COMP390

2021/22

Board Game AI Programming: Simplified Civilization Game

Student Name: Boxuan Zhu

Student ID: 201521571

Supervisor Name: Ullrich Hustadt

DEPARTMENT OF

COMPUTER SCIENCE

University of Liverpool

Liverpool L69 3BX

Dedicated to Ravenfield

Acknowledgements

I want to thank my parents, my roommate, and my two mentors.

In particular, I would like to send this statement to my supervisor, Dr Hustadt:

Sincerely thank you for your guidance to my study and work over
the past year!

During this year of study, I have truly experienced the whole process of taking a project from plan to finished product, which is extremely precious to me.

I am reminded of what you said to me last year "promise is simple", and now I have a deeper understanding of this saying - only with hard work can people deliver on their promises.

I hope that I can fulfill my promise to my ideals through hard work in my future life and work.

COMP390

2021/22

Board Game AI Programming: Simplified Civilization Game

DEPARTMENT OF

COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Table of Contents

Abstract

This paper focuses on developing a 3D strategy game with artificial intelligence elements - a simplified version of the civilization game. Based on Unity engine and C#, the project builds a turn-based artificial intelligence decision-making system by using a variety of modern programming tools including PyTorch and ML-Agents. Ultimately, the game allows human users to play against computer players driven by artificial intelligence models, and to decide winners and losers on a map containing a large number of hexagonal cells.

1. Introduction

This project is a strategy computer game developed by the Unity engine based on the classic strategy game "Civilization".

The project includes a game mode that allows human players and AI players to operate virtual empires in the game, competing with each other to win the final victory. In this mode, human players are allowed to adjust the level of intelligence (game difficulty)

of the AI players. Also, human players can choose to just watch the computer players play and not participate in the game.

In the game, players will start the game on a board containing a large number of hexagonal cells - starting with the initial cells located on the edge of the map. The game defines three game entities (settler, soldier, city), three game behaviors (production, war, movement), and three resources (money, attack bonus, defense bonus). Players use cities to acquire resources and produce soldiers and settlers, use settlers to build cities and capture cells, and wage war with soldiers enhanced by bonus. In the end, the side that occupies all or the most cells within the specified time wins.

To complete this project, a complete turn-based game system, a set of game interfaces, an AI model (composed of three subsystems corresponding to three entities), and a set of additional game assets (art resources and music resources) were successfully developed. The project is developed based on the Unity engine and presented in 3D. In addition, the project was tested and evaluated to improve its completion.

Compared with the original setting of Detailed Proposal, the number of entities, resources, and behaviors of the official version of the project are clearly defined as three, which ensures the completeness of the game rules and the simplicity of the game logic. In addition, the AI model in the original design had only one system, and its function and composition were not well defined. In the official version, this AI model is clearly defined and developed as a reinforcement learning model with three subsystems.

2. Aims & Objectives

The aim of this project is to develop a strategy computer game with artificial intelligence players.

The aim means that after the project development is completed, the project should have the following characteristics: 1. The project is a 3D strategy game on the PC platform. 2. The game includes 1 game mode that allows users and AI players to play and compete on various elements of the virtual world, and the player with the largest virtual world territory at the end of the game wins. 3. The project has an AI model

that can be adjusted to play against human players. 4. The user's identity can be switched: the user can participate in the game or observe the AI player's battle as an observer.

To implement this aim, the project needs to implement the following objectives:

Maintainability: A specific computer language should become the development language of the project. It should meet the needs of object-oriented programming to ensure that the game can be structured and updated efficiently. To test this objective, multiple test components or game subsystems will be deployed for analysis.

3D-based development: A specific 3D game engine will undertake the main task of development, ensuring that various codes and art resources of the game are efficiently utilized. To test this goal, the game will use this engine for load testing.

Gameplay: The completed project (game) should allow users to fully experience the game, and the experience process includes a main loop and multiple sub-loops. The main loop is the loop of the game from start to finish (restart). Sub-cycles refer to cycles such as game turns and game resource cycles. Also, the game will be turn-based. In each round (multiple turns form a round), a player needs to perform a series of actions and move on to the next round after the other players have completed their actions. To verify the achievement of this goal, a test will be held on a third-party evaluation platform to objectively analyze the gameplay of the project.

AI: The game should contain AI players who can make decisions and act autonomously, and who also act according to the rules of the game. AI should share the same behavior types, resource types, and entity types as humans. The AI models that drive these players will be built on a specific AI technology. To verify the achievement of this objective (AI effectiveness), a specific set of settings will be deployed into the game, which allow the user to adjust the level of intelligence of the AI and allow the user to observe the battle between AIs of different intelligence as an observer. The above setup will show the difference in decision-making ability between different AIs and demonstrate the autonomy of AIs - they do not rely on input from human players.

Others: The project may have some additional features, such as soundtracks and

other game assets. As verification, these assets will be recorded and displayed.

3. Background

In order to further provide sufficient knowledge for developing and understanding the project, it is necessary to carry out background knowledge investigations in the relevant areas of the project. In this project, all required background knowledge can be divided into programming and technical platform knowledge, gameplay design knowledge, artificial intelligence knowledge, and auxiliary resource development (such as art and music resources) knowledge.

Gameplay design: the core gameplay and game loop of the game

The design results of the gameplay play a guiding role in the development of the game. All programming, art, and music development serve the gameplay, which ultimately embodies the gameplay into the actual game [1].

In terms of gameplay, this project is a simplified version of the "Civilization" game. This means that the project is a strategy game with 4X elements ("eXplore, eXpand, eXploit, eXterminate"). Although the game system of this project will be streamlined, the designed "entity-behavior-resource" game framework will still make the gameplay of the project contain 4X elements. More importantly, like "civilization", the core of this project's gameplay is also a turn-based system. Turn-based operations will run throughout the game. Each player has to take a turn to perform some action. After the operation is over, wait for the operation of other players to end before entering the next round (another set of turns) [2].

In addition, strategy games, including "Civilization" and this project, can use "loops" to describe their processes. The first is the main loop that represents the overall flow of the game: (1) dispatch units to explore. (2) Occupy neutral cells and build cities. (3) Compete with other players and conquer hostile cells. (4) Harvest resources on occupied cells and produce entities (settler and soldier). (5) Repeat actions (1)-(4) until the game is over.

The main loop is composed of some sub-loops, such as the productivity-tech-

population cycle in the "Civilization" series: "population" produces "productivity", and "productivity" is used to buy "tech" and various buildings, these things will eventually accelerate the growth rate of the "population", thereby realizing a positive cycle [3]. Different from the "civilization" game, this project deploys some independently developed sub-loops constructed according to the "entity-behavior-resource" framework. Like the "money-settler-city" cycle: Players use money to buy settlers, and the settler builds cities, and the city reaps more money.

Game program: scripting language and unified engine

Scripting languages simplify the traditional write-compile-link-run process, making games more efficient. This project uses the currently popular C# as the game scripting language.

The Unity engine is a reusable and rapid game development platform that reduces the workload of game development. The Unity engine integrates a large number of development tools such as renderers, physics engines and scripting engines [4]. By using the Unity engine, developers can centralize all development tasks on one platform, greatly improving the efficiency of game development and intuitively connecting gameplay and game programs.

Artificial intelligence in games: AI players in civilization games

In most 4X games, computer players contribute significantly to the gaming experience. In order to make the behavior of computer players more complex and effective, artificial intelligence technology needs to be introduced into game development [5]. Correspondingly, games can also be a testing ground for advanced AI—researchers can use games to train AI to deal with real-world problems [6].

With the advancement of technology, AI in games began to introduce behavioral models such as decision trees and state machines, which further absorbed research results in the field of machine learning [7]. For example, in "Civilization 4", the game AI can adjust the development strategy in time to make up for the lack of buildings or troops after analyzing the difference in military strength and territorial area [8].

In this project, computer players will use reinforcement learning technology to make decisions - by dividing the decision-making system into three subsystems corresponding to three game entities, each subsystem will have an independent AI model trained by reinforcement learning algorithms. They will work together to drive computer players against human players, ultimately improving the gameplay.

4. Design

Game Rules Design:

This part will introduce the project at the game level, the design details and implementation details will be clarified in the following parts.

Game process overview: Four players operate their own virtual empire on a virtual map with a large number of cells. The operation mode includes three game behaviors: movement, production, and war. The game process is divided into a large number of "rounds", and each "round" has **four** "turns". In different "turns", different players need to act within the constraints of the rules and definitions, and end turn.

Game end condition: One of the four players occupied all the cells in the map by operating the virtual empire.

Game entity definition:

"Settler": The entity that builds the city entity, which can be moved.
"Soldier": The entity that waged war. It can destroy the opponent's soldier entity, city entity and settler entity. can be moved.
"City": The entity that produces the settler and soldier entities. It harvests the cell's resources for the player and uses those resources to produce entities. Unmovable.

Game property definition: In this paragraph, private properties refer to properties that can only be accessed by one player, and public properties refer to properties that can be accessed by all players.
"Money": The player's private property (private resource), which is used by the city entity to produce the settler entity (4 money) and the soldier entity (2 money). Every time a player captures a new cell, that cell increases the player's "money" value by a

certain amount each round (4 turns). This increase will stop when the cell is neutralized or captured by another player.
"Attack bonus": A private attribute of the player (private resource), which is used by the soldier entity to enhance attack capabilities. Every time a player occupies a new cell, the cell will increase the player's "attack bonus" by a certain amount at one time. This increase will stop when the cell is neutralized or captured by another player.
"Defend bonus": A private attribute of the player (private resource), which is used by the soldier entity to enhance defenses. Every time a player occupies a new cell, the cell will increase the player's "defend bonus" by a certain amount at one time. This increase will stop when the cell is neutralized or captured by another player.
"Force": The public attribute of a soldier, used to measure the combat capability (attack or defense) of a soldier entity. soldier entities can increase the "force" value by annexing other friendly "soldiers" and reduce the "force" value through war.
"Wealth": The public attribute of the cell (public resource), which is used to measure the wealth of a cell. This attribute is a set consisting of an integer and two floats that refer to "money", "attack bonus" and "defend bonus" that can be enhanced for players.
"Attribution": the public attribute of the cell, which is used to indicate the owner of a certain cell. At the beginning of the game, all cells are neutral - they do not belong to any player.

Introduction to game behavior:

"Movement": This action takes effect on the settler entity and the soldier entity. The behavior consists of three elements - the origin cell, the destination cell, and the entity. The effect of this behavior is to transfer the entity from the origin cell to the destination cell. Any entity may initiate this behavior only once per turn. When any entity initiates this behavior, the distance between the starting cell and the destination cell of the behavior cannot exceed 2 units. Additionally, multiple settler entities are allowed to exist on the same cell, and once the soldier entities are brought together, they are merged into a new soldier entity with a "force" value equal to the original soldier entity The sum of the "force" values.

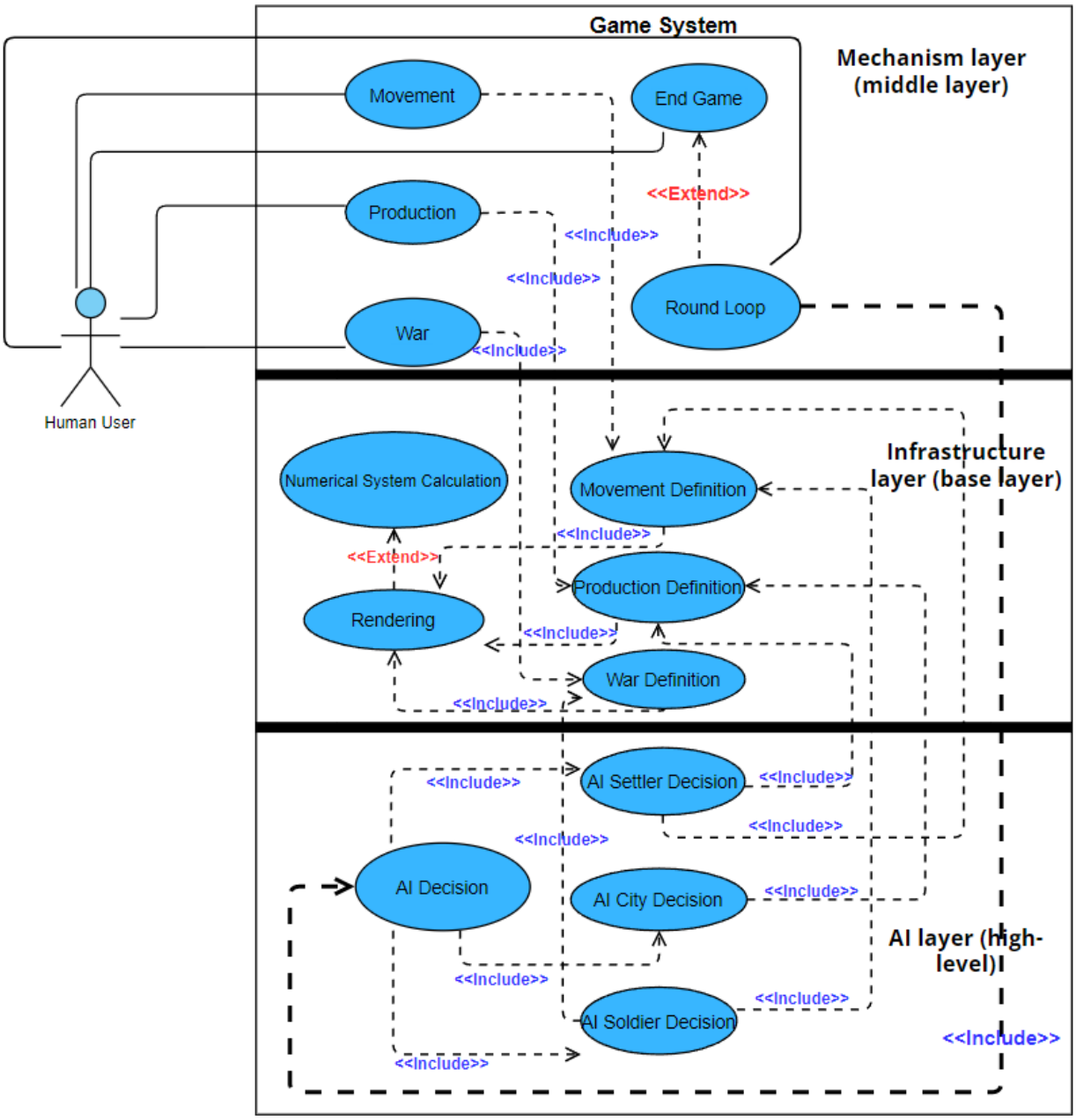
"Production": This behavior takes effect on the city entity and the settler entity. The behavior consists of two elements - the entity, the cell in which the entity is located. The effect of this behavior is (a) if the initiating entity is a city entity, then on the cell it is in, a new settler entity or soldier entity will be spawned. (b) If the entity is a settler,

