Real-Time Strategy 3D Game: Mathematical and Computational Foundations

Abstract This paper provides a comprehensive overview of the core mathematical models and computational frameworks essential for developing modern Real-Time Strategy (RTS) 3D games. It details the theoretical underpinnings and practical implementation of three critical systems: pathfinding, AI-driven decision-making, and combat resolution. We present a formal analysis of the A* algorithm for navigation, explore Markov Decision Processes (MDPs) and Q-learning for intelligent agent behavior, and model combat dynamics using both deterministic and stochastic approaches, including Negative Binomial distributions and Absorbing Markov Chains. The paper provides concrete implementation guidance for the Unity engine, including C# architecture and performance optimization with the C# Job System. Finally, we propose a methodology for conducting reproducible experiments to validate and tune these systems, ensuring robust and engaging gameplay.

1. Introduction

Real-Time Strategy (RTS) games represent one of the most computationally demanding genres in interactive entertainment. Unlike turn-based games, RTS titles require the continuous, simultaneous simulation of hundreds or even thousands of autonomous agents, each navigating complex environments, making tactical decisions, and engaging in combat—all under strict real-time constraints where a frame must be rendered every few milliseconds. The development of a modern 3D RTS game is therefore not merely an exercise in software engineering, but a deep, multidisciplinary challenge at the intersection of applied mathematics, computer science, and artificial intelligence.

The success of an RTS game hinges on the seamless and believable operation of its core systems. A unit that gets stuck on a wall, an AI opponent that makes nonsensical decisions, or a battle that feels arbitrary and unpredictable can quickly shatter the player's immersion and sense of strategic agency. Consequently, the foundations of these systems must be both mathematically sound and computationally efficient. This paper is written for a dual audience: for the applied mathematician, it demonstrates a compelling, high-impact domain for modeling complex systems; for the Unity developer, it provides the theoretical rigor needed to move beyond ad-hoc solutions and build robust, scalable, and extensible gameplay mechanics.

We structure our exploration around the three pillars of RTS gameplay:

1. **Pathfinding and Navigation:** How do units efficiently find and traverse optimal paths through a complex 3D world? We will analyze the A* algorithm, the industry-standard solution, from a formal perspective (Local Resource: Pathfinding in Real-Time Strategy Games).
2. **AI for Decision-Making:** How does an agent decide what to do, where to go, and which enemy to engage? We will model this problem using frameworks of increasing sophistication, from simple utility theory to the powerful formalisms of Markov Decision Processes and reinforcement learning (Local Resource: Mathematical Models and Algorithmic Frameworks for Decision-Making in RTS).

3. **Combat Modeling and Resolution:** How can we predict and simulate the outcome of a conflict between two opposing forces? We will investigate both probabilistic models for quick assessment and stochastic simulation techniques for high-fidelity resolution (Local Resource: Combat dynamics models in RTS).

For each of these pillars, we will first derive the mathematical principles and then discuss their practical implementation within the Unity game engine, with a focus on writing high-performance C# code. Finally, we establish a framework for empirical validation, emphasizing that in game development, theoretical elegance must be accompanied by experimental proof of efficacy and performance.

2. Pathfinding and Navigation in a 3D World

Effective navigation is the bedrock of any RTS experience. Without the ability for units to move intelligently and reliably from one point to another, all higher-level strategic concepts become moot. The core challenge is to find an optimal path—typically the shortest—from a starting point to a destination while avoiding obstacles, a task that must be performed for hundreds of units simultaneously. While a 3D game world appears continuous to the player, pathfinding algorithms almost universally operate on a discrete abstraction of that world (Local Resource: Pathfinding in Real-Time Strategy Games).

2.1. World Discretization

The first step in implementing pathfinding is to impose a discrete structure onto the continuous game world. The most common approach is the **uniform grid**, where the environment is partitioned into a set of regular, equal-sized cells (tiles). A cell can be marked as either "walkable" or "unwalkable." This abstraction transforms the pathfinding problem from an intractable continuous search into a finite graph traversal problem, where cells are the graph's vertices and movements between adjacent cells are its edges.

Other representations exist, such as **navigation meshes (NavMeshes)**, which use a collection of convex polygons to represent walkable areas. NavMeshes are often more efficient for representing large, open spaces but add complexity to the algorithm. For the sake of formal analysis and ease of implementation, we will focus on the grid-based approach.

*2.2. The A Search Algorithm**

The A* (pronounced "A-star") algorithm is the canonical solution for pathfinding in games and robotics. It is a best-first search algorithm that efficiently finds the least-cost path from a start node to a goal node in a graph. A* achieves its efficiency by intelligently prioritizing which nodes to explore next, based on an estimate of how promising they are. This is formalized through its core evaluation function for any given node n:

$$f(n) = g(n) + h(n)$$

Where:

- ***g(n)*** is the exact cost of the path from the starting node to node *n*. This value is known and tracked as the algorithm progresses.
- ***h(n)*** is the **heuristic**, an *estimated* cost of the cheapest path from node *n* to the goal node. The quality of this estimate is critical to A*'s performance.
- ***f(n)*** is the resulting estimated cost of the cheapest solution that passes through node *n*.

The algorithm maintains two lists: an **OPEN list**, which is a priority queue of discovered nodes that have not yet been fully evaluated, and a **CLOSED list**, which contains nodes that have already been evaluated. The algorithm proceeds as follows [1]:

1. Initialize the OPEN list with the starting node. Set its *g(n)* value to 0 and calculate its *h(n)*.
2. While the OPEN list is not empty: a. Dequeue the node *n* with the lowest *f(n)* value from the OPEN list. b. If *n* is the goal node, the search is complete. The path can be reconstructed by backtracking from *n* via the parent pointers stored at each node. c. Move *n* to the CLOSED list. d. For each neighbor *n'* of *n*: i. If *n'* is in the CLOSED list, ignore it. ii. Calculate a tentative *g(n')* value as *g(n) + cost(n, n')*. iii. If *n'* is not in the OPEN list, add it. Record *n* as its parent and store its *g* and *h* values. iv. If *n'* is already in the OPEN list, check if this new path is better by comparing the new tentative *g(n')* with the existing one. If it is, update its parent and *g* value.

