# JPacMan Technical Report

JPacMan is a Java-based implementation of the classic Pac-Man arcade game, developed using Java 24 and Java Swing. This report provides an in-depth technical analysis of the project's architecture, design patterns, and game mechanics.

## Table of Contents



## 1. Overview

JPacMan is structured as a modular Java application with a clear separation of concerns following the Model-View-Controller (MVC) architectural pattern. The project successfully uses inheritance and polymorphism to model game entities through a shared abstraction while allowing specialized behaviors in derived classes. This approach enables Pac-Man and the ghosts to share common attributes and methods defined in the base `Character` class, while Pachman and Ghost overrides methods to implement distinct movement patterns and behaviors.

The game state is represented as a 2D matrix of strings (`String[][]`), where each string encodes the entities present at a specific coordinate on the board. This design supports multiple entities occupying the same position simultaneously by concatenating their identifier characters into a single string.

The core game loop runs at a fixed interval of 300 milliseconds (approximately 3 times per second) using `javax.swing.Timer`; this timer-based approach provides a predictable and smooth gameplay experience while maintaining simplicity in the implementation.

JPacMan demonstrates several key object-oriented design principles:

- **Encapsulation**: Game state and logic are properly encapsulated within their respective classes

- **Inheritance**: The `Character` abstract class provides a common foundation for `PacMan` and `Ghost` subclasses

- **Polymorphism**: The `CharacterActions` interface defines a contract that both player and enemy characters must fulfill

- **Separation of Concerns**: Clear boundaries between model, view, and controller components

# 2. Project File Structure

The JPacMan project follows a well-organized directory structure that separates code from resources:

```
JPacMan/
├── src/
│ ├── scripts/ # Java source files
│ │ ├── Main.java # Application entry point
│ │ ├── Game.java # Core game controller
│ │ ├── GUI.java # Graphical user interface
│ │ ├── Character.java # Abstract base class for entities
│ │ ├── CharacterActions.java # Interface for character behaviors
│ │ ├── PacMan.java # Player character implementation
│ │ ├── Ghost.java # Enemy AI implementation
│ │ ├── UserInput.java # Keyboard input handler
│ │ ├── GameEvents.java # Event processing and game logic
│ │ ├── MatrixFromFileExtractor.java # Map loading utility
│ │ ├── SpritesLoader.java # Asset management for sprites
│ │ └── SoundPlayer.java # Audio playback system
│ ├── Files/ # Game data files
│ ├── Sprites/ # Image assets (.png, .gif)
│ ├── Sounds/ # Audio files (.wav)
│ └── TileMap.txt # Game board layout definition
├── JPacManUmlClassesDiagram.pdf # Class diagram documentation
├── TECHNICAL_REPORT.pdf # Technical documentation (PDF)
└── README.md # Project overview and instructions
```

Here is the consolidated technical documentation for the JPacMan project, structured by the Model-View-Controller (MVC) architectural pattern.

# 3. Model-View-Controller architecture and Class Roles

The JPacMan project implements an MVC-inspired architecture to separate data structures, presentation logic, and input handling. This organization enhances modularity, maintainability, and testability by decoupling the internal game mechanics from the user interface.

---

### 1. Model Layer

The **Model** encapsulates the game state, entities, and core logic. It operates independently of the user interface, strictly defining *what* the game is and how its rules function.

- **Game.java**
  - **Role:** The central hub managing the game loop, state, and coordination.
  - **Responsibilities:** Maintains references to all major objects (PacMan, ghosts, GUI) and manages state variables like score, lives, and the game board matrix. It orchestrates the execution order of operations in every frame.
- **Character.java**
  - **Role:** Abstract base class for all moving entities.
  - **Responsibilities:** Defines common attributes (current coordinates [x,y], starting position, direction) and provides getters/setters. It implements the CharacterActions interface to enforce movement contracts.
- **CharacterActions.java**
  - **Role:** Interface defining essential character behaviors, that must be defined later
  - **Responsibilities:** Declares checkCollisionAndMove() for logic verification and teleportAt() for instant position changes, ensuring polymorphic behavior across different entity types.
- **PacMan.java**
  - **Role:** Player-controlled character implementation (extends Character).
  - **Responsibilities:** Handles collision detection for consumables (dots, fruits, power-ups), validates moves against walls, and updates the board matrix. It triggers sound effects and score updates upon item collection.
- **Ghost.java**
  - **Role:** Enemy AI implementation (extends Character).
  - **Responsibilities:** Implements autonomous movement logic using java.util.Random. It filters valid directions to prevent wall collisions and ensures realistic pursuit by preventing backward movement. Each ghost is identified by a unique color letter.
- **GameEvents.java**
  - **Role:** Logic coordinator for specific game occurrences.
  - **Responsibilities:** Manages ghost spawning sequences, portal teleportation, and win/loss conditions. It also handles the "invincibility mode" logic when power-ups are consumed.
- **MatrixFromFileExtractor.java**
  - **Role:** Data persistence and loading utility.
  - **Responsibilities:** parses TileMap.txt into a 2D string array using BufferedReader, creating the initial game board state and supporting map resets via deep copying.

