DEPARTMENT OF
SOFTWARE ENGINEERING

PROJECT REPORT

VISUAL PROGRAMMING LAB

PREPARED BY

MUHAMMAD MOOSA KHALIL (01-131222-035)
TAYYAB AAMIR ALI (01-131222-048)
TALHA BIN TAHIR (01-131222-047)

SUBMIT TO
MAAM RAHEELA

Abstract

The Pac-Man Windows Forms Application is a comprehensive desktop-based game developed using the .NET Framework and Windows Forms. This project reimagines the classic Pac-Man arcade game, integrating advanced features and modern programming techniques to deliver a seamless, interactive gaming experience. Designed as a cross-threaded application, the project emphasizes efficient resource management, responsive user interactions, and a visually appealing interface.

The gameplay involves controlling the iconic Pac-Man character to navigate through custom-designed maps, collect points by consuming pellets, and avoid ghost characters, which are driven by multithreaded AI. The application utilizes threading to enable simultaneous character movements and collision detection, ensuring a dynamic and engaging user experience. Furthermore, the project incorporates modular components, allowing flexibility in map design and game customization.

Graphics and sound effects are seamlessly integrated to enhance the immersive quality of the game. User interactions are managed through event-driven programming, which supports real-time responses to player inputs. The application also showcases the effective use of object-oriented principles to structure game components, such as player characters, ghost behaviors, and score management.

The Pac-Man Windows Forms Application demonstrates the potential of Windows Forms for game development, particularly in leveraging threading and modular design for interactive applications. This project is an exploration of how classic games can be adapted for modern platforms while preserving their nostalgic appeal. Future enhancements could include support for multiplayer gameplay, additional levels, and improved AI for ghost behaviors, offering a broader scope for further development.

By combining technical innovation with classic gameplay mechanics, this project not only highlights the strengths of the .NET Framework but also provides a valuable learning experience in game development, user interface design, and performance optimization.

Table Of Contents

Introduction

1.1 Background

Pac-Man is one of the most iconic arcade games in gaming history, originally released in 1980. Known for its simple yet engaging gameplay, the game challenges players to navigate a maze, collect pellets, and avoid ghosts, which are controlled by basic AI. This project reimagines the Pac-Man game for a modern desktop platform, utilizing .NET Windows Forms to deliver an interactive and enhanced version of the classic experience.

The desktop-based game has been developed as a part of an academic project to demonstrate practical implementation of object-oriented programming, multithreading, and user interface design. It serves as a platform to explore how modern tools and technologies can be employed to recreate timeless games with improved features and functionality.

1.2 Problem Statement

Classic arcade games like Pac-Man were built using limited technology and were restricted to dedicated gaming hardware. Modernizing such games while retaining their original charm requires implementing advanced programming techniques and a structured software architecture. Furthermore, maintaining smooth performance during concurrent actions, such as player movements and ghost AI, presents a significant challenge in game development.

This project addresses these challenges by leveraging the .NET Framework and Windows Forms to build a cross-threaded application that ensures smooth gameplay while maintaining the nostalgic essence of the original Pac-Man.

1.3 Objectives

The primary objectives of the project are:

- To design and implement a desktop-based Pac-Man game using .NET Windows Forms.
- To utilize threading techniques for simultaneous gameplay actions and ghost movements.
- To create a visually appealing user interface with interactive game elements.
- To ensure modularity, enabling easy customization of game components such as maps and characters.
- To integrate sound effects and graphics for an immersive gaming experience.

1.4 Significance of the Project

This project is significant as it demonstrates the practical application of programming concepts such as multithreading, event-driven programming, and object-oriented design. It also showcases how modern development tools like the .NET Framework and Windows Forms can be utilized for game development. Beyond academic purposes, the project serves as a foundation for further exploration into interactive application development and performance optimization.

By reimagining a classic arcade game, the project not only pays homage to the legacy of Pac-Man but also provides insights into the challenges and solutions associated with developing engaging and responsive software applications.

1.5 Project Overview:

The project is a **Pacman Game** implemented using **Windows Forms** in C#. It creates a simulation of the classic arcade game **Pacman**, where the Pacman character moves around a grid, interacting with walls and doors while following specific movement logic. This code is structured into different classes and methods that define the behavior of the game, focusing on the **Pacman** character and its movement.

Key Components of the Project

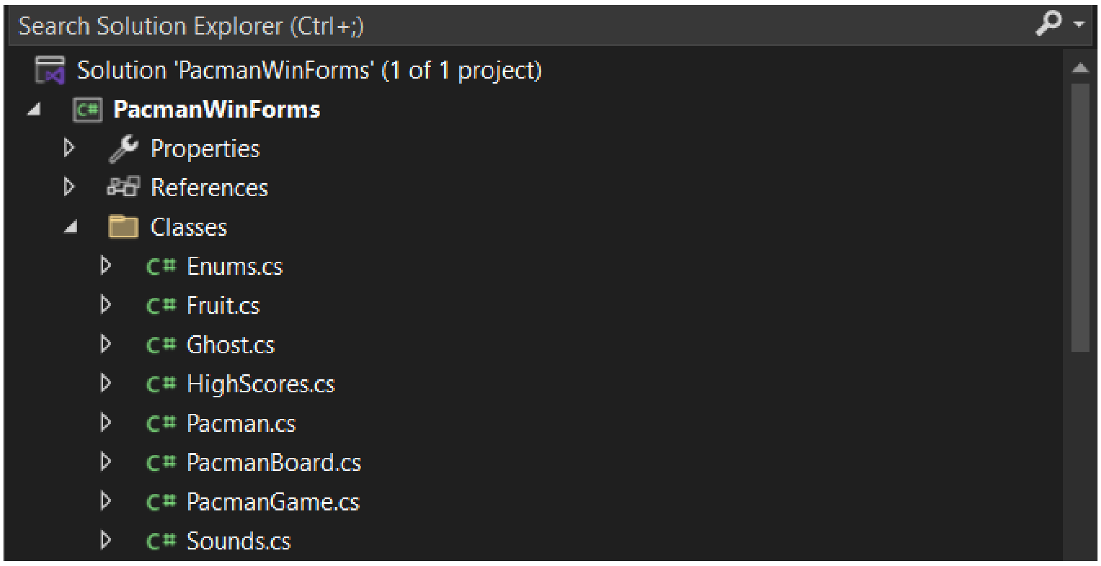
1. **Pacman Class**:
   - **Purpose**: The Pacman class represents the Pacman character on the game board. It defines the character's position (Point) and its movement direction (Direction).
   - **Core and Perimeter**: The class also defines the **core** and **perimeter** of the Pacman, which are points representing the inside and boundary of the Pacman character. These points are used for collision detection and rendering the character on the screen.
2. **PacmanRun Class**:
   - **Purpose**: The PacmanRun class manages the movement and behavior of the Pacman character during the game. It initializes the game state, handles input for movement, checks for collisions, and controls the game's flow.
   - **Game States**: The class uses the GameState enumeration to manage different phases of the game (such as GAMEOVER or GAMERUN).
   - **Movement Logic**: The Pacman’s movement is handled by the runPacman method, which continuously updates the position of Pacman and checks for collisions with walls and doors. The direction and speed of movement are managed using the pacmanNextDirection variable and the Delay property.
   - **Collision Detection**: The collisionCheck method ensures that the Pacman character does not move through walls or other obstacles by checking if the

intended movement causes a conflict with the game environment (walls, doors, etc.).

3. **Game Mechanics**:
   - **Pacman Movement**: The game logic checks for possible moves in four directions (UP, DOWN, LEFT, RIGHT). If a move is valid (i.e., no collision occurs), the Pacman moves to the next position. If a move is blocked, Pacman either stops or waits for the next valid move.
   - **Teleportation**: Special cases are handled for Pacman moving through tunnels on the board. When Pacman reaches the left or right tunnel, it wraps around the board to the other side, continuing its movement in the opposite direction.
4. **Rendering and Interaction**:
   - **Game Board**: The board is a grid where the Pacman character, walls, and other objects are drawn. The PacmanBoard class manages the rendering of these elements on the screen.
   - **Drawing Pacman**: The position and direction of the Pacman are updated in real-time by the DrawPacMan method, which renders Pacman in its current position and direction on the game board.
   - **Resetting the Game**: The reset method allows the game to be restarted with a new stage, clearing the Pacman's previous position and reinitializing the map.
5. **Game Configuration**:
   - **Stage Handling**: The game supports multiple stages (levels), with each stage having different starting positions for Pacman. The map is reloaded for each new stage, and the Pacman's starting position is adjusted accordingly.
   - **Direction Control**: Users can set the direction of Pacman's movement through keypresses, with the setDirection method controlling the movement based on player input and collision checks.
6. **Concurrency**:
   - **Asynchronous Execution**: The Task object is used to run the game loop in the background, allowing Pacman's movement to be processed asynchronously. This ensures that the game runs smoothly without blocking the main UI thread.

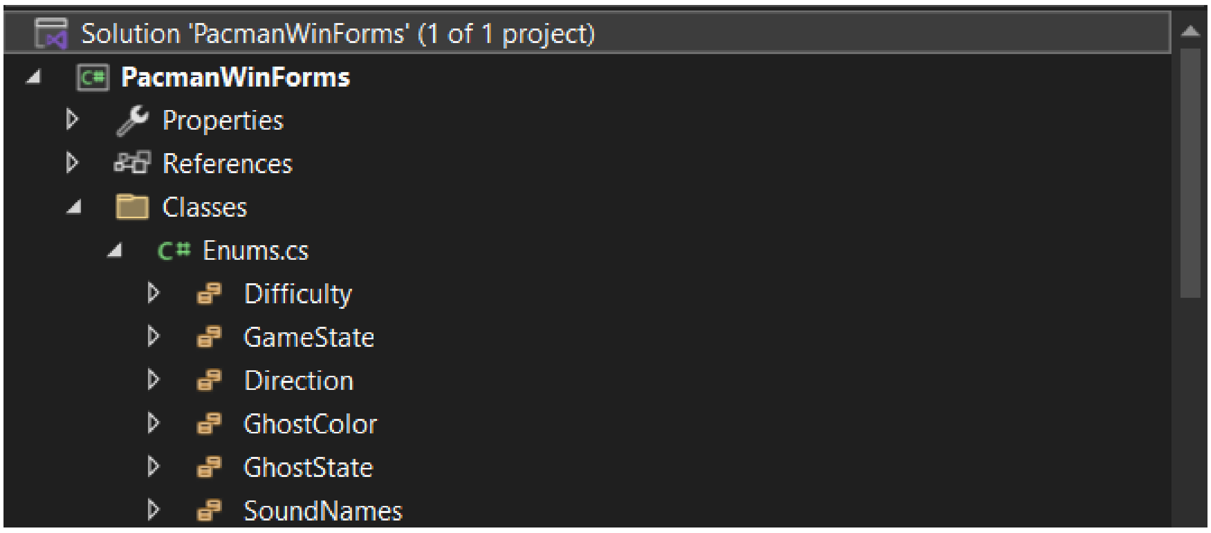
Project structure:

CLASSES:



Search Solution Explorer (Ctrl+;)
Solution 'PacmanWinForms' (1 of 1 project)
PacmanWinForms
Properties
References
Classes
Enums.cs
Fruit.cs
Ghost.cs
HighScores.cs
Pacman.cs
PacmanBoard.cs
PacmanGame.cs
Sounds.cs

Enum.cs:



Solution 'PacmanWinForms' (1 of 1 project)
PacmanWinForms
Properties
References
Classes
Enums.cs
Difficulty
GameState
Direction
GhostColor
GhostState
SoundNames

This class defines several enumerations (enums) to manage different game elements and states in the Pac-Man game.

