INDUSTRIAL ORIENTED MINI PROJECT

On

DYNAMIC SIGN LANGUAGE SYNTHESIS

Submitted in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

By

Krithi Chippada- 22261A1236

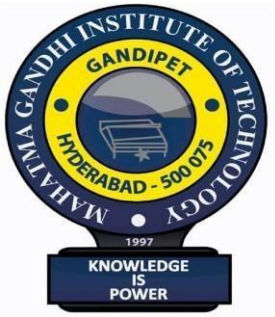
Satvik Dabbara – 22261A1217

Under the guidance of

Dr. U. Chaitanya

Assistant Professor

Department of IT



DEPARTMENT OF INFORMATION TECHNOLOGY

MAHATMA GANDHI INSTITUTE OF TECHNOLOGY

(AUTONOMOUS)

(Affiliated to JNTUH, Hyderabad; Eight UG Programs Accredited by NBA; Accredited by NAAC with 'A++' Grade)

Gandipet, Hyderabad, Telangana, Chaitanya Bharati (P.O), Ranga Reddy District, Hyderabad– 500075, Telangana

2024-2025

CERTIFICATE

This is to certify that the **Industrial Oriented Mini Project** entitled **DYNAMIC SIGN LANGUAGE SYNTHESIS** submitted by **Krithi Chippada (22261A1236), Satvik Dabbara (22261A1217)** in partial fulfillment of the requirements for the Award of the Degree of Bachelor of Technology in Information Technology as specialization is a record of the bona fide work carried out under the supervision of **Dr. U. Chaitanya** , and this has not been submitted to any other University or Institute for the award of any degree or diploma.

Project Guide:

Dr. U. Chaitanya
Assistant Professor
Dept. of IT

Project Coordinator:

Dr. U. Chaitanya
Assistant Professor
Dept. of IT

EXTERNAL EXAMINAR

Dr. D. Vijaya Lakshmi
Professor and HOD
Dept. of IT

DECLARATION

We hear by declare that the **Industrial Oriented Mini Project** entitled **DYNAMIC SIGN LANGUAGE SYNTHESIS** is an original and bonafide work carried out by us as a part of fulfilment of Bachelor of Technology in Information Technology, Mahatma Gandhi Institute of Technology, Hyderabad, under the guidance of **Dr. U. Chaitanya, Assistant Professor**, Department of IT, MGIT.

No part of the project work is copied from books /journals/ internet and wherever the portion is taken, the same has been duly referred in the text. The report is based on the project work done entirely by us and not copied from any other source.

Krithi Chippada – 22261A1236

Satvik Dabbara – 22261A1217

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without introducing the people who made it possible and whose constant guidance and encouragement crowns all efforts with success. They have been a guiding light and source of inspiration towards the completion of the **Industrial Oriented Mini Project.**

We would like to express our sincere gratitude and indebtedness to our project guide **Dr. U. Chaitanya**, **Assistant Professor**, Dept. of IT, who has supported us throughout our project with immense patience and expertise.

We are also thankful to our honourable Principal of MGIT **Prof. G. Chandramohan Reddy** and **Dr. D. Vijaya Lakshmi**, Professor and HOD, Department of IT, for providing excellent infrastructure and a conducive atmosphere for completing this **Project Stage-I** successfully.

We are also extremely thankful to our IOMP Supervisor **Dr. U. Chaitanya**, Assistant Professor, Department of IT, and Senior Faculty **Mrs B. Meenakshi,** Assistant Professor Department of IT for their valuable suggestions and guidance throughout the course of this project.

We convey our heartfelt thanks to the lab staff for allowing us to use the required equipment whenever needed.

Finally, we would like to take this opportunity to thank our families for their support all through the work. We sincerely acknowledge and thank all those who gave directly or indirectly their support for completion of this work.

Krithi Chippada – 22261A1236

Satvik Dabbara -22261A1217

ABSTRACT

Sign language plays a crucial role in enabling effective communication for the Deaf and Hard-of-Hearing (DHH) community. However, many existing sign language interpretation systems rely heavily on static images or pre-recorded animations, which often fail to replicate the fluid and natural motions characteristic of real-life signing. This lack of realism can hinder effective communication and reduce user engagement, particularly in dynamic or conversational settings.

The objective of this project is to create a real-time system that translates multimodal inputs including spoken, and text inputs into synchronized and dynamic sign language animations. It converts speech into English text, and produces realistic sign language by using technologies like Deep Learning, speech recognition technology, natural language processing (NLP), and computer vision technology.

A key feature is its ability to maintain synchronization between the original input and the generated signing, preserving the natural flow of communication. This software helps reduce communication gaps and promotes accessibility for the DHH community online.

By enabling more natural, real-time, and accurate sign language translations, this project aims to bridge communication barriers and foster greater inclusivity and accessibility for the DHH community, especially in digital and educational environments

TABLE OF CONTENTS

Chapter No	Title	Page No
	CERTIFICATE	**I**
	DECLARATION	**ii**
	ACKNOWLEDGEMENT	**iii**
	ABSTRACT	**iv**
	TABLE OF CONTENT	**v**
	LIST OF FIGURES	**vi**
	LIST OF TABLES	**vii**
1	**INTRODUCTION**	**1**
	1.1 MOTIVATION	1
	1.2 PROBLEM STATEMENT	1
	1.3 EXISTING SYSTEM	1
	1.3.1 LIMITATIONS	2
	1.4 PROPOSED SYSTEM	2
	1.4.1 ADVANTAGES	2
	1.5 OBJECTIVES	3
	1.6 HARDWARE AND SOFTWARE REQUIREMENTS	3
2	**LITERATURE SURVEY**	**5**
3	**ANALYSIS AND DESIGN**	**8**
	3.1 MODULES	8
	3.2 ARCHITECTURE	9
	3.3 UML DIAGRAMS	10
	3.3.1 USE CASE DIAGRAM	10
	3.3.2 CLASS DIAGRAM	12
	3.3.3 ACTIVITY DIAGRAM	13
	3.3.4 SEQUENCE DIAGRAM	15
	3.3.5 COMPONENT DIAGRAM	16
	3.3.6 DEPLOYMENT DIAGRAM	17
	3.4 METHODOLOGY	18
4	**CODE AND IMPLEMENTATION**	**19**
	4.1 CODE	19
	4.2 IMPLEMENTATION	30
5	**TESTING**	**31**
6	**RESULTS**	**34**
7	**CONCLUSION AND FUTURE ENHANCEMENTS**	**40**
	7.1 CONCLUSION	40
	7.2 FUTURE ENHANCEMENTS	40
8	**REFERENCES**	

List of Figures

Fig. 3.2.1 Architecture of Dynamic Sign Language Synthesis 9
Fig. 3.3.1 Use Case Diagram 10
Fig. 3.3.2 Class Diagram 12
Fig. 3.3.3 Activity Diagram 13
Fig. 3.3.4 Sequence Diagram 15
Fig. 3.3.5 Component Diagram 16
Fig. 3.3.6 Deployment Diagram 17

LIST OF TABLES

Table 2.1 Literature Survey of Dynamic Sign Language Synthesis 5
Table 5.1 Test Cases of Dynamic Sign Language Synthesis 32

1. INTRODUCTION

1.1 MOTIVATION

In today's increasingly digital world, communication accessibility remains a critical challenge for the Deaf and Hard-of-Hearing (DHH) community. Sign language is the primary mode of communication for DHH individuals, enabling them to express ideas, emotions, and complex thoughts through visual and gestural cues. However, most public-facing and digital communication systems are still inaccessible to DHH users due to a lack of real-time, multimodal translation tools.

Existing sign language systems are typically limited to pre-recorded animations or static gesture libraries, which fail to capture the fluidity, expressiveness, and contextual depth of natural sign language. Moreover, these systems often require manual operation and support only offline or text-based input, severely restricting their usefulness in real-time scenarios such as education, emergency alerts, healthcare, and customer service.

This project aims to address these limitations by creating a real-time sign language translation system that processes spoken or text inputs and dynamically converts them into synchronized, lifelike sign animations using deep learning, computer vision, and natural language processing technologies.

1.2 PROBLEM STATEMENT

DHH people face profound communication challenges in their daily lives due to the lack of accessible, real-time sign language translation systems. Existing tools heavily rely on static signs or pre-recorded animations, which cannot accurately capture the fluidity, contextual depth, or expressiveness of natural sign language. Moreover, most systems support only one input format—typically text—and lack the ability to process spoken language audio.

This significantly limits the DHH community's access to essential services such as education, healthcare, public information, and digital media. There is a clear need for a dynamic, real-time, and multimodal solution that bridges this communication gap by converting spoken language and text inputs into synchronized, lifelike sign language animations that reflect the natural flow and expressiveness of human signing.

1.3 EXISTING SYSTEM

The existing sign language translation systems are limited in both functionality and scope, primarily relying on pre-recorded gesture videos or static animations. These systems lack the flexibility to interpret real-time input and often support only text-based translation, ignoring spoken language or audio extracted from video content. Most of them require manual intervention to operate and cannot dynamically adjust to different languages, users, or conversational contexts.

Furthermore, they fail to incorporate key linguistic elements such as natural sign grammar, facial expressions, and body movements, which are essential for accurately conveying meaning in sign language. The synchronization between input and sign output is typically weak, resulting in delayed or disjointed communication. Due to these shortcomings, such systems are

impractical for real-world applications where accurate, expressive, and real-time communication is essential—especially in education, public services, and emergency scenarios.

1.3.1 Limitations

- **Lack of Expressiveness in Current Models:**
  Most existing sign language synthesis systems fail to capture the full range of human expression. They often neglect crucial non-verbal cues such as facial expressions, body posture, and emotional tone, which are essential for conveying meaning in sign language. This results in robotic or unnatural animations that may be confusing or insufficient for effective communication.

- **Limited Multimodal Integration:**
  Current systems are generally unimodal, relying solely on a single input type—usually text. They lack the capability to process multiple modalities such as spoken language in addition to text, which limits their applicability in dynamic or real-time environments like live conversations, lectures, or emergency alerts. A more integrated multimodal approach—specifically combining speech and text inputs—can enable richer and more responsive sign language synthesis.

1.4 PROPOSED SYSTEM

The proposed system addresses these challenges by introducing a real-time, multimodal sign language translation framework that supports inputs in the form of speech and text only. The system first converts speech to text using speech recognition models like OpenAI's Whisper. The processed text is then translated into a sign language representation using SignWriting, a formal writing system for sign languages.

To generate lifelike animations, the system utilizes pose estimation tools such as OpenPose or MediaPipe to produce accurate skeletal keypoints, which are then animated to reflect natural signing movements, including hand shapes, orientations, and facial expressions. This output is synchronized with the original input for seamless communication.

The proposed system is designed to be scalable, supports multiple languages, and enhances accessibility for the Deaf and Hard-of-Hearing community by enabling more fluid, expressive, and real-time sign language synthesis for educational content, public information, and online media.

1.4.1 ADVANTAGES

1. **Enhanced Naturalness and Expressiveness:**
   By incorporating facial expressions, body posture, and hand gestures in the animation, the system produces more realistic and human-like sign language, significantly improving communication clarity and user engagement.

2. **Multimodal Input Processing:**
   The system supports both speech and text inputs, enabling accurate and context-aware sign language synthesis in dynamic scenarios. This multimodal capability makes it more versatile than traditional models that rely solely on text.

3. **Real-Time and Scalable Application:**
   Designed for real-time performance, the system is suitable for deployment in live settings such as virtual classrooms, customer service interfaces, and public announcements. Its scalable architecture also supports future expansion to different sign languages and dialects.

1.5 OBJECTIVES

- To develop a real-time system that translates spoken audio and text inputs into sign language animations.
- To generate dynamic and realistic sign language animations that closely mimic natural human signing, including hand gestures, facial expressions, and body movements.
- To ensure synchronization between the original input and the sign output, preserving the natural flow and timing of conversation.

1.6 HARDWARE AND SOFTWARE REQUIREMENTS

Hardware Requirements

1. **Processor (CPU):**
   A multi-core processor (e.g., Intel i7 or AMD Ryzen 7) to efficiently handle complex computations involved in speech recognition, natural language processing, and real-time sign language animation rendering.
2. **Graphics Processing Unit (GPU):**
   A dedicated GPU (e.g., NVIDIA GTX 1660 or higher) for accelerating deep learning model training and rendering high-quality 3D animations in real-time.
3. **RAM:**
   At least 16 GB of RAM to support smooth data processing, especially when running deep neural networks and handling multiple concurrent tasks.
4. **Storage:**
   Minimum 500 GB SSD for fast data storage and retrieval of speech/text datasets, pre-trained models, and animation assets.
5. **Microphone:**
   A quality microphone to capture clear speech input for accurate speech-to-text conversion.

Software Requirements

- **Programming Language**
  - Python: Used for implementing machine learning models, performing data processing, and integrating computer vision and NLP libraries.
- **Front-End**
  - HTML, CSS, JavaScript and TypeScript: Used for building the user interface of the web application for input/output interaction and visualization.
- **Back-End**
  - Node.js: Handles API creation, server-side logic, database communication, and integration with machine learning modules.
- **Cloud Platform**
  - Firebase: Used for real-time database, user authentication, hosting, and cloud storage functionalities.
- **Dataset**
  - SignBank Dataset: A comprehensive dataset containing standardized sign language glosses and translations used to train and validate the translation models.
- **Image Processing Libraries**
  - OpenCV: Used for basic image processing, frame extraction from video, and preprocessing for pose detection.
  - OpenPose: Used for detecting body, hand, and face keypoints crucial for sign language pose estimation.

- **Compact Language Detector**
  - Language Identification: Automatically detects the language of the input text or audio for multilingual translation support.
- **Pose Estimation Libraries**
  - OpenPose and MediaPipe: Employed to extract skeletal pose keypoints and hand landmarks to animate signing gestures accurately.
- **Speech Recognition**
  - OpenAI Whisper: Converts spoken language from audio into accurate text form, serving as the foundation for further translation into sign language.

