НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
Комп'ютерних наук
_____ Голуб Б. Л.
(підпис)

"____" _____ 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Програмна система процедурної параметричної генерації віртуальних локацій для гри в жанрі платформер»

Спеціальність F3 – «Комп'ютерні науки»

Гарант освітньої програми

Д.е.н., професор _____ Руденський Р.А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

к.е.н., доцент _____ Назаренко В. А.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Ольшанський Д. В.
(підпис) (ПІБ)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук

_____/ Голуб Б.Л. , доцкнт , к.т.н./
підпис

"____" _____2025 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту
Ольшанському Дмитру Володимировичу

Спеціальність F3 – «Комп'ютерні науки»

1. Тема бакалаврської кваліфікаційної роботи
«Програмна система процедурної параметричної генерації віртуальних локацій для гри в жанрі платформер»

Затверджена наказом ректора НУБіП України від 16.12.2024р. №2246с

2. Термін подання завершеної роботи на кафедру _____
рік, місяць, число

3. Вихідні дані до бакалаврської кваліфікаційної роботи
Створення програмної системи процедурної параметричної генерації віртуальних локацій для гри в жанрі платформер

4. Перелік питань що розглядається:
   1. Системний аналіз предметної області інформаційної системи
   2. Проєктування інформаційного та програмного забезпечення
   3. Розробка прикладного програмного забезпечення
   4. Рекомендації щодо впровадження та експлуатації системи
   5. Висновки

Керівник бакалаврської кваліфікаційної роботи_____/ В.А. Назаренко/
підпис ініціали та прізвище

Завдання прийняв до виконання_____/ Д.В. Ольшанський/
підпис ініціали та прізвище

Дата отримання завдання _____. ____. _____
Рік місяць, число

АНОТАЦІЯ

У бакалаврській кваліфікаційній роботі представлено розробку інструменту процедурної генерації ігрових локацій для двовимірних проєктів у жанрі платформер. Метою роботи є створення універсального плагіну, інтегрованого в середовище розробки Unity, який автоматизує побудову рівнів на основі алгоритмів випадкової прогулянки, шуму Перліна та ступінчастої генерації платформ.

Розроблене рішення містить графічний інтерфейс користувача з підтримкою параметричного налаштування, візуалізацію ігрових карт із застосуванням Tilemap-технології, а також функціонал збереження конфігурацій генерації у вигляді об’єктів типу Scriptable Object у форматі Unity Asset.

У роботі виконано огляд сучасних підходів до процедурної генерації контенту, проаналізовано їхні переваги та обмеження, сформульовано функціональні й нефункціональні вимоги до системи. Проєктування реалізовано з використанням UML-діаграм, включаючи діаграми класів, варіантів використання та активностей. Архітектура плагіну реалізована у вигляді модульної структури та протестована на демонстраційній грі.

Запропонований інструмент орієнтований на потреби інді-розробників і технічних геймдизайнерів, які прагнуть підвищити ефективність створення ігрових рівнів. Плагін легко адаптується до інших жанрів 2D-ігор, масштабований і підтримує розширення шляхом інтеграції нових алгоритмів генерації. Його впровадження дозволяє скоротити витрати на ручне проєктування рівнів, прискорити процес прототипування та покращити загальну гнучкість побудови ігрового середовища.

ABSTRACT

The bachelor's qualification thesis presents the development of a tool for procedural generation of game locations for two-dimensional projects in the platformer genre. The work aims to create a universal plugin integrated into the Unity development environment, which automates the construction of levels based on random walk algorithms, Pearl noise, and stepped generation of platforms.

The developed solution contains a graphical user interface with support for parametric settings, visualization of game maps using Tilemap technology, and the functionality of saving generation configurations in the form of Scriptable Object objects in the Unity Asset format.

The work reviews modern approaches to procedural content generation, analyzes their advantages and limitations, and formulates functional and non-functional requirements for the system. The design is implemented using UML diagrams, including diagrams of classes, use cases, and activities. The plugin architecture is implemented as a modular structure and tested on a demo game.

The proposed tool is aimed at the needs of indie developers and technical game designers who seek to increase the efficiency of creating game levels. The plugin easily adapts to other genres of 2D games, is scalable, and supports expansion by integrating new generation algorithms. Its implementation allows you to reduce the cost of manual level design, speed up the prototyping process, and improve the overall flexibility of building a game environment.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ , СИМВОЛІВ ,ОДИНИЦЬ , СКОРОЧЕНЬ І ТЕРМІНІВ

Плагін (Plug-in) – програмний модуль , що підключається до самої програми та розширює її функціонал

2D (двовимірний) - простір, що визначається двома координатами (X, Y);

3D (тривимірний) – простір , що визначається трьома координатами (X, Y, Z);

LOD (Level of Detail) - технологія візуалізації, що змінює деталізацію об’єктів залежно від відстані до камери.

DOTS (Data-Oriented Technology Stack) - набір технологій від Unity для високопродуктивної роботи з великими обсягами даних.

ECS (Entity Component System) - архітектурний шаблон Unity, що дозволяє розділити дані та логіку, підвищуючи продуктивність.

Blueprint - система візуального програмування в Unreal Engine.

Seed (Seed-значення) - початкове значення для генератора випадкових чисел, що забезпечує повторюваність процедурної генерації.

Inspector - панель Unity, яка дозволяє редагувати властивості об'єктів у середовищі розробки.

Canvas - компонент в Unreal Engine, який використовується для побудови підрівнів або візуалізації UI.

Prefab - попередньо створений шаблон об’єкта в Unity, який можна багаторазово використовувати у сценах.

Tile - одиничний елемент (плитка) в системі Tilemap у Unity, що складає частину рівня.

ScriptableObject - тип об’єкта в Unity, який зберігає дані незалежно від сцени.

Pixel Art - стиль графіки, що базується на піксельних зображеннях.

ВСТУП

Актуальність теми. Сучасна індустрія комп’ютерних ігор, зокрема інді-сегмент, демонструє динамічне зростання, що супроводжується зростаючим попитом на створення великої кількості оригінального ігрового контенту. Одним із найбільш ресурсоємних етапів розробки є побудова рівнів, особливо у жанрі 2D-платформерів, де композиція, структура та гнучкість ігрового середовища відіграють ключову роль. Враховуючи потребу у швидкому прототипуванні та постійній змінюваності геймдизайну, актуальним стає використання процедурної генерації - методу, який дозволяє автоматизувати процес створення рівнів, забезпечуючи їх варіативність, масштабованість та економію людських ресурсів.

Зважаючи на це, особливу практичну цінність має розробка інструменту процедурної генерації, інтегрованого безпосередньо в середовище розробки Unity. Такий інструмент має не лише пришвидшити роботу дизайнерів, а й забезпечити контрольовану параметризацію, що є критично важливою для підтримки якості йгрового досвіду.

Мета та завдання роботи. Метою цієї кваліфікаційної роботи є створення універсального плагіну процедурної генерації 2D-локацій для ігор у жанрі платформер, який реалізується у вигляді редакторського розширення в Unity Engine та використовує такі алгоритми, як випадкова прогулянка (*Random Walk*), шум Перліна (*Perlin Noise*) та ступінчасте розміщення платформ. Для досягнення цієї мети було поставлено такі завдання: провести аналіз предметної області та сучасних підходів до генерації ігрового контенту; обґрунтувати вибір алгоритмів та параметричних методів побудови рівнів; реалізувати модуль генерації з підтримкою налаштувань через інтерфейс Unity Editor; забезпечити візуалізацію локацій за допомогою Tilemap та збереження конфігурацій у форматі Asset; виконати тестування функціональності системи; продемонструвати результати на прикладі демо-гри.

Об’єкт дослідження - процес створення ігрових рівнів у двовимірних проєктах, зокрема у платформерах. Предмет дослідження - інструментальний засіб процедурної параметричної генерації локацій, реалізований у вигляді плагіну для Unity з можливістю редагування параметрів та візуального налаштування.

У роботі використано методи системного аналізу, об’єктно-орієнтованого проєктування, моделювання із застосуванням UML-діаграм, та ітеративного підходу до реалізації програмних рішень. Для реалізації інструменту обрано ігровий рушій Unity, мову програмування C#, а також засоби розширення редактора Unity: EditorGUILayout, ScriptableObject, Tilemap та JSON для збереження параметричних конфігурацій.

Пояснювальна записка складається з чотирьох розділів: Розділ 1 присвячений системному аналізу предметної області, постановці завдання, аналізу наявних підходів та вибору алгоритмів генерації; Розділ 2 містить проєктування інформаційної структури системи, розробку логічної моделі даних, архітектури, функціоналу та інтерфейсу; Розділ 3 описує реалізацію прикладного програмного забезпечення: кодування модулів генерації, створення інтерфейсу, приклади візуалізації та демо-гру; Розділ 4 включає результати тестування, вимоги до середовища виконання, опис інсталяційного пакету та рекомендації щодо подальшого розвитку системи.

Апробація результатів бакалаврської кваліфікаційної роботи здійснена шляхом публікації тез доповіді у збірнику матеріалів науково-практичного заходу «Теоретичні та прикладні аспекти розробки комп’ютерних систем '2025», де система описувалась лише для «алгоритму випадкової прогулянки» , на етапі його тестування.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

Розробка ігор (з англ. Game Development) - є областю розробки програмного забезпечення , з надзвичайно динамічним та швидким розвитком. Вона надає повну свободу реалізації ідей розробників та на даний момент вважається найприбутковішою з галузей цифрових технологій.

Ігрові проєкти неформально класифікують за масштабом , бюджетом ,що виділений на його розробку , технологічністю та командою розробки. В межах такої класифікації , визначають ігри :

- “AAA”– зазвичай розробляються великими студіями з великим бюджетом на розробку, є дуже масштабними та функціональними.[1]
- “AA” мають середній бюджет та менший розмір студій у порівнянні з AA. Під цю категорії прийнято класифікувати ігри , які можуть мати якісну графіку ,та цікаві механіки , але поступатись функціоналу та масштабу AAA проєктів.
- Інді (Indie) категорія відзначається фокусом на реалізацію оригінальних ідей та механік , незалежними розробниками , або невеликими командами.[2]
- Мобільні та казуальні – проекти , що розробляються для відповідної платформи , або розраховані на короткі ігрові сесії. Такі ігри частіше за все розраховані на широку аудиторію.

Після цієї класіфікації , ігри поділяють на жанри в залежності від реалізованих механік стилем ігрового процесу та взаємодії з гравцем . Основними загально прийнятими жанрами є : екшени (Action games) , стратегії ,рольові та пригодницькі ігри.

Платформні ігри отримали свою назву завдяки взаємодії користувача з платформами , з яких складаються ігрові рівні. Концепція гри полягає

проходженні різноманітних перешкод та шляхів шляхом використання механіки стрибків .Техдизайнери ,формуючи рівні часто вдаються до використання елементів головоломок та зазвичай формують їх з урахуванням прогресії складності , що може підсилювати інтерес користувача до гри .[3]

Процедурна генерація — це підхід до автоматичного створення ігрового контенту, який активно застосовується в іграх різних жанрів:

- У платформерах вона дозволяє динамічно створювати рівні з унікальним розташуванням перешкод, що робить кожне проходження іншим.
- У roguelike іграх забезпечує випадкове формування кімнат або лабіринтів для збереження непередбачуваності.
- У пісочницях використовується для створення відкритих світів з безмежною кількістю варіацій ландшафту.
- У стратегіях генерує карти та умови розміщення ресурсів, підвищуючи варіативність сценаріїв.
- У рольових іграх процедурна генерація дозволяє створювати нові предмети, персонажів або квести, що змінюються в кожній грі.

Такий підхід дає змогу розширити контент без ручної розробки кожного елемента та підтримувати інтерес гравця тривалий час.

Предметною областю даної бакалаврської кваліфікаційної роботи є розробка інструменту процедурної генерації ігрових локацій, що може бути інтегрований у середовище розробки ігор для автоматичного створення ігрових рівнів, локацій ,кімнат, підземель або ландшафтів. Такий підхід дозволяє значно оптимізувати процес створення контенту, зробити ігровий процес більш різноманітним та непередбачуваним. Основна ідея полягає у створенні гнучкого та масштабованого інструменту, який надасть користувачеві можливість налаштовувати параметри генерації, керувати структурою майбутніх рівнів та впроваджувати готові шаблони кімнат або зон. Сфера застосування охоплює інді-проєкти, прототипи, ігри жанрів roguelike , dungeon crawler, платформери та інші, де потрібна автоматизація побудови рівнів.

1.2 Аналіз існуючих підходів і рішень

1.2.1 Існуючі алгоритми процедурної генерації

Процедурна генерація сьогодні широко застосовується в ігровій індустрії , в тому числі в платформерах та іграх з подібними механіками . Автоматичне створення рівнів робить ігри більш різноманітними та цікавими для гравця , оскільки кожен рівень є унікальним. Вона спрощує розробку відсутністю необхідності ручного редагування кожного елементу рівня чи його структури. Серед головних загально прийнятих алгоритмів процедурної генерації можна виділити алгоритми :

- На основі шумів – генерують природні форми ландшафту та інші структури , застосовуються шляхом використання математичних формул шуму.
- Клітинних автоматів – для клітинок задаються набори правил і кожна клітинка змінює свій стан залежно від сусідніх у форматі проходу ,або стіни.
- Випадкової прогулянки – починаючи з певної точки , клітинка на кожному кроці обирає випадковий напрямок , а слід ,що був залишений клітинкою формує локацію.
- Бінарного поділу простору (Binary space partitioning , або BSP) – поділяє ігрову карту на прямокутні області ,В яких розміщуються кімнати , а між ними -коридори.
- На основі графів – представляють локацію у вигляді графів та дозволяють будувати рівні з контролем переходів на основі готових частин рівня.
- Колапсу хвильової функції - базується на правилах сумісності тайлів - з кожним кроком вибирається допустимий елемент, поки карта не заповниться без конфліктів.

1.2.2 Існуючі ігрові продукти з процедурною генерацією

В рамках створення інструменту , що надасть розробникам можливість використовувати дані алгоритми без прив’язки до певного проєкту , спочатку

необхідно провести аналіз існуючих ігрових проєктів ,що використовують даний підхід. Ціль такого аналізу – ознайомлення з популярними методами , визначення впливу на ігровий процес та визначення загальних особливостей для створення універсального інструменту.

Rogue можна назвати грою , завдяки якій з’явився підхід процедурної генерації. Створена в 1980 році Майклом Тоєм та Ґленном Вічманом для операційної системи Unix , вона поклала початок новому жанру в ігровій індустрії , який отримав назву на її честь – “Roguelike” , або ,як її його часто називають користувачі спільноти “Рогалик”. Ідея гри полягає в пошуку скарбів та сутичках з противниками в межах процедурно згенерованого підземелля , яке має структуру , що складається з кімнат та коридорів. Незважаючи на простий дизайн , який складається з системних символів , що представлений на рисунку 1.1 , саме автоматичне створення локацій з динамічним наповненням створює для гравця цікавий досвід дослідження рівнів , кожен з яких має унікальну структуру та кардинально повпливало на сферу розробки ігор. [4]

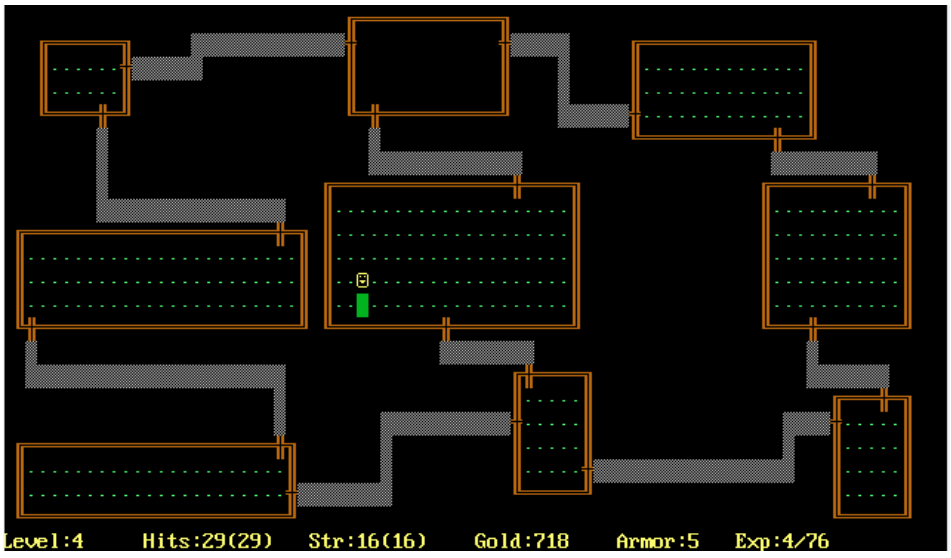


Рис 1.1 Загальний вигляд структури рівня в rogue

No Man’s Sky можна назвати справді масштабною демонстрацією можливостей процедурної генерації в сфері відеоігор , випущена в 2016 році студією Hello Games , вона , однією з небагатьох була зарахована до видатних ігор десятиліття. Основна ідея гри полягає в дослідженні космосу та планет , подорожуючи на космічному кораблі. Незважаючи на неоднозначні відгуки спільноти , які були обґрунтовані рядом технічних нюансів , масштаби гри

ламають уявлення про розміри. За словами розробників , у грі існує понад 18 квінтильйонів унікальних планет , кожна з яких процедурно створюється в процесі гри , включно з ландшафтами ,унікальними флорою та фауною ,ресурсними покладами , архітектурними елементами та навіть звуками .Такі масштаби роблять повне дослідження гри фізично неможливим для людини , оскільки , за оцінками розробників , на це можуть знадобитися сотні мільярдів років , навіть за умови того , що на дослідження однієї планети гравець буде витрачати 1 секунду. Використання процедурної генерації в даному проєкті не лише створює неймовірний масштаб ігрового всесвіту ,без прямої необхідності ручного проєктування та розробки внутрішнього контенту , а й допомагає підтримувати дослідницький інтерес гравців. Головною ж особливістю в контексті розробки став унікальний підхід студії в комбінуванні математичних функцій генерації з художніми фільтрами , що дозволило створити збалансовані , гармонійні та візуально привабливі локації. Варто також зазначити ,що всі алгоритми працюють на основі seed-значення , що дозволяє як створювати унікальні локації , так і за необхідності згенерувати ідентичну локацію за тим же значенням.[5]

Minecraft вважається найпродаванішою відеогрою в історії на даний момент,а також належить до ігор з найширшою аудиторією. Створена в 2009 році , гра одразу завоювала масову підтримку від спільноти геймерів , через свої унікальні особливості , серед яких :

- мінімалістичний кубічний дизайн,
- механіки створення ігрових об’єктів
- процедурно згенерований ігровий світ , з можливостями знищення та встановлення блоків
- Наявність декількох режимів гри

В контексті генерації локацій , дана гра стала еталоном процедурної генерації в ігровій індустрії . Вона включає в себе генерацію ландшафту , “біоми” , що являють собою природні зони , відповідно до яких створюється оточення , яке включає себе флору та фауну , що продемонстровані на рисунку 1.2 , підземелля та готові структури , які містять в собі випробування для гравця та винагороду .

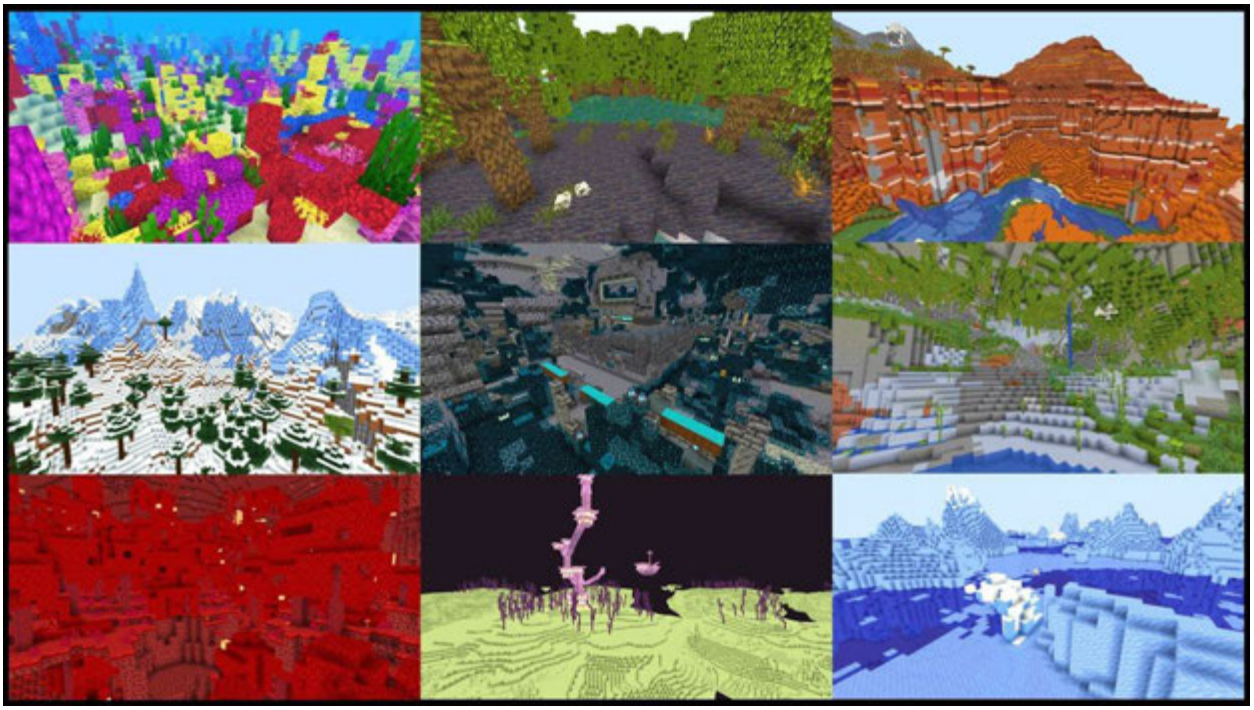


Рис 1.2 Різноманітність біомів гри Minecraft

Саме досконаліть генерації та унікальність геймплею надали грі масову зацікавленість гравців та назавжди вписали її в історію ігор. Генерація “біомів” робить гру максимально наближеною до реальності , що ще викликає ще більший інтерес в гравців . Мінімалістичний дизайн та кубічні форми в свою чергу надали популярності проєктам з графікою в подібному стилі та pixel art дизайном.

Terraria повноцінно можна назвати 2D аналогом попередньо згаданої гри Minecraft. Дана гра є платформером з відкритим процедурно згенерованим світом .В ній наявні ті ж механіки “крафтів” – створення нових ігрових об’єктів з інших , та можливість знищувати і створювати власні форми оточення та структури. В грі також наявні “біоми” , що розділяють природні зони ,а за унікальність генерації відповідають алгоритми на основі seed-значень , що в свою чергу за потреби дозволяють повторити раніше згенеровані ландшафти.

Після формування основних форм рівня відбувається створення різноманітних структур ,таких як підземелля , тунелі , підземні замки та інші , які в подальшому наповнюються ігровими об’єктами відповідно до сценаріїв гри. Враховуючи простоту такого жанру , як платформер , гра все ж змогла сформувати власну фан-базу та активну спільноту .

Таким чином дана гра є хорошим прикладом того , як завдяки ігровим механікам та процедурній генерації інді-проєкт може досягти світового успіху.[6]

1.2.3 Існуючі процедурні генератори

Для визначення вимог розширення процедурного генератора , що буде створений ,необхідно провести аналіз існуючих аналогів процедурних генераторів для роботи в двовимірному просторі . Він необхідний для визначення сильних і слабких їх сторін , порівняння підходів до взаємодії користувача з ним .

Edgar Pro - Procedural Dungeon Generator являє собою процедурний генератор для створення 2D- рівнів для рушія Unity , що поєднує граф-орієнтований підхід з повним контролем над згенерованими рівнями , що створює структуру рівня подібну до тієї , що використовується в іграх жанру roguelike з коридорами та кімнатами . Основа роботи генератору полягає в тому , що користувач може повністю створити шаблони кімнат з відповідним наповненням , а також структуру коридорів з позначеннями точок з’єднання з кімнатами , після чого ,в редакторі графів встановити кімнати , шаблони кімнат для них , та вцілому сформувати загальну логіку взаємозв’язків розміщення кімнат між собою , що продемонстровано на рисунку 1.3. Графічний інтерфейс користувача представлений вбудованими компонентами рушія , такими як Inspector та Graph editor ,а сам генератор представлений у вигляді компоненту. [14]

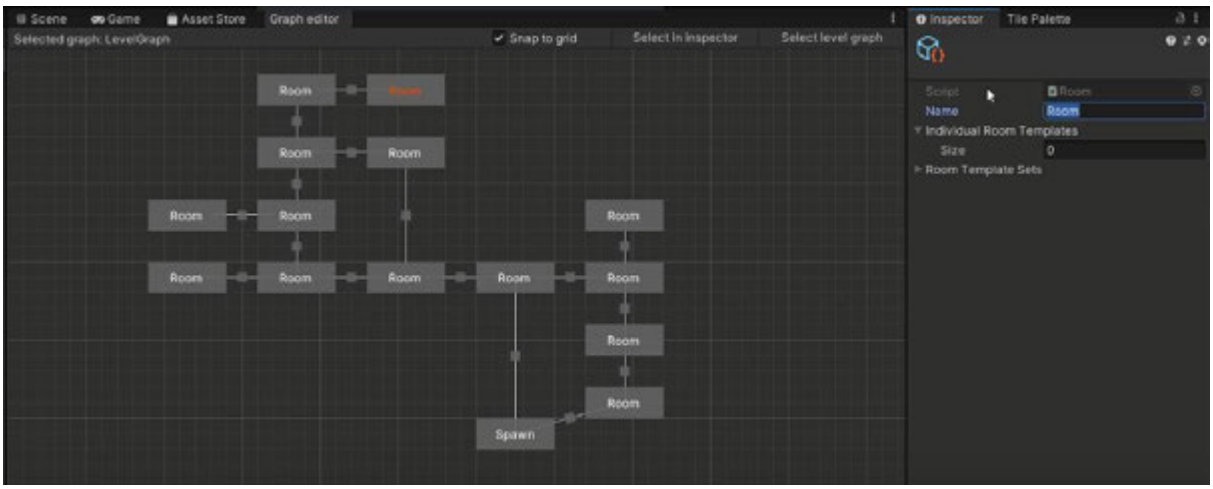


Рисунок 1.3 Формування структури рівня та логіки формування локації в Edgar Pro - Procedural Dungeon Generator

Frigga - Procedural Dungeon Generator -процедурний генератор ,що також використовується в Unity Engine , він пропонує ширший функціонал , як для 2D , так і для 3D проєктів. Даний генератор містить в собі декілька загальноприйнятих алгоритмів процедурної генерації і може створювати локації як повністю з нуля , за правилами за замовчуванням , так і за правилами , що створені користувачем , має підтримку вручну створених структур , а також містить встановлення правил для розміщення текстур . Інтерфейс користувача як і в попередньо згаданому генераторі , реалізований за допомогою вбудованих інструментів рушія та представлений у вигляді компоненту , загальний вигляд елементів керування генератором представлений на рисунку 1.4.[15]

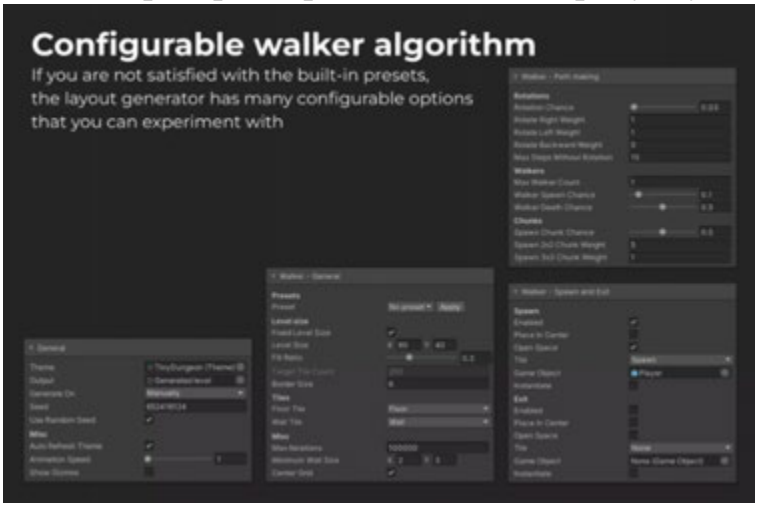


Рис 1.4 Загальний вигляд елементів управління генератором Frigga

Semi-procedural Dungeon Generator, розроблений для Unreal engine , для роботи в 3D просторі. Основна концепція полягає в створені підрівнів із

використанням спеціального Blueprint-компонента “Canvas” , в межах якого розробник може розміщувати будь які об’єкти ,які бажає бачити в результаті на своїй локації ,після чого створена частина рівня додається до списку кімнат в генераторі , який автоматично визначає положення та розміщує випадковим чином ,формуючи лабіринт з коридорів та кімнат , що чудово підходить для ігор жанрів dungeon crawler , RPG та пригодницьких ігор з процедурно згенерованими рівнями. Інтерфейс користувача базується на використанні BluePrint-компонентів у середовищі Unreal Engune .Процедурний генератор , для Unreal engine в 3D просторі, основна концепція якого полягає в створені підрівнів із використанням спеціального Blueprint-компонента “Canvas” , в межах якого розробник може розміщувати будь які об’єкти ,які бажає бачити в результаті на своїй локації ,після чого створена частина рівня додається до списку кімнат в генераторі , який автоматично визначає положення та розміщує випадковим чином ,формуючи лабіринт з коридорів та кімнат , що чудово підходить для ігор жанрів dungeon crawler , RPG та пригодницьких ігор з процедурно згенерованими рівнями.Інтерфейс користувача базується на використанні BluePrint-компонентів у середовищі Unreal Engune .[16]

1.2.4 Переваги та недоліки доступних технологій

Аналіз сучасних ігрових проєктів із процедурною генерацією та існуючих генераторів дозволяє виокремити їхні переваги та недоліки для подальшого створення універсального інструменту.

