МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАПЛЯ ГЕОРГІЙ ОЛЕКСАНДРОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2025 р.

РОЗРОБКА ІГРОВОГО ДОДАТКУ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ШТУЧНОГО ІНТЕЛЕКТУ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (магістерська) робота

Науковий керівник:
Роман БАБАКОВ, професор кафедри
інформаційних технологій,
д. т. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

АНОТАЦІЯ

Капля Г.О. Розробка ігрового додатку з використанням технологій штучного інтелекту. Спеціальність 122 «Комп’ютерні науки», освітня програма «Data science». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній (магістерскій) роботі проаналізовано застосування штучного інтелекту для процедурної генерації контенту. Практичним результатом дослідження стала розробка ігрового додатка на базі Unreal Engine 5, в якому реалізовано нейронну мережу для автоматизованої побудови рівнів.

Ключові слова: Штучний інтелект, Unreal Engine 5, генерація ігрових рівнів, умовний варіаційний автоенкодер (CVAE), Blueprint, C++, Python.

71 ст. 22 рис., 0 табл., 1 дод., 50 джерел.

ABSTRACT

Kaplia H. O. Development of a gaming application using artificial intelligence technologies. Specialty 122 «Computer Science», educational program «Data Science». Vasyl Stus Donetsk National University, Vinnytsia, 2025.

The qualification (master's) thesis analyzed the application of artificial intelligence for procedural content generation. The practical result of the research was the development of a game application based on Unreal Engine 5, in which a neural network was implemented for automated level construction.

Keywords: Artificial intelligence, Unreal Engine 5, game level generation, conditional variational autoencoder (CVAE), Blueprint, C++, Python.

ЗМІСТ

ВСТУП

Стрімкий розвиток ігрової індустрії та зростання вимог до різноманітності контенту зумовлюють потребу у створенні інструментів, здатних автоматизувати частину процесу розробки. Традиційна побудова рівнів потребує значних часових і людських ресурсів, що робить процес дорогим і маломасштабованим. Саме тому дедалі більшої уваги набувають методи процедурної генерації та технології штучного інтелекту, які дозволяють автоматизувати створення ігрових просторів та адаптувати їх до цілей геймплею.

Сучасні генеративні моделі, зокрема умовні варіаційні автоенкодери (CVAE), дають можливість формувати нові ігрові рівні на основі отриманих прикладів, відтворюючи їхню структуру та логіку розташування об’єктів. Водночас актуальним залишається питання практичної інтеграції таких моделей у комерційні рушії — зокрема Unreal Engine 5, який є одним із провідних інструментів створення тривимірних ігрових середовищ. Поєднання процедурної генерації, методів глибинного навчання та інструментів Unreal Engine створює основу для формування системи, здатної генерувати повноцінні ігрові рівні в реальному часі.

Метою даної магістерської роботи є дослідження можливостей використання нейронних мереж для генерації структурованих тривимірних рівнів та розробка ігрового додатку, що демонструє роботу моделі.

Для реалізації поставленої мети необхідно вирішити такі завдання:

1. Огляд сучасних процедурних і нейромережевих методів генерації рівнів з метою вибору придатної архітектури моделі.
2. Створення системи збирання та підготовки навчальних даних у середовищі Unreal Engine.
3. Навчання обраної генеративної моделі на сформованому наборі даних.

4. Інтеграція моделі в Unreal Engine для забезпечення генерації рівнів у режимі реального часу.
5. Розробка тестового середовища для взаємодії користувача з параметрами генерації.
6. Створення демонстраційної мінігри, у якій гравець визначає, який рівень створено моделлю.
7. Розробка повноцінного ігрового застосунку, що включає систему меню й механізми перемикання між рівнями.

Об’єктом дослідження є процес генерації ігрових рівнів.

Предметом дослідження — методи та засоби застосування нейронних мереж для побудови тривимірних сцен у рушії Unreal Engine.

У роботі застосовано методи глибинного навчання, математичного моделювання, аналізу структурованих даних, а також технічні засоби Unreal Engine 5, Blueprint, Python та C++. Практичне значення полягає у створенні робочої системи генерації рівнів, яку можна застосовувати у дослідницьких, навчальних або ігрових проєктах, розширюючи можливості автоматизованої розробки контенту.

Проміжні результати дослідження пройшли апробацію на V Всеукраїнській науково-практичній конференції вчених, аспірантів та студентів «Комп’ютерні ігри і мультимедіа як інноваційний підхід до комунікації - 2025» із доповіддю «Використання штучного інтелекту для генерації ігрових рівнів» та III Міжнародної науково-практичної конференції «Прикладні аспекти сучасних міждисциплінарних досліджень» (2024р.) із доповіддю «Використання штучного інтелекту в комп’ютерних іграх»

Структурно робота складається з трьох розділів, що відображають послідовні етапи дослідження та розробки. Перший розділ присвячений аналізу теоретичних засад процедурної генерації та нейронних мереж. У другому подано процес створення датасету, навчання моделі та її інтеграцію в Unreal Engine. Третій розділ описує кінцевий ігровий застосунок, реалізацію рівнів, мінігри та систему навігації між сценами.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ЗАСОБІВ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕНТУ

1.1 Процедурна генерація в ігровій індустрії

Сучасна ігрова індустрія стрімко рухається у напрямку автоматизації процесів створення контенту [17][18]. Одним із ключових кроків у цьому напрямі стала процедурна генерація – технологія, яка дозволяє автоматично створювати елементи ігрового світу за допомогою алгоритмів, правил чи математичних моделей [34]. Замість того, щоб дизайнер вручну розміщував кожен об’єкт на сцені, система формує структуру рівня, ландшафт, архітектуру або навіть цілі планети, спираючись на набір параметрів і випадкових значень. Це зменшує кількість ручної праці, пришвидшує процес розробки та робить гру більш різноманітною.

Із розвитком відкритих світів потреба у таких методах лише зростала. Людська праця не може забезпечити створення достатнього обсягу якісного контенту у масштабних проектах, тому автоматизація стала природним етапом еволюції геймдизайну. Генераційні алгоритми можуть працювати як на етапі розробки, так і безпосередньо під час гри, створюючи унікальний досвід для кожного користувача. Проте, як і будь-який інструмент, процедурна генерація має свої слабкі сторони.

Найпоширеніша проблема – відсутність повного контролю над результатом. Без ретельно налаштованих фільтрів чи логічних обмежень створені світи можуть бути непридатними для гри або виглядати випадково [16][45]. Інша проблема – повторюваність структури. Навіть якщо алгоритм створює тисячі варіацій, гравець все одно починає помічати знайомі патерни. До того ж, описати всі дизайнерські обмеження у вигляді формул часто складніше, ніж зробити рівень вручну.

Серед основних підходів до процедурної генерації можна виділити кілька напрямів. Просторові методи – такі як шум Перліна або фрактальні функції – добре підходять для створення природних ландшафтів [12][25][46]. Графові підходи дозволяють будувати архітектурно логічні структури, наприклад підземелля або міста, де враховується зв'язність кімнат і маршрутів [49]. Часто застосовується і комбінований підхід – коли рівень збирається з готових частин (тайлів), які поєднуються за заданими правилами [50].

Показовим прикладом процедурної генерації є No Man's Sky, у якій цілий всесвіт із мільярдами планет створюється автоматично. Кожна планета має унікальні рельєфи, атмосферу, флору й фауну, сформовані з допомогою складних шумових функцій та стохастичних алгоритмів. Масштаб проєкту був вражаючим, але після релізу гравці відзначили, що більшість світів здаються надто схожими. Величезний простір виявився однотипним, а дослідження – монотонним. Це показало, що навіть складні алгоритми не замінюють дизайнерського підходу, який задає ідею, сенс і ритм гри.

Інший приклад – Minecraft, де весь світ створюється процедурно, включно з формою гір, печер, водойм і розподілом ресурсів. Проста структура на основі вокселів дозволяє створювати практично нескінченні варіації світу, при цьому зберігаючи впізнаваність та логіку. Однак, попри вражаючу генерацію ландшафтів, підземелля (данжі) у грі часто критикують за одноманітність. Вони радше нагадують випадкові споруди з лутом, а не справжні пригодницькі локації, тож швидко втрачають загадковість після кількох досліджень.

Подібні методи використовуються і в іграх жанру roguelike, як-от The Binding of Isaac або Spelunky. Там рівні формуються з готових кімнат, які комбінуються у випадковому порядку. Такий підхід дозволяє досягти балансу між передбачуваністю й випадковістю: гра залишається контрольованою, але кожен забіг відрізняється від попереднього.

У підсумку можна сказати, що процедурна генерація задовольняє потреби більшості сучасних проєктів, але її можливості все ще обмежені. Вона

схильна до шаблонності, бо рішення, які приймає алгоритм, завжди залежать від наперед визначених правил. Такий підхід створює світ, який формально виглядає різноманітним, але насправді часто позбавлений справжньої непередбачуваності. Щоб подолати цю механічність, у процес генерації почали впроваджувати методи штучного інтелекту – системи, які можуть навчатися на прикладах, робити висновки й адаптувати свої рішення під контекст гри [26][35][44].

1.2 Використання нейронних мереж у генерації ігрових рівнів

Побудова нейронної мережі для автоматичної генерації рівнів у відеоіграх вимагає ретельного підходу до вибору архітектури, способу збору даних та визначення цільової структури вихідних результатів [1][36]. Найважливішим етапом є формування навчального набору даних, оскільки якість і різноманітність прикладів безпосередньо впливають на здатність моделі створювати цікаві, збалансовані й логічно зв’язані рівні[11].

Збір даних може виконуватися різними методами. Найочевидніший – це акумуляція наявних рівнів, створених професійними геймдизайнерами, адже вони є найкращим зразком оптимального дизайну [14]. Однак цей підхід потребує великих ресурсів, адже збирання та підготовка навіть кількох сотень таких карт займає значний час і вимагає попередньої ручної адаптації для машинного навчання [32].

Інший підхід полягає у використанні рівнів із попередніх частин гри. Це дозволяє задіяти вже готовий контент, який має спільні механіки, стилістику та архітектурні принципи. Завдяки цьому нейромережа отримує навчальний матеріал, максимально наближений до цільового середовища, що підвищує точність та якість генерації.

Ще одним цінним джерелом можуть стати карти, створені гравцями у майстернях на зразок Steam Workshop. Такі набори містять тисячі унікальних варіацій, але не всі з них мають прийнятну якість – частина рівнів може бути незбалансованою, непридатною для проходження або навіть повністю

зруйнованою. Щоб уникнути цього, можна запровадити систему попереднього фільтрування: відбір кращих рівнів на основі рейтингу гравців, ручна модерація або застосування автоматичних метрик, які визначають валідність карти за певними критеріями (наявність старту, виходу, мінімальна кількість ігрових об’єктів тощо).

Якщо ж реальні дані зібрати складно, можна вдатися до процедурної генерації навчального набору. У цьому випадку спеціальні алгоритми створюють штучні рівні, які потім використовуються як тренувальні приклади для нейромережі. Хоча такі карти можуть бути менш цікавими, вони дають змогу отримати великі обсяги структурованих даних і контролювати параметри, важливі для навчання (щільність об’єктів, форма простору, кількість зон тощо).

Після формування набору даних постає питання вибору типу нейронної мережі. Різні архітектури мають свої переваги й обмеження, тому доцільно розглянути кілька варіантів, які найчастіше застосовуються для генерації просторових або послідовних структур.

Автоенкодери (Autoencoders) є базовим типом нейронних мереж, який застосовується для навчання стислих представлень даних. Їхня структура складається з двох частин: **енкодера** та **декодера** [37]. Енкодер отримує на вхід багатовимірний вектор (у нашому випадку – опис рівня у вигляді координат об’єктів, їх типів або воксельного представлення) і поступово зменшує його розмірність, створюючи компактне латентне представлення z. Декодер, своєю чергою, виконує зворотну операцію – реконструює початкові дані з латентного простору [2].

На рисунку 1.1 зображено типову архітектуру автоенкодера, де потік даних проходить від вхідного шару через кілька прихованих шарів енкодера, далі стискається у вектор z, після чого розгортається назад декодером. У процесі навчання мінімізується функція втрат, яка оцінює різницю між вхідними та відновленими даними. Таким чином, мережа навчається зберігати найважливішу структурну інформацію про рівень у стислому вигляді.

Автоенкодери добре підходять для виявлення прихованих закономірностей у рівнях, однак мають тенденцію до перенавчання, коли модель просто запам’ятовує карти замість узагальнення їх структури.

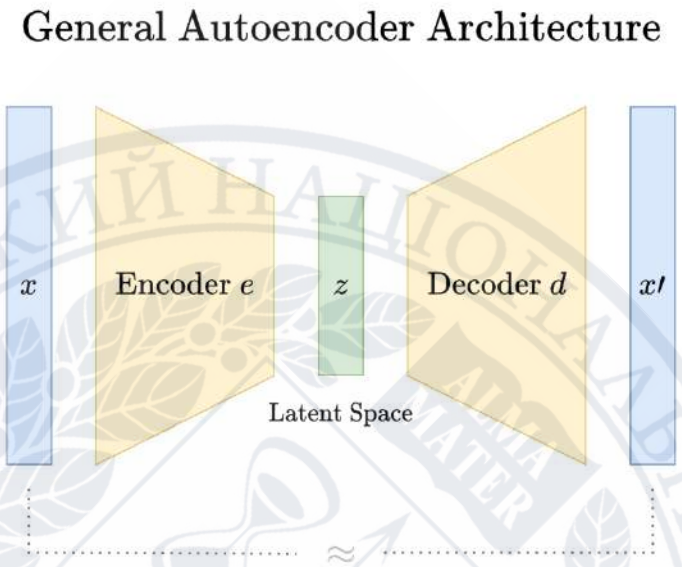

Рисунок 1.1 – Архітектура автоенкодера

Варіаційні автоенкодери (VAE) розвивають цю ідею, додаючи стохастичну компоненту у процес кодування. На відміну від звичайних автоенкодерів, які стискають дані у фіксований вектор z, варіаційні автоенкодери описують цей вектор як випадкову змінну, що підпорядковується певному розподілу (зазвичай нормальному). На рисунку 1.2 наведено схему архітектури VAE: енкодер перетворює вхідні дані на два вектори – середнє значення μ\muμ і стандартне відхилення σ\sigmaσ, які описують параметри нормального розподілу. Далі із цього розподілу вибирається конкретне латентне представлення $z = \mu + \sigma \cdot \varepsilon$, де ε – випадкова змінна, що моделює шум [27].

Декодер реконструює рівень із отриманого зразка, створюючи нові комбінації об’єктів, яких не було у вихідних даних. Завдяки стохастичній природі, VAE здатен генерувати нові унікальні рівні, змінюючи параметри латентного простору. Проте стохастичність має і зворотний бік: результати

часто є “усередненими”, тобто рівні можуть втратити чітку структуру або виглядати надмірно згладженими.

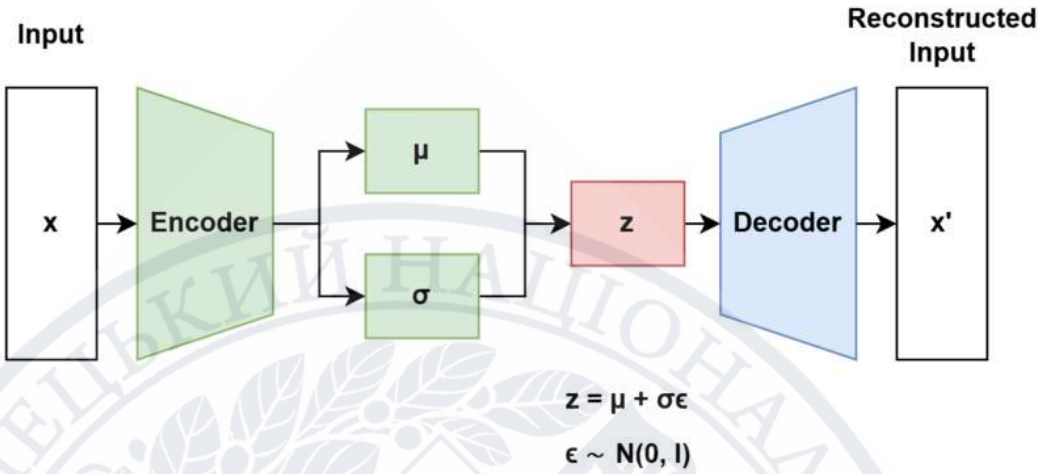

Рисунок 1.2 – Архітектура варіаційного автоенкодера (VAE)

Генеративно-змагальні мережі (GAN, Generative Adversarial Networks) працюють на принципі конкуренції двох моделей – генератора та дискримінатора. Генератор отримує на вхід випадковий вектор шуму z і намагається створити рівень, який виглядав би реалістично, тоді як дискримінатор оцінює, чи є цей рівень “справжнім” (тобто взятим із навчального набору) чи згенерованим. Обидві частини навчаються одночасно: генератор прагне “обманути” дискримінатор, а дискримінатор намагається розпізнати підробку [3][48].

