Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему: «Розробка алгоритму процедурної генерації тривимірних ландшафтів у середовищі Unity»

Виконав: студент групи К23-1М

Спеціальність 122 Комп'ютерні науки

Погорілий Д.С.
(прізвище та ініціали)

Керівник к.т.н., доц. Ульяновська Ю. В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Дніпровський державний технічний університет
(місце роботи)

доцент кафедри математичного моделювання та системного аналізу
(посада)

к.т.н., доц. Волосова Н.М.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Погорілий Д.С. Розробка алгоритму процедурної генерації тривимірних ландшафтів у середовищі Unity.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 122 «Комп’ютерні науки». – Університет митної справи та фінансів, Дніпро, 2025.

Об’єктом дослідження є процес процедурної генерації тривимірних ландшафтів.

Предметом дослідження є методи 3D-рендерингу та процедурної генерації текстур із застосуванням алгоритму Marching Cubes.

Мета роботи – розробка та програмна реалізація алгоритму Marching Cubes для процедурної генерації тривимірних ландшафтів у середовищі Unity.

Робота включає аналіз методів процедурної генерації ландшафтів, таких як карти висот, іррегулярні сітки, тайлові системи, алгоритми Marching Cubes та Marching Tetrahedra. Обґрунтовано вибір алгоритму Marching Cubes завдяки його ефективності та високій деталізації. Проведено огляд програмних засобів для реалізації, серед яких Autodesk Maya, Three.js та Unity, з подальшим вибором Unity як найбільш придатного середовища для реалізації алгоритму. У ході роботи було розроблено алгоритм процедурної генерації, протестовано його ефективність та відповідність вимогам до реалістичності та продуктивності в інтерактивних 3D-сценах.

Ключові слова: процедурна генерація, Marching Cubes, Unity, 3D-ландшафти, комп'ютерна графіка, воксельна графіка.

ABSTRACT

Pohorilyi D. Development of an Algorithm for Procedural Generation of 3D Landscapes in Unity

This project for obtaining a master's degree in speciality 122 "Computer Science." - University of Customs and Finance, Dnipro, 2025.

The object of the study is the process of procedural generation of three-dimensional landscapes.

The subject of the study is the methods of procedural generation of landscapes using the Marching Cubes algorithm in the Unity environment.

The purpose of the work is to develop an algorithm for the procedural generation of landscapes to create realistic three-dimensional scenes in the Unity environment.

The work includes the study of procedural generation methods, including height maps, irregular grids, tile-based systems, as well as the Marching Cubes and Marching Tetrahedra algorithms. The choice of Unity as the primary development platform is justified by its extensive support for voxel data processing and C# scripting. During the research, the Marching Cubes algorithm was implemented, enabling the automated creation of large and realistic landscapes for video games and simulation environments. The algorithm was tested for performance, detail, and rendering stability.

Keywords: procedural generation, Marching Cubes, Unity, 3D landscapes, voxel graphics, computer graphics.

ЗМІСТ

ВСТУП

Розвиток сучасних комп'ютерних технологій суттєво вплинув на створення тривимірних графічних моделей, зокрема процедурної генерації ландшафтів, яка широко використовується у відеоіграх, анімації, симуляційних середовищах та архітектурній візуалізації. Процедурні методи створення ландшафтів стали невід’ємною частиною індустрії цифрових технологій завдяки своїй здатності автоматично створювати складні візуальні сцени з високим рівнем деталізації.

Процедурна генерація ландшафтів — це алгоритмічний підхід до створення тривимірних поверхонь за допомогою математичних методів. Зростання складності 3D-сцен, особливо у великих відкритих світах відеоігор, зумовлює підвищені вимоги до швидкості обробки, якості зображення та контролю над параметрами генерації. Одним із ключових алгоритмів, який дозволяє досягти реалістичності та деталізації, є алгоритм Marching Cubes.

Marching Cubes є методом створення полігональних поверхонь на основі воксельних даних, що дозволяє з високою точністю моделювати складні поверхні, такі як рельєфи ландшафтів, печери або органічні структури. Алгоритм отримав широку популярність у сферах ігрової індустрії, медичної візуалізації та наукових симуляцій завдяки здатності працювати з великими обсягами даних та зберігати високий рівень продуктивності.

Дослідження алгоритму Marching Cubes є актуальним через потребу у створенні ефективних методів генерації 3D-ландшафтів для використання у віртуальних середовищах, де важливим є баланс між реалістичністю сцени та продуктивністю обробки.

Мета дослідження – удосконалення процедурної генерації тривимірних ландшафтів у середовищі Unity шляхом розробки та програмної реалізації алгоритму Marching Cubes.

Для досягнення мети поставлено такі завдання:

1. Провести аналіз наукових джерел щодо методів процедурної генерації ландшафтів.

2. Дослідити алгоритми Marching Cubes та Marching Tetrahedra.

3. Визначити критерії вибору оптимального алгоритму для генерації тривимірних ландшафтів.

4. Реалізувати алгоритм Marching Cubes у середовищі Unity.

5. Провести тестування розробленого алгоритму для оцінки його ефективності та реалістичності результатів.

Об'єктом дослідження є процес процедурної генерації тривимірних ландшафтів.

Предметом дослідження – методи 3D-рендерингу та процедурної генерації текстур.

Практичне значення отриманих результатів полягає у створенні алгоритму процедурної генерації, який може бути використаний у розробці відеоігор, симуляторів та інших інтерактивних 3D-просторів. Це дозволить автоматизувати процес створення великих і реалістичних ландшафтів, мінімізуючи ручну роботу дизайнерів.

Новизна роботи полягає у реалізації процедурної генерації ландшафтів на основі оптимізованого алгоритму Marching Cubes у середовищі Unity.

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 69 сторінок основного тексту, 48 рисунків та 1 таблиці.

РОЗДІЛ 1. АНАЛІЗ ПУБЛІКАЦІЙ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз публікацій щодо поверхневого рендерингу зображень

Створення та візуалізація віртуальних ландшафтів є актуальним завданням у різних сферах, зокрема у розробленні навчальних і моделюючих середовищ, відеоігор, створенні візуальних матеріалів для кінематографу та анімації, а також у процесах моделювання природних явищ. В умовах зростання популярності технологій віртуальної реальності ця проблема набуває ще більшої значущості, що обумовлює потребу у розробленні нових високоефективних методів генерації ландшафтів.

Сучасні алгоритми генерації ландшафтів часто потребують подальшого ручного доопрацювання дизайнерами або ж автоматизованої деталізації складними алгоритмами, що імітують природні процеси, такі як ерозія, кліматичні зміни та погодні умови. Такий підхід є ефективним у випадках створення обмежених за розміром сцен або коли ландшафт налаштовується вручну, проте він не є оптимальним для генерації великих територій у повністю автоматизованому режимі [1].

Процедурна генерація ландшафтів має відповідати трьом ключовим критеріям:

1. Швидкість обчислень – алгоритми повинні використовувати мінімальні обчислювальні ресурси сучасних комп'ютерних систем.
2. Різноманітність і реалістичність – результати мають бути як випадковими, так і структурованими, щоб забезпечити створення різноманітного та достовірного візуального контенту.
3. Керованість і зручність – процес генерації має бути інтуїтивно зрозумілим для користувача.

Ландшафт виконує дві основні функції у візуалізації сцени:

- По-перше, він слугує базовою основою, на якій розміщуються всі інші об’єкти сцени, що сприяє просторовій орієнтації у віртуальному середовищі.
- По-друге, він суттєво підвищує рівень реалістичності сцени, додаючи деталі, які неможливо відтворити лише за допомогою текстур [3].

Для створення ландшафту в цифровому форматі необхідно отримати дані про місцевість, що може бути реалізовано за допомогою одного з трьох основних підходів:

1. Сканування місцевості:

Метод передбачає використання камер або інших пристроїв для фіксації поверхні реального ландшафту. Високоякісні зображення поверхні Землі, отримані за допомогою супутників або аерофотозйомки, забезпечують високу реалістичність цифрових копій місцевості.

2. Моделювання місцевості:

Даний підхід базується на створенні рельєфу вручну у спеціалізованому програмному забезпеченні для 3D-моделювання, такому як Blender, Cinema 4D, Maya або 3DS Max. Він дозволяє повністю контролювати кінцевий вигляд місцевості, що є перевагою при створенні художніх або фантастичних світів, проте моделювання реальних місцевостей потребує значних витрат часу та зусиль.

3. Процедурна генерація місцевості:

Цей підхід передбачає використання алгоритмів, які автоматично генерують дані про рельєф поверхні. Він є ефективним, коли потрібно створити великі віртуальні території за умови обмеженого обсягу пам’яті, оскільки алгоритми можуть генерувати нескінченно великі ландшафти в реальному часі без значних витрат дискового простору. Проте якість результату залежить від складності обраного алгоритму: простіші методи працюють швидше, але менш реалістично, тоді як складніші алгоритми забезпечують кращу якість зображення, але вимагають більше ресурсів.

4. Комбінований підхід:

Поєднання сканування реальних даних із подальшим алгоритмічним уточненням або ж генерації базової місцевості з подальшим ручним доопрацюванням. Такий метод дозволяє досягти балансу між деталізацією та контролем над кінцевим виглядом сцени.

Перші відомі випадки використання процедурної генерації датуються початком 1980-х років, коли через обмежені обчислювальні ресурси комп'ютерів створення великих і різноманітних світів зі згенерованим вмістом було неможливим. Внаслідок цього ранні приклади процедурної генерації застосовувалися переважно у 2D-іграх, де процес генерації не потребував значних обчислювальних потужностей і складних правил створення контенту. Яскравими представниками таких ігор є Rogue та Elite [2].

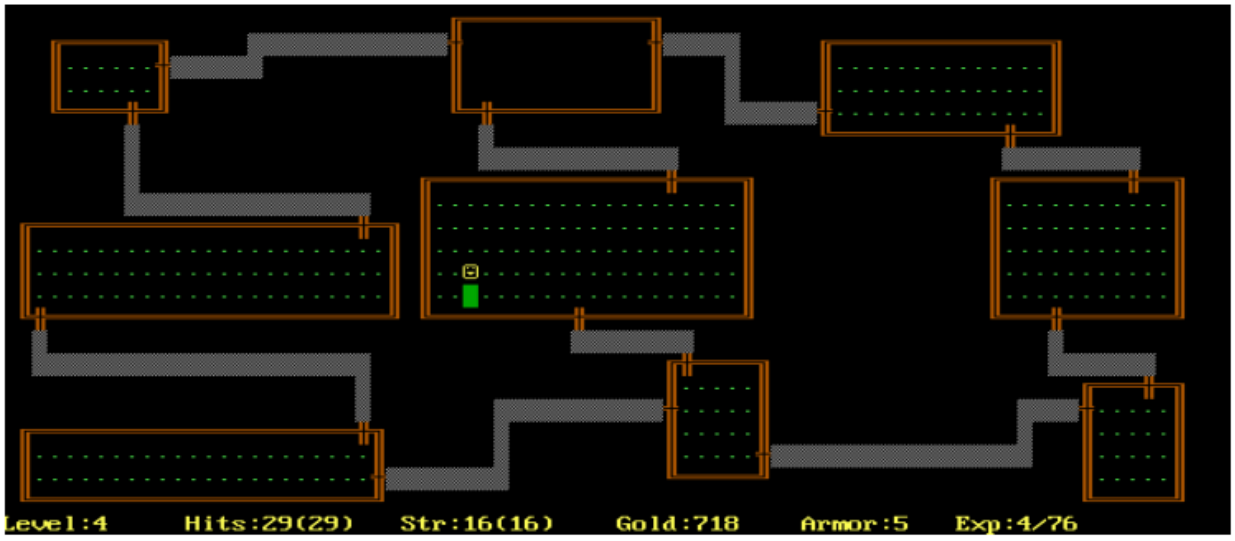


Рисунок 1.1 – Згенерований рівень Rogue

З розвитком комп’ютерних технологій та зростанням обчислювальних ресурсів з’явилася можливість застосування процедурної генерації у масштабних комерційних відеоіграх, що сприяло зростанню їхньої популярності. Використання процедурної генерації підвищувало інтерес гравців, оскільки забезпечувало унікальний ігровий досвід під час кожного проходження. Це створювало умови для багаторазового повторного проходження гри з отриманням нових вражень та змінюваних сценаріїв.

Прикладом такої гри є рольовий бойовик Diablo від компанії Blizzard, у якому процедурна генерація застосовувалася для створення ігрових карт, визначення кількості монстрів і предметів. У даному випадку цей метод не використовувався для повного створення нових об’єктів або рівнів, а слугував допоміжним засобом, що автоматично генерував характеристики озброєння та змінював окремі елементи структури ігрових рівнів [2]. На рисунку 1.2 зображено процедурно згенероване підземелля.

Рисунок 1.2 – Гра Diablo

Одним із сучасних прикладів застосування процедурної генерації є досвід компанії CD Projekt Red, представлений у звіті розробників гри The Witcher 3 [5]. Під час створення даної відеоігри однією з основних проблем стало формування великих відкритих просторів, оскільки розмір ігрового світу був значним, а ресурсів для його ручного створення було недостатньо. Наявна кількість художників і дизайнерів у компанії не дозволяла вручну опрацювати

всю територію, оскільки це потребувало б значних часових витрат або залучення великої команди фахівців. Відтак постала необхідність у пошуку оптимального підходу до автоматизації процесу генерації ігрових ландшафтів.

Для вирішення цього завдання було розроблено спеціалізовані програмні інструменти, призначені для процедурної генерації ландшафтів та рослинності. Обраний підхід виявився оптимальним з точки зору співвідношення витрат часу, ресурсів та отриманого результату. Програмістами було створено алгоритм, здатний розрізняти два типи поверхонь: природні (ґрунт, пісок, кам’янисті елементи) та штучні (цегляні, дерев’яні покриття, бруківка). Це дозволило реалізувати диференційоване накладання текстур та візуальних ефектів залежно від типу матеріалу. Наприклад, відображення снігу на поверхні ґрунту відрізнялося від його вигляду на кам’яній стіні.

Розроблений інструментарій також сприяв досягненню високої візуальної достовірності шляхом автоматичного створення плавних переходів між текстурами. Зокрема, це дозволило реалістично інтегрувати бруківку з природними елементами, такими як ґрунт, зберігаючи при цьому видимість окремих каменів на поверхні (рис. 1.3). Додатково процедурна генерація застосовувалася для автоматизованого розміщення дрібних об'єктів, таких як каміння та рослинність, що значно знизило трудомісткість процесу, оскільки ручне налаштування таких елементів для кожної локації потребувало б значних ресурсів.

Рисунок 1.3 – Інтеграція бруків в ланшафт землі у грі The Witcher 3

Таким чином, завдяки використанню процедурної генерації художникам було достатньо створити лише десять базових варіантів трави, а алгоритм автоматично адаптував їхній колір відповідно до кольорової гами поверхні, на якій вони розміщувалися. Для цього левел-артисти використовували пігментну карту — текстуру локації, відображену з верхнього ракурсу у низькій роздільній здатності. Колір конкретного поля або лугу накладався на текстуру трави за допомогою градієнта, що забезпечувало природний ефект: ближче до коріння відтінок трави набував кольору землі, тоді як верхня частина рослин поступово зберігала свій оригінальний колір. Такий підхід дозволяє досягти реалістичного вигляду рослинності без спотворення природних відтінків плодів та суцвіть [7].

1.2 Аналіз методів та підходів процедурної генерації ландшафтів та поверхонь

Для зберігання та відтворення даних про ландшафт у цифрових середовищах використовуються різноманітні методи представлення інформації. Вибір підходу визначає рівень деталізації, реалістичності та ефективності обробки ландшафтних даних, що є критично важливим у комп'ютерній графіці, ігровій індустрії, а також у симуляційному моделюванні. Серед основних принципів представлення даних для зберігання інформації про ландшафт виділяють:

- генерацію на основі карт висот (HeightMap);
- генерацію за допомогою іррегулярної сітки;
- генерацію на основі тайлів;
- генерацію із застосуванням алгоритму Marching Cubes;
- генерацію із використанням алгоритму Marching Tetrahedra.

Кожен із цих методів має власні переваги, недоліки та області застосування, залежно від вимог до деталізації, обчислювальної складності та обсягів даних [4, 8].

1.2.1 HeightMap

У комп’ютерній графіці карта висот, або поле висот, є ефективним методом представлення тривимірних даних поверхні у вигляді растрового зображення, де кожен піксель кодує висотну характеристику відповідної точки ландшафту. Вона використовується для моделювання топографії, створення реалістичних ландшафтів та формування складних поверхонь у 3D-сценах. Карти висот зберігаються у форматах графічних файлів, що забезпечує зручність редагування та візуальної перевірки даних. У такому форматі дві координати відповідають положенню пікселя на площині зображення, тоді як третя координата, що визначає висоту, закодована у значенні кольору: чим

яскравіший піксель, тим вища висота точки. Зазвичай використовуються монохромні карти висот, які обмежуються 256 градаціями сірого, однак застосування кольорових карт дозволяє досягти значно вищої деталізації завдяки розширеному діапазону значень [9].

Карти висот широко застосовуються у комп'ютерній графіці для різних цілей:

- при моделюванні нерівностей поверхні для коректного обчислення тіней у матеріалах;
- у методі displacement mapping (відображення переміщень) для геометричного зміщення вершин сітки відповідно до значень карти висот;
- при створенні тривимірних моделей ландшафтів шляхом перетворення карти висот у тривимірну полігональну сітку [10].

