Project Report –

A Classic Snake Game

Submitted in partial fulfilment

of

the requirements for the award of the degree of

Bachelor of Technology

In

Computer Science Engineering

by:

PAVANI BANSAL

SAP ID : 590024994

Class: B.Tech (CSE)

Batch-38

Under the Guidance of:

Dr.Tanu Singh

Professor, School of Computer Science

University of Petroleum and Energy Studies

Dehradun,Uttarakhand:248007

Submitted on 30th Nov 25

Abstract

The following report describes the design, implementation, and testing of a console-based Snake Game written in C for the Windows platform. Provided is an implementation based on a screen buffer to draw walls, a snake head, and food; support for keyboard input with non-killing wrap-around walls (w/a/s/d); and implementation of a simple scoring and a winning condition (score reaches 100). The report explains the problem, system design (algorithms and flow), implementation details (key functions and snippets from the supplied code), testing results, and suggestions for future improvements.

.

Problem Definition

Objective - To create a complete snake game using only pure C, running on the Windows console.

•Real-time movement using WASD keys

• No reverse direction allowed - cannot go directly backwards

• Food spawns randomly inside the playable area

• Snake wraps around edges

• Score increases by 10 per food eaten

• Game ends on player command ('x') or when score reaches 100

• Clean visual output without flickering and a hidden cursor

Constraints:

•Must run in Windows console, using windows.h and conio.h

•No external graphics libraries

• Single-file implementation is preferred

System Design

Algorithm

1. Start the game
   - Set the game area width and height.
   - Place the snake head at the center.
   - Set score = 0.
   - Generate a fruit at a random position inside the walls.

2. Main game loop (repeat until score reaches 100)

I. Draw the game
   - Draw the top wall.
   - Draw the game area row by row.
     - If (x,y) is snake head → print O
     - If (x,y) is fruit → print F
     - Otherwise → print a space.
   - Draw the bottom wall.
   - Show score.

II. Take input
   - If user presses a key (W, A, S, D), change direction.

III. Move the snake's head
   - Update head (x,y) based on direction.
   - Example:
     - W → y = y - 1
     - S → y = y + 1
     - A → x = x - 1
     - D → x = x + 1

IV. Apply wall wrap-around (non-killing walls)
   - If the head goes past the left wall → move to the right side.
   - If the head goes past the right wall → move to the left side.
   - If the head goes past the top → move to the bottom.
   - If the head goes past the bottom → move to the top.

