МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

ЗДРОК ДМИТРО ОЛЕКСАНДРОВИЧ

РОЗРОБКА ТА ДОСЛІДЖЕННЯ АЛГОРИТМІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО СВІТУ ТА РЕАЛІЗАЦІЯ МЕХАНІЗМІВ ВЗАЄМОДІЇ МІЖ ГРАВЦЯМИ.

Спеціальність: 122 Комп'ютерні науки Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології Робота на здобуття освітнього ступеня "магістр"

Науковий керівник:
СОБЧУК ВАЛЕНТИН ВОЛОДИМИРОВИЧ,
доктор технічних наук, професор

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних наук
та кібербезпеки
від _____ 20___ р.
Завідувач кафедри
(_____)_____
(підпис) ПІБ

ЛУЦЬК − 2025

ЗМІСТ

ВСТУП

Актуальність теми. Актуальність теми цієї роботи зумовлена сучасними тенденціями розвитку ігрової індустрії, де все більше проєктів орієнтуються на створення масштабних, варіативних та багатокористувацьких середовищ. Процедурна генерація світів виступає ключовим інструментом у цьому процесі, оскільки дозволяє автоматично створювати великі й різноманітні ігрові простори без надмірних витрат ресурсів та часу на ручне проєктування. Це забезпечує підвищену реіграбельність, динамічність і непередбачуваність ігрового досвіду, що особливо важливо для жанрів виживання, стратегій та відкритих світів. Додатковим фактором актуальності є зростання попиту на багатокористувацькі ігри серед різних категорій користувачів, що стимулює розробників до пошуку інноваційних підходів для оптимізації мережевої взаємодії, підвищення продуктивності та безпеки. Таким чином, дослідження алгоритмів процедурної генерації у поєднанні з методами побудови багатокористувацьких компонентів не лише відповідає сучасним потребам індустрії, але й має значний практичний потенціал для впровадження у реальні ігрові проєкти.

Мета роботи – дослідити та розробити систему процедурної генерації ігрового контенту (біоми світу, ресурси) та методи взаємодії між гравцями на прикладі багатокористувацької гри.

Об’єкт дослідження: процес створення та наповнення ігрових світів; реалізація багатокористувацього режиму гри на ігровому рушії Unity.

Предмет дослідження: алгоритми та програмні рішення для процедурної генерації ігрових середовищ та способи реалізації багатокористувацьх ігор.

Апробація результатів роботи: тези було опубліковано у збірнику II Міжнародної науково-практичної конференції “Проблеми комп’ютерних наук, програмного моделювання та безпеки цифрових систем” (9 – 11 червня 2025 р).

РОЗДІЛ 1
ОГЛЯД ТА АНАЛІЗ АЛГОРИТМІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ СВІТІВ ТА ІНСТРУМЕНТІВ БАГАТОКОРИСТУВАЦЬКОЇ КОМПОНЕНТИ У ВІДЕОІГРАХ

1.1 Що таке процедурна генерація світів

Процедурна генерація – це спосіб автоматичного створення ігрового контенту, зокрема віртуальних світів, за допомогою алгоритмів, а не лише ручної роботи художників чи дизайнерів. Такий підхід ґрунтується на математичних функціях, певних правилах і випадкових (псевдовипадкових) генераторах. Використовуючи початкове число (seed), алгоритм формує структуровані й відтворювані світи. На практиці це означає, що замість того, щоб зберігати кожен біт даних про світ, гра зберігає лише seed і сам алгоритм. Завдяки цьому можна створювати великі й деталізовані світи, не витрачаючи багато пам’яті, а кожен новий seed дарує гравцеві унікальний досвід і підвищує реіграбельність.

Ідеї процедурної генерації мають довгу історію у відеоіграх. Ранні приклади включають Rogue із процедурно згенерованими підземеллями та Elite, де алгоритм створював зоряні системи. Ці ігри показали, що навіть прості алгоритми можуть породжувати велику різноманітність ігрового простору. Сучасні підходи розвиваються завдяки новим математичним методам, потужнішим обчислювальним можливостям і гібридним технікам, які комбінують шум, граматики, симуляції та модулі для ще більш цікавих і реалістичних світів. [1]

1.2 Види процедурної генерації світів: їх особливості та недоліки

1.2.1 Генерація на основі шуму (Noise-based generation)

В основі алгоритму лежить використання функцій шуму, які генерують псевдовипадкові значення з плавною зміною в просторі. На відміну від справжньої випадковості, шум має когерентність – близькі точки мають схожі значення, що створює природні градієнти та органічні форми. (Рис. 1.1) [2]

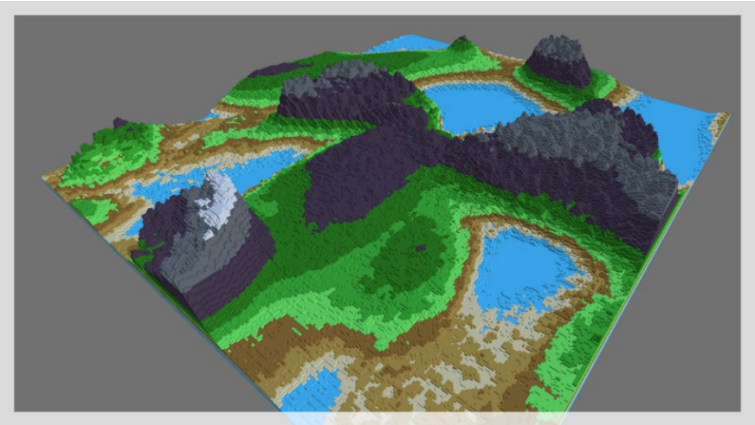

Рис. 1.1 – Приклад генерації світу на основі шуму

Застосування алгоритму генерації на основі шуму:

- генерація висот ландшафту та рельєфу місцевості;
- створення природних текстур (хмари, вода, кора дерев);
- розподіл ресурсів та біомів у відкритих світах;
- моделювання природних явищ (туман, пилові бурі).

Для моделювання природних структур і генерації плавних, неперіодичних текстур у комп’ютерній графіці та цифровій обробці сигналів широко застосовуються різні види процедурного шуму. Такі алгоритми дозволяють отримувати складні візуальні та числові патерни на основі відносно простих математичних правил. Найбільш поширеними та практично значущими є шум Перліна, фрактальний шум та шум Ворлі.

Процес генерації шуму Перліна (1983) складається з наведених нижче етапів.

1. Створення градієнтної решітки. Розміщення векторів градієнтів у вузлах регулярної сітки.
2. Обчислення відстаней. Для кожної точки визначаються відстані до чотирьох найближчих вузлів.
3. Скалярний добуток. Обчислення скалярного добутку між градієнтами та векторами відстаней.
4. Інтерполяція. Плавне змішування значень за допомогою кубічної або квінтичної функції.
5. Нормалізація. Приведення результату до діапазону [-1, 1].

Фрактальний шум (FBM) формується шляхом накладання кількох шарів базового шуму та включає наведені нижче кроки.

1. Визначення базових параметрів. Встановлення частоти, амплітуди та кількості октав.
2. Генерація першої октави. Створення базового шару шуму з найнижчою частотою.
3. Додавання октав. Накладання додаткових шарів з подвоєною частотою та зменшеною амплітудою.
4. Нормалізація результату. Масштабування фінального значення до потрібного діапазону.

Алгоритм генерації шуму Ворлі (1996) передбачає виконання таких дій.

1. Розміщення зародкових точок. Випадкове розставляння контрольних точок у просторі.
2. Обчислення відстаней. Для кожної точки знаходження відстані до всіх зародків.
3. Вибір мінімальної відстані. Використання найкоротшої відстані як основного значення.
4. Застосування функції. Перетворення відстані за допомогою математичних операцій.

У таблиці 1.1 наведені переваги та недоліки використання алгоритму генерації на основі шуму.

Таблиця 1.1

Переваги та недоліки генерації на основі шуму

Переваги	Недоліки
Створює плавні, природні та органічні ландшафти	Чистий шум може бути одноманітним або занадто хаотичним
Ефективний та швидкий, підходить для генерації великих відкритих світів	Не підходить для створення чітких, штучних структур без додаткових алгоритмів
Легко масштабується та комбінується з іншими алгоритмами	Складно контролювати специфічні особливості ландшафту
Дозволяє точний контроль деталізації через систему октав	Може створювати непрактичні для геймплею зони

1.2.2 Генерація на основі граматик (Grammar-based generation)

Граматичні системи використовують набір правил переписування, які поступово трансформують початковий символ у складну структуру. Процес нагадує ріст живих організмів або розвиток мовних конструкцій.

Застосування алгоритму генерації на основі граматик:

- створення архітектурних споруд з логічною структурою;
- генерація органічних форм (дерева, рослини, корали);
- процедурне створення міст з урахуванням планувальних принципів;
- генерація складних лабіринтів та підземель.

Залежно від способу опису та інтерпретації правил розрізняють кілька основних типів граматичних систем, послідовність дій яких наведено нижче.

L-системи (Lindenmayer Systems) є формальним граматичним підходом до процедурної генерації структур, алгоритм роботи яких включає такі етапи.

1. Визначення аксіоми. Встановлення початкового символу або послідовності.

2. Створення правил заміщення. Визначення як кожен символ замінюється новою послідовністю.
3. Ітеративне застосування. Послідовне перетворення рядка згідно з правилами заданої кількості разів.
4. Інтерпретація символів. Перетворення фінального рядка в геометричні об’єкти.
5. Застосування параметрів. Додавання змінних для контролю розмірів та кутів.

Контекстно-вільні граматики застосовуються для побудови складних структур шляхом визначення правил заміщення та рекурсивного розгортання; процес їх використання реалізується у вигляді таких кроків.

1. Створення ієрархії правил. Визначення як складні структури складаються з простіших.
2. Рекурсивне розгортання. Заміщення нетерміналів відповідно до правил.
3. Контроль глибини. Обмеження рекурсії для запобігання нескінченному росту.
4. Застосування обмежень. Врахування просторових та логічних обмежень.
5. Фінальна інтерпретація. Перетворення граматичного дерева в 3D структури.

Граф-граматики застосовуються для процедурної генерації структур на основі графів, де зміни топології та геометрії визначаються правилами заміщення. Процес їх використання реалізується у вигляді таких кроків.

1. Ініціалізація базового графу. Створення початкової топологічної структури.
2. Застосування правил заміщення. Заміна підграфів згідно з визначеними шаблонами.
3. Контроль топологічної коректності. Забезпечення збереження правильних з’єднань.
4. Додавання геометричних атрибутів. Присвоєння просторових властивостей вузлам.
5. Генерація фінальної геометрії. Створення 3D моделей на основі графу.

У таблиці 1.2 наведені переваги та недоліки використання алгоритму генерації на основі граматик.

Таблиця 1.2

Переваги та недоліки генерації на основі граматик

Переваги	Недоліки
Дозволяє створювати складні, ієрархічні та логічні структури	Розробка правил може бути складною та трудомісткою
Ідеально підходить для архітектури та органічних форм	Результати можуть виглядати штучно при недостатній різноманітності правил
Забезпечує високий рівень контролю над результатом	Важко збалансувати між складністю та передбачуваністю
Може створювати самоподібні фрактальні структури	Потребує глибокого розуміння предметної області

1.2.3 Генерація на основі симуляцій (Simulation-based generation)

Симуляційна генерація моделює фізичні та природні процеси протягом тривалого часу, дозволяючи системі еволюціонувати до реалістичного стану. Замість миттєвого створення світу, алгоритм “проживає” його історію формування.

Застосування:

- створення реалістичних континентів з логічним розташуванням гір та водойм;
- моделювання кліматичних зон та погодних патернів;
- генерація природних екосистем з правдоподібним розподілом видів;
- симуляція історичного розвитку цивілізацій та міст.

Кожен тип симуляції характеризується власним набором правил та послідовністю дій, що забезпечують поступове формування реалістичних

моделей. Нижче наведено основні типи симуляцій та покроковий опис алгоритмів їхнього виконання.

Тектонічна симуляція.

1. Створення початкових плит. Розбиття території на тектонічні плити з різними властивостями.
2. Визначення напрямків руху. Встановлення векторів руху для кожної плити.
3. Моделювання колізій. Обчислення взаємодії між плитами при зіткненні.
4. Формування рельєфу. Створення гір при конвергенції, долин при дивергенції.
5. Стабілізація системи. Поступове уповільнення процесів до рівноважного стану.

Гідрологічна симуляція.

1. Розподіл початкових опадів. Розміщення води на поверхні згідно з кліматичними моделями.
2. Симуляція стоку. Обчислення напрямків течії води згідно з градієнтами висот.
3. Моделювання ерозії. Видалення ґрунту та породи під дією водних потоків.
4. Транспорт седиментів. Перенесення зруйнованого матеріалу течією.
5. Седиментація. Відкладення матеріалу при зменшенні швидкості течії.

Кліматична симуляція.

1. Розрахунок сонячної інсоляції. Визначення кількості сонячної енергії для кожної ділянки.
2. Моделювання атмосферної циркуляції. Обчислення руху повітряних мас.
3. Симуляція випаровування. Моделювання надходження вологи в атмосферу.
4. Розподіл опадів. Визначення зон випадання дощу та снігу.
5. Формування кліматичних зон: Встановлення стабільних температурних та режимів вологості.

Екологічна симуляція.

1. Розміщення початкових видів. Випадкове або кероване розселення організмів.

2. Моделювання росту популяцій. Симуляція розмноження згідно з ресурсними обмеженнями.
3. Симуляція міграції. Моделювання переміщення видів у пошуках кращих умов.
4. Міжвидова конкуренція. Обчислення взаємодії між різними видами за ресурси.
5. Еволюція екосистем. Довгострокові зміни у видовому складі та структурі.

У таблиці 1.3 наведені переваги та недоліки використання алгоритму генерації на основі симуляції. [3]

Таблиця 1.3

Переваги та недоліки генерації на основі симуляції

Переваги	Недоліки
Забезпечує найвищий рівень реалізму та візуальної достовірності	Надзвичайно обчислювально інтенсивний та повільний процес
Створює логічно взаємопов’язані системи з причинно-наслідковими зв’язками	Складний для налаштування, налагодження та контролю параметрів
Дозволяє моделювати складні природні явища та їх взаємодію	Може створювати непрактичні для ігрового процесу результати
Результати мають наукове обґрунтування та природну логіку	Потребує глибоких знань у відповідних наукових галузях

1.2.4 Генерація на основі мозаїки (Assembly-based generation)

Мозаїчна генерація використовує набір попередньо створених “префабів” або модулів, які алгоритмічно поєднуються за певними правилами. Це схоже на складання пазлу, де кожна частина має визначені точки з’єднання.

Застосування:

- генерація підземель та лабіринтів у RPG та roguelike іграх;

- створення архітектурних структур з логічним плануванням;
- процедурна генерація інтер’єрів будівель та споруд;
- створення дорожніх мереж та міських планувань.

Мозаїчна генерація охоплює кілька підтипів, кожен із яких має власні принципи формування та послідовність дій, наведені нижче.

Тайлова генерація використовується для процедурного створення просторових структур шляхом комбінування окремих елементів (тайлів) за визначеними правилами. Нижче наведено основні етапи цього процесу.

1. Підготовка набору тайлів. Створення колекції квадратних/гексагональних елементів з визначеними типами країв.
2. Визначення правил сумісності. Встановлення які типи країв можуть з’єднуватися між собою.
3. Послідовне заповнення. Проходження по сітці та вибір сумісних тайлів для кожної позиції.
4. Перевірка валідності. Контроль сумісності з уже розміщеними сусідніми тайлами.
5. Застосування ваг. Врахування ймовірностей появи різних типів тайлів.

Модульна генерація застосовується для створення складних просторів шляхом комбінування готових елементів або модулів за визначеними правилами. Нижче наведено основні етапи цього процесу.

1. Створення бази модулів. Підготовка набору готових кімнат, коридорів різних розмірів та функцій.
2. Розміщення основних елементів. Випадкове або керовані правилами розміщення ключових модулів.
3. З’єднання модулів. Знаходження оптимальних шляхів між модулями та розміщення з’єднувальних елементів.
4. Перевірка доступності. Забезпечення можливості дістатися до всіх важливих зон.
5. Оптимізація планування. Покращення компактності та логічності розташування.

Префабрикована генерація використовується для створення складних просторів шляхом розміщення готових великих елементів (префабів) з урахуванням умов середовища та контексту. Основні етапи цього процесу подано нижче.

1. Аналіз біому/контексту. Визначення типу території та умов для розміщення префабів.
2. Вибір сумісних префабів: Фільтрація елементів відповідно до умов середовища.
3. Розміщення за сіткою. Систематичне розміщення на регулярній або нерегулярній сітці.
4. Створення переходів. Генерація з'єднувальних елементів між великими префабами.
5. Деталізація країв. Згладжування меж та адаптація до навколишнього ландшафту.

У таблиці 1.3 наведені переваги та недоліки використання алгоритму генерації на основі симуляції. [3]

Таблиця 1.4

Переваги та недоліки генерації на основі симуляції

Переваги	Недоліки
Забезпечує найвищий контроль над якістю окремих елементів	Сильно обмежується розмаїттям наявних модулів та префабів
Швидкий і передбачуваний процес генерації	Може створювати повторювані візуальні патерни при недостатній різноманітності
Легко інтегрується з ручним дизайном та художньою роботою	Потребує значних зусиль для створення достатньої кількості різноманітних елементів
Дозволяє створювати складні архітектурні структури з правильними пропорціями	Складно забезпечити природні, органічні переходи між модулями

1.3 Гібридні підходи та їх переваги

Сучасні ігри рідко покладаються на один тип процедурної генерації. Натомість розробники комбінують різні підходи, використовуючи сильні сторони кожного методу для створення більш досконалих та різноманітних світів. Такий гібридний підхід дозволяє подолати обмеження окремих алгоритмів та досягти результатів, які були б неможливими при використанні лише одного методу.

Комбінація шуму та граматичних систем є однією з найпоширеніших стратегій. Алгоритми шуму формують основний органічний ландшафт, створюючи природні висоти, долини та водойми, тоді як граматичні системи додають структуровані елементи, такі як будівлі, дерева або штучні споруди. Цей підхід особливо ефективний у іграх, де потрібно поєднати природне середовище з архітектурними елементами. Dwarf Fortress є прикладом такого поєднання, де базовий ландшафт генерується через шум, а складні структури цивілізацій створюються за допомогою правил та граматик.

Поєднання симуляційних методів з мозаїчною генерацією дозволяє створювати світи з науково обґрунтованою макроструктурою та детально продуманими локальними елементами. Симуляція формує великомасштабні природні процеси - континенти, кліматичні зони, річкові системи, тоді як мозаїчні техніки забезпечують детальне та контрольоване наповнення конкретних локацій. Civilization VI демонструє цей підхід, де глобальні геологічні та кліматичні процеси визначають загальну структуру світу, а локальні деталі створюються через систему тайлів та префабів.

Багатошарова генерація представляє найскладніший, але й найпотужніший гібридний підхід. У цій системі різні алгоритми відповідають за окремі аспекти світу, працюючи паралельно або послідовно. Наприклад, шум може визначати базові висоти ландшафту, симуляція - розподіл біомів відповідно до кліматичних умов, граматичні системи - розташування та структуру поселень, а мозаїчні

техніки - деталізацію конкретних будівель та інтер'єрів. Такий підхід забезпечує надзвичайну гнучкість та дозволяє точно контролювати кожен аспект згенерованого світу.