then on the cell it is in, a new city entity will be generated, and the original settler will be deleted. In particular, if a new city entity is created on a neutral cell, the ownership of the cell is changed to the name of the city owner. The production behavior of the settler entity is the only way to capture a cell in the game.

"War": This behavior applies to the soldier entity. This behavior is an extension of the behavior movement - the soldier entity must first initiate the movement behavior to move to another cell that contains the opposing entity. After that, the war action will be automatically initiated, and the player who initiates the movement is called the attacker, and the other side is called the defender. The behavior consists of four elements - the attacker soldier entity, the defender soldier entity, the defender city entity, and the defender settler entity. After this action is initiated, a combat power comparison based on the player's "force" attribute value will be initiated.
For the attacker, combat power x=("force”) * ("attack bonus") + 0.
For a defender whose behavior cell has a city entity, combat strength y=("force") * ("defend bonus") + 1.
For a defender whose behavior cell does not have a city entity, y=("force") * ("defend bonus") + 0.
If x>y, all entities active on the cell by the defender will be removed (soldier, city, settler), and the cell will become neutral. If y>x, the attacker's soldier entity will be deleted.

Game software structure design

This project consists of three layers (subroutines) -
Infrastructure layer (base layer): rendering routine, behavior definition routine and numerical system routine
Mechanism layer (middle layer): a set of turn-based game system programs
AI layer (high layer): Reinforcement learning algorithms and models that drive computer players



In this hierarchical system, the middle layer must be built on top of the base layer, without the construction of the base layer, the middle layer will not be able to operate. In the same way, the high layer must be built on the first two, and the absence of any layer will cause the artificial intelligence to fail to function properly.

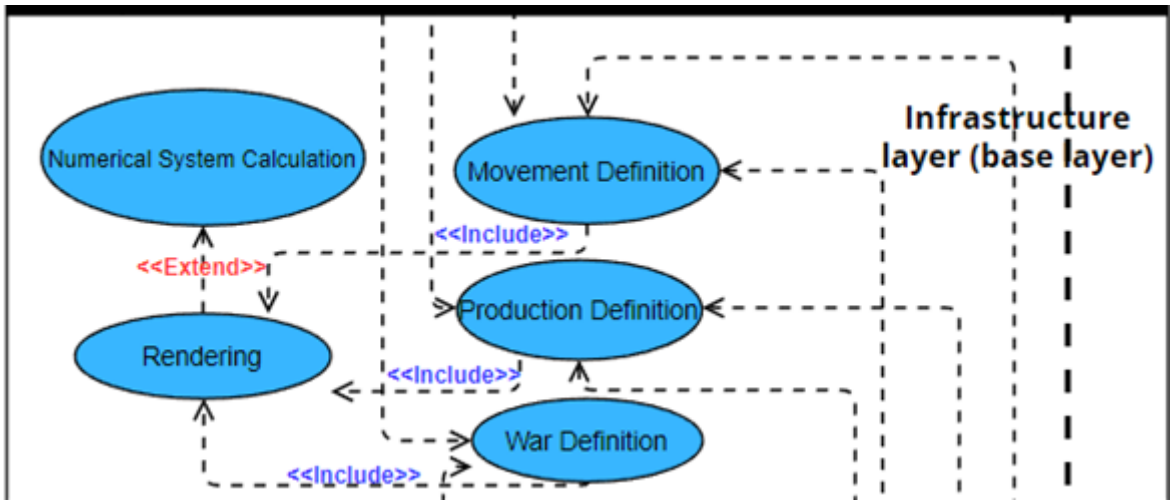
The data structures used in this project are mainly arrays and array-like custom data structures. First, in order to build the numerical system of the project, high-dimensional mutable arrays are defined. Second, during project development, a data structure type called "cell" was defined to store the cells randomly generated by the map generation algorithm. "cell" is essentially an array with several properties.

This project contains a series of algorithms, each layer has one or more specific algorithms to drive the subroutines within the layer.

First, the base layer is constructed by a set of rendering algorithms based on hexagon generation, a set of behavior algorithms that describe three game behaviors, and a set of numerical calculation algorithms.

Secondly, the middle layer is constructed by a set of turn-based algorithms that combine the game engine and the base layer. The algorithm is responsible for the round of the game and calling the behavior algorithm and numerical calculation algorithm of the base layer. In addition, the rendering algorithm of the base layer will render (update) game elements on the UI based on the results of the behavioral algorithm and the numerical calculation algorithm. In order to connect with the high-layer, the behavior algorithm in the base layer will be repackaged, and the interface will be exposed by the middle layer for high-layer and human users to call freely.

Finally, the high-layer will be built by a suite of reinforcement learning algorithms, where the AI model will collect in-game data to make decisions and make decisions by invoking behavioral interfaces. After the decision loop is over, the AI model will get feedback to correct itself.



Infrastructure layer (base layer):

This layer is the basis for the game to be fully executed. In this project, this layer (subroutines) mainly consists of three parts - rendering routine, behavior definition routine and numerical system routine.

(1) The first point, for the rendering routine, (1a) its first task is to render the entire

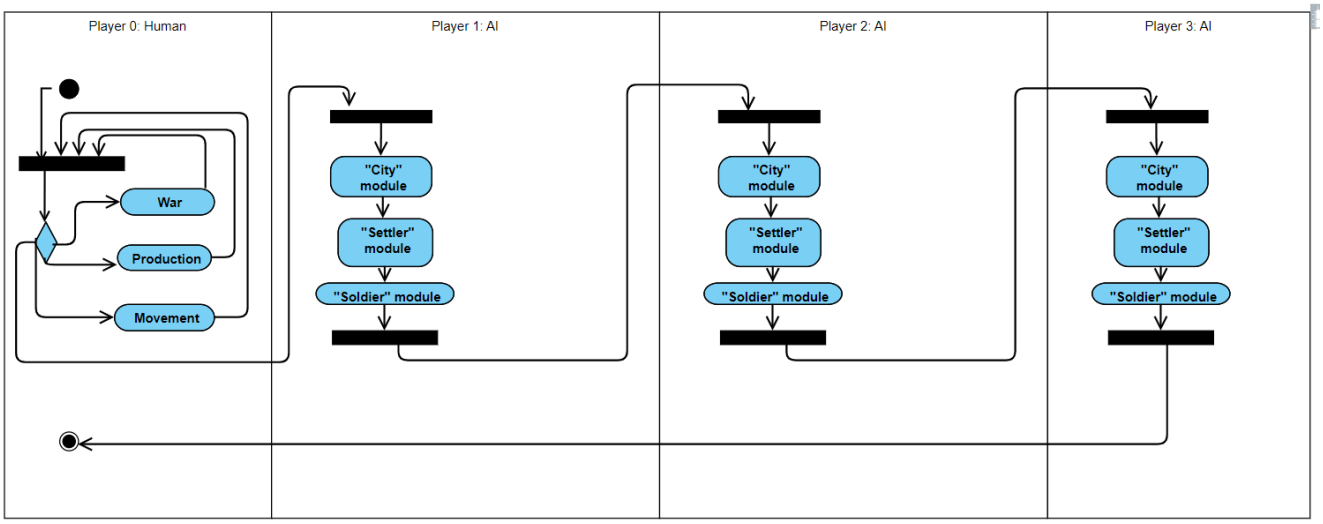
map at the beginning of the game, the size of the map will be specified by the player, and the properties of each cell in the map will be random. (1b) The second task of the rendering routine is to correctly represent all elements of the game that need to be displayed on the game screen, including all living game entities (settlers, soldiers, cities) and the game's numerical system. The various values that are output. It should be noted that these output values include the public attributes of each cell and the private attributes of each player. (1c) In addition, the rendering routine also ensures that the game screen is updated in time when the elements in the game change: for example, after a war, the following changes may occur to the game elements - the color change of the cell in which the war occurred (cell). change of ownership), destruction of soldiers or city entities. These changes must be reflected in the screen in time.

(2) The second point, the behavior definition routine of the base layer needs to define and standardize the three behaviors of the game to allow these behaviors to be repeatedly called by players (human and computer players). (2a) For behavior movement, the behavior definition routine needs to cooperate with the rendering routine to move an entity from its current position to a new position, and update the properties in the numerical system that may need to be modified

(For example, the movement of the soldier will bring the change of the attribute "force" of the cell at the starting point and the cell at the end point). (2b) For behavior production, the behavior definition routine needs the cooperation of the rendering routine, responds to the construction request of a specific entity (settler or city), and deploys a new entity (city, settler, soldier) in place). After this, the behavior definition routine will instruct the value system routine to change some attributes of the cell where the event is located (the ownership of the cell or the force value of the cell) and some attributes of the player, and call the rendering routine again to complete any possible changes - such as the settler's production behavior is equivalent to declaring sovereignty on a cell, the rendering routine will change the color of that cell. (2c) For war behavior, behavior-defining routines require the cooperation of numerical system routines. The behavior-defining routine can determine the victor of the war by comparing the force values that judge both sides of the war. After that, the behavior definition routine will invoke the rendering routine to delete the lost soldier and city (if the defender of the war has a city on the cell).

(3) The third point, the numerical system routines of the base layer are composed of multiple high-dimensional variable arrays and calculation codes scattered in other routines. (3a) The first array is the "entity owner" array, which is responsible for

recording the ownership of all game entities active on the map. Whenever new entities are born or old entities are deleted, this array is written with new data by some specific calculation code to ensure that each player has the correct number and names of entities. (3b) The second array is the "economy" array, which is responsible for recording the economic situation of each player, including his "money" attribute, "offensive bonus" attribute (this attribute provides offensive rewards in war), "Defensive bonus" attribute (this attribute provides defensive bonuses in war). Each new cell will give players bonuses to these three attributes as their virtual empire expands. For example, a cell has "2 money, 0.01 attack bonus, 0.02 defense bonus", which means that after a player captures the cell, the numerical system will be updated as follows: The player will get an extra 2 "money" points at the end of each round (4 turns). The player will get bonuses permanently - in battle, the force value of all of the player's soldier entities will inflate to 1.01 times the original value when attacking, or 1.02 times the original value when defending. (3c) The third array is an auxiliary array used to calculate the "money" income of each player after the end of each round (4 turns), consisting of integer variables.



Mechanism layer (middle layer):

In this project, this layer (subroutine) makes the game playable. It contains a turn-based game system routine. The goal of this routine is to (1) build a game flow that defines the start, end, and loop of a turn (2) further wrap the base layer behavior definition routines so that each behavior has a direct and independent callable Interfaces - These interfaces ensure high cohesion and low coupling of the project. (3) The program can respond to the operation request of the human or computer player

at any time and call the packaged behavior interface and the functional interface of the game engine program according to the type of the operation request.

