CSE 4128: Image processing and Computer Vision Laboratory

# Hill Climb Adventure: Hand Gestures Automation

By,

Sonjoy Roy

Roll:1907073

Date of Submission: 02-09-2024

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh

**Contents**

1 Introduction

2 Objective

3 Project Idea

4 Project Overview

5 Methodology

6 Programming Environment

7 Programming Language

8 Key Features

9 Final Output

10 Limitations

11 Discussion

12 Conclusion

## Introduction

**Hill Climb Adventure: Hand Gestures Automation** is an innovative project that utilizes computer vision techniques to enhance the gaming experience by allowing users to control the popular Hill Climb game using hand gestures. The project utilizes OpenCV for image processing and pyautogui for simulating keyboard inputs, creating an interactive and immersive way for players to engage with the game.

## Objective

The main objectives of the **Hill Climb Adventure: HandGestures Automation project** are:

1. Develop a Real-Time Hand Gesture Recognition System.
2. Accurately Count Fingers for Gesture Recognition.
3. Integrate Hand Gestures with Game Controls.
4. Provide Real-Time Feedback and Visualization.
5. Calibrate the system to adapt to different lighting conditions and hand positions.

## Project Idea.

Hill Climb Adventure: HandGestures Automation is designed to revolutionize the way players interact with the Hill Climb game by replacing traditional keyboard controls with intuitive hand gestures. This project combines the power of computer vision to recognize hand gestures in real-time and translate them into game actions, creating an engaging and immersive gaming experience. The system captures video frames from a webcam, processes them to detect hand movements, and uses these movements to control the game's vehicle. This innovative approach not only enhances the gameplay but also demonstrates the potential of hand gesture recognition technology in gaming and other interactive applications.

## Project Overview

The project utilizes OpenCV, a powerful library for computer vision, to process video frames captured by a webcam. The key steps involved in the project include:

1. Capturing video input from the webcam.
2. Defining a region of interest (ROI) for hand gesture detection.
3. Calibrating the background for accurate hand segmentation.
4. Detecting and segmenting the hand region.
5. Contour Analysis and Finger Counting.
6. The recognized gestures are mapped to specific game actions using pyautogui.

## Methodology

### 1 Video Capture and Preprocessing

Frames are captured from a webcam using OpenCV (cv2.VideoCapture). Then parameters such as frame dimensions (FRAME_HEIGHT, FRAME_WIDTH), calibration time (CALIBRATION_TIME), background update weight (BG_WEIGHT), object threshold (OBJ_THRESHOLD), and buffers for gesture recognition are initialized. An initial image (test.jpg) is loaded using cv2.imread() function from local disk for manipulation and the image is resized to a smaller size (150x150) for overlaying on camera frames using cv2.resize() function.

