CS – GY 6643

PROJECT: SNAKE GAME

TEAM:

HARSH JALUTHARIA – *hj2607*

MUDIT NIGAM – *mn3439*

RHUTHVIK DENDUKURI – *rd3377*

Table of Contents

Introduction

Background

The original Snake game, popularized by Nokia's monochrome mobile phones in the late 1990s, is a classic arcade-style video game. The objective is to control a continuously growing snake by guiding it to eat food items while avoiding collisions with its own body or the boundaries. As the snake grows longer with each piece of food consumed, the challenge lies in navigating through an increasingly crowded and confined space.

The game's simplicity and portability contributed to its enduring popularity, making it an ideal testbed for computer vision projects focused on tracking movement, detecting collisions, and developing advanced features like obstacle avoidance or path planning algorithms.

Motivation

A great chance to create and test computer vision algorithms in a controlled setting is the Snake game. It presents various non-trivial issues for computer vision systems despite having straightforward gameplay mechanisms. Robust vision algorithms are needed to identify objects, follow the snake's movement accurately,

and detect impacts with objects or the snake's own body. Furthermore, the continuous nature of the game and its rising complexity as the snake lengthens can be utilized to assess how well vision techniques operate and how scalable they are under different circumstances. Starting with this well-defined and well-known game allows the focus to be on creating and applying unique computer vision techniques without being distracted by intricate deep learning models.

Objectives

- ## Implement a Snake Game

  The primary objective is to develop the core mechanics of the snake game, encompassing essential elements such as snake movement, food and obstacle generation and a scoring system. By recreating the classic gameplay, we aim to provide a familiar yet enjoyable experience for players.

- ## Integrate Computer Vision

  Leveraging computer vision algorithms, we seek to detect objects within the game environment and enable players to control the snake’s movements using real-world objects captured by the system’s web camera. This integration adds a unique layer of interactivity to the game, allowing players to engage with the virtual world in innovative ways.

- ## Enhance Gameplay Experience

  The goal is to create an engaging and intuitive gameplay experience by seamlessly blending computer vision technology with the classic game mechanics.

- ## Scalability and Performance

  It is crucial to design the system to be scalable and efficient, capable of running in real-time in standard hardware while delivering a smooth experience. By optimizing algorithms and leveraging hardware acceleration where possible, we aim to ensure that the game performs reliably across a range of devices and environments.

Project Overview

Description

This project reimagines the Snake Game by integrating computer vision. Players control the snake, collecting food and avoiding obstacles. What sets it apart is the ability to control the snake using real-world objects detected through a camera feed. Computer vision algorithms analyze the feed in real-time, translating object positions into commands for the snake's movement. This immersive gameplay experience blends reality with virtuality, offering dynamic interaction. Whether navigating household items or custom obstacles, players enjoy a fusion of nostalgia and innovation.

Technologies Used

1. Python
2. OpenCV
3. Numpy
4. Tkinter
5. Random Module

CV techniques Used

Color – based Segmentation

This involves segmenting the camera feed based on color to isolate the objects of interest from the scene. This is typically done using color thresholding techniques, where pixels within a certain color range are considered part of the object, while the others are discarded. In our scenario, this process is implemented using the range from the color codes “(29, 86, 18)” (lowest green HSV code) to “(93, 255, 255)” (highest green HSV code).

Preprocessing

Before further analysis, the segmented image may undergo preprocessing steps like erosion and dilation, which are further discussed, to enhance the object boundaries and remove noise. These operations helps refine the object masks and improve accuracy of subsequent processing steps.

Contour detection

Once the preprocessed image is computed, contours are detected using “cv2.find-Contours()” function. Contours represent the boundaries of connected regions within the segmented image and are essential for identifying and analyzing ob-jects.

The input parameters of this function take an image, contour retrieval mode and contour approximation method. For our necessity, the contour retrieval mode is set to cv2.RETR_EXTERNAL which retrieves only the external contours (outer boundary of the object) and the contour approximation method is set to

cv2_CHAIN_APPROX_SIMPLE which compresses horizontal, vertical and diagonal segments leaving only their endpoints.

Object Localization and Identification

Object localization and identification are critical components that enable precise interaction between the virtual snake and real-world items. Through meticulous contour analysis, the system extracts vital geometric information from observed objects, including their sizes, shapes, and spatial orientations. This analysis ensures the accurate localization and tracking of objects within the gaming environment, facilitating realistic interaction with the virtual snake.

Leveraging the derived contour information, the game dynamically responds to the player's actions, offering seamless and captivating gameplay experiences. By applying additional criteria to filter out extraneous contours and identify objects of interest, the system distinguishes between valid objects and noise or background elements. This process enhances gameplay dynamics by ensuring accurate interaction between the virtual snake and detected objects, delivering a more immersive and responsive gaming experience for players.

Contour detection here is used to detect a circular object in the image. Another way to find a circle in the image is by using Hough Transform which was explored but it had several complexities.

Using Hough Transform with edge orientation was a challenge we faced in this project. This method involves detecting edges and computing edge orientations before applying hough transform. This is a complex computational approach as the frames give a continuous input to the system to which hough transform has to be applied and the result has to be provided within no time. The accuracy of edge detection affects the performance of hough transform.

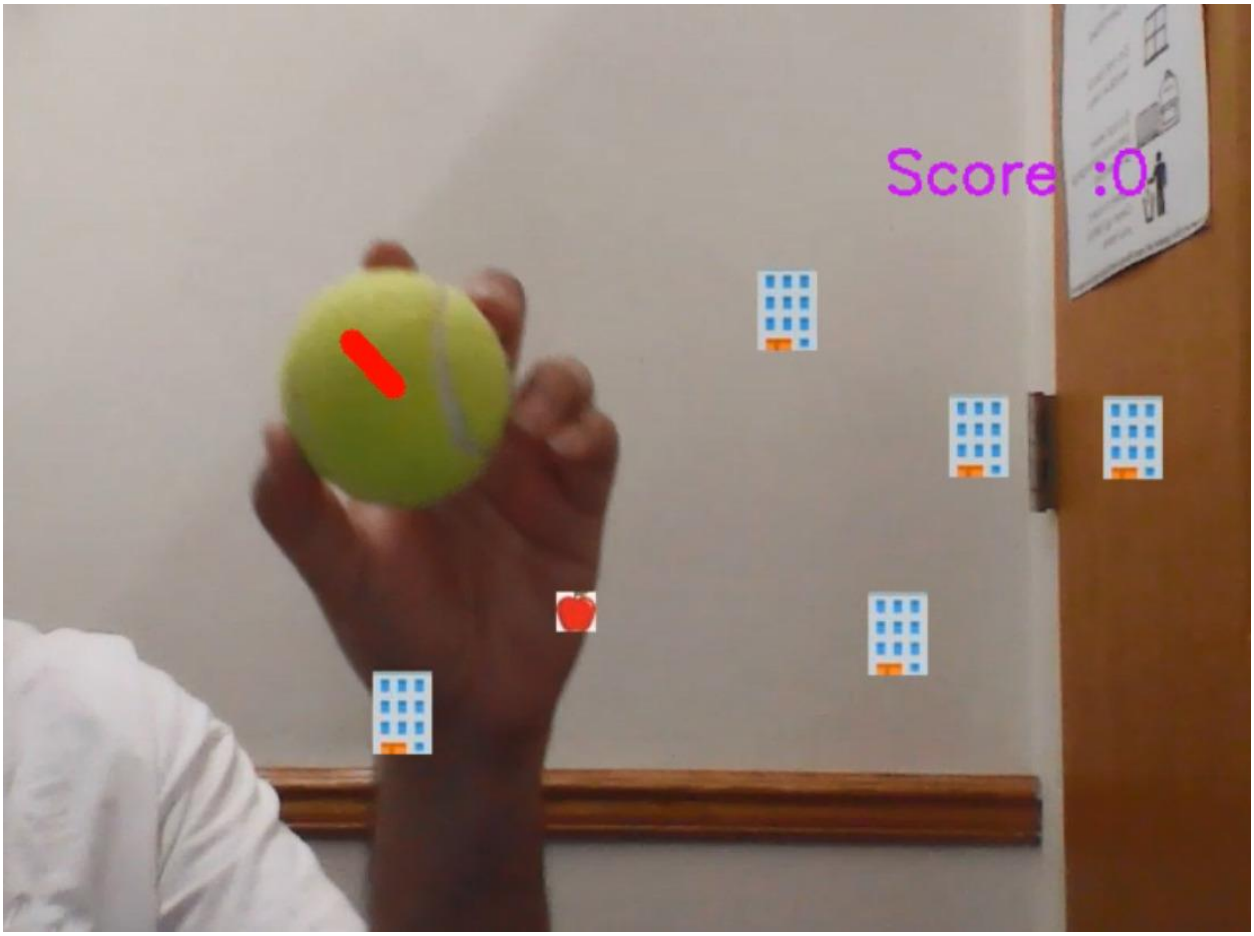
In this scenario, the factors like light conditions, object textures and noise levels play a strong role in quality of edge detection. In addition, edge detection is sensitive to noise in the image. Though a gaussian blur is used to smoothen the image and lessen the noise, the computed edges were not sufficiently clear for the hough transform to perform with accuracy. Also, performing edge detection and

orientation estimation for every frame in such a real – time scenario imposes significant computational overhead.

