Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

European Master in Software Engineering

Master Thesis

New WoA Interface, Adapting and Improving an Old Agents Practice using Unity 3D

Author: Jose Rodríguez de Santiago

June, 2025

Master Thesis

European Master in Software Engineering

Title: New WoA Interface, Adapting and Improving an Old Agents Practice using Unity 3D
June, 2025

Author: Jose Rodríguez de Santiago

Supervisor:	*Co-supervisor:*
Ricardo Imbert Paredes	Jose María Barambones Ramírez
Ph.D. in Computer Science	Ph.D. in Computer Science
UPM	UPM
DLSIIS	DLSIIS
UPM	UPM

Abstract

This Master's thesis presents the design and development of an enhanced graphical interface for the educational simulation platform New World of Agents (NewWoA), leveraging Unity 3D to modernize an outdated agent-based exercise used in graduate-level university practice. The new interface serves as a visualization layer that communicates with agent logic implemented via the JADE framework, interpreting student-generated API calls into real-time, animated interactions.

The work focuses on rebuilding the platform's usability, accessibility, and visual appeal by implementing a fully modular Unity architecture. The system renders unit actions such as movement, attacks, and eliminations, while abstracting away the game's decision-making logic to the student-developed agents. A thorough analysis of existing agent-based and Unity-driven educational tools is conducted to situate the contribution within the state of the art.

The project is developed following agile methodology across several structured sprints, with all code, configurations, and interfaces prepared for deployment and future extension.

The result is a stable, functional tool ready for deployment in educational contexts. The interface significantly improves the understanding of the agents' behaviour and provides clarity for both monitoring and evaluation. The whole system has been designed with a modular architecture, thinking about its maintenance and future evolution by other teams, including explicit guidelines for maintaining, expanding, and handing over the system, ensuring long-term sustainability in an academic context.

Resumen

Este Trabajo Fin de Máster presenta el diseño y desarrollo de una interfaz gráfica mejorada para la plataforma de simulación educativa New World of Agents (NewWoA), utilizando Unity 3D para modernizar un ejercicio basado en agentes previamente utilizado en prácticas universitarias de la asignatura de Agent-based Software Development. La nueva interfaz actúa como una capa de visualización que se comunica con la lógica de los agentes implementada a través del framework JADE, interpretando llamadas a la API generadas por los estudiantes en interacciones animadas en tiempo real.

El trabajo se centra en reconstruir la usabilidad, accesibilidad y atractivo visual de la plataforma mediante la implementación de una arquitectura modular en Unity. El sistema representa acciones de las unidades como movimientos, ataques y eliminaciones, delegando toda la lógica de toma de decisiones a los agentes desarrollados por los estudiantes. Además, se realiza un análisis exhaustivo de herramientas educativas existentes basadas en agentes y desarrolladas con Unity, con el fin de contextualizar la contribución dentro del estado del arte.

El proyecto se desarrolla siguiendo una metodología ágil estructurada en varios sprints, dejando todo el código, configuración e interfaces preparados para su despliegue y futura ampliación.

El resultado es una herramienta estable y funcional lista para su uso en contextos educativos. La nueva interfaz mejora significativamente la comprensión del comportamiento de los agentes y facilita tanto su seguimiento como evaluación. Todo el sistema ha sido diseñado con una arquitectura modular, pensando en su mantenimiento y evolución futura por parte de otros equipos, incluyendo directrices explícitas para su mantenimiento, ampliación y traspaso, garantizando así su sostenibilidad a largo plazo en un entorno académico.

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 1
- 1.3 Objectives . . . 2
- 1.4 Document Structure . . . 3
- 1.5 Methodology and Way Of Working . . . 4

2 Context: Game Rules **5**
- 2.1 Motivation . . . 5
- 2.2 Map . . . 5
- 2.3 Starting distribution . . . 5
- 2.4 Phases . . . 5
- 2.5 Activity visualization . . . 6
- 2.6 Units . . . 7
- 2.7 Results . . . 7
- 2.8 Development recommendations . . . 8

3 State of the Art of the used Technologies **9**
- 3.1 State of the Art: Agent-Based Systems (Agents Methodology) . . . 9
  - 3.1.1 Theoretical Foundations and Evolution . . . 9
  - 3.1.2 Current Advancements in Agent-Based Systems . . . 9
  - 3.1.3 Educational Applications and Relevance to NewWoA . . . 10
  - 3.1.4 Challenges and Future Directions . . . 11
  - 3.1.5 Relevance to NewWoA . . . 11
- 3.2 State of the Art: Unity 3D . . . 11
  - 3.2.1 Theoretical Foundations and Evolution . . . 12
  - 3.2.2 Current Advancements in Unity 3D . . . 12
  - 3.2.3 Educational Applications and Relevance to NewWoA . . . 13
  - 3.2.4 Challenges and Future Directions . . . 13
  - 3.2.5 Relevance to NewWoA . . . 14
- 3.3 State of the Art: APIs and Supporting Technologies . . . 14
  - 3.3.1 Theoretical Foundations and Evolution . . . 14
  - 3.3.2 Current Advancements in APIs and Supporting Technologies 15
  - 3.3.3 Educational Applications and Relevance to NewWoA . . . 15
  - 3.3.4 Challenges and Future Directions . . . 16
  - 3.3.5 Relevance to NewWoA . . . 16

4 Developed Scripts **17**
- 4.1 Script Architecture Explanation . . . 17
- 4.2 CameraController.cs - Explanation . . . 18
  - 4.2.1 Overview . . . 18
  - 4.2.2 Singleton Pattern . . . 18
  - 4.2.3 Centering and Zooming the Camera . . . 19
  - 4.2.4 Adjusting Zoom . . . 19
  - 4.2.5 Camera Movement . . . 19
- 4.3 UIManager.cs - Explanation . . . 20
  - 4.3.1 Overview . . . 20

4.3.2 Singleton Pattern 20
4.3.3 UI Initialization 20
4.3.4 Starting and Ending the Game 20
4.3.5 Updating the Results Text 21
4.4 MapGenerator.cs - Explanation 21
4.4.1 Overview 21
4.4.2 Singleton Pattern 22
4.4.3 Map Initialization 22
4.4.4 Tile Dictionaries Initialization 22
4.4.5 Generating Tiles 23
4.4.6 Map Generation Logic 23
4.4.7 Legend display* 23
4.5 UnitManager.cs - Explanation 24
4.5.1 Overview 24
4.5.2 Singleton Pattern 24
4.5.3 Unit Prefabs and Team Colors 24
4.5.4 Spawning Units 25
4.5.5 Moving Units Smoothly 25
4.5.6 Handling Attacks 25
4.5.7 Displaying Damage Text 26
4.5.8 Updating Unit Health 26
4.5.9 Killing Units 26
4.5.10 Finding Units by ID 26
4.6 UnityMainThreadDispatcher.cs - Explanation 26
4.6.1 Overview 26
4.6.2 Execution Queue 27
4.6.3 Enqueue Method 27
4.6.4 Update Method 27
4.6.5 OnApplicationQuit Method 27
4.7 HttpListenerServer.cs - Explanation 28
4.7.1 Overview 28
4.7.2 Class Setup and Initialization 28
4.7.3 Handling HTTP Requests 28
4.7.4 Endpoint Handlers 29
4.7.4.1 Initialize Map Endpoint 29
4.7.4.2 Create Unit Endpoint 30
4.7.4.3 Move Unit Endpoint 30
4.7.4.4 Attack Endpoint 30
4.7.4.5 Kill Unit Endpoint 31
4.7.4.6 Update Health Endpoint 31
4.7.4.7 Start Game Endpoint 31
4.7.4.8 End Game Endpoint 32
4.7.4.9 Invalid Endpoint Handling 32
4.7.5 Cleanup on Application Quit 32
4.7.6 Request Models 33
4.8 RealSimulation.py - Explanation 34
4.8.1 Overview 34
4.8.2 Imports and Dependencies 34
4.8.3 Game Configuration 34
4.8.4 Unit Definitions 35
4.8.5 State Management 35

4.8.6 Grid Setup . . . 35
4.8.7 API Interaction . . . 36
4.8.8 Game Initialization . . . 36
4.8.9 Unit Deployment . . . 36
4.8.10 Unit Actions . . . 37
4.8.11 Unit Elimination . . . 37
4.8.12 Game End and Scoring . . . 37
4.8.13 Main Simulation Loop . . . 38
4.9 Development Decisions . . . 38
4.9.1 Overview . . . 38
4.9.2 Singleton Usage . . . 39
4.9.2.1 Justification for Using Singleton in Some Scripts . . . 39
4.9.2.2 Justification for Not Using Singleton in Other Scripts 40
4.9.2.3 Architectural Rationale . . . 40
4.9.3 API Integration with HttpListenerServer and Petition Queuing 41
4.9.3.1 Technical Implementation . . . 41
4.9.3.2 Benefits of the Design . . . 42
4.9.3.3 Alternative Approaches . . . 42
4.9.3.4 Rationale for Chosen Approach . . . 44
4.10 Game Rules for Student Implementation . . . 44

5 Unity Structure and Visuals 48
5.1 Gameobjects and Component Structure . . . 48
5.2 Functional GameObject Analysis . . . 49
5.2.1 Main Camera . . . 49
5.2.2 Map . . . 49
5.2.3 GameManager . . . 49
5.2.4 UnitManager . . . 50
5.2.5 HttpServer . . . 50
5.3 UI Elements Analysis . . . 51
5.3.1 Start Game Panel . . . 51
5.3.1.1 StartGameMainBackground . . . 51
5.3.1.2 StartGameBorders . . . 51
5.3.1.3 StartGameTextBackGround . . . 52
5.3.1.4 StartGameText . . . 52
5.3.2 Start Game Panel . . . 53
5.3.2.1 EndGameMainBackGround . . . 53
5.3.2.2 EndGameBorders . . . 53
5.3.2.3 EndgameTextBackGround . . . 53
5.3.2.4 EndgameText . . . 54
5.3.3 Legend . . . 54
5.3.3.1 LegendBorders . . . 54
5.3.3.2 LegendBackGround . . . 55
5.3.3.3 LegendSliders . . . 55
5.4 Dinamically Created GamObjects . . . 57
5.4.1 Map . . . 57
5.4.2 Units . . . 58
5.4.3 Attacks . . . 59

6 Test Game Representation 61

7 Methodology Used 65

7.1 Introduction to Sprint Documentation 65
7.1.1 Agile Methodology 65
7.1.2 User Story Point Estimation 65
7.1.3 Developed Epics 65
7.1.4 Sprint Structure 66
7.2 Sprint 1 - Map Generation with Code and Visual 66
7.2.1 User Stories 67
7.2.1.1 US 1.1 Initialize Grid Layout 67
7.2.1.2 US 1.2 Adjust Grid Size Dynamically 67
7.2.1.3 US 1.3 Apply Visual Tilemap 67
7.2.1.4 US 1.4 Document Sprint Progress 67
7.2.2 Stats of the Sprint 68
7.2.3 Expected Results of the Sprint 68
7.2.4 Sprint Results 68
7.2.5 Generated Products 68
7.2.6 Retrospective 68
7.2.6.1 What went well? 68
7.2.6.2 What could be improved? 68
7.2.6.3 What will we do differently? 69
7.3 Sprint 2 - Unit Creation and Movement 69
7.3.1 User Stories 69
7.3.1.1 US 2.1 Spawn Units on Grid 69
7.3.1.2 US 2.2 Display Units as White Squares 69
7.3.1.3 US 2.3 Move Units to Target Position 69
7.3.1.4 US 2.4 Ensure Consistent Movement 70
7.3.1.5 US 2.5 Document Sprint Progress 70
7.3.2 Stats of the Sprint 70
7.3.3 Expected Results of the Sprint 70
7.3.4 Sprint Results 70
7.3.5 Generated Products 71
7.3.6 Retrospective 71
7.3.6.1 What went well? 71
7.3.6.2 What could be improved? 71
7.3.6.3 What will we do differently? 71
7.4 Sprint 3 - API Integration and First Build 71
7.4.1 User Stories 71
7.4.1.1 US 3.1 Implement API for Unit Creation 71
7.4.1.2 US 3.2 Implement API for Unit Movement 72
7.4.1.3 US 3.3 Generate First Build 72
7.4.1.4 US 3.4 Create YAML Configuration 72
7.4.1.5 US 3.5 Create Postman Collection 72
7.4.1.6 US 1.4 Document Sprint Progress 73
7.4.2 Stats of the Sprint 73
7.4.3 Expected Results of the Sprint 73
7.4.4 Sprint Results 73
7.4.5 Generated Products 73
7.4.6 Retrospective 74
7.4.6.1 What went well? 74
7.4.6.2 What could be improved? 74
7.4.6.3 What will we do differently? 74
7.5 Sprint 4 - Attacks and API Integration 74

7.5.1 User Stories . . . 74
7.5.1.1 US 4.1 Implement Attack Functionality . . . 74
7.5.1.2 US 4.2 Implement API for Attacks . . . 74
7.5.1.3 US 4.3 Generate Second Build . . . 75
7.5.1.4 US 4.4 Update YAML and Postman Collection . . . 75
7.5.1.5 US 4.5 Document Sprint Progress . . . 75
7.5.2 Stats of the Sprint . . . 75
7.5.3 Expected Results of the Sprint . . . 75
7.5.4 Sprint Results . . . 76
7.5.5 Generated Products . . . 76
7.5.6 Retrospective . . . 76
7.5.6.1 What went well? . . . 76
7.5.6.2 What could be improved? . . . 76
7.5.6.3 What will we do differently? . . . 76
7.6 Sprint 5 - API Overhaul and Visual Enhancements . . . 76
7.6.1 User Stories . . . 77
7.6.1.1 US 5.1 Assign Unit and Team IDs . . . 77
7.6.1.2 US 5.2 Update APIs with ID-Based Parameters . . . 77
7.6.1.3 US 5.3 Add Game Start Endpoint . . . 77
7.6.1.4 US 5.4 Add Game End Endpoint . . . 77
7.6.1.5 US 5.5 Add Unit State Endpoints . . . 78
7.6.1.6 US 5.6 Enhance Unit Visuals . . . 78
7.6.1.7 US 5.7 Document Sprint Progress . . . 78
7.6.2 Stats of the Sprint . . . 78
7.6.3 Expected Results of the Sprint . . . 79
7.6.4 Sprint Results . . . 79
7.6.5 Generated Products . . . 79
7.6.6 Retrospective . . . 79
7.6.6.1 What went well? . . . 79
7.6.6.2 What could be improved? . . . 79
7.6.6.3 What will we do differently? . . . 79
7.7 Sprint 6 - Visual Improvements and UI . . . 80
7.7.1 User Stories . . . 80
7.7.1.1 US 6.1 Add Unit Type Sprites . . . 80
7.7.1.2 US 6.2 Implement Start Game UI . . . 80
7.7.1.3 US 6.3 Implement End Game UI . . . 80
7.7.1.4 US 6.4 Generate Third Build . . . 80
7.7.1.5 US 6.5 Document Sprint Progress . . . 81
7.7.2 Stats of the Sprint . . . 81
7.7.3 Expected Results of the Sprint . . . 81
7.7.4 Sprint Results . . . 81
7.7.5 Generated Products . . . 81
7.7.6 Retrospective . . . 82
7.7.6.1 What went well? . . . 82
7.7.6.2 What could be improved? . . . 82
7.7.6.3 What will we do differently? . . . 82
7.8 Sprint 7 - Final Features and Demo Script . . . 82
7.8.1 User Stories . . . 82
7.8.1.1 US 7.1 Implement Zoom Scroll . . . 82
7.8.1.2 US 7.2 Update Start Game UI Panels . . . 82
7.8.1.3 US 7.3 Create Demo Script . . . 83

7.8.1.4 US 7.4 Generate Fourth Build 83
7.8.1.5 US 7.5 Document Sprint Progress 83
7.8.2 Stats of the Sprint 83
7.8.3 Expected Results of the Sprint 84
7.8.4 Sprint Results 84
7.8.5 Generated Products 84
7.8.6 Retrospective 84
7.8.6.1 What went well? 84
7.8.6.2 What could be improved? 84
7.8.6.3 What will we do differently? 84
7.8.7 Final Retrospective 85

8 Future Steps, Handover Guide and Conclusions 86
8.1 Conclusions 86
8.2 Vertical Improvements: Expanding Functional Capabilities 87
8.2.1 Overview 87
8.2.2 Replay System with Playback Controls 87
8.2.3 Dynamic Unit Spawning During Match 87
8.2.4 API-Driven Game Configuration 88
8.2.5 Obstacle Integration in the Game Map 88
8.3 Horizontal Improvements: Enhancing Existing Capabilities 89
8.3.1 Overview 89
8.3.2 Modular User Interface with Tabbed Panels 89
8.3.3 Enhanced Camera and Visual Navigation Controls 90
8.3.4 Structured Match Summary Screen 90
8.4 Maintenance and Future Development Guidelines 91
8.4.1 Updating Initial Unit Health Points 91
8.4.2 Adding New Unit Types 91
8.4.3 Adding New API Endpoints 92
8.4.4 General Guidelines for Future Contributors 92

9 Annexes 95

1 Introduction

1.1 Context

Agent-based systems, where entities operate within a defined environment, are central to artificial intelligence and computational modeling, with applications in robotics, game AI, simulations, and distributed computing. For master's students, engaging with these systems through practical exercises provides a valuable opportunity to apply theoretical knowledge to real-world scenarios. Designing and analyzing agent behaviors helps students explore concepts like interaction, coordination, and environmental response, building a foundation for advanced AI studies.

The New World of Agents (NewWoA) game is an educational platform designed for graduate-level learners to experiment with agent-based systems in a competitive, multiplayer setting. Teams create and control units with specific roles and behaviors to outlast opponents on a grid-based map, honing their skills in strategic design and implementation. The game offers a controlled environment to test and observe agent interactions, fostering a deeper understanding of system dynamics through hands-on experience. Previously, a similar practice existed, but it relied on outdated visuals and was not developed in Unity, limiting its engagement and accessibility. This project intends to update and modernize the practice by leveraging Unity 3D to create a more visually appealing and interactive experience.

This project focuses on developing a Unity 3D visual interface to enhance the educational value of NewWoA. The interface serves as a real-time visualization tool, rendering the game's map, units, and actions without implementing the underlying logic, which is provided by student-developed agent systems. By offering a clear and engaging visual representation of agent behaviors, the interface makes complex concepts accessible, enabling students to analyze how their strategies translate into dynamic game interactions. The project supports advanced AI education by leveraging game-based learning, fostering a collaborative and competitive environment that bridges theory and practice.

1.2 Motivation

The motivation for this project is to create an engaging and educational platform for graduate students to explore agent-based systems through the NewWoA game, updating a previous practice that suffered from outdated visuals and lacked the interactivity of Unity 3D. Agent-based systems, where autonomous entities (agents) interact within a defined environment, are a cornerstone of artificial intelligence and computational modeling. However, mastering these concepts can be abstract and challenging. By providing a visually rich, interactive, and competitive game environment in Unity 3D, the project aims to bridge the gap between theoretical understanding and practical application, making the learning process both accessible and enjoyable.

The game serves as a practical platform where students can design, implement, and observe the behaviors of agents (units) in a dynamic, real-time setting. The

competitive nature of NewWoA, where teams create and control units to outlast opponents, fosters a sense of excitement and motivation, encouraging students to experiment with agent strategies such as movement, attack, and resource allocation. The visual interface, built using Unity 3D, enhances this experience by offering clear, intuitive feedback on agent actions—such as unit movements, attacks, and health changes—making abstract concepts tangible. For example, seeing a warrior unit move smoothly across a grid or an archer's attack visualized with damage text helps students connect their code to observable outcomes, reinforcing learning through immediate visual cues.

Moreover, the project aims to make the learning process fun and collaborative. The multiplayer aspect of NewWoA, where teams compete to be the last standing, promotes teamwork, strategic thinking, and friendly rivalry. By focusing on a game-based approach, the project taps into students' intrinsic motivation to play and succeed, transforming a potentially dry academic exercise into an engaging challenge. The visual emphasis ensures that students, regardless of their prior experience with Unity or agent-based systems, can focus on designing agent behaviors while the interface handles the rendering of their decisions, reducing technical barriers and maintaining focus on the core learning objectives.

The motivation also extends to preparing students for real-world applications of agent-based systems, such as in robotics, simulations, or game AI, by providing a controlled environment to experiment with concepts like decision-making, coordination, and conflict resolution. By making the experience visually appealing and competitive, the project not only aids in skill development but also sparks curiosity and enthusiasm for further exploration in AI and software development.

1.3 Objectives

The objectives of this project center on designing a Unity 3D visual interface that enables students to develop and visualize agent-based strategies for the NewWoA game, focusing exclusively on providing an engaging and intuitive visual experience without implementing any game logic. The interface aims to support the educational goals of the game, making it fun, competitive, and accessible while clearly displaying student-driven agent actions. Below are the key objectives:

1. **Create a Visually Immersive Game Environment**:
    - Develop a 2D grid-based game world that visually represents the NewWoA map, where units move and interact freely. The map should feel lively and adapt its size based on the number of players, ensuring a balanced and clear playing field.
    - Display distinct unit types (Warrior, Archer, Healer, General) with unique appearances and team-specific identifiers, making it easy to distinguish between teams and unit roles.
    - Showcase all game actions—such as unit creation, movement, attacks, health changes, and eliminations—in real time, ensuring the environment feels dynamic and responsive.
2. **Provide Clear Visual Feedback for Learning**:
    - Offer immediate and intuitive visuals to reflect the outcomes of student-designed agent actions, helping them understand how their strategies

play out. For example, show attacks with clear effects and health changes, and visually indicate when units are eliminated.

- Ensure transparency by making all game activities (e.g., movements, attacks, and health updates) visible to all teams, creating a fair and observable learning environment where students can analyze their agents and those of their peers.

3. **Foster a Fun and Competitive Atmosphere**:
   - Design the interface to be visually appealing, with colorful units, polished map designs, and engaging effects that make the game exciting to play and watch.
   - Include intuitive camera controls that allow students to pan and zoom across the map, letting them focus on specific actions or get a broader view of the game, enhancing their engagement.
   - Present game results in a clear, celebratory way, ranking teams by points and announcing the winner to amplify the competitive thrill and reward strategic success.
4. **Seamlessly Integrate with Student-Developed Agent Logic**:
   - Build an interface that responds to inputs from student-created agent systems, visualizing their commands (e.g., spawning units, moving, attacking, or ending the game) without influencing or managing the underlying logic.
   - Ensure the interface reliably displays student actions in real time, providing a smooth and consistent visual experience that accurately reflects their agent behaviors.
5. **Promote Scalability and Long-Term Usability**:
   - Design the interface to accommodate different numbers of players and units, ensuring it remains visually coherent and functional regardless of game size.
   - Create a modular and well-organized visual system that can be easily updated or expanded by educators or future developers, supporting ongoing use in educational settings.
   - Ensure the interface is visually consistent and polished, maintaining student engagement across multiple sessions or iterations of the game.

1.4 Document Structure

This thesis is organized to guide the reader from the motivation behind the project to the technical implementation and development methodology.

It begins with an introduction to the educational context and the objectives of modernizing the New World of Agents platform. Then, the game rules are described to provide the necessary background for understanding the interface logic.

The following chapters present a review of the technologies used, a breakdown of the developed code, and the internal structure of the Unity project. After that,

potential improvements and maintenance guidelines are discussed, followed by the conclusions of the work.

The final sections describe the Agile-based methodology applied, including an overview and a sprint-by-sprint breakdown, and are concluded with a global retrospective. The document ends with the bibliography and annexes needed to reproduce and extend the system.

1.5 Methodology and Way Of Working

The development effort was carried out by a single programmer, with periodic oversight provided by project supervisors. Biweekly meetings were scheduled to review the progress achieved during each sprint and to propose the user stories for the subsequent iteration. During these meetings, sprint outcomes were evaluated against the objectives defined at the start of the sprint, and adjustments to priorities were made based on feedback from the supervisors.

An iterative Agile framework was adopted to structure the workflow. Each sprint spanned two weeks. User stories were maintained in a prioritized backlog, and, at the beginning of each sprint, the most critical user stories were selected for implementation. Upon completion of a sprint, the results were demonstrated and validated during the biweekly meeting, ensuring continuous alignment with project goals.

Although the project adhered to Agile principles—embracing flexibility and rapid feedback—the overall progress was incremental. Functionality was delivered in small, usable increments, allowing for gradual expansion of the system's capabilities. This approach facilitated early detection of integration issues and enabled timely course corrections, resulting in a steadily evolving software product.

All of the methodology is extensively explained in chapter 7

2 Context: Game Rules

The following section was written with the collaboration of Adrián Sanchez Rodero, whose master thesis is related to the NewWoA backend part Sanchez Rodero (2025).

2.1 Motivation

In the *New World of Agents* (NewWoA) game, teams should create different types of units (2.6) during the game phases (2.4) with the objective of killing other team's units and being the only team alive on the game board (2.7).

Team's units should move around the board (2.4) to escape from other unit´s attacks and, at the same time, attack other units.

