ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод випадкової генерації рівнів для гри в жанрі Платформер 2.5D»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Валерій БІЛОУС
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63

Валерій БІЛОУС

Керівник: Олесь ДІБРІВНИЙ
доктор філософії (PhD)

Рецензент: _____
науковий ступінь,
вчене звання

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Білоусу Валерію Сергійовичу

1. Тема кваліфікаційної роботи: «Метод випадкової генерації рівнів для гри в жанрі Платформер 2.5D»

керівник кваліфікаційної роботи Олесь ДІБРІВНИЙ, доктор філософії (PhD),

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: наукова-технічна література, вимоги до точності розміщення об'єктів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження наявних методів та підходів для генерації випадкових рівнів у іграх жанру Платформер 2.5D.
2. Розробка власного алгоритму для генерації випадкових рівнів у іграх цього жанру.
3. Інтеграція розробленого алгоритму в ігрову модель.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об’єкт та предмет дослідження.
2. Актуальність роботи.
3. Порівняння методів створення рівня жанру платфомер 2.5D.
4. Математичне представлення створення рівня.
5. Алгоритм генерації рівня.
6. Приклади генерації рівня.
7. Екранна форма РМР.
8. Результати дослідження.
9. Висновки.

6. Дата видачі завдання«16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення літературних джерел	24.10-28.10.2024	
3	Складання плану роботи	29.10-01.11.2024	
4	Узгодження плану роботи та списку використаних джерел з науковим керівником	02.11-03.11.2024	
5	Дослідження інформаційних технологій	04.11-05.11.2024	
6	Розробка модифікованої математичної моделі та проведення дослідження	06.11-15.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	16.11-20.11.2024	
8	Розробка демонстраційних матеріалів	21.11-27.11.2024	
9	Попередній захист роботи	28.11.2024	

Здобувач вищої освіти _____ Валерій БІЛОУС
(підпис)

Керівник
кваліфікаційної роботи _____ Олесь ДІБРІВНИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 75 ст., 23 рис, 1 табл, 25 джерел.

Мета дослідження: покращення якості створення рівнів для ігор жанру Платформер 2.5D шляхом використання автоматичної генерації.

Об'єкт дослідження: процес випадкової генерації рівнів у грі жанру Платформер 2.5D.

Предмет дослідження: технології для генерації рівнів за допомогою штучного інтелекту.

Короткий зміст роботи: результати проведеного наукового дослідження свідчать, що використані математичні методи, алгоритмічні моделі та програмні засоби дозволяють вдосконалити процес створення випадкових рівнів для гри жанру Платформер 2.5D.

Практична цінність отриманих результатів полягає в тому, що на основі проведених теоретичних досліджень було розроблено новий метод покращення якості створення рівнів для ігор жанру 2.5D платформер. Ці результати дозволяють у подальшому прискорити роботу алгоритмів штучного інтелекту для генерації рівнів.

КЛЮЧОВІ СЛОВА: Платформер 2.5D, ГЕНЕРАЦІЯ РІВНЯ

ABSTRACT

Text part of the master`s qualification work: 75 pages, 23 figures, 1 table, 25 sources.

Purpose of the study: improving the quality of level creation for Platformer 2.5D games by using automatic generation.

Object of research: the process of random generation of levels in the game genre Platformer 2.5D.

Subject of research: algorithms for level generation using artificial intelligence

Summary of the work: The results of the conducted scientific research show that the used mathematical methods, algorithmic models and software tools allow to improve the process of creating random levels for the game genre Platformer 2.5D.

The practical value of the obtained results is that, based on the theoretical research, a new method for improving the quality of level creation for 2.5D platformer games was developed. These results allow us to further speed up the work of artificial intelligence algorithms for level generation.

KEYWORDS: 2.5D platformer, level generation

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Pl 2.5D – Platformer 2.5D

LvlGen – Генерація рівнів

PMP – Розроблений метод генерації рівнів

NS – Перлинний шум

CA – Клітинний автомат

ВСТУП

Сьогодні кожен, навіть той, хто дуже любить працю, потребує відпочинку. Один із найкращих способів розслабитися - це гра в відеоігри. Особливо це актуально, коли ти їдеш у метро або в іншому транспорті і хочеш зняти напругу.

Жанр Платформер 2.5D приваблює гравців завдяки поєднанню класичного двовимірного геймплею з тривимірною графікою, що створює унікальний візуальний стиль, надає можливість для інноваційного дизайну рівнів і забезпечує глибокий та різноманітний геймплей з елементами дослідження та вирішення головоломок.

Зростаючий інтерес до жанру Платформер 2.5D стимулює індустрію постійно шукати нові способи його покращення. Один із ключових аспектів цього процесу - розробка нових методів генерації рівнів, які дозволяють створювати їх швидше та якісніше.

Завдання дослідження, які необхідно виконати для досягнення поставленої мети:

1. Дослідження наявних методів та підходів для генерації випадкових рівнів у іграх жанру Платформер 2.5D.
2. Розробка власного алгоритму для генерації випадкових рівнів у іграх цього жанру.
3. Інтеграція розробленого алгоритму в ігрову модель.
4. Проведення тестування розробленого алгоритму.
5. Порівняння результатів з існуючими методами.

Об'єкт дослідження: процес випадкової генерації рівнів у грі жанру Платформер 2.5D.

Предмет дослідження: алгоритми для генерації рівнів за допомогою штучного інтелекту.

Мета дослідження: покращення якості створення рівнів для ігор жанру Платформер 2.5D шляхом використання автоматичної генерації.

Результати проведених наукових досліджень демонструють, що використання математичних методів, алгоритмічних моделей і програмних засобів дозволяє значно покращити процес створення випадкових рівнів для ігор жанру Платфомер 2.5D.

Практична цінність отриманих результатів полягає в тому, що на основі проведених теоретичних досліджень розроблено новий метод підвищення якості створення рівнів в іграх жанру Платфомер 2.5D. Це дослідження дозволяє в майбутньому прискорити роботу алгоритмів штучного інтелекту для генерації рівнів.

Актуальність даної роботи полягає в необхідності покращення якості та різноманітності генерації випадкових рівнів в іграх жанру Платфомер 2.5D. Одним із ключових аспектів успіху таких ігор є створення цікавих і різноманітних рівнів, які надають гравцям непередбачуваний і захоплюючий ігровий досвід.

Наукова новизна роботи полягає в розробці методу випадкової генерації рівнів для ігор жанру Платфомер 2.5D. Запропонований метод дозволяє автоматично створювати різноманітні, інтерактивні та збалансовані ігрові рівні, враховуючи як естетичні, так і ігрові аспекти. На відміну від існуючих підходів, новий метод забезпечує високу варіативність рівнів, зберігаючи належний рівень складності та інтерактивності для гравців. Крім того, метод дозволяє адаптивно генерувати рівні в реальному часі, що підвищує динаміку і відтворюваність ігрового процесу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття відеоігор і жанру Платформер 2.5D

Відеоігри - це інтерактивні розважальні програми, які включають візуальні, аудіо та інші мультимедійні компоненти для створення віртуального світу або імітації реальних чи уявних ситуацій. Ці ігри можуть бути розроблені для різних платформ, таких як персональні комп’ютери, ігрові консолі, мобільні пристрої та інші.

Вони можуть варіюватися від простих аркадних ігор до складних симуляторів, стратегій, пригодницьких ігор, шутерів та інших жанрів. Гравець взаємодіє з відеоіграми за допомогою контролерів, клавіатур, мишок, сенсорних екранів або інших пристроїв введення.

Відеоігри можуть мати як одиночний, так і багатокористувацький режими. Вони часто мають сюжетну лінію або завдання, які гравець повинен виконати, або вони можуть запропонувати відкритий світ для дослідження та вільного розвитку.

Окрім розважальної функції, відеоігри також використовуються для навчання, навчання, соціальної взаємодії та інших цілей. Вони стали важливою частиною сучасної поп-культури та мають значний вплив на суспільство.

Одним із популярних жанрів відеоігор є «платформер». Платформери відомі своїм наголосом на переміщенні головного героя рівнями, які зазвичай складаються з платформ і перешкод, таких як прірви, вороги тощо. Головна мета гравця — пройти ці рівні, уникнути або знищити ворожі перешкоди та досягти кінцева мета, яка може бути представлена завершенням рівня або досягненням конкретної мети.

Платформер 2.5D - це різновид жанру, де гра використовує 3D-графіку для створення візуального ефекту глибини, але ігровий процес залишається здебільшого у 2D-площині. Це означає, що головний герой рухається вгору, вниз,

вліво і вправо, але також може рухатися в глибину, тобто вглиб екрана. Такий підхід дозволяє поєднувати класичний геймплей платформера з ефектними 3D-зображеннями.

Такий підхід дозволяє розширити можливості ігрового процесу, додавши нові аспекти взаємодії з навколишнім середовищем, такі як рух у глибину, приховані об’єкти або головоломки, які гравцеві потрібно вирішити. Платформери 2.5D можуть бути яскравими та захоплюючими іграми, які поєднують класичну механіку з сучасними візуальними ефектами.

1.1.1 Приклади ігор жанру Платформер 2.5D

Платформери 2.5D - це ігри, які поєднують елементи класичних платформерів з тривимірним середовищем і рухом у двовимірній площині.

LittleBigPlanet - це серія ігор від Media Molecule, яка поєднує в собі унікальний ручний дизайн із пригодницьким геймплеєм. Гравці мають можливість створювати власні рівні, використовуючи різноманітні об’єкти та механіки, а потім ділитися ними з іншими користувачами онлайн.

Рис 1.1 Кадр з гри LittleBigPlanet

У LittleBigPlanet гравці керують персонажем на ім’я Sackboy або Sackgirl, який повинен проходити різні рівні, збираючи предмети та розгадуючи головоломки. Гра має яскравий художній стиль і милу атмосферу, що допомагає створити приємну ігрову атмосферу.

У LittleBigPlanet гравці керують персонажем на ім'я Sackboy або Sackgirl, які мають просуватися через різноманітні рівні, збираючи предмети та розв'язуючи головоломки. Гра має яскравий арт-стиль та милу атмосферу, що допомагає створити приємну ігрову атмосферу.

Основна увага серії LittleBigPlanet зосереджена на спільноті гравців та їхній творчості. Гравці можуть створювати власні рівні, персонажів, музику та інші елементи гри за допомогою інструментів, доступних у редакторі рівнів.

LittleBigPlanet відрізняється великою кількістю доступного контенту, включаючи тисячі рівнів, створених гравцями з усього світу. Це дозволяє гравцям постійно насолоджуватися новими викликами та творчими ідеями.

Загалом, LittleBigPlanet - це неперевершений досвід для творчих гравців, які цінують можливість співпрацювати та ділитися своїми творіннями з іншими.

Inside - відеогра, розроблена та опублікована Playdead. Гра була випущена в 2016 році і є духовним наступником їхньої попередньої роботи, Limbo. Inside отримав надзвичайно позитивні відгуки від критиків і гравців, особливо за його атмосферу, глибокий сюжет і геймплей.

Рис 1.2 Кадр з гри Inside

Гра має художній стиль, який використовує похмуру та таємничу атмосферу, яка допомагає поглибити емоційний досвід гравця. Чудова графіка і деталізація відображають небезпечний і таємничий світ, який відправляється досліджувати персонаж.

З точки зору ігрового процесу, Inside - це платформер з елементами головоломки та прихованими таємницями. Гравець керує безіменним хлопчиком, який подорожує темним світом, повним небезпек і таємниць. Йому потрібно виконувати різноманітні завдання, зокрема перешкоджати ворогам, розгадувати головоломки та використовувати середовище для виживання.

Однією з головних переваг Inside є його захоплюючий сюжет. Гравцям потрібно розгадувати та інтерпретувати різні головоломки та події в грі, щоб зрозуміти справжнє значення та мотиви захоплюючої подорожі головного героя. Також гра славиться своєю несподіваністю і непередбачуваністю, що змушує гравців сидіти на краю своїх місць до самого кінця.

Загалом Inside - це вражаючий і унікальний досвід для тих, хто цінує захоплюючі сюжетні лінії, атмосферну графіку та непередбачувану ігрову механіку.

Little Nightmares 2 - це хоррор-платформер, розроблений Tarsier Studios і виданий Bandai Namco Entertainment. Випущена у 2021 році гра є продовженням популярних «Little Nightmares», і знову переносить гравців у сюрреалістичний та химерний світ, який вражає своєю атмосферою та глибиною.

Рис 1.3 Кадр з гри Little Nightmares 2

У Little Nightmares 2 гравець бере на себе роль хлопчика на ім'я Моно, який разом з дівчинкою на ім'я Гір подорожує світом, наповненим жахливими та загадковими істотами, а також лабіринтами головоломок і пасток. Гравцеві доведеться пройти безліч рівнів, де кожен крок може виявитися смертельно небезпечним.

Гра відрізняється відмінною атмосферою і детальним виконанням. Його графіка та дизайн допомагають створити напружену та моторошну атмосферу, де кожен об’єкт і персонаж мають свою унікальну ауру таємниці.

Little Nightmares 2 робить великий акцент на головоломках і взаємодії з оточенням. Гравцеві потрібно вирішувати різні головоломки, шукати приховані шляхи і уникати пасток, щоб просуватися в ігровому світі.

Одним із ключових елементів Little Nightmares 2 є його сюжет. Гравці занурюються в таємничий і химерний світ, де кожен персонаж і подія мають свою

історію і таємниці. Сюжет розгортається поступово, надихаючи гравців досліджувати та розгадувати таємниці.

Загалом, Little Nightmares 2 - це захоплююча та напружена гра, яка пропонує гравцям неперевершений досвід у світі моторошних фантазій і таємниць.

TRY AGAIN — це платформер, розроблений і виданий Alva Majo. Гра вийшла в 2020 році для Microsoft Windows і macOS.

Рис 1.4 Кадр з гри Try Again

У грі TRY AGAIN гравці контролюють маленький квадрат, який повинен подолати різноманітні перешкоди та виклики на шляху до фінішу. Це відбувається у формі швидких і ритмічних рівнів, де реакція та точність гравця є вирішальними для успіху.

Гра має простий, але стильний художній стиль, який додає відчуття мінімалізму та сучасності. Музика та звукове оформлення також допомагають створити відповідну атмосферу та підкреслюють ритмічність гри.

TRY AGAIN відрізняється високою складністю і вимагає від гравців швидкої реакції та спритності. Невеликий розмір квадрата і швидкість гри роблять кожен рівень захоплюючим і викликають бажання пройти його знову.

Усі ці елементи роблять TRY AGAIN чудовим вибором для любителів випробувань і платформерів, які цінують швидкість, точність і реакцію.

1.2 Методи створення рівнів в жанрі Платформер 2.5D

Створення рівнів у жанрі платформерів 2.5D - це творчий процес, що включає в себе багото методів та підходів. Ось деякі з них:

1. Ручного створення рівнів;
2. Генерація випадкових рівнів;
3. Комбінований підхід;
4. Тестування та зміни;
5. Перлинний шум(Perlin Noise);
6. Шум Симплексу(Simplex Noise);
7. Клітинний автомат (Cellular Automata);
8. Поділ простору (Space Partitioning);
9. Моделі на основі шаблонів (Pattern-based Models);
10. Генерація на основі графів (Graph-based Generation);
11. Генетичні алгоритми;
12. Wave Function Collapse (WFC);

1.2.1 Метод ручного створення рівнів

Метод створення рівня вручну в жанрі платформера 2.5D - це творчий процес, який передбачає детальне проектування та розміщення елементів гри. Розробники вручну створюють платформи, перешкоди, об'єкти і визначають їх точне розташування на рівні. Такий підхід дозволяє забезпечити високу якість та унікальність кожного рівня, але потребує значних ресурсів часу та зусиль.

На початковому етапі створюється концепція рівня, яка включає визначення теми, цілей та основних характеристик. Потім розробники використовують спеціальні редактори рівнів для реалізації своєї ідеї,

розташовуючи об'єкти відповідно до заданої концепції. Цей процес враховує естетичні аспекти, логіку проходження та інтерактивність.

Після створення структури базового рівня проводиться тестування, щоб перевірити баланс ігрового процесу та загальної якості. На цьому етапі визначаються можливі помилки або недоліки, які необхідно виправити. Зміни та покращення рівня вносяться на основі результатів тестування, щоб забезпечити гравцям оптимальний досвід.

Останній етап – тестування підсумкового рівня. Після підтвердження рівень включається в остаточну версію гри, яка потім оприлюднюється. Цей підхід дозволяє створювати детальні та захоплюючі рівні, які відповідають очікуванням гравців.

Рис 1.5 Приклад методу створення рівнів

1.2.2 Метод генерації випадкових рівнів

Метод створення рівня вручну в жанрі платформера 2.5D - це творчий процес, який передбачає детальне проектування та розміщення елементів гри. Розробники вручну створюють платформи, перешкоди, об'єкти і визначають їх

точне розташування на рівні. Такий підхід дозволяє забезпечити високу якість та унікальність кожного рівня, але потребує значних ресурсів часу та зусиль.

На початковому етапі створюється концепція рівня, яка включає визначення теми, цілей та основних характеристик. Потім розробники використовують спеціальні редактори рівнів для реалізації своєї ідеї, розташовуючи об'єкти відповідно до заданої концепції. Цей процес враховує естетичні аспекти, логіку проходження та інтерактивність.

Після створення структури базового рівня проводиться тестування, щоб перевірити баланс ігрового процесу та загальної якості. На цьому етапі визначаються можливі помилки або недоліки, які необхідно виправити. Зміни та покращення рівня вносяться на основі результатів тестування, щоб забезпечити гравцям оптимальний досвід.

Останній етап - тестування підсумкового рівня. Після підтвердження рівень включається в остаточну версію гри, яка потім оприлюднюється. Цей підхід дозволяє створювати детальні та захоплюючі рівні, які відповідають очікуванням гравців.