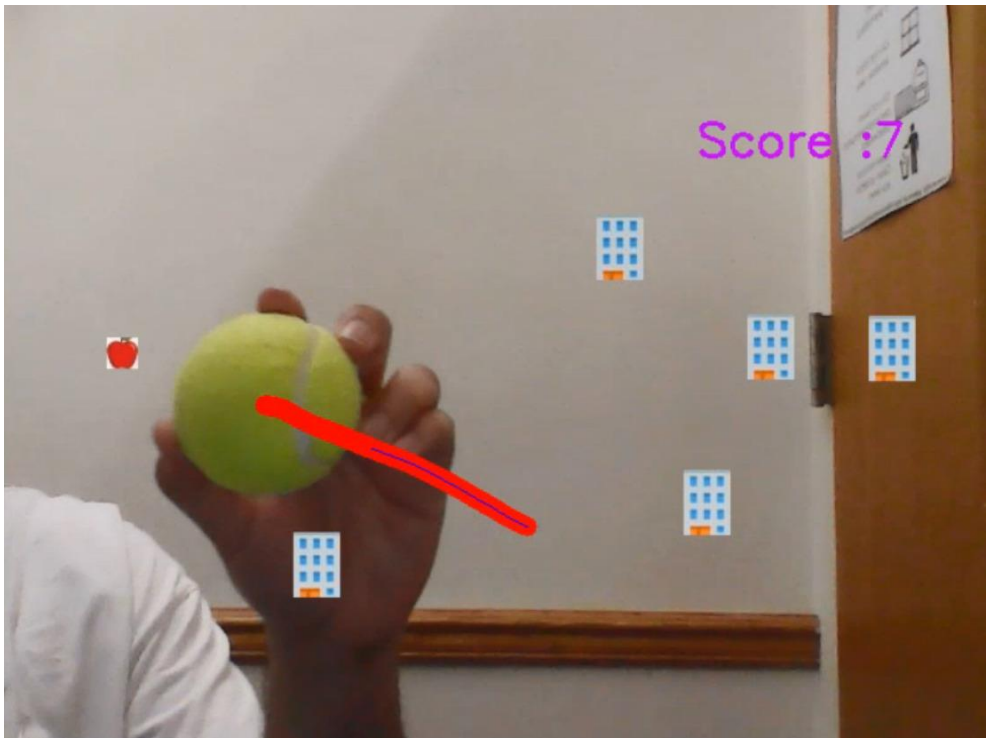
In conclusion, since such edge detection and orientation need algorithms with strict time constraints, “cv2.findContours” performed both accurately and faster.

Game and General Implementations

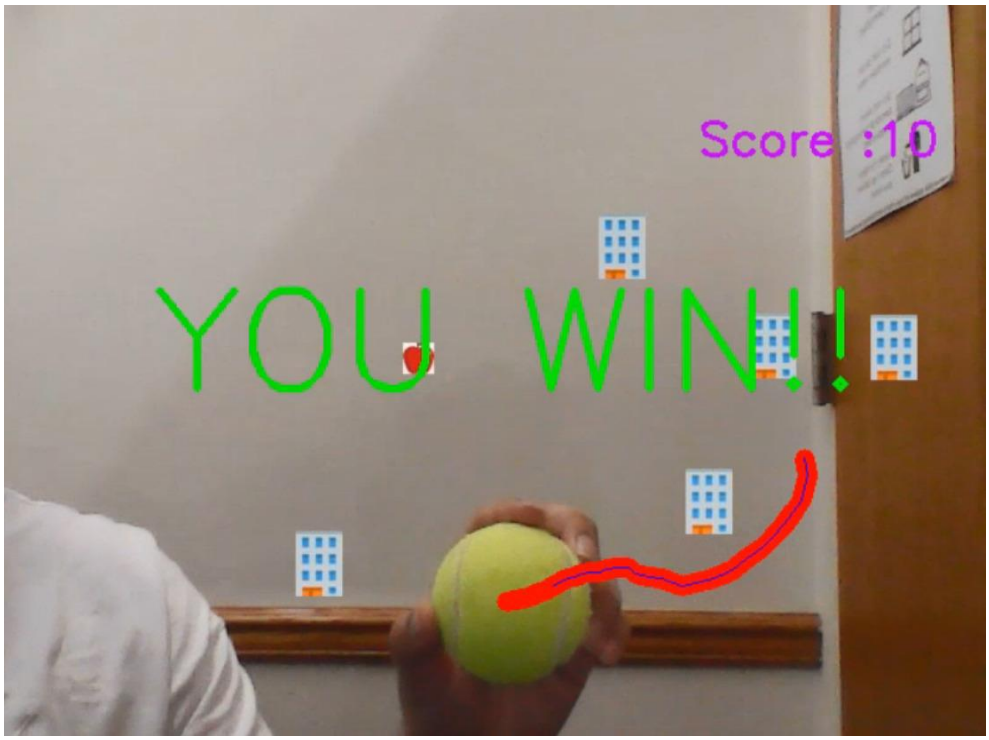
Initially, the snake is small. The obstacles are generated randomly at start of game. Food location is also randomized.



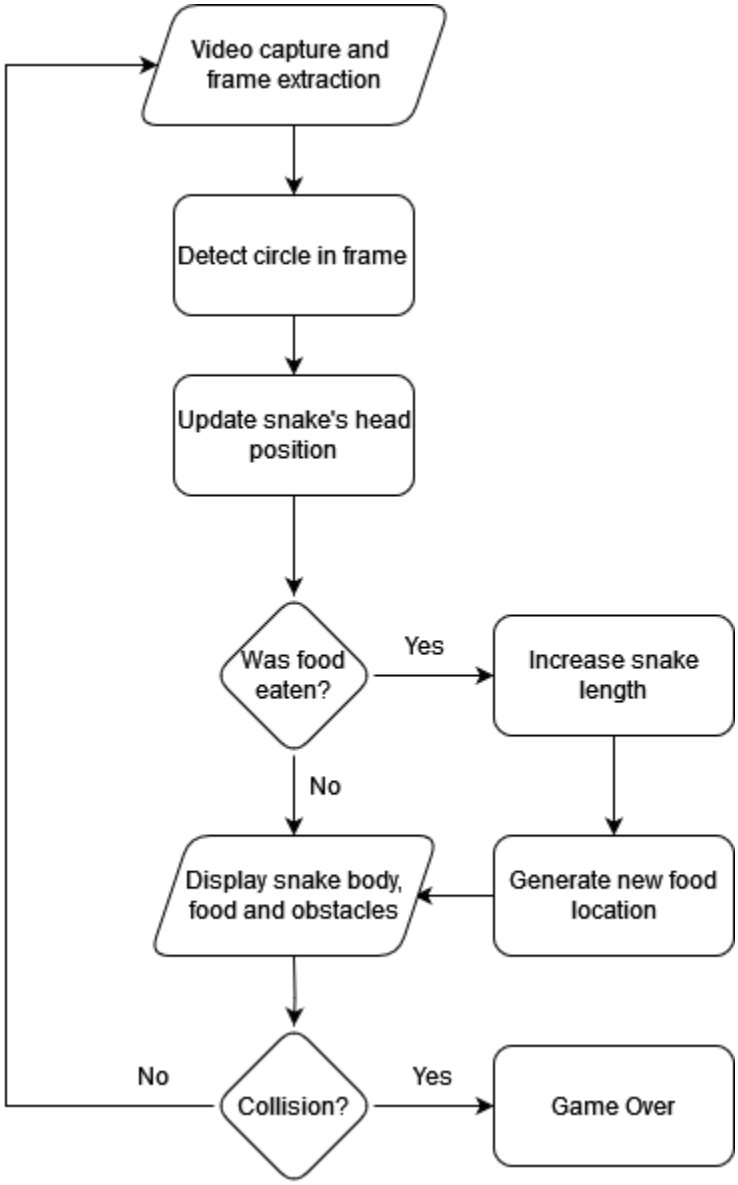
When score is 7, snake is considerably longer.



When score reaches 10, game is won. The following screen is displayed:



- Game Loop



  - Video capture and frame extraction: The game loop starts by capturing a frame from the webcam using “cv2.VideoCapture”. This step is responsible for getting the live video feed from the camera.
  - Detect circle in frame: The captured frame is processed to detect a green object (presumably a green ball or marker) using color thresholding and contour detection techniques from OpenCV. The largest

contour is assumed to be the green object, and Welzl's algorithm is applied to find the minimum enclosing circle around the contour.

