# AstroStreet AR

## Software Design Document (SDD)

Assignment 14

CS 3338 – Software Engineering Tools

California State University, Los Angeles

**Group 2**

Royce Jamerson
Khalid Jamil
Michael Lieng
Ricardo Ibanez

Date: December 11, 2025

# Contents

# 1 Version Description

This section tracks the revision history of the *AstroStreet AR* Software Design Document (SDD) for Group 2 in CS 3338 – Software Engineering Tools.

| Version | Description | Date |
|---|---|---|
| 1.0 | Initial version of the Software Design Document for *AstroStreet AR*, prepared for Assignment 14. | December 2, 2025 |
| 2.0 | Expanded SDD with detailed component responsibilities, AR session lifecycle and error handling, key runtime scenarios, and refined database design for *AstroStreet AR*. | December 7, 2025 |
| 3.0 | Final version of the SDD including game state management details, UI navigation flow, performance and resource considerations, and notes on future extensibility (e.g., online leaderboards and additional game modes). | December 11, 2025 |

Future revisions of this document can be recorded by adding new rows to the table with updated version numbers, dates, and brief descriptions of the changes.

# 2 Introduction

## 2.1 Purpose

The purpose of this Software Design Document (SDD) is to describe the overall design and internal structure of *AstroStreet AR*, a mobile augmented reality (AR) shooter game developed by Group 2 for CS 3338 – Software Engineering Tools at California State University, Los Angeles. This document translates the high-level requirements of the project into a concrete design that can guide implementation, testing, and future maintenance.

## 2.2 System Overview

*AstroStreet AR* is a smartphone game in which players go outside, use their phone's camera, and see virtual asteroids and alien ships overlaid on the real world. The player rotates their body and phone to aim and taps on the screen to fire projectiles and destroy incoming targets. The game tracks score, player health or shields, and increasing difficulty as more enemies appear over time.

At a high level, the system consists of:

- A mobile client that renders the AR scene, handles user input, manages game logic, and displays the user interface.
- AR services provided by the underlying platform (e.g., ARCore/ARKit via an engine such as Unity) to track device position and orientation and to overlay virtual objects on the camera feed.
- Local data storage to persist high scores and basic player settings on the device.

## 2.3 Intended Audience

This document is intended for:

- Members of Group 2, who will use the design as a blueprint for implementing *AstroStreet AR*.
- The course instructor and teaching staff, who will review the design as part of Assignment 14.
- Future developers or maintainers who may extend the game's features, port it to new platforms, or integrate additional services (such as online leaderboards).

## 2.4 Document Overview

The remainder of this SDD is organized as follows:

- **System Architecture** describes the major components of *AstroStreet AR*, their responsibilities, and how they interact.
- **User Interface** presents the planned screens and HUD elements, including the main menu, AR gameplay view, and score displays.
- **Database Design and Explanation** outlines the data structures used to store high scores and game settings, and explains how the application accesses this data.
- **Glossary** defines key technical terms and acronyms used throughout the document.
- **References** lists external resources, documentation, and tools consulted while designing *AstroStreet AR*.

# 3 System Architecture

## 3.1 Architecture Overview

The architecture of *AstroStreet AR* is organized around a single mobile client application that runs on a smartphone and communicates with platform-specific AR services. Within the client, the game is decomposed into modular components that manage AR tracking, game state, enemy behavior, player input, user interface, and local data storage.

At a high level, the system consists of the following layers:

- **Presentation Layer**: Responsible for rendering the camera feed, drawing virtual objects (asteroids, alien ships, projectiles), and displaying the heads-up display (HUD) and menus.
- **Game Logic Layer**: Manages the core game loop, including spawning enemies, updating the player state, handling collisions, tracking score, and determining win/lose conditions.

- **AR and Device Services Layer**: Interfaces with the underlying AR framework (e.g., ARCore/ARKit via Unity) to obtain camera images, device pose (position and orientation), and other sensor data.
- **Data Layer**: Handles reading and writing persistent data, such as high scores and basic settings, on the device.

Figure 1 conceptually illustrates the relationships between the major components.

