PACMAN GAME APPLICATION

UCS2265 – Fundamentals and Practice of Software Development

A PROJECT REPORT

Submitted By

Thirumurugan RA - 3122 23 5001 149

Varun Prakash - 3122 23 5001 154

Vishal Muralidharan - 3122 23 5001 162

Department of Computer Science and Engineering

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

Kalavakkam – 603110

June 2024

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this project report titled "**PACMAN GAME APPLICATION**" is the bonafide work of Thirumurugan RA (3122235001149), Varun Prakash (3122235001154) and Vishal Muralidharan (3122235001162) who carried out the project work in the UCS2265 – Fundamentals and Practice of Software Development during the academic year 2023-24.

Internal Examiner External Examiner

Date:

TABLE OF CONTENTS

Abstract:

This project presents a modern implementation of the classic arcade game PACMAN, developed using the C programming language and the Raylib library for the graphical interface.

The objective is to recreate the iconic PACMAN gameplay, where players navigate a maze, collect pellets, avoid ghosts, and aim to achieve high scores.

The project leverages the efficiency and control provided by C for game logic and the simplicity and power of Raylib for rendering graphics and handling user input.

Key features include ghost AI, responsive controls, power-ups, and score tracking, all designed to deliver an engaging and nostalgic gaming experience.

This project serves as both a tribute to a timeless classic and a practical exercise in game development using contemporary programming tools.

Acknowledgements:

We would like to thank God for being able to complete this Project Work with success.

We are thankful to and fortunate enough to get constant encouragement, support, and guidance from our project guides Dr. R. Priyadharsini, M.E., Ph.D., Associate Professor, Computer Science and Engineering Department, Dr. Chitra Babu, M.E., M.S., Ph.D., Professor, Computer Science and Engineering Department and Dr. R. Sagaya Milton M.E., Ph.D., Professor, Computer Science and Engineering Department who helped us in successfully completing our Project Work.

On this mean time, we would like to thank our Parents who gave this wonderful opportunity to be in SSN.

We respect and thank our management for providing us an opportunity to do the project work in SSN College of Engineering and giving us all support and facilities, which helped us completing the project work.

We take this opportunity to express our profound gratitude and deep regards to Dr. T.T Mirnalinee M.E., Ph.D., HoD of Computer Science and Engineering Department for her exemplary guidance, monitoring, and constant encouragement throughout the course of this project.

1. Introduction:

Our PACMAN project, an exciting venture that combines the classic gameplay of PACMAN with modern programming techniques.

This project is implemented in the C programming language and utilizes the Raylib library for the graphical interface. Our PACMAN project is a faithful recreation of the classic arcade game.

Players navigate PACMAN through a maze, collecting pellets, avoiding ghosts, and aiming to clear the board. The project includes all the essential elements of the original game, including the four unique ghosts and power pellets.

2. Problem Statement:

Develop a PACMAN Game Application in which the user is allowed to control Pacman, a hungry yellow character, navigating through a maze, gobbling up pellets while avoiding the ghosts. The user plays in such a way that the Pacman consumes all the pellets scattered throughout the maze without being captured by ghosts.

Input

- Instructions to users who play the game.
- The player can use keyboard inputs to control Pacman's movement.

Output

- Display of the maze with Pacman, pellets, ghosts, and walls.
- Current score and lives remaining
- Messages of winning or losing the game

3. Further Exploration of the Problem Statement:

1. **Scoring System:** The scoring system has been changed in our game to help motivate the players. For example, the points awarded for eating a ghost increase for each consecutive ghost eaten while a single power pellet is active.
    a. We double the points for each consecutive ghost is eaten while a single power pellet is active (that is, 9 seconds)

2. **Sound Effects and Music:** The game includes sound effects and music to enhance the gaming experience. For example, a sound could play in the loading screen, when Pacman eats a pellet and when Pacman eats a super pellet, etc.

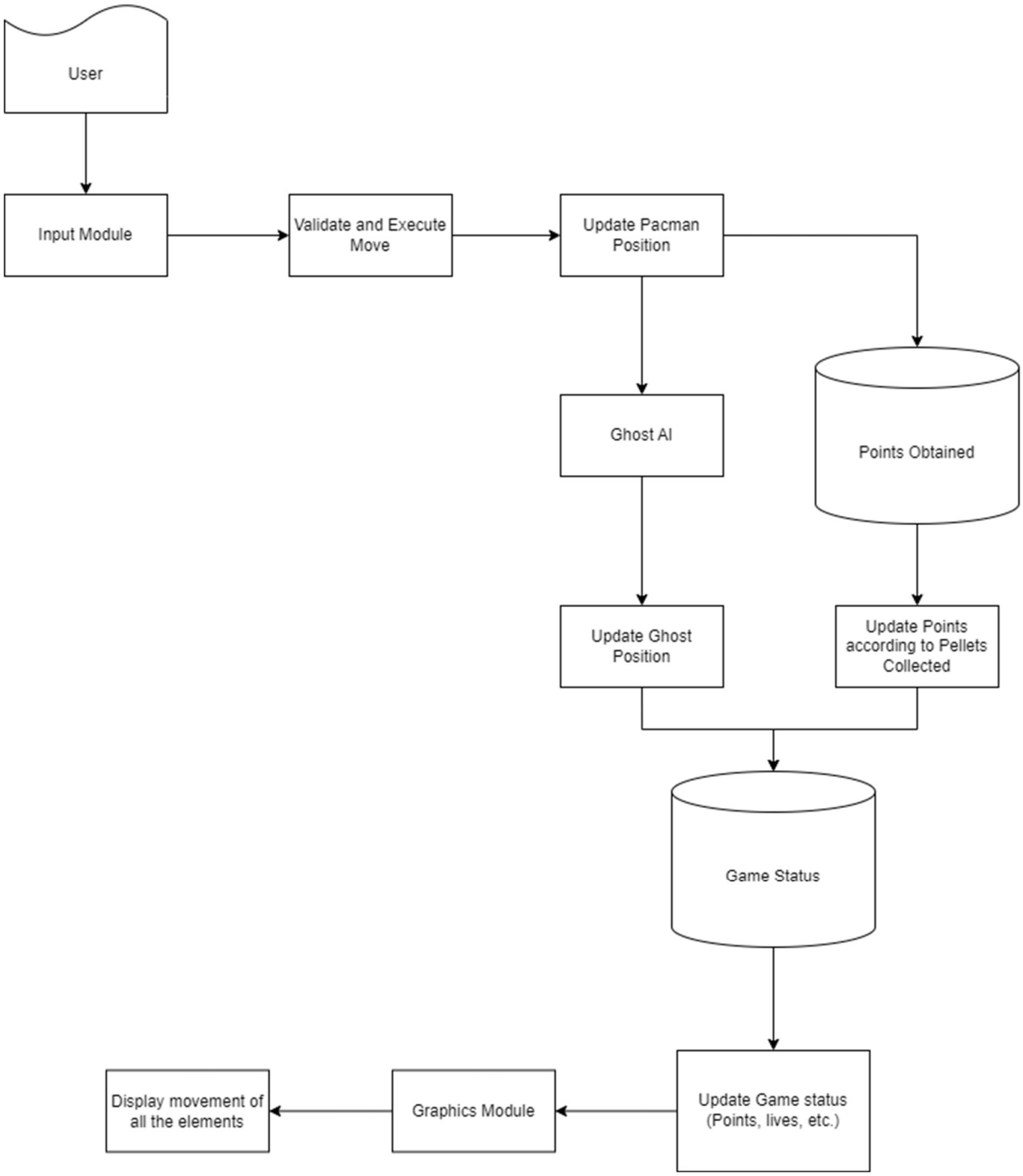
3. **Graphics:** The game includes animations for Pacman and the ghosts, and the when the Pacman dies, a death animation takes place.

4. **User Interface:** The game includes a user interface that displays the ghost information, points allotted to different pellets, current score, high score, the number of lives remaining.

5. **Difficulty Selection:** The game includes 3 difficulty levels, that is Easy, Medium, and Hard. These levels can be selected by the Player and the ghosts' speed increases in increasing hardness.
    a. The high scores are kept separate for the different difficulty levels.

6. **Ghost AI:** Each ghost targets the Pacman differently, allowing the ghosts to use different strategies to try to catch Pacman, such as trying to corner him or cut off his escape routes.

4. Architecture Diagram:



Explanation:

An architecture diagram is a visual representation that depicts the high-level structure of a system. It provides an overview of a system and its components, and how those components interact with each other.

The User first inputs the moves using the keyboard into the Input module. Then the move is validated and then executed. Then accordingly Pacman's position is

updated. Then the movement data of Pacman is given as input to the Ghost AI Database and the Points Database which store this data. They update the Ghosts' Position and the Points (according to the number and kind of pellets collected) respectively.

Then all this is given as input to the Game Status Database which stores these data. Then updates Game status such as points, remaining lives, etc.

All these movements of the Pacman and Ghosts is displayed on the screen using the Graphics Module (Raylib Library).

5. Structured Analysis Using Data Flow Diagram:

a. Level 0 DFD:

Explanation:

The user refers to the player interacting with the Pacman game interface. This game is a one-player type. As the user opens the game file, he is directly brought to the graphical user interface via the C program.

b. Level 1 DFD:

Explanation:

The user moves which is stored in the data cache. The program then has a module which validates the move, another module in order to update the position and one more that updates the game score. The game state is then established, and the next move may be inputted by the player.

c. Level 2 DFD:

Explanation:

The module which validates the move, takes the move from the user as input and first checks if the Pacman is still alive and then it checks if the Pacman collides with the wall, if not then it sends the valid move to the next module.

The module which updates the move, gets the output from Validate Move Module as input, that is the valid move then checks the direction in which Pacman should move, then increments the move counter in the given direction, and then finally moves the Pacman in the given direction.

The module which updates the score, gets the output from Update Position Module as input, then checks if the Pacman is in contact with pellets or super pellets, and then checks is Pacman has collected enough super pellets to eat ghosts and if yes, then it updates the control such that Pacman is allowed to consume ghosts and rewards points accordingly to the number of ghosts consumed. Then it updates the score by recording the number of pellets and super pellets consumed and the points are calculated and rewarded accordingly. Then the new score is displayed finally.

6. Structure Charts derived from Data Flow Diagrams:

Structure Chart partitions the system into black boxes (functionality of the system is known to the users, but inner details are unknown).

