**Software Design Document (SWDD) Template**

Note: This is only an example and has not been reviewed by LANL to ensure it is truly IEEE 1016 or Ch. 21 compliant.

Background

Software design is a process by which the software requirements are translated into a representation of software components, interfaces, and data necessary for the implementation phase. The SWDD shows how the software system will be structured to satisfy the requirements. It is the primary reference for code development and, therefore, it must contain all the information required by a programmer to write code. The SWDD is performed in two stages. The first is a preliminary design in which the overall system architecture and data architecture is defined. In the second stage—i.e., the detailed design stage—more detailed data structures are defined and algorithms are developed for the defined architecture.

This template is an annotated outline for a software design document adapted from the *IEEE Recommended Practice for Software Design Descriptions*. The *IEEE Recommended Practice for Software Design Descriptions* have been reduced in order to simplify this assignment while still retaining the main components and providing a general idea of a project definition report. For your own information, please refer to IEEE Std 1016[[1]](#footnote-0) for the full *IEEE Recommended Practice for Software Design Descriptions*.

SWDD- [obtain number from Conduct of Engineering Document Numbering [SharePoint site](https://coe.lanl.gov/APs/DocNum/SitePages/Home.aspx)]

Team 1

**Arcade Games**

#### Software Design Document

Name (s): Rey Emmanuel Adag, Mileah Zoei Dy, Kent Vincent Godinez Section: CPE 2201 SOFTWARE DESIGN

Workstation:

Date: (04/26/2025)

**TABLE OF CONTENTS**

[**1.0 INTRODUCTION 4**](#_3feh26ywed49)

[**1.1 Purpose 4**](#_d6bvs3i25mwz)

[**1.2 Scope 4**](#_dhepst1dv8qa)

[**1.3 Overview 4**](#_jqctncsolbad)

[**1.4 Reference Material 4**](#_lhk8e2rwygmu)

[**1.5 Definitions and Acronyms 4**](#_9jvxkn6air6x)

[**2.0 SYSTEM OVERVIEW 4**](#_1453sqghmyfe)

[**2.1 Product Overview 4**](#_quuq5fntnow1)

[**3.0 SYSTEM ARCHITECTURE 6**](#_irzhiawrxaz0)

[**3.1 Architectural Design 6**](#_349p9qbcez91)

[**3.2 Decomposition Description 7**](#_3m33i2b3xouf)

[**3.3 Design Rationale 7**](#_b1ux9agvrni1)

[**4.0 DATA DESIGN 7**](#_89ya3mbs6n51)

[**4.1 Data Description 7**](#_7n7ocpm8d79)

[**4.2 Data Dictionary 8**](#_wi2vt141pzz1)

[**5.0 COMPONENT DESIGN 17**](#_fb8cqxcglg0i)

[**6.0 HUMAN INTERFACE DESIGN 17**](#_5oyt55tbt5xq)

[**6.1 Overview of User Interface 17**](#_ikkxu7t16bra)

[**6.2 Screen Images 17**](#_k3a7p748jbhl)

[**6.3 Screen Objects and Actions 25**](#_pt7nj9r5pddg)

[**7.0 REQUIREMENTS MATRIX 26**](#_kd1tox35e3ba)

[**8.0 APPENDICES 26**](#_hbm5pgrl2czo)

## **INTRODUCTION**

## **Purpose**

Identify the purpose of this SWDD and its intended audience. (e.g. “This software design document describes the architecture and system design of XX. ….”).

## **Scope**

Provide a description and scope of the software and explain the goals, objectives and benefits of your project. This will provide the basis for the brief description of your product.

## **Overview**

Provide an overview of this document and its organization.

## **Reference Material**

*This section is optional.*

List any documents, if any, which were used as sources of information for the test plan.

## **Definitions and Acronyms**

*This section is optional.*

Provide definitions of all terms, acronyms, and abbreviations that might exist to properly interpret the SWDD. These definitions should be items used in the SWDD that are most likely not known to the audience.

## **SYSTEM OVERVIEW**

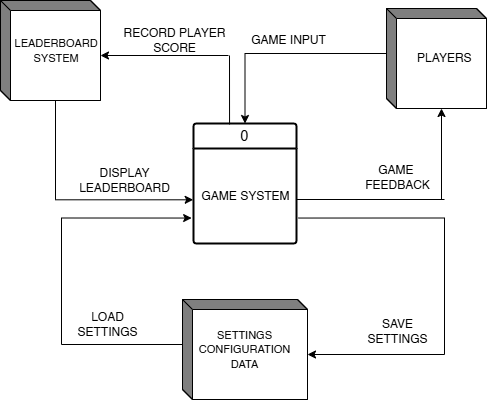
Give a general description of the functionality, context and design of your project. Provide any background information if necessary.

## **2.1 Product Overview**

Arcade Games is a desktop application designed to provide users with a nostalgic gaming experience by offering a collection of classic arcade games, including Pac-Man, Bomberman, and Space Invaders. The platform is intended to be a lightweight, offline gaming solution that allows users to enjoy these games without the need for an internet connection. The application is designed to run on PCs and laptops, providing a simple and intuitive user interface for easy navigation and gameplay.

