HAND-GESTURE APPLICATION

A PROJECT REPORT

ON COMPUTER VISION

Submitted by

P.K.AISHWARYA (913121104007)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING
COMPUTER SCIENCE AND ENGINEERING

VELAMMAL COLLEGE OF ENGINEERING AND TECHNOLOGY

(AUTONOMOUS)

MADURAI-625009

DECEMBER-2023

TABLE OF CONTENTS

ABSTRACT

The Hand-Tracking Paint Application represents a novel approach to interactive digital art creation, leveraging advanced computer vision techniques and hand-tracking technology. This innovative application enables users to draw on a virtual canvas using natural hand gestures captured in real-time by a webcam. The project focuses on delivering a seamless and engaging user experience by incorporating intuitive color selection and canvas-clearing gestures.

The core functionality of the application lies in the precise detection and tracking of hand landmarks, facilitated by the integration of the Mediapipe library. Users can choose from a palette of four colors (blue, green, red, and yellow) by positioning their hands in designated areas on the canvas. The fluidity of the color selection process enhances the user's creative expression.

One of the distinctive features of the application is the implementation of gesture-based canvas clearing. A specific hand movement—bringing the index finger close to the thumb—triggers the reset of the canvas, providing users with an effortless means to start new drawings.

The project's success is evidenced by the creation of an interactive and dynamic drawing environment, fostering creativity and user engagement. The robustness of the hand-tracking algorithm ensures accurate detection of gestures, contributing to the application's overall responsiveness.

The abstract also outlines potential avenues for improvement and future development, including the implementation of additional features such as varied brush sizes and shapes, optimization of hand tracking for increased accuracy, enhanced user feedback mechanisms, and exploration of multi-user interaction capabilities.

Overall, the Hand-Tracking Paint Application stands at the intersection of computer vision and digital art, offering a unique and accessible platform for users to explore their creativity through gesture-driven painting. The project's outcomes and recommendations provide valuable insights for further advancements in interactive digital art applications.

INTRODUCTION

The Hand-Drawn Painting Application represents an innovative and engaging exploration at the intersection of computer vision and interactive technology. In a digital era where traditional drawing tools are increasingly being complemented by advanced computing capabilities, this project introduces a unique and immersive approach to digital art creation. By harnessing the power of Python programming language and incorporating renowned libraries like OpenCV and Mediapipe, this application empowers users to embark on a creative journey through the expressive medium of hand-drawn digital painting.

1.1 Evolution of Digital Art:

The evolution of digital art has been marked by a continual quest to bridge the gap between traditional artistic expression and cutting-edge technology. As the boundaries between the physical and virtual worlds blur, artists and technologists alike seek novel ways to facilitate creativity. The Hand-Drawn Painting Application takes inspiration from this evolution, aiming to provide users with a dynamic and intuitive platform that seamlessly integrates the physical act of drawing with the computational prowess of computer vision.

1.2 Technological Foundation:

At its core, the project relies on two key technological pillars – OpenCV and Mediapipe. OpenCV, a widely-used open-source computer vision library, serves as the backbone for capturing real-time video frames from the webcam, image processing, and facilitating the drawing functionalities. On the other hand, Mediapipe, a versatile library developed by Google, specializes in hand tracking. Through the intricate analysis of hand landmarks, this library enables the application to interpret hand gestures, thus transforming hand movements into strokes on a virtual canvas.

User-Centric Design:

The user-centric design philosophy of the Hand-Drawn Painting Application places the creative experience at the forefront. By leveraging the natural dexterity of human hands, the application removes the barriers imposed by conventional input devices, such as a stylus or mouse. Users are invited to explore the digital canvas with the fluidity of their hand movements, fostering a more intuitive and direct connection between the artist and the artwork.

1.3 Inclusive Digital Creativity:

This project embodies the democratization of digital creativity, as it does not require users to possess specialized artistic tools or skills. The interactive nature of the Hand-Drawn Painting Application ensures that individuals from various backgrounds can engage in the artistic process, fostering inclusivity and making digital art accessible to a broader audience. As technology converges with artistic expression, this application represents a gateway for both seasoned artists and novices to explore the limitless possibilities of digital creation.

REQUIREMENTS

Software Requirements:

Python:

Python is the primary programming language used in this project.