Ігри на зразок Rogue започаткували підхід процедурної побудови підземель із простими алгоритмами, які й досі актуальні для жанру roguelike, хоча графічна примітивність та обмеження інтерфейсу не дозволяють повноцінно розкрити їхній потенціал у сучасних проєктах. No Man’s Sky продемонстрував надзвичайну масштабованість та різноманітність контенту - згенеровані ландшафти, флора, фауна та архітектура формуються за seed-значенням, але висока технічна складність і початкові проблеми з оптимізацією стали викликами для розробників. Minecraft став еталоном процедурної генерації завдяки простому, але гнучкому підходу до створення світу, біомів і підземель,

хоча його візуальна мінімалістичність обмежує адаптацію в серйозніших проєктах. Terraria, як 2D-аналог Minecraft, доводить, що навіть простий візуальний стиль у поєднанні з процедурною генерацією може забезпечити глибоку реіграбельність і сформувати стійку фан-спільноту, хоча й поступається масштабом 3D-ігор. Головною особливістю всіх перерахованих ігрових проєктів , можна зазначити сильну прив’язку до ігрових сценаріїв та стилістики гри.

Щодо інструментів, Edgar Pro пропонує повний контроль структури рівня в Unity на основі графів і шаблонів, проте потребує складного налаштування й обмежується жанром dungeon crawler. Frigga демонструє гнучкість - підтримує 2D і 3D, кілька алгоритмів, вручну створені правила та візуальне керування, однак вимагає більш глибоких знань середовища Unity. Генератор Semi-procedural Dungeon Generator для Unreal Engine надає просту інтеграцію через Blueprints, використовуючи концепт “Canvas” для створення підрівнів, що автоматично об’єднуються в готову локацію, проте інтерфейс не має повноцінного візуального редактора і орієнтований більше на дизайнерів із досвідом роботи з рушієм. Зроблений огляд дозволяє врахувати як переваги автоматизації та масштабованості, так і важливість збереження контролю та зручності інтерфейсу для майбутнього інструменту процедурної генерації. Головним же недоліком всіх перерахованих генераторів можна виділити відсутність централізованого інтерфейсу користувача. Порівняння для існуючих плагінів процедурної генерації представлено у таблиці 1.1 .

Таблиця 1.1

Порівняння існуючих плагінів процедурної генерації

Назва генератора	Edrag-pro	Frigga	Semi-procedural
Рушій	Unity	Unity	Unreal
Робочі виміри	2D	2D/3D	3D
Інтерфейс	Інспектор , редактор графів	Інспектор, компоненти	Blueprint Canvas

Таблиця 1.1 (Продовження)

Призначення	Локації для ігор roguelike	Універсальний для 2D/3D	Ігри Dungeon crawler /RPG
Гнучкість	Висока, але потребує досвіду	Висока, але потребує досвіду	Середня , має просту інтеграцію
Недоліки	Складне налаштування	Високий поріг входу	Відсутність ui редактора
Підхід до генерації	Граф – орієнтований, заготовки локацій	Декілька алгоритмів, заготовки	Композиція з заготовок

1.3 Опис предметної області

1.3.1 Загальні відомості

Моделювання предметної області є ключовим етапом в розробці проєктів програмного забезпечення , незалежно від сфери , його застосування. Воно дозволяє абстрактно ,узагальнено , формалізовано описати знання про систему , її функціональні особливості ,внутрішню логіку та взаємодію користувачів. Для моделювання предметної області даної кваліфікаційної роботи , використані діаграми UML.

UML (Unified Modeling Language) - уніфікована мова моделювання , що використовується розробниками програмного забезпечення для візуального представлення програмного продукту, роботи його: модулів , процесів , залежностей та взаємодій компонентів і користувачів. Вона являє собою набір правил та стандартів , що використовуються для графічного представлення у вигляді діаграм , що дозволяють універсально описати продукт без прив’язки до засобів його реалізації ,мов програмування , апаратного та програмного забезпечення , що буде використовуватись. Такий підхід дозволяє зрозуміло пояснити роботу системи на різних рівнях абстракції не лише для технічних спеціалістів , а й безпосередньо для замовників. [7]

1.3.2 Концептуальна модель

Концептуальна модель відображає основні сутності предметної області , їх властивості та взаємозв’язки , які необхідні для функціонування інструменту процедурної генерації . Модель описує, яким чином користувач взаємодіє з системою: задає параметри, обирає шаблони або правила, ініціює генерацію та отримує результат у вигляді сформованого рівня. Концептуальна структура не враховує технічної реалізації, а слугує узагальненою логічною основою для подальшого створення функціональної та інформаційної моделей системи.

1.3.3 Взаємодія користувача

Взаємодія користувачів з плагіном продемонстрована на рисунку 1.5 у вигляді UML діаграми прецедентів. Діаграма прецедентів являє собою графічне представлення взаємодії користувачів , розподілених за ролями , з узагальненими функціональними блоками варіантів використання системи , та визначають функціональні вимоги , що будуть визначені в пункті 1.4.5 .

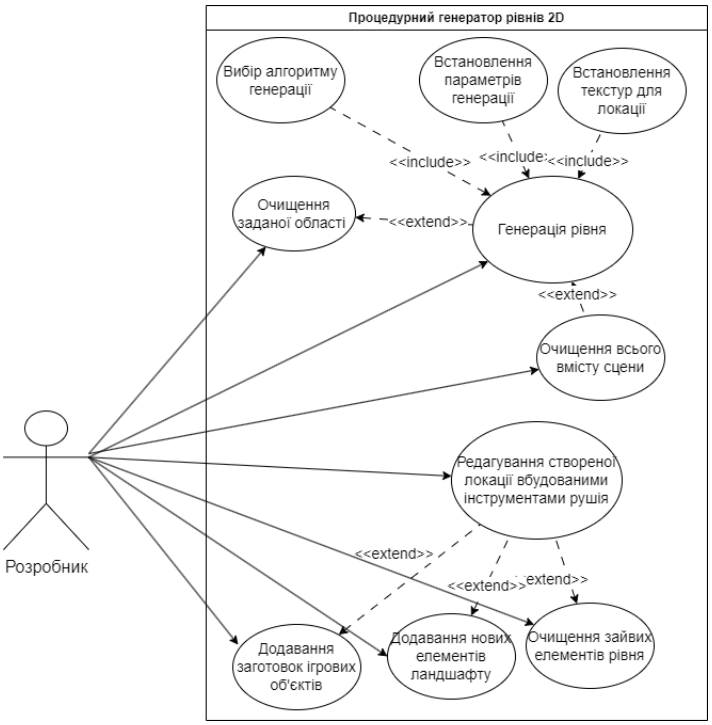


Рис 1.5 діаграма прецедентів для розширення процедурної генерації

На даній діаграмі представлений лише один актор - “Розробник” , що узагальнює гейм дизайнерів , та розробників як користувачів даного продукту . Серед основних прецедентів , представлених на діаграмі наявні наступні :

- Генерація рівня - головна функція генератора , що включає в себе створення та візуалізацію в ігровому просторі локації , для подальшого використання в ігровому проєкті. Даний прецедент обов’язково в себе включає:
    - Вибір алгоритму генерації - функція , що дозволяє обрати алгоритм та відобразити необхідні йому параметри для генерації рівня , або його частини.
    - Встановлення параметрів генерації – використання параметрів з відображених полів для створення локації.
    - Встановлення текстур для локації – використання встановлених користувачем графічних одиниць побудови рівня для стилізації рівня відповідно до потреб проєкту.

Також в розширенні необхідні функції для очищення всього рівня та окремих областей , які не є обов’язковими для процесу створення локації , але будуть корисними для користувача , серед них :

- Очищення всього вмісту сцени – знищення всіх елементів локації ,які були створені генератором , чи безпосередньо користувачем.
- Очищення заданої області - знищення всіх елементів локації , в заданій користувачем області .
- Редагування створеної локації вбудованими інструментами рушія – можливість ручного редагування елементів локації для розширеного контролю коректності створеного рівня , що включає в себе :
    - Додавання готових ігрових об’єктів
    - Додавання нових елементів ландшафту
    - Очищення зайвих елементів рівня.

1.2.4 Послідовність дій при взаємодії

Для демонстрації послідовності з урахуванням часових особливостей передачі та прийому даних між елементами системи , прийнято використовувати діаграму послідовності UML (Sequence diagram). Діаграма послідовності , що описує загалом роботу процедурного генератора ,зображена на рисунку 1.6.

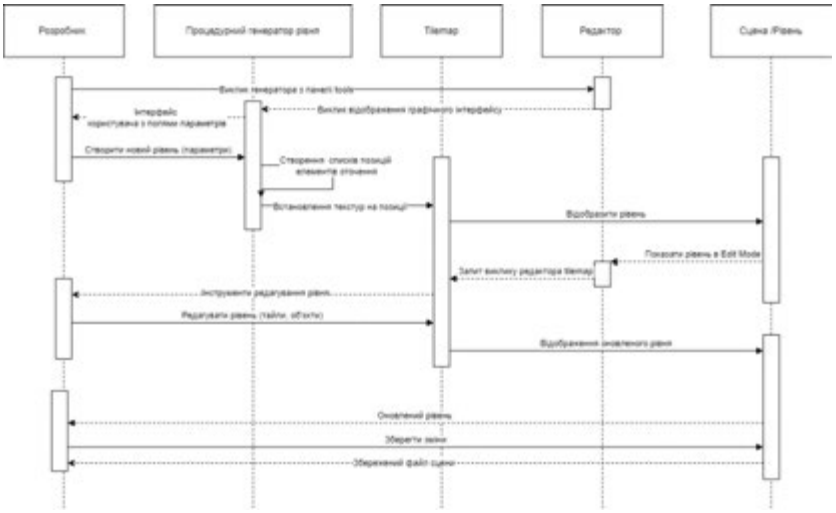


Рис 1.6 Діаграма послідовності для процедурного генератора

На даній діаграмі можемо побачити п’ять основних об’єктів , серед яких :

- Розробник - безпосередньо користувач системи , що викликає генератор та використовує його функціонал.
- Процедурний генератор – Розширення , всередині рушія , з яким взаємодіє розробник.
- Tilemap - ігровий об’єкт , що дозволяє будувати рівень в форматі заповнення сітки графічними елементами.
- Редактор – середовище ігрового рушія , з якого викликається процедурний генератор , та через який буде відбуватися взаємодія користувача.
- Сцена /Рівень – ігровий об’єкт , що є ігровим оточенням та містить в собі всі елементи , які гравець зможе бачити в грі .

1.4 Постановка завдання

1.4.1 Мета та призначення розширення

Метою розробки є створення розширення , що реалізує функціональну систему процедурної генерації двовимірних рівнів на основі комбінацій локацій , створених різними алгоритмами , що може в подальшому бути відредагованим за допомогою вбудованих інструментів редактору рушія. Це розширення дозволяє автоматизувати процес побудови ігрового середовища , зменшуючи потребу в ручному проєктуванні та пришвидшуючи створення локацій та рівнів зі складною структурою.

Розширення включає інтерфейс для налаштування як параметрів генерації , що безпосередньо впливають на форму та розміри локації , так і на графічну складову ,що включає встановлення текстурних елементів.

Створене розширення має модульну архітектуру та може бути адаптованим під інші алгоритми чи типи рівнів в майбутньому , що надає йому гнучкість в подальшому використанні , розробці та масштабуванні.

1.4.2 Цільова аудиторія

Розширення призначене для використання в процесі розробки двовимірних ігор , зокрема проєктів , що потребують створення рівнів з високим ступенем варіативності , великими розмірами та великою кількістю самих рівнів .

Основними користувачами інструменту є інді-розробники та технічні геймдизайнери , основне завдання яких полягає в створенні або тестуванні рівнів на різних етапах розробки ігрового проєкту. Плагін є особливо корисним у випадках , коли потрібно :

- Швидко генерувати прототипи рівнів з різною структурою .
- Створювати велику кількість унікальних локацій в умовах обмеженого терміну.
- Проводити тестування ігрової логіки на різних топологіях середовища.

1.4.3 Аналіз вимог

Після проведеного аналізу предметної області та існуючих рішень в пунктах 1.1 – 1.4.2 , коли визначені головні потреби на основі переваг та недоліків , необхідно визначити головні функціональні та нефункціональні вимоги , що в свою чергу допоможе краще визначити головні потреби та завдання для розробки кінцевого продукту.

Функціональні вимоги – це функції , які повинна мати та реалізовувати система. Для мого процедурного генератора , функціональними вимогами будуть наступні :

- Наявність інтерфейсу користувача , зосередженого в окремому вікні
- Збереження встановлених параметрів між сесіями та на різних ігрових рівнях
- Вибір алгоритму генерації
- Генерація локації
- Встановлення параметрів генерації
- Можливість очищення попередньої генерації
- Інтеграція з ігровим рушієм

Нефункціональні вимоги визначають загальні характеристики , що не є пов’язаними з функціональністю

- Зручність у використанні -Інтерфейс користувача повинен бути інтуїтивно зрозумілим , не вимагати знань програмування для базової генерації рівнів та містити всі параметри в межах одного вікна.
- Усі ключові дії (генерація, очищення, вибір шаблонів) мають бути доступні через панель управління у редакторі.
- Продуктивність - час генерації рівня не повинен перевищувати 2 секунд.
- Плагін повинен працювати без зависань або втрати кадрів навіть при великих розмірах карти.
- Сумісність - Плагін має підтримувати останню стабільну версію рушія.

1.4.4 Алгоритми процедурної генерації

Алгоритми є основною складовою процедурного генератора . Опираючись на специфіку жанру , та наявні алгоритми , для початкової версії плагіну було обрано три незалежні алгоритми процедурної генерації рівнів , кожен з яких має свої особливості , параметри та область застосування , що дозволяє створювати різноманітні за структурою та складністю двовимірні локації. Серед обраних алгоритмів наявні:

- Випадкової прогулянки , який базується на концепції випадкового вибору з'єднаних між собою позицій елементів , що можуть розширюватись в різних напрямках з заданої початкової позиції , що забезпечує максимальну непередбачуваність та мінімальний набір параметрів від користувача. Для генератора гарним варіантом стане розширений варіант даного алгоритму з можливостями побудови roguelike- подібних за структурою локацій ,з кімнатами та коридорами , що значно розширює варіативність генерацій та може бути використаним для декількох жанрів.
- Шуму Перліна ,який на відміну від попереднього , є більш передбачуваним та використовує шумову функцію Перліна для побудови локації. Перевага даного методу полягає в створенні локацій в більш «органічних» формах ,створюючи локацію з великою кількістю розгалужень.
- Ступінчатих платформ – працює за принципом створення платформ на різних висотах , імітуючи структуру сходинок. Завдяки гнучкому налаштуванні параметрів , даний алгоритм є незамінним при створенні рівня в платформерах , оскільки наявність платформ є основою даного жанру.

Висновки до розділу 1

В результаті системного аналізу предметної області було визначено , що технології процедурної генерації активно та широко застосовуються в сучасній ігровій індустрії , забезпечуючи автоматизоване створення локацій , унікальність

контенту та значне скорочення витрат на ручне проєктування. Аналіз ігрових продуктів підтвердив ефективність та актуальність різних підходів до генерації рівнів , водночас виявивши їх обмеження , пов’язані з інтеграційною гнучкістю. Розгляд популярних генераторів дозволив виявити функціональні особливості доступних рішень. Проте недоліками можуть стати складність інтеграції для початківців та відсутність централізованого інтерфейсу.

На основі проведеного аналізу було сформульовано предметну область даної кваліфікаційної роботи – створення універсального інструменту процедурної генерації 2D- рівнів із підтримкою декількох алгоритмів , графічним інтерфейсом налаштувань , модульною структурою та можливістю інтеграції у середовище розробки ігор. Було визначено функціональні та нефункціональні вимоги до системи , які стали основою для побудови UML діаграм , що моделюють ключові сценарії взаємодії користувача з інструментом . Концептуальна модель описала головні об’єкти системи та їхні зв’язки , що дозволяє перейти до етапу проєктування та реалізації ефективного розширення для автоматичного створення двовимірних ігрових локацій.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Загальна архітектура системи

Проєктування інформаційного забезпечення є невід'ємною та ключовою складовою процесу розробки програмної системи. Основним завданням будь-якого програмного продукту є взаємодія з даними, які можуть бути вхідними, проміжними або вихідними. На цьому етапі визначається не лише склад і типи даних, що обробляються системою, але й архітектура їх зберігання, логічна структура, підходи до організації функціоналу та особливості представлення через інтерфейс. Це забезпечує цілісне розуміння інформаційної основи розроблюваного програмного продукту та формує базу для його подальшої реалізації.

Обов'язковою частиною в формуванні інформаційного забезпечення програмної системи є проєктування його архітектури. Воно дозволяє визначити програмні модулі та їх взаємодію , що значно допомагає спростити подальшу розробку та реалізацію.

Враховуючи постановку завдання та функціональні вимоги (Підрозділ 1.4) , можна виділити основні компоненти програмної системи , серед яких :

- Ядро генератора – Основний обчислювальний модуль , що містить в собі :
    - Алгоритми процедурної генерації
    - Систему візуалізації згенерованих локацій
- Інтерфейс користувача , що слугує компонентом зв'язку між користувачем та ядром генератора. Він включає в себе :
    - Модуль контролю алгоритмів процедурної генерації та їх параметрів.
    - Допоміжні модулі для управління рівнем , очищення та виділення області для комфортного використання.

- Модуль збереження параметрів для графічного інтерфейсу та дозволить зберігати встановлені параметри між сесіями
- Демо гри - Додатковий модуль , що містить базові ігрові механіки гри жанру «платформер» для подальшого тестування згенерованих локацій.

Розподілення на такі компоненти створює гнучкий підхід , що має позитивний вплив на подальшу підтримку та розширення процедурного генератора як програмної системи. Графічне представлення архітектури системи представлено в розділі 3 у вигляді діаграми пакетів.

2.2 Логічна модель даних

Логічна модель даних призначена для формального опису структури даних , які використовуються в інформаційній системі , незалежно від реалізації в мові програмування , чи базі даних , та слугує посереднім етапом між концептуальною та фізичною моделями.

ER (Entity-Relationship) – діаграма слугує основним засобом для візуалізації логічної моделі даних. ER діаграма має три головні складові , серед яких :

- Сутність (Entity) – об’єкт предметної області , що узагальнює реальний ,організаційний ,або абстрактний елементи , чи подію , які в сукупності формують структуру системи.
- Атрибут (Attribute) – складова , з сукупності яких формується сутність , яка зазвичай визначає певну характеристику та може бути вимірюваною в контексті типів даних системи.
- Зв’язок (Relationship) – опис відношення між сутностями , сукупність яких формують цілісність та структуру моделі даних . За своєю специфікацією зазвичай вказують кратність входжень сутностей , що з’єднані.

Опираючись на специфіку проєкту ,та відсутність необхідності постійного зберігання великої кількості даних , була спроєктована логічна модель системи , що представлена на рисунку 2.1 .

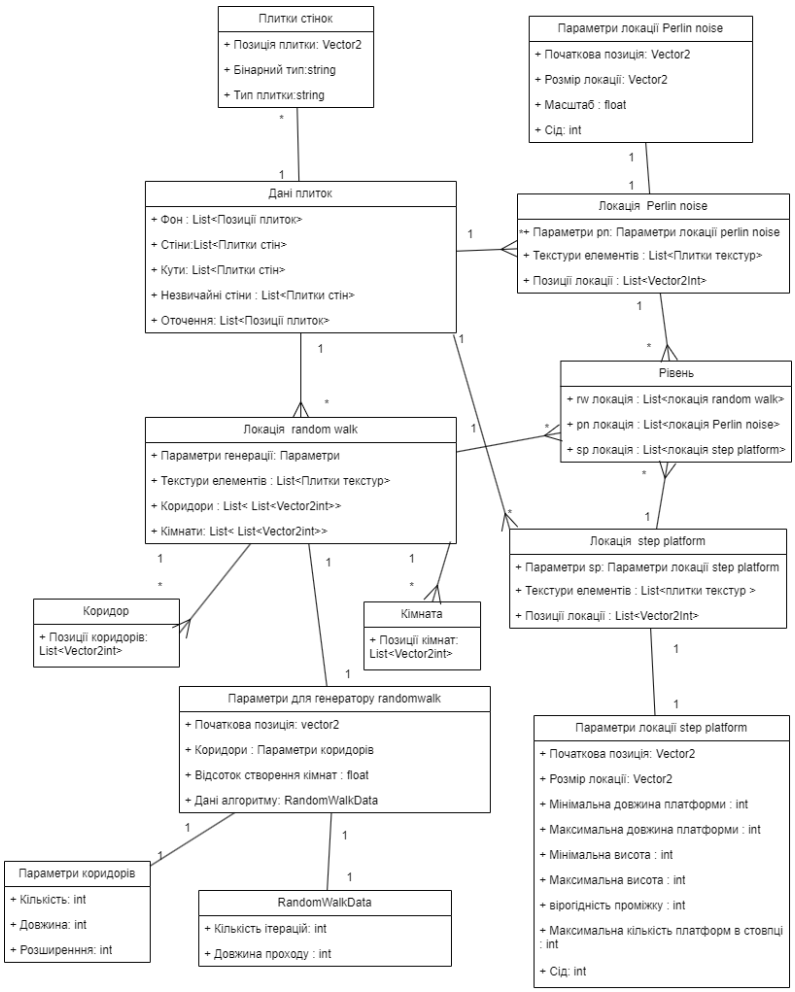


Рис. 2.1 Логічна модель системи процедурної генерації у вигляді ER діаграми

Подана ER-модель фактично поєднує в собі риси спрощеної UML-діаграми класів та структури зберігання об’єктів у NoSQL-підході. На відміну від класичної реляційної моделі, де зв’язки строго нормалізовані, тут реалізовано абстрактне подання сутностей і їхніх вкладених властивостей, які можуть зберігатися у вигляді вкладених документів або JSON-структур. Наприклад, кожна локація містить в собі набір параметрів, списки координат, текстурних плиток, що відповідає вкладеним документам у NoSQL-парадигмі. Зв’язки між сутностями мають характер односпрямованої композиції або агрегації, а множинність виражена через використання колекцій (List< >). У моделі також не передбачено явного первинного ключа, що властиво документно-орієнтованому зберіганню, де ідентифікатори є внутрішньою частиною структури або формуються автоматично.

На даній діаграмі ми можемо побачити 14 сутностей , кожна з яких має власне призначення , що описано в табл. 2.1 .

Таблиця 2.1

Сутності логічної моделі та їх призначення

№	Назва сутності	Призначення сутності
1	Рівень	Узагальнює згенеровані локації за різними алгоритмами.
2	Локація random walk	Представляє частину рівня, згенеровану за алгоритмом випадкової прогулянки.
3	Локація Perlin noise	Представляє частину рівня, згенеровану за шумом Перліна.
4	Локація Step platform	Представляє частину рівня , побудовану алгоритмом ступінчатих платформ.
5	Параметри генерації random walk	Визначають налаштуваня генерації для алгоритму random walk.
6	Random walk data	Визначає параметри для створення кімнат як складової частини локації.
7	Параметри коридорів	Визначають властивості коридорів , як складових локації.
8	Кімната	Представляє область, згенеровану як кімната.
9	Коридор	Представляє шлях або прохід між кімнатами.
10	Плитки стінок	Описує проміжні дані в процесі визначення типу графічного об’єкта для подальшої візуалізації.
12	Дані плиток	Сховище графічних елементів всіх необхідних текстур.
13	Параметри Perlin noise	Визначають налаштування алгоритму шуму Перліна.
14	Параметри Step platform	Визначають налаштування алгоритму побудови ступінчатих платформ.

2.3 Вибір і структура сховища даних

У процесі розробки інструменту процедурної генерації рівнів важливим етапом є вибір відповідного типу сховища для збереження параметрів генерації, шаблонів, конфігурацій, текстурних елементів. Враховуючи характер проєкту, можна розглядати два основних підходи: використання повноцінної SQL-подібної системи управління базами даних (СУБД) або застосування конфігураційних файлів. Для визначення того , в якому вигляді краще зберігати інформацію, необхідно проаналізувати та порівняти між собою обидва підходи. Такий аналіз допоможе більш доцільно обрати спосіб зберігання даних.

2.3.1 SQL – подібні СУБД

СУБД (система управління базами даних) – програмне забезпечення , що забечує керування даними в базі даних , включаючи створення , зберігання , оновлення , та вилучення даних.

SQL (Structured Query Language) – структурована мова запитів, що застосовується для роботи з реляційними базами даних.

SQL- подібними СУБД прийнято називати системи, що працюють з реляційними базами даних. Такі бази даних пропонують формалізоване зберігання даних у вигляді таблиць з чітко визначеними типами , зв’язками та правилами цілісності .Вони підходять для складних клієнт – серверних систем , веб сервісів , а також в контексті ігрової розробки – для великих багатокористувацьких ігрових продуктів. До основних особливостей, що притаманні більшості SQL- подібних СУБД відносять :

- Використання реляційної структури ,яка передбачає організацію даних в таблицях , що пов’язані між собою через ключові атрибути та кожна таблиця має чітко визначену структуру з атрибутів ,що мають встановлені типи даних.
- Використання мови SQL для всіх базових операцій , що дає можливість виконання складних запитів , таких як : фільтрація , сортування , об’єднання таблиць та агрегація.

- Забезпечення виконання операцій в межах транзакцій з властивостями ACID ,що забезпечує : атомарність , цілісність , ізольованість та довговічність даних.
- Розгортання бази даних як на локальному комп’ютері , так і на віддаленому вузлі .Таким чином вона може використовуватись одним користувачем , так і декількома одночасно.
- Оптимізація для великих обсягів даних та великої кількості одночасних запитів.

2.3.2 Конфігураційні файли

Конфігураційними файлами прийнято називати файли , що містять параметри налаштувань та структуровані дані , що використовуються програмним забезпеченням. Вони дозволяють зберігати інформацію поза програмним кодом , що значно спрощує налаштування , масштабування та подальшу підтримку програмного продукту. Такі файли як правило мають текстовий формат , що забезпечує легке зчитування як людиною , так і програмами. Конфігураційні файли мають можливість серіалізації та десеріалізації. Серіалізація в свою чергу являє собою процес перетворення об’єкта , або структури даних у формат файлу. Десеріалізація виконує зворотню функцію – зчитування з файлу інформації та перетворення її в структуру даних , або в об’єкт.

Таким чином , конфігураційні файли є незамінною частиною будь якої програми , чи програмного модуля , що мають наступні загальніособливості:

- Пітримка вкладених елементів ,списків , масивів.
- Зміна параметрів без перекомпіляції коду.
- Збереження окремо від основної логіки програми.
- Придатність для серіалізації/десеріалізації.

2.3.3 Порівняння підходів збереження даних

Для узагальнення викладеної вище інформації та наочного зіставлення ключових характеристик обох підходів до зберігання даних , доцільно порівняти ключові особливості в Таблиці 2.2 . Порівняння обох підходів допоможе визначити який з них більш підходить для процедурного генератора.

Таблиця 2.2

Порівняння особливостей

Критерій	SQL-подібні СУБД	Конфігураційні файли
Тип структури	Таблична (реляційна)	Ієрархічна або вкладена
Формат зберігання	Бінарний або серверна БД	Текстові файли (JSON, XML, YAML, INI)
Гнучкість структури	Низька (схема чітко визначена)	Висока (можна легко змінювати або доповнювати дані)
Складність налаштування	Висока (потрібна інтеграція, драйвери)	Низька (достатньо зчитування файлу)
Редагування вручну	Ускладнене або неможливе	Просте (через будь-який текстовий редактор)
Інтеграція з рушієм	Потребує сторонніх бібліотек або API	Нативна підтримка в Unity та Unreal Engine
Придатність до версіювання (Git)	Частково, через дампи або міграції	Повна, оскільки файли — текстові
Застосування в проєкті	Надлишкове для локального плагіна	Оптимальне для збереження параметрів
Масштабованість	Висока	Обмежена
Продуктивність	Висока при великому обсязі даних	Висока для невеликих обсягів (параметри, шаблони)

На основі проведеного аналізу та порівняння двох підходів збереження інформацій , було обрано конфігураційний файл як найкращий варіант. Висока гнучкість його роботи , відсутність необхідності встановлення додаткових

драйверів та програмного забезпечення та можливість швидко змінювати дані та їх структуру , надають йому значну перевагу перед SQL -подібними СУБД. Невеликий об’єм даних , без вимоги чіткої структуризації та доступність для інструменту , роблять реляційні СУБД абсолютно не актуальними в контексті даного розширення. Вибір формату конфігураційного для збереження параметрів буде детальніше описано в розділі 3.

Для ігрової демо-частини системи було прийняте рішення також зберігати інформацію у вигляді елементів файлової системи через відсутність необхідності залучення потужної бази даних. Такий підхід був застосований опираючись на жанр гри та часткову реалізацію ігрового проєкту виходячи з його призначення.

2.3.4 Визначення ключових параметрів для збереження

Після формування логічної структури даних системи та способу зберігання даних , необхідно визначити ключові параметри для збереження. Такий аналіз допоможе точно сформувати структуру даних для забезпечення подальшого збереження параметрів між сесіями використання плагіну.

Враховуючи логічну структуру плагіну та перелік атрибутів сутностей , структура файлу , що буде використовуватись для збереження параметрів , можемо виділити наступні категорії даних :

- Параметри вікна інтерфейсу користувача:
    - Обраний алгоритм
    - Активна вкладка
- Параметри компонентів генерації , що містять налаштування для алгоритмів :
    - Random walk,
    - Perlin noise,
    - Step platform.
- Параметри візуалізатора , що являють собою всі встановлені графічні елементи , які будуть застосовані для побудови локації.

Збереження параметрів генератора допоможе покращити користувацький досвід використання продукту та зробить розширення повноцінною складовою середовища розробки.

2.4 Проєктування функціоналу

Проєктування функціональної частини системи передбачає визначення основних дій, які має виконувати програмний продукт, відповідно до поставлених цілей. У межах розширення повинні бути реалізованими ключові функції:

- генерація ігрового рівня на основі вибраного алгоритму,
- налаштування параметрів генерації,
- візуалізація результату на сцені редактора,
- можливість очищення сцени або окремої області.

Крім того, система повинна підтримувати збереження та завантаження конфігурацій у вигляді зовнішніх файлів.

Для формування структури функціональної частини розширення , було спроєктовано діаграму класів UML (Рис. 2.2) , яка дозволить розподілити необхідні компоненти системи на повноцінні програмні модулі , визначивши необхідні параметри та функції , що будуть реалізовані в середовищі розробки.

Представлена діаграма є початковою версію, що відображає загальну логічну структуру основних компонентів програмної системи на етапі проєктування. Ця модель слугує базою для подальшої реалізації функціоналу.На даній діаграмі було сформовано 10 основних класів системи , які включають :

- ProceduralGeneratorWindow — головне вікно редактора, через яке користувач обирає алгоритм, вводить параметри, керує конфігурацією та запускає генерацію.
- AbstractProceduralGenerator — базовий абстрактний клас для всіх генераторів. Містить Tilemap-візуалізатор, стартову позицію, методи ініціалізації та запуску генерації.