1.2.3 Комбінований підхід

Гібридний підхід до створення рівнів у жанрі платформера 2.5D поєднує в собі переваги ручного проектування та автоматичного створення рівнів, забезпечуючи баланс між контрольованим дизайном і випадковістю. Перший етап - це створення шаблонів базового рівня, які розробники вважають типовими або ідеальними для гри. Ці шаблони містять різні типи платформ, перешкод, ворогів та інших елементів, а також зони для розміщення додаткових об'єктів.

При створенні нового рівня програма використовує один із цих шаблонів як основу. Потім за допомогою алгоритмів випадкової генерації додаються додаткові об’єкти, такі як платформи, вороги або бонуси, які розміщуються у випадкових місцях, створюючи унікальний досвід для кожного рівня.

Після автоматичної генерації розробники вручну налаштовують деталі рівня, підлаштовуючи його для забезпечення збалансованого ігрового процесу.

Цей етап може включати додавання або видалення об'єктів, зміну їх розташування та інші налаштування, спрямовані на покращення ігрового процесу.

На наступному етапі рівень проходить процес тестування та налагодження. Перевіряється баланс і якість ігрового процесу. У разі виявлення недоліків розробники вносять необхідні зміни. Цей процес повторюється до досягнення бажаного результату. Завершальним етапом є включення рівня в гру після завершення всіх виправлень і фінального тестування. Це гарантує, що рівень відповідає загальним стандартам якості та забезпечує позитивний досвід для гравців.

Таким чином, комбінований підхід дозволяє створювати різноманітні, якісні та цікаві для гравців рівні, поєднуючи структуру шаблонів і креативність автоматичної генерації.

1.2.4 Тестування та зміни

Методологія тестування та вдосконалення жанру платформера 2.5D базується на систематичному підході до тестування рівнів, виявлення недоліків та їх усунення для покращення ігрового процесу. Спочатку проводиться тестування геймплея, яке перевіряє рівень на наявність помилок і недоліків. Тестери досліджують рівень, випробовуючи різні стратегії та шляхи, щоб знайти проблеми в дизайні, балансі або взаємодії гравця з елементами гри.

На основі отриманих результатів розробки вносяться зміни. Це може включати виправлення помилок, зміну розташування об’єктів, коригування складності або додавання нових елементів, які покращують ігрову якість процесу. Після внесення змін до додаткового тестування, щоб переконатися, що проблеми усуваються, а нові зміни позитивно впливають на геймплей. Якщо у вас є нові недоліки, цикл повторюється.

Процес тестування та внесення змін може бути ітераційним, тобто повторюватися кілька разів до досягнення бажаного результату. Завдяки цьому забезпечується оптимальний баланс між якістю рівня та задоволенням від гри.

Якщо всі завдання вирішено, рівень проходить фінальне тестування, яке підтверджує його готовність до включення у фінальну версію гри.

Завдяки цьому методу розробники можуть систематично вдосконалювати рівні, створюючи продукт високої якості, який дарує гравцям захопливий і добре збалансований ігровий досвід.

1.2.5 Перлинний шум(Perlin Noise)

Перлинний шум (Perlin Noise) - це метод, що використовується для створення плавних, псевдовипадкових переходів, які нагадують природні ландшафти. Його широко застосовують у 2D і 3D іграх для генерації таких елементів, як рельєф, ландшафт або платформи. У 2.5D платформерах цей метод також може бути корисним для генерації розташування платформи, забезпечуючи непрямолінійні рівні з природними переходами висоти.

Перлинний шум створює гладкий «плавний» шум, у якому значення між точками змінюється поступово. Це робить його ідеальним для моделювання природних структур, таких як рельєф. Шум генерує однакові значення за однаковими координатами, що забезпечують стабільні та передбачувані результати. Контролюючи параметри частоти та амплітуди, можна впливати на масштаб і рівень деталей: низька частота створює плавні переходи, а висока забезпечує більшу деталізацію.

Перлинний шум можна використовувати для генерації висоти платформи. Це дозволяє створювати природні підйоми та спуски. Координата Y платформи змінюється відповідно до значення шуму для координати X, що забезпечує гармонійний рівень вигляду, уникаючи випадковості.

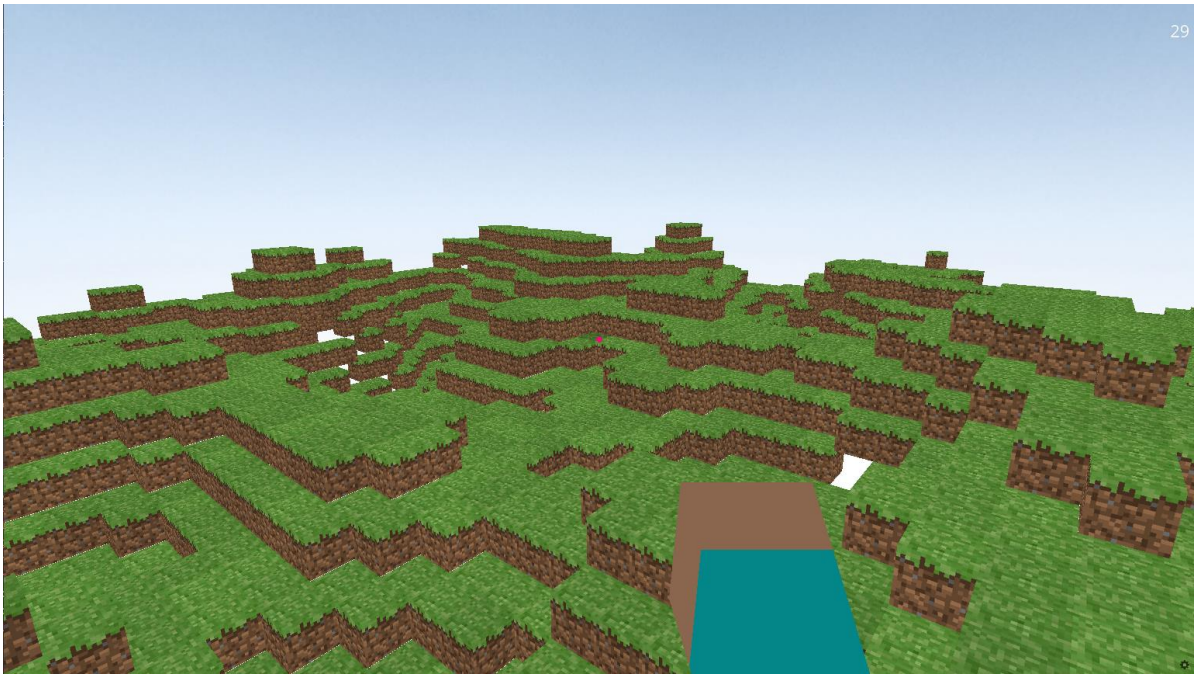


Рис 1.6 Приклад методу Perlin Noise

Шум також ефективно використовується для створення ландшафтів або підземних печерних споруд. Це додає візуальну глибину та різноманітність рівню, створюючи плавні переходи між платформами та іншими елементами. Наприклад, зміни значень шуму можуть визначити наявність стін або інших об'єктів на рівні.

Динамічно змінювати складність рівнів можна шляхом впливу на параметри шуму. Наприклад, збільшення частоти шуму ускладнює проходження рівня через більш різкі зміни висоти платформ.

Шум також можна використовувати для випадкового генерування об’єктів, таких як бонуси, вороги або декоративні елементи, розміщуючи їх на різній висоті. Це додає різноманітності та створює більш цікавий ігровий процес.

Перламутровий шум дозволяє створити більш природні майданчики та рівні елементи, що робить дизайн більш візуально привабливим. Можливість регулювати параметри шуму забезпечує гнучкість у створенні рівнів різної складності. Крім того, шум генерує стабільний результат з однаковими параметрами, що значно полегшує тестування та внесення змін.

1.2.6 Шум Симплексу(Simplex Noise)

Симплексний шум — це сучасна техніка процедурної генерації, яка використовується для створення різноманітних елементів у відеоіграх, у тому числі рівнів 2,5D-платформерів. Його особливістю є здатність генерувати природні, плавні та органічні форми, що додає реалістичності рівням та забезпечує якісну інтеграцію елементів у загальний дизайн. Цей алгоритм було створено як удосконалення класичного шуму Перліна, усунувши обмеження його кубічної решітки та зменшивши обчислювальну складність. У контексті 2.5D-платформерів Simplex Noise дозволяє генерувати рівні зі складними ландшафтами, динамічною висотою платформи та змінними областями.

Основна ідея Simplex Noise заснована на використанні симплексів - багатовимірних форм, таких як трикутники в 2D і тетраедри в 3D. На відміну від шуму Perlin, який використовує кубічну сітку, Simplex Noise використовує трикутну сітку, яка забезпечує більш рівномірний розподіл точок і зменшує ймовірність артефактів. Це особливо важливо для платформерів, де візуальні та геометричні проблеми можуть заважати ігровому процесу. Кожній точці сітки присвоюється значення на основі її відстані від сусідніх вершин симплекса, що забезпечує плавні переходи між різними областями рівня.

У 2.5D платформерах симплексний шум використовується для створення різних елементів рівня. Наприклад, висоту платформ можна визначити за значеннями шуму: високі значення відповідають пікам, тоді як низькі значення можуть вказувати на ями або відкриті простори. Це дозволяє плавно та природно змінювати висоту, що виглядає гармонійно та додає динамічності ігровій карті. Крім того, шум можна використовувати для визначення розташування ворогів, бонусів, пасток або декоративних елементів, таких як дерева чи скелі, які додають атмосферу рівню.

Однією з переваг симплексного шуму є його здатність працювати у великих розмірах з мінімальними обчислювальними витратами. Це означає, що навіть складні рівні з великою кількістю деталей можна генерувати в реальному часі без істотного впливу на продуктивність гри. Це робить алгоритм ідеальним для

динамічної генерації рівнів, які змінюються під час гри. Наприклад, платформи можуть змінювати висоту або рухатися залежно від зміни значень шуму, створюючи додатковий виклик для гравця.

Simplex Noise також дозволяє регулювати різні аспекти генерації, такі як частота, амплітуда та кількість октав. Низька частота може створювати великі плавні перепади, тоді як висока частота додає дрібні деталі, що збільшує складність рівня. Використання кількох октав дозволяє накладати різні рівні деталізації, що особливо корисно для створення складних пейзажів або змінних платформ.

Однак, незважаючи на свої переваги, шумовий алгоритм Simplex має свої труднощі. Його ефективне використання вимагає розуміння параметрів і налаштувань, що може бути складним для новачків. Крім того, для великих рівнів може знадобитися додаткова оптимізація, щоб уникнути проблем із продуктивністю. Але ці труднощі компенсуються якістю результатів, оскільки рівні, створені за допомогою шуму Simplex, виглядають природно, працюють стабільно та додають унікальності кожній грі.

Загалом Simplex noise є потужним інструментом для розробників 2.5D платформерів. Це не тільки забезпечує гнучкість дизайну рівнів, але й дозволяє створювати унікальні ігрові світи з реалістичною геометрією та складними елементами. Його інтеграція в процес розробки дозволяє підвищити якість ігрового процесу, роблячи кожен рівень цікавішим і захоплюючим для гравців.

1.2.7 Клітинний автомат (Cellular Automata)

Cellular Automata — це алгоритм генерації рівнів, який базується на використанні сітки клітинок, де кожна клітинка може перебувати в одному з кількох станів. У контексті 2.5D-платформування цей метод використовується для створення неправильних і природних форм, таких як печери, лабіринти або складні платформи. Такий підхід дозволяє створювати унікальні рівні з динамічними формами, які ідеально підходять для середовищ з хаотичним розташуванням елементів.

Метод починається з ініціалізації сітки клітинок, яка відповідає простору гри. Кожній комірці випадковим чином призначається початковий стан, наприклад «платформа» або «пустота». Це робиться за допомогою імовірнісного заповнення, де певний відсоток комірок позначається як платформи. Потім застосовуються правила, які визначають, як змінюватиметься стан кожної комірки залежно від станів її сусідів. Наприклад, комірка залишається платформою, якщо поруч з нею достатньо інших платформ, і стає порожньою, якщо таких сусідів недостатньо.

Процес зміни станів клітин повторюється кілька разів (звичайно 4–6 разів). Це дозволяє сітці стабілізуватися, створюючи структури, які виглядають природно. Результатом є платформи, які можуть бути нерегулярними та хаотичними, але водночас функціональними. Після цього розробники додають деталі рівня: з’єднують платформи мостами, додають пастки, ворогів або бонуси для створення цікавого ігрового процесу.

Однією з головних переваг клітинного автомата є природність і органічність форм. На відміну від звичайних платформ, створених за допомогою інших методів, клітинний автомат генерує нерівні, хаотичні рівні, які можуть нагадувати реальні природні середовища, такі як печери чи скелі. Крім того, алгоритм дозволяє легко змінювати такі параметри, як початкова щільність платформ або правила зміни станів, щоб отримувати різні версії рівнів.

Ще однією важливою перевагою є простота реалізації. Незважаючи на хаотичність кінцевого результату, сам алгоритм досить простий і не вимагає складних обчислень. Його можна легко інтегрувати в ігровий движок, наприклад Unity, і швидко отримувати різноманітні результати, які не виглядатимуть шаблонно.

Для 2.5D-платформера цей метод особливо корисний, коли потрібно створити рівні з великою кількістю змінних платформ. Це дозволяє уникнути монотонності, зберігаючи при цьому можливість налаштування рівнів під конкретні завдання. Завдяки своїй гнучкості клітинний автомат може бути

основою для створення як складних лабіринтів, так і відкритих просторів з платформами, забезпечуючи різноманітний ігровий досвід.

Цей метод надає можливість створювати унікальні рівні з різноманітними природними структурами та лабіринтами. Платформи виглядатимуть природно, ніби створені навколишнім середовищем, а не розміщені хаотично.

Переваги методу клітинного автомата для генерації рівнів у 2.5D-платформерах полягають у його здатності створювати органічні природні структури, які імітують печери чи підземні тунелі, додаючи грі унікальний візуальний стиль. Такий підхід дозволяє створити цікавий і унікальний ландшафт, який не виглядає випадковим і має природне відчуття. Крім того, настроювані параметри, такі як початкова щільність заливки, кількість ітерацій і правила сусідства, дозволяють гнучко змінювати структуру рівнів, адаптуючи їх до потрібної складності або стилю гри. Клітинний автомат також ефективний і багаторазовий — згенеровані рівні будуть різноманітними навіть за однакових правил, що сприяє більшій варіативності ігрового процесу.

1.2.8 Поділ простору (Space Partitioning)

Поділ простору - це техніка створення рівнів, яка використовується для створення чітко структурованих і логічно організованих рівнів. У контексті 2.5D-платформера цей підхід дозволяє розділити ігровий простір на менші зони, такі як кімнати, платформи або коридори, і з’єднати їх разом. Метод забезпечує добре організовану структуру, яка сприяє цікавому ігровому процесу, надаючи простір для додавання ворогів, бонусів та інших елементів.

Основна ідея методу полягає в рекурсивному розділенні простору на сектори за допомогою деревоподібної структури, такої як дерево BSP (Binary Space Partitioning). Початковий простір рівня (зазвичай прямокутник) розділений на дві частини, горизонтально або вертикально. Цей процес повторюється для кожного сектора, доки не буде досягнуто бажаного рівня деталізації. У цих секторах створюються кімнати, які можуть мати різні розміри та форми, а також платформи чи інші об’єкти, що стосуються ігрового процесу.

Після створення секторів вони з’єднуються за допомогою коридорів або платформ, що додає рівню цілісності. Наприклад, коридори можуть з’єднувати кімнати, створюючи природні шляхи для переміщення гравця. Ви також можете налаштувати унікальні переходи, такі як вертикальні платформи або ліфти, щоб зробити рівень більш динамічним.

Цей метод має багато переваг. По-перше, це дозволяє створювати логічно організовані рівні, які легко поділяються на функціональні частини (кімнати, коридори, майданчики). По-друге, це дозволяє контролювати такі параметри генерації, як розмір і кількість кімнат, що дозволяє адаптувати рівні до різних рівнів складності або стилю гри. По-третє, такий підхід сприяє візуальному різноманіттю, оскільки кожна кімната може мати унікальний дизайн або такі елементи, як вороги, бонуси чи пастки. Нарешті, метод легко розширити: його можна легко доповнити новою механікою або типами об’єктів, не порушуючи загальну структуру рівня.

Для 2.5D-платформерів просторовий розподіл є ідеальним вибором, коли потрібна чітка структура з багатьма кімнатами, переходами та зонами для гри. Такий підхід забезпечує варіативність рівнів при збереженні логіки їх оформлення, що важливо для створення цікавого та збалансованого ігрового процесу.

1.2.9 Моделі на основі шаблонів (Pattern-based Models)

Моделі на основі шаблонів - популярний підхід до створення рівнів у 2.5D, заснований на використанні створених шаблонів (патернів). Цей спосіб дозволяє легко контролювати структуру рівнів, зберігаючи їх цілісність, логічність і естетичну привабливість. Розробники платформи створюють набір готових модулів - розділів рівнів, які можуть плоти, перешкоди, вороги, бонуси та декоративні елементи. При генерації такі модулі об'єднуються, створюючи повноцінну ігрову карту.

Основні переваги шаблонних моделей - це проста реалізація та передбачуваність результатів. Рівні, створені за допомогою цього методу, мають

чітку структуру, хоча кожен шаблон попередньо перевірено на придатність до гри та баланс. Це дозволяє уникнути ситуацій, коли рівень стає непрохідним або нецікавим. Наприклад, шаблон може включати платформу з пасткою, яка вимагає від гравця певних навичок, або зону з бонусами, які стимулюють дослідження.

Шаблони можуть бути різних типів. Наприклад, вони можуть включати стандартні платформи, спеціальні частини з рухомими платформами, складні лабіринти або вертикальні секції для більш динамічного ігрового процесу. Деякі шаблони можна адаптувати для конкретних подій або труднощів: наприклад, більш складні ділянки для просунутих гравців або зони відпочинку з бонусами після напруженої битви. Усі ці модулі об’єднані в єдину карту за певними визначеними правилами.