A* is guaranteed to find the shortest path if its heuristic function, *h(n)*, is **admissible—**that is, if it never overestimates the true cost to reach the goal. This property is crucial for predictable unit behavior (Local Resource: Pathfinding in Real-Time Strategy Games).

2.3. Heuristic Functions

The choice of heuristic is a trade-off between accuracy and speed. A more accurate heuristic reduces the number of nodes A* must explore, but the heuristic itself must be computationally cheap. For a grid where movement is allowed in 8 directions (horizontally, vertically, and diagonally), common heuristics include:

- **Manhattan Distance:** `h(n) = D * (abs(n.x - goal.x) + abs(n.y - goal.y))`. This is admissible when diagonal movement is disallowed or has the same cost as axial movement. *D* is the cost of moving between adjacent squares.
- **Euclidean Distance:** `h(n) = D * sqrt((n.x - goal.x)^2 + (n.y - goal.y)^2)`. This is the true straight-line distance and is therefore always admissible. However, the square root calculation can be computationally expensive.
- **Diagonal (Chebyshev) Distance:** `h(n) = D * max(abs(n.x - goal.x), abs(n.y - goal.y))`. This is admissible when diagonal movement costs the same as axial movement.

In practice, a common and effective choice is the **Octile distance**, which correctly accounts for different costs for axial and diagonal moves [2].

2.4. Complexity and Performance

The performance of A* depends heavily on the heuristic. In the worst case, with a poor or zero-valued heuristic ($h(n) = 0$), A* degenerates into Dijkstra's algorithm and explores nodes in expanding circles from the start, with a time complexity proportional to $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges. With a perfect heuristic, A* would only explore the nodes along the optimal path. For game development, the number of nodes expanded is the most critical performance metric. In large maps (e.g., 512x512 grids), a single A* search can become too slow, creating noticeable lag. This is a key motivation for the parallel implementation techniques discussed in Section 5.

3. AI for Strategic and Tactical Decision-Making

Beyond mere navigation, the intelligence of an RTS game's agents—whether they are player-controlled units following commands or computer-controlled opponents—is paramount for engaging gameplay. AI decision-making encompasses a broad spectrum, from low-level unit behaviors like target prioritization to high-level strategic planning and resource management. We will explore several mathematical frameworks for modeling these decisions, ranging from simple utility functions to sophisticated reinforcement learning techniques. (Local Resource: Mathematical Models and Algorithmic Frameworks for Decision-Making in RTS)

3.1. Utility-Based Decision Models

For many immediate, tactical decisions, a **utility-based model** provides a straightforward and efficient approach. Each possible action is assigned a utility score based on predefined criteria, and the AI agent simply chooses the action with the highest utility. This model is particularly effective for scenarios where the consequences of actions are immediate and well-defined.

Consider a unit AI needing to choose a target in a skirmish. The utility of attacking a specific enemy unit `E` could be a function of several factors:

- `DamageOutput(E)`: The damage per second (DPS) the AI unit can inflict on `E`.
- `ThreatLevel(E)`: The DPS `E` can inflict on the AI unit or its allies.
- `CurrentHealth(E)`: Prioritizing low-health targets to eliminate threats faster.
- `Distance(E)`: Proximity to the enemy.

A simple utility function might look like:

$$U\left(Action_{attack,E}\right) =$$
$$= w_1 \cdot racCurrentHealth_{max,E} - CurrentHealth_E CurrentHealth_{max,E} +$$
$$+ w_2 \cdot ThreatLevel(E) - w_3 \cdot Distance(E)$$

where w_1, w_2, w_3 are weights representing the relative importance of each factor, tuned by designers. The AI unit then selects the enemy E^* that maximizes $U\left(Action_{attack,E}\right)$. While easy to implement, utility functions are limited by their explicit, hand-tuned nature and struggle with long-term planning or uncertainty.

3.2. Markov Decision Processes (MDPs) for Long-Term Strategy

For more complex, sequential decision-making problems, where actions influence future states and rewards, a **Markov Decision Process (MDP)** provides a robust mathematical framework. An MDP is defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$:

- $\mathcal{S}$: A finite set of **states** representing the current situation of the game (e.g., number of units, resource counts, map control).
- $\mathcal{A}$: A finite set of **actions** available to the agent in each state (e.g., “build barracks,” “train marine,” “attack enemy base”).
- $\mathcal{T}(s'|s,a)$: The **transition probability function**, giving the probability of reaching state s' from state s after taking action a. This captures the stochastic nature of game environments (e.g., unit production times, combat outcomes).
- $\mathcal{R}(s,a,s')$: The **reward function**, which specifies the immediate reward received after transitioning from state s to state s' by taking action a. Rewards guide the AI towards desirable outcomes (e.g., positive for destroying an enemy building, negative for losing a unit).
- $\gamma \in [0,1)$: The **discount factor**, which determines the present value of future rewards. A γ closer to 0 makes the AI more short-sighted, while a γ closer to 1 encourages long-term planning.

The goal of an AI agent in an MDP is to find an optimal **policy** $\pi^*(s)$, which is a mapping from states to actions that maximizes the expected cumulative discounted reward over time [3].

3.3. The Bellman Equation and Value Iteration

To find the optimal policy, we first need to determine the **optimal value function** $V^*(s)$, which represents the maximum expected cumulative reward starting from state s and acting optimally thereafter. This value function is defined by the **Bellman optimality equation**:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^*(s')]$$

This equation states that the optimal value of a state s is the maximum expected sum of the immediate reward and the discounted optimal value of the next state s', over all possible actions a.

The Bellman equation can be solved iteratively using **value iteration**. Starting with an arbitrary initial value function $V_0(s)$, we repeatedly apply the Bellman update:

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V_k(s')]$$

This iteration converges to $V^*(s)$ as $k \to \infty$. Once $V^*(s)$ is known, the optimal policy $\pi^*(s)$ can be directly derived:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a)[\mathcal{R}(s,a,s') + \gamma V^*(s')]$$

While powerful, solving MDPs explicitly requires knowledge of $\mathcal{T}$ and $\mathcal{R}$, which can be difficult or impossible to obtain for complex RTS environments with a vast number of states and actions.

