

Role of Web-Based 3D Visualization in Facilitating Universal Two-Way Sign Language communication

Manasvi Bisht¹, Vidit Bhambhani¹, Tushar¹, Dr. Suman Goyat²

¹ Student, School of Engineering & Technology, Sushant University, Gurugram, India

² Professor, School of Engineering & Technology, Sushant University, Gurugram, India

ABSTRACT

Good communication opens doors, but for millions of deaf and hard-of-hearing people, daily conversations can be tough just because so few folks know sign language. This study looks at how web-based 3D visualization can change that, building a two-way sign language system that actually works for everyone whether you sign or not. The system combines web technology, 3D animated avatars, gesture recognition, and AI to turn text or speech into sign language, and sign language back into spoken or written words. Instead of making people depend on human interpreters or clunky one-off apps, this system runs straight from any regular web browser. That makes it affordable, easy to scale up, and available anywhere you've got internet. Tools like Three.js, WebGL, MediaPipe, Python, and advanced machine learning models are used to build this system. This creates a digital environment where hearing and non-hearing individuals can communicate naturally and understand each other in real time. The 3D avatars help make the signs clear and easy to follow, and the AI boosts translation so the conversation feels natural. This system isn't just for casual chats. It helps in classrooms, at work, in public spaces, or anywhere people connect. By blending advanced graphics with smart communication tools, this project shows how modern web tech breaks down barriers and helps everyone take part fully in society.

Keywords: Sign Language Translation, Support Vector Machine (SVM), MediaPipe, Natural Language Processing (NLP), 3D Avatar, Three.js.

1. INTRODUCTION

The global deaf and hard-of-hearing population faces persistent communication barriers in everyday interactions [1]. While assistive technologies have improved accessibility, many existing solutions remain prohibitively expensive or require specialized hardware such as sensor-equipped gloves [2]. Vision-based approaches using standard webcams offer a more scalable alternative, but they present unique challenges including hand scale variation, distance sensitivity, and the need for real-time performance [3]. Simultaneously, effective translation from text to sign language requires more than simple word substitution [4]. ASL possesses its own distinct grammar—omitting articles (a, an, the), auxiliary verbs (am, is, are), and relying on spatial and temporal cues [5]. A naïve word-for-word translation produces unnatural, often incomprehensible signing [5].

This research addresses both directions through an integrated, browser-based system. The key contributions are:

1. A 3D landmark-based recognition pipeline using 63 features (X, Y, Z per landmark) with bounding-box normalization for scale invariance [6].
2. An optimized SVM classifier with hyperparameter tuning via GridSearchCV for lightweight, accurate gesture classification [7].
3. An NLP-driven ASL grammar converter using spaCy for part-of-speech tagging, stop-word removal, and lemmatization [8].

4. A real-time 3D avatar renderer built on Three.js with cross-fading animation sequencing [9].
5. A bidirectional web interface allowing seamless switching between translation modes [10].

2. LITERATURE REVIEW

2.1 Vision-Based Sign Language Recognition

Early sign language recognition systems relied on wearable devices such as sensor gloves and accelerometers, which were often expensive and inconvenient for daily use. Traditional image processing methods later introduced camera-based recognition using skin segmentation, contour detection, and edge analysis. Although these approaches reduced hardware dependency, they were highly sensitive to lighting, background conditions, and user variability. Deep learning models such as CNNs improved gesture recognition accuracy by learning visual patterns automatically, but they required large labelled datasets and significant computational resources [11]. Google's MediaPipe Hands framework provided a more practical solution by extracting 21 real-time hand landmarks from a standard webcam [6]. This lightweight landmark-based approach balances efficiency, affordability, and accuracy for modern web-based sign language systems.

2.2 Coordinate-Based Classification

After hand landmarks are extracted, their X, Y, and Z coordinates can be used as structured numerical features for classification. This method focuses on hand geometry rather than raw images, reducing unnecessary visual complexity and background interference. Traditional machine learning models, especially Support Vector Machines (SVMs), have shown excellent performance on coordinate-based datasets when training data is limited [7]. SVMs are computationally efficient, making them suitable for real-time applications on standard consumer hardware [12]. Bounding-box normalization further improves model robustness by ensuring scale and distance invariance. As a result, coordinate-based classification provides an accurate, lightweight, and scalable solution for sign recognition.