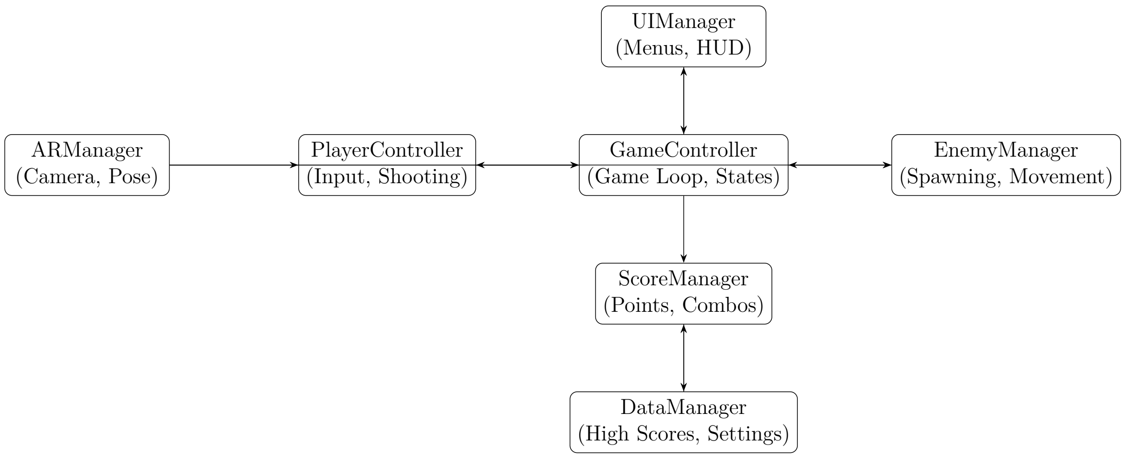

Figure 1: High-level component architecture of *AstroStreet AR*.

## 3.2 Major Components

The main components of the client application are described below.

### 3.2.1 ARManager

The **ARManager** is responsible for:

- Initializing the AR session when the game starts.
- Accessing the device camera feed and sensor data from the AR framework.
- Providing the current device pose (position and orientation) so that virtual objects can be placed consistently in the player's view.

Other components (such as the **GameController** and **EnemyManager**) rely on the **AR-Manager** to ensure that objects are rendered relative to the player's real-world orientation.

### 3.2.2 GameController

The **GameController** coordinates the overall game loop:

- Starting and stopping gameplay sessions.

- Updating the game state each frame (time remaining, player health or shields, active enemies, and projectiles).
- Notifying other components when the game transitions between states (e.g., from “Main Menu” to “In Game” to “Game Over”).

It serves as the central point of control, invoking the **EnemyManager**, **PlayerController**, and **UIManager** as needed during each update cycle.

#### 3.2.3 EnemyManager

The **EnemyManager** handles all enemy-related behavior:

- Spawning asteroids and alien ships according to the current difficulty level.
- Updating enemy movement and checking when enemies become too close to the player.
- Informing the **GameController** when enemies are destroyed or when they collide with the player.

The **EnemyManager** uses AR pose information from the **ARManager** to decide where to place enemies in the player’s field of view.

#### 3.2.4 PlayerController

The **PlayerController** interprets user input and updates the player’s actions:

- Reading touch input (e.g., tapping the screen to shoot).
- Aligning the virtual crosshair or reticle with the player’s current viewing direction.
- Requesting the creation of projectiles when the player fires and reporting hits back to the **GameController**.

It acts as the bridge between physical device movements/touches and in-game actions.

#### 3.2.5 UIManager

The **UIManager** manages the user interface and HUD:

- Displaying the main menu, pause menu, and game over screen.
- Rendering HUD elements during gameplay, such as the score, health or shield bar, time remaining, and any active power-ups.
- Receiving navigation commands from the user (e.g., starting a new game, returning to the main menu).

The **UIManager** communicates with the **GameController** and **ScoreManager** to update visual information.

#### 3.2.6 ScoreManager

The **ScoreManager** tracks and updates the player's score:

- Awarding points when enemies are destroyed.
- Applying score multipliers or bonuses for streaks.
- Exposing the current score to the **UIManager** for display during gameplay.

At the end of a game session, the **ScoreManager** collaborates with the **DataManager** to save high scores.

#### 3.2.7 DataManager

The **DataManager** is responsible for:

- Persisting high scores and basic settings (such as sound preferences and control sensitivity) to local storage on the device.
- Loading saved data when the application starts.
- Providing a simple interface for other components to read and write data without dealing directly with the storage mechanism.