1.2.2 Генерацію за допомогою іррегулярної сітки

Одним із сучасних підходів до представлення даних для зберігання інформації про ландшафт є використання іррегулярної сітки, яка складається з вершин і зв'язків між ними [11]. Цей метод, на відміну від традиційних карт висот, дозволяє суттєво зменшити обсяг даних, що зберігаються, оскільки фіксуються лише значення висот кожної вершини та інформація про їх з'єднання. Такий підхід є особливо актуальним у спеціалізованих програмних середовищах, орієнтованих на розроблення відеоігор або роботу з тривимірною графікою, де дані зберігаються у вигляді компактних тривимірних моделей.

Ключовою перевагою є її здатність оптимізувати передачу та обробку даних у реальному часі. Наприклад, при використанні AGP (Accelerated Graphics Port) значно зменшується навантаження на пропускну здатність каналу завдяки скороченню обсягу переданих даних. Це робить її ефективним інструментом для створення динамічних і реалістичних візуалізацій, що можуть бути адаптовані до різних умов виконання.

Додатковою особливістю цього підходу є його гнучкість у відображенні складних топографічних структур, які важко реалізувати за допомогою традиційних карт висот. Іррегулярна сітка дозволяє деталізувати окремі ділянки ландшафту без необхідності збільшення обсягу даних для всієї сцени, що забезпечує не лише ефективність, але й високий рівень реалізму. Приклад іррегулярної сітки представлено на рисунку 1.4, що демонструє її застосування для відображення складних поверхонь.

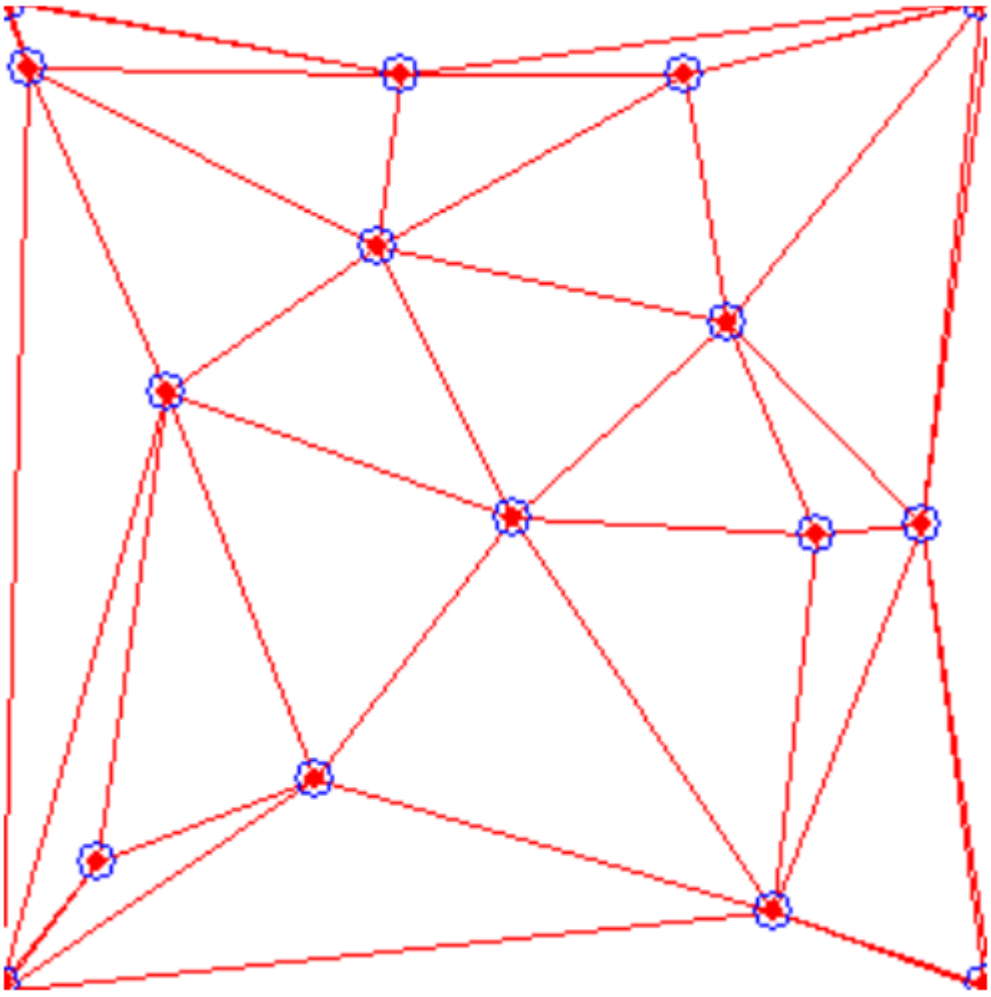


Рисунок 1.4 – Приклад зображення іррегулярної сітки

1.2.3 Генерація на основі тайлів

Генерація ландшафтів на основі тайлів є поширеним підходом, переважно використовується для створення двовимірного ігрового контенту,

зокрема рівнів у 2D-іграх. Цей метод передбачає формування ігрового простору шляхом компонування невеликих графічних елементів — плиток (тайлів), які мають квадратну, рідше прямокутну, паралелограмну або шестикутну форму. Плитки розміщуються у вигляді сітки, створюючи цілісну ігрову зону.

Візуальна цілісність рівнів побудованих на основі тайлів часто приховує їх модульну природу і для гравця структура з окремих плиток може бути непомітною. Повний набір графічних елементів, доступний для створення ігрового середовища, називається tilemap або набором тайлів. Він зазвичай містить текстурні елементи, необхідні для побудови різноманітних локацій, таких як підлога, стіни, перешкоди або декоративні об’єкти.

Ігри створені за цим принципом здебільшого використовують перспективу зверху вниз, вигляд збоку або псевдо-3D. Тайлова генерація є особливо ефективною для створення процедурних рівнів, оскільки дозволяє швидко змінювати структуру сцени, забезпечуючи як випадковість, так і контрольовану архітектуру простору. Такий підхід часто застосовується в жанрах платформерів, RPG та стратегій у реальному часі.

Приклад рівня, згенерованого за допомогою тайлів, представлено на рисунку 1.5, де демонструється, як модульні елементи комбінуються для створення повноцінного ігрового середовища [12].

Рисунок 1.5 – Згенерований рівень на основі тайлової системи

1.2.4 Marching Cubes

Алгоритм Marching Cubes є методом, що використовується у комп’ютерній графіці для побудови тривимірних поверхонь на основі скалярного поля. Принцип роботи алгоритму полягає в тому, що він послідовно обходить скалярне поле, беручи групу з восьми сусідніх точок, які утворюють уявний куб. Після цього алгоритм аналізує, які полігони необхідні для представлення частини ізоповерхні, що перетинає межі цього куба. У результаті всі визначені полігони об'єднуються для створення цілісної тривимірної поверхні.

Процес генерації базується на заздалегідь розрахованому масиві, що містить 256 можливих конфігурацій многокутників для куба. Примітно, що лише 15 з них є унікальними, тоді як решта формується шляхом обертань і симетричних перетворень базових конфігурацій. Такий підхід дозволяє

ефективно створювати складні 3D-моделі, зокрема для візуалізації медичних даних, геометричних об'єктів або процедурно згенерованих ландшафтів.

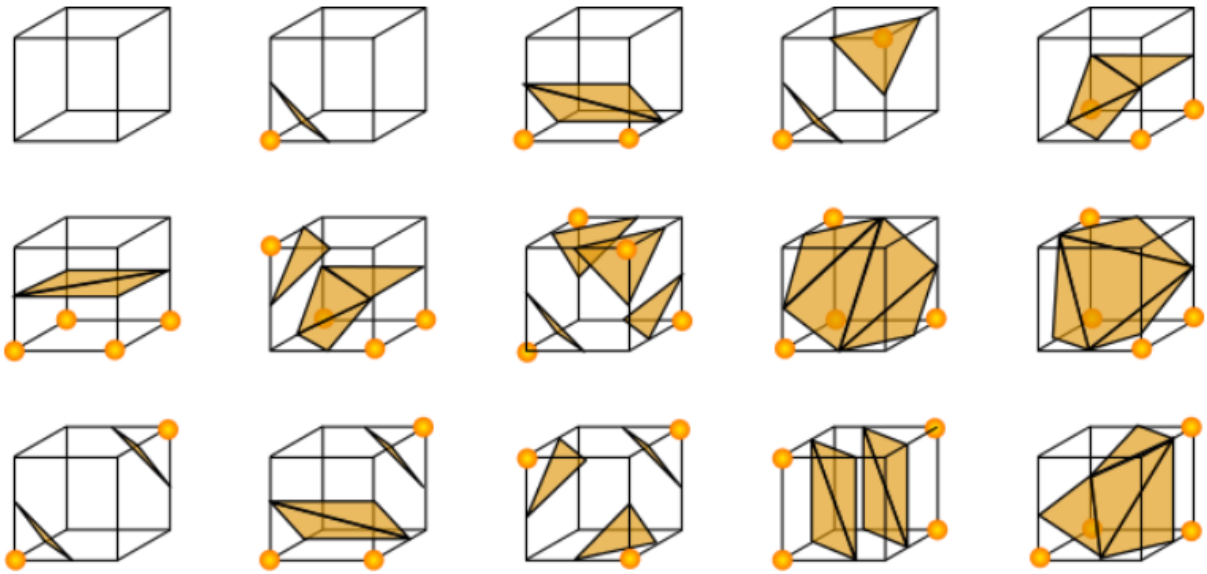


Рисунок 1.6 – 15 унікальних конфігурацій куба у алгоритмі Marching cubes

Алгоритм Marching Cubes увійшов в історію комп’ютерної графіки не лише завдяки своїй ефективності, але й через скандал у сфері патентного права. Незважаючи на відносну очевидність запропонованого підходу, метод було запатентовано, що викликало суперечки щодо можливості патентування алгоритмів у галузі програмного забезпечення. Як альтернативу, з метою обійти патентні обмеження та усунути проблему неоднозначності, у деяких конфігураціях було розроблено алгоритм Marching Tetrahedra, що використовує тетраедри замість кубів для поділу простору, забезпечуючи вищу точність у складних областях моделювання [13].

1.2.5 Marching Tetrahedra

Алгоритм Marching Tetrahedra є модифікацією методу Marching Cubes, яка застосовує тетраедри замість кубів для генерації тривимірних поверхонь у скалярних полях. Основною відмінністю цього алгоритму є використання

лише восьми можливих конфігурацій для тетраедра, що значно спрощує обчислювальний процес порівняно з 256 конфігураціями у Marching Cubes (рис. 1.7).

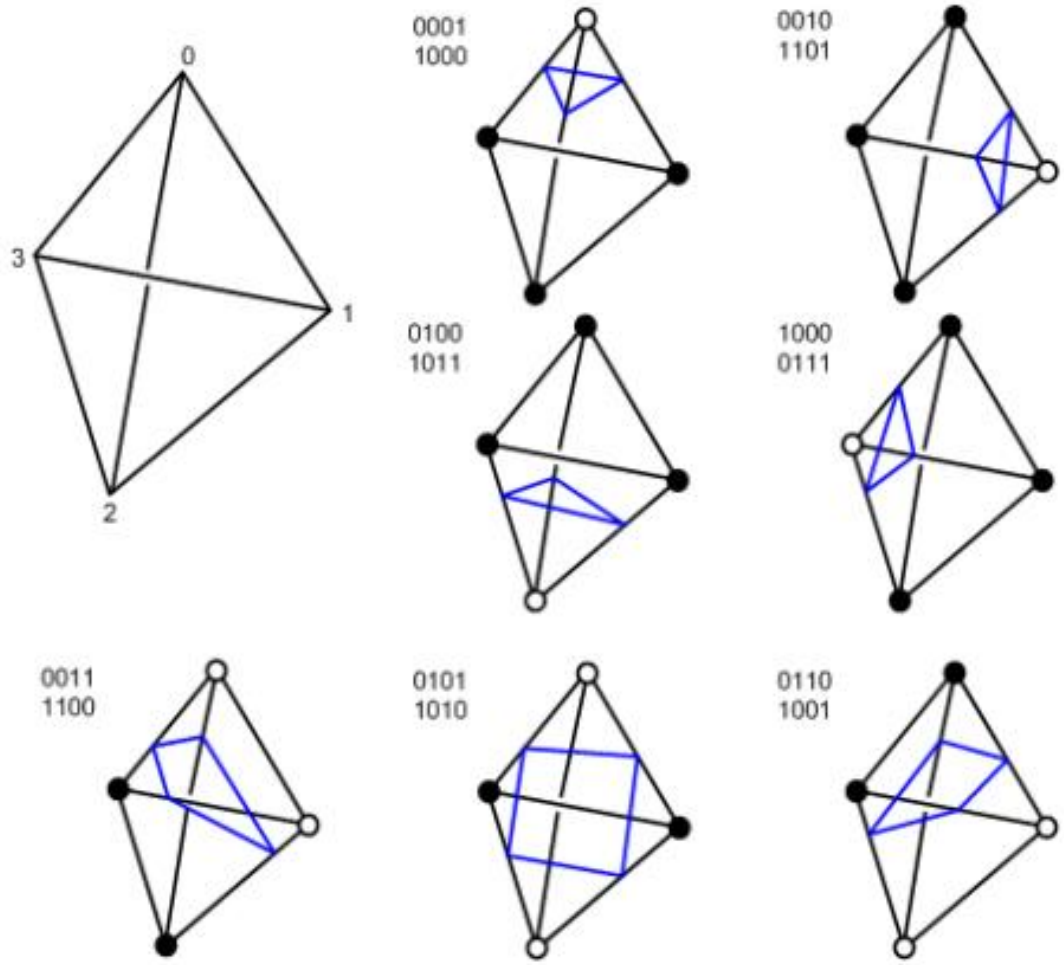


Рисунок 1.7 – 8 унікальних конфігурацій тетраедра

Принцип роботи алгоритму полягає у поділі кожного куба на шість рівних тетраедрів. Це досягається шляхом триразового діагонального розрізання куба, де кожна з трьох пар протилежних граней розділяється навпіл по діагоналі (рис. 1.9).

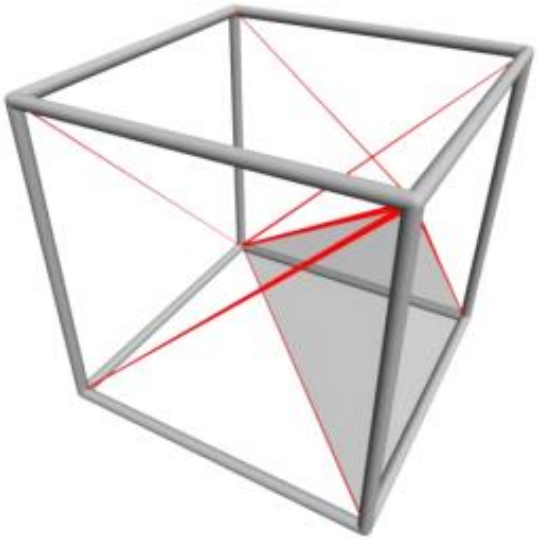


Рисунок 1.9 – Куб, розділений на шість тетраедрів, один із тетраедрів затінений

У результаті, всі тетраедри мають одну спільну головну діагональ куба. Такий підхід змінює топологію куба, збільшуючи кількість ребер з 12 до 19: зберігаються вихідні 12 ребер, додаються 6 діагоналей граней і головна діагональ. Як і у випадку з Marching Cubes, перетини цих ребер із ізоповерхнею обчислюються шляхом лінійної інтерполяції значень у вузлах сітки [14].

Спрощена структура тетраедрів та трикутний характер геометрії роблять цей алгоритм ефективним для апаратного прискорення у графічних процесорах. Завдяки цій властивості алгоритм Marching Tetrahedra став об'єктом численних досліджень у галузі GPGPU (General-Purpose Computing on Graphics Processing Units), де він активно використовується для оптимізації обчислювальних процесів під час візуалізації складних поверхонь [9].

1.3 Висновок до першого розділу

Створення та візуалізація віртуальних ландшафтів є актуальним завданням у різних сферах, зокрема у розробленні навчальних і моделюючих середовищ, відеоігор, створенні візуальних матеріалів для кінематографу та анімації, а також у процесах моделювання природних явищ. В умовах зростання популярності технологій віртуальної реальності ця проблема набуває ще більшої значущості, що обумовлює потребу у розробленні нових високоефективних методів генерації ландшафтів.

Сучасні алгоритми генерації ландшафтів часто потребують подальшого ручного доопрацювання дизайнерами або ж автоматизованої деталізації складними алгоритмами, що імітують природні процеси, такі як ерозія, кліматичні зміни та погодні умови. Такий підхід є ефективним у випадках створення обмежених за розміром сцен або коли ландшафт налаштовується вручну, проте він не є оптимальним для генерації великих територій у повністю автоматизованому режимі [1].

Процедурна генерація ландшафтів має відповідати трьом ключовим критеріям: швидкість обчислень, різноманітність і реалістичність, керованість і зручність.

У даному розділі проведено дослідження методів та підходів процедурної генерації ландшафтів, включаючи генерацію на основі карт висот, іррегулярних сіток, тайлів, а також алгоритми Marching Cubes та Marching Tetrahedra. Проаналізовано ключові критерії оцінки алгоритмів, такі як швидкість обчислень, рівень реалістичності, можливість редагування та придатність для тривимірного моделювання.

Особливу увагу приділено алгоритму Marching Cubes, який дозволяє створювати складні тривимірні ландшафти з високим рівнем деталізації при мінімальних обчислювальних витратах.

Метою дослідження є розробка алгоритму процедурної генерації ландшафтів на основі Marching Cubes для створення реалістичних тривимірних сцен у віртуальних середовищах.

Для досягнення поставленої мети було визначено та виконано такі завдання дослідження:

1. Проведено аналіз наукових джерел щодо процедурної генерації ландшафтів.
2. Досліджено алгоритми Marching Cubes та Marching Tetrahedra.
3. Визначено критерії для вибору оптимального методу генерації ландшафтів.
4. Сформульовано вимоги до програмної реалізації алгоритму.
5. Виконано програмну реалізацію алгоритму в середовищі рушія Unity.
6. Проведено тестування програмного продукту для перевірки його відповідності функціональним і нефункціональним вимогам.