1.4 Розширені приклади застосування

У Minecraft процедурна генерація світу поєднує багатошарові шумові функції для формування базового рельєфу з додатковими правилами й шаблонами для печер, структур та чутливих до біомів елементів. Висотна карта ґрунту будується на "низькочастотному" шумі (Perlin-style noise) для великих форм рельєфу й декількох шарів шуму для дрібнішої деталізації (FBM). Печери та підземні структури часто створюються за допомогою 3D-шуму з пороговими значеннями, а будівлі та села – частково префабричні шаблони, розміщені з урахуванням місцевого рельєфу й біомів. Застосування стрімінгу чанків і seed-підходу робить світ відтворюваним і відносно компактним для збереження та передачі. [4]

No Man's Sky – Hello Games будували гру навколо детерміністичної, seed-керованого всесвіту: комбінація шумів, параметризованих процедурних граматик і наборів формул визначає планети, їхні поверхні, флору і фауну. Система генерує цілісні планети з біомами, погодою, рослинністю і унікальними істотами на основі одного набору генеративних правил і seed, що дозволяє отримувати безмежну кількість варіантів при відтворюваному, хоча й варіативному, дизайні. У No Man's Sky видно сильний акцент на масштабі й одночасно на повторюваних шаблонах – це і джерело величі, і критики за відчуття схожості. [5]

У Dwarf Fortress процедурна генерація поєднує глобальні симуляції (геологія, клімат, історичні симуляції цивілізацій) із детальною локальною генерацією. Історична симуляція ("history generation") створює покоління цивілізацій, війни, міграції та культурні артефакти – через це світ набуває

унікальної “біографії”, яка потім впливає на локальні карти і події. Такий підхід показує, як симуляції можуть додати глибини, але вимагає значних обчислювальних та проектних ресурсів. [6]

Terraria використовує модифіковані одномірні та шаруваті шумові підходи для формування висотного профілю, а також правила й скрипти для генерації печер, з’єднаних камер і біомів у підземеллях. Архітектурні елементи й структури часто поєднують процедурні шари з набором шаблонів, при цьому багато генерацій відбувається “поетапно” (шари рельєфу, потім печери, потім біоми й структури). Це класичний приклад поєднання шуму з модульними/шаблонними техніками для 2D-платформера.

У серії Cities: Skylines процедурні та алгоритмічні техніки застосовуються більше до створення міської інфраструктури й потоків (дороги, зонування, симуляція трафіку), ніж до “ландшафтного” шуму. Алгоритми для створення дорожніх мереж, планування зон та симуляції економічних/транспортних потоків працюють у тандемі, а також є інструменти для імпорту геоданих і автоматичної побудови карт. Це показовий кейс, де процедурні методи інтегровані у складні системи управління ресурсами та симуляції поведінки агентів. [7]

1.5 Переваги процедурної генерації для виживачів-стратегій

Для жанру виживач-стратегія шумові алгоритми особливо корисні: вони дозволяють створювати природні, правдоподібні ландшафти (важливі для дослідження та виживання), забезпечують непередбачуваність і високий рівень реіграбельності (різні розташування ресурсів, стратегічні точки тощо), і масштабованість – можливість створювати великі світи навіть на пристроях з обмеженою продуктивністю. Оскільки шум зазвичай обчислюється доволі швидко, його легко застосувати у великих відкритих світах або в потоковій генерації (streaming chunks).

1.6 Проблеми та обмеження процедурної генерації

Процедурна генерація має кілька ключових проблем: креативна (брак унікальних, запам’ятовуваних локацій та емоційної складової без авторського контролю), технічна (стикування регіонів, балансування великих просторових систем, продуктивність симуляцій і забезпечення детермінізму на різних платформах) і геймдизайнерська (збереження балансу складності та створення природної кривої навчання у світі, що змінюється). Вирішення цих проблем зазвичай вимагає гібридних підходів: поєднання алгоритмічного генератора з ручними “якірними” елементами, додатковими контролями якості та інструментами для тестування й валідації результатів.

1.7 Інструменти для розробки багатокористувацьх ігор

На сьогодні розробка багатокористувацьких ігор майже неможлива без використання спеціалізованих інструментів, що будуть забезпечувати обробку мережевої взаємодії, синхронізацію станів об’єктів та управлінням з’єднання між клієнтами та сервером. Найбільш поширеним таким інструментом в ігровому рушії Unity виступає Mirror Networking. Він виник через відносну старість та неактуальність на той момент існуючого UNet, і зараз використовується для розробки кооперативних ігор або ж невеликих онлайн проєктів. Головною перевагою Mirror виступає його відносна простота у використанні, активна підтримка та розробка спільнотою та доступність через свою безкоштовну модель. Варто зазначити, що ця бібліотека менш придатна для широкомаштабних проектів, де потрібна маштабованість і розподіл ресурсів з використанням хмарних середовищ. [8]

Альтирнативою в цій сфері можна вважати інструмент Photon, який має кілька різних версій, а саме: PUN, Fusion та Quantum. Photon має готову інфраструктуру із зручним налаштуванням, що дає змогу розробникам уникнути

створення власних сервісів. Зараз цей інструмент найчастіше використовується для створення мобільних ігор, де над важливо забезпечувати стабільність та швидкість підключень гравців до сесій. Його головною перевагою є наявність relay-сервісів (проміжні сервер, що дозволяє організовувати з'єднання між клієнтами в багатокористувацьких іграх), системи підбору матчів та можливість маштабування. Головним недоліком Photon є комерційна ліцензія, для великих проєктів необхідна підписка з оплатою, що робить цю модель не надто привабливою для інді-розробників (малі студії або розробники одинаки). Ще одним недоліком є його обмежена гнучкість у разі, коли потрібна специфічна архітектура серверної логіки. [9]

Окремо можна виділити Unity Netcode for GameObjects, що є офіційною бібліотекою Unity. Вона інтегрується з іншими сервісами від компанії, зокрема Unity Relay та Lobby, що робить його перспективним інструментом у майбутньому, проте, наразі, вона не є на стільки популярною серед розробників, через його відносну новизну, і не завжди підходить для великих ігор з високими вимогами для продуктивності. [10]

Ще одним з можливих рішень є FishNet. Ця бібліотека є найбільшою альтирнативою Mirror та робить акцент на продуктивності та оптимізації. FishNet працює краще з великою кількістю клієнтів і надає гнучкі можливості по управлінню мережею. Вона підходить для проєктів, де потрібна стабільність та розширені механізми оптимізації, але має меншу спільноту розробників, ніж Mirror чи Photon, що ускладнює пошук необхідних прикладів та готових рішень. [11]

До інструментів розробки належать не лише лише фреймворки, а й серверні технології та допоміжні сервіси. Використання відокремлених серверів допомагає розділити клієнтську та серверну логіку, що надає кращу безпеку та котроль за ігровим процесом. Хмарні сервіси, такі як AWS GameLift чи Azure PlayFab, надають можливість масштабування та підтримки великої кількості одночасних гравців. Крім того, допоміжні інструменти на кшталт PlayFab чи GameAnalytics забезпечують реалізацію вторинних функцій: збереження данних,

інвентарю, статистки, аналітики і т.д. Усі ці інструменти створюють складне середовище для розробки багатокористувацьких ігор. В залежності від масштабу та складності проекту розробники різним чином комбінують сервіси та фреймворки для створення оптимального рішення. [12, 13]

У таблиці 1.4 наведена порівняльна характеристика інструментів Mirror, Photon та FishNet.

1.8 Огляд наявних інструментів для процедурної генерації

На сьогодні існує значна кількість програмних рішень та інструментів, що надають розробникам можливість створювати складні світи із заданими правилами та рівнем випадковості. Серед них особливу увагу заслуговують такі системи, як World Machine та MapMagic, які активно використовуються у сучасній ігровій індустрії для генерації рельєфів і середовищ.

Таблиця 1.4

Порівняльна характеристика інструментів для розробки багатокористувацьких ігор

Критерій	Mirror	Photon	FishNet
Простота використання	Простий	Складний	Відносно простий
Масштабованість	Підходить для невеликих ігор або кооперативних сесій	Висока масштабованість завдяки	Добре масштабується завдяки оптимізованій обробці пакетів

Вартість	Безкоштовний	Комерційна модель: безкоштовно до певної кількості CCU	Безкоштовний
Продуктивність	Середня	Висока	Висока
Типові сценарії використання	Невеликі проєкти	Масштабні проєкти	Середній масштаб проєкту

1.8.1 World Machine

World Machine – це професійний інструмент для генерації реалістичних тривимірних ландшафтів, який широко застосовується у сфері розробки відеоігор, візуальних ефектів та симуляцій. Його основною особливістю є використання процедурного підходу на основі вузлової системи (node-based workflow), що дозволяє користувачеві створювати складні рельєфи шляхом комбінації різних генераторів, фільтрів та ерозійних процесів (Рис. 1.2).

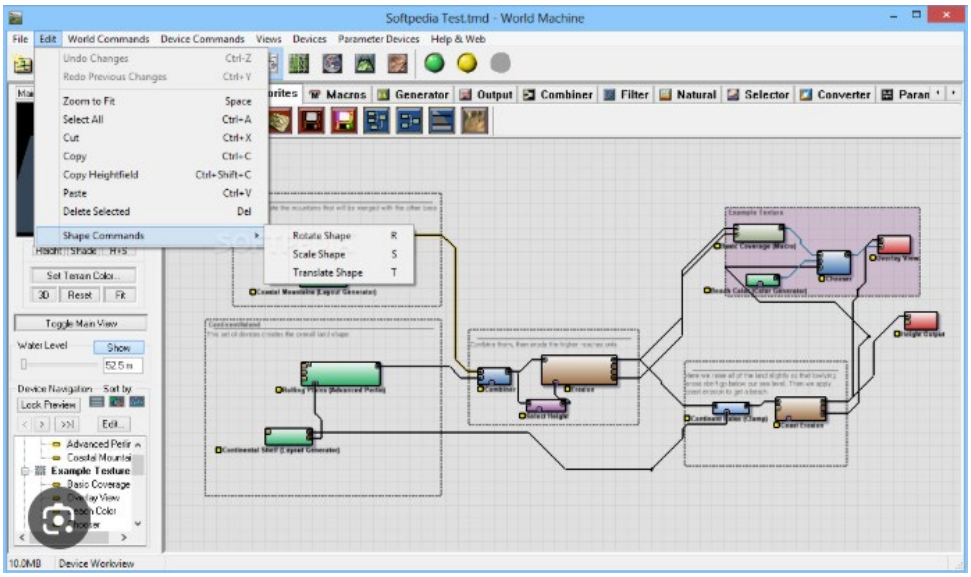

Рис. 1.2 – Використання вузлів для створення правил генерації

World Machine дає змогу симулювати природні геологічні явища такі, як: ерозію, відкладення осадів, потоки води та інші фактори, які формують правдоподібний рельєф. Завдяки цьому розробники отримують можливість створювати детальні й реалістичні карти, що можуть бути експортовані у формати, сумісні з ігровими рушіями, такими як Unity або Unreal Engine. Інструмент також може генерувати як невеликі сцени для прототипів, так і величезні карти з високою роздільною здатністю, що підходять для відкритих світів. Основними перевагами World Machine є гнучкість налаштувань, реалістичність результатів (Рис. 1.3) і можливість точного контролю над усіма етапами генерації. До недоліків можна віднести відносно складний поріг входу для новачків і необхідність подальшої інтеграції з ігровим рушієм для динамічної генерації під час виконання гри. [14]

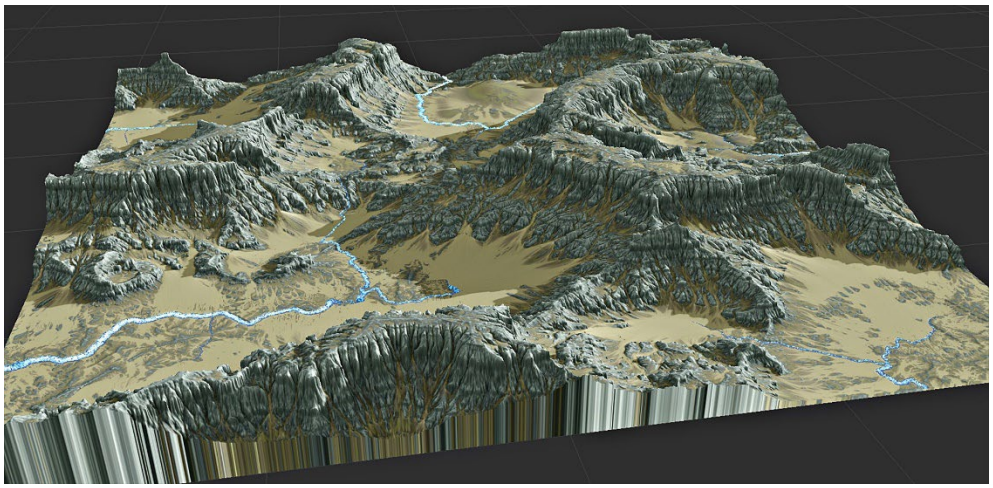

Рис. 1.3 – Приклад згенерованої місцевості за допомогою World Machine

1.8.2 MapMagic World Generator

MapMagic World Generator – це плагін для Unity, який забезпечує процедурну генерацію ландшафтів у реальному часі безпосередньо всередині рушія. На відміну від World Machine, що є окремим додатком, MapMagic інтегрується безпосередньо у робоче середовище Unity, дозволяючи розробникам бачити зміни миттєво й налаштовувати параметри без необхідності експорту чи

імпорту даних. MapMagic базується на тій самій концепції вузлової побудови (node-based system), проте орієнтований на динамічну генерацію ігрових світів, що можуть створюватися або змінюватися під час гри. Це робить його особливо корисним для проєктів із відкритим світом або нескінченними картами, де середовище має генеруватися поступово під час пересування гравця.

Серед основних можливостей MapMagic – генерація висотних карт, текстур, об’єктів (дерев, каменів, будівель), біомів та водних поверхонь. Система підтримує багаторівневу структуру, що дозволяє комбінувати різні генератори та модулі для досягнення складних і природних результатів.

До переваг інструменту належать інтегрованість з Unity, висока швидкість роботи та можливість розширення через власні скрипти на C#. Недоліком можна вважати те, що якість генерації часто поступається більш реалістичним рішенням на зразок World Machine, особливо при створенні великих статичних карт, однак MapMagic має значну перевагу в інтерактивності та зручності використання під час розробки. На рисунку 1.4 зображено приклад згенерованого ландшафту та використання інструменту. [15]

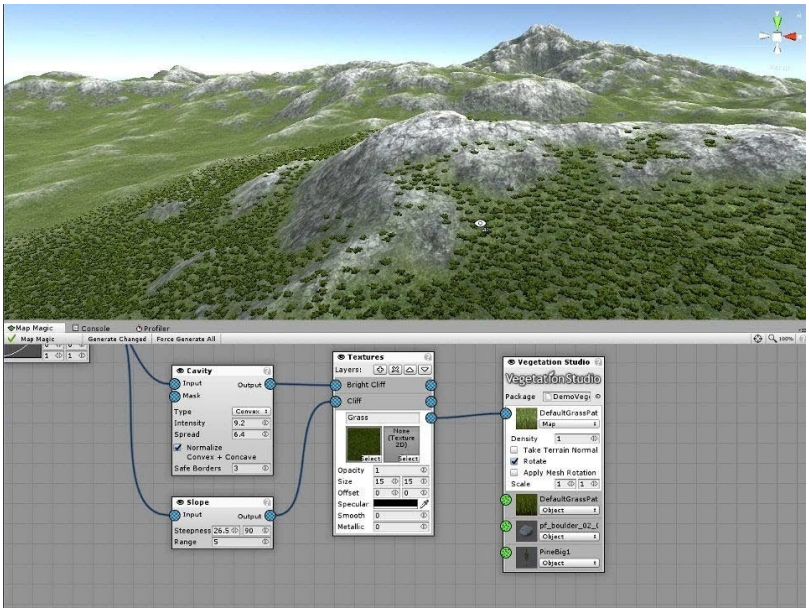

Рис. 1.4 – Процедурна генерація за допомогою MapMagic

1.9 Підходи розробки багатокористувацьких ігор

Разом з інструментами також дуже важливо вибрати правильну архітектуру та метод синхронізації гри. Найпопулярнішою моделлю в сучасній практиці є client-server: сервер виконує роль центрального вузла, яка обробляє важливу логіку гри. Цей підхід забезпечує високий рівень безпеки, оскільки клієнти не мають прямого доступу до внутрішньої логіки, а також дозволяє підтримувати цілісність ігрового процесу незалежно від якості з'єднання у гравців. Основною проблемою цієї моделі це додаткові витрати на виділений сервер, а також збільшує складність розробки та маштабування гри. [16]

Іншим варіантом є модель peer-to-peer, де всі гравці взаємодіють один з одним, а іноді один з них виступає у ролі хоста. Загалом це спрощує початкову реалізацію, та дозволяє уникнути витрат на додаткові сервера чи сервіси. Такий підхід є менш надійним тому, що це робить гру вразливою до різного роду шахрайства за допомогою підміни змінних під час гри, проте це не так важливо якщо таку систему використовують не для змагальних дисциплін, а для кооперативних. Загалом такий вид з'єднання використовують для невеликих проектів тому, що він є простіший та швидший. [17]

Окрім архітектурних моделей, важливо, також, звертати увагу на методи синхронізації та оптимізації мережевої взаємодії. У багатокористувацьких іграх потрібно забезпечити узгодженість стану між усіма клієнтами, це все досягається за допомогою механізмів синхронізації змінних, виклику віддалених процедур та передачі оновлень лише у випадку зміни параметрів. Важливою складовою є оптимізація, яка реалізується за допомогою інтерполяції та екстраполяції руху, компенсацію затримки та управління інтересами, коли клієнт отримує лише ту інформацію, яка стосується його оточення у грі.

Якщо розглядати аспект безпеки, то у сучасних підходах саме сервер виступає арбітром, який перевіряє всі дії гравців, що дозволяє запобігати шахрайству та маніпуляції серед клієнтів. Окрім цього додатково застосовують різні методи шифрування трафіку та моніторингу дій користувачів для виявлення підозрілої поведінки.

РОЗІДЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА АЛГОРИТМУ ГЕНЕРАЦІЇ СВІТУ ТА МЕТОДІВ ВЗАЄМОДІЇ МІЖ ГРАВЦЯМИ В БАГАТОКОРИСТУВАЦЬКІЙ ГРІ

2.1 Постановка задачі

Метою проєкту є розробка детерміністичного, масштабованого алгоритму процедурної генерації ігрового світу, який забезпечує узгодженість і відтворюваність на всіх клієнтах у багатокористувацькому середовищі, а також проектування та реалізація надійних механізмів взаємодії між гравцями. Ігрова механіка виживання і захисту центральної споруди використовується як приклад застосування алгоритму і мережевих методів, але не є предметом основного дослідження.

Ключовий функціонал і завдання проекту:

- алгоритм процедурної генерації світу – розробка підходу (детермінований seed, біоми, баланс розподілу ресурсів та зв’язність) з вимогами до відтворюваності на клієнтах і ефективного мережевого оновлення;
- створення та керування ігровим лобі і сесіями – механізм створення/приєднання до лобі, старт/завершення ігрової сесії;
- синхронізація ігрових даних у багатокористувацькому середовищі – синхронізація підключень, дій персонажів, споруд та ресурсів ігрового світу;
- комунікаційні методи між гравцями – інтеграція внутрішньоігрового текстового чату та голосового чату.

Результатом має стати описана та реалізована система, яка гарантує:

- відтворюваність та консистетність згенерованих світів для всіх учасників сесії;

- стабільну синхронізацію ігрових подій і станів за обмежених мережевих ресурсів;
- інтегровані засоби текстової та голосової комунікації, що працюють у контексті багатокористувацької архітектури.

Технічне завдання даного проєкту наведено в додатку А.

2.2 Методологія дослідження

2.2.1 Загальний підхід

Дослідження має прикладний характер і поєднує теоретичний аналіз алгоритмів процедурної генерації та підходів до реалізації мультиплеєрної компоненти з експериментальною перевіркою їх ефективності у програмному прототипі. Використовується поєднання аналітичних, порівняльних та експериментальних методів.