## 3.3 Component Interactions

During a typical gameplay session, the components interact as follows:

1. The user selects "Start Game" from the main menu. The **UIManager** notifies the **GameController** to begin a new session.
2. The **GameController** initializes the game state and requests the **ARManager** to start the AR session.
3. The **EnemyManager** begins spawning enemies in positions derived from the AR pose provided by the **ARManager**.
4. Each frame, the **PlayerController** reads user input (screen taps) and creates projectiles along the player's current viewing direction.
5. The **GameController** updates all active entities, checks for collisions between projectiles and enemies, and adjusts the player's health or shields if enemies get too close.
6. The **ScoreManager** updates the score when enemies are destroyed, and the **UIManager** refreshes the HUD to show the latest score and health values.
7. When the game ends (e.g., the player's health reaches zero), the **GameController** notifies the **ScoreManager** and **DataManager** to save the final score, then instructs the **UIManager** to display the game over screen.

This modular architecture is intended to keep responsibilities clearly separated, making the system easier to implement, test, and extend with future features such as additional enemy types, power-ups, or online leaderboards.

## 3.4 Component Responsibilities

The AstroStreet AR system is organized into a small set of core components. Each component has a clear set of responsibilities and limited knowledge of other components in order to keep the design modular and maintainable.

Table 1 summarizes the main runtime components and how they collaborate.

By clearly separating responsibilities into these components, the team can modify game rules, UI behavior, or persistence details with minimal impact on the AR integration layer, and vice versa.

## 3.5 AR Session Lifecycle and Error Handling

AstroStreet AR relies on a mobile AR session to align virtual enemies and projectiles with the player's physical environment. The `ARController` encapsulates the AR session lifecycle and exposes a simplified interface to the rest of the game.

At a high level, the AR session progresses through the following states:

- **Not Initialized**: The app has launched, but no AR session has been created yet. The `GameManager` displays the Main Menu during this state.
- **Initializing**: After the player taps "Start Game", the `ARController` requests camera permissions (if needed), configures the AR session, and begins tracking.
- **Tracking**: The AR SDK reports a stable pose and tracking state. Enemies can be spawned at positions derived from detected planes or from the device's forward direction, and gameplay proceeds normally.
- **Limited / Lost Tracking**: The AR SDK reports that tracking is degraded (e.g., due to low light, fast motion, or lack of visual features). During this state, the game temporarily pauses spawning new enemies and may reduce or pause collision checks to avoid jarring behavior.

The `ARController` periodically reports the current tracking status to the `GameManager`. When tracking is limited or lost, the `UIManager` displays a simple overlay message such as "Move your phone slowly" or "Point your camera at a textured surface". Once tracking returns to the **Tracking** state, normal gameplay resumes.

If the player denies camera or AR permissions, or if the device cannot support AR, the `ARController` reports a fatal initialization error to the `GameManager`. In this case, the `UIManager` shows an error dialog and routes the user back to the Main Menu, where they can either retry or exit the game.

## 3.6 Key Runtime Scenarios

This subsection describes two core runtime scenarios in terms of the collaboration between components. These scenarios serve as an informal sequence-diagram-style description.

Table 1: Core Component Responsibilities

| **Component** | **Responsibilities** | **Collaborates With** |
|---|---|---|
| `GameManager` | Owns the overall game state (e.g., Main Menu, Playing, Paused, Game Over). Initializes subsystems on game start, triggers wave progression, and coordinates transitions between screens. | `EnemyManager`, `PlayerController`, `UIManager`, `DataManager`, `ARController` |
| `EnemyManager` | Spawns, updates, and removes enemies (asteroids and alien ships) within the AR scene. Controls enemy wave logic, difficulty scaling, and despawn rules when enemies move out of range or time out. | `GameManager`, `ARController` |
| `PlayerController` | Reads device input (screen taps) and device pose/orientation from the AR subsystem. Responsible for firing projectiles in the current view direction and notifying the `GameManager` when a hit occurs. | `GameManager`, `ARController`, `UIManager` |
| `UIManager` | Manages all UI screens (Main Menu, AR HUD, Pause Menu, Game Over, Settings, High Scores). Updates on-screen information such as score, health, and wave number based on notifications from the `GameManager`. | `GameManager`, `DataManager` |
| `DataManager` | Provides a simple persistence API for saving and loading local data such as high scores, basic player statistics, and user settings. Hides the underlying storage mechanism (e.g., SQLite or key–value store) from the rest of the system. | `GameManager`, `UIManager` |
| `ARController` | Wraps the underlying AR SDK (e.g., ARCore/ARKit via Unity AR Foundation). Manages the AR session lifecycle, pose tracking, and raycasts. Provides utility methods to place enemies and projectiles in world space relative to the player's real-world position and orientation. | `GameManager`, `EnemyManager`, `PlayerController` |

