Outpost Zero

Project Report

Author:
Mayookh Ghosh Chowdhury

November 14, 2025

Contents

1 Introduction

Outpost Zero is a two-dimensional tower defense game developed in **C++** using the **raylib** library. The project's primary goal was to design and implement a complete, playable game from the ground up, demonstrating core object oriented principles and gameplay mechanics.

The fundamental problem presented to the player is to defend a strategic outpost from waves of hostile entities. These enemies spawn at a designated entry point and travel along a pre-defined path toward a base. The player must strategically place and upgrade a variety of defensive turrets to eliminate the enemies before they reach the end of the path. Each enemy that successfully traverses the path depletes the player's base health. The game is lost if the base health reaches zero.

The game is designed to be challenging and to reward strategic thinking. Players must manage a limited economy, make critical decisions about which turrets to build and upgrade, and adapt their defensive strategy to counter an increasingly difficult and diverse roster of enemies. The inclusion of multi-stage difficulty scaling and climactic boss battles ensures that the gameplay remains engaging and demanding into the late game.

Source code of project

2 Tools and Technologies Used

Category	Tool / Library
Programming Language	C++
Graphics Library	Raylib
Editor / IDE	Neovim
Operating System	Ubuntu
Version Control	Git & GitHub
Build System	GNU Make

Table 1: Tools and Technologies

3 Proposed Approach & System Architecture

The architectural approach for Outpost Zero was centered on creating a modular, scalable, and maintainable codebase. The design philosophy was to separate distinct game logic into self-contained modules and manage game objects through a centralized, polymorphic entity system.

3.1 Scene-Based Game Management

The highest level of control is a scene manager implemented in `main.cpp`. The game is broken down into distinct scenes (e.g., `INTRO`, `GAME`, `SETTINGS`), with a main game loop that delegates updates and rendering to the currently active scene. This design allows for easy expansion, such as adding a main menu, pause screen, or game-over screen, without altering the core gameplay logic.

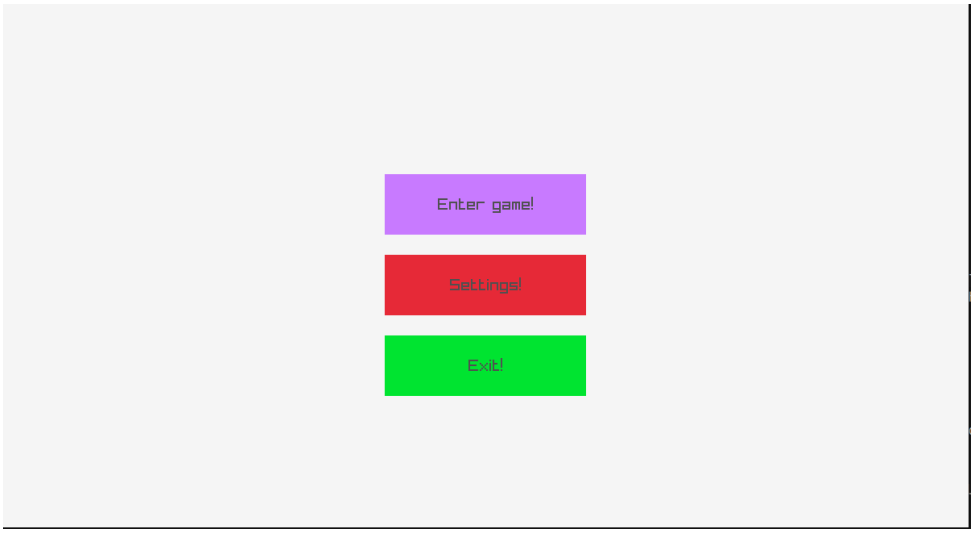

Figure 1: Scene Management Diagram

3.2 Core Gameplay Loop

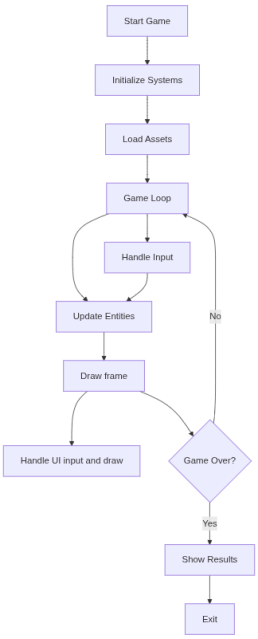

Figure 2: Core Gameplay Loop

3.3 Inheritance-Based Entity System

At the heart of the game's architecture is an entity management system inspired by the Entity-Component-System (ECS) pattern, adapted for simplicity using an inheritance model.

- **Base `Entity` Class:** All dynamic game objects, including `Turret`, `Enemy`, and `Projectile`, inherit from a common abstract `Entity` class. This base class defines a common interface with pure virtual functions like `Update()` and `Draw()`.