### 2. View Layer

The **View** observes the Model and renders the game state visually and audibly. It contains no gameplay logic, focusing solely on presentation.

- **GUI.java**
  - **Role:** Main graphical user interface (extends `JFrame`).
  - **Responsibilities:** Renders the 21x21 grid using a `GridLayout`, displays score/lives, and updates sprite visuals (e.g., directional Pac-Man, weakened ghosts). It refreshes the display by re-adding components each frame.
- **SpritesLoader.java**
  - **Role:** Asset management system.
  - **Responsibilities:** Loads image resources into a `HashMap<String, ImageIcon>` for efficient lookup, mapping character identifiers to specific sprites (including directional variations).
- **SoundPlayer.java**
  - **Role:** Audio playback system.
  - **Responsibilities:** Manages asynchronous audio threads for sound effects (WAV format) to ensure the game loop remains unblocked during playback.

### 3. Controller Layer

The **Controller** bridges the gap between the user and the system, interpreting inputs and translating them into Model updates.

- **UserInput.java**
  - **Role:** Input handler (implements `KeyListener`).
  - **Responsibilities:** Captures arrow key presses, translates them into direction vectors, and validates them against the local board reference before updating the player's direction.
- **Main.java**
  - **Role:** Application entry point.
  - **Responsibilities:** Bootstraps the game by instantiating the `Game` class and setting up the initial environment.

---

**Architectural Benefits**

- **Separation of Concerns:** Distinct responsibilities for each layer simplify navigation.
- **Maintainability:** AI logic changes in the Model do not break the GUI code.
- **Scalability:** New characters (Model) or themes (View) can be added without refactoring core logic.
- **Testability:** Core game rules can be unit-tested without initializing the graphical interface.

# 4. How the Game Works :

When the `Main` class is executed, it instantiates a `Game` object, which serves as the central coordinator for the entire game.

## 4.1 Game initialization process

1.The game board matrix is loaded from `TileMap.txt` using `MatrixFromFileExtractor`
2.Sprite assets are loaded into a HashMap using `SpritesLoader`
3. A `PacMan` object is created at position `[10, 19]` with initial direction `[0, 0]` (stationary)
4. An empty `Ghost` array of size 4 is initialized (ghosts spawn later)
5. The `GUI` is instantiated and displayed
6. A `UserInput` handler is created and attached as a key listener to the GUI
7. A `GameEvents` processor is instantiated to handle game logic events 8. The game clock (timer) is started with a 300ms interval

## 4.2 Game Loop Architecture :

JPacMan three themes every second executes the following sequence of operations:

```
1. Update Display (Score & Lives)
2. Spawn Ghosts (if cooldown expired)
3. Move Pac-Man (also picks up collectibles)
4. Check Game Over (if the player collides with a
ghost)
5. Move All Ghosts
6. Check Game Over (if player collides with a ghost)
7. Decrement Cooldowns (invincibility and ghost spawner)
8. Process Portal Teleportation
9. Refresh Screen (GUI update)
10. Check Victory Condition and Map Reset
```

## 4.3 How Pac-Man is Moved :

Pac-Man's movement system consists of two distinct phases: **direction input validation** and **actual movement execution**.