### 3.6.1 Starting a New AR Game Session

1. The player launches the app and sees the Main Menu rendered by the `UIManager`.
2. The player taps the "Start Game" button.
3. The `UIManager` notifies the `GameManager` that a new game has been requested.
4. The `GameManager` instructs the `ARController` to start the AR session. The `ARController` requests permissions if necessary and transitions through the **Initializing** state into **Tracking**.
5. Once tracking is stable, the `GameManager` updates its internal state to **Playing** and tells the `UIManager` to display the AR HUD.
6. The `EnemyManager` begins spawning enemies at positions computed via the `ARController` (e.g., in front of the player's current pose or on detected planes).

### 3.6.2 Firing at an Enemy and Updating the Score

1. While in the **Playing** state, the player taps on the screen.
2. The `PlayerController` receives the tap input and queries the `ARController` for the current device pose and forward direction.
3. The `PlayerController` spawns a projectile traveling along the forward direction and registers it with the game world.
4. During the next update cycle, the game checks for collisions between projectiles and enemies managed by the `EnemyManager`.
5. If a collision is detected, the `EnemyManager` removes the enemy and notifies the `GameManager` that an enemy has been destroyed.
6. The `GameManager` increments the player's score and updates any wave-related counters. It then instructs the `UIManager` to refresh the on-screen score display.
7. At the end of the session (e.g., when the player runs out of health), the `GameManager` compares the final score against previously saved high scores via the `DataManager`. If a new high score is achieved, the `DataManager` persists it before the `UIManager` transitions to the Game Over screen.

## 3.7 Game State Management

AstroStreet AR uses an explicit game state model to keep the behavior of the `GameManager` predictable and easy to modify. The main high-level states are:

- **MainMenu**: The player is on the title screen and can choose to start a game, view high scores, open settings, or quit.

- **InitializingAR**: The game has received a request to start a new game and is waiting for the AR session to initialize and reach a stable tracking state.
- **Playing**: The AR session is active, enemies are spawning, and the player can move, aim, and fire.
- **Paused**: Gameplay is temporarily suspended; enemies and projectiles stop updating and the Pause Menu is visible.
- **GameOver**: The player has lost all health or another end condition has been reached; the final score is shown and may be saved.

State transitions are driven by player actions and AR session events. At a high level:

- **MainMenu** → **InitializingAR**: Player taps "Start Game".
- **InitializingAR** → **Playing**: The `ARController` reports that tracking has reached a stable state.
- **Playing** → **Paused**: Player taps the pause button or the app is backgrounded.
- **Paused** → **Playing**: Player selects "Resume" from the Pause Menu.
- **Playing** → **GameOver**: Player health reaches zero or another game-ending condition is met.
- **GameOver** → **MainMenu**: Player confirms going back to the main menu after reviewing their score.

The `GameManager` owns the current game state and exposes methods such as `StartNewGame()`, `PauseGame()`, and `EndGame()`. Other components (e.g., `UIManager`, `EnemyManager`) react to state changes via simple notifications, which keeps state transitions centralized and reduces the risk of inconsistent behavior.

## 3.8 Performance and Resource Considerations

As a mobile AR title, AstroStreet AR must balance visual fidelity with performance and battery usage. The following design decisions aim to keep the experience smooth on typical student devices:

- **Target Frame Rate**: The game targets at least 30 frames per second during gameplay. Enemy spawn rates, projectile counts, and particle effects are tuned so that the total number of active objects remains within a safe limit.
- **Object Pooling**: Frequently created and destroyed objects such as projectiles are good candidates for object pooling. Instead of allocating and freeing these objects every time, the system can reuse a small pool to reduce garbage collection overhead.

- **Capped Enemy Count**: The `EnemyManager` enforces a maximum number of active enemies at any given time. When the cap is reached, new enemies are not spawned until existing enemies are destroyed or despawned.
- **Lightweight UI**: HUD elements use simple text and shapes rather than heavy animations to minimize overdraw. This is particularly important because the camera feed itself already fills the background.
- **AR Query Throttling**: Expensive AR queries (such as plane detection or raycasting) are performed at a limited frequency instead of every frame, providing a balance between responsiveness and performance.