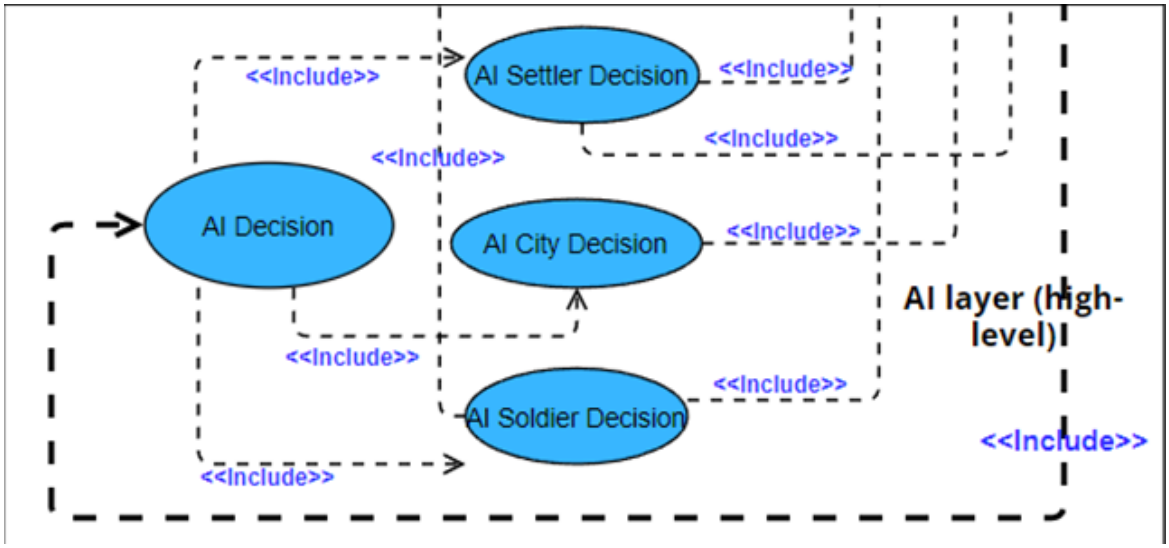
(1) In order to construct the game flow, the following definitions of "turn" are declared: (1a) Cyclic: There are a large number of "turns" in the flow, and the "turns" have the same structure. Essentially, a collection of "turns" results from repeated calls to the same routine. (1b) Start ability: the start of each "turn" means the end of the previous "turn". In any case, the end of the previous "turn" must result in the beginning of a new "turn". (1c) End ability: each "turn" must be allowed to be ended at a certain moment, and at other times, the "turn" cannot be ended.

As can be seen from the above definition, in terms of turns, the rights that AI models (high layer) and human users essentially have are the rights to end the turn. Furthermore, in the execution of each turn, the game flow is effectively suspended, and it cannot arbitrarily advance to the next turn. This pausing gives the AI model (high layer) and the human user the opportunity to acquire information and perform actions.

(2) To further wrap the behavior-defining routines of the base layer, a series of smaller subroutines are deployed to the middle layer, including: (2a) Three subroutines corresponding to the three behaviors are deployed to the middle layer, their main goal is to call the three behavior-defining routines of the base layer, and use a mutex structure to ensure that calls to multiple behaviors do not conflict. At the same time, they also verify that the player who initiated the action is eligible (if the player cannot initiate the action at this time, the game state is rolled back). The new three subroutines can be called directly by human users or computer players. (2b) An additional behavioral interface routine responsible for capturing human user commands is deployed into the game engine's frame generation routine (Update() function in the C# context), a structure designed to allow capturing operations Cover the whole process of the game - Humans may use external input devices to issue commands at any time in the game. In addition, since AI model-driven computer players do not need external input devices, they will directly call behavioral interface routines in the middle layer. (2c) Further, a special subroutine to end the current turn is deployed to the middle layer, which will wait for calls from both the human player (via routine 2b) and the computer player. Once the routine is invoked, it settles and updates the number of turns and player economics and starts a new turn when all work is done.

(3) As described in (2), for human user requests, routine 2b will follow up and process

the request in real time. For computer players driven by the AI model, the middle layer will allow them to directly call routine 2a (routine group) and actively call routine 2c to end the turn after the AI's decision-making routine is over.



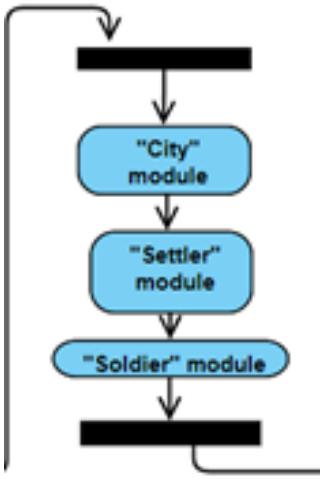
AI layer (high layer):

In this project, this layer (subroutine) will drive computer players against human players. In order to reduce the complexity of the algorithm and the final model, improve the robustness of the model and improve the development efficiency, the artificial intelligence layer of this project is divided into a set of reinforcement learning algorithms and AI models generated under the algorithm training.

Except for some basic rules that have been entered, the training process of the rest of the AI model will be real-time. This means that computer players will learn and optimize themselves as they play. In addition, due to the characteristics of reinforcement learning algorithms, the training process of AI models will not rely on the "training set-validation set" mode. This means that the AI can directly participate in the game on randomly generated maps without pre-training on dedicated training maps.

This layer can be divided into three parts (corresponding to the three kinds of entities in the game). Each part contains a sub-algorithm and sub-model. The algorithm structure of each part is similar, but the specific parameters and final purpose are different. Ultimately, three relatively simple models together form a multi-agent network that achieves intelligence through self-organization and collaboration.

Whenever the game enters a new turn that requires computer player action, the following parts of this hierarchy will be invoked in sequence to complete the computer player's decision-making process:



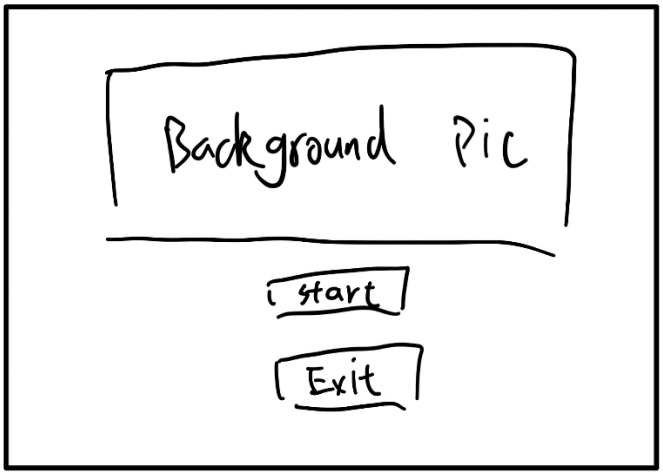
(1) "city" module: responsible for manufacturing settler entities and soldier entities according to the current economic situation and give priority to the construction of settler entities when conditions permit. This module will traverse all city entities owned by the player to ensure that the locations of new entities are evenly distributed.

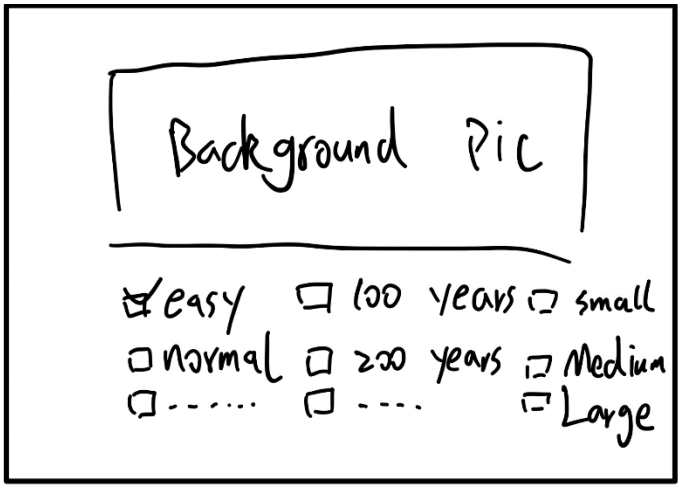
(2) "settler" module: This module will traverse all settler entities, and try to let each settler find the richest cell it can reach, and occupy this cell after arriving.

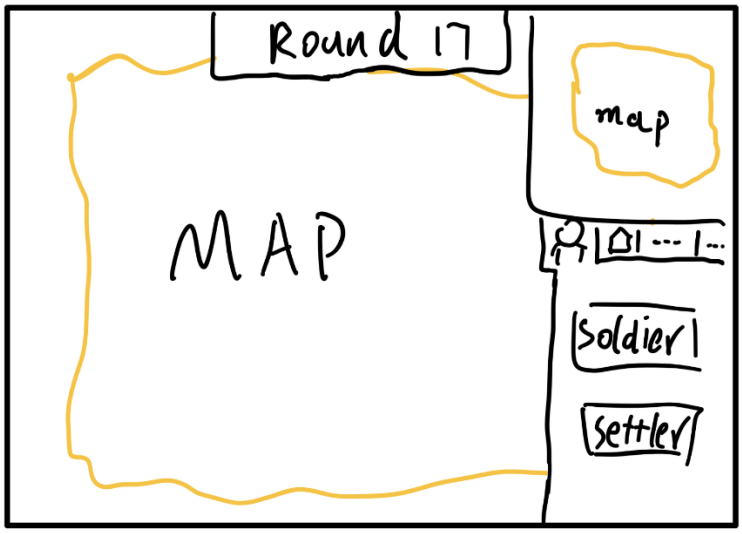
(3) "soldier" module: This module will traverse all soldier entities and mark all non-own city entities on the map. Each soldier entity will be assigned a target and will try to reach the target in order to start a war (destroy the city, neutralize the ownership of the cell).

