Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Створення гри на Unity з автоматичною генерацією рівнів

Текстова частина до кваліфікаційної роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник кваліфікаційної роботи
доц. Жежерун О.П.
_____
(підпис)
«____» _____ 2025 р.

Виконав студент Янченко Б.І.
«____» _____ 2025 р.

Київ 2025

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем,
к.ф.-м.н.
_____ О. П. Жежерун
(підпис)
«____»_____2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту Янченку Богдану Ігоровичу факультету інформатики 4 курсу
ТЕМА «Створення гри на Unity з автоматичною генерацією рівнів»
Зміст ТЧ до кваліфікаційної роботи:

Зміст
Анотація
Вступ
1. Аналіз предметної області
2. Обґрунтування вибору засобів розробки
3. Розробка застосунку
Висновки
Список літератури
Додатки

Дата видачі «___» _____ 2024 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Створення гри на Unity з автоматичною генерацією рівнів
Календарний план виконання роботи:

№ п/п	Назва етапу	Термін виконання етапу	Примітка
1.	Отримання теми кваліфікаційної роботи	Жовтень 2024	
2.	Огляд технічної літератури за темою роботи	26.12.2024	
3.	Аналіз предметної області	06.01.2024	
4.	Розробка комп’ютерної гри	21.03.2025	
5.	Написання текстової частини кваліфікаційної роботи	15.05.2025	
10.	Створення слайдів для доповіді та написання доповіді.	16.05.2025	
11.	Остаточне оформлення роботи та слайдів	27.05.2025	
12.	Захист кваліфікаційної роботи	04.06.2025	

Студент _____
Керівник _____
“_____” _____2024 р.

ЗМІСТ

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці прототипу 2D-гри жанру «аркада» з автоматичною генерацією рівнів у середовищі Unity. У роботі реалізовано систему процедурного створення рівнів з використанням різних алгоритмів генерації: перлинного шуму, фрактального шуму, симплексного шуму, синусоїдальних функцій, а також їхніх комбінацій із поступовим збільшенням складності.

Ключові слова: Unity, автоматична генерація рівнів, Sprite Shape, Перлинний шум, фрактальний шум, симплексний шум, синусоїда, procedural generation.

ВСТУП

Індустрія геймдеву продовжує стрімко, невтомно розвиватись. Постійно з’являються та покращуються способи та методи створення нових ігор. Одним із таких напрямів є автоматична генерація контенту (англ. procedural generation), яка дозволяє нам створювати різні ігрові рівні, нові світи або об'єкти за допомогою алгоритмів. Це не тільки зменшує обсяг всієї ручної роботи для розробників, а й забезпечує варіативність, унікальність та повторювану придатність ігрового процесу. Дуже корисною така генерація є в мобільних аркадних іграх, де часто використовуються нескінченні рівні. У цих проєктах дуже важливо економити всі ресурси пристрою, особливо пам’ять. І замість того, щоб зберігати більшу кількість раніше підготовлених елементів, ця гра може просто створювати їх «на льоту» , тобто під час нашої гри. Це дозволяє нам зменшити розмір самої гри і також забезпечити різноманітність.

Предметом дослідження є алгоритми автоматичної генерації ігрових рівнів.

Метою кваліфікаційної роботи є розробка прототипу 2D-гри з автоматичною генерацією рівнів, що базується на різних методах формування ландшафту, зокрема на перлинному шумі, фрактальному шумі, синусоїдальних функціях та їх комбінаціях.

Для досягнення мети було поставлено такі завдання:

- дослідити сучасні підходи до автоматичної генерації рівнів;
- реалізувати прототип гри на Unity;
- розробити декілька алгоритмів генерації з різними характеристиками складності;
- провести порівняння створених рівнів за параметрами складності, плавності та ігрової ефективності.

ОСНОВНА ЧАСТИНА

Розділ 1: Аналіз предметної області та огляд підходів до процедурної генерації

1.1 Автоматична генерація рівнів у комп’ютерних іграх

Процес створення рівнів у комп’ютерних іграх є одним з найголовніших аспектів, що визначає загальну якість, тривалість, реіграбельність та унікальність геймплею. Протягом останніх десятиліть підхід до розробки рівнів зазнав значних змін. Якщо раніше левову частку контенту розробники створювали вручну, то сьогодні все більшої популярності набуває використання процедурної генерації — автоматизованого створення вмісту за допомогою алгоритмів, здатних генерувати унікальні середовища, що не повторюються.

Автоматична генерація рівнів (або procedural level generation, PLG) — це сукупність алгоритмів і методів, які дозволяють формувати архітектуру ігрових просторів із заданими властивостями. У класичному підході ігровий дизайнер розміщує платформи, перешкоди, об’єкти вручну, контролюючи композицію та баланс. У процедурному підході ці елементи розміщуються на основі псевдовипадкових алгоритмів з визначеними правилами.

Серед переваг такого підходу можна виділити[1]:

- Економія людських ресурсів: не потрібно витрачати час на розробку великої кількості рівнів вручну;
- Масштабованість: можна створити майже необмежену кількість унікальних рівнів;
- Варіативність геймплею: кожне проходження гри може бути іншим;
- Інтеграція адаптивних механік: рівень може підлаштовуватися під навички гравця;
- Покращення користувацького досвіду: за рахунок непередбачуваності та новизни.

У більшості сучасних мобільних та інді-ігор процедурна генерація рівнів стала стандартною практикою. Вона особливо ефективна в таких жанрах, як endless runner, platformer, roguelike, sandbox та tower defense. Наприклад, у грі Subway Surfers рівень генерується у режимі реального часу — гра постійно створює нові ділянки траси попереду гравця, імітуючи нескінченний ігровий простір (Див. рис. 1.1). Такий підхід дозволяє суттєво зменшити обсяг збережених даних і забезпечити високу реіграбельність без потреби в ручному створенні кожного фрагмента середовища.

Рисунок 1.1 – Гра Subway Surfers

1.2 Огляд підходів до створення процедурного контенту в іграх

Процедурна генерація контенту (Procedural Content Generation, PCG) охоплює широкий спектр методів створення динамічного вмісту, що не обмежується лише рівнями. Вона включає також генерацію середовищ, персонажів, предметів, ворогів, сюжетів, звуків та навіть правил гри. Метою PCG є надання гравцеві унікального досвіду, який неможливо досягнути при класичному ручному підході.

Основні категорії процедурного контенту [2]:

- Просторовий контент — карти, лабіринти, кімнати, рівні, траси, печери, архітектурні споруди;
- Ігрові об’єкти — вороги, NPC, пастки, ресурси, бонуси;
- Предмети та спорядження — зброя, броня, артефакти з випадковими властивостями;
- Сценарії та квести — варіативні сюжетні лінії, завдання, події;
- Аудіо-візуальний контент — текстури, моделі, музика, звуки.

Процедурна генерація дозволяє вирішувати такі завдання:

- Забезпечення реіграбельності (гравець повертається, бо гра щоразу нова);
- Створення великих світів без потреби у великій кількості ресурсів;
- Реалізація адаптивності, коли гра змінюється залежно від дій або рівня гравця;
- Оптимізація пам’яті та зберігання: замість збереження кожного рівня зберігається лише алгоритм і параметри генерації.

У таблиці нижче наведено приклади застосування процедурної генерації в різних жанрах:

Жанр гри	**Тип процедурного контенту**	**Приклади ігор**
Roguelike	Dungeon-карти, вороги, лут, події	The Binding of Isaac, Hades, Spelunky
Sandbox	Світ, біоми, структури, ресурси	Minecraft (Див рис 1.2), Terraria, Starbound
Endless runner	Нескінченні траси, перешкоди, бонуси	Subway Surfers, Temple Run

RPG	Локації, персонажі, спорядження, квести	Diablo II, Torchlight, Path of Exile
Симулятори виживання	Генерація світу, клімату, тварин, рослин	Don’t Starve (Див рис 1.3), Valheim, Rust
Космічні симулятори	Планети, зоряні системи, біоми, місії	No Man’s Sky, Elite Dangerous

Рисунок 1.2 – гра Minecraft

Рисунок 1.3 – карта гри Don’t starve

1.3 Аналіз гри Hill Climb Racing як референсу

1.3.1. Ігрова механіка та базові компоненти геймплею

Hill Climb Racing [3]— це 2D-аркадна гра, відома своїм простим, але захопливим геймплеєм (Див. рис. 1.4). Гравець керує автомобілем, який має проїхати якнайдалі, долаючи пагорби, підйоми та спуски. Управління здійснюється всього двома кнопками — газ і гальмо. Однак під час гри необхідно враховувати фізику: інерцію, обертання автомобіля у повітрі, зчеплення з поверхнею та інші нюанси. Важливу роль відіграє баланс між прискоренням і стабільністю — надмірне натискання газу може призвести до перекидання, а гальмування — до втрати швидкості або зупинки.