OpenCV:

OpenCV is an open-source computer vision library used for image and video processing.
Install it using: pip install opencv-python

NumPy:

NumPy is used for numerical operations in Python.
Install it using: pip install numpy

Mediapipe:

Mediapipe is a library developed by Google for hand tracking and other related tasks.
Install it using: pip install mediapipe

Hardware Requirements:

Webcam:

A webcam is necessary for capturing the video feed in real-time. Most laptops havebuilt-in webcams, or you can use an external USB webcam.

Computer:

A computer with sufficient processing power to handle real-time video processing. Modern laptops or desktops should work well.

Operating System:

The code is platform-independent and should work on Windows, Linux, or macOS.

Python Environment:

It's recommended to use a virtual environment to manage project dependencies.

Mediapipe Compatibility:

Ensure that the version of Mediapipe you install is compatible with your Python version.

Webcam Compatibility:

Make sure that your webcam is functional and properly connected to your computer.

Additional Dependencies:

Check if any additional dependencies are required based on your operating system.

Example Setup:

Python Version:

Python 3.x

IDE (Optional):

You can use any Python-compatible IDE or a text editor for code development (e.g., VSCode, PyCharm).

Environment:

Create a virtual environment for the project to manage dependencies.

Webcam:

Ensure that your webcam is connected and functioning.

Libraries:

Install required libraries using the provided installation commands.

PROJECT EXECUTION

Importing Libraries:
cv2: OpenCV library for computer vision tasks.
numpy: Library for numerical operations.
mediapipe: Google's Mediapipe library for hand tracking.
deque: A double-ended queue, used for storing color points.

```
import cv2
import numpy as np
import mediapipe as mp
from collections import deque
```

Color Point Handling:
Four deques (bpoints, gpoints, rpoints, ypoints) are created to store color points for blue, green, red, and yellow respectively.
Four indices (blue_index, green_index, red_index, yellow_index) are initialized to keep track of the current index in each deque.

```
bpoints = [deque(maxlen=1024)]
gpoints = [deque(maxlen=1024)]
rpoints = [deque(maxlen=1024)]
ypoints = [deque(maxlen=1024)]
blue_index = 0
green_index = 0
red_index = 0
yellow_index = 0
```

Kernel for Dilation:
kernel is a 5x5 matrix of ones used for dilation purposes.

```
kernel = np.ones((5,5), np.uint8)
```

Color Information:
colors is a list of tuples representing the BGR values for blue, green, red, and yellow.
colorIndex is an index used to identify the selected color.

```
colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (0, 255, 255)]
```

colorIndex = 0

Canvas Setup:
paintWindow is initialized as a white canvas with a size of 471x636.
Rectangles and text are drawn on the canvas to create color selection buttons.

paintWindow = np.zeros((471,636,3)) + 255
 # (Rectangles and Text Drawing Code)
cv2.namedWindow('Paint', cv2.WINDOW_AUTOSIZE)
Mediapipe Hand Tracking Initialization:
mpHands and hands are initialized for hand tracking using Mediapipe.

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

Webcam Initialization:
OpenCV's VideoCapture is used to capture video from the default camera (index 0).

cap = cv2.VideoCapture(0)

Main Loop:
The program enters a loop where it continuously captures frames from the webcam.
The frame is flipped vertically to prevent it from being mirrored.
Hand landmarks are detected using Mediapipe.
Landmarks are processed to get the coordinates of the fingertips.
The position of the fingertips is used to determine user actions (e.g., selecting colors).
Color points are updated based on user actions.
Lines are drawn on the canvas and frame to visualize the drawing.
The loop continues until the user presses the 'q' key.

```
while ret:
    # (Frame Capture and Processing Code)
    # (Landmark Detection and Processing Code)
    # (User Action Processing Code)
    # (Drawing Code)

    if cv2.waitKey(1) == ord('q'):
        break
```

Cleanup:
The webcam is released, and all OpenCV windows are destroyed when the program exits.

cap.release()
cv2.destroyAllWindows()

This code creates a simple drawing application where the user can draw using their hand gestures in different colors on a canvas. The color selection and canvas clearing are controlled by the position of the hand in the frame.