Процес розміщення шаблонів може бути як лінійним, так і випадковим. При лінійному підході модулі розміщуються в певному порядку, що забезпечує поступове зростання складності та послідовності сюжету. У випадковому підході модулі вибираються випадковим чином, що дозволяє створювати різноманітні та унікальні рівні. Платформери 2.5D часто використовують комбінацію цих підходів: базова структура визначається заздалегідь, але деталі додаються випадковим чином.

Крім того, моделі на основі шаблонів дозволяють легко масштабувати рівні. Розробники можуть створити набір невеликих базових модулів і використовувати їх для формування як коротких, так і довгих рівнів. Це особливо корисно для процедурно згенерованих ігор, де кожен новий рівень має бути унікальним, але залишатися логічним і гармонійним. Щоб підтримувати різноманітність, ви можете створювати різні набори шаблонів для окремих біомів або тематичних областей.

Ще одна перевага цього методу полягає у можливості створення інтерактивних елементів у шаблонах. Наприклад, у модулі може бути інтегрований механізм, який змінює структуру рівня, коли гравець натискає важіль або проходить через тригер. Це додає геймплею динаміки та непередбачуваності, роблячи кожне проходження рівня унікальним.

Однак використання шаблонів також має свої виклики. Основною проблемою є ризик повторюваності, коли гравець помічає однакові ділянки в різних рівнях. Це може зробити гру менш цікавою. Щоб уникнути цього, розробники створюють велику кількість різноманітних шаблонів і застосовують механізми випадкового змішування. Додатково можна вбудовувати в модулі варіації елементів, наприклад, змінювати розташування ворогів чи бонусів.

Загалом, моделі на основі шаблонів - це простий і ефективний підхід до створення рівнів для платформерів 2.5D. Вони забезпечують контроль над структурою рівнів, спрощують процес розробки та дозволяють створювати як статичні, так і процедурно згенеровані карти. Завдяки цьому метод залишається одним із найпоширеніших підходів у дизайні ігор, особливо для проєктів із великим акцентом на якість ігрового процесу.

1.2.10 Генерація на основі графів (Graph-based Generation)

Генерація на основі графів - це процедурний метод генерації рівня, який використовує структуру графа для визначення зв’язків між елементами на рівні. У контексті 2.5D-платформерів граф - це сукупність вузлів (вершини) і ребер (з’єднання між вузлами), що дозволяє чітко формалізувати архітектуру рівня. Кожен вузол може відповідати певному елементу, наприклад платформі, області з ворогами, головоломці або точці збору бонусів, тоді як ребра визначають шлях, яким гравець може пройти між цими елементами.

Цей метод забезпечує високий рівень контролю над структурою рівнів. Дизайнери можуть вказати певні правила для створення графіків, дозволяючи створювати рівні з логічною послідовністю. Наприклад, вузол може вказувати на платформу, яку необхідно перетнути, перш ніж досягти наступного вузла, який представляє пастку або бонусну зону. Грані визначають доступність між вузлами, наприклад шлях через тунель або необхідність переходу з однієї платформи на іншу.

Одна з ключових переваг генерації на основі графів - це можливість планування ігрового процесу. Наприклад, гравець може почати в стартовому

вузлі, поступово рухаючись до вузла-фінішу через серію проміжних випробувань. Ця структура дозволяє створювати рівні з різними шляхами проходження. Графи можуть включати розгалуження, що пропонують альтернативні маршрути, наприклад, легкий шлях із меншими винагородами або складний, який потребує більше зусиль, але обіцяє цінніші бонуси.

Генерація рівнів за допомогою графів також дозволяє легко впроваджувати тематичні зони чи біоми. Вузли графа можуть бути класифіковані за певними типами зон, наприклад, ліс, печера, чи механічний завод. Це дозволяє автоматично змінювати типи платформ, ворогів або перешкод у відповідності до тематики, додаючи рівням більше різноманіття та глибини.

Ще однією перевагою цього методу є його гнучкість у створенні складних інтерактивних систем. Наприклад, вузол може представляти механізм, який відкриває доступ до іншої частини рівня після виконання певної дії, наприклад, активації важеля. Такі елементи сприяють створенню нелінійного геймплею, що робить гру цікавішою для гравців. Додатково, графи можна доповнити вагами ребер, які представляють складність або вартість проходження певного маршруту, дозволяючи динамічно змінювати складність рівнів.

Процес генерації рівнів за допомогою графів зазвичай складається з кількох етапів. Спочатку створюється граф - структура з вузлів і ребер. Потім кожен вузол наповнюється відповідним контентом, наприклад, платформою, ворогами або бонусами. Після цього весь граф інтегрується у віртуальний простір, де вузли перетворюються на фізичні елементи рівня. Для забезпечення плавного переходу між вузлами можуть використовуватися спеціальні алгоритми, які визначають оптимальне розташування платформ або з'єднань.

Незважаючи на свої переваги, метод генерації на основі графів також має певні труднощі. Основна проблема полягає в тому, щоб правильно збалансувати графік, щоб уникнути надто простих або надто складних рівнів. Крім того, реалізація цього підходу може вимагати глибоких знань алгоритмів графів, що ускладнює його використання новачкам у розробці ігор. У той же час для

досвідчених розробників цей спосіб відкриває величезні можливості для створення унікальних, логічних і цікавих рівнів.

Загалом, генерація на основі графіків є потужним інструментом для створення складних і різноманітних рівнів у 2.5D платформерах. Він забезпечує чітку структуру, дозволяє ввести нелінійність і забезпечує контроль над послідовністю ігрового процесу. Завдяки своїй гнучкості цей підхід можна використовувати як у статичній, так і в процедурній генерації, що робить його універсальним вибором для багатьох проектів.

1.2.11 Генетичні алгоритми

Генетичні алгоритми - це методи оптимізації, які імітують процес природного відбору. У контексті 2.5D-платформерів цей підхід можна використовувати для процедурної генерації рівнів, які відповідають певним критеріям, таким як складність, баланс або задоволення від гри. Генетичні алгоритми працюють ефективно там, де існує велика кількість можливих конфігурацій рівнів і потрібно вибрати найкращі. Основна ідея генетичних алгоритмів полягає у створенні «популяції» вихідних рішень, кожне з яких представляє потенційний рівень. Кожна «особина» в цій популяції кодується як «геном», який може бути, наприклад, масивом чисел або символів. У випадку з платформером це може бути послідовність даних, що описують розташування платформ, ворогів, перешкоди та бонуси. Алгоритм використовує еволюційні принципи, такі як схрещування, мутація та відбір, щоб поступово покращувати популяцію.

Процес роботи генетичного алгоритму починається з генерації випадкової сукупності рівнів. Кожен рівень оцінюється за допомогою функції відповідності, яка визначає, наскільки рівень відповідає заданим критеріям. Наприклад, рівень може отримати високий бал, якщо він має збалансовану складність, забезпечує цікавий ігровий процес і містить достатньо бонусів і викликів. Фітнес-функція може враховувати різноманітні фактори, такі як кількість платформ, відстань між ними, кількість ворогів, розташування пасток тощо.

За результатами оцінювання відбираються найкращі рівні (особини) для створення наступного покоління. Цей процес передбачає схрещування, під час якого геноми двох батьків поєднуються для створення нових рівнів. Наприклад, частину платформи з одного рівня можна поєднати з іншим рівнем, утворюючи нову структуру. Також додається елемент мутації, коли в геном вносяться невеликі зміни, щоб забезпечити різноманітність і уникнути застрягання в локальному мінімумі. Мутація може включати, наприклад, зміну висоти платформи або додавання нового ворога.

Процес еволюції повторюється, доки не буде досягнуто поставленої мети, наприклад, створення рівня з ідеальною складністю або певної структури. Після завершення роботи алгоритму результатом є рівень або набір рівнів, які найкраще відповідають заданим критеріям.

Генетичні алгоритми мають кілька переваг для створення рівнів у 2.5D платформерах. По-перше, вони дозволяють автоматично знаходити оптимальні конфігурації навіть у складних середовищах, де інші методи неефективні. По-друге, такий підхід забезпечує гнучкість: розробники можуть налаштувати адаптивну функцію та інші параметри алгоритму відповідно до специфіки гри. Наприклад, можна створювати рівні, складність яких поступово зростає, що важливо для ігор з нелінійним прогресом.

Ще одна перевага генетичних алгоритмів полягає в тому, що вони можуть генерувати унікальні рівні під час кожного запуску алгоритму. Це особливо корисно для ігор, які покладаються на процедурну генерацію для створення нескінченних або постійно мінливих рівнів. Крім того, цей підхід може враховувати досвід гравця, адаптуючи рівні до індивідуальних уподобань або навичок гравця.

Однак генетичні алгоритми також мають певні проблеми. Основна проблема полягає у виборі правильної фітнес-функції, яка точно оцінює якість рівнів. Якщо функція не продумана, алгоритм може генерувати рівні, які технічно відповідають критеріям, але нецікаві чи непридатні для гри. Крім того, цей метод

може бути ресурсомістким, оскільки вимагає багатьох поколінь обчислень, особливо якщо кількість елементів на рівні дуже велика.

Підсумовуючи, генетичні алгоритми є потужним інструментом для створення процедурно згенерованих рівнів у 2.5D платформерах. Вони забезпечують баланс між автоматизацією та контролем, дозволяючи створювати цікаві, збалансовані та різноманітні рівні. Завдяки їх здатності адаптуватися до різних умов і вимог, цей метод ідеально підходить для сучасних ігор, які вимагають динамічної та непередбачуваної генерації контенту.

1.2.12 Wave Function Collapse (WFC)

Колапс хвильової функції (WFC) — це процедурний алгоритм генерації вмісту, який використовує принципи теорії інформації та подібний до того, як обмеження працюють у математиці. Метод набув популярності завдяки здатності створювати складні цілісні структури на основі набору вхідних даних. У платформерах 2.5D WFC дозволяє створювати рівні з логічною структурою та естетичною цілісністю, автоматично враховуючи обмеження, встановлені розробником.

За своєю суттю WFC працює за принципом «колапсу хвильової функції». Кожна точка (або «плитка») у просторі генерації спочатку має набір можливих станів. Ці стани відповідають попередньо визначеним елементам, таким як платформи, перешкоди, порожні області або декоративні об’єкти. Алгоритм поступово зменшує можливості для кожної плитки, враховуючи такі обмеження, як суміжність певних елементів, доки не буде досягнуто унікального результату.

Перевагою WFC у 2.5D-платформерах є його здатність створювати послідовні та цікаві рівні без необхідності ручного втручання. Наприклад, алгоритм може стежити за тим, щоб вороги завжди з'являлися на платформі, а не в повітрі, або щоб бонусні об'єкти розміщувалися у важкодоступних місцях. Це досягається за допомогою спеціальних правил сумісності, які визначають, які елементи можуть бути розташовані поруч. Такі правила можуть враховувати як логіку гри, так і дизайнерські рішення.

Процес WFC починається зі створення набору вхідних шаблонів (шаблонів), які відображають бажану структуру рівня. Наприклад, в платформері це можуть бути фрагменти рівнів з платформами, сходами, перешкодами і бонусами. Алгоритм аналізує ці шаблони, визначає можливі комбінації елементів і генерує на їх основі карту рівнів. Важливо, що WFC враховує не лише локальні зв’язки, а й загальну структуру, забезпечуючи послідовність і гармонію.

Алгоритм WFC відмінно підходить для створення динамічних і унікальних рівнів. У процедурній генерації це дозволяє створювати нескінченну кількість варіацій рівнів за допомогою одного набору шаблонів, що значно скорочує час розробки. Крім того, завдяки своїй здатності враховувати складні правила, WFC може надавати різноманітні ігрові ситуації, зберігаючи при цьому баланс ігрових можливостей.

Використання WFC у платформерах 2.5D також дозволяє інтегрувати різні тематичні зони чи біоми на одному рівні. Наприклад, алгоритм може динамічно змінювати візерунки в залежності від зони, переміщаючись від лісу до печери або механічної фабрики, зберігаючи плавний перехід між ними. Це додає ігровому процесу глибини та різноманітності.

Однією з ключових переваг WFC є його здатність створювати складні та узгоджені рівні навіть із невеликим набором вхідних даних. Алгоритм ефективно використовує інформацію із зразків, поширюючи її на весь рівень. Це особливо корисно для невеликих інді-команд, які хочуть створювати вражаючий вміст з мінімальними ресурсами. У той же час розробники можуть легко налаштувати алгоритм, змінивши набір шаблонів або правила сумісності для досягнення бажаних результатів.

Незважаючи на свої переваги, WFC також має певні проблеми. Основною проблемою є ризик того, що алгоритм застрягне, якщо він не зможе знайти рішення через конфліктні обмеження. Щоб уникнути цього, розробники повинні ретельно тестувати та налаштовувати вхідні дані та правила. Цей метод також може бути обчислювально дорогим для великих рівнів із великою кількістю

елементів, хоча оптимізація алгоритму може значно зменшити споживання ресурсів.

Підсумовуючи, Wave Function Collapse є потужним інструментом для створення процедурно згенерованих рівнів у 2.5D платформерах. Його здатність автоматично враховувати складні обмеження та створювати естетично гармонійні структури робить його ідеальним для сучасних ігор. За допомогою WFC можна створювати як прості, так і складні рівні, зберігаючи баланс між випадковістю і керованістю, що дозволяє значно підвищити якість ігрового процесу.

1.3 Загальні принципи генерації ігрових рівнів

Загальні принципи створення ігрових рівнів є важливою складовою для створення захоплюючого та непередбачуваного ігрового досвіду. Перш за все, баланс між викликом і винагородою є ключовим аспектом у процесі розвитку рівня. Гравець повинен відчувати, що кожне завдання варте його зусиль і часу, отримуючи адекватну винагороду у вигляді нових можливостей або задоволення від перемоги. Прогресуюча складність рівнів дозволяє гравцеві поступово розвивати свої навички та адаптуватися до нових викликів гри. Починаючи з простих завдань, гравець поступово переходить до більш складних, відчуваючи при цьому постійний прогрес у своєму розвитку.

Унікальність і різноманітність рівнів роблять гру більш цікавою і запам'ятовується гравцеві. Кожен рівень повинен мати свою індивідуальність і унікальні елементи, які роблять його особливим. Використання власних можливостей гри дозволяє максимально використовувати унікальні механіки та особливості гри. Рівні створені таким чином, щоб гравець відчував причетність до використання всіх доступних можливостей.

2 АНАЛІЗ МОДЕЛЕЙ ТА МЕТОДІВ ВИПАДКОВОЇ ГЕНЕРАЦІЇ РІВНІВ ДЛЯ ГРИ В ЖАНРІ ПЛАТФОРМЕР 2.5D

2.1 Аналіз та постановка проблеми існуючих методів для вирішення поставленої задачі

Для розробки власного методу генерації рівнів для гри в жанрі 2.5D платформера було проаналізовано різні підходи до автоматизації та генерації окремих компонентів алгоритму побудови рівнів. Після вивчення методів генерації карт можна скласти таблицю з перевагами та недоліками кожного з розглянутих підходів.

Таблиця 2.1

Порівняльна характеристика методів генерації мапи для рівня

Метод	Опис	Переваги	Недоліки
Ручне створення рівнів	Рівні створюються вручну дизайнером.	1. Точний контроль 2. Високоякісний дизайн.	1. Трудомісткий процес. 2. Не підходить для великих ігор.
Генерація випадкових рівнів	Рівні створюються за допомогою випадкових чисел.	1. Легко реалізувати 2. Різноманітність рівнів.	1. Ризик створення незбалансованих або незручних рівнів.

Продовження таблиці 2.1

Порівняльна характеристика методів генерації мапи для рівня

Метод	Опис	Переваги	Недоліки
Комбінований підхід	Поєднання ручного дизайну та процедурної генерації.	1. Баланс між контролем і автоматизацією. 2. Творчий потенціал.	1. Більш складна реалізація.
Генетичні алгоритми	Використання еволюційного підходу для відбору найкращих варіантів рівнів.	1. Може створювати унікальні й оптимальні рівні.	1. Вимагає багато ресурсів і часу на обчислення.
Моделі на основі шаблонів	Генерація рівнів на основі заздалегідь створених блоків або шаблонів.	1. Контрольовані результати 2. Різноманітність через комбінації.	1. Може стати повторюваним при обмеженій кількості шаблонів.
Розроблений метод	Процедурна генерація платформ, бонусів та ворогів із дотриманням відстаней і випадковим вибором об'єктів.	1. Простота реалізації. 2. Різноманітність рівнів	1. Складно досягти ідеального балансу між елементами.

При створенні розробленого методу генерації рівнів для ігор в жанрі 2.5D Platformer важливо враховувати не тільки технічні аспекти реалізації, а й особливості ігрового процесу, який повинен бути цікавим, складним і збалансованим для гравця. . Важливим моментом є створення ігрових рівнів, які не тільки забезпечують різноманітність, але й зберігають логічну цілісність і структурованість, дозволяючи гравцеві відчувати, що він прогресує в грі, а не блукає в хаосі випадкових елементів. При цьому гравця чекають не тільки стандартні платформи, але й унікальні ситуації, що додають цікавості та непередбачуваності. Розробка такого методу вимагає поєднання кількох факторів, включаючи випадковість, елементи шаблону та структуровані правила, які дозволяють отримати бажаний баланс.

Розроблений метод, який поєднує в собі випадковість і логіку, ідеально підходить для генерації рівнів 2.5D Platformer. Використання випадковості дозволяє створювати нові, унікальні рівні під час кожного проходження гри, що підвищує повторність і інтерес гравців. Кожен рівень, створений за розробленою методикою, не буде виглядати ідентично попередньому, що зберігає відчуття новизни та захоплює гравця. При цьому метод включає структурні елементи, що визначають базову архітектуру рівня - це дозволяє зберегти логіку і послідовність розміщення об'єктів, таких як платформи, бонуси, вороги і пастки. Тому, хоча рівень змінюється в результаті випадковості, він завжди має певну структуру, яка дає гравцеві відчуття контролю та ясності.

