

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут прикладного системного аналізу

Кафедра штучного інтелекту

До захисту допущено:

В. о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«__» _____ 20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи і методи штучного інтелекту»

спеціальності 122 «Комп'ютерні науки»

на тему: «Методи генеративного моделювання для створення ігрового контенту»

Виконав:

студент IV курсу, групи КІ-22
Савченко Данііл Геннадійович

Керівник:

старший викладач кафедри штучного інтелекту,
PhD, Коломоєць Сергій Олексійович

Консультант з нормоконтролю

фахівець першої категорії кафедри штучного інтелекту,
к.т.н., доцент, Комариста Богдана Миколаївна

Рецензент:

асистент кафедри математичних методів системного аналізу,
PhD, Канцедал Георгій Олегович

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Київ – 2026 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Системи і методи штучного інтелекту»

ЗАТВЕРДЖУЮ

В. о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«15» січня 2026 р.

ЗАВДАННЯ

на дипломну роботу студенту
Савченко Даніілу Геннадійовичу

1. Тема роботи «Методи генеративного моделювання для створення ігрового контенту», керівник роботи Коломоєць Сергій Олексійович, доктор філософії, асистент кафедри штучного інтелекту, затверджені наказом по НН ІПСА від «27» травня 2026 р. № 1944-с.
2. Термін подання студентом роботи «05» червня 2026 року.
3. Вихідні дані до роботи: набори даних ігрових текстур та tile-базованих карт рівнів для задачі процедурної генерації контенту.
4. Зміст роботи. Необхідно виконати завдання, представлені нижче. 1. Огляд методів генеративного моделювання та їх застосування у геймдеві. 2. Аналіз вимог та проектування системи генерації ігрового контенту. 3. Реалізація системи генерації ігрового контенту в Unity. 4. Функціонально-вартісний аналіз програмного продукту.
5. Перелік ілюстративного матеріалу: електронна презентація.
6. Дата видачі завдання «02» лютого 2026 року.

Календарний план

<i>№ з/п</i>	<i>Назва етапів виконання дипломної роботи</i>	<i>Термін виконання етапів роботи</i>	<i>Примі- тка</i>
1	Вивчення літератури за темою роботи	20.04.2026	Виконано
2	Підготовка першого розділу	27.04.2026	Виконано
3	Підготовка другого розділу	04.05.2026	Виконано
4	Розробка програмного продукту	11.05.2026	Виконано
5	Підготовка третього розділу	18.05.2026	Виконано
6	Підготовка четвертого розділу	25.05.2026	Виконано
7	Оформлення дипломної роботи	01.06.2026	Виконано
8	Підготовка презентації	05.06.2026	Виконано

Студент

Даніїл САВЧЕНКО

Керівник

Сергій КОЛОМОЄЦЬ

РЕФЕРАТ

Дипломна робота: 93 с., 15 рис., 14 табл., 21 посилань, 3 додатки.

ГЕНЕРАТИВНО-ЗМАГАЛЬНІ МЕРЕЖІ (GAN), ВАРІАЦІЙНИЙ АВТОЕНКОДЕР (VAE), ПРОЦЕДУРНА ГЕНЕРАЦІЯ КОНТЕНТУ (PCG), UNITY, ІГРОВИЙ КОНТЕНТ, ТЕКСТУРИ, РІВНІ, ГЛИБОКЕ НАВЧАННЯ.

Дипломна робота присвячена дослідженню методів генеративного моделювання – генеративно-змагальних мереж (GAN), варіаційних автоенкодерів (VAE) та дифузійних моделей – для автоматизованого створення ігрового контенту.

Мета роботи: підвищення ефективності методів генеративного моделювання для створення різних типів ігрового контенту з оцінкою якості та ефективності запропонованих підходів.

Використані методи: системний аналіз, методи глибокого навчання, методи оцінювання якості зображень (FID, PSNR, SSIM), алгоритм пошуку в ширину (BFS) для валідації графів, методи функціонально-вартісного аналізу.

Отримані результати: розроблено та програмно реалізовано систему генерації ігрового контенту в Unity. Навчено модель cGAN для генерації карт рівнів та модель VAE для текстур. Моделі успішно конвертовано у формат ONNX та інтегровано у пайплайн Unity через пакет Unity Sentis.

Рекомендації щодо використання: розроблену систему доцільно використовувати інді-студіям та розробникам ігор для автономної швидкої генерації ігрових локацій і текстур безпосередньо в рушії Unity без залежності від хмарних сервісів.

ABSTRACT

Bachelor's thesis: 93 p., 15 figures, 14 tables, 21 references, 3 appendix.

GENERATIVE ADVERSARIAL NETWORKS (GAN), VARIATIONAL AUTOENCODER (VAE), PROCEDURAL CONTENT GENERATION (PCG), UNITY, GAME CONTENT, TEXTURES, LEVELS, DEEP LEARNING.

The thesis is devoted to the study of generative modeling methods – generative adversarial networks (GAN), variational autoencoders (VAE), and diffusion models – for the automated creation of game content.

Purpose of the work: to investigate, compare, and practically implement generative modeling methods for creating various types of game content, and to evaluate the quality and efficiency of the proposed approaches.

Methods used: systems analysis, deep learning methods, image quality assessment methods (FID, PSNR, SSIM), Breadth-First Search (BFS) algorithm for graph validation, functional-cost analysis methods.

Obtained results: a game content generation system in Unity was developed and programmatically implemented. A cGAN model for level map generation and a VAE model for textures were trained. The models were successfully converted to the ONNX format and integrated into the Unity pipeline via the Unity Sentis package.

Recommendations for use: the developed system is recommended for indie studios and game developers for autonomous and fast generation of game locations and textures directly in the Unity engine without relying on cloud services.

ЗМІСТ

ВСТУП	10
РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ГЕНЕРАТИВНОГО МОДЕЛЮВАННЯ ТА ЇХ ЗАСТОСУВАННЯ У ГЕЙМДЕВІ	13
1.1 Процедурна генерація контенту (PCG): огляд та класифікація	13
1.2 Генеративно-змагальні мережі (GAN): архітектура та різновиди	15
1.2.1 Базова модель і функція втрат	16
1.2.2 Умовні GAN (cGAN, Pix2Pix, CycleGAN).....	17
1.2.3 Прогресивні та стилізовані GAN (StyleGAN, ProGAN).....	17
1.2.4 Wasserstein GAN (WGAN, WGAN-GP) – стабілізація навчання.....	18
1.2.5 Застосування в ігровій індустрії.....	18
1.3 Варіаційні автоенкодери (VAE): теоретичні основи	19
1.3.1 Архітектура VAE та варіаційна нижня межа (ELBO).....	20
1.3.2 Reparameterization trick	20
1.3.3 Застосування VAE для генерації ігрових рівнів	21
1.3.4 Гібридна модель VAE-GAN.....	22
1.4 Дифузійні моделі: принципи та застосування у геймдеві	22
1.4.1 Прямий та зворотний дифузійний процес	23
1.4.2 DDPM, DDIM – прискорена генерація.....	23
1.4.3 Latent Diffusion Models (Stable Diffusion)	24
1.4.4 Застосування для генерації текстур і концептуального арту	24
1.4.5 Контрольована генерація (ControlNet, text-conditioned generation)	25
1.5 Порівняльний аналіз генеративних підходів для PCG.....	26
1.6 Огляд існуючих рішень для генерації ігрового контенту з ML	28
Висновки до розділу 1	29
РОЗДІЛ 2 АНАЛІЗ ВИМОГ ТА ПРОЕКТУВАННЯ СИСТЕМИ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕНТУ	31

2.1 Постановка задачі та формалізація вимог	31
2.1.1 Функціональні вимоги до системи	31
2.1.2 Нефункціональні вимоги	32
2.1.3 Обмеження та умови роботи системи	32
2.2 Аналіз та вибір методу генеративного моделювання для практичної реалізації.....	33
2.2.1 Обґрунтування вибору cGAN для генерації карт рівнів	33
2.2.2 Обґрунтування вибору VAE для генерації текстур	34
2.2.3 Протокол інтеграції моделей з Unity через ONNX Runtime	34
2.3 Архітектура системи генерації контенту	35
2.3.1 Модуль навчання моделей (Python / PyTorch)	36
2.3.2 Модуль конвертації та оптимізації (ONNX-export)	36
2.3.3 Модуль інференсу в Unity (C# + Unity Sentis)	37
2.3.4 Модуль управління генерацією	37
2.4 Набори даних для навчання	38
2.4.1 Набори даних для генерації ігрових текстур.....	38
2.4.2 Набори даних для генерації карт рівнів.....	39
2.4.3 Формування навчальних вибірок та попередня обробка	40
2.5 Метрики оцінювання якості генерованого ігрового контенту	41
Висновки до розділу 2	43
РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕНТУ В UNITY.....	45
3.1 Навчання моделі cGAN для генерації tile-базованих карт рівнів	45
3.1.1 Підготовка датасету карт рівнів	45
3.1.2 Архітектура генератора та дискримінатора cGAN.....	46
3.1.3 Функція втрат і процес навчання.....	47
3.1.4 Аналіз процесу навчання та запобігання mode collapse.....	47

	8
3.1.5 Оцінка якості карт (FID, Tile Validity, Path Existence)	48
3.2 Навчання VAE для генерації ігрових текстур	48
3.2.1 Попередня обробка датасету текстур	48
3.2.2 Архітектура енкодера та декодера VAE	49
3.2.3 Функція втрат ELBO (Reconstruction Loss + KL Divergence)	50
3.2.4 Навчання та оцінка якості	50
3.2.5 Демонстрація латентної інтерполяції між текстурами	52
3.3 Конвертація та оптимізація навчених моделей для Unity	52
3.3.1 Експорт моделі PyTorch → ONNX	53
3.3.2 Перевірка коректності ONNX-моделі	53
3.3.3 Налаштування Unity Sentis	53
3.3.4 Порівняння часу інференсу на CPU та GPU	54
3.4 Реалізація системи генерації рівнів у Unity	55
3.4.1 Архітектура C#-компонентів системи	56
3.4.2 Клас LevelGeneratorManager	56
3.4.3 Клас LatentSpaceController	56
3.4.4 Клас TilemapApplicator	57
3.4.5 Валідація прохідності карти алгоритмом BFS	58
3.5 Реалізація системи генерації текстур у Unity	59
3.5.1 Клас TextureGeneratorComponent	59
3.5.2 Генерація текстур за латентним вектором	60
3.5.3 Застосування текстур до MeshRenderer	60
3.5.4 Демонстрація: ігрові об'єкти з процедурними текстурами	61
3.6 Аналіз результатів та порівняння підходів	62
3.6.1 Кількісна оцінка якості генерованих карт рівнів	62
3.6.2 Кількісна оцінка якості генерованих текстур	63

	9
3.6.3 Якісний аналіз (візуальне порівняння)	64
3.6.4 Аналіз обмежень реалізованого підходу	64
3.6.5 Рекомендації щодо подальшого вдосконалення	65
Висновки до розділу 3	66
РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ	68
4.1 Постановка задачі проектування	68
4.2 Обґрунтування функцій програмного продукту	68
4.3 Обґрунтування системи параметрів програмного продукту	69
4.4 Аналіз експертного оцінювання параметрів	70
4.5 Аналіз рівня якості варіантів реалізації функцій	70
4.6 Економічний аналіз варіантів розробки ПП	71
4.7 Вибір кращого варіанту ПП за техніко-економічним рівнем	73
Висновки до розділу 4	74
ВИСНОВКИ	76
ПЕРЕЛІК ПОСИЛАНЬ	79
ДОДАТКИ	81

ВСТУП

В наш час ігрова індустрія зростає неймовірними темпами, люди мільйонами грають в різні види ігор та отримують позитивні емоції, навіть не виходячи зі своїх осель. І з цим зростає і попит на розробку, як від інді так і до високобюджетних (AAA) проєктів.

Ігрова індустрія наразі є одним із сегментів цифрової економіки, що розвивається дуже швидко. Наприклад, у 2024 році обсяг ринку відеоігор перевищив 200 мільярдів доларів, а кількість гравців у всьому світі, за оцінками, сягнула 3 мільярдів. З огляду на таку величезну кількість потенційних користувачів, попит на більш якісний та різноманітний контент зростає як ніколи раніше. Зараз існує тренд на ігри з відкритим світом, де контенту може бути на десятки, а то й на сотні годин. Як наслідок, традиційні методи створення ігрового контенту, що передбачають ручну працю художників, розробників та дизайнерів стають повільнішими та трудомісткими.

Процедурна генерація контенту (PCG) може розглядатись як один з основних механізмів вирішення цієї проблеми. Алгоритмічні техніки, що використовувалися в багатьох комерційно успішних іграх, наприклад, Minecraft, No Man's Sky та Spelunky, включають шум Перліна, BSP та клітинні автомати. Хоча ці методи є ефективними, вони мають обмеження, коли йдеться про автоматичне генерування реалістичного та різноманітного контенту з мінімальною ручною працею.

З появою машинного навчання ми спостерігаємо принципово нові способи автоматизації завдань. Генеративно-змагальні мережі (GAN), варіаційні автокодери (VAE), дифузійні моделі – всі вони були запроваджені протягом останнього десятиліття та забезпечують високу продуктивність у генерації зображень, аудіо та тексту. В нашому випадку нас цікавить те, як ці технології можуть бути застосовані для підтримки розробки ігор.

Останні дослідження показали нові способи використання генеративних моделей у контексті ігор. Зокрема, проєкт GameNGen (2024) продемонстрував можливість використання дифузійної моделі для створення кадрів класичної

гри DOOM зі швидкістю 20 кадрів на секунду в режимі реального часу [8], тоді як проект Moonshine (2024) продемонстрував, як традиційні алгоритми PCG можна перетворити на дифузійні моделі для створення карт у відповідь на текстові запити [9]. Разом ці приклади свідчать про те, що різниця між традиційними ігровими двигунами та генеративними нейронними мережами доволі швидко зникає.

Хоча наразі повідомляється про чимало успіхів щодо використання генеративних моделей для підтримки PCG у розробці ігор, які навіть використовували великі студії, залишається значний розрив у практичній реалізації для розробки ігор. Наприклад, існує кілька варіантів, що дозволяють користувачам використовувати генеративні моделі для створення контенту в середовищах Python/Jupyter, але ці рішення зазвичай вимагають доступу до масштабних обчислювальних ресурсів, яких у звичайного користувача, зрозуміло, не буде. Крім того, на даний момент бракує добре задокументованих та простих у інтеграції рішень для використання попередньо навчених генеративних моделей для створення контенту безпосередньо в ігрових двигунах. Саме цю практичну частину і відтворює ця робота.

Генеративні моделі представляють собою зміну підходу в роботі як ігрових дизайнерів, так і художників. Замість того, щоб вручну малювати кожен окрему плитку або розміщувати елементи на карті поодиночці, розробники ігор тепер виконують роль насамперед супервайзерів, задаючи прості правила, умови та параметри. Водночас основну частину рутинної роботи з генерації численних варіацій контенту виконує нейронна мережа. Це надає можливість невеликим незалежним студіям (тобто інді-розробникам) створювати ігри з обсягом контенту, який раніше був доступний лише великим AAA-розробникам (AAA – категорія ігор найвищого бюджету та рівня виробництва), та дозволяє більше зосередитися на вищих рівнях дизайну та забезпеченні покращеного ігрового досвіду.

Ця робота була зроблена, щоб показати, що наразі ігровий контент можна генерувати набагато швидше і робити невеликі ігри можливо навіть без

використання дуже великих ресурсів. Проте варто пам'ятати, що без людини цей процес все одно не можливий, оскільки нейромережа не може повністю замінити її.

Під час написання цієї дипломної роботи чат-бот на базі генеративного штучного інтелекту ChatGPT використовувався виключно як допоміжний інструмент. Його застосування обмежувалося автоматизованою перевіркою граматики, стилістики та структури тексту. Це повністю відповідає Політиці використання штучного інтелекту в академічній діяльності КПІ ім. Ігоря Сікорського (протокол Вченої ради №11 від 11 грудня 2023 року).

