Air Drawing App using Computer Vision

**Subtitle**: Statement Code: CV-06

**By**: Tarlana Vikas

**Team:** Rising Sparks

**Hackathon**: HackZilla

Table of Contents

* Introduction
* Problem Statement
* Objective
* Required Libraries & Tools
* Methodology
* Code
* Code Explanation
* Output
* Output Screenshots
* Challenges Faced
* Future Enhancements
* Conclusion
* References

Introduction

This project is an innovative and interactive computer vision-based application that enables users to draw on a digital canvas using a colored object as a virtual pen. Built using Python and the OpenCV library, the application captures real-time video from the user's webcam and detects a specific color range through HSV (Hue, Saturation, Value) filtering. Trackbars are provided for users to adjust HSV values dynamically, making it easier to isolate and track the colored object accurately under varying lighting conditions.

Once the object is detected, contour detection is applied to find its position on the screen. These coordinates are then continuously recorded and used to draw lines on a virtual canvas, creating an experience similar to freehand drawing. The application interface includes virtual buttons at the top of the screen for changing brush colors (Blue, Green, Red, Yellow) and clearing the canvas. When the user moves the object over these buttons, the program responds accordingly by changing the drawing color or resetting the canvas.

This project not only demonstrates real-time object tracking and drawing but also highlights gesture-based interaction without any physical contact. It’s a fun and creative way to explore computer vision and has potential use in art applications, education, and touchless UI systems.

Problem Statement: Air Drawing App – CV-06

**Objective:**

The aim of this project is to develop an interactive Air Drawing application that allows users to draw digitally on a screen using the movement of a colored object, such as a marker or cap, held in front of a webcam. The application should be able to detect the object based on its color and track its motion in real-time to translate that movement into drawing actions on a virtual canvas.

**Detailed Description:**

This application leverages computer vision techniques to recognize and track a user-defined colored object using HSV (Hue, Saturation, Value) color filtering. Once detected, the position of the object is continuously monitored frame-by-frame through the webcam feed. The motion trail of the object is used to draw lines on a blank virtual canvas, mimicking the effect of drawing with a pen in mid-air. The application also includes on-screen buttons to change drawing colors and clear the canvas, providing users with simple gesture-based controls. The interface and drawing process should be responsive and visually smooth to enhance the user experience.

**Constraints:**

* The application must ensure accurate and consistent tracking of the object.
* It should provide real-time drawing feedback with minimal lag.
* Color detection should be adaptable under different lighting conditions.

Objective

The primary objective of this project is to develop a real-time virtual drawing application that enables users to draw on a digital canvas using a colored object, such as a marker or cap, as a pen. This is accomplished by utilizing a webcam to continuously capture the live video feed and applying computer vision techniques to detect and track the movement of the colored object. By identifying the object based on its unique color range in the HSV (Hue, Saturation, Value) color space, the system can determine its position on the screen.

As the object moves, its coordinates are recorded and used to simulate freehand drawing on a virtual canvas. Additionally, the application incorporates an intuitive user interface that allows users to switch between multiple colors—Blue, Green, Red, and Yellow—by hovering the object over designated on-screen buttons. A "Clear All" button is also available to reset the canvas with a simple gesture.

This project combines the power of image processing and gesture recognition to deliver a touchless, engaging drawing experience. It encourages creativity, offers an interactive way to learn computer vision, and demonstrates how technology can be used to develop contact-free human-computer interaction systems for various applications like education, art, and accessibility tools.

Required Libraries & Tools

To develop the Air Drawing application, several essential tools and libraries are used to implement real-time computer vision functionalities, object tracking, and drawing mechanisms. Below is a detailed description of each component:

1. **Python 3.x:**

Python serves as the core programming language for this project. Its simplicity, readability, and vast ecosystem of libraries make it ideal for rapid development of computer vision applications.

1. **OpenCV (Open Source Computer Vision Library):**

OpenCV is the primary library used in this project for image processing and real-time computer vision tasks. It provides functions to capture video from the webcam, convert images to HSV color space, create masks for color detection, find contours, and draw on both the live frame and virtual canvas.

1. **NumPy:**

NumPy is used for handling numerical operations and matrix manipulations. It helps in defining kernels for image morphological transformations like erosion and dilation, and is essential for efficient pixel-level operations.

1. **Webcam (Built-in or External):**

A webcam is necessary to capture live video input. The application utilizes the webcam feed to detect and track the colored object in real time.