- Update snake's head position: If the radius of the minimum enclosing circle is greater than a certain threshold, the center of the circle is considered as the new head position of the snake. The “updateSnake” function is called with this new head position to update the snake's body.
- Was food eaten?: Inside the c function, the code checks if the new head position overlaps with the food location (“self.foodLocation”). If the food was eaten, the game proceeds to the "Increase snake length" step.
- Increase snake length: If the food was eaten, the “self.totalLength” (maximum allowed length of the snake) is increased by “LENGTH_INCREASE_FROM_FOOD” (a constant value), effectively increasing the snake's length.
- Generate new food location: After increasing the snake's length due to eating the food, a new random food location is generated using the “generateRandomFoodLocation” method. This method ensures that the new food location is at least “MIN_DISTANCE_FOOD_OBSTACLE” distance away from all obstacle locations.
- Display snake body, food, and obstacles: The game frame is updated by calling the “drawObjects” method, which draws the snake's body, the food item, and the obstacles on the frame.
- Collision?: Inside the “updateSnake” function, the code checks for two types of collisions: collision between the snake's head and obstacles, and collision between the snake's head and its own body. If a collision is detected, the game proceeds to the "Game Over" step.
- Game Over: If a collision is detected, the “gameFinished” function is called with “win=False”, indicating a game over condition. This function displays the "GAME OVER!!" message on the screen and prompts the user to play again or exit the application. If there is no collision, the game continues by capturing the video and extracting the frame to the game to run.

The loop continues until the user decides to exit the application or the game ends due to a collision or other conditions (e.g., reaching a win score). The provided code implements various helper functions and classes to handle different aspects of the game, such as obstacle generation, collision detection, and updating the snake's position and length.

- ## UI Design

The UI design of the game is implemented using Tkinter, a python library to create graphical user interfaces.

- ### Camera Feed Display

  The UI prominently features a display area showing the real-time camera feed captured by the device's camera. This feed serves as the backdrop for the game environment, providing visual context for the player's interactions.

- ### Snake Representation

  Within the camera feed display, the snake is represented by a continuous line segment or series of connected points, denoting its current position and trajectory. The snake's appearance may vary in color or thickness to enhance visibility and distinguish it from other elements in the environment.

- ### Food and Obstacle Visualization

  Food items and obstacles are visually represented within the camera feed display. Food items appear as distinct objects or icons that the snake must consume to grow and earn points. Obstacles are depicted

as static elements or barriers that the snake must navigate around to avoid collision.

- Score Display

A dedicated area of the UI is allocated for displaying the player's current score. This numerical value updates dynamically as the player successfully consumes food items and earns points throughout the gameplay session.

- Game Over/Win Condition Message

Tkinter's message box or label widgets are utilized to display relevant messages on the UI in the event of game over or achievement of win conditions. Tkinter's message box module may be employed to prompt the player with options to restart the game or exit the application.

- User Interface Prompts

Interactive buttons or widgets created using Tkinter are incorporated into the UI to facilitate user interaction. Tkinter's button widget can be configured to trigger actions such as restarting the game or exiting the application in response to player input.

- Visual Feedback and Animations

The UI may include visual feedback elements or animations to enhance the user's engagement. Tkinter's canvas widget supports animation capabilities, allowing for the creation of dynamic feedback elements.

- ## Snake Movement Mechanism

The snake's movement is controlled by tracking a green object (presumably a green ball or marker) using color thresholding and contour detection techniques from the OpenCV library. These are discussed later.

Every time the "updateSnake" function is called, a new position of the head is passed ("NewHeadLocation"). This new position is stored in the "self.points" array. The length of the newly added segment (distance between "NewHeadLocation" and "self.previousHead") is calculated and stored in the "self.lengths" array. If the total length of all these lengths exceeds the maximum allowed length ("self.totalLength"), the earliest added points and lengths are removed until the snake's length is less than the maximum length. This process maintains a smooth movement of the snake's body while ensuring that its length does not exceed the specified maximum length.

- ## Food and Obstacle Generation

  - ### Food Generation (Apple)

    - The "generateRandomFoodLocation" method is called to generate a new food location whenever the snake eats the existing food item.
    - The method generates random x and y coordinates within the game screen boundaries (between 50 and "VIDEO_WIDTH-50" for x, and between 50 and "VIDEO_HEIGHT-50" for y). This ensures that the food item is not placed too close to the edges of the screen.
    - The method then checks if the randomly generated food location is at least "MIN_DISTANCE_FOOD_OBSTACLE" distance

away from all the existing obstacle locations ("self.obstacleLocations"). This distance is set to 60 pixels by default.

- If the generated food location is too close to any obstacle, the method discards that location and generates a new random location.
- This process continues until a suitable location is found that satisfies the minimum distance constraint from all obstacles.
- Once a valid location is found, it is stored in "self.foodLocation", and the corresponding food image ("self.foodImage") is drawn at that location using the "drawImageOverFrame" function.
- The "drawImageOverFrame" function takes the game frame, the food image, and the location as input, and it draws the food image centered at the specified location by overlaying it on the frame.

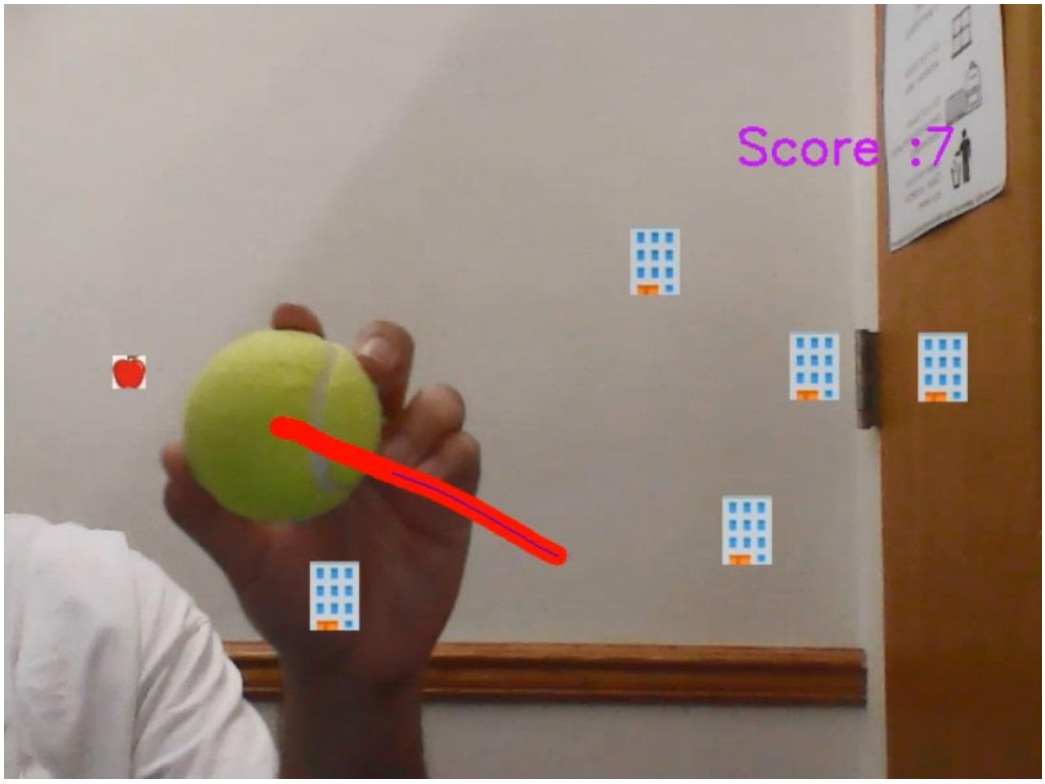
Obstacle Generation (Building)

