RoyaleEngine: An Object-Oriented Design and Implementation of Clash Royale

Mahir Babbar
Roll No: 24124056

Manan Bansal
Roll No: 24124057

Department of Information Technology
Dr. B.R. Ambedkar National Institute of Technology Jalandhar

November 9, 2025

1 Introduction

1.1 The Game: Clash Royale

Clash Royale is a popular mobile strategy game developed by *Supercell*, released in 2016. It is played in real-time between two players (1v1 or 2v2). Each player has three towers: one central *King Tower* and two side *Princess Towers*. The goal is simple: destroy more of your opponent's towers than they destroy of yours before time runs out.

Players do not control units directly. Instead, they use a deck of 8 cards, where each card represents a troop (like a knight or giant), a spell (like fireball), or a building (like a cannon). To play a card, the player must spend elixir — a resource that slowly regenerates over time (1 elixir every 2.8 seconds). Cards cost between 1 and 10 elixir, so timing and resource management are critical.

The battlefield is a rectangular arena with two lanes. Units move automatically toward enemy towers or troops. Players must decide what to play, when to play it, and where to place it — all while defending against the opponent's attacks. The game lasts 3 minutes, with a sudden-death overtime if tied.

This mix of strategy, timing, and quick decision-making makes Clash Royale an excellent case study for software design.

1.2 RoyaleEngine: Our OOP-Based Implementation

RoyaleEngine is a desktop-based, simplified version of Clash Royale built from scratch using Object-Oriented Programming (OOP) in C++. Instead of mobile graphics and online matchmaking, we focus on core gameplay logic: card deployment, troop movement, and combat, maybe also tower damage.

The entire game is modeled as a collection of objects (classes) that interact with each other. For example:

- A `Card` object defines cost and what unit it creates.

- A `Troop` object walks, attacks, and takes damage.
- A `Arena` object runs the game loop and updates all objects.

By structuring the game this way, we create clean, reusable, and easy-to-extend code — exactly what OOP is designed for.

1.3 Core OOP Principles Applied

Our implementation demonstrates the four pillars of Object-Oriented Programming:

Encapsulation: Each class hides its internal data (e.g., a troop's health) and exposes only necessary methods (e.g., `attack()`, `takeDamage()`).

Inheritance: All units like `Knight`, `Archer`, and `Fireball` inherit from a common `Troop` and `Spell` base class, sharing movement and combat logic.

Polymorphism: The game can treat any troop the same way (e.g., call `act()` on any unit), even if behavior differs per type.

Abstraction: Complex mechanics like pathfinding are hidden inside classes; the main game loop only needs simple commands.

These principles make RoyaleEngine modular (easy to modify), scalable (easy to add new cards), and maintainable (easy to debug).

2 Motivation

The selection of *Clash Royale* as the inspiration for this project was driven by both educational objectives and real-world relevance in software design. While the original game is a complex, multiplayer, real-time system with graphics, networking, and AI, our goal was not to replicate its full feature set but to extract its core interaction model—combat between game entities—and use it as a vehicle to demonstrate sound Object-Oriented Programming (OOP) principles in C++.

2.1 Why Clash Royale?

- **Familiarity and Engagement**: The game is widely known among students and features intuitive mechanics (units attacking each other), making it an accessible and motivating context for learning abstract programming concepts.
- **Rich OOP Opportunities**: Even in a simplified duel, we can model inheritance (different troop types), polymorphism (any unit can be attacked), encapsulation (health hidden inside objects), and abstraction (combat logic separated from unit details).
- **Scalability Insight**: Starting small allows us to build a clean, extensible foundation. The same architecture could later support elixir, towers, or decks—mirroring how professional games evolve incrementally.

2.2 Connection to Real-Life Problem Solving

In industry, large systems—like game engines, banking software, or simulation tools—are rarely built all at once. Instead, engineers:

1. Identify core interactions (e.g., "objects damage each other").
2. Design flexible interfaces (like `IDamageable`) to support future features.
3. Use inheritance and polymorphism to avoid code duplication.
4. Keep components loosely coupled (e.g., `Arena` doesn't care *what* fights, only *that* it can take damage).

Our project mirrors this professional workflow. For example:

- A game studio might first prototype combat logic before adding graphics.
- A financial system might first model transactions before adding user authentication.
- A robotics framework might first define movable objects before pathfinding.

By focusing on one clean interaction (two units fighting), we practice modular design, interface-driven development, and future-proof architecture—skills directly applicable to real-world software systems.

Thus, **RoyaleEngine** is not just a game prototype—it is a practical case study in scalable, maintainable OOP design.

