, as Figure 4.9 illustrates. For this project, two Motion Capture systems were used to capture the signs performed by an LGP speaker. These are full-body tracking suits utilizing sensors in critical points of a person's body that transmit data to a computer running its respective software to record the motion data in a 3D environment. In this work, several LGP speakers wore the suit several times and recorded long takes of glosses of the most

commonly used words.

Figure 4.9: Recording the gloss ["ABERTO"] with the rokoko suit.

Although both systems function similarly, they process data differently and have slightly different skeleton hierarchy with their avatars. This implies that to be able to use the motion captures made by both systems in one avatar, it is necessary to filter the data from both skeletons to a single universal skeleton. To achieve this, a Maya middleware was used, namely the Inverse Kinematics (IK), which is a run time solution for creating believable, interactive character animation for 3D environments that features full-body inverse kinematics and, more importantly for this work, real-time re-targeting technology, enabling characters to interact realistically with their environment.

The IK feature makes it possible to "transfer" the information from one skeleton to another, assuming both skeletons are humanoid skeletons with a similar structure. If both skeletons are compatible with IK, a control Rigg can be created for the skeleton from the captured motion data and our desired avatar, displayed in Figures 4.10 and 4.11. Once created, it's possible to assign the main skeleton to one controller to imitate it so the animation from one skeleton can be replicated in the other.

Figure 4.10: Maya IK Control Rigg setup.

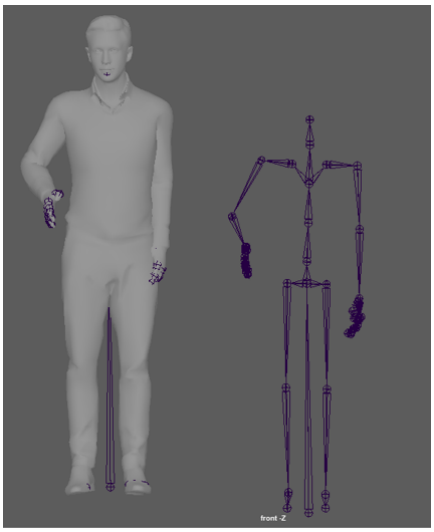

Figure 4.11: Animation transference with a Control Rigg.

After transferring the animation to the target skeleton, some adjustments to the motion capture data should be made before exporting the final animation to the database. These adjustments usually are slight movements caused by some interference when capturing the movement; for example, a finger is entirely straight when it should be curved into a fist, or a shoulder is rotated slightly outward. To do these corrections without disrupting the whole animation, the adjustment is made on frame 1 by creating a new keyframe for the skeleton. After that, with the help of Maya's graph editor, as highlighted in Figure

4.12, which helps visualize the animation curve of a bone in a given time frame, it is possible to visualize the difference between the keyframe values of the correct position and rotation of the selected bone to the rest of the animation. Seeing this difference is just a matter of adjusting the rest of the keyframe values following the first frame. This way, it's possible to maintain the visual fidelity of the captured movement while correcting any small mistakes; however, it is time-consuming to get a good-looking animation.

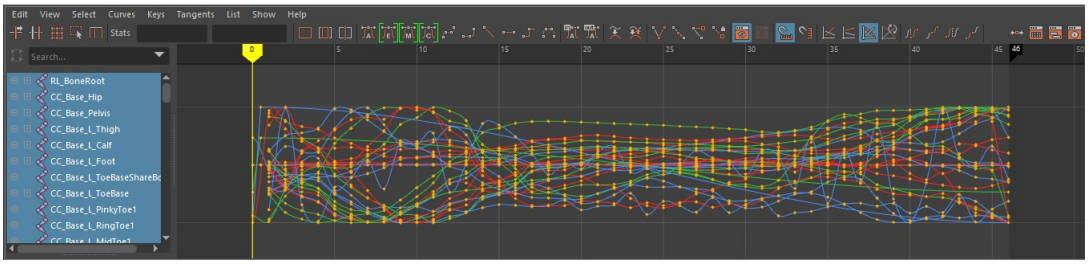

Figure 4.12: Maya Graph Editor.

With this animation collecting system assembled and a clean-up pipeline, the project requires an avatar for the animation. To create the avatar displayed in figure 4.13, represented directly in Unity, a combination of two software was used: ICLONE and Character Creator 4. The combination of these programs allows for the creation of several avatars with drastically different meshes, giving the whole system the ability to have several other models but with the same hierarchy of bones in the skeleton. This means that any model created with this combination of software will be able to be animated directly in Unity by using the same animations instead of requiring the re-recording of signs for each new model.

This avatar, combined with the motion capture data acquired by the previously mentioned systems, provides both Unity and Blender with a very realistic set of animations of the glosses, giving way to a very realistic Portuguese Sign Language speaking model.

Figure 4.13: Iclone Avatar.

4.6 Unity Transitions

Unity's primary function is for game and app development, so it can sequence several animations using a single avatar with relative ease through a state machine and compile it into an app for ease of use.

A state machine is a sort of "container" that stores the status of something at any given time, in this case, which state is the Avatar in. Then, when input is given, it can provide an output based on the current state, transitioning to a new state in the process. Unity typically uses the state machine to associate an animation to a character state, whether the character is running, jumping, or punching, for example, and assigns the corresponding animation.

The state machine is perfect for ordering the various animations needed for a translation.

Initially, to test Unity's capabilities when compiled into an app, a state machine with around 100 states was created to test the system regarding animation transitions, as shown in Figure 4.14. Even though the Unity project was intended to be a straightforward application, it resulted in a considerable App when it was compiled. This was due to

the sheer number of animations stored inside the project when first compiling. Although this poses no issue in this state of development and testing, it could start to add up to a massive app that would take a lot of storage and require a fresh install whenever new animations are acquired and added.

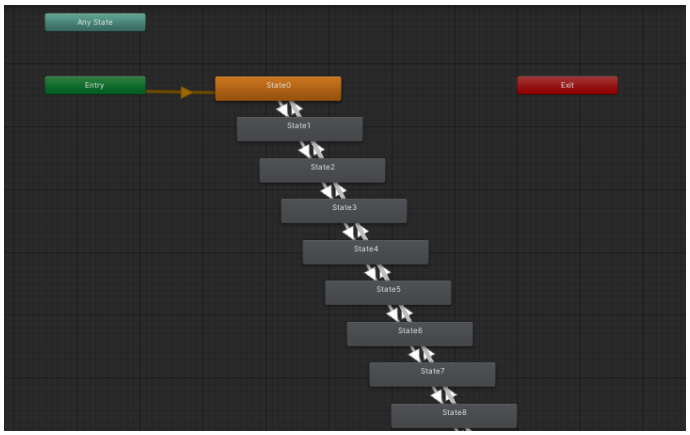

Figure 4.14: The first iteration of the state machine.

To solve this problem, an external database was created to house every animation that had been acquired through the process of this work. With this database, whenever new animations are acquired, they can be stored in it, making the compiled application very small in storage and easier to maintain and upgrade when needed. Whenever Unity requires animations, it calls the database for all the required ones. It then downloads all the necessary ones and orders them accordingly. When making new calls to the database, every animation previously downloaded is deleted if they are not needed for the new translation. In case the animation required for the translation is missing in the database, Unity will instead retrieve all the necessary letters to spell the gloss and spell the missing gloss. This spelling method is mainly used for personal names but is an excellent backup to maintain an understandable conversation in case some animations are missing.

However, spelling every personal name or missing animations vastly increases the number of states needed to animate the translated message properly. For example, if the translated message is represented by the glosses "[OLA]," "[CONSULTA]," "[QUERER]," and "[MARCAR]"(hello, appointment, want, book), it would need only four states. But if the second and last glosses are not in the database, the number of states required to animate the message goes from 4 to 16 due to spelling those missing glosses. Unity could

run out of states to store the animations needed if the message is long enough.

This problem was resolved by building a state machine with three states, those being idle, state1, and state2, illustrated in Figure 4.15, and by implementing an algorithm that saves the order of all glosses needed and, when required, overrides the animations in the states with new animations. While state1 and state2 display the signs, idle is the default animation when nothing happens. When it receives the required animations, it organizes them into the appropriate states and sets a condition stating that when one state's animation concludes, the next stage starts. An animation with one or two glosses animates what is required before returning to idle. If the animation, for instance, contains four glosses, the algorithm activates, and it works like the following:

1. Animate Gloss 1 in State1;
2. Animate Gloss 2 in State2 (and change the animation in State1 to Gloss 3);
3. Animate Gloss 3 in State1 (and change the animation in State2 to Gloss 4);
4. Animate Gloss 4 in State2;
5. Go to idle;

Theoretically, this approach provides Unity with limitless scalability when animating consecutive glosses. This scalability holds significant importance in LGP communication, as a person's name in LGP is invariably spelt using the LGP alphabet.

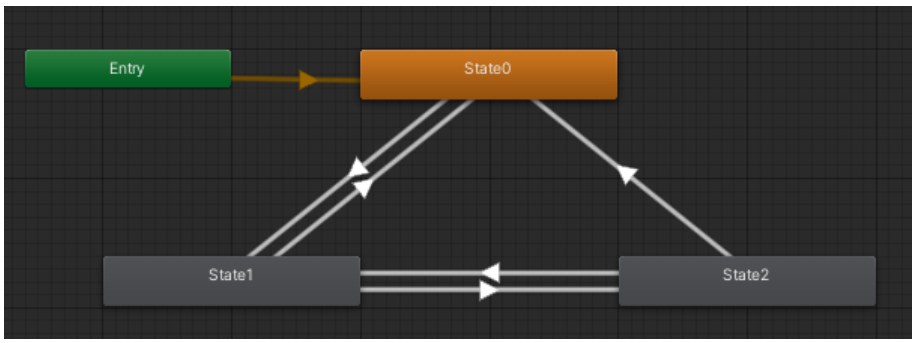

Figure 4.15: Final State Machine.

Unfortunately, there is an issue that has become apparent with this approach. That is, the avatar will always return to the "neutral" position of all gloss animations because

the avatar won't start the following animation until the previous one has finished playing out. Due to the lengthy translation and frequent pauses, the animated message becomes challenging to understand. It is, fortunately, possible to define a "hasExitTime" and "ExitTime" in each state of the state machine to address this issue.

This condition indicates Unity will start changing states, in this case, to the next state. When activating this condition, Unity will begin the next animation after 75%, by default, of the current one has been played. By having both animations running simultaneously, transition works by comparing the positions and rotations of each bone of the skeleton of each animation and, over the defined "ExitTime", approaching the values of the first animation skeleton with those of the second. Using this feature, the first animation doesn't stop abruptly and smoothly transitions to the second. This process required a lot of refinement, especially the transition time. If the time were short, the avatar would have pauses between animations, as mentioned above. Still, if the transition time was large, the avatar could not complete a gesture and would start the next one in advance, losing the animation sense.

