Human Computer Interaction CS449 – CS549
Assignment-5: Development of Gesture-Based Interaction Using Mediapipe

Group Members
Nisa Defne Aksu - 36506
Mohammad Javadian Farzaneh - 36239
Pelin Karadal - 36504
Hasan Barış Aygen - 36386

In this assignment, the MediaPipe library is used to create an interactive system that uses hand gesture recognition. Some hand position landmarks are used to generate the gestures. A live stream from the camera on the computer is used to identify the movements. The gestures cause a triggering event for the buttons on the graphical user interface. Users can perform their chosen actions using hand gestures.

Overview of the System

The chosen hand gestures are click, scroll, and zoom. The scroll gesture's directions are up, down, left, and right. The zoom gesture includes zooming in and out options.

Click Gesture: The click gesture is applied by connecting the tip of the thumb and the tip of the index finger. All fingers are up. The used landmarks are thumb tip (4) and index finger tip (8). When you move the red pointer over an image, you can use the gesture below to click and trigger the select action. When an image is selected, it is highlighted with a green background.

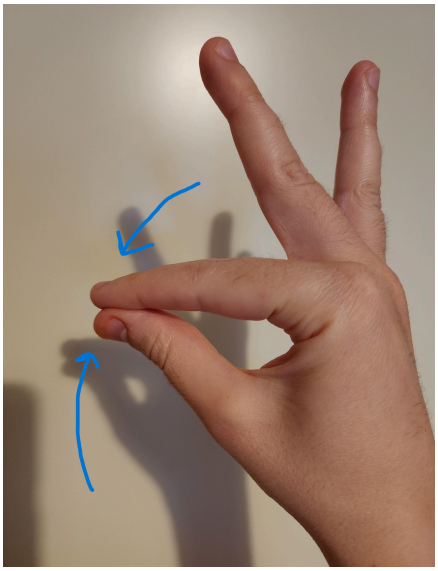

Figure 1. *Hand-indicated click gesture*

Scroll Gesture: The scroll gesture is implemented similarly to the peace sign, but there is no gap between the index and the middle finger. The index and middle finger are up; the pinky and the ring finger are down, and the tip of the thumb is over the pip of the ring finger. The users can scroll by performing the hand gesture and moving their hand straight to the preferred direction (up, down, left, right). The landmarks used are index finger tip (8), middle finger tip (12), ring finger pip (14), and thumb tip (4). There are 2 scrollbars in the app which are placed on the below and right. By using these gestures, the user can interact with those scrollbars and move through the image gallery in the directions of left, right, up, down. This is useful since there might be more images then a single page could take and therefore users should have this ability. When this gesture is used, the scrollbar moves synchronized with the hand so that the user can understand which part of the application he/she interacts with.

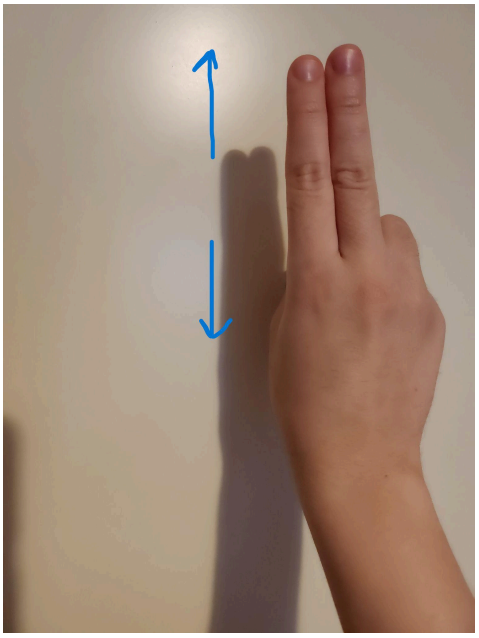

Figure 2. *Hand-indicated scroll up and down gesture*

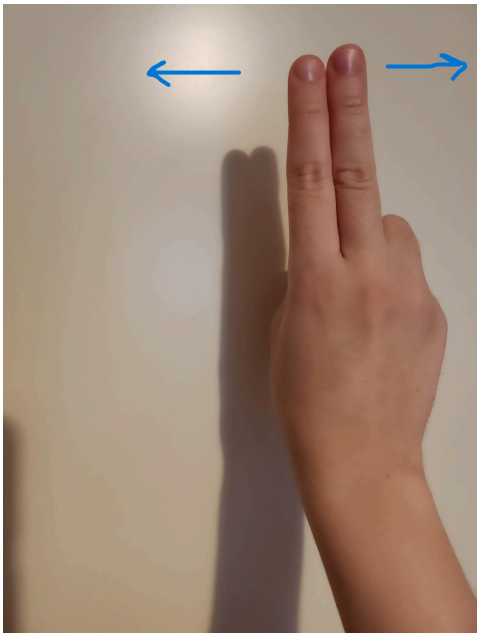

Figure 3. *Hand-indicated scroll left and right gesture*

Zoom Gesture: The zoom gesture is performed by the thumb, index, and middle fingers; the rest of the fingers are down. The index and middle fingers are moved in unison and there is no gap between them. To zoom in, the distance between the thumb and the connected (index-middle) fingers are expanded. To zoom out, the distance between the thumb and the connected (index-middle) fingers is decreased, but there is always space; the thumb and the connected (index-middle) fingers don't touch. The landmarks used are index finger tip (8), middle finger tip (12), thumb tip (4), and ring finger pip (14). The ring finger is used for calculation purposes. This provides a zooming option to the user. Users can make an image bigger or smaller by changing its size.