2.2.2 Методи дослідження

Методи дослідження охоплюють як теоретичні, так і практичні аспекти. На початковому етапі здійснюється аналіз літератури та джерел, спрямований на вивчення сучасних алгоритмів процедурної генерації, серед яких розглядаються Perlin noise, FBM, Voronoi, L-системи та симуляційні підходи. Паралельно проводиться ознайомлення з інструментами для багатокористувацької взаємодії, зокрема такими як Mirror, Photon і FishNet. На основі цього виконується порівняльний аналіз, що дозволяє оцінити переваги й обмеження різних методів з точки зору реалістичності, продуктивності та відтворюваності у багатокористувацькому середовищі. Подальшим етапом стає моделювання, у межах якого формується алгоритмічна модель генерації світу, заснована на шумових функціях та системі біомів. Важливою складовою є також проектування архітектури клієнт–серверної системи з урахуванням механізмів синхронізації даних між учасниками. Для практичної перевірки розроблених підходів

здійснюється програмна реалізація прототипу в ігровому рушії Unity з використанням мови C# та бібліотеки Mirror. Завершальним етапом виступає експериментальне тестування, яке включає перевірку працездатності генератора світу та оцінку ефективності мережевих механізмів за такими параметрами, як затримка, консистентність, кількість одночасних гравців та частота кадрів.

2.2.3 Інструментальні засоби

Інструментальні засоби, що застосовуються у дослідженні, охоплюють як середовище розробки, так і спеціалізовані бібліотеки та допоміжні сервіси. Основною платформою для створення ігрового середовища виступає Unity, який забезпечує зручні засоби для розробки та тестування інтерактивних прототипів. Програмна логіка реалізується мовою C# у середовищі Visual Studio, що дозволяє ефективно організувати процес написання та відлагодження коду. Для реалізації багатокористувацької взаємодії використовується бібліотека Mirror Networking, яка забезпечує побудову клієнт–серверної архітектури та синхронізацію дій між учасниками мережевої гри. Додатково застосовуються допоміжні сервіси, зокрема інструменти профілювання Unity, що дозволяють аналізувати продуктивність і виявляти потенційні вузькі місця у роботі прототипу.

2.2.4 Критерії оцінювання результатів

Критерії оцінки результатів у дослідженні охоплюють як технічні, так і функціональні аспекти. Важливим показником є відтворюваність світу на різних клієнтах за однакового початкового значення (seed), що забезпечує узгодженість ігрового середовища для всіх учасників. Не менш значущою є стабільність синхронізації ігрових подій, яка визначає коректність багатокористувацької взаємодії у реальному часі. Оцінюється також продуктивність системи, зокрема швидкість генерації карти, ефективність використання апаратних ресурсів та частота кадрів під час роботи прототипу. Додатковим критерієм виступає зручність і надійність комунікації між гравцями, що впливає на загальну якість ігрового досвіду.

2.3 Теоретичні аспекти дослідження

2.3.1 Процедурна генерація ігрового світу

У процесі розробки системи генерації світу для багатокористувацької гри було обрано підхід, що поєднує процедурну генерацію карти на основі шумових функцій та мережеву синхронізацію елементів гри за допомогою бібліотеки Mirror. Такий вибір зумовлений потребою створювати унікальні ігрові простори для кожної сесії гри, зберігаючи при цьому однакову картину світу для всіх користувачів. Шумові алгоритми дозволяють отримати реалістичний розподіл висот і формування рельєфу, тоді як мережеві виклики забезпечують узгодженість між клієнтами й сервером у багатокористувацькому середовищі. Таким чином, у центрі системи опиняється генерація даних за допомогою математичних моделей і їх подальша інтерпретація та поширення серед гравців у реальному часі.

Основу архітектури складають два класи, які тісно взаємодіють між собою. Перший з них – це статичний клас Noise, що відповідає за створення двовимірної карти висот. У його методі використовується класичний Перлін-шум, до якого додається можливість обчислення з кількома октавами. Це означає, що на карту накладається кілька рівнів шуму: від великих плавних контурів, що визначають загальні обриси континентів чи островів, до дрібніших деталей рельєфу, які створюють нерівності та підвищують реалістичність світу. Для кожної октави застосовується коефіцієнт амплітуди й частоти, які змінюють характер отриманих значень. У результаті формується масив чисел, який нормалізується у межах від нуля до одиниці. Цей масив стає основою для подальшого визначення того, які саме біоми чи об’єкти будуть розташовані в тій чи іншій точці карти.

Другий ключовий елемент системи – клас MapGenerator, що наслідує NetworkBehaviour і завдяки цьому може інтегруватися в багатокористувацьку архітектуру. Саме він керує процесом генерації світу, використовуючи дані з шумової карти для розташування тайлів, ресурсів і спеціальних об’єктів. Цей

клас працює у двох режимах: у першому формує саму карту острова, а в другому – відповідає за розміщення ресурсів у вже згенерованому світі. При створенні острова спочатку викликається метод генерації шуму, після чого застосовується додаткова функція обрізання карти по колу, щоб сформувати берегову лінію. Далі виконується аналіз висот і кожній ділянці карти призначається певний біом, наприклад, рівнина, ліс чи море. У цьому ж режимі на відповідній ділянці визначається місце для головної будівлі гравців. Алгоритм перевіряє, чи достатньо рівна і вільна площа для розміщення такої споруди, після чого визначає початкові точки появи персонажів. Розташування будівлі та спавн-поінтів синхронізуються між клієнтами, що гарантує однаковий результат для всіх користувачів.

У режимі генерації ресурсів робота алгоритму відбувається дещо інакше. Система отримує карту висот з головного генератора і перевіряє, чи відповідає кожна конкретна клітинка параметрам для розміщення певного ресурсу. Якщо висота на карті потрапляє у визначений діапазон, у цьому місці може з’явитися відповідний об’єкт, наприклад, дерево, камінь чи інший ресурс. Для уникнення надмірного скупчення перед розміщенням перевіряється відстань до вже існуючих ресурсів. Якщо поруч немає подібних об’єктів, ресурс створюється на сервері та розповсюджується серед клієнтів через механізм NetworkServer.Spawn. У такий спосіб формується збалансована й природна карта розташування корисних об’єктів, до якої мають доступ усі учасники гри.

Важливим елементом роботи MapGenerator є обробка тайлів карти. Кожне значення висоти, отримане з шумової карти, порівнюється з діапазонами, визначеними для біомів. Якщо клітинка належить до водного біома, на неї розміщується тайл води, в інших випадках – тайл суші з невеликими випадковими варіаціями кольору, що додає різноманітності та візуальної привабливості. Для цього використовується масив заздалегідь визначених значень яскравості, з яких випадковим чином обирається коефіцієнт кольору. Завдяки цьому навіть великі рівнинні ділянки не виглядають одноманітно.

Усі ключові зміни, пов’язані з картою, будівлями чи персонажами, поширюються серед клієнтів через виклики, позначені атрибутом ClientRpc. Наприклад, після визначення місця для головної будівлі сервер повідомляє всіх клієнтів про її координати, а при переміщенні персонажів відбувається синхронізація їхніх позицій у світі. Це дозволяє уникнути розбіжностей між користувачами та підтримувати єдину картину гри.

У підсумку створюється гнучка система, в якій математична модель шуму задає фундаментальні параметри світу, а клас-генератор формує з них цілісну карту з урахуванням особливостей ігрового процесу. Інтеграція з мережею робить цей процес узгодженим для багатьох гравців, що особливо важливо у випадку багатокористувацьких ігор. Таким чином, архітектура поєднує простоту модульного підходу й потужність процедурної генерації, створюючи основу для розширення й подальшої адаптації під потреби конкретного ігрового проєкту.

2.3.2 Методи комунікації між гравцями

У процесі проектування багатокористувацької гри важливим завданням є організація ефективної взаємодії між гравцями. Для цього застосовуються різні підходи, серед яких використання текстового чату для обміну повідомленнями та голосового чату для реального голосового спілкування. У даній роботі реалізовано обидва ці методи на базі бібліотеки Mirror, що забезпечує мережеву взаємодію в Unity. Такий вибір обумовлений тим, що Mirror надає прості у використанні інструменти для синхронізації даних між клієнтом і сервером, підтримує архітектуру клієнт-сервер та дозволяє легко інтегрувати різні форми комунікації. В результаті утворюється модульна система, де кожен елемент (чат, голосова передача, інтерфейс користувача) виконує свою роль, але при цьому взаємодіє з іншими компонентами у межах єдиної архітектури.

2.3.2.1. Система текстового чату

За роботу текстового обміну відповідає клас ChatNetwork, що успадковується від NetworkBehaviour. Основним елементом у його структурі є

колекція SyncList<string> messages, яка використовується для зберігання повідомлень. Ця структура автоматично синхронізується між клієнтами, забезпечуючи відображення однакових даних у всіх гравців.

Ініціалізація відбувається у методі Awake(), де створюється єдиний екземпляр класу (реалізація шаблону Singleton). При підключенні клієнта викликається OnStartClient(), де до списку повідомлень підключається метод OnMessagesChanged. Його завдання – оновлювати інтерфейс чату через ChatUI.Instance.Refresh(messages) у випадку зміни вмісту списку.

Відправлення повідомлень здійснюється через команду CmdSendMessage, що виконується на сервері. Вона отримує текст, перевіряє його довжину та формує фінальний рядок із часовою міткою та іменем відправника. Якщо кількість повідомлень перевищує допустимий максимум, найстаріший запис видаляється. Таким чином забезпечується оптимальне використання пам’яті та контроль обсягу даних, що синхронізуються. На рисунку 2.1 зображено вигляд текстового чату.

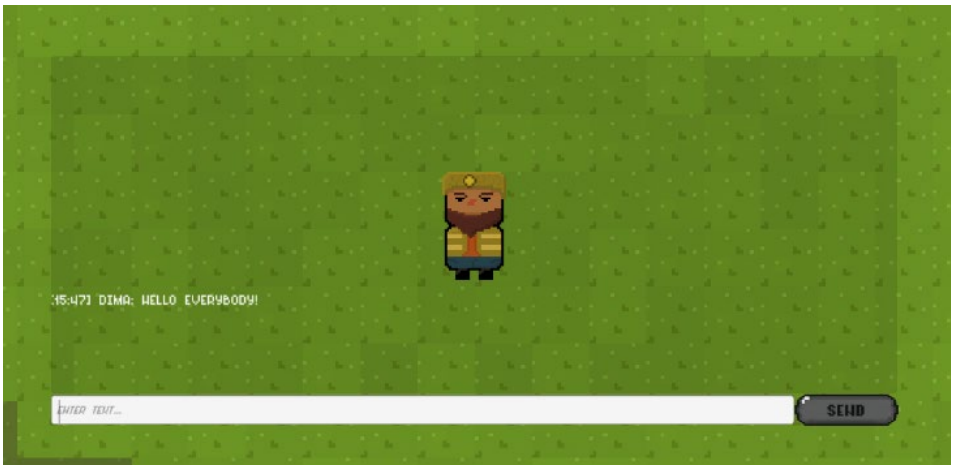

Рис. 2.1 – Текстовий чат

2.3.2.2 Система голосового чату

Для реалізації голосового зв’язку використовується клас VoiceChatNetwork, який також базується на NetworkBehaviour і вимагає наявності компонента

AudioSource. Він відповідає за запис голосу з мікрофона та його подальшу передачу іншим клієнтам.

У методі Start() ініціалізується аудіоджерело, створюється спеціальний AudioClip для відтворення отриманих даних та запускається відтворення у циклі. Система перевіряє наявність доступного мікрофона і в разі його відсутності повідомляє про помилку.

У методі Update() відбувається контроль натискання клавіші V, що слугує тригером для початку чи завершення запису. Запис ініціалізується методом StartMic(), де створюється аудіокліп і виділяється буфер під дані. При відпусканні клавіші викликається StopMic(), що завершує запис.

Передача даних здійснюється у методі SendMicData(). Він отримує нові семпли із мікрофонного буфера та розділяє їх на частини (chunks) для надсилання через мережу. Відправлення відбувається за допомогою команди CmdSendAudioChunk, що виконується на сервері та викликає RpcReceiveAudioChunk для доставки даних усім клієнтам, окрім відправника.

На стороні клієнта отримані семпли додаються у чергу playbackQueue, звідки вони зчитуються методом OnAudioRead(). Це забезпечує безперервне відтворення голосу віддалених гравців через локальний AudioSource.

2.3.3 Мережева архітектура гри

Для забезпечення багатокористувацького режиму гри використовується клієнт-серверну архітектуру. Архітектура клієнт-сервер є одним із архітектурних шаблонів програмного забезпечення та є домінуючою концепцією у створенні розподілених мережних застосунків і передбачає взаємодію та обмін даними між ними. Основною ідеєю цієї архітектури це є розподілення відповідальності та методи передачі інформації між різним клієнтами (рис. 2.2). Таким чином готовий ігровий продукт поділяється на два види застосунку серверний та клієнтський. [18]

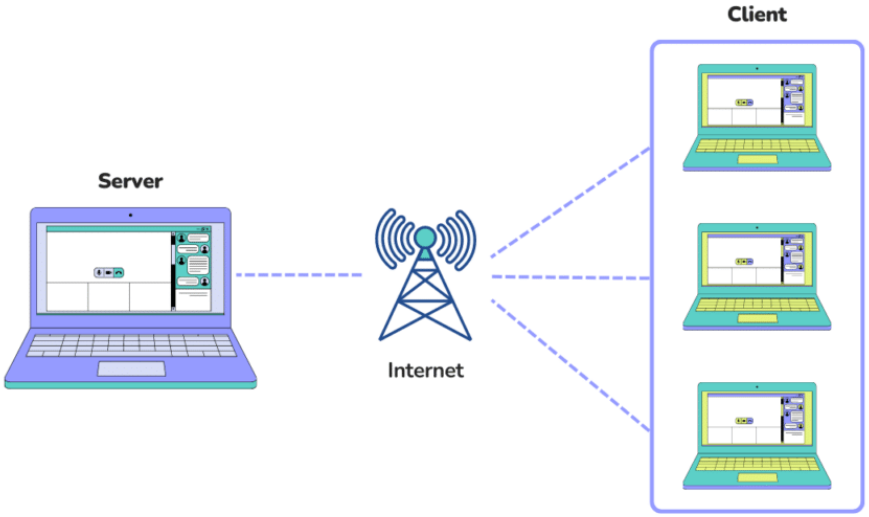

Рис. 2.2 – Архітектура клієнт-сервер

Клієнтська частина відповідає за відображення гри на екрані гравця та включає в себе графіку, анімацію, інтерфейс користувача та інші візуальні елементи. Вона також обробляє введення від гравця, наприклад клацання мишею або натискання клавіш, і перетворює ці дії на повідомлення, які відправляються на сервер для подальшої обробки. Крім того, клієнтська частина може здійснювати локальне зберігання даних, наприклад налаштувань гравця або локального стану гри, що дозволяє покращити продуктивність та забезпечити більш плавний ігровий процес.

Серверна частина є головним обчислювальним центром гри і відповідає за обробку всіх дій, що відбуваються у грі. Вона здійснює обробку введення від клієнтів, розрахунки фізики, взаємодію об’єктів гри та синхронізацію стану між усіма гравцями. Крім того, сервер відповідає за централізоване зберігання даних, включаючи світ гри, стан персонажів, інвентарі та інші глобальні параметри. Сервер також керує мережею, отримує введення від клієнтів, обробляє його та надсилає зміни стану гри назад клієнтам, забезпечуючи узгоджене функціонування всієї системи.

Синхронізація у грі забезпечується мережею, через яку клієнти взаємодіють із сервером, відправляючи та отримуючи повідомлення, що містять

інформацію про дії гравців та зміни стану гри. Важливим аспектом є також керування затримками мережі: сервер повинен ефективно їх контролювати та гарантувати, що всі гравці бачать однаковий стан гри одночасно.

Для реалізації цієї архітектури я створюю клієнтську та серверну частину застосунку, а за синхронізацію та керуванням затримкою відповідає уже інтерфейс Mirror. Більш детально взаємодію сервера та клієнтів зображено на діаграмах послідовності в додатку Б.

В нашому випадку при відсутності зовнішнього хмарного серверу було вибрано рішення використання одного з клієнтів як сервер. Таким чином клієнт, який розгортає сервер на своїй машині називається хостом. При такому рішенні виникає потреба знаходження усіх клієнтів та хоста в одній локальній фізичній чи віртуальній мережі. Такий підхід часто використовують інді компанії через нестачу коштів для орендування або створення власних серверів для підтримки багатокористувацького режиму в грі. На рис. 2.3 зображено діаграму розгортання гри-застосунку.

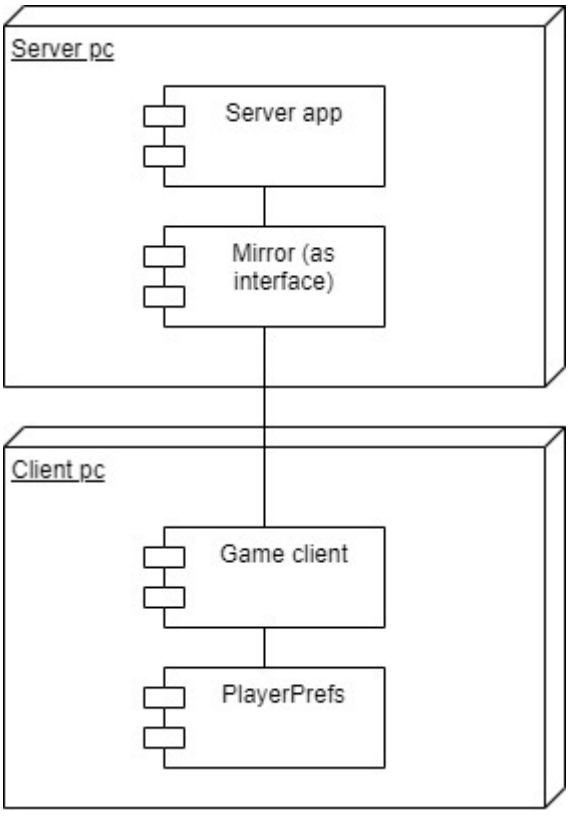

Рис. 2.3 – Діаграма розгортання системи

2.3.4 Мережева компонента гри

Super Tower Survival є складною ігровою системою яка складається з великої кількості компонентів оскільки рушій Unity є компонентно орієнтованою. В основному система складається з двох блоків таких як сервер та клієнт. Серверний блок має в собі серверну частину застосунку та бібліотеку Mirror, яка слугує інтерфейсом між серверним застосунком та клієнтським. Блок клієнта в свою чергу має такі частини як сам ігровий клієнт (Game client) та PlayerPrefs, який зберігає всю необхідну інформацію про користувача. [19]

Networking – компонента, що відповідає за мережеве з'єднання. В свою чергу вона складається з таких підкомпонентів:

- NetworkGamePlayer;
- NetworkManagerLobby;
- NetworkRoomPlayerLobby;
- PlayerSpawnSystem.

NetworkGamePlayer – це компонента, що відповідає за реєстрацію клієнта на сервері і в подальшому його переміщення до лобі зі створенням компоненти NetworkManagerLobby.

NetworkManagerLobby – це компонента, що відповідає за підключення клієнта до серверу, за відключення клієнта від серверу та створення компоненти NetworkRoomPlaerLobby уже для взаємодії клієнта з лобі для очікування. Також ця компонента відповідає за реєстрацію об'єктів, які в подальшому будуть з'являтись на сцені, без реєстрації цих об'єктів інтерфейс бібліотеки Mirror унеможливлює створення та синхронізацію об'єктів між клієнтами та серверами, тому це є необхідним. Також ця компонента відповідає за сам початок гри.

NetworkRoomPlayerLobby – це компонента, яка відповідає за взаємодію клієнта з лобі. Також ця компонента дозволяє зареєструвати шаблон персонажа для кожного клієнта, ця інформація про реєстрацію далі надходить до PlayerSpawnSystem для подальшого породження персонажів гравців на новій сцені.

PlayerSpawnSystem використовуючи данні, що надійшли від NetworkRoomPlayerLobby та після початку гри з компоненти NetworkManagerLobby віднаходить певні позиції для породження персонажів уже на новій сцені, ініціалізує їх та синхронізує між клієнтами та сервером.

