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

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 519.688

«До захисту допущено»
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2022 р.

Магістерська дисертація

на здобуття ступеня магістра

освітньо-науковою програмою
«Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Алгоритмічно-програмний метод процедурної генерації воксельного ландшафту із 3D-тайлів»

Виконав:
студент ІІ курсу, групи КП-01мн
Брусенцов Юрій Олексійович _____

Керівник:
Завідувачка кафедри ПЗКС, д.т.н., доцент,
Сулема Євгенія Станіславівна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Завідувачка кафедри ММСА ІПСА, к.т.н., доцент,
Тимощук Оксана Леонідівна _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ Іван ДИЧКА

«___»_____2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Брусенцову Юрію Олексійовичу

1. Тема дисертації «Алгоритмічно-програмний метод процедурної генерації воксельного ландшафту із 3D-тайлів», науковий керівник дисертації Сулема Євгенія Станіславівна, д.т.н., доцент, затверджені наказом по університету від «05» травня 2022 р. № НС/201/2022

2. Термін подання студентом дисертації «13» травня 2022 р.

3. Об’єкт дослідження: процес процедурної генерації воксельного ландшафту з 3D-тайлів.

4. Предмет дослідження: методи, алгоритми та програмне забезпечення для генерації воксельного ландшафту.

5. Перелік завдань, які потрібно розробити:
   - проаналізувати існуючі рішення та методи процесуального створення ландшафту;
   - розробити власний метод процедурної генерації воксельного ландшафту;
   - розробити алгоритмічне та програмне забезпечення реалізації розробленого методу;
   - провести аналіз розоробленого метод;
   - зробити висновок за результатами дослідження.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
   - діаграма потоку даних алгоритмічно-програмного методу процедурної генерації воксельного ландшафту із 3D-тайлів;
   - діаграма компонентів алгоритмічно-програмного методу процедурної генерації воксельного ландшафту із 3D-тайлів.

7. Орієнтовний перелік публікацій:

- Тези доповіді на конференції Прикладна математика та комп'ютинг ПМК-2021 "Algorithmic-Software Method of Procedural Generation of Voxel Landscape from 3D-Tiles"

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС		

9. Дата видачі завдання «11» жовтня 2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	21.12.2020	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	05.04.2021	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	07.05.2021	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення; підготовка матеріалів доповіді на конференції ПМК-2020	11.10.2021	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	13.12.2021	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	21.02.2022	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; робота над розділом з охорони праці	25.04.2022	
8.	Оформлення текстової і графічної частини магістерської дисертації	07.05.2022	

Студент Юрій БРУСЕНЦОВ

Науковий керівник Євгенія СУЛЕМА

РЕФЕРАТ

Актуальність теми. Процедурна генерація контенту – це процес створення певних медіа даних із використанням комп’ютерних алгоритмів. Алгоритми та методи процедурної генерація дозволяють вирішувати цілу низку нагальних задач, таких як зменшення об'єму даних контенту, зменшення швидкості створювання контенту та збільшення обсягів створеного контенту, тощо. Більшість алгоритмів процедурної генерації створюють ландшафти, які потребують подальшої ручної обробки та доопрацювання. Створення та візуалізація ландшафтів є актуальним завданням при розробленні навчальних та моделюючих середовищ, відеоігор, створення матеріалів для фільмів та анімації. Із популяризацією використання воксельної та низькополігональної графіки алгоритми процедурної генерації все частіше використовуються при створенні контенту.

Об’єктом дослідження є процес процедурної генерації воксельного ландшафту з 3D-тайлів.

Предметом дослідження є методи, алгоритми та програмне забезпечення для генерації воксельного ландшафту.

Мета роботи: оптимізувати процеси процедурної генерації ландшафту.

Методи дослідження. В роботі використовуються методи методи комп’ютерного моделювання, методи оптимізації та теорія програмування.

Наукова новизна. Запропоновано алгоритмічно-програмний метод генерації воксельних ландшафтів, який відрізняється від існуючих застосуванням колапсу хвильової функції та методу послідовної генерації тайлів, що дозволяє підвищити швидкодію програмної обробки даних в середньому у 2 рази та зменшити вірогідність некоректної генерації мапи воксельного ландшафту.

Практична цінність роботи полягає у можливості застосування отриманих результатів для ефективної генерації ландшафтів на основі 3D-тайлів.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювались на XIV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2021.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень.

У першому розділі розглянуто та проаналізовано найпопулярніші методи програмної генерації ландшафту, наведені їх основні переваги та недоліки та наведено приклади генерації.

У другому розділі описано постановку задачі та вимоги до розроблюваного алгоритмічно-програмного методу, проаналізовано та наведено приклад роботи послідовного методу генерації воксельного ландшафту та методу генерації із застосуванням колапсу хвильової функції. Було запропоновано власний метод генерації воксельного ландшафту на основі 3D-тайлів на основі комбінації двох розглянутих методів генерації.

У третьому розділі обґрунтовано вибір засобів реалізації розроблюваного програмного забезпечення та наведена архітектура розроблюваної системи із детальним описом кожного окремого модулю. У даному розділі було описано основні етапи генерації воксельного ландшафту: попередня обробка тайлів, аналіз та обробка мапи чанків та генерація чанку.

У четвертому розділі здійснено опис тестування розробленого програмного забезпечення, зроблено порівняльний аналіз розробленого методу, надано оцінку часової ефективності генерації із використання

різних методів процедурної генерації та оцінку часової ефективності методу генерації із застосуванням механізму кешування даних тайлсету.

У висновках проаналізовано отримані результати роботи.

У додатках наведено діаграму потоку даних запропонованого методу генерації воксельного ландшафту, алгоритм побудови воксельного ландшафту, діаграму модулів розробленого алгоритмічно-програмного методу та діаграму класів розробленого алгоритмічно-програмного методу.

Робота виконана на 75 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 26 найменувань. У роботі наведено 36 рисунків та 3 таблиці.

Ключові слова: процедурна генерація, комп’ютерна графіка, генерація ландшафтів, воксель.

ABSTRACT

Theme urgency. Procedural content generation is the process of creating certain media data using computer algorithms. Algorithms and methods of procedural generation allow to solve a number of urgent problems, such as reducing the amount of content data, reducing the speed of content creation and increasing the amount of content created, and so on. Most procedural generation algorithms create landscapes that require further manual processing and refinement. Creating and visualizing landscapes is an urgent task in the development of learning and modeling environments, video games, creating materials for films and animation. With the popularization of voxel and low-polygon graphics, procedural generation algorithms are increasingly used in content creation.

Object of research is the process of procedural generation of the voxel landscape from 3D tiles.

Subject of research is methods, algorithms and software for voxel landscape generation.

Research objective: to optimize the processes of procedural generation of the landscape.

Research methods. Computer modeling methods, optimization methods and programming theory.

Scientific novelty. An algorithmic-software method of voxel landscape generation is proposed, which differs from the existing ones by the use of wave function collapse and sequential tile generation method, which allows to increase the speed of software data processing by 2 times on average and reduce the probability of incorrect voxel landscape map generation.

Practical value lies in the possibility of applying the results to effectively generate landscapes based on 3D tiles.

Approbation. The basic points and outcomes of the research have been presented and discussed at the 14th scientific conference for students and postgraduates «Applied mathematics and computing» PMK-2021.

Structure and content of the thesis. The master thesis consists of the introduction, four chapters, conclusions and appendixes.

The introduction presents the general description of the research, gives the overview on a current state of the scientific problem, explains the research topicality.

In the first chapter the most popular methods of software generation of the landscape, their main advantages and disadvantages.

In the second chapter the sequential method of voxel landscape generation and the method of generation using the collapse of the wave function are described; a proprietary method for generating a voxel landscape based on 3D tiles based on a combination of the two considered generation methods was proposed.

In the third chapter the choice of means of implementation of the developed software and the architecture of the developed system with a detailed description of each module are described; the main stages of voxel landscape generation: tile pre-processing, analysis and processing of fragment maps, and fragment generation are described.

In the fourth chapter the description of testing of the developed software is carried out, the comparative analysis of the developed method is made, the estimation of time efficiency of generation from use of various methods of procedural generation and an estimation of time efficiency of method of generation with application of the mechanism of caching of a tileset is given.

In the conclusions the general conclusions on the presented thesis are given; the obtained results are analyzed.

In the appendixes the data flow diagram of the proposed method of voxel landscape generation, an algorithm for constructing a voxel landscape, a

diagram of the modules of the developed algorithmic-program method and a class diagram of the developed algorithmic-program method are presented.

The thesis is presented in 75 pages, it contains 3 appendixes and 26 references to the used information sources. 36 figures and 3 tables are given in the thesis.

Key words: procedural generation, computer graphics, landscape generation, voxel.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Тайл (від англ. tile – плитка) **–** невеликі зображення однакових розмірів, які служать фрагментами великої картини.

Тайлсет **–** набір тайлів які використовуються для створення контенту.

Вóксель (від англ. Volume та англ. pixel) **–** елемент простору, позначає значення певної величини в клітинках рівномірної просторової ґратки.

Чанк (від англ. chunk – шматок) – шматок фіксованого розміру генерованої мапи, дозволяє розбити генерацію великої мапи на багато малих шматків.

WFC (від англ. Wave Function Collapse – колапс хвильової функції) – алгоритм, який генерує бітові зображення, локально подібні до вхідного бітового зображення.

Карта висот – це растрове зображення, яке використовується для збереження значень висот поверхні, для зображення у вигляді комп'ютерної 3D-графіки.

Кеш (від англ. cache – схованка) – проміжний буфер даних з швидким доступом до нього, який містить інформацію, що може бути запрошена з найбільшою ймовірністю.

Ray casting – один з методів рендерингу в комп'ютерній графіці, при якому сцена будується на основі вимірів перетину променів з поверхнею, що візуалізується.

Колапсування тайлу – процес перетворення множини елемнтів у матриці можливих тайлів до одного елементу.

Скрипт (від англ. script – сценарій) **–** короткий опис дій, що виконується системою.

CLI (від англ. Command Line Interface – інтерфейс командного рядка) – різновид текстового інтерфейсу користувача, в якому інструкції комп'ютеру можна дати тільки введенням із клавіатури текстових рядків (команд).

GUI (від англ. Graphical user interface – графічний інтерфейс користувача) – тип інтерфейсу, який дає змогу користувачам взаємодіяти з електронними пристроями через графічні зображення та візуальні вказівки.

ПЗ – програмне забезпечення

ООП – об’єктно-орієнтовне програмування.

CLR – Common Language Runtime – це компонент стандартного пакету Microsoft .NET Framework, віртуальна машина, на якій виконуються всі мови платформи .NET Framework.

GDScript – високоуровнева динамічно типізована скриптова мова програмування, розроблена для роботи із рушієм Godot.

Drag&Drop – форма виконання певних дій у графічних інтерфейсах користувача (GUI), що передбачає використання комп'ютерної миші або сенсорного екрана.

GC (від англ. garbage collection – збирання сміття) – одна з форм автоматичного керування оперативною пам'яттю комп'ютера під час виконання програм.

Фреймворк – набір абстракцій, які надають базову функціональність програмного забезпечення для подальшої композиції та модифікації. Програмна платформа, що визначає структуру програмної системи.

Зерно генерації (англ. seed або random seed) – це число (або вектор), що використовується для ініціалізації генератора псевдовипадкових чисел.

Shallow copy – однин із методів копіювання об'єкта у програмній інженерії, який для примітивних типів копіює оригінальне значення, а при копіюванні складних типів (об’єктів) копіює посилання на об’єкт, не створюючи його повну копію.

ВСТУП

Процедурна генерація контенту – це процес створення певних медіа даних із використанням комп’ютерних алгоритмів. Процедурна генерація дозволяє вирішувати цілу низку нагальних задач, наприклад: зменшення об'єму даних контенту, зменшення швидкості створювання контенту та збільшення обсягів створеного контенту.

З розширенням ринку та популяризацією гіперказуальних ігор, використання воксельної та низькополігональної графіки [1] значно збільшилось, тому розроблений метод генерації ландшафтів може допомогти у створенні контенту розробникам, дизайнерам та іншим митцям які створюють проєкти на базі воксельної та low-poly графіки.

Подібні методи генерації вже активно застосовуються для генерації контенту, але в основному методи генерації які базуються на використанні тайлів використовуються при генерації 2D-контенту.

Використання генерації воксельної мапи на основі тайлів дозволяє створювати ландшафти які можуть бути використані при моделюванні певних середовищ, при створенні фільмів, анімацій або ігор.

Метою даної магістерської дисертації є підвищення ефективності та оптимізація процедурної генерації воксельного ландшафту із 3D-тайлів використовуючи поєднання різних методів та алгоритмів генерації.

В результаті дослідження, що було проведено в рамках виконання даної магістерської дисертації, було проведено аналіз різних методів та підходів до процедурної генерації двовимірних та тривимірних ландшафтів, та запропоновано алгоритмічно-програмний метод генерації воксельних ландшафтів із 3D-тайлів, який відрізняється від існуючих застосуванням методу послідовної генерації тайлів та колапсу хвильової функції, що дозволяє підвищити швидкодію програмної обробки даних та зменшити вірогідність некоректної генерації мапи воксельного ландшафту.

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ПОСТАВЛЕНОЇ ЗАДАЧІ

1.1. Загальні положення та аналіз предметної області

У обчисленнях процедура генерація – це метод алгоритмічного створення даних на відміну від ручного, як правило, за допомогою комбінації створених людиною активів та алгоритмів, поєднаних із випадковою обробкою та обчислювальною потужністю. У комп'ютерній графіці процедурна генерація зазвичай використовується для створення текстур та 3D-моделей. У відеоіграх вона використовується для автоматичного створення великої кількості вмісту в грі. Залежно від реалізації, переваги процедурної генерації можуть включати менший розмір файлу, більший обсяг вмісту та випадковість для менш передбачуваного ігрового процесу. Процедурна генерація – це галузь синтезу засобів масової інформації [2].

Створення та візуалізація віртуальних пейзажів є актуальним завданням у різних сферах, таких, як розроблення навчальних та моделюючих середовищ, відеоігор, створення матеріалів для фільмів та анімації, а також середовища для моделювання різних природних процесів.

Одним із чудових прикладів використання процедурної генерації є гра 2016 року випуску – "No Man's Sky", відеоігра, де вся маркетингова схема була структурована навколо її процедурно створеного всесвіту. Трейлер гри та реклама обіцяли своїм гравцям 18 446 744 073 709 551 616 (вісімнадцять квінтильйонів) унікальних планет, усі з яких були процедурно створені. Іншими словами, розробники створювали не ексклюзивні профілі для кожної окремої планети, а натомість програмували гру таким чином, щоб планети будувалися за домогою програмного коду. Цей спосіб створення контенту є суттю процедурної генерації [3].

Процедурна генерація – це дуже потужний інструмент для генерації контенту, проте процедурний вміст може бути важко контролювати. Інколи

отримані внаслідок генерації результати не є дуже реалістичними, тому підлягають переробки дизайнерами.

1.2. Відомі рішення

Існує кілька основних принципів представлення даних для зберігання інформації про ландшафт:

- генерація на основі карт висот (HeightMap);
- генерація на основі іррегулярної сітки;
- генерація на основі тайлів;
- генерація на основі алгоритму Marching cubes;
- генерація на основі алгоритму Marching tetrahedra.

1.2.1. Генерація на основі карт висот

У комп'ютерній графіці карта висоти або поле висоти - це растрове зображення, що використовується в основному як дискретна глобальна сітка при моделюванні вторинних висот. Кожен піксель зберігає значення, такі як дані про висоту поверхні, для відображення у 3D-комп’ютерній графіці.

Зазвичай карту висот зберігають в файлах картинок. Це дозволяє легко вносити зміни і більш-менш наочно переглядати дані. Тоді двома координатами буде положення конкретного пікселя на зображенні, а третя координата буде представлена кольором (чим вище значення, пряма залежність від яскравості пікселя - тим більше значення висоти для цієї точки). Зазвичай такі картинки містяться в монохромному варіанті, але можна використовувати будь-які кольори. Другий варіант дає нам більше градацій висоти, ніж передбачувані 256 градацій в разі монохромного уявлення [4].

Карту висоти можна використовувати при зіставленні нерівностей для обчислення, де ці тривимірні дані створюватимуть тінь у матеріалі, при відображенні переміщень, щоб змістити фактичне геометричне положення точок над текстурованою поверхнею, або для місцевості, де карта висоти

перетворюється у тривимірну сітку [5]. Приклад карти висот можна побачити на рис. 1.1.

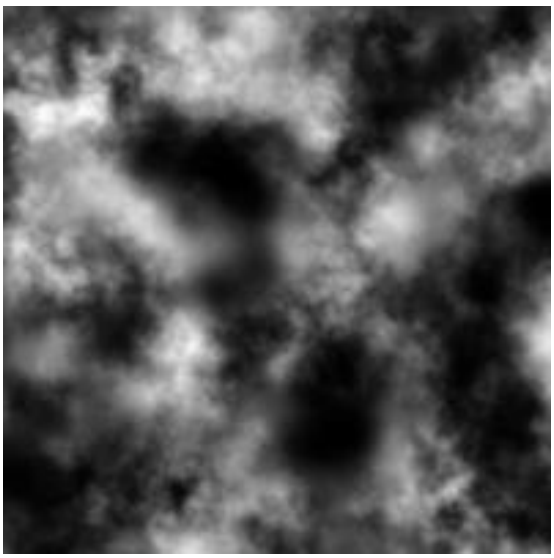


Рис. 1.1. Карта висот

Чим значення пікселя світліше – тим вище точка висоти у данному пікселі. Тобто на білих ділянках карти висоти будуть гори, а на чорних – западини або рівнини. Із карти висот можна побудувати тривимірний об’єкт який буде нагадувати ландшафт. Наприклад, якщо карту висот із рис. 1.1 перетворити на 3D-об’єкт, отримаємо результат, який продемонстровано на рис. 1.2.

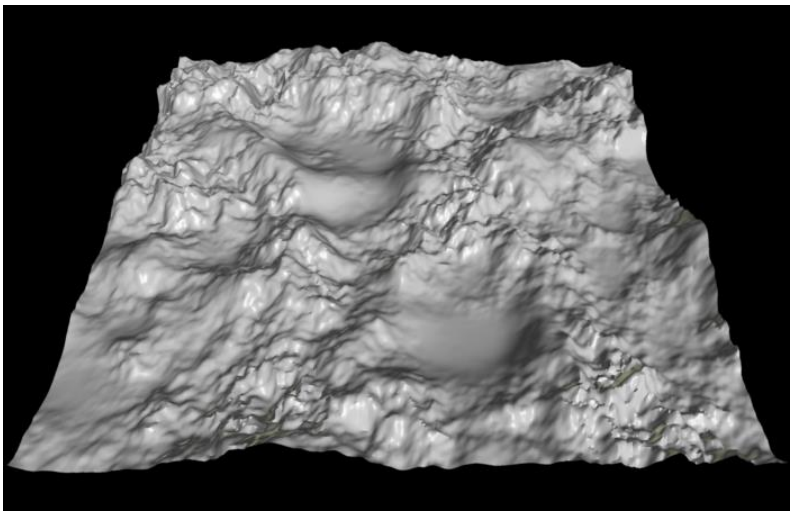


Рис. 1.2. Карта висот перетворена у 3D-об’єкт

Можемо побачити, що базова карта висот перетворена у 3D-об’єкт схожа гірський ландшафт.

1.2.2. Генерація на основі іррегулярної сітки

Ще один спосіб представлення даних для ландшафтів - іррегулярна сітка вершин і зв'язків їх з'єднують [4]. Найчастіше такі рішення застосовуються в спеціалізованих пакетах для ігор або спеціальних пакетах для роботи з тривимірною графікою та зберігаються у вигляді тривимірних моделей. Це дає основний виграш в порівнянні з картами висот - використовується значно менше інформації для побудови ландшафту. Необхідно зберігати тільки значення висот кожної вершини і зв'язку ці вершини з'єднують. Це дає нам виграш в швидкості при передачі величезних масивів інформації по AGP, в процесі візуалізації ландшафту. Приклад зображення іррегулярної сітки можна побачити на рис. 1.3.

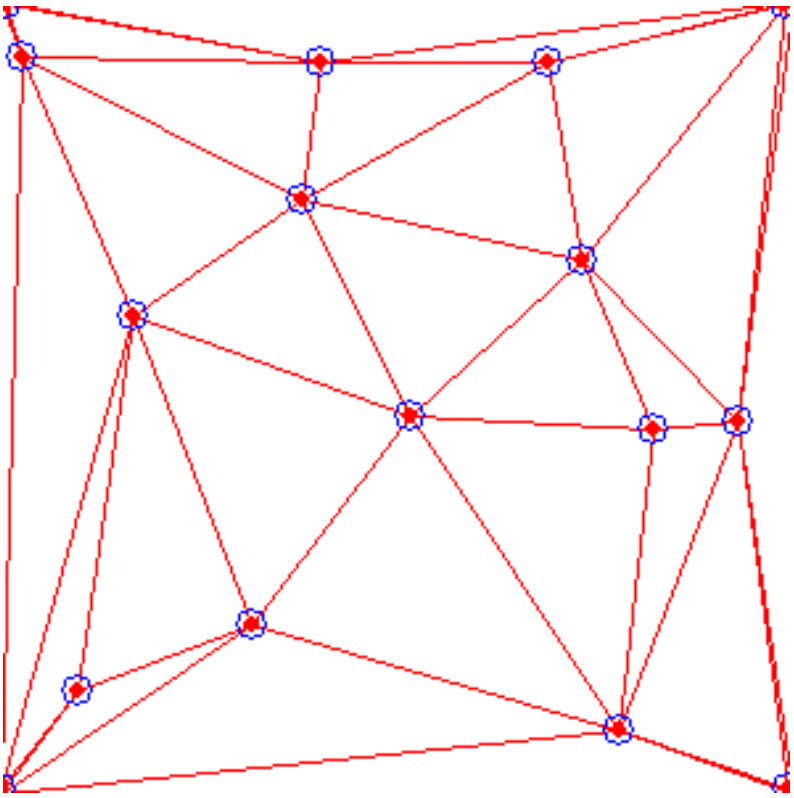


Рис. 1.3. Приклад графічного зображення іррегулярної сітки

1.2.3. Генерація на основі тайлів

Генерацію на основі тайлів використовують в основному для створення двовимірного контенту, наприклад для побудови рівнів двовимірної гри. Особливістю ігор які побудовані за допомогою тайлів є те, що ігрова зона складається з невеликих квадратних (або, набагато рідше, прямокутних, паралелограмних або шестикутних) графічних зображень, які називаються плитками, викладеними в сітку. Те, що екран зроблений з такої плитки, є технічною відмінністю, і це може бути не очевидним для людей, які грають у гру. Повний набір плиток, доступний для використання в ігровій зоні, називається набором плиток або tilemap. Ігри на основі тайлів зазвичай імітують вид зверху вниз, вид збоку або 2.5D-ігровий майданчик і майже завжди є двовимірними [7].

Приклад згенеровного рівня на основі 2D-тайлів можна побачити на рис. 1.4.

Рис. 1.4. Приклад генерації ігрового рівня на основі тайлів

Основною ідеєю генерації на основі тайлів є те, що для генерації створюється певний тайлсет, тобто набір тайлів, за допомогою яких здійснюється генерація мапи.

1.2.4. Генерація на основі алгоритму Marching cubes

Алгоритм Marching cubes проходиться по скалярному полю, бере вісім сусідніх точок (формуючи уявний куб), потім визначає які полігони потрібні щоб представити частину ізоповерхні, що проходить крізь цей куб. Потім всі полігони з'єднуються в бажану поверхню [6].

Для кожної точки площини створюється індекс до обчисленого попередньо масиву з 256 можливих конфігурацій многокутників всередині куба. Тобто для створення ландшафту використовується 256 можливих конфігурацій куба, 15 із яких є унікальними, а інші створюються за допомогою повороту даних основних конфігурацій.

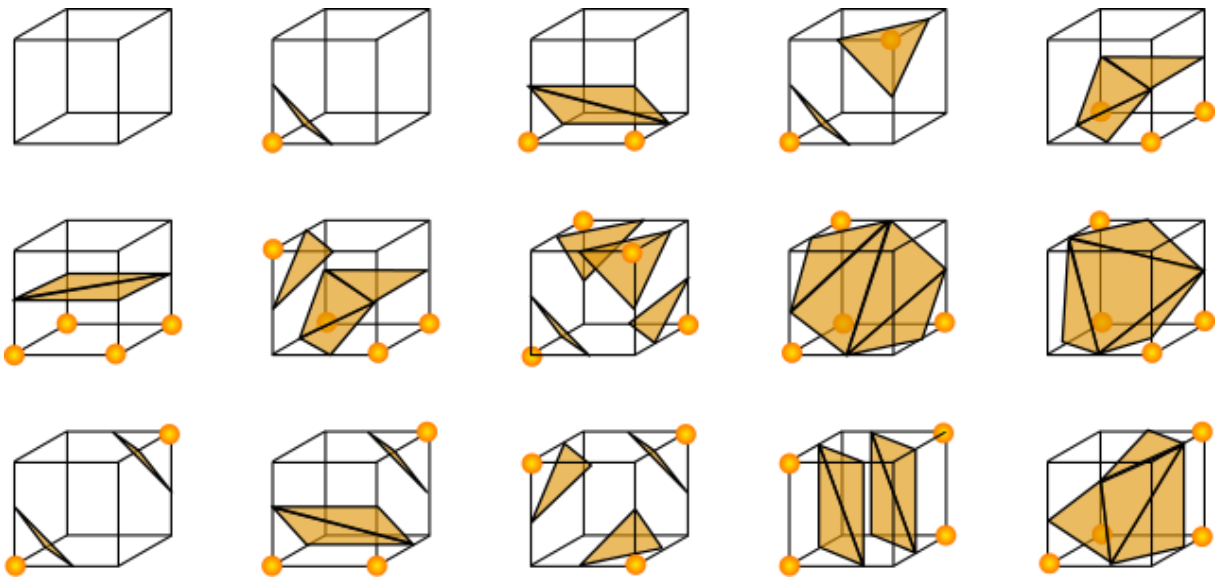


Рис. 1.5. 15 унікальних конфігурацій куба у алгоритмі Marching cubes

Застосування цього алгоритму в основному пов'язані з медичними візуалізаціями, такими як комп'ютерна томографія та магнітно-резонасний знімок, метакулі або інші метаповерхні (рис. 1.6).

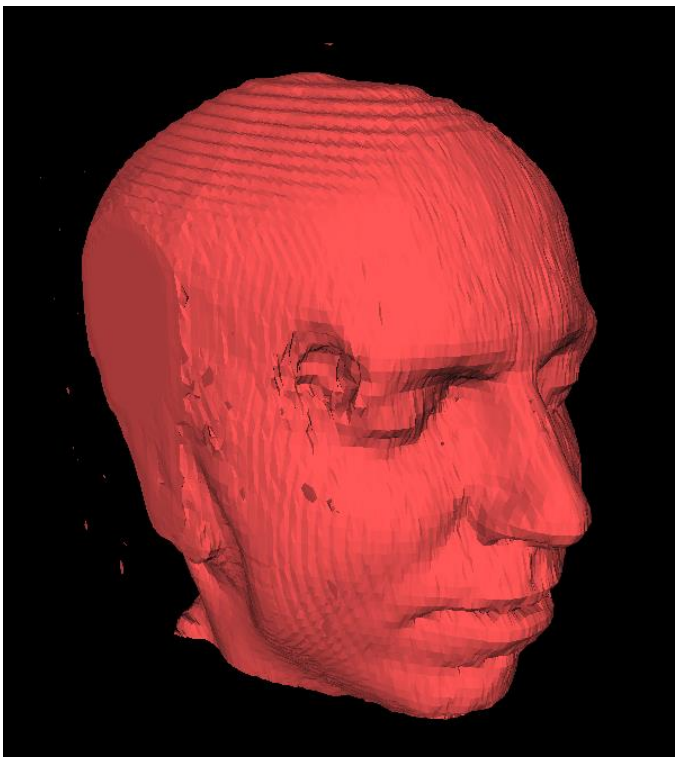


Рис. 1.6 Голова зроблена з 150 магнітно-резонансних знімків перероблених алгоритмом Marching cubes

Алгоритм Marching Cubes був першим прикладом у галузі комп'ютерної графіки, що спровокував скандал у галузі патентування ПЗ. Він був запатентований, незважаючи на відносну очевидність вирішення задачі генерації поверхні. Алгоритм під назвою Marching tetrahedra, був розроблений, щоб обійти патент і проблему неоднозначності в деяких конфігураціях.

