МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
« » червня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Багатокористувацька комп'ютерна гра жанру покрокова стратегія з тематикою відбудови міст»
здобувача групи ІН – 01 Оболонського Дениса Володимировича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Денис ОБОЛОНСЬКИЙ
(підпис)

Керівник, старший викладач кафедри компьютерних наук, к.т.н., доцент _____ Борис КУЗІКОВ
(підпис)

Суми – 2024

Сумський державний університет
Центр заочної, дистанційної та вечірньої форм навчання
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-01 Оболонського Дениса Володимировича

1. Тема роботи: «Багатокористувацька комп'ютерна гра жанру покрокова стратегія з тематикою відбудови міст»
затверджую наказом по СумДУ від *«22» квітня 2024 р. № 0414-VI*
2. Термін здачі здобувачем кваліфікаційної роботи до 29 травня 2024 року
3. Вхідні дані до кваліфікаційної роботи _
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Інформаційний огляд жанру стратегії. 2) Огляд існуючих рішень мережевої ірхутертури багатокористуавцьких ігор, методів обміну та синхронізації даних. 3) Постановка задачі. 4)Розробка дизайну гри, механік та ігрового процесу. 5) Проектування мережевох архітектури додатку. 6) Програмна реалізація ігрової логіки на рушії Godot. 7) Створення інтерфейсу програми 8) Аналіз результатів
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)____
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	Аналіз предметної області	08.04.2024 – 10.04.2024	
2	Огляд існуючих рішень створення багатокористувацьких ігор	11.04.2024 – 15.04.2024	
3	Створення представлення, тематики та цілей гри	16.04.2024 – 17.04.2024	
4	Розробка ігрових механік	18.04.2024 – 20.04.2024	

5	Проектування мережевої структури проекту	21.04.2024 – 22.04.2024	
6	Вибір інструментів реалізації	23.04.2024 – 23.04.2024	
7	Програмна реалізація ігрової мапи та механік	24.04.2024 – 30.04.2024	
8	Створення інтерфейсу користувача	01.05.2024 – 05.05.2024	
9	Реалізація мережевої гри	06.05.2024 – 10.05.2024	
10	Тестування додатку	11.05.2024 – 11.05.2024	
11	Оформлення пояснювальної записки	12.05.2024 – 16.05.2024	

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 64 стр., 23 рис., 1 додаток, 23 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуально, оскільки піднімає важливе на сьогоднішній день питання по відновленню та обороні міста з використанням легкого та популярного методу - ігор.

Об’єкт дослідження —багатокористувацькі ігрові додатки.

Мета роботи — розробка багатокористувацької комп’ютерної гри з використанням ігрового рушію Godot та скриптової мови програмування GDScript.

Методи дослідження — інструменти розробки ігрових додатків рушію Godot та його скриптової мови GDScript, технологія Remote Procedure Call для зв’язку гравців по мережі

Результати — розроблено покрокову двомірну стратегію, в яку можуть грати два користувача одночасно. Реалізовано різні типи ігрового процесу для обох користувачів. Створено ігрову мапу та систему зберігання та оновлення її стану. Розроблено та реалізовано ігрові механіки. Додано головне меню, меню лобі та HUD-інтерфейси.

КОМП’ЮТЕРНА ГРА, БАГАТОКОРИСТУВАЦЬКА ГРА
ПОКРОКОВА СТРАТЕГІЯ, GODOT, 2D, GDSCRIPT

ЗМІСТ

ВСТУП

Актуальність. Нам важко уявити світ без ігор – карткові ігри, шахи, рухливі ігри відомі людству вже тисячі років і кожен з нас грав в них в дитинстві. Проте ігри, як і усе в нашому світі, не стоять на місці і йдуть в ногу з часом, тому все більшої популярності набирають саме комп'ютерні ігри. З кожним роком все більше людей відкривають їх для себе, що чудово демонструє статистика найбільш популярного інтернет магазину відеоігор для персональних комп'ютерів Steam, кількість одночасних користувачів котрого постійно росте [14]. І нещодавня пандемія підтвердила, що відеоігри є чудовим методом для проведення вільного часу [7]. І це не дивно, враховуючи можливості, які надають відеоігри своїм користувачам: головоломки дозволяють гарно розім'яти свій мозок, сюжетні проекти – насолодитися історіями, шутери – випробувати свою реакцію, а багатокористувацькі ігри – змагатися або кооперуватися з людьми з усього світу тощо.

Проте, окрім розважальної, ігри також можуть відігравати пропагандистську функцію. Як будь-які книга або кіно несуть у собі ідею, яку вони доносять до глядача, так і гра, неважливо містить вона гарно прописаний сюжет чи просто створені декорації, буде інструментом просування тих чи інших думок і поглядів - навіть якщо не безпосередньо, то побічно, через сам ігровий процес. Це робить ігри чудовим вибором для підняття і демонстрації необхідної автору теми, оскільки їхня інтерактивність сприяє глибокому зануренню гравця в те, що відбувається.

Об'єкт дослідження: багатокористувацькі ігрові додатки.

Предмет дослідження. Практичні та теоретичні аспекти розробки двомірних багатокористувацьких ігор жанру покрокова стратегія.

Новизна. Гра бере за основу важливу й актуальну в сучасних реаліях тему захисту й відновлення міст і піднімає її, використовуючи доволі прості

для розуміння механіки, реалізуючи водночас різний геймплей для однокористувацького та багатокористувацького режимів гри.

Структура. Дана робота складається зі вступу, аналітичного огляду, постановки задачі, моделювання та проектування гри, програмної реалізації геймплею гри, створення інтерфейсу додатку, реалізації мережевого з’єднання, висновків, списку використаних джерел та додатку.

1. АНАЛІТИЧНИЙ ОГЛЯД

1.1.Покрокові стратегії

Визначення жанру стратегії не є легкою справою, так як багато ігор так чи інакше стимулюють гравця створювати певну стратегію поведінки для проходження рівня чи перемоги в онлайн-матчі, а більшість ігор взагалі створюються як сукупність декількох жанрів. Тим не менш, основним принципом визначення стратегічних ігор є їх фокусування на довгостроковому плануванні дій гравця, а не на рефлекторних, миттєвих рішеннях «тут і зараз» [8].

Дане визначення все ще є доволі обширним і покриває собою багато несхожих один на одного ігор, саме тому стратегії прийнято поділяти на більш конкретніші під-жанри, основними з яких є [15]:

- Гранд стратегії – фокусуються на довготривалій грі з ігровими сесіями в декілька годин, впродовж яких гравець має керувати багатою кількістю ресурсів і обирати серед десятків варіантів розвитку подій. Одним з найпопулярніших типів гранд стратегій є 4X (explore, expand, exploit, exterminate), в яких гравцю пропонується побудувати та розвивати власну імперію, починаючи з малих наявних ресурсів, використовуючи дипломатію, воєнні дії, наукових досліджень тощо.
- Тактики – фокусуються на короткотривалих сесіях, де гравець має продумувати алгоритм дій для швидкого досягнення успіху.
- Настільні ігри – ігри, які по формі та геймплею відповідають фізичним настільним іграм
- Карткові ігри – ігри, в яких гравець в основному використовує гральні карти, які можуть бути персонажами, ресурсами, діями тощо

- MOBA (Multiplayer Online Battle Arena) – відносно швидкі ігри, які фокусуються на командній грі в межах однієї ігрової сесії задля перемоги над іншою командою на невеликій карті.
- Симулятори – ігри, які концептуально є моделями на основі речей з реального світу.
- Логістичні – ігри, в яких головний акцент робиться на керуванні та плануванні деякого виробництва.

Також стратегії поділяють на дві великі групи, в залежності від того, як реалізовано час – це покрокові ігри та ігри в реальному часі.

В першому випадку час в грі є дискретним і сам гральний процес поділений на ходи, протягом яких гравець може виконувати дії. Такий варіант найкраще підходить для одиночних ігор, так як гравець має достатньо часу для обдумування кроків та аналізу ситуації. В той же час він показує себе гірше в багатокористувацьких іграх, так як довгі очікування ходу часто змушують гравців нудьгувати і інтерес до гри втрачається. Тому покроковість гри модифікують, або додаючи обмеження часу на один крок або роблячи так, щоб гравці могли робити кроки одночасно.

Ігри в реальному часі навпаки мають неперервний потік часу, як в реальному світі, так що ігровий процес є також постійним. Такий підхід сам по собі створює напруження для користувача, обмежуючи його час на прийняття рішень, але і більше підходить для кращого занурення в гру. Також він ідеально підходить для багатокористувацьких ігор, так як користувачі можуть грати одночасно.

1.2.Мультиплеєрні ігри

Головне завдання будь-якої мультиплеєрної гри – це забезпечення плавного ігрового процесу та синхронізації даних для всіх гравців. Будь-яка помилка може призвести до того, що гра по мережі буде просто нестерпною.

Тому в ході того як розвивався інтернет та зокрема ігри, створювалися різноманітні варіанти розробки проектів розрахованих на велику кількість користувачів.

1.2.1. Архітектурні рішення

На початку створення будь-якого проекта завжди стоїть питання про те, як виглядатиме наша мережа фізично.

Так, одним із способів комунікації користувачів є peer-to-peer модель, де немає центрального сервера, до якого звертаються гравці, при цьому користувачі спілкуються безпосередньо між собою [17]. Даний варіант відмінно підходить для невеликих онлайнових проектів, оскільки для цього не потрібно використання серверу, а отже, і витрат на нього не буде. Проте спілкуватися безпосередньо між собою гравці просто так не можуть через перетворення локальної адреси гравця за допомогою NAT (Network Address Translation) [3]. Оскільки гравці найчастіше виходять у мережу не безпосередньо, а через роутери, то їх локальна адреса підміняється зовнішньою адресою цього пристрою. (Рисунок 1.1) [9]. Сам роутер зберігає інформацію про це з'єднання, тому коли він отримує відповідь від зовнішнього ресурсу він знає куди його необхідно перенаправити. Проблема полягає в тому, що для цього нам необхідно знати який порт та яку IP адресу використовувати для підключення до гравця, так як NAT буде блокувати невідомі запити. Для цього існують різні способи як обійти це обмеження, наприклад користувач може вручну відкрити порти або можна використовувати UPNP для автоматичного відкриття (хоча ця опція теж може бути відключена у гравця і йому доведеться все ще вручну її активувати). Так як не всі користувачі добре знаються на техніці, то змушуючи їх самостійно проводити подібні маніпуляції, розробники лише відштовхнуть їх від гри, крім того це вносить небезпеку для самого користувача, так як його пристрій тепер

може бути доступним ззовні. Але при цьому гравці можуть спокійно грати в одній мережі. Ще однією проблемою даного підходу є незахищеність сесій від чітерів тому, що вся інформація про гру зберігається на комп'ютерах користувачів, вони можуть її спокійно модифікувати, що нехай і не критично для ігор кооперативного проходження або для дружніх матчів, але є серйозною проблемою для ігор для змагань.

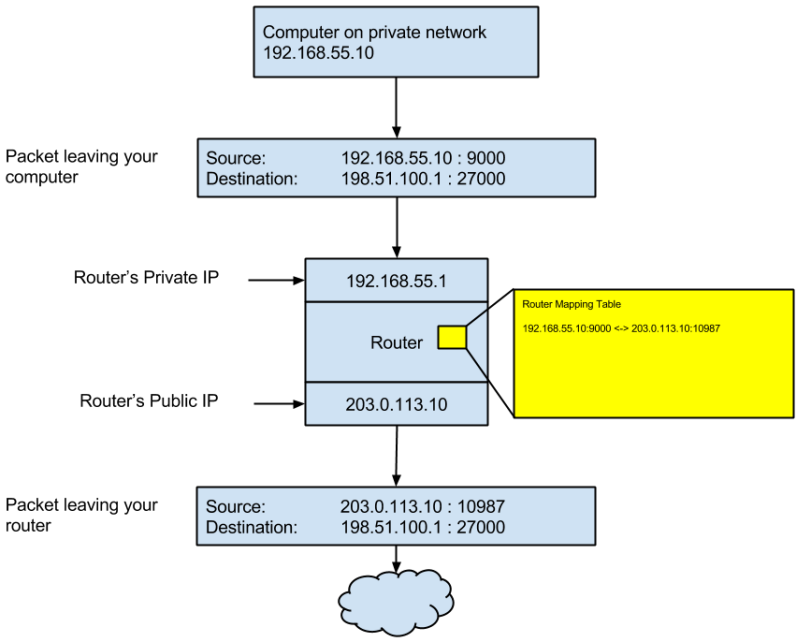

Рисунок 1.1 – NAT

Інший варіант це клієнт-серверна структура, яка передбачає використання центрального сервера, через який спілкуються гравці [17]. Маючи головний вузол зв'язку для всіх користувачів, розробники можуть реалізовувати авторизацію гравців та безпечне зберігання їх даних, контролювати як гравці обмінюються інформацією, запускати саму гру, проводити підбір гравців для окремих сесій тощо [5].