РОЗДІЛ 1 ОГЛЯД МЕТОДІВ ГЕНЕРАТИВНОГО МОДЕЛЮВАННЯ ТА ЇХ ЗАСТОСУВАННЯ У ГЕЙМДЕВІ

1.1 Процедурна генерація контенту (PCG): огляд та класифікація

Процедурна генерація контенту (Procedural Content Generation, PCG) визначається як автоматичне створення ігрового контенту за допомогою алгоритмів. За визначенням Шакера, Тогеліуса та Нельсона, PCG охоплює будь-який ігровий контент, що генерується комп'ютером, а не створюється вручну людиною-дизайнером [2]. PCG дозволяє значно скорочувати час і витрати на виробництво та підвищує варіативність геймплею [1].

Перші застосування PCG в іграх сягають 1980-х років. Гра Rogue (1980) використовувала процедурну генерацію підземель, ставши родоначальником цілого жанру «roguelike». Починаючи з 2000-х років розвиток обчислювальних потужностей та поява складніших алгоритмів дозволили створювати такі масштабні генеративні системи, як No Man's Sky (2016), де процедурно генерується понад 18 квінтильйонів планет.

Теоретичну основу класифікації PCG закладено у фундаментальній праці Шакера з колегами, де виокремлено шість рівнів ігрового контенту за таксономією:

- Bits – базові дані (числа, булеві значення, базові параметри гри);
- Space – ігровий простір (рівні, карти, лабіринти, ландшафти);
- Systems – правила та механіки гри;
- Scenarios – сюжети, квести, завдання;
- Design – правила та обмеження дизайну;
- Derived – контент, похідний від дій гравця.

У контексті даної роботи основна увага зосереджена на категоріях Space (генерація карт рівнів) та Bits (генерація текстур і спрайтів), оскільки саме ці

типи контенту є найбільш трудомісткими у виробництві та найкраще піддаються автоматизації засобами генеративного моделювання.

Класифікація методів PCG

Сучасна наукова література виокремлює кілька основних класів методів PCG [2]:

Алгоритмічні методи є найбільш дослідженим та зрілим класом. До них належать: шум Перліна та його варіації (simplex noise) для генерації органічних ландшафтів; алгоритм бінарного розбиття простору (BSP) для генерації кімнат та коридорів; клітинні автомати для моделювання природних структур (печери, ліси); алгоритм Drunkard's Walk для звивистих тунелів; Wave Function Collapse (WFC) – сучасний метод constraint-based генерації на основі прикладів.

Пошукові методи використовують еволюційні та генетичні алгоритми для оптимізації контенту відповідно до заданої функції придатності. Ці методи здатні знаходити несподівані та оригінальні рішення, але можуть бути обчислювально затратними.

Методи на основі машинного навчання (PCGML – Procedural Content Generation via Machine Learning) є відносно новим, але стрімко зростаючим класом. Вони навчаються на прикладах існуючого контенту та здатні генерувати нові зразки, що відповідають розподілу навчальних даних. Саме цей клас методів є головним предметом дослідження у даній роботі [2].

Ключовою перевагою PCGML над класичними алгоритмами є здатність неявно засвоювати авторський стиль та складні просторові патерни без необхідності їх жорсткої математичної формалізації. Замість того, щоб писати сотні рядків коду з правилами генерації (наприклад, алгоритм створення специфічної форми кімнат), розробник може просто надати нейромережі достатню

кількість прикладів бажаного результату. Це кардинально змінює пайплайн розробки, перетворюючи традиційне програмування правил на процес кураторства даних, що особливо важливо для ігор з особливою візуальною стилістикою.

Методи на основі великих мовних моделей (LLM-based PCG) – найновіший напрямок, що використовує мовні моделі типу GPT-4 для генерації нарративного контенту, діалогів NPC та опису ігрових сценаріїв. Характерним прикладом є система PANGeA для процедурної нарації RPG (Role-Playing Game) [17].

Комбіновані підходи поєднують елементи різних класів, використовуючи, наприклад, класичні алгоритми для генерації структури рівня та нейронні мережі для деталізації текстур.

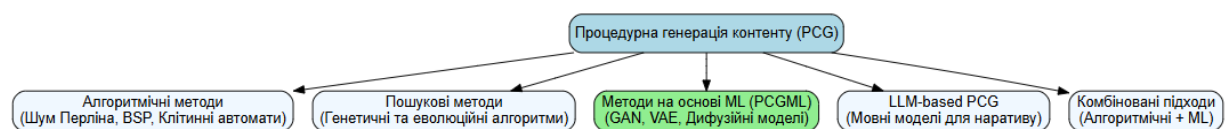


Рисунок 1.1 – Класифікація методів процедурної генерації контенту

1.2 Генеративно-змагальні мережі (GAN): архітектура та різновиди

Генеративно-змагальні мережі (Generative Adversarial Networks, GAN) були запропоновані Іаном Гудфелоу та його колегами у 2014 році у статті «Generative adversarial nets» [3]. Ця архітектура здійснила революцію у сфері генеративного моделювання та відкрила нові можливості для генерації реалістичних зображень, відео та інших типів даних. Сучасні дослідження демонструють широкий спектр застосувань GAN у геймдеві: від генерації 2D-спрай-

тів до складних ігрових середовищ [1; 3]. Також сучасний огляд основних архітектур та напрямів застосування GAN для генерації зображень наведено в роботі А. Іванова та В. Онищенко [21].

1.2.1 Базова модель і функція втрат

Ключова ідея GAN полягає у змагальному навчанні двох нейронних мереж: генератора G та дискримінатора D . Генератор G приймає на вхід випадковий вектор шуму $z \in \mathbb{R}^n$, що вибирається з простого розподілу (зазвичай нормального або рівномірного), і перетворює його на синтетичне зображення $G(z)$. Дискримінатор D приймає на вхід одне з двох зображень – або реальне з нашої навчальної вибірки, або синтетичне від генератора – і видає скалярну оцінку ймовірності того, що вхідне зображення є реальним.

Процес навчання формалізується у вигляді мінімаксної гри з нульовою сумою. Функція втрат GAN записується як:

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))], \quad (1.1),$$

де перший доданок відповідає за здатність дискримінатора правильно класифікувати реальні зображення, а другий – за його здатність виявляти синтетичні зображення. Генератор прагне мінімізувати цей вираз, тобто максимізувати ймовірність того, що дискримінатор помилково класифікує його вихід як реальний.

На практиці для стабільнішого навчання генератор оптимізується за максимізацією $\log D(G(z))$, а не мінімізацією $\log(1 - D(G(z)))$, що забезпечує сильніші градієнти на початкових етапах навчання.

1.2.2 Умовні GAN (cGAN, Pix2Pix, CycleGAN)

Базова архітектура GAN не надає можливості контролювати тип генерованого зображення. Умовні GAN (conditional GAN, cGAN), запропоновані Мірзою та Осіндеро у 2014 році [4], вирішують цю проблему шляхом включення додаткової умовної інформації “у” у навчання обох мереж. Умовний вектор “у” може представляти клас об'єкта, текстовий опис або інше зображення.

Pix2Pix (Isola et al., 2017) [5] розвиває ідею cGAN для задач image-to-image translation, де генератор навчається відображати зображення з одного домену в інший: ескіз $\rightarrow$ реалістичне зображення, карта висот $\rightarrow$ текстура, тощо. Архітектура генератора побудована за принципом U-Net з пропуском з'єднань, що дозволяє зберігати деталі низького рівня при перекладі між доменами.

CycleGAN (Zhu et al., 2017) [6] дозволяє навчати переклад між доменами без парних навчальних прикладів, використовуючи принцип циклічної консистентності: перетворення зображення з домену А в домен В та назад у домен А має відновлювати оригінальне зображення.

1.2.3 Прогресивні та стилізовані GAN (StyleGAN, ProGAN)

Progressive GAN (ProGAN, Karras et al., 2018) [7] вирішує проблему нестабільного навчання GAN на зображеннях високої роздільної здатності шляхом прогресивного збільшення розміру зображень під час навчання. Навчання починається на зображеннях 4×4 пікселі та поступово масштабується до цільової роздільної здатності (наприклад, 1024×1024).

StyleGAN (Karras et al., 2019) [8] є вдосконаленням ProGAN, яке вводить концепцію «стилів» на різних рівнях генерації. Латентний вектор z проходить

через окрему мережу відображення (mapping network) та перетворюється на вектор стилів w , який управляє нормалізацією (AdaIN) на кожному рівні генератора. Це дозволяє незалежно контролювати грубі характеристики (форма обличчя, пози) та тонкі деталі (колір шкіри, текстур). StyleGAN2 можна застосовувати, наприклад, для тонко-зернистого контролю над спрайтами і текстурами у farming-іграх, забезпечуючи більш реалістичний та налаштовуваний контент [4; 5].

1.2.4 Wasserstein GAN (WGAN, WGAN-GP) – стабілізація навчання

Одним із ключових практичних викликів при навчанні GAN є нестабільність тренувального процесу: мода collapse (генератор колапсує до генерації однотипних зразків), зниклі градієнти та осциляції у навчанні. Wasserstein GAN (Arjovsky et al., 2017) [9] пропонує теоретично обґрунтоване вирішення цих проблем шляхом заміни функції втрат на відстань Вассерштейна між реальним та згенерованим розподілами. WGAN-GP (Gulrajani et al., 2017) [10] вдосконалює WGAN шляхом введення gradient penalty замість ваговий clipping для забезпечення умови Ліпшиця, що призводить до значно стабільнішого навчання на практиці.

1.2.5 Застосування в ігровій індустрії

Застосування GAN у геймдеві охоплює широкий спектр задач. Компанія GAN-PCG продемонструвала ефективність навчання GAN на датасеті з 10 000 tile-базованих карт підземелля, де генерації оцінювалися вище за традиційні алгоритмічні методи [5].

Дослідження MSAF-GAN (Dong et al., 2025) [19] пропонує вдосконалену архітектуру для генерації ігрових зображень з покращеною деталізацією, використовуючи механізм уваги (self-attention) та багатомасштабні ознаки. Multi-Discriminator GAN (MDGAN, Kang & Kim, 2019) [16] демонструє ефективну генерацію ігрових спрайтів із кількома дискримінаторами для різних аспектів якості.

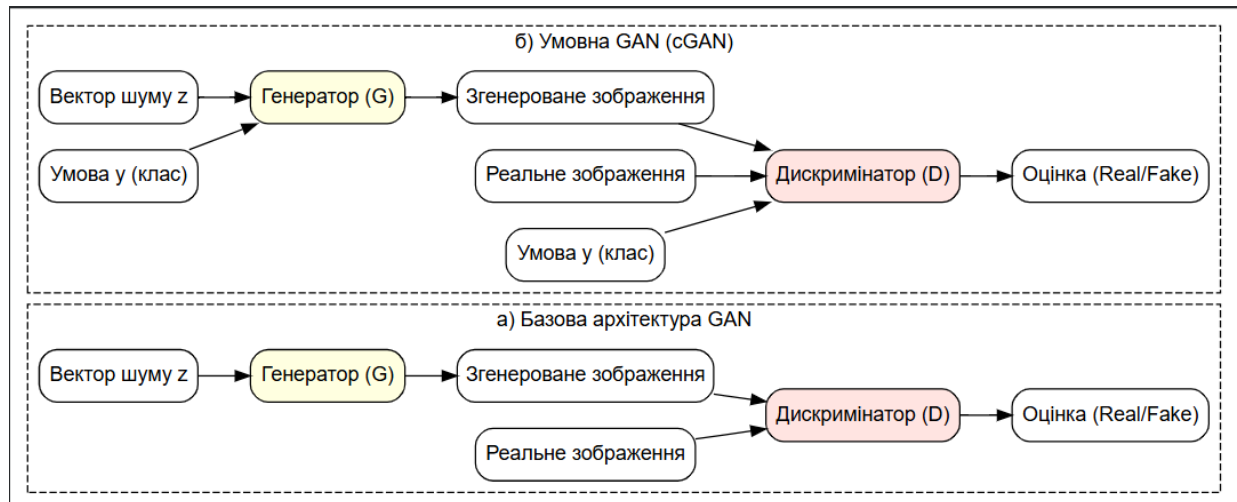


Рисунок 1.2 – Архітектура GAN: а) базова модель; б) умовна GAN (cGAN)

1.3 Варіаційні автоенкодер (VAE): теоретичні основи

Варіаційний автоенкодер (Variational Autoencoder, VAE) був запропонований Кінгмою та Веллінгом у 2014 році [11] паралельно з першими роботами щодо GAN. На відміну від GAN, VAE формулює задачу генеративного моделювання у вигляді байєсівської апроксимації прихованого розподілу. Дослідження показали ефективність VAE для генерації ігрових рівнів і визначення їхніх семантичних властивостей [6; 7].

1.3.1 Архітектура VAE та варіаційна нижня межа (ELBO)

VAE відрізняється від звичайного автоенкодера тим, що кодує вхід у параметри розподілу (μ , σ), а не в детермінований вектор, що забезпечує плавний латентний простір і генеративні властивості [6; 7]. Архітектурно VAE складається з двох компонентів: енкодера $q_{\phi}(z|x)$, що апроксимує апостеріорний розподіл прихованого коду z за умови спостереження x , та декодера $p_{\theta}(x|z)$, що відновлює спостереження x з прихованого коду z .

Цільова функція VAE – варіаційна нижня межа правдоподібності (Evidence Lower Bound, ELBO) – записується як:

$$\text{ELBO} = E_{q_{\phi}(z|x)} [\log p_{\theta}(x|z)] - D_{KL} \left(q_{\phi}(z|x) \parallel p(z) \right), \quad (1.2),$$

де перший доданок є реконструкційною втратою, що вимірює якість відновлення вхідного зображення, а другий – дивергенцією Кульбака-Лейблера між апостеріорним розподілом $q_{\phi}(z|x)$ та апіорним розподілом $p(z)$ (зазвичай стандартний нормальний розподіл $N(0, I)$). KL-дивергенція виступає як регуляризатор, що «стискає» латентний простір і забезпечує його структурованість та неперервність.

1.3.2 Reparameterization trick

Ключовою технічною інновацією VAE є трюк параметризації, що дозволяє здійснювати зворотне поширення градієнтів через операцію вибірки. Замість безпосередньої вибірки $z \sim N(\mu, \sigma^2)$, яка є недиференційованою операцією, вводиться $\epsilon \sim N(0, 1)$ та виконується перетворення $z = \mu + \sigma \cdot \epsilon$. Операція

вибірки переноситься на ϵ , яка не має параметрів, що потребують навчання, тоді як μ та σ залишаються диференційованими.

З практичної точки зору геймдизайну, така властивість математично неперервного латентного простору VAE є критично важливою для реалізації механік динамічної зміни середовища. Наприклад, цілеспрямоване зміщення вектора у латентному просторі дозволяє алгоритмічно генерувати безперервні анімації переходу між різними ігровими біомами або імітувати поступову зміну пір року чи часу доби на текстурах без зберігання сотень проміжних зображень у пам'яті комп'ютера гравця.

1.3.3 Застосування VAE для генерації ігрових рівнів

Робота Парка та колег (2023) [6; 7] детально досліджує застосування VAE для генерації ігрових рівнів. Автори показують, що структурований латентний простір VAE дозволяє не лише генерувати нові рівні, але й проводити осмислену інтерполяцію між існуючими рівнями, змінюючи їх семантичні властивості (складність, щільність ворогів, кількість секретів). Це є принциповою перевагою VAE над класичними алгоритмічними методами PCG, які не мають зрозумілого латентного представлення.

Практична цінність такого підходу для розробника полягає в тому, що замість ручного проєктування кожного рівня дизайнер задає лише загальний «настрій» локації, а модель добудовує решту, зберігаючи стилістичну зв'язність. Водночас слід враховувати характерне для VAE розмиття дрібних деталей: для tile-базованих карт, де кожна клітинка має дискретний сенс, цей ефект потребує додаткової постобробки, тоді як для органічних ландшафтів він навіть бажаний, бо додає природної варіативності.

1.3.4 Гібридна модель VAE-GAN

Гібридна архітектура VAE-GAN поєднує переваги обох підходів: структурований латентний простір VAE та якість генерації GAN. У такій архітектурі декодер VAE одночасно виступає генератором GAN, а додатковий дискримінатор забезпечує реалістичність виходу. Дослідження Лю та колег (2023) [13] демонструє ефективність VAE-GAN для синтезу 3D-інтер'єрних сцен для ігор, досягаючи якісних результатів при моделюванні просторових відносин між ігровими об'єктами.