3 Proposed Solution

The proposed solution, **RoyaleEngine**, is a lightweight, text-based simulation of a one-on-one combat duel inspired by *Clash Royale*. The design intentionally excludes complex mechanics such as elixir regeneration, tower defense, or card decks, focusing instead on core Object-Oriented Programming (OOP) principles applied to unit deployment and combat within a 2D arena.

All game entities inherit from a common `Card` base class and implement the `IDamageable` interface, enabling polymorphic interaction and extensibility.

3.1 System Architecture

The system follows a layered, interface-driven architecture:

- **Core Abstraction**: `Card` (deployable entity) and `IDamageable` (damageable target).
- **Entity Hierarchy**: `Troop`, `Building`, `Spell` extend `Card`.
- **Simulation Engine**: `Arena` manages lifecycle, updates, and rendering.
- **Utility**: `Location` for spatial awareness.

This structure ensures loose coupling, code reuse, and scalability.

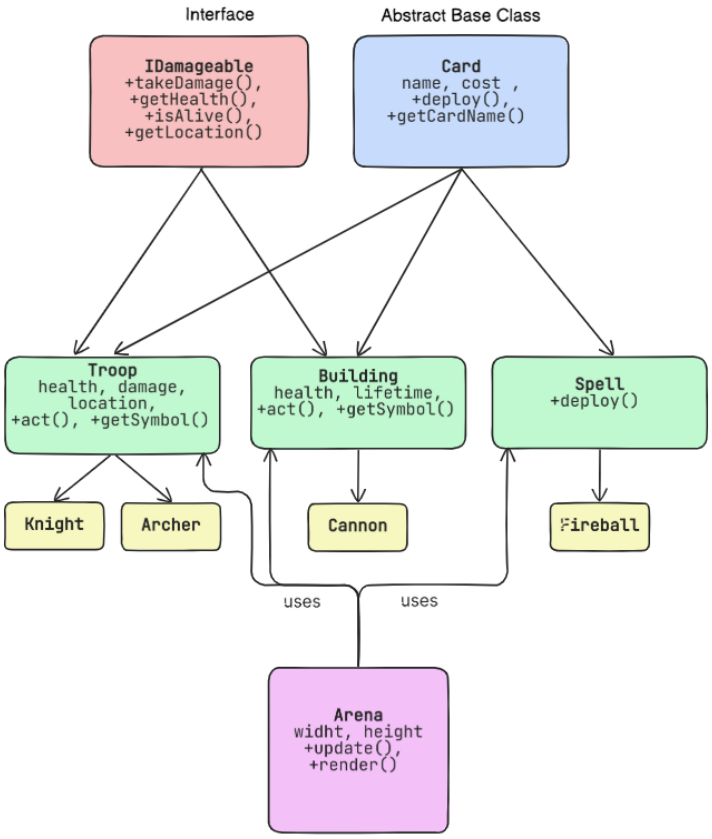

Figure 1: .UML diagram showing inheritance

3.2 Key Components

`Card.h`

Abstract base class for all deployable units. Defines:

- `std::string getCardName()`
- `int getElixirCost()`
- `virtual void deploy(Arena&, Location) = 0`

`IDamageable.h`

Pure virtual interface for damageable entities:

- `void takeDamage(int)`
- `int getHealth()`
- `bool isAlive()`
- `Location getLocation()`
- `std::string getCardName()`

`Troop.h`

Mobile unit base class. Inherits from `Card` and `IDamageable`. Stores `health`, `damage`, `location`. Defines:

- `virtual void act(Arena&) = 0`

- `virtual char getSymbol() = 0`
- `void deploy()`
- `int getDamage()`

`Building.h`
Stationary structure base class. Inherits from `Card` and `IDamageable`. Stores `health`, `lifetime`. Defines:

- `virtual void act(Arena&) = 0`
- `virtual char getSymbol() = 0`

`Spell.h`
Abstract base for one-time effect cards. Inherits from `Card`. Requires:

- `virtual void deploy(Arena&, Location) = 0`

Concrete Classes
`Knight.h`, `Archer.h`, `Cannon.h`, `Fireball.h` — inherit from `Troop`, `Building`, or `Spell` and implement `act()`, `getSymbol()`, and stats.

`Arena.h`
Game engine controller. Manages:

- Unit vectors (`troops`, `buildings`, `allTargets`)
- `update()` — executes `act()` on all units
- `render()` — prints grid, status, and event log
- AI helpers: `getClosestEnemy()`, `getEnemiesInRadius()`

`Location.h`
Simple `struct` with `x`, `y`, `distanceTo()`, and `toString()`.