2. LITERATURE SURVEY

Table 2.1 Literature Survey of A Dynamic Sign Language Synthesis

Sno.	Author & Year of Publication	Journal or Conference	Methodology/Algorithm or Techniques Used	Merits	Demerits	Research Gap
1	Anant Kaulage, Ajinkya Walunj, Akash Bhandari, Aneesh Dighe, Anish Sagri (2024)	IEEE ICDSIS	Whisper model for STT; BERT, T5, TextRank for summarization; ConceptNet + BERT for Q&A; Multilingual translation; ISL support via NLP techniques	95.12% accuracy in subtitle generation; 3.43% WER in STT; Multilingual translation, summarization, Q&A, ISL support	Lack of Public Dataset; Models may not generalize with Indian Language	Improve ISL automation; Extend to corporate/ media platforms
2	Vi N. T. Truong, Chuan-Kai Yang, Quoc-Viet Tran (2016)	IEEE GCCE	A Translator for ASL to Text and Speech; AdaBoost + Haar classifiers; Dataset: 28,000 positive, 11,100 negative images	Recognizes static ASL alphabets; 98.7% precision; Real-time video to text/speech	Recognizes static ASL alphabets only	Extend to dynamic gestures and other languages; Add grammar-aware sentence formation
3	Lipisha Chaudhary, Tejaswini Ananthanarayana, Enjamamul Hoq, Ifeoma Nwogu (2022)	IEEE TPAMI	Dual Transformer Networks; Metric embedding for sign similarity; Cross-feature fusion; Keypoint-based pose inputs; BLEU score evaluation	Two-way transformer translation; Metric embedding based on sign similarity	Focused only on GSL; Computationally expensive; No facial/gesture nuance	Extend to more sign languages; Avatar-based output; Train with diverse signers; Real-time deployment
4	Navroz Kaur Kahlon,	Springer	Literature review across 148 studies; Classified	Reviewed 148 studies;	Classified translation approaches;	Use deep learning for

	Williamjeet Singh (2021)		techniques: statistical, neural, hybrid; Survey of evaluation metrics and generation techniques	Classified translation approaches; Manual vs. automatic evaluation	Manual vs. automatic evaluation	translation; Standardize evaluation metrics; Improve sign clarity; Develop end-to-end systems
5	Hao Zhou et al. (2024)	ACM IoTDI	Meta-learning approach; Data from 14 native users; Avoided synthetic data; Vocabulary of 1057	Meta-learning-based wearable ASL recognition; Achieved 26.9% WER; Avoided flaws of virtual data	26.9% WER is relatively high; Limited to ASL; No multilingual support	Support continuous signing; Include gestures and facial cues

Anant Kaulage et al. (2024) proposed a multilingual English-to-Indian Sign Language (ISL) translation system leveraging the Whisper model for speech-to-text conversion, BERT and T5 models for text summarization, and ConceptNet combined with BERT for question answering. Their approach supports ISL through advanced NLP techniques and achieves a high subtitle generation accuracy of 95.12% with a low word error rate (WER) of 3.43% in speech-to-text. Despite these merits, the system suffers from a lack of public datasets for training and may face challenges in generalizing across diverse Indian languages. The authors highlight the need to improve ISL automation and suggest extending applications to corporate and media platforms for broader impact [1].

Vi N. T. Truong et al. (2016) developed a real-time translator for American Sign Language (ASL) alphabets to text and speech using AdaBoost classifiers combined with Haar features. Their system demonstrated excellent precision of 98.7% in recognizing static ASL alphabets from a dataset consisting of 28,000 positive and 11,100 negative images. However, the methodology is limited to static gestures only and does not handle dynamic signing or grammatical sentence construction. The authors identify research gaps in extending the system to dynamic gestures and other languages, as well as integrating grammar-aware sentence formation modules [2].

Lipisha Chaudhary et al. (2022) introduced a dual transformer network architecture for bidirectional sign language translation, utilizing metric embeddings to capture sign similarity

and cross-feature fusion with keypoint-based pose inputs. Their system was evaluated using BLEU scores and is notable for supporting two-way translation between sign language and spoken language. The study focused specifically on Ghanaian Sign Language (GSL) and encountered computational challenges due to the complexity of the transformer models. Additionally, it did not address facial expressions or subtle gesture nuances, leaving room for future work on avatar-based outputs, training with diverse signers, and real-time deployment [3].

Navroz Kaur Kahlon and Williamjeet Singh (2021) conducted an extensive literature review covering 148 studies related to sign language translation. They classified existing approaches into statistical, neural, and hybrid techniques, and surveyed various evaluation metrics and generation methods. Their review highlighted distinctions between manual and automatic evaluation strategies. While comprehensive, the study pointed out the need for deep learning-based translation models, standardized evaluation metrics, enhanced clarity in sign generation, and the development of fully end-to-end translation systems as critical areas for future research [4].

Hao Zhou et al. (2024) proposed a meta-learning framework for wearable ASL recognition, collecting data from 14 native users and consciously avoiding synthetic datasets. Their approach featured a vocabulary of 1057 signs and achieved a word error rate (WER) of 26.9%, demonstrating the benefits of using real user data over virtual data. However, the relatively high WER indicates room for improvement, and the system currently supports only ASL without multilingual capabilities. The authors suggest future research should focus on supporting continuous signing and incorporating additional gesture and facial cues for richer sign language recognition [5].

3. ANALYSIS AND DESIGN

Analysis:
The project tackles the communication barrier faced by the DHH community by synthesizing dynamic sign language through a multimodal framework. Unlike current systems that rely on static gestures or prerecorded animations, this approach captures hand gestures, facial expressions, and body posture for accurate meaning. By integrating speech and text inputs, it offers a more natural communication interface. The system supports multiple sign languages like ASL, ISL, and GSL, aiming for scalable, real-time deployment across assistive devices and educational platforms.

Design:
The system uses a layered architecture: input processing, semantic interpretation, gesture synthesis, and rendering. Speech and text inputs are parsed via NLP models to extract intent and structure. These cues feed into a gesture generation module using deep learning sequence-to-sequence models (e.g., Transformers) trained on annotated datasets. A 3D avatar engine animates gestures dynamically, synchronizing hand movements, facial expressions, and gaze for lifelike signing.

3.1 MODULES

Spoken-to-Signed Translation Pipeline

a. Speech Recognition and Speech-to-Text

- **Function:** Converts spoken audio input into written text using Automatic Speech Recognition (ASR) models such as OpenAI Whisper.
- **Input:** Spoken audio.
- **Output:** Transcribed text.

b. Text to Sign Language Notation (Glosses)

- **Function:** Translates written text into gloss sequences using a combination of lemmatization, rule-based reordering, and Neural Machine Translation (NMT) techniques.
- **Input:** Transcribed or typed text.
- **Output:** Sign language glosses.

c. Gloss to Pose Sequence

- **Function:** Maps gloss sequences to skeletal pose sequences by referencing a pre-built sign lexicon and concatenating the corresponding poses.
- **Input:** Gloss sequence.
- **Output:** Pose sequence representing each sign.

d. Pose to Video Animation

- **Function:** Converts skeletal pose sequences into sign language video using image-to-image translation models like Pix2Pix, producing realistic signing avatars or animations.
- **Input:** Pose sequence.
- **Output:** Rendered sign language video.

3.2 ARCHITECTURE

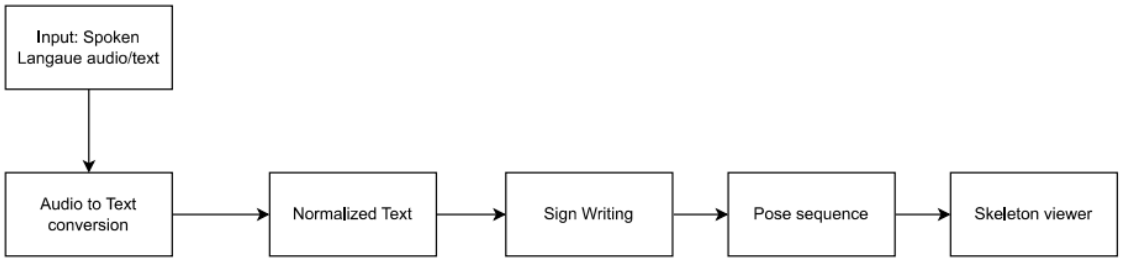


Figure 3.2.1 Architecture of Dynamic Sign Language Synthesis

Figure 3.2.1 This diagram represents the architecture of a Spoken-to-Signed Language Translation System that transforms spoken audio input into a visual sign language output.

The diagram illustrates the architecture of a spoken-to-signed language translation system that converts spoken language audio into sign language video using a multimodal processing pipeline. The process begins with capturing spoken audio, which is processed by a language identification module to determine the spoken language. This is followed by automatic speech recognition, where the audio is transcribed into text using models like Whisper. The resulting text is passed through a text normalization step to clean and structure it appropriately. This normalized text is then converted into SignWriting, a symbolic representation of sign languages. From the SignWriting output, a pose sequence is generated by mapping the signs to predefined gesture movements.

The generated pose sequences are then visualized through tools like the skeleton viewer, which displays the skeletal structure of the sign gestures. Additionally, a 3D avatar is used to render the sign language in a human-like animation for better accessibility and realism. For even more lifelike translations, the output can be enhanced using a human GAN, which synthesizes realistic signing videos. This integrated architecture supports real-time, personalized sign language generation, bridging communication gaps for the deaf and hard-of-hearing communities by combining audio processing, language modeling, gesture synthesis, and advanced visualization.

3.3 UML Diagrams

3.3.1 Use Case Diagram

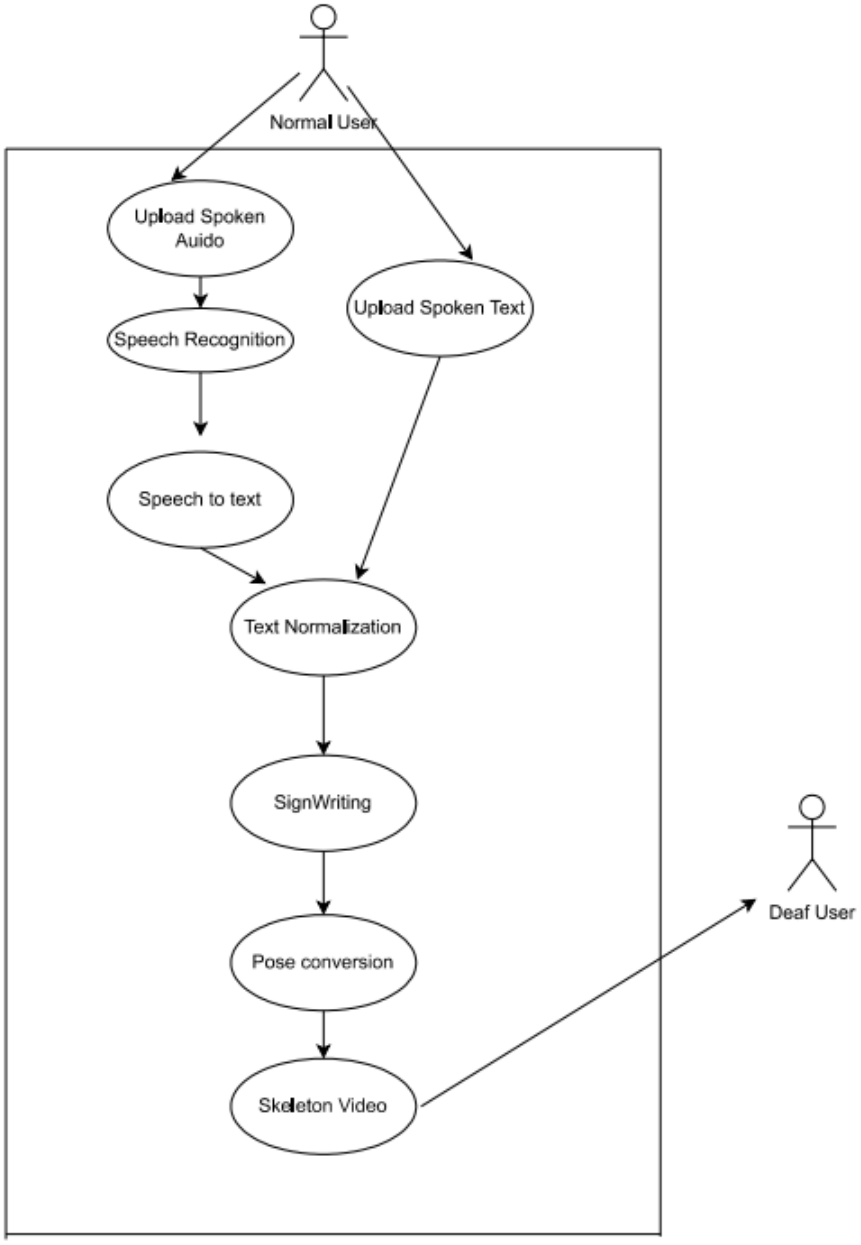


Fig. 3.3.1 Use Case Diagram of Dynamic Sign Language Synthesis

Figure 3.3.1 This use case diagram demonstrates the interaction between normal and deaf users with the system. Normal users can upload spoken audio or text, which undergoes speech recognition and normalization before being translated into sign language through sign writing, pose conversion, and skeleton video generation. The final output is directed toward deaf users for comprehension.

The use case diagram focuses on the actors (users) and how they interact with the system. There are two main actors: the Normal User, who uploads audio or text to be converted into sign

language, and the Deaf User, who is the end recipient of the translated output in the form of an animated sign language video.

For the normal user, key use cases include uploading input, receiving feedback or errors, and viewing the output video. For the deaf user, the primary concern is accessing the final sign language content. The system bridges the gap between spoken/written language and visual language, ensuring communication accessibility. This diagram is helpful for stakeholders and designers because it clearly identifies system boundaries and the responsibilities of each user group. It also informs testing scenarios, such as verifying that the uploaded input results in a valid output for the intended audience.

3.3.2 CLASS DIAGRAM

The class diagram presents the system's object-oriented structure, modelling the core software entities involved in the transformation pipeline. At the top is the Input Handler base class, from which Text Input and Audio Input inherit. These classes include properties like the original input data, detected language, and methods like Speech Recognition () and Cleaned Text().