The product is a self-contained system that does not rely on external services or online connectivity, making it accessible to a wide range of users. The application includes features such as a leaderboard system to track high scores, pause/resume functionality for convenience, and audio settings to customize the gaming experience. The goal of Arcade Games is to deliver a fun and engaging experience for users who enjoy retro-style games.

System Context Diagram:



**2.2 Product Functionality**

The Arcade Games platform is designed to provide users with a seamless and enjoyable gaming experience by offering a variety of classic arcade games and essential features. The system’s major functionalities include:

* Game Selection: Users can select from three classic arcade games: Pac-Man, Bomberman, and Space Invaders.
* Gameplay Mechanics:
  + Pac-Man: Players navigate a maze, collecting pellets while avoiding enemies. Power-ups allow players to temporarily defeat enemies.
  + Bomberman: Players place bombs to destroy obstacles and enemies while avoiding their own explosions.
  + Space Invaders: Players shoot at descending rows of enemies before they reach the bottom of the screen.
* Leaderboard System: Tracks and displays high scores for each game, allowing players to compete for the top spot.
* Pause/Resume Functionality: Players can pause the game at any time and resume from where they left off.
* Audio Settings: Users can adjust the volume of sound effects and background music to suit their preferences.
* Main Menu: Provides easy navigation to game selection, leaderboards, and audio settings.

**2.3 Design and Implementation Constraints**

The development of the Arcade Games platform is subject to several design and implementation constraints that guide the system’s architecture and functionality. These constraints include:

1. Platform: The application must run on Windows-based PCs and laptops.
2. Offline Functionality: The system must operate without requiring an internet connection.
3. Hardware Requirements: The application should be lightweight and run efficiently on standard hardware without requiring high-end specifications.
4. Development Tools: The system will be developed using the COMET method for software design and UML modeling for system documentation.
5. Programming Language: The application will be developed using a language suitable for desktop applications, such as C# or Java.
6. User Interface: The interface must be simple and intuitive, with clear navigation options for game selection, leaderboards, and settings.

**2.4 Assumptions and Dependencies**

The successful implementation of the Arcade Games platform relies on several assumptions and dependencies that influence the system’s design and functionality. These include:

Assumptions:

* Users have basic familiarity with classic arcade games and their mechanics.
* The application will be used primarily for entertainment purposes.
* The system will not require frequent updates or online connectivity.
* The hardware on which the application runs will meet the minimum requirements for smooth gameplay.

Dependencies:

* The system relies on standard input devices such as a keyboard and mouse.
* The application depends on the availability of audio output devices for sound effects and music.
* The leaderboard system stores data locally, so it does not depend on external databases or cloud services.

## **SYSTEM ARCHITECTURE**

## **Architectural Design**

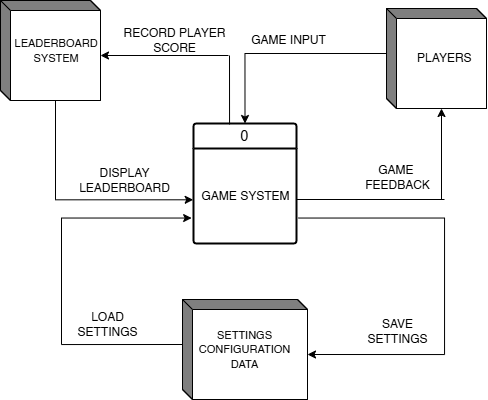
Develop a modular program structure and explain the relationships between the modules to achieve the complete functionality of the system. This is a high level overview of how responsibilities of the system were partitioned and then assigned to subsystems. Identify each high level subsystem and the roles or responsibilities assigned to it. Describe how these subsystems collaborate with each other in order to achieve the desired functionality. Don’t go into too much detail about the individual subsystems. The main purpose is to gain a general understanding of how and why the system was decomposed, and how the individual parts work together. Provide a diagram showing the major subsystems and data repositories and their interconnections. Describe the diagram if required.

The system adopts a modular MVC architecture with three primary layers:

1. Presentation Layer: Handles UI Rendering (Java Swing)
2. Game Logic Layer: Manages core gameplay mechanics
3. Data Layer: Local storage for leaderboards and settings

## **Decomposition Description**

Provide a decomposition of the subsystems in the architectural design. Supplement with text as needed. You may choose to give a functional description or an object­oriented (OO) description. For a functional description, put top­level data flow diagram (DFD) and structural decomposition diagrams. For an OO description, put subsystem model, object diagrams, generalization hierarchy diagram(s) (if any), aggregation hierarchy diagram(s) (if any), interface specifications, and sequence diagrams here.

****

## **Design Rationale**

Discuss the rationale for selecting the architecture described in 3.1 including critical issues and trade/offs that were considered. You may discuss other architectures that were considered, provided that you explain why you didn’t choose them.

## **DATA DESIGN**

## **Data Description**

Explain how the information domain of your system is transformed into data structures. Describe how the major data or system entities are stored, processed and organized. List any databases or data storage items.

| **Data Item** | **Data Type** | **Description** |
| --- | --- | --- |
| Player | Single instance of Player | The player character information such as coordinates, number of lives, power up status, and animation sprites. |
| Enemy | ArrayList<Enemy> | ArrayList of enemies active on the play area. |
| Tiles | Tile[] | Array of the different map tile types with each of their collision status and image. The base maps are stored as a local text file. |
| Map Tile Number | int[][] | A 2D array of tile indices depending on their placement on the map. |
| Player Score | ArrayList<PlayerScore> | ArrayList of PlayerScore objects holding each of their names and scores. File management would be used to store player scores locally |

## **Data Dictionary**

Alphabetically list the system entities or major data along with their types and descriptions. If you provided a functional description in Section 3.2, list all the functions and function parameters. If you provided an OO description, list the objects and its attributes, methods and method parameters.

