CANNON SHOOTER 2.0

DSA PROJECT REPORT

MUHAMMAD SUBHAN KHAN (CT-24075)

MUHAMMAD UZAIR (CT-24079)

SUNIL MAGHNANI (CT-24095)

INSTRUCTOR: MISS SAMIA MASOOD

ABSTRACT

This project is a small, polished 2D arcade game, *Cannon Shooter 2.0* built with SDL. The player controls cannon that can move **left** and **right** and **aim**, shooting enemies that spawn around the playfield. The game includes a friendly login and welcome flow, animated enemy sprites (or fallback shapes), soundless but smooth visuals, and clear on-screen feedback: health bars, score, level text, and victory/defeat screens that show earned stars. Controls are simple and intuitive, and enemies have basic AI that times their shots and aims at the player. Performance minded details such as a pooled bullet system (recycling bullet objects instead of repeatedly allocating) and texture initialization with error checks make the game responsive and robust.

On the systems side, player profiles and scores persist in a compact CSV-like storage and a leaderboard module that sorts, highlights the current player, and gracefully handles missing or malformed files. The codebase is neatly modular: separate classes handle bullets, enemies, player, login, game loop, and storage, which makes it easy to read, maintain, and extend (for example adding new enemy types, weapons, or sound). Overall, the project balances approachable gameplay with solid engineering choices which are ideal as a learning project or as a foundation to add polish like sound effects, power-ups, or online leaderboards.

INTRODUCTION:

In *Cannon Shooter*, you play as a cannon operator trying to survive against enemies coming from different sides. You move, aim, and shoot to protect yourself and clear each level. As you go forward, enemies get faster and harder to beat. You earn points by hitting them and staying alive. Your aim, timing, and quick moves decide how far you can go in the game.

GAMEPLAY MECHANICS:

1. Login & Profile Initialization

The game begins with a login screen rendered by Login::updateAndRender(), where the player enters a name.
On "START", main() loads existing profiles through StorageHash::loadFromFile("profiles.dat"), initializes or creates a Profile with score 0 if new, and displays a welcome screen showing current score/rank from StorageHash::getScore() and getRank().

2. Welcome Screen & Score Carryover

After login, a 2-second welcome screen (in main.cpp) shows the player's persistent score and leaderboard rank using profile data. Crucially, game.setPermanentScore(savedScore) stores the prior score Player::score, meaning each session starts fresh. Profile data is saved before gameplay begins through StorageHash::saveToFile().

3. Core Gameplay Loop

The Game class drives gameplay:
Game::handleEvents() processes movement (←/→), aim-and-shoot through S key (triggers shootDirection() toward nearest alive enemy), and mouse input for UI.
Game::update() runs physics, updates BulletManager::update() (checking collisions with Enemy::isAlive()), awards +10 per kill through player.addScore(), and triggers enemy respawns at milestones (e.g., spawnEnemies(2) after 1st/2nd kill).
Health-based damage: player.takeDamage() on enemy bullet hit (tracked through static BulletManager::playerHitThisFrame).

4. Enemy & Bullet System (Custom Pool-Based Manager)

Enemies (Enemy) are stored in a std::vector<Enemy>, each with health, alive state, and auto-firing logic (Enemy::shouldShoot() → shoot()). Bullets are managed through a custom fixed-size pool (BulletManager) using a circular free queue no dynamic allocation. Collision uses

SDL_HasIntersection(); on hit, Enemy::hit() sets alive = false and justKilled = true, which Game::update() detects to increment totalDeaths and score.

5. Win/Loss Conditions & Scoring

Victory occurs at 10 total kills (totalDeaths >= 10); defeat at 0 player health. Post-game, Game::render() shows stars based on kill ratio (e.g., 9+ kills = 3 stars), and saves the session score through StorageHash::insertOrUpdate() and saveToFile(). The "LEADERBOARD" button loads rankings through Leaderboard::loadFromFile(), which internally sorts profiles by score/name and renders top entries with current player highlighted.