The middle part of the diagram introduces a Gloss Converter class responsible for converting cleaned text into glosses. Following that, the Pose Generator class handles gloss-to-pose transformation using models trained on gloss-to-pose datasets. Finally, the Video Generator class animates the poses and produces a playable skeletal video.

This diagram is critical for backend developers, especially if you're designing a system using Python, Java, or another object-oriented language. It promotes inheritance, encapsulation, and modularity, and ensures that each class has a single, well-defined responsibility in the pipeline.

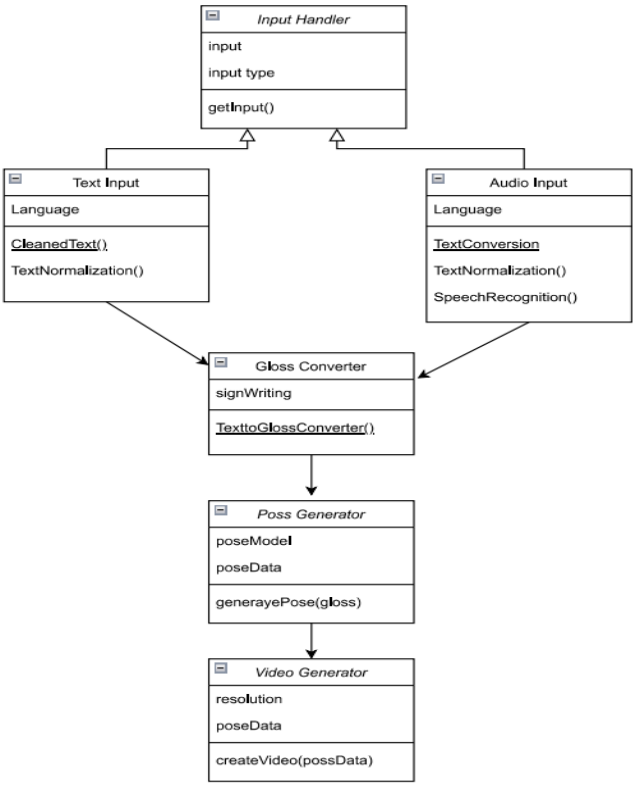


Fig. 3.3.2. Class Diagram of Dynamic Sign Language Synthesis

Fig 3.3.2 This class diagram shows the object-oriented design of the spoken-to-sign translation system. It includes classes for input handling, text and audio processing, gloss conversion, pose generation, and video generation. Inheritance and method definitions illustrate the structured data flow and modular architecture of the translation pipeline.

3.3.3 ACTIVITY DIAGRAM

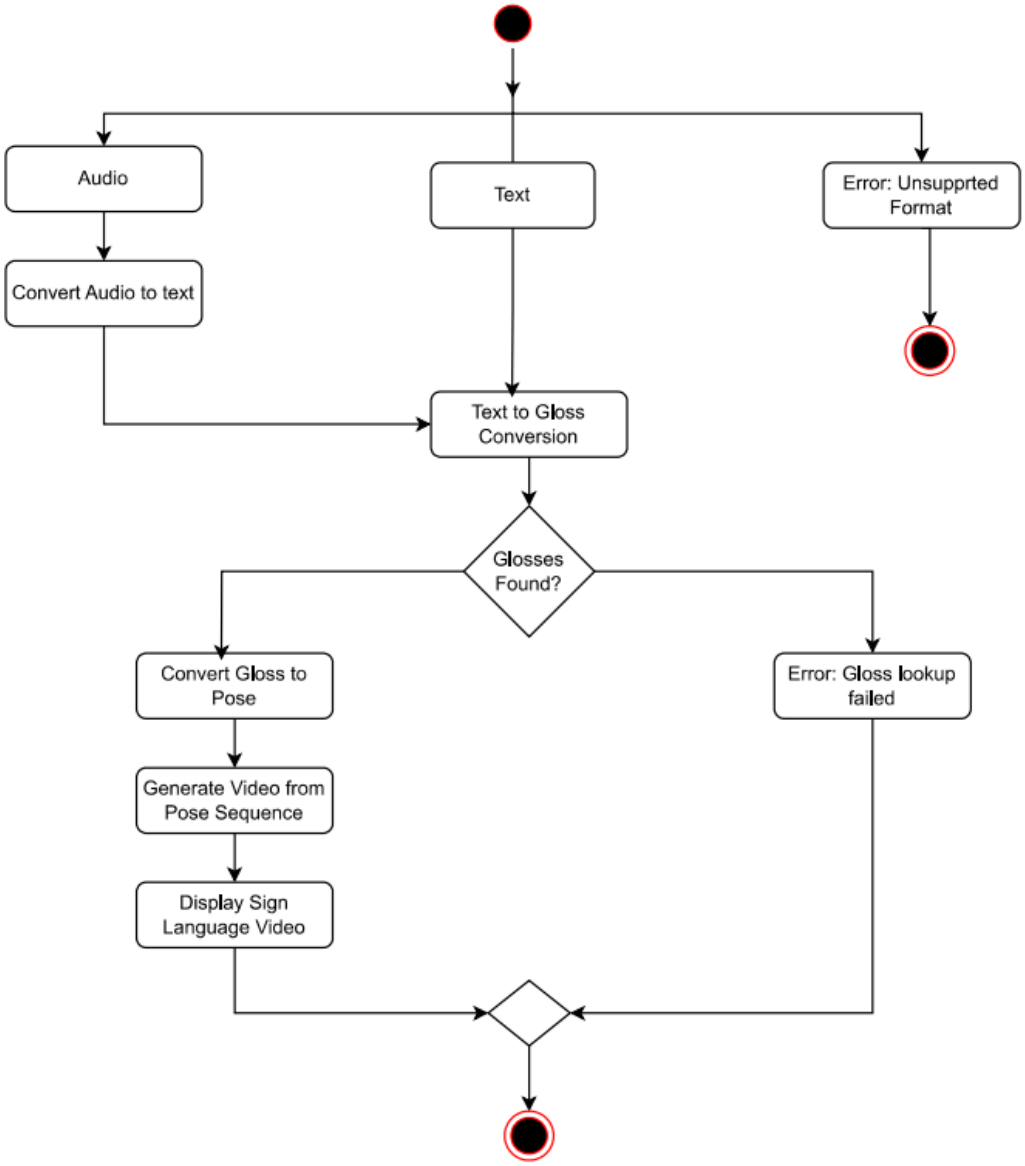


Fig. 3.3.3 Activity Diagram of Dynamic Sign Language Synthesis

Fig 3.3.3 This activity diagram is a variation of the earlier architecture diagram (Figure 3.2.2), enhanced with visual markers for start and end points. It represents the spoken-to-signed translation process from input to video output, including branches for error handling. The red-circled nodes mark system start and termination points.

The activity diagram maps out the dynamic behavior of the system, illustrating the step-by-step process a user's input follows through the pipeline. The flow starts with either audio or text input. If audio is provided, the system invokes the SpeechRecognition process. Afterward, the text undergoes cleaning and normalization, which includes removing punctuation and correcting grammar issues to better suit gloss translation.

The system then tries to convert the normalized text into sign language glosses using a predefined lexicon or model. A decision node checks whether valid glosses could be extracted.

If not, the system gracefully handles the error by sending back a user-readable message. If successful, the glosses move on to the pose generator, which creates a sequence of keypoints. These poses are animated into a skeletal video using OpenPose or similar tools, and finally rendered as output.

The use of decisions and parallel branches makes this diagram particularly valuable in understanding how the system responds under different scenarios (e.g., unsupported input, missing glosses, etc.), giving a more realistic view of execution paths.

3.3.4 SEQUENCE DIAGRAM

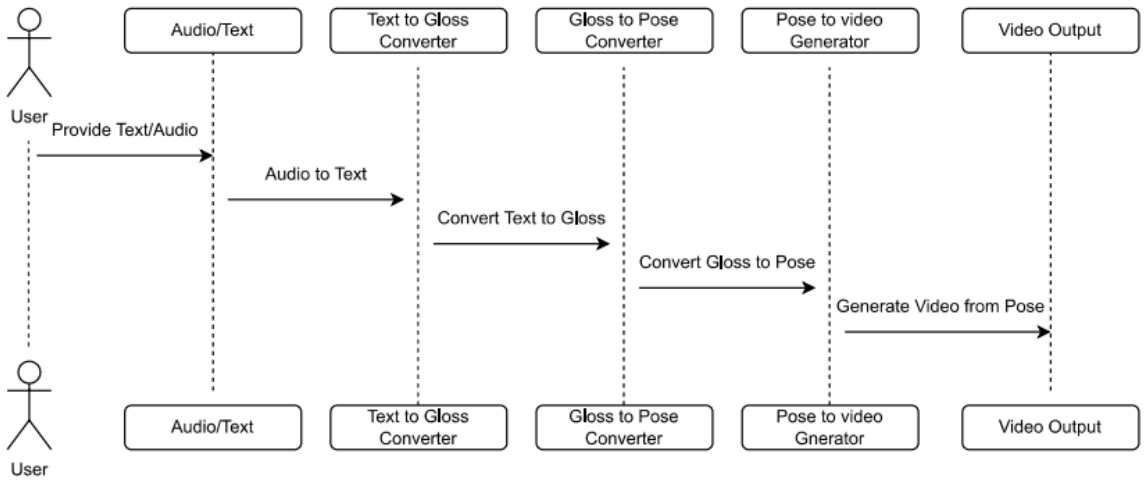


Fig. 3.3.4 Sequence Diagram of Dynamic Sign Language Synthesis

Fig 3.3.4 This sequence diagram depicts the chronological interaction among system components involved in converting text or audio to sign language video. It illustrates how a user provides input, which flows through modules for audio-to-text conversion, text-to-gloss translation, gloss-to-pose transformation, and final video rendering, before being output back to the user.

The sequence diagram illustrates the temporal order of operations that occur when a user submits input. Starting from the User, a request is sent to the TextToGlossConverter, which processes the input and sends the gloss to the GlossToPoseConverter. This component generates a sequence of poses that are then sent to the PoseToVideoGenerator, which finally passes the completed animation to the VideoOutput.

Each interaction is represented as a message arrow, making it easy to trace how data flows over time. The diagram also shows how intermediate objects like GlossSequence or PoseFrame are exchanged between components. Optional logic (such as checking if glosses are valid or handling audio preprocessing) can be represented using activation bars or conditionals in an extended version.

This type of diagram is especially useful for debugging and integration, as it helps trace exactly where data may be failing, and which component is responsible for handling what part of the request in the runtime flow.

3.3.5 COMPONENT DIAGRAM

The component diagram drills down into how the overall system is decomposed into independently working modules and shows how they communicate. It emphasizes loose coupling and high cohesion by dividing the workflow into distinct blocks: Audio Input, Text Input, Text to Gloss, Gloss to Pose, Pose to Video, and finally, Video Output. Each component exposes specific input and output interfaces (e.g., “text”, “gloss”, or “pose sequence”), allowing other components to interact with it.

A key feature of this diagram is that it illustrates how both audio and text eventually converge into a shared “Text to Gloss” module, making the design efficient and reducing redundancy. It also visually highlights reusable components like the pose generator and video renderer, which remain the same regardless of the original input modality. This diagram is particularly useful for developers or software engineers working on different parts of the system, as it clarifies the data contracts and boundaries between modules.

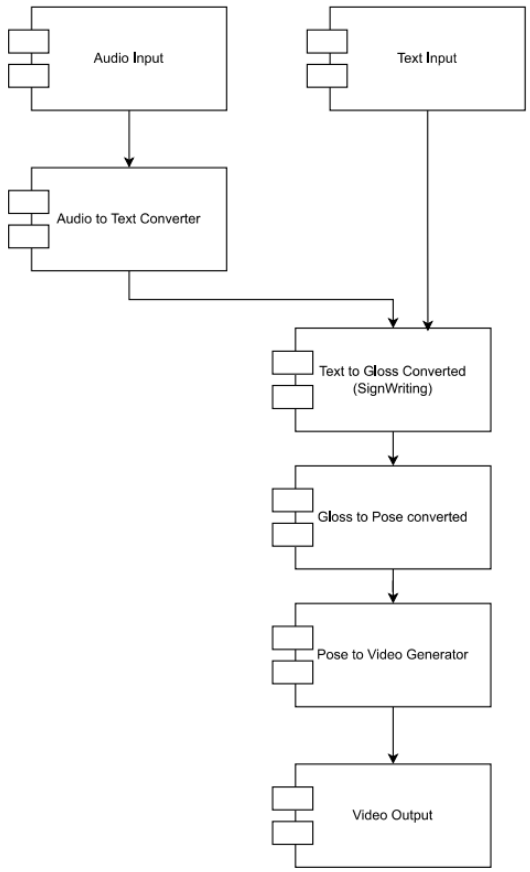


Fig. 3.3.5 Component Diagram of Dynamic Sign Language Synthesis

Fig 3.3.5 This component diagram represents the modular structure of the sign language translation system. It highlights the key processing blocks: audio-to-text converter, text-to-gloss converter, gloss-to-pose converter, pose-to-video generator, and final video output. Each module is shown with its interfaces, illustrating how audio and text inputs are independently processed and merged in the pipeline.