- RandomWalkLocationGenerator — реалізує алгоритм випадкової прогулянки, створює кімнати та коридори на основі RandomWalkData.
- PerlinNoiseGenerator — реалізує генерацію локацій за допомогою шуму Перліна з урахуванням масштабу, seed та викривлення карти.
- StepPlatformGenerator — будує платформи на різних рівнях з урахуванням висоти, довжини та щільності розміщення.
- RandomWalkData — параметри генерації випадкової прогулянки: кількість ітерацій і довжина кроку.
- WallGenerator — генерує стіни та кути на основі згенерованих локацій, підтримує базові та кутові плитки.
- WallTypes — зберігає типи плиток стін як набір позицій.
- Direction2D — надає напрями (основні, діагональні, випадкові) у вигляді векторів та оптимізує вибір напрямку.
- TileMapVisualizer — відображає плитки на Tilemap: очищує частини сцени, розміщує внутрішні/зовнішні плитки, використовує карту текстур.

Під час проєктування діаграми класів , було визначено декілька ключових модулів , з яких складається плагін , до них входять:

- Модуль інтерфейсу користувача – містить сукупність полів для введення параметрів та відповідає за відображення вікна редактора, вибір алгоритму, запуск генерації та збереження конфігурацій , розміщується в класі “ProceduralGeneratorWindow”.
- Ядро генерації , що призначене реалізувати логіку побудови рівня за допомогою різних алгоритмів на основі заданих параметрів. Розподіляється ядро між класами “AbstractProceduralGenerator”, “RandomWalkLocationGenerator”, “PerlinNoiseGenerator”, “StepPlatformGenerator”.

- Модуль параметрів і утиліт - забезпечує зберігання налаштувань генерації, побудову стін і допоміжні обчислення , що необхідні для коректної роботи генератору. Даний модуль розподілений між класами “RandomWalkData”, “WallGenerator”, “WallTypes”, “Direction2D” .
- Модуль візуалізації - відображає згенеровані рівні на сцені редактора, виконує очищення та розміщення плиток у Tilemap та розміщений в класі “TileMapVisualizer”.
- Також наявний неявно визначений модуль збереження інформації , поміщений всередині класу “Procedural Generator Window” ,який забезпечує зчитування та запис параметрів генерації у зовнішні файли, що дозволяє зберігати сесію та переносити налаштування.

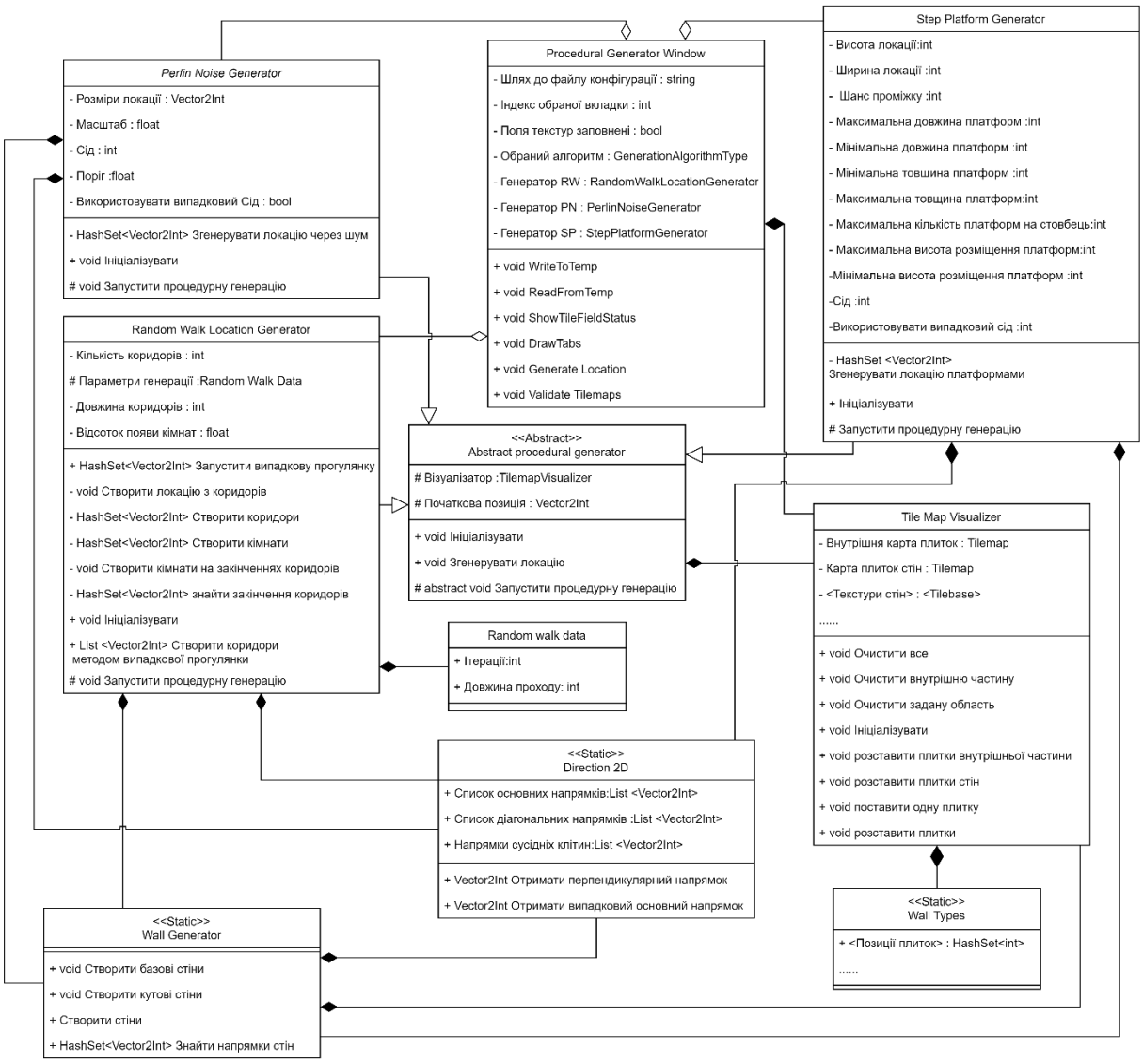


Рис. 2.2 Діаграма класів

Усі модулі взаємодіють між собою через чітко визначені зв’язки, забезпечуючи цілісну та гнучку систему процедурної генерації. У процесі деталізації алгоритмів та програмування окремих модулів (у підрозділі 3.3) дана структура буде доповнена новими класами та функціональними можливостями.

2.5 Проєктування інтерфейсу користувача

Інтерфейс користувача є важливою складовою , оскільки забезпечує доступ до основного функціоналу без необхідності прямої взаємодії з кодом. Його проєктування спрямоване на створення інтуїтивно зрозумілого, логічно організованого та функціонально повного середовища для налаштування параметрів генерації, вибору алгоритмів, керування конфігураціями та перегляду результатів. У межах проєкту інтерфейс реалізовано у вигляді окремого вікна редактора, що інтегрується у середовище розробки. Перед безпосередньо самою розробкою необхідно розробити мокапи інтерфейсу.

Мокапами інтерфейсу прийнято називати попередньо створені макети інтерфейсу користувача , де вже розташовані основні елементи ,наявна необхідна структура, подібний або ідентичний зовнішній вигляд без реалізації обчислювальної логіки. Мокапи слугують початковим шаблоном інтерфейсу , для подальшого відтворення та присвоєння логіки в середовищі розробки. Вони допомагають сформувати уявлення про компонування параметрів ,кнопок ,полів введення та інших елементів , пришвидшити та спростити реалізацію продукту. На рисунку 2.3 представлений мокап інтерфейсу плагіну для вкладки загальних налаштувань з демонстрацією загальної структури.

На даному рисунку інтерфейс зображений в мінімалістичному стилі з навігацією , що реалізована через вкладки. На вкладці загальних налаштувань ми можемо побачити структуру , що складається з випадаючого списку з вибором алгоритмів генерації , початковою позицією та виглядом полів параметрів. Знизу наявні 3 кнопки , що будуть виконувати генерацію , очищення частини рівня та його частини. На другій вкладці (Рис. 2.4) зображено мокап інтерфейсу вкладки з налаштуванням текстур. На ній зображена структура , що містить поля для встановлення текстурних елементів , а також кнопку виклику вікна , на якому

буде зображена графічна схема розміщення плиток відповідно до полів ,у вигляді самих плиток з підписами назв полів. Такий підхід , хоч і робить встановлення параметрів складнішим завданням , але забезпечує повний контроль над всіма графічними елементами рівня

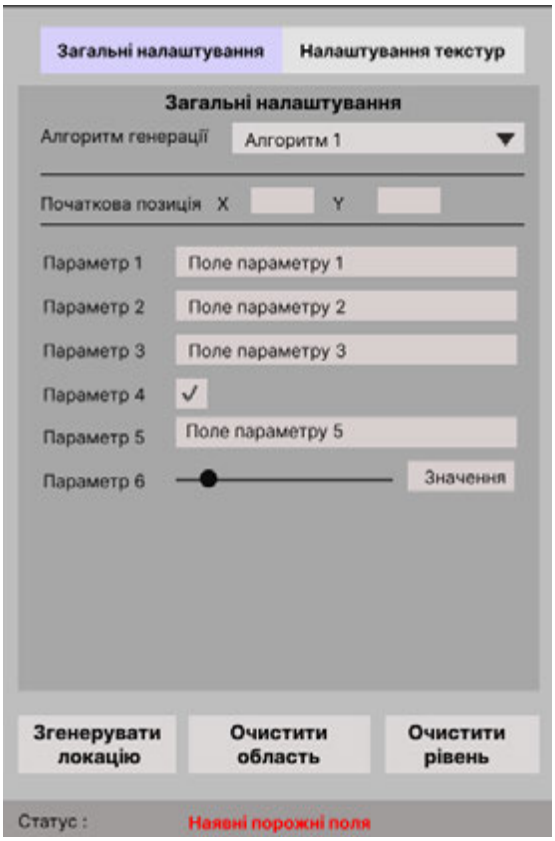


Рис. 2.3 Мокап інтерфейсу головної вкладки

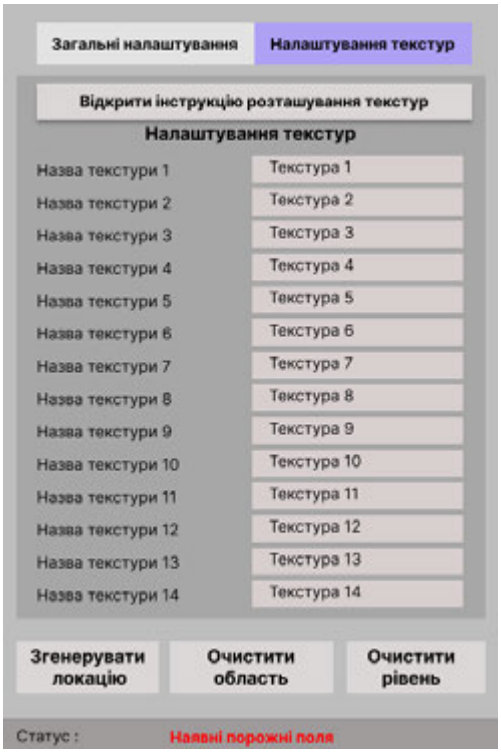


Рис 2.4 Мокап інтерфейсу вкладки налаштування текстур

Висновок до розділу 2

Даний розділ охоплював ключові аспекти інформаційного забезпечення програмної системи, що має визначальне значення для її подальшої реалізації та ефективного функціонування. На основі аналізу цілей та вимог проєкту було сформовано загальну архітектуру, яка поділяє систему на окремі логічні модулі: ядро генерації, інтерфейс користувача, модуль візуалізації, збереження параметрів та демо-частину. Така структурна організація забезпечує гнучкість, масштабованість і полегшує супровід проєкту.

У межах логічного проєктування була побудована ER-модель, яка реалізує підхід, наближений до NoSQL-парадигми. Це дозволяє зберігати параметри генерації у вигляді вкладених структур без надмірної нормалізації, що є доцільним для невеликих автономних систем на кшталт редакторського плагіна.

Проведено обґрунтоване порівняння двох підходів до організації збереження даних — SQL-подібних СУБД і конфігураційних файлів. В результаті було обрано конфігураційні файли як оптимальний варіант зберігання даних для локальної системи, що не потребує складної структури запитів чи масштабної підтримки транзакцій. Такий підхід забезпечує зручність редагування, простоту інтеграції з рушієм та високу адаптивність під час розробки.

Окрему увагу приділено визначенню ключових параметрів, які зберігатимуться між сесіями, що забезпечує безперервність роботи та комфорт для користувача. Описана діаграма класів дала змогу сформувати попередню структуру функціональних компонентів, яка стане основою для подальшої деталізації та програмування. Також було здійснено початкове проєктування інтерфейсу користувача — через створення мокапів, які відображають логіку та компоновку основних елементів керування у вікні редактора.

Загалом, результати розділу забезпечують цілісне уявлення про інформаційну складову проєкту та створюють міцну базу для розробки архітектури модулів, алгоритмів генерації й взаємодії з користувачем, що будуть реалізовані у наступному розділі.

3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вибір інструментальних засобів

Вибір інструментальних засобів грає ключову роль в подальшій реалізації системи , оскільки саме від них залежить ефективність розробки , зручність інтеграції модулів , масштабованість , стабільність роботи плагіну та його оптимізація використовуваних ресурсів. Коректно підібрані інструменти визначають не лише технічні можливості , а й швидкість впровадження , зручність підтримки та подальшу актуальність.

3.1.1 Вибір ігрового рушія та мови програмування

Визначення середовища розробки ігор грає ключову роль в реалізації . Рушій формує основу системи генерації , та впливає на технічну сумісність , продуктивність і гнучкість системи .Від вибору цих інструментів залежить можливість та спосіб реалізації алгоритмів процедурної генерації , зручність роботи з графічними елементами та доступ до необхідних компонентів. Для даного проекту були розглянуті три варіанти ігрових рушіїв : Unreal Engine , Unity та Godot

Unreal Engine - ігровий рушій , що створений компанією Epic Games та підтримується нею , що початково був призначений для створення ігор жанру шутер від першого лиця ще в кінці 1990-х років , однак з виходом подальших версій набув популярності в розробці ігор найрізноманітніших жанрів. З початку свого існування , рушій славився можливостями рендерингу та динамічного освітлення , власним редактором рівнів та підтримкою багато користувацького режиму ще на перших своїх версіях , чим швидко здобув авторитет серед спільноти розробників та геймерів по всьому світу. На даний момент представлена найактуальніша версія – Unreal Engine 5 , яку можна назвати еталоном середовища для розробки AAA ігор . Таку репутацію рушій отримав через свій функціонал , який включає надпотужну систему для роботи з графікою

, підтримку різних підходів розробки , популярність серед спільноти та оптимізацію .

Графічна складова забезпечується технологіями віртуалізованої геометрії “Native” , системою динамічного глобального освітлення “Lumen” та інтеграцією бібліотеки сканованих моделей і текстур Quixel Megascans , які в сукупності з новими інструментами для створення нових світів створюють максимально фотореалістичну картинку . Графічні можливості рушія вже набули актуальності не лише в ігровій , а й і в кіноіндустрії , що ще раз підкреслює його актуальність як інструменту.

Оптимізація проєктів даного рушія здійснюється шляхом використання нехай і трішки застарілої , але досі надзвичайно актуальної та потужної в обчислювальних можливостях мови C++, однак може відбуватись в форматі нод що називається “Blueprints” та може бути використовуваним розробниками які є початківцями , та яким не подобається стандартний спосіб через програмний код.[8]

Unity Engine є також дуже потужним та не менш популярним інструментом для розробки багатоплатформових проєктів. Створений в 2005 році та представлений як доступний інструмент , що стане чудовою альтернативою існуючим рушіям ,для популяризації ігрової розробки , він набув значної популярності , закріпивши за собою статус головного середовища для інді розробників. Unity має низку факторів, що зробили його популярним та актуальним на сьогодні середовищем ігрової розробки.

Даний рушій має інтуїтивно зрозумілий інтерфейс ,що дозволяє легко освоїтися навіть початківцям. Він оформлений у вигляді різних вікон і панелей , що є достатньо схожим на стандартне представлення операційних систем, однак інтерфейс редактора є достатньо гнучким , що дозволяє користувачу змінити розміщення його елементів на свій розсуд.

Потужний графічний рушій , що дозволяє легко створювати гарні візуальні ефекти та анімації. Редактор також має вбудовану систему матеріалів та текстур , що надає можливості створення різноманітного оточення в іграх.

Широка документація та спільнота є ще одним з дуже важливих факторів успішності даного продукту. Його розробники допомагають початківцям навчатись за допомогою активної бази знань , великої та розширеної документації та навіть власних курсів з відеоуроками.

Окрім ігрової розробки , рушій активно використовується в сфері реклами. За його допомогою створюють інтерактивний 2D та 3D контент , що часто підвищує ефективність реклами та ще раз підкреслює важливість даного продукту в екосистемі ігрової розробки.

Unity використовує мову програмування C# , що може вплинути на продуктивність через високий рівень мови , однак спрощує розробку , оскільки такі мови як правило простіші для написання коду. Також в даному середовищі наявна власна система візуального програмування Visual Scripting , що має структуру ідентичну до Blueprints в Unreal Engine, однак потребує додаткового налаштування.[9]

Godot game engine являє собою ігровий рушій загального призначення , що є універсальним для найрізноманітніших проєктів. Початкова розробка розпочалася ще в 2001 році , однак до 2014 року рушій був повністю переписаний та вдосконалений і як наслідок став продуктом з відкритим кодом , що дозволяє розробникам гнучко його налаштовувати відповідно до потреб свого проєкту. Дане середовище пропонує інтуїтивно зрозумілий інтерфейс , що постійно вдосконалюється , що позитивно впливає на досвід його використання.

Він аналогічно попередньо згаданим рушіям містить потужну систему для роботи з графікою. Його графічний рушій забезпечує високий рівень візуалізації в режимі реального часу, при цьому зберігаючи низькі системні вимоги та гнучкість у налаштуванні. Для 2D-графіки Godot має окремий візуальний рушій, що дозволяє працювати зі спрайтами, анімацією, тайлсетами, освітленням, частинками та шейдерами. У 3D-режимі підтримуються освітлення, тіні, фізично-коректні матеріали , постобробка та ефекти — включно з рейтрейсингом у версії Godot 4.

Godot використовується не лише для створення 2D та 3D ігор для різних платформ як повноцінних додатків , завдяки вбудованому модулю експорту у WebAssembly / WebGL , продукти створені на йому часто мають форму браузерних ігор , а також інтерактивної реклами.

Щодо мов програмування – даний рушій може використовувати одразу декілька з них . GDScript є основною мовою Godot , що спеціально створена для його , вона тісно інтегрована з рушієм для легкої взаємодії з об’єктами на рівні та їх властивостями. Мова C# використовується рушієм для побудови складних проєктів та використання бібліотек .Net. VisualScript – блокова мова програмування , що має схожу структуру з раніше згаданими в Unreal Engine елементами Blueprints. Для нативних модулів для кращої продуктивності використовується мова C++.[10]

Порівняння ігрових рушіїв за ключовими параметрами є необхідним етапом , який допоможе визначити особливості кожного з них та прийняти найбільш оптимальне рішення щодо інструменту ,який найкраще підійде для реалізації плагіну процедурної генерації , що заодно допоможе з вибором мови програмування . Таке порівняння представлено в таблиці 3.1 :

Таблиця 3.1

Порівняння ігрових рушіїв за ключовими параметрами

Критерій	Unreal Engine 5	Unity	Godot
Ліцензія	Безкоштовно до $1 млн + роялті	Безкоштовно/ платна Pro	Повністю безкоштовна (MIT)
Мова	C++, Blueprints	C#, Visual Scripting	GDScript, C#, C++
Графіка	Висока (AAA-рівень)	Потужна, модульна (URP/HDRP)	Хороша для простих 3D/2D
Підтримка 2D	Обмежена	Повноцінна	Окремий 2D-рушій

Таблиця 3.1 (Продовження)

Простота освоєння	Висока складність	Середня	Найпростіший у входженні
Спільнота	Активна, орієнтована на AAA	Найбільша, різнорівнева	Швидко зростає, фокус на інді
Використання	AAA-ігри, симуляції	Мобільні, інді, VR/AR	Інді, прототипи, навчання

Проаналізувавши дану таблицю , опираючись на особливості поставленого завдання , було прийняте рішення обрати Unity Engine та мову C# в ролі основних засобів для розробки розширення процедурної генерації.

Рушій Unity на відміну від Unreal Engine має повноцінну підтримку 2D розробки ,простіший інтерфейс та мову програмування , а також більше фокусується на інді розробці.

Godot аналогічно міг би підійти для розробки даного продукту , так як основним напрямком розробки також є інді, однак поступається Unity в популярності серед спільноти розробників ігор , що в свою чергу є ключовим фактором для актуальності розширення.

Саме масштабність та активність спільноти значно спрощує процес розробки та скорочує час на вирішення технічних проблем. Спільнота також часто ділиться створеними плагінами та редакторськими інструментами , що позитивно впливає як на процес початкової розробки , так і подальшої підтримки через отримання зворотного зв’язку. І найголовніше - через велике ком’юніті поціновувачів Unity плагін буде актуальним та може бути застосованим в великій кількості проєктів та спростити розробку багатьом програмістам.

3.1.2 Вибір редактора коду

Редактор написання коду є не менш важди Оскільки C# є основною мовою програмування обраного рушія , в ролі редактора коду було вибрано інтегроване середовище розробки (IDE) Visual Studio , оскільки воно є офіційно рекомендованим для роботи з Unity та має повну сумісність з його проєктною структурою. Окрім базових характеристик ,що властиві більшості сучасних IDE

, таких як : підсвітка синтаксису та дебагер для кращого аналізу помилок логіки та логіки , тут наявна підтримка IntelliSense. IntelliSense – функція , яка значно спрощує процес написання коду ,дана функція може :автоматично доповнювати код та показувати підказки щодо помилок в синтаксисі , інформацію про параметри методів , опис змінних і функцій прямо під час розробки.[23]

3.1.3 Вибір інструментів рушія для реалізації плагіну

Після вибору рушія, як бази для створення плагіну , необхідно обрати вбудовані інструменти , що допоможуть реалізувати поставлене завдання.

Для реалізації інтерфейсу користувача було прийняте рішення використати компонент EditorWindow , оскільки він надає можливість створити вікно з таким же дизайном та властивостями , які мають вікна інструментів редактора ,що забезпечить цілісний візуальний вигляд редактора та плагіну , а також допоможе гнучко розміщувати створене вікно до будь якого елементу інтерфейсу.

Як основа для створення самого рівня я обрав компонент Tilemap ,так як він дозволяє користувачу зручно будувати рівень , використовуючи вирівнювання по сітці для цілісної структури та має власні інструменти редагування , що дає можливість використовувати їх для подальшого та більш точного редагування рівня та його структури залежно від потреб його проєкту.

В ролі конфігураційного файлу для збереження параметрів генерації між сесіями , було розглянуто файли форматів JSON та ASSET , що створюється на основі ScriptableObject, Серед них файл на основі ScriptabjeObject виявився ,більш доцільним варіантом для збережених даних , оскільки він забезпечує нативну інтеграцію з рушієм та може зберігати будь які серіалізовані дані , на відміну від JSON.

3.2 Організаційна структура ПЗ

3.2.1 Діаграма пакетів

Архітектура системи визначає загальну структуру програмного продукту , вона допомагає забезпечити зрозумілу організацію логіки плагіну. Найкращим способом представлення архітектури є діаграма пакетів , основним завданням якої є відображення основних логічних блоків системи у вигляді окремих модулів та зв’язків між ними. У межах даного проєкту діаграма пакетів (рисунок 3.1) , що створена відповідно до спроєктованої загальної архітектури ілюструє модульну архітектуру плагіну процедурної генерації: від інтерфейсу користувача до ядра генерації, утиліт та системи збереження конфігурацій.

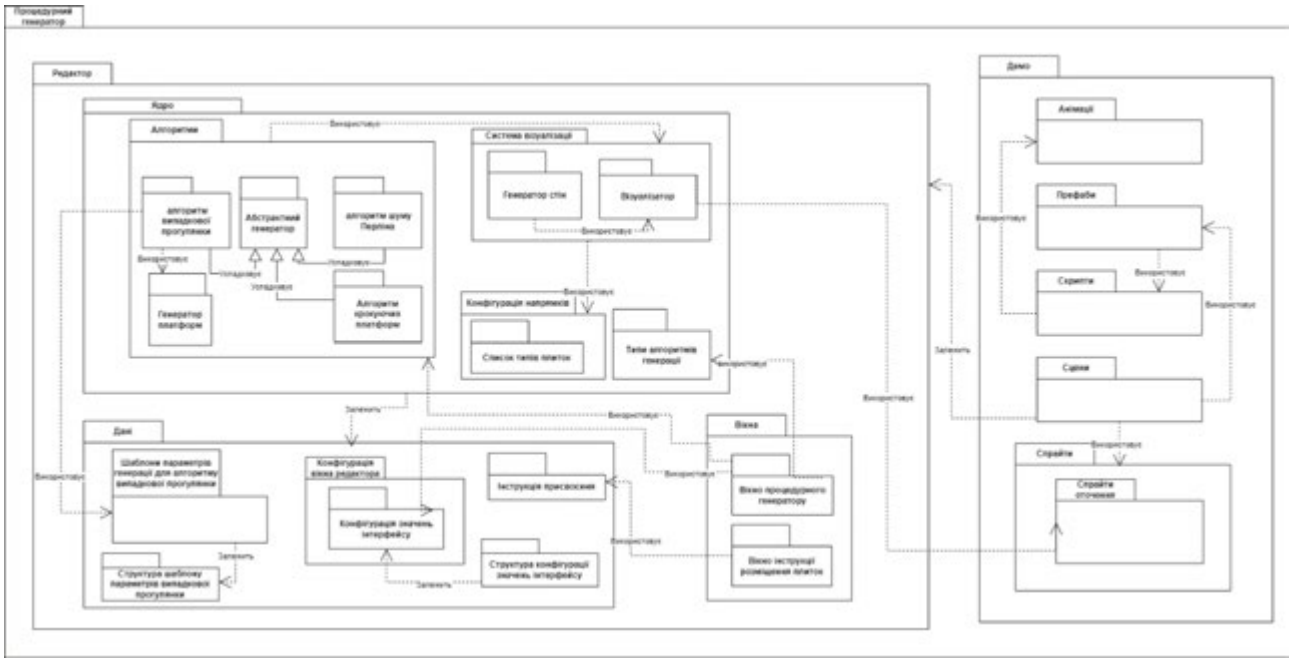


Рисунок 3.1 Діаграма пакетів

На поданій діаграмі пакетів представлена високорівнева архітектура програмної системи процедурного генератора, яка структурована на чотири основні логічні підсистеми: Редактор, Ядро, Дані та Демо. Така модульна побудова дозволяє забезпечити чітке розмежування обов’язків між компонентами та полегшує розширення й підтримку плагіна.

Редактор є початковою точкою взаємодії користувача з інструментом. У межах цього пакету знаходиться вікно процедурного генератора, яке є основним елементом інтерфейсу. Саме тут користувач обирає алгоритм генерації,

встановлює параметри та виконує запуск побудови рівня. Додатково в пакеті «Вікна» передбачено вікно інструкції розміщення плиток, яке допомагає у візуалізації структури тайлів та їхнього розміщення в межах Tilemap, що значно підвищує зручність при роботі з візуальними даними.

Ядро системи реалізує основну обчислювальну логіку та генерацію рівнів. Воно поділено на кілька внутрішніх модулів. Перший з них — модуль алгоритмів, який містить реалізації трьох основних алгоритмів: алгоритм випадкової прогулянки, алгоритм шуму Перліна та алгоритм ступінчатих платформ. Усі вони успадковують функціональність абстрактного генератора, що забезпечує уніфіковану структуру для ініціалізації, запуску генерації та взаємодії з візуалізатором. Окремим елементом у складі ядра виступає генератор платформ, який реалізує специфічну логіку для побудови платформ у платформер-проєктах.

Ще одним важливим компонентом ядра є система візуалізації, яка відповідає за графічне представлення згенерованого рівня на сцені редактора. Вона складається з візуалізатора, що безпосередньо працює з Tilemap, та генератора стін, який формує зовнішні та внутрішні границі кімнат, коридорів та інших елементів рівня. Ці компоненти використовують структуру даних про типи плиток, що дозволяє адаптувати зовнішній вигляд рівнів під потреби конкретного проєкту.

Окремий блок ядра — конфігурація напрямків, яка містить список типів плиток і типи алгоритмів генерації. Ці дані використовуються як у логіці генерації, так і в інтерфейсах для забезпечення узгодженості між візуалізацією, редактором та алгоритмами.

Підсистема "Дані" відповідає за зберігання конфігурацій та шаблонів, які дозволяють зберігати стан між сесіями, автоматизувати підстановку параметрів і спростити повторне використання налаштувань. Тут зберігаються шаблони параметрів генерації для кожного з алгоритмів, зокрема для випадкової прогулянки. Вони включають у себе повну структуру параметрів, яка може бути автоматично зчитана або збережена. Також у цьому пакеті зберігається

конфігурація вікна редактора, що містить значення інтерфейсу та інструкції присвоєння параметрів, які забезпечують автоматичне відновлення стану полів під час відкриття вікна. Для цього реалізовано окрему структуру конфігурації значень інтерфейсу.

Пакет "Демо" містить допоміжні компоненти, які використовуються для тестування плагіна в умовах гри. Тут зосереджено анімації, префаби, скрипти, сцени, а також окрему підпапку зі спрайтами оточення, які використовуються для візуального наповнення рівнів. Демо дозволяє перевірити, як згенеровані рівні працюють у реальному середовищі платформи, не виходячи за межі плагіна.

Загалом, представлена діаграма демонструє чітке розділення логіки між ядром, даними, вікнами редактора та демонстраційною частиною, що відповідає принципам модульності, повторного використання та масштабованості. Така структура дозволяє швидко вносити зміни до окремих частин проєкту без необхідності змінювати всю архітектуру, а також сприяє спрощенню процесу тестування та підтримки.

3.3 Алгоритмізація та програмування модулів

3.3.1 Реалізація алгоритмів генерації

Алгоритми процедурної генерації є основою функціональної частини плагіна , оскільки саме вони відповідають за побудову ігрових локацій на основі заданих параметрів. Основу для реалізації структури проєкту, а також логіки візуалізації локацій на сцені було частково запозичено з відкритого навчального курсу “Unity Procedural Dungeon Generation” від Sunny Valley Studio [11], значно вдосконалено, додано нові компоненти і алгоритми генерації та адаптовано під гру жанру платформер відповідно до постановки завдання.