3.4. Reinforcement Learning: Q-Learning

When the model of the environment (i.e., $\mathcal{T}$ and $\mathcal{R}$) is unknown, **reinforcement learning (RL)** techniques can be employed. **Q-learning** is a popular model-free RL algorithm that directly learns the optimal action-value function, $Q^*(s,a)$, which represents the maximum expected cumulative reward from taking action a in state s and then following the optimal policy thereafter [3].

The Q-value update rule is central to the algorithm: when an agent is in state s, takes action a, observes an immediate reward r, and transitions to a new state s', it updates its estimate of $Q(s,a)$ as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Where:

- $\alpha \in (0,1]$ is the **learning rate**, determining how much new information overrides old information.
- r is the immediate reward.
- $\max_{a'} Q(s',a')$ is the estimate of the optimal future value in the new state s'.

The agent iteratively explores the environment, performs actions, and updates its Q-values. Over time, with sufficient exploration, $Q(s,a)$ converges to $Q^*(s,a)$. The optimal policy is then simply to choose the action a that maximizes $Q(s,a)$ for any given state s.

A critical aspect of Q-learning (and RL in general) is the **exploration-exploitation trade-off.** The agent must explore novel actions to discover potentially better strategies, but also exploit its current knowledge to maximize rewards. This is often managed using an ϵ-greedy policy, where the agent takes a random action with probability ϵ (exploration) and chooses the action with the highest Q-value with probability $1 - \epsilon$ (exploitation).

Q-learning allows RTS AI to learn sophisticated strategies directly from experience, potentially adapting to novel situations and player tactics without explicit programming of every scenario. However, its application to RTS games faces challenges due to the enormous state-action space, requiring techniques like function approximation (e.g., neural networks) for practical implementation.

4. Modeling and Simulating Combat Dynamics

Combat is a central element of RTS games, and its accurate and engaging simulation is crucial for player experience and strategic depth. Game designers often balance determinism (predictable outcomes) with stochasticity (randomness and uncertainty) to create thrilling encounters. This section delves into mathematical models for predicting and resolving combat outcomes, which are central to RTS gameplay balance and AI evaluation. (Local Resource: Combat dynamics models in RTS)

4.1. Deterministic vs. Stochastic Models

Traditionally, some combat models, such as Lanchester's Laws, provided a deterministic approach to warfare, predicting outcomes based on differential equations describing attrition rates. For instance, Lanchester's Linear Law, appropriate for ancient combat with independent targeting, and Square Law, for modern combat with concentrated fire, are:

Linear Law: $\frac{dx}{dt} = -\beta y$ and $\frac{dy}{dt} = -\alpha x$ Square Law: $\frac{dx}{dt} = -\beta y$ and $\frac{dy}{dt} = -\alpha x$

where x and y are the number of units in two opposing forces, and α, β are their respective combat effectiveness coefficients. While historically significant, these deterministic models often oversimplify the complexities of individual unit interactions, terrain effects, and varied unit types inherent in RTS games.

Modern RTS combat often incorporates **stochasticity** to introduce an element of uncertainty, making battles more exciting and less predictable. This randomness can come from hit probabilities, critical hits, dodge chances, and damage ranges. To model these effects accurately, probabilistic and simulation-based approaches are necessary.

4.2. Probabilistic Models of Combat

4.2.1. The Negative Binomial Distribution

Consider a scenario where a unit needs to land a specific number of successful hits to destroy an enemy, and each attack has a certain probability of success. The **Negative Binomial distribution** is well-suited to model the number of *failures* (missed attacks) before a fixed number of *successes* (hits) occur.

Let k be the number of successful hits required to destroy an enemy unit, and p be the probability of a single attack landing a hit. The probability mass function (PMF) of the Negative Binomial distribution, which gives the probability of observing x failures before the k-th success, is:

$$P(X = x|k, p) = \binom{x + k - 1}{k - 1} p^k (1 - p)^x$$

for $x = 0,1,2, \ldots$. The expected number of attacks needed to destroy a unit (including both hits and misses) is k/p. This model can be extended to consider multiple attacking units, but its complexity increases significantly.

4.2.2. Absorbing Markov Chains

For battles involving small numbers of units with discrete states (e.g., alive or dead), **Absorbing Markov Chains** provide a powerful framework for analyzing battle outcomes. A Markov Chain is a sequence of random variables where the probability of moving to the next state depends only on the current state (the Markov property). In an absorbing Markov Chain, some states are "absorbing," meaning once entered, they cannot be left (e.g., a unit is dead, a side has won).

Consider a battle between two forces, A and B. The state of the system can be defined by the number of remaining units for each side, e.g., (N_A, N_B). The absorbing states are those where $N_A = 0$ or $N_B = 0$. From any non-absorbing state, the chain can transition to other states based on probabilities derived from unit attack rates, damage, and health.

The key to analyzing absorbing Markov Chains is the **fundamental matrix** $N = (I - Q)^{-1}$, where I is the identity matrix and Q is the submatrix of transition probabilities between transient (non-absorbing) states. From N, we can derive:

- The expected number of times the chain is in each transient state before absorption.
- The expected number of steps (e.g., rounds of combat) before absorption.
- The probability of being absorbed into each absorbing state.

For example, to calculate the probability that side A wins (i.e., team B reaches 0 units) given an initial state (N_A, N_B), we would use the matrix $B = N \times R$, where R is the submatrix of transition probabilities from transient to absorbing states. The elements of B would give the absorption probabilities into each absorbing state. This allows for precise calculations of win probabilities and expected battle durations for small-scale engagements. (Local Resource: Combat dynamics models in RTS)

4.3. The Monte Carlo Method for Combat Simulation

While analytical models like Absorbing Markov Chains are powerful, they become computationally intractable for large-scale RTS battles involving many units, diverse unit types with special abilities, complex terrain, and dynamic targeting. In such scenarios, the **Monte Carlo method** becomes an invaluable tool.

A Monte Carlo combat simulator involves running a large number of stochastic simulations of the same battle scenario. Each simulation proceeds by:

1. Initializing the state of all units (health, position, targets).
2. In discrete time steps, for each unit: a. Determine its target based on AI rules. b. Roll for hit chance, critical chance, damage, and other random effects. c. Apply damage, reducing target health. d. Check for unit deaths.
3. Repeating until one side is entirely eliminated.