Key Components:

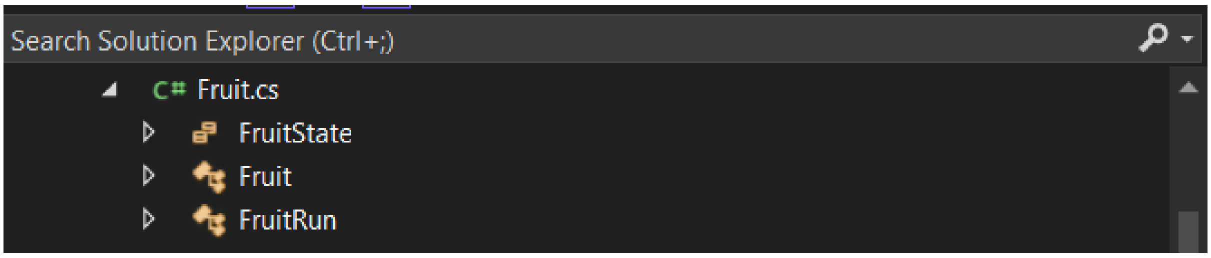
- **Difficulty**: Represents the game's difficulty levels (Easy, Normal, Hard, Original AI).
- **GameState**: Tracks the current state of the game (e.g., running, paused, waiting, or over).
- **Direction**: Specifies movement directions (Up, Down, Left, Right) and Stop for game objects.

- **GhostColor and GhostState**: Represents the types of ghosts by color and their states (e.g., normal, bonus, eaten).
- **SoundNames**: Enumerates sound effects used in the game (e.g., Siren, Waka).

Features:

- **Simplifies Management**: Enums provide a clean and organized way to manage and refer to various game elements and states.
- **Improved Readability**: Using enums enhances code readability and reduces the chance of errors in referring to game elements.

Fruit.cs:



Search Solution Explorer (Ctrl+;)

```
◢ C# Fruit.cs
    ▷ FruitState
    ▷ Fruit
    ▷ FruitRun
```

This class defines the Fruit and FruitRun classes, which manage the behavior and movement of fruit objects in the Pac-Man game.

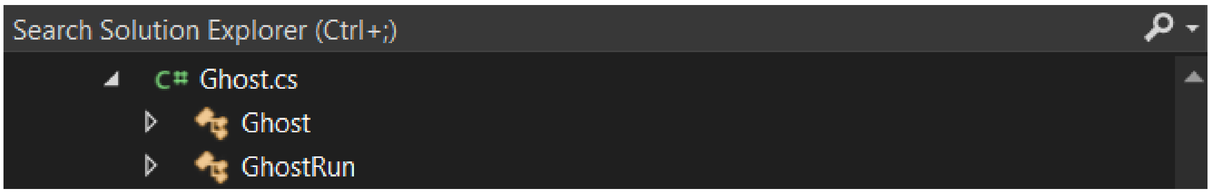
Key Components:

- **Fruit Class**: Represents individual fruit objects, with properties for position, movement direction, and methods for collision detection and rendering.
- **FruitRun Class**: Manages the fruit's movement, collision detection, randomization, and rendering on the game board. It also handles fruit state and stage-specific logic.

Features:

- **Movement**: Fruit moves across the board while avoiding walls and box doors, using cryptographic randomization for direction.
- **Collision Detection**: Ensures the fruit avoids collisions with walls and doors.
- **Game Integration**: Works with the PacmanBoard to update the fruit’s position and render it dynamically.
- **Concurrency**: Utilizes a Task to run the fruit’s movement independently, ensuring smooth gameplay.

Ghost.cs:


Search Solution Explorer (Ctrl+;)

- Ghost.cs
  - Ghost
  - GhostRun

The Ghost and GhostRun classes manage the behavior and movement of ghosts in the Pac-Man game, incorporating AI logic and collision detection.

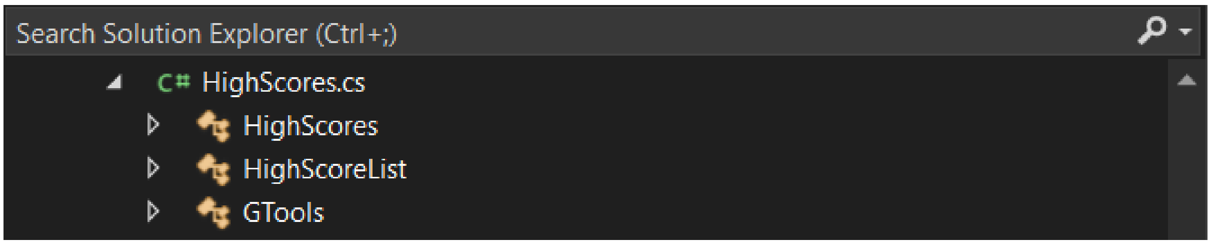
Key Components:

- **Ghost Class**: Manages ghost position, direction, and perimeter/core calculations for collision detection.
- **GhostRun Class**: Handles ghost behavior using AI algorithms like A*, Best-First Search, and BFS. Manages state transitions and updates.

Features:

- **Movement Algorithms**: AI algorithms control ghost movement based on color and game logic.
- **State Management**: Ghosts switch between states (e.g., NORMAL, BONUS) to affect movement and player interaction.
- **Collision and Pathfinding**: Validates movements to avoid collisions with walls, boxes, and the player.
- **Multithreading**: Ghost movement runs independently in a separate task, ensuring smooth gameplay.

HighScores.cs


Search Solution Explorer (Ctrl+;)

- HighScores.cs
  - HighScores
  - HighScoreList
  - GTools

This code manages high scores for a Pacman game in WinForms, supporting persistence, sorting, and file handling for score data.

Key Components:

- **HighScores Class**: Represents an individual high score entry with properties like score, player name, coins added, and difficulty.
- **HighScoreList Class**: Manages high scores, ensuring only the top 10 scores are kept and provides methods for adding, removing, and retrieving scores.
- **GTools Class**: Provides utility methods for saving, loading, and converting high scores to a DataTable, with file management capabilities.

Features:

- **Persistence**: Stores high scores in a JSON file in the AppData folder.
- **Sorting**: High scores are sorted in descending order.
- **Safety**: Includes error handling for file and serialization issues.
- **Dynamic Utility**: Converts high scores to a DataTable for easy display in WinForms controls.

Pacman.cs

Search Solution Explorer (Ctrl+;)

```
◢ C# Pacman.cs
    ▷ Pacman
    ▷ PacmanRun
```

The class defines the Pacman and PacmanRun classes, which manage Pac-Man's behavior, movement, and interactions within the game.

Key Components:

- **Pacman Class**: Represents the Pac-Man character with attributes for position, direction, perimeter, and core. Methods calculate the boundary (perimeter) and core points based on Pac-Man's position.
- **PacmanRun Class**: Manages the game logic for Pac-Man's movement, collision detection, and state transitions. It handles the continuous movement of Pac-Man and manages game stages and speed.

Features:

- **Movement**: Pac-Man's movement is controlled by the PacmanRun class, which updates its position and checks for collisions with walls and restricted areas.
- **State Management**: Tracks the game state (e.g., GAMERUN, GAMEOVER) and manages stage-specific settings.
- **Collision Detection**: Ensures Pac-Man avoids walls and restricted areas during movement.

- **Game Loop**: Uses a Task to continuously update Pac-Man's movement in the game loop, providing smooth gameplay.
- **Modular Design**: Separates Pac-Man's visual representation and gameplay mechanics for easier maintenance and modification.

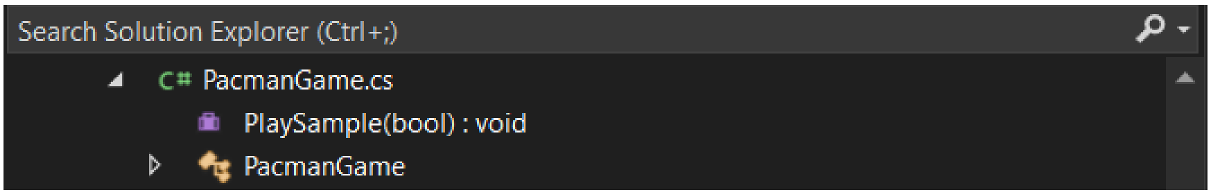
PacmanBoard.cs

The PacmanBoard class handles rendering and management of game elements, including Pac-Man, ghosts, and fruits within the Pac-Man game.

Key Components:

- **Managing Fruits**: Adds/removes fruit images using PictureBox controls on the panel.
- **Drawing Pac-Man**: Draws Pac-Man at specific coordinates with appropriate colors and directions.
- **Handling Ghosts**: Manages different ghost images (RED, BLUE, PINK, YELLOW) with various states and directions.
- **Positioning Elements**: Accurately positions game elements (fruits, ghosts, Pac-Man) based on the grid.
- **Image Handling**: Utilizes Bitmap resources for images, adjusting them according to game states.
- **Clearing and Updating Images**: Clears previous images and updates them with new positions or states.
- **Angle Calculations**: Computes angles for smooth movement arcs of Pac-Man.
- **Drawing with Graphics**: Uses the Graphics object to render elements directly onto the panel.
- **Managing Panel State**: Tracks and updates the game state using counters and state variables.
- **Interaction with GUI**: Handles dynamic updates and user interface interactions within the game board.