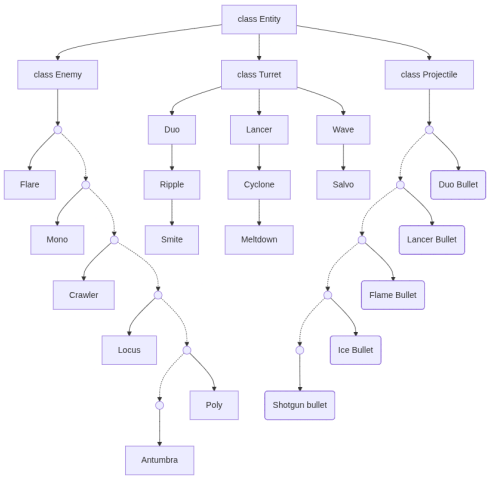

Figure 3: Entity Class Diagram

- **Polymorphic Management:** All entities are stored in a single, centralized container within the main game scene: `std::vector<std::unique_ptr<Entity>>`.

This approach has several key advantages:

1. **Simplified Game Loop:** The main loop can iterate over this single vector to update and draw every object on screen, regardless of its specific type. The use of virtual functions ensures the correct `Update()` and `Draw()` implementation is called for each entity.
2. **Automatic Memory Management:** By using `std::unique_ptr`, we leverage modern C++ features for robust memory management. When an entity is removed from the vector (e.g., a destroyed enemy or an expired projectile), its memory is automatically deallocated, preventing memory leaks.

- **Specialized Update Passes:** While the base `Update()` and `Draw()` calls are polymorphic, more complex interactions are handled in specialized passes. For example, after the initial update pass, the game creates temporary vectors of raw pointers (`std::vector<Enemy*>`, `std::vector<Turret*>`) to facilitate the targeting logic, where turrets need a list of all active enemies.

3.4 Modular Code Design

The project is broken down into several key modules, each with a clear and distinct responsibility:

- `Turret.h / Enemy.h / Projectile.h`**:** These headers define the class hierarchies for the core gameplay objects. Each file contains the base class and all its derived subclasses (e.g., `duo_turret`, `lancer_turret`, etc.). This keeps related logic encapsulated and easy to navigate.
- `Wave.h`**:** The `WaveManager` class acts as a state machine (`WAITING`, `SPAWNING`, `WAVE_IN_PROGRESS`) that orchestrates the entire flow of enemy spawns. It reads from a pre-defined script of `SpawnCommand` structs, managing the type of enemy to spawn and the delay between each spawn. It is also responsible for tracking game stages and triggering the global increase in enemy difficulty.
- `Map.h`**:** This module handles the procedural generation and rendering of the game grid. It defines the path and buildable tiles and provides a simple interface for other systems to query tile information (e.g., `getTileFromMouse()`).
- `Config.h` & `Config.cpp`**:** All magic numbers and balancing variables (turret costs, enemy health, damage values, etc.) are centralized in these files. This is a critical design choice that allows for rapid iteration and fine-tuning of game balance without having to search through the entire codebase.
- `utils.h`**:** A collection of stateless helper functions, primarily for mathematical calculations (e.g., `CalculateInterceptPoint`, `GetDamageFalloff`). These are provided as `inline` functions in a single header to simplify the build process and avoid linker errors related to multiple definitions.
- **GUI (**`raygui.h`**):** The user interface is handled by the `raygui` library. UI elements like buttons and info panels are drawn at the end of each frame. This approach keeps UI logic separate from gameplay simulation and is easy to scale and debug.

4 Data Types and Structures Utilized

The selection of data structures was crucial for creating a system that is both efficient and easy to manage.

4.1 Core Classes and Inheritance

- `class Entity`: The abstract base for all game objects. Its interface (`virtual void Update()`, `virtual void Draw()`) is the foundation of the polymorphic game loop.
- `class Turret : public Entity`: The base class for all defensive towers. It manages common properties like `range`, `fireTimer`, and `gunRotation`.
  - **Subclasses (e.g., `duo_turret`, `cyclone_turret`):** Each specific turret type is a subclass that overrides the virtual functions to implement its unique behavior (e.g., firing a single projectile vs. a continuous beam) and drawing logic.
- `class Enemy : public Entity`: The base class for all hostile units. It handles common logic such as pathfinding (by iterating through a vector of target coordinates), health management, and applying status effects.
  - **Subclasses (e.g., `mono_enemy`, `locus_enemy`, `antumbra_enemy`):** Each enemy type implements its own `Draw()` method and is initialized with unique stats (health, speed, reward) from the `Config` file.
- `class Projectile : public Entity`: The base class for all fired projectiles. It manages properties like `pierce_count` and lifetime.
- `class WaveManager`: Responsible for orchestrating enemy spawns throughout the game. It processes a predefined sequence of `SpawnCommand` instances and controls progression-related parameters such as the stage counter and the global enemy health multiplier. Through these mechanics, it regulates the overall difficulty curve of the game.
- `class StatsManager`: Tracks and aggregates all gameplay statistics, including total damage dealt, total enemies defeated, per-enemy-type kill counts, and total currency spent. These collected statistics are presented on the Game Over screen to summarize the player's performance.