Однак сервер може бути використаний і для налаштування peer-to-peer з'єднання між гравцями за допомогою методу NAT punch-through. Його сенс полягає у тому, що центральний сервер знає публічну адресу кожного клієнта,

а тому може повідомляти його іншим. Таким чином гравці знають куди надсилати запити один одному. Але цей спосіб не є на 100% надійним, оскільки деякі роутери можуть не прийняти зовнішній запит або створене з’єднання буде видалено, якщо певний час не використовується [9].

Зрештою, сервер може бути використаний як точка передачі інформації між гравцями, тим самим “симулюючи” peer-to-peer з'єднання між ними, не завантажуючи сервер обробкою даних [3, 5]

1.2.2. Методи синхронізації даних

На сьогоднішній день найпопулярнішим варіантом є клієнт-серверна модель, яку також називають моделлю авторитарного сервера, тому що в ній сервер відповідає за всі розрахунки всередині ігрового світу та відповідає за те, щоб вони були коректними [1]. Усі клієнти мають синхронізовувати стан своєї гри з даними серверу; при чому, будь-які розбіжності зі сторони клієнта нівелюються, так як стан клієнта буде перезаписано. Цей спосіб є популярним, оскільки має багато переваг для мультиплеєрної гри, деякими з яких є [16]:

1) Існування одного головного сервера, до якого повинні підключатися гравці, полегшує для них мережеве навантаження, оскільки не потрібно спілкуватися одночасно з рештою клієнтів.
2) Запускаючи гру на сервері розробників ми ховаємо дані гри від нечесних користувачів, які могли б дістати їх локально. Це дозволяє підвищити чесність та якість ігрового досвіду.
3) Гру, яка обчислюється на авторитарному сервері легше розробляти, оскільки будь-які помилки у розрахунках легше відстежуються.

Дану модель не варто плутати з клієнт-серверною архітектурою мережі, тому що вона описує фізичні пристрої, а дана модель є логічною схемою синхронізації даних, яка може бути реалізована як у клієнт-серверній, так і

peer-to-peer архітектурах, в останній один з гравців відіграватиме роль авторитарного сервера (хоста).

1.2.3. Обмін даними

Для обміну інформацією можна використовувати або вхідні дані гравця, такі як (хід на 3 клітинки вліво, кидок монетки), або стан гри (гравець знаходиться на клітці (13; 7), на монеті випала решка). [4] Кожен із варіантів має плюси та мінуси в залежності від контексту, в якому вони використовуються.

У будь-якому випадку, передаючи вхідні дані ми використовуємо менше трафіку, ніж нам потрібно було б для того, щоб передавати дані про стан гри, з урахуванням того, що для деяких ігор розмір цих даних може бути досить великим.

При зв'язку клієнт → сервер, передача вхідних даних є більш надійною, ніж передача стану гри від гравця, адже він не може вважатися надійним джерелом.

При зворотному зв'язку виходить інша ситуація, оскільки стан гри з сервера є коректним, на відміну можливого розрахунку гри клієнта.

Зазначене вище призводить до того, що найчастіше використовуються обидва підходи: передача ввідних даних від клієнта до сервера і передача стану гри від сервера клієнту.

1.2.4. Обробка ходу

Те, як сервер обробляє отримані дані користувача дуже впливає на ігровий процес. Для покрокових ігор ідеально підходять прості моделі з обробкою вхідних даних відразу при отриманні від гравця, якщо вони ходять по черзі, або ж із затримкою, поки інформація не надійде від усіх гравців, якщо вони здійснюють хід одночасно [12]. Навіть затримки та очікування в секунду в даному випадку не є критичними, оскільки немає швидкого темпу гри.

Проте для ігор в реальному часі, такий підхід звісно не спрацює добре через те, що гравці мають різну якість інтернет з’єднання, очікування ввідних даних кожного з них призводить до того, що усі гравці будуть мати таку затримку, яку має користувач з найбільш повільним з’єднанням. Тому в подібних проектах сервер не чекає доки надійде інформація від усіх клієнтів, а опрацьовує усі отримані дані й оновлює стан ігрового світу з певним інтервалом. Таким чином гра продовжує йти в «реальному часі» й надавати гравцям актуальну інформацію [2]

1.3. Постановка задачі

Метою роботи є розробка та реалізація багатокористувацької комп'ютерної гри жанру покрокова стратегія з тематикою відбудови міст. Для цього необхідно:

1. Сформувати концепцію гри
2. Розробити ігрові механіки
3. Розробити мережеву архітектуру проекту
4. Обрати інструменти реалізації додатку
5. Програмно реалізувати ігрову логіку
6. Створити інтерфейс гравця

2. Моделювання та проектування гри

2.1. Дизайн гри

Перед початком розробки будь-якої гри потрібно провести роботу над її дизайном, щоб визначити основні компоненти, такі як: тематика, цілі гравця, візуальне подання та механіки.

Основною **тематикою** нашої гри є відновлення зруйнованого війною міста. Гравцеві належить розміновувати місцевість, будувати нові споруди і стежити за якістю життя в місті. При цьому між однокористувацьким і багатокористувацьким режимами існує певна різниця в тематиці, яка полягає у відсутності та наявності бойових дій відповідно. Коли користувач грає сам, він перебуває в післявоєнному мирному часі, коли нічого його не відволікає від основної мети. Під час гри з кимось один гравець (далі атакуючий) відіграє роль сторони агресора і продовжує обстрілювати місто, не даючи другому (далі реконструктор) побудувати нове місто.

Беручи до уваги тематику, гра матиме назву ReBuilder.

Головною геймплейною **метою гравця**-реконструктора є отримання заданої кількості очок за задану кількість ходів. Атакуючий гравець зі свого боку повинен не дозволити цих очок набрати.

Візуальне представлення гри. Стратегії, а особливо покрокові, часто використовують сітку для поділу мапи на окремі блоки, клітинки. Це як спрощує життя розробникам (за такого поділу і легше програмувати ігрову логіку, і контролювати ігровий процес), так і дає гравцям можливість структурувати ігровий світ для себе. За багато років в іграх було створено і використано багато різних варіантів сіток: квадратні, шестикутні, трикутні, радіальні або просто графи без патернів поділу. Однак найпопулярнішими сьогодні є перші два типи, їх можна побачити у великій кількості стратегій

сьогодні. Стратегії, які фокусуються на побудові міст, найчастіше використовують квадратну сітку, оскільки вона має більш природний вигляд при формуванні мапи міста. Водночас стратегії, які орієнтуються на русі юнітів, часто використовують шестикутну сітку, бо вона дає більше простору для руху, а значить і тактики.

У нашій грі, нехай вона і фокусується на містобудуванні, не ставиться завданням саме побудови міста, та й будівлі займають лише одну клітину, тож квадратна сітка не дає нам переваг, проте шестикутна, у свою чергу, дає змогу легше візуалізувати ефекти, що застосовуються за радіусом, тож обрано було саме її.

У підсумку наш ігровий світ представлений шестикутною сіткою, з якою взаємодіють користувачі. На ній відображаються споруди, а також ефекти і візуальні дані.

Механіки – це те, як гравець, гра та її компоненти взаємодіють між собою [13]. Основа будь-якої механіки - це її можливість впливати на перебіг гри, стан ігрового світу. Вона визначає, як ігровий процес розвиватиметься далі, до яких наслідків призведе та чи інша дія гравця. [19]

Оскільки гра онлайн надає різний тип ігрового процесу для двох гравців, то і механіки у них відрізняються. Наступні механіки було реалізовано для гравця реконструктора:

Спорудження будівель – гравець може створювати будівлі на карті. Будівлі дають різний ефект в залежності від їх типу, але кожна будівля надає певну кількість очок перемоги за своє спорудження. Також кожне нове спорудження одного типу будівлі призводить до збільшення її ціни, що не дає змоги гравцеві користуватися одним типом для здобуття перемоги, а мотивує використовувати також інші.

Використання активностей – під час гри користувач може застосовувати дії на клітини, змінюючи їхній стан. Так само як і будівлі - використання активностей призводить до збільшення їхньої ціни, що змушує гравця ретельніше ставитися до їх використання.

Спорудження на руїнах – для безпечного спорудження будівель, гравець насамперед повинен очищати клітинку від руїн і розміновувати її, але так само він має можливість ризикнути і побудувати без попередньої роботи, в такому випадку є шанс отримати вибух. Під час вибуху гравець втрачає гроші, витрачені на зведення будівлі та робітників, які повинні були там працювати. Шанс вибуху визначається наявністю на цій клітині міни - якщо вона є, то вибух буде безперечно. На початку ігрової сесії на карті випадковим чином розставляються міни залежно від заданого значення замінованості від 0 до 1 (0 - мін взагалі не буде, 1 - міни будуть на кожній клітині).

Механіки, побудовані на випадковості, додають грі інтересу, адже гравець стає перед питанням ризику, коли можна отримати бажане, витрачаючи набагато менше, але водночас маючи можливість втратити навпаки ще більше. Азарт підігріває інтерес гравців і навіть програш викликає велику кількість емоцій, роблячи гру незабутнім досвідом. [20]

Заробіток очок – для перемоги в грі, користувач повинен набирати достатню кількість очок. Є два джерела заробітку - від спорудження будівель і від виробництва очок - деякі будівлі під час своєї роботи генерують певну кількість очок за хід.

Заробіток і витрата ресурсів – Під час сесії, гравець має займатися менеджментом ресурсів. У грі є три види ресурсів: гроші (вони ж валюта), робочі та їжа.

1) Гроші витрачаються на всі дії в грі - використання активностей, спорудження будівель і підтримання роботи деяких типів будівель.

50 од. валюти надходить гравцеві щоразу, коли він починає новий крок. Так само спеціалізовані будівлі генерують певну кількість валюти за крок.

2) Робітники необхідні для роботи деяких типів будівель. Гравець отримує робітників зі спорудження житлових будинків. Цей ресурс також може пропадати у гравця внаслідок описаних вище вибухів або описаних нижче травм, що робить його найскладнішим в утриманні.
3) Їжа потрібна для побудови житлових будівель. Гравець отримує їжу з побудови спеціалізованих будівель.

Забруднення території - кожна клітинга ігрового поля має властивість забрудненості. Вона може набувати значень від 0 до 100. Наприкінці кожного кроку розраховується загальна середня забрудненість ігрової мапи, і залежно від цього значення у гравця знімається певна кількість очок. Виділяється 4 рівні забрудненості:

- Легкий – значення від 15 до 35, за крок знімається 25 очок
- Середній – значення від 35 до 60, за крок знімається 50 очок
- Високий – значення від 60 до 90, за крок знімається 75 очок
- Екстремальний – значення від 90 до 100, за крок знімається 100 очок

Гравець забруднює територію шляхом побудови і запуску роботи виробництв, але так само має можливість очищати територію, використовуючи спеціальні активності та будуючи очисні споруди.

Активація і деактивація будівель - деякі типи будівель мають можливість зупиняти свою роботу. Будівлі, що працюють у такий спосіб, потребують ресурсів кожен хід, тому гравець отримує свободу виділяти ресурси на потрібні йому на даний момент напрямки, урізноманітнюючи геймплей.

Травматизація - працівники можуть отримувати травми на виробництві,

що призводить до їх втрати. Гравець може боротися з цим будуючи лікарні, які рятують робітників, що дозволяє не втратити їх назавжди.

Ця механіка тісно пов’язана з попередньою механікою зупинки виробництв для того, щоб зберегти робітників на потім, або перекинути їх на новий напрямок, якщо інші робітники втрачені.

Завершення ігрового кроку - час у покроковій грі дискретний і опрацювання всіх зроблених за хід дій відбувається в момент його завершення.

Захист будівель - під час гри онлайн, гравцеві-реконструктору надається можливість захисту своїх досягнень, шляхом спорудження спеціалізованих будівель захисту. Таким чином, гравець не залишається беззахисним перед своїм опонентом.

Далі йдуть механіки створені для гравця-атакуючого:

Бомбардування - гравець може завдавати шкоди шляхом бомбардування будівель іншого гравця. Цим самим він зносить побудовані будівлі, мінує територію і отримує натомість цього гроші для продовження гри. Коли будівля знищується, половина очок, які вона принесла під час будівництва, знімається у реконструктора.

Забруднення території - Атакуючий може бомбардувати територію спеціальним типом бомб для підвищення її рівня забруднення.

Розвідка місцевості - гравець отримує два види розвідданих: пряму візуалізацію будівель на мапі, під час застосування радара, або ж числове значення кількості будівель, що знаходяться навколо тієї клітини, куди був нанесений удар. Останнє було натхненне відомою всім грою "Сапер", в якій під час відкритті клітинки гравець отримує інформацію про те, скільки мін знаходиться по сусідству. Розвіддані допомагають атакуючому грати цілеспрямовано і створювати стратегію своєї поведінки, а не намагатися бити противника навмання.