1. **Optional Tools – Jupyter Notebook / IDE:**

Development can be carried out in any Python-supporting IDE such as PyCharm, VS Code, or Jupyter Notebook for better visualization, debugging, and iterative testing during development.

Methodology

The Air Drawing application is built using real-time computer vision techniques to allow users to draw on a virtual canvas by moving a colored object in front of a webcam. This project leverages OpenCV and NumPy libraries in Python to perform color detection, object tracking, and user interaction through gesture-based inputs. The following steps describe the working of the system in a detailed and sequential manner:

**Step 1: Access Webcam and Capture Frames**

The first step involves initializing the webcam using OpenCV’s cv2.VideoCapture(0) function, where 0 refers to the default webcam. This function begins capturing video input frame-by-frame. Each frame is a snapshot of the video at a given instant and serves as the input for the object detection and drawing mechanism. The frames are read continuously inside a while loop to allow for real-time tracking and responsiveness.

**Step 2: Convert BGR Frames to HSV**

Once a frame is captured, it is converted from the default BGR (Blue, Green, Red) color space to HSV (Hue, Saturation, Value) color space using OpenCV’s cv2.cvtColor() function. The HSV model is more effective for color-based segmentation because it separates color information (hue) from brightness (value), making it easier to isolate specific colors even under varying lighting conditions. This step is crucial for reliable object detection.

**Step 3: Use HSV Range Sliders (Trackbars) to Detect a Colored Marker**

To enable flexibility in detecting different colored objects, the application provides six trackbars (sliders) for adjusting the upper and lower HSV values. These trackbars allow the user to define the range of hue, saturation, and value that correspond to the color of the object they are using (e.g., a red cap or blue marker). OpenCV’s cv2.createTrackbar() function is used to create these interactive sliders. Using the specified HSV ranges, a binary mask is created using cv2.inRange() which highlights only the desired color and sets the rest of the frame to black.

**Step 4: Apply Morphological Operations to Clean the Mask**

The initial binary mask generated from color filtering may contain noise due to environmental factors such as lighting or background clutter. To refine this mask, morphological operations like erosion, opening, and dilation are applied using a kernel (typically a 5x5 matrix of ones). These operations remove small false positives, smooth the edges, and enhance the shape of the detected object. The goal is to isolate a single, clean contour representing the colored object.

**Step 5: Detect Contours and Locate the Center of the Marker**

Once the mask is cleaned, contours are extracted using cv2.findContours(). Contours are simply curves that join all continuous points of the same intensity or color. Among the detected contours, the one with the largest area is considered to be the colored marker. A minimum enclosing circle is drawn around this contour to visually indicate its detection. The centroid (center) of this circle is calculated using image moments, which provides the (x, y) coordinates of the object in the frame. This center point is then used to draw on the virtual canvas and interact with on-screen buttons.

**Step 6: Use Marker Position to Perform Actions**

This is the core functional step where the marker’s position drives various actions in the application. Based on the coordinates of the detected center, the system performs the following:

* **Draw Lines on Canvas:**

If the marker is within the drawing area (below the button region), its coordinates are appended to a deque (a fast and efficient double-ended queue). Each color has its own list of deque objects to store drawing paths. Using cv2.line(), the program connects the points from these deques to create smooth lines on both the live frame and the virtual canvas. This creates the illusion of drawing in air.

* **Switch Colors:**

The top portion of the screen contains virtual buttons represented by colored rectangles. If the marker’s center point is detected within one of these rectangles, the corresponding color index is updated. The next lines drawn will appear in that color. This interaction is completely touchless and relies solely on gesture-based positioning.

* **Clear Canvas:**

A “CLEAR” button is also placed among the color options. If the marker hovers over this button, all deques are reset and the canvas is wiped clean. This functionality provides a way to restart the drawing without manually restarting the program.

**Step 7: Display Final Canvas and Live Frame**

The application displays three windows using cv2.imshow():

1. **Tracking Window:** Shows the live webcam feed with marker detection and virtual buttons.
2. **Paint Window:** Shows the clean canvas with the final drawing.
3. **Mask Window:** Displays the binary mask for visual debugging of the color detection process.

These windows are updated continuously as the loop runs, creating a seamless and interactive experience. The program continues to run until the user presses the 'q' key, at which point the webcam is released and all OpenCV windows are closed.