Однією з головних переваг розробленого методу є гнучкість налаштувань. Розробники можуть варіювати частоту появи різних елементів на рівнях, що дозволяє точніше налаштовувати складність і динаміку гри. Це особливо важливо для створення рівнів, які повинні задовольнити різні типи гравців: від початківців до більш досвідчених. Наприклад, розробник може зменшити кількість ворогів або додати більше бонусів для початкових рівнів, або збільшити кількість пасток і складність механіки на наступних етапах гри. Всі ці налаштування дозволяють досягти потрібного рівня варіативності, що робить гру цікавою і захоплюючою.

Розроблений метод також добре адаптується та добре інтегрується з популярними ігровими движками, такими як Unity, що забезпечує легкість впровадження та ефективність. Оскільки Unity підтримує широкий спектр інструментів для роботи з 2D і 3D графікою, метод легко інтегрується в будь-який проект, надаючи розробникам зручні інтерфейси для створення рівнів. Крім того, розроблений метод дозволяє контролювати не тільки розміщення елементів, а й їх взаємодію з іншими об’єктами, наприклад, фізичні властивості платформ і гравця. Це дозволяє досягти тонкого балансу між складністю та можливістю проходження рівня, що критично важливо для підтримки інтересу гравця.

Порівняно з іншими методами генерації рівнів розроблений метод має ряд суттєвих переваг. Методи, засновані на клітинних автоматах або шумах, можуть бути занадто складними для реалізації, що знижує їх ефективність у певних проектах. Крім того, вони можуть створювати надто хаотичні рівні або нездатні досягти бажаного рівня структури. У той же час поєднання випадкових елементів і шаблонних правил у розробленому методі дозволяє досягти чудового балансу між непередбачуваністю та логікою, створюючи цікаві та зрозумілі для гравця рівні. Цей підхід також дозволяє зменшити кількість непередбачуваних ситуацій, які можуть негативно вплинути на гру, і робить рівні більш адаптованими до різних ігрових ситуацій і типів гравців.

Загалом розроблений метод є ефективним засобом для досягнення бажаного рівня складності, варіативності та інтересу, що підвищує відтворюваність гри. Він поєднує в собі оптимальні аспекти випадковості, логіки та настроюваних елементів, що дозволяє створювати рівні, які відповідають вимогам жанру та інтересам гравців. Використання цього методу дозволяє розробникам ефективно та зручно створювати нові рівні, не втрачаючи контролю над процесом гри.

2.2 Дослідження можливостей удосконалення генерації рівнів у 2.5D платформерах

Розвиток генерації рівнів для ігор 2.5D Platformer відкриває багато можливостей для покращення як процесу створення вмісту, так і самого ігрового досвіду. Одним із шляхів є вдосконалення алгоритмів генерації, зокрема шляхом аналізу успішних рівнів. Це дозволяє створювати нові карти, які відповідають заданим критеріям якості, зберігаючи інтригу та цікавий ігровий процес. Алгоритми, засновані на методах машинного навчання або оптимізації, можуть автоматично адаптувати рівні з урахуванням попереднього досвіду гравців і навіть змінювати конфігурацію рівнів в реальному часі в залежності від їх гри.

Ще одним перспективним напрямком є впровадження процедурної генерації, яка може забезпечити унікальність локацій в межах одного рівня. Використання шумових функцій або графічних структур для створення унікальних ландшафтів і розташування об’єктів дозволяє кожному запуску гри пропонувати новий досвід. Це підвищує інтерес до гри і спонукає гравців повертатися до неї знову і знову, оскільки кожен рівень стає своєрідним викликом з новими можливостями для дослідження.

Також важливо створити гнучку систему генерації об'єктів, яка включає не тільки платформи, але і ворогів, бонуси та інші елементи. Така система дозволяє створювати рівні, які поєднують різноманітні механіки та викликають у гравців активну взаємодію. Він надає можливість змінювати параметри генерації в залежності від складності рівня або стилю гри, що дозволяє підтримувати інтерес гравців на різних етапах гри.

Розробка інтуїтивно зрозумілих інструментів для налаштування генерації рівнів також може значно спростити роботу розробників. Завдяки зручним графічним інтерфейсам навіть новачки зможуть експериментувати з різними параметрами генерації, створюючи рівні, які відповідають певним вимогам або стилям гри. Це знижує бар’єр входження в процес розробки ігор і дає можливість ширше залучати до творчого процесу широке коло користувачів.

Не менш важливим є тестування згенерованих рівнів, що дозволяє оцінити баланс складності та виявити потенційні проблеми, наприклад, надто складні чи, навпаки, надто легкі області. Інструменти автоматизованого тестування можуть допомогти розробникам виявити слабкі місця в рівнях і зробити гру більш збалансованою. Це дозволяє створювати захоплюючі та приємні для гравців рівні, зберігаючи при цьому оптимальний рівень складності.

Експерименти з естетикою та стилем рівнів також можуть значно покращити ігровий процес. Використання різних візуальних концепцій, таких як зміна колірної палітри або дизайну рівнів залежно від часу доби, сезону чи ходу гри, створює атмосферу, яка посилює занурення. Ця різноманітність дизайну рівнів робить гру більш захоплюючою та додає їй унікального персонажа.

Інтеграція нових ігрових механізмів, які заохочують гравців досліджувати, може зробити рівні ще цікавішими. Додавання прихованих шляхів, динамічних елементів або нових об’єктів для взаємодії може зробити кожен рівень унікальним. Це також може збільшити відтворюваність гри, дозволяючи гравцям досліджувати нові механіки чи секрети з кожним проходженням.

Таким чином, подальший розвиток генерації рівнів для ігор 2.5D Platformer може не тільки підвищити якість ігрового процесу, але й створити нові можливості для творчих підходів у розробці. Використання новітніх технологій і методів генерації може змінити спосіб створення рівнів, підвищуючи інтерес до гри та залучаючи гравців до довготривалої взаємодії з проектом.

2.3 Формування математичної моделі генерації випадкових рівнів для жанру Platformer 2.5D

Модель генерації рівнів у 2.5D-платформерах - це потужний інструмент, який дозволяє створити глибокий і насичений ігровий досвід для гравця. Якщо це тільки один рівень, його структура повинна бути розроблена таким чином, щоб кожен новий запуск гри забезпечував, а не передбачав середовище. Важливою частиною цього процесу є використання алгоритмів для генерації, які враховують

не тільки стандартні параметри, такі як висота та відстань, але й можливість інтегрувати правила для динамічних елементів, які змінюються під час проходження. Наприклад, спеціальні платформи, які можуть рухатися або змінювати форму, ставлять гравцям додатковий виклик і змушують шукати нові шляхи досягнення мети.

Крім того, важливою складовою є генерація ворогів, таких як ніндзя та вороги дальнього бою, поведінку яких можна адаптувати до різних умов рівня. Вони можуть по-різному взаємодіяти з ігровим середовищем, можуть пересуватися по стінах, телепортуватися або використовувати, наприклад, інші спеціальні механізми, що робить їх дійсно небезпечними і додає елемент непередбачуваності. Ці характеристики роблять кожну зустріч з ворогами випробуванням не тільки швидкості реакції, а й стратегії. Гравець змушений називати не тільки розташування ворогів, але і те, як вони можуть змінити свою позицію або поведінку в залежності від того, як змінюється рівень.

Розподіл бонусів і монет є важливою частиною ігрового процесу, яка зменшує стимул для гравців досліджувати рівень і створює додаткові цілі, які сприяють більшій залученості в гру. Бонуси можна розміщувати в місцях, де вони вимагають не тільки обережності, але й певних навичок для отримання, наприклад, проходження небезпечних зон або використання динамічної платформи. Це також дозволяє отримати більше можливостей для проходження рівня, деякі бонуси можуть надавати тимчасові послуги, які відкривають нові можливості для пересування або боротьби з ворогами. Монети, розміщені в стратегічних точках, можуть спонукати гравців ризикувати та шукати альтернативні шляхи, хоча їх можна отримати, додаючи перешкоди або вибираючи складніші маршрути. Вони також можуть впливати на оцінку рівня або визначати додаткові винагороди для гравця, що додає елемент змагання, особливо з точки зору підтримки таблиць лідерів або високих результатів.

Важливою частиною цієї моделі є створення цілісної системи, яка збалансує всі елементи та забезпечує приємний і складний ігровий процес. Правильна комбінація платформ, ворогів, бонусів і монет дозволяє створити

захоплюючий рівень, який привертає увагу гравця і зберігає цікавість гри. Наприклад, рівень може мати кілька можливих шляхів, кожен зі своїми перевагами та недоліками. Гравець може вибрати безпечніший шлях, який включає більше платформ, або ризикнути, вибравши складніший маршрут із більшою кількістю бонусів і монет. Усі ці параметри підвищують рівень відтворюваності, дозволяючи решті гравців кожного разу вибирати різні стратегії проходження.

Збалансування рівня складності та різноманітності є критичним аспектом для підтримки інтересу гравців. Генерація рівня не повинна бути надто легкою чи надто складною. Це досягається шляхом поєднання різних типів платформ, які вимагають від гравців адаптації, і ворогів, які мають різну поведінку та рівні складності. Кожен новий підхід або новий тип ворога змушує гравця переглянути свою стратегію та підхід до рівня. Наприклад, введення ворогів, які можуть з'являтися на різних платформах або телепортуватися, змінює динаміку гри, і це робить гравця ще більш важливим для кожного кроку.

Ця модель генерації рівнів також дає розробникам багато свободи для творчості. Використовуючи різноманітні комбінації платформ, ворогів і бонусів, можна створювати безліч варіацій одного рівня, що дозволяє розробникам адаптувати гру під різні типи гравців. Деякі бачать кращу платформу і безпечніші маршрути, інші шукають можливості для отримання великої кількості бонусів і монет. Це забезпечує гнучкість в налаштуваннях ігрового процесу і дозволяє кожному гравцеві знайти власний підхід до проходження.

Як наслідок, ця модель генерації рівнів створює захоплюючий ігровий досвід, який постійно змінюється, і вимагає від гравця креативності та адаптації до нових умов, що змінюються. Різноманітність платформ, ворогів, бонусів і монет забезпечує постійну потребу в стратегії та розумінні, роблячи кожен новий запуск гри цікавим і непередбачуваним. Це значно підвищує реіграбельність гри та стимулює довготривалий інтерес гравців до рівня.

2.3.1 Структура генерації рівня

Модель задає чіткі правила для генерації елементів ігрового середовища, таких як платформи, вороги (ніндзя та рейндж єнемі), бонуси та монети. Це дозволяє створювати інтерактивні рівні, що забезпечують цікавий і унікальний ігровий досвід. Кожен елемент рівня генерується з урахуванням визначених параметрів, які формують баланс між складністю і захопливістю гри.

Основні етапи генерації рівня:

- Визначення кількості платформ, необхідних для створення ігрового маршруту;
- Генерація початкових координат для перших платформ, які служать основою для побудови решти рівня;
- Визначення позицій платформ із врахуванням умов доступності, висоти стрибка персонажа та відстані між елементами;
- Розташування ворогів, бонусів і монет на платформах за допомогою алгоритмів, що використовують випадкові ймовірності.

Алгоритм генерації метода в грі:

1. Ініціаліазація параметрів рівня;
2. Генерація платформ
    - Кількість платформ визначається випадковим чином у заданому діапазоні;
    - Генерація початкових координат: першій платформі задається фіксована позиція, яка є стартовою точкою для гравця;
    - Визначення позицій для наступних платформ:
        1. Враховується висота стрибка персонажа для забезпечення доступності;
        2. Уникається перекриття платформ або їх занадто тісне розташування;

3. Позиції платформ формуються так, щоб створити різноманітні маршрути.

3. Генерація ворогів
   - Визначаються типи ворогів:
     1. Ніндзя розміщуються на платформах, які передбачають швидке пересування та маневри;
     2. Рейндж єнемі з’являються у відкритих ділянках, де вони можуть використовувати дальню атаку;
   - Враховуються умови безпеки: стартова платформа залишається вільною від ворогів.
4. Генерація бонусів:
   - Ймовірність появи бонусів задається як низька, щоб зробити їх рідкісними і цінними;
   - Монети можуть бути розташовані у важкодоступних місцях, стимулюючи гравця до дослідження рівня.
5. Фіналізація рівня
   - Усі об'єкти (платформи, вороги, бонуси) перевіряються на унікальність їхніх позицій, щоб уникнути перекриття;
   - Підтверджується, що рівень відповідає вимогам доступності та балансу складності.

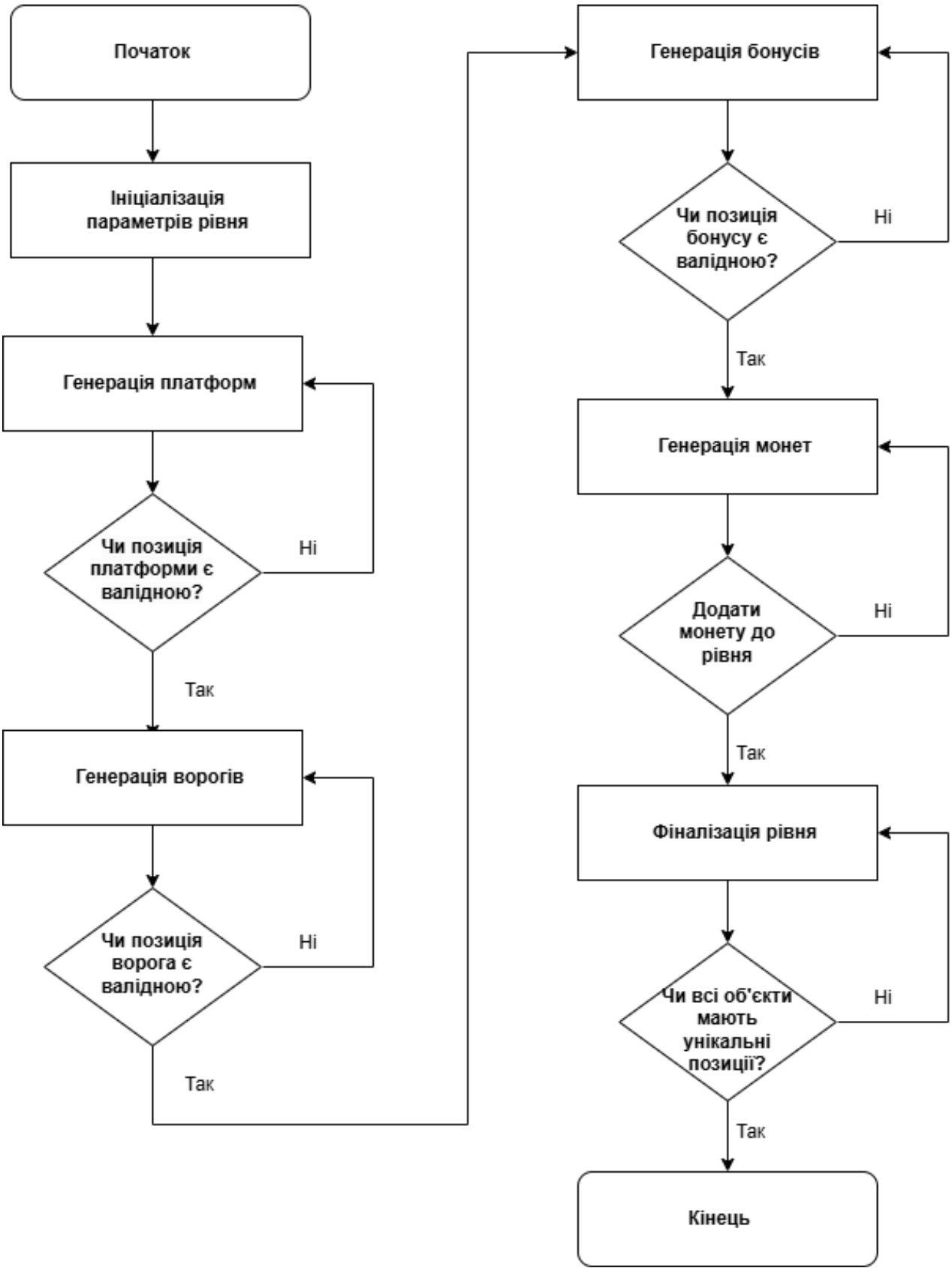


Рис. 2.1 Алгоритм генерації метода в грі

2.3.2 Формування математичної моделі створення випадкових рівнів для гри жанру Platformer 2.5D

Розробка математичної моделі створення випадкових рівнів для гри в жанрі Platformer 2.5D передбачає врахування особливостей цього жанру. Серед них — забезпечення структурної цілісності рівня, баланс складності, а також уникнення конфліктів між розташуванням ігрових елементів (платформ, ворогів, бонусів, монет тощо). Модель має враховувати як елементи випадковості, так і визначені закономірності, що забезпечують ігрову динаміку.

Кількість платформ також генерується в межах визначеного діапазону $[p_{min}, p_{max}]$, що задається параметрами рівня. Це дозволяє створювати рівні з різною структурною складністю та доступністю для гравця:

$$p_{count} = R[p_{min}, p_{max}] \quad (2.1)$$

де:

- p_{count} - кількість платформ, які будуть створені для рівня;
- $R[p_{min}, p_{max}]$ - функція випадкового вибору в межах діапазону;
- p_{min} - мінімальна кількість платформ;
- p_{max} - максимальна кількість платформ.

Кількість ворогів на рівні визначається випадковим вибором у межах діапазону $[e_{min}, e_{max}]$. Ця формула дозволяє регулювати складність рівня, додаючи різну кількість ворогів:

$$e_{count} = R[e_{min}, e_{max}] \quad (2.2)$$

де:

- e_{count} - кількість ворогів на рівні;
- $R[e_{min}, e_{max}]$ - функція випадкового вибору з діапазону $[e_{min}, e_{max}]$;
- e_{min} - мінімальна кількість ворогів;
- e_{max} - максимальна кількість ворогів.

