

Clash Royale

by

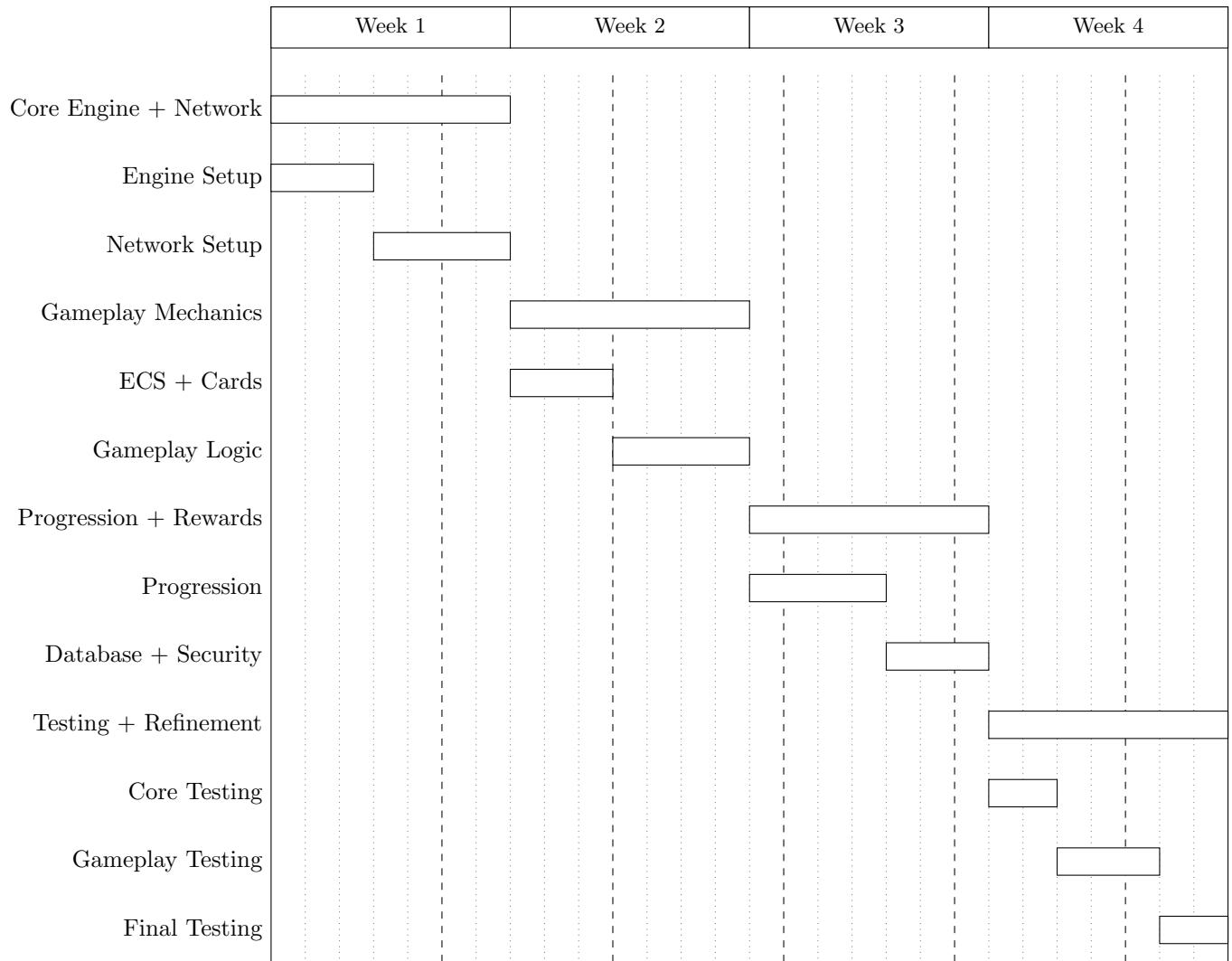
Paul Hodges

March 12, 2025

Contents

1	Development Schedule	2
2	Problem Outline	2
3	Problem Description	3
3.1	Game Overview	3
3.2	Game Objectives	3
3.3	Core Game Rules	3
3.4	Gameplay Flow	3
3.5	Scoring System	4
3.6	Winner Determination	4
3.7	Advanced Game Mechanics and Strategies	4
4	Requirements	4
4.1	Functional Requirements	4
4.2	Non-Functional Requirements	6
5	Design	6
5.1	Core System Design	6
5.1.1	Engine	6
5.1.2	Network	7
5.1.3	2D with 3D Illusion	8
5.2	Algorithm Design	8
5.2.1	Server-Side Game Management	8
5.2.2	Card Deployment Process	9
5.2.3	Battle Update Logic	10
5.2.4	Target Selection Algorithm	11
5.2.5	Chest Reward System	11

1 Development Schedule



2 Problem Outline

The purpose of this project is to develop a simplified but functional online version of Clash Royale using Python. This real-time strategy game implementation will focus on core gameplay mechanics and network functionality, demonstrating practical software development skills in a gaming context.