На рисунку 1.3 зображено типову архітектуру GAN: дві мережі взаємодіють у зворотному зв’язку, утворюючи динамічний процес навчання. Цей підхід дозволяє створювати високореалістичні карти, але є дуже чутливим до якості даних та параметрів навчання. Невдала ініціалізація або занадто сильний дискримінатор можуть призвести до mode collapse – ситуації, коли генератор починає створювати лише один тип рівня [31].

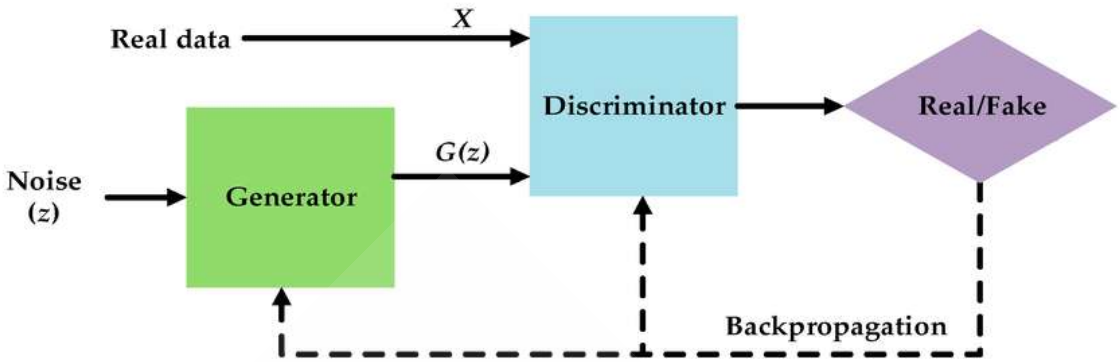

Рисунок 1.3 – Архітектура генеративно-змагальної мережі (GAN)

Умовні генеративні моделі (Conditional GAN, Conditional VAE) додають до процесу генерації керуючі параметри, що дозволяють цілеспрямовано впливати на результат. На рисунку 1.4 показано, що разом із вектором шуму генератор отримує додатковий вхід – умову (наприклад, розмір карти, тип середовища, кількість ворогів, координати входу чи виходу). Дискримінатор також приймає цю умову, перевіряючи, чи відповідає згенерований рівень заданим характеристикам [4] [13] [21][47].

Такий підхід забезпечує контрольовану генерацію, що є особливо важливим для геймдизайну [19][20], де необхідно утримувати баланс між варіативністю та іграбельністю. Недолік полягає у складності навчання: модель мусить одночасно зберігати узгодженість простору і відповідати зовнішнім умовам, що значно збільшує вимоги до обчислювальних ресурсів та якості розмітки датасету.

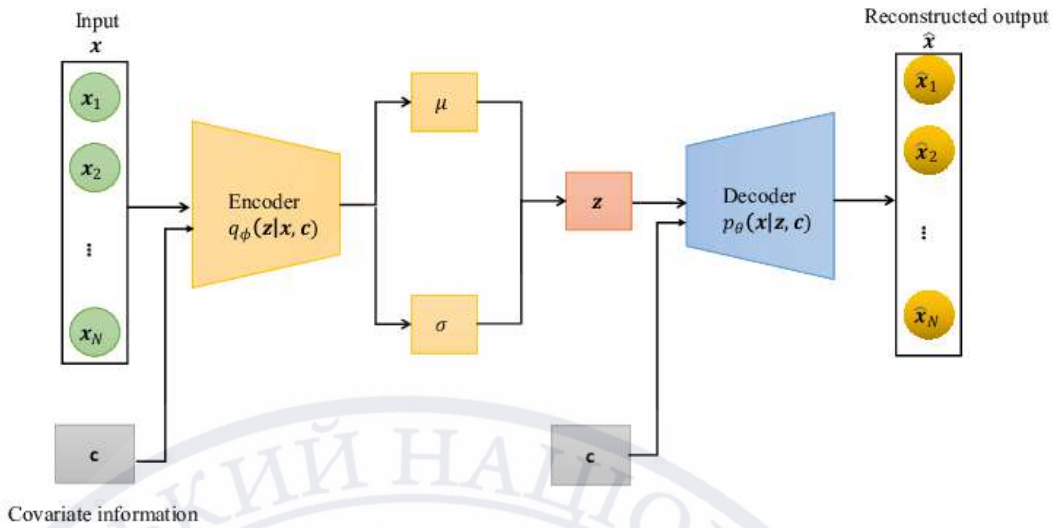

Рисунок 1.4 – Архітектура умовно-варіаційного автоенкодера (CVAE)

Рекурентні мережі (RNN, LSTM, GRU) розглядають рівень не як двовимірну структуру, а як послідовність дій або об’єктів – наприклад, “додати кімнату”, “розмістити ворога”, “поставити двері” (рис 1.5). Такий підхід дає змогу моделювати логіку створення рівня, ніби нейромережа “мислить” як дизайнер [38]. Головна перевага – здатність запам’ятовувати контекст, однак з цим пов’язана і головна слабкість: при занадто довгих послідовностях пам’ять мережі починає “забувати” попередні елементи, що призводить до логічних помилок у структурі карти.

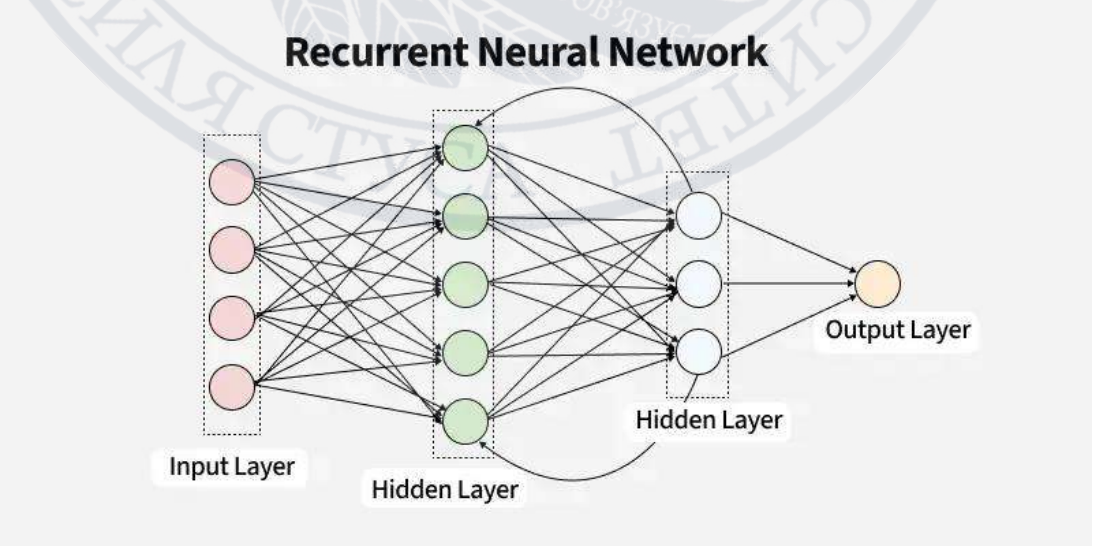

Рисунок 1.5 – Архітектура рекурентної нейронної мережі (RNN)

Трансформери (Transformers) є найсучаснішим підходом до генерації рівнів. Їх ключова перевага – механізм уваги (attention), який дозволяє враховувати зв’язки між усіма елементами карти одночасно, а не лише між сусідніми, як у RNN [10][39]. На рисунку 1.6 показано базову архітектуру трансформера, де кожен елемент входу порівнюється з усіма іншими, формуючи вагову матрицю залежностей. Завдяки цьому модель може розуміти, як різні частини рівня пов’язані між собою – наприклад, як розташування початкової точки впливає на розміщення фінального виклику.

Попри надзвичайну потужність, трансформери вимагають великих обсягів даних і значних обчислювальних ресурсів. Вони добре працюють при генерації великих складних карт, але можуть бути надлишковими для невеликих проєктів або обмежених датасетів.

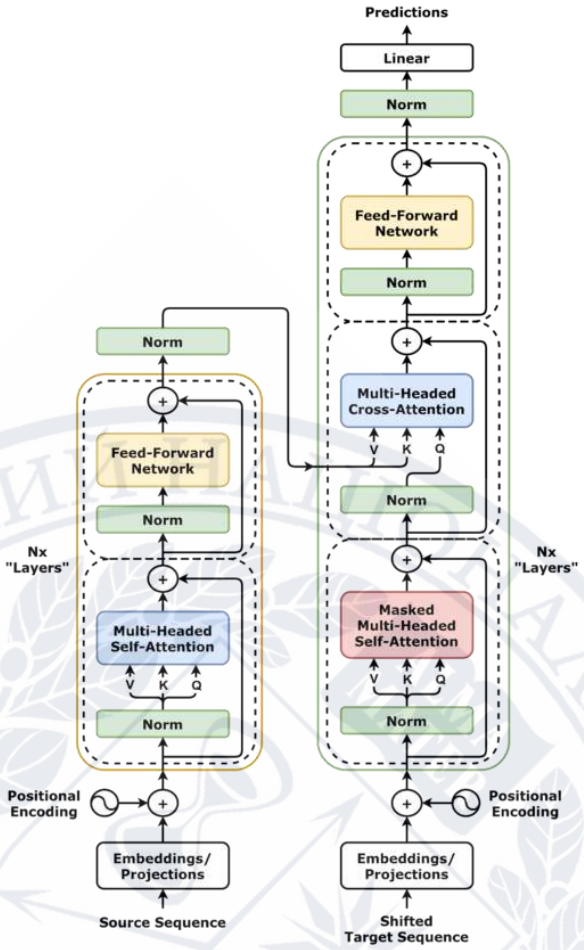

Рисунок 1.6 – Архітектура трансформера (Tranformer)

Таким чином, кожен тип нейронної мережі має свої переваги і недоліки залежно від цілей розробника. Автоенкодери підходять для аналізу структури, VAE – для стохастичної варіативності, GAN – для реалістичних результатів, Conditional моделі – для контрольованої генерації, RNN – для покрокового створення, а трансформери – для моделювання складних просторових залежностей. Вибір архітектури визначається не лише бажаним результатом, а й доступними ресурсами, типом даних та рівнем інтеграції в ігровий рушій.

1.3 Вибір технологій

Розробка системи генерації ігрових рівнів на основі штучного інтелекту вимагає комплексного підходу до вибору технологій, які забезпечують стабільність, гнучкість і сумісність між різними етапами розробки. Для

побудови та інтеграції такої системи було використано комбінацію інструментів, які поєднують середовище розробки ігрового простору, засоби машинного навчання та універсальні формати зберігання структурованих даних. До ключових компонентів належать рушій Unreal Engine 5, мова програмування Python з бібліотекою PyTorch, а також формат JSON (JavaScript Object Notation), який виконує роль проміжного рівня обміну даними між рушієм та нейронною мережею.

Unreal Engine 5 є сучасним рушієм для створення інтерактивних тривимірних середовищ, який завдяки своїй архітектурі дозволяє поєднувати високопродуктивне програмування на C++ із візуальним проєктуванням за допомогою Blueprint [22][40][43]. Це забезпечує можливість як швидкого прототипування, так і створення масштабних систем із глибокою оптимізацією. У межах поставленого завдання Unreal Engine виконує роль середовища, у якому результати роботи нейромережі перетворюються на об’єкти сцени, що мають геометричні, матеріальні та поведінкові властивості. Рушій також надає інтерфейси для інтеграції зовнішніх модулів, що дозволяє здійснювати двосторонній обмін даними з нейронною моделлю безпосередньо або через посередницькі файли.

Мова програмування Python у свою чергу виступає інструментом для проєктування, навчання та тестування нейронної мережі. Її синтаксична простота, розширюваність і наявність великої кількості бібліотек роблять Python основною платформою для досліджень у галузі машинного навчання. Зокрема, бібліотека PyTorch забезпечує гнучку побудову нейронних архітектур завдяки динамічному графу обчислень [7][41]. Це дозволяє під час навчання змінювати структуру мережі, аналізувати поведінку окремих шарів і швидко тестувати нові гіпотези. PyTorch також підтримує автоматичну оптимізацію параметрів моделі, обчислення похідних за допомогою механізму автодиференціювання та повноцінну роботу на графічних процесорах, що критично важливо при генеративних завданнях. Така гнучкість робить його

придатним не лише для навчання, але й для подальшої інтеграції моделі в ігровий рушій.

Формат JSON було обрано як основний засіб для зберігання та передачі інформації між середовищем Unreal Engine та Python. На відміну від CSV (Comma-Separated Values), який має лінійну структуру та підходить переважно для зберігання табличних наборів даних, JSON підтримує ієрархічну організацію, що є ключовою при описі складних структур – наприклад, ігрових рівнів, які складаються з множини об’єктів із власними координатами, матеріалами та додатковими параметрами [9]. Завдяки вкладеній структурі JSON забезпечує збереження логічних зв’язків між елементами сцени, дозволяючи передавати не лише геометрію, а й метадані. Крім того, цей формат легко обробляється як у Python, так і у рушії Unreal Engine, має мінімальний розмір і не вимагає спеціального програмного парсингу. Таким чином, JSON виступає ефективним універсальним контейнером для даних, що поєднує простоту текстового представлення з гнучкістю об’єктної моделі.

Окремим аспектом є інтеграція моделей у Unreal Engine. Серед можливих рішень існує плагін Neural Network Engine (NNE) [15], орієнтований на виконання моделей у форматах ONNX та TorchScript. Однак повна функціональність цього інструменту має комерційні обмеження. Альтернативою є використання Python API [6] або зовнішніх серверних інтерфейсів, але такі підходи ускладнюють роботу системи і не підходять для генерації в реальному часі. Тому у межах цієї роботи було обрано варіант із використанням ONNX Runtime — універсальної бібліотеки для виконання моделі у форматі ONNX напряму з C++, що забезпечує незалежність системи та достатню продуктивність [23][24].

1.4 Постановка задачі

У рамках цієї роботи ігровий рівень розглядається як структурована тривимірна сцена, що складається з множини акторів — геометричних

об’єктів, декоративних елементів, світлових джерел та інших компонентів, кожен із яких описується власним набором параметрів: типом мешу, матеріалом, масштабом, позицією, орієнтацією та додатковими властивостями. Сукупність цих параметрів утворює цифрову модель рівня, придатну для аналізу, перетворення та реконструкції засобами машинного навчання.

Основною задачею є побудова повного циклу генерації рівнів, який починається зі збирання даних у рушії Unreal Engine, їх експорту у форматі JSON та подальшої обробки в Python-середовищі. На цьому етапі JSON виступає універсальним носієм структурованої інформації, що дає змогу перетворювати рівні на навчальний набір, здійснювати попереднє опрацювання та формувати дані для подальшого використання моделлю. На основі цих даних Python-скрипт повинен мати можливість генерувати нові конфігурації рівня, які зворотно імпортуються в Unreal Engine і відтворюються у вигляді сцени.

У межах проєкту такий файловий обмін виступає проміжним етапом. Після реалізації базової генерації через JSON ставиться наступна ціль — експорт навченої моделі у формат ONNX та її інтеграція безпосередньо в Unreal Engine. Це дозволяє виконувати інференс нейронної мережі всередині рушія без необхідності звернення до зовнішніх скриптів або проміжних файлів, забезпечуючи швидшу й компактнішу систему.