2.4 Обґрунтування вибору інструментальних засобів

Вибір інструментів розробки є одним з найбільш головних етапів у створенні програмного забезпечення, оскільки від цього залежить загальний розвиток проєкту, зокрема ефективність реалізації та його маштабованість.

У межах даної роботи було прийнято рішення використовувати: генерацію світу на основі шумових алгоритмів, ігровий ришій Unity із звстосуванням середовища програмування VisualStudio та мовою програмування C#. Також для розробки мережевих компонентів гри було використано високорівневу бібіліотеку Mirror, що дає змогу ефективно організовувати багатокористувацьку взаємодію.

Вибір алгоритму генерації ігрового світу на основі шумів був зумовлений необхідністю створення природних, варіативних і водночас логічно структурованих середовищ.

Шумові функції на основі шуму Перліна та подібні до нього алгоритми дають змогу отримувати плавні та реалістичні переходи між різними елементами ландшафту. Цей підхід дозволяє позбутися надмірної хаотичності, яка притаманна випадковим розподілам, та в свою чергу гарантує достатню варіативність результатів, що дає змогу створювати цікаві для дослідження ігрові світи. В бібілотеці MathF у мові C# уже є реалізований саме шум Перліна, та конкретно в цьому проекті не була потрібна додаткова швидкість для генерації шуму, оскільки ця генерація відбувається перед стартом ігрової сесії і, тому, було використано шум Перліна на відміну його швидшого аналога Сиплекс.

Unity було обрано як основний ігровий рушій завдяки його універсальності, широким можливостям для створення 2D та 3D проєктів і розвиненій екосистемі. Цей рушій надає інтегровані засоби роботи з фізикою, анімацією та графікою, що суттєво скорочує час на реалізацію базових функціональних компонентів. Крім того, Unity має потужну спільноту розробників та підтримує мультиплатформенність, що дозволяє масштабувати проєкт під різні пристрої без значних змін у коді. У ролі середовища розробки обрано Visual Studio, оскільки воно забезпечує тісну інтеграцію з Unity, підтримує роботу з мовою C# та надає широкий спектр інструментів для відлагодження, тестування і рефакторингу коду. Використання саме C# пояснюється тим, що ця мова є базовою для Unity, поєднує високу продуктивність із зручністю синтаксису та має розвинений набір бібліотек для розв’язання різноманітних задач. [20, 21, 22]

Для реалізації багатокористувацької взаємодії було обрано бібліотеку Mirror. Її перевага полягає в оптимальному балансі між простотою використання та гнучкістю налаштувань, що дозволяє адаптувати рішення до конкретних вимог проєкту. Mirror забезпечує необхідний набір інструментів для організації клієнт-серверної архітектури, синхронізації стану об’єктів і передачі даних у реальному часі. Важливою причиною вибору є також активна підтримка бібліотеки спільнотою та сумісність із сучасними версіями Unity, що гарантує її надійність і придатність для використання у практичних розробках.

2.5 Етапи програмної реалізації

Для початку перед тим як працювати з генерацією світу для гравців, спочатку потрібно організувати їх мережеве з’єднання, і в цьому допоможе високо рівнева бібліотека Mirror Networking для Unity. Хоч з самого початку за допомого NetworkManager та NetworkingHUD в цій біліотеці можливо зробити легке з’єднання між клієнтами, цей спосіб не доже підходить для цього проекту,

тому було додатково розроблено такі класи як: JoinLobbyMenu, MainMenu, PlayerNameInput, NetworkGamePlayer, NetworkManagerLobby, NetworkRoomPlayerLobby та PlayerSpawnSystem.

JoinLobbyMenu, MainMenu та PlayerNameInput є частиною типу Lobby і відповідають уже за реакцію інтерфейсу на дії користувача та використання уже мережевих компонентів: NetworkGamePlayer, NetworkManagerLobby, NetworkRoomPlayerLobby та PlayerSpawnSystem.

NetworkGamePlayer відповідає за потворення кімнати для очікування та додавання в кімнату нових клієнтів. NetworkManagerLobby в свою чергу уже відповідає за поведінку, приєднання та від’єднання клієнтів в кімнаті для очікування. Цей клас також має такі події: OnClientConnected, OnClientDisconnected та подію OnServerRedied типу NetworkConnection. Перші дві події реагують на приєднання нових клієнтів та відключення наявних, а OnServerRedied повинен викликатись при старті уже самої гри, тобто коли уже всі необхідні підготовки готові і можна переносити усіх клієнтів уже на нову сцену гри. Код класу NetworkManagerLobby наведено в додатку В.

NetworkRoomPlayerLobby – це клас який існує лише при приєднанні та очікуванні гравця в лобі. Відповідно це клас і є об’єктом який визначає самого клієнта на сцені кімнати очікування. Цей клас відповідає за готовність гравця та його вибір персонажа перед початком гри. Код класу NetworkRoomPlayerLobby наведено в додатку Г.

Після старту гри перенесені усіх гравців на нову сцену за своє діло повинен братись клас PlayerSpawnSystem, який використовує інформацію про вибраних персонажів та відповідно створює для кожного клієнта вибраного ним персонажа. Код класу PlayerSpawnSystem наведено в додатку Д.

Перед тим як перейти до генерації світу варто зазначити яким саме чином відбувається синхронізація будь яких дій між клієнтами та сервером у бібліотеці Mirror. Mirror має кілька основних підходів до синхронізації дій та станів об’єктів між клієнтом та сервером. Наведемо декілька з них які будуть фігурувати у розроблюваних частинах коду.

SyncVar – Використовується для автоматичної синхронізації простих змінних (int, float, string, bool, Vector3, тощо). Значення зберігається на сервері і автоматично розсилається всім клієнтам. Підтримує хуки (callback), щоб реагувати на зміну змінної на клієнті.

Commands ([Command]) – Викликаються на клієнті, але виконуються на сервері.

ClientRpc ([ClientRpc]) – Викликається на сервері, виконується на усіх клієнтах. Зручно для “широкомовлення” (наприклад, показати ефект вибуху всім).

Server ([Server]) – Використовується для методів, які можна викликати тільки на сервері.

SyncLists та SyncDictionaries – Спеціальні колекції (аналог List / Dictionary), які Mirror синхронізує автоматично.

Насамперед для реалізації генератора світу потрібно підготувати деякі компоненти на самій сцені, де буде відбуватись генерація, а саме компоненти, які відповідають за розміщення та відображення результату класу MapGenerator.

Тут присутні три компоненти: Grid, Tilemap, Tilemap Renderer та Tilemap Collider 2D. Ці компоненти тісно пов’язані й утворюють основу для побудови двовимірних світів на основі плиток.

Усе починається з компоненти Grid, який створює сітку певного розміру та структури. Саме він визначає координатний простір, у якому розташовуються клітинки. Сітка може бути квадратною, ізометричною чи шестикутною, а розмір клітинок та відстань між ними задаються у властивостях Grid. На цьому рівні ще немає графіки, це лише “каркас”, у який згодом вставляються інші елементи.

На основі Grid додається Tilemap – це свого роду шар, у який можна “намалювати” плитки. Tilemap працює як полотно, де розміщуються спрайти плиток, їх колір чи навіть анімації. Завдяки йому можна створювати різні рівні чи локації, використовуючи єдину сітку. Наприклад, один Tilemap може містити землю, інший – об’єкти оточення, а ще один – декоративні елементи на передньому плані. У цьому випадку в проекті присутні чотири Tilemap: Water,

Land, Obtainables (ресурси) та Structures. Конкретно в MapGenerator використовуються перші три.

Щоб намальовані плитки стали видимими в грі, до Tilemap додається Tilemap Renderer. Цей компонент відповідає за відображення й дозволяє контролювати, як саме будуть рендеритися плитки: чи поодинці, чи об’єднаними групами для оптимізації. Саме він також визначає порядок відображення шарів.

Останнім важливим елементом є Tilemap Collider 2D. Він автоматично створює колайдери на основі форми намальованих плиток. Завдяки цьому персонаж має здатність ходити по землі, упиратися в стіни чи взаємодіяти з об’єктами оточення. Для кращої продуктивності Tilemap Collider часто поєднують із Composite Collider 2D, щоб уникнути великої кількості дрібних колайдерів і перетворювати їх у більш цілісні геометричні форми. Після попередніх підготовок можна розпочати реалізацію генератора світу.

Для процедурної генерації карти на основі шуму було відповідно створено статичний клас Noise який відповідно мав статичний метод GenerateNoiseMap який приймає висоту та ширину полотна шуму яку потрібно згенерувати, зерно (seed), масштаб шуму (scale), кількість октав (octaves), коефіцієнт зменшення амплітуди при переході до наступної октави (persistance), коефіцієнт збільшення частоти для наступної октави (lacunarity) та двохвимірний вектор зміщення (offset). Спочатку формується масив для збереження результатів і ініціалізується генератор випадкових чисел на основі заданого зерна (seed). Для кожної октави шуму визначається випадкове зміщення, щоб уникнути повторюваних візерунків і зробити карту більш природною. Далі виконується подвійний цикл, який проходить через усі точки майбутньої карти. Для кожної точки обчислюється шум Перліна з урахуванням кількох октав. Перша октава задає базову форму, а наступні додають деталі, змінюючи амплітуду та частоту відповідно до параметрів persistance і lacunarity. Отримане значення зберігається у масив і одночасно оновлюється інформація про мінімальне та максимальне значення шуму. Після заповнення карти усі значення в масиві нормалізуються і далі

результат виконання цього методу буде використовуватись уже безпосередньо в класі MapGenerator. Код класу Noise представлений в додатку Е.

Клас MapGenerator є центральним елементом генерації ігрової карти у проєкті. Він наслідується від NetworkBehaviour, що дозволяє йому працювати не лише з локальними даними, а й синхронізувати зміни між сервером та клієнтами. Основним його завданням є створення шумової карти, перетворення її у набір біомів, розміщення головних будівель та ресурсів, а також візуалізація результатів у вигляді тайлів на карті.

Головний метод у цьому класі – це GenerateMap, який виконується лише на сервері. Першим кроком він очищує основний тайлмап від попередніх даних, щоб забезпечити коректний старт нової генерації. Далі виконується виклик функції з класу Noise, що створює двовимірний масив чисел у діапазоні від нуля до одиниці. Це значення висот, які у подальшому інтерпретуються як різні типи територій. Отримана карта шуму зберігається у властивості NoiseMap, і вже на її основі залежно від вибраного режиму формується подальша логіка.

У випадку режиму Island метод намагається розмістити головну будівлю на суші. Для цього використовується допоміжна функція TryPlaceMainHall, яка шукає підходящу ділянку відповідно до висотних значень карти. Після цього до карти застосовується модифікатор у вигляді функції CutIsland. Вона поступово зменшує значення висот ближче до країв масиву, створюючи ефект острова. Лише після цього викликається PlaceTiles, яка проходить крізь усі координати карти, визначає для кожної точки відповідний біом і розташовує тайли на відповідних тайлмапах. У випадку водних біомів тайли потрапляють у спеціальний водяний шар, а для суші підбираються відтінки кольору, що створюють різноманітність у межах одного біому.

У випадку режиму Resources генератор не створює саму землю, а накладає на вже згенеровану карту ресурси. Для цього він звертається до іншого екземпляра MapGenerator, що відповідає за сушу, і використовує його карту шуму як основу. Потім функція PlaceObtainables проходить усі координати й визначає, чи підходить конкретна точка для розташування об’єкта з набору ресурсів. У

процесі перевіряється як висота на карті суші, так і додаткові умови на власній шумовій карті. Щоб уникнути скупчення ресурсів, застосовується перевірка відстані до вже розміщених об’єктів, і лише за умови достатнього простору ресурс створюється у світі. Його поява супроводжується викликом NetworkServer.Spawn, що дозволяє синхронізувати об’єкт між усіма клієнтами.

Усі важливі дії, що змінюють вигляд карти чи розташування об’єктів, реалізовані через методи з атрибутом ClientRpc. Це означає, що сервер виконує ключові обчислення і після цього надсилає команди клієнтам, які відображають результати. Таким чином, функції PlaceTile, PlaceWaterTile, PlaceBase і MovePlayer відповідають за візуалізацію змін на стороні клієнта. Тайли на основному та водному шарах відтворюються однаково для всіх гравців, головна будівля з’являється у вибраній точці, а персонажі телепортуються на визначені місця.

Окремим аспектом роботи класу є система перетворення координат. Оскільки ігрові об’єкти існують у світових координатах Unity, а шумова карта зберігається у вигляді двовимірного масиву, необхідно мати механізм переводу одних координат в інші. Для цього існують дві перевантажені версії функції WorldCoordsToNoiseArray, які на основі заданих позицій повертають індекси в масиві шуму. Це дозволяє, наприклад, визначити біом у будь-якій точці світу через метод TileNameOnCoords.

Загалом MapGenerator поєднує в собі математичні методи генерації шуму з механікою побудови тайлів і мережею синхронізації. Він забезпечує цілісний цикл створення карти: від числових даних шуму через інтерпретацію у вигляді біомів – до візуалізації у грі та розподілу ресурсів. На рисунку 2.4 зображена блок схема алгоритму генерації світу. Код класу MapGenerator представлений в додатку Ж.

Клас ChatUI у даному проекті займає ключову роль у керуванні текстовим чатом у грі. Він наслідується від MonoBehaviour, тому інтегрується безпосередньо у життєвий цикл Unity-об’єкта та працює як частина сцени. Його призначення

полягає у взаємодії з гравцем через користувацький інтерфейс, обробляючи введення тексту та водночас комунікуючи з мережею через клас ChatNetwork.

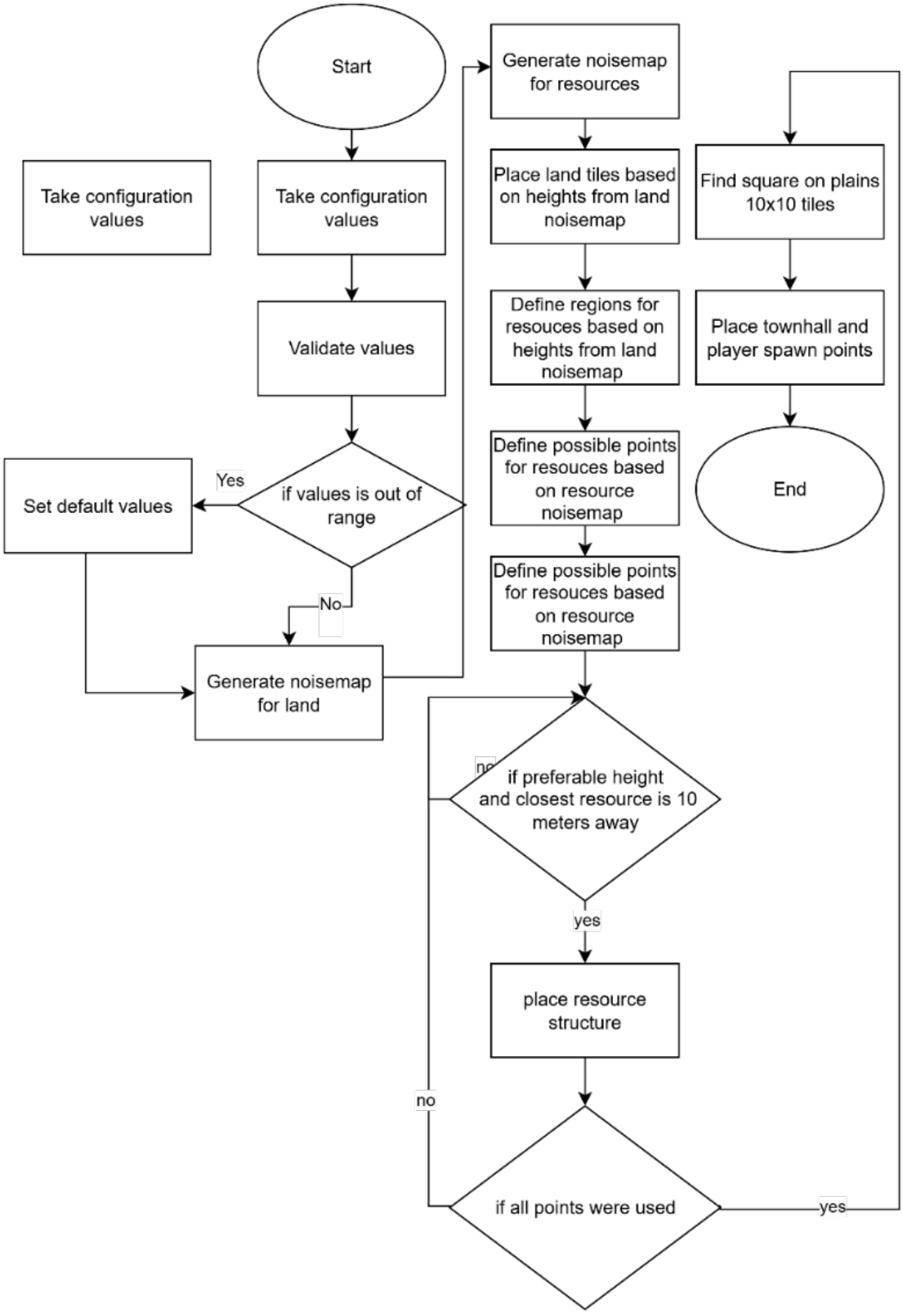

Рис. 2.4 – Блок-схема алгоритму генерації світу

На сам перед, для зручності користування з будь якої частини коду, для цього класу було використано патерн Singletone. На етапі ініціалізації класу також у внутрішню змінну зберігається ім’я гравця для подальшого його відображенні у текстовому чаті.

Наступна логіка починає працювати у методі Start, де налаштовується зв’язок з елементами інтерфейсу користувача: кнопка відправки повідомлення отримує обробник, який викликає функцію SendFromUI, а поле введення налаштовується так, щоб при натисканні клавіші Enter також здійснювалась відправка. Якщо користувач приєднався до гри пізніше, то викликається функція Refresh, яка оновлює журнал повідомлень у вікні чату, завантажуючи уже наявні записи.

Функція Refresh відповідає за відображення списку повідомлень у текстовому полі. Вона будує лог за допомогою StringBuilder, після чого змінює позицію прогортання, автоматично прокручуючи вікно вниз, аби користувач бачив останні повідомлення.

Якщо розглядати метод SendFromUI, то він фактично передає текст у мережу. Він перевіряє, чи поле введення не порожнє та чи клієнт підключений до сервера через NetworkClient.active. Якщо усі умови виконано, то викликається команда CmdSendMessage, яка вже відповідає за розповсюдження повідомлення серед усіх гравців. Після чого поле очищується та курсор повертається в активний стан, що спрощує роботу з ним. Код даного класу наведено в додатку И.

Клас ChatNetwork є центральною частиною мережевої взаємодії текстового чату у грі. Він наслідується від NetworkBehavior, що робить його невід’ємною частиною мережевої архітектури та дозволяє виконувати синхронізацію даних між усіма клієнтами та сервером. Його завдання полягає в тому, щоб обробляти повідомлення, які надсилають гравці, зберігати їх у спільному списку та поширювати серед усіх підключених користувачів.

Цей клас також реалізовано з використанням шаблону Singleton. Ключовим елементом у класі є змінна messages типу SyncList – синхронізований список

рядків. Його особливість полягає втому, що Mirror автоматично відслідковує зміни та передає їх усім клієнтам, що забезпечує однаковий стан повідомлень між усіма клієнтами. Кожне додавання, видалення чи оновлення елемента цього списку поширюється мережею без необхідності ручного дублювання даних.

При підключенні клієнта спрацьовує метод OnStartClient, у якому клас підписується на подію Callback у синхронізованому списку, коли будь яким чином змінюватиметься цей список буде викликатись функція OnMessageChanged. У цій функції перевіряється, чи існує активний об’єкт чату, і якщо так, то відбувається оновлення візуальної частини через Refresh. Таким чином будь-яке нове повідомлення з’являється зразу у всіх користувачів. При завершені роботи клієнта у методі OnStopClient видаляється підписка на подію, аби уникнути при новому підключенні подвоєнь повідомлення через уже існуючі підписи.