User Interface Design and Navigation

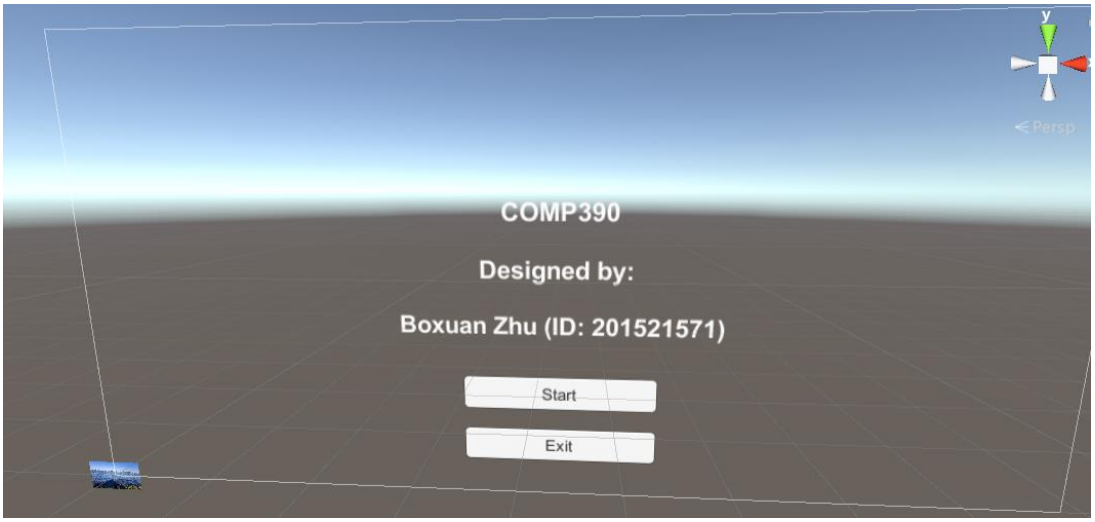
Sketches

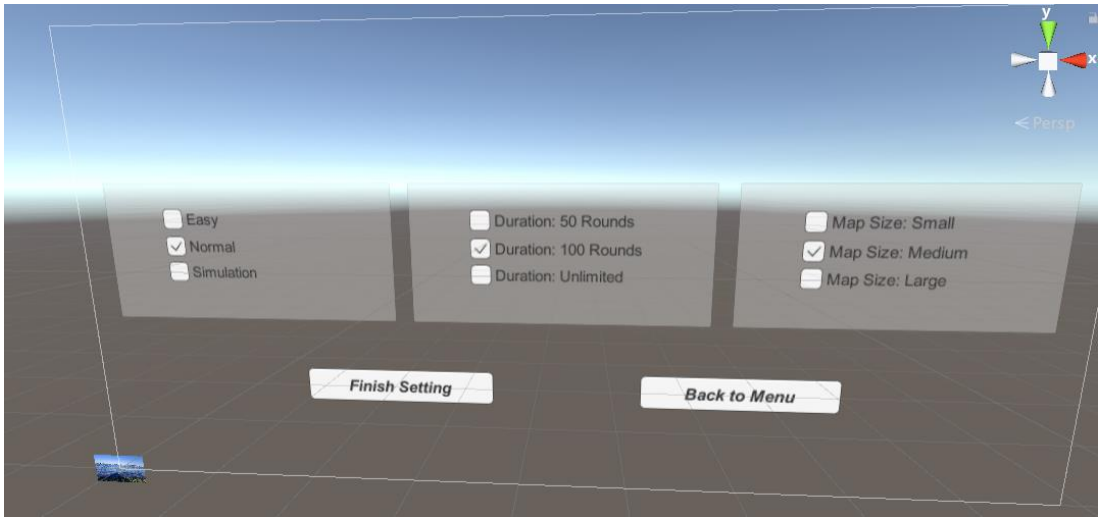


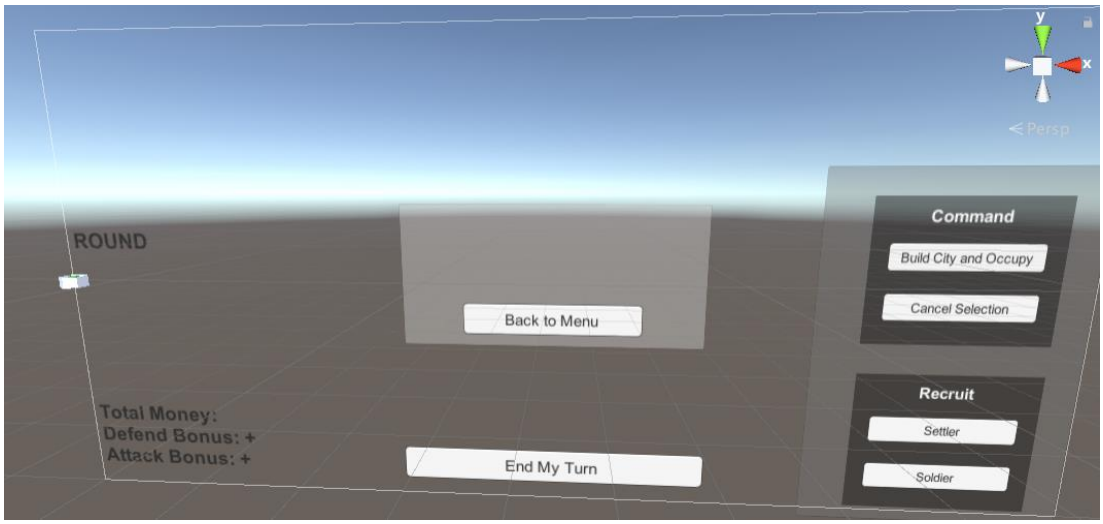




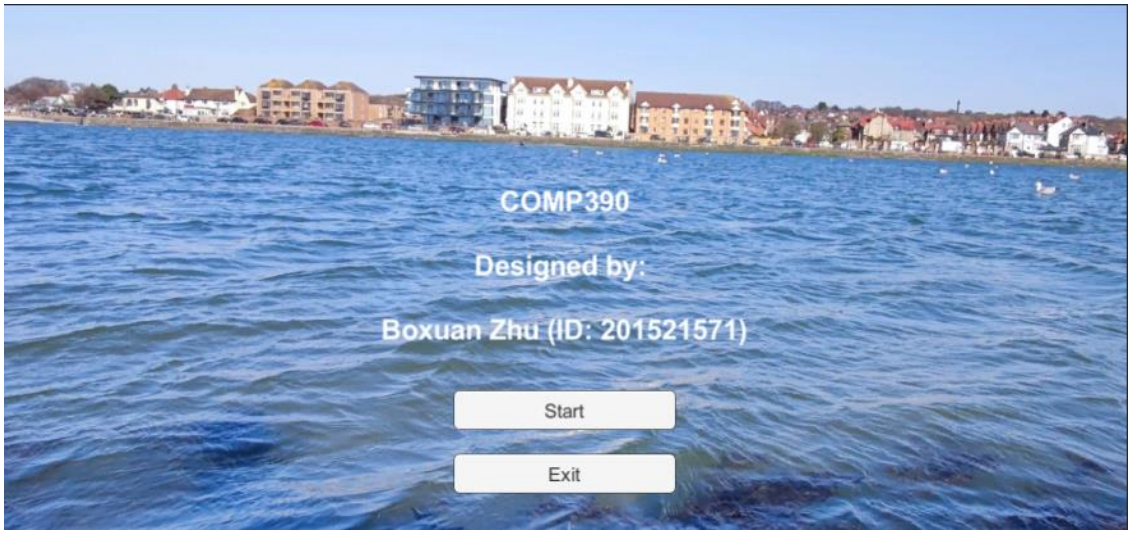
Wireframes & Mockups







Prototypes & Navigation



5. Implementation:

This project is implemented through Unity game engine and C# language.
The AI part of this project is implemented through Unity's ML-Agent toolkit and PyTorch toolkit - these toolkits allow developers to use C# statements to build reinforcement learning models in the Unity environment.

Implementation: the programming implementation of the basic elements of the game

Player, Entity and Property Implementation:

(1) The system describes the player through a combination of multiple arrays and functions: (1a) The high-dimensional variable string array "powerbelong" is used to record and track all properties of all players (three entities) (1b) The function PowerAssign(name, playerindex, category) is used to register an entity under the name of a player. The specific way is to add the name of the entity to powerbelong[playerindex][category]. (1c) The function PowerResign(name, playerindex, category) is used to deregister an entity to be deleted from a player's name. The specific way is to delete the name string under powerbelong[playerindex][category]. (1d) The high-dimensional array "score" is used to record the economic situation of each player (money, attack bonus, defend bonus) (1e) The high-dimensional array "citycluster" is used to record the number of city entities owned by each player, thereby tracking players The "money" value to increase per round.

(2) Realization of game ending conditions: (2a) The system determines whether the game should end by tracking the length of the sub-array 0 of the array "powerbelong" (essentially the number of cities) - once the number of cities under a player's name is equal to the number of cities in the map The number of cells, then the game is over, the winner is the player (2b) When the human user chooses to add a time limit to the game, the system will track the number of rounds to determine whether the game should end. In addition, the system will determine the winner of the game by comparing the number of city entities owned by each player.

(3) Game entity definition and implementation: In the development process, there are two feasible technical routes: one is to use OOP (object-oriented programming) technology to abstract three game entities into three C# classes. The other is "decentralized definition", which spreads the properties and functions describing the entity among other programs. The project adopts a "decentralized definition". The reason is that in the later stages of the game, there will be a large number of game entities on the map - at this point, if the game system is required to track every entity defined with a C# class, it will put a burden on the update and rendering system. Therefore, the three entities of the game are defined separately in turn.

(3a) settler:

Property "name" - All instances of this class, their name and owner will be registered in the array "powerbelong" for tracking.

Property "location" - the name of each settler entity will be registered directly in the

cell it is in (property "seedname"). By calling both the cell's sequence number and the "seedname" property, the location of each settler entity can be tracked and referenced in time.

Attribute "Owner" - The number of the owner of each settler entity will be registered directly in the cell where it is located (attribute "seedence").

Initialization function: The settler entity can and can only be created (initialized) by the city entity. The initialization function is the "SeedDeploy(objname, playerIndex)" function under the "CellsGrid" script. The mode of operation of this function will be explained later.

Position change function: The settler entity can be moved by human or computer players. The position change function is the "UnitMove(objname, indexPos, playerIndex)" function under the "CellsGrid" script. The mode of operation of this function will be explained later.

(3b) soldier:

Property "name" - All instances of this class, their name and owner will be registered in the array "powerbelong" for tracking.

Attribute "location" - The name of each soldier entity is registered directly in the cell it is in (attribute "existernam"). By calling the cell's serial number and the "existernam" property at the same time, the location of each soldier entity can be tracked and referenced in time. What is different from the settler entity is that the location of the soldier entity is exclusive - that is, there can only be a soldier entity belonging to one player on a cell, and any soldier entity belonging to another player entering the cell will cause war behavior.

Attribute "Owner" - The number of the owner of each soldier entity is registered directly in the cell it is in (attribute "existence").

Attribute "force" - The initial force value of each soldier entity is the integer 2. Soldier entities are allowed to merge with each other. The force value of the new merged entity (which will inherit the name attribute of one of the parties in the merge process) is equal to the sum of the force values of the entities participating in the merge. The force attribute can be temporarily changed by the player attributes "attack bonus" and "defend bonus".

Initialization function: Soldier entities can and can only be created (initialized) by city entities. The initialization function is the "ArmyDeploy(objname, playerIndex)" function under the "CellsGrid" script. The mode of operation of this function will be explained later.

Position change function: Soldier entities can be moved by human or computer players. The position change function is the "UnitMove(objname, indexPos, playerIndex)" function under the "CellsGrid" script. The mode of operation of this function will be explained later.
War Function: Only soldier entities can initiate war. The "WarWithIn(defindex, invasindex)" function under the "CellsGrid" script is called when a soldier is moved to a non-own cell. The player who initiates the move (the owner of the cell numbered invasindex) is called the attacker, and the other side will automatically become the defender. The mode of operation of this function will be explained later.

(3c) city:
Property "name" - All instances of this class, their name and owner will be registered in the array "powerbelong" for tracking.
Attribute "location" - the name of each city entity will be registered directly in its cell (attribute "occupyernam"). By calling the cell's serial number and the "occupyernam" property at the same time, the location of each city entity can be tracked and referenced in time.
Attribute "Owner" - The number of the owner of each city entity will be directly registered in the cell where it is located (attribute "occupyer").
Harvest function: The city entity has the ability to harvest resources (money, attack bonus and defend bonus) for the player. This function is divided into multiple pieces of code and deployed in the round settlement function "RoundendClick()" in the middle layer. Whenever the turn in the round is ready to end, the "RoundendClick()" function will calculate the total sum of the attribute "moneypoint" in all the city entities owned by the current player, so as to obtain the increase value of "money" in this round. Finally, this value is registered into the array "score". On the other hand, for "attack bonus" and "defend bonus", every time a player occupies a new cell, the "attack point" attribute and "defend point" attribute under the cell are read, and are added separately to the array "score". The bonus provided by this cell will persist in the "score" array until the cell is neutralized (or captured by another player) due to war.
Production function: city entities have the ability for players to produce new settler entities and soldier entities. When the player tries to instruct the city entity to spawn a new entity, the "Deploy" function group under the "CellsGrid" script is called - this function group also points to the initialization functions of the settler entity and the soldier entity. Therefore, there is such a logical relationship: the production function of

the city is equivalent to the sum of the initialization functions of the settler entity and the soldier entity.

Behavior Implementation:

(4) "Production" behavior: production function / settler entity initialization function / soldier entity initialization function
The function uses the "objname" parameter to locate the city entity that initiated the initialization request, and the "playerIndex" parameter to determine the owner of the new entity. After determining where the new entity will be deployed, a replica of the prefab made with Unity's 3D modeling tools will be deployed into the game. After the rendering process is complete, the "SeedDeploy(objname, playerIndex)" function (or the "ArmyDeploy(objname, playerIndex)" function) will return a parameter with the name of the new entity and allow "powerbelong" to register this parameter.

(5) "Movement" behavior: position change function
This function uses the "objname" parameter to locate the settler or soldier entity that initiated the position change request, and the "playerIndex" parameter to determine the owner of the entity. After using indexPos to determine the destination cell of the entity, the three-dimensional coordinates of the entity will be modified by the system to the coordinates of the destination cell, thereby completing the position change.

(6) "War" behavior: war function
This function uses defindex and invasindex to locate the position of both sides of the war, the number of entities participating in the war and the combat power. At the beginning of the function, a set of if statements are deployed to determine whether the city entity is involved in the war - if there is a cell entity on the defender's cell, the original "force" value of the defender's soldier entity is increased by 1 . Then, the "force" value of both sides begins to enter the calculation process: (the total "force" value of the attacker) = (the initial "force" value of the soldier entity participating in the war) * (the "attack bonus" of the attacker's player). (total "force" value of the defender) = (initial "force" value of the soldier entity participating in the battle) * ("defend bonus" of the defender player). After comparing the total "force" value of both sides, the winner of the war will be determined. After that, the end of the war can be divided into two situations: (a) the attacker wins and the defender loses (b)

the attacker loses and the defender wins.
If (a) happens, the victor's soldier entity will retain the residual "force" value and neutralize the cell where the war occurred. The soldier entity, city entity, and settler entity that the loser exists in the cell will all be deleted (logged out). The loser will lose the cell and lose the "money", "attack bonus" and "defend bonus" support provided by the cell.
If (b) happens, the victor's situation can be further subdivided (bα) the victor loses the soldier entity, but preserves the city entity. In this case both the winner and loser soldier entities will be deleted. (bβ) The victor saves the soldier entity and the city entity. In this case only the Soldier entity of the loser will be deleted. Finally, the function ends, and the result of the war is returned to the superior program as a boolean value.

Implementation: Programming Implementation of Game Architecture

After the implementation of the basic elements of the game is completed, the game developer gains the ability to further build the game architecture by combining these basic elements.