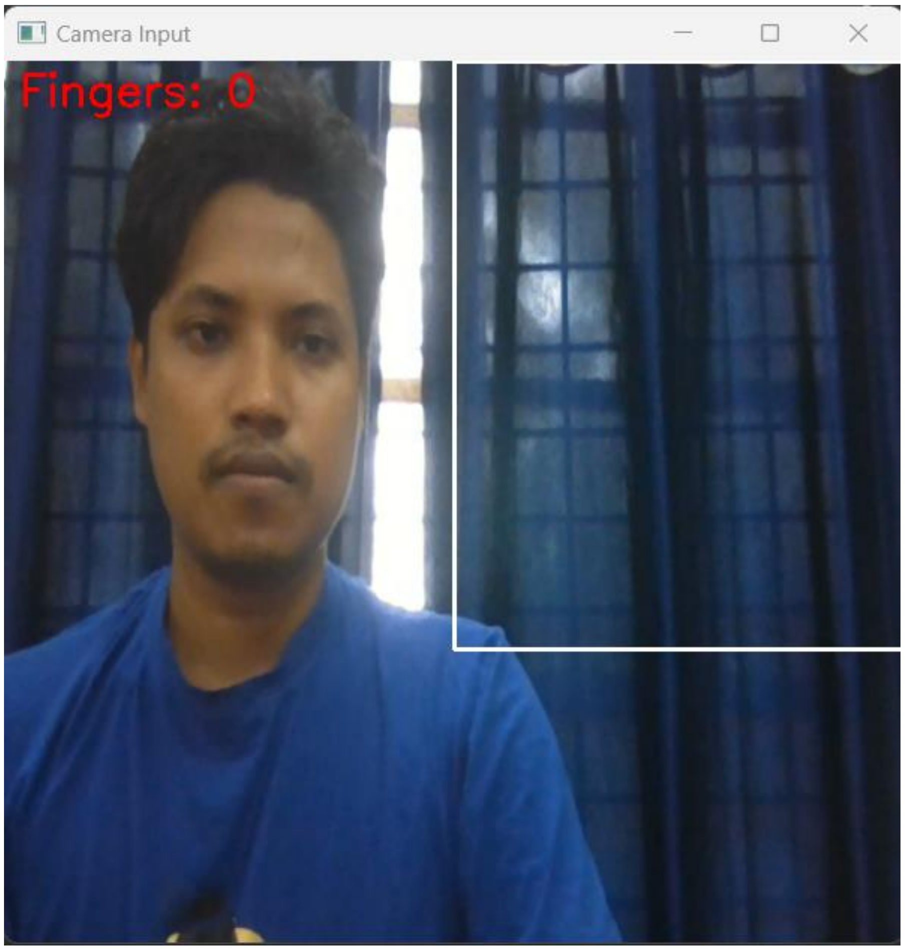

Figure 1: Initial Frame and Test Image.

## 2 Region of Interest (ROI) Definition

A specific region in the frame is designated to detect hand gestures, reducing computational load and focusing processing on the area where hand gestures are expected. `A function named` get_region(frame) function is used to extract the region from the frame and converts it to grayscale using cv2.cvtColor. Gaussian blur using cv2.GaussianBlur is used to reduce noise, aiding in more robust background subtraction.

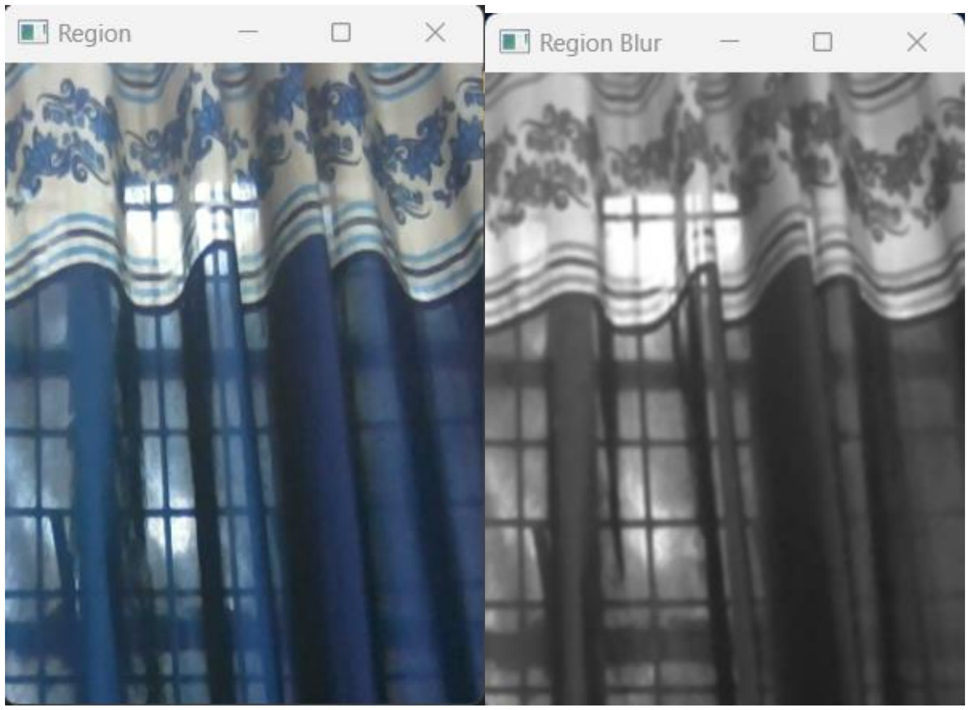

Figure 2: Region of Interest.

## 3 Background Calibration

The get_average(region_blur) function calculates the weighted average of the blurred region to build a model of the background over the calibration period (CALIBRATION_TIME). This helps in later steps to differentiate between the hand and the static background using cv2.accumulateWeighted.