By running thousands or tens of thousands of such simulations, we can gather statistics on the battle's outcome:

- **Win probabilities:** The percentage of simulations won by each side.

- **Expected casualties:** The average number of units lost by each side.
- **Expected battle duration:** The average time steps taken for a battle to conclude.
- **Distribution of outcomes:** The variance and range of possible results.

The power of Monte Carlo lies in its ability to handle arbitrary complexity. Any game mechanic, no matter how intricate or conditional, can be directly simulated. The accuracy of the results improves with the number of simulations. This method is extensively used for balancing units and abilities, predicting large-scale engagements for AI strategic planning, and even for resolving combat in real-time within the game engine, especially when a definitive outcome is needed quickly without complex analytical models [4].

5. Implementation in the Unity Engine

The theoretical models discussed thus far require practical implementation to come to life within a 3D RTS game. The Unity engine, with its component-based architecture and C# scripting, provides a powerful platform. However, the performance demands of RTS games—especially those simulating hundreds of units—necessitate careful architectural design and optimization strategies.

5.1. C# Architecture for Key Systems

Effective implementation begins with a clean, modular architecture.

Pathfinding System: A dedicated `PathfindingManager` (or similar singleton) would orchestrate path requests. Units don't compute paths themselves; they request them.

- **`PathRequest` Struct:** Contains `startPosition`, `targetPosition`, `requesterCallback` (a delegate or event to notify the unit when the path is ready).
- **`GridManager`:** Responsible for generating and maintaining the grid representation of the world. It exposes methods to query cell properties (walkable, obstructed) and convert world coordinates to grid coordinates and vice versa.
- **`Node` Class/Struct:** Represents a single cell in the grid, storing its `gridX`, `gridY`, `worldPosition`, `isWalkable`, `gCost`, `hCost`, and `parent` (for path reconstruction).
- **`Pathfinder` Class:** Implements the A* algorithm. It takes a `PathRequest`, performs the search, and returns a list of `Vector3` waypoints or `Node` objects.

Example C# skeleton for a `PathRequest` and `PathfindingManager`:

```csharp
using UnityEngine;
using System.Collections.Generic;

// Delegate for path completion callback
public delegate void PathCallback(Vector3[] path, bool success);

// Struct to hold a single path request
public struct PathRequest
{
    public Vector3 PathStart;
    public Vector3 PathEnd;
    public PathCallback Callback;

    public PathRequest(Vector3 _start, Vector3 _end, PathCallback _callback)
    {
        PathStart = _start;
        PathEnd = _end;
        Callback = _callback;
    }
}

public class PathfindingManager : MonoBehaviour
{
    Queue<PathRequest> pathRequestQueue = new Queue<PathRequest>();
    PathRequest currentPathRequest;
    Pathfinder pathfinder; // Reference to the actual A* solver
    bool isProcessingPath;

    public static PathfindingManager Instance { get; private set; }

    void Awake()
    {
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            Instance = this;
        }
        pathfinder = GetComponent<Pathfinder>(); // Assuming Pathfinder is on the same GameObject
    }

    public static void RequestPath(Vector3 pathStart, Vector3 pathEnd, PathCallback callback)
    {
        PathRequest newRequest = new PathRequest(pathStart, pathEnd, callback);
        Instance.pathRequestQueue.Enqueue(newRequest);
        Instance.TryProcessNextPath();
    }

    void TryProcessNextPath()
    {
        if (!isProcessingPath && pathRequestQueue.Count > 0)
        {
            currentPathRequest = pathRequestQueue.Dequeue();
            isProcessingPath = true;
            // Start pathfinding on a separate thread or Job System to avoid blocking main thread
            // For now, simulate synchronous call, will optimize later.
            // StartCoroutine(pathfinder.FindPath(currentPathRequest));
            Debug.Log("Processing path request...");
        }
    }

    public void FinishedProcessingPath(Vector3[] path, bool success)
    {
        currentPathRequest.Callback(path, success);
        isProcessingPath = false;
        TryProcessNextPath();
    }
}

```

AI Decision-Making System: AI agents often have an `AIController` script that manages their behavior.

- **`UnitAI` Component:** Attached to each unit, this component references an `AIController` script. It might handle sensor data (seeing enemies, resources) and execute commands.
- **`AIState` Enum/Class:** Define different behaviors like `Idle`, `Patrolling`, `Attacking`, `Harvesting`. A state machine pattern is common.
- **`DecisionEvaluator`:** For utility-based AI, this class would take current game state and evaluate the utility of various actions.
- **`QTable` / `NeuralNetwork`:** For RL-based AI, this would store or compute Q-values based on observed states.

Combat Simulation System:

- **`CombatUnit` Component:** Holds combat-relevant stats (health, damage, attack range, attack speed) and methods for `TakeDamage(float amount)`.
- **`CombatSimulator` Class:** An independent system that, given two lists of `CombatUnit` data, runs a Monte Carlo simulation. It doesn't modify actual game units but predicts outcomes.
- **`BattleResolver` Component:** For real-time combat, this script would manage active engagements, applying damage and resolving deaths in the game world based on game time.

5.2. Performance Optimization with the C# Job System and Burst Compiler

The main thread in Unity is responsible for rendering, input, and physics, making it a critical bottleneck. Complex calculations like A* on large grids or thousands of Monte Carlo simulations can easily cause frame drops. Unity's **C# Job System** and **Burst Compiler** are designed to address this by enabling safe and efficient multi-threading.

The C# Job System allows developers to write thread-safe code that Unity schedules to run on multiple CPU cores. It works with `IJob` interfaces and requires data to be passed by value or via native containers (e.g., `NativeArray<T>`) to ensure data integrity across threads. The Burst Compiler then takes these jobs written in C# and compiles them into highly optimized machine code (leveraging SIMD instructions) at runtime, often achieving performance comparable to native C++ code.