These considerations are not exhaustive, but they define a baseline performance profile that can be refined in future iterations without requiring a major rewrite of the core architecture.

## 3.9 Extensibility and Future Enhancements

Although the current implementation of AstroStreet AR focuses on a single–player, local high-score experience, the design intentionally leaves room for future extensions. Examples include:

- **Online Leaderboards**: The existing `HighScore` and `PlayerStats` entities can be synchronized with a remote service. The `DataManager` would act as a façade in front of both local and remote storage without requiring major changes to the rest of the game.
- **Additional Enemy Types and Behaviors**: New enemies with different movement patterns or attack styles can be added by extending the `EnemyManager` and related enemy components. The overall architecture (AR placement, game state management) remains unchanged.
- **New Game Modes**: Timed challenges, endless modes, or wave-based progression systems can be introduced primarily by updating the `GameManager`'s state transitions and scoring logic.
- **Accessibility Options**: Additional settings such as colorblind modes, adjustable HUD sizes, or alternative input schemes can be added by extending the `Settings` entity and updating the `UIManager`.

By documenting these potential extensions, the final SDD provides guidance for future contributors who may wish to grow AstroStreet AR beyond the scope of the course project.

# 4 User Interface

## 4.1 Overview

The user interface (UI) of *AstroStreet AR* is designed to be simple and usable outdoors while the player is holding a smartphone. Navigation is primarily tap-based, and the most important information (score, health, and basic controls) is always visible during gameplay.

The main UI elements include:

- A **main menu** for starting the game and accessing other screens.
- An **AR gameplay view** that overlays enemies and HUD elements on top of the camera feed.
- A **pause menu** for temporarily stopping gameplay.
- A **game over screen** that summarizes the player's performance.
- A **high scores** screen.
- A **settings** screen for basic configuration options.

## 4.2 Main Menu

When the application launches, the user is shown the main menu screen. This screen contains:

- The game title: **AstroStreet AR**.
- A brief subtitle or tagline (e.g., "Aim at the sky and blast incoming asteroids!").
- A set of buttons:
  - **Start Game** – begins a new AR gameplay session.
  - **High Scores** – opens the high scores screen.
  - **Settings** – opens the settings screen.
  - **Exit** (optional on platforms that support it) – closes the application.

The main menu is displayed on top of a static or subtly animated background image related to outer space or the night sky. From this screen, the player can reach all other major UI flows.

## 4.3 AR Gameplay Screen

The AR gameplay screen is the core of the user experience. It uses the device camera feed as the background and overlays virtual game elements on top.

The key elements of this screen are:

- **Camera View**: The live camera feed fills the entire screen to show the player's real-world surroundings.
- **Crosshair or Reticle**: A small graphical reticle is centered on the screen and indicates where the player's shots are aimed.
- **Score Display**: The current score is shown in the upper-left corner of the screen.
- **Health or Shield Bar**: The player's remaining health or shield is represented as a bar or set of segments in the upper-right corner.
- **Timer** (if used): If the game mode uses a time limit, the remaining time is displayed near the top of the screen.
- **Pause Button**: A small pause icon appears in one corner (e.g., top-right), allowing the user to open the pause menu.

To interact with this screen, the user:

- Physically rotates or tilts the phone to aim the reticle at enemies that are rendered in the AR view.
- Taps the screen to fire projectiles in the direction of the reticle.

Enemies (asteroids and alien ships), projectiles, explosions, and power-ups are all rendered as virtual objects anchored in the AR scene.

## 4.4 Pause Menu

The pause menu appears when the user taps the pause button during gameplay. When the game is paused, the AR scene is dimmed or frozen, and a small overlay panel is shown with the following options:

- **Resume** – returns to the AR gameplay screen and continues the session.
- **Settings** – opens a subset of settings (for example, sound on/off and sensitivity) without leaving the session.
- **Quit to Main Menu** – ends the current game and returns to the main menu.

## 4.5 Game Over Screen

When the player's health reaches zero or the game session ends for another reason (such as a time limit), the game over screen is shown.

This screen contains:

- A prominent "Game Over" title.
- The player's final score.
- An indication if a new high score was achieved.
- Buttons for:
  - **Play Again** – start a new game session.
  - **Return to Main Menu** – go back to the main menu.