Початково для всіх алгоритмів реалізований абстрактний клас “Abstract Procedural Generator” , що містить об’єкт модуля візуалізації TilemapVisualizer , початкову позицію та абстрактний метод RunProceduralGenerator , що буде в подальшому перевизначений в класах нащадках відповідних алгоритмів

генерації. Плагін буде будувати основу для рівня – локації у вигляді наборів тайлів на сцені , без заповнення елементами для взаємодії (Різноманітними ресурсами та NPC) , такий підхід застосований через залежність розміщення елементів від ігрових сценаріїв та видів визначних в грі елементів, що передбачити в межах генератору неможливо. Всі алгоритми процедурної генерації будуть визначати позиції стін , а компонент візуалізації буде відображати їх структуру з використанням графічних елементів -тайлів.

Код класу (Abstract Procedural Generator)

```
public abstract class AbstractProceduralGenerator :ScriptableObject
{
	[SerializeField] protected TileMapVizualizer _tileMapVizualizer ;
	[SerializeField] protected Vector2Int _startPosition;

	public void GenerateLocation()
	{

		RunProceduralGeneration();
	}
	public void Initialize(TileMapVizualizer tileMapVizualizer)
	{
		_tileMapVizualizer = tileMapVizualizer;
	}
		protected abstract void RunProceduralGeneration();
	}
```

Алгоритм випадкової прогулянки в класичному представленні являє собою побудову послідовного ланцюга клітинок , які починаючи з певної початкової позиції рухаються у випадковому напрямку на встановлену довжину на кожному кроці просування змінюючи свій напрямок на випадковий. Особливості цього алгоритму дозволяють йому формувати непередбачуваний результат навіть за однакових параметрів , що в контексті процедурної генерації вносить різноманітність в ігровий процес та остаточну структуру рівня.

Як було зазначено в пункті 1.4.4 ,для даного плагіну , алгоритм випадкової прогулянки буде будувати локації в стилі roguelike з частинами кімнат та коридорів. Для створення такої структури, логіку генерації було розділено між чотирма основними класами :

- RandomWalkAlgorithm , що містить основу алгоритму для створення кімнат та являє собою реалізацію алгоритму в чистому вигляді.

- RandomWalkRoguelikeLocationConstruction , який містить функції для побудови кімнат та коридорів , що їх з’єднують.
- RandomWalkLocationWithCorridorsGenerator – клас , що містить фінальну реалізацію формування локації з подальшою візуалізацією та додатковими параметрами.
- Direction2D – клас ,що містить базові напрямки представлені в векторному вигляді для коректності роботи генератора.

Параметрами для алгоритму випадкової прогулянки є кількість ітерацій , їх довжина і необхідність починати кожну ітерацію випадковим чином . Загальний алгоритм представлений на блок схемі , зображеній на рисунку 3.2. Програмні коди реалізованого алгоритму представлений в додатку Б.

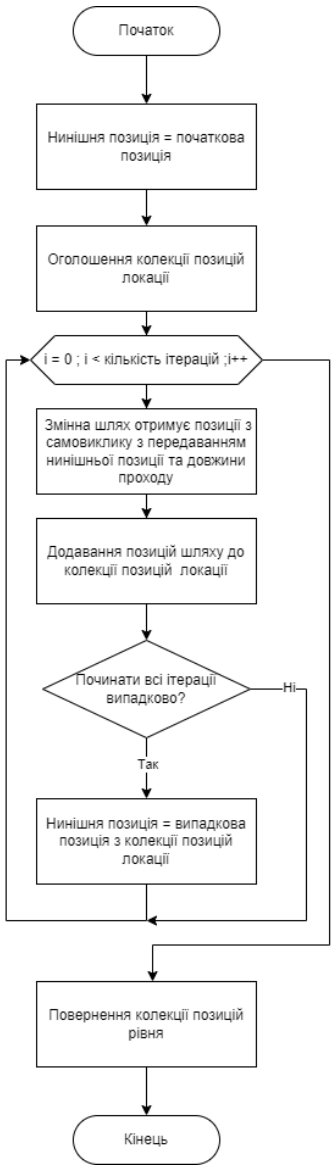


Рис. 3.2 Блок схема алгоритму випадкової прогулянки

Набори параметрів генерації було винесено в окремий файл формату .asset реалізований через клас ScriptableObject , оскільки параметри генерації вливають лише на фінальний розмір рівня , однак не можуть забезпечити повторюваність структури елементу локації для кожного набору значень.

Логіка побудови коридорів реалізована шляхом послідовного будування шляху заданої довжини з вибором випадкового напрямку для повороту. Алгоритм представлений графічно у вигляді блок схеми на рисунку 3.3 .

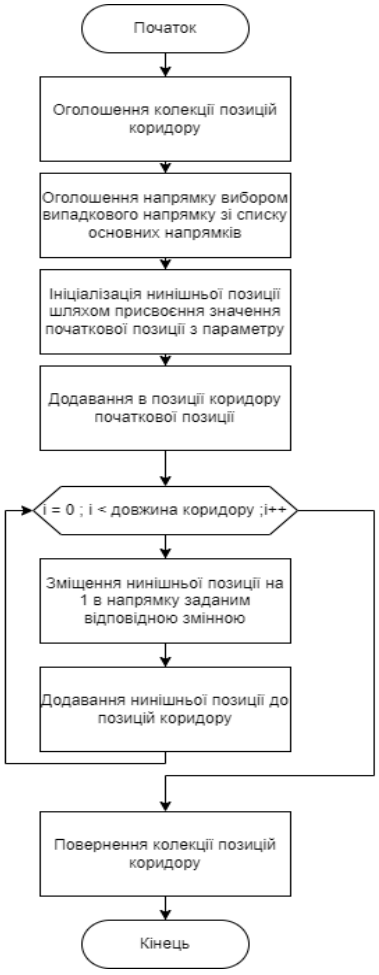


Рис 3.3 Блок схема алгоритму побудови коридору

Фінальний алгоритм передбачає початкове створення коридорів , після чого відбувається створення кімнат яке відбувається відповідно до встановленого значення імовірності їх генерації , пошук закінчень коридорів без кімнат , та розміщення кімнат в їх межах .Також для генерації фінальної версії рівня був створений перелік додаткових функцій для урізноманітнення створеної локації серед яких :

- очищення внутрішньої частини рівня , що представлена підлогою ,

- очищенняя поодиноких блоків , які можуть виникати через особливості алгоритму,
- додавання фону в форматі заповнення вільного простору навколо сформованих обрисів локації з додаванням рамки заданої товщини для повноцінного візуального вигляду.
- розміщення елементів платформ всередині створених обрисів використанням алгоритму випадкової прогулянки.

Щодо останньої додаткової функції був створений новий алгоритм генерації ,який працює шляхом виділення порожньої області в межах локації та розділення її на задані користувачем сектори з подальшим автоматичним встановленням параметрів для алгоритму випадкової прогулянки які створюються залежно від початкових розмірів сектору. Такий підхід дозволяє створювати платформи непередбачуваної форми , що в свою чергу завдяки вбудованим елементам редактора може бути вдосконаленим до структур , що зможуть додати різноманітності рівню. Код даної функції та всіх описаних класів представлений в додатку Б.

Підхід до реалізації алгоритму генерації з використанням структури локації як в іграх жанру roguelike дозволив створити універсальний алгоритм , що може створювати як порівняно прості локації, що виглядають як суцільна структура шляхом встановлення невеликого значення параметру довжини коридорів та великими значеннями параметрів побудови кімнат, так і локації , що мають структуру лабіринтів та візуально нагадують структуру рівня гри жанку rogue like.Комбінування різних параметрів та як наслідок створення локацій різних типів в межах одного рівня зроблять ігровий процес більш різноманітним та значно спростять роботу розробника.

Алгоритм шуму Перліна базується на використанні математичної функції , основним призначенням якої є генерація плавних псевдовипадкових значень , що змінюються без різких стрибків. Дана функція була створена Кеном Перліном спеціально для комп’ютерної графіки в 1983 році та активно

використовується й досі в різних сферах комп’ютерної графіки , а також в розробці ігор.[12]

В стандартному контексті процедурної генерації , шум Перліна використовується для ігрових проєктів для побудови ландшафтів та карт висот . Виконується така генерація шляхом обчислення значень шуму в певних точках простору та подальшого використання цих значень для побудови карти або структури. Розглянемо його графічне представлення (див Рис. 3.6) , на якому продемонстровані значення шуму в чорному та білому кольорах. Якщо розглянути це з точки зору алгоритму для процедурної генерації , то якщо до певного значення кольору задати параметр , що буде розміщувати певний елемент локації , або певну висоту (наприклад для світлих відтінків кольору висота більша ,а для темних - менша), то на основі такого підходу можна сформувати раніше згадані ландшафт та карту висот.

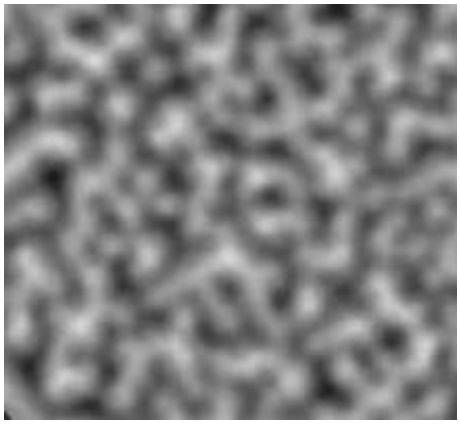


Рис. 3.6 Графічне представлення шуму Перліна

Такий же підхід був використаний для побудові локації в плагіні процедурної генерації , проте для 2D рівня буде використовуватись співвідношення кольорів (в математичному представленні - значення функції) відносно параметру порогового значення , що дозволить сформувати структуру схожу на візуальне представлення ,де один з кольорів буде представлений стінами ,а інший -пустотою, та коригувати значення кількості стін в межах локації (за меншого значення порогу стін буде менше та навпаки). Також велике значення мають параметри:

- Масштаб , що визначає кількість нерівностей в межах локації (чим масштаб більший , тим дрібнішими будуть елементи локації)
- Seed – значення , що використовується як початкове при ініціалізації генератора випадкових чисел, який в результаті формує зміщення шуму. Для кожного значення seed існує своя структура локації , відповідно для однакових значень цього параметру буде створено однакові локації.

Візуалізація алгоритму роботи з шумом Перліна для процедурного генератора представлена на рис. 3.4 .

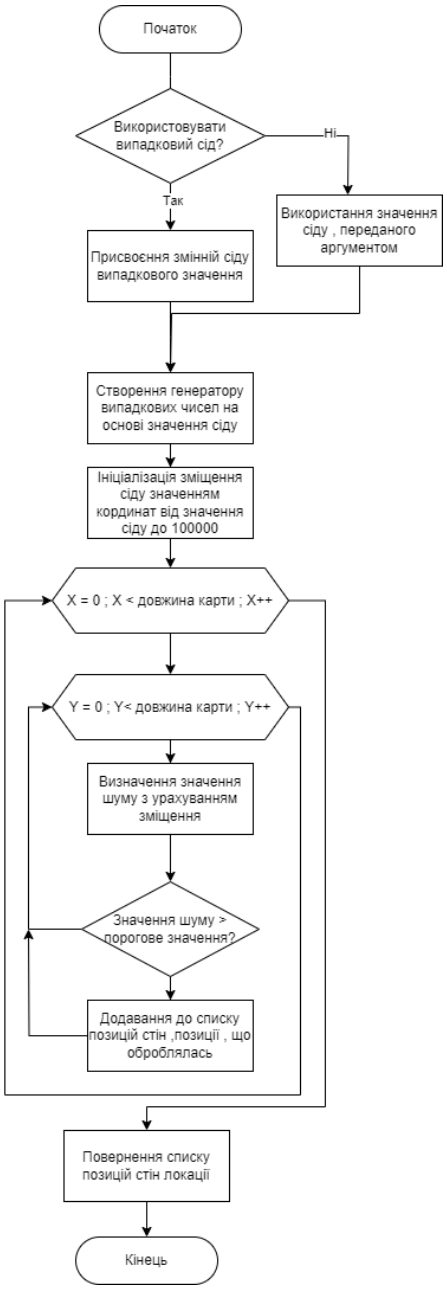


Рис. 3.4 Блок схема алгоритму побудови локації з використанням шуму Перліна

Реалізований алгоритм дозволяє створювати локації які можуть бути схожими як на печери , так і на літаючі острови , залежно від порогового значення. Для його була також додана додаткова функція , що формує рамку навколо локації. Фінальна реалізація у вигляді програмного коду наведена в додатку Б.

Алгоритм ступінчатих платформ не є загальноприйнятим терміном у сфері ігрової розробки та процедурної генерації і був сформований визначенням ключових особливостей зі статті про створення генератора безкінечного рівня для платформера [13].

Основа алгоритму полягає в представленні області локації у вигляді сітки та випадковому розміщенні прямокутних структурних частин – платформ , що мають різні довжину та товщину , а також шар , представлений координатами Y для для кожного значення координати X , що формує структуру , що схожа на таблицю , але з елементами , що виступають за межі комірок , а сама “таблиця” заповнена нерівномірно, що схоже на сходинки.

Даний алгоритм реалізований в плагіні шляхом створення ігрової області та виключення з неї блоків, що представляють платформи. Визначеними необхідними параметрами для генерації є :

- Розміри локації,
- Мінімальна та максимальна довжини платформ,
- Мінімальна та максимальна товщини платфор,
- Мінімальна та максимальна координати Y в заданій області,
- Імовірність пропуску платформ,
- Значення seed.

На початку викликається метод GenerateFullPlatformBlocks(), що формує набір платформ у вигляді прямокутних областей випадкової довжини, товщини та висоти. Їх кількість і розміщення визначаються генератором випадкових чисел, який може ініціалізуватися фіксованим seed, що дозволяє відтворювати одні й ті самі результати при повторному запуску генерації.

Платформи розміщуються із випадковим кроком по горизонталі. У деяких місцях генерація може пропускатись із заданою ймовірністю (параметр _gapChancePercent), щоб створити варіативність у щільності рівня. Після генерації набір координат платформ додається на основну локацію, а решта простору інтерпретується як порожня частина. Ця область передається у візуалізатор, який малює встановлені плитки, додає стіни, заповнює внутрішні ділянки та може додатково їх очищати або коригувати за допомогою додаткових методів. У результаті формується рівень зі структурованим, але випадковим ігровим середовищем, адаптованим для платформерів. Візуальне представлення алгоритму у вигляді блок-схеми представлено в додатку А.

3.3.2 Система візуалізації рівня

Система візуалізації процедурного генератора була розділена між чотирма класами відповідно до їх призначення:

- TileMapVisualizer - Головний клас даної підсистеми , що малює всі тайли та визначає який тайл для кожного типу стіни необхідно встановити,
- WallGenerator - Статичний клас ,що генерує базові та кутові стіни на основі сусідства з Tilemap порожньої частини рівня.
- Direction2D - Статичний клас , що містить набір статичних даних , що визначає напрямки .
- WallTypesHelper - Статичний клас , що містить всі можливі типи стін у форматі бітових масок.

Алгоритми візуалізації, реалізовані для даного плагіну, відповідають за генерацію стін на базі раніше створених позицій порожнього простору локації.

Початковим джерелом інформації є набір координат EmptySpacePositions, які позначають місця, де вже існує рівень в представленні його внутрішньої частини , яка може бути реалізованою у вигляді підлоги для 2D проєктів з видом зверху (наприклад жанру roguelike), а також у вигляді порожньої області для ігор з видом з боку (наприклад жанру platformer). На основі цього набору алгоритм

визначає, які клітинки не містяться з внутрішньою частиною, але безпосередньо межують із нею, - це й є потенційні позиції для стін. Для цього використовується метод FindWallsInDirections, який перевіряє як основні напрямки (вгору, вниз, ліво, право), так і діагональні (наприклад, правий верхній кут), щоб знайти сусідів підлоги, які самі не є плитками внутрішньої частини.

У процесі створення стін система поділяє стіни на дві великі категорії: базові та кутові. Базові стіни генеруються на клітинках, що межують із підлогою лише в основних чотирьох напрямках. Для кожної такої клітинки формується бінарний код (наприклад, "1010"), де кожна цифра позначає, чи є сусід у певному напрямку. Цей код потім перетворюється у десяткове число і порівнюється з наборами масок у WallTypesHelper, щоб визначити тип стіни (наприклад, верхня, ліва, права, нижня, або повна). Аналогічним чином ,але з урахуванням всіх восьми напрямків. Метод CreateCornerWalls обчислює восьмибітний бінарний код, який відображає наявність сусідів навколо поточної клітинки. Знову ж, цей код порівнюється з наборами масок у WallTypesHelper, щоб визначити точний тип стіни, включаючи внутрішні та зовнішні кути, висячі стіни, прямі вертикальні, навислі або повністю оточені.

Загальна послідовність дій для визначення типу стіни починається з вибору позиції стіни , тип якої необхідно знайти , на Рисунку 3.7 зображена схема визначення типу стіни на прикладі однієї з кутових стін ,для якої використане обчислення восьмибітного бінарного коду . На схемі синім кольором зображена позиція стіни , для якої потрібно визначити тип. Білим кольором позначені позиції уже сформованих обрисів внутрішньої частини локації , а червоним – позиції , які не входять до внутрішньої частини. Таким чином , рухаючись за годинниковою стрілкою починаючи з комірки в другому стовпці першого рядка даної таблиці , необхідно сформувати двійковий код для визначення типу ,який формується записом “1” для позицій ,що належать внутрішній частині та “0” для позицій , що їй не належать. Таким чином , виконавши прохід по всіх сусідніх позиціях , можна отримати запис “11000011” , що являє собою бінарний код , який міститься в класі WallTypeHelper та в подальшому переводиться в

десяткове значення використанням в класі TilemapVisualizer для подальшого встановлення спрайту відповідно визначеного типу через PaintSingleWall. Якщо тайл вже існує в даній позиції, малювання пропускається.

Клас WallTypesHelper відіграє ключову роль у класифікації типів стін. Він містить численні хеш-набори чисел, що представляють бітові маски для кожного типу стіни , які можуть зустрічатись на сцені. Ці маски кодують певні комбінації наявності сусідів і дозволяють швидко та ефективно визначати, який саме тайл слід використати в конкретній клітинці.

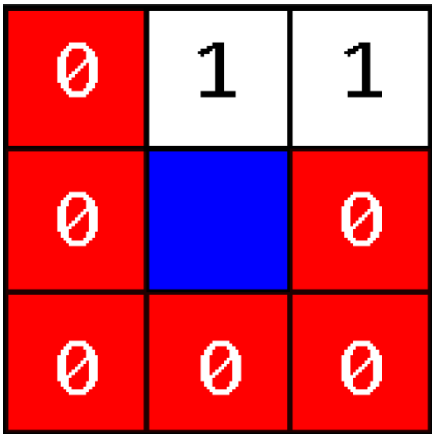


Рис. 3.7 Схема визначення типу стіни

Візуалізацію стін та підлоги здійснює клас TileMapVisualizer. Він ініціалізується двома Tilemap-об'єктами — для підлоги та стін — і відповідним набором TileBase, кожен з яких представляє певний тип тайла. Цей клас забезпечує методи для малювання підлоги, окремих стін, заповнення порожніх зон, очищення тайлів, видалення рамок тощо. При потребі, TileMapVisualizer може також видалити одинокі тайли всередині платформи або замінити їх на інші.

Завдяки цій архітектурі, система дозволяє гнучко та ефективно створювати детальні сцени, адаптуючи типи стін до їх оточення без ручного розміщення. Бінарні маски забезпечують швидку ідентифікацію оточення кожної клітинки, а структура на основі хеш-наборів гарантує високу продуктивність навіть на великих мапах. У підсумку, ця реалізація демонструє організовану процедурну генерацію структури рівня на основі аналізу просторових зв’язків між елементами. Програмні коди ключових класів розміщені в додатку Б.

3.3.3 Інтерфейс редактора та керування параметрами

Інтерфейс користувача плагіну реалізовано за допомогою editor window , що надає йому вбудовані параметри дизайну середовища , що у поєднанні з параметром a “dockable” надає вікну можливість приєднуватися до інших вікон редактора для формування цілісної структури середовища. Використання інструментів редактора також забезпечує інтерактивний підхід до встановлення параметрів способом перетягування елементів з ієрархії проєкту одразу в поле параметру.

Інтерфейс був розроблений англійською мовою ,оскільки в самому середовищі рушія відсутня українська, реалізація та використання мови , що буде відрізнятись від встановленої в редакторі, буде недоцільним. Станом на 2020 рік , опираючись на дані Nexwell[18] та VentureBeet[19] серед 1.5 млн розробників , що активно використовують Unity , українських розробників лише 3 тисячі , що становить 0.2% від загальної кількості , що в свою чергу підкреслює доцільність використання саме англійської мови для інтерфейсу.

Для кращого орієнтування в призначенні параметрів , для кожного поля параметрів були створені випадаючі підказки , що дозволяє зрозуміти призначення без аналізу коду. Загальний вигляд дизайну , що представлений у вигляді мокапів, був повністю реалізований в Unity (Рисунок 3.8) , головна вкладка містить вибір алгоритмів генерації , їх параметри та додаткові інструменти , що допомагають гнучко взаємодіяти зі створеною локацією.

Серед додаткових функцій передбачено визначення початкової точки на сцені ,яку можна обрати просто натиснувши на необхідну позицію на сцені , а також області для генерації , шляхом її виділення . Така інтеграція з інструментами сцени значно пришвидшує роботу розробника та зручність задання параметрів. Кнопки очищення вмісту рівня реалізовані з прив’язкою до встановлених об’єктів з типом “Tilemap” , внаслідок чого , вони будуть відображатись лише коли встановлені ці параметри , для коректної роботи розширення ,а для генерації у випадку відсутності таких параметрів , вони будуть автоматично знайдені ,або створені на сцені відповідно.

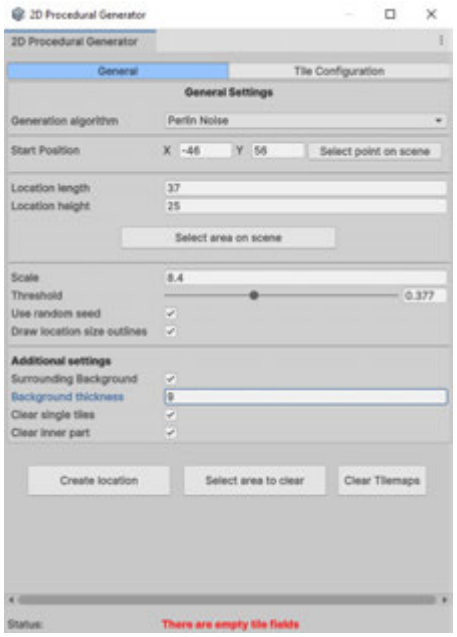


Рис. 3.8 Дизайн головної вкладки реалізованого інтерфейсу

Вкладка з встановленням конфігурації плиток представлена у такому ж стилі , як і попередня . На ній знаходяться поля ,в які будуть встановлюватись тайли , а також кнопка , що викликатиме інструкцію розміщення цих тайлів у вигляді окремого вікна .Вікно інструкції містить зображення елементів окремого набору ресурсів [17], який містить всі необхідні тайли з підписами назв , для кращого орієнтування користувача серед параметрів. Вигляд реалізованих інтерфейсів вкладки встановлення тайлів представлений на рисунку 3.9 ,а вікна інструкцій на рисунку 3.10.

Рис. 3.9 Дизайн вкладки налаштування тайлів реалізованого інтерфейсу

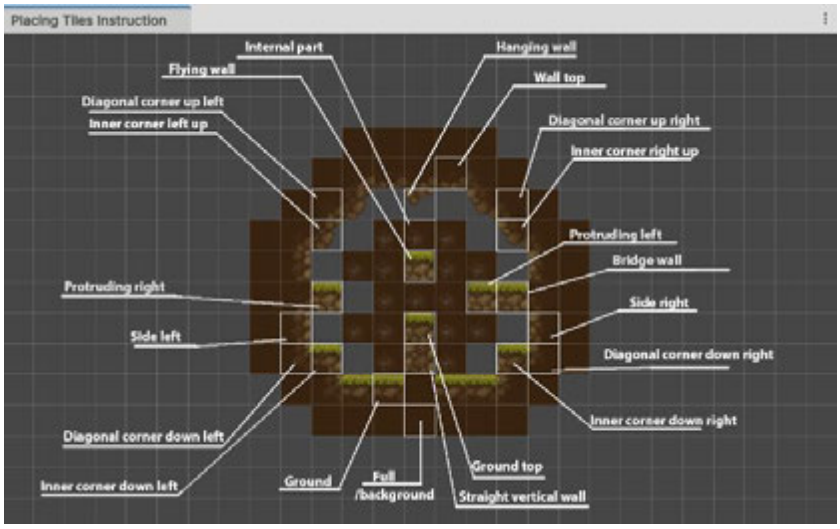


Рис. 3.10 Дизайн вікна інструкцій реалізованого інтерфейсу

3.3.4 Демонстраційна версія гри

Для демонстрації прикладу використання плагіну , було розрозблено демонстраційну версію гри – платформера , що представлена набором готових шаблонів елементів рівня ,включаючи ігрових персонажів та елементи оточення , з якими може взаємодіяти гравець , використаний набір ресурсів та створені анімації , без складної реалізації всіх елементів повноцінної гри. Для розробки був використаний набір ресурсів в мінімалістичному стилі з графікою в стилі “Pixel art” [20] , перелік елементів якого представлений на рисунку 3.11 .

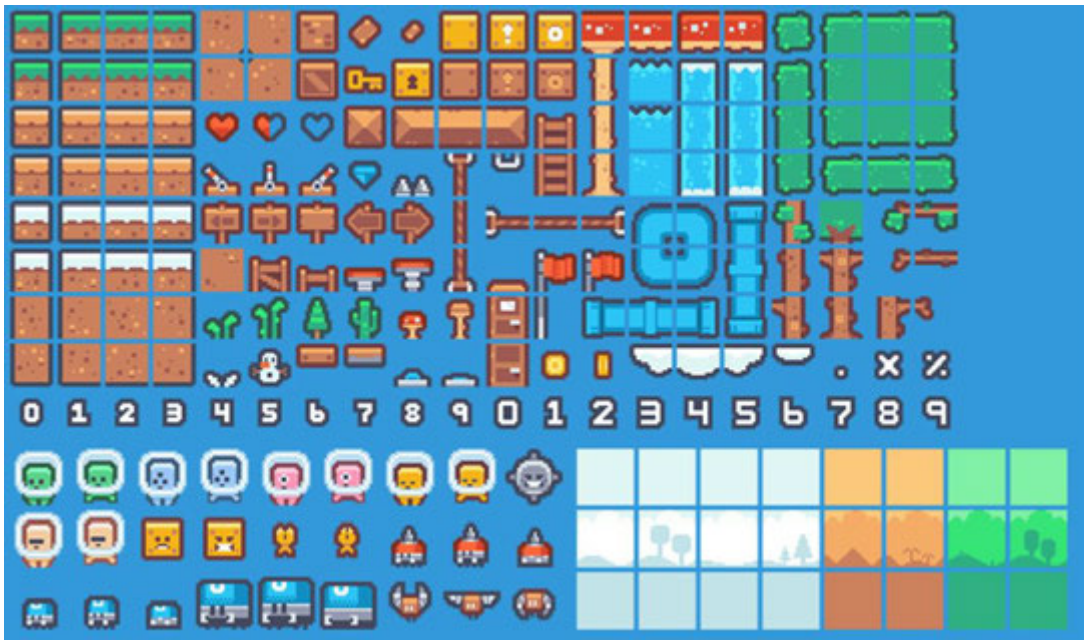


Рисунок 3.11 Набір елементів ,що був використаний для демонстраційної версії гри

Демонстраційна версія гри є дуже важливою частино , оскільки вона дозволяє розробнику :

- Побачити як саме, як саме згенеровані локації можуть використовуватись у реальній ігровій сцені.
- Миттєве тестування результату генерації у ігровому середовищі - наприклад, проходженням рівеня, перевіркою логіку руху персонажа або взаємодію з платформами.
- Використовувати як приклад правильної конфігурації та допомагають новим користувачам швидко зрозуміти, як використовувати генератор для проєкту та встановлювати параметри.
- Повторно використовувати ігрові об’єкти ,або адаптовувати їх для власних потреб - як основу майбутнього прототипу гри.

В процесі реалізації були створені наступні основні типи персонажів та об’єктів:

- Головний персонаж має можливості переміщення , може отримувати шкоду від противників та завдавати їм.
- Наземний противник – рухається доки не зіткнеться з стіною ,або не закінчиться земля по якій він переміщується . Може бути знищеним гравцем , за потрапляння останнього на вразливу зону.
- Літаючий противник – переміщується між двома встановленими точками , тимчасово затримуючись на кожній з них.
- Вдосконалений літаючий противник – може переміщуватись одразу між декількома точками та створювати ігрові об’єкти , що будуть падати та наносити шкоду гравцю .
- Пастка, що обертається – має точку опори , навколо якої здійснює коливання та представляється у вигляді перепони для гравця.

- Підземна пастка – захована під шаром ігрового рівня та висовується з під землі , формуючи перепону для гравця , який при контакті отримувати шкоду.
- Приєднані платформи – представлені плоскими продовгуватими елементами , на які можуть переміщувати гравця, прив’язуючись до контрольних точок , обертатись , стфорюючи перепону , або коливатись під вагою гравця.
- Монета – елемент винагороди , які має збирати гравець.
- Точка завершення рівня – елемент , у вигляді прапорця ,який при зіткненні з гравцем завершує гру.

Для даного прототипу гри існує простий цикл ,який працює поки гравець не дійде до контрольної точки. Ігровий цикл демо версії представлений у вигляді блок схеми в додатку А .

Розроблені програмні коди , що реалізують логіку основних компонентів циклу гри наведено в додатку Б. Приклад побудови рівня на основі використання демо версії гри та плагіну процедурної генерації представлений в додатку В.

Висновок до розділу 3

В даному розділі було реалізовано основну частину прикладного проєкту - створення функціонального плагіну процедурної генерації рівнів для ігор у середовищі Unity. Було обґрунтовано вибір рушія Unity Engine як оптимального інструменту завдяки його підтримці 2D-розробки, простому у засвоєнні інтерфейсу та потужному інструментарію. Мовою програмування обрано C# — мову, що добре підтримується в межах середовища та забезпечує баланс між зручністю і продуктивністю.