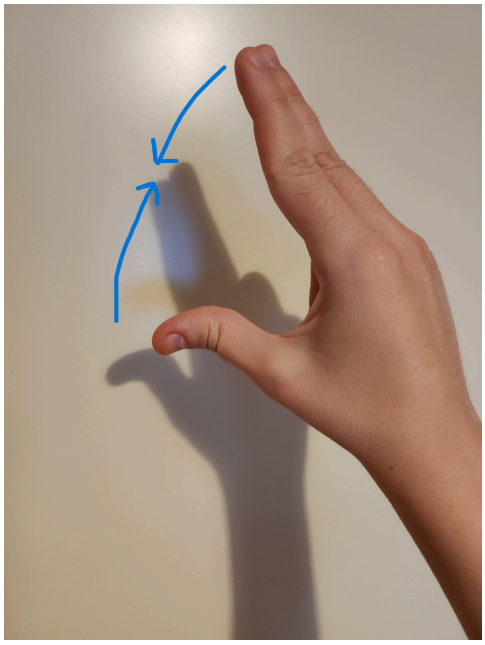

Figure 4. *Hand-indicated zoom out gesture*

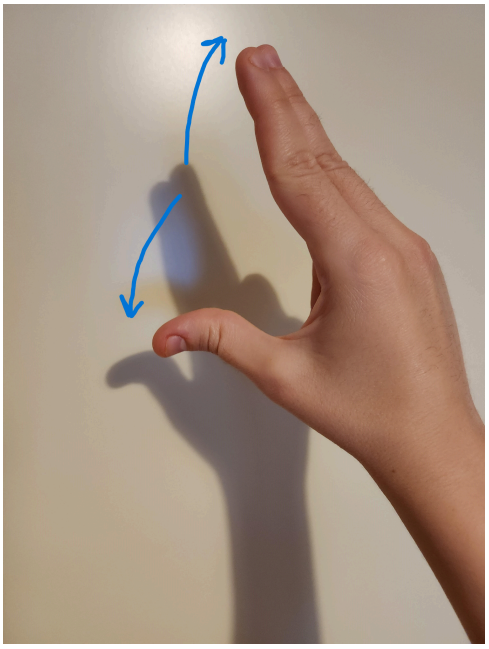

Figure 5. *Hand-indicated zoom in gesture*

Hovering Over Images: There is a red circle that follows the user's finger and its direction, necessary for hovering over images and selecting them. The user chooses an image he/she wants to select and hovers the pointer over it. When the pointer hovers over the candidate images, the images are highlighted with a blue colour. When the user reaches the target image and clicks, the image is selected and highlighted with a green colour. This is mainly used to track user behaviour and better visualise user actions, as well as to achieve certain gesture-triggered events.

Implementation Details

- For writing and implementing gesture functionality:

The main function used to calculate distance between landmarks is calculate_distance() that takes two parameters: landmark1 and landmark2. The euclidean distance formula is used in this function. Additionally, one second cooling time is implemented for each gesture.

Click Gesture: The click gesture's landmarks, which are thumb tip (4) and index tip (8) are put into the calculate_distance function. If the distance is less than 20, then click gesture is detected.

Scroll Gesture: For the scroll gesture, first the horizontal alignment of the index tip (8) and the middle tip (12) is checked. This is done by getting the absolute value of the distances of the y coordinates of landmarks 8 and 12. If the value is less than 20, the index fingertip and the middle fingertip are aligned horizontally. Then, the distance between the thumb tip (4) and the ring finger (14) is calculated using the calculate_distance function. If the distance is less than 20, the

landmarks are aligned. When both of the alignments take place, it means that the user is performing the scroll gesture.

- **Scroll Direction:** In order to detect the scroll direction, the previous location of the index finger is checked and is compared with the current location and decides the directions. The x and y coordinates of the index finger (8) are saved. If no previous position is recorded yet, the current position is set as the starting point and movement is not calculated. Two thresholds are set to determine if the movement is big enough to count. 20 (major threshold) is for the minimum amount the finger must be moved for the code to count it as scrolling. 5 (minor threshold) is for allowing small wobbles or side movements without changing the direction. After, the difference between the current position and the previous position is calculated in delta_x for horizontal movement and delta_y for vertical movement. Vertical movement happens when the finger is moved up or down larger than the threshold (abs(delta_y) > major_threshold) and doesn't move much side-to-side (abs(delta_x) < minor_threshold). If delta_y > 0, it means the finger moved down. Otherwise, it moved up. Horizontal movement happens when the finger is moved left or right larger than the threshold values (abs(delta_x) > major_threshold) and doesn't move much up or down (abs(delta_y) < minor_threshold). If delta_x > 0, it means the finger moved right. Otherwise, it moved left. If none of the above conditions are met, it decides there's no clear direction (direction = "none"). After the direction is decided, the current position is updated as the previous position.

Zoom Gesture: The zoom gesture detection contains two distances. The first one is the distance between index fingertip (8) and middle fingertip (12); again, calculated using calculate_distance function. If the distance is less than 18, it means that the fingers are aligned together. The second detection is the distance between the thumb tip (4) and ring finger (14); it is calculated using the calculate_distance function. The thumb-ring distance should be in the range of 50 to 100 to be differentiated from the scrolling gesture. When both distances are in the previously mentioned ranges, the gesture is considered as zoom.

- **Zoom Direction:** Three landmarks are used in zoom direction detection: index tip (8), thumb tip (4), and middle fingertip (12). The distances between thumb and index and thumb and middle finger are detected separately using calculate_distance function. Then, the average of both distances are taken for the average distance of zooming. If the average distance increases (average_distance > previous_distance + 10), zoom out is detected. If the average distance decreases (average_distance < previous_distance - 10), zoom in is detected.