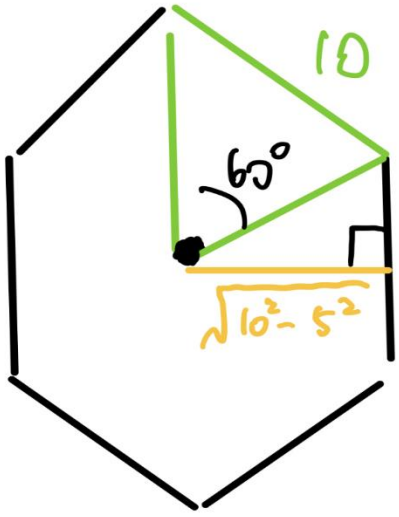
Infrastructure layer (base layer): script "CellDetails.cs", script "CellCoords.cs" and script "CellGrids.cs"

(1) Behavior Definition Program: The implementation of this program has been discussed in (4)(5)(6) of the "Behavior Implementation" section.
(2) Numerical System Program: The implementation of this program has been discussed in (1)(2)(3) of the "Player, Entity and Property Implementation" section.
(3) Rendering program: hexagonal coordinate system, triangle-based rendering strategy
(3a) Through the discussion in the design part and the implementation part, a common requirement was discovered - for map rendering and cell generation programs, it needs to sort and track all generated cells in order to generate cells are assigned different

attribute values. On the other hand, for the behavior definition program, it needs a way to determine the location information of the requester initiating the behavior (such as the starting point and the end point in the movement behavior),

And a method for converting three-dimensional space position information into cell position information that can be read by a game system. It can be seen that both programs essentially need to establish a connection between the three-dimensional space coordinates and the cell sequence. In the early days of project development, an informal design was to convert a specific three-dimensional space coordinate to a cell's serial number in real time through some formula. This design was quickly rejected - the cells in this project were all hexagonal, which made their arrangement staggered, and the original, mutually perpendicular 3D coordinate system axes could not accurately align and track each cell . This error becomes more pronounced as the map area expands. Therefore, a new, hexagonal cell-based map axis needs to be developed.

First, the geometric properties of the cell are determined to specific values: in the script "CellDetails", the side length of the hexagonal cell is set to an integer of 10, and based on this value, the distance from the center of the cell to the vertex of the cell can also be calculated as 10, And the vertical distance from the cell center to the edge is (√100-25) (approximately equal to 8.66).



To preserve subsequent scalability, the vertical distance from the cell center to the edge is converted to 0.866 times the edge length. After the two basic quantities are determined, the relative coordinates of the six vertices of the cell can be calculated (without considering the y-axis perpendicular to the game map):

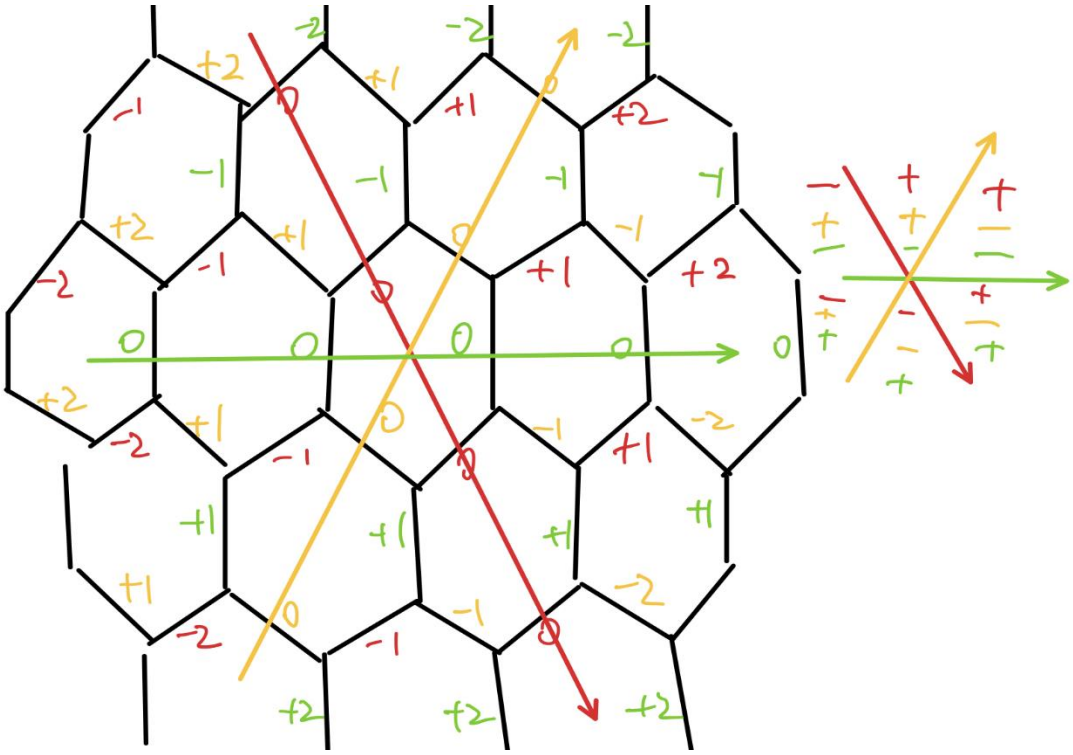
```
public class CellDetails : MonoBehaviour
{
    public const float outR = 10f;
    public const float innR = outR * 0.866f;
    public static Vector3[] vertexLocation =
    {
        new Vector3(0f, 0f, outR),
        new Vector3(innR, 0f, 0.5f*outR),
        new Vector3(innR, 0f, -0.5f*outR),
        new Vector3(0f, 0f, -outR),
        new Vector3(-innR, 0f, -0.5f*outR),
        new Vector3(-innR, 0f, 0.5f*outR),
        new Vector3(0f, 0f, outR)
    };
}
```

Second, in order for the new axis to align with the cells, the new axis group excluding the y axis should still contain three axes, and the angle between the axes should be exactly sixty degrees (each axis is perpendicular to the cell edge). In such a coordinate system, it is easy to obtain that the sum of the coordinate values of each axis of any point is 0. Therefore, in the class script "CellCoords", only the axis0 coordinate value and the axis1 value are stored, and the axis2 coordinate value can be directly derived from the symmetry.

Finally, when the game starts, the cells are generated according to the rule "left to right, bottom to top", and the serial number of each cell is also assigned according to this rule.

After the geometric properties of the cell and the new coordinate system have been developed, the method of converting the coordinates of the Cartesian coordinate system to the coordinates of the new coordinate system can also be implemented: for a point on a map, first find the coordinate value of axis0 in the horizontal direction (with the x-axis coordinate of the original Cartesian coordinate system and the key value of 8.66), which will directly derive the coordinate value of axis1 in the opposite direction to it. After that, by using the z value of the original Cartesian coordinate system and the key value of 8.66, an offset caused by the different angle of the

coordinate axis can be obtained, and the axis0 coordinate value and the axis1 coordinate value need to be calculated according to this offset. So far, all the Cartesian coordinates that are exactly at the center of the cells can be converted to the new coordinates.



In order to further complete the transformation of other possible points, additional if judgment statements are deployed in the transformation process, they aim to assign the coordinates of each possible point to the nearest cell center to complete the coordinate transformation (this means that points that are not in the center of the cell will also be considered to be in the center of the cell after the transformation is complete).

Such assignments also prepare the new axis coordinates for eventual conversion to cell serial numbers.

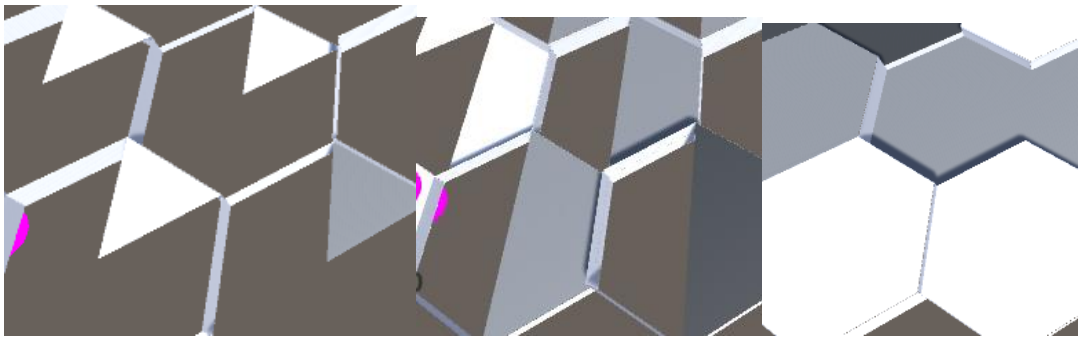
```
for (int i = 0; i<cells.Length; i++)
{
    Vector3 core = cells[i].transform.localPosition;
    for(int o = 0; o<6; o++)
    {
        int pointCount = pointsInSix.Count;
        pointsInSix.Add(core);
        pointsInSix.Add(core + CellDetails.vertexLocation[o]);
        pointsInSix.Add(core + CellDetails.vertexLocation[o+1]);
        trisInSix.Add(pointCount);
        trisInSix.Add(pointCount + 1);
        trisInSix.Add(pointCount + 2);
        colors.Add(cells[i].color);
        colors.Add(cells[i].color);
        colors.Add(cells[i].color);
    }


}
allCellMesh.vertices = pointsInSix.ToArray();
allCellMesh.triangles = trisInSix.ToArray();
allCellMesh.colors = colors.ToArray();
allCellMesh.RecalculateNormals();
meshCollider.sharedMesh = allCellMesh;
```

Finally, the new axis coordinates can be converted to cell serial numbers through the following process:

The cell sequence number is equal to axis0 plus one-half of axis1 plus the product of axis1 and the map width value.

The principle of this process is that axis0 is the horizontal coordinate position of the current cell, and it can be inferred that the serial number of the current cell is at least greater than this value. The remaining value should be equal to the number of rows below the current cell multiplied by the width of the map - since axis1 is the axis of a non-rectangular coordinate system, half of the axis1 value should be added.

(3b) In the unity engine, all polygons must be built based on triangles. Therefore, thanks to the hexagonal vertex coordinates set in (3a), the rendering process of the cell can be divided into 6 triangle rendering sub-processes:



Mechanism layer (middle layer): script "SymbolicOrder.cs"

In order to accomplish the three goals proposed in the design part, the implementation process is divided into three parts:

(1) "start, end and loop" implementation of turns

(1a) The start of game turns depends on the "RoundEnter()" function in the script "SymbolicOrder", in which the way the turns are started will depend on the identity of the current player (human or computer player). If the computer player is currently requesting to enter the turn, the "RoundEnter()" function will invoke the "AIhelper(playerIndex)" function and hand over control of the game to the AI model of the High layer. If a human player is currently requesting to enter the turn, then the "RoundEnter()" function sets the elements of the UI that are available for human interaction with input devices (such as buttons and cursor tracking) to be open - that is, transfers control to " Update()" function. Each element corresponds to a special integer variable that can only take a value between 0 or 1, so these variables can also be approximated as Boolean variables:

The "movementflag" variable, which controls the permissions related to the movement behavior, can only be assigned a value of 1 when "armyselectflag" or "seedselectflag" is equal to 1 (meaning the player is allowed to perform the movement behavior).

The "seedselectflag" variable is responsible for determining whether any settler entity on the map is selected by the cursor. It can only be assigned a value of 1 if the other flag variables are equal to 0 (to ensure that only one entity can be selected by the human player at a time).

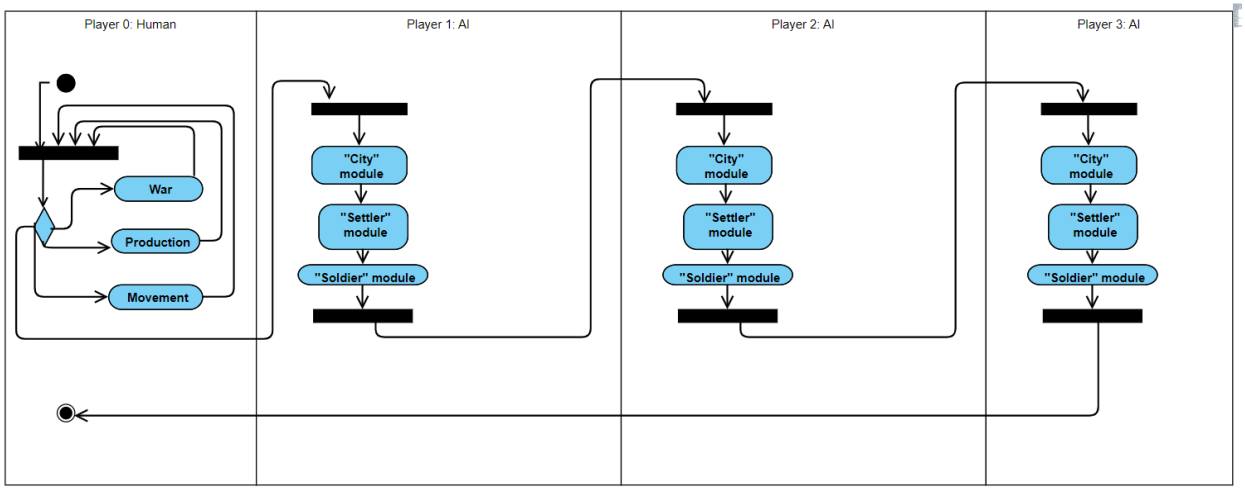
The "cityselectflag" variable is responsible for determining whether any city entity on the map is selected by the cursor. It can only be assigned a value of 1 if the other flag variables are equal to 0.

The "armyselectflag" variable is responsible for determining whether any soldier entity

on the map is selected by the cursor. It can only be assigned a value of 1 if the other flag variables are equal to 0.

The above variables can only be assigned in the Update() function or the behavior function. Therefore, such a structure ensures the sustainability of human player operations.