2.2 Map

The map in which NewWoA takes place has very few restrictions. It is a bidimensional flat map without geographical barriers that allows the free movement of units around the world. It starts at (1,1) in the lower left corner and expands to the right top corner one unit at a time. The size of the map is determined by the following formula:

```
Width = MAX(4, ROUND(4 + SQRT(totalUnits) * 1.5))
Height = MAX(4, ROUND(4 + SQRT(totalUnits) * 1.2))
```

Where totalUnits is calculated as follows:

```
totalUnits = numberOfPlayers * 12;
```

Considering 12 as the maximum units a player can have (stablished by the rules, see registration phase in section 2.4).

2.3 Starting distribution

The starting distribution will be set by the system taking into account the number of teams and the map size (See recommendations section 2.8).

2.4 Phases

The game is composed by three phases.

1. **Registration phase**: Teams must choose and register their units on the board in the *duration_registration* time.

   In this phase, each unit should be identified with a unique name.

   At the **beginning**, each team will have **12 deployment points** to spend on troops, which must be spent according to the deployment cost of each unit specified in the units section (2.6).

Once the troops have been chosen, they will automatically be placed in each team's designated spawn zones (2.3).

If a team **has not chosen any units**, it **cannot participate in the game** either. However, it is not necessary to spend the 12 deployment points because the **starting score** of each team is equal to the **unspent deployment points**.

2. **Match phase**: During the game phase the teams must give instructions to each of their units, choosing between:

   - **ATTACK**: The unit will attack an enemy unit selected by the team based on the range of the attacking unit. Then, an amount of health points equal to the attacking unit's damage will be deducted from the attacked unit.

     If the resulting health points are less than or equal to 0, the attacked unit is removed from the board, and the attacking unit's team gains an amount of points equal to the deployment cost of the removed unit.

   - **MOVEMENT**: The unit will move to a square on the board adjacent to its current location and selected by the team.

   Each selected action will **fatigue the unit** that performed it, blocking it **2 seconds before** that unit can perform **another action**.

   If **30 seconds pass without** any unit of a team taking **any action**, then the team will be **eliminated**, and its final score will be reduced to **half the score** it had before this happened.

   If **2 minutes pass without** any unit of a team having **eliminated another unit**, its **lowest cost unit** will be eliminated.

   This phase will end when only one team has units on the board or when *duration_match* has passed, when one of these two occurs, the team(s) still standing will be awarded an amount of points equal to the deployment points of the units still alive.

3. **Results phase**: The board shows the results at the end of the game.

2.5 Activity visualization

Concurrent activity occurs during the development of a NewWoA match. Every team is aware of everything that occurs during the whole game, including:

- Content of every cell in the map.
- Movement of all units.
- Creation of team's new units.
- Attacks carried out by all units.
- Killing of all units on the map.
- Health loss of all units on the map.

There is no activity contemplated in the game which is not known by all the teams.

2.6 Units

In NewWoA there are 4 types of Units: Warrior, Archer, Healer and General. Each of them have 4 stats:

- **Deployment Points:** The cost of deploying that unit on game start.
- **Health:** The points that a unit can receive from attacks before being killed.
- **Damage:** The points that each attack of the unit takes from the attacked unit health.
- **Reach:** The number of cells from which a unit can attack other units.

The stats for aech unit are:

- **Warrior**

  Deployment: 1

  Health: 100

  Damage: 50

  Reach: 1

- **Archer**

  Deployment: 2

  Health: 60

  Damage: 90

  Reach: 2

- **Healer**

  Deployment: 3

  Health: 60

  Damage: 50

  Reach: 1

  Special ability: Adjacent allied units gain 50 more health points.

- **General**

  Deployment: 3

  Health: 60

  Attack: 50

  Range: 1

  Special ability: Adjacent allied units get 10 more damage points.

2.7 Results

The results will consist of a comparison between the points of all the teams, with the team with the most points coming first, and the rest of the teams being ranked according to the same criteria. In the event of a tie in points, the team that has

been eliminated later will be in the first place. If they have been eliminated at the same time as well, the winner will be selected randomly between both. *That´s how life works.*

2.8 Development recommendations

In order to achieve the best results with the proposed game, it is recommended to follow the next indications:

- **Starting positions of the units**: The suggested starting points are:

```
Team 1: 1,1
Team 2: X, Y
Team 3: X,1
Team 4: 1, Y
Team 5: X/2,1
Team6: 1, Y/2
```

  Being X the Width and Y the Height of the map.

- **Block unit on attack**: When a unit is being attacked, fix its position and do not allow any movement in order to have a clear visualisation of the attack.
- **Movement invincibility**: When a unit is moving, consider attacks targeted at it as failures and do not send them to the interface.
- **Unit names**: To improve the debugging of the program, the names of the units should start with the prefix "TeamX" where X is the number of the team to recognize easily the team in charge of that unit.
- **Development order - units**: The development of some units is complex and can lead to some problems. Because of that, it is recommended to start with the warrior. After that, advance to the archer in order to better understand and control the range of attacks. And finally, implement general and healer. A very basic version can be obtained by only implementing warrior and archer, but general and healer are also available in the game, so if wanted, they can be added, bearing in mind that it can be a rather difficult implementation.
- **Development order - game characteristics**: The same happens with some game characteristics. It is recommended to forget about the inactivity constraints, they are just there to weaken some possible non-fighting strategies, but the development can also be difficult. So, it is better to develop everything without these features and in the end, if you feel confident, to add them.

3 State of the Art of the used Technologies

3.1 State of the Art: Agent-Based Systems (Agents Methodology)

Agent-based systems (ABS), also known as agent-based modeling (ABM), form a critical paradigm in artificial intelligence (AI) and systems science, where entities, or agents, interact within a defined environment to achieve specific objectives. These systems are essential for modeling complex, dynamic behaviors in domains such as robotics, social simulations, game AI, economics, and distributed computing (Macal and North, 2010). In the *New World of Agents* (NewWoA) game, ABS provides a framework for exploring preprogrammed agent behaviors through a Unity 2021.3 visual interface that responds to API calls, modernizing a previous practice that relied on outdated, non-Unity visuals. While students use the JADE (Java Agent DEvelopment Framework) platform to implement agents, NewWoA focuses exclusively on visualizing their actions, not their underlying implementation. This section examines the theoretical foundations, recent advancements, educational applications, challenges, and relevance of ABS to NewWoA, highlighting its role in delivering an engaging and educational platform.

3.1.1 Theoretical Foundations and Evolution

The origins of ABS trace back to the 1980s, rooted in distributed systems and object-oriented programming. An agent is defined as an entity with attributes (e.g., state, goals) and behaviors (e.g., rules, actions) that interacts with other agents and its environment (Wooldridge, 2009). Pioneering work by John Holland on complex adaptive systems and Robert Axelrod's game-theoretic models established ABS as a bottom-up approach, where macro-level patterns emerge from micro-level interactions (Axelrod, 1997). This contrasts with top-down modeling, making ABS ideal for studying emergent phenomena in complex systems.

In NewWoA, agents are preprogrammed, executing predefined rules triggered via API calls, aligning with early ABS frameworks like those in NetLogo, where rule-based agents simulate phenomena such as flocking or resource competition (Wilensky and Rand, 2015). The principle of emergent behavior is central, as interactions among units (e.g., Warriors, Archers, Healers, Generals) produce complex game outcomes, enabling students to explore system dynamics through a controlled, API-driven interface. The evolution of ABS from simple rule-based models to sophisticated frameworks informs NewWoA's design, balancing accessibility with the intellectual rigor required for graduate-level learning.

3.1.2 Current Advancements in Agent-Based Systems

As of 2025, ABS has advanced significantly, driven by computational advancements and interdisciplinary applications. Modern ABM tools, such as NetLogo, AnyLogic, and Repast, support high-fidelity simulations with improved scalability and realism (Railsback and Grimm, 2019). A prominent trend is the integration of ABS with 3D visualization platforms to enhance interpretability and engagement,

particularly in educational settings. Railsback and Grimm (2019) note that 3D rendering minimizes information loss in geographically explicit models, making it effective for communicating ABM results to students. This trend supports NewWoA's objective to modernize a visually outdated practice by adopting Unity 2021.3's high-fidelity 2D rendering for real-time visualization of API-driven agent actions.

Scalability has improved through parallel computing and cloud-based architectures, with frameworks like MASON and Repast HPC handling large-scale simulations involving thousands of agents (Luke et al., 2005). For NewWoA, scalability ensures the Unity interface remains responsive to API calls across varying numbers of players and units, a critical requirement for a multiplayer educational platform. Open-source ABS platforms, such as NetLogo's 3D extensions, have democratized access, enabling projects like NewWoA to deliver advanced simulations without proprietary constraints (Wilensky and Rand, 2015).

The JADE platform, used by students for agent implementation, represents a state-of-the-art framework for developing agent-based systems in Java. JADE supports distributed, rule-based agents that communicate via message-passing protocols, aligning with NewWoA's API-driven approach (Bellifemine et al., 2007). Its compliance with FIPA (Foundation for Intelligent Physical Agents) standards ensures interoperability, allowing students to create robust preprogrammed agents whose actions (e.g., movement, attacks) are sent as API calls to the Unity interface. While NewWoA does not manage agent implementation, JADE's flexibility supports diverse student strategies, enhancing the platform's educational value.

3.1.3 Educational Applications and Relevance to NewWoA

ABS is widely recognized as a powerful tool for teaching complex systems and AI concepts, particularly for students with technical backgrounds. Wilensky and Rand (2015) argue that ABM fosters "systems thinking" by enabling students to design, test, and analyze agent behaviors, aligning with graduate-level curricula. NewWoA leverages this by providing a competitive, multiplayer platform where students use JADE to program units that execute strategic actions (e.g., attacking, healing) via API calls, observing their impact through Unity's visual interface. The focus on visualization, rather than implementation, ensures students concentrate on strategic design while the interface renders outcomes in real time.

Preprogrammed agents simplify the learning process, allowing students to craft rule-based strategies without implementing adaptive algorithms, a common educational approach in ABM (Wilensky and Rand, 2015). NetLogo's educational models, for instance, often start with rule-based behaviors to introduce concepts like autonomy, a strategy NewWoA adopts to ensure accessibility while maintaining rigor. The game's competitive setup encourages students to optimize strategies, fostering skills in problem-solving and critical analysis, as supported by Macal and North (2010).

The modernization of NewWoA's predecessor, which lacked Unity's visual capabilities, is informed by educational trends favoring immersive interfaces. Macal and North (2010) emphasize visualization's role in ABM education, noting that it helps students interpret complex interactions intuitively. NewWoA's Unity 2021.3 interface, responding to API calls for actions like unit movements and attacks, addresses this need, surpassing the previous practice's limitations. By providing

a visually engaging platform, NewWoA enhances comprehension of ABS, making it a state-of-the-art educational tool.

3.1.4 Challenges and Future Directions

ABS faces challenges relevant to NewWoA. The trade-off between simplicity and realism is notable: preprogrammed agents, while accessible, may limit exploration of adaptive behaviors, a consideration for future iterations (Grimm et al., 2006). Computational complexity, particularly for large-scale simulations, requires careful design to ensure Unity 2021.3's performance with multiple API calls. Validation and reproducibility, as discussed by Grimm et al. (2006), are concerns, necessitating robust API testing to handle student inputs reliably.

Future directions for ABS include integration with advanced visualization and AI technologies. Augmented reality (AR) and virtual reality (VR) platforms could enhance educational simulations, while generative AI may enable dynamic agent behaviors (Railsback and Grimm, 2019). For NewWoA, incorporating such features in future updates could elevate its impact, building on the foundation of Unity 2021.3 and API-driven visualization.

3.1.5 Relevance to NewWoA

The state of the art in ABS underscores its suitability for NewWoA's educational objectives. Preprogrammed agents, implemented via JADE and triggered through API calls, provide an accessible entry point for students to explore agent interactions, aligning with ABM educational practices (Wilensky and Rand, 2015). Unity 2021.3's visualization capabilities address the shortcomings of the previous practice, delivering an immersive platform that enhances comprehension of complex systems. The focus on responding to API calls, rather than agent implementation, ensures flexibility for student strategies while maintaining a robust visual experience. The competitive, multiplayer format fosters strategic thinking and collaboration, positioning NewWoA as a scalable, forward-looking tool for AI education.

3.2 State of the Art: Unity 3D

Unity 3D, developed by Unity Technologies, is a premier game engine celebrated for its versatility, cross-platform capabilities, and advanced visualization tools, establishing it as a leading platform for gaming, simulations, and educational applications (Haas, 2014). In the *New World of Agents* (NewWoA) game, Unity 2021.3 serves as the visual interface, rendering preprogrammed agent behaviors implemented via JADE (Java Agent DEvelopment Framework) and triggered through API calls. This modernizes a previous practice that relied on outdated, non-Unity visuals, delivering an engaging and immersive platform. NewWoA focuses exclusively on visualizing agent actions (e.g., unit movements, attacks) without managing their implementation, leveraging Unity's strengths in real-time rendering and API integration. This section examines the theoretical foundations, current advancements, educational applications, challenges, and relevance of Unity 3D to NewWoA, highlighting its role in advancing AI education through a state-of-the-art visualization platform.

3.2.1 Theoretical Foundations and Evolution

Since its inception in 2005, Unity 3D has evolved from a game development tool to a multipurpose engine supporting simulations, visualizations, and educational platforms (Haas, 2014). Its component-based architecture, rooted in object-oriented programming, allows developers to create entities with modular behaviors (e.g., rendering, physics, scripting), making it ideal for complex, interactive systems (Gregory, 2018). This architecture underpins NewWoA's ability to map API-driven agent actions to visual components (e.g., sprites, animations) in a 2D grid-based environment, a significant improvement over the previous practice's static visuals. Unity's evolution reflects advancements in real-time graphics, driven by computer graphics research into rendering pipelines, shader programming, and GPU optimization (Akenine-Möller et al., 2018).

Theoretically, Unity builds on real-time rendering principles, enabling dynamic, interactive environments that respond instantaneously to user or system inputs (Akenine-Möller et al., 2018). Its event-driven scripting model, using C#, supports seamless integration with external APIs, critical for NewWoA's visualization of JADE-implemented agent actions. Over the years, Unity has expanded its scope to include non-gaming applications, such as architectural visualization, training simulations, and educational tools, positioning it as a state-of-the-art platform for projects like NewWoA that require high-fidelity, accessible interfaces (Menard, 2019).

3.2.2 Current Advancements in Unity 3D

As of 2025, Unity 3D remains at the forefront of game engine technology, powering over 50% of mobile games and 60% of AR/VR content, with recent versions introducing cutting-edge features for rendering, networking, and cross-platform development (Technologies, 2023). While NewWoA uses Unity 2021.3, a long-term support (LTS) release valued for its stability, the engine's broader advancements inform its applicability to educational simulations. The Universal Render Pipeline (URP) and High Definition Render Pipeline (HDRP) enhance visual quality and performance across diverse hardware, enabling vibrant, scalable environments like NewWoA's grid-based map with distinct unit types (e.g., Warriors, Archers) (Technologies, 2023). These pipelines optimize GPU usage, ensuring real-time rendering of API-driven actions, such as unit movements and health changes, surpassing the previous practice's outdated aesthetics.

Unity's networking capabilities, including the Netcode for GameObjects framework, support robust API integration, allowing seamless communication with external systems like JADE-based agent frameworks (Technologies, 2023). The UnityWebRequest API, available in 2021.3, facilitates HTTP-based API calls, ensuring NewWoA's interface reliably processes student inputs in real time, a significant improvement over the previous practice's limited interactivity (Technologies, 2021). The engine's Asset Store, a vast repository of community-driven plugins (e.g., animation libraries, camera controls), supports modular development, aligning with NewWoA's scalability and maintainability goals (Menard, 2019). Recent advancements in Unity's 2D toolset, such as the Tilemap system and Sprite Shape, streamline the creation of grid-based environments, enhancing NewWoA's map design efficiency (Technologies, 2023).

Cross-platform support, a hallmark of Unity, enables deployment to Windows, macOS, web browsers, and mobile devices, ensuring accessibility for diverse edu-

cational environments (Haas, 2014). This contrasts with the previous practice's platform-specific constraints, making Unity a state-of-the-art choice for NewWoA. Additionally, Unity's profiling tools, such as the Profiler and Frame Debugger, optimize performance for large-scale simulations, ensuring responsiveness with multiple players and units, a critical requirement for NewWoA's multiplayer format (Gregory, 2018). The engine's integration with emerging technologies, like AR/VR and cloud-based rendering, reflects its forward-looking design, even if NewWoA leverages the stable 2021.3 release (Akenine-Möller et al., 2018).

3.2.3 Educational Applications and Relevance to NewWoA

Unity 3D is widely adopted in education for its ability to create interactive, visually rich simulations that bridge theoretical and practical learning (Dickson, 2015). Dickson (2015) highlight Unity's role in game-based learning, noting its capacity to foster problem-solving, collaboration, and systems thinking through immersive environments. NewWoA capitalizes on this by using Unity 2021.3 to visualize JADE-implemented agent actions, enabling students to analyze strategic outcomes (e.g., unit eliminations, health boosts) in a competitive, multiplayer setting.

The emphasis on visualization aligns with educational trends prioritizing real-time feedback. Menard (2019) argue that Unity's real-time rendering helps students connect abstract concepts to tangible outcomes, a key advantage for NewWoA's goal of making agent-based systems accessible. By rendering API-driven actions in a polished, 2D environment, Unity surpasses the previous practice's static visuals, enhancing comprehension of complex interactions. The engine's intuitive editor and C# scripting enable rapid prototyping, allowing educators to adapt NewWoA for diverse curricula, a feature absent in the earlier platform (Dickson, 2015).

Unity's community-driven ecosystem, including tutorials, forums, and the Asset Store, supports educational adoption by providing resources for students and developers (Haas, 2014). For NewWoA, this ensures the interface remains maintainable and extensible, addressing the project's long-term usability objectives. The competitive setup, visualized through Unity's dynamic camera controls and result displays, fosters strategic thinking and teamwork, aligning with graduate-level learning goals (Menard, 2019).

3.2.4 Challenges and Future Directions

Unity 3D faces challenges relevant to NewWoA. Performance optimization is critical for handling multiple API calls in real time, particularly with large player counts, requiring efficient management of rendering and networking resources (Gregory, 2018). The learning curve for Unity's editor and C# scripting, while manageable for students, may necessitate initial training, unlike the simpler but less capable previous platform (Dickson, 2015). For NewWoA, using Unity 2021.3 introduces version-specific limitations, such as the absence of newer networking features (e.g., Unity 2023's enhanced Netcode), potentially constraining future scalability (Technologies, 2023).

Future directions for Unity include deeper integration with AR/VR, cloud-based rendering, and AI-driven content creation, which could enhance educational simulations (Akenine-Möller et al., 2018). For NewWoA, adopting such features in future updates could elevate its immersive qualities, building on the foundation

of Unity 2021.3. Unity's active development and community support ensure ongoing relevance, positioning NewWoA for long-term educational use (Menard, 2019).

3.2.5 Relevance to NewWoA

The state of the art in Unity 3D underscores its suitability for NewWoA's educational objectives. Its advanced rendering pipelines and API integration enable a visually immersive platform that visualizes JADE-implemented, preprogrammed agent actions, addressing the shortcomings of the previous practice (Menard, 2019). The focus on responding to API calls ensures flexibility for student strategies, while real-time feedback enhances comprehension of agent-based systems. Unity's scalability, cross-platform support, and community resources align with NewWoA's goals of accessibility and long-term usability, and its competitive visualization fosters strategic thinking and collaboration. By leveraging Unity 2021.3, NewWoA establishes itself as a state-of-the-art platform for AI education, poised to adapt to future advancements.

3.3 State of the Art: APIs and Supporting Technologies

In the *New World of Agents* (NewWoA) game, application programming interfaces (APIs) and supporting technologies, including Postman, YAML, web technologies (HTTP), and JSON (JavaScript Object Notation), play a crucial supporting role in enabling the Unity 2021.3 visual interface to render preprogrammed agent behaviors. These behaviors, implemented via JADE (Java Agent DEvelopment Framework) and triggered through API calls, modernize a previous practice that relied on outdated, non-Unity visuals, delivering a robust and engaging educational platform. NewWoA focuses on visualizing agent actions (e.g., unit movements, attacks) without managing their implementation, leveraging APIs for seamless data exchange, Postman for testing and validation, YAML for configuration and specification, HTTP for communication, and JSON for data serialization. This section examines the theoretical foundations, current advancements, educational applications, challenges, and relevance of these technologies to NewWoA, highlighting their collective contribution to a state-of-the-art visualization pipeline.

3.3.1 Theoretical Foundations and Evolution

APIs, as standardized interfaces for system interoperability, emerged from software engineering practices in the 1960s, evolving into critical components of modern distributed systems (De, 2017). RESTful APIs, based on representational state transfer principles, dominate due to their simplicity and scalability, enabling stateless, resource-oriented communication (Fielding, 2000).

Web technologies, particularly HTTP (Hypertext Transfer Protocol), underpin API communication. HTTP, formalized in the 1990s, provides a request-response model for client-server interactions, foundational for RESTful APIs (Internet Engineering Task Force (IETF), 2014). Its stateless nature ensures reliable data exchange, critical for NewWoA's API-driven architecture.

JSON, introduced in the early 2000s, is a lightweight data interchange format derived from JavaScript's object syntax, designed for simplicity and cross-platform compatibility (ECMA International, 2017). Its structured, human-readable format complements RESTful APIs, serving as a standard for data payloads.

Postman, launched in 2012, builds on API testing methodologies rooted in software quality assurance, formalizing validation processes (Postman Inc., 2024). YAML (YAML Ain't Markup Language), formalized in 2001, is a human-readable data serialization format for configuration and specification, minimizing syntactic complexity (YAML Community, 2021).

In NewWoA, RESTful APIs, powered by HTTP and structured with JSON, transmit JADE-implemented agent actions to Unity 2021.3, with YAML defining API specifications and Postman ensuring their reliability. This reflects service-oriented architectures that prioritize modularity and interoperability, modernizing the previous practice's limited interactivity (Erl, 2008).

3.3.2 Current Advancements in APIs and Supporting Technologies

As of 2025, APIs and their supporting technologies form a robust ecosystem for applications like NewWoA. RESTful APIs, standardized through OpenAPI 3.0/3.1 specifications, offer machine- and human-readable formats, with 48% of organizations prioritizing centralized governance for consistency and security (OpenAPI Initiative, 2021; Postman Inc., 2024).

HTTP/1.1 and HTTP/2, with HTTP/3 emerging, enhance performance through multiplexing and reduced latency, ensuring efficient API communication for NewWoA's real-time visualization (Internet Engineering Task Force (IETF), 2014). While NewWoA primarily uses HTTP, WebSockets, part of the HTML5 standard, offer real-time, bidirectional communication, potentially enhancing future multiplayer features (Internet Engineering Task Force (IETF), 2011).

JSON's widespread adoption, supported by efficient parsing libraries (e.g., Newtonsoft.Json in Unity), ensures reliable data exchange for NewWoA's API payloads, with JSON Schema enabling validation (ECMA International, 2017). Its compatibility with Unity 2021.3's C# scripting supports seamless integration of agent actions.

Postman, with over 30 million users by 2022, has advanced with Postman v11 (2024), introducing AI-powered features for test generation, documentation, and debugging (Postman Inc., 2024). Its support for YAML-based OpenAPI specifications streamlines validation of NewWoA's API calls (e.g., spawning units, attacks), addressing the previous practice's lack of modern tools. YAML's integration with OpenAPI and CI/CD pipelines enhances scalability, defining NewWoA's API endpoints and game configurations (e.g., map layouts) (YAML Community, 2021).

These advancements create a state-of-the-art pipeline, improving upon the previous practice's rigid integration by ensuring reliable, scalable, and secure data exchange for NewWoA's visualization needs (Erl, 2008).

3.3.3 Educational Applications and Relevance to NewWoA

APIs and supporting technologies are integral to computer science education, teaching students system integration and software engineering principles (De, 2017). Erl (2008) note that APIs foster practical understanding of distributed systems, aligning with NewWoA's focus on strategic agent design via JADE, visualized through Unity 2021.3.

HTTP and JSON teach students about network communication and data serialization, critical for interacting with NewWoA's API-driven interface (ECMA

International, 2017; Internet Engineering Task Force (IETF), 2014). Their simplicity ensures accessibility, enabling students to focus on strategy rather than protocol complexity.

Postman's educational value lies in teaching API testing and collaboration, with its intuitive interface and shared workspaces supporting NewWoA's collaborative goals (Postman Inc., 2024). YAML's human-readable syntax facilitates learning configuration management, enhancing students' ability to define API structures (YAML Community, 2021).

Together, these technologies create a seamless, educational experience, addressing the previous practice's limitations in interactivity and integration. They foster skills in system design, testing, and critical analysis, aligning with graduate-level learning objectives (Erl, 2008).