- For writing and implementing main code files that include gestures and GUI:

gestures.py: “gestures.py” is written to detect various hand gestures using a set of landmarks representing the positions of key points on the hand. The gestures are designed for interaction with a graphical user interface (GUI), such as zooming, scrolling, clicking, and hovering over UI elements. It has 7 different functions:

1. is_zoom_detected(): The is_zoom_detected function identifies a zoom gesture by analyzing the distances between specific hand landmarks: thumb tip, ring finger PIP, index tip, and middle tip.

2. is_hover_gesture(): The is_hover_gesture function checks if the index fingertip hovers over any GUI element by comparing its position with the bounding boxes of those elements. If the fingertip’s position is inside the boundaries of a box, the function returns the index of the hovered element. Or else it returns -1.

3. detect_zoom_direction(): The detect_zoom_direction function determines whether a zoom gesture represents zooming in or out by calculating the average distance between the thumb tip, index tip, and middle tip. By comparing the current average distance with the previous_distance, it detects whether the fingers are moving closer (zoom in) or far away (zoom out).

4. is_scroll_gesture(): The is_scroll_gesture function detects scrolling gestures by checking the placement of specific fingers. It is scrolling when the index and middle fingertips are parallel and the thumb tip is near the ring finger PIP joint.

5. detect_scroll_direction(): The detect_scroll_direction function understands the direction of scrolling (up, down, left, or right) based on the movement of the index fingertip. It compares the fingertip’s position with the previous frame and uses thresholds to classify significant movement patterns.

6. is_click_gesture(): The is_click_gesture function detects a click gesture when the thumb tip and index fingertip are close together, with a distance of less than 20 units. It returns True for a detected click.

7. detect_hover_gesture(): The detect_hover_gesture function checks if the user is hovering over a GUI element by checking the index fingertip's position. If it lies within the bounding boxes of GUI elements, it returns the index of the hovered element.

mediaPipeHandler.py: “mediaPipeHandler.py” integrates gesture detection with a GUI application using MediaPipe for hand tracking and Tkinter for the GUI. It integrates hand gesture detection with a GUI application using MediaPipe, OpenCV, and Tkinter. The GestureDetection class processes video input to detect gestures such as hover, click, scroll, and zoom, and maps

these to GUI interactions. It uses MediaPipe for hand landmark tracking, OpenCV for camera processing and visual feedback, and Tkinter for GUI rendering. The code uses a cursor system where fingertip movements are mapped to GUI dimensions, let users interact naturally. Visual feedback, including annotations and a ripple effect for the fingertip trail, enhances interactivity. The integration between gesture detection and GUI is modular, with actions like gesture_hover, gesture_click, gesture_scroll, and gesture_zoom defined in the GUI logic.

utils.py: Two functions that deal with points in 2D space are included in this file. The calculate_distance function uses a formula based on the Pythagorean theorem to determine the straight-line distance between two places. With the middle point serving as the angle's vertex, the calculate_angle function determines the angle produced by three points. It accomplishes this by treating the points as vectors, figuring out their directions, and determining their alignment using the dot product. After that, the angle is changed from radians to degrees. To prevent errors, the angle is adjusted to 0 if any vector length is zero (such as overlapping points). Geometry, graphics, and point relationship analysis can all benefit from this.

image_gallery_app.py: Using Tkinter, this Python program develops a GUI application for controlling an image gallery that incorporates gesture-based controls with MediaPipe. A scrollable gallery area that shows thumbnails of photographs, a buttons panel for other functions like loading images or seeing a live camera feed, and a guide panel for user instructions and gesture controls make up the app's three primary components. Users can utilize motions or mouse movements to navigate, scroll, select, or zoom in/out of photographs after loading them into the gallery and viewing them in a centered viewer. The application uses the PIL package to manage image resizing and transformations while preserving quality, and it makes use of threading to provide a fluid camera feed. A user-friendly interface for browsing and adjusting photos is provided by the design's strong features, which include gesture-based hover and click detection, thumbnail previews, and support for dynamic updates. An overview of each function in the ImageGalleryApp class is provided below:

1. Setup and Initialization

__init__(): It creates the main application window, separates it into panels (like left, right and gallery panels), and sets up important elements like the cursor and camera feed.

2. Functions of the Gallery

update_selected_image(): Displays the chosen image with the proper scaling and centering in a fixed-size viewer.
_update_image_in_fixed_window(): Makes sure the image that is now displayed precisely fits the viewer's specified dimensions.
load_images(): Selects several photos, generates thumbnails, and shows them in the gallery section by opening a file dialog.
open_image(): Opens the chosen image in a new window so you may view it in detail.

3. Functions of Gesture Control

gesture_click(): Highlights the chosen thumbnail in the gallery and simulates a click gesture to choose an image.
gesture_scroll(): Allows for both vertical and horizontal gallery navigation by simulating scrolling motions.
gesture_hover(): When the cursor is over a thumbnail, it highlights it without selecting it.
gesture_zoom(): Manages zooming in and out of the displayed image using zoom motions.

4. Zooming Functions

zoom_in(): Enlarges the image that is shown by scaling it up.
zoom_out(): Reduces the size of the image that is displayed.

5. Cursor and Gesture Functions

detect_hover(): Returns the index of the hovered picture after determining whether the pointer is over a certain thumbnail.
get_thumbnail_positions(): Retrieves each gallery thumbnail's on-screen location so that interaction can be detected.
update_cursor(): Dynamically modifies the cursor's location in response to mouse or gesture input.

6. Panel of Guidance

load_gesture_images(): Enables the user to choose a folder containing gesture images, shows them in the guide panel as thumbnails, and resets the panel if necessary.

7. App and Camera Functions

setup_camera_feed(): Uses the MediaPipe library to launch a separate thread for gesture recognition and live camera feed.
on_closing(): Manages cleanup, including halting the camera feed thread, when the application is closed.