Кількість бонусів на рівні визначається випадковим чином у межах заданого діапазону $[b_{min}, b_{max}]$. Ця формула дозволяє контролювати загальну кількість корисних елементів на рівні, забезпечуючи як випадковість, так і відповідність геймплейним вимогам:

$$b_{count} = R[b_{min}, b_{max}] \tag{2.3}$$

де:

- b_{count} - кількість бонусів, які будуть згенеровані на рівні;
- $R[b_{min}, b_{max}]$ - функція випадкового вибору з діапазону $[b_{min}, b_{max}]$;
- b_{min} - мінімальна кількість бонусів;
- b_{max} - максимальна кількість бонусів.

Перевірка валідності позиції здійснюється для забезпечення правильного розташування об’єктів у просторі рівня:

$$V_{valid} = \begin{cases} True, \text{ якщо } |p - p_{occupied}| \geq d_{min} \; \forall \; p_{occupied} \wedge |p - p_{player}| \geq r_{avoid} \\ False, \text{ якщо не виконується} \end{cases} \tag{2.4}$$

де:

- V_{valid} - результат перевірки валідності позиції (істина або хибність);
- p - поточна згенерована позиція;
- $p_{occupied}$ - множина вже зайнятих позицій;
- d_{min} - мінімальна допустима відстань між p та $p_{occupied}$;
- p_{player} - початкова позиція гравця;
- r_{avoid} - радіус уникнення гравця.

Запропонована модель дозволяє створювати унікальні рівні, забезпечуючи баланс між випадковістю та логікою. Вона легко адаптується під вимоги розробників і мінімізує ризики генерації непридатних до гри рівнів.

2.3.3 Вимоги до коректності генерації рівня

Усі елементи рівня, включаючи платформи, ворогів, бонуси та монети, повинні мати унікальні координати, щоб запобігти виникненню колізій чи накладань. Це дозволяє уникнути ситуацій, коли елементи гри перекривають одне одного, порушуючи логіку ігрового процесу або створюючи технічні проблеми. Кожен об’єкт має чітке місце на карті, що сприяє впорядкованості рівня та комфортному проходженню.

Кожен елемент рівня повинен бути розташований так, щоб гравець міг до нього дістатися, враховуючи фізичні можливості персонажа, як-от висота стрибка чи дальність руху. Недоступність ключових елементів, таких як платформи чи бонуси, може зробити рівень непрохідним і знизити якість геймплею. Генерація враховує ці параметри, забезпечуючи коректне розташування всіх об’єктів на рівні. Розташування ворогів також має бути зваженим, особливо на початку рівня. Вороги не повинні знаходитися надто близько до стартової точки гравця, щоб забезпечити час на адаптацію та ознайомлення з рівнем. Це створює справедливі умови гри, дозволяючи гравцеві підготуватися до викликів, які чекають попереду.

Рівень повинен бути збалансованим щодо складності та підтримки. Генерація враховує співвідношення ворогів, пасток, бонусів і монет. Надто легкі рівні можуть зробити гру передбачуваною й нудною, тоді як надто складні викликають фрустрацію. Баланс між випробуваннями та нагородами підтримує інтерес гравця та додає динаміки. Кожен рівень має відповідати загальній концепції гри, забезпечуючи логічність і послідовність проходження. Початок, основний маршрут і фінішна точка повинні бути зрозумілими, щоб гравець чітко знав, у якому напрямку рухатися. Це сприяє створенню структурованого ігрового досвіду, який відповідає очікуванням гравця.

Дуже важливо уникати ситуацій, які роблять рівень непрохідним. Недосяжні платформи чи блокування маршрутів порушують геймплей. Алгоритми генерації перевіряють доступність усіх елементів і коректність їхнього розташування, забезпечуючи безперервність ігрового процесу. Дизайн

рівня повинен відповідати стилістичним і тематичним вимогам гри. Якщо гра відображає певну атмосферу, як-от підземелля чи космічний простір, елементи рівня повинні відповідати цій тематиці. Це покращує занурення гравця у світ гри, роблячи її більш цілісною та захопливою.

Кількість елементів на рівні, таких як платформи, вороги, бонуси та монети, має відповідати встановленим параметрам. Це дозволяє створити передбачувану складність і забезпечити різноманітність, необхідну для підтримки інтересу гравця. Правильно розподілені елементи роблять кожен рівень унікальним і добре продуманим. Після створення рівня проводиться його валідація, щоб виключити будь-які технічні помилки чи конфлікти. Це важливо для стабільної роботи гри, уникнення збоїв і перевищення ресурсних лімітів. Валідація гарантує, що рівень готовий до проходження гравцем без непередбачуваних проблем.

3 СТВОРЕННЯ МОДЕЛІ ВИПАДКОВОЇ ГЕНЕРАЦІЇ РІВНІВ ДЛЯ ГРИ В ЖАНРІ PLATFORMER 2.5D

3.1 Опис застосованих програмних інструментів

3.1.1 Unity

Розробка рівнів для ігор 2.5D Platformer здебільшого виконується за допомогою ігрового движка Unity, який є однією з найпопулярніших платформ серед розробників. Це пов'язано з кількома факторами. По-перше, Unity надає широкий набір інструментів, які дозволяють створювати інтерактивні 2.5D середовища, ідеальні для платформерів. Від підтримки анімації та фізичних ефектів до можливості інтеграції з іншими технологіями, Unity дозволяє зручно працювати з усіма аспектами розробки ігор.

Рис. 3.1. Unity

По-друге, Unity має величезну бібліотеку готових активів, що дозволяє значно прискорити процес розробки. Це особливо важливо для інді-розробників і невеликих студій, які не мають великих фінансових ресурсів для створення унікальних моделей і текстур.

Крім того, Unity безкоштовний для невеликих команд і незалежних розробників, що робить його доступним для широкого кола користувачів. Це дозволяє розробникам з високим рівнем досвіду створювати високоякісні ігри без необхідності вкладати значні кошти в програмне забезпечення. Це робить Unity

ідеальним вибором для багатьох проектів, яким потрібен ефективний і доступний інструмент для створення 2.5D платформерів.

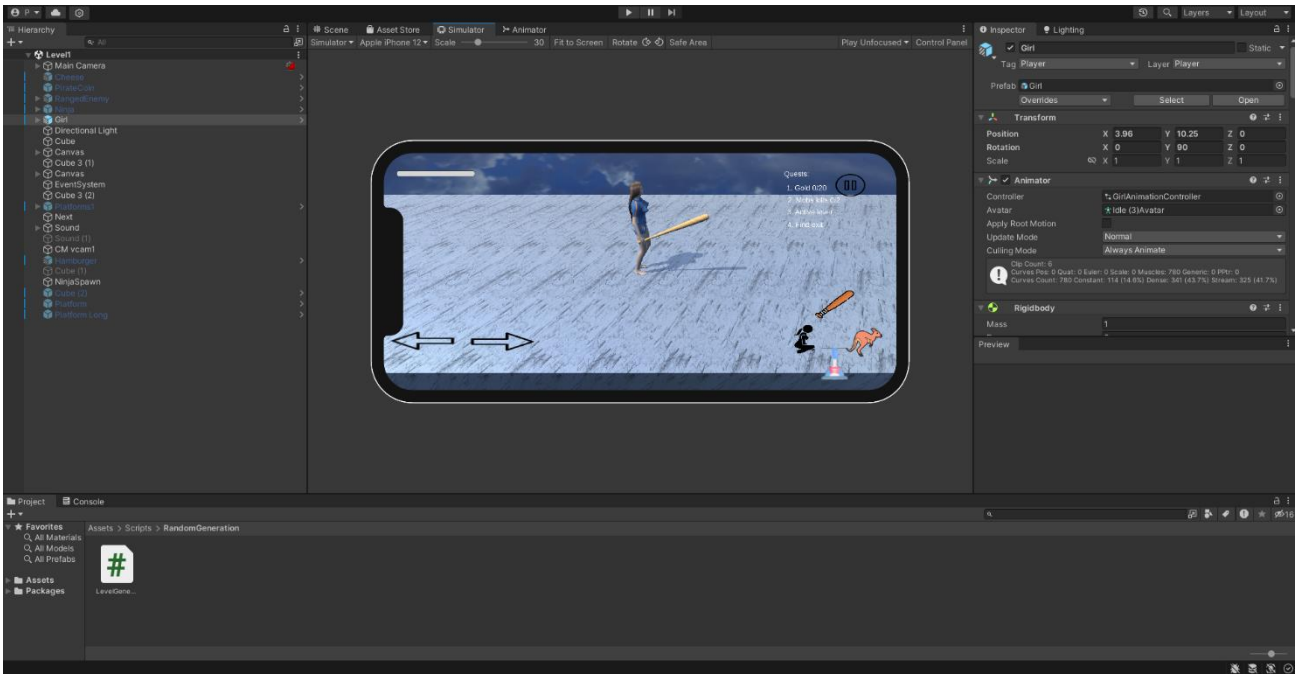


Рис. 3.2. Приклад інтерфейсу двигуна Unity

Unity - потужний і універсальний кросплатформовий ігровий двигун, який також виступає як інтегроване середовище розробки для створення інтерактивних 2D і 3D додатків. Спочатку розроблений для створення відеоігор, Unity тепер знайшов застосування у багатьох інших областях, таких як симуляції, навчальні програми, архітектурна візуалізація та інші візуальні інтерфейси. Його головна перевага – можливість працювати на широкому спектрі платформ, включаючи персональні комп'ютери, мобільні пристрої, ігрові консолі, системи віртуальної реальності та навіть веб-браузери.

Двигун Unity пропонує розробникам великий набір інструментів, які роблять процес створення ігор зручнішим та швидшим. Він оснащений потужним графічним двигуном, системами фізичного моделювання, звуковими інструментами та інструментами для роботи з анімацією. Всі ці можливості забезпечують високий рівень гнучкості під час роботи над різними типами проектів.

Однією з ключових переваг Unity є його кросплатформність. Завдяки цій можливості розробники можуть створювати програми, що працюють на різних операційних системах та пристроях, від мобільних телефонів та планшетів до складних VR-гарнітур. Це дозволяє суттєво розширити аудиторію користувачів та забезпечити доступність програм на різних платформах.

Ще однією важливою перевагою Unity є його модульність та розширюваність. Двигун дозволяє інтегрувати численні плагіни та модулі, які додають нові можливості або оптимізують робочі процеси. Крім того, завдяки вбудованій підтримці розширень та відкритості системи розробники можуть створювати власні інструменти для вирішення конкретних завдань.

Unity також має широкий функціонал для створення якісного контенту. Графічні інструменти дозволяють створювати реалістичні текстури, освітлення та візуальні ефекти, а штучний інтелект та фізичні системи забезпечують реалістичну поведінку персонажів та об'єктів у грі. Можливість гнучкої оптимізації програм під різні платформи гарантує стабільну роботу навіть на пристроях з низькою продуктивністю.

Двигун надає розробникам зручні інструменти для керування ресурсами, побудови сцен, написання ігрової логіки та організації взаємодії з користувачем. Все це значно прискорює процес створення ігор та мультимедійних програм, скорочуючи час і витрати на розробку.

Окремо варто відзначити Unity. Він безкоштовний для невеликих команд та інді-розробників, що дозволяє новачкам легко освоїти інструмент та створювати власні проекти без суттєвих фінансових вкладень. А інтуїтивно зрозумілий інтерфейс і велика база навчальних матеріалів роблять процес освоєння двигуна набагато простіше.

Unity також підтримує активну спільноту розробників, які діляться своїм досвідом, створюють корисні ресурси, скрипти та навчальні матеріали. Це сприяє постійному вдосконаленню інструментарію двигуна та полегшує процес вирішення складних завдань.

Завдяки всім цим характеристикам Unity є одним із найпопулярніших інструментів для розробки ігор та мультимедійних програм у світі. Його універсальність, багатофункціональність та зручність роблять його ідеальним вибором як для новачків, так і для досвідчених розробників.

3.1.2 Мова програмування C#

Було обрано мову програмування C#, оскільки він добре інтегрується з ігровим движком Unity, який є основною платформою для створення ігор. Unity дозволяє розробникам писати сценарії за допомогою C#, які дозволяють їм безпосередньо взаємодіяти з усіма аспектами гри: від створення ігрової механіки до керування фізикою, анімацією та штучним інтелектом противника. Це робить C# важливим інструментом розробки, який полегшує роботу з Unity API.

Рис. 3.3. Мова програмування C#

C# має ряд переваг, які роблять його ідеальним вибором для розробки ігор. Його строгий контроль типів зменшує кількість помилок під час компіляції, що особливо важливо при роботі над великими проектами. Мова підтримує об'єктно-орієнтоване програмування, що дозволяє легко структурувати код і забезпечує його масштабованість, необхідну при створенні складних ігрових рівнів.

Серед важливих особливостей C# варто виділити його підтримку асинхронного програмування через async/await, що дозволяє ефективно працювати з ігровими ресурсами, такими як: Наприклад, ви можете організувати завантаження текстур або обробляти ігрові події без шкоди для продуктивність. C# також підтримує управління пам'яттю за допомогою автоматичного збирання

сміття, що знижує ризик помилок, пов'язаних з некоректною роботою з ресурсами.

Ще однією вагомою причиною вибрати C# є його популярність і підтримка великих компаній, таких як Microsoft. Це дає вам доступ до великої кількості документації, навчальних матеріалів і зразків коду. Спільнота розробників C# є однією з найбільших, що дозволяє легко знайти відповіді на технічні питання або вирішити проблеми, які можуть виникнути під час роботи.

Крім того, C# характеризується кросплатформенністю завдяки фреймворку .NET, який дозволяє створювати ігри, які працюють на різних платформах, включаючи ПК, мобільні пристрої, консолі та навіть веб-браузери. Це відкриває широкі можливості для розширення та масштабування проекту та забезпечення максимальної аудиторії для гри.

Окрім технічних переваг, C# має інтуїтивно зрозумілий синтаксис, який робить його доступним для новачків, але водночас потужним для досвідчених розробників. Це означає, що ви можете швидше навчатися та ефективніше працювати над проектами. Мова також постійно оновлюється та вдосконалюється з додаванням нових функцій, що підвищує її актуальність у сучасному розвитку.

Усе це робить C# найкращим вибором для розробки ігор у жанрі платформи 2.5D, де важливо поєднати продуктивність, простоту, гнучкість і підтримку сучасних технологій.

3.1.3 Середовище розробки Rider

Вибір припав на середовище розробки Rider завдяки численним перевагам, які роблять його одним із найефективніших інструментів для створення ігор на базі Unity. Rider, розроблений компанією JetBrains, пропонує розробникам інтуїтивно зрозумілий інтерфейс, високу продуктивність і широкий набір функцій, необхідних для ефективної роботи з ігровими проектами.

Рис. 3.4. Середовище розробки Rider

Однією з головних причин вибору Rider є його глибока інтеграція з Unity. Середовище автоматично розпізнає проекти Unity та пропонує спеціальні інструменти для взаємодії з ігровим движком. Наприклад, він підтримує налагодження проектів Unity, дозволяє відстежувати поведінку скриптів у реальному часі та взаємодіяти з ігровими об’єктами прямо з IDE. Крім того, Rider автоматично синхронізується з Unity і відображає зміни в реальному часі, дозволяючи розробникам швидко тестувати свої рішення.

Редактор коду Rider є одним із найпотужніших доступних середовищ розробки. Він пропонує інтелектуальне автозавершення, аналізує код у реальному часі та визначає потенційні помилки або області для оптимізації. Це дозволяє уникнути багатьох проблем на етапі розробки та покращити якість написаного коду. Завдяки підтримці рефакторинга розробники можуть швидко змінювати структуру коду, не впливаючи на логіку програми.

Райдер також пропонує високу швидкість. Завдяки оптимізованому механізму індексування він ефективно працює навіть у великих проектах, швидко знаходячи потрібні класи, методи та змінні. Це особливо важливо при розробці ігор, де кількість коду часто велика, а час на виконання завдань обмежений.

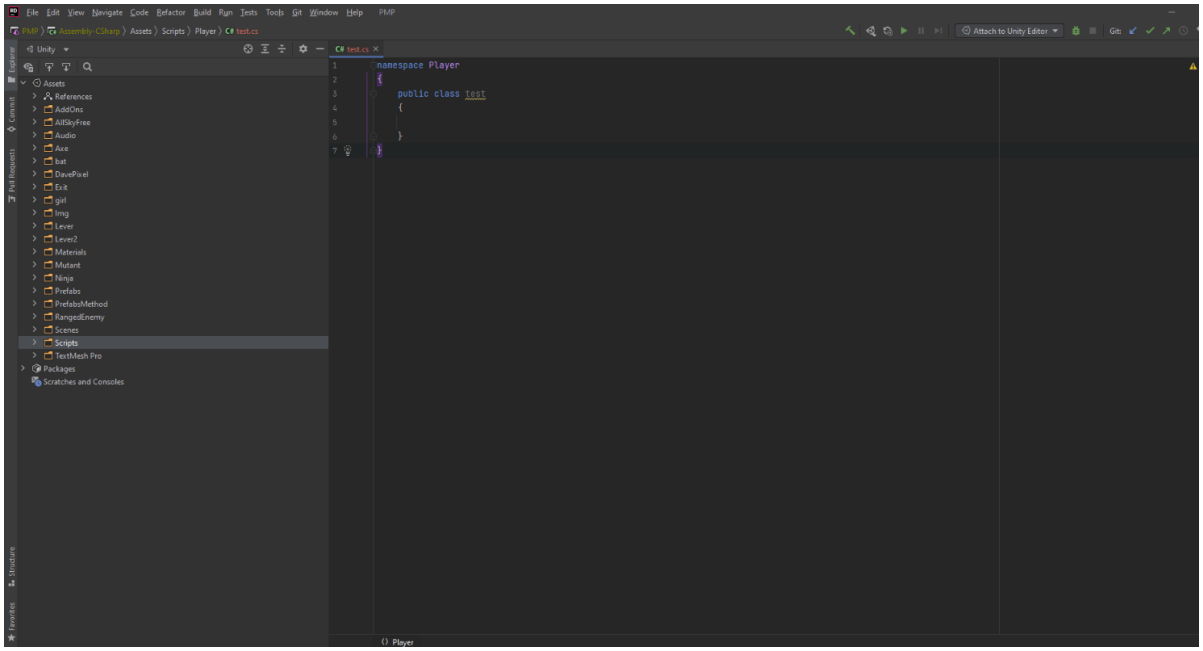


Рис. 3.5. Приклад інтерфейсу середовища розробки Rider

Середовище підтримує мультиплатформенність, тому ви можете використовувати його в Windows, macOS і Linux. Це дає розробникам гнучкість у виборі операційної системи, а також полегшує командну роботу, коли учасники можуть працювати на різних платформах. Крім того, Rider має зручний і адаптивний інтерфейс, який можна налаштувати під індивідуальні потреби користувача.

