21st OCTOBER 2024

IMAGE PROCESSING & COMPUTER VISION

3D Tongue Tip Tracking

Federico Mura, f.mura@student.utwente.nl, s3497313, MSc Robotics
Federico Ugolini, f.ugolini@student.utwente.nl, s3542882, Electrical Engineering
Celia Gómez Campo, c.gomezcampo@utwente.nl, s3548589, MSc Biomedical Engineering
Peixuan Wu, p.wu-2@student.utwente.nl, s3505235, MSc Computer Science

***Abstract*—This project presents a 3D tracking system for monitoring tongue movement to assess rehabilitation progress in patients undergoing oral surgery or radiotherapy. A three-camera setup captures multiple angles, and the Kanade-Lucas-Tomasi (KLT) optical flow method is used to track the tongue tip. A 3D coordinate system, centered at the tip of the patient's nose, allows for precise spatial measurements. Tracked tongue positions are recorded from Left-Middle and Right-Middle stereo views, with accuracy evaluated by comparing coordinate errors between these views. The system demonstrates reliable tracking, with user intervention required when tracking precision is challenged.**

***Keywords*—Camera Calibration, 3D Reference Coordinate System, Tracking Facial Features, Stereo Videos**

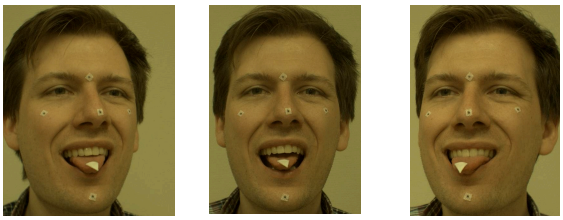

Figure 1. Patient recorded from three positions

I. Introduction

Patients undergoing surgery or radiotherapy in the oral region, particularly the tongue, often experience reduced tongue mobility. Such limitations can severely impact essential oral functions, including speech, swallowing, food transport, and chewing. To support rehabilitation efforts, clinicians monitor the tongue's "Range of Motion" (ROM) as an indicator of functional recovery. By measuring ROM before and after treatment, it becomes possible to assess how different surgical or therapeutic interventions affect oral functions over time.

The main challenge is accurately capturing and analyzing tongue movement in three dimensions to provide consistent and reliable measurements. This requires a precise method for tracking the tongue's position relative to the patient's head. For this purpose, a triple-camera system records patients as they move their tongue to specific, standardized positions: left, right, forward, downward, and upward. However, to make meaningful comparisons between pre- and post-treatment data, the recorded tongue positions must be expressed within a stable, head-fixed coordinate system. Without such a reference, variations in patient posture or camera positioning could reduce the accuracy of the results, limiting their usefulness for clinical evaluation.

This report addresses this problem by developing a method to track the tongue tip and select facial landmarks, such as the nose and chin, to establish a reliable 3D coordinate system centered at the tip of the nose. This system enables precise, repeatable measurements of tongue motion within a head-fixed framework, facilitating accurate comparison of movement over time. The report outlines the system's development process, explains the video-based tracking method, and

evaluates the method's effectiveness. Limitations of the approach are also discussed, along with recommendations for future improvement, ensuring this method can better support clinical assessments of oral rehabilitation progress.

A. Materials

Three videos capturing the subject from left, center, and right positions, as shown in figure 1. The subject has small markers placed on the chin, nose, tongue tip, cheeks, and between the eyebrows for precise tracking. Each camera view also includes a folder of calibration images featuring a 10mm x 10mm checkerboard pattern for stereo calibration. For software we use Python 3.10, utilizing libraries such as OpenCV (cv2), NumPy, dlib, and glob for computer vision processing and data import.

B. Methods

1) Analysis:

The aim of this project is to track tongue movement in three dimensions accurately, which can assist in monitoring the rehabilitation of patients after oral treatments. The tracking system is designed within a fixed 3D coordinate system attached to the nose tip, allowing consistent comparison of tongue movement across different times and sessions. The method uses three-camera stereo views to achieve reliable 3D tracking of the tongue tip.