PacmanGame.cs:


Search Solution Explorer (Ctrl+;)

C# PacmanGame.cs

PlaySample(bool) : void

PacmanGame

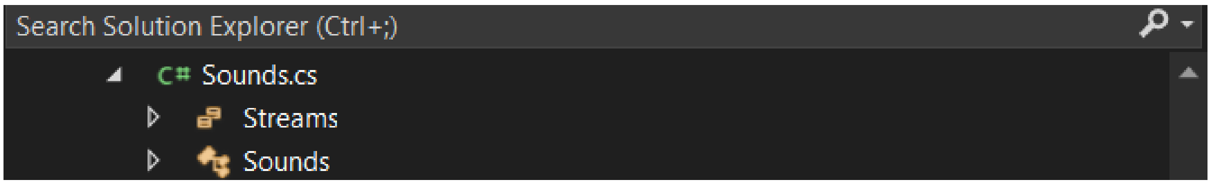
The PacmanGame class manages game mechanics, player interactions, and state transitions in a Pac-Man-like game.

Key Components:

- **eatDots**: Removes dots from the board when Pac-Man eats them, updates the score, and plays a sound.
- **eatBonus**: Removes bonus items, updates score, changes ghosts to "BONUS" state, and plays a sound.
- **switchDifficulty**: Returns the current game difficulty (Easy, Normal, Hard, or Original_AI).
- **checkForLose**: Checks for collisions between Pac-Man and ghosts, transitions to a "wait" state if a collision occurs, reduces lives, and plays a death sound.
- **checkForWin**: Checks if all dots and bonuses are eaten, progresses to the next level, resets items, and plays a victory sound.
- **eatFruit**: Handles Pac-Man eating a fruit, updates the score, advances fruit state, and plays a sound.
- **ghostEatenCounter & eatGhost**: Manages interactions with ghosts, awarding points for each ghost eaten, and ensures ghosts aren't already in the "EATEN" state.
- **resetPrintables**: Clears visual elements like bonus scores to ensure smooth game visuals.
- **reset**: Resets game state (score, lives, level) for a new session.
- **nextLevel**: Advances to the next level, adjusts difficulty, and resets game elements for the new level.

This class ensures smooth gameplay, handling player actions and game state transitions.

Sounds.cs:


Search Solution Explorer (Ctrl+;)

C# Sounds.cs

Streams

Sounds

The Sounds class manages audio playback in the Pac-Man game, handling background music, sound effects, and volume control. It integrates sounds with gameplay events like eating dots, eating fruit, and transitioning levels.

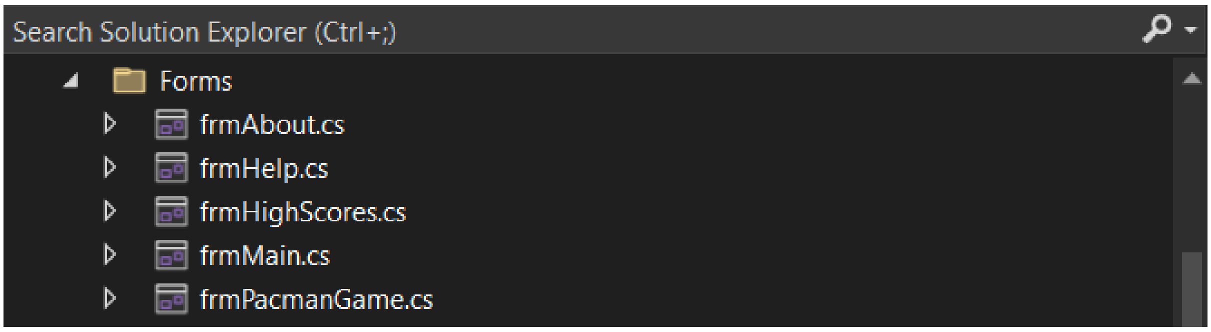
Key Components:

- **Audio Playback**: Manages background music and sound effects using MediaPlayer.
- **Volume Control**: Retrieves and adjusts system volume via waveOutGetVolume and waveOutSetVolume.
- **Background Music**: Plays and loops music based on the game state.
- **Sound Effects**: Manages effects like eating dots and "waka-waka", ensuring correct playback.
- **File Path Handling**: Dynamically locates and loads sound resources.
- **Event Handling**: Automatically restarts audio playback when finished.
- **Customizable Volume**: Allows volume adjustments based on user preference or game state.
- **Gameplay Integration**: Links sounds to in-game events like eating dots and level transitions.

Features:

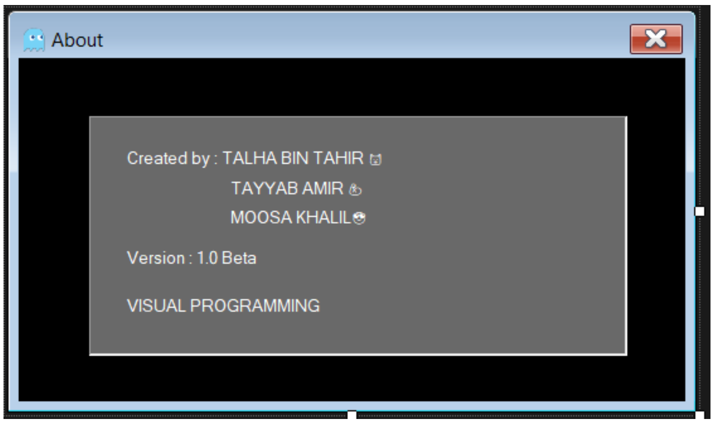
- **Playback Management**: Controls background music and sound effects.
- **Volume Control**: Adjusts audio volume programmatically.
- **Music Looping**: Loops background music based on the game state.
- **Sound Reset**: Resets sound effects after playback.
- **Dynamic File Paths**: Locates audio files dynamically.
- **Audio Restart**: Restarts sounds when they finish.
- **Volume Customization**: Adjusts volume to user preferences.
- **Event Integration**: Ties sounds to gameplay actions.

Forms UI:


Search Solution Explorer (Ctrl+;)

- Forms
  - frmAbout.cs
  - frmHelp.cs
  - frmHighScores.cs
  - frmMain.cs
  - frmPacmanGame.cs

frmAbout.cs: This form provides information about the Pac-Man game, including details about the game's development, creators, and credits. It serves as an informational screen for users to learn more about the game. The form typically appears when the user selects "About" from the menu or a similar option.

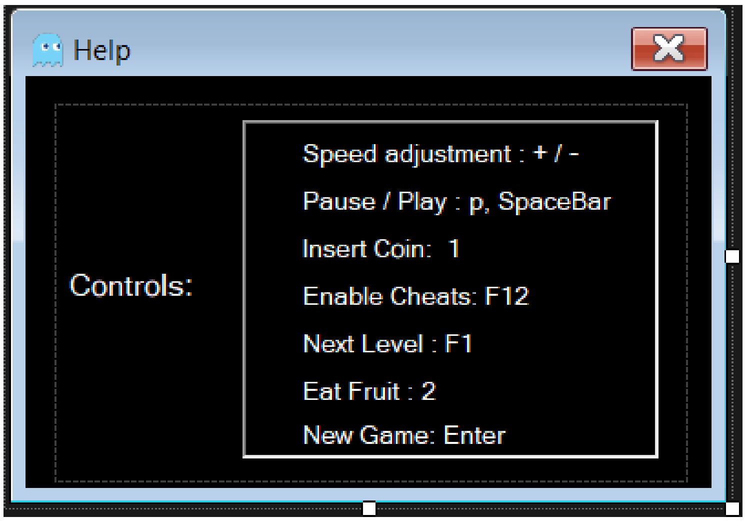

About

Created by : TALHA BIN TAHIR
TAYYAB AMIR
MOOSA KHALIL

Version : 1.0 Beta

VISUAL PROGRAMMING

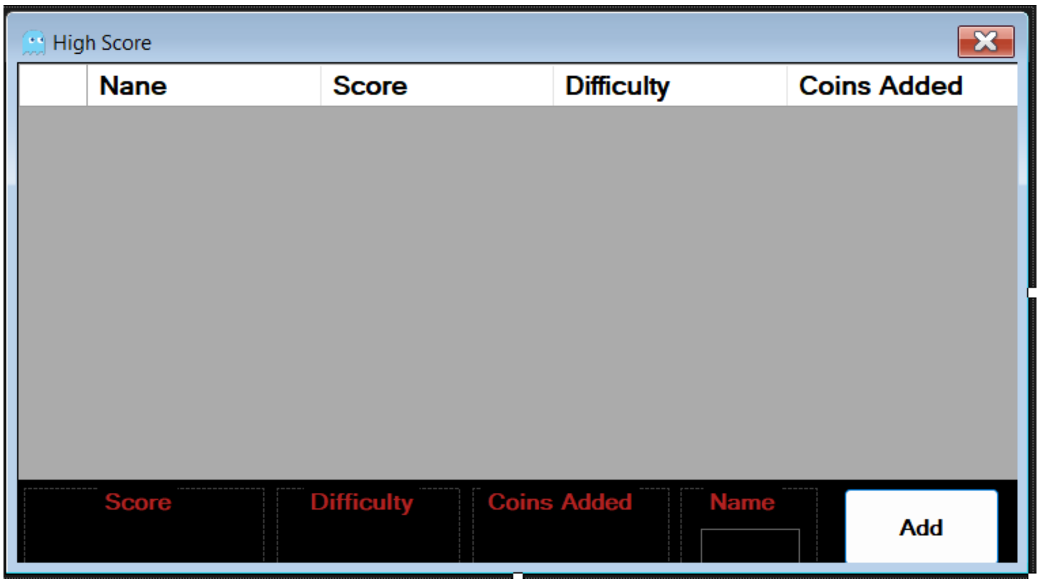
frmHelp.cs: This form provides users with instructions on how to play the Pac-Man game, including controls and gameplay tips. It offers guidance on game mechanics, such as how to navigate, eat dots, and avoid ghosts. The form can be accessed from the game menu to assist players in understanding the game.