Finally, this whole component can be compiled into a Web Graphics Library application, displayed in Figure 4.16, which can later be implemented in a web app or a smartphone, only requiring an internet connection. The format Web Graphics Library was chosen due to its compatibility to be integrated, with relative ease, in a web app, web browser, or smartphone application, and the ability to make function calls programmed in the unity application from the outside that affects it. Unity receives the text it needs to translate and animate through its capacity to make useful calls.

Since the processing power behind every translation will be located right in the local Web Graphics Library application, this technique for overall system operation and smooth animation transition appears to be more efficient in terms of scalability and user wait time. In other words, the user will only be reliant on the speed of their Internet connection to download all the required animations for each translation. In this manner, the server hosting the animation database's bandwidth will be the system's sole point of failure.

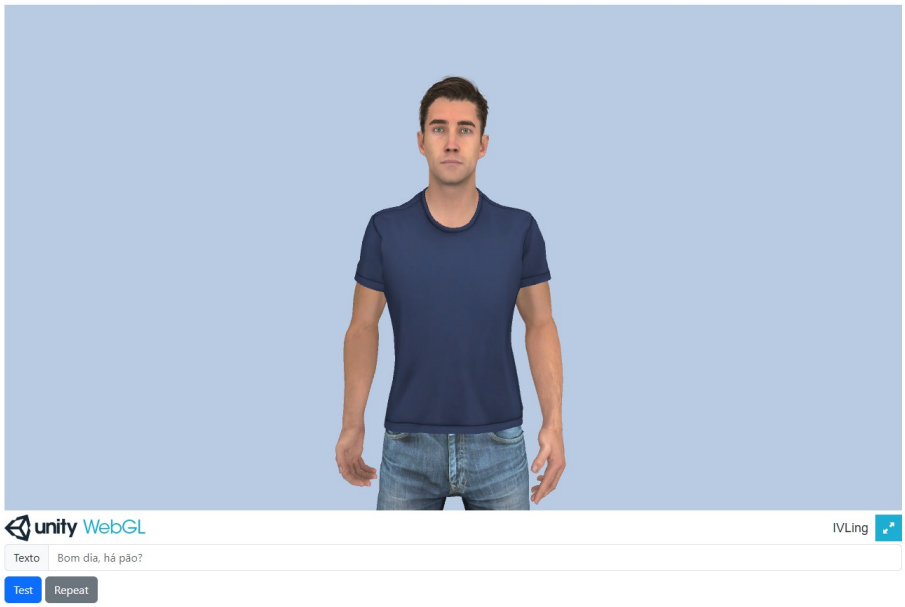

Figure 4.16: WebGL application.

4.7 Blender Transitions

Blender is a free, open-source 3D computer graphics software application for 3D animation, modelling, and rendering with a large community. It's a versatile and powerful tool with a user-friendly design and a plethora of functionality. Being open-source, code can be written and implemented in Blender to create several tools and add-ons to customize. In the case of this work, the Blender feature NLA editor can be modified to develop an automatic animation module. Utilizing Blender's NLA editor, a similar effect can be achieved as the state machine in Unity in which several animations are ordered with transitions between them to create the illusion of one unified animation. The NLA editor, illustrated in Figure 4.17, was created to accelerate the animation process in a non-linear fashion by forgoing the usage of keyframes and utilizing Actions. Blender achieves this by creating action blocks based on keyframe animations that can be freely manipulated and given to any object in the scene on a timeline. To merge actions, the NLA editor provides each object with several tracks with different priority levels: the lowest track has the least priority, and the highest track has the highest priority. This means that the actions in the top tracks will be prioritized over those in the lower tracks and will overwrite the lower actions with the top ones. This feature is primarily used to speed up background characters' shared animations, such as walking on a sidewalk, and to combine two distinct

animations on one object to produce a custom animation. For instance, you could combine a humanoid character's walking animation with a hand-waving animation to produce a walking person waving their hand. This feature to fuse two separate animations could prove helpful in other types of Sign language. It is not needed for LGP.

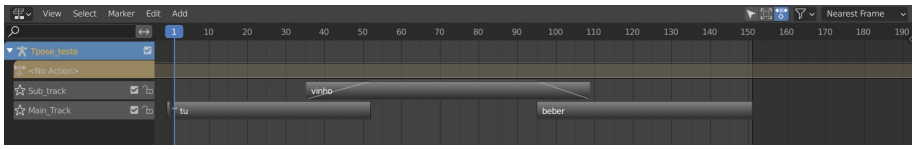

Figure 4.17: Example of an NLA editor.

However, for this to be a usable component of animation transitions for this work, there needs to be some customization of this feature, and to achieve it, some preparations were needed. First, there needs to be an object that will be able to receive all animations that will later be exported, in this case, a simple skeleton with all bones completely straight with the arms of the skeleton straight, forming a T shape so that it has the same neutral starting position as every animation. Next, the animations are inserted into the Blender scene. While in Unity, one could import the necessary assets and continue to work, Blender requires more preparation when importing assets. First, every animation must be exported from Maya with different settings because Blender is also a 3D modelling and animation software, mainly how the animation is "baked". When exporting an animation with an FBX file, the software will start a process called "Baking," the 3D software adds a keyframe for the location, rotation, and scale between every keyframe created on the skeleton on a given timeline. This process allows the final file to have precisely the created motion, reducing the probability of the software misreading the animation. This is necessary because, as stated earlier, Blender possesses a feature that allows it to fuse two or more animations into a single new one, and it achieves this by only animating the bones that have more keyframes than just a start and end keyframes leaving the other bones static. For example, the animation for the gloss "BOM_DIA"(good morning) only animates the dominant hand of the avatar while leaving the non-dominant arm in a resting position. When transferring this animation data to the skeleton in Blender, since only the dominant hand and arm are animated, the non-dominant arm will remain in the skeleton's T-pose, giving the final animation a strange appearance. To remedy

this, when exporting the animations from Maya, it is necessary to bake every keyframe of the animation in every bone of the avatar's skeleton to ensure that Blender maintains the animation's fidelity. Creating and correctly naming the respective action is necessary when importing an animation. This is easily done in the NLA Editor, and once the action has been created, it can be manipulated and inserted into another object in the NLA editor. When fully organized, the Blender scene became as shown in Figure 4.18.

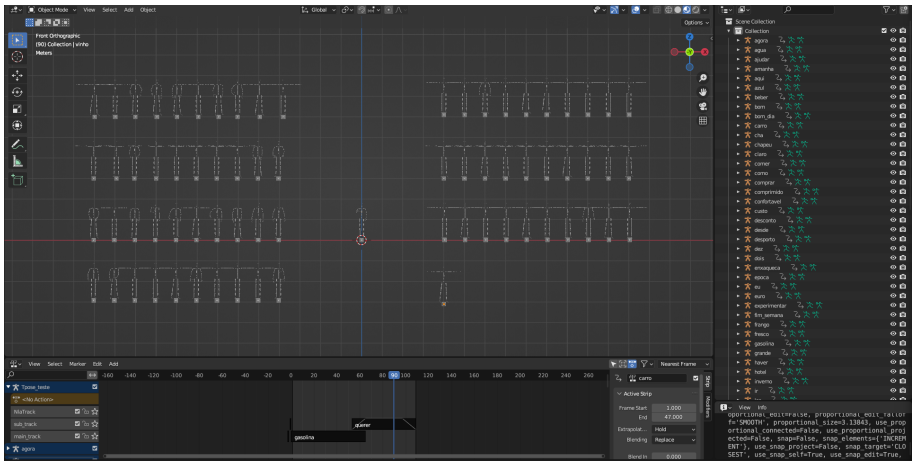

Figure 4.18: Final Blender instance.

The transitions can be created now that Blender has all the necessary actions. There are two ways to make a transition between two animations in the NLA tracks. The first is to line two animations in a single track and add a transition action block between them. Although straightforward, this method produces inconsistent results because every animation has a starting and ending T-pose frame. This means that when making the transition between two animations, the Avatar would "flap" his arms due to the last frame of the first animation and the first frame of the second animation, and this would be a persistent problem with every single animation afterwards.

The second way is to have another track above the main one to alternate the animation block with the main track. This means that the first animation will be on the main track, the second animation on the new track, the third on the main track again, the fourth on the new track, and so forth, and blend every animation in the second track with the main track. Using this method, Blender can merge animations in different tracks by slowly mixing the second animation with the first, meaning when the first animation is about to end, the second one has already started and continues with the second animation. This

blending can be tweaked so that the transitions begin in later animation frames, making them nearly impossible to detect. This second method was chosen due to better results overall and the elimination of the arms flapping due to the T-poses the first method had. Finally, an algorithm was created that, when receiving a string of glosses with the set of animations required, will first check to see if it possesses all the animations requested, then clear out any unnecessary animation block of both tracks currently in the base skeleton, and then will automatically assign them to their respective tracks and blend them. After that, it will export the whole sequence of animations to a predetermined folder that will later be sent to the Unity application to animate.

While Unity's module requires an external database to house all the animation acquired, all animations will be inside the blender file that creates the final animation. This was done because while it is possible to develop an algorithm to import all the necessary animations from a database to Blender and then fuse them to create the final animation, with transitions, and then export the final animation, it would increase substantially the time required to develop the whole translation while having every animation inside Blender speeds up the entire process.

At this stage of development, a problem was apparent. While Blender has support for custom algorithms, it requires a manual trigger of said algorithms to execute, and for this to be a viable component, it must be automated. In searching for ways to solve this problem, it was discovered that in the past, Blender had a dedicated Game engine in which users could go from 3D modelling and animating to developing games without ever leaving Blender. With this game engine, Blender's Python receives a significant improvement, gaining access to the creation of controllers that are constantly running, making it possible to have several scripts and algorithms associated with said controllers that can be running every second or external source can call that, be it local or online, or even create a dedicated WebSocket server that can send, receive, and process data while eliminating every unnecessary function and feature that the normal Blender software possesses. All of this is compiled into a smaller executable program. Using the now-known as Uchronia Project Blender Game Engine (UPBGE) to compile all the work done in Blender has a run time application, it became possible to have a less bloated and more focused version of Blender to fuse and export animations.

To test the viability of this newly compiled Blender application, a bare-bones interface was created, as shown in Figure 4.19, to input the animations to be merged, the directory in which the final animation will be exported and to show the time the application took to export it.