The primary inputs for this system include video sequences from three cameras positioned to capture left, middle, and right views of the patient's face. Calibration data for each camera pair (left-middle and middle-right) is also provided to ensure accurate 3D mapping of tracked points. The desired output is a series of 3D coordinates representing the tongue tip's position over time, referenced to a coordinate system centered at the nose tip.

To achieve precise 3D tracking of the tongue tip, the following steps are followed:

1. Stereo Calibration: Use calibration matrices to project 2D pixel coordinates in each camera view into 3D space.
2. Feature Detection: Identify and initialize the tracking of the nose, chin, and tongue tip in the first frame as initial reference points.
3. Tracking: Use KLT optical flow method to monitor these reference points across each video frame, updating their locations as the frames progress.
4. Triangulation: Calculate the 3D coordinates of the tongue tip using data from each stereo view.
5. Reference coordinate system: Create a reference coordinate system attached to the face
6. Evaluation: Compare the 3D coordinates obtained for each different stereo pair to evaluate tracking accuracy and consistency.

We conduct stereo calibration for each pair of cameras to precisely acquire the extrinsic parameters, establishing the spatial relationship between them. Then we use dlib to detect the landmarks on the face, and using the distance, between landmarks 9 and 52 (Figure 2), for detecting if the mouth is open or closed, using the thresholds, as well as to identify if the tongue movement is unusually large or small.(Mansi, 2022; Rosebrock, 2017) This feedback system helps adjust the tracking during runtime and handles situations where automatic tracking might fail. When the mouth is open we can select the tip of the tongue in order to track it. For the track we use Lukas-Kanade-Tomasi Optical Flow, where:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v = -\frac{\partial I}{\partial t}$$

Also we have an update for the threshold through the prompt, given by the fact that we have the tongue out, but without the teeth. The 2D coordinates of the tongue tip are captured for each frame. Triangulation is then used to estimate the 3D coordinates of the tongue tip by combining the 2D tracked coordinates from the stereo pairs.

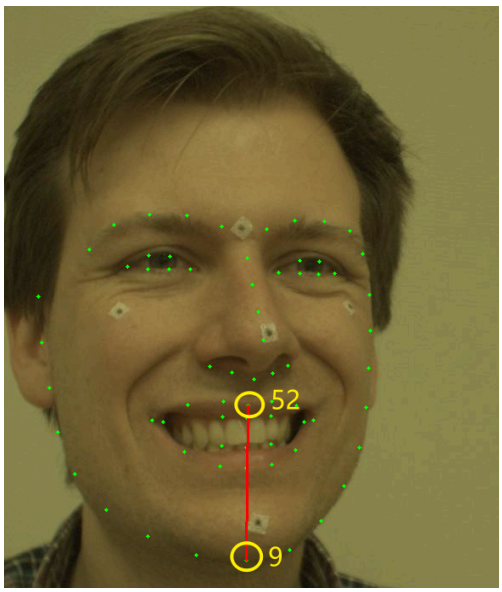

Figure 2. Landmarkers

For each frame, two independent 3D coordinates are generated for the left-middle and middle-right camera pairs. However, as these 3D points are relative to the camera's viewpoint, we define a reference system. The nose tip served as the origin, with the direction between the cheek points forming the Y-axis, and the normal vector of the plane formed by the cheek and forehead points defining the X-axis. The Z-axis direction was then calculated based on orthogonality.(Hu et al., 2009) In this framework, we achieve continuous tracking of the nose tip, chin, and tongue, creating a robust basis for interpreting facial movements and tongue dynamics. Then we use the points of both LM and RM with the new reference system and use it to create the 3D graphs. To evaluate the performance of this method we compute the error between the point of LM and the point of RM. Ideally, differences between corresponding coordinates should be zero, indicating perfect alignment. To assess reliability, we calculate the standard deviation of each coordinate, where lower values indicate greater consistency.