Вхідними даними для алгоритму є параметри генерації, такі як карта висот, тип сітки та алгоритмічні налаштування, що дозволяють автоматизувати процес створення віртуальних ландшафтів із заданим рівнем деталізації.

РОЗДІЛ 2. АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ АЛГОРИТМУ ГЕНЕРАЦІЇ ЛАНДШАФТУ

2.1 Вибір методу реалізації генерації процедурної анімації

Зігідно проаналізованих методів процедурної генерації було сформовано таблцю 2.1 за кількома критеріями, такими як – тип сітки, обсяг даних, редагованість, рівень складності та придатність для тривимірного моделювання.

Таблиця 2.1

Порівняльний аналіз методів

Метод генерації	Тип сітки	Обсяг даних	Редаго-ваність	Рівень склад-ності	Прида-тність для 3D
Генерація на основі карти висот	Регулярна	Великий	Висока	Середній	Частково
Генерація на основі іррегулярної сітки	Нерегулярна	Низький	Низька	Високий	Висока
Генерація на основі тайлів	Регулярна (тайли)	Середній	Висока	Низький	Низька
Marching Cubes	Регулярна (куби)	Середній	Середня	Високий	Висока
Marching Tetrahedra	Нерегулярна (тетраедри)	Середній	Середня	Високий	Висока

Узагальнюючи наведені методи процедурної генерації ландшафтів, можна зробити висновок, що кожен з підходів має свої унікальні переваги та

недоліки, які визначають їхнє застосування у різних сферах. Генерація на основі карти висот забезпечує високу наочність і простоту редагування, проте потребує зберігання великого обсягу даних. Іррегулярні сітки ефективніші з точки зору обсягу даних, але складніші для редагування та динамічного освітлення. Тайлові системи підходять для 2D-генерації, хоча й обмежені у використанні для 3D-контенту.

Алгоритми Marching Cubes та Marching Tetrahedra є ефективними інструментами для створення складних 3D-ландшафтів. Зокрема, Marching Cubes дозволяє генерувати реалістичні поверхні з воксельних даних, забезпечуючи баланс між деталізацією та продуктивністю. Завдяки процедурному підходу він дозволяє створювати унікальні сцени з мінімальними витратами на ручне моделювання. Це робить Marching Cubes оптимальним вибором для розробників, які прагнуть створити реалістичні тривимірні ландшафти при обмежених обчислювальних ресурсах. Його гнучкість і відкритість сприяють широкому застосуванню у відеоіграх, наукових симуляціях та медичній візуалізації.

2.2 Вибір програмного забезпечення для реалізації алгоритму

У сфері процедурної генерації ландшафтів існує широкий спектр спеціалізованих програмних рішень, кожне з яких орієнтоване на конкретні завдання. Деякі з них призначені для створення 3D-моделей ландшафтів, інші — для моделювання інженерних конструкцій або анімацій. Вибір відповідного програмного забезпечення відіграє ключову роль, оскільки від нього залежить якість, ефективність та швидкість реалізації поставлених завдань.

У межах цієї роботи було визначено перелік програмних продуктів для аналізу та порівняння їхніх функціональних можливостей. На основі проведеного дослідження буде обрано програмне забезпечення, яке найкраще підходить для створення детальної 3D-моделі поверхні ландшафту з використанням процедурної генерації.

2.2.1 Autodesk Maya

Autodesk Maya — програмний комплекс, розроблений компанією Autodesk, який призначений для 3D-моделювання, композитингу, анімації та рендерингу. Найчастіше застосовується для створення художніх і анімаційних фільмів, а також у телевізійній індустрії. Приклад використання програмного середовища зображений на рисунку 2.1.

У Maya інтегровано скриптову мову Maya Embedded Language (MEL), за допомогою якої можна створювати скрипти для виконання різноманітних дій, що забезпечує практично необмежені можливості для користувача [17].

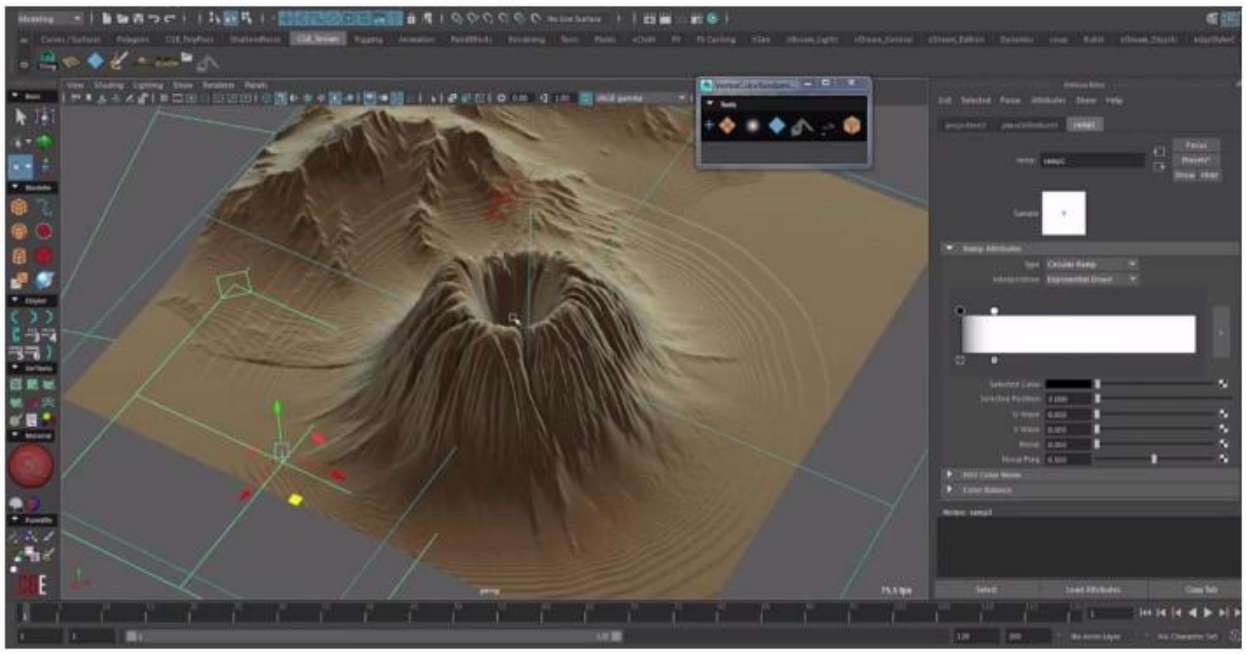


Рисунок 2.1 – Вигляд Autodesk Maya

Autodesk Maya є потужним програмним продуктом, який значно спрощує процес створення персонажів завдяки наявності всіх необхідних інструментів. Програма підтримує автоматизоване моделювання персонажів, пропорційних людському тілу. Додавання текстур здійснюється за допомогою

функції згортки, що забезпечує максимальну відповідність творчому задуму. Також передбачені інструменти для створення волосся та шерсті.

Основні можливості Autodesk Maya включають:

- роботу з кривими, зокрема NURBS;
- полігональне моделювання;
- різні способи накладання текстур і матеріалів;
- інструменти для скульптурування;
- широкий набір функцій для створення анімації;
- симуляцію твердо- та м’якотілих об'єктів;
- моделювання рідин;
- створення спеціальних ефектів, таких як дим, хмари та атмосферні явища.

Важливою особливістю цього програмного забезпечення є модуль PaintEffects, який дозволяє малювати віртуальним пензлем 3D-об'єкти, такі як квіти, траву, об’ємні візерунки тощо.

Попри значні переваги, Autodesk Maya має і певні недоліки: програму складно освоїти, вона висуває високі вимоги до апаратного забезпечення та має високу вартість ліцензії. Крім того, програмне забезпечення в першу чергу орієнтоване на створення анімації, що робить його менш зручним для моделювання ландшафтів.

2.2.2 Three.js

Three.js — це потужна JavaScript-бібліотека, призначена для створення 3D-графіки у веб-браузері за допомогою WebGL. Вона спрощує процес розробки інтерактивних 3D-сцен, анімацій та візуалізацій без необхідності глибоких знань у сфері WebGL. Three.js часто використовується для створення веб-ігор, візуалізацій архітектурних об'єктів, інтерактивних сайтів та віртуальної реальності [18].

Бібліотека Three.js підтримує такі функції:

- створення та управління 3D-сценами, камерами та об'єктами;
- використання примітивів для моделювання (куби, сфери, циліндри тощо);
- підтримка полігонального моделювання;
- різні види освітлення (точкові, спрямовані, навколишні тощо);
- кілька форматів текстурування та матеріалів, зокрема PBR (Physically Based Rendering);
- інструменти для створення анімації, включаючи ключові кадри та скелетну анімацію;
- симуляція частинок для створення спецефектів (вибухи, вогонь, дощ);
- можливості для створення шейдерів за допомогою GLSL.

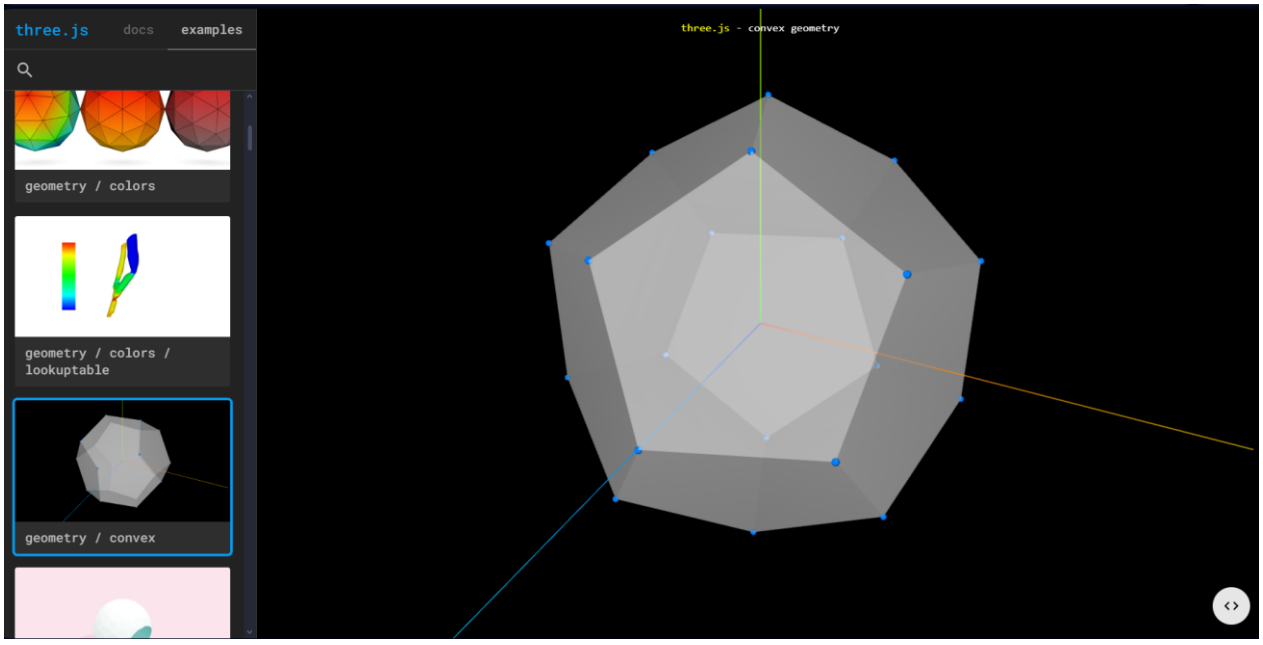


Рисунок 2.2 – Three.js

Головною особливістю Three.js є легкість інтеграції з HTML/CSS та підтримка WebGL, що дозволяє створювати візуально насичені сцени без додаткових плагінів.

Недоліки бібліотеки:

- висока вимогливість до продуктивності системи для складних сцен;
- необхідність базових знань JavaScript і комп'ютерної графіки для використання;
- складність оптимізації для великих сцен та обмежена підтримка мобільних пристроїв.

2.2.3 Unity

Unity — це популярний кросплатформний ігровий рушій, розроблений компанією Unity Technologies, який використовується для створення 2D і 3D-ігор, інтерактивних симуляцій, анімацій та візуалізацій. Завдяки зручному інтерфейсу та потужному функціоналу, Unity підходить як для інді-розробників, так і для великих студій. Найчастіше використовується для розробки відеоігор, VR/AR-проектів та інтерактивних презентацій [15].

Рушій Unity має такі можливості:

- підтримка 2D та 3D-графіки;
- вбудовані інструменти для фізики (PhysX для 3D, Box2D для 2D);
- полігональне моделювання та робота зі спрайтами;
- декілька способів накладання текстур і матеріалів, зокрема PBR (Physically Based Rendering);
- анімаційна система на базі механізму Animator з підтримкою ключових кадрів та скелетної анімації;
- можливість програмування мовами C# через інтеграцію з Visual Studio;
- інтеграція з пакетом Unity Asset Store для швидкого доступу до готових моделей, текстур та скриптів;
- підтримка VR/AR-платформ (Oculus, HTC Vive, HoloLens);
- кросплатформеність із можливістю експорту на Windows, macOS, Android, iOS, WebGL, PlayStation, Xbox тощо.

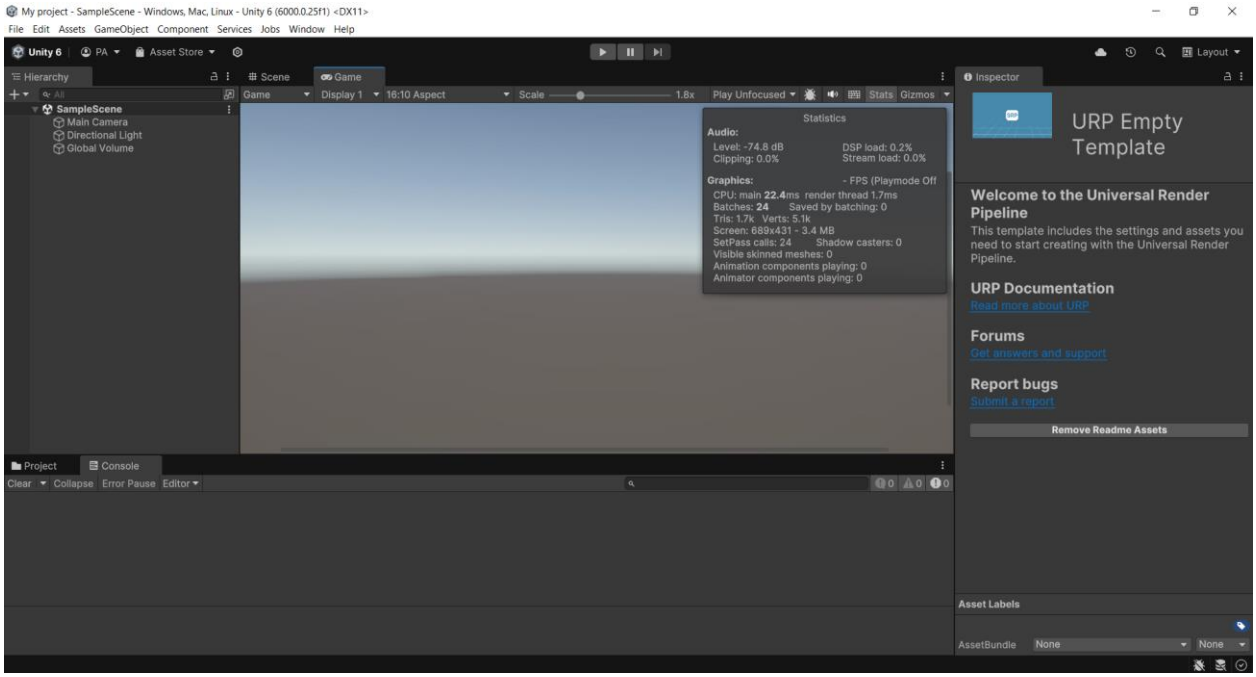


Рисунок 2.3 – Unity

Головною особливістю Unity є модульна структура, яка дозволяє розширювати функціональність через додаткові пакети (наприклад, URP та HDRP для покращеного рендерингу).

Недоліки рушія:

- висока вимогливість до продуктивності при роботі зі складними сценами;
- складність оптимізації для мобільних пристроїв;
- обмежені можливості для фотореалістичної графіки порівняно з рушіями на зразок Unreal Engine;
- безкоштовна версія має обмеження щодо монетизації та функціональності.

Отже вибір програмного забезпечення для реалізації алгоритму Marching Cubes у процедурній генерації ландшафтів є ключовим рішенням, що впливає на ефективність розробки, продуктивність та якість візуалізації. Розглянуто широкий спектр інструментів, які можуть бути використані для цієї задачі – Autodesk Maya, Three.js та Unity.

Розгляд вибору Unity як основного рушія для впровадження алгоритму Marching Cubes ґрунтується на низці переваг:

- розширені можливості для роботи з тривимірною графікою,
- містить вбудований фізичний модуль PhysX,
- використання мови програмування C#,

які роблять його придатним для створення процедурно згенерованих ландшафтів.

2.3 Формування вимог до проекту

Вимоги до алгоритму процедурної генерації Marching Cubes визначають ключові характеристики, необхідні для створення ефективної системи генерації тривимірних ландшафтів з використанням процедурних шумів. Визначення цих вимог забезпечує відповідність функціональності потребам користувача, стабільності роботи алгоритму та високій якості візуалізації.

Функціональні вимоги:

1. Генерація шуму
2. Обробка воксельних даних
3. Реалізація алгоритму Marching Cubes для генерації полігональних сіток на основі воксельних даних.
4. Інтеграція алгоритму генерації ландшафтів у Unity.
5. Можливість реального часу оновлення ландшафту (Regenerate).