Code

import numpy as np

import cv2

from collections import deque

#default called trackbar function

def setValues(x):

print("")

# Creating the trackbars needed for adjusting the marker colour

cv2.namedWindow("Color detectors")

cv2.createTrackbar("Upper Hue", "Color detectors", 153, 180,setValues)

cv2.createTrackbar("Upper Saturation", "Color detectors", 255, 255,setValues)

cv2.createTrackbar("Upper Value", "Color detectors", 255, 255,setValues)

cv2.createTrackbar("Lower Hue", "Color detectors", 64, 180,setValues)

cv2.createTrackbar("Lower Saturation", "Color detectors", 72, 255,setValues)

cv2.createTrackbar("Lower Value", "Color detectors", 49, 255,setValues)

# 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), colors[0], -1)

paintWindow = cv2.rectangle(paintWindow, (275,1), (370,65), colors[1], -1)

paintWindow = cv2.rectangle(paintWindow, (390,1), (485,65), colors[2], -1)

paintWindow = cv2.rectangle(paintWindow, (505,1), (600,65), colors[3], -1)

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, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(paintWindow, "GREEN", (298, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(paintWindow, "RED", (420, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(paintWindow, "YELLOW", (520, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (150,150,150), 2, cv2.LINE\_AA)

cv2.namedWindow('Paint', cv2.WINDOW\_AUTOSIZE)

# Loading the default webcam of PC.

cap = cv2.VideoCapture(0)

# Keep looping

while True:

# Reading the frame from the camera

ret, frame = cap.read()

#Flipping the frame to see same side of yours

frame = cv2.flip(frame, 1)

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

u\_hue = cv2.getTrackbarPos("Upper Hue", "Color detectors")

u\_saturation = cv2.getTrackbarPos("Upper Saturation", "Color detectors")

u\_value = cv2.getTrackbarPos("Upper Value", "Color detectors")

l\_hue = cv2.getTrackbarPos("Lower Hue", "Color detectors")

l\_saturation = cv2.getTrackbarPos("Lower Saturation", "Color detectors")

l\_value = cv2.getTrackbarPos("Lower Value", "Color detectors")

Upper\_hsv = np.array([u\_hue,u\_saturation,u\_value])

Lower\_hsv = np.array([l\_hue,l\_saturation,l\_value])

# Adding the colour buttons to the live frame for colour access

frame = cv2.rectangle(frame, (40,1), (140,65), (122,122,122), -1)

frame = cv2.rectangle(frame, (160,1), (255,65), colors[0], -1)

frame = cv2.rectangle(frame, (275,1), (370,65), colors[1], -1)

frame = cv2.rectangle(frame, (390,1), (485,65), colors[2], -1)

frame = cv2.rectangle(frame, (505,1), (600,65), colors[3], -1)

cv2.putText(frame, "CLEAR ALL", (49, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(frame, "BLUE", (185, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(frame, "GREEN", (298, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(frame, "RED", (420, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 2, cv2.LINE\_AA)

cv2.putText(frame, "YELLOW", (520, 33), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (150,150,150), 2, cv2.LINE\_AA)

# Identifying the pointer by making its mask

Mask = cv2.inRange(hsv, Lower\_hsv, Upper\_hsv)

Mask = cv2.erode(Mask, kernel, iterations=1)

Mask = cv2.morphologyEx(Mask, cv2.MORPH\_OPEN, kernel)

Mask = cv2.dilate(Mask, kernel, iterations=1)

# Find contours for the pointer after idetifying it

cnts,\_ = cv2.findContours(Mask.copy(), cv2.RETR\_EXTERNAL,

cv2.CHAIN\_APPROX\_SIMPLE)

center = None

# Ifthe contours are formed

if len(cnts) > 0:

# sorting the contours to find biggest

cnt = sorted(cnts, key = cv2.contourArea, reverse = True)[0]

# Get the radius of the enclosing circle around the found contour

((x, y), radius) = cv2.minEnclosingCircle(cnt)

# Draw the circle around the contour

cv2.circle(frame, (int(x), int(y)), int(radius), (0, 255, 255), 2)

# Calculating the center of the detected contour

M = cv2.moments(cnt)

center = (int(M['m10'] / M['m00']), int(M['m01'] / M['m00']))

# Now checking if the user wants to click on any button above the screen

if 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 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)

# Show all the windows

cv2.imshow("Tracking", frame)

cv2.imshow("Paint", paintWindow)

cv2.imshow("mask",Mask)

# If the 'q' key is pressed then stop the application

if cv2.waitKey(1) & 0xFF == ord("q"):

break

# Release the camera and all resources

cap.release()

cv2.destroyAllWindows()

Code Explanation

This section breaks down the important code segments and explains how each part contributes to the working of the Air Drawing application. The code is structured for clarity and interactivity using OpenCV in Python.