## 4 Background Subtraction

The segment (region, region_blur) function computes the absolute difference between the current frame and the background. Binary thresholding using cv2.threshold function is used to create a binary image where the hand (foreground) appears white, and the background is black.

Morphological operations (cv2.morphologyEx) are applied to remove noise and close gaps, ensuring clear contour detection.

## 5 Skin Detection

### 5.1 Skin Mask Creation

The get_skin_mask(frame) function converts the frame to YCrCb and HSV color spaces. These spaces are chosen because they separate chrominance (color) from luminance (brightness), making it easier to detect skin tones. Skin color falls within specific ranges in YCrCb and HSV, which are used to create binary masks (cv2.inRange). The masks from both color spaces are combined using cv2.bitwise_or to improve robustness in different lighting conditions. Further morphological operations and median blurring are applied to refine the mask.

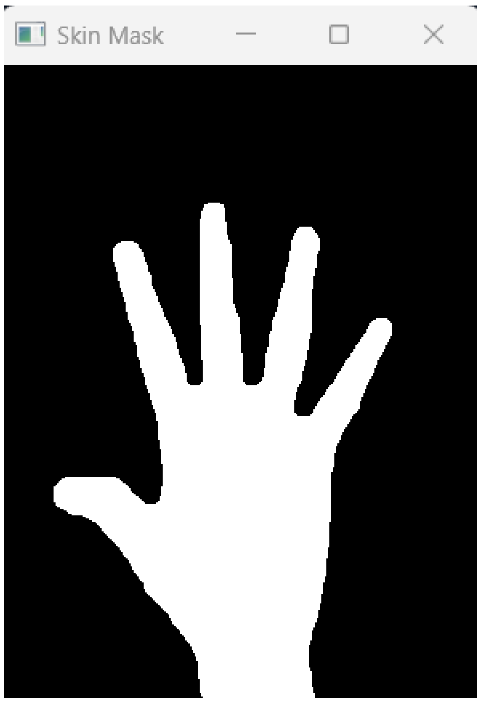

Figure 3: Skin Mask.

## 6 Combining Masks

### 6.1 Mask Combination

The combine_masks(background_mask, skin_mask) function is used a binary mask from background subtraction which is combined with the skin mask to isolate the hand more accurately. This ensures that only the detected skin regions that differ from the background are considered.

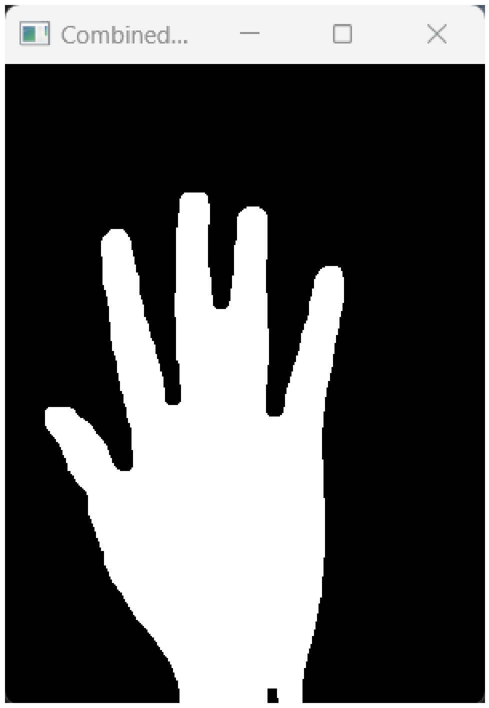

Figure 4: Combined Mask.

## 7 Contour Detection and Hand Structure Confirmation

### 7.1 Find Contours

Contours are found by using cv2.findContours() in the binary image.cv2.RETR_EXTERNAL retrieves only the outermost contours, which is ideal for detecting the outline of the hand. cv2.CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and diagonal segments, keeping only their endpoints, thus reducing memory usage.

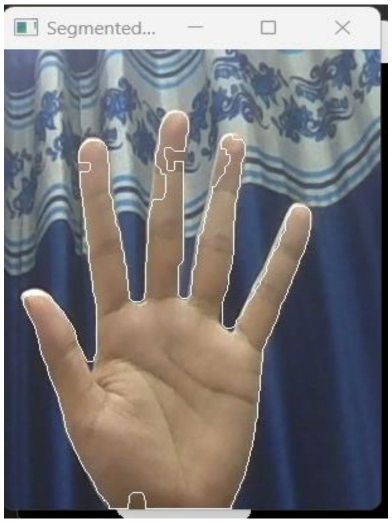

Figure 5: Contour Image.