Нефункціональні вимоги:

1. Продуктивність:
   - Мінімізація обчислювальних витрат шляхом оптимізації генерації шуму.
   - Підтримка великих воксельних сіток із збереженням стабільної продуктивності.
   - Оптимізація роботи з полігональними сітками для зменшення кількості вершин.

2. Стабільність та надійність:

- Коректна робота алгоритму з різними розмірами воксельної сітки.
- Відсутність артефактів при генерації сіток, включаючи перетин поверхонь та неправильні нормалі.

3. Якість візуалізації:

- Забезпечення плавних переходів між рівнями деталізації.
- Відсутність графічних артефактів при генерації складних поверхонь.
- Підтримка параметру SmoothNormals для покращення візуальної якості.

4. Сумісність:

- Повна інтеграція з середовищем Unity.
- Сумісність із стандартними компонентами MeshFilter та MeshRenderer.

Ці вимоги забезпечують створення потужного та гнучкого алгоритму процедурної генерації тривимірних моделей, що може бути використаний для створення ігрових ландшафтів, симуляторів та інших візуалізаційних проектів.

2.4 Проектування алгоритму процедурної генерації ландшафту

Процедурна генерація тривимірних ландшафтів за допомогою алгоритму Marching Cubes складається з двох ключових етапів: генерації шуму (Noise) для формування основи ландшафту та побудови полігональної сітки на базі цього шуму (рис. 2.1).

Рисунок 2.1 – Етапи алгоритму

На першому етапі використовується алгоритм генерації псевдовипадкового шуму, Noise, для створення скалярного поля, яке визначає висоту ландшафту. Шум застосовується для створення базової карти висот, де значення кожної точки відповідає рівню висоти або щільності в певній області простору. Цей етап дозволяє контролювати основні характеристики ландшафту, такі як масштаб, частота та рівень деталізації, забезпечуючи можливість створення як гладких пагорбів, так і складних печерних систем.

На другому етапі алгоритм Parallel Marchine Cubes інтерпретує створене скалярне поле, перетворюючи його в тривимірну полігональну сітку. Для цього простір розділяється на вокселі (куби) і на основі значень скалярного поля у вершинах кожного куба (паралельно в окремому потоці), визначається чи проходить через нього поверхня ландшафту. Алгоритм аналізує локальні значення та будує трикутники, що формують поверхню. Завдяки цьому процесу можна отримати складні але оптимізовані для рендерингу поверхні. Етапи роботи алгоритму:

1) Ініціалізації змінних, визначення 8 вершин куба

Визначення восьми вершин куба знаходяться всередині або поза поверхнею (рис. 2.2). Це здійснюється шляхом перевірки кожної вершини куба з пороговим значенням. Якщо значення менше то вона вважається всередині

поверхні. Та створюється бітове число - це створює унікальне ціле число від 0 до 255 (2^8 комбінацій для 8 вершин куба).

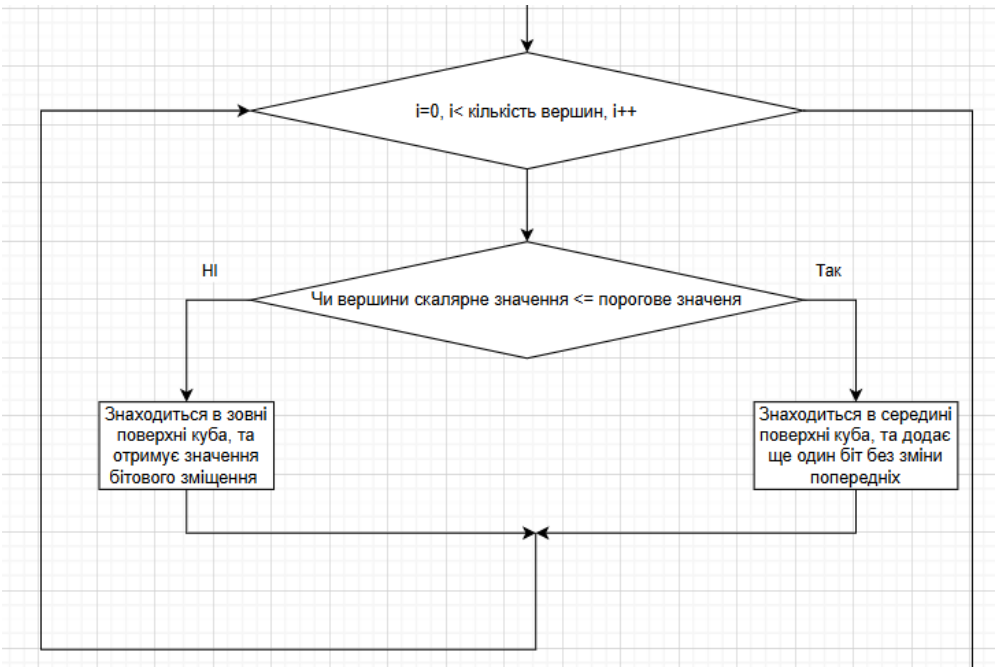


Рисунок 2.2 – Діаграма роботи алгоритму (крок 1)

2) Визначення пересічених ребер куба

Визначається, які ребра куба формують поверхню (рис. 2.3). Якщо жодне ребро не перетинається, алгоритм завершує обробку для цього куба, так як з ним не можливо створити поверхню ландшафту. Якщо дві вершини, які формують ребро, мають значення по різні боки від (одна всередині, інша зовні), то поверхня проходить через це ребро.

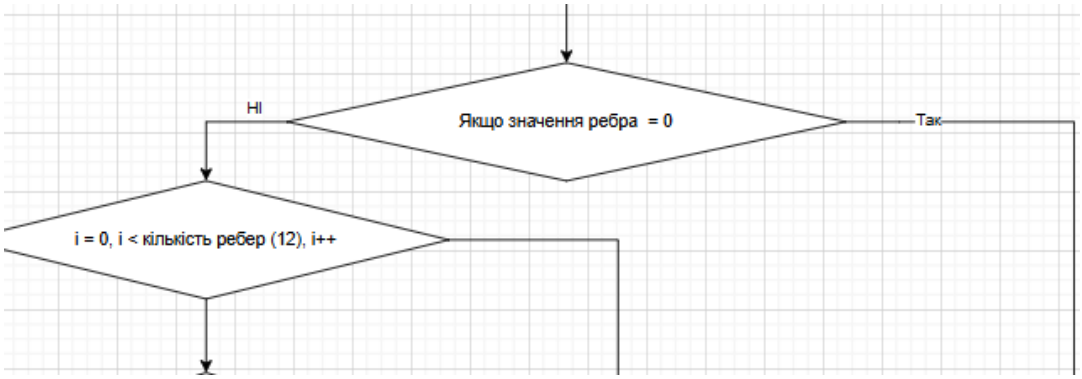


Рисунок 2.3 – Діаграма роботи алгоритму (крок 2)

3) Обчислення точок перетину поверхні з кожним ребром

Якщо знайдені перетнуті ребра (рис. 2.4), алгоритм обчислює точку перетину поверхні з кожним ребром. Це робиться шляхом лінійної інтерполяції між двома вершинами ребра на основі їхніх значень.

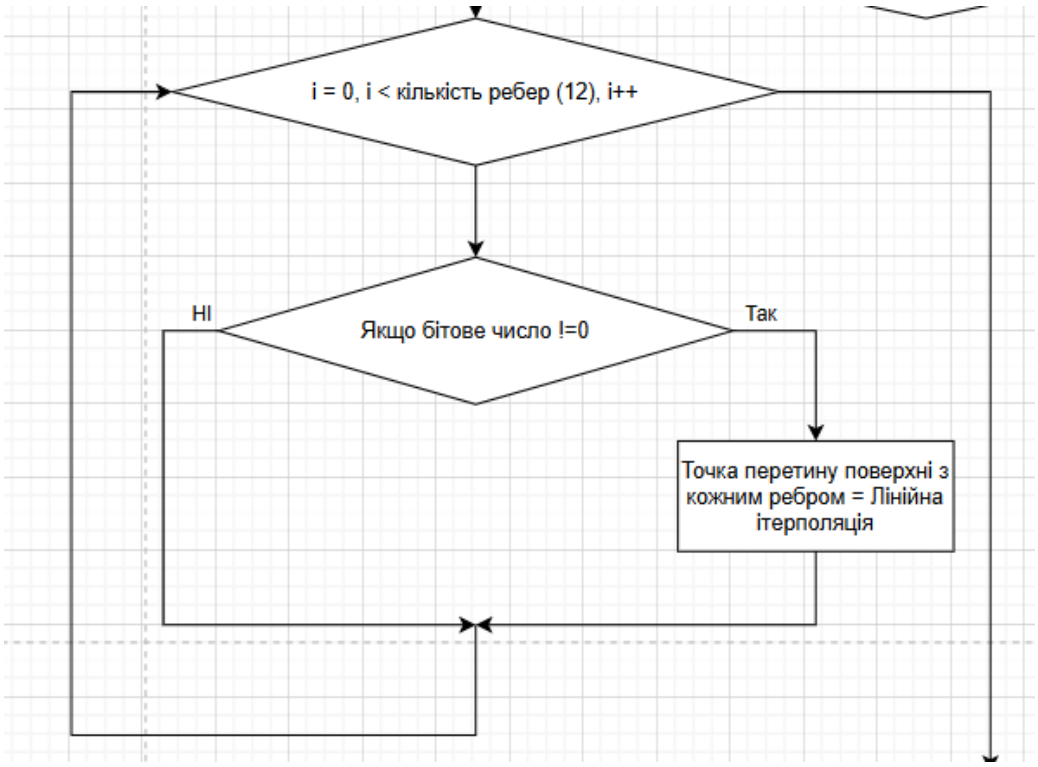


Рисунок 2.4 – Діаграма роботи алгоритму (крок 3)

4) Генерація трикутників

Останній етап (рис. 2.5) — створення трикутників. Після обчислення точок перетину алгоритм створює трикутники, що формують полігональну сітку. Для цього використовується таблиця трикутників, яка містить інформацію про те, які вершини формують трикутники для кожної можливої комбінації перетинів. Можливо згенерувати до 5 трикутників на один куб.

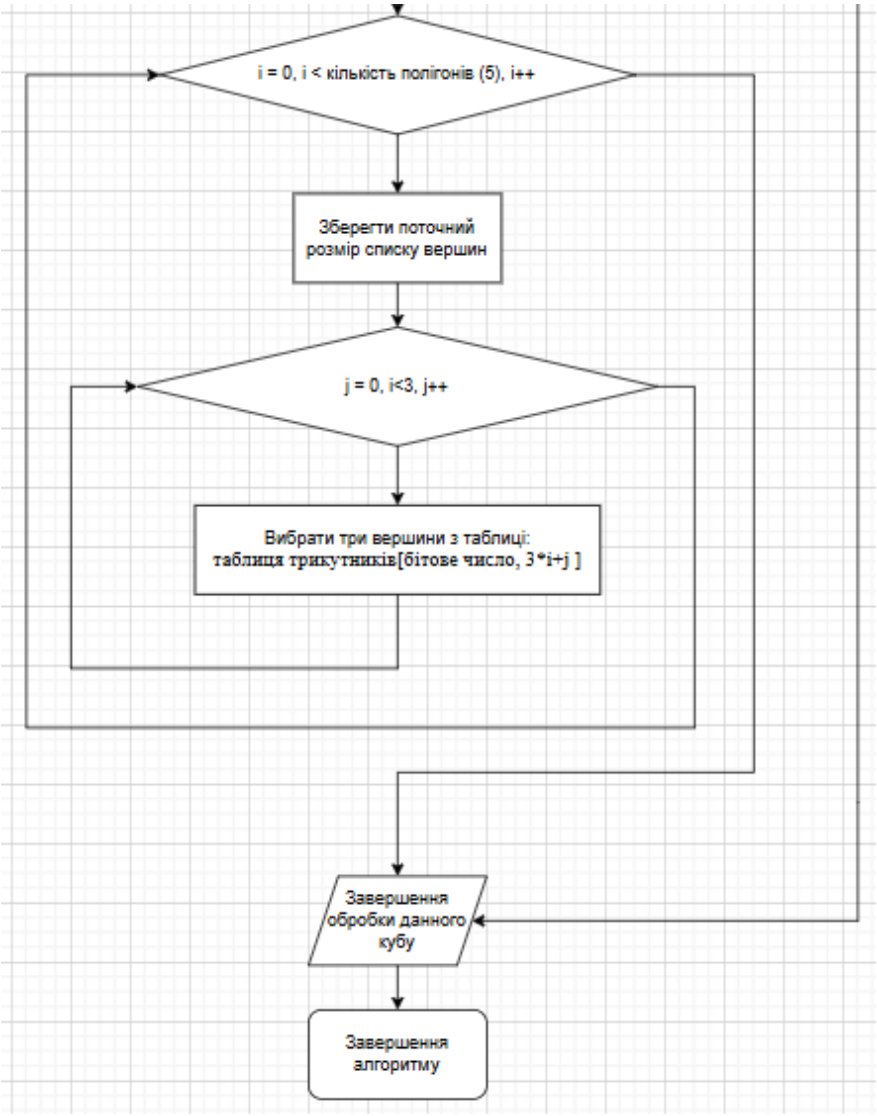


Рисунок 2.5 – Діаграма роботи алгоритму (крок 4)

Контрольованість шуму дозволяє налаштовувати параметри, такі як частота, амплітуда, та масштаб. Це дає змогу створювати як плавні пагорби, так і складніші гірські рельєфи або підводні структури, що важливо для різноманітності в ігрових світах.

Гладкість і безперервність є ключовими для створення реалістичних форм. Алгоритми, як-от Perlin Noise або Simplex Noise, генерують плавні переходи між значеннями, що запобігає появі різких переходів та артефактів у фінальному ландшафті.

Фрактальність шуму означає, що його структури можуть повторюватися на різних масштабах, створюючи складні деталі ландшафту. Це дозволяє

відтворювати природні особливості, такі як гірські хребти або узбережжя, з високим рівнем деталізації.

Комбінація процедурного шуму та паралельної реалізації алгоритму Marching Cubes дозволяє створювати реалістичні, детально опрацьовані ландшафти із високим рівнем контролю над їхньою формою та складністю.

2.4 Висновок до другого розділу

В другому розділі кваліфікаційної роботи зігідно проаналізованих методів процедурної генерації було проведено аналіз за кількома критеріями, такими як – тип сітки, обсяг даних, редагованість, рівень складності та придатність для тривимірного моделювання.

Узагальнюючи наведені методи процедурної генерації ландшафтів, можна зробити висновок, що кожен з підходів має свої унікальні переваги та недоліки, які визначають їхнє застосування у різних сферах. Генерація на основі карти висот забезпечує високу наочність і простоту редагування, проте потребує зберігання великого обсягу даних. Іррегулярні сітки ефективніші з точки зору обсягу даних, але складніші для редагування та динамічного освітлення. Тайлові системи підходять для 2D-генерації, хоча й обмежені у використанні для 3D-контенту.

Алгоритми Marching Cubes та Marching Tetrahedra є ефективними інструментами для створення складних 3D-ландшафтів. Зокрема, Marching Cubes дозволяє генерувати реалістичні поверхні з воксельних даних, забезпечуючи баланс між деталізацією та продуктивністю.

У розділі було проведено аналіз засобів реалізації алгоритму процедурної генерації ландшафтів, включаючи програмні рішення для створення тривимірних сцен, такі як Unity, Autodesk Maya та Three.js. Було здійснено порівняльний аналіз цих інструментів за критеріями функціональності, підтримки алгоритмів процедурної генерації, продуктивності та гнучкості налаштувань.

Рушій Unity обрано основним середовищем для реалізації алгоритму Marching Cubes. Такий вибір обумовлений широкими можливостями рушія для роботи з тривимірними воксельними даними, підтримкою C# для розробки скриптів, а також наявністю вбудованих фізичних та візуальних компонентів, що дозволяють створювати складні 3D-ландшафти з високим рівнем деталізації.

Для досягнення поставлених цілей було виконано такі завдання:

1. Проаналізовано доступні програмні засоби для реалізації процедурної генерації ландшафтів.
2. Оцінено функціональні можливості Unity, Autodesk Maya та Three.js.
3. Обґрунтовано вибір Unity як найбільш придатного середовища для розробки алгоритму Marching Cubes.

Результати даного дослідження стали основою для подальшого етапу роботи, присвяченого проектуванню та програмній реалізації алгоритму процедурної генерації ландшафтів.

РОЗДІЛ 3. РОЗРОБКА АЛГОРИТМУ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ЛАНДШАФТУ ГРИ

3.1 Розробка алгоритму

Проектування системи процедурної генерації ландшафтів поділяється на дві основні частини, кожна з яких відповідає за окремий етап обробки даних та формування кінцевого результату (рис. 3.1).