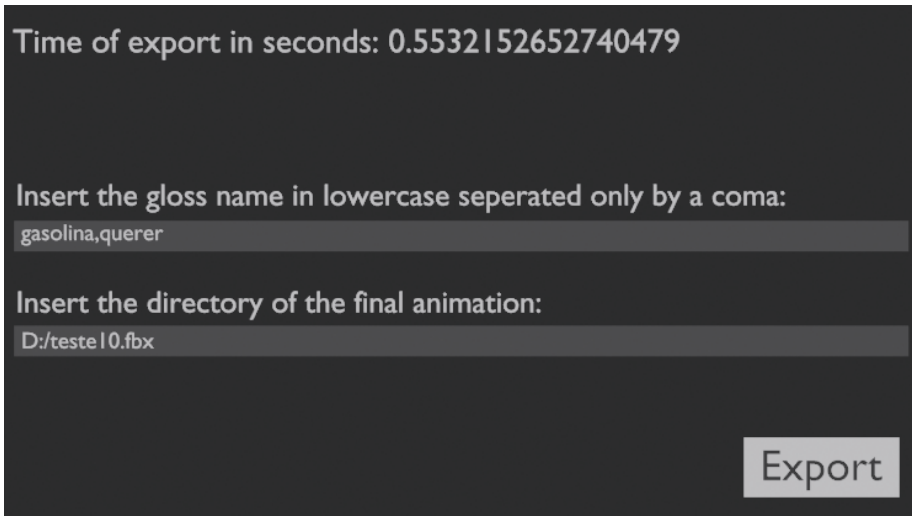

Figure 4.19: Compiled Blender Application.

Although this compiled version solved our main problem, it created another, less problematic issue. To update this new version with more animations, one must compile a new application with the updated animations each time. This problem could easily be solved by either creating an algorithm that can import FBX files or by implementing a database with all the animations and connecting Blender to it; however, due to time restraints, this was not possible. This problem does not affect the testing of the component developed here. Just an improvement for future work.

4.8 First Impressions of both modules

Initial local tests after the development of both components show a significant difference in how the avatar transitions between glosses. Unity makes the transitions between animations very fast, which in most cases goes unnoticed; however, if the glosses have large arm and hand movements, the transition looks rushed, while the Blender transitions show better and smoother final animation, having a very realistic transition between each gloss. In terms of the full system, Unity is noticeably fast, depending only on the

download speed of the individual animations to present the user with the translated animations. Here is where the Blender system struggles because after receiving a request, Blender takes some time to combine all the animations required. However, this time has been greatly reduced and optimized in the UPBGE compiled version of Blender, going from an average of 2 seconds to export to an average of 0.64 seconds. When integrating the Blender component with the whole system, a problem became apparent. In which the software developed could not export more than one complete animation at a time when receiving several requests. When attempting, the software would ignore the new request until the first one is exported. While this shouldn't be a problem for when the system is used with just one person, in a real-world scenario, where it is expected to have several hundred, or even thousand, users sending requests to the system, this would quickly fail. Several attempts at fixing this issue were met with disappointing results, from creating a multi-thread algorithm that could export several animations simultaneously to opening a new instance of the software in the server to make the new animation whenever a request was received. Although not optimal, the final solution was to implement a messaging system in which Blender stores every request received in order and exports them individually. While this enables multiple users to access the system and receive their respective animations, in a real-world scenario, this solution would introduce a considerable time gap between sending a message and seeing the avatar getting animated.

4.9 Conclusion

This chapter began by discussing the overall system architectures and some of the more detailed components that make up the system, focusing more on the elements critical to completing the work outlined here, starting by showing the system's basic framework. Then, some possible routes for the complete system were discussed, and it was determined that Unity would be the engine to house the Avatar since it allows a lot of flexibility between every possible architecture. After this, the final architecture for the overall system was decided and explained. Then, the two possible routes for the animation system, Unity's state machine and Blender NLA track implementations, were explained briefly with their respective architectures, thought processes, and external sources needed, which

are the required database and a text-to-speech API. Regarding these external sources, two databases are necessary with this architecture: the relational database containing information about the Asset Bundles and their animations and a file system that stores them. Before providing the complete Gloss-to-Avatar architecture, a brief description of the state machine and why it is essential in this architecture was also given.

The second part of this chapter gives a detailed presentation of the project components and the solutions implemented. Firstly, a description is given of the method used to obtain a library of animations to be used by both components developed. Secondly, the Unity transition component was created using the Unity state machine. Due to the nature of the system, it was determined that the best way to utilize the Unity instance on the user side is to have an "infinite" state machine to animate any number of glows. After that, Blender's alternative transition component was developed by understanding how the NLA editor and tracks work and creating scripts to manipulate a track to have all the necessary animations and create the transition between them. After testing the scripts, the problem of activating them manually became apparent, and a solution was found in the UPBGE software, developed by the community and derived from a discontinued Blender game engine. With it, it was possible to compile Blender's scripts and functions into a small program, increasing its performance and improving Python's capabilities in Blender. Finally, a local test was conducted between both components to check that they worked correctly within the system.

Chapter 5

Tests and Results

Both components will be tested in this chapter, and the results obtained when using them to carry out translations will be analyzed and discussed. These tests are essential because one of the project's objectives is to develop a workable solution that allows deaf users to receive translations quickly and efficiently. These tests will be split into two parts. The first is focused on the quality of the animation and transitions produced by each module independently, and the second test will focus on how effective each module is in the whole system. This means how long each module takes to create the animation and how much data was used from the moment it was received. These tests were all conducted in the computer mentioned in 4.3 and limiting internet trafficking to a speed of 4g, up to 100 Mbps. After the tests, their results will be presented and discussed, which will allow a fair comparison between both Unity and Blender animation modules

5.1 Animation Transition Test

To test both modules developed here regarding animation and transition quality, a series of 15 dialogues were given to the system to animate, record, and upload to a questionnaire. These recordings were then shown to evaluate the transition quality between the animations without revealing what animation came from what system. Per phrase, the evaluators could choose which recording they found better in quality or if there was no difference. The testing phase took two weeks to finish, so we asked LGP interpreters and teachers to respond to the questionnaire. The goal of 5 to 10 participants was met with 6

because the questionnaire took a long time to complete. The analysis of the unprocessed data was saved and will be presented in the ensuing sections. Although the number of participants is small, the answers given are provided by experts in the field. Since the tests aim to compare which has better animations, we think their quality matters more than whether or not regular users can understand the animated signs.

As shown in both Figures 5.1 and 5.2, Blender seems to have produced an on-par or better animation than Unity in the majority of the videos, with Blender being the preferred module in 4 videos, with Unity only having the better animation in one of the videos recorded and both being considered equals in the rest.

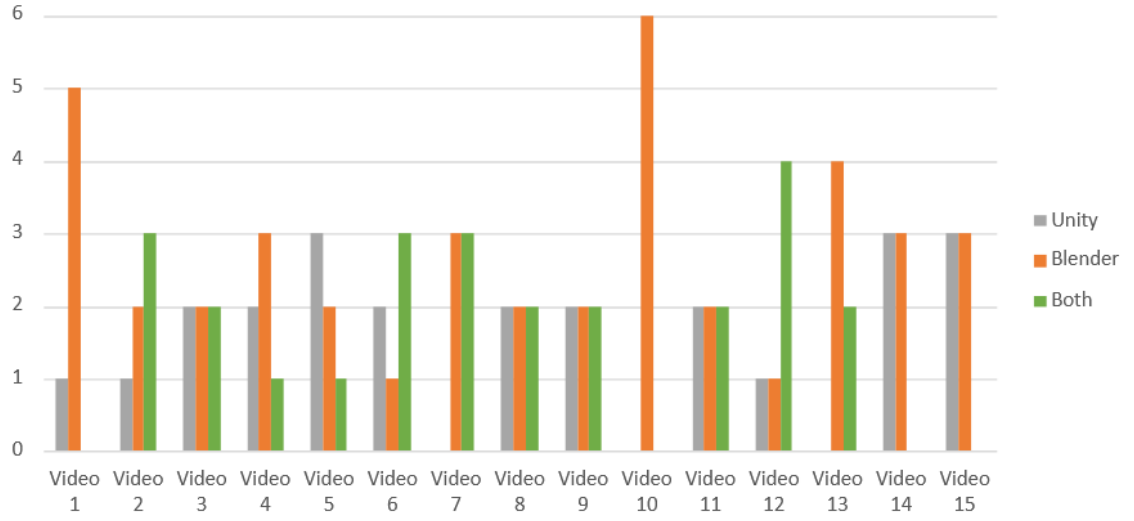

Figure 5.1: Results of the questionnaire

These results show that Blender possesses superior animations compared to Unity's. The percentage shows that Blender was the preferred module for producing the animations. However, one must consider that both systems can be fine-tuned when creating animations and that the Blender module could have just been better tuned. This means that the Unity module could still be improved to bring the results in its favour and that we will need to analyze both modules' effectiveness to choose the better one.

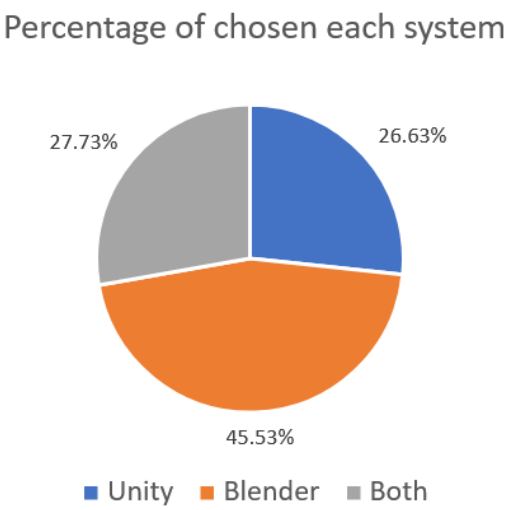

Figure 5.2: Percentage choice

5.2 System effectiveness

Although the animations are, by far, the most crucial part of this work, something that cannot be ignored is the time and resources it takes to be able to reproduce those animations, for it needs to be both fast and effective for it to be a successful system. To test both components' efficiency in the whole system, while recording every phrase used in the previous test, the time it took from the moment of submission of each word until the start of the animation and the amount of data downloaded was noted from both components for comparison between the two. To measure the time, both components had a script that would signal the moment the request was sent and the moment the Avatar would start to animate. After every request, we noted all the files downloaded in the browser's network monitor to calculate the amount of data used. Observing the results illustrated in Figures 5.4 and 5.3, it is apparent that not only is the Unity module incredibly fast, having every single translation animation started in less than one second but also that by just sending each animation required individually, it can be a less data-draining system when compared to Blender' slow time and large file size.