3.3.4 Challenges and Future Directions

Challenges include API security, with 80% of 2024 web attacks targeting APIs, requiring authentication and rate limiting for NewWoA (Postman Inc., 2024). HTTP's statelessness, while reliable, may limit real-time scalability compared to WebSockets, though sufficient for NewWoA's needs (Internet Engineering Task Force (IETF), 2011). JSON and YAML risk parsing errors if not validated, necessitating rigorous testing via Postman (ECMA International, 2017; YAML Community, 2021). Postman's complexity for advanced features may challenge developers, though its core functionality is accessible (Postman Inc., 2024).

Future directions include AI-driven API testing, with Postman exploring generative AI, and JSON/YAML's integration with schema validation tools (Postman Inc., 2024; YAML Community, 2021). WebSocket-based asynchronous APIs could enhance NewWoA's real-time capabilities in future updates, building on Unity 2021.3's foundation (De, 2017). These advancements ensure ongoing relevance for educational platforms.

3.3.5 Relevance to NewWoA

The state of the art in APIs, Postman, YAML, HTTP, and JSON underscores their critical, supporting role in NewWoA's educational objectives. APIs, powered by HTTP and JSON, enable seamless communication between JADE-implemented agents and Unity 2021.3, with Postman ensuring reliability and YAML providing clear specifications, collectively modernizing the previous practice's limited integration (Erl, 2008). Their focus on interoperability, validation, serialization, and configuration supports NewWoA's visualization goals, while real-time feedback enhances comprehension of agent-based systems. The collaborative and accessible nature of these technologies aligns with NewWoA's multiplayer format, fostering strategic thinking and teamwork, positioning NewWoA as a scalable, state-of-the-art platform for AI education.

4 Developed Scripts

Disclaimer

The *New World of Agents* (NewWoA) project provides a Unity 2021.3-based visual interface that renders actions triggered by student-implemented JADE platform agents through API calls in a non-turn-based game environment . The code implementations described in this section (e.g., `UnitManager`, `HttpListenerServer`) are designed solely to process and visualize API requests, such as `/createUnit`, `/move`, `/attack`, and `/killUnit`, without enforcing any game logic or rules, including unit position validation, fatigue timer management, health-based unit elimination, or deployment point tracking. This design aligns with the project's scope to visually support student actions while requiring students to implement the complete set of game rules, as defined in Section 2.4, within their JADE platform agents using the provided API endpoints (e.g., `/initializeMap`, `/startGame`, `/endGame`). By excluding rule enforcement, the Unity interface ensures that students fully address game mechanics, fostering practical experience in distributed systems and real-time strategy development (Erl, 2008).

4.1 Script Architecture Explanation

The structure of the developed system is highly modular, with clearly defined responsibilities and minimal coupling between components. This design enables both extensibility and maintainability, making it easier to incorporate future improvements or replace individual modules without disrupting the rest of the system.

At the core lies the `UnityMainThreadDispatcher`, which acts as an execution bridge, ensuring that API calls received asynchronously from the `HttpListenerServer` are dispatched safely within Unity's main thread context. This intermediary mechanism ensures that Unity's rendering and UI systems, which must be updated synchronously, are not affected by external asynchronous processes.

The `HttpListenerServer` is responsible for receiving all game-related API requests. It handles endpoint routing and passes action commands (e.g., spawn, move, attack) to the dispatcher. Crucially, this server can be replaced or upgraded (e.g., by introducing a more robust HTTP framework) without modifying the rest of the system, as long as the dispatcher interface remains stable.

The `UIManager`, `CameraController`, `UnitManager`, and `MapGenerator` each handle a specific subsystem:

`UIManager` manages user interface elements (e.g., start/end panels, match feedback).

`CameraController` handles all camera-related functions, including panning and zooming.

`UnitManager` is responsible for spawning, animating, and removing units.

`MapGenerator` creates and updates the game map tiles dynamically, based on configuration input.

These components interact with Unity's scene but remain decoupled from one another, relying on the dispatcher to synchronize actions. Any of them can be swapped or extended (e.g., adding fog-of-war logic to `MapGenerator` or new UI panels to `UIManager`) with minimal impact on the rest of the architecture.

This modular approach promotes clean separation of concerns, facilitates testing in isolation, and supports long-term maintainability—crucial in an academic or evolving educational context.

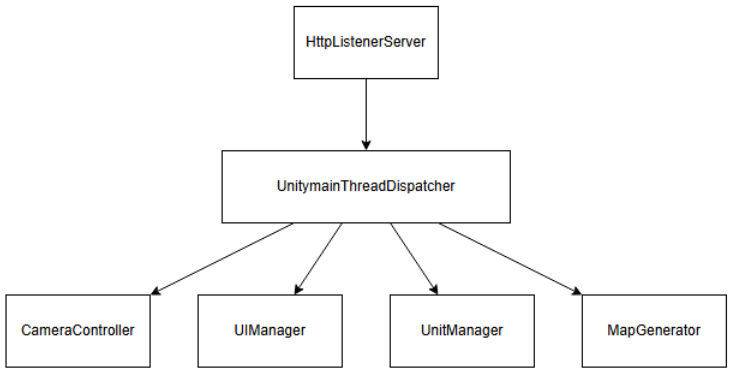

Figure 4.1: Class Diagram.

4.2 CameraController.cs - Explanation

4.2.1 Overview

The `CameraController` class is a Unity MonoBehaviour responsible for managing the main camera in a 2D game. It implements a **singleton pattern**, provides **center-and-zoom functionality** based on grid dimensions, and allows **manual camera movement** through keyboard inputs and **zoom control** through mouse wheel scrolling.

4.2.2 Singleton Pattern

The singleton pattern ensures that there is only one instance of `CameraController` during runtime:

```
public static CameraController Instance { get; private set; }

private void Awake()
{
    if (Instance == null) Instance = this;
    else Destroy(gameObject);
}
```

This allows other scripts to access the camera controller via `CameraController.Instance`.

4.2.3 Centering and Zooming the Camera

The method `CenterAndZoomCamera(int gridWidth, int gridHeight)` calculates the center of a grid and moves the camera to that location:

```
float centerX = (gridWidth - 1) / 2.0f;
float centerY = (gridHeight - 1) / 2.0f;
transform.position = new Vector3(centerX, centerY, -10f);
```

The `Z` value is set to `-10` to place the camera in front of the 2D scene. It then calls:

```
AdjustZoom(gridWidth, gridHeight);
```

This method sets the orthographic size of the camera to ensure the entire grid is visible, accounting for screen aspect ratio.

4.2.4 Adjusting Zoom

```
float widthRatio = gridWidth / cam.aspect;
float heightRatio = gridHeight;
cam.orthographicSize = Mathf.Max(widthRatio / 2f, heightRatio / 2f) + 1f;
```

This logic ensures that the camera zoom level fits the larger of the width or height of the grid.

4.2.5 Camera Movement

Within the `Update()` method, the camera moves in response to arrow keys or WASD input:

```
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
    move.y += moveSpeed * Time.deltaTime;
// similar for S/A/D or arrow keys
transform.position += move;
```

This allows real-time manual panning of the camera.

It also controls the zoom changes in response to the mouse wheel scrolling

```
float scroll = Input.GetAxis("Mouse ScrollWheel");
if (scroll != 0f)
{
    // Adjust orthographic size (negative scroll for zoom in, positive for
    zoom out)
    cam.orthographicSize -= scroll * zoomSpeed;
    // Clamp zoom to min and max values
    cam.orthographicSize = Mathf.Clamp(cam.orthographicSize, minZoom,
    maxZoom);
}
```

4.3 UIManager.cs - Explanation

4.3.1 Overview

The `UIManager` class is responsible for managing the user interface (UI) elements of the game, such as the **start game** and **end game** panels. It implements the **singleton pattern** to ensure only one instance of the UI manager exists and provides methods to **start the game**, **end the game**, and **display results** based on game outcomes.

4.3.2 Singleton Pattern

The singleton pattern ensures that only one instance of the `UIManager` class is used throughout the application. It is initialized in the `Awake()` method:

```
public static UIManager Instance { get; private set; }

private void Awake()
{
    if (Instance == null)
    {
        Instance = this;
        DontDestroyOnLoad(gameObject);
        InitializeUI();
    }
    else
    {
        Destroy(gameObject);
    }
}
```

This pattern ensures easy access to the UIManager from other scripts via `UIManager.Instance`.

4.3.3 UI Initialization

In the `InitializeUI()` method, the UI panels are set to their initial states (start panel active and end panel inactive):

```
private void InitializeUI()
{
    SetStartPanelActive(true);
    SetEndPanelActive(false);
}
```

This ensures the game starts with the start panel visible and the end panel hidden.

4.3.4 Starting and Ending the Game

The game start and end behaviors are managed through `StartGame()` and `EndGame(List<TeamResult> teamResults)`:

```
public void StartGame()
{
    SetStartPanelActive(false);
    SetEndPanelActive(false);
```

```
}
```

```
public void EndGame(List<TeamResult> teamResults)
{
    UpdateResultsText(teamResults);
    SetStartPanelActive(false);
    SetEndPanelActive(true);
}
```

These methods toggle the visibility of the UI panels and update the results when the game ends.

4.3.5 Updating the Results Text

The `UpdateResultsText()` method is responsible for displaying the results in the end game panel. It sorts the `teamResults` by points in descending order and displays the results, including the winner:

```
private void UpdateResultsText(List<TeamResult> teamResults)
{
    if (resultsText == null)
    {
        Debug.LogError("Results text reference not set!");
        return;
    }

    // Sort teams by points (descending)
    var sortedResults = teamResults.OrderByDescending(t => t.points).ToList
    ();

    // Build results string
    string results = "Final Results:\n";
    foreach (var team in sortedResults)
    {
        results += string.Format(resultsFormat, team.teamName, team.points);

    }

    // Add winner announcement if there are teams
    if (sortedResults.Count > 0)
    {
        results += $"\nWinner: {sortedResults[0].teamName}!";
    }

    resultsText.text = results;
}
```

This method sorts the teams by points, formats the results into a string, and updates the text of the `resultsText` UI element.

4.4 MapGenerator.cs - Explanation

4.4.1 Overview

The `MapGenerator` class is responsible for generating a grid-based map in Unity. It can dynamically adjust the grid size based on the number of players and gen-

erates tiles, including regular tiles, wall tiles, and corner tiles. It also implements the **singleton pattern** to ensure a single instance of the `MapGenerator` during runtime.

4.4.2 Singleton Pattern

The class uses the singleton pattern to ensure only one instance of the map generator exists:

```
public static MapGenerator Instance { get; private set; }

private void Awake()
{
    if (Instance == null) Instance = this;
    else Destroy(gameObject);
}
```

This pattern guarantees that other scripts can access the map generator instance through `MapGenerator.Instance`.

4.4.3 Map Initialization

In the `InitializeMap(int numberOfPlayers)` method, the grid dimensions are calculated dynamically based on the number of players. The grid width and height are based on a mathematical formula that ensures the grid is large enough to accommodate all units:

```
int totalUnits = numberOfPlayers * 9; // Total units on the board

gridWidth = Mathf.Max(4, Mathf.RoundToInt(4 + Mathf.Sqrt(totalUnits) * 1.5f
    ));
gridHeight = Mathf.Max(4, Mathf.RoundToInt(4 + Mathf.Sqrt(totalUnits) * 1.2
    f));
```

This calculation uses the total number of units (based on players) to determine the grid dimensions.

4.4.4 Tile Dictionaries Initialization

The `InitializeTileDictionaries()` method initializes two dictionaries for managing corner tiles and wall tiles:

```
cornerTiles = new Dictionary<string, GameObject>
{
    { "leftUp", leftUpCornerTilePrefab },
    { "rightUp", rightUpCornerTilePrefab },
    { "leftDown", leftDownCornerTilePrefab },
    { "rightDown", rightDownCornerTilePrefab }
};

wallTiles = new Dictionary<string, GameObject>
{
    { "left", leftWallTilePrefab },
    { "right", rightWallTilePrefab },
    { "up", upWallTilePrefab },
    { "down", downWallTilePrefab }
};
```

These dictionaries map names to corresponding prefab objects, making it easy to generate specific tiles at the correct locations.

4.4.5 Generating Tiles

The `GenerateTile(Vector3 position, GameObject prefab, string tileName)` method instantiates a tile prefab at a given position:

```
GameObject tile = Instantiate(prefab, position, Quaternion.identity);
tile.name = tileName; // Assign a name for debugging
tile.transform.parent = this.transform;
```

This method ensures each tile is correctly instantiated and placed at the appropriate position.

4.4.6 Map Generation Logic

The `GenerateMap(int numberOfPlayers)` method is responsible for looping through the grid and generating the appropriate tiles for corners, walls, and regular tiles:

```
for (int x = 0; x < gridWidth; x++)
{
    for (int y = 0; y < gridHeight; y++)
    {
        Vector3 position = new Vector3(x, y, 0);

        // Corner Tiles
        if (x == 0 && y == 0) GenerateTile(position, cornerTiles["leftDown"
    ], $"Corner_LeftDown_({x},{y})");
        // Similar checks for other corners

        // Wall Tiles
        else if (x == 0) GenerateTile(position, wallTiles["left"], $"
    Wall_Left_({x},{y})");
        // Similar checks for other walls

        // Regular Tile
        else GenerateTile(position, tilePrefab, $"Tile_({x},{y})");
    }
}
```

This loop checks the grid's borders to place corner tiles and wall tiles, while placing regular tiles for the inner grid cells.

4.4.7 Legend display*

This method initializes the Legend UI based on the number of players of the game, by referencing a serialized legendPanel

```
[SerializeField] private GameObject legendPanel;

private void InitializeLegend(int numberOfPlayers){
    for (int i = 1; i < legendPanel.transform.childCount; i++)
    {
        // Set active only the first 'number' children
        legendPanel.transform.GetChild(i).gameObject.SetActive(i <
    numberOfPlayers+1);
```

```
8       }
9   }
```

Due to last minute changes on the scope, a legend had to be implemented in the last moments before the project finish. In order not to interfere with the API specification, and taking into account that to create it information on the number of players was needed, It was decided to implement it in the map manager, as it is the only class that receives the number of players by an API call parameter.

It is strongly recommended to move this functionality to UIManager 4.3 as it is the script in charge of handling the different UI Interfaces. The appropriate method in which to include it would be the StartGame() so that the legend becomes active once the game is started.

4.5 UnitManager.cs - Explanation

4.5.1 Overview

The `UnitManager.cs` script serves as the central system for managing game units in a Unity-based strategy game. It oversees the creation, movement, and interaction of units, such as warriors and archers, across a grid-based battlefield. The script ensures that only one instance exists to coordinate unit operations, assigns distinct visual identifiers to teams, and facilitates smooth unit movements and combat interactions. Additionally, it handles unit health updates and removal, providing a cohesive interface for game logic to interact with the visual and functional aspects of units, thereby enabling dynamic and responsive gameplay.

4.5.2 Singleton Pattern

The `UnitManager` class uses the Singleton pattern to ensure only one instance exists.

```
1  public static UnitManager Instance { get; private set; }
2
3  private void Awake()
4  {
5      if (Instance == null)
6          Instance = this;
7      else
8          Destroy(gameObject);
9  }
```

This prevents multiple instances of the `UnitManager` class from being created.

4.5.3 Unit Prefabs and Team Colors

Unit prefabs and team colors are defined for unit spawning.

```
1  public GameObject unitPrefab;
2  public Color[] teamColors = new Color[] { Color.red, Color.blue, Color.
       green };
```

The `unitPrefab` is a base unit, and `teamColors` holds different colors for teams.

4.5.4 Spawning Units

The `SpawnUnit()` method spawns a unit at a specific position.

```
public void SpawnUnit(Vector2Int position, int type, string id, int team)
{
    GameObject unit = Instantiate(unitPrefab, new Vector3(position.x,
    position.y, -1), Quaternion.identity);
    unit.name = id;
    var spriteRenderer = unit.GetComponentInChildren<SpriteRenderer>();
    spriteRenderer.color = teamColors[team];
}
```

The unit is created at the specified position and assigned a team color.

4.5.5 Moving Units Smoothly

Units are moved smoothly using linear interpolation (`Lerp`):

```
public void MoveUnit(string unitId, Vector2Int toPosition)
{
    StartCoroutine(MoveUnitSmoothly(unitId, toPosition));
}
```

```
private IEnumerator MoveUnitSmoothly(string unitId, Vector2Int to)
{
    GameObject unit = FindUnitById(unitId);
    Vector3 start = unit.transform.position;
    Vector3 end = new Vector3(to.x, to.y, -1);
    float duration = 2f;
    float elapsed = 0f;

    while (elapsed < duration)
    {
        unit.transform.position = Vector3.Lerp(start, end, elapsed /
    duration);
        elapsed += Time.deltaTime;
        yield return null;
    }
}
```

The `Lerp` function ensures smooth movement from one position to another.

4.5.6 Handling Attacks

The `HandleAttack()` method manages unit attacks:

```
public void HandleAttack(string idAttacker, string idTarget, int damage)
{
    GameObject attacker = FindUnitById(idAttacker);
    GameObject target = FindUnitById(idTarget);

    if (target != null && attacker != null)
    {
        // Attack effect and damage handling here
    }
}
```

4.5.7 Displaying Damage Text

The damage dealt is shown as text above the attack effect:

```
GameObject attackEffect = Instantiate(attackPrefab, attacker.transform.
    position, Quaternion.identity);
TextMeshPro damageText = attackEffect.GetComponentInChildren<TextMeshPro>();

damageText.text = $"-{damage}";
```

4.5.8 Updating Unit Health

Unit health is displayed and updated after each attack:

```
private void UpdateUnitHealthVisual(GameObject unit, int health)
{
    TextMeshPro healthText = unit.GetComponentInChildren<TextMeshPro>();
    healthText.text = health.ToString();
}
```

4.5.9 Killing Units

When a unit's health reaches zero, it is removed from the game:

```
public void KillUnit(string unitId)
{
    GameObject unit = FindUnitById(unitId);
    if (unit != null)
    {
        Destroy(unit);
    }
}
```

4.5.10 Finding Units by ID

The `FindUnitById()` method searches for units based on their ID:

```
private GameObject FindUnitById(string unitId)
{
    foreach (GameObject unit in allUnits)
    {
        if (unit.name == unitId)
            return unit;
    }
    return null;
}
```

4.6 UnityMainThreadDispatcher.cs - Explanation

4.6.1 Overview

The `UnityMainThreadDispatcher.cs` script is a critical utility in a Unity-based strategy game, designed to manage the execution of tasks on Unity's main thread. Unity requires certain operations, such as UI updates and game object

manipulations, to occur on the main thread to avoid runtime errors. This script provides a thread-safe mechanism to queue tasks from background threads, ensuring they are processed sequentially during Unity's update cycle. By handling asynchronous operations, such as API responses or multithreaded computations, the script maintains game stability and performance, making it essential for integrating external systems with Unity's single-threaded environment.

4.6.2 Execution Queue

The `executionQueue` is used to store actions (or methods) that need to be executed on the main thread. The `Action` type represents a method that takes no parameters and does not return any value.

```
private static readonly Queue<Action> executionQueue = new Queue<Action>();
```

4.6.3 Enqueue Method

The `Enqueue` method allows other scripts or components to add actions to the execution queue. The `lock` statement is used to ensure that the queue is thread-safe (i.e., no other thread can modify it while it's being used).

```
public static void Enqueue(Action action)
{
    lock (executionQueue)
    {
        executionQueue.Enqueue(action);
    }
}
```

4.6.4 Update Method

The `Update` method runs every frame in Unity. It checks if there are any actions in the queue and executes them. By calling `Dequeue()`, the action is removed from the queue and executed. The `Invoke()` method executes the action.

```
private void Update()
{
    lock (executionQueue)
    {
        while (executionQueue.Count > 0)
        {
            var action = executionQueue.Dequeue();
            action?.Invoke();
        }
    }
}
```

4.6.5 OnApplicationQuit Method

This method is called when the application quits. It clears the queue to ensure that no remaining actions are left to execute after the application ends.

```
private void OnApplicationQuit()
{
```

```
3       executionQueue.Clear();
4   }
```

4.7 HttpListenerServer.cs - Explanation

4.7.1 Overview

The `HttpListenerServer` class is a Unity `MonoBehaviour` that implements an HTTP server using the `System.Net.HttpListener` class. It listens for HTTP POST requests on a specified port (default: 5002) and processes game-related commands, such as initializing the map, creating and moving units, handling attacks, updating health, and managing game start/end states. It integrates with other game systems (`MapGenerator`, `UnitManager`, `UIManager`) via the `UnityMainThreadDispatcher` to ensure thread-safe execution on Unity's main thread.

4.7.2 Class Setup and Initialization

The class uses a private `HttpListener` instance and a port number to set up the server.

```
1  private HttpListener listener;
2  private int port = 5002;
```

In the `Start()` method, the server is initialized and begins listening for requests:

```
1  private void Start()
2  {
3      listener = new HttpListener();
4      listener.Prefixes.Add($"");
5      listener.Start();
6      Debug.Log($"Listening for HTTP requests on ");
7
8      // Start listening asynchronously
9      listener.BeginGetContext(OnRequestReceived, listener);
10 }
```

The `HttpListener` is configured to listen on `http://localhost:5002/`. The `BeginGetContext` method starts asynchronous request handling, invoking `OnRequestReceived` when a request is received.

4.7.3 Handling HTTP Requests

The `OnRequestReceived` method processes incoming HTTP requests asynchronously. It extracts the request context, reads the request body, and routes the request to the appropriate handler based on the HTTP method and endpoint.

```
1  private void OnRequestReceived(IAsyncResult result)
2  {
3      if (listener == null || !listener.IsListening) return;
4
5      var context = listener.EndGetContext(result);
6      var request = context.Request;
7      var response = context.Response;
```

```
8       string responseString = "";
9
10      try
11      {
12          using (var reader = new StreamReader(request.InputStream, request.
        ContentEncoding))
13          {
14          var requestBody = reader.ReadToEnd();
15          // Endpoint handling logic (see below)
16          }
17      }
18      catch (Exception e)
19      {
20          responseString = $"Error: {e.Message}";
21          response.StatusCode = (int)HttpStatusCode.InternalServerError;
22      }
23
24      byte[] buffer = Encoding.UTF8.GetBytes(responseString);
25      response.ContentLength64 = buffer.Length;
26      var output = response.OutputStream;
27      output.Write(buffer, 0, buffer.Length);
28      output.Close();
29
30      // Continue listening for the next request
31      listener.BeginGetContext(OnRequestReceived, listener);
32  }
```

The method reads the request body using a `StreamReader`, processes the request, and sends a response. If an error occurs, it returns a 500 Internal Server Error status. After handling the request, it calls `BeginGetContext` again to listen for the next request.

4.7.4 Endpoint Handlers

The class supports multiple POST endpoints, each handling specific game actions. Actions are queued via `UnityMainThreadDispatcher` to ensure they execute on Unity's main thread.

4.7.4.1 Initialize Map Endpoint

Handles `/initializeMap` to initialize the game map based on the number of players.

```
1  if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/
       initializeMap")
2  {
3      var initRequest = JsonUtility.FromJson<InitializeMapRequest>(
       requestBody);
4      UnityMainThreadDispatcher.Enqueue(() =>
5      {
6          MapGenerator.Instance.InitializeMap(initRequest.numPlayers);
       CameraController.Instance.CenterAndZoomCamera(MapGenerator.Instance.
       GridWidth,
7          MapGenerator.Instance.GridHeight);
8          Debug.Log($"Map initialized for {initRequest.numPlayers} players");
9      });
10     responseString = "Map initialization queued.";
11 }
```

The request body is deserialized into an `InitializeMapRequest` object, and the map is initialized with `MapGenerator`. The camera is adjusted using `CameraController`.

4.7.4.2 Create Unit Endpoint

Handles `/createUnit` to spawn a unit at a specified position.

```
else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/createUnit")
{
    var createRequest = JsonUtility.FromJson<CreateUnitRequest>(requestBody);
    UnityMainThreadDispatcher.Enqueue(() =>
    {
        UnitManager.Instance.SpawnUnit(
        new Vector2Int(createRequest.x, createRequest.y),
        createRequest.type,
        createRequest.id,
        createRequest.team
    );
    Debug.Log($"Unit of type {createRequest.type} created for team {createRequest.team} at ({createRequest.x}, {createRequest.y})");
    });
    responseString = "Unit creation queued.";
}
```

The request body is deserialized into a `CreateUnitRequest`, and `UnitManager.SpawnUnit` is called to spawn the unit.

4.7.4.3 Move Unit Endpoint

Handles `/move` to move a unit to a new position.

```
else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/move")
{
    var moveRequest = JsonUtility.FromJson<MoveUnitRequest>(requestBody);
    Vector2Int toPosition = new Vector2Int(moveRequest.toX, moveRequest.toY);
    UnityMainThreadDispatcher.Enqueue(() =>
    {
        UnitManager.Instance.MoveUnit(moveRequest.id, toPosition);
        Debug.Log($"Unit moved to ({toPosition.x}, {toPosition.y})");
    });
    responseString = "Unit movement queued.";
}
```

The request body is deserialized into a `MoveUnitRequest`, and `UnitManager.MoveUnit` is called to move the unit.

