Snake game report

# Introduction

### **OVERVIEW**

This project is an implementation of The Snake Game in C programming language. The Snake Game is a game in which a snake moves along the playing area and grows longer by consuming food that randomly appears on the screen. Every time the snake consumes food, player’s score increases. Game ends (snake dies) when the snake runs into one of the four walls of the playing field or into itself.

Our game only works on Windows operating systems because it uses functions like *getch()* and *kbhit()* from the *conio.h* library, which are not available on Linux or Mac.

### **WHY DO WE USE GRID?**

We use a grid-based system because it makes it easier for us to control the game’s logic, track the snake’s position, detect collisions and place food. The grid divides the game area into individual cells, each of which can represent either an empty space, part of snake’s body, food, or game boundary. It also lets the user see the current game state in a clear and organized way.

# System design

### **GRID REPRESENTATION**

The playing field is defined with 10 rows and 25 columns and represented by a 2D array:

char area[ROWS][COLUMNS];

Each cell of this grid is updated to reflect the game state. Elements are represented by:

* ' ' — Empty cell
* '-' — Border of the grid
* 'O' — Snake’s head
* 'o' — Snake’s body segment
* '\*' — Food

The grid is initialized in the *create\_area()* function, which sets the borders using dashes (-) and leaves the rest empty.

### **SNAKE REPRESENTATION**

The snake is made of 2 parts: head and body. The snake’s body grows in length with every consumed food, while the head is the one making the movement and remains the same from the beginning.

* **Snake head:** tracked with 2 element array:

int snake\_head\_pos[2] = {row, col};

* **Snake Body:** represented by a 2D array where each part of the body has its own position (row and column):

int snake\_body[100][2];

# Functionality

### **MOVEMENT**

The player controls the snake using the w, a, s, and d keys, which represent up, left, down, and right. The *\_kbhit()* function from the *<conio.h>* library checks if a key has been pressed, and *\_getch()* reads the input immediately without waiting for user to press enter.

if (\_kbhit()) {

char act = \_getch();}

*act* variable takes the value of the key pressed by the user.

The *direction\_set* variable stores the current direction the snake is moving in. It stops the player from making a 180 degree turn. For example, if the snake is moving up (“w”), the player can't suddenly move it down (“s”), because that would make it crash into its own body.

if ((act == 'w' && direction\_set != 's') ||

(act == 'a' && direction\_set != 'd') ||

(act == 's' && direction\_set != 'w') ||

(act == 'd' && direction\_set != 'a')) {

direction\_set = act; }

If the input is valid, then the *direction\_set* variable gets the value of *act.*

The snake’s head position is updated based on the final direction.

if (direction\_set == 'w') {

snake\_head\_pos[0]--;

} else if (direction\_set == 'a') {

snake\_head\_pos[1]--;

} else if (direction\_set == 's') {

snake\_head\_pos[0]++;

} else if (direction\_set == 'd') {

snake\_head\_pos[1]++;

}

We store the head direction in an array that holds 2 numbers:

* *snake\_head\_pos[0]* – rows
* *snake\_head\_pos[1]* – columns

Since in a 2D grid the rows are numbered starting from 0 at the top, moving up means decreasing the row index (--), while moving down means increasing the row index (++). The columns are indexed starting from 0 on the left, so moving left decreases the column index (--), while moving right increases it (++).

Each time the snake moves, its previous head position becomes the first segment of its body. The body segments are updated by shifting them forward in the *snake\_body* array.

for (int i = \*length; i>0;i--) {

        snake\_body[i][0] = snake\_body[i-1][0];

        snake\_body[i][1] = snake\_body[i-1][1]; }

    if (1) {

        snake\_body[0][0] = prev\_row;

        snake\_body[0][1] = prev\_col; }

### **COLISSION DETECTION**

The program checks if the snake's head *( snake\_head\_pos[0]* and *snake\_head\_pos[1])* reaches any of the boundary positions:

if (snake\_head\_pos[0] == 0 || snake\_head\_pos[0] == ROWS - 1 || snake\_head\_pos[1] == 0 || snake\_head\_pos[1] == COLUMNS - 1) {

game\_over(); }

*First row – 0*

*Last row – ROWS-1*

*First column – 0*

*Last column – COLUMNS-1*

If the snake’s head *(snake\_head\_pos[0])* is at the first row or the last row, or if the head *(snake\_head\_pos[1])* is at the first column or the last column, snake has collided with the wall and *game\_over()* function is called.