### -Direction Input Validation (UserInput & PacMan)

When the player presses an arrow key:

1.**Input Capture**: The `UserInput` class's `keyPressed()` method captures the key event
2. **Direction Translation**: The arrow key is translated into a direction vector:
    -UP: `{0, -1}` (negative Y moves up in the matrix)
    -DOWN: `{0, 1}` (positive Y moves down)
    -LEFT: `{-1, 0}` (negative X moves left)
    -RIGHT: `{1, 0}` (positive X moves right)

3.**Path Validation**: The `verifyDirectionUpdate()` method checks if the target tile is not a wall

4. **Direction Update**: If the path is clear, Pac-Man's direction is updated the next tick packman will move in that direction

This validation approach creates smooth, responsive controls. Players can input a direction change while Pac-Man is still moving in another direction, and the new direction will be applied on the next valid movement opportunity. This is crucial for navigating tight corners and making quick directional changes.

### Movement Execution (PacMan.checkCollisionAndMove)

During each game loop iteration, Pac-Man attempts to move in its current direction:

1.**Target Tile Identification**: Calculate the target tile position:

2. **Collectible Processing**: Before moving, check and process any collectibles on the target tile:
    - **Food (`.`)**: Increases score by 2 points, plays eating sound
    - **Power-up (x)**: Grants 30 frames (10 seconds) of invincibility, plays
    - power-up sound
    - **Fruit (`f`)**: Increases lives by 1, plays fruit eating sound

3. **Collision Detection**: Check if the target tile is passable (not a wall, spawn zone, or portal marker):

4. **Position Update**: If the path is clear:
    - Remove Pac-Man's identifier (`"P"`) from the current position
    - Add `"P"` to the target position
    - If the old tile becomes empty, replace with a space character (`" "`)
    - Update Pac-Man's coordinate array to reflect the new position

5. **Responsive turns** : If the target tile is blocked, Pac-Man retains the current direction. This allows Pac-Man to enter openings when they become available.

## 4.4 How Ghosts are Moved and Spawned

The ghost system in JPacMan consists of two key components: a **sequential spawn mechanism** and a **random pathfinding algorithm**.

### Ghost Spawning System

Ghosts are spawned sequentially rather than all at once, ghost spawning is handled by the class GameEvents. Here's how the spawn sequence works:

1. **Initial Delay**: When the game starts, `ghostSpawnerCooldown` is set to 18 frames (6 seconds) e
2. **Warning Message**: When no ghosts exist, the GUI displays "Ghosts are Coming, HURRY!" to warn the player

3. **Sequential Release**: When the cooldown reaches 0, ghosts spawn one at a time in this order:
- **Red Ghost (r)**: Spawns first at position `[10, 13]` moving right `{1, 0}`
- **Pink Ghost (p)**: Spawns second at the same position moving left `{-1, 0}`
- **Orange Ghost (o)**: Spawns third at the same position moving right `{1, 0}`
- **Blue Ghost (b)**: Spawns last at the same position moving left `{-1, 0}`

4. **Cooldown Reset**: After each spawn, the cooldown is reset to 18 frames (6 seconds)

**Spawn Events:** - Ghosts respawn after Pac-Man loses a life - Ghosts respawn after completing a level (victory) - Defeated ghosts (during invincibility mode) are removed from the array and respawn after the full cooldown cycle

### Ghost Movement Algorithm:

Each ghost implements a simple but effective random pathfinding AI that creates unpredictable behavior while maintaining realistic movement constraints.
**Movement Logic:**

1. **Direction Constraints**: Ghosts determine valid directions based on their current heading to prevent backward movement:

    -Moving Up `{0, -1}`: Can go forward (up), left, or right
    -Moving Down `{0, 1}`: Can go forward (down), left, or right

-Moving Left `{-1, 0}`: Can go forward (left), up, or down
-Moving Right `{1, 0}`: Can go forward (right), up, or down

2.**Wall Detection**: The ghost filters out any direction that would lead to a wall tile (`"W"`):

3. **Random Selection**: From the remaining valid directions, the ghost randomly chooses one:

4. **Position Update**: The ghost updates the game board matrix:

-Remove the ghost's color identifier from the old position
-Append the color identifier to the target position (allowing multiple entities on one tile)
-Update the ghost's coordinate array

Ghosts never reverse direction - Ghosts can overlap with food, power-ups, and even Pac-Man (enabling collisions) . Each ghost has an independent random generator, creating varied behavior. Collision with pac man reduces Pac-Man's lives by 1 and resets all characters, but when pacman is invincible Collision removes the ghost from the game and awards 200 points

## 4.5 How the Map Works

The map system in JPacMan is built on a string-based matrix representation that elegantly handles multiple entities occupying the same position while maintaining simplicity and efficiency.