Кінцевою метою роботи є створення інтерактивної механіки для гри, у якій гравцеві необхідно визначити, який із двох рівнів був створений нейронною мережею, а який — вручну. Для цього модель генерації має бути інтегрована у фінальний застосунок так, щоб нові рівні могли створюватися безпосередньо під час гри на основі ONNX-моделі. Таким чином, завдання роботи включає побудову повного циклу: від експорту сцени, підготовки навчального набору та генерування рівнів через JSON, до створення самостійної моделі, її конвертації у формат ONNX і подальшої інтеграції в Unreal Engine для використання в реальному ігровому сценарії.

РОЗДІЛ 2. РОЗРОБКА ТА ІНТЕГРАЦІЯ ГЕНЕРАТИВНОЇ НЕЙРОМЕРЕЖІ У UNREAL ENGINE 5

2.1 Створення системи запису та генерації рівня з JSON

Як зазначалося вище, рівень у межах даного проєкту трактується як обмежена область простору, у межах якої розташовані різноманітні актори, що утворюють композицію ігрового середовища. Для реалізації такого підходу було створено Blueprint-клас BP_Collision, який представляє собою просту колізійну область (box collision) без додаткової логіки. Цей клас виконує роль контейнера рівня або рамки меж, всередині якої розміщуються всі об’єкти, що належать до конкретного рівня..

Для забезпечення можливості експорту та імпорту рівнів у форматі JSON було розроблено набір UI-елементів на базі класу Utility Widget. На відміну від звичайних ігрових інтерфейсів, такі віджети призначені не для взаємодії з гравцем, а для допомоги розробнику під час створення або тестування рівнів.

Основними елементами системи є:

BP_Generate_Level_Widget – інтерфейс, що містить чотири текстові поля (TextBox) та кнопку. У полях вводяться: порядковий номер рівня (якщо у файлі міститься кілька рівнів), а також координати, за якими необхідно здійснити генерацію. Після натискання кнопки ініціюється процес імпорту рівня з JSON-файлу та його побудова у світі.

BP_Button_For_Extract – віджет, який містить кнопку для запуску процедури збереження поточних рівнів. Після активації виконується збір інформації про всі екземпляри BP_Collision, присутні на сцені, а також про всі актори, що знаходяться всередині кожного з них. Зібрані дані серіалізуються у структуру JSON і зберігаються у файл (рис. 2.1).

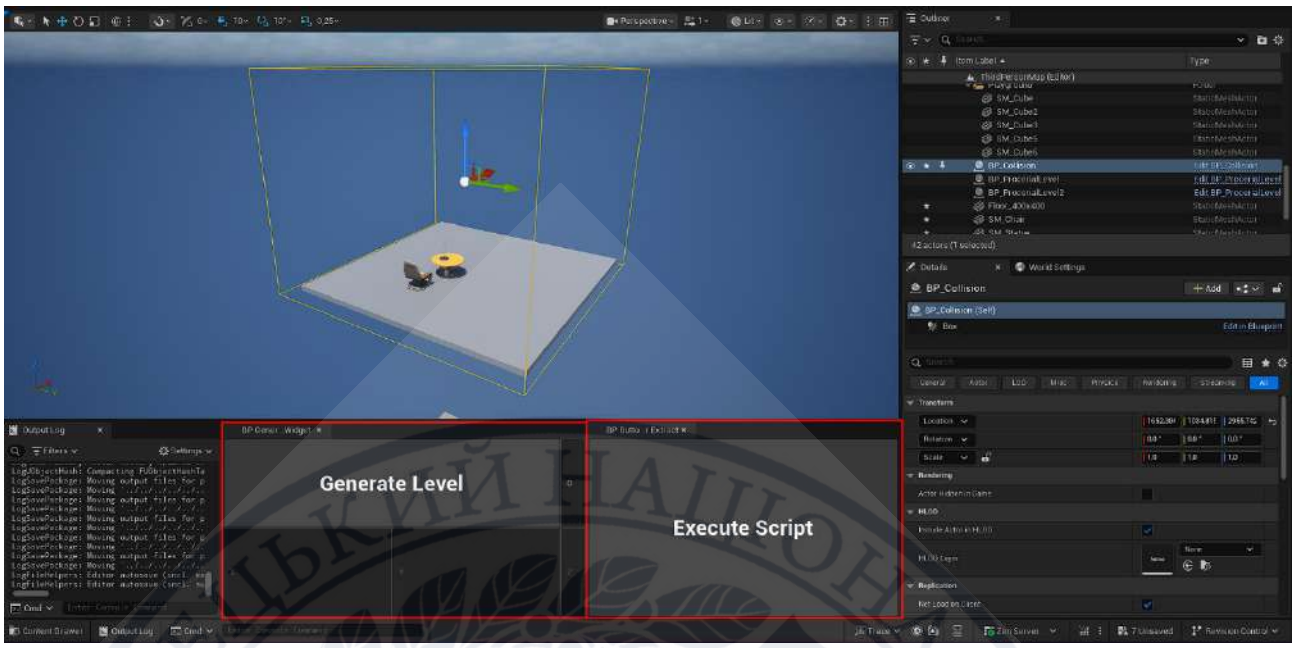

Рисунок 2.1 – зовнішній вигляд рівня та BP_Generate_Level_Widget та BP_Button_For_Extract

Щоб ці віджети могли виконувати свої функції, у проєкті було створено C++ клас UCreateDataset, який наслідує UBlueprintFunctionLibrary – клас, що дозволяє реалізувати набір функцій, доступних для виклику безпосередньо з Blueprint. Такий підхід дозволив поєднати гнучкість візуального середовища з продуктивністю C++ [28].

У класі UCreateDataset реалізовано три методи: два публічні (GenerateDataset і GenerateLevelAtLocation) та один приватний (SpawnLevel). Перший відповідає за створення датасету, другий – за генерацію рівня у світі, а третій виконує безпосереднє спавнення акторів на основі зчитаних параметрів.

Метод GenerateDataset виконує побудову датасету, збираючи інформацію про всі об’єкти в межах кожного BP_Collision. На початку роботи функція отримує шлях до класу-маркера BP_Collision, після чого здійснює пошук усіх його екземплярів у світі. Для кожного знайденого елемента перевіряється наявність акторів типу BP_In та BP_Out, що відповідають за вхід

і вихід із рівня. Якщо хоча б один із них відсутній, ця область не вважається повноцінним рівнем і пропускається, щоб уникнути некоректних даних у майбутньому датасеті.

Після проходження перевірки функція фіксує основні параметри рівня – висоту (height), довжину (length) та ширину (width). Також до файлу заносяться координати, обертання та масштаб елементів входу і виходу. Далі система переходить до збору інформації про всі інші актори всередині меж рівня. Для кожного з них записуються ключові характеристики:

- Type – шлях до класу актора, необхідний для його точного відтворення;
- Location, Rotation, Scale – положення, обертання та масштаб відносно рамок рівня;
- Mobility – тип мобільності: статичний (Static) або рухомий (Movable);
- bSimulatePhysics – булевий параметр, що визначає, чи взаємодіє об’єкт із фізичною системою світу.

Крім загальних параметрів, окремі типи акторів мають власні специфічні поля. Наприклад, для StaticMeshActor зберігаються посилання на mesh і material, що визначають геометрію та зовнішній вигляд об’єкта. Актори типу BP_ProceduralWall, які використовуються для побудови стін, мають додаткові змінні length і width, що описують їхні розміри.

Після збору всіх необхідних даних система формує структурований JSON-файл, який містить опис усіх рівнів, знайдених у сцені. Файл зберігається за шляхом Saved/Dataset/LevelDataset.json, де накопичуються результати для подальшого аналізу або використання нейронною мережею (рис. 2.2) [30].

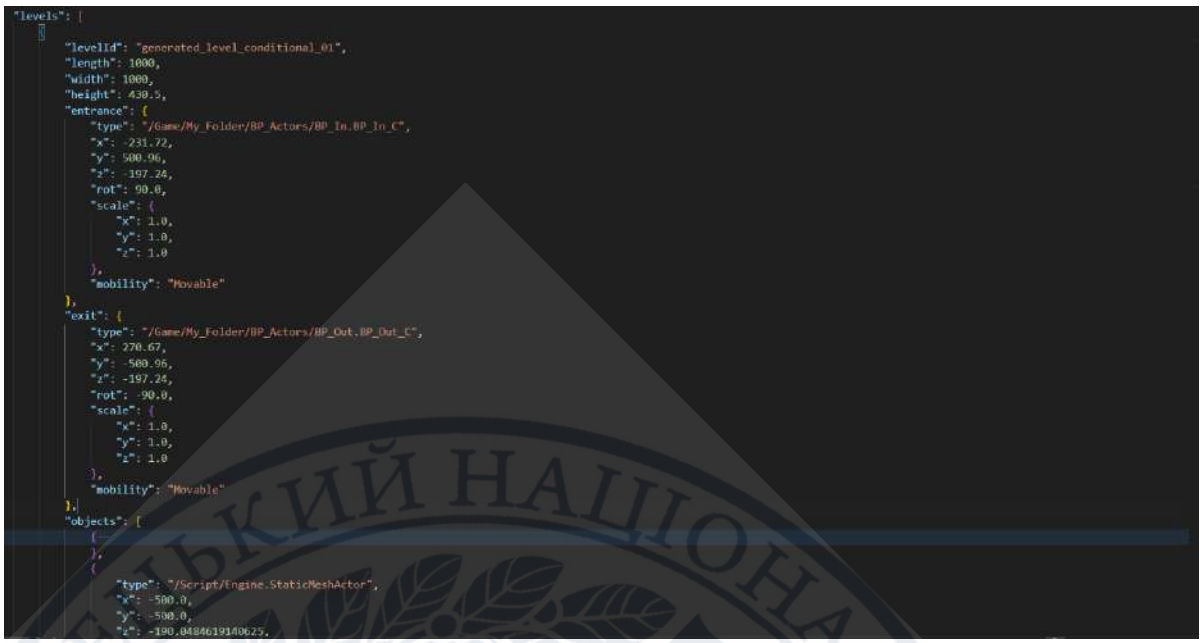

Рисунок 2.2 – приклад вигляду рівня у LevelDataset.json

Метод GenerateLevelAtLocation виконує зворотну задачу – зчитує дані з JSON-файлу та відтворює рівень у зазначеній позиції сцени. Він приймає три аргументи: контекст світу, порядковий номер рівня в датасеті та вектор координат, що визначає, де саме потрібно створити рівень. На основі отриманих даних функція через SpawnLevel створює кожен актор, відновлюючи його положення, обертання, масштаб, матеріал, фізичні властивості та інші параметри, збережені у файлі.

Завдяки реалізації цього класу Blueprint отримує доступ до нових функцій, які можна безпосередньо викликати у вже створених віджетах – BP_Generate_Level_Widget та BP_Button_For_Extract, що дозволяє керувати процесом збереження та відтворення рівнів без необхідності роботи з кодом (рис. 2.3).

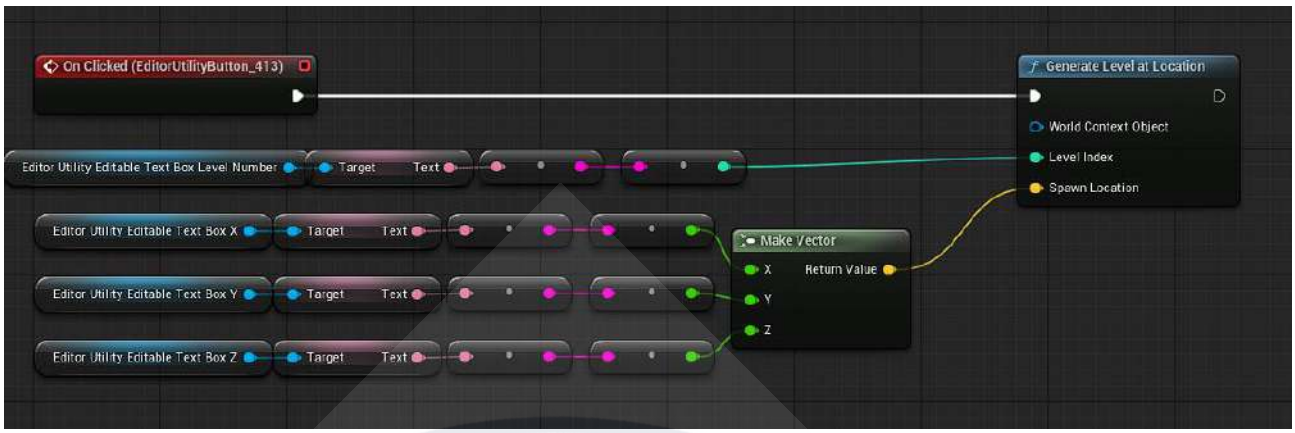

Рисунок 2.3 – підключення генерації рівня до віджету

2.2 Створення допоміжних акторів BP_In, BP_Out, BP_ProceduralWall

У проєкті рівень логічно розглядається як обмежена «коробка», всередині якої розташовані актори, декорації й ігрові точки. В такій моделі ключовою задачею стає коректне визначення точок входу і виходу – не просто маркерів, а фізично правдоподібних дверних прорізів, через які гравець або логіка переходів має проходити без артефактів. Просте рішення у вигляді одного великого меша-стіни з вирізаним отвором виглядає естетично, але технічно воно ускладнює задачу генерації: розтягування одного меша для отримання різних довжин призводить до спотворень, а навчати нейронну мережу розпізнавати й розміщувати кілька варіантів «стіни з отвором» (по суті – три-чотири різні моделі для кожного боку) – це зайве ускладнення і надмірне розширення простору ознак. Саме тому для маркування реальних дверних прорізів у сцені введено два простих допоміжних актора: BP_In та BP_Out (рис 2.4).

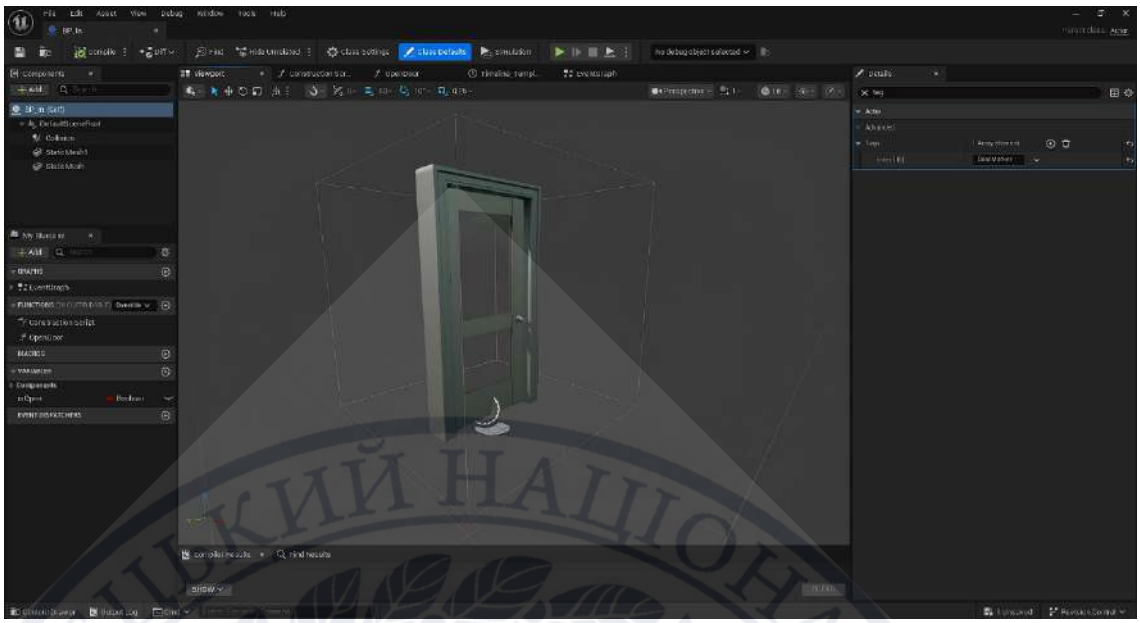

Рисунок 2.4 – BP_In

Актори BP_In та BP_Out – це два схожих класи, що різняться лише назвою, але мають однакову структуру та логіку. Вони складаються зі статичних мешів дверей і дверної рами, а також мають колізійну область навколо них. У налаштуваннях кожного з цих акторів задано спеціальний тег Doormaker, який пізніше використовується стіною BP_ProceduralWall як маркер, що саме у цьому місці потрібно створити дверний отвір.