2.3 Natural Language Processing for ASL Grammar

English-to-ASL translation requires grammatical restructuring because ASL follows different linguistic rules than spoken English [5]. ASL commonly removes articles, auxiliary verbs, and certain prepositions while emphasizing topic-comment structure. Direct word-for-word translation often produces unnatural signing and reduces comprehension. NLP tools such as spaCy and NLTK automate this transformation using tokenization, part-of-speech tagging, stop-word removal, and lemmatization [8]. These techniques simplify English input into ASL-compatible sentence structures, improving translation quality. Therefore, NLP plays a critical role in making text-to-sign communication more natural and linguistically accurate.

2.4 3D Avatars in Assistive Technology

Traditional text-to-sign systems often depended on pre-recorded videos, which limited flexibility and required significant storage [13]. WebGL and Three.js introduced dynamic 3D avatar systems capable of rendering signs directly in the browser [9]. These avatars can perform fingerspelling, word-based signs, and smooth animation transitions in real time. Cross-fading and animation blending improve visual continuity and reduce abrupt movements [14]. Compared to video-based systems, 3D avatars reduce bandwidth requirements while increasing scalability and customization. This makes them a highly effective solution for modern assistive communication platforms.

3. METHODOLOGY

This section outlines the software methodology used to develop the proposed bidirectional sign language translation system. It combines computer vision, machine learning, natural language processing, and 3D web technologies within a client-server framework. The process includes capturing sign gestures through webcam input, extracting landmarks, classifying signs using an optimized SVM model, translating them into text, and converting text into structured sign sequences through NLP. Real-time frontend-backend communication and Three.js-based 3D avatar rendering further ensure accurate, efficient, and accessible communication.

3.1 System Architecture Overview

The system follows a client-server architecture [10]. The frontend runs in a standard web browser, handling webcam access, landmark extraction, and 3D rendering [9]. The backend Python Flask server hosts the SVM model and NLP pipeline, exposing REST API endpoints for prediction and translation [15].

