Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

РОЗРОБКА ГРИ НА ПЛАТФОРМІ UNITY З ВИКОРИСТАННЯМ ГЕНЕРАТИВНО-ЗМАГАЛЬНИХ МЕРЕЖ
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-21-2
Ласкавий О. С.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Шафроненко А.Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ласкавому Олександру Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка гри на платформі Unity з використанням генеративно-змагальних мереж

затверджена наказом університету від 19 травня 2025 року № 381Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 2 червня 2025 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд методів генерації оточення та побудови ігрового процесу.
2. Аналіз теоретичних основ генеративно-змагальних мереж, їх архітектури та застосувань у процедурній генерації контенту для ігор.
3. Розробка та імплементація ігрового проєкту на платформі Unity з інтеграцією GAN для автоматичної генерації ігрових карт і контенту.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми обробки зображень, постановка задачі, тестові зображення.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	07.04.2025	
2	Аналіз завдання, підбір літератури	08.04.25-10.04.25	
3	Аналіз літератури з досліджуваної проблеми	11.04.25-14.04.25	
4	Аналіз технічних засобів	15.04.25-20.04.25	
5	Розробка методу	21.04.25-27.04.25	
6	Програмна реалізація	28.04.25-11.05.25	
7	Оформлення пояснювальної записки	12.05.25-20.05.25	
8	Перевірка на нормоконтроль	21.05.25-01.06.25	
9	Перевірка на плагіат	21.05.25-01.06.25	
10	Рецензування	21.05.25-01.06.25	
11	Підготовка презентації та доповіді	21.05.25-18.06.25	
12	Занесення роботи в електронний архів	02.06.25-18.06.25	
13	Попередній захист кваліфікаційної роботи	02.06.25-18.06.25	

Дата видачі завдання 7 квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Шафроненко А.Ю.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 70 с., 3 табл., 27 рис., 31 джерело.

МАШИННЕ НАВЧАННЯ, ГЕНЕРАТИВНО-ЗМАГАЛЬНІ МЕРЕЖІ, ГЕНЕРАЦІЯ ОТОЧЕННЯ, UNITY, C#, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, СИСТЕМА ІГРОВИХ МЕХАНІК.

Об’єктом роботи є впровадження нейронних мереж у гру для генерації ігрового оточення.

Метою роботи є розробка методів генерації ігрового контенту із застосуванням технологій штучного інтелекту на основі генеративних нейронних мереж. Подальше заповнення ігрового оточення на основі цих методів сприяє підвищенню різноманітності ігрового процесу.

Використано методи машинного навчання для інтеграції генеративно-змагальних мереж у процес розробки гри. Застосовано шум Перліна як вхідні дані для навчання моделі з метою створення ландшафту місцевості. Розроблено алгоритм інтеграції нейронної моделі в Unity для автоматизованого створення унікальних ігрових локацій. Реалізовано методи процедурної генерації для заповнення оточення контентом та побудови системи ігрових механік.

У результаті здійснено програмну реалізацію гри на платформі Unity.

MACHINE LEARNING, GENERATIVE ADVERSARIAL NETWORKS, ENVIRONMENT GENERATION, UNITY, C#, PROCEDURAL GENERATION, GAME MECHANICS SYSTEM.

The object of the work is the implementation of neural networks in a game for the purpose of generating game environment.

The aim of the work is to develop methods for generating game content using artificial intelligence technologies based on generative neural networks. Further population of the game environment using these methods contributes to increased diversity of gameplay.

Machine learning methods were employed to integrate Generative Adversarial Networks into the game development process. Perlin noise was used as input data for training the model in order to generate terrain landscapes. An integration algorithm was developed to embed the neural model into Unity, enabling automated creation of unique game locations. Procedural generation methods were implemented to populate the environment with content and to construct the game mechanics system.

As a result, a game was developed on the Unity platform.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 7
Вступ 8
1 Аналіз методів генерації оточення та побудови ігрового процесу 9
1.1 Визначення процедурної генерації ігрового оточення 9
1.2 GAN як один з методів процедурної генерації 9
1.2.1 Генерація карт з використанням GAN 9
1.2.2 Порівняння з іншими методами генерації 11
1.3 Підходи до побудови ігрового процесу 13
1.3.1 Ігровий процес 13
1.3.2 Система ігрових механік 14
1.4 Ігри з відкритою взаємодією 15
1.4.1 Визначення ігор з відкритою взаємодією 15
1.4.2 Приклади ігор з відкритою взаємодією 16
1.5 Постановка задачі 19
2 Вибір технології генерації оточення та проєктування дизайну гри 21
2.1 Використання GAN для генерації базової структури карти 21
2.1.1 Архітектура GAN 21
2.1.2 GAN на основі шуму Перліна 26
2.1.3 Python 28
2.2 Вибір середовища розробки гри 29
2.2.1 Платформа Unity 29
2.2.2 C# та Visual Studio 31
2.3 Процедурна генерація та заповнення згенерованого оточення 32
2.4 Проєктування ігрових механік 33
2.5 Методи тестування та перевірки гри на працездатність 34
2.6 Узагальнення обраного інстументарію 35
3 Практична реалізація гри з використанням обраного інструментарію 37
3.1 Реалізація генерації ігрової карти 37

3.1.1 Створення набору даних 37
3.1.2 Реалізація GAN 43
3.1.3 Генерація зображення на основі навченої GAN 49
3.2 Використання згенерованої карти в Unity 51
3.2.1 Інтеграція генеративної мережі в Unity 51
3.2.2 Використання Tilemap для процедурного генерування 53
3.2.3 Заповнення ігрового середовища 55
3.3 Створення логіки ігрового процесу 57
3.3.1 Реалізація механік управління гравцем 57
3.3.2 Реалізація взаємодії з ігровим оточенням 59
3.3.3 Приклад застосованої оптимізації 63
3.4 Демонстрація роботи 64
Висновки 67
Перелік джерел посилання 68

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

GAN – Generative Adversarial Networks (генеративно-змагальні мережі)

UI – User Interface (користувальницький інтерфейс)

RGB – Red, Green, Blue (колірна модель, де колір утворюється шляхом змішування трьох основних кольорів: червоного, зеленого та синього)

CIELAB – Commission Internationale de l'Éclairage LAB (кольорова модель для точного опису кольорів так, щоб відображати сприйняття кольору людиною)

GPU – Graphics Processing Unit (графічний процесор)

PNG – Portable Network Graphics (формат зображень із стисненням без втрат)

JPEG – Joint Photographic Experts Group (формат стиснення зображень з втратами)

ВСТУП

У сучасному світі досягнення в сфері технологій відкривають багато нових можливостей, особливо в розробці інноваційних ігор. Однією з таких можливостей є поєднання нейронних мереж із ігровими платформами для створення динамічного та неповторного оточення. Мета даної роботи – це розробка гри на платформі Unity, що використовує генеративно-змагальні мережі для автоматичної генерації ігрового світу. Цей підхід дозволяє створювати унікальні локації та природні елементи, щоб забезпечити високу варіативність та більш глибоке занурення в ігровий процес.

Ігрове оточення грає важливу роль у відеоіграх через те, що воно не тільки є фоном, а й має не аби який вплив на ігровий процес. Водночас використання генеративних нейронних мереж автоматизує створення нових, унікальних локацій, що може підвищити можливість повторного ігрового досвіду і додає значної інтерактивності в гру.

Актуальність роботи полягає у застосуванні алгоритмів генерації ігрового світу для розробки відеоігор, що дозволяє створювати нові і унікальні сценарії та середовища для кожного проходження гри. Використання генеративно-змагальних мереж для автоматичного створення ландшафтів та природних елементів дозволяє вдосконалювати процес розробки ігор, в тому числі, забезпечуючи варіативність і непередбачуваність ігрового досвіду. Описаний підхід дозволяє розробникам створювати складні та оригінальні світи, що підвищують інтерес гравців і сприяють довготривалій взаємодії з грою.

Робота спрямована на розвиток методів автоматизованого створення ігрового контенту, що може бути застосовано не лише в ігровій індустрії, а й у суміжних галузях, таких як віртуальні світи, симуляції та інші інтерактивні середовища, які потребують варіативних даних.

1 АНАЛІЗ МЕТОДІВ ГЕНЕРАЦІЇ ОТОЧЕННЯ ТА ПОБУДОВИ ІГРОВОГО ПРОЦЕСУ

1.1 Визначення процедурної генерації ігрового оточення

Процедурна генерація карт – це процес автоматичного створення ігрових світів або рівнів за допомогою алгоритмів, основою яких є випадковість. Вона дозволяє створювати унікальні та непередбачувані світи, що змінюються при кожному перезапуску гри. Такий підхід економить ресурси розробників, оскільки дозволяє генерувати нескінченні варіанти карт без потреби вручну створювати кожен елемент.

На відміну від ручного наповнення, де кожна карта або рівень створюється дизайнером індивідуально, процедурна генерація забезпечує масштабованість та додає варіативність у гру при повторному проходженні. Ручне створення дозволяє досягти високого рівня деталізації та креативного контролю, але потребує значних часових витрат і людських ресурсів. Крім того, гравці швидко вивчають фіксовані світи, що знижує інтерес до повторного проходження.

Процедурна генерація також може динамічно адаптувати контент до дій гравця. Такий підхід особливо актуальний для ігор з відкритою взаємодією, стратегій та ігор з відкритим світом, де варіативність оточення відіграє важливу роль у глибині ігрового процесу [1, 2].

1.2 GAN як один із методів процедурної генерації

1.2.1 Генерація карт з використанням GAN

Генеративно-змагальні мережі – це вид нейронних мереж, що складається з двох компонентів: генератора та дискримінатора. Генератор створює нові зразки даних (у нашому випадку – карти), намагаючись зробити

їх максимально схожими на справжні. Дискримінатор, у свою чергу, намагається відрізнити «реальні» карти з навчального набору даних від «штучних», тих що створені генератором [3].

Процес тренування GAN показано на рисунку 1.1 . Він побудований на змаганні між мережами: генератор з кожним кроком краще створює підробки, а дискримінатор краще їх викриває. У результаті, після достатнього тренування, генератор може створювати нові, унікальні карти, які складно відрізнити від оригінальних зразків.

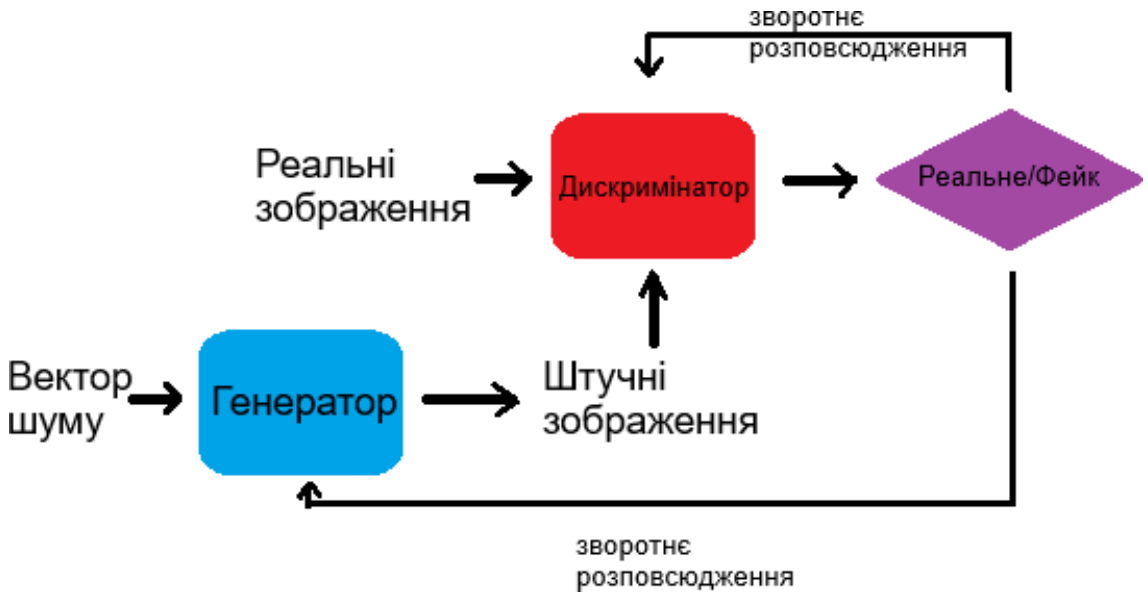


Рисунок 1.1 – Процес тренування GAN

Використання GAN для генерації ігрового оточення – це особливо цікавий підхід, тому що він дозволяє створювати контент, що схожий на «ручну» роботу дизайнерів, але цей процес більш автоматизований. GAN може навчитися стилю, логіці структури та просторовим закономірностям карт, якщо вони присутні в навчальному наборі, і перенести ці особливості на нові рівні без необхідності задавати жорсткі правила вручну.

1.2.2 Порівняння з іншими методами генерації

Генеративно-змагальні мережі мають значні відмінності від традиційних підходів генерації ігрових карт, таких як шум Перліна або кліткові автомати, що насамперед обумовлюється завдяки своїй здатності відтворювати стилістично складні та креативні структури без ручного програмування правил.

Шум Перліна – це популярна техніка, що дозволяє створювати плавні, природоподібні зміни висоти, температури або інших параметрів. Він особливо ефективний для генерації ландшафтів: гір, долин та водного простору. Хоча метод дозволяє генерувати правдоподібні карти, його математична передбачуваність обмежує різноманітність отриманих результатів. Приклад створення ігрової карти за допомогою шуму Перліна продемонстровано на рисунку 1.2. Вплив на кінцеву форму карти користувач має лише через параметри шуму, а не через зміст чи логіку розміщення об’єктів. Він не вміє враховувати концепцію дизайну рівня – лише форму [4].

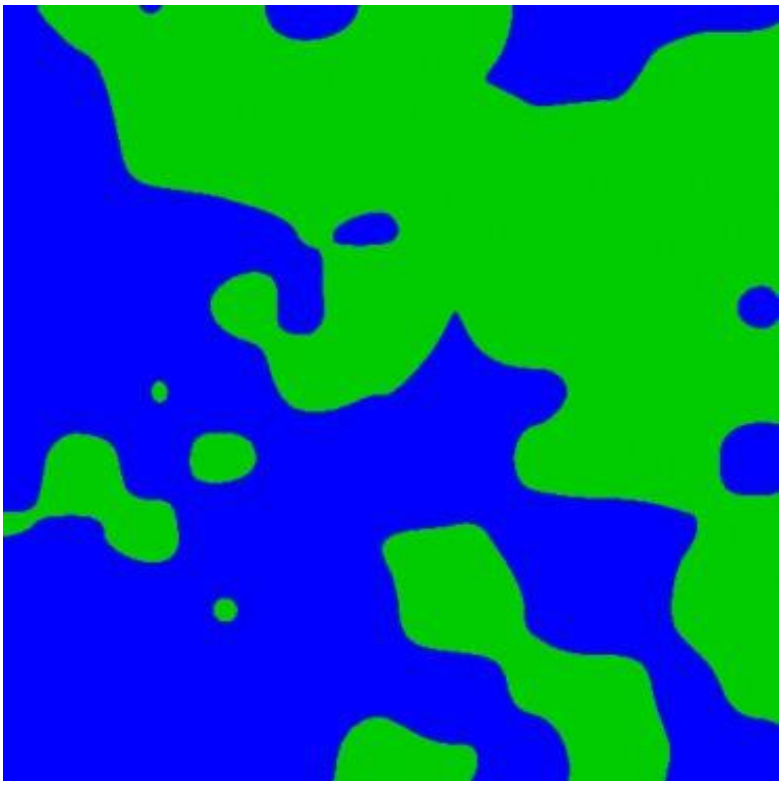


Рисунок 1.2 – Генерація карти за допомогою шуму Перліна

Кліткові автомати – ще один класичний метод, який часто використовують для генерації карт печер або лабіринтів. У ньому кожна клітинка на сітці «живе» або «вмирає» за простими правилами, що імітують природні структури. Як показано на рисунку 1.3, після декількох ітерацій карта набуває схожості на печеру або підземелля. Такий підхід ефективний для створення псевдореалістичних замкнених структур, але він обмежений у різноманітті та практично не адаптується до стилістичних особливостей дизайну чи цільової структури [5].

Крім того, кліткові автомати схильні до утворення ізольованих сегментів, які потребують додаткової обробки для забезпечення ігрової логіки. Вони не враховують такі аспекти, як наративне розміщення об’єктів або сценарії ігрового процесу. Через це результати часто вимагають додаткової обробки вручну або з використанням додаткових алгоритмів, що враховують контекст гри. Незважаючи на ці обмеження, їх простота та передбачуваність роблять кліткові автомати популярним вибором на ранніх етапах генерації рівнів.