The game will allow players to:

- Engage in real-time, online strategic battles against other players
- Build and customize card decks from their collection
- Progress through a skill-based arena system
- Earn rewards through a chest system to expand their card collection
- Experience the core strategic elements that make Clash Royale engaging

This implementation serves as both an educational tool for understanding game development principles and a demonstration of networking, database management, and software architecture skills in a cohesive application.

3 Problem Description

3.1 Game Overview

Clash Royale is a real-time strategy game where players deploy cards representing troops, spells, and buildings onto a battlefield to defeat their opponent. The game combines elements of collectible card games, tower defense, and multiplayer online battle arenas (MOBAs) into a unique gaming experience.

3.2 Game Objectives

The primary objective is to destroy more of the opponent's towers than they destroy of yours within the allotted battle time (typically 3 minutes, with a 2 minute overtime if tied). The game features:

- Three towers per player: two Princess Towers and one King Tower
- Destroying a Princess Tower grants one crown
- Destroying the King Tower grants three crowns and ends the match immediately
- The player with more crowns at the end of the time limit wins

3.3 Core Game Rules

- Each player begins with four cards from their eight-card deck
- Cards are deployed using elixir, a regenerating resource
- Elixir regenerates at a rate of 1 unit every 2.8 seconds (doubles in the first minute of overtime and then triples in the second minute)
- Maximum elixir capacity is 10 units
- Card costs range from 1 to 8 elixir
- After playing a card, it's replaced by another from the player's deck
- Cards can only be deployed on the player's side of the arena (with exceptions for spells and certain cards)
- Units automatically move toward and attack enemy units and towers based on their targeting preferences

3.4 Gameplay Flow

1. Players join a match through the matchmaking system
2. Both players start with the same amount of elixir
3. Players strategically deploy cards to attack opponent towers while defending their own
4. Units and buildings have specific health, damage, speed, and targeting parameters
5. Spells cause immediate effects (damage, slow, etc.) in their area of effect
6. Destroyed units are removed from the battlefield
7. The battle continues until a King Tower falls or time expires

3.5 Scoring System

- Destroying a Princess Tower: 1 crown
- Destroying the King Tower: 3 crowns (automatic victory)
- Most crowns at time expiration: Victory
- Equal crowns at time expiration: Draw
- Victory awards trophies that contribute to arena progression
- Defeat results in trophy loss

3.6 Winner Determination

- The player who destroys the opponent's King Tower instantly wins
- If no King Tower is destroyed within the time limit, the player with more crowns wins
- If both players have the same number of crowns when time expires, the match enters overtime
- In overtime, the first player to gain a crown advantage wins
- If neither player gains an advantage during overtime, the match ends in a draw

3.7 Advanced Game Mechanics and Strategies

- **Elixir Management:** Efficient use of elixir is critical; overspending creates vulnerability
- **Counter Deployment:** Specific cards counter others effectively (e.g., air units counter ground-only attackers)
- **Card Synergies:** Certain combinations of cards create powerful offensive or defensive capabilities
- **Lane Pressure:** Applying pressure in one lane to force defensive card usage, then attacking the other
- **Elixir Counting:** Tracking opponent's elixir to identify advantageous attack opportunities
- **Card Cycle Management:** Cycling through cards quickly to reach key cards in your deck
- **Tower Trading:** Strategic sacrifice of a tower to gain elixir advantage for a stronger counter-push
- **Spell Value:** Using spells to eliminate multiple units simultaneously for positive elixir trades
- **Unit Placement:** Precise placement affects unit interactions and pathing
- **Timing:** Deploying cards at optimal moments to maximize their effectiveness

4 Requirements

4.1 Functional Requirements

1. FR1: Server-Client Architecture

- The system shall implement a client-server model supporting real-time multiplayer
- The server shall handle game state synchronization across clients
- The server shall manage concurrent games with minimal latency
- The system shall handle connection drops and reconnections gracefully