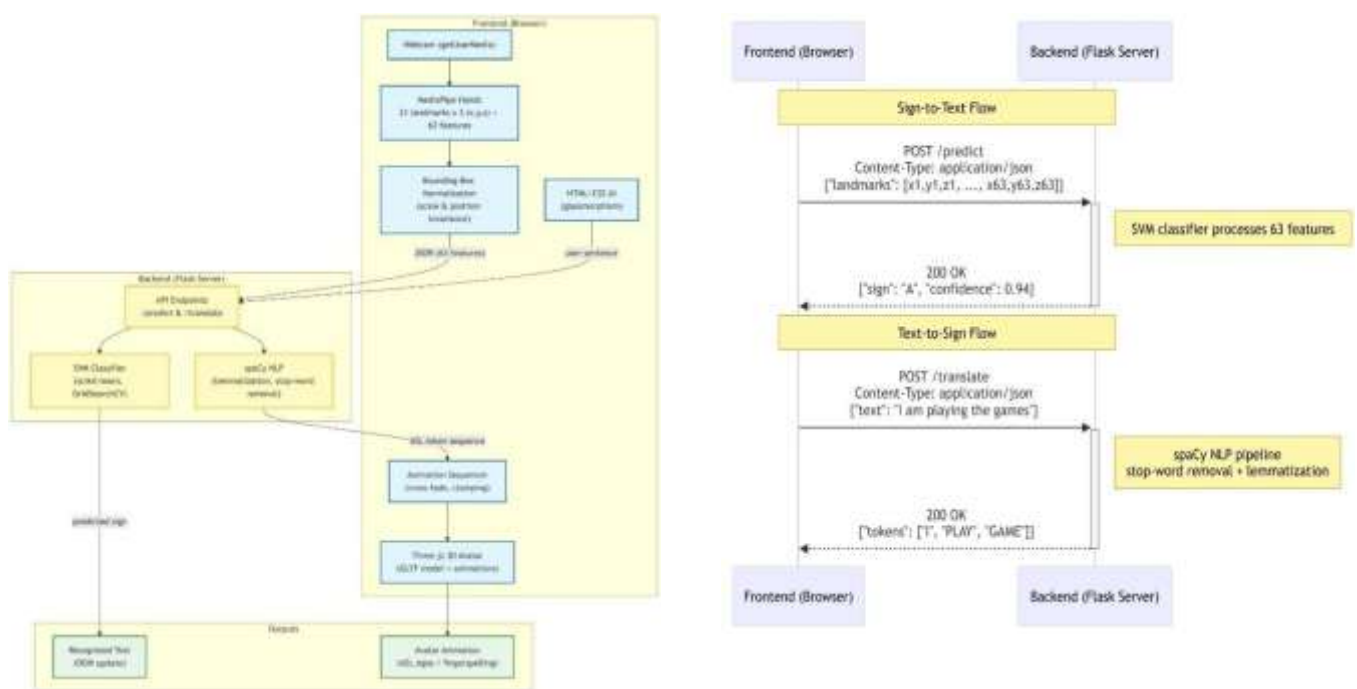


Fig. 3.1 System Architecture Overview

3.2 Dataset Creation and Feature Extraction

A custom dataset was created containing approximately 100 images per ASL letter (A–Z) captured under varying lighting conditions and hand positions [16]. For each image, the MediaPipe Hands pipeline extracts 21 landmarks, each with X, Y, and Z coordinates—producing 63 raw features [6]. To ensure scale and distance invariance, each hand is normalized using bounding-box normalization [17]. This

normalization anchors the hand to a virtual origin and scales it uniformly, allowing the SVM to learn position- and distance-invariant features [17].

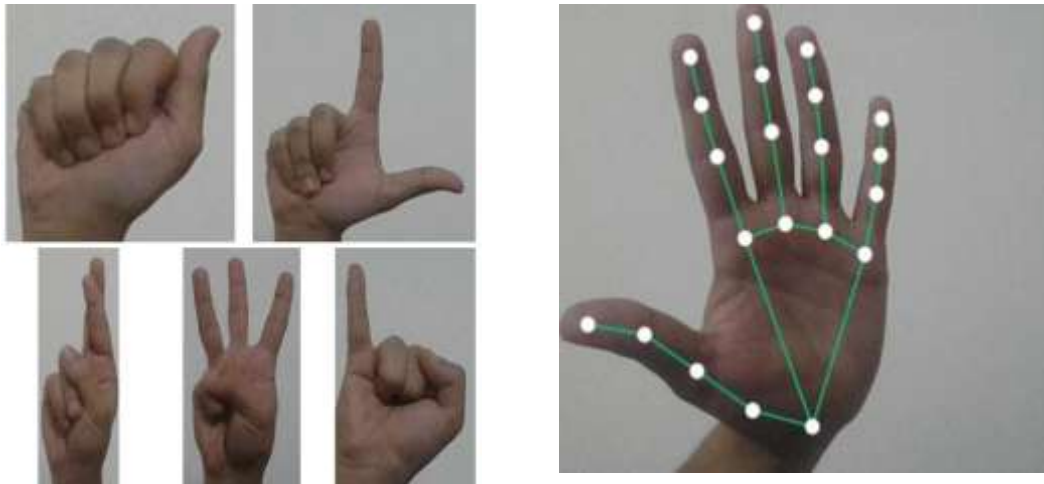


Fig. 3.2 Dataset Creation and Feature Extraction

3.3 Model Training and Optimization

An RBF-kernel Support Vector Machine was used for efficient classification of high-dimensional data with limited samples [7]. GridSearchCV with cross-validation optimized model performance, ensuring accuracy with lightweight deployment [18].

3.4 Sign-to-Text Pipeline (Frontend)

The frontend JavaScript acquires webcam input using the getUserMedia API [19]. Each frame is processed by MediaPipe Hands for landmark extraction [6]. These normalized coordinates are then transmitted asynchronously to the backend for SVM classification [15]. Prediction throttling was implemented to reduce server load while maintaining responsive user feedback [10].