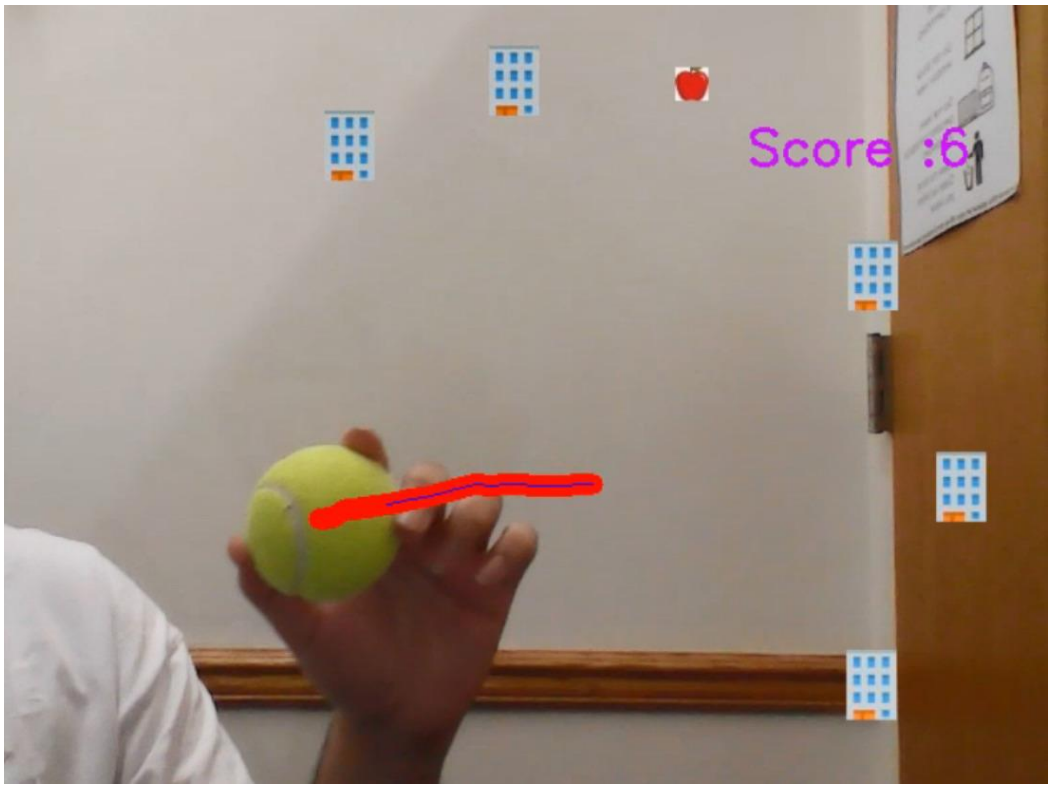
- The "generateObstacles" method is called to generate the obstacle locations during the game initialization.
- The method first initializes an empty list "self.obstacleLocations" to store the obstacle locations.
- It then enters a loop that continues until the number of obstacles in "self.obstacleLocations" reaches "self.obstacleCount", which is set to "MAX_OBSTACLES" (5 by default).
- Inside the loop, a random location ("temp_point") is generated within the game screen boundaries (between 50 and "VIDEO_WIDTH-50" for x, and between 50 and "VIDEO_HEIGHT-50" for y). This ensures that the obstacles are not placed too close to the edges of the screen.
- The method then checks if the randomly generated "temp_point" is at least "MIN_DISTANCE_BW_OBSTACLES" distance away from all the existing obstacle locations in "self.obstacleLocations". This distance is set to 60 pixels by default.

- If the generated “temp_point” is too close to any existing obstacle, the method discards that location and generates a new random location.
- This process continues until a valid location is found that satisfies the minimum distance constraint from all other obstacles.
- Once a valid location is found, it is added to “self.obstacleLocations”.
- After the loop completes, “self.obstacleLocations” contains the locations of all the obstacles.
- The obstacles are drawn on the game screen using the “drawObjects” method, which iterates over “self.obstacleLocations” and draws the obstacle image (“self.obstacleImage”) at each location using the “drawImageOverFrame” function, similar to how the food image is drawn.

The minimum distance constraints (“MIN_DISTANCE_FOOD_OBSTACLE” and “MIN_DISTANCE_BW_OBSTACLES”) ensure that the food item and obstacles are not placed too close to each other or to the edges of the screen, providing a fair and playable game environment.

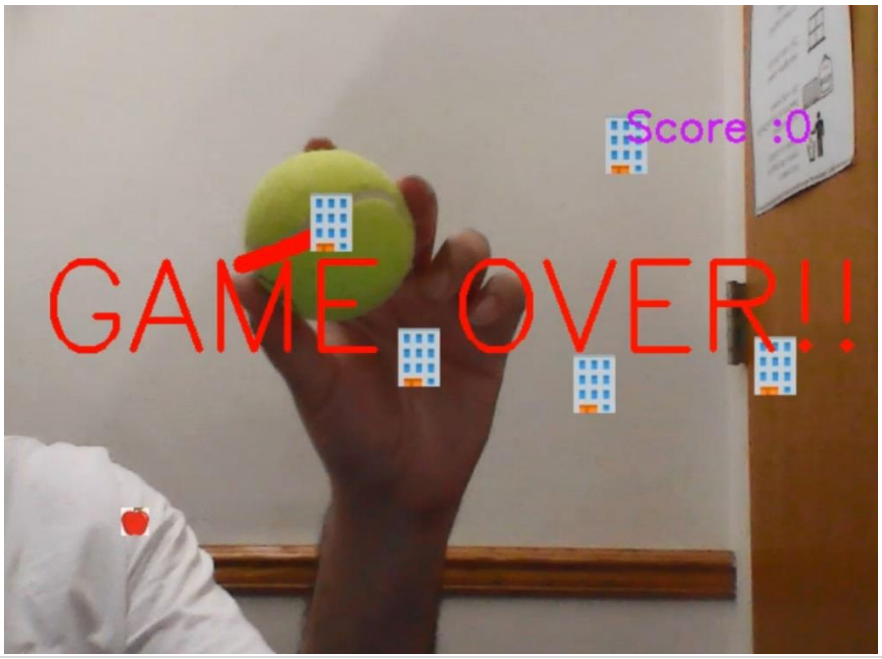
The following 2 screenshots are from different runs of the game and show the obstacles and food being generated randomly.





- Collision Detection
    - Collision with Obstacles

        - For each obstacle location in “self.obstacleLocations”, the method calls the “checkPointOverlapImage” function, passing the new head location (“NewHeadLocation”), the obstacle location, and the obstacle image size (“self.obstacleImageSize”).
        - The “checkPointOverlapImage” function calculates the bounding box around the obstacle image based on its location and size. It checks if the new head location falls within this bounding box.
        - If the new head location overlaps with any obstacle's bounding box, it means a collision has occurred.
        - In case of a collision, the “gameFinished” method is called with “win=False”, indicating a game over condition.
        - The snake's body is not a perfect circle or rectangle, making it challenging to detect collisions accurately. We had to use the “cv2.polylines” and “cv2.pointPolygonTest” functions to handle this irregularity.

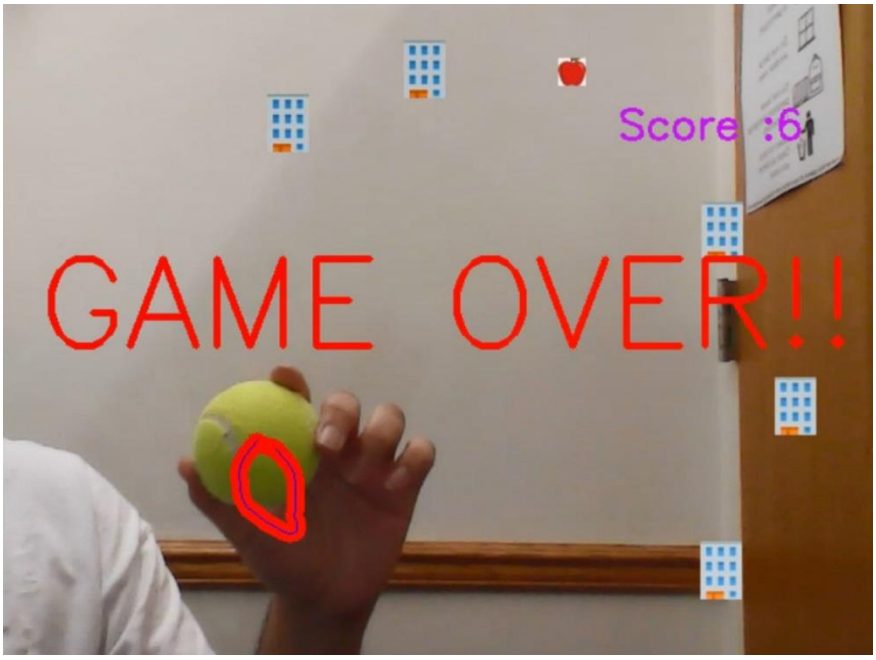


- Collision with Snake's Body

    - The method creates a numpy array "check_pts" containing all the points of the snake's body except the last few points (excluding the head and a few segments near the head).
    - The "cv2.polylines" function is used to draw these points as a polygon on the game frame.
    - The "cv2.pointPolygonTest" function is then used to calculate the minimum distance between the new head location ("NewHeadLocation") and the polygon formed by the snake's body points.
    - If the minimum distance is between -1 and 1 (inclusive), it means the new head location is within or on the boundary of the polygon, indicating a collision with the snake's body.
    - In case of a collision, the "gameFinished" method is called with "win=False", indicating a game over condition.
    - Checking collisions between the snake's head and every point in its body can be computationally expensive, especially as the snake grows longer. We had to optimize the process by excluding the few points near the head from the collision check, as they are unlikely to collide with the head.