6. Restart & Persistence Design

Pressing SPACE on end screens resets: enemies.clear(), player = Player(...), and spawnEnemies(3). Profile data is persisted across runs in profiles.dat (CSV: name,score). The leaderboard is reloaded on demand through leaderboard.loadFromFile(), ensuring live rank updates.

7. Player Mechanics & State Management

The Player class (initialized in Game::init() through Player(400, 610, 90, 90) and initTexture()) handles movement with handleInput() (←/→ keys), boundary clamping in update(), and aim-assist targeting when the S key is pressed which calculates the nearest alive enemy through distance-squared comparison and orients the cannon using setAngle() (converting vector to degrees). On hit, player.takeDamage() reduces health (max 3) and deducts 5 points (floored at 0); the health bar renders dynamically in Player::render(), changing color (green → yellow → red) as health drops.

ACHIEVEMENT AND REWARD SYSTEM:

The achievement and reward system tell rewards accurate play, survival, and clearing levels. Players get points when enemies die, lose a little score when they take damage, and earn a star rating at the end of a run. Final results (score and stars) are saved to the local storage and shown on the leaderboard so players can track progress over time.

Core Mechanics:

- **Points per kill:** Each enemy death gives **10 points** through player.addScore() (called in Game::update())
- **Penalty on hit**: Taking damage reduces health and subtracts **5 points** using Player::takeDamage() function.
- **End-of-run stars:** Star count is computed in Game::render() using totalDeaths / 10.0f.
- **Save & leaderboard:** Final score is stored and updated with storage.insertOrUpdate() and written to disk with storage.saveToFile("profiles.dat"); the leaderboard shows saved results.

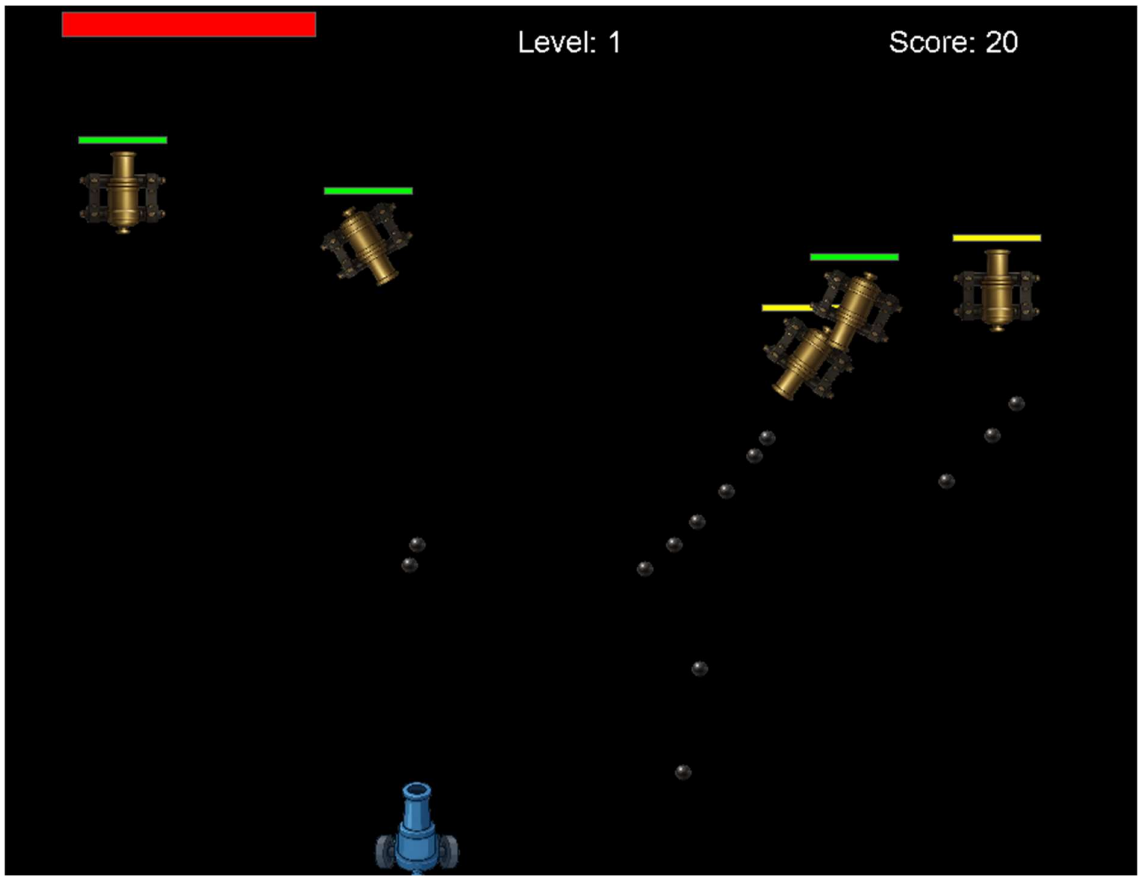
GAME MAP:

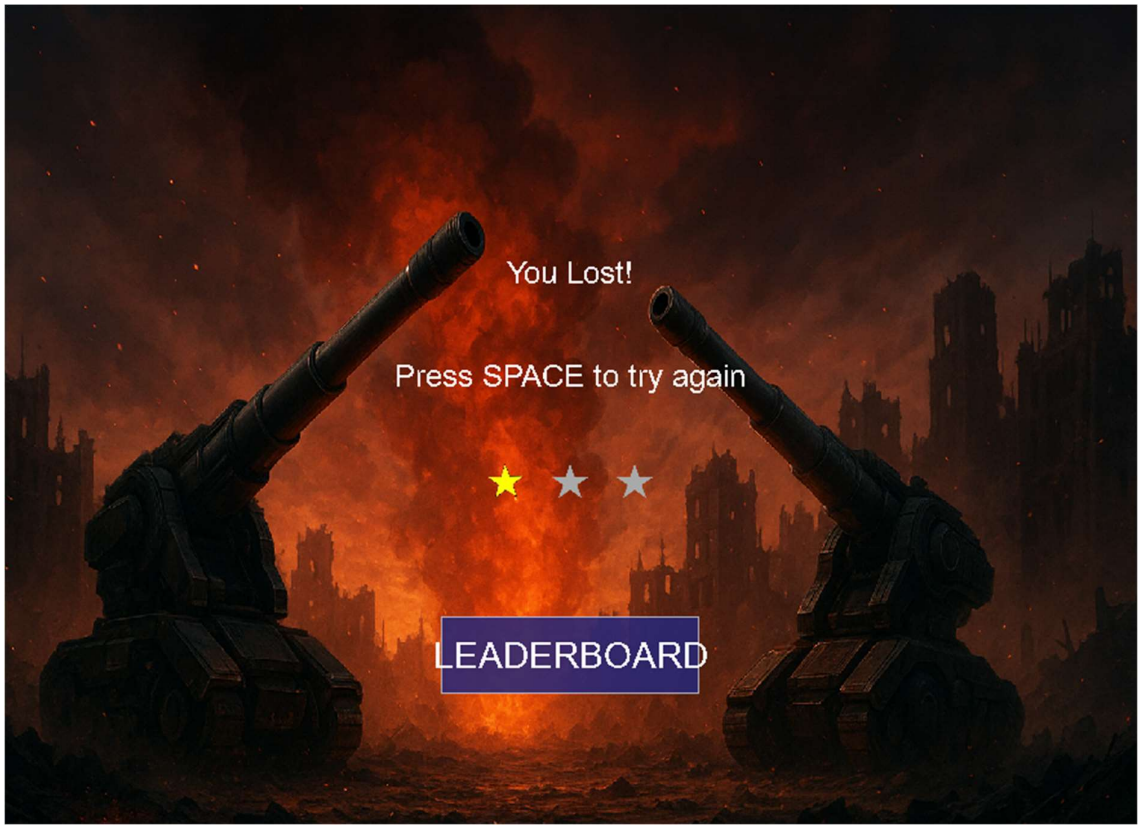
The game map is a simple 2D battlefield designed for clear and fast gameplay. The player stands on a flat ground area, while enemies appear from fixed points around the arena. Background images are used in menus and screens, but the fighting area stays plain to keep focus on action.