У Таблиця 1 можна побачити всі створені для гри будівлі, а також в Таблиця 1.2 - всі активності, які усі разом реалізують описані вище механіки. Кожна з будівель і кожна активність унікальна, і використання практично кожної необхідне протягом гри, адже механіки створювалися в сукупності і доповнювали одна одну.

Таблиця 1.1 Будівлі

Малюнок	Назва англійською	Опис
	House	Будинок додає 3 робітників гравцю. Для побудови потрібно 20 од. грошей та 2 од. їжі. Додає 15 очок при побудові.
	Store	Магазин надає гравцеві 5 од. їжі, поки є активним. Для побудови потрібно 40 од. грошей та 1 робітник. Додає 10 очок при побудові.
	Factory	Фабрика виробляє 30 од. грошей за крок та забруднює сусідні клітинки в радіусі 3 на 5 пунктів та всю карту на 1 пункт кожний крок, поки активована. Шанс травмування 20 %. Для побудови потребує 60 од. грошей та 4 робітників. Додає 50 очок при побудові.
	Park	Парк очищає сусідні клітинки в радіусі 3 на 3 пункти та всю карту на 1 пункт, поки активний. Шанс травмування 3%. Для побудови потребує 30 од. грошей та 1 робітника. Додає 30 очок при побудові.

Малюнок	Назва англійською	Опис
	Science lab	Лабораторія виробляє 15 очок поки активна. Ганс травмування 15% Для побудови потребує 200 од. грошей та 5 робітників. Додає 50 очок при побудові.
	Monument	Монумент додає 40 очок при побудові. Для побудови потребує 40 од. грошей.
	Clinic	Лікарня може лікувати до 5 робітників за крок поки активна. Для побудови потребує 70 од. грошей та 30 од. грошей кожен крок, щоб бути активною. Додає 30 очок при побудові.
	Anti-air weapon	ППО надає можливість відбити один удар бомбою за крок, поки активна. Шанс травмування – 10%. Для побудови потребує 50 од. грошей та 2 робітників. Додає 30 очок при побудові. Доступна лише у багатокористувацькому режимі.

Таблиця 1.2 Активності

Малюнок	Назва англійською	Опис
	Sapper	Сапер розмінує клітинку, на якій застосовується.
	Cleaner	Очищає клітинку на 50 пунктів.

Малюнок	Назва англійською	Опис
Активності для користувача-атакуючого		
	Bomb	Підриває споруди на клітинці, з ймовірністю 50% мінує її, перетворює її в руїни. Дає атакуючому інформацію о кількості споруд навколо клітини.
	Airstrike	Працює як бомба, але застосовується на клітину та усіх її сусідів в радіусі 1 та не надає інформації щодо навколишніх споруд.
	Dirty bomb	Забруднює клітину та усі сусідні в радіусі 1 на 30 пунктів.
	Radar	Надає атакуючому інформацію о спорудах на клітині та усіх сусідніх в радіусі 1

2.2.Моделювання ігрового процесу

Моделювання ігрового процесу є корисним для розуміння, що буде роботи гравець від час гри. Ми розробили три flow-діаграми, які відображають алгоритм дій гравця-реконструктора, гравця-атакуючого та діаграму загального використання додатку. Таким поділ було зроблено по декільком причинам: по-перше гравець-реконструктор та гравець-атакуючий мають різний підхід до гри та різні можливості, по-друге ігровий процес було виокремлено в окремі діаграми, щоб не змішувати логіку меню та логіку гри між собою.

Отже на Рисунок 2.1 ми бачимо діаграму реконструктора. Його алгоритм дій виокремлює те, що на початку гри він обов’язково має побудувати будинок та магазин, які дають ресурси для початку гри. Вже далі гравцеві надається

свобода у своїх діях. Загально крок виглядає таким чином, що гравець аналізує наявний стан гри і відповідно цього робить вибір чи потрібно і чи може він використати активність або побудувати потрібну споруду, що він може повторювати декілька разів за крок, потім завершуючи його. В кінці кроку відбувається розрахунок стану і гравець переходить на наступний крок. І так продовжується до тих пір, поки кількість кроків не сягне максимального значення, коли перевіряється чи досяг гравець потрібного значення очок, відповідно до чого він або програє або перемагає.

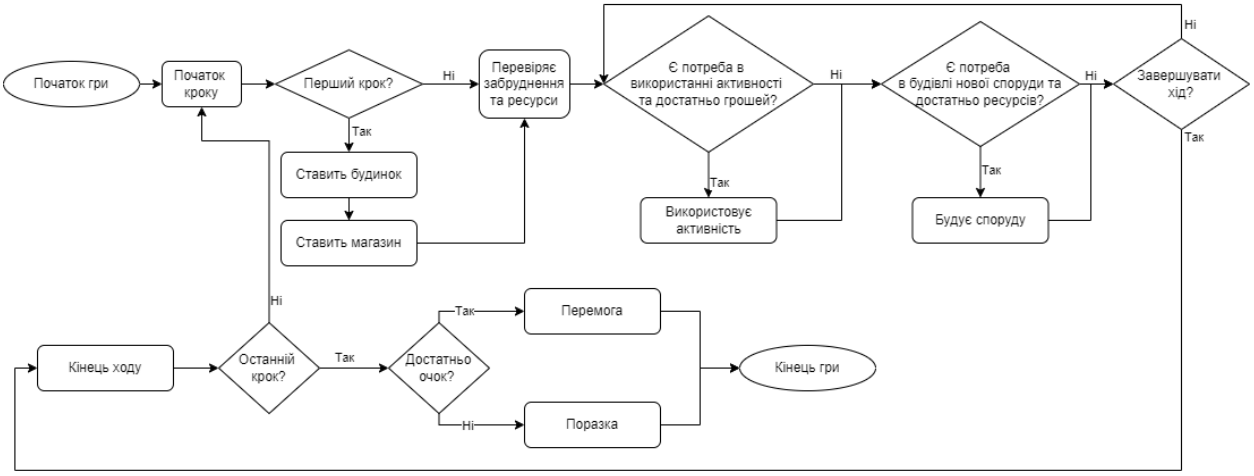

Рисунок 2.1 – Flow-діаграмма процесу гри для реконструктора

На Рисунок 2.2 зображена діаграма атакуючого. Гравець не має початкового обов’язку робити певні дії і з самого початку він може діяти як хоче. Загально його крок виглядає таким чином: він аналізує дані щодо споруд супротивника, які в нього є, і якщо має можливість та потребу збирає нові. В залежності від цього він застосовує той тип зброї, який йому потрібен. Чим відрізняється ігровий процес атакуючого гравця, так це тим, що він не має змоги завершити крок самостійно і очікує допоки це зробить інший гравець. Коли ж гра дійде до кінця, то гравець перемагає, якщо противник не зміг набрати потрібної кількості очок.

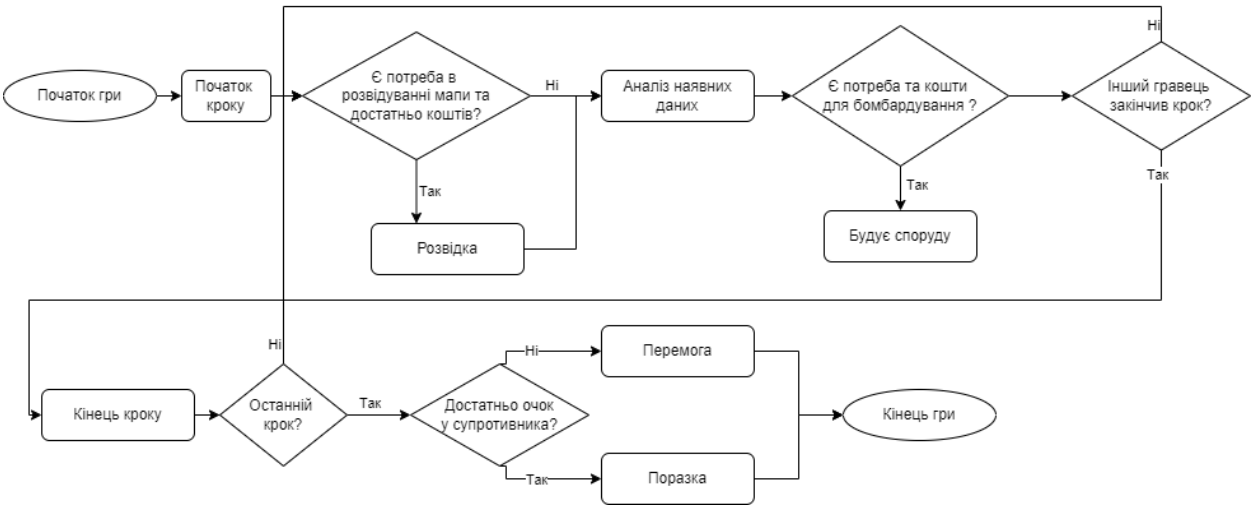

Рисунок 2.2 – Flow-діаграмма процессу гри для атакуючого

Загальний алгоритм використання додатку зображено на Рисунок 2.3. Якщо гравець бажає бути реконструктором, то він створює новий сервер та гру, налаштовує її та чекає допоки інший гравець приєднається. Тільки він може запускати та налаштовувати гру. Інший же гравець, повинен ввести адресу по якій він буде приєднуватися, після чого очікувати початку гри. Після закінчення самої гри, щоб почати нову сесію, гравцям треба буде повторити даний алгоритм.

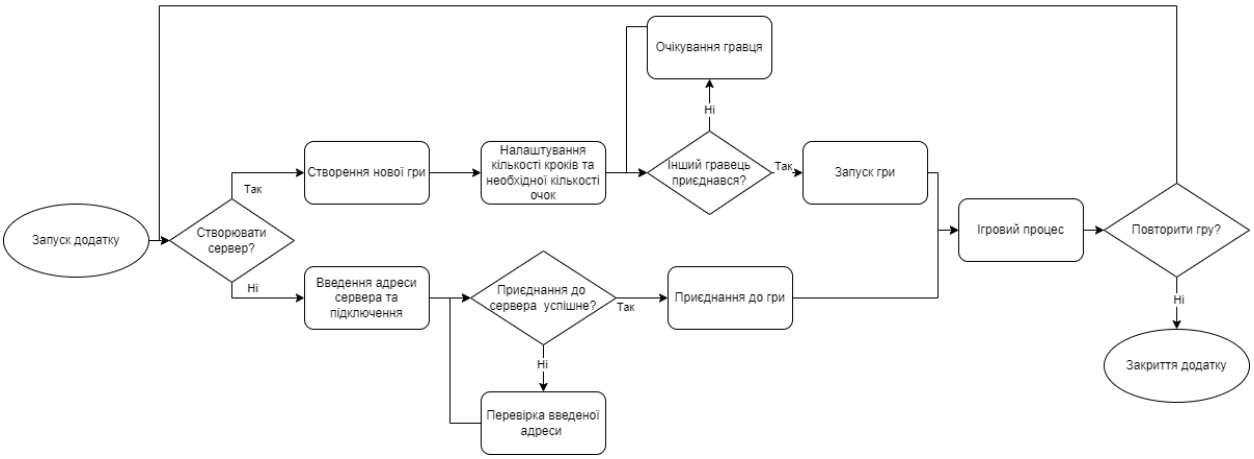

Рисунок 2.3 – Flow-діаграма користування додатком

2.2.1. Ігрові стани

Машини станів, а більш конкретніше – скінчена машина станів, майже завжди використовувалися в розробці ігор і використовуються по цей день. FSM - це прості машини станів, в яких різні стани пов'язані між собою деякими умовами. Якщо певні умови виконуються, то ми переходимо з одного стану в інший. Усередині стану ми можемо виконувати певні дії або реалізовувати алгоритми [10]. Виокремлюючи стани та реалізуючи специфічну для них логіку, ми досягаємо зменшення зв’язаності коду та полегшуємо собі роботу. Часто машини станів використовують для анімацій, описання різних стані контрольованого персонажа та штучного інтелекту[11]

В нашій грі ми також використовуємо машину станів для контролю дій гравця. В залежності від його дій він буде знаходитися в певному стані. Так на Рисунок 2.4 зображена машина станів для гравця-реконструктора. Вона складається з трьох станів:

- Initial State – початковий стан гравця, в якому він має здійснити перші побудови для початку гри.
- View State – стан в якому гравець переглядає карту та може взаємодіяти зі спорудами на ній – почати/зупинити роботу промислових будівель, побачити радіус забруднення споруди. Також в цьому стані гравець може активувати перегляд загального стану забрудненості мапи.
- Place State – гравець переходить в цей стан, коли він хоче здійснити активність або побудувати споруду. Так як загалом логіка обох дій криється в відображені гравцю куди буде виконана дія, обирання клітинки та виклик ефекту будівлі/активності, то ці дві дії було поєднано в один стан.