**File Format (TileMap.txt):** The game board is defined in a text file with space separated character codes:

```
W W W W W W W W W W W W W W W W W W W W W
W . . . . . W . . x . . . . . . W . . . . W
W . W W W . W . W W W W W W W . W . W W . W
```

**Each character represents a specific tile:**

**- `W`: Wall (impassable)**

**- `.` : Food pellet (collectible)**

**- x: Power-up (grants invincibility)**

**- `f`: Fruit/cherry (grants extra life)**

**- 0: Portal A (left side teleporter)**

**- `0`: Portal B (right side teleporter)**

**- (space): Empty passable tile**

### Map Loading and Initialization :

1. `MatrixFromFileExtractor.MatrixExtractor()` reads the file using `InputStream` and `BufferedReader`

2. Each line is split by spaces to create a row array

3. Rows are assembled into an `ArrayList<String[]>` during reading

4. The ArrayList is converted to a `String[][]` array for faster access

5. A deep copy of the original map is stored for reset functionality

This approach allows the game to reset the map without re-reading the file, improving performance during level restarts and victories.

## Dynamic Entity Representation :

The map matrix serves dual purposes:

1. **Static Environment**: Walls, portals, and spawn zones remain constant

2. **Dynamic State**: Character positions and collectible status are continuously updated

**3. Multi-Entity Support:** The string-based approach allows multiple entities on a single tile through string concatenation: - A ghost on a food tile: `"r."` (red ghost + food) - Pac-Man on an empty tile: `"P"` - An empty tile after food collection: `" "`

When entities move: - Their identifier is removed from the source tile using `replace(identifier, "")` - Their identifier is added to the target tile (appended or standalone) - Empty tiles are represented as `" "` to maintain matrix consistency

## Map Reset Mechanism :

When Pac-Man completes a level (all food collected):

1. `Game.resetGameBoard()` retrieves the stored deep copy of the original map

2. The game board reference is replaced with a fresh copy

3. All food pellets are restored to their original positions

4. Power-ups are reset

5. A special fruit tile is spawned as a bonus reward

## How Victory and Defeat Happens

JPacMan implements clear win and lose conditions with appropriate state resets and player feedback.

### Victory Condition

**Trigger:** Victory is achieved when Pac-Man collects all food pellets (`.` characters) from the game board.

**Detection Process:** The `GameEvents.checkVictory()` method scans the entire game board matrix after each frame:

```
boolean victoryArchieved = true;
for (String[] row : gameBoard) {
    for (String element : row) {
        if (element.contains(".")) {
            victoryArchieved = false;
            break;
        }
    }
}
```

This nested loop examines every tile, checking if any contain the food character (`.`). If even one food pellet remains, victory is not yet achieved.

### Victory Sequence:

When all food is collected:

1. **Character Reset**: Pac-Man is teleported back to starting position

2. **Direction Reset**: Pac-Man's direction is set to `{0, 0}` (stationary) 3. **Ghost Removal**: All ghosts are removed from the board and the ghost array is cleared

4. **Cooldown Reset**: Ghost spawner cooldown is reset to 18 frames (6 seconds)

5. **Map Reload**: The game board is reset from the stored original map copy

6. **Bonus Spawn**: A fruit tile (`f`) is spawned

7. **Visual Feedback**: The lives label displays "YOU WIN, Congrats!"

8. **Audio Feedback**: The victory sound effect plays

### Defeat Conditions

**Trigger:** Defeat occurs when Pac-Man collides with a ghost while not in invincibility mode and runs out of lives.

**Collision Detection:** The `GameEvents.checkGameOver()` method compares coordinates after every character movement:

```
int[] pacmanCoordinnatesXY = pacman.getCoordinatesXY();
for (Ghost ghost : ghosts) {
     if (ghost != null) {
          int[] ghostCollisionCoordinatesXY = ghost.getCoordinatesXY(); if
          (Arrays.equals(pacmanCoordinnatesXY, ghostCollisionCoordinatesX// Collision detected
          }
     }
}
```

## Collision Detection Logic and Collision events

To ensure precision, the game executes collision checks **twice per frame**: once immediately after Pac-Man moves, and again after the ghosts move.

