**ANALYSIS OF ALGORITHM SEMESTER PROJECT**

HAMMAD AHMAD 46874 BILAL AHMED KHALIL 44926

**1. Introduction**

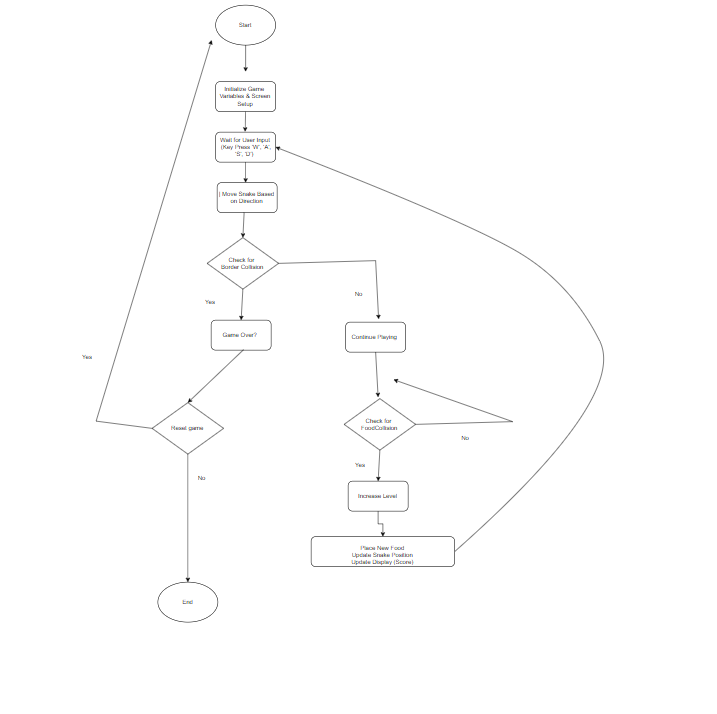
The project is a classic implementation of the Snake game using Python's turtle graphics module. In this game, the player controls a snake that moves around the screen, collecting food to grow longer. The game ends if the snake collides with the screen borders or with itself. The objective is to score as many points as possible by collecting food items.

**2. Functionality Overview**

The main functionalities of the game include:

* **Snake Movement**: The snake can move in four directions using the keys "W" (up), "S" (down), "A" (left), and "D" (right).
* **Food Generation**: Food appears at random locations on the screen. When the snake eats the food, its length increases.
* **Score Tracking**: The game keeps track of the current score and the high score.
* **Collision Detection**: The game detects collisions with the borders and with the snake's own body, ending the game when a collision occurs.

**Flow Chart**:

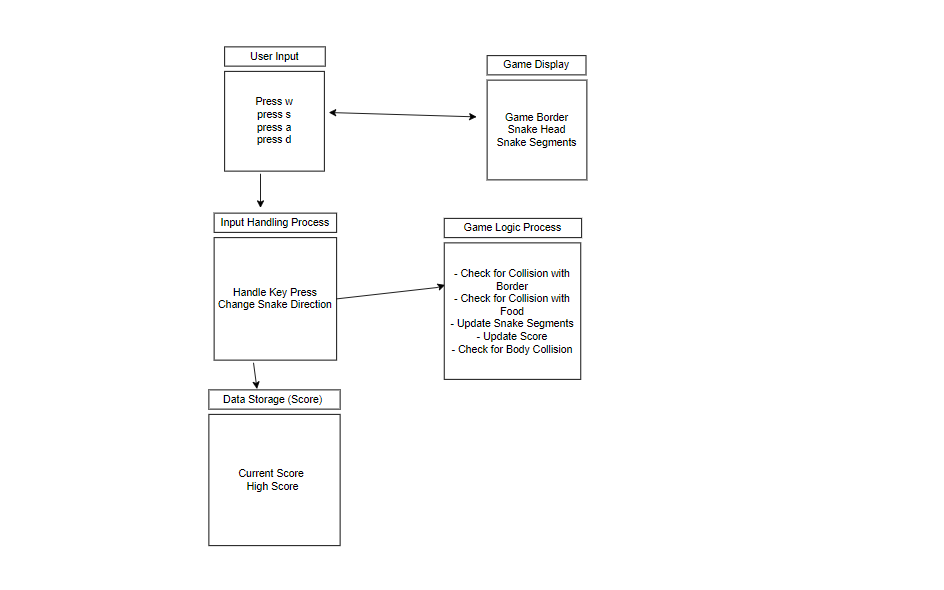


**3. Code Structure**

The game is structured into various components:

* **Screen Setup**: Initializes the game window using turtle.Screen().
* **Snake and Food Initialization**: Creates the snake's head and food using turtle.Turtle() objects.
* **Game Loop**: Continuously updates the game state, handles input, moves the snake, checks for collisions, and updates the score.
* **Functions**: Separate functions are defined for handling movement, food consumption, and collision detection.