CODING

```
# All the imports go here
import cv2
import numpy as np
import mediapipe as mp
from collections import deque


# Giving different arrays to handle colour points of different colour
bpoints = [deque(maxlen=1024)]
gpoints = [deque(maxlen=1024)]
rpoints = [deque(maxlen=1024)]
ypoints = [deque(maxlen=1024)]


# These indexes will be used to mark the points in particular arrays of specific colour
blue_index = 0
green_index = 0
red_index = 0
yellow_index = 0

#The kernel to be used for dilation purpose
kernel = np.ones((5,5),np.uint8)

colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (0, 255, 255)]
colorIndex = 0

# Here is code for Canvas setup
paintWindow = np.zeros((471,636,3)) + 255
paintWindow = cv2.rectangle(paintWindow, (40,1), (140,65), (0,0,0), 2)
paintWindow = cv2.rectangle(paintWindow, (160,1), (255,65), (255,0,0), 2)
paintWindow = cv2.rectangle(paintWindow, (275,1), (370,65), (0,255,0), 2)
paintWindow = cv2.rectangle(paintWindow, (390,1), (485,65), (0,0,255), 2)
paintWindow = cv2.rectangle(paintWindow, (505,1), (600,65), (0,255,255), 2)

cv2.putText(paintWindow, "CLEAR", (49, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
0, 0), 2, cv2.LINE_AA)
cv2.putText(paintWindow, "BLUE", (185, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
0, 0), 2, cv2.LINE_AA)
cv2.putText(paintWindow, "GREEN", (298, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
```

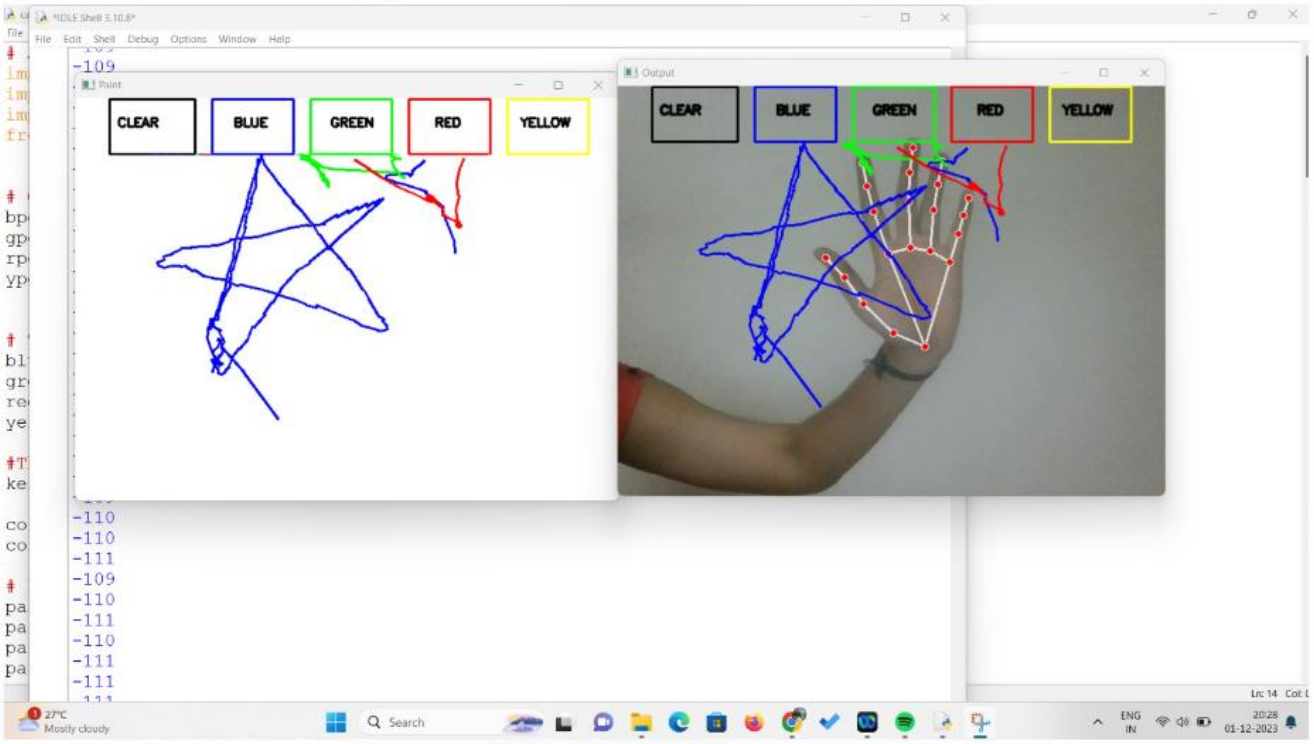
```
(0, 0, 0), 2, cv2.LINE_AA)
  cv2.putText(paintWindow, "RED", (420, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,
0, 0), 2, cv2.LINE_AA)
  cv2.putText(paintWindow, "YELLOW", (520, 33), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (0, 0, 0), 2, cv2.LINE_AA)
  cv2.namedWindow('Paint', cv2.WINDOW_AUTOSIZE)


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


  # Initialize the webcam
  cap = cv2.VideoCapture(0)
  ret = True
  while ret:
      # Read each frame from the webcam
      ret, frame = cap.read()

      x, y, c = frame.shape

      # Flip the frame vertically
      frame = cv2.flip(frame, 1)
      #hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
      framergb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

      frame = cv2.rectangle(frame, (40,1), (140,65), (0,0,0), 2)
      frame = cv2.rectangle(frame, (160,1), (255,65), (255,0,0), 2)
      frame = cv2.rectangle(frame, (275,1), (370,65), (0,255,0), 2)
      frame = cv2.rectangle(frame, (390,1), (485,65), (0,0,255), 2)
      frame = cv2.rectangle(frame, (505,1), (600,65), (0,255,255), 2)
      cv2.putText(frame, "CLEAR", (49, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0,
0), 2, cv2.LINE_AA)
      cv2.putText(frame, "BLUE", (185, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0),
2, cv2.LINE_AA)
      cv2.putText(frame, "GREEN", (298, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0,
0), 2, cv2.LINE_AA)
      cv2.putText(frame, "RED", (420, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0),
2, cv2.LINE_AA)
```