1. **GamePanelBM class**

The GamePanel class is responsible for initializing game screen properties and handling graphics for the BomberMan Game.

* 1. **setupGame**

**Purpose:** To setup assets and initial game state

**Prototype:** public void setupGame()

**Inputs:** None

**Outputs:** None

**Called by:** Main.main()

**Calls:**

* setEnemy()
  1. **resetGame**

**Purpose:** To reset the game state to its initial state

**Prototype:** public void resetGame()

**Inputs:** None

**Outputs:** None

**Called by:** GamePanelBM.update()

**Calls:**

* player.setDefaultVal()
* bombs.clear()
* explosions.clear()
* enemies.clear()
* tileMgr.loadMap()
* setupGame()
  1. **startGameThread**

**Purpose:** To execute the thread to run the flow of the game loop

**Prototype:** public void startGameThread()

**Inputs:** None

**Outputs:** None

**Called by:** GamePanelBM.update()

**Calls:**

* gameThread.start()
  1. **run**

**Purpose:** To update the game logic and repaint the screen for every frame

**Prototype:** public void resetGame()

**Inputs:** None

**Outputs:** None

**Called by:** None

**Calls:**

* GamePanelBM.update()
* repaint()
  1. **update**

**Purpose:** To reset the game state to its initial state

**Prototype:** public void update()

**Inputs:** None

**Outputs:** None

**Called by:** GamePanelBM.run()

**Calls:**

* explode()
* resetGame()
* e.update()
* b.update()
* player.update()
* player.getInvincible()
* player.setInvincible()
* player.setHit()
* player.getX()
* player.getY()
* e.getX()
* e.getY()
* player.removeBombsPlaced()
* e.getExpStatus()
* enemy.update()
* b.getBombStatus()
  1. **addExplosions**

**Purpose:** Creates and adds a new explosion effect at the given (x, y) position in the game world

**Prototype:** public void addExplosions(int x, int y)

**Inputs:** x - Integer. y - Integer. X and Y positions where the explosion will take place

**Outputs:** None

**Called by:** explode()

**Calls:**

* explosions.add()
  1. **explode**

**Purpose:** Handles the explosion of bomb detonation by spreading explosions in all x and y directions until it is blocked or reaches its maximum radius

**Prototype:** public void resetGame()

**Inputs:** None

**Outputs:** None

**Called by:** GamePanelBM.update()

**Calls:**

* b.getX()
* b.getY()
* addExplosions()
* checkTileExp()
* player.getBombRadius()
  1. **checkTileExp**

**Purpose:** Checks whether a tile is empty or destructible and returns a boolean depending on the status of that tile.

**Prototype:** public boolean checkTileExp(int x, int y, boolean empty)

**Inputs:** x - Integer. Tile x position. y - Integer. Tile y position. empty - Boolean. Tile path status.

**Outputs:** empty - Boolean. Updated tile path status

**Called by:** explode()

**Calls:**

* player.setDefaultVal()
  1. **paintComponent**

**Purpose:** To reset the game state to its initial state

**Prototype:** public void resetGame()

**Inputs:** g - Graphics. Context used to draw elements.

**Outputs:** None

**Called by:** GamePanelBM.update()

**Calls:**

* super.paintComponent()
* tileMgr.draw()
* b.draw()
* player.draw()
* enemy.draw()
* e.draw()
* ui.draw()

1. **UIBM**

Responsible for drawing User Interface (UI) elements onto the game screen depending on the game state.

* 1. **draw**

**Purpose:** Draws the appropriate UI elements depending on the current game state.

**Prototype:** public void draw(Graphics2D g2)

**Inputs:** g2 - Graphics2D. Context used to draw elements.

**Outputs:** None

**Called by:** GamePanelBM.paintComponent()

**Calls:**

* drawPauseScreen()
* drawGameoverScreen()
* drawWinScreen()
  1. **drawPauseScreen**

**Purpose:** Draws the appropriate UI elements for the pause screen.

**Prototype:** public void draw(Graphics2D g2)

**Inputs:** None

**Outputs:** None

**Called by:** UIBM.draw()

**Calls:**

* getTextX()
* g2.drawString()
  1. **drawGameoverScreen**

**Purpose:** Draws the appropriate UI elements for the game over screen.

**Prototype:** public void draw(Graphics2D g2)

**Inputs:** None

**Outputs:** None

**Called by:** UIBM.draw()

**Calls:**

* getTextX()
* g2.drawString()
  1. **drawWinScreen**

**Purpose:** Draws the appropriate UI elements for the level completion screen.

**Prototype:** public void draw(Graphics2D g2)

**Inputs:** None

**Outputs:** None

**Called by:** UIBM.draw()

**Calls:**

* getTextX()
* g2.drawString()
  1. **getTextX**

**Purpose:** Gets the X value for centering text on the screen.

**Prototype:** public void draw(Graphics2D g2)

**Inputs:** text - String. Text to be displayed on screen.

**Outputs:** x - Integer. The calculated x-coordinate for centered text

**Called by:** drawPauseScreen(), drawGameoverScreen(), drawWinScreen()

**Calls:** None

1. **CollisionChecker**

Detects collisions between entities and tiles to enforce movement restrictions.