1.2.5. Генерація на основі алгоритму Marching tetrahedra

Алгоритм Marching tetrahedra є певною модифікацією методу Marching Cubes. Даний алгоритм працює із тетраедрами, а не з кубами, та кількість можливих конфігурацій тетраедра становить лише 8 (рис. 1.6).

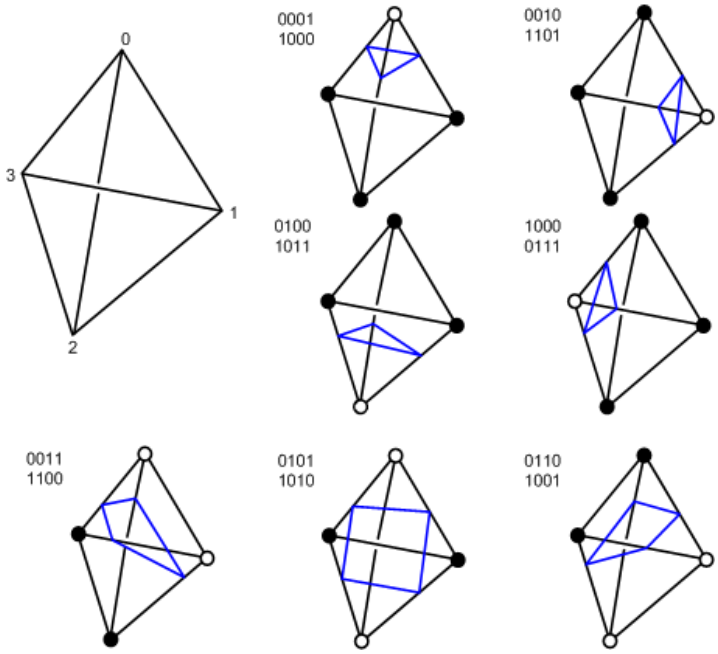


Рис. 1.6. 8 унікальних конфігурацій тетраедра у алгоритмі Marching tetrahedra

У алгоритмі Marching tetrahedra кожен куб розбивається на шість рівних тетраедрів, розрізаючи куб навпіл тричі, розрізаючи по діагоналі кожну з трьох пар протилежних граней (рис. 1.7). Таким чином, всі тетраедри мають одну з головних діагоналей куба. Замість дванадцяти ребер куба ми тепер маємо дев’ятнадцять ребер: вихідні дванадцять, шість діагоналей граней і головна діагональ. Так само, як і в Marching Cubes, перетини цих ребер із ізоповерхнею апроксимуються шляхом лінійної інтерполяції значень у точках сітки [8].

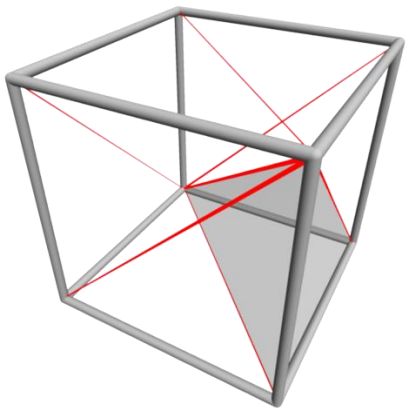


Рис. 1.7. Куб, розділений на шість тетраедрів, один із тетраедрів затінений

Завдяки простоті та тому, що це, по суті, алгоритм, керований трикутником, алгоритм маршових тетраедрів користувався численними дослідженнями груп GPGPU щодо його апаратного прискорення [9].

1.3. Порівняння існуючих рішень

1.3.1. Генерація на основі карти висот

Метод процедурної генерації ландшафтів на основі карти висот має свої переваги та недоліки.

Основні переваги генерації на основі карти висот:

- наочність, в будь-якій програмі перегляду графічних файлів можна відразу побачити всю інформацію;
- простота зміни цих самих даних, так як існує безліч програм для роботи з растровою графікою;
- в таких картах можна зберігати не тільки дані про висоту. Наприклад, припустимо, що для зберігання висоти ми використовуємо 16 біт, тобто дві кольорові компоненти, що дорівнює 256*256 = 65536 градацій висоти. Решта 8 біт можна використовувати для зберігання інформації про будь-які особливості ландшафту, наприклад, розташування будівель, будов, мостів, рослинності і так далі;
- так як вершинні точки розташовані регулярно і досить близько, можна більш правильно і досить акуратно проводити динамічне освітлення (найчастіше, освітленість вершини прямо залежить від відстані від цієї вершини до джерела освітлення). Це і є та сама користь від надмірності даних.

Але у даного методу є один істотний недолік – занадто багато описів для точок, а також, в деяких випадках, спостерігається надмірність даних (наприклад, коли задається проста площина, то, для побудови простої

площині буде використовуватися безліч точок, хоча можна б було обійтися трьома.

1.3.2. Генерація на основі іррегулярної сітки

У методу генерації поверхні на основі іррегулярної сітки є певні переваги та недоліки.

Перевага: використовується значно менше інформації для побудови ландшафту. Необхідно зберігати тільки значення висот кожної вершини і зв'язок цих вершин. Це дає виграш у швидкості при передачі величезних масивів інформації по AGP, в процесі візуалізації ландшафту.

Недоліки:

- алгоритми побудови ландшафтів в основному призначені для регулярних карт висот. Оптимізація таких алгоритмів під цей спосіб зажадає значних зусиль;
- складнощі при динамічному освітленні - вершини розташовані досить далеко один від одного і нерівномірно;
- зберігання, перегляд, модифікація такого ландшафту також представляє складнощі. При використанні карт висот можна використовувати досить прості та стандартні засоби перегляду та редагування піксельної графіки. Для перегляду іррегулярної сітки потрібне більш складне та вагоме програмне забезпечення.

1.3.3. Генерація на основі тайлів

Розглянемо основні переваги та недоліки методів генерації контенту на основі тайлів.

Переваги:

- згенерований контент не потребує додаткової обробки;
- згенерований контент можна легко редагувати.

Недоліки:

- для генерації контенту необхідно створити початковий тайлсет;
- в основному використовується для роботи із 2D-контентом. На даний момент не має популярного методу генерації ландшафтів на основі 3D-тайлів.

1.3.4. Генерація на основі алгоритму Marching cubes

Метод процедурної генерації ландшафтів на основі Marching cubes має свої переваги та недоліки.

Перевагою методу є те, що він досить простий у своїй реалізації.

Проте у методу є ряд недоліків:

- необхідно розглядати багато варіантів конфігурацій куба. Якщо використовувати 15 конфігурацій, то отримати суцільну поверхню неможливо;
- Marching cubes не дозволяє створювати поверхню із гострими кутами.

1.3.5. Генерація на основі алгоритму Marching cubes

Оскільки даний метод є модифікацією Marching cubes, то метод генерації на основі алгоритму Marching tetrahedra також наслідує дані недоліки.

Проте, якщо порівнювати його із методом Marching cubes, даний метод має кілька переваг:

- додаткові перетини забезпечують трохи кращу роздільну здатність вибірки;
- алгоритм може працювати на структурованих та неструктурованих сітках.

1.4. Висновки до розділу

У першому розділі магістерської дисертації було розглянуто та проаналізовано найпопулярніші методи програмної генерації ландшафту, наведені їх основні переваги та недоліки та наведено приклади генерації.

У розділі було розглянутий метод генерації на основі карт висот, метод на основі іррегулярної сітки, метод генерації на основі тайлів, метод генерації на основі алгоритму Marching cubes та метод генерації на основі алгоритму Marching tetrahedra. Також було наведено особливості використання розглянутих методів генерації ландшафту, наведено їх переваги та недоліки.

2. РОЗРОБЛЕННЯ АЛГОРИТМІЧНО-ПРОГРАМНОГО МЕТОДУ ГЕНЕРАЦІЇ ВОКСЕЛЬНОГО ЛАНДШАФТУ ІЗ 3D-ТАЙЛІВ

2.1. Постановка задачі

Необхідно розробити алгоритмічно-програмний метод генерації воксельного ландшафту із 3D-тайлів.

Ландшафт повинен складатися із попередньо заготовлених тайлів користувачем: тайли повинні бути 3D-об’єктами, імпортованими, наприклад, у форматі .obj, та мати однаковий розмір та однакові кольори вокселів у тайлах, які користувач планує з’єднувати між собою.

Алгоритмічно програмний метод повинен генерувати ландшафт розміром 16x16 вокселів не довше ніж за 16 мілісекунд, щоб забезпечувати плавну генерацію при частоті оновлення 60 кадрів на секнду.

Алгоритмічно програмний метод повинен бути розширюваним, тобто на його основі можна побудувати більший програмний модуль із більшим функціоналом, наприклад розбиття генерованого ландшафту на чанки та генерація ландшафту на основі передчасно заготовленої його деякої частини.

Моделі тайлів які подаються на вхід алгоритмічно-програмному методу мають містити однакову кількість вокселів на бічних сторонах по всіх осях. Розміри тайлів у тайлсеті, який подається на вхід алгоритмічно-програмному наданому повинні бути однакового розміру та на всіх тайлах.

Алгоритмічно-програмний метод повинен здійснювати автоматичний поворот тайлів у горизонтальній площині, тобто для створення одного тайлу із різним кутом повороту користувачу потрібно буде надати лише один тайл та вказати необхідну кількість поворотів. Також алгоритмічно-програмний метод повинен надавати користувачу методу можливість вказати певний відносний відсоток шансу генерації кожного окремого тайлу у тайлсеті, тобто надати можливість знизити або збільшити шанс генерації певних обраних користувачем тайлів.

2.2. Опис запропонованого методу

Запропоновано використати метод процедурної генерації воксельного ландшафту із 3D-тайлів. В переважній більшості випадків 3D-ландшафт генерують за допомогою карт висот, але методи процедурної генерації на основі карт висот дозволяють користувачу отримати абсолютно випадковий ландшафт, у той час як запропонований метод надає можливість згенерувати мапу на основі попередньо оброблених або згенерованих частин.

Для використання методу, йому на вхід потрібно подати створений тайлсет із 3D-об’єктів, тобто тайлів, із яким буде працювати розроблений алгоритм.

На рис. 2.1 зображено невеликий приклад створеного тайлсету, який складається із 6 тайлів. Даний тайлсет було створено у програмі MagicaVoxel [10].

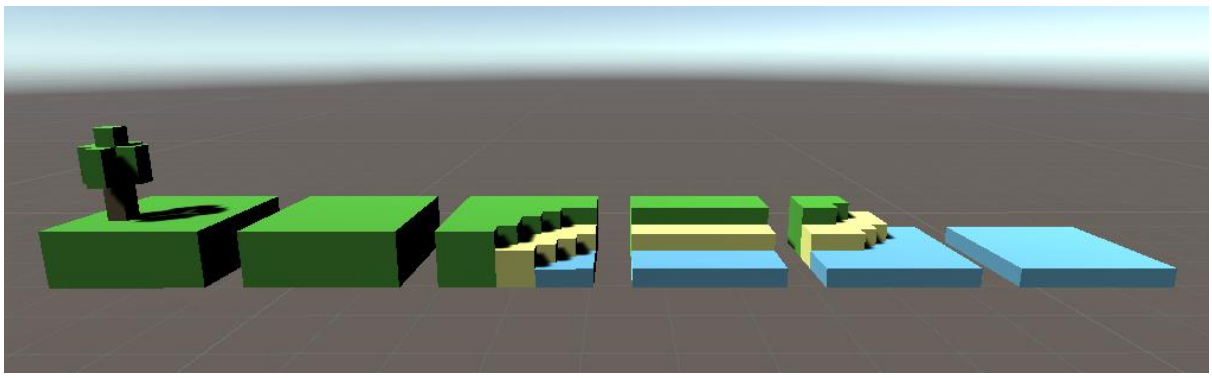


Рис. 2.1. Приклад воксельного тайлсету

Суть методу полягає в тому щоб розмістити тайли так, щоб зовнішні сторони воксельних тайлів, які стоять поруч, мали однакові кольори. На рис. 2.2 можна побачити приклад правильного та неправильного розташування суміжних тайлів.

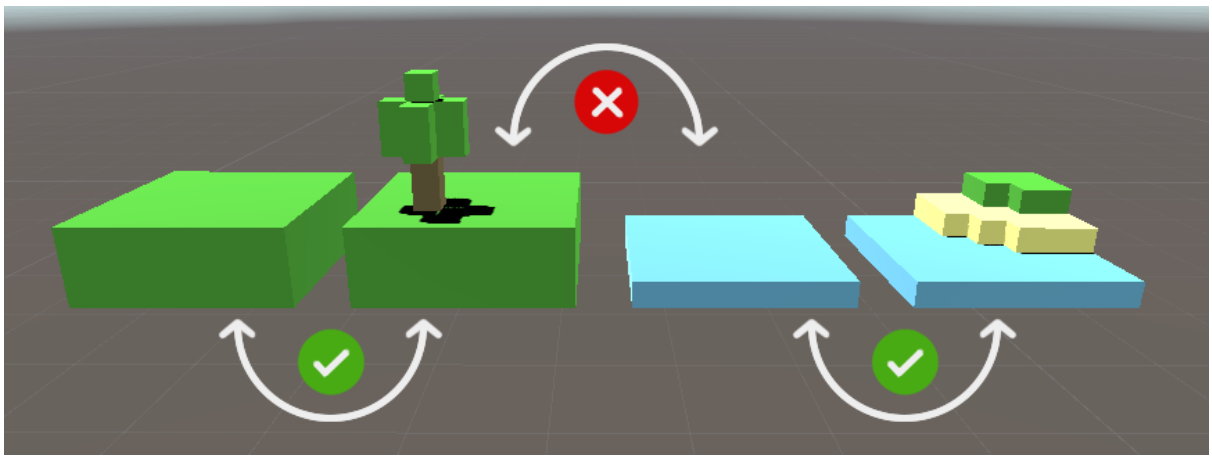


Рис. 2.2. Приклад розстановки тайлів

На рисунку бачимо, що 2 пари тайлів зліва і справа можна розмістити разом, проте центральні тайли розмістити разом не можна.

Тайли також можуть повертатися навколо своєї осі задля коректної постановки, тобто користувачу не потрібно створювати 4 однакові тайли із різним кутом повороту.

Приклад згенерованого ландшафту за допомогою воксельного тайлсету можна побачити на рис. 2.3. Даний ландшафт сгенеровано на основі тайлсету зображеного на рис. 2.1.

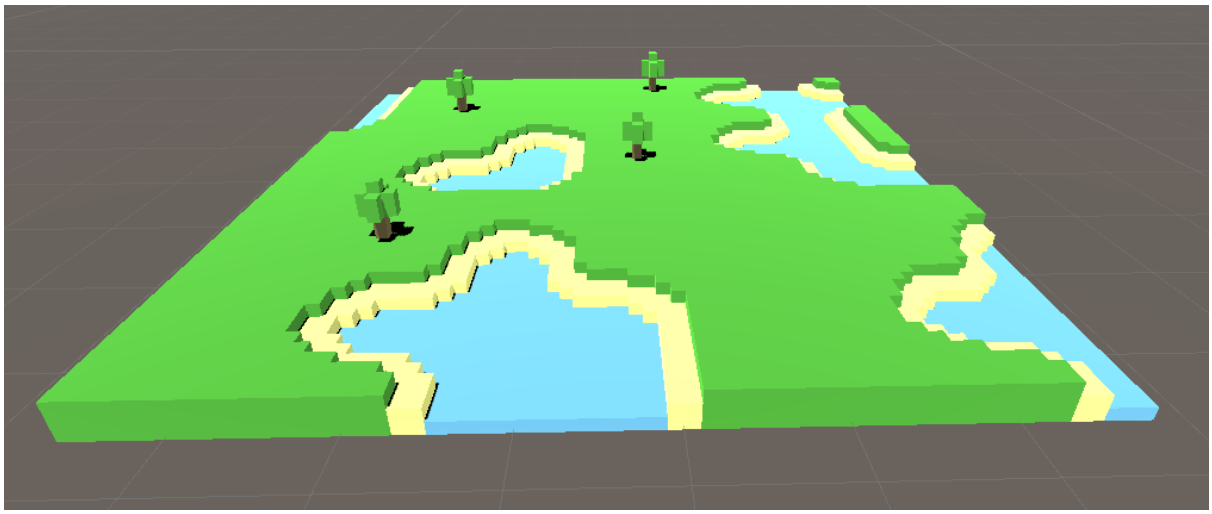


Рис. 2.3. Приклад згенерованого ландшафту за допомогою запропонованого алгоритмічно-програмного методу

Також слід зауважити, що метод, умовно кажучи, дивиться не лише на кольори, але і на наявність самого вокселю в конкретній позиції тайлу. Тобто, якщо у певній позиції тайлу вокселю нема, тоді й у тайлу який планує бути сусідом поточного елементу не повинно бути вокселю у суміжній позиції.

Запропонований метод генерації передбачає представлення мапи у вигляді матриці тайлів (рис. 2.4). Кожна клітинка матриці містить масив можливих для підстановки тайлів. Кожна клітинка матриці аналізує тайли у сусідніх клітинках й поступово видаляє тайли із матричних елементів, завдяки чому і відбувається процес генерації. Тобто, як результат процесу генерації мапи ми повинні отримати матрицю тайлів із одним елементом у кожній клітинці матриці.

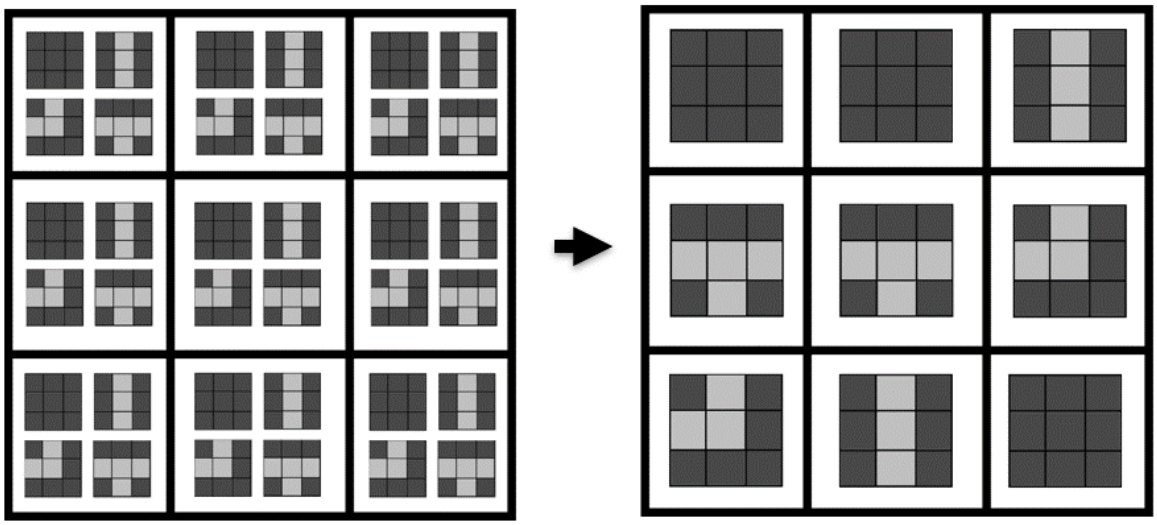


Рис. 2.4. Приклад представлення генерованої мапи у вигляді матриці елементів

2.2.1. Послідовний метод генерації

Одним із запропонованих методів обходу матриці елементів є метод послідовний метод. У цьому методі координати для встановлення тайлів йдуть послідовно, тобто у циклі.

У послідовному методі генерації виконується послідовний обхід всіх елементів матриці, та послідовний обхід всіх тайлів у поточному елементі матриці.

Якщо поточний тайл можна поставити поруч хоча б із одним тайлом у кожному із сусідніх клітинок, тоді даний тайл зберігається у поточному елементі, інакше, даний тайл видаляється із масиву можливих тайлів у поточній клітинці тайлу. Якщо після видалення тайлів у поточному елементі залишається більше ніж один, тоді фінальний тайл обирається випадково.

Обхід матриці можливих тайлів можна записати у наступному вигляді:

$$M[x,y]_i = M[x,y]_i \cap V, \tag{2.1}$$

де M – це матриця генерованої мапи;

i – індекс поточного тайлу;

x, y – координати поточного елементу матриці;

V – це множина можливих для підстановки тайлів поточного елементу.

Множина можливих для підстановки тайлів елементу формується перед ітераційним процесом, тобто для кожного тайлу із тайлсету формується масив можливих для підстановки елементів у кожному із напрямків.

Приклад генерації чанку даним методом продемонстровано на рис. 2.5. На рисунку зеленим кольором позначено поточний елемент обходу матриці.

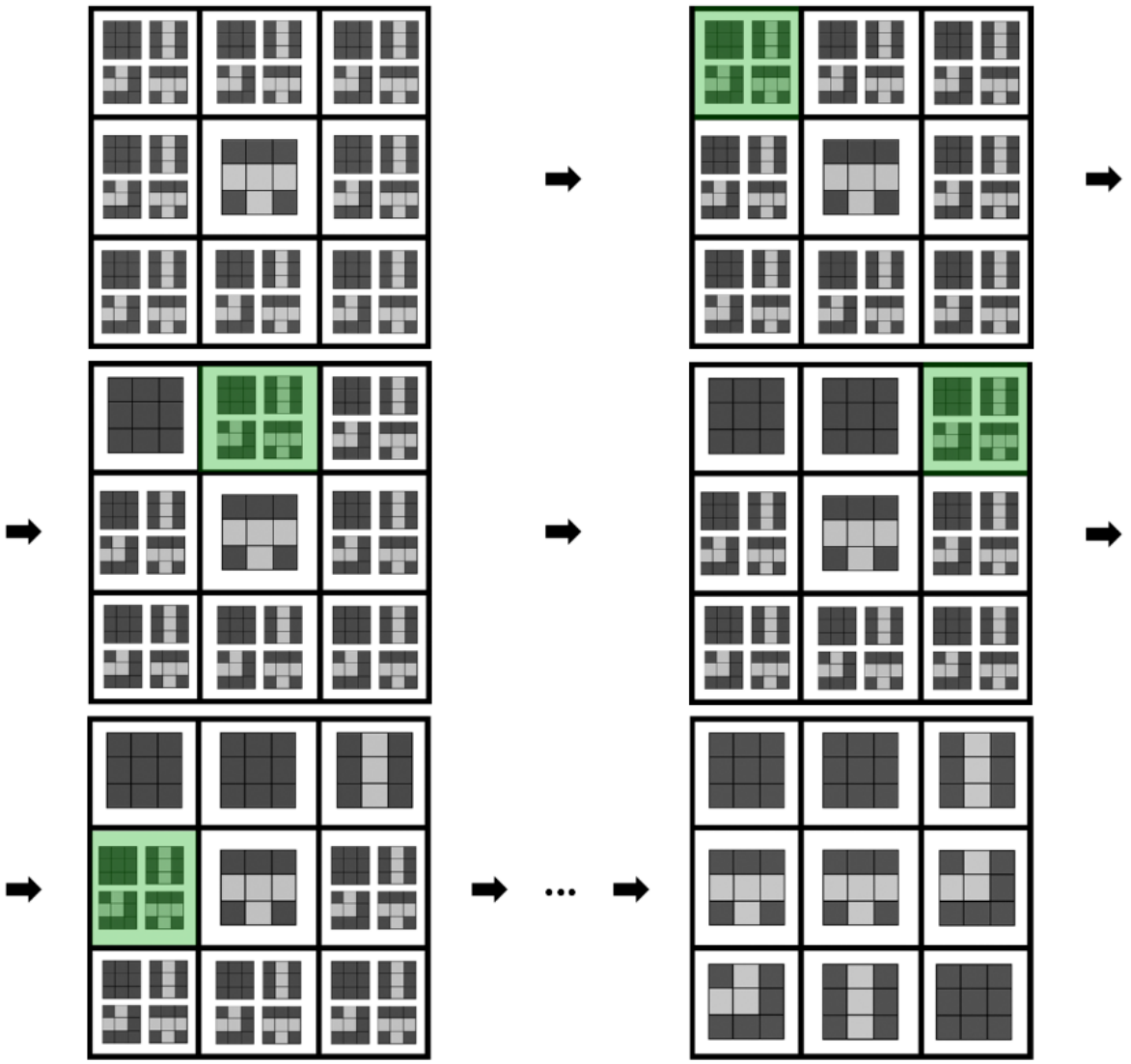


Рис. 2.5. Приклад генерації мапи із використанням послідовного методу обирання поточного елементу матриці

Швидкодія методу генерації воксельного ландшафту із використання послідовної підстановки тайлів є доволі швидкою, проте, існує доволі висока вірогідність що уході роботи даний метод згенерує мапу із дірками. Дана проблема виникає через неможливість встановки жодного тайлу у поточний елемент обходу матриці. Приклад генерації мапи із дірками продемонстровано на рис. 2.6.

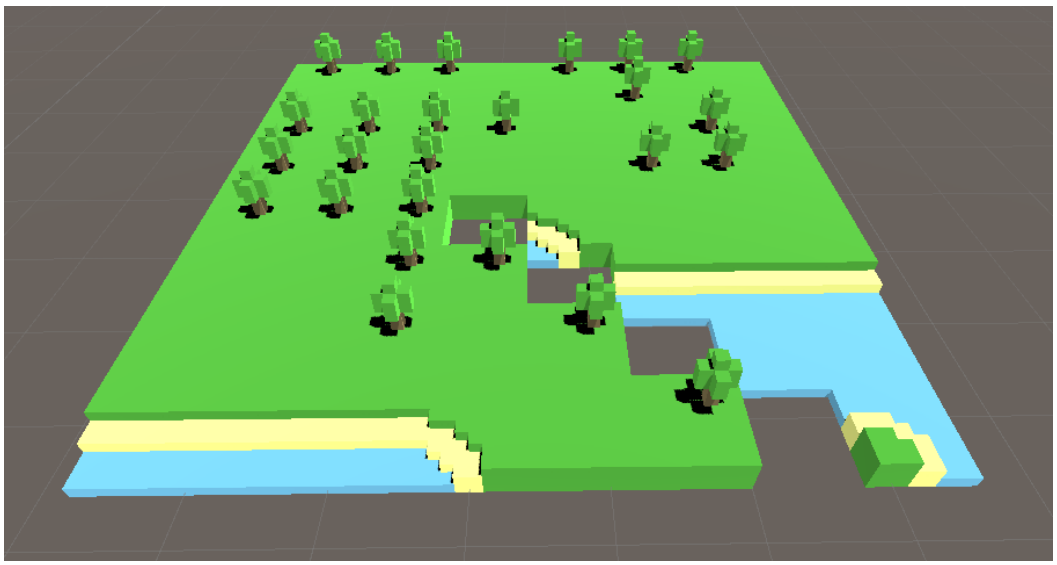


Рис. 2.6. Приклад некоректної генерації мапи із «дірками»

Одним із доволі простих рішень цієї проблеми може бути перезапуск алгоритму доки не буде створено мапи без дірок. Проте, на досить великих тайлсетах відсоток успішних генерацій буде низьким. Для вирішення проблеми некоректної генерації мапи можна використати Wave Function Collapse алгоритм.

2.2.2. Генерація на основі WFC

WFC (Wave Function Collapse) – алгоритм, який генерує бітові зображення, локально подібні до вхідного бітового зображення [11].

Запропоновано метод генерації воксельних ландшафтів на основі WFC, який буде оперувати не пікселями зображень, а поданими на вхід 3D-тайлами [12].

Даний метод є ітераційним та використовує чергу для обходу матричних елементів.

Ітераційний процес ділиться на два основних етапи: оброблення черги генерації та колапсування тайлу.

Колапсування тайлу (від англ. collapse) – процес перетворення множини елементів у матриці можливих тайлів до одного елементу.

Перед початком ітераційного процесу до черги генерації додаються початкові елементи, із яких буде розпочато генерацію. Запропоновано використовувати кутові елементи матриці як початкові елементи генерації.