*Example: A Pathfinding with Job System:** Instead of `StartCoroutine(pathfinder.FindPath(currentPathRequest));`, the `PathfindingManager` would schedule a job:

```
using Unity.Jobs;
using Unity.Collections;
using Unity.Burst;

// [BurstCompile] attribute enables Burst compilation for the job
[BurstCompile]
public struct FindPathJob : IJob
{
    public NativeArray<NodeData> Grid; // Simplified grid data structure for job
    public Vector2Int StartNodeCoord;
    public Vector2Int EndNodeCoord;
    // NativeArrays for OPEN/CLOSED lists, path result, etc.

    public void Execute()
    {
        // Implement A* algorithm here, operating on NativeArray data
        // Cannot use Unity API calls (e.g., GameObject.Find), only pure data operations
        // Store results into output NativeArrays
    }
}

// In PathfindingManager:
// ...
JobHandle pathJobHandle;
// ...
void TryProcessNextPath()
{
    if (!isProcessingPath && pathRequestQueue.Count > 0)
    {
        currentPathRequest = pathRequestQueue.Dequeue();
        isProcessingPath = true;

        // Prepare job data (e.g., populate NativeArrays from GridManager)
        // ...
        FindPathJob pathJob = new FindPathJob {
            // Assign grid data, start/end nodes, etc.
        };
        pathJobHandle = pathJob.Schedule(); // Schedule the job
    }
}

void Update() // Or a dedicated Job completion check system
{
    if (isProcessingPath && pathJobHandle.IsCompleted)
    {
        pathJobHandle.Complete(); // Ensure job is finished and data is safe to read
        // Retrieve results from job's output NativeArrays
        // currentPathRequest.Callback(pathResult, success);
        isProcessingPath = false;
        TryProcessNextPath();
    }
}

```

This offloads the computationally intensive A* search from the main thread, allowing the game to maintain a smooth frame rate even during peak path request periods.

5.3. Data Logging for Analysis and Machine Learning

To tune AI, balance combat, and validate experimental designs, robust data logging is essential. In Unity, this can involve writing game state, agent decisions, and combat outcomes to persistent storage.

A dedicated `Logger` class could provide static methods for recording events:

```
using System.IO;
using UnityEngine;

public static class GameLogger
{
    private static string logFilePath;
    private static StreamWriter writer;

```

```csharp
    public static void Initialize(string fileName = "game_log.csv")
    {
        logFilePath = Path.Combine(Application.persistentDataPath, fileName);
        writer = new StreamWriter(logFilePath, append: true); // Append to existing file or create new
        Debug.Log($"Game log initialized at: {logFilePath}");
    }

    public static void LogEvent(string eventType, string details)
    {
        if (writer == null) return;
        string timestamp = System.DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss.fff");
        writer.WriteLine($"{timestamp},{eventType},{details}");
        writer.Flush(); // Ensure data is written to disk immediately
    }

    public static void LogPathfindingResult(Vector3 start, Vector3 end, float timeMs, int nodesVisited, bool success)
    {
        LogEvent("Pathfinding", $"Start:{start},End:{end},Time:{timeMs}ms,Nodes:{nodesVisited},Success:{success}");
    }

    public static void LogCombatOutcome(string initiator, string target, bool win, float duration, int unitsLostA, int unitsLostB)
    {
        LogEvent("Combat", $"Initiator:{initiator},Target:{target},Win:{win},Duration:{duration}s,LostA:{unitsLostA},LostB:{unitsLostB}");
    }

    public static void Close()
    {
        if (writer != null)
        {
            writer.Close();
            writer.Dispose();
            writer = null;
        }
    }
}

```

This system can log various events:

- **Pathfinding metrics:** Start/end points, computation time, number of nodes explored, path length.
- **AI decisions:** Agent ID, state, chosen action, perceived utility, elapsed decision time.
- **Combat events:** Engagement start/end, participants, winner, loser, units lost, total damage dealt, battle duration.

Logged data can be saved as CSV for easy import into spreadsheet software or data analysis tools (e.g., Python with Pandas/Matplotlib) for post-game analysis, AI training, or visualization. This data is critical for understanding system behavior and iterative improvement.

6. Experimental Design and Validation

The mathematical models and implementation strategies discussed are only as valuable as their ability to deliver a compelling and well-performing game. Rigorous experimental design and validation are crucial for tuning parameters, identifying bottlenecks, and ensuring the systems behave as expected. This section outlines a framework for reproducible experimentation in RTS game development.

6.1. Defining Variables and Metrics

Clear definition of independent and dependent variables, along with measurable metrics, is paramount for meaningful experiments.

Independent Variables (Factors to control):

- **Map Complexity:** Size of the grid (e.g., 64x64, 128x128), obstacle density, terrain height variations.
- **Number of Units:** Scaling from a few units to hundreds or thousands.
- **AI Sophistication:** Different AI policies (e.g., utility-based vs. Q-learned), exploration rates (ϵ).
- **Unit Parameters:** Health, damage, attack speed, movement speed, pathfinding heuristic.
- **Simulation Parameters:** Number of Monte Carlo iterations, combat resolution granularity.

Dependent Variables (Metrics to measure):

- **Pathfinding Performance:**
  - `Path Computation Time (ms)`: Average time to find a path.
  - `Nodes Explored`: Number of grid cells visited during search.
  - `Path Length`: Number of segments or distance of the resulting path.
  - `Success Rate`: Percentage of path requests that successfully find a path.
- **AI Decision Performance:**
  - `Decision Latency (ms)`: Time taken for an AI to select an action.
  - `Strategic Effectiveness`: Win rate against other AIs, resource gathering efficiency, unit production rate.
  - `Micro-effectiveness`: Kills/deaths ratio in skirmishes, ability usage effectiveness.
- **Combat System Accuracy/Performance:**
  - `Combat Prediction Accuracy`: How closely Monte Carlo predictions align with actual in-game results.
  - `Simulation Time (ms)`: Time to run a set of Monte Carlo simulations.
  - `Game Balance Metrics`: Win rates for different unit compositions, unit time-to-kill (TTK).

6.2. Reproducibility and Statistical Significance

Given the inherent stochasticity in combat models and AI exploration, ensuring reproducibility and drawing statistically significant conclusions requires careful experimental setup.