Help

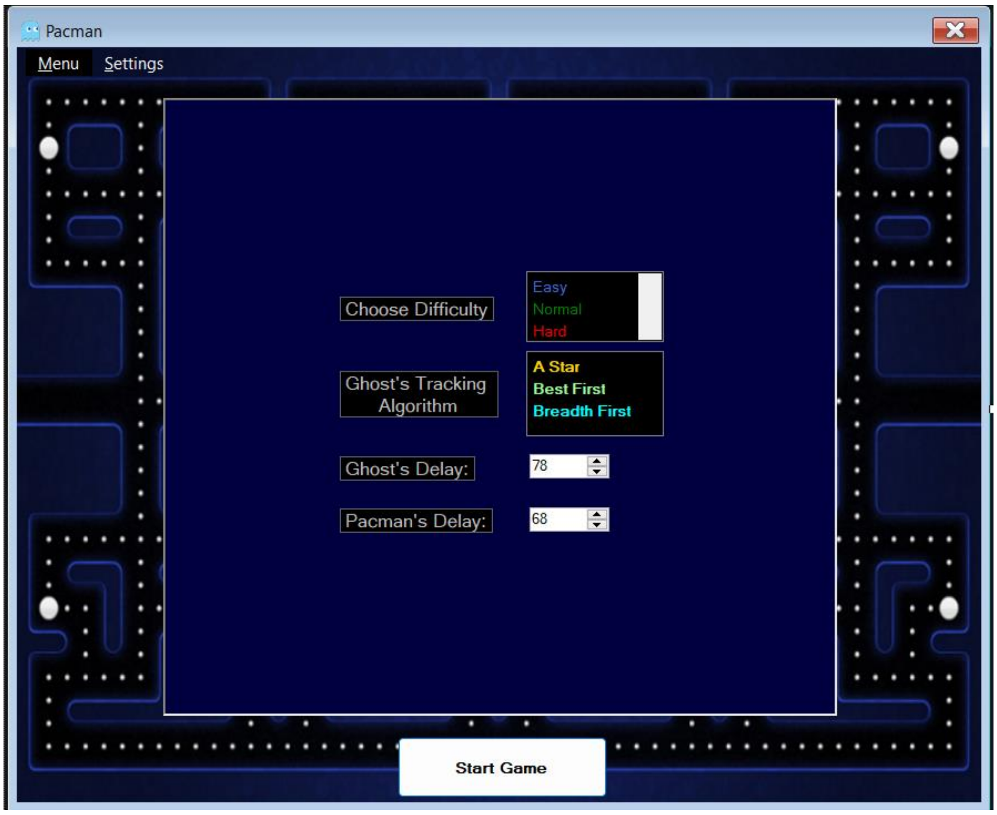
Controls:

Speed adjustment : + / -
Pause / Play : p, SpaceBar
Insert Coin: 1
Enable Cheats: F12
Next Level : F1
Eat Fruit : 2
New Game: Enter

frmHighScores.cs: This form displays a list of the highest scores achieved in the Pac-Man game. It allows players to view top scores and track their progress against others. The form also provides an option to reset the high scores, ensuring the leaderboard stays updated.



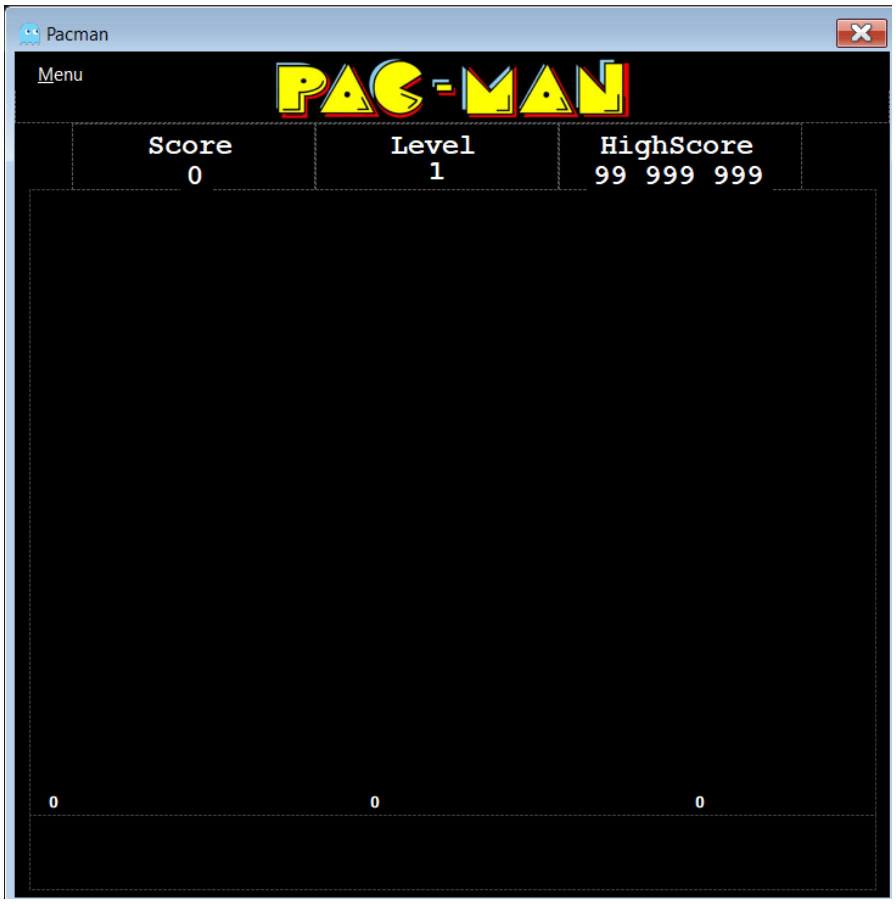
frmMain.cs: This is the main form of the Pac-Man game, serving as the game's central interface. It includes controls for starting, pausing, and resetting the game, along with displaying the game board. Players can also access settings, help, and high score features from this form.



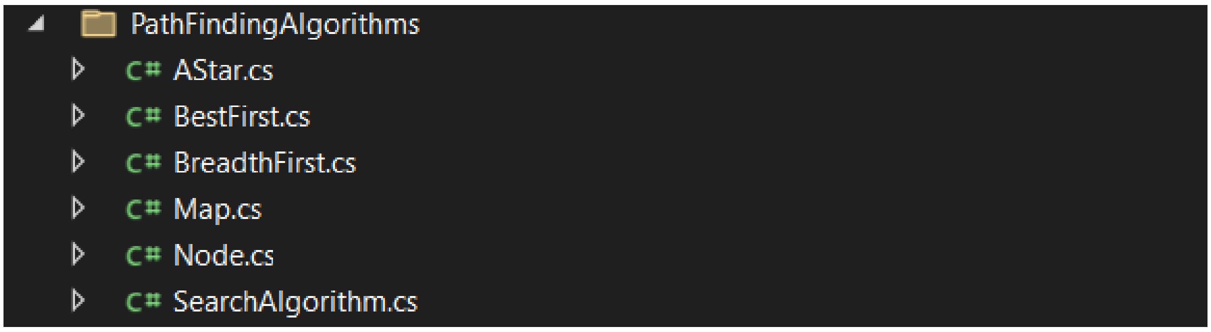
Ghost's Delay: 78

Pacman's Delay: 68

frmPacmanGame.cs: This form handles the core gameplay mechanics of the Pac-Man game, including rendering the game board, player movements, and interactions with ghosts, dots, and fruits. It also manages game state transitions such as winning, losing, and progressing to the next level. The form is responsible for running the game loop and updating the game visuals in real-time.



Algorithms and related classes:



```
◢ PathFindingAlgorithms
    ▷ C# AStar.cs
    ▷ C# BestFirst.cs
    ▷ C# BreadthFirst.cs
    ▷ C# Map.cs
    ▷ C# Node.cs
    ▷ C# SearchAlgorithm.cs
```

AStar Class

Overview:

- **Purpose**: Implements the A* search algorithm to find the shortest path to a target node on a game map, considering obstacles.
- **Fields**: _open – a list of nodes to be evaluated.
- **Constructors**:
  - AStar() – Initializes an empty A* search.
  - AStar(Map map) – Initializes A* with a specific map.
- **Run Method**: Starts the search by adding the root node to the open list.
- **Reset Method**: Clears the open list and resets the search.
- **Update Method**: Finds the node with the lowest f-score, expands neighbors, and checks if the target is reached.
- **Heuristic Calculation**: Evaluates nodes based on cost and distance to the target.
- **State Management**: Tracks node progress with states (e.g., unvisited, in progress, visited).
- **Performance Measurement**: Uses a stopwatch to measure search time.
- **Output**: Finds the shortest path while avoiding obstacles.

BestFirst Class

Overview:

- **Purpose**: Implements Best First Search to navigate nodes based on proximity to the target, prioritizing nodes with the best heuristic.
- **Fields**: _open – a list of nodes to be evaluated.
- **Constructors**:
  - BestFirst() – Initializes an empty Best First Search.

    - BestFirst(Map map) – Initializes Best First Search with a specific map.

- **Run Method**: Starts the search by adding the root node to the open list and setting up the initial state.
- **Reset Method**: Clears the open list and resets the search.
- **Update Method**: Finds the node with the lowest heuristic value, expands neighbors, and stops if the target is found. Ends when the open list is empty.
- **Heuristic Calculation**: Evaluates nodes based on their estimated closeness to the target.
- **State Management**: Tracks node progress through various states (e.g., unvisited, in progress, visited).
- **Performance Measurement**: Uses a stopwatch to measure the time taken for the search.
- **Output**: Finds the fastest path by focusing on nodes closest to the target, though it may ignore obstacles and cause inaccuracies.

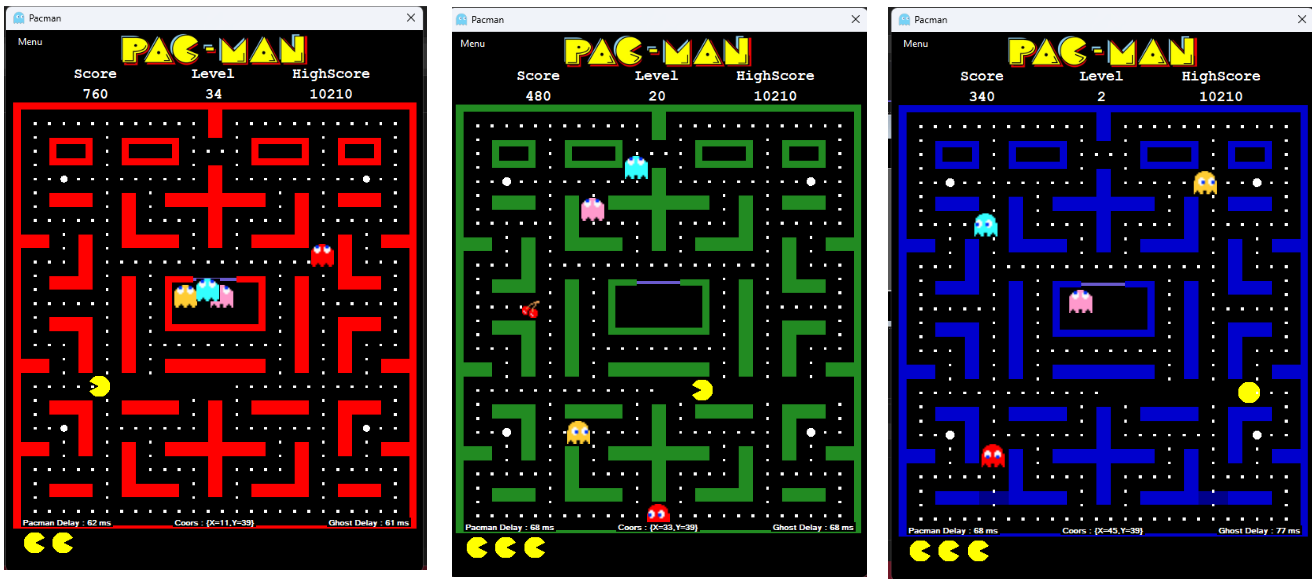
BreadthFirst Class

Overview:

- **Purpose**: Implements Breadth First Search to explore nodes level by level, expanding outward from the root node.
- **Fields**:
    - _open – a queue for managing nodes to be explored.
- **Constructors**:
    - BreadthFirst() – Initializes an empty Breadth First Search.
    - BreadthFirst(Map map) – Initializes Breadth First Search with a specific map.
- **Run Method**: Starts the search by setting up the initial state and adding the root node to the queue.
- **Reset Method**: Clears the open queue and resets the search.
- **Update Method**: Explores nodes by dequeuing the front node, marking it as visited, and adding unvisited neighbors to the queue. The search ends when the target is found or the queue is empty.
- **State Management**: Nodes are assigned states (unvisited, in progress, visited) to track the search progress.
- **Performance**: Breadth First Search is accurate but slower, as it explores all nodes at each depth level before moving deeper.