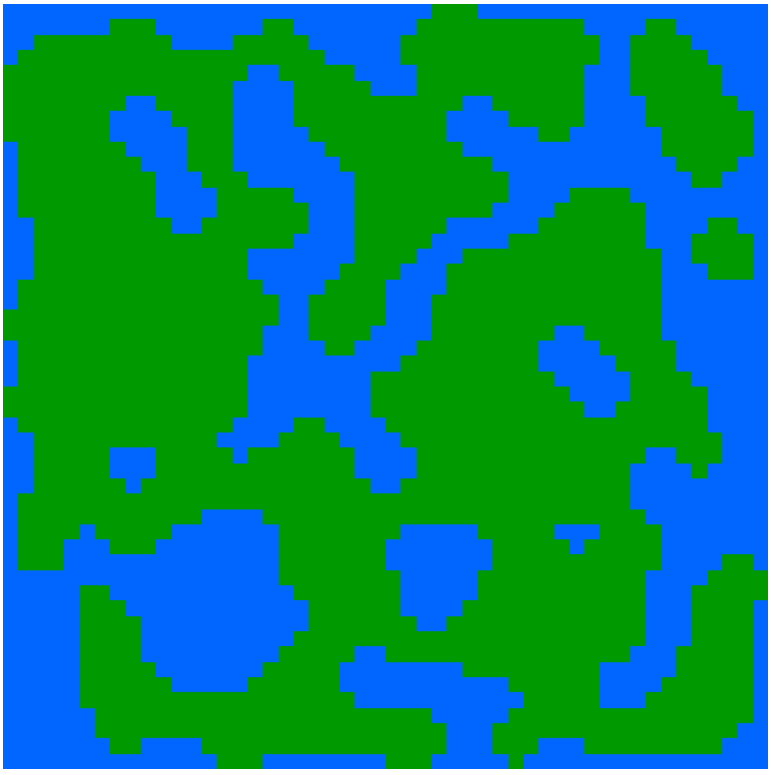


Рисунок 1.3 – Генерація карти на допомогою клітинних автоматів

На відміну від традиційних методів, GAN не потребують ручного прописування правил. Замість цього модель навчається безпосередньо на прикладах реальних карт або рівнів, самостійно вивчаючи логіку розміщення об’єктів, масштаб, стилістику й навіть баланс між елементами. Завдяки цьому результати виходять більш різноманітними, глибокими й візуально виразними, ніж ті, що створені за допомогою статичних алгоритмів. GAN здатна генерувати карти, які виглядають так, ніби їх проєктували професійні геймдизайнери – але без участі людини в процесі побудови.

Крім того, GAN – єдина з цих технологій, що може імітувати стиль. Наприклад, якщо мережу навчити на картах конкретної гри, вона зможе генерувати нові рівні, які не копіюють, а вільно стилізують контент у тому ж дусі. Це відкриває простір для автоматичної адаптації контенту під гру, гравця або навіть поточний настрій.

У результаті GAN перемагає в задачах, де потрібна семантика, художність, гнучкість і складна логіка, тоді як шум Перліна та кліткові автомати залишаються цінними для простих або структурно зрозумілих задач генерації.

1.3 Підходи по побудови ігрового процесу

1.3.1 Ігровий процес

Ігровий процес – це система взаємодій, механік, правил і дій, які визначають, як гравець взаємодіє з грою в цілому. Це охоплює всі аспекти гри, включаючи її основну мету, способи досягнення цієї мети, можливості гравця в межах гри, а також механіки, що керують розвитком подій у грі [6].

Загалом ігровий процес можна уявити як набір правил і можливостей, які створюють інтерфейс між гравцем і світом гри. Від того, як саме вибудований ігровий процес, залежить досвід і задоволення гравця: чи буде

це більше орієнтоване на розповідь, чи на стратегію, чи на дослідження, чи на вивчення нових механік.

Важливим аспектом побудови ігрового процесу для ігор з відкритою є надання гравцеві можливості для розвитку та дослідження, що, в свою чергу, створює відчуття постійного прогресу. Ігровий процес часто вимагає від гравця здатності адаптуватися до нових умов, що міняються з кожним перезапуском, адже елементи світу, як правило, змінюються – це стосується як локацій, так і розташування ворогів або предметів.

Гравець часто має справу з обмеженими ресурсами, які потрібно правильно використовувати, щоб досягти мети. Цей аспект забезпечує значну частину ігрового виклику для гравця, а також він зазвичай характеризується високою складністю та швидкими рішеннями, що утримують інтерес гравця, надаючи йому постійний стимул для досягнення нових результатів.

1.3.2 Система ігрових механік

Система механік в ігровому процесі визначає набір правил і механізмів, що керують поведінкою персонажа, навколишнім світом та взаємодією між ними [7, 8]. Вона охоплює всі аспекти ігрового процесу, від бойових систем до управління ресурсами, що також є основою для створення інтерфейсу взаємодії з гравцем. Вона забезпечує підтримку реалізму, балансу і викликів в грі, створюючи захоплюючий і динамічний досвід.

Одним із важливих елементів є механіка життя, яка включає показники здоров’я, енергії та емоційного стану персонажа. Вона визначає, як персонаж може стійко реагувати на стрес, поранення або інші виклики в світі. Бойова система є ще однією ключовою механікою, яка може варіюватися від тактичних стратегій до швидких рефлексивних атак, в залежності від стилю гри. Ресурси, як їжа, вода чи матеріали для виготовлення, також є важливою

частиною механіки, оскільки вони визначають виживання персонажа та розвиток сюжету.

1.4 Ігри з відкритою взаємодією

1.4.1 Визначення ігор з відкритою взаємодією

Ігри з відкритою взаємодією – це такі ігри, де гравець має свободу вибирати, що робити, в якому порядку і як досягати своїх цілей. На відміну від звичайних лінійних ігор зі строго прописаним сюжетом, тут ти сам досліджуєш світ, можеш змінювати його, спілкуватися з персонажами та взаємодіяти з різними об’єктами так, як хочеш. Головне те, що ти не просто слідуєш готовому сценарію, а створюєш власну історію і впливаєш на те, що відбувається у грі [9].

Такий підхід до ігрового процесу робить кожне проходження особливим. Гравець сам вирішує, чи піти досліджувати печеру, чи допомогти місцевим жителям, чи просто влаштувати хаос у віртуальному світі. Немає «правильного» чи «неправильного» шляху – усе залежить від вибору, інтересів і навіть настрою. Це не просто гра за сценарієм, а справжній світ, у якому можна жити, експериментувати і будувати власну історію.

У цьому й полягає головна привабливість ігор з відкритою взаємодією – вони не ставлять рамок. Тут можна грати у власному темпі: хтось годинами досліджує кожен куточок карти, збирає ресурси чи розмовляє з усіма неігровими персонажами, а хтось відразу кидається у вир подій. Такі ігри підлаштовуються під гравця, дають йому свободу бути ким завгодно – героєм, шукачем пригод, фермером чи навіть лиходієм. Це середовище, яке не змушує, а надихає на дію.

Таким чином, ігри з відкритою взаємодією поєднують свободу, варіативність та алгоритмічну складність, перетворюючи досвід гри на динамічну і неповторну пригоду, що формується у прямій залежності від

вибору гравця. Процедурна генерація в такому випадку стає не просто технічним інструментом, а основним рушієм геймплею.

1.4.2 Приклади ігор з відкритою взаємодією

У сучасній сфері розробки ігор процедурна генерація стала не просто технічним інструментом – вона стала частиною ігрового дизайну, яка формує фундамент геймплею. Це особливо яскраво проявляється в таких іграх з відкритою взаємодією, як Minecraft та Don’t Starve, де кожен світ – це не заздалегідь створена сцена, а результат алгоритмічного «творення» нової реальності, у якій гравець щоразу опиняється в новому контексті з новими викликами. Приклади карт обох ігор зображені на рисунках 1.4 та 1.5.

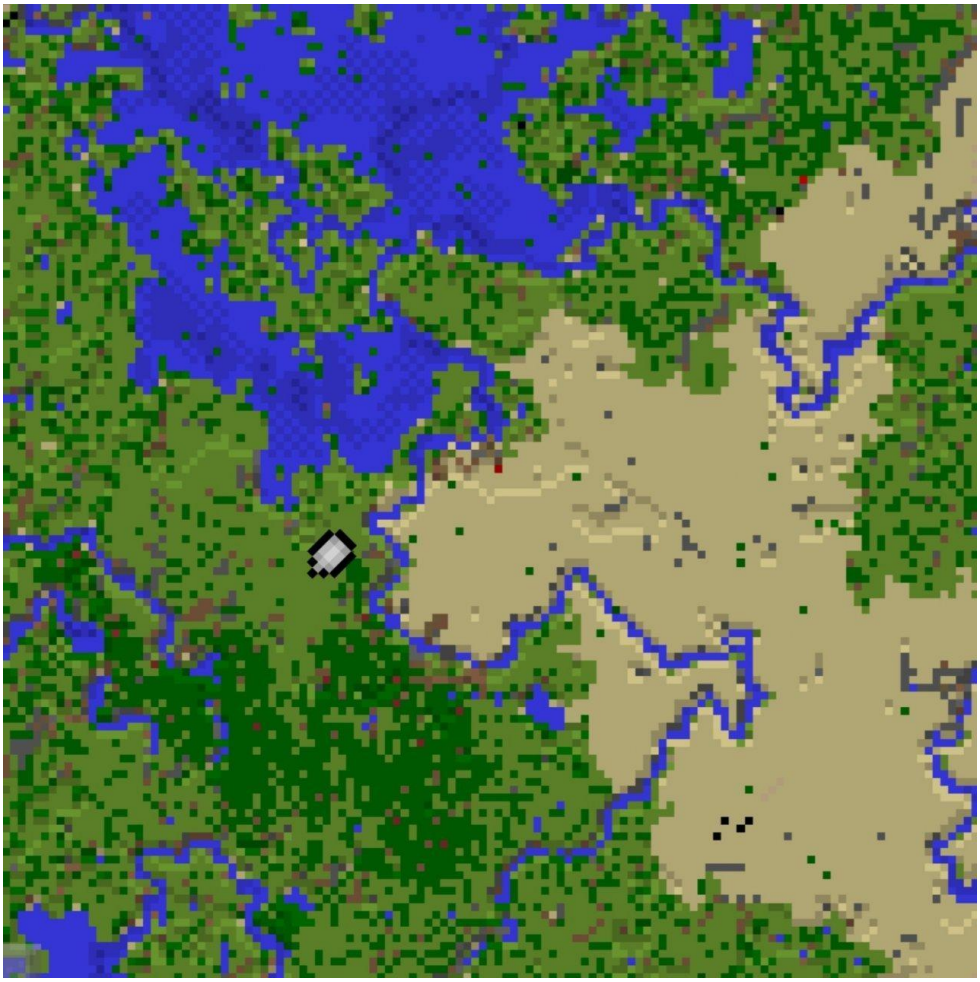


Рисунок 1.4 – Приклад згенерованої карти в Minecraft

Рисунок 1.5 – Приклад згенерованої карти в Don’t Starve

У Minecraft генерація світу – це центральна механіка, навколо якої побудована вся гра. Коли гравець створює новий світ, алгоритм на основі заданого або випадкового початкового значення генерації генерує нескінченну мозаїку з блоків 16×16, які формуються в реальному часі в міру дослідження. Для створення цих ландшафтів використовуються шум Перліна, Сімплекса та інші функції, що дозволяють імітувати природні форми – пагорби, долини, печери, океани. Всі ці форми накладаються шарами – наприклад, спершу формується рельєф, потім – тип природної зони, а далі – розміщення структур, таких як села, підземелля, бастіони або зруйновані портали. Процедурна генерація в Minecraft працює з великою кількістю змінних: температура, висота над рівнем моря, тип місцевості, щільність дерев чи розміщення руд. Усе це створює світ, який виглядає природно, хоча його ніколи не бачив дизайнер.

Доцільність цього підходу очевидна: гравець отримує нескінченний світ, який завжди цікавий для дослідження. Це дозволяє гравцям будувати будь-що будь-де, вивчати географію світу, шукати рідкісні ресурси, формувати власні бази в унікальних умовах. Саме це й робить Minecraft грою

без кінця – вона не вичерпується. Крім того, процедурна генерація робить гру універсальною для будь-якого стилю гри. При цьому кожен варіант можливий, бо світ – адаптивний і відкритий.

У Don’t Starve, на відміну від Minecraft, основний акцент зроблено не на будівництві, а на виживанні в агресивному світі. І тут процедурна генерація служить зовсім іншим цілям – створенню невизначеності та виклику. Кожна нова сесія – це новий світ, який має власну унікальну топологію: одні природні зони можуть бути ізольованими, інші – розкидані плямами, одні з великою кількістю їжі, інші – з ворогами. Гра генерує не тільки карту, а й розміщення ворогів, ресурсів, об’єктів інтересу, прихованих порталів і навіть підземних світів. Генерація враховує баланс між загрозами й винагородами, але навмисне не робить світ «справедливим». Іноді гравець може з`явитися поруч із болотом, повним ворожих істот, а іноді – біля ресурсів, які допомагають легше вижити перші дні.

Саме через таку «несправедливу» генерацію Don’t Starve підтримує свою основну ідею – випадковість як головна небезпека. Гравець ніколи не знає, який світ чекає наступного разу, і не може повністю покластися на пам’ять або шаблони дій. Кожна сесія вимагає адаптації. Механіки виживання, психічного здоров’я, зміни пір року, непередбачуваних подій (наприклад, атаки собак, землетруси чи пожежі) створюють глибоку систему, де кожне рішення – критичне. Процедурна генерація тут не просто додає різноманіття – вона створює гру, як таку, бо без неї кожна нова спроба перетворилася б на повторення попередньої.

Таким чином, обидві гри реалізують процедурну генерацію по-різному і з різною метою. У Minecraft вона відкриває перед гравцем нескінченне плато можливостей, на якому він може творити власний світ. У Don’t Starve – це метод створити загрозливе середовище, яке щосекунди кидає виклик. У першій – це гра про свободу. У другій – про обмеження, і те, як з ними впоратися. Але в обох випадках генерація – не другорядний інструмент, а

ядро формули ігрового процесу, яке дозволяє цим пісочницям не набридати навіть після сотень годин гри.

1.5 Постановка задачі

Таким чином, створення гри з автоматично згенерованим ігровим світом є актуальним завданням для дослідження процедурних методів генерації та сучасних підходів в розробці ігор. Особливої уваги заслуговує використання GAN для побудови карт, що базуються на наборі даних, з подальшим їх наповненням через класичні методи процедурної генерації. Це дозволяє поєднати переваги машинного навчання з традиційними ігровими алгоритмами.

Об’єктом дослідження є процес генерації ігрового середовища на основі GAN, процедурної генерації та підготовлених даних з вихідними локаціями.

Метою роботи є дослідження та реалізація методів генерації ігрових карт, які поєднують машинне навчання (через GAN), класичну процедурну генерацію та алгоритмічне додаткове оброблення [10], інтеграція цих рішень у систему побудови ігрового світу, придатну для використання в двовимірній грі та створення ігрових механік.

Для досягнення мети необхідно вирішити такі завдання:

- провести аналіз методів генерації карт, включаючи шум Перліна, шаблонну генерацію, а також GAN;
- дослідити способи побудови та штучного збагачення набору даних, які можуть бути використані для навчання генеративної моделі;
- реалізувати GAN-модель, яка здатна генерувати базову структуру карти на основі заданої палітри та топології;
- реалізувати етап додаткової обробки карти за допомогою процедурної генерації для заповнення об’єктами, зонами, подіями;

- забезпечити вивід результатів генерації графічно у вигляді ігрового світу;
- інтегрувати механізмів взаємодії з навколишнім світом, таких як фізика об’єктів, взаємодія з ресурсами та зміна середовища в результаті дій гравця;
- провести тестування та балансування ігрового досвіду;
- оптимізувати генерації карти для зменшення часу завантаження та покращення продуктивності, враховуючи обмеження апаратного забезпечення та вимоги гри.

2 ВИБІР ТЕХНОЛОГІЙ ГЕНЕРАЦІЇ ОТОЧЕННЯ ТА ПРОЄКТУВАННЯ ДИЗАЙНУ ГРИ

2.1 Використання GAN для генерації базової структури карти

2.1.1 Архітектура GAN

Архітектура GAN складається з двох основних компонентів: генератора та дискримінатора [11, 12]. Обидві нейронні мережі мають свої набори шарів, ваг і мету навчання, що взаємодіють один з одним у процесі змагання. Така взаємодія сприяє поступовому вдосконаленню генератора, що дозволяє йому створювати дедалі реалістичніші зображення або інші типи даних.