Ітераційний процес починається із обробки всіх елементів у черзі генерації:

- видаляються всі неможливі для постановки тайли у поточному елементі;
- якщо тайлів після видалення не залишилось, то поточний елемент знов додається до черги разом із сусідніми елементами, матриці можливих тайлів сусідніх елементів скидуються до початкових значень;
- якщо було видалено хоч один тайл з поточного елементу, то сусідні до поточного елементи додаються до черги генерації.

Далі відбувається колапсування одного із елементів матриці можливих тайлів:

- обирається елемент матриці із найменшою кількістю можливих для встановлення тайлів;
- якщо можливих для підстановки тайлів у обраному елементі матриці більше ніж один, то фінальний тайл обирається за певним алгоритмом;
- сусідні до зколапсованого елементу клітинки матриці додаються до черги генерації.

Ітераційний процес продовжується, поки у всіх елементах матриці не залишиться по одному тайлу, або кількість ітерацій перевищить допустиме значення.

Розглянемо приклад роботи методу на основі WFC для генерації вкосельної мапи (рис. 2.7).

Алгоритм виконує обхід матриці попередньо заданого для генерації розміру. У даному випадку матриця тайлів має 3 стовпці та 3 рядки, та кожен тайл із тайлсету має розмір 3 на 3 на 3 у осях x, y та z відповідно.

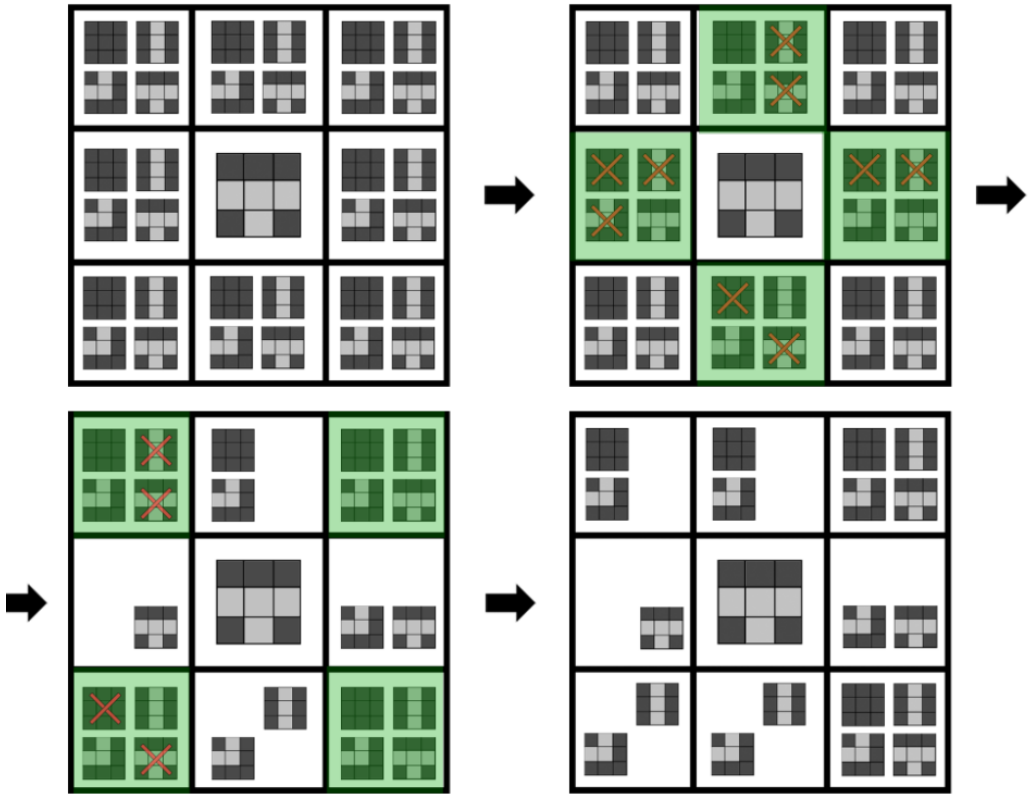


Рис. 2.7. Приклад ітераційного процесу методу генерації на основі колапсу хвильової функції

Можна побачити, що на початку першої ітерації даного прикладу вже один зколапсований елемент, цим елементом є центральна клітинка матриці. Отже до перед початком ітераційного процесу до черги генерації було додано сусідів центрального тайлу. Тобто лівий центральний, верхній цетральний, правий центральний, та нижній центральний елементи додано до черги генерації.

Надалі у елементах із черги генерції було вилучено всі неможливі для підстановки тайли. Сусідні елементи матриці, з яких було видалено хоч один тайл було додано до черги генерації. Отже кутові елементи матриці також було додано до черги генерації та із них було вилучено неможливі для підстановки тайли. Далі до черги генерації було також додано сусідні до лівого нижнього та лівого верхнього елементу клітинки, проте в них не було вилучено жодного тайлу.

Завершенням першої ітерації є колапсування лівого центрального елементу, адже це є елемент із найменшою кількістю доступних для встановлення тайлів.

Надалі ітераційний процес продовжується, доки у всіх елементах матриці не залишиться по одному тайлу (рис. 2.8).

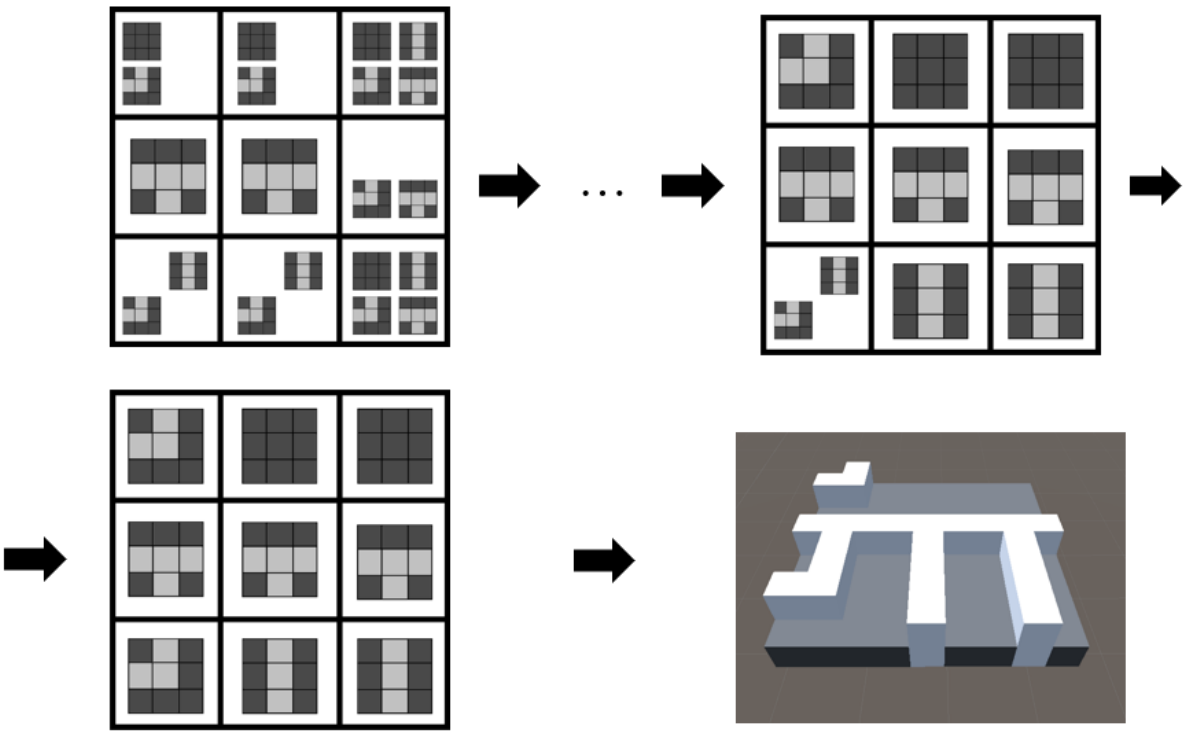


Рис. 2.8. Результат роботи методу генерації на основі колапсу хвильової функції

Слід зауважити, що колапс хвильової функції можна використовувати із різною глибиною.

Глибина – це умовно кажучи дальність погляду алгоритму WFC. У попередньому прикладі наведено використання колапсу хвильової функції із глибиною яка дорівнює одиниці, тобто навколо обраного тайлу генерується умовне кільце можливих для постановки тайлів товщиною у одну клітинку. Дане значення змінюється за допомогою зміни кількості елементів, які додаються до черги генерації. Тобто якщо до черги генерації

додається підматриця розміром 5 на 5 елемнтів, не враховуючи поточний тайл посередині підматриці, тоді глибина генерації буде дорівнювати двом.

Також слід зауважити, що генерація у запропонованому методі генерація можлива не лише по горизонтальній площині. Запропонований метод може порівнювати всі 6 сторін 3D-тайлу для генерації мапи.

2.2.3. Комбінований метод генерації

Запропоновано використати метод генерації воксельних ландшафтів, який використовує поєднання генерації за допомогою методу послідовної підстановки тайлів та методу із використанням колапсу хвильової функції.

Перша спроба генерації надається методу послідовної підстановки тайлів, якщо спроба генерації видалася невдалою, генерація мапи виконується за допомогою методу із використанням WFC.

Запропонований метод генерації воксельного ландшафту передбачає наступні етапи:

1. Отримання даних тайлсету.
    - Вхідні дані: Тайлсет (3D-моделі, колірний атлас)
    - Результат: Сформована структура даних на основі вхідних даних
2. Валідація отриманих даних.
    - Вхідні дані: Наданий тайлсет
    - Результат: Провалідований тайлсет – перевірені чи вірні елементи тайлсету було подано на вхід, розмір та кольори, тощо.
3. Оброблення отриманих даних.
    - Вхідні дані: Валідний тайлсет
    - Результат: Закешований та розширений двосторонніми та чотиристоронніми тайлами тайлсет
4. Генерація чанку комбінованим методом.
    - Вхідні дані: Оброблений тайлсет, границі сусідніх чанків
    - Результат: Згенерована матриця тайлів чанку

5. Встановлення тайлів.
    - Вхідні дані: Матриця тайлів генерованого чанку
    - Результат: Згенерований чанк

Діаграму потоку даних запропонованого методу генерації воксельних ландшафтів продемонстровано на рис. 2.9.

На вхід методу від користувача надається тайлсет у вигляді 3D-моделей, у форматі **.obj** та колірного атласу у форматі зображення розміром 1 на 256 пікселів.

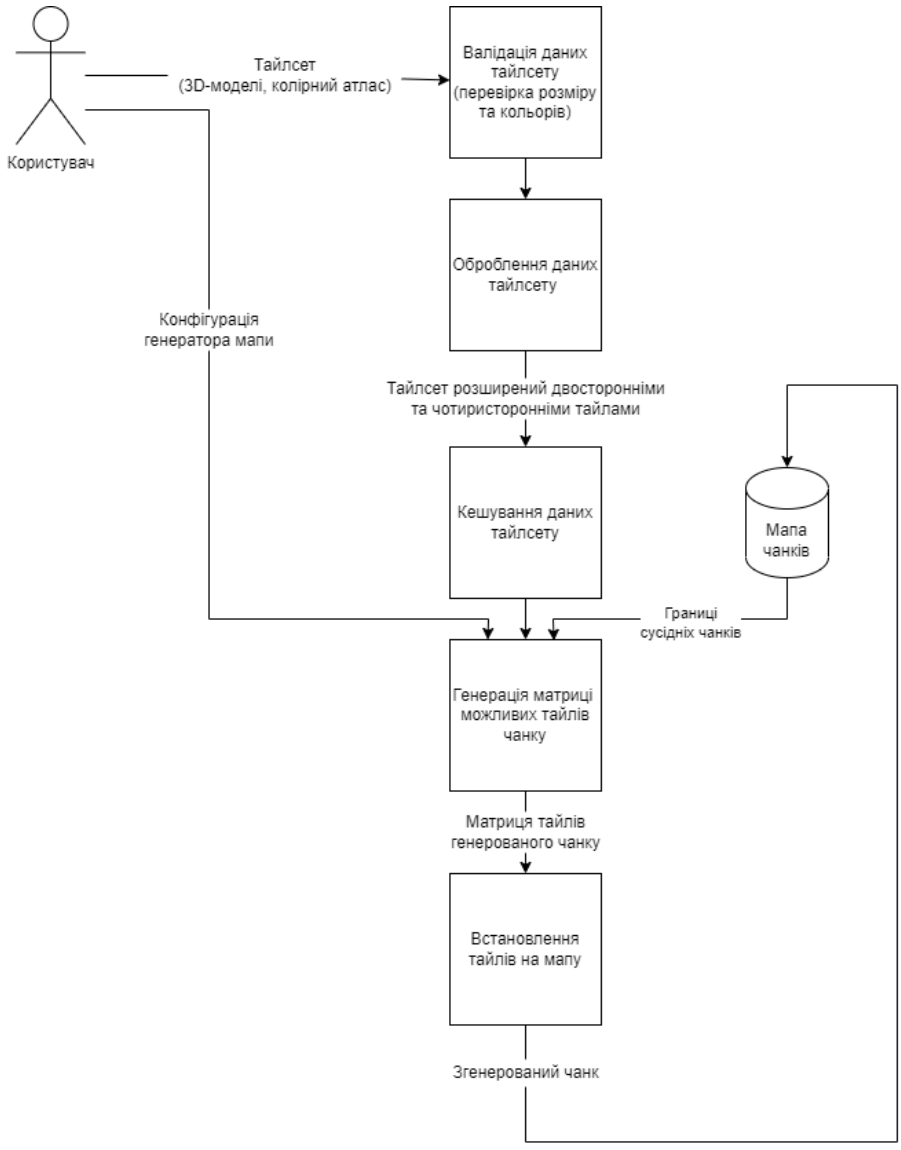


Рис. 2.9. Діаграма потоку даних запропонованого методу генерації воксельного ландшафту

На діаграмі потоку даних продемонстровано, що дані тайлсету перед потраплянням до модулю генерації повинні пройти валідацію та попередню обробку. Далі отримані дані кешуються та подаються на вхід до модулю генерації.

На вхід до модулю генерації подаються:

- закешовані та оброблені дані тайлсету;
- границі сусідніх чанків;
- конфігурація генератора мапи.

Кешування даних тайлсету

Для збільшення швидкодії алгоритму запропоновано закешувати дані про сусідів кожного тайлу. Перед початком ітераційного процесу потрібно пройти усі елементи тайлсету та визначити «сусідів» поточного тайлу для кожного із напрямків.

Для кожного елементу тайлсету створюється множина можливих для підстановки сусідніх елементів:

$$\forall a_{ij} \in \mathrm{A}_{\mathrm{n}\times\mathrm{n}} \cap b_{ij} \in B_{n\times n} \begin{bmatrix} a_{1i}=b_{ni} \Rightarrow B \cup V \\ a_{ni}=b_{1i} \Rightarrow B \cup V \\ a_{i1}=b_{in} \Rightarrow B \cup V \\ a_{in}=b_{i1} \Rightarrow B \cup V \end{bmatrix}, \quad (2.2)$$

де A – це матриця вокселів обраного елементу;

B – матриця вокселів, із яким йде порівняння;

V – масив тайлів, що підходять для підстановки;

n – розмір тайлу;

i – індекс поточного вокселю (i = 1..n);

a_{in} у даному випадку позначає колір поточного вокселю тайлу. Якщо кольори вокселів суміжних сторін тайлів співпадають – тайл із яким йде порівняння додається до масиву можливих для підстановки сусідніх елементів.

Запропонований метод кешування даних тайлсету дозволить припбрати етап порівняння вокселів між тайлами під час ітераційного процесу. Порівняння вокселів тайлів буде зроблено лише один раз на початку процесу генерації.

Графічне зображення можливих сусідніх елементів до тайлу продемонстровано на рис. 2.10. До кожної із сторін тайлу створюється масив можливих сусідніх елементів.

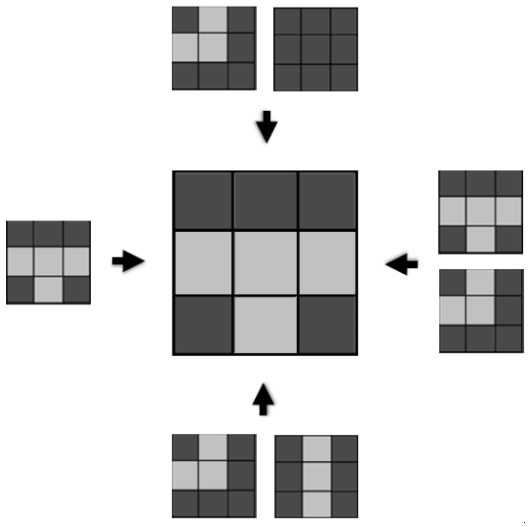


Рис. 2.10. Приклад можливих сусідніх елементів тайлу

Розроблено алгоритм побудови воксельного ландшафту із використанням комбінованого методу. Алгоритм продемонстровано на рис. 2.11. На діаграмі детально зображено 4 етап із методу генерації воксельного ландшафту, а саме етап генерації чанку комбінованим методом.

Перша спроба генерації здійснюється методом послідовної підстановки тайлів. Якщо результат генерації послідовним методом виявився негативним, тобто генерація не була успішною – здійснюється генерація за допомогою методу генерації із використанням колапсу хвильової функції. Якщо генерація за допомогою методу на основі WFC виявилася успішною – далі відбувається процес встановлення тайлів на мапу.

Швидкість генерації воксельної мапи за допомогою методу із використанням колапсу хвильової функції є значно меншою, проте надає значно більший шанс вірної генерації мапи.

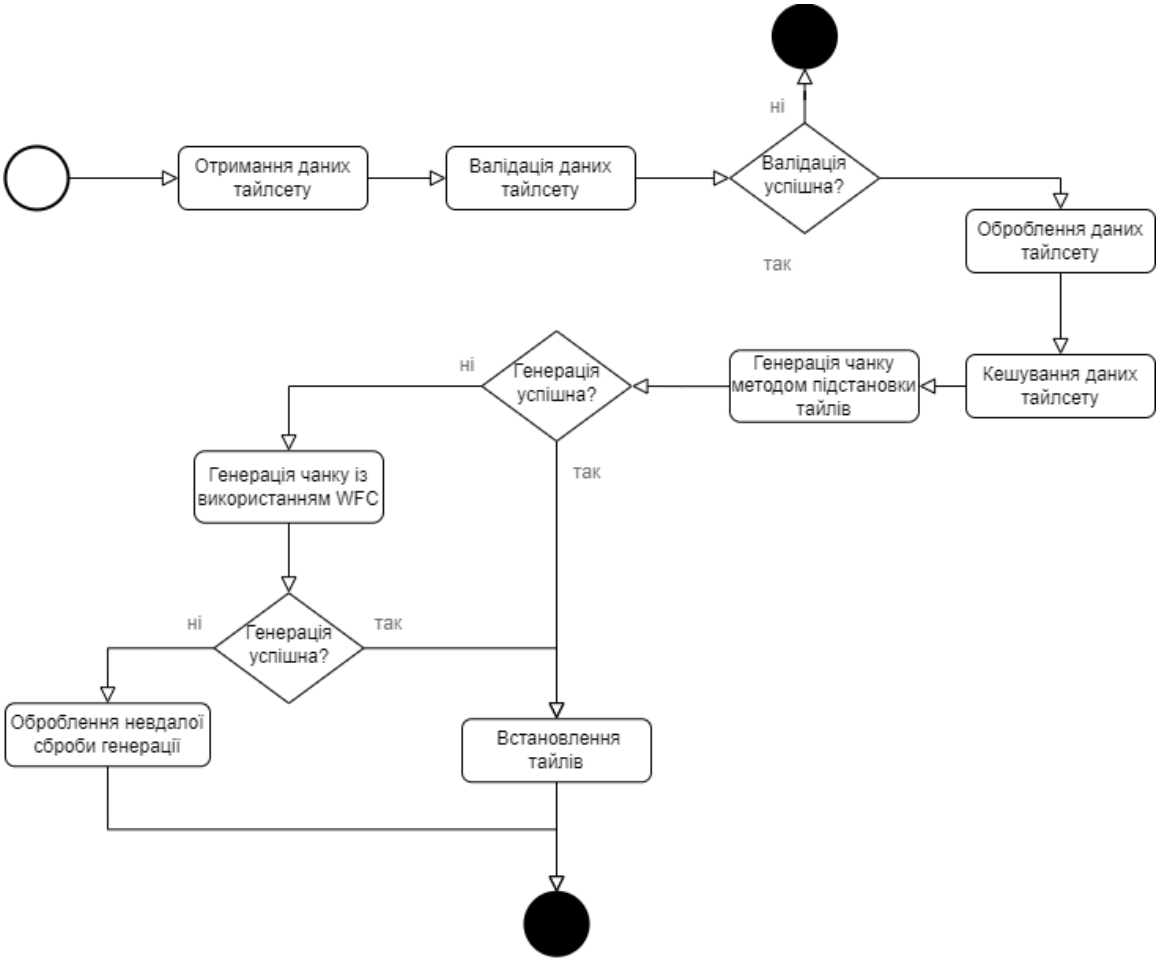


Рис. 2.11. Алгоритм побудови воксельного ландшафту

Особливості запропонованого методу

Особливістю запропонованого методу є використання поєднання генерації за допомогою методу послідовної підстановки тайлів та методу із використанням колапсу хвильової функції. Використання комбінованого методу генерації воксельної мапи надасть можливість досягти більшої швидкості та точності генерації воксельного ландшафту.

2.3. Висновки до розділу

У даному розділі було описано постановку задачі та вимоги до розроблюваного алгоритмічно-програмного методу, проаналізовано та наведено приклад роботи послідовного методу генерації воксельного ландшафту та методу генерації із застосуванням колапсу хвильової функції. Було запропоновано власний метод генерації воксельного ландшафту на основі 3D-тайлів на основі комбінації двох розглянутих методів генерації. У даному розділі було запропоновано архітектуру розроблюваної системи, а саме, наведено запропонований алгоритм побудови воксельного ландшафту та діаграму потоку даних запропонованого методу генерації.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДУ ГЕНЕРАЦІЇ ВОКСЕЛЬНОГО ЛАНДШАФТУ ІЗ 3D-ТАЙЛІВ

3.1. Обґрунтування вибору засобів реалізації ПЗ

Розробити програмного забезпечення можна у різному вигляді, як, наприклад, реалізувати програмне забезпечення у вигляді окремого додатку, чи у вигляді додаткового програмного забезпечення до ігрового рушія [13].

Засоби для реалізації алгоритмічно-програмного модулю генерації воксельного ландшафту мають відповідати наступним вимогам:

- забезпечувати можливість реалізації парадигми ООП;
- мати реалізовані інструменти для рендерингу графіки, зокрема роботи із променями (Raycasting);
- забезпечувати механізм асинхронності.

Результатом аналізу функціональних вимог до розроблюваного програмного забезпечення стало рішення реалізувати програмне забезпечення у вигляді додаткового програмного забезпечення до ігрового рушія.

Використання рушія замість реалізації окремого додатку для генерації надасть ряд переваг у певних випадках:

- Більшість функціоналу для роботи із комп’ютерною графікою та 3D-моделями вже реалізовано у рушіях.
- Можливість безпосередньої інтеграції згенерованої мапи із іншим контентом реалізованим у обраному рушії та окремому проєкті.

Було розглянуто та проаналізовано три найпопулярніші рушії, такі як: Unity, Unreal Engine та Godot.

Unity

Unity – міжплатформне середовище розробки комп'ютерних ігор, розроблене американською компанією Unity Technologies. Unity дозволяє створювати програми, що працюють на більш ніж 25 різних платформах, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-програми та інші [14].

Редактор Unity має простий Drag&Drop інтерфейс, а також встановлення плагінів KALI, який легко налаштовувати, що складається з різних вікон, завдяки чому можна проводити налагодження гри прямо в редакторі. Рушій Unity є одним із найпопулярніших рушіїв та використовує для написання скриптів C#.

C# – це мультипарадигменна мова загального призначення для розробки програм на платформі Microsoft [15]. Необхідною умовою роботи мови є встановлений у системі фреймворк .NET. C# розроблявся як мова програмування прикладного рівня для CLR, який надає C#, як і всім іншим .NET-орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування наприклад механізм збирання сміття (GC) [16].

Unreal Engine

Unreal Engine – ігровий рушій, що розробляється і підтримується компанією Epic Games. Написаний мовою C++, двигун дозволяє створювати ігри для більшості операційних систем та платформ [17]. Рушій підтримує можливість візуального програмування за допомогою техногології Blueprint [18].

C++ – це мультипарадигменна мова програмування загального призначення. Дана мова програмування має багату стандартну бібліотеку, яка включає поширені контейнери і алгоритми, введення-виведення, регулярні висловлювання, підтримку багатопоточності та інші можливості.

C++ поєднує властивості як високорівневих, і низькорівневих мов програмування. Дана мова програмування широко використовується для розробки програмного забезпечення, та є однією з найпопулярніших мов програмування [19].

Godot

Godot – відкритий кросплатформовий ігровий рушій під ліцензією MIT, який розробляється спільнотою Godot Engine Community [20]. Godot підтримує різноманітні мови програмування для створення ігор, включаючи інтегровану мову GDScript [21], C++ і C#. Архітектура рушія побудована навколо концепції дерева «вузлів». Вузли організовані всередині «сцен», які є повторно використовуваними, екземплярними, успадковуваними та вкладеними групами вузлів. Рушій також підтримує візуальне програмування із використанням вбудованої мови VisualScript, розробленої як візуальний еквівалент GDScript.

GDScript – високорівнева динамічно типізованої скриптової мови програмування. Дана мова програмування синтаксично схожа на Python. На відміну від Python, GDScript оптимізовано для архітектури та до мови додано можливість вказувати тип змінних.

Для реалізації запропонованого методу було обрано використати рушій Unity та мову програмування C#, тому фрагменти коду для демонстрації архітектури методу та структур даних будуть надані саме із використанням даного рушія та мови програмування.

3.2. Архітектура розроблюваної системи

Було розроблено діаграму запропонованого алгоритмічно-програмного методу для генерації воксельних ландшафтів, яку зображено на рис. 3.1.

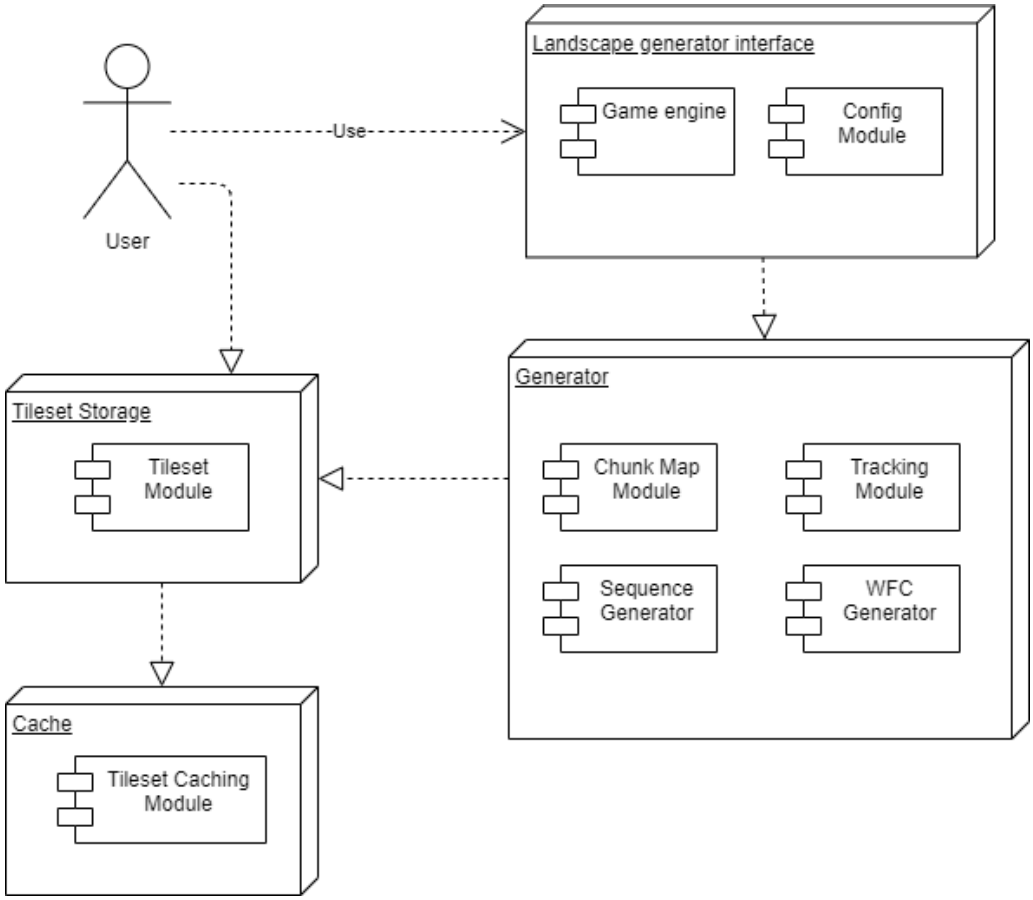


Рис. 3.1. Діаграма модулів розробленого алгоритмічно-програмного методу

Розроблена система складається із таких модулів:

- модуль інтерфейсу генератору;
- модуль генерації ландшафту;
- модуль зберігання тайлсетів;
- модуль кешування даних.