The "duringflag" and "turninflag" variables are responsible for protecting the current player's turn from being terminated unexpectedly. The "turninflag" variable is mainly for the High Layer, and it will be assigned to true at the end of the "RoundendClick()" function and false at the end of the "RoundEnter()" function. The "duringflag" variable is assigned a value of 1 at the beginning of the "RoundEnter()" function and a value of 0 at the end of the "RoundendClick()" function.



(1b) The end of the game turns depends on the "RoundendClick()" function in the script "SymbolicOrder". Inside this function, all flag variables (except "duringflag" and "turninflag") and cache variables will be reset. And after the reset is over, a settlement function will be called to calculate the current player's payoff in the current round (ie, the current set of 4 turns). After that, the judgment function "Endthankschecking()" used to determine whether the game can end at the current time will also be called. It will judge the occupied situation and time situation of the current map, and decide whether to continue the game or not.

(1c) In particular, since the actions of human players are unpredictable, the "Update()" function will always monitor the actions of human players in turn. A large number of decision statements are deployed into functions to ensure that the game system can respond to human behavior requests (or other requests) in a timely manner. In addition, human input devices include the mouse and keyboard. The mouse is used to control the cursor to select entities or click buttons, while the keyboard is responsible for

moving the camera - an important part of the game screen output.

(2) Repackaging and interface exposure of behavior definition functions in the base layer

(2a) As discussed in (1) of this section, the functional functions of the middle layer are divided into two types: human-oriented and computer-oriented players - although they still call the same basic behavior definition function in the end.

The "ChessSeedDeploy()" function is a repackage of the production function "SeedDeploy", adding the registration of the "powerbelong" array to the original function, and finally facing human players. The function of the same name prefixed with the word "AI" is intended for computer players.

"ChessArmyDeploy()" is a repackage of the production function "ArmyDeploy()", adding the registration of the "powerbelong" array to the original function, and finally facing human players. The function of the same name prefixed with the word "AI" is intended for computer players.

"ChessCityDeploy()" is a repackage of the production function "CityDeploy()", adding the registration of the "powerbelong" array to the original function, and finally facing human players.

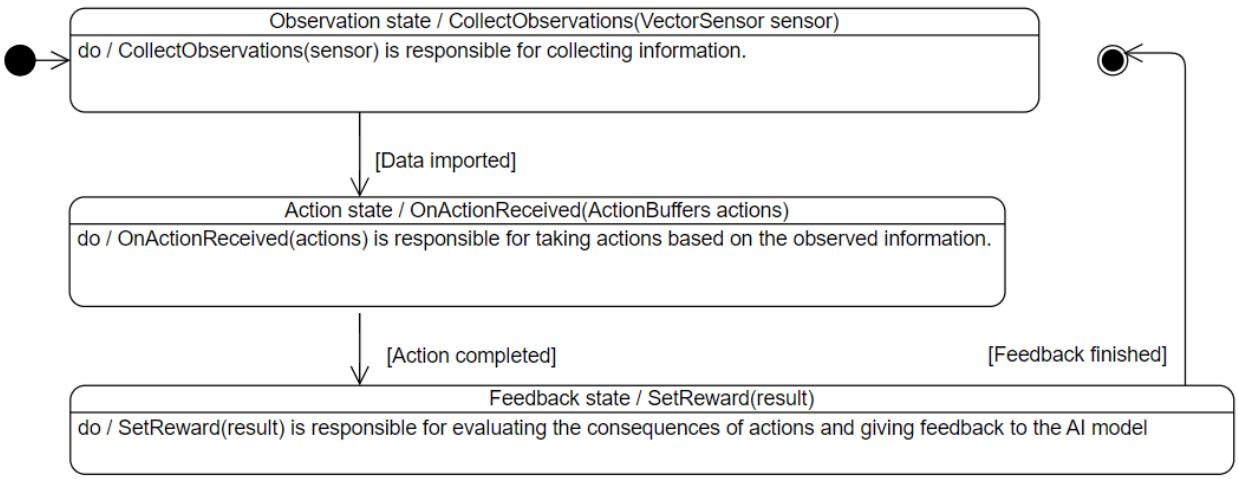
It should be noted that the re-wrapping of "CityDeploy()" for computer players is incorporated into the "AIChessSeedMove(cellIndex)" function to match the AI model's decision.

The "UnitMove(objname, hit.point, playerIndex)" function of the base layer is repackaged into the "Update()" function, which adds a check on whether the human player can initiate the movement behavior based on the original function.

(2b) The interfaces of the above repackaged functions are exposed to UI elements such as buttons responsible for direct interaction with human players.

(3) As mentioned in (2b), the response of the game system to the human player is mainly accomplished through button and cursor tracking. On the other hand, computer players will be able to directly call the various functions implemented in (2a) to complete their own decisions.

Artificial intelligence layer (High-layer): script "AmadeusMove.cs", script "AmadeusProd.cs" and script "AmadeusAttack.cs"



The artificial intelligence layer (High-layer) in this project is implemented by a reinforcement learning algorithm (PPO). The implementation tools of this layer are ML-Agents, which allow to establish a connection between the Python virtual environment and the Unity engine, and call a series of toolkits (such as PyTorch) within the Python virtual environment. In addition, ML-Agents also allow to start training the AI model directly while the Unity engine is running, and monitor the training progress through a visual window.

Due to the nature of reinforcement learning algorithms, the AI model of this project does not need to be trained multiple times using additional training scenarios (training sets). This means that after using ML-Agents to complete the model initialization (initial training), the AI model can be extracted and used in the official game - in reinforcement learning algorithms, the parameters of the model will be self-correcting through the "observation-action-feedback" loop (unsupervised learning).

In order to construct the loop, some meta-parameters and meta-rules need to be set for the model, which will represent the final goal of the model, the base observation mode and the base action mode, respectively.

The implementation of this layer can be divided into three parts, corresponding to three game entities. By using the multi-agent joint function of the ML-Agents tool, the AI models contained in the three parts will share the same set of reward values (feedback) to ensure that the models can learn a certain degree of cooperative strategy during the learning process. Furthermore, after decomposing the overall decision-making task of the AI layer into three sub-decision tasks, the complexity of each AI model is reduced. This structure also reduces the difficulty of implementation - both the metaparameters and metarules of the models can be defined in parallel without interfering with each other. Ultimately, three relatively simple models together form a

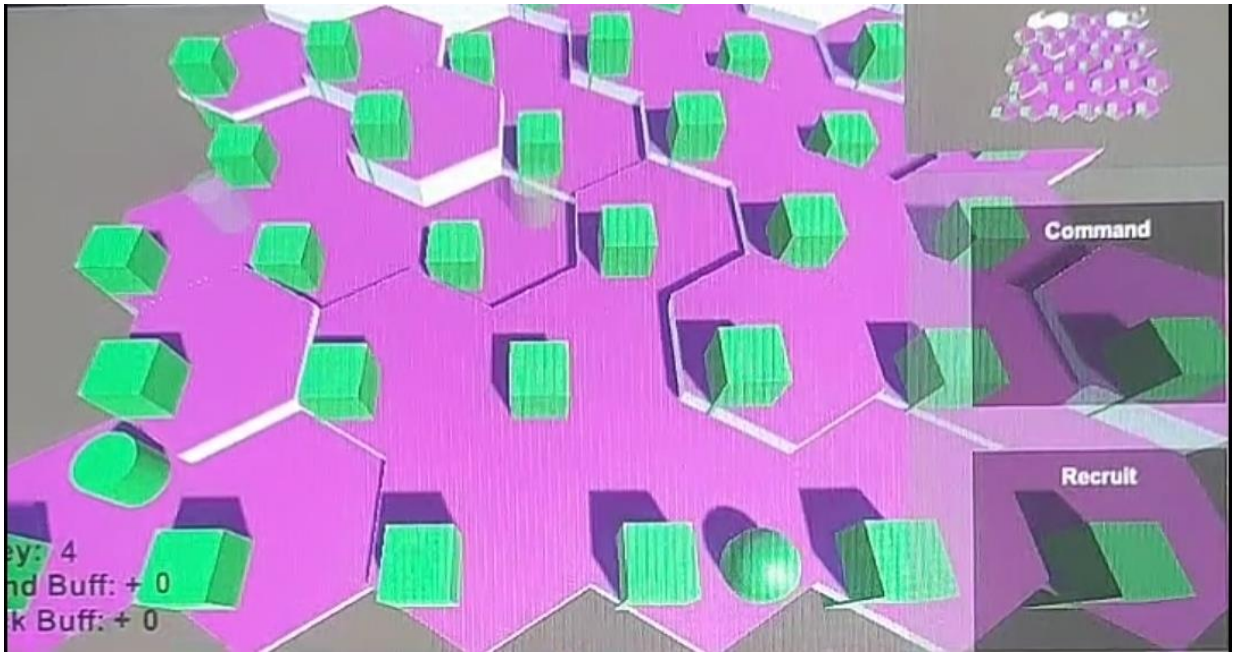
multi-agent network that achieves intelligence through self-organization and collaboration.

It should be noted that the following modules only serve one player, and in the end, the multi-agent system was replicated three times in the project to serve three computer players in addition to human players. Also, the entities manipulated by the following modules are limited to those owned by a player (not all entities on the map).

(1) "settler" module: script "AmadeusMove"

The goal of this mod is to drive as many settler entities as possible to travel outward and capture as many neutral cells as possible. Building on this goal, the module also attempts to drive settler entities to occupy the richest cells they can reach. The following are the meta-rules and meta-parameters developed to achieve the goal of the module: (1a) The outer function "AIhelper(playerIndexs)" will traverse all the "settlers" owned by the current player and call the "CollectObservations(sensor)" function for them in turn to start observing the environment. The specific observation elements include: the entity waiting for a decision, and the information about the cells within a distance of 2 units ("money" attribute, "bonus" attribute and owner). (1b) After that, the AI model outputs an integer as the decision result. The interval of the integer is [0, 17], representing 18 candidate destination cells respectively. The "OnActionReceived(actions)" function that receives the decision result will call the "AIChessSeedMove()" function of the middle layer to move the entity waiting for the decision, and call the "ChessCityDeploy()" function after the movement action is completed to initiate "Production" behavior (build a new city with that settler). (1c) As a result, the feedback obtained by the AI model is directly related to the richness of the occupied cells, which makes the poorer cells to be skipped in subsequent decisions. In addition, in order to prevent the AI model from making invalid decisions, the end of the module will additionally detect whether the movement behavior and production behavior have been successfully initiated, and provide additional feedback to the AI model.

Ultimately, after multiple decisions, the AI model will tend to take actions that are effective and rewarded with more feedback.



(2) "city" module: script "AmadeusProd"

The goal of this mod is for city entities to produce as many settler and soldier entities as possible. Building on this goal, the module also tries to prioritize the production of settler entities when resources are insufficient to produce both settler entities and soldiers. (2a) The outer function "AIhelper(playerIndexs)" will traverse all city entities owned by the current player, and call the "CollectObservations(sensor)" function for them in turn to start observing the environment. The specific observation elements include: the entity waiting for a decision, and the resource currently owned by the player (money). (2b) After that, the "OnActionReceived(actions)" function will directly try to build the settler entity and, if the conditions allow, the soldier entity. (2c) As a result, there is a correlation between the feedback the AI model gets and whether the entity is successfully produced, which makes the AI tend not to take ineffective actions such as "repeatedly trying to produce an entity when there are not enough money points".

(3) "soldier" module: script "AmadeusAttack"

The goal of this module is to search for and destroy enemy city entities. Based on this goal, the module also attempts to direct friendly soldier entities as close as possible to the cells where the enemy entities are located. Eventually, the soldier entity that reaches the enemy cell will wage war and accomplish the goal of destroying (weakening) the enemy entity. (3a) "AIhelper(playerIndexs)" will traverse all soldier entities owned by the current player and call the "CollectObservations(sensor)" function for them in turn to start observing the environment. The specific observation

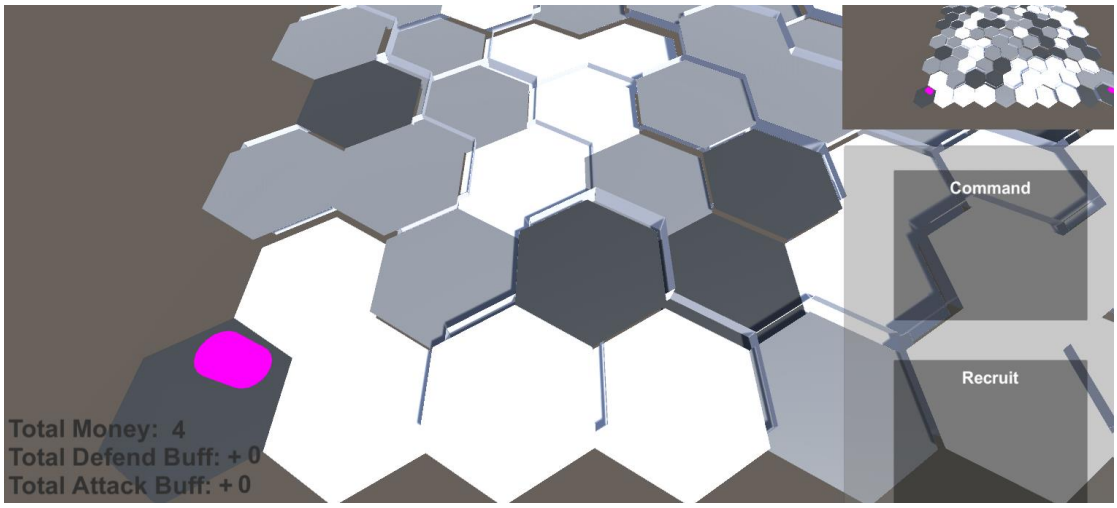
elements include: the entity waiting for a decision, the serial numbers of the cells within a distance of 2 units around the entity, and the information (distance, force) of any non-neutral cell on the map. (3b) After that, the AI model outputs an integer as the decision result. The interval of the integer is [0, 17], representing 18 candidate destination cells respectively. The "OnActionReceived(actions)" function that receives the decision result will call the "AIChessSeedMove()" function of the middle layer to move the entity waiting for the decision. Since the functions involved in the war will be automatically called when the two entities encounter each other, the AI model does not need additional operations at this time. (3c) The key to implementing this module lies in the feedback part - after completing the action, the AI model will receive a distance difference as feedback, which will determine whether the new position after the action is closer to the enemy cell than the original position. After many actions, the AI model will tend to drive the soldier entity through the shortest straight-line distance to reach the enemy cell.