**1. Trackbar Creation and Usage**

Trackbars are created using cv2.createTrackbar() under the window titled "Color detectors". These allow users to dynamically adjust HSV (Hue, Saturation, Value) values to detect a specific colored object (e.g., a cap or marker). The following trackbars are defined:

cv2.createTrackbar("Upper Hue", "Color detectors", 153, 180, setValues)

Six trackbars (3 for upper and 3 for lower HSV) are used. The HSV values from the sliders are fetched in real time using cv2.getTrackbarPos(), providing flexibility and accuracy in object detection.

**2. Color Mask and Filtering Using HSV**

After converting the captured BGR frame to HSV using:

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

The HSV values from the trackbars are used to create an upper and lower threshold for filtering:

Mask = cv2.inRange(hsv, Lower\_hsv, Upper\_hsv)

This mask highlights only the detected colored object in white and sets the background to black. Morphological operations like erosion, opening, and dilation are then applied to clean the mask and remove noise, improving contour detection.

**3. Drawing Canvas and Color Buttons**

A blank white canvas paintWindow is created using NumPy to act as the drawing surface. Colored rectangles and a "CLEAR" button are drawn at the top of the frame and canvas:

paintWindow = cv2.rectangle(paintWindow, (160,1), (255,65), colors[0], -1)

cv2.putText(paintWindow, "BLUE", (185, 33), cv2.FONT\_HERSHEY\_SIMPLEX, ...)

These buttons allow the user to select the color they wish to draw with or clear the entire canvas by hovering the object over them.

**4. Contour Detection and Center Calculation**

Contours are extracted from the binary mask using:

cnts, \_ = cv2.findContours(Mask.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

The largest contour is selected and a circle is drawn around it. The center of this contour is calculated using image moments:

M = cv2.moments(cnt)

center = (int(M['m10'] / M['m00']), int(M['m01'] / M['m00']))

This center point is used to determine marker location for drawing and button interaction.

**5. Drawing Logic Using Deques**

Each color has its own list of deques to store drawing points (paths). The current center is appended to the corresponding color deque:

bpoints[blue\_index].appendleft(center)

These points are then connected using cv2.line() to draw smooth strokes on both the frame and the canvas. New deques are added when no object is detected to separate drawing strokes.

**6. Color Selection and Clearing Mechanism**

When the detected center falls within the coordinate bounds of a button, actions are triggered:

* **Color Switch:** The colorIndex is updated.
* **Clear Canvas:** All deques are reset, and the canvas is set to white.

if 40 <= center[0] <= 140: # Clear Button

bpoints = [deque(maxlen=512)]

...