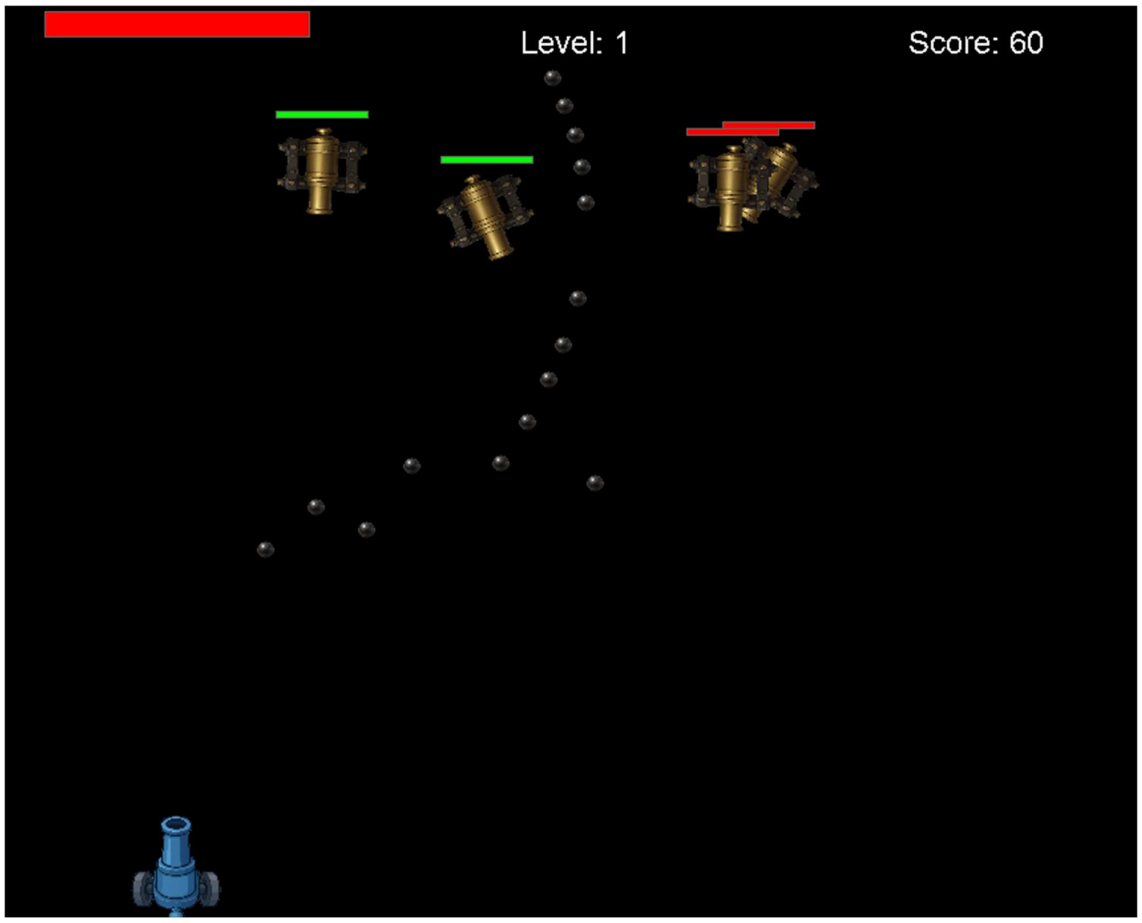
Key Areas:

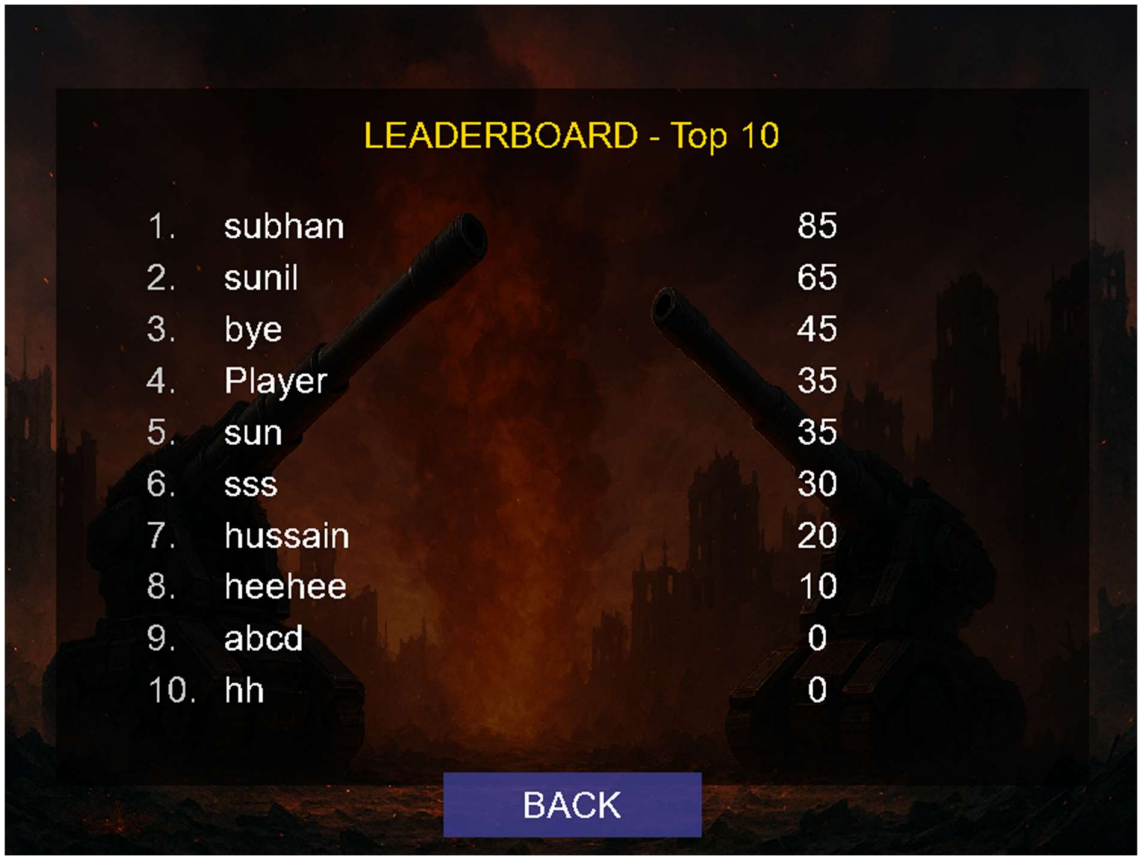
1. Player ground platform
2. Top enemy spawn point
3. Left enemy spawn point
4. Right enemy spawn point
5. Bottom support/spawn zone

Snippets:

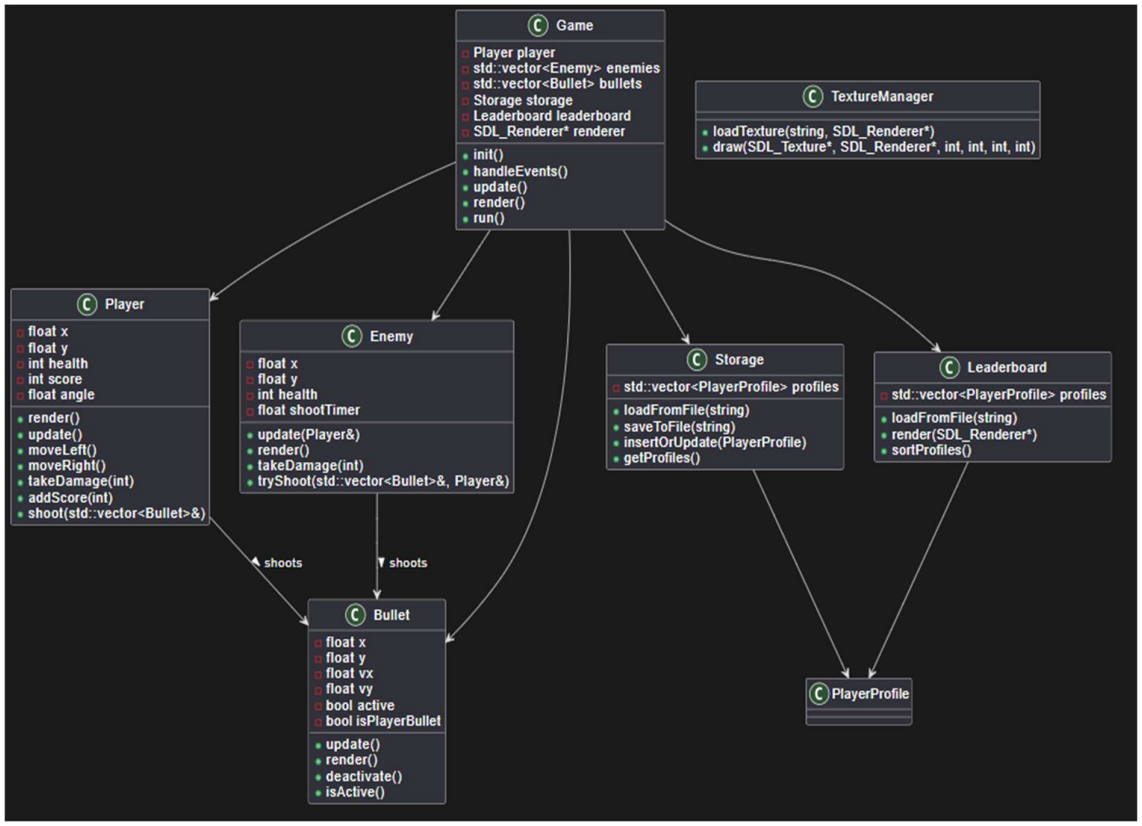



You Lost!
Press SPACE to try again
LEADERBOARD


Level: 1
Score: 60


LEADERBOARD - Top 10
1. subhan 85
2. sunil 65
3. bye 45
4. Player 35
5. sun 35
6. sss 30
7. hussain 20
8. heehee 10
9. abcd 0
10. hh 0
BACK

UML DIAGRAM:


Game
Player player
std::vector<Enemy> enemies
std::vector<Bullet> bullets
Storage storage
Leaderboard leaderboard
SDL_Renderer* renderer
init()
handleEvents()
update()
render()
run()
TextureManager
loadTexture(string, SDL_Renderer*)
draw(SDL_Texture*, SDL_Renderer*, int, int, int, int)
Player
float x
float y
int health
int score
float angle
render()
update()
moveLeft()
moveRight()
takeDamage(int)
addScore(int)
shoot(std::vector<Bullet>&)
Enemy
float x
float y
int health
float shootTimer
update(Player&)
render()
takeDamage(int)
tryShoot(std::vector<Bullet>&, Player&)
Storage
std::vector<PlayerProfile> profiles
loadFromFile(string)
saveToFile(string)
insertOrUpdate(PlayerProfile)
getProfiles()
Leaderboard
std::vector<PlayerProfile> profiles
loadFromFile(string)
render(SDL_Renderer*)
sortProfiles()
shoots
shoots
Bullet
float x
float y
float vx
float vy
bool active
bool isPlayerBullet
update()
render()
deactivate()
isActive()
PlayerProfile

CONTRIBUTIONS:

1. Muhammad Uzair:
   (BULLET SYSTEM, STORAGE SYSTEM & GRAPHICS SUPPORT)

Uzair implemented the bullet management system, which handles shooting, movement, collision, and recycling of bullets to ensure smooth gameplay. He also built the game's profile storage system, enabling player data to be saved, updated, and loaded across sessions. His work ensured fast, organized score tracking and efficient data access. Uzair further supported graphical integration by managing textures and visual consistency, helping bind the gameplay elements together. Additionally, he handled the preparation and formatting of the project report.