Найважливішим методом цього класу є CmdSendMessage, який позначений атрибутом Command. Це означає, що виклик виконується з боку клієнта проте сама логіка виконується на серверній частині. Саме в цьому методі перевіряється валідність тексту: порожні або надто довгі повідомлення обрізаються чи відкидаються, потім формується фінальний рядок, який включає часову позначку, ім’я гравця та текст повідомлення. Якщо список перевищує встановлений ліміт повідомлень то найстаріше з них видаляється, щоб уникнути набмірного зростання історії. Код класу ChatNework зображено у додатку К.

Клас VoiceChatNetwork наслідується від NetworkBehavior, що на дає можливість працювати на пряму з мережею. Головною задачею цього класу є захоплення голосу з мікрофону, поділ аудіо на невеликі блоки, пересилання цих блоків мережею та відтворення на клієнтах.

При добавлянні цього компонти класу на ігровий об’єкт в Unity автоматично створюється додатково компонента AudioSource яка відповідно дозволяє відтворювати звуки у середовищі. У методі Start налаштовується аудіосистема: створюється об’єкт типу AudioClip для відтворення вхідних даних від інших клієнтів, вмикається його постійне програвання та перевіряється

наявність мікрофона. Якщо пристрій знайдено, його ім’я зберігається, інакше виводиться повідомлення про помилку.

У методі Update клас перевіряє, чи належить даний об’єкт локальному користувачу, якщо так то виконання коду проходить далі, інакше виходить з методу. Це необхідно робити щоби запобігти дублювання голосів від інших гравців. Натискання клавіші V активується запис через метод StartMic, який запускає захоплення аудіо з мікрофона в режимі реального часу. Тут створюється буфер, зберігається позиція початку запису, а система чека поки мікрофон активується та почне запис аудіо. При відпусканні клавіші мікрофон перестає захоплювати данні за допомогою методу StopMic. Якщо мікрофон активний на кожному кадрі виконується метод SendMicData, який відповідно передає дані по мережі.

Метод SendMicData обробляє звукові данні із буфера. Спочатку визначається кількість нових записів, які з’явилися з часу останнього зчитування, та ділить їх на менші частини відповідно до розміру chunkSize. Це дає можливість ефективно відправляти дані невеликими блоками, зменшуючи навантаження на мережу. Кожен блок формується масивом чисел із плаваючою комою, що представляють амплітуду сигналу, та передається за допомогою методу CmdSendAudioChunk.

Якщо розглядати останній згаданий метод, то він позначений атрибутом Command, що дозволяє клієнтам виконувати код на сервері. CmdSendAudioChunk в собі викликає RpcReceiveAudioChunk, який має атрибут ClientRpc. Це означає, що метод буде виконано на всіх клієнтах окрім відправника. Тут отримані аудіофрагменти додаються у playbackQueue, чергу, з якої данні зчитуються під час відтворення.

Відворення аудіо реалізоване у методі OnAudioRead. Він викликається коли наявна аудіодоріжка завершується та їй потрібна нова порція аудіозапису. Звідси вибираються семпли з черги та передаються у масив data, що відповідно використовується аудіосистемою для синтезу звуку. Якщо данних не достатньо, то решта заповнюється нулями для уникнення шумів. У результаті гравець у

реальному часі має змогу чути та розмовляти з іншими гравцями. Код класу VoiceChatNetwork наведено в додатку Л.

Після завершення усіх етапів програмної реалізації також було написано інструкцію користувача, яка подана у додатку М.

2.6 Організація тестування та налагодження програмного засобу

Було проведено аналіз якості розробленої програми. Для цього були виконані різноманітні випробування, спрямовані на перевірку працездатності всіх функцій програми та виявлення можливих недоліків, які можуть вплинути на її роботу. Для проведення аналізу були складені відповідні тест-плани, включаючи тестові випадки для перевірки функціональних вимог. Кожен з них містив опис виконаних кроків та очікуваний результат, який порівнювався з фактичними результатами для визначення успішності тесту.

2.6.1 Розробка тестів

Для проведення тестування розроблюваної гри було створено функціональні тести [23], проведено тестування ігрової мережі та Endurance тестування [24].

Для тест-кейсів було визначено очікувані результати та порівняно їх з дійсними.

Тестування буде вважатись успішним при дотриманні даних критерій:

- відсутність помилок, що можуть потенційно вивести гру з ладу;
- функціональні вимоги покриті тестами на 85%;
- найдені дефекти в процесі тестування успішно усунуто;
- усі тести успішно пройдені.

Мережеве тестування гри та Endurance тестування будуть вважатись успішними при стабільній роботі гри протягом усього часу тестування.

Функціональне тестування. Для проведення тестування програми на відповідність функціональним вимогам було створено ряд функціональних тестів. Усі тестові випадки можна переглянути в додатку ?.

У табл. 2.1 наведено кількість складених тестів для кожної з розроблюваних характеристик.

Таблиця 2.1.

Кількість пройдених тестів

№	*Тест кейс*	*Кількість тестів*
1	Розробка мережевої компоненти	4
2	Генерація світу	2
3	Текстовий чат	2
4	Голосовий чат	3
	Сумарно	11

Мережеве тестування гри. Для тестування активності мережі було застосовано два запущених клієнта гри та приєднаних до однієї сесії гри. За допомогою таких тестів ми перевіряємо роботу та поведінку програми в мережі, мережевий трафік. Тестування мережі є необхідним для подальшої оптимізації та визначення необхідних характеристик системи для безперешкодного використання застосунку.

Моніторинг мережі відбувався за допомогою вбудованої програми Resource Monitor в операційній системі Windows.

Endurance тестування. Endurance тестування використовується для перевірки ефективності ПЗ протягом довгого часу. Для тестування було використано вбудований плагін Profiler в Unity. В цьому плагіні є можливість записувати активність комплектуючих ПК та загалом час виконання всіх операцій в грі. Завдяки Endurance тестуванню можна виявити чи гра споживає приблизно рівну кількість ресурсів протягом усього часу. [25]

2.6.2 Результати функціонального тестування

Під час тестування було виявлено проблему лише в одному тест-кейсі пов’язаним з голосовим чатом: було виявлено нестабільну роботу, переривання звуку та некоректне накладання голосів усіх гравців. За допомогою введення циклічного буфера програвання, було виправлено дані недоліки.

2.6.3 Результати мережевого тестування

Під час тестування мережі було виявлено, що з’єднання в локальній мережі є цілісним. Також при тестуванні було виявлено середній даних при передачі та отриманні, а саме приблизно 6000 байтів за секунду. Також при тестуванні мережі не було виявлено втрати пакетів. Загальна затримка синхронізації між клієнтами становить не більше 1 секунди. Метрики з тестування зображено на рис. 2.5.

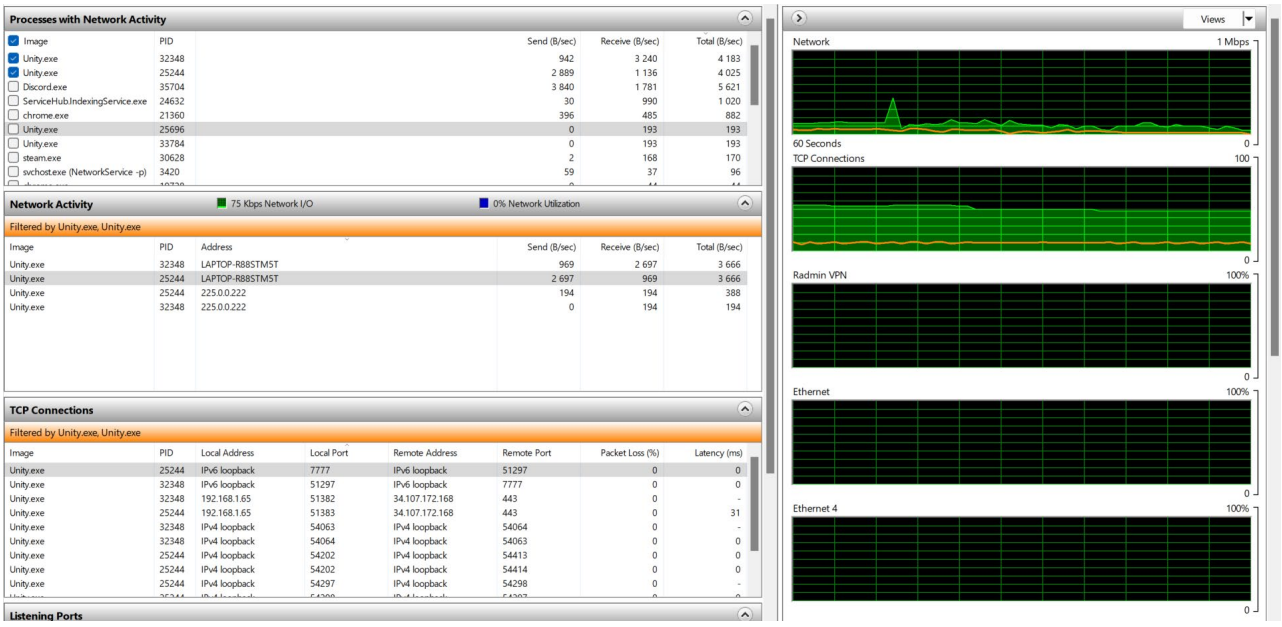

Рис. 2.5 – Метрики активності мережі

2.6.4 Результати Endurance тестування

Під час тестування було виявлено різкий скачок у використанні ресурсів, що призводило до втрати частоти кадрів за секунду раз в секунду з 200 кадрів на секунду до 60. Проте провівши аналіз отриманих метрик було виявлено, що ці

скачки є причиною циклу самого редактора Unity, тому на саму гру, а точніше на збірку гри це не повинно ніяк впливати. Загалом було записано 156739 кадрів та проаналізовано останні 300 для визначення проблеми. Протягом усього тесту окрім циклів редактора не було помічено нічого аномального, та гра працює зі стабільними 200 кадрів за секунду. Аналіз останніх 300 кадрів можна подивитись на рис. 2.6.

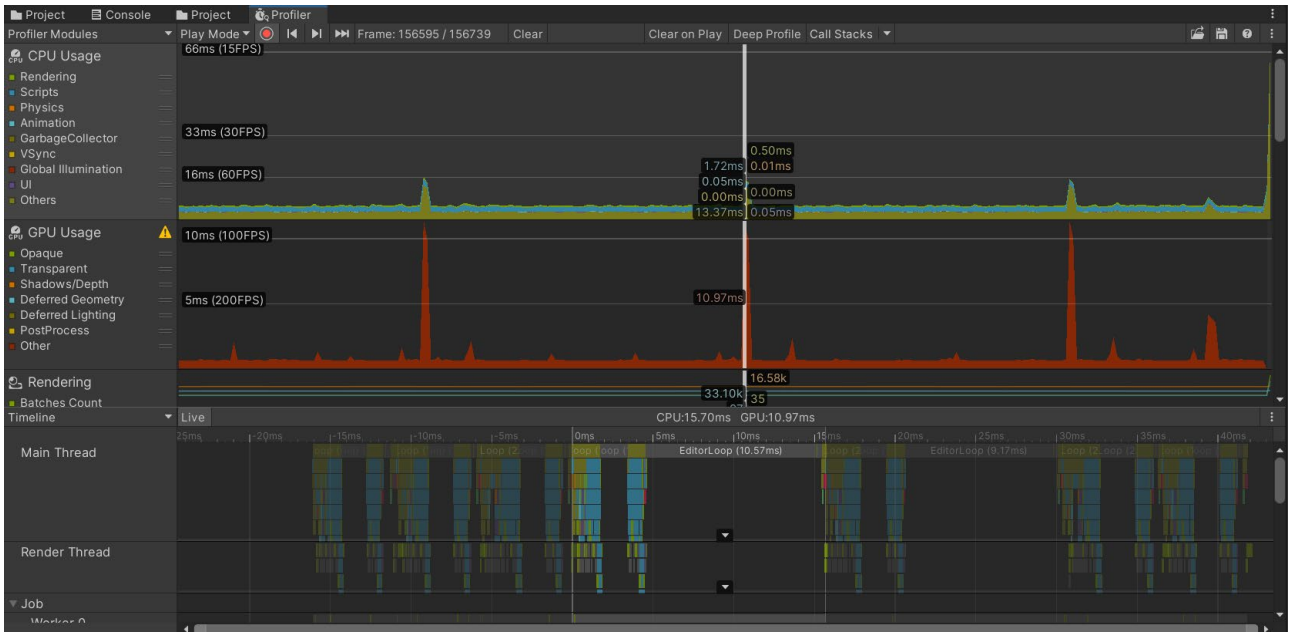

Рис. 2.6 – Результати Endurance тестування

2.7 Аналіз отриманих результатів дослідження, рекомендації щодо використання та впровадження

Проведене тестування програмного забезпечення показало високу стабільність і працездатність розробленої системи процедурної генерації світу та багатокористувацької взаємодії. За результатами функціонального тестування усі основні компоненти такі як: генератор світу, текстовий і голосовий чат та мережева синхронізація працюють коректно. Єдина виявлена проблема стосувалася короткочасних збоїв у передачі голосових даних, що була усунена

впровадженням циклічного буфера програвання, після чого робота чату стабілізувалася.

Під час мережевого тестування було зафіксовано середню швидкість передачі даних на рівні 6000 байтів за секунду, без втрати пакетів та із затримкою синхронізації не більше 1 секунди між клієнтами. Це свідчить про достатню оптимізацію мережевої архітектури на основі бібліотеки Mirror та ефективну обробку подій у клієнт–серверній структурі. Дані результати можна вважати репрезентативними для локальної мережі середнього навантаження.

Результати Endurance тестування підтвердили стабільність роботи системи при тривалому навантаженні. Протягом багатогодинного циклу тестування гра підтримувала середню частоту 200 кадрів за секунду, за винятком коротких спадів FPS до 60, які були спричинені внутрішніми циклами редактора Unity і не впливають на збірку гри. Після усунення даного впливу кінцевий продукт демонструє стабільну продуктивність без суттєвих витрат ресурсів.

Аналіз споживання ресурсів показав, що використання процесора під час активної сесії гри не перевищує 25-30%, а оперативної пам’яті близько 500-600 МБ (рис. 2.7), що є прийнятним для більшості сучасних пристроїв. Час генерації карти середнього розміру становить 1,5–2 секунди, що забезпечує динамічний старт сесії без помітних затримок.

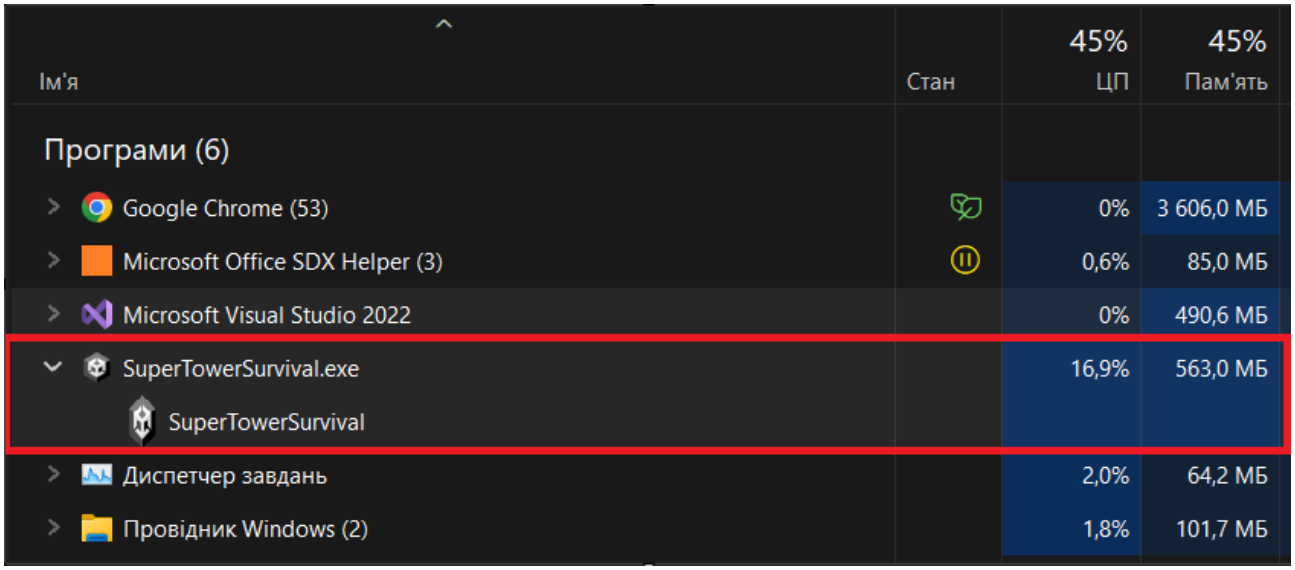

Рис. 2.7 – Використання ресурсів комп’ютера програмою

Отже, розроблене програмне забезпечення відповідає заявленим вимогам щодо ефективності, відтворюваності світу та стабільності багатокористувацької взаємодії. Його практичне застосування можливе у розробці навчальних, експериментальних та інди-ігор, де важлива процедурна варіативність та реалістична комунікація між гравцями. Проте не зважаючи на задовільні результати, для такого рівня проекту можлива подальша оптимізація процесів, оскільки є наявне вузьке горло при використанні потужності графічного процесора, що в результаті зменшує кількість кадрів на секунду від можливого ідеального результату.

Для коректного використання програми рекомендується використання таких програмних та апаратних засобів:

- операційна система – Windows 10 або вище;
- центральний процесор – Intel Core i5-4750HQ;
- оперативна пам’ять – 4 ГБ;
- відеокарта – Nvidia GeForce GT 930M.

Загалом отримані результати свідчать про готовність системи до подальшого вдосконалення та масштабування. Подальші етапи розробки можуть включати оптимізацію мережевих алгоритмів для роботи через Інтернет, розширення параметрів генерації світу та впровадження механізмів адаптивної синхронізації для зменшення затримок при великій кількості клієнтів.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено комплексне дослідження теоретичних, методологічних та практичних аспектів процедурної генерації ігрових світів і реалізації багатокористувацької взаємодії. Основною метою дослідження було створення системи, що поєднує детерміністичні методи побудови світу з ефективною архітектурою мережевої синхронізації, здатною забезпечити стабільну взаємодію між гравцями у спільному середовищі.

Було проаналізовано сучасний стан і тенденції розвитку ігрової індустрії, у якій процедурна генерація виступає ключовим інструментом підвищення реіграбельності, варіативності та масштабності ігрових світів. Розглянуто основні типи алгоритмів процедурної генерації, зокрема ті, що ґрунтуються на шумових функціях (Perlin, Simplex, Voronoi), граматичних системах (L-системи, граф-граматики), симуляційних моделях і мозаїчних підходах. Для кожного типу визначено особливості використання, переваги, недоліки та придатність до різних жанрів відеоігор. Окрему увагу приділено гібридним системам, які поєднують кілька методів з метою досягнення більшої реалістичності й різноманітності результатів.

Дослідження інструментів для процедурної генерації показало, що професійні системи подібні до World Machine орієнтовані на створення реалістичних статичних ландшафтів, тоді як MapMagic World Generator є придатним для динамічної генерації під час виконання гри, що робить його доцільним вибором у проєктах з відкритими або нескінченними світами. Проведений аналіз дозволив обґрунтувати вибір підходів і програмних засобів, застосованих у практичній частині роботи.

Другим напрямом дослідження стала розробка та інтеграція багатокористувацької компоненти. Проаналізовано різні архітектури з точки зору продуктивності, безпеки, стабільності з’єднань і масштабованості. Проведено порівняльний аналіз найпоширеніших інструментів мережевої взаємодії Mirror, Photon та FishNet, за різними критеріями. На основі результатів аналізу

обґрунтовано вибір бібліотеки Mirror, яка найбільш відповідає потребам інді-розробки та дозволяє ефективно реалізувати мережеву логіку без додаткових комерційних сервісів.