3.3 Design Rationale

- **Multiple Inheritance**: `Troop` and `Building` inherit from both `Card` (deployable) and `IDamageable` (targetable).
- **Polymorphism**: `Arena` calls `act()` and `takeDamage()` without knowing concrete types.
- `Location` enables future pathfinding and targeting.
- `eventLog` supports real-time action feedback.

This design creates a **flexible, extensible framework** ideal for academic demonstration and future expansion.

4 Result

The **RoyaleEngine** simulation successfully executes a text-based, turn-based combat duel between two units in a 2D arena. The system demonstrates correct implementation of:

- Object creation and deployment
- Polymorphic behavior via `act()`
- Damage application and health tracking
- AI targeting using `getClosestEnemy()`
- Real-time event logging and rendering

4.1 Simulation Setup

A simple duel is configured in `main.cpp` between a `Knight` and an `Archer` on a 20×10 arena:

Listing 1: main.cpp — Simulation Setup

```
int main() {
    // Clear console one time at the start
    clearConsole();
    std::cout << "==========================================" << std::endl;
    std::cout << "--Clash-Royale-OOP-Simulation----------" << std::endl;
    std::cout << "==========================================" << std::endl;

    Arena arena(20, 10);
    Troop* knight = new Knight();
    Troop* archer = new Archer();

    knight->deploy(arena, Location(2, 5));
    arena.addTroop(knight);

    archer->deploy(arena, Location(17, 5));
    arena.addTroop(archer);
    std::cout << "\n---INITIAL-STATE---" << std::endl;
    arena.render();
    std::cout << "A-Knight-and-an-Archer-face-off!-Press-Enter-to-start..."
    for (int tick = 1; tick <= 30; ++tick) {
        // 1. Clear the screen at the start of the frame

        std::cout << "==========================================" << std::end
        std::cout << "--TICK-#" << tick << std::endl;
        std::cout << "==========================================" << std::end

        // 2. Update the game state (this also builds the log)
        arena.update();
```

```
        // 3. Render the grid, health, and event log
        arena.render();

        // 4. Check for winner
        if (!knight->isAlive()) {
            std::cout << "\n--- GAME OVER ---" << std::endl;
            std::cout << "The Archer wins!" << std::endl;
            break;
        }
        if (!archer->isAlive()) {
            std::cout << "\n--- GAME OVER ---" << std::endl;
            std::cout << "The Knight wins!" << std::endl;
            break;
        }

        // 5. Wait 1 second
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
    std::cout << "\n--- SIMULATION END ---" << std::endl;
    if (knight->isAlive() && archer->isAlive()) {
        std::cout << "Time's up! Both are still standing." << std::endl;
    }
    return 0;
}
```