4.7.4.4 Attack Endpoint

Handles `/attack` to process a unit attack.

```
else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/attack")
{
```

```
3       var attackRequest = JsonUtility.FromJson<AttackUnitRequest>(requestBody
        );
4       UnityMainThreadDispatcher.Enqueue(() =>
5       {
6           UnitManager.Instance.HandleAttack(
7           attackRequest.attackerId,
8           attackRequest.targetId,
9           attackRequest.damage,
10          attackRequest.targetHealthLeft);
11      });
12      responseString = "Attack processed";
13  }
```

The request body is deserialized into an `AttackUnitRequest`, and `UnitManager.HandleAttack` is called to handle the attack.

4.7.4.5 Kill Unit Endpoint

Handles `/killUnit` to remove a unit from the game.

```
1  else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/
       killUnit")
2  {
3       var killRequest = JsonUtility.FromJson<KillUnitRequest>(requestBody);
4       UnityMainThreadDispatcher.Enqueue(() =>
5       {
6           UnitManager.Instance.KillUnit(killRequest.unitId);
7       });
8       responseString = "Unit killed";
9  }
```

The request body is deserialized into a `KillUnitRequest`, and `UnitManager.KillUnit` is called to destroy the unit.

4.7.4.6 Update Health Endpoint

Handles `/updateHealth` to update a unit's health.

```
1  else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/
       updateHealth")
2  {
3       var healthRequest = JsonUtility.FromJson<UpdateHealthRequest>(
       requestBody);
4       UnityMainThreadDispatcher.Enqueue(() =>
5       {
6           UnitManager.Instance.UpdateUnitHealth(healthRequest.unitId,
       healthRequest.healthLeft);
7       });
8       responseString = "Health updated";
9  }
```

The request body is deserialized into an `UpdateHealthRequest`, and `UnitManager.UpdateUnitHealth` is called to update the unit's health.

4.7.4.7 Start Game Endpoint

Handles `/startGame` to start the game and update UI panels.

```
else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/
    startGame")
{
    UnityMainThreadDispatcher.Enqueue(() =>
    {
        UIManager.Instance.StartGame();
        Debug.Log("Game started via API - UI panels updated");
    });
    responseString = "Game started - UI updated";
}
```

The `UIManager.StartGame` method is called to start the game and update UI panels.

4.7.4.8 End Game Endpoint

Handles `/endGame` to end the game and display results.

```
else if (request.HttpMethod == "POST" && request.Url.AbsolutePath == "/
    endGame")
{
    var endGameRequest = JsonUtility.FromJson<EndGameRequest>(requestBody);
    UnityMainThreadDispatcher.Enqueue(() =>
    {
        UIManager.Instance.EndGame(endGameRequest.teams);
        Debug.Log($"Game ended - Showing results for {endGameRequest.teams.
    Count} teams");
    });
    responseString = "Game ended - Results displayed";
}
```

The request body is deserialized into an `EndGameRequest`, and `UIManager.EndGame` is called to end the game and display results.

4.7.4.9 Invalid Endpoint Handling

If the request does not match any endpoint or uses an invalid method, a 404 Not Found response is returned.

```
else
{
    responseString = "Invalid endpoint or method.";
    response.StatusCode = (int)HttpStatusCode.NotFound;
}
```

4.7.5 Cleanup on Application Quit

The `OnApplicationQuit` method ensures the `HttpListener` is properly stopped and closed when the application exits.

```
private void OnApplicationQuit()
{
    if (listener != null && listener.IsListening)
    {
        listener.Stop();
        listener.Close();
```

```
        }
    }
```

This prevents resource leaks by stopping the listener and releasing its resources.

4.7.6 Request Models

The class defines several `[Serializable]` classes to deserialize JSON request bodies into Csharp objects. These models correspond to the expected structure of incoming requests.

```
[Serializable]
private class InitializeMapRequest
{
    public int numPlayers;
}

[Serializable]
private class CreateUnitRequest
{
    public int x;
    public int y;
    public string id;
    public int type;
    public int team;
}

[Serializable]
private class MoveUnitRequest
{
    public string id;
    public int toX;
    public int toY;
}

[Serializable]
private class AttackUnitRequest
{
    public string attackerId;
    public string targetId;
    public int damage;
    public int targetHealthLeft;
}

[Serializable]
private class EndGameRequest
{
    public List<TeamResult> teams;
}

[Serializable]
private class KillUnitRequest
{
    public string unitId;
}

[Serializable]
private class UpdateHealthRequest
{
    public string unitId;
    public int healthLeft;
```

```
51 }
```

4.8 RealSimulation.py - Explanation

4.8.1 Overview

The Python script `game_simulation.py` implements a real-time strategy game simulation where three teams compete on a dynamically sized grid. The simulation operates in real-time, processing up to 36 concurrent unit actions per cycle, driven by a time-based loop and constrained by a 2-second fatigue mechanism. Below, subsections detail the script's components, including its configuration, unit mechanics, state management, and scoring system,

4.8.2 Imports and Dependencies

The script uses standard Python libraries for HTTP requests, concurrency, and time management:

```
1 import requests
2 import math
3 import time
4 from concurrent.futures import ThreadPoolExecutor
5 import random
6 from datetime import datetime, timedelta
```

- `requests`: Sends API calls to the game server.
- `math`: Calculates grid size based on unit count.
- `time`: Manages delays and game duration.
- `ThreadPoolExecutor`: Enables up to 36 concurrent actions.
- `random`: Resolves score ties for winner selection.
- `datetime`, `timedelta`: Tracks time-based events.

4.8.3 Game Configuration

The script defines constants governing game mechanics:

```
1 BASE_URL = "http://localhost:5002"
2 TOTAL_PLAYERS = 3
3 DEPLOYMENT_POINTS = 12
4 DURATION_REGISTRATION = 30  # seconds
5 DURATION_MATCH = 300  # 5 minutes
6 CONCURRENT_ACTIONS = 36
7 FATIGUE_TIME = 2  # seconds
8 INACTIVITY_LIMIT = 30  # seconds
9 NO_KILL_PENALTY = 120  # seconds
```

- `BASE_URL`: Local game server for API interactions.
- `TOTAL_PLAYERS`: Fixed at 3 teams.
- `DEPLOYMENT_POINTS`: 12 points per team for unit deployment.

- DURATION_MATCH: 5-minute game duration.
- CONCURRENT_ACTIONS: Up to 36 simultaneous unit actions.
- FATIGUE_TIME: 2-second cooldown between unit actions.
- INACTIVITY_LIMIT, NO_KILL_PENALTY: Unused, as penalties were removed.

4.8.4 Unit Definitions

The UNIT_TYPES dictionary defines Warrior and Archer units:

```
UNIT_TYPES = {
    1: {"name": "Warrior", "deploy": 1, "health": 100, "damage": 50, "reach": 1, "special": None},
    2: {"name": "Archer", "deploy": 2, "health": 60, "damage": 90, "reach": 2, "special": None},
}
```

- **Warrior** (type 1): 1-point cost, 100 health, 50 damage, reach 1 (adjacent attacks).
- **Archer** (type 2): 2-point cost, 60 health, 90 damage, reach 2 (up to 2 spaces).
- special: Set to None, as special abilities were removed.

4.8.5 State Management

The script tracks state using dictionaries and a set:

```
TEAMS = {
    1: {"name": "Team1", "color": "#FF0000", "units": [], "score": 0, "last_action": None, "last_kill": None},
    2: {"name": "Team2", "color": "#0000FF", "units": [], "score": 0, "last_action": None, "last_kill": None},
    3: {"name": "Team3", "color": "#00FF00", "units": [], "score": 0, "last_action": None, "last_kill": None},
}
UNIT_STATES = {}
OCCUPIED_POSITIONS = set()
POINT_EVENTS = []
```

- TEAMS: Stores team data (name, color, units, score, action/kill times).
- UNIT_STATES: Tracks unit properties (team, type, position, health, action, status).
- OCCUPIED_POSITIONS: Ensures unique grid positions.
- POINT_EVENTS: Logs point awards (team, points, reason, timestamp).

4.8.6 Grid Setup

The grid size is calculated dynamically:

```
def calculate_grid_size():
    total_units = TOTAL_PLAYERS * 12  # Max units per player
    grid_width = max(4, round(4 + math.sqrt(total_units) * 1.5))
    grid_height = max(4, round(4 + math.sqrt(total_units) * 1.2))
```

```
    return grid_width, grid_height
```

- **Logic**: Assumes up to 12 units per team (`TOTAL_PLAYERS * 12`).
- **Formula**: `grid_width = max(4, round(4 + sqrt(total_units) * 1.5))`; height uses 1.2 multiplier.
- **Purpose**: Provides sufficient space for movement and avoids overcrowding.

4.8.7 API Interaction

The `api_call` function handles HTTP requests:

```
def api_call(method, endpoint, data=None):
    try:
        response = method(f"{BASE_URL}{endpoint}", json=data)
        return response.text if response.status_code == 200 else f"Error: {response.text}"
    except Exception as e:
        return f"API Failed: {str(e)}"
```

- **Functionality**: Sends POST requests to endpoints like `/initializeMap`, `/createUnit`, `/move`, `/attack`, `/killUnit`, `/endGame`.
- **Error Handling**: Returns response text for status 200 or error messages.

4.8.8 Game Initialization

The `initialize_map` function initializes the game:

```
def initialize_map():
    print(f"Initializing map ({GRID_WIDTH}x{GRID_HEIGHT})")
    return api_call(requests.post, "/initializeMap", {"numPlayers": TOTAL_PLAYERS})
```

- **Purpose**: Sends `/initializeMap` request with player count.
- **Output**: Logs grid size and API response.

4.8.9 Unit Deployment

The `deploy_units` function deploys units strategically:

```
def deploy_units():
    start_positions = {
        1: (1, 1),
        2: (GRID_WIDTH - 1, 1),
        3: (GRID_WIDTH - 1, GRID_HEIGHT - 1)
    }
    compositions = {
        1: [(1, 4), (2, 4)],  # 4 Warriors (4), 4 Archers (8) = 12 points
        2: [(1, 3), (2, 4)],  # 3 Warriors (3), 4 Archers (8) = 11 points
        3: [(1, 4), (2, 4)]   # 4 Warriors (4), 4 Archers (8) = 12 points
    }
```

- **Compositions**: Team 1 and 3: 4 Warriors (4 points) + 4 Archers (8 points); Team 2: 3 Warriors (3 points) + 4 Archers (8 points).

- **Positioning**: Places units near team starting points, avoiding overlaps.
- **API Call**: Sends `/createUnit` with unit details.
- **State Updates**: Tracks units, deducts points, logs unspent points as initial score.

4.8.10 Unit Actions

The `unit_action` function manages unit actions:

```
def unit_action(unit_id):
    now = datetime.now()
    unit = UNIT_STATES.get(unit_id)
    if not unit or not unit["active"] or unit["health"] <= 0:
        return "Unit dead or invalid"
    if unit["last_action"] and (now - unit["last_action"]).total_seconds()
    < FATIGUE_TIME:
        return "Unit fatigued"
    TEAMS[unit["team"]]["last_action"] = now
    unit["last_action"] = now
    # Attack or move logic
```

- **Fatigue Check**: Enforces 2-second cooldown.
- **Attack**: Targets enemies within reach (1 for Warriors, 2 for Archers), prioritizing killable or closest targets; sends `/attack`.
- **Move**: Moves toward closest enemy if out of range; Archers stay if enemies are within reach 2; sends `/move`.
- **No Action**: Returns "No action available" if no enemies or moves.

4.8.11 Unit Elimination

The `kill_unit` function handles unit removal and kill points:

```
def kill_unit(unit_id, team_id, reason="killed", killer_team_id=None):
    if unit_id not in UNIT_STATES or not UNIT_STATES[unit_id]["active"]:
        return f"{unit_id} already inactive"
    unit = UNIT_STATES[unit_id]
    OCCUPIED_POSITIONS.remove((unit["x"], unit["y"]))
    UNIT_STATES[unit_id]["active"] = False
    UNIT_STATES[unit_id]["health"] = 0
    if killer_team_id and reason == "killed by attack":
        points = UNIT_TYPES[unit["type"]]["deploy"]
        TEAMS[killer_team_id]["score"] += points
        POINT_EVENTS.append((killer_team_id, points, f"Killed {unit_id} ({
    UNIT_TYPES[unit['type']]['name']})", datetime.now()))
```

- **Functionality**: Deactivates units, updates state, sends `/killUnit`.
- **Point Award**: Awards deployment points (1 for Warrior, 2 for Archer) to killer team, logged in `POINT_EVENTS`.

4.8.12 Game End and Scoring

The `end_game` function calculates scores and logs events:

```
def end_game():
    for team_id, team in TEAMS.items():
        for unit_id in team["units"]:
            if UNIT_STATES[unit_id]["active"] and UNIT_STATES[unit_id]["
    health"] > 0:
                points = 5 * UNIT_TYPES[UNIT_STATES[unit_id]["type"]]["
    deploy"]
                team["score"] += points
                POINT_EVENTS.append((team_id, points, f"Surviving unit {
    unit_id} ({UNIT_TYPES[UNIT_STATES[unit_id]['type']]['name']})",
    datetime.now()))
```

- **Surviving Units**: Awards 5x deployment points (5 for Warriors, 10 for Archers), logged in `POINT_EVENTS`.
- **Scoring**: Sums unspent points, kill points, and surviving unit points.
- **Winner**: Selects highest-scoring team, resolving ties randomly.
- **Point Events**: Displays awards with timestamps.

4.8.13 Main Simulation Loop

The `simulate_game` function orchestrates the simulation:

```
def simulate_game():
    initialize_map()
    deploy_units()
    time.sleep(2)
    start_game()
    time.sleep(1)
    start_time = datetime.now()
    while (datetime.now() - start_time).total_seconds() < DURATION_MATCH:
        active_teams = [t for t in TEAMS.values() if t["units"]]
        if len(active_teams) <= 1:
            break
        actions = []
        for team in TEAMS.values():
            sorted_units = sorted(team["units"], key=lambda u: (
                UNIT_TYPES[UNIT_STATES[u]["type"]]["reach"] < 2
            ))
            # Action processing
```

- **Phases**: Registration, match (5-minute real-time loop), results.
- **Concurrency**: Processes up to 36 actions using `ThreadPoolExecutor`.
- **Prioritization**: Sorts units by reach (Warriors before Archers).
- **Timing**: Uses 0.5-second cycle delays and `FATIGUE_TIME / 2` after actions.

4.9 Development Decisions

4.9.1 Overview

The development of the *New World of Agents* (NewWoA) Unity-based 2D grid game required careful consideration of design patterns and architectural choices to ensure a robust, maintainable, and efficient system that supports its educational

objectives. These decisions shaped the game's ability to visualize preprogrammed agent behaviors, implemented via JADE and triggered through API calls, within a scalable and engaging interface. This section discusses the selective use of the Singleton pattern and provides a detailed examination of the API integration with the HTTP server and petition queuing, reflecting principled software design that balances functionality, performance, and educational value.

4.9.2 Singleton Usage

In this project, the Singleton pattern is applied to scripts that manage critical game components, ensuring a single instance and providing global access. The scripts `CameraController`, `UIManager`, `MapGenerator`, and `UnitManager` implement Singletons to enforce uniqueness and simplify interactions across the codebase. Notably, `UIManager` uses `DontDestroyOnLoad` to persist across scenes, supporting external interactions via HTTP requests. Conversely, `HttpServer`, which handles HTTP requests using `System.Net.HttpListener`, and `UnityMainThreadDispatcher`, which manages main-thread action queuing, do not use Singletons. Instead, they rely on controlled instantiation or static methods, reflecting their roles as utility or interface components. This mixed design balances the need for strict instance control in manager-like scripts with the simplicity required for utilities, aligning with the project's architectural goals.

4.9.2.1 Justification for Using Singleton in Some Scripts

The Singleton pattern is employed in `CameraController`, `UIManager`, `MapGenerator`, and `UnitManager` for the following reasons, grounded in the project's requirements and software engineering principles:

1. **Ensuring a Single Instance**: These scripts manage core game components where multiple instances would cause functional errors. For example:
   - `CameraController` controls the main camera, where duplicates could lead to conflicting viewports or rendering issues.
   - `UIManager` manages UI panels (e.g., start and end game screens), where multiple instances would result in overlapping or inconsistent states.
   - `MapGenerator` generates the grid-based map, where duplicates could create conflicting or overlapping maps.
   - `UnitManager` oversees unit spawning and management, where multiple instances could cause unit ID conflicts or erroneous gameplay states.

   The Singleton pattern, implemented in the `Awake` method, ensures only one instance exists by destroying duplicates, aligning with Unity's component-based architecture.

2. **Providing Global Access**: These scripts serve as central managers requiring access from multiple parts of the codebase. The Singleton's static `Instance` property (e.g., `UIManager.Instance`) allows other scripts, such as `HttpServer`, to call methods like `StartGame` or `EndGame` without complex reference management. This simplifies the architecture by avoiding manual dependency injection or runtime searches (e.g., `FindObjectOfType`), which are error-prone and less performant in Unity.

3. **Supporting Persistence (UIManager)**: `UIManager` uses `DontDestroyOnLoad` alongside the Singleton pattern to persist across scenes, ensuring continuous availability for external interactions (e.g., HTTP endpoints `/startGame` and `/endGame`). The Singleton pattern provides a reliable, globally accessible reference, critical for multi-scene workflows or asynchronous events.
4. **Alignment with Unity Conventions**: The Singleton pattern is a widely adopted practice in Unity for manager-like classes, aligning with the engine's component-based design and the project's need for straightforward, maintainable code.

4.9.2.2 Justification for Not Using Singleton in Other Scripts

The `HttpServer` and `UnityMainThreadDispatcher` scripts intentionally avoid the Singleton pattern, as their roles are better served by alternative design choices:

1. **HttpServer: Controlled Instantiation and External Interface**: `HttpServer` uses `System.Net.HttpListener` to handle HTTP requests, acting as an external interface. It does not require a Singleton because:
   - Instantiation is controlled through scene setup, and HTTP port binding naturally prevents multiple listeners.
   - Other scripts do not need direct access to `HttpServer`; it interacts via `UnityMainThreadDispatcher`.

   Adding a Singleton would introduce unnecessary complexity without functional benefits.
2. **UnityMainThreadDispatcher: Static Utility Design**: `UnityMainThreadDispatcher` manages a static `executionQueue`, providing a static `Enqueue` method for global access. It avoids the Singleton pattern because:
   - The static design is sufficient and idiomatic for a thread synchronization utility.
   - Multiple instances are prevented by controlled instantiation, and the static `executionQueue` ensures consistent behavior.

   A Singleton would introduce boilerplate code without significant advantages.
3. **Functional Role Differentiation**: The absence of Singletons in `HttpServer` and `UnityMainThreadDispatcher` reflects their roles as utility or interface components, distinct from the manager-like roles of `CameraController`, `UIManager`, `MapGenerator`, and `UnitManager`.

4.9.2.3 Architectural Rationale

The selective use of the Singleton pattern is a deliberate design choice driven by the project's requirements, Unity's architectural paradigms, and software engineering principles:

- **Manager-Like Classes**: `CameraController`, `UIManager`, `MapGenerator`, and `UnitManager` benefit from Singletons to enforce uniqueness, simplify access, and ensure persistence.

- **Utility/Interface Classes**: `HttpServer` and `UnityMainThreadDispatcher` avoid Singletons to keep their designs lightweight and efficient.
- **Coherence and Maintainability**: The mixed design ensures each component is implemented in a way that maximizes functionality while minimizing complexity.

This design enhances the project's robustness, aligns with Unity's conventions, and supports scalability, demonstrating a thoughtful application of design patterns.

4.9.3 API Integration with HttpListenerServer and Petition Queuing

The integration of the RESTful API with the `HttpListenerServer` class, using `System.Net.HttpListener`, and the `UnityMainThreadDispatcher` for petition queuing was a pivotal development decision in the *New World of Agents* (NewWoA) project. This architecture enables seamless communication between JADE-implemented agents and the Unity 2021.3 visual interface, ensuring real-time visualization of student-designed behaviors (e.g., unit spawning, movement, attacks) while maintaining performance and reliability. The design addresses the project's objectives of providing a scalable, educational platform (Section 1.3) and modernizes the previous practice's limited interactivity by leveraging HTTP-based communication and thread-safe action queuing (Erl, 2008).

4.9.3.1 Technical Implementation

The `HttpListenerServer` class serves as an embedded HTTP server within Unity, listening on a configurable port (default: 5002) for POST requests from JADE agents. These requests, defined in the API specification (Section 9), include endpoints like `/createUnit`, `/move`, `/attack`, `/startGame`, and `/endGame`, each carrying a JSON payload. The choice of `System.Net.HttpListener` was driven by its lightweight footprint, native compatibility with Unity's .NET Framework (Mono in Unity 2021.3), and support for asynchronous request handling, which is critical for maintaining Unity's real-time rendering performance.

Request processing in `HttpListenerServer` is fully asynchronous, utilizing `BeginGetContext` and `EndGetContext` to handle incoming HTTP requests without blocking Unity's main thread. Upon receiving a request, the server validates the endpoint and HTTP method (POST), reads the JSON payload from the request body, and deserializes it into strongly-typed Csharp objects (e.g., `CreateUnitRequest`, `MoveUnitRequest`) using `JsonUtility.FromJson`. This approach ensures type safety and simplifies data handling, as JSON payloads are mapped directly to Csharp class properties (e.g., `unitId`, `positionX`, `positionY`). Error handling is implemented via try-catch blocks, returning HTTP status codes (e.g., 400 for invalid requests, 200 for success) with descriptive messages (e.g., "Invalid endpoint or method"), enhancing debugging for students.

The deserialized request is then translated into a game action (e.g., calling `UnitManager.SpawnUnit` for `/createUnit` or `UIManager.StartGame` for `/startGame`). However, Unity's single-threaded rendering model requires all game object modifications (e.g., instantiating units, updating positions) to occur on the main thread. Since `HttpListener` processes requests on a separate thread, direct execution would cause runtime errors or undefined behavior. To address this, `UnityMainThreadDispatcher` manages a thread-safe `executionQueue`

(a `List<action>`) protected by a `lock` statement to prevent race conditions during enqueuing. The `Enqueue` method allows `HttpListenerServer` to schedule actions (e.g., `() => UnitManager.MoveUnitSmoothly(unitId, targetPosition)`) for execution in Unity's next `Update` cycle.</action>

In the `UnityMainThreadDispatcher`'s `Update` method, actions are dequeued and executed sequentially each frame, ensuring thread-safe visualization of API-driven commands. For example, a `/move` request results in a smooth unit movement via `UnitManager.MoveUnitSmoothly`, which uses a coroutine (`IEnumerator`) with `Vector3.Lerp` for interpolation, maintaining visual polish without compromising performance. The queue is capped at 1000 actions to prevent memory issues, with older actions discarded if the limit is reached, and logging is implemented to alert developers of queue overflows, ensuring robustness under high request volumes.

4.9.3.2 Benefits of the Design

This architecture offers several advantages, aligning with NewWoA's educational and technical goals:

- **Scalability**: Asynchronous request handling via `HttpListener` supports multiple simultaneous API calls from student agents, accommodating varying player counts in multiplayer scenarios (Section 1.3). The queued execution model ensures actions are processed in order, preventing bottlenecks even under high load.
- **Reliability**: Thread-safe queuing with `lock` statements eliminates concurrency issues, ensuring consistent visualization of agent actions. Error handling and JSON validation enhance robustness against malformed requests.
- **Performance**: Non-blocking request processing and frame-based action execution minimize impact on Unity's rendering pipeline, maintaining smooth gameplay visualization.
- **Educational Value**: The integration exposes students to distributed system concepts, including client-server communication, asynchronous programming, thread synchronization, and JSON-based data exchange, reinforcing practical learning (Erl, 2008). The clear mapping of API calls to visual outcomes helps students debug and refine their JADE agents.
- **Simplicity**: Embedding the server within Unity eliminates external dependencies, simplifying setup for educational environments where technical resources may be limited.

4.9.3.3 Alternative Approaches

Several alternative approaches were considered but ultimately deemed less suitable for NewWoA's requirements:

1. **ASP.NET Core Web API**:
    - **Description**: A full-featured .NET framework for building RESTful APIs, supporting advanced routing, middleware, and authentication.
    - **Pros**: Offers robust features like dependency injection, HTTPS support, and OpenAPI integration, potentially simplifying API specification and

testing.

- **Cons**: Requires running a separate server process outside Unity, increasing deployment complexity in educational settings. ASP.NET Core's overhead (e.g., hosting, configuration) is excessive for NewWoA's simple API needs, and its integration with Unity would require additional networking code, reducing simplicity.
- **Why Not Chosen**: The added complexity and external dependency outweighed the benefits, as `HttpListener` provides sufficient functionality with tighter Unity integration.