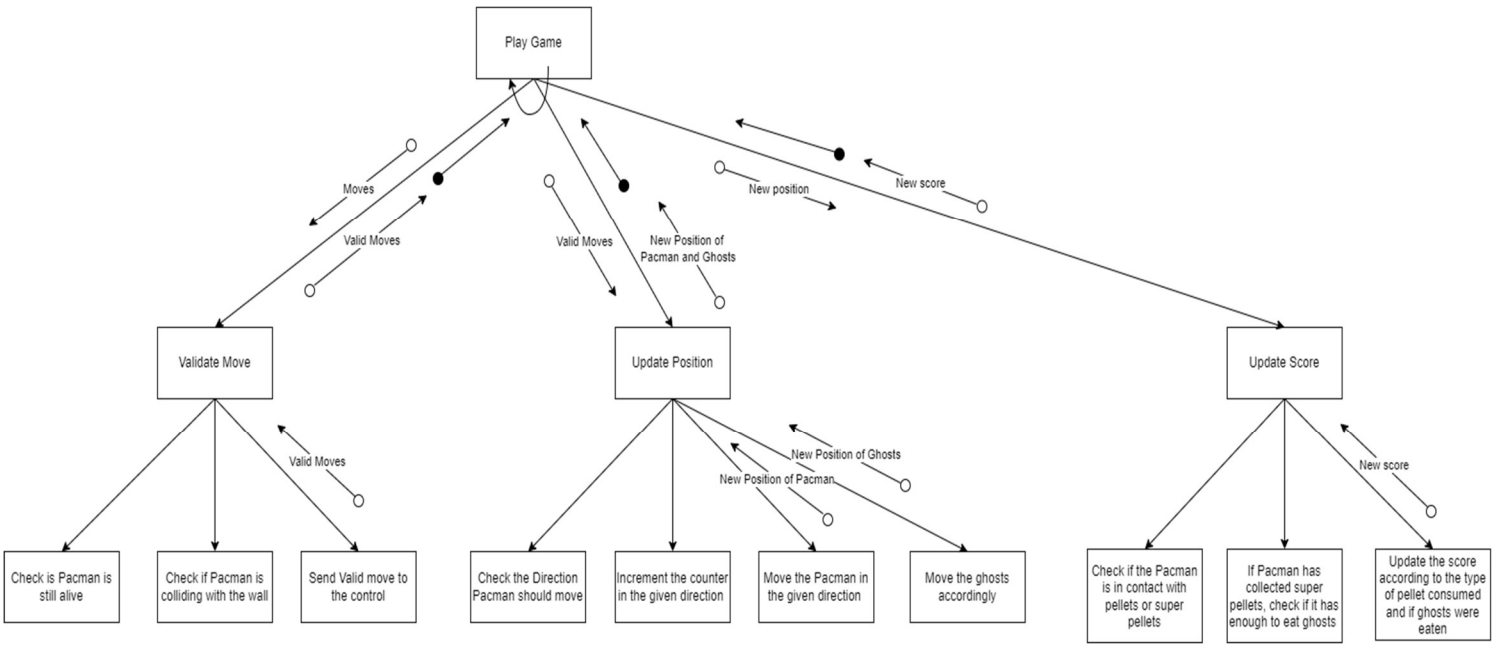
a. Inputs are given to the black boxes and appropriate outputs are generated.
b. Modules at the top level are called modules at low level.
c. Components are read from top to bottom and left to right.
d. When a module calls another, it views the called module as a black box, passing the required parameters and receiving results.

There are 2 ways of converting a DFD into a structure chart: Transform Analysis, and Transaction Analysis.

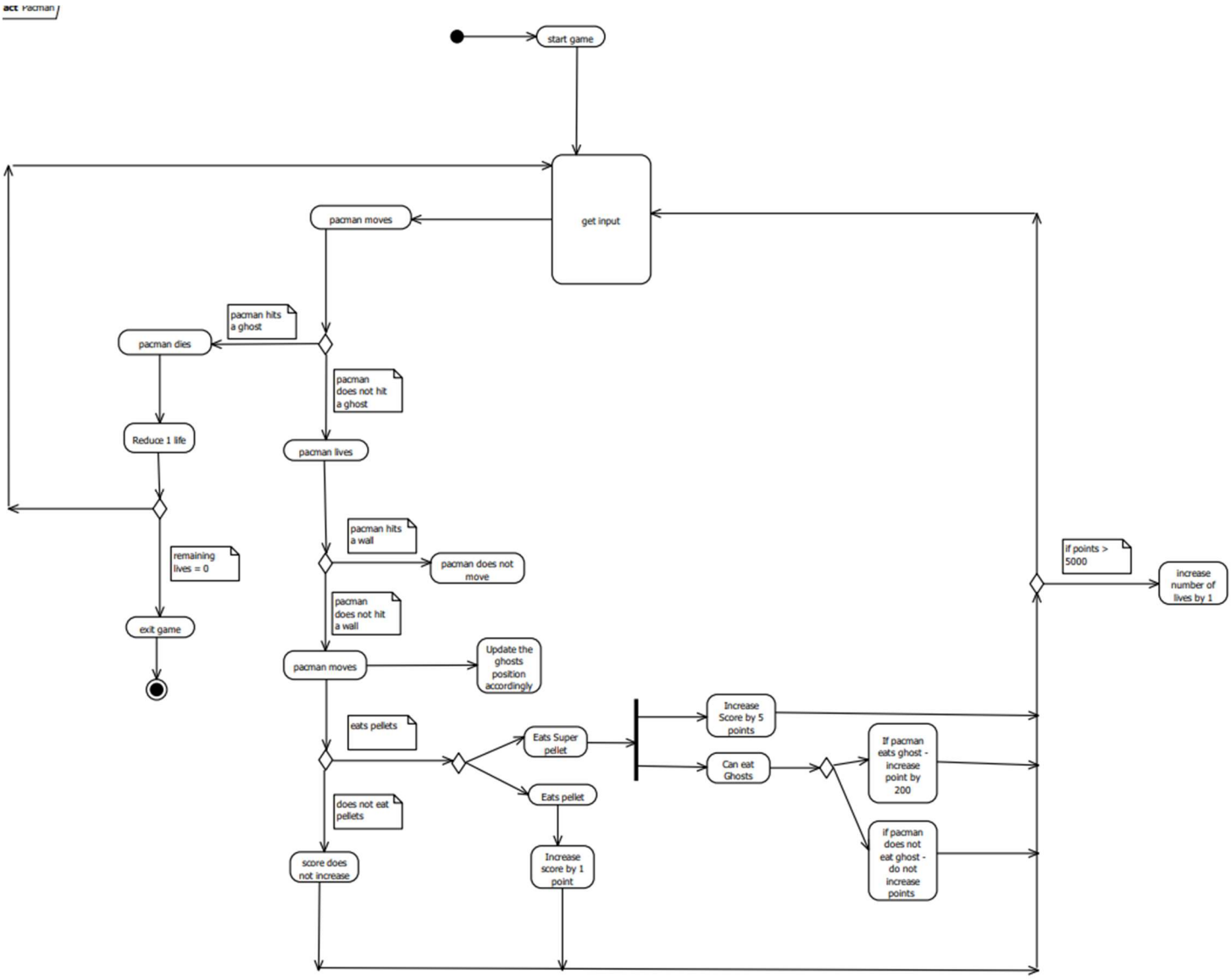
Transform Analysis is suited for systems with very few options, where a single type of data processing occurs, while Transaction Analysis is suited for systems where there are many possible options for the user to choose.

Since in our project, the user only gives input moves for the Pacman through the keyboard, we thought Transform Analysis was the best.

The main Play Game module is split into 3 modules, one for validating the moves, one for updating the positions, and one for updating score.



7. Structured Analysis Using Activity Diagram:



Explanation:

An activity diagram is essentially an elaborate control flow representation of the program. In a game such as Pacman, such a diagram better states the functionality as compared to a data flow diagram.

As displayed in the diagram, it contains all the sequences and possible outcomes of each move carried out by the player - whether he encounters a maze wall, collides with a ghost, the effects when he has eaten super pellet, etc.

If player gets hit by a ghost without having eaten a super pellet, the number of lives is reduced by one and the game restarts, until player has lost three lives after which the game concludes.

Score is kept track of based on Pacman's consumption of pellets and ghosts when powered up; when a score of 5000 is reached, Pacman gains back a life.

The game ends with a win when all pellets and super pellets present on the game screen are consumed by Pacman.

Key Components:

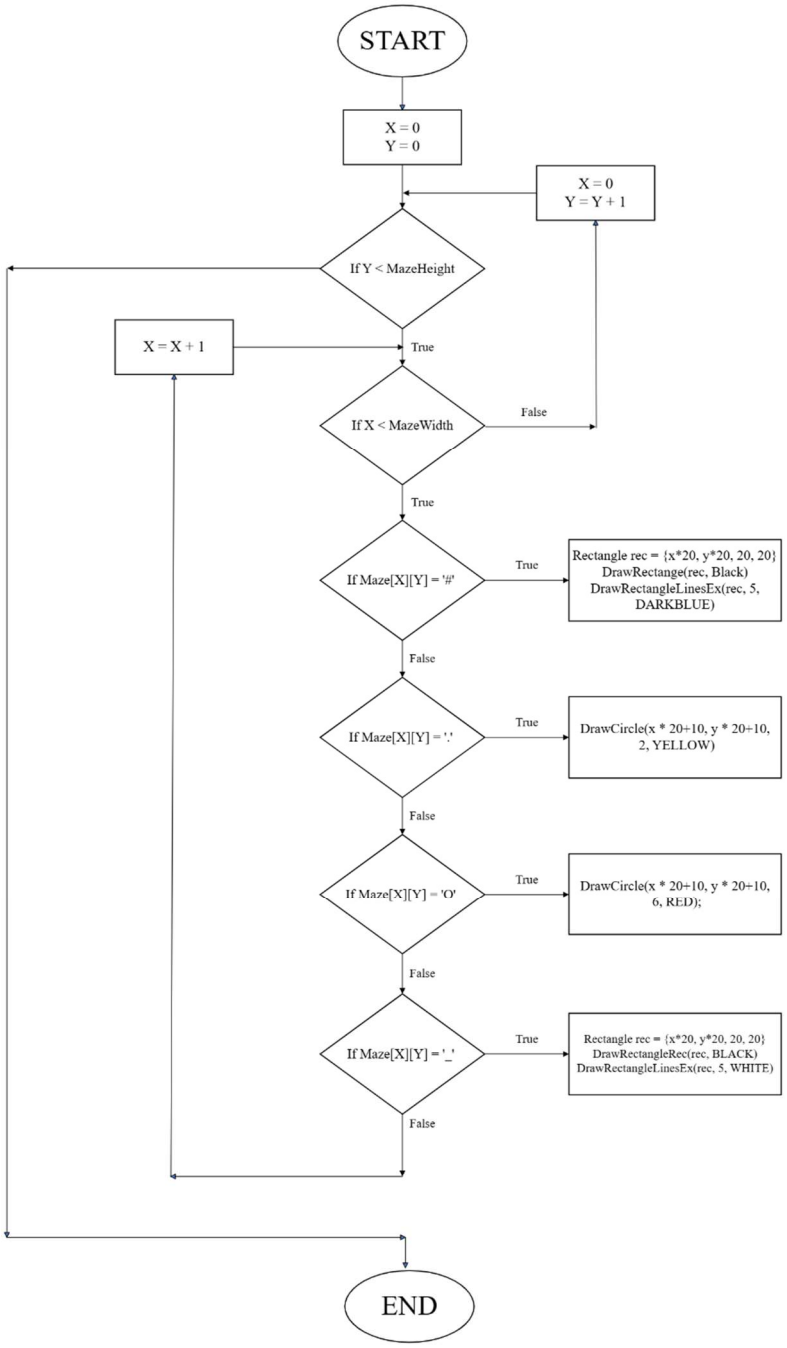
1. **Player Actions:**
   - Movement: The player controls Pac-Man, moving him up, down, left, or right through the maze.
   - Pellet Consumption: Pac-Man eats small pellets as he moves over them.
   - Super Pellet Consumption: When Pac-Man eats a super pellet, he gains the ability to eat ghosts temporarily.

2. **Game States and Transitions:**
   - Normal State: The default state where Pac-Man navigates the maze, eats pellets, and avoids ghosts.
   - Powered-Up State: Triggered when Pac-Man eats a super pellet, allowing him to eat ghosts for a limited time.
   - Life Lost State: When Pac-Man collides with a ghost without being powered-up, he loses a life.
   - Game Over State: Reached when Pac-Man loses all his lives.
   - Win State: Achieved when Pac-Man eats all pellets and super pellets in the maze.