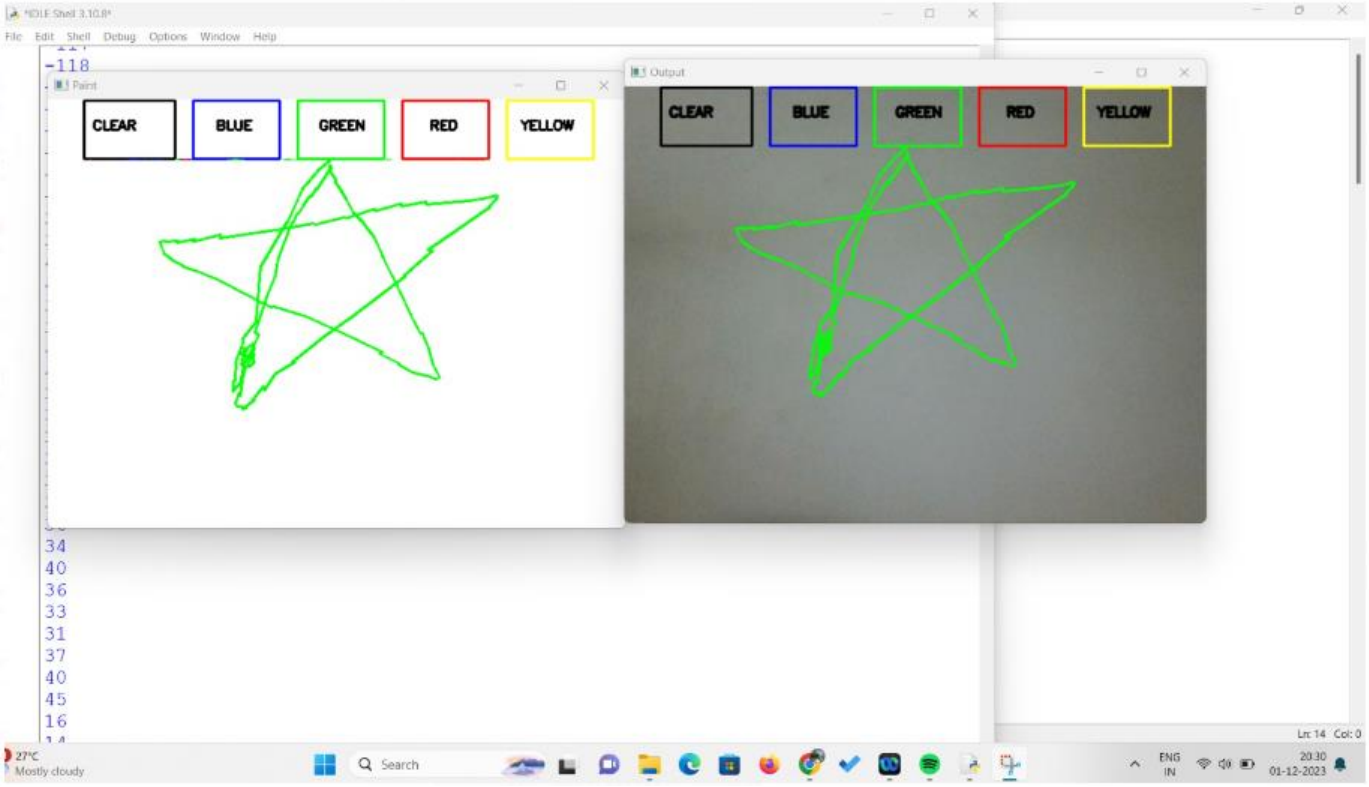
```
    cv2.putText(frame, "YELLOW", (520, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2, cv2.LINE_AA)
    #frame = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

    # Get hand landmark prediction
    result = hands.process(framergb)

    # post process the result
    if result.multi_hand_landmarks:
        landmarks = []
        for handslms in result.multi_hand_landmarks:
            for lm in handslms.landmark:
                # # print(id, lm)
                # print(lm.x)
                # print(lm.y)
                lmx = int(lm.x * 640)
                lmy = int(lm.y * 480)

                landmarks.append([lmx, lmy])



            # Drawing landmarks on frames
            mpDraw.draw_landmarks(frame, handslms, mpHands.HAND_CONNECTIONS)
        fore_finger = (landmarks[8][0],landmarks[8][1])
        center = fore_finger
        thumb = (landmarks[4][0],landmarks[4][1])
        cv2.circle(frame, center, 3, (0,255,0),-1)
        print(center[1]-thumb[1])
        if (thumb[1]-center[1]<30):
            bpoints.append(deque(maxlen=512))
            blue_index += 1
            gpoints.append(deque(maxlen=512))
            green_index += 1
            rpoints.append(deque(maxlen=512))
            red_index += 1
            ypoints.append(deque(maxlen=512))
            yellow_index += 1

        elif center[1] <= 65:
            if 40 <= center[0] <= 140: # Clear Button
                bpoints = [deque(maxlen=512)]
```

```
            gpoints = [deque(maxlen=512)]
            rpoints = [deque(maxlen=512)]
            ypoints = [deque(maxlen=512)]

            blue_index = 0
            green_index = 0
            red_index = 0
            yellow_index = 0

            paintWindow[67:,:,:] = 255
        elif 160 <= center[0] <= 255:
                colorIndex = 0 # Blue