Архітектура програмного забезпечення була побудована на модульному підході, що забезпечує гнучкість, повторне використання компонентів і зручність подальшого розширення. Всі компоненти розподілено між

підсистемами "Редактор", "Ядро", "Дані" та "Демо", що спрощує інтеграцію, підтримку та тестування плагіну.

Ключовим функціональним елементом розділу стала реалізація трьох алгоритмів процедурної генерації: випадкової прогулянки, шуму Перліна та ступінчатих платформ. Для всіх методів генерації був реалізований алгоритм виконання у формах програмного коду та блок схем, реалізовано засоби візуалізації рівнів на основі тайлів, а також додано додаткові функції. Система візуалізації, побудована на базі Tilemap, дозволяє автоматично формувати типи стін за допомогою бітових масок та їх класифікації.

Окрему увагу було приділено створенню зручного інтерфейсу користувача з можливістю динамічного налаштування параметрів генерації, вибору алгоритмів, інтеграції з інструментами сцени та підтримкою збереження конфігурацій. Створене демо-гри ,що дозволяє перевірити працездатність алгоритмів у реальному геймплеї, а також спростити ознайомлення нових користувачів з розширенням.

Отриманий результат підтверджує доцільність обраного підходу та інструментів, а також дозволяє розглядати розроблений плагін як готовий до використання інструмент для створення різноманітних типів ігрових локацій з мінімальними затратами часу на ручне проєктування.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Впровадження системи – ще один важливий етап її розробки, являє собою перехід від розробки до експлуатації користувачами та подальшої підтримки продукту. Кожен проєкт перед публікацією проходить через етап тестування , для демонтрації та впровадження повністю працюючого продукту.

Тестування програмного забезпечення являє собою процес перевірки коректності його роботи, який допомагає визначити відповідність вимогам , правильність виконання логіки модулів та наскільки вихідні дані відповідають очікуванням.

Тестування програмного забезпечення поділяють за видом на функціональне та нефункціональне , а також за методом - на автоматичне та ручне. Вони відрізняються реалізацією та видами дефектів , що визначаються.

Для функціонального тестування об’єктами дослідження є функції та поведінка програми , в той час, як для нефункціональне оцінює відповідність якісним вимогам , таким як зручність , безпека та продуктивність.

Методи тестування відрізняються використанням програмних засобів та інструментів для автоматичного та повним контролем процесу перевірки для ручного.[21]

Тестування для даної системи буде проводитись в змішаному форматі , ручним методом з відтворенням різних сценаріїв використання системи. Сценарії будуть включати в себе такі аспекти :

- Технічні аспекти ,пов’язані з автоматичним створенням та визначенням ігрових об’єктів на сцені;
- Візуальна відповідність створених локацій з описаними раніше описаними очікуваними результатами генерацій (див підрозділ 3.3);

- Логічна обробка введених параметрів генерації ;

Почати варто з перевірки створення локацій на повністю порожній сцені. Для даного тесту була створена нова сцена та задані коректні параметри для побудови локації методом випадкової прогулянки без внутрішньої частини та поодиноких блоків , не включаючи встановлення Tilemap як параметрів.

В результаті тесту були створені об’єкти в ієрархії , що формують сітку та дві мапи плиток та побудована локація(Рис 4.1).

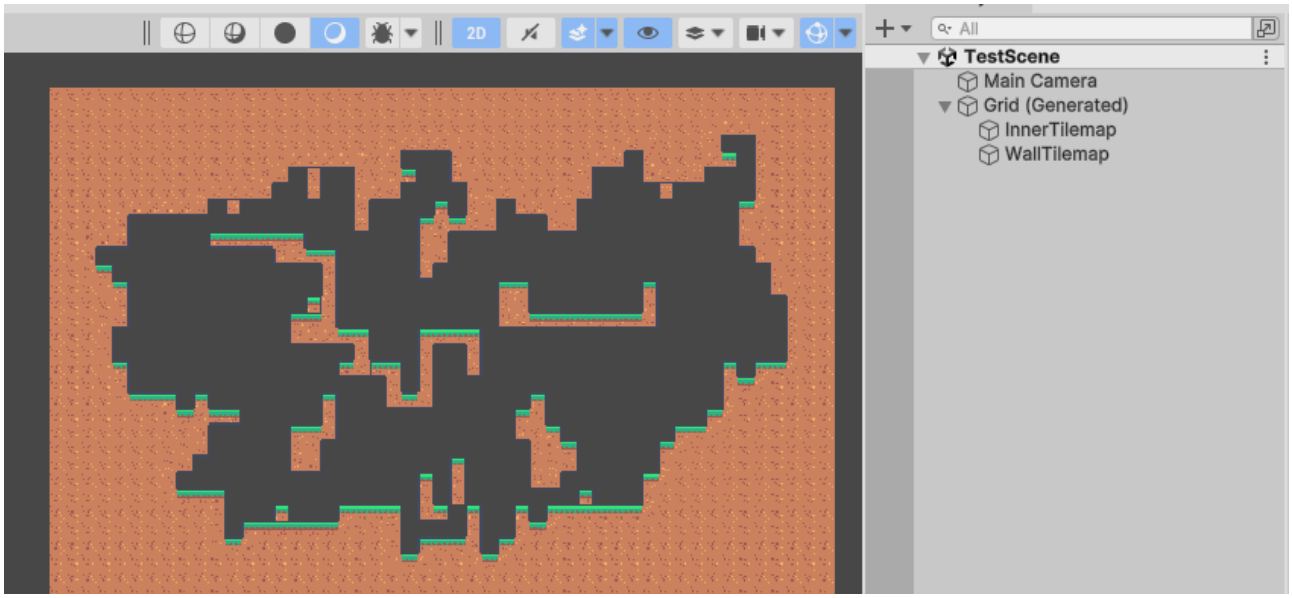


Рис 4.1 Створені елементи в ієрархії та побудована локація

Тепер варто перевірити чи зможуть створені елементи бути автоматично визначеними ,після видалення їх з полів параметрів (рис. 4.2).

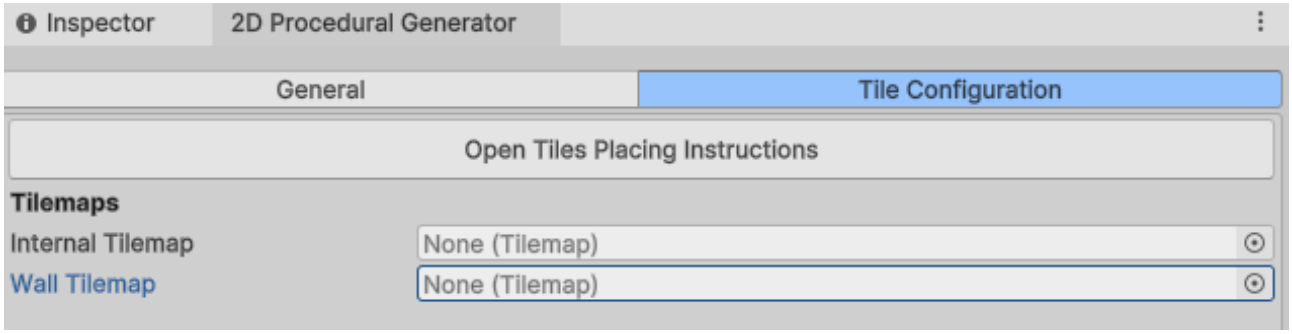


Рис. 4.2 Вигляд полів параметрів до проведення тесту

В результаті тесту поля були заповнені коректно та побудована локація (рис. 4.3).

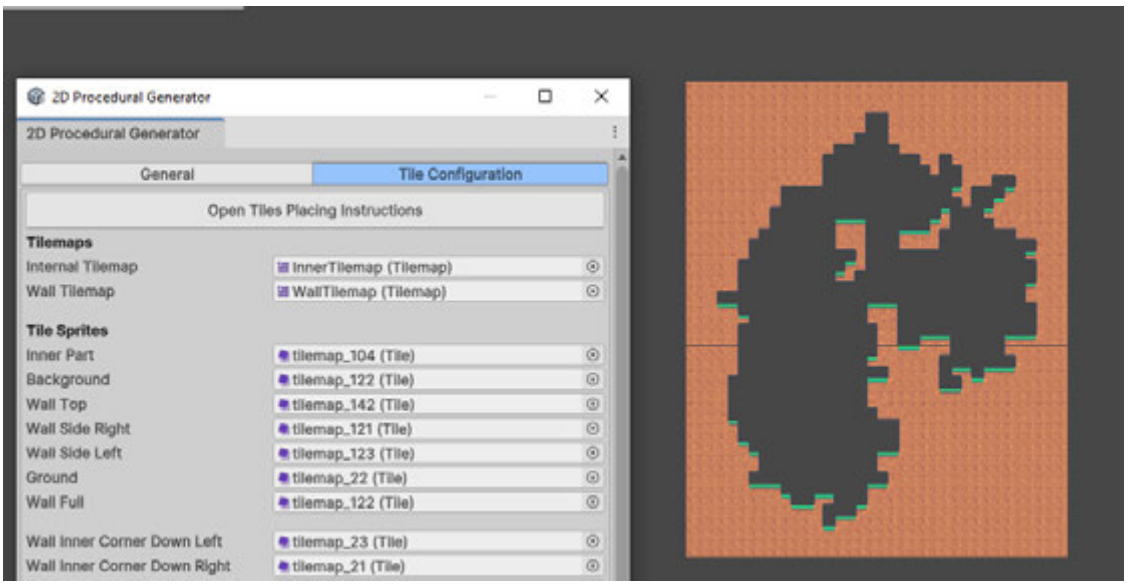


Рис. 4.3 Згенерована локація з автоматично визначеними полями

Наступним кроком стане перевірка вводу від’ємних значень для різних алгоритмів .Для алгоритму випадкової прогулянки очікуваним результатом буде сформована рамка рівня без оточення та платформ всередині , через особливості логіки алгоритму ,а також повідомлення в консоль про помилку встановленого значення для генерації платформ. Результат тестування співпав з очікуваним та продемонстрований на рисунку 4.4.

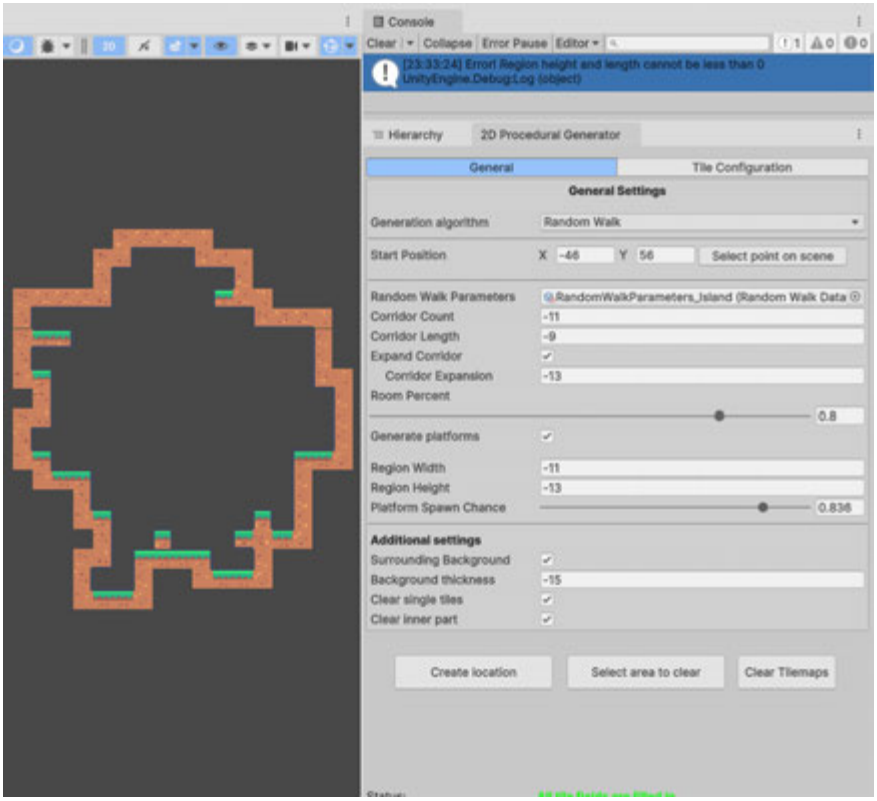


Рис. 4.4 Результат тестування на від’ємні параметри для алгоритму випадкової прогулянки

Аналогічну перевірку можна провести для алгоритму шуму Перліна , де в результаті також очікується повідомлення про помилку через встановлену перевірку. В результаті тесту ,локацію не було створено , а в консолі отримана помилка про хибно встановлені значення (рис. 4.5).

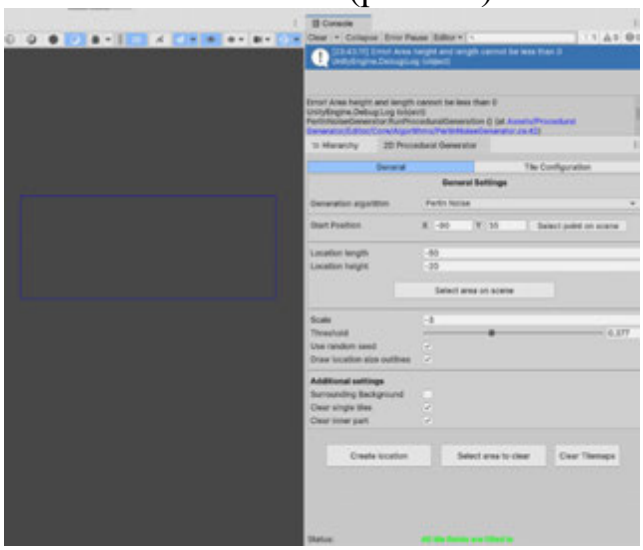


Рис.4.5 Результат тестування на від’ємні параметри для алгоритму шуму Перліна

Для алгоритму ступінчатих платформ реалізована велика кількість перевірок , для відсутності накладання значень та створення неможливих проміжків числових даних , тому очікуваний результат для даного алгоритму також виведення повідомлення в консоль без створення локації . Результат тестування також співпадає з очікуваним і в консолі виводиться повідомлення про помилку , та локація не створюється (рис. 4.6).

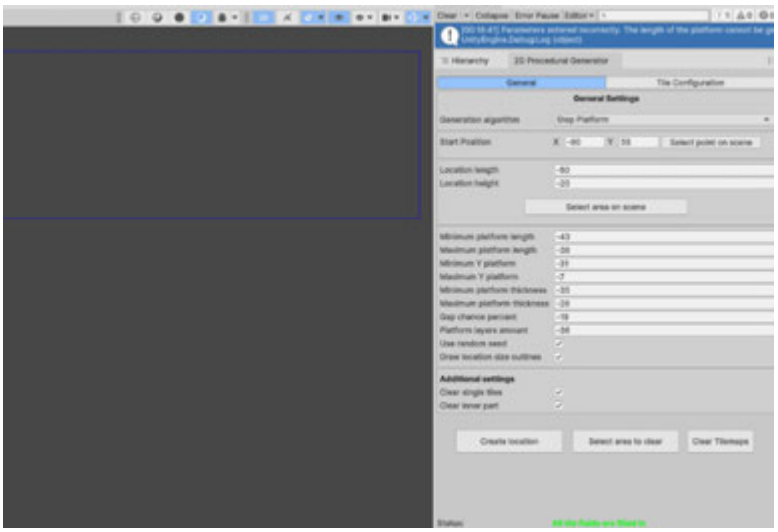


Рис. 4.6 Результат тестування на від’ємні параметри для алгоритму ступінчатих платформ

Тепер необхідно провести тестування для структури генерованих локацій . Для алгоритму випадкової прогулянки передбачалось створення простих та лабіринтоподібних локацій , відповідно при встановленні невеликих значень кількості та довжини коридорів , структура мапи має формуватись у вигляді кімнати ,або декількох кімнат ,що накладаються одна на одну і в результаті локація має нагадувати кімнату непередбачуваної форми та збільшеного розміру відносно параметрів створення кімнат, а для параметрів з більшою довжиною коридорів та їх довжини повинна будуватись лабіринтоподібна локація. Результати тестування (рис 4.7 , рис 4.8) реалізують генерацію локації в очікуваних формах , алгоритм працює коректно.

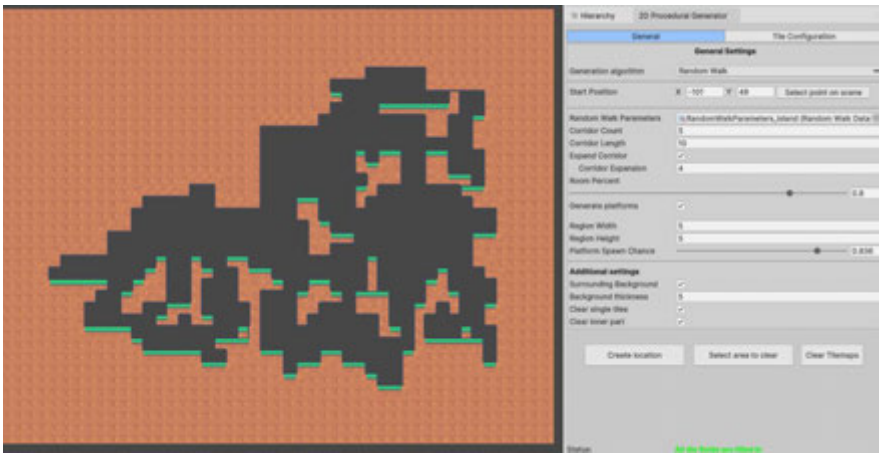


Рис. 4.7 Результат тестування при малих значеннях для алгоритму випадкової прогулянки.

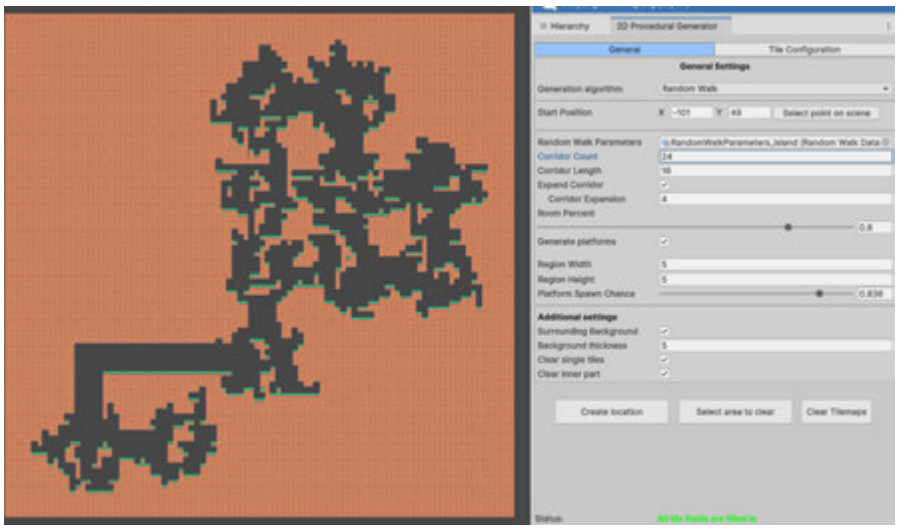


Рис. 4.8 Результат тестування при великих значеннях для алгоритму випадкової прогулянки

Для алгоритму шуму Перліна ключовими параметрами є значення сіду ,яке має повторювати структуру генерації за однакового значення та поріг , який має визначати кількість наявних стін та за більшого значення зменшувати її , а також параметр масштабу , що має змінювати структуру локації , створюючи складнішу структуру локації.

В результаті перевірки коректності роботи модулів було визначено , що при однакових значеннях параметру seed , генерується однакова локація , а при більшому значенні порогу значно змінюється наповненість рівня , на рисунку 4.9 це продемонстровано . На ньому зображені дві локації , остання з яких оточення синьою рамкою , для кожної з них застосований однаковий сід , але різне значення порогу , що помітно через різний рівень наповненості локацій та схожих помітних елементах структури , що відповідає очікуваним результатам . Щодо унікальної генерації при застосуванні різних значень сіду - результат перевірки можемо побачити на рисунку 4.10 , де окрім унікального значення сіду збільшено масштаб для другої генерації (що оточена рамкою синього кольору) , внаслідок чого , структура локації складніша за однакового значення порогу , що свідчить про успішне тестування .

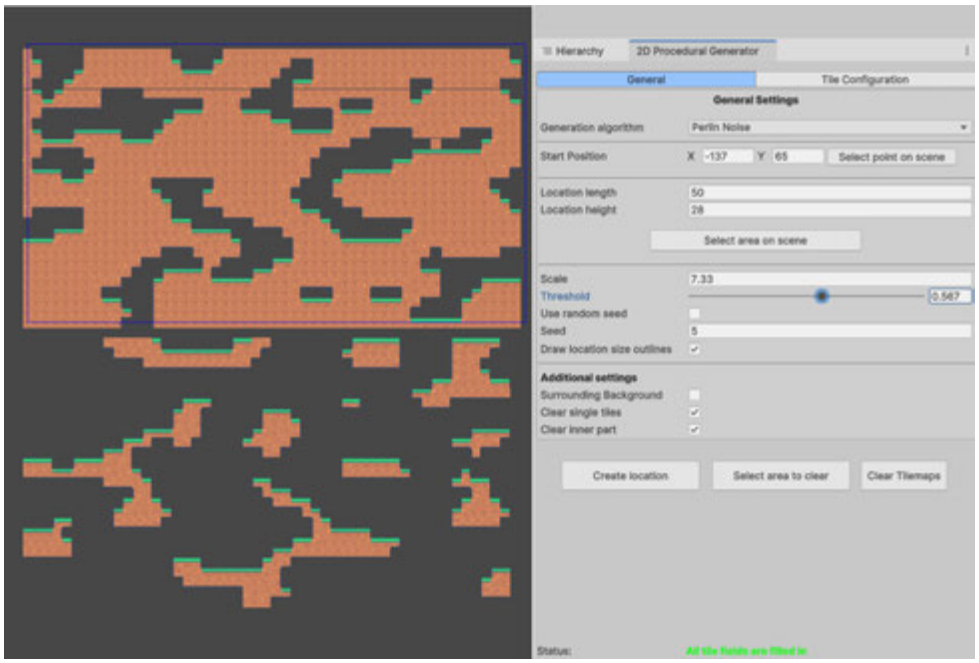


Рис. 4.9 Результат тестування алгоритму шуму Перліна на відповідність сіду та роботу порогового значення

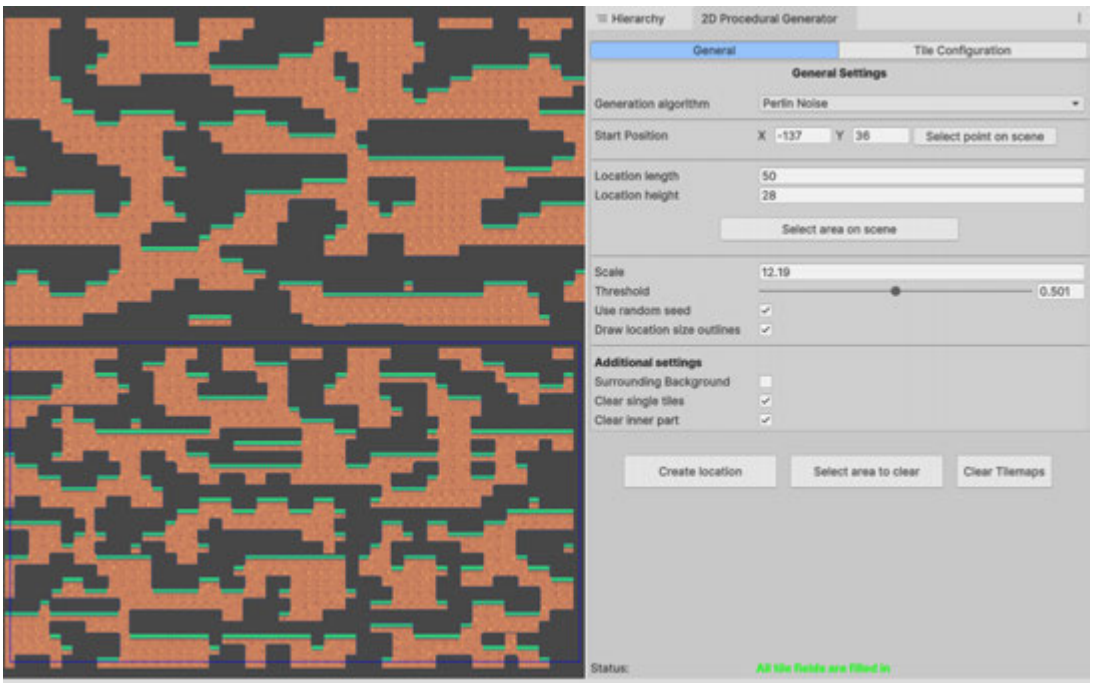


Рис. 4.10 Результат тестування алгоритму шуму Перліна при різному значенні seed та різному масштабі

Для алгоритму ступінчатих платформ найкращим способом проведення тестування є визначення коректності роботи значення seed та кількості платформ за рівнем наповненості рівня , оскільки інші параметри використовують випадкові значення та відстежити помітні зміни в них буде значно складніше. Представлений результат на рисунку 4.11 містить значну різницю між сформованими локаціями , які сформовані з трьома шарами для першої генерації та двадцятьма трьома для другої , які за ступенем заповненості підтверджують залежність від даного параметру. Щодо значення seed – локація трішки відрізняється за структурою через багатократне використання випадкових значень для генерації ,проте, може бути повністю однаково відтвореною за випадку ідентичності всіх параметрів , що продемонстровано на рисунку 4.12. ,де для обох локацій використані ідентичні параметри. Унікальність генерації при різних значеннях сіду продемонстрована на рисунку 4.13 , де ми можемо побачити абсолютно різний вигляд для двох сформованих локацій , попри всі однакові параметри , доводить успішне тестування для алгоритму .

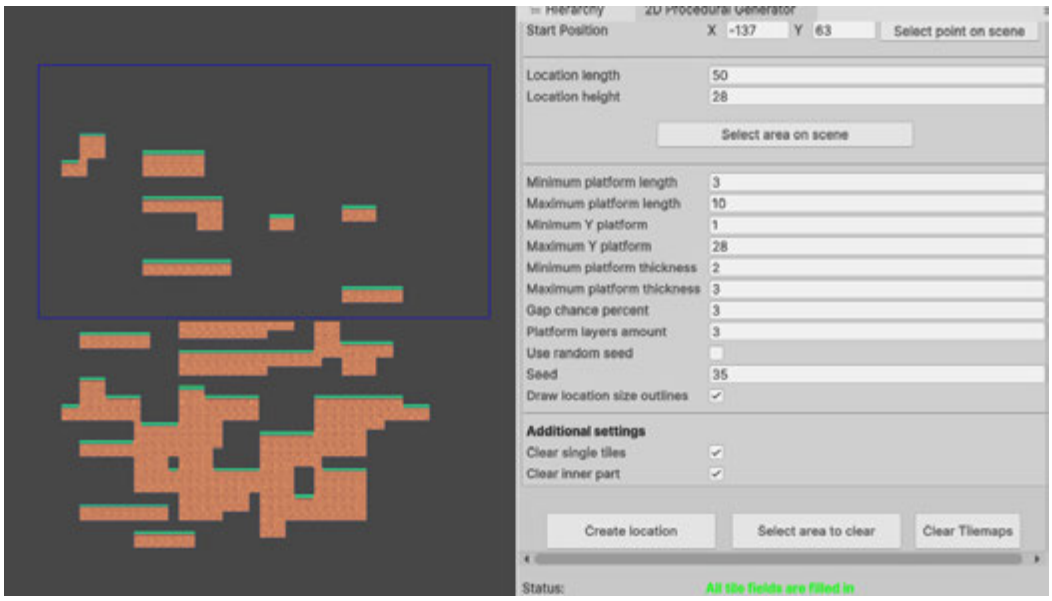


Рис. 4.11 Результат тестування на заповненість локації для алгоритму ступінчатих платформ.

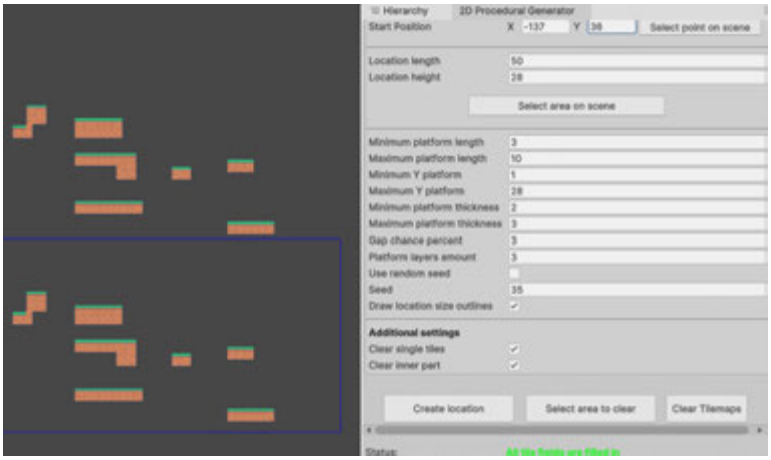


Рис.4.12 Результат тестування на повторюваність генерації від значення сіду

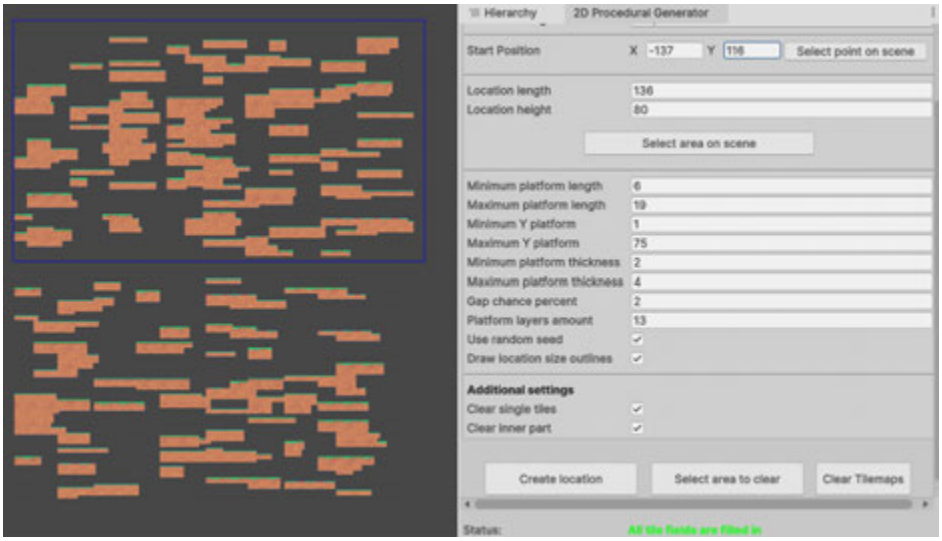


Рис. 4.13 Результат тестування унікальності при різних значеннях сіду для алгоритму ступінчатих платформ.