Генератор отримує на вхід випадковий вектор (шум) та перетворює його у зображення або інші типи даних. Метою генератора є створення таких даних, які виглядатимуть максимально схоже на реальні, аби обдурити дискримінатор. Процес навчання генератора базується на зворотному зв’язку від дискримінатора – чим краще генератор «обманює» дискримінатор, тим вищою є якість згенерованого контенту. У типовій архітектурі генератора застосовуються кілька шарів, зокрема згорткові або транспоновані згорткові, що поступово формують повноцінне зображення з початкового вектору випадкових чисел. Архітектура генератора зазвичай складається з кількох шарів, включаючи:

– вхідний шар, на якому генератор отримує випадковий вектор, який містить шуми. Він не містить жодної корисної інформації сам по собі, але є вихідним елементом для подальших перетворень;

– шари трансформування, що виконують перетворення вектора випадкових чисел у більш складні та детальніші представлення даних. Зазвичай для цього використовуються повністю з’єднані шари, що дозволяють відновлювати деякі необхідні характеристики для генерованих зображень;

– шари активації, що перетворюють отримані результати на функціональні дані. Функції активації [13], такі як ReLU або Leaky ReLU сприяють нелінійності, що необхідно для кращої генерації складних даних;

– вихідний шар − останній шар генератора відповідає за побудову фінального зображення. Для зображень це може бути шар з функцією активації tanh чи sigmoid, в залежності від необхідної шкали значень пікселів (наприклад, [0, 1] для sigmoid або [-1, 1] для tahn). На рисунках 2.1-2.4 зображені графіки використаних функцій активації.

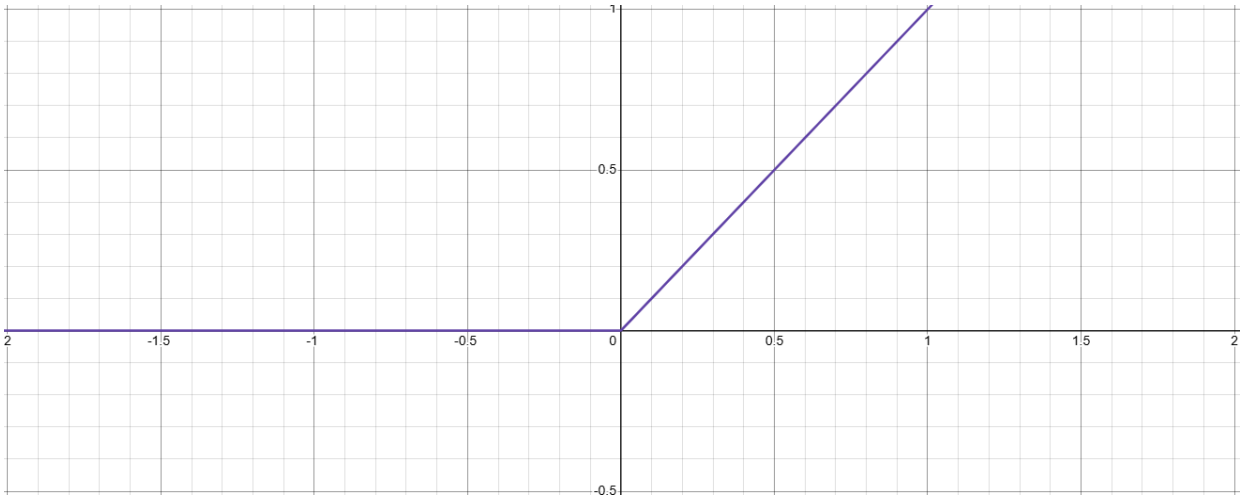


Рисунок 2.1 – Графік функції активації ReLU

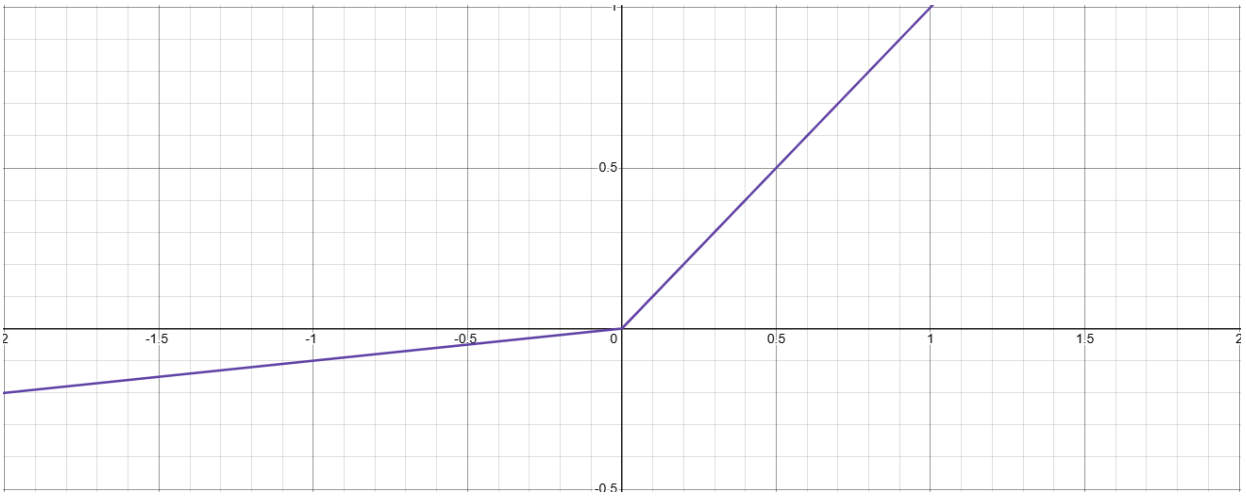


Рисунок 2.2 – Графік функції активації Leaky ReLU

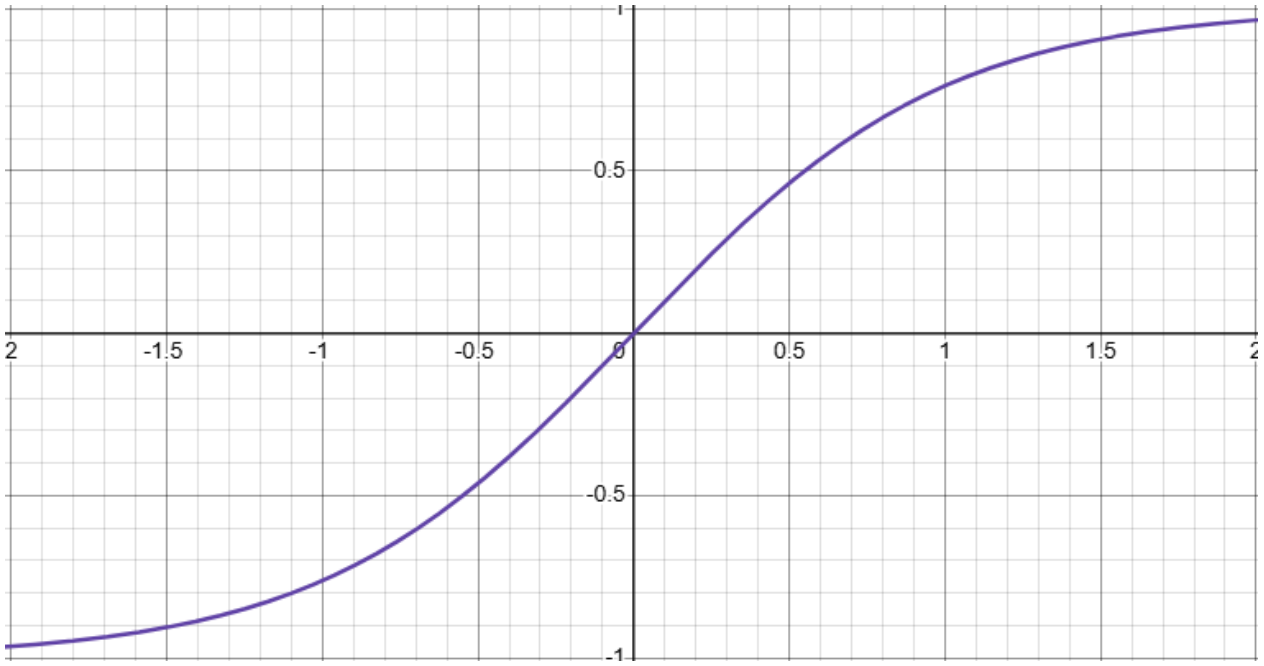


Рисунок 2.3 – Графік функції активації Than

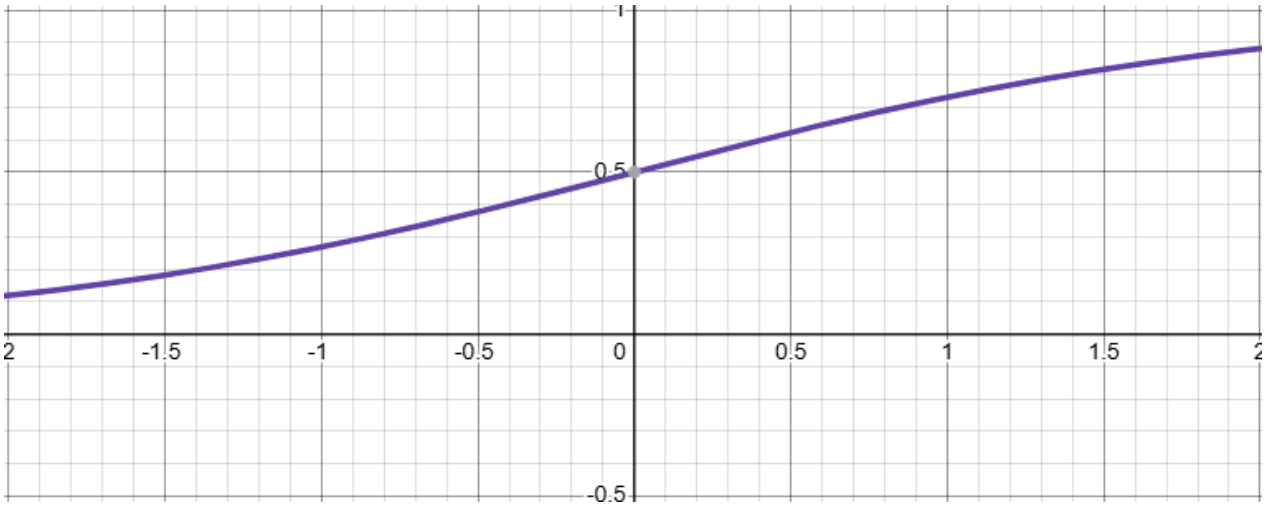


Рисунок 2.4 – Графік функції активації Sigmoid

Головна мета генератора – навчитися створювати такі зображення, які дискримінатор сприйматиме як справжні. Щоб цього досягти, він намагається мінімізувати функцію втрат, яка відображає, наскільки добре йому вдається вводити дискримінатор в оману. Зазвичай для цього використовують логарифмічні функції втрат або їхні варіації, які добре підходять для моделювання змагального процесу між мережами. Схема архітектури генератора зображена на рисунку 2.5.

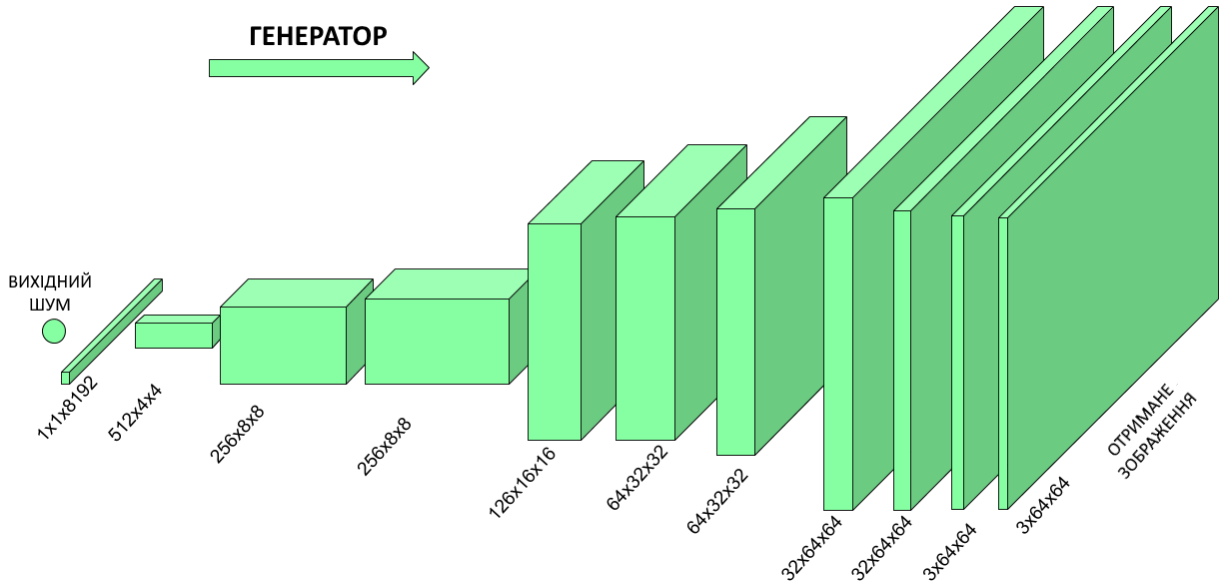


Рисунок 2.5 – Шари в архітектурі генератора

Дискримінатор – це теж нейронна мережа, завдання якої полягає у визначенні, чи є вхідні дані справжніми чи згенерованими. За своєю суттю, це задача бінарної класифікації: мережа повинна вирішити, чи зображення взято з реального набору даних, чи воно створене генератором. На вхід дискримінатор отримує зображення, представлене у вигляді вектора пікселів, яке далі обробляється через згорткові шари. Ці шари дозволяють виявляти ключові просторові ознаки – наприклад, контури, текстури та інші візуальні патерни. Як функції активації на цьому етапі зазвичай використовують ReLU або Leaky ReLU, які допомагають уникнути проблеми «мертвих нейронів» і забезпечують ефективну обробку навіть негативних значень. У фінальному шарі застосовується функція активації sigmoid – вона повертає значення від 0 до 1, що інтерпретується як ймовірність «справжності» зображення.

Дискримінатор вчиться мінімізувати свою помилку при класифікації зображень як реальних або згенерованих. Він прагне максимізувати ймовірність правильної класифікації, намагаючись уникнути помилок при

оцінці зображень генератора. Його вигляд дуже схожий на архітектуру генератора, але шари йдуть в зворотному порядку (рис. 2.6).

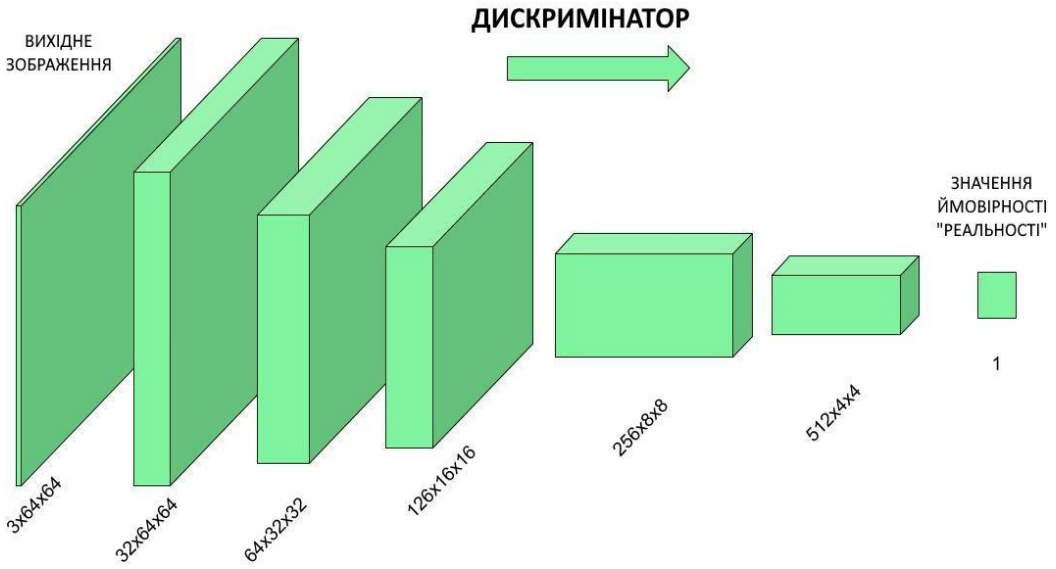


Рисунок 2.6 – Шари в архітектурі дискримінатора

Архітектура GAN базується на принципі змагання між двома мережами. Генератор намагається створити максимально правдоподібні зображення, а дискримінатор намагається правильно класифікувати їх як реальні чи згенеровані. Генератор отримує зворотний зв’язок від дискримінатора, і на основі цього коригує свої ваги, щоб покращити результати. Дискримінатор, у свою чергу, коригує свої ваги для поліпшення точності класифікації [14, 15].

Ваги генератора та дискримінатора визначають важливість кожного з елементів на всіх етапах навчання. Оскільки генератор намагається обдурити дискримінатор, ваги генератора мають критичне значення для того, щоб створювати зображення, схожі на реальні. Ваги дискримінатора відповідають за точність оцінки ймовірності реальності зображень, а також за здатність відрізняти згенеровані дані від реальних.

Навчання генератора і дискримінатора проходить в умовах постійного змагання: генератор намагається поліпшити свої результати, а дискримінатор – покращити свою здатність виявляти фальшиві зображення. Цей цикл оптимізації триває до тих пір, поки не досягнуто стану, коли генератор створює зображення, практично не відрізняються від реальних.