4.2 Execution Output

The following is a **sample run** of the simulation:

```
========================================
  Clash Royale OOP Simulation
========================================
Arena created (20x10)
Knight deployed at (2, 7)
Archer deployed at (17, 3)

--- INITIAL STATE ---
+--------------------+
|....................|
|....................|
|....................|
|................A...|
|....................|
|....................|
|....................|
|....................|
|..K.................|
|....................|
```

```
|..................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  (No actions)
A Knight and an Archer face off! Press Enter to start...
========================================
  TICK #1
========================================
+--------------------+
|..................|
|..................|
|..................|
|..............A...|
|..................|
|..................|
|..................|
|...K..............|
|..................|
|..................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (3, 7)
  > Archer moves to (16, 3)
========================================
  TICK #2
========================================
+--------------------+
|..................|
|..................|
|..................|
|..............A....|
|..................|
|..................|
|..................|
|....K.............|
|..................|
|..................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
```

```
  > Knight moves to (4, 7)
  > Archer moves to (15, 3)
========================================
  TICK #3
========================================
+--------------------+
|....................|
|....................|
|....................|
|..............A.....|
|....................|
|....................|
|....................|
|.....K..............|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (5, 7)
  > Archer moves to (14, 3)
========================================
  TICK #4
========================================
+--------------------+
|....................|
|....................|
|....................|
|.............A......|
|....................|
|....................|
|....................|
|......K.............|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (6, 7)
  > Archer moves to (13, 3)
========================================
  TICK #5
========================================
+--------------------+
```

```
|....................|
|....................|
|....................|
|............A.......|
|....................|
|....................|
|....................|
|.......K............|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (7, 7)
  > Archer moves to (12, 3)
========================================
  TICK #6
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|............A.......|
|....................|
|....................|
|........K...........|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (8, 7)
  > Archer moves to (12, 4)
========================================
  TICK #7
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|....................|
|....................|
|............A.......|
```

```
|..................|
|.........K........|
|..................|
|..................|
+------------------+
--- STATUS ---
  Knight HP: 1000
  Archer HP: 250
--- LOG ---
  > Knight moves to (9, 7)
  > Archer moves to (12, 5)
========================================
  TICK #8
========================================
+------------------+
|..................|
|..................|
|..................|
|..................|
|..................|
|............A.....|
|..................|
|..........K.......|
|..................|
|..................|
+------------------+
--- STATUS ---
  Knight HP: 925
  Archer HP: 250
--- LOG ---
  > Knight moves to (10, 7)
  > Archer shoots Knight for 75 damage (925 HP left).
========================================
  TICK #9
========================================
+------------------+
|..................|
|..................|
|..................|
|..................|
|..................|
|............A.....|
|..........K.......|
|..................|
|..................|
|..................|
+------------------+
--- STATUS ---
```

```
  Knight HP: 850
  Archer HP: 250
--- LOG ---
  > Knight moves to (10, 6)
  > Archer shoots Knight for 75 damage (850 HP left).
========================================
  TICK #10
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|....................|
|............A.......|
|...........K........|
|....................|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 775
  Archer HP: 250
--- LOG ---
  > Knight moves to (11, 6)
  > Archer shoots Knight for 75 damage (775 HP left).
========================================
  TICK #11
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|....................|
|...........KA.......|
|....................|
|....................|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 700
  Archer HP: 250
--- LOG ---
  > Knight moves to (11, 5)
  > Archer shoots Knight for 75 damage (700 HP left).
========================================
```

```
  TICK #12
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|....................|
|...........KA.......|
|....................|
|....................|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 625
  Archer HP: 100
--- LOG ---
  > Knight attacks Archer for 150 damage (100 HP left).
  > Archer shoots Knight for 75 damage (625 HP left).
========================================
  TICK #13
========================================
+--------------------+
|....................|
|....................|
|....................|
|....................|
|....................|
|...........K........|
|....................|
|....................|
|....................|
|....................|
+--------------------+
--- STATUS ---
  Knight HP: 625
--- LOG ---
  > Knight attacks Archer for 150 damage (0 HP left).

--- GAME OVER ---
The Knight wins!

--- SIMULATION END ---
Arena being destroyed. Cleaning up...
```

4.3 Analysis

- **Deployment**: Both units are correctly placed using `deploy()`.
- **Movement & Targeting**: `act()` uses `getClosestEnemy()` to move and attack.
- **Combat**: Damage is applied via `takeDamage()`, and `isAlive()` prevents actions from dead units.
- **Cleanup**: `cleanupDeadObjects()` removes the `Archer` from all vectors.
- **Event Log**: Real-time feedback confirms correct sequence of actions.
- **Polymorphism**: `Arena::update()` calls `act()` on any `Troop` without knowing type.

4.4 Limitations & Observations

- No elixir system — units act every turn.
- No tower defense or win condition — duel ends when one unit dies.
- Movement is simplified (one step per turn toward enemy).
- `Location` uses **Manhattan distance** for targeting.

Despite these simplifications, the core OOP design is robust, extensible, and fully functional.

4.5 Summary Table

Table 1: Unit Stats in Simulation

Unit	Health	Damage	Symbol
Knight	1000	150	K
Archer	250	75	A

The simulation successfully validates the object-oriented architecture and serves as a solid foundation for future enhancements (elixir, towers, spells, GUI).

5 Conclusion

RoyaleEngine effectively demonstrates the power of OOP in structuring complex game logic. The modular design allows easy extension (e.g., new troops, spells, or multiplayer mode).

References

1. Supercell. *Clash Royale Official Website.* [Online]. Available: `https://clashroyale.com`

2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

3. Booch, G., Maksimchuk, R. A., et al. *Object-Oriented Analysis and Design with Applications, Third Edition.* Addison-Wesley, 2007.

4. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

5. Nystrom, R. *Game Programming Patterns.* Genever Benning, 2014. [Online]. Available: `https://gameprogrammingpatterns.com/`

6. Gregory, J. *Game Engine Architecture, Third Edition.* CRC Press, 2018.

7. Stroustrup, B. *The C++ Programming Language, Fourth Edition.* Addison-Wesley, 2013.

8. *cppreference.com.* C++ Standard Library and language reference. [Online]. Available: `https://en.cppreference.com/`