### 7.2 Hand Structure Confirmation

The max_contour = max(contours, key=cv2.contourArea) function is used to select the maximum contour. The selected area is assumed the hand. This is based on the assumption that the hand is the most significant object in the region. The bounding rectangle and aspect ratio is calculated using cv2.boundingRect() function. The bounding rectangle around the largest contour helps in verifying the contour's validity. Aspect ratio and Extent is calculated by,

aspect_ratio = w / float(h) and extent = area / float(rect_area) …………(1)

The aspect ratio and extent (ratio of contour area to bounding rectangle area) are used to confirm the hand's structure. Acceptable ranges ensure the detected contour is likely a hand (neither too elongated nor too compact).

The cv2.drawContours(region, [hand_contour], -1, (255, 255, 255)) function is used to draw the detected hand contour on the region for visual confirmation. This step aids in debugging and ensures that the hand's outline is correctly identified.

## 8 Feature Extraction

The get_hand_data(segmented_image, hand) function extracts hand features by identifying the fingers using the convex hull of the hand contour.

### 8.1 Convex Hull Calculation

Once the hand contour is identified, the next step is to calculate the convex hull, which helps outline the outer boundary of the hand.

- **Convex Hull Calculation**: The cv2.convexHull() function is called with the hand contour as input. This function returns the convex hull points, which are stored in the hand.hull attribute.
- **Purpose of the Convex Hull**: The convex hull represents a 'tight' wrapping around the hand, effectively outlining the tips of the fingers while ignoring the concave areas between fingers. This makes it easier to distinguish fingertips from other parts of the hand.

## 8.2 Finding Convexity Defects

Convexity defects are used to identify the points between fingers (the valleys), which are essential for counting the number of extended fingers.

- **Convexity Defects Calculation**: The cv2.convexityDefects() function is called with the hand contour and its convex hull. This function identifies the points where the contour deviates inward from the convex hull, which usually occurs between fingers.
- **Defect Storage**: These defects are stored in the hand.defects attribute for further processing.

## 9 Counting Fingers

### 9.1 Counting Fingers based on defects

The core of the finger-counting logic involves analyzing these convexity defects. Each defect typically represents a valley between two raised fingers.

- **Initialization**: The finger count is initialized to zero.
- **Loop through Defects**: The code loops through each detected defect to analyze the start, end, and far points:
  - **Start and End Points**: These are the points on the contour where a convexity defect begins and ends. They usually represent the points near the base of the fingers.
  - **Far Point**: This is the deepest point of the defect, located between the start and end points. It is the lowest point in the valley between two fingers.
- **Angle Calculation**: To determine whether a defect corresponds to a valid finger separation, the angle at the far point is calculated using the cosine rule:

$$\text{angle}= \arccos \left(\frac{b^2+c^2-a^2}{2bc}\right)$$

Here:

  - a, b, and c are the distances between the start, end, and far points.

    - An angle less than 90 degrees (π/2 radians) indicates that the fingers are sufficiently separated.
- **Distance Calculation**: The distance between the start and end points is also measured to ensure the separation is significant.
- **Counting Fingers**: If both the angle and distance criteria are met, it is counted as a finger. The code increments the finger_count for each valid defect. A minimum distance threshold helps filter out noise and small movements.

### 9.2 Adjusting Finger Count

- **Base of the Hand**: The thumb or the base of the hand may not be detected as a defect, so the code adds one to the finger_count to account for it.
- **Limiting Finger Count**: The finger count is capped at five to avoid counting errors due to noise or other anomalies.

## 10 Game Play

### 10.1 Game Play Mode

The final count is then used to determine the appropriate game action:

- **One Finger**: Simulates pressing the right arrow key (pyautogui.keyDown('right')).
- **Two Fingers**: Simulates pressing the left arrow key (pyautogui.keyDown('left')).
- **Three Fingers**: Simulates pressing the enter key (pyautogui.press('enter')).
- **Other Counts**: Release all keys if the count doesn't match expected actions.

## 11 Visualization

The segmented region displays the portion of the frame where the hand is detected. It helps verify if the hand is correctly segmented from the background.