4.2 Key Structs and Enums

- `struct Tile`: A fundamental data structure in `Map.h` that holds the `Rectangle` for its position, its `TileType` (PATH, BUILDABLE), and its state (`hasTurret`).
- `struct SpawnCommand`: Used by the `WaveManager`, this struct pairs an `EnemyType` enum with a `float delayUntilNext`, creating a simple yet powerful scripting system for designing waves.
- `struct Particle`: A lightweight data structure used by the `Projectile` class to represent an individual visual particle. It stores attributes such as position, color, velocity, lifetime, and size. Groups of these particles are spawned to create visual effects, with randomized parameters providing a more natural and dynamic appearance.

- `enum class TurretType`, `enum class EnemyType`, `enum class ProjectileType`: These strongly-typed enums are used throughout the codebase to clearly and safely identify different types of entities. This is far more robust than using integers or strings, as it provides compile-time checking and prevents errors.

4.3 Major Data Structures

- `std::vector<std::unique_ptr<Entity>> entities`: This is the single most important data structure in the game.
  - `std::vector`: Chosen for its contiguous memory layout (which can be cache-friendly) and efficient iteration, which is the most common operation performed on it (updating and drawing every frame). While insertions (spawning) and deletions (destroying) occur, they are batched and handled once per frame, making the performance acceptable.
  - `std::unique_ptr`: Chosen to enforce single ownership of each entity and automate memory management. This is a modern C++ best practice that eliminates the need for manual `new` and `delete`, drastically reducing the risk of memory leaks.
- `std::vector<std::vector<Tile>> grid`: A 2D vector used in the `Map` class to represent the game grid. This provides a simple and intuitive way to access any tile using `grid[row][col]`.
- `std::vector<Vector2> targets`: A simple vector of coordinates that defines the enemy path. Enemies iterate through this vector, moving from one point to the next.
- `std::unordered_set<int> current_colliding`: Used within the `Projectile` class, this hash set efficiently tracks the IDs of enemies a projectile is currently colliding with. This prevents a single piercing projectile from dealing damage to the same enemy multiple times in subsequent frames. An `unordered_set` was chosen for its average O(1) insertion and lookup time.

5 Gameplay and Features

This game provides a complete gameplay loop that demonstrates the successful integration of the systems described above.

1. **Building and Strategy:** The player starts with a set amount of currency. The UI, drawn with `raygui`, presents buttons for building the three base turret types (`Duo`, `Lancer`, `Wave`). When a build button is clicked, the UI panel displays the stats and cost of the selected turret, allowing the player to make an informed decision. The player can then place the turret on any valid "buildable" tile.

2. **Wave Combat:** The player initiates the next wave via a UI button. The `WaveManager` begins executing its spawn script, creating `Enemy` objects and adding them to the main entity vector.

   - Turrets automatically detect and fire upon enemies within their range. The `Turret::Update` logic finds the closest target and, for projectile-based turrets, uses the `CalculateInterceptPoint` utility to lead its shots.
   - Specialized turrets perform their unique actions. The `cyclone_turret` fires a piercing beam that damages multiple enemies in a line, the `meltdown_turret` fires a sustained beam from turret mouth to range which damages everything it intersects with, the `wave_turret` applies a `SLOWED` status effect to all enemies within its radius.
   - Projectile and enemy collisions are checked based on a hashing set that stores currently colliding entities so that multiple collisions are not detected in the same frame, and accordingly the enemy's `TakeDamage` method is called, its health is reduced, and a particle effect is spawned. Upon death, the enemy is removed from the game and the player is awarded currency.
   - Some turrets that have specialized functions handle collision with enemies on their own, inside their `Update()` loop, calculate damage to deal using `GetDamageFalloff()` and `enemies_hit`.

3. **Upgrading:** When a placed turret is clicked, the UI panel switches to an upgrade/sell menu. This menu displays the stats for the next turret in its upgrade path and the associated cost. If the player has enough currency, they can click the "Upgrade" button. This action destroys the old turret entity and creates a new, more powerful one in its place.

4. **Scaling and Bosses:** The game progresses through stages. After every 10 waves, the `WaveManager` increases a global `enemy_health_multiplier` and spawns a boss wave. For example, Wave 10 features the powerful `Antumbra` boss, which appears alongside supporting minions. Defeating the boss is a major challenge that requires a well-upgraded and diversified defense. Successfully clearing the boss wave marks the beginning of the next, harder stage. A UI counter keeps the player informed of when the next boss will appear, creating anticipation and driving the player to strengthen their defenses.