3.3.6 DEPLOYMENT DIAGRAM

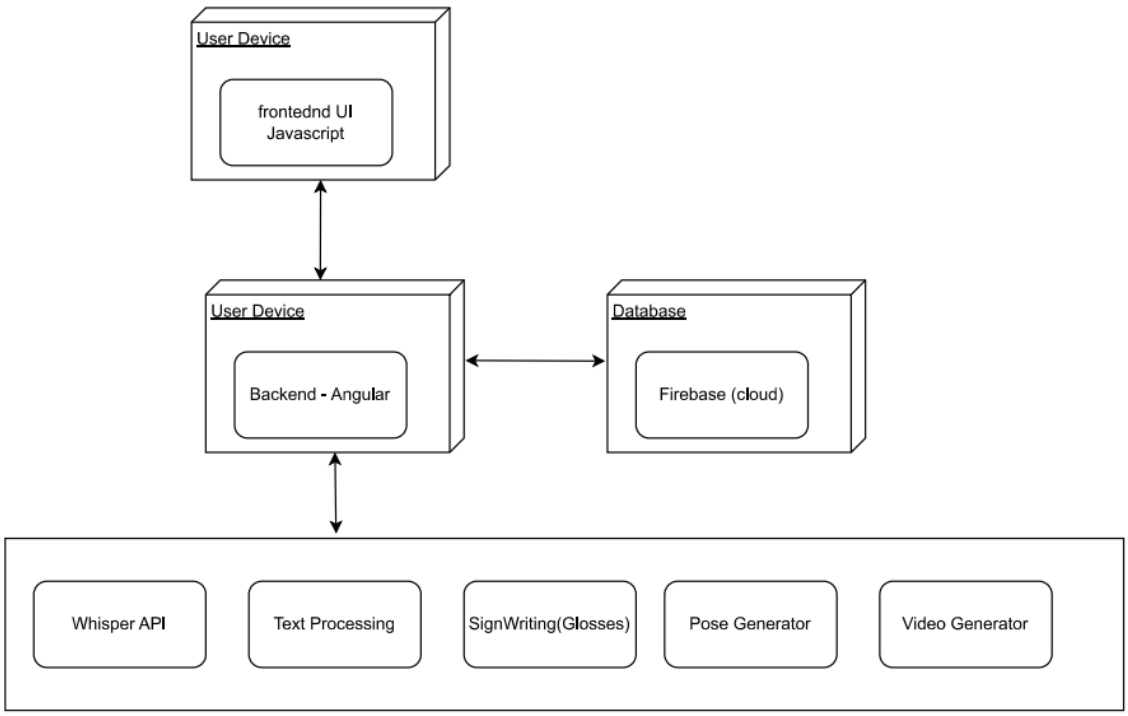


Fig 3.3.6 Deployment Diagram of Dynamic Sign Language Synthesis

Fig 3.3.6 represents Deployment Diagram of Dynamic Sign Language Synthesis showing the connection between frontend, backend and database

This deployment diagram illustrates the architecture of a dynamic sign language translation system designed to convert English speech or text into animated sign language videos using SignWriting and pose-based rendering. The system operates primarily on the user device, where the frontend interface, developed in JavaScript, enables users to upload videos, audio, or text inputs. This frontend communicates with an Angular-based backend on the same device, which manages the data flow and oversees the processing pipeline. To support storage and retrieval, the backend interacts with a Firebase cloud database that holds input data, intermediate translations, user settings, and generated sign language outputs.

The backend orchestrates a modular translation pipeline that begins with the Whisper API for speech-to-text conversion, followed by text processing to clean and prepare the input. The processed text is then translated into a structured sign language gloss format using the SignWriting module. Subsequently, the Pose Generator creates skeletal animations based on these glosses, and the Video Generator compiles these animations into cohesive sign language videos. This design ensures a smooth, scalable, and accessible user experience, specifically tailored to support real-time sign language translation for deaf and hard-of-hearing users.

3.4 Methodology

The proposed system follows a modular pipeline approach to convert spoken or written input into dynamic sign language videos. The process is divided into four main stages:

1. Speech Recognition

To handle spoken language input, the system employs WhisperX, an advanced Automatic Speech Recognition (ASR) model developed by OpenAI. WhisperX transcribes audio input into plain text with high accuracy, providing timestamps and speaker diarization features that assist in contextual processing of the spoken content.

2. Text to Gloss Conversion

Once plain text is available (either directly from user input or via speech recognition), the next step is to convert it into a sign language gloss format, which represents the grammatical structure and vocabulary of sign languages. This stage involves a hybrid approach combining:

- **Simple Lemmatizer**: Reduces inflected words to their base forms.
- **SpaCy Lemmatizer**: Uses advanced NLP techniques for context-aware lemmatization.
- **Rule-Based Word Reordering and Dropping**: Applies handcrafted linguistic rules to restructure sentences into the word order used in sign language, while dropping unnecessary words like articles and prepositions.
- **Neural Machine Translation (NMT)**: Translates natural language sentences directly into gloss format using deep learning models trained on parallel text-gloss corpora.

3. Gloss to Pose

The gloss text is then mapped to human body poses using a Lexicon Lookup method. Each gloss term corresponds to a predefined pose sequence. These sequences are retrieved from a sign lexicon database and concatenated to form a full pose timeline that represents the input sentence.

4. Pose to Video

Finally, the generated pose data is translated into a realistic sign language video using the Pix2Pix model, a popular Image-to-Image Translation framework. Pix2Pix takes skeletal pose frames and renders them as photorealistic sign language avatars or videos, preserving the nuances of hand shapes, motion, and facial expressions.

4. CODE AND IMPLEMENTATION

4.1 CODE

Spoken-to-signed-translation

Asssets/fingerspelling

1. create_index.py

```
import csv
from pathlib import Path
with Path('data.csv').open('r') as data_file:
   rows = list(csv.DictReader(data_file))
for row in rows:
   if row['spoken_language'] == 'en'
      for spoken_language, signed_language in [('fr', 'fsl')]:
         new_row = row.copy()
         new_row['spoken_language'] = spoken_language
         new_row['signed_language'] = signed_language
         rows.append(new_row)
with Path('index.csv').open('w', newline='') as index_file:
  writer = csv.DictWriter(index_file, fieldnames=rows[0].keys())
   writer.writeheader()
   writer.writerows(rows)
```

2.download.py

```
import csv
from pathlib import Path
from google.cloud import storage
client = storage.Client()
bucket_name = 'sign-mt-poses'
bucket = client.bucket(bucket_name)
def download_file_from_gcs(bucket, source_filename, destination_path):
   """Download a file from GCS."""
   blob = bucket.blob(source_filename)
   blob.download_to_filename(destination_path)
   print(f"Downloaded {source_filename} to {destination_path}")
csv_file_path = Path('data.csv')
with csv_file_path.open('r') as csvfile:
   reader = csv.DictReader(csvfile)
   for row in reader:
      full_path = Path(row['path'])
      if full_path.exists():
         continue
      filename = full_path.name
      Path(row['signed_language']).mkdir(exist_ok=True)
      download_file_from_gcs(bucket, filename, str(full_path))
```

3.preprocess_files.py

```
from collections import defaultdict
from pathlib import Path
from pose_anonymization.appearance import remove_appearance
from pose_format import Pose
```

```
from pose_format.utils.generic import reduce_holistic, get_body_hand_wrist_index,
get_hand_wrist_index
from tqdm import tqdm
from spoken_to_signed.gloss_to_pose.concatenate import trim_pose, normalize_pose,
scale_normalized_pose
ONLY_RIGHT_HAND = {"ase", "sgg", "gsg"}
durations = defaultdict(list)
print("MAKE SURE THIS SCRIPT ONLY RUNS ONCE"
for file in tqdm(Path.cwd().rglob('*.pose')):
    with open(file, "rb") as f:
        pose = Pose.read(f.read())
    pose = trim_pose(pose)
    durations[file.parent.name].append(len(pose.body.data) / pose.body.fps)
    original_pose = pose
    pose = normalize_pose(pose)
    scale_normalized_pose(pose)
    pose = remove_appearance(pose)
    if file.parent.name in ONLY_RIGHT_HAND:
        left_hand_index = get_hand_wrist_index(pose, "left")
        pose.body.data[:, :, left_hand_index:left_hand_index + 21] = 0
        pose.body.confidence[:, :, left_hand_index:left_hand_index + 21] = 0
        left_wrist_index = get_body_hand_wrist_index(pose, "left")
        pose.body.data[:, :, left_wrist_index] = 0
        pose.body.confidence[:, :, left_wrist_index] = 0
    with open(file, "wb") as f:
        pose.write(f)
duration_averages = {k: sum(v) / len(v) for k, v in durations.items()}
for language, duration in duration_averages.items():
    print(language, duration
for file in tqdm(Path.cwd().rglob('*.pose')):
    with open(file, "rb") as f:
        pose = Pose.read(f.read())
    interpolate = False
    original_fps = pose.body.fps
    interpolation_fps = pose.body.fps
    if duration_averages[file.parent.name] > 1.1:
        speed_factor = duration_averages[file.parent.name]
        interpolation_fps /= speed_factor
        interpolate = True
    if original_fps > 30:
        interpolation_fps /= 2
        original_fps /= 2
        interpolate = True
    interpolation_fps = round(interpolation_fps)
    if interpolate:
        print("Interpolating", file, interpolation_fps)
        pose = pose.interpolate(interpolation_fps)
        pose.body.fps = original_fps
        with open(file, "wb") as f:
            pose.write(f)
```

2. Gloss-to-pose

Gloss-to-pose/lookup

1.__init__.py

```
from .lookup import PoseLookup
from .csv_lookup import CSVPoseLookup
```

4.lookup.py

```
import math
import os
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor
from typing import List
from pose_format import Pose
from spoken_to_signed.gloss_to_pose.languages import LANGUAGE_BACKUP
from spoken_to_signed.gloss_to_pose.lookup.lru_cache import LRUCache
from spoken_to_signed.text_to_gloss.types import Gloss
class PoseLookup:
    def __init__(self, rows: List,
                 directory: str = None,
                 backup: "PoseLookup" = None,
                 cache: LRUCache = None):
        self.directory = directory
        self.words_index = self.make_dictionary_index(rows, based_on="words")
        self.glosses_index = self.make_dictionary_index(rows, based_on="glosses")
        self.backup = backup
        self.file_systems = {}
        self.cache = cache if cache is not None else LRUCache()
    def make_dictionary_index(self, rows: List, based_on: str):
        languages_dict = defaultdict(lambda: defaultdict(lambda: defaultdict(list)))
        for d in rows:
            term = d[based_on]
            lower_term = term.lower()
            languages_dict[d['spoken_language']][d['signed_language']][lower_term].append({
                "path": d['path'],
                "term": term,
                "start": int(d['start']),
                "end": int(d['end']),
                "priority": int(d['priority']),
            })
        return languages_dict
    def read_pose(self, pose_path: str):
        if pose_path.startswith('gs://'):
            if 'gcs' not in self.file_systems:
                import gcsfs
                self.file_systems['gcs'] = gcsfs.GCSFileSystem(anon=True)
            with self.file_systems['gcs'].open(pose_path, "rb") as f:
                return Pose.read(f.read())
        if pose_path.startswith('https://'):
            raise NotImplementedError("Can't access pose files from https endpoint")
```

```
        if self.directory is None:
            raise ValueError("Can't access pose files without specifying a directory")
        pose_path = os.path.join(self.directory, pose_path)
        with open(pose_path, "rb") as f:
            return Pose.read(f.read())
    def get_pose(self, row):
        cached_pose = self.cache.get(row["path"])
        if cached_pose is None:
            pose = self.read_pose(row["path"])
            self.cache.set(row["path"], pose)
        pose = self.cache.get(row["path"])
        frame_time = 1000 / pose.body.fps
        start_frame = math.floor(row["start"] // frame_time)
        end_frame = math.ceil(row["end"] // frame_time) if row["end"] > 0 else -1
        return Pose(pose.header, pose.body[start_frame:end_frame]
    def get_best_row(self, rows, term: str):
        rows = sorted(rows, key=lambda x: x["priority"])
        for row in rows:
            if term == row["term"]:
                return row
        return rows[0
    def lookup(self, word: str, gloss: str, spoken_language: str, signed_language: str, source: str
= None) -> Pose:
        lookup_list = [
            (self.words_index, (spoken_language, signed_language, word)),
            (self.glosses_index, (spoken_language, signed_language, word)),
            (self.glosses_index, (spoken_language, signed_language, gloss)),
        ]
        for dict_index, (spoken_language, signed_language, term) in lookup_list:
            if spoken_language in dict_index:
                if signed_language in dict_index[spoken_language]:
                    lower_term = term.lower()
                    if lower_term in dict_index[spoken_language][signed_language]:
                     rows = dict_index[spoken_language][signed_language][lower_term]
                     return self.get_pose(self.get_best_row(rows, term))
        if signed_language in LANGUAGE_BACKUP:
            return self.lookup(word, gloss, spoken_language,
LANGUAGE_BACKUP[signed_language], source)
        if self.backup is not None:
            return self.backup.lookup(word, gloss, spoken_language, signed_language, source
        raise FileNotFoundError
    def lookup_sequence(self, glosses: Gloss, spoken_language: str, signed_language: str,
source: str = None):
        def lookup_pair(pair):
            word, gloss = pair
            if word == "":
                return None
            try:
                return self.lookup(word, gloss, spoken_language, signed_language)
            except FileNotFoundError as e:
```

```
            print(e)
            return None
        with ThreadPoolExecutor() as executor:
            results = executor.map(lookup_pair, glosses)
        poses = [result for result in results if result is not None]
        if len(poses) == 0:
            gloss_sequence = ' '.join([f"{word}/{gloss}" for word, gloss in glosses])
            raise Exception(f"No poses found for {gloss_sequence}")
        return poses
```

gloss-to-pose

1.__init__.py

```
from typing import Union
from pose_format import Pose
from .concatenate import concatenate_poses
from .lookup import PoseLookup, CSVPoseLookup
from ..text_to_gloss.types import Gloss
def gloss_to_pose(glosses: Gloss,
                  pose_lookup: PoseLookup,
                  spoken_language: str,
                  signed_language: str,
                  source: str = None,
                  anonymize: Union[bool, Pose] = False) -> Pose:
    poses = pose_lookup.lookup_sequence(glosses, spoken_language, signed_language,
source)
    if anonymize:
        try:
            from pose_anonymization.appearance import remove_appearance,
transfer_appearance
        except ImportError as e:
            raise ImportError("Please install pose_anonymization. "
                              "pip install git+https://github.com/sign-language-processing/pose-
anonymization") from e
        if isinstance(anonymize, Pose):
            print("Transferring appearance...")
            poses = [transfer_appearance(pose, anonymize) for pose in poses]
        else:
            print("Removing appearance...")
            poses = [remove_appearance(pose) for pose in poses]
    return concatenate_poses(poses)
```