2) Performance & Accuracy evaluation:

The accuracy assessment involved comparing 3D coordinates from each stereo pair for each frame and calculating positional errors along the X, Y, and Z axes. Mean Absolute Error (MAE) and Standard deviation was used as the primary metric to represent average positional discrepancy between the stereo views, with standard deviation calculations providing insight into tracking consistency across frames.

$$\text{Error}_X = |T_{3D,LM,X} - T_{3D,MR,X}|$$

III. Results

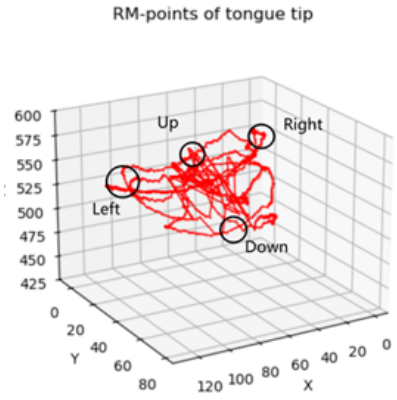

Figure 3. tracking of the tip of the tongue in RM camera systems

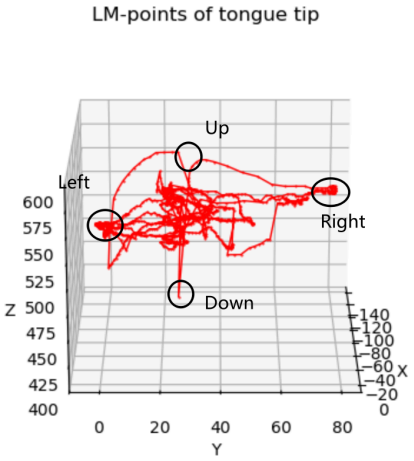

Figure 4. tracking of the tip of the tongue in LM camera systems

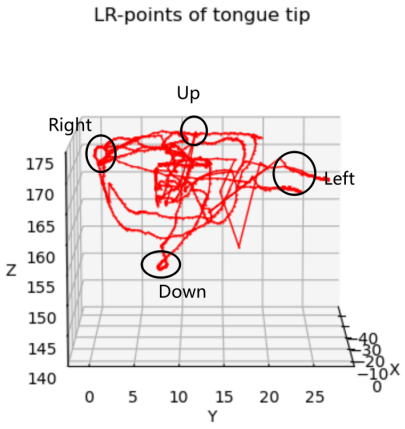

Figure 5. tracking of the tip of the tongue in LR camera systems

All three systems roughly captured the tongue tip's trajectory, but discrepancies with real movement made evaluation challenging. To address this, a facial reference coordinate system was established based on the 3D points obtained from the RM camera system. Subsequently, similar reference coordinate systems were established using the 3D points obtained from the LM and RM camera systems, respectively. The coordinates from each system were then projected into the video. This 3D reference coordinate system was projected onto the left, middle and right camera's video, as shown in the figure 6.

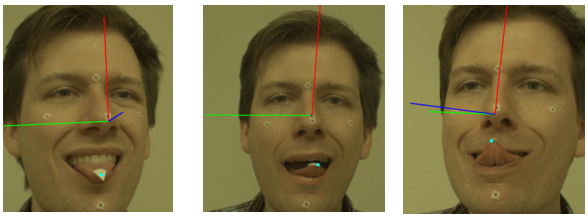

Figure 6. the reference coordinate system attached to face

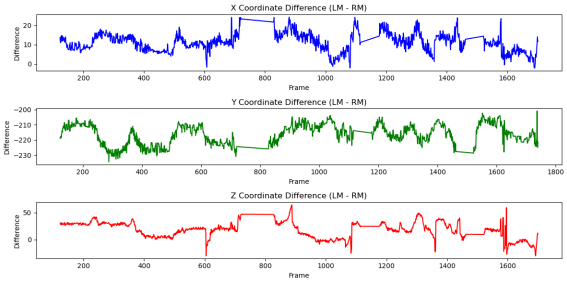

Figure 7. The error of axis between the reference coordinate system from LM and RM system