## 4.6 High Scores Screen

The high scores screen displays a list of the top scores recorded on the device. A simple layout might include:

- A title such as "High Scores".
- A table or vertical list showing rank, score value, and date achieved.
- A button to return to the main menu.

## 4.7 Settings Screen

The settings screen allows the user to adjust basic configuration options for *AstroStreet AR*. Possible settings include:

- **Sound**: toggle sound effects on or off.
- **Control Sensitivity**: adjust how quickly the reticle responds to device movement.
- **Brightness or UI Contrast**: optional adjustments to help make the HUD more readable outdoors.

The settings screen is accessible from both the main menu and the pause menu, and includes a button to return to the previous screen.

Overall, the user interface is designed to minimize on-screen clutter so that players can clearly see both the real world and the virtual objects they are interacting with while playing *AstroStreet AR*.

### 4.8 UI Navigation Flow

The user interface is organized around a small set of screens connected in a simple, predictable flow. This subsection summarizes the primary navigation paths a player can follow.

- **Main Menu**:
  - **Start Game** → transitions to the **InitializingAR** state and then to the **AR Gameplay** screen once tracking is stable.
  - **High Scores** → opens the **High Scores** screen, where the player can review past scores and then return to the Main Menu.
  - **Settings** → opens the **Settings** screen, where the player can adjust audio and sensitivity options before returning to the Main Menu.
  - **Exit** → closes the application (if supported by the platform) or minimizes it.
- **AR Gameplay Screen**:
  - Displays the camera feed, HUD (score, health, wave), and pause button.
  - **Pause Button** → opens the **Pause Menu** while keeping the current game state in memory.
- **Pause Menu**:
  - **Resume** → returns to the AR Gameplay screen and restores the **Playing** state.
  - **Main Menu** → discards the current session and returns the player to the Main Menu.
- **Game Over Screen**:
  - Shows the final score and may indicate if a new high score was achieved.
  - Offers actions such as **Play Again** (start a new session) or **Main Menu**.
- **High Scores and Settings Screens**:
  - Both screens provide a simple “Back” or “Return” action to go back to the Main Menu.

This flow is intentionally minimal to keep cognitive load low when the player is outdoors and focusing on the AR gameplay environment.

# 5 Database Design and Explanation

## 5.1 Overview

AstroStreet AR uses lightweight local storage to persist player progress and configuration data on the device. The `DataManager` component provides a small API for saving and loading this data so that the rest of the system does not depend on any specific storage technology. In a production deployment, this could be implemented with a mobile database (e.g., SQLite) or a key–value storage mechanism provided by the game engine.

Version 2.0 of the SDD refines the database design by specifying explicit fields, types, and constraints for three logical entities:

- **HighScore**: stores individual high score entries.
- **PlayerStats**: stores aggregate player statistics across sessions.
- **Settings**: stores configurable game options such as audio and control sensitivity.

These entities are intentionally minimal to keep the project within the assignment scope while still supporting meaningful persistence and future extensions.

## 5.2 HighScore Entity

The `HighScore` entity stores individual completed game results that may be displayed on the High Scores screen. Although the v1 design only required a single best score, the refined schema supports storing multiple entries and sorting them for display.

| **Field** | **Type** | **Constraints** | **Description** |
|---|---|---|---|
| `id` | INTEGER | Primary key, auto-increment | Unique identifier for this high score entry. |
| `playerName` | TEXT | NOT NULL | Player initials or nickname associated with the score. |
| `score` | INTEGER | NOT NULL | Final score achieved at the end of a game session. |
| `dateTime` | TEXT | NOT NULL, ISO 8601 | Timestamp when the session ended, stored as an ISO 8601 string (e.g., `2025-12-08T13:45:00`). |

Typical operations on this entity include:

- Inserting a new high score at the end of a game session.
- Querying the top $N$ scores ordered by `score` descending.
- Deleting or truncating old entries beyond a fixed limit (e.g., keeping only the top 10 scores).