8. Scrolling Functions

_on_mouse_wheel_vertical(): Allows the gallery to be scrolled vertically with the mouse wheel.
_on_mouse_wheel_horizontal(): Shift + mouse wheel can be used to scroll horizontally.

In summary, to ensure modularity and maintainability in this project, we implemented the GUI and gestures as separate but parallel phases in 2 groups (Defne-Barış and Pelin-Mohammad). Using MediaPipe, the gesture functionality was created in a separate file to recognise and decipher hand gestures such as zooming, clicking and scrolling. These movements were converted into actions, such as interacting with photos or exploring the gallery. Using Tkinter,

the GUI was created in a separate file with different areas for a user manual, camera feed and image gallery. After their separate creation, the two parts were smoothly combined by all of us, integrating gesture-detected events to GUI operations. This enabled dynamic interactions, such as using gestures to zoom, select, scroll and hover between photos. This approach optimised development.

Code

The program files include four main files. The main part and entrance of the program is the "image_gallery_app.py" file, which is shown below. It contains "ImageGalleryApp" class, where all of the UI elements and functions that are related to interaction of GUI and gestures are written here.

```
import tkinter as tk
from tkinter import ttk, filedialog
from PIL import Image, ImageTk
import os
import cv2
import threading
import time
import mediaPipeHandler as mph

class ImageGalleryApp:

    def __init__(self, root):
        # Initialize the main application window
        self.root = root
        self.root.title("Image Gallery App")
        self.root.geometry("1000x800")  # Set the window size

        # Configure layout of the main window with resizable grids
        self.root.rowconfigure(0, weight=1)  # Row 0 is expandable
        self.root.columnconfigure(1, weight=1)  # Column 1 is
expandable

        # User Guide and Gesture Panel (Left side)
        self.guide_panel = tk.Frame(self.root, width=500, bg="gray")
        self.guide_panel.grid(row=0, column=0, sticky="ns")  # The
left panel will stretch vertically
```

```
        # Label for user instructions in the guide panel
        self.guide_label = tk.Label(
            self.guide_panel,
            text="User Guide:\n\n1. Scroll to browse images.\n2.
Click an image to view.\n3. Use Zoom buttons.",
            bg="gray",
            fg="white",
            font=("Arial", 10),
            justify="left",
        )
        self.guide_label.pack(pady=10, padx=10)

        # Button to load gesture images
        self.load_gestures_button = tk.Button(self.guide_panel,
text="Load Gestures", command=self.load_gesture_images, width=20)
        self.load_gestures_button.pack(pady=10)

        # Load gesture images by default when the app starts
        self.load_gesture_images()

        # Main Image Gallery Area (Middle section)
        self.gallery_frame = tk.Frame(self.root, bg="white")
        self.gallery_frame.grid(row=0, column=1, sticky="nsew")  #
Center the gallery frame

        # Scrollable canvas for displaying thumbnails
        self.canvas = tk.Canvas(self.gallery_frame, bg="white",
highlightthickness=0)
        self.scroll_y = ttk.Scrollbar(self.gallery_frame,
orient="vertical", command=self.canvas.yview)
        self.scroll_x = ttk.Scrollbar(self.gallery_frame,
orient="horizontal", command=self.canvas.xview)
        self.scroll_y.pack(side="right", fill="y")
        self.scroll_x.pack(side="bottom", fill="x")
        self.canvas.pack(fill="both", expand=True)
        self.canvas.configure(yscrollcommand=self.scroll_y.set,
xscrollcommand=self.scroll_x.set)
```

```
        # Content frame inside the canvas to hold images
        self.gallery_content = tk.Frame(self.canvas, bg="white")
        self.canvas.create_window((0, 0),
window=self.gallery_content, anchor="nw")
        self.gallery_content.bind("<Configure>", lambda e:
self.canvas.configure(scrollregion=self.canvas.bbox("all")))

        # Mouse wheel scrolling
        self.canvas.bind("<MouseWheel>",
self._on_mouse_wheel_vertical)
        self.canvas.bind("<Shift-MouseWheel>",
self._on_mouse_wheel_horizontal)

        # Buttons Panel (Right side)
        self.buttons_panel = tk.Frame(self.root, width=400,
bg="lightgray")
        self.buttons_panel.grid(row=0, column=2, sticky="ns")  # The
right panel will stretch vertically

        # Label to show the camera feed
        self.camera_label = tk.Label(self.buttons_panel,
text="Loading Camera...", bg="lightgray")
        self.camera_label.pack(pady=10)

        self.camera_label = tk.Label(self.buttons_panel,
bg="lightgray", width=400, height=400)
        self.camera_label.pack(pady=10)

        # Image Viewer Area (Bottom section)
        self.viewer_label = tk.Label(
            self.root,
            text="Selected Image will appear here",
            bg="white"
        )
        self.viewer_label.grid(row=1, column=1, sticky="nsew")

        # Store the currently displayed image
        self.current_image = None
```

```
        self.transformed_image = None
        self.original_image = None  # To keep a reference of the
original image
        self.selected_index = None  # Track the index of the
selected image

        self.cursor = tk.Label(self.root, text="O", bg="red",
fg="white")
        self.cursor.place(x=0, y=0)  # Initialize at (0, 0)

        # Button to load images from files
        self.load_button = tk.Button(self.buttons_panel, text="Load
Images", command=self.load_images, width=20)
        self.load_button.pack(pady=10, side="bottom")


        # List to store loaded image file paths
        self.setup_camera_feed()

    def update_selected_image(self, image_path):
        try:
            self.original_image = Image.open(image_path)

            # Fixed dimensions for the viewer
            target_width, target_height = 400, 300
            original_width, original_height =
self.original_image.size

            # Calculate scale to fit the image within the target
dimensions
            scale = min(target_width / original_width, target_height
/ original_height)
            new_width = int(original_width * scale)
            new_height = int(original_height * scale)

            # Resize the image while maintaining aspect ratio
```

```
            self.transformed_image =
self.original_image.resize((new_width, new_height),
Image.Resampling.LANCZOS)

            # Center the resized image in the fixed dimensions
            centered_image = Image.new("RGB", (target_width,
target_height), "white")
            x_offset = (target_width - new_width) // 2
            y_offset = (target_height - new_height) // 2
            centered_image.paste(self.transformed_image, (x_offset,
y_offset))

            # Update the viewer with the centered image
            self.current_image = ImageTk.PhotoImage(centered_image)
            self.viewer_label.config(image=self.current_image,
text="")
        except Exception as e:
         print(f"Error loading image: {e}")


    def _update_image_in_fixed_window(self):
        """Ensure the image is displayed within the fixed viewer
window."""
        try:
            # Fixed dimensions for the viewer
            target_width, target_height = 400, 300  # Example fixed
dimensions

            # Create a blank canvas to center the image
            centered_image = Image.new("RGB", (target_width,
target_height), "white")

            # Calculate offsets to center the image
            image_width, image_height = self.transformed_image.size
            x_offset = max(0, (target_width - image_width) // 2)
            y_offset = max(0, (target_height - image_height) // 2)

            # Paste the resized image onto the blank canvas
```

```
            centered_image.paste(self.transformed_image, (x_offset,
y_offset))

            # Update the viewer label with the centered image
            self.current_image = ImageTk.PhotoImage(centered_image)
            self.viewer_label.config(image=self.current_image)
        except Exception as e:
            print(f"Error updating image: {e}")


    def update_cursor(self, x, y):
        self.cursor.place(x=x, y=y)

    def detect_hover(self, cursor_x, cursor_y):
        positions = self.get_thumbnail_positions()
        for index, (x1, y1, x2, y2) in enumerate(positions):
            if x1 <= cursor_x <= x2 and y1 <= cursor_y <= y2:
                return index
        return None

    def load_gesture_images(self):
        # Clear all child widgets from self.guide_panel, effectively
reset its contents.
        for widget in self.guide_panel.winfo_children():
            widget.destroy()

        # Open a folder selection dialog for gesture images
        gestures_folder = filedialog.askdirectory(title="Select
Gesture Folder")
        if not gestures_folder:  # If the user cancels the folder
selection
            return

        # Get all gesture image files from the selected folder