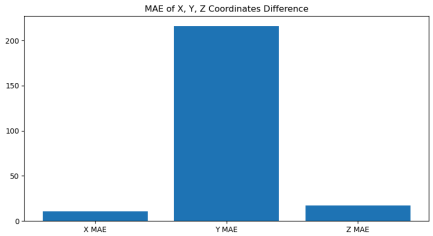

Figure 8. MAE of X,Y,Z coordinate difference

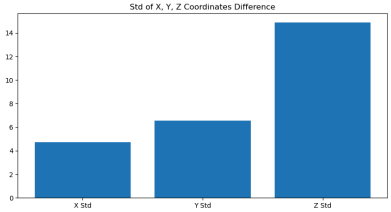

Figure 9. Standard deviation of X,Y,Z coordinate difference

IV. Discussion

The obtained trajectory of the tongue tip effectively captured its movement; however, we observed that while the actual trajectory is nearly symmetrical, our recorded data was not entirely consistent. Some frames showed deviations in the recorded positions. When using optical flow to track points

that move rapidly, the tracking results tended to lag behind the actual positions. Additionally, as the tongue tip approached the point of disappearing from view, occlusion by the lips caused significant tracking errors, resulting in overly dense points in the middle region of the trajectory. Furthermore, when the tongue tip moved quickly past the lips, the tracking could fail.

We employed a combination of manual annotation and automated processing using the OpenCV library, which improved accuracy and proved more reliable than using OpenCV alone. The key aspect for evaluation was assessing how the points obtained from different camera coordinate systems differed once converted into a common facial reference coordinate system. Ideally, these differences should be zero; however, due to recording biases and calibration errors, discrepancies were inevitable.

Discrepancies along the Y-axis were notably large, which could be attributed to calibration errors that amplified along the line connecting the two cheek points, making this axis particularly sensitive to minor angular deviations between cameras. Although the Y-axis, representing the horizontal direction across the face, showed a relatively small standard deviation and stable coordinates, there remained a consistent difference in the coordinate points obtained from the two camera systems. This suggests potential calibration inconsistencies or inherent discrepancies between the systems, which warrant further investigation. Tracking errors for points moving horizontally further exacerbated these discrepancies, and occlusions during tongue movements contributed to the instability in the Y-axis data.

For the Z-axis, which corresponds to the vertical direction (aligned with the upward direction in the image plane), the variability was more pronounced. This indicates that measurements along the height direction were less stable across different viewpoints. Such variability could arise from calibration precision issues and the sensitivity of vertical movements to camera alignment. Minor calibration inaccuracies could amplify these fluctuations. Additionally, limitations in viewing angles and potential occlusions when capturing vertical movements, along with image noise and lighting variations, further contributed to the instability of the Z-axis data.

The figure of standard error indicates that the X-axis, representing the depth direction (perpendicular to the camera plane), has the lowest standard deviation, suggesting that the tracking system performs best when measuring depth. This reflects the high accuracy in capturing depth information, possibly due to well-calibrated camera systems and consistent stereo disparity results.

One significant limitation of this study is the exclusive use of the optical flow method for tracking the movement of points. While this approach was effective in certain scenarios, it did not incorporate alternative tracking algorithms that could potentially offer more robust performance, especially during rapid movements or complex scenarios. Additionally, the combination of manual and automated point recording proved to be time-consuming and labor-intensive. The process required manual definition and frequent updates of the thresholds for mouth opening and closing based on the actual conditions, which, although increasing accuracy, significantly slowed down the processing speed for each video.

Another limitation is the external factor of the experimental setup. The background light on the screen was green, which may have led to low contrast in the images. This lower contrast could interfere with point tracking accuracy, as the detection of feature points could be influenced by background interference. Although this aspect was not thoroughly tested in the current study, it may have contributed to some level of inaccuracy in the results.

In future work, we plan to integrate multiple tracking algorithms to enhance the robustness and accuracy of point tracking. By incorporating methods such as deep learning based tracking or SIFT tracking algorithms(Mozaffari & Lee, 2020; Zhu et al., 2019), we aim to address the limitations we observed during rapid movements and

instances of occlusion. Additionally, we intend to automate the process further by developing a system that can dynamically adjust thresholds for mouth opening and closing without manual intervention. This will streamline the workflow, reduce human error, and significantly improve the processing efficiency of our system.We also plan to improve the experimental setup by optimizing the background to use colors with higher contrast and implementing backlight compensation techniques. These changes are expected to reduce the potential impact of background interference and improve the accuracy of point tracking. Furthermore, we aim to apply this method to different experimental scenarios and conditions to test its generalizability and adaptability, ensuring that the system is effective across a broader range of applications.