Далі було створено прототип багатокористувацької гри, у якій реалізовано детерміністичний алгоритм процедурної генерації світу на основі багатошарових шумових функцій. Розроблений алгоритм забезпечує стабільну відтворюваність результатів на всіх клієнтах при використанні однакового seed, що є критично важливим для мережевого режиму. На основі карти висот формується система біомів, виконується розподіл ресурсів і визначення точок взаємодії між гравцями. Додатково реалізовано механізм перевірки рівності рельєфу для розміщення основних об’єктів і стартових позицій, що підвищує баланс і зручність ігрового процесу.

Також було реалізовано такі комунікаційні засоби між гравцями, як текстовий і голосовий чат, що використовує методи передачі аудіопотоку. Ці рішення забезпечують повноцінну комунікацію в межах спільної ігрової сесії без сторонніх сервісів.

У результаті було розроблено комплексну архітектуру, що поєднує процедурну генерацію з мережевою синхронізацією на основі бібліотеки Mirror.

Також було проведене функціональне, мережеве та endurance-тестування підтвердило працездатність системи та її здатність забезпечувати стабільну роботу при підключенні кількох клієнтів. Середні результати тестів засвідчили низьку затримку оновлення (до 1 с), стабільну частоту кадрів понад 200 FPS та відсутність розсинхронізації між клієнтами при передачі даних у реальному часі.

У ході дослідження розроблено алгоритмічну і програмну базу для подальшого розвитку систем процедурної генерації та багатокористувацьких ігор. Практична цінність отриманих результатів полягає у створенні універсального підходу, який може бути адаптований для різних жанрів і типів ігор. Таким чином, мета магістерської роботи була досягнута повністю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Procedural content generation for games / M. Hendrikx et al. *ACM transactions on multimedia computing, communications, and applications*. 2013. Vol. 9, no. 1. P. 1–22. URL: https://doi.org/10.1145/2422956.2422957 (date of access: 1.09.2025).
2. Chapter 5. implementing improved perlin noise. *NVIDIA Developer*. URL: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-5-implementing-improved-perlin-noise (date of access: 1.09.2025).Freiknecht J., Effelsberg W. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*. 2017. Vol. 1, Issue 1. Article 6.
3. Terrain generation using procedural models based on hydrology / J.-D. Génevaux et al. *ACM transactions on graphics*. 2013. Vol. 32, no. 4. P. 1–13. URL: https://doi.org/10.1145/2461912.2461996 (date of access: 4.09.2025).
4. Contributors to Minecraft Wiki. Noise generator. *Minecraft Wiki*. URL: https://minecraft.fandom.com/wiki/Noise_generator/ (date of access: 4.09.2025).
5. RambusPress. The algorithms of No Man's Sky. *Rambus*. URL: https://www.rambus.com/blogs/the-algorithms-of-no-mans-sky-2/ (date of access: 4.09.2025).
6. World generation - dwarf fortress wiki. *Dwarf Fortress Wiki*. URL: https://dwarffortresswiki.org/index.php/World_generation#The_generation_process (date of access: 4.10.2025).
7. Pinos J., Vozenilek V., Pavlis O. Automatic geodata processing methods for real-world city visualizations in cities: skylines. *ISPRS international journal of geo-information*. 2020. Vol. 9, no. 1. P. 17. URL: https://doi.org/10.3390/ijgi9010017 (date of access: 6.09.2025).
8. General | mirror. *Mirror Networking | Mirror*. URL: https://mirror-networking.gitbook.io/docs/manual/general (date of access: 6.09.2025).

9. Multiplayer game development made easy | photon engine. *Multiplayer Game Development Made Easy | Photon Engine*. URL: https://www.photonengine.com/ (date of access: 6.09.2025).
10. Networking & netcode software solution | unity. *Unity*. URL: https://unity.com/products/netcode (date of access: 7.09.2025).
11. What Is FishNet? | Fish-Net: Networking Evolved. *What Is FishNet? | Fish-Net: Networking Evolved*. URL: https://fish-networking.gitbook.io/docs (date of access: 13.10.2025).
12. Amazon GameLift Family. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/gamelift/ (date of access: 8.09.2025).
13. Full Stack LiveOps, Real-Time Control. *PlayFab*. URL: https://playfab.com/ (date of access: 8.09.2025).
14. World Machine: The Leading 3D Terrain Generation Software. *World Machine: The Leading 3D Terrain Generation Software*. URL: https://www.world-machine.com/ (date of access: 8.09.2025).
15. MapMagic world generator | terrain | unity asset store. *The Best Assets for Game Making | Unity Asset Store*. URL: https://assetstore.unity.com/packages/tools/terrain/mapmagic-world-generator-56762?srsltid=AfmBOopRhy0xdeRQuCbV7XkoNnRvvnlr0dMpyYt0AKbRXwL_boCf1wOc (date of access: 10.09.2025).
16. Contributors to Wikimedia projects. Client–server model - Wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Client–server_model (date of access: 10.09.2025).
17. Contributors to Wikimedia projects. Peer-to-peer - wikipedia. *Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Peer-to-peer (date of access: 18.09.2025).
18. Yādava S. C. An introduction to client/server computing. New Delhi : New Age International (P) Ltd., Publishers, 2009.

19. Engelbrecht D. Building multiplayer games in unity. Berkeley, CA : Apress, 2022. URL: https://doi.org/10.1007/978-1-4842-7474-3 (date of access: 18.09.2025).
20. Unity real-time development platform | 3D, 2D, VR & AR engine. *Unity*. URL: https://unity.com/ (date of access: 20.09.2025).
21. Visual studio: IDE and code editor for software development. *Visual Studio*. URL: https://visualstudio.microsoft.com/ (date of access: 21.09.2025).
22. C# - a modern, open-source programming language | .NET. *Microsoft*. URL: https://dotnet.microsoft.com/en-us/languages/csharp (date of access: 22.09.2025).
23. What is functional testing and how to run it? - QA madness. *QA Madness*. URL: https://www.qamadness.com/knowledge-base/what-is-functional-testing-and-how-to-run-it/ (date of access: 25.09.2025).
24. GeeksforGeeks. Endurance testing - software testing - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/software-testing/software-testing-endurance-testing/ (date of access: 13.10.2025).
25. Unity - manual: unity profiler. *Unity - Manual: Unity 6.2 User Manual*. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/Profiler.html (date of access: 13.10.2025).

ДОДАТКИ

Додаток А. Технічне завдання проєкту

Вступ

Повна назва програмного забезпечення: “SuperTowerSurvival”.

Програмний засіб призначений для створення та дослідження процедурно згенерованих ігрових світів у багатокористувацькому середовищі. Гра розроблена для демонстрації роботи алгоритмів детермінованої генерації на основі шумів, а також реалізації механізмів мережевої синхронізації між клієнтами. Розробка створюється в межах дослідження технологій процедурної генерації середовищ та взаємодії гравців у режимі реального часу. Особливістю проєкту є поєднання детермінованої генерації світу та мережевої складової на базі бібліотеки Mirror.

Підстави для розробки

Робота виконується на підставі кваліфікаційної роботи “Розробка та дослідження алгоритмів процедурної генерації ігрового світу та реалізація механізмів взаємодії між гравцями”.

Призначення розробки

Призначенням розробки є створення програмного продукту, що забезпечує автоматизовану генерацію унікальних світів, синхронізованих між усіма клієнтами гри. Програма реалізує можливість обміну даними в реальному часі, створення сесій, приєднання гравців, комунікації за допомогою текстового та голосового чату, а також спільної взаємодії у згенерованому середовищі.

Експлуатаційне призначення

Програмний продукт призначений для експлуатації на персональних комп’ютерах користувачів, об’єднаних у спільну мережу (локальну або

віртуальну). Гру можуть використовувати студенти, розробники або дослідники для тестування алгоритмів процедурної генерації та реалізації мережевої синхронізації.

Вимоги до програмного продукту

Програмне забезпечення повинне бути реалізоване в середовищі Unity з використанням мови C#. Програма має підтримувати створення і приєднання до багатокористувацьких сесій, відтворення процедурно згенерованих карт за однаковим seed на всіх клієнтах, а також забезпечувати стабільну роботу систем комунікації. Розробка повинна бути сумісна з основними операційними системами родини Windows і не потребувати додаткового встановлення сторонніх бібліотек окрім тих, що інтегровані в збірку гри.

Вимоги до функціональних характеристик

Програмний засіб повинен забезпечувати такі функції:

- створення та приєднання до ігрових сесій у межах локальної мережі;
- процедурну генерацію світу з використанням шумових алгоритмів;
- синхронізацію об’єктів між клієнтами за допомогою бібліотеки Mirror;
- керування персонажем (рух, взаємодія, використання предметів);
- текстову та голосову комунікацію між гравцями;
- відображення карти світу, інвентаря, панелі навичок і параметрів персонажа;
- можливість зміни налаштувань управління, звуку та відео в процесі гри.

Вимоги до надійності

Програма повинна стійко працювати у випадках нестабільного мережевого з’єднання та гарантувати збереження стану сесії при тимчасовій втраті клієнта. Програма має передбачати перевірку коректності даних під час обміну мережею, а також контроль невалідних або небезпечних дій користувача. При виникненні

критичних помилок створюється звіт через систему UnityCrashHandler, який може бути переданий розробнику.

Умови експлуатації

Для роботи програми необхідні:

- не менше двох пристроїв, з’єднаних у спільну мережу;
- затримка передачі даних у межах 100 мс;
- доступ до мікрофона (для голосового чату) та стабільне підключення до мережі.

Для налаштування ігрового процесу необхідно ввести ім’я гравця, створити або приєднатись до лобі та очікувати запуску гри хостом.

Вимоги до інформаційної та програмної сумісності

Гру розроблено на ігровому рушії Unity з використанням мови C# та у середовищі Microsoft VisualStudio.

Рекомендоване ПЗ та обладнання:

- операційна система – Windows 10 або вище;
- центральний процесор – Intel Core i5-4750HQ;
- оперативна пам’ять – 4 ГБ;
- відеокарта – Nvidia GeForce GT 930M.

Необхідні файли для коректної роботи програми перелічені в таблиці А.1.

Таблиця А.1.

Набір файлів для коректної роботи програми

№	Файл/папка	Призначення	Належить проекту
1	MonoBleedingEdge	Містить файли, необхідні для роботи Mono runtime, що дозволяє запускати гру на комп’ютерах без встановленого .NET Framework.	Двигун гри

№	Файл/папка	Призначення	Належить проекту
2	SuperTowerSurvival_Data	Містить всі дані та ресурси гри, такі як скрипти, текстури, моделі, сцени, звукові файли та інші необхідні файли.	Проект гри
3	SuperTowerSurvival.exe	Виконуваний файл гри.	Проект гри
4	UnityCrashHandler64.exe	Застосовується для збору інформації про збої гри та створення звітів про помилки.	Двигун гри
5	UnityPlayer.dll	Динамічна бібліотека, що містить основний код ігрового рушія Unity, необхідний для запуску гри.	Двигун гри

Вимоги до програмної документації

Для користувачів і розробників мають бути розроблені такі документи:

- інструкція користувачу;
- опис роботи програми та архітектури;
- технічна документація для підтримки та розгортання проєкту.

Стадії та етапи розробки

Таблиця А.2.

Етапи розробки

Етап	Зміт роботи	Результат
1	Розробка технічного завдання	Формування вимог і специфікацій до проєкту
2	Проєктування архітектури системи та алгоритмів генерації	Блок-схеми, UML-діаграми, опис логіки
3	Вибір середовища та інструментів розробки	Unity, Visual Studio, Mirror

4	Реалізація програмного продукту	Ігрова збірка з реалізованими функціями
5	Тестування та налагодження	Виправлення помилок, перевірка стабільності
6	Документування та передача в експлуатацію	Повний комплект документації та збірка гри

Порядок контролю та приймання

Програмний продукт має бути перевірений на працездатність і стабільність шляхом:

- функціонального тестування;
- мережевого тестування (багатокористувацький режим);
- тестування продуктивності (FPS, навантаження процесора);

Додаток Б.

Діаграми послідовності взаємодії сервера та клієнтів

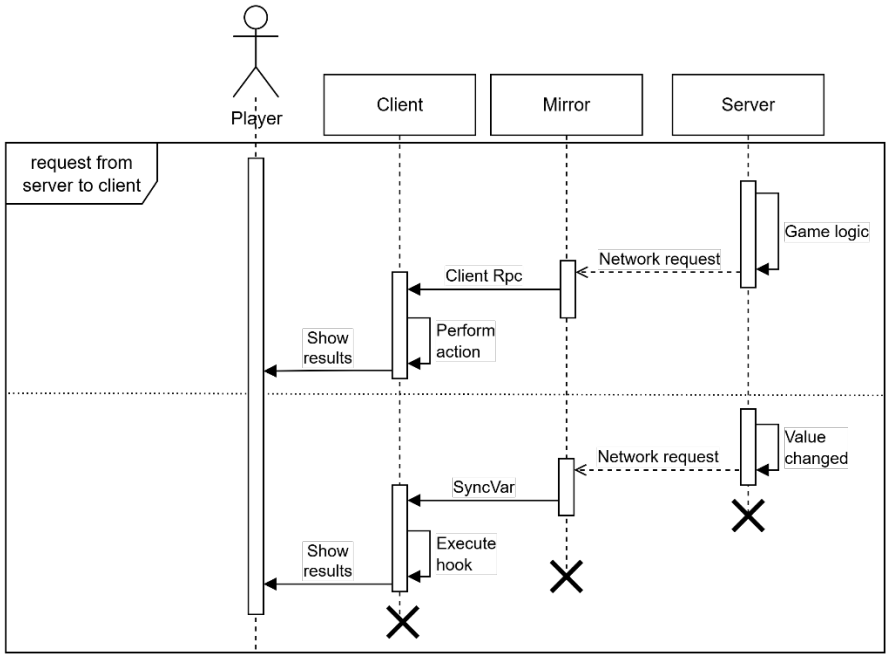

Рис. Б.1 – Запит сервера до клієнтів

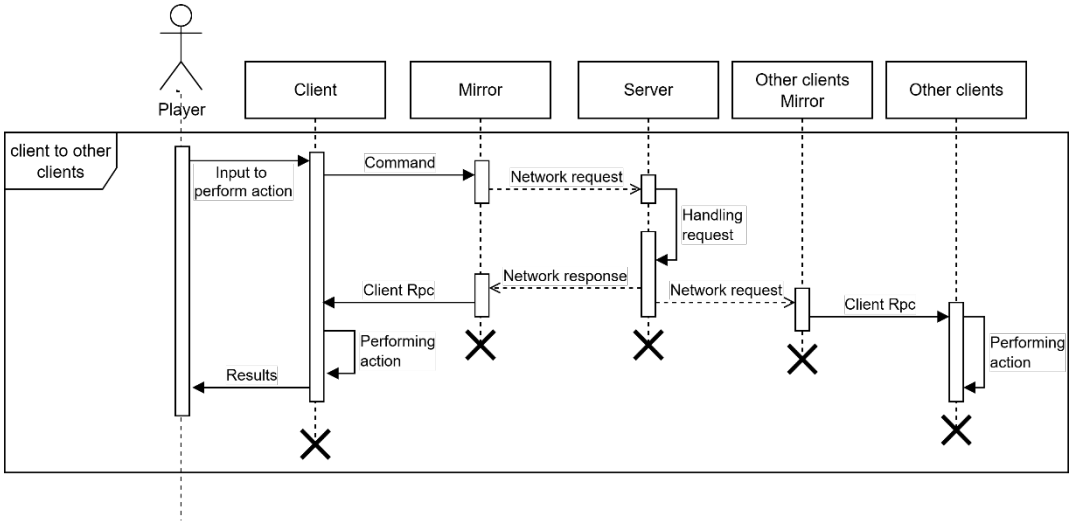

Рис. Б.2 – Запит клієнта до інших клієнтів

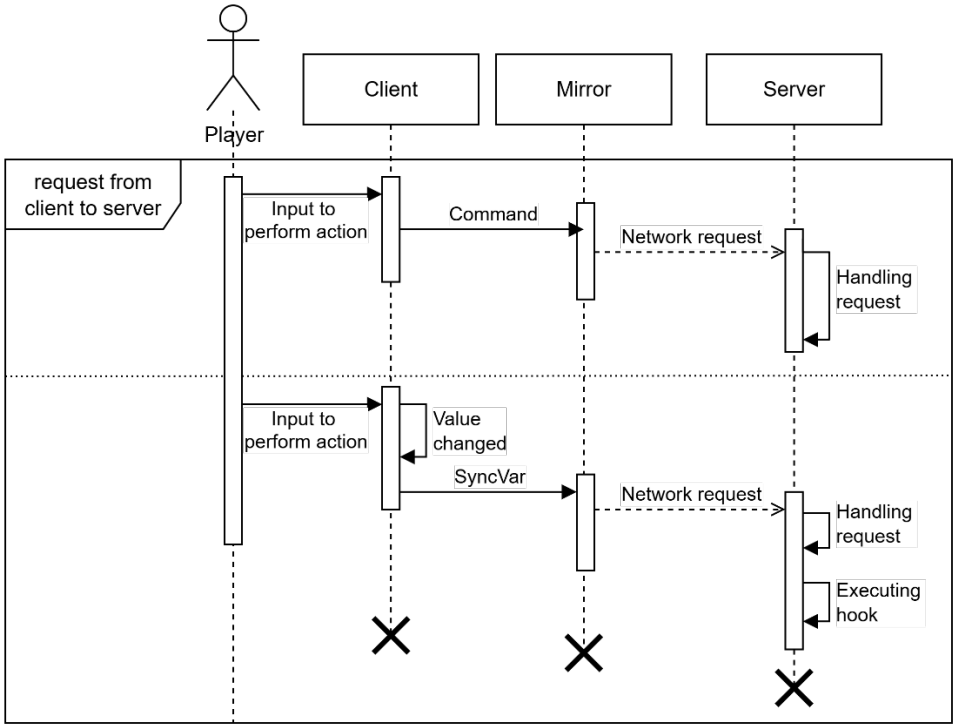

Рис. Б.3 – Запит клієнта до сервера

Додаток В.