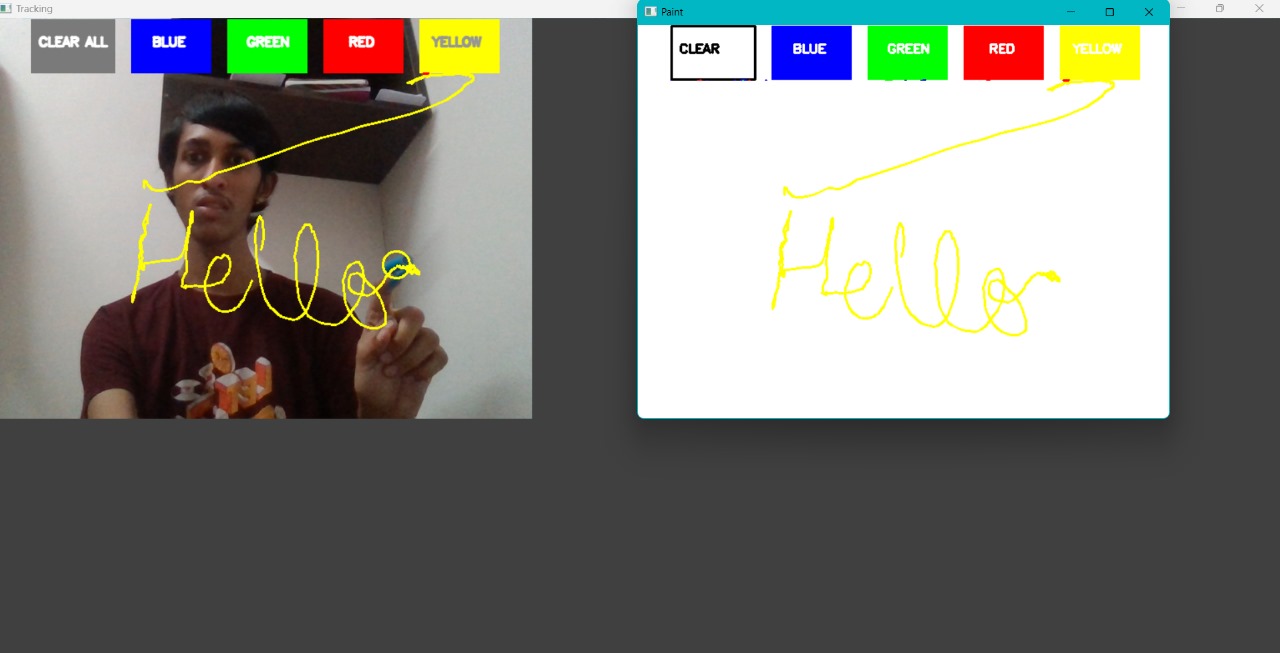
paintWindow[67:,:,:] = 255

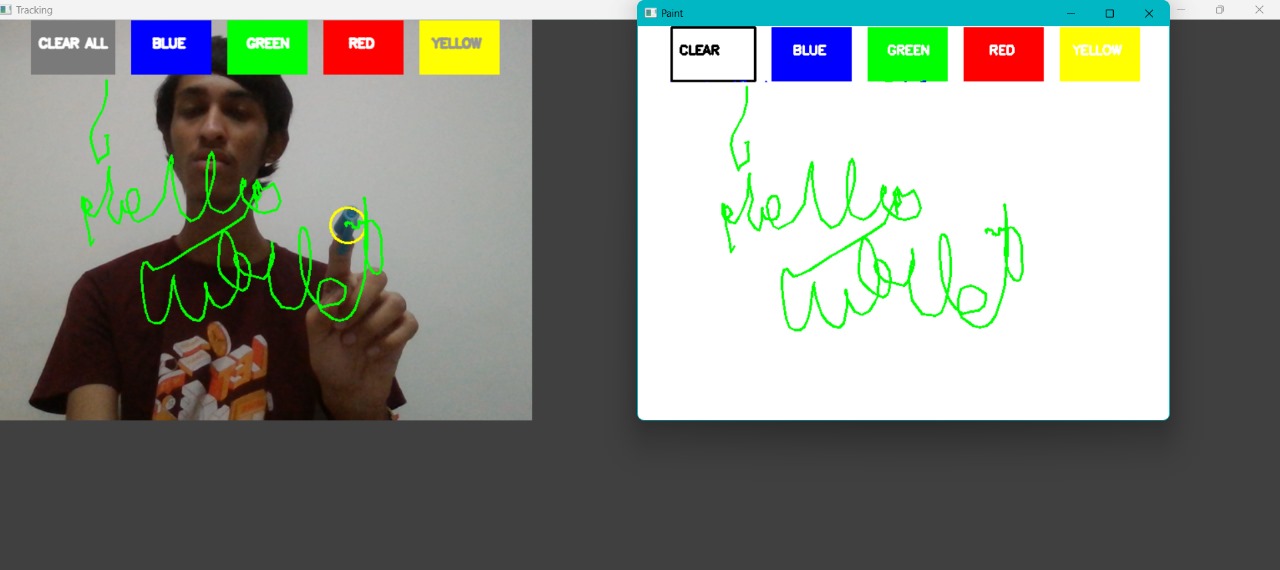
This logic allows for intuitive, gesture-based control without physical contact.

Output

1. **Initial Interface with Trackbars:** This screenshot displays the main interface when the program starts. It includes the trackbars for adjusting the HSV values, enabling precise control over color detection. The virtual canvas is initialized, with color selection and the “CLEAR” button positioned at the top.
2. **Marker Detection in Action:** This image highlights how the program successfully detects the colored marker using HSV filtering. The marker is enclosed within a circular boundary, demonstrating accurate tracking of its position in real-time.
3. **Color Switching While Drawing:** A screenshot showing the user switching between different colors—Blue, Green, Red, and Yellow—by hovering over the corresponding virtual buttons. This step verifies the smooth gesture-based interaction for color selection.
4. **Clear Button in Use:** This captures the moment when the user activates the "CLEAR" button. All stored drawing points reset, wiping the canvas clean. The functionality ensures users can restart without restarting the program.
5. **Final Painted Output:** Displays the complete drawing created using marker movements, demonstrating the application’s ability to simulate freehand air drawing.