2.Concatenate.py

```
from typing import List, Tuple
import numpy as np
from pose_format import Pose
from pose_format.utils.generic import reduce_holistic, correct_wrists,
pose_normalization_info, normalize_pose_size
from spoken_to_signed.gloss_to_pose.smoothing import smooth_concatenate_pose
class ConcatenationSettings:
    is_reduce_holistic = True
def normalize_pose(pose: Pose) -> Pose:
    return pose.normalize(pose_normalization_info(pose.header))
```

```
def get_signing_boundary(pose: Pose, wrist_index: int, elbow_index: int) -> Tuple[int, int]:
    pose_length = len(pose.body.data)
    wrist_exists = pose.body.confidence[:, 0, wrist_index] > 0
    first_non_zero_index = np.argmax(wrist_exists).tolist()
    last_non_zero_index = pose_length - np.argmax(wrist_exists[::-1]
    wrist_y = pose.body.data[:, 0, wrist_index, 1]
    elbow_y = pose.body.data[:, 0, elbow_index, 1]
    wrist_above_elbow = wrist_y < elbow_y
    first_active_frame = np.argmax(wrist_above_elbow).tolist()
    last_active_frame = pose_length - np.argmax(wrist_above_elbow[::-1])
    return (max(first_non_zero_index, first_active_frame - 5),
            min(last_non_zero_index, last_active_frame + 5))
def trim_pose(pose, start=True, end=True):
    if len(pose.body.data) == 0:
        raise ValueError("Cannot trim an empty pose")
    first_frame = len(pose.body.data)
    last_frame = 0
    hands = ["LEFT", "RIGHT"]
    for hand in hands:
        wrist_index = pose.header._get_point_index(f"POSE_LANDMARKS",
f"{hand}_WRIST")
        elbow_index = pose.header._get_point_index(f"POSE_LANDMARKS",
f"{hand}_ELBOW")
        boundary_start, boundary_end = get_signing_boundary(pose, wrist_index,
elbow_index)
        first_frame = min(first_frame, boundary_start)
        last_frame = max(last_frame, boundary_end
    if not start:
        first_frame = 0
    if not end:
        last_frame = len(pose.body.data)
    pose.body.data = pose.body.data[first_frame:last_frame]
   pose.body.confidence = pose.body.confidence[first_frame:last_frame]
    return pose
def concatenate_poses(poses: List[Pose], trim=True) -> Pose:
    if ConcatenationSettings.is_reduce_holistic:
        print('Reducing poses...')
        poses = [reduce_holistic(p) for p in poses]
    print('Normalizing poses...')
    poses = [normalize_pose(p) for p in poses
    if trim:
        print('Trimming poses...')
        poses = [trim_pose(p, i > 0, i < len(poses) - 1) for i, p in enumerate(poses)]
    print('Smooth concatenating poses...')
    pose = smooth_concatenate_poses(poses
    print('Correcting wrists...')
    pose = correct_wrists(pose)
    print('Scaling pose...')
    normalize_pose_size(pose)
```

```
    return pose
```
3.pose-to-video

__init__.py

```
from pose_format import Pose
def pose_to_video(pose: Pose, video_path: str):
    raise NotImplementedError
```
4.text-to-gloss

2.gpt.py

```
import json
import os
import re
from functools import lru_cache
from pathlib import Path
from typing import List
from openai import OpenAI
from dotenv import load_dotenv
from spoken_to_signed.text_to_gloss.types import Gloss, GlossIte
SYSTEM_PROMPT = """
You are a helpful assistant, who helps glossify sentences into sign language glosses.
Your task is to convert spoken language text into glossed sign language sentences following specific formatting rules.
Follow these guidelines
1. **Sentence Structure**:
   - Gloss each sentence separately. Break down long sentences into distinct, meaningful glosses for clarity.
   - Respond with a list of glossed sentences, each reflecting the structure of the original spoken sentence, using glosses and corresponding words.
   - Prefer SOV (Subject-Object-Verb) word order for glossing.
2. **Glossing Rules**:
   - Translate words into glosses in uppercase.
   - Retain the original spoken words alongside their glosses.
   - Place a slash `/` between the gloss and the original word to denote a direct translation. For example, "HELLO/Hello" or "NAME/name."
3. **Mouthing Notation**:
   - For sign languages with mouthing (such as German Sign Language - `gsg`), include the mouthing symbol `⌘` with the gloss.
   - Use open brackets to indicate the gloss that matches the mouth movement, for example: `⌘schön(SCHÖN/schöne)`.
4. **Named Entities**:
   - For proper nouns or named entities, use the `%` symbol in place of glossing. This denotes spelling out the entity instead of providing a gloss, for instance, `%(Inigo Montoya)`.
Use these rules and examples to produce accurate and readable glosses for each sentence provided.
""".strip()
@lru_cache(maxsize=1)
def get_openai_client():
    load_dotenv()
    api_key = os.environ.get("OPENAI_API_KEY", None)
    return OpenAI(api_key=api_key)
@lru_cache(maxsize=1)
```

```
def few_shots():
    data_path = Path(__file__).parent / "few_shots.json"
    with open(data_path, 'r', encoding="utf-8") as file:
        data = json.load(file
    messages = []
    for entry in data:
        messages.append({
            "role": "user",
            "content": json.dumps({
                "spoken_language": entry['spoken_language'],
                "signed_language": entry['signed_language'],
                "text": entry['text'],
            })
        })
        messages.append({"role": "assistant", "content": json.dumps(entry['sentences'])})
    return messages
def sentence_to_glosses(sentence: str) -> GlossItem:
    for item in sentence.split(" "):
        regex_with_mouthing = r"⌘(.*?)\((.*?)\)"
        if match := re.match(regex_with_mouthing, item):
            mouthing = match.group(1)
            content = match.group(2)
        else:
            mouthing = None
            content = item
        for sub_item in content.split(" "):
            if "/" in sub_item:
                sub_item_gloss, sub_item_word = sub_item.split("/")
            else:
                sub_item_gloss = sub_item_word = sub_item
            yield sub_item_word, sub_item_gloss
def text_to_gloss(text: str, language: str, signed_language: str, **kwargs) -> List[Gloss]:
    messages = [{"role": "system", "content": SYSTEM_PROMPT}] + few_shots() + [{
        "role": "user",
        "content": json.dumps({
            "spoken_language": language,
            "signed_language": signed_language,
            "text": text,
        })}
    ]
    response = get_openai_client().chat.completions.create(
        model="gpt-4o-mini",
        temperature=0,
        seed=42,
        messages=messages,
        max_tokens=500
    )
    prediction = response.choices[0].message.content
    print(prediction)
    sentences = json.loads(prediction)
```

```
    return [list(sentence_to_glosses(sentence)) for sentence in sentences]
if __name__ == '__main__':
    text = "Kleine kinder essen pizza."
    language = "de"
    signed_language = "sgg"
    print(text_to_gloss(text, language, signed_language))
```

3.nmt.py

```
import os
import tarfile
import requests
import torch as pt
import sentencepiece as spm
import sockeye.inferenc
from sockeye import inference, model
from typing import Dict, List, Any
from .types import Gloss
MODELS_PATH = './models
def download_and_extract_file(url: str, filepath: str):
    print("Attempting to download and extract: %s" % url)
    r = requests.get(url)
    filepath_tar_ball = filepath + ".tar.gz"
    open(filepath_tar_ball, 'wb').write(r.content)
    tar = tarfile.open(filepath_tar_ball)
    tar.extractall(path=MODELS_PATH)
    tar.close()
    print("Model saved to : %s" % filepath)
def download_model_if_does_not_exist(sockeye_paths: Dict[str, str])
    model_path = sockeye_paths["model_path"]
    url = sockeye_paths["url"]
    if not os.path.exists(model_path):
        download_and_extract_file(url, model_path)
    assert os.path.exists(model_path), "Model folder '%s' does not exist after " \
                                       "attempting to download and extract." % model_path
def load_sockeye_models():
    os.makedirs(MODELS_PATH, exist_ok=True)
    spm_name = "sentencepiece.model
    sockeye_paths_dict = {
        "dgs_de": {
                    "model_path": os.path.join(MODELS_PATH, "dgs_de"),
                    "spm_path": os.path.join(MODELS_PATH, "dgs_de", spm_name),
                    "url": "https://files.ifi.uzh.ch/cl/archiv/2022/easier/dgs_de.tar.gz"
                   }
    }
    sockeye_models_dict = {}
    device = pt.device('cpu')
    for model_name in sockeye_paths_dict.keys()
        sockeye_paths = sockeye_paths_dict[model_name]
        download_model_if_does_not_exist(sockeye_paths)
        model_path = sockeye_paths["model_path"]
        spm_path = sockeye_paths["spm_path"]
```

```
        sockeye_models, sockeye_source_vocabs, sockeye_target_vocabs =
model.load_models(
            device=device, dtype=None, model_folders=[model_path], inference_only=True)
        sockeye_models_dict[model_name] = {"sockeye_models": sockeye_models,
                                           "spm_model":
spm.SentencePieceProcessor(model_file=spm_path),
                                           "sockeye_source_vocabs": sockeye_source_vocabs,
                                           "sockeye_target_vocabs": sockeye_target_vocabs}
    return device, sockeye_paths_dict, sockeye_models_dict
device, sockeye_paths_dict, sockeye_models_dict = load_sockeye_models()
def apply_pieces(text: str, spm_model: spm.SentencePieceProcessor) -> str:
    text = text.strip(
    pieces = spm_model.encode(text, out_type=str
    return " ".join(pieces)
def remove_pieces(translation: str) -> str:
    """
    :param translation:
    :return:
    """
    translation = translation.replace(" ", "")
    translation = translation.replace("▁", " ")
    return translation.strip()
def add_tag_to_text(text: str, tag: str) -> str:
    text = text.strip()
    if text == "":
        return ""
    tokens = text.split(" ")
    tokens = [tag] + tokens
    return " ".join(tokens)
def translate(text: str,
              source_language_code: str = "de",
              target_language_code: str = "dgs",
              nbest_size: int = 3) -> Dict[str, Any]:
    if source_language_code == "de":
        model_name = "dgs_de"
    else:
        raise NotImplementedError()
    sockeye_models = sockeye_models_dict[model_name]["sockeye_models"]
    sockeye_source_vocabs = sockeye_models_dict[model_name]["sockeye_source_vocabs"]
    sockeye_target_vocabs = sockeye_models_dict[model_name]["sockeye_target_vocabs"]
    spm_model = sockeye_models_dict[model_name]["spm_model"]
    pieces = apply_pieces(text, spm_model)
    tag_str = '<2{}>'.format(target_language_code)
    tagged_pieces = add_tag_to_text(pieces, tag_str)
    beam_size = nbest_size
    translator = inference.Translator(device=device,
                                      ensemble_mode='linear',
                                      scorer=inference.CandidateScorer(),
                                      output_scores=True,
```

```
                                batch_size=1,
                                beam_size=beam_size,
                                beam_search_stop='all',
                                nbest_size=nbest_size,
                                models=sockeye_models,
                                source_vocabs=sockeye_source_vocabs,
                                target_vocabs=sockeye_target_vocabs)
    input_ = inference.make_input_from_plain_string(0, tagged_pieces)
    output = translator.translate([input_])[0]
    translations = output.nbest_translations
    translations = [remove_pieces(t) for t in translations]
    return {
        'source_language_code': source_language_code,
      'target_language_code': target_language_code,
        'nbest_size': nbest_size,
        'text': text,
        'translations': translations,
    }
def text_to_gloss(text: str, language: str, nbest_size: int = 3, **kwargs) -> List[Gloss]:
    if language == "de":
        translations_dict = translate(text=text,
                                source_language_code="de",
                                target_language_code="dgs",
                                nbest_size=nbest_size)
    else:
        raise NotImplementedError()
    best_translation = translations_dict["translations"][0]
    glosses = best_translation.split(" ")
    tokens = [None] * len(glosses)
    return [list(zip(tokens, glosses))]
```

5.simple.py

```
from typing import List
from simplemma import simple_tokenizer
from simplemma import text_lemmatizer as simple_lemmatizer
from simplemma.strategies.dictionaries.dictionary_factory import
SUPPORTED_LANGUAGE
from .types import Gloss
def text_to_gloss(text: str, language: str, **unused_kwargs) -> List[Gloss]:
    if language in SUPPORTED_LANGUAGES:
        words = [w.lower() for w in simple_tokenizer(text)]
        lemmas = [w.lower() for w in simple_lemmatizer(text, lang=language)]
    else:
        words = lemmas = text.lower().split(' ')
    return [list(zip(words, lemmas))]
```

4.2 IMPLEMENTATION STEPS

The proposed system was implemented using a modular, end-to-end pipeline designed to convert spoken or written input into expressive and synchronized sign language animations. The architecture is structured to support flexibility, ease of debugging, and future extensibility. The core logic was divided into four main stages: input processing (speech/text to gloss), gloss-to-pose mapping, pose-to-animation rendering, and full pipeline execution with UI integration.

Project Setup and Folder Structure

A clear directory structure was maintained for modular development. It was organized as:

```
spoken-to-signed-translation/
├── spoken_to_signed/
   ├── text_to_gloss/
   ├── gloss_to_pose/
   ├── pose_to_video/
   └── pipeline/
```

- Each folder contained scripts and models for specific tasks, allowing modular experimentation and testing.

Environment Configuration

Python 3.8+ was used as the base environment. Required packages were installed using pip, including:

- torch, transformers, and scikit-learn for model operations
- numpy and pandas for data processing
- opencv-python and PIL for image/video manipulation
- flask for any server-side integration (optional)

Text-to-Gloss Conversion

The system then applied rule-based and model-based techniques to generate sign language glosses:

- Rule-based word reordering and token filtering
- Lemmatization using either a simple lemmatizer or SpaCy
- Optional Neural Machine Translation (NMT) to improve gloss accuracy

Gloss-to-Pose Mapping

The glosses were translated into skeletal key point data representing hand, arm, and body positions by:

- Performing a lexicon lookup for each gloss token
- Concatenating predefined skeletal sequences to form full sentence gestures
- Pose-to-Video Animation

  The generated pose sequences were used to create lifelike animations using:
- A generative model like Pix2Pix for converting pose skeletons to avatar-like images
- The frames were compiled into video clips using OpenCV and FFmpeg

Pipeline Integration

A complete Python script was developed under spoken to_signed/pipeline/ which:

- Accepts user input (typed or audio-converted)
- Executes the full flow: text → gloss → pose → video
- Outputs an animated video file of the sign language translation.