Код класу NetworkManagerLobby

```
using Mirror;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class NetworkManagerLobby : NetworkManager
{
    [SerializeField] private int _minPlayers = 1;
    [Scene] [SerializeField] private string _menuScene = string.Empty;

    [Header("Room")] [SerializeField] private NetworkRoomPlayerLobby _roomPlayerLobbyPrefab = null;

    [Header("Game")] [SerializeField] private NetworkGamePlayer _gamePlayerPrefab = null;
    [SerializeField] private GameObject playerSpawnSystem = null;
    [SerializeField] private string _newSceneName = "Scene_Map_02";

    public static event Action OnClientConnected;
    public static event Action OnClientDisconnected;
    public static event Action<NetworkConnection> OnServerReadied;

    public List<NetworkRoomPlayerLobby> RoomPlayers { get; } = new List<NetworkRoomPlayerLobby>();
    public List<NetworkGamePlayer> GamePlayers { get; } = new List<NetworkGamePlayer>();

    public override void OnStartServer() => spawnPrefabs =
Resources.LoadAll<GameObject>("SpawnablePrefabs").ToList();

    private List<int> _chosenCharacters = new List<int>();

    public override void OnStartClient()
    {
        var spawnableObjects = Resources.LoadAll<GameObject>("SpawnablePrefabs");

        foreach (var prefab in spawnableObjects)
        {
            NetworkClient.RegisterPrefab(prefab);
        }
    }

    public override void OnClientConnect()
    {
        base.OnClientConnect();
        OnClientConnected?.Invoke();
    }

    public override void OnClientDisconnect()
    {
        base.OnClientDisconnect();
        SceneManager.LoadScene(offlineScene);
        OnClientDisconnected?.Invoke();
    }

    public override void OnServerConnect(NetworkConnectionToClient conn)
    {
        if (numPlayers >= maxConnections)
        {
            conn.Disconnect();
            return;
        }

        if (SceneManager.GetActiveScene().path != _menuScene)
        {
            conn.Disconnect();
            return;
        }
    }
```

```
public override void OnServerAddPlayer(NetworkConnectionToClient conn)
{
    if (SceneManager.GetActiveScene().path == _menuScene)
    {
        bool isLeader = RoomPlayers.Count == 0;
        NetworkRoomPlayerLobby roomPlayerInstance = Instantiate(_roomPlayerLobbyPrefab);

        roomPlayerInstance.IsLeader = isLeader;

        NetworkServer.AddPlayerForConnection(conn, roomPlayerInstance.gameObject);
    }
}

public override void OnServerDisconnect(NetworkConnectionToClient conn)
{
    if (conn.identity != null)
    {
        var player = conn.identity.GetComponent<NetworkRoomPlayerLobby>();

        RoomPlayers.Remove(player);

        NotifyPlayersOfReadyState();
    }

    base.OnServerDisconnect(conn);
}

public override void OnStopServer()
{
    RoomPlayers.Clear();
}

public void NotifyPlayersOfReadyState()
{
    foreach (var player in RoomPlayers)
    {
        player.HandleReadyToStart(IsReadyToStart());
    }
}

private bool IsReadyToStart()
{
    if (numPlayers < _minPlayers) return false;

    foreach (var player in RoomPlayers)
    {
        if (!player.IsReady) return false;
    }

    return true;
}

public void StartGame()
{
    if (SceneManager.GetActiveScene().path == _menuScene)
    {
        if (!IsReadyToStart()) return;
        _chosenCharacters.AddRange(RoomPlayers.Select(x => x.CurrentCharacter));
        ServerChangeScene(_newSceneName);
    }
}

public override void ServerChangeScene(string newSceneName)
{
    if (SceneManager.GetActiveScene().path == _menuScene && newSceneName.StartsWith("Scene_Map"))
    {
        for (int i = RoomPlayers.Count - 1; i >= 0; i--)
        {
            var conn = RoomPlayers[i].connectionToClient;
            var gamePlayerIstance = Instantiate(_gamePlayerPrefab);

            gamePlayerIstance.SetDisplayName(RoomPlayers[i].DisplayName);

            NetworkServer.Destroy(conn.identity.gameObject);
            NetworkServer.ReplacePlayerForConnection(conn, gamePlayerIstance.gameObject);
        }
```

```
        }

        base.ServerChangeScene(newSceneName);
        // StartCoroutine(LoadSceneAsync());
    }

    //public override void OnClientSceneChanged()
    //{
    //    base.OnClientSceneChanged();
    //    StartCoroutine(LoadSceneAsync());
    //}

    IEnumerator LoadSceneAsync()
    {
        GameObject loadingScreen = GameObject.FindWithTag("LoadingScreen");
        loadingScreen.GetComponent<CanvasGroup>().alpha = 1;
        Image fill = loadingScreen.transform.GetChild(1).GetComponent<Image>();

        while (!loadingSceneAsync.isDone)
        {
            fill.fillAmount = loadingSceneAsync.progress;
            yield return null;
        }
    }

    public override void OnServerSceneChanged(string sceneName)
    {
        if (sceneName.StartsWith("Scene_Map"))
        {
            GameObject playerSpawnSystemInstance = Instantiate(playerSpawnSystem);

playerSpawnSystemInstance.GetComponent<PlayerSpawnSystem>().ChosenCharacters.AddRange(_chosenCharacters);
            NetworkServer.Spawn(playerSpawnSystemInstance);
            Debug.Log("spawned player spawn system");
        }
    }

    public override void OnServerReady(NetworkConnectionToClient conn)
    {
        base.OnServerReady(conn);

        OnServerReadied?.Invoke(conn);
        Debug.Log("on server ready");
    }
}
```

Додаток Г.

Код класу NetworkRoomPlayerLobby

```
using JetBrains.Annotations;
using Mirror;
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class NetworkRoomPlayerLobby : NetworkBehaviour
{
    [Header("UI")]
    [SerializeField] private GameObject lobbyUI = null;
    [SerializeField] private TMP_Text[] playerNameTexts = new TMP_Text[5];
    [SerializeField] private TMP_Text[] playerReadyTexts = new TMP_Text[5];
    [SerializeField] private Button startGameButton = null;
    [SerializeField] private Button readyButton = null;

    [SyncVar]
    public int CurrentCharacter = 0;

    [SerializeField] private List<Button> _choseCharacterButton = new List<Button>();

    [SyncVar(hook = nameof(HandleDisplayNameChanged))]
    public string DisplayName = "Loading...";
    [SyncVar(hook = nameof(HandleReadyStatusChanged))]
    public bool IsReady = false;

    private bool isLeader;

    public bool IsLeader
    {
        set
        {
            isLeader = value;
            startGameButton.gameObject.SetActive(true);
        }
    }

    private NetworkManagerLobby room;

    public NetworkManagerLobby Room
    {
        get
        {
            if (room != null) return room;

            return room = NetworkManager.singleton as NetworkManagerLobby;
        }
    }

    private void Update()
    {
        if (IsReady)
        {
            var colors = readyButton.colors;
            colors.normalColor = Color.red;
            colors.highlightedColor = Color.red;
            colors.selectedColor = Color.red;
            readyButton.colors = colors;
            readyButton.GetComponentInChildren<TMP_Text>().text = "Not Ready";
        }
        else
        {
            var colors = readyButton.colors;
            colors.normalColor = Color.green;
            colors.highlightedColor = Color.green;
            colors.selectedColor = Color.green;
            readyButton.colors = colors;
            readyButton.GetComponentInChildren<TMP_Text>().text = "Ready";
        }
```

```
}

public override void OnStartAuthority()
{
    CmdSetDisplayName(PlayerNameInput.DisplayName);
    lobbyUI.SetActive(true);
}

public override void OnStartClient()
{
    Room.RoomPlayers.Add(this);

    UpdateDisplay();
}

public override void OnStopClient()
{
    Room.RoomPlayers.Remove(this);

    UpdateDisplay();
}

public void HandleReadyStatusChanged(bool oldValue, bool newValue) => UpdateDisplay();
public void HandleDisplayNameChanged(string oldValue, string newValue) => UpdateDisplay();

private void UpdateDisplay()
{
    if (!isOwned)
    {
        foreach (var player in Room.RoomPlayers)
        {
            if (player.isOwned)
            {
                player.UpdateDisplay();
                break;
            }
        }

        return;
    }


    for (int i = 0; i < playerNameTexts.Length; i++)
    {
        playerNameTexts[i].text = "Waiting for player...";
        playerReadyTexts[i].text = string.Empty;
    }

    for (int i = 0; i < Room.RoomPlayers.Count; i++)
    {
        playerNameTexts[i].text = Room.RoomPlayers[i].DisplayName;
        playerReadyTexts[i].text = Room.RoomPlayers[i].IsReady ?
            "<color=green>Ready</color>" :
            "<color=red>Not Ready</color>";
    }
}

public void HandleReadyToStart(bool readyToStart)
{
    if (!isLeader) return;

    startGameButton.interactable = readyToStart;
}

[Command]
public void CmdSetDisplayName(string displayName)
{
    DisplayName = displayName;
}

[Command]
public void CmdReadyUp()
{
    IsReady = !IsReady;

    Room.NotifyPlayersOfReadyState();
```

```
    }

    [Command]
    public void CmdStartGame()
    {
        if (Room.RoomPlayers[0].connectionToClient != connectionToClient) return;

        Room.StartGame();
    }

    [Command]
    public void UpdateChosenCharacter(int chosenCharacter)
    {
        CurrentCharacter = chosenCharacter;
    }

    public void ChangeCharacter(int characterIndex)
    {
        if (!IsReady)
        {
            _choseCharacterButton[CurrentCharacter].GetComponent<Outline>().effectColor = Color.white;
            _choseCharacterButton[characterIndex].GetComponent<Outline>().effectColor = Color.green;
            UpdateChosenCharacter(characterIndex);
        }
    }

    public void Disconnect()
    {
        if (isServer)
        {
            FindObjectOfType<NetworkManagerLobby>().StopHost();
        }
        else
        {
            NetworkClient.Disconnect();
        }
    }
}
```

Додаток Д.

Код класу PlayerSpawnSystem

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

public class PlayerSpawnSystem : NetworkBehaviour
{
    [SerializeField] private List<GameObject> playerPrefabs = new List<GameObject>();

    [SyncVar]
    public List<int> ChosenCharacters = new List<int>();

    private static List<Transform> spawnPoints = new List<Transform>();

    private int nextIndex = 0;

    public static void AddSpawnPoint(Transform transform)
    {
        spawnPoints.Add(transform);

        spawnPoints = spawnPoints.OrderBy(x => x.GetSiblingIndex()).ToList();
        Debug.Log("add spawnpoint");
    }

    public static void RemoveSpawnPoint(Transform transform) => spawnPoints.Remove(transform);

    public override void OnStartServer() => NetworkManagerLobby.OnServerReadied += SpawnPlayer;

    [ServerCallback]

    private void OnDestroy() => NetworkManagerLobby.OnServerReadied -= SpawnPlayer;

    [Server]
    public void SpawnPlayer(NetworkConnection conn)
    {
        Transform spawnPoint = spawnPoints.ElementAtOrDefault(nextIndex);

        if (spawnPoint == null)
        {
            Debug.LogError($"Missing spawn point for player{nextIndex}");
            return;
        }

        GameObject playerIstance = Instantiate(playerPrefabs[ChosenCharacters[nextIndex]],
spawnPoints[nextIndex].position, Quaternion.identity);
        NetworkServer.Spawn(playerIstance, conn);

        if (nextIndex == ChosenCharacters.Count - 1)
        {
            //last player spawned
            StartWaiting();
            System.Random random = new();
            var gameInit = FindObjectOfType<GameInitializer>();
            var seed = random.Next(0, 10000);
            gameInit.GenerateMaps(seed);
            FinishWaiting();

            Debug.Log("generated maps with seed " + seed);
        }

        nextIndex++;
    }

    [ClientRpc]
    public void StartWaiting()
```

```
    {
        FindObjectOfType<GameInitializer>().ShowWaitingCanvas();
    }

    [ClientRpc]
    public void FinishWaiting()
    {
        FindObjectOfType<GameInitializer>().HideWaitingCanvas();
    }
}
```

Додаток Е.

Код класу Noise

```
public static class Noise
{
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, int seed, float scale, int octaves,
        float persistance, float lacunarity, Vector2 offset)
    {
        float[,] noiseMap = new float[mapWidth, mapHeight];

        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves];
        for (int i = 0; i < octaves; i++)
        {
            float offsetX = prng.Next(-100000, 100000) + offset.x;
            float offsetY = prng.Next(-100000, 100000) + offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if (scale <= 0)
        {
            scale = 0.0001f;
        }

        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        float halfWidth = mapWidth / 2f;
        float halfHeight = mapHeight / 2f;


        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;

                for (int i = 0; i < octaves; i++)
                {
                    float sampleX = (x - halfWidth) / scale * frequency + octaveOffsets[i].x;
                    float sampleY = (y - halfHeight) / scale * frequency + octaveOffsets[i].y;

                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
                    noiseHeight += perlinValue * amplitude;

                    amplitude *= persistance;
                    frequency *= lacunarity;
                }

                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                }
                else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                noiseMap[x, y] = noiseHeight;
            }
        }

        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }
```

```
        return noiseMap;
    }
}
```

Додаток Ж.

Код класу MapGenerator

```
public class MapGenerator : NetworkBehaviour
{
    public enum GenerationMode
    {
        Island,
        Resources
    };

    [SerializeField] private GenerationMode _generationMode;

    [SerializeField] private int _mapWidth;
    [SerializeField] private int _mapHeight;
    [SerializeField] private float _noiseScale;

    [SerializeField] private int _octaves;
    [SerializeField] [Range(0, 1)] private float persistance;
    [SerializeField] private float lacunarity;

    [SerializeField] private int seed;
    [SerializeField] private Vector2 offset;

    public bool AutoUpdate;

    [SerializeField] private Biome[] _regions;

    [Header("Island settings")] [SerializeField]
    private float _islandRadius;

    [SerializeField] private int _mainHallRequiredSpace;
    [SerializeField] private string _townHallSpawnBiome = "Plains";
    [SerializeField] private float _islandRadiusOffset;
    [SerializeField] private Tilemap _waterTilemap;

    [FormerlySerializedAs("_landTilemap")] [SerializeField]
    private Tilemap _mainTilemap;

    [Header("Resource settings")] [SerializeField]
    private MapGenerator _landMapGenerator;

    [SerializeField] private ObtainableProps[] _obtainablesData;

    [SerializeField] private Transform _testPos;

    public float[,] NoiseMap { get; private set; }

    private float[,] _temperatureMap;
    private float[] _randomColourValues = { 1, 0.99f, 0.98f, 0.97f, 0.96f, 0.95f, 0.94f };
    private MapDisplay _mapDisplay;
    private bool _townHallIsPlaced;

    public void GenerateMap(int seedFromPlayer)
    {
        if (!isServer)
        {
            return;
        }

        _mainTilemap.ClearAllTiles();

        switch (_generationMode)
        {
            case GenerationMode.Island:
                NoiseMap = Noise.GenerateNoiseMap(_mapWidth, _mapHeight, seedFromPlayer, _noiseScale,
_octaves,
                    persistance,
                    lacunarity, offset);
                TryPlaceMainHall();
                CutIsland(NoiseMap);
```

```
                PlaceTiles(seedFromPlayer);
                break;
            case GenerationMode.Resources:
                NoiseMap = Noise.GenerateNoiseMap(_mapWidth, _mapHeight, seedFromPlayer, _noiseScale,
_octaves,
                    persistance,
                    lacunarity, offset);
                _landMapGenerator = FindObjectsOfType<MapGenerator>().First(o => o.name.Equals("Island Map
Generator"));
                PlaceObtainables();
                break;
        }
    }

    private void PlaceObtainables()
    {
        var _placedObtainables = new List<Transform>();

        for (int y = 0; y < _mapHeight; y++)
        {
            for (int x = 0; x < _mapWidth; x++)
            {
                float heightOnLand = _landMapGenerator.NoiseMap[x, y];

                for (int i = 0; i < _obtainablesData.Length; i++)
                {
                    if (!(heightOnLand >= _obtainablesData[i].MinHeight) ||
                        !(heightOnLand <= _obtainablesData[i].MaxHeight))
                    {
                        continue;
                    }

                    float heightForResource = NoiseMap[x, y];
                    if (!(heightForResource >= _obtainablesData[i].MinHeight) ||
                        !(heightForResource <= _obtainablesData[i].MaxHeight))
                    {
                        continue;
                    }

                    var expectedPosition = new Vector3(x - _mapWidth / 2, y - _mapHeight / 2, 0);
                    var (closest, distanceToClosest) = _placedObtainables.Closest(expectedPosition);
                    if (closest is null || distanceToClosest >= _obtainablesData[i].DistanceThreshold)
                    {
                        var obtainable = Instantiate(_obtainablesData[i].ObtainablePrefab,
                            expectedPosition,
                            Quaternion.identity, _mainTilemap.transform);

                        NetworkServer.Spawn(obtainable.gameObject);
                        _placedObtainables.Add(obtainable.transform);
                        break;
                    }
                }
            }
        }
    }

    private void PlaceTiles(int seedFromPlayer)
    {
        Random random = new(seedFromPlayer);

        for (int y = 0; y < _mapHeight; y++)
        {
            for (int x = 0; x < _mapWidth; x++)
            {
                float currentHeight = NoiseMap[x, y];

                for (int i = 0; i < _regions.Length; i++)
                {
                    if (currentHeight > _regions[i].MinHeight && currentHeight <= _regions[i].MaxHeight)
                    {
                        var tilePosition = new Vector3Int(x - _mapWidth / 2, y - _mapHeight / 2);

                        if (IsWaterTile(i))
                        {
                            PlaceWaterTile(tilePosition, i);
                            break;
```

```
                    }

                    var index = random.Next(0, _randomColourValues.Length - 1);
                    var randomNumber = _randomColourValues[index];
                    PlaceTile(tilePosition, i, randomNumber);
                    break;
                }
            }
        }
    }
}

public (int, int) WorldCoordsToNoiseArray(Vector2 worldPosition)
{
    var x = MathF.Truncate(worldPosition.x) + 250.0f;
    var y = -1.0f * (Math.Round(worldPosition.y / 10.0) * 10.0) + 250.0f;
    return ((int)x, (int)y);
}

public (int, int) WorldCoordsToNoiseArray(int worldX, int worldY)
{
    var x = MathF.Truncate(worldX) + _mapWidth / 2.0f;
    var y = -1.0f * (Math.Round(worldY / 10.0) * 10.0) + _mapHeight / 2.0f;
    return ((int)x, (int)y);
}

public string TileNameOnCoords(Vector2 world)
{
    var (x, y) = WorldCoordsToNoiseArray(world);
    var height = NoiseMap[x, y];
    for (int i = 0; i < _regions.Length; i++)
    {
        if (height > _regions[i].MinHeight && height <= _regions[i].MaxHeight)
        {
            return _regions[i].Name;
        }
    }

    return null;
}

private void TryPlaceMainHall()
{
    int initialX = 0, initialY = 0;
    int x = 0, y = 0;
    int yOffset = 0;

    for (int i = 0; i < _mapHeight; i++)
    {
        (x, y) = WorldCoordsToNoiseArray(initialX, initialY);
        initialY += yOffset;

        if (AreaIsSuitable(x, y, i))
        {
            break;
        }

        yOffset *= -1;
        yOffset += 1;
    }
    Debug.Log($"base found on x:{initialX} y:{initialY}");

    Vector3Int initialPos = new Vector3Int(initialX, initialY, 0);
    PlaceBase(initialPos);

    var spawnPoints = GameObject.FindGameObjectWithTag("SpawnPoints").transform;
    spawnPoints.position = initialPos;

    var characters = FindObjectsOfType<Character>();
    for (int j = 0; j < characters.Length; j++)
    {
        var child = spawnPoints.GetChild(j);
        MovePlayer(characters[j].netId, child.position);
    }

    _townHallIsPlaced = true;
```

```
    }

    private bool AreaIsSuitable(int x, int y, int i)
    {
        for (int j = x - _mainHallRequiredSpace; j < x + _mainHallRequiredSpace; j++)
        {
            for (int k = y - _mainHallRequiredSpace; k < y + _mainHallRequiredSpace; k++)
            {
                if (NoiseMap[j, k] < _regions[i].MinHeight || NoiseMap[j, k] > _regions[i].MaxHeight)
                {
                    return true;
                }
            }
        }

        return false;
    }

    [ClientRpc]
    private void PlaceBase(Vector3Int tilePosition)
    {
        var mainHall = GameObject.FindGameObjectWithTag("MainHall").transform;
        mainHall.position = tilePosition;
    }

    [ClientRpc]
    private void MovePlayer(uint netId, Vector3 newPosition)
    {
        var character = FindObjectsOfType<Character>().First(c => c.isOwned);

        if (character.netId == netId)
        {
            character.transform.SetPositionAndRotation(newPosition, quaternion.identity);
        }
    }

    [ClientRpc]
    private void PlaceWaterTile(Vector3Int tilePosition, int regionIndex)
    {
        _waterTilemap.SetTile(tilePosition, _regions[regionIndex].Tile);
    }