- **RNG Seeding:** All random number generators used in simulations (e.g., hit chances, AI exploration, Monte Carlo rolls) **must** be explicitly seeded at the start of each experiment run. This ensures that the exact same sequence of random numbers is generated, allowing for identical reruns of an experiment to verify results. Unity's `Random.InitState()` is crucial here.
- **Sample Sizes:** For stochastic simulations (e.g., Monte Carlo combat, Q-learning training), a sufficiently large number of samples (simulations or training episodes) is required to ensure that observed averages and distributions are representative. Statistical power analysis can help determine appropriate sample sizes.
- **Confidence Intervals:** When reporting average metrics from stochastic processes, always include **confidence intervals** (e.g., 95% CI). This provides a measure of the reliability of the estimate and helps distinguish genuine effects from random fluctuations. Plots should include error bars.
- **Control Groups:** To isolate the effect of a specific change, experiments should include a control group (e.g., baseline AI) against which experimental groups are compared.

6.3. Expected Plots and Visualization

Visualizing experimental results is key to interpreting complex data.

- **Pathfinding:**
  - Plot: Path computation time vs. map size (log-log scale) for different heuristics.
  - Plot: Nodes explored vs. obstacle density.
  - Heatmaps: Visualize grid cells explored by A* for different heuristics on a specific map.
- **AI Decision-Making:**
  - Plot: AI win rate vs. training episodes for Q-learning (showing convergence).
  - Plot: Average reward per episode vs. training episodes.
  - Box plots: Distribution of decision latencies for different AI complexities.
- **Combat Simulation:**
  - Histograms: Distribution of battle durations or units remaining for Monte Carlo simulations.
  - Scatter plots: Predicted win probability (Monte Carlo) vs. actual win rate (in-game trials).
  - Time series plots: Unit counts over time during a simulated battle.

By adhering to these experimental principles, developers can systematically evaluate and refine the mathematical models and algorithms underpinning their RTS game, moving from qualitative observations to quantitative, data-driven improvements.

7. How This Paper Used the Local Resources

The foundational concepts and initial theoretical frameworks for this paper were significantly informed and inspired by the local markdown documents provided: `Combat dynamics models in RTS.md`, `Mathematical Models and Algorithmic Frameworks for Decision-Making in RTS.md`, and `Pathfinding in Real-Time Strategy Games.md`. These resources served as primary guides, outlining key mathematical techniques, algorithmic considerations, and common challenges within their respective domains. Specifically, they provided a structured starting point for developing the in-depth discussions on A* pathfinding algorithms, various AI decision-making paradigms (from utility theory to MDPs and Q-learning), and the application of probabilistic and simulation methods for combat resolution (e.g., Negative Binomial, Markov Chains, and Monte Carlo). The content from these documents was integrated, expanded upon with formal derivations, and contextualized within the broader scope of 3D RTS game development.

8. Conclusion

The development of a sophisticated Real-Time Strategy 3D game is a formidable engineering and mathematical undertaking, demanding elegant solutions to complex problems in pathfinding, artificial intelligence, and combat simulation. This paper has systematically explored the core mathematical models and computational techniques that underpin these critical systems. We began with a formal exposition of the A* search algorithm, demonstrating its efficiency and optimality for unit navigation in discrete game environments, highlighting the importance of admissible heuristics. We then delved into the realm of AI decision-making, progressing from straightforward utility-based models for immediate actions to the more powerful and adaptive frameworks of Markov Decision Processes (MDPs) and Q-learning, which enable agents to learn optimal long-term strategies even in stochastic and partially observable environments. For combat resolution, we presented both probabilistic models, such as the Negative Binomial distribution and Absorbing Markov Chains for analytical insights, and the highly flexible Monte Carlo simulation method for handling intricate, large-scale engagements.

Crucially, we bridged the gap between theory and practice by outlining practical implementation strategies within the Unity engine. This included architectural considerations for C# scripting and, critically, leveraging the C# Job System and Burst Compiler to unlock multi-threaded performance, addressing the demanding real-time constraints of RTS titles. Finally, we emphasized the necessity of rigorous experimental design, stressing the importance of clear metrics, RNG seeding, statistical analysis with confidence intervals, and effective data logging to systematically validate and refine these complex systems.

While this paper provides a comprehensive overview, the field of RTS game development continues to evolve. Future work could explore hierarchical pathfinding for improved scalability on vast maps, integrate more advanced reinforcement learning architectures (e.g., Deep Q-Networks or Actor-Critic methods) to handle continuous state/action spaces, or investigate hybrid combat resolution systems that blend analytical and simulation approaches for optimal performance and accuracy. The ongoing quest to create ever more intelligent, realistic, and engaging RTS experiences will continue to drive innovation at the intersection of applied mathematics and game technology.

9. References

[1] P. Hart, N. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, 1968. (Referencing: https://en.wikipedia.org/wiki/A*_search_algorithm) [2] "A* Pathfinding for Beginners," *Red Blob Games*. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/. [Accessed: Nov. 26, 2025]. [3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. (Referencing: http://incompleteideas.net/book/the-book.html) [4] "Monte Carlo method," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method. [Accessed: Nov. 26, 2025]. [5] "Absorbing Markov chain," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Absorbing_Markov_chain. [Accessed: Nov. 26, 2025]. [6] "Unity Manual," *Unity Technologies*. [Online]. Available: https://docs.unity3d.com/Manual/index.html. [Accessed: Nov. 26, 2025].

10. Appendix

10.1. Full Derivation of the Fundamental Matrix for an Absorbing Markov Chain

Consider an absorbing Markov chain with t transient states and r absorbing states. We can arrange the transition probability matrix P into a canonical form:

$$P = \begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}$$

Where:

- Q is a $t \times t$ matrix representing transition probabilities between transient states.
- R is a $t \times r$ matrix representing transition probabilities from transient states to absorbing states.
- 0 is an $r \times t$ matrix of zeros (no transitions from absorbing to transient states).
- I is an $r \times r$ identity matrix (transitions from an absorbing state to itself with probability 1).

The **fundamental matrix** N for an absorbing Markov chain is given by:

$$N = (I_t - Q)^{-1}$$

where I_t is the $t \times t$ identity matrix. Each element n_{ij} of N represents the expected number of times the chain is in transient state j, given that it started in transient state i, before it is absorbed.