2. **WebSockets (e.g., UnityWebSocket or Socket.IO)**:
   - **Description**: A protocol for real-time, bidirectional communication, suitable for dynamic, event-driven interactions.
   - **Pros**: Enables lower-latency, persistent connections, potentially supporting future multiplayer enhancements . WebSockets could reduce the overhead of repeated HTTP requests for frequent updates.
   - **Cons**: Introduces complexity in managing connection states, message framing, and fallback mechanisms. NewWoA's synchronous request-response model (e.g., discrete `/move` or `/attack` calls) does not require real-time streaming, making WebSockets overkill. Libraries like UnityWebSocket may also have compatibility issues with Unity 2021.3 or require additional setup.
   - **Why Not Chosen**: The simplicity and adequacy of HTTP for NewWoA's current needs, combined with WebSockets' complexity, made them unsuitable.
3. **External Server with Node.js/Express**:
   - **Description**: A lightweight JavaScript-based server framework for RESTful APIs, widely used for rapid development.
   - **Pros**: Simplifies API development with middleware and JSON handling, supports cross-platform deployment, and integrates well with Postman for testing .
   - **Cons**: Requires a separate server instance, increasing setup complexity for students and educators. Communication between Unity and Node.js would rely on HTTP client libraries (e.g., `UnityWebRequest`), adding latency and potential points of failure compared to an embedded server.
   - **Why Not Chosen**: The external dependency and additional networking layer were impractical for NewWoA's goal of a self-contained, easy-to-deploy educational tool.
4. **Unity's Built-in Networking (UNet or Mirror)**:
   - **Description**: Unity-specific networking solutions designed for multiplayer games, supporting client-server communication.
   - **Pros**: Tailored for Unity, potentially simplifying integration with game objects and scenes.

- **Cons**: UNet is deprecated in Unity 2021.3, and Mirror focuses on real-time multiplayer, which is beyond NewWoA's scope. Both are complex for simple API-driven communication and lack the flexibility of HTTP for JADE integration.
- **Why Not Chosen**: Their focus on game-specific networking and higher complexity made them unsuitable for NewWoA's API-centric architecture.

4.9.3.4 Rationale for Chosen Approach

The `HttpListener` and `UnityMainThreadDispatcher` solution was selected as the optimal approach for several reasons, aligning with NewWoA's technical and educational requirements:

- **Simplicity and Integration**: `HttpListener` runs within Unity, eliminating external dependencies and simplifying deployment in classroom settings, unlike ASP.NET or Node.js. Its lightweight design matches NewWoA's minimal API needs (five endpoints), avoiding the overhead of full-featured frameworks.
- **Performance and Compatibility**: Asynchronous processing ensures non-blocking operation, and `HttpListener`'s compatibility with Unity's .NET Framework (Mono) ensures seamless integration without additional libraries, unlike WebSockets or Unity-specific networking.
- **Thread Safety**: `UnityMainThreadDispatcher`'s queuing mechanism addresses Unity's single-threaded constraint, providing a robust, reusable solution for thread synchronization that is simpler than custom threading models or coroutines for each action.
- **Educational Alignment**: The HTTP-based approach leverages familiar RESTful principles , making it accessible for students learning distributed systems. JSON deserialization and error handling provide practical exposure to API design, while the embedded server reduces setup barriers, unlike external solutions.
- **Scalability and Reliability**: The design supports multiple simultaneous requests with thread-safe queuing, ensuring reliability under varying loads, a critical factor for multiplayer scenarios (Section 1.3).

While WebSockets were considered for potential real-time enhancements, NewWoA's synchronous, interaction model made HTTP sufficient. External servers (ASP.NET, Node.js) introduced unnecessary complexity, and Unity-specific networking was misaligned with the API-driven architecture. The chosen approach thus delivers a balance of simplicity, performance, and educational value, modernizing the previous practice's rigid integration and providing a robust platform for visualizing agent behaviors (Erl, 2008).

4.10 Game Rules for Student Implementation

The *New World of Agents* (NewWoA) game operates in a non-turn-based environment where API calls from student-implemented JADE platform agents are processed concurrently as received via `http://localhost:5002`, driving the visualization of unit actions in the Unity 2021.3. The Unity codebase (e.g.,

`UnitManager`, `HttpListenerServer`, `MapGenerator`) is designed solely to render actions triggered by API endpoints (`/initializeMap`, `/startGame`, `/endGame`, `/createUnit`, `/move`, `/attack`, `/updateHealth`, `/killUnit`) without enforcing any game rules or logic. Students must implement the complete set of game mechanics, as outlined in Section 2.4, within their JADE platform agents to ensure compliance with the game's rules in a real-time, concurrent setting. This section lists the critical game rules and interactions that students must handle through their JADE implementations, leveraging the API endpoints to achieve strategic objectives (Erl, 2008).

1. **Unit Position Validation:** Units must be placed or moved to legal grid coordinates that lie within the defined map boundaries—ranging from (1,1) to (Width, Height), as detailed in Section 2.2—and must not occupy already-used or obstructed cells. Since Unity doesn't check for validity or overlaps when processing `/createUnit` and `/move`, it becomes essential that each agent calculate the map size using the total number of units and ensure that every action respects these boundaries. It's also the agent's job to track which positions are already in use before sending movement or spawn commands.

2. **Handling Zero-Health Units:** Once a unit's health drops to zero or below—typically after receiving an `/attack`—it should be immediately removed using the `/killUnit` endpoint. This also impacts the attacking team's score, which is increased by the defeated unit's deployment cost. Unity does not enforce this removal or award points, so agents need to actively monitor health changes and trigger the elimination themselves when necessary, also keeping track of how many points are earned through combat.

3. **Enforcing Action Cooldowns:** After performing an action such as moving or attacking, each unit must wait at least two seconds before acting again. Unity doesn't enforce this cooldown, so it's possible for agents to exploit it unless controlled. To prevent this, agents must implement internal timers per unit and only allow actions to be issued once the delay has elapsed.

4. **Managing Deployment Points:** Teams begin the game with a fixed number of deployment points (12), which are spent during the registration phase to summon units. Any unspent points contribute to the initial score. Since Unity doesn't track or enforce the point total, agents need to compute and respect the cost of each `/createUnit` call, ensuring they stay within budget. Any points left over must be recorded for inclusion in the end-of-game scoring.

5. **Avoiding Inactivity Penalties:** There are built-in penalties for inaction during the match: if a team fails to perform any actions for 30 seconds, it's eliminated and its score is halved; if it fails to eliminate an enemy unit within 2 minutes, its cheapest remaining unit is automatically removed. These penalties are not enforced by Unity, so agents are responsible for monitoring time intervals and taking action accordingly—either by issuing moves and attacks regularly or triggering the necessary unit elimination after prolonged inactivity.

6. **Validating Attack Targets:** Attacks can only be directed at enemy units that actually exist and fall within the attacking unit's range (e.g., Warriors can reach 1 cell away, Archers 2). Unity accepts all attack requests without

verification, so agents must check that a target is not on the same team and is within reach based on grid position. This requires maintaining an updated view of the game state and computing Manhattan distances before issuing attacks.

7. **Applying Special Abilities:** Healers and Generals offer passive bonuses to nearby allies—Healers provide additional health, while Generals enhance damage. These effects aren't automatically handled by Unity, meaning agents need to detect adjacency during the game and apply the bonuses manually. This involves issuing `/updateHealth` calls when a Healer is next to a unit, or increasing the attack damage when a General is nearby.

8. **Preventing Overlaps:** It's important to ensure that no two units share the same position at any point, whether during placement or movement. Since Unity does not validate this, agents must internally track all unit positions and avoid issuing any commands that would result in two units occupying the same cell.

9. **Respecting Spawn Zones:** Each team has a designated spawn zone based on its ID—for example, Team 1 typically starts in the bottom-left corner, Team 2 in the top-right. While Unity doesn't enforce these zones, it's important for agents to adhere to them to ensure fairness and clarity during deployment. Spawn positions should follow the recommended layout as closely as possible.

10. **Blocking Movement During Attacks:** For visual clarity, units being attacked should remain stationary until the attack animation completes. Since Unity handles actions as they come in, it won't block movement during this time. Agents should avoid issuing `/move` commands for any unit currently under attack, giving time for the visual sequence to complete before allowing further actions.

11. **Handling Movement Invincibility:** Attacks directed at units that are mid-movement must be considered invalid. The system assumes that a moving unit cannot be successfully attacked, but Unity doesn't enforce this logic. To stay compliant with the rules, agents should avoid sending attacks targeting units that are flagged as in transit.

12. **Consistent Unit Naming:** For debugging and visualization purposes, unit names should follow a clear pattern starting with their team (e.g., `Team1_Archer1`). Unity will accept any string, so agents must handle naming consistently and meaningfully to help both developers and evaluators understand what's happening during the game.

13. **Proper Game Initialization:** Before any unit interactions occur, the map must be initialized by specifying the number of players through `/initializeMap`. Unity won't validate whether the game has been properly set up before receiving other API calls, so it's up to agents to send this command early, with a valid player count between 2 and 6.

14. **Detecting Match End Conditions:** The game should conclude either when only one team has units remaining or when the match duration expires. Since Unity relies on agents to detect these conditions, agents must monitor the match timer and keep track of which teams still have active units. Once the game is over, they need to issue a call to `/endGame`, supplying the final

scores.

15. **Final Score Calculation:** Final scores are based on three factors: unspent deployment points, the cost of enemy units eliminated, and the value of surviving friendly units. Unity doesn't compute these, so each agent must tally these components itself and resolve any ties based on the timing of elimination or, if necessary, random selection. The results should be passed to Unity through the `/endGame` command in the correct format.

These game rules require students to develop JADE platform agents that fully manage game state and mechanics using the provided API endpoints in a concurrent, non-turn-based environment. By implementing these rules, students gain hands-on experience in API-driven development, real-time strategy, and distributed systems, aligning with NewWoA's educational goals (Section 1.3) (Erl, 2008).

5 Unity Structure and Visuals

5.1 Gameobjects and Component Structure

NewWoa is composed by various Gameobjects, each of them with different components and functions. A json generator was used to get the information of all Gameobjects before the start of the game. It can be noticed that there is no mention for the map or the units, as they are created once the game is started. The following figure represents the structure of this GameObjects in the Unity Editor.

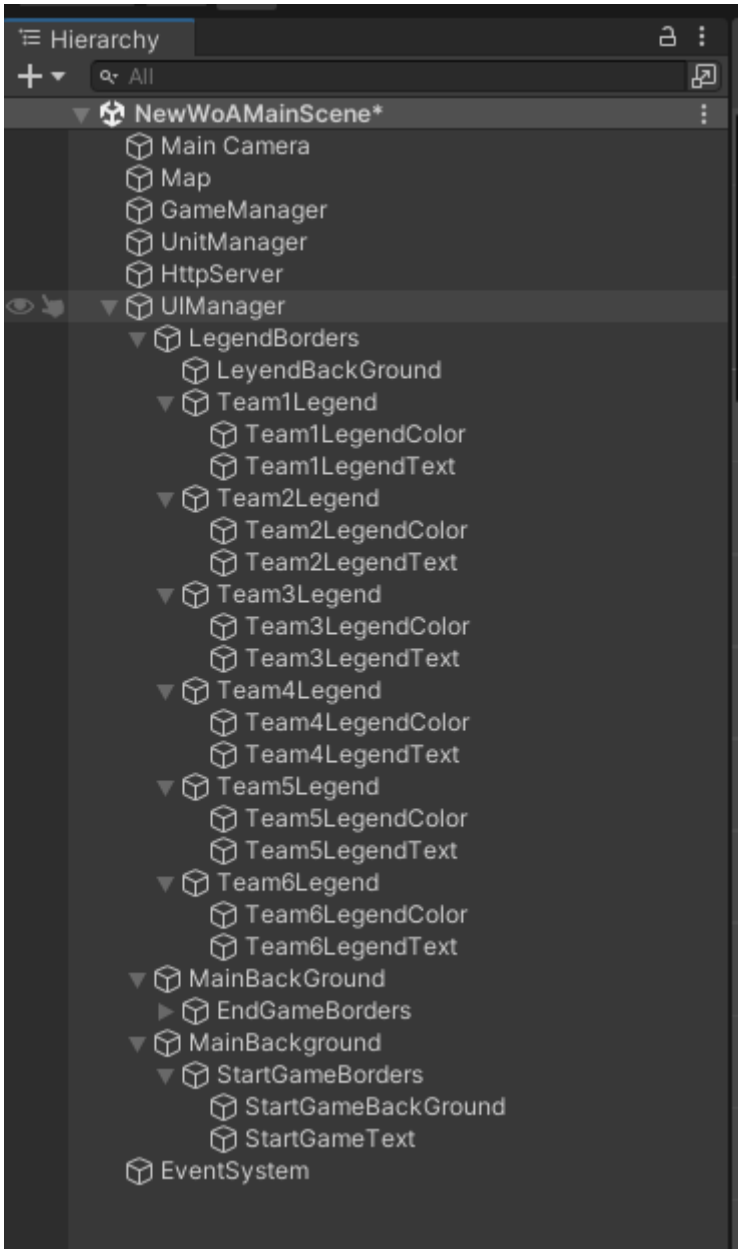

Figure 5.1: Unity GameObject Hierarchy.

5.2 Functional GameObject Analysis

5.2.1 Main Camera


```
{
    "name": "Main Camera",
    "isActive": true,
    "tag": "MainCamera",
    "layer": "Default",
    "components": [
        "Transform",
        "Camera",
        "AudioListener",
        "CameraController"
    ],
    "transform": {
        "position": "(0.00, 0.00, -10.00)",
        "rotation": "(0.00, 0.00, 0.00)",
        "scale": "(1.00, 1.00, 1.00)"
    },
    "children": []
}
```


Renders the game's view as an orthographic camera. Contains the `CameraController` script, which manages camera movement (panning via WASD/arrow keys) and zooming/centering based on grid size.

5.2.2 Map


```
{
    "name": "Map",
    "isActive": true,
    "tag": "Untagged",
    "layer": "Default",
    "components": [
        "Transform",
        "MapGenerator"
    ],
    "transform": {
        "position": "(0.00, 0.00, 0.00)",
        "rotation": "(0.00, 0.00, 0.00)",
        "scale": "(1.00, 1.00, 1.00)"
    },
    "children": []
}
```


Serves as the container for the game's grid-based map. Contains the `MapGenerator` script, which dynamically generates tiles (corners, walls, regular) based on player count.

5.2.3 GameManager


```
{
    "name": "GameManager",
    "isActive": true,
    "tag": "Untagged",
    "layer": "Default",
    "components": [
```

```
        "Transform",
        "SceneAnalyzer"
    ],
    "transform": {
        "position": "(4.60, 5.57, 0.27)",
        "rotation": "(0.00, 0.00, 0.00)",
        "scale": "(1.00, 1.00, 1.00)"
    },
    "children": []
}
```


Manages scene-wide operations. Contains the `SceneAnalyzer` script, which is used to perform this analysis

5.2.4 UnitManager


```
{
    "name": "UnitManager",
    "isActive": true,
    "tag": "Untagged",
    "layer": "Default",
    "components": [
        "Transform",
        "UnitManager"
    ],
    "transform": {
        "position": "(-3.49, 11.59, 0.13)",
        "rotation": "(0.00, 0.00, 0.00)",
        "scale": "(1.00, 1.00, 1.00)"
    },
    "children": []
}
```


Manages game units (spawning, moving, attacking, health updates, removal). Contains the `UnitManager` script, which handles unit creation, smooth movement, attack visuals, and destruction via API calls.

5.2.5 HttpServer


```
{
    "name": "HttpServer",
    "isActive": true,
    "tag": "Untagged",
    "layer": "Default",
    "components": [
        "Transform",
        "HttpListenerServer",
        "UnityMainThreadDispatcher"
    ],
    "transform": {
        "position": "(6.87, 6.08, -0.08)",
        "rotation": "(0.00, 0.00, 0.00)",
        "scale": "(1.00, 1.00, 1.00)"
    },
    "children": []
}
```


Handles HTTP requests for game actions (e.g., map initialization, unit actions).

Contains the `HttpListenerServer` script, which processes POST requests, and `UnityMainThreadDispatcher`, which queues actions for main-thread execution.

5.3 UI Elements Analysis

5.3.1 Start Game Panel

5.3.1.1 StartGameMainBackground

```
{
    "name": "StartGameMainBackground",
    "type": "Image",
    "isActive": true
}
```

Displays a background image for the main game UI, in this case for the start game panel. Managed by the `UIManager` script, it is initially active, hidden during the game. The image used for it is an AI generated image of a medieval era castle:

Figure 5.2: Background Image.

5.3.1.2 StartGameBorders

```
{
    "name": "StartGameBorders",
    "type": "Image",
    "isActive": true
}
```

Description: Provides border graphics for the start game panel. Managed by the `UIManager` script, it is active initially via `SetStartPanelActive(true)` in `InitializeUI()`, framing the start screen. The image used for it is a downloaded image of rock menu:

Figure 5.3: Background Image.

5.3.1.3 StartGameTextBackGround

```
{
    "name": "StartGameTextBackGround",
    "type": "Image",
    "isActive": true
}
```

Description: Serves as the background image for the start game panel. Managed by the `UIManager` script, it is active by default in `InitializeUI()`, forming the start screen's backdrop. It is just a light brwon square to allow readability for the start game panel

5.3.1.4 StartGameText

```
{
    "name": "StartGameText",
    "type": "TextMeshProUGUI",
    "isActive": true
}
```

Description: Displays text for the start game panel (e.g., instructions or title). Managed by the `UIManager` script, it is active initially in `InitializeUI()`, part of the start screen UI. It is a simple text that contains a message saying that the game is waiting for players

When combined, this is the start game panel:

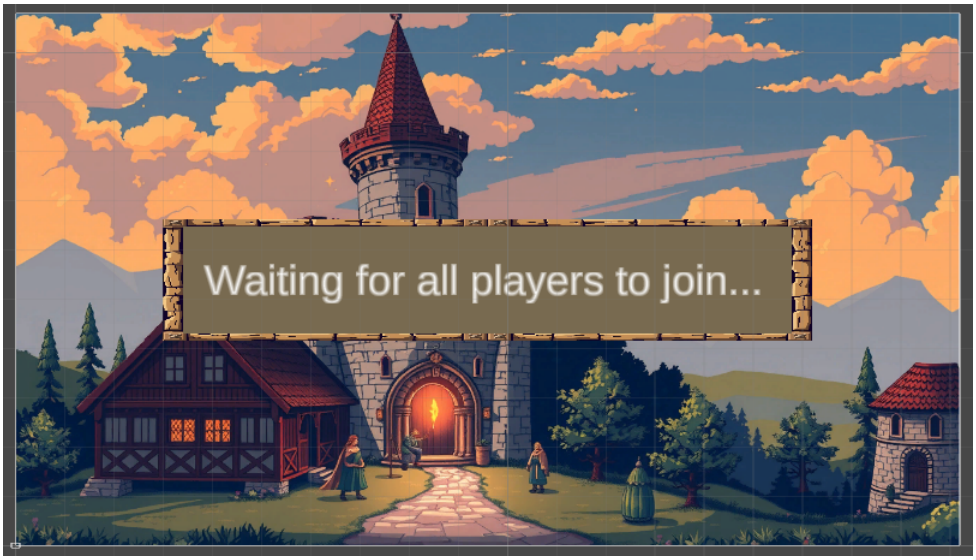

Figure 5.4: Start Game Panel.

5.3.2 Start Game Panel

5.3.2.1 EndGameMainBackGround

```
{
    "name": "EndGameMainBackGround",
    "type": "Image",
    "isActive": false
}
```

Displays a background image for the main game UI, in this case for the end game panel. Managed by the `UIManager` script, it is initially inactive, shown during the end game. The image used for it is the same as 5.2

5.3.2.2 EndGameBorders

```
{
    "name": "EndGameBorders",
    "type": "Image",
    "isActive": false
}
```

Provides border graphics for the end game panel. Managed by the `UIManager` script, it is activated via `SetEndPanelActive(true)` when `EndGame()` is called, framing the end game results. The image used for it is the same as 5.3

5.3.2.3 EndgameTextBackGround

```
{
    "name": "EndgameTextBackGround",
    "type": "Image",
    "isActive": false
}
```

Serves as the background image for the end game panel. Managed by the `UIManager` script, it is shown when `SetEndPanelActive(true)` is triggered in `EndGame()`, displaying the final results. It is just a light brown square to allow readability for the end game panel

5.3.2.4 EndgameText

```
{
    "name": "EndgameText",
    "type": "TextMeshProUGUI",
    "isActive": false
}
```

Displays the final game results text (team rankings, points, winner). Managed by the `UIManager` script, it is updated via `UpdateResultsText()` in `EndGame()`, showing sorted team results and the winner. It is a simple text that contains the points of the teams and the name of the winner

When combined, this is the end game panel:

Figure 5.5: End Game Panel.

.

5.3.3 Legend

The legend follows a similar structure of the start and end game panels, A border, a background and the diferent sliders for each of the six teams:

5.3.3.1 LegendBorders

```
{
    "name": "LegendBorders",
    "type": "Image",
    "isActive": true
 },
```

Provides border graphics for the end game panel. Managed by the `MapGenerator` script, it is activated via `SetLegendBorders(true)` when `(InitializeLegend)` is called, framing the legend. The image used for it is the same as 5.3

5.3.3.2 LegendBackGround

```

{
    "name": "LeyendBackGround",
    "type": "Image",
    "isActive": true
},
```

Serves as the background image for the legend. Managed by the `MapGenerator` script, it is activated via `SetLegendBorders(true)` when `(InitializeLegend)` is called, framing the legend. It is just a brown square,

When combined, the following result is obtained:

Figure 5.6: Legend skeleton.

.

5.3.3.3 LegendSliders

The sliders also consist on a border, a background with the color of the team and the name of the team.

```
            {
                "name": "Team1Legend",
                "type": "Image",
                "isActive": true
            },
            {
                "name": "Team1LegendColor",
                "type": "Image",
                "isActive": true
            },
            {
                "name": "Team1LegendText",
```

```
13                  "type": "TextMeshProUGUI",
14                  "isActive": true
15              }
```

Figure 5.7: Legend single team slider.

.

When introduced in the legend border and background according to the number of teams, the result is the following, varying if the six teams are present or if less of them are:

Figure 5.8: Legend for six teams.

.

When less than six teams are present, a space is left in where their slider should be

Figure 5.9: Legend for three teams.

.

As mentioned in 4.4, this is not the most elegant nor optimal solution, however last minute scope changes where done and the legend had to be included. Based on the time constraints as well as for the sake of maintaining the structure of the project and the coherence with the backend developer Sanchez Rodero (2025) this was the best solution found,

5.4 Dinamically Created GamObjects

5.4.1 Map

Once the map is created and the Game is started, the start game panel is hidden and the map appears. In order to create the map a tilemap was used to pick the desired "pieces" to create the map.

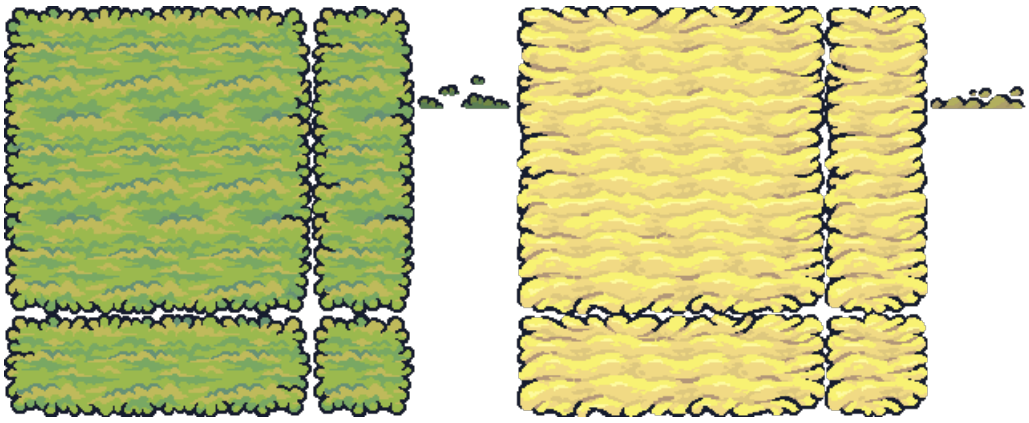

Figure 5.10: TileMap for creating the map.

.

The desired tiles were the top left bigger square in the green color, creating that way a square scalable map whose size depends on the number of players

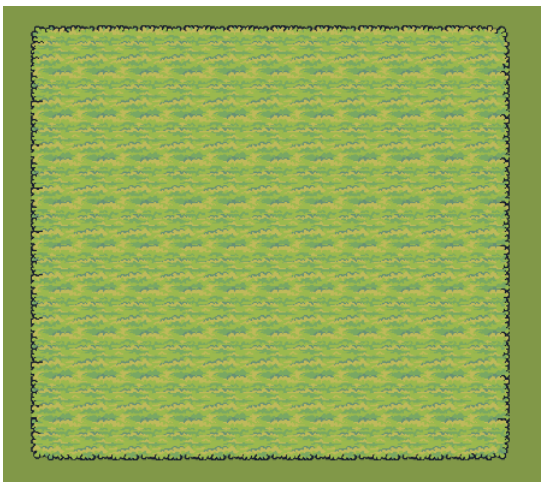

Figure 5.11: NewWoa Map.

.

5.4.2 Units

Once the map is ready Units can be instantiated. This units are prefabs containing a border, a background with the color of their team, a textbox with their current health and a little sprite indicating their type. There are four types of units in the game

Figure 5.12: NewWoa Units.