After the three modules are developed, the multi-agent network composed of them begins to cooperate to demonstrate the macro strategy capability: after the entity is supplemented by the city module, the settler module will drive a large number of settler entities to expand the virtual empire. And the new city entity built by the settler entity will produce more soldier entities. These soldier entities will destroy the hostile city entity and hostile soldier entity to ensure that the new settler entity can successfully occupy more cells. Over time, AI models will increasingly be inclined to make such decisions to maximize feedback gains.

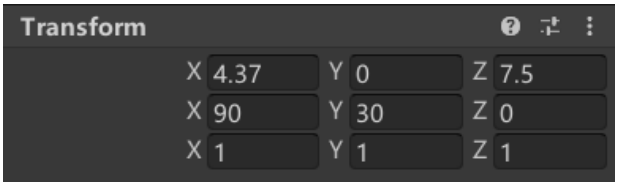
Development problems and solutions

(1) Map modeling dislocation

Problem description: In the development of the infrastructure layer (base layer) in the first stage, there was a situation where the map modeling deviates from the cell grid.



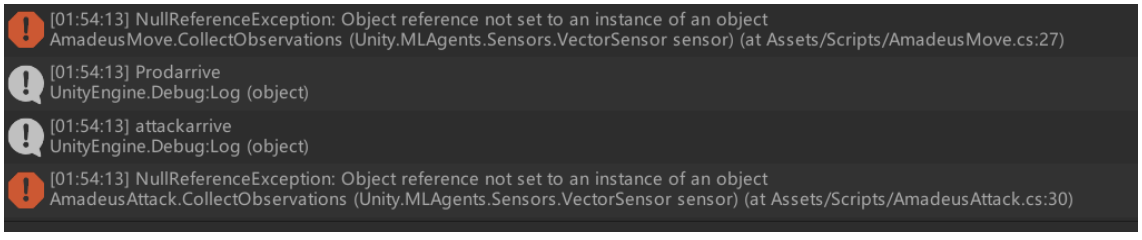
Problem analysis: After careful investigation, the generation of rendering grids and cells did not appear to be faulty or abnormal, so it can be inferred that the fault lies in the map modeling. After many tests, the map modeling component "EdgeSix" was found to have an abnormal local coordinate drift, which caused the map modeling to be shifted to the right by 4.37 units as a whole, causing a visual dislocation.



Problem Solution: The map modeling component "EdgeSix" was re-bound into the game space and its local coordinates were reset to 0.

(2) Null pointer error of round system

Problem description: During the development of the third-stage artificial intelligence layer (High-layer), a serious null pointer error occurred, which directly led to the stagnation of the game process.



Problem analysis: In the analysis, the AI model based on ML-Agents has undergone many revisions, but the occurrence of null pointer errors is still not avoided. After several tests, an unusual parameter was located in the "SymbolicOrder.cs" script. This

parameter (which is essentially a null parameter) is passed to the "AmadeusMove.cs" and "AmadeusAttack.cs" scripts of the AI layer (High-layer), which triggers a null pointer error. Therefore, it can be inferred that the failure is in the round system of the mechanism layer (middle layer).

Problem study: After a detailed analysis of the features of the ML-Agents toolkit, a hypothesis was proposed: Although the computer players are designed to enter rounds sequentially to start their respective turns, inside the turns, the computer players need to call the "Amadeus" script The group comes to request the AI model to make a decision - this decision-making process is parallel to the program process of the mechanism layer (middle layer). This parallel relationship raises the problem that the mechanics layer (middle layer) may open another new turn directly before the AI layer (High-layer) has completed its decision, thus giving new computer players access to the wrong public parameters. Since the common parameters passed between the middle layer and the high-layer are usually entity names, the AI models of new computer players will not be able to correctly identify and reference these entity names. Once this failure occurs once, subsequent computer players can only make a decision request with a null variable as a parameter, which eventually results in a null pointer error.

Problem solved: Add a frame rate based wait mechanism at the end of each turn. In a computer system with a refresh rate of 60 Hz, the mechanism layer (middle layer) will wait an additional 0.5 seconds for the artificial intelligence layer (High-layer) to ensure that the decision can be successfully completed.

```
if (turninflag == false && duringflag ==0)
{
    //ALERT!!!!!!!!!!NEED CHANGE HERE
    framer++;
    if (framer>=30)
    {
        framer = 0;
        RoundendClick();
    }

}else if(turninflag == true && duringflag ==0)
{
    framer++;
    if (framer >=30)
    {
        framer = 0;
        RoundEnter();
    }
}
```

6. Test and Evaluation

This project adopts white box testing as the testing method.

The tests of this project can be divided into unit tests for various core functional functions and integration tests for AI effectiveness.

Unit tests

Core function: CellCoord PositionRelocated(Vector3 position)
The CellCoord returned by this function is a hexagonal coordinate system unique to the project. By verifying the correctness of the returned value, it can be determined whether the function can correctly convert the three-dimensional rectangular coordinate system into the hexagonal coordinate system coordinates.

Test case	*Input*	*Expect output*	*Actual output*	*Result*
(1a)	(21.3, 4.0, 56.0)	(-1, 4)	(-1, 4)	Success
(1b)	(0.0, 1.0, 30.0)	(-1, 2)	(-1, 2)	Success
(1c)	(86.6, 1.0, 0.0)	(5, 0)	(5, 0)	Success

Core function: int[] UnitMove(string objname, Vector3 indexPos, int playerIndex)
The array of integers returned by this function indicates the result of the Movement's behavior - the first bit of the array represents whether the selected entity was successfully moved. The second digit of the array represents the difference between the serial number of the destination cell and the serial number of the originating cell.

Test case	*Input*	*Expect output*	*Actual output*	*Result*
(2a)	("ChessSeedTest (Clone) 0", (95.3, 4.0, 45.0), 1)	(1, 7, 0, 0)	(1, 7, 0, 0)	Success
(2b)	("ChessSeedTest (Clone) 1", (77.9, 0.0, 45.0), 2)	(1, 2, 0, 0)	(1, 2, 0, 0)	Success
(2c)	("ChessSeedTest (Clone) 2", (95.3, 1.0, 15.0), 3)	(1, -9, 0, 0)	(1, -9, 0, 0)	Success

Core function (function group): string Deploy(string objname, int playerIndex)

This function group includes three functions of similar structure: SeedDeploy(string objname, int playerIndex), ArmyDeploy(string objname, int playerIndex) and CityDeploy(string objname, int playerIndex). They will return a string representing the nascent entity.

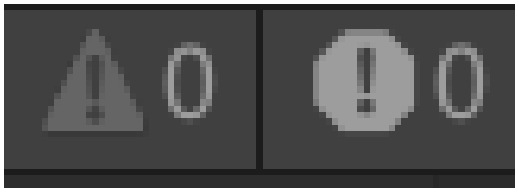
Test case	*Input*	*Expect output*	*Actual output*	*Result*
(3a)	("ChessSeedTest(Clone)0", 1)	("ChessCityTest(Clone)0")	("ChessCityTest(Clone)0")	Success
(3b)	("ChessCityTest(Clone)1", 1)	("ChessArmyTest(Clone)0")	("ChessArmyTest(Clone)0")	Success
(3c)	("ChessCityTest(Clone)2", 1)	("ChessSeedTest(Clone)4")	("ChessSeedTest(Clone)4")	Success

Integration Testing and Evaluation: AI Effectiveness

A feature of this project is the deep involvement of AI in the game. When a human user starts the project in "simulation" mode, a game involving only computer players will begin - the human user will watch the game as an observer. Therefore, direct observation of the pattern of the game also becomes a means of integration testing. More importantly, in this mode, the AI models of the three computer players participating in the game have differences in intelligence (difficulty) - the "settler" modules of player 2 and player 3 cannot obtain correct feedback based on their actions, which so that they cannot effectively identify rich cells. This design is also reused in the difficulty setting of the project: if a human player starts the project in "easy" mode, each of the three computer players in the game will have a weakened "settler" module.
Therefore, conducting integration tests through the "simulation" mode of the project will not only verify the core functionality in the unit test, but also verify the effectiveness of the AI at the same time.

In the integration test, the project was locked in the following environments: "Simulation" mode, "50-round quick game", "6*6 map". This means that the game will end at turn 50, and the game system will decide the computer player with the largest virtual empire wins.

A total of ten rounds of integration tests were performed without any errors. In the test, player 1 with the highest intelligence (difficulty) wins 6 games, player 2 wins 1 game, and player 3 wins 3 games. It can be found that AI with higher intelligence can expand the virtual empire more quickly in the same time, and finally gain a greater advantage in the game.



Evaluation: Questionnaire

Around the operability and playability of the project, a set of questionnaires was designed.

The questionnaire consisted of ten questions:

1. Play time 2. Play results 3. Numerical system 4. Strategy and economic advantages 5. Strategy and geography 6. War and war outcomes 7. War determinants 8. Function of city entity 9. Program bug report 10. Gameplay Vulnerability report

Detailed topics are in the appendix.

Through such a set of questionnaires, projects can be further evaluated. In addition, all private information involved in this questionnaire has been properly kept and protected by relevant network protocols and laws [13].
During the investigation and evaluation, no obvious bugs or glitches were found in the gameplay and procedures. However, a problem that has been mentioned many times is the lack of an interface for efficient viewing of cell properties in the game. Therefore, an additional information bar is deployed to the upper left corner of the screen, which is used to display the details of the cell currently selected by the human user. Further, in the improved project, richer cells will have darker colors. This feature will help human users to distinguish cells in different situations.

Evaluation: Original Aim and Objectives

Aim: The aim of this project is to develop a strategy computer game with artificial intelligence players.
Objectives: Maintainability, 3D-based development, Gameplay, AI, Others

First, Objectives are all achieved from:

- Maintainability: C#, Object-Oriented Programming
- 3D-based development: Unity engine, hexagonal map with cells,
other interfaces
- Gameplay: Turn-based game flow, game rules
constructed by three behaviors, three operable entities
corresponding to three behaviors
- AI: Reinforcement learning (PPO) algorithm
- Others: Soundtrack

Therefore the aim of the project is achieved.

7. Conclusion:

This project is a simplified version of the Civilization game. Through the use of reinforcement learning technology, 3D engine technology, object-oriented programming ideas and technologies, this project has achieved all the design

requirements in the development plan and has become a 3D strategy turn-based game with AI elements. Through the UI interface, human users who use the project can modify the difficulty (AI intelligence) of the computer players participating in the game, the game time, and the size of the game map, so as to experience the respective characteristics of the game as much as possible. Further, human users can also choose not to directly participate in the game, but choose to observe the game between computer players driven by AI models.

8. BCS Project Criteria & Self-Reflection

An ability to apply practical and analytical skills gained during the degree programme.

In the development of the program, a number of competencies learned in the degree program were applied, such as:

(1) In the design part, the knowledge of COMP201 about software development is applied, which mainly includes how to design requirements and construct project prototypes. In addition, the drafting skills learned from COMP201 and COMP208 are also applied to the project.

(2) In the implementation part, COMP222's knowledge about the Unity engine is applied -

On the one hand, the relevant knowledge of the engine, on the other hand, the C# programming skills learned in the class are also used and combined with the Unity engine.

(3) In the artificial intelligence implementation part, the knowledge about artificial intelligence and machine learning of COMP219 is applied - this includes many key knowledge and skills such as model training and parameter setting.

Innovation and/or creativity.