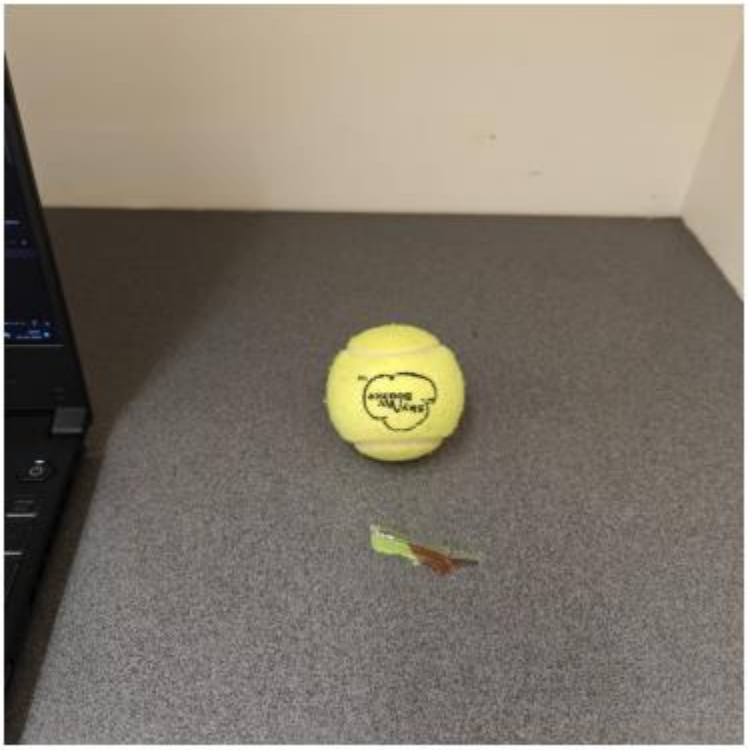
There were edge cases to consider, such as when the snake's head is exactly on the boundary of its body or an obstacle. We had to handle these cases carefully to ensure accurate collision detection. Collision detection had to be integrated with the computer vision components of the game, which added complexity. The new head location was obtained from the minimum enclosing circle around the green object's contour, and this had to be correctly used for collision checks.

Implementing collision detection required careful consideration of geometric calculations, edge cases, performance optimization, and integration with the computer vision components of the game.


Score :6
GAME OVER!!

Computer Vision Technique Implementations

Various computer vision techniques were used to perform circular object detection in the extracted frames. The following image will be used to show the effect of these processes:



- ## Blur

  Gaussian blur is a common image processing technique used to reduce noise and smooth out images by averaging the pixel values in a local neighborhood around each pixel. In this project, the blur operation is applied using a Gaussian kernel with a size of 11x11 pixels and a standard deviation of 2.

  By blurring the image, high-frequency noise is attenuated, and small-scale details are smoothed out, resulting in a more uniform appearance. This can be beneficial for subsequent processing steps, such as edge detection or object recognition, as it helps to enhance the clarity of important features

while reducing the impact of irrelevant or distracting elements in the image. Overall, Gaussian blur is a versatile tool commonly used in various computer vision applications to improve the quality and interpretability of images.

In our computer vision-based snake game, we utilized the highly optimized cv2.GaussianBlur function from the OpenCV library instead of implementing a manual Gaussian blur computation. This decision was driven by the need for efficient real-time performance when processing continuous video frames from the webcam.

Manually implementing Gaussian blur would have a time complexity of O(mn), where m and n are the dimensions of the input image, making it computationally expensive for real-time applications. In contrast, the cv2.GaussianBlur function leverages separable filtering and integral image techniques, reducing the time complexity to O(n), where n is the total number of pixels.

Additionally, OpenCV is optimized for hardware acceleration, multi-threading, and SIMD instructions, further improving the performance of image processing operations like Gaussian blurring. By leveraging these optimizations, cv2.GaussianBlur can process video frames efficiently, ensuring smooth and responsive gameplay.

Implementing Gaussian blurring manually would have introduced significant computational overhead and potential latency, degrading the overall user experience. Utilizing the optimized OpenCV function minimized the computational burden while still benefiting from the noise reduction and smoothing effects of Gaussian blurring, enabling real-time performance in our continuous video processing scenario.

Applying gaussian blur to the image mentioned earlier produces the following result:

- Mask

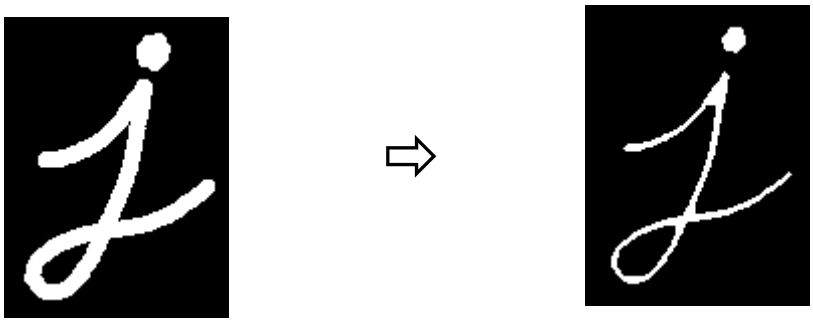
The mask is a binary image derived from an image using thresholding and logical operations. It isolates regions in the image where the pixel values correspond to the specified green color range defined by GREEN_LOWER_THRESHOLD and GREEN_UPPER_THRESHOLD. White pixels in the mask indicate the presence of green colors within the specified range, while black pixels represent non-green areas. This mask is crucial for identifying and segmenting objects or features of interest in the image, facilitating subsequent processing steps such as object detection or tracking.

Applying the green mask to the blurred image above produces the following binary image:

Note how there is still some noise being detected on the left side of the image.

- Erode

The function erode performs erosion on a binary mask. Erosion is a morphological operation that shrinks the boundaries of foreground objects in an image. It is commonly used to remove small objects, smooth contours, or separate overlapping objects.



Here's how the function works:

- Inputs:
  - mask: A binary image where foreground objects are represented by white pixels (255) and background by black pixels (0).
  - kernel_size: A tuple specifying the size of the kernel used for erosion. Kernel of odd size (3, 5, 7). In this project, it's a 3x3 square kernel.
  - iterations: The number of times erosion is applied. In this project, we have set it to 1.

- Initialization:
  - The function initializes a square structuring element (kernel) with the specified size using NumPy's np.ones() function. The structuring element is a binary matrix that defines the neighborhood used for the erosion operation.

- Erosion Operation:
    - The erosion operation is applied iterations times to the input mask. For each iteration, the function performs the following steps:
        - It loops through each pixel in the mask.
        - At each pixel, it computes the minimum value of the neighborhood defined by the kernel.
        - The minimum value represents the erosion effect, where the center pixel of the kernel is set to the minimum value of the pixels within its neighbourhood.
        - This process effectively shrinks the boundaries of the foreground objects in the mask.

- Return Value:
    - After applying erosion for the specified number of iterations, the function returns the resulting eroded mask.

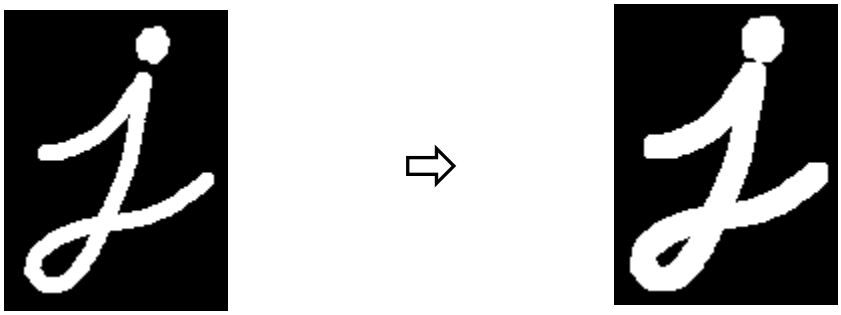
Overall, the “erode()” function provides a way to perform erosion on binary masks, which can be useful for various image processing tasks, such as noise reduction, segmentation, and shape analysis. Adjusting the kernel size and number of iterations allows for control over the erosion effect and its extent.

Eroding the binary ball image, we get the following image:

The noise has been removed but the white regions have also shrunk in size.

- Dilate

The function dilate(mask, iterations=2) is the sister function of erosion and performs dilation on a binary mask. Dilation is a morphological operation that expands the boundaries of foreground objects in an image. It is commonly used to fill gaps, join fragmented objects, or increase the size of objects.



Here's how the function works:

- Inputs:
    - mask: A binary image where foreground objects are represented by white pixels (255) and background by black pixels (0).
    - iterations: The number of times dilation is applied. In this project, we have set it to 2.