Архітектура GAN побудована на основі взаємодії двох основних компонентів: генератора та дискримінатора, кожен з яких має свій набір шарів і ваг, що впливають на ефективність мережі. Завдяки постійному змаганню між генератором і дискримінатором досягається висока якість генерованих зображень, що робить GAN потужним інструментом для процедурної генерації даних [16].

2.1.2 GAN на основі шуму Перліна

Використання GAN на основі шуму Перліна допомагає генерувати більш складні та різноманітні карти. Шум Перліна сам по собі є потужним інструментом для створення текстур і ландшафтів, але має свої обмеження. В порівнянні з ним, GAN дозволяє створювати значно більш складні структури, генерувати органічні переходи між різними природними зонами, а також отримувати більш детальні й природні результати. Однак, застосування GAN на основі шуму Перліна є ефективним підходом для генерації карт у порівнянні з використанням лише перлинного шуму [17].

Шум Перліна може створювати текстури та ландшафти, що виглядають натурально, однак, він інколи має схильність до однорідності і повторюваності, що знижує різноманітність карт. На відміну від цього, GAN дозволяє додавати варіативність і складність до генерації карт, поєднуючи її з шумом Перліна. Це дозволяє створювати карти, що виглядають органічніше, з детальними змінами у ландшафтах і переходами між природними зонами, що робить їх більш цікавими для дослідження.

Щоб краще уявити, як виглядає базовий шум Перліна, розгляньмо рисунок 2.7. Це дозволить зрозуміти, чому звичайний перлинний шум може виглядати однорідно, особливо коли мова йде про великі території. Як правило, зображення шуму Перліна виглядають як покриті більш-менш рівномірними шаблонами, що можуть бути схожі на природні ландшафти, але часто їх повторюваність створює певний візуальний ефект механічної повторюваності.

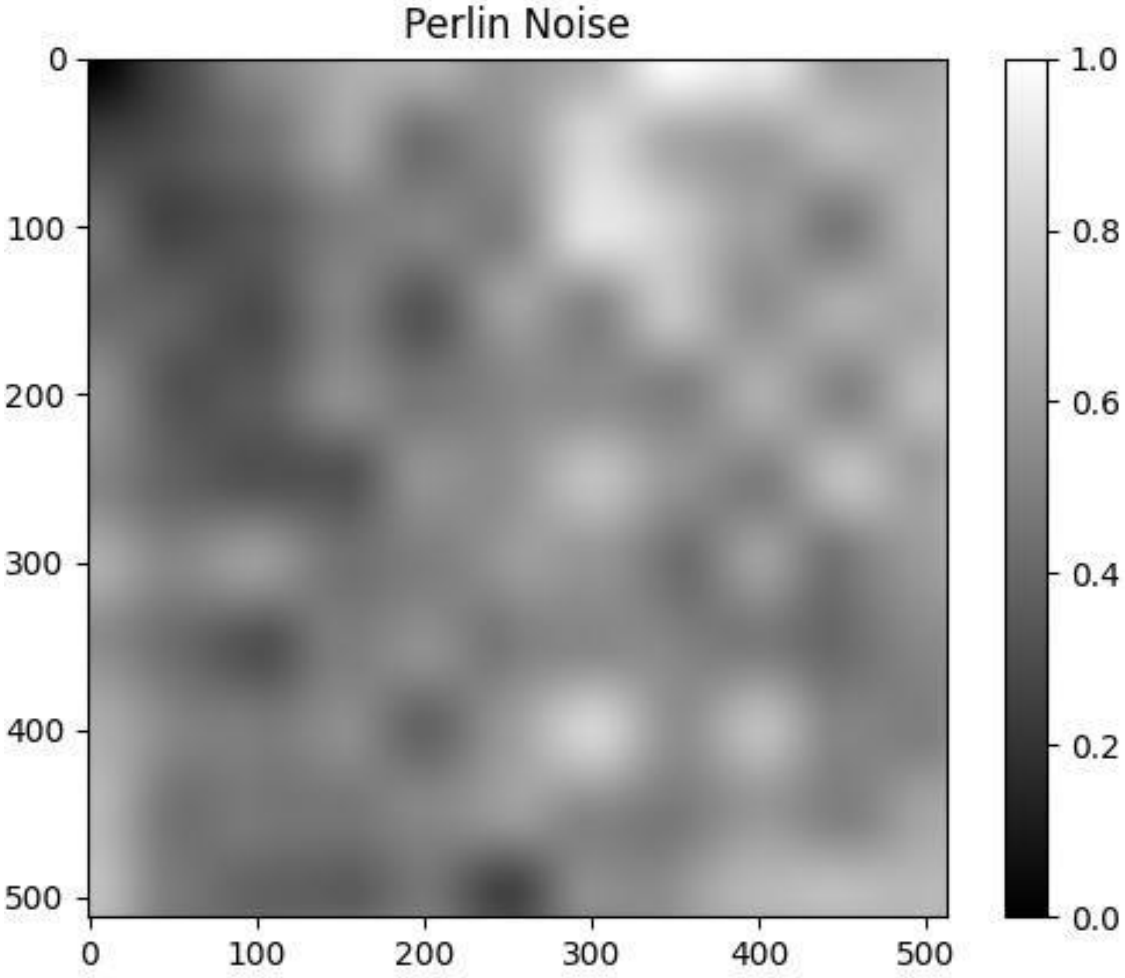


Рисунок 2.7 – Вигляд базового шуму Перліна

Завдяки GAN, ці обмеження можуть бути подолані, додаючи до карт більше органічних і складних переходів між природними зонами та

підвищуючи деталізацію, що робить світ виглядати більш живим і реалістичним.

Однією з основних переваг GAN є здатність генерувати плавні та природні переходи між різними частинами карти. Шум Перліна може створювати чудові текстури, але інколи ці текстури можуть виглядати непослідовно, особливо коли йдеться про зміни між зонами, що мають різні властивості, як, наприклад, між пустелею і лісом. Використання GAN дозволяє значно покращити ці переходи, забезпечуючи більш органічні зміни в ландшафтах.

Додатково, GAN може генерувати складні структури, такі як гірські хребти, річки, рівнини або інші природні утворення, поєднуючи їх з перлинним шумом для досягнення більш натуральних і детальних результатів. Це дозволяє не просто створювати базову текстуру, а й формувати багатошарові і складні карти, які відповідають специфічним вимогам гри чи проєкту.

Використання GAN на основі шуму Перліна дозволяє знизити необхідність в ручному налаштуванні параметрів, оскільки модель автоматично адаптується до найбільш природних шаблонів і варіацій для генерації карт. Це робить процес генерації карт значно ефективнішим і менш трудомістким, при цьому забезпечуючи високий рівень деталізації та різноманіття.

2.1.3 Python

Python є одним із найбільш популярних мов програмування для розробки моделей машинного навчання, зокрема для створення GAN. Це зумовлено його простотою, великим набором бібліотек та широкою спільнотою, що активно розвиває нові інструменти та підходи. Однією з основних переваг Python є підтримка таких бібліотек, як TensorFlow,

PyTorch, Keras, які значно спрощують розробку складних моделей глибокого навчання. Ці бібліотеки надають функціональність для побудови нейронних мереж, оптимізації моделей, роботи з великими наборами даних і багато іншого, що робить Python ідеальним вибором для задач, пов’язаних із генерацією даних за допомогою GAN.

Зокрема, для створення частини GAN за допомогою Python, основна роль належить бібліотекам для обробки та аналізу даних. PyTorch і TensorFlow надають інтуїтивно зрозумілий інтерфейс для побудови, тренування та оцінки нейронних мереж. Використання Python дозволяє зручно реалізувати архітектуру GAN, оскільки ця мова підтримує високу гнучкість при налаштуванні мереж і коригуванні параметрів, що є необхідним для точного налаштування таких складних моделей. Завдяки розвинутій екосистемі бібліотек та інструментів, Python забезпечує швидке моделювання та тестування різних підходів до генерації даних, що є важливим аспектом для досягнення високої ефективності моделей GAN.

2.2 Вибір середовища розробки гри

2.2.1 Платформа Unity

Unity є однією з найбільш популярних і широко використовуваних платформ для розробки відеоігор, завдяки своїй універсальності, доступності та потужному інструментарію для створення як двовимірних, так і тривимірних ігор. Однією з головних переваг Unity є її здатність підтримувати кросплатформність, що дозволяє розробникам створювати ігри, які працюватимуть на різних операційних системах та пристроях, таких як Windows, macOS, Android, iOS, PlayStation, Xbox і навіть VR/AR-пристроях. Ця можливість економить час і ресурси, оскільки розробник може створювати одну гру і випускати її на безліч платформи без необхідності переписувати код для кожної з них [18].

Однією з причин популярності Unity є її простота у використанні та навчанні. Завдяки підтримці C# як основної мови програмування, Unity дає змогу швидко освоїти основи програмування і почати створювати ігри, навіть для початківців. Мова C# є однією з найбільш популярних мов, що дозволяє легко інтегрувати сторонні бібліотеки, а також має потужні можливості для роботи з різноманітними даними та алгоритмами.

Порівнюючи Unity з іншими відомими ігровими движками, такими як Unreal Engine, можна виділити кілька переваг. Unreal Engine, хоч і є потужним і високопродуктивним інструментом для створення тривимірних ігор з високоякісною графікою, використовує C++ як основну мову програмування, що може бути складним для початківців. Крім того, Unreal має вищі системні вимоги, що може обмежити його використання на менш потужних комп’ютерах, в той час як Unity, як правило, має менш вимогливу до апаратних ресурсів архітектуру, що дозволяє працювати на більш широкому спектрі пристроїв. Водночас Unity має величезну кількість готових інструментів, активну спільноту та великий асортимент ресурсів в Asset Store, що дозволяє швидко інтегрувати нові функції та елементи в гру [19].

У порівнянні з іншими платформами, такими як Godot, Unity також виграє за рахунок широкої документації, підтримки великих команд розробників і постійних оновлень. Хоча Godot є легким і зручним для невеликих проєктів, Unity має набагато більшу кількість інструментів і ресурсів, що дозволяють розробляти як малі, так і великі ігри.

Отже, Unity є відмінним вибором для розробки ігор завдяки своїй простоті, гнучкості та кросплатформності, що дозволяє швидко реалізовувати проєкти будь-якої складності та випускати їх на безліч платформ.

2.2.2 C# та Visual Studio

C# є однією з найбільш популярних мов програмування в індустрії ігор, і саме її використовує Unity для написання скриптів. Однією з основних переваг C# є його зрозуміла синтаксична структура та потужні можливості для роботи з об’єктно-орієнтованим програмуванням. Це робить мову легкою для вивчення і використання, особливо для новачків, та одночасно достатньо гнучкою для досвідчених розробників. C# підтримує всі основні принципи ООП, такі як інкапсуляція, наслідування та поліморфізм, що дозволяє ефективно організовувати код і створювати складні ігрові системи.

Visual Studio є одним із найбільш популярних середовищ для розробки програмного забезпечення на C#, і є ідеальним інструментом для роботи з Unity. Visual Studio має потужний набір інструментів для налагодження коду, автозавершення, підсвітки синтаксису та рефакторингу, що значно прискорює розробку і дозволяє зменшити кількість помилок. Завдяки інтеграції з Unity, Visual Studio забезпечує зручний доступ до бібліотек і скриптів Unity, а також дозволяє безпосередньо запускати проєкти для тестування в редакторі Unity без необхідності перемикатися між кількома програмами.

Використання Visual Studio разом з C# при розробці в Unity є важливим, оскільки це дозволяє не лише підвищити продуктивність програміста, а й зменшити ймовірність помилок за рахунок автоперевірки коду, інтелектуальної підказки і інтеграції з системою управління версіями. Ці функціональності значно полегшують розробку ігор, особливо в командних проєктах, де важливо мати зручне середовище для співпраці. Завдяки цим інструментам, розробка ігор на Unity з використанням C# стає більш зручною і ефективною, що робить їх вибір оптимальним для створення сучасних інтерактивних ігор.

2.3 Процедурна генерація та заповнення згенерованого оточення

Після створення базової структури ігрової карти за допомогою генеративної змагальної мережі, розпочинається етап процедурного заповнення оточення відповідно до згенерованої карти. На цьому етапі система аналізує типи ландшафтів, що сформувалися в результаті роботи GAN – наприклад, ліси, рівнини, гори, водні об’єкти – та для кожної природної зони розміщує характерні для неї елементи. Наприклад, лісах можуть автоматично з’являтися дерева, кущі, гриби, каміння; на рівнинах – трава, дрібні рослини чи польові квіти; в гірських регіонах – скелі, кам’яні уламки, рідкісні руди тощо [20].

Це заповнення не є статичним – кожен об’єкт у світі пов’язаний із системою динамічних змін. Коли гравець взаємодіє з об’єктом, наприклад, зрубує дерево або знищує кущ, відбувається випадіння з нього відповідного ресурсу – дерева дають деревину, кущі – ягоди або листя. Після цього у зоні, де об’єкт був знищений, через деякий час або в певному радіусі з випадковим зміщенням генерується новий об’єкт, що підтримує баланс і забезпечує безперервність ресурсоутворення. Таким чином, гра підтримує власне середовище, яке адаптується до дій гравця, підвищуючи інтерес до повторного проходження та створюючи відчуття живого, органічного світу.

Ця система процедурного заповнення дозволяє не тільки економити ресурси розробки, бо немає потреби вручну розставляти кожен об’єкт, але й робить кожну сесію унікальною. Важливо також, що це дозволяє легко масштабувати світ гри, адаптувати його до нових сценаріїв, природних зон чи механік, без потреби переписувати логіку генерації з нуля.

2.4 Проєктування ігрових механік

У відкритих світах, де гравець може діяти вільно, базові механіки – це основа всього ігрового досвіду. Вони включають пересування по карті (ходьба, біг, стрибки, повзання), базову фізику (гравітація, зіткнення), взаємодію з об’єктами через компоненти зіткнення та тригери (підняття предметів, натискання кнопок, відкриття дверей), добування ресурсів (наприклад, зрубати дерево, добути камінь), бойову систему (удари, стрільба, отримання пошкоджень), а також системи базових потреб – голод і спрагу, які з часом знижуються і впливають на ефективність гравця. Усе це реалізовується через компоненти Unity: Rigidbody, Collider, скрипти на C#, анімації та взаємодію через події.

Додаткові, механіки розширюють і збагачують світ, роблячи його глибшим і більш реалістичним. Наприклад, зміна погоди може впливати на поведінку ворогів чи змінювати ефективність гравця (вночі зменшення видимості, під дощем – гасне вогонь). Економіка дозволяє створити систему торгівлі з NPC або іншими гравцями. Розвиток персонажа додає мотивацію досліджувати світ і виконувати завдання. Механіка карми дозволяє будувати нелінійні сюжетні розвилки в залежності від дій. Інші приклади – будівництво бази, приручення тварин, риболовля тощо. Усі ці механіки не є обов’язковими, але при правильному виконанні значно підвищення бажання до повторного проходження гри.

Щоб реалізувати все це, механіки будуються на модульній архітектурі: кожна система – окремий компонент, який взаємодіє з іншими через події, делегати чи інтерфейси. Наприклад, система потреб кожні кілька секунд зменшує рівень води та їжі, а якщо ці показники опускаються нижче порогу, передає команду до компоненту руху зменшити швидкість. Крафт працює через рецепти, що звіряються з наявністю предметів в інвентарі. Всі дані зберігаються в скриптових об’єктах або JSON-файлах для зручності оновлення. Взаємодія з оточенням реалізується через Raycast, OverlapSphere

або тригери, які визначають, який саме об’єкт ігровий персонаж бачить або торкає.

Розробка механік зазвичай відбувається поетапно. Спочатку створюється ядро: система переміщення, інвентар, UI для показу потреб і ресурсів, прості дії взаємодії [21]. Потім додаються залежності: голод впливає на швидкість або відновлення здоров’я, вода необхідна для виживання, дерево – основний ресурс для виготовлення предметів, який можна здобути лише певним інструментом. Згодом ці механіки з’єднуються в логічну екосистему, де дії гравця змінюють світ: наприклад, вирубав дерево – ресурс випав, за деякий час поряд проросте нове. На основі цього формується баланс, економіка та динаміка гри.

Такий підхід дозволяє не тільки створити цілісний ігровий цикл, а й масштабувати гру в майбутньому без необхідності переписувати код. Механіки взаємодіють, утворюючи органічний світ, де дії гравця мають прямі наслідки, а геймплей залишається свіжим завдяки численним варіантам розвитку подій.