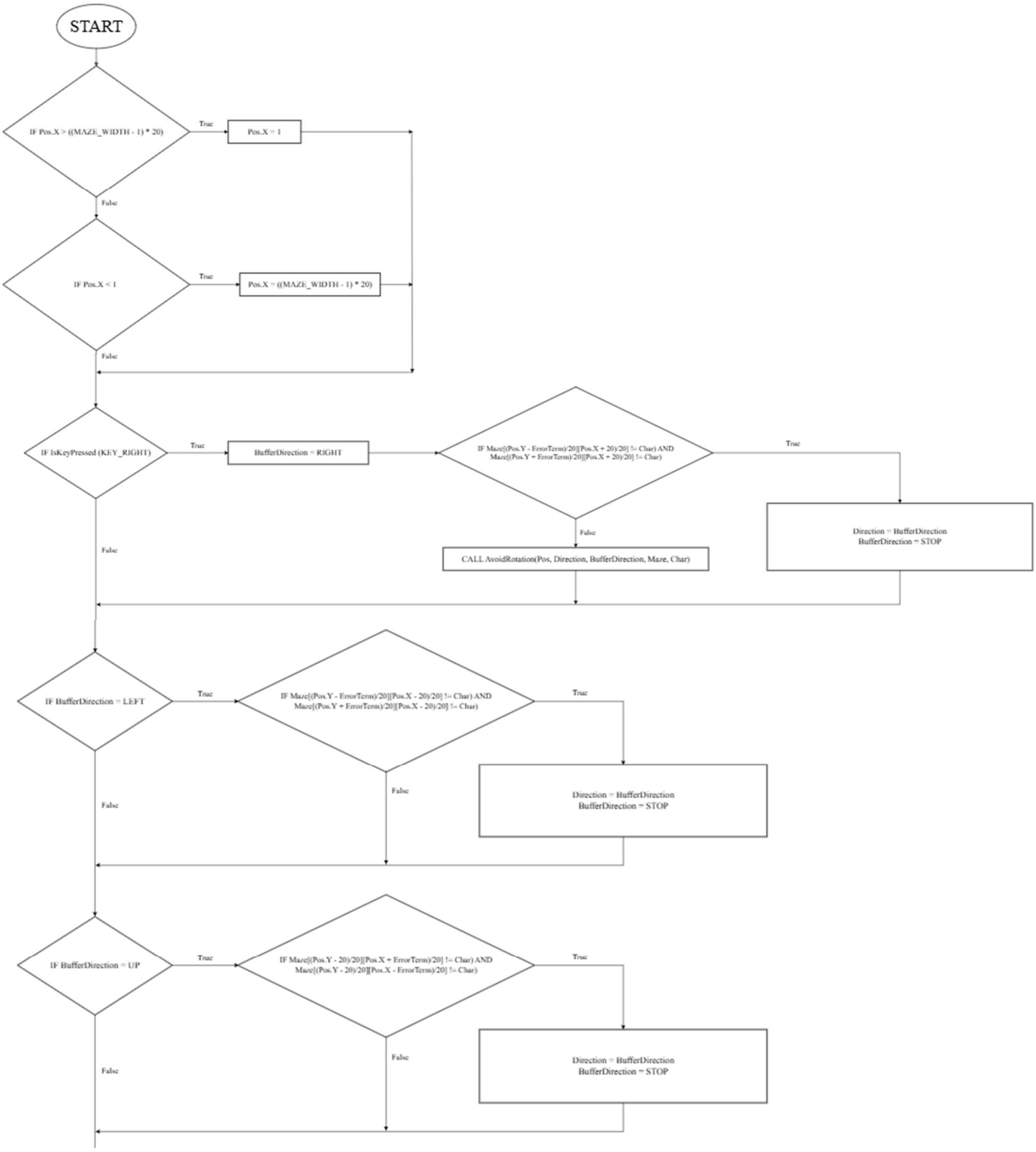
3. **Scoring and Lives Management:**
   - Pellet Score: Points awarded for each pellet eaten.
   - Ghost Score: Extra points awarded for each ghost eaten when powered-up.
   - Extra Life: An extra life is awarded when the score reaches 5000 points.
   - Lives Counter: Tracks the number of lives remaining. The game restarts if lives remain after Pac-Man is hit by a ghost.

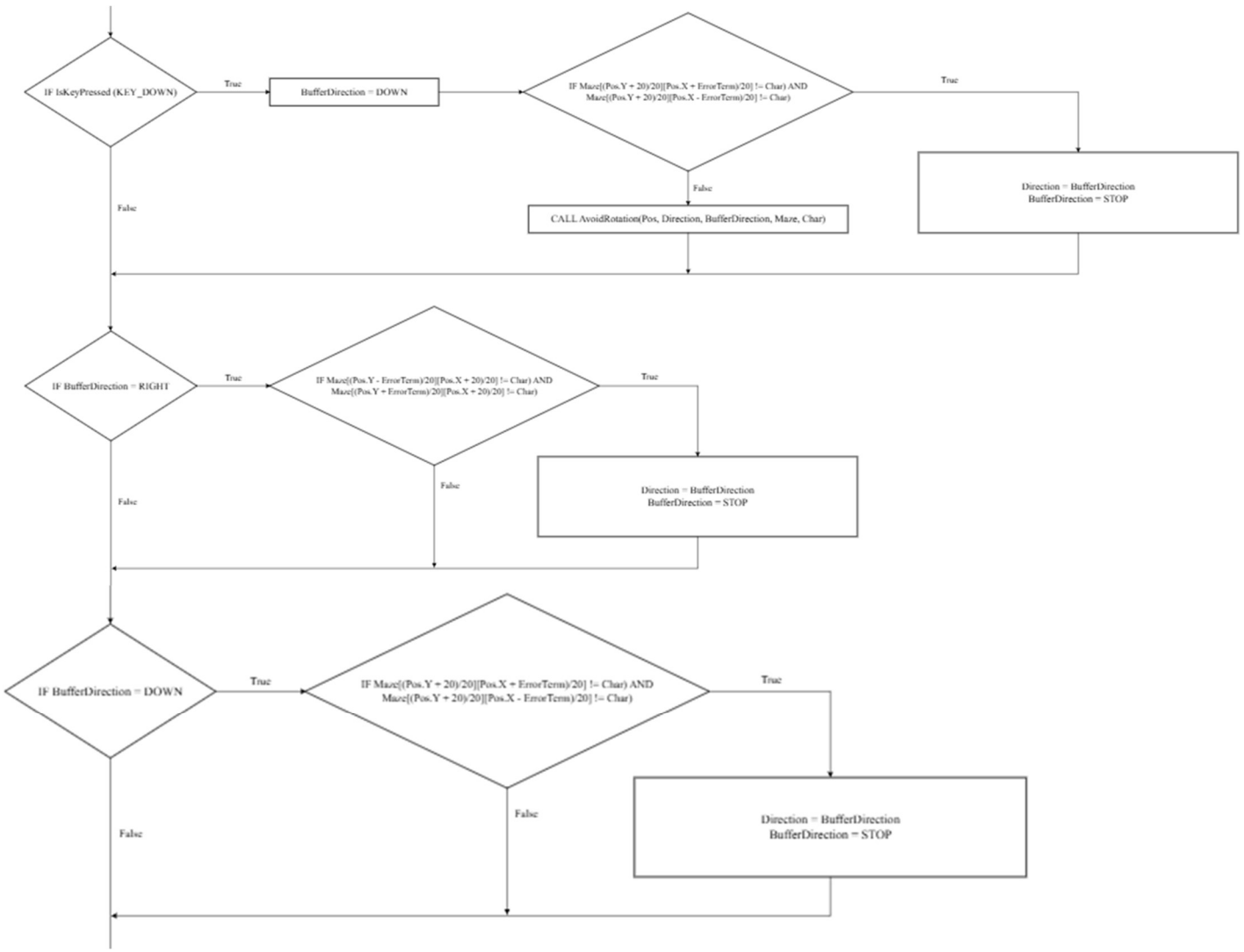
8. Functions Flowcharts:

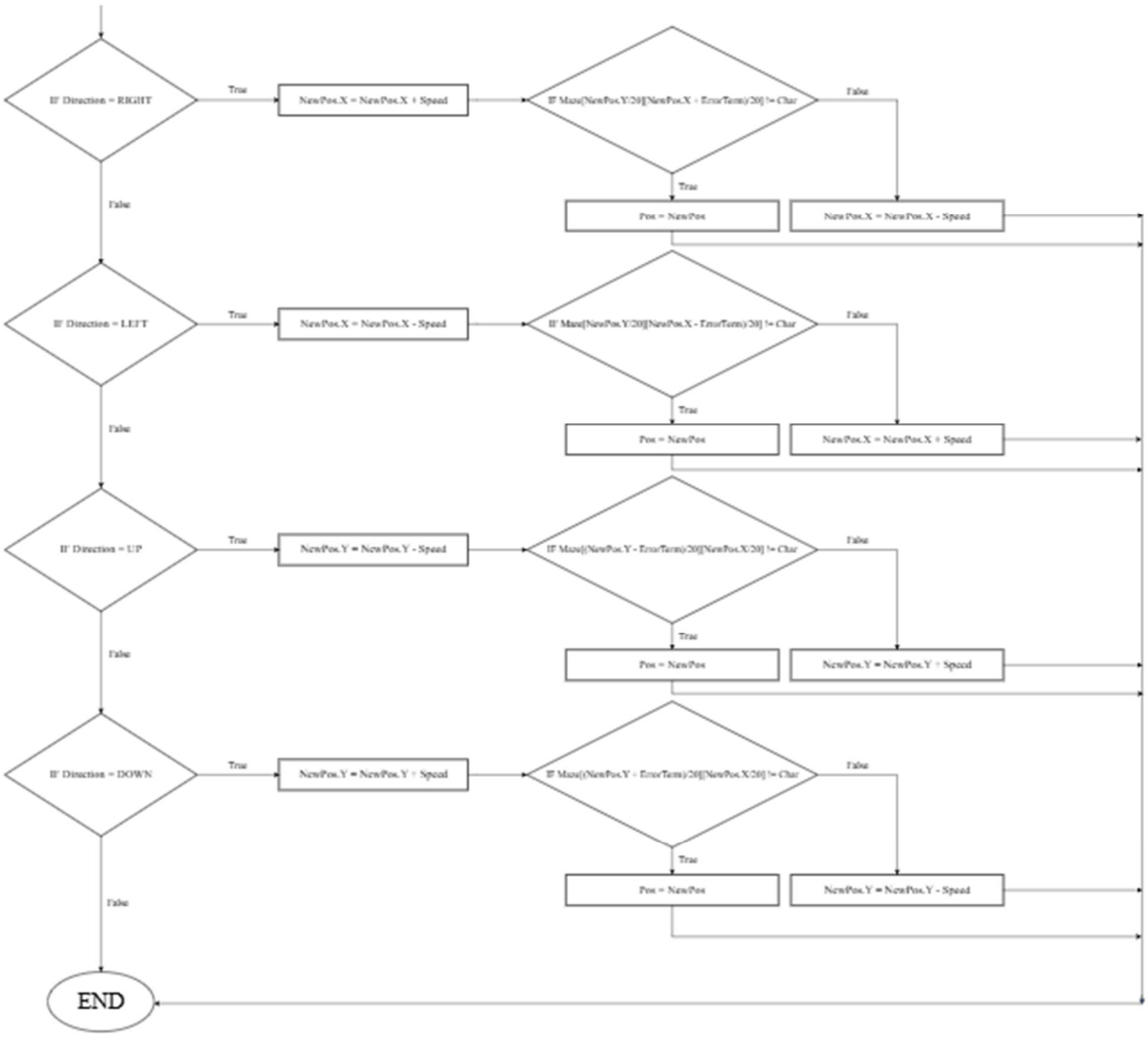
`void drawMaze(char maze[MAZE_HEIGHT][MAZE_WIDTH])`