This dual-check system prevents a bug where Pac-Man and a ghost could swap tiles simultaneously without registering contact.

**1. Standard Collision (Player Damage)** When Pac-Man collides with a ghost while vulnerable:

- **Penalty:** The player loses one life, and the defeat sound effect plays. If the player has no lives the game loop is halted and the game ends.
- **State Reset:** Pac-Man is teleported to the starting coordinate `[10, 19]` with his movement direction reset to stationary (`{0,0}`).
- **Ghost Reset:** All active ghosts are cleared from the board. The spawner cooldown is reset to

provide the player a brief safe period before ghosts reappear.

**2. Invincibility Mode (Ghost Defeat)** When Pac-Man collides with a ghost while a power-up is active:

- **Reward:** The player earns **200 points**, and the specific ghost defeat sound plays.
- **Outcome:** The ghost is removed from the board and added back to the respawn queue. Pac-Man takes no damage and continues his current trajectory.

## 4.6 How GUI is Updated

The GUI update system in JPacMan uses a complete redraw approach, refreshing the entire display every frame to reflect the current game state accurately.

### GUI Architecture

The `GUI` class extends `JFrame` and organizes the display into three main sections:

- **Top Panel**: Displays score and lives using `JLabel` components
  Left side: Score label (`"Score: X"`)
  Right side: Lives label (`"Lives: X"`)
  Background: Black with white text for classic arcade aesthetics

-**Game Board Panel**: A 21×21 `GridLayout` containing `JLabel` elements

-Each grid cell represents one tile from the game board matrix

-Labels contain `ImageIcon` objects for visual representation
-Background: Black to match the classic Pac-Man aesthetic

-**Content Panel**: A `BorderLayout` container organizing the top panel and game board
**Update Cycle**

### GUI Refresh Loop

The GUI update process occurs in three distinct phases during each game loop iteration:

- **Phase 1: Score and Lives Update (Before Gameplay)**

```
userGui.updateScoreDisplay();
userGui.updatesLifesDisplay();
```

These methods query the `Game` class for current values:

- `updateScoreDisplay()`: Retrieves score via `Game.getScore()` and updates the score label

- `updatesLifesDisplay()`: Retrieves lives via `Game.getLives()` and updates the lives label

This update happens at the beginning of each frame to ensure the display reflects any points gained or lives lost in the previous frame.

- **Phase 2: Game Board Refresh (After All Game Logic)**

```
userGui.refreshGameScreen(gameBoard, spriteMap, PacMan);
```

—

1. **Clear Existing Display**:

```
gameBoardDisplayJPanel.removeAll();
```

All existing `JLabel` components are removed from the game board panel.

2. **Iterate Through Game Board Matrix**: The method processes each tile in the 21×21 matrix row by row:

```
for(String[] row : gameBoard) {
    for(String string : row) {
        char firstChar = string.charAt(0);
        String firstCharString = String.valueOf(firstChar);
        // Process tile...
    }
}
```

3. **Pac-Man Rendering (Directional Sprites)**: When a `"P"` character is detected, the method determines which sprite to display based on Pac-Man's current direction:

```
int[] actualDirection = pacMan.getcurrentDirectionXY();
if(Arrays.equals(actualDirection, new int[]{0, -1})) {
// Display pacmanUp.gif
} else if(Arrays.equals(actualDirection, new int[]{0, 1})) {
    // Display pacmanDown.gif
} else if(Arrays.equals(actualDirection, new int[]{1, 0})) {
    // Display pacmanRight.gif
} else if(Arrays.equals(actualDirection, new int[]{-1, 0})) {
    // Display pacmanLeft.gif
}
```

This creates the visual effect of Pac-Man facing his movement direction. The sprites are animated GIFs, the classic pacman animation.

4. **Ghost Rendering (Invincibility Visual Feedback)**: When a ghost identifier is detected (`r`, p, o, b), the method checks invincibility status:

```
int invincibleModeCooldown = Game.getInvincibility();
if(invincibleModeCooldown > 0 && (firstCharString.equals("b") ||
    ...))// Display weakened ghost sprite
} else {
    // Display normal ghost sprite
}
```

During invincibility mode, all ghosts are rendered with a blue "scared" sprite instead of their normal colors, providing immediate visual feedback that ghosts are vulnerable.