2.5 Методи тестування та перевірки гри на працездатність

Інтеграційне тестування є критично важливим для перевірки взаємодії між окремими модулями гри. Наприклад, система голоду повинна правильно працювати із системою інвентарю: коли гравець вживає їжу, рівень голоду має змінюватись, а сама їжа зникати або зменшуватись у кількості. Якщо хоча б один з компонентів – наприклад, обробник предметів або статистики гравця – не синхронізований, це може спричинити логічні помилки. У рамках інтеграційного тестування це перевіряється як цілісний процес: від взаємодії з UI до оновлення значень у базовому контролері стану [22].

Ще один приклад – зміни в ігровому світі у відповідь на дії гравця. Якщо гравець зрубує дерево, воно має зникнути, залишити ресурси, і

викликати оновлення довкілля через певний час. Це перевіряється на рівні взаємодії між модулями генерації, обробки подій та розподілу ресурсів. Важливо протестувати, чи всі ці дії виконуються послідовно і без помилок: якщо, наприклад, ресурс не додається в інвентар, але дерево вже зникло – це вказує на збій у ланцюжку взаємодій. Завдяки інтеграційному тестуванню такі випадки виявляються до публічного релізу гри.

Тестування продуктивності дозволяє виявити і усунути вузькі місця, які можуть погіршити гравецький досвід, особливо у відкритому світі. Один із поширених прикладів – надмірне навантаження на процесор або графічну систему через занадто часте використання Update() для складних NPC-логік. У таких випадках за допомогою профайлерів, як-от Unity Profiler або Deep Profiling Mode, можна побачити, скільки ресурсів споживає кожен скрипт. Після оптимізації, наприклад, перенесення неважливих обчислень у рідше викликані події або кешування змінних, продуктивність помітно зростає.

Ще одна типова ситуація – сцена, перевантажена однаковими об’єктами, наприклад, деревами або камінням, які візуалізуютюся окремо. Без оптимізації, наприклад, через використання Static Batching, GPU Instancing або LOD Group, кадри в секунду можуть впасти навіть на потужних машинах. Після впровадження оптимізованих методів обробки віддалених об’єктів навантаження на GPU зменшується, а сцена виглядає плавно. Подібні перевірки особливо важливі, коли у грі присутня процедурна генерація, бо вона може створювати складні структури, які неочікувано впливають на продуктивність.

2.6 Узагальнення обраного інструментарію

Наведено загальний огляд обраного інструментарію, що застосовувався для досягнення цілей дослідження. Інформацію щодо характеристик та застосування інструментів представлено в таблиці 2.1.

Таблиця 2.1 – Етапи розробки та використаний інструментарій

Етап розробки	**Інструменти**	**Опис**
Генерація ігрової карти	VS Code, Python	Використання Python і бібліотек TensorFlow чи PyTorch для навчання GAN, що генерує ігрові карти. Кодування здійснюється у VS Code.
Заповнення згенерованого оточення	Unity, Visual Studio, C#	Заповнення оточення передбачає інтеграцію згенерованої карти в Unity, розміщення об’єктів гри за допомогою скриптів на C#.
Створення ігрового процесу	Unity, Visual Studio, C#	Розробка ігрових механік в Unity та C#, створення систем взаємодій, NPC та інтерактивних елементів.
Проєктування ігрових механік	Unity, Visual Studio, C#	Проєктування і реалізація механік у Unity, включаючи взаємодію персонажів, бої та управління ресурсами.
Проведення тестування	Unity, Visual Studio, C#	Проведення тестування в Unity для перевірки працездатності механік та оптимізації продуктивності. C# та Visual Studio для автоматичних тестів і перевірки взаємодій між системами.

Для реалізації та тестування концепції гри було використано інструментарій, включаючи Python, Unity, C# та Visual Studio. Кожен інструмент виконував свою функцію на етапах розробки – від генерації карти з використанням GAN до реалізації ігрових механік та тестування.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ГРИ З ВИКОРИСТАННЯМ ОБРАНОГО ІНСТРУМЕНТАРІЮ

3.1 Реалізація генерації ігрової карти

3.1.1 Створення набору даних

У цьому розділі детально розглянуто практичну реалізацію створення карти, яка складається з кількох взаємопов’язаних кроків: генерації висотної карти за допомогою шуму Перліна, класифікація пікселів за висотними зонами та деталізація рівнинної території відповідно до широтних поясів, очищення випадкових артефактів «самотніх» пікселів та збереження готового результату в набір даних. Такий підхід дозволяє гнучко моделювати ландшафт зі складною «острівною» топологією та різноманітними формами рельєфу [23, 24].

Загальна послідовність реалізації виглядає так:

Крок 1. Ініціалізація параметрів. Задаються розміри зображення, кількість ітерацій (кількість карт, що треба згенерувати), параметри мушу Перліна (масштаб, октави, persistence, lacunarity), а також кольорові мітки для природних зон.

Крок 2. Генерація висотної карти. З використанням параметрів шуму Перліна створюється шумова карта висот. Приклад методу її створення описано в лістингу 3.1. Для кожного пікселя зображення визначається значення висоти, яке формується на основі рівня шуму у відповідній точці. В результаті генерується двовимірна матриця значень, що представляє висоти для кожної точки карти.

Крок 3. Нормалізація та первинна класифікація. Після генерації висотна карта нормалізується до діапазону [-1, 1], щоб вирівняти мінімальні та максимальні значення. На основі цього виконується первинна класифікація природних зон: вода, пляж, ліс, гори тощо. Для кожної зони

задаються відповідні діапазони висот (наприклад, вода – від -1 до -0.2, гори – від 0.2 до 1), і кожен піксель отримує свою категорію.

Крок 4. Деталізація рівнинної зони. Зона рівнин (наприклад, висоти від -0.2 до 0.2) додатково деталізується за допомогою шуму Перліна з меншим масштабом. Це додає варіації висоти, створюючи ефект лугів, невеликих пагорбів або розломів, підвищуючи реалістичність рівнинної поверхні.

Крок 5. Очищення одиночних пікселів. Здійснюється фільтрація карти для видалення поодиноких пікселів, які не відповідають оточуючим зонам. Перевіряються сусідні клітини кожної точки, і якщо піксель значно відрізняється від оточення, він змінюється на домінуючий тип ландшафту навколо нього.

Крок 6. Побудова RGB-зображення та збереження. Згідно з класифікованими зонами формується RGB-зображення. Кожна зона отримує свій колір відповідно до встановлених параметрів. Готове зображення експортується у формат PNG або JPG та зберігається в заданій директорії.

Лістинг 3.1 Використання функції шуму Перліна для подальшого застосування:

```
def generate_noise_map(size, scale, octaves, persistence, lacunarity):
    base = np.random.randint(0, 100)
    x_offset, y_offset = np.random.uniform(0, 1000, 2)
    map_data = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            x = (i + x_offset) / scale
            y = (j + y_offset) / scale
            value = noise.pnoise2(x, y, octaves=octaves, persistence=persistence, lacunarity=lacunarity, repeatx=1024, repeaty=1024, base=base)
```

Опис параметрів шуму:

– scale – чим більше, тим більш «розтягнута» шумова карта;

– octaves – кількість шарів шуму: кожна наступна «октава» додає дрібніші деталі;

– persistence – зменшення амплітуди кожної октави;

– lacunarity – збільшення частоти для кожної наступної октави.

Щоб створити «острів», значення шуму посилюються ближче до центру: відстань до краю нормалізується, і за певних порогів до значення шуму додається додатковий «підсилювач». Приклад реалізації показано у лістингу 3.2.

Лістинг 3.2 Приклад створення ефекту острова:

```
        if distance < 0.3:
            edge_boost = (distance - 0.3) / 0.9
            value += edge_boost * 1.5
        if distance < 0.1:
            edge_boost = (distance - 0.1) / 0.9
            value += edge_boost * 1.5
        map_data[i][j] = value
    return map_data
```

«Острів» створюється для того щоб реалізувати природні границі ігрового світи та для їх нативного сприйняття гравцем.

Далі кожен піксель відноситься до одного з чотирьох основних природних зон: вода, пляж, рівнина, гори. Порогові значення обрано експериментально, щоб зазвичай отримати приблизно порівну з кожної зони. Це забезпечує різноманітний, але збалансований ландшафт.

Щоб уникнути одноманітності великої ділянки рівнини, застосовується окремий шар шуму з іншими параметрами. За допомогою цієї карти кожен піксель рівнинної зони класифікується в один із чотирьох природних поясів:

луки, ліс, поле, болото. Приклад генерації додаткових природних зон показано в лістингу 3.3.

Лістинг 3.3 Деталізація рівнинної зони на природні пояси:

```
def assign_forest_subbiomes(biome_map):
    subbiome_map = np.copy(biome_map)
    height, width = biome_map.shape
    base_noise = generate_noise_map(width, scale=30, octaves=15, persistence=0.4, lacunarity=2.0)
    for i in range(height):
        for j in range(width):
            if biome_map[i, j] != 2:
                continue
            value = base_noise[i, j]
            if value < -0.2:
                subbiome_map[i, j] = 4 # Луг
            elif value < 0:
                subbiome_map[i, j] = 5 # Поле
            elif value < 0.2:
                subbiome_map[i, j] = 6 # Густий ліс
            else:
                subbiome_map[i, j] = 7 # Болото
    return subbiome_map
```

Інформація про природні зони як вертикальної так і горизонтальної зональності зображено у таблиці 3.1. Колір кожної природної зони задається словником RGB-триплетів. За індексом природної зони витягується потрібний колір і формується масив color_map розміром. Цей масив можна безпосередньо конвертувати в зображення.

Таблиця 3.1 – Класифікація природних зон

Основні природні зони			**Додаткові природні пояси**		
Назва	**Границі значень**	**Колір, RGB**	**Назва**	**Границі значень**	**Колір, RGB**
Вода	-1-0,6	(0, 0, 255)			
Пляж	0,6-0,66	(255, 255, 0)			
Рівнини	0,66-0,825		Луки	-1-0,2	(255, 0, 0)
			Поле	-0,2-0	(0, 255, 255)
			Ліс	0-0,2	(0, 255, 0)
			Болото	0,2-1	(255, 0, 255)
Гори	0,825-1	(0, 0, 0)			

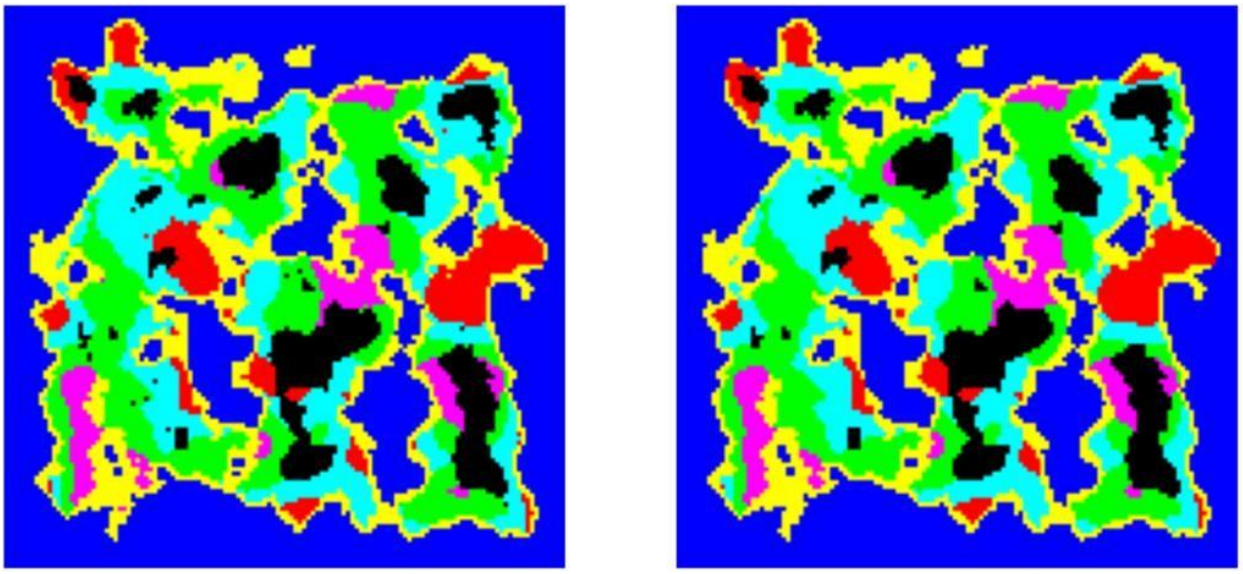
Під час класифікації можуть виникати окремі пікселі, колір яких відрізняється від оточуючого фону [25]. Це можуть бути випадкові «невідповідні» кольори серед однорідної зони, що порушують цілісність зображення. Для їх виправлення використовується функція remove_lonely_pixels, яку показано у лістингу 3.4. Вона послідовно проходить по всій карті, окрім крайніх пікселів, аналізуючи кожен із них. Якщо піксель має одного або менше сусідів того ж кольору, він вважається «самотнім». У такому випадку його колір змінюється на домінантний серед восьми навколишніх сусідів, що забезпечує більш природний вигляд карти.

Лістинг 3.4 Функція післяобробки отриманого набору даних:

```
def remove_lonely_pixels(image, biome_colors):
    h, w, _ = image.shape
    cleaned = image.copy()
    for i in range(1, h-1):
        for j in range(1, w-1):
            center = tuple(image[i, j])
```

```
            neighbors = [tuple(image[i+di, j+dj]) for di in [-1,0,1] for dj in [-1,0,1] if (di,dj) != (0,0)]
            if sum(1 for c in neighbors if c == center) <= 1:
                most_common = Counter(neighbors).most_common(1)[0][0]
                cleaned[i, j] = np.array(most_common)
        return cleaned
```

На рисунку 3.1 (а) представлено приклад зображення до обробки. Унаслідок застосування функції післяобробки спостерігається помітне зменшення шуму та заміна артефактів на більш природні елементи [26], що зображено на рисунку 3.1 (б).



(а) (б)

Рисунок 3.1 – Приклад післяобробки елемента з набору даних:
(а) до обробки; (б) після

Отриманий набір даних характеризується розміром зображень 128×128 пікселів, що забезпечує оптимальний баланс між деталізацією ландшафту та продуктивністю при обробці. Всі згенеровані карти збережено у форматі PNG, який гарантує збереження якості без втрат і є широко сумісним з більшістю інструментів для подальшої роботи з графікою. Загалом було

створено 5000 унікальних зразків, причому кількість зображень, що генеруються, можна легко регулювати за допомогою параметра num_images у скрипті. Файли організовані в директорії dataset, де кожне зображення має послідовне ім’я виду biome_map_<номер>.png, що спрощує їх пошук та обробку.

Різноманітність карт забезпечується насамперед за рахунок використання випадкових зміщень при генерації шумових карт, а також варіативності додаткових природніх поясів у межах основної природної зони рівнини. Завдяки цьому кожна створена карта має унікальний ландшафт з природним розподілом водних об’єктів, пустель, луків, лісів, боліт та гірських районів. Такий підхід не лише підвищує реалізм, але й дає змогу формувати багатогранні ігрові світи з мінімальними затратами ресурсів.

Таким чином, запропонована реалізація відзначається поєднанням простоти коду з високою гнучкістю, що досягається за рахунок регулювання параметрів шуму Перліна та впровадження механізмів очищення «шумових» артефактів. В результаті створюється якісна база зображень островів із різноманітним рельєфом і природними зонами, які можуть бути використані в різних ігрових жанрах. Крім того, дана база легко масштабується – змінюючи роздільність зображень або кількість зразків, можна швидко адаптувати її до конкретних завдань розробки ігрових світів.

3.1.2 Реалізація GAN

У рамках поставленого завдання було розроблено генеративну змагальну мережу для генерації зображень процедурно створених ігрових карт. Модель реалізована з використанням бібліотеки PyTorch та включає дві основні складові: генератор і дискримінатор.

Генератор отримує на вхід випадковий вектор-шуми (latent vector) розмірності 100, який трансформується послідовністю транспонованих

згорткових шарів (ConvTranspose2d) для поетапного збільшення просторових розмірностей вихідного тензора. Починаючи з розмірності 1×1, в результаті послідовних шарів генератор створює RGB-зображення розміром 128×128 пікселів з трьома кольоровими каналами.

Для стабілізації навчання на кожному кроку застосовуються операції нормалізації батчу (BatchNorm2d), а функція активації ReLU забезпечує нелінійність. Заключним шаром є активація tanh, що нормалізує вихідні значення пікселів до діапазону [-1, 1], що узгоджується з подальшим нормалізованим форматом вхідних зображень. Реалізацію структури генератора показано в лістингу 3.5.

Лістинг 3.5 Клас, що описує архітектуру генератора:

```
class Generator(nn.Module):
    def __init__(self, z_dim):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(z_dim, 1024, 4, 1, 0, bias=False),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
            nn.ConvTranspose2d(1024, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
```

```
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 32, 4, 2, 1, bias=False),
            nn.BatchNorm2d(32),
            nn.ReLU(True),

            nn.ConvTranspose2d(32, 3, 3, 1, 1, bias=False),
            nn.Tanh()
        )
    def forward(self, z):
        return self.main(z)
```

Дискримінатор, навпаки, приймає на вхід зображення 128×128×3 і послідовно зменшує його розмірність за допомогою згорткових шарів Conv2d зі зменшенням просторових розмірностей [27]. Кожен згортковий шар супроводжується нормалізацією батчу (BatchNorm2d) та нелінійною активацією LeakyReLU, що дозволяє уникнути «затухання» градієнтів.