.

From left to right: Warrior, Archer, General and Healer.

These differrent sprites were chosen from a collection with more characters, preferring to choose the front view for all cases except for the archer, for which the side view was chosen to enhance clarity

Figure 5.13: Units Collection.

5.4.3 Attacks

Once all the units were displayed, last part was to choose an attack animation, for which a explosion sprite was selected.

Figure 5.14: Attack Sprite.

This sprite, when an attack is triggered, is generated in the position of the attacker unit and travels in 0,5sec to the attacked unit's position

6 Test Game Representation

In the following example, the main functioning of the game can be seen via a simulation for a real game made using the specified script for the demo 4.8.

First, the game starts by waiting for teams to register and create their initial units:

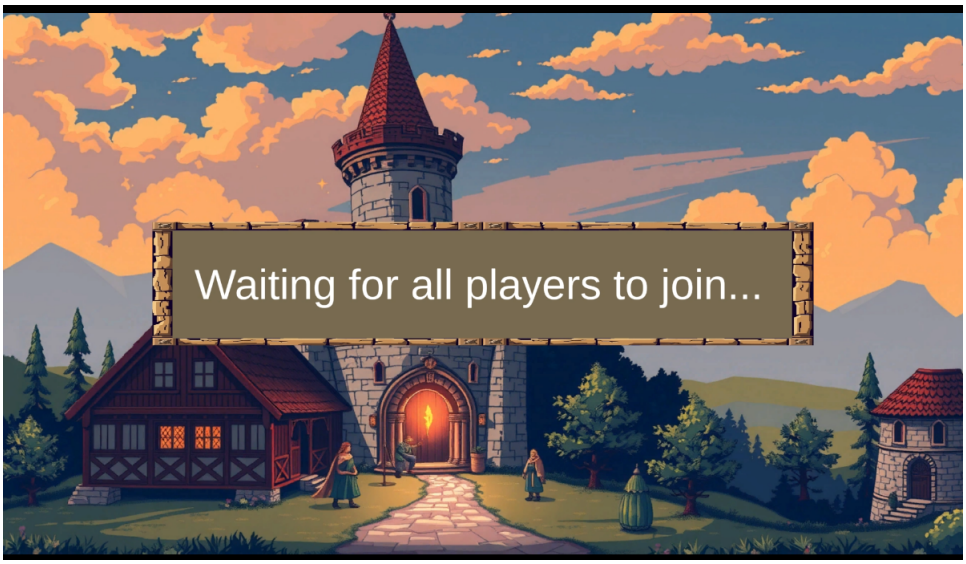

Figure 6.1: Start Game Panel.

Once all the units have been created and the Start Game method is called, the map is created (with the units in it), as well as the Legend for the team colors, both of them based on the final number of players

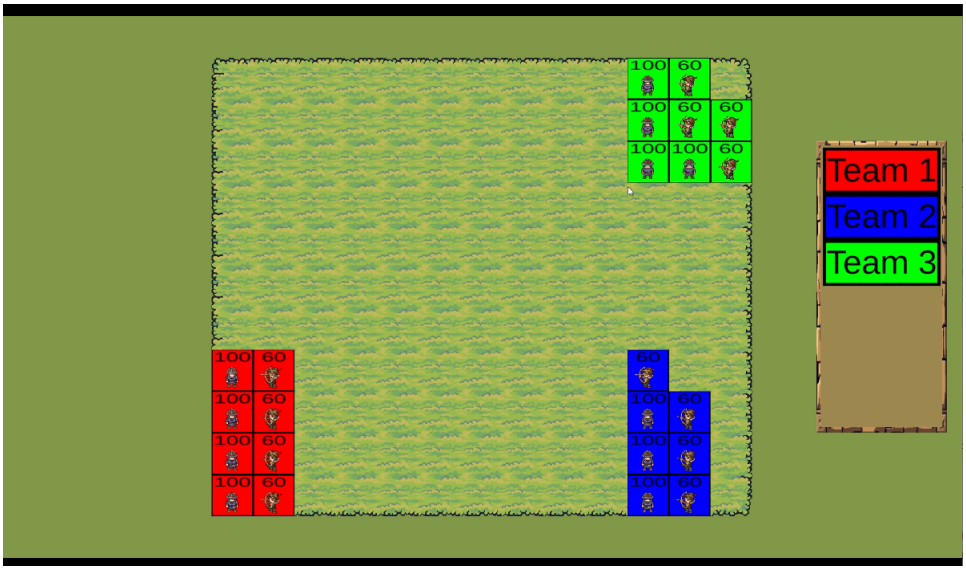

Figure 6.2: Initial LineUp.

Once the map is created with the units, they can start to move around freely

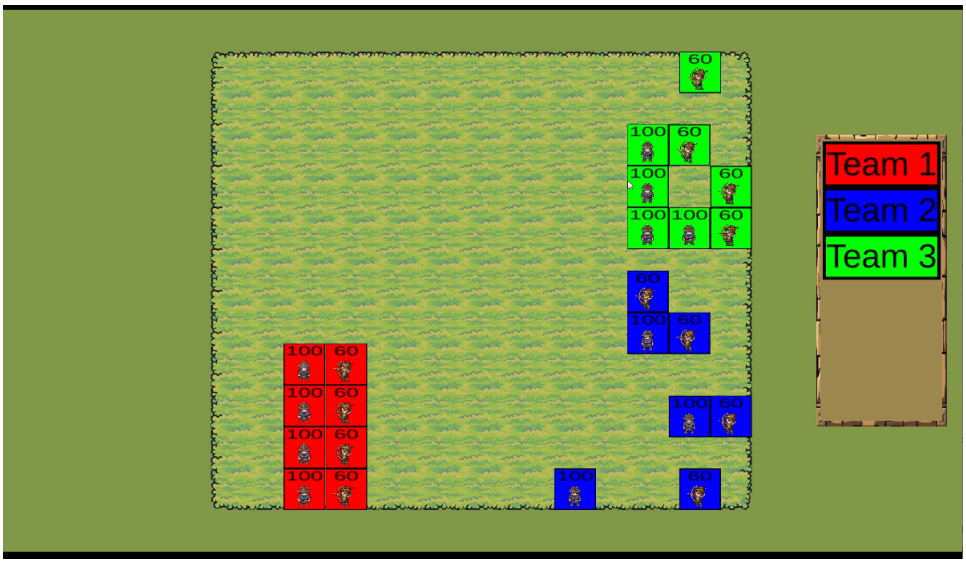

Figure 6.3: Initial Moves.

When units come across ones of the other teams, they can choose to attack them.

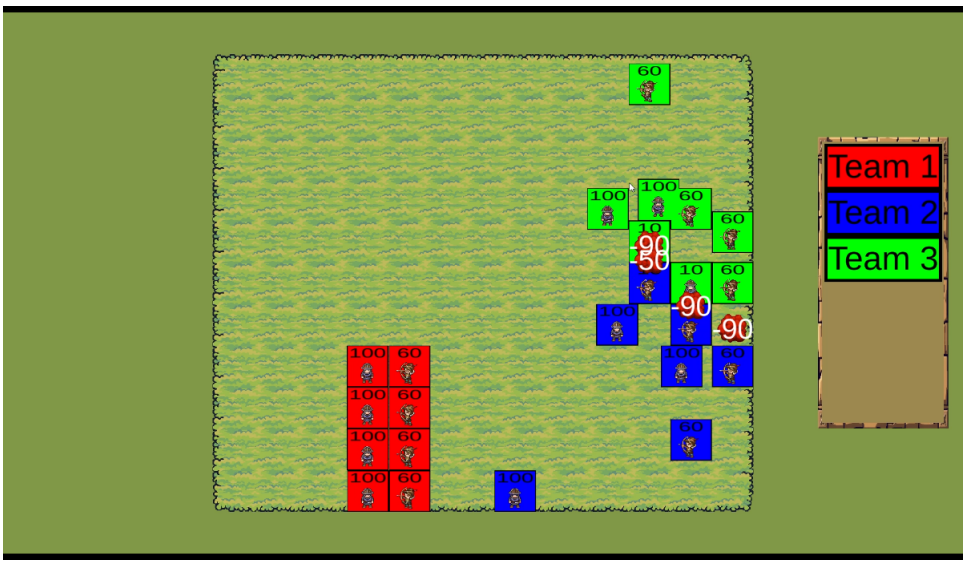

Figure 6.4: Units attacking each other.

As the game goes on Units start to get killed, and only few of them survive

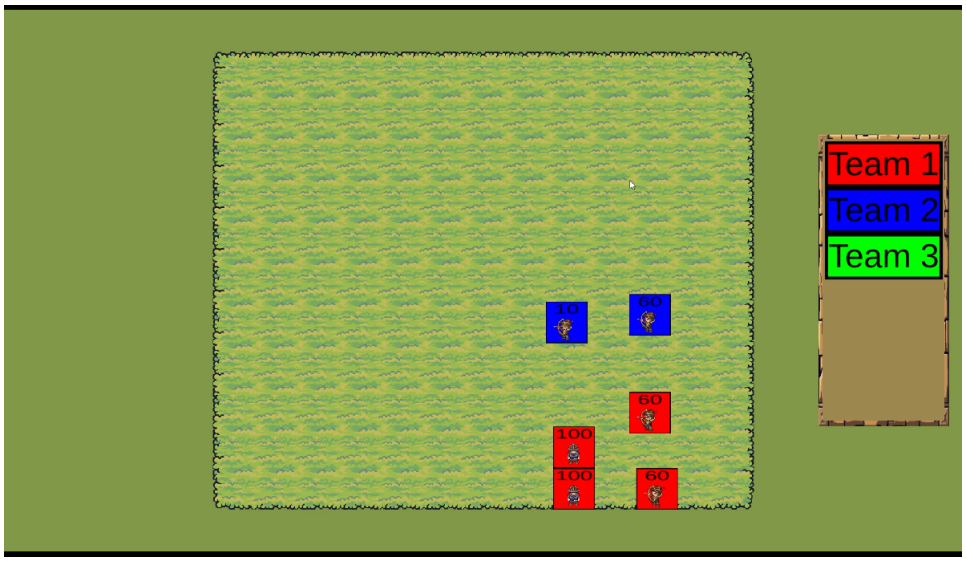

Figure 6.5: Last fight between two teams.

And once one eliminates its last competitor, we reach the conclusion of the game

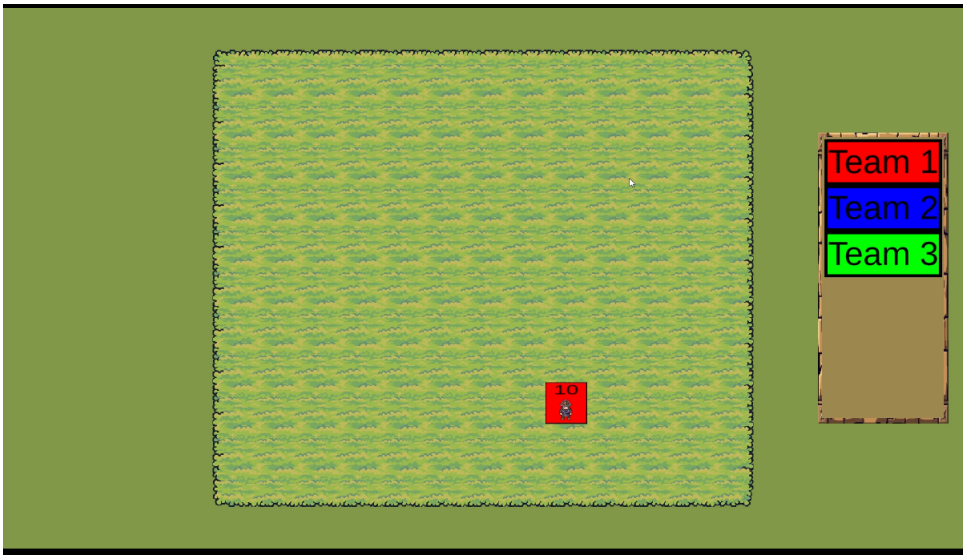

Figure 6.6: End state of the match.

Thus, showing the results of the match, calculating the points for each of them

Figure 6.7: Result of the match.

7 Methodology Used

7.1 Introduction to Sprint Documentation

This section outlines the Agile development process employed for the project, detailing the methodology, estimation approach, developed epics, and the structure of the sprint documentation. The project, spanning from 23 January 2025 to 21 May 2025, was executed through seven sprints, each contributing to the creation of a grid-based strategy game with external API control and enhanced visual and user interface elements.

7.1.1 Agile Methodology

Agile methodology was adopted to guide the project's development, emphasizing iterative progress, collaboration, and adaptability to evolving requirements. By organizing work into time-boxed sprints, we delivered incremental functionality, allowing for continuous feedback and refinement. This approach enabled to respond effectively to challenges, prioritize features based on educational and functional goals, and maintain a steady pace toward project completion. Each sprint included planning, development, testing, and a retrospective to ensure continuous improvement in the processes and outcomes.

7.1.2 User Story Point Estimation

To estimate the effort and complexity of tasks, user story points were utilized, a standard practice in Agile development. User stories were assigned points based on their perceived complexity, scope, and required resources, with one story point approximately equivalent to two hours of focused team effort. This estimation facilitated workload planning, sprint capacity management, and tracking of progress across the project's duration, ensuring balanced and achievable sprint goals.

7.1.3 Developed Epics

The project was organized into several key epics, each representing a major functional area of the game. These epics were incrementally addressed across the seven sprints, building a cohesive and feature-rich application. The epics are:

- **Map Creation**: Establishing a dynamic grid-based map that adjusts size based on the number of players and features an appealing visual tilemap.
- **Unit Management**: Creating units with position fields, displayed as white squares, and later enhanced with team affiliations and unique identifiers.
- **Unit Movement**: Enabling units to move across the grid using coordinate-based inputs, later refined with ID-based controls.
- **Attack Mechanics**: Implementing unit attack functionality with visual representation and state updates, integrated via API.
- **API Integration**: Developing API endpoints for unit creation, movement, attacks, game lifecycle management, and unit state updates, transitioning from position-based to ID-based parameters.

- **Build Generation**: Producing executable builds, YAML configurations, and Postman collections to support testing and external interaction.
- **Visual Enhancement**: Improving unit visuals with team colors, health text, and type-specific sprites for better gameplay clarity.
- **UI Development**: Creating start and end game user interfaces, later refined for polish and intuitiveness.
- **Camera Controls**: Adding zoom and scroll functionality to enhance grid navigation.
- **Documentation**: Documenting sprint progress and producing comprehensive final documentation for project maintenance.

These epics collectively delivered a fully functional strategy game with robust external control and a polished user experience, aligning with the project's educational objectives.

7.1.4 Sprint Structure

Each sprint's documentation follows a consistent structure to provide a clear and comprehensive record of progress, outcomes, and reflections. The structure includes the following subsections:

- **User Stories**: Descriptions of the features or tasks implemented, including their epic, description, acceptance criteria, story points, and completion status. Each sprint contains 4–7 user stories, with Sprint 5 featuring 7 to reflect its extensive API overhaul.
- **Stats of the Sprint**: Key metrics, including the number of epics addressed, total user stories, and story points.
- **Expected Results of the Sprint**: Planned outcomes, including completed stories, story points, sprint duration, and expected velocity (story points per day).
- **Sprint Results**: Actual outcomes, including completed story points, achieved velocity, sprint goal status, and any bugs or issues encountered.
- **Generated Products**: Tangible deliverables produced, such as executables, YAML configurations, Postman collections, or final documentation.
- **Retrospective**: Reflections on what went well, what could be improved, and planned improvements for future sprints, fostering continuous process enhancement.

This structure ensures that each sprint is thoroughly documented, providing insights into the development process, challenges, and achievements. The following sections detail the seven sprints, from initial map creation to final feature polish and documentation, illustrating the project's evolution through Agile practices.

7.2 Sprint 1 - Map Generation with Code and Visual

Duration: 21 days (23/01/2025–12/02/2025)
Objective: Implement a grid-based map with dynamic sizing and an appealing visual tilemap, and document the sprint.

7.2.1 User Stories

7.2.1.1 US 1.1 Initialize Grid Layout

Epic: Map Creation
Description: As a developer, I want to initialize a grid layout to create the map's base structure.
Acceptance Criteria:

- Given the game scene, when the grid is initialized, then an empty grid structure is created.
- Given the grid layout, when inspected, then it has the correct number of rows and columns.

Story Points: 2
Status: Completed – Implemented in the current version.

7.2.1.2 US 1.2 Adjust Grid Size Dynamically

Epic: Map Creation
Description: As a player, I want the map size to adjust based on the number of players for a balanced playing field.
Acceptance Criteria:

- Given 2 players, when the game starts, then the map is a smaller grid.
- Given 4 players, when the game starts, then the map is a larger grid.

Story Points: 1
Status: Completed – Implemented in the current version.

7.2.1.3 US 1.3 Apply Visual Tilemap

Epic: Map Creation
Description: As a player, I want the map to have an appealing tilemap to enhance the visual experience.
Acceptance Criteria:

- Given the grid is initialized, when the map is rendered, then a visually appealing tilemap is applied.
- Given the map is displayed, when inspected, then the tilemap is clear and attractive.

Story Points: 2
Status: Completed – Implemented in the current version.

7.2.1.4 US 1.4 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track progress and outcomes.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.

- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 2
Status: Completed – Implemented in the current version.

7.2.2 Stats of the Sprint

- **Number of epics addressed**: 2 (Map Creation, Documentation)
- **Total number of user stories**: 4
- **Story points**: 7

7.2.3 Expected Results of the Sprint

- **Completed**: 4 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 7
- **Duration of the sprint**: 21 days
- **Expected Sprint velocity**: 0.33 story points per day (7 ÷ 21).

7.2.4 Sprint Results

- **Total story points completed**: 7
- **Velocity achieved**: 0.33 story points per day.
- **Sprint Goal Achieved**: Yes – We implemented a dynamic grid-based map with a visual tilemap and documented the sprint.
- **Bugs or Issues Encountered**: Minor issues with grid alignment were resolved.

7.2.5 Generated Products

No products were generated from this sprint apart from the documentation of the sprint itself.

7.2.6 Retrospective

7.2.6.1 What went well?

- We established the map's core structure, setting a foundation for the project.
- The tilemap enhanced the game's visual appeal, engaging players early.

7.2.6.2 What could be improved?

- The 7-story-point workload over 21 days left significant idle time, reflecting our cautious start.
- Tilemap selection was slow due to unfamiliarity with visual assets.

7.2.6.3 What will we do differently?

- In the next sprint, we will increase the workload to better utilize team capacity.
- We will streamline asset selection with predefined visual guidelines.

7.3 Sprint 2 - Unit Creation and Movement

Duration: 14 days (12/02/2025–26/02/2025)
Objective: Create units with position fields, implement movement functionality, and document the sprint.

7.3.1 User Stories

7.3.1.1 US 2.1 Spawn Units on Grid

Epic: Unit Management
Description: As a player, I want units to be created on the grid to see my units in the game.
Acceptance Criteria:

- Given a position with x,y coordinates, when a unit is spawned, then the unit appears at the specified position.
- Given a unit is spawned, when inspected, then it is visible as a white square.

Story Points: 2
Status: Completed – Implemented in the current version.

7.3.1.2 US 2.2 Display Units as White Squares

Epic: Unit Management
Description: As a player, I want units displayed as white squares to identify them visually.
Acceptance Criteria:

- Given a unit is spawned, when rendered, then it appears as a white square.
- Given multiple units, when inspected, then all units are consistently white squares.

Story Points: 2
Status: Completed – Implemented in the current version.

7.3.1.3 US 2.3 Move Units to Target Position

Epic: Unit Movement
Description: As a player, I want to move units to position them strategically.
Acceptance Criteria:

- Given a unit and fromx,fromy,tox,toy coordinates, when moved, then the unit relocates to the target position.
- Given a unit is moved, when complete, then the unit is at tox,toy coordinates.

Story Points: 3
Status: Completed – Implemented in the current version.

7.3.1.4 US 2.4 Ensure Consistent Movement

Epic: Unit Movement
Description: As a player, I want unit movement to be consistent across grid sizes for reliable gameplay.
Acceptance Criteria:

- Given a small grid, when a unit is moved, then it reaches the target position correctly.
- Given a larger grid, when a unit is moved, then the movement behavior is consistent.

Story Points: 2
Status: Completed – Implemented in the current version.

7.3.1.5 US 2.5 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track progress and outcomes.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 1
Status: Completed – Implemented in the current version.

7.3.2 Stats of the Sprint

- **Number of epics addressed**: 2 (Unit Management, Unit Movement, Documentation)
- **Total number of user stories**: 5
- **Story points**: 10

7.3.3 Expected Results of the Sprint

- **Completed**: 5 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 10
- **Duration of the sprint**: 14 days
- **Expected Sprint velocity**: 0.71 story points per day (10 ÷ 14).

7.3.4 Sprint Results

- **Total story points completed**: 10
- **Velocity achieved**: 0.71 story points per day.

- **Sprint Goal Achieved**: Yes – We created and moved units as white squares and documented the sprint.
- **Bugs or Issues Encountered**: Minor grid edge positioning issues were resolved.

7.3.5 Generated Products

No products were generated from this sprint apart from the documentation of the sprint itself.

7.3.6 Retrospective

7.3.6.1 What went well?

- We successfully implemented unit creation and movement, advancing core gameplay.
- The 10-story-point workload, up from Sprint 1's 7, improved productivity as planned.

7.3.6.2 What could be improved?

- The workload was still slightly light, leaving some capacity unused.
- Edge case testing for unit positioning took longer than expected.

7.3.6.3 What will we do differently?

- In the next sprint, we will further increase workload to maximize capacity.
- We will enhance early testing for edge cases to reduce debugging time.

7.4 Sprint 3 - API Integration and First Build

Duration: 14 days (26/02/2025–11/03/2025)
Objective: Implement API endpoints for unit creation and movement, generate the first build, and document the sprint.

7.4.1 User Stories

7.4.1.1 US 3.1 Implement API for Unit Creation

Epic: API Integration
Description: As a developer, I want an API to create units for external control.
Acceptance Criteria:

- Given x,y coordinates, when the creation API is called, then a unit spawns at the position.
- Given the API is called, when inspected, then the unit appears as a white square.

Story Points: 3
Status: Completed – Implemented in the current version.

7.4.1.2 US 3.2 Implement API for Unit Movement

Epic: API Integration
Description: As a developer, I want an API to move units for external repositioning.
Acceptance Criteria:

- Given fromx,fromy,tox,toy coordinates, when the movement API is called, then the unit moves to the target position.
- Given the API is called, when complete, then the unit is at tox,toy coordinates.

Story Points: 2
Status: Completed – Implemented in the current version.

7.4.1.3 US 3.3 Generate First Build

Epic: Build Generation
Description: As a developer, I want to generate the first build to test the game with APIs.
Acceptance Criteria:

- Given the game code, when the build is generated, then an executable (.exe) runs the game with API functionality.
- Given the build, when run, then unit creation and movement via API work as expected.

Story Points: 1
Status: Completed – Implemented in the current version.

7.4.1.4 US 3.4 Create YAML Configuration

Epic: Build Generation
Description: As a developer, I want a YAML configuration for APIs to enable external interaction.
Acceptance Criteria:

- Given the APIs, when the YAML is created, then it describes the creation and movement endpoints.
- Given the YAML, when reviewed, then it is accurate and complete.

Story Points: 1
Status: Completed – Implemented in the current version.

7.4.1.5 US 3.5 Create Postman Collection

Epic: Build Generation
Description: As a developer, I want a Postman collection to test APIs externally.
Acceptance Criteria:

- Given the APIs, when the Postman collection is created, then it includes requests for creation and movement.
- Given the collection, when tested, then all requests execute correctly.

Story Points: 1
Status: Completed – Implemented in the current version.

7.4.1.6 US 1.4 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track progress and outcomes.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 2
Status: Completed – Implemented in the current version.

7.4.2 Stats of the Sprint

- **Number of epics addressed**: 2 (API Integration, Build Generation)
- **Total number of user stories**: 5
- **Story points**: 10

7.4.3 Expected Results of the Sprint

- **Completed**: 6 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 10
- **Duration of the sprint**: 14 days
- **Expected Sprint velocity**: 0.71 story points per day (10 ÷ 14).

7.4.4 Sprint Results

- **Total story points completed**: 10
- **Velocity achieved**: 0.71 story points per day.
- **Sprint Goal Achieved**: Yes – We implemented APIs for unit creation and movement, generated the first build, and created YAML and Postman assets.
- **Bugs or Issues Encountered**: Minor API validation issues were resolved.

7.4.5 Generated Products

- Game executable (.exe)
- YAML configuration file for API endpoints
- Postman collection for unit creation and movement APIs

7.4.6 Retrospective

7.4.6.1 What went well?

- We enabled external control with APIs, a key milestone, building on Sprint 2's units.
- The first build and testing assets (YAML, Postman) supported efficient development.

7.4.6.2 What could be improved?

- API validation for invalid inputs took longer than anticipated.
- The build process was complex due to initial setup challenges.