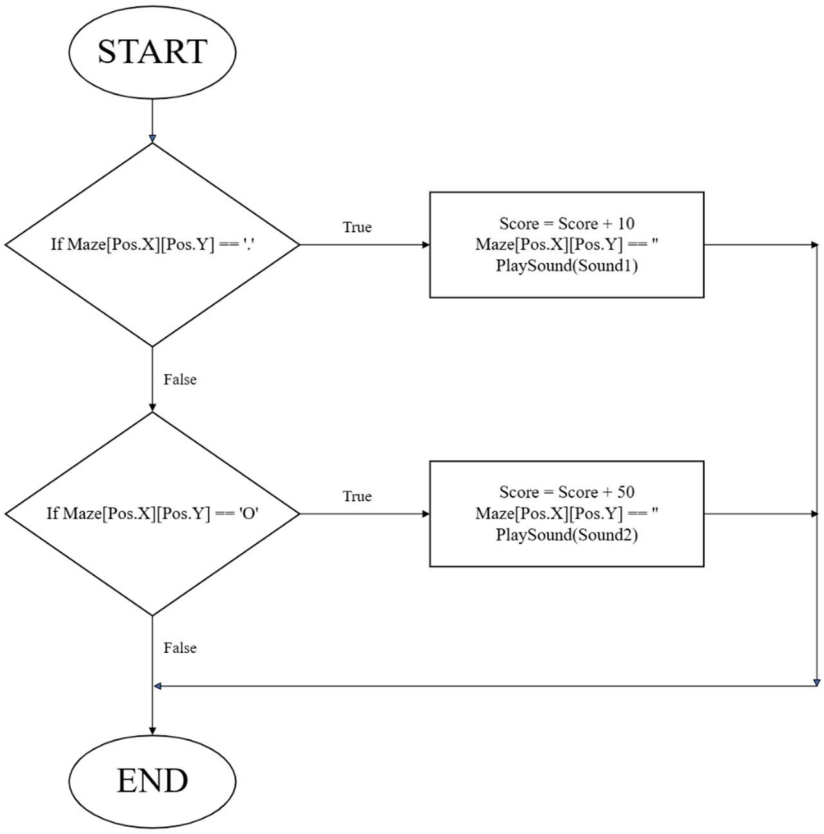
void updatePosition(Vector2 *pos, Direction *direction, Direction *buffer_direction, char maze[MAZE_HEIGHT][MAZE_WIDTH], char Char)



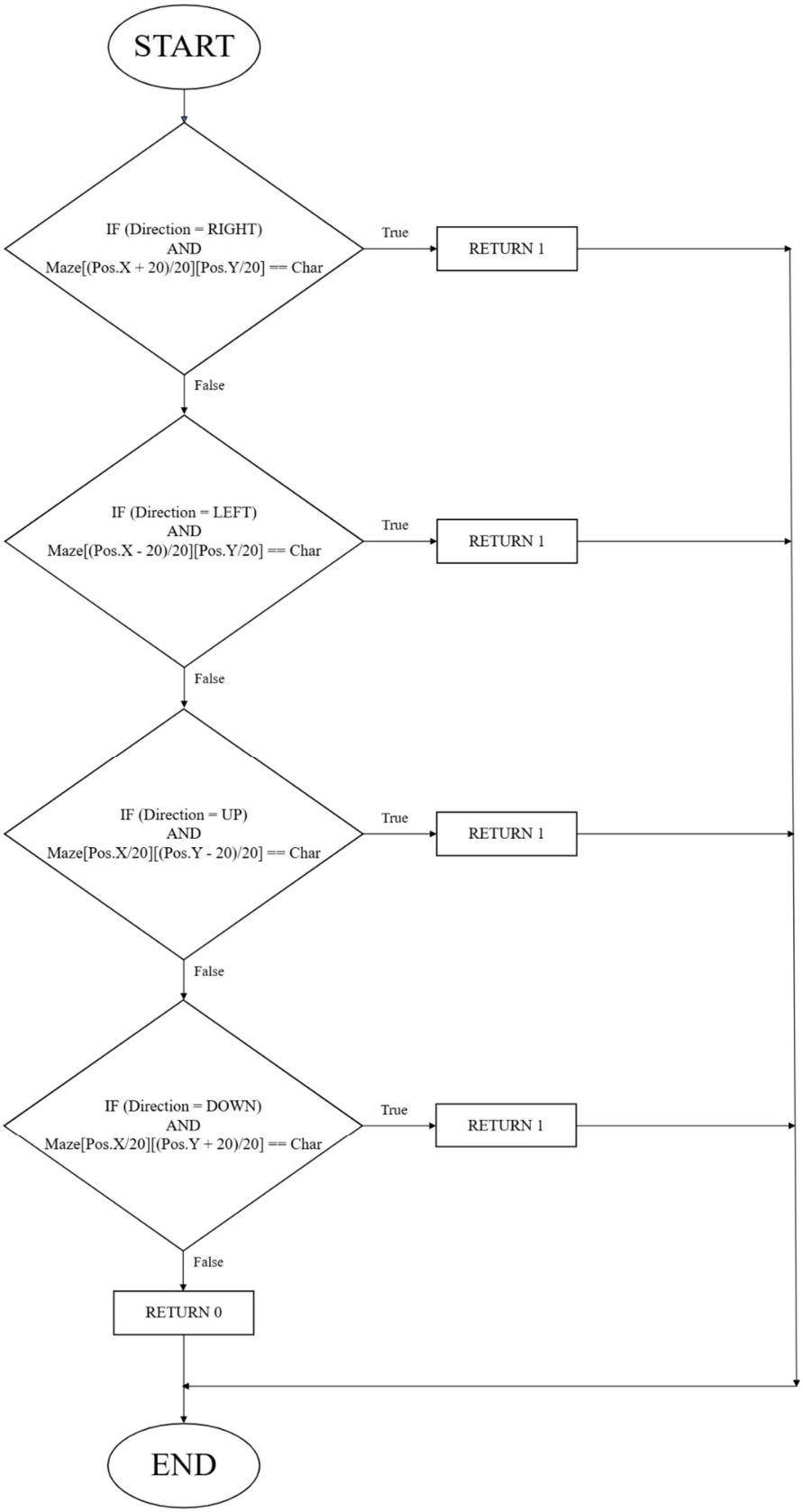

IF IsKeyPressed (KEY_DOWN)
True
BufferDirection = DOWN
IF Maze[(Pos.Y + 20)/20][Pos.X + ErrorTerm)/20] != Char) AND
Maze[(Pos.Y + 20)/20][Pos.X - ErrorTerm)/20] != Char)
True
Direction = BufferDirection
BufferDirection = STOP
False
CALL AvoidRotation(Pos, Direction, BufferDirection, Maze, Char)
False
IF BufferDirection = RIGHT
True
IF Maze[(Pos.Y - ErrorTerm)/20][Pos.X + 20)/20] != Char) AND
Maze[(Pos.Y + ErrorTerm)/20][Pos.X + 20)/20] != Char)
True
Direction = BufferDirection
BufferDirection = STOP
False
False
IF BufferDirection = DOWN
True
IF Maze[(Pos.Y + 20)/20][Pos.X + ErrorTerm)/20] != Char) AND
Maze[(Pos.Y + 20)/20][Pos.X - ErrorTerm)/20] != Char)
True
Direction = BufferDirection
BufferDirection = STOP
False
False


If Direction = RIGHT
True
NewPos.X = NewPos.X + Speed
If Maze[NewPos.Y/20][NewPos.X + ErrorTerm)/20] != Char
False
True
Pos = NewPos
NewPos.X = NewPos.X - Speed
False
If Direction = LEFT
True
NewPos.X = NewPos.X - Speed
If Maze[NewPos.Y/20][NewPos.X - ErrorTerm)/20] != Char
False
True
Pos = NewPos
NewPos.X = NewPos.X + Speed
False
If Direction = UP
True
NewPos.Y = NewPos.Y - Speed
If Maze[(NewPos.Y - ErrorTerm)/20][NewPos.X/20] != Char
False
True
Pos = NewPos
NewPos.Y = NewPos.Y + Speed
False
If Direction = DOWN
True
NewPos.Y = NewPos.Y + Speed
If Maze[(NewPos.Y + ErrorTerm)/20][NewPos.X/20] != Char
False
True
Pos = NewPos
NewPos.Y = NewPos.Y - Speed
False
END

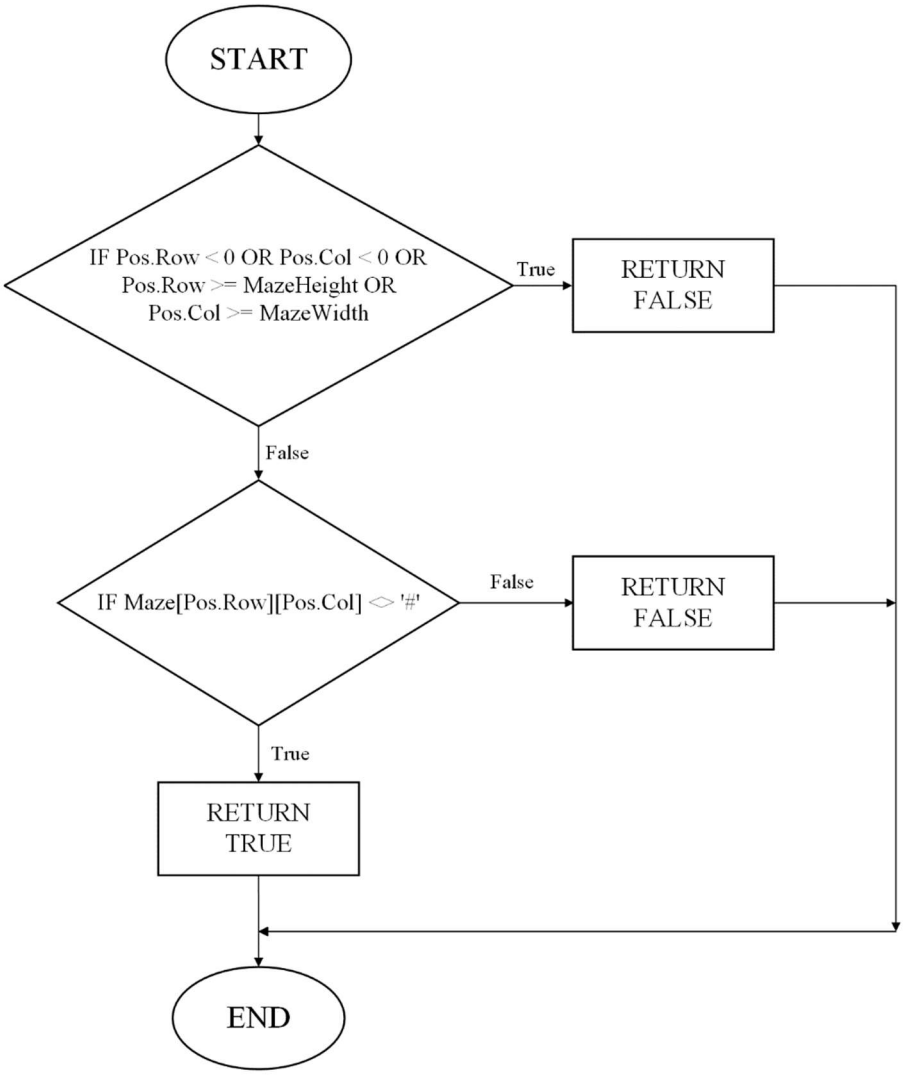
void checkCollisionWithPellets(Vector2 pos, char maze[MAZE_HEIGHT][MAZE_WIDTH], int *score, Sound *sound1, Sound *sup_siren)