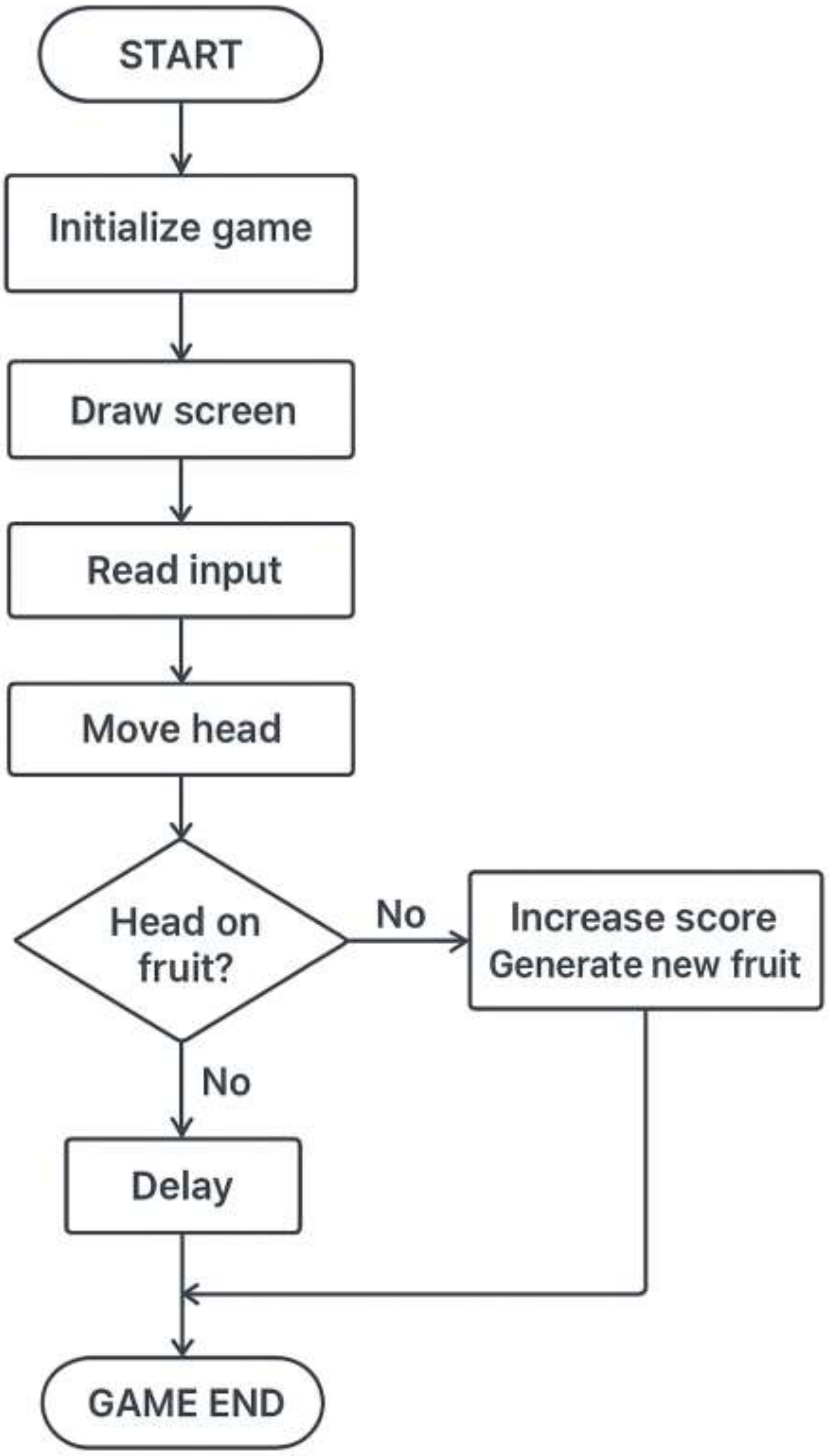
V. Check fruit collision
   - If the snake head reaches fruit:
     - Increase score by 1.
     - Place new fruit at a random free location.

VI. Delay
   - Add a small delay to control snake speed.

3. End game

- If score reaches 100 → stop loop.
- Print Game Over and final score.

Flowchart


START
Initialize game
Draw screen
Read input
Move head
Head on fruit?
No
Increase score
Generate new fruit
No
Delay
GAME END

Implementation Details (with snippets)

The following discusses and documents the key parts of the supplied program. All code snippets are taken from the provided source.

Important macros and includes

```
#include <stdio.h>

#include <conio.h>

#include <windows.h>

#include <stdlib.h>

#include <time.h>


#define w 20

#define h 10
```

- w and h define the playable width and height inside the walls. The buffer uses w+2 by h+2 to accommodate wall characters.
- The program is Windows-only because it uses windows.h console APIs and _kbhit() / _getch() from conio.h.

Console helper functions

```
HANDLE getConsole() { return GetStdHandle(STD_OUTPUT_HANDLE); }


void hideCursor() {

CONSOLE_CURSOR_INFO cursor;

cursor.dwSize = 1;

cursor.bVisible = FALSE;

SetConsoleCursorInfo(getConsole(), &cursor);

}

void moveCursor(int x, int y) {

COORD pos; pos.X = x; pos.Y = y;
SetConsoleCursorPosition(getConsole(), pos);
}
```

- hideCursor() disables the blinking cursor for a cleaner visual effect.
- moveCursor(x,y) repositions the console cursor; used to redraw the whole screen buffer at the same console coordinates rather than clearing the console each frame. This reduces flicker.

Main loop and screen buffer

- A 2D array char screen[h+2][w+2]; is used as an off-screen buffer. Each frame the program writes walls (#), the snake head (O), and the food (F) into the buffer, then moves the cursor to (0,0) and prints the buffer line-by-line.