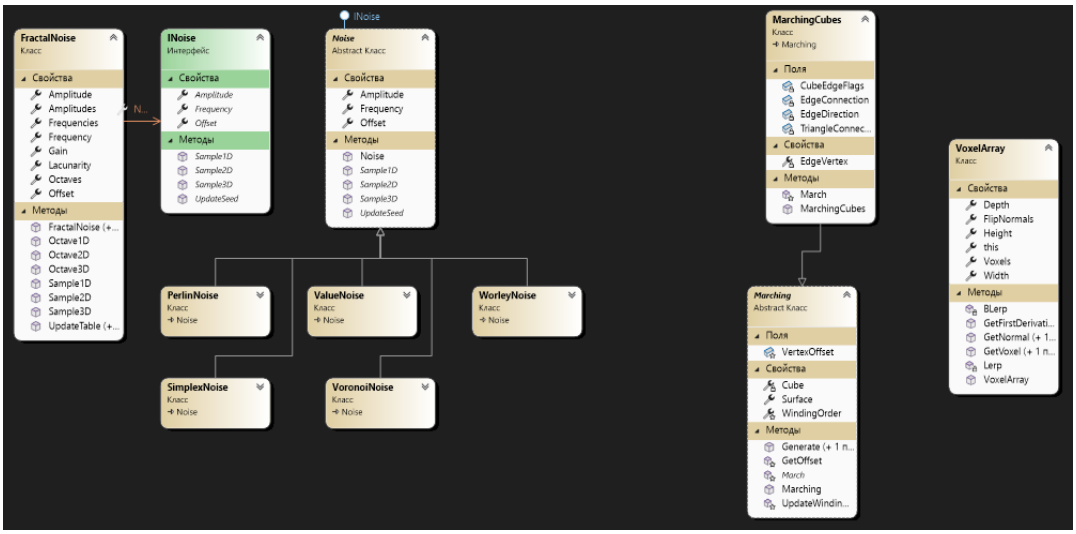


Рисунок 3.1 – Діаграма класів проекту

Перша частина передбачає розробку базових та спеціалізованих класів для генерації шуму, який використовується як основа для формування карти висот. Базовим елементом системи є абстрактний інтерфейс INoise, який визначає загальну структуру методів для створення процедурних шумів. Він слугує основою для всіх подальших реалізацій.

Клас Noise виступає базовою реалізацією, яка може включати стандартні методи ініціалізації, масштабування та нормалізації шумових значень. Він забезпечує спільний функціонал для всіх типів шуму: PerlinNoise, ValueNoise, WorleyNoise, SimplexNoise, VoronoiNoise, FractalNoise.

Друга частина проекту зосереджена на побудові тривимірного ландшафту на основі згенерованих шумів. Вона реалізована через наступний набір класів, відповідальних за обробку воксельних даних та побудову полігональної сітки: MarchingCubes, Marching, VoxelArray.

Щоб розпочати створення процедурного ландшафту, необхідно спочатку реалізувати систему для генерації шуму, яка буде відповідальною за створення базового скалярного поля. Дані класи визначають форму та висоту рельєфу, на основі якого буде побудована полігональна сітка у наступних етапах. Робота з шумами організовується через низку класів, які забезпечують різні методи генерації випадкових значень (рис. 3.2).

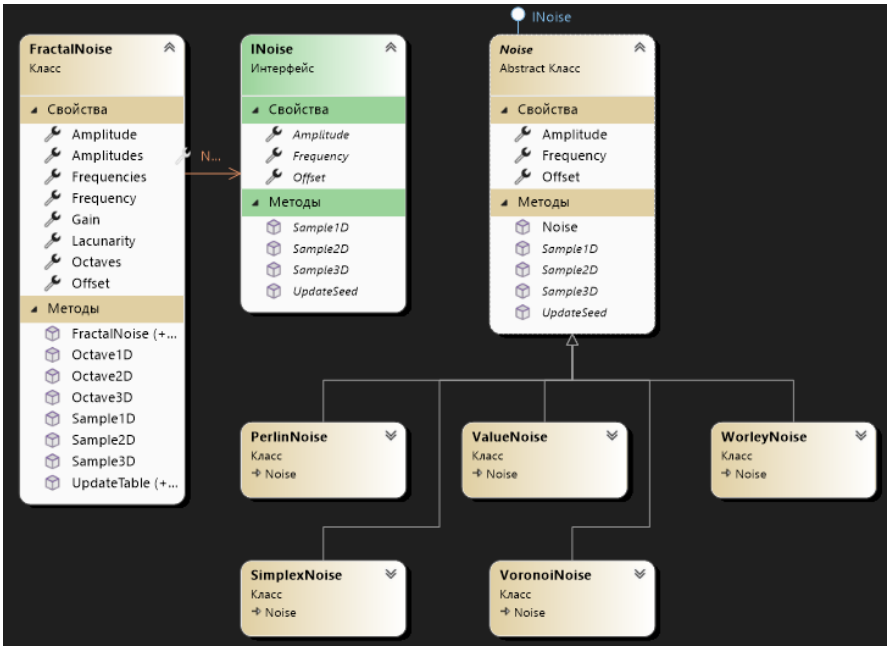


Рисунок 3.2 – Класи, що відповідають за генерацію шуму

Перший крок – створення інтерфейсу INoise, який визначає базовий набір функціональних можливостей для генерації процедурного шуму у тривимірному просторі. Він розроблений таким чином, щоб забезпечити єдиний підхід до створення різних алгоритмів шуму, таких як Perlin Noise, Simplex Noise та інші. Це дозволяє дотримуватися принципів поліморфізму та

інкапсуляції, що покращує гнучкість і розширюваність системи. Даний інтерфейс має 3 ключові параметри:

- Frequency, властивість, яка задає частоту шуму, тобто кількість повторень патерну в заданому просторі. Висока частота призводить до дрібніших деталей у згенерованому ландшафті, тоді як низька частота створює більш плавні та великі структури.
- Amplitude – визначає амплітуду (інтенсивність) значень шуму. Вона контролює масштаб зміни висоти рельєфу, де більші значення створюють більш контрастні висоти і глибини.
- Offset (Зсув) – Властивість Vector3, що дозволяє змістити шум у просторі на певний вектор.

Крім цього інтерфейс реалізує функції, щоб отримати значення шуму для різних наборів осей у просторі (рис. 3.3):

```
Ссылок: 8
float Sample1D(float x);

Ссылок: 8
float Sample2D(float x, float y);

Ссылок: 8
float Sample3D(float x, float y, float z);
```

Рисунок 3.3 – Реалізація функцій, для отримання значень шуму

Методи Sample1D(float x), Sample2D(float x, float y) та Sample3D(float x, float y, float z) в інтерфейсі INoise виконують вибірку значень шуму у просторі різної розмірності, забезпечуючи генерацію випадкових даних для різних цілей. Метод Sample1D використовується для одновимірних ефектів, таких як генерація лінійних висотних карт. Sample2D дозволяє працювати з двовимірними площинами, що є типовим для створення висотних карт ландшафтів, текстур або карт рельєфу. Sample3D розширює можливості до

тривимірного простору, що особливо корисно для воксельних світів, печерних систем або симуляцій об'ємних структур, де значення шуму змінюється у всіх трьох координатах. Усі ці методи приймають координати у відповідному вимірі та повертають числове значення, що відображає інтенсивність шуму в даній точці.

Реалізація методу UpdateSeed(int seed), дає можливість в похідних класах, зберегти конкретну генерацію ландшафту за числовим значенням, або для ефективного тестування алгоритму.

```
Ссылок: 6
public abstract class Noise : INoise
{

    Ссылок: 36
    public float Frequency { get; set; }

    Ссылок: 21
    public float Amplitude { get; set; }

    Ссылок: 36
    public Vector3 Offset { get; set; }

    Ссылок: 0
    public Noise()
    {

    }
    Ссылок: 8
    public abstract float Sample1D(float x);

    Ссылок: 8
    public abstract float Sample2D(float x, float y);

    Ссылок: 8
    public abstract float Sample3D(float x, float y, float z);

    Ссылок: 6
    public abstract void UpdateSeed(int seed);

}
```

Рисунок 3.4 – Клас Noise

Абстрактний клас Noise (рис. 3.4) забезпечує базову структуру для створення різних типів процедурного шуму, використовуючи принципи абстракції та наслідування. Він реалізує інтерфейс INoise та містить основні властивості для управління параметрами шуму: Frequency, Amplitude, Offset.

Конструктор Noise() є порожнім, оскільки об'єкт цього класу безпосередньо не створюється — він лише задає основу для дочірніх класів. Основні методи Sample1D(float x), Sample2D(float x, float y) та Sample3D(float x, float y, float z) оголошені як абстрактні, що означає, що кожен клас-нащадок

повинен реалізувати їх відповідно до конкретного алгоритму шуму. Це дозволяє створювати різні варіанти шуму (наприклад, Perlin, Worley, Simplex) з однаковою структурою.

Метод UpdateSeed(int seed) також оголошений як абстрактний для того, щоб кожен алгоритм міг реалізувати власний механізм ініціалізації, що забезпечує контрольованість і повторюваність результатів.

Клас PerlinNoise (рис. 3.5) реалізує алгоритм Перліна для генерації процедурного шуму. Цей алгоритм, є одним із найпопулярніших методів генерації згладжених випадкових значень, які часто використовуються для створення висотних карт, текстур та процедурних ландшафтів.

```
public class PerlinNoise : Noise
{

    Ссылок: 16
    private PermutationTable Perm { get; set; }

    Ссылок: 3
    public PerlinNoise(int seed, float frequency, float amplitude = 1.0f)
    {

        Frequency = frequency;
        Amplitude = amplitude;
        Offset = Vector3.zero;

        Perm = new PermutationTable(1024, 255, seed);
    }

    Ссылок: 2
    public override void UpdateSeed(int seed)
    {
        Perm.Build(seed);
    }

    Ссылок: 4
    public override float Sample1D( float x )
    {
        x = (x + Offset.x) * Frequency;
```

Рисунок 3.5 – Клас PerlinNoise

Клас PerlinNoise успадковує базовий абстрактний клас Noise та реалізує всі абстрактні методи, зокрема Sample1D(), Sample2D() і Sample3D() для вибірки шуму у просторах різної розмірності. Його завдання — обчислювати значення шуму для заданих координат, застосовуючи алгоритм Перліна, який створює плавні градієнтні переходи між точками.

Конструктор ініціалізує параметри шуму, зокрема Frequency, Amplitude та Offset. Створює об'єкт PermutationTable для керування випадковими значеннями та контролю повторюваності через seed.

Впровадження абстрактних методів має наступний вигляд: Sample1D(float x) – вибірка у 1D-просторі. Алгоритм обчислює значення шуму для однієї координати, визначаючи цілісну (ix0) та дробову (fx0) частини координати. Потім обчислюється згладжене інтерпольоване значення шуму між двома сусідніми точками. Sample2D(float x, float y) – виконує генерацію вибірку у 2D-просторі.

Sample3D (рис. 3.6) – працює у 3D-просторі. Використовується розширена версія алгоритму, яка враховує вісім сусідніх точок у кубічному просторі. Значення обчислюються через багатоступеневу інтерполяцію (спочатку по z, потім по y і x).

```
public override float Sample3D( float x, float y, float z )
{
    x = (x + Offset.x) * Frequency;
    y = (y + Offset.y) * Frequency;
    z = (z + Offset.z) * Frequency;
    int ix0, iy0, iz0;
    float fx0, fy0, fz0, fx1, fy1, fz1;
    float s, t, r;
    float nxy0, nxy1, nx0, nx1, n0, n1;
    ix0 = (int)Mathf.Floor(x);
    iy0 = (int)Mathf.Floor(y);
    iz0 = (int)Mathf.Floor(z);
    fx0 = x - ix0;
    fy0 = y - iy0;
    fz0 = z - iz0;
    fx1 = fx0 - 1.0f;
    fy1 = fy0 - 1.0f;
    fz1 = fz0 - 1.0f;
    r = FADE( fz0 );
    t = FADE( fy0 );
    s = FADE( fx0 );
    nxy0 = Grad(Perm[ix0, iy0, iz0], fx0, fy0, fz0);
    nxy1 = Grad(Perm[ix0, iy0, iz0 + 1], fx0, fy0, fz1);
    nx0 = LERP( r, nxy0, nxy1 );

    nxy0 = Grad(Perm[ix0, iy0 + 1, iz0], fx0, fy1, fz0);
    nxy1 = Grad(Perm[ix0, iy0 + 1, iz0 + 1], fx0, fy1, fz1);
    nx1 = LERP( r, nxy0, nxy1 );

    n0 = LERP( t, nx0, nx1 );

    nxy0 = Grad(Perm[ix0 + 1, iy0, iz0], fx1, fy0, fz0);
    nxy1 = Grad(Perm[ix0 + 1, iy0, iz0 + 1], fx1, fy0, fz1);
    nx0 = LERP( r, nxy0, nxy1 );

    nxy0 = Grad(Perm[ix0 + 1, iy0 + 1, iz0], fx1, fy1, fz0);
    nxy1 = Grad(Perm[ix0 + 1, iy0 + 1, iz0 + 1], fx1, fy1, fz1);
    nx1 = LERP( r, nxy0, nxy1 );
    n1 = LERP( t, nx0, nx1 );
```

Рисунок 3.6 – Функція Sample3D

Також в класі використовується допоміжні методи серед яких функція згладжування, що використовує поліноміальну формулу. Вона

використовується для створення плавних переходів між значеннями шуму. Також використовується лінійна інтерполяція між двома значеннями a та b: private float LERP(float t, float a, float b) return a + t * (b - a).

Методи Grad (рис. 3.7) у класі PerlinNoise відповідають за обчислення градієнтів у просторах різної розмірності та використовуються для створення випадкових напрямків, що впливають на формування значень шуму. Градієнт є ключовим елементом алгоритму Перліна, оскільки визначає напрямок та інтенсивність зміни значення шуму між точками вибірки. Усі версії методу отримують випадковий hash та координати точки, після чого обирають напрямок зміщення, ґрунтуючись на маніпуляціях з бітами хешу.

У 1D просторі метод лише повертає значення, помножене на випадковий знак. У інших варіантах функції використовуються різні комбінації координат (x, y, z, t), що забезпечує складніші патерни шуму. Градієнтний напрямок змінюється залежно від молодших бітів хешу, а результати масштабуються відповідно до відстані від базової точки. Це дозволяє створювати плавні переходи між областями з різними значеннями шуму, забезпечуючи реалістичність процедурних текстур та ландшафтів.

```
Ссылок: 2
private float Grad(int hash, float x)
{
    int h = hash & 15;
    float grad = 1.0f + (h & 7);
    if ((h & 8) != 0) grad = -grad;
    return (grad * x);
}

Ссылок: 4
private float Grad(int hash, float x, float y)
{
    int h = hash & 7;
    float u = h < 4 ? x : y;
    float v = h < 4 ? y : x;
    return ((h & 1) != 0 ? -u : u) + ((h & 2) != 0 ? -2.0f * v : 2.0f * v);
}

Ссылок: 8
private float Grad(int hash, float x, float y, float z)
{
    int h = hash & 15;
    float u = h < 8 ? x : y;
    float v = h < 4 ? y : h == 12 || h == 14 ? x : z;
    return ((h & 1) != 0 ? -u : u) + ((h & 2) != 0 ? -v : v);
}

Ссылок: 0
private float Grad(int hash, float x, float y, float z, float t)
{
    int h = hash & 31;
    float u = h < 24 ? x : y;
    float v = h < 16 ? y : z;
    float w = h < 8 ? z : t;
    return ((h & 1) != 0 ? -u : u) + ((h & 2) != 0 ? -v : v) + ((h & 4) != 0 ? -w : w);
}
```

Рисунок 3.7 – Функції Grad()

Наступний клас ValueNoise, що реалізує алгоритм процедурного шуму, відомий як Value Noise (шум на основі значень). Цей алгоритм є спрощеною альтернативою класичного шуму Перліна, де замість градієнтів використовується інтерполяція випадкових значень.

Клас ValueNoise успадковує базовий клас Noise та реалізує інтерфейс INoise. Основним завданням цього класу є генерація шуму шляхом інтерполяції між випадковими значеннями, отриманими з таблиці перестановок PermutationTable.

Ключова відмінність від PerlinNoise полягає в тому, що ValueNoise працює з випадковими значеннями у вузлах сітки, а не з градієнтами. Це робить його менш детальним, але водночас трохи швидшим у обчисленнях.

Конструктор даного класу створює об'єкт PermutationTable для зберігання випадкових значень із використанням параметру seed.

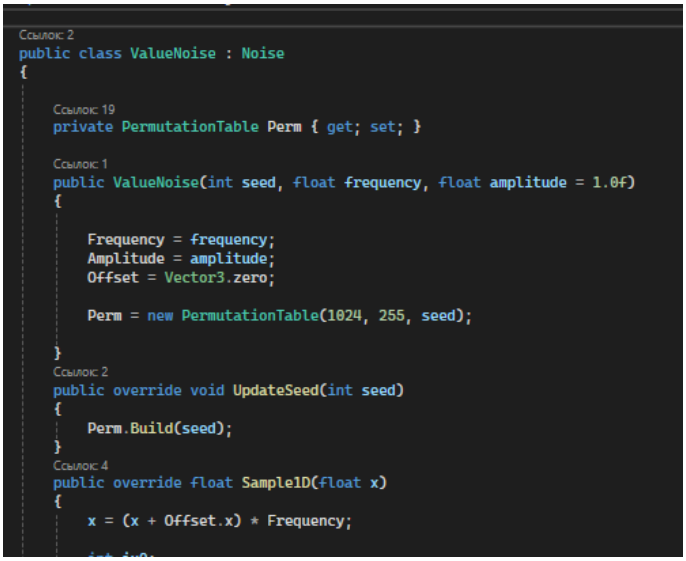


Рисунок 3.8 – Клас ValueNoise

Методи Sample1D(float x), Sample2D(float x, float y) та Sample3D(float x, float y, float z) у класі ValueNoise виконується алгоритм шляхом поділу простору на регулярну сітку, де кожен вузол містить випадкове значення з таблиці перестановок PermutationTable.