- **Output**: The algorithm explores all neighbors at the current depth before advancing to the next, ensuring a thorough and systematic search.

Map Class



Overview:

- **Purpose**: Represents the game map, managing the nodes that make up the map grid, including walls and walkable areas.
- **Fields**:
  - map: A 2D array of Node objects representing the map grid.
- **Constructor**:
  - Map(List<Point> wallList, int rows = 31, int cols = 28): Initializes the map with the specified number of rows and columns, and marks the walls based on the given wallList.
- **Properties**:
  - LengthX: Returns the number of columns in the map.
  - LengthY: Returns the number of rows in the map.
- **Methods**:
  - Reset(): Resets the state of all nodes in the map to their initial state.
  - GetNeighbours(Node node): Returns a list of unvisited neighboring nodes of a given node, considering wraparound for the map's edges.
  - TryAdd(List<Node> neighbours, int x, int y): Attempts to add a neighboring node to the list if it is within bounds and unvisited.

- **State Representation**:
    - 0: Impassable
    - 1: Unvisited
    - 2: In frontier
    - 3: Visited
    - 4: Path
- **Output**: The class provides functionality for managing the grid and finding valid neighboring nodes during pathfinding.

Node Class

Overview:

- **Purpose**: Represents a single node (or cell) in the game map, used for pathfinding algorithms to track positions, state, predecessors, and costs.
- **Fields**:
    - _location: The coordinates (X, Y) of the node in the game map.
    - _predecessor: The preceding node in the path, used to trace the path from the target back to the start.
    - _cumulativeCost: The cost associated with reaching this node from the starting point.
    - _state: The current state of the node, representing its status in the pathfinding process.
- **Constructor**:
    - Node(int x, int y, int s): Initializes the node at the specified (X, Y) location with a given state s.
- **Methods**:
    - Reset(): Resets the node's state to unvisited and clears the predecessor and cumulative cost.
- **Properties**:
    - Location: Returns the node's location (X, Y).
    - State: Gets or sets the node's state (e.g., unvisited, visited).
    - Predecessor: Gets or sets the node's predecessor, used for backtracking in pathfinding algorithms.

    - Cost: Gets or sets the cumulative cost to reach this node.
- **State Representation**:
    - 0: Impassable
    - 1: Unvisited
    - 2: In frontier
    - 3: Visited
    - 4: Path
- **Output**: The class is used to manage individual nodes in the game map and support pathfinding by storing essential information such as location, cost, and state.

SearchAlgorithm class

The SearchAlgorithm class is an abstract base for implementing pathfinding algorithms like A*, BFS, and DFS in a Pac-Man-like game. It manages searching for a path from a start node to a target node, providing functionality such as heuristic calculation, path reconstruction, and performance tracking.

Key Features:

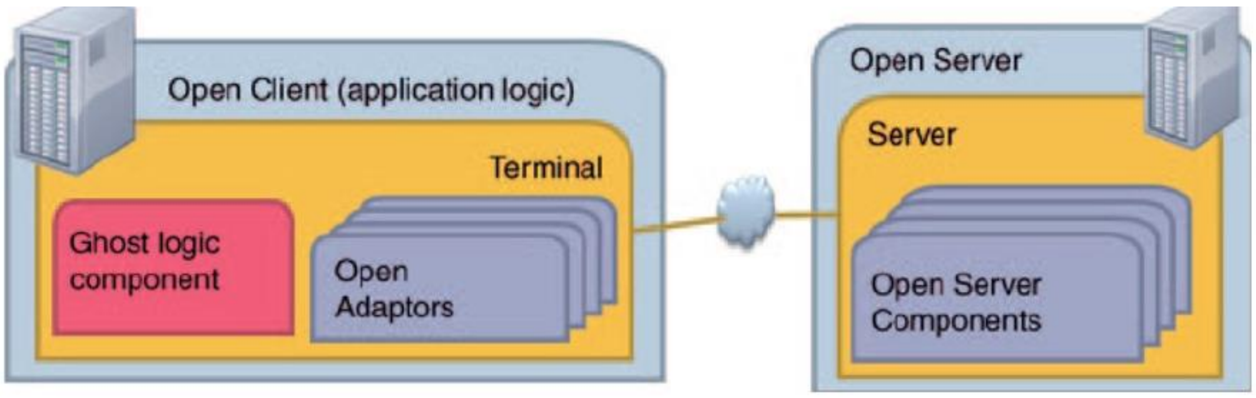
- **Fields**: Includes _map, _stopwatch, _found, _root, _target, _pathLength, and _title.
- **Constructor**: Initializes with a specific map to search on.
- **Heuristic & F-Score**: Calculates Manhattan distance and F-score for pathfinding.
- **Run Method**: Starts the search, reconstructs the path, and returns the movement direction.
- **Search Method**: Runs the search and tracks time.
- **Pathfinding**: Reconstructs the path and marks it with State = 4 (Path).
- **Update Method**: Abstract method to be implemented by subclasses for specific algorithm logic.
- **State Management**: Manages node states (impassable, unvisited, visited, path).

Usage:

A concrete algorithm (e.g., A* or BFS) inherits from SearchAlgorithm and implements the Update() method to define specific search logic.

System Architecture

Client-Server Architecture



- **Frontend (Client):** Manages the game's user interface, player inputs, and rendering.
- **Backend (Server):** Handles game logic, state management, and sound processing.

Frontend (Client)

Technologies

- C#: Windows Forms (WinForms) or Windows Presentation Foundation (WPF) for the UI.
- Rendering Graphics: DirectX or OpenGL.
- Sound Processing: Libraries like NAudio.

Components

1. **Game Window**
   - Main Game Window: Displays the game board, score, and lives.
   - Menu Screens: Includes start, pause, and game-over screens.
2. **Rendering Engine**
   - Tile-Based Rendering: Draws the maze, pellets, ghosts, and Pac-Man.
   - Sprite Management: Manages the sprites for animations.
3. **Input Handler**
   - Keyboard Input: Detects player's key presses to control Pac-Man.
   - Event Handling: Processes events like starting the game or pausing.

4. **Sound Engine**
   - NAudio Library: Manages sound effects and background music.
   - Sound Triggering: Plays sounds based on game events (e.g., eating pellets, ghost encounters).
5. **User Interface**
   - Score Display: Shows the current score.
   - Life Counter: Displays the number of remaining lives.
   - Menus: Provides navigation through different screens (start, pause, game-over).

Backend (Server)

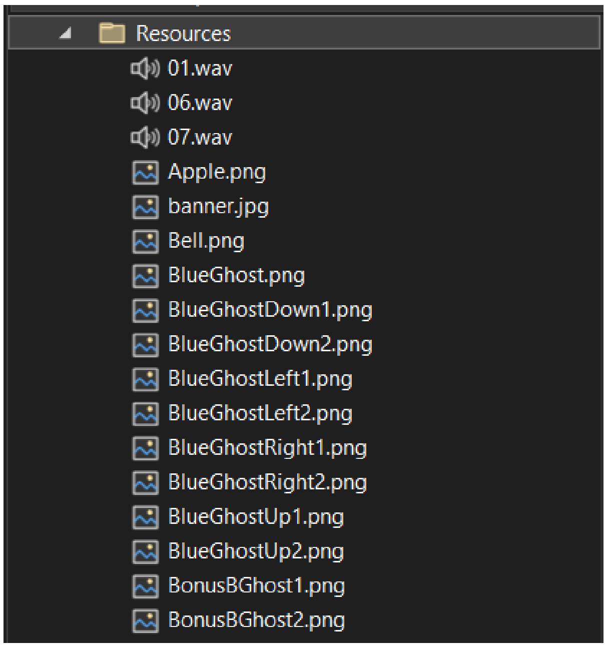
Technologies

- C#: Classes and methods for game logic and state management.

Components

1. **Game Logic**
   - Game Loop: Continuously updates the game state and renders frames.
   - Collision Detection: Checks collisions between Pac-Man, ghosts, and pellets.
   - State Machine: Manages different game states (running, paused, game-over).
2. **AI Engine**
   - Ghost AI: Implements the behavior of the ghosts using algorithms like A* for pathfinding.
   - Finite State Machine: Manages ghost states (chase, scatter, frightened).
3. **Game Objects**
   - Entities: Defines classes for Pac-Man, ghosts, pellets, and walls.
   - Attributes and Behaviors: Manages properties and actions of game entities.
4. **Sound Management**
   - Sound Effects: Plays sounds based on game events.
   - Background Music: Controls looping background music.