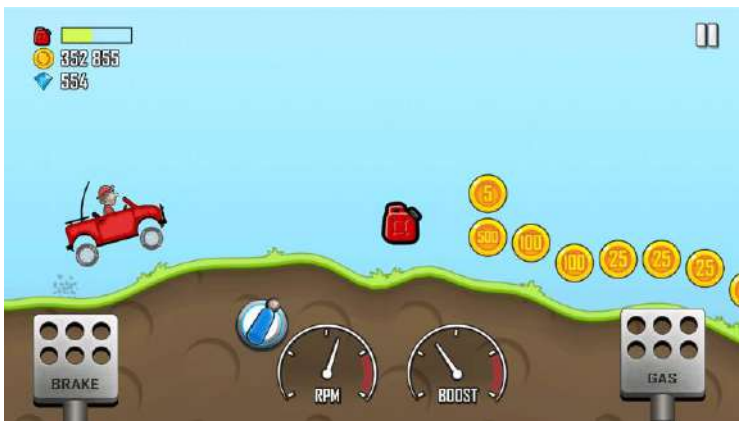

Рисунок 1.4 – геймплей гри Hill Climb Racing

Крім базового управління, у грі присутні:

- Система пального — гравець повинен збирати каністри, щоб продовжити рух.
- Монети та апгрейди — зароблені монети можна витратити на поліпшення транспорту або розблокування нових машин й локацій.
- Багато видів транспортних засобів — кожен з власними параметрами фізики (маса, потужність, пружність підвіски).
- Фіксовані траси — кожен рівень має наперед заданий, вручну створений ландшафт.

1.3.2. Стиль і тип генерації рельєфу

Важливою особливістю Hill Climb Racing є те, що рівні створюються вручну — вони мають статичний дизайн і не змінюються при повторному проходженні. Це означає, що трасу можна вивчити напам’ять, а всі підйоми, спуски та розміщення пального будуть завжди на одному й тому ж місці. Такий підхід спрощує балансування, дозволяє розміщувати перешкоди з точністю та гарантує контрольований рівень складності.

Проте для багатьох гравців, включно зі мною як розробником, цей аспект є значним обмеженням. Незмінність рівнів призводить до швидкої втрати інтересу, оскільки гравець з часом знає, що його чекає далі. Відсутність відчуття новизни і несподіванок зменшує реіграбельність і ускладнює тривале залучення користувача.

Саме ця особливість і стала основним мотивом для розробки автоматичного генератора рівнів у моєму дипломному проєкті. Метою було створити систему, яка щоразу генерує унікальний рельєф, завдяки чому гравець не знає, який поворот або підйом чекає його далі, — тим самим підвищуючи як складність, так і інтерес до гри.

1.3.3. Рівні, транспорт, фізика, управління

Hill Climb Racing пропонує гравцю широкий вибір транспорту — від стандартного джипа до автобусів, тракторів, танків і навіть місяцеходів (Див. рис. 1.5). Кожен тип транспорту має унікальну фізичну модель: відрізняється масою, стійкістю, прискоренням, витратою пального тощо. Гравець може покращувати техніку, підвищуючи характеристики двигуна, підвіски, шин, а також розширювати паливний бак.

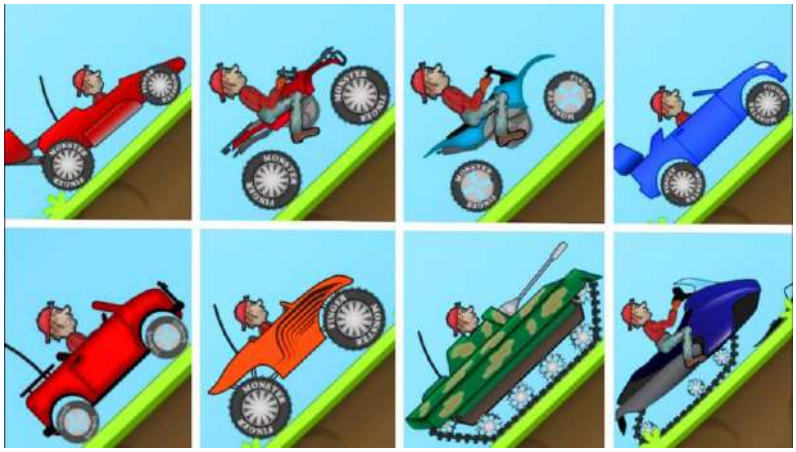

Рисунок 1.5 – транспортні засоби гри

Фізика гри, хоч і спрощена для мобільних пристроїв, враховує багато параметрів: реакцію підвіски на рельєф, силу обертання під час стрибків, інерцію, зчеплення з землею тощо. Це робить гру не простою аркадою, а мініатюрним фізичним симулятором із прикладним геймдизайном.

Управління гранично просте, але за ним ховається стратегічна глибина: вибір моменту для стрибка, нахилу в повітрі, економія пального, адаптація до типу рельєфу — все це потребує уваги і навичок.

1.3.4. Що було взято за основу в дипломному проєкті

Під час аналізу Hill Climb Racing для дипломної роботи, я звернув особливу увагу на сильні сторони гри — інтуїтивне керування, приємну фізику, зрозумілу петлю прогресу, різноманіття транспорту. Водночас найбільше враження на мене

справив саме недолік генерації рівнів. Враховуючи, що траси залишаються незмінними, гравець втрачає мотивацію з часом.

У результаті, для дипломного проєкту я:

- Зберіг основну концепцію 2D-фізичної аркади з поступовим ускладненням;
- Використав управління через дві кнопки (газ/гальмо);
- Створив робочий білд цієї гри на Android;
- Розробив модуль автоматичної генерації рельєфу, що формує новий рівень під час руху.

Цей останній компонент став головною інновацією: тепер кожна спроба — це нова траса, з непередбачуваними схилами, платформами, перешкодами.

Розділ 2. Аналіз алгоритмів процедурної генерації рівнів

2.1. Загальний огляд процедурної генерації контенту

2.1.1. Поняття процедурної генерації

Процедурна генерація [4](англ. Procedural Generation) — це метод створення контенту на основі алгоритмів, які використовують певні математичні або логічні правила. На відміну від ручного дизайну, де кожен об’єкт, рівень або елемент створюється індивідуально дизайнером, ПГ дозволяє формувати велику кількість унікального контенту без необхідності створення кожного варіанту вручну. (Див. рис. 2.1)

Процедурна генерація може мати як повністю випадкову, так і керовану (детерміновану) природу. В основі такого підходу зазвичай лежить генератор випадкових чисел або спеціальні алгоритми (наприклад, шумові функції), що дають змогу контролювати складність, масштаб та форму згенерованого об’єкта. У випадку процедурної генерації рівнів — це означає, що ігрове середовище формується динамічно, часто під час гри, що гарантує варіативність і унікальність досвіду гравця.

Застосування ПГ дозволяє вирішити низку проблем: підвищення реіграбельності, зменшення витрат на розробку рівнів, оптимізація збереження даних (наприклад, збереження лише сиду — числа, з якого формується цілий світ), а також підтримка елементу несподіванки в геймплеї.

Рисунок 2.1 – приклад процедурної генерації рівня в 3D грі

2.1.2. Сфери застосування у відеоіграх

Процедурна генерація має широке застосування у відеоігровій індустрії, особливо у тих жанрах, де важлива реіграбельність, дослідження або нескінченний контент. Найчастіші сфери включають:

- Генерація рівнів (ландшафтів, підземель, карт) — зокрема в іграх жанру roguelike (The Binding of Isaac, Spelunky), виживання (Minecraft, Terraria (Див. рис. 2.2)), гонках (Distance, Trackmania) тощо.
- Створення світів та галактик — наприклад, у No Man's Sky, де за допомогою процедурної генерації створено понад 18 квінтильйонів планет.

Рисунок 2.2 – карта гри Terraria

- Генерація об’єктів і сутностей — включаючи предмети, ворогів, NPC (як-от у Diablo), що значно розширює контент без значних витрат на художнє наповнення.

- Генерація анімацій або музики — хоча менш поширено, цей підхід дозволяє створювати унікальні візуальні ефекти, рухи, навіть звуки.
- Інтерактивні сюжети та квести — де сценарії або місії комбінуються випадковим чином, створюючи динамічний наратив.

Сучасні рушії (Unity, Unreal Engine, Godot) мають у своїх інструментах підтримку процедурної генерації через API, плагіни або можливість створення кастомних генераторів, що робить цей підхід зручним навіть для інді-розробників.

2.2. Класифікація підходів до генерації рівнів

Процедурна генерація ігрових рівнів охоплює широкий спектр методів, які можуть суттєво відрізнятись за принципами побудови, ступенем випадковості, складністю реалізації та кінцевим результатом. Для систематизації існуючих рішень доцільно виділити кілька основних категорій алгоритмів, які найчастіше застосовуються у сучасному геймдеві. У цьому підрозділі наведено класифікацію підходів до процедурної генерації ігрового простору, з прикладами та характеристиками кожного класу.