        gesture_files = os.listdir(gestures_folder)
        gesture_images = [f for f in gesture_files if
f.lower().endswith(('.png', '.jpg', '.jpeg'))]
```

```
        # Notify user if no gesture images are found
        if not gesture_images:
            tk.Label(
                self.guide_panel,
                text="No gesture images found.\nAdd images to the
selected folder.",
                bg="gray",
                fg="yellow",
                font=("Arial", 10),
            ).pack(pady=10)
            return

        # Label to indicate gesture section
        tk.Label(
            self.guide_panel,
            text="Loaded Gestures:",
            bg="gray",
            fg="white",
            font=("Arial", 12, "bold")
        ).pack(pady=10)

        # Frame to hold gesture images
        gestures_frame = tk.Frame(self.guide_panel, bg="gray")
        gestures_frame.pack(fill="both", expand=True)

        # Number of columns for the table
        num_columns = 3
        current_row = tk.Frame(gestures_frame, bg="gray")
        current_row.pack(fill="x")

        # Display each gesture image in a table-like structure
        col_count = 0
        for gesture_image in gesture_images:
            img_path = os.path.join(gestures_folder, gesture_image)

            try:
                # Load and display thumbnail image
                img = Image.open(img_path)
```

```
                img.thumbnail((100, 100))
                img = ImageTk.PhotoImage(img)

                # Create a frame for each image and label
                frame = tk.Frame(current_row, bg="gray", padx=5,
pady=5)
                frame.pack(side="left", padx=10, pady=10)

                img_label = tk.Label(frame, image=img, bg="gray")
                img_label.image = img  # Keep a reference to avoid
garbage collection
                img_label.pack()

                # Display the name of the gesture image
                name = os.path.splitext(gesture_image)[0]
                name_label = tk.Label(frame, text=name, bg="gray",
fg="white")
                name_label.pack()

                # Update column count
                col_count += 1
                if col_count >= num_columns:
                    col_count = 0
                    current_row = tk.Frame(gestures_frame, bg="gray")
                    current_row.pack(fill="x")

            except Exception as e:
                print(f"Error loading {img_path}: {e}")

    def setup_camera_feed(self):
        self.stop_event = threading.Event()
        self.gesture_detection =
mph.GestureDetection(self.stop_event, self.camera_label,
self.root,self)
        self.gesture_detection.start()

    def on_closing(self, stop_event):
```

```
        # Stop the camera feed thread and close the application
window
        stop_event.set()
        self.root.destroy()

    def gesture_click(self, index):
        print("Clicking Event")
        """Handle the clicking gesture and update the selection
highlight."""
        thumbnails = self.gallery_content.winfo_children()

        # Reset the background color of the previously selected
image
        if self.selected_index is not None:
            thumbnails[self.selected_index].configure(bg="white")

        # Highlight the newly selected image
        if index is not None:
            frame = thumbnails[index]
            frame.configure(bg="green")  # Use a distinct color for
selection highlight
            self.selected_index = index  # Update the selected index

            # Open the selected image
            file_path = self.image_files[index]
            self.update_selected_image(file_path)

    def gesture_scroll(self, direction):
        """Simulate scrolling in the specified direction."""
        if direction == "up":
            self.canvas.yview_scroll(-1, "units")
        elif direction == "down":
            self.canvas.yview_scroll(1, "units")
        elif direction == "left":
            self.canvas.xview_scroll(-1, "units")
        elif direction == "right":
            self.canvas.xview_scroll(1, "units")
```

```
    def get_thumbnail_positions(self):
        positions = []
        for widget in self.gallery_content.winfo_children():
            x1 = widget.winfo_rootx() - self.root.winfo_rootx()
            y1 = widget.winfo_rooty() - self.root.winfo_rooty()
            x2 = x1 + widget.winfo_width()
            y2 = y1 + widget.winfo_height()
            positions.append((x1, y1, x2, y2))
        return positions

    def gesture_hover(self, index):
        """Simulate hovering over an image thumbnail."""
        thumbnails = self.gallery_content.winfo_children()

        for i, widget in enumerate(thumbnails):
            # Reset to white unless the image is selected
            if i != self.selected_index:
                widget.configure(bg="white")

        # Highlight hover if it's not the selected image
        if index is not None and index != self.selected_index:
            thumbnails[index].configure(bg="lightblue")  # Hover
color

    def _on_mouse_wheel_vertical(self, event):
        # Scroll vertically using the mouse wheel.
        self.canvas.yview_scroll(-1 * int(event.delta / 120),
"units")

    def _on_mouse_wheel_horizontal(self, event):
        # Scroll horizontally using Shift + mouse wheel.
        self.canvas.xview_scroll(-1 * int(event.delta / 120),
"units")

    def load_images(self):
        # Use a file dialog to select multiple image files
        files = filedialog.askopenfilenames(filetypes=[("Image
Files", "*.png;*.jpg;*.jpeg")])
```

```
        if not files:
            return

        # Clear existing thumbnails in the gallery
        for widget in self.gallery_content.winfo_children():
            widget.destroy()

        self.image_files = list(files)

        # Display the selected images as thumbnails
        for index, file in enumerate(self.image_files):
            img = Image.open(file)
            img.thumbnail((100, 100))  # Create a thumbnail of each
image
            img = ImageTk.PhotoImage(img)

            frame = tk.Frame(self.gallery_content, bg="white",
padx=5, pady=5)
            frame.grid(row=index // 6, column=index % 6, padx=10,
pady=10)

            label = tk.Label(frame, image=img)
            label.image = img  # Keep a reference
            label.pack()
            label.bind("<Button-1>", lambda e, path=file:
self.open_image(path))

            # Display the image file name under the thumbnail
            name = file.split("/")[-1]  # Get the file name
            name_label = tk.Label(frame, text=name if name else
str(index + 1), bg="white")
            name_label.pack()

    def open_image(self, image_path):
        print("New Image Opened")
        # Open an image in a new window for viewing
        viewer = tk.Toplevel(self.root)
        viewer.title("Image Viewer")
```

```
        viewer.geometry("600x400")

        # Open and transform the image
        original_image = Image.open(image_path)
        transformed_image = original_image.copy()
        img_display = ImageTk.PhotoImage(transformed_image)
        label = tk.Label(viewer, image=img_display)
        label.image = img_display  # Keep a reference to avoid
garbage collection
        label.pack(fill="both", expand=True)

        # Control buttons for zooming
        btn_frame = tk.Frame(viewer)
        btn_frame.pack(side="bottom", fill="x", padx=10, pady=5)

    def zoom_in(self):
        """Zoom in on the displayed image."""
        if self.original_image:
            # Increase the scale of the image
            width, height = self.transformed_image.size
            scale = 1.2
            new_width, new_height = int(width * scale), int(height *
scale)

            # Resize the image
            self.transformed_image =
self.original_image.resize((new_width, new_height),
Image.Resampling.LANCZOS)

            # Fit the scaled image within the fixed viewer
dimensions
            self._update_image_in_fixed_window()

    def zoom_out(self):
        """Zoom out on the displayed image."""
        if self.original_image:
            # Decrease the scale of the image
            width, height = self.transformed_image.size
```

```
            scale = 0.8
            new_width, new_height = int(width * scale), int(height *
scale)

            # Resize the image
            self.transformed_image =
self.original_image.resize((new_width, new_height),
Image.Resampling.LANCZOS)

            # Fit the scaled image within the fixed viewer
dimensions
            self._update_image_in_fixed_window()


    def gesture_zoom(self, zoom_direction):
        print("Zoom Direction" + zoom_direction)
        """Handle zoom gestures."""
        if zoom_direction == "in":
            self.zoom_in()
        elif zoom_direction == "out":
            self.zoom_out()

if __name__ == "__main__":
    root = tk.Tk()
    app = ImageGalleryApp(root)
    root.protocol("WM_DELETE_WINDOW", lambda:
app.on_closing(app.stop_event))  # Ensure camera is released when
closing the app
    root.mainloop()
```