При виклику методу Sample, для поточної координати визначаються її цілочисельна та дробова частини, після чого відбувається інтерполяція між значеннями у сусідніх вузлах сітки. Для зменшення графічних артефактів використовується функція згладжування FADE, яка застосовується до дробової частини координат перед інтерполяцією.

```
Ссылок: 4
public override float Sample1D(float x)
{
    x = (x + Offset.x) * Frequency;
    int ix0;
    float fx0;
    float s, n0, n1;
    ix0 = (int)Mathf.Floor(x);
    fx0 = x - ix0;
    s = FADE(fx0);
    n0 = Perm[ix0];
    n1 = Perm[ix0 + 1];
    float n = LERP(s, n0, n1) * Perm.Inverse;
    n = n * 2.0f - 1.0f;

    return n * Amplitude;
}
Ссылок: 4
public override float Sample2D(float x, float y)
{
    x = (x + Offset.x) * Frequency;
    y = (y + Offset.y) * Frequency;

    int ix0, iy0;
    float fx0, fy0, s, t, nx0, nx1, n0, n1;

    ix0 = (int)Mathf.Floor(x);
    iy0 = (int)Mathf.Floor(y);
    fx0 = x - ix0;
    fy0 = y - iy0;
    t = FADE(fy0);
    s = FADE(fx0);
    nx0 = Perm[ix0, iy0];
    nx1 = Perm[ix0, iy0 + 1];
    n0 = LERP(t, nx0, nx1);
    nx0 = Perm[ix0 + 1, iy0];
    nx1 = Perm[ix0 + 1, iy0 + 1];
    n1 = LERP(t, nx0, nx1);
```

Рисунок 3.9 – Методи Sample класу ValueNoise

Основна відмінність між ValueNoise та PerlinNoise полягає у способі вибірки значень у вузлах сітки. У ValueNoise для кожного вузла зберігається випадкове значення, яке інтерполюється між сусідніми вузлами, тоді як PerlinNoise використовує градієнти — вектори напрямку, що впливають на значення шуму. Це робить PerlinNoise більш плавним і деталізованим, тоді як ValueNoise є простішим, але може виглядати менш природно через дискретність випадкових значень у вузлах. Value Noise також має нижчу обчислювальну складність, оскільки не потребує обчислення векторних градієнтів.

Клас PermutationTable (рис. 3.10) призначений для створення таблиці перестановок, яка використовується в алгоритмах процедурного шуму для генерації псевдовипадкових значень. Конструктор PermutationTable(int size, int max, int seed) приймає три параметри: size (розмір таблиці), max (максимальне значення для нормалізації) та seed (число для генерації випадкових значень). Поля Size, Max та Seed ініціалізуються значеннями з параметрів, тоді як Wrap використовується для обмеження індексації при доступі до елементів таблиці. Поле Inverse обчислюється як обернене значення Max і використовується для нормалізації значень у діапазоні [0, 1].

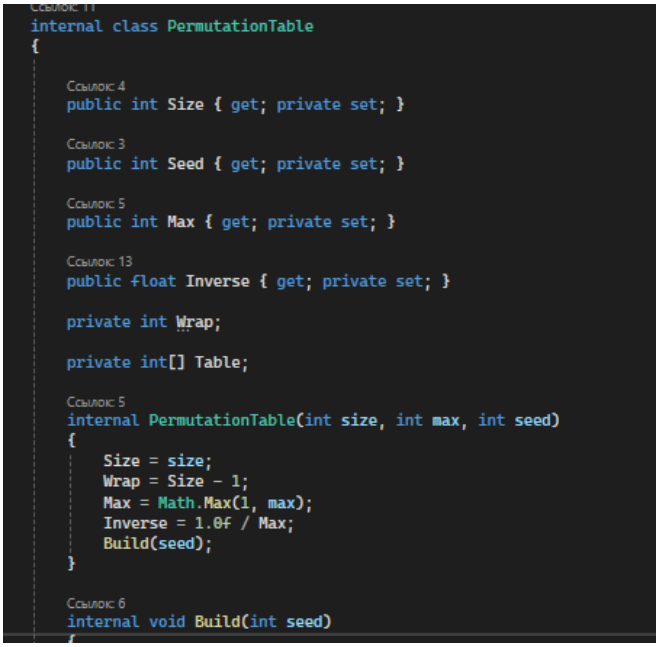


Рисунок 3.10 – Клас PermutationTable

Метод Build (рис. 3.11) відповідає за ініціалізацію таблиці перестановок з новим параметром seed. Він створює новий масив Table розміром Size та заповнює його випадковими значеннями за допомогою стандартного генератора випадкових чисел System.Random. У класі реалізовані індексатори (this[int i], this[int i, int j], this[int i, int j, int k]), які дозволяють отримувати випадкові значення для одновимірних, двовимірних та тривимірних координат. Вони працюють за принципом вкладених обчислень, де значення j

та k додаються до попередніх результатів із використанням операцій для обмеження виходу за межі таблиці (Wrap).

Клас забезпечує унікальні випадкові значення на основі переданого параметру (seed), що дозволяє створювати або відтворювані патерни шуму. Цей підхід важливий для контролю над генерацією ландшафтів, текстур та інших процедурних елементів, оскільки однаковий seed завжди дає однаковий результат, що необхідно для тестування та відтворення сцен.

```
Ссылок 6
internal void Build(int seed)
{
    if (Seed == seed && Table != null) return;

    Seed = seed;
    Table = new int[Size];

    System.Random rnd = new System.Random(Seed);

    for(int i = 0; i < Size; i++)
    {
        Table[i] = rnd.Next();
    }
}

Ссылок 32
```

Рисунок 3.11 – Метод Build

Клас VoronoiNoise реалізує алгоритм генерації процедурного шуму (Voronoi Diagrams). В основі VoronoiNoise лежить концепція діаграм Вороного, де простір розбивається на комірки, а для кожної точки обчислюється відстань до найближчих випадково розташованих контрольних точок (feature points). Це призводить до утворення клітинних структур із чіткими межами між областями, що робить його ідеальним для симуляції природних тріщин, кластерних утворень або біологічних текстур.

Головна відмінність VoronoiNoise у тому, що він працює з дискретними осередками та відстанями до контрольних точок. Це робить VoronoiNoise придатним для створення чітких клітинних текстур, тоді як PerlinNoise та ValueNoise краще підходять для плавних органічних форм.

У конструкторі (рис. 3.12) задаються початкові значення параметрів: Frequency, Amplitude, Offset, Distance та Combination. Таблиця перестановок ініціалізується з використанням заданого seed для контролю відтворюваності результатів.

```
namespace ProceduralNoiseProject
{

    Ссылок: 22
    public enum VORONOI_DISTANCE { EUCLIDIAN, MANHATTAN, CHEBYSHEV };

    Ссылок: 10
    public enum VORONOI_COMBINATION { D0, D1_D0, D2_D0 };

    Ссылок: 2
    public class VoronoiNoise : Noise
    {

        Ссылок: 4
        public VORONOI_DISTANCE Distance { get; set; }

        Ссылок: 2
        public VORONOI_COMBINATION Combination { get; set; }

        Ссылок: 21
        private PermutationTable Perm { get; set; }

        Ссылок: 1
        public VoronoiNoise(int seed, float frequency, float amplitude = 1.0f)
        {

            Frequency = frequency;
            Amplitude = amplitude;
            Offset = Vector3.zero;

            Distance = VORONOI_DISTANCE.EUCLIDIAN;
            Combination = VORONOI_COMBINATION.D1_D0;

            Perm = new PermutationTable(1024, int.MaxValue, seed);

        }
```

Рисунок 3.12 – Клас VoronoiNoise

Методи класу VoronoiNoise відрізняються від методів у PerlinNoise та ValueNoise насамперед принципом генерації шуму та підходом до обчислення значень.

Метод Distance1(), Distance2() та Distance3() використовуються для обчислення різних типів відстаней між точками (Euclidean, Manhattan, Chebyshev), що дозволяє контролювати форму шуму.

Метод Insert(Vector3 arr, float value) реалізує сортування масиву та зберігання найближчих контрольних точок за принципом вставки, допомагаючи ефективно знаходити найближчі точки до заданої координати вибірки.

ProbLookup(float value) відповідає за визначення кількості випадкових точок у кожному кубі простору, використовуючи розподіл Пуассона, що додає варіативності шуму.

Метод Combine (рис. 3.13) об'єднує результати обчислення відстаней залежно від вибраного режиму комбінації (D0, D1_D0, D2_D0), що впливає на деталізацію та контрастність згенерованих текстур. Ці методи разом забезпечують унікальну клітинну структуру шуму VoronoiNoise, відмінну від більш плавних та безперервних текстур, створюваних PerlinNoise та ValueNoise.

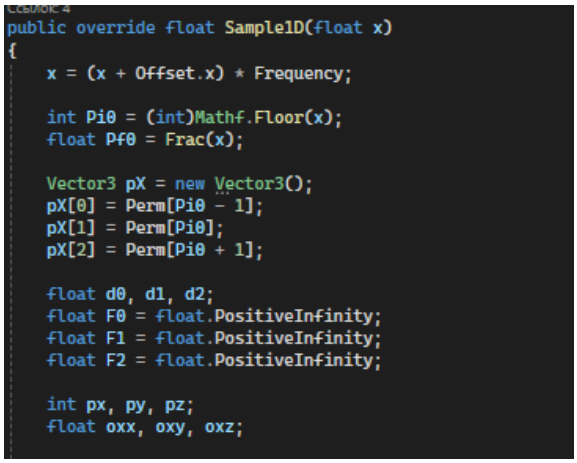
```
Ссылок: 3
private float Combine(Vector3 arr)
{
    switch(Combination)
    {
        case VORONOI_COMBINATION.D0:
            return arr[0];

        case VORONOI_COMBINATION.D1_D0:
            return arr[1] - arr[0];

        case VORONOI_COMBINATION.D2_D0:
            return arr[2] - arr[0];
    }

    return 0;
}
```

Рисунок 3.13 – Метод Combine

Наступний клас WorleyNoise, формує алгоритм процедурного шуму на основі діаграм Ворлі, який генерує текстури з характерними клітинними структурами, що часто використовуються для моделювання природних текстур, таких як осередки в камені, текстури тріщин або органічні поверхні, це робить схожим на алгоритм Вороного.

Основним елементом алгоритму є пошук найближчих контрольних точок у кожному осередку та обчислення відстаней до них за допомогою різних метрик (Euclidean, Manhattan, Chebyshev). Основний метод вибірки Sample (рис. 3.14) розбиває простір на регулярні кубічні осередки, після чого

для кожного осередку визначаються випадкові контрольні точки, координати яких зміщені за допомогою параметра Jitter.



```
public override float Sample1D(float x)
{
    x = (x + Offset.x) * Frequency;

    int Pi0 = (int)Mathf.Floor(x);
    float Pf0 = Frac(x);

    Vector3 pX = new Vector3();
    pX[0] = Perm[Pi0 - 1];
    pX[1] = Perm[Pi0];
    pX[2] = Perm[Pi0 + 1];

    float d0, d1, d2;
    float F0 = float.PositiveInfinity;
    float F1 = float.PositiveInfinity;
    float F2 = float.PositiveInfinity;

    int px, py, pz;
    float oxx, oxy, oxz;
```

Рисунок 3.14 – Метод Sample

Для знаходження найближчих точок використовується метод порівняння відстаней, після чого застосовується метод Combine(), який дозволяє комбінувати результати за кількома підходами (D0, D1_D0, D2_D0).

Унікальними особливостями цього класу є використання фіксованих коефіцієнтів K та Ko для нормалізації значень та згладження шуму, а також використання спеціальних допоміжних методів Frac та Mod (рис. 3.15) для обчислення дробових частин координат. WorleyNoise ідеально підходить для створення як абстрактних текстур, так і текстур для симуляції природних поверхонь, забезпечуючи широкий контроль над формою та деталізацією шуму за допомогою параметрів Frequency, Jitter, Distance та Combination.

```
Ссылок 6
private float Mod(float x, float y)
{
    return x - y * Mathf.Floor(x / y);
}

Ссылок 18
private float Frac(float v)
{
    return v - Mathf.Floor(v);
}
```

Рисунок 3.15 – Допоміжні методи Frac() та Mod()

Алгоритми Вороного (Voronoi Diagram) та Ворлі (Worley Noise) базуються на розбитті простору на області, але мають різні принципи роботи та застосування. Діаграма Вороного ділить простір на чіткі полігональні області, де кожна точка належить найближчому контрольному центру, створюючи дискретні межі, що використовуються для картографії, геометричних обчислень та аналізу даних.

Шум Ворлі, навпаки, генерує випадкові точки всередині осередків сітки та обчислює відстані до найближчих контрольних точок, створюючи градієнтний шум із плавними переходами. Він застосовується для процедурної генерації текстур, таких як тріщини, клітинні структури та природні поверхні. Основна відмінність полягає в тому, що Вороного створює статичне розбиття простору, тоді як Ворлі використовується для динамічного шуму з випадковими точками, що підходить для текстуризації та симуляції органічних поверхонь.

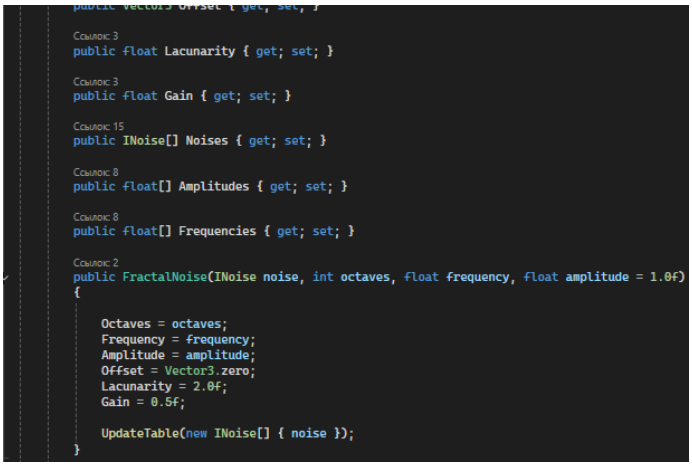


Рисунок 3.16 – Клас FractalNoise

Клас FractalNoise створюється алгоритм фрактального шуму, який генерує складні текстури та рельєфи шляхом комбінування декількох шарів шуму із різними частотами (Frequency) та амплітудами (Amplitude). Основний принцип алгоритму полягає у накладанні декількох рівнів базового шуму (INoise) з різною деталізацією, де кожна наступна ітерація має вищу частоту, але зменшену амплітуду, контрольовану параметрами Lacunarity (збільшення частоти) та Gain (зменшення амплітуди).

Відмінною особливістю FractalNoise є можливість об'єднання декількох джерел шуму (INoise[] Noises), що дозволяє змішувати різні алгоритми шуму для створення складних текстур. Методи Sample1D(), Sample2D() та Sample3D() обчислюють підсумковий результат шляхом сумування усіх значень, використовуючи збережені таблиці Frequencies та Amplitudes. Гнучкість алгоритму забезпечується через можливість змінювати кількість параметру (Octaves) та оновлювати параметри шуму за допомогою методу UpdateTable. Цей підхід дозволяє створювати реалістичні ландшафти та текстури з різним рівнем деталізації, роблячи FractalNoise ідеальним для процедурної генерації складних природних форм у комп'ютерній графіці.

```
Ссылок 3
protected virtual void UpdateTable(INoise[] noises)
{
    Amplitudes = new float[Octaves];
    Frequencies = new float[Octaves];
    Noises = new INoise[Octaves];

    int numNoises = noises.Length;

    float amp = 0.5f;
    float frq = Frequency;
    for(int i = 0; i < Octaves; i++)
    {
        Noises[i] = noises[Math.Min(i, numNoises - 1)];
        Frequencies[i] = frq;
        Amplitudes[i] = amp;
        amp *= Gain;
        frq *= Lacunarity;
    }

}
```

Рисунок 3.17 – Метод UpdateTable

Наступний кроком потрібно розглянути класи, що відповідають за формування алгоритму MarchingCubes (рис. 3.18). Цей алгоритм є ключовим компонентом процедурної генерації тривимірних ландшафтів та об’ємних моделей, оскільки дозволяє конвертувати скалярні поля у полігональні сітки. Принцип роботи MarchingCubes базується на розділенні простору на вокселі (куби), де для кожного куба визначається, які його вершини знаходяться всередині або поза поверхнею на основі заданого порогового значення.

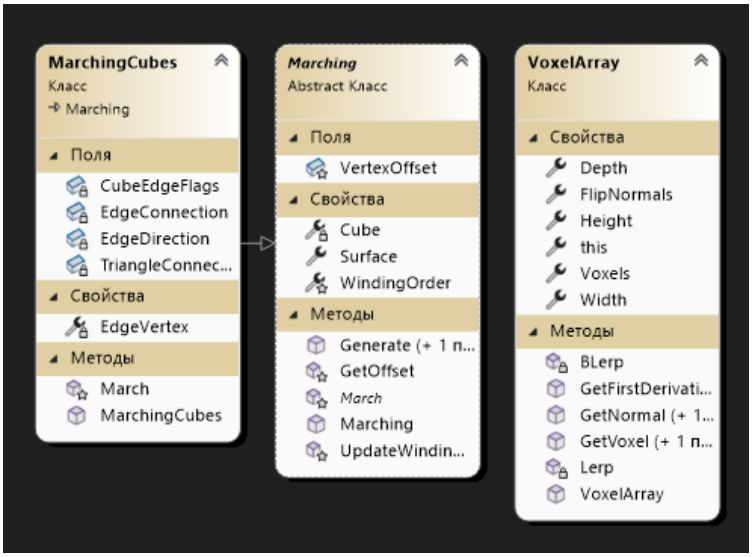


Рисунок 3.18 – Друга частина проекту (MarchingCubes)

Клас VoxelArray (рис. 3.19) є ключовим компонентом для зберігання та маніпулювання воксельними даними, необхідними для алгоритму MarchingCubes. Вокселі представляють собою тривимірний масив значень, де кожен елемент (float) зберігає щільність або скалярне значення, яке використовується для визначення поверхні всередині воксельної сітки.