2.2.1. Генерація за правилами (Rule-Based Generation)

Цей підхід ґрунтується на чітко заданих правилах, які керують процесом побудови рівня. В основі — набір умов або обмежень (constraints), що описують, як елементи повинні розміщуватися, з’єднуватися або змінюватися. Часто використовується в системах логіки гри, головоломках, покрокових RPG, де кожен об'єкт повинен задовольняти певні умови для розміщення.

Переваги[5]:

- Високий рівень контролю над результатом;
- Простота відлагодження та валідації;
- Відсутність непередбачуваних ситуацій.

Недоліки:

- Обмежена варіативність;
- Велика кількість ручної роботи для створення правил.

Приклади:

- Головоломки, де предмети не повинні перекриватись;
- Сіткові рівні з певними правилами з’єднання кімнат (наприклад, в Dungeons of Dredmor).

2.2.2. Генерація на основі шаблонів (Template-Based Generation)

Цей підхід використовує заздалегідь підготовлені шматки рівня — шаблони [6], або «prefabs», які комбінуються під час генерації. Наприклад, гра може мати бібліотеку з 50 блоків рівня (кімнат, платформ, перешкод), які з’єднуються випадковим або напівкерованим чином.

Рисунок 2.3 – карта гри Binding of Isaac

Переваги:

- Швидке створення якісного контенту;
- Висока керованість художнім стилем;
- Можливість ручної перевірки кожного блоку.

Недоліки:

- Менша варіативність (кількість унікальних комбінацій обмежена набором шаблонів);
- Потрібно створювати багато шаблонів вручну.

Приклади:

- Spelunky — кімнати створені вручну, але порядок їх появи — випадковий;
- Dead Cells [7] — модулі рівня з'єднуються згідно з певною логікою.
- Binding of Issac [8] — рівень будується з готових кімнат, що з'єднуються випадковим способом. (Див. рис. 2.3).

2.2.3. Просторові алгоритми (Spatial Algorithms)

Просторові методи побудови рівнів базуються на геометричних розрахунках або розбитті простору. У цьому підході рівень розглядається як площина або об'єм, який поступово заповнюється ігровими елементами, кімнатами, коридорами або об'єктами.

До найвідоміших алгоритмів належать:

- Binary Space Partitioning (BSP) — простір рівня ділиться рекурсивно на підобласті, які потім слугують кімнатами.
- Cellular Automata — клітинна система, де кожна клітинка змінює стан за правилами, як у моделюванні печер.
- Room-and-Corridor — генерація випадкових кімнат і з'єднання їх коридорами.

- Maze Generation — генерація лабіринтів (DFS, Prim, Kruskal).

Переваги:

- Дає змогу створювати логічно зв’язані структури;
- Добре підходить для dungeon-style ігор;
- Підтримка топологічної цілісності.

Недоліки:

- Може потребувати складної логіки перевірки з’єднаності;
- Важче досягти природності форми.

Приклади:

- Diablo II — BSP для генерації підземель;
- Brogue — Cellular Automata для печер.

2.2.4. Алгоритми, керовані даними (Data-Driven Generation)

У цьому підході структура рівня створюється на основі набору даних або зовнішніх параметрів. Це можуть бути JSON-файли, XML, статистика гравця, музика, мапа місцевості тощо. Ідея полягає у створенні контенту, який адаптується до зовнішнього середовища.

Переваги:

- Динамічна адаптація під гравця або ситуацію;
- Можна інтегрувати з іншими системами (аналітика, телеметрія).

Недоліки:

- Потрібна складна обробка даних;
- Важче забезпечити якість результату.

Приклади:

- Soundodger+ — рівні генеруються за музикою;
- Adaptive game systems — підлаштовування рівня під навички гравця.

2.2.5. Алгоритми навчання (Machine Learning-Based Generation)

Методи машинного навчання дозволяють навчити модель створювати нові рівні на основі прикладів.

Це можуть бути:

- Генеративні нейромережі (GANs) [9];
- Sequence-to-sequence моделі;
- Reinforcement Learning для оптимізації структури рівня [10].

Переваги:

- Висока здатність до генерації нових ідей;
- Можливість навчання на основі існуючих рівнів;
- Автоматичне виявлення прихованих шаблонів.

Недоліки:

- Необхідність великого датасету;
- Важко передбачити результат;
- Проблеми з валідацією (рівень може бути непридатним для гри).

Приклади:

- Mario AI [11] — створення нових рівнів на основі карт Super Mario (Див. рис. 2.4);
- PCGML (Procedural Content Generation via Machine Learning) — академічні експерименти.

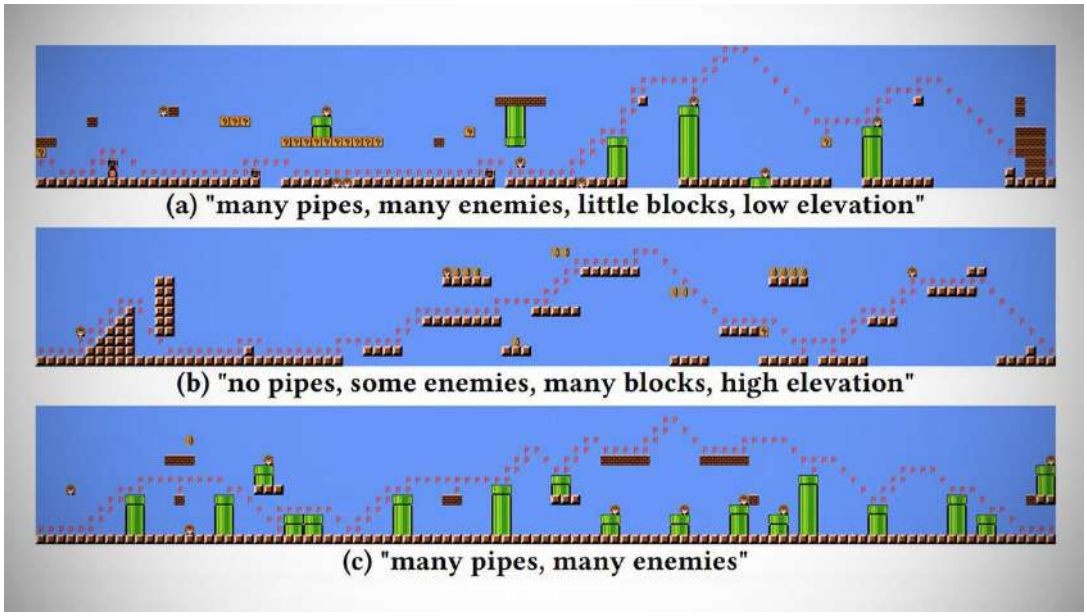

Рисунок 2.4 – генерація рівнів для Super Mario за допомогою ШІ

2.2.6. Гібридні методи

У реальних проєктах розробники часто комбінують кілька підходів для досягнення кращого балансу між варіативністю, контролем та продуктивністю. Наприклад:

- шаблони + шум;
- BSP + Cellular Automata;
- rule-based + GAN postprocessing.

Переваги:

- Гнучкість у налаштуванні результату;
- Поєднання найкращих властивостей окремих методів.

Недоліки:

- Зростання складності системи;
- Потрібна додаткова логіка об’єднання методів.

Приклади:

- No Man’s Sky [12] — шаблони + шум + атрибути з бази даних;

•Minecraft [13] — шум + rule-based генерація структур (Див. рис. 2.5).