Принцип роботи дверей доволі простий. Коли будь-який об’єкт перетинає або залишає межі колізійної області, відбувається перевірка – чи знаходиться актор гравця (BP_Third_Person_Character) всередині цієї зони. Якщо гравець присутній, двері починають плавно відкриватися, у протилежному випадку – плавно зачиняються. Для цього використовується нода Timeline, що дозволяє змінювати значення з плином часу, створюючи ефект плавного руху. Таким чином, коли перевірка проходить успішно, двері обертаються приблизно на 100 градусів по осі Z, а при виході гравця повертаються до вихідного положення.

Тепер оскільки у нас є дверні актори, треба якось додати зробити таку стіну, що зможе грамотно додати до них дверні отвори. Тому замість того щоб

вставляти постійно стіну що підходить, будемо генерувати її процедурно за допомогою вбудовного в Unreal Engine плагіну Procedural Mesh Component.

Procedural Mesh – це геометрія, що формується програмно з набору вершин, трикутників, нормалей, UV-координат та тангентів, замість того щоб завантажуватись із готової 3D-моделі (.fbx) [5][43]. Такий підхід дає змогу генерувати стіни будь-якої довжини та висоти, з дверними отворами в довільному місці. Завдяки цьому кількість варіацій об’єктів суттєво зменшується, а майбутній нейромережі потрібно буде враховувати менше параметрів – лише основні розміри стіни, що значно спрощує процес навчання.

Для реалізації цього підходу створено C++ клас CreateWall, який наслідує UBlueprintFunctionLibrary. Саме тут розміщено основну логіку побудови стін через функцію CreateWallWithDoor, що приймає такі параметри:

- UProceduralMeshComponent* ProcMesh – ProceduralMeshComponent компонент, у який створюється геометрія.
- float WallLength – довжина стіни.
- float WallHeight – висота стіни.
- float WallThickness – ширина стіни.
- float DoorCenterX – значення координати по довжині стіни, у якому буде стояти центр стіни.
- float DoorWidth – довжина двірного отвору.
- float DoorHeight – висота двірного отвору.
- bool bCreateCollision – прапорець, який визначає, чи потрібно створювати колізію (за замовчуванням – true).

У класі також створено допоміжну структуру, яка зберігає всі геометричні дані – позиції вершин, індекси трикутників, нормалі, UV-координати та тангенти. Окрім цього, є допоміжна функція BuildQuad, яка будує одну грань із чотирьох векторів, обчислює нормалі, UV-координати та формує з них два трикутники. Параметр UVScale визначає масштаб текстури

(за замовчуванням 100), що дозволяє уникнути розтягування матеріалів при зміні розмірів стіни.

Під час виконання CreateWallWithDoor спочатку перевіряється, чи передано коректний ProceduralMeshComponent. Якщо так – функція очищає попередню геометрію, щоб уникнути накладання текстур, після чого розраховує координати вершин для створення стіни. Якщо дверний отвір не задано, будується звичайна суцільна стіна з шести граней. Якщо ж параметри дверей вказані, то формується набір граней з урахуванням вирізу: три фронтальні та три тильні грані, бокові поверхні, а також внутрішні частини прорізу.

Щоб використовувати створену функцію безпосередньо в редакторі, було створено Blueprint-клас BP_ProceduralWall. Він містить ProceduralMeshComponent – саму геометрію стіни, та BoxCollision, який виступає у ролі сенсора. У класі визначено дві публічні змінні – Length і Height, що задають основні параметри стіни. Для побудови використовується функція BuildWall, яка викликає CreateWallWithDoor і передає до неї відповідні параметри. Позиція дверного прорізу розраховується автоматично, а його розміри були підібрані експериментальним шляхом для точної відповідності дверям BP_In та BP_Out.

Функція BuildWall викликається у розділі Construction Script, що дозволяє побачити створену стіну прямо в редакторі, ще до запуску гри. Тут також реалізована функція MakeSensor, яка підлаштовує розміри колізії під параметри стіни, а також використовується SetMaterial для відображення матеріалу на створеній поверхні.

У EventGraph реалізовано подію OnComponentBeginOverlap, що реагує на перетин сенсора іншими об’єктами. Під час активації події перевіряються всі актори, які потрапили в межі колізії, і якщо в них є тег Doormaker, система визначає положення дверей. Для цього використовується функція InverseTransformLocation, яка дозволяє обчислити координату X центру дверного прорізу. Це значення передається функції BuildWall, яка будує стіну

з дверним отвором у правильному місці, узгодженому з позицією дверей BP_In або BP_Out.

Таким чином, створена система дозволяє процедурно генерувати стіни потрібного розміру та автоматично додавати дверні отвори у відповідних місцях, забезпечуючи більшу гнучкість під час генерації рівнів та зменшуючи кількість унікальних об’єктів, необхідних для навчання нейромережі (рис 2.5).

Рисунок 2.5 – BP_ProceduralWall та BP_In

2.3 Процедурна генерація навчального датасету

Під час навчання нейронної мережі, що займається генерацією рівнів, одним із найважливіших етапів є підготовка великого та різноманітного датасету. Саме якість і різноманіття навчальних даних визначають, наскільки модель зможе узагальнювати просторові закономірності, правильно розміщувати актори та відтворювати логічну структуру рівня. Однак отримати готовий датасет із нашими специфічними вимогами – зокрема, використанням власних класів BP_ProceduralWall, BP_In, BP_Out, а також збереженням усіх акторів рівня у структурованому форматі – практично неможливо. Створення

ж великої кількості рівнів вручну є надзвичайно трудомістким процесом, що виходить за межі можливостей у рамках даної роботи.

Тому було прийнято рішення реалізувати автоматизований процес генерації простих рівнів, який би дозволяв швидко отримувати велику кількість навчальних прикладів. Такий підхід не лише значно спрощує підготовку датасету, а й забезпечує контроль над усіма параметрами структури рівня. За процедурну генерацію відповідає спеціальний актор – BP_ProcedurialLevel, який виступає центральною сутністю у створенні базових рівнів для подальшого аналізу.

BP_ProcedurialLevel являє собою коробчасту колізію з низкою публічних параметрів, які описують майбутній рівень. Зокрема:

- BP_In_Wall – Параметр типу Enumerator WallType (що був власноруч створений). Даний параметр вказує на тип стінки до якого буде доданий BP_In (а саме Front, Right, Back, Left).
- BP_Out_Wall – Такий самий параметр що до BP_In_Wall, однак замість BP_In обирається BP_Out.
- LevelType – Тип рівня (integer).
- X High Range – найбільше значення яке може набувати розмір рівня по X координаті.
- X Low Range – Найменше значення яке може набувати розмір рівня по X координаті.
- Y High Range – найбільше значення яке може набувати розмір рівня по Y координаті.
- Y Low Range – Найменше значення яке може набувати розмір рівня по Y координаті.
- Z High Range – найбільше значення яке може набувати розмір рівня по Z координаті.
- Z Low Range – Найменше значення яке може набувати розмір рівня по Z координаті.

У Construction Script актора створюється початкова візуалізація рівня у вигляді коробчастої колізії. За допомогою функції Random Float in Range генеруються випадкові розміри по кожній осі в межах заданих параметрів. Отримані значення передаються у Set Box Extend, що дозволяє одразу побачити у редакторі, який розмір матиме рівень після генерації. Це суттєво спрощує візуальне налаштування та дає змогу уникати взаємного перекриття кількох рівнів.

Попри те, що логічним кроком було б виконати всю генерацію безпосередньо в Construction Script, це рішення має серйозне обмеження. Дана частина Blueprint не підтримує створення акторів у процесі роботи редактора. Це зроблено навмисно, адже Construction Script викликається при кожній зміні параметрів або навіть під час простого переміщення актора у сцені. У результаті це призвело б до масового створення нових об’єктів при кожному русі мишею, що швидко зруйнувало б структуру сцени.

Існує також технічна альтернатива – створення елементів як компонентів усередині BP_ProcedurialLevel, однак це значно ускладнює архітектуру, робить об’єкт надмірно важким і не дозволяє легко обробляти ці елементи в межах функції GenerateDataset, яка зчитує лише дані про актори, а не їхні внутрішні компоненти. Саме тому було прийнято остаточне рішення реалізувати генерацію рівня в Event Graph, де спавн акторів виконується лише під час запуску симуляції. Це гарантує стабільність роботи редактора та забезпечує можливість збереження всіх параметрів рівня лише після його повного формування.

Побудова рівня починається з формування його каркасу – базової структури, на основі якої згодом створюватиметься повноцінний ігровий простір. Спершу генерується головний контейнер – актор BP_Collision, який відповідає за фізичні межі рівня. Його створення відбувається через функцію SpawnActor, де задається трансформація, отримана з GetRelativeTransform. Завдяки цьому BP_Collision завжди повністю охоплює весь простір рівня, і надалі всі інші актори можуть орієнтуватися на його розміри та позицію.

Після цього створюється підлога – звичайний StaticMeshActor, який виконує роль базової поверхні рівня. Для визначення його точного розташування використовується функція GetWorldTransform коробчастої колізії, а для масштабування – GetBoxExtent. Це дозволяє точно позиціонувати підлогу в нижній межі коробки, з правильним масштабом по осях. Після створення актору задається стандартний меш “Floor_400x400”, який є частиною рушія Unreal Engine, а також прикріплюється базовий матеріал M_Basic_Floor, щоб зробити підлогу візуально відмінною від стін.

Наступним етапом є побудова стін, представлених акторами BP_ProceduralWall. Ці стіни мають власну логіку побудови геометрії, що дозволяє їм підлаштовуватися під задані розміри рівня. Їхні центри координат знаходяться у нижній частині мешу, тому при спавні важливо точно розрахувати позицію, щоб стіни розташовувалися по межах коробчастої колізії, утворюючи замкнений простір. Для цього знову використовуються функції GetWorldTransform і GetBoxExtent, що дозволяють визначити необхідні координати для кожної з чотирьох стін.

При цьому довжина та висота кожної стіни передаються у вигляді публічних параметрів актора, тому при створенні кожної BP_ProceduralWall у функції SpawnActor можна одразу задати правильні значення Length та Height. Для бічних стін додатково застосовується обертання, щоб вони розташовувалися перпендикулярно до фронтальної та задньої частини рівня. Завдяки цьому досягається точне вирівнювання геометрії та узгодженість масштабу.

Після формування основного каркасу створюються двері – актори BP_In та BP_Out, що виконують роль точок входу та виходу. Щоб уникнути дублювання коду, для їхнього створення розроблена спеціальна функція SpawnDoors, яка приймає три ключових параметри: булеве значення Is_Out, координату X та тип стіни WallType. На основі цих параметрів визначається, який саме клас актора буде створено (вхід чи вихід), і на якій із чотирьох стін він має бути розміщений. Якщо координата X не задана вручну, двері

позиціонуються випадково, що дозволяє створювати більшу варіативність рівнів. Якщо ж координата задана, двері встановлюються у фіксованій позиції, що може бути корисно для контрольованих експериментів під час тренування нейромережі.

Для кожної сторони – Front, Right, Back, Left – у системі передбачено власні математичні розрахунки позицій розміщення, які враховують не лише розміри колізійного боксу, а й орієнтацію актора дверей у просторі. Завдяки цьому BP_In та BP_Out спавняться у коректних місцях із точними параметрами трансформації, що забезпечує правильну геометрію та логіку рівня.

Після цього алгоритм переходить до вузла Switch, який визначає тип майбутнього рівня. Якщо параметр LevelType дорівнює -1, обирається випадковий шаблон заповнення. Наразі реалізовано два базові варіанти рівнів: тип 0 – невеликий простір із невисокою перегородкою, яку можна подолати за допомогою фізичного об’єкта; тип 1 – розширена версія першого варіанту, із більшою перегородкою та трьома фізичними об’єктами, що дозволяють гравцеві побудувати сходи для переходу через неї.

Для 0-го рівня створюється StaticMeshActor із мешем Wall_400x200, що виконує роль перегородки. Її позиція визначається через GetWorldTransform, GetBoxExtend та додаткові математичні обчислення. Щоб уникнути монотонності, перегородка розміщується не строго по центру, а з невеликим випадковим зміщенням у межах коробки – іноді ближче до входу, іноді ближче до виходу. Це забезпечує легку варіативність сцен.

Після цього обчислені координати передаються у логіку, що спавнить об’єкт для перестрибування перегородки. Тут важливо зберегти баланс: актор має створюватися у випадкових, але логічно допустимих межах, щоб не з’явитися за перегородкою, інакше гравець фізично не зможе подолати перешкоду. Далі через вузол Switch випадковим чином обирається меш для цього об’єкта – куб (Cube), циліндр (Shape_Cylinder) або клиноподібна

сходинка (Shape_Wedge_B). Усі ці варіанти дозволяють гравцеві пройти рівень, додаючи при цьому різноманіття візуальної структури.

Для таких об’єктів обов’язково вмикається мобільність (Mobility → Movable) і фізична симуляція (Set Simulate Physics = true). На завершення до них застосовується матеріал MI_Solid_Blue, що контрастує зі стінами і сигналізує гравцеві про можливість взаємодії.

Створення рівня типу 1 має схожий алгоритм, однак перед початком формується перевірка відповідності розмірів. Потрібно переконатися, що сцена має достатньо місця для трьох фізичних об’єктів і що висота перегородки відповідає заданим параметрам. У протилежному випадку можуть виникнути помилки: наприклад, якщо перегородка буде занадто низькою – гравець зможе легко перестрибнути межі рівня і впасти за мапу, а якщо рівень виявиться надто вузьким – об’єкти не матимуть достатньо простору, щоб сформувати прохід або побудувати сходи. Якщо перевірка не пройдена, система автоматично генерує рівень типу 0. В іншому випадку створюється новий рівень: перегородка має висоту приблизно у півтори рази більшу, ніж у попередньому типі, а замість одного об’єкта для перестрибування спавняються три.

На цьому етапі випадковий вибір мешів частково обмежується – щоб уникнути ситуації, коли всі три об’єкти виявляться однаковими клинами (Shape_Wedge_B) і гравець не зможе побудувати прохідну конструкцію. Тому лише останній елемент є сходинкою, а два попередніх випадковим чином обираються між циліндром і кубом.

У підсумку формується набір простих, але варіативних процедурних рівнів (рис. 2.6). Оскільки ці сцени створюються під час симуляції, їх неможливо одразу зберегти через віджет BP_Button_To_Extract. Для вирішення цього завдання у граф рівня додається логіка, яка викликає раніше реалізовану функцію GenerateDataset. У прикладі її запуск відбувається під час натискання клавіші “1”, що дозволяє записати всі дані навіть у процесі симуляції.

Рисунок 2.6 – результат роботи BP_ProceduralLevel

Таким чином, розробник може розмістити на сцені сотні, а то й тисячі таких процедурних рівнів, запустити симуляцію та натиснути одну клавішу для збору великого, хоч і не надто різноманітного, але цілком робочого датасету, який у подальшому буде використано для навчання нейронної мережі.

2.4 Навчання нейроної мережі

Навчання нейронної мережі здійснюється поза межами рушія Unreal Engine, із використанням мови програмування Python та бібліотеки PyTorch, що є стандартом для розробки й дослідження сучасних моделей глибинного навчання. На поточному етапі взаємодія між рушієм та нейронною мережею реалізована у вигляді обміну JSON-файлами. З боку рушія створюється навчальний датасет, який зберігається у форматі JSON і передається до Python-скриптів для подальшої обробки та навчання моделі. У зворотному напрямку,

вже після завершення навчання, нейронна мережа генерує власний JSON-файл із описом нового рівня, який потім інтерпретується в Unreal Engine та відтворюється у вигляді повноцінної сцени.

Для навчання було сформовано датасет обсягом 2048 рівнів, що були згенеровані за допомогою актора BP_ProceduralLevel на порожній сцені рушія. Даної кількості згенерованих рівнів було достатньо для отримання оптимального результату генерування моделі.

Після формування навчального набору даних було обрано архітектуру умовного варіаційного автоенкодера (Conditional Variational Autoencoder, CVAE). Даний тип нейронних мереж поєднує властивості класичного автоенкодера та генеративної моделі, що дозволяє не лише відновлювати структуру даних, а й керувати процесом генерації через задання певних умов (розмірів рівня, позицій входу та виходу тощо). Це робить CVAE природним вибором для задачі створення ігрових рівнів, де важливо забезпечити не випадковість, а керованість та узгодженість результату.