In the artificial intelligence implementation part, a set of AI models based on reinforcement learning algorithms are built. This is rare in strategy games—current mainstream strategy games still rely on classic models such as state machines and decision trees to drive computer players. Further, this AI model is decomposed into three subsystems, and finally constitutes a multi-agent system, which is a more innovative design.

Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution.

In the early stage of development, developers need to choose from a variety of mature game development solutions and prepare a development route based on the knowledge they already have. After investigation and analysis, the "Unity-C#-Reinforcement Learning" route is suitable for developers - first of all, Unity is highly scalable, has excellent 3D screen rendering capabilities, and has an intuitive visual interface. On the other hand, C# is a computer language that developers have learned and mastered. Using C# as a development language can save a lot of learning costs. Ultimately, the AI model built by the reinforcement learning algorithm will guarantee the innovation of the project, and the simple structure of the algorithm also saves the time for developers to debug parameters.

That your project meets a real need in a wider context.

(1) After the development, this project can become an official game published on the Internet

(2) The AI model of this project can be extracted and used in other fields (such as resource allocation procedures).

An ability to self-manage a significant piece of work.

In the detailed proposal of the project, the developers proposed a main aim and a series of objectives. Together these objectives make up the project. It should be pointed out that although there are many subsystems in this project, and there are some difficult parts (artificial intelligence part), by decomposing the objectives into smaller tasks, the developers successfully completed the objectives one by one. In addition, by selectively completing some basic tasks first during development, the development speed of the project is increased. As a result, the developers took the project from a completely non-existent idea to a release-ready game. Therefore, developers have the ability to self-manage.

Critical self-evaluation of the process.

Disadvantage 1: limited artistic expression

Early in the development of the project, the developers had planned to spend a week

or two learning the techniques of designing complex 3D models, but this plan was abandoned as other parts of the development took too long. As a result, all three entities in the project can only exist in the game in the form of simple geometry.

Disadvantage 2: The game occupies too much volume during the development process

Since the project needs to use ML-Agents and a Python virtual environment, and ML-Agents needs to have a built-in PyTorch tool library in the game, the size of the game has been expanding - after the development was completed, the total size of the game reached 7G.

Disadvantage 3: AI intelligence needs to be improved

It can be seen from the description of the implementation part that the AI system of this project needs to call the three components of "city" - "settler" - "soldier" in sequence, which actually affects the strategic ability of computer players. Compared with computer players, human players can repeatedly initiate a variety of behaviors and achieve strategic goals through appropriate combinations.

Advantage 1: High cohesion and low coupling

By encapsulating the behavior functions and related functions of the round system and retaining limited interfaces, the reusability of the program is greatly improved - which significantly reduces the failure rate of the project and improves the maintainability of the project. Further, in future maintenance and upgrades, this project can also rely on these interfaces and structures for rapid deployment.

Advantage 2: Advanced AI Architecture

Although the current AI model still has room for improvement, the AI architecture based on reinforcement learning in this project is still competitive. Through an efficient learning loop, the AI model of this project can adapt to a variety of randomly generated maps, regardless of their size and properties.

Benefit 3: A first-principles development process

This project is a simplification of the Civilization game. But how to reduce game elements reasonably so that the final product can still retain the basic elements (4X elements) of strategy games? Developers use first principles—that is, to return to the most basic conditions of things, split them into elements for deconstruction analysis, and find the best way to achieve the goal. Through first principles, developers creatively divide the behaviors in the game into three categories, and develop three corresponding entities.

9. References

[1] P. J. Diefenbach, "Practical Game Design and Development Pedagogy," *IEEE Computer Graphics and Applications,* vol. 31, no. 3, pp. 84-88, 2011, doi: 10.1109/MCG.2011.45.

[2] M. Arcidiacono, J. J. Dominic, and R. A. Heru, "HUBERT, Our FreeCiv AI."

[3] Z. S. Griffith, "Salvation: Into the Cosmos: Board Game Project Blending 4X and Eurogame Styles," 2017.

[4] J. Haas, "A history of the unity game engine," *Diss. WORCESTER POLYTECHNIC INSTITUTE,* 2014.

[5] M. C. Machado, B. S. L. Rocha, and L. Chaimowicz, "Agents Behavior and Preferences Characterization in Civilization IV," in *2011 Brazilian Symposium on Games and Digital Entertainment*, 7-9 Nov. 2011 2011, pp. 43-52, doi: 10.1109/SBGAMES.2011.17.

[6] S. Wender and I. Watson, "Using reinforcement learning for city site selection in the turn-based strategy game Civilization IV," in *2008 IEEE Symposium On Computational Intelligence and Games*, 15-18 Dec. 2008 2008, pp. 372-377, doi: 10.1109/CIG.2008.5035664.

[7] I. Athaillah, S. M. S. Nugroho, and M. Hariadi, "NSGA-II for City Building Placement Optimization in the Turn-based Game Civilization VI," in *2019 12th International Conference on Information & Communication Technology and System (ICTS)*, 18-18 July 2019 2019, pp. 60-64, doi: 10.1109/ICTS.2019.8850976.

[8] C. Amato and G. Shani, "High-level reinforcement learning in strategy games," in *AAMAS*, 2010, vol. 10, pp. 75-82.

[9] A. Juliani *et al.*, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627,* 2018.

[10] M. Bosnjak and T. Orehovacki, "Measuring quality of an indie game developed using unity framework," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 21-25 May 2018 2018, pp. 1574-1579, doi: 10.23919/MIPRO.2018.8400283.

[11] T. Martyniuk, "Implementing Hexmap Generation Framework using Cube Coordinate System in Unity3D," 2021.

[12] Hexagonal Grids, *redblobgames.com*, 2013. [online]. Available: https://www.redblobgames.com/grids/hexagons/, [March 11, 2013]

[13] Wenjuanxing, *Wenjuanxing*, 2022. [online]. Available: *https://www.wjx.cn/* , [May 3, 2022]

10. Appendices

Complete documentation of integration tests

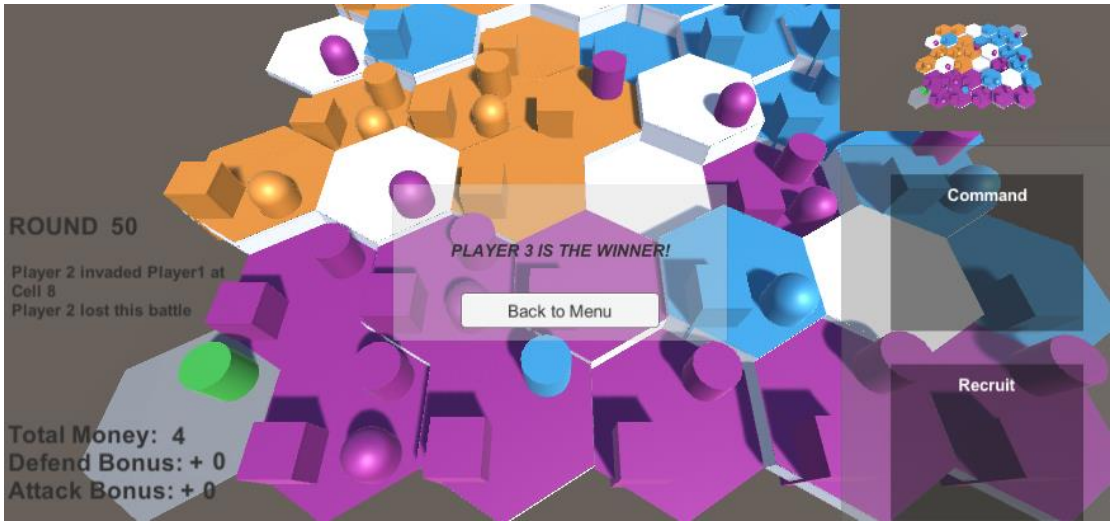

ROUND 50
Player 1 invaded Player3 at
Cell 28
Player 1 won this battle
PLAYER 1 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 3 invaded Player2 at
Cell 1
Player 3 lost this battle
PLAYER 2 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0

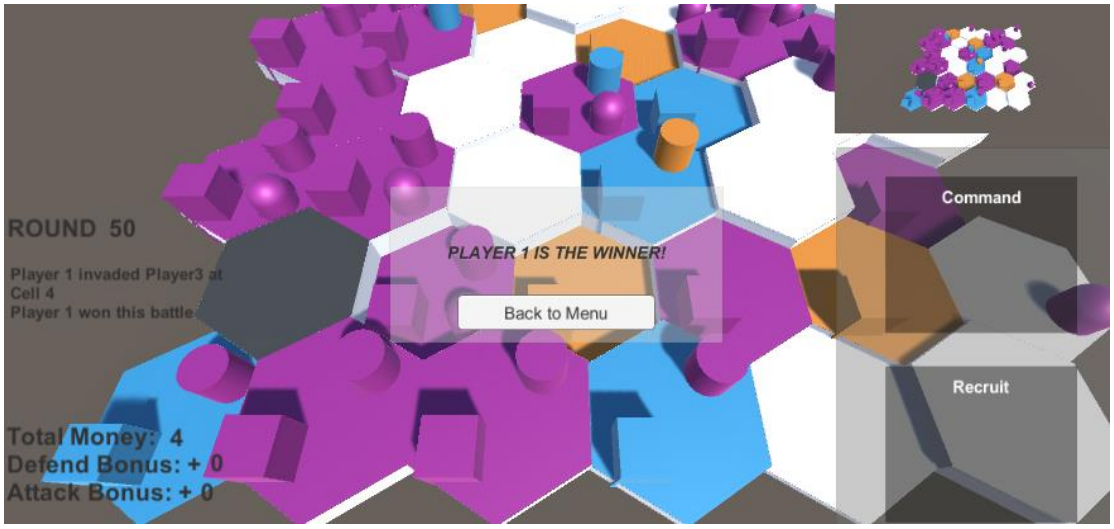

ROUND 50
Player 2 invaded Player1 at
Cell 27
Player 2 won this battle
PLAYER 2 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0

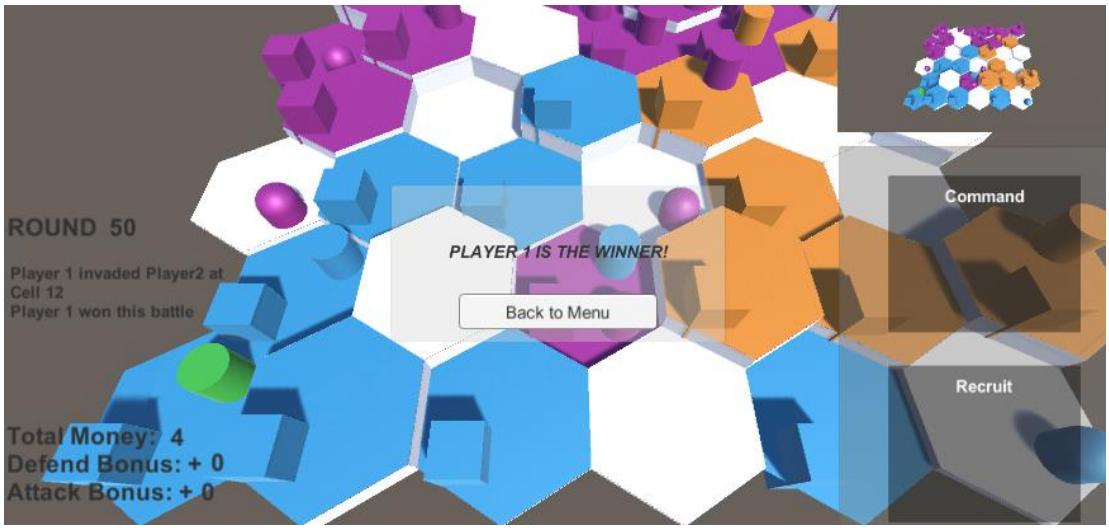

ROUND 50
Player 2 invaded Player1 at
Cell 25
Player 2 lost this battle
PLAYER 1 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 3 invaded Player2 at
Cell 17
Player 3 won this battle
PLAYER 1 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 2 invaded Player1 at
Cell 8
Player 2 lost this battle
PLAYER 3 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 1 invaded Player3 at
Cell 29
Player 1 won this battle
PLAYER 2 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 1 invaded Player3 at
Cell 4
Player 1 won this battle
PLAYER 1 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0


ROUND 50
Player 1 invaded Player2 at
Cell 12
Player 1 won this battle
PLAYER 1 IS THE WINNER!
Back to Menu
Command
Recruit
Total Money: 4
Defend Bonus: + 0
Attack Bonus: + 0

Questionnaire:

1. How long do you play?
2. What is the result of your game? Who became the victor?
3. Did you use the numerical system in your game and take advantage of economic advantages to expand rapidly?
4. Have you discovered the differences between cells in the game and expanded the economic advantage by capturing as many rich cells as possible?
5. Can you read the map information in the game and help you discover the differences between cells?
6. Have you ever started a war in the game? If so, have you lost a war?
7. Do you realize in the game that the decisive factor of war is the force value? Have you found a way to increase the force value of soldier entities?
8. Did you use the city entity to create new entities in the game?
9. What gameplay bugs have you encountered in the game?
10. What bugs have you encountered in your game?