7.4.6.3 What will we do differently?

- In the next sprint, we will allocate more time for API validation testing.
- We will automate parts of the build process to streamline future builds.

7.5 Sprint 4 - Attacks and API Integration

Duration: 14 days (11/03/2025–25/03/2025)
Objective: Implement attack functionality, add attack API, generate the second build, and document the sprint.

7.5.1 User Stories

7.5.1.1 US 4.1 Implement Attack Functionality

Epic: Attack Mechanics
Description: As a player, I want units to perform attacks to engage opponents.
Acceptance Criteria:

- Given a unit at fromx,fromy and a target at tox,toy, when an attack is executed, then the attack is visually represented.
- Given an attack, when inspected, then the target unit's state is updated.

Story Points: 3
Status: Completed – Implemented in the current version.

7.5.1.2 US 4.2 Implement API for Attacks

Epic: API Integration
Description: As a developer, I want an API to trigger attacks for external control.
Acceptance Criteria:

- Given fromx,fromy,tox,toy coordinates, when the attack API is called, then the unit at fromx,fromy attacks the unit at tox,toy.
- Given the API, when complete, then the target unit's state reflects the attack.

Story Points: 3
Status: Completed – Implemented in the current version.

7.5.1.3 US 4.3 Generate Second Build

Epic: Build Generation
Description: As a developer, I want an updated build to test attack functionality.
Acceptance Criteria:

- Given the game code, when the build is generated, then an executable (.exe) runs the game with attacks.
- Given the build, when run, then map, units, and attacks function as expected.

Story Points: 1
Status: Completed – Implemented in the current version.

7.5.1.4 US 4.4 Update YAML and Postman Collection

Epic: Build Generation
Description: As a developer, I want updated YAML and Postman assets to include the attack API.
Acceptance Criteria:

- Given the attack API, when YAML is updated, then it describes all endpoints.
- Given the attack API, when the Postman collection is updated, then it includes attack requests.

Story Points: 2
Status: Completed – Implemented in the current version.

7.5.1.5 US 4.5 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track progress and outcomes.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 1
Status: Completed – Implemented in the current version.

7.5.2 Stats of the Sprint

- **Number of epics addressed**: 3 (Attack Mechanics, API Integration, Build Generation, Documentation)
- **Total number of user stories**: 5
- **Story points**: 12

7.5.3 Expected Results of the Sprint

- **Completed**: 5 user stories

- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 12
- **Duration of the sprint**: 14 days
- **Expected Sprint velocity**: 0.86 story points per day (12 ÷ 14).

7.5.4 Sprint Results

- **Total story points completed**: 12
- **Velocity achieved**: 0.86 story points per day.
- **Sprint Goal Achieved**: Yes – We implemented attacks, added attack API, generated the second build, and documented the sprint.
- **Bugs or Issues Encountered**: Minor attack range validation issues were resolved.

7.5.5 Generated Products

- Game executable (.exe)
- Updated YAML configuration file for API endpoints
- Updated Postman collection for unit creation, movement, and attack APIs

7.5.6 Retrospective

7.5.6.1 What went well?

- We advanced gameplay with attacks and their API, fulfilling Sprint 3's validation improvements.
- The 12-story-point workload, up from 10, optimized capacity as planned in Sprint 2.
- Automated build processes from Sprint 3 reduced configuration time.

7.5.6.2 What could be improved?

- Position-based API complexity grew, suggesting a need for unit identifiers.
- Attack range validation required extra effort due to edge cases.

7.5.6.3 What will we do differently?

- In the next sprint, we will introduce unit IDs to simplify API interactions.
- We will enhance validation testing to handle complex features efficiently.

7.6 Sprint 5 - API Overhaul and Visual Enhancements

Duration: 16 days (25/03/2025–09/04/2025)
Objective: Overhaul the API with unit/team IDs, add game lifecycle and state endpoints, enhance unit visuals, and document the sprint.

7.6.1 User Stories

7.6.1.1 US 5.1 Assign Unit and Team IDs

Epic: API Enhancement
Description: As a developer, I want units to have unique IDs and team affiliations for precise API control.
Acceptance Criteria:

- Given a unit is spawned, when the API is called, then the unit has a unique ID and team ID.
- Given multiple units, when inspected, then each unit has distinct IDs and team affiliations.

Story Points: 3
Status: Completed – Implemented in the current version.

7.6.1.2 US 5.2 Update APIs with ID-Based Parameters

Epic: API Enhancement
Description: As a developer, I want APIs to use unit IDs for simpler action management.
Acceptance Criteria:

- Given a unit ID, when the movement API is called with a target position, then the unit moves to the position.
- Given a unit ID, when the attack API is called with a target unit ID, then the attack targets the unit.

Story Points: 3
Status: Completed – Implemented in the current version.

7.6.1.3 US 5.3 Add Game Start Endpoint

Epic: API Enhancement
Description: As a developer, I want a start game endpoint to initialize the game externally.
Acceptance Criteria:

- Given a game session, when the start API is called, then the game initializes with units and teams.
- Given the API, when called, then the game state is set to active.

Story Points: 2
Status: Completed – Implemented in the current version.

7.6.1.4 US 5.4 Add Game End Endpoint

Epic: API Enhancement
Description: As a developer, I want an end game endpoint to conclude the game externally.
Acceptance Criteria:

- Given a game session, when the end API is called, then the game concludes and results are recorded.

- Given the API, when called, then the game state is set to complete.

Story Points: 2
Status: Completed – Implemented in the current version.

7.6.1.5 US 5.5 Add Unit State Endpoints

Epic: API Enhancement
Description: As a developer, I want endpoints to kill units and update health for external state management.
Acceptance Criteria:

- Given a unit ID, when the kill API is called, then the unit is removed from the grid.
- Given a unit ID and health value, when the health API is called, then the unit's health updates.

Story Points: 2
Status: Completed – Implemented in the current version.

7.6.1.6 US 5.6 Enhance Unit Visuals

Epic: Visual Enhancement
Description: As a player, I want units with team colors and health text to track status.
Acceptance Criteria:

- Given a unit, when rendered, then it displays a team color and health text.
- Given multiple units, when inspected, then units show accurate team colors and health.

Story Points: 2
Status: Completed – Implemented in the current version.

7.6.1.7 US 5.7 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track the API overhaul.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the API changes.

Story Points: 1
Status: Completed – Implemented in the current version.

7.6.2 Stats of the Sprint

- **Number of epics addressed**: 3 (API Enhancement, Visual Enhancement, Documentation)
- **Total number of user stories**: 7

- **Story points**: 15

7.6.3 Expected Results of the Sprint

- **Completed**: 7 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 15
- **Duration of the sprint**: 16 days
- **Expected Sprint velocity**: 0.94 story points per day (15 ÷ 16).

7.6.4 Sprint Results

- **Total story points completed**: 15
- **Velocity achieved**: 0.94 story points per day.
- **Sprint Goal Achieved**: Yes – We overhauled the API with unit/team IDs, added lifecycle and state endpoints, enhanced visuals, and documented the sprint.
- **Bugs or Issues Encountered**: Minor health text alignment issues were fixed.

7.6.5 Generated Products

No products were generated from this sprint apart from the documentation of the sprint itself.

7.6.6 Retrospective

7.6.6.1 What went well?

- We transformed the API with ID-based parameters, simplifying interactions as planned in Sprint 4.
- The 15-story-point workload, our highest yet, fully utilized capacity, building on Sprint 3–4's increases.
- Team-colored units with health text greatly improved player experience.

7.6.6.2 What could be improved?

- The API overhaul required extensive refactoring, stretching testing timelines.
- Visual testing for health text alignment was initially insufficient.

7.6.6.3 What will we do differently?

- In the next sprint, we will enhance visual testing to catch rendering issues early.
- We will streamline refactoring to balance feature development and stability.

7.7 Sprint 6 - Visual Improvements and UI

Duration: 29 days (09/04/2025–07/05/2025)
Objective: Enhance unit visuals with type-specific sprites, implement start/end game UIs, generate the third build, and document the sprint.

7.7.1 User Stories

7.7.1.1 US 6.1 Add Unit Type Sprites

Epic: Visual Enhancement
Description: As a player, I want units to display type-specific sprites to distinguish unit types.
Acceptance Criteria:

- Given a unit type (e.g., Warrior, Archer), when rendered, then it displays a type-specific sprite.
- Given multiple units, when inspected, then each unit shows its sprite, color, and health.

Story Points: 3
Status: Completed – Implemented in the current version.

7.7.1.2 US 6.2 Implement Start Game UI

Epic: UI Development
Description: As a player, I want a start game UI to initiate the game clearly.
Acceptance Criteria:

- Given the game is launched, the start UI is displayed.

Story Points: 3
Status: Completed – Implemented in the current version.

7.7.1.3 US 6.3 Implement End Game UI

Epic: UI Development
Description: As a player, I want an end game UI to view game results clearly.
Acceptance Criteria:

- Given the game ends, when the end UI is displayed, then it shows results and rankings.
- Given the end UI, when inspected, then it is clear and accurate.

Story Points: 3
Status: Completed – Implemented in the current version.

7.7.1.4 US 6.4 Generate Third Build

Epic: Build Generation
Description: As a developer, I want an updated build to test new visuals and UIs.
Acceptance Criteria:

- Given the game code, when the build is generated, then an executable (.exe) runs with all features.

- Given the build, when run, then sprites, UIs, and APIs function as expected.

Story Points: 1
Status: Completed – Implemented in the current version.

7.7.1.5 US 6.5 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track progress and outcomes.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 1
Status: Completed – Implemented in the current version.

7.7.2 Stats of the Sprint

- **Number of epics addressed**: 3 (Visual Enhancement, UI Development, Build Generation, Documentation)
- **Total number of user stories**: 5
- **Story points**: 12

7.7.3 Expected Results of the Sprint

- **Completed**: 5 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 12
- **Duration of the sprint**: 29 days
- **Expected Sprint velocity**: 0.41 story points per day (12 ÷ 29).

7.7.4 Sprint Results

- **Total story points completed**: 12
- **Velocity achieved**: 0.41 story points per day.
- **Sprint Goal Achieved**: Yes – We enhanced units with sprites, implemented start/end UIs, generated the third build, and documented the sprint.
- **Bugs or Issues Encountered**: Minor UI scaling issues across resolutions were resolved.

7.7.5 Generated Products

- Game executable (.exe)
- Updated YAML configuration file for API endpoints

- Updated Postman collection for all APIs

7.7.6 Retrospective

7.7.6.1 What went well?

- We improved visuals with sprites, enhancing clarity, as planned after Sprint 5's visual focus.
- The 12-story-point workload was well-balanced despite the 29-day duration, learning from Sprint 5's intensity.

7.7.6.2 What could be improved?

- UI scaling required extra testing, echoing Sprint 5's visual testing needs.
- The long duration lowered velocity, suggesting better workload distribution.

7.7.6.3 What will we do differently?

- In the next sprint, we will implement automated visual testing to catch scaling issues early.

7.8 Sprint 7 - Final Features and Demo Script

Duration: 14 days (07/05/2025–21/05/2025)
Objective: Add zoom scroll, update start game UI, create final documentation, generate the fourth build, and document the sprint.

7.8.1 User Stories

7.8.1.1 US 7.1 Implement Zoom Scroll

Epic: Camera Controls
Description: As a player, I want to zoom and scroll the camera to view the grid flexibly.
Acceptance Criteria:

- Given the game is running, when scroll input is used, then the camera zooms in or out.
- Given different zoom levels, when adjusted, then the grid remains navigable.

Story Points: 1
Status: Completed – Implemented in the current version.

7.8.1.2 US 7.2 Update Start Game UI Panels

Epic: UI Development
Description: As a player, I want updated start UI panels for intuitive game initiation.
Acceptance Criteria:

- Given the game is launched, when the start UI is displayed, then updated panels offer clear options.
- Given the start UI, when interacted with, then it responds smoothly.

Story Points: 1
Status: Completed – Implemented in the current version.

7.8.1.3 US 7.3 Create Demo Script

Epic: Documentation
API Integration: As a developer, I want a demo script to simulate a real game in my environment.
Acceptance Criteria:

- Given the main game, when I execute the demo script then both parts interact perfectly.
- Given the main game, when I execute the demo script then the actions are logical and can be seen as real.

Story Points: 11
Status: Completed – Implemented in the current version.

7.8.1.4 US 7.4 Generate Fourth Build

Epic: Build Generation
Description: As a developer, I want a final build to test all features.
Acceptance Criteria:

- Given the game code, when the build is generated, then an executable (.exe) runs with all features.
- Given the build, when run, then all mechanics and UIs function as expected.

Story Points: 1
Status: Completed – Implemented in the current version.

7.8.1.5 US 7.5 Document Sprint Progress

Epic: Documentation
Description: As a developer, I want to document the sprint to track final progress.
Acceptance Criteria:

- Given the sprint is complete, when documentation is created, then it includes user stories, stats, and retrospective.
- Given the documentation, when reviewed, then it accurately reflects the sprint's achievements.

Story Points: 1
Status: Completed – Implemented in the current version.

7.8.2 Stats of the Sprint

- **Number of epics addressed**: 3 (Camera Controls, UI Development, Documentation, Build Generation)
- **Total number of user stories**: 5
- **Story points**: 15

7.8.3 Expected Results of the Sprint

- **Completed**: 5 user stories
- **In progress**: 0
- **Remaining**: 0
- **Story Points**: 15
- **Duration of the sprint**: 14 days
- **Expected Sprint velocity**: 1.07 story points per day (15 ÷ 14).

7.8.4 Sprint Results

- **Total story points completed**: 15
- **Velocity achieved**: 1.07 story points per day.
- **Sprint Goal Achieved**: Yes – We added zoom scroll, updated start UI, created the demo script, generated the fourth build, and documented the sprint.
- **Bugs or Issues Encountered**: Minor zoom responsiveness issues were fixed.

7.8.5 Generated Products

- Game executable (.exe)
- Updated YAML configuration file for API endpoints
- Updated Postman collection for all APIs
- Demo script for the game

7.8.6 Retrospective

7.8.6.1 What went well?

- We completed the project with polished features, achieving our highest velocity (1.07) due to experience from Sprint 6.
- The 15-story-point workload reflected optimized planning from Sprint 1's underutilization.
- A polished demo allowed us to see the results of our developments.

7.8.6.2 What could be improved?

- Zoom responsiveness required fine-tuning, indicating ongoing visual testing needs.
- The high workload strained testing resources slightly.

7.8.6.3 What will we do differently?

- In future projects, we will develop the demo earlier, to fix errors from the start.

7.8.7 Final Retrospective

Over the course of the project, a total of 7 sprints were completed, each planned with a duration of approximately two weeks. These sprints resulted in 4 functional builds and the implementation of more than 35 individual user stories, grouped under 7 defined epics.

All core features planned at the start of the project were completed. The communication between Unity and the backend via HTTP endpoints was fully established by sprint 5, and visual feedback for all game actions was implemented by sprint 6. Visual UI components—including start and end panels—were progressively refined in the final two sprints, based on the foundational structure laid earlier.

Each sprint ended with a functioning system state, starting from sprint 2, which allowed continuous testing and incremental validation. From sprint 3 onward, builds were stable enough to test end-to-end interaction between the simulation script and Unity. The final build includes full support for map generation, unit creation and interaction, dynamic camera, attack logic, and end-of-game summary.

The system remains modular, with five major Unity components developed independently and loosely coupled via centralized dispatching. The Python backend was kept decoupled and required no structural modifications after sprint 4.

No critical regressions or rewrites occurred after sprint 3, indicating that architectural decisions made early on remained valid and that the development flow was consistent. The only adjustments made were additive, extending functionality without invalidating previous work.

All planned sprint meetings with supervisors were held, and feedback from those sessions directly influenced sprint planning. The project stayed within its original functional scope and was completed on schedule.

8 Future Steps, Handover Guide and Conclusions

8.1 Conclusions

Throughout this work, a completely new graphical interface for the World of Agents (WoA) exercise has been developed, replacing a previous version with significant visual and technical limitations. The main objective was to adapt the system to serve as a modern and didactic tool, integrating a clear visualisation in Unity 3D that allows to represent the actions executed by the agents developed by students.

One of the main achievements of the project has been to completely decouple the logic of the agents, which are still executed using JADE, from the visualisation and graphical execution, which now takes place in Unity. This was achieved by means of a communication system via REST API that allows the agents to send commands to the environment, which are interpreted by the visual interface. This design not only improves the modularity of the system, but also allows students to focus on developing their decision logic without worrying about graphical or execution details.

During development, all the game flow necessary for teaching practice has been implemented, including the registration, combat, and completion phases, with their respective rules and constraints. Units are correctly created on screen, execute their actions with specific animations, and the system evaluates victory or defeat conditions automatically. In addition, a visible scoring system was added to reinforce the idea of controlled competition within the educational environment.

From a technical point of view, a clean and scalable structure has been achieved, with modular code divided into components such as the unit controller, map generator, camera management and user interface. This facilitates the future addition of new elements without breaking the existing system.

In terms of the initial objectives, it can be stated that all of them have been met: a new interface adapted to Unity 3D has been created, communication with external agents has been maintained, and a complete and functional solution has been offered that significantly improves the educational experience of the original practice. The system now allows for greater clarity in the interpretation of autonomous behaviours and facilitates both assessment and student learning.

With this final version, the project is in a stable, deployable state and ready for use in teaching contexts, providing a solid foundation for future practices in artificial intelligence, autonomous agents and simulation.

Regarding the methodology, the application of an Agile to the development of this project was not done following hard rules. There were no daily stand-ups, no formal burndown charts, and velocity was not tracked quantitatively. Instead, Agile served as a guiding principle—a framework for disciplined iteration and continuous improvement rather than a rigid methodology.

Ultimately, this adaptation of Agile proved well-suited to the academic context.

It supported both the technical development and the collaborative supervision dynamic, allowing the project to evolve organically while ensuring consistent forward momentum. The result is a fully functional, extensible, and visually rich simulation platform that aligns with both the original scope and the practical needs of the target audience.

8.2 Vertical Improvements: Expanding Functional Capabilities

8.2.1 Overview

The following vertical improvements are proposed to extend the functional scope of the New World of Agents (NewWoA) educational platform. These improvements aim not only to diversify the strategic landscape available to students but also to enhance the adaptability, realism, and analytical power of the system. The features outlined here are designed to integrate seamlessly with the current Unity 3D front-end and the JADE-based back-end agent systems via the established API communication pipeline. In particular,four key functional extensions are presented: a replay system for post-match analysis, dynamic unit spawning during the game phase, parameterization of game settings through API calls, and the introduction of obstacles in the game map.

8.2.2 Replay System with Playback Controls

One of the most pedagogically impactful improvements is the implementation of a replay system capable of reconstructing past matches for analytical purposes. The replay mechanism involves capturing all executed API calls (e.g., unit creation, movement, attack, and elimination) in a time-ordered log during the course of a game. Each entry in this log is timestamped with the game clock and serialized into a structured format (such as JSON or binary) that is stored upon game conclusion.

To facilitate replay visualization, a new component, `ReplayManager`, would be introduced in Unity. This manager reinterprets the saved log data and replicates the match progression frame-by-frame, independent of real-time user interaction. A dedicated user interface would allow users to control the playback with standard media functionalities such as pause, step forward/backward, fast-forward, and jump-to-event. Furthermore, event markers (e.g., first elimination, highest damage dealt) can be used to annotate the timeline for instructional navigation.

This functionality provides a critical feedback loop in the learning process. Students can review their agents' behaviors in detail, assess the consequences of specific actions, and compare alternate strategies by replaying successive iterations of their code. Instructors can also use replays to conduct guided analysis in classroom settings or during evaluations.

8.2.3 Dynamic Unit Spawning During Match

The traditional model of static unit deployment during the initial registration phase is limited in scope and restricts strategic adaptability. By permitting dynamic unit spawning during the match phase, NewWoA would support a more fluid and realistic decision-making environment. This functionality intro-

duces a new class of agent behavior: resource-constrained, context-sensitive reinforcement.

A new RESTful API endpoint, such as `/api/spawnUnit`, would allow agent systems to request the creation of additional units during the active match phase. The request must include the type of unit to be created, the intended spawn location, and an identifier for the unit. Upon receipt, the back-end would validate several constraints, including available deployment points, cell occupancy, team territory bounds, and overall unit limits.

On successful validation, the Unity interface would instantiate the unit with the appropriate visual attributes and perform any associated spawn animation. This enables teams to reinforce threatened positions, shift offensive pressure, or execute late-game tactics such as pincer movements or decoys.

This improvement adds a compelling tactical layer to gameplay, requiring agents not only to plan ahead but also to react dynamically to evolving battlefield conditions. From an educational perspective, it challenges students to design agents capable of mid-game reassessment and opportunistic exploitation.

8.2.4 API-Driven Game Configuration

Currently, NewWoA relies on hardcoded constants or precompiled configurations for setting key game parameters such as match duration, registration time, unit fatigue thresholds, and maximum unit count per team. This inflexibility hampers experimentation and scenario-specific tuning.

To address this, the implementation of an API-based configuration system is proposed, allowing dynamic parameter specification prior to match start. A dedicated endpoint, such as `/api/configureGame`, would accept a JSON payload containing all adjustable parameters. This can include, but is not limited to:

- `durationMatch`: total duration of the match phase (in seconds)
- `durationRegistration`: duration of the registration phase
- `fatigueTime`: delay between allowed actions per unit
- `maxUnitsPerTeam`: cap on units per team
- `mapSeed`: seed value for deterministic map generation

Upon receiving this payload, the back-end would store the configuration in session memory, and the Unity interface would query and apply these parameters during its initialization routine. If no configuration is provided, default parameters would be used to ensure backward compatibility.

This improvement greatly enhances the pedagogical utility of the system. Instructors can design custom sessions tailored to specific instructional goals, such as reduced match lengths to test high-tempo strategies, or increased fatigue penalties to illustrate resource management. Additionally, students can conduct comparative studies by varying parameters and observing corresponding changes in agent performance.

8.2.5 Obstacle Integration in the Game Map

The current NewWoA map is a uniformly traversable two-dimensional grid, which, while simplifying initial development, lacks spatial constraints that encourage

complex strategic behaviors. By incorporating obstacles into the map, the system can simulate a more realistic environment and foster the development of advanced agent logic, such as pathfinding and territorial control.

Obstacles would be defined in the map initialization payload provided by the back-end to the Unity interface. Each tile in the payload would include a `type` field specifying whether the tile is walkable, blocked (obstacle), or special terrain (e.g., with increased movement cost). Unity would render these tiles with distinct textures or overlays to visually convey their properties.

In the initial implementation, obstacle tiles would simply prevent unit movement, serving as impassable areas. Future extensions may introduce terrain types with associated movement penalties, enabling instructional modules on algorithms like A*, Dijkstra's, or heatmap-based path planning.

This improvement significantly raises the strategic ceiling of the game. Agents would need to consider navigation constraints, bottlenecking, and zone control in their decision processes. It also offers a fertile ground for coursework or labs focused on spatial reasoning, cooperative traversal, and environmental interaction in autonomous systems.

8.3 Horizontal Improvements: Enhancing Existing Capabilities

8.3.1 Overview

Whereas vertical improvements aim to introduce new functionalities, horizontal improvements focus on refining and expanding the quality, usability, and clarity of features that already exist within the system. In the case of NewWoA, such improvements are particularly relevant in the domains of user interface design, visual navigation, and post-match evaluation. These enhancements do not alter the pedagogical structure or the core logic of the game but substantially elevate the user experience and accessibility of information. The following sections detail three principal proposals: a modular, tabbed user interface; advanced camera and map view controls; and a structured match summary screen.

8.3.2 Modular User Interface with Tabbed Panels

The current interface architecture in NewWoA, while functional, presents all informational elements on a single visual layer, leading to potential clutter as the game progresses and visual congestion increases. To address this, a modular user interface based on tabbed panels is proposed. This redesign organizes game information into clearly delineated sections, each accessible via a tab selector, allowing users to focus selectively on the information relevant to their current needs.

The interface would be structured into the following core tabs:

- **Scoreboard Tab:** Displays current team rankings, point tallies, number of active units, and accumulated kills.
- **Unit Details Tab:** Activated when a unit is selected; shows health, damage, range, fatigue state, and recent actions of the selected unit.

- **Action Log Tab:** Chronologically lists all significant events (e.g., attacks, eliminations, spawns), with timestamps and links to focus the camera on each event.

This modular structure improves usability by separating static and dynamic data, and by minimizing the overlap between informational components and the main map. Implemented in Unity using either the UI Toolkit or Unity's traditional Canvas system, the tabbed interface can be toggled using a global controller that animates transitions and manages panel visibility states. This approach also simplifies future expansions, allowing the addition of further informational tabs (e.g., team strategies, agent diagnostics) with minimal architectural disruption.

8.3.3 Enhanced Camera and Visual Navigation Controls

Another key improvement concerns the player's ability to follow in-game events, especially during high-tempo interactions or on large-scale maps. The default camera controls—manual panning and zooming—can become cumbersome and detract from the player's situational awareness. As such, the implementation of automated and contextual camera behavior is proposed to enhance the navigability of the game environment.

