SESSION 3 - Project HandShield (Dr. Strange Filter)

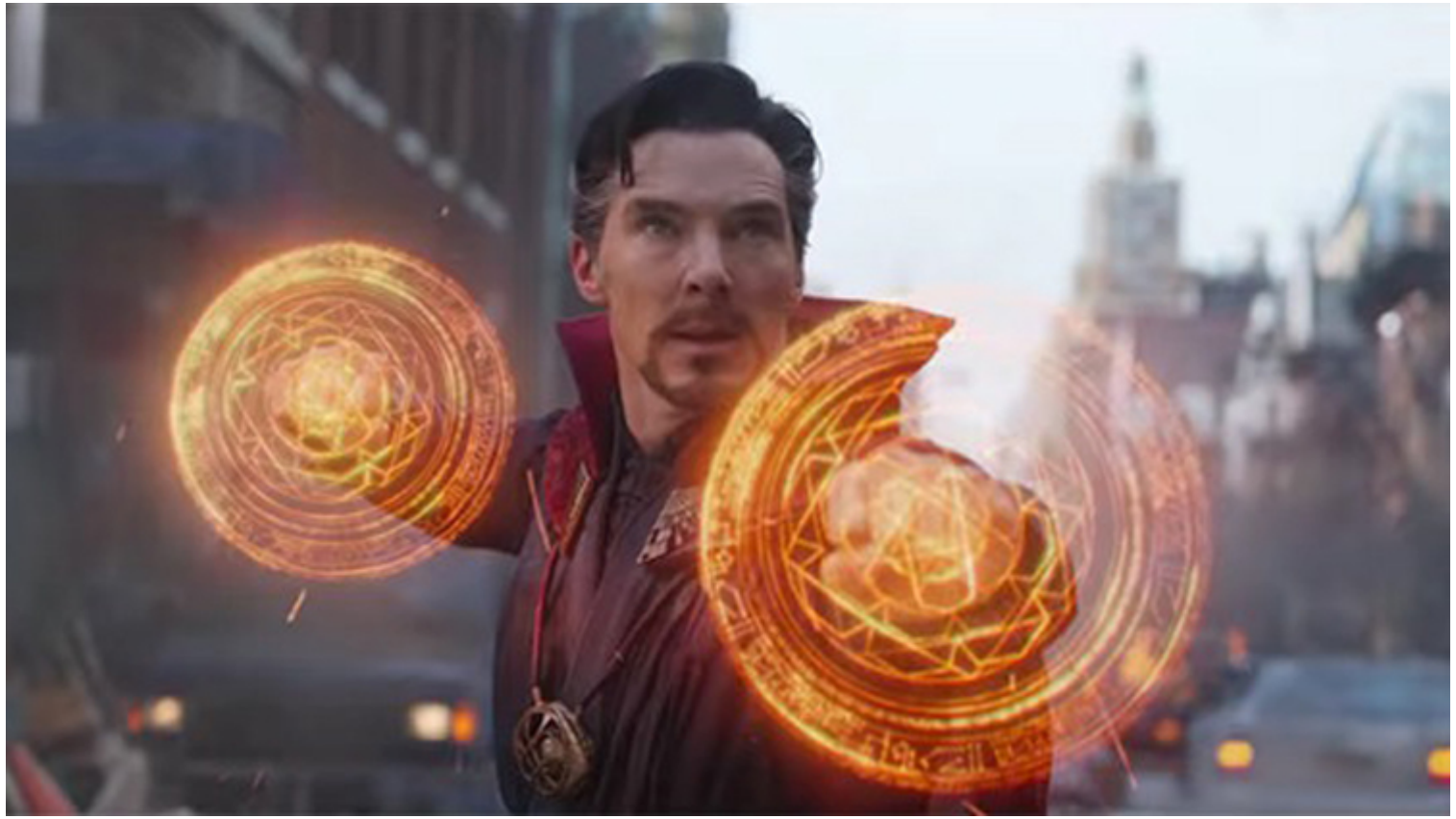
TOC :

Introducing `HandShield` - a real-time interactive augmented reality application that tracks hand gestures and draws a virtual shield on your hand, protecting you from any imaginary dangers!

Built using the power of `OpenCV and MediaPipe` , HandShield tracks the landmarks of your hand in real-time and creates a shield that moves along with your hand movements. The application can distinguish between an open hand gesture and a closed fist, and reacts accordingly to draw the virtual shield.