Project Resources


Resources
01.wav
06.wav
07.wav
Apple.png
banner.jpg
Bell.png
BlueGhost.png
BlueGhostDown1.png
BlueGhostDown2.png
BlueGhostLeft1.png
BlueGhostLeft2.png
BlueGhostRight1.png
BlueGhostRight2.png
BlueGhostUp1.png
BlueGhostUp2.png
BonusBGhost1.png
BonusBGhost2.png

1. Graphical Resources

- **Sprites:**
    - Pac-Man: pacman.png
    - Ghosts (with directional frames):
        - Blue Ghost: BlueGhost.png, BlueGhost[Direction][Frame].png (e.g., BlueGhostDown1.png)
        - Pink Ghost: PinkGhost.png, PinkGost[Direction][Frame].png
        - Red Ghost: RedGost.png, RedGos[Direction][Frame].png
        - Yellow Ghost: YellowGost.png, YellowGost[Direction][Frame].png
    - Ghost Eyes: eyes[Direction].png
    - Bonus Ghosts: BonusBGhost1.png, BonusBGhost2.png, BonusWGhost1.png, BonusWGhost2.png
- **Pellets and Fruits:**
    - Apple.png, Cherry.png, Melon.png, Key.png, Orange.png, Strawberry.png, Galaxian Boss.png
- **Maze and Environment:**
    - maze2.png, spritesheet.png
- **Icons and Backgrounds:**
    - Game Icon: Pacman 3.ico
    - Background Images: banner.jpg, Wiki-background.jpg

2. Audio Resources

- **Sound Effects:**
    - Pac-Man eating: Pacman_Waka_Waka.wav, wakka_wakka1.wav, wakka_wakka2.wav
    - Ghost interactions:
        - Eating Ghost: Pacman_Eating_Ghost.wav
        - Pac-Man Dies: Pacman_Dies.wav
    - Bonus actions:
        - Eating Cherry: Pacman_Eating_Cherry.wav
        - Extra Life: Pacman_Extra_Live.wav
    - Siren Sounds: Pacman_Siren.wav, siren.wav, siren2.wav
    - Other effects: dot1.wav, dot2.wav, Sound Effect (6).wav
- **Background Music:**
    - Opening Song: Pacman_Opening_Song.wav
    - Intermission Music: Pacman_Intermission.wav

3. UI Resources

- **Backgrounds and Decorative Assets:**
    - Used in start, pause, and game-over screens (e.g., banner.jpg, Wiki-background.jpg).

4. Data Organization

- **Spritesheets:** spritesheet.png for optimized sprite rendering.

Advanced Implementations

1. **Difficulty Selection**

- The game allows users to choose between multiple difficulty levels:
    - **Normal**
    - **Hard**
    - **Original AI**
- Each difficulty level adjusts parameters such as:
    - Ghost behavior algorithms.
    - Speed of Pac-Man and ghosts.

2. Ghost Tracking Algorithms

The game includes advanced algorithms for ghost movement, offering users the ability to select their preferred AI behavior:

- **A* Algorithm**: Uses a heuristic approach to find the shortest path to Pac-Man.
- **Best-First Search:** Prioritizes nodes closer to Pac-Man, optimizing ghost movement.
- **Breadth-First Search:** Explores all possible paths equally, creating varied movement patterns.

3. Dynamic Delay Adjustments

- Players can customize the delay settings for:
    - **Ghost Movement:** Changes the reaction speed of ghosts.
    - **Pac-Man's Movement:** Alters Pac-Man's responsiveness.
- These settings allow fine-tuning for gameplay precision and challenge.

4. Grid Positioning and Maze Logic

- The maze is implemented as a **grid-based system**:
    - Each grid cell corresponds to a specific game element (e.g., walls, pellets, Pac-Man, ghosts).
    - Pathfinding algorithms (like A*) dynamically calculate movement based on grid positions.
- The grid system ensures smooth collision detection and interaction between Pac-Man, ghosts, and pellets.

5. Level Customization

- Levels are designed to adapt based on difficulty and AI settings:
    - The placement of power-ups, fruits, and pellets changes with higher difficulties.
    - Ghost spawn patterns and behavior are tailored to enhance challenge.

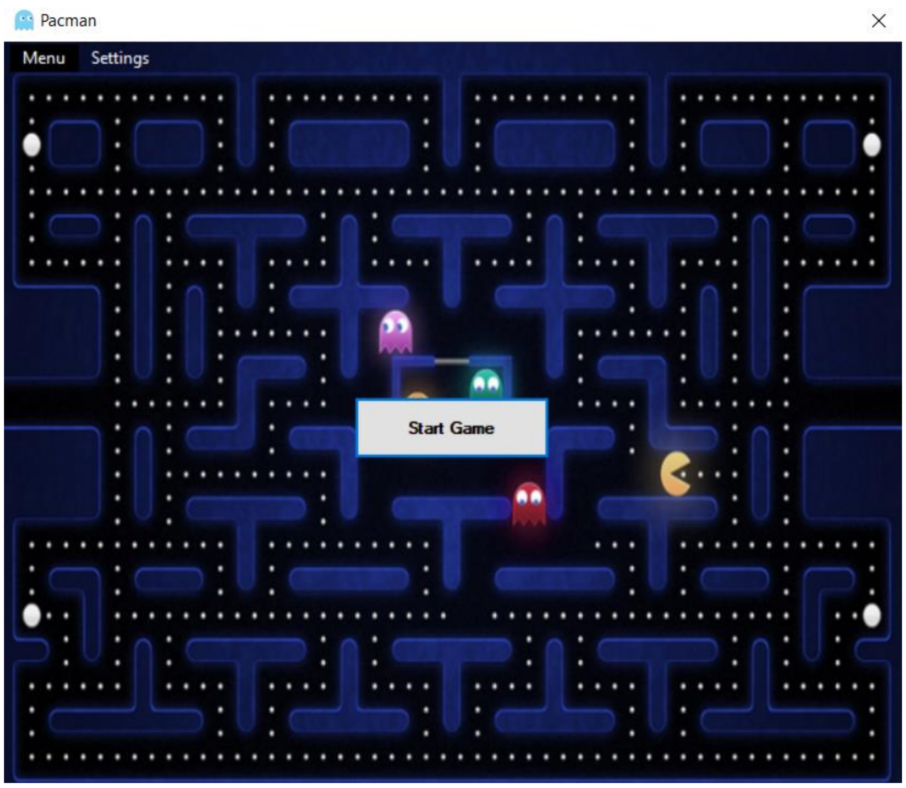
6. Configurable Game Settings

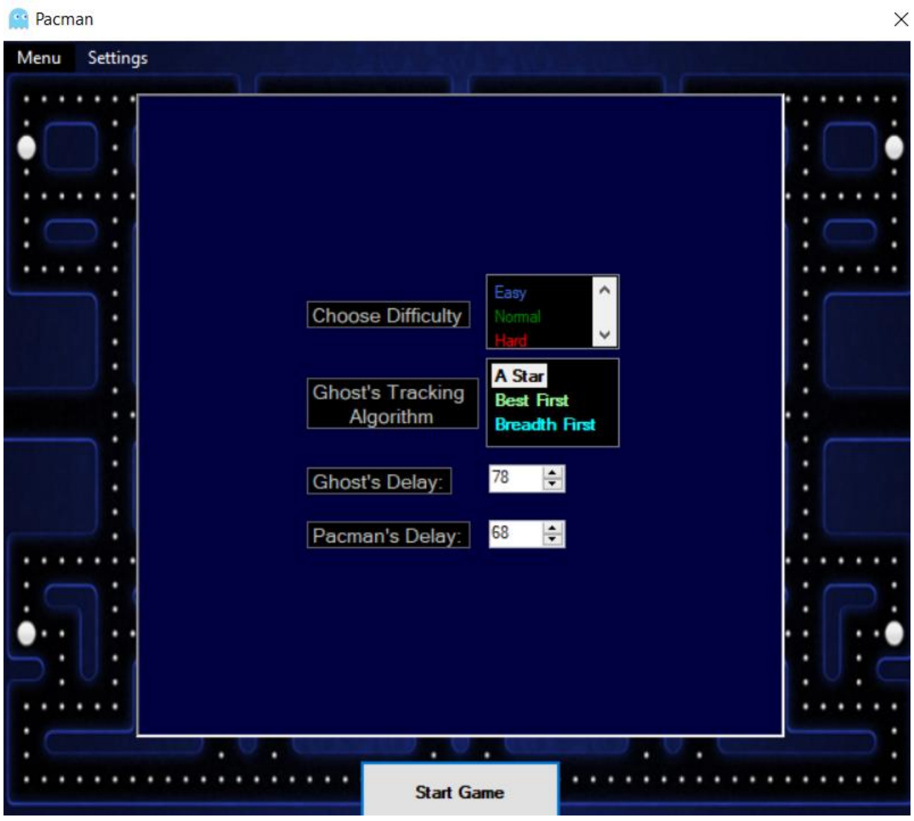
- The settings menu allows players to adjust gameplay preferences without modifying the code:
    - Select algorithms, difficulties, and delays.
    - Offers a user-friendly interface for managing game dynamics.

These advanced implementations enhance gameplay and provide a high degree of customization, catering to both casual and competitive players.