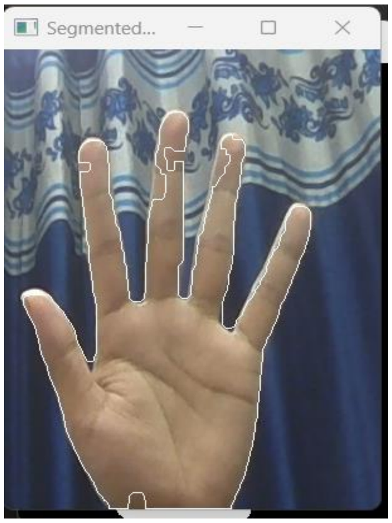

Figure 6: Segmented Image.

The skin mask shows the result of skin color detection, highlighting areas likely to contain skin. The background mask displays the binary mask of the detected hand contour.

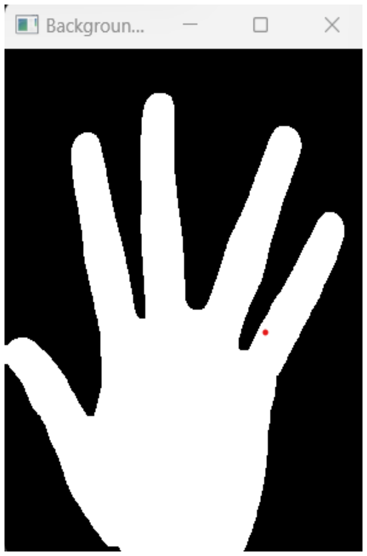

Figure 7: Background Mask.

The combined mask merges the skin and background masks to isolate the hand more accurately.

The final segmented image displays the segmented hand with contours drawn, aiding in visual confirmation of hand detection.

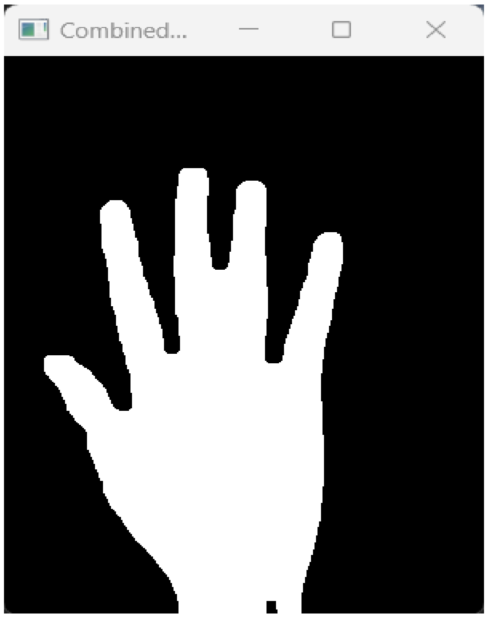

Figure 8: Combined Mask.