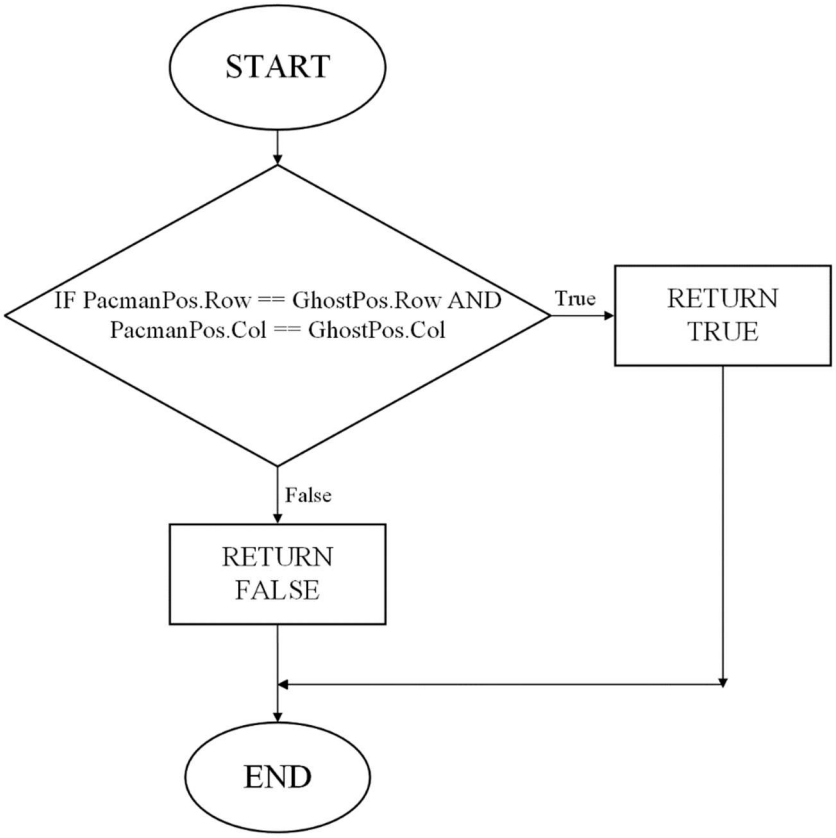
int avoidRotation(Vector2 *pos, Direction *direction,Direction *buffer_direction, char maze[MAZE_HEIGHT][MAZE_WIDTH], char Char)



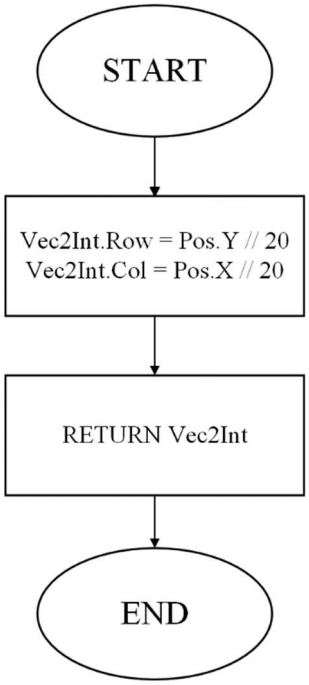
bool isValidPosition(Vector2Int position)



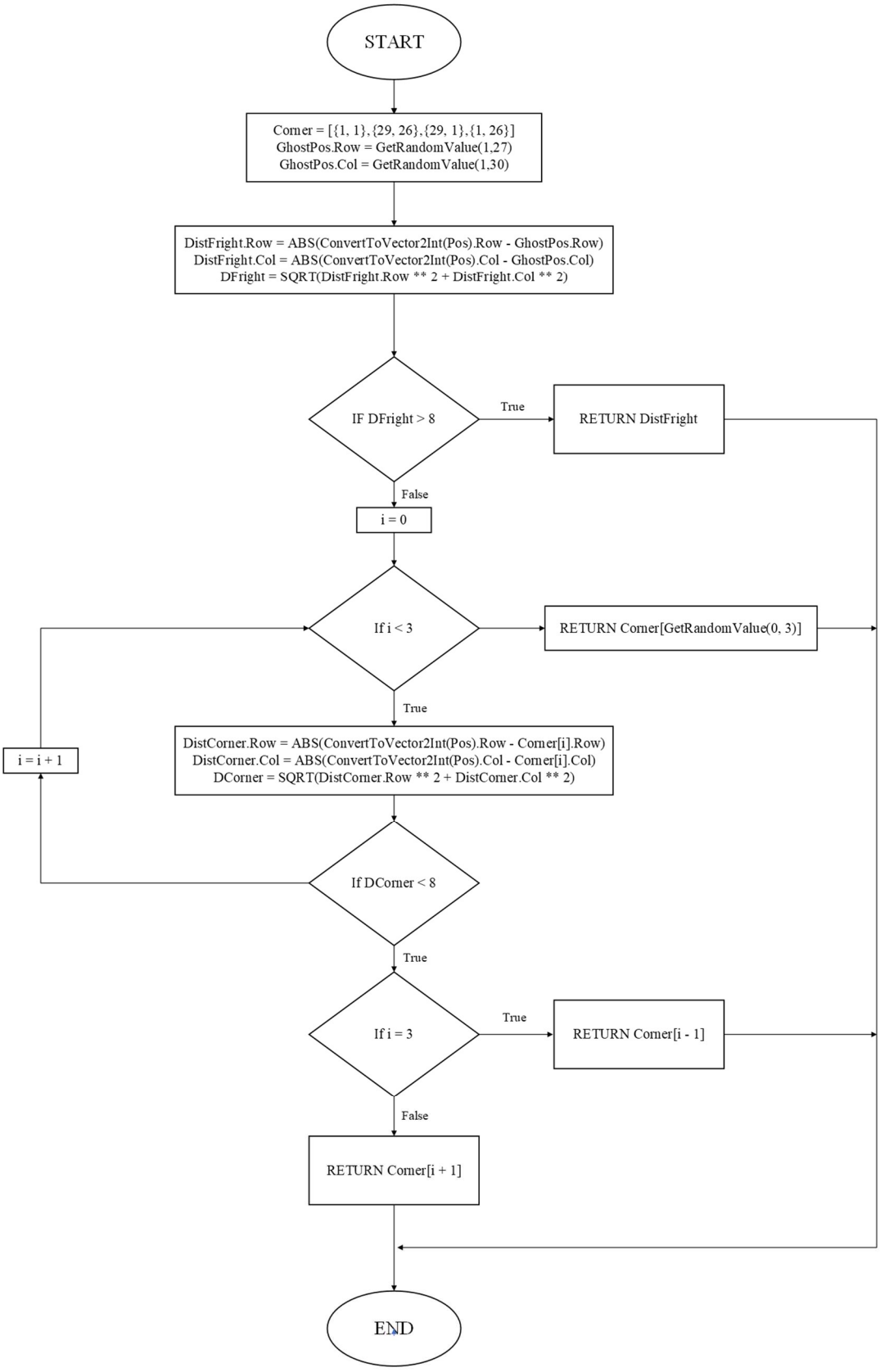
bool ghostcollisionpacman(Vector2 pos, Vector2Int ghostPos)



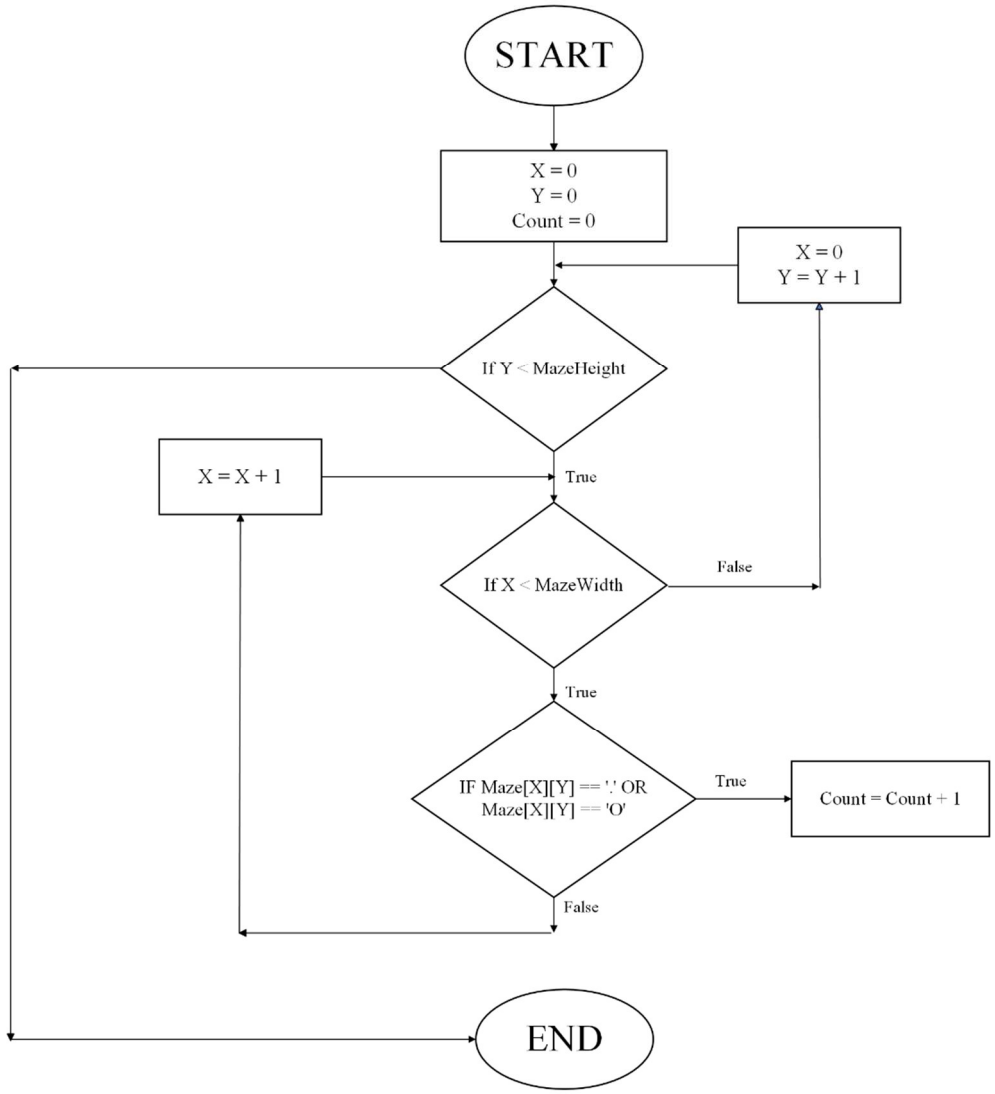
Vector2Int ConvertToVector2Int(Vector2 pos)



Vector2Int RandomPos(Vector2 pos, Vector2Int ghostPos)



```
int countPellets(char maze[MAZE_HEIGHT][MAZE_WIDTH])
```