Важливим аспектом є інтеграція Rider з іншими інструментами JetBrains, такими як ReSharper. Цей функціонал надає ще більше можливостей для оптимізації коду та аналізу проекту та його структури. Rider також підтримує системи контролю версій, такі як Git, що значно спрощує співпрацю над проектом, надаючи зручний доступ до інструментів для об’єднання гілок, перевірки історії змін і вирішення конфліктів.

Крім того, варто зазначити, що Rider підтримує широкий спектр плагінів, які розширюють його функціональність. Це дозволяє адаптувати середовище до конкретних потреб проекту та автоматизувати рутинні завдання. Крім того, середовище має чудову документацію та активну спільноту розробників, що

дозволяє швидко знаходити відповіді на більшість питань, що виникають під час роботи.

Тому Rider був обраний за його продуктивність, адаптивність і спеціальні інструменти для роботи з Unity. Це середовище дозволяє зменшити кількість помилок, прискорити розробку, а також забезпечує комфортну та ефективну роботу з ігровими проектами.

3.2 Характеристика інтерфейсу

Для розробленого рівня, який реалізує створений метод, було спроектовано наступний інтерфейс:

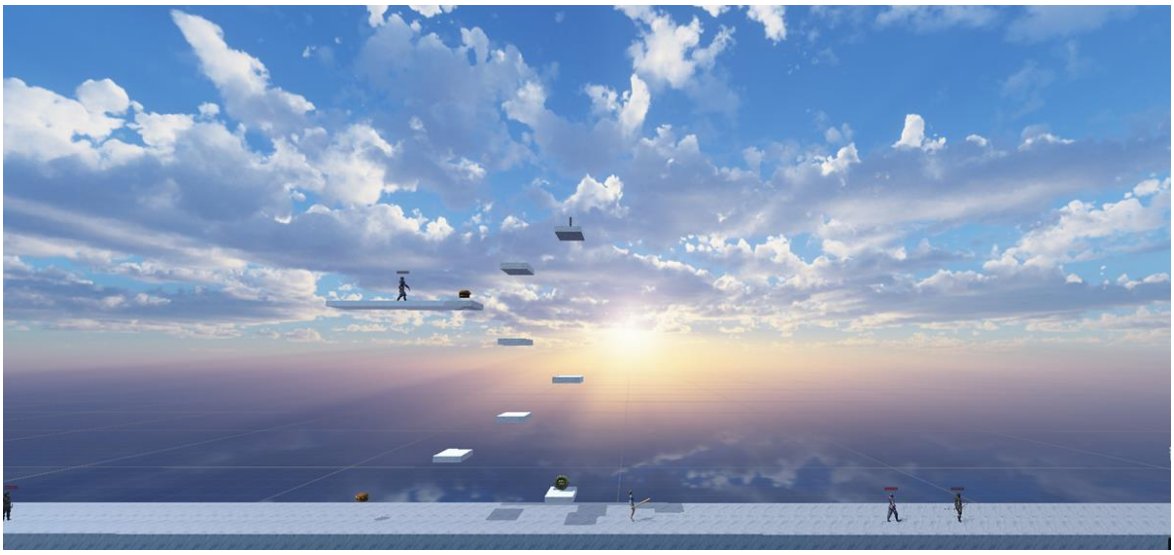


Рис. 3.6. Приклад першого рандомно сгенерованого рівня

Рівень генерується абсолютно випадковим чином, що забезпечує унікальний ігровий досвід кожного разу, коли ви починаєте нову гру. Платформи генеруються з початковою позицією, вибраною випадковим чином у вказаних розмірах карти. Це дозволяє уникнути одноманітності розташування ігрових об'єктів. Подальший напрямок руху платформи також визначається випадковим чином: з імовірністю 50% платформа продовжує свій шлях вліво, а з імовірністю 50% вправо. Такий підхід допомагає створювати непередбачувані, але в той же час цікаві маршрути для проходження рівня.

В процесі генерації обов'язково додається платформа з ворогом. Тип ворога визначається випадковим чином, причому ймовірність ніндзя або стрільця розподіляється порівну: 50% для кожного. Це забезпечує збалансовану складність і різноманітність ворогів на рівні. Крім того, існує 10% ймовірність появи бонусу або монети на платформі. Бонуси та монети додають елемент дослідження та заохочують гравця виконувати завдання, щоб прогресувати в грі.

Особливої уваги заслуговують рухомі платформи, які створюють додаткові випробування для гравця. Вони рухаються або зліва направо, або навпаки, причому напрямок руху визначається випадковим чином. Крім того, траєкторія їхнього руху генерується таким чином, що додає рівню динамічності та підвищує інтерес до ігрового процесу. Це також вимагає від гравця прорахунку своїх дій, що підвищує рівень залученості.

В кінці генерації рівня неминуче додається платформа з ричагом, що завершує логічну структуру рівня. Цей ричаг відіграє ключову роль в ігровій механіці, наприклад, він може відкрити доступ до наступного рівня або активувати певну подію. Завдяки такому підходу рівень має чітко визначені початок, основний шлях і кінцеву точку.

Кожен створений рівень ретельно перевіряється на відповідність критеріям валідності. Важливо, щоб усі платформи були доступні для гравця, уникали накладання об’єктів, мали правильний баланс між складністю та підтримкою, а також відповідали загальній концепції гри. Ця структура забезпечує створення різноманітних рівнів, які залишаються логічними, збалансованими та цікавими. Випадкова генерація в поєднанні з контрольним тестуванням забезпечує високі рівні якості без необхідності вручну проектувати кожен рівень.

Рис. 3.7. Приклад другого рандомно сгенерованого рівня

Також на рівні з вирогіднісю 90% може заспавнитись 2 вороги, двух різних типів, таких як ніндзя, або стрілець, ще з вирогідністю 40% ще 2 вороги цих двух типів. Ворогів теж потрібно вбити, для виконання квесту.

Так як вороги і монетки генеруються рандомно, то було зробленно динамічну квест систему, ось приклад:

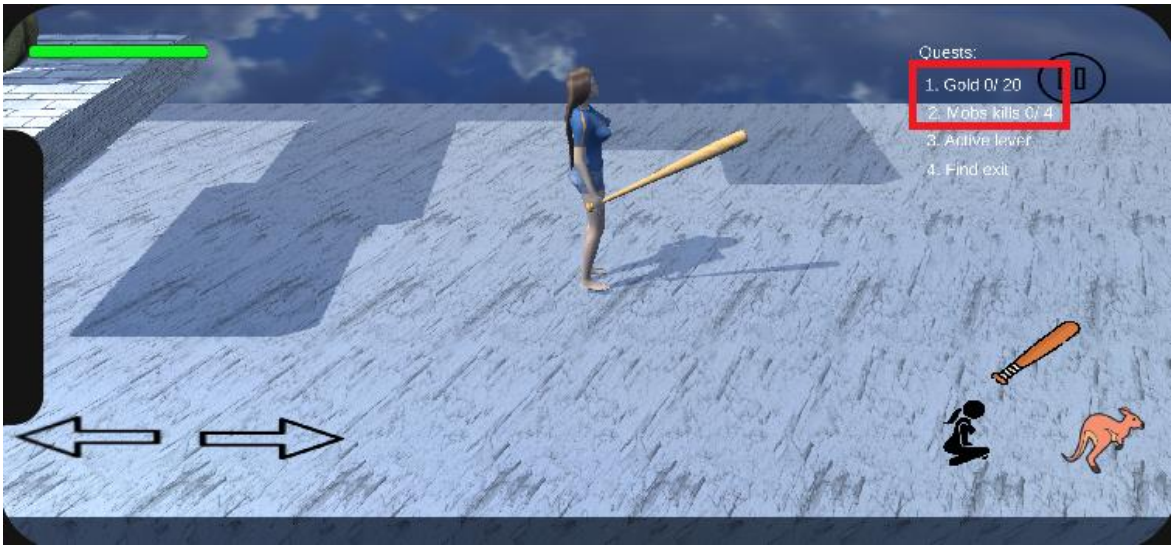


Рис. 3.8. Зображення інтерфейсу користувача, де помічено динамічні завдання

За допомогою інспектора в Unity можна ефективно керувати параметрами об'єктів, що значно полегшує процес розробки. Інтерфейс цього інструменту

автоматично адаптується під обраний об'єкт, дозволяючи розробнику швидко змінювати його характеристики. Наприклад, якщо вибрано платформу, можна відредагувати її розміри, розташування, текстуру та інші параметри, без необхідності заглиблюватися в код. Це дозволяє зберегти час, оскільки всі налаштування відбуваються в реальному часі в редакторі.

Для більш детального контролю за рівнями, інспектор дає можливість налаштовувати значення, такі як позиції платформ, ворогів, бонусів, кількість об'єктів на рівні і навіть специфічні параметри поведінки гравця чи ворогів. Всі ці налаштування змінюються безпосередньо в редакторі, що дає можливість візуально бачити зміни та швидко відкоригувати будь-які неточності.

Більш того, в Unity можна створювати кастомізовані інспектори для власних класів, що дозволяє створювати спеціалізовані редактори для конкретних типів об'єктів. Це дає додаткові можливості для оптимізації і покращення процесу розробки. Таким чином, інтерфейс Unity є потужним інструментом для керування і налаштування всіх аспектів гри, включаючи елементи рівнів, що безпосередньо впливають на геймплей.

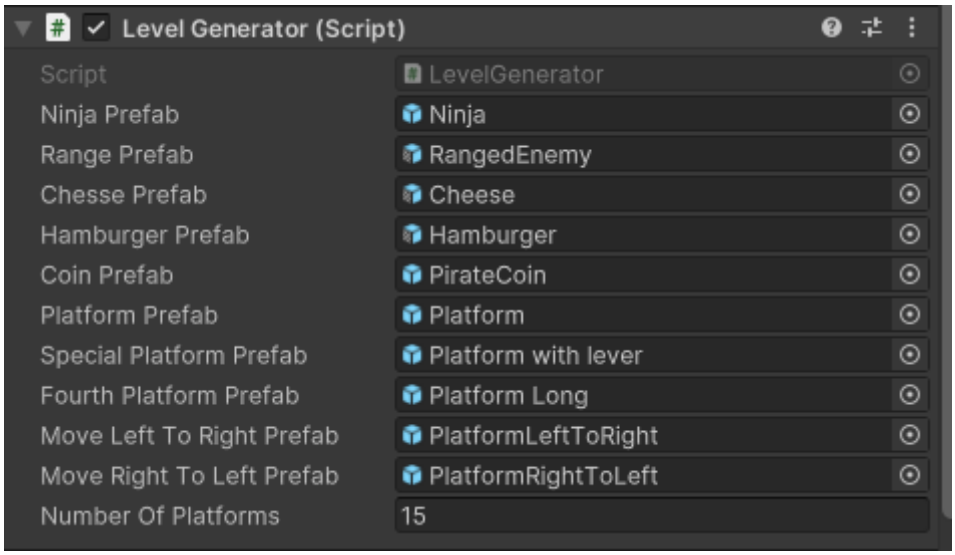


Рис. 3.9. Використання інспектору в Unity для налаштувань рівня

Як показано на рисунку, кількість платформ може бути до 15. Для створення рівня також важливою є вибірка елементів, які використовуються для візуалізації згенерованої карти, зокрема бонуси, монети та вороги.

3.3 Опис класів

Код LevelGenerator у Unity визначає основні параметри та механізми для генерації рівнів у 2.5D платформері. Він використовує поля з атрибутом [SerializeField], щоб дозволити налаштовувати префаби ворогів (ніндзя, стрільці), бонусів (сир, гамбургери), монет і платформ (стандартних, спеціальних, рухомих) безпосередньо в інспекторі Unity. Змінні numberOfPlatforms, occupiedPositions,minDistanceBetweenObjects,playerPosition,playerAvoidanceRang e забезпечують контроль над кількістю платформ, уникають перетинів об’єктів та враховують зону безпеки навколо стартової позиції гравця. Метод GenerateLevel() запускає процес генерації рівня через виклик окремих методів, які відповідають за створення платформ, ворогів, бонусів та монет, гарантуючи різноманітність і структурованість.

Для розробки генератора рівня у середовищі Unity був створений наступний код:

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

2 asset usages
public class LevelGenerator : MonoBehaviour
{
    [SerializeField] public GameObject ninjaPrefab;  Ninja
    [SerializeField] public GameObject rangePrefab;  Changed in 1 asset
    [SerializeField] public GameObject chessePrefab;  Changed in 1 asset
    [SerializeField] public GameObject hamburgerPrefab;  Changed in 1 asset
    [SerializeField] public GameObject coinPrefab;  PirateCoin
    [SerializeField] public GameObject platformPrefab;  Platform
    [SerializeField] public GameObject specialPlatformPrefab;  Platform with lever
    [SerializeField] public GameObject fourthPlatformPrefab;  Platform Long
    [SerializeField] public GameObject moveLeftToRightPrefab;  PlatformLeftToRight
    [SerializeField] public GameObject moveRightToLeftPrefab;  PlatformRightToLeft

    public int numberOfPlatforms;  Changed in 1 asset
    private List<Vector3> occupiedPositions = new List<Vector3>();
    private float minDistanceBetweenObjects = 5f;
    private Vector3 playerPosition = new Vector3(x:3, y:10, z:0);
    private float playerAvoidanceRange = 20f;

    Event function
    void Start()
    {
        GenerateLevel();
    }

    1 usage
    void GenerateLevel()
    {
        PlatformGeneration();
        NinjaGeneration();
        RangeGeneration();
        BonusGeneration();
        CoinGeneration();
    }
```

Рис. 3.10. LevelGenerator

Клас LevelGenerator у Unity визначає основні параметри та механізми для генерації рівнів у 2.5D платформері. Він використовує поля з атрибутом [SerializeField], щоб дозволити налаштовувати префаби ворогів (ніндзя, стрільці), бонусів (сир, гамбургери), монет і платформ (стандартних, спеціальних, рухомих) безпосередньо в інспекторі Unity. Змінні numberOfPlatforms, occupiedPositions,minDistanceBetweenObjects,playerPosition,playerAvoidanceRang e забезпечують контроль над кількістю платформ, уникають перетинів об’єктів та враховують зону безпеки навколо стартової позиції гравця. Метод GenerateLevel() запускає процес генерації рівня через виклик окремих методів, які відповідають за створення платформ, ворогів, бонусів та монет, гарантуючи різноманітність і структурованість.

```
bool IsPositionAvailable(Vector3 position)
{
    foreach (var occupiedPosition Vector3 in occupiedPositions)
    {
        if (Vector3.Distance( a: position, b: occupiedPosition) < minDistanceBetweenObjects || Mathf.Abs( f: position.x - playerPosition.x) < playerAvoidanceRange)
        {
            return false;
        }
    }
    return true;
}

8 usages
Vector3 GetRandomPosition(float xMin, float xMax, float yMin, float yMax)
{
    Vector3 position;
    do
    {
        float x = Random.Range(xMin, xMax);
        float y = Random.Range(yMin, yMax);
        position = new Vector3(x, y, z: 0);
    }
    while (!IsPositionAvailable(position));

    occupiedPositions.Add(position);
    return position;
}
```

Рис. 3.11. IsPositionAvailable, GetRandomPosition

Код IsPositionAvailable(Vector3 position) перевіряє, чи є вказана позиція доступною для розміщення нового об'єкта. Він виконує дві основні перевірки. По-перше, визначається, чи знаходиться позиція достатньо далеко від усіх інших зайнятих місць на рівні. Це здійснюється за допомогою Vector3.Distance, яка порівнює відстань між новою позицією та кожною з уже зайнятих. Якщо відстань менша за мінімально допустиму (minDistanceBetweenObjects), позиція

вважається недоступною. По-друге, перевіряється, чи не знаходиться позиція занадто близько до стартової точки гравця. Це обмеження встановлюється через параметр playerAvoidanceRange, який гарантує, що нові об'єкти не з'являються в безпосередній близькості до гравця.

Код GetRandomPosition(float xMin, float xMax, float yMin, float yMax) генерує випадкову позицію в межах заданих координат по осях X і Y. Спочатку вибираються випадкові значення для X і Y за допомогою Random.Range, і ці координати комбінуються в нову позицію. Після цього виконується перевірка, чи доступна ця позиція за допомогою методу IsPositionAvailable. Якщо позиція недоступна, процес генерації повторюється до тих пір, поки не буде знайдена придатна для розміщення. Після того як генерується доступна позиція, вона додається до списку occupiedPositions, щоб уникнути її повторного використання в майбутньому. Це дозволяє гарантувати, що нові об'єкти не перекриваються з іншими та правильно розміщуються на рівні.