        elif 275 <= center[0] <= 370:
                colorIndex = 1 # Green
        elif 390 <= center[0] <= 485:
                colorIndex = 2 # Red
        elif 505 <= center[0] <= 600:
                colorIndex = 3 # Yellow
    else :
        if colorIndex == 0:
            bpoints[blue_index].appendleft(center)
        elif colorIndex == 1:
            gpoints[green_index].appendleft(center)
        elif colorIndex == 2:
            rpoints[red_index].appendleft(center)
        elif colorIndex == 3:
            ypoints[yellow_index].appendleft(center)
# Append the next deques when nothing is detected to avois messing up
else:
    bpoints.append(deque(maxlen=512))
    blue_index += 1
    gpoints.append(deque(maxlen=512))
    green_index += 1
    rpoints.append(deque(maxlen=512))
    red_index += 1
    ypoints.append(deque(maxlen=512))
    yellow_index += 1

# Draw lines of all the colors on the canvas and frame
points = [bpoints, gpoints, rpoints, ypoints]
# for j in range(len(points[0])):
```

```
    #         for k in range(1, len(points[0][j])):
    #             if points[0][j][k - 1] is None or points[0][j][k] is None:
    #                 continue
    #             cv2.line(paintWindow, points[0][j][k - 1], points[0][j][k], colors[0], 2)
    for i in range(len(points)):
        for j in range(len(points[i])):
            for k in range(1, len(points[i][j])):
                if points[i][j][k - 1] is None or points[i][j][k] is None:
                    continue
                cv2.line(frame, points[i][j][k - 1], points[i][j][k], colors[i], 2)
                cv2.line(paintWindow, points[i][j][k - 1], points[i][j][k], colors[i], 2)

    cv2.imshow("Output", frame)
    cv2.imshow("Paint", paintWindow)

    if cv2.waitKey(1) == ord('q'):
        break

# release the webcam and destroy all active windows
cap.release()
cv2.destroyAllWindows()
```