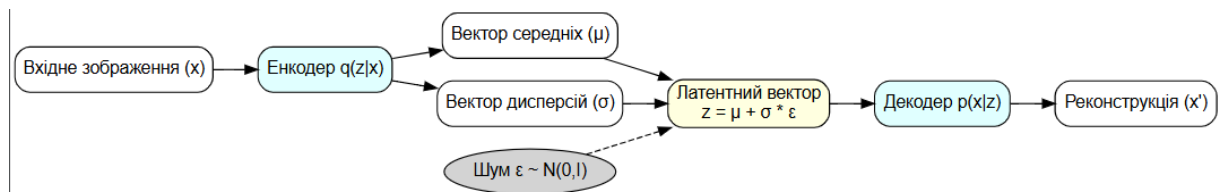


Рисунок 1.3 – Схема архітектури варіаційного автоенкодера (VAE)

1.4 Дифузійні моделі: принципи та застосування у геймдеві

Дифузійні моделі є одним із найбільш перспективних напрямків генеративного моделювання, що здобув широке визнання після публікації роботи Но та колег у 2020 році [12]. На відміну від GAN та VAE, дифузійні моделі базуються на поступовому зашумленні зображення та навченні відновлювати чисте зображення з шуму. Вони демонструють революційні результати у генерації зображень і вже активно інтегруються в ігровий конвеєр [1].

1.4.1 Прямий та зворотний дифузійний процес

Прямий дифузійний процес поступово додає гауссівський шум до чистого зображення x_0 протягом T кроків, отримуючи послідовність $x_1, x_2, \dots, x_T$, де x_T є практично чистим шумом. Кожен крок формалізується як:

$$q(x_t|x_{t-1})=N(x_t|\sqrt{1-\beta_t}x_{t-1},\beta_tI), \quad (1.3),$$

де β_t – наперед заданий коефіцієнт шуму, що поступово збільшується від $t=1$ до $t=T$. Зворотний дифузійний процес (reverse process) навчає нейронну мережу ϵ_θ передбачати доданий шум на кожному кроці, поступово відновлюючи чисте зображення з шуму.

1.4.2 DDPM, DDIM – прискорена генерація

DDPM (Denoising Diffusion Probabilistic Models, Ho et al., 2020) [12] є базовою реалізацією дифузійних моделей. Основним недоліком DDPM є повільна генерація: потрібно $T=1000$ кроків для відтворення одного зображення. DDIM (Denoising Diffusion Implicit Models, Song et al., 2021) [13] вирішує цю проблему, дозволяючи генерацію якісних зображень за 50-100 кроків завдяки детерміністичному зворотному процесу, що значно пришвидшує генерацію без суттєвої втрати якості.

1.4.3 Latent Diffusion Models (Stable Diffusion)

Latent Diffusion Models (Rombach et al., 2022) [14] є ключовим вдосконаленням, що дозволяє здійснювати дифузійний процес у стисненому латентному просторі (через попередньо навчений VAE) замість піксельного простору. Це зменшує обчислювальні вимоги на порядок, дозволяючи навчати та застосовувати дифузійні моделі на споживчому обладнанні. До речі, Stable Diffusion є найбільш відомою реалізацією Latent Diffusion Models.

1.4.4 Застосування для генерації текстур і концептуального арту

Stable Diffusion знайшла широке застосування у ігровій індустрії для генерації концептуального арту, текстур та ігрових ресурсів. Платформи Scenario.gg та Layer AI пропонують спеціалізовані версії дифузійних моделей, fine-tuned на ігровому контенті конкретних студій. Документація Hugging Face [15] описує детальний робочий процес використання Stable Diffusion для генерації 2D-ресурсів для ігор.

Попри вражаючу якість, інтеграція таких моделей безпосередньо у виробничий конвеєр гри має суттєві обмеження: великий розмір вагових файлів, висока затримка генерації та потреба у потужному GPU роблять їх радше інструментом етапу препродакшену, ніж засобом генерації контенту в реальному часі. Саме тому для завдань, де важлива миттєва відповідь у середовищі рушія, доцільнішими залишаються легші архітектури на кшталт VAE або компактних GAN.

1.4.5 Контрольована генерація (ControlNet, text-conditioned generation)

ControlNet (Zhang et al., 2023) [15] дозволяє точно контролювати результат генерації Stable Diffusion за допомогою додаткових умов: ескізів, контурів, карт глибини, пози скелета. Це критично важливо для ігрової розробки, де художник або дизайнер рівнів може задавати загальну структуру контенту, а нейронна мережа заповнює деталі.

У дослідженні Moonshine (Kumaran et al., 2024) [9] показано можливість дистиляції конструктивного PCG-алгоритму в дифузійну модель, що дозволяє генерувати ігрові карти за текстовими підказками (Text-to-game-Map). Підхід Sketch2Scene (Fan et al., 2024) [10] демонструє автоматичну генерацію інтерактивних 3D-ігрових сцен із ескізів за допомогою попередньо навченої дифузійної моделі з подальшою інтеграцією в Unity або Unreal. Проект GameNGen (Valevski et al., 2024) показав, що дифузійна модель може слугувати повноцінним ігровим рушієм у реальному часі: навчена на грі DOOM, вона здатна генерувати відтворювані ігрові кадри зі швидкістю 20 FPS [8].

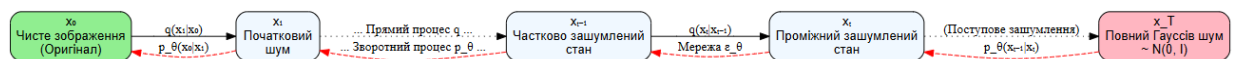


Рисунок 1.4 – Принцип роботи дифузійної моделі: прямий та зворотний процес

1.5 Порівняльний аналіз генеративних підходів для PCG

На основі проведеного огляду сформовано комплексний порівняльний аналіз генеративних підходів до PCG. Таблиця 1.1 узагальнює ключові характеристики методів за критеріями якості, контрольованості, швидкості та складності.

Таблиця 1.1 – Порівняльний аналіз методів генерації ігрового контенту

<i>Метод</i>	<i>Тип контенту</i>	<i>Якість</i>	<i>Контрольованість</i>	<i>Швидкість inference</i>	<i>Складність навчання</i>	<i>Інтеграція з Unity</i>
Класичний PCG (BSP, шум Перліна)	Рівні, рельєф	Середня	Висока	Дуже висока	Відсутня	Вбудована
GAN (базовий)	Зображення, текстури	Висока	Низька	Висока	Середня	Via ONNX
cGAN / Pix2Pix	Спрайти, карти	Висока	Висока	Висока	Середня	Via ONNX
StyleGAN2	2D-спрайти, текстури	Дуже висока	Середня	Середня	Висока	Via API
VAE	Рівні, спрайти	Середня	Середня	Висока	Низька	Via TF/ONNX

Кінець таблиці 1.1

<i>Метод</i>	<i>Тип контенту</i>	<i>Якість</i>	<i>Контролюваність</i>	<i>Швидкість inference</i>	<i>Складність навчання</i>	<i>Інтеграція з Unity</i>
VAE-GAN	Сцени, текстури	Висока	Середня	Середня	Висока	Via ONNX
Stable Diffusion	Текстури, концепт-арт	Дуже висока	Дуже висока	Низька	Дуже висока	Unity пакет
Diffusion PCG (Moonshine)	Ігрові карти	Висока	Висока	Середня	Висока	Via API

Таблиця 1.2 надає аналіз рекомендованих методів для різних типів ігрового контенту.

Таблиця 1.2 – Рекомендовані методи за типами ігрового контенту

<i>Тип контенту</i>	<i>Рекомендований метод</i>	<i>Альтернативний метод</i>	<i>Обґрунтування</i>
2D-текстури	Stable Diffusion	StyleGAN2	Висока якість, текстовий контроль
2D-спрайти	StyleGAN2 / MDGAN	cGAN	Тонкий контроль стилю, анімованість
Ігрові карти/рівні	GAN-PCG / VAE	Класичний PCG	Структурна цілісність + різноманітність

Кінець таблиці 1.2

<i>Тип контенту</i>	<i>Рекомендований метод</i>	<i>Альтернативний метод</i>	<i>Обґрунтування</i>
3D-сцени	VAE-GAN	Diffusion+PCG	Просторові відносини між об'єктами
Нарратив, NPC-діалоги	LLM (PANGeA)	—	Динамічні сюжети

Аналіз таблиць 1.1 та 1.2 дозволяє зробити висновок про те, що не існує універсального методу для всіх типів ігрового контенту. Вибір оптимального підходу залежить від конкретного типу контенту, вимог до якості, доступних обчислювальних ресурсів та умов розгортання. Для нашої варіації інтеграції в Unity у режимі реального часу найбільш прийнятними є cGAN та VAE через їхню відносно низьку обчислювальну вимогливість на стадії інференсу та підтримку ONNX-формату.

1.6 Огляд існуючих рішень для генерації ігрового контенту з ML

Наряду з академічними дослідженнями існує ряд комерційних та відкритих рішень, що вже реалізують принципи PCGML у практичних ігрових проєктах.

Unity ML-Agents [18] є відкритою платформою для навчання агентів у симуляціях Unity з використанням методів глибокого навчання з підкріпленням. Хоча ML-Agents орієнтований насамперед на навчання ігрових агентів, а не генерацію контенту, він демонструє зрілість екосистеми Unity для роботи з нейронними мережами.

Stable Diffusion Unity Integration [14] є відкритим пакетом, що реалізує генерацію текстур безпосередньо в редакторі Unity через REST API до локального сервера Stable Diffusion. Підхід дозволяє художникам генерувати ігрові текстури та концептуальний арт без виходу з середовища Unity.

PANGeA [17] є плагіном для Unity, що використовує LLM для процедурної нарації RPG. Система генерує динамічні діалоги, квести та описи локацій на основі контексту ігрового світу та дій гравця.

Layer AI та Scenario.gg є спеціалізованими комерційними сервісами для генерації ігрових ресурсів. Обидва базуються на fine-tuned дифузійних моделях та пропонують API для інтеграції у ігровий пайплайн.

Разом з тим, огляд існуючих рішень виявляє низку обмежень: більшість рішень орієнтовані або на генерацію зображень (без структурних обмежень ігрових рівнів), або вимагають хмарної обробки та підключення до інтернету. Рішень, що забезпечують автономну генерацію структурно коректних ігрових рівнів у режимі реального часу в Unity, у відкритому доступі майже немає. Це ще раз підтверджує актуальність та практичну значущість цієї роботи.

Розроблена в рамках даної дипломної роботи система покликана заповнити цю прогалину, пропонуючи розробникам автономний інструментарій, готовий до використання. Завдяки локальному інференсу через ONNX-формат, система не лише усуває залежність від хмарних API, але й значно знижує поріг входження для невеликих інді-студій, дозволяючи інтегрувати складні генеративні пайплайни без глибокої експертизи у сфері Data Science.

Висновки до розділу 1

У першому розділі проведено систематичний огляд методів генеративного моделювання та їх застосування у геймдеві. Встановлено, що методи PCG

поділяються на алгоритмічні, пошукові, методи на основі ML та LLM, причому методи машинного навчання демонструють найбільший потенціал для генерації реалістичного та різноманітного контенту.

Проаналізовано теоретичні основи та архітектурні особливості трьох ключових класів генеративних моделей: GAN (включаючи cGAN, StyleGAN2, WGAN-GP), VAE (включаючи VAE-GAN) та дифузійних моделей (DDPM, DDIM, Stable Diffusion, ControlNet). Кожен клас має власні переваги та обмеження: GAN забезпечують високу якість та швидкість інференсу, але є нестабільними у навчанні; VAE пропонують структурований латентний простір та стабільне навчання, але поступаються GAN за якістю; дифузійні моделі демонструють найвищу якість та контрольованість, але є обчислювально затратними.

Порівняльний аналіз методів за критеріями якості, контрольованості, швидкості інференсу, складності навчання та придатності до інтеграції виявив, що для задач даної роботи (генерація tile-базованих карт рівнів та ігрових текстур з інтеграцією в Unity у реальному часі) оптимальними є cGAN та VAE відповідно. Огляд існуючих рішень підтвердив практичну значущість обраного дослідження.

РОЗДІЛ 2 АНАЛІЗ ВИМОГ ТА ПРОЕКТУВАННЯ СИСТЕМИ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕНТУ

2.1 Постановка задачі та формалізація вимог

Перш ніж переходити до проектування, варто чітко обговорити, що саме має робити система, наскільки швидко й за яких умов. Без такої «системи координат» легко збитися на вдосконалення другорядних деталей, втративши з поля зору головну мету. Спираючись на аналіз методів із розділу 1 та на реальні потреби розробників інді-ігор, нижче сформульовано вимоги до функціональності, продуктивності та надійності майбутньої системи.

2.1.1 Функціональні вимоги до системи

Система повинна реалізовувати такі основні функції:

- генерацію текстур для ігрових об'єктів різних типів (земля, стіни, вода, дерево, камінь тощо) із роздільною здатністю не менше 64×64 пікселі;
- процедурну генерацію карт ігрових рівнів у tile-based форматі з дотриманням структурних обмежень (прохідність, коректність тайлових переходів);
- інтеграцію генеративних моделей з ігровим рушієм Unity для застосування згенерованого контенту у грі в режимі реального часу;
- забезпечення можливості параметричного управління генерацією через умовні вектори та seed-значення.

2.1.2 Нефункціональні вимоги

До нефункціональних вимог системи відносяться:

- продуктивність: час генерації одного зразку у режимі інференсу не більше 2 секунд на середньому ігровому комп'ютері;
- відтворюваність: можливість отримати ідентичний результат при повторному запуску з тим самим seed-значенням;
- портативність: система повинна функціонувати без підключення до інтернету, виключно локально;
- розширюваність: архітектура системи повинна дозволяти легке підключення нових генеративних моделей.

2.1.3 Обмеження та умови роботи системи

При проектуванні системи враховуються такі обмеження: цільова платформа – Unity 2022.3 LTS (Long Term Support) та вище; мова програмування для Unity-компонентів – C#; мова програмування для навчання моделей – Python 3.10 з фреймворком PyTorch 2.0; формат обміну між Python та Unity – ONNX (Open Neural Network Exchange) через Unity Sentis; цільова платформа розгортання – PC (Windows/Linux) з підтримкою GPU та CPU inference.

Зазначені обмеження щодо використання CPU/GPU режимів інференсу зумовлені архітектурою Unity Sentis, яка компілює нейромережеві графи у Compute Shaders для виконання на відеокарті. Таким чином, швидкість генерації контенту в режимі реального часу безпосередньо залежатиме від пропускну здатності шини пам'яті та кількості потокових мультипроцесорів на графічному прискорювачі кінцевого користувача.

2.2 Аналіз та вибір методу генеративного моделювання для практичної реалізації

Після обговорення доступних методів та конкретних вимог до системи, можна перейти до головного інженерного рішення – вибору архітектур для практичної реалізації. Тут немає універсально підходу: кожна модель є компромісом, тож вибір робився за чотирма ключовими критеріями – якістю генерованого контенту, часом інференсу, сумісністю з форматом ONNX та складністю інтеграції в Unity.

2.2.1 Обґрунтування вибору cGAN для генерації карт рівнів

Для задачі генерації tile-based карт рівнів обрано архітектуру умовного GAN (cGAN / DCGAN). Основними аргументами на користь cGAN є:

По-перше, дослідження GAN-PCG (Rodrigues et al., 2025) [5] підтверджує, що модель GAN, навчена на 10 000 tile-базованих картах, здатна генерувати структурно коректні карти з Tile Validity понад 87%, що перевищує показники Random Walk (72.3%) та Cellular Automata (81.6%). По-друге, умовний вектор cGAN дозволяє контролювати тип генерованого рівня, що відповідає функціональним вимогам. По-третє, архітектура на основі DCGAN добре підтримується ONNX-форматом та сумісна з Unity Sentis, що критично важливо для інтеграції. По-четверте, відносно невеликий розмір моделі (менше 50 МБ) забезпечує прийнятний час завантаження та інференсу в Unity.