Окрім цього, через високу одноманітність датасету (обмежена варіативність простору і розташування об’єктів) використання складніших архітектур, таких як GAN (Generative Adversarial Network), не є доцільним [33]. GAN-моделі потребують великого обсягу різноманітних даних для стабільного навчання та часто демонструють нестійку збіжність при роботі з малими або однорідними вибірками. У випадку даного проєкту основною метою є не досягнення фотореалістичності рівнів, а створення функціональної системи, здатної узагальнювати структуру рівня та відтворювати його у нових комбінаціях. Саме тому CVAE є оптимальним компромісом між складністю реалізації та контрольованістю генерації.

Сам процес тренерування нейроної мережі складається з трьох основних етапів, реалізованих у трьох окремих Python-скриптах:

- reprocess.py – підготовка та попередня обробка даних;
- train.py – створення та навчання моделі на оброблених даних;

– generate.py – генерація нового рівня на основі навчених параметрів моделі.

Першим етапом навчання є підготовка даних, яка реалізована у скрипті preprocess.py. Основне завдання даного етапу полягає у перетворенні необробленого датасету, сформованого в рушії Unreal Engine, у числову форму, придатну для подачі в нейронну мережу.

Після завантаження JSON-файлу скрипт виконує обробку у два послідовні проходи.

Під час першого проходу створюються словники категоріальних ознак (vocabularies) для типів об’єктів, параметрів мобільності, назв статичних мешів і матеріалів. Кожному унікальному значенню призначається власний числовий індекс, який у подальшому використовується для формування one-hot векторів. Одночасно з цим етапом виконується збір усіх числових характеристик (позицій, масштабу, фізичних параметрів, розмірів стін), що необхідні для подальшої нормалізації за допомогою скейлерів.

На другому проході використані скейлери дозволяють нормалізувати числові параметри об’єктів, після чого формується фінальний набір тензорів PyTorch, де кожен рівень представлено у вигляді сукупності ознак обмеженої кількості об’єктів. Максимальна кількість об’єктів у межах одного рівня задається константою MAX_OBJECTS.

Оскільки архітектура CVAE не є рекурентною, модель не здатна створювати нові об’єкти динамічно після кожного кроку генерації. Тому для кожного потенційного об’єкта заздалегідь формується вектор ознак, а для визначення необхідності його існування додається булева змінна exist. Якщо кількість реальних об’єктів менша за MAX_OBJECTS, параметр exist для решти елементів дорівнює нулю. Такий підхід забезпечує фіксовану структуру вхідних та вихідних даних при збереженні гнучкості кількості згенерованих об’єктів.

Важливою особливістю даного етапу є дискретизація координат об’єктів. Замість безперервних значень позицій використовується сітка

розміром GRID_SIZE = 20. Таким чином, кожен об’єкт може бути розміщений лише у певній клітинці сітки, що значно спрощує процес навчання моделі.

Завдяки цьому нейронна мережа не намагається досягти надмірної точності в позиціонуванні елементів, а оперує дискретними координатами, що особливо важливо для об’єктів, розташованих на крайніх межах рівня (наприклад, стін або підлоги). Подібним чином реалізовано й дискретизацію обертання: замість подання кута як безперервної величини, він розбивається на 24 стани з кроком у 15°. Це дозволяє уникнути неоднозначностей при роботі з обертаннями (наприклад, між 0° та 359°) і спрощує простір ознак, не впливаючи на якість відтворення.

У результаті роботи preprocess.py формується стандартизований та нормалізований набір даних, який є оптимізованим для навчання моделі. На цьому етапі хаотичні дані з рушія перетворюються у впорядковану структуру, яка зберігає семантику рівня, але усуває надлишкову деталізацію, що заважала б процесу генерації. Саме на базі цих даних у подальшому виконується навчання моделі у скрипті train.py, що формує узагальнене представлення ігрових рівнів для їх подальшої генерації.

Після завершення попередньої обробки даних розпочинається другий етап – навчання нейронної мережі, реалізований у скрипті train.py. Основна мета цього етапу полягає у створенні та тренуванні моделі умовного варіаційного автоенкодера (Conditional Variational Autoencoder, CVAE), який здатен узагальнювати структуру ігрових рівнів і генерувати нові, логічно узгоджені конфігурації об’єктів. Архітектура CVAE у цьому проєкті складається з двох ключових компонентів – енкодера та декодера. Енкодер аналізує повну інформацію про рівень, включаючи його розміри (довжину, ширину, висоту) та характеристики кожного об’єкта (координати, тип, матеріал, масштаб тощо), і перетворює ці дані у компактне числове представлення – латентний вектор (у даному випадку розмірністю 512 нейронів). Під час цього процесу мережа навчається обчислювати два параметри – середнє значення (μ) та дисперсію (σ^2), що описують розподіл

латентного простору. Для забезпечення стабільного стохастичного навчання застосовується процедура репараметризації, яка дозволяє моделі зберігати градієнтну зв’язність навіть при випадковій вибірці латентних векторів.

Декодер у свою чергу отримує на вхід базову інформацію про геометрію рівня (довжину, ширину, висоту), параметри входів і виходів, а також створений енкодером стислий опис – латентний вектор. Використовуючи ці дані, декодер реконструює структуру рівня, відновлюючи позиції, типи та властивості об’єктів, які мають сформувати цілісну ігрову сцену. Таким чином, CVAE не просто відтворює наявні зразки, а навчається виявляти закономірності між об’єктами та узагальнювати їх у нових комбінаціях.

Для оцінки точності навчання у моделі використовується функція втрат, що складається з кількох компонентів. Зокрема, для коректної обробки змішаних типів даних застосовуються два підходи: CrossEntropyLoss для категоріальних параметрів (тип об’єкта, матеріал, меш тощо) і MSELoss для неперервних числових параметрів (позиції, масштаб, ротація). Кожен параметр має власну вагу у функції втрат, що дозволяє регулювати вплив окремих характеристик на процес навчання. Під час проходження батчу даних функція розраховує втрати для кожного з об’єктів, враховуючи не лише правильність відтворення, але й існування самого об’єкта. Оскільки модель здатна генерувати до десяти об’єктів (визначено параметром MAX_OBJECTS), навіть якщо реальна сцена містить менше, для уникнення хибного штрафування створюється спеціальна маска існування. Вона відключає обчислення втрат для тих об’єктів, які у вихідних даних відсутні (означені параметром exist = 0).

Крім основних втрат, у моделі присутній ще один важливий компонент – регуляризаційний член у вигляді дивергенції Кульбака–Лейблера (Kullback–Leibler Divergence, KLD), який є невід’ємною частиною архітектури варіаційних автоенкодерів. Цей додатковий елемент контролює розподіл латентного простору, запобігаючи його деградації та забезпечуючи плавне й узгоджене формування латентних представлень рівнів.

У зв’язку з тим, що навчання моделі здійснюється на графічному процесорі AMD, стандартна версія бібліотеки PyTorch із підтримкою CUDA виявилася несумісною. Для вирішення цієї проблеми у проєкті використано альтернативну бібліотеку torch_directml, яка функціонує через API DirectML та забезпечує можливість використання обчислень на GPU навіть без апаратної підтримки CUDA [8]. Це дозволило значно прискорити процес навчання, проте водночас наклало низку технічних обмежень. Зокрема, torch_directml не підтримує функцію BCEWithLogitsLoss, яка зазвичай використовується для бінарних ознак (таких як наявність об’єкта або параметр симуляції фізики), а також має проблеми сумісності зі стандартним оптимізатором Adam. Через це під час розробки було обрано оптимізатор RMSprop, який, попри дещо повільнішу збіжність, показав стабільну роботу й забезпечив належну якість навчання за обмежених обчислювальних можливостей.

Процес навчання моделі побудовано у кілька етапів. Спершу програма зчитує попередньо створені скриптом preprocess.py файли, визначає розмірність вхідного та вихідного шару нейронної мережі й готує дані до подачі в модель. Розмірність вхідного шару (умовного вектора) становить 17 елементів, що включають базові параметри рівня (довжина, ширина, висота) та ознаки вхідних і вихідних об’єктів. Вихідний шар має розмірність 4480 нейронів, що обумовлено сітковим представленням координат (20×20 для кожного з десяти потенційних об’єктів), а також додатковими параметрами, такими як ротація, масштаб, тип об’єкта, матеріал, мобільність та фізичні властивості. Таким чином, модель здатна одночасно генерувати повний набір ознак для кожного з десяти можливих акторів рівня.

Процес навчання починається з фази реконструкції, під час якої мережа навчається відновлювати структуру рівнів без урахування стохастичної складової. Протягом перших 1000 епох коефіцієнт ваги для дивергенції Кульбака–Лейблера (KLD) дорівнює нулю, тому модель фокусується виключно на відтворенні початкових даних. Після цього вага поступово

збільшується до 0.0025 до 5000-ї епохи, що дозволяє збалансувати реконструкційну точність і регуляризацію латентного простору. Навчання триває до 10000-ї епохи, після чого фінальна версія моделі зберігається у файл cvae_model.pth. Ця модель виступає основою для подальшої генерації рівнів на наступному етапі.

Заключний етап процесу реалізовано у скрипті generate.py, який відповідає за створення нового JSON-файлу рівня на основі навченого автоенкодера. На початку роботи скрипт завантажує збережену модель (використовуючи лише частину декодера), а також усі допоміжні файли – словники категорій об’єктів і скейлери нормалізації, сформовані на етапі попередньої обробки даних. Далі користувач задає основні параметри майбутнього рівня: довжину, ширину, висоту та положення елементів входу й виходу. Ці дані перетворюються у вектор умов, який подається на вхід декодера разом із латентним вектором z, що генерується випадковим чином.

Після проходження даних через декодер модель формує багатовимірний вихідний вектор, який містить інформацію про кожен потенційний об’єкт рівня – його тип, координати, обертання, масштаб, матеріал та інші властивості. Потім результати проходять етап денормалізації за допомогою завантажених скейлерів, перетворюються у зручний формат і записуються у JSON-файл, сумісний із рушієм Unreal Engine. Отриманий файл можна розмістити у каталозі Saved/Dataset проєкту, після чого система зчитує його та автоматично відтворює сцену за згенерованими параметрами.

У результаті роботи скрипта формується повноцінний рівень, створений нейронною мережею без участі людини (рис. 13).

Рисунок 2.7 – Створений нейромережею рівень.

Хоча згенеровані рівні ще не мають ідеальної геометрії, вони демонструють коректну логічну структуру, що підтверджує здатність моделі узагальнювати та відтворювати ключові закономірності побудови сцени. Отримані результати свідчать про працездатність запропонованого підходу та закладають основу для подальшого вдосконалення системи, зокрема шляхом розширення датасету та адаптації архітектури мережі до складніших сценаріїв генерації.

2.5 Експорт моделі з ONNX Runtime в Unreal Engine

У попередніх розділах було продемонстровано, що нейронна мережа успішно генерує працездатні ігрові рівні, однак сам процес генерації на цьому етапі реалізовано у вигляді запуску Python-скриптів, які створюють JSON-файли, що пізніше імпортуються у рушій Unreal Engine для побудови сцени.

Подібний підхід є прийнятним для тестових експериментів, проте він не забезпечує потрібної інтерактивності та швидкодії. Зокрема, такий механізм не дозволяє виконувати генерацію рівнів у режимі реального часу під час симуляції гри, що робить його непридатним для інтеграції у повноцінний ігровий процес. Для усунення цих обмежень необхідно перенести модель безпосередньо в C++-частину Unreal Engine, що потребує експорту вже навченої нейронної мережі у формат, сумісний із рушієм.

Одним з найпоширеніших стандартів для представлення нейромереж поза межами Python є формат ONNX (Open Neural Network Exchange). Використання цього формату дозволяє інтегрувати модель у різні середовища, включно з C++, та виконувати її безпосередньо під час роботи програми. У межах даного проєкту ONNX використовується разом із бібліотекою ONNX Runtime, яка забезпечує ефективне виконання нейронних мереж у Unreal Engine, не вимагаючи наявності Python або сторонніх інтерпретаторів.

Для експорту декодера CVAE-моделі було створено окремий скрипт export_onnx_decoder.py. Оскільки у процесі генерації рівнів використовується виключно декодер (а енкодер необхідний лише на етапі навчання), скрипт імпортує лише клас декодера з generate.py, а також пов’язані з ним параметри — розмірність латентного простору, максимальну кількість об’єктів, розміри умовного вектора та шляхи до файлів скейлерів і словників, сформованих на попередніх етапах.

Для підготовки моделі до експорту у скрипті реалізовано функцію build_decoder, яка відповідає за завантаження словників, зчитування нормалізаційних скейлерів та створення екземпляра декодера з урахуванням усіх параметрів CVAE-моделі. Константа MAX_OBJECTS визначає кількість об’єктів, яку декодер має потенційно відтворювати, що дозволяє сформувати коректну розмірність вихідного вектора. Після ініціалізації усіх параметрів модель переходить до основного етапу експорту.

У функції main здійснюється безпосереднє перетворення моделі у формат ONNX за допомогою стандартної функції torch.onnx.export. Під час

експорту задається екземпляр моделі, тестові вхідні дані для побудови структури графа, шлях для збереження файлу, а також імена вхідних та вихідних тензорів. У результаті формується файл декодера у форматі .onnx, який можна безпосередньо використовувати в Unreal Engine через ONNX Runtime.

Після експорту моделі виникає потреба адаптувати й допоміжні дані, необхідні для правильної роботи декодера. Оскільки скейлери збережено у форматі .pkl, який не може бути прочитаний у C++, було створено додатковий скрипт export_scalers.py. Його завдання полягає у зчитуванні параметрів скейлерів із бінарного файлу та подальшому перетворенні їх у формат JSON. Це дозволяє легко й коректно застосовувати нормалізацію та денормалізацію в C++-коді Unreal Engine, що є необхідною умовою для узгодженого функціонування мережі.

Наступним кроком буде підключення бібліотеки до проекту Unreal Engine, щоб надалі за допомогою C++ можна було б використовувати onnx модель. Для цього було завантажено бібліотеку ONNX Runtime та збережено у теці ThirdParty/onnxruntime. Надалі треба підключити дану бібліотеку у спеціальному скрипті GeneratingAI.Build.cs, що створюється автоматично при створені проекту, і що являє собою скрипт, що описує як збирати проект. У конструкторі даного класу для підключення бібліотеки були використані функції Path.GetFullPath та Path.Combine для отримання директорії бібліотек, а також PublicAddtionalLibraries.Add та RuntimeDependencies підключаємо бібліотеки з вказаного шляху, і додаємо правило щоб бібліотека підключалась у фінальну збірку із готовим .exe файлом.

Після успішного підключення ONNX Runtime у проєкті було реалізовано окремий C++ клас FONNXRunner, який інкапсулює повний цикл взаємодії із завантаженою моделлю — від ініціалізації середовища до формування вихідних даних. Конструктор класу приймає шлях до моделі та виконує низку перевірок на існування файлу моделі й доступність динамічної бібліотеки onnxruntime.dll, що дозволяє запобігти аварійному завершенню

програми у разі відсутності необхідних компонентів. Після проходження перевірок створюється об’єкт середовища Ort::Env і конфігуруються параметри виконання через Ort::SessionOptions. Зокрема, кількість потоків обчислення обмежується одним, а параметр оптимізації графа встановлюється на рівень ORT_ENABLE_EXTENDED, що дозволяє ONNX Runtime автоматично перебудовувати граф виконання для підвищення продуктивності без зміни функціональної логіки моделі. На основі цих налаштувань створюється об’єкт Ort::Session, який містить у собі завантажену ONNX-модель та відповідає за виконання її інференсу. Додатково у конструкторі створюється стандартний аллокатор Ort::AllocatorWithDefaultOptions, який використовується для виділення та керування пам’яттю під час роботи моделі. Оскільки всі компоненти класу використовують розумні вказівники, у деструкторі передбачене ручне звільнення ресурсів шляхом скидання відповідних полів, що гарантує коректне завершення роботи і попереджує витоки пам’яті.