В процесі тестування алгоритмів було виявлено успішне проходження тесту всіма алгоритмами плагіну , що демонструє коректність його роботи та готовність до публікації та подальшої експлуатації розробниками та ігровими дизайнерами . Порядок дій для використання даного плагіну на прикладі створення рівня для ігрового демо представлений в додатку В.

4.2 Вимоги до апаратного та програмного забезпечення

Апаратні та програмні вимоги являють собою набір характеристик ,яким має відповідати обладнання та програми пристрою користувача для коректної роботи того чи іншого програмного продукту. Для плагіну процедурної генерації , що був розроблений в межах даної роботи була сформована Таблиця 4.1 ,що відображає мінімальні та рекомендовані вимоги для комфортної роботи з плагіном та середовищем розробки.

Таблиця 4.1

Вимоги до апаратного забезпечення

Компонент	Мінімальні вимоги	Рекомендовані вимоги
Процесор	Intel Core i3 / AMD Ryzen 3	Intel Core i5 або Ryzen 5 і вище
Оперативна пам’ять	8 ГБ	16 ГБ або більше
Графічний адаптер	Intel HD 5000 або NVIDIA GT 1030	NVIDIA GTX 1060 / AMD RX 580 або краще
Накопичувач	10 ГБ вільного місця	SSD з 20+ ГБ вільного місця для пришвидшеної роботи
Дисплей	Роздільна здатність 1366×768	Роздільна здатність 1920×1080 і вище

Програмні вимоги

- Unity Editor версії 2023.1 або вище (Unity 6)
- .NET SDK (вбудований у Unity, версія 4.x або .NET Standard 2.1)
- Microsoft Visual Studio / VS Code – середовище розробки з підтримкою C#
- Операційна система: Windows 10 / 11 або macOS 11+

Бібліотеки та компоненти Unity:

- UnityEngine.Tilemaps – для виводу згенерованих рівнів
- UnityEditor – для створення вікон інтерфейсу
- ScriptableObject – для збереження параметрів генерації та конфігурації плагіну
- UnityEngine.Random / System.Random – для роботи алгоритмів генерації
- System.Collections.Generic – для коректної обробки даних та роботи всіх компонентів плагіну
- UnityEngine.Mathf – для роботи алгоритму шуму Перліна

4.3 Склад інсталяційного пакету

4.3.1 Вибір типу файлу для експортування плагіну

Для експортування файлу з подальшими можливостями імпорту іншими користувачами був обраний формат .unitypackage , опираючись на його особливості та можливості . Файл даного типу у порівнянні з бібліотеками типу .dll та каталогом для менеджера пакетів Unity (UPM) має значну перевагу по багатьох критеріях , серед яких :

- Відкритий вихідний код , що забезпечує повну свободу редагування та дозволяє розробнику вносити зміни до реалізованого коду відповідно до вимог свого проєкту.
- Просто встановлюється , для цього достатньо просто “перетягнути” файл в структуру проєкту
- Висока сумісність з різними версіями рушія
- Підходить для публікації в asset store в перспективі розвитку.

4.3.2 Вміст інсталяційного пакету

Інсталяційний пакет , представлений у форматі .unitypackage містить в собі всі елементи ,визначені в діаграмі пакетів, згруповані в чітку ієрархію директорій для виокремлення окремих модулів плагіну(див. Рис.4.14) , а саме :

- Ядро процедурного генератора ,яке відповідає за логіку побудови рівня , та містить:
    - Алгоритми процедурної генерації включно з усіма основними та допоміжними класами логіки побудови локації
    - Систему візуалізації, що перетворює набір позицій в візуальну структуру рівня
    - Списки напрямків та типів плиток, які є допоміжними для системи візуалізації
- Сховище для конфігураційних файлів ,яке включає в себе :
    - Шаблони кімнат для алгоритму випадкової прогулянки
    - Картинку інструкції розміщення тайлів , що використовується для відповідного вікна
    - Конфігураційний файл з набором всіх параметрів генератора для збереження даних між сесіями
- Інтерфейс користувача , що представлений в форматі вікон та має:
    - Вікно процедурного генератора , що керує логікою всіх алгоритмів процедурної генерації та системи візуалізації у вигляді інтерфейсу користувача
    - Вікно інструкцій розміщення тайлів
- Демо -версія гри , що згідно визначеній структурі містить :
    - Набір ресурсів в форматі набору спрайтів для рівня
    - Приклади побудованих рівнів
    - Готові ігрові елементи
    - Анімації для ігрових об’єктів
    - Скрипти , що реалізують ігрову логіку

Для встановлення плагіну немає потреби завантажувати додаткові бібліотеки , чи використовувати серверні технології , оскільки проєкт містить лише компоненти рушія Unity та призначений для локального розгортання та подальшої експлуатації в ролі інструменту редактора. Для всіх видів конфігураційних файлів були створені шаблони , які дозволяють створювати екземпляри конфігурацій прямо в структурі проєкту через окремо винесений пункт “PCG” в контексті функції “Create” . Для використання демо немає необхідності встановлювати конфігурацію тайлів самостійно , вона вже вбудована в генератор ,у випадку використання іншого набору ресурсів , необхідно змінити встановлені тайли , використавши інші в форматі Tileset.

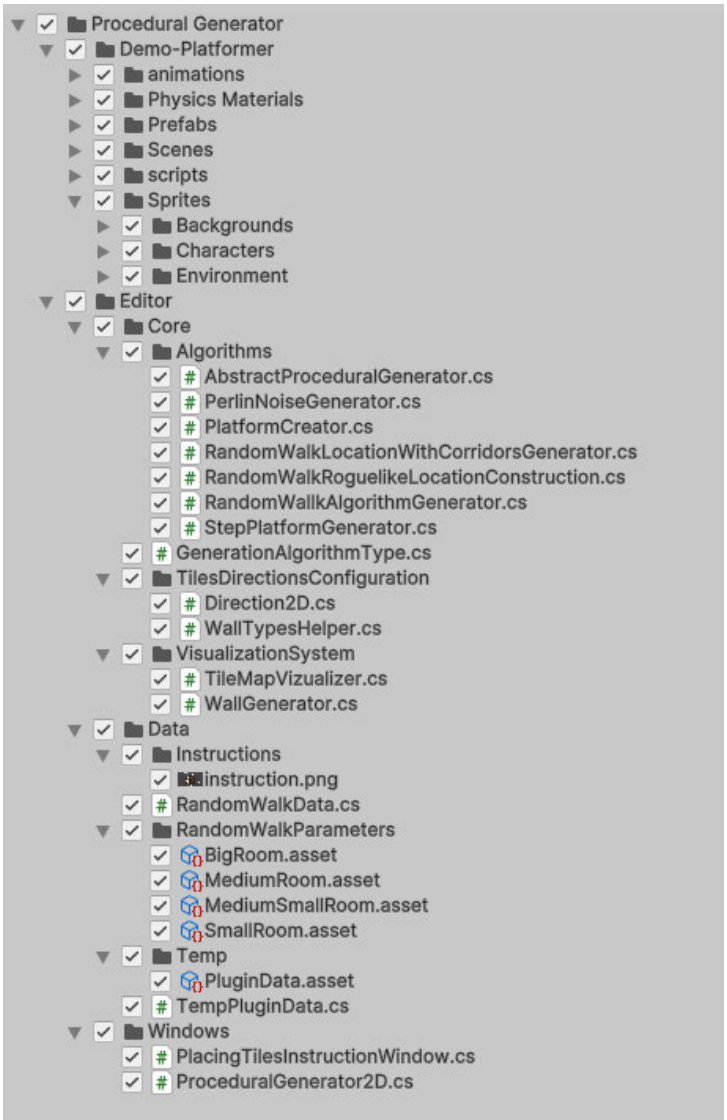


Рис.4.14 Структура інсталяційного пакету плагіну процедурної генерації

4.5 Перспективи розвитку

У сучасній індустрії ігрового розроблення спостерігається стійке зростання попиту на інструменти процедурної генерації, що дозволяють створювати динамічні, унікальні рівні при мінімальних витратах часу. Плагін, створений у рамках даного проєкту, вже реалізує базові алгоритми генерації для 2D-ігор, однак має значний потенціал для подальшого розвитку.

Одним з найбільш перспективних напрямків є інтеграція інструментів штучного інтелекту. Сучасні дослідження вказують на ефективність використання великих мовних моделей (LLM) і генеративного ШІ для автоматичної побудови рівнів на основі заданих умов, аналізу стилістики гравця чи адаптивного контенту. Unity також активно розвиває власні ШІ-інструменти, що у 2025 році увійшли в офіційну дорожню карту рушія Unity Roadmap, 2025[22]. Інтеграція таких засобів дозволить суттєво покращити користувацький досвід і автоматизувати рутинні процеси генерації.

Ще один напрямок — підтримка сучасних можливостей самого рушія Unity. Зокрема, нові функції, як-от Deferred+ рендеринг, Variable Rate Shading, оптимізовані анімації, а також DOTS і Burst Compiler, дають змогу значно покращити продуктивність і масштабованість рішень. Впровадження ECS-моделі дозволить працювати з великою кількістю об'єктів без значного навантаження на систему, що є критично важливим при генерації масивних сцен.

З точки зору алгоритмічних можливостей, потенціал полягає в реалізації складніших підходів до генерації. Наприклад, алгоритм Wave Function Collapse дозволяє створювати послідовні та узгоджені рівні з урахуванням локальних обмежень і зразків, що вже активно застосовується в проєктах типу roguelike.Додатково, включення адаптивної генерації, що змінюється в залежності від дій гравця, може стати важливою інновацією плагіна.

Також перспективним є розширення до тривимірного простору та підтримка VR/AR. Створення процедурних 3D-сцен, що підтримують взаємодію в реальному часі, значно розширює сферу застосування плагіна — від ігор до симуляцій, навчальних платформ і архітектурних візуалізацій. Unity уже

пропонує засоби, зокрема XR Interaction Toolkit, що дозволяють швидко інтегруватися з VR/AR пристроями та побудовувати взаємодію у просторі.

Загалом, розвиток плагіна може орієнтуватися як на розширення функціональності всередині редактора Unity, так і на зовнішню інтеграцію з AI, оптимізацію та багатоплатформену підтримку. Такий підхід забезпечить відповідність плагіна сучасним стандартам і дозволить використовувати його в найрізноманітніших типах проєктів — від 2D платформерів до VR-середовищ і мобільних застосунків.

Висновок до розділу 4

У даному розділі було розглянуто ключові аспекти впровадження, тестування та експлуатації розробленого плагіну процедурної генерації рівнів для середовища Unity. Проведене тестування продемонструвало коректність роботи всіх реалізованих алгоритмів, відповідність вихідних результатів очікуваній логіці генерації, а також стабільність системи при введенні граничних та некоректних значень.

На основі результатів перевірки було підтверджено повторюваність генерації за однакових вхідних параметрів (seed) та ефективну обробку помилок у випадках відхилень. Визначено залежність структури локацій від конфігураційних параметрів. Також було сформовано вимоги до апаратного та програмного забезпечення, необхідного для ефективної роботи плагіну.

Для впровадження інструменту для подальшої експлуатації було створено інсталяційний пакет у форматі .unitypackage, що включає всі необхідні компоненти: ядро генерації, систему візуалізації, параметри, інтерфейс та демонстраційний проєкт гри.

Також , опираючись на сучасні тенденції , були розглянуті перспективи розвитку плагіну , серед яких було виокремлено розширення переліку алгоритмів, інтеграцію з ШІ та введення інструментів для роботи з 3D, VR/AR, що не лише підвищить функціональність плагіну, а й забезпечить його конкурентоспроможність серед аналогічних рішень.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було створено плагін процедурної параметричної генерації 2D-локацій для рушія Unity. Система дозволяє генерувати ігрові рівні на основі трьох алгоритмів - випадкової прогулянки, шуму Перліна та ступінчатих платформ - з можливістю задавати параметри, зберігати конфігурації та інтегрувати результат безпосередньо в сцену редактора.

Результати тестування підтвердили стабільну та передбачувану роботу плагіну при різних сценаріях, включно з обробкою помилкових параметрів. Усі алгоритми працюють згідно очікуваних характеристик, забезпечуючи повторюваність генерації та варіативність структури рівнів. Реалізована архітектура плагіну є модульною, що спрощує розширення функціоналу у майбутньому.

Основною перевагою плагіну є централізований інтерфейс користувача у вигляді окремого редакторського вікна, що поєднує налаштування, запуск генерації та візуалізацію. На відміну від деяких аналогів, розроблене рішення не потребує складного налаштування або сторонніх залежностей, що робить його доступним для широкого кола користувачів - від інді-розробників до студентів.

З техніко-економічної точки зору система є ефективною: вона не потребує серверної частини чи складної інфраструктури, зберігає дані у конфігураційних файлах, що забезпечує простоту та швидкість роботи. Порівняно з аналогічними рішеннями, плагін вигідно вирізняється гнучкістю, інтуїтивністю інтерфейсу та простотою інтеграції.

Результати роботи можуть бути використані в розробці інді-ігор, створенні ігрових прототипів, а також у навчальних цілях. У подальшому доцільним є розширення плагіну новими алгоритмами генерації, підтримкою 3D-структур та публікацією на Asset Store для спрощення розповсюдження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Словник геймера: що таке "ААА" проєкти та як виникла така класифікація відеоігор [Електронний ресурс] // 24 Канал. – Режим доступу: https://games.24tv.ua/shho-take-aaa-igri-istoriya-viniknennya-populyarnoyi-klasifikatsiyi_n2574291 (дата звернення: 24.05.2025).
2. Інді-ігри: що це таке, походження та найкращі назви [Електронний ресурс] // El Output. – Режим доступу: https://uk.eloutput.com/гра/звітів/Інді-ігри/ (дата звернення: 24.05.2025).
3. Ігрові жанри [Електронний ресурс] // Онлайн-курси QATestLab. – Режим доступу: https://training.qatestlab.com/blog/technical-articles/games-genres/ (дата звернення: 24.05.2025).
4. Учасники проєктів Вікімедіа. Rogue (відеогра) [Електронний ресурс] // Вікіпедія. – Режим доступу: https://uk.wikipedia.org/wiki/Rogue_(відеогра) (дата звернення: 24.05.2025).
5. Учасники проєктів Вікімедіа. No Man's Sky [Електронний ресурс] // Вікіпедія. – Режим доступу: https://uk.wikipedia.org/wiki/No_Man's_Sky (дата звернення: 24.05.2025).
6. Учасники проєктів Вікімедіа. Terraria [Електронний ресурс] // Вікіпедія. – Режим доступу: https://uk.wikipedia.org/wiki/Terraria (дата звернення: 24.05.2025).
7. Maxym Z. Моделювання даних (data modelling) [Електронний ресурс] // Maxym Zosym. – Режим доступу: https://www.maxzosim.com/data-modelling/ (дата звернення: 23.05.2025).
8. Unreal Engine: як розвивався та змінював геймдев найпопулярніший ігровий рушій [Електронний ресурс] // ITC.ua. – Режим доступу: https://itc.ua/ua/blogs/unreal-engine-yak-vin-rozvyvavsya-i-zminyuvav-gejmdev/ (дата звернення: 24.05.2025).

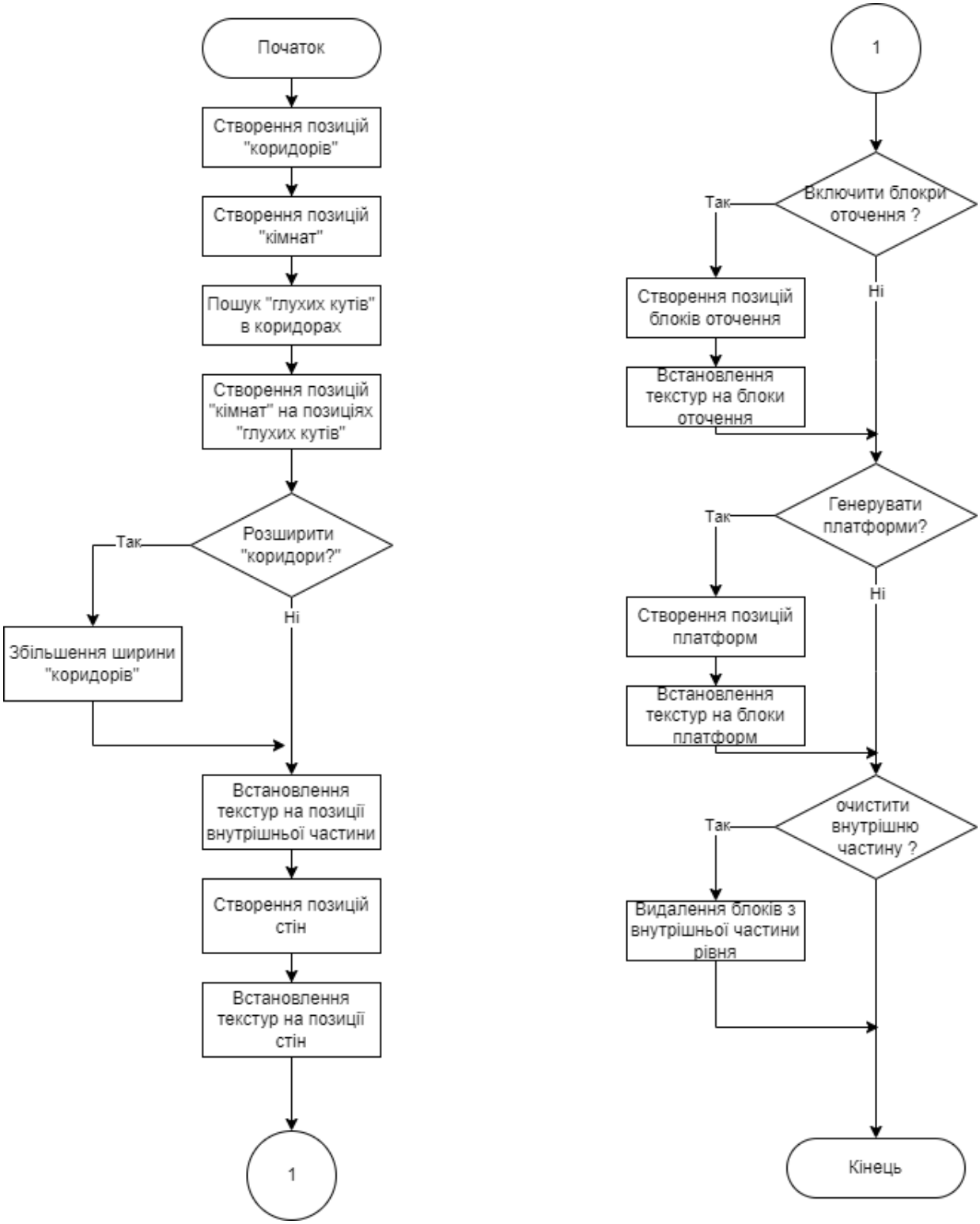
9. Що таке Unity і для чого використовується [Електронний ресурс] // Lemon School. – Режим доступу: https://lemon.school/blog/shho-take-unity (дата звернення: 24.05.2025).
10. Вступ до Godot [Електронний ресурс] // Godot Engine documentation. – Режим доступу: https://docs.godotengine.org/uk/4.x/getting_started/introduction/introduction_to_godot.html (дата звернення: 24.05.2025).
11. Sunny Valley Studio. Unity procedural dungeon generation [Електронний ресурс] // YouTube. – Режим доступу: https://www.youtube.com/playlist?list=PLcRSafycjWFenI87z7uZHFv6cUG2Tzu9v (дата звернення: 24.05.2025).
12. Peremot S. Математика в геймдеві: що таке шум Перліна та як його використовувати при створенні ігор [Електронний ресурс] // DOU. – Режим доступу: https://gamedev.dou.ua/articles/mathematics-gamedev-perlin-noise/ (дата звернення: 24.05.2025).
13. Wolf D. Building a simple procedurally-generated platformer [Електронний ресурс] // Game Developer. – Режим доступу: https://www.gamedeveloper.com/programming/building-a-simple-procedurally-generated-platformer (дата звернення: 24.05.2025).
14. Edgar Pro – Procedural Dungeon Generator [Електронний ресурс] // Unity Asset Store. – Режим доступу: https://assetstore.unity.com/packages/tools/utilities/edgar-pro-procedural-dungeon-generator-212735 (дата звернення: 24.05.2025).
15. Frigga – Procedural Dungeon Generator [Електронний ресурс] // Unity Asset Store. – Режим доступу: https://assetstore.unity.com/packages/tools/utilities/frigga-procedural-dungeon-generator-227242 (дата звернення: 24.05.2025).
16. L'hoest B. Semi-procedural Dungeon Generator [Електронний ресурс] // Fab. – Режим доступу: https://www.fab.com/listings/d794b679-f885-43bb-8a70-edbbfaa9917c (дата звернення: 24.05.2025).

17. Pixel Art Platformer – Village Props [Електронний ресурс] // Unity Asset Store. – Режим доступу: https://assetstore.unity.com/packages/2d/environments/pixel-art-platformer-village-props-166114 (дата звернення: 24.05.2025).
18. Overview of game development industry in Ukraine [Електронний ресурс] // Newxel. – Режим доступу: https://newxel.com/blog/game-development-team/?utm_source=chatgpt.com (дата звернення: 24.05.2025).
19. Takahashi D. Unity files for IPO, reveals $163 million loss for 2019 and 1.5 million monthly users [Електронний ресурс] // VentureBeat. – Режим доступу: https://venturebeat.com/business/unity-files-for-ipo-reveals-163-million-loss-for-2019-and-1-5-million-monthly-users/ (дата звернення: 24.05.2025).
20. Pixel Platformer [Електронний ресурс] // OpenGameArt.org. – Режим доступу: https://opengameart.org/content/pixel-platformer-0 (дата звернення: 24.05.2025).
21. Тестування програмного забезпечення: етапи та методи [Електронний ресурс] // FoxmindEd. – Режим доступу: https://foxminded.ua/testuvannia-prohramnoho-zabezpechennia/ (дата звернення: 24.05.2025).
22. What's next: Unity Engine 2025 Roadmap [Електронний ресурс] // Unity. – Режим доступу: https://unity.com/blog/unity-engine-2025-roadmap (дата звернення: 24.05.2025).
23. Macrosoft Store. Microsoft Visual Studio: що це, для чого це потрібно та як це працює [Електронний ресурс] / Macrosoft Store. – Режим доступу: https://macrosoft.store/uk/blog/post/29-для-чого-потрібна-microsoft-visual-studio. (Дата звернення: 24.05.2025)

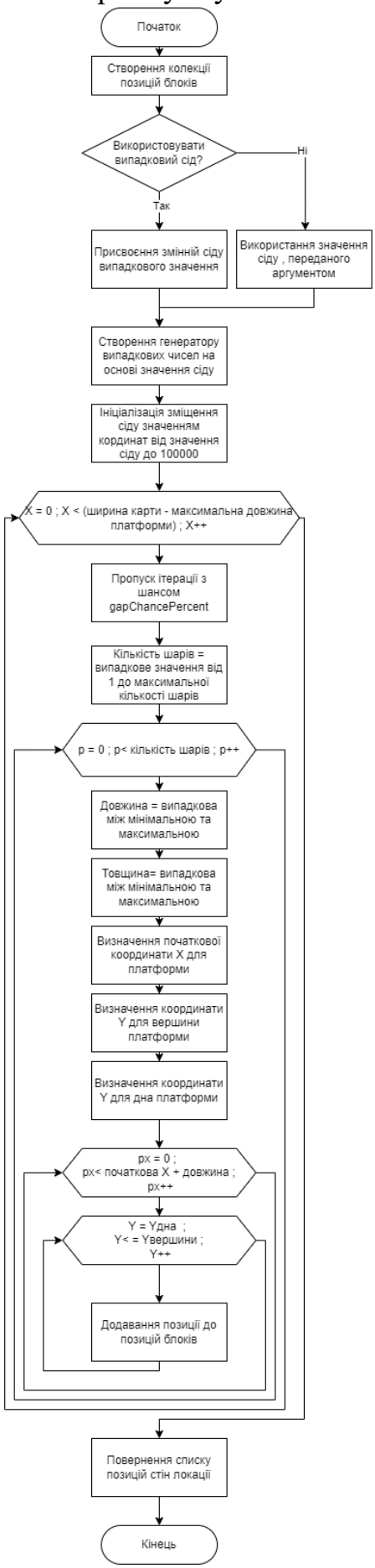
ДОДАТКИ

Додаток А

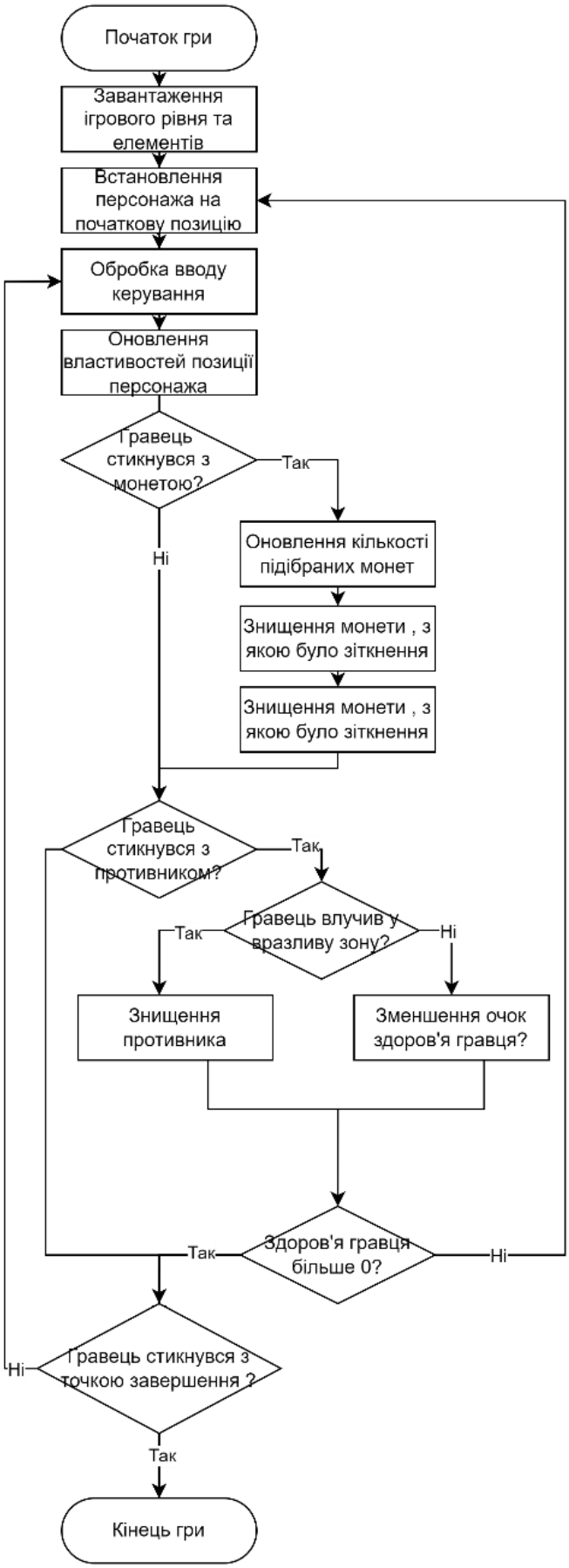
Блок схема алгоритму генерації локації за допомогою алгоритму випадкової прогулянки



Блок схема алгоритму ступінчатих платформ



Блок схема циклу демо версії гри



Додаток Б

Програмний код алгоритмів процедурної генерації

Код класу алгоритму випадкової прогулянки

```
public class RandomWallkAlgorithmGenerator : AbstractProceduralGenerator
{
   protected RandomWalkData _randomWalkParameters;

    public void Setup(RandomWalkData RandomWalkParameters )
    {
        _randomWalkParameters = RandomWalkParameters;
    }

    protected override void RunProceduralGeneration()
    {
        HashSet<Vector2Int> floorPositions = RunRandomWalk(_randomWalkParameters,
_startPosition);
        _tileMapVizualizer.PaintInnerTiles(floorPositions);
        WallGenerator.CreateWalls(floorPositions, _tileMapVizualizer);
    }

    protected HashSet<Vector2Int> RunRandomWalk(RandomWalkData parameters , Vector2Int
position)
    {
        var currentPosition = position;
        HashSet<Vector2Int> floorPositions = new HashSet<Vector2Int>();
        for (int i = 0; i < _randomWalkParameters.iterations; i++)
        {
            var path =
RandomWalkRoguelikeLocationConstruction.RandomWalk(currentPosition,
_randomWalkParameters.walkLength);
            floorPositions.UnionWith(path);
            if (_randomWalkParameters.startRandomlyEachIteration)
                currentPosition = floorPositions.ElementAt(Random.Range(0,
floorPositions.Count));
        }
        return floorPositions;
    }
}
```

Код класу формування структури локації для створення локації методом випадкової прогулянки

```
public static class RandomWalkRoguelikeLocationConstruction
{
    public static HashSet<Vector2Int> RandomWalk(Vector2Int startPosition, int
walkLength)
    {
        HashSet<Vector2Int> path = new HashSet<Vector2Int>();

        path.Add(startPosition);
        var previousposition = startPosition;

        for (int i = 0; i < walkLength; i++)
        {
            var newPosition = previousposition +
Direction2D.GetRandomCardinalDirection();
            path.Add(newPosition);
            previousposition = newPosition;
        }
```

```
        return path;
    }

    public static List<Vector2Int> RandomWalkCorridor(Vector2Int startPosition, int
corridorlength)
    {
        List<Vector2Int> corridor = new List<Vector2Int>();
        var direction = Direction2D.GetRandomCardinalDirection();
        var currentPosition = startPosition;
        corridor.Add(currentPosition);

        for (int i = 0; i < corridorlength; i++)
        {
            currentPosition += direction;
            corridor.Add(currentPosition);
        }
        return corridor;
    }

}
```

