Team 6: Visual Speech-Aware Perceptual 3D Facial Expression Reconstruction from Videos

Thanmai B, Hemalatha M, Sri Sravan Kumar M, Puneeth P

INTRODUCTION

Our project focused on reconstructing highly accurate 3D facial expressions from 2D video input, especially during spoken communication. By leveraging visual speech cues such as lip movements and jaw positioning, we aimed to produce expressive and perceptually natural 3D facial animations. This process enhances realism and can significantly benefit applications in virtual avatars, human-computer interaction, animation, and telepresence. While the concept was exciting, the development journey presented multiple technical and practical challenges that we had to systematically overcome.

CHALLENGES

Environment Setup and Dependency Issues

Setting up the development environment was one of the earliest hurdles. We initially attempted to work locally but encountered Torch-CUDA compatibility issues. After multiple failed attempts to install PyTorch3D and other dependencies manually, we pivoted to Google Colab. Although Colab provided GPU access and convenience, we still had to resolve several configuration problems such as unsupported CUDA versions and runtime disconnections.

Additionally, the provided environment files were not usable, which forced us to manually install dependencies. This process involved considerable trial and error, especially when dealing with version conflicts and obscure package requirements. Our inexperience with GitHub project structures also led to issues with pathing and module imports, which slowed down initial development.

Dataset Access and Usage

Another key challenge was obtaining the required datasets. Access to benchmark datasets like LRS3 and MEAD required completing agreement forms and waiting for permissions, which

delayed our early testing. Once accessed, these datasets proved invaluable for evaluation, benchmarking, and customization of pre-trained models.

We specifically used:

- LRS3: For lip-reading data across diverse speakers.
- MEAD: To improve emotional accuracy and realism in 3D facial expressions.

Since we were not training models from scratch, the datasets were mainly used for fine-tuning, testing, and demonstrations of real-world variability.

PRE-TRAINED MODEL INTEGRATION

We used SPECTRE, a pre-trained model, to extract 3D facial geometry from video frames. However, working with pre-trained components like DECA, 3DMM-based encoders, and feature extractors (ResNet/CNNs) brought their own difficulties. Model outputs were often misaligned or lacked precision in unfamiliar scenarios, requiring additional tuning.

The implementation pipeline included:

1. Input video → frame extraction
2. Pre-trained model loading
3. Face cropping & warping
4. Chunk-wise processing

IMPLEMENTATION

During this project, we encountered several challenges that required significant time and effort to resolve. The first challenge arose when we started setting up the project environment on Google Colab. Initially, Colab seemed like a straightforward solution to get quick access to GPUs without local setup overhead. However, we quickly ran into problems with PyTorch3D installation and CUDA version mismatches. Pre-built wheels for PyTorch3D were not available for Colab's environment, and manual builds failed repeatedly due to missing dependencies. These repeated failures cost us a significant amount of time.

After considerable research, we discovered that Colab's dynamic backend updates (frequent changes in Python, CUDA, and driver versions) were causing major instability for our deep

learning stack. Moreover, runtime disconnections and resource limitations (such as short maximum session times) made it difficult to handle large datasets like LRS3 and MEAD efficiently. These issues made it clear that we needed a more stable and persistent environment.

To overcome this, we transitioned our project to Lightning AI, a platform designed for scalable deep learning development. Setting up on Lightning AI wasn't entirely straightforward either — initially, we had to carefully define the environment by explicitly specifying the versions of PyTorch, PyTorch Lightning, PyTorch3D, and CUDA compatibility to ensure that everything worked together seamlessly. But once the setup was stabilized, it significantly improved our workflow: we had persistent access to GPU nodes, eliminated random disconnects, and could handle larger datasets more effectively.

After resolving the environment issues, the next stage was setting up the data pipeline for facial reconstruction. We started by processing input videos to extract frames using OpenCV. Each frame was passed through a face detector to crop and warp the face region for normalization. Although this step seemed simple, we initially faced some challenges with inconsistent cropping due to varying head poses in the videos. We fine-tuned the cropping parameters based on face bounding box ratios to ensure more consistent pre-processing.

Loading the pretrained models was the next hurdle. We integrated the SPECTRE model along with facial landmark extractors like DECA and 3DMM encoders into the pipeline. There were initial failures when passing certain video frames into the model — some frames failed to produce valid meshes due to poor lighting or extreme poses. To fix this, we added a frame quality filter that skipped frames where face detection confidence was below a threshold, improving the overall reconstruction quality.

The processing pipeline was modularized into chunks to efficiently handle large videos. Chunking helped avoid GPU memory overflow by splitting videos into manageable frame groups and reconstructing them sequentially.

Once the core 3D reconstruction was functional, the next challenge was linking the output back to a visual display. Initially, the reconstructed 3D meshes were stored frame by frame but weren't automatically stitched into a playable animation. To solve this, we wrote a post-processing module using ffmpeg that merged individual frames into a video output synchronized with the original audio.

Finally, we built a basic front-end interface using Streamlit to allow users to upload a video and view the generated 3D facial animation. However, integrating dynamic input handling posed an issue: the pipeline originally assumed static file paths, meaning every new input required manual changes. We resolved this by using Python's subprocess library to dynamically adjust file paths and invoke reconstruction commands based on user-uploaded content.

During front-end development, one last issue emerged: although the 3D video was being generated, it wasn't displaying correctly on the Streamlit app. After investigating, we discovered

that the video file path was not being properly refreshed after generation. Updating the reference path dynamically and ensuring file caching was handled properly finally fixed the issue. We also added a download option allowing users to save their generated 3D videos directly from the web app.

Thus, through persistent debugging and environment adaptation, we successfully implemented a robust system capable of reconstructing realistic, perceptual 3D facial expressions from video inputs.

CONCLUSION

Despite numerous setbacks related to setup, dependencies, dataset access, and model integration, we successfully developed a working system that reconstructs 3D facial expressions from video. Leveraging visual speech cues allowed us to generate natural, emotion-aware facial animations that move beyond traditional modeling. This project not only sharpened our technical skills but also prepared us to handle real-world constraints in deploying complex machine learning pipelines.

Our success was a result of adaptability, persistence, and collaborative problem-solving, and we believe this lays a promising foundation for future research and applications in expressive virtual communication.