Project Screenshots

1. Main User Interface



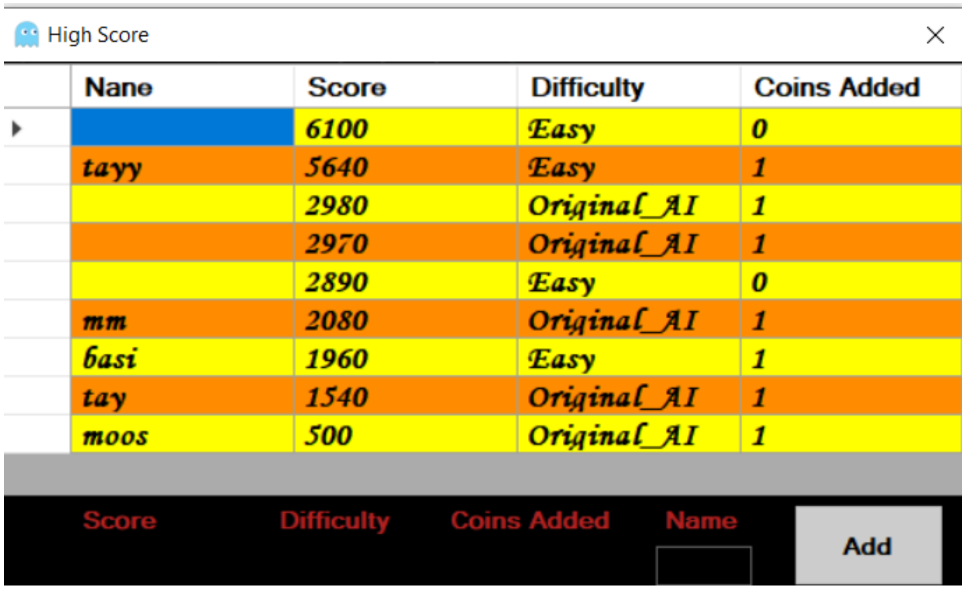


Choose Difficulty: Easy / Normal / Hard

Ghost's Tracking Algorithm: A Star / Best First / Breadth First

Ghost's Delay: 78

Pacman's Delay: 68

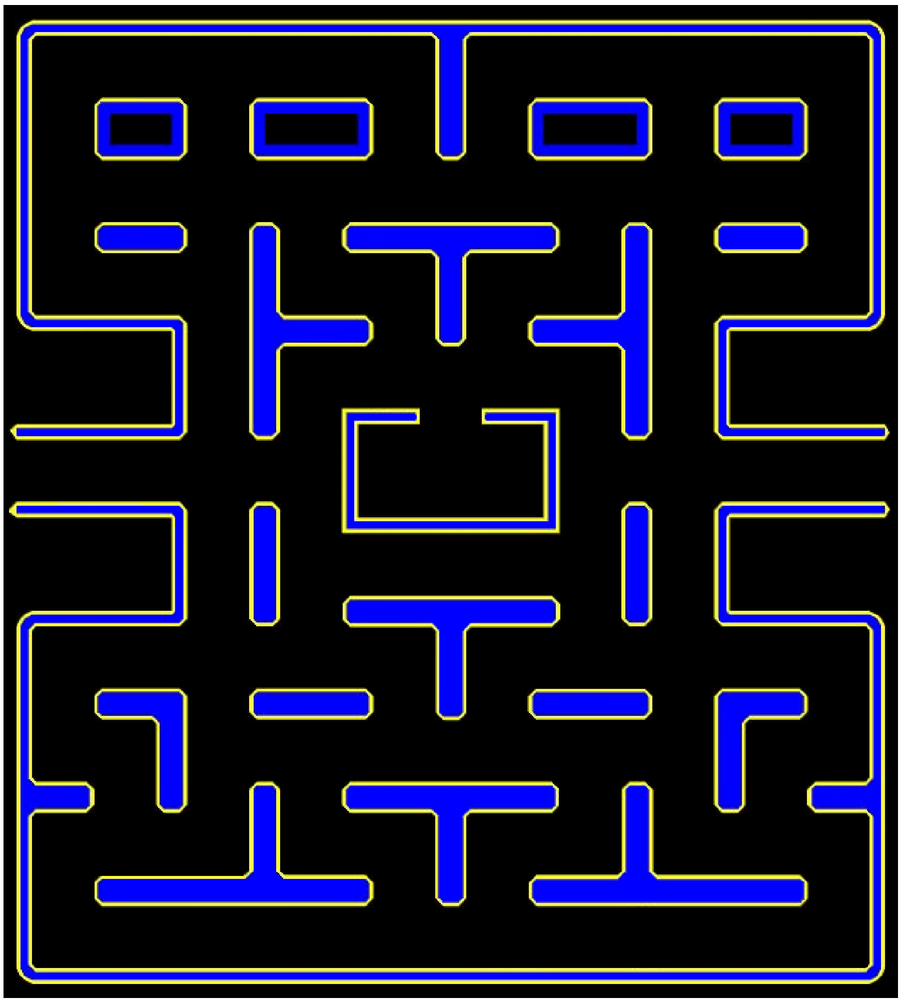


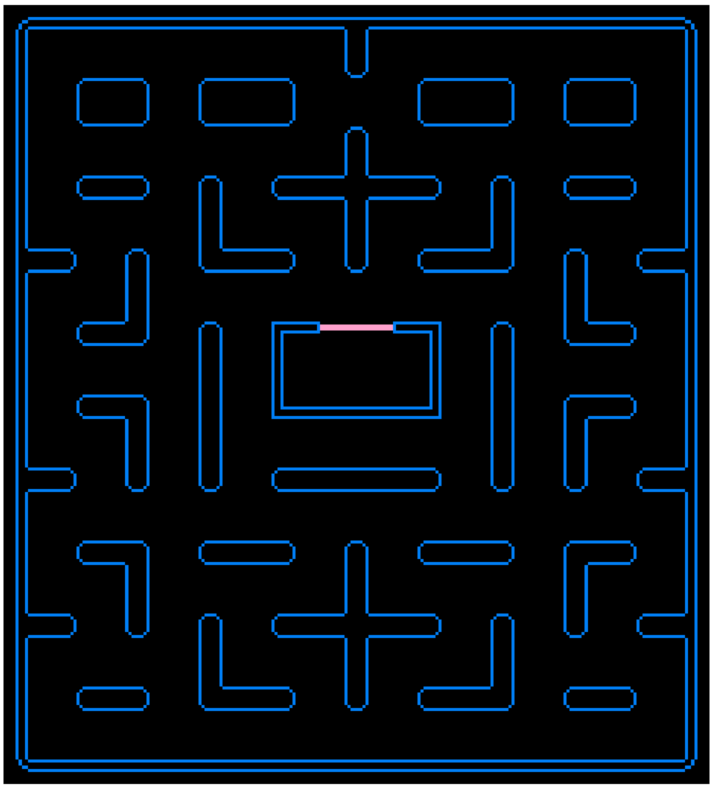
High Score

Nane	Score	Difficulty	Coins Added
	6100	Easy	0
tayy	5640	Easy	1
	2980	Original_AI	1
	2970	Original_AI	1
	2890	Easy	0
mm	2080	Original_AI	1
basi	1960	Easy	1
tay	1540	Original_AI	1
moos	500	Original_AI	1