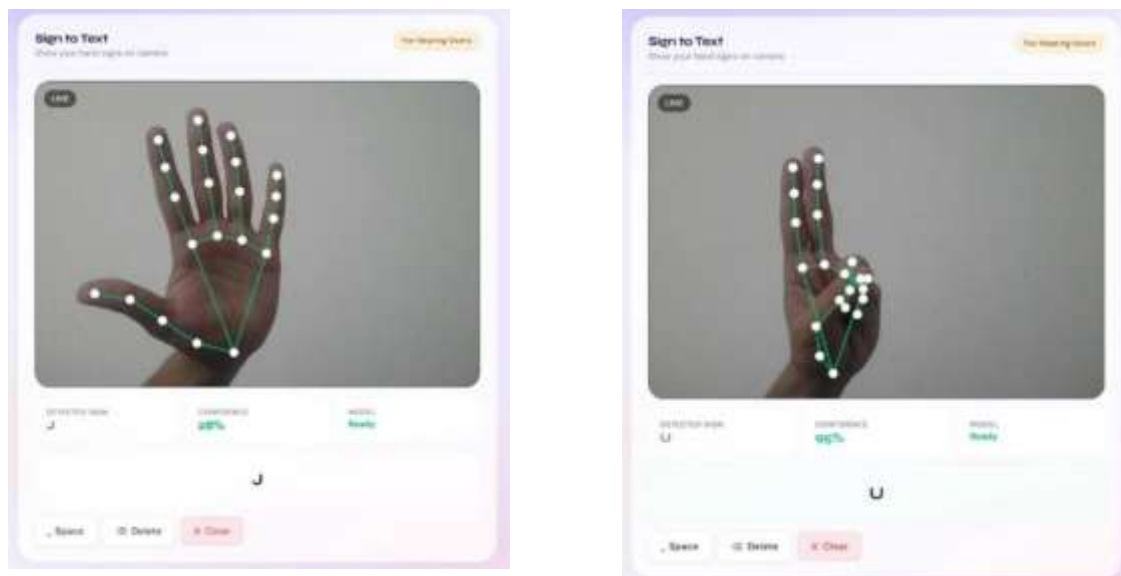


Fig. 3.4 Sign-to-Text Pipeline

3.5 Model Training and Optimization

User-entered English text is processed through the spaCy NLP engine [8]. Tokenization, POS tagging, stop-word removal, lemmatization, and punctuation filtering transform standard English sentences into ASL-compatible token sequences [5], [8]. This significantly improves grammatical realism in generated sign output [5].



Fig. 3.5 Text-to-Sign Pipeline (NLP)

3.6 Avatar Rendering

The frontend initializes a Three.js scene with GLTF model loading, Draco compression, and AnimationMixer sequencing [9]. Individual animation clips represent letters, common words, and idle states [14]. Cross-fading ensures fluid sign transitions while preserving realism [14]. The system dynamically sequences signs word-by-word or letter-by-letter depending on vocabulary availability [9].