9. Breadth First Search (BFS) Algorithm Code:

```
Vector2Int* findShortestPath(Vector2Int start, Vector2Int target, int*
pathLength) {
    Vector2Int* queue = (Vector2Int*)malloc(sizeof(Vector2Int) *
MAZE_HEIGHT * MAZE_WIDTH);
    bool visited[MAZE_HEIGHT][MAZE_WIDTH];
    memset(visited, false, sizeof(visited));
    int front = 0, rear = 0;

    Vector2Int parent[MAZE_HEIGHT][MAZE_WIDTH];
    memset(parent, -1, sizeof(parent));

    queue[rear++] = start;
    visited[start.row][start.col] = true;

    while (front < rear) {
        Vector2Int current = queue[front++];
        if (current.row == target.row && current.col == target.col) {
            Vector2Int* path = (Vector2Int*)malloc(sizeof(Vector2Int) *
MAZE_HEIGHT * MAZE_WIDTH);
            int length = 0;
            while (!(current.row == start.row && current.col ==
start.col)) {
                path[length++] = current;
                current = parent[current.row][current.col];
            }
            path[length++] = start;

            *pathLength = length;
            Vector2Int* reversedPath =
(Vector2Int*)malloc(sizeof(Vector2Int) * length);
            for (int i = 0; i < length; i++) {
                reversedPath[i] = path[length - 1 - i];
            }
            free(path);
            free(queue);
            return reversedPath;
        }

        Vector2Int directions[] = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        for (int i = 0; i < 4; i++) {
            Vector2Int next = {current.row + directions[i].row,
current.col + directions[i].col};
            if (isValidPosition(next) && !visited[next.row][next.col])
{
                queue[rear++] = next;
```

```
                visited[next.row][next.col] = true;
                parent[next.row][next.col] = current;
            }
        }
    }

    free(queue);
    return NULL;
}


void moveGhost(Texture2D ghost, Vector2Int* ghostPos, Vector2Int
targetPosghost, int* ghostMoveCounter, Vector2Int** path, int*
pathIndex, int* pathLength, int speed) {
    if (*path == NULL || *pathIndex >= *pathLength) {
        if (*path != NULL) {
            free(*path);
        }
        *path = findShortestPath(*ghostPos, targetPosghost,
pathLength);
        *pathIndex = 0;
    }

    (*ghostMoveCounter)++;
    if (*ghostMoveCounter >= speed) {
        if (*path != NULL && *pathIndex < *pathLength) {
            *ghostPos = (*path)[*pathIndex];
            (*pathIndex)++;
        }
        *ghostMoveCounter = 0;
    }
    DrawTexture(ghost, ghostPos->col * 20, ghostPos->row * 20, WHITE);
}
```

10. Description of Each Module:

```
void drawMaze(char maze[MAZE_HEIGHT][MAZE_WIDTH])
```

This function is responsible for drawing the maze on the screen based on a 2D array representation.

- **Parameters:** char maze - a 2D array representing the maze layout.
- **Functionality:** It iterates through each cell in the maze array and draws different shapes based on the cell value:
    - # - draws a wall.
    - . - draws a small pellet.
    - - draws a larger pellet.
    - _ - draws a special marked area.

```
int avoidRotation(Vector2 *pos, Direction *direction, Direction *buffer_direction, char maze[MAZE_HEIGHT][MAZE_WIDTH], char Char)
```

This function checks if the next position in the intended direction of movement is valid (not blocked by a wall).

- **Parameters:**
    - Vector2 pos - the current position of Pac-Man.
    - Direction direction - the current movement direction of Pac-Man.
    - Direction buffer_direction - the direction input buffered for the next move.
    - char maze - a 2D array representing the maze layout.
    - char Char - the character representing a wall.
- **Functionality:** It checks if the next cell in the intended direction is a wall. If it is, the buffered direction is reset to STOP. Returns 1 if there's a wall and 0 otherwise.

```
void updatePosition(Vector2 *pos, Direction *direction, Direction *buffer_direction, char maze[MAZE_HEIGHT][MAZE_WIDTH], char Char)
```

This function updates the position of Pac-Man based on the current and buffered direction inputs.

- **Parameters:**
    - Vector2 pos - the current position of Pac-Man.
    - Direction direction - the current movement direction of Pac-Man.
    - Direction buffer_direction - the direction input buffered for the next move.

    - char maze - a 2D array representing the maze layout.
    - char Char - the character representing a wall.
- **Functionality:** It updates the position based on the direction while checking for collisions with walls and wrapping around the maze boundaries.

```
void checkCollisionWithPellets(Vector2 pos, char
maze[MAZE_HEIGHT][MAZE_WIDTH], int *score, Sound *sound1, Sound
*sup_siren)
```

This function checks if Pac-Man has collided with pellets or power pellets and updates the score accordingly.

- **Parameters:**
    - Vector2 pos - the current position of Pac-Man.
    - char maze - a 2D array representing the maze layout.
    - int score - a pointer to the current score.
    - Sound sound1 - sound to play when a small pellet is eaten.
    - Sound sup_siren - sound to play when a large pellet is eaten.
- **Functionality:** It checks the current cell for pellets (.) or power pellets (O), updates the score, plays the appropriate sound, and removes the pellet from the maze.

```
void drawPacman(Vector2 pos, Direction direction, Texture2D sprite)
```

This function draws Pac-Man on the screen.

- **Parameters:**
    - Vector2 pos - the current position of Pac-Man.
    - Direction direction - the current movement direction of Pac-Man.
    - Texture2D sprite - the texture to use for drawing Pac-Man.
- **Functionality:** It adjusts the position, rotation, and draws Pac-Man based on the current direction.

```
void drawScore(int score)
```

This function displays the current score on the screen.

- **Parameters:** int score - the current score.
- **Functionality:** It displays the score and the high score at a fixed position on the screen.

```
void drawlives(Texture2D spriteOpen)
```

This function displays the remaining lives on the screen.

- **Parameters:** Texture2D spriteOpen – The picture of Pacman
- **Functionality:** It draws the numbers of lives remaining at a fixed position on the screen.

```
bool isValidPosition(Vector2Int position)
```

This function checks whether a given position within a maze is valid and accessible.

- **Parameters:** Vector2Int position - A structure representing a 2D position within the maze. It contains two integer members: position.row: The row index of the position and position.col: The column index of the position.
- **Functionality:** This function checks if the given position is within the bounds of the maze and if the cell at that position is not a wall.

```
bool ghostcollisionpacman(Vector2 pos, Vector2Int ghostPos)
```

This function checks if the position of Pacman is the same as the position of a ghost, indicating a collision.

- **Parameters:**
  - Vector2 pos - A structure representing the position of Pac-Man in floating-point coordinates (in pixels).
  - Vector2Int ghostPos - A structure representing the position of a ghost in integer coordinates (grid-based).
- **Functionality:** This function converts Pac-Man's floating-point position to an integer grid position and checks if it matches the ghost's position. If they match, it means Pac-Man has collided with the ghost.

```
Vector2Int* findShortestPath(Vector2Int start, Vector2Int target, int* pathLength)
```

The findShortestPath function uses the Breadth-First Search (BFS) algorithm to find the shortest path in a maze from a start position to a target position. It returns the path as an array of Vector2Int coordinates and sets the path length via a pointer.

- **Parameters:**
  - Vector2Int start - The starting position in the maze.
  - Vector2Int target - The target position in the maze.
  - int* pathlength - A pointer to an integer where the length of the path will be stored.
- **Functionality:** The function uses BFS to explore the maze from the start position and finds the shortest path to the target position. It constructs and returns the path if it exists.

BFS Algorithm Explanation

1. **Initialization:**
   a. Queue: A queue is initialized to keep track of positions to explore. BFS uses a queue to ensure it explores positions in the order they were discovered, which guarantees the shortest path in an unweighted grid.
   b. Visited Array: A 2D boolean array (visited) tracks which positions have been visited to avoid reprocessing them.
   c. Parent Map: A 2D array (parent) records the parent (previous position) of each position to reconstruct the path once the target is reached.

2. **Enqueue Start Position:**
   a. The starting position is added to the queue.
   b. The starting position is marked as visited.

3. **BFS Traversal:**
    a. The main BFS loop runs while there are positions in the queue.
    b. The current position is dequeued and checked if it is the target position. If it is, the path reconstruction begins.
    c. For the current position, all possible adjacent positions (up, down, left, right) are explored.
    d. If an adjacent position is valid (within bounds, not a wall, and not visited), it is enqueued, marked as visited, and its parent is set to the current position.

4. **Path Reconstruction:**
    a. When the target position is found, the path is reconstructed by backtracking from the target position to the start position using the parent map.
    b. The path is stored in an array, and then reversed to get the path from start to target.

5. **Return Path:**
    a. The function returns the path and its length.

```
Vector2Int ConvertToVector2Int(Vector2 pos)
```

The ConvertToVector2Int function converts a position from floating-point coordinates (Vector2 - in pixels) to integer grid-based coordinates (Vector2Int – grid-based).

- **Parameters:**
    - Vector2 pos - A structure representing a position in floating-point coordinates.
    - pos.x - The x-coordinate of the position.
    - pos.y - The y-coordinate of the position.
- **Return Value:**
    - Vector2Int - A structure containing integer grid-based coordinates after conversion.
    - vec2Int.row - The row index in the grid.
    - vec2Int.col - The column index in the grid.

- **Functionality:** The function divides the x and y coordinates of the input pos by 20 (the grid cell size) and casts the result to an integer. This effectively converts the continuous floating-point position to discrete integer grid coordinates.

`Vector2Int RandomPos(Vector2 pos, Vector2Int ghostPos)`

The RandomPos function is designed to determine a random position for a ghost in a game, based on certain criteria related to Pac-Man's position (pos) and the current position of other ghosts (ghostPos). When the ghost is frightened the target position for the ghosts is found out using this function.

- **Parameters:**
  - Vector2 pos - Represents the position of Pac-Man in floating-point coordinates.
  - Vector2Int ghostPos - Represents the position of another ghost, which is modified within the function.

- **Local Variables**
  - Vector2Int distfright - Stores the distance vector between Pac-Man's position and the randomly generated ghost position (ghostPos).
  - Vector2Int distcorner - Stores the distance vector between Pac-Man's position and each corner position in the corner array.
  - Vector2Int corner[4] - An array containing four corner positions in the maze grid.

- **Functionality:**
  1. Generate Random Ghost Position
  2. Calculate Distance from Pacman to Random Ghost Position
  3. Check if Pacman is close to Ghost Random Position
  4. If Pacman is close to Ghost Random Position, then check Distance between the Pacman and the Corners
  5. If the Pacman is close to a corner, the return another corner which is not close to the Pacman
  6. If all else fails, return a Random Corner

```
void moveGhost(Texture2D ghost, Vector2Int* ghostPos, Vector2Int targetPosghost, int* ghostMoveCounter, Vector2Int** path, int* pathIndex, int* pathLength, int speed)
```

The moveGhost function controls the movement of a ghost in a game based on a calculated path to a target position (targetPosghost).

- **Parameters:**
  - Texture2D ghost - The texture representing the ghost to be drawn on the screen.
  - Vector2Int* ghostPos - Pointer to a Vector2Int structure holding the current position of the ghost.
  - Vector2Int targetPosghost - The target position where the ghost is moving towards.
  - int* ghostMoveCounter - Pointer to an integer representing the counter for ghost movement.
  - Vector2Int** path - Pointer to a pointer of Vector2Int, storing the calculated path for the ghost to follow.
  - int* pathIndex - Pointer to an integer storing the current index of the ghost's position in the path.
  - int* pathlength - Pointer to an integer holding the length of the current path.
  - int speed - Integer representing the speed of the ghost's movement.