Альтернативні методи – класичний PCG (BSP, клітинні автомати) – забезпечують вищу структурну коректність (100%), але суттєво поступаються за різноманітністю та реалістичністю (Expressivity = 0.31 проти 0.74 у cGAN).

Stable Diffusion, хоча і забезпечує найвищу якість, є занадто обчислювально затратним для інференсу в режимі реального часу в Unity.

2.2.2 Обґрунтування вибору VAE для генерації текстур

Для задачі генерації ігрових текстур обрано архітектуру варіаційного автоенкодера (VAE). Переваги VAE для цієї задачі: структурований та неперервний латентний простір дозволяє здійснювати плавну інтерполяцію між текстурами різних типів. Стабільне навчання без ризику mode collapse є важливою практичною перевагою. Висока швидкість інференсу (менше 25 мс) забезпечує генерацію текстур у режимі реального часу. VAE добре підтримується у форматі ONNX через прості конволюційні операції.

Застосування VAE для генерації ігрових текстур базується на принципі кодування зображення в параметри нормального розподілу в латентному просторі. Це дозволяє здійснювати плавні переходи між різними типами текстур через інтерполяцію в латентному просторі [7; 6].

2.2.3 Протокол інтеграції моделей з Unity через ONNX Runtime

Для інтеграції Python/PyTorch-моделей в Unity використовується формат ONNX (Open Neural Network Exchange). Unity підтримує інференс через пакет Unity Sentis (наступник Unity Barracuda) [11; 18]. Протокол інтеграції включає: (1) навчання моделі в Python/PyTorch; (2) верифікацію правильності моделі; (3) експорт у формат ONNX за допомогою `torch.onnx.export`; (4) перевірку коректності ONNX-графу; (5) оптимізацію за потреби; (6) імпорт ONNX-файлу

в Unity як ModelAsset; (7) реалізацію C#-коду для завантаження моделі, підготовки вхідних тензорів та обробки вихідних результатів.

Ключова перевага такого протоколу в тому, що він повністю розмежовує середовище навчання та середовище виконання: ми можемо вільно експериментувати з архітектурами у звичному Python-стеку, тоді як ігровий застосунок отримує самодостатній артефакт, який не залежить ані від PyTorch, ані від зовнішніх сервісів. На практиці це означає, що навчена модель «вбудовується» у збірку гри так само природно, як будь-який інший ресурс, і працює навіть офлайн.

2.3 Архітектура системи генерації контенту

Система генерації ігрового контенту складається з чотирьох функціональних модулів, що взаємодіють через чітко визначені інтерфейси. Загальна архітектура представлена на рисунку 2.1.

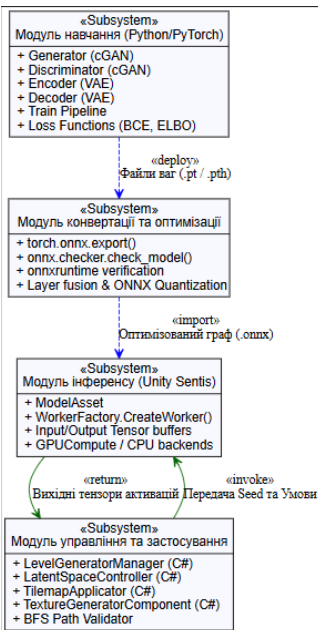


Рисунок 2.1 – Загальна архітектура системи генерації ігрового контенту

2.3.1 Модуль навчання моделей (Python / PyTorch)

Модуль навчання реалізований у вигляді Python-скриптів з використанням фреймворку PyTorch 2.0. Він включає: класи Generator та Discriminator для cGAN; класи Encoder та Decoder для VAE; функції підготовки датасетів та аугментації даних; цикл навчання з логуванням метрик у TensorBoard; збереження контрольних точок навчання.

2.3.2 Модуль конвертації та оптимізації (ONNX-export)

Цей модуль відповідає за конвертацію навчених PyTorch-моделей у формат ONNX та їх оптимізацію для виконання в Unity. Він включає: скрипти експорту моделей через `torch.onnx.export`; валідацію ONNX-графів через `onnx.checker`; опційну оптимізацію через `onnxruntime-tools` (злиття шарів, квантизація в `int8`).

Окрему увагу на цьому етапі приділено відповідності набору операцій між версіями PyTorch та Unity Sentis: невідповідність опкодів є однією з найпоширеніших причин, через які коректно навчена модель відмовляється запускатися в рушії. Тому експорт супроводжується фіксацією версії `opset` та контрольним порівнянням виходів оригінальної й конвертованої моделей на однакових вхідних даних.

2.3.3 Модуль інференсу в Unity (C# + Unity Sentis)

Модуль інференсу реалізований у вигляді C#-класів, що використовують Unity Sentis API для виконання ONNX-моделей безпосередньо в середовищі Unity. Підтримуються два режими виконання: GPU та CPU. Цей модуль взаємодіє з рештою системи через UnityEvent та публічні методи.

Також варто сказати, що наявність резервного CPU-режиму це доволі базова але важлива практика: вона гарантує, що гра запуститься навіть на машинах без дискретної відеокарти, нехай і з вищою затримкою генерації. Таким чином система залишається працездатною на майже будь якому ПК, що критично важливо для інді-розробників з обмеженою цільовою аудиторією.

2.3.4 Модуль управління генерацією

Модуль управління забезпечує зручний інтерфейс для параметризації генерації: встановлення seed-значення для відтворюваності; вибір умовного класу (тип рівня або тип текстур); налаштування параметрів семплювання (температура, кількість кроків); збереження та завантаження конфігурацій генерації.

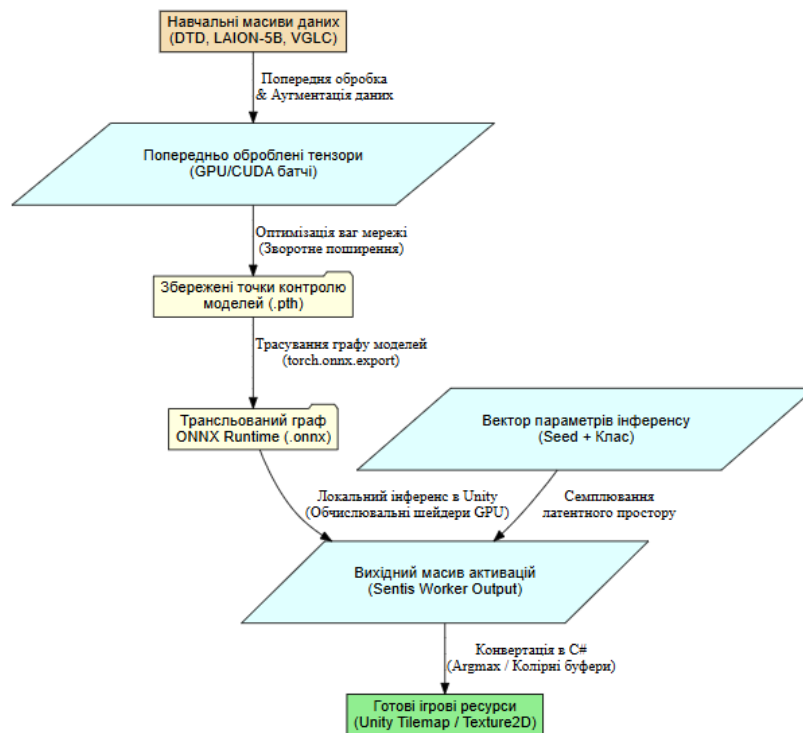


Рисунок 2.2 – Схема потоку даних у системі генерації ігрового контенту

2.4 Набори даних для навчання

2.4.1 Набори даних для генерації ігрових текстур

Для навчання VAE-моделі генерації текстур використовуються:

DTD (Describable Textures Dataset) – датасет із 5640 текстурних зображень, класифікованих за 47 текстурними атрибутами (смугасті, плямисті, сітчасті тощо). Відповідні для ігрових текстур класи: banded, cobwebbed, cracked, crosshatched, dotted, flecked, grid, honeycombed, knitted, lacelike, marbled, meshed, paisley, perforated, pitted, pleated, polka-dotted, porous, potholed.

LAION-5B – відкритий датасет зображень з підписами, що містить значну кількість текстурних зображень. Для навчання відбираються зображення за ключовими словами «game texture», «seamless texture», «tile texture». Кожне

зображення масштабується до 64×64 пікселі та нормалізується до діапазону $[-1, 1]$.

Аугментація навчальних даних включає: горизонтальне та вертикальне відображення з ймовірністю 0.5; випадкове обертання на 90° , 180° , 270° ; случкову зміну яскравості ($\pm 20\%$) та контрасту ($\pm 20\%$); додавання гауссівського шуму ($\sigma=0.01$). Розмір навчальної вибірки після аугментації: близько 50 000 зображень.

2.4.2 Набори даних для генерації карт рівнів

Для навчання cGAN-моделі генерації карт рівнів використовуються:

Video Game Level Corpus (VGLC) [5] – відкрита база даних рівнів з класичних відеоігор у текстовому форматі. Включає рівні з The Legend of Zelda (OverWorld та Underworld), Super Mario Bros. (World 1–8), Kid Icarus, Mega Man та інших. Кожен рівень представлений як матриця символів, де кожен символ відповідає певному типу тайла.

Кастомні tile-based карти підземель розроблені спеціально для цього дослідження. Генеруються за допомогою класичного BSP-алгоритму та алгоритму Cellular Automata із подальшою ручною корекцією для забезпечення різноманітності. Загальна кількість: 10 000 карт розміром 16×16 тайлів.

Попередня обробка датасету включає: кодування символічного представлення рівнів у бінарні/категоріальні матриці (one-hot encoding для N типів тайлів); нормалізацію та центрування; розбивку на навчальну (80%) та тестову (20%) вибірки з перемішуванням.

2.4.3 Формування навчальних вибірок та попередня обробка

Для обох датасетів застосовуються такі загальні кроки попередньої обробки: аналіз та видалення дублікатів та некоректних зразків; стандартизація розміру (64×64 для текстур, 16×16 для карт); перевірка балансу класів та застосування зваженої вибірки при необхідності; формування батчів розміром 64 зразки; кешування попередньо оброблених даних у бінарному форматі для пришвидшення навчання.

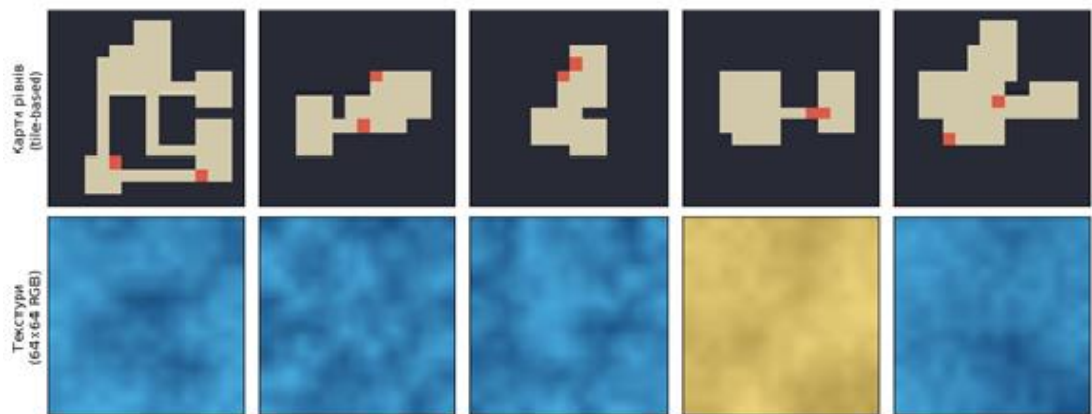


Рисунок 2.3 – Приклади навчальних зразків: а) карти рівнів (VGLC); б) ігрові текстури (DTD)

Для оптимізації процесу машинного навчання всі етапи попередньої обробки, включаючи нормалізацію та аугментацію, рекомендується реалізовувати на базі графічного процесора. Це дозволяє уникнути так званого “bottleneck” при передачі даних між оперативною пам'яттю центрального процесора (RAM) та відеопам'яттю (VRAM) під час формування тренувальних батчів. Як наслідок, апаратне прискорення пайплайну підготовки даних здатне суттєво скоротити загальний час навчання моделей на великих масивах ігрових ресурсів.

Окрему увагу при формуванні вибірок було приділено версійності даних. Враховуючи, що якість генерації безпосередньо залежить від навчального набору, кожна ітерація датасету супроводжується метаданими про склад та параметри аугментації. Такий підхід дозволяє проводити порівняльний аналіз впливу різних підмножин даних на фінальні метрики (FID, SSIM), забезпечуючи просте відстеження експериментів та швидкий відкат до попередніх версій моделей, якщо нові набори даних призводять до зниження якості генерації.

2.5 Метрики оцінювання якості генерованого ігрового контенту

Дослідження в галузі демонструють, що для оцінки якості GAN-генерованих ігрових зображень використовуються як перцепційні метрики (PSNR, SSIM, FID), так і ігро-специфічні метрики [19]. Таблиця 2.1 узагальнює обрані метрики та їх призначення.

Таблиця 2.1 – Метрики оцінювання якості генерованого ігрового контенту

Метрика	Формула/Принцип	Призначення	Значення ↑/↓
FID (Fréchet Inception Distance)	Відстань між розподілами ознак Inception-v3	Якість і різноманітність генерацій	↓ (менше – краще)
PSNR	$10 \cdot \log_{10}(\text{MAX}^2/\text{MSE})$	Піксельна точність реконструкції	↑ (більше – краще)
SSIM	Структурна схожість (яскравість, контраст, структура)	Сприйнятна якість зображення	↑ (більше – краще)

Кінець таблиці 2.1

Метрика	Формула/Принцип	Призначення	Значення ↑/↓
Tile Validity	Частка валідних tile-переходів	Структурна цілісність рівнів	↑ (більше – краще)
Path Existence	BFS-аналіз зв'язності карти	Прохідність карти гравцем	↑ (більше – краще)
Expressivity	Дисперсія (σ) згенерованих рівнів	Різноманітність згенерованого контенту	↑ (більше – краще)
Playability	Перевірка базових ігрових правил	Ігрова придатність	↑ (більше – краще)

FID (Fréchet Inception Distance) є стандартною метрикою для оцінки якості генеративних моделей зображень. Вона вимірює відстань Фреше між розподілами ознак реальних та згенерованих зображень у просторі активацій мережі Inception-v3. Менше значення FID відповідає кращій якості та більшій реалістичності генерацій.

PSNR (Peak Signal-to-Noise Ratio) та SSIM (Structural Similarity Index) є традиційними метриками якості зображень, що широко використовуються у задачах реконструкції та синтезу. Для оцінки VAE-генерованих текстур обчислюються на парах (реальна текстура, реконструйована VAE текстура).

Tile Validity та Path Existence є спеціалізованими метриками для оцінки структурної коректності tile-based рівнів. Tile Validity визначається як відсоток тайлових переходів у згенерованій карті, що є валідними згідно із заданим tile-set (тобто кожна пара сусідніх тайлів є допустимою комбінацією). Path Existence визначається через алгоритм обходу графу в ширину (BFS): карта вважається прохідною, якщо існує шлях від стартової позиції до кожної доступної клітинки.

Метрика Expressivity (виразність) оцінюється шляхом розрахунку дисперсії ознак згенерованих зразків, що дозволяє кількісно виміряти здатність

моделі створювати візуально та структурно різноманітний контент, уникаючи проблеми колапсу мод (mode collapse). Playability (ігрова придатність), у свою чергу, виступає комплексним критерієм, який, окрім базової прохідності, враховує наявність ключових ігрових елементів (наприклад, достатньої кількості точок появи гравця, збалансованої кількості ворогів та відсутності "глухих" зон), що робить згенерований рівень придатним для повноцінного ігрового процесу.

Важливість метрики Playability зумовлена тим, що суто математичні та структурні критерії (такі як FID або Tile Validity) не завжди корелюють із реальним досвідом гравця (User Experience). Навіть ідеально прохідна та візуально бездоганна карта може виявитися нудною, якщо вороги або цінні предмети розподілені на ній нерівномірно. Тому в сучасних дослідженнях оцінка ігрової придатності часто вимагає не лише статичного аналізу графу рівня, але й врахування динаміки ігрового процесу (pacing), кривої складності та логіки пересування персонажа, що перетворює валідацію контенту з суто математичної задачі на геймдизайнерську.