Two principal features would be introduced:

1. Contextual Focus via Double-Click: Users would be able to double-click on any unit to automatically reposition and zoom the camera to center on that unit. This functionality facilitates quick inspection of individual agents and their immediate surroundings without manual navigation.

2. Event-Triggered Auto-Focus: Significant game events—such as unit eliminations, concentrated attacks, or multiple actions within a short time frame—would trigger automatic camera transitions to the relevant map area. These transitions would utilize Unity's Cinemachine system, which provides smooth, priority-based blending between multiple virtual camera states.

Together, these features reduce cognitive load by allowing users to focus on strategic evaluation rather than manual map management. They also ensure that key events are not missed, even when the user is observing another part of the map or is temporarily inactive.

8.3.4 Structured Match Summary Screen

Upon completion of each match, NewWoA currently concludes with a simple indication of the winner. While this satisfies the basic requirement of declaring a victor, it offers little in the way of diagnostic insight or performance feedback. To enhance the educational value of each match, the implementation of a structured summary screen that compiles and presents comprehensive statistics for all participating teams is proposed.

The match summary would include, but not be limited to, the following data points:

- **Units Survived and Eliminated:** Per-team count of remaining units and total losses.
- **Damage Dealt and Received:** Total numerical values indicating offensive and defensive effectiveness.

- **Action Frequencies:** Breakdown of how many times each team executed movements, attacks, and spawns.
- **Event Timestamps:** Time of first kill, last action, and longest period of inactivity.

These statistics would be visualized using bar charts, tables, or radial graphs within a dedicated overlay panel. The underlying data would be accumulated during runtime by a centralized `MatchStats` object, which listens to all relevant events and maintains per-team aggregates. At the end of the match, this object populates the summary UI with structured, ready-to-display data.

The inclusion of a match summary screen directly supports reflection and post-match evaluation. It enables students to identify not only tactical errors but also inefficiencies or suboptimal behaviors in their agent design. For instructors, it provides an objective basis for assessment and can be exported (e.g., as CSV or JSON) for further analysis or grading purposes.

8.4 Maintenance and Future Development Guidelines

In anticipation of project deployment and future handover to new development teams, this section outlines essential maintenance tasks and extension procedures for the NewWoA platform. These are intended to simplify the onboarding process for future collaborators, whether they are researchers, students, or developers tasked with adapting or extending the system.

8.4.1 Updating Initial Unit Health Points

Each unit's base health is defined within the `UnitManager.cs` script under the `CreateUnit` or `SpawnUnit` method.

1. Open `UnitManager.cs` in the Unity project.
2. Locate the section where unit stats (health, damage, range) are initialized, typically via a `switch` or `if-else` structure on `unitType`.
3. Modify the `unit.health` parameter for the relevant type.
4. Save changes and recompile the project.
5. Optional: Update the health value in any related Postman or YAML configuration files used in test scripts.

8.4.2 Adding New Unit Types

To introduce a new unit type (e.g., "Tank" or "Scout"), the following steps are necessary:

1. In `UnitManager.cs`, extend the logic handling unit creation to account for the new type:
   - Define health, damage, range, deployment cost.
   - Optionally define a special ability.
2. Add a corresponding prefab or sprite in the Unity `Resources/Units` folder.

3. Extend any relevant UI components (e.g., unit details panel) to recognize and label the new type.
4. Update the back-end API schema or documentation to allow this type to be accepted in unit creation requests.
5. Validate the new unit through test matches using the simulation script (`RealSimulation.py`).

8.4.3 Adding New API Endpoints

If new types of agent actions or gameplay features are added (e.g., healing zones, power-ups), corresponding API endpoints must be added to the `HttpListenerServer.cs` component.

1. Open `HttpListenerServer.cs` and add a new route in the `HandleRequest` method.
2. Implement the handler method to process the new action (e.g., `HandleShieldBoost()`).
3. If Unity is required to respond visually, enqueue a Unity task using `UnityMainThreadDispatcher.Enqueue()`.
4. Add the new endpoint to the Postman collection and YAML documentation under `docs/api-spec.yaml`.
5. Optionally, implement validation and error handling for malformed requests or in-game rule violations.

8.4.4 General Guidelines for Future Contributors

- **Code Architecture:** Follow the singleton design pattern where applicable for managers (e.g., `UIManager`, `MapGenerator`).
- **Extensibility:** All scripts are organized by functionality in the `Scripts/` directory. Use component-based logic whenever possible.
- **Testing:** Use the provided `RealSimulation.py` script to simulate a full game run with automated agents.
- **Configuration:** Default parameters can be overridden using the proposed `/api/configureGame` endpoint. In the future, avoid hardcoding game constants.
- **Documentation:** Maintain internal documentation updating the API spec with each new feature.

These procedures will ensure that NewWoA remains functional, adaptable, and educationally effective as it evolves under new stewardship.

Bibliography

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., and Hillaire, S. (2018). *Real-Time Rendering*. CRC Press, 4 edition.

Axelrod, R. (1997). *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton University Press.

Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Wiley.

De, B. (2017). *API Management: An Architect's Guide to Developing and Managing APIs*. Apress.

Dickson, P. E. (2015). Using unity to teach game development: When you've never written a game. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 75–80.

ECMA International (2017). *The JSON Data Interchange Syntax*. ECMA International.

Erl, T. (2008). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Gregory, J. (2018). *Game Engine Architecture*. CRC Press, 3 edition.

Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S. K., Huse, G., et al. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, 198(1-2):115–126.

Haas, J. K. (2014). A history of the unity game engine. *Worcester Polytechnic Institute*.

Internet Engineering Task Force (IETF) (2011). *The WebSocket Protocol (RFC 6455)*. IETF.

Internet Engineering Task Force (IETF) (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content (RFC 7231)*. IETF.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multi-agent simulation environment. *Simulation*, 81(7):517–527.

Macal, C. M. and North, M. J. (2010). Tutorial on agent-based modeling and simulation. *Journal of Simulation*, 4(3):151–162.

Menard, M. (2019). *Game Development with Unity*. Cengage Learning, 2 edition.

OpenAPI Initiative (2021). *OpenAPI Specification v3.1.0*. OpenAPI Initiative.

Postman Inc. (2024). *2024 State of the API Report*. Postman Inc.

Railsback, S. F. and Grimm, V. (2019). *Agent-Based and Individual-Based Modeling: A Practical Introduction*. Princeton University Press, 2 edition.

Sanchez Rodero, A. (2025). *Ontology Generator and Backend Environment for New World of Agents Game*. PhD thesis, UPM.

Technologies, U. (2021). *Unity 2021.3 Documentation*. Unity Technologies.

Technologies, U. (2023). *Unity 2023.2 Documentation*. Unity Technologies.

Wilensky, U. and Rand, W. (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley, 2 edition.

YAML Community (2021). *YAML 1.2 Specification*. YAML Community.

9 Annexes

Annexe 1 : YAML API Documentation

```yaml
# API Specification for Unity HttpListenerServer
openapi: 3.0.3
info:
  title: Unity Game HTTP API
  description: API for controlling a Unity-based game server, handling map initialization, unit management, and game state.
  version: 1.0.0
servers:
  - url: http://localhost:5002
    description: Local Unity game server
paths:
  /initializeMap:
    post:
      summary: Initialize game map
      description: Initializes the game map for a specified number of players.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/InitializeMapRequest"
      responses:
        "200":
          description: Map initialization queued
          content:
            text/plain:
              schema:
                type: string
                example: Map initialization queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /createUnit:
    post:
      summary: Create a unit
      description: Spawns a unit at specified coordinates with given type, ID, and team.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/CreateUnitRequest"
      responses:
        "200":
          description: Unit creation queued
          content:
            text/plain:
              schema:
                type: string
                example: Unit creation queued.
        "500":
          description: Server error
```

```
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /move:
    post:
      summary: Move a unit
      description: Moves a unit to the specified coordinates.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/MoveUnitRequest"
      responses:
        "200":
          description: Unit movement queued
          content:
            text/plain:
              schema:
                type: string
                example: Unit movement queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /attack:
    post:
      summary: Perform an attack
      description: Handles an attack from one unit to another with
    specified damage.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/AttackUnitRequest"
      responses:
        "200":
          description: Attack queued
          content:
            text/plain:
              schema:
                type: string
                example: Attack queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /killUnit:
    post:
      summary: Kill a unit
      description: Removes a unit from the game.
      requestBody:
        required: true
        content:
```

```yaml
          application/json:
            schema:
              $ref: "#/components/schemas/KillUnitRequest"
      responses:
        "200":
          description: Unit kill queued
          content:
            text/plain:
              schema:
                type: string
                example: Unit kill queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /updateHealth:
    post:
      summary: Update unit health
      description: Updates the health of a specified unit.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/UpdateHealthRequest"
      responses:
        "200":
          description: Health update queued
          content:
            text/plain:
              schema:
                type: string
                example: Health update queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /startGame:
    post:
      summary: Start the game
      description: Triggers the game start and updates UI panels.
      requestBody:
        required: false
        content:
          application/json:
            schema:
              type: object
      responses:
        "200":
          description: Game start queued
          content:
            text/plain:
              schema:
                type: string
                example: Game start queued.
        "500":
          description: Server error
```

```
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
  /endGame:
    post:
      summary: End the game
      description: Ends the game and displays results for the specified
    teams.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/EndGameRequest"
      responses:
        "200":
          description: Game end queued
          content:
            text/plain:
              schema:
                type: string
                example: Game end queued.
        "500":
          description: Server error
          content:
            text/plain:
              schema:
                type: string
                example: "Error: [error message]"
components:
  schemas:
    InitializeMapRequest:
      type: object
      required:
        - numPlayers
      properties:
        numPlayers:
          type: integer
          description: Number of players for the game map
          example: 2
    CreateUnitRequest:
      type: object
      required:
        - x
        - y
        - id
        - type
        - team
      properties:
        x:
          type: integer
          description: X-coordinate for unit spawn
          example: 10
        y:
          type: integer
          description: Y-coordinate for unit spawn
          example: 20
        id:
          type: string
          description: Unique identifier for the unit
          example: unit_123
```

```
        type:
          type: integer
          description: Type of the unit
          example: 1
        team:
          type: integer
          description: Team ID the unit belongs to
          example: 1
    MoveUnitRequest:
      type: object
      required:
        - id
        - toX
        - toY
      properties:
        id:
          type: string
          description: Unique identifier of the unit to move
          example: unit_123
        toX:
          type: integer
          description: Destination X-coordinate
          example: 15
        toY:
          type: integer
          description: Destination Y-coordinate
          example: 25
    AttackUnitRequest:
      type: object
      required:
        - attackerId
        - targetId
        - damage
        - targetHealthLeft
      properties:
        attackerId:
          type: string
          description: Unique identifier of the attacking unit
          example: unit_123
        targetId:
          type: string
          description: Unique identifier of the target unit
          example: unit_456
        damage:
          type: integer
          description: Amount of damage dealt
          example: 50
        targetHealthLeft:
          type: integer
          description: Remaining health of the target unit
          example: 50
    KillUnitRequest:
      type: object
      required:
        - unitId
      properties:
        unitId:
          type: string
          description: Unique identifier of the unit to kill
          example: unit_123
    UpdateHealthRequest:
      type: object
      required:
```

```
306         - unitId
307         - healthLeft
308       properties:
309         unitId:
310           type: string
311           description: Unique identifier of the unit
312           example: unit_123
313         healthLeft:
314           type: integer
315           description: Updated health value for the unit
316           example: 75
317     EndGameRequest:
318       type: object
319       required:
320         - teams
321       properties:
322         teams:
323           type: array
324           description: List of team results
325           items:
326             type: object
327             description: Team result details (structure not specified in
      code)
```

Annexe 2 : Postman API Collection


```
{
  "info": {
    "name": "Unity Game Server API",
    "_postman_id": "7b8f3a2e-9c1d-4c7b-8f2a-5e6c8b7f1234",
    "description": "Postman collection for interacting with the NewWoA API running on http://localhost:5002.",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
  },
  "item": [
    {
      "name": "Initialize Map",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"numPlayers\": 2\n}"
        },
        "url": {
          "raw": "http://localhost:5002/initializeMap",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["initializeMap"]
        },
        "description": "Initializes the game map for a specified number of players."
      },
      "response": [
        {
          "name": "Success",
          "status": "OK",
          "code": 200,
          "header": [],
          "body": "Map initialization queued."
        },
        {
          "name": "Error",
          "status": "Internal Server Error",
          "code": 500,
          "header": [],
          "body": "Error: {{error_message}}"
        }
      ]
    },
    {
      "name": "Create Unit",
      "request": {
        "method": "POST",
        "header": [
          {
```

```
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"x\": 10,\n  \"y\": 20,\n  \"id\": \"unit_123\",\n  \"type\": 1,\n  \"team\": 1\n}"
        },
        "url": {
          "raw": "http://localhost:5002/createUnit",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["createUnit"]
        },
        "description": "Spawns a unit at specified coordinates with given type, ID, and team."
      },
      "response": [
        {
          "name": "Success",
          "status": "OK",
          "code": 200,
          "header": [],
          "body": "Unit creation queued."
        },
        {
          "name": "Error",
          "status": "Internal Server Error",
          "code": 500,
          "header": [],
          "body": "Error: {{error_message}}"
        }
      ]
    },
    {
      "name": "Move Unit",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"id\": \"unit_123\",\n  \"toX\": 15,\n  \"toY\": 25\n}"
        },
        "url": {
          "raw": "http://localhost:5002/move",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["move"]
        },
        "description": "Moves a unit to the specified coordinates."
      },
      "response": [
        {
          "name": "Success",
```

```
            "status": "OK",
            "code": 200,
            "header": [],
            "body": "Unit movement queued."
          },
          {
            "name": "Error",
            "status": "Internal Server Error",
            "code": 500,
            "header": [],
            "body": "Error: {{error_message}}"
          }
        ]
      },
      {
        "name": "Attack Unit",
        "request": {
          "method": "POST",
          "header": [
            {
              "key": "Content-Type",
              "value": "application/json"
            }
          ],
          "body": {
            "mode": "raw",
            "raw": "{\n  \"attackerId\": \"unit_123\",\n  \"targetId\": \"
      unit_456\",\n  \"damage\": 50,\n  \"targetHealthLeft\": 50\n}"
          },
          "url": {
            "raw": "http://localhost:5002/attack",
            "protocol": "http",
            "host": ["localhost"],
            "port": "5002",
            "path": ["attack"]
          },
          "description": "Handles an attack from one unit to another with
      specified damage."
        },
        "response": [
          {
            "name": "Success",
            "status": "OK",
            "code": 200,
            "header": [],
            "body": "Attack queued."
          },
          {
            "name": "Error",
            "status": "Internal Server Error",
            "code": 500,
            "header": [],
            "body": "Error: {{error_message}}"
          }
        ]
      },
      {
        "name": "Kill Unit",
        "request": {
          "method": "POST",
          "header": [
            {
              "key": "Content-Type",
```

```
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"unitId\": \"unit_123\"\n}"
        },
        "url": {
          "raw": "http://localhost:5002/killUnit",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["killUnit"]
        },
        "description": "Removes a unit from the game."
      },
      "response": [
        {
          "name": "Success",
          "status": "OK",
          "code": 200,
          "header": [],
          "body": "Unit kill queued."
        },
        {
          "name": "Error",
          "status": "Internal Server Error",
          "code": 500,
          "header": [],
          "body": "Error: {{error_message}}"
        }
      ]
    },
    {
      "name": "Update Unit Health",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"unitId\": \"unit_123\",\n  \"healthLeft\": 75\n}"
        },
        "url": {
          "raw": "http://localhost:5002/updateHealth",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["updateHealth"]
        },
        "description": "Updates the health of a specified unit."
      },
      "response": [
        {
          "name": "Success",
          "status": "OK",
          "code": 200,
          "header": [],
          "body": "Health update queued."
```

```
        },
        {
          "name": "Error",
          "status": "Internal Server Error",
          "code": 500,
          "header": [],
          "body": "Error: {{error_message}}"
        }
      ]
    },
    {
      "name": "Start Game",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{}"
        },
        "url": {
          "raw": "http://localhost:5002/startGame",
          "protocol": "http",
          "host": ["localhost"],
          "port": "5002",
          "path": ["startGame"]
        },
        "description": "Triggers the game start and updates UI panels."
      },
      "response": [
        {
          "name": "Success",
          "status": "OK",
          "code": 200,
          "header": [],
          "body": "Game start queued."
        },
        {
          "name": "Error",
          "status": "Internal Server Error",
          "code": 500,
          "header": [],
          "body": "Error: {{error_message}}"
        }
      ]
    },
    {
      "name": "End Game",
      "request": {
        "method": "POST",
        "header": [
          {
            "key": "Content-Type",
            "value": "application/json"
          }
        ],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"teams\": [{}]\n}"
```

```
          },
          "url": {
            "raw": "http://localhost:5002/endGame",
            "protocol": "http",
            "host": ["localhost"],
            "port": "5002",
            "path": ["endGame"]
          },
          "description": "Ends the game and displays results for the
      specified teams."
        },
        "response": [
          {
            "name": "Success",
            "status": "OK",
            "code": 200,
            "header": [],
            "body": "Game end queued."
          },
          {
            "name": "Error",
            "status": "Internal Server Error",
            "code": 500,
            "header": [],
            "body": "Error: {{error_message}}"
          }
        ]
      }
    ]
}
```

Annexe 3: Full Installation Guide

Overview

The provided folder contains:

- **Unity game executables** for Windows (`My Project (1).exe`), macOS (`BuildMac.app`), and Linux (`BuildLinux.x86_64`), each with an embedded API server to control game functionality.
- A **Postman JSON collection** (`UnityGameServer.postman_collection.json`) with pre-configured API requests to interact with the game's API.
- A **Swagger YAML file** (`api-spec.yaml`) documenting the API's endpoints, parameters, and responses.
- A **Python demo script** (`real_simulation.py`) demonstrating API interaction with the game.

This guide covers:

- Running the appropriate Unity game executable for your platform and its embedded API.
- Importing and using the Postman collection to send API requests.
- Visualizing the Swagger YAML to explore API documentation.
- Running the Python demo script to interact with the API.
- Ensuring all prerequisites are met.

The API is hosted within the Unity game executable, so the game must be running to access the API at `http://localhost:5002`.

Prerequisites

Before proceeding, ensure the following requirements are met:

Software Requirements

- **Operating System**: Windows, macOS, or Linux (compatible with the respective Unity executable).
- **Postman**: Installed for the Postman collection. Download from Postman (free version sufficient).
- **Python**: Version 3.6+ installed for `real_simulation.py`. Download from Python. Ensure `requests` library is installed (e.g., `pip install requests`).
- **Web Browser**: For accessing Swagger tools (e.g., Google Chrome, Firefox).
- **Swagger-Compatible Tool**: Use Swagger Editor (online) or a local Swagger UI instance to visualize `api-spec.yaml`.
- **Unity Runtime Dependencies**:

  - Windows: DirectX, Visual C++ Redistributable (download from Microsoft if needed).
  - macOS: No additional libraries typically required.
  - Linux: Dependencies like `libGL` or `libSDL2` (e.g., `sudo apt-get install libgl1-mesa-glx libSDL2-2.0-0` on Ubuntu).

Network Requirements

- **Local Network Access**: The API is accessible at `http://localhost:5002`. Ensure no other application uses this port.
- **Internet Connection**: Required for downloading Postman, Python, or accessing Swagger Editor (optional for local Swagger UI).

Provided Files

Ensure the following files are present in the provided folder:

- **Unity Executables**:
  - Windows: `My Project (1).exe` and associated `Data` folder.
  - macOS: `BuildMac.app` bundle.
  - Linux: `BuildLinux.x86_64` binary and associated files.
- **Postman Collection**: `UnityGameServer.postman_collection.json`.
- **Swagger YAML**: `api-spec.yaml`.
- **Python Demo Script**: `real_simulation.py`.

Running the Unity Game Executable

The Unity game executable hosts its own API server. Select the executable for your platform and follow these steps:

1. **Locate the Executable**:
   - Find the appropriate executable in the provided folder:
     - Windows: `My Project (1).exe`.
     - macOS: `BuildMac.app`.
     - Linux: `BuildLinux.x86_64`.
   - Ensure associated files (e.g., `Data` folder) are in the same directory.
2. **Check System Compatibility**:
   - Windows: Install Visual C++ Redistributable if needed (Microsoft).
   - macOS: Right-click `BuildMac.app` and select **Open** if macOS flags it as quarantined.
   - Linux: Install dependencies (e.g., `sudo apt-get install libgl1-mesa-glx libSDL2-2.0-0`).
3. **Launch the Game**:

- Windows: Double-click `My Project (1).exe`.
- macOS: Double-click `BuildMac.app`.
- Linux: Run via terminal (e.g., `chmod +x BuildLinux.x86_64; ./BuildLinux.x86_64`).
- Allow firewall access if prompted to enable API communication.
- The API server starts at `http://localhost:5002`.

4. **Verify API Accessibility**:
   - Test the API with a browser or `curl` (e.g., `curl http://localhost:5002`).
   - A response (e.g., JSON data or status code) confirms the server is running.
5. **Troubleshooting**:
   - **Executable Fails**: Verify data files and runtime libraries. Re-download if corrupted.
   - **API Unreachable**: Ensure the game is running and port `5002` is free (check with `netstat -an` or `lsof -i :5002`).
   - **Port Conflict**: Stop conflicting applications.
   - **Firewall Issues**: Allow the executable through the firewall.

Importing and Using the Postman Collection

The Postman collection enables API interaction with the game. Follow these steps:

1. **Install Postman**:
   - Download from Postman and install.
   - Launch Postman and sign in (or use offline mode).
2. **Import the Collection**:
   - Click **Import** in Postman.
   - Select `UnityGameServer.postman_collection.json` from the folder.
   - Click **Open** to import into the **Collections** tab.
3. **Explore the Collection**:
   - View requests (e.g., `Start Game`, `Move Player`) in the **Collections** tab.
   - Each request includes:
     - **HTTP Method**: GET, POST, etc.
     - **Endpoint**: e.g., `http://localhost:5002/initializeMap`.
     - **Headers/Body**: Pre-configured (e.g., `Content-Type: application/json`).
4. **Send API Requests**:
   - Ensure the game is running.

- Select a request (e.g., `POST http://localhost:5002/initializeMap`).
- Edit the JSON body if needed (e.g., `{"numPlayers": 2}`).
- Click **Send** and review the response (e.g., `{"sessionId": "abc123"}`).

5. **Troubleshooting**:
   - **400 Bad Request**: Verify request body/parameters against `api-spec.yaml`.
   - **Connection Refused**: Ensure the game is running and port `5002` is open.

Visualizing the Swagger YAML

The Swagger YAML file documents the API in OpenAPI format. Use it to explore endpoints interactively:

1. **Locate the Swagger YAML**:
   - Find `api-spec.yaml` in the folder.
   - Optionally, review its structure in a text editor.
2. **Access Swagger Editor**:
   - Navigate to Swagger Editor in a browser.
3. **Import the YAML**:
   - Click **File > Import File** and select `api-spec.yaml`.
   - Alternatively, paste the YAML content into the left panel.
   - The right panel renders interactive API documentation.
4. **Explore the API**:
   - View endpoints grouped by tags (e.g., `Game`, `Player`).
   - Each endpoint includes:
     - **HTTP Method**: GET, POST, etc.
     - **Path**: e.g., `/game/start`.
     - **Parameters**: Query, path, or body.
     - **Responses**: Status codes (e.g., 200 OK) and schemas.
     - **Security**: Authentication details (if applicable).
5. **Troubleshooting**:
   - **Invalid YAML**: Check syntax with Swagger Editor or YAML Lint.
   - **Missing Endpoints**: Ensure `api-spec.yaml` matches the game's API.

Running the Python Demo Script

The Python demo script (`real_simulation.py`) demonstrates API interaction with the game. Follow these steps:

1. **Install Python and Dependencies**:
    - Ensure Python 3.6+ is installed (Python).
    - Install the `requests` library: `pip install requests`.
2. **Locate the Script**:
    - Find `real_simulation.py` in the provided folder.
    - Review the script in a text editor to understand its functionality (e.g., sending requests to `http://localhost:5002`).
3. **Run the Script**:
    - Ensure the Unity game is running to host the API.
    - Open a terminal and navigate to the folder containing `real_simulation.py`.
    - Run the script: `python real_simulation.py`.
    - The script will execute API calls (e.g., initialize game, move player) and display responses.
4. **Troubleshooting**:
    - **Module Not Found**: Install missing libraries (e.g., `pip install requests`).
    - **Connection Errors**: Verify the game is running and port `5002` is accessible.
    - **Script Errors**: Check the script's code for compatibility with your Python version or consult `api-spec.yaml` for correct API usage.

Conclusion

This guide provides instructions for running Unity game executables (`My Project (1).exe`, `BuildMac.app`, `BuildLinux.x86_64`) with an embedded API, using the `UnityGameServer.postman_collection.json` collection, visualizing `api-spec.yaml`, and running `real_simulation.py`. By meeting the prerequisites and following these steps, you can control the game via its API. Refer to troubleshooting section*s for assistance or consult additional resources as needed.