## 5.3 PlayerStats Entity

The `PlayerStats` entity aggregates statistics across multiple sessions. These statistics support future tuning of difficulty, progression systems, or player feedback features (e.g., accuracy percentage). Even if not fully exposed in the v1 implementation, designing this schema in v2 ensures the system can evolve without major architectural changes.

| **Field** | **Type** | **Constraints** | **Description** |
|---|---|---|---|
| `id` | INTEGER | Primary key, constant (e.g., always 1) | Identifier for the single row of aggregate statistics. |
| `totalGamesPlayed` | INTEGER | NOT NULL, default 0 | Number of games the player has completed. |
| `bestScore` | INTEGER | NOT NULL, default 0 | Highest score the player has ever achieved. |
| `totalShotsFired` | INTEGER | NOT NULL, default 0 | Cumulative number of projectiles fired by the player. |
| `totalHits` | INTEGER | NOT NULL, default 0 | Cumulative number of successful hits on enemies. |
| `totalPlayTimeSeconds` | INTEGER | NOT NULL, default 0 | Total time spent in active gameplay, measured in seconds. |

From these fields, the game can compute derived values such as average session length or firing accuracy. Updates to this entity typically occur at the end of each game session.

## 5.4 Settings Entity

The `Settings` entity stores user-configurable options. For simplicity, the design uses a key–value format that can be backed by a single table or a built-in key–value store in the engine.

| Field | Type | Constraints | Description |
|---|---|---|---|
| `key` | TEXT | Primary key, NOT NULL | Unique identifier for the setting (e.g., `"soundEnabled"`). |
| `value` | TEXT | NOT NULL | Serialized value for the setting (e.g., `"true"`, `"0.75"`). |

Common keys include:

- `"soundEnabled"`: `"true"` or `"false"`.
- `"musicVolume"`: numeric value in $[0.0, 1.0]$, stored as text.
- `"sensitivity"`: numeric value controlling look sensitivity.

The `DataManager` is responsible for converting these string values into the appropriate in-memory types when loading settings and for serializing them back to strings when saving.

## 5.5 Data Access Patterns

The `DataManager` provides a façade over the underlying storage and exposes simple method-style operations, such as:

- `SaveHighScore(entry)` and `GetTopHighScores(limit)`.
- `LoadPlayerStats()` and `SavePlayerStats(stats)`.
- `GetSetting(key, defaultValue)` and `SetSetting(key, value)`.

By centralizing all persistence logic inside the `DataManager`, the rest of the game logic remains decoupled from database details. This design also allows the storage implementation to be swapped (for example, moving from a local-only database to a remote leaderboard service) with minimal changes to the rest of the codebase.

# 6 Glossary

This section defines key terms and acronyms used throughout the *AstroStreet AR* Software Design Document.

| **Term** | **Definition** |
|---|---|
| AR (Augmented Reality) | A technology that overlays virtual objects (such as asteroids and alien ships) onto a live view of the real world, typically using a camera and device sensors. |
| UI (User Interface) | The visual elements and interactive controls that the player uses to navigate the application, such as buttons, menus, and icons. |
| HUD (Heads-Up Display) | On-screen information shown during gameplay, including the score, health or shield bar, timer, and other status indicators. |
| FPS (Frames Per Second) | The number of times per second that the game updates and redraws the screen; higher FPS generally results in smoother gameplay. |
| API (Application Programming Interface) | A defined set of functions or endpoints that one software component can use to communicate with another, such as an AR framework or a web service. |
| ARCore / ARKit | Platform-specific AR frameworks provided by Google (ARCore) and Apple (ARKit) that supply camera images, device pose, and tracking capabilities to mobile applications. |
| Component | A logical part of the system with a specific responsibility, such as **GameController**, **EnemyManager**, or **DataManager**. |
| Entity | A logical representation of a data object stored by the system, such as a **HighScore** or **Setting**. |
| Game Session | A single continuous playthrough that starts when the player begins a game and ends when the player loses, quits, or returns to the main menu. |
| Persistent Data | Information that is saved to storage so that it remains available after the application is closed, such as high scores and user settings. |

## 7 References

This section lists external resources, tools, and documentation that inspired or informed the design of *AstroStreet AR*.

## References

[1] Unity Technologies, *Unity Manual: AR Foundation*. Available at: https://docs.unity3d.com/Packages/com.unity.xr.arfoundation

[2] Google, *ARCore Developer Guide*. Available at: https://developers.google.com/ar

[3] Apple Inc., *ARKit Overview*. Available at: https://developer.apple.com/augmented-reality

[4] Jason Gregory, *Game Engine Architecture*. A K Peters/CRC Press, 2nd edition, 2014.

[5] Leslie Lamport, *LaTeX: A Document Preparation System*. Addison-Wesley, 2nd edition, 1994.