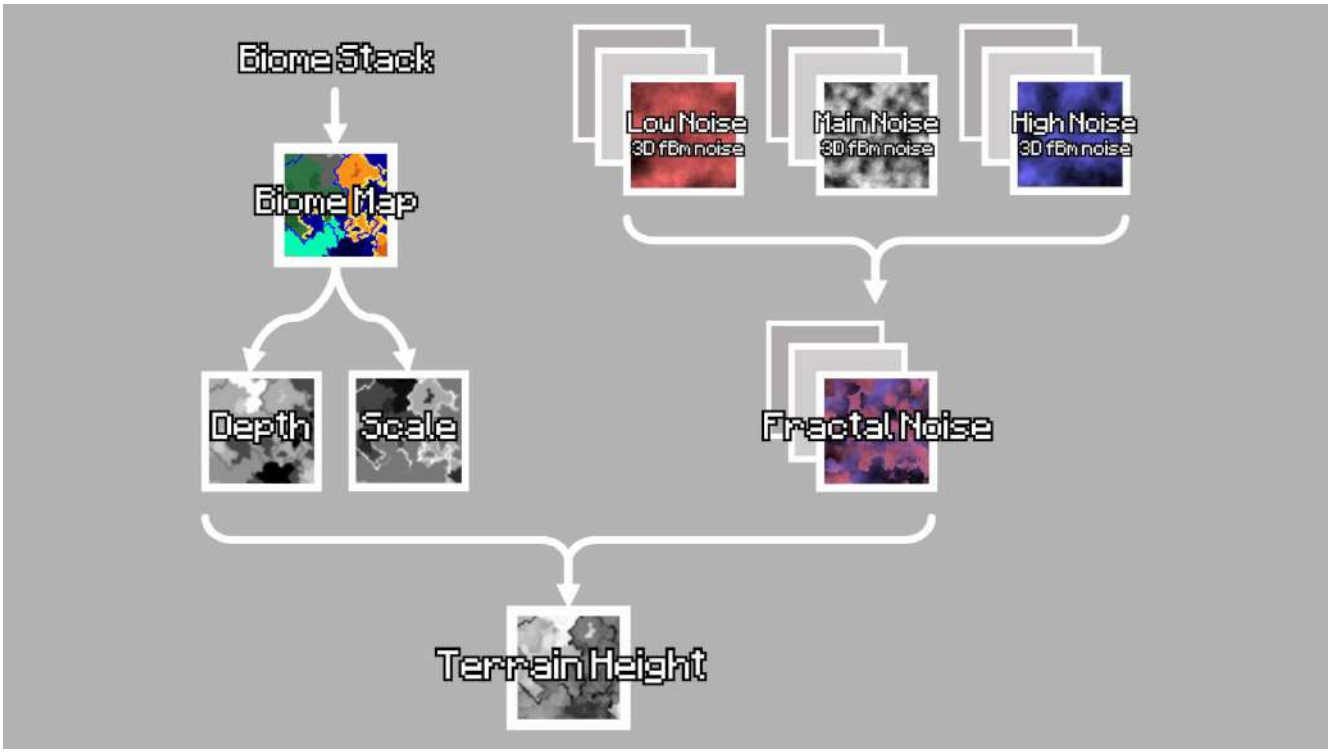

Рисунок 2.5 – генерація рівнів в грі Minecraft

Розділ 3. Розробка гри-клону Hill Climb Racing з підтримкою автоматичної генерації рівнів

3.1. Інструменти розробки

Під час створення гри-клону Hill Climb Racing із процедурною генерацією рівнів було обрано ігровий рушій Unity, який забезпечив зручне середовище для реалізації ключових функціональних можливостей проєкту. Unity надає широкий набір вбудованих інструментів, що дозволяють ефективно створювати двовимірні ігри, працювати з фізикою, анімацією, UI-компонентами, а також реалізовувати генерацію рівнів у реальному часі (Див. рис. 3.1).

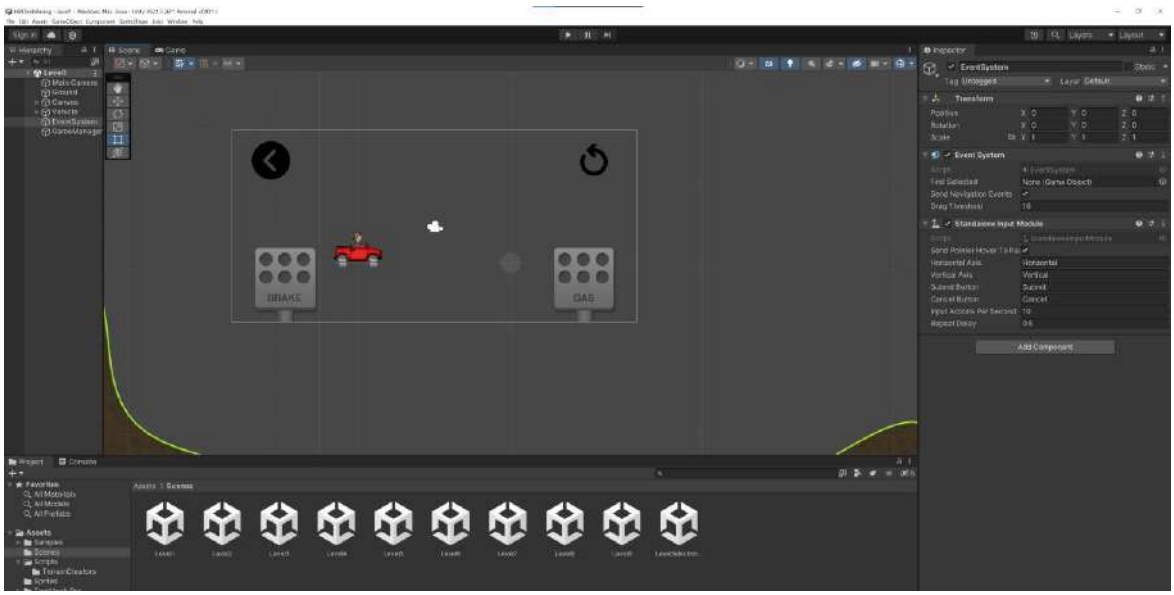

Рисунок 3.1 – інтерфейс користувача в Unity

Серед ключових переваг Unity я б виділив підтримку мови C#, наявність величезної маси документації, гайдів, туторіалів, та дуже зручний візуальний редактор сцен.

Багато ролі в моєму проєкті зіграли такі інструменти Unity:

- Sprite Shape Controller [14] — компонент, що дозволяє створювати гнучкий контур траси. Він став основою для генерації рівнів: на основі масиву точок формується рельєф, який може бути як плавним, так і різко змінним, залежно від обраного алгоритму генерації.

- Edge Collider 2D [15] — колайдер, що додається до контурів Sprite Shape і забезпечує фізичну взаємодію машини з трасою. Він автоматично адаптується під форму рельєфу.
- WheelJoint2D [16] — спеціалізований фізичний компонент, який дозволяє прикріпити колеса до корпусу машини з дотриманням певної моделі підвіски. Цей компонент імітує поведінку реального з’єднання, де колесо не просто "приклеєне", а має можливість вертикального руху завдяки підвісці. У моєму проєкті я використав WheelJoint2D для з'єднання кожного колеса з корпусом автомобіля. Це дозволило створити більш реалістичну модель руху — під час наїзду на горби чи падіння з висоти колеса амортизують, створюючи враження ваги та інерції.
- Rigidbody2D [17] — базовий компонент для будь-якого фізичного об’єкта в Unity. У моєму проєкті RigidBody2D було навішено як на корпус машини, так і на обидва колеса. Це забезпечило повноцінну фізичну симуляцію — маса, сила тяжіння, сила від мотора, інерція, зіткнення з рельєфом тощо.
- Polygon Collider 2D [18] — колайдер складної форми, який я використав для корпусу машини. Це дозволило точно відтворити форму транспортного засобу й забезпечити коректну взаємодію з поверхнею траси.
- Circle Collider 2D [19] — простий круглий колайдер (Див. рис. 3.2), який я застосував до коліс. Завдяки своїй формі він забезпечує плавне й стабільне обертання та взаємодію з нерівностями рельєфу.