- **Functionality:**
  1. Path Calculation:
     - Checks if there is no valid path (*path == NULL) or if the ghost has reached the end of the current path (*pathIndex >= *pathLength).
     - If so, frees the memory allocated for the current path (if any) and calculates a new shortest path (findShortestPath) from the current ghost position (*ghostPos) to the target position (targetPosghost).
  2. Incrementing Movement Counter
  3. Moving the Ghost
     - Checks if the ghost move counter (*ghostMoveCounter) has reached the specified speed (speed).
     - If true, moves the ghost to the next position in the path ((*path)[*pathIndex]), increments the path index (*pathIndex), and resets the move counter.

4. Drawing the Ghost
    - Draws the ghost (ghost) at its current position (ghostPos->col * 20, ghostPos->row * 20) on the screen.

```
int countPellets(char maze[MAZE_HEIGHT][MAZE_WIDTH])
```

The countPellets function is used to count the number of pellets (both regular pellets and power pellets) in a maze.

- **Parameters:**
    - char maze[MAZE_HEIGHT][MAZE_WIDTH] - A 2D array representing the layout of the maze. Each cell in the maze can contain various characters representing different elements like walls, empty spaces, regular pellets ('.'), and power pellets ('O').
- **Functionality:** The function iterates over every cell in the maze to count the total number of pellets present.

11. Implementation:

a. Data Organization:

```
typedef enum Direction {
    STOP,
    UP,
    DOWN,
    LEFT,
    RIGHT
} Direction;
```

- This code is defining a new enumerated data type called Direction that can have one of five values: STOP, UP, DOWN, LEFT, or RIGHT.
- This is used to set the moving direction of the PAC-MAN.

```
typedef enum GameScreen {
    DESCRIBE,
    DIFFICULTY,
    DIFF_SELECTION,
    LOADING,
    GAMEPLAY,
    REMAININGLIVES,
    GAMEOVER,
    GAMEWIN
} GameScreen;
```

- This code is defining a new enumerated data type called GameScreen that can have one of three values: DESCRIBE, LOADING, GAMEPLAY
- This is used to alternate between consecutive windows of our PACMAN application via switch – case statements.
- DESCRIBE displays the information about all the ghosts and the points obtained when pellets and super pellets are collected.
- DIFFICULTY and DIFFSELECTION displays the difficulty level and lets the user choose the difficulty level.
- LOADING displays the loading screen before the game.
- GAMEPLAY displays the whole game.
- REMAININGLIVES displays the remaining lives after the Pacman has died.
- GAMEOVER screen is displayed when all 3 lives of Pacman have been used.
- GAMEWIN screen is displayed when all the pellets in the maze have been collected before all lives are lost.

```
typedef struct {
    int row;
    int col;
} Vector2Int;
```

- The Vector2Int structure is used to represent coordinates or positions in a 2D grid where both the row and the column are integer values.

```
const char initialmaze[MAZE_HEIGHT][MAZE_WIDTH] = {}
```

- This is a character array containing the maze.
- The MAZE_WIDTH is 28 and MAZE_HEIGHT is 31.
- The maze contains ‘#’ which represents Walls, ‘.’ which represents pellets and ‘O’ which represents super pellets.

Text Files

- This code snippet is used to update the high score for a game based on the current score.
- It determines which high score file to use based on the game level, reads the existing high score from the file, and updates the file if the current score is higher than the existing high score.

b. Libraries Used:

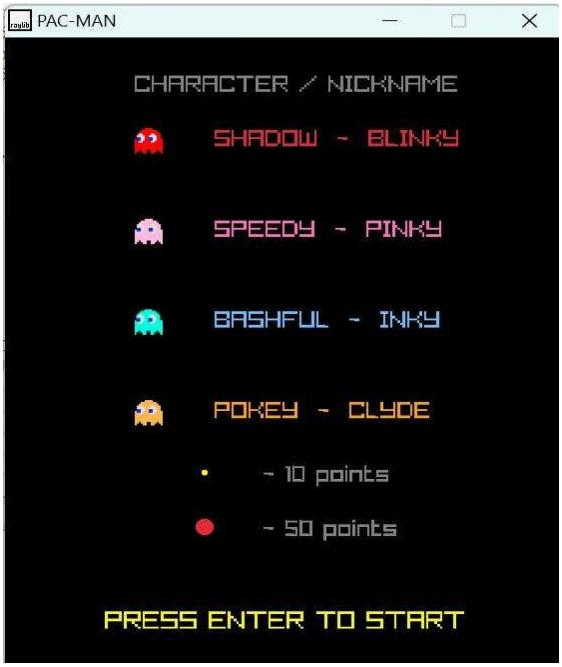
Raylib Library

The Raylib library is a simple and easy-to-use tool for creating graphical applications, including games. In the context of making a Pac-Man game, Raylib provides a set of functions that help handle graphics, input, and other game development tasks such as:

- Initialization and Window Management
- Input Handling
- Drawing Textures and Shapes
- Animations
- Collision Detection
- Audio
- Game State Management
- Pathfinding and Ghost Movement

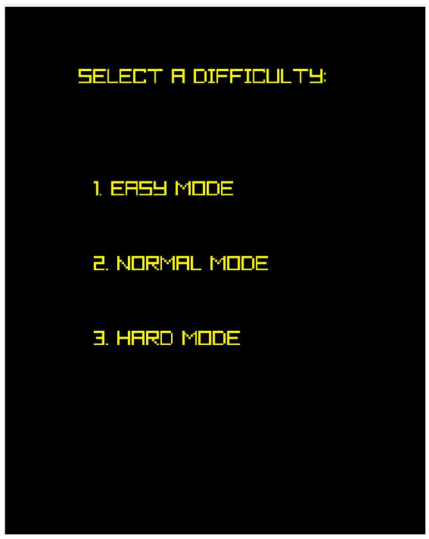
c. User Interface Design:

i. Character info & Start screen:



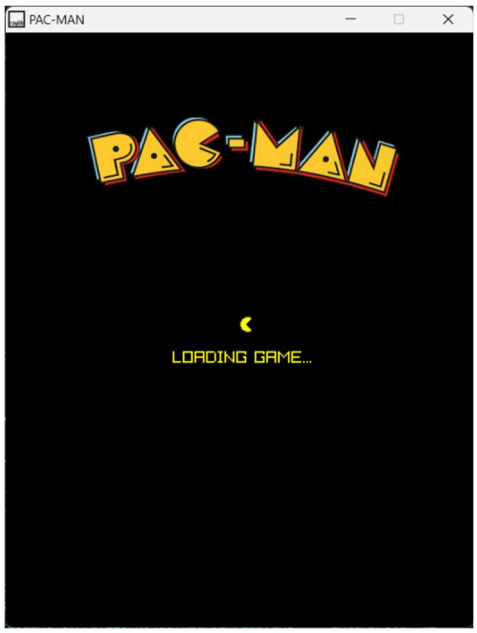
The first screen, listed as DESCRIBE in the GameScreen dataset, displays the character info of the Ghosts (although all of them will not be included) as well as info on the points scored by consuming the pellets – we wanted to recreate the originality of the game. Then, a blinking "Press enter to start" prompt is displayed.

ii. Game Difficulty Selection:



The DESCRIBE screen leads to the difficulty selection screen, namely DIFFICULTY and DIFF_SELECTION. Difficulty is increased by speeding up the ghost. Once a choice is selected, a blinking “Press Enter to Start” message is displayed.

iii. Loading Screen:

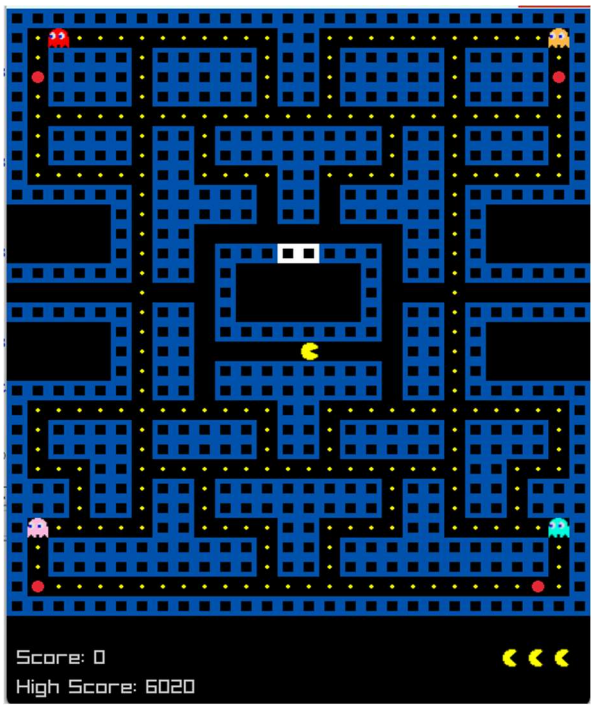


Once the “Enter” key is pressed on the keyboard, the screen changes from DIFF_SELECTION to LOADING. To simulate the game starting up, we designed a loading screen which is synced up with the Pac-Man start up music – the game is shown to be “loaded” as soon as the theme finishes playing.

iv. Gameplay Screen:

The gameplay screen displays the maze, score and Pac-Man is spawned below the ghost’s cage. From here, the player inputs movements to Pac-Man via the arrow keys.

v. Scatter Mode (Extra Feature):



This screen shows the “Scatter Mode” feature - whereby the ghosts first move to different positions before targeting Pacman; this occurs during the start of the game and each time a life is lost.

vi. Early Move Detection (Extra Feature):

Another extra feature we were able to include, mirroring the original Pac-Man game, is the early move detection. That is, when travelling through a straight tunnel path, if the player tries to move in another direction, Pac-Man will automatically move into the next available tunnel in that direction.

If the player tries to move in opposite direction, that happens automatically.

In fig. A, if the player attempts to move right after crossing the tunnel, Pac-Man will move by itself into the next available tunnel in right direction (fig. B).

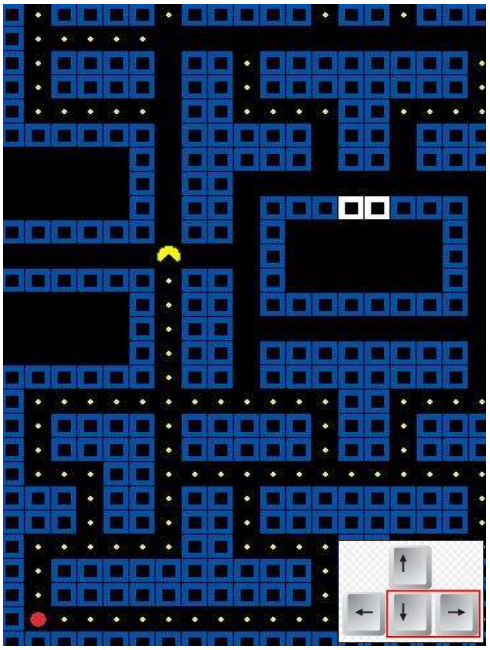


Fig. a

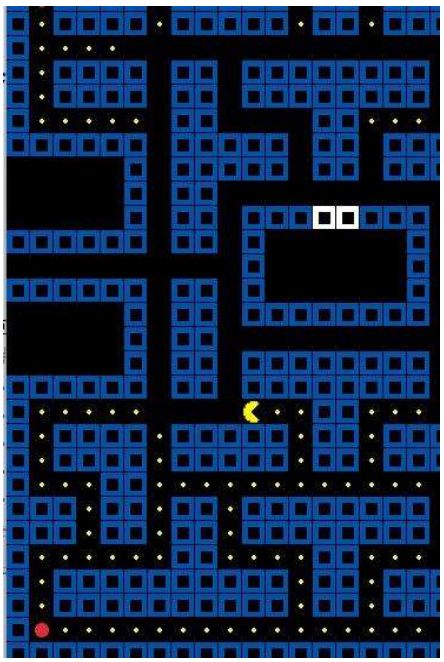
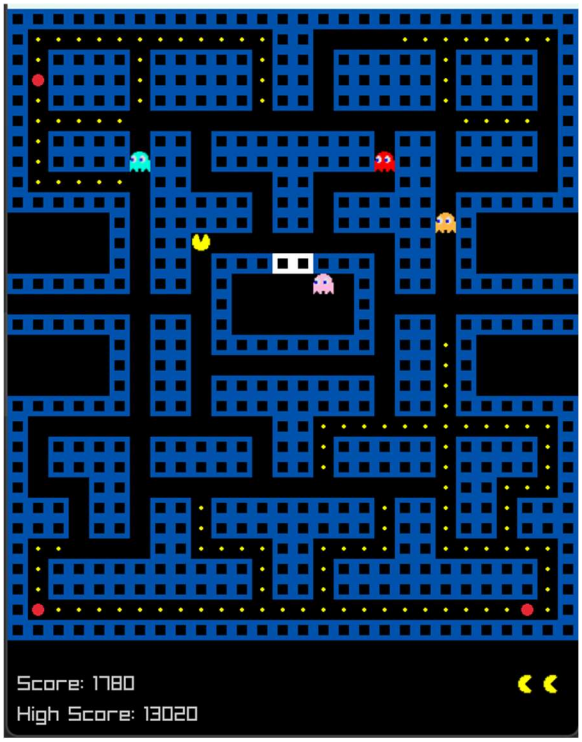