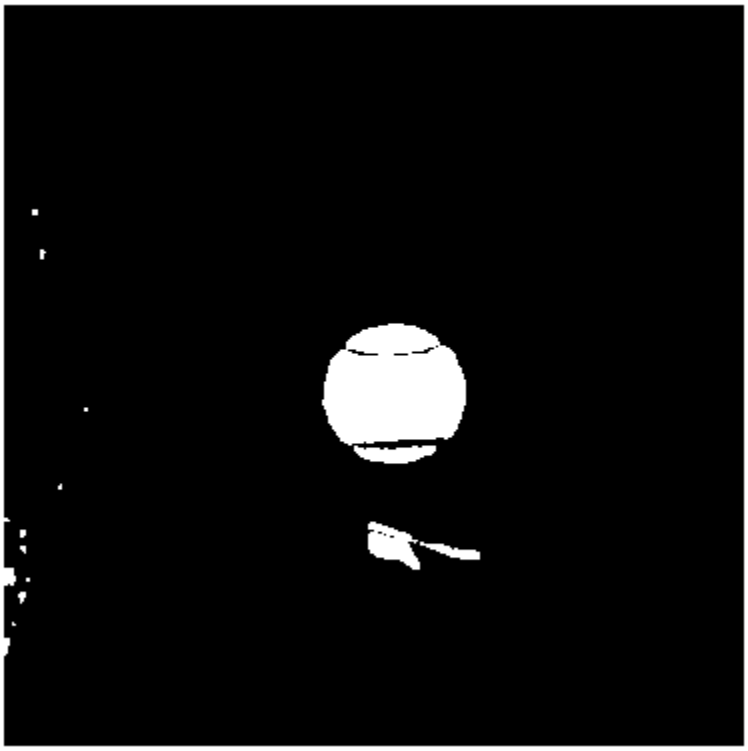
- Dilation Operation:

  The dilation operation is applied iterations times to the input mask. For each iteration, the function performs the following steps:
    - It creates a copy of the input mask called dilated_mask.
    - It loops through each pixel in the mask.
    - At each pixel, it computes the maximum value of the neighborhood defined by a structuring element.
    - The maximum value represents the dilation effect, where the center pixel of the structuring element is set to the maximum value of the pixels within its neighborhood.
    - This process effectively expands the boundaries of the foreground objects in the mask.

- Return Value:
    - After applying dilation for the specified number of iterations, the function returns the resulting dilated mask.

Overall, the dilate() function provides a way to perform dilation on binary masks, which can be useful for various image processing tasks, such as filling gaps, connecting components, and enhancing features. Adjusting the number of iterations allows for control over the dilation effect and its extent.

Applying dilate function on the eroded image above, we get the following result:

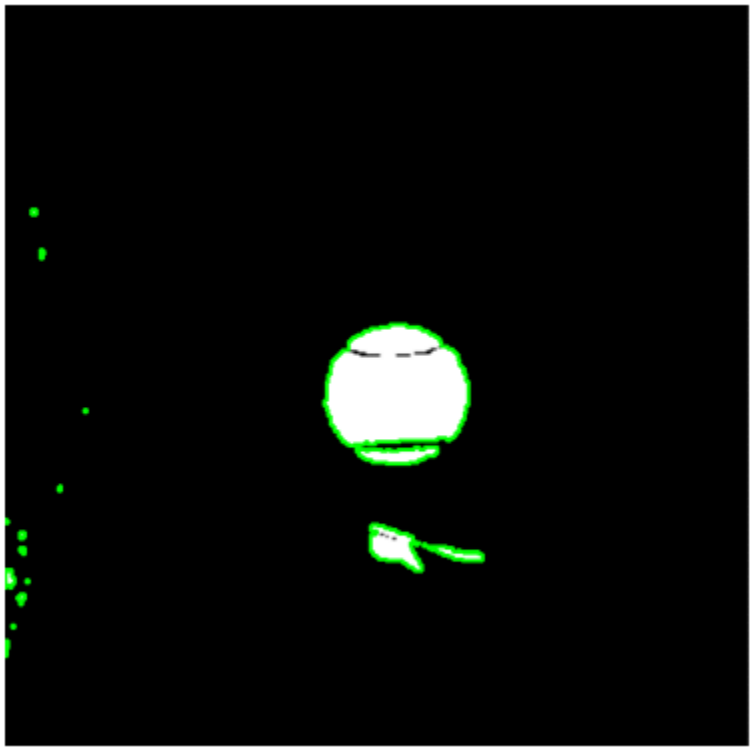


- ## Find Contour

  “findContours” is used to identify regions of interest within an image. First, the function detects contours in a binary image mask, which typically results from thresholding or other segmentation techniques. After finding the contours, the function to find line contours is executed. This step is necessary for compatibility and consistency across different versions of OpenCV. The grab_contours function reshapes the contours into a standardized format suitable for further processing. This ensures that the contours can be easily manipulated and analyzed in subsequent steps of the image processing pipeline.

  Overall, these lines of code play a crucial role in identifying and extracting regions of interest from the input image, facilitating tasks such as object detection, shape analysis, and image segmentation in various computer vision applications.

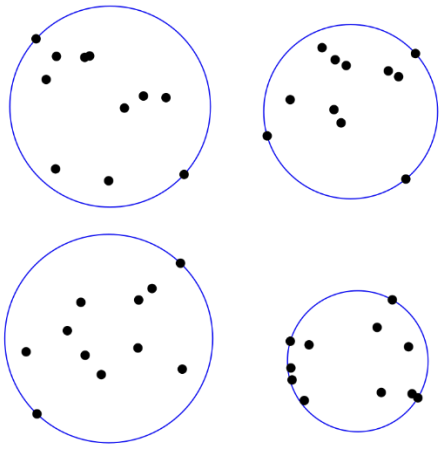
Applying the “findcontours” algorithm on the final binary image, we get the following contours marked in green:



The ball has been divided into 2 different contours. But the circle detection pipeline is robust enough to handle such inconsistencies. The biggest contour is selected as the ball and the Minimum Enclosing Circle for the contour is found using Welzl’s algorithm.

- ## Find MEC (Welzl's Algorithm)

  The Welzl function implements the Welzl's algorithm, which recursively computes the minimum enclosing circle for a set of points. It takes a list of 2D points as input and returns the minimum enclosing circle. The Recursive algorithm that can find the minimal circle in O(n) time for a set of n points.



Here's how the function works:

- Inputs:

  The function takes a list of 2D points (P) as input. These points represent the set of points for which the minimum enclosing circle needs to be found.

- Initialization: The function initializes some helper functions and classes:
  - Point_W: Represents a 2D point with X and Y coordinates.
  - Circle_W: Represents a 2D circle with a center (C) and a radius (R).
  - dist_2points(a, b): Calculates the Euclidean distance between two points.
  - point_inside_circle(c, p): Checks if a point lies inside or on the boundaries of a circle.

- is_circle_valid(c: Circle, P: list[Point_W]): Checks if the given circle encloses all points in the list by verifying each point's position relative to the circle.
- get_circle_trivial(P: list[Point_W]): function determines the Minimum Enclosing Circle (MEC) for a small set of points (0, 1, 2, or 3) by handling base cases and exploring combinations of points to find the MEC efficiently.
- get_circle_2points (A: Point_W, B: Point_W): This function takes two points (A and B) as input and returns the smallest circle that passes through both points, with the center being the midpoint of the two points and the radius being half the distance between them.
- get_circle_3points (A: Point_W, B: Point_W, C:Point_W): Given three distinct points (A, B, and C), this function calculates the center and radius of the unique circle that passes through all three points, utilizing algebraic equations derived from the properties of circles and lines.

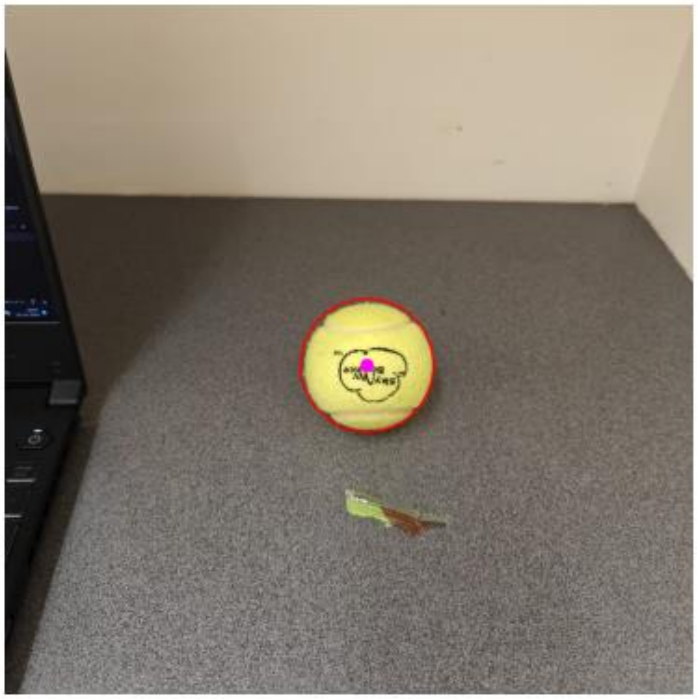
- Working:
  - The main function welzl() is a wrapper function that shuffles the input points randomly and then calls the helper function welzl_helper() with the shuffled points, an empty list (representing the set of points on the circle boundary), and the number of points.
  - The welzl_helper() function is a recursive function that computes the minimum enclosing circle.
  - At each recursive call, it selects a random point from the input points.
  - It then recursively computes the minimum enclosing circle for the remaining points without the selected point.
  - If the selected point lies inside the computed circle, the circle is returned.
  - Otherwise, the selected point is added to the boundary set, and the minimum enclosing circle is computed recursively with the updated sets of points.