    [ClientRpc]
    private void PlaceTile(Vector3Int tilePosition, int regionIndex, float randomNumber)
    {
        _mainTilemap.SetTile(tilePosition, _regions[regionIndex].Tile);
        _mainTilemap.SetTileFlags(tilePosition, TileFlags.None);
        _mainTilemap.SetColor(tilePosition, new Color(randomNumber, randomNumber, randomNumber));
    }

    private bool IsWaterTile(int i)
    {
        return _regions[i].Name is "Sea" or "Ocean";
    }

    private void CutIsland(float[,] noiseMap)
    {
        Vector2 centerPoint = new Vector2(_mapWidth / 2f, _mapHeight / 2f);
        Vector2 currentPoint = new Vector2();

        for (int y = 0; y < _mapHeight; y++)
        {
            for (int x = 0; x < _mapWidth; x++)
            {
                currentPoint.x = x;
                currentPoint.y = y;

                var distanceFromCenterToPoint = Vector2.Distance(centerPoint, currentPoint);
                if (distanceFromCenterToPoint > _islandRadius)
                {
                    var multiplier = 1.0f - Mathf.InverseLerp(_islandRadius, _islandRadius +
_islandRadiusOffset,
                        distanceFromCenterToPoint);
                    noiseMap[x, y] *= multiplier;
                }
            }
```

```
        }
    }

    private bool IsLandAround()
    {
        for (int x = -1; x < 1; x++)
        {
            for (int y = -1; y < 1; y++)
            {
                if (x == 0 && y == 0)
                {
                    continue;
                }
                //check if height of one of xy tiles is suitable for land
                //if true, then create collider
            }
        }

        return false;
    }

    private void OnValidate()
    {
        if (_mapWidth < 1)
        {
            _mapWidth = 1;
        }

        if (_mapHeight < 1)
        {
            _mapHeight = 1;
        }

        if (lacunarity < 1)
        {
            lacunarity = 1;
        }

        if (_octaves < 0)
        {
            _octaves = 0;
        }
    }

    public void ClearMap()
    {
        FindObjectOfType<MapDisplay>().Tilemap.ClearAllTiles();
    }

    public void DefineTile()
    {
        TileNameOnCoords(_testPos.position);
    }
}
```

Додаток И.

Код класу ChatUI

```
using Mirror;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

public class ChatUI : MonoBehaviour
{
    public static ChatUI Instance;
    private const string PlayerPrefsNameKey = "PlayerName";

    [Header("UI References")]
    [SerializeField] private TMP_InputField inputField;
    [SerializeField] private TextMeshProUGUI logText;    // Multi-line text in a ScrollView
    [SerializeField] private ScrollRect scrollRect;
    [SerializeField] private Button sendButton;

    [Header("Player")]
    [SerializeField] private string playerName;

    public TMP_InputField InputField { get => inputField;}

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

        if (string.IsNullOrWhiteSpace(playerName))
            playerName = PlayerPrefs.GetString(PlayerPrefsNameKey);
    }

    private void Start()
    {
        if (sendButton != null)
            sendButton.onClick.AddListener(SendFromUI);

        // Submit on Enter
        if (inputField != null)
            inputField.onSubmit.AddListener(_ => SendFromUI());

        // If we joined late, render existing messages
        if (ChatNetwork.Instance != null)
            Refresh(ChatNetwork.Instance.messages);
    }

    public void Refresh(System.Collections.Generic.IList<string> messages)
    {
        if (logText == null) return;

        // Build log
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        for (int i = 0; i < messages.Count; i++)
        {
            sb.AppendLine(messages[i]);
        }
        logText.text = sb.ToString();

        // Auto-scroll to bottom next frame
        if (scrollRect != null)
            Canvas.ForceUpdateCanvases();
        if (scrollRect != null)
        {
            scrollRect.verticalNormalizedPosition = 0f;
            Canvas.ForceUpdateCanvases();
        }
    }
```

```
    private void SendFromUI()
    {
        if (inputField == null || string.IsNullOrWhiteSpace(inputField.text))
            return;

        if (ChatNetwork.Instance == null)
            return;

        // Only allow sending once connected as client or host
        if (!NetworkClient.active)
            return;

        ChatNetwork.Instance.CmdSendMessage(inputField.text, playerName);
        inputField.text = string.Empty;
        inputField.ActivateInputField();
        inputField.Select();
    }
}
```

Додаток К.

Код класу ChatNetwork

```
using Mirror;
using Mirror.Examples.Chat;
using System;
using UnityEngine;

public class ChatNetwork : NetworkBehaviour
{
    public static ChatNetwork Instance;

    public readonly SyncList<string> messages = new SyncList<string>();

    [Header("Limits")]
    [SerializeField] private int maxMessages = 200;

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

    public override void OnStartClient()
    {
        messages.Callback += OnMessagesChanged;
    }

    public override void OnStopClient()
    {
        messages.Callback -= OnMessagesChanged;
    }

    private void OnMessagesChanged(SyncList<string>.Operation op, int index, string oldItem, string newItem)
    {
        if (ChatUI.Instance != null)
            ChatUI.Instance.Refresh(messages);
    }

    [Command(requiresAuthority = false)]
    public void CmdSendMessage(string text, string sender)
    {
        if (string.IsNullOrWhiteSpace(text)) return;

        text = text.Trim();
        if (text.Length > 512) text = text.Substring(0, 512);

        var stamp = DateTime.Now.ToString("HH:mm");
        var final = $"[{stamp}] {sender}: {text}";

        if (messages.Count >= maxMessages)
            messages.RemoveAt(0);

        messages.Add(final);
    }
}
```

Додаток Л.

Код класу VoiceChatNetwork

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

[RequireComponent(typeof(AudioSource))]
public class VoiceChatNetwork : NetworkBehaviour
{
    [Header("Mic Settings")]
    public int sampleRate = 16000;      // lower sample rate to save bandwidth
    public int chunkSize = 1024;

    private AudioSource audioSource;
    private string micDevice;
    private bool isRecording = false;
    private AudioClip micClip;          // store the microphone clip
    private float[] micBuffer;
    private int lastPos = 0;

    private Queue<float> playbackQueue = new Queue<float>();

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        audioSource.loop = true;
        audioSource.playOnAwake = false;

        // Create a clip for remote playback
        AudioClip remoteClip = AudioClip.Create("RemoteVoice", 48000, 1, sampleRate, true, OnAudioRead);
        audioSource.clip = remoteClip;
        audioSource.Play();

        if (Microphone.devices.Length > 0)
            micDevice = Microphone.devices[0];
        else
            Debug.LogError("No microphone detected!");
    }

    void Update()
    {
        if (!isOwned) return;

        if (Input.GetKeyDown(KeyCode.V)) StartMic();
        if (Input.GetKeyUp(KeyCode.V)) StopMic();

        if (isRecording) SendMicData();
    }

    private void StartMic()
    {
        if (isRecording || micDevice == null) return;

        audioSource.mute = false;
        micClip = Microphone.Start(micDevice, true, 1, sampleRate);

        // Wait until microphone starts
        while (!(Microphone.GetPosition(micDevice) > 0)) { }

        micBuffer = new float[micClip.samples * micClip.channels];
        lastPos = 0;
        isRecording = true;
    }

    private void StopMic()
    {
        if (!isRecording) return;
```

```
        Microphone.End(micDevice);
        isRecording = false;
    }

    private void SendMicData()
    {
        if (micClip == null) return;

        int pos = Microphone.GetPosition(micDevice);
        if (pos < lastPos) pos += micBuffer.Length;

        int samplesToSend = pos - lastPos;
        if (samplesToSend <= 0) return;

        micClip.GetData(micBuffer, 0);

        int sent = 0;
        while (sent < samplesToSend)
        {
            int size = Mathf.Min(chunkSize, samplesToSend - sent);

            float[] chunk = new float[size];

            int bufferIndex = (lastPos + sent) % micBuffer.Length;
            int firstPart = Mathf.Min(size, micBuffer.Length - bufferIndex);
            int secondPart = size - firstPart;

            System.Array.Copy(micBuffer, bufferIndex, chunk, 0, firstPart);

            if (secondPart > 0)
            {
                System.Array.Copy(micBuffer, 0, chunk, firstPart, secondPart);
            }

            CmdSendAudioChunk(chunk);
            sent += size;
        }

        lastPos = pos % micBuffer.Length;
    }

    [Command]
    private void CmdSendAudioChunk(float[] chunk)
    {
        RpcReceiveAudioChunk(chunk);
    }

    [ClientRpc(includeOwner = false)]
    private void RpcReceiveAudioChunk(float[] chunk)
    {
        foreach (var f in chunk)
            playbackQueue.Enqueue(f);
    }

    private void OnAudioRead(float[] data)
    {
        for (int i = 0; i < data.Length; i++)
        {
            if (playbackQueue.Count > 0)
                data[i] = playbackQueue.Dequeue();
            else
                data[i] = 0f;
        }
    }
}
```

Додаток М.

Інструкція користувача

Л.1. Загальні відомості

Позначення і найменування програми: Super Tower Survival – багатокористувацька гра, розроблена на ігровому рушії Unity з використанням мови C# у середовищі Microsoft Visual Studio.

Л.2. Функціональне призначення

За допомогою програмного засобу користувач може створювати або приєднуватися до ігрових сесій, досліджувати світ, взаємодіяти з іншими гравцями, а також використовувати текстовий і голосовий чат для комунікації.

Л.3. Умови застосування програми

Для роботи потрібні файли збірки Unity: SuperTowerSurvival.exe, SuperTowerSurvival_Data, UnityPlayer.dll, MonoBleedingEdge. Гра не потребує додаткового встановлення баз даних чи вебсервера. Для гри через інтернет потрібне використання віддаленої локальної мережі

Встановлення ПЗ

Для встановлення гри необхідно помістити вміст архіву зі збіркою гри (папку «build») в окрему папку. Після цього розархівуйте файли та запустіть «SuperTowerSurvival.exe».

Налаштування ПЗ

Для зручності користувача в налаштуваннях проекту можна ознайомитися з клавішами управління за замовчуванням або, при необхідності, змінити їх.

4. Повідомлення оператору

У разі виникнення помилок гра може виводити повідомлення, що вказують на проблему:

- Connection failed – відсутнє з’єднання або неправильна IP-адреса;
- Network timeout – втрачено зв’язок із сервером;
- No microphone detected – не виявлено мікрофон;
- Missing game data – відсутні необхідні файли збірки.

У випадку появи таких повідомлень необхідно перевірити стабільність мережі, цілісність файлів гри або пристрої введення. Якщо проблему не вдається вирішити – надіслати звіт розробнику.

5. Опис роботи програми

Після запуску SuperTowerSurvival.exe користувач вводить ім’я гравця та натискає Confirm.

Далі можна:

- створити нову ігрову сесію (Host Lobby);
- приєднатися до вже створеної (Join Lobby) шляхом введення IP-адреси.

Після входу до лобі гравці обирають персонажа, натискають Ready, і коли всі готові — творець сесії запускає гру кнопкою Start Game.

Під час гри доступні такі основні дії:

- Рух – клавіші W, A, S, D;
- Відкрити/закрити мапу – M;
- Збільшити/зменшити мапу – колесо миші;
- Відкрити/закрити інвентар – I;
- Використати здібність – Q або E + ліва кнопка миші;
- Викликати меню налаштувань – Esc;
- Відкрити/закрити вікно текстового чату – T;
- Говорити до інших гравців – натиснути та тримати V;

У грі передбачено текстовий і голосовий чат для взаємодії між гравцями. Для завершення роботи необхідно відкрити меню (Esc) і вибрати Exit.

Додаток Н.

Тестові випадки

Таблиця Н.1.

CASE01. LobbyLocalHost

Summary:	Перевірка розгортання сервера в локальній мережі
Pre-conditions:	Запустити два застосунки гри в локальній мережі
Steps:	Expected results:
1) Перший застосунок створює нову сесію; 2) Другий застосунок приєднується до застосунку 1 за допомогою IPv4 адреси першого комп'ютера.	1) Гравець успішно приєднався до сесії.
Result: passed 1.	

Таблиця Н.2.

CASE02. PlayerLobbyStatus

Summary:	Перевірка відображення статусу гравця в лобі для очікування
Pre-conditions:	Два гравця знаходяться в лобі для очікування
Steps:	Expected results:
1) Гравець 1 ставить статус «"Ready" 2) Гравець 2 ставить статус "Ready"	1) Гравець 1 бачить результат зміни статусу готовності гравця 2; 2) Гравець 2 бачить результат зміни статусу готовності гравця; 3) Гравець 1 за умови готовності усіх гравців може розпочати гру.
Result: passed 3.	

Таблиця Н.3.

CASE03. SerializedPlayerPrefs

Summary:	Перевірка запам’ятовування попередньо введеного імені користувача
Pre-conditions:	Відсутні
Steps:	Expected results:
1) Запустити гру та ввести своє ім’я; 2) Розпочати нову гру; 3) Завершити застосунок; 4) Запустити застосунок знову.	1) При повторному вході у гру у полі для введення імені вже прописане попереднє ім’я.
Result: passed 1.	

Таблиця Н.4.

CASE04. ChoosePlayer

Summary:	Перевірка вибору персонажа лише при статусі гравця “Not Ready”
Pre-conditions:	Створити чи приєднатись до лобі для очікування
Steps:	Expected results:
1) Вибрати персонажа; 2) Поставити статус “Ready” 3) Повторити спробу вибору персонажа.	1) При статусу “Ready” гравець не повинен мати можливості вибрати іншого персонажа.
Result: passed 1.	

Таблиця Н.5.

CASE05. Генерація карти

Summary:	Перевірка генерації
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Натиснуто кнопку “Start game”.	1) Мапа генерується у формі острова (земля оточена водою); 2) Присутні усі регіони (океан, море, пляж, рівнини, ліс, темний ліс, кам’янисті землі); 3) Точка породження гравців разом з головною спорудою знаходяться у регіоні рівнин.
Result: passed 3.	

Таблиця Н.6.

CASE06. Генерація колізій

Summary:	Перевірка генерації – колізії
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Знайти регіон пляжу; 2) Підійти впритул до води і спробувати йти в напрямку моря.	1) Гравець впреться у воду і не зможе йти по ній.
Result: passed 1.	

Таблиця Н.7.

CASE07. Надсилання повідомлення з хоста

Summary:	Перевірка отримання повідомлення всіх клієнтів від хоста.
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Відкрити текстовий чат на клавішу T, на машині хоста; 2) Надіслати повідомлення іншим користувачам.	1) Після надсилання усі клієнти отримали повідомлення.
Result: passed 1.	

Таблиця Н.8.

CASE08. Надсилання повідомлення з клієнта

Summary:	Перевірка отримання повідомлення всіх клієнтів та хоста від іншого клієнта.
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Відкрити текстовий чат на клавішу T, на машині клієнта; 2) Надіслати повідомлення іншим користувачам.	1) Після надсилання усі клієнти отримали повідомлення; 2) Після надсилання хост отримав повідомлення.
Result: passed 2.	

Таблиця Н.9.

CASE09. Зчитування голосу з мікрофона тавідтворення на локальній машині

Summary:	Перевірка коректного запису звуку з мікрофона на його відтворення на локальній машині
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Відкрити текстовий чат на клавішу V та почати говорити.	1) Чути особистий голос з невеликою затримкою.
Result: passed 1.	

Таблиця Н.10.

CASE10. Надсилання голосового повідомлення з хоста

Summary:	Перевірка отримання голосового повідомлення всіх клієнтів від хоста.
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Відкрити текстовий чат на клавішу V та почати говорити на машині хоста.	1) Усі клієнти чують голос з машини хоста з невеликою затримкою.
Result: passed 1.	

Таблиця Н.11.

CASE11. Надсилання повідомлення з клієнта

Summary:	Перевірка отримання голосового повідомлення всіх клієнтів та хоста від іншого клієнта.
Pre-conditions:	Гравцем розпочато гру
Steps:	Expected results:
1) Відкрити текстовий чат на клавішу V та почати говорити на машині клієнта.	1) Усі клієнти чують голос з машини клієнта з невеликою затримкою. 2) Хост чує голос з машини клієнта з невеликою затримкою.
Result: passed 2.	

ABSTRACT

Zdrok D.O. Development and research of algorithms for procedural generation of a game world and implementation of player interaction mechanisms. Manuscript.

Master's qualification paper for obtaining the educational degree "Master" in specialty 122 Computer Science. Lesya Ukrainka Volyn National University, Lutsk, 2025.

The work provides an overview and analysis of modern algorithms for procedural generation of game worlds, including noise-based methods (Perlin noise, FBM, Voronoi), grammar-based systems, simulation approaches, and mosaic techniques. Their characteristics, advantages, limitations, and hybrid models combining several approaches to enhance realism and diversity of virtual environments are examined. Particular attention is paid to tools for creating procedural landscapes, such as World Machine and MapMagic, and their application in game engines.

The research also examines technologies for developing multiplayer components in video games. Modern networking frameworks for Unity – Mirror, Photon, FishNet, Unity Netcode for GameObjects – are reviewed, describing their properties, advantages, and areas of application. The work also outlines network interaction architectures (client-server, peer-to-peer), state synchronization optimization methods, client consistency mechanisms, and techniques for organizing communication between players.

Special attention is devoted to the practical implementation of a procedural world generation algorithm based on a deterministic seed and biome system, as well as the creation of a functional multiplayer component using the Unity engine and Mirror framework. The thesis includes the development of an island map generation system, resource placement mechanisms, server-side data synchronization logic, implementation of text and voice chat, networking architecture for sessions, lobby system, and player spawn mechanisms. Comprehensive testing was conducted: functional, network, and long-term endurance testing, with results analyzed and presented.

The developed prototype demonstrates the ability to generate a synchronized world for all clients, ensures stable operation of network mechanisms, and provides effective player interaction in a shared game environment. The obtained results may be applied in the development of multiplayer games with procedurally generated maps and dynamic interaction systems.

Keywords: procedural generation, Perlin noise, biomes, simulation algorithms, multiplayer games, networking frameworks, Unity, Mirror, client–server, synchronization, text chat, voice chat.

АНОТАЦІЯ

Здрок Д.О. Розробка та дослідження алгоритмів процедурної генерації ігрового світу та реалізація механізмів взаємодії між гравцями. Рукопис.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки. Волинський національний університет імені Лесі Українки, Луцьк, 2025 р.

У роботі проведено огляд та аналіз сучасних алгоритмів процедурної генерації ігрових світів, зокрема методів на основі шумових функцій (Perlin noise, FBM, Voronoi), граматичних систем, симуляційних підходів та мозаїчних технік. Розглянуто їхні особливості, переваги й обмеження, а також проаналізовано гібридні моделі, що поєднують кілька підходів для підвищення реалістичності та різноманітності віртуальних середовищ. Окрему увагу приділено інструментам для створення процедурних ландшафтів, таким як World Machine та MapMagic, та їх застосуванню в ігрових рушіях.

Також досліджено технології розробки багатокористувацьких компонент у відеоіграх. Розглянуто сучасні мережеві фреймворки для Unity – Mirror, Photon, FishNet, Unity Netcode for GameObjects – їхні особливості, переваги та області застосування. Описано архітектури побудови мережевої взаємодії (client-server, peer-to-peer), методи оптимізації синхронізації станів, забезпечення узгодженості клієнтів та способи організації комунікації між гравцями.

Особливу увагу приділено практичній реалізації алгоритму процедурної генерації світу на основі детермінованого seed та системи біомів, а також створенню функціональної мережевої частини гри з використанням рушія Unity та фреймворку Mirror. У роботі розроблено систему генерації карти острова, механізми розміщення ресурсів, серверну логіку синхронізації даних, реалізовано текстовий і голосовий чат, мережеву архітектуру сесій, систему лобі та механізми спавну гравців. Проведено комплексне тестування: функціональне, мережеве та довготривале Endurance-тестування, результати якого наведено та проаналізовано.

Розроблений прототип демонструє можливість узгодженого генерування світу для всіх клієнтів, стабільну роботу мережевих механізмів та ефективну взаємодію гравців у спільному ігровому середовищі. Отримані результати можуть бути використані у створенні багатокористувацьких ігор із процедурно згенерованими картами та динамічними системами взаємодії.

Ключові слова: процедурна генерація, шум Перліна, біоми, симуляційні алгоритми, багатокористувацькі ігри, мережеві фреймворки, Unity, Mirror, клієнт-сервер, синхронізація, текстовий чат, голосовий чат.