5.1 Relevant screenshots

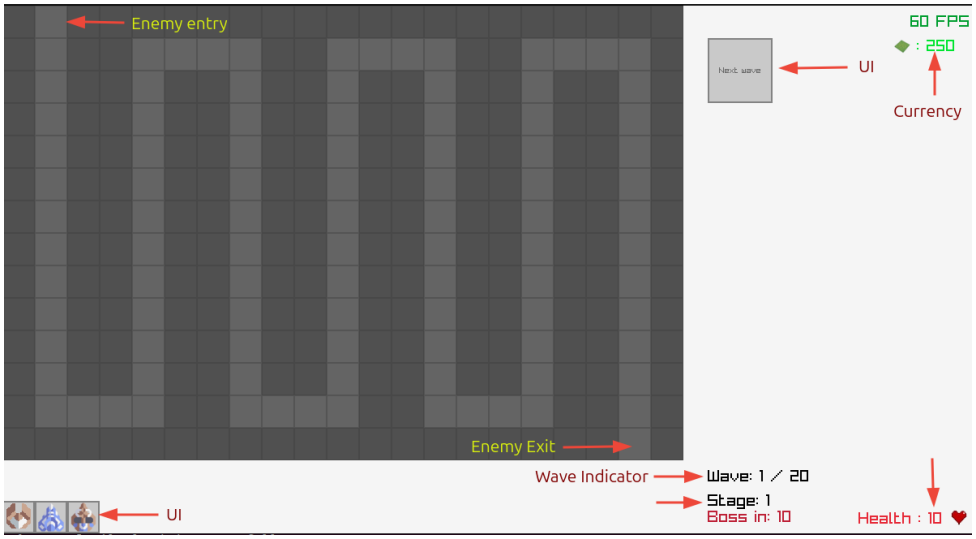

Figure 4: Start of game

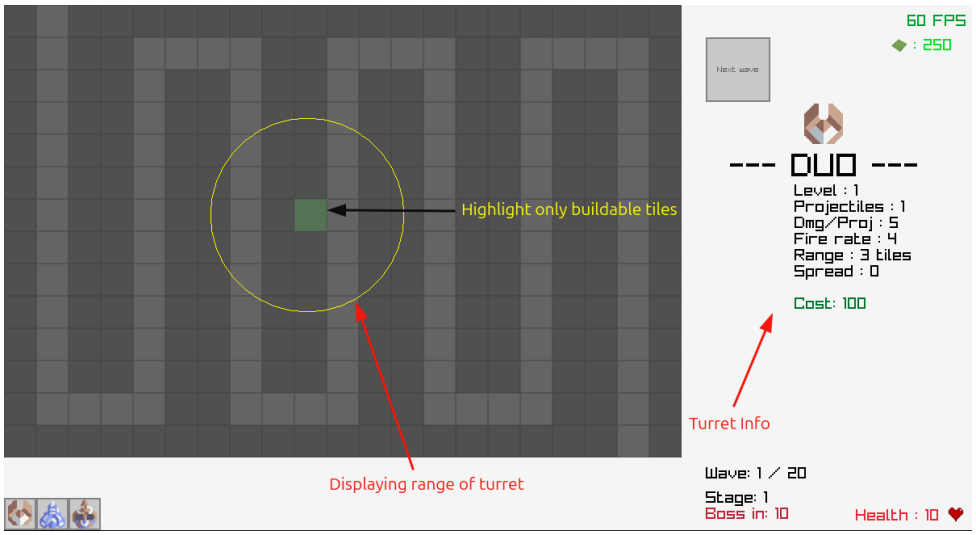

Figure 5: UI elements in the game

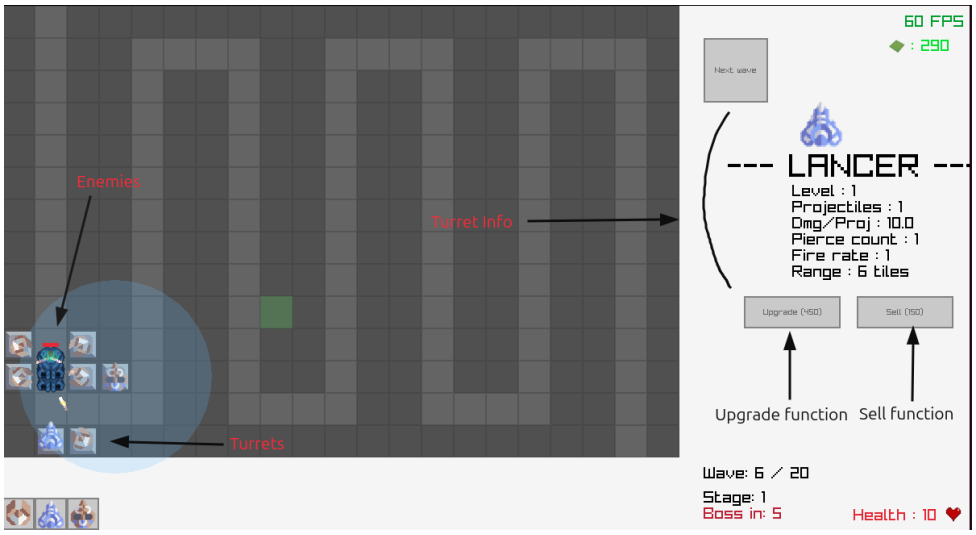

Figure 6: Combat in the game

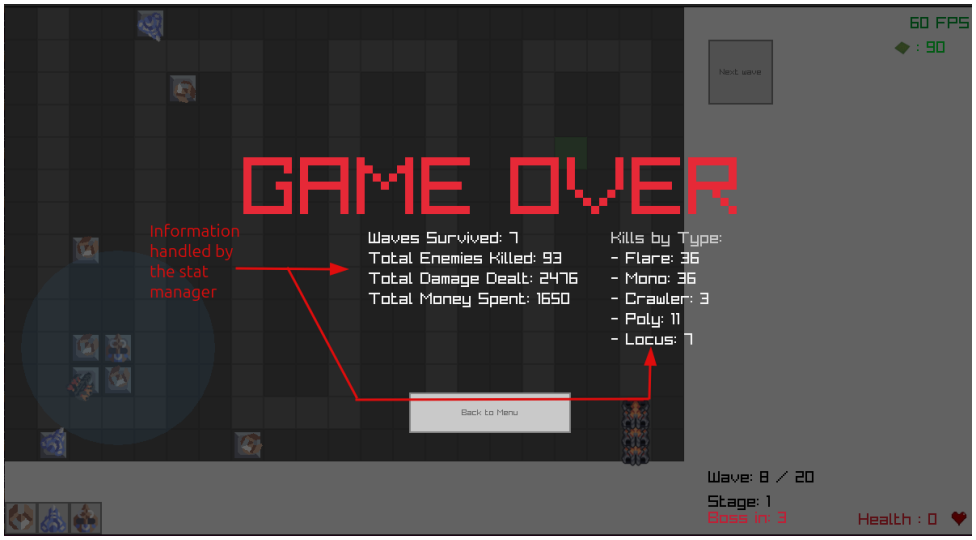

Figure 7: Game Over screen

6 Gameplay Elements: Turrets and Enemies

The strategic depth of Outpost Zero is built upon the interaction between its diverse roster of defensive turrets and hostile enemies. Each unit is designed with specific strengths and weaknesses, encouraging the player to adapt their strategy rather than relying on a single solution.

6.1 Defensive Turrets

The player has access to three base turret types, each of which can be upgraded along a distinct path. This results in a total of eight unique turrets, categorized by their function.

6.1.1 Projectile Turret Path (Crowd Control)

This path focuses on dealing with large numbers of enemies through high fire rates and area-of-effect damage.

- **Duo (Tier 1):** The foundational turret. It is cost-effective and fires single, low-damage projectiles at a high rate. It is reliable for handling the early waves of weak enemies but is quickly overwhelmed by durable or numerous foes.
- **Ripple (Tier 2):** An upgrade that transforms the turret into a short-range flamethrower. It fires a continuous stream of flame projectiles, making it exceptionally effective at clearing dense swarms of low-health enemies.
- **Smite (Tier 3):** The final projectile upgrade. The Smite functions as a powerful shotgun, firing a wide spread of ten high-pierce projectiles per shot. It excels at dealing massive burst damage to enemies clustered at close range.

6.1.2 Laser Turret Path (High-Value Target Elimination)