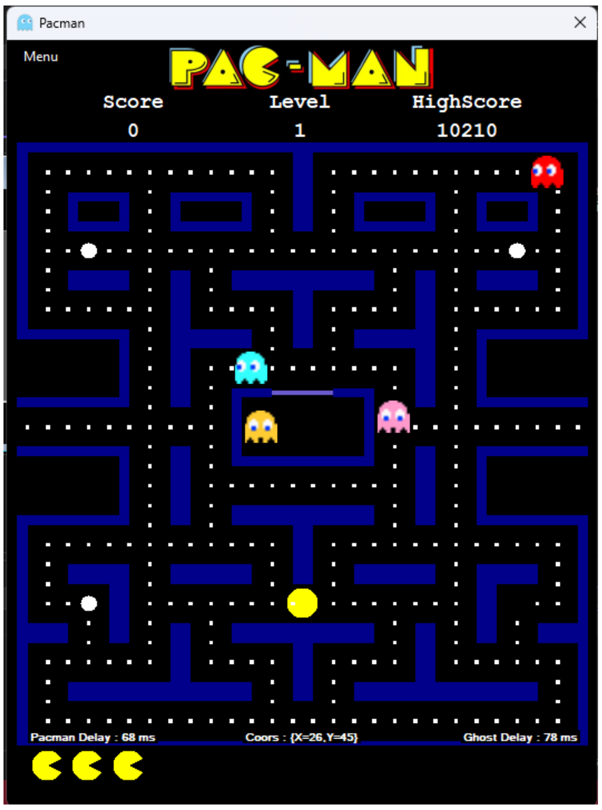
2. **Maze**

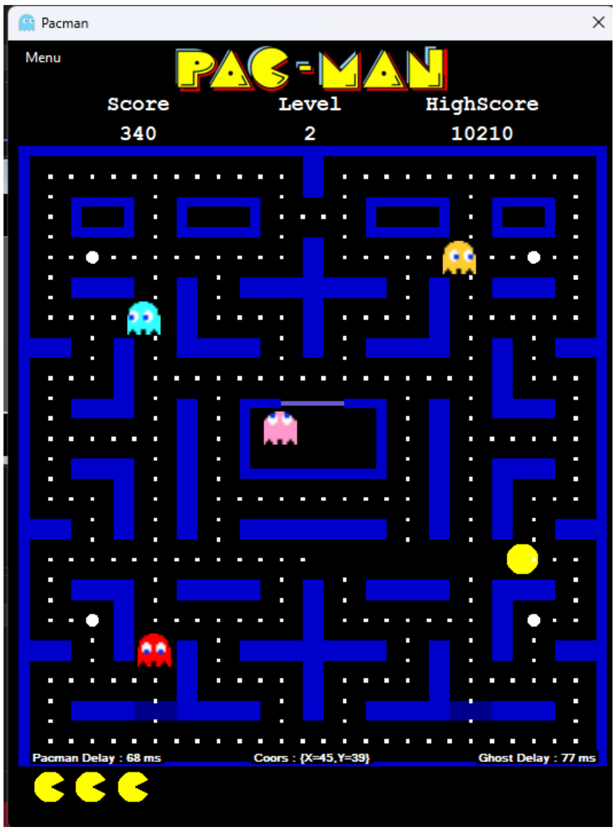
•

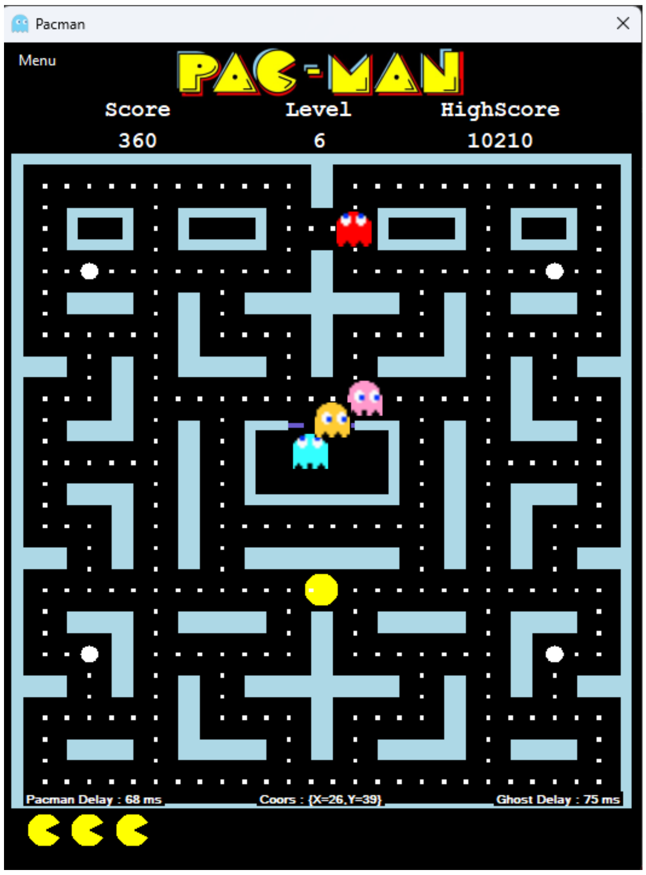


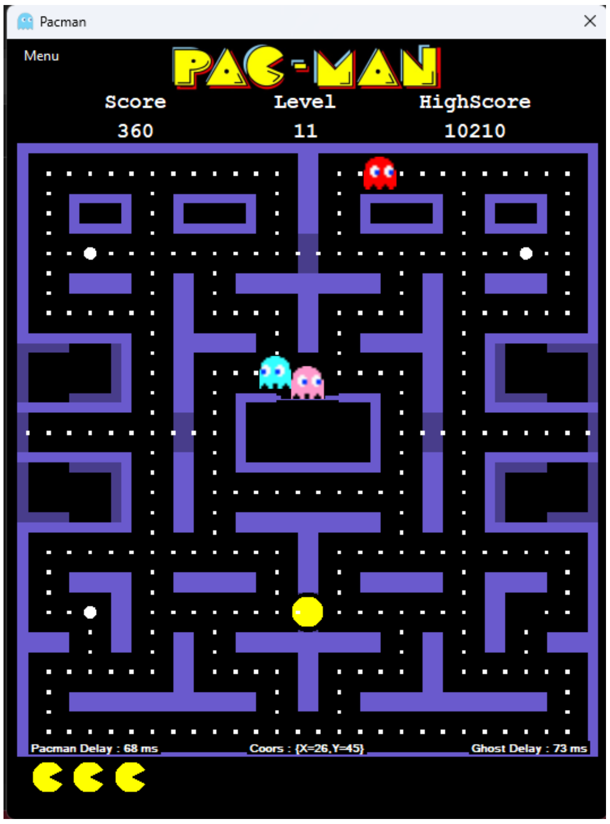


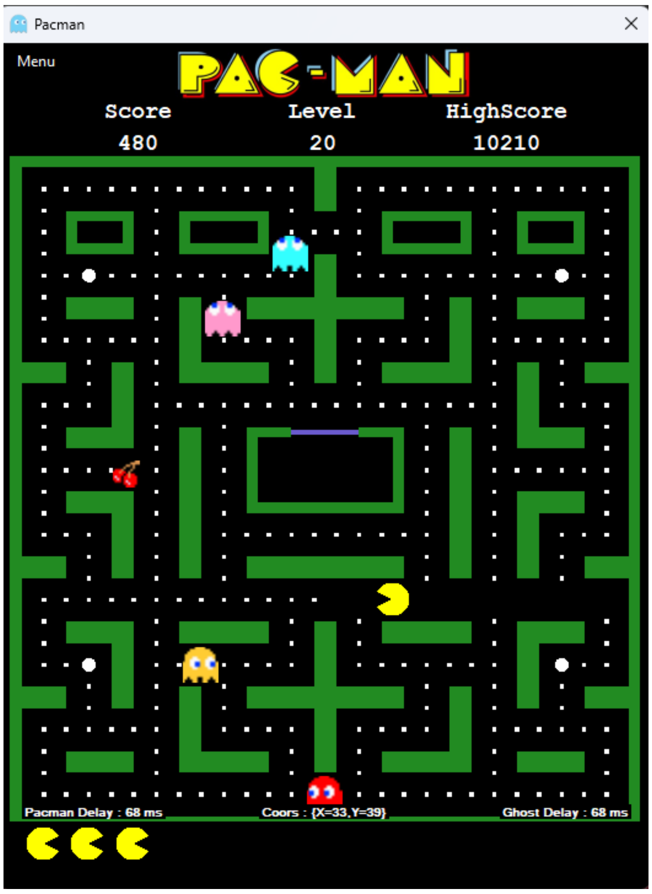
3. Levels

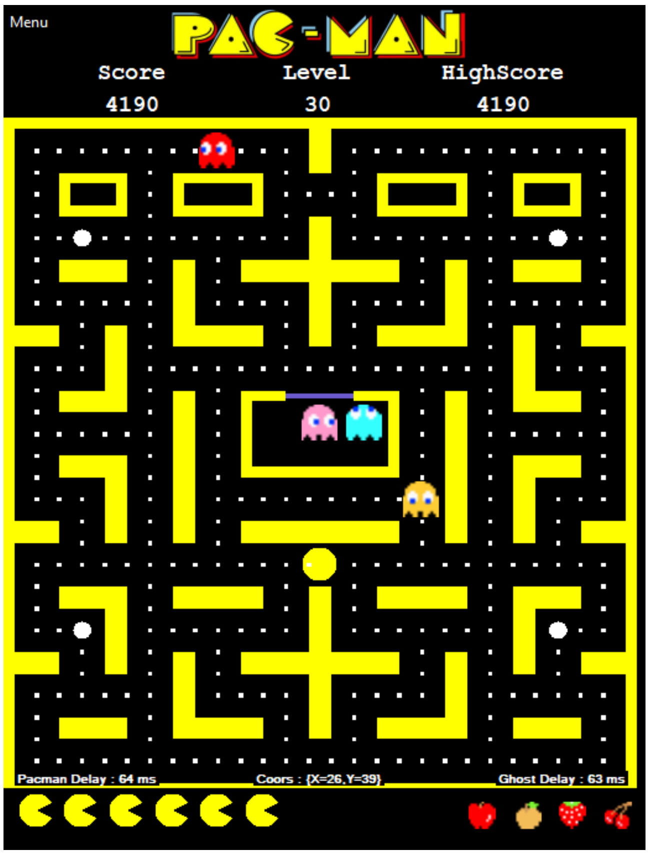

Pacman
Menu
PAC-MAN
Score
0
Level
1
HighScore
10210
Pacman Delay : 68 ms
Coors : {X=26,Y=45}
Ghost Delay : 78 ms

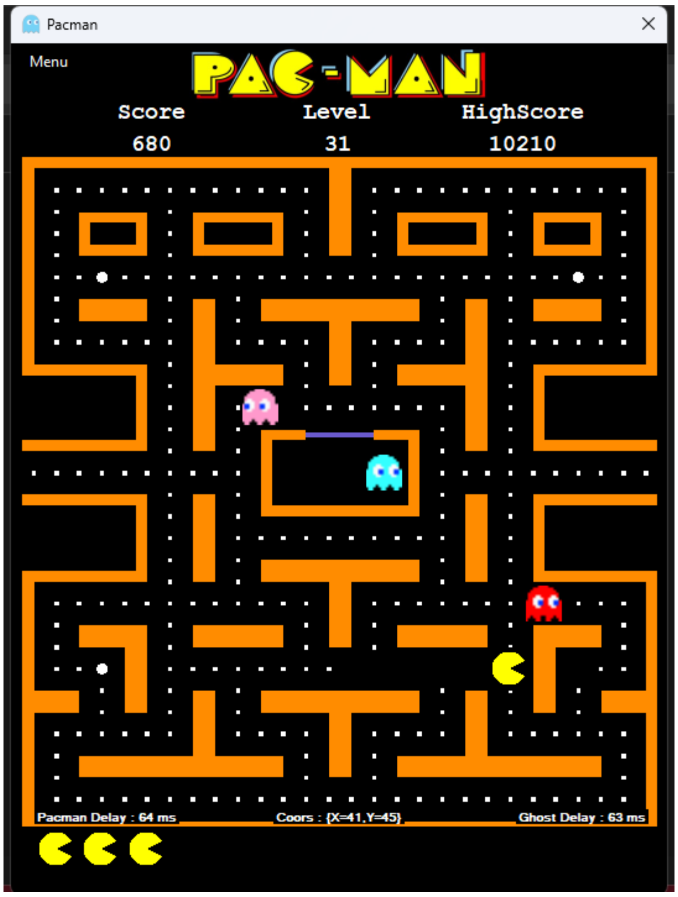

Pacman
Menu
PAC-MAN
Score
340
Level
2
HighScore
10210
Pacman Delay : 68 ms
Coors : {X=45,Y=39}
Ghost Delay : 77 ms


Pacman
Menu
PAC-MAN
Score
360
Level
6
HighScore
10210
Pacman Delay : 68 ms
Coors : {X=26,Y=39}
Ghost Delay : 75 ms


Pacman
Menu
PAC-MAN
Score
360
Level
11
HighScore
10210
Pacman Delay : 68 ms
Coors : {X=26,Y=45}
Ghost Delay : 73 ms


Pacman
Menu
PAC-MAN
Score
480
Level
20
HighScore
10210
Pacman Delay : 68 ms
Coors : {X=33,Y=39}
Ghost Delay : 68 ms


Menu
PAC-MAN
Score
4190
Level
30
HighScore
4190
Pacman Delay : 64 ms
Coors : {X=26,Y=39}
Ghost Delay : 63 ms


Pacman
Menu
PAC-MAN
Score
680
Level
31
HighScore
10210
Pacman Delay : 64 ms
Coors : {X=41,Y=45}
Ghost Delay : 63 ms

Conclusion

The Pac-Man project is a comprehensive implementation of a classic arcade game that integrates advanced computational techniques and user-centric design principles. This project has successfully combined engaging gameplay mechanics with technical sophistication, providing players with an experience that balances entertainment and challenge. Through careful planning, algorithmic design, and resource optimization, the game effectively demonstrates the potential of modern programming practices in recreating timeless games.

Technical Achievements

One of the standout features of this project is the implementation of dynamic ghost tracking algorithms, such as A*, Best-First Search, and Breadth-First Search. These algorithms provide a high degree of adaptability and realism in ghost behavior, creating a challenging environment for players. The ability to select between these algorithms further enhances user customization, allowing players to experience the game at varying levels of complexity.

Additionally, the grid-based maze system ensures precision in pathfinding and collision detection, enabling smooth interactions between game elements like Pac-Man, ghosts, pellets, and walls. The use of this grid system not only simplifies the underlying logic but also ensures scalability for future level designs or additional game features.

The project also integrates a dynamic settings menu that allows players to adjust key gameplay parameters, such as difficulty levels, ghost delays, and Pac-Man's speed. This flexibility caters to a wide range of players, from casual gamers to competitive enthusiasts, and demonstrates the importance of user-friendly interfaces in game development. The inclusion of difficulty modes (Normal, Hard, and Original AI) further highlights the game's adaptability, ensuring replayability and sustained user engagement.

Resource Utilization

The game leverages high-quality graphical and audio resources to recreate the nostalgic essence of the original Pac-Man while introducing modern enhancements. The use of spritesheets and sound libraries like NAudio ensures efficient rendering and immersive audio feedback. These resources, combined with an optimized rendering engine, create a visually appealing and acoustically engaging experience that resonates with players.

Educational and Practical Value

From an educational perspective, this project demonstrates the application of theoretical concepts such as artificial intelligence, pathfinding, and state machine management in a real-world scenario. It bridges the gap between academic learning and practical implementation, showcasing how algorithms can be adapted to solve game design challenges. Moreover, the project's modular design serves as a blueprint for aspiring developers, illustrating the importance of clean code architecture and resource management.

Future Enhancements

The modular nature of the project opens up numerous opportunities for expansion. Future enhancements could include additional levels, multiplayer functionality, or advanced AI algorithms for ghost behavior. Integration of modern rendering techniques, such as particle effects or lighting, could further enhance the game's visual appeal. Additionally, adapting the game for mobile platforms or virtual reality could broaden its accessibility and user base.

Another area for improvement could be the inclusion of a level editor, allowing players to design and share custom mazes. This feature would encourage community engagement and creativity, as users could create unique challenges and gameplay experiences. Moreover, integrating leaderboards or achievement systems could further enhance the game's replayability and foster a competitive spirit among players.