5. TESTING

INTRODUCTION TO TESTING

Testing is a critical aspect of software development aimed at verifying, validating, and ensuring the quality and reliability of various software components. It helps to minimize risks and optimize resource utilization throughout the development lifecycle. Although testing can be applied at any stage, implementing it early allows for the identification and resolution of defects before they escalate. Testing involves evaluating the software under different conditions and environments to assess its functionality, performance, and other essential attributes. Various testing methodologies are employed depending on the nature and objectives of the software being developed. In this project, the testing phase ensures that every module functions effectively, thereby contributing to the system's overall reliability and performance.

Testing was an integral part of this project, focusing on assessing the system's ability to accurately predict cryptocurrency prices, specifically Ethereum and Bitcoin, over different time horizons—namely 1 week, 1 month, 3 months, and 6 months. The objective was to ensure that the price prediction system delivers consistent, accurate, and reliable results under diverse scenarios and dynamic market conditions.

The test cases were carefully designed to cover the key aspects of the system's functionality, including:

- Accurate transition to required pages.
- Accurate prediction of Ethereum and Bitcoin prices across the specified future periods.
- Stability and robustness of predictions.
- Proper graph generation.

Each test case was executed systematically, and the results were thoroughly documented to compare actual outcomes against expected results.

TEST CASES

Table 5.1 Test Cases of Dynamic sign language synthesis

Test Case ID	Test Case Name	Test Case Description	Expected Output	Actual Output	Remarks
TC01	Speech-to-Text Conversion	Check if audio input is correctly converted to English text using Whisper.	Text matches spoken input accurately.	Text matches spoken input accurately.	Success
TC02	Text Normalization	Check if input text is cleaned and normalized before gloss conversion.	Output is cleaned of punctuation and normalized.	Output is cleaned of punctuation and normalized.	Success
TC03	Text-to-Gloss Conversion	Verify conversion of normalized text to correct gloss representation.	Correct sign language gloss generated.	Correct sign language gloss generated.	Success
TC04	Gloss-to-Pose Generation	Check if gloss is accurately converted to pose keypoints using the pose model.	Sequence of poses representing the gloss.	Sequence of poses representing the gloss.	Success
TC05	Pose-to-Video Rendering	Verify pose sequence is converted into skeletal animation video.	Accurate animation matching input gloss.	Accurate animation matching input gloss.	Success
TC06	End-to-End Flow (Audio)	Test complete pipeline from audio to ASL video output.	Sign language animation corresponds to input speech.	Sign language animation corresponds to input speech.	Success
TC07	End-to-End Flow (Text)	Test complete pipeline from text to ASL video output.	Sign language animation corresponds to input text.	Sign language animation corresponds to input text.	Success

Table 5.1, shows the results of the test cases completed. It can be concluded that the system redirects properly and there are no issues in generated graphs and tables based on the selected coin and forecast period

TC01: Speech-to-Text Conversion

Tests the system's ability to accurately transcribe spoken English audio into text using the Whisper model, ensuring the transcribed text matches the original speech for reliable downstream use.

TC02: Text Normalization
Verifies that input text is properly cleaned and standardized by removing punctuation, normalizing casing, and formatting consistently to improve the accuracy of later stages.

TC03: Text-to-Gloss Conversion
Checks that normalized text is correctly converted into a sequence of sign language gloss terms representing the intended signs for use in pose generation.

TC04: Gloss-to-Pose Generation
Assesses the system's capability to transform gloss sequences into precise skeletal pose keypoints that correspond naturally to the signs.

TC05: Pose-to-Video Rendering
Validates the conversion of pose sequences into smooth skeletal animation videos that clearly and accurately depict the sign language.

TC06: End-to-End Flow (Audio)
Ensures the entire pipeline—from raw audio input through speech-to-text, gloss conversion, pose generation, and video rendering—works seamlessly to produce accurate ASL animations..

TC07: End-to-End Flow (Text)
Similar to TC06, this test confirms the full process starting from text input results in coherent and accurate ASL video output.

6. RESULTS

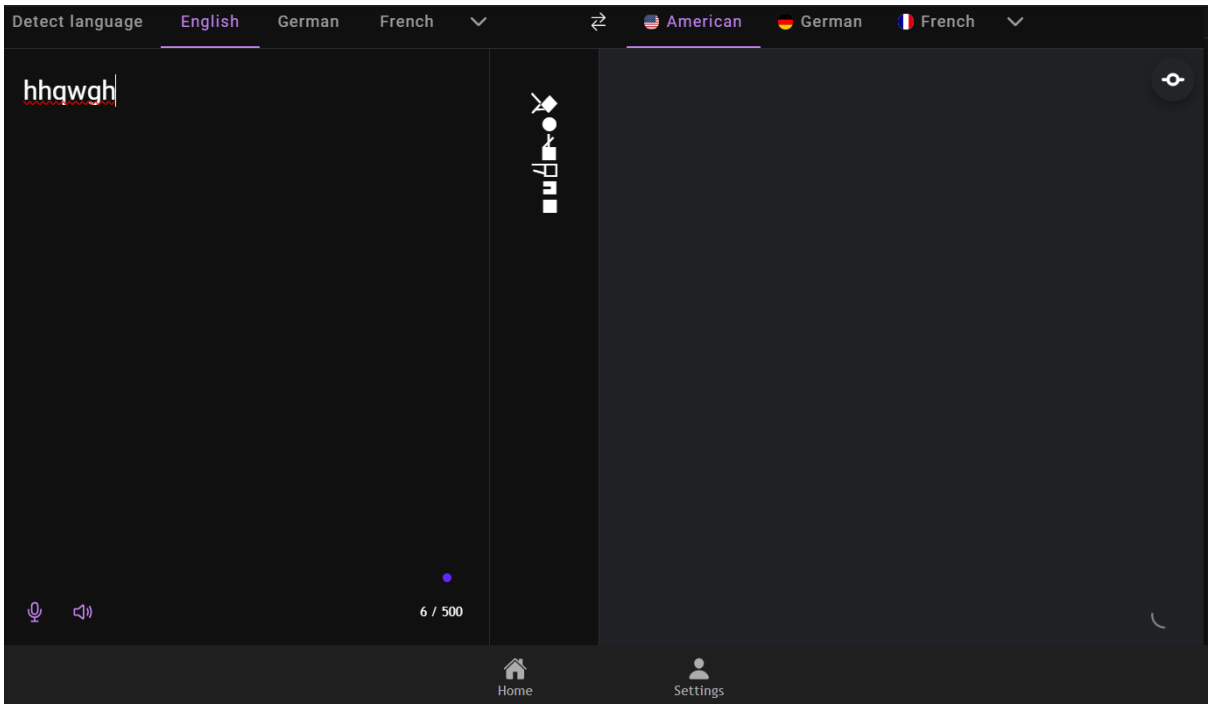


Fig 6.1. Taking input text

Fig 6.1 represents the test conducted to check if the system is taking input text

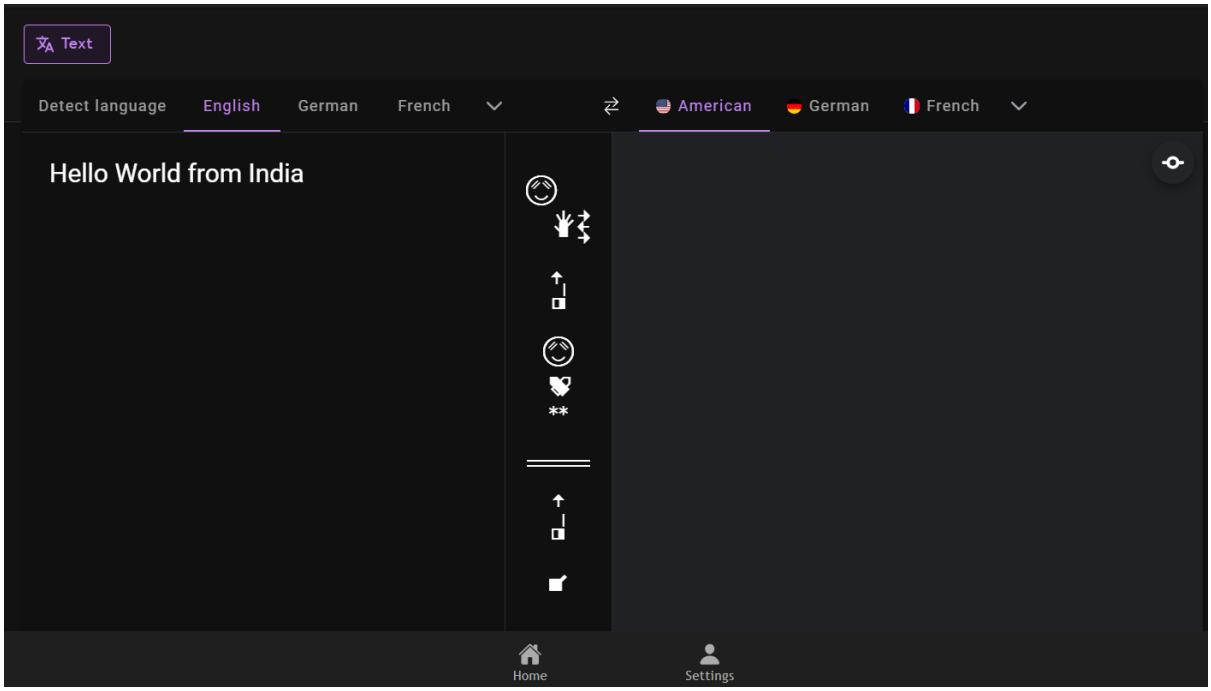


Fig 6.2. Gloss generate from input text

Fig 6.2 shows how the glosses are generated when the input is given as text

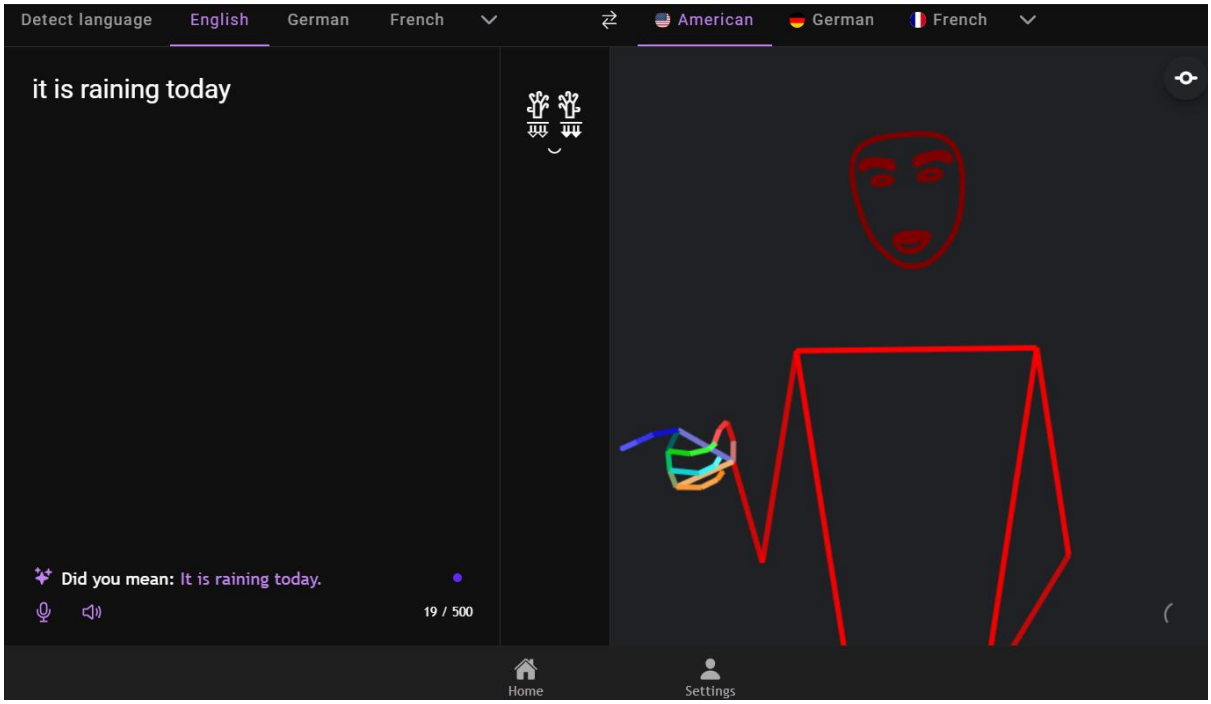


Fig 6.3. Input text to sign language pipeline

Fig 6.3 shows the full working pipeline of text to sign language synthesis when input is given as text the glosses are generated and then the video is produced