Основна функціональність класу реалізована у методі Run, який виконує обчислення вихідних даних моделі на основі двох векторів: латентного простору та умов генерації рівня. На початку функції здійснюється зчитування кількості та імен усіх вхідних і вихідних тензорів моделі — це необхідно для коректної передачі даних у сесію ONNX Runtime. Після цього метод формує два вхідні тензори у форматі (1 × N) — один містить латентний вектор, другий описує умови генерації. Таке представлення є типовим для моделей, що генерують один зразок за один виклик і відповідає структурі CVAE-декодера, використаного у проєкті.

Після формування тензорів викликається метод Session->Run, який здійснює запуск моделі та повертає набір вихідних тензорів. Основний вихідний тензор містить повний набір параметрів, що описують згенеровані об’єкти рівня: категоріальні ознаки, позиції, ротації, масштаби, індикатори існування, фізичні властивості та інші параметри. Завершальним етапом є копіювання даних цього тензора у масив Out, який передається назад у Unreal

Engine і використовується для побудови рівня на основі результатів роботи моделі.

Після створення інкапсулятора для виконання ONNX-моделі в Unreal Engine наступним кроком стало розроблення класу DecoderBlueprintLib, який забезпечує зручний інтерфейс для генерації рівнів безпосередньо з Blueprint. Цей клас об’єднує у собі завантаження моделі та всіх допоміжних даних, побудову умовного вектору, запуск декодера ONNX-моделі, обробку отриманого вихідного вектору та формування кінцевого опису рівня з подальшим розміщенням об’єктів у ігровому світі. Основні функції, що відповідають за цей процес, — LoadDecoderAndData, GenerateLevelDesc та GenerateAndSpawn.

Функція LoadDecoderAndData виконує повне завантаження ONNX-моделі та відповідних допоміжних файлів, створених на етапі preprocess.py та експорту моделі. Цей метод запускається один раз при старті роботи системи або перед першою генерацією рівня. Усередині нього викликається функція LoadFilesInternal, яка по черзі завантажує три JSON-файли: метадані моделі, скейлери та словники категорій. Завантаження здійснюється через ReadJsonFile. Далі, ParseMeta відновлює ключові параметри моделі — розмір латентного простору, кількість об’єктів, довжини сегментів вихідного вектору та структуру одного об’єкта. Потім ParseScalers завантажує усі скейлери, необхідні для денормалізації числових ознак, а ParseVocabs формує словники категоріальних значень (типи об’єктів, мобільність, назви мешів та матеріалів). Після успішної перевірки всіх залежностей створюється екземпляр FONNXRunner, який відповідає за безпосереднє виконання нейронної моделі в ONNX Runtime.

Метод GenerateLevelDesc є основним механізмом побудови нового рівня, оскільки саме він перетворює параметри, отримані з Blueprint, у структурований опис рівня. Спершу функція перевіряє, чи всі дані моделі завантажені. Потім формується умовний вектор за допомогою BuildCondition, який містить розміри рівня, позицію входу та виходу, а також їх орієнтацію й

масштаб. Отриманий вектор нормалізується за допомогою ScalerConditions, щоб він відповідав формату даних, на яких навчалася нейромережа.

Після цього генерується латентний вектор Z. Він формується як послідовність випадкових чисел, керованих початковим значенням seed, що дозволяє отримувати як повністю випадкові рівні, так і детерміновані, якщо використано однаковий seed. Далі умовний вектор та латентний простір передаються до Runner->Run, який виконує ONNX-модель і повертає вихідний числовий вектор.

Найскладніша частина — це подальша його інтерпретація у функції PostProcessToLevel. Вона послідовно аналізує кожен сегмент вихідного вектору: визначає, чи повинен бути створений об’єкт, знаходить його позицію у дискретній сітці, перетворює індекс ротації у реальний кут, денормалізує масштаб і координату Z, визначає параметри фізики, а потім відновлює категоріальні характеристики за допомогою словників. Спеціальна логіка також застосовується для процедурних стін, для яких генеруються додаткові параметри. У результаті формується структура FLevelDesc, яка містить повний опис рівня у вигляді набору об’єктів із їхніми параметрами.

Функція GenerateAndSpawn завершує цикл роботи з моделлю, поєднуючи генерацію рівня та його фактичне створення у світі Unreal Engine. Спочатку вона викликає GenerateLevelDesc та отримує структурований опис рівня. Після цього дані передаються у UCreateDataset::SpawnLevelFromDesc, що створює всі об’єкти у світі, застосовуючи відповідний масштаб, матеріали, клас актора, фізичні параметри та спеціальні властивості. Крім того, функція повертає масив створених акторів, що значно спрощує подальше керування рівнем, його очищення або перегенерацію.

Усі описані функції можуть бути викликані безпосередньо з Blueprint. Для тестування системи був модифікований utility-віджет BP_Generate_Level, у який додано кнопку Generate from ONNX (рис. 2.8) із відповідною логікою виклику (рис. 2.9). Оскільки GenerateAndSpawn містить велику кількість

параметрів, їх було передано безпосередньо всередині виклику, що достатньо для цілей тестування.

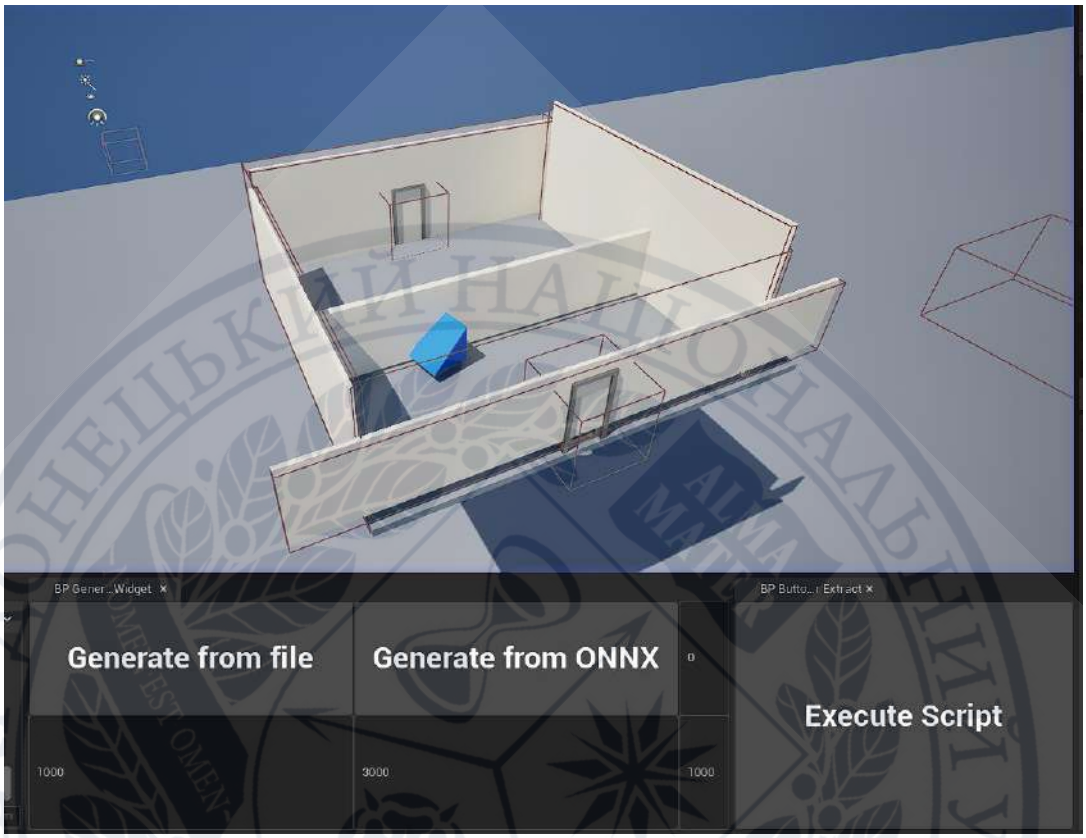

Рисунок 2.8 – результат роботи Generate from ONNX

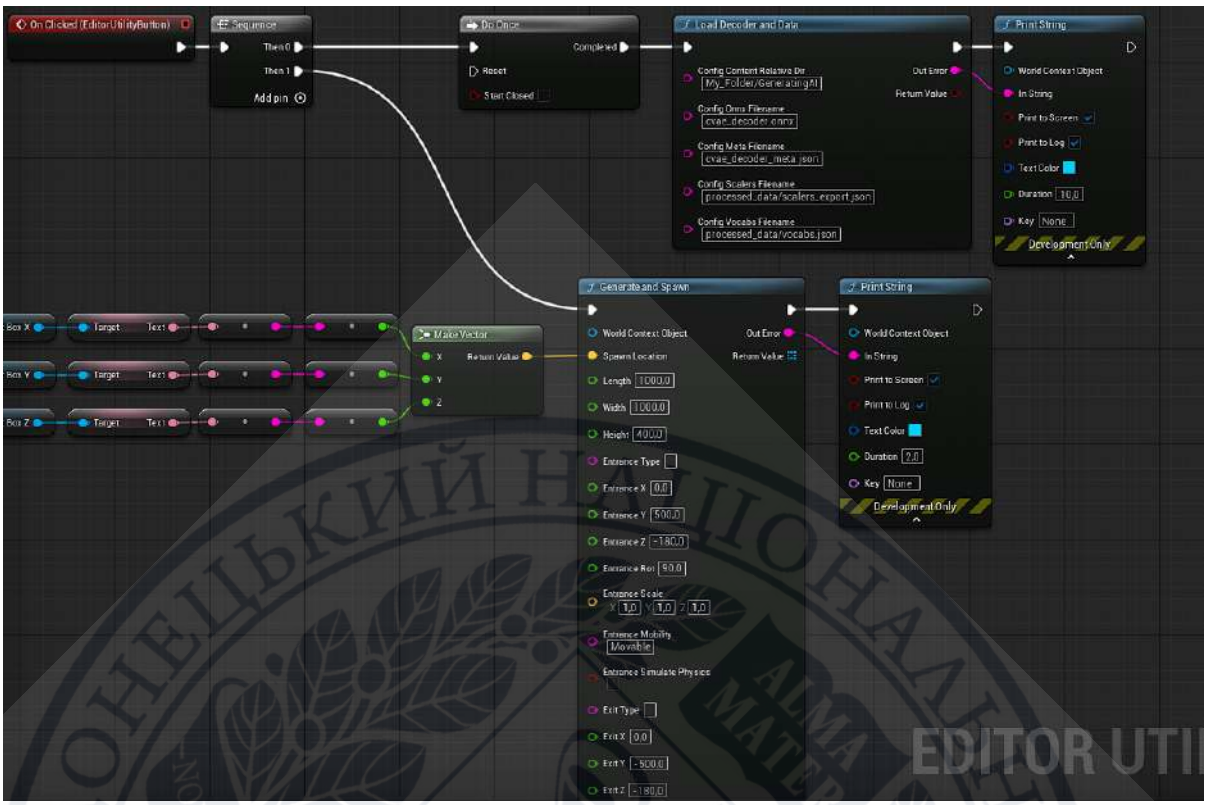

Рисунок 2.9 – результат роботи Generate from ONNX

На практиці генерація рівня відбувається миттєво, без будь-яких відчутних затримок, що відкриває можливість подальшого використання цієї системи під час реального ігрового процесу.

РОЗДІЛ 3. РОЗРОБКА ІГРОВОГО ДОДАТКУ

3.1 Рівень пісочниця

Після успішного експорту нейронної моделі до Unreal Engine постало питання практичного тестування її роботи в умовах реальної симуляції. Необхідно було оцінити, як модель реагує на різні параметри входу, наскільки стабільно відбувається генерація сцен та чи можна забезпечити коректну поведінку системи безпосередньо під час гри. Для цього було створено спеціальний тестовий рівень — Sandbox_Level, який виконує роль пісочниці. Він дозволяє як розробнику, так і майбутньому гравцю експериментувати з параметрами генерації та спостерігати результати у режимі реального часу.

Логіка роботи Sandbox_Level полягає у наданні користувачу можливості динамічно змінювати вхідні параметри генерації та негайно спостерігати результат. Кожен параметр — координати входу чи виходу, розміри рівня або інші атрибути сцен — представлений окремим інтерактивним елементом інтерфейсу. Після налаштування потрібних значень користувач може ініціювати створення нового рівня. Таким чином Sandbox_Level виконує роль панелі керування генеративною моделлю, а для реалізації цієї взаємодії було розроблено спеціальний набір універсальних віджетів: слайдерів, кнопок і текстових елементів, які далі інтегруються в окремі актори та розміщуються у просторі рівня.

Першим кроком стало створення базового віджета-слайдера WB_Slider. Він використовується як основний елемент керування, з яким безпосередньо взаємодіє гравець. У дизайнерській частині Blueprint для WB_Slider було розміщено Horizontal Box, який містить елемент Slider та текстове поле TextBox. Для Slider були додані дві ключові події: On Value Changed — викликається під час руху повзунка, та On Mouse Capture End — спрацьовує після завершення зміни значення користувачем.

У віджеті створено три змінні: MinValue, MaxValue та Value, що задають межі слайдера та його поточний стан. Для коректного оновлення UI була

реалізована функція UpdateWidget, яка встановлює мінімальні й максимальні значення повзунка, коригує актуальне значення згідно з обмеженнями та синхронізує текстове поле зі станом слайдера. Ця функція викликається як у події Event Construct, так і в обробнику події OnValueChanged, де додатково оновлюється змінна Value.

Для забезпечення можливості зв'язку з рівнем був створений Event Dispatcher OnChange, який активується під час завершення зміни значення (OnMouseCaptureEnd). Це дозволяє уникнути надмірної кількості викликів подій, що неминуче виникли б при обробці кожного руху слайдера.

Наступним елементом є віджет-кнопка WB_Button, що складається з кнопки та текстового поля, яке задає назву дії. У властивостях кнопки створена подія OnClicked, а через аналогічний механізм Event Dispatcher OnPressed забезпечується можливість відловлювати натискання на рівні. Функція UpdateWidget відповідає за встановлення текстового значення під час створення віджета.

Третім віджетом є WB_ShowText, що забезпечує відображення текстових підказок та інформації. Він містить змінну типу Text і функцію ChangeText, яка дозволяє рівню змінювати показуване значення в будь-який момент роботи застосунку.

На основі цих віджетів були створені актори, що здатні відображати UI-елементи у 3D-сцені. Першим таким актором став BP_SliderActor, який містить компонент Widget Component. У події BeginPlay за допомогою ноди Create Widget створюється екземпляр WB_Slider, йому передаються значення MinValue та MaxValue, після чого його інтерфейс відображається через Widget Component. Подія OnChange WB_Slider була прив'язана до відповідної події BP_SliderActor, щоб рівень міг реагувати на зміну значень.

Додатково актор має функції GetValue, ChangeMaxValue та ChangeMinValue, що дозволяють рівню зчитувати й змінювати стан віджета. Подібним чином були створені BP_TextActor і BP_ButtonActor. У текстового

актора передбачена змінна для початкового значення тексту, а BP_ButtonActor реалізує власний Event Dispatcher, аналогічний до кнопкового віджета.

Після створення необхідного набору віджет-акторів вони були розміщені на сцені Sandbox_Level. На рівні розташовано п’ять слайдерів для керування координатами входу, виходу та розмірними параметрами рівня, кнопку для запуску генерації та три текстові актори для пояснень інтерфейсу. Загальний вигляд інтерфейсу представлено на рисунку 3.1.