This path specializes in high-damage, long-range attacks, ideal for eliminating durable, high-priority targets.

- **Lancer (Tier 1):** A precision weapon that fires a high-velocity laser bolt. It offers higher single-target damage and longer range than the Duo, making it an effective counter to more resilient early-game enemies.
- **Cyclone (Tier 2):** A significant strategic upgrade. The Cyclone fires an instantaneous, long-range beam that pierces multiple enemies. Its damage falls off with distance, rewarding placements that allow it to hit enemies in a straight line close to the turret. Its slow fire rate is balanced by its high base damage and piercing capability.
- **Meltdown (Tier 3):** The pinnacle of single-target damage. The Meltdown charges up and fires a sustained, high-DPS beam at a single target for several seconds before entering a cooldown period. It is designed to melt the health of the toughest boss enemies.

6.1.3 Status Effect Turret Path (Utility)

This path focuses on utility, applying status effects to enemies to make them more vulnerable to other turrets.

- **Wave (Tier 1):** This turret does not deal damage directly. Instead, it periodically emits a pulse that applies a `SLOWED` status effect to all enemies within its range. This is critical for managing fast-moving swarms and revealing invisible units.
- **Salvo (Tier 2):** An upgrade to the Wave turret that adds an offensive capability. It fires a rapid stream of ice projectiles that deal minor damage but also apply the `SLOWED` effect on hit, allowing for targeted crowd control.

6.2 Hostile Enemies

The game features a variety of enemies, each designed to test a different aspect of the player's defense.