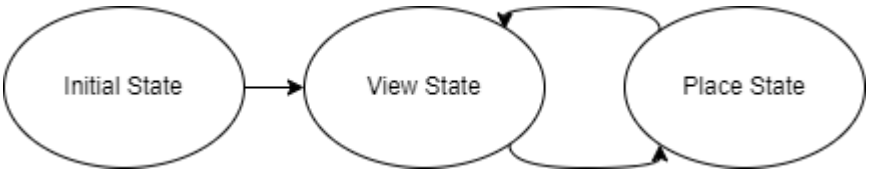

Рисунок 2.4 – Машина станів гравця-реконструктора

Для гравця-атакуючого машина станів зображена на Рисунок 2.5. В ній присутній аналогічний ViewState, але додано окремий War State, який схожий по своїм потребам на Place State (відображені гравцю куди буде виконана дія, обирання клітинки та виклик ефекту), але через особливість реалізації мережевого коду, його було перенесено до окремого стану.

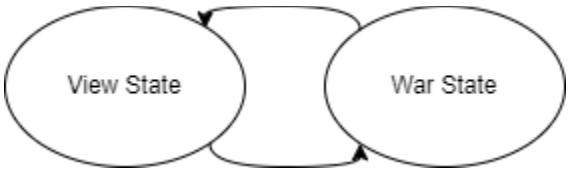

Рисунок 2.5 – Машина станів гравця-атакуючого

2.3.Мережева архітектура проекту

2.3.1. Клієнт-серверна архітектура

Тепер обговоримо, як гра працюватиме по мережі. Тут потрібно почати з архітектури і наш проект використовує модель клієнт-сервер з прямим з’єднанням між гравцями (Рисунок 2.6). Один з гравців створює гру, тим самим відіграючи роль сервера, до якого будуть надалі приєднуватися інші гравці. Так як проект невеликий, то і розрахунку на створення та використання окремого сервера немає, бо це включає в себе зайві затрати, які далеко не факт, що себе окуплять. При цьому, ми надаємо можливість людям самим створювати собі гру та грати в неї без обмежень, які могли б трапитися при використані власного серверу.

Рисунок 2.6 – Мережева структура проекту

2.3.2. Синхронізація та обробка даних

Для синхронізації даних ми будемо використовувати модель авторитарного сервера, так як один із гравців буде виступати в ролі сервера і зберігати в себе весь стан ігрового світу. Як уже було зазначено раніше, система авторитарного сервера має на увазі, що цей сервер займається обчисленнями і кожен гравець повинен просто отримувати від нього дані у відповідь на своє введення. Оскільки ми реалізовуємо покрокову гру, в якій усі дії виконуються миттєво, нам також не потрібно розбиратися із забезпеченням плавності геймплею, використовуючи техніки на кшталт інтерполяції та/або екстраполяції стану ігрових об'єктів.

Розглянемо спілкування клієнта і сервера між собою. Клієнт передаватиме серверу свої вступні дані для здійснення ходу, далі сервер визначає, чи може цей хід узагалі бути здійснений, якщо так, то відбуваються відповідні обчислення і сервер повертає гравцеві результат, який він зобов'язаний застосувати. Цей процес може бути повторений безліч разів за крок, поки він не закінчиться.

2.4.Вибір ігрового рушію

Створення ігор - це комплексний процес, адже для кінцевого результату потрібна робота з абсолютно різними аспектами гри, такими як звукове оформлення, робота з графікою, створення інтерфейсу, саме програмування ігрової логіки, розробка візуальних ефектів тощо.

Великі студії, які мають достатньо коштів, можуть собі позволити написання особистих ігрових рушіїв, з якими потім вони і працюють. Таким, наприклад, є рушій Frostbite, розроблений компанією DICE, на якому створювалися такі ігри як Dead Space (ремейк 2023 року), Battlefield 2042, FIFA 2023 [18].

Проте створення свого рушія це дуже важка справа, яка потребує багато людино-годин та грошей, багато студій великих і маленьких використовують вже розроблені рушії інших компаній, які вклали в це достатньо коштів і надали доступ до них світові. На даний момент одними з найпопулярніших рушіїв можна назвати: Unreal Engine, Unity та Godot.

Unreal Engine за довгі роки свого існування напрацював собі репутацію рушія для великих 3D проектів. Все це тому, що рушій сам по собі є доволі складним у вивчені та використанні і причин цьому декілька. По-перше Unreal Engine намагається бути інструментом для всього – в самій програмі при створені нового проекту ми можемо побачити на вибір створення нової гри, архітектурного проекту, симуляції або проекту для створення фільмів. Через це рушій просто становиться переповненим багатою кількістю різних функцій. Також при знайомстві з Unreal Engine ви одразу розумієте, що багатий функціонал не завжди працює так, як те потрібно вам, через це цей функціонал треба «дороблювати» під себе. І не можна забути про те, що рушій намагається бути флагманом в своїй області. Нові технології які з’являються в нових версіях рушія приваблюють до себе багато нових користувачів, проте рушій в

свою чергу стає все більш тяжким для комп’ютерів, як і ігри, що робляться на ньому. Через свою складність та надмірну потужність для невеликих проектів, часто він остається осторонь в цій категорії.

Unity навпаки зарекомендував себе як найпопулярніше рішення серед розробників одинаків та невеликих команд. І це не дивно, він є порівняно легким у використані, не є настільки важким як Unreal Engine і добре підходить для багатьох платформ, в тому числі і мобільних, які є найбільшим ринком відеоігор на даний момент. Також даний рушій доволі різноманітний в плані типів ігор, які можна на ньому створити – це як великі та потужні 3д ігри з відкритим світом, так і маленькі 2д ігри. Проте Unity є закритою платформою без відритого коду, тому його не можна настроювати під себе, якщо це стане в край необхідно, а також ви повністю залежите від компанії, якій належить рушій. Так, у 2023 році, Unity вже зіпсувало свою репутацію спробою ввести комісію за кількість встановлень гри, що викликало велику хвилю обурень серед розробників.

Godot [6] в свою чергу є легким рушієм з повністю відритим кодом без обмежень у його використані. За останні роки він набирає все більшу популярність завдяки тому, що він ідеально підходить для створення 2д проектів, з яких найчастіше і починають новачки, а також є доволі легким у вивчені. Він має достатньо великий функціонал для вирішення задач розробки ігор, так, наприклад стосовно нашої гри, Godot надає зручне API для створення онлайн ігор, засноване на використанні UDP протоколів, які вважаються стандартними для ігрової індустрії та має інструменти для розробки мапи з гексагональною сіткою.

Тому, для написання гри буде використовуватись саме рушій Godot, як легке, безкоштовне, але достатньо функціональне рішення.

3. Програмна реалізація

3.1. Архітектура додатку

При роботі з рушієм Godot, ми розробляємо гру як древоподібну сукупність нодів, які ми поєднуємо в сцени. Ноди - це найменші будівельні блоки гри, такі як ігровий спрайт, камера, форма колізії – тобто це найменші самодостатні класи, які присутні у рушії і які можемо створювати самостійно. Ми поєднуємо ноди у сцени, щоб створити більш складну логіку, наприклад персонажа або мапу, які потім ми можемо використовувати щоб створювати інші сцени. [21]

Для нашої гри було розроблено наступну архітектуру додатку, представлену також на Рисунок 3.1:

- Root – головне вікно та головна нода програми, до якої ми приєднуємо наші сцени
- Main Scene – головна сцена додатку, яку створюємо ми. Це головний екран гри, який бачить користувач, як тільки запускає додаток. Головна сцена міститиме опції для створення та приєднання до гри.
- Lobby – сцена, в якій користувачі збираються перед початком гри. Також гравець, який створює гру, може налаштувати її саме тут і запустити.
- Game Scene – сцена, в якій відбувається увесь ігровий процес.
- Playmap – сцена, яка представляє ігрову мапу, з якою взаємодіє гравець.
- Explosion – сцена, яка є налаштованим партиклем для візуалізації вибуху.
- HUD (Heads-Up Display) – сцена, яка представляє інтерфейс гравцеві-реконструктору під час гри.

- WarHUD – сцена, яка представляє інтерфейс гравцеві-атакуючому під час гри.
- MenuButton – кнопка, яка використовується для динамічної побудови меню у сценах HUD та WarHUD.

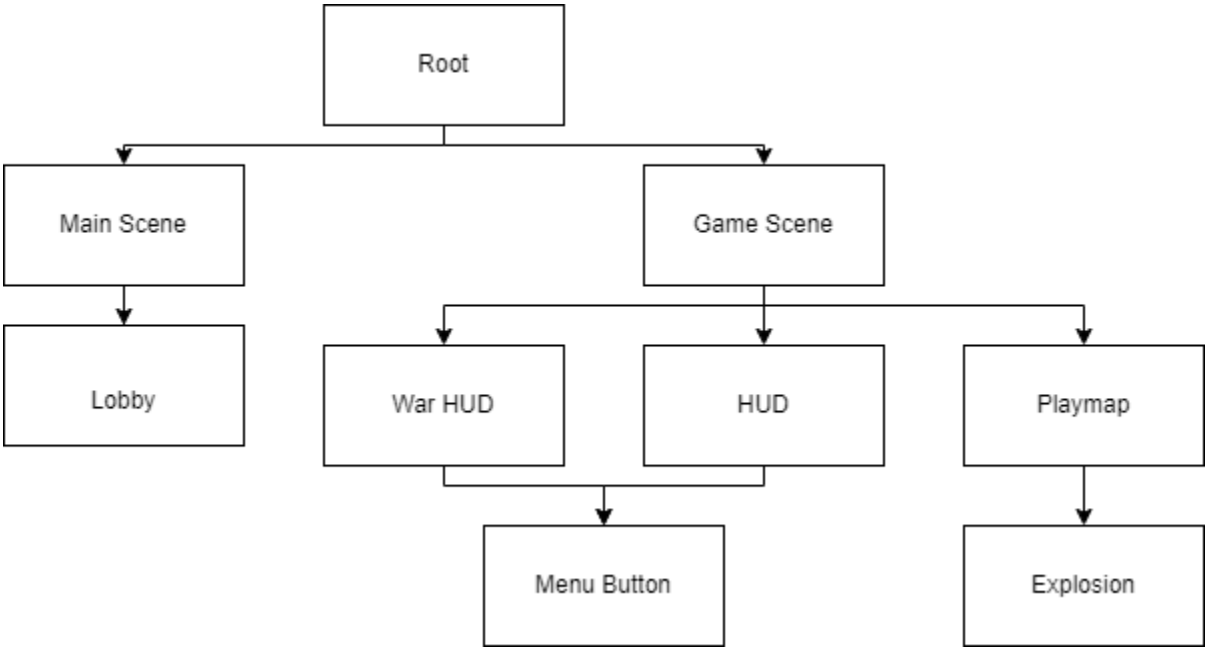

Рисунок 3.1 – архітектура сцен додатку

3.2.Створення ігрової мапи

3.2.1. Створення мапи

Як було зазначено раніше, наша мапа представляє шестикутну сітку. Для роботи з нею ми будемо використовувати клас TileMap. Загалом тайлмапи є доволі популярним рішенням при розробці 2д ігор. Воно полягає в тому, щоб створювати великі візуальні об’єкти, такі як мапа, або фонове зображення, з маленьких зображень, які називають тайлами. Часто вони мають прямокутну або шестикутну форму та створюються таким чином, щоб при поєднані створювалося враження цілісної текстури [23] . В годоті тайлмапа потребує для роботи створення тайлсету – набору тих самих тайлів (Рисунок 3.2). В налаштуваннях вказуємо, що тайл має бути розміру 64 на 64 пікселі та мати

шестикутну форму. Також опція UV Clipping дозволяє не виводити на екран ті частини текстури, які виходять за межі шестикутника.

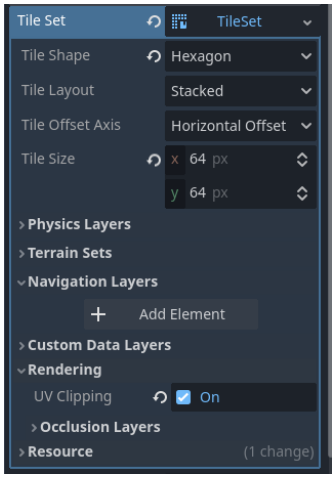

Рисунок 3.2 – Налаштування тайлсету

За допомогою програми Aseprite було створено тайлову текстуру розміром 320 на 320 пікселів, що відповідає 25 текстурам 64 на 64 пікселів у 5 рядів по 5 штук на кожний (Рисунок 3.3). Зображення були створені у вигляді піксельної графіки. Піксельна графіка, або піксель арт, це такий тип цифрової графіки, де художник контролює кожний окремим піксель зображення [22]. Даний тип графіки дуже популярний серед розробників-одинаків та невеликих студій, так як дозволяє навіть не художникам створити більш-менш приємне на вигляд зображення. Хоча піксельна графіка все ще залишається одним з різновидів створення малюнку, тому для створення великих та пропрацьованих зображень вона потребує вмінь та знань.