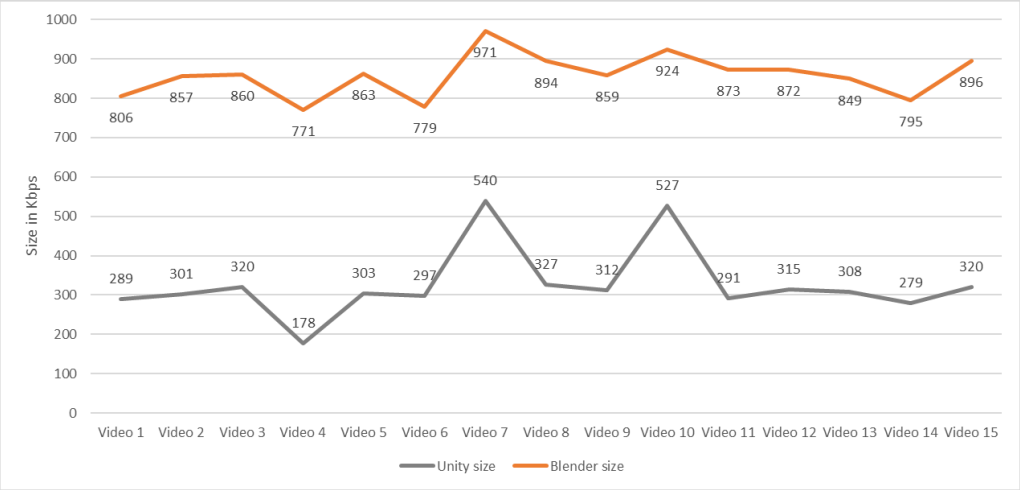

Figure 5.3: Size of resulting animation requests

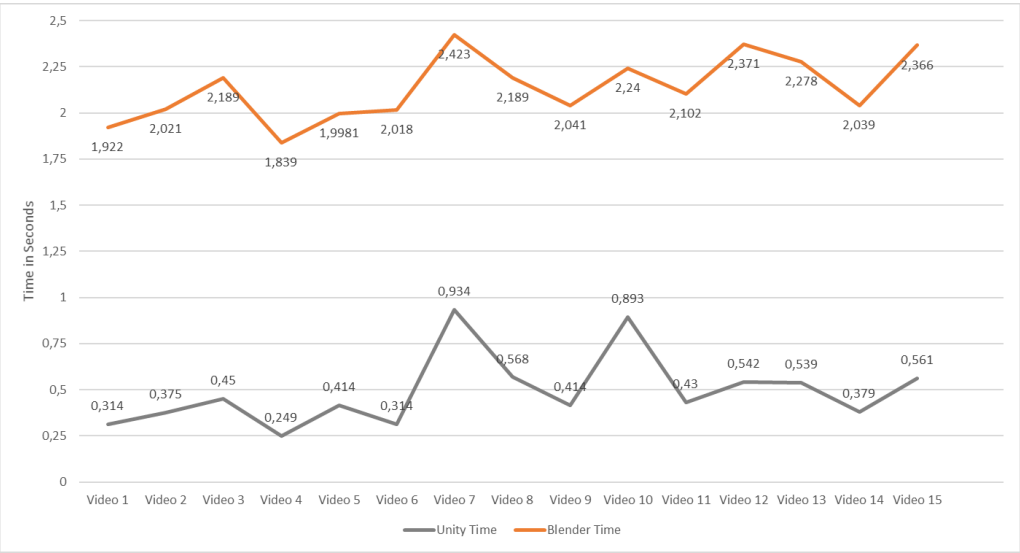

Figure 5.4: Time taken of resulting animation requests

Most Blender results can easily be attributed to the fact that when receiving a request, it takes some time for the animation to be combined and exported, in addition to downloading the exported file. Another point that needs to be reinforced here is that the Blender component cannot process more than one request at a time, meaning that in a real-world scenario, the time it would take for a user to get the animation of a request increases tremendously depending on how many people are using the application. At the same time, Unity does not have this problem.

5.3 Conclusion

After performing the necessary tests on both solutions, a comparison must be made to select which is better. To summarize the results gathered previously, Blender was the system with the most well-evaluated animations; however, the time it took to produce the animation was, in many cases, more than double the time it took Unity to achieve the same effect. At the same time, Unity was the most effective system in terms of both the amount of data required by the user being lower and the speed at which it could animate a translation being incredibly fast. Unity brings a better user experience, with massive scalability due to the processing power in the user's phone and the ability to handle thousands of requests without delay. This is a critical flaw that the Blender module has: it can only take requests one at a time, which, in a real-world scenario, there will be massive delays between the moment a user sends a request and the moment the Avatar finally animates. For that reason, we determined that the Unity module is the best option to be used in the final application despite Unity's animation not being on par with Blender's.

Chapter 6

Conclusions and future work

6.1 Conclusions

Social inclusion has received much attention, with some studies concentrating on the different computer programs that must be developed for various circumstances, like physical limitations and mobility. The development and application of systems for deaf people to communicate with the hearing community is one area of application that, despite some advancements over the years, is still lagging in this evolution of the development path of work for social inclusion. In most communities, especially Portuguese ones, it is common knowledge that a deaf person can read Portuguese text, but this is untrue, as almost all deaf people use Portuguese Sign Language as their primary language.

The work presented here aims to propose an architecture for an animation platform between Portuguese and Portuguese Sign Language and, mainly, to study, develop, and demonstrate a viable solution for animating the translation between Portuguese and Portuguese Sign Language. Most solutions implemented in this area, regardless of what sign language they are developed on, discard the animation module, opting to focus more on translating written text to glosses and barely acknowledge the ability to represent a clear and readable animation that said translation. On the other hand, this document's solution focuses on producing the most realistic animation and smoother transition, giving the deaf user a better "reading" experience than most systems.

The results show that developing one of these animation modules is a viable component for a mobile application that can create video calls and have a 3D Avatar translating

from Portuguese to LGP, with only a small waiting time between sending a text message and having it translated and animated. However, through testing, it was noticed that there is a lot of room for improvement in both the chosen modules, namely fine-tuning the transition between animations for the chosen Unity module and developing a more efficient architecture for the Blender module and further exploration of its capabilities.

In light of all of this, it can be concluded that the primary goal of this work was accomplished. It is now expected that the knowledge and skills gained here will be applied to future research and the development of applications that facilitate effective communication between hearing and deaf individuals with the same ease as these applications facilitate communication between speakers of other non-sign languages. All stages of the methodology adopted and described in chapter 1.3 were followed throughout the various sections presented in this work, with the Problem Identification and Motivation activity described in section 1.1 and chapter 3, Definition of Objectives for the Solution activity presented in section 1.2, Design, Development and Demonstration activities are offered in chapter 4, Evaluation activity in chapter 5, and, finally, the Communication and Dissemination activity carried out through the publication of a scientific article at an international conference, presented in Appendices A.

6.2 Future work

Since this is a work that explores an approach to the animation of automatic translation between Portuguese and the glossary of Portuguese Sign Language, and also considering the current state of the existing bibliography on Sign Language animation in general, an easy future job is to fine-tune and improve the Unity State Machine, on the client side, to have a better and more realistic moving avatar and to include several different Avatars to give the final application some customisation. Although this is already happening, one task that will need to continue is increasing the number of animations captured for translation into the database. A possible future project is to turn this system into an educational tool to help people learn and understand LGP through its visualisation. Another possible path is to continue exploring Blender's capabilities, either to perfect the work done here and find a way to create a compiled version that can be integrated into a web browser or smartphone

app or to create new software from scratch to be used only for animating glosses, using other tools and programming languages. A final possible path is to develop an automatic learning model to make animations based on video recordings of LGP gestures.

References

[1] Associação Portuguesa de Surdos. *Língua Gestual Portuguesa - APSurdos*. URL: https://apsurdos.org.pt/?page_id=3725.

[2] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24 (3 Dec. 2007), pp. 45–77. ISSN: 0742-1222. DOI: 10.2753/MIS0742-1222240302.

[3] I. N. d. Estatístic. *Censos 2021*.

[4] Sandra Silva Isabel Mesquita. *Guia Prático de Língua Gestual Portuguesa*. Editora Nova Educação, 2009. ISBN: 9789728200909.

[5] Maria Raquel Delgado Martins Amandio Coutinho Maria Augusta Amaral. *Para Uma Gramática da Língua Gestual Portuguesa*. Editorial Caminho, 1994. ISBN: 9789722109819.

[6] Rute Ana Ferreira Rodrigues. *Compreensão da Língua Gestual Portuguesa em Crianças Surdas*. 2017.

[7] Apple Inc. *Arkit Documentation*. URL: https://developer.apple.com/documentation/arkit/arfaceanchor/blendshapelocation.

[8] C. Webster. *Animation: The Mechanics of Motion (1st ed.)* Routledge, 2005.

[9] University of Virginia. *3D Animation: Keyframing*. 2017. URL: https://web.arch.virginia.edu/arch545/handouts/keyframing.html.

[10] Blender Foundation. *Blender - a 3D modeling and rendering package*. Version 2.93.5. URL: https://www.blender.org.

[11] Autodesk, INC. *Maya*. Version 2022. URL: https://www.autodesk.com/maya.

[12] Reallusion Inc(2022). *Character Creator 4*. URL: https://www.reallusion.com/character-creator/.

[13] Reallusion Inc(2022). *Iclone8*. URL: https://www.reallusion.com/iclone/.

[14] Autodesk, INC. *FBX*. Version 2022. URL: https://www.autodesk.com/products/fbx/overview.

[15] Khronos Group. *glTF*. URL: https://www.khronos.org/gltf/.

[16] Microsoft. *Microsoft Kinnect*. URL: https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows.

[17] Rokoko. *Rokoko Studio*. URL: https://www.rokoko.com/products/studio.

[18] Noitom. *Perception Neuron: Capturing The Essence Of Motion*. URL: https://neuronmocap.com/pages/academic.

[19] Epic Games. *Unreal Engine*. Version 4.22.1. Apr. 25, 2019. URL: https://www.unrealengine.com.

[20] Juan Linietsky; Ariel Manzur. *Godot*. URL: https://godotengine.org/.

[21] Unity Technologies. *Unity*. Version 2020.3.38f1. URL: https://unity3d.com/unity/.

[22] UPBGE. *Uchronia Project Blender Game Engine*. Version 0.36.1. URL: https://upbge.org.

[23] Blender Foundation. *Blender Game Engine*. Version 2.79. URL: https://docs.blender.org/manual/en/2.79/game_engine/index.html.

[24] Matthew J Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* (Mar. 2021), n71. ISSN: 1756-1833. DOI: 10.1136/bmj.n71.

[25] Ruben Emanuel Ramires dos Santos. "PE2LGP: From Text to Sign Language (and vice versa)". In: (2016).

[26] Matilde Gonçalves et al. *PE2LGP: translating European Portuguese into Portuguese Sign Language glosses*. URL: https://github.com/own-pt/openWordnet-PT.

[27] Paula Escudeiro et al. “Virtual Sign – A Real Time Bidirectional Translator of Portuguese Sign Language”. In: *Procedia Computer Science* 67 (2015), pp. 252–262. ISSN: 18770509. DOI: 10.1016/j.procs.2015.09.269.

[28] Khronos Group. *WebGL*. URL: https://www.khronos.org/webgl/wiki/Main_Page.

[29] Melissa Real, Martin Santamaria, and Gonzalo Olmedo. “An architecture for the implementation of the Ecuadorian sign language into digital television system”. In: IEEE, Oct. 2020, pp. 1–5. ISBN: 978-1-7281-9365-6. DOI: 10.1109/ANDESCON50619.2020.9272101.