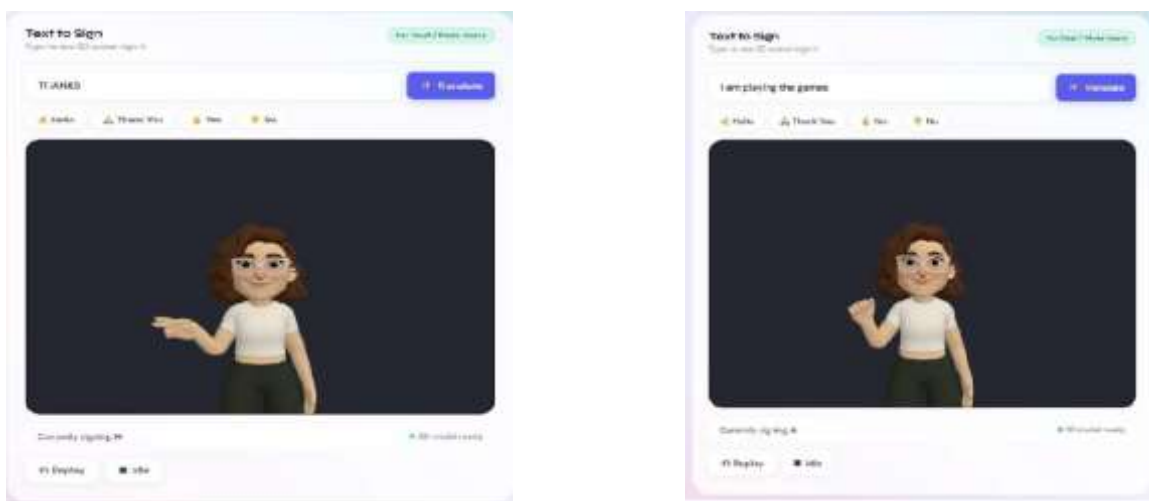


Fig. 3.6 (a) Animation Sequence Example

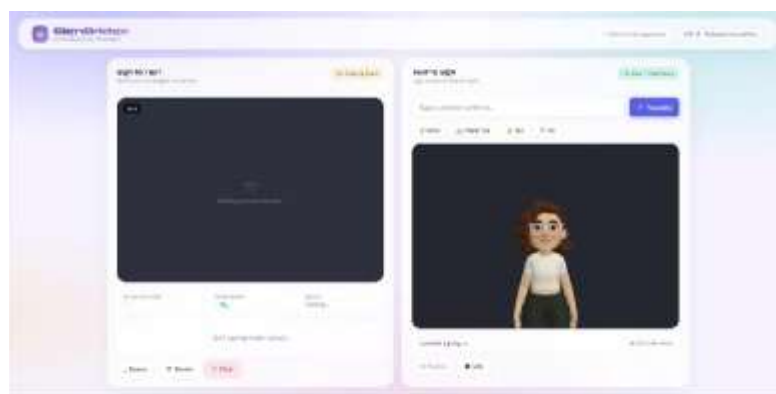


Fig. 3.6 (b) Full Application UI (Desktop View)

3.7 Code Development

```

# main.py
from flask import Flask, request, jsonify, render_template
import numpy as np
import pickle
import os
import time

# Load the spacy English model
# We put it on the top so it only loads into memory once when the server starts
try:
    nlp = spacy.load("en_core_web_md")
except OSError:
    print("Downloading spacy model...")
    from spacy.cli import download
    download("en_core_web_md")
    nlp = spacy.load("en_core_web_md")

app = Flask(__name__)

# ----- Load model.p -----
MODEL_PATH = "model.p" # keep this path to app.py

if not os.path.exists(MODEL_PATH):
    raise FileNotFoundError(f"Model file not found at {MODEL_PATH}")

with open(MODEL_PATH, "rb") as f:
    model = pickle.load(f)

# Your training code here (model)
model = saved_model

@app.route("/")
def home():
    # serve your site at (templates/index.html)
    return render_template("index.html")

@app.route("/api/predict", methods=["POST"])
def api_predict():
    """
    Accepts JSON
    { "features": [F1, F2, ..., FN] }
    where features are exactly:
    """
    # model.predict returns label string (e.g., 'A', 'B', ...)
    try:
        pred = model.predict(x)
    except Exception as e:
        print(f"Error: {e}")
        return jsonify({"error": "Model prediction failed"}), 500

    label = str(pred)

    # optional confidence via predict_proba
    conf = None
    if hasattr(model, "predict_proba"):
        try:
            probs = model.predict_proba(x)[0]
            # find probability of prediction class
            class_index = list(model.classes_).index(label)
            conf = float(probs[class_index])
        except Exception:
            conf = None

    return jsonify({
        "prediction": label,
        "confidence": conf
    })

```

Fig. 3.7(a) Backend Programming Workflow for REST API Integration and NLP Processing

```

# create_dataset.py
import os
import cv2
import numpy as np
import time

# Set up MultiPipe
mp = MultiPipe()
img_paths = mp.paths.get_img_paths(img_dir)
img = cv2.imread(img_path)

# Open a CSV file to write our data
with open(MULTI_PIPE_FILE, mode="a", newline="") as f:
    writer = csv.writer(f)

# Loop through every folder (A, B, C, ...)
for dir_name in os.listdir(DATA_DIR):
    dir_path = os.path.join(DATA_DIR, dir_name)
    if not os.path.isdir(dir_path):
        continue

    print(f"Processing folder: {dir_name}...")

    # Loop through every image in that folder
    for img_name in os.listdir(dir_path):
        img_path = os.path.join(dir_path, img_name)
        img = cv2.imread(img_path)

        # MultiPipe needs RGB images, not BGR (OpenCV's default)
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        results = mp.process(img_rgb)

        # If results multi hand landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            # 1. Extract X, Y, and Z coordinates
            x, y, z = [landmark.x for landmark in hand_landmarks.landmark]
            # 2. Find the bounding box for X and Y
            min_x, min_y = min(x), min(y)
            max_x, max_y = max(x), max(y)

            width = max_x - min_x
            height = max_y - min_y
            scale = max(width, height)

            # 3. Apply Advanced Normalization (including Z)
            features = []
            for i in range(len(hand_landmarks.landmark)):
                normalized_x = (hand_landmarks.landmark[i].x - min_x) / scale
                normalized_y = (hand_landmarks.landmark[i].y - min_y) / scale
                # Z is relative to the wrist origin in MultiPipe, so we just scale it
                normalized_z = (hand_landmarks.landmark[i].z - z) / scale

                features.append(normalized_x)
                features.append(normalized_y)
                features.append(normalized_z) # NEW! Add Z to the list

            # 4. Save to CSV: label first, then the 21 features
            row = [dir_name] + features
            writer.writerow(row)

    print(f"Done! All data saved to {MULTI_PIPE_FILE}")