Рушій Godot автоматично імпортує усі файли, що знаходяться всередині проекту, тому додатково опрацьовувати зображення для рушію не потрібно. І зображення просто було додано до тайлсету.

Рисунок 3.3 – Створені асети для тайлмапи

Надалі було намальовано мапу руїн 12 на 12, використовуючи відповідну текстуру з тайлсету (Рисунок 3.4). Для легшої роботи з об’єктом TileMap було створено різні шари, на яких можна малювати текстури. Загалом таких шарів було створено 4:

- Background – текстура самої клітини, на цьому шарі створено руїни
- Buildings – для відображення споруд на клітині
- Outline – використовується для відображення, що клітина виділена.
- Overlay – для відображення додаткової інформації, як підсвічення радіусу забруднення або передпоказу встановлення будівлі.

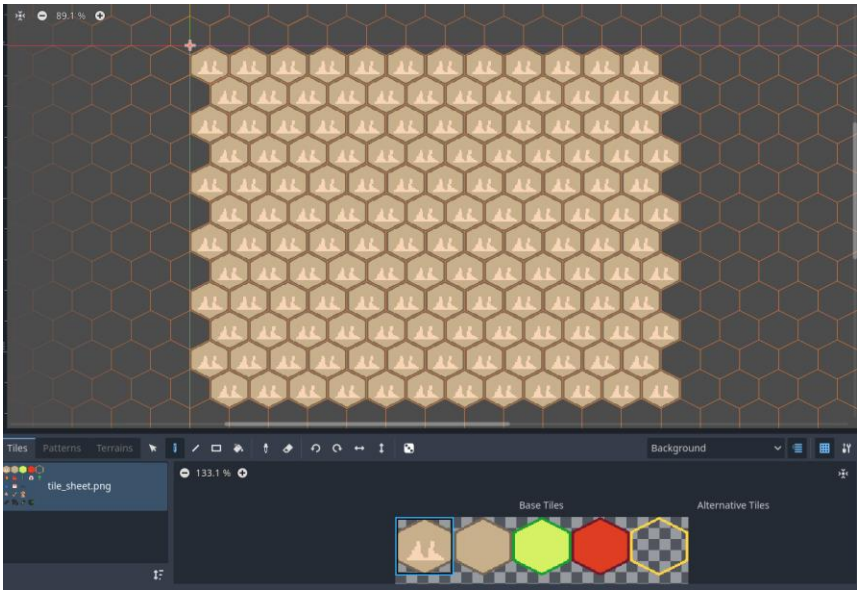

Рисунок 3.4 – Створена мапа

Для того, щоб гравцеві було легше працювати з мапою, було створено скрипт ViewController, який додається як дочірній об’єкт до будь-якого об’єкту сцени та надає йому наступні можливості: приближати, віддаляти та переміщати його. В кінцевому варіанті сцена Playmap має наступний вигляд:

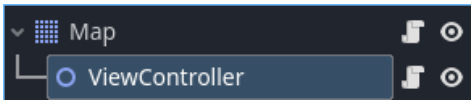

Рисунок 3.5 структура сцени Playmap

3.2.2. Ресурси клітинки

Під час гри ми маємо зберігати усі зміни, що відбуваються з мапою. Так як клас TileMap, вбудований в рушій, використовується загалом для відображення тайлів та збереження статичної інформації, тому нам довелось розширити клас та додати класи для збереження усіх потрібних нам даних. Діаграма класів зображена на Рисунок 3.6. Загальний опис класів представлений в Таблиця 3.1.

Таблиця 3.1 Загальний опис класів діаграми PlayMap

Клас	Призначення
PlayMap	Розширений клас TileMap, до якого додано масив клітинок мапи, які представлені класом TileInfo. Також у цьому класі реалізовано допоміжні функції для покращення роботи з масивом та мапою.
TileInfo	Зберігає стан клітинки мапи.
TileItem	Батьківський клас для усіх класів об’єктів, які можуть бути застосовані до клітини.
TileResource	Використовується для збереження загальної та візуальної інформацію для кожного з типів TileItem-ов

Клас	Призначення
TIBuilding	Батьківський клас усіх інших класів споруд. Додає функцію прибирання споруди з клітини.
TIAction	Батьківський клас усіх активностей у грі

Описання інших класів не має сенсу, оскільки вони відповідають представленій раніше інформації щодо будівель та активностей гри.

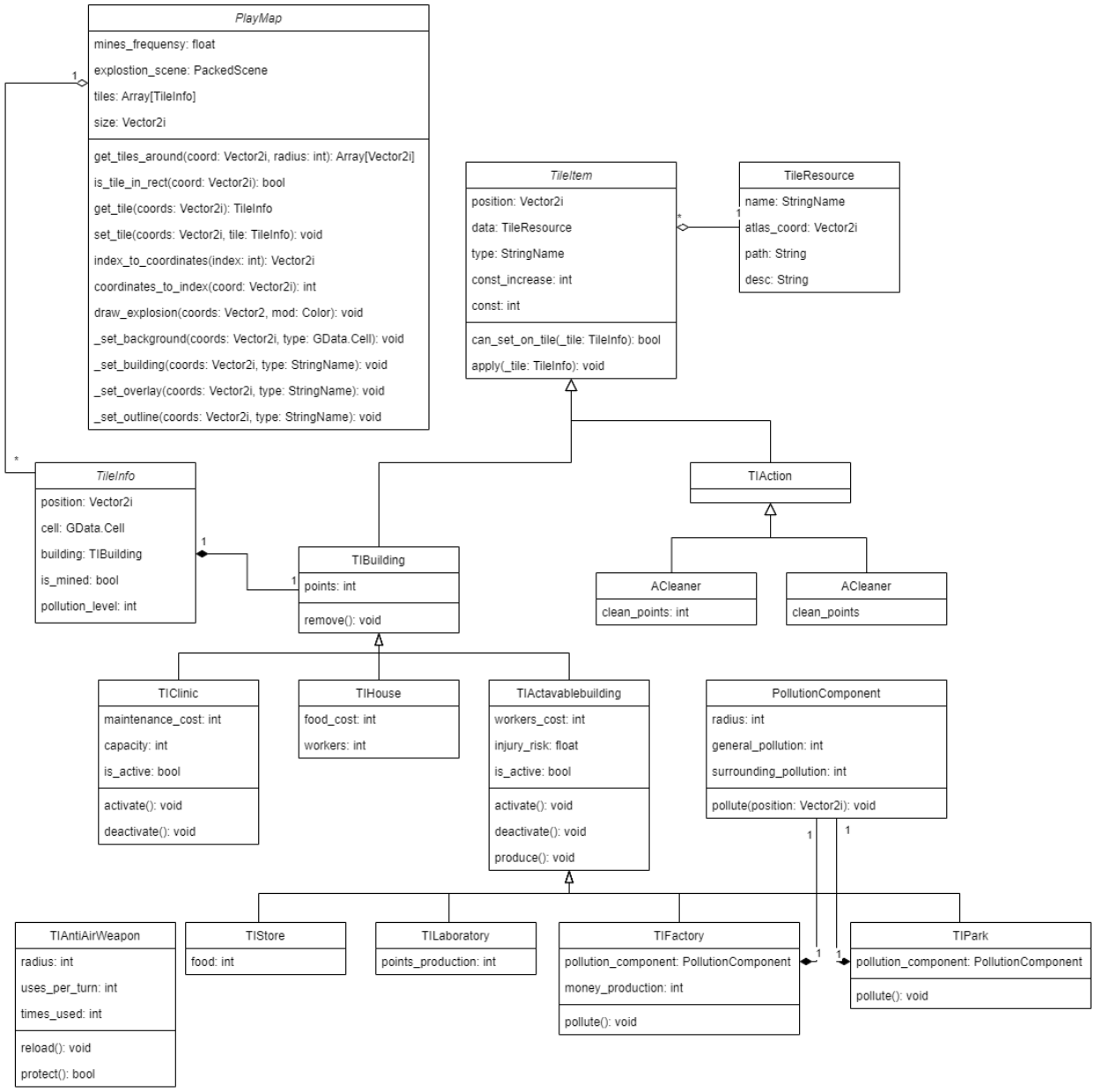

Рисунок 3.6 – Діаграма класів PlayMap

3.3.Створення HUD сцен

3.3.1. HUD

Інтерфейс гравця реконструктора має відповідати наступним вимогам:

- Гравець повинен бачити кількість ресурсів (їжі, робітників та грошей), які в нього є на даний момент часу
- Гравець має бачити на якому кроці він зараз знаходиться та скільки очок він вже зміг отримати.
- Гравець повинен мати можливість переглядати рівень забрудненості
- Гравець повинен мати доступ до усіх активностей, які він може застосувати, та споруд, які він може побудувати
- Гравець повинен мати можливість завершити крок

Беручи дані вимоги до уваги, було розроблено наступний варіант інтерфейсу, який зображено на Рисунок 3.7.

Рисунок 3.7 – Інтерфейс HUD

При натисканні кнопок Actions та Buildings гравцеві буде відображатися меню, в якому можна обирати активність або споруду. Саме меню створюється під час роботи додатку, для цього була створена окрема сцена – кнопка меню

Рисунок 3.8. До цієї кнопки також було додано скрипт за допомогою якого її можна модифікувати, тим самим встановлювати потрібні назву, текстуру та ресурси (Рисунок 3.9).

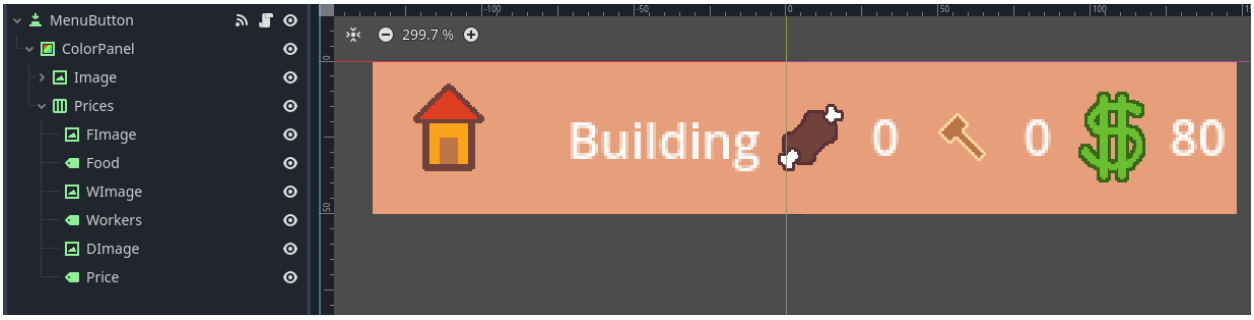

Рисунок 3.8 – Сцена MenuButton

```gdscript
func set_image(texture: Texture2D) -> void:
    $ColorPanel/Image.texture = texture;

func set_desc(desc: String) -> void:
    $ColorPanel/Image/Name.text = desc;

func set_price(price: int) -> void:
    $ColorPanel/Prices/Price.text = str(price);

func set_workers(workers: int) -> void:
    $ColorPanel/Prices/WImage.show();
    $ColorPanel/Prices/Workers.show();
    $ColorPanel/Prices/Workers.text = str(workers);

func set_food(food: int) -> void:
    $ColorPanel/Prices/FImage.show();
    $ColorPanel/Prices/Food.show();
    $ColorPanel/Prices/Food.text = str(food);
```

Рисунок 3.9 – Функції модифікації кнопки меню

Для відображення забрудненості мапи було створено клас TextOverlay, який створює текстові надписи для кожного з тайлів та додає червоний оверлей на мапі, що зображено на Рисунок 3.10.

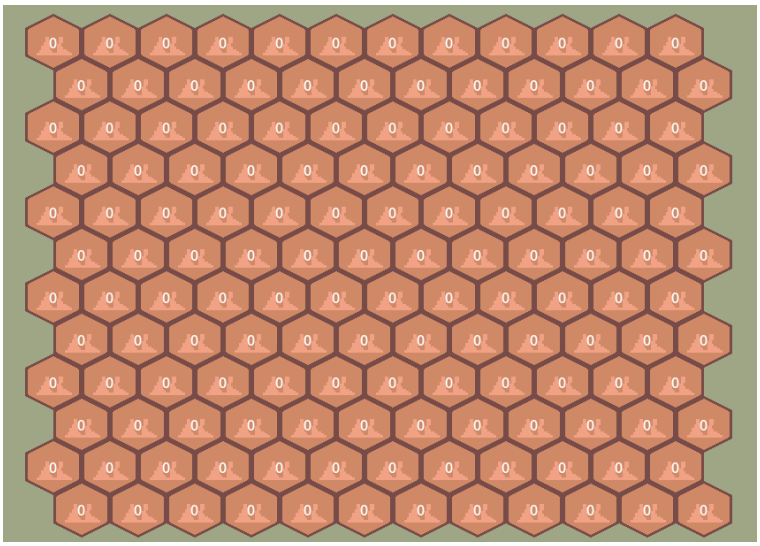

Рисунок 3.10 – відображення забрудненості

3.3.2. WarHUD

Інтерфейс гравця-атакуючого дуже схожий на інтерфейс реконструктора і представляє собою спрощену його варіацію (Рисунок 3.11), так як він не має можливості для перегляду забруднення території, побудови споруд, закінчення кроку та йому не потрібне ніякі ресурси окрім грошей.