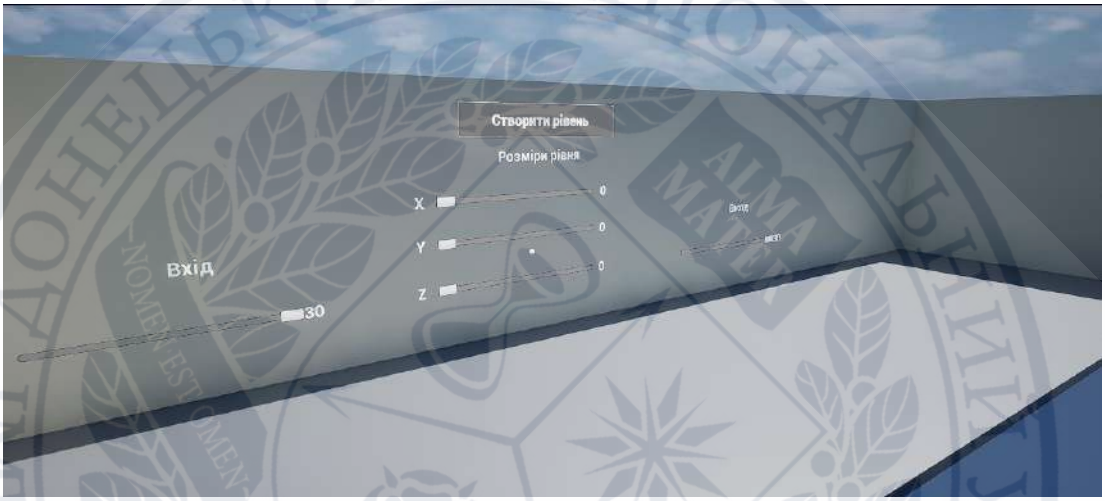

Рисунок 3.1 – Віджет актори рівня-пісочниці

Хоча віджет-актори вже розміщені на сцені й здатні взаємодіяти з логікою рівня, сам гравець за замовчуванням не має засобів керувати ними у 3D-просторі. Для цього до персонажа було додано компонент WidgetInteraction, який формує промінь взаємодії та може передавати UI-події віджетам у світі. Компонент прив’язано до камери персонажа, завдяки чому взаємодія набуває інтуїтивного характеру: користувач просто спрямовує погляд на елемент і, натискаючи або відпускаючи ліву кнопку миші, запускає функції PressPointerKey та ReleasePointerKey, що генерують коректні сигнали для слайдерів і кнопок. Це дозволяє користувачеві працювати з віджетами так само природно, як з будь-якими 3D-об’єктами.

У процесі тестування було виявлено, що рівні, згенеровані нейронною мережею, не завжди гарантують стабільне середовище, і гравець може

випадково опинитися за межами ігрового простору без можливості повернення. Щоб розв’язати цю проблему, була додана окрема система повернення у стартове положення. На відміну від керування віджетами, яке не потребує гнучкого переналаштування, команда повернення має бути змінною на розсуд користувача, тому її реалізовано через Enhanced Input. Створено дію IA_ResetLevel, яка активується під час натискання клавіші та має тип boolean. Цю дію додано у IMC_Default, де за кнопку скидання призначено клавішу R. Далі її прив’язано у Blueprint персонажа до відповідної події, яка через Event Dispatcher надсилає рівню сигнал перемістити гравця у стартову точку [29].

Для точнішого прицілювання променя взаємодії був доданий простий екранний елемент - статичний приціл у вигляді точки по центру. Його реалізація складається з трьох компонентів: віджету WB_Sight, який містить лише зображення прицілу; класу Hud_Sight, що створює цей віджет під час події BeginPlay та відображає його через Add to Viewport; та ігрового режиму GM_Sandbox, який визначає використання даного HUD та відповідного персонажа для поточного рівня. Така структура є зрозумілою: кожен клас виконує одну вузьку роль, а GameMode забезпечує їхню інтеграцію у загальну логіку гри. Після прив’язки режиму до рівня приціл автоматично з’являється під час запуску, полегшуючи взаємодію з інтерфейсом.

Після підготовки інтерфейсу можна переходити до реалізації основної логіки рівня. У події BeginPlay на початку викликається функція LoadDecoderAndData, яка завантажує нейромережеву модель разом із потрібними скейлерами, словниками та іншими метаданими. Далі налаштовуються прив’язки подій: сигнал скидання з персонажа поєднується з внутрішньою подією Reset, що відповідає за повернення у початкову позицію; зміни значень слайдерів також пов’язуються з відповідними функціями. Наприклад, зміна довжини рівня по осі X викликає ChangeMaxValueSliders, яка коригує допустимі межі розташування дверей, щоб уникнути ситуацій, у яких вони могли б опинитися за межами кімнати. Для цього використовується параметр DoorPadding, що задає мінімальний відступ від стін.

Ключовим елементом роботи Sandbox_Level є обробник кнопки генерації рівня, який складається з двох послідовних кроків: спершу видаляються актори попереднього рівня, а потім створюється новий. Значення всіх слайдерів зчитуються через функції GetValue, а всі параметри, які гравцеві вводити немає потреби (наприклад, координати Y і Z дверей, їх орієнтація чи масштаб), обчислюються автоматично відповідно до загальної конфігурації. Додатково на рівні розміщено актор TargetPoint, який використовується для визначення місця генерації сцени: змінюючи його положення у редакторі, можна коригувати позицію згенерованого рівня без модифікації коду. Після створення рівня функція повертає масив акторів, що були згенеровані; вони зберігаються у змінну та видаляються перед наступною генерацією за допомогою ноди Destroy Actor, щоб уникнути накопичення об’єктів.

У підсумку гравець отримує зручний 3D-інструментарій для зміни параметрів генерації та перегляду отриманих результатів у реальному часі, що дозволяє повною мірою оцінити роботу нейромережевої моделі в інтерактивному середовищі (рис. 3.2).

Рисунок 3.2 – результат роботи Sandbox_Level

У процесі тестування було встановлено, що сама генерація рівнів відбувається практично миттєво й не спричиняє відчутних затримок навіть під час багаторазових послідовних викликів. Модель стабільно відтворює типову

структуру рівня, зберігаючи коректне розташування основних елементів навіть тоді, коли вхідні параметри суттєво відхиляються від даних, використаних під час навчання (рис. 3.3). Водночас окремі конфігурації все ж можуть призводити до аномалій - трапляються випадки накладання двох стін в одному куті, завищене положення підлоги, що перекриває дверний отвір, або інші дрібні геометричні помилки. Такі результати вказують як на загальну життєздатність підходу, так і на потребу подальшого вдосконалення моделі та розширення навчального набору даних.

Рисунок 3.3 – Генерація рівня із нетиповими параметрами

3.2 Рівень вгадування згенерованих рівнів

Після створення рівня-пісочниці та успішного тестування нейромережевої генерації постає наступний етап - реалізація ігрового режиму, у якому гравець має визначити, який із запропонованих рівнів був створений моделлю. Для цього створено окремий рівень Guess_the_level. Керування ігровим процесом здійснюється за допомогою натискних кнопок-акторів: одна кнопка запускає раунд, інші - дозволяють вибрати варіант, який, на думку гравця, був згенерований нейромережею. Логіка створення й очищення рівнів реалізована у Blueprint самого рівня, а додатковий інтерфейс гравця відображає кількість послідовно відгаданих спроб.

Основу системи взаємодії становить актор BP_Button - натискна плита, що реагує на присутність гравця. Він складається з двох статичних мешів (каркаса та рухомої частини), колізійного компонента, який визначає факт наступання на кнопку, та widget component із текстовим віджетом ShowText, створеним у попередньому розділі. Актор містить три основні змінні: ButtonState (показує, натиснута кнопка чи ні), MaterialNumber (визначає її колір) та Text (підпис, який відображає віджет).

Головне завдання кнопки - передавати сигнал про натискання або відпускання. Для цього вона має контролювати, які актори знаходяться у зоні її колізії, та реагувати лише на персонажа гравця. Логіка реалізована у функції CheckActors: за допомогою ноди GetOverlappingActors отримується перелік усіх акторів, що перетинають кнопку, після чого відбувається перевірка на наявність гравця. Якщо він присутній, змінюється ButtonState та викликається диспетчер подій OnPressed. Функція прив’язана до подій OnComponentBeginOverlap та OnComponentEndOverlap, що дозволяє точно фіксувати момент активування й деактивування кнопки. Для відображення натискання рухома частина кнопки анімується за допомогою Timeline і зміни локальної позиції компонента.

Додатково кнопці надано можливість змінювати матеріал, щоб урізноманітнити її зовнішній вигляд. Це виконується у Construction Script за допомогою ноди SetMaterial - обраний колір відображається одразу в редакторі. Для коректного читання тексту на віджеті додано функцію TextRotation, яка щокадрово обертатиме панель з текстом у напрямку камери гравця. Необхідне обертання обчислюється за допомогою GetPlayerCameraManager та FindLookAtRotation, що гарантує видимість тексту під будь-яким кутом. Оновлення тексту віджета виконано у події BeginPlay.

Далі виконується розміщення на сцені трьох кнопок, кожній з них задаються власний текст та варіант матеріалу, щоб вони виконували різні функції та візуально відрізнялися одна від одної (рис. 3.4), що забезпечує зрозумілість інтерфейсу та полегшує взаємодію гравця з елементами управління.

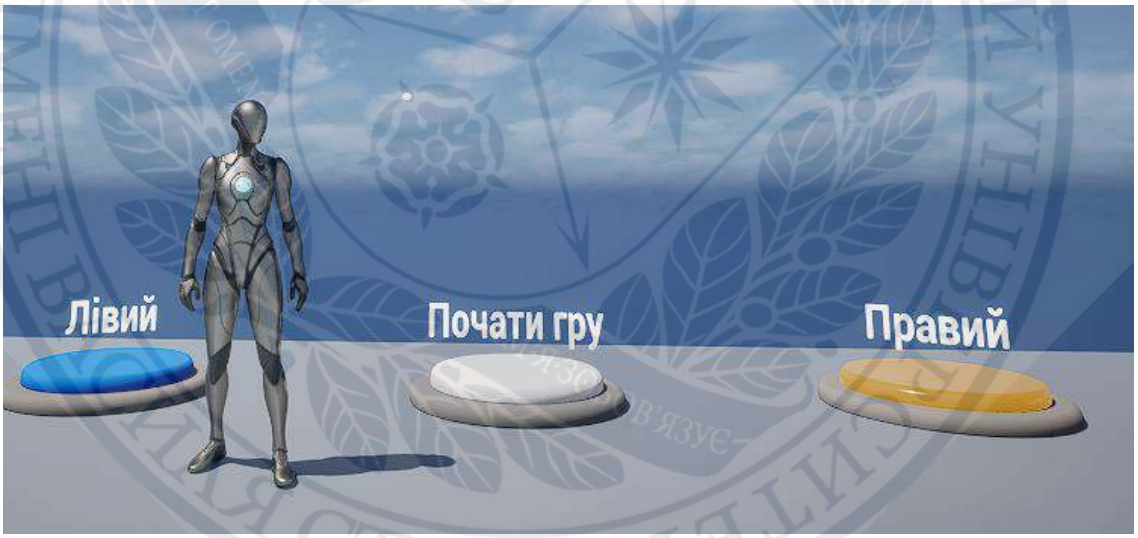

Рисунок 3.4 – Кнопки BP_Button

Наступним кроком є створення системи нарахування балів за правильні відповіді. Для цього розроблено віджет WB_Score, який містить текстове поле у лівому верхньому куті екрану. Структура віджета включає змінну Score та дві функції - ChangeScore та UpdateText. Перша відповідає за зміну числового значення кількості балів, а друга - за оновлення тексту у форматі “Score: N”.

Функція UpdateText викликається як під час створення віджета (Construct), так і після зміни значення Score.

Для того щоб віджет Score відображався у грі, створено спеціальні класи Hud_GuessLevel та GM_GuessLevel. Їхня робота аналогічна системі прицілу, однак доповнена логікою оновлення значення балів. У HUD додано функцію UpdateWidgets, яка змінює текст WB_Score через виклик ChangeScore. У GameMode реалізована функція ChangeGameScore, яка отримує нове значення лічильника (передане рівнем), зберігає його у HUD та ініціює оновлення інтерфейсу. Таким чином гравець отримує зручну систему індикації послідовних правильних відповідей.

Основна логіка мінігри реалізується у Blueprint самого рівня. Під час події BeginPlay виконуються знайомі з попереднього розділу кроки: завантаження декодера та допоміжних даних, прив’язування дії повернення гравця на стартову позицію, а також отримання посилання на GameMode для подальшої роботи із системою балів. Після цього конфігуруються кнопки. Центральна кнопка слугує для початку гри: її подія OnPressed прив’язується до користувацької події StartGame, яка виконується лише один раз. Вона запускає функцію CreateLevel, активує логіку для двох інших кнопок (вибір лівого або правого рівня) та видаляє стартову кнопку зі сцени.

Функція CreateLevel керує повним циклом оновлення мінігри. Спочатку вона викликає ClearArea, яка видаляє рівні, створені на попередньому кроці. Після цього запускається SpawnLevels, що генерує нову пару рівнів - один процедурний і один отриманий через нейромережу. Генерація параметрів виконується випадковим чином у межах діапазонів LowRange та HighRange. Паралельно вибирається, яка зі сцен буде нейромережевою, а яка - стандартною. Це значення зберігається у змінну WhichAILevel.

Далі обчислюються координати спавну обох рівнів відносно актора TargetPoint, який виконує роль маркера позиції. Через функцію SetLevelsPosition визначаються дві точки розташування - з невеликим проміжком між ними, щоб уникати перетинів у разі некоректної генерації

мережі. Після цього створюється процедурний рівень (BP_ProceduralLevel) та викликається GenerateAndSpawn для рівня, побудованого нейромережею. Посилання на всі створені актори зберігаються у змінних рівня для подальшого очищення.

Кнопки-гадки реалізовані через події LeftLevelGuess та RightLevelGuess. Обидві викликають функцію ChangeScore, яка порівнює вибір гравця із значенням WhichAILevel. Якщо відповідь правильна - викликається ChangeGameScore(CurrentScore + 1) у GameMode; якщо ні - значення скидається на нуль. Після цього, незалежно від результату, запускається черговий виклик CreateLevel, що формує нову пару сцен.

Таким чином рівень Guess_the_level реалізує повний цикл мінігри: від запуску, генерації рівнів і обробки вибору гравця до підрахунку серії вгаданих спроб (рис 3.5). Це дозволяє оцінити властивості нейромережевої генерації у ігровій формі та продемонструвати її можливості у безпосередній інтеракції з користувачем.

Рисунок 3.5 – Guess_the_level

3.3 Головне меню та збірка проекту

Після розроблення основних рівнів виникає практична проблема: у фінальній збірці гравець не має доступу до редактора Unreal Engine, а отже не

може самостійно обирати потрібну сцену чи переходити між режимами гри. Для усунення цього обмеження було створено головне меню, яке викликається натисканням клавіші Escape та дозволяє переключатися між рівнями або завершувати гру.

Основою інтерфейсу меню став віджет BP_Menu, що представляє собою панель із вертикальним списком кнопок. До нього включено три елементи керування: кнопку переходу до рівня вгадування, кнопку запуску рівня-пісочниці та кнопку виходу з програми (рис. 3.6). Для кожної з кнопок реалізовані події OnClicked. Кнопка завершення гри викликає ноду QuitGame, тоді як дві інші запускають відповідні рівні через OpenLevel, одночасно вимикаючи показ курсора та переводячи систему введення у режим Game Only для повноцінної взаємодії з 3D-світом.