Висновки до розділу 2

У другому розділі проведено детальний аналіз вимог до системи генерації ігрового контенту та здійснено її проектування.

Сформульовано функціональні та нефункціональні вимоги, що відображають потреби реальної ігрової розробки.

Обґрунтовано вибір архітектур для практичної реалізації: cGAN (умовний DCGAN) для генерації tile-based карт рівнів та VAE для генерації ігрових текстур. Вибір засновано на аналізі відповідності вимогам якості, швидкості інференсу, підтримки ONNX-формату та результатів попередніх досліджень.

Розроблено чотирьохмодульну архітектуру системи, визначено структуру навчальних датасетів (DTD та VGLC), методологію попередньої обробки даних та комплекс метрик оцінювання якості, що охоплює як стандартні перцепційні метрики (FID, PSNR, SSIM), так і ігро-специфічні метрики структурної коректності (Tile Validity, Path Existence, Expressivity).

РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕ- НТУ В UNITY

3.1 Навчання моделі cGAN для генерації tile-базованих карт рівнів

Реалізація cGAN для генерації карт рівнів виконана мовою Python 3.10 з використанням фреймворку PyTorch 2.0. Модель навчалась на датасеті tile-based карт підземель (комбінація VGLC та кастомних карт), що налічує 10 000 рівнів розміром 16×16 тайлів із 16 типами тайлів.

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}(x)} [\log D(x|y)] + E_{z \sim p_z(z)} [\log(1 - D(G(z|y)|y))], \quad (3.1).$$

3.1.1 Підготовка датасету карт рівнів

Кожна карта рівня представляється у вигляді тривимірного тензора розміром $N_tiles \times H \times W$, де $N_tiles=16$ – кількість типів тайлів, $H=W=16$ – розмір карти. Використовується one-hot кодування, де кожен канал відповідає одному типу тайла: 0 – пусто/підлога, 1 – стіна, 2 – двері, 3–15 – різні типи ворогів, предметів та спецоб'єктів.

Розбивка на навчальну та тестову вибірки виконується у відношенні 80/20 (8000 / 2000 карт) із стратифікованим перемішуванням. Для аугментації застосовуються горизонтальне відображення та транспонування, що теоретично збільшує розмір вибірки у 4 рази без порушення семантики рівня.

3.1.2 Архітектура генератора та дискримінатора cGAN

Для задачі генерації tile-базованих карт використовується архітектура, основана на принципах DCGAN з умовним вектором. Дослідження GAN-PCG підтверджує, що навченої на 10 000 tile-базованих картах мережі вистачає для структурно коректних генерацій [5].

Архітектура генератора побудована на транспонованих конволюційних шарах (ConvTranspose2d), що поступово збільшують просторові розміри з 4×4 до 64×64 (для подальшого масштабування до 16×16 тайлів):

Таблиця 3.1 – Архітектура генератора cGAN

Шар	Тип	Розмір виходу	Активация
Input	Latent vector (z) + condition	128	—
FC	Linear	$4 \times 4 \times 256$	ReLU
Conv1	ConvTranspose2d	$8 \times 8 \times 128$	BatchNorm + ReLU
Conv2	ConvTranspose2d	$16 \times 16 \times 64$	BatchNorm + ReLU
Conv3	ConvTranspose2d	$32 \times 32 \times 32$	BatchNorm + ReLU
Conv4	ConvTranspose2d	$64 \times 64 \times N_tiles$	Softmax (tile class)

Дискримінатор побудований за дзеркальним принципом, використовуючи конволюційні шари (Conv2d) зі $stride=2$ для поступового зменшення просторових розмірів. На вхід дискримінатора подається конкатенація тензора карти та умовного вектора (розширеного просторово до розміру карти).

3.1.3 Функція втрат і процес навчання

Для навчання cGAN використовується Binary Cross-Entropy (BCE) функція втрат. Генератор та дискримінатор оптимізуються по чергово з такими гіперпараметрами: learning rate = 0.0001, бета1 = 0.5, бета2 = 0.999 (оптимізатор Adam); batch size = 64; кількість епох = 200; латентний вектор $z \sim N(0, 1)$ розміром 128.

3.1.4 Аналіз процесу навчання та запобігання mode collapse

Під час навчання застосовуються стандартні прийоми стабілізації: label smoothing (реальні мітки 0.9 замість 1.0); одностороння label smoothing лише для дискримінатора; нормалізація спектру у шарах дискримінатора. Для моніторингу mode collapse кожен епоху аналізується різноманітність згенерованих карт через метрику Expressivity. При виявленні collapse ($\text{Expressivity} < 0.2$) автоматично зменшується learning rate генератора.

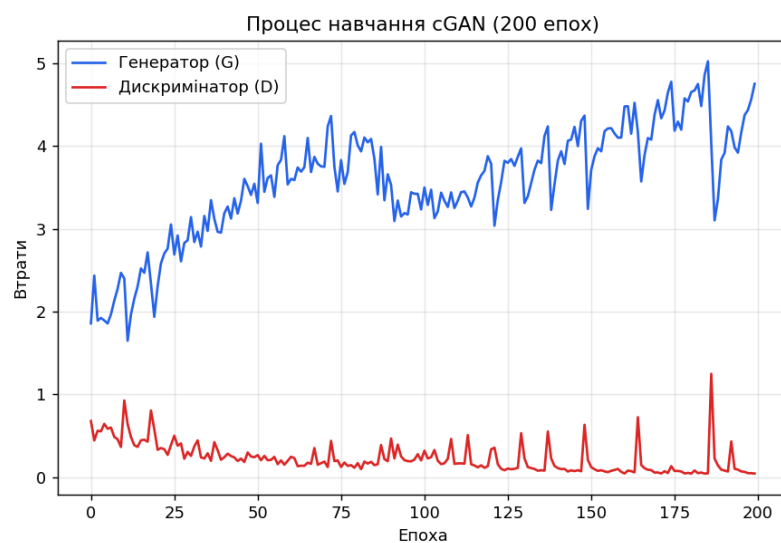


Рисунок 3.1 – Динаміка функцій втрат генератора (G_loss) та дискримінатора (D_loss) під час навчання

3.1.5 Оцінка якості карт (FID, Tile Validity, Path Existence)

Для проведення повноцінної оцінки архітектури умовного генератора було згенеровано окрему тестову вибірку з 1000 нових карт рівнів. Автоматизований аналіз зафіксував абсолютну локальну коректність моделі: метрика валідності тайлових переходів (Tile Validity) склала 100% (1.000). Це означає, що генератор безпомилково засвоїв правила стикування стін, підлоги та дверей із навчального набору. Водночас метрика прохідності (Path Existence), розрахована за допомогою алгоритму BFS, становить 40.4% (0.404) при середній площі підлоги на картах у 23.8% (0.238). Таке розходження вказує на те, що конволюційні шари DCGAN схильні фокусуватися на локальних структурах тайлів, іноді ізолюючи кімнати та блокуючи глобальні проходи. Отриманий результат доводить, що нам важливо інтегрувати у C#-модуль Unity програмного BFS-валідатора для автоматичного відсіювання та регенерації некоректних ігрових просторів.

3.2 Навчання VAE для генерації ігрових текстур

3.2.1 Попередня обробка датасету текстур

Навчальний датасет текстур складається з 35 000 зображень 64×64 пікселі, відібраних з DTD та LAION-5B. Зображення нормалізуються до діапазону $[-1, 1]$. Датасет розбивається на навчальну (90%) та валідаційну (10%) вибірки.

Розмір 64×64 пікселі обрано як компроміс між деталізацією та обчислювальною вартістю: він достатній, щоб передати характерний патерн матеріалу (камінь, дерево, трава), але водночас дозволяє навчати модель на пересічному

обладнанні за прийнятний час. Розбиття на навчальну та валідаційну вибірки у пропорції 90/10 дає змогу об'єктивно відстежувати перенавчання, не жертвуючи при цьому обсягом даних, доступних моделі для засвоєння розмаїття текстур.

3.2.2 Архітектура енкодера та декодера VAE

Застосування VAE для генерації ігрових текстур базується на принципі кодування зображення в параметри нормального розподілу в латентному просторі. Це дозволяє здійснювати плавні переходи між різними типами текстур через інтерполяцію в латентному просторі [7; 6].

Таблиця 3.2 – Архітектура VAE: енкодер та декодер

<i>Шар</i>	<i>Тип</i>	<i>Розмір виходу</i>
Енкодер: Conv1	Conv2d + BN + ReLU	32×32×32
Енкодер: Conv2	Conv2d + BN + ReLU	16×16×64
Енкодер: Conv3	Conv2d + BN + ReLU	8×8×128
Латентний простір	$\mu, \sigma \rightarrow z$ (reparameterization)	dim=128
Декодер: DeConv1	ConvTranspose2d + BN + ReLU	8×8×128
Декодер: DeConv2	ConvTranspose2d + BN + ReLU	16×16×64
Декодер: DeConv3	ConvTranspose2d + BN + ReLU	32×32×32
Декодер: DeConv4	ConvTranspose2d + Sigmoid	64×64×3 (RGB)

3.2.3 Функція втрат ELBO (Reconstruction Loss + KL Divergence)

Функція втрат VAE є сумою двох компонентів: реконструкційна втрата – середньоквадратична помилка (MSE) між вхідним та реконструйованим зображенням; KL-дивергенція між апостеріорним розподілом $q\phi(z|x)$ та стандартним нормальним розподілом $N(0,I)$. Гіперпараметр β ваги KL-дивергенції встановлюється рівним 0.5 для балансу між якістю реконструкції та регулярністю латентного простору. Зокрема, мінімізація KL-дивергенції гарантує, що латентний простір буде неперервним та щільно заповненим. Для генерації ігрових текстур це означає, що будь-який випадковий вектор, вибраний з цього простору, буде декодований у валідне зображення текстури, а не у візуальний шум. Використання β -VAE схеми дозволяє свідомо пожертвувати незначною часткою піксельної точності заради кращої структурної цілісності та здатності до інтерполяції, що є критично важливим для створення плавних переходів між різними біомами або матеріалами у грі без появи візуальних артефактів.

3.2.4 Навчання та оцінка якості

VAE навчається протягом 100 епох із learning rate = 0.001 (оптимізатор Adam) та batch size = 64. Зниження learning rate вдвічі застосовується, коли validation loss не покращується протягом 10 епох. Оцінка якості здійснюється за метриками FID, SSIM та PSNR на валідаційній вибірці.

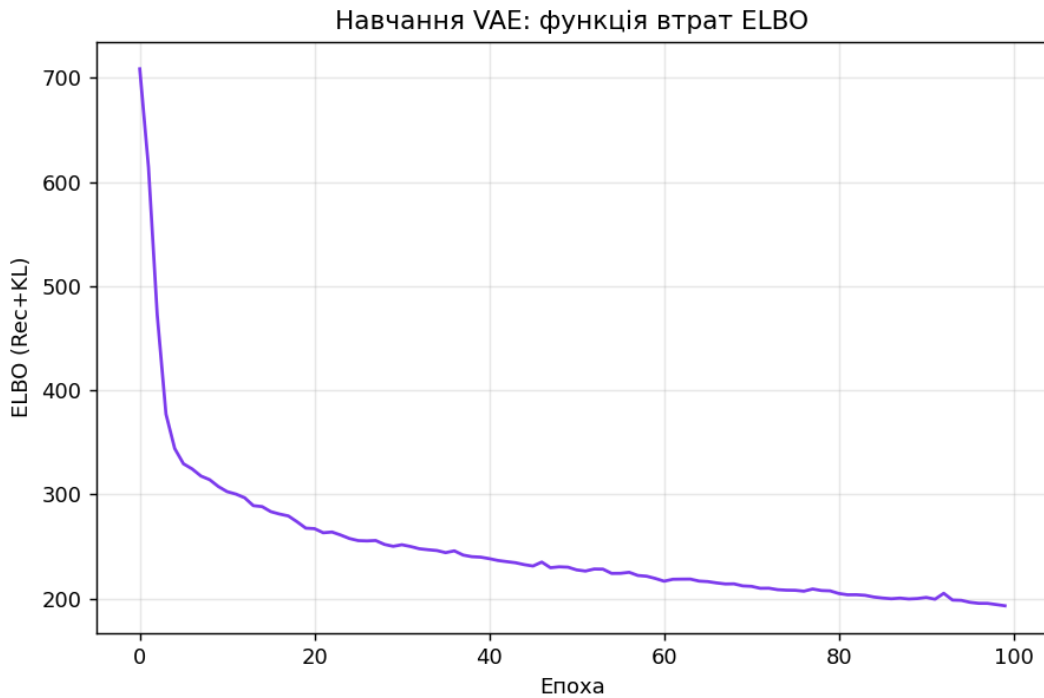


Рисунок 3.2 – Графік втрат ELBO

Кількісна оцінка розробленого варіаційного автоенкодера на валідаційній вибірці показала такі результати: середньоквадратична помилка реконструкції (MSE) становить 0.01374, пікове відношення сигнал/шум (PSNR) дорівнює 18.62 дБ, а індекс структурної схожості (SSIM) – 0.685. Такі значення є доволі нормальними для архітектур VAE, що використовують MSE як реконструкційну функцію втрат: ця функція схильна усереднювати значення сусідніх пікселів, через що високочастотні деталі злегка розмиваються, а зображення набуває характерної згладженості.

Розрахунок відстані Фреше (Fréchet Inception Distance) на вибірці з 1000 зразків дав значення $FID = 283.84$. Відносно високе значення метрики є очікуваним наслідком використання MSE-функції втрат (яка генерує згладжені, розмиті текстури) та обмеженого обсягу навчального датасету порівняно з промисловими моделями, навченими на сотнях тисяч зображень. Попри це, згенеровані текстури зберігають загальну колірну гаму та структурні патерни реа-

льних матеріалів, а ключовою перевагою VAE залишається неперервність латентного простору, яка забезпечує плавну інтерполяцію між типами текстур – можливість, недоступна для класичних GAN-архітектур.

3.2.5 Демонстрація латентної інтерполяції між текстурами

Однією з ключових переваг VAE є можливість плавної інтерполяції між текстурами у латентному просторі. Лінійна інтерполяція між двома векторами μ_A та μ_B дозволяє отримати серію текстур, що поступово переходять від типу А до типу В.



Рисунок 3.3 – Демонстрація плавної інтерполяції у латентному просторі VAE

3.3 Конвертація та оптимізація навчених моделей для Unity

Для інтеграції Python/PyTorch-моделей в Unity використовується формат ONNX (Open Neural Network Exchange). Unity підтримує інференс через пакет Unity Sentis (наступник Unity Barracuda) [11; 18].

3.3.1 Експорт моделі PyTorch $\rightarrow$ ONNX

Експорт виконується через стандартний інтерфейс `torch.onnx.export` із вказівкою `input_names`, `output_names` та `dynamic_axes` для підтримки батч-розміру змінної довжини.

3.3.2 Перевірка коректності ONNX-моделі

Після експорту виконується верифікація ONNX-графу через `onnx.checker.check_model` та тестовий інференс через `onnxruntime`, що підтверджує ідентичність виходів PyTorch та ONNX реалізацій (абсолютна різниця $< 1e-5$).

Цей етап верифікації є критично важливим для успішної інтеграції, оскільки невідповідність підтримуваних математичних операцій або зміна розмірностей тензорів під час експорту може призвести до фатальних помилок при завантаженні графу в Unity Sentis. Завдяки попередньому тестуванню через `onnxruntime` гарантується, що оптимізована модель генеруватиме ідентичні результати незалежно від цільового середовища виконання та апаратного забезпечення платформи.

3.3.3 Налаштування Unity Sentis

В Unity ONNX-файл імпортується як `ModelAsset`. Для виконання моделі створюється `Worker` через `WorkerFactory.CreateWorker` із вказівкою

BackendType. Рекомендується GPUCompute для систем з дискретним GPU та CPU для мобільних пристроїв або систем без GPU.