На виході дискримінатора формується одновимірний тензор з одним значенням, яке інтерпретується як ймовірність того, що подане зображення є реальним (з тренувального набору) або підробленим (створеним генератором). Для цього використовується сигмоїдна активація, що нормалізує вихід до інтервалу [0, 1]. Все це реалізовано класом, що показано у лістингу 3.6.

Лістинг 3.6 Клас, що описує архітектуру генератора:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
```

```
            nn.Conv2d(64, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, 4, 2, 1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(512, 1024, 4, 2, 1, bias=False),
            nn.BatchNorm2d(1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(1024, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.main(x).view(-1, 1)
```

Для тренування GAN використовувався спеціально підготовлений набір даних процедурно згенерованих карт з роздільністю 128×128 px [28]. Дані попередньо оброблялися та нормалізувалися (масштабування значень пікселів до діапазону [-1, 1]) за допомогою бібліотеки torchvision.transforms.

Основні гіперпараметри моделі:

- розмір латентного простору (*z_dim*): 100;
- розмір батчу: 8;
- кількість епох навчання: 100;
- початкові швидкості навчання: 0.0001 для генератора та 0.0004 для дискримінатора.

Для підвищення стабільності навчання швидкість навчання динамічно коригується в ході епох, плавно знижуючись, що допомагає уникнути ефектів перенавчання або «заїдання» в локальних мінімумах.

Навчання GAN побудовано на змагальному принципі: генератор намагається створювати максимально реалістичні зображення, здатні «обманути» дискримінатор, а дискримінатор – відрізняти реальні дані від підроблених.

Для кожного батчу вибираються реальні зображення із набору даних, а генератор створює відповідну кількість фейкових зображень із випадкових векторів. Дискримінатор оновлює свої ваги, намагаючись правильно класифікувати реальні та підроблені зразки. Потім оновлюються ваги генератора, орієнтуючись на здатність «обдурити» дискримінатор.

У реалізації застосована оптимізація процесу: дискримінатор оновлюється не на кожному кроці, а через два, що дозволяє зменшити коливання в процесі навчання.

Після кожної п’ятої епохи згенеровані зображення зберігаються на диск, що дозволяє візуально контролювати прогрес навчання.

Для можливості паузи та подальшого відновлення навчання у коді реалізована система збереження контрольних точок (checkpointing). Усі параметри моделей і оптимізаторів зберігаються у файлах формату .pt, що дозволяє відновити тренування з будь-якої епохи без втрати проміжних результатів.

Після завершення навчання генератор здатен створювати високоякісні зображення ігрових карт з різноманітним ландшафтом, який відповідає особливостям вхідного набору даних. Для демонстрації якості створених зображень реалізована функція візуалізації декількох зразків, що дозволяє швидко оцінити результат. Результат навчання на кожній п’ятій епосі показано на рисунках 3.2–3.5.

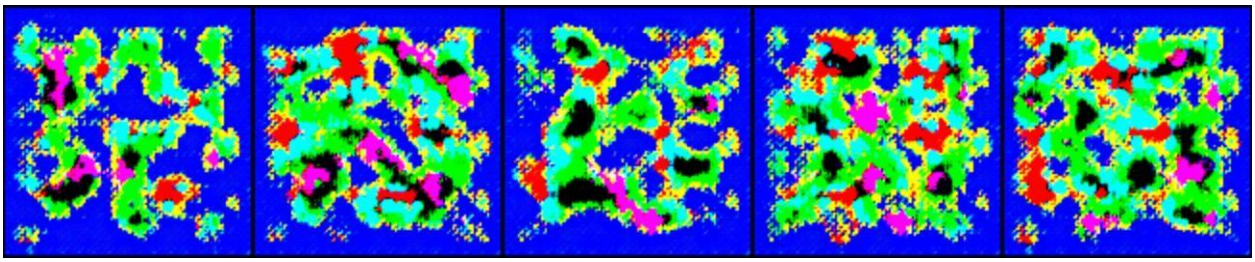


Рисунок 3.2 – Результат навчання GAN на 5-ій епосі

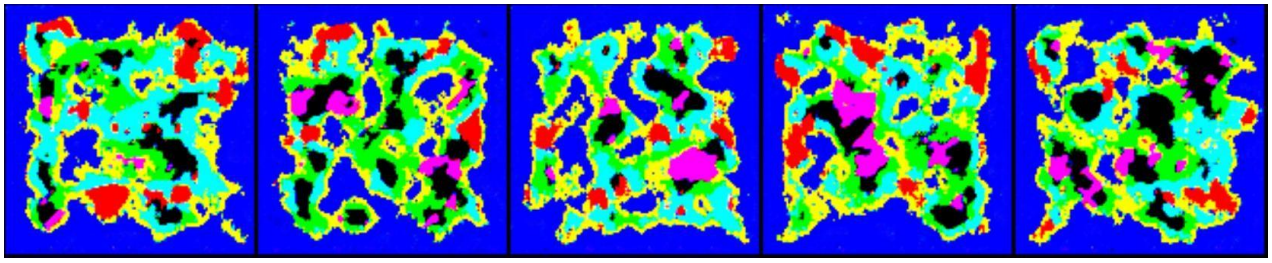


Рисунок 3.3 – Результат навчання GAN на 10-ій епосі

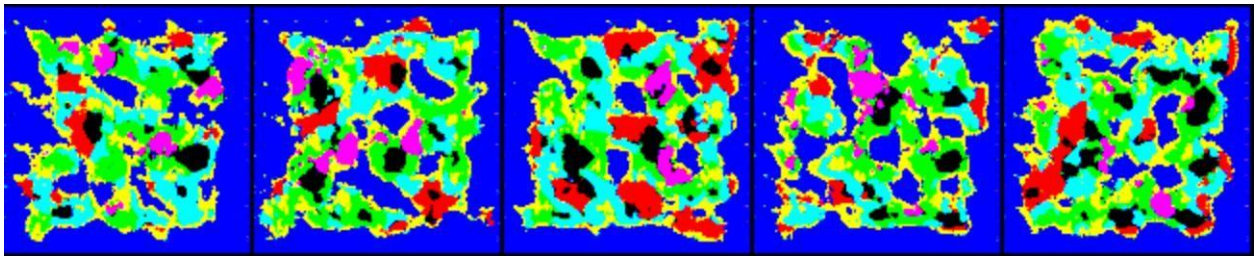


Рисунок 3.4 – Результат навчання GAN на 15-ій епосі

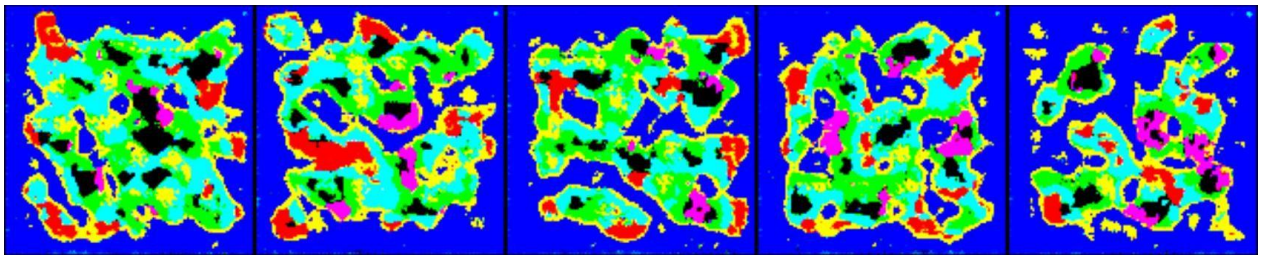


Рисунок 3.5 – Результат навчання GAN на 20-ій епосі

Таким чином, реалізована GAN-система успішно вирішує задачу генерації процедурних карт для ігрових застосувань, демонструючи хорошу якість і стабільність навчання завдяки грамотній архітектурі, підбору гіперпараметрів та оптимізованому процесу тренування.

3.1.3 Генерація зображення на основі навченої GAN

У рамках дослідження для генерації нових зображень процедурних карт була розроблена генеративна нейронна мережа, архітектура якої базується на послідовності транспонованих згорткових шарів (ConvTranspose2d). Дана модель приймає на вхід випадковий латентний вектор розмірності 100 та трансформує його у кольорове зображення розміром 128×128 пікселів з трьома каналами (RGB). В якості функції активації на останньому шарі застосовується tanh, що масштабує значення пікселів у діапазон [-1, 1].

Для відтворення та подальшого використання результатів навчання моделі була реалізована можливість завантаження збережених ваг у форматі .pt. Це дозволяє відновити навчений стан генератора без необхідності повторного тренування. Ваги моделі завантажуються з контрольної точки за допомогою функції torch.load із урахуванням наявності або відсутності графічного процесора (GPU). Після завантаження модель переводиться в режим оцінки (eval()), що відключає навчальні операції, такі як оновлення ваг чи застосування Dropout.

Генерація зображень відбувається через подачу на генератор випадкових латентних векторів, отриманих із стандартного нормального розподілу. Після формування батчу зображень, результати нормалізуються та зберігаються на диск у папку Generated у форматі PNG за допомогою функції save_image.

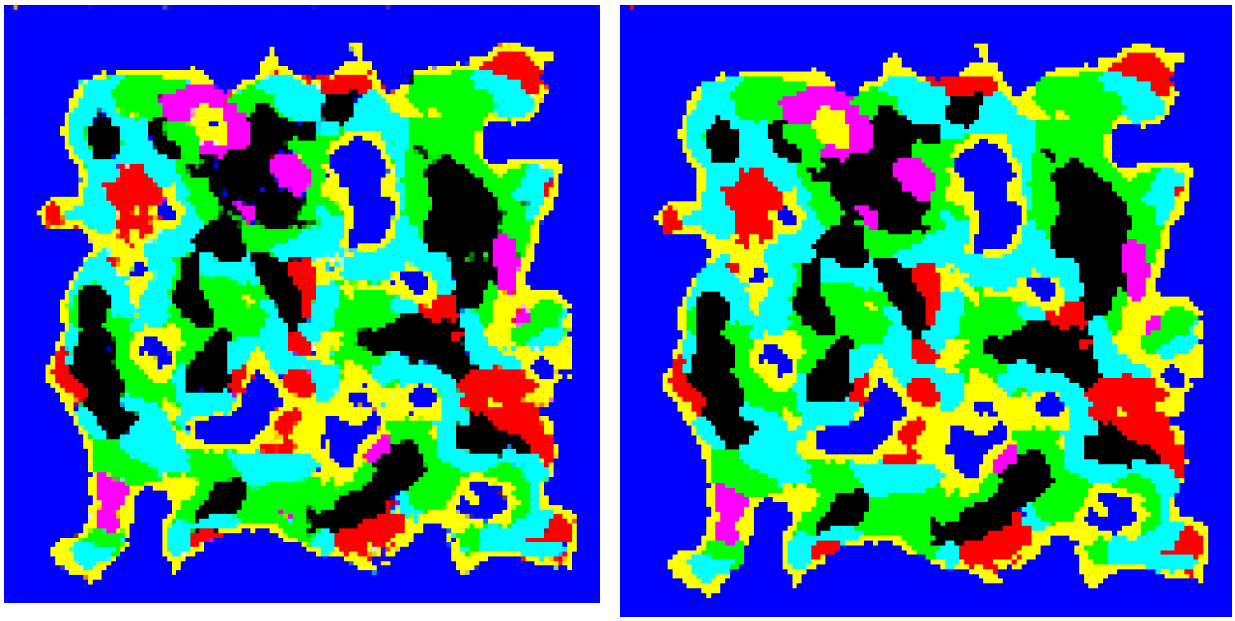
Однак, оскільки згенеровані зображення мають безперервний спектр кольорів, не завжди відповідають дискретним категоріям природних зон, для досягнення узгодженості із заданою палітрою застосовується післяобробка.

Післяобробка полягає у приведенні кожного пікселя до найближчого кольору із заздалегідь визначеного набору кольорів природних зон. Для цього використовується колірна модель CIELAB, яка забезпечує більш адекватне сприйняття різниці кольорів людиною. Кожен піксель зображення

конвертується в простір LAB, після чого для нього знаходиться найближчий колір із палітри природних зон за евклідовою відстанню в цьому просторі. Таким чином формується кольорова карта, що складається тільки з дискретних кольорів, асоційованих із різними типами природних зон, що показано на рисунку 3.6 (а).

Додатково застосовується алгоритм очищення «одиноких» пікселів - точок, які мають менше одного сусіда з таким самим кольором. В таких випадках колір пікселя замінюється на домінантний колір серед його восьми сусідів. Цей крок знижує «шум» на карті та робить переходи між природними зонами більш плавними і природними.

Результат післяобробки зберігається окремо і візуалізується для контролю якості на рисунку 3.6 (б). Таким чином, отримані зображення відповідають заданій палітрі природних зон, що полегшує їх подальше використання для розробки ігрових світів.



(а) (б)

Рисунок 3.6 – Згенероване зображення ігрової карти:

(а) до обробки; (б) після

3.2 Використання згенерованої карти в Unity

3.2.1 Інтеграція генеративної мережі в Unity

Для підвищення рівня інтерактивності та варіативності ігрового процесу було реалізовано автоматичний запуск Python-скрипта при натисканні кнопки «Нова гра» у Unity. Це дозволяє кожного разу під час старту нової сесії створювати унікальну карту, що зберігається у внутрішніх файлах гри та відображається на ігровій сцені.

Запуск скрипта виконується за допомогою класу Process з бібліотеки System.Diagnostics, що дозволяє безпосередньо запускати зовнішні процеси з Unity. У даному випадку – це Python-інтерпретатор із зазначенням шляху до скрипта, що показано у лістингу 3.7.

Лістинг 3.7 Налаштування запуску Python-скрипта:

```
// Отримання шляху до Python-скрипта, що зберігається в Assets/Python
string scriptPath = Path.Combine(Application.dataPath, "Python", "generate.py");
UnityEngine.Debug.Log("Шлях до скрипта: " + scriptPath);
// Налаштування процесу для запуску Python-скрипта
ProcessStartInfo start = new ProcessStartInfo();
start.FileName = "python"; // Назва інтерпретатора
start.Arguments = $"\"{scriptPath}\""; // Вказівка шляху до скрипта
start.UseShellExecute = false;
start.RedirectStandardOutput = true;
start.RedirectStandardError = true;
start.CreateNoWindow = true;
Process process = new Process();
process.StartInfo = start;
```