- **Flare:** The standard grunt enemy. It is relatively slow and has low health, serving as the primary threat in early waves.
- **Mono:** A fast-moving but fragile enemy. Monos appear in swarms and are designed to rush past defenses that have a slow fire rate or poor targeting priority.
- **Locus:** A durable tank enemy. It moves slowly but has a very high health pool, requiring sustained, high-damage fire to eliminate. It acts as a "damage sponge" that can shield other enemies behind it.
- **Poly:** A dedicated support unit. It moves at the same speed as the Locus and periodically heals all nearby non-Poly enemies. Its presence can make durable enemies significantly harder to kill, making the Poly a high-priority target.
- **Crawler:** A stealth unit. It is invisible to most turrets and can only be targeted when revealed by the `SLOWED` status effect (from a Wave or Salvo turret) or when it takes damage.
- **Antumbra (Boss):** A massive, exceptionally durable boss enemy. It is very slow but has an enormous health pool, serving as the ultimate test of a player's defensive setup at the end of each game stage.

7 Relevant Code snippets

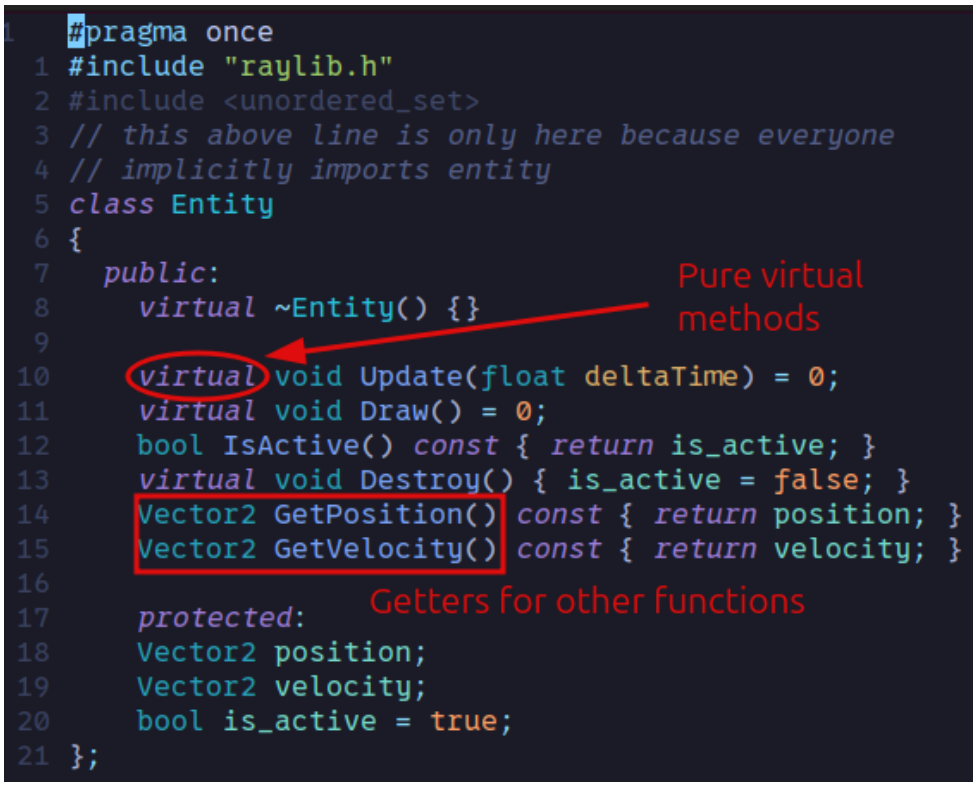

Figure 8: Code for entity class showing virtual functions

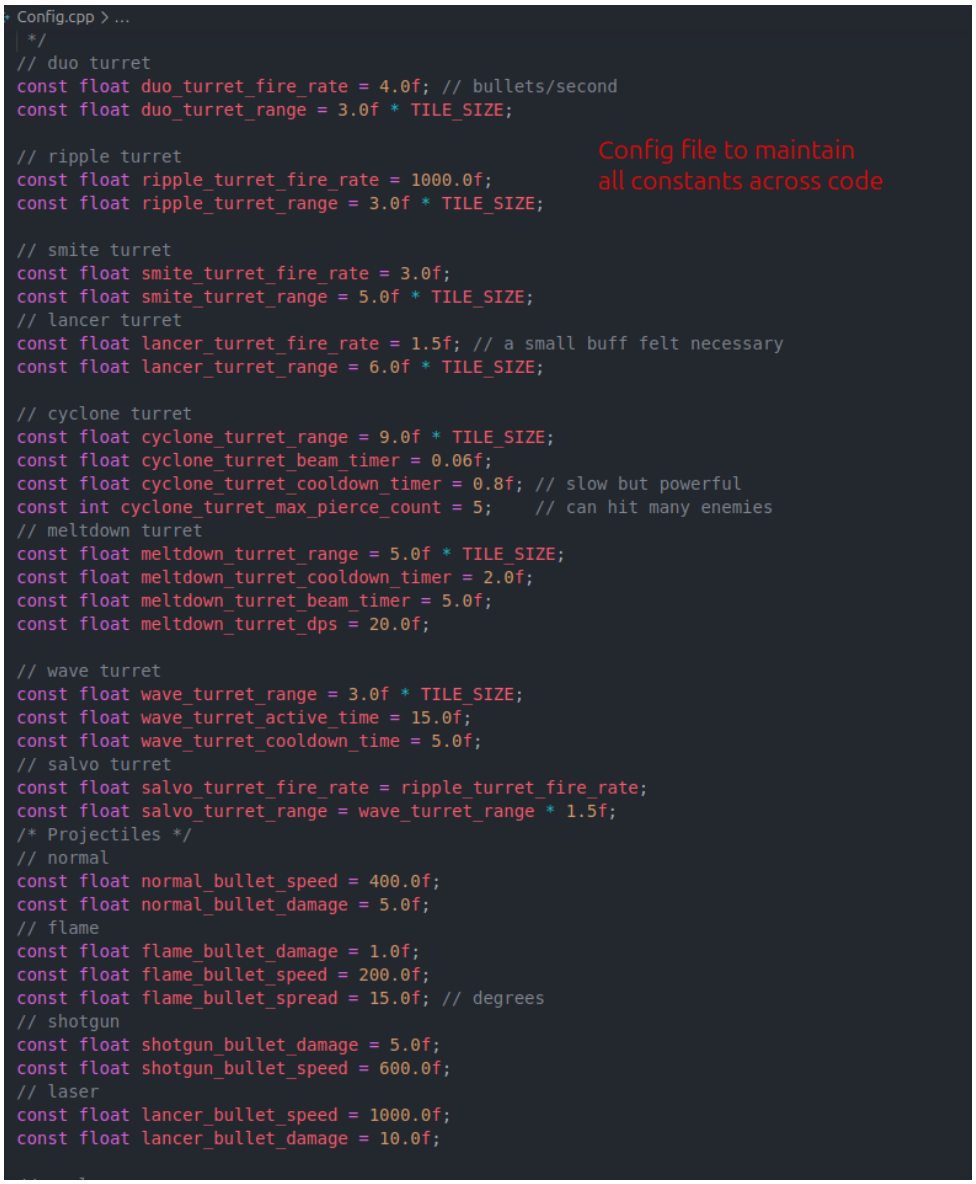

Figure 9: `Config.cpp` file that stores global constants

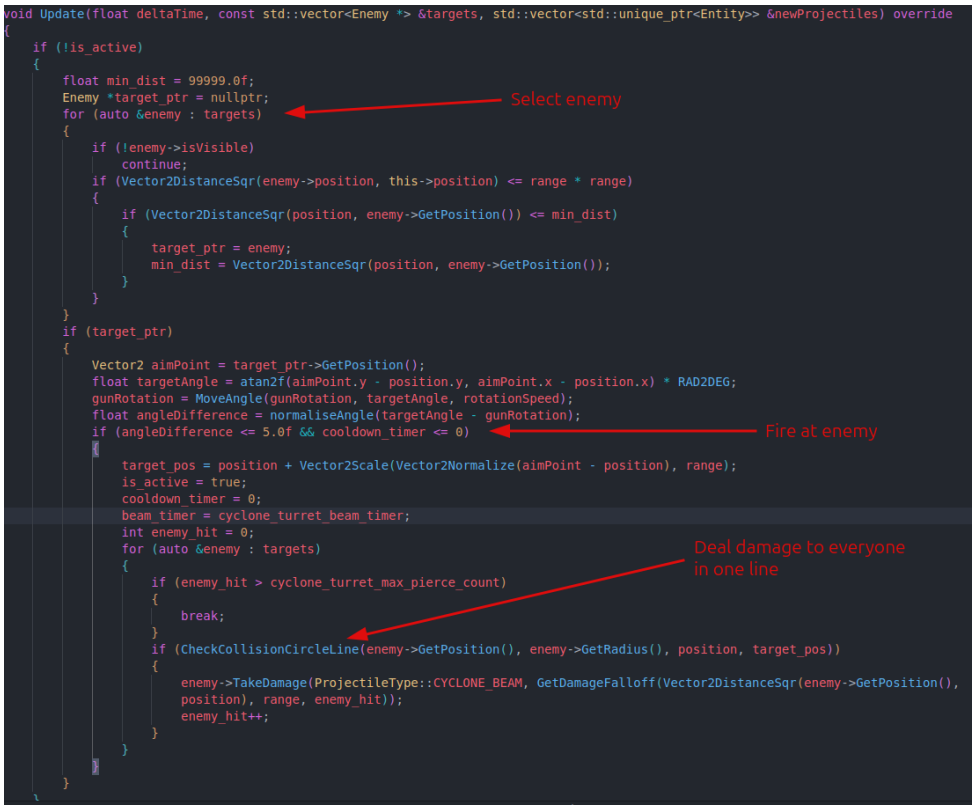

Figure 10: `Update()` of the cyclone turret

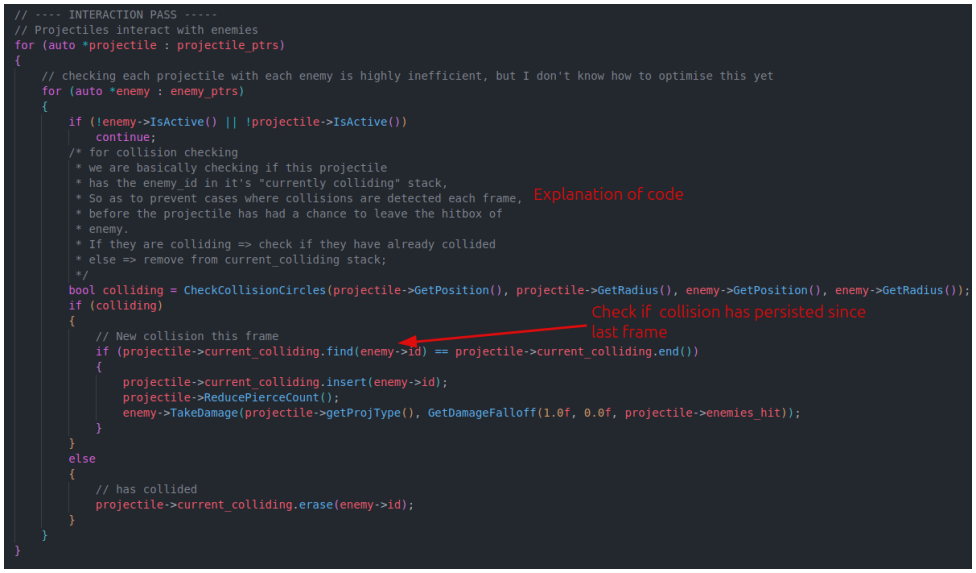

Figure 11: Collision check in `Game.cpp`

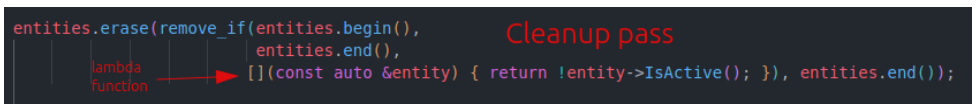

Figure 12: Cleanup pass for inactive entities

8 Conclusion and Future Work

8.1 Conclusion