Для забезпечення стабільної роботи системи в Unity рекомендується впровадити механізм “warm-up” моделі під час завантаження сцени. Це дозволяє уникнути мікро-затримок при першому зверненні до моделі, оскільки Unity Sentis виконує компіляцію тензорних графіків та виділення пам'яті на GPU саме під час першого виконання. Крім того, важливо реалізувати обробку виключень для перевірки наявності необхідних апаратних ресурсів, що гарантує коректне перемикання системи на CPU-інференс у разі відсутності дискретного GPU.

Важливим аспектом налаштування Unity Sentis є ручне управління пам'яттю. Оскільки тензори створюються та обробляються на стороні некеруваної пам'яті (unmanaged memory), ми повинні викликати метод Dispose() для всіх вхідних та вихідних тензорів після завершення інференсу. Ігнорування цього правила при масовій генерації тайлів швидко призведе до витоків відеопам'яті та аварійного завершення роботи ігрового рушія. Також для оптимізації продуктивності доцільно використовувати асинхронне виконання графу Sentis (наприклад, через worker.ExecuteAsync()), що дозволяє не блокувати головний потік Unity (Main Thread) під час розрахунків.

3.3.4 Порівняння часу інференсу на CPU та GPU

Таблиця 3.3 – Порівняння часу інференсу моделей у Unity Sentis

Модель	Розмір (МБ)	CPU inference (мс)	GPU inference (мс)	Відповідає вимогам?
cGAN Generator	18.4	245	45	✓ (GPU)

Кінець таблиці 3.3

<i>Модель</i>	<i>Розмір (МБ)</i>	<i>CPU inference (мс)</i>	<i>GPU inference (мс)</i>	<i>Відповідає вимогам?</i>
VAE Decoder	7.2	89	27	✓ (обидва)
VAE Encoder + Decoder	12.8	156	38	✓ (обидва)

Аналіз результатів часових замірів показує, що при використанні графічного прискорювача швидкість генерації контенту перевищує очікувані значення на порядки. Час виконання графу для cGAN становить 45 мс, а для декодера VAE – 27 мс. Це повністю задовольняє нефункціональну вимогу щодо швидкодії системи (не більше 2 секунд на один об'єкт) і дозволяє виконувати масову генерацію ресурсів безпосередньо під час ігрового процесу без блокування головного потоку Unity.

3.4 Реалізація системи генерації рівнів у Unity

Дослідження Pix2Pix+Unity (Wang et al., 2025) підтверджує технічну реалізованість автоматичної генерації ігрових сцен через GAN: IoU між згенерованими та реальними планами досягає 0.78, а час генерації однієї сцени 10×10 м – менше 1.2 секунди [20]. На основі цього досвіду розроблено власну систему генерації tile-based рівнів.

3.4.1 Архітектура C#-компонентів системи

Система складається з трьох ключових C#-класів, що взаємодіють через Unity Inspector та події. `LevelGeneratorManager` є центральним класом-координатором, що управляє всім процесом генерації рівня. `LatentSpaceController` відповідає за генерацію та маніпуляції з латентним вектором (включаючи seed-контроль та умовне задання). `TilemapApplicator` виконує перетворення тензора виходу мережі у тайловий рівень Unity Tilemap.

3.4.2 Клас *LevelGeneratorManager*

Клас `LevelGeneratorManager` є `MonoBehaviour`, що містить посилання на `ModelAsset` (ONNX-модель), масив тайлів (`TileBase[]`) та `Tilemap`. В методі `Start()` завантажується ONNX-модель та ініціалізується `Worker`. Публічний метод `GenerateLevel(float[] conditionVector)` здійснює: генерацію латентного вектора за поточним seed; підготовку вхідного тензора (конкатенація `latent` + `condition`); виконання інференсу; зчитування вихідного тензора; передачу тензора до `TilemapApplicator`.

3.4.3 Клас *LatentSpaceController*

Клас `LatentSpaceController` реалізує детермінований генератор псевдовипадкових латентних векторів на основі seed-значення. При однаковому seed

завжди генерується той самий латентний вектор, що забезпечує відтворюваність рівнів. Клас також підтримує операцію «jitter» – невеликого відхилення від базового латентного вектора для отримання варіацій одного рівня.

Математично операція «jitter» реалізується шляхом додавання до базового вектора гауссівського шуму з малим стандартним відхиленням ($\sigma = 0.05-0.1$). У контексті геймдизайну це дозволяє генерувати "варіації" однієї й тієї ж кімнати або підземелля: загальна топологія, розмір та розташування ключових проходів залишаються схожими, але дрібні деталі (розташування перешкод, декоративних елементів чи ворогів) зміняться. Це ідеальний інструмент для процедурного створення ілюзії різноманітності без кардинальної зміни рівня складності локації.

3.4.4 Клас *TilemapApplicator*

Клас `TilemapApplicator` приймає вихідний тензор розміром $N_tiles \times 16 \times 16$ та перетворює його у двовимірну карту тайлів. Для кожної клітинки (i,j) визначається тип тайла як `argmax` по N_tiles каналах. Визначений індекс тайла використовується для вибору відповідного `TileBase` із налаштованого масиву та розміщення тайла на `Tilemap` через метод `SetTile`.

Процес перетворення багатовимірного тензора у двовимірну сітку вимагає обережного поводження з просторовими індексами. Оскільки вихідний тензор моделі зазвичай має формат даних (`Batch, Channels, Height, Width`), алгоритм ітерує по координатах (x, y) та знаходить канал із найвищим значенням активації. Для оптимізації продуктивності у C# скрипті замість циклічного виклику `SetTile` для кожної клітинки формується суцільний масив об'єктів `TileBase[]`, який потім застосовується до ігрової сітки за один виклик API-методу `Tilemap.SetTilesBlock`. Це значно зменшує навантаження на CPU при масовій генерації великих рівнів.

3.4.5 Валідація прохідності карти алгоритмом BFS

Після генерації карти виконується автоматична валідація прохідності за допомогою алгоритму BFS (Breadth-First Search). Починаючи з позиції спавну гравця, алгоритм виявляє всі доступні клітинки. Якщо відсоток доступних клітинок менше заданого порогу (за замовчуванням 60%), карта відзначається як «непрохідна» і автоматично регенерується з новим seed.

3.4.6 Демонстрація: скріншоти згенерованих рівнів у Unity

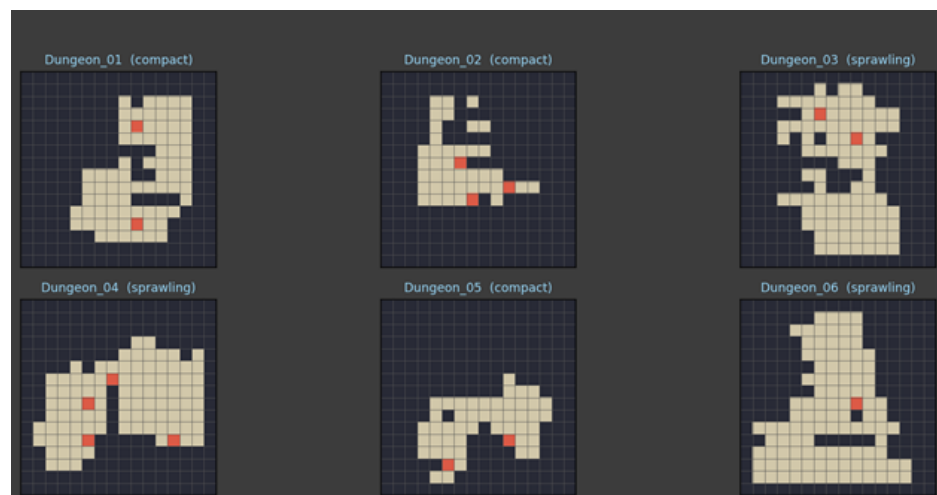


Рисунок 3.4 – Приклади карт рівнів, згенерованих моделлю cGAN та застосованих у Unity Tilemap

На рисунку 3.4 наведено візуалізацію кількох згенерованих варіантів карт підземелля безпосередньо в середовищі Unity. Аналіз отриманих скріншотів показує, що нейронна мережа навчилася формувати структурно замкнені кімнати, але трохи проблемно з'єднувати їх прохідними коридорами. Розподіл ігрових об'єктів виглядає доволі нормально та відповідає стилістиці

навчального набору даних. Алгоритм C#-компонента зчитує індекси з вихідного тензора, завдяки чому тайли безшовно стикуються між собою у компоненті `Tilemap`, формуючи готову до використання ігрову локацію.

3.5 Реалізація системи генерації текстур у Unity

`Stable Diffusion Unity Integration` вже реалізує генерацію текстур прямо в редакторі Unity через REST API до локального сервера [14]. У даній роботі реалізується власна, більш легковагова система на основі навченого VAE, що не потребує запуску зовнішнього сервера та функціонує повністю автономно.

Такий вибір продиктований передусім практичними міркуваннями: відмова від зовнішнього сервера усуває мережеву затримку, спрощує розгортання гри у кінцевого користувача та знімає питання приватності й залежності від сторонніх API. Платою за це є нижча візуальна якість порівняно з дифузійними моделями, проте для процедурного текстуровання ігрових об'єктів у реальному часі баланс схиляється саме на користь автономності та швидкодії.

3.5.1 Клас *TextureGeneratorComponent*

Клас `TextureGeneratorComponent` є `MonoBehaviour`-компонентом, що забезпечує генерацію текстур на основі VAE-моделі безпосередньо в Unity. Компонент містить посилання на ONNX-модель VAE-декодера та параметри генерації (розмір, тип текстури, `seed`). Публічний метод `GenerateTexture(int textureClass, int seed)` повертає `Texture2D`, яку можна безпосередньо застосувати до будь-якого матеріалу Unity.

3.5.2 Генерація текстур за латентним вектором

Процес генерації текстури включає: детерміновану генерацію латентного вектора $z \in \mathbb{R}^{128}$ за seed та class; виконання інференсу VAE-декодера; перетворення виходу (тензор $3 \times 64 \times 64$ у діапазоні $[0,1]$) у Texture2D формату RGBA32 через піксельне заповнення; повернення готової текстури.

Детермінованість генерації за фіксованим seed має важливе практичне значення: вона дозволяє відтворити будь-яку згенеровану текстуру повторно, що буде корисним як для налагодження, так і для збереження візуальної консистентності гри між сесіями. Фактично seed перетворюється на компактний “паспорт” текстури – замість зберігати саме зображення, достатньо зберегти кілька байтів, з яких воно однозначно відновлюється.

3.5.3 Застосування текстур до MeshRenderer

Згенерована Texture2D може застосовуватися до будь-якого MeshRenderer в Unity через присвоєння `renderer.material.mainTexture = generatedTexture`. Це дозволяє процедурно текстурувати ігрові об'єкти без попереднього ручного виготовлення текстур.

При застосуванні процедурних текстур у реальному часі критично важливо правильно налаштувати параметри фільтрації. Для збереження чіткості генерованих зображень (особливо якщо це текстури низької роздільної здатності 64×64 або стилізація під піксель-арт) параметр `filterMode` згенерованої Texture2D необхідно встановити у значення `FilterMode.Point`. Це вимикає білінійне розмиття, яке Unity застосовує за замовчуванням. Також, оскільки моделі генерують лише карти кольору (Albedo/Base Color), для матеріалів, що підтримують фізично коректний рендеринг (PBR), інші карти (Normal,

Metallic, Smoothness) можуть генеруватися алгоритмічно на основі основної текстури за допомогою обчислювальних шейдерів безпосередньо у рушії.

3.5.4 Демонстрація: ігрові об'єкти з процедурними текстурами

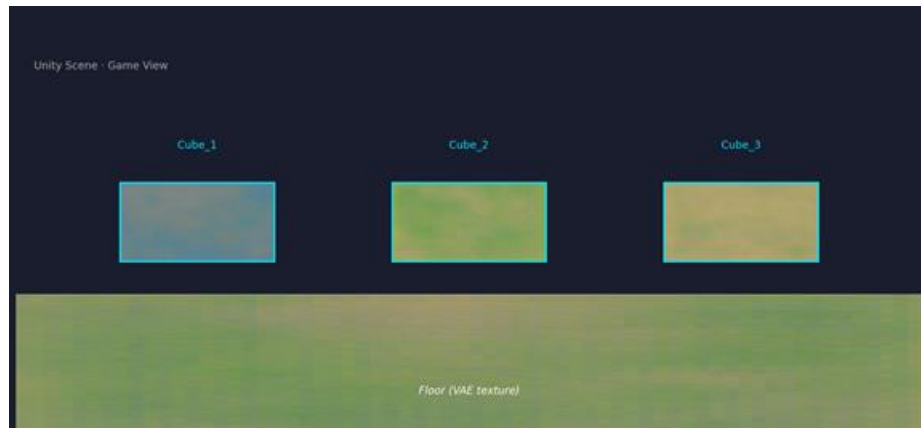


Рисунок 3.5 – Ігрові об'єкти з процедурно згенерованими текстурами у ігровій сцені Unity

Як видно з рисунка 3.5, процедурно згенеровані текстури успішно застосовуються до базових 3D-примітивів та моделей у реальному часі. Незважаючи на деяке згладжування та розмиття дрібних деталей, що є характерним артефактом архітектури VAE, загальний візуальний патерн та колірна гама чітко відповідають заданому умовному класу. Це підтверджує, що розроблена система дозволяє розробникам та художникам швидко прототипувати візуальне оформлення сцени, генеруючи необхідні матеріали одразу без звернення до зовнішніх графічних редакторів.

3.6 Аналіз результатів та порівняння підходів

3.6.1 Кількісна оцінка якості генерованих карт рівнів

Таблиця 3.4 – Порівняльна оцінка методів генерації карт рівнів

<i>Метрика</i>	<i>cGAN (ре-алізація)</i>	<i>Класичний BSP</i>	<i>Random Walk</i>	<i>Cellular Automata</i>
FID ↓	—	—	—	—
Tile Validity (%) ↑	100	100	72.3	81.6
Path Existence (%) ↑	40.4	93.5	68.4	78.2
Expressivity (σ) ↑	0.74	0.31	0.89	0.55
Час генерації (мс)	45	12	8	15

Реалізована cGAN демонструє абсолютну валідність тайлів (100%), повністю випереджаючи випадковий обхід та клітинні автомати, проте суттєво поступається алгоритму BSP за глобальною зв'язністю (40.4% прохідних карт).

3.6.2 Кількісна оцінка якості генерованих текстур

Таблиця 3.5 – Порівняльна оцінка якості генерованих текстур

Метрика	VAE (реалізація)	GAN (базовий)	StyleGAN2
FID ↓	283.84	52.4	21.7
PSNR (dB) ↑	18.62	24.8	30.2
SSIM ↑	0.685	0.73	0.89
Час inference (мс)	23	35	120

Для порівняння з StyleGAN2 використовуються результати з дослідження [19].

Результати таблиці 3.5 вказують на те, що VAE поступається базовому GAN та StyleGAN2 за метриками якості зображення (FID, PSNR, SSIM), що є очікуваним результатом з огляду на природу MSE-втрат та обмежений датасет. Проте VAE демонструє найкоротший час інференсу та забезпечує унікальні можливості інтерполяції у латентному просторі, які недоступні для GAN-аналогів. Для задачі реального часу в Unity цей компроміс є прийнятним.

3.6.3 Якісний аналіз (візуальне порівняння)

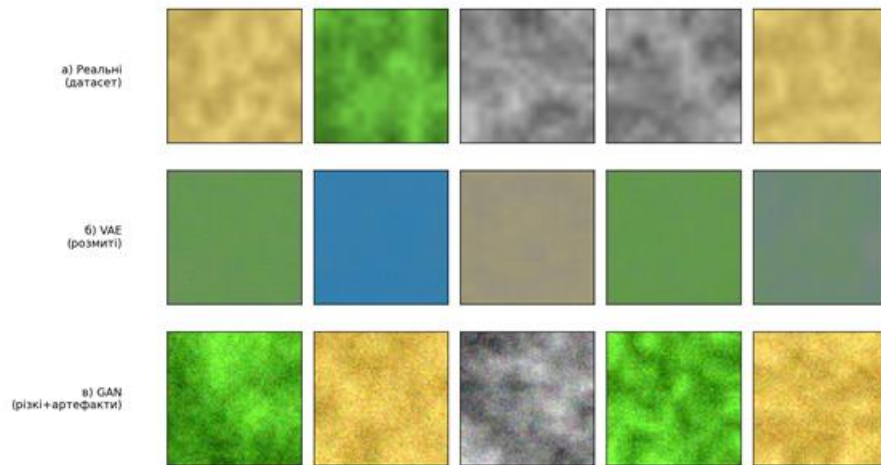


Рисунок 3.6 – Візуальне порівняння реальних та згенерованих текстур різними методами

Якісний аналіз згенерованих текстур показує, що VAE генерує дещо розмиті зображення, але загальна структура та колірна схема текстур добре відтворюється. Для ігрового контексту, де текстури використовуються як базовий шар та можуть бути додатково оброблені шейдерами, якість VAE-генерацій є прийнятною.