Derivation: Let N_k be a matrix whose ij-th entry is the probability of being in transient state j after k steps, given that we started in transient state i. $N_0 = I_t$ (start in state i, means 1 visit to i at step 0). $N_1 = Q$ (after 1 step, probability of being in j is q_{ij}). $N_2 = Q^2$ ... $N_k = Q^k$

The expected number of times the chain is in state j starting from state i is the sum of probabilities of being in state j at each step k:

$$N = \sum_{k=0}^{\infty} Q^k$$

This is a geometric series of matrices. If the spectral radius of Q is less than 1 (which it is for an absorbing Markov chain), this series converges to:

$$N = (I_t - Q)^{-1}$$

Once N is computed, we can also determine the **absorption probabilities.** Let B be a $t \times r$ matrix where b_{ij} is the probability that the chain starting in transient state i is absorbed into absorbing state j. This matrix is given by:

$$B = N \times R$$

The elements b_{ij} can be directly used to calculate win probabilities for each side in a combat model by mapping absorbing states to “side A wins” or “side B wins.”

*10.2. Sample C# Implementation of Grid-Based A**

This skeleton provides the core data structures and logic for a basic A* pathfinder.

```csharp
using System.Collections.Generic;
using UnityEngine;

// Node struct to represent a grid cell
public struct Node
{
    public int GridX;
    public int GridY;
    public Vector3 WorldPosition;
    public bool IsWalkable;

    public int GCost; // Cost from start node
    public int HCost; // Heuristic cost to end node
    public int FCost => GCost + HCost; // Total cost

    public Node Parent; // For path reconstruction

    public Node(int _gridX, int _gridY, Vector3 _worldPos, bool _isWalkable)
    {
        GridX = _gridX;
        GridY = _gridY;
        WorldPosition = _worldPos;
        IsWalkable = _isWalkable;

        GCost = 0;
        HCost = 0;
        Parent = new Node(); // Default struct, will be overwritten
    }

    // Equality check for Nodes
    public override bool Equals(object obj)
    {
        if (!(obj is Node)) return false;
        Node other = (Node)obj;
        return GridX == other.GridX && GridY == other.GridY;
    }

    public override int GetHashCode()
    {
        return GridX.GetHashCode() ^ GridY.GetHashCode();
    }
}

public class Pathfinder : MonoBehaviour
{
    public GridManager Grid; // Reference to your grid data structure

    public Vector3[] FindPath(Vector3 startWorldPos, Vector3 targetWorldPos)
    {
        Node startNode = Grid.NodeFromWorldPoint(startWorldPos);
        Node targetNode = Grid.NodeFromWorldPoint(targetWorldPos);

        if (!startNode.IsWalkable || !targetNode.IsWalkable)
        {
            Debug.LogWarning("Start or target node is unwalkable.");
            return null; // Or handle error appropriately
        }

        List<Node> openSet = new List<Node>(); // Could be a min-heap/priority queue for efficiency
        HashSet<Node> closedSet = new HashSet<Node>();

        openSet.Add(startNode);

        while (openSet.Count > 0)
        {
            Node currentNode = openSet[0];
            for (int i = 1; i < openSet.Count; i++)
            {
                if (openSet[i].FCost < currentNode.FCost || (openSet[i].FCost == currentNode.FCost && openSet[i].HCost < currentNode.HCost))
                {
                    currentNode = openSet[i];
                }
            }

            openSet.Remove(currentNode);
            closedSet.Add(currentNode);

            if (currentNode.Equals(targetNode))
            {
                return ReconstructPath(startNode, currentNode);
            }

            foreach (Node neighbour in Grid.GetNeighbours(currentNode))
            {
                if (!neighbour.IsWalkable || closedSet.Contains(neighbour))
                {
                    continue;
                }

```

```csharp
                int newMovementCostToNeighbour = currentNode.GCost + GetDistance(currentNode, neighbour);
                if (newMovementCostToNeighbour < neighbour.GCost || !openSet.Contains(neighbour))
                {
                    neighbour.GCost = newMovementCostToNeighbour;
                    neighbour.HCost = GetDistance(neighbour, targetNode);
                    neighbour.Parent = currentNode;

                    if (!openSet.Contains(neighbour))
                    {
                        openSet.Add(neighbour);
                    }
                }
            }
        }
        return null; // Path not found
    }

    Vector3[] ReconstructPath(Node startNode, Node endNode)
    {
        List<Vector3> path = new List<Vector3>();
        Node currentNode = endNode;

        while (!currentNode.Equals(startNode))
        {
            path.Add(currentNode.WorldPosition);
            currentNode = currentNode.Parent;
        }
        path.Reverse();
        return path.ToArray();
    }

    // Heuristic function: Manhattan Distance example
    int GetDistance(Node nodeA, Node nodeB)
    {
        int distX = Mathf.Abs(nodeA.GridX - nodeB.GridX);
        int distY = Mathf.Abs(nodeA.GridY - nodeB.GridY);

        if (distX > distY)
            return 14 * distY + 10 * (distX - distY); // Diagonal + straight cost
        return 14 * distX + 10 * (distY - distX);
    }
}

// Dummy GridManager for context. In a real project, this would manage your grid data.
public class GridManager : MonoBehaviour
{
    public Node NodeFromWorldPoint(Vector3 worldPoint)
    {
        // Placeholder: Convert world position to grid node.
        // In a real implementation, this would involve raycasting or
        // simple division/modulo based on grid origin and cell size.
        return new Node((int)worldPoint.x, (int)worldPoint.z, worldPoint, true);
    }

    public List<Node> GetNeighbours(Node node)
    {
        // Placeholder: Return 8 neighbours (for 8-directional movement)
        List<Node> neighbours = new List<Node>();
        for (int x = -1; x <= 1; x++)
        {
            for (int y = -1; y <= 1; y++)
            {
                if (x == 0 && y == 0) continue;
```

```

                    int checkX = node.GridX + x;
                    int checkY = node.GridY + y;

                    // Assuming a fixed grid size for bounds checking
                    if (checkX >= 0 && checkX < 100 && checkY >= 0 && checkY < 100)
                    {
                        // In a real grid, you'd fetch the actual Node object from a 2D array or similar
                        // For this example, we just create a dummy walkable node.
                        neighbours.Add(new Node(checkX, checkY, new Vector3(checkX, 0, checkY), true));
                    }
                }
            }
            return neighbours;
        }
}