HandShield is a fun and interactive way to explore augmented reality technology, and its open-source code allows for customization and experimentation. Users can change the design and color of the shield, as well as adjust the size and rotation speed of the shield based on their preferences.

With its easy-to-use interface and interactive capabilities, HandShield is a perfect project for those interested in computer vision, augmented reality, and creative programming. Whether you're looking to learn more about computer vision or just have some fun with augmented reality, HandShield is a great project to get started with. Try it out today and see the magic of augmented reality come to life on your hand!



Keywords: - Computer Vision, Hand Tracking, Gesture Recognition, Image Processing, OpenCV, MediaPipe.

1. Setup

Packages to be installed:

1. pip install opencv-python
2. pip install mediapipe

Note :

Installing Visual C++ Redistributable is necessary for the HandShield project because it installs the Microsoft C and C++ (MSVC) runtime libraries that are required by many applications built using Microsoft C and C++ tools. These libraries are used by the OpenCV and MediaPipe libraries, which are

written in C++ and require a C++ compiler to build. By installing the Visual C++ Redistributable, the necessary runtime libraries are installed on the target system, ensuring that the HandShield application can run smoothly without any missing dependencies.

Click here to install the Visual C++ Redistributable.

2. Installing Packages

```
pip install opencv-python
```

```
pip install mediapipe
```

3. Project - Folder Structure

```
.
├── Models                  # Models to be used when creating the filter
│   ├── Inner Circles       # Circle models to be found on the inner side
│   └── Outer Circles       # Circle models to be found on the outer side
├── functions.py
├── main.py
├── requirements.txt
```

The project folder contains two Python code files named `functions.py` and `main.py`. Additionally, it includes a folder containing images for the `inner and outer circles`. There is also a `requirements.txt` file that lists the required Python packages for running the project. Overall, the folder provides all the necessary files for running the Handshield project.

4. `functions.py`

This module is designed to work with OpenCV to process and draw on video frames captured from a webcam. It includes several functions that perform different tasks.

The position_data function calculates the position of specific points on a hand detected by a hand tracking algorithm. For example, it can calculate the position of the tip of a finger or the wrist.

The calculate_distance function helps to calculate the distance between two given points.

The draw_line function allows drawing a line between two points on a video frame. This function can be useful for visualizing the position of specific points on the hand or to draw a line indicating a gesture.

Finally, the asd function overlays an image on the video frame at a specified location. The function also allows you to resize the image if needed. This function is useful when you want to display an image, such as an icon, on the video frame.

In [ ]:
```
import cv2 as cv
# imports the OpenCV library and aliases it to "cv" for easier use in the code.

LINE_COLOR = (0, 140, 255)
# sets a variable named "LINE_COLOR" to a tuple containing the values for blue, green, and red color channels, respectively. In th
```

The `position_data` function takes in a list of landmarks (lmlist) that correspond to the different parts of a hand detected by a hand tracking algorithm. The function extracts the coordinates of specific points on the hand, such as the tips of the fingers, the base of the thumb, and the MCP (metacarpophalangeal) joint of each finger. These coordinates are stored in tuples, which are then returned as a list containing all the points in a specific order. The order of the points is important and must match the order in which the points were extracted by the hand tracking algorithm.

In [ ]:
```
def position_data(lmlist):
    # Assign the (x, y) coordinates of the first landmark point (typically the base of the hand) to the variable wrist.
    wrist = (lmlist[0][0], lmlist[0][1])

    # Assign the (x, y) coordinates of the fifth landmark point (typically the tip of the thumb) to the variable thumb_tip.
    thumb_tip = (lmlist[4][0], lmlist[4][1])

    # Assign the (x, y) coordinates of the sixth landmark point (typically the metacarpophalangeal joint of the index finger) to t
    index_mcp = (lmlist[5][0], lmlist[5][1])

    # Assign the (x, y) coordinates of the ninth landmark point (typically the tip of the index finger) to the variable index_tip.
    index_tip = (lmlist[8][0], lmlist[8][1])

    # Assign the (x, y) coordinates of the tenth landmark point (typically the metacarpophalangeal joint of the middle finger) to
    midle_mcp = (lmlist[9][0], lmlist[9][1])

    # Assign the (x, y) coordinates of the thirteenth landmark point (typically the tip of the middle finger) to the variable midl
    midle_tip = (lmlist[12][0], lmlist[12][1])

    # Assign the (x, y) coordinates of the seventeenth landmark point (typically the tip of the ring finger) to the variable ring_
    ring_tip  = (lmlist[16][0], lmlist[16][1])
```

```
    # Assign the (x, y) coordinates of the twenty-first landmark point (typically the tip of the pinky finger) to the variable pin
    pinky_tip = (lmlist[20][0], lmlist[20][1])

    # Return a list of all the coordinate tuples calculated above, in the order: wrist, thumb_tip, index_mcp, index_tip, midle_mcp
    return [wrist, thumb_tip, index_mcp, index_tip, midle_mcp, midle_tip, ring_tip, pinky_tip]
```