**Data flow Diagram (DFD):**

****

**4. Time Complexity Analysis**

The game loop runs continuously and is responsible for updating the game state. It checks for collisions, updates positions, handles input, and manages the score. Here’s a closer look at each operation within the loop and its time complexity:

1. **Screen Update**:
   * wn.update() is called at the start of each loop iteration. This function updates the graphics on the screen and is generally considered O(1) because it does not depend on the number of segments or other game elements.
2. **Collision with Borders**:
   * The conditions:

python

Copy code

if head.xcor() > 290 or head.xcor() < -290 or head.ycor() > 290 or head.ycor() < -290:

This check involves four comparisons to see if the snake's head has gone beyond the screen boundaries. Each comparison takes constant time, so this entire operation is O(1).

1. **Collision with Food**:
   * The condition:

python

Copy code

if head.distance(food) < 20:

Here, the distance() function calculates the distance between the snake's head and the food. This involves some arithmetic operations (subtraction, squaring, and taking a square root) which all run in constant time. Thus, this check is also O(1).

1. **Collision with Body Segments**:
   * The most complex part of the loop is checking if the head collides with any of the body segments:

python

Copy code

for segment in segments:

if segment.distance(head) < 20:

* + Here, the loop iterates over the segments list. If n is the number of segments, this loop will run n times. Each iteration involves a constant-time distance check, leading to a total complexity of O(n) for this operation.

1. **Movement of the Snake**:
   * The movement of the snake’s segments also involves a loop:

python

Copy code

for index in range(len(segments) - 1, 0, -1):

x = segments[index - 1].xcor()

y = segments[index - 1].ycor()

segments[index].goto(x, y)

* + This loop again iterates over the segments, moving each segment to the position of the segment in front of it. This operation runs in O(n) time since it processes all n segments.

1. **Updating Segment 0 Position**:
   * After moving the segments, the position of segment 0 is updated to match the head:

python

Copy code

if len(segments) > 0:

x = head.xcor()

y = head.ycor()

segments[0].goto(x, y)

* + This operation checks the length of segments (constant time, O(1) and updates segment 0’s position (also O(1)). So, this is still O(1)

1. **Resetting After Collision**:
   * If a collision is detected, resetting the snake's position and clearing segments involves:

python

Copy code

head.goto(0, 0)

head.direction = "stop"

for segment in segments:

segment.goto(1000, 1000)

segments.clear()

* + Setting the head position and stopping direction are **O(1)** operations. The loop that moves each segment off-screen is **O(n),** and clearing the list is **O (1)** Therefore, resetting after a collision takes **O(n)** in the worst case.

**Total Time Complexity for One Iteration of the Game Loop**

Now, let's summarize the time complexities of each component within a single iteration of the game loop:

* Screen update: O(1)
* Border collision check: O(1)
* Food collision check: O(1)
* Body collision check: O(n)
* Moving segments: O(n)
* Updating segment 0 position: O(1)
* Resetting after collision: O(n) (in the worst case)

A screenshot of a computer screen

Description automatically generated

Combining all these operations, the total time complexity for one iteration of the game loop is:

O(1)+O(1)+O(1)+O(n)+O(n)+O(1)+O(n)=O(n)

Thus, we conclude that the overall time complexity for each iteration of the main game loop is O(n), where n is the number of segments in the snake. This is primarily due to the body collision check and the movement of the segments. As the snake grows longer (more segments), the time taken for these operations increases linearly.

**5. Space Complexity Analysis**

The space complexity of the game is primarily determined by the data structures used:

* **Snake Segments**:  
  Each segment is represented by a turtle.Turtle() object and stored in the segments list. Thus, the space complexity is **O(n)**, where **n** is the number of segments in the snake.
* **Other Variables**:  
  Variables like score, high\_score, delay, head, and food consume a constant amount of space.  
  **Space Complexity**: **O(1)**.