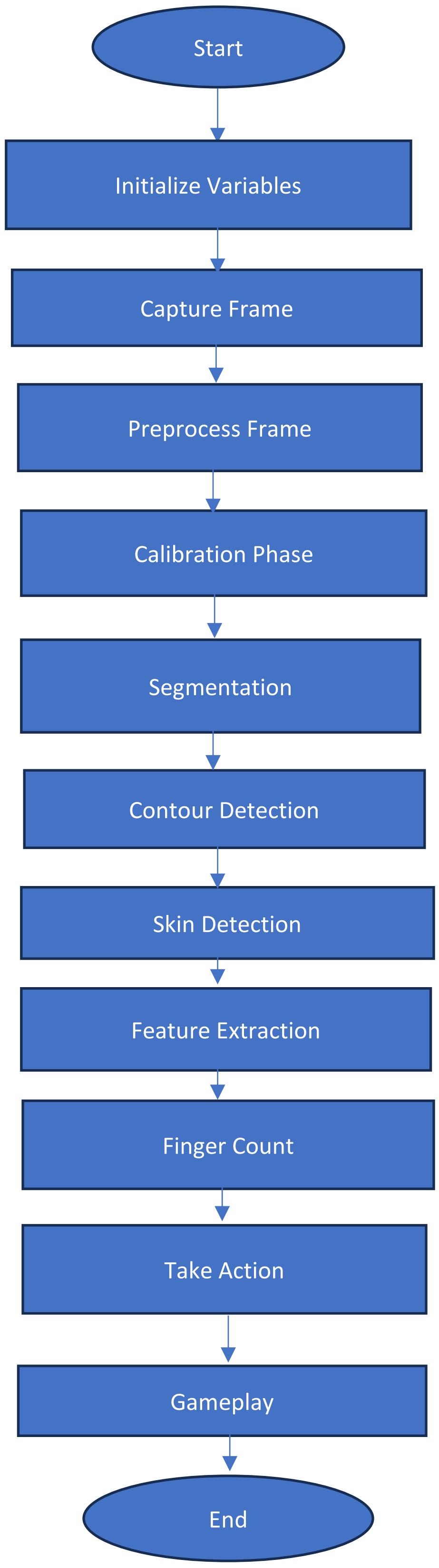

Figure 9:  Flowchart of Hill Climb Game Automation.

## Programming Environment

1. Visual Studio Code
2. Spider

## Programming Language

1. Python 3.11.3

## Key Features

1. **Real-time Gesture Recognition**: The system processes video frames in real-time to detect and interpret hand gesture.
2. **Background Calibration**: The system calibrates the background to enhance hand segmentation accuracy.
3. **Gesture Interpretation**: The system differentiates between finger counts and plays different options.

## Final Output

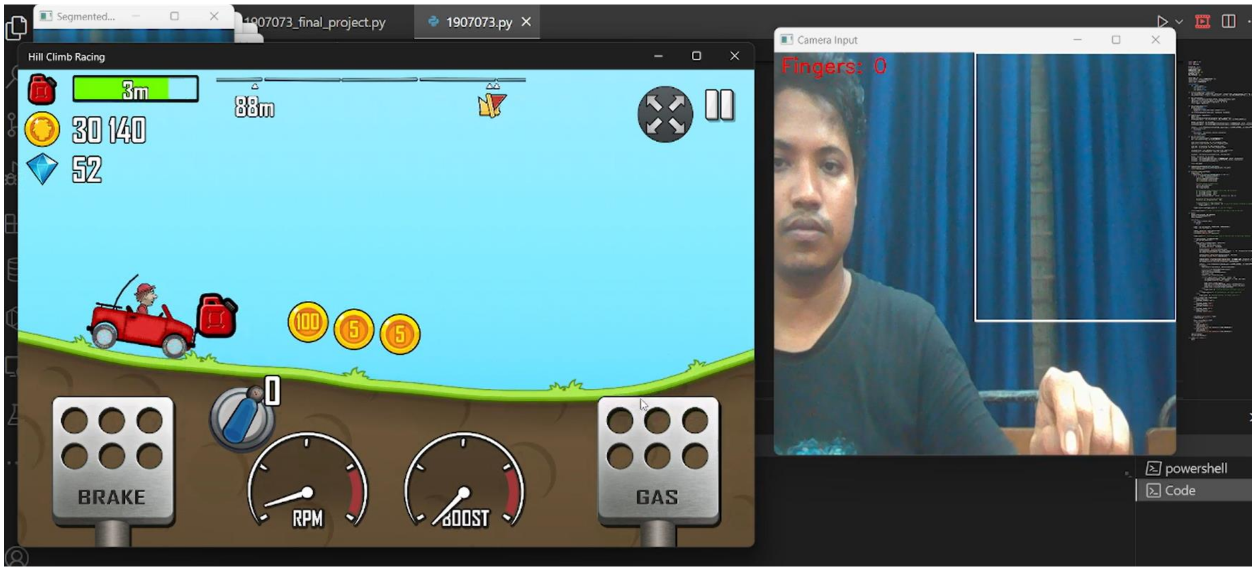

Figure 10: Initial or Neutral when the finger count is 0.