Конструктор класу (рис. 3.19) приймає параметри розмірів width, height та depth для визначення розмірів тривимірного масиву Voxels та ініціалізує масив із значенням за замовчуванням. Властивості Width, Height, Depth забезпечують доступ до розмірів воксельного масиву, тоді як FlipNormals дозволяє інвертувати нормалі при обчисленні поверхні, що може бути корисним для роботи з об’ємними моделями, такими як печери або перевернуті ландшафти.

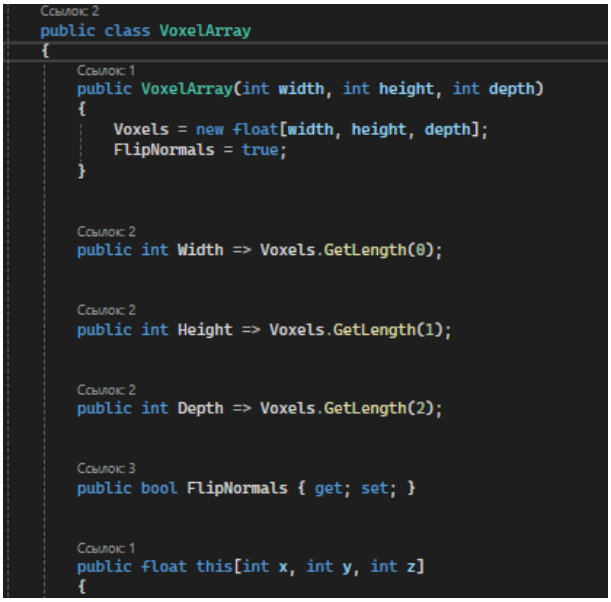
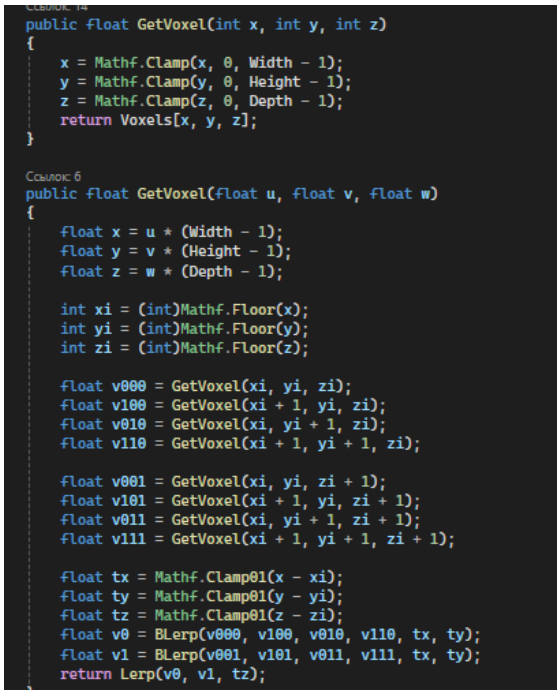


Рисунок 3.19 – Клас VoxelArray

Основними методами класу є доступ до значень вокселів та обчислення нормалей. Метод GetVoxel (рис. 3.20) забезпечує доступ до значень вокселів із автоматичним обмеженням координат у межах масиву (Clamp). Для більшої гнучкості метод GetVoxel(float u, float v, float w) дозволяє здійснювати вибірку вокселів у нормалізованих координатах (0-1), застосовуючи інтерполяцію для

обчислення значення між найближчими вузлами сітки. Це досягається шляхом обчислення вагових коефіцієнтів для сусідніх вокселів (tx, ty, tz) та комбінування їх за допомогою функцій Lerp (лінійна інтерполяція) та BLerp (білінійна інтерполяція). Такий підхід дозволяє більш плавно відображати поверхню під час генерації сітки.



```
public float GetVoxel(int x, int y, int z)
{
    x = Mathf.Clamp(x, 0, Width - 1);
    y = Mathf.Clamp(y, 0, Height - 1);
    z = Mathf.Clamp(z, 0, Depth - 1);
    return Voxels[x, y, z];
}

Ссылок: 6
public float GetVoxel(float u, float v, float w)
{
    float x = u * (Width - 1);
    float y = v * (Height - 1);
    float z = w * (Depth - 1);

    int xi = (int)Mathf.Floor(x);
    int yi = (int)Mathf.Floor(y);
    int zi = (int)Mathf.Floor(z);

    float v000 = GetVoxel(xi, yi, zi);
    float v100 = GetVoxel(xi + 1, yi, zi);
    float v010 = GetVoxel(xi, yi + 1, zi);
    float v110 = GetVoxel(xi + 1, yi + 1, zi);

    float v001 = GetVoxel(xi, yi, zi + 1);
    float v101 = GetVoxel(xi + 1, yi, zi + 1);
    float v011 = GetVoxel(xi, yi + 1, zi + 1);
    float v111 = GetVoxel(xi + 1, yi + 1, zi + 1);

    float tx = Mathf.Clamp01(x - xi);
    float ty = Mathf.Clamp01(y - yi);
    float tz = Mathf.Clamp01(z - zi);
    float v0 = BLerp(v000, v100, v010, v110, tx, ty);
    float v1 = BLerp(v001, v101, v011, v111, tx, ty);
    return Lerp(v0, v1, tz);
```

Рисунок 3.20 – Методи GetVoxel

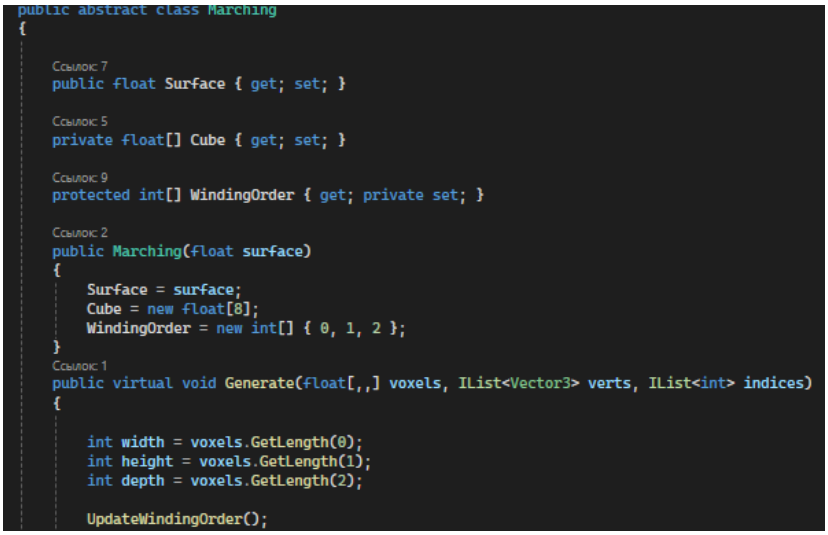
Ще однією важливою функціональністю класу є обчислення нормалей та градієнтів воксельної сітки. Метод GetNormal (рис. 3.21) повертає нормаль у вказаній точці, використовуючи метод GetFirstDerivative, який обчислює градієнт значень за допомогою центральних різниць (скінченні різниці). Це дозволяє визначити напрямок, у якому значення вокселів змінюється найшвидше, що є ключовим для коректного побудови нормалей у MarchingCubes. Існує також варіант цих методів для нормалізованих координат (GetNormal(float u, float v, float w) та GetFirstDerivative (рис. 3.21). Клас також підтримує обчислення нормалей у зворотному напрямку (FlipNormals), що може бути корисним при побудові внутрішніх об'ємів або перевернутих моделей.

```
Ссылок: 0
public Vector3 GetNormal(int x, int y, int z)
{
    var n = GetFirstDerivative(x, y, z);

    if (FlipNormals)
        return n.normalized * -1;
    else
        return n.normalized;
}
Ссылок: 1
public Vector3 GetNormal(float u, float v, float w)
{
    var n = GetFirstDerivative(u, v, w);

    if (FlipNormals)
        return n.normalized * -1;
    else
        return n.normalized;
}
Ссылок: 1
public Vector3 GetFirstDerivative(int x, int y, int z)
{
    float dx_pl = GetVoxel(x + 1, y, z);
    float dy_pl = GetVoxel(x, y + 1, z);
    float dz_pl = GetVoxel(x, y, z + 1);

    float dx_ml = GetVoxel(x - 1, y, z);
    float dy_ml = GetVoxel(x, y - 1, z);
    float dz_ml = GetVoxel(x, y, z - 1);

    float dx = (dx_pl - dx_ml) * 0.5f;
```

Рисунок 3.21 – Методи GetNormal, GetFirstDerivative

Таким чином, VoxelArray є критично важливим класом для зберігання даних про вокселі, виконання вибірки значень та обчислення нормалей, необхідних для алгоритму MarchingCubes.

Далі абстрактний базовий клас Marching (рис. 3.22), що використовується для побудови полігональних сіток на основі воксельних даних. Основне завдання цього класу — обробка масиву вокселів та перетворення його в набір вершин (verts) і індексів (indices), які формують полігональну сітку. Він має параметр поверхні (Surface), що визначає порогове значення, на основі якого відбувається відсікання. Якщо значення вокселя перевищує це порогове значення, вважається, що ця точка знаходиться всередині об'єкта, якщо ж менше — зовні.



```
public abstract class Marching
{

    Ссылок 7
    public float Surface { get; set; }

    Ссылок 5
    private float[] Cube { get; set; }

    Ссылок 9
    protected int[] WindingOrder { get; private set; }

    Ссылок 2
    public Marching(float surface)
    {
        Surface = surface;
        Cube = new float[8];
        WindingOrder = new int[] { 0, 1, 2 };
    }
    Ссылок 1
    public virtual void Generate(float[,,] voxels, IList<Vector3> verts, IList<int> indices)
    {

        int width = voxels.GetLength(0);
        int height = voxels.GetLength(1);
        int depth = voxels.GetLength(2);

        UpdateWindingOrder();
```

Рисунок 3.22 – Абстрактний клас

Ключові методи класу Marching зосереджені на генерації сітки з воксельного масиву через алгоритм Marching Cubes. Метод Generate (рис. 3.23) здійснює основний процес генерації сітки. Він проходить по всіх вокселях масиву, аналізуючи кожен куб (восьмикутний фрагмент воксельної сітки) та обчислюючи значення щільності у його вершинах. Для цього використовується масив Cube, який зберігає значення вокселів поточного куба. Після отримання цих значень метод March (абстрактний) викликається для подальшої обробки куба та визначення того, які трикутники необхідно створити для поточного набору даних. Існує також перевантажений метод Generate(IList<float> voxels, int width, int height, int depth, IList<Vector3> verts, IList<int> indices), який працює з одновимірним масивом для зберігання вокселів, забезпечуючи більш гнучкий підхід до зберігання даних.

```
public virtual void Generate(float[,,] voxels, IList<Vector3> verts, IList<int> indices)
{

    int width = voxels.GetLength(0);
    int height = voxels.GetLength(1);
    int depth = voxels.GetLength(2);

    UpdateWindingOrder();

    int x, y, z, i;
    int ix, iy, iz;
    for (x = 0; x < width - 1; x++)
    {
        for (y = 0; y < height - 1; y++)
        {
            for (z = 0; z < depth - 1; z++)
            {
                for (i = 0; i < 8; i++)
                {
                    ix = x + VertexOffset[i, 0];
                    iy = y + VertexOffset[i, 1];
                    iz = z + VertexOffset[i, 2];

                    Cube[i] = voxels[ix, iy, iz];
                }

                //Perform algorithm
                March(x, y, z, Cube, verts, indices);
            }
        }
    }
}
```

Рисунок 3.23 – Метод Generate

Клас містить метод UpdateWindingOrder, який відповідає за встановлення порядку перегляду вершин трикутників (WindingOrder). Якщо поверхня позитивна (Surface > 0), то порядок перегляду змінюється, щоб забезпечити правильний напрямок нормалей. Це важливо для правильного відображення полігональної сітки та уникнення артефактів рендерингу.

```
Ссылок: 2
protected virtual void UpdateWindingOrder()
{
    if (Surface > 0.0f)
    {
        WindingOrder[0] = 2;
        WindingOrder[1] = 1;
        WindingOrder[2] = 0;
    }
    else
    {
        WindingOrder[0] = 0;
        WindingOrder[1] = 1;
        WindingOrder[2] = 2;
    }
}
```

Рисунок 3.24 – Метод UpdateWindingOrder

Ще одним важливим елементом класу є статичний масив VertexOffset, який визначає відносні позиції 8 вершин куба у тривимірному просторі. Цей

масив використовується для ітерації по вершинах куба під час аналізу кожного вокселя. Метод GetOffset (рис. 3.25) обчислює точку перетину між двома значеннями щільності (v1 та v2) для коректного визначення позиції вершин трикутників. Це дозволяє створювати плавні поверхні між дискретними значеннями вокселів за допомогою лінійної інтерполяції.

```
Ссылок: 2
protected virtual float GetOffset(float v1, float v2)
{
    float delta = v2 - v1;
    return (delta == 0.0f) ? Surface : (Surface - v1) / delta;
}
```

Рисунок 3.25 – Метод GetOffset

Таким чином, Marching є фундаментальним класом для реалізації алгоритму Marching Cubes, надаючи інструменти для обробки воксельних даних, генерації полігональних сіток та управління параметрами поверхні. Його абстрактна структура дозволяє створювати підкласи для конкретних варіантів алгоритму, таких як Marching Tetrahedra або інші модифікації. Завдяки чіткій організації та методам для роботи з вокселями, клас є універсальним для задач процедурної генерації геометрії у тривимірному просторі.

Останній клас з якого починається завантаження ландшафту Example. Використовується для візуалізації процедурно згенерованих 3D-моделей з використанням алгоритму Marching Cubes. Його основна функція — генерувати тривимірні полігональні сітки на основі процедурного шуму (PerlinNoise, FractalNoise) і відображати їх у середовищі Unity. Клас містить публічні параметри для управління генерацією сітки, такі як sliderSpeed для регулювання швидкості обертання моделі, inputField для введення числового значення генератора шуму (seed).

```
Ссылок: 3
public enum MARCHING_MODE {  CUBES, TETRAHEDRON };

Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 0
public class Example : MonoBehaviour
{
    public Slider sliderSpeed;
    public InputField inputField;

    public Material material;

    public MARCHING_MODE mode = MARCHING_MODE.CUBES;

    public int seed = 0;

    public bool smoothNormals = false;

    public bool drawNormals = false;

    private List<GameObject> meshes = new List<GameObject>();

    private NormalRenderer normalRenderer;

    Сообщение Unity | Ссылок: 0
    void Start()
    {

    }

    Ссылок: 1
    public void GenerateMesh()
    {
        INoise perlin = new PerlinNoise(int.Parse(inputField.text), 1.0f);
        FractalNoise fractal = new FractalNoise(perlin, 3, 1.0f);
```

Рисунок 3.26 – Клас Example

Основний метод GenerateMesh (рис. 3.27) відповідає за генерацію процедурної 3D-моделі з використанням воксельних даних. Він спочатку створює екземпляр PerlinNoise на основі введеного числового значення та використовує його для ініціалізації FractalNoise. Далі створюється VoxelArray заданих розмірів (width, height, depth), і кожен воксель заповнюється значенням процедурного шуму, що генерує складний рельєф. На основі воксельних даних викликається метод marching.Generate(), який створює списки вершин (verts) та індексів (indices) для полігональної сітки. Якщо параметр smoothNormals увімкнено, обчислюються нормалі для кожної вершини, використовуючи метод GetNormal() із класу VoxelArray, що дозволяє зробити сітку візуально плавнішою.

```
Ссылок: 1
public void GenerateMesh()
{
    INoise perlin = new PerlinNoise(int.Parse(inputField.text), 1.0f);
    FractalNoise fractal = new FractalNoise(perlin, 3, 1.0f);

    Marching marching = null;
    if (mode == MARCHING_MODE.TETRAHEDRON)
        marching = new MarchingTertrahedron();
    else
        marching = new MarchingCubes();

    marching.Surface = 0.0f;


    int width = 96;
    int height = 12;
    int depth = 96;

    var voxels = new VoxelArray(width, height, depth);

    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            for (int z = 0; z < depth; z++)
            {
                float u = x / (width - 1.0f);
                float v = y / (height - 1.0f);
                float w = z / (depth - 1.0f);

                voxels[x, y, z] = fractal.Sample3D(u, v, w);
            }
```

Рисунок 3.27 – Метод GenerateMesh

Методи CreateMesh32 (рис. 3.28) та CreateMesh16 відповідають за створення та відображення тривимірної полігональної сітки в Unity, використовуючи стандартний компонент MeshFilter. CreateMesh32 створює сітку з використанням 32-бітної індексації, що дозволяє використовувати понад 65,000 вершин в одній сітці.

Метод CreateMesh16 використовується для створення великих моделей з обмеженням на 65,000 вершин, розбиваючи сітку на кілька менших об'єктів (GameObject). Обидва методи додають компоненти MeshFilter та MeshRenderer до об'єктів сцени та застосовують до них заданий матеріал (material).