```

Fig. 3.7 (b) Feature Extraction and CSV Generation Process

3.8 Block diagram

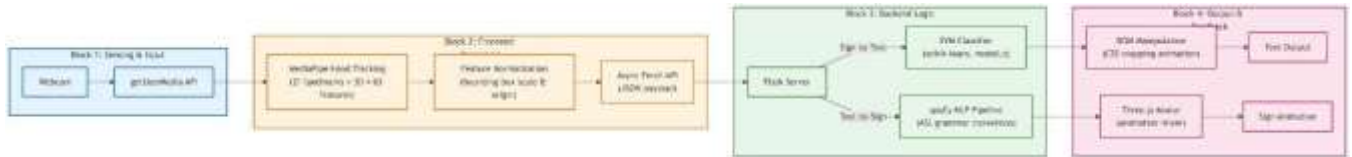


Fig. 3.8 Block Diagram of Bidirectional Sign Language Translation System Architecture

3.9 Flowchart diagram



Fig. 3.9 Workflow Diagram of Sign-to-Text and Text-to-Sign Translation Process

3.10 Software Components

This section describes the key software technologies used for system development, real-time communication, gesture recognition, machine learning, and 3D visualization.

S. No.	Software/Tool	Type	Function
1	HTML	Markup Language	Creates the basic structure of the webpage, including video feed, buttons, and interface containers
2	CSS	Styling Language	Designs and enhances webpage appearance using layouts, colours, and visual effects
3	JavaScript	Programming Language	Controls frontend logic, user interactions, and API communication
4	WebRTC (getUserMedia API)	Browser API	Captures real-time webcam video input for gesture detection
5	MediaPipe Hands	Computer Vision Library	Detects and tracks 21 hand landmarks for sign recognition
6	Three.js	JavaScript Library	Renders and animates the 3D avatar in the web browser
7	WebGL	Graphics Rendering API	Provides GPU-accelerated 3D rendering for browser-based visualization
8	Flask	Python Framework	Develops backend REST APIs for prediction and data processing

9	SVM / Random Forest	Machine Learning Models	Classifies hand gestures into corresponding sign language outputs
10	spaCy	NLP Library	Processes text through tokenization, lemmatization, and ASL grammar conversion
11	REST API	Communication Interface	Connects frontend and backend through HTTP request-response handling
12	JSON	Data Format	Enables structured data exchange between client and server
13	Blender	3D Modeling Software	Designs, rigs, and animates the 3D avatar for sign gestures

4. RESULTS

4.1 Classification Accuracy

The SVM model demonstrated strong classification accuracy on held-out test data, validating the effectiveness of structured landmark-based gesture recognition [7]. By using 63 normalized hand landmark features, the system achieved reliable distinction between multiple ASL hand signs [17]. Bounding-box normalization significantly improved consistency by reducing errors caused by hand scale and camera distance variation [17]. Compared to traditional 2D coordinate baselines, the inclusion of Z-axis depth improved recognition of visually similar gestures with subtle spatial differences [6]. This was particularly beneficial for signs with overlapping finger structures but different depth positioning. Overall, the results confirm that combining MediaPipe landmarks with SVM classification offers an accurate and computationally efficient recognition framework.