2. FR2: Card Management System

- Players shall be able to view their card collection
- Players shall be able to build decks with exactly 8 cards

- The system shall validate deck composition rules
- The system shall implement card rarity and level mechanics
- Players shall be able to upgrade cards when meeting requirements

3. **FR3: Real-time Battle Simulation**

- The system shall manage battles in real-time
- The system shall implement elixir generation mechanics
- The system shall handle unit deployment, movement, and targeting
- The system shall calculate damage, health, and tower destruction
- The system shall implement the 3-minute standard time and overtime rules

4. **FR4: Progression System**

- The system shall track player trophies
- The system shall implement arena progression based on trophy count
- The system shall match players based on similar trophy counts
- Different arenas shall unlock different cards

5. **FR5: Reward System**

- The system shall reward players with chests after victories
- Chests shall contain cards and gold based on arena level
- The system shall implement various chest types with different rewards
- The system shall track chest unlocking timers

6. **FR6: Clan System**

- Players shall be able to create and join clans
- Clans shall support basic member management
- The system shall implement a clan chat feature
- The system shall display clan member activity

7. **FR7: User Authentication**

- The system shall securely authenticate users
- The system shall protect user credentials
- The system shall manage login sessions with tokens
- The system shall enforce password security requirements

8. **FR8: Database Integration**

- The system shall persist all player data
- The system shall record battle history
- The system shall implement transaction safety for critical operations
- The system shall backup data regularly

4.2 Non-Functional Requirements

1. NFR1: Performance

- The system shall handle at least 100 concurrent connections
- Battle simulations shall update at least 10 times per second
- Server response time shall not exceed 200ms under normal load
- Client frame rate shall maintain at least 30 FPS on target devices

2. NFR2: Reliability

- The system shall have 99% uptime
- The system shall recover from crashes without data loss
- The system shall handle network instability gracefully
- The system shall implement data integrity checks

3. NFR3: Scalability

- The architecture shall support horizontal scaling
- The database shall handle growing player data efficiently
- The matchmaking algorithm shall scale with player population

4. NFR4: Maintainability

- The code shall follow PEP 8 style guidelines
- The system shall use modular design patterns
- The codebase shall include comprehensive documentation
- The system shall implement logging for debugging

5. NFR5: Security

- User passwords shall be stored using strong hashing
- Communication shall be encrypted
- The system shall protect against common attack vectors
- The system shall validate all client input

5 Design

5.1 Core System Design

The game will be built using a custom Entity Component System (ECS) engine and a custom network implementation. The visual style will be 2D with a simulated 3D look achieved through layered sprites and shadow rendering.

5.1.1 Engine

The engine is written in Python, based on the design off of one of my previous Rust engines, "Oxidized." Key features include:

5.1.1.1 ECS Architecture

The Entity Component System (ECS) architecture separates data (components) from logic (systems). Entities are simple IDs, and components are data containers (e.g., Position, Velocity, Sprite). Systems operate on entities that have the required components. This design promotes data-oriented programming, improving cache efficiency and making the code more modular and maintainable. In this Python implementation, we'll leverage dictionaries and lists to store and manage entities, components, and systems. For example, a "MovementSystem" would iterate through entities with "Position" and "Velocity" components, updating their positions based on their velocities. This decoupling of data and logic allows for flexible entity composition and avoids the complexities of traditional inheritance-based object-oriented programming.

5.1.1.2 2D/3D Cameras

The engine will support both orthographic and perspective cameras. Orthographic cameras provide a parallel projection, useful for classic 2D views where depth isn't emphasized. Perspective cameras simulate real-world vision, where objects appear smaller as they recede into the distance. This is crucial for creating the simulated 3D look. The camera implementation will likely involve a transformation matrix that converts world coordinates to screen coordinates. For perspective cameras, this matrix will include a perspective projection component, while orthographic cameras will use a simple scaling and translation matrix. We will have functions to easily switch between camera types, and to adjust the field of view of the perspective camera.

5.1.1.3 GameObjects and Components

GameObjects serve as containers for components. By assigning different combinations of components to a GameObject, we can create a wide variety of entities with unique behaviors. For example, a "Player" GameObject might have "Position," "Velocity," "Sprite," "Input," and "Health" components. A "Tree" GameObject might have "Position" and "Sprite" components. This modularity allows for easy creation and modification of game entities without modifying core engine code. The component data will be stored in data structures that are easily accessible by the systems.