Another developed class name is “GestureDetection” in “mediaPipeHandler.py”, which is used to initialize the Mediapipe Library, detecting landmarks, showing the trails, showing the cursor, and triggering the events when a gesture is detected.

```
import cv2
import gestures
import time
import mediaPipeHandler as mph
```

```
import tkinter as tk
from tkinter import ttk, filedialog
from PIL import Image, ImageTk

mpHands = mp.solutions.hands
hands = mpHands.Hands(max_num_hands=1, min_detection_confidence=0.7)
mpDraw = mp.solutions.drawing_utils

default_landmark_spec = mpDraw.DrawingSpec(color=(0, 0, 255),
thickness=2, circle_radius=2)  # Red color
default_connection_spec = mpDraw.DrawingSpec(color=(255, 255, 255),
thickness=2)  # White color

# Printing Commands on the screen
def print_command(frame,command_text):
    cv2.putText(frame, command_text, (50,50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)

class GestureDetection:
    def __init__(self, stop_event, camera_label, root,app):
        self.stop_event = stop_event
        self.camera_label = camera_label
        self.root = root
        self.camera = cv2.VideoCapture(0)
        self.trail = []
        self.trail_max_length = 10
        self.trail_color = (0, 255, 0)
        self.trail_start_radius = 10
        self.previous_index_x = None
        self.previous_index_y = None
        self.previous_distance = None
        self.last_gesture_time = time.time()
        self.app = app


    def move_cursor(self, index_tip):
        gui_width = self.root.winfo_width()
        gui_height = self.root.winfo_height()
```

```
        cursor_x = int(index_tip[0] / 400 * gui_width)
        cursor_y = int(index_tip[1] / 400 * gui_height)

        # Update the cursor position in the GUI
        self.app.update_cursor(cursor_x, cursor_y)

        # Check for hover over thumbnails
        hovered_index = self.app.detect_hover(cursor_x, cursor_y)
        if hovered_index is not None:
            self.app.gesture_hover(hovered_index)


    def update_frame(self):
        if self.stop_event.is_set():
            self.camera.release()
            return

        isCapturedFrameSuccessful, frame = self.camera.read()
        if not isCapturedFrameSuccessful:
            print("An error happened.")
            self.camera.release()
            return

        frame = cv2.resize(frame, (400, 400))
        frame = cv2.flip(frame, 1)
        framergb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Process hand landmarks
        hand_process_result = mph.hands.process(framergb)

        if hand_process_result.multi_hand_landmarks:
            for hand_landmarks in
hand_process_result.multi_hand_landmarks:
                landmarks = []
                for idx, lm in enumerate(hand_landmarks.landmark):  #
Iterate through each landmark
                    x = int(lm.x * frame.shape[1])  # Get x
coordinate
```

```
                    y = int(lm.y * frame.shape[0])  # Get y
coordinate
                    landmarks.append([x, y])

                # Detect hover gesture
                hovered_index = gestures.is_hover_gesture(landmarks,
self.app.get_thumbnail_positions())
                if hovered_index != -1:  # Ensure hovered_index is
valid
                    self.app.gesture_hover(hovered_index)

                # Draw connections first
                mph.mpDraw.draw_landmarks(frame, hand_landmarks,
mph.mpHands.HAND_CONNECTIONS, mph.default_landmark_spec,
mph.default_connection_spec)

                # Draw the filled green circle for the index finger
tip
                index_tip = landmarks[8]
                cv2.circle(frame, (index_tip[0], index_tip[1]), 5,
(0, 255, 0), -1)
                self.move_cursor(index_tip)

                # Add the index finger position to the trail
                self.trail.append(index_tip)
                if len(self.trail) > self.trail_max_length:
                    self.trail.pop(0)

                # Draw the ripple trail
                for i, point in enumerate(self.trail):
                    radius = self.trail_start_radius - int((i /
self.trail_max_length) * self.trail_start_radius)
                    cv2.circle(frame, point, radius,
self.trail_color, 1)

                # Gesture detection with visual feedback
                if gestures.is_click_gesture(landmarks) and
time.time() - self.last_gesture_time > 1:
```

```
                    cv2.putText(frame, "Click", (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA)
                    gui_width = self.root.winfo_width()
                    gui_height = self.root.winfo_height()
                    cursor_x = int(index_tip[0] / 400 * gui_width)
                    cursor_y = int(index_tip[1] / 400 * gui_height)
                    hovered_index = self.app.detect_hover(cursor_x,
cursor_y)
                    self.app.gesture_click(hovered_index)
                    self.last_gesture_time = time.time()

                elif gestures.is_scroll_gesture(landmarks) and
time.time() - self.last_gesture_time > 1:
                    direction, self.previous_index_x,
self.previous_index_y = gestures.detect_scroll_direction(
                        landmarks, self.previous_index_x,
self.previous_index_y
                    )
                    if direction != "none":
                        cv2.putText(frame, f"Scrolling: {direction}",
(50, 100), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE_AA)
                        self.last_gesture_time = time.time()
                        self.app.gesture_scroll(direction)

                elif gestures.is_zoom_detected(landmarks) and
time.time() - self.last_gesture_time > 1:
                    zoom_direction, self.previous_distance =
gestures.detect_zoom_direction(landmarks, self.previous_distance)
                    if zoom_direction != "":
                        cv2.putText(frame, f"Zooming:
{zoom_direction}", (50, 150), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0,
255), 2, cv2.LINE_AA)
                        self.app.gesture_zoom(zoom_direction)
                        self.last_gesture_time = time.time()

        # Convert frame to ImageTk format
        img = Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))
        img_tk = ImageTk.PhotoImage(image=img)
```

```
        # Update the label
        self.camera_label.img_tk = img_tk  # Keep a reference
        self.camera_label.configure(image=img_tk)

        # Schedule the next frame update
        self.root.after(10, self.update_frame)

    def start(self):
        self.update_frame()
```