Код класу створення локації алгоритмом випадкової прогулянки

```
public class RandomWalkLocationWithCorridorsGenerator : RandomWallkAlgorithmGenerator
{
     private int _corridorLength;
     private int _corridorCount;
     private bool _expandCorridor ;
     private int _corridorExpansion ;
     private bool _includeBackground ;
     private bool _clearInnerPart;
     private bool _generatePlatforms;
     private int _platformSelectionRegionWidth;
     private int _platformSelectionRegionHeight;
     private float _platformSpawnChance;
     private int _backgroundThickness;
    [Range(0.1f , 1)] private float _roomPercent;

    public void Initialize(int corridorLength, int corridorCount, bool expandCorridor,
int corridorExpansion, float roomPercent,
        RandomWalkData RandomWalkParameters, TileMapVizualizer tileMapVizualizer,
        bool IncludeBackground,int BackgroundThickness, bool ClearInnerPart,bool
GeneratePlatforms,int regionWidth , int regionHeight , float SpawnChance ,Vector2Int
StartPosition)
    {
        _corridorLength=corridorLength;
        _corridorCount=corridorCount;
        _expandCorridor=expandCorridor;
        _corridorExpansion=corridorExpansion;
        _roomPercent=roomPercent;
        _randomWalkParameters = RandomWalkParameters;
        _tileMapVizualizer = tileMapVizualizer;
        _includeBackground = IncludeBackground;
        _backgroundThickness = BackgroundThickness;
        _clearInnerPart = ClearInnerPart;
        _generatePlatforms = GeneratePlatforms;
        _platformSelectionRegionWidth = regionWidth;
        _platformSelectionRegionHeight = regionHeight;
        _platformSpawnChance =SpawnChance;
        _startPosition = StartPosition;
    }

    protected override void RunProceduralGeneration()
    {
        CorridorFirstGeneration();
    }
```

```
    private void CorridorFirstGeneration()
    {
        HashSet<Vector2Int> floorPositions = new HashSet<Vector2Int>();
        HashSet<Vector2Int> potentialRoomPositions = new HashSet<Vector2Int>();
        List<List <Vector2Int>> corridors = CreateCorridors(floorPositions,
potentialRoomPositions);
        HashSet<Vector2Int> roomPositions = CreateRooms(potentialRoomPositions);
        List<Vector2Int> deadEnds = FindAllDeadEnds(floorPositions);
        CreateRoomsAtDeadEnd(deadEnds , roomPositions);
        floorPositions.UnionWith(roomPositions);
        if (_expandCorridor)
        {
            for (int i = 0; i < corridors.Count; i++)
            {
                corridors[i] = IncreaseCorridors(corridors[i] , _corridorExpansion);
                floorPositions.UnionWith(corridors[i]);
            }
        }
        _tileMapVizualizer.PaintInnerTiles(floorPositions);
        WallGenerator.CreateWalls(floorPositions, _tileMapVizualizer);
        if (_includeBackground)
        {
            _tileMapVizualizer.PaintBackgroundTiles(new
HashSet<Vector2Int>(floorPositions.Union(roomPositions)),_startPosition,_backgroundThi
ckness);
        }
        if (_generatePlatforms)
        {
            PlatformCreator platformGenerator =
ScriptableObject.CreateInstance<PlatformCreator>();
            platformGenerator.Initialize(_tileMapVizualizer,
_platformSelectionRegionWidth, _platformSelectionRegionHeight, _platformSpawnChance);
            platformGenerator.PlacePlatforms();
        }
        if (_clearInnerPart)
        {
            _tileMapVizualizer.ClearInnerPart(floorPositions);
        }
    }
    private void CreateRoomsAtDeadEnd(List<Vector2Int> deadEnds, HashSet<Vector2Int>
roomFloors)
    {
        foreach (var position in deadEnds)
        {
            if (!roomFloors.Contains(position))
            {
                var room = RunRandomWalk(_randomWalkParameters , position);
                roomFloors.UnionWith(room);
            }
        }
    }
    private List<Vector2Int> FindAllDeadEnds(HashSet<Vector2Int> floorPositions)
    {
        List<Vector2Int> deadEnds = new List<Vector2Int>();
        foreach (var position in floorPositions)
        {
            int neighboursCount = 0;
            foreach (var direction in Direction2D.cardinalDirectionList)
            {
                if (floorPositions.Contains(position + direction))
                    neighboursCount++;
            }
            if (neighboursCount == 1)
                deadEnds.Add(position);
```

```
        }
        return deadEnds;
    }
    private HashSet<Vector2Int> CreateRooms(HashSet<Vector2Int> potetialRoomPositions)
    {
        HashSet<Vector2Int> roomPositions = new HashSet<Vector2Int>();
        int roomToCreateCount = Mathf.RoundToInt( potetialRoomPositions.Count *
_roomPercent);

        List<Vector2Int> roomsToCreate = potetialRoomPositions.OrderBy(x =>
Guid.NewGuid()).Take(roomToCreateCount).ToList();

        foreach (var roomPosition in roomsToCreate)
        {
            var roomFloor = RunRandomWalk(_randomWalkParameters, roomPosition);
            roomPositions.UnionWith(roomFloor);
        }
        return roomPositions;
    }
    private List<List<Vector2Int>> CreateCorridors(HashSet<Vector2Int> floorPositions,
HashSet<Vector2Int> potentialRoomPositions)
    {
        var currentPosition = _startPosition;
        potentialRoomPositions.Add(currentPosition);
        List<List<Vector2Int>> corridors = new List<List<Vector2Int>>();

        for (int i = 0; i < _corridorCount; i++)
        {
            var corridor =
RandomWalkRoguelikeLocationConstruction.RandomWalkCorridor(currentPosition,
_corridorLength);
            corridors.Add(corridor);
            currentPosition = corridor[corridor.Count -1];
            potentialRoomPositions.Add(currentPosition);
            floorPositions.UnionWith(corridor);
        }
        return corridors;
    }
    private List<Vector2Int> IncreaseCorridors(List<Vector2Int> corridor ,int width)
    {
        List<Vector2Int> newCorridor = new List<Vector2Int>();
        int offset = ValidateWidth(width)/2;
        for (int i = 1; i < corridor.Count; i++){
            for (int x = -offset; x < offset+1; x++){
                for (int y = -offset; y < offset+1; y++){
                    newCorridor.Add(corridor[i-1] + new Vector2Int(x, y));
                }
            }
        }
        return newCorridor;
    }

}
```

Код класу генератора алгоритму шуму Перліна

```
public class PerlinNoiseGenerator : AbstractProceduralGenerator
{
    private Vector2Int _mapSize ;
    private float _scale ;
    private float _threshold;
    private bool _useRandomSeed;
    private int _seed;
    private bool _clearInnerPart;
    private bool _includeBackground;
    private int _backgroundThickness;
    protected override void RunProceduralGeneration()
    {
        if (_mapSize.x >0  && _mapSize.y >0)
        {
            HashSet<Vector2Int> emptySpacePositions = GenerateMapWithPerlin();
            _tileMapVizualizer.PaintInnerTiles(emptySpacePositions);
            WallGenerator.CreateWalls(emptySpacePositions, _tileMapVizualizer);
            _tileMapVizualizer.FillEmptyAreasInside(emptySpacePositions);
            if (_includeBackground){
                _tileMapVizualizer.PaintBackgroundTiles(emptySpacePositions,
_startPosition, _backgroundThickness);
            }
            else{
                _tileMapVizualizer.RemoveBorderTiles(emptySpacePositions);
            }
            if (_clearInnerPart){
                _tileMapVizualizer.ClearInnerPart(emptySpacePositions);
            }
        }
        else{Debug.Log("Error! Area height and length cannot be less than 0");}}
    public void Initialize(Vector2Int MapSize, float Scale, float Threshold, bool
UseRandomSeed, int Seed,
        TileMapVizualizer vizualizer ,Vector2Int StartPosition , bool clearInnerPart ,
bool includeBackground , int BackgroundThickness)
    {   _mapSize =MapSize;
        _scale = Scale;
        _threshold = Threshold;
        _useRandomSeed = UseRandomSeed;
        _seed = Seed;
        _tileMapVizualizer = vizualizer;
        _startPosition = StartPosition;
        _clearInnerPart = clearInnerPart;
        _includeBackground = includeBackground;
        _backgroundThickness =BackgroundThickness; }

    private HashSet<Vector2Int> GenerateMapWithPerlin(){
        HashSet<Vector2Int> floorPositions = new HashSet<Vector2Int>();
        int usedSeed = _useRandomSeed ? Random.Range(0, int.MaxValue) : _seed;
        System.Random prng = new System.Random(usedSeed);
        Vector2 noiseOffset = new Vector2(prng.Next(0, 100000), prng.Next(0, 100000));
        int halfWidth = _mapSize.x / 2;
        int halfHeight = _mapSize.y / 2;
        for (int x = 0; x < _mapSize.x; x++){
            for (int y = 0; y < _mapSize.y; y++){
                float xCoord = (float)x / _mapSize.x * _scale + noiseOffset.x;
                float yCoord = (float)y / _mapSize.y * _scale + noiseOffset.y;
                float noise = Mathf.PerlinNoise(xCoord, yCoord);
                if (noise >= _threshold){
                    Vector2Int worldPos = new Vector2Int(_startPosition.x + x -
halfWidth, _startPosition.y + y - halfHeight);
                    floorPositions.Add(worldPos);}}}
        return floorPositions;}}
```

Код класу генератора алгоритму ступінчатих платформ

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

public class StepPlatformGenerator : AbstractProceduralGenerator
{
    private int _mapWidth ;
    private int _mapHeight ;
    private int _minPlatformLength;
    private int _maxPlatformLength;
    private int _minY;
    private int _maxY;
    private int _minPlatformThickness;
    private int _maxPlatformThickness;
    private int _gapChancePercent;
    private int _maxPlatformsPerColumn;

    private bool _clearInnerPart;
    private bool _useRandomSeed;
    private int _seed;

    public void Initialize(int MapWidth, int MapHeight, int MinPlatformLength ,int
MaxPlatformLength ,int MinY ,int MaxY
        ,int MinPlatformThickness ,int MaxPlatformThickness, int GapChancePercent ,int
MaxPlatformsPerColumn ,bool UseRandomSeed , int Seed,
        TileMapVizualizer vizualizer , bool clearInnerPart ,Vector2Int CenterPosition)
    {
        _mapWidth = MapWidth;
        _mapHeight = MapHeight;
        _minPlatformLength = MinPlatformLength;
        _maxPlatformLength = MaxPlatformLength;
        _minY = MinY;
        _maxY = MaxY;
        _minPlatformThickness =MinPlatformThickness;
        _maxPlatformThickness = MaxPlatformThickness;
        _gapChancePercent = GapChancePercent;
        _maxPlatformsPerColumn = MaxPlatformsPerColumn;

        _clearInnerPart = clearInnerPart;
        _useRandomSeed = UseRandomSeed;

        _seed = Seed;
        _tileMapVizualizer = vizualizer;
        _startPosition = new Vector2Int(
            CenterPosition.x - _mapWidth / 2,
            CenterPosition.y - _mapHeight / 2);
    }
    protected override void RunProceduralGeneration()
    {
        if (_mapHeight>0 && _mapHeight>0 && _minPlatformLength >0 &&
_maxPlatformLength >0
            && _minY >0 && _maxY >0 && _minPlatformLength >0 && _maxPlatformLength >0
&&
            _gapChancePercent >0)
        {
            var platformBlocks = GenerateFullPlatformBlocks();


            HashSet<Vector2Int> fullMapArea = new HashSet<Vector2Int>();
            for (int x = _startPosition.x + 1; x < _startPosition.x + _mapWidth - 1;
x++)
            {
                for (int y = _startPosition.y + 1; y < _startPosition.y + _mapHeight -
1; y++)
```

```
                {
                    fullMapArea.Add(new Vector2Int(x, y));
                }
            }

            var emptySpace = new HashSet<Vector2Int>(fullMapArea);
            emptySpace.ExceptWith(platformBlocks);

            _tileMapVizualizer.PaintInnerTiles(emptySpace);
            WallGenerator.CreateWalls(emptySpace, _tileMapVizualizer);
            _tileMapVizualizer.FillEmptyAreasInside(emptySpace);

            if (_clearInnerPart)
                _tileMapVizualizer.ClearInnerPart(emptySpace);
            _tileMapVizualizer.RemoveBorderTiles(emptySpace);
        }
        else
        {
            Debug.Log("Error!All parameters must be greater than 0");
        }
    }

    private HashSet<Vector2Int> GenerateFullPlatformBlocks()
    {
        HashSet<Vector2Int> blocks = new HashSet<Vector2Int>();
        System.Random rng = _useRandomSeed ? new System.Random() : new
System.Random(_seed);

        for (int x = 1; x < _mapWidth - _maxPlatformLength; x += rng.Next(2,
_maxPlatformLength))
        {
            if (rng.Next(100) < _gapChancePercent)
                continue;
            int platformsPerColumn = rng.Next(1, _maxPlatformsPerColumn + 1);

            for (int p = 0; p < platformsPerColumn; p++)
            {
                int length = rng.Next(_minPlatformLength, _maxPlatformLength + 1);
                int thickness = rng.Next(_minPlatformThickness, _maxPlatformThickness
+ 1);

                int startX = Mathf.Clamp(x, 1, _mapWidth - length - 1);
                int yTop = rng.Next(_minY + thickness, _maxY);
                int yBottom = Mathf.Clamp(yTop - thickness + 1, _minY, yTop);

                for (int px = startX; px < startX + length; px++)
                {
                    for (int y = yBottom; y <= yTop; y++)
                    {
                        blocks.Add(new Vector2Int(px + _startPosition.x, y +
_startPosition.y));
                    }
                }
            }
        }
        return blocks;
    }

}
```

Код класу візуалізатора алгоритмів

```
public class TileMapVizualizer : ScriptableObject
{
    private Tilemap _innerTilemap;
    private Tilemap _wallTileMap;
    private bool _clearSingleTiles;
    private TileBase _floorTile;
    private TileBase _wallTop;
    private TileBase _wallSideRight;
    private TileBase _wallSideLeft;
    private TileBase _wallBottom;
    private TileBase _wallFull;
    private TileBase _wallInnerCornerDownLeft;
    private TileBase _wallInnerCornerDownRight;
    private TileBase _wallInnerCornerRightUp;
    private TileBase _wallInnerCornerLeftUp;
    private TileBase _wallIDiagonalCornerDownLeft;
    private TileBase _wallIDiagonalCornerDownRight;
    private TileBase _wallIDiagonalCornerUpRight;
    private TileBase _wallIDiagonalCornerUpLeft;
    private TileBase _bridgeWall;
    private TileBase _flyingWall;
    private TileBase _hangingWall;
    private TileBase _protrudingWallRight;
    private TileBase _protrudingWallLeft;
    private TileBase _straightVerticalWall;
    private TileBase _groundTopWall;
    private TileBase _backGroundTile;
    private List<Vector3Int> _singleTilesPositions;
    public void Initialize(Tilemap innerTilemap, Tilemap wallTileMap,bool
clearSingleTiles,
        TileBase floorTile, TileBase wallTop, TileBase wallSideRight, TileBase
wallSideLeft, TileBase wallBottom, TileBase wallFull,
        TileBase wallInnerCornerDownLeft, TileBase wallInnerCornerDownRight, TileBase
wallInnerCornerRightUp, TileBase wallInnerCornerLeftUp,
        TileBase wallIDiagonalCornerDownLeft, TileBase wallIDiagonalCornerDownRight,
TileBase wallIDiagonalCornerUpRight, TileBase wallIDiagonalCornerUpLeft,
        TileBase BridgeWall ,TileBase flyingWall, TileBase hangingWall, TileBase
protrudingWallRight, TileBase protrudingWallLeft, TileBase straightVerticalWall,
TileBase groundTopWall, TileBase backGroundTile)
    {
        _innerTilemap =innerTilemap;
        _wallTileMap = wallTileMap;
        _clearSingleTiles = clearSingleTiles;
        _floorTile = floorTile;
        _wallTop = wallTop;
        _wallSideRight=wallSideRight;
        _wallSideLeft=wallSideLeft;
        _wallBottom=wallBottom;
        _wallFull=wallFull;
        _wallInnerCornerDownLeft=wallInnerCornerDownLeft;
        _wallInnerCornerDownRight=wallInnerCornerDownRight;
        _wallInnerCornerRightUp=wallInnerCornerRightUp;
        _wallInnerCornerLeftUp=wallInnerCornerLeftUp;
        _wallIDiagonalCornerDownLeft=wallIDiagonalCornerDownLeft;
        _wallIDiagonalCornerDownRight=wallIDiagonalCornerDownRight;
        _wallIDiagonalCornerUpRight=wallIDiagonalCornerUpRight;
        _wallIDiagonalCornerUpLeft=wallIDiagonalCornerUpLeft;
        _bridgeWall = BridgeWall;
        _flyingWall=flyingWall;
        _hangingWall=hangingWall;
        _protrudingWallRight=protrudingWallRight;
        _protrudingWallLeft=protrudingWallLeft;
        _straightVerticalWall=straightVerticalWall;
```

```
            _groundTopWall=groundTopWall;
            _backGroundTile=backGroundTile;
            _singleTilesPositions = new List<Vector3Int>();
        }
        public Tilemap GetInnerTileMap()
        {
            return _innerTilemap;
        }
        public Tilemap GetWallTileMap()
        {
            return _wallTileMap;
        }
        public void PaintInnerTiles(IEnumerable<Vector2Int> EmptySpacePositions)
        {
            PaintTiles(EmptySpacePositions, _innerTilemap, _floorTile);
        }
        private void PaintTiles(IEnumerable<Vector2Int> positions, Tilemap tilemap,
    TileBase tile)
        {
            foreach (var position in positions)
            {
                PaintSingleTile(tilemap, tile, position);
            }
        }
        private void PaintSingleTile(Tilemap tilemap, TileBase tile, Vector2Int position)
        {
            var tilePosition = tilemap.WorldToCell((Vector3Int)position);
            tilemap.SetTile(tilePosition, tile);
        }
        internal void PaintSingleWall(Vector2Int position, string binaryType)
        {
            int typeAsInt = Convert.ToInt32(binaryType, 2);
            TileBase tile = null;
            if (WallTypesHelper.wallTop.Contains(typeAsInt))
            {
                tile=_wallTop;
            }
            else if (WallTypesHelper.wallSideRight.Contains(typeAsInt))
            {
                tile=_wallSideRight;
            }
            else if (WallTypesHelper.wallSideLeft.Contains(typeAsInt))
            {
                tile=_wallSideLeft;
            }
            else if (WallTypesHelper.wallBottom.Contains(typeAsInt))
            {
                tile=_wallBottom;
            }
            else if (WallTypesHelper.wallFull.Contains(typeAsInt))
            {
                tile=_wallFull;
            }
            if (tile!=null)
                PaintSingleTile(_wallTileMap, tile, position);
        }
        internal void PaintSingleCornerWall(Vector2Int position, string binaryType)
        {
            int typeAsInt = Convert.ToInt32(binaryType, 2);
            TileBase tile = null;
            if (_wallTileMap.GetSprite((Vector3Int)position)==null)
            {
                if (WallTypesHelper.wallSideRight8Directions.Contains(typeAsInt))
                {
                    tile = _wallSideRight;
```

```
}
else if (WallTypesHelper.wallSideLeft8Directions.Contains(typeAsInt))
{
    tile = _wallSideLeft;
}
else if (WallTypesHelper.wallBottom8Directions.Contains(typeAsInt))
{
    tile = _wallBottom;
}
else if (WallTypesHelper.wallTop8Directions.Contains(typeAsInt))
{
    tile = _wallTop;
}
else if (WallTypesHelper.wallInnerCornerDownLeft.Contains(typeAsInt))
{
    tile = _wallInnerCornerDownLeft;
}
else if (WallTypesHelper.wallInnerCornerDownRight.Contains(typeAsInt))
{
    tile = _wallInnerCornerDownRight;
}
else if (WallTypesHelper.wallDiagonalCornerDownLeft.Contains(typeAsInt))
{
    tile = _wallIDiagonalCornerDownLeft;
}
else if (WallTypesHelper.wallDiagonalCornerDownRight.Contains(typeAsInt))
{
    tile = _wallIDiagonalCornerDownRight;
}
else if (WallTypesHelper.wallDiagonalCornerUpRight.Contains(typeAsInt))
{
    tile =_wallIDiagonalCornerUpRight;
}
else if (WallTypesHelper.wallDiagonalCornerUpLeft.Contains(typeAsInt))
{
    tile =_wallIDiagonalCornerUpLeft;
}
else if (WallTypesHelper.wallFull8Directions.Contains(typeAsInt))
{
    tile =_wallFull;
}
else if (WallTypesHelper.BridgeWall.Contains(typeAsInt))
{
    tile = _bridgeWall;
}
else if (WallTypesHelper.flyingWallsDirections.Contains(typeAsInt))
{
    if (_clearSingleTiles)
    {
        tile = null;
        _singleTilesPositions.Add((Vector3Int)position);
        PaintSingleTile(_innerTilemap, _floorTile, position);
    }
    else
    {
        tile =_flyingWall;
    }
}
else if (WallTypesHelper.wallInnerCornerUpRight.Contains(typeAsInt))
{
    tile =_wallInnerCornerRightUp;
}
else if (WallTypesHelper.wallInnerCornerUpLeft.Contains(typeAsInt))
{
    tile =_wallInnerCornerLeftUp;
```

```
            }
            else if (WallTypesHelper.straightVerticalWall.Contains(typeAsInt))
            {
                tile=_straightVerticalWall;
            }
            else if (WallTypesHelper.hangingWall.Contains(typeAsInt))
            {
                tile =_hangingWall;
            }
            else if (WallTypesHelper.protrudingWallLeft.Contains(typeAsInt))
            {
                tile =_protrudingWallLeft;
            }
            else if (WallTypesHelper.protrudingWallRight.Contains(typeAsInt))
            {
                tile =_protrudingWallRight;
            }
            else if (WallTypesHelper.groundTopWall.Contains(typeAsInt))
            {
                tile =_groundTopWall;
            }
            if (tile!=null)
            {
                PaintSingleTile(_wallTileMap, tile, position);
            }
        }
    }
    internal void FillPlatformInside(HashSet<Vector2Int> areaPositions)
    {
        foreach (var position in areaPositions)
        {
            if (!_wallTileMap.HasTile((Vector3Int)position))
                PaintSingleTile(_wallTileMap, _wallFull, position);
        }
    }
    internal void FillEmptyAreasInside(HashSet<Vector2Int> areapos)
    {
        int minX = int.MaxValue, maxX = int.MinValue;
        int minY = int.MaxValue, maxY = int.MinValue;
        foreach (var pos in areapos)
        {
            if (pos.x < minX) minX = pos.x;
            if (pos.x > maxX) maxX = pos.x;
            if (pos.y < minY) minY = pos.y;
            if (pos.y > maxY) maxY = pos.y;
        }
        for (int x = minX; x <= maxX; x++)
        {
            for (int y = minY; y <= maxY; y++)
            {
                Vector3Int tilePos = new Vector3Int(x, y, 0);
                if (!_wallTileMap.HasTile(tilePos) && !_innerTilemap.HasTile(tilePos))
                {
                    _wallTileMap.SetTile(tilePos, _wallFull);
                }
            }
        }
    }
    internal void ClearInnerPart(HashSet<Vector2Int> positions)
    {
        List<Vector2Int> EmptySpacePositions = positions.ToList<Vector2Int>();

        if (_clearSingleTiles)
        {
            foreach (var pos in _singleTilesPositions)
```

```
            {
                EmptySpacePositions.Add((Vector2Int)pos);
            }
        }
        foreach (var position in EmptySpacePositions)
        {

            PaintSingleTile(_innerTilemap,null, position);
        }
    }
    internal void PaintBackgroundTiles(HashSet<Vector2Int> mapPositions, Vector2Int
startPosition, int borderThickness)
    {
        if (mapPositions == null || mapPositions.Count == 0)
            return;
        int minX = int.MaxValue, maxX = int.MinValue;
        int minY = int.MaxValue, maxY = int.MinValue;
        foreach (var pos in mapPositions)
        {
            if (pos.x < minX) minX = pos.x;
            if (pos.x > maxX) maxX = pos.x;
            if (pos.y < minY) minY = pos.y;
            if (pos.y > maxY) maxY = pos.y;
        }
        minX -= borderThickness;
        maxX += borderThickness;
        minY -= borderThickness;
        maxY += borderThickness;

        for (int x = minX; x <= maxX; x++)
        {
            for (int y = minY; y <= maxY; y++)
            {
                Vector2Int position = new Vector2Int(x, y);
                if (!mapPositions.Contains(position) &&
                    _innerTilemap.GetTile((Vector3Int)position) == null &&
                    _wallTileMap.GetTile((Vector3Int)position) == null)
                {
                    PaintSingleTile(_wallTileMap, _backGroundTile, position);
                }
            }
        }
    }
    public void RemoveBorderTiles( HashSet<Vector2Int> innerArea)
    {
        if (innerArea == null || innerArea.Count == 0) return;
        int minX = int.MaxValue, maxX = int.MinValue;
        int minY = int.MaxValue, maxY = int.MinValue;

        foreach (var pos in innerArea)
        {
            if (pos.x < minX) minX = pos.x;
            if (pos.x > maxX) maxX = pos.x;
            if (pos.y < minY) minY = pos.y;
            if (pos.y > maxY) maxY = pos.y;
        }
        minX -= 1;
        maxX += 1;
        minY -= 1;
        maxY += 1;
        for (int x = minX; x <= maxX; x++)
        {

            PaintSingleTile(_wallTileMap,null, new Vector2Int(x, minY));
            PaintSingleTile(_wallTileMap,null, new Vector2Int(x, maxY));
```

```
        }
        for (int y = minY + 1; y < maxY; y++)
        {
            PaintSingleTile(_wallTileMap, null, new Vector2Int( minX, y));
            PaintSingleTile(_wallTileMap, null, new Vector2Int( maxX, y));
        }
    }
}
```

Код допоміжного класу з визначенням напрямків

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

public static class Direction2D
{
    public static List<Vector2Int> cardinalDirectionList = new List<Vector2Int>
    {
        new Vector2Int(0,1) , //UP
        new Vector2Int(1,0) , //RIGHT
        new Vector2Int(0,-1), //DOWN
        new Vector2Int(-1,0)  //LEFT
    };

    public static List<Vector2Int> diagonalDirectionList = new List<Vector2Int>
    {
        new Vector2Int(1,1) , //UP - RIGHT
        new Vector2Int(1,-1) , //RIGHT - DOWN
        new Vector2Int(-1,-1), //DOWN - LEFT
        new Vector2Int(-1,1)  //LEFT - UP
    };

    public static List<Vector2Int> neighboursDirectionsList = new List<Vector2Int>
    {
        new Vector2Int(0,1) , //UP
        new Vector2Int(1,1) , //UP - RIGHT
        new Vector2Int(1,0) , //RIGHT
        new Vector2Int(1,-1), //RIGHT - DOWN
        new Vector2Int(0,-1), //DOWN
        new Vector2Int(-1,-1),//DOWN - LEFT
        new Vector2Int(-1,0), //LEFT
        new Vector2Int(-1,1), //LEFT - UP
    };

    public static Vector2Int GetRandomCardinalDirection()
    {
        return cardinalDirectionList[Random.Range(0, cardinalDirectionList.Count)];
    }
    public static Vector2Int GetDirection90From(Vector2Int direction)
    {
        if (direction == Vector2Int.up)
            return Vector2Int.right;
        if (direction == Vector2Int.right)
            return Vector2Int.down;
        if (direction == Vector2Int.down)
            return Vector2Int.left ;
        if (direction == Vector2Int.left)
            return Vector2Int.up;
        return Vector2Int.zero;
    }
}
```

Програмний код демо версії гри

Код класу контролю ігрового процесу

```
public class Main : MonoBehaviour
{
    public void Lose()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
    public void EndGame()
    {
#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
#else
        Application.Quit();
#endif
    }
}
```

Код класу контролю завершення гри

```
public class Checkpoint : MonoBehaviour
{
    public Main mainController;
    private void OnTriggerEnter2D(Collider2D other)
    {

        if (other.CompareTag("Player"))
        {
            if (mainController != null)
            {
                mainController.EndGame();
            }
            else
            {
                Debug.LogWarning("Main controller not found in the scene.");
            }
        }
    }
}
```

Код класу противника

```
public class Enemy : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.tag == "Player" )
        {
            collision.gameObject.GetComponent<Player>().RecountHp(-1);
            collision.gameObject.GetComponent<Rigidbody2D>().AddForce(transform.up
*10f , ForceMode2D.Impulse);
        }
    }

    private IEnumerator Death()
    {

        GetComponent<Rigidbody2D>().bodyType = RigidbodyType2D.Dynamic;
        GetComponent<Collider2D>().enabled = false;
        GetComponentInChildren<Collider2D>().enabled = false;
        transform.GetChild(0).GetComponent<Collider2D>().enabled=false;
```

```
        yield return new WaitForSeconds(2f);
        Destroy(gameObject);
    }

    public void StartDeath()
    {
        StartCoroutine(Death());
    }

    private void OnCollisionStay2D(Collision2D collision)
    {

    }
}
```

Код класу гравця

```
public class Player : MonoBehaviour
{
    [SerializeField]private float _speed;
    private KeyCode _jumpKey;
    private KeyCode _rightMoveKey;
    private KeyCode _leftMoveKey;
    private Animator _anim;
    private bool _isGrounded;
    private bool _isHit;
    private int _currentHp;
    private int _maxHp;
    private Rigidbody2D _rb;
    public Transform groundCheck;
    public float jumpHeight;
    public Main main;
    private int _coins = 0;
    void Start()
    {
        _rb = GetComponent<Rigidbody2D>();
        _anim = GetComponent<Animator>();
        SetConfig();
    }
    void Update()
    {
        CheckGround();

        if (Input.GetAxis("Horizontal") == 0 && (_isGrounded))
        {
            _anim.SetInteger("State", 1);
        }
        else
        {
            Flip();
            if (_isGrounded)
            {
                _anim.SetInteger("State", 2);
            }
        }

        if (Input.GetKeyDown(_jumpKey) && _isGrounded)
            Jump();

    }
    private void Jump()
    {
        _rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
    }
```

```
    private void SetConfig()
    {
        _maxHp = 3;
        _currentHp = _maxHp;
        _rightMoveKey = KeyCode.RightArrow;
        _leftMoveKey = KeyCode.LeftArrow;
        _jumpKey = KeyCode.UpArrow;
    }
    private void FixedUpdate()
    {
        _rb.linearVelocity = new Vector2(Input.GetAxis("Horizontal") * _speed,
_rb.linearVelocity.y);
    }


    private void Flip()
    {
        if (Input.GetKey(_rightMoveKey))
        {
            transform.localRotation = Quaternion.Euler(0, 180, 0);
        }

        if (Input.GetKey(_leftMoveKey))
        {
            transform.localRotation = Quaternion.Euler(0, 0, 0);
        }
    }

    void CheckGround()
    {
        Collider2D[] colliders = Physics2D.OverlapCircleAll(groundCheck.position,
0.2f);
        _isGrounded = colliders.Length > 1;
        if (!_isGrounded)
        {
            _anim.SetInteger("State", 1);
        }
    }

    public void RecountHp(int deltaHp)
    {
        if (_currentHp > 0)
        {
            _currentHp += deltaHp;
        }
        else
        {
            _isHit = true;
            GetComponent<CapsuleCollider2D>().enabled = false;
            Invoke("Lose", 1.5f);
        }

        if (deltaHp < 0)
        {
            StopCoroutine(OnHit());
            _isHit = true;
            StartCoroutine(OnHit());
        }
    }
    IEnumerator OnHit()
    {
        if (_isHit)
            GetComponent<SpriteRenderer>().color = new Color(1f,
GetComponent<SpriteRenderer>().color.g - 0.02f, GetComponent<SpriteRenderer>().color.b
- 0.02f);
```

```
        else
            GetComponent<SpriteRenderer>().color = new Color(1f,
GetComponent<SpriteRenderer>().color.g + 0.02f, GetComponent<SpriteRenderer>().color.b
+ .02f);


        if (GetComponent<SpriteRenderer>().color.g == 1f)
        {
            StopCoroutine(OnHit());
        }

        if (GetComponent<SpriteRenderer>().color.g <= 0)
        {
            _isHit = false;
        }

        yield return new WaitForSeconds(0.01f);
        StartCoroutine(OnHit());
    }
    private void Lose() =>
        main.GetComponent<Main>().Lose();

    private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.gameObject.tag == "Ladder")
            {
                _rb.bodyType = RigidbodyType2D.Kinematic;
                _rb.linearVelocity = Vector2.zero;
                Debug.Log($"Trigger Enter On {collision.gameObject.tag}");
            }
        }
    private void OnTriggerStay2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Ladder")
        {
            if (Input.GetAxis("Vertical") != 0f)
            {
                transform.Translate(Vector3.up * Input.GetAxis("Vertical") * _speed *
0.5f * Time.deltaTime);
            }
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Ladder")
        {
            _rb.bodyType = RigidbodyType2D.Dynamic;
            Debug.Log($"Trigger Exit On {collision.gameObject.tag}");

        }
    }
    public void AddCoin()
    {
        _coins++;
        Debug.Log($"кількість монет = {_coins}");
    }

    public int GetCoins()
    {
        return _coins;
    }

}
```

Додаток В

Алгоритм послідовності дій для створення локації

Для початку роботи з генератором , спочатку необхідно імпортувати файл unitypackage . Це можна зробити через натискання правої кнопки по вікні проєкту та вибором пунктів Import Package -> Custom Package (рис. В.1).