[30] Alexis Heloir and Michael Kipp. “EMBR: A realtime animation engine for interactive embodied agents”. In: IEEE, Sept. 2009, pp. 1–2. ISBN: 978-1-4244-4800-5. DOI: 10.1109/ACII.2009.5349524.

[31] Kipp Michael, Heloir Alexis, and Nguyen Quan. *Intelligent Virtual Agents*. Ed. by Hannes Högni Vilhjálmsson et al. Vol. 6895. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23973-1. DOI: 10.1007/978-3-642-23974-8.

[32] Lynette van Zijl and Dean Barker. “South African Sign Language Machine Translation System”. In: ACM, Feb. 2003, pp. 49–52. ISBN: 1581136439. DOI: 10.1145/602330.602339.

[33] Jeffrey Harper. *Mastering Autodesk 3ds Max 2013*. John Wiley & Sons, 2012.

[34] Web3D Consortium. *VRML*. URL: https://www.web3d.org/.

[35] Ying He et al. “Design and Implementation of a Sign Language Translation System for Deaf People”. In: IEEE, Mar. 2021, pp. 150–154. ISBN: 978-1-6654-1411-1. DOI: 10.1109/ICNLP52887.2021.00031.

[36] Ijya Chugh et al. “Techniques for key frame extraction: Shot segmentation and feature trajectory computation”. In: *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*. 2016, pp. 463–466. DOI: 10.1109/CONFLUENCE.2016.7508164.

[37] Microsoft Corporation. *Windows Presentation Foundation*. URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0.

[38] M. Punchimudiyanse and R. G. N. Meegama. "3D signing avatar for Sinhala Sign language". In: *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*. 2015, pp. 290–295. DOI: 10.1109/ICIINFS.2015.7399026.

[39] Jing Wang, Yanfeng Sun, and Lichun Wang. "Chinese Sign Language Animation System on Mobile Devices". In: *2010 Second International Conference on Information Technology and Computer Science*. 2010, pp. 52–55. DOI: 10.1109/ITCS.2010.19.

[40] Khronos Group. *OpenGL ES*. URL: https://www.khronos.org/opengles/.

[41] Kamel Ayadi, Yahya O.M. Elhadj, and Ahmed Ferchichi. "Prototype for Learning and Teaching Arabic Sign Language Using 3D Animations". In: *2018 International Conference on Intelligent Autonomous Systems (ICoIAS)*. 2018, pp. 51–57. DOI: 10.1109/ICoIAS.2018.8493979.

[42] VCom 3D company. *V-Communicator*. URL: https://www.vcom3d.com/?id=mobile.

[43] European Union's Fifth Framework. *eSIGN at UEA*. URL: http://www.visicast.cmp.uea.ac.uk/eSIGN.

[44] Thomas Hanke. *HamNoSys—Representing sign language data in language resources and language processing contexts*. May 2004.

[45] Paramat Ittisarn and Nunnapad Toadithep. "3D Animation Editor and Display Sign Language System case study: Thai Sign Language". In: *2010 3rd International Conference on Computer Science and Information Technology*. Vol. 4. 2010, pp. 633–637. DOI: 10.1109/ICCSIT.2010.5564559.

[46] Microsoft Corporation. *Microsoft XNA*. URL: https://learn.microsoft.com/en-us/previous-versions/windows/xna/bb196990(v=xnagamestudio.42).

[47] Extensible Markup Language (XML). *World Wide Web Consortium (W3C)*. URL: https://www.w3.org/XML/.

[48] Pankaj Sonawane et al. "Speech To Indian Sign Language (ISL) Translation System". In: *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*. 2021, pp. 92–96. DOI: 10.1109/ICCCIS51004.2021.9397097.

[49] Google LLC. *Android*. URL: https://www.android.com.

[50] Google LLC. *Speech-to-Text: Automatic Speech Recognition*. URL: https://cloud.google.com/speech-to-text/docs?hl=pt-br.

[51] Fabrizio Nunnari, Shailesh Mishra, and Patrick Gebhard. "Augmenting Glosses with Geometrical Inflection Parameters for the Animation of Sign Language Avatars". In: *2023 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)*. 2023, pp. 1–5. DOI: 10.1109/ICASSPW59220.2023.10193227.

[52] Sindey Carolina Bernal Villamarin et al. "Application design sign language colombian for mobile devices VLSCApp (Voice Colombian sign language app) 1.0". In: *2016 Technologies Applied to Electronics Teaching (TAEE)*. 2016, pp. 1–5. DOI: 10.1109/TAEE.2016.7528378.

[53] DAZ Productions. *DAZ 3D*. URL: https://www.daz3d.com/.

[54] Smith Micro Software. *Poser - 3D Rendering and Animation Software*. URL: https://www.posersoftware.com/.

[55] Yosra Bouzid and Mohamed Jemni. "tuniSigner: A Virtual Interpreter to Learn Sign Writing". In: *2014 IEEE 14th International Conference on Advanced Learning Technologies*. 2014, pp. 601–605. DOI: 10.1109/ICALT.2014.176.

[56] Valerie Sutton. *SignWriting*. 2000.

[57] Oussama ElGhoul and Mohamed Jemni. *WebSign: A system to make and interpret signs using 3D Avatars*. 2011.

[58] Vasco António Gonçalves Alves et al. *Glossifica : tradutor automático de português para a Glosa da língua gestual portuguesa*. 2023.

[59] Vasco Alves et al. *Neural Machine Translation Approach in Automatic Translations between Portuguese Language and Portuguese Sign Language Glosses*. 2022. DOI: 10.23919/CISTI54924.2022.9820212.

[60] Vasco Alves et al. "A Gloss Based Translation From European Portuguese to Portuguese Sign Language". In: *2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2023, pp. 1–4. DOI: 10.1109/IWSSIP58668.2023.10180304.

[61] Bruno Ribeiro et al. “A Translation System from European Portuguese to Portuguese Sign Language”. In: *2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2023, pp. 1–5. DOI: 10.1109/IWSSIP58668.2023.10180292.

[62] Bruno Ribeiro et al. “Capturing and Processing Sign Animations to a Portuguese Sign Language 3D Avatar”. In: *2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP)*. 2023, pp. 1–5. DOI: 10.1109/IWSSIP58668.2023.10180233.

Appendices

Appendix A

Published Work

A Translation System from European Portuguese to Portuguese Sign Language

Bruno Ribeiro[1], Duarte Dias[1], Vasco Alves[1], Pedro Miguel Faria[1], Luís Romero[1]
[1] ADIT-LAB, Instituto Politécnico de Viana do Castelo, Viana do Castelo, Portugal
{bfiliperibeiro,duartedias,vascoa}@ipvc.pt
{pfaria,romero}@estg.ipvc.pt

2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP) | 979-8-3503-3729-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/IWSSIP58668.2023.10180292

***Abstract*—The deaf community is usually isolated from the rest of the hearing community, and this problem is found in many countries, including Portugal, where only about 1% of the population know Portuguese Sign Language. There are still no suitable applications to help them and, therefore, in this work, a system is proposed that translates European Portuguese into Portuguese Sign Language, showing the translation to an end user through a 3D Avatar. The system is divided into two components, the first being the textual translation system and the second the translations' animated representation, using a 3D Avatar. The former uses a hybrid solution of rule-based and neural approaches. The latter uses a motion capture system to represent the signs' gestures and creates a visual interface to animate the translation through an avatar. The implementation consists in a system receiving text in Portuguese, converting it into Portuguese Sign language, getting the corresponding animations from a database and animating a 3D Avatar with these animations. The system aims to assist deaf people communicating in all scenarios of the everyday life. The first results obtained are very promising for the added value of the system under development.**

***Index Terms*—LGP, Portuguese Sign Language, Neural Network, Virtual Avatar, Gloss Notation.**

I. Introduction

Regarding Sign Languages (SL), William Stokoe [1] explained that one of the first questions presented by those unacquainted with the subject is whether there is a universal SL. Although humans universally make use of facial expressions, arms and other parts of the body to signal, this is not the case when these signs are structured to create complex systems of words and sentences. These grammatical-lexical systems are unique, not only to specific nations, but to some bigger nations, which have multiple SLs and/or dialects. The Língua Gestual Portuguesa (LGP – Portuguese Sign Language) is one of the three official national languages of Portugal, being the one used by the deaf community. It is estimated that 30.000 people in Portugal are deaf, according to [2], and with the surrounding community, such as teachers, relatives, among others, it reaches around 110.000 people. This represents only 1% of the national population, according to the values of Censos 2021 (Portugal's Census) [3]. Looking at these numbers, it is easy to understand the difficulties deaf people go through when trying to communicate with the general population, since most people do not know anything about SL or how to use it. To overcome this difficulty, there is the possibility of using mobile devices that offer translation capabilities between both languages.

The work reported in this paper was supported by FEDER, Program Compete2020, through the POCI-01-0247-FEDER-068605 project.

This paper proposes a system that translates European Portuguese Language to LGP and uses a 3D avatar to show deaf people an understandable SL, through a mobile application. The system should receive a text or an audio from the user, and translate it to LGP, using a textual translation system. The textual representation of a sign is known as a gloss (the plural being glosses) and the system outputs a sequence of glosses. With the translated text, the system proceeds to get the correct animations from a database, sequencing them for the user to see. These animations may be captured with a motion-capture (MoCap) suit to speed up the process of capturing the animations to create a glosses dictionary.