Клас також містить методи для оновлення сцени та анімації обертання об'єкта. Метод Update() обертає модель навколо осі Y зі швидкістю, яка контролюється слайдером sliderSpeed. Метод Regenerate() дозволяє повторно згенерувати сітку при зміні параметрів генерації.

```
private void CreateMesh32(List<Vector3> verts, List<Vector3> normals, List<int> indices, Vector3 position)
{
    Mesh mesh = new Mesh();
    mesh.indexFormat = IndexFormat.UInt32;
    mesh.SetVertices(verts);
    mesh.SetTriangles(indices, 0);

    if (normals.Count > 0)
        mesh.SetNormals(normals);
    else
        mesh.RecalculateNormals();

    mesh.RecalculateBounds();

    GameObject go = new GameObject("Mesh");
    go.transform.parent = transform;
    go.AddComponent<MeshFilter>();
    go.AddComponent<MeshRenderer>();
    go.GetComponent<Renderer>().material = material;
    go.GetComponent<MeshFilter>().mesh = mesh;
    go.transform.localPosition = position;

    meshes.Add(go);
}
```

Рисунок 3.28 – Метод CreateMesh32

3.2 Тестування алгоритму

Наступним етапом є завантаження проекту в середовищі Unity для перевірки результатів генерації процедурних 3D-моделей. Для цього слід переконатися, що всі скрипти (Example, Marching, VoxelArray, PerlinNoise, FractalNoise тощо) коректно підключені до сцени.:

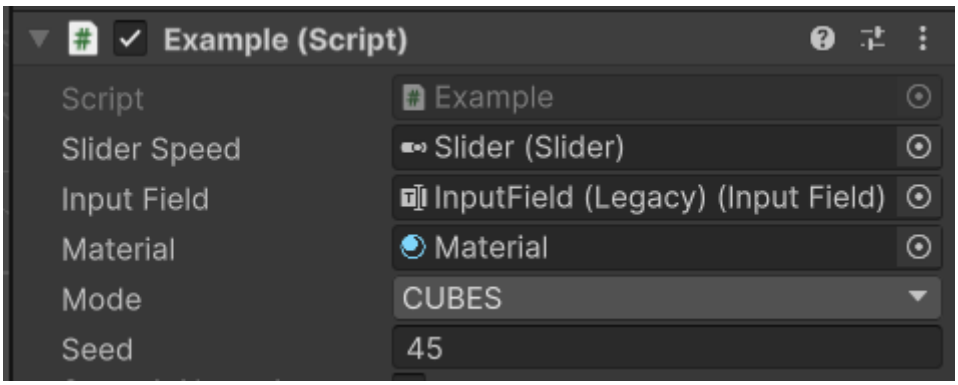


Рисунок 3.29 – Налаштування об’єкту сцени

Після цих дій завантажити проект. Користувач повинен спостерігати відповідний інтерфейс додатку:

Рисунок 3.30 – Інтерфейс для тестування алгоритму

Даний інтерфейс використовується для тестування алгоритму побудови ландшафту. Для роботи потрібно спочатку натиснути Regenerate, щоб з’явився відповідний воксельний ландшафт (рис. 3.31):

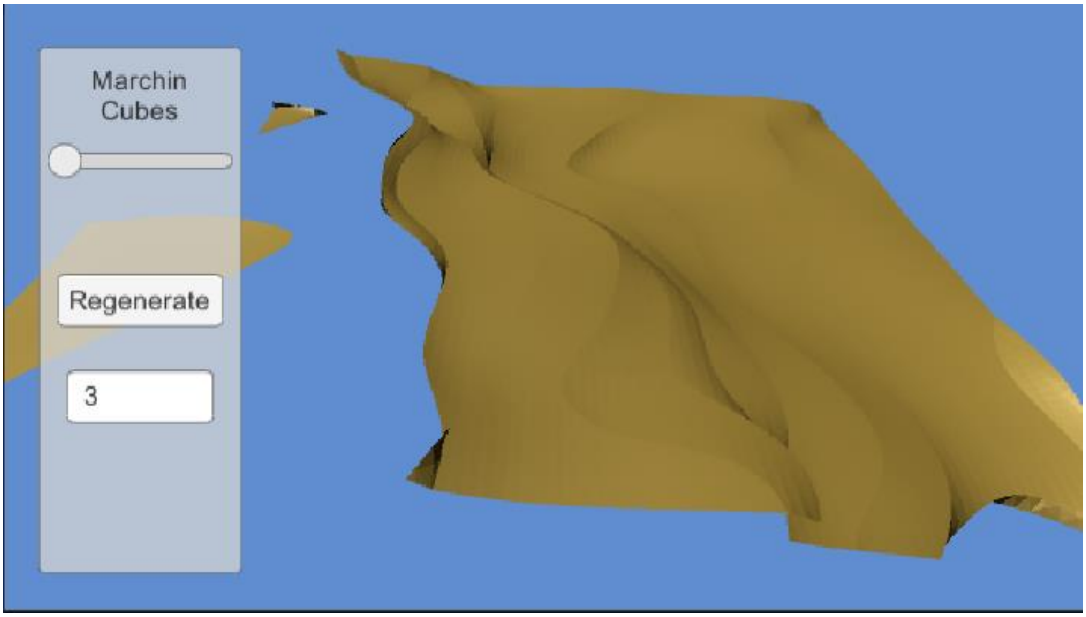


Рисунок 3.31 – Відображення ландшафту

Відповідно на екрані має з'явитися процедурно згенерована тривимірна модель, який можна розглянути з різних боків, використовуючи слайдер:

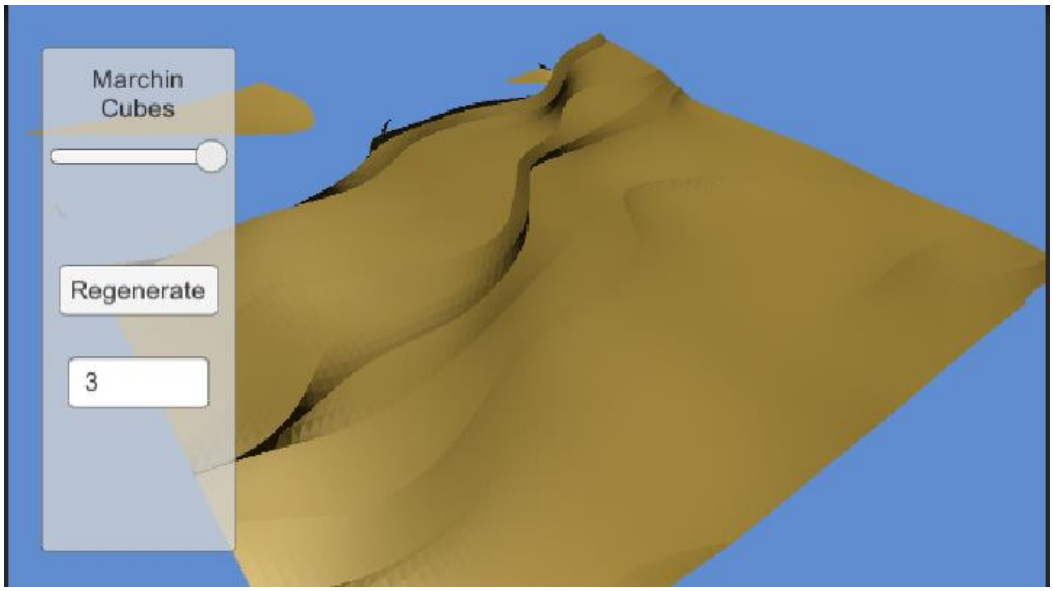


Рисунок 3.32 – Обертання моделі

Відповідно в залежності від положення слайдера становитиме швдкість обертання моделі, що додає можливість власноруч контролювати швидкість при якій буде розглядатися модель.

Крім цього однією з функцій також можна відмітити це покращення деталізації сітки за допомогою параметру SmoothNormals:

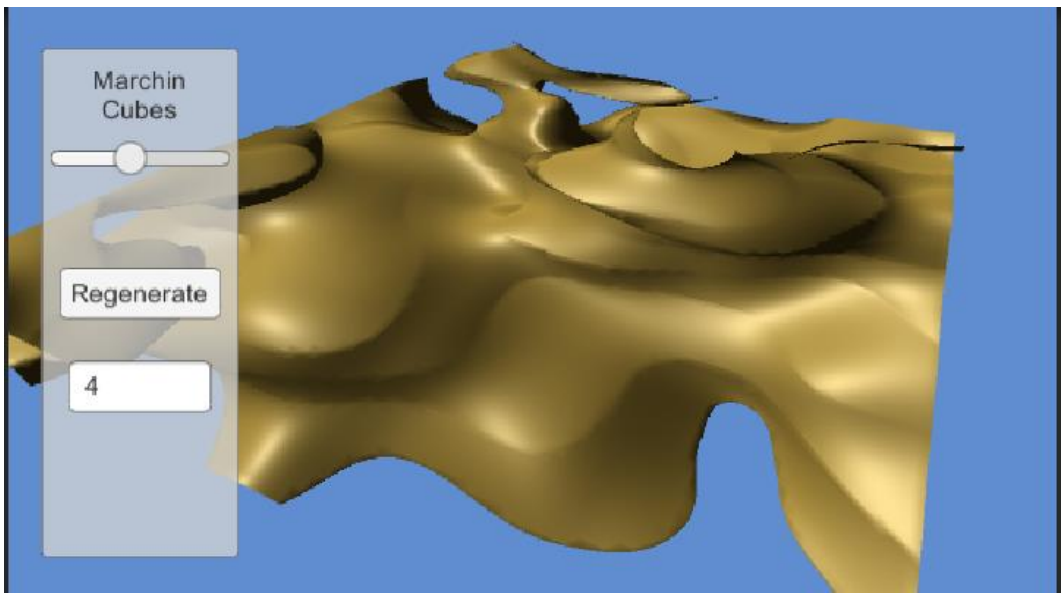


Рисунок 3.33 – Покращення деталізації ландшафту

Відповідно якщо, зняти параметр покращення, то можна спостерігати повернення в більш звичний вигляд ландшафту:

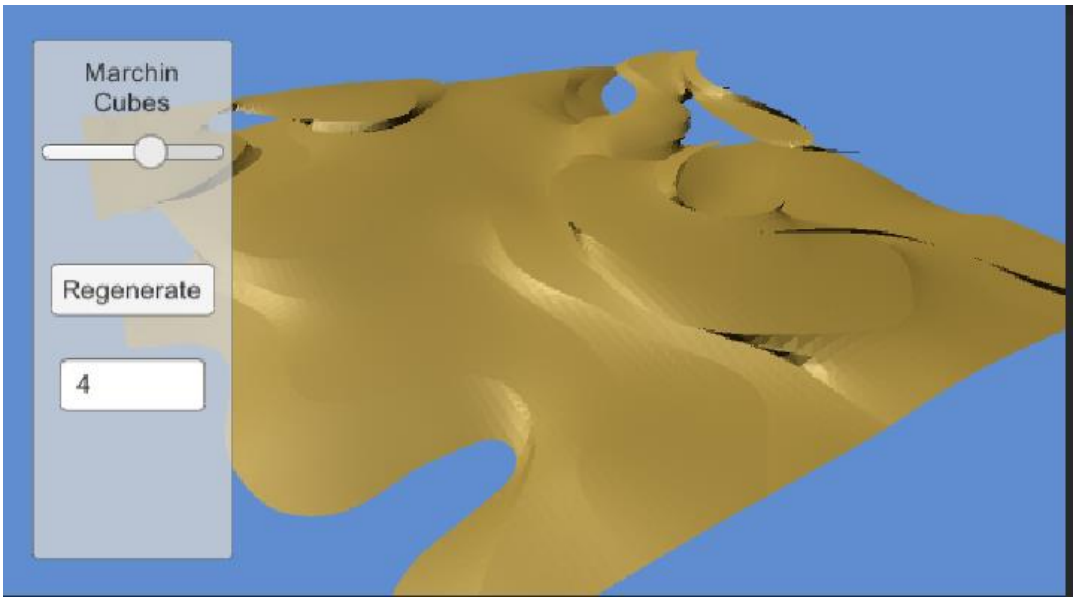


Рисунок 3.34 – Повернення зовнішнього вигляду ландшафту

3.5 Висновок до третього розділу

У третьому розділі кваліфікаційної роботи показано розроблений алгоритм та його програмна розробка алгоритму процедурної генерації ландшафту гри та його реалізація.

Проектування системи процедурної генерації ландшафтів поділяється на дві основні частини, кожна з яких відповідає за окремий етап обробки даних та формування кінцевого результату.

Щоб розпочати створення процедурного ландшафту, необхідно спочатку реалізувати систему для генерації шуму, яка буде відповідальною за створення базового скалярного поля. Дані класи визначають форму та висоту рельєфу, на основі якого буде побудована полігональна сітка у наступних етапах. Робота з шумами організовується через низку класів, які забезпечують різні методи генерації випадкових значень.

Реалізація методу UpdateSeed(int seed), дає можливість в похідних класах, зберегти конкретну генерацію ландшафту за числовим значенням, або для ефективного тестування алгоритму.

Результатом розробки є створений алгоритм процедурної генерації тривимірних моделей на основі алгоритму Marching Cubes з використанням різних методів шуму, таких як PerlinNoise, FractalNoise та WorleyNoise. Реалізований підхід дозволяє створювати складні об'ємні поверхні, включаючи природні ландшафти, органічні форми та текстури. Завдяки інтеграції з Unity, користувач отримує інтуїтивно зрозумілий інтерфейс для взаємодії з параметрами генерації. Загалом, реалізований алгоритм є потужним інструментом для створення процедурних 3D-моделей, що може бути використаний у відеоіграх, симуляторах та системах візуалізації даних.

ВИСНОВОК

У процесі виконання кваліфікаційної роботи на тему «Процедурна генерація ландшафтів на основі алгоритму Marching Cubes» було проведено комплексне дослідження методів процедурної генерації тривимірних ландшафтів, проаналізовано сучасні алгоритми та реалізовано програмний продукт для створення реалістичних сцен. Робота включала теоретичний аналіз методів генерації, розробку програмної реалізації та тестування створеного продукту.

Основними завданнями, які були вирішені під час виконання роботи, стали:

1. Проведення аналізу існуючих алгоритмів процедурної генерації ландшафтів, включаючи HeightMap, Marching Cubes, Marching Tetrahedra та інші.
2. Обґрунтування вибору алгоритму Marching Cubes як основи для генерації тривимірних поверхонь через його баланс між деталізацією та обчислювальною ефективністю.
3. Розробка алгоритму процедурної генерації ландшафтів із використанням шумів Перліна та Value Noise для створення реалістичних ландшафтів.
4. Програмна реалізація алгоритму у середовищі Unity з використанням C#, що забезпечило інтеграцію процедурної генерації у середовище розробки ігор та інтерактивних додатків.
5. Тестування програмного продукту для перевірки відповідності функціональним і нефункціональним вимогам, включаючи стабільність роботи, продуктивність та якість візуалізації.

Було проведено детальний аналіз інструментів для реалізації генерації ландшафтів, зокрема Unity, Autodesk Maya та Three.js. Вибір рушія Unity був обґрунтований його можливостями для роботи з воксельними даними, підтримкою C# та вбудованими засобами для тривимірного моделювання.

У процесі роботи були сформовані вимоги до програмного забезпечення:

- Функціональні вимоги: генерація процедурних ландшафтів із можливістю налаштування рівня деталізації, інтеграція алгоритму в Unity, підтримка реального часу оновлення сцени.
- Нефункціональні вимоги: мінімізація обчислювальних витрат, стабільна продуктивність, відсутність графічних артефактів.

Розроблений алгоритм було успішно протестовано у середовищі Unity, що підтвердило його ефективність для створення складних тривимірних ландшафтів із мінімальними обчислювальними ресурсами. Він може бути використаний у розробці відеоігор, симуляторів та інтерактивних додатків для візуалізації великих тривимірних сцен.

Таким чином, поставлена мета щодо розробки алгоритму процедурної генерації ландшафтів на основі Marching Cubes була досягнута, а отримані результати можуть бути використані для подальших досліджень у сфері комп'ютерної графіки та розробки віртуальних середовищ.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. F. Kenton Musgrave, Craig E. Kolb andRobert S. Mace: The Synthesis and Ren-dering of Eroded Fractal Terrains. ComputerGraphics, Volume 23, Number 3, July 1989. - 41-50p.

2. Plessinger P. Voxel terrain engine Archived 2013-11-13 at the Wayback Machine", introduction. In a coder's mind, 2005. - 22p.

3. A. Lenzhen, K. Rafi, J. Tao,The shadow of a Thurston geodesic to the curve graph. Journal of Topology, 2015. P.1085-1118.

4. Levenber G J. Fast view-dependent level-of-detail rendering using cached geometry. In: Proceedings of IEEE Visualization, Boston, USA: IEEE Press, 2002: 259-266. DOI: 10.1109/visual.2002.1183783

5. Ebert, D., Musgrave, K., Peachy, D., Perlin, K.,Androwely, S..Texturing& Modeling, a Procedural Approach, 3rd ed. Morgan Kaufmann/Elsevier, Amsterdam. 2003. - 74p

6. Golossary for Perlin noise – [Електронний ресурс]. – Режим доступу: http://libnoise.sourceforge.net/glossary/.

7. Animated low poly characters – [Електронний ресурс]. – Режим доступу: https://www.theseus.fi/bitstream/handle/10024/72726/Jolma_Valtteri.pdf

8. Procedural Generation – [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Procedural_generation

9. The Grounded Heightmap Tree – [Електронний ресурс]. – Режим доступу: https://upcommons.upc.edu/bitstream/handle/2117/86910/R08-30.pdf

10. Heightmap – [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Heightmap

11. Processing Irregular Grid – [Електронний ресурс]. – Режим доступу: https://medium.com/@mishaheesakkers/process-ing-generative-irregulargrid-8f0d712dfaa4

12. Tile-based video game : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Tile-based_video_game

13. Marching cubes : [Електронний ресурс]. Режим доступу: https://www.wikiwand.com/uk/Marching_cubes

14. Marching Tetrahedra : [Електронний ресурс]. Режим доступу: https://daac.hpc.mil/gettingStarted/Marching_Tetrahedra.html

15. Документація Unity3D [Електронний ресурс]. – Режим доступу: https://docs.unity.com

16. Maya Documentation – [Електронний ресурс]. – Режим доступу: https://www.autodesk.com/support/technical/article/caas/tsarticles/ts/lC3jaffqnWFyQoLPEPm7n.html

17. three.js – [Електронний ресурс]. – Режим доступу: https://threejs.org/docs/index.html