- The recursion terminates when either all points are processed or the boundary set contains three points (which determines the circle uniquely).

- Return Value:
  - The function returns the minimum enclosing circle that encloses all the input points.

Overall, the Welzl function efficiently computes the minimum enclosing circle for a set of 2D points using the Welzl's algorithm, which has a time complexity of O(n) on average, where n is the number of input points.

Applying Welzl's algorithm on the biggest contour in the binary image, we get the minimum enclosing circle that contains all the contour points. The circle detected has been marked in red with the center being marked in purple. The circle detection is accurate and fast enough to be applied in real-time applications.



Conclusion

This project successfully implemented a computer vision-based Snake game that allows players to control the snake's movement using real-world objects detected through a camera feed. By integrating advanced computer vision algorithms, we achieved an engaging and immersive gameplay experience that blends the classic mechanics of the Snake game with innovative interaction methods.

The implementation leveraged various computer vision techniques, including color-based segmentation, contour detection, and object localization algorithms. The Welzl's algorithm was employed to efficiently compute the minimum enclosing circle around the detected green object, enabling precise tracking of the snake's head position.

Furthermore, the project addressed several challenges encountered during the development process. Handling collision detection between the snake's head and obstacles, as well as with its own body, required careful consideration of geometric calculations, edge cases, and performance optimization. The integration of computer vision components with the game logic added an extra layer of complexity, which was successfully navigated.

Through meticulous coding and optimization efforts, we achieved real-time performance, ensuring a smooth and responsive gameplay experience. The use of highly optimized computer vision libraries, such as OpenCV, and leveraging hardware acceleration capabilities contributed to the overall efficiency of the system.

The project not only delivered an entertaining Snake game experience but also demonstrated the potential of computer vision technology in enhancing traditional gaming experiences. By seamlessly blending virtual and real-world elements, we created a unique and captivating gaming environment that encourages player interaction and engagement.

Overall, this project serves as a testament to the versatility and power of computer vision techniques, paving the way for future innovations in gaming and interactive applications. The successful implementation of this computer vision-based Snake game opens up exciting possibilities for further exploration and development in the field of interactive media and entertainment.

Team Contributions

Harsh Jalutharia (hj2607):

- Game Mechanics Implementation: Harsh took charge of implementing the core mechanics of the Snake game, ensuring smooth snake movement, food and obstacle generation, and a functional scoring system.
- Algorithm Optimization: He optimized algorithms to ensure efficient performance, focusing on scalability and real-time execution across various hardware configurations.
- Gameplay Experience Enhancement: Harsh contributed to enhancing the gameplay experience by fine-tuning the game mechanics to provide an engaging and intuitive experience for players.
- Presentation: Harsh meticulously crafted every aspect of the presentation document, ensuring it displayed elegance in line with the project's requirements. His presentation skillfully highlighted not only his individual contributions but also effectively showcased the collaborative efforts of the entire team throughout the project's development.

Mudit Nigam (mn3439):

- Computer Vision Integration: Mudit spearheaded the integration of computer vision into the game, enabling the detection of real-world objects through the camera feed.
- Object Detection Algorithms: He developed and fine-tuned computer vision algorithms to accurately detect objects within the game environment in real-time, enabling players to control the snake's movements using detected objects.
- User Interface Design: Mudit also contributed to designing the user interface, ensuring seamless interaction between the virtual game world and the real-world objects captured by the camera feed.

- Documentation: Mudit played a role in documentation efforts by adding images and detailed function descriptions to the project documentation. His contributions helped in illustrating the project's functionality and providing clear instructions for users and developers alike.

Rhuthvik Dendukuri (rd3377):

- Innovative Gameplay Features: Rhuthvik focused on introducing innovative gameplay features by leveraging computer vision technology. He brainstormed and implemented features that allowed players to control the snake using real-world objects, adding a unique layer of interactivity to the classic game.
- Obstacle Generation: Rhuthvik contributed to the generation of obstacles within the game environment, enhancing the gameplay challenge and variety.
- User Interface Design: Rhuthvik participated in designing the user interface, ensuring a visually appealing and intuitive interface for players to interact with.
- Performance Testing: Rhuthvik was responsible for performance testing, ensuring that the game ran smoothly across different devices and environments. He conducted rigorous testing to identify and address any performance bottlenecks, optimizing the game for optimal performance.
- Documentation: Rhuthvik also contributed to documenting the project's progress to showcase the team's efforts and achievements. His documentation helped in capturing the project's journey and sharing insights regarding several aspects of the project.