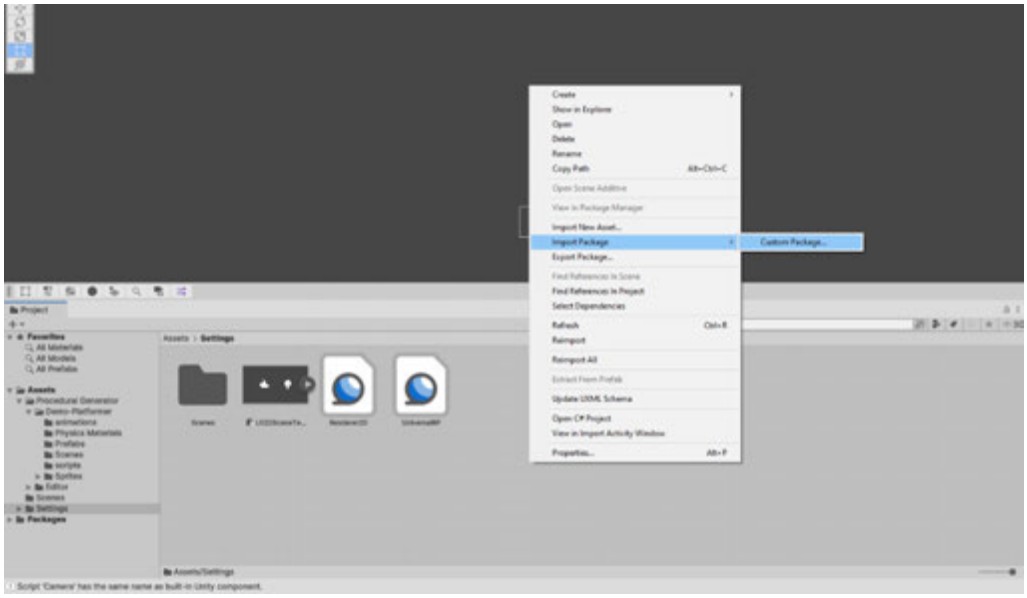


Рис. В.1 Імпортування через вікно проєкту

Після вказаних дій відкриється вікно з вибором елементів ,які будуть імпортовані (Рис. В.2). Для коректної роботи плагіну необхідні всі файли , тому достатньо буде просто натиснути на кнопку import. Після імпортування , в верхній панелі з’явиться нова вкладка “Tools” ,а в ній сам генератор , який можна викликати вікно плагіну (Рис В.3)

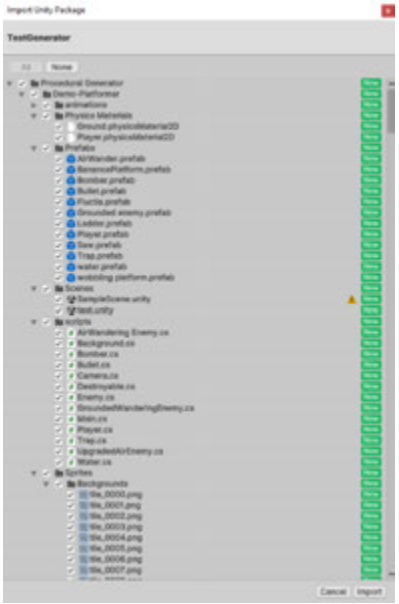


Рис. В.2 Вікно вибору файлів для імпортування

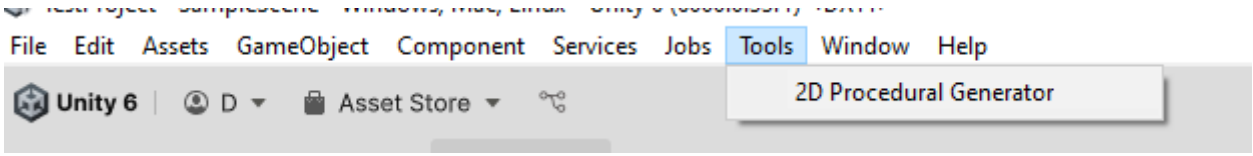


Рис. В.3 Нова вкладка з елементом виклику плагіну

При відкритті вікна всі параметри будуть встановлені , окрім двох елементів Tilemap .Це відбувається через те , що дані витягуються з файлу конфігурації ,а два елементи Tilemap пропущені через те , що вони прив’язані до сцени ,а плагін не прив’язується до конкретної сцени , що дозволяє йому генерувати локації перемикаючись між ними. В контексті даного опису послідовності дій буде використовуватись встановлений в редактор набір ресурсів ,однак , якщо необхідно використати інший набір , можна викликати інструкцію натисканням кнопки “Open tile placing instruction” та по одному встановити нові тайли , або очистити конфігураційний файл , що знаходиться за шляхом “Assets/Procedural Generator /Editor/Data/Temp/”.Для порожніх полів тайлів генерація не буде відбуватись , доки не будуть заповнені всі поля та поле статус не змінить колір на зелений з надписом “All Tile fields are filled in”.

Необхідно встановити поля з елементами Tilemap зі сцени . Для цього спочатку потрібно створити на сцені ці елементи , після чого необхідно просто перетягнути їх на місця полів генератора.

Після перерахованих вище маніпуляцій можна генерувати основу для рівня. Для основи був використаний алгоритм випадкової прогулянки(Рис. В.4).

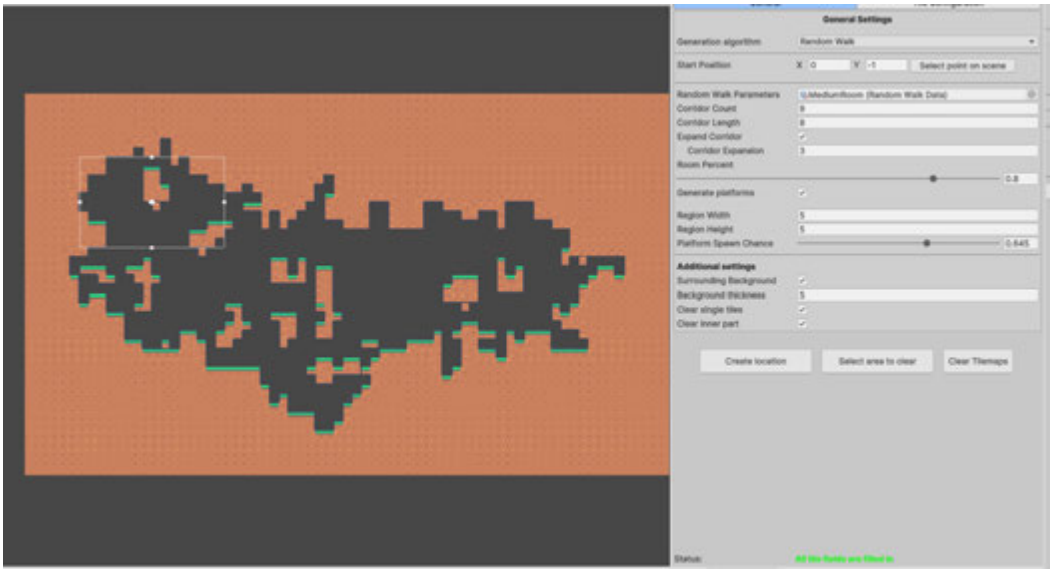


Рис. В.4 Створена основа для рівня алгоритмом випадкової прогулянки

Переглядаючи дану основу можна зрозуміти , що не всі платформи розміщені доречно, крім того рівень виходить надто малим. Необхідно використати інструмент “Select area to clear ” ,або вбудовані інструменти для роботи з TilePallete , щоб очистити зайві елементи. Після очищення рівня (Рис. В.5), можна продовжити його , використавши алгоритм шуму Перліна з формуванням рамки. Після очищення проходу ,недоречно розміщених платформ та формування продовження рівня (рис. В.5) ,можна заповнити велику порожню частину першої половини рівня платформами ,використовуючи метод ступінчатих платформ.

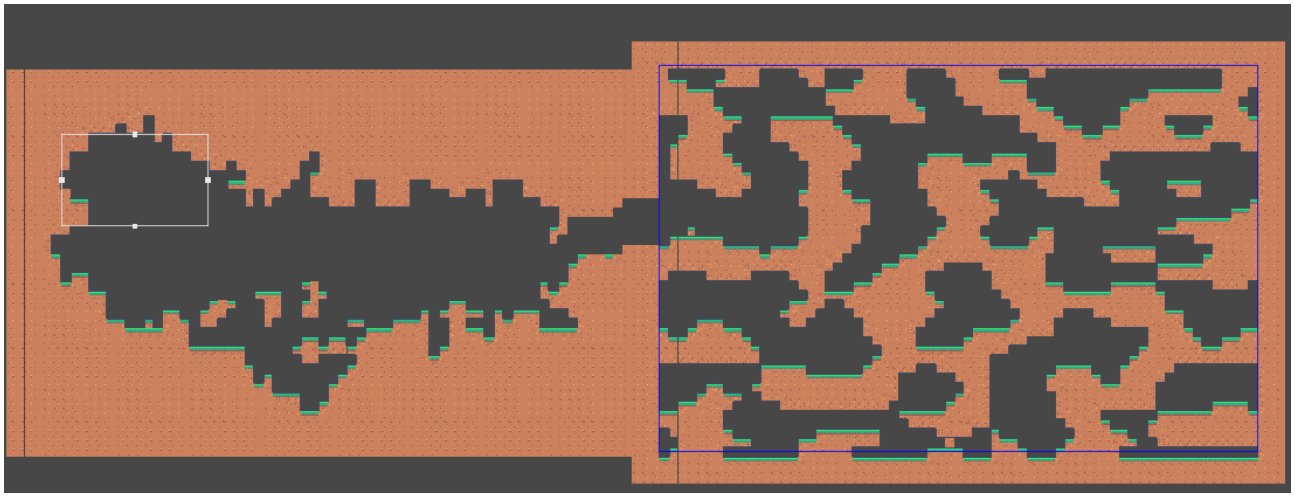


Рис. В.5 Згенероване продовження рівня

Після етапу формування основної частини рівня , що представлена на рисунку В.6 , можна використанням вбудованих інструментів рушія зробити більше проходів для гравця на другій половині рівня .

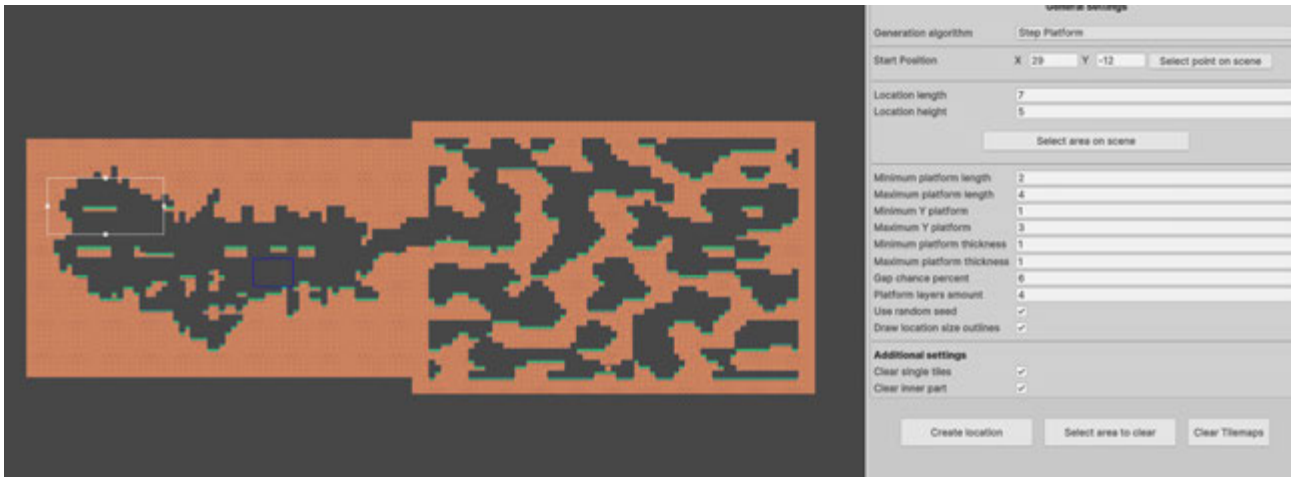


Рис. В.6 Сформована основа рівня

Після вдосконалення проходимості рівня(рис В.7) можна додати ігрові об’єкти ,пастки самого гравця та точку завершення рівня .

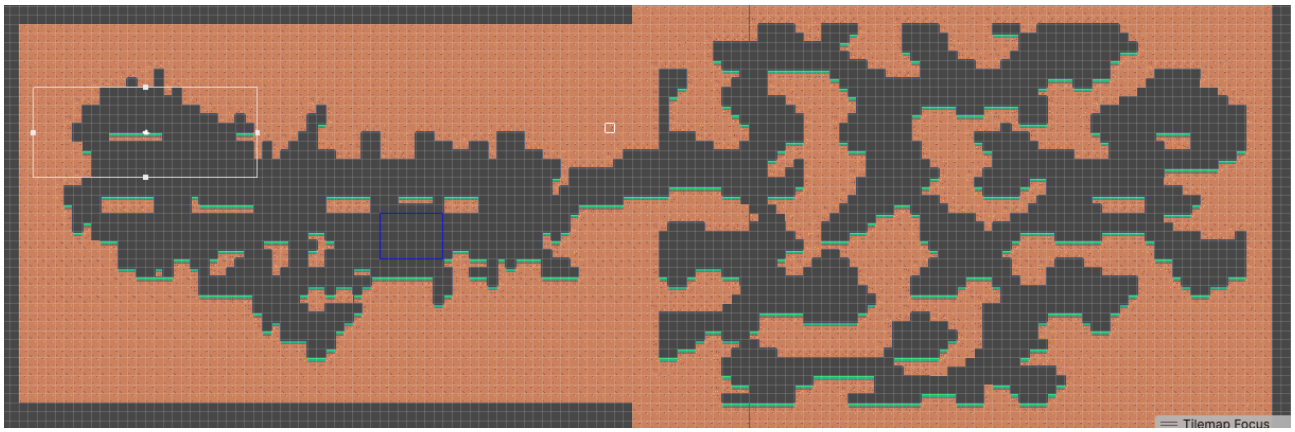


Рис. В.7 вдосконалена структура рівня

Спочатку необхідно додати об’єкти монет в складно доступні місця на карті , для мотивування гравця пошуку шляху до них та платформи , що зможуть перемістити гравця туди , якщо він не зможе дострибнути , виставивши позиції відповідних цільових точок , як показано на рисунку В.8 .

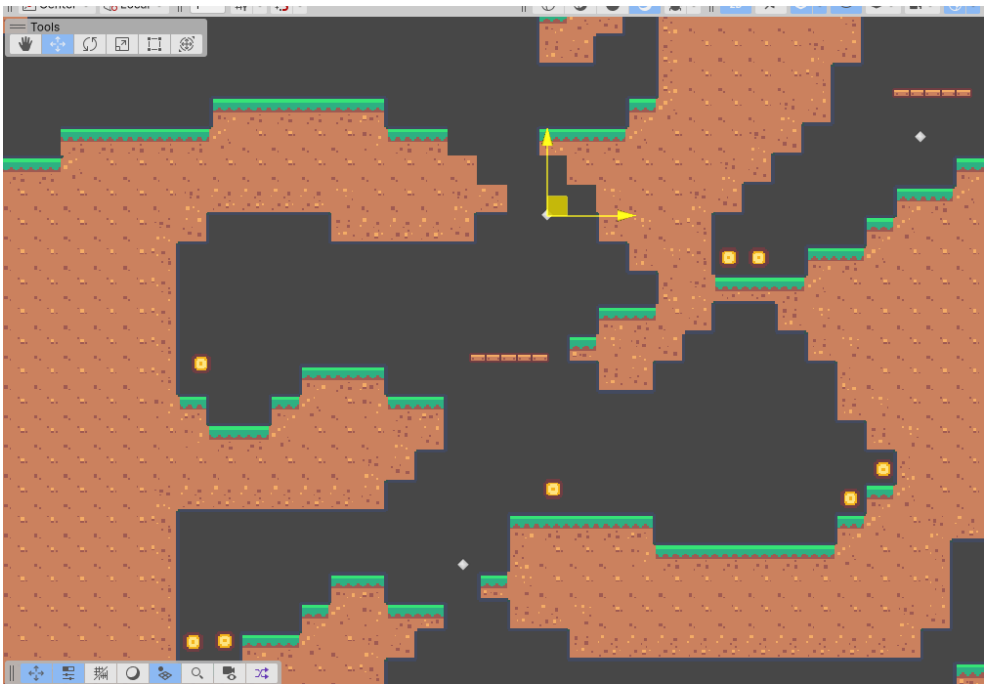


Рис.В.8 Встановлення цільових точок для руху платформи

Також можна додати платформи , що обертаються біля вузьких проходів , щоб ускладнити можливість проходу для гравця. Обертання налаштовується параметрами компоненту “Hinge joint” , а саме – “use motor” та “Motor speed”, як

показано на рисунку В.9 . Вигляд рівня з встановленими перепонами показаний на рисунку В.10.

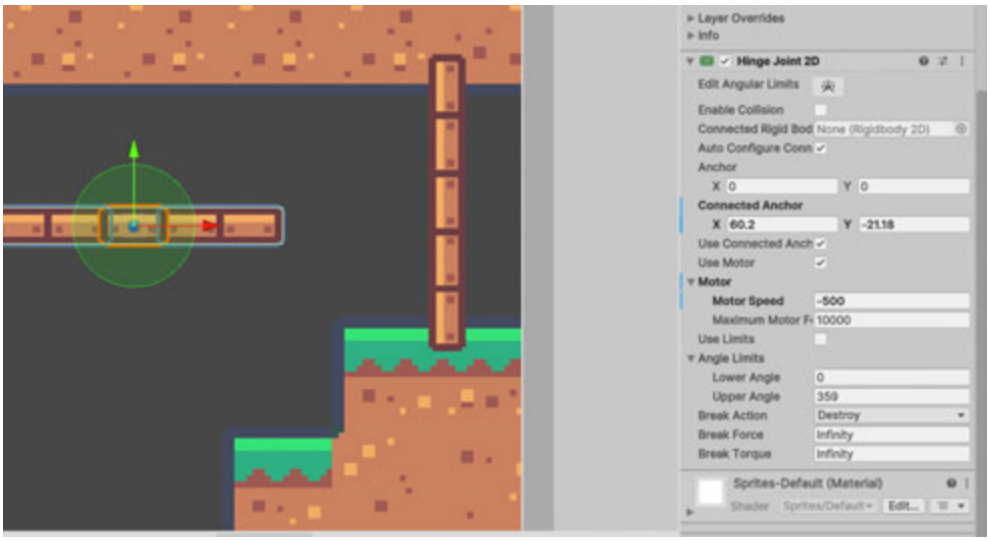


Рис. В.9 Параметри компонента для обертання платформи

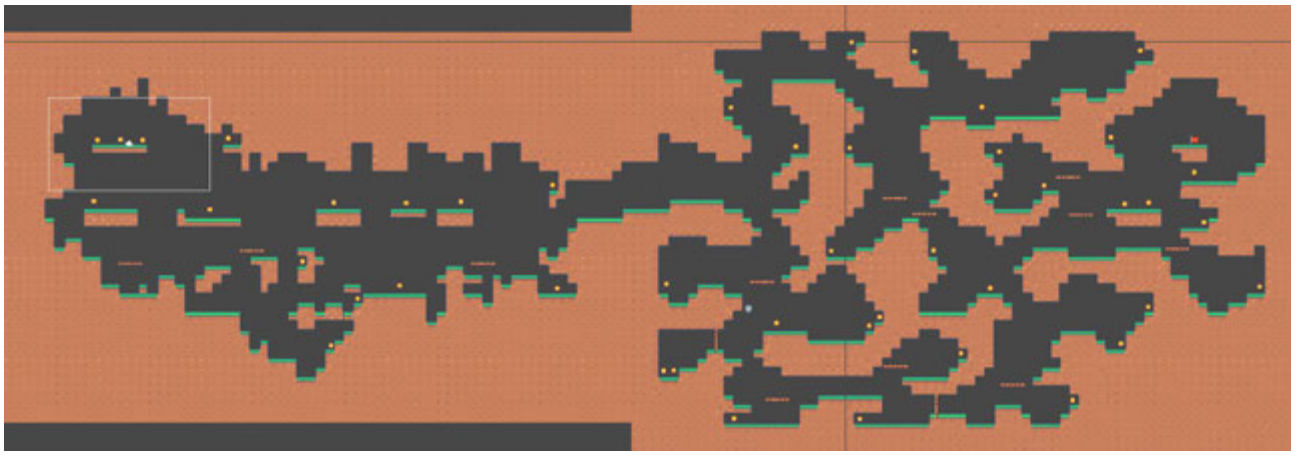


Рис. В.10 Приклад

Після встановлення перепон ,можна перейти до встановлення противників та пасток . Першим з таких об’єктів є прихована пастка, вона розміщується одразу біля вершини поверхні рівня та переміщується вверх та вниз зі встановленим періодом часу та наносять шкоду при контакті з гравцем. Такі пастки краще за все розміщувати на прямому шляху до монет та плоских поверхнях. Приклад коректного розміщення таких пасток зображений на рисунку В.11. Ще одним варіантом статичного противника є пила , яку можна розмістити в проходах .

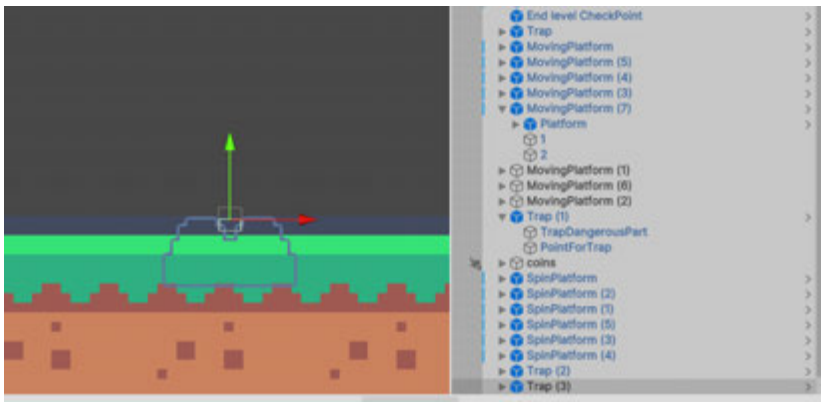


Рис. В11 Приклад розміщення прихованої пастки

Наступними йдуть противники , перший з яких наземний. Логіка його роботи полягає в безкінечному русі вліво та право , тому він ідеально підходить для плоских продовгуватих ділянок. Цей противник також має вразливу область зверху , і у випадку , якщо гравець до неї потрапить , противник буде знищений.

Звичайний літаючий противник відрізняється тим , що на його не впливає сила тяжіння та переміщуватись він може лише між двома точками , в той час , як покращений його вид може переміщуватись між масивом точок та випускати об’єкти , що наносять шкоду .

Для роботи гри буде достатньо створити в ієрархії порожній об’єкт та призначити йому компонент Main , після чого, створивши гравця та елемент End level checkpoint рівень буде повністю функціонувати. Створений рівень з наявними ігровими об’єктами , що представлений на рисунку В.12 ,можна урізноманітнити використанням різних декоративних елементів та фону,що наявні в наборі текстур через використання внутрішньої tilemap , зарання змістивши її позицію по третій координаті для коректного відображення та відсутності накладань зі стінами та наявними ігровими об’єктами .

Додавши на рівень більше декору , та прикріпивши фон до камери як дочірній об’єкт , можна запускати рівень та в подальшому додавати до гри. Створений рівень з декором продемстрований на рисунку В.13 , а безпоседеньо ігровий процес – на рисунку В.14 .

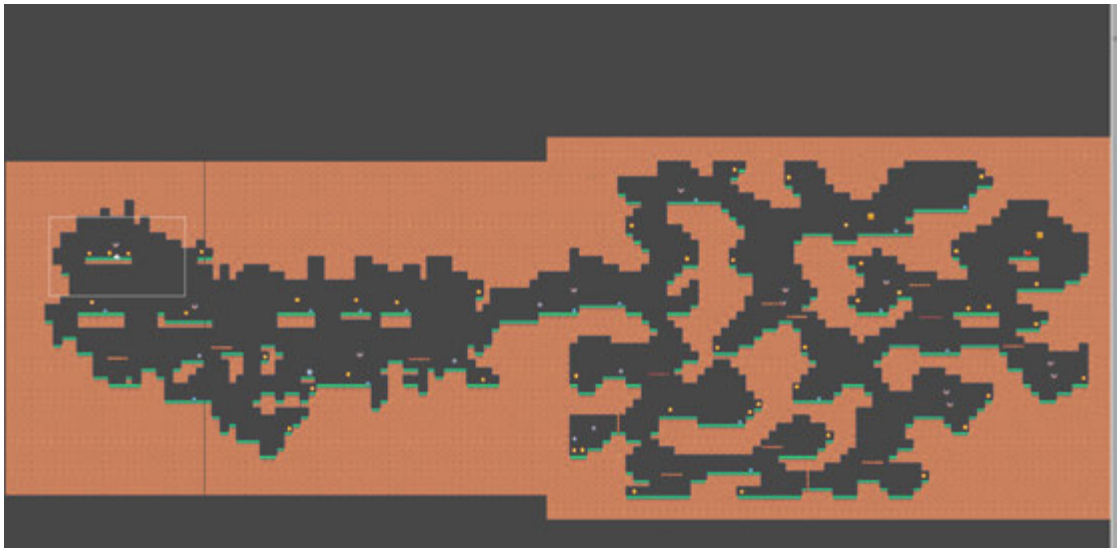


Рис. В.12 Рівень з розміщеними ігровими елементами.

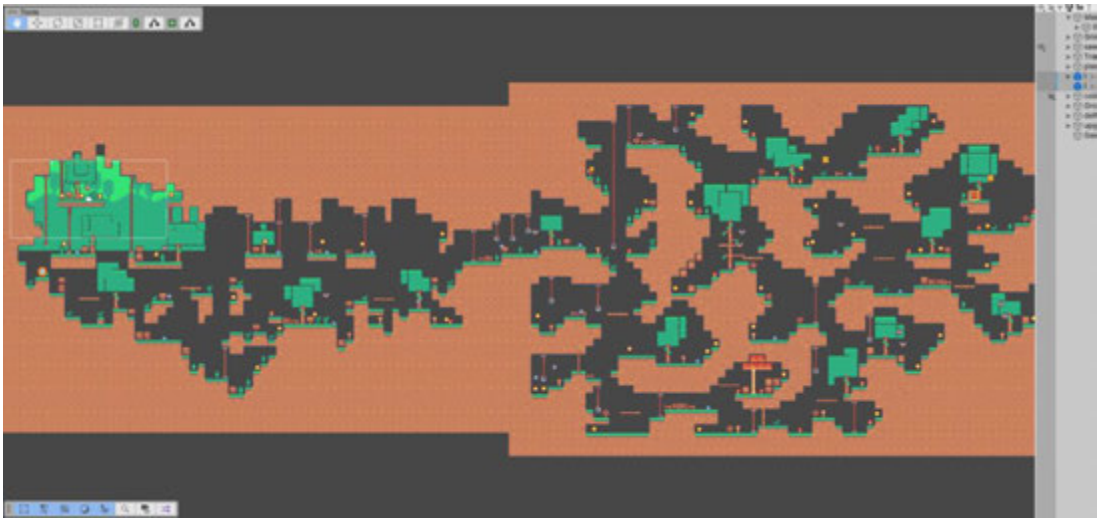


Рис. В.13 Допрацьований рівень з декором та виділеними гравцем та точкою закінчення рівня

Рис. В.14 Застосування сформованого рівня для ігрового процесу