5.1.1.4 Layered Sprite Rendering

To create the illusion of depth in a 2D environment, we'll use layered sprites. Sprites representing objects closer to the "camera" will be rendered on top of sprites representing objects further away. We'll implement a sorting mechanism based on the Y-coordinate (or a custom depth value) of the entities to determine the rendering order. This will allow for overlapping sprites to create a sense of depth. Furthermore, we will have the capability to assign a layer number to each sprite, allowing for more manual control over the draw order, and enabling the creation of backgrounds, midgrounds, and foregrounds.

5.1.1.5 Rendering

The rendering for the python engine are not done using conventional methods like "pygame". I use a custom library that i created myself which uses SDL2 to draw to the screen and provides more complex methods for other things like textures, images and keyboard input. I create the python bindings (for the cpython module) using pybin11. The library is compiled, the binds are compiled and then the library is linked with the binds to produce a cpython library. The stub file for the LSP is then generated using stubgen, which is shipped with python.

5.1.2 Network

The network uses custom socket communication with binary data transmission:

5.1.2.1 Custom Binary Protocol

Instead of relying on human-readable text-based protocols, we'll use a custom binary protocol. This protocol will define the structure of packets sent between the client and server, specifying the types and sizes of data fields. Binary protocols are more compact and efficient, reducing network overhead and latency. We will use python's struct module to pack and unpack data into binary format. We will define packet types, and each packet type will have a defined data structure.

5.1.2.2 Client-Server Architecture

We'll implement a client-server architecture, where the server acts as the authoritative source of game state. Clients send input to the server, and the server processes the input, updates the game state, and sends updates back to the clients. This architecture helps prevent cheating and ensures consistency across all clients. The server will maintain the master copy of the game world, and the clients will maintain local copies that are kept in sync with the server.

5.1.2.3 Asynchronous Sockets

To avoid blocking the main game loop, we'll use asynchronous sockets. Asynchronous sockets allow the game to continue running while waiting for network data. We will use python's `asyncio` library or similar, to handle the asynchronous network operations. This will prevent the game from freezing or becoming unresponsive during network communication.

5.1.2.4 Packet Listener and Broadcaster

The server will have a packet listener to receive incoming packets from clients and a packet broadcaster to send packets to all connected clients. The listener will parse incoming packets and dispatch them to the appropriate game logic. The broadcaster will efficiently distribute game state updates to all clients, minimizing network traffic. We will implement packet queuing and throttling to prevent network congestion. The broadcaster will only send data that has changed, to reduce the amount of data sent.

5.1.3 2D with 3D Illusion

The 3D look is achieved through:

5.1.3.1 Layered Sprites

As mentioned earlier, layered sprites are crucial for creating the illusion of depth. By rendering sprites in a specific order based on their perceived depth, we can simulate the effect of objects being in front of or behind each other. This will involve sorting sprites by their Y-coordinate or a custom depth value before rendering. We will use a depth buffer, or an equivalent sort algorithm, to ensure that sprites are drawn in the correct order.

5.1.3.2 Shadow Rendering

Adding shadows to objects can significantly enhance depth perception. We'll implement shadow rendering by projecting a simplified silhouette of each object onto the ground. The darkness and length of the shadow will vary based on the object's height and the light source's position. This will give objects a sense of grounding and add visual depth to the scene. We will use sprite masks to create the shadow silhouettes.

5.1.3.3 Perspective Scaling

To further enhance the 3D illusion, we'll implement perspective scaling. Objects farther away from the "camera" will appear smaller, while objects closer will appear larger. This will be achieved by adjusting the sprite's scale based on its distance from the camera. We will use a perspective projection matrix to calculate the scale factor. This will be done in conjunction with the Layered Sprite rendering, to create a believable 3D effect.

5.2 Algorithm Design

Below are the core algorithms for the main game systems. These algorithms use proper control structures and follow logic flow to handle the key game mechanics.