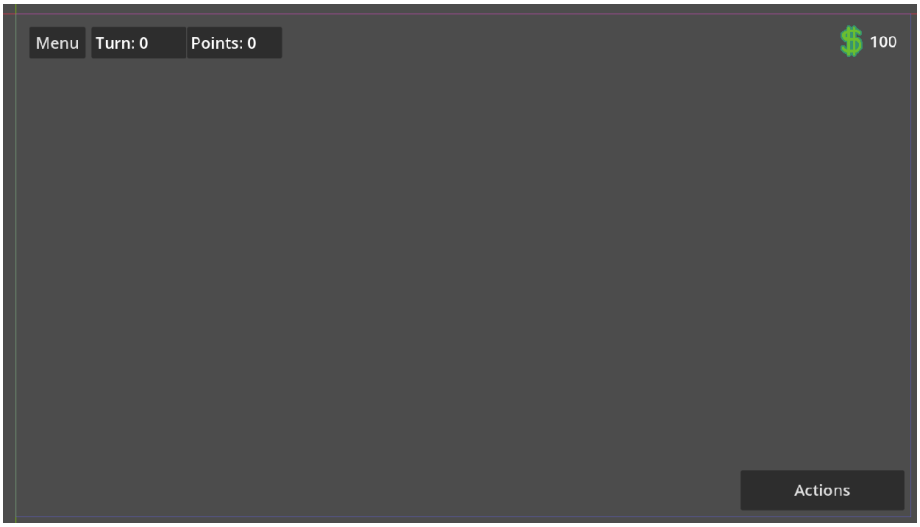

Рисунок 3.11 – WarHUD

3.4.Створення сцен головного меню

Головне меню складається з двох сцен: це саме головне меню та лобі. В головному меню, гравець повинен мати можливість ввести свій нікнейм, створити сервер або ввести адресу іншого користувача та доєднатися до нього (Рисунок 3.12). В лобі обидва користувачі бачать список гравців, а гравець-сервер має можливість налаштовувати гру, клієнт же може лише бачити, що налаштовує сервер. Також лобі містить кнопку запуску гри (Рисунок 3.13).

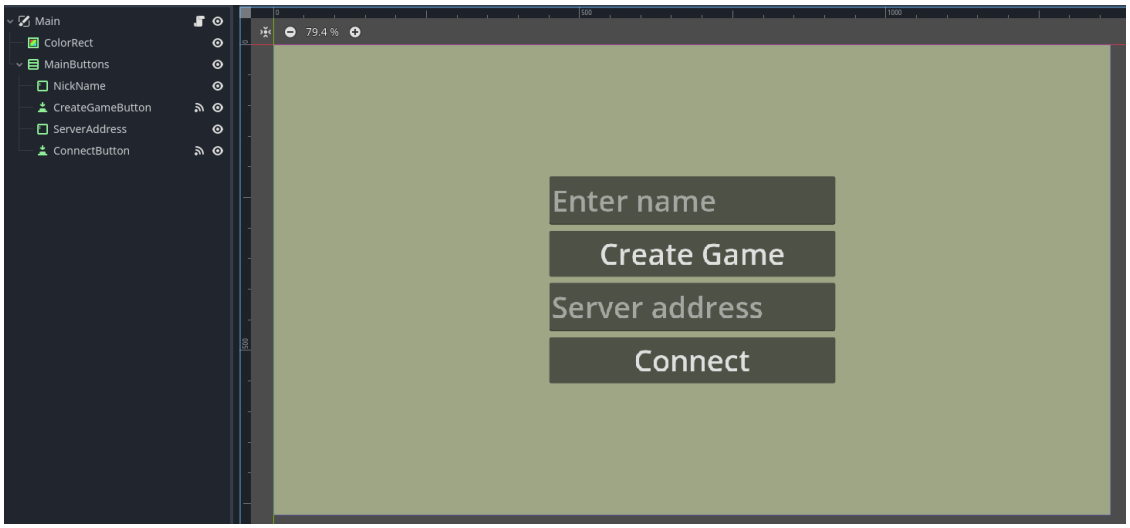

Рисунок 3.12 – інтерфейс головного меню

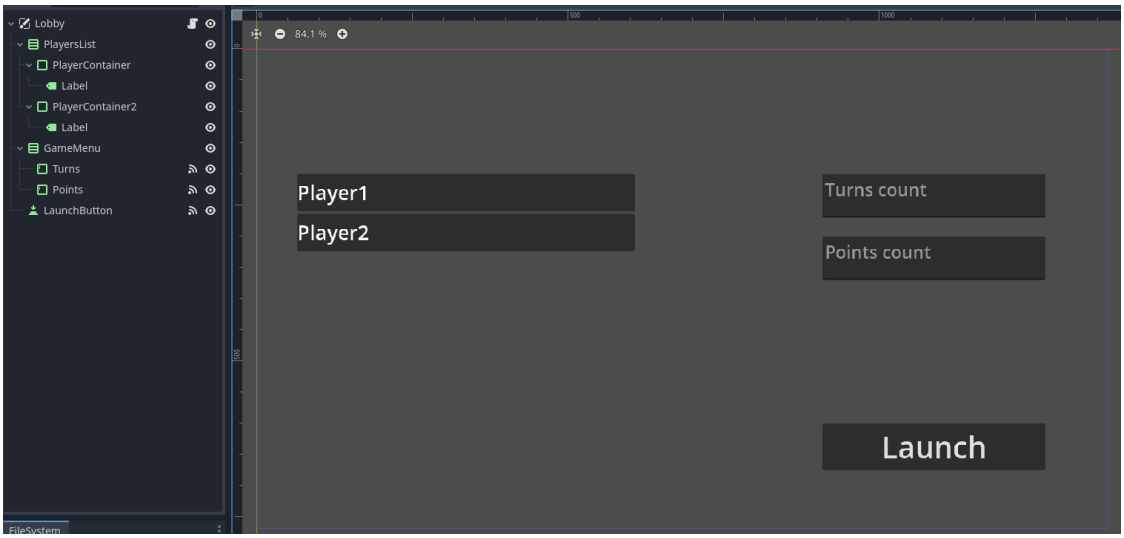

Рисунок 3.13 – інтерфейс лобі

В цих сценах починається використання мережевого коду. Для спілкування клієнтів по мережі, Godot використовує RPC (remote procedure call або виклик віддалених процедур). За допомогою цього ми на одному клієнті маємо можливість викликати функції на іншому клієнті. Рушій вимагає, що функції мають бути створені на обох клієнтах та мати однаковий шлях в дереві сцен. В даному випадку це не є проблемою, так як і клієнт і сервер будуть використовують одну і ту ж сцену головного меню.

Коли новий гравець приєднується до гри, то на всіх користувачах викликається функція на додавання цього гравця в загальний список гравців та відобразити у меню лобі (Рисунок 3.14).

```
func _on_peer_connected(id: int) -> void:
    Server.add_player.rpc_id(id, multiplayer.get_unique_id());
    lobby.construct_player.rpc_id(id, nickname_field.text)
```

```
@rpc("any_peer", "call_local")
func construct_player(nickname: String) -> void:
    if !nickname:
        nickname = str(multiplayer.get_remote_sender_id());
    Server.players[multiplayer.get_remote_sender_id()].name = nickname;

    var panel := PanelContainer.new();
    panel.custom_minimum_size = Vector2i(0, 60);
    player_list.add_child(panel);
    var label := Label.new();
    label.text = nickname;
    label.add_theme_font_size_override("font_size", 32);
    panel.add_child(label);
```

Рисунок 3.14 – код додавання приєднаного клієнта

Крім того, щоб оновлювати дані на клієнті, коли сервер налаштовує гру, було також створено спеціальні функції, які викликаються при зміні значення текстового поля. Це досягається завдяки системі сигналів – вони викликаються, коли відбувається певна подія. До них можна прив’язати виклик інших функцій, що дуже корисно, щоб не перевіряти постійно чи не трапилась та чи інша подія, ми просто чекаємо на її виконання. Таким чином реалізовано і в нашому випадку, що продемонстровано на Рисунок 3.15. Також ми бачимо, що

і початок нового кроку на сервері буде викликати це і на клієнті.

```
func _on_turns_text_changed():
    update_clients_turns.rpc(turns.text);

func _on_points_text_changed():
    update_clients_points.rpc(points.text)

@rpc("authority","call_remote")
func update_clients_turns(text: String) -> void:
    turns.text = text;

@rpc("authority","call_remote")
func update_clients_points(text: String) -> void:
    points.text = text;

@rpc("authority", "call_local")
func launch_game() -> void:
    get_tree().change_scene_to_file("res://scenes/game/game.tscn");
```

Рисунок 3.15 – оновлення введених сервером даних на клієнті

3.5. Мережеве з’єднання

Клас Server є головним за зв’язок клієнта та сервера під час гри. Так як геймплейної мережевої логіки в нашій грі небагато, то вся вона міститься в цьому класі. Він відповідає за виконання активностей гравця-атакуючого, а також зберігає список гравців.

Як вже було зазначено раніше – для роботи rpc потрібно, щоб шлях дерева до виклика функції був однаковий з обох сторін, для цього в рушії ми додамо наш клас сервер до списку Autoload, який створює об’єкт заданого класу на початку гри та розміщує його як дитину Root – кореня усього дерева гри. Таким чином ми отримуємо рішення схоже на патерн синглтон з глобальним доступом до об’єкту цього класу. Проте хрестоматійним виконанням синглтону це рішення не є, так як все ще можна зробити новий об’єкт даного класу і використовувати його.

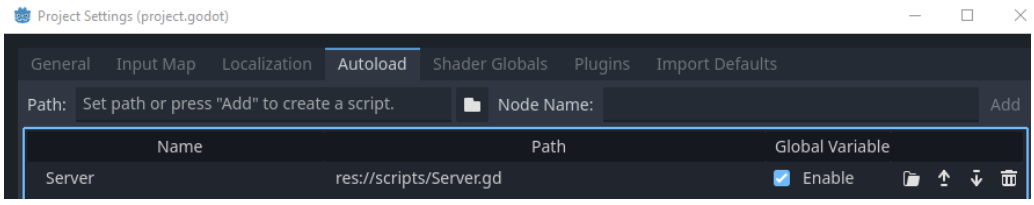

Рисунок 3.16 – додавання серверу до Autoload

ВИСНОВОК

Ігри на сьогоднішній день є дуже популярним способом проведення вільного часу і з кожним днем привертають до себе все більше користувачів по всьому світу, що явно демонструє статистика популярних платформ.

У ході аналітичної роботи ми з'ясували, що жанр стратегії є вкрай широким і містить у собі безліч інших піджанрів. Досліджували різні підходи до створення мультиплеєрних проектів: підходи до архітектури мережного зв'язку між гравцями, такі як peer-to-peer та клієнт-серверна модель; використання на сьогоднішній день моделі синхронізації типу «авторитарний сервер»; використання вхідних даних користувача та даних стану гри для обміну інформацією між гравцями та сервером; а також розглянули способи обробки даних на сервері, що ґрунтуються на миттєвих обчисленнях при надходженні даних, обчислень з постійним інтервалом.

У ході виконання кваліфікаційної роботи було виконано такі завдання:

1. Сформовано концепцію гри: ігрова мапа представляє собою двомірну сітку, кожна клітина якої є шестикутником; гравців поділено на реконструктора та атакуючого, які мають різні цілі та геймплейні особливості.
2. Розроблено загалом дванадцять взаємопов’язаних ігрових механік, таких як спорудження будівель, використання активностей, спорудження на руїнах, заробіток очок, менеджмент ресурсів, забруднення території, активація будівель, травматизація, завершення ігрового кроку, захист споруд та бомбардування з розвідкою.
3. Спроектовано мережеву архітектуру додатку, яка представляє собою реалізацію клієнт-серверної моделі, з використанням моделі авторитарного серверу як способу синхронізації та обробки даних.

4. Обрано ігровий рушій Godot, як інструмент для програмної реалізації гри та програму Aseprite для створення візуальної частини додатку.
5. Програмно реалізовано ігрову мапу та систему для зберігання та обробки її стану, створено класи активностей та будівель, які є інструментами взаємодії з ігровим світом. Реалізовано машину станів для контролю ігрового процесу користувачів.
6. Створено інтерфейс для гравців, який включає в себе головне меню, меню лобі та інтерфейс для взаємодії з ігровим світом.