* 1. **checkTile**

**Purpose:** Checks if the entity will collide with a solid tile based on its direction and position, and updates its collision status

**Prototype:** public void checkTile(Entity entity)

**Inputs:** entity - Entity. The entity which has its collision verified.

**Outputs:** None

**Called by:** Entity.update(), Player.update()

**Calls:** None

* 1. **checkEnemy**

**Purpose:** Checks if the entity will collide with a solid tile based on its direction and position, and updates its collision status

**Prototype:** public boolean checkEnemy(Entity entity, ArrayList<Enemy> targets)

**Inputs:** entity - Entity. The entity which has its collision verified. targets - ArrayList<Enemy>. The list of enemies which have their collision verified.

**Outputs:** hit - Boolean. Result of collision checking, where it is true if there is collision and false if otherwise

**Called by:** Player.update()

**Calls:** None

* 1. **checkPlayer**

**Purpose:** Checks if the entity collides with the player based on the entity’s direction and position

**Prototype:** public boolean checkPlayer(Entity entity)

**Inputs:** entity - Entity. The entity which has its collision verified.

**Outputs:** hit - Boolean. Result of collision checking, where it is true if there is collision and false if otherwise

**Called by:** Entity.update()

**Calls:** None

1. **AssetSetter**

The AssetSetter class is responsible for placing game objects and entities onto the game map during the setup phase of the game.

* 1. **setEnemy**

**Purpose:** Sets up enemy entities on the map

**Prototype:** public void setEnemy()

**Inputs:** None

**Outputs:** None

**Called by:** GamePanel.setupGame()

**Calls:**

* gp.enemies.add()

1. **ImageScaler**

The ImageScaler class is responsible for pre-scaling images to a set size before they are drawn in order to reduce runtime.

* 1. **scale()**

**Purpose:** Scales images and sprites to a fixed width and height

**Prototype:** public BufferedImage scale(int height, int width, BufferedImage image)

**Inputs:** height - Integer. The target height for the scaled image. width - Integer. The target width for the scaled image. image - BufferedImage. The original image to be scaled.

**Outputs:** scaled - BufferedImage. The scaled version of the original image.

**Called by:** Entity.setupImage(), Explosion.setupImage(), Bomb.setupImage(), TileManager.setupImage()

**Calls:**

* scaled.createGraphics()
* g2.drawImage

1. **KeyHandler**

The KeyHandler class is responsible for handling key inputs.

* 1. **keyPressed**

**Purpose:** Handles player key inputs

**Prototype:** public void keyPressed(KeyEvent e)

**Inputs:** e - KeyEvent. The key pressed by the player

**Outputs:** None

**Called by:** Player.update()

**Calls:**

* e.getKeyCode()
  1. **keyReleased**

**Purpose:** Handles player key inputs

**Prototype:** public void keyReleased(KeyEvent e)

**Inputs:** e - KeyEvent. The key pressed by the player

**Outputs:** None

**Called by:** Player.update()

**Calls:**

* e.getKeyCode()

1. **GamePanelPM class**

The GamePanelPM class is responsible for initializing game screen properties and handling graphics for the PacMan Game.

* 1. **setupGame**

**Purpose: To setup assets and initial game state**

**Prototype: public void setupGame()**

**Inputs: None**

**Outputs: None**

**Called by: Main.main()**

**Calls:**

* **loadSprites()**
* **setGhosts()**
* **resetGame()**
  1. **resetGame**

**Purpose: To reset the game state to its initial state**

**Prototype: public void resetGame()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **player.reset()**
* **ghosts.clear()**
* **pelletMgr.reset()**
* **setupGame()**
  1. **startGameThread**

**Purpose: To execute the thread to run the flow of the game loop**

**Prototype: public void startGameThread()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **gameThread.start()**
  1. **run**

**Purpose: To update the game logic and repaint the screen for every frame**

**Prototype: public void run()**

**Inputs: None**

**Outputs: None**

**Called by: None**

**Calls:**

* **GamePanel.update()**
* **repaint()**
  1. **update**

**Purpose: To update all game entities and check game state**

**Prototype: public void update()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.run()**

**Calls:**

* **player.move()**
* **checkCollisions()**
* **ghosts.update()**
* **checkPellets()**
* **checkPowerUpTimer()**
* **checkLevelComplete()**
  1. **checkCollisions**

**Purpose: Checks collisions between player and walls/ghosts**

**Prototype: private void checkCollisions()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **player.getBounds()**
* **ghosts.getBounds()**
* **wallMgr.checkCollision()**
* **player.loseLife()**
  1. **checkPellets**

**Purpose: Handles pellet collection and score updates**

**Prototype: private void checkPellets()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **pelletMgr.collect()**
* **player.addScore()**
* **activatePowerUp()**
  1. **activatePowerUp**

**Purpose: Enables ghost vulnerability when power-up is collected**

**Prototype: private void activatePowerUp()**

**Inputs: None**

**Outputs: None**

**Called by: checkPellets()**

**Calls:**

* **ghosts.setVulnerable()**
* **startPowerUpTimer()**
  1. **checkPowerUpTimer**