In the “gestures.py”, our methods for detecting gestures are written.

```
import utils

def is_zoom_detected(landmarks):
    # Draw the filled green circle for the index finger tip (after
drawing connections)
    index_tip = landmarks[8]  # Index finger tip
    middle_tip =landmarks[12] # middle finger tip
    thumb_tip = landmarks[4]
    ring_pip = landmarks[14]

    index_middle_distance =
utils.calculate_distance(middle_tip,index_tip)
    thumb_ring_distance =
utils.calculate_distance(thumb_tip,ring_pip)


    if (50 < thumb_ring_distance < 100) and (index_middle_distance <
18):
        return True
    return False


def is_hover_gesture(landmarks, element_positions):
```

```
    """
    Detect if the index fingertip is hovering over any UI element.
    :param landmarks: List of hand landmark positions.
    :param element_positions: List of bounding box positions [(x1,
y1, x2, y2), ...].
    :return: Index of the hovered element or -1 if no hover is
detected.
    """
    index_tip = landmarks[8]  # Index fingertip position (x, y)

    for idx, (x1, y1, x2, y2) in enumerate(element_positions):
        if x1 <= index_tip[0] <= x2 and y1 <= index_tip[1] <= y2:
            return idx  # Return the index of the hovered element

    return -1  # No hover detected

def detect_zoom_direction(landmarks,previous_distance):
    index_tip = landmarks[8]  # Index finger tip
    thumb_tip = landmarks[4]  # Thumb tip
    middle_tip =landmarks[12]

    result=""

    # Calculate the distances
    thumb_index_distance = utils.calculate_distance(thumb_tip,
index_tip)
    thumb_middle_distance = utils.calculate_distance(thumb_tip,
middle_tip)

    # Calculate the average distance for zooming
    average_distance = (thumb_index_distance + thumb_middle_distance)
/ 2
    # Determine zoom gesture
    if previous_distance is not None:
        if average_distance < previous_distance - 10:  # Zoom in
            result = "out"
        elif average_distance > previous_distance + 10:  # Zoom out
            result = "in"
```

```
    previous_distance = average_distance
    return result,previous_distance


def is_scroll_gesture(landmarks):
    """Detects scrolling when index (8) and middle finger (12) are
aligned."""
    index_tip = landmarks[8]  # Tip of the index finger
    middle_tip = landmarks[12]  # Tip of the middle finger
    ring_pip = landmarks[14]
    thumb_tip = landmarks[4]

    is_aligned_horizontally = abs(index_tip[0] - middle_tip[0]) < 20
# y-coordinates aligned
    is_ring_with_thumb = utils.calculate_distance(ring_pip,thumb_tip)
< 50

    # Return True if aligned either vertically or horizontally
    # return is_aligned_vertically or is_aligned_horizontally
    return is_ring_with_thumb and is_aligned_horizontally


# Detecting the scroll direction
def detect_scroll_direction(landmarks, previous_index_x,
previous_index_y):
    """Detects if the user is scrolling and determines the direction,
with deviation tolerance."""
    index_tip_x = landmarks[8][0]  # x-coordinate of the index
finger tip
    index_tip_y = landmarks[8][1]  # y-coordinate of the index
finger tip

    # If this is the first frame, initialize the previous position
    if previous_index_x is None or previous_index_y is None:
        return "none", index_tip_x, index_tip_y

    # Define thresholds
```

```
    major_threshold = 20  # Major movement threshold (to determine
primary direction)
    minor_threshold = 5   # Minor deviation threshold (to ignore
small side movements)

    # Calculate differences
    delta_x = index_tip_x - previous_index_x
    delta_y = index_tip_y - previous_index_y

    # Determine scrolling direction with tolerance
    if abs(delta_y) > major_threshold and abs(delta_x) <
minor_threshold:
        # If vertical movement is significant and horizontal
movement is within minor deviation
        if delta_y > 0:
            direction = "down"
        else:
            direction = "up"
    elif abs(delta_x) > major_threshold and abs(delta_y) <
minor_threshold:
        # If horizontal movement is significant and vertical
movement is within minor deviation
        if delta_x > 0:
            direction = "right"
        else:
            direction = "left"
    else:
        # No significant movement or mixed movement
        direction = "none"

    # Update previous positions
    return direction, index_tip_x, index_tip_y

def is_click_gesture(landmarks):
    """Detects the CLICK gesture."""
    thumb_tip = landmarks[4]
    index_tip = landmarks[8]
    distance = utils.calculate_distance(thumb_tip, index_tip)
```

```
    return distance < 20  # Adjust threshold as needed
```