```
// Запуск процесу
    process.Start();
    // Асинхронне читання виводу
    process.BeginOutputReadLine();
    process.BeginErrorReadLine();
    // Очікування завершення процесу
    process.WaitForExit();
    // Після генерації карти, завантажуємо основну сцену гри
    SceneManager.LoadScene(1);
```

Python-скрипт, який запускається при натисканні кнопки «Нова гра», зберігає результат у вигляді PNG-зображення у визначену директорію (Assets/Python/Generated), яку показано на рисунку 3.7. Це зображення після запуску нової сцени завантажується в Unity як текстура, що використовується для відображення ландшафту на ігровій мапі. Таким чином, текстура виконує роль основи для подальшої побудови ігрового світу.

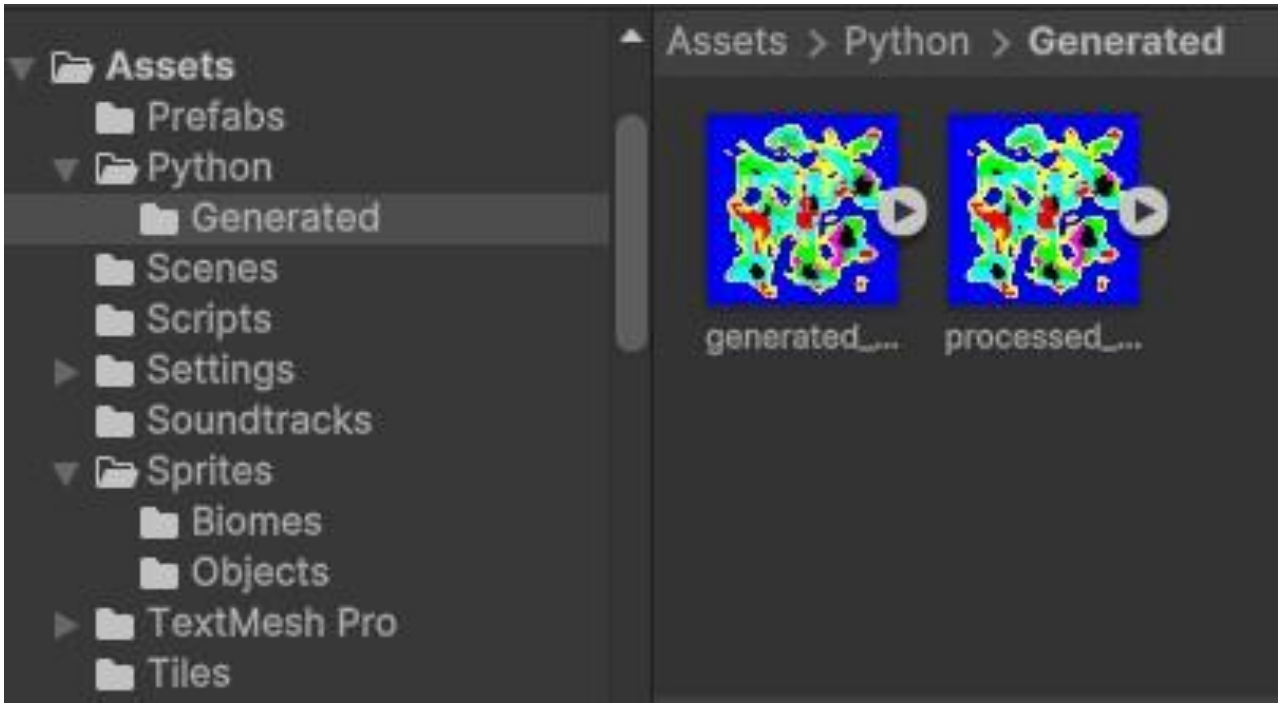


Рисунок 3.7 – Збереження згенерованого зображення у визначену директорію

Інтеграція генерації процедурних карт при запуску нової гри в Unity дозволила досягти високого рівня варіативності ігрового світу [29]. Завдяки динамічному виклику Python-скрипта, кожен старт нової сесії генерує унікальний ландшафт, що підвищує інтерес до повторного проходження та додає елементи дослідження у гру. Це рішення виявилося ефективним завдяки правильній організації процесів між Unity та Python, що забезпечило стабільну взаємодію та коректне завантаження результатів у середовище Unity.

3.2.2 Використання Tilemap для процедурного генерування

Для візуалізації процедурно згенерованих карт у середовищі Unity було реалізовано автоматичне відтворення ігрової мапи у вигляді плиток. Це забезпечує динамічне створення світу, який повністю узгоджується зі згенерованими зображеннями, а також оптимізує використання ресурсів завдяки розподіленому підходу до відображення елементів ландшафту [30].

Unity пропонує зручний інструмент для побудови двовимірних рівнів – Tilemap, який дозволяє ефективно розміщувати об’єкти на сітці. У реалізації використовується скрипт BiomeMapGen, що здійснює перетворення згенерованої карти у вигляді PNG-зображення в набір плиток на Tilemap.

Основні елементи, що використовуються для створення Tilemap:

– tilemap – посилання на Tilemap у Unity, куди будуть розміщені плитки;

– biomeRuleTiles – масив RuleTile, що визначає правила відображення кожної природної зони;

– biomeColors – палітра кольорів, що відповідає типу зони;

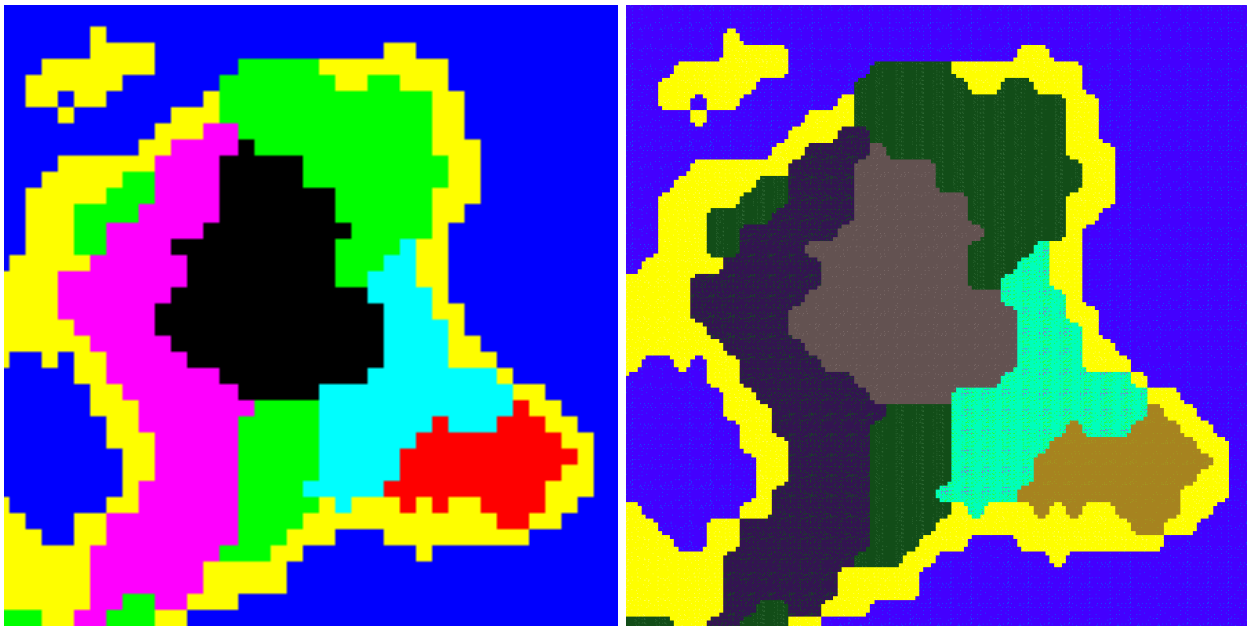
– biomeMap – текстура, що завантажується з файлу та слугує основою для побудови ландшафту.

Щоб підвищити деталізацію ігрової карти, початкове зображення масштабується та згладжується перед генерацією плиток.

Функція ScaleTexture збільшує розмір текстури відповідно до заданого множника *scaleFactor*. Це дозволяє створювати більші ігрові світи на основі тієї ж карти. Масштабування відбувається з урахуванням кольорової інформації кожного пікселя. Це означає, що кожний піксель згенерованого за допомогою нейронної мережі зображення 128×128 дробиться на декілька ще. Тобто при значенні *scaleFactor = 4*, кожен піксель ділится на 16 інших, бо дроблення відбувається по двох координатах, тому кількість отриманих пікселей в $scaleFactor^2$ разів більше, ніж було згенеровано. Тому ми отримуємо ігрову карту розміром 512×512 пікселей (тобто, ігрових метрів). Таким чином можна спокійно масштабувати карту в залежності від потреб та технічних вимог.

Після масштабування карта проходить через алгоритм згладжування, який усуває різкі переходи між зонами та робить зображення більш природним. Функція SmoothTexture обробляє кожен піксель текстури, аналізуючи кольори його сусідів у заданому радіусі. Вона визначає найпоширеніший колір серед сусідніх пікселів і порівнює його з кольором поточного пікселя. Якщо поточний колір значно відрізняється від переважаючого, то він замінюється, щоб краще відповідати оточенню.

Далі відбувається побудова карти на Tilemap через метод GenerateMap. Ця функція проходить по кожному пікселю текстури та перевіряє його колір. Якщо колір відповідає кольору одної з природних зон, на відповідну позицію на Tilemap розміщується RuleTile, який вказує, який тип зони там має бути відображений. Для кращого розуміння процесу генерації та заповнення ігрового оточення природними зонами на рисунку 3.8 (а) представлено ілюстрацію частини зображення розміром 32×32 та на рисунку 3.8 (б) цю ж частину отриманого після проведення попередніх маніпуляцій зображення розміром 128×128.



(а) (б)

Рисунок 3.8 – Приклад перетворення вихідного зображення в Tilemap: (а) піксельне зображення 32×32; (б) текстуроване зображення 128×128

Одразу після генерації карти гравець автоматично переміщується у визначену природну зону, яка вказується через індекс *targetBiomeIndex*. Для цього функція MovePlayerToNearestBiome аналізує всі пікселі карти, знаходить найближчий до центру піксель заданої зони і переміщує гравця у відповідну точку.

Реалізація процедурної генерації карти у вигляді плиток дозволила інтегрувати згенеровані зображення у форматі PNG у Unity і динамічно будувати ігровий світ із чітким розмежуванням природних зон. Завдяки цьому кожен запуск гри створює унікальну карту, що підвищує інтерес до дослідження світу та розширює можливості для інтерактивності.

3.2.3 Заповнення середовища

Для підвищення реалізму та деталізації процедурно згенерованого світу в Unity було реалізовано механізм створення об’єктів у відповідності до зон,

представлених на Tilemap. Це дозволяє оживити ігровий світ деревами, кущами, скелями та іншими декоративними або функціональними елементами, відповідно до типу місцевості.

Для організації автоматичного створення використовується клас BiomeObjectSpawner, що має наступні ключові поля:

- tilemap – посилання на Tilemap з розміткою зон, яка була сформована на основі процедурної генерації;
- biomeTiles – масив тайлів, кожен з яких відповідає певній зоні. Порядок елементів є важливим для ідентифікації зоною за плиткою;
- biomeData – масив структур BiomeObjects, кожен з яких містить набір шаблонів ігрових об’єктів для відповідної природної зони;
- spawnChance – ймовірність появи об’єкта на одній плитці, що дозволяє контролювати густоту насичення середовища;
- objectOffsets – масив векторів зсувів для кожної зони, які задають початкову позицію об’єкта відносно центра плитки;
- rndShift – випадкове додаткове зміщення в межах деякого відсотку від розміру плитки, що робить розміщення об’єктів більш природним.

Основна логіка появи об’єктів виконується у методі SpawnObjects() та описується наступними кроками:

Крок 1. За допомогою tilemap.cellBounds визначається діапазон координат плиток, які потрібно обробити.

Крок 2. Для кожної плитки отримується відповідний об’єкт, яким плитка буде заповнюватись.

Крок 3. Викликається функція GetBiomeIndex(tile), яка повертає індекс природної зони за плиткою. Якщо плитка не належить жодній зоні, ця плитка пропускається.

Крок 4. За індексом зони отримується масив шаблонів об’єктів, які можуть бути створені на цій плитці.

Крок 5. З урахуванням ймовірності spawnChance вирішується, чи слід розміщувати об’єкт на цій плитці.

Крок 6. Вибирається випадковий шаблон об’єкта із масиву.

Крок 7. Обчислюється позиція появи об’єкта, що включає:

– базову позицію плитки в світі (через tilemap.CellToWorld);

– зсув для природної зони з масиву objectOffsets;

– випадкове зміщення в межах rndShift для додання природності розміщенню;

– фіксовану Z-позицію для коректного відображення в двовимірному просторі.

Крок 8. Створюється шаблон об’єкта з обчисленою позицією.

Крок 9. За допомогою компонента BoxCollider2D та SpriteRenderer виконується сортування об’єкта за глибиною, щоб забезпечити правильний порядок накладання графічних елементів залежно від їх положення.

3.3 Створення логіки ігрового процесу

3.3.1 Реалізація механік управління гравцем

Для реалізації управління персонажем у 2D середовищі було використано компонент Rigidbody2D для фізичної симуляції руху та взаємодії з об’єктами світу. Також було застосовано BoxCollider2D або CircleCollider2D для обробки зіткнень.

Гравець може рухатись у чотирьох напрямках за допомогою клавіш:

– *W* – вгору;

– *A* – вліво;

– *S* – вниз;

– *D* – вправо.

Для цього реалізовано скрипт, який щосекунди зчитує введення з клавіатури та змінює швидкість Rigidbody2D. Метод зміни положення гравця при умові натискання однієї з клавіш показано в лістингу 3.8.

Лістинг 3.8 Ілюстрація метода руху персонажа:

```
void FixedUpdate()
{
    Vector2 direction = Vector2.zero;

    if (Input.GetKey(KeyCode.A)) direction.x -= 1;
    if (Input.GetKey(KeyCode.D)) direction.x += 1;
    if (Input.GetKey(KeyCode.W)) direction.y += 1;
    if (Input.GetKey(KeyCode.S)) direction.y -= 1;

    direction = direction.normalized * moveSpeed;
    rb.linearVelocity = direction;
}
```

Коли гравець наближається до спеціального об'єкта (наприклад, предмета, що можна зламати), і натискає клавішу *E*, виконується дія – об’єкт знищується або змінюється.

Якщо поблизу є ворог, при натисканні *E* або ліва кнопка миші гравець атакує. Ворог отримує пошкодження через просту логіку, що показана в лістингу 3.9.

Лістинг 3.9 Логіка нанесення пошкодження:

```
public class Enemy : MonoBehaviour
{
    public int health = 3;