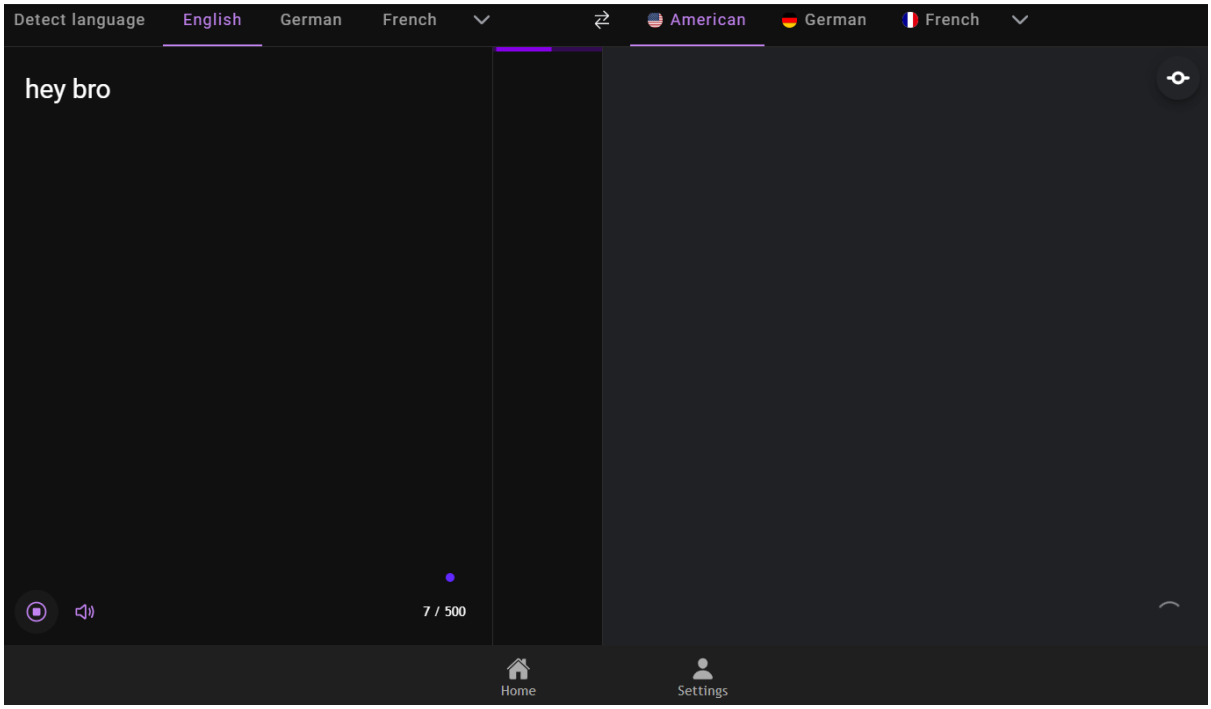


Fig 6.4. Taking Voice inputs

Fig 6.4 shows the working of voice inputs

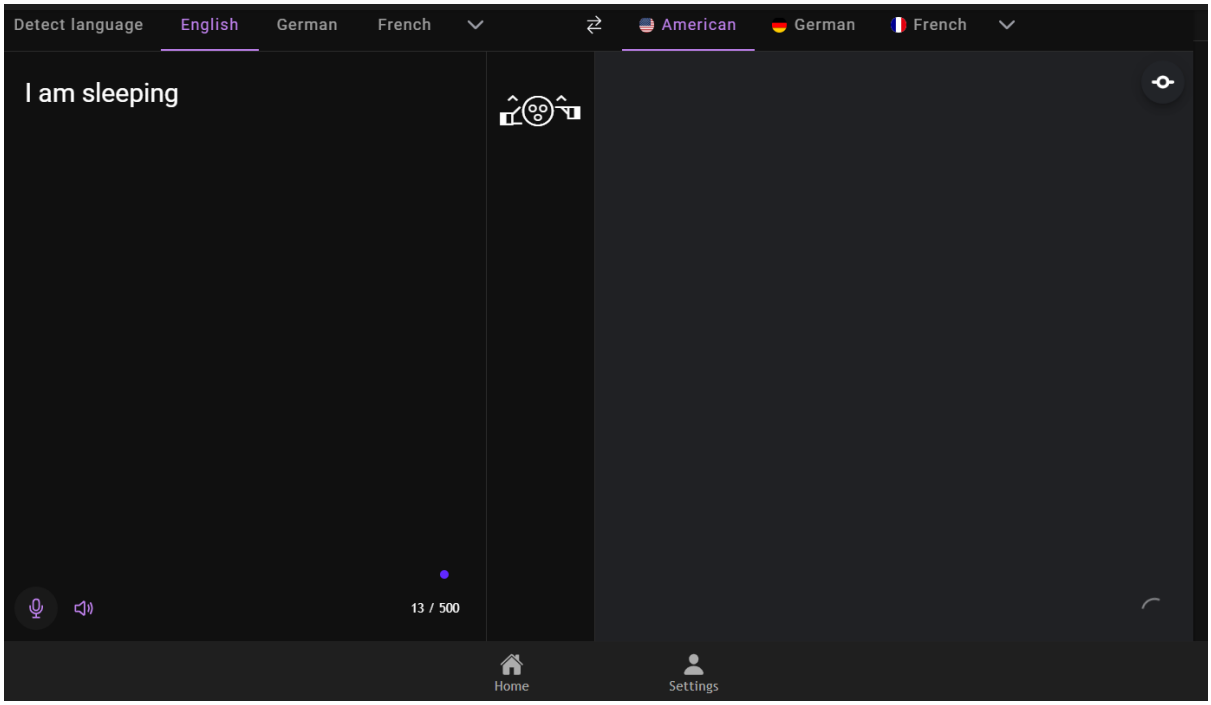


Fig 6.5. Gloss generation when input is voice

Fig 6.5 shows the generation of glosses when the input is given as voice

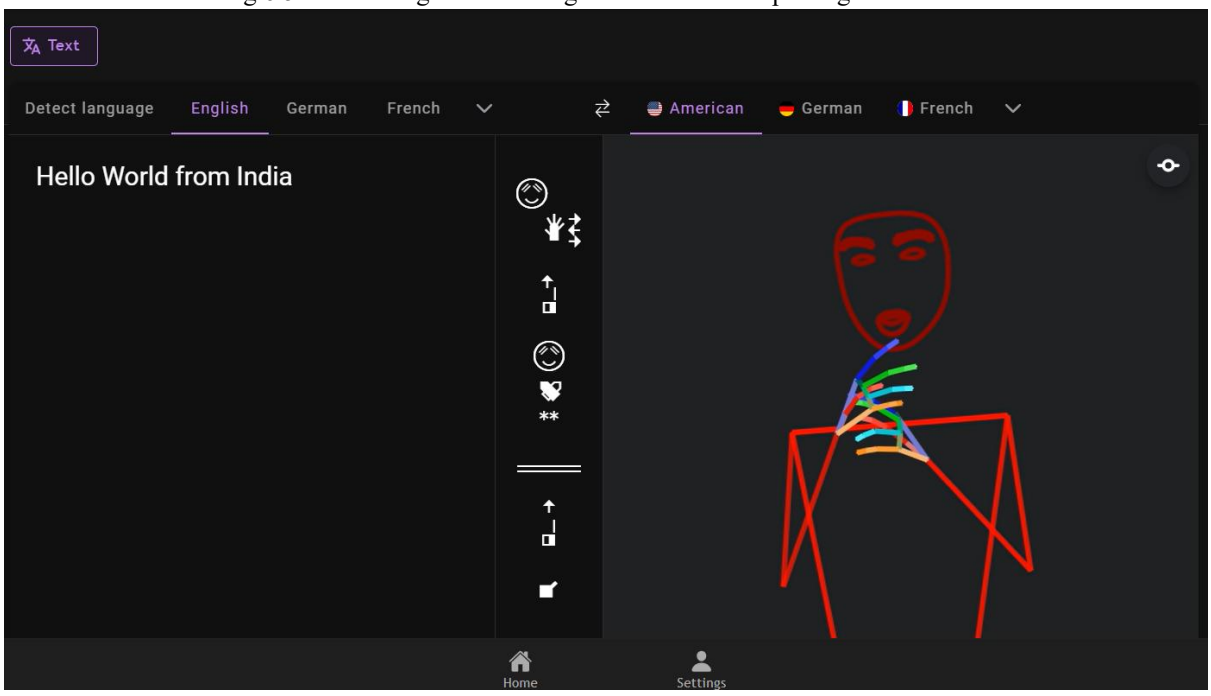


Fig 6.6. Input audio to sign language generation pipeline

Fig 6.6 shows the full working pipeline of audio to sign language synthesis when input is given as audio the glosses are generated and then the video is produced

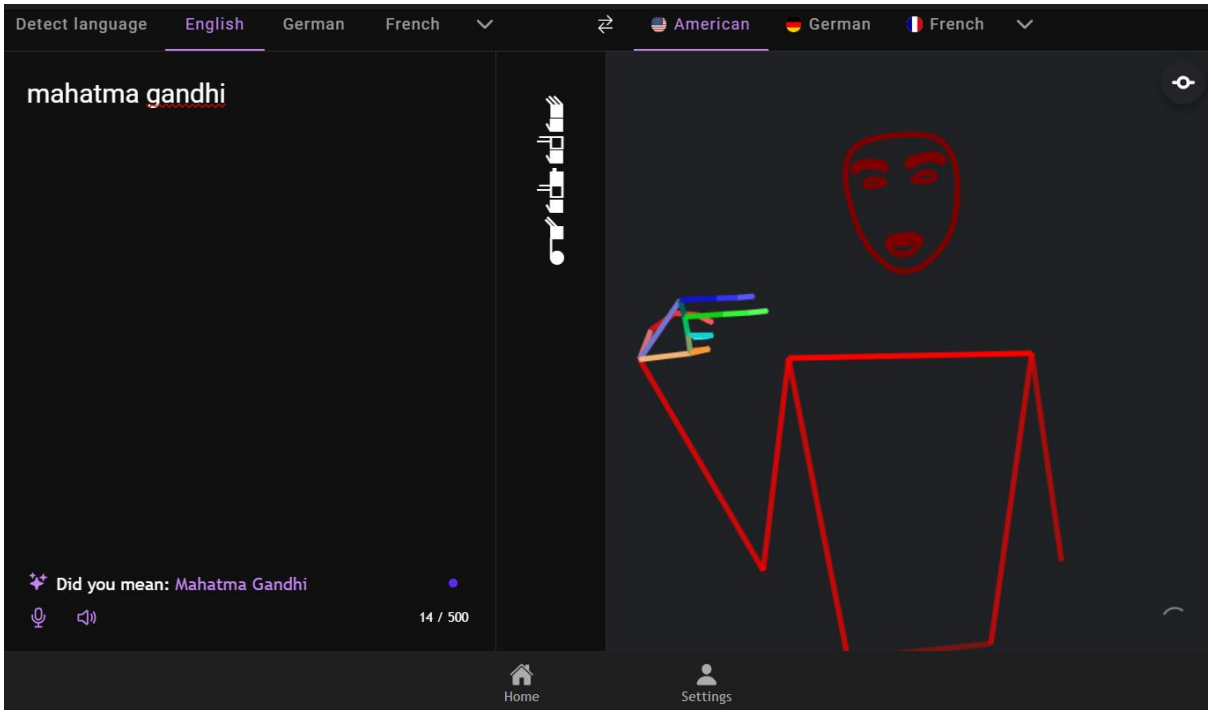


Fig 6.5. Finger Spelling when the word doesn't exist in the sign language letter – h

Fig 6.5 shows that when a certain word doesn't exist in a sign language then the word is spelled out using finger spelling

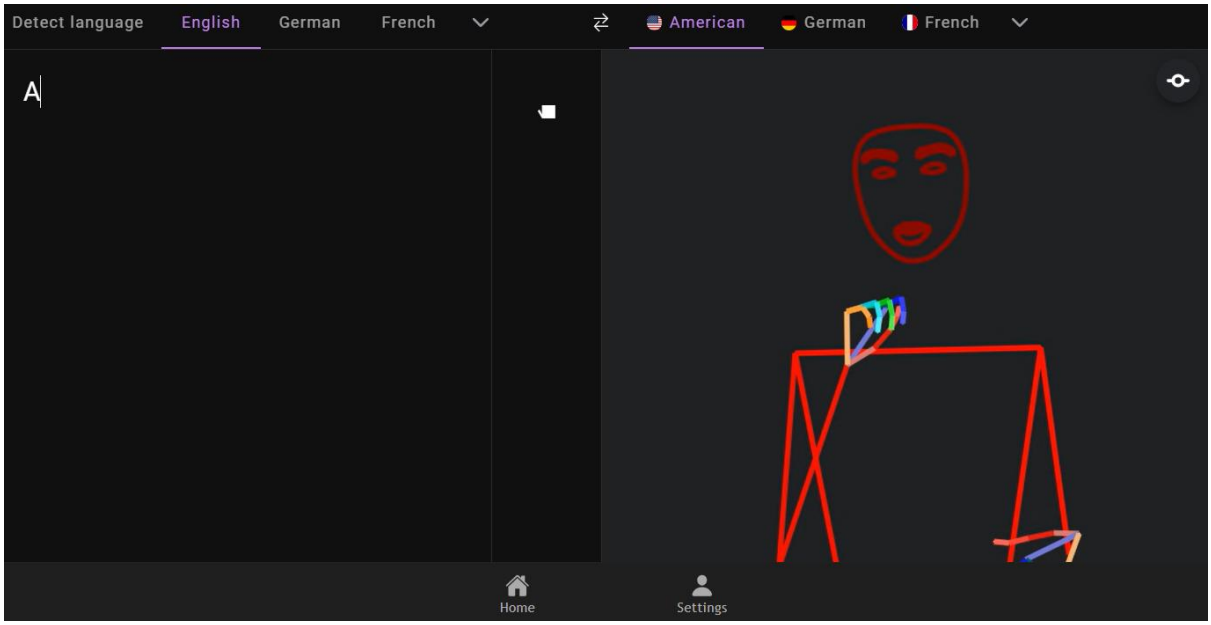


Fig 6.7. Captial Letter A

Fig 6.7 shows that the letter 'a' and 'A' have the same sign

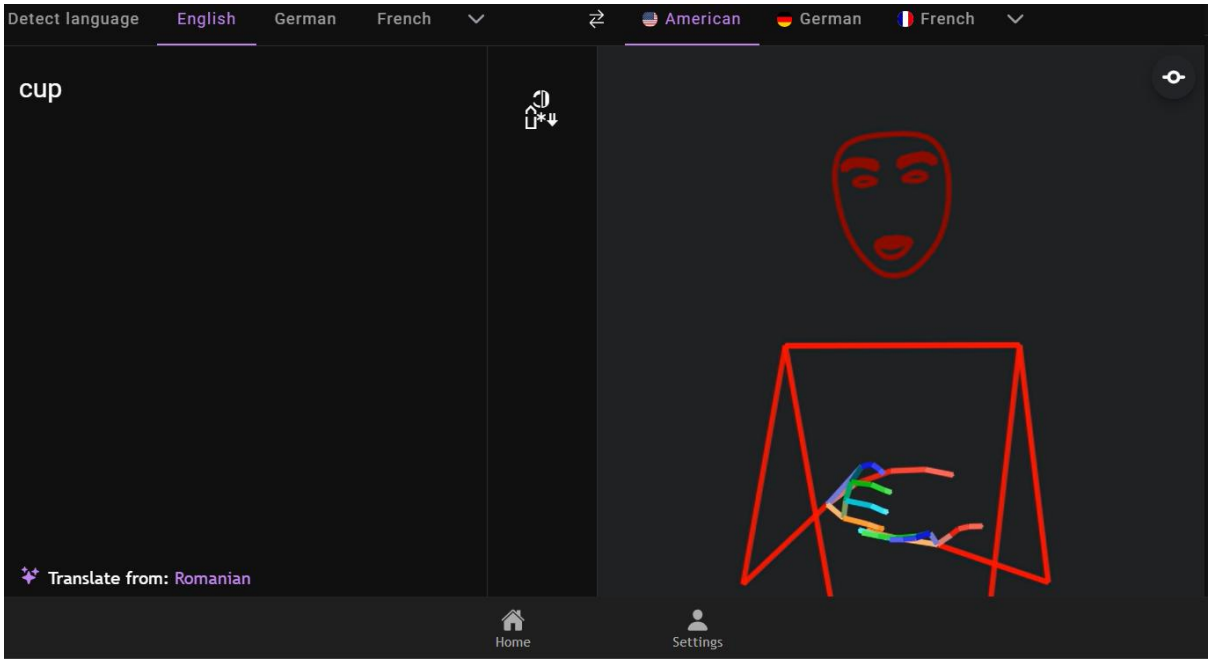


Fig 6.8. Translation of entire word directly it is an existing common word in sign language