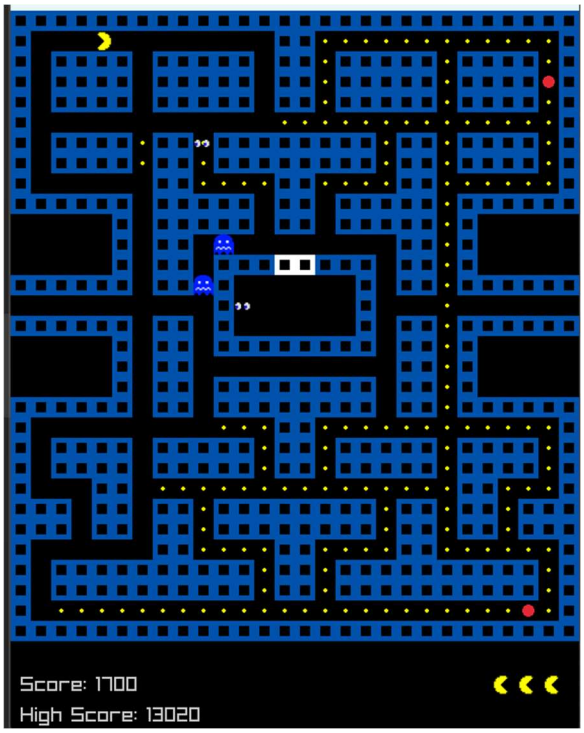
Fig. b

viii. GAMEPLAY Snapshots:



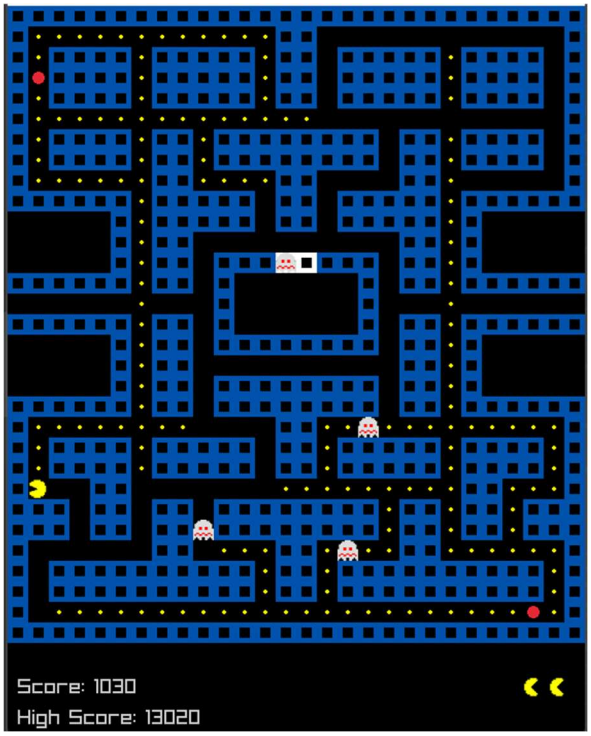
Pacman being chased down by the ghosts, showing the working of BFS algorithm and how the different paths of each ghost "work together" to corner Pacman.

Score High score and Life count can be seen displayed below the maze

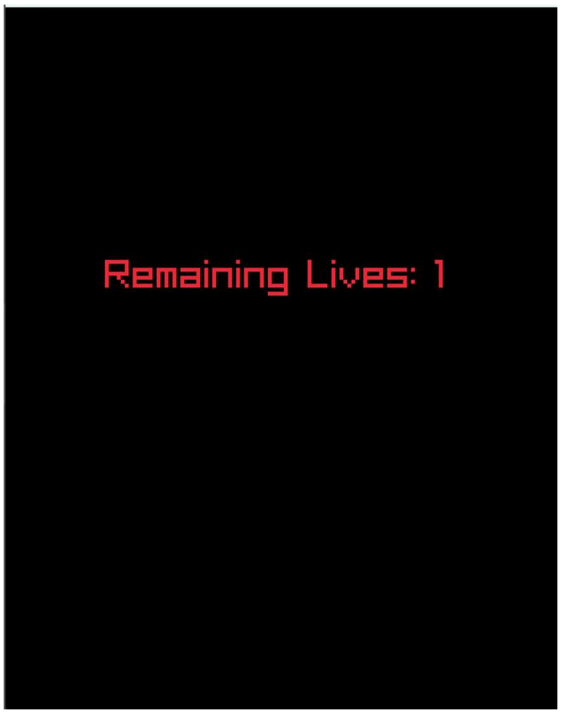


When Pacman eats a super pellet(red), the ghosts become “frightened” and move randomly away from Pacman. If Pacman manages to eat a ghost, it becomes a pair of eyes which travels back to the ghost cage.

During frightened mode, the ghosts cannot capture Pacman.



Before Pacman’s “super pellet boost” runs out, the ghosts flash white, indicating that they will turn back to normal, which serves as a warning to the player that Pacman should not approach them at this time.



Suppose if the player loses a life, after the death animation plays, this screen is displayed.

After this screen, Pacman is reset back to the middle and ghosts are reset back into the cage, after which they undergo scatter mode and then proceed to chase Pacman.

viii. Game-End Screens:

If the player either collects all the pellets, or loses all his lives, the Game-End screen is displayed accordingly. If the user decides to play again by pressing ‘Y’, the game restarts at the same difficulty and the Gameplay starts immediately. By pressing ‘N’, the user exits from the game window.

d. Platform used for Code Development:

We used Visual Studio Code for the Code Development and Google Drive for Sharing the code among the teammates. We obtained the pictures, audio and the maze structure for the Pacman game from Google.

12. Observations with Respect to Society:

Developing a Pacman game is not just about coding and gameplay mechanics, it also intersects with broader societal aspects.

Beyond Gaming: When Pacman arrived in 1980, it transcended the realm of video games. It became a cultural phenomenon, spawning merchandise, cartoons, and even a hit song. The yellow circle with an insatiable appetite left an indelible mark on popular culture.

Fashion and Literature: Pacman inspired a Martin Amis novel and even influenced fashion collections. Its iconic imagery found its way into various creative expressions.

Design Innovation: Pacman's design innovation paved the way for impactful gameplay. By exploring these fundamental techniques, game developers continue to broaden the expressive possibilities in modern games.

Timeless Gameplay: Pacman's enduring appeal lies in its timeless gameplay mechanics. The pursuit of dots, evading ghosts, and gobbling power pellets resonate with players across generations.

Arcade Boom: Before Pacman, games like Space Invaders dominated arcades worldwide. These games fuelled the arcade boom, creating a social space for gamers.

Nostalgia: Pacman evokes nostalgia for a simpler gaming era. Its iconic characters and maze-chase gameplay evoke fond memories for those who grew up playing it.

13. Legal and Ethical Perspectives:

Legal Perspective:

- **Copyright Law:** The original Pac-Man game is protected by copyright law. This means that you cannot use the exact graphics, sounds, or significant portions of the original game's code without permission from the copyright holder.
- **Trademark Law:** The name "Pac-Man" and the distinctive character designs are likely protected by trademark law. Using these in your clone could potentially infringe on these trademarks.

Ethical Perspective:

- **Plagiarism:** Even if it's technically legal to create a clone, copying another game's design without adding anything new or unique could be considered unethical, as it doesn't respect the original creator's effort and creativity.
- **Learning vs. Profit:** If you're creating the clone purely for personal learning and not distributing it or making money from it, this is generally seen as more acceptable.

14. Limitations of Our Solution

Learning Curve:

While Raylib simplifies many aspects of game development, there is still a learning curve involved. Understanding how to use the library effectively can take time.

Limited Community and Resources:

Compared to more popular languages or libraries, there are fewer community resources, tutorials, or examples available for Raylib.

Performance:

While C and Raylib can offer good performance, creating a highly optimized game can still be challenging and require a deep understanding of both the language and the library.

Business Viability:

The game offers basic functionality with limited features. To be commercially viable, consider adding features such as multiplayer modes, achievements, and power-ups.

Lack of Modularity:

The code lacks modularity and separation of concerns. Functions are tightly coupled, and there is significant repetition. For example, direction handling and collision detection could be abstracted into separate functions to enhance readability and maintainability.

15. Learning Outcomes:

- We will be able to implement Data Type Declarations, Input/Output Statements, Operators, Expressions and Conditional Statements in C.
- We will be able to familiarise various Looping constructs (For Loop and While Loop).
- We will be able to define Functions with appropriate parameters and return statements.
- We will be able to use one-dimensional and two-dimensional Arrays and pass them into Functions.
- We will be able to implement user-defined data types using Structures and pass them into Functions.
- We will be able to read, write and append Text Files.
- We will be able to use the functions in the Raylib Library to make the user interface.

16. Timeline:


1
19/04/24
Studying all the modules in the RayLib Library and identify the necessary modules required.
WEEK 1
2
26/04/24
Making table or chart of all the variables and functions needed in the program.
WEEK 2
3
03/05/24
Making a diagram to understand the flow of the entire code and assigning ground rules.
WEEK 3
4
10/05/24
Beginning of the coding stage of the functions required for smooth operation.
WEEK 4
5
17/05/24
Starting to implement graphics part of the game into the program
WEEK 5
6
31/05/24
Completion of the code for the program. Testing all the test case scenarios and debugging.
WEEK 7
7
14/06/24
Program is complete and ready for submission.
WEEK 9

17. References:

For Understanding the Pac-Man Game:

https://pacman.fandom.com/wiki/Pac-Man_(game)

For Graphical User Interface (GUI):

https://www.Raylib.com/

For Understanding Activity Diagrams:

https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/

For Making Activity Diagrams:

https://gaphor.org/en/

For Understanding Data Flow Diagrams:

https://en.wikipedia.org/wiki/Data-flow_diagram

For Making Data Flow Diagrams:

https://www.microsoft.com/en-in/microsoft-365/visio/flowchart-software

For Understanding Structure Charts:

https://www.geeksforgeeks.org/software-engineering-structure-charts/

https://en.wikipedia.org/wiki/Structure_chart

https://gyansanchay.csjmu.ac.in/wp-content/uploads/2022/04/Structure-Chart-1.pdf

For Making Structure Charts:

https://app.diagrams.net/