```
void NinjaGeneration()
{
    if (Random.value <= 0.9f)
    {
        Vector3 ninjaPosition1 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(ninjaPrefab, ninjaPosition1, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }

    if (Random.value <= 0.4f)
    {
        Vector3 ninjaPosition2 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(ninjaPrefab, ninjaPosition2, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }
}

1 usage
void RangeGeneration()
{
    if (Random.value <= 0.9f)
    {
        Vector3 rangePosition1 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(rangePrefab, rangePosition1, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }

    if (Random.value <= 0.4f)
    {
        Vector3 rangePosition2 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(rangePrefab, rangePosition2, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }
}
```

Рис. 3.12. NinjaGeneration, RangeGeneration

Методи NinjaGeneration() та RangeGeneration() відповідають за випадкову генерацію ворогів (ніндзя та стрільців) на рівні.

У кожному з цих методів виконується перевірка з випадковим шансом, визначеним через Random.value (це випадкове значення від 0 до 1). Якщо значення менше або рівне заданому порогу, то генерується новий ворог.

У методі NinjaGeneration() перший ворог (ніндзя) генерується з ймовірністю 90% (якщо Random.value <= 0.9f). Якщо умова виконується, викликається метод GetRandomPosition для отримання випадкової позиції на рівні (у межах X від -115 до 100 та Y від 10 до 15). Позиція передається в Instantiate, щоб створити об'єкт ніндзя на цій позиції з обертанням по осі Y на 90 градусів.

Другий ворог (ніндзя) генерується з ймовірністю 40% (якщо Random.value <= 0.4f). Для цього також генерується нова випадкова позиція, і ніндзя створюється на цій позиції.

Аналогічно працює метод RangeGeneration(). Перший стрілець генерується з ймовірністю 90%, з використанням тієї ж логіки для отримання випадкової позиції. Другий стрілець генерується з ймовірністю 40%, також на випадковій позиції.

Обидва методи використовують Quaternion.Euler(0, 90, 0) для обертання об'єктів навколо осі Y на 90 градусів, щоб вони правильно орієнтувалися в грі.

```
1 usage
void RangeGeneration()
{
    if (Random.value <= 0.9f)
    {
        Vector3 rangePosition1 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(rangePrefab, rangePosition1, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }

    if (Random.value <= 0.4f)
    {
        Vector3 rangePosition2 = GetRandomPosition(xMin: -115, xMax: 100, yMin: 10, yMax: 15);
        Instantiate(rangePrefab, rangePosition2, rotation: Quaternion.Euler(x: 0, y: 90, z: 0));
    }
}

1 usage
void BonusGeneration()
{
    if (Random.value <= 0.8f)
    {
        Vector3 bonusPosition1 = GetRandomPosition(xMin: -50, xMax: 50, yMin: 12, yMax: 15);
        Instantiate(chessePrefab, bonusPosition1, Quaternion.identity);
    }

    if (Random.value <= 0.8f)
    {
        Vector3 bonusPosition2 = GetRandomPosition(xMin: -50, xMax: 50, yMin: 12, yMax: 15);
        Instantiate(hamburgerPrefab, bonusPosition2, Quaternion.identity);
    }
}
```

Рис. 3.13. NinjaGeneration, RangeGeneration

Методи NinjaGeneration() та RangeGeneration() відповідають за випадкову генерацію ворогів (ніндзя та стрільців) на рівні.

У кожному з цих методів виконується перевірка з випадковим шансом, визначеним через Random.value (це випадкове значення від 0 до 1). Якщо значення менше або рівне заданому порогу, то генерується новий ворог.

У методі NinjaGeneration() перший ворог (ніндзя) генерується з ймовірністю 90% (якщо Random.value <= 0.9f). Якщо умова виконується, викликається метод GetRandomPosition для отримання випадкової позиції на рівні (у межах X від -115 до 100 та Y від 10 до 15). Позиція передається в Instantiate, щоб створити об'єкт ніндзя на цій позиції з обертанням по осі Y на 90 градусів.

Другий ворог (ніндзя) генерується з ймовірністю 40% (якщо Random.value <= 0.4f). Для цього також генерується нова випадкова позиція, і ніндзя створюється на цій позиції.

Аналогічно працює метод RangeGeneration(). Перший стрілець генерується з ймовірністю 90%, з використанням тієї ж логіки для отримання випадкової позиції. Другий стрілець генерується з ймовірністю 40%, також на випадковій позиції.

Обидва методи використовують Quaternion.Euler(0, 90, 0) для обертання об'єктів навколо осі Y на 90 градусів, щоб вони правильно орієнтувалися в грі.

```
void CoinGeneration()
{
    if (Random.value <= 0.6f)
    {
        Vector3 coinPosition1 = GetRandomPosition(xMin: -60, xMax: 60, yMin: 12, yMax: 15);
        Quaternion coinRotation = Quaternion.Euler(x: 90, y: 180, z: 0);
        Instantiate(coinPrefab, coinPosition1, coinRotation);
    }

    if (Random.value <= 0.4f)
    {
        Vector3 coinPosition2 = GetRandomPosition(xMin: -60, xMax: 60, yMin: 12, yMax: 15);
        Quaternion coinRotation = Quaternion.Euler(x: 90, y: 180, z: 0);
        Instantiate(coinPrefab, coinPosition2, coinRotation);
    }
}
```

Рис. 3.14. NinjaGeneration, RangeGeneration

Метод CoinGeneration() відповідає за створення монет у грі. Він використовує випадковість для визначення, де монети з'являтимуться, і скільки їх буде.

У першому випадку монета генерується з ймовірністю 60%. Умовою для її створення є те, що значення Random.value повинно бути меншим або рівним 0.6. Якщо це так, викликається метод GetRandomPosition, щоб отримати випадкові

координати для монети в межах X від -60 до 60 і Y від 12 до 15. Після цього за допомогою Instantiate монета з'являється на цих координатах.

Монета також отримує обертання, яке задається через Quaternion.Euler(90, 180, 0). Це означає, що монета буде повернена на 90 градусів навколо осі X і на 180 градусів навколо осі Y, що встановлює її орієнтацію у просторі гри.

У другому випадку монета генерується з ймовірністю 40%. Якщо значення Random.value менше або рівне 0.4, створюється ще одна монета. Вона генерується так само, як і перша, з тими ж обмеженнями на координати та орієнтацію.

Таким чином, метод CoinGeneration() додає варіативність у гру, випадково генеруючи монети на різних позиціях з різними шансами.

```
void PlatformGeneration()
{
    float X = Random.Range(-10, 50);
    float Y = 15;

    int numberOfPlatforms = Random.Range(5, 16);
    int lastDirection = 0;

    int randomFourthPlatformIndex;
    int randomMovingPlatformIndex;

    do
    {
        randomFourthPlatformIndex = Random.Range(2, numberOfPlatforms - 2);
        randomMovingPlatformIndex = Random.Range(2, numberOfPlatforms - 2);
    } while (randomMovingPlatformIndex == randomFourthPlatformIndex || randomMovingPlatformIndex + 1 == randomFourthPlatformIndex);

    bool moveRightToLeft = Random.value < 0.5f;

    for (int i = 0; i < numberOfPlatforms; i++)
    {
        Vector3 platformPosition = new Vector3(X, Y, z: 0);
        Quaternion platformRotation = Quaternion.Euler(x: 0, y: 0, z: 0);
        GameObject platformInstance;