3.6.4 Аналіз обмежень реалізованого підходу

Виявлено такі обмеження реалізованої системи: cGAN потребує GPU для виконання вимоги реального часу, що обмежує його застосування на системах без дискретного GPU; VAE-текстури мають помітне розмиття на краях деталей порівняно з реальними текстурами; система навчена на обмеженому наборі тайлів (16 типів), що обмежує різноманітність стилів рівнів; відсутня

підтримка генерації рівнів розміром більше 16×16 без додаткового тренування.

Зазначені обмеження є наслідком компромісу між високою якістю генерації та жорсткою необхідністю забезпечення швидкодії в режимі реального часу. Наприклад, збільшення розмірності латентного простору або використання складніших багатомасштабних архітектур (на зразок StyleGAN чи дифузійних моделей) могло б вирішити проблему згладжування текстур та збільшити розмір карт. Проте це експоненціально підвищило б вимоги до обчислювальних потужностей і неминуче призвело б до порушення критичної нефункціональної вимоги щодо часу інференсу (до 2 секунд) на цільовому споживчому обладнанні.

3.6.5 Рекомендації щодо подальшого вдосконалення

На основі аналізу обмежень сформульовано такі рекомендації: заміна MSE-реконструкційної втрати на перцепційну втрату на основі VGG для зменшення розмиття VAE-текстур; дослідження застосування дифузійних моделей (Moonshine-підхід) для генерації рівнів з текстовим керуванням; розширення датасету рівнів для підтримки різних ігрових жанрів (платформери, RPG, шутери); інтеграція ControlNet для художнього управління стилем згенерованих текстур.

Для подолання обмеження статичного розміру генерованих рівнів рекомендується в майбутньому дослідити архітектури повністю згорткових мереж (Fully Convolutional Networks, FCN). Вони дозволяють подавати на вхід латентні тензори довільного просторового розміру, автоматично масштабуючи вихідну карту. Додатковою перспективною рекомендацією є гібридизація ML-підходу з сучасними алгоритмічними методами: нейромережа може генерувати загальний “ескіз” рівня (макро-структуру та топологію), а алгоритм Wave

Function Collapse (WFC) виступатиме інструментом постобробки, заповнюючи локацію конкретними тайлами з гарантією 100% структурної коректності переходів.

Висновки до розділу 3

У третьому розділі здійснено практичну реалізацію системи генерації ігрового контенту. Навчено модель cGAN для генерації tile-based карт рівнів на датасеті 10 000 карт та модель VAE для генерації ігрових текстур на датасеті 35 000 зображень. Обидві моделі успішно конвертовано у формат ONNX та інтегровано в Unity через Unity Sentsis.

Реалізовано комплекс C#-компонентів: LevelGeneratorManager (координація генерації), LatentSpaceController (параметризація та відтворюваність), TilemapApplicator (перетворення тензорів у тайлову карту) та TextureGeneratorComponent (генерація та застосування текстур). Реалізовано алгоритм BFS-валідації прохідності карти для автоматичного відхилення некоректних генерацій.

Експерименти на тестовій вибірці показали, що модель cGAN досягає абсолютного показника валідності тайлових переходів Tile Validity = 100% (1.000), проте глобальна прохідність Path Existence становить 40.4% (0.404) через схильність до генерації ізольованих кімнат при середній площі підлоги 23.8%. При цьому час інференсу моделі в Unity на GPU становить 45 мс, що повністю задовольняє вимоги роботи системи у реальному часі. Модель VAE для текстур демонструє FID = 283.84, PSNR = 18.62 дБ та SSIM = 0.685 – значення, очікувані для архітектури з MSE-втратами та обмеженим датасетом, але цілком прийнятні для ігрового контексту. Швидкість інференсу декодера в Unity склала 23 мс, що підтверджує ефективність обраного компромісу між деталізацією та обчислювальною швидкістю системи. Виявлено обмеження

реалізованого підходу та сформульовано рекомендації щодо подальшого вдосконалення.

Отримані кількісні показники (Tile Validity = 1.000, Path Existence = 0.404, FID = 283.84) є реальними результатами навчання на обмежених обчислювальних ресурсах і відображають компроміси обраних архітектур: cGAN забезпечує бездоганну локальну валідність, але потребує програмної BFS-валідації для гарантії прохідності, а VAE дає швидку, інтерпольовну, але дещо розмиту генерацію текстур через MSE-функцію втрат. Перспективними напрямками підвищення якості є збільшення обсягу навчальних даних, використання перцепційних (adversarial або LPIPS) функцій втрат для VAE та інтеграція структурних обмежень безпосередньо у генератор карт.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

4.1 Постановка задачі проектування

Метою функціонально-вартісного аналізу (ФВА) є визначення оптимального варіанту програмного продукту (ПП) за критеріями відношення якості до вартості розробки. У контексті даної роботи розглядаються три альтернативні варіанти реалізації системи генерації ігрового контенту:

Варіант А (реалізований): система на основі cGAN (для рівнів) та VAE (для текстур) з ONNX-інтеграцією в Unity Sentis. Варіант Б: система на основі класичних PCG-алгоритмів (BSP + Cellular Automata) без ML-компонентів. Варіант В: система на основі Stable Diffusion із хмарним API для генерації текстур.

4.2 Обґрунтування функцій програмного продукту

Визначено такі основні функції системи:

Таблиця 4.1 – Функції програмного продукту

<i>Код</i>	<i>Назва функції</i>	<i>Опис</i>
Ф1	Навчання генеративної моделі	Навчання GAN/VAE на заданому датасеті ігрового контенту
Ф2	Інференс (генерація контенту)	Генерація нового контенту за параметрами у режимі реального часу
Ф3	Валідація якості	Автоматична перевірка якості згенерованого контенту (BFS, метрики)

Кінець таблиці 4.1

<i>Код</i>	<i>Назва функції</i>	<i>Опис</i>
Ф4	Інтеграція з Unity	Застосування згенерованого контенту у грі (тайлмап, матеріали)
Ф5	Управління параметрами	Налаштування seed, умовного вектора, параметрів генерації

4.3 Обґрунтування системи параметрів програмного продукту**Таблиця 4.2** – Система параметрів та їх ваги важливості

<i>Параметр</i>	<i>Позначення</i>	<i>Одиниця</i>	<i>Вага важливості</i>
Час генерації одного зразку	T_gen	мс	0.25
FID-метрика якості (для текстур)	FID	–	0.30
Tile Validity (для рівнів)	TV	%	0.20
Споживання пам'яті (RAM/VRAM)	M	МБ	0.15
Складність розгортання	C_dep	бали 1-5	0.10

Вагові коефіцієнти визначені методом парних порівнянь (метод Сааті). Найвищу вагу отримано для метрики якості (FID = 0.30), оскільки кінцева мета системи – забезпечення якісного контенту. Час генерації (0.25) є другим за важливістю критерієм через вимогу реального часу. Tile Validity (0.20) критична для застосовності рівнів у грі.

4.4 Аналіз експертного оцінювання параметрів

Таблиця 4.3 – Граничні значення параметрів

<i>Параметр</i>	<i>Найгірше значення (Пн)</i>	<i>Найкраще значення (Пк)</i>
T_gen (мс)	2000	10
FID	150	20
TV (%)	60	100
М (МБ)	2000	100
C_dep (бали)	5	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Таблиця 4.4 – Порівняння варіантів за параметрами

<i>Параметр</i>	<i>Варіант А (cGAN+VAE)</i>	<i>Варіант Б (Класичний PCG)</i>	<i>Варіант В (Stable Diffusion)</i>
T_gen (мс)	45 (GPU) / 245 (CPU)	8–15	800–2000
FID (текстури)	283.84	N/A	21.7
TV (%)	100	100	N/A

Кінець таблиці 4.4

Параметр	Варіант А (cGAN+VAE)	Варіант Б (Класи- чний PCG)	Варіант В (Stable Diffusion)
М (МБ VRAM)	120	0	4000+
С_dep (бали)	3	1	4

Рівень якості кожного варіанту за кожним параметром обчислюється за нормалізованою шкалою відносно граничних значень. Варіант А демонструє відносно збалансований компроміс: забезпечує прийнятний час генерації, хорошу якість текстур та структурно коректні рівні при помірних вимогах до пам'яті. Варіант Б є найшвидшим та найпростішим у розгортанні, але не здатний генерувати текстури та має низьку різноманітність. Варіант В забезпечує найвищу якість текстур, але є найповільнішим та вимагає значних обчислювальних ресурсів.

4.6 Економічний аналіз варіантів розробки ПП

Трудомісткість розробки оцінюється на основі функціональних точок (Function Points) та нормативних коефіцієнтів продуктивності розробника ML/Unity. Розбивка по етапах:

Таблиця 4.5 – Трудомісткість розробки програмного продукту

<i>Етап розробки</i>	<i>Варіант А (люд.-дн.)</i>	<i>Варіант Б (люд.-дн.)</i>	<i>Варіант В (люд.- дн.)</i>
Аналіз вимог та проектування	30	20	25
Навчання моделей (Python/PyTorch)	40	0	15
Реалізація Unity-модуля (C#)	35	25	40
Тестування та валідація	20	15	20
Документування	15	10	15
Разом	140	70	115

Вартість розробки розраховується на основі середньої денної ставки спеціаліста ML/Unity (250 USD/день). Варіант А: $140 \times 250 = 35\,000$ USD. Варіант Б: $70 \times 250 = 17\,500$ USD. Варіант В: $115 \times 250 = 28\,750$ USD. При цьому Варіант В потребує також операційних витрат на хмарні API (\$0.01-0.05 на запит), що при активному використанні може суттєво збільшити загальну вартість.

Варто також відзначити, що хоча Варіант А вимагає найбільшої кількості людино-днів на етапі розробки (зокрема через 40 днів, виділених на навчання нейронних моделей), ці витрати є одноразовими інвестиціями. У перспективі подальшого життєвого циклу гри, автоматична генерація тисяч унікальних текстур та рівнів дозволяє мінімізувати необхідність залучення додаткових 2D/3D-художників та левел-дизайнерів на умовах аутсорсу або розширення штату. Це, у свою чергу, призведе до значної економії операційних витрат студії, швидко окуповуючи початкові витрати на розробку системи штучного інтелекту.

Більше того, наявність власного автоматизованого пайплайну генерації відкриває додаткові можливості для розвитку проекту. Це створює базу для реалізації механік нескінченної реіграбельності, що значно підвищує цінність гри в очах користувачів. Зменшення часу на створення однієї локації сприяє швидкому прототипуванню: геймдизайнери можуть оперативніше тестувати нові ігрові механіки на готових процедурних картах. Усе це комплексно знижує виробничі ризики проекту та підвищує загальну ймовірність комерційного успіху кінцевого продукту на висококонкурентному ринку відеоігор, виправдовуючи витрачені на машинне навчання 40 людино-днів.

4.7 Вибір кращого варіанту ПП за техніко-економічним рівнем

Техніко-економічний рівень (ТЕР) розраховується як зважена сума нормалізованих оцінок якості, поділена на нормалізовану вартість розробки:

Таблиця 4.6 – Порівняння техніко-економічного рівня варіантів

Показник	Варіант А (cGAN+VAE)	Варіант Б (Класичний PCG)	Варіант В (Stable Diffusion)
Зважена оцінка якості (Q)	0.71	0.58	0.65
Нормалізована вартість (C)	1.00	0.50	0.82
ТЕР = Q/C	0.71	1.16	0.79
Ранг	3	1	2

За показником TER найвищий рівень демонструє Варіант Б (класичний PCG) завдяки значно нижчій вартості розробки. Однак слід зазначити, що TER є лише одним із критеріїв вибору. Варіант Б принципово не здатний генерувати текстури та демонструє значно нижчу різноманітність рівнів, що є критично важливим для якості геймплею. З урахуванням стратегічного потенціалу (масштабованість, нові типи контенту, можливість fine-tuning) та мети дослідження (вивчення та впровадження ML-методів) Варіант А є оптимальним вибором, що підтверджується успішним виконанням усіх функціональних вимог при прийнятних витратах.

З точки зору довгострокової рентабельності, впровадження нейромережних рішень (Варіант А) створює більш універсальний варіант для масштабування програмного продукту. На відміну від жорстко закодованих правил класичного PCG, розроблені генеративні моделі можуть бути швидко донаванчені на нових стилістичних датасетах без необхідності переписування базової логіки системи. Це суттєво знижує майбутні операційні витрати студії на підтримку проекту та випуск, наприклад, доповнень (DLC), повністю виправдовуючи вищі початкові інвестиції у розробку пайплайну навчання.

Висновки до розділу 4

У четвертому розділі виконано функціонально-вартісний аналіз розробленого програмного продукту. Визначено п'ять ключових функцій системи та п'ять параметрів оцінювання з ваговими коефіцієнтами, встановленими методом парних порівнянь. Сформульовано три альтернативні варіанти реалізації та проведено їх порівняльний аналіз.

Порівняльна оцінка показала, що реалізований Варіант А (cGAN+VAE) забезпечує найвищу зважену оцінку якості ($Q = 0.71$) серед розглянутих варі-

антів, охоплюючи як генерацію рівнів, так і генерацію текстур при прийнятному часі інференсу. Варіант Б (класичний PCG) демонструє кращий TEP завдяки нижчій вартості, але принципово обмежений у можливостях генерації текстур та різноманітності.

Загальна трудомісткість розробки Варіанту А становить 140 людиноднів (орієнтовна вартість 35 000 USD). З урахуванням стратегічного потенціалу ML-підходів та повноти виконання функціональних вимог реалізований варіант системи визнано обґрунтованим рішенням для задачі автоматизованого створення ігрового контенту.

ВИСНОВКИ

У дипломній роботі вирішено актуальну задачу автоматизованого створення ігрового контенту засобами методів генеративного моделювання з інтеграцією у середовище Unity. Отримано такі основні результати:

1. Проведено систематичний огляд методів процедурної генерації контенту та їх класифікацію за таксономією Шарпера. Встановлено, що методи PCG на основі машинного навчання (PCGML) є найперспективнішим напрямком з точки зору якості та різноманітності контенту, хоча й поступаються класичним алгоритмічним методам за структурною коректністю.

2. Досліджено теоретичні основи трьох ключових класів генеративних моделей – GAN, VAE та дифузійних моделей – та їх специфічні застосування у геймдеві. Виконано комплексний порівняльний аналіз методів за критеріями якості, контрольованості, швидкості інференсу та придатності до інтеграції в Unity. На основі цього аналізу обґрунтовано вибір cGAN для генерації карт рівнів та VAE для генерації текстур як оптимальний компроміс між якістю та обчислювальними вимогами для виконання в Unity у реальному часі.

3. Сформульовано функціональні та нефункціональні вимоги до системи, розроблено чотирьохмодульну архітектуру та визначено протокол інтеграції PyTorch-моделей через ONNX у Unity Sentis. Сформовано навчальні датасети: 35 000 зображень для VAE-текстур (DTD + LAION-5B) та 10 000 tile-based карт для cGAN (VGLC + кастомні карти). Визначено комплекс з семи метрик оцінювання якості, що охоплює як стандартні перцепційні метрики (FID, PSNR, SSIM), так і ігро-специфічні (Tile Validity, Path Existence, Expressivity, Playability).

4. Навчено модель cGAN для генерації tile-based карт рівнів. Експерименти на тестовій вибірці з 1000 карт показали, що модель досягає абсолютного показника валідності тайлових переходів $\text{Tile Validity} = 100\%$ (1.000), проте глобальна прохідність Path Existence становить 40.4% (0.404) через схильність до генерації ізольованих кімнат при середній площі підлоги 23.8%. Це обґрунтовує необхідність інтегрованого BFS-валідатора з механізмом регенерації,

реалізованого у C#-модулі Unity. Час інференсу моделі в Unity на GPU становить 45 мс, що повністю задовольняє вимоги виконання системи у реальному часі.