Fig 6.8 shows that certain common words such as ‘cup’ has their own sign and need not use fingerspelling

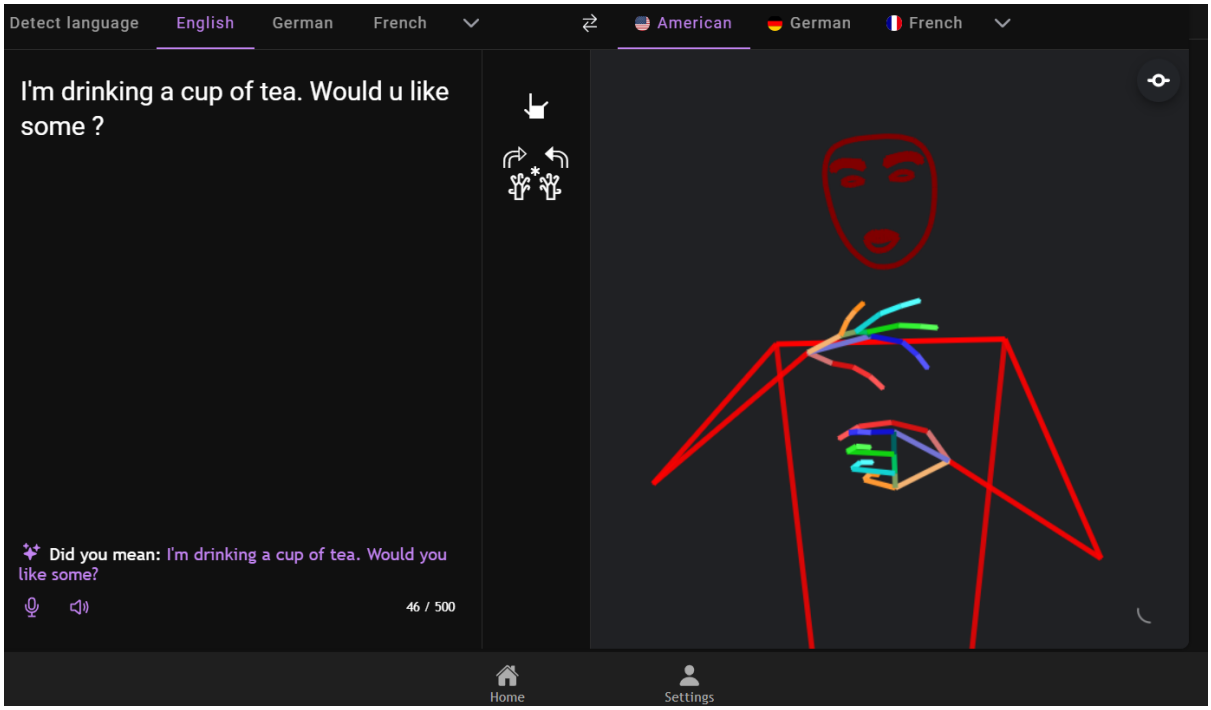


Fig 6.9. Multi-Sentence translation

Fig 6.9 shows the translation of multiple sentences

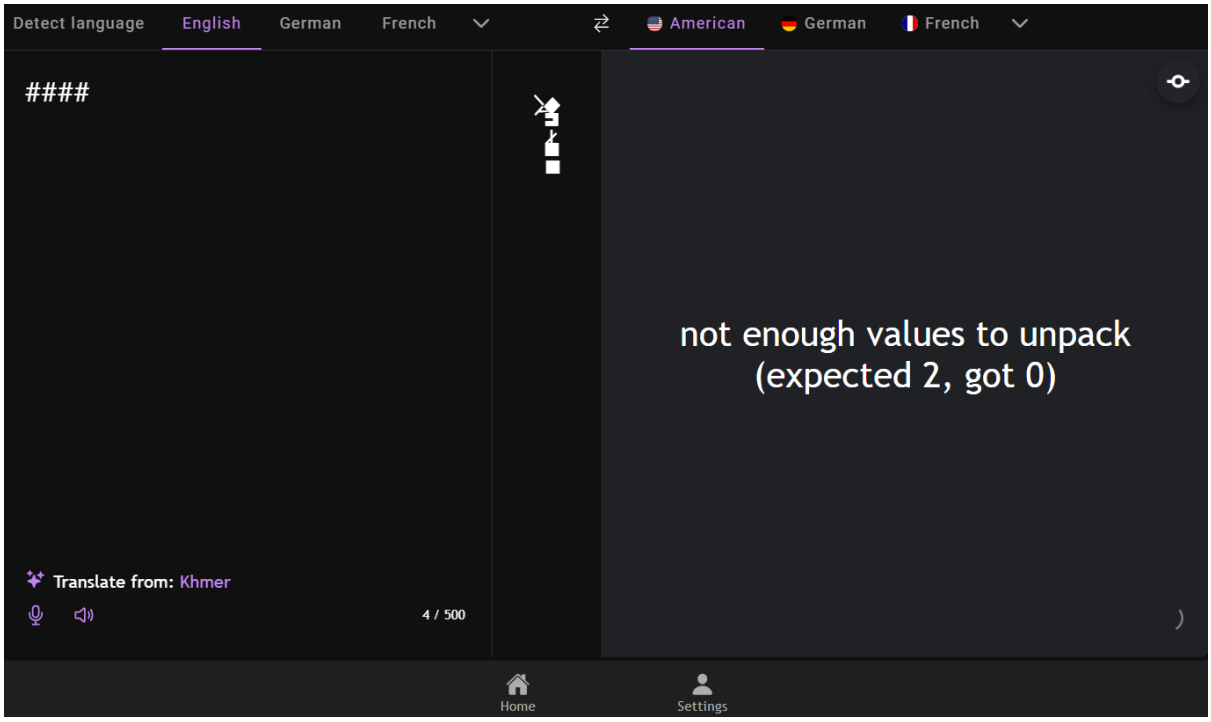


Fig 6.10. Cannot translate symbols with no meaning

Fig 6.10 shows that the system cannot translate symbols that have no meaning associated with it and are not used in sign language

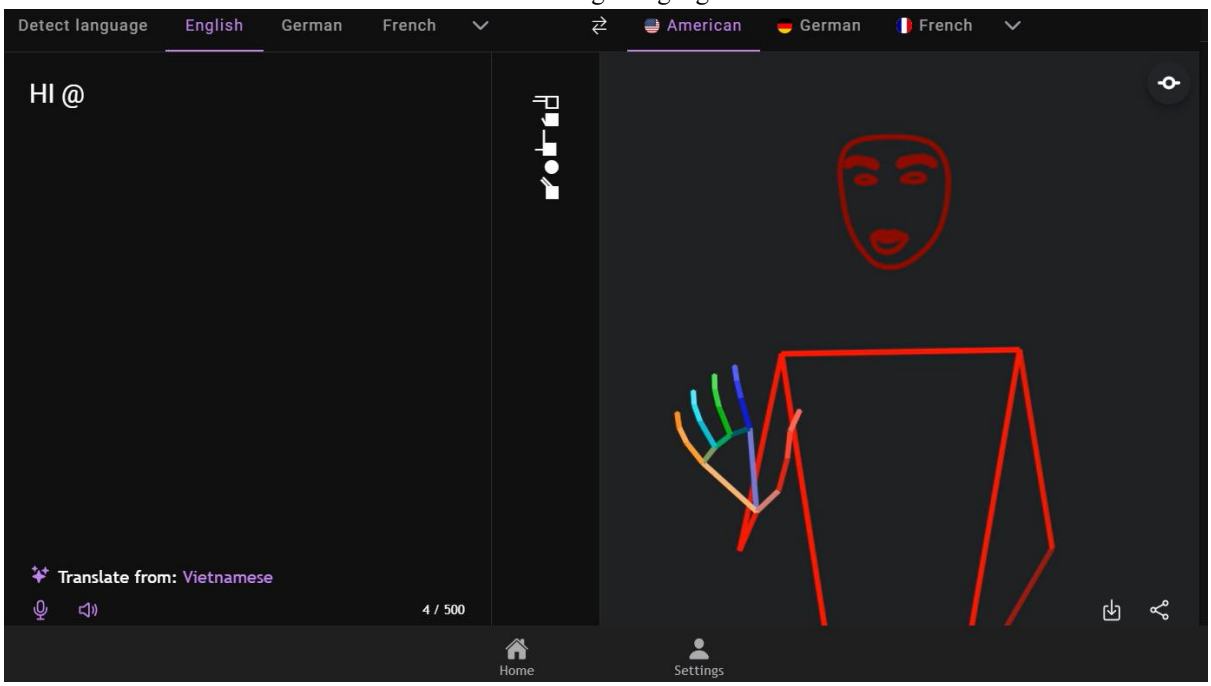


Fig 6.11. Text Cleaning – Ignores Symbols with no meaning

Fig 6.11 shows that when a sentences is passed with a symbol that symbol is ignored in text cleaning and normalization

7. CONCLUSION AND FUTURE ENHANCEMENTS

7.1 CONCLUSION

The project presents a comprehensive solution to address the communication barriers faced by the Deaf and Hard-of-Hearing (DHH) community through a real-time spoken-to-signed language translation system. By incorporating both text and speech inputs, the system offers flexibility and accessibility in diverse communication scenarios such as education, public service, and online media. The use of advanced techniques like automatic speech recognition, gloss generation, pose estimation, and avatar-based video rendering ensures that the sign language output is not only accurate but also expressive and lifelike. Through this approach, the project successfully bridges the gap between spoken language and sign language, offering synchronized, human-like animations that incorporate essential elements such as hand gestures, facial expressions, and body posture.

By eliminating the reliance on pre-recorded videos and static sign representations, the system introduces a scalable and dynamic framework that can be adapted for different sign languages in the future. Its layered design and modular architecture ensure ease of integration into assistive tools and real-time communication platforms.

7.2 FUTURE ENHANCEMENT

While the current implementation provides a robust pipeline for converting text and speech into lifelike sign language animations, several enhancements can further elevate the system's functionality. One potential improvement is the integration of emotion and sentiment detection in speech inputs to better capture tone and intent, resulting in more emotionally resonant sign animations.

Another area for enhancement includes the addition of more sign languages beyond ASL (such as ISL or GSL) to support regional and linguistic diversity. Incorporating user feedback and personalization mechanisms would allow for customizable avatar settings and signing styles, further improving user engagement and comfort.

Lastly, the system can be extended into interactive or bidirectional communication, where the user could receive spoken responses from a signing avatar through sign-to-text conversion—though currently not supported, this could form the foundation for a complete, multimodal communication loop in future versions.

REFERENCES

[1] A. Kaulage, A. Walunj, A. Bhandari, A. Dighe and A. Sagri, "Edu-lingo: A Unified NLP Video System with Comprehensive Multilingual Subtitles," 2024 Second International Conference on Data Science and Information System (ICDSIS), Hassan, India, 2024, pp. 1-8, doi: 10.1109/ICDSIS61070.2024.10594128

[2] V. N. T. Truong, C. -K. Yang and Q. -V. Tran, "A translator for American sign language to text and speech," 2016 IEEE 5th Global Conference on Consumer Electronics, Kyoto, Japan, 2016, pp. 1-2, doi: 10.1109/GCCE.2016.7800427

[3] L. Chaudhary, T. Ananthanarayana, E. Hoq and I. Nwogu, "SignNet II: A Transformer-Based Two-Way Sign Language Translation Model," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 11, pp. 12896-12907, 1 Nov. 2023, doi: 10.1109/TPAMI.2022.3232389

[4] N. K. Kahlon and W. Singh, "Machine translation from text to sign language: a systematic review," Springer-Verlag, 2021. DOI: https://doi.org/10.1007/s10209-021-00823-1.

[5] Hao Zhou, Taiting Lu, Kennath Dehaan, Mahanth Gowda “ASLRing: American Sign Language Recognition with Meta-Learning on Wearables” 2015 International Conference on Pervasive Computing (ICPC) DOI https://doi.org/10.1109/IoTDI61053.2024.00022

[6] A.Moryossef, "sign.mt: Real-Time Multilingual Sign Language Translation Application," arXiv preprint arXiv:2310.05064, 2024.

[7] SignLanguageTranslation Toolkit, GitHub Repository. Available: https://github.com/SignLanguageTranslation/SignLanguageTranslation

[8] A.Moryossef, S. Goldberg and A. Oron, "Real-Time Automatic Sign Language Detection using OpenPose and Gloss Alignment," arXiv preprint, arXiv:2002.00165, 2020. Available: https://arxiv.org/abs/2002.00165

[9] Sign2Text: ASL to Text Conversion using MediaPipe, GitHub Repository. Available: https://github.com/Dey-R/sign2text

[10] DeepASL: End-to-End ASL Recognition using Deep Learning, GitHub Repository. Available: https://github.com/yzhou359/DeepASL