**Purpose: Tracks power-up duration and disables when expired**

**Prototype: private void checkPowerUpTimer()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **ghosts.setNormal()**
  1. **checkLevelComplete**

**Purpose: Checks if all pellets are collected to advance level**

**Prototype: private void checkLevelComplete()**

**Inputs: None**

**Outputs: None**

**Called by: GamePanel.update()**

**Calls:**

* **pelletMgr.isEmpty()**
* **levelMgr.nextLevel()**
* **resetGame()**

## **COMPONENT DESIGN**

In this section, we take a closer look at what each component does in a more systematic way. If you gave a functional description in section 3.2, provide a summary of your algorithm for each function listed in 3.2 in procedural description language (PDL) or pseudocode. If you give an OO description, summarize each object member function for all the objects listed in 3.2 in PDL or pseudocode. Describe any local data when necessary.

## **HUMAN INTERFACE DESIGN**

## **Overview of User Interface**

Describe the functionality of the system from the user’s perspective. Explain how the user will be able to use your system to complete all the expected features and the feedback information that will be displayed for the user.

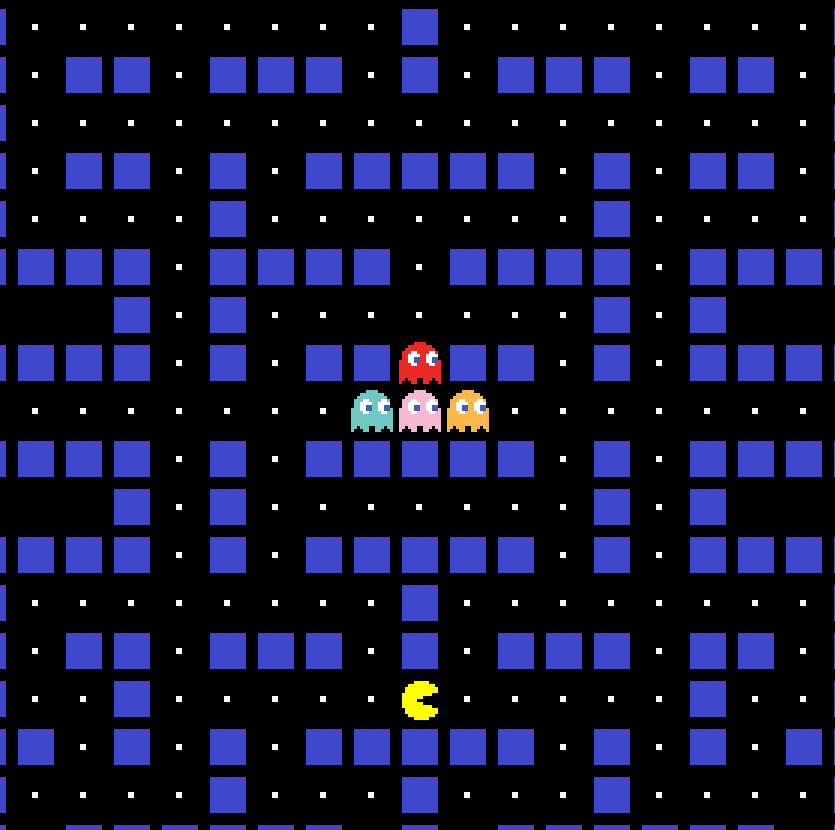
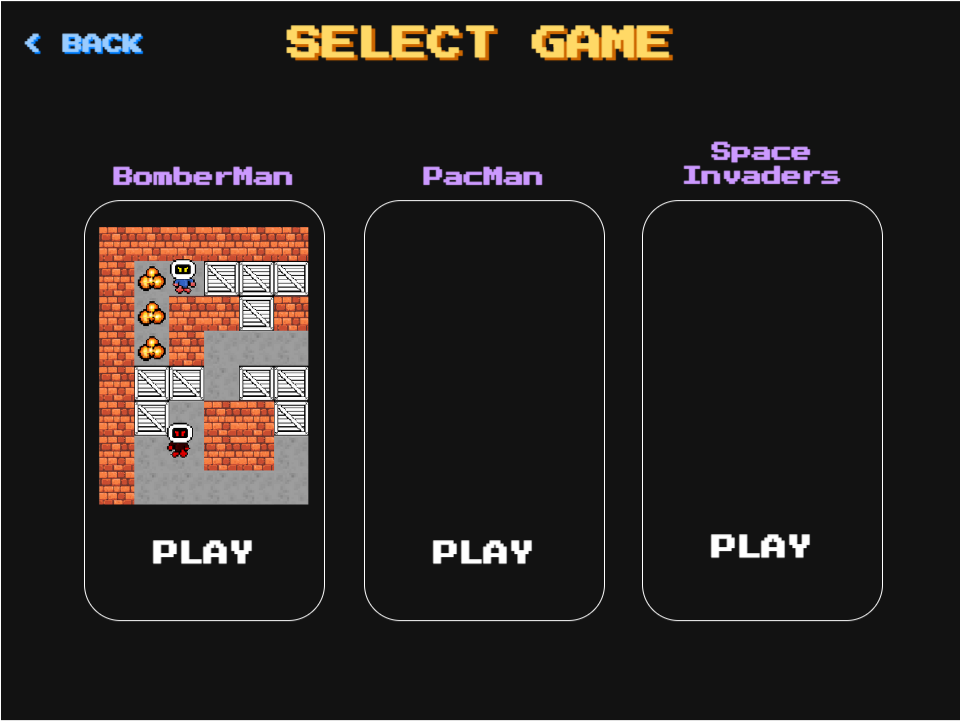
The system’s user interface starts on the Main Menu Screen upon application startup. From here, the user can access the Settings screen, Leaderboard screen, and the Select Game screen. From the Game Select screen, the user can access the BomberMan Game Screen, PacMan Game Screen, and Space Invaders Game Screen. Within each of the games, the user can access each of their Pause and Game Over Screens. The user can enter the Pause screen by pressing the pause button (‘ESC’ by default). The Game Over Screen is displayed once the player loses the game by losing all their lives.