|  |  |  |
| --- | --- | --- |
| Component / Function | Description | Space Complexity |
| Global Variables | delay, score, high\_score | O(1) |
| Screen Setup (wn) | turtle.Screen() object for game window | O(1) |
| Border Drawing (border) | turtle.Turtle() object to draw borders | O(1) |
| Score Display (pen) | turtle.Turtle() object for displaying score | O(1) |
| Snake Head (head) | turtle.Turtle() object for the snake's head | O(1) |
| Food (food) | turtle.Turtle() object for the food item | O(1) |
| Snake Segments (segments list) | List of turtle.Turtle() objects for snake body | O(n) |
| Keyboard Binding | Event listeners for key presses | O(1) |
| Food Collision Check | Checks if the snake's head is near the food | O(1) |
| Border Collision Check | Checks if the snake's head hits the border | O(1) |
| Self-Collision Check | Checks if the snake's head collides with its body | O(n) |
| Move Function (move()) | Updates the snake's head position based on direction | O(1) |
| Segment Movement | Updates each segment's position based on the previous segment | O(n) |
| Adding New Segment | Adds a new segment when the snake eats food | O(1) |
| Pen Update | Clears and writes new score on the screen | O(1) |
| Main Game Loop | Runs continuously, handling updates and rendering | O(1) |

**Total Space Complexity: O(n)**The overall space complexity is O(n) due to the dynamic memory used by the snake segments (segments list). As the snake grows longer, more memory is needed to store **each new** segment.

**6. Strengths of the Implementation**

* **Clear Structure**: The code is well-organized, with separate functions for handling different aspects of the game (movement, collision detection, score updates).
* **Real-time Keyboard Input**: Uses wn.listen() and wn.onkeypress() for handling keyboard input, which allows for smooth, real-time control of the snake.
* **Dynamic Difficulty**: The speed of the game increases as the snake grows longer, making it progressively challenging.

**7. Conclusion**

Your Snake Game is a solid implementation of the classic game using Python's turtle module. It efficiently handles essential gameplay elements like movement, collision detection, and score tracking. However, there are opportunities for optimization in terms of both performance and user experience. By implementing some of the suggested improvements, you can make your game more robust, efficient, and enjoyable for players.

**Python Code:**

import turtle

import time

import random

delay = 0.1

# Score

score = 0

high\_score = 0

# Set up the screen

wn = turtle.Screen()

wn.title("Snake Game by Hammad")

wn.bgcolor("green")

wn.setup(width=600, height=600)

wn.tracer(0)  # Turns off the screen updates

# Draw border

border = turtle.Turtle()

border.speed(0)

border.color("white")

border.penup()

border.goto(-300, 300)

border.pendown()

border.pensize(3)

for \_ in range(4):

    border.forward(600)

    border.right(90)

border.hideturtle()

# Snake head

head = turtle.Turtle()

head.speed(0)

head.shape("square")

head.color("black")

head.penup()

head.goto(0, 0)

head.direction = "stop"

# Snake food

food = turtle.Turtle()

food.speed(0)

food.shape("circle")

food.color("red")

food.penup()

food.goto(0, 100)

segments = []

# Pen

pen = turtle.Turtle()

pen.speed(0)

pen.shape("square")

pen.color("white")

pen.penup()

pen.hideturtle()

pen.goto(0, 260)

pen.write("Score: 0  High Score: 0", align="center", font=("Courier", 24, "normal"))

# Functions

def go\_up():

    if head.direction != "down":

        head.direction = "up"

def go\_down():

    if head.direction != "up":

        head.direction = "down"

def go\_left():

    if head.direction != "right":

        head.direction = "left"

def go\_right():

    if head.direction != "left":

        head.direction = "right"

def move():

    if head.direction == "up":

        y = head.ycor()

        head.sety(y + 20)

    if head.direction == "down":

        y = head.ycor()

        head.sety(y - 20)

    if head.direction == "left":

        x = head.xcor()

        head.setx(x - 20)

    if head.direction == "right":

        x = head.xcor()

        head.setx(x + 20)

# Keyboard bindings

wn.listen()

wn.onkeypress(go\_up, "w")

wn.onkeypress(go\_down, "s")

wn.onkeypress(go\_left, "a")

wn.onkeypress(go\_right, "d")

# Main game loop

while True:

    wn.update()

    # Check for a collision with the border

    if head.xcor() > 290 or head.xcor() < -290 or head.ycor() > 290 or head.ycor() < -290:

        time.sleep(1)

        head.goto(0, 0)

        head.direction = "stop"

        # Hide the segments

        for segment in segments:

            segment.goto(1000, 1000)

        # Clear the segments list

        segments.clear()

        # Reset the score

        score = 0

        # Reset the delay

        delay = 0.1

        pen.clear()

        pen.write("Score: {}  High Score: {}".format(score, high\_score), align="center", font=("Courier", 24, "normal"))

    # Check for a collision with the food

    if head.distance(food) < 20:

        # Move the food to a random spot

        x = random.randint(-290, 290)

        y = random.randint(-290, 290)

        food.goto(x, y)

        # Add a segment

        new\_segment = turtle.Turtle()

        new\_segment.speed(0)

        new\_segment.shape("square")

        new\_segment.color("grey")

        new\_segment.penup()

        segments.append(new\_segment)