II. Related Works

Since there are so many different SLs, it is challenging to develop a universal system. As a result, the majority of systems implemented up to this point have only addressed one SL. When converting spoken language to SL, it is necessary to establish a system that enables a deaf person to view and comprehend an animation, without reading the pre-translated sentence. Prior to performing this, a translation from a written sentence (in a spoken language) to glosses is required because spoken languages and SLs have different grammar and structural elements. The original sentence is analysed by a system that then transforms it into glosses, that make sense in the structure of the intended SL. The Gloss Notation consists in a text label for each sign in a SL [14]. The glosses are written in all uppercase letters and in the source language (e.g., LGP is written in European Portuguese). The HamNoSys system is the most widely used and is a pictograph notation system. Containing around 330 symbols that represent motions of a sign, this system works better with multiple SLs, since it does not depend on spoken languages. However, it is harder to learn by users. It is worth noting that none of the notation systems are accepted as standard written forms of a SL [15]. Several works related with translating spoken language into SL have been done. Some focus on translation into Glosses or other notation systems, others focus on the representation of SL via avatars or videos, while other focus in both parts. In Table I we can verify a summary of works, focused on those containing the translation and representation modules. On the translation column, there are two main approaches, the

Figure A.1: A Translation System from European Portuguese to Portuguese Sign Language [61]

Capturing and Processing Sign Animations to a Portuguese Sign Language 3D Avatar

Bruno Ribeiro[1], Duarte Dias[1], Pedro Miguel Faria[1], Luís Romero[1]

[1] ADIT-LAB, Instituto Politécnico de Viana do Castelo, Viana do Castelo, Portugal

(bfiliperibeiro,duartedias)@ipvc.pt (pfaria,romero)@estg.ipvc.pt

2023 30th International Conference on Systems, Signals and Image Processing (IWSSIP) | 979-8-3503-3729-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/IWSSIP58668.2023.10180233

***Abstract*—Usually, the deaf community faces enormous difficulties in communicating with the hearing community. Unfortunately, there are few systems available to the public to assist in the communication between the deaf community and the hearing community. Some countries, including Portugal, are trying to develop systems to mitigate this problem. Thus, in this paper, it is proposed a system that captures signs, using motion capture systems and produces the needed animations, represented by a 3D Avatar, so that a deaf user may understand them. This first implementation consists of a component, receiving text in Glosses format, from the Portuguese Sign Language, getting the corresponding animations from a database, and animating a 3D Avatar with these animations. The work being carried out is part of a larger system, that includes a user interface and another component that translates Portuguese Spoken Language to Portuguese Sign Language, through a Neural Network. The system aims to assist deaf people in communicating in all scenarios of everyday life. The first results show great potential for the deaf community to be able to use the proposed system.**

***Index Terms*—Deaf, Assistive Technology, Portuguese Sign Language, Avatar, Animation, 3D Models, 3D, Glosses, Motion Capture, State Machine, LGP.**

I. Introduction

Portuguese Sign Language (LGP - *Língua Gestual Portuguesa*) is one of the three official national languages, which is predominately used by the deaf community to communicate with each other. According to the Portuguese Deaf Association [1], this population is thought to consist of around 30.000 people in Portugal, or about 110 thousand users of LGP nationwide when taking into account everyone in the surrounding community, including friends, teachers, technicians, etc. According to the results of the 2021 Census, this represents only around 1% of the national population. However, in the pockets of almost everyone, there may be a solution to mitigate these difficulties. In the same way that it is already perfectly normal to use smartphones for translations between Portuguese Spoken language and other languages, why not also use them for translation between spoken language and sign language?

This paper proposes a system that uses a 3D avatar to show deaf people understandable sign language through a mobile application. The system should receive a text that lists a sequence of signs, which is generated by a neural network responsible for the translation. This written representation of a sign is known as a gloss (the plural being glosses) and should be in all upper case letters. With the translated text, the system proceeds to get the correct animations from a database and animate them for the user to see them. These animations were captured with several motion-capture (MoCap) suits to speed up the process of capturing the animations for the gloss dictionary and to give a more realistic look to the signs.

This paper is organized into five sections: the first is the current introduction, followed by a section to present related work, in which some research projects are presented. Section three presents the global architecture of the developed system. Section four presents a solution for the acquisition of the animations required by the system. Section five will explore the way the Unity software processes animations and how the transitions between them work. Section six will discuss the results of some captured animations, validated by LGP speakers. Finally, conclusions and future work are presented, as well as some references used to do this work.

The work reported in this paper was supported by FEDER, Program Compete2020, through the POCI-01-0247-FEDER-068605 project.

II. Related Works

Since there are so many sign languages, it is difficult to create a universal system and, consequently, most solutions implemented so far focus only on one sign language. When translating spoken language to sign language, is required to have a system that allows a deaf person to watch and understand a sequence of glosses, without reading the pre-translated sentence. Before doing this, a translation from a written sentence (in a spoken language) to glosses is necessary since the grammar and structure of spoken languages and sign languages are different. This translation is made through a system that first processes the original sentence, and then converts it into glosses that make sense in the targeted sign language's structure. These glosses define the order that the glosses should be shown so that a deaf person can watch and understand the original sentence without reading it.

In this context, the use of 3D avatars is the most common implementation nowadays, since the translation requires to have a "person" able to do a translated gesture. An Avatar is a model made in 3D software with an appearance similar to a human being, that has a Rigg, which is a skeleton that gives the model the ability to be animated, and a mesh, a "skin" of polygons that covers the skeleton, giving it a human appearance. Most solutions that translate spoken language into

Figure A.2: Capturing and Processing Sign Animations to a Portuguese Sign Language 3D Avatar [62]

Appendix B

Code

Listing B.1: Maya automation script

```
import maya.cmds as cmds
import maya.mel as mel
mel.eval("FBXExportSplitAnimationIntoTakes -clear;")

cmds.clearCache( all=True )

# Get the currently selected object
selected_objects = cmds.ls(selection=True)

# Check if there is exactly one object selected
if len(selected_objects) != 1:
    cmds.error(selected_objects)
else:
    # Get the selected object
    selected_object = selected_objects[0]



cmds.select(selected_objects, r= True)
startingFrames=[90 ,195,290,375,460,605,705,800,880,960
    ,1060,1150,1250,1345,1450,1545,1750,1850,1965,2080,2190,2285,2375,
2470,2560,2700,2810,2915,3020,3125,3235,3325,3435,3540,3635,3730,
3830,3920,4020,4115,4200,4315,4420,4510,4590,4700,4790,4880,5000,
```

```
5100,5210,5310,5420,5610,5740,5840,5940,6050,6140,6240,6420,6540,
6630,6720,6830,6920,7020,7120,7210,7295,7400]
endingFrames =[160,265,350,435,550,670,765,855,930,1030,1125,1215,1300,1400,
1520,1600,1830,1930,2025,2150,2255,2350,2440,2540,2650,2770,
2875,2970,3080,3195,3300,3400,3500,3600,3710,3800,3900,3985,4090,
4175,4280,4390,4480,4560,4670,4760,4840,4960,5060,5160,5280,5380,
5490,5700,5800,5900,6015,6110,6210,6320,6500,6600,6700,6800,6900,
6990,7090,7180,7270,7380,7490]
fbxnames["bom_dia","senhor","mulher","minha","ementa","poder_interrogacao",
"trazer_interrogacao","trazer","sim","por_favor","aqui","escolher","tu",
"a_vontade","nome","meu","dizer_eu","sardinha","fresca","epoca",
"agora","esta","grelhar","com","azeite","deliciosa","muito_bom",
"dose","poder","sugerir","meia","nossa","substancial","bebida",
"vinho","casa","gostar","provar","tinto","branco","preferir",
"muito_bem","ate_ja","obrigado","ananas","haver","nao","semana",
"passada","esgotar","desde","nao_haver","uva","gramas","querer",
"amanha","preta","haver","mais_interrogacao","cereja","conta_valor",
"apenas","esperar","preco","baixar","dinheiro","macas","ja",
"chegar","kg","euro"]

necessaryName="face@"
caraFrameDiff= 0
print(len(startingFrames))
print(len(endingFrames))
print(len(fbxnames))
for i in range(len(startingFrames)):
    print(i)
    cmds.playbackOptions(edit=True,animationStartTime=(startingFrames[i]+caraFrameDiff))
    cmds.playbackOptions(edit=True,animationEndTime=(endingFrames[i])+caraFrameDiff)
    cmds.playbackOptions(edit=True,minTime=(startingFrames[i])+caraFrameDiff)
    cmds.playbackOptions(edit=True,maxTime=(endingFrames[i])+caraFrameDiff)
    mel.eval('FBXExportBakeComplexAnimation -v true ')
    mel.eval('FBXExport -f "C:/Users/rp/desktop/fbx/face/dialogo turismo 3
    e frutaria/'+ necessaryName +fbxnames[i] +'.fbx" -s')
```

Listing B.2: Maya automation script for T-pose

```
string
    $b[]={"RL_BoneRoot","CC_Base_Hip","CC_Base_Pelvis","CC_Base_Waist","CC_Base_Spine01",
"CC_Base_Spine02","CC_Base_NeckTwist01","CC_Base_NeckTwist02","CC_Base_Head",
"CC_Base_FacialBone","CC_Base_JawRoot","CC_Base_Tongue01","CC_Base_Tongue02",
"CC_Base_Tongue03","CC_Base_Teeth02","CC_Base_R_Eye","CC_Base_L_Eye",
"CC_Base_UpperJaw","CC_Base_Teeth01",
"CC_Base_L_Clavicle","CC_Base_L_Upperarm","CC_Base_L_Forearm",
"CC_Base_L_ForearmTwist01","CC_Base_L_ForearmTwist02",
"CC_Base_L_ElbowShareBone","CC_Base_L_Hand","CC_Base_L_Pinky1",
"CC_Base_L_Pinky2","CC_Base_L_Pinky3","CC_Base_L_Ring1",
"CC_Base_L_Ring2","CC_Base_L_Ring3","CC_Base_L_Mid1",
"CC_Base_L_Mid2","CC_Base_L_Mid3","CC_Base_L_Index1","CC_Base_L_Index2",
"CC_Base_L_Index3","CC_Base_L_Thumb1","CC_Base_L_Thumb2",
"CC_Base_L_Thumb3","CC_Base_L_UpperarmTwist01","CC_Base_L_UpperarmTwist02",
"CC_Base_L_RibsTwist","CC_Base_L_Breast","CC_Base_R_RibsTwist",
"CC_Base_R_Breast","CC_Base_R_Clavicle",
"CC_Base_R_Upperarm","CC_Base_R_Forearm","CC_Base_R_ElbowShareBone",
"CC_Base_R_ForearmTwist01","CC_Base_R_ForearmTwist02",
"CC_Base_R_Hand","CC_Base_R_Ring1","CC_Base_R_Ring2",
"CC_Base_R_Ring3","CC_Base_R_Mid1","CC_Base_R_Mid2",
"CC_Base_R_Mid3","CC_Base_R_Thumb1",
"CC_Base_R_Thumb2","CC_Base_R_Thumb3",
"CC_Base_R_Index1","CC_Base_R_Index2","CC_Base_R_Index3","CC_Base_R_Pinky1",
"CC_Base_R_Pinky2","CC_Base_R_Pinky3","CC_Base_R_UpperarmTwist01",
"CC_Base_R_UpperarmTwist02"};


float
    $rotateValues[]={0,0,0,89.99993896,0.04132473469,0.0001892731962,4.300259113,
0.003677089466,0.3294214308,16.08040428,-0.05783012137,0.2847456634,
-21.08679962,-0.1737579405,-0.2100645304,-1.558607817,0.1717923284,
0.05706044286,12.90053844,-0.6039410233,-0.4323600233,
6.220348358,0.6798685193,0.861243546,-12.56467628,0.02754488587,
-0.6278169751,90.00000763,3.415094397,90.00000763,-0.008202859201,
0.001312406617,89.99750519,
```