4.2 Real-Time Performance

The system maintained practical real-time responsiveness on standard consumer-grade hardware without requiring GPU acceleration [6]. End-to-end latency from webcam capture to prediction output remained low enough for smooth communication [15]. Client-side landmark extraction through MediaPipe reduced backend processing overhead and improved scalability [10]. Prediction throttling further optimized performance by preventing excessive API requests while preserving stable user interaction [10]. This balance between responsiveness and computational efficiency allowed continuous operation on laptops and web browsers. The lightweight design makes the system suitable for accessible browser-based assistive communication.

4.3 NLP Translation Efficiency

The spaCy NLP pipeline processed English text rapidly, enabling near-instant transformation into ASL-compatible token sequences [8]. Tokenization and part-of-speech tagging effectively identified

unnecessary grammatical components such as determiners and auxiliary verbs [5]. Stop-word removal reduced sentence complexity and minimized redundant avatar animations. Lemmatization further improved efficiency by converting words into simpler root forms for easier sign mapping [8]. These transformations enhanced both processing speed and linguistic realism. As a result, the NLP system significantly improved the naturalness and practicality of text-to-sign translation.

4.4 Animation Fluidity

The Three.js avatar delivered smooth real-time rendering performance across desktop and mobile-compatible browsers [9]. AnimationMixer sequencing allowed precise control over sign playback, ensuring consistent gesture timing. Cross-fading between animation clips eliminated abrupt transitions and improved visual continuity [14]. Idle-state integration also helped maintain a more natural avatar presence between sign sequences. Dynamic word-level and letter-level rendering increased flexibility for both vocabulary-based and fallback fingerspelling communication [9]. Together, these features created a more immersive, understandable, and user-friendly sign language visualization system.

5.DISCUSSION

5.1 Technical Challenges and Solutions

One major technical challenge involved coordinate shape mismatch between frontend and backend processing modules, where inconsistent feature dimensions initially caused prediction errors [10]. This issue was resolved by standardizing 63-feature landmark extraction across dataset creation, frontend normalization, and backend classification pipelines. Another challenge was scale sensitivity, as variations in hand distance from the webcam significantly affected recognition consistency. Bounding-box normalization successfully addressed this by ensuring scale and distance invariance across different user positions [17]. Lighting variation and camera angle differences also introduced occasional landmark instability, requiring careful preprocessing adjustments [6]. Additionally, dynamic ASL letters such as J and Z remain difficult because the current system uses static frame-by-frame classification rather than temporal sequence modelling [20]. These solutions improved reliability, but they also highlighted important areas for future enhancement.

5.2 Design Choices

Support Vector Machine (SVM) was selected over deep learning architectures because smaller datasets are more vulnerable to overfitting when used with large neural networks [7]. SVM provided strong performance on structured landmark coordinates while maintaining computational efficiency on standard consumer devices. MediaPipe was integrated on the client side to perform browser-based hand landmark extraction, significantly reducing backend server workload [6]. This design improved scalability by distributing computational tasks more effectively between frontend and backend systems. Prediction throttling was intentionally implemented to prevent excessive server requests and reduce repetitive unstable outputs [10]. Together, these design choices prioritized lightweight deployment, accessibility, and real-time performance over computational complexity.

5.3 Future Improvement Opportunities

Future improvements can significantly expand the system's capabilities by integrating temporal deep learning models such as LSTM or GRU for dynamic gesture recognition [20]. Expanding from single-

hand to dual-hand tracking would improve vocabulary complexity and real-world ASL coverage. More advanced NLP frameworks could also improve contextual understanding and sentence-level sign grammar generation [8]. Avatar systems may benefit from facial expression synthesis and emotional gesture rendering for greater linguistic realism [13]. Mobile optimization would further broaden accessibility by making the system available on smartphones and tablets. These advancements could transform the platform into a more universal, intelligent, and socially impactful communication system.

5.4 Limitations

Despite its effectiveness, the current system has several limitations that impact broader sign language coverage. Static-only recognition restricts the system's ability to identify motion-based gestures such as J and Z, which require temporal analysis [20]. Single-hand tracking also limits recognition of more advanced ASL signs that depend on two-handed coordination [6]. The text-to-sign module relies on predefined animation libraries, which constrains vocabulary flexibility and expressive depth [14]. Additionally, MediaPipe landmark detection can become less reliable under poor lighting, heavy background clutter, or extreme camera angles [6]. Addressing these limitations will require temporal modeling, expanded gesture libraries, and more adaptive environmental robustness in future system versions.