        # Shorten the delay

        delay -= 0.001

        # Increase the score

        score += 10

        if score > high\_score:

            high\_score = score

        pen.clear()

        pen.write("Score: {}  High Score: {}".format(score, high\_score), align="center", font=("Courier", 24, "normal"))

    # Move the end segments first in reverse order

    for index in range(len(segments) - 1, 0, -1):

        x = segments[index - 1].xcor()

        y = segments[index - 1].ycor()

        segments[index].goto(x, y)

    # Move segment 0 to where the head is

    if len(segments) > 0:

        x = head.xcor()

        y = head.ycor()

        segments[0].goto(x, y)

    move()

    # Check for head collision with the body segments

    for segment in segments:

        if segment.distance(head) < 20:

            time.sleep(1)

            head.goto(0, 0)

            head.direction = "stop"

            # Hide the segments

            for segment in segments:

                segment.goto(1000, 1000)

            # Clear the segments list

            segments.clear()

            # Reset the score

            score = 0

            # Reset the delay

            delay = 0.1

            # Update the score display

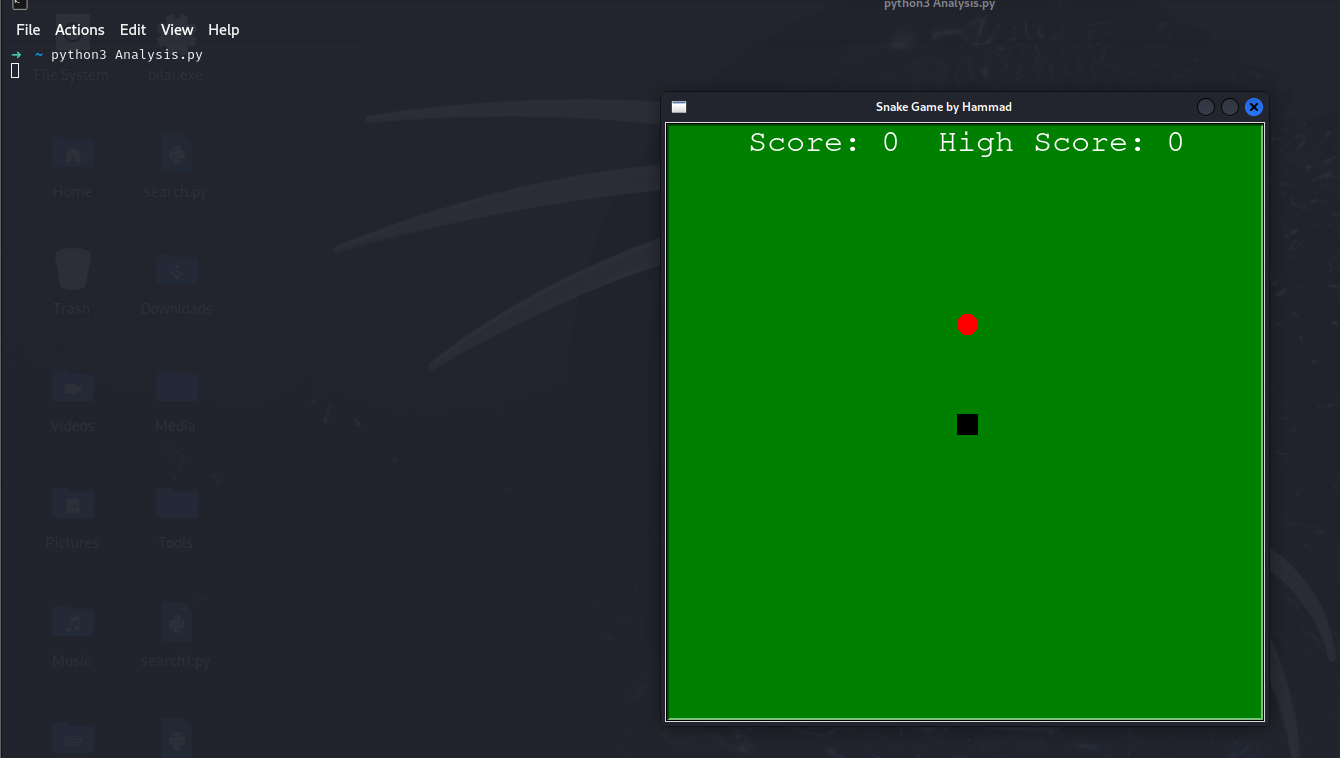
            pen.clear()

            pen.write("Score: {}  High Score: {}".format(score, high\_score), align="center", font=("Courier", 24, "normal"))

    time.sleep(delay)

wn.mainloop()

**POC:**

****