Модуль інтерфейсу генератору

Користувач алгоритмічно-програмного методу здійснює користування методом використовуючи даний модуль. Інтерфейс модулю може бути представлений у вигляді командного інтерфейсу (CLI), окремого додатку у вигляді графічного інтерфейсу користувача (GUI) або бути

інтегрованим у ігровий рушій, якщо генерація ландшафту відбувається у мужах ігрового рушія. У розробленому програмному забезпеченні користувацький інтерфейс надається у вигляді інтеграції розробленого скрипту у рушії Unity (рис. 3.2).

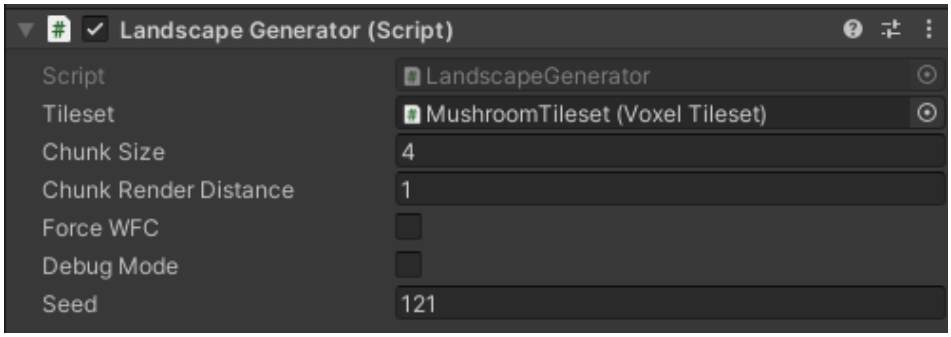


Рис. 3.2. Користувацький інтерфейс розробленого методу у рушії Unity

Модуль генерації ландшафту

Даний модуль відповідає безпосередньо за генерацію воксельного ландшафту із наданих 3D-тайлів. Даний модуль включає у себе наступні підмодулі: підмодуль збереження мапи чанків, підмодуль послідовної генерації, підмодуль генерації на основі колапсу хвильової функції та підмодуль відстеження (трекінгу), який надає функціональні можливості відстеження затрат часу на генерацію чанків для тестування процесу генерації та подальшої оптимізації тайлсетів та розробленого програмного забезпечення.

Модуль зберігання тайлсетів

Даний модуль відповідає за зберігання наданих користувачем тайлсетів. Модуль зберігання тайлсетів надає змогу змінити активний тайлсет для генерації та додавати змінювати тайли існуючих тайлсетів. Також даний модуль дозволяє здійснити конфігурацію тайлів, а саме задати значення повороту тайлів та вагу тайлів.

Модуль кешування даних

Даний модуль відповідає за процес кешування даних тайлсетів для збільшення швидкодії методу генерації під час ітераційного процесу генерації ландшафту. Даний модуль виконує процес кешування даних про сусідні елементи тайлсету під час запуску методу генерації воксельного ландшафту.

Діаграму класів розробленого програмного забезпечення зображено на рис. 3.3.

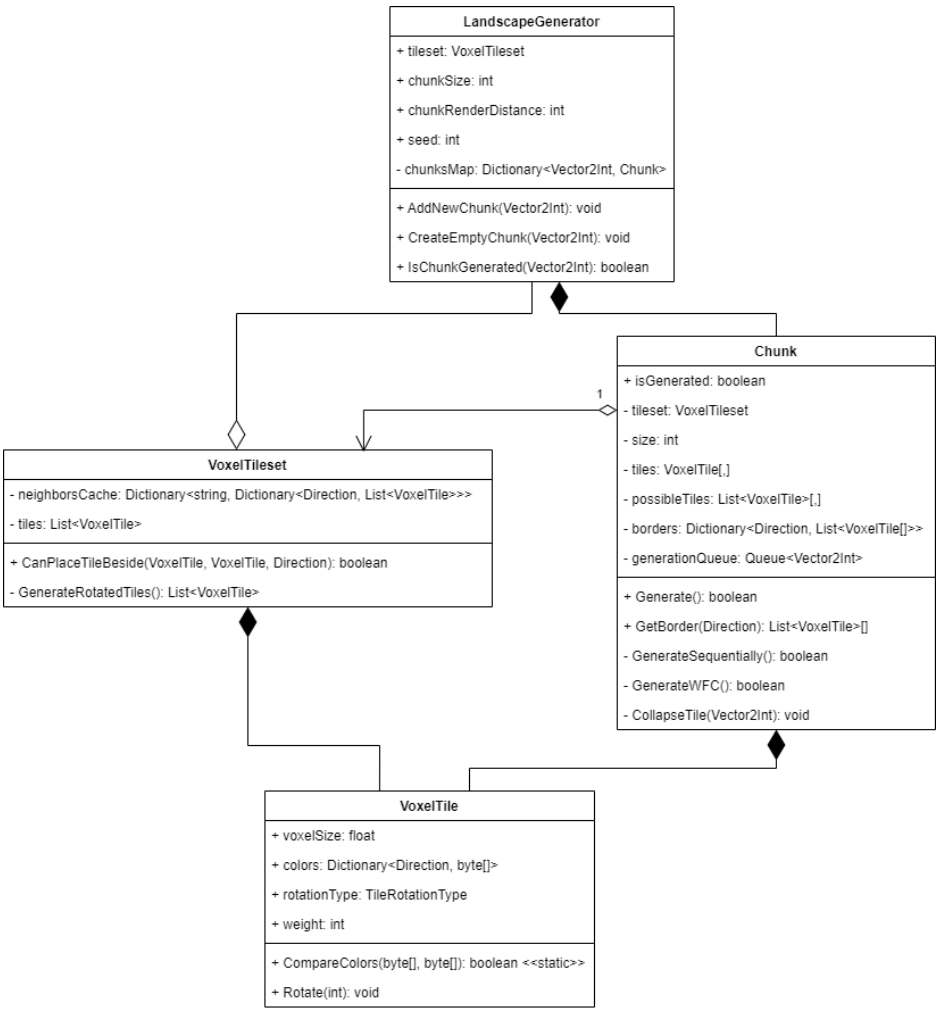


Рис. 3.3. Діаграма класів розробленого алгоритмічно-програмного методу

Клас **LandscapeGenerator** відповідає за запуск процесу генерації та зберігання даний про конфігурацію генератора та мапи чанків.

Клас **Chunk** відповідає за зберігання інформації про чанк воксельної мапи, а саме: матриця можливих тайлів, границі чанку та позиція чанку у координатах мапи чанків. Саме клас чанку реалізує методи генерації із застосуванням послідовного методу та методу генерації на основі WFC.

Клас **VoxelTileset** призначений для зберігання та кешування даних про воксельний тайлсет.

Клас **VoxelTile** призначений для зберігання інформації про воксельний тайл, а саме інформацію про:

- розмір вокселів у тайлі;
- кольори вокселів зовнішніх сторін тайлу;
- тип повороту тайлу;
- вагу тайлу.

Процес генерації воксельного ландшафту запропонованим методом можна розбити на декілька етапів:

- попередня обробка тайлів;
- аналіз та обробка мапи чанків;
- генерація чанку.

3.3. Попередня обробка тайлів

У ході імплементації запропонованого методу було використано моделі 3d-моделі із розширенням **.obj**. Всі необхідні моделі тайлів потрібно імпортувати до рушія.

Приклад моделі воксельного тайлу можна побачити на рис. 3.4.

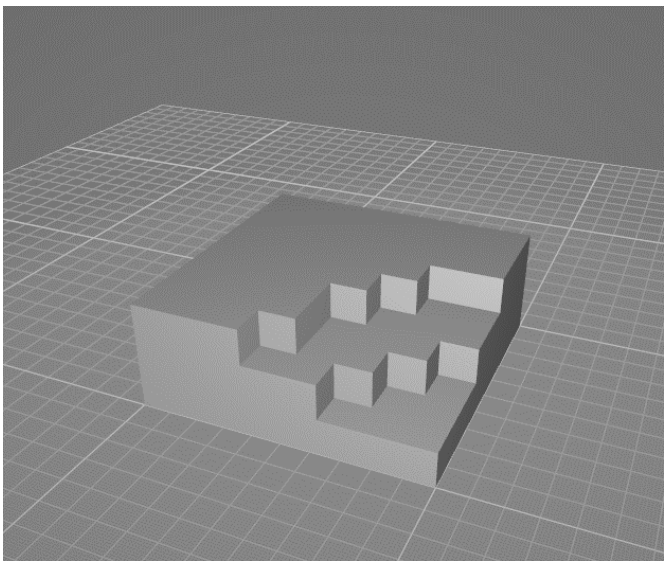


Рис. 3.4. Приклад моделі воксельного тайлу у форматі .obj

Для «фарбуваннння» моделей використовується колірний атлас тайлсету, який зберігається у вигляді зображення розміром 1 на 256 пікселів.

Приклад колірного атласу розміром 1 на 7 пікселів можна побачити на рис. 3.5.

Рис. 3.5. Приклад колірного атласу розміром 1 на 7 пікселів

Перед початком процесу генерації алгоритм повинен знати які тайли можна з’єднувати між собою.

Для забезпечення алгоритму цією інформацією потрібно дізнатися кольори всіх зовнішніх вокселів обраного тайлу.

Було створено клас **VoxelTile** для збереження даних про тайл та опису необхідних методів для оперування тайлами. Оскільки колірний атлас тайлсету складається із 256 кольорів, то один колір можна зберігати у змінній типу **byte**.

Для кожної зовнішньої сторони тайлу було створено масив кольорів вокселів, як показано в лістингу 3.1.

Лістинг 3.1. Оголошення полів класу для збереження масивів кольорів вокселів.

```
[HideInInspector] public byte[] ColorsRight;
[HideInInspector] public byte[] ColorsForward;
[HideInInspector] public byte[] ColorsLeft;
[HideInInspector] public byte[] ColorsBack;
```

Було вирішено зберігати ці значення у окремих масивах, а не у вигляді словника, щоб зберегти можливість робити копію об’єкту за допомогою shallow copy [22], що у свою чергу збільшить швидкодію копіювання об’єкту та убереже розробника від можливих проблем із реалізацією глибокої копії об’єкту.

Заповнити значення масивів кольорів вокселів зовнішніх сторін тайлів можна за допомогою техніки кидання променів (Ray casting) [23].

У кожний зовнішній тайл вокселю алгоритм кидає промінь та отримує і зберігає інформацію про колір кожного вокселю у позиції тайлу. Якщо вокселю у заданій позиції немає, то позиція позначається як пуста, тобто значення кольору у позиції буде дорівнювати нулю.

Лістинг 3.2. Отримання кольору вокселю

```
if (Physics.Raycast(new Ray(rayStart,
Direction.ConvertToVector3(direction)), out RaycastHit hit,
vox))
{
    byte colorIndex = (byte)(hit.textureCoord.x * 256);

    if (colorIndex == 0) Debug.LogWarning("Found color 0 in mesh
palette, this can cause conflicts");

    return colorIndex;
}
return 0;
```

На рис. 3.6 зображено приклад кидання променів на тайл, зеленим кольором позначено промені, які алгоритм кидає на тайл.

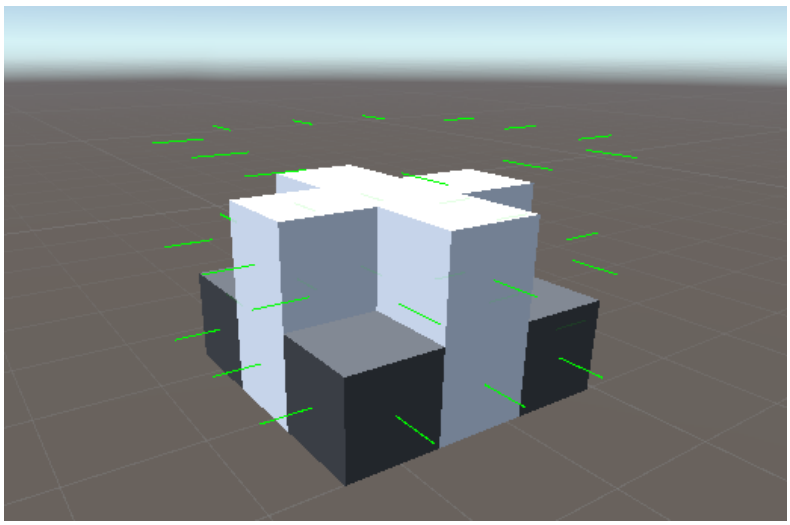


Рис. 3.6. Приклад кидання променів на тайл

У класі VoxelTile було реалізовано механізм повороту тайлів. Також було описано тип для позначення повороту тайлів: односторонній, двосторонній, та чотиристоронній відповідно, що можна побачити у лістингу 3.3.

Лістинг 3.3. Оголошення типу для позначення повороту тайлів.

```
public enum TileRotationType
{
    OneSided,
    TwoSided,
    FourSided
}
public TileRotationType RotationType =
TileRotationType.OneSided;
```

Перед початком генерації для двосторонніх тайлів потрібно створити копію із та повернути її на 180 градусів у площині Y, а для чотиристоронніх тайлів потрібно створити копії із кутами повороту 90, 180 та 270 градусів відповідно. Для односторонніх тайлів копії створювати не потрібно.

Приклад тайлсету зі створеними копіями для двосторонніх та чотиристоронніх тайлів можна побачити на рис. 3.7. З лівої сторони на рисунку зображено надані користувачем тайли, а з правої сторони зображено клоновані тайли які були згенеровані поворотом двосторонніх та

чотиристоронніх елементів тайлсету. Тобто користувачем на вхід було подано 6 тайлів, а ще 10 тайлів були згенеровані за допомогою повороту тайлів, які були подані на вхід тайлсету.

Рис. 3.7. Створення копій для двосторонніх та чотиристоронніх тайлів

Для того щоб порівняти кольори бокових сторін тайлів потрібно порівняти послідовно усі елементи масивів їх кольорів. Функцію для порівняння кольорів вокселів у тайлах наведено у лістингу 3.4.

Лістинг 3.4. Оголошення функції порівняння кольорів вокселів у тайлах.

```
public static bool CompareColors(byte[] colors1, byte[] colors2)
{
    if (colors1.Length != colors2.Length)
    {
        return false;
    }

    for (int i = 0; i < colors1.Length; i++)
    {
        if (colors1[i] != colors2[i])
        {
            return false;
        }
    }
    return true;
}
```

Для оперування тайлсетом було створено клас **VoxelTileset**. Даний клас містить у собі список усіх тайлів які входять до тайлсету, та кешовані [24] про можливих сусідів усіх тайлів у тайлсеті. Оголошення основних полів класу **VoxelTileset** наведено у лістингу 3.5.

Лістинг 3.5. Оголошення основних полів класу **VoxelTileset**.

```
[SerializeField] private List<VoxelTile> tilePrefabs = new
List<VoxelTile>();
public List<VoxelTile> TilePrefabs { get => tilePrefabs; }
private readonly NeighborsCache neighborsCache = new
NeighborsCache();
```

Поле **tilePrefabs –** зберігає у собі список тайлів які входять до даного тайлсету.

Поле **neighborsCache** зберігає у собі словник із усіма можливими сусідами усіх тайлів по всіх напрямках. Тобто для кожного елементу **tilePrefabs** за допомогою даного словника можна отримати масив можливих для підстановки сусідніх тайлів. Заповняється даний словник за допомогою методу **FillNeighborsCache**, який послідовно ітерується по заданим тайлам та заповнює словник із можливих сусідів. Тіло методу **FillNeighborsCache** наведено у лістингу 3.6.

Лістинг 3.6. Оголошення функції кешування даних про можливих сусідів усіх тайлів у тайлсеті.

```
private void FillNeighborsCache()
{
    List<VoxelTile> GetPossibleNeighbors(VoxelTile tile,
Direction3 direction)
    {
        var neighbors = new List<VoxelTile>();
        foreach (var tileToAppend in tilePrefabs)
        {
            if (CanAppendTile(tile, tileToAppend, direction))
            {
                neighbors.Add(tileToAppend);
            }
        }

        return neighbors;
    }
```

Продовження лістингу 3.6

```
        foreach (var tile in tilePrefabs)
        {
            neighborsCache[tile.Name] = new Dictionary<Direction3,
    List<VoxelTile>>();

            var directions = new Direction3[] { Direction3.Right,
    Direction3.Left, Direction3.Forward, Direction3.Back };
            foreach (var direction in directions)
            {
                neighborsCache[tile.Name][direction] =
    GetPossibleNeighbors(tile, direction);
            }
        }
    }
```

Таким чином під час ітераційного процесу генерації, алгоритму не потрібно кожен раз порівнювати кольори вокселів, алгоритм для порівняння тайлів буде використовувати словник **neighborsCache**.

3.4. Аналіз та обробка мапи чанків

Генерований ландшафт розбивається на певні частини заданого розміру, тобто – чанки [25].

Для оперування чанками було створено клас **Chunk**. Даний клас містить у собі поля пов’язані із конфігурацією генерації чанку, наприклад: розмір чанку, тобто скільки тайлів буде містити бокова сторона чанку, позиція чанку у координатах мапи чанків, тайлсет для генерації, масиви елементів границь із іншими чанками та інші внутрішні поля для налагоджування.

Для оперування генератором ландшафту було створено клас **LandscapeGenerator**. Даний клас містить у собі поля пов’язані із генерацією, наприклад: розмір чанку, дистанція рендеру чанків, зерно для генерації [26], тайлсет для генерації, мапу чанків та внутрішні поля для налагоджування генератора.

Клас **LandscapeGenerator** створює об’єкту класу **Chunk** та заносить їх до мапи чанків.

Мапа чанків – це словник який зберігає у собі об'єкти типу Chunk по ключу типу Vector2Int. Оголошення поля та тип об’єкту мапи чанків у класі **Chunk** наведено у лістингу 3.7.

Лістинг 3.7. Оголошення поля мапи чанків у класі **Chunk**

```
private readonly var _chunksMap = new Dictionary<Vector2Int,
Chunk>();
```

Мапа чанків зберігає не лише згенеровані чанки, але й ще незгенеровані чанки для яких вже створені сусіди. Зберігання ще не згенеровних чанків мапи потрібно для того, щоб частково зберігати границі цих чанків, та враховувати їх при генерації їх сусідів.

Приклад розподілу генерованої мапи на чанки продемонстровано на рис. 3.8.

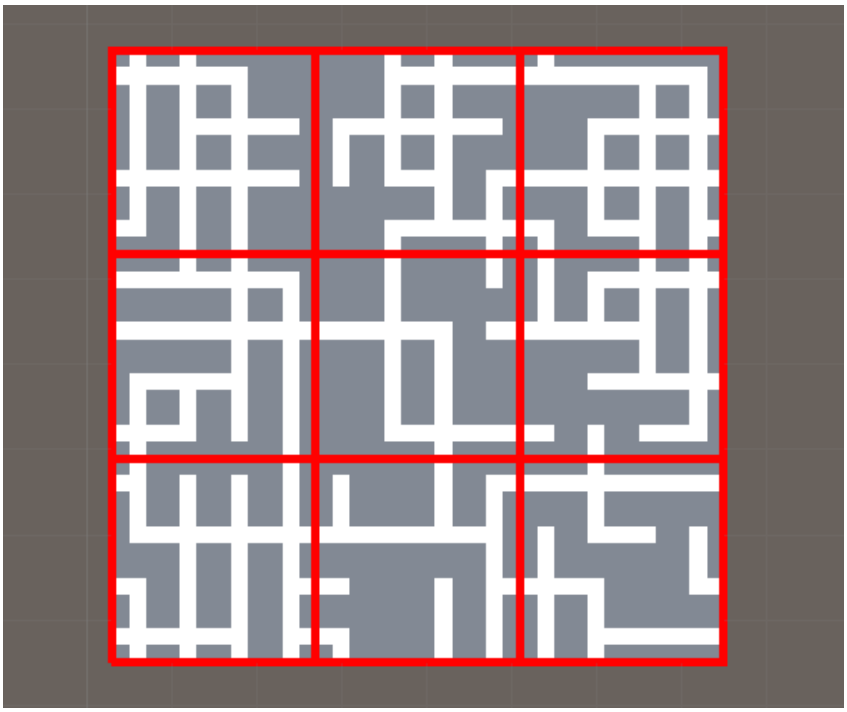


Рис. 3.8. Приклад розподілу мапи на чанки

Медот **AddChunk** реалізовує механізм додавання нового чанку до мапи чанків.

При генерації нового чанку до мапи чанків додається сам генерований чанк, якщо він ще не був доданий до мапи чанків, та сусідні до нього чанки. До ініціалізації сусідніх чанків додаються границі поточного чанку: до лівого чанку - права границя, до право чанку - ліва границя, до переднього чанку - задня границя та до заднього чанку - передня границя.

Заповнення границь чанків до початку їх генерації вирішує ряд проблем із генерацією тайлів у кутових елементах чанку. Приклад елементів границь поточного чанку наведено на рис. 3.9.

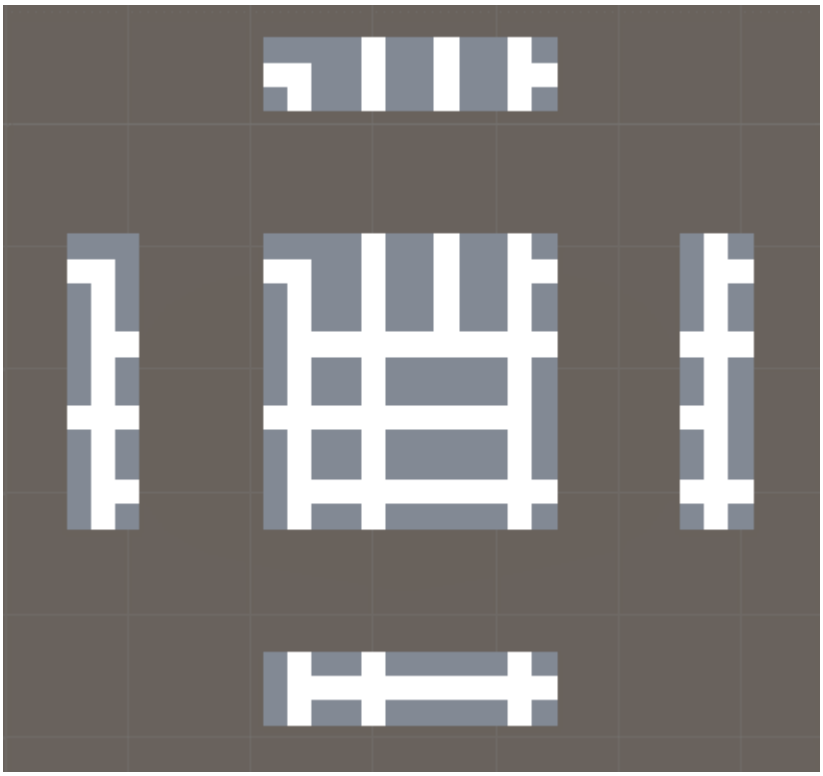


Рис. 3.9. Приклад елементів границь поточного чанку

3.5. Генерація чанку

Генерація чанку імплементована у методі **Generate** класу Chunk.

Даний метод використовує два приватні методи класу: **GeneratePossibleTilesBasic –** метод для генерації чанку без використання алгоритму WFC, та **GeneratePossibleTilesWFC** – метод для генерації чанку із використанням WFC.

Перед початком процесу генерації у об’єкті класу Chunk потрібно заповнити поля для конфігурації чанку, а саме:

- **size –** розмір чанку, тобто скільки тайлів буде складати бокова частина чанку, типу **int**;
- **position** – позиція генерованого чанку на мапі чанків, типу **Vector2Int**;
- **tileset** – обраний тайлсет для генерації, типу **VoxelTileset**;
- **borders** – словник границь поточного чанку, типу **Dictionary<Direction2, List<VoxelTile>[]>**

У класі **Chunk** оголошено також приватні поля **_possibleTiles** та **_borders**.

Поле **_possibleTiles** зберігає матрицю можливих тайлів разом із елементами границь сусідніх чанків. Дана матриця має розмір який дорівнює змінній **size** збільшений на 2. Кутові елемент даної матриці залишаються незадіяними у генераційному процесі.

На рис. 3.10. продемонстровано приклад графічного зображення матриці можливих тайлів. Зеленим кольором позначено границі сусідніх чанків, продемонстровано заповнення нижньої та лівої границь сусідніх чанків.

Варто зазначити, що границі матриці можливих тайлів можуть бути пустими, це відбувається у випадках, коли здійснюється генерація чанку у якого немає суміжного чанку хоча б у одній із сторін. У продемонстрованому прикладі права та верхня границі сусідніх не містять жодного елементу.

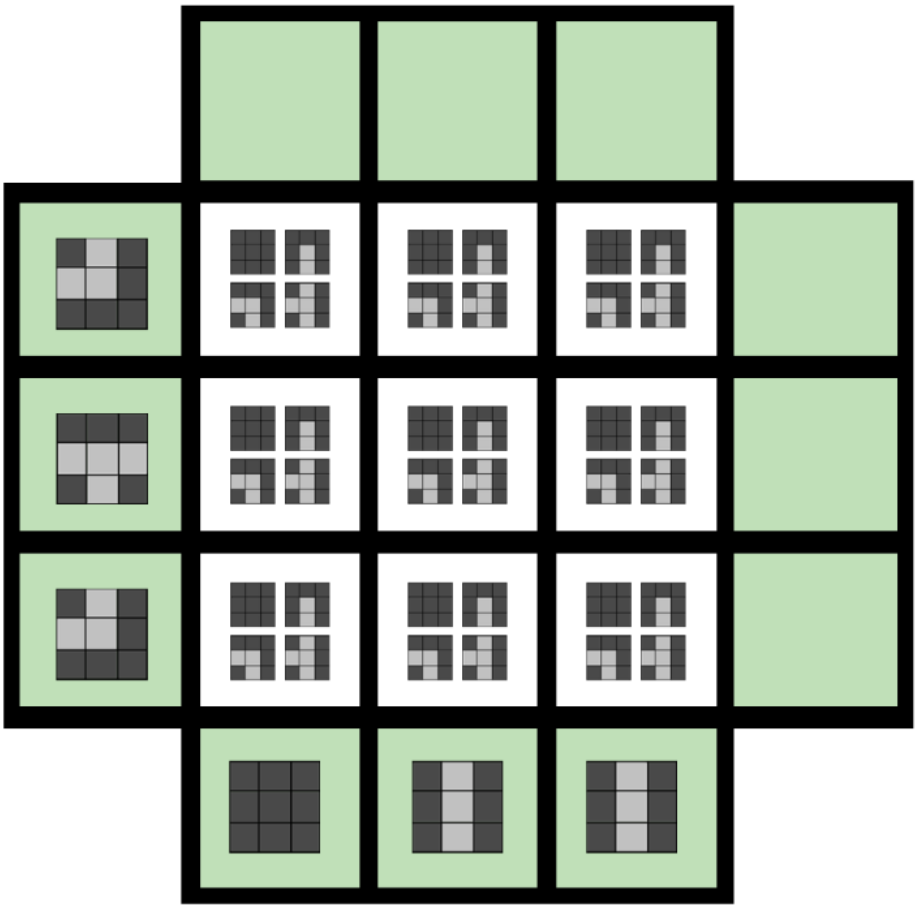


Рис. 3.10. Приклад елементів границь поточного чанку

Генерація за допомогою WFC

Алгоритм генерації імплементовано згідно опису у пункті 2.2.2.