Also, we have defined some extra functions in “utils.py” for distance and angle detection used in the gesture detection part.

```
import math
import numpy as np

# Calculate distance between two points
def calculate_distance(point1, point2):
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] -
point2[1])**2)


def calculate_angle(pointA, pointB, pointC):
    # Vector AB
    AB = np.array([pointB[0] - pointA[0], pointB[1] - pointA[1]])
    # Vector BC
    BC = np.array([pointC[0] - pointB[0], pointC[1] - pointB[1]])

    # Dot product and magnitudes
    dot_product = np.dot(AB, BC)
    magnitude_AB = np.linalg.norm(AB)
    magnitude_BC = np.linalg.norm(BC)

    # Calculate the angle in radians and then convert to degrees
    if magnitude_AB * magnitude_BC == 0:
        return 0
    angle = math.acos(dot_product / (magnitude_AB * magnitude_BC))
    angle_degrees = math.degrees(angle)

    return angle_degrees
```

Github

Our GitHub repository link: https://github.com/BarisAygen/HCI_Assignment_5

Pelin and Mohammad worked on: (You can check mediapipe_gesture_implementation.ipynb, gestures.py, utils.py)

- Gesture creation and recognition using MediaPipe library for clicking, zooming and scrolling (Together)
- Adding trail features to the pointer (Together)
- Modularizing the gesture-related codes (Together)

Defne and Barış worked on: (You can check image_gallery_app_only_gui.py)

- Coding GUI and its functionalities (Defne)
- Adding camera and its functionality to GUI (Barış)
- Fixing GUI related issues and problems (Together)

The whole team worked on: (You can check gestures.py, utils.py, mediaPipeHandler.py, image_gallery_app.py)

- Connecting the backend and GUI part

Video Demo

Short video demonstrating our system in action: **https://youtu.be/dk_h8NhgZAg**