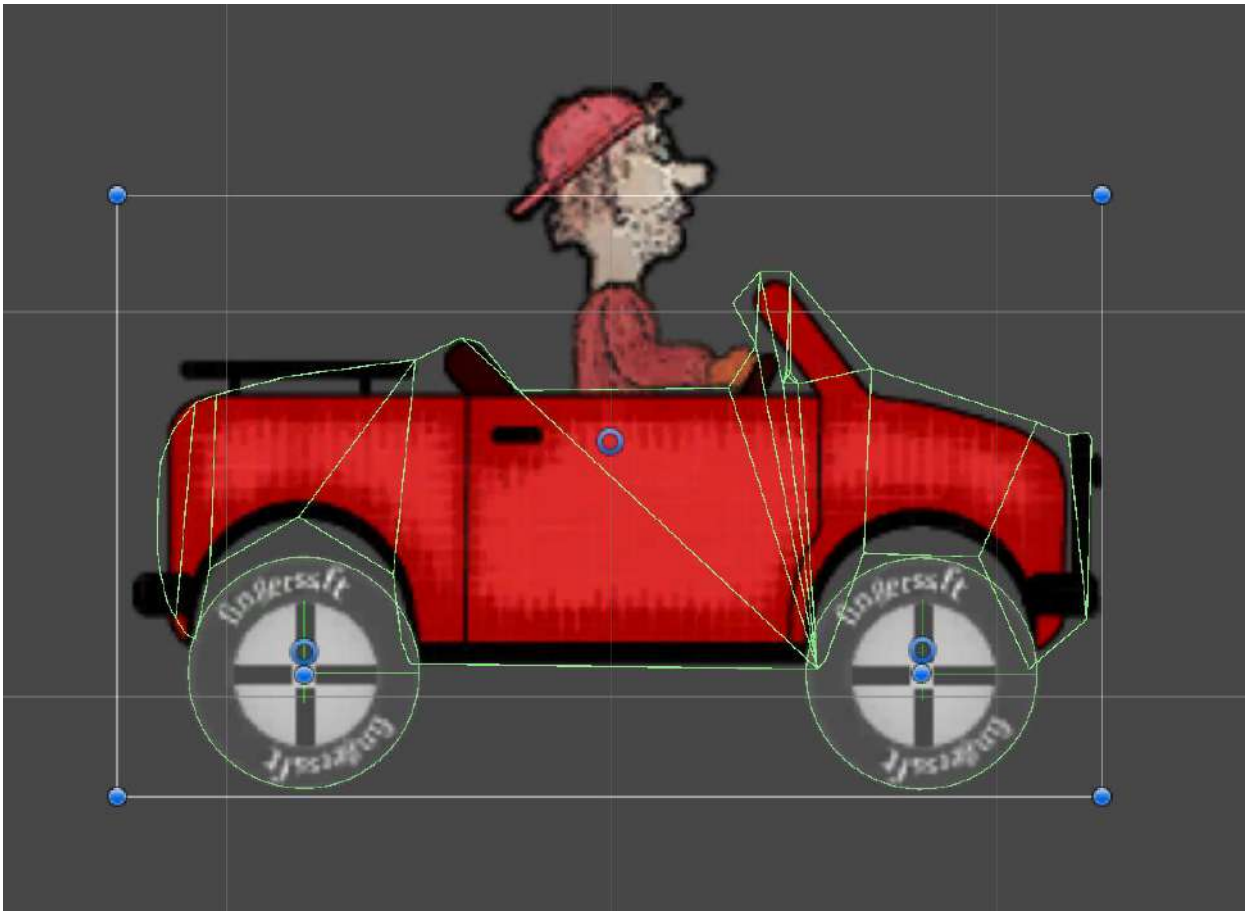

Рисунок 3.2 – колайдери на машині та колесах

3.2. Архітектура гри та структура проєкту

Головна сцена — вибір рівня

Після запуску гри гравець потрапляє на головну сцену вибору рівня, де він може обрати один із доступних семи рівнів. Сцена реалізована у вигляді зручного UI-інтерфейсу з кнопками рівнів від 1 до 9. Перший рівень фіксований (ручної побудови), інші — процедурно згенеровані за різними алгоритмами (Див. рис. 3.3).

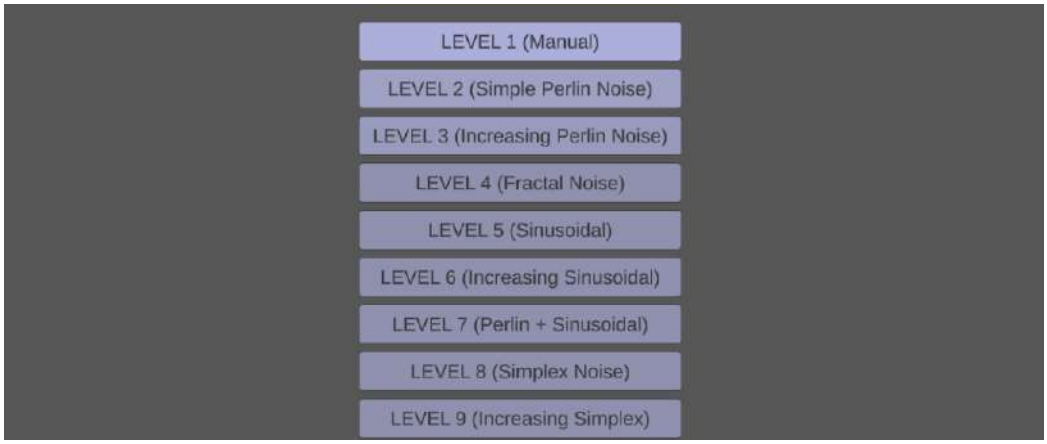

Рисунок 3.3 – сцена вибору рівня

Користувацький інтерфейс (UI) на ігровій сцені

Після вибору рівня гравець переходить на ігрову сцену. Тут відображається мінімалістичний, але функціональний інтерфейс (Див. рис. 3.4):

- Дві великі кнопки-педалі: газ (права) і гальмо/назад (ліва);
- Кнопка рестарту — перезапускає поточну сцену;
- Кнопка виходу — повертає до меню вибору рівня.

UI адаптований як для клавіатури, так і для сенсорного управління на мобільних пристроях.

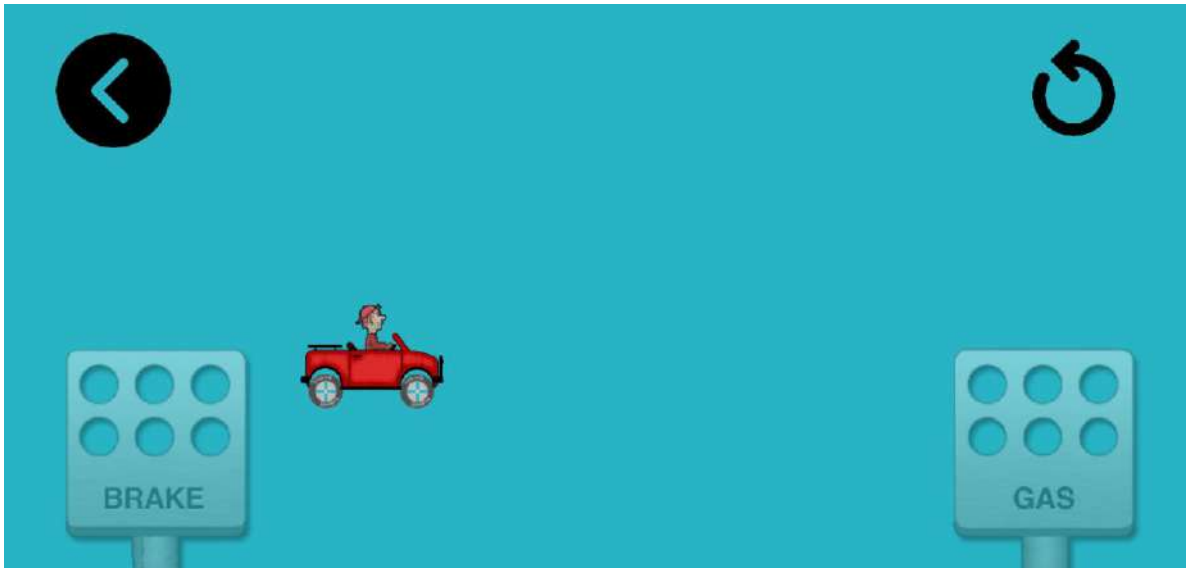

Рисунок 3.4 – користувацький інтерфейс гри

Логіка руху машини

Центральним елементом ігрового процесу є автомобіль, що керується гравцем. Його фізична поведінка реалізована через компонент CarController, який оперує трьома Rigidbody2D: для корпусу та двох коліс. Колеса приєднані до корпусу через WheelJoint2D, що дозволяє моделювати підвіску з пружністю та амортизацією.

Код CarController розділяє логіку керування на два типи:

1. Клавіатурне керування (PC) — за допомогою Input.GetAxis("Horizontal"), що дозволяє рухатись вперед і назад.
2. Мобільне керування (touch) — через метод CarInputMobile(int dir), який прив’язаний до кнопок педалей на екрані.

Слідування камери за машиною

Щоб гравець завжди бачив транспортний засіб у кадрі, використано скрипт CameraController, який постійно слідкує за позицією машини:

```
transform.position = target.position + offset;
```

GameManager

Для керування загальним станом гри використовується GameManager — окремий скрипт, який відповідає за перезапуск сцени та вихід у меню.