Appendix

Code

```
import cv2
import imutils
import numpy as np
import math
from tkinter import messagebox as mb
import random

# Snake game in Python


WIN_SCORE = 10
MAX_OBSTACLES = 5
MIN_DISTANCE_BW_OBSTACLES = 60
MIN_DISTANCE_FOOD_OBSTACLE = 60
LENGTH_INCREASE_FROM_FOOD = 20
STARTING_LENGTH = 70
SNAKE_COLOR = (0, 0, 255)
SNAKE_THICKNESS = 12
GREEN_LOWER_THRESHOLD = (29, 86, 18)
GREEN_UPPER_THRESHOLD = (93, 255, 255)
VIDEO_WIDTH = 640
VIDEO_HEIGHT = 480

# Distance function
def distance(pt1, pt2):
    return np.sqrt((pt1[0] - pt2[0]) ** 2 + (pt1[1] - pt2[1]) ** 2)

# Check if a point on overlaps an image on the frame
def checkPointOverlapImage(pointLocation, imageLocation, imageSize):
    return (imageLocation[0] - imageSize[0]//2 < pointLocation[0] < imageLoca-
tion[0] + imageSize[0]//2 and \
        imageLocation[1] - imageSize[1]//2 < pointLocation[1] < imageLocation[1]
+ imageSize[1]//2)

# Draw image over frame
def drawImageOverFrame(frame, image, imageLocation):
    imageSize = image.shape[1], image.shape[0]
```

```
    frame[(imageLocation[1] - (imageSize[1]//2)) : (imageLocation[1] + (im-
ageSize[1]//2)),
           (imageLocation[0] - (imageSize[0]//2)) : (imageLocation[0] + (im-
ageSize[0]//2))] = image

# Class for running the game logic
class SnakeGame:
    def __init__(self, foodImagePath, obstacleImagePath) -> None:
        self.points = []
        self.lengths = []
        self.totalLength = STARTING_LENGTH
        self.currentLength = 0
        self.obstacleLocations = []
        self.obstacleCount = MAX_OBSTACLES
        self.previousHead = 0, 0
        self.score = 0
        self.gameOver = False

        self.obstacleImage = cv2.imread(obstacleImagePath, cv2.IMREAD_UNCHANGED)
        self.obstacleImageSize = self.obstacleImage.shape[1], self.obstacleI-
mage.shape[0]
        self.generateObstacles()

        self.foodImage = cv2.imread(foodImagePath, cv2.IMREAD_UNCHANGED)
        self.foodImageSize = self.foodImage.shape[1], self.foodImage.shape[0]
        self.foodLocation = 0, 0
        self.generateRandomFoodLocation()

    # Randomly generate obstacles
    def generateObstacles(self):
        self.obstacleLocations = []
        while len(self.obstacleLocations) < self.obstacleCount:
            temp_point = (random.randint(50, VIDEO_WIDTH-50), random.randint(50,
VIDEO_HEIGHT-50))
            if all(distance(temp_point,p) >= MIN_DISTANCE_BW_OBSTACLES for p in
self.obstacleLocations):
                self.obstacleLocations.append(temp_point)

    # Randomly generate food location
    def generateRandomFoodLocation(self):
        while True:
            self.foodLocation = random.randint(50, VIDEO_WIDTH-50), random.rand-
int(50, VIDEO_HEIGHT-50)
            if all(distance(self.foodLocation, obstacle) > MIN_DISTANCE_FOOD_OB-
STACLE for obstacle in self.obstacleLocations):
```

```
                break

    # Draw snake, food and obstacles
    def drawObjects(self, image):
        # Draw snake
        if self.points:
            for i, _ in enumerate(self.points):
                if i != 0:
                    cv2.line(image, self.points[i], self.points[i-1],
SNAKE_COLOR, SNAKE_THICKNESS)
            cv2.circle(image, self.points[-1], int(SNAKE_THICKNESS/2),
SNAKE_COLOR, cv2.FILLED)

        # Draw food
        drawImageOverFrame(image, self.foodImage, self.foodLocation)

        # Draw obstacles
        for location in self.obstacleLocations:
            drawImageOverFrame(image, self.obstacleImage, location)

        return image

    # Reset game variables
    def reset(self):
        self.points = []
        self.lengths = []
        self.totalLength = STARTING_LENGTH
        self.currentLength = 0
        self.obstacleLocations = []
        self.obstacleCount = MAX_OBSTACLES
        self.previousHead = 0, 0
        self.score = 0
        self.gameOver = False
        self.generateObstacles()
        self.generateRandomFoodLocation()

    # Called when game is finished
    def gameFinished(self, frame, win: bool):
        if win:
            cv2.putText(frame, 'YOU WIN!!', (100, 250), cv2.FONT_HERSHEY_SIMPLEX,
3, (0, 255, 0), 3)
        else:
            cv2.putText(frame, 'GAME OVER!!', (30, 250), cv2.FONT_HERSHEY_SIM-
PLEX, 3, (0, 0, 255), 3)
        cv2.imshow("Camera", frame)
```

```
        res = mb.askquestion('Exit Application', 'Play again?')
        if res == 'yes' :
            self.reset()
        else:
            global playGame
            playGame = False

    # Actual game loop is implemented here
    def updateSnake(self, image, NewHeadLocation):
        self.points.append(NewHeadLocation)
        dist_to_last_head = distance(self.previousHead, NewHeadLocation)
        self.lengths.append(dist_to_last_head)
        self.currentLength += dist_to_last_head
        self.previousHead = NewHeadLocation

        # Reduce current length if more than total length
        if self.currentLength > self.totalLength:
            for len_i, length in enumerate(self.lengths):
                self.currentLength -= length
                self.points.pop(len_i)
                self.lengths.pop(len_i)
                if self.currentLength < self.totalLength:
                    break

        # Check if food was eaten
        if checkPointOverlapImage(NewHeadLocation, self.foodLocation,
self.foodImageSize):
            self.totalLength += LENGTH_INCREASE_FROM_FOOD
            self.score += 1
            self.generateRandomFoodLocation()

        # Display snake, food, and obstacles
        image = self.drawObjects(image)

        # Display score
        cv2.putText(image, 'Score :' + str(self.score), (450, 100), cv2.FONT_HER-
SHEY_SIMPLEX, 1, (255, 0, 203), 2)

        # Check collision with obstacles
        for location in self.obstacleLocations:
            if checkPointOverlapImage(NewHeadLocation, location, self.obstacleI-
mageSize):
                self.gameFinished(image, False)

        # Check collision with snake body
```

```
        check_pts = np.array(self.points[:-6], np.int32)
        check_pts = check_pts.reshape((-1,1,2))
        cv2.polylines(image, [check_pts], False, (255,0,0), 0)
        minimum_dist = cv2.pointPolygonTest(check_pts, NewHeadLocation, True)
        if -1 <= minimum_dist<= 1:
            self.gameFinished(image, False)

        # Check win condition
        if self.score >= WIN_SCORE:
            self.gameFinished(image, True)

        return image

# Function for erosion
def erode(mask, kernel_size=(3,3), iterations=1):
    eroded_mask = mask.copy()
    for _ in range(iterations):
        eroded_mask = np.minimum.reduce([
            eroded_mask[i:mask.shape[0]-kernel_size[0]+i+1, j:mask.shape[1]-ker-
nel_size[1]+j+1]
            for i in range(kernel_size[0])
            for j in range(kernel_size[1])])
    return eroded_mask

# Function for dilation
def dilate(mask, iterations=2):
    dilated_mask = mask.copy()
    for _ in range(iterations):
        dilated_mask[1:-1, 1:-1] |= dilated_mask[:-2, 1:-1]
        dilated_mask[1:-1, 1:-1] |= dilated_mask[2:, 1:-1]
        dilated_mask[1:-1, 1:-1] |= dilated_mask[1:-1, :-2]
        dilated_mask[1:-1, 1:-1] |= dilated_mask[1:-1, 2:]
    return dilated_mask

# 2D point for welzl's algorithm
class Point_W:
    def __init__(self, X=0, Y=0) -> None:
        self.X=X
        self.Y=Y

# Circle for welzl'a algorithm
class Circle_W:
    def __init__(self, c=Point_W(), r=0) -> None:
        self.C=c
        self.R=r
```

```
# Distance between 2 point_w
def dist_2points(a: Point_W, b: Point_W) -> float:
    return math.sqrt((a.X - b.X)**2 + (a.Y - b.Y)**2)

# Check if point is inside circle
def point_inside_circle(c: Circle_W, p: Point_W) -> bool:
    return dist_2points(c.C, p) <= c.R

# Check if points P lie in circle
def is_circle_valid(c: Circle_W, P: list[Point_W]):
    for p in P:
        if (not point_inside_circle(c, p)):
            return False
    return True

# Get circle for trivial cases: len(P) <= 3
def get_circle_trivial(P: list[Point_W]):
    if not P:
        return Circle_W()
    elif (len(P) == 1):
        return Circle_W(P[0], 0)
    elif (len(P) == 2):
        return get_circle_2points(P[0], P[1])
    for i in range(3):
        for j in range(i + 1,3):
            c = get_circle_2points(P[i], P[j])
            if (is_circle_valid(c, P)):
                return c
    return get_circle_3points(P[0], P[1], P[2])

# Returns smallest circle with the 2 points on boundary
def get_circle_2points(A: Point_W, B: Point_W):
    Center = Point_W((A.X + B.X) / 2.0, (A.Y + B.Y) / 2.0 )
    return Circle_W(Center, dist_2points(Center, A))

# Returns smallest circle with the 3 points on boundary
def get_circle_3points(A: Point_W, B: Point_W, C: Point_W):
    baX = B.X - A.X
    baY = B.Y - A.Y
    caX = C.X - A.X
    caY = C.Y - A.Y
    tB = baX**2 + baY**2
    tC = caX**2 + caY**2
    tD = baX*caY - baY*caX
```

```
    Center = Point_W(A.X + (caY*tB - baY*tC)/(tD*2), A.Y + (baX*tC -
caX*tB)/(tD*2))
    return Circle_W(Center, dist_2points(Center, A))

# Recursive welzl helper function
def welzl_helper(P: list[Point_W], R: list[Point_W], n: int):
    if (n == 0 or len(R) == 3) :
        return get_circle_trivial(R)
    r_index = random.randint(0,n-1)
    random_point = P[r_index]
    P[r_index],P[n-1] = P[n-1],P[r_index]
    temp_C = welzl_helper(P, R.copy(), n-1)
    if (point_inside_circle(temp_C, random_point)):
        return temp_C
    R.append(random_point)
    return welzl_helper(P, R.copy(), n-1)

# Applies Welzl's algorithm on points P
def welzl(P: list[Point_W]):
    P_copy = P.copy()
    random.shuffle(P_copy)
    return welzl_helper(P_copy, [], len(P_copy))

playGame = True
game = SnakeGame("apple.jpg", "hurdle1small.jpg")
cap = cv2.VideoCapture(0)
cap.set(3, 720)
cap.set(4, 720)


while playGame:
    ret, frame = cap.read()
    frame = cv2.flip(frame, 1)

    # Image preprocessing
    img = cv2.GaussianBlur(frame, (11, 11), 2)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    mask = ((np.logical_and(np.all(img >= GREEN_LOWER_THRESHOLD, axis = -1),
np.all(img <= GREEN_UPPER_THRESHOLD, axis = -1))).astype(np.uint8))*255
    mask = erode(mask)
    mask = dilate(mask)

    # Find contours in image
    contours = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = imutils.grab_contours(contours)
```

```
    if len(contours) > 0:
        ball_contour = max(contours, key=cv2.contourArea)
        # Assuming ball_contour is the largest contour obtained from the binary
mask image

        # Convert the contour to a list of points
        points = [Point_W(x[0][0], x[0][1]) for x in ball_contour]

        # Find the minimum enclosing circle using Welzl's algorithm
        min_enclosing_circle = welzl(points)
        center = (int(min_enclosing_circle.C.X), int(min_enclosing_circle.C.Y))
        radius = int(min_enclosing_circle.R)

        # If radius big enough, run game loop
        if radius > 10:
            frame = game.updateSnake(frame, center)
        # otherwise, only display game objects
        else:
            frame = game.drawObjects(frame)
    else:
        frame = game.drawObjects(frame)

    cv2.imshow("Camera", frame)
    cv2.imshow("Mask", mask)
    if cv2.waitKey(1) == 27:
        break

cv2.destroyAllWindows()
cap.release()
```