V. Conclusion

A tracking system was developed to monitor the movement of the tongue tip in video footage, intended to assess the rehabilitation progress of patients recovering from recent oral surgery. Utilizing the Lucas-Kanade-Tomasi (LKT) algorithm with Python, the system tracks the tip of the tongue. While the system effectively follows the chosen points, it occasionally prompts the user to confirm the tongue's position or to change the threshold. This feature enhances tracking accuracy, though it requires occasional user input, limiting the system's full autonomy.

The primary objective of this project—creating a semi-autonomous tongue-tracking system—has been successfully realized to a reasonable extent. Future work could be to implement this tracking without the limitation of having input from the user.

Python CODE FOR THE RESEARCH

```
import cv2 as cv
import dlib
import numpy as np

detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

# create a VideoCapture object
cap = cv.VideoCapture('./subject1/proefpersoon 1.2_L.avi')

# Parameters for the Lucas-Kanade optical flow method
lk_params = dict(winSize=(20, 20), maxLevel=2, criteria=(cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 20, 0.03))

# Check if the video opened successfully
if not cap.isOpened():
    print("can't open the video")
    exit()

# Open the file in write mode
with open("./Points/tongue_LeftTracking_L1_2_test.txt", "w") as log_file:
    # log_file.write("Frame      Tongue_X        Tongue_Y\n")

    # Variables to control playback state
    paused = False
    tongue_visible = False
    tongue_point = None
    prev_tongue_point = None

    # Define some distance thresholds
    mouth_open_threshold = 203  # Threshold to detect if the mouth is open
    mouth_close_threshold = 175  # Threshold to detect if the mouth is closed
    tongue_movement_threshold = 30  # Threshold for excessive tongue tip movement


    tongue_movement_min_threshold = 0.01  # Threshold for minimal tongue tip movement

    # Frame counter
    frame_counter = 0
    check_frames_interval = 20  # Check every 20 frames
    check_tracking = True
    frame_cnt = 0

    # Define a function to manually select the tongue tip point
    def select_tongue_point(frame):
        roi = cv.selectROI("Select Tongue Point", frame, fromCenter=False, showCrosshair=True)
        cv.destroyAllWindows()
        x = int(roi[0] + roi[2] / 2)
        y = int(roi[1] + roi[3] / 2)
        return np.array([[x, y]], dtype=np.float32).reshape(-1, 1, 2)

    # Start reading the video
    while cap.isOpened():
        frame_cnt += 1
        if not paused:  # Only read the next frame if not paused
            ret, frame = cap.read()
            if not ret:
                break

            # Convert to grayscale, as dlib face detector requires grayscale images
            gray_frame = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

            # Use dlib to detect faces
            faces = detector(gray_frame)

            # Iterate over each detected face
            for face in faces:



        # Get 68 facial landmarks
        landmarks = predictor(gray_frame, face)

        # Draw each landmark
        for i in range(68):
            x = landmarks.part(i).x
            y = landmarks.part(i).y
            cv.circle(frame, (x, y), 2, (0, 255, 0), -1)  # Mark all landmarks in green

        # Get the coordinates of the 51st and 8th landmarks
        x_51, y_51 = landmarks.part(51).x, landmarks.part(51).y  # 51st landmark
        x_8, y_8 = landmarks.part(8).x, landmarks.part(8).y  # 8th landmark

        # Calculate the Euclidean distance between the two landmarks
        distance = np.sqrt((x_51 - x_8) ** 2 + (y_51 - y_8) ** 2)

        # Display distance on the video frame
        cv.putText(frame, f"Dist: {distance:.2f} Frame: {frame_cnt}", (10, 30), cv.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 255), 2)
        cv.line(frame, (x_51, y_51), (x_8, y_8), (0, 0, 255), 2)

        # Determine if the tongue tip needs to be manually marked
        if distance > mouth_open_threshold and not tongue_visible:
            # Prompt user if they see the tongue tip