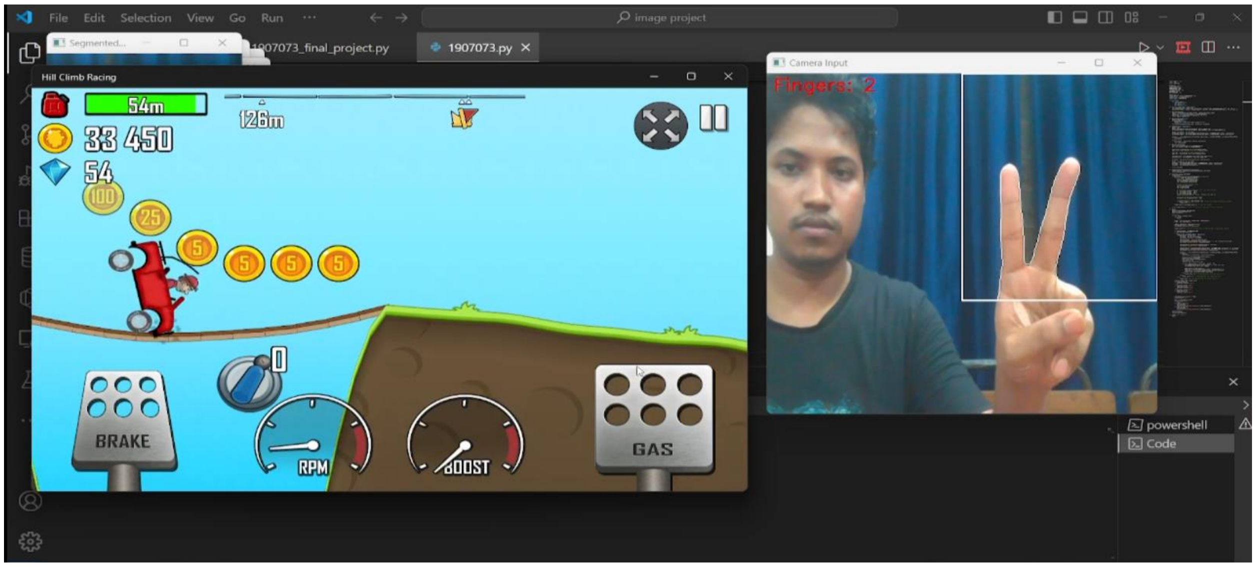

Figure 10: Pressing Break for finger count=2.

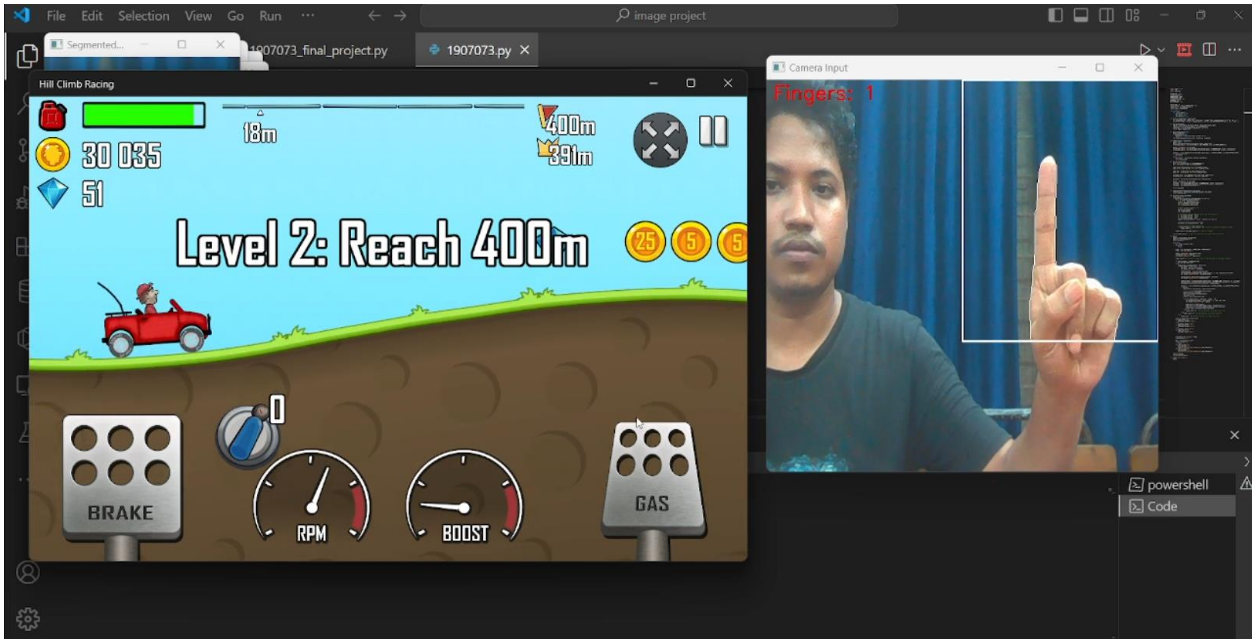

Figure 13: Pressing Gass for finger count=1.