Для забезпечення послідовного проходу елементів та імплементації черги використовується поле **_generationQueue** типу **Queue<Vector2Int>**, що продемонстровано у лістингу 3.8.

Лістинг 3.8. Оголошення поля черги генерації тайлів у класі **Chunk**

```
private readonly Queue<Vector2Int> _generationQueue = new
Queue<Vector2Int>();
```

Процес генерації починається з того, що черга генерації тайлів очищується, та до черги **_generationQueue** додаються кутові елементи, що продемонстровано у лістингу 3.9.

Лістинг 3.9. Початкове заповнення черги **_generationQueue** кутовими елементами чанку.

```
_generationQueue.Clear();
_generationQueue.Enqueue(new Vector2Int(1, 1));
_generationQueue.Enqueue(new Vector2Int(1, _chunkSize));
_generationQueue.Enqueue(new Vector2Int(_chunkSize, 1));
_generationQueue.Enqueue(new Vector2Int(_chunkSize,
_chunkSize));
```

Варто зазначити, що елементи границь сусідніх чанків, тобто границі матриці **_possibleTiles**, або матриці можливих чанків, ніколи не потрапляють у чергу генерації, оскільки черга це призведе до зміни елементів границь сусідніх чанків які були передані у конструктор класу та некоректної генерації чанку у подальшому.

Далі у кожному елементі матриці можливих елементів чанку заповнюється список доступних тайлів **_possibleTiles**. Елементи матриці заповнюються послідовно рядок за рядком у вкладеному циклі. Внаслідок обходу матриці кожна її клітинка містить у собі список всіх доступних тайлів із наданого на вхід алгоритму тайлсету. Обхід матриці та заповнення всіх її клітинок списком доступних тайлів продемонстровано у лістингу 3.10.

Лістинг 3.10. Оголошення методу **PrefillPossibleTiles** який імплементує заповнення тайлів зі списку усіх доступних тайлів.

```
private void PrefillPossibleTiles()
{
    for (int y = 1; y <= _chunkSize; y++)
    {
        for (int x = 1; x <= _chunkSize; x++)
        {
            _possibleTiles[x, y] = new
List<VoxelTile>(_tileset.TilePrefabs);
        }
    }
}
```

Також заповнюються граничні елементи матриці можливих тайлів елементами границь поточного чанку.

Лістинг 3.11. Оголошення методу **FillBorders** який імплементує заповнення границь поточного чанку.

```
private void FillBorders()
{
    if (!(_borders[Direction2.Bottom] is null))
        for (int i = 1; i <= _chunkSize; i++)
            _possibleTiles[i, 0] =
_borders[Direction2.Bottom][i];

    if (!(_borders[Direction2.Top] is null))
        for (int i = 1; i <= _chunkSize; i++)
            _possibleTiles[i, _chunkSize + 1] =
_borders[Direction2.Top][i];

    if (!(_borders[Direction2.Left] is null))
        for (int i = 1; i <= _chunkSize; i++)
            _possibleTiles[0, i] = _borders[Direction2.Left][i];

    if (!(_borders[Direction2.Right] is null))
        for (int i = 1; i <= _chunkSize; i++)
            _possibleTiles[_chunkSize + 1, i] =
_borders[Direction2.Right][i];
}
```

Далі розпочинається ітераційний процес у якому, під час кожної ітерації, фіналізується щонайменше один тайл із матриці можливих тайлів.

У циклі викликається функція для обробки черги можливих для генерації тайлів, тобто поки черга не буде оброблена та повністю очищена – алгоритм не переходить до наступної ітерації.

Лістинг 3.12. Умова переходу алгоритму до наступної ітерації.

```
while (_generationQueue.Count > 0 && innerIterations++ <
maxInnerIterations)
{
    ProcessTileInGenerationQueue();
}
```

Метод **ProcessTileInGenerationQueue** валідує позицію тайла у черзі, перевіряючи чи позиція знаходиться всередині чанку та чи позиція не є вже фіналізованою, та реалізує видалення всіх неможливих для підстановки елементів у поточному елементі матриці можливих тайлів.

Лістинг 3.13. Валідація позиції тайлу у черзі генерації всередині методу **ProcessTileInGenerationQueue**

```
private void ProcessTileInGenerationQueue()
{
    Vector2Int position = _generationQueue.Dequeue();
    if (!IsPositionInsideChunk(position))
        throw new Exception($"Position is not inside the chunk:
{position}");

    if (IsTileGeneratedInPosition(position))
        return;

    int numberOfRemovedTiles = RemoveImpossibleTiles(position);
```

Якщо у поточному елементі матриці можливих не залишилось жодного елемента, тобто у поточну позицію не можна встановити жоден із тайлів, то поточний елемент та усі сусідні елементи заносяться у чергу генерації та елементи у матриці можливих тайлів встановлюються усі можливі тайли, тобто обнуляються.

Лістинг 3.14. Обробка елементу матриці можливих тайлів, у якому не залишилось жодного елементу.

```
if (_possibleTiles[position.x, position.y].Count == 0)
{
    // no possible tiles, try to recalculate this tile and
neighbors
    _possibleTiles[position.x,
position.y].AddRange(_tileset.TilePrefabs);

    void ResetTilesInNeighbor(Direction2 direction)
    {
        var neighborPosition = position +
Direction.ConvertToVector(direction);
        if (IsPositionInsideChunk(neighborPosition))
            _possibleTiles[neighborPosition.x,
neighborPosition.y] = new List<VoxelTile>(_tileset.TilePrefabs);
    }

    ResetTilesInNeighbor(Direction2.Right);
    ResetTilesInNeighbor(Direction2.Left);
    ResetTilesInNeighbor(Direction2.Top);
    ResetTilesInNeighbor(Direction2.Bottom);
    _generationQueue.Enqueue(position);
    AddNeighborsToGenerationQueue(position);

    _backtracks += 1;
}
```

Якщо було видалено хоч один елемент у поточному елементі матриці можливих тайлів, то до черги генерації додаються сусіди поточного елементу:

Лістинг 3.15 Додавання до черги генерації сусідів поточного елементу.

```
else if (numberOfRemovedTiles > 0)
{
    AddNeighborsToGenerationQueue(position);
}
```

Таким чином у кожній ітерації здійснюється рекурсивний обхід усіх сусідів елементів матриці у яких були модифіковані елементи можливих для підстановки тайлів.

Коли всі тайли із черги оброблено, алгоритм фіналізує один із елементів матриці можливих тайлів. Для цього обирається елемент із найменшою кількістю тайлів та обирається один тайл із можливих. Також ітераційний процес переривається коли елемент із найбільшою кількість тайлів налічує лише один тайл.

Лістинг 3.16. Фіналізація та встановлення елементу матриці можливих тайлів.

```
var maxCountTilePosition = GetPositionWithMaxPossibleTiles();
var maxCountTile = _possibleTiles[maxCountTilePosition.x,
maxCountTilePosition.y];

if (maxCountTile.Count == 1)
{
    return true;
}

var minCountTilePosition = GetPositionWithMinPossibleTiles();
CollapseTile(minCountTilePosition);
```

Функція **CollapseTile** обирає один тайл зі списку можливих тайлів у заданому елементі матриці. Також після встановлення єдиного тайлу у елемент матриці, метод додає сусідів поточного елементу до черги генерації.

Лістинг 3.17 Функція **CollapseTile** яка реалізує обирання одного тайл зі списку можливих тайлів у заданому елементі матриці.

```
/// <summary>
/// Collapse possible tiles in position to one tile
/// </summary>
/// <param name="position">Position in chunk to collapse</param>
private void CollapseTile(Vector2Int position)
{
    VoxelTile tileToCollapse =
GetRandomTile(_possibleTiles[position.x, position.y]);
    _possibleTiles[position.x, position.y] = new List<VoxelTile>
{ tileToCollapse };
    AddNeighborsToGenerationQueue(position);
}
```

Ітераційний процес вважається завершеним після того, як у всіх елементах матриці залишилось по одному тайлу. Далі алгоритм просто створює екземпляри моделей об’єктів та встановлює їх у відповідні позиції, після чого чанк можна вважати згенерованим.

Лістинг 3.18. Функція **PlaceTile** для встановлення тайлу до мапи.

```
private void PlaceTile(int x, int y)
{
    List<VoxelTile> availableTiles = _possibleTiles[x, y];

    if (availableTiles.Count == 0) return;

    var selectedTile = GetRandomTile(availableTiles);
    var localTilePosition = new Vector3(x, 0, y) *
_tileset.TileSize;
    var chunkPosition = new Vector3(_position.x, 0, _position.y)
* _chunkSize * _tileset.TileSize;
    var position = chunkPosition + localTilePosition;
    _tiles[x, y] = GameObject.Instantiate(selectedTile,
position, selectedTile.transform.rotation);
}
```

Якщо алгоритм було перервано через брак ітерацій, або неможливість встановлення тайлів, тобто список доступних тайлів у поточному елементі матриці буде пустим, то дана позиція чанку залишається пустою.

Генерація за допомогою послідовного методу

Для покращення швидкості алгоритму було імплементовано також генерацію чанку за допомогою послідовного методу генерації, тобто за допомогою послідовного обходу матриці можливих елементів.

Даний метод обходить матрицю можливих тайлів за допомогою вкладеного циклу, тобто рядок за рядком. Метод перевіряє чи можна у встановити у поточний елемент матриці хоча б один із елементів тайлсету, базуючись на тайли, які вже встановлено у сусідні елементи матриці.

Лістинг 3.19. Функція **GeneratePossibleTilesBasic** яка реалізує базовий метод генерації чанку.

```
private bool GeneratePossibleTilesBasic()
{
    FillBorders();
    for (int y = 1; y < _chunkSize + 1; y++)
        for (int x = 1; x < _chunkSize + 1; x++)
        {
            List<VoxelTile> availableTilesInCell = new
List<VoxelTile>();
            foreach (VoxelTile tilePrefab in
_tileset.TilePrefabs)
            {
                if ((_possibleTiles[x - 1, y] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x - 1, y][0],
tilePrefab, Direction3.Left)) &&
                    (_possibleTiles[x + 1, y] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x + 1, y][0],
tilePrefab, Direction3.Right)) &&
                    (_possibleTiles[x, y - 1] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x, y - 1][0],
tilePrefab, Direction3.Back)) &&
                    (_possibleTiles[x, y + 1] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x, y + 1][0],
tilePrefab, Direction3.Forward)))
                    availableTilesInCell.Add(tilePrefab);
            }
            if (availableTilesInCell.Count == 0)
                return false;
           var selectedTile =
GetRandomTile(availableTilesInCell);
            _possibleTiles[x, y] = new List<VoxelTile> {
selectedTile };
        }
    return true;
}
```

Метод повертає булеве значення, що дозволяє простежити, чи була спроба згенерувати чанк успішною.

У випадку якщо спроба генерації чанку базовим методом була не успішною алгоритм використовує метод генерації із використанням WFC. Це дасть змогу надати першу спробу генерації простому алгоритму, та, якщо він його результат не буде успішним, буде застосований більш часо та ресурсозатратний, але надійніший алгоритм генерації чанку.

Додавання ваги тайлів

Для розширення спектру можливих ландшафтів для генерації було додано можливість змінювати відсоток шансу генерації кожного окремого тайлу у тайлсеті.

Вагою тайлу вважається ціле число від 1 до 100, яке користувач може задати для кожного тайлу із тайлсету. За замовчуванням вага всіх тайлів у тайлсеті дорівнює 50.

Лістинг 3.20. Оголошення поля ваги тайлу у класі **VoxelTile**

```
[Range(1, 100)]
public int Weight = 50;
```

Варто зауважити, що для двосторонніх та чотиристоронніх тайлів вага ділиться автоматично на 2 та 4 відповідно, оскільки потрібно зберегти початковий шанс генерації. Тобто якщо користувач створює двосторонній тайл із вагою 40, то фактично створюються два тайли, кожен із яких має вагу яка дорівнює 20.

Вага тайлу враховується при виборі одного тайлу з поміж інших можливих варіантів, наприклад на етапі колапсування тайлу, коли у клітинці поточного елементу матриці кількість можливих тайлів більша за 1.

Лістинг 3.21. Оголошення функції **GetRandomTile** для отримання єдиного тайлу із масиву можливих тайлів.

```
private static VoxelTile GetRandomTile(List<VoxelTile>
availableTiles)
{
    var chances = new List<float>();
    foreach (var tile in availableTiles)
    {
        chances.Add(tile.Weight);
    }

    float value = Random.Range(0, chances.Sum());
    float sum = 0;

    for (int i = 0; i < chances.Count; i++)
    {
        sum += chances[i];
        if (value < sum)
        {
            return availableTiles[i];
        }
    }

    return availableTiles[0];
}
```

Таким чином користувач може більш точно коригувати налаштування генератора, додаючи шанс появи певних тайлів більше ніж іншим.

3.6. Висновки до розділу

У даному розділі було обґрунтовано вибір засобів реалізації розроблюваного програмного забезпечення. Була наведена архітектура розроблюваної системи із детальним описом кожного окремого модулю. У даному розділі було описано основні етапи генерації воксельного ландшафту: попередня обробка тайлів, аналіз та обробка мапи чанків та генерація чанку. У розділі наведено діаграму класів розробленого програмного забезпечення та приклади програмної реалізації окремих етапів алгоритмічно-програмного методу.

4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МОЖЛИВІ ПОКРАЩЕННЯ

4.1. Тестування модуля генерації воксельного ландшафту

Для тестування розробленого алгоритмічно-програмного методу було створено спеціальне тестове середовище у вигляді тестової сцени за допомогою засобів ігрового рушію Unity. Для тестування генератора, було додано можливість пересуватися камерою по сцені та можливість додавати генерувати нові чанки при зміні положення камери. Було додано функціональні можливості для трекінгу часу генерації чанків. Для тестування розробленого алгоритмічно-програмного методу було створено декілька тестових воксельних тайлсетів.

Клас **LandscapeGenerator** має певну кількість публічних полів які користувач може конфігурувати. Такими полями є (рис. 4.1):

- **tileset** – тайлсет який було обрано для генерації, типу VoxelTileset;
- **chunkSize** – розмір чанку, дане значення позначає бічну кількість тайлів у чанку;
- **chunkRenderDistance** – кількість чанків які будуть генеруватися у всіх напрямках при пересуванні камери. Наприклад, при встановленому значенні яке дорівнює трьом, клас **LandscapeGenerator** буде генерувати мапу глибиною у 3 чанки у всі сторони від камери;
- **forceWFC** – поле яке дозволяє вимкнути спробу генерації чанку за допомогою базового методу генерації. При встановленому значенні **true** для генерації чанку буде використовуватися лише метод із використанням колапсу хвильової функції;
- **debugMode** – при встановленому значенні **true** генератор буде відображати границі чанків;
- **seed** – значення зерна для генератора.

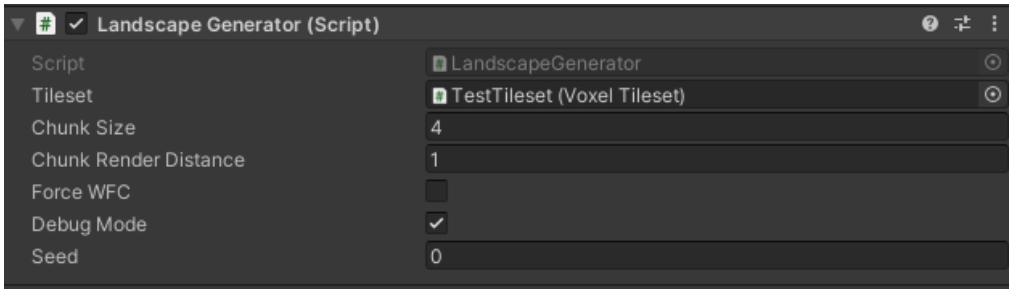


Рис. 4.1. Доступні для конфігурації поля класу **LandscapeGenerator**.

Для тестування методу було використано 3 різні тайлсети, які мають різну кількість тайлів.

Тестовий Тайлсет №1

Тайлсет №1 складається із шести тайлів, два з яких є односторонніми, один є двостороннім та ще три є чотиристоронніми:

- надана кількість тайлів: 6;
- фактична кількість тайлів: 16;
- розмір тайлу: 3 вокселів.

Всі елементи тестового тайлсету №1 зображено на рис. 3.4.

Тестовий Тайлсет №2

Тайлсет №2 складається із шести тайлів, два з яких є односторонніми, та ще чотири є чотиристоронніми:

- надана кількість тайлів: 6;
- фактична кількість тайлів: 18;
- розмір тайлу: 8 вокселів.

Всі елементи тестового тайлсету №2 зображено на рис. 4.2.

Тестовий Тайлсет №3

Тайлсет №3 складається із двадцяти шести тайлів, чотири з яких є односторонніми, та ще двадцять два є чотиристоронніми:

- надана кількість тайлів: 26;
- фактична кількість тайлів: 92;
- розмір тайлу: 16 вокселів.

Всі елементи тестового тайлсету №3 зображено на рис. 4.3.

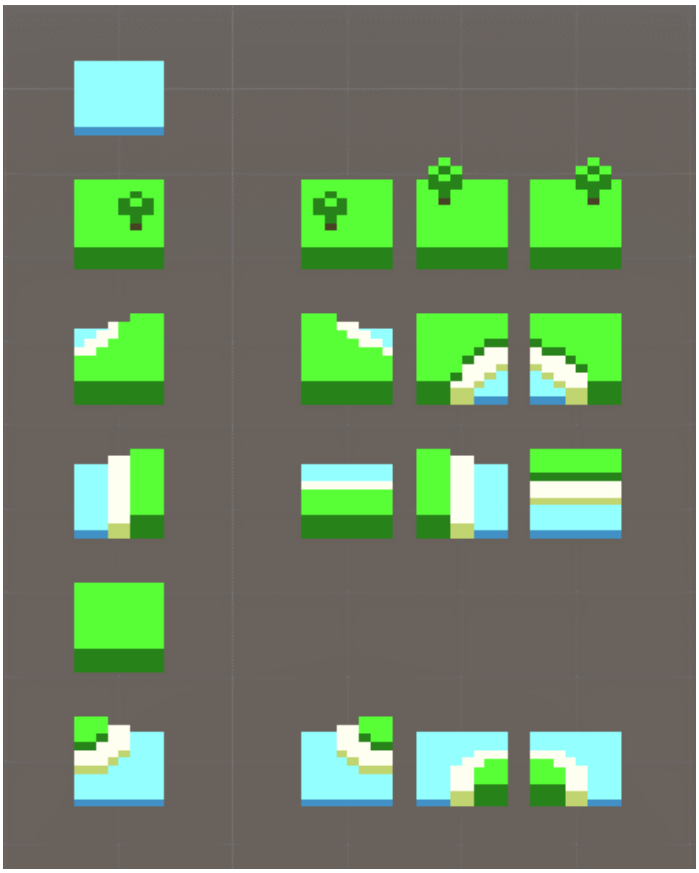


Рис. 4.2. Тестовий тайлсет №2

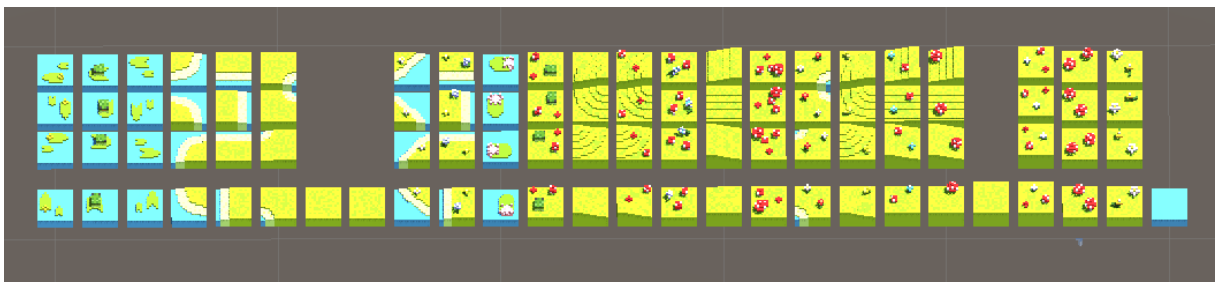


Рис. 4.3. Тестовий тайлсет №3

Під час тестування розробленого алгоритмічно-програмного методу було зроблено по 10 тестових запусків генератори без змін конфігурації генератора.

Задана конфігурація класу **LandscapeGenerator** під час тестування:

- **chunkSize**: 8 тайлів;
- **chunkRenderDistance**: 16 чанків у всіх напрямках;
- **debugMode**: вимкнений;
- **seed**: 0.

4.2. Аналіз отриманих результатів

У ході роботи було протестовано 3 варіанти розробленого методу:

- метод послідовної підстановки – метод який використовує послідовну підстановку тайлів та не використовує WFC;
- метод із застосуванням тільки WFC – метод який використовує тільки колапс хвильової функції;
- комбінований метод – метод у якому перша спроба генерації надається послідовному методу підстановки тайлів, та у випадку невдачі, застосовується метод генерації на основі колапсу хвильової функції.

Внаслідок застосування методу колапсу хвильової функції, середній час генерації воксельних ландшафтів зростає (табл. 4.1) у порівнянні з базовим методом генерації (без імплементації WFC-алгоритму). Натомість, запропонований метод забезпечує меншу вірогідність того, що чанк згенерується невірно.

Порівняти базовий метод та метод із застосуванням WFC напряму ми можемо тільки на прикладі першого тайлсету, оскільки у всіх інших тестових випадках не всі чанки мапи можна згенерувати базовим методом. Можемо побачити, що середній час генерації у базового методу істотно нижчий ніж у методу із застосуванням WFC – 2.363мс та 9.691мс відповідно.

Таблиця 4.1

Оцінка часової ефективності методів, мс

Номер тайлсету	Метод послідовної підстановки (без застосування WFC)	Комбінований метод	Метод із застосуванням WFC
1	2.363мс	2.363мс, 0 чанків WFC	9.691мс
2	-	4.161мс, 182 чанки WFC	11.559мс
3	-	6450.841мс, 689 чанки WFC	8440.334мс

Також можемо побачити, що при застосуванні комбінованого методу генерації на тайлсетах №2 та №3 є значна кількість чанків яка була згенерована за допомогою WFC методу. Можемо побачити що це число істотно зросло від 182 до 689 при зміні тайлсету із №2 на №3. Можемо зробити висновок, що при збільшенні розміру тайлсету, зменшується шанс на генерацію тайлу за допомогою базового методу генерації.

Тестування механізму кешування

Було проведено порівняльний аналіз часової ефективності методів із застосуванням кешування даних про сусідів усіх елементів тайлсету та без застосування механізму кешування (табл. 4.2). Для тестування кешування було застосовано комбінований метод генерації. Конфігурація генератора та тайлсетів залишалася незмінною з тестування оцінки часової ефективності методів

Таблиця 4.2

Оцінка часової ефективності кешування даних про сусідів

Номер тайлсету	Комбінований метод без кешування даних про сусідів	Комбінований метод із застосуванням механізму кешування
1	2.363мс	2.551мс
2	4.161мс	4.033мс
3	6450.841мс	6234.727 мс

Можемо побачити що на тайлсеті №1 середній час генерації чанку виріс із 2.363мс до 2.551мс, проте на тайлсетах №2 та №3 час генерації зменшився. Це може свідчити, що доречно використовувати кешування даних про сусідів елементів тайлсету у тих випадках, коли більшість чанків генерується за допомогою WFC-алгоритму. Застосування механізму кешування на базовому методі демонструє негативний вплив на швидкість алгоритму.

Тестування роботи ваги тайлів

Окрім часових характеристик генератора також було протестовано імплементацію зміни ваги тайлів.

Для тестування ваг було згенеровано мапу на основі конфігурації ваг у тайлсеті. На рис. 4.4 пронумеровано тестовий тайлсет №2 для подальшого тестування зміни ваги окремих елементів.

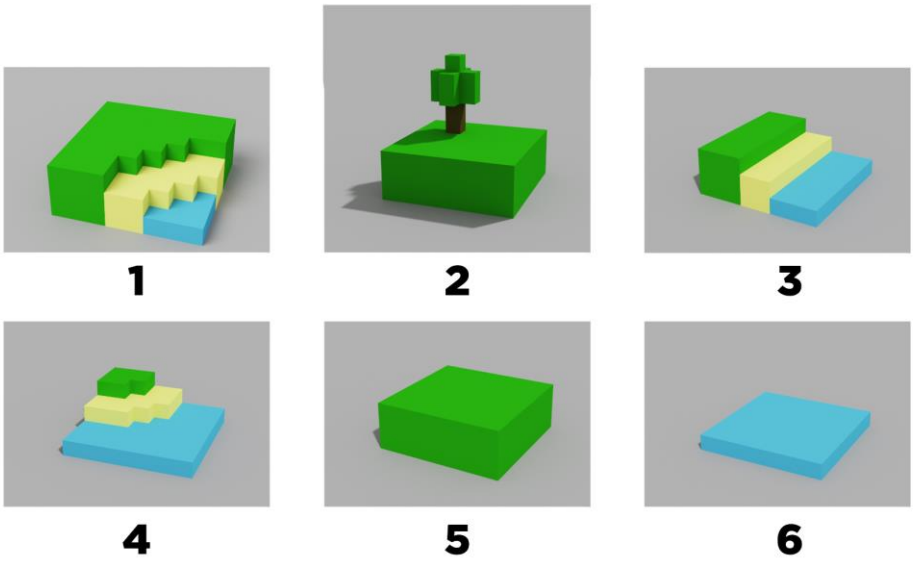


Рис. 4.4. Пронумеровані елементи тестового тайлсету №2

Було протестовано декілька конфігурацій налаштування ваги тайлів для обраного тайлсету. У табл. 4.3 наведено значення ваг тайлів для тестових випадків.

Таблиця 4.3

Значення ваги тайлів у тестових прикладах

Номер тайлу	Вага тайлу (від 1 до 100)	
	Тестовий приклад №1	Тестовий приклад №2
1	20	50
2	70	10
3	20	60
4	20	70
5	100	20
6	20	80

У тестовому прикладі №1 всім тайлам які мають зелені вокселі було назначено вагу у 20 балів. Таким чином кількість тайлів яка містить бакитні

вокселі має бути більша ніж кількість тайлів із зеленими вокселями. Згенерований чанк за допомогою даної конфігурації ваг можна побачити на рис. 4.5.

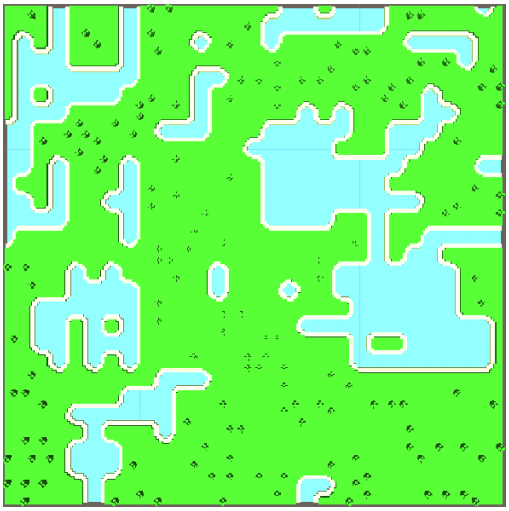


Рис. 4.5. Результат генерації чанку у тестовому прикладі №1

У тестовому випадку №2 тайлам із зеленими вокселями було назначено вагу у 10 да 20 балів, а тайлам які мають блакитні вокселі було назаначено вагу більше ніж 50 балів кожному. Таким чином кількість тайлів яка містить блакитні води має бути більша ніж кількість тайлів із зеленими вокселями. Згенерований чанк за допомогою даної конфігурації ваг можна побачити на рис. 4.6.