The function `calculate_distance` takes in two points as input parameters and calculates the Euclidean distance between them. The coordinates of the two points are extracted and assigned to x1, y1, x2, and y2. The Euclidean distance formula is then applied to the coordinates to compute the distance between the two points. The result is returned as a float value representing the length of the line segment connecting the two points.

In [ ]:

```
def calculate_distance(p1, p2):
    # This function takes in two parameters p1 and p2, which are expected to be tuples of (x, y) coordinates for two points.
    x1, y1, x2, y2 = p1[0], p1[1], p2[0], p2[1]
    # This line unpacks the x and y coordinates from the two input tuples into four separate variables for easier calculation.
    length = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** (1.0 / 2)
    # This line calculates the Euclidean distance between the two points using the distance formula: sqrt((x2 - x1)^2 + (y2 - y1)^
    return length
    # This line returns the calculated distance as a float value.
```

The function `draw_line` takes in four parameters: frame, p1, p2, color, and size. It draws a line on the frame between points p1 and p2 with a specified color and size. Additionally, it draws a second line with half the specified size in white to create an outline effect. The function then returns the modified frame.

In [ ]:

```
def draw_line(frame, p1, p2, color=LINE_COLOR, size=5):
    # This function draws a line on the input frame between two given points.
    # Arguments:
    # - frame: The input image on which to draw the line.
    # - p1: The first point as a tuple (x, y) to start the line.
    # - p2: The second point as a tuple (x, y) to end the line.
    # - color: The color of the line to draw, specified as an RGB tuple (R, G, B).
    # - size: The thickness of the line to draw, specified as an integer.
    # Returns:
    # - frame: The original frame with the drawn line added to it.

    # Draw the primary line with the given color and size.
    cv.line(frame, p1, p2, color, size)

    # Draw a white line on top of the primary line to make it stand out more.
    # The white line is half the thickness of the primary line.
    cv.line(frame, p1, p2, (255, 255, 255), round(size / 2))
```

```
    # Return the original frame with the drawn line added to it.
    return frame
```

This function `asd` overlays a target image onto a frame at specified coordinates. An optional parameter can be used to resize the target image. It splits the target image into color channels and an alpha channel, merges the color channels, and creates a mask from the alpha channel. The function selects a region of interest in the frame to place the overlaid image and uses bitwise operations to combine the ROI with a copy of itself and the inverted mask to create a black background. It then applies the masked overlay color to the black background using bitwise operations and combines the black background and masked overlay color to create the final overlaid image.

In [ ]:

```
def asd(targetImg, frame, x, y, size=None):
    # Check if a custom size is provided, and resize the target image accordingly
    if size is not None:
        targetImg = cv.resize(targetImg, size)

    # Create a copy of the original frame
    newFrame = frame.copy()

    # Split the target image into separate color channels and alpha channel
    b, g, r, a = cv.split(targetImg)

    # Merge the color channels back together into an RGB image
    overlay_color = cv.merge((b, g, r))

    # Create a mask from the alpha channel of the target image, and apply median blur to smooth it
    mask = cv.medianBlur(a, 1)

    # Get the height and width of the overlaid image
    h, w, _ = overlay_color.shape

    # Select a region of interest (ROI) in the frame to place the overlaid image
    roi = newFrame[y:y + h, x:x + w]

    # Use bitwise operations to combine the ROI with a copy of itself and the inverted mask to create a black background
    img1_bg = cv.bitwise_and(roi.copy(), roi.copy(), mask=cv.bitwise_not(mask))

    # Use bitwise operations to apply the masked overlay color to the black background
    img2_fg = cv.bitwise_and(overlay_color, overlay_color, mask=mask)

    # Combine the black background and masked overlay color to create the final overlaid image
    newFrame[y:y + h, x:x + w] = cv.add(img1_bg, img2_fg)
```

```
    # Return the final frame with the overlaid image
    return newFrame
```

5. main.py

The given code is a Python script that uses the OpenCV and Mediapipe libraries to track hand landmarks and overlay images on top of the hand.