            print("We detect the mouth is open. Do you see the tip of the tongue? (y/n)")
            user_input = input().strip().lower()
            if user_input == 'y':
                # Pause video to manually select the tongue tip
                paused = True
                tongue_point = select_tongue_point(frame)  # Manually select tongue tip
                tongue_visible = True
                prev_tongue_point = tongue_point.copy()
                paused = False
```

```
# If the tongue tip exists and the mouth is closed, prompt the user to clear the tongue tip
if tongue_visible and distance < mouth_close_threshold:
    print("We detect the mouth is closed. Do you see that the mouth is closed? (y/n)")
    user_input = input().strip().lower()
    if user_input == 'y':
        tongue_visible = False
        tongue_point = None
        # Prompt the user to update the threshold
        print("Update the threshold? (y/n)")
        user_input = input().strip().lower()
        if user_input == 'y':
            print("Input the new threshold:")  # Preferably update to 194
            mouth_open_threshold = int(input().strip())

# If the tongue tip exists, track it using optical flow
if tongue_visible and tongue_point is not None:
    # Track the tongue tip using optical flow
    tongue_point_curr, st, err = cv.calcOpticalFlowPyrLK(prev_gray, gray_frame, tongue_point, None, **lk_params)

    # Check if optical flow is valid and the point was successfully tracked (st value 1 indicates successful tracking)
    if tongue_point_curr is not None and st[0][0] == 1:
        tongue_point = tongue_point_curr  # Update tongue tip position
        tongue_x, tongue_y = tongue_point.ravel()

        # Check if tongue tip movement is excessive
        tongue_movement = np.linalg.norm(tongue_point - prev_tongue_point)
        if tongue_movement > tongue_movement_threshold:
            # If tongue movement is excessive, pause video and prompt user
            paused = True
            print("Always select yes in the part? (y/n)")
            user_input = input().strip().lower()
            if user_input == 'y':



                # User confirms an error; reselect tongue tip
                tongue_point = select_tongue_point(frame)
            prev_tongue_point = tongue_point.copy()
            paused = False
        elif tongue_movement <= tongue_movement_min_threshold:
            # If tongue movement is minimal, pause video and prompt user
            paused = True
            print("The movement of the tip is too low. Is it an error? (y/n)")
            user_input = input().strip().lower()
            if user_input == 'y':
                # User confirms an error; reselect tongue tip
                tongue_point = select_tongue_point(frame)
            prev_tongue_point = tongue_point.copy()
            paused = False
        else:
            prev_tongue_point = tongue_point.copy()

    # Write the current frame number and tongue tip coordinates to the file
    log_file.write(f"{frame_cnt} {int(tongue_x)} {int(tongue_y)}\n")

    # Check tracking correctness every 20 frames
    if check_tracking:
        frame_counter += 1
        if frame_counter >= check_frames_interval:
            paused = True
            print("Is the tracking correct? (y/n)")
            user_input = input().strip().lower()
            if user_input == 'n':
                # User confirms an error; reselect tongue tip
                tongue_point = select_tongue_point(frame)
            elif user_input == 'y':
                # User confirms correct tracking; stop checking every 10 frames
                check_tracking = False



                        frame_counter = 0  # Reset frame counter
                        paused = False

                cv.circle(frame, (int(tongue_x), int(tongue_y)), 5, (0, 0, 255), -1)  # Mark tongue tip in red

    # Display each processed frame
    cv.imshow('Face and Tongue Tracking', frame)

    # Detect keyboard input
    key = cv.waitKey(10) & 0xFF
    if key == ord('q'):
        break
    elif key == ord('p'):
        paused = True
        print("Is tracking wrong? (y/n)")
        user_input = input().strip().lower()