0.1351131797,-0.1172541603,7.771546841,
0.006728072651,0.2221876085,10.57964611,-0.0002704041835,
1.143534973e-05,0.01989206858,-179.9906311,-0.01127020083,
0.02291015908,-89.99003601,-90,-89.99003601,
-89.99002838,-90,-89.99002838,-0.0002262317867,
-0.00481121894,90.00059509,179.9987488,0.003909145948,
0.02019201405,-16.02268982,6.762303829,-88.77315521,
14.54679966,0.8021324277,-0.2037478536,0.8558676839,
0.009431066923,-0.256344825,-1.85470617e-05,-1.111321035e-10,
-0.0006866208278,0.0005059725954,
-2.947621924,0.0006675709155,0.3898548186,0.04851454124,
-0.06267192215,-0.6647051573,-0.1711040288,-0.9495327473,
0.9120881557,0.5459570885,-3.598847389,
1.157283068,-0.5759678483,-3.137159824,1.643995643,
1.365989447,11.97604847,0.4256863892,0.1575044543,
-0.5564429164,0.4926328659,0.1979444027,-6.289815903,
2.333194733,0.2330210358,2.91206193,1.66437459,
0.1310482472,-2.186019659,1.272958875,0.272562027,
-3.994382858,0.8773228526,0.4976036847,4.037228107,0.5652157664,
-0.0602523908,-1.435972929,0.7416988611,0.651330173,
-4.992014408,-2.289025545,-0.06724517047,-0.3124500811,
49.41002274,-7.205043316,-11.13240051,-21.40246582,
0.2697051466,7.128509521,0.1402158588,-0.4185310304,
4.442451,0.0003350652405,-9.418246918,1.942332347,
0.0001500894869,-1.946461221,1.486100155,83.38825989,
-0.001387835247,179.8074951,0.3993059993,0.02454182133,
0.001581050688,83.38827515,-0.001336395391,179.8074493,
0.3993366063,0.02454216965,0.001580737997,
-16.01123238,-6.733486652,88.50652313,14.6105814,
-0.8844421506,0.4955264628,0.7033563256,0.01508807018,
0.1471126825,0.3898358941,-0.04850429669,
0.0626456365,-9.171419151,5.4610389131,0.0006823251024,
0.0005061480915,2.947937894,-0.0006674109609,
-0.5841748714,0.3175287247,0.9312614799,0.4259839654,-0.1738752723,
0.5720202327,0.4786630571,-0.2929253876,6.116501808,2.289946318,

```
-0.1845304519,-2.808357716,1.671956658,-0.1725800037,
2.1816082,1.206059575,-0.3313822746,3.882966995,0.9274022579,-0.4372599125,
-3.891358376,49.36967087,6.986506939,11.05290508,-21.41380692,
-0.1218233928,-7.151700974,0.1471791267,0.4545444846,-4.44709301,
0.5842584968,0.001944287447,1.566385865,0.7137252688,
-0.6929427385,4.862435341,-2.005750418,0.05104552954,
-0.07366282493,0.9289814234,-0.5588511825,3.593444347,
1.04165256,0.6002991199,1.398695111,1.703966618,-1.280348182,-11.87012291,
0.0003376007953,1.331361364,-2.697387754,0.0001535420743,1.555095115,-1.160599004};

int $counter=0;

for($l in $b){
    print($l);
    setAttr($l +".rotateX", $rotateValues[$counter]);
    setKeyframe ($l+".rx");
    setAttr(($l +".rotateY"), $rotateValues[$counter+1]);
    setKeyframe ($l+".ry");
    setAttr(($l +".rotateZ"), $rotateValues[$counter+2]);
    setKeyframe ($l+".rz");
    $counter +=3;
}
```

Listing B.3: Unity logic control script

```
    using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.UI;
using System.Linq;
using Monobehaviours;

public class Controls : MonoBehaviour
```

```
{
    public string queryUrl;
    public string bundleUrl;
    [SerializeField] private Text currentClip;
    [SerializeField] private Animator animator;
    [SerializeField] private Animator animatorFace;
    [SerializeField] private AnimatorOverrider animatorOverrider;
    [SerializeField] private AnimatorOverrider animatorFaceOverrider;
    [SerializeField] private AnimatorOverrideController animatorOverrideCont;
    IDictionary<string, string> animNameToName = new Dictionary<string,
        string>();
    bool crRunning = false;
    //[SerializeField] private AnimatorOverrider overriderFace;
    //[SerializeField] private GameObject myRecordingSettings;

    protected AnimationClip[] fullAnimationsList;
    private List<Clip> neededClipsList, neededFaceClipsList;

    int bundleCount,
        clipUpdated = 0,
        lastUpdated = -1;
    string[] bundleArray, animsArray, animNames;

    void Start()
    {
        queryUrl = "https://ivling.estg.ipvc.pt:9900/traduz?texto=";
        bundleUrl = "https://ivling.estg.ipvc.pt:9900/static/AssetBundles/";
        neededClipsList = new List<Clip>();
        neededFaceClipsList = new List<Clip>();
        //test2();
#if !UNITY_EDITOR && UNITY_WEBGL
        WebGLInput.captureAllKeyboardInput = false;
#endif
    }
```

```
public void test2()
{
    ReceiveMessage("No proximo sabado vou a uma festa no Porto");
}

public void test()
{
    ReceiveMessage("Eu vou ler um livro");
}

public void ReceiveMessage(string s)
{
    clipUpdated = 0;
    neededClipsList.Clear();
    //neededFaceClipsList.Clear();
    lastUpdated = -1;
    animNameToName.Clear();
    StartCoroutine(WaitForDownloads(s));
}

IEnumerator WaitForDownloads(string s)
{
    crRunning = true;

    yield return StartCoroutine(GetBundles(s));

    yield return StartCoroutine(BDConnection());
    crRunning = false;
}

IEnumerator GetBundles(String text)
{
    String queryUrl1 = queryUrl + text;
```

```
using (UnityWebRequest web = UnityWebRequest.Get(queryUrl1))
{
    var cert = new CertificateOverride();
    web.certificateHandler = cert;
    yield return web.SendWebRequest();
    cert.Dispose();

    switch (web.result)
    {
        case UnityWebRequest.Result.ConnectionError:
        case UnityWebRequest.Result.DataProcessingError:
            Debug.LogError("Error: " + web.error);
            break;
        case UnityWebRequest.Result.ProtocolError:
            Debug.LogError("HTTP Error: " + web.error);
            break;
        case UnityWebRequest.Result.Success:
            Debug.Log("Recebeu da api: " + web.downloadHandler.text);

            //PARSING THE QUERY RESULT
            String query = web.downloadHandler.text;
            var message = JsonUtility.FromJson<Message>(query);

            // gets the number of bundles and animations
            // and stores them in new arrays
            bundleCount = message.traducoes.bundles_anims.Count;
            int animsCount = message.traducoes.anims.Count;
            bundleArray = new string[bundleCount];
            animsArray = new string[animsCount];
            animNames = new string[animsCount];

            for (int i = 0; i < bundleCount; i++)
            {
                bundleArray[i] = message.traducoes.bundles_anims[i];
            }
```

```
                for (int i = 0; i < animsCount; i++)
                {
                    animsArray[i] = message.traducoes.anims[i].ToLower();
                    animNames[i] = message.traducoes.anim_names[i];
                    if (!animNameToName.ContainsKey(animsArray[i]))
                        animNameToName.Add(animsArray[i], animNames[i]);
                }

                break;
            }
        }
    }

    IEnumerator BDConnection()
    {
        int totalLength = bundleCount;
        for (int i = 0; i < totalLength; i++)
        {
            string bundleName = bundleArray[i];
            var bundleUrl_atual = bundleUrl + bundleName;
            //Debug.Log(bundleUrl_atual);
            using (UnityWebRequest web =
                UnityWebRequestAssetBundle.GetAssetBundle(bundleUrl_atual))
            {
                var cert = new CertificateOverride();
                web.certificateHandler = cert;
                yield return web.SendWebRequest();
                cert.Dispose();

                if (web.result != UnityWebRequest.Result.Success)
                {
                    Debug.Log(web.error);
                }
```

```
            if (web.result != UnityWebRequest.Result.Success)
            {
                Debug.LogError("Failed to download AssetBundle from RP!");
                yield break;
            }
            else
            {
                Debug.Log("Fez download de: " + bundleName);
                AssetBundle bundle =
                    DownloadHandlerAssetBundle.GetContent(web);
                fullAnimationsList = bundle.LoadAllAssets<AnimationClip>();
                if (bundleName.StartsWith("face"))
                {
                    //animationScript(animsArray, true);
                }
                else
                {
                    animationScript(animsArray, false);
                }
                bundle.Unload(false);
            }
        }
    }
}

void Update()
{
    if (animator.GetCurrentAnimatorStateInfo(0).IsName("Base Layer.State0")
        && lastUpdated != 0 && clipUpdated < neededClipsList.Count() &&
        !crRunning)
    {
        Clip c = neededClipsList.Find(i => i.state == clipUpdated);
        //Clip fc = neededFaceClipsList.Find(i => i.state == clipUpdated);
        animator.SetBool("St1toSt2", false);
        animator.SetBool("St2toSt1", false);
```

```
        //animatorFace.SetBool("St1toSt2", false);
        //animatorFace.SetBool("St2toSt1", false);

        animator.SetTrigger("startAnimation");
        //animatorFace.SetTrigger("startAnimation");

        if (c != null)
        {
            animatorOverrideCont["State1"] = c.clip;
            //animatorOverrideCont["Face1"] = fc.clip;
            animatorOverrider.SetAnimations(animatorOverrideCont);
            //animatorFaceOverrider.SetAnimations(animatorOverrideCont);
            Debug.Log(c.clip);
        }

        clipUpdated++;
        lastUpdated = 0;

    }
    else if (animator.GetCurrentAnimatorStateInfo(0).IsName("Base
        Layer.State1")
        //&& animatorFace.GetCurrentAnimatorStateInfo(1).IsName("Face
            Layer.Face1")
        && lastUpdated != 1
        && clipUpdated < neededClipsList.Count())
    {
        Clip c = neededClipsList.Find(i => i.state == clipUpdated);
        //Clip fc = neededFaceClipsList.Find(i => i.state == clipUpdated);

        animator.SetBool("St1toSt2", true);
        animator.SetBool("St2toSt1", false);
        //animatorFace.SetBool("St1toSt2", true);
        //animatorFace.SetBool("St2toSt1", false);

        if (c != null)
```

```
        {
            animatorOverrideCont["State2"] = c.clip;
            //animatorOverrideCont["Face2"] = fc.clip;
            animatorOverrider.SetAnimations(animatorOverrideCont);
            //animatorFaceOverrider.SetAnimations(animatorOverrideCont);
        }

        clipUpdated++;
        lastUpdated = 1;
    }
    else if (animator.GetCurrentAnimatorStateInfo(0).IsName("Base
        Layer.State2")
        //&& animatorFace.GetCurrentAnimatorStateInfo(1).IsName("Face
            Layer.Face2")
        && lastUpdated != 2
        && clipUpdated < neededClipsList.Count())
    {
        Clip c = neededClipsList.Find(i => i.state == clipUpdated);
        //Clip fc = neededFaceClipsList.Find(i => i.state == clipUpdated);

        animator.SetBool("St1toSt2", false);
        animator.SetBool("St2toSt1", true);
        //animatorFace.SetBool("St1toSt2", false);
        //animatorFace.SetBool("St2toSt1", true);

        if (c != null)
        {
            animatorOverrideCont["State1"] = c.clip;
            //animatorOverrideCont["Face1"] = fc.clip;
            animatorOverrider.SetAnimations(animatorOverrideCont);
            //
            //animatorFaceOverrider.SetAnimations(animatorOverrideCont);
        }
```

```
            clipUpdated++;
            lastUpdated = 2;
        }

        if (animator.GetCurrentAnimatorClipInfo(0).Length > 0)
        {
            string name = animator.GetCurrentAnimatorClipInfo(0)[0].clip.name;
            if (name != "idle" && name != "__")
                currentClip.text = animNameToName[name];
            else
                currentClip.text = "";
        }
    }

    public void animationScript(String[] animsArray, bool face)
    {
        if (!face)
        {
            foreach (AnimationClip a in fullAnimationsList)
            {
                int[] animationNames = FindAllIndexOf(animsArray,
                    a.name.ToLower());
                for (int i = 0; i < animationNames.Length; i++)
                {
                    int index = animationNames[i];
                    string statename = "State" + (index + 1);

                    neededClipsList.Add(new Clip(index, a));

                    //var state = baseLayer.AddState(statename);
                    //state.motion = a;

                    //Debug.Log(statename + " -> " + a.name);
                }
            }
```

```
    }
    else
    {
        foreach (AnimationClip a in fullAnimationsList)
        {
            int[] animationNames = FindAllIndexOf(animsArray,
                a.name.ToLower());
            for (int i = 0; i < animationNames.Length; i++)
            {
                int index = animationNames[i];
                string statenameface = "FaceState" + (index + 1);

                neededFaceClipsList.Add(new Clip(index, a));

                //var state = baseLayer.AddState(statename);
                //state.motion = a;

                Debug.Log(statenameface + " -> " + a.name);
            }
        }
    }
}

private int ContainsAnimation(AnimationClip[] bundle, String name)
{
    int i = 0;
    foreach (AnimationClip clip in bundle)
    {
        if (clip.name == name)
        {
            return i;
        }
        else
        {
            i++;
```

```
            }
        }

        return -1;
    }

    public int[] FindAllIndexOf<T>(IEnumerable<T> values, T val)
    {
        return values.Select((b, i) => object.Equals(b, val) ? i : -1).Where(i =>
            i != -1).ToArray();
    }

    public void Repeat()
    {
        clipUpdated = 0;
        lastUpdated = -1;
        animator.SetTrigger("startAnimation");
    }

    public void SwitchCaptions()
    {
        currentClip.gameObject.SetActive(!currentClip.gameObject.activeInHierarchy);
    }

    public void SliderChange(float value)
    {
        Time.timeScale = value;
    }
}
```