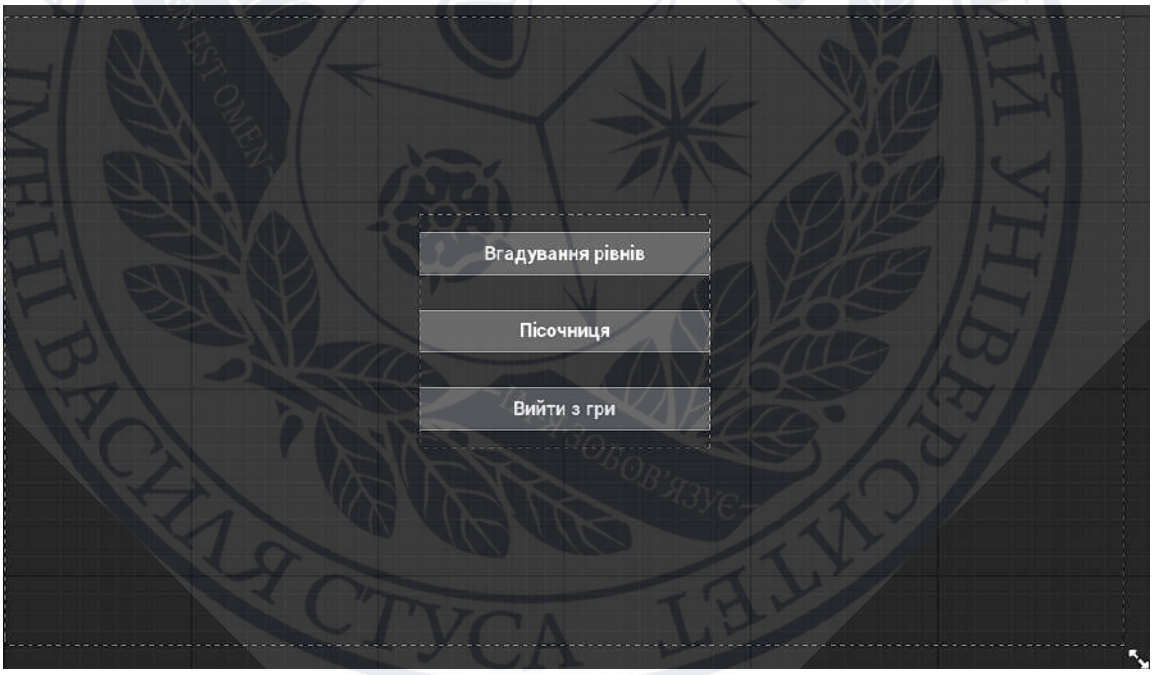

Рисунок 3.6 – BP_Menu

Щоб меню могло викликатися у будь-який момент гри, було розширено логіку раніше створених HUD-класів. У події BeginPlay кожен HUD створює екземпляр BP_Menu і зберігає на нього посилання, але не відображає його негайно. Для забезпечення коректного перемикання між ігровим керуванням

та взаємодією з UI створений інтерфейс I_ShowMenu, підключений до HUD. Він містить одну функцію, яка у Blueprint-графі реалізує показ або приховування меню, керує видимістю курсора та перемикає режим введення між UI Only та Game Only.

Для ініціювання цієї функції створено спеціальний клас PC_Main, що наслідує PlayerController. На подію натискання клавіші Escape PlayerController звертається до HUD, указаний у GameMode, та викликає метод інтерфейсу I_ShowMenu. Завдяки цьому логіка меню централізована й працює незалежно від активного рівня. Після створення даний контролер підключається до відповідних Gamemode-класів, що забезпечує можливість відкривати меню під час гри.

Щоб після запуску програми гравець не опинявся одразу на одному з ігрових рівнів, було створено окрему стартову сцену Main_Menu. Це порожній рівень, який містить лише Blueprint-логіку для відображення BP_Menu. Для нього розроблено окремий Gamemode - GM_MainMenu, що не використовує HUD чи персонажа, оскільки ці елементи не потрібні в межах головного екрану. Щоб зробити Main_Menu стартовою сценою, у Project Settings у розділі Maps & Modes значення Game Default Map встановлюється на цей рівень.

Після завершення всіх налаштувань проєкт стає готовим до пакування. У меню Unreal Engine вибирається пункт Platforms - Windows, а далі виконується Package Project у конфігурації Shipping, призначеній для фінальної збірки. Через деякий час у вибраній теці з’являється готова папка з виконуваним файлом гри.

У зібраному застосунку всі системи працюють стабільно: перемикання між рівнями відбувається швидко, інтерфейс реагує без затримок, а механізми генерації рівнів не спричиняють відчутних просідань продуктивності. Єдине, що вирізняється - дещо збільшений час першого запуску рівня через завантаження моделі та допоміжних даних, однак подальша робота відбувається миттєво та без лагів (рис. 3.7).

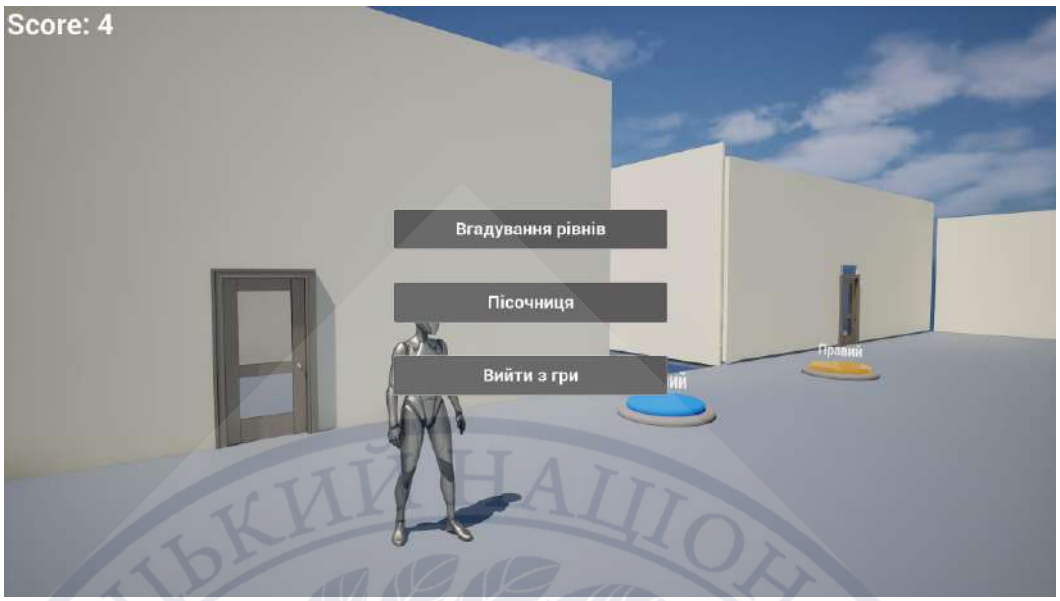

Рисунок 3.7 – Результат роботи фінальної збірки

ВИСНОВКИ

У магістерській роботі реалізовано повноцінний цикл створення системи генерації ігрових рівнів із використанням технологій штучного інтелекту та її інтеграції в рушій Unreal Engine 5. Проведене дослідження підтвердило ефективність застосування умовного варіаційного автоенкодера (CVAE) для формування структурованих тривимірних сцен, а також продемонструвало можливість використання моделей глибокого навчання безпосередньо під час ігрового процесу.

Було розроблено систему процедурної генерації навчального датасету, що включає автоматичне створення 2048 варіантів рівнів у середовищі Unreal Engine. На основі цього датасету створено та навчено модель CVAE, яка здатна відтворювати типову структуру рівня та генерувати нові конфігурації з урахуванням заданих параметрів у реальному часі. Незважаючи на окремі геометричні неточності (накладання стін, неправильне положення підлоги тощо), модель показала стабільність та узагальнювальну здатність навіть у випадку параметрів, що суттєво відрізняються від навчальних.

Інтеграція моделі в Unreal Engine була виконана за допомогою ONNX Runtime, що забезпечило незалежність від зовнішніх серверів і відсутність затримок під час генерації. Практичне тестування в рамках рівня Sandbox_Level підтвердило, що створення сцен відбувається миттєво, а перше завантаження пов’язане лише з ініціалізацією моделі. Крім того, реалізовано окрему мінігру Guess_the_level, яка дозволяє оцінити результати генерації у ігровому режимі та взаємодіяти з моделлю у форматі змагального процесу.

На завершення створено головне меню, налагоджено навігацію між рівнями та виконано пакування проєкту у вигляді фінального застосунку. Робота демонструє, що інтеграція нейронних мереж у ігрові рушії є перспективним напрямом, здатним суттєво спростити процес створення ігрового контенту та підвищити варіативність геймплею. Результати дослідження можуть бути використані як основа для подальшого розширення

функціональності моделі, збільшення різноманітності навчального набору та впровадження складніших архітектур генеративних моделей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Summerville A., Snodgrass S., Guzdial M., Riedl M. O. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games*. 2018. Vol. 10, No 3. P. 257–270.
2. Kingma D. P., Welling M. Auto-Encoding Variational Bayes. *arXiv preprint*. 2015. arXiv:1312.6114. URL: https://arxiv.org/abs/1312.6114 (дата звернення: 21.10.2025).
3. Goodfellow I., Pouget-Abadie J., Mirza M., Xu B. Generative Adversarial Nets. *Advances in Neural Information Processing Systems (NeurIPS)*. 2015. P. 2672–2680.
4. Sarkar A., Cooper S. Controllable Dungeon Generation using Conditional Variational Autoencoders. *Proceedings of the 16th International Conference on the Foundations of Digital Games (FDG)*. 2021. P. 1–10.
5. Epic Games, Inc. Procedural Mesh Component in Unreal Engine. *Unreal Engine 5 Documentation*. 2024. URL: https://docs.unrealengine.com/5.3/en-US/procedural-mesh-component-in-unreal-engine/ (дата звернення: 21.10.2025).
6. Epic Games, Inc. Python Scripting in Unreal Engine. *Unreal Engine 5 Documentation*. 2024. URL: https://docs.unrealengine.com/5.3/en-US/python-scripting-in-unreal-engine/ (дата звернення: 21.10.2025).
7. Paszke A., Gross S., Massa F., Lerer A. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*. 2019. P. 8024–8035.
8. Microsoft. TORCH-DIRECTML: PyTorch with DirectML on Windows. *PyPI*. 2022. URL: https://pypi.org/project/torch-directml/ (дата звернення: 21.10.2025).
9. ECMA-404. The JSON Data Interchange Standard. 2nd ed. 2017. 16 p. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/ (дата звернення: 21.10.2025).

10. Vaswani A., Shazeer N., Parmar N., Uszkoreit J. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*. 2017. P. 5998–6008.
11. Millington I., Funge J. *Artificial Intelligence for Games*. 3rd ed. Boca Raton: CRC Press, 2019. 856 p.
12. Perlin K. An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*. Vol. 19, No 3. P. 287–296.
13. Sohn K., Lee H., Yan X. Learning Structured Output Representation using Deep Conditional Generative Models. *Advances in Neural Information Processing Systems (NeurIPS)*. 2015. P. 3483–3491.
14. Guzdial M., Riedl M. O. Learning to Blend Computer-Generated and Human-Designed Game Levels. *Proceedings of the 2018 AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 2018. P. 12–18.
15. Epic Games, Inc. NNE (Neural Network Engine) Overview. *Unreal Engine 5 Documentation*. 2024. URL: https://docs.unrealengine.com/5.3/en-US/nne-overview/ (дата звернення: 21.10.2025).
16. Alvarez J., Togelius J., Ashlock D. Investigating the Relationship Between Procedural Content Generation and Game Metrics for 2D Platformers. *Computational Intelligence and Games (CIG), 2018 IEEE Conference on*. 2018. P. 1–8.
17. Liu J., et al. Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration. *arXiv preprint arXiv:2410.15644*. 2024. URL: https://arxiv.org/abs/2410.15644 (дата звернення: 21.11.2025).
18. Ratican J., Hutson J. Generative AI in Game Development: A Qualitative Research Synthesis. *arXiv preprint arXiv:2509.11898*. 2025. URL: https://arxiv.org/abs/2509.11898 (дата звернення: 21.11.2025).
19. Sarkar A., Cooper S. Conditional Level Generation and Game Blending. *IEEE Conference on Games (CoG)*. 2020. P. 1–8.

20. Aylagas M. V., Bergdahl J. Improving Conditional Level Generation Using Automated Validation in Match-3 Games. *IEEE Transactions on Games*. 2024. Vol. 16. URL: https://ieeexplore.ieee.org/document/10666000 (дата звернення: 21.11.2025).
21. Torrado R., et al. Conditional Image Generation by Conditioning Variational Auto-Encoders. *OpenReview*. 2022. URL: https://openreview.net/forum?id=7MV6uLzOChW (дата звернення: 21.11.2025).
22. Procedural Content Generation with Unreal Engine 5. *GitHub*. 2024. URL: https://github.com/PacktPublishing/Procedural-Content-Generation-with-Unreal-Engine-5 (дата звернення: 21.11.2025).
23. ONNX Runtime: Cross-platform, high performance ML inferencing and training accelerator. *Microsoft*. 2024. URL: https://onnxruntime.ai/docs/ (дата звернення: 21.11.2025).
24. Nguyen T. ONNX Runtime in Unreal Engine 5. *Tu Nguyen Blog*. 2023. URL: https://boxibi24.github.io/posts/unreal/machinelearning/onxx-runtime-in-unreal-engine-5/ (дата звернення: 21.11.2025).
25. Shaker N., Togelius J., Nelson M. J. *Procedural Content Generation in Games*. Springer, 2016. 226 p.
26. Yannakakis G. N., Togelius J. *Artificial Intelligence and Games*. Springer, 2018. 329 p.
27. Doersch C. Tutorial on Variational Autoencoders. *arXiv preprint arXiv:1606.05908*. 2021. URL: https://arxiv.org/abs/1606.05908 (дата звернення: 21.11.2025).
28. Sheers R. *Unreal Engine 5 Character Creation, Animation, and Cinematics*. Packt Publishing, 2023. 472 p.
29. Romero M., Sewell B. *Blueprints Visual Scripting for Unreal Engine 5: Unleash the true power of Blueprints to create impressive games*. 3rd ed. Packt Publishing, 2022. 570 p.

30. JSON Data Support in Unreal Engine. *Epic Games Developer Network*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/json-in-unreal-engine (дата звернення: 20.10.2025).
31. Volz V., Schrum J., Liu J., Lucas S. M., Smith A. M., Risi S. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. *GECCO*. 2018. P. 221–228.
32. Karth G. et al. Neural Level Generation for Platformers via Tile-based Encoding. *IEEE Conference on Games (CoG)*. 2020. P. 1–8.
33. Sarkar A. et al. Controllable Procedural Content Generation via Conditional VAEs. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 2021.
34. Hendrikx M., Meijer S., Van Der Velden J., Iosup A. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*. 2015. Vol. 9, No 1. P. 1–22.
35. Togelius J., Yannakakis G. N., Stanley K. O., Browne C. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*. 2015. Vol. 3, No 3. P. 172–186.
36. Liu J., Snodgrass S., Khalifa A., Risi S., Yannakakis G. N., Togelius J. Deep learning for procedural content generation. *Neural Computing and Applications*. 2021. Vol. 33. P. 19–48.
37. Hinton G. E., Salakhutdinov R. R. Reducing the dimensionality of data with neural networks. *Science*. 2016. Vol. 313, No 5786. P. 504–507.
38. Summerville A., Mateas M. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*. 2016. URL: https://arxiv.org/abs/1603.00930 (дата звернення: 21.11.2025).
39. Sudhakaran S., González-Duque M., Freiberger M., Glanois C., Najarro E., Risi S. MarioGPT: Open-ended text2level generation through large

language models. *Advances in Neural Information Processing Systems*. 2024. Vol. 36. P. 54213–54227.

40. Sanders A., Walliss J. *The Unreal Engine Game Development: From Concept to Publishing*. CRC Press, 2022. 384 p.
41. Ketkar N., Moolayil J. *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*. 2nd ed. Apress, 2021. 472 p.
42. Sherif W. M. *Unreal Engine 5 Game Programming Blueprints: Unlock the power of Blueprints to build impressive games and applications*. Packt Publishing, 2024. 420 p.
43. Ebeida M. S., Mitchell S. A., Davidson A. A., Patney A., Knupp P. M., Owens J. D. Efficient and Good Delaunay Meshes from Random Points. *Computer-Aided Design*. 2015. Vol. 43, No 11. P. 1506–1515.
44. Khalifa A., Bontrager P., Earle S., Togelius J. PCGRL: Procedural Content Generation via Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2020. Vol. 16, No 1. P. 95–101.
45. Liapis A., Yannakakis G. N., Togelius J. Constrained novelty search: A study on game content generation. *Evolutionary Computation*. 2015. Vol. 23, No 1. P. 101–129.
46. Snodgrass S., Ontañón S. Generating maps using Markov chains. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2015. Vol. 10, No 1. P. 25–31.
47. Guzdial M., Liao N., Chen J., Chen S. Y., Shah S., Shah V., Reno J., Smith G., Riedl M. O. Friend, collaborator, student, manager: How design of an AI-driven game level editor affects creators. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019. P. 1–13.
48. Green M. C., Khalifa A., Alsoughayer A., Surana D., Liapis A., Togelius J. Two-step Procedural Content Generation using Generative Adversarial Networks. *Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games (CIG)*. 2018. P. 1–8.

49. Hastings E. J., Guha R. K., Stanley K. O. Automatic content generation in the Galactic Arms Race video game. *IEEE Transactions on Computational Intelligence and AI in Games*. 2019. Vol. 1, No 4. P. 245–263.
50. Dormans J. Adventures in level design: Generating missions and spaces for action adventure games. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 2015. P. 1–8.