Assignment #2

Submitted by:
Asad Tauqeer

Roll No:
23021519-160

Class:
BS-CS-23-V

Section:
B

Course Code:
CS-303

Course Title:
Advance Programming Techniques

Submitted to:
Mr. Ehtisham Rasheed

Department of Computer Science

UNIVERSITY OF GUJRAT

A WORLD CLASS UNIVERSITY

Snake Game - Design & Technical Documentation

1. Game Overview

This is a modern iteration of the classic Snake arcade game built using **C# Windows Forms**. The game features progressive difficulty, a dynamic food system with status effects, directional sprite rendering, and persistent high scores. The objective is to navigate the snake to eat food, grow in length, and achieve the highest possible score without colliding with walls, obstacles, or the snake's own tail.

2. Gameplay Mechanics & Design Choices

A. Gameplay Mechanics

The core objective is to navigate the snake to consume food items while avoiding collisions with walls, the snake's own body, and screen boundaries.

- **Controls:**
    - **Movement:** Arrow Keys (Up, Down, Left, Right)
    - **System:** Enter (Start/Restart), P or Space (Pause/Resume)
- **Scoring:**
    - **Base Score:** +10 points per normal food.
    - **High Score:** Automatically saved to and loaded from highscore.txt.

B. Progressive Difficulty (Leveling)

The game implements a **Threshold-Based Leveling System**.

- **Logic:** Every **50 points**, the difficulty level increases.
- **Effect:** The game speed increases (timer interval decreases by 5ms), and new obstacles (Rocks) are randomly generated on the map.
- **Design Choice:** Using a threshold (e.g., Score >= nextLevelThreshold) instead of a modulo operator ensures that if a user eats a 50-point bonus and skips a specific number (e.g., jumping from 40 to 90), the level-up event still triggers reliably.

C. Food System & Power Ups

The game employs a dual-layer food generation system, instead of a single food item, where items spawn randomly with distinct effects:

Food Type	Color / Icon	Effect	Score Impact
Normal	**Red / Apple**	**Standard growth.**	**+10**
Fast	**Orange**	**Increases game speed (harder).**	**+10**

Food Type	Color / Icon	Effect	Score Impact
Slow	**Blue / Eggplant**	**Decreases game speed (easier).**	**+10**
Bonus	**Gold / Lemon**	**No speed change, high reward.**	**+50**

D. Visual Rendering (Directional Sprites)

To move away from simple geometric shapes, the game uses a **Sprite-Based Rendering System**.

- **Head & Tail:** The code detects the current direction (Up, Down, Left, Right) to render the correct facing image (head-up.png, tail-left.png, etc.).
- **Body Segments:** The render loop calculates the relationship between the previous and next body segments to decide whether to draw a **Horizontal** or **Vertical** body texture.

3. User Interface (UI) Design

The Grid System

- **Resolution:** The game board uses a virtual grid where each cell is 24x24 pixels (Settings.Width = 24).
- **Collision:** All logic (eating, dying, hitting walls) is calculated using these grid coordinates (X, Y) rather than raw pixels. This simplifies collision detection to simple integer comparisons.

Game Canvas (Left/Center)

This is the main canvas where different game states appear.

- **Visuals:** Has a textured background (sand).
- **States:** Three states:
    - **Start State:** Title and instructions box overlay.
    - **Playing State:** The main game.
    - **Pause State:** An overlay display, pausing the game.
- **Audio:** Plays a looping intro track to set the mood, which automatically stops when the game begins. A food-eating and a dying sound played on a specific occasion.

HUD (Heads-Up Display)

During gameplay, essential information is displayed on the side panel to avoid obstructing the view:

- **Score & High Score:** Real-time feedback.
- **Difficulty Level:** Shows current progression.
- **Effect Status:** A dedicated label (lblEffect) momentarily displays text like "Speed Up!" or "+50 Points!" in color-coded text (Red, Blue, Gold) to provide immediate feedback on the item consumed.

4. Technical Architecture

Core Classes

- **Form1 (Game Engine):** Handles the main game loop, asset loading, collision detection (Die, Eat), and rendering (pbCanvas_Paint). It manages the List<Circle> collections for the Snake body and Walls.
- **Settings (Global State):** A static class maintaining global configurations such as Width, Height, Speed, Score, and GameState. This separates data from logic, allowing easy access across methods.
- **Input (Control Handler):** Uses a Hashtable to track key states. This design choice prevents "input ghosting" and ensures smoother control responsiveness compared to standard WinForms key events.
- **Circle (Entity Data):** A simple data structure representing coordinate points (X, Y) for the snake segments, walls, and food items.

The Game Loop

- **Timer:** A System.Windows.Forms.Timer drives the game loop.
- **Tick Event:** On every tick, the game:
    1. Updates the snake's position based on direction.
    2. Checks collisions (Walls, Body, Food).
    3. Updates game state (Level up, Score).
    4. Calls pbCanvas.Invalidate() to force a repaint of the graphics.
- **Asset Management:** The LoadAssets method utilizes robust error handling (try-catch blocks) to attempt loading external images and sounds (.wav). If files are missing, the game gracefully degrades to using colored geometric shapes (ellipses/rectangles) instead of crashing.

5. Key Design Choices

- **Dynamic Difficulty Scaling**: Instead of a static speed, the game implements a nextLevelThreshold. When the score crosses this threshold, difficultyLevel increments, reducing the timer interval (making the snake faster) and adding random obstacles (Walls) to the map.
- **Collision Detection Algorithm:** The MovePlayer method performs sequential checks:
    1. **Border Check:** Verifies coordinates are within maxWidth/maxHeight.
    2. **Self-Collision:** Iterates through the body list to check for coordinate overlap.
    3. **Obstacle Collision:** Checks against the Walls list.
- **State Management:** The GameState enum (Start, Playing, GameOver) strictly controls what is rendered and what inputs are valid, preventing logic errors like moving the snake while on the "Game Over" screen.