        if user_input == 'y':
            tongue_point = select_tongue_point(frame)
            paused = False
    elif key == ord('c'):
        paused = False

    prev_gray = gray_frame.copy()
```

```
import numpy as np

# Function to load 3D points
def load_3d_points(file_path):
    points = {}
    with open(file_path, 'r') as f:
        for line in f:
            frame, x, y, z = line.strip().split()
            points[int(frame)] = np.array([float(x), float(y), float(z)])
    return points

# Load 3D coordinate points
forehead_points = load_3d_points('./Points/3d_forehead_points_LM.txt')
nose_points = load_3d_points('./Points/3d_nose_points_LM.txt')
right_cheek_points = load_3d_points('./Points/3d_right_cheek_points_LM.txt')
left_cheek_points = load_3d_points('./Points/3d_left_cheek_points_LM.txt')

# Get all common frames
common_frames = set(forehead_points.keys()) & set(nose_points.keys()) & set(right_cheek_points.keys()) & set(left_cheek_points.keys())

# Define a dictionary to store the reference coordinate system for each frame
reference_coordinate_systems = {}
reference_coordinate_systems_copy = {}

# Dynamically calculate the coordinate system for each frame
for frame in sorted(common_frames):
    # Extract points for the corresponding frame
    forehead_pt = forehead_points[frame]
    nose_pt = nose_points[frame]  # Used as the origin of the coordinate system
    right_cheek_pt = right_cheek_points[frame]
    left_cheek_pt = left_cheek_points[frame]




    # Y-axis: Vector from the left cheek to the right cheek
    y_axis = right_cheek_pt - left_cheek_pt
    y_axis = y_axis / np.linalg.norm(y_axis)  # Normalize

    # X-axis: Calculated using the normal vector
    # Create a plane with forehead, right cheek, and left cheek points, then calculate the normal vector of this plane as the X-axis
    cheek_plane_normal = np.cross(right_cheek_pt - nose_pt, left_cheek_pt - nose_pt)
    x_axis = cheek_plane_normal / np.linalg.norm(cheek_plane_normal)  # Normalize

    # Z-axis: Cross product of X and Y axes
    z_axis = np.cross(x_axis, y_axis)
    z_axis = z_axis / np.linalg.norm(z_axis)  # Normalize

    # Store the coordinate system for each frame
    reference_coordinate_systems[frame] = {
        'origin': nose_pt,
        'x_axis': x_axis,
        'y_axis': y_axis,
        'z_axis': z_axis
    }

    # Output the reference coordinate system for each frame
    print(f"Frame {frame}:")
    print(f"Origin (Nose Tip): {nose_pt}")
    print(f"X Axis (Normal to cheek plane): {x_axis}")
    print(f"Y Axis (From left cheek to right cheek): {y_axis}")
    print(f"Z Axis (Cross product of X and Y): {z_axis}")
    print("="*50)



# Now you can use `reference_coordinate_systems` to transform other 3D points into the reference coordinate system for each frame.
# Load the 3D coordinates of the tongue tip in the LM system, transform these coordinates into the reference coordinate system,
# calculate the X, Y, Z coordinates for each frame, and store them in a dictionary.
tongue_points = load_3d_points('./tongue_3d_pointsLM.txt')

# Store the X, Y, Z coordinates of the tongue tip in the reference coordinate system for each frame
tongue_transformed_coords = {}
tongue_transformed_coords_copy = {}

# For each frame, transform the tongue tip coordinates into the reference coordinate system
for frame in sorted(common_frames):
    if frame not in tongue_points:
        continue

    tongue_pt = tongue_points[frame]

    # Get the reference coordinate system for the frame
    ref_coord = reference_coordinate_systems[frame]
    origin = ref_coord['origin']
    x_axis = ref_coord['x_axis']
    y_axis = ref_coord['y_axis']
    z_axis = ref_coord['z_axis']

    # Calculate the vector from the tongue tip to the nose tip (origin)
    tongue_vector = tongue_pt - origin

    # Calculate the X, Y, Z coordinates of the tongue tip in the reference coordinate system
    tongue_x = np.dot(tongue_vector, x_axis)  # Projection onto the X-axis
    tongue_y = np.dot(tongue_vector, y_axis)  # Projection onto the Y-axis
    tongue_z = np.dot(tongue_vector, z_axis)  # Projection onto the Z-axis









    # Output the transformed results for each frame