When snake’s head runs into it’s own body, a collision occurs and *game\_over()* function is called. We check if the position of the head(row or column) corresponds to “o”, which is the way we denote snake body segments in our code.

if (area[snake\_head\_pos[0]][snake\_head\_pos[1]] == 'o') {

game\_over(); }

**FOOD GENERATION**

Food is denoted by “\*” and the position for it is randomly chosen by spawn\_food() function.

while (1) {

rand\_row = rand() % (ROWS - 1) + 1;

rand\_col = rand() % (COLUMNS - 1) + 1;

if (area[rand\_row][rand\_col] == ' ') {

break;

}

}

area[rand\_row][rand\_col] = '\*';

The function keeps on picking random positions for rows and columns until it finds a blank space (not occupied by snake). If the snakes’s head has moved on a cell with food inside it:

1. Player’s score increases.
2. Snake’s body gets longer.
3. New food spans in a different cell.

if (area[snake\_head\_pos[0]][snake\_head\_pos[1]] == '\*') {

(\*score)++;

(\*length)++;

spawn\_food(area, snake\_head\_pos);

}

# Implementation

### **LIBRARIES**

1. <stdio.h>: For input/output functions like printf(), scanf().
2. <conio.h>: For real-time keyboard input (getch(), kbhit()).
3. <stdlib.h>: For rand() and exit() functions.
4. <time.h>: Provides time(NULL) to seed the random number generator.
5. <unistd.h>: Allows us to control game speed.

Our game uses programming concepts like functions, arrays, loops, conditionals.

### **CHALLENGES ENCOUNTERED AND SOLUTIONS APPLIED**

We faced several challenges during the development of the game.

Getting input from the user without stopping the game/pressing enter was achieved using *\_kbhit()* and *\_getch()* functions.

We used a 2D array to track the snake's body and moved its segments to follow the head. We detected collisions and ended the game when the snake hit the walls or itself.

Food was generated at random positions that didn't overlap with the snake.

To prevent flickering, system("cls") was used to refresh the screen.

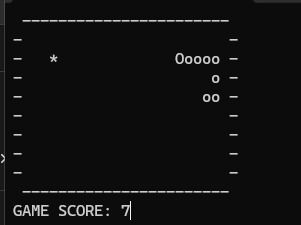
We also prevented the snake from reversing direction and controlled the game speed with usleep().

These solutions helped create a smooth, enjoyable game.

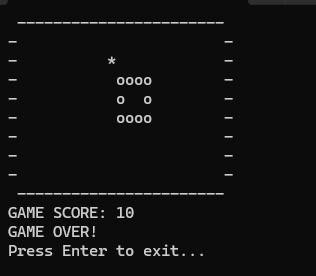
# Sample usage



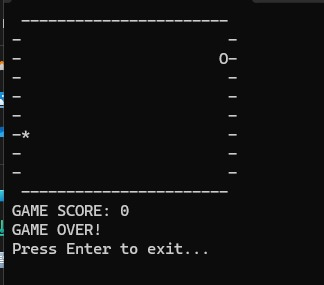
Beginning of the game



Game process



Snake dies when eating it’s tail



Snake dies when hitting the wall

# Testing

The game was tested by playing it and checking if each part worked correctly.

* **Snake Movement:**  
  We tested moving the snake in all directions using the keyboard. It moves properly does not let the player go in the opposite direction (like from up to down immediately).
* **Collision Detection:**  
  We made the snake crash into the wall and into itself. The game showed “Game Over” when this happened.
* **Food and Score:**  
  We checked that food always appears in an empty spot and not on the snake. When the snake eats it, the score goes up and a new food appears.
* **Grid Display:**  
  Every time the snake moves, the screen updates properly and shows the snake’s new position.