**CS2203 AI PROJECT – IIT PATNA**

**Khaled Hussain Mohammed – 2301CS24**

**Srichetan Reddy – 2301CS54**

**G. Praneeth Reddy – 2301CS36**

**B. Rahen - 2301CS37**

**Koushik – 2302CS02**

# Fruit Ninja Bot Analysis

## Overview

This script is a bot designed to automate gameplay for what appears to be Fruit Ninja or a similar fruit-slicing game. The bot uses computer vision techniques to detect fruits and bombs on the screen, then executes mouse movements to slice fruits while avoiding bombs.

## Key Components

### Technical Foundation

- Uses Python with libraries like OpenCV, MSS (screen capture), NumPy, and Win32API
- Captures game screen in a specific region of the display
- Processes images to detect objects based on color ranges
- Can record gameplay videos when run with the 'save' parameter

### Game Interaction

- Performs "swipes" by generating mouse movements to slice detected fruits
- Prioritizes fruits closest to the screen edges (likely to fall out of view soon)
- Implements bomb avoidance by predicting bomb trajectories
- Uses a caching system to avoid slicing the same area repeatedly

### Object Detection

- Identifies fruits and bombs through color masking and contour detection
- Uses different color boundaries for various fruit types
- Calculates precise object centers for accurate slicing

### Safety Features

- Checks if swiping paths intersect with bombs or their predicted trajectories
- Tries different slicing angles when bombs block the direct path
- Includes safeguards to prevent cursor movement outside game boundaries

**Technical Implementation Details**

**Screen Handling**

- Game resolution is set to 1109x742 pixels
- Places game in the top-right corner of the screen
- Uses a multi-threaded approach to handle swipes without blocking main detection loop

**Bomb Avoidance Algorithm**

- Maintains lists of bombs from current and previous frames
- Predicts bomb trajectories based on previous positions
- Enforces minimum safe distance (58 pixels) from bombs
- Tests multiple slicing angles when direct paths are unsafe

**Performance Optimization**

- Caches recently swiped areas to avoid redundant actions
- Sorts fruits by priority (those closest to falling out of view)
- Uses threading to separate detection from mouse movement execution

**Use Cases**

- Automation of gameplay for high scores
- Testing game mechanics and performance
- Creating demonstration videos with the recording functionality

**Limitations**

- Relies on specific screen coordinates and color values
- May require recalibration for different display settings
- Performance depends on processing power and screen capture speed

This bot demonstrates sophisticated computer vision techniques for real-time game automation, balancing speed and accuracy to maximize fruit slicing while avoiding bombs.

**Key Code Sections Explained**

**Screen Capture Setup**

gameScreen = {'top': 25, 'left': screenWidth - width, 'width': width, 'height': height}

This defines the region of the screen where the game is displayed. It positions the capture area at the top-right corner of the screen, with dimensions matching the game's resolution (1109x742).

**Coordinate Translation**

```
def realCoord(x, y):
    rx = int(x)
    ry = int(y)
    rx, ry = sanitizeMargins(int(x), int(y))
    rx += gameScreen['left']
    ry += gameScreen['top']
    return rx, ry
```

This crucial function translates coordinates within the game screen to absolute screen coordinates. It ensures actions happen at the correct position on your monitor by adding the game screen's offset, and includes margin checking to prevent out-of-bounds actions.

Swipe Implementation

```
def swipe(_x1, _y1, _x2, _y2):
    x1, y1 = realCoord(_x1, _y1)
    x2, y2 = realCoord(_x2, _y2)
    points = 251
    for i in range(points+1):
        moveMouse((x1 * (points - i) + x2 * i)/points, (y1 * (points - i) + y2 * i)/points)
        if (i == 0):
            win32api.mouse_event(win32con.MOUSEEVENTF_LEFTDOWN, 0, 0, 0, 0)
            moveMouse((x1 * (points - i) + x2 * i)/points, (y1 * (points - i) + y2 * i)/points)
    win32api.mouse_event(win32con.MOUSEEVENTF_LEFTUP, 0, 0, 0, 0)
```

This function creates a smooth slicing motion by:

1. Converting game coordinates to screen coordinates
2. Breaking the swipe into 251 small incremental moves
3. Pressing the mouse down at the start point
4. Moving gradually along the path using linear interpolation
5. Releasing the mouse at the end point

The high number of points (251) ensures the game registers the swipe as a continuous motion.

Bomb Safety Check

```
def lineIsSafe(x1S, y1S, x2S, y2S):
    global bombMinDistance
    for bomb in bombs:
        xBomb, yBomb = bomb
        # Code to find previous bomb position...

        # Predict bomb trajectory
        predicts = [(xBomb + (xBomb - xPrev) * 3, yBomb + (yBomb - yPrev) * 3)]

        if (yBomb < bomgHeightCond):
            predicts.append((xBomb + (xBomb - xPrev) * 3, yBomb + bomdDown))
            predicts.append((xBomb, yBomb + bomdDown))

        # Check if swipe intersects with bomb path...
```

This sophisticated function:

1. Takes a potential swipe line as input
2. For each detected bomb, finds its position in the previous frame
3. Calculates expected bomb trajectories based on movement vectors
4. Samples multiple points along both the swipe path and bomb trajectories
5. Returns false if any point on the swipe path comes too close to a bomb's current or predicted position

Object Detection Loop

```
objectBoundaries = [([15, 180, 130], [35, 237, 209], 120, True), # fruit
                    ([0, 40, 10], [30, 170, 50], 100, True), # fruit
                    ([10, 50, 220], [40, 250, 255], 80, True), # fruit
                    ([30, 30, 20], [60, 70, 60], 60, False), # bomb
                   ]

# In the main loop:
for boundary in objectBoundaries:
```

```
    lower, upper, minPoints, isFruit = boundary
    tmpMask = cv2.inRange(img, np.array(lower), np.array(upper))
    contours, hierarchy = cv2.findContours(tmpMask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

This code defines color ranges for different game objects and processes each one by:

1. Creating a binary mask that isolates pixels within the specified color range
2. Finding contours (shapes) in that mask
3. Filtering contours based on size (minPoints)
4. Categorizing objects as fruits or bombs based on the boolean flag

Fruit Prioritization

```
def orderFruitsBy(tup):
    # always slice the fruit closer to the edge
    x, y = tup
    toFilter = [height - y, x]
    if x > width * 4/5:
        toFilter.append(width - x)
    return min(toFilter)

# In the main loop:
possibleSwipes.sort(key=lambda tup: orderFruitsBy(tup))
```

This prioritizes fruits that are:

1. Lower on the screen (closer to falling out of view)
2. At the left edge
3. At the right edge (if in the rightmost fifth of the screen)

The sort function uses the minimum of these values, ensuring that fruits in the most critical positions are sliced first.

These code sections work together to create an intelligent bot that continuously monitors the game screen, identifies objects, prioritizes targets, and executes precise slicing movements while avoiding dangerous objects.