Listing B.4: NLA Ordering script

```
import bge
import bpy
import time
```

```
main_object= bpy.data.objects['Tpose_teste']
bpy.ops.object.select_all(action='DESELECT')

bpy.data.objects['Tpose_teste'].select_set(True)

starting_time= time.time()

# gets the nla track associated with the object
tracks = main_object.animation_data.nla_tracks

main_track=tracks["main_track"]
sub_track=tracks["sub_track"]

animation_list=["bom_dia","Amanha_tpose","cha_tpose","qestupido","agua
    tpose","agora_t"]

def clear_main_track():

    stringToSplit= str(main_track.strips)
    mainTrackx=stringToSplit.split("[")
    mainTracky=mainTrackx[1].split("]")
    mainTrackz=int(mainTracky[0])

    for i in range(mainTrackz,1,-1):
        main_track.strips.remove(main_track.strips[i-1])

def clear_sub_track():

    stringToSplit= str(sub_track.strips)
    subTrackx=stringToSplit.split("[")
    subTracky=subTrackx[1].split("]")
    subTrackz=int(subTracky[0])

    for i in range(subTrackz,1,-1):
```

```
        print(i)
        sub_track.strips.remove(sub_track.strips[i-1])

#metodo para adicionar uma nova action
#tracks[2].strips.new(action.name,int(tracks[2].strips[-1].frame_end+1),action)

def order_actions(animations):
    counter=0
    tweaking=12.0
    strip_list=[]
    for a in animations:
        if(counter % 2) ==0:
            print(a)
            action= bpy.data.actions[a]
            #print(action1)
            #need to check if its the first animation
            if counter == 0:
                #criar nova strip com a acao que contem a animacao desejada
                strip = main_track.strips.new(action.name,0,action)
                #defenir que comece e acaba mais cedo de modo a nao mostrar as T
                    pose
                strip.action_frame_start=3.0
                strip.action_frame_end -= 4.0
                # garantir que nao congela o frame final da animacao
                strip.extrapolation='NOTHING'
                #adicionar a lista de strips usadas de modo a saber a sua posicao
                    na timeline de modo a poder colocar os seguintes no local certo
                strip_list.append(strip)
            else:
                strip =
                    main_track.strips.new(action.name,int(strip_list[int(counter-1)]
                .frame_end-tweaking),action)
                strip.action_frame_start=3.0
                strip.action_frame_end -= 4.0
                strip.extrapolation='NOTHING'
```

```
                strip_list.append(strip)
            #para garantir que nao exista tpose no inicio ou fim podes modificar
                os frames

            #print(strip_list[0].frame_end)
            counter+=1
        else:
            action= bpy.data.actions[a]
            strip = sub_track.strips.new(action.name,
            int(strip_list[int(counter-1)].frame_end-tweaking),action)
            \#para garantir que nao exista tpose no inicio ou fim podes modificar
                os frames
            strip.action_frame_start=3.0
            strip.action_frame_end -= 4.0
            strip.blend_in=tweaking
            strip.blend_out=tweaking
            strip.extrapolation='NOTHING'
            strip_list.append(strip)
            counter+=1

    bpy.context.scene.frame_end = int(strip_list[counter-1].frame_end)
    print("SCRIPT RUN COMPLETE")


def animationListPrep(list):

    finalList= list.split(',')
    for a in finalList:
        print(a)

    return finalList



clear_sub_track()
```

```
clear_main_track()
textToAnimate= bpy.data.objects['Insert Text'].data.body
textTime= bpy.data.objects['Time'].data.body;



cont = bge.logic.getCurrentController()
own = cont.owner

sensorMouseLeftClick = cont.sensors['MouseLeftClick']
sensorMouseOver= cont.sensors['MouseOver']

directory= bpy.data.objects['Directory Text'].data.body

if sensorMouseLeftClick.positive and sensorMouseOver.positive:
        print("a primir no rato")
        order_actions(animationListPrep(textToAnimate))

        bpy.ops.export_scene.fbx(filepath = directory, use_selection =
            True,bake_anim_use_all_bones=True,bake_anim_use_nla_strips=False,
        bake_anim_use_all_actions=False)

        print("Final time=")
        print(time.time()-starting_time)
        bpy.data.objects['Time'].data.body= "Time of export in seconds: " +
            str(time.time()-starting_time)
```

Listing B.5: Blender Directory Interface interaction code

```
import bge
import bpy



controllerGlossDirectoryText = bge.logic.getCurrentController()# THIS
own = controllerGlossDirectoryText.owner
```

```
textToAnimate= bpy.data.objects['Directory Text'].data.body
sensorGlosses = controllerGlossDirectoryText.sensors['KeyboardDirectory']
sensorTextBoxDirectoryOver=
    controllerGlossDirectoryText.sensors['MouseOverDirectory']
sensorTextBoxDirectoryPress=
    controllerGlossDirectoryText.sensors['MousePressDirectory']


def main():

    #global activeBox
    print(own['activeDirectory'])
    #sensorGlosses = controllerGlossTextBox.sensors['Keyboard']
    #JUST_ACTIVATED = bge.logic.KX_INPUT_JUST_ACTIVATED

    if sensorTextBoxDirectoryPress.positive and
        sensorTextBoxDirectoryOver.positive:
        own['activeDirectory']=True

    if sensorTextBoxDirectoryPress.positive and
        sensorTextBoxDirectoryOver.positive==False:
        own['activeDirectory']= False


    if sensorGlosses.positive and own['activeDirectory']==True:
        print(bge.logic.keyboard.events[55])
        #print(bge.events.CAPSLOCKKEY)
        if bge.logic.keyboard.events[55]==1:
            print(bge.events.EventToCharacter(sensorGlosses.events[0][0],True))
            bpy.data.objects['Directory Text'].data.body +=
            bge.events.EventToCharacter(sensorGlosses.events[0][0],True)
        else:
            print(bge.events.EventToCharacter(sensorGlosses.events[0][0],False))
            bpy.data.objects['Directory Text'].data.body +=
            bge.events.EventToCharacter(sensorGlosses.events[0][0],False)
```

```
        # Code to delete keys
        if sensorGlosses.getKeyStatus(59)==2:
            bpy.data.objects['Directory Text'].data.body =
            bpy.data.objects['Directory Text'].data.body[:-1]



main()
```

Listing B.6: Blender Gloss Interface interaction code

```
import bge
import bpy



controllerGlossTextBox = bge.logic.getCurrentController()
own = controllerGlossTextBox.owner
textToAnimate= bpy.data.objects['Insert Text'].data.body
sensorGlosses = controllerGlossTextBox.sensors['Keyboard']


my_property=own['active']


sensorTextBoxGlossOver= controllerGlossTextBox.sensors['MouseOverGlosses']
sensorTextBoxGlossPress= controllerGlossTextBox.sensors['MouseLecftClickGlosses']


def main():


    #global activeBox
    #print(own['active'])
    #sensorGlosses = controllerGlossTextBox.sensors['Keyboard']
    #JUST_ACTIVATED = bge.logic.KX_INPUT_JUST_ACTIVATED


    if sensorTextBoxGlossPress.positive and sensorTextBoxGlossOver.positive:
        own['active']=True
```

```
    if sensorTextBoxGlossPress.positive and
        sensorTextBoxGlossOver.positive==False:
        own['active']= False


    if sensorGlosses.positive and own['active']==True:
        #print(bge.logic.keyboard.events[55])
        #print(bge.events.CAPSLOCKKEY)
        if bge.logic.keyboard.events[55]==1:
            #print(bge.events.EventToCharacter(sensorGlosses.events[0][0],True))
            bpy.data.objects['Insert Text'].data.body +=
            bge.events.EventToCharacter(sensorGlosses.events[0][0],True)
        else:
            #print(bge.events.EventToCharacter(sensorGlosses.events[0][0],False))
            bpy.data.objects['Insert Text'].data.body +=
            bge.events.EventToCharacter(sensorGlosses.events[0][0],False)
        # Codigo para apagar texto
        if sensorGlosses.getKeyStatus(59)==2:
            bpy.data.objects['Insert Text'].data.body =
            bpy.data.objects['Insert Text'].data.body[:-1]



main()
```