**COMPUTER VISION**

**CS – GY 6643**

**PROJECT: SNAKE GAME**

**TEAM:**

**HARSH JALUTHARIA – *hj2607***

**MUDIT NIGAM – *mn3439***

**RHUTHVIK DENDUKURI – *rd3377***

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 food items, 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 researchers to concentrate on creating unique vision techniques without being distracted by intricate rendering or physics 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 hurdle 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

# Features

* Real - Time Object Detection: The game utilizes CV algorithms to detect real – world objects, allowing players to controls the snake’s movements using their surroundings.
* Dynamic Hurdle generation: Hurdles are dynamically generated within the game frame, providing challenges for players to navigater around.
* Restriction – less movement: Unlike the classic game, the movement of the snake is not restricted to the 4 directional movements. It is made to be a free flow all around the game frame.

Computer Vision Technique Implementations

## 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.

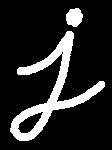
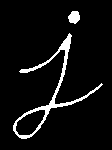
## Mask

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.

## Erode and Dilate

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.

-> 

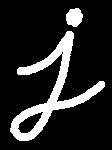
Here's how the function works:

1. 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.
2. Initialization: The function initializes a square structuring element (kernel) with the specifiedsize using NumPy's np.ones() function. The structuring element is a binary matrix that defines the neighborhood used for the erosion operation.
3. Erosion Operation: The erosion 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 eroded\_mask.
     + It loops through each pixel in the mask.
     + At each pixel, it computes the minimum value of the neighbourhood defined by the kernel.
     + The minimum value represents the erosion effect, where the centre 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.
4. 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.

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:

1. 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.
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 neighbourhood defined by a structuring element.
     + The maximum value represents the dilation effect, where the centre pixel of the structuring element is set to the maximum value of the pixels within its neighbourhood.
     + This process effectively expands the boundaries of the foreground objects in the mask.
3. 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.

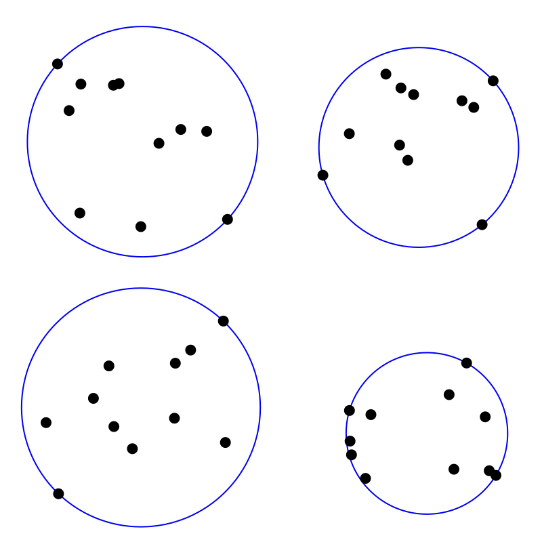
## Find Contour

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.

## Find MEC (Welzl)

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:

1. 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.

1. Initialization: The function initializes some helper functions and classes:
   * + Point: Represents a 2D point with X and Y coordinates.
     + Circle: Represents a 2D circle with a center (C) and a radius (R).
     + dist(a, b): Calculates the Euclidean distance between two points.
     + is\_inside(c, p): Checks if a point lies inside or on the boundaries of a circle.
     + get\_circle\_center(bx, by, cx, cy): Helper method to compute the center of a circle given three points. For 3 points, the circle is the circumcircle of the triangle formed by the points.
     + def circle\_from1(A: Point, B: Point): Computes the MEC intersecting two points by setting the center at their midpoint and radius as half the distance
     + def circle\_from2(A: Point, B: Point, C: Point): Computes a unique MEC intersecting three points using geometric calculations for the center and radius.
     + def is\_valid\_circle(c: Circle, P: list[Point]): Checks if the given circle encloses all points in the list by verifying each point's position relative to the circle.
     + def min\_circle\_trivial(P: list[Point]): Determines the MEC for a small number of points (0, 1, 2, or 3) directly, handling base cases.
2. 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).
3. 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.