    public void TakeDamage(int damage)
    {
        health -= damage;
```

```
        if (health <= 0)
        {
            Destroy(gameObject); // Ворог загинув
        }
    }
}
```

3.3.2 Реалізація взаємодії с ігровим оточенням

У грі реалізована можливість руйнувати різні об’єкти ігрового середовища, приклад якого показано на рисунку 3.9 (а). Кожен об’єкт має визначену кількість «здоров’я», яке зменшується при взаємодії з гравцем. Після повного руйнування об’єкта він зникає, а натомість у світі з’являються певні ресурси, які гравець може підбирати, як показано на рисунку 3.9 (б). Ці ресурси автоматично додаються до інвентаря персонажа, що дозволяє використовувати їх у подальшому для створення об’єктів або харчування.

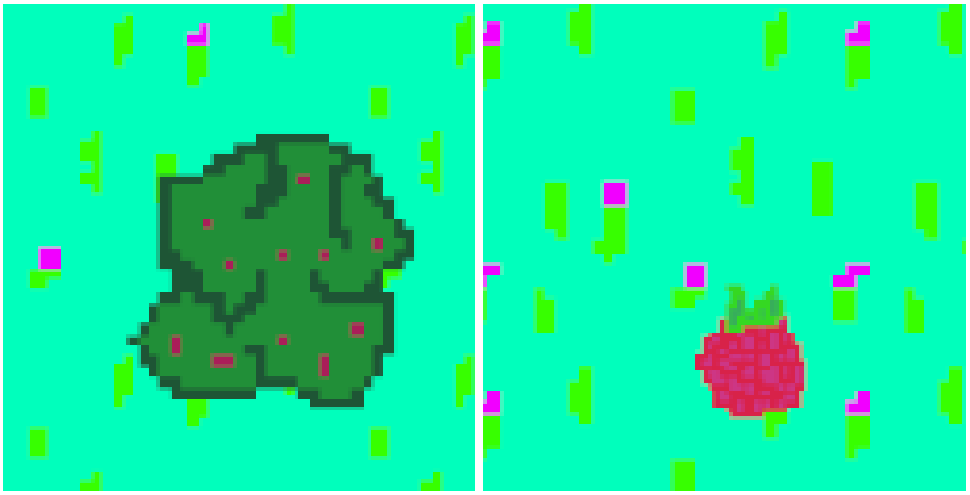


Рисунок 3.9 – Взаємодія між персонажем та ігровим об’єктом:
(а) зображення об’єкта взаємодії;
(б)зображення ресурсу, що випадає після руйнування об’єкта

При руйнуванні кожного об’єктів з них можуть випадати різні ресурси, використання яких теж може бути конкретним та залежатиме від ресурсу. Відповідність між ігровим об’єктом, ресурсом що випадає з нього та його призначення зображено у таблиці 3.2.

Таблиця 3.2 – Класифікація ігрових об’єктів, природних зон та ресурсів

Ігровий об’єкт	**Природна зона в межах якої він з’являється**	**Ресурс, що з нього випадає**	**Призначення ресурсу**
Водорість	Вода	Сира водорість	Їжа
Кактус	Пляж	Плод кактуса	Їжа
Куст	Луг	Ягода	Їжа
Пшениця	Поле	Зерно	Їжа
Дерево	Ліс	Деревина	Виготовлення предметів
Очерет	Болото	Волокна	Виготовлення предметів
Каміння	Гори	Залізна руда	Виготовлення предметів

Інвентар реалізований таким чином, щоб гравець міг зручно керувати підібраними предметами – переглядати, вибирати, використовувати або скидати їх (рис. 3.10). Інтерфейс інвентаря оптимізовано для швидкого доступу до об’єктів та забезпечення інтуїтивного керування навіть у динамічних ситуаціях. Така система робить взаємодію з ігровим світом більш гнучкою і надає більше стратегічних можливостей під час проходження гри, дозволяючи гравцю адаптуватися до змін, керувати ресурсами, комбінувати предмети або готуватися до майбутніх викликів. Крім того, модульна структура інвентаря забезпечує можливість масштабування та подальшого розширення функціональності.

Рисунок 3.10 – Ілюстрація інвентаря з деякими заповненими комірками

В ігровому світі присутні неігрові істоти, які мають власний штучний інтелект. Деякі неігрові істоти перебувають у стані спокою і патрулюють визначену територію, однак при виявленні гравця у межах певного радіусу вони активуються і починають переслідувати його.

Таке переслідування змушує гравця бути уважним і швидко реагувати, оскільки неігрові істоти можуть завдати шкоди при безпосередньому контакті, що робить їх небезпечними супротивниками і ускладнює проходження рівнів.

Гравець може отримувати пошкодження від неігрових істот під час їхнього контакту або атаки, що веде до зменшення показника здоров’я. Система бою передбачає, що при отриманні пошкоджень здоров’я гравця падає, що ставить під загрозу його життя у грі. Водночас гравець має можливість завдавати пошкодження неігровим істотам за допомогою власної зброї або інших засобів.

Ця взаємодія реалізує динамічний бойовий процес, який поєднує в собі елементи тактики та швидкої реакції. Взаємне нанесення пошкоджень створює баланс між агресією та обороною, стимулюючи гравця розвивати навички та правильно планувати дії.

Для моделювання стану персонажа використовуються три основні шкали: здоров’я, насиченість їжею та насиченість водою, їх вигляд в інтерфейсі показано на рисунку 3.11. Здоров’я відображає загальний фізичний стан гравця і зменшується при отриманні пошкодження або при

нестачі харчування та води. Насиченість їжею і водою з часом поступово зменшується, і якщо рівень хоч одного з цих параметрів падає нижче 50%, починає знижуватися здоров’я.

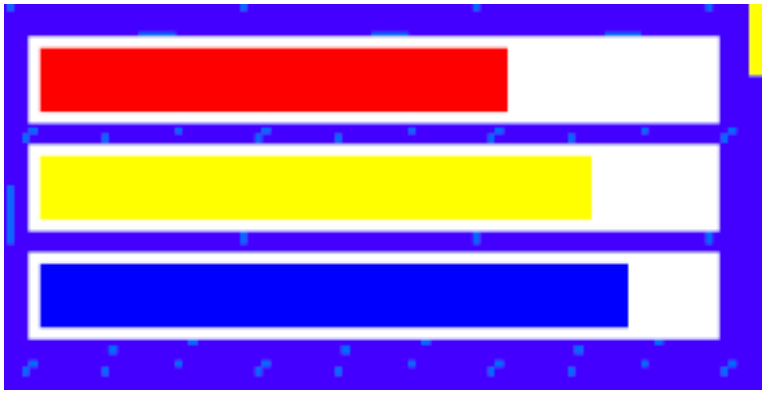


Рисунок 3.11 – Вигляд шкал потреб: здоров’я, голоду та спраги

Такий підхід створює реалістичну систему виживання, де гравцеві необхідно постійно контролювати свої потреби, своєчасно харчуватися та пити, щоб підтримувати оптимальний фізичний стан і уникнути негативних наслідків голоду та зневоднення.

Підібрані у світі гри предмети, які відносяться до категорії їжі, можуть бути використані гравцем для відновлення рівня насиченості їжею. Це дозволяє своєчасно запобігати голоду і підтримувати здоров’я персонажа на достатньому рівні.

При знаходженні гравця поблизу природних джерел води реалізовано механіку автоматичного збільшення показника насиченості водою. Це дозволяє гравцеві швидко поповнювати рівень гідратації без необхідності використання предметів з інвентарю.

Це робить гру більш реалістичною, адже природне джерело води стає важливим ресурсом для виживання, а також стимулює дослідження ігрового світу у пошуках необхідних для життя ресурсів.

Реалізовані механіки взаємодії з ігровим світом – руйнування об’єктів, підбір ресурсів, поведінка неігрових істот із ШІ, бойова система, а також контроль за станом здоров’я, голоду та спраги – є ключовими елементами

сучасних ігор з відкритим світом. Вони забезпечують глибоку інтерактивність і реалістичність геймплею, стимулюючи гравця до активного дослідження та планування своїх дій. Завдяки процедурній генерації карти при кожному запуску ігрового світу, а також завдяки вищезазначеним механікам, створюється унікальний ігровий досвід, який підтримує інтерес гравця і мотивує до повторного проходження гри.

3.3.3 Приклади застосованої оптимізації

Для забезпечення стабільної роботи гри та підвищення продуктивності було впроваджено ряд оптимізаційних рішень, які мінімізують навантаження на апаратні ресурси. Один із ключових механізмів полягає у обмеженні активації штучного інтелекту неігрових істот. Замість того, щоб вони постійно сканували простір і реагували на гравця, їх поведінка активується лише тоді, коли гравець опиняється в межах визначеного радіуса. Такий підхід дозволяє значно зменшити кількість одночасно активних процесів, що особливо важливо для ігор з великою кількістю неігрових істот і складним ігровим світом.

Подібний принцип використовується і для пошуку ресурсів у світі гри. Зокрема, коли персонаж потребує відновлення насиченості водою, пошук найближчого джерела води виконується не по всій карті, а лише в обмеженій зоні навколо гравця. Це дозволяє уникнути зайвих обчислень і прискорює виконання ігрової логіки, зберігаючи при цьому функції ігрового процесу в повному обсязі.

Також для оптимізації продуктивності реалізовано механізм відключення ігрових об’єктів, які знаходяться далеко від персонажа. Віддалені предмети або істоти не оновлюють свою логіку і не відображаються на екрані, що суттєво знижує навантаження на систему, особливо в великих відкритих світах. У разі наближення гравця об’єкти

знову активуються і відновлюють свою функціональність, що забезпечує баланс між якістю відображення і швидкодією гри.

Таким чином, комплекс заходів з оптимізації дозволяє підтримувати стабільну кількість кадрів на секунду та забезпечує комфортний ігровий досвід навіть на пристроях із обмеженими ресурсами. Ці рішення є важливою складовою при розробці ігор з відкритим світом, де велика кількість активних елементів може призводити до значних технічних труднощів.

3.4 Демонстрація роботи гри

Реалізовані механіки ігрового процесу продемонстровано на рисунках 3.12–3.15, які ілюструють основні елементи інтерфейсу та взаємодії користувача з грою. Представлені зображення головного меню, меню опцій, дві сцени безпосереднього ігрового процесу. Кожен скріншот відображає ключові етапи та функціональні можливості проєкту, що дозволяє краще зрозуміти структуру та логіку гри.

Рисунок 3.12 – Головне меню гри

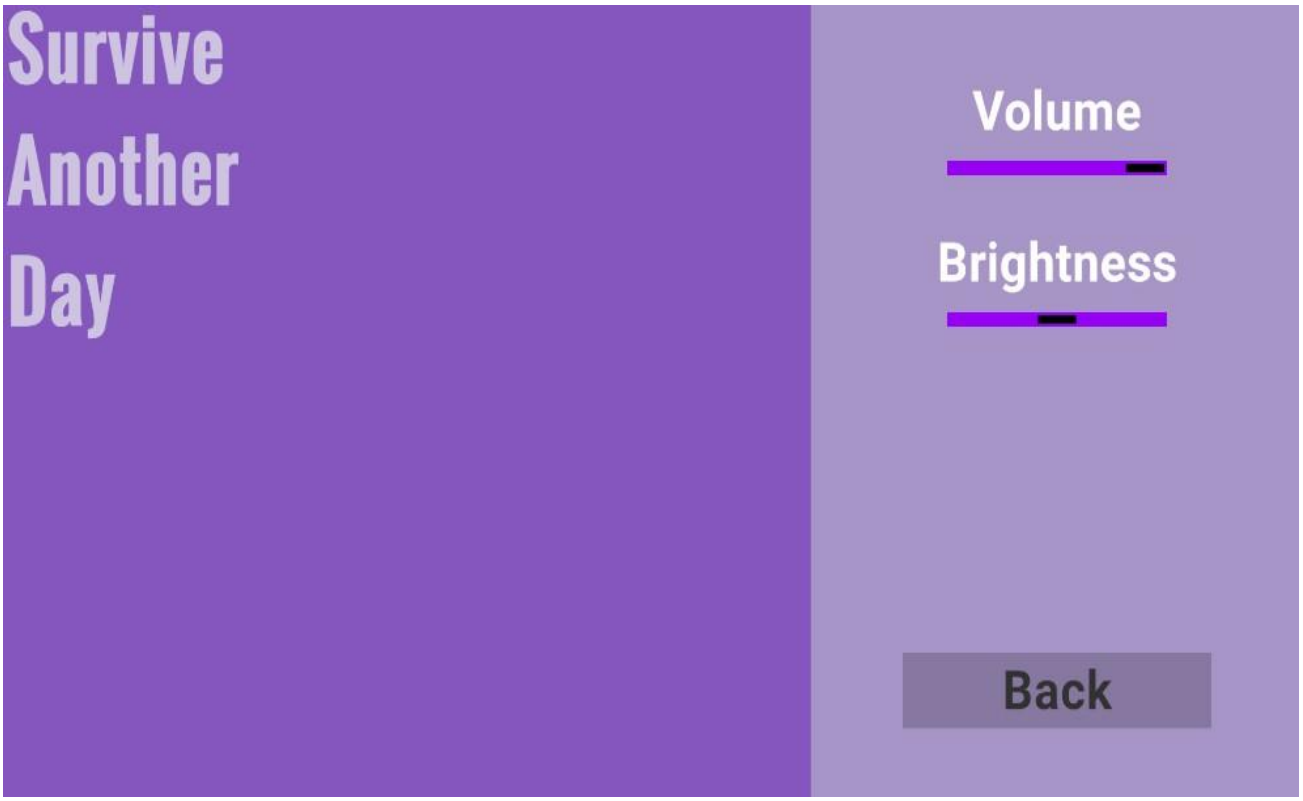


Рисунок 3.13 – Налаштування гучності та яскравості

Рисунок 3.14 – Перший приклад сцени ігрового процесу

Рисунок 3.15 – Другий приклад сцени ігрового процесу

Узагальнюючи, наведені скріншоти відображають цілісну картину розробленого програмного продукту, демонструючи як користувацький інтерфейс, так і основні аспекти взаємодії з ігровим світом. Візуальні матеріали підтверджують працездатність реалізованих механік, зокрема управління параметрами гри, перебіг геймплею та реакцію системи на дії гравця.

ВИСНОВКИ

У рамках кваліфікаційної роботи була розроблена гра на платформі Unity з використанням генеративно-змагальних мереж.

Основною метою роботи було створення динамічно сформованого середовища, виживання в різних умовах і взаємодія з унікальними зонами карти. Така система створення світу не тільки підвищує варіативність, але й дозволяє масштабувати гру без ручного дизайну кожної локації.

Під час розробки гри з процедурною генерацією карти на основі перлін-шуму були використані сучасні технології та інструменти. Для генерації ігрового світу використовувався Python – мова, що дозволила реалізувати складні алгоритми шуму, класифікацію природних зон, побудову висотних карт та їх подальшу візуалізацію. Серед основних бібліотек – noise, numpy, matplotlib, які забезпечили зручну роботу з даними, обробку зображень та створення великої кількості унікальних карт.

Для реалізації самої гри використовувалася мова програмування C# у зв’язці з ігровим рушієм Unity. C# відповідає за логіку гри, завантаження згенерованих карт у сцену, взаємодію гравця з навколишнім світом, обробку подій та UI. Такий поділ задач між Python та C# дозволив створити гнучку архітектуру проєкту, в якій генерація світу та ігрова механіка функціонують як окремі, незалежні блоки.

Результатом проєкту стала функціональна гра з процедурною генерацією карти, яка може слугувати основою для створення повноцінного геймплею з унікальними світами та середовищем.

Результати роботи апробовано у вигляді тез доповідей під час XXIX Міжнародного молодіжного форуму «Радіоелектроніка і молодь у XXI столітті» [31].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Rodrigues, L., Bonidia, R., & Brancher, J. (2020). Procedural versus human level generation: Two sides of the same coin?. *International Journal of Human-Computer Studies*, *141*, 102465.

2. Tvoroshenko, I., & Almakaieva, A. (2020). Application of procedural generation of game content using software algorithms.

3. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, *27*.

4. Merizzi, F. (2024). Procedural terrain generation with style transfer. *arXiv preprint arXiv:2403.08782*.

5. Gellel, A., & Sweetser, P. (2020, September). A hybrid approach to procedural generation of roguelike video game levels. In *Proceedings of the 15th International Conference on the Foundations of Digital Games* (pp. 1-10).

6. Fabricatore, C. (2007). Gameplay and game mechanics: a key to quality in videogames.

7. Sicart, M. (2008). Defining game mechanics. *Game studies*, *8*(2), 1-14.

8. Mizutani, W. K., Daros, V. K., & Kon, F. (2021). Software architecture for digital game mechanics: A systematic literature review. *Entertainment Computing*, *38*, 100421.

9. Cartlidge, J. (2024). Genre, prototype theory, and the ‘Berlin interpretation’of roguelikes. *Game Studies*, *24*(3).

10. Кобилін, О. А., & Творошенко, І. С. (2021). Методи цифрової обробки зображень.

11. Tvoroshenko, I., Pomazan, V., Gorokhovatskyi, V., & Kobylin, O. (2023). Application of video data classification models using convolutional neural networks.

12. Chakraborty, T., KS, U. R., Naik, S. M., Panja, M., & Manvitha, B. (2024). Ten years of generative adversarial nets (GANs): a survey of the state-of-the-art. *Machine Learning: Science and Technology*, *5*(1), 011001.

13. Bilonoh, B., Bodyanskiy, Y., Kolchygin, B., & Mashtalir, S. (2021, May). Tunable activation functions for deep neural networks. In *International Scientific Conference "Intellectual Systems of Decision Making and Problem of Computational Intelligence"* (pp. 624-633). Cham: Springer International Publishing.

14. Gorokhovatskyi, V. O., Tvoroshenko, I. S., & Peredrii, O. O. (2020). Image classification method modification based on model of logic processing of bit description weights vector. *Telecommunications and Radio Engineering*, *79*(1).

15. Tvoroshenko, I., Gorokhovatskyi, V., Kobylin, O., & Tvoroshenko, A. (2023). Application of deep learning methods for recognizing and classifying culinary dishes in images.

16. Pomazan, V., Tvoroshenko, I., & Gorokhovatskyi, V. (2023). Development of an application for recognizing emotions using convolutional neural networks.

17. Gorokhovatskiy, V. A., Kobylin, O. A., & Kulikov, Y. A. (2015). Application of Granulation of Feature Descriptions in Structural Image Recognition. *Telecommunications and Radio Engineering*, *74*(6).

18. Pérez, C., Verón, J., Pérez, F., Moraga, M., Calero, C., & Cetina, C. (2024). A comparative analysis of energy consumption between the widespread unreal and unity video game engines. *arXiv preprint arXiv:2402.06346*.

19. Tvoroshenko, I., & Kuznetsov, M. (2021). Research results of functional, white box and smoke testing methods for mobile applications.

20. Bodyanskiy, Y., Pliss, I., & Shafronenko, A. (2022). Adaptive neurofuzzy clustering of distorted data based on prototype-centroid strategy using evolutionary procedures. *Artificial Intelligence*, *27*, 239-244.

21. Johnson, D., & Wiles, J. (2003). Effective affective user interface design in games. *Ergonomics*, *46*(13-14), 1332-1345.

22. Dyadun, S., Bodyanskiy, Y. V., Korobchynskyi, K., & Kobylin, O. (2022). Methods For Increasing The Reliability Of The Functioning Of Pipeline Systems. In *ProfIT AI* (pp. 104-109).

23. Kobylin, O., & Putiatina, O. (2025, April). Some aspects of real-time image denoising influenced by shot noise and compound Poisson noise. In *Proceedings of the 5th Edge Computing Workshop (doors 2025), Zhytomyr, Ukraine* (pp. 109-117).

24. Кобилін, О. А., & Путятіна, О. Є. (2024). Real-time image denoising corrupted by shot-noise. *Системи обробки інформації*, (1 (176)), 46-51.

25. Zeleniy, O., Rudenko, D., Lyubchenko, V., & Lyashenko, V. (2022). Image Processing as an Analysis Tool in Medical Research.

26. Abu-Jassar, A., Rudenko, D., & Abdalla, H. (2024). Digital medical image as an object of processing and analysis. *Multidisciplinary Journal of Science and Technology*, *4*(1), 28-35.

27. Kovtunenko, A., Yakovleva, O., Liubchenko, V., & Yanholenko, O. (2020). Research of the joint use of mathematical morphology and convolutional neural networks for the solution of the price tag recognition problem. *Bull. Natl. Tech. Univ. KhPI Ser. Syst. Anal. Control Inf. Technol*, *1*, 24-31.

28. Tian, Y., Wang, Q., Huang, Z., Li, W., Dai, D., Yang, M., ... & Fink, O. (2020). Off-policy reinforcement learning for efficient and effective gan architecture search. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VII 16* (pp. 175-192). Springer International Publishing.

29. Sendroiu, P. (2023). Python Compiler and Interpreter Tool for the Unity Engine The Development of an Interpreter Tool within the Unity Game Engine.

30. Bourke, P. (2013). A space filling algorithm for generating procedural geometry and texture. *GSTF Journal on Computing (JoC)*, *3*, 1-5.

31. Ласкавий, О. С. (2025). *Розробка гри на платформі unity з використанням генеративно-змагальних мереж* (Thesis, ХНУРЕ).