5.2.1 Server-Side Game Management

```
1 FUNCTION RunServer():
2     INITIALIZE server
3     INITIALIZE player_pool = []
4     INITIALIZE active_games = []
5
6     WHILE server_running:
7         FOR each incoming_connection:
8             IF is_new_connection THEN
9                 player = CreatePlayer(connection)
10                player_pool.APPEND(player)
11                SEND_WELCOME_MESSAGE(connection)
12            ELSE IF is_matchmaking_request THEN
13                player = GetPlayerFromConnection(connection)
```

```

14         MatchmakingQueue.ADD(player)
15     ELSE IF is_game_action THEN
16         game = GetGameFromPlayer(connection)
17         ProcessGameAction(game, connection, action)
18     END IF
19 END FOR
20
21 # Matchmaking
22 WHILE MatchmakingQueue.COUNT >= 2:
23     player1 = MatchmakingQueue.GET_NEXT()
24     player2 = MatchmakingQueue.FIND_OPPONENT(player1)
25     IF player2 != NULL THEN
26         game = CreateGame(player1, player2)
27         active_games.APPEND(game)
28         MatchmakingQueue.REMOVE(player2)
29     ELSE
30         MatchmakingQueue.RETURN(player1)
31         BREAK
32     END IF
33 END WHILE
34
35 # Update all active games
36 FOR each game IN active_games:
37     UpdateGame(game)
38     IF game.is_finished THEN
39         ProcessGameResults(game)
40         active_games.REMOVE(game)
41     END IF
42 END FOR
43 END WHILE
44 END FUNCTION

```

Listing 1: Main Server Loop

5.2.2 Card Deployment Process

```

1 FUNCTION ProcessCardDeployment(game, player, card_id, position_x, position_y):
2     # Validate the deployment
3     IF NOT IsValidDeployPosition(game, player, position_x, position_y) THEN
4         SEND_ERROR(player, "Invalid deployment position")
5         RETURN False
6     END IF
7
8     # Get the card from player's hand
9     card = player.GetCardFromHand(card_id)
10    IF card == NULL THEN
11        SEND_ERROR(player, "Card not in hand")
12        RETURN False
13    END IF
14
15    # Check elixir cost
16    IF player.current_elixir < card.elixir_cost THEN
17        SEND_ERROR(player, "Insufficient elixir")
18        RETURN False
19    END IF
20
21    # Deploy the card
22    player.current_elixir = player.current_elixir - card.elixir_cost
23    player.RemoveCardFromHand(card_id)
24    player.AddNextCardToHand()
25
26    # Create the unit/spell/building on the field
27    IF card.type == "TROOP" THEN
28        unit = CreateTroop(card, position_x, position_y, player.side)
29        game.AddUnit(unit)
30    ELSE IF card.type == "SPELL" THEN
31        ApplySpellEffect(game, card, position_x, position_y, player.side)
32    ELSE IF card.type == "BUILDING" THEN
33        building = CreateBuilding(card, position_x, position_y, player.side)
34        game.AddBuilding(building)
35    END IF
36

```

```

37 # Notify both players
38 BROADCAST_DEPLOYMENT(game, player.id, card_id, position_x, position_y)
39
40 RETURN True
41 END FUNCTION

```

Listing 2: Card Deployment

5.2.3 Battle Update Logic

```

1 FUNCTION UpdateGame(game):
2     # Update game time
3     game.current_time = game.current_time + TICK_RATE
4
5     # Check for game end conditions
6     IF game.current_time >= game.max_time AND NOT game.in_overtime THEN
7         IF game.player1.crowns == game.player2.crowns THEN
8             game.in_overtime = True
9             game.overtime_end = game.current_time + OVERTIME_DURATION
10            BROADCAST_OVERTIME_START(game)
11        ELSE
12            game.is_finished = True
13            RETURN
14        END IF
15    END IF
16
17    IF game.in_overtime AND game.current_time >= game.overtime_end THEN
18        game.is_finished = True
19        RETURN
20    END IF
21
22    # Regenerate elixir for both players
23    regeneration_amount = CALCULATE_ELIXIR_REGEN(game)
24    game.player1.current_elixir = MIN(game.player1.current_elixir + regeneration_amount,
25    MAX_ELIXIR)
26    game.player2.current_elixir = MIN(game.player2.current_elixir + regeneration_amount,
27    MAX_ELIXIR)
28
29    # Update all units on the field
30    FOR each unit IN game.units:
31        IF unit.health <= 0 THEN
32            game.RemoveUnit(unit)
33            CONTINUE
34        END IF
35
36        # Find target for the unit
37        IF unit.current_target == NULL OR unit.current_target.health <= 0 THEN
38            unit.current_target = FindTarget(game, unit)
39        END IF
40
41        # Update unit position and attack
42        IF unit.current_target != NULL THEN
43            IF IsInRange(unit, unit.current_target) THEN
44                PerformAttack(unit, unit.current_target)
45            ELSE
46                MoveTowards(unit, unit.current_target)
47            END IF
48        END IF
49    END FOR
50
51    # Update all buildings
52    FOR each building IN game.buildings:
53        IF building.health <= 0 OR building.lifetime <= 0 THEN
54            game.RemoveBuilding(building)
55            CONTINUE
56        END IF
57
58        building.lifetime = building.lifetime - TICK_RATE
59
60        # Find target for the building
61        IF building.can_attack THEN