## **Screen Images**

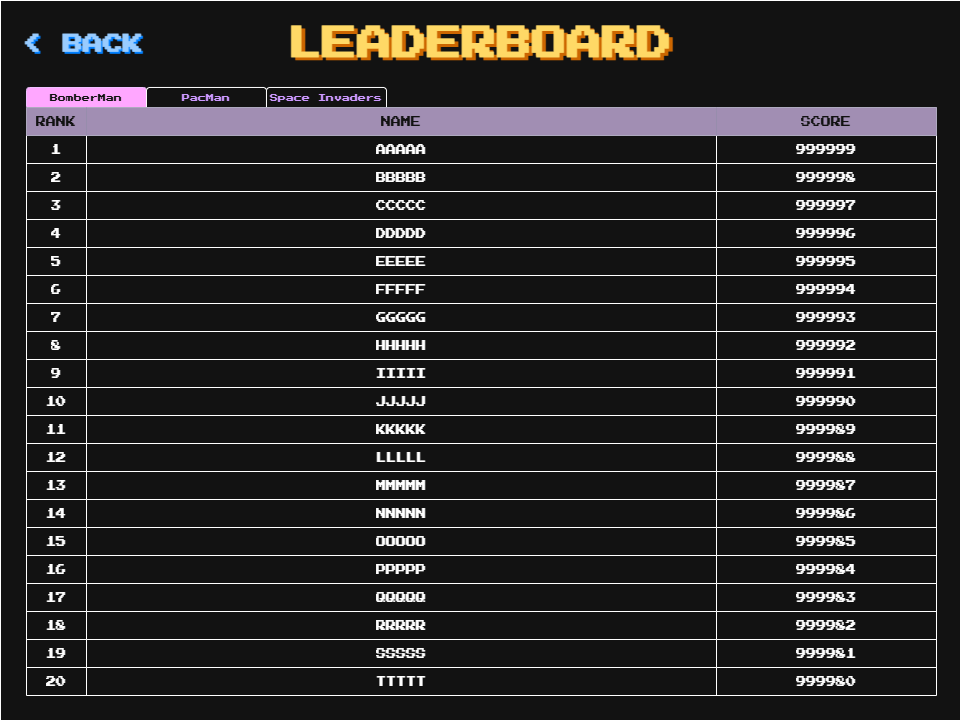
Display screenshots showing the interface from the user’s perspective. These can be hand­ drawn or you can use an automated drawing tool. Just make them as accurate as possible. (Graph paper works well.)