The script starts by importing the necessary libraries and defining some constants. It then sets up the camera and the mediapipe hands module.

In the main loop, the script reads a frame from the camera and converts it to RGB format. The mediapipe hands module is then used to detect hand landmarks in the frame. If landmarks are detected, the script extracts the coordinates of the hand landmarks and calculates the distance between certain landmark points to determine if the hand is in a specific position.

If the hand is in the desired position, the script draws lines connecting the hand landmarks on the frame. If the hand is in a different position, the script overlays two images of circles rotating in opposite directions on top of the hand. The size and position of the circles are based on the hand landmarks.

Finally, the script displays the modified frame and waits for the "q" key to be pressed to exit the loop and close the window.

In [ ]:
```
# Import required libraries
import cv2 as cv
import mediapipe as mp
from functions import position_data, calculate_distance, draw_line, asd

# Paths for the inner and outer circle images
INNER_CIRCLE = "Models/inner_circles/orange.png"
OUTER_CIRCLE = "Models/outer_circles/orange.png"
```

In [ ]:
```
# Camera Setup
cap = cv.VideoCapture(0)
# 0 is passed to VideoCapture to use the default camera. If you have multiple cameras, you can select a specific camera by passing

cap.set(3, 1280) # set width to 1280 pixels
cap.set(4, 720) # set height to 720 pixels

# Mediapipe setup for handlandmarks
```

```
mpDraw = mp.solutions.drawing_utils # module that provides drawing functions for visualization
mpHands = mp.solutions.hands # module for hand landmark detection
hands = mpHands.Hands() # instantiate the Hands object

# Initials
inner_circle = cv.imread(INNER_CIRCLE, -1) # Load inner circle image with alpha channel (-1)
outer_circle = cv.imread(OUTER_CIRCLE, -1) # Load outer circle image with alpha channel (-1)

LINE_COLOR = (0, 140, 255) # Set line color to a shade of orange
deg = 0 # Initialize degree variable to 0. This will be used later on.
```