```

```

60         IF building.current_target == NULL OR building.current_target.health <= 0
        THEN
61             building.current_target = FindTarget(game, building)
62         END IF
63
64         # Attack if target in range
65         IF building.current_target != NULL AND IsInRange(building, building.
current_target) THEN
66             PerformAttack(building, building.current_target)
67         END IF
68     END IF
69 END FOR
70
71 # Check tower status
72 CheckTowerStatus(game)
73
74 # Send updated game state to both players
75 BROADCAST_GAME_STATE(game)
76 END FUNCTION

```

Listing 3: Battle Update Logic

5.2.4 Target Selection Algorithm

```

1 FUNCTION FindTarget(game, attacker):
2     closest_target = NULL
3     min_distance = INFINITY
4
5     # Define potential targets based on attacker type
6     potential_targets = []
7
8     # Filter targets based on attacker's capabilities
9     IF attacker.attacks_ground AND attacker.attacks_air THEN
10         potential_targets = game.GetAllUnitsAndBuildings(GetOpponentSide(attacker.side))
11     ELSE IF attacker.attacks_ground THEN
12         potential_targets = game.GetGroundUnitsAndBuildings(GetOpponentSide(attacker.
side))
13     ELSE IF attacker.attacks_air THEN
14         potential_targets = game.GetAirUnits(GetOpponentSide(attacker.side))
15     END IF
16
17     # Add towers to potential targets
18     towers = game.GetTowers(GetOpponentSide(attacker.side))
19     potential_targets.EXTEND(towers)
20
21     # Apply targeting preference
22     IF attacker.targeting_preference == "BUILDINGS" THEN
23         buildings = [target FOR target IN potential_targets IF target.type == "BUILDING"
OR target.type == "TOWER"]
24         IF buildings.LENGTH > 0 THEN
25             potential_targets = buildings
26         END IF
27     END IF
28
29     # Find closest target
30     FOR each target IN potential_targets:
31         distance = CalculateDistance(attacker.position, target.position)
32         IF distance < min_distance THEN
33             min_distance = distance
34             closest_target = target
35         END IF
36     END FOR
37
38     RETURN closest_target
39 END FUNCTION

```

Listing 4: Target Finding

5.2.5 Chest Reward System

```

1 FUNCTION GenerateChestForPlayer(player, chest_type):
2     # Create a new chest with appropriate type
3     chest = CreateChest(chest_type)
4
5     # Set unlock time based on chest type
6     IF chest_type == "SILVER" THEN
7         chest.unlock_time = 3 * HOURS
8     ELSE IF chest_type == "GOLD" THEN
9         chest.unlock_time = 8 * HOURS
10    ELSE IF chest_type == "MAGICAL" THEN
11        chest.unlock_time = 12 * HOURS
12    ELSE IF chest_type == "GIANT" THEN
13        chest.unlock_time = 16 * HOURS
14    END IF
15
16    # Determine gold amount based on arena and chest type
17    chest.gold = CALCULATE_GOLD_REWARD(player.arena, chest_type)
18
19    # Determine card counts based on arena and chest type
20    common_count, rare_count, epic_count, legendary_count = CALCULATE_CARD_COUNTS(player
21    .arena, chest_type)
22
23    # Generate specific cards
24    available_cards = GetAvailableCardsForArena(player.arena)
25
26    FOR i = 1 TO common_count:
27        card = SelectRandomCard(available_cards, "COMMON")
28        chest.AddCard(card)
29    END FOR
30
31    FOR i = 1 TO rare_count:
32        card = SelectRandomCard(available_cards, "RARE")
33        chest.AddCard(card)
34    END FOR
35
36    FOR i = 1 TO epic_count:
37        card = SelectRandomCard(available_cards, "EPIC")
38        chest.AddCard(card)
39    END FOR
40
41    FOR i = 1 TO legendary_count:
42        card = SelectRandomCard(available_cards, "LEGENDARY")
43        chest.AddCard(card)
44    END FOR
45
46    # Add chest to player's inventory
47    player.AddChest(chest)
48
49    # Notify player
50    SEND_CHEST_RECEIVED(player, chest)
51
52    RETURN chest
53 END FUNCTION

```

Listing 5: Chest Generation