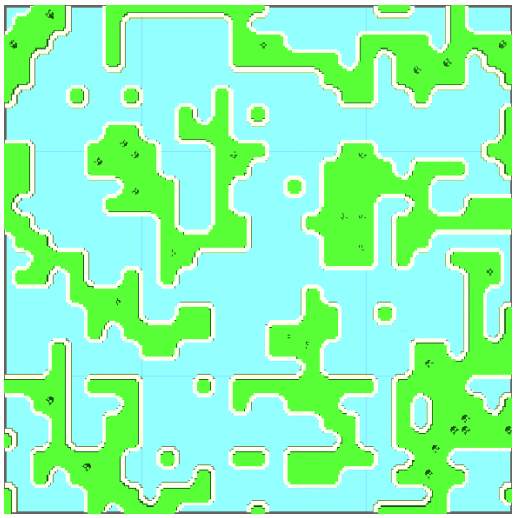


Рис. 4.6. Результат генерації чанку у тестовому прикладі №2

Можемо побачити, що генерація чанків була здійснена згідно виставленим вагам тайлів. У мапі згенерованій у тестовому прикладі №1 домінують тайли із зеленими вокселями, а у мапі згенерованій у тестовому прикладі №2 домінують тайли із блакитними вокселями.

Також були протестовані граничні значення ваги тайлів, тобто всім тайлам у тестовому тайлсеті було призначено вагу яка дорівнювала одиниці, та у другому тестовому випадку всім тайлам було призначено вагу яка дорівнювала 100. Якщо всі тайли тайлсету мають вагу яка дорівнює максимальному значенню – тайли розставляються випадково, із однаковим шансом їх вибору. Проте якщо всі тайли тайлсету мають вагу яка дорівнює мінімально можливому значенню – генератор починає генерувати повторювані патерни із початкового чанку та найбільший пріоритет буде мати тайл який має останній індекс у масиві тайлів тайлсету.

Тестування генерації із різним зерном

Також було протестовано генерацію чанків мапи із використанням різного зерна генерації.

Оскільки зерно генерації визначено значенням типу **integer**, то існує більше двох мільярдів варіантів генерації початкового чанку мапи.

Було виконано декілька спроб генерації із різним значенням зерна. Таким чином було виконано генерацію чанку із значенням зерна яке дорівнювало «121» (рис. 4.7) та значенням зерна яке дорівнювало «512623» (рис. 4.8).

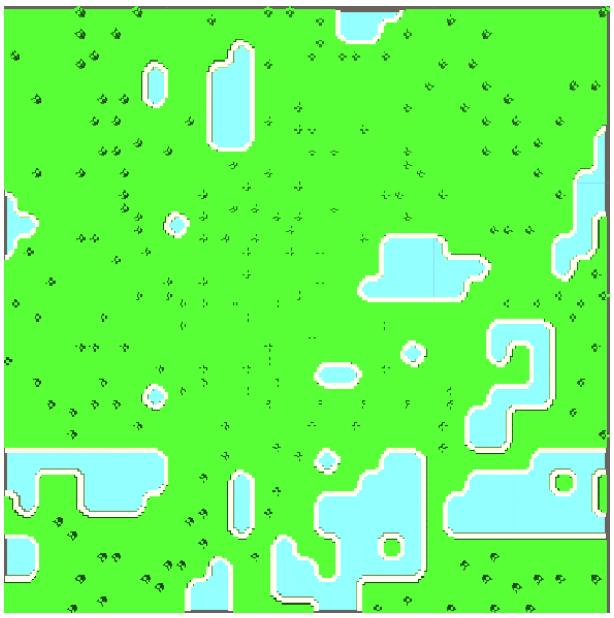


Рис. 4.7. Результат генерації чанку із зерном «121»

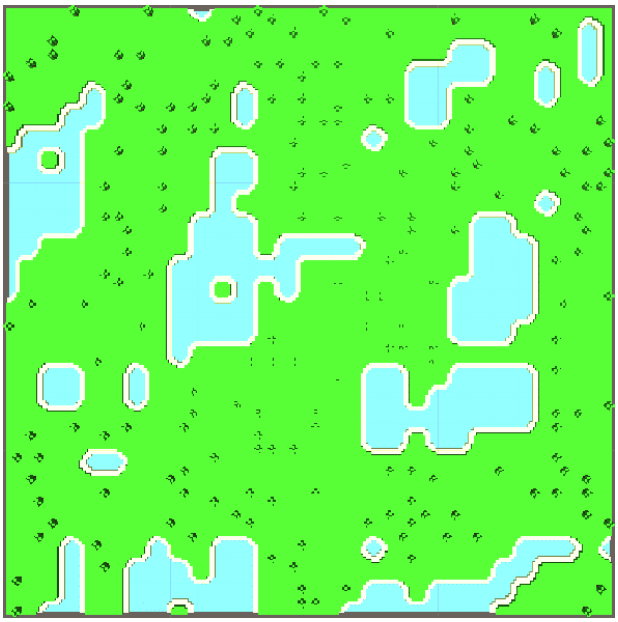


Рис. 4.8. Результат генерації чанку із зерном «512623»

Можемо побачити що мапи згенеровані за допомогою різного зерна генерації істотно відрізняються.

Також для тестування було виконано 10 спроб генерації чанків із однаковим зерном. Внаслідок тестування було виявлено, що при однаковому зерні генерації генератор надає на вихід однакому мапу.

Проте, при однаковому зерні генерації однакові мапи генеруються лише при умові, що було здійснено генерацію чанків у такому ж порядку. Тобто, якщо згенерувати чанки порядку, який не відповідає порядку генерації у попередньому випаду – результат генерації окремих чанаків може відрізнятись. Дана примітка дійсна лише для випадків генерації із урахуванням чанків сусідніх елементів, тобто якщо генерація чанків мапи здійснюється без урахування границь чанків (або здійснюється генерація лише одного чанку), то однакове зерно генерації завжди буде надавати один і той же результат генерації.

Також варто зауважити, що при зміні ваги тайлів у тайлсеті який використовується для генерації мапи, змінюється і результат генерації. Тобто, якщо у двох генераторів зерно генерації співпадає, але у значення ваги тайлів у тайлсеті відрізняється, то дані генератори будуть надавати різний результат генерації.

4.3. Висновки до розділу

У даному розділі було здійснено опис тестування розробленого програмного забезпечення, зроблено порівняльний аналіз розробленого методу, надано оцінку часової ефективності генерації із використання різних методів процедурної генерації та оцінку часової ефективності методу генерації із застосуванням механізму кешування даних тайлсету. У розділі наведено результати тестування певних частин розробленого алгоритмічно-програмного методу, а саме проведено тестування роботи ваги тайлів та роботи методу із різним зерном генерації.

ВИСНОВКИ

В ході даної магістерської дисертації було проаналізовано ряд існуючих алгоритмів та методів процедурної генерації ландшафтів та запропоновано алгоритмічно-програмний метод генерації воксельних ландшафтів із 3D-тайлів, який відрізняється від існуючих застосуванням колапсу хвильової функції та методу послідовної генерації тайлів, що дозволяє підвищити швидкодію програмної обробки даних та зменшити вірогідність некоректної генерації мапи воксельного ландшафту.

В роботі було виконано розробку програмного забезпечення. Тестування розробленого програмного забезпечення показало, що швидкодія програмної обробки даних для генерації воксельного ландшафту зросла в середньому у 2 рази із використання запропонованого у даній роботі алгоритмічно-програмного методу.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Animated low poly characters : [Електронний ресурс]. Режим доступу: https://www.theseus.fi/bitstream/handle/10024/72726/Jolma_Valtteri.pdf
2. Procedural Generation : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Procedural_generation
3. Procedural Generation : [Електронний ресурс]. Режим доступу: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html
4. The Grounded Heightmap Tree : [Електронний ресурс]. Режим доступу: https://upcommons.upc.edu/bitstream/handle/2117/86910/R08-30.pdf
5. Heightmap : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Heightmap
6. Processing Irregular Grid : [Електронний ресурс]. Режим доступу: https://medium.com/@mishaheesakkers/process-ing-generative-irregular-grid-8f0d712dfaa4
7. Tile-based video game : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Tile-based_video_game
8. Marching cubes : [Електронний ресурс]. Режим доступу: https://www.wikiwand.com/uk/Marching_cubes
9. Marching Tetrahedra : [Електронний ресурс]. Режим доступу: https://daac.hpc.mil/gettingStarted/Marching_Tetrahedra.html
10. MagicaVoxel. A free lightweight GPU-based voxel art editor and interactive path tracing renderer : [Електронний ресурс]. Режим доступу: https://ephtracy.github.io/index.html
11. Wave Function Collapse : [Електронний ресурс]. Режим доступу: https://github.com/mxgmn/WaveFunctionCollapse
12. Доступное объяснение алгоритма коллапса волновой функции : [Електронний ресурс]. Режим доступу: https://habr.com/ru/post/461323

13. Choosing the right game engine : [Електронний ресурс]. Режим доступу: https://www.gdquest.com/tutorial/getting-started/learn-to/choosing-a-game-engine/
14. Unity – real-time development platform : [Електронний ресурс]. Режим доступу: https://unity.com/solutions
15. The C# programming language / A. Hejlsberg, M. Torgersen, S. Wiltamuth, P. Golde. — Pearson Education, 2011. — 844 p.
16. Fundamentals of garbage collection platform : [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals
17. Unreal Engine: The most powerful real-time 3D creation tool : [Електронний ресурс]. Режим доступу: https://www.unrealengine.com/en-US/features
18. Blueprints Visual Scripting tool : [Електронний ресурс]. Режим доступу: https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/
19. Welcome back to C++ - Modern C++ : [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp
20. Godot Engine - Free and open source 2D and 3D game engine : [Електронний ресурс]. Режим доступу: https://godotengine.org/features
21. GDScript basics : [Електронний ресурс]. Режим доступу: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
22. Shallow Copy and Deep Copy in C# : [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/shallow-copy-and-deep-copy-in-c-sharp/
23. Ray casting : [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Ray_casting
24. Cache (computing) : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Cache_(computing)

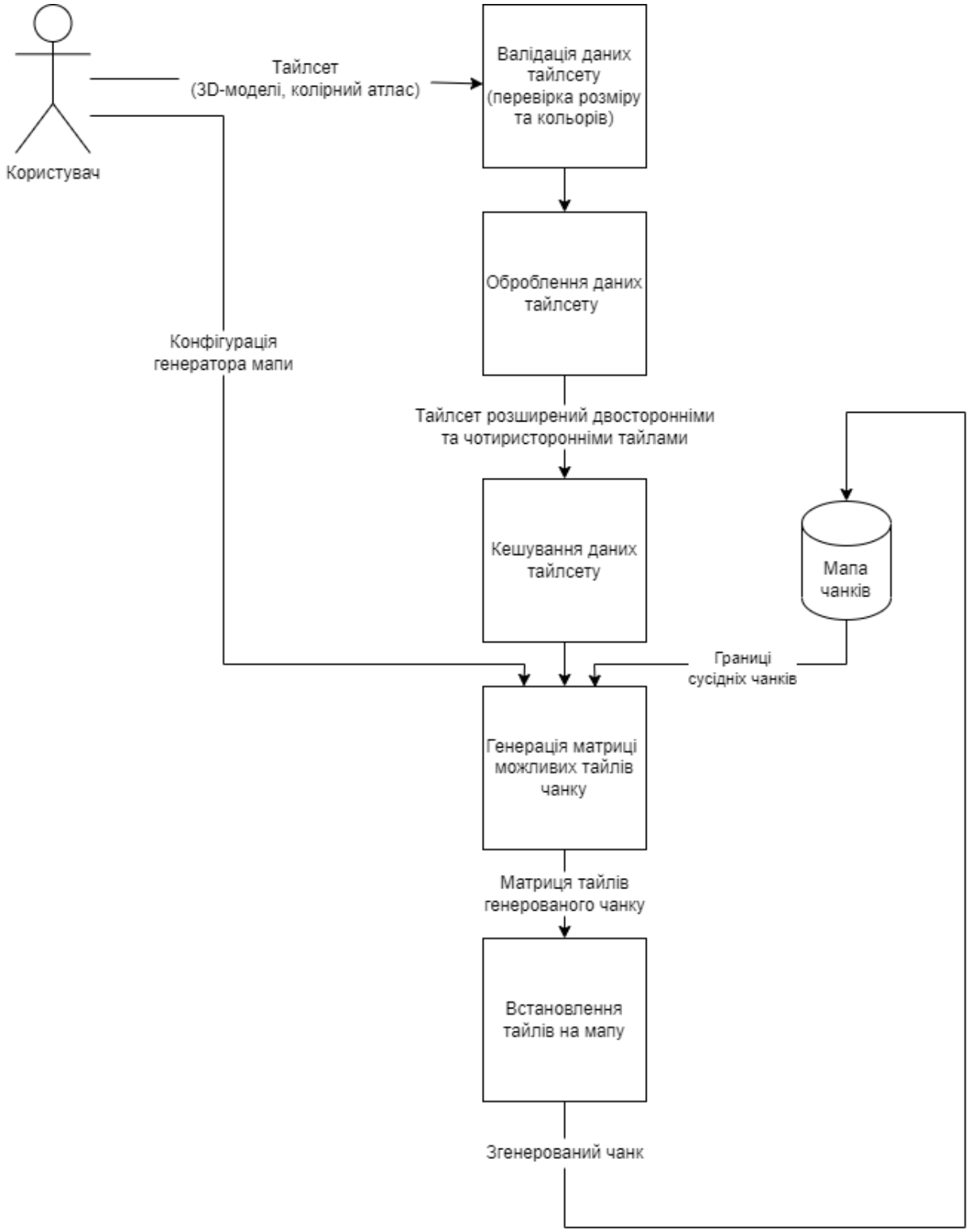
25. Chunking (computing) : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Chunking_(computing)
26. Random seed : [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Random_seed

ДОДАТКИ

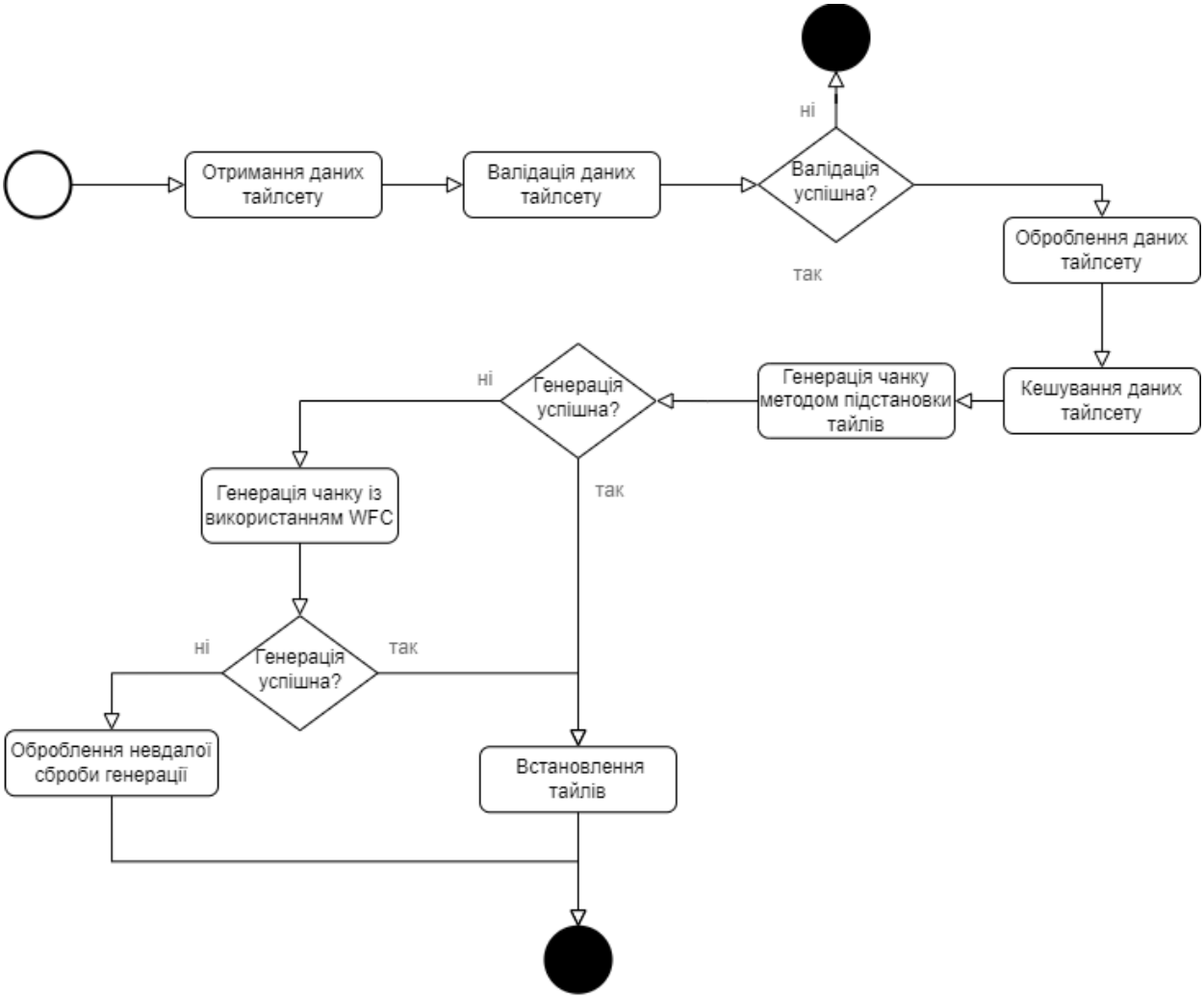
Додаток 1

Копії графічних матеріалів

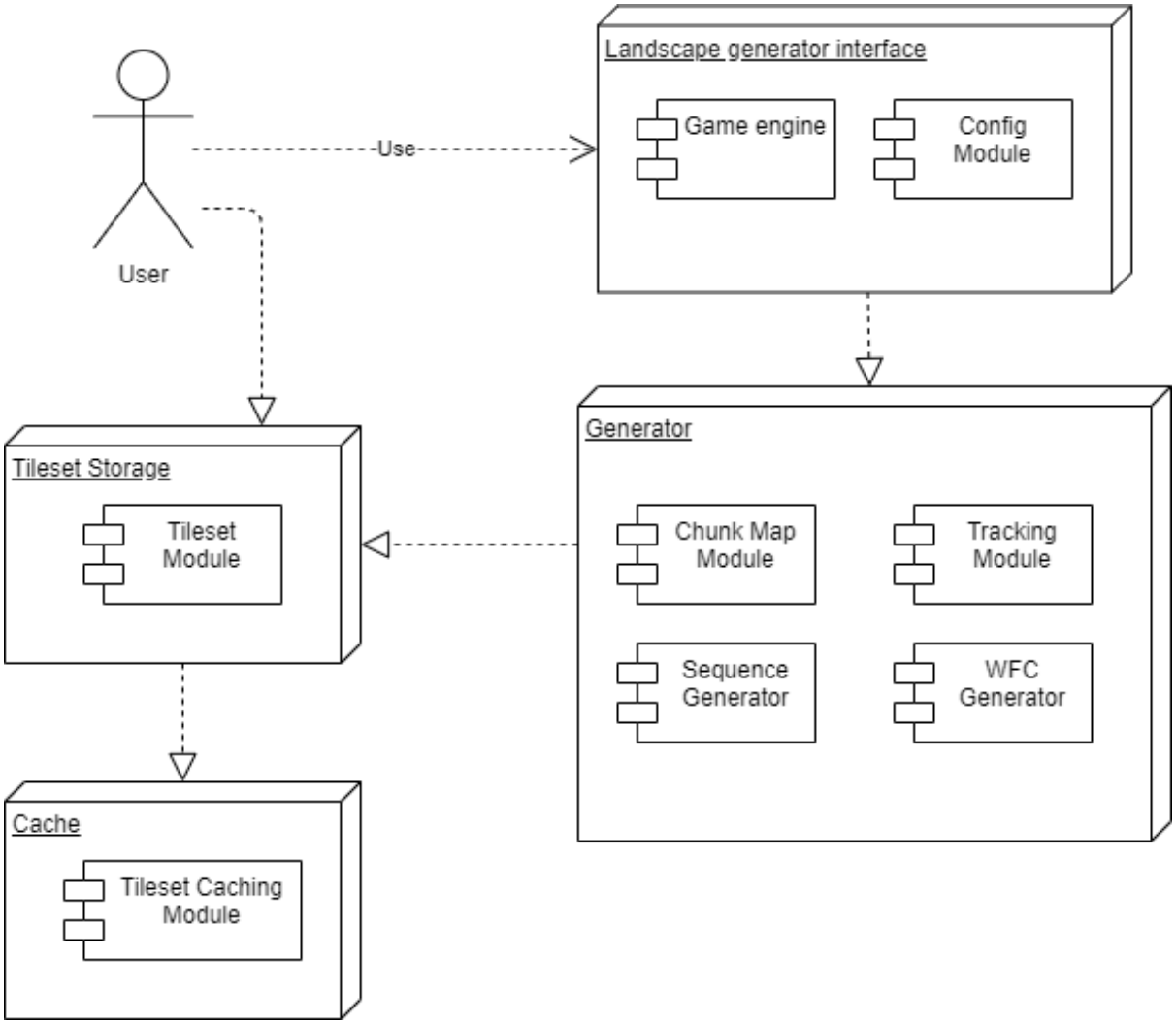
Діаграма потоку даних запропонованого методу генерації воксельного ландшафту



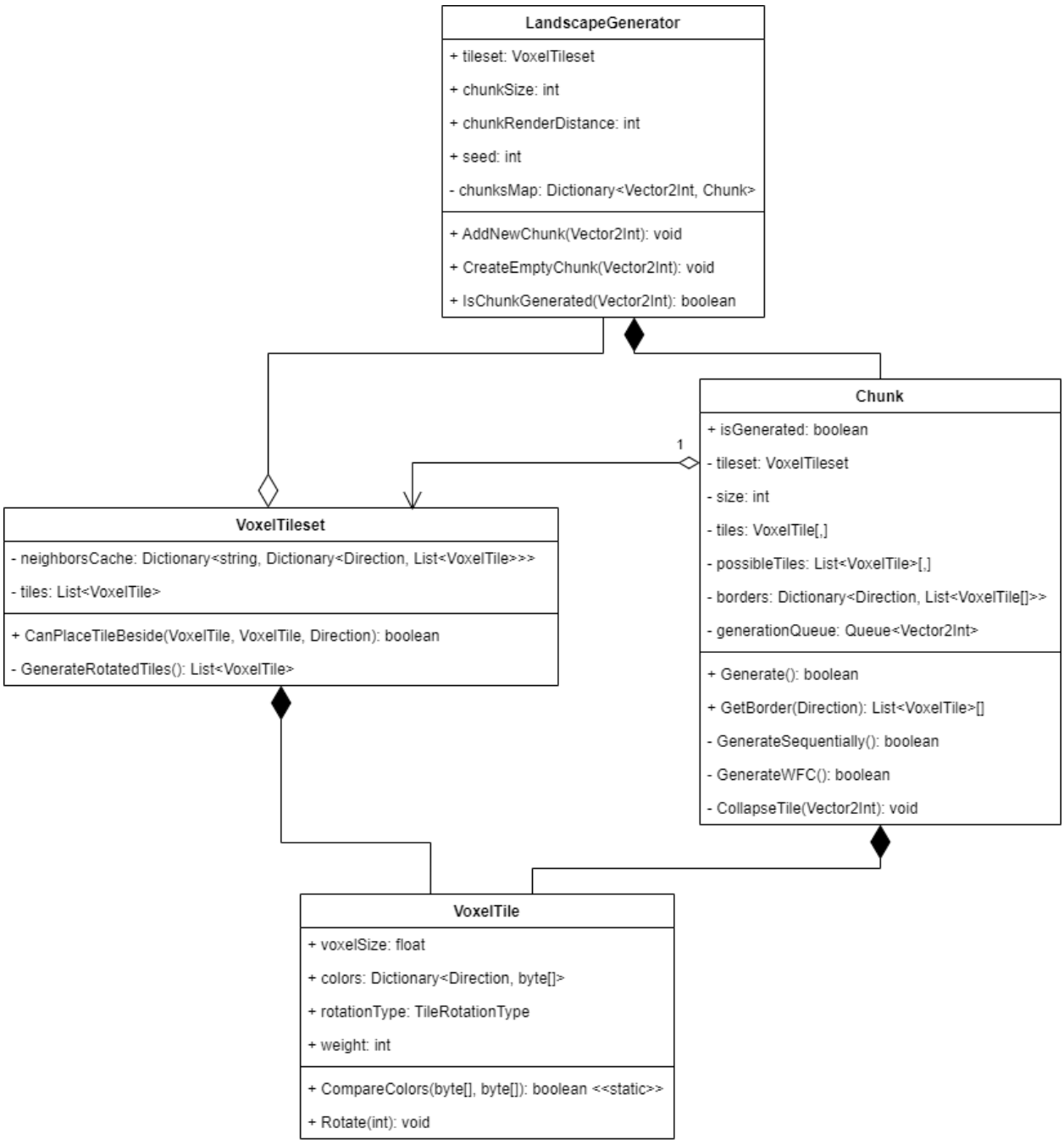
Алгоритм побудови воксельного ландшафту



Діаграма модулів розробленого алгоритмічно-програмного методу



Діаграма класів розробленого алгоритмічно-програмного методу



Додаток 2

Фрагменти тексту програми

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using UnityEngine;
using Debug = UnityEngine.Debug;
using Random = UnityEngine.Random;

public class LandscapeGenerator : MonoBehaviour
{
    [SerializeField] private VoxelTileset tileset;

    [Min(0)]
    [SerializeField]
    private int chunkSize = 8;
    [SerializeField] private int chunkRenderDistance = 2;
    [SerializeField] private bool forceWFC = false;
    [SerializeField] private bool debugMode = false;
    [SerializeField] private int seed = 0;

    private readonly Dictionary<Vector2Int, Chunk> _chunksMap
= new Dictionary<Vector2Int, Chunk>();
    private int _chunksGenerated = 0;
    private int _chunksWithWFC = 0;

    private readonly Stopwatch _stopWatch = new Stopwatch();
    private TimeSpan _totalGenerationTime = new TimeSpan();

    private void Start()
    {
        Random.InitState(seed);

        // Add chunk on start coords
        AddNewChunk(new Vector2Int(0, 0));
    }

    public void OnPlayerMove(Vector3 playerPosition)
    {
        StartCoroutine(nameof(AddChunkOnPlayerMovement),
playerPosition);
    }

    public IEnumerator AddChunkOnPlayerMovement(Vector3
playerPosition)
    {
        var chunkPosition = GetChunkPosition(new
Vector2(playerPosition.x, playerPosition.z));

        // Generate chunks across the player
        // Generate vertical top chunks
```

```
        for (int y = chunkPosition.y; y < chunkPosition.y +
chunkRenderDistance + 1; y++)
        {
            // Generate horizontal right chunks
            for (int x = chunkPosition.x; x < chunkPosition.x
+ chunkRenderDistance + 1; x++)
            {
                AddNewChunk(new Vector2Int(x, y));
                yield return new WaitForEndOfFrame();
            }

            // Generate horizontal left chunks
            for (int x = chunkPosition.x; x > chunkPosition.x
- chunkRenderDistance - 1; x--)
            {
                AddNewChunk(new Vector2Int(x, y));
                yield return new WaitForEndOfFrame();
            }
        }

        // Generate vertical bottom chunks
        for (int y = chunkPosition.y; y > chunkPosition.y -
chunkRenderDistance - 1; y--)
        {
            for (int x = chunkPosition.x; x < chunkPosition.x
+ chunkRenderDistance + 1; x++)
            {
                AddNewChunk(new Vector2Int(x, y));
                yield return new WaitForEndOfFrame();
            }

            for (int x = chunkPosition.x; x > chunkPosition.x
- chunkRenderDistance - 1; x--)
            {
                AddNewChunk(new Vector2Int(x, y));
                yield return new WaitForEndOfFrame();
            }
        }
    }

    private Vector2Int GetChunkPosition(Vector2 position)
    {
        return Vector2Int.FloorToInt(new Vector2(position.x /
(chunkSize * tileset.TileSize), position.y / (chunkSize *
tileset.TileSize)));
    }

    private void CreateEmptyChunk(Vector2Int chunkPosition)
    {
        if (IsChunkExistOnMap(chunkPosition))
        {
```