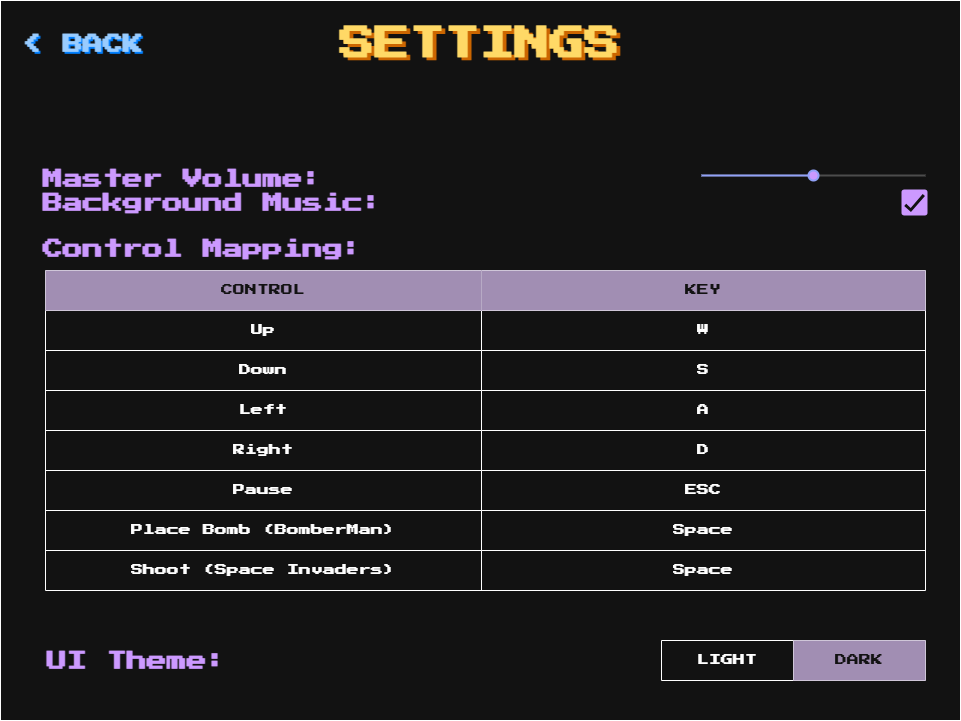
**Figure 6.2.1.** Main Menu Screen



**Figure 6.2.2.** Game Select Screen



**Figure 6.2.3.** Leaderboard Screen



**Figure 6.2.4.** Settings Screen



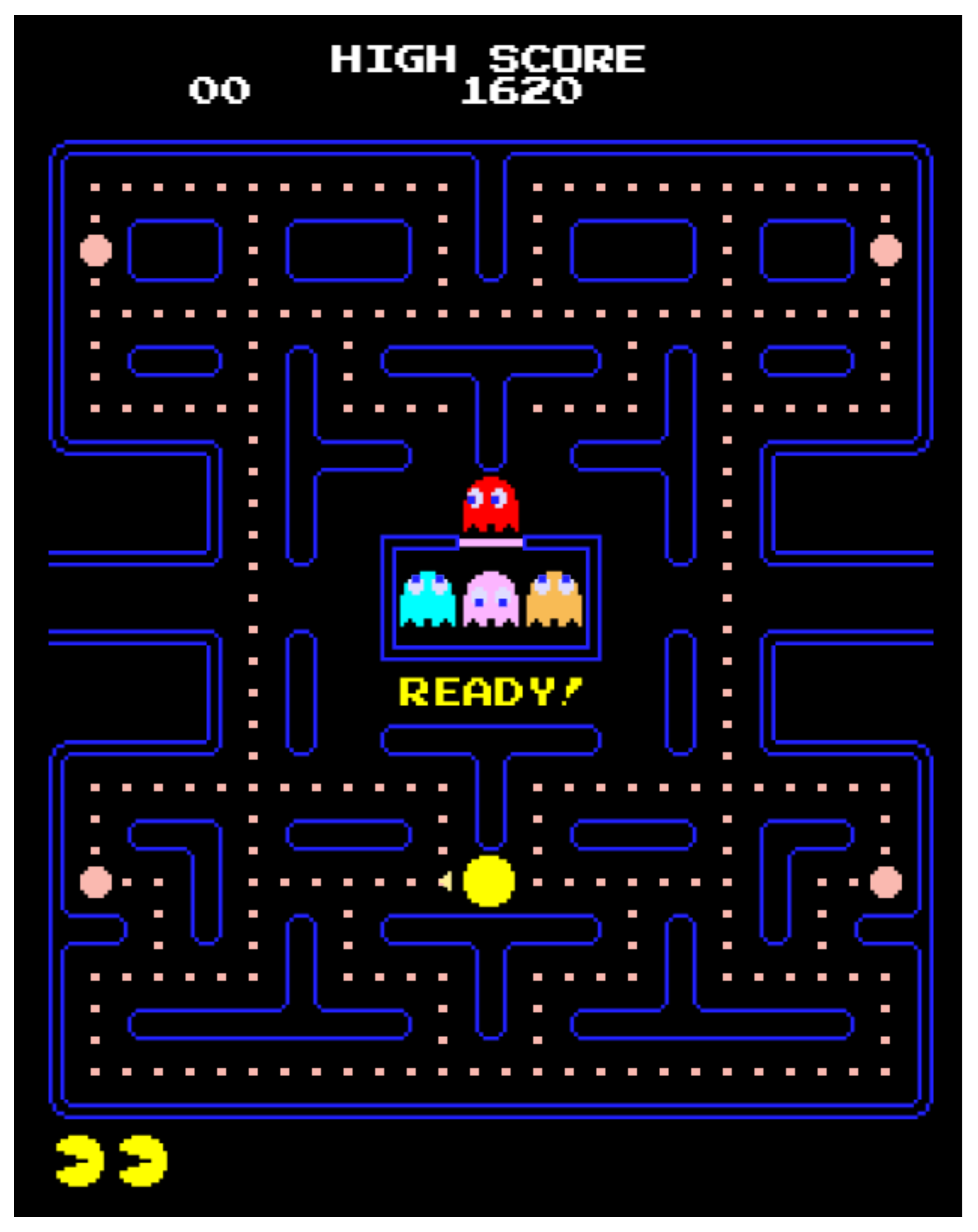
**Figure 6.2.5.** BomberMan In-Game Screen