```

10.3. Sample C# Implementation of a Monte Carlo Combat Simulator

This skeleton demonstrates how to run multiple simulations of a simple battle between two teams.

```
using UnityEngine;
using System.Collections.Generic;
using System.Linq;

// Simple struct to represent a unit's combat stats
public struct CombatStats
{
    public float MaxHealth;
    public float CurrentHealth;
    public float Damage;
    public float AttackSpeed; // Attacks per second
    public float HitChance;   // Probability of hitting
    public float CriticalChance; // Probability of critical hit
    public float CriticalMultiplier; // Damage multiplier for critical hits

    public CombatStats(float hp, float dmg, float atkSpd, float hitC, float critC, float critMult)
    {
        MaxHealth = hp;
        CurrentHealth = hp;
        Damage = dmg;
        AttackSpeed = atkSpd;
        HitChance = hitC;
        CriticalChance = critC;
        CriticalMultiplier = critMult;
    }

    public bool IsAlive => CurrentHealth > 0;

    // Method to apply damage to this unit
    public void TakeDamage(float amount)
    {
        CurrentHealth -= amount;
        if (CurrentHealth < 0) CurrentHealth = 0;
    }
}

public class MonteCarloCombatSimulator : MonoBehaviour
```

```
{
    public int NumberOfSimulations = 1000;
    public float TimeStep = 0.1f; // Simulation granularity in seconds

    // Simulates a single combat instance
    // Returns true if Team A wins, false otherwise
    public bool SimulateSingleCombat(List<CombatStats> initialTeamA, List<CombatStats> initialTeamB, System.Random rng)
    {
        // Create deep copies to avoid modifying initial lists and ensure each simulation is fresh
        List<CombatStats> teamA = initialTeamA.Select(s => new CombatStats(s.MaxHealth, s.MaxHealth, s.Damage, s.AttackSpeed, s.HitChance, s.CriticalChance, s.CriticalMultiplier)).ToList();
        List<CombatStats> teamB = initialTeamB.Select(s => new CombatStats(s.MaxHealth, s.MaxHealth, s.Damage, s.AttackSpeed, s.HitChance, s.CriticalChance, s.CriticalMultiplier)).ToList();

        float currentTime = 0;
        const float maxSimulationTime = 300f; // Prevent infinite loops in balanced or stalemated scenarios

        while (teamA.Any(u => u.IsAlive) && teamB.Any(u => u.IsAlive) && currentTime < maxSimulationTime)
        {
            // Simulate attacks for Team A
            for (int i = 0; i < teamA.Count; i++)
            {
                if (!teamA[i].IsAlive) continue; // Skip dead units

                // Determine if unit attacks in this timestep
                if (rng.NextDouble() < (teamA[i].AttackSpeed * TimeStep))
                {
                    // Select a random living target from Team B
                    List<CombatStats> livingTargetsB = teamB.Where(u => u.IsAlive).ToList();
                    if (livingTargetsB.Any())
                    {
                        CombatStats target = livingTargetsB[rng.Next(0, livingTargetsB.Count)];
                        int targetIndex = teamB.IndexOf(target); // Find original index to update

                        // Check for hit
                        if (rng.NextDouble() < teamA[i].HitChance)
                        {
                            float actualDamage = teamA[i].Damage;
                            // Check for critical hit
                            if (rng.NextDouble() < teamA[i].CriticalChance)
                            {
                                actualDamage *= teamA[i].CriticalMultiplier;
                            }
                            target.TakeDamage(actualDamage);
                            teamB[targetIndex] = target; // Update the unit in the list
                        }
                    }
                }
            }

            // Simulate attacks for Team B
            for (int i = 0; i < teamB.Count; i++)
            {
                if (!teamB[i].IsAlive) continue; // Skip dead units

                if (rng.NextDouble() < (teamB[i].AttackSpeed * TimeStep))
                {
                    List<CombatStats> livingTargetsA = teamA.Where(u => u.IsAlive).ToList();
```

```csharp
                    if (livingTargetsA.Any())
                    {
                        CombatStats target = livingTargetsA[rng.Next(0, livingTargetsA.Count)];
                        int targetIndex = teamA.IndexOf(target);

                        if (rng.NextDouble() < teamB[i].HitChance)
                        {
                            float actualDamage = teamB[i].Damage;
                            if (rng.NextDouble() < teamB[i].CriticalChance)
                            {
                                actualDamage *= teamB[i].CriticalMultiplier;
                            }
                            target.TakeDamage(actualDamage);
                            teamA[targetIndex] = target;
                        }
                    }
                }
            }

            currentTime += TimeStep;
        }

        return teamA.Any(u => u.IsAlive); // If any unit in team A is alive, team A wins
    }

    public void RunCombatSimulations(List<CombatStats> teamAUnits, List<CombatStats> teamBUnits, int seed)
    {
        int teamAWins = 0;
        System.Random masterRng = new System.Random(seed); // Master RNG for reproducibility

        for (int i = 0; i < NumberOfSimulations; i++)
        {
            // Use a new System.Random instance for each simulation, seeded from the master RNG,
            // to ensure independent random sequences per simulation but reproducible overall results.
            System.Random simulationRng = new System.Random(masterRng.Next());

            if (SimulateSingleCombat(teamAUnits, teamBUnits, simulationRng))
            {
                teamAWins++;
            }
        }

        float winRateA = (float)teamAWins / NumberOfSimulations;
        Debug.Log($"Simulations complete for seed {seed}. Team A Win Rate: {winRateA:P2}");
        // Further analysis can be added here, e.g., average units remaining, average duration, etc.
    }

    // Example usage
    void Start()
    {
        List<CombatStats> teamA = new List<CombatStats>
        {
            new CombatStats(100, 10, 1.0f, 0.8f, 0.1f, 1.5f), // Unit 1
            new CombatStats(100, 10, 1.0f, 0.8f, 0.1f, 1.5f)  // Unit 2
        };
        List<CombatStats> teamB = new List<CombatStats>
        {
            new CombatStats(120, 8, 0.9f, 0.9f, 0.05f, 1.8f), // Unit 1
            new CombatStats(120, 8, 0.9f, 0.9f, 0.05f, 1.8f)  // Unit 2
        };

        RunCombatSimulations(teamA, teamB, 12345); // Run with a fixed seed for reproducibility
    }
}

```