```
            return;
        }
        var emptyBorders = new Dictionary<Direction2,
List<VoxelTile>[]>();

        var chunk = gameObject.AddComponent<Chunk>();
        chunk.Setup(chunkSize, chunkPosition, tileset,
emptyBorders, forceWFC, debugMode);
        _chunksMap[chunkPosition] = chunk;
    }

    private Dictionary<Direction2, List<VoxelTile>[]>
GetChunkBorders(Vector2Int chunkPosition)
    {
        var borders = new Dictionary<Direction2,
List<VoxelTile>[]>();

        var rightChunkCoords = chunkPosition +
Vector2Int.right;
        var leftChunkCoords = chunkPosition + Vector2Int.left;
        var topChunkCoords = chunkPosition + Vector2Int.up;
        var bottomChunkCoords = chunkPosition +
Vector2Int.down;

        borders[Direction2.Left] =
IsChunkExistOnMap(leftChunkCoords)
            ? _chunksMap[leftChunkCoords].GetRightBorder()
            : null;

        borders[Direction2.Right] =
IsChunkExistOnMap(rightChunkCoords)
            ? _chunksMap[rightChunkCoords].GetLeftBorder()
            : null;

        borders[Direction2.Top] =
IsChunkExistOnMap(topChunkCoords)
            ? _chunksMap[topChunkCoords].GetBottomBorder()
            : null;

        borders[Direction2.Bottom] =
IsChunkExistOnMap(bottomChunkCoords)
            ? _chunksMap[bottomChunkCoords].GetTopBorder()
            : null;

        return borders;
    }

    private void CreateNeighborChunks(Vector2Int
chunkPosition)
    {
        var borders = GetChunkBorders(chunkPosition);
```

```
        var rightChunkCoords = chunkPosition +
Vector2Int.right;
        var leftChunkCoords = chunkPosition + Vector2Int.left;
        var topChunkCoords = chunkPosition + Vector2Int.up;
        var bottomChunkCoords = chunkPosition +
Vector2Int.down;

        void CreateChunk(Vector2Int coords, Direction2
direction)
        {
            if (!IsChunkExistOnMap(coords))
                CreateEmptyChunk(coords);
            _chunksMap[coords].SetBorder(direction,
borders[Direction.GetOppositeDirection(direction)]);
        }

        CreateChunk(topChunkCoords, Direction2.Top);
        CreateChunk(bottomChunkCoords, Direction2.Bottom);
        CreateChunk(leftChunkCoords, Direction2.Left);
        CreateChunk(rightChunkCoords, Direction2.Right);
    }

    private void AddNewChunk(Vector2Int chunkPosition)
    {
        if (IsChunkGenerated(chunkPosition))
        {
            return;
        }

        _stopWatch.Start();

        var chunk = gameObject.AddComponent<Chunk>();
        var borders = GetChunkBorders(chunkPosition);
        chunk.Setup(chunkSize, chunkPosition, tileset,
borders, forceWFC, debugMode);
        chunk.Generate();
        _chunksMap[chunkPosition] = chunk;
        CreateNeighborChunks(chunkPosition);
        _chunksGenerated += 1;
        if (chunk.UsedWFC)
        {
            _chunksWithWFC += 1;
        }

        _stopWatch.Stop();
        var chunkGenerationTime = _stopWatch.Elapsed;
        _totalGenerationTime += chunkGenerationTime;
        _stopWatch.Reset();

        if (debugMode)
```

```
        {
            Debug.Log($"chunk #{_chunksGenerated}, time:
{chunkGenerationTime}");
        }
    }

    private bool IsChunkExistOnMap(Vector2Int chunkPosition)
    {
        return _chunksMap.ContainsKey(chunkPosition);
    }

    private bool IsChunkGenerated(Vector2Int chunkPosition)
    {
        if (IsChunkExistOnMap(chunkPosition))
        {
            return _chunksMap[chunkPosition].isGenerated;
        }

        return false;
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using Random = UnityEngine.Random;
using System.Collections;

public class Chunk : MonoBehaviour
{
    private int _chunkSize;
    private int _chunkSizeWithBorders;
    private VoxelTileset _tileset;
    private Dictionary<Direction2, List<VoxelTile>[]>
_borders;

    private Vector2Int _position;
    private VoxelTile[,] _tiles;
    private List<VoxelTile>[,] _possibleTiles;

    private int _backtracks = 0;
    private readonly int maxBacktracks = 50;
    private bool _debugMode;
    private bool _forceWFC;
    private bool _isReadyForGeneration = false;
    public bool isGenerated = false;

    private bool _usedWFC = false;
    public bool UsedWFC
    {
```

```
        get => _usedWFC;
    }

    private readonly Queue<Vector2Int> _generationQueue = new
Queue<Vector2Int>();

    public void Setup(
        int size,
        Vector2Int position,
        VoxelTileset tileset,
        Dictionary<Direction2, List<VoxelTile>[]> borders,
        bool forceWFC,
        bool debugMode)
    {
        _chunkSize = size;
        _chunkSizeWithBorders = size + 2;
        _tileset = tileset;
        _borders = borders;
        _position = position;
        _tiles = new VoxelTile[_chunkSizeWithBorders,
_chunkSizeWithBorders];
        _forceWFC = forceWFC;
        _debugMode = debugMode;

        _isReadyForGeneration = true;
    }

    private void PlaceTileOnTestLayer(VoxelTile tile,
Vector2Int position)
    {
        if (!tile)
        {
            return;
        }

        var yPosition = 3;

        var tilePosition = new Vector3(position.x, yPosition,
position.y) * tile.TileSize
                        + new Vector3(_position.x, 0,
_position.y) * _chunkSize * tile.TileSize;

        GameObject.Instantiate(tile, tilePosition,
tile.transform.rotation);
    }

    public void Generate()
    {
        if (_debugMode)
        {
            Debug.Log($"Generate chunk {_position}");
```

```
        }

        if (isGenerated)
        {
            throw new Exception("Chunk is already generated");
        }

        if (!_isReadyForGeneration)
        {
            throw new Exception("Chunk is not ready for
generation. Please call Setup method");
        }

        _possibleTiles = new
List<VoxelTile>[_chunkSizeWithBorders, _chunkSizeWithBorders];

        int maxAttempts = 10;
        int attempts = 0;

        if (_forceWFC || !GeneratePossibleTilesBasic())
        {
            _usedWFC = true;
            while (attempts++ < maxAttempts)
            {
                // start generation attempt
                bool success = GeneratePossibleTilesWFC();

                if (success) break;
            }
        }

        PlaceAllTiles();

        isGenerated = true;
    }

    private void PrefillPossibleTiles()
    {
        for (int x = 1; x <= _chunkSize; x++)
        {
            for (int y = 1; y <= _chunkSize; y++)
            {
                _possibleTiles[x, y] = new
List<VoxelTile>(_tileset.TilePrefabs);
            }
        }
    }

    private void FillBorders()
    {
        if (!(_borders[Direction2.Bottom] is null))
```

```
            for (int i = 1; i <= _chunkSize; i++)
                _possibleTiles[i, 0] =
_borders[Direction2.Bottom][i];

        if (!(_borders[Direction2.Top] is null))
            for (int i = 1; i <= _chunkSize; i++)
                _possibleTiles[i, _chunkSize + 1] =
_borders[Direction2.Top][i];

        if (!(_borders[Direction2.Left] is null))
            for (int i = 1; i <= _chunkSize; i++)
                _possibleTiles[0, i] =
_borders[Direction2.Left][i];

        if (!(_borders[Direction2.Right] is null))
            for (int i = 1; i <= _chunkSize; i++)
                _possibleTiles[_chunkSize + 1, i] =
_borders[Direction2.Right][i];
    }

    private void PlaceAllTiles()
    {
        for (int y = 1; y <= _chunkSize; y++)
            for (int x = 1; x <= _chunkSize; x++)
            {
                PlaceTile(x, y);
            }

        if (_debugMode)
        {
            DrawChunkBorders();
        }
    }

    private void PlaceTile(int x, int y)
    {
        List<VoxelTile> availableTiles = _possibleTiles[x, y];

        if (availableTiles.Count == 0) return;

        var selectedTile = GetRandomTile(availableTiles);

        var localTilePosition = new Vector3(x, 0, y) *
_tileset.TileSize;
        var chunkPosition = new Vector3(_position.x, 0,
_position.y) * _chunkSize * _tileset.TileSize;
        var position = chunkPosition + localTilePosition;
        _tiles[x, y] = GameObject.Instantiate(selectedTile,
position, selectedTile.transform.rotation);
    }
```

```
    private bool IsTileGeneratedInPosition(Vector2Int
position)
    {
        return _possibleTiles[position.x, position.y].Count ==
1;
    }

    private bool GeneratePossibleTilesBasic()
    {
        FillBorders();

        for (int y = 1; y < _chunkSize + 1; y++)
            for (int x = 1; x < _chunkSize + 1; x++)
            {
                List<VoxelTile> availableTilesInCell = new
List<VoxelTile>();

                // fill available tiles
                foreach (VoxelTile tilePrefab in
_tileset.TilePrefabs)
                {
                    if ((_possibleTiles[x - 1, y] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x - 1, y][0],
tilePrefab, Direction3.Left)) &&
                        (_possibleTiles[x + 1, y] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x + 1, y][0],
tilePrefab, Direction3.Right)) &&
                        (_possibleTiles[x, y - 1] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x, y - 1][0],
tilePrefab, Direction3.Back)) &&
                        (_possibleTiles[x, y + 1] is null ||
_tileset.CanPlaceTileBeside(_possibleTiles[x, y + 1][0],
tilePrefab, Direction3.Forward)))
                    {
                        availableTilesInCell.Add(tilePrefab);
                    }
                }

                if (availableTilesInCell.Count == 0)
                {
                    return false;
                }

                var selectedTile =
GetRandomTile(availableTilesInCell);
                _possibleTiles[x, y] = new List<VoxelTile> {
selectedTile };
            }

        return true;
    }
```

```
    private bool GeneratePossibleTilesWFC()
    {
        int maxIterations = 100;
        int iterations = 0;

        _generationQueue.Clear();
        _generationQueue.Enqueue(new Vector2Int(1, 1));
        _generationQueue.Enqueue(new Vector2Int(1,
_chunkSize));
        _generationQueue.Enqueue(new Vector2Int(_chunkSize,
1));
        _generationQueue.Enqueue(new Vector2Int(_chunkSize,
_chunkSize));

        PrefillPossibleTiles();
        FillBorders();

        while (iterations++ < maxIterations && _backtracks <
maxBacktracks)
        {
            int maxInnerIterations = 10000;
            int innerIterations = 0;

            while (_generationQueue.Count > 0 &&
innerIterations++ < maxInnerIterations)
            {
                ProcessTileInGenerationQueue();
            }

            if (innerIterations == maxInnerIterations) break;

            var maxCountTilePosition =
GetPositionWithMaxPossibleTiles();
            var maxCountTile =
_possibleTiles[maxCountTilePosition.x,
maxCountTilePosition.y];

            if (maxCountTile.Count == 1)
            {
                if (_debugMode)
                {
                    Debug.Log($"Generated for {iterations}
iterations, with {_backtracks} backtracks");
                }
                return true;
            }

            var minCountTilePosition =
GetPositionWithMinPossibleTiles();
            CollapseTile(minCountTilePosition);
```

```
        }

        if (_debugMode)
        {
            Debug.Log($"Failed, run out of iterations with
{_backtracks} backtracks");
        }
        return false;
    }

    private void ProcessTileInGenerationQueue()
    {
        Vector2Int position = _generationQueue.Dequeue();
        if (!IsPositionInsideChunk(position))
        {
            throw new Exception($"Position is not inside the
chunk: {position}");
        }

        if (IsTileGeneratedInPosition(position))
        {
            return;
        }

        int numberOfRemovedTiles =
RemoveImpossibleTiles(position);

        if (_possibleTiles[position.x, position.y].Count == 0)
        {
            // no possible tiles, try to recalculate this tile
and neighbors
            _possibleTiles[position.x,
position.y].AddRange(_tileset.TilePrefabs);

            void ResetTilesInNeighbor(Direction2 direction)
            {
                var neighborPosition = position +
Direction.ConvertToVector(direction);
                if (IsPositionInsideChunk(neighborPosition))
                    _possibleTiles[neighborPosition.x,
neighborPosition.y] = new
List<VoxelTile>(_tileset.TilePrefabs);
            }

            ResetTilesInNeighbor(Direction2.Right);
            ResetTilesInNeighbor(Direction2.Left);
            ResetTilesInNeighbor(Direction2.Top);
            ResetTilesInNeighbor(Direction2.Bottom);

            _generationQueue.Enqueue(position);
            AddNeighborsToGenerationQueue(position);
```

```
            _backtracks += 1;
        }
        else if (numberOfRemovedTiles > 0)
        {
            AddNeighborsToGenerationQueue(position);
        }
    }

    /// <summary>
    /// Collapse possible tiles in position to one tile
    /// </summary>
    /// <param name="position">Position in chunk to
collapse</param>
    private void CollapseTile(Vector2Int position)
    {
        VoxelTile tileToCollapse =
GetRandomTile(_possibleTiles[position.x, position.y]);
        _possibleTiles[position.x, position.y] = new
List<VoxelTile> { tileToCollapse };
        AddNeighborsToGenerationQueue(position);
    }

    private Vector2Int GetPositionWithMaxPossibleTiles()
    {
        var maxCountTile = _possibleTiles[1, 1];
        var maxCountTilePosition = new Vector2Int(1, 1);

        for (int x = 1; x <= _chunkSize; x++)
            for (int y = 1; y <= _chunkSize; y++)
            {
                if (_possibleTiles[x, y].Count >
maxCountTile.Count)
                {
                    maxCountTilePosition = new Vector2Int(x,
y);
                    maxCountTile =
_possibleTiles[maxCountTilePosition.x,
maxCountTilePosition.y];
                }
            }

        return maxCountTilePosition;
    }

    /// <summary>
    /// Returns the position of the _possibleTiles element
with the minimum number of possible tiles.
    /// <para>Method does not count cells where only 1 tile is
possible to set</para>
    /// </summary>
```

```
    private Vector2Int GetPositionWithMinPossibleTiles()
    {
        var minTile = _possibleTiles[1, 1];
        var minPosition = new Vector2Int(1, 1);

        for (int x = 1; x <= _chunkSize; x++)
            for (int y = 1; y <= _chunkSize; y++)
            {
                var currentTile = _possibleTiles[x, y];

                if (currentTile.Count <= 1)
                {
                    continue;
                }

                if (currentTile.Count < minTile.Count ||
minTile.Count <= 1)
                {
                    minTile = _possibleTiles[x, y];
                    minPosition = new Vector2Int(x, y);
                }
            }

        return minPosition;
    }

    /// <summary>
    /// Remove impossible tiles from _possibleTiles element
    /// </summary>
    /// <param name="tilePosition">Position to remove
impossible tiles</param>
    private int RemoveImpossibleTiles(Vector2Int tilePosition)
    {
        var updatedTiles = new List<VoxelTile>();
        var tilesHere = _possibleTiles[tilePosition.x,
tilePosition.y];
        int numberOfTilesBeforeRemoval = tilesHere.Count;

        for (int i = 0; i < numberOfTilesBeforeRemoval; i++)
        {
            var tile = tilesHere[i];
            if (IsTilePossible(tile, tilePosition))
            {
                updatedTiles.Add(tile);
            }
        }

        _possibleTiles[tilePosition.x, tilePosition.y] =
updatedTiles;
        return numberOfTilesBeforeRemoval -
updatedTiles.Count;
```

```
    }

    /// <summary>
    /// Checks if the tile can be installed in the provided
position
    /// </summary>
    /// <param name="tile">Tile to check</param>
    /// <param name="position">Position for checking the
possibility of installing a provided tile</param>
    private bool IsTilePossible(VoxelTile tile, Vector2Int
position)
    {
        var right = new Vector2Int(position.x + 1,
position.y);
        var left = new Vector2Int(position.x - 1, position.y);
        var up = new Vector2Int(position.x, position.y + 1);
        var down = new Vector2Int(position.x, position.y - 1);

        bool CanAppendAtLeastOneTile(Vector2Int p, Direction3
direction)
        {
            var tilesHere = _possibleTiles[p.x, p.y];
            for (int i = 0; i < tilesHere.Count; i++)
            {
                if (_tileset.CanPlaceTileBeside(tile,
tilesHere[i], direction))
                {
                    return true;
                }
            }
            return false;
        }

        if (IsPositionInsideChunk(left) ||
(IsPositionOnBorder(left) && !(_possibleTiles[left.x, left.y]
is null)))
        {
            bool isTilePossible =
CanAppendAtLeastOneTile(left, Direction3.Right);
            if (!isTilePossible) return false;
        }

        if (IsPositionInsideChunk(right) ||
(IsPositionOnBorder(right) && !(_possibleTiles[right.x,
right.y] is null)))
        {
            bool isTilePossible =
CanAppendAtLeastOneTile(right, Direction3.Left);
            if (!isTilePossible) return false;
        }
```

```
        if (IsPositionInsideChunk(down) ||
(IsPositionOnBorder(down) && !(_possibleTiles[down.x, down.y]
is null)))
        {
            bool isTilePossible =
CanAppendAtLeastOneTile(down, Direction3.Forward);
            if (!isTilePossible) return false;
        }

        if (IsPositionInsideChunk(up) ||
(IsPositionOnBorder(up) && !(_possibleTiles[up.x, up.y] is
null)))
        {
            bool isTilePossible = CanAppendAtLeastOneTile(up,
Direction3.Back);
            if (!isTilePossible) return false;
        }
        return true;
    }

    private void AddNeighborsToGenerationQueue(Vector2Int
position)
    {
        void EnqueueNeighbor(Direction2 direction)
        {
            var neighborPosition = position +
Direction.ConvertToVector(direction);
            if (IsPositionInsideChunk(neighborPosition) &&
_possibleTiles[neighborPosition.x, neighborPosition.y].Count >
1)
                _generationQueue.Enqueue(neighborPosition);
        }

        EnqueueNeighbor(Direction2.Bottom);
        EnqueueNeighbor(Direction2.Left);
        EnqueueNeighbor(Direction2.Right);
        EnqueueNeighbor(Direction2.Top);

        _generationQueue.Distinct();
    }

    private static VoxelTile GetRandomTile(List<VoxelTile>
availableTiles)
    {
        var chances = new List<float>();
        foreach (var tile in availableTiles)
        {
            chances.Add(tile.Weight);
        }

        float value = Random.Range(0, chances.Sum());
```

```
        float sum = 0;

        for (int i = 0; i < chances.Count; i++)
        {
            sum += chances[i];
            if (value < sum)
            {
                return availableTiles[i];
            }
        }

        return availableTiles[0];
    }

    public void SetBorder(Direction2 borderDirection,
List<VoxelTile>[] border)
    {
        _borders[borderDirection] = border;
    }

    public List<VoxelTile>[] GetBorder(Direction2 direction)
    {
        if (direction == Direction2.Left)
            return GetLeftBorder();

        if (direction == Direction2.Right)
            return GetRightBorder();

        if (direction == Direction2.Top)
            return GetTopBorder();

        if (direction == Direction2.Bottom)
            return GetBottomBorder();

        throw new ArgumentException("Wrong direction");
    }

    public List<VoxelTile>[] GetTopBorder()
    {
        if (_possibleTiles is null)
        {
            return new List<VoxelTile>[_chunkSizeWithBorders];
        }

        var top = new List<VoxelTile>[_chunkSizeWithBorders];
        for (var x = 1; x <= _chunkSize; x++)
        {
            top[x] = _possibleTiles[x, _chunkSize];
        }

        return top;
```

```
    }

    public List<VoxelTile>[] GetRightBorder()
    {
        if (_possibleTiles is null)
        {
            return new List<VoxelTile>[_chunkSizeWithBorders];
        }

        var right = new
List<VoxelTile>[_chunkSizeWithBorders];
        for (var y = 1; y <= _chunkSize; y++)
        {
            right[y] = _possibleTiles[_chunkSize, y];
        }

        return right;
    }

    public List<VoxelTile>[] GetBottomBorder()
    {
        if (_possibleTiles is null)
        {
            return new List<VoxelTile>[_chunkSizeWithBorders];
        }

        var bottom = new
List<VoxelTile>[_chunkSizeWithBorders];
        for (var x = 1; x <= _chunkSize; x++)
        {
            bottom[x] = _possibleTiles[x, 1];
        }

        return bottom;
    }

    public List<VoxelTile>[] GetLeftBorder()
    {
        if (_possibleTiles is null)
        {
            return new List<VoxelTile>[_chunkSizeWithBorders];
        }

        var left = new List<VoxelTile>[_chunkSizeWithBorders];
        for (var y = 1; y <= _chunkSize; y++)
        {
            left[y] = _possibleTiles[1, y];
        }

        return left;
    }
```

```
    public void Remove()
    {
        foreach (var tile in _tiles)
        {
            if (tile != null)
GameObject.Destroy(tile.gameObject);
        }
    }

    private bool IsPositionInsideChunk(Vector2Int position)
    {
        return
            position.x >= 1
            && position.y >= 1
            && position.x <= _chunkSize
            && position.y <= _chunkSize;
    }

    private bool IsPositionOnBorder(Vector2Int position)
    {
        return
            position.x == 0
            || position.y == 0
            || position.x == _chunkSize + 1
            || position.y == _chunkSize + 1;
    }

    private void DrawChunkBorders()
    {
        var myLine = new GameObject();
        myLine.transform.position = new Vector3(_position.x,
0, _position.y);
        myLine.AddComponent<LineRenderer>();
        var lineRenderer =
myLine.GetComponent<LineRenderer>();
        lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));

        var lineColor = Color.red;
        lineRenderer.startColor = lineColor;
        lineRenderer.endColor = lineColor;
        lineRenderer.startWidth = 0.1f;
        lineRenderer.endWidth = 0.1f;

        var tileSize = _tileset.TileSize;
        var chunkOffsetVector = new Vector3(_position.x *
_chunkSize * tileSize, 0, _position.y * _chunkSize *
tileSize);

        var yPosition = 2 * tileSize;
```

```
        var leftBottom = new Vector3(0.5f, yPosition, 0.5f) *
tileSize + chunkOffsetVector;
        var leftTop = new Vector3(0.5f, yPosition, _chunkSize
+ 0.5f) * tileSize + chunkOffsetVector;
        var rightTop = new Vector3(_chunkSize + 0.5f,
yPosition, _chunkSize + 0.5f) * tileSize + chunkOffsetVector;
        var rightBottom = new Vector3(_chunkSize + 0.5f,
yPosition, 0.5f) * tileSize + chunkOffsetVector;

        var positions = new Vector3[]
        {
            leftBottom,
            leftTop,
            rightTop,
            rightBottom,
            leftBottom,
        };

        lineRenderer.positionCount = positions.Length;
        lineRenderer.SetPositions(positions);
    }
}

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

public class VoxelTileset : MonoBehaviour
{
    [SerializeField] private List<VoxelTile> tilePrefabs = new
List<VoxelTile>();

    public List<VoxelTile> TilePrefabs
    {
        get => tilePrefabs;
    }

    // Dictionary<Direction3, List<VoxelTile>> - possible
neighbors of a tile for all directions
    private class NeighborsCache : Dictionary<string,
Dictionary<Direction3, List<VoxelTile>>> { }

    private readonly NeighborsCache neighborsCache = new
NeighborsCache();

    public float TileSize => tilePrefabs[0].TileSize;

    enum RotationDegree
    {
        _90,
        _180,
```

```
        _270,
    }

    private void GenerateRotatedTiles()
    {
        var prefabsCount = tilePrefabs.Count;
        for (var i = 0; i < prefabsCount; i++)
        {
            void CreateRotatedClone(RotationDegree rotation)
            {
                var clone = Instantiate(tilePrefabs[i],
tilePrefabs[i].transform.position + Vector3.right,
Quaternion.identity);

                if (rotation == RotationDegree._90)
                {
                    clone.Rotate90();
                    clone.transform.position += Vector3.right;
                    clone.Name = $"{clone.Name} 90";
                }

                if (rotation == RotationDegree._180)
                {
                    clone.Rotate90();
                    clone.Rotate90();
                    clone.transform.position += 2 *
Vector3.right;
                    clone.Name = $"{clone.Name} 180";
                }

                if (rotation == RotationDegree._270)
                {
                    clone.Rotate90();
                    clone.Rotate90();
                    clone.Rotate90();
                    clone.transform.position += 3 *
Vector3.right;
                    clone.Name = $"{clone.Name} 270";
                }

                tilePrefabs.Add(clone);
            }

            switch (tilePrefabs[i].RotationType)
            {
                case VoxelTile.TileRotationType.OneSided:
                    break;

                case VoxelTile.TileRotationType.TwoSided:
                    tilePrefabs[i].Weight /= 2;
                    CreateRotatedClone(RotationDegree._90);
```

```
                    break;

                case VoxelTile.TileRotationType.FourSided:
                    tilePrefabs[i].Weight /= 4;
                    CreateRotatedClone(RotationDegree._90);
                    CreateRotatedClone(RotationDegree._180);
                    CreateRotatedClone(RotationDegree._270);
                    break;
                default:
                    throw new ArgumentOutOfRangeException();
            }
        }
    }

    private void FillNeighborsCache()
    {
        List<VoxelTile> GetPossibleNeighbors(VoxelTile tile,
Direction3 direction)
        {
            var neighbors = new List<VoxelTile>();
            foreach (var tileToAppend in tilePrefabs)
            {
                if (CanAppendTile(tile, tileToAppend,
direction))
                {
                    neighbors.Add(tileToAppend);
                }
            }

            return neighbors;
        }

        foreach (var tile in tilePrefabs)
        {
            neighborsCache[tile.Name] = new
Dictionary<Direction3, List<VoxelTile>>();

            var directions = new Direction3[] {
Direction3.Right, Direction3.Left, Direction3.Forward,
Direction3.Back };
            foreach (var direction in directions)
            {
                neighborsCache[tile.Name][direction] =
GetPossibleNeighbors(tile, direction);
            }
        }
    }

    private bool CanAppendTile(VoxelTile tile1, VoxelTile
tile2, Direction3 direction)
    {
```

```
        return
VoxelTile.CompareColors(tile1.GetColors(direction),
tile2.GetColors(Direction.GetOppositeDirection(direction)));
    }

    private void Start()
    {
        foreach (VoxelTile tilePrefab in tilePrefabs)
        {
            tilePrefab.CalculateSidesColors();
        }

        GenerateRotatedTiles();
        FillNeighborsCache();
    }

    /// <summary>
    /// Check if can append tiles to each other
    /// Uses cached value from the "neighborsDictionary"
    /// </summary>
    public bool CanPlaceTileBeside(VoxelTile tile1, VoxelTile
tile2, Direction3 direction)
    {
        if (neighborsCache is null)
            return true;

        foreach (var tile in
neighborsCache[tile1.Name][direction])
        {
            if (tile.Name == tile2.Name)
            {
                return true;
            }
        }

        return false;
    }
}

using System;
using UnityEngine;

public class VoxelTile : MonoBehaviour
{
    public float VoxelSize = 0.1f;
    public int TileSideVoxels = 8;
    public float TileSize => VoxelSize * TileSideVoxels;
    public string Name = Guid.NewGuid().ToString("n")[..8];

    [Range(1, 100)]
    public int Weight = 50;
```

```
    public enum TileRotationType
    {
        OneSided,
        TwoSided,
        FourSided
    }
    public TileRotationType RotationType =
TileRotationType.OneSided;

    // Store it in separate arrays to make an ability to
easily shallow copy the tile
    [HideInInspector] public byte[] ColorsRight;
    [HideInInspector] public byte[] ColorsForward;
    [HideInInspector] public byte[] ColorsLeft;
    [HideInInspector] public byte[] ColorsBack;

    public void CalculateSidesColors()
    {
        ColorsRight = new byte[TileSideVoxels *
TileSideVoxels];
        ColorsForward = new byte[TileSideVoxels *
TileSideVoxels];
        ColorsLeft = new byte[TileSideVoxels *
TileSideVoxels];
        ColorsBack = new byte[TileSideVoxels *
TileSideVoxels];

        for (int y = 0; y < TileSideVoxels; y++)
        {
            for (int x = 0; x < TileSideVoxels; x++)
            {
                ColorsRight[y * TileSideVoxels + x] =
GetVoxelColor(x, y, Direction3.Right);
                ColorsForward[y * TileSideVoxels + x] =
GetVoxelColor(x, y, Direction3.Forward);
                ColorsLeft[y * TileSideVoxels + x] =
GetVoxelColor(x, y, Direction3.Left);
                ColorsBack[y * TileSideVoxels + x] =
GetVoxelColor(x, y, Direction3.Back);
            }
        }
    }

    public void Rotate90()
    {
        transform.Rotate(0, 90, 0);

        byte[] colorsRightNew = new byte[TileSideVoxels *
TileSideVoxels];
```

```
        byte[] colorsForwardNew = new byte[TileSideVoxels *
TileSideVoxels];
        byte[] colorsLeftNew = new byte[TileSideVoxels *
TileSideVoxels];
        byte[] colorsBackNew = new byte[TileSideVoxels *
TileSideVoxels];

        for (int y = 0; y < TileSideVoxels; y++)
        {
            for (int x = 0; x < TileSideVoxels; x++)
            {
                colorsRightNew[y * TileSideVoxels + x] =
ColorsForward[y * TileSideVoxels + TileSideVoxels - x - 1];
                colorsForwardNew[y * TileSideVoxels + x] =
ColorsLeft[y * TileSideVoxels + x];
                colorsLeftNew[y * TileSideVoxels + x] =
ColorsBack[y * TileSideVoxels + TileSideVoxels - x - 1];
                colorsBackNew[y * TileSideVoxels + x] =
ColorsRight[y * TileSideVoxels + x];
            }
        }