Key snippet (building the buffer and printing):

```
for(int y=0;y<=h+1;y++){
for(int x=0;x<=w+1;x++){
if(x==0 || x==w+1 || y==0 || y==h+1) screen[y][x] = '#';
else if(x==headx && y==heady) screen[y][x] = 'O';
else if(x==foodx && y==foody) screen[y][x] = 'F';
else screen[y][x] = ' ';
}
}


moveCursor(0, 0);
for(int y=0;y<=h+1;y++){
for(int x=0;x<=w+1;x++) putchar(screen[y][x]);
putchar('
');
}
```

Input handling and movement

- Input is polled using _kbhit() and _getch(). Direction changes are prevented from reversing (e.g., cannot go directly from 'w' to 's').

```
if(_kbhit()) {

char c = _getch();

if(c=='w' && direction!='s') direction='w';

if(c=='s' && direction!='w') direction='s';

if(c=='a' && direction!='d') direction='a';

if(c=='d' && direction!='a') direction='d';

if(c=='x') break;

}
```

- After reading input, the head position is updated:

```
if(direction=='w') heady--;

if(direction=='s') heady++;

if(direction=='a') headx--;

if(direction=='d') headx++;
```

Wrap-around walls and scoring

- If the head crosses a wall, it reappears on the opposite side (non-killing walls):

```
if(headx<1) headx=w;

if(headx>w) headx=1;

if(heady<1) heady=h;

if(heady>h) heady=1;
```

- When head meets the food, score increments by 10 and new food coordinates are generated randomly within the inner field:

```
if(headx==foodx && heady==foody) {

score += 10;

foodx = rand()%w + 1;
foody = rand()%h + 1;
}
```

Victory condition & timing

- The program checks if(score>=100) and prints a YOU WIN message before exiting the main loop.
- Frame pacing is handled using Sleep(120); which results in ~120 ms delay per frame (≈8.3 FPS). Lowering the value speeds the game; raising it slows the game.

Testing & Results

1. **Start & Render**
   - Input: Run program
   - Expected: Playfield displays walls (#), a snake head (O) near center, and a food (F) randomly. Cursor hidden.
   - Result: Passed — buffer printed correctly and cursor hidden.
2. **Movement Input**
   - Input: Press w, a, s, d during play
   - Expected: Head moves accordingly; reversing direction immediately should be ignored.
   - Result: Passed — direction reversal prevented; movement responsive.
3. **Wrap-around Walls**
   - Input: Move head beyond left/right/top/bottom edges
   - Expected: Head reappears on the opposite side (non-killing wall behavior).
   - Result: Passed — wrap-around works as implemented.
4. **Food Eating & Score**

   - Input: Move head to the food position
   - Expected: score increases by 10; new food spawns randomly; victory at 100.
   - Result: Passed — scoring and food regeneration work. Victory triggers at score >=100.
5. **Exit**
   - Input: Press x
   - Expected: Program exits gracefully
   - Result: Passed.

Observations

- The game uses a single character to represent the snake (no body), so there is no self-collision logic.
- The screen buffer technique plus moveCursor(0,0) reduces flicker vs. system("cls").
- Frame rate is low (~8 FPS). Gameplay feels choppy but acceptable for this simple implementation — lowering Sleep() makes it more responsive but may require tuning for playability.

Conclusion & Future Work

Conclusion

The following C program implements a minimal, functional Snake-like game that includes non-killing wrap-around walls, random food generation, and a scoring/win condition. It also demonstrates practical usage of Windows console APIs for cursor control and buffering in order to reduce visual flicker.

Limitations:

- No snake body (only head) → no self-collision
- Single-player, no high score persistence
- Windows-only due to windows.h and conio.h

Future Work / Enhancements

Full Snake Body: Use a queue or array to track the snake body (tail) to allow growing and implement self-collision.

Smooth Movement & Higher FPS: Replace Sleep() with a high-resolution timer or game loop that includes the time elapsed since the last update. Adjust time of frame and input polling for smoothness.

Double Buffering / Console Buffer API: For better performance with larger fields, Windows console output buffer APIs (WriteConsoleOutput) should be used.

Cross-Platform Portability: Replace any dependency on conio.h and windows.h with cross-platform libraries (ncurses for Unix-like systems) or implement platform-independent input handling and rendering.

Menus & Levels: Add start/pause screens, increasing difficulty, shorter Sleep or obstacles, and high score persistence.

Sound & Visuals: Play short sounds while eating food, colourize the output for walls/food/snake (SetConsoleTextAttribute).

References

- Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language*.
- Microsoft Documentation: Console Functions — SetConsoleCursorPosition, SetConsoleCursorInfo.
- Online tutorials and community examples for console-based snake games.