**Output Screenshots:**





Challenges Faced

* Developing the Air Drawing App presented various challenges that required careful tuning and optimization to ensure smooth functionality. One of the primary hurdles was **tuning the HSV values for accurate marker detection**. Since lighting conditions can vary significantly, the system needed adaptive HSV thresholds to detect the colored object reliably. Manual adjustments via trackbars were introduced to allow real-time modifications and improve detection accuracy.
* Another challenge was **managing deques efficiently to prevent unwanted drawing** when the marker was not detected. If the system failed to recognize the object at certain moments, residual points could create unintended lines. This issue was mitigated by appending empty deques when the marker was absent, ensuring drawing continuity only when actual detection occurred.
* **Ensuring smooth color switching** presented difficulties, as users needed an intuitive way to transition between different colors. The system relied on predefined virtual buttons, but detecting accurate selections required careful calibration. Implementing boundary checks prevented accidental color switches and ensured precise selection when the marker hovered over the designated areas.
* Lastly, **handling frame rate and lag** was a concern for maintaining fluid interaction. Processing real-time video frames and tracking objects could introduce latency. Optimization techniques such as limiting deque sizes, applying lightweight image processing functions, and reducing redundant computations were employed to sustain a high frame rate and responsive experience.

Future Enhancements

* **Touchless Color Selector Using Gestures**
  + Eliminates the need for virtual buttons.
  + Users can switch colors by making specific gestures with their hand.
  + Enhances the intuitive, contact-free experience.
* **Shape Drawing (Circles, Rectangles)**
  + Allows users to draw geometric shapes with simple gestures.
  + Could include predefined gestures for selecting shapes.
  + Improves precision for structured drawings.
* **Save Canvas as an Image File**
  + Adds functionality to save artwork in PNG or JPG format.
  + Enables users to preserve and share their drawings.
  + Increases usability for digital artists or educational tools.
* **Multi-Marker Support**
  + Enables multiple objects for collaborative drawing.
  + Each marker could correspond to a different color or function.
  + Expands possibilities for interactive team-based drawing.
* **Voice-Based Commands**
  + Integrates speech recognition for hands-free interactions.
  + Commands like "change color to red" or "clear canvas" could be used.
  + Enhances accessibility for users with mobility impairments.

Conclusion

This project effectively showcases the power of computer vision in enabling real-time human-computer interaction without any physical contact. By leveraging Python and OpenCV, it provides an interactive experience where users can draw on a virtual canvas simply by moving a colored object in front of a webcam. The system tracks the object’s position using HSV-based color detection and converts its motion into digital strokes on the canvas.

One of the key highlights of this project is its touch-free functionality, making it a suitable solution for a variety of applications, including interactive kiosks, assistive technology for individuals with motor impairments, and engaging learning tools for children. The integration of virtual buttons allows users to switch colors effortlessly or clear the canvas, ensuring a seamless experience.

Additionally, the project serves as an excellent example of gesture-based interfaces, illustrating how computer vision algorithms can replace traditional input methods like a mouse or touchscreen. The implementation of real-time contour detection and tracking ensures a smooth and responsive interaction.

Moving forward, this technology could be extended to support multi-marker recognition, gesture-based shape drawing, AI-assisted corrections, and voice-enabled commands. Overall, the Air Drawing App merges creativity and technology, proving how computer vision can enhance intuitive, contact-free human-computer interaction, opening doors to future advancements in digital art, accessibility, and education.

References

To support the development of the Air Drawing application, several resources and technical documentation were referenced:

* **OpenCV Documentation:** Provides comprehensive details on OpenCV functions used for image processing, color detection, contour finding, and real-time object tracking.
  + Link: https://docs.opencv.org/
* **Stack Overflow Discussions:** Various coding challenges were solved by referring to relevant Stack Overflow threads discussing HSV tuning, frame processing, and performance optimization in Python.
* **GitHub Code Examples:** Open-source repositories with similar virtual drawing applications were examined and adapted to enhance functionality, particularly in color switching and gesture-based interaction.
* **Tutorials & Guides:** Online tutorials on computer vision, OpenCV basics, HSV color filtering, and object tracking helped in structuring the project efficiently. Some guides covered contour detection techniques and real-time video processing optimizations.