Надалі планується додавання до гри штучного інтелекту, щоб надати можливість користувачам грати в багатокористувацький режим одноосібно. Штучний інтелект буде реалізовано як для гравця-реконструктора, так і атакуючого. А також додавання нових типів клітин мапи, таких як водні клітини, що буде урізноманітнювати ігровий процес за допомогою ускладнення мапи. І створення нових механік для покращення та поглиблення ігрового досвіду користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Client-Server Game Architecture - Gabriel Gambetta. *Gabriel Gambetta*. URL: https://www.gabrielgambetta.com/client-server-game-architecture.html (дата звернення: 08.05.2024).
2. Entity interpolation - gabriel gambetta. *Gabriel Gambetta*. URL: https://www.gabrielgambetta.com/entity-interpolation.html (дата звернення: 08.05.2023).
3. Authoritative Servers, Relays & Peer-To-Peer - Understanding Networking Types and their Benefits for each Game Types. *Edgegap - Game Server Hosting, Solved*. URL: https://edgegap.com/blog-en/explainer-series-authoritative-servers-relays-peer-to-peer-understanding-networking-types-and-their-benefits-for-each-game-types (дата звернення: 08.05.2024).
4. Game Networking Demystified, Part I: State vs. Input. *Ruoyu Sun*. URL: https://ruoyusun.com/2019/03/28/game-networking-1.html (дата звернення: 08.05.2024).
5. Game Networking Demystified, Part IV: Server and Network Topology. *Ruoyu Sun*. URL: https://ruoyusun.com/2019/04/07/game-networking-4.html (дата звернення: 08.05.2024).
6. Godot Engine - Free and open source 2D and 3D game engine. *Godot Engine*. URL: https://godotengine.org/ (дата звернення: 08.05.2024).
7. Growth in time spent gaming during COVID-19 worldwide 2020 | Statista. *Statista*. URL: https://www.statista.com/statistics/1188545/gaming-time-spent-covid/ (дата звернення: 08.05.2024).
8. Hosch W. L. Electronic strategy game | History & Examples. *Encyclopedia Britannica*. URL: https://www.britannica.com/topic/electronic-strategy-game (дата звернення: 08.05.2024).

9. NAT Punch-through for Multiplayer Games. *Keith Johnston*. URL: https://keithjohnston.wordpress.com/2014/02/17/nat-punch-through-for-multiplayer-games/ (дата звернення: 08.05.2024).
10. Jagdale D. Finite State Machine in Game Development. *International Journal of Advanced Research in Science, Communication and Technology*. 2021. С. 384–390. URL: https://doi.org/10.48175/ijarsct-2062 (дата звернення: 08.05.2024).
11. Design and Implementation of an Intelligent Gaming Agent Using A* Algorithm and Finite State Machines / A. A. Adegun et al. *International Journal of Engineering Research and Technology*. 2020. Vol. 13, no. 2. P. 191. URL: https://doi.org/10.37624/ijert/13.2.2020.191-206 (дата звернення: 08.05.2024).
12. Quing Wei Lim. How do multiplayer games sync their state? Part 1. *Medium*. URL: https://medium.com/@qingweilim/how-do-multiplayer-games-sync-their-state-part-1-ab72d6a54043 (дата звернення: 08.05.2023).
13. Video Game Mechanics: A Beginner's Guide (with Examples). *Game Design Skills*. URL: https://gamedesignskills.com/game-design/video-game-mechanics/ (дата звернення: 08.05.2024).
14. Steam charts. *SteamDB*. URL: https://steamdb.info/app/753/charts/ (дата звернення: 08.05.2024).
15. The Beginner's Guide To Strategy Game Subgenres. *Cultured Vultures*. URL: https://culturedvultures.com/strategy-game-subgenres/ (дата звернення: 08.05.2024).
16. The Role of Authoritative Dedicated Servers in Live Game Development. *AccelByte*. URL: https://accelbyte.io/blog/the-role-of-authoritative-dedicated-servers-in-live-game-development (дата звернення: 08.05.2024).

17. What Every Programmer Needs To Know About Game Networking | Gaffer On Games. *Gaffer On Games*. URL: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/ (дата звернення: 08.05.2024).
18. Electronic Arts. Frostbite Engine - The most adopted platform for game development - EA. *Electronic Arts Inc.* URL: https://www.ea.com/frostbite (дата звернення: 08.05.2024).
19. Dhule M. Exploring Game Mechanics. Berkeley, CA : Apress, 2022. URL: https://doi.org/10.1007/978-1-4842-8873-3 (дата звернення: 08.05.2024).
20. Abarbanel B., Johnson M. R. Gambling engagement mechanisms in Twitch live streaming. *International Gambling Studies*. 2020. Т. 20, № 3. С. 393–413. URL: https://doi.org/10.1080/14459795.2020.1766097 (дата звернення: 08.05.2024).
21. Overview of Godot's key concepts. *Godot Engine documentation*. URL: https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html (дата звернення: 08.05.2024).
22. Samuelson G. Pixel art - The Medium of Limitation: A qualitative study on how experienced artists perceive the relationship between restrictions and creativity. *Informatik Student Paper Bachelor (INFSPB)*. 2020. С. 28. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2 (дата звернення: 08.05.2024).
23. Tiles and tilemaps overview - Game development | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Games/Techniques/Tilemaps (дата звернення: 12.05.2024).

ДОДАТОК

1. Клас TileResource.gd

```
extends Resource
class_name TileResource

var name: StringName;

var atlas_coord: Vector2i;
var path: String;
var desc: String;

func _init(_name: StringName = "", _atlas_coord: Vector2i = Vector2i.ZERO,
                        _path: String = "", _desc: String = "") -> void:
        name = _name;
        atlas_coord = _atlas_coord;
        path = _path;
        desc = _desc;
```

2. Клас TileItem.gd

```
extends Resource
class_name TileItem;

signal cost_changed(int);

@export var position: Vector2i = Vector2i(0, 0);
@export var data: TileResource = null;

@export var type: StringName = "";

@export var cost_increase: int = 0;
@export var cost: int = 0:
        set(value):
                cost = value;
                cost_changed.emit(cost);

func _init(_type: StringName = "", _cost: int = 0, _data: TileResource = null):
        type = _type;
        cost = _cost;
        data = _data;

func can_set_on_tile(_tile: TileInfo) -> bool:
        return false;
```

```
func apply(_tile: TileInfo) -> void:
	pass;
```

3. Клас TIBuilding.gd

```
extends TileItem
class_name TIBuilding;

@export var points: int;

func _init(_type: StringName = "", _cost: int = 0,
				_points = 0, _data: TileResource = null):
	super(_type, _cost, _data);
	points = _points;

func can_set_on_tile(tile: TileInfo) -> bool:
	return !tile.building;

func apply(tile: TileInfo) -> void:
	position = tile.position;
	tile.building = self;
	GameState.points += points;

func remove() -> void:
	var tile: TileInfo = GameState.game.map.get_tile(position);
	tile.building = null;
	GameState.points -= points / 2;
```

4. Клас TIHouse.gd

```
extends TIBuilding
class_name TIHouse;

@export var food_cost: int;

@export var workers: int;

func _init(_type: StringName = "", _price: int = 0,
				_food_cost: int = 0, _workers: int = 0,
				_points = 0, _data: TileResource = null):
	super(_type, _price, _points, _data);

	food_cost = _food_cost;
```

```
	workers = _workers;

func can_set_on_tile(tile: TileInfo) -> bool:
	return super.can_set_on_tile(tile) && GameState.food >= food_cost;

func apply(tile: TileInfo) -> void:
	super.apply(tile);
	GameState.workers += workers;
	GameState.food -= food_cost;

func remove() -> void:
	super.remove();
	GameState.food += food_cost;
	GameState.workers -= workers;
```

5. TIClinic.gd

```
extends TIBuilding
class_name TIClinic;

@export var maintenance_cost: int;
@export var capacity: int;

@export var is_active: bool;

func _init(_type: StringName = "", _price: int = 0,
                        _maintenance_cost: int = 0, _capacity: int = 0,
                        _points = 0, _data: TileResource = null):
	super(_type, _price, _points, _data);

	maintenance_cost = _maintenance_cost;
	capacity = _capacity;

func apply(tile: TileInfo) -> void:
	super.apply(tile);
	activate();

func remove() -> void:
	super.remove();
	deactivate();

func activate() -> void:
	if is_active: return;
```

```
        is_active = true;
        GameState.clinics.append(self);
        GameState.clinics_capacity += capacity;

func deactivate() -> void:
        if !is_active: return;
        is_active = false;
        GameState.clinics.erase(self);
        GameState.clinics_capacity -= capacity;
```

6. Клас TIActivableBuilding.gd

```
extends TIBuilding
class_name TIActivableBuilding;

@export var workers_cost: int;
@export var is_active: bool = false;
@export var injury_risk: float = 0;

func _init(_type: StringName = "",
                        _price: int = 0, _workers_cost: int = 0,
                        _injury_risk: float = 0,
                        _points = 0, _data: TileResource = null):
        super(_type, _price, _points, _data);

        workers_cost = _workers_cost;
        injury_risk = _injury_risk;

func can_set_on_tile(tile: TileInfo) -> bool:
        return super.can_set_on_tile(tile) && GameState.workers >= workers_cost;

func apply(tile: TileInfo) -> void:
        super.apply(tile);
        GameState.producers.append(self);
        activate();

func remove() -> void:
        super.remove();
        GameState.producers.erase(self);
        deactivate();

func activate() -> void:
        if is_active: return;
        if GameState.workers >= workers_cost:
```

```
            is_active = true;
            GameState.workers -= workers_cost;

func produce() -> void:
    if randf() < injury_risk:
            deactivate();
            GameState.workers -= 1;
            GameState.sick_workers += 1;

func deactivate() -> void:
    if !is_active: return;
    is_active = false;
    GameState.workers += workers_cost;
```

7. Клас PollutionComponent.gd

```
extends Resource
class_name PollutionComponent;

@export var radius: int;
@export var general_pollution: int;
@export var surrounding_pollution: int;

func pollute(position: Vector2i) -> void:
    var tiles: Array[Vector2i] = GameState.game.map.get_tiles_around(position, radius);
    var tile: TileInfo;
    for coord in tiles:
            tile = GameState.game.map.get_tile(coord);
            tile.pollution_level = clamp(tile.pollution_level + surrounding_pollution, 0, 100);
```

8. Клас TIAntiAirWeapon.gd

```
extends TIActivableBuilding
class_name TIAntiAirWeapon;

@export var radius: int;
@export var uses_per_turn: int;

@export var times_used: int = 0;

func _init(_type: StringName = "", _price: int = 0,
                    _workers_cost: int = 0, _injury_risk: float = 0,
                    _radius: int = 0, _uses_per_turn: int = 0,
                    _points = 0, _data: TileResource = null):
```

```
	super(_type, _price, _workers_cost, _injury_risk, _points, _data);

	radius = _radius;
	uses_per_turn = _uses_per_turn;
	cost_increase = 30;

func reload() -> void:
	times_used = 0;

func protect() -> bool:
	if uses_per_turn <= times_used:
		return false;
	times_used += 1;
	return true;

func apply(tile: TileInfo) -> void:
	super.apply(tile);
	GameState.aaws.append(self);

func remove() -> void:
	super.remove();
	GameState.aaws.erase(self);

func produce() -> void:
	pass;
```

9. Клас TIFactory.gd

```
extends TIActivableBuilding
class_name TIFactory;

@export var money_production: int;
@export var pollution_component: PollutionComponent;

func _init(_type: StringName = "", _price: int = 0,
			_workers_cost: int = 0, _injury_risk: float = 0,
			_radius: int = 0, _general_pollution: int = 0,
			_surrounding_pollution: int = 0, _money_production: int = 0,
			_points = 0, _data: TileResource = null):
	super(_type, _price, _workers_cost, _injury_risk, _points, _data);

	pollution_component = PollutionComponent.new();
	pollution_component.radius = _radius;
	pollution_component.general_pollution = _general_pollution;
```

```
        pollution_component.surrounding_pollution = _surrounding_pollution;
        money_production = _money_production;
        cost_increase = 30;

func apply(tile: TileInfo) -> void:
        super.apply(tile);
        GameState.pollutions.append(self);

func remove() -> void:
        super.remove();
        GameState.pollutions.erase(self);

func produce() -> void:
        super.produce();
        GameState.money += money_production;

func pollute() -> void:
        pollution_component.pollute(position);
```

10.Клас TILaboratory.gd

```
extends TIActivableBuilding
class_name TILaboratory;

@export var points_production: int;

func _init(_type: StringName = "",
                        _price: int = 0, _workers_cost: int = 0,
                        _injury_risk: float = 0, _points_production: int = 0,
                        _points = 0, _data: TileResource = null):
        super(_type, _price, _workers_cost, _injury_risk, _points, _data);

        points_production = _points_production;
        cost_increase = 30;

func produce() -> void:
        super.produce();
        GameState.points += points_production;
```