        if (i == 0)
        {
            platformInstance = Instantiate(platformPrefab, platformPosition, platformRotation);
        }
        else if (i == numberOfPlatforms - 1)
        {
            platformInstance = Instantiate(specialPlatformPrefab, platformPosition, platformRotation);
        }
        else if (i == randomMovingPlatformIndex)
```

Рис. 3.15. NinjaGeneration, RangeGeneration

Метод PlatformGeneration() відповідає за генерацію платформ на рівні гри. Спочатку задаються початкові значення для позицій платформ: X та Y. Позиція X вибирається випадково в діапазоні від -10 до 50, а Y встановлюється на значення 15. Потім визначається кількість платформ, що будуть згенеровані, це значення вибирається випадковим чином в діапазоні від 5 до 15.

Далі, для визначення місць для спеціальних елементів рівня, генеруються випадкові індекси для randomFourthPlatformIndex і randomMovingPlatformIndex. Ці індекси визначають, де саме на рівні будуть знаходитися спеціальна платформа (четверта) та рухома платформа. Щоб уникнути їхнього перетину, процес повторюється, поки індекси не стануть різними.

Також, створюється змінна moveRightToLeft, яка випадковим чином визначає напрямок руху для рухомої платформи. Якщо значення Random.value менше 0.5, то платформа буде рухатись справа наліво, інакше — зліва направо.

У циклі for, що проходить по кожній платформі, визначається тип платформи для кожної ітерації. Перша платформа завжди є стандартною, для останньої платформи обирається спеціальна платформа. Якщо поточний індекс співпадає з індексом для рухомої платформи, то в залежності від напрямку руху генерується відповідна платформа. Позиція X змінюється в залежності від того, в який бік рухається платформа.

Якщо індекс платформи збігається з індексом для четвертої платформи, її позиція зміщується по осі X, і на цій платформі генерується або ніндзя, або стрілець. Якщо жоден із цих варіантів не підходить, створюється стандартна платформа. Крім того, для цих платформ випадковим чином генеруються бонуси, такі як монети або гамбургери. Кожен раз після створення платформи її позиція додається до списку occupiedPositions, що дозволяє контролювати, щоб платформи не накладалися одна на одну.

3.4 Результати розробленого методу

Результати дослідження для жанру платформер 2.5D показують значні відмінності у ефективності різних методів рандомної генерації рівнів.

Розроблений метод (PMP) продемонстрував найвищу ефективність, досягнувши 92% правильних генерацій та лише 8% неправильно згенерованих рівнів. Це свідчить про високу точність і надійність методу, що дозволяє створювати різноманітні та логічно послідовні рівні. Таким чином, цей метод є оптимальним для використання в жанрі платформер 2.5D, де важлива не тільки варіативність рівнів, але й їхня стабільність і відповідність певним критеріям. Порівняно з розробленим методом, інші підходи показали меншу ефективність. Найгірший результат був зафіксований для методу Генерації випадкових рівнів, де тільки 75% генерацій були правильними, а 25% рівнів не відповідали встановленим критеріям. Це вказує на значні недоліки в стабільності цього методу, оскільки велика частина згенерованих рівнів виявилася непридатною для гри. Зокрема, метод Генерації випадкових рівнів не забезпечує достатнього контролю над якістю генерованих елементів, що призводить до частих помилок в побудові рівнів.

Інші методи, такі як Метод Комбінованого підходу, Генетичні алгоритми та Моделі на основі шаблонів, також продемонстрували помітні результати, однак не змогли досягти такої високої точності, як розроблений метод. Це підтверджує важливість правильного поєднання випадковості і стабільності для забезпечення надійності та якості рівнів у платформер 2.5D, що має бути основною задачею рандомної генерації.

Метод	Точність моделювання (%)	Швидкість обчислень (мс/кадр)	Стійкість до помилок (%)
Метод PMP	92%	8	8%
Метод Ручного створення рівнів	100%	0	0%
Метод Генерації випадкових рівнів	75%	25	25%
Метод Комбінованого підходу	88%	12	12%
Метод Генетичних алгоритмів	85%	15	15%
Метод Моделей на основі шаблонів	90%	10	10

Рис. 3.16. Результати дослідження

ВИСНОВКИ

Після проведеного аналізу та досліджень магістерської роботи можна зробити такі висновки:

1. Проаналізовано загальний підхід до процедурної генерації рівнів у жанрі платформерів 2.5D, що враховує різноманітні елементи, такі як платформи, вороги та бонуси.
2. Розроблений метод створення рівнів забезпечує адаптивність та варіативність шляхом використання ймовірнісного підходу до розміщення ворогів, бонусів і платформ, а також механізму унікального розташування об'єктів з урахуванням мінімальних відстаней, що сприяє уникненню повторюваності в ігровому процесі та підтримці інтересу гравця.
3. Система процедурної генерації рівнів забезпечує масштабованість, дозволяючи легко впроваджувати нові елементи, типи ворогів або об'єкти без необхідності змінювати базову архітектуру методу.
4. Бачучи результати дослідження, можна зробити висновок, що метод PMP є найефективнішим для рандомної генерації рівнів у жанрі платформер 2.5D, забезпечуючи 92% правильних генерацій і лише 8% неправильно згенерованих рівнів. У порівнянні з іншими методами, найгірший результат показав метод генерації випадкових рівнів, з 75% правильних генерацій і 25% неправильно згенерованих рівнів, що свідчить про значні проблеми зі стабільністю та якістю генерацій.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях та статті:

Статті:

1. Білоус В.С. Дібрівний О.А. Застосування методів випадкової генерації рівнів для гри в жанрі Платформер 2.5D // Зв’язок. №1, 2025. подана до друку.

Тези доповідей:

1. Білоус В.С., Дібрівний О.С., Використання відеоігор у навчанні штучного інтелекту. // IV Всеукраїнська Науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». – Київ: ДУІКТ, 2024. – С.148-149.
2. Білоус В.С., Дібрівний О.С., Роль ігор у навчанні географії для людей за допомогою інтерактивної гри на карті світу. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ». – Київ: ДУІКТ, 2024. – С.376-377.

ПЕРЕЛІК ПОСИЛАНЬ

1. Lemon School. Що таке Unity? Переваги і недоліки платформи для створення ігор // [Електронний ресурс]: https://lemon.school/blog/shho-take-unity . Дата публікації:31.10.2023 р.
2. Unity Technologies. Unity Manual: 2D or 3D Projects // [Електронний ресурс]: https://docs.unity3d.com/ru/2019.4/Manual/2Dor3D.html . Дата публікації: 15.06.2019 р.
3. Procedural Generation: An Overview: // [Електронний ресурс]: https://kentpawson123.medium.com/procedural-generation-an-overview-1b054a0f8d41 . Дата публікації: 19.02.2021 р.
4. What is a good newbie engine to learn for a 2.5D game? // [Електронний ресурс]: https://www.reddit.com/r/gamedev/comments/18ssnrb/what_is_a_good_newbie_engine_to_learn_for_a_25d/?rdt=38107 . Дата публікації: 28.12.2023 р.
5. Working on a 2.5d platformer: What's the best method for building levels? // [Електронний ресурс]: https://www.reddit.com/r/Unity3D/comments/8k9j1y/working_on_a_25d_platformer_whats_the_best_method/ . Дата публікації: 18.05.2018 р.
6. JetBrains. Rider for Unity // [Електронний ресурс]: https://www.jetbrains.com/lp/dotnet-unity/ . Дата публікації: 2023 р.
7. Game development for Unity // [Електронний ресурс]: https://www.jetbrains.com/help/rider/Unity.html Дата публікації: 21.11.2024 р.
8. What is Procedural Generation – Complete Guide // [Електронний ресурс]: https://gamedevacademy.org/what-is-procedural-generation/ . Дата публікації: 23.12.2023 р.
9. Procedural Level Generation in Games Tutorial: Part 1 // [Електронний ресурс]: https://www.kodeco.com/2637-procedural-level-generation-in-games-tutorial-part-1 . Дата публікації: 11.10.2021 р.

10. Procedural Materials // [Електронний ресурс]: https://docs.unity3d.com/550/Documentation/Manual/ProceduralMaterials.html Дата публікації: 15.05.2023 р.
11. Help with procedural level generation (different room sizes) // [Електронний ресурс]: https://discussions.unity.com/t/help-with-procedural-level-generation-different-room-sizes/775898 . Дата публікації: 9.02.2020 р.
12. How to procedurally generate maps using layered perlin noise // [Електронний ресурс]: https://www.reddit.com/r/proceduralgeneration/comments/i756qa/how_to_procedurally_generate_maps_using_layered/ . Дата публікації: 10.08.2020 р.
13. Working with Simplex Noise // [Електронний ресурс]: https://cmaher.github.io/posts/working-with-simplex-noise/ Дата публікації: 2021 р.
14. Procedural Level Generation in Games using a Cellular Automaton: Part 1 // [Електронний ресурс]: https://www.kodeco.com/2425-procedural-level-generation-in-games-using-a-cellular-automaton-part-1 Дата публікації: 30.08.2022 р.
15. Cellular Automata // [Електронний ресурс]: https://procedural-content-generation.fandom.com/wiki/Cellular_Automata Дата публікації: 21.02.2021 р.
16. Game Design: Level Generation Using Binary Space Partitioning // [Електронний ресурс]: https://medium.com/nice-things-ios-android-development/game-design-level-generation-using-binary-space-partitioning-e229230ab8b5 . Дата публікації: 04.08.2018 р.
17. Procedural generation of geometric patterns for thin shell fabrication // [Електронний ресурс]: https://www.sciencedirect.com/science/article/pii/S0097849324000931 Дата публікації: 13.05.2023 р.

18. The Best Ways to Use Procedural Generation in Games // [Електронний ресурс]: https://game-ace.com/blog/procedural-generation-in-games/ 15.05.2024 р.
19. Procedural Content Generation Overview // [Електронний ресурс]: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-overview . Дата публікації: 10.03.2021 р.
20. Map Graph – Node-based Random Map Generation // [Електронний ресурс]: https://discussions.unity.com/t/map-graph-node-based-random-map-generation/802981 . Дата публікації: 04.02.2023 р.
21. Red Blob Games. Genetic Algorithms for Procedural Generation // [Електронний ресурс]: https://www.redblobgames.com/articles/genetic-algorithms/ . Дата публікації: 10.01.2022 р.
22. Example of genetic algorithms in procedural generation of levels? // [Електронний ресурс]: https://www.reddit.com/r/gamedev/comments/6ccp14/example_of_genetic_algorithms_in_procedural/ . Дата публікації: 25.11.2020 р.
23. Wave Function Collapse // [Електронний ресурс]: https://excaliburjs.com/blog/Wave%20Function%20Collapse/ Дата публікації: 01.06.2024 р.
24. The Wavefunction Collapse Algorithm explained very clearly // [Електронний ресурс]: https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/ . Дата публікації: 17.12.2018 р.
25. The Power of Procedural Generation in Gaming: Endless Possibilities Unleashed // [Електронний ресурс]: https://kedarshukla007.medium.com/the-power-of-procedural-generation-in-gaming-endless-possibilities-unleashed-8482a52b71ef . Дата публікації: 29.05.2023 р.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

Метод випадкової генерації рівнів для гри в жанрі Платформер 2.5D

Виконав: студент групи ПДМ-63 Білоус Валерій Сергійович

Керівник: доктор філософії (Phd) ІПЗ Дібрівний Олесь Андрійович

Київ - 2025

МЕТА, ОБ'ЄКТА ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення якості створення рівнів для ігор жанру Платформер 2.5D шляхом використання автоматичної генерації.

Об'єкт дослідження: процес випадкової генерації рівнів у грі жанру Платформер 2.5D.

Предмет дослідження: технології для генерації рівнів за допомогою штучного інтелекту.

АКТУАЛЬНІСТЬ РОБОТИ

- Сучасні гравці все частіше очікують інноваційного підходу до генерації рівнів, які забезпечують унікальний ігровий досвід та постійно тримають інтерес, що зумовлює зростаючий попит на складні алгоритми, які можуть створювати різноманітний і збалансований контент, що відповідає вимогам сучасних ігор.
- Інтеграція автоматизованих методів створення рівнів дозволяє значно скоротити витрати часу і ресурсів, необхідних для розробки, що особливо важливо для інді-команд і невеликих студій, які прагнуть конкурувати з великими розробниками.
- Особливу актуальність має адаптивна генерація рівнів, яка враховує стиль гри, потреби користувача, що не лише покращує взаємодію з грою, а й забезпечує її релевантність для широкої аудиторії, що є критичним фактором успіху в умовах високої конкуренції в індустрії ігор.

3

Порівняння методів створення рівня жанру платфомер 2.5D

Показник	Метод Ручного створення рівнів	Метод Генерації випадкових рівнів	Метод Комбінованого підхіду	Метод Генетичних алгоритмів	Метод Моделей на основі шаблонів	Метод РМР
Простота реалізації	+	+	-	-	+	+
Різноманітність рівнів	-	+	+	+	+	+
Масштабованість	-	+	+	+	-	-
Швидкість обчислень	+	+	-	-	-	+
Адаптація до умов	-	-	+	+	+	+
Складність налаштування	-	+	-	-	+	-

6

Математичне представлення створення рівня

Опис параметрів:

p_{min}- мінімальна кількість платформ, яку потрібно розмістити на рівні.
p_{max}- максимальна кількість платформ, яку потрібно розмістити на рівні.
$p_{platform}$- позиція кожної платформи, яка визначається випадково в межах$[x_{min}, x_{max}]$ і $[y_{min}, y_{max}]$
d_{min}- мінімальна відстань між платформами для уникнення накладів.
r_{avoid}- радіус, в межах якого платформи не повинні бути розміщені занадто близько до гравця.
p_{ninja}- ймовірність створення ворога типу "ніндзя"
p_{range}- ймовірність створення ворога типу “стрілець”.
e_{enemy}- тип ворога, який генерується на основі ймовірностей p_{new} та p_{range}
R - операція випадкового вибору позиції для розміщення елементів (платформи, вороги, бонуси) у межах заданих координат.
p_{coin}- ймовірність створення бонуса типу "монета".
p_{bonus}- ймовірність створення бонуса типу "спеціальний бонус".
b_{bonus}- тип бонуса, який генерується на основі ймовірностей p_{new} та p_{range}.
$p_{occupied}$- список вже зайнятих позицій.
p_{player}- позиція гравця на рівні.
V_{valid}- функція перевірки валідності позиції p, яка визначає, чи є позиція валідною для розміщення об'єкта, залежно від мінімальної відстані до інших об'єктів і гравця.
p_{count}- кількість елементів (платформ, ворогів або бонусів), яка генерується випадковим чином у межах $[p_{min}, p_{max}]$, де p_{min} — мінімальна кількість, а p_{max} - максимальна кількість.
p_{new}- нові позиції для елементів, генеруються випадковим чином в межах заданих мінімальних і максимальних значень для координат x і y на рівні.

Кількість платформ

$$p_{count} = R[p_{min}, p_{max}]$$

Кількість ворогів

$$e_{count} = R[e_{min}, e_{max}]$$

Кількість бонусів

$$b_{count} = R[b_{min}, b_{max}]$$

Тип бонусу

$$b_{bonus} = \begin{cases} \text{coin, з ймовірністю } p_{coin} \\ \text{specia, з ймовірністю } p_{bonus} \\ \emptyset, \text{якщо ймовірність не спрацювала} \end{cases}$$

Перевірка валідності позиції

$$V_{valid} = \begin{cases} True, \text{якщо } |p - p_{occupied}| \geq d_{min} \; \forall \, p_{occupied} \wedge |p - p_{player}| \geq r_{avoid} \\ False, \text{якщо не виконується} \end{cases}$$

Генерування нової позиції

$$p_{new} = R[x_{min}, x_{max}, y_{min}, y_{max}]$$

Позиція платформи

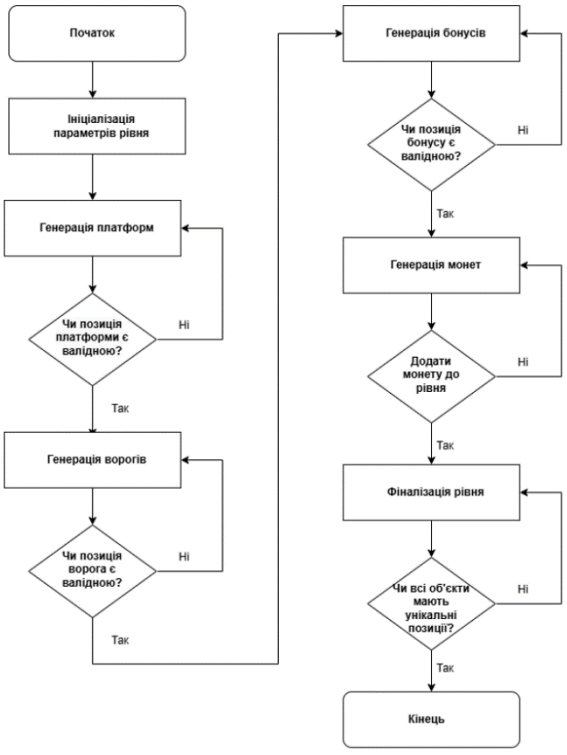
$$P_{platform} = \begin{cases} R[x_{min}, x_{max}, y_{min}, y_{max}] \text{, якщо позиція валідна} \\ GenerateNewPosition, \text{якщо позиція не валідна} \end{cases}$$

Тип ворога

$$e_{enemy} = \begin{cases} \text{ninja, з ймовірністю } p_{ninja} \\ \text{range, з ймовірністю } p_{range} \\ \emptyset, \text{якщо ймовірність не спрацювала} \end{cases}$$

5

Алгоритм генерації рівня

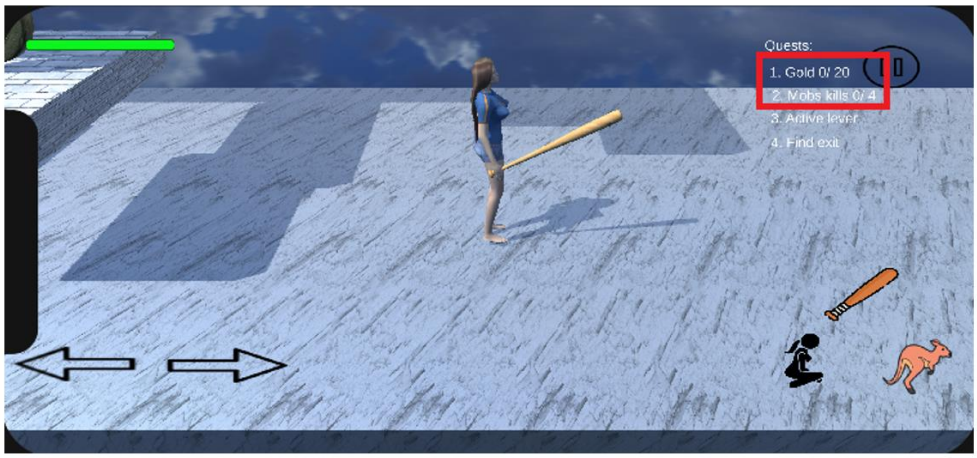


6

Приклади генерації рівня

7

ЕКРАННА ФОРМА РМР



8

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Метод	Точність моделювання (%)	Швидкість обчислень (мс/кадр)	Стійкість до помилок (%)
Метод PMP	92%	8	8%
Метод Ручного створення рівнів	100%	0	0%
Метод Генерації випадкових рівнів	75%	25	25%
Метод Комбінованого підхіду	88%	12	12%
Метод Генетичних алгоритмів	85%	15	15%
Метод Моделей на основі шаблонів	90%	10	10

8

ВИСНОВКИ

1. Проаналізовано загальний підхід до процедурної генерації рівнів у жанрі платформерів 2.5D, що враховує різноманітні елементи, такі як платформи, вороги та бонуси.
2. Розроблений метод створення рівнів забезпечує адаптивність та варіативність шляхом використання ймовірнісного підходу до розміщення ворогів, бонусів і платформ, а також механізму унікального розташування об'єктів з урахуванням мінімальних відстаней, що сприяє уникненню повторюваності в ігровому процесі та підтримці інтересу гравця.
3. Система процедурної генерації рівнів забезпечує масштабованість, дозволяючи легко впроваджувати нові елементи, типи ворогів або об'єкти без необхідності змінювати базову архітектуру методу.
4. Бачучи результати дослідження, можна зробити висновок, що метод PMP є найефективнішим для рандомної генерації рівнів у жанрі платформер 2.5D, забезпечуючи 92% правильних генерацій і лише 8% неправильно згенерованих рівнів. У порівнянні з іншими методами, найгірший результат показав метод генерації випадкових рівнів, з 75% правильних генерацій і 25% неправильно згенерованих рівнів, що свідчить про значні проблеми зі стабільністю та якістю генерацій.

10

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

1. Білоус В.С. Дібрівний О.А. Застосування методів випадкової генерації рівнів для гри в жанрі Платформер 2.5D // Зв'язок. №1, 2025. подана до друку.

Тези доповідей:

1. Білоус В.С., Дібрівний О.С., Використання відеоігор у навчанні штучного інтелекту. // IV Всеукраїнська Науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті». – Київ: ДУІКТ, 2024. – С.148-149.
2. Білоус В.С., Дібрівний О.С., Роль ігор у навчанні географії для людей за допомогою інтерактивної гри на карті світу. // Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в ІКТ». – Київ: ДУІКТ, 2024. – С.376-377.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

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

public class LevelGenerator : MonoBehaviour
{
    [SerializeField] public GameObject ninjaPrefab;
    [SerializeField] public GameObject rangePrefab;
    [SerializeField] public GameObject chessePrefab;
    [SerializeField] public GameObject hamburgerPrefab;
    [SerializeField] public GameObject coinPrefab;
    [SerializeField] public GameObject platformPrefab;
    [SerializeField] public GameObject specialPlatformPrefab;
    [SerializeField] public GameObject fourthPlatformPrefab;
    [SerializeField] public GameObject
moveLeftToRightPrefab;
    [SerializeField] public GameObject
moveRightToLeftPrefab;

    public int numberOfPlatforms;
    private List<Vector3> occupiedPositions = new
List<Vector3>();
    private float minDistanceBetweenObjects = 5f;
    private Vector3 playerPosition = new Vector3(3, 10, 0);
    private float playerAvoidanceRange = 20f;

    void Start()
    {
        GenerateLevel();
    }

    void GenerateLevel()
    {
        PlatformGeneration();
        NinjaGeneration();
        RangeGeneration();
        BonusGeneration();
        CoinGeneration();
    }

    bool IsPositionAvailable(Vector3 position)
    {
        foreach (var occupiedPosition in occupiedPositions)
        {
            if (Vector3.Distance(position, occupiedPosition) <
minDistanceBetweenObjects || Mathf.Abs(position.x -
playerPosition.x) < playerAvoidanceRange)
            {
                return false;
            }
        }
        return true;
    }

    Vector3 GetRandomPosition(float xMin, float xMax, float
yMin, float yMax)
    {
        Vector3 position;
        do
        {
            float x = Random.Range(xMin, xMax);
            float y = Random.Range(yMin, yMax);
            position = new Vector3(x, y, 0);
        }
        while (!IsPositionAvailable(position));

        occupiedPositions.Add(position);
        return position;
    }

    void NinjaGeneration()
    {
        if (Random.value <= 0.9f)
        {
            Vector3 ninjaPosition1 = GetRandomPosition(-115,
100, 10, 15);
            Instantiate(ninjaPrefab, ninjaPosition1,
Quaternion.Euler(0, 90, 0));
        }

        if (Random.value <= 0.4f)
        {
            Vector3 ninjaPosition2 = GetRandomPosition(-115,
100, 10, 15);
```

```
        Instantiate(ninjaPrefab, ninjaPosition2,
Quaternion.Euler(0, 90, 0));
      }
   }

   void RangeGeneration()
   {
      if (Random.value <= 0.9f)
      {
         Vector3 rangePosition1 = GetRandomPosition(-115,
100, 10, 15);
         Instantiate(rangePrefab, rangePosition1,
Quaternion.Euler(0, 90, 0));
      }

      if (Random.value <= 0.4f)
      {
         Vector3 rangePosition2 = GetRandomPosition(-115,
100, 10, 15);
         Instantiate(rangePrefab, rangePosition2,
Quaternion.Euler(0, 90, 0));
      }
   }

   void BonusGeneration()
   {
      if (Random.value <= 0.8f)
      {
         Vector3 bonusPosition1 = GetRandomPosition(-50,
50, 12, 15);
         Instantiate(chessePrefab, bonusPosition1,
Quaternion.identity);
      }

      if (Random.value <= 0.8f)
      {
         Vector3 bonusPosition2 = GetRandomPosition(-50,
50, 12, 15);
         Instantiate(hamburgerPrefab, bonusPosition2,
Quaternion.identity);
      }
   }

   void CoinGeneration()
   {
      if (Random.value <= 0.6f)
      {
         Vector3 coinPosition1 = GetRandomPosition(-60, 60,
12, 15);
         Quaternion coinRotation = Quaternion.Euler(90, 180,
0);
         Instantiate(coinPrefab, coinPosition1, coinRotation);
      }

      if (Random.value <= 0.4f)
      {
         Vector3 coinPosition2 = GetRandomPosition(-60, 60,
12, 15);
         Quaternion coinRotation = Quaternion.Euler(90, 180,
0);
         Instantiate(coinPrefab, coinPosition2, coinRotation);
      }
   }
void PlatformGeneration()
{
   float X = Random.Range(-10, 50);
   float Y = 15;

   int numberOfPlatforms = Random.Range(5, 16);
   int lastDirection = 0;

   int randomFourthPlatformIndex;
   int randomMovingPlatformIndex;

   do
   {
      randomFourthPlatformIndex = Random.Range(2,
numberOfPlatforms - 2);
      randomMovingPlatformIndex = Random.Range(2,
numberOfPlatforms - 2);
   } while (randomMovingPlatformIndex ==
randomFourthPlatformIndex || randomMovingPlatformIndex
+ 1 == randomFourthPlatformIndex);

   bool moveRightToLeft = Random.value < 0.5f;

   for (int i = 0; i < numberOfPlatforms; i++)
   {
      Vector3 platformPosition = new Vector3(X, Y, 0);
      Quaternion platformRotation = Quaternion.Euler(0, 0,
0);
      GameObject platformInstance;
```

```
        if (i == 0)
        {
            platformInstance = Instantiate(platformPrefab,
platformPosition, platformRotation);
        }
        else if (i == numberOfPlatforms - 1)
        {
            platformInstance = Instantiate(specialPlatformPrefab,
platformPosition, platformRotation);
        }
        else if (i == randomMovingPlatformIndex)
        {
            if (moveRightToLeft)
            {
                platformInstance =
Instantiate(moveRightToLeftPrefab, platformPosition,
platformRotation);
                X = platformPosition.x - 1;
            }
            else
            {
                platformInstance =
Instantiate(moveLeftToRightPrefab, platformPosition,
platformRotation);
                X = platformPosition.x + 1;
            }

            Y = platformPosition.y + 7;
        }
        else if (i == randomFourthPlatformIndex)
        {
            float offset = lastDirection == 0 ? -12f : 12f;
            platformPosition.x += offset;

            platformInstance = Instantiate(fourthPlatformPrefab,
platformPosition, platformRotation);

            if (Random.value < 0.5f)
            {
                Vector3 ninjaPosition = platformPosition + new
Vector3(0, 1.5f, 0);
                Instantiate(ninjaPrefab, ninjaPosition,
Quaternion.Euler(0, 90, 0));
            }
            else
            {
                Vector3 rangePosition = platformPosition + new
Vector3(0, 1.5f, 0);
                Instantiate(rangePrefab, rangePosition,
Quaternion.Euler(0, 90, 0));
            }
        }
        else
        {
            platformInstance = Instantiate(platformPrefab,
platformPosition, platformRotation);
            float chance = Random.value;

            if (chance >= 0.9f)
            {
                Vector3 coinPosition = platformPosition + new
Vector3(0, 2.1f, 0);
                Quaternion coinRotation = Quaternion.Euler(90,
180, 0);
                Instantiate(coinPrefab, coinPosition, coinRotation);
            }
            else if (chance >= 0.8f)
            {
                Vector3 hamburgerPosition = platformPosition +
new Vector3(0, 1.5f, 0);
                Instantiate(hamburgerPrefab, hamburgerPosition,
Quaternion.identity);
            }

            lastDirection = Random.Range(0, 2);
            int direction = lastDirection == 0 ? -1 : 1;
            X += direction * 10;
            Y += 7;
        }

        occupiedPositions.Add(platformPosition);
    }
}
}
```