OUTPUT:


CLEAR
BLUE
GREEN
RED
YELLOW
CLEAR
BLUE
GREEN
RED
YELLOW


CLEAR
BLUE
GREEN
RED
YELLOW
CLEAR
BLUE
GREEN
RED
YELLOW

PROJECT OUTCOME

Overview:
The Hand-Tracking Drawing Application successfully delivers an interactive and engaging experience for users, leveraging hand-tracking technology and computer vision. This section elaborates on the key outcomes of the project.

1. Interactive Drawing Application:
The primary achievement of the project is the creation of an interactive drawing application. Users can draw on a virtual canvas using hand gestures captured by a webcam. The real-time nature of the application provides a dynamic and immersive drawing experience.

2. Gesture-Based Color Selection:
The application introduces an intuitive color selection mechanism based on hand gestures. Users can choose between four distinct colors—blue, green, red, and yellow—by positioning their hand in the corresponding color selection areas on the canvas. This gesture-driven color selection adds a layer of user-friendly interaction to the drawing process.

3. Canvas Clearing with Gestures:
The project successfully implements a gesture for clearing the canvas. Users can reset the drawing canvas by performing a specific hand gesture, involving bringing the index finger close to the thumb. This functionality enhances the user experience by providing a quick and natural way to start afresh.

Recommendations for Improvement
While the Hand-Tracking Drawing Application achieves its core objectives, there are opportunities for enhancement and expansion. The following recommendations outline potential areas for improvement and future development:

1. Implement Additional Features:
To further enrich the user experience, consider implementing additional features, such as different brush sizes or shapes. Introducing variety in drawing tools allows users to express their creativity in diverse ways. The incorporation of brush customization options could enhance the application's versatility and appeal to a broader audience.

2. Optimize Hand Tracking for Improved Accuracy:
Improving the accuracy of hand tracking is crucial for the application's overall performance. Optimizing the hand-tracking algorithm, adjusting parameters, or exploring advanced techniques can contribute to more precise and reliable detection of hand movements. Enhanced accuracy ensures a smoother and more responsive drawing experience for users.

3. Enhance User Feedback:
Providing clear and informative feedback to users is essential for a seamless interaction. Consider enhancing visual cues for selected colors, such as highlighting the chosen color or displaying a tooltip. This improvement aids users in understanding their current drawing settings and contributes to a more user-friendly interface.

4. Explore Multi-User Interaction:
To expand the application's capabilities, consider exploring multi-user interaction. Implementing support for multiple hands simultaneously can enable collaborative drawing experiences. This feature opens up possibilities for shared creativity, making the application suitable for group activities or educational purposes.

Implementation Details:
1. Additional Features:
Brush Sizes and Shapes:
Introduce options for users to customize the brush size and shape. This could include a menu for selecting different brush presets or dynamically adjusting the brush size based on hand movements.
2. Optimization Techniques:
Parameter Tuning:
Experiment with different parameters in the hand-tracking algorithm to find optimal values for the current application. Fine-tuning parameters can lead to improved accuracy in detecting hand gestures.

Advanced Hand Tracking Techniques:
Explore advanced hand tracking techniques, such as integrating machine learning models for hand pose estimation. This could potentially enhance the robustness of hand tracking in various scenarios.

3. User Feedback Enhancements:
Color Highlighting:
When a user selects a color, consider implementing a visual highlight or border around the chosen color button. This provides immediate feedback on the selected color and reinforces the user's action.

Tooltip Display:
Display tooltips or brief descriptions when users hover over color buttons or other interactive elements. This helps users understand the purpose and functionality of each component.

4. Multi-User Interaction:

Support for Multiple Hands:
Extend the application to support multiple hands simultaneously. This feature could be particularly valuable for collaborative drawing sessions, allowing multiple users to contribute to a shared canvas.

Distinct Color Assignments:
Implement a system for assigning distinct colors to each user's hand, enabling collaborative drawing with identifiable contributions from each participant.

CONCLUSION:

The Hand-Tracking Drawing Application successfully delivers an interactive drawing experience, and the recommended improvements provide a roadmap for enhancing its features and performance. By addressing these suggestions, the application can evolve into a more versatile and user-friendly tool, appealing to a broader audience and facilitating collaborative creativity.

REFERENCE:

https://developers.google.com/mediapipe/solutions/vision/ python

https://docs.opencv.org/4.x/

https://pytorch.org/tutorials/

https://www.tensorflow.org/tutorials

https://blog.research.google/

https://blogs.nvidia.com/

https://pyimagesearch.com/blog/

Multiple View Geometry in Computer Vision by Richard Hartley and Andrew Zisserman

Computer Vision: Algorithms and Applications by Richard Szeliski