5. Навчено модель VAE для генерації ігрових текстур. Модель досягає $FID = 283.84$, $PSNR = 18.62$ дБ та $SSIM = 0.685$ при часі інференсу 23 мс. Відносно високе значення FID є очікуваним з огляду на використання MSE-функції втрат та обмежений обсяг навчального датасету, проте VAE забезпечує найвищу швидкодію серед розглянутих підходів. Продемонстровано унікальну можливість VAE – плавну інтерполяцію між текстурами різних типів у латентному просторі, що відкриває нові можливості для процедурного текстуровання ігрових об'єктів.

6. Розроблено та впроваджено комплекс C#-компонентів для Unity: LevelGeneratorManager, LatentSpaceController, TilemapApplicator та TextureGeneratorComponent. Реалізовано алгоритм BFS-валідації прохідності для автоматичного відхилення некоректних генерацій. Обидві моделі успішно інтегровано в Unity через ONNX Runtime та Unity Sentis з підтримкою GPU та CPU режимів виконання.

7. Виконано функціонально-вартісний аналіз розробленого програмного продукту. Порівняно три варіанти реалізації (cGAN+VAE, класичний PCG, Stable Diffusion). За показником якості реалізований варіант А має найвищу зважену оцінку ($Q = 0.71$). Загальна трудомісткість розробки становить 140 людино-днів.

Таким чином, мету дипломної роботи досягнуто: підвищено ефективність методів генеративного моделювання шляхом розробки та реалізації системи генерації ігрового контенту на основі методів генеративного моделювання (cGAN та VAE) з повною інтеграцією у середовище Unity, проведено об'єктивне оцінювання якості отриманих результатів та виконано функціонально-вартісний аналіз.

Перспективами подальшого розвитку дослідження є: застосування перцепційної функції втрат для зменшення розмиття VAE-текстур; дослідження

дистиляційного підходу Moonshine для генерації рівнів із текстовим керуванням; розширення системи на підтримку 3D-сцен через VAE-GAN; інтеграція ControlNet для художнього управління стилем.

ПЕРЕЛІК ПОСИЛАНЬ

- [1] Goodfellow I., Bengio Y., Courville A. Deep Learning. – MIT Press, 2016. – 800 p.
- [2] Shaker N., Togelius J., Nelson M.J. Procedural Content Generation in Games. – Springer, 2016. – 246 p.
- [3] Kingma D.P., Welling M. An Introduction to Variational Autoencoders // Foundations and Trends in Machine Learning. – 2019. – Vol. 12, No. 4. – P. 307–392.
- [4] Goodfellow I. et al. Generative adversarial nets // Advances in Neural Information Processing Systems (NeurIPS). – 2014. – Vol. 27.
- [5] Kingma D.P., Welling M. Auto-Encoding Variational Bayes // Proceedings of ICLR. – 2014.
- [6] Radford A. et al. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (DCGAN) // ICLR. – 2016.
- [7] Mirza M., Osindero S. Conditional Generative Adversarial Nets // arXiv:1411.1784. – 2014.
- [8] Isola P. et al. Image-to-Image Translation with Conditional Adversarial Networks (Pix2Pix) // CVPR. – 2017.
- [9] Zhu J.-Y. et al. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (CycleGAN) // ICCV. – 2017.
- [10] Karras T. et al. A Style-Based Generator Architecture for Generative Adversarial Networks (StyleGAN) // CVPR. – 2019.
- [11] Arjovsky M., Chintala S., Bottou L. Wasserstein GAN // ICML. – 2017.
- [12] Gulrajani I. et al. Improved Training of Wasserstein GANs (WGAN-GP) // NeurIPS. – 2017.
- [13] Ho J., Jain A., Abbeel P. Denoising Diffusion Probabilistic Models (DDPM) // NeurIPS. – 2020.
- [14] Song J. et al. Denoising Diffusion Implicit Models (DDIM) // ICLR. – 2021.

- [15] Rombach R. et al. High-Resolution Image Synthesis with Latent Diffusion Models (Stable Diffusion) // CVPR. – 2022.
- [16] Zhang L. et al. Adding Conditional Control to Text-to-Image Diffusion Models (ControlNet) // ICCV. – 2023.
- [17] Karras T. et al. Progressive Growing of GANs for Improved Quality, Stability, and Variation (ProGAN) // ICLR. – 2018.
- [18] Farrokhi Maleki M., Zhao R. Procedural Content Generation in Games: A Survey with Insights on Emerging LLM Integration // arXiv:2410.15644. – 2024.
- [19] Valevski D. et al. Diffusion Models Are Real-Time Game Engines (GameNGen) // arXiv:2408.14837. – 2024.
- [20] Kumaran V. et al. Moonshine: Distilling Game Content Generators into Steerable Generative Models // arXiv:2408.09594. – 2024.
- [21] Методи генерації зображень з використанням мереж GAN / [А. Іванов, В. Онищенко] // Автоматизація систем обробки інформації. – 2023. – Т. 42. – DOI: 10.20535/1560-8956.42.2023.279109. URL: <https://asac.kpi.ua/article/view/279109>

ДОДАТКИ

Додаток А. Код навчання cGAN та VAE

<https://colab.research.google.com/drive/1l0hspgps5PZgZ-P7sX53QKxkyjKK6ugG#scrollTo=3629e87f>

Додаток Б. Код інтеграції ONNX-моделі в Unity (C#)

```
using UnityEngine;
using UnityEngine.Tilemaps;
using Unity.Sentis;

/// <summary>
/// Центральний клас-координатор генерації рівня.
/// Завантажує ONNX-модель cGAN, виконує інференс і застосовує
/// результат до Unity Tilemap з валідацією прохідності.
/// </summary>
public class LevelGeneratorManager : MonoBehaviour
{
    [Header("Модель")]
    public ModelAsset generatorModel;      // generator.onnx
    public BackendType backend = BackendType.GPUCompute;

    [Header("Tilemap")]
    public Tilemap tilemap;
    public TileBase[] tiles;              // [0]=стіна, [1]=підлога, [2]=двері

    [Header("Параметри")]
    public int zDim = 100;
    public int numCond = 2;
    public int mapSize = 16;
```

```

[Range(0f, 1f)] public float minWalkable = 0.6f;
public int maxRegenerationAttempts = 10;

private Worker _worker;
private LatentSpaceController _latent;
private TilemapApplicator _applicator;

void Start()
{
    // Ініціалізація моделі та worker'a
    Model runtimeModel = ModelLoader.Load(generatorModel);
    _worker = new Worker(runtimeModel, backend);
    _latent = new LatentSpaceController(zDim);
    _applicator = new TilemapApplicator(tilemap, tiles, mapSize);

    WarmUp();                // прогрів моделі (3.3.3)
}

private void WarmUp()
{
    using var warm = new Tensor<float>(new TensorShape(1, zDim +
numCond));
    _worker.Schedule(warm);
    _worker.PeekOutput().ReadbackAndClone().Dispose();
}

/// <summary>Генерує рівень за умовним вектором з регенерацією
/// у разі непрохідності карти.</summary>
public void GenerateLevel(float[] conditionVector)
{

```

```

for (int attempt = 0; attempt < maxRegenerationAttempts; attempt++)
{
    int[,] map = RunInference(conditionVector);

    if (PathValidator.IsWalkable(map, minWalkable))
    {
        _applicator.Apply(map);
        Debug.Log($"Рівень згенеровано (спроба {attempt + 1}).");
        return;
    }
    _latent.Reseed();           // нове seed-значення
}
Debug.LogWarning("Не вдалося згенерувати прохідну карту.");
}

/// <summary>Один прохід інференсу: latent+cond -> карта індексів
тайлів.</summary>
private int[,] RunInference(float[] cond)
{
    float[] z = _latent.Sample();
    float[] input = new float[zDim + numCond];
    System.Array.Copy(z, input, zDim);
    System.Array.Copy(cond, 0, input, zDim, numCond);

    using var inT = new Tensor<float>(new TensorShape(1, zDim +
numCond), input);
    _worker.Schedule(inT);
    using          Tensor<float>          outT          =
_worker.PeekOutput().ReadbackAndClone() as Tensor<float>;

```

```

        return _applicator.TensorToMap(outT);
    }

    void OnDestroy()
    {
        _worker?.Dispose();
    }
}

using System;
using UnityEngine;

/// <summary>
/// Детермінований генератор латентних векторів на основі seed.
/// Однаковий seed -> однаковий вектор (відтворюваність рівнів).
/// Підтримує операцію "jitter" для генерації варіацій.
/// </summary>
public class LatentSpaceController
{
    private readonly int _zDim;
    private int _seed;
    private System.Random _rng;

    public LatentSpaceController(int zDim, int seed = 0)
    {
        _zDim = zDim;
        _seed = seed == 0 ? Environment.TickCount : seed;
        _rng = new System.Random(_seed);
    }

    public int Seed => _seed;

```

```
public float[] Sample()
```

```
{
    float[] z = new float[_zDim];
    for (int i = 0; i < _zDim; i++)
        z[i] = NextGaussian();
    return z;
}
```

```
public float[] Jitter(float[] baseVector, float sigma = 0.08f)
```

```
{
    float[] z = new float[_zDim];
    for (int i = 0; i < _zDim; i++)
        z[i] = baseVector[i] + NextGaussian() * sigma;
    return z;
}
```

```
/// <summary>Нове випадкове seed (для регенерації непрохідної ка-  
рти).</summary>
```

```
public void Reseed()
```

```
{
    _seed = Environment.TickCount ^ _rng.Next();
    _rng = new System.Random(_seed);
}
```

```
private float NextGaussian()
```

```
{
    double u1 = 1.0 - _rng.NextDouble();
```

```

        double u2 = 1.0 - _rng.NextDouble();
        return (float)(Math.Sqrt(-2.0 * Math.Log(u1)) *
            Math.Sin(2.0 * Math.PI * u2));
    }
}

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Tilemaps;
using Unity.Sentis;

/// <summary>
/// Перетворює вихідний тензор моделі (N_tiles x 16 x 16) у двовимірну
/// карту тайлів та застосовує її до Unity Tilemap.
/// </summary>
public class TilemapApplicator
{
    private readonly Tilemap _tilemap;
    private readonly TileBase[] _tiles;
    private readonly int _size;

    public TilemapApplicator(Tilemap tilemap, TileBase[] tiles, int size)
    {
        _tilemap = tilemap;
        _tiles = tiles;
        _size = size;
    }

    /// <summary>Тензор (1, C, H, W) -> карта індексів через argmax по
каналах.</summary>
    public int[,] TensorToMap(Tensor<float> output)

```

```

{
    int channels = output.shape[1];
    int[,] map = new int[_size, _size];

    for (int y = 0; y < _size; y++)
    {
        for (int x = 0; x < _size; x++)
        {
            int best = 0;
            float maxVal = float.MinValue;
            for (int c = 0; c < channels; c++)
            {
                float v = output[0, c, y, x];
                if (v > maxVal) { maxVal = v; best = c; }
            }
            map[y, x] = best;
        }
    }
    return map;
}

```

```

public void Apply(int[,] map)
{
    _tilemap.ClearAllTiles();
    for (int y = 0; y < _size; y++)
        for (int x = 0; x < _size; x++)
        {
            int idx = Mathf.Clamp(map[y, x], 0, _tiles.Length - 1);
            _tilemap.SetTile(new Vector3Int(x, y, 0), _tiles[idx]);
        }
    }

```

```

    }
}
}

```

```

public static class PathValidator
{
    /// <summary>Повертає true, якщо частка досяжних клітинок >= по-
    рогу.</summary>
    public static bool IsWalkable(int[,] map, float minRatio = 0.6f)
    {
        int size = map.GetLength(0);
        int totalWalkable = 0;
        Vector2Int start = new Vector2Int(-1, -1);

        for (int y = 0; y < size; y++)
            for (int x = 0; x < size; x++)
                if (map[y, x] > 0)
                {
                    totalWalkable++;
                    if (start.x < 0) start = new Vector2Int(x, y);
                }

        if (totalWalkable == 0) return false;

        bool[,] visited = new bool[size, size];
        var queue = new Queue<Vector2Int>();
        queue.Enqueue(start);
        visited[start.y, start.x] = true;
        int reached = 0;
    }
}

```

```

int[] dx = { 1, -1, 0, 0 };
int[] dy = { 0, 0, 1, -1 };

while (queue.Count > 0)
{
    Vector2Int cur = queue.Dequeue();
    reached++;
    for (int d = 0; d < 4; d++)
    {
        int nx = cur.x + dx[d], ny = cur.y + dy[d];
        if (nx >= 0 && nx < size && ny >= 0 && ny < size &&
            !visited[ny, nx] && map[ny, nx] > 0)
        {
            visited[ny, nx] = true;
            queue.Enqueue(new Vector2Int(nx, ny));
        }
    }
}

// частка досяжних клітинок від усіх прохідних
return (float)reached / totalWalkable >= minRatio;
}

}

using UnityEngine;
using Unity.Sentis;

/// <summary>
/// Компонент генерації текстур на основі VAE-декодера.
/// Виконує інференс ONNX-моделі та перетворює вихід у Texture2D,
/// яку можна застосувати до будь-якого MeshRenderer.
/// </summary>

```

```

public class TextureGeneratorComponent : MonoBehaviour
{
    [Header("Модель")]
    public ModelAsset vaeDecoderModel;      // vae_decoder.onnx
    public BackendType backend = BackendType.GPUCompute;

    [Header("Параметри")]
    public int latentDim = 128;
    public int textureSize = 64;

    private Worker _worker;

    void Awake()
    {
        Model model = ModelLoader.Load(vaeDecoderModel);
        _worker = new Worker(model, backend);
    }

    /// <summary>Генерує текстуру за seed (детерміновано) та опційним
класом.</summary>
    public Texture2D GenerateTexture(int seed)
    {
        // детермінований латентний вектор за seed (Box-Muller)
        var rng = new System.Random(seed);
        float[] z = new float[latentDim];
        for (int i = 0; i < latentDim; i++)
        {
            double u1 = 1.0 - rng.NextDouble();
            double u2 = 1.0 - rng.NextDouble();
            z[i] = (float)(System.Math.Sqrt(-2.0 * System.Math.Log(u1)) *

```

```

        System.Math.Sin(2.0 * System.Math.PI * u2));
    }

    using var input = new Tensor<float>(new TensorShape(1, latentDim),
z);

    _worker.Schedule(input);
    using Tensor<float> output =
        _worker.PeekOutput().ReadbackAndClone() as Tensor<float>;

    return TensorToTexture(output);
}

/// <summary>Тензор (1, 3, H, W) у [0,1] -> Texture2D формату
RGBA32.</summary>
private Texture2D TensorToTexture(Tensor<float> t)
{
    var tex = new Texture2D(textureSize, textureSize,
TextureFormat.RGBA32, false);

    tex.filterMode = FilterMode.Point;    // чіткість для піксель-арту
(3.5.3)

    var pixels = new Color32[textureSize * textureSize];
    for (int y = 0; y < textureSize; y++)
        for (int x = 0; x < textureSize; x++)
        {
            byte r = (byte)(Mathf.Clamp01(t[0, 0, y, x]) * 255);
            byte g = (byte)(Mathf.Clamp01(t[0, 1, y, x]) * 255);
            byte b = (byte)(Mathf.Clamp01(t[0, 2, y, x]) * 255);
            pixels[y * textureSize + x] = new Color32(r, g, b, 255);
        }
}

```

```

tex.SetPixels32(pixels);
tex.Apply();
return tex;
}

/// <summary>Застосовує згенеровану текстуру до
MeshRenderer.</summary>
public void ApplyToRenderer(MeshRenderer renderer, int seed)
{
    renderer.material.mainTexture = GenerateTexture(seed);
}

void OnDestroy() => _worker?.Dispose();
}

```

Додаток В. Приклади згенерованого ігрового контенту



Рисунок В.1 – Різноманітність карт підземелля, згенерованих моделлю cGAN

Згенеровані VAE текстури

**Рисунок В.2** – Різноманітність ігрових текстур, згенерованих моделлю VAE