In [ ]:

```
while cap.isOpened():
# Read camera frame
    _, frame = cap.read()
    # Flip frame horizontally
    frame = cv.flip(frame, 1)

    # Convert frame to RGB color space
    rgbFrame = cv.cvtColor(frame, cv.COLOR_BGR2RGB)

    # Detect hand landmarks using Mediapipe
    results = hands.process(rgbFrame)

    # If hand landmarks are detected, iterate through each hand and draw landmarks
    if results.multi_hand_landmarks:
        for hand in results.multi_hand_landmarks:
            # Initialize empty list for landmark coordinates
            lmLists = []

            # Iterate through each landmark and add its coordinates to lmLists
            for id, lm in enumerate(hand.landmark):
                # Get frame dimensions
                h, w, _ = frame.shape

                # Append landmark coordinates to lmLists
                lmLists.append([int(lm.x * w), int(lm.y * h)])

                # The x and y coordinates of the landmarks are normalized with respect to the width and height of the frame, so th
                # The landmark coordinates are appended to lmLists as a list of [x,y] values.

            # lmLists contains the pixel coordinates of each landmark for the current hand
            # This information can be used to draw connections between landmarks or to perform other actions on the hand landmarks
```

```
        # Get the wrist, thumb tip, index finger metacarpal, index finger tip, middle finger metacarpal, middle finger tip, ring f
        coordinates = position_data(lmLists)
        wrist, thumb_tip, index_mcp, index_tip, midle_mcp, midle_tip, ring_tip, pinky_tip = coordinates[0],coordinates[1], coordin

        # Calculate the distance between the index finger metacarpal and wrist, and the distance between the
        # index finger tip and pinky finger tip
        index_wrist_distance = calculate_distance(wrist, index_mcp)
        index_pinks_distance = calculate_distance(index_tip, pinky_tip)

        # Calculate the ratio between the two distances
        ratio = index_pinks_distance/index_wrist_distance

        if (1.3 > ratio > 0.5):
            frame=draw_line(frame, wrist, thumb_tip)
            frame=draw_line(frame, wrist, index_tip)
            frame=draw_line(frame, wrist, midle_tip)
            frame=draw_line(frame, wrist, ring_tip)
            frame=draw_line(frame, wrist, pinky_tip)
            frame=draw_line(frame, thumb_tip, index_tip)
            frame=draw_line(frame, thumb_tip, midle_tip)
            frame=draw_line(frame, thumb_tip, ring_tip)
            frame=draw_line(frame, thumb_tip, pinky_tip)
        # Checks if "ratio" is between 0.5 and 1.3
            if (1.3 > ratio > 0.5):
                frame=draw_line(frame, wrist, thumb_tip) # Draws a line between "wrist" and "thumb_tip" on "frame" and updates "fr
                frame=draw_line(frame, wrist, index_tip) # Draws a line between "wrist" and "index_tip" on "frame" and updates "fr
                frame=draw_line(frame, wrist, midle_tip) # Draws a line between "wrist" and "midle_tip" on "frame" and updates "fr
                frame=draw_line(frame, wrist, ring_tip) # Draws a line between "wrist" and "ring_tip" on "frame" and updates "fram
                frame=draw_line(frame, wrist, pinky_tip) # Draws a line between "wrist" and "pinky_tip" on "frame" and updates "fr
                frame=draw_line(frame, thumb_tip, index_tip) # Draws a line between "thumb_tip" and "index_tip" on "frame" and upd
                frame=draw_line(frame, thumb_tip, midle_tip) # Draws a line between "thumb_tip" and "midle_tip" on "frame" and upd
                frame=draw_line(frame, thumb_tip, ring_tip) # Draws a line between "thumb_tip" and "ring_tip" on "frame" and updat
                frame=draw_line(frame, thumb_tip, pinky_tip) # Draws a line between "thumb_tip" and "pinky_tip" on "frame" and upd

            elif (ratio > 1.3):
            # Executes if "ratio" is greater than 1.3.
                centerx = midle_mcp[0]
                centery = midle_mcp[1]
                # Sets the x and y coordinates of the center of the circle to "centerx" and "centery", respectively, based on the
                shield_size = 3.0
                # Sets the size of the shield.
                diameter = round(index_wrist_distance * shield_size)
                # Computes the diameter of the circle based on the distance between "index_tip" and "wrist" and the shield size.
                x1 = round(centerx - (diameter / 2))
```

```
                y1 = round(centery - (diameter / 2))
                # Computes the x and y coordinates of the top-left corner of the bounding box that will contain the circle.
                h, w, c = frame.shape
                if x1 < 0:
                    x1 = 0
                elif x1 > w:
                    x1 = w
                if y1 < 0:
                    y1 = 0
                elif y1 > h:
                    y1 = h
                # Ensures that the top-left corner of the bounding box does not exceed the dimensions of the "frame".
                if x1 + diameter > w:
                    diameter = w - x1
                if y1 + diameter > h:
                    diameter = h - y1
                # Adjusts the diameter of the circle if the bounding box exceeds the dimensions of the "frame".
                shield_size = diameter, diameter
                # Updates the shield size based on the computed diameter.
                ang_vel = 2.0
                deg = deg + ang_vel
                if deg > 360:
                    deg = 0
                # Computes the angular velocity of the shield and updates the degree of rotation. Resets the degree of rotation to
                hei, wid, col = outer_circle.shape
                cen = (wid // 2, hei // 2)
                M1 = cv.getRotationMatrix2D(cen, round(deg), 1.0)
                M2 = cv.getRotationMatrix2D(cen, round(360 - deg), 1.0)
                rotated1 = cv.warpAffine(outer_circle, M1, (wid, hei))
                rotated2 = cv.warpAffine(inner_circle, M2, (wid, hei))
                # Computes the rotation matrix and applies the rotation to the outer and inner circles.
                if (diameter != 0):
                    frame = asd(rotated1, frame, x1, y1, shield_size)
                    frame = asd(rotated2, frame, x1, y1, shield_size)
                # If the diameter of the circle is non-zero, overlays the rotated outer and inner circles onto the "frame".

    cv.imshow("Image", frame)
    # Display the image frame in a window named "Image"
    if cv.waitKey(1) == ord("q"):
        break
    # Wait for a key event, and if the pressed key is "q" (which has an ASCII code of 113), break out of the loop

cap.release()
```

```
cv.destroyAllWindows()
# Release the video capture object and destroy all windows created by OpenCV
```