Наприклад, для перезапуску гри використовується метод:

```
public void RestartLevel()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

3.3. Реалізація рівнів

У грі реалізовано дев’ять типів рівнів, кожен із яких відрізняється способом генерації рельєфу та складністю проходження, що дозволяє створити різноманітний ігровий досвід.

Рівень 1 створений вручну та має фіксовану трасу з заздалегідь продуманим рельєфом. Цей рівень слугує базою для порівняння з іншими, процедурно згенерованими варіантами (Див. рис. 3.5).

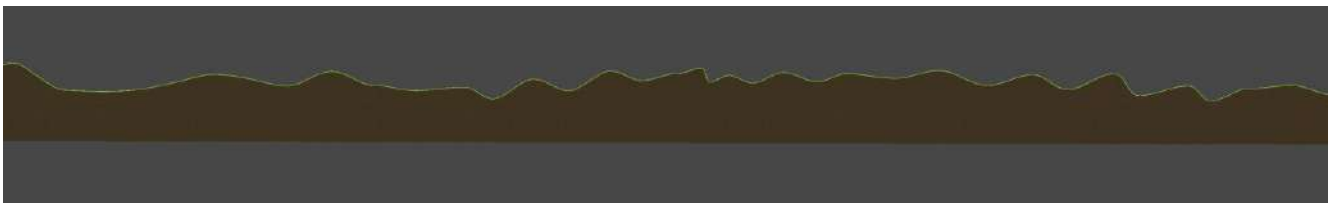

Рисунок 3.5 – рівень 1

Рівень 2 генерується за допомогою класичного Перлинного шуму (Perlin Noise), що створює плавні природні коливання рельєфу. Такий підхід дозволяє отримати рівень з природнім вигином і помірною складністю (Див. рис. 3.6).

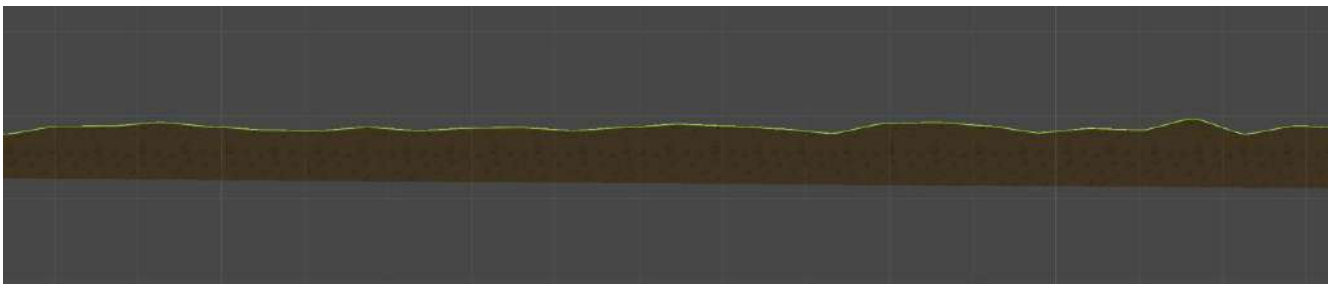

Рисунок 3.6 – рівень 2

Перлинний шум (Perlin Noise) [20] — це алгоритм генерації згладженого псевдовипадкового шуму, який широко використовується у процедурній генерації ландшафтів, текстур, хмар та інших природних візерунків. Алгоритм був розроблений Кеном Перліном у 1983 році для створення більш природних, плавних і візуально приємних форм у комп’ютерній графіці.

Основна ідея полягає в тому, що простір заповнюється ґраткою з фіксованими векторами градієнтів. Для кожної точки, де потрібно обчислити шум, алгоритм:

- Визначає положення точки всередині найближчої осередкової ґратки (наприклад, квадратної у 2D).

- Визначає вектори напрямку (градієнти) у вершинах цього осередку. Вектори фіксовані або обчислюються псевдовипадково, але є детермінованими на основі координат.
- Обчислює вектори від кожної вершини осередку до точки, для якої рахується шум.
- Обчислює скалярний добуток між градієнтними векторами і векторами до точки. Ці добутки показують, наскільки напрям вектора «підсилює» або «послаблює» шум у цій точці.
- Виконує інтерполяцію між отриманими значеннями за допомогою спеціальної згладжувальної функції (зазвичай використовується функція fade — кубічна або п’ятикратна функція для згладження переходів між значеннями).

Завдяки такому підходу Перлин шум генерує згладжену поверхню, де значення змінюються плавно, без різких стрибків. Результат має вигляд природної, органічної хвилястості, що ідеально підходить для моделювання природних форм.

У Unity реалізовано Перлинний шум [21], який доступний через функцію Mathf.PerlinNoise(x, y). Вона приймає дві координати x і y, і повертає значення в діапазоні від 0.0 до 1.0. Алгоритм не є повністю випадковим: значення змінюються поступово, формуючи хвилеподібну структуру, що ідеально підходить для генерації рельєфів.

В моєму проєкті висота кожної наступної точки вираховується за такою формулою:

$$y_i = PerlinNoise(x_i * k, offset) * h$$

де:

- y_i – висота точки рельєфу;
- x_i - координата точки по осі X;

- *k* – масштаб;
- *offset* - випадковий зсув по Y, щоб уникнути однакових шаблонів;
- *h* — коефіцієнт амплітуди.

Практично це реалізовано так:

- Береться Sprite Shape Controller (Див. рис. 3.7), який містить лише чотири стартові точки;
- Для кожної нової точки генерується висота через Mathf.PerlinNoise(x * noiseScale, offset) з випадковим offset;
- Кожна точка додається у spline;
- В кінці задається фінальна точка, яка завершує трасу.

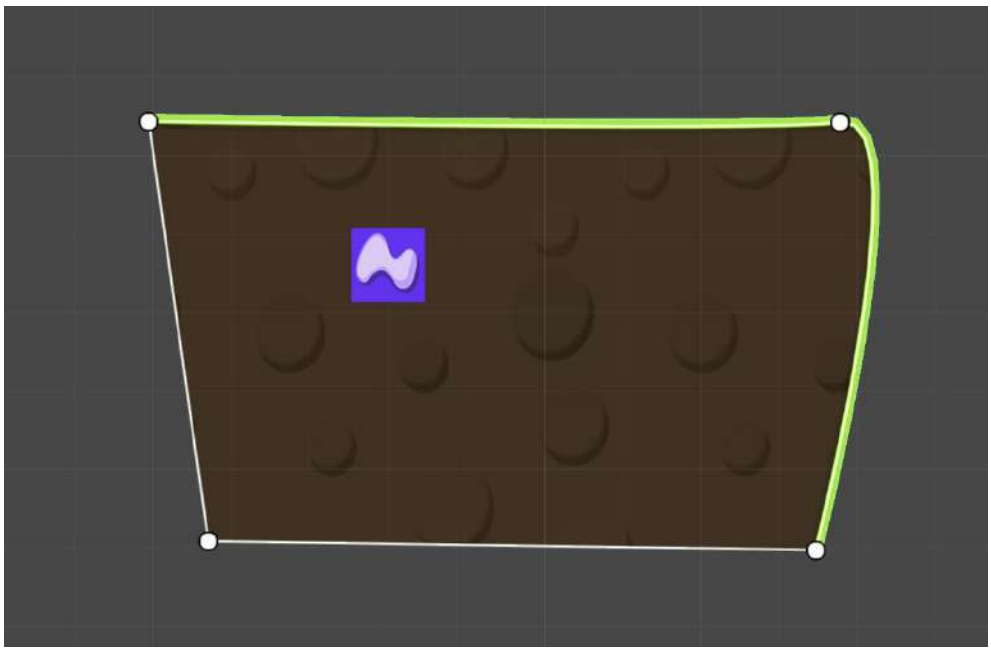

Рисунок 3.7 – базовий Sprite Shape Controller

Рівень 3 також використовує Перлинний шум, однак із динамічною складністю — амплітуда та масштаб шуму поступово зростають під час генерації (Див. рис. 3.8). Це означає, що спочатку рельєф є відносно рівним і простим для гравця, але з просуванням уперед він стає все більш складним, нерівномірним та «агресивним».

Рисунок 3.8 – рівень 3

Формально, модифікований рівень використовує ті ж самі базові властивості Перлинного шуму, однак вводить змінні коефіцієнти масштабу та амплітуди:

$$y_i = y_{base} + ([PerlinNoise(x_i * k_i, offset) - 0{,}5] * 2) * h_i$$

$$k_i = k_0 + i * \Delta k$$

$$h_i = h_0 + i * \Delta h$$

де:

- y_i – висота точки на координаті x_i;
- k_i - поточний масштаб Перлинного шуму (що зростає з кожною ітерацією);
- h_i - поточна амплітуда шуму (також зростає);
- $\Delta k, \Delta h$ - малі прирости, які задають швидкість ускладнення рельєфу;
- *offset* випадковий зсув по другій координаті.

В скрипті, з кожною новою точкою збільшується currentNoiseScale та currentHeightMultiplier, що дозволяє формувати рівень, який розвивається.

Рівень 4 реалізовано за допомогою фрактального шуму [22], що є розширенням класичного Перлинного шуму, при якому кілька «шарів» шуму з різною частотою та амплітудою накладаються один на одного (Див. рис. 3.9). Це дозволяє створити більш деталізований, природний і варіативний рельєф, у порівнянні з використанням лише одного шару шуму. Також, в цьому рівні реалізовано поступове збільшення складності.

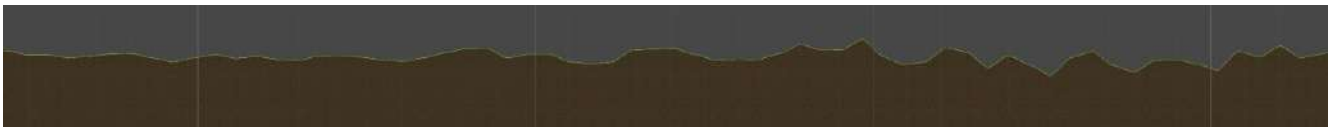

Рисунок 3.9 – рівень 4

Фрактальний шум (англ. Fractal Noise) — це метод поєднання декількох октав Перлинного шуму з різними масштабами (частотами) та амплітудами. Кожна наступна октава має вищу частоту (менший масштаб) і меншу амплітуду, створюючи дрібні деталі, які додають складності та реалістичності.

Формула фрактального шуму виглядає так:

$$f(x) = \frac{1}{A}\sum_{i=0}^{n-1} PerlinNoise\left(x * scale * \lambda^i\right) * p^i$$

де:

- n - кількість октав (рівнів шуму);
- λ *(lacunarity)* - коефіцієнт зростання частоти на кожну октаву;
- p *(persistence)* - коефіцієнт зменшення амплітуди;
- *scale* - базовий масштаб шуму;
- $A = \sum_{i=0}^{n-1} p^i$ - нормалізуючий множник

Lacunarity керує тим, як швидко зростає частота з кожною новою октавою, а persistence — як швидко зменшується амплітуда. При правильному налаштуванні параметрів рельєф стає більш реалістичним, але все ще контрольованим.

В проєкті я реалізував це наступним чином:

```
float y = baseGroundY + ((GenerateFractalNoise(x, offset, baseScale, octaves, lacunarity,
persistence) - 0.5f) * 2f * heightMultiplier);
```

Рівень 5 генерується за допомогою синусоїдальної функції, що створює регулярні хвилеподібні коливання (Див. рис. 3.10). Цей рівень має більш передбачувану структуру та помірну складність.

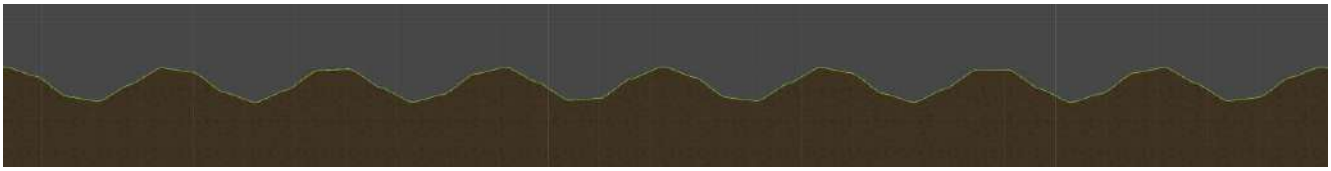

Рисунок 3.10 – рівень 5

Синусоїда є однією з найпростіших і найпоширеніших періодичних функцій у математиці. Вона описується формулою:

$$y(x) = A * \sin(f * x) + y_0$$

де:

- A – амплітуда;
- f – частота;
- x – координата по горизонталі:
- y_0 - вертикальне зміщення, або базовий рівень землі.

В коді я реалізував це таким чином:

```
float y = baseGroundY + Mathf.Sin(x * frequency) * amplitude;
```

Ця функція ідеально підходить для створення регулярного, але плавного рельєфу, що повторюється з певною періодичністю. Такий рельєф не є випадковим, як при використанні шуму Перліна, і саме тому рівень стає більш стабільним і читабельним для гравця.

Рівень 6 є ускладненою версією хвильового рельєфу, реалізованого на рівні 5. У цьому випадку синусоїдальна функція зберігається як основа генерації, однак з кожною наступною точкою поступово зростають амплітуда та частота, що призводить до збільшення складності рельєфу — хвилі стають вищими та щільнішими (Див. рис. 3.11).

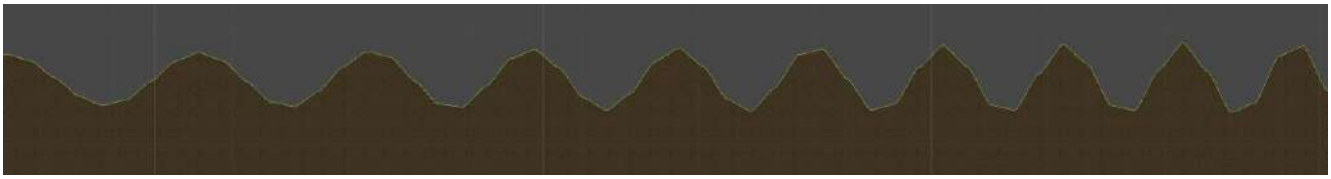

Рисунок 3.11 – рівень 6

Формула, яка використовується для генерації висоти кожної точки, така ж як у класичній синусоїді, однак її параметри змінюються динамічно:

$$y(x) = A(x) * \sin(f(x) * x) + y_0$$

А реалізував я це так:

```
float y = baseGroundY + Mathf.Sin(x * currentFrequency) * currentAmplitude;
```

Параметри *currentAmplitude* та *currentFrequency* на кожній ітерації збільшуються відповідно до таких величин:

```
currentAmplitude += amplitudeIncreaseRate;
currentFrequency += frequencyIncreaseRate;
```

Це означає, що впродовж генерації террейну на початку рівня гравець рухається по плавних хвилях (подібно до рівня 5), але з часом хвилі стають все крутішими і щільнішими, вимагаючи від гравця точнішого контролю та реакції.

Рівень 7 поєднує дві принципово різні моделі генерації ландшафту — регулярну синусоїдальну функцію та псевдовипадковий Перлинний шум. Така комбінація дозволяє отримати природний, але водночас передбачувано-хаотичний рельєф, що виглядає реалістично та динамічно (Див. рис. 3.12). Також, в цьому рівні я накинув поступове збільшення складності.

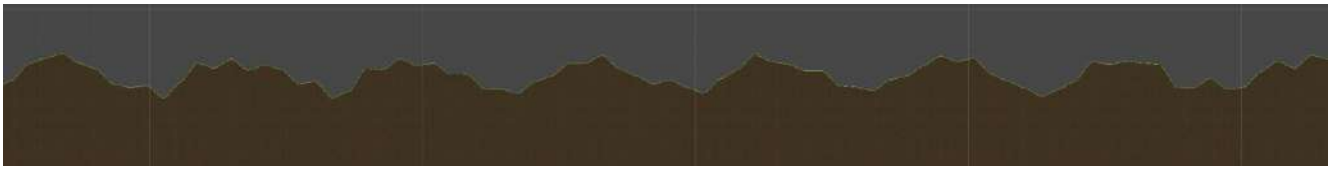

Рисунок 3.12 – рівень 7

Основна формула для обчислення висоти точки на цьому рівні має вигляд:

$$y(x) = A * \sin(f * x) + N(x) + y_0$$

де:

- $A * \sin(f * x)$ – синусоїда;
- $N(x)$ – перлинний шум, який в свою чергу являє собою;
- $N(x) = (Perlin(x * s) - 0.5) * 2 * S3$ масштабом s та силою S;
- y_0 - базовий рівень землі.

Ця формула складається з двох незалежних складових:

1. Синусоїда задає регулярну хвилю, яка створює основну форму рівня.
2. Перлинний шум додає хаотичні, але плавні флуктуації, роблячи кожну хвилю унікальною, ламаючи монотонність.

В Unity я реалізував це як:

```
float sineWave = Mathf.Sin(x * baseFrequency) * baseAmplitude;
float noise = (Mathf.PerlinNoise(x * noiseScale, noiseOffset) - 0.5f) * 2f *
noiseStrength;
float y = baseGroundY + sineWave + noise;
```

Рівень 8 я вирішив реалізувати як Симплексний шум [23] (Див. рис. 3.13). Simplex Noise -це алгоритм згладженого шуму, розроблений Кеном Перліном у 2001 році як вдосконалення класичного Perlin Noise.

Рисунок 3.13 – рівень 8

На відміну від Перлін-шуму, який працює у гіперкубічних ґратках, Simplex Noise працює з симплексами — геометричними примітивами: трикутники (у 2D), тетраедри (у 3D) тощо [24].

Формула вагового внеску кожної вершини симплексу:

$$n_i = t_i^4 * (\overrightarrow{g_i} * \overrightarrow{x_i})$$

де:

- $t_i = 0.5 - ||\overrightarrow{x_i}||$ - функція згасання;

- $\overrightarrow{g_i}$ – градієнтний вектор у вершині;
- $\overrightarrow{x_i}$ - вектор від вершини до точки.

Фінальне значення шуму — це сума всіх внесків $n_0 + n_1 + n_2$ масштабована коефіцієнтом (у реалізації – 70)

Оскільки в Unity був вбудований перлинний шум, то я його використовував як Mathf.PerlinNoise(x, y). Симплексного ж шуму в Unity нема, і тому я вирішив його реалізувати власноруч.

Основні компоненти реалізації включають:

- Таблиця перестановок perm[] використана для генерації псевдовипадкових градієнтів.
- Градієнтні вектори задаються масивом grad3[].
- Визначення симплексу базується на сумі координат із коефіцієнтом F2:

$$F2 = \frac{\sqrt{3} - 1}{2}$$

- Компенсація відбувається за допомогою G2:

$$G2 = \frac{3 - \sqrt{3}}{6}$$

- Кожен із трьох трикутників симплексу вносить свій вклад у фінальне значення.

Використав simplex noise [25] в генерації рівнів я, по суті, так само, як і рівень 2:

$$y_i = SimplexNoise(x_i * k, offset) * h$$

Рівень 9 реалізований так само, як минулий рівень, але зі збільшенням складності протягом гри (аналогічно до рівнів 2 та 3) (Див. рис. 3.14).

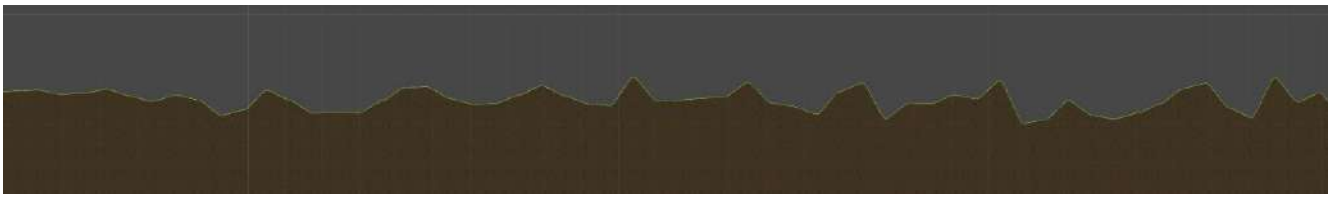

Рисунок 3.14 – рівень 9

3.3. Аналіз результатів

У ході дослідження було реалізовано дев’ять рівнів з різними підходами до генерації ландшафту. Я спробую оцінити ефективність кожного з них за такими критеріями:

- Складність рельєфу – наскільки важко гравцеві долати перешкоди;
- Різноманітність – ступінь варіативності у формі ландшафту;
- Прогнозованість – наскільки легко гравцеві передбачити подальшу форму траєкторії;
- Придатність для нескінченних рівнів – наскільки добре рівень підходить для процедурної генерації у реальному часі.

Загальна характеристика рівнів:

Рівень	1	2	3	4	5	6	7	8	9
Складність	●●●	●●	●●●	●●●	●●	●●●	●●●●	●●●	●●●●
Різноманітність	●●	●●	●●●	●●●●	●	●●	●●●●	●●●●	●●●●
Прогнозованість	●●●●	●●●	●●	●●	●●●●●	●●●●	●●	●●●	●●
Придатність для нескінченних ігор	●	●●●	●●●●	●●●●	●	●●	●●●●	●●●	●●●●

● - умовне позначення рівня за критерієм (від 1 до 5)

Ручна розробка рівню є найпростішою, але найменш адаптивною. Для простих ігр, з обмеженими трасами, гарний варіант, але для нескінченних рівнів я б не радив використовувати.

Перлинний та симплексний шуми – гарні варіанти розробки. Самі по собі вони є одноманітними, але якщо їх використовувати, наприклад, з динамічною складністю, то вийде досить непоганий і цікавий рівень.

Фрактальний шум, на мою думку є найцікавішим. В рівні присутні як плавні та прості ділянки, так і складні, з різкими перепадами, та схилами.

Синусоїда – гарний спосіб демонстрації роботи генератора рівня, але я б не використовував її без комбінацій (наприклад, з якимось шумом).

ВИСНОВКИ

У даній кваліфікаційній роботі було розглянуто й реалізовано підхід до автоматичної генерації рівнів у 2D-грі на рушії Unity, що базується на використанні математичних функцій та шумових алгоритмів, зокрема різних шумів, синусоїдальних функцій та їх комбінацій.

У теоретичній частині роботи було проаналізовано сучасні методи процедурної генерації ігрових рівнів. Розглянуто переваги такого підходу для мобільних аркадних ігор, серед яких — економія пам’яті пристрою, забезпечення різноманітності контенту, а також можливість створення нескінченних рівнів, що не повторюються. Окрема увага була приділена порівнянню різних математичних моделей генерації рельєфу: від базових функцій шуму до комбінацій синуса та перліну, що дозволяють створювати цікаві та поступово ускладнені ігрові ділянки.

У практичній частині було створено ігровий прототип, який імітує механіку відомої гри Hill Climb Racing, однак з повністю автоматизованою генерацією рівня. Для реалізації було використано компонент Sprite Shape Controller, що дозволив будувати ландшафт у реальному часі. Кожен з рівнів у грі відповідає окремому алгоритму генерації, що дозволило наочно продемонструвати вплив обраного методу на геймплей.

У результаті дослідження та розробки було досягнуто поставленої мети — створено робочий прототип гри з автоматичною генерацією рівнів, які є унікальними, неповторними та варіативними. Реалізовані алгоритми дозволяють контролювати складність та плавність рельєфу, що може бути застосовано не лише в аркадах, а й в інших жанрах ігор із процедурною генерацією.

СПИСОК ЛІТЕРАТУРИ

[1] Pros and Cons of Procedural Level Generation
URL: https://schier.co/blog/pros-and-cons-of-procedural-level-generation
[2] Procedural Generation: An Overview
URL: https://kentpawson123.medium.com/procedural-generation-an-overview-1b054a0f8d41
[3] Hill Climb Racing in 2025: Why This Classic Mobile Game Still Rocks
URL: https://itmunch.com/hill-climb-racing-review-2025/
[4] Challenges and Benefits of Procedural Generation in Game Development
URL: https://gamespublisher.com/procedural-generation-challenges-benefits-in-games/
[5] Rule-based Procedural Generation of Item in Role-playing Game
URL: https://www.researchgate.net/publication/320747216_Rule-based_Procedural_Generation_of_Item_in_Role-playing_Game
[6] Constructive Generation Methods for Dungeons and Levels
URL: https://antoniosliapis.com/articles/pcgbook_dungeons.php
[7] The Level Design of Dead Cells
URL: https://deepnight.net/tutorial/the-level-design-of-dead-cells-a-hybrid-approach/
[8] Dungeon Generation in Binding of Isaac
URL: https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/
[9] Learning to Generate Levels From Nothing
URL: https://arxiv.org/pdf/2002.05259
[10] PCGRL: Procedural Content Generation via Reinforcement Learning
URL: https://arxiv.org/pdf/2001.09212
[11] MarioGPT is an AI-Powered Super Mario Bros. Level Maker That Generates New Levels from Text Prompts
URL: https://www.techeblog.com/mariogpt-ai-super-mario-bros-level-maker/
[12] Perlin Noise: The Evolving Algorithm Behind the Diverse Universes of “No Man’s Sky
URL: https://medium.com/@pratyaksh.notebook/perlin-noise-the-evolving-algorithm-behind-the-diverse-universes-of-no-mans-sky-f2cc8ddacd52
[13] The World Generation of Minecraft
URL: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/
[14] Unity Technologies. Unity - Manual: Sprite shape renderer
URL: https://docs.unity3d.com/Manual/sprite/shape-renderer/shape-renderer-landing.html
[15] Unity Technologies. Unity - Manual: Edge Collider 2D
URL: https://docs.unity3d.com/Manual/2d-physics/collider/edge-collider-2d-reference.html
[16] Unity Technologies. Unity - Manual: Wheel Joint 2D fundamentals
URL: https://docs.unity3d.com/Manual/2d-physics/joints/wheel-joint-2d-fundamentals.html
[17] Unity Technologies. Unity - Manual: Introduction to Rigidbody 2D

URL: https://docs.unity3d.com/Manual/2d-physics/rigidbody/introduction-to-rigidbody-2d.html
[18] Unity Technologies. Unity - Manual: Polygon Collider
URL: https://docs.unity3d.com/Manual/2d-physics/collider/polygon-collider-2d-reference.html
[19] Unity Technologies. Unity - Manual: Circle Collider 2D
URL: https://docs.unity3d.com/Manual/2d-physics/collider/circle-collider-2d-reference.html
[20] Perlin Noise: A Procedural Generation Algorithm
URL: https://rtouti.github.io/graphics/perlin-noise-algorithm
[21] Unity Technologies. Unity - Manual: PerlinNoise
URL:https://docs.unity3d.com/6000.1/Documentation/ScriptReference/Mathf.PerlinNoise.html
[22] Procedural Generation - Fractal Noise
URL: https://ninjapretzel.github.io/ProcGen/02fractal.html
[23] Perlin Noise: Implementation, Procedural Generation, and Simplex Noise
URL: https://garagefarm.net/blog/perlin-noise-implementation-procedural-generation-and-simplex-noise
[24] Simplex Noise
URL: https://catlikecoding.com/unity/tutorials/pseudorandom-noise/simplex-noise/
[25] Working with Simplex Noise
URL: https://cmaher.github.io/posts/working-with-simplex-noise/