2. Sunil Maghnani:
   (LEADERBOARD, LOGIN SYSTEM & GAME INTEGRATION)

Sunil was responsible for building the login system and designing the leaderboard feature of the game. He created the complete flow for entering player names, validating input, and transitioning from the login screen into the game. He also developed the leaderboard interface, including the layout, profile handling, and score sorting. Sunil integrated both systems into the main game loop so that the player's name, score, and ranking carried smoothly across screens.

3. Muhammad Subhan Khan:
   (MAIN GAME STRUCTURE, ENEMY & PLAYER MECHANICS)

Subhan developed the core gameplay components, including player behavior, enemy logic, and overall game flow. He established the main structural logic of the game, including initialization, event handling, updates, and transitions between game states. Subhan integrated the enemy and player modules into the central system, ensuring smooth gameplay, responsive controls, and balanced difficulty. He also handled the main program setup and connected all major features into the final working build.

Data Structures & Their Justification

1. std::unordered_map<std::string, Profile> (in StorageHash)

Used as the core profile storage backend through the static g_profiles map
(key = normalized lowercase name, value = Profile).

Justification:

- It gives O(1) average-case lookup/insert/update for login (find()), score retrieval (getScore()), and saving (insertOrUpdate()), critical for responsive UI.
- It Handles case-insensitive, whitespace-robust name matching through key normalization (toLower(trim())), ensuring " Alice " and "alice" map to the same entry a practical UX necessity.

2. std::vector<Enemy> (in Game class)

Stores all active and inactive enemy instances in contiguous memory.

Justification:

- Cache-friendly iteration in Game::update() and BulletManager::update() enemies are looped over every frame; vector's locality minimizes cache misses.
- Simple lifetime management: enemies.clear() on restart fully resets state; no need for complex ownership (e.g., pointers/unique_ptr). Dead enemies are skipped through isAlive() guard efficient for small N (≤10).

3. Fixed-Size Object Pool + Circular Queue (in BulletManager)

Implements bullet allocation using:
Bullet bullets[MAX_BULLETS] (static array no heap allocations)
int freeQueue[MAX_BULLETS], front, rear, count (circular buffer for free indices).

Justification:

Zero dynamic allocation during gameplay eliminates frame hitch from new/delete, crucial for 60 FPS stability. Deterministic performance & memory footprint ideal for real-time systems; avoids fragmentation and ensures worst-case O(1) shoot()/freeBullet().

4. std::vector<Profile> + Sort (in StorageHash::getRank() & Leaderboard)

Used to generate ranked leaderboards by sorting profiles through custom comparator.

Justification:

- Simplicity & readability for small datasets (≤100 players) std::sort is highly optimized and easier to audit than custom trees.
- Enables stable ranking with secondary sort (score ↓, then name ↑) in one pass essential for fair tiebreaking, as seen in Leaderboard::loadFromFile().

5. (Implicit) Struct-Based Composition

Profile, Bullet, SDL_Rect are all plain structs/classes with value semantics.

Justification:

- No indirection overhead copying/moving small structs (e.g., Profile) is faster than pointer chasing.
- RAII alignment: Resources (textures, fonts) are owned by classes (Game, Player) and cleaned in destructors no dangling pointers.

6. Singly Linked List of Active Bullets (in BulletManager)

Every active bullet is linked through Bullet::next, with BulletManager::activeHead pointing to the most recently fired bullet.

Used in:

update(): traverses list to move/hit-test bullets

render(): draws all live bullets

shoot(): inserts at head (b.next = activeHead; activeHead = &b;).

Justification:

O(1) insertion and traversal without array shifting ideal for dynamic bullet counts.

Zero wasted iteration unlike scanning a full array, the list only visits active bullets (e.g., 3 bullets = 3 loop steps), making collision and rendering highly efficient even with 100 pooled slots.