5. **Standard Tile Rendering**: For all other tiles, the method looks up the corresponding sprite from the sprite map:

6. **Component Refresh**: After all tiles are added, the panel is refreshed:

```
gameBoardDisplayJPanel.revalidate();
gameBoardDisplayJPanel.repaint();
```

- `revalidate()`: Recalculates the layout and positions of all components
- `repaint()`: Triggers a visual redraw of the panel

**Phase 3: Special Message Updates (Event-Driven)**

The lives label serves double duty as a status message display: - `updateLifesLabelText(String text)`: Allows custom messages to replace the lives count
Used for: - `"Ghosts are Coming, HURRY!"`: Warning before first ghost spawn - `"YOU WIN, Congrats!"`: Victory message - `"GAME OVER"`: Defeat message

## Performance Considerations

**Complete Redraw Approach:** While removing and re-adding all 441 components (21×21) might seem inefficient, this approach is justified because:

1. **Simplicity**: Avoids complex state tracking for changed tiles
2. **Performance**: At 3 FPS (300ms intervals), the redraw cost is negligible.
3. **Swing Optimization**: Swing batches component additions and repaints efficiently

**4. Sprite Caching:** All sprites are loaded once during initialization and stored in a `HashMap`: - No file I/O during gameplay - Instant lookup by character identifier - Shared `ImageIcon` instances across frames (no memory duplication)

## 4.7How Side Portals Work

The portal system in JPacMan implements teleportation across the game board, allowing characters to instantly travel from one side of the maze to the other. The portal teleportation system is handled by the `GameEvents.PortalTeleport()` method, which is called during each game loop iteration after character movement but before GUI update, if an entity reaches a portal coordinate.

### - Pac-Man Portal Detection:

```
int[] pacmanXY = pacman.getCoordinatesXY();
if (Arrays.equals(pacmanXY, portalAxy)) { // portal 1
      pacman.teleportAt(gameBoard, portalBxy);
      portalCrossed = true; // to avoid infinite teleport
     }
     else if (Arrays.equals(pacmanXY, portalBxy)) { //portal 2
          pacman.teleportAt(gameBoard, portalAxy);
          portalCrossed = true;
}
```

The method compares Pac-Man's current coordinates with both portal positions.
When a match is found:

◦ Pac-Man is instantly teleported to the opposite portal
◦ A flag is set to trigger the portal sound effect 3.

### -Ghost Portal Detection:

```
       for (Ghost ghost : ghosts) {
                 if (ghost != null) {
                     int[] ghostXY = ghost.getCoordinatesXY();
                     if (Arrays.equals(ghostXY, portalAxy)) {
                         ghost.teleportAt(gameBoard, portalBxy);
                         portalCrossed = true;
                     } else if (Arrays.equals(ghostXY, portalBxy))
                          {
                         ghost.teleportAt(gameBoard, portalAxy);
                         portalCrossed = true;
                     }
                 }
```

Both `PacMan` and `Ghost` classes implement the `teleportAt()` method defined in the `CharacterActions` interface, but with slightly different implementations; because ghosts needs class must be capable of teleporting various types of ghost, while pacman needs just to teleport a single instance `here there are some considerations about the rules of the teleportation:`

**-Timing:** Portal teleportation occurs after character movement but before GUI update, ensuring: - Characters appear at the new location immediately on the next frame - No visual "jumping" or flickering occurs - Collision detection on the next frame uses the updated positions correctly

**-Bidirectional:** Portals work equally in both directions—entering from either side teleports to the opposite portal. This symmetry makes the mechanic intuitive.

**-Multiple Characters:** The system handles multiple characters using portals simultaneously without conflicts. Each character's teleportation is processed independently, and the sound effect plays once per frame regardless of how many characters teleported.

**Direction Preservation:** Characters retain their movement direction after teleportation, allowing smooth continued movement through the destination portal's corridor.

The portal system demonstrates effective use of coordinate comparison and the `teleportAt()` interface method, adding a classic Pac-Man mechanic while maintaining clean, maintainable code structure.

## 5. Conclusion

JPacMan successfully demonstrates the implementation of a classic arcade game using modern software engineering principles and object-oriented design patterns. The project showcases several key achievements that make it both an effective learning tool and an enjoyable gaming experience.