11.Клас TIPark.gd

```
extends TIActivableBuilding
class_name TIPark;
```

```
@export var pollution_component: PollutionComponent;

func _init(_type: StringName = "",
                        _price: int = 0, _workers_cost: int = 0,
                        _injury_risk: float = 0,  _radius: int = 0,
                        _general_pollution: int = 0, _surrounding_pollution: int = 0,
                        _points = 0, _data: TileResource = null):
        super(_type, _price, _workers_cost, _injury_risk, _points, _data);

        pollution_component = PollutionComponent.new();
        pollution_component.radius = _radius;
        pollution_component.general_pollution = _general_pollution;
        pollution_component.surrounding_pollution = _surrounding_pollution;
        cost_increase = 15;

func apply(tile: TileInfo) -> void:
        super.apply(tile);
        GameState.pollutions.append(self);

func remove() -> void:
        super.remove();
        GameState.pollutions.erase(self);

func pollute() -> void:
        pollution_component.pollute(position);
```

12.Клас TIStore.gd

```
extends TIActivableBuilding
class_name TIStore;

@export var food: int;

func _init(_type: StringName = "", _price: int = 0,
                        _workers_cost: int = 0, _injury_risk: float = 0, _food: int = 0,
                        _points = 0, _data: TileResource = null):
        super(_type, _price, _workers_cost, _injury_risk, _points, _data);

        food = _food;

func activate() -> void:
        if is_active: return;
        if GameState.workers >= workers_cost:
                is_active = true;
```

```
                GameState.workers -= workers_cost;
                GameState.food += food;

func deactivate() -> void:
        if !is_active: return;
        if GameState.food - food >= 0:
                is_active = false;
                GameState.workers += workers_cost;
                GameState.food -= food;
```

13.Клас ACleaner.gd

```
extends TIAction
class_name ACleaner;

@export var clean_points: int;

func _init(_type: StringName = "", _cost: int = 0, _clean_points: int = 0, _data: TileResource =
null):
        super(_type, _cost, _data);
        clean_points = _clean_points;
        cost_increase = 10;

func can_set_on_tile(_tile: TileInfo) -> bool:
        return true;

func apply(tile: TileInfo) -> void:
        tile.pollution_level = max(tile.pollution_level - clean_points, 0);
```

14.Клас ASapper.gd

```
extends TIAction
class_name ASapper;

func _init(_type: StringName = "",
                        _cost: int = 0, _data: TileResource = null):
        super(_type, _cost, _data);
        cost_increase = 2;

func can_set_on_tile(tile: TileInfo) -> bool:
        return tile.cell == GData.Cell.RUINS;

func apply(tile: TileInfo) -> void:
        tile.cell = GData.Cell.FREE;
```

15.Клас AWarAction.gd

```
extends TIAction
class_name AWarAction;

@export var radius: int = 0;

func _init(_type: StringName = "", _radius: int = 0,
                        _cost: int = 0, _data: TileResource = null):
        super(_type, _cost, _data);
        radius = _radius;
```

16.Клас TileInfo.gd

```
extends Resource
class_name TileInfo;

@export var position: Vector2i;

@export var cell: GData.Cell = GData.Cell.RUINS;
@export var building: TIBuilding;

@export var is_mined: bool = false;
@export var pollution_level = 0;

func _init(
        _cell: GData.Cell = GData.Cell.RUINS,
        _building: TIBuilding = null
):
        cell = _cell;
        building = _building;
```

17.Клас GameStateData.gd

```
extends Node
class_name GameStateData;

signal workers_changed(int);
signal money_changed(int);
signal food_changed(int);
signal turn_changed(int);
signal points_changed(int);
signal pollution_changed();

signal game_won;
```

```
signal game_lost;

var win_conditions = {
	turns = 10,
	points = 100
};

var game: Game;

var turn: int = 1:
	set(value):
		turn = value;
		turn_changed.emit(value);
		if multiplayer.is_server():
			update_turn_on_peers.rpc(value);

var workers: int:
	set(value):
		workers = value;
		if value < 0:
			for producer in producers:
				producer.deactivate();
				if workers >= 0:
					break;
		workers_changed.emit(workers);
var money: int:
	set(value):
		money = value;
		money_changed.emit(value);

var war_money: int:
	set(value):
		war_money = value;
		update_money_on_peers.rpc(value);

var food: int:
	set(value):
		food = value;
		food_changed.emit(value);
var points: int:
	set(value):
		points = value;
		points_changed.emit(value);
		if multiplayer.is_server():
```

```
                update_points_on_peers.rpc(value);

var producers: Array[TIActivableBuilding];
var pollutions: Array[TIActivableBuilding];

var clinics: Array[TIClinic];
var sick_workers: int;
var clinics_capacity: int;

var aaws: Array[TIAntiAirWeapon];

@rpc("authority", "call_remote")
func update_points_on_peers(value: int) -> void:
	points = value;

@rpc("authority", "call_remote")
func update_turn_on_peers(value: int) -> void:
	turn = value;

@rpc("authority", "call_remote")
func update_money_on_peers(value: int) -> void:
	money += value;

@rpc("authority", "call_remote")
func update_game_result(win: bool) -> void:
	if win:
		game_won.emit();
	else:
		game_lost.emit();

func init_server_game(sgame: Game) -> void:
	game = sgame;
	var server_hud = load("res://scenes/hud/hud.tscn");
	game.hud = server_hud.instantiate();

func init_client_game(sgame: Game) -> void:
	game = sgame;
	var client_hud = load("res://scenes/hud/war_hud.tscn");
	game.hud = client_hud.instantiate();

func process_turn() -> void:
	var pollution_level: int = 0;
	if !pollutions.is_empty():
		var general_pollution: int = 0;
```

```
                for pbuilding in pollutions:
                        if pbuilding.is_active && pbuilding.pollution_component:
                                pbuilding.pollute();
                                general_pollution                                       +=
pbuilding.pollution_component.general_pollution;

                for i in game.map.tiles.size():
                        game.map.tiles[i].pollution_level += general_pollution;
                        pollution_level += game.map.tiles[i].pollution_level;
                pollution_changed.emit();

        points += GData.pollution_points(pollution_level / game.map.tiles.size());

        for prod in producers:
                if prod.is_active:
                        prod.produce();

        if sick_workers > 0:
                workers += clampi(sick_workers, 0, clinics_capacity);
                sick_workers = 0;

        for clinic in clinics:
                if money < clinic.maintenance_cost:
                        clinic.deactivate();
                else:
                        money -= clinic.maintenance_cost;

        if win_conditions.turns == turn:
                update_game_result(points >= win_conditions.points);
                update_game_result.rpc(points < win_conditions.points);
                return;

        for aaw in aaws:
                aaw.reload();

        turn += 1;


func reset() -> void:
        turn = 1;
        points = 0;

        workers = 0;
        money = 100;
```

```
	food = 1;

	producers.clear();
	pollutions.clear();
	clinics.clear();

	sick_workers = 0;
	clinics_capacity = 0;
```

18. Клас GServer.gd

```
extends Node
class_name GServer;

signal bomb_info_update(Vector2i, int);

var players = {

}

@rpc("any_peer", "call_local")
func add_player(id: int) -> void:
	players[id] = {};

@rpc("authority", "call_local")
func remove_player(id: int) -> void:
	players.erase(id);

func check_aaw(coords: Vector2i) -> bool:
	for aaw in GameState.aaws:
		if GameState.game.map.get_distance(coords, aaw.position) <= aaw.radius &&
aaw.protect():
			return true;
	return false;

func bomb_tile(coords: Vector2i) -> int:
	var profit: int = 0;
	var tile = GameState.game.map.get_tile(coords);
	if tile.building:
		profit = tile.building.cost;
		tile.building.remove();
		if tile.building is TIActivableBuilding:
			GameState.workers -= tile.building.workers_cost;
		tile.building = null;
```

```
	tile.cell = GData.Cell.RUINS;
	tile.is_mined = randf() < GameState.game.map.mines_frequency;
	GameState.game.map.draw_explosion(GameState.game.map.map_to_local(coords));
	GameState.game.map.set_tile(coords, tile);
	return profit;

@rpc("any_peer", "call_remote")
func attack_bomb(coords: Vector2i) -> void:
	if GameState.war_money < GData.ACTIONS.get(&"bomb").cost:
		return;
	GameState.war_money -= GData.ACTIONS.get(&"bomb").cost

	var results: Dictionary = {"pos": Vector2i(0, 0), "buildings_around": 0, "destroyed": false};
	results.pos = coords;

	if !check_aaw(coords):
		GameState.war_money += bomb_tile(coords);

		var surroundings: Array[Vector2i] = GameState.game.map.get_tiles_around(coords, 1);
		for sur in surroundings:
			if GameState.game.map.get_tile(sur).building && sur != coords:
				results.buildings_around += 1;
	else:
		results.destroyed = true;

	Server.bomb_result.rpc_id(multiplayer.get_remote_sender_id(), results);

@rpc("any_peer", "call_remote")
func airstrike_attack(coords: Vector2i) -> void:
	if GameState.war_money < GData.ACTIONS.get(&"airstrike").cost:
		return;
	GameState.war_money -=GData.ACTIONS.get(&"airstrike").cost

	var results: Dictionary = {};
	var surroundings: Array[Vector2i] = GameState.game.map.get_tiles_around(coords, 1);
	surroundings.append(coords);
	for sur in surroundings:
		if !check_aaw(sur):
			GameState.war_money += bomb_tile(sur);
			results[sur] = true;
		else:
			results[sur] = false;
```

```
	Server.airstrike_result.rpc_id(multiplayer.get_remote_sender_id(), results);

@rpc("any_peer", "call_remote")
func dirtybomb_attack(coords: Vector2i) -> void:
	if GameState.war_money < GData.ACTIONS.get(&"dirtybomb").cost:
		return;

	GameState.war_money -= GData.ACTIONS.get(&"dirtybomb").cost

	var results: Dictionary = {"pos": Vector2i(0, 0), "destroyed": false};
	results.pos = coords;

	if !check_aaw(coords):
		var surroundings: Array[Vector2i] = GameState.game.map.get_tiles_around(coords, 1);
		for sur in surroundings:
			var tile = GameState.game.map.get_tile(sur);
			tile.pollution_level += 30;

	GameState.game.map.draw_explosion(GameState.game.map.map_to_local(sur), Color.GREEN);
	else:
		results.destroyed = true;

	Server.dirtybomb_result.rpc_id(multiplayer.get_remote_sender_id(), results);

@rpc("any_peer", "call_remote")
func activate_radar(coords: Vector2i) -> void:
	if GameState.war_money < GData.ACTIONS.get(&"radar").cost:
		return;
	GameState.war_money -=GData.ACTIONS.get(&"radar").cost
	var results: Dictionary = {};
	var surroundings: Array[Vector2i] = GameState.game.map.get_tiles_around(coords, 1);
	surroundings.append(coords);
	for sur in surroundings:
		var tile = GameState.game.map.get_tile(sur);
		if tile.building:
			results[sur] = tile.building.type;

	Server.radar_result.rpc_id(multiplayer.get_remote_sender_id(), results);

@rpc("authority", "call_remote")
func bomb_result(res: Dictionary) -> void:
	if res.destroyed:
```

```
		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(res.pos),
Color.DARK_KHAKI);
		else:
			var surroundings: Array[Vector2i] =
GameState.game.map.get_tiles_around(res.pos, 1);
			for sur in surroundings:
				bomb_info_update.emit(sur, -1);
			GameState.game.map.set_tile(res.pos, TileInfo.new());

		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(res.pos));
			bomb_info_update.emit(res.pos, res.buildings_around);

@rpc("authority", "call_remote")
func airstrike_result(res: Dictionary) -> void:
		for key in res.keys():
			if res.get(key):
				var tile: TileInfo = TileInfo.new();
				GameState.game.map.set_tile(key, tile);

		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(key));
			else:

		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(key),
Color.DARK_KHAKI);

@rpc("authority", "call_remote")
func dirtybomb_result(res: Dictionary) -> void:
		if res.destroyed:

		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(res.pos),
Color.DARK_KHAKI);
		else:
			var surroundings: Array[Vector2i] =
GameState.game.map.get_tiles_around(res.pos, 1);
			surroundings.append(res.pos);
			for sur in surroundings:

		GameState.game.map.draw_explosion(GameState.game.map.map_to_local(sur),
Color.GREEN);

@rpc("authority", "call_remote")
func radar_result(res: Dictionary) -> void:
		for key in res.keys():
```

```
var tile = GameState.game.map.get_tile(key);
tile.building = GData.BUILDINGS.get(res[key]);
GameState.game.map.set_tile(key, tile);
```