**Figure 6.2.6.** BomberMan Pause Screen



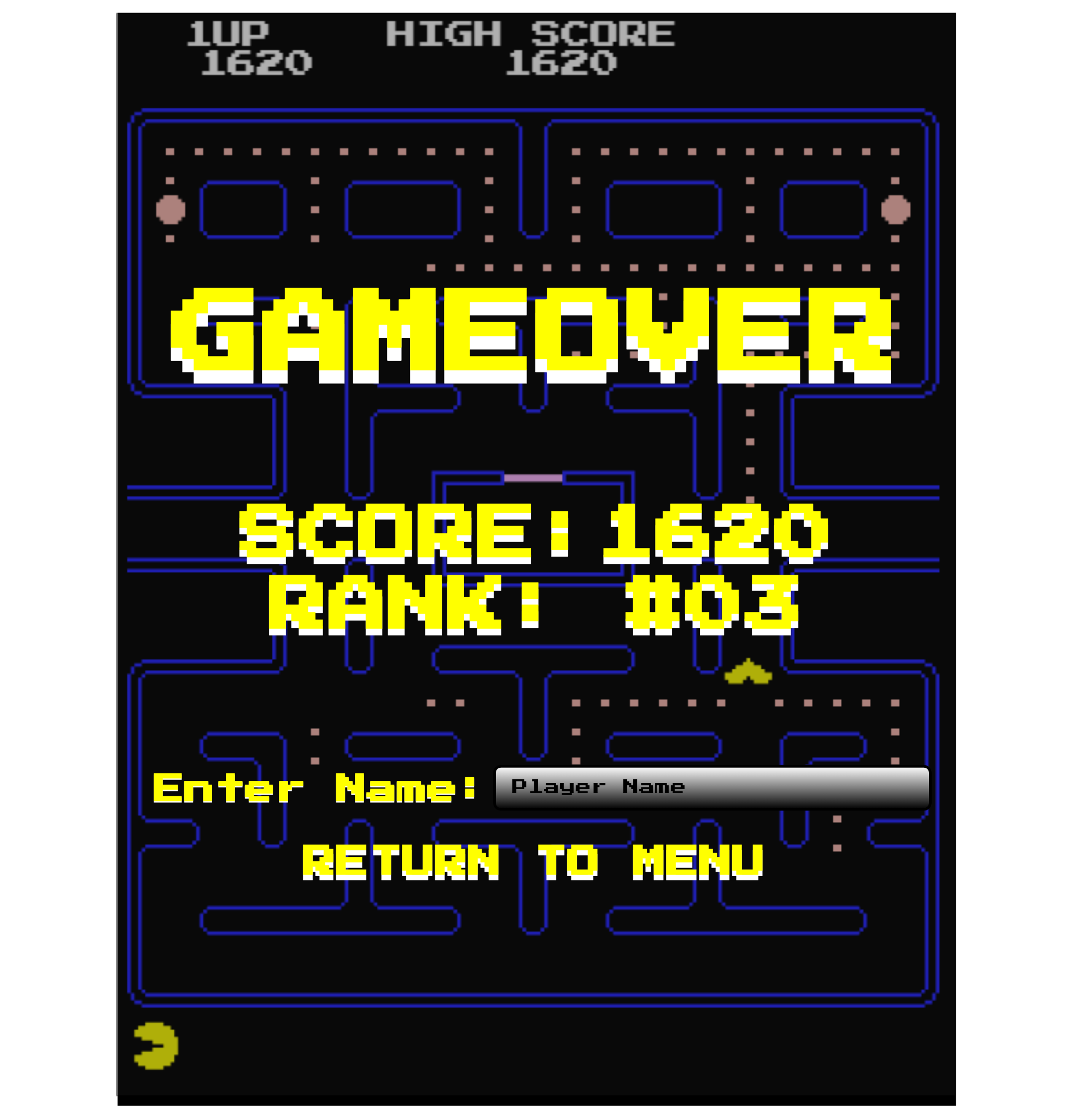
**Figure 6.2.7.** BomberMan Game Over Screen



**Figure 6.2.8.** Pacman In-Game Screen



**Figure 6.2.9.** Pacman Pause Screen



**Figure 6.2.10.** Pacman Game Over Screen



**Figure 6.2.11.** Space Invaders In-Game Screen



**Figure 6.2.12.** Space Invaders Pause Screen



**6.2.13.** Space Invaders Game Over Screen

## **Screen Objects and Actions**

A discussion of screen objects and actions associated with those objects.

In the Main Menu, the user can click on text buttons for Game Select, Leaderboard, Settings, and Exit to proceed to each of their menus or exit the game.

The Game Select menu shows the selection of games and a Play text button that the user can click to play that game. There is also a Back button for the user to return to the menu.

The Leaderboard page has separate button tabs for each game found at the top of the leaderboard window which the user can click to access each of their leaderboards. There is also a Back button for the user to return to the menu.

The Settings menu has several possible actions. A slider can be adjusted to change the game’s master volume. A checkbox can be selected or deselected to turn on and off any background music. For control mapping, the user can click the key found on the right side of the table which they wish to change. After which, they click any button on their keyboard that they want to remap that control to. The UI Theme can also be chosen by clicking the dark or light mode buttons. There is also a Back button for the user to return to the menu.

The In-Game screens for each game have no on-screen buttons to click, however they rely on keyboard input.

The Pause menu has three text buttons which the user can click to Resume, Restart, or Exit the game.

The Game Over screen has a textbox which the user can type in to enter their name which will be added to the leaderboard. A Return To Menu text button can also be found at the bottom which allows the user to return to the main menu once clicked.

## **REQUIREMENTS MATRIX**

Provide a cross reference that traces components and data structures to the requirements in your softwarerequirements specification (SWRS) document.

Use a tabular format to show which system components satisfy each of the functional requirements from the SWRS. Refer to the functional requirements by the numbers/codes that you gave them in the SWRS.

## **APPENDICES**

*This section is optional.*

Appendices may be included, either directly or by reference, to provide supporting details that could aid in the understanding of the Software Design Document.

1. Available to LANL users at <http://ieeexplore.ieee.org/browse/standards/collection/ieee> [↑](#footnote-ref-0)