    print(f"Frame {frame}: Tongue Tip in Face-Reference Coordinate System: X: {tongue_x}, Y: {tongue_y}, Z: {tongue_z}")

# Save the results to a file
with open('./3d_tongue_transformed_LM.txt', 'w') as f:
    for frame, (tongue_x, tongue_y, tongue_z) in tongue_transformed_coords.items():
        f.write(f"{frame} {tongue_x} {tongue_y} {tongue_z}\n")

print("LM Tongue 3D coordinates in reference coordinate system saved successfully.")
```

```python
import numpy as np
import matplotlib.pyplot as plt

# Now you can calculate the coordinate differences in X, Y, Z coordinates of the tongue tip for each frame in the LM and RM systems and plot them
tongue_points_LM = load_3d_points('./3d_tongue_transformed_LM.txt')
tongue_points_RM = load_3d_points('./3d_tongue_transformed_RM.txt')

common_frames = set(tongue_points_LM.keys()) & set(tongue_points_RM.keys())

# Store differences
x_diffs = []
y_diffs = []
z_diffs = []

# For each frame, calculate the difference in X, Y, Z coordinates between LM and RM systems
for frame in sorted(common_frames):
    # Get the 3D coordinates of the tongue tip in the LM and RM systems
    tongue_LM = tongue_points_LM[frame]
    tongue_RM = tongue_points_RM[frame]

    # Calculate the differences in X, Y, Z coordinates
    x_diff = tongue_LM[0] - tongue_RM[0]
    y_diff = tongue_LM[1] - tongue_RM[1]
    z_diff = tongue_LM[2] - tongue_RM[2]

    x_diffs.append(x_diff)
    y_diffs.append(y_diff)
    z_diffs.append(z_diff)



# Define IQR method to remove outliers
def remove_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    return [x for x in data if lower_bound <= x <= upper_bound]

# Remove outliers from the differences in X, Y, Z coordinates
x_diffs_filtered = remove_outliers(x_diffs)
y_diffs_filtered = remove_outliers(y_diffs)
z_diffs_filtered = remove_outliers(z_diffs)

# Calculate the variance of the differences in X, Y, Z coordinates after removing outliers
x_variance_filtered = np.std(x_diffs_filtered)
y_variance_filtered = np.std(y_diffs_filtered)
z_variance_filtered = np.std(z_diffs_filtered)

# Print the variance results after removing outliers
print(f"Filtered Variance of X coordinates difference: {x_variance_filtered}")
print(f"Filtered Variance of Y coordinates difference: {y_variance_filtered}")
print(f"Filtered Variance of Z coordinates difference: {z_variance_filtered}")

# Plot the filtered differences in X, Y, Z coordinates over frames
frames = sorted(common_frames)

plt.figure(figsize=(12, 6))

plt.subplot(3, 1, 1)
plt.plot(frames[:len(x_diffs_filtered)], x_diffs_filtered, label='X Difference', color='r')
plt.title('X Coordinate Difference (LM - RM) (Filtered)')
plt.xlabel('Frame')


plt.ylabel('Difference')

plt.subplot(3, 1, 3)
plt.plot(frames[:len(z_diffs_filtered)], z_diffs_filtered, label='Z Difference', color='b')
plt.title('Z Coordinate Difference (LM - RM) (Filtered)')
plt.xlabel('Frame')
plt.ylabel('Difference')

plt.tight_layout()
plt.show()

# Plot the variance after removing outliers
plt.figure(figsize=(10, 5))
plt.bar(['X Std', 'Y Std', 'Z Std'], [x_variance_filtered, y_variance_filtered, z_variance_filtered])
plt.title('Std of X, Y, Z Coordinates Difference')
plt.show()
```

REFERENCES

Hu, P., Cao, W., Li, H., & Lin, Z. (2009). A 3D Face Coordinate System for Expression-Changed Pose Alignment. *2009 Fifth International Conference on Image and Graphics*, 847–852. https://doi.org/10.1109/ICIG.2009.133

Mansi, K. (2024). *Mansikataria/tongue-tip-detection-and-tracking* [Python]. https://github.com/mansikataria/tongue-tip-detection-and-tracking (Original work published 2022)

Mozaffari, M. H., & Lee, W.-S. (2020). *Deep Learning for Automatic Tracking of Tongue Surface in Real-time Ultrasound Videos, Landmarks instead of Contours* (No. arXiv:2003.08808). arXiv. https://doi.org/10.48550/arXiv.2003.08808

Rosebrock, A. (2017, April 3). Facial landmarks with dlib, OpenCV, and Python. *PyImageSearch*. https://pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/

Zhu, J., Styler, W., & Calloway, I. (2019). *A CNN-based tool for automatic tongue contour tracking in ultrasound images* (No. arXiv:1907.10210). arXiv. https://doi.org/10.48550/arXiv.1907.10210