6. CONCLUSION AND FUTURE WORK

This paper presented a comprehensive bidirectional sign language translation system that combines real-time hand tracking, MediaPipe landmark extraction, SVM-based gesture classification, NLP-driven ASL grammar transformation, and Three.js-powered 3D avatar rendering into a unified browser-based platform [6], [7], [8], [9]. By integrating both Sign-to-Text and Text-to-Sign communication modes, the system addresses two-way interaction barriers between hearing and deaf or hard-of-hearing individuals. The use of standard webcam hardware and browser-native technologies demonstrates that effective assistive communication can be achieved without expensive sensors, specialized gloves, or high-end computational infrastructure [10]. Bounding-box normalization, structured coordinate classification, and NLP preprocessing collectively improved recognition reliability, grammatical accuracy, and communication realism. Additionally, the system's lightweight architecture makes it scalable, affordable, and accessible for educational, professional, and everyday communication environments. Overall, this research highlights the potential of combining artificial intelligence, web technologies, and assistive design principles to reduce communication inequalities and promote inclusive digital interaction.

Future work will focus on several important enhancements to further improve system capability and real-world applicability:

1. Temporal modelling through Long Short-Term Memory (LSTM) or other sequence-based neural networks to recognize dynamic gestures such as J and Z, as well as more complex motion-dependent signs [20].
2. Vocabulary expansion by increasing the animation and linguistic database to include a wider range of ASL words, phrases, and contextual expressions for richer conversational communication [14].
3. Mobile-first optimization to improve smartphone and tablet compatibility, enabling broader accessibility and on-the-go communication support through portable devices [19].
4. Two-hand gesture recognition and multilingual sign language adaptation to support more advanced signing structures and regional sign language variations.

5. Enhanced avatar realism through facial expression synthesis and emotional gesture rendering, which are essential for natural and context-aware sign communication.

With these future developments, the proposed system has the potential to evolve into a more intelligent, universally accessible, and socially impactful assistive communication platform.

7. REFERENCES

- [1] World Health Organization (WHO). (2021). World Report on Hearing.
- [2] R. Rastgoo, K. Kiani, & S. Escalera. (2021). Sign Language Recognition: A Deep Survey. Expert Systems with Applications.
- [3] G. Bradski. (2000). The OpenCV Library.
- [4] D. Jurafsky & J. H. Martin. (2023). Speech and Language Processing.
- [5] W. Sandler & D. Lillo-Martin. (2006). Sign Language and Linguistic Universals. Cambridge University Press.
- [6] C. Lugaresi et al. (2019). MediaPipe: A Framework for Building Perception Pipelines. arXiv:1906.08172.
- [7] C. Cortes & V. Vapnik. (1995). Support-Vector Networks. Machine Learning.
- [8] M. Honnibal & I. Montani. (2017). spaCy 2: Natural Language Understanding.
- [9] J. Dirksen. (2018). Learn Three.js: Programming 3D Animations and Visualizations for the Web.
- [10] M. Grinberg. (2018). Flask Web Development. O'Reilly.
- [11] Y. LeCun, Y. Bengio, & G. Hinton. (2015). Deep Learning. Nature.
- [12] T. M. Mitchell. (1997). Machine Learning. McGraw-Hill.
- [13] M. Kipp. (2011). Gesture Generation by Animated Conversational Agents.
- [14] K. Matsuda & R. Lea. (2013). WebGL Programming Guide.
- [15] Mozilla Developer Network (MDN). (2024). Fetch API Documentation.
- [16] M. Costello. (2008). Random House Webster's American Sign Language Dictionary.
- [17] R. Szeliski. (2022). Computer Vision: Algorithms and Applications.
- [18] F. Pedregosa et al. (2011). Scikit-learn: Machine Learning in Python.
- [19] Mozilla Developer Network (MDN). (2024). MediaDevices.getUserMedia() Documentation.
- [20] S. Hochreiter & J. Schmidhuber. (1997). Long Short-Term Memory.