Working on **Outpost Zero** was a huge learning experience for me, both technically and creatively. When I started, my main goal was just to make something that moved on screen and reacted to input. But as the project grew, I had to think about how real games are structured under the hood; how every system, from enemy spawning to collision detection, has to work together in sync. I learned how to design classes that actually make sense, how to use inheritance and polymorphism properly, and how to manage memory safely using smart pointers instead of plain references.

One of the biggest lessons came from balancing the game. What seemed simple: adjusting turret range, enemy health, or spawn rate; turned out to change the difficulty and pacing completely. It showed me how much iteration goes into fine-tuning a playable game. I also realized that modular code is worth the effort: having separate systems for enemies, turrets, projectiles, and UI made testing and debugging so much easier later on.

There were plenty of frustrating moments too: weird bugs, logic errors, or things breaking for no clear reason; but each one forced me to understand the code more deeply instead of just patching over problems. It developed my understanding of the compiler, the linker, and most importantly, the debugger, which was a crucial tool in the finishing of this project. By the end, I didn't just have a game that runs; I had a clearer idea of how professional games are built layer by layer. This project gave me confidence in my ability to plan, code, debug, and polish something from start to finish, and it's definitely the kind of project that made me want to keep improving as a developer.

8.2 Future Work

While the current version of Outpost Zero is a complete experience, its modular architecture provides a strong foundation for future expansion. Potential areas for future development include:

- **Expanded Wave Design:** The current 20 waves can be expanded into a much longer, potentially endless mode. Future waves could be designed to be even more complex, featuring synchronized multi-lane attacks or environmental challenges.
- **New Enemy Types:** The introduction of new enemies with unique abilities, such as a "Disruptor" that temporarily disables turrets or a "Summoner" that spawns weaker minions, could add new strategic layers to the late game.
- **Further Turret Evolution:** The `Salvo` turret currently ends its upgrade path at Tier 2. A Tier 3 upgrade could be implemented, and entirely new turret categories could be added to increase strategic variety.
- **Code Refactoring and Optimization:**
  - The `DrawTurretInfo()` and `DrawBuildInfo()` methods in `Turret.h` contain significant code duplication. They could be refactored into a more generalized drawing function to improve maintainability.

  - The collision detection loop in `Game.cpp`, which checks every projectile against every enemy, could be optimized to improve late-game performance.

- **Quality-of-Life Enhancements:** The addition of sound effects for turret fire and enemy destruction, background music, and a more detailed main menu with save/load functionality would greatly enhance the overall player experience.

9 Acknowledgements

- Several external resources were used while developing this project.
- Some textures were adapted from the open-source game **Mindustry**.
- Parts of the build system were inspired by various public Makefiles available on GitHub as well as official Raylib example projects.
- Raylib's official documentation was extensively referenced throughout development.
- Several textures were generated using Google Gemini.
- Parts of the Makefile were written with assistance from an experienced developer.