## Limitations

Despite the effectiveness of the hand gesture recognition system for controlling *Hill Climb Racing*, several limitations are inherent to the approach:

1. **Lighting Conditions**: The performance of the hand detection and gesture recognition system heavily depends on consistent and appropriate lighting conditions. Poor lighting, shadows, or overly bright environments can affect the accuracy of hand segmentation and feature extraction, leading to incorrect gesture recognition.
2. **Background Complexity**: The system is designed to detect hands against a relatively simple and non-cluttered background. Complex or dynamic backgrounds can introduce noise and errors in hand segmentation, as the background subtraction may not perform well, leading to inaccurate hand contour detection.
3. **Skin Tone Variation**: The algorithm utilizes color-based segmentation to detect the hand, which may not work equally well for all skin tones. Extreme variations in skin color might result in reduced accuracy in hand detection, requiring additional calibration or adaptive skin tone detection mechanisms.
4. **Real-time Processing Limitations**: The system requires significant computational resources to process frames in real time, particularly when performing multiple morphological operations and convexity defect calculations. On lower-end hardware, this can result in lag, reducing the responsiveness of the gesture-based controls.
5. **Limited Gesture Set**: The system is designed to recognize a limited number of gestures (up to five fingers). Complex gestures, such as those involving specific finger movements or combinations, are not supported, limiting the versatility of the interaction.
6. **False Positives and Negatives**: Due to variations in hand positioning, size, and occlusion, the system can occasionally misinterpret gestures. False positives (detecting gestures when none are performed) or false negatives (failing to detect actual gestures) can occur, affecting the reliability of the system.
7. **User Adaptation**: Users may need to adapt their hand movements and positions to align with the system's recognition capabilities. For example, holding the hand too close to the camera or at awkward angles may result in misinterpretation of gestures.

## Discussion

The project demonstrates the feasibility and potential of using hand gestures to control a video game, specifically *Hill Climb Racing*. By employing computer vision techniques such as background subtraction, contour detection, and convex hull analysis, the system effectively translates hand gestures into game commands. The implementation showcases the integration of image processing and real-time interaction, opening up new possibilities for hands-free gaming experiences.

However, the project also highlights the challenges associated with gesture recognition systems. Factors such as lighting, background complexity, and computational requirements need to be carefully managed to ensure consistent and accurate performance. Moreover, the limitation in the number of detectable gestures restricts the system's applicability to simple control schemes, which may not be sufficient for more complex gaming scenarios.

To improve the system, future work could focus on enhancing robustness against environmental variations by incorporating adaptive algorithms and machine learning techniques. Additionally, expanding the gesture set and improving gesture differentiation could make the system more versatile and applicable to a wider range of applications beyond gaming, such as virtual reality, augmented reality, and human-computer interaction interfaces.

## Conclusion

This project successfully demonstrates a hand gesture recognition system that can control *Hill Climb Racing* using real-time computer vision techniques. By leveraging contour detection, convex hull analysis, and convexity defect identification, the system effectively counts fingers and maps these gestures to game controls. The approach provides a novel and intuitive way of interacting with a video game, offering a hands-free experience that could enhance user engagement.

While the system has limitations, such as sensitivity to lighting conditions and a limited set of recognizable gestures, it lays a solid foundation for further development in gesture-based control systems. Addressing these limitations and expanding the capabilities of the system could lead to broader applications, including enhanced gaming experiences and innovative interaction methods in various fields.

Overall, the project illustrates the potential of gesture recognition as a natural interface for human-computer interaction and paves the way for future advancements in this exciting area of research and development.

## References

**[1]** Oudah, Munir, Ali Al-Naji, and Javaan Chahl. "Hand gesture recognition based on computer vision: a review of techniques." *journal of Imaging* 6.8 (2020): 73.

[2] Sulyman, ABD Albary, et al. "REAL-TIME NUMERICAL 0-5 COUNTING BASED ON HAND-FINGER GESTURES RECOGNITION." *Journal of Theoretical & Applied Information Technology* 95.13 (2017).