        ColorsRight = colorsRightNew;
        ColorsForward = colorsForwardNew;
        ColorsLeft = colorsLeftNew;
        ColorsBack = colorsBackNew;
    }

    private byte GetVoxelColor(int x, int y, Direction3
direction)
    {
        var meshCollider =
GetComponentInChildren<MeshCollider>();

        float vox = VoxelSize;
        float offset = VoxelSize / 2;

        bool isRightOrForward = direction == Direction3.Right
|| direction == Direction3.Forward;

        float xPosition = isRightOrForward
            ? x * vox
            : (TileSideVoxels - x - 1) * vox;

        Vector3 colliderBounds = isRightOrForward ?
meshCollider.bounds.min : meshCollider.bounds.max;

        Vector3 rayStart = colliderBounds;
        if (direction == Direction3.Right)
        {
```

```
            rayStart += new Vector3(-offset, 0, offset +
xPosition);
        }
        else if (direction == Direction3.Forward)
        {
            rayStart += new Vector3(offset + xPosition, 0, -
offset);
        }
        else if (direction == Direction3.Left)
        {
            rayStart += new Vector3(offset, 0, -offset -
xPosition);
        }
        else if (direction == Direction3.Back)
        {
            rayStart += new Vector3(-offset - xPosition, 0,
offset);
        }
        else
        {
            throw new ArgumentException("Wrong direction
value, should be Direction3.Left/Right/Back/Forward",
                nameof(direction));
        }

        rayStart.y = meshCollider.bounds.min.y + offset + y *
vox;

        Debug.DrawRay(rayStart,
Direction.ConvertToVector3(direction) * .1f, Color.green,
20000);

        if (Physics.Raycast(new Ray(rayStart,
Direction.ConvertToVector3(direction)), out RaycastHit hit,
vox))
        {
            byte colorIndex = (byte)(hit.textureCoord.x *
256);

            if (colorIndex == 0) Debug.LogWarning("Found color
0 in mesh palette, this can cause conflicts");

            return colorIndex;
        }

        return 0;
    }

    public static bool CompareColors(byte[] colors1, byte[]
colors2)
    {
```

```
        if (colors1.Length != colors2.Length)
            return false;

        for (int i = 0; i < colors1.Length; i++)
        {
            if (colors1[i] != colors2[i])
            {
                return false;
            }
        }
        return true;
    }

    public byte[] GetColors(Direction3 direction)
    {
        if (direction == Direction3.Right)
        {
            return ColorsRight;
        }

        if (direction == Direction3.Left)
        {
            return ColorsLeft;
        }

        if (direction == Direction3.Forward)
        {
            return ColorsForward;
        }

        if (direction == Direction3.Back)
        {
            return ColorsBack;
        }

        throw new ArgumentException($"Direction {direction} is
not supported");
    }
}
```

Додаток 3

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

Алгоритмічно-програмний метод процедурної генерації воксельного ландшафту із 3D-тайлів

Доповідач: Брусенцов Ю. О.

Науковий керівник: д.т.н., доцент Сулема Є. С.

Київ – 2022

ПОСТАНОВКА ЗАДАЧІ

Об'єкт дослідження: процес процедурної генерації воксельного ландшафту з 3D-тайлів.

Предмет дослідження: методи, алгоритми та програмне забезпечення для генерації воксельного ландшафту.

ПОСТАНОВКА ЗАДАЧІ

Завдання дослідження: розробити алгоритмічно-програмний метод процедурної генерації ландшафту.
Мета дослідження: оптимізувати процеси процедурної генерації ландшафту.

Список завдань

1. Проаналізувати існуючі рішення та методи процесуального створення ландшафту.
2. Розробити власний метод процедурної генерації воксельного ландшафту.
3. Розробити алгоритмічне та програмне забезпечення реалізації розробленого методу.
4. Провести аналіз розорobленого методу.
5. Зробити висновок за результатами дослідження.

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

- Створення та візуалізація віртуальних пейзажів є актуальним завданням у таких сферах, як розроблення графічних матеріалів для освітніх ресурсів, відеоігор, фільмів та анімації, а також середовищ для моделювання природних процесів.
- Процедурна генерація – це потужний інструмент для генерації контенту, проте процедурний вміст інколи важко контролювати.

ТЕРМІНОЛОГІЯ (1)

Процедурна генерація – це
метод алгоритмічного створення даних за допомогою комбінацій алгоритмів, поєднаних із випадковістю.
У комп'ютерній графіці він зазвичай використовується для створення текстур та 3D-моделей.

Воксель (від англ. *Volume* та англ. *pixel*) – елемент простору, позначає значення певної величини в клітинках рівномірної просторової ґратки. Аналогічний пікселю, у двовимірних зображеннях.

ТЕРМІНОЛОГІЯ (2)

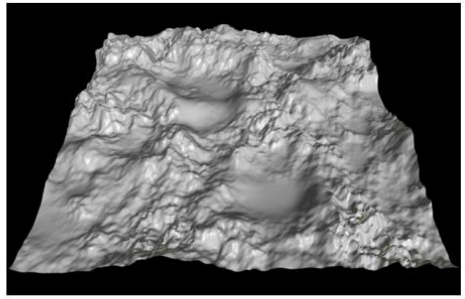
Тайл (від англ. tile – плитка) – невеликі зображення однакових розмірів, які служать фрагментами великої картини.

Тайлсет – набір тайлів які використовуються для створення контенту.

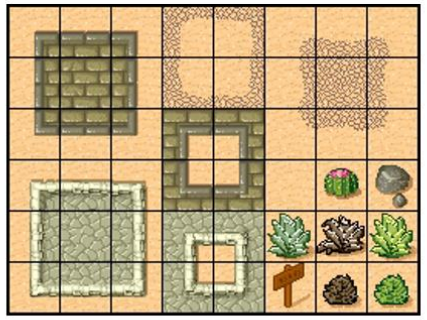
Чанк (від англ. chunk – шматок) – шматки фіксованого розміру генерованої мапи. Використання чанків дозволяє розбити генерацію великої мапи на багато малих шматків

НАЙБІЛЬШ ВІДОМІ РІШЕННЯ ЦІЄЇ НАУКОВОЇ ПРОБЛЕМИ

- Генерація карт висот (HeightMaps)
  - Шум Перліна
  - Діаграма Вороного
  - Diamond-square algorithm



- Генерація плиткової структури (TileMaps)



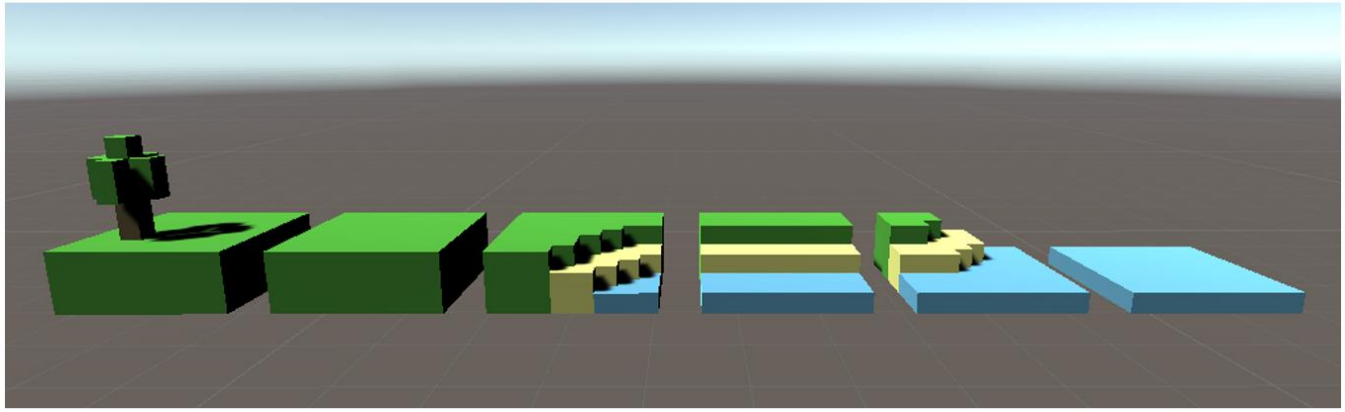
ГІПОТЕЗА

Оптимізація та вибір алгоритмів процедурної генерації дасть можливість досягти більшої швидкості та точності генерації воксельного ландшафту.

ЗАПРОПОНОВАНИЙ МЕТОД (1)

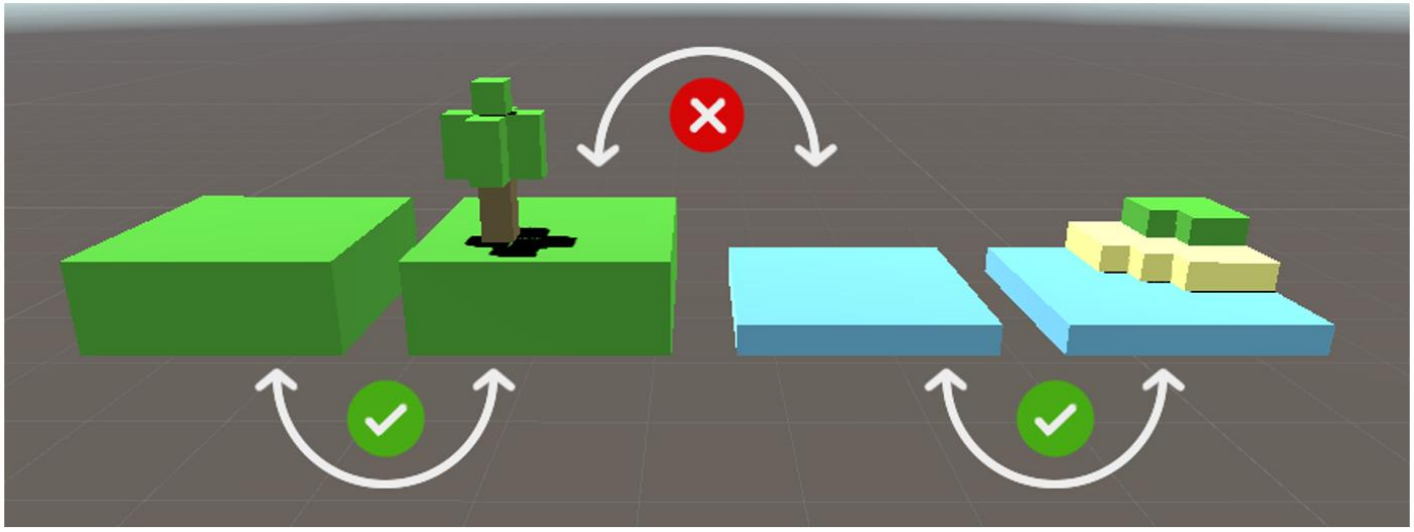
Запропоновано використати метод процедурної генерації воксельного ландшафту із 3D-тайлів.

Приклад воксельного тайлсету:



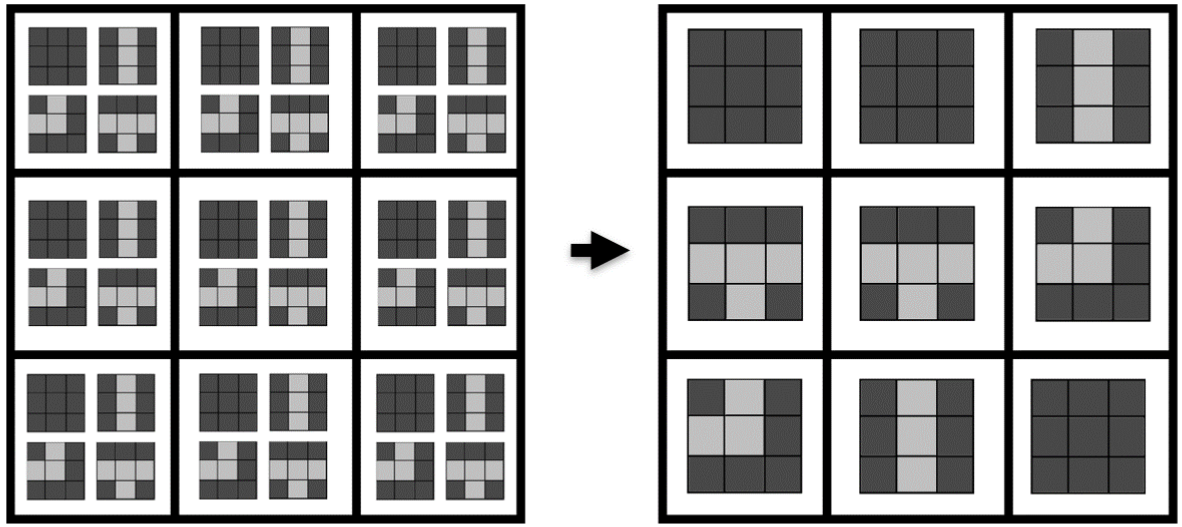
ЗАПРОПОНОВАНИЙ МЕТОД (2)

Основна ідея методу полягає в тому, щоб розмістити тайли так, щоб зовнішні сторони вокселів тайлів, які стоять поруч, мали однакові кольори.



ЗАПРОПОНОВАНИЙ МЕТОД (3)

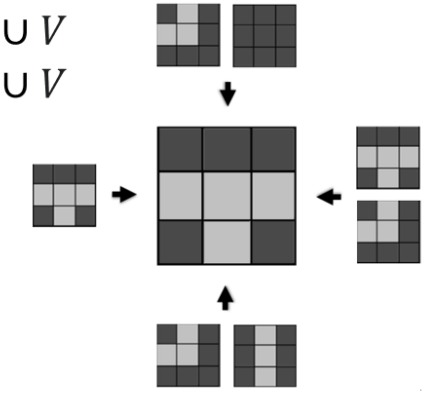
Запропонований метод генерації передбачає представлення мапи у вигляді матриці тайлів.



ЗАПРОПОНОВАНИЙ МЕТОД (4)

Для кожного елементу тайлсету створюється множина можливих для підстановки сусідніх елементів:

$$\forall a_{ij} \in \mathrm{A}_{n\times n} \cap b_{ij} \in \mathrm{B}_{n\times n} \begin{cases} a_{1i} = b_{ni} \Rightarrow B \cup V \\ a_{ni} = b_{1i} \Rightarrow B \cup V \\ a_{i1} = b_{in} \Rightarrow B \cup V \\ a_{in} = b_{i1} \Rightarrow B \cup V \end{cases}$$



A – матриця вокселів обраного елементу.
B – матриця вокселів, із яким йде порівняння.
V – масив тайлів, що підходять для підстановки.
n – розмір тайлу.
i – індекс поточного вокселю (i = 1 .. n)

ПОСЛІДОВНИЙ МЕТОД ГЕНЕРАЦІЇ (1)

Одним із запропонованих методів обходу матриці елементів є метод послідовний метод. У цьому методі координати для встановлення тайлів йдуть послідовно, тобто у циклі.

Обхід матриці можливих тайлів можна записати у наступному вигляді:

$$M[x, y]_i := M[x, y]_i \cap V,$$

M – це матриця генерованої мапи,

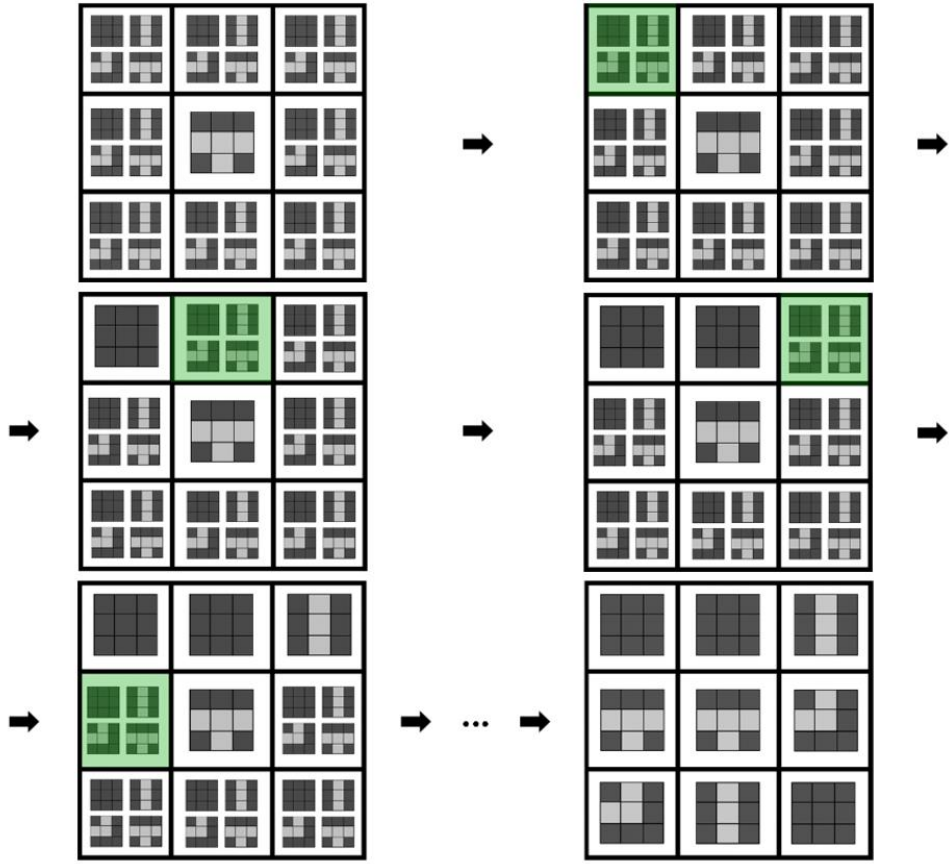
V – множина можливих для підстановки тайлів поточного елементу,

i – індекс поточного тайлу,

x та y – координати поточного елементу матриці.

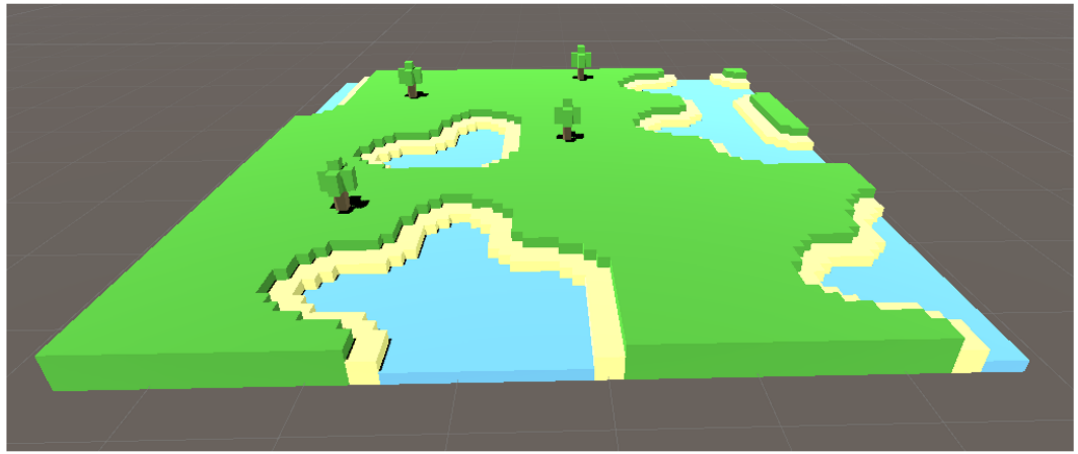
ПОСЛІДОВНИЙ МЕТОД ГЕНЕРАЦІЇ (2)

Приклад генерації мапи послідовним методом:



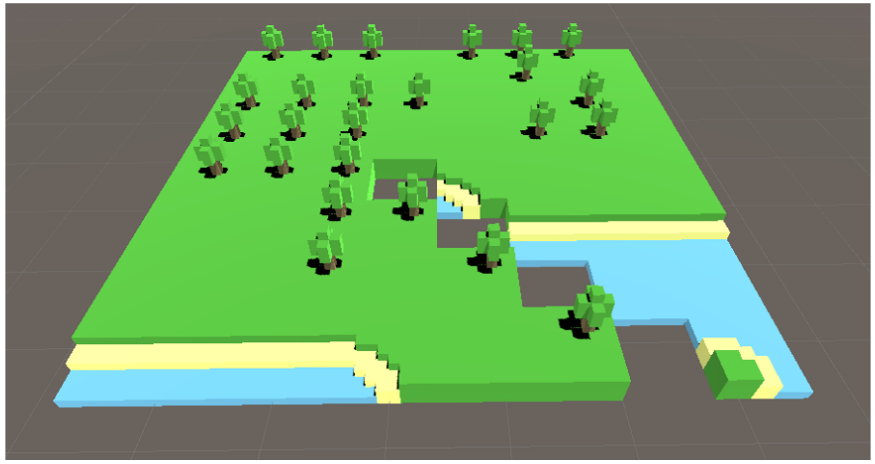
ПОСЛІДОВНИЙ МЕТОД ГЕНЕРАЦІЇ (3)

Приклад згенерованого ландшафту за допомогою воксельного тайлсету:



ПОСЛІДОВНИЙ МЕТОД ГЕНЕРАЦІЇ (4)

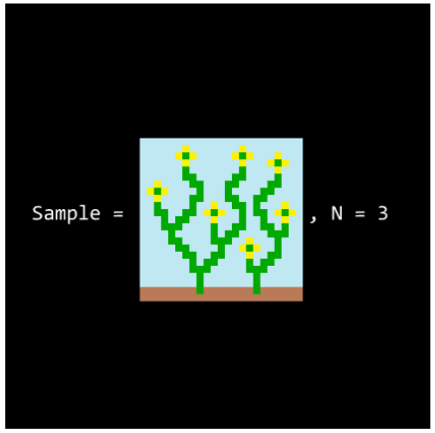
Проте доволі часто, при послідовній постановці тайлів, даний метод створює мапи із дірками:



МЕТОД ГЕНЕРАЦІЇ НА ОСНОВІ WFC (1)

WFC (від англ. Wave Function Collapse – колапс хвильової функції) – алгоритм, який генерує бітові зображення, локально подібні до вхідного бітового зображення.

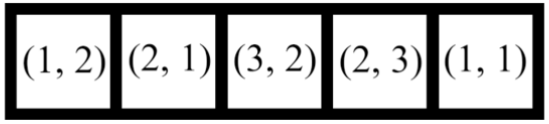
Приклад роботи Wave Function Collapse:



МЕТОД ГЕНЕРАЦІЇ НА ОСНОВІ WFC (2)

Для генерації ландшафтів без «дірок» пропонується реалізувати алгоритм на основі WFC, який працюватиме не з пікселями зображень, а з наданими 3D-тайлами.

Даний метод є ітераційним та використовує чергу для обходу матричних елементів.



Ітераційний процес ділиться на два основних етапи: оброблення черги генерації та колапсування тайлу.

МЕТОД ГЕНЕРАЦІЇ НА ОСНОВІ WFC (3)

Ітераційний процес починається із обробки всіх елементів у черзі генерації:

$$R \coloneqq RemoveImpossible(q_{n,m}, V)$$
$$q_{n,m} = R \Rightarrow Q \coloneqq Q \cup \{q_{n,m}\}$$
$$q_{n,m} \cap \bar{R} = \emptyset \Rightarrow Q \coloneqq Q \cup GetNeighbors(q_{n,m}, M)$$

$$n = i - d \,..\, i + d$$
$$m = j - d \,..\, j + d$$

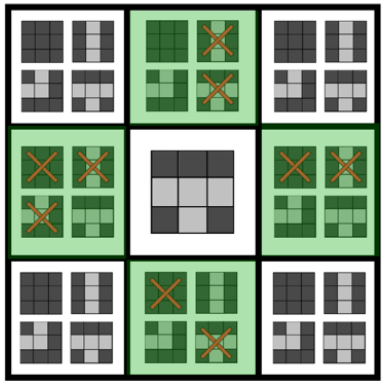
Q – черга обходу матричних елементів,

i, j – індекси поточного тайлу,

d – глибина WFC,

$RemoveImpossible(q_{n,m}, V)$ – повертає множину видалених тайлів, які неможливі для підстановки у поточний елемент.

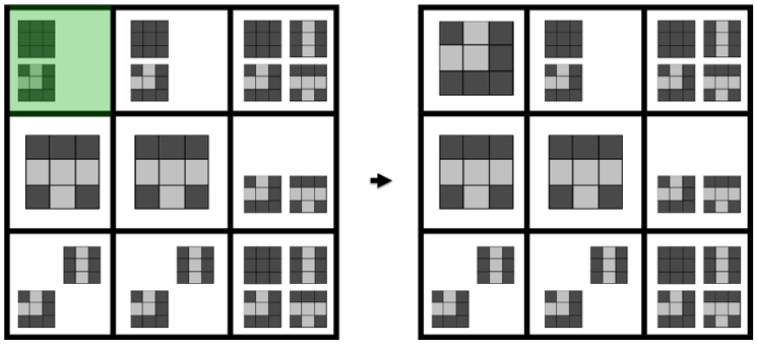
$GetNeighbors(q_{n,m}, M)$ – повертає множину сусідніх елементів поточного тайлу



МЕТОД ГЕНЕРАЦІЇ НА ОСНОВІ WFC (4)

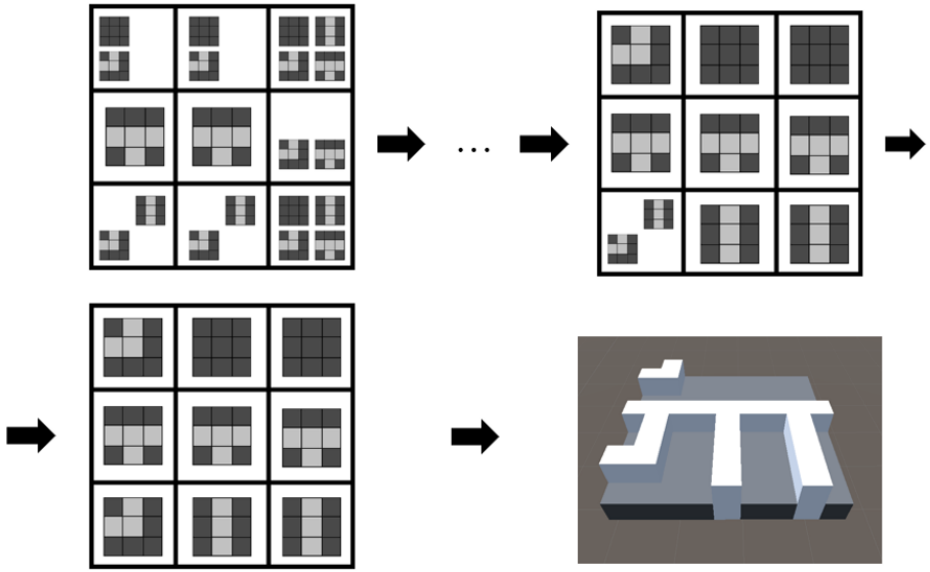
Колапсування тайлу (від англ. collapse) – процес перетворення множини елементів у матриці можливих тайлів до одного елементу.

```
/// <summary>
/// Collapse possible tiles in position to one tile
/// </summary>
/// <param name="position">Position in chunk to collapse</param>
private void CollapseTile(Vector2Int position)
{
    VoxelTile tileToCollapse = GetRandomTile(_possibleTiles[position.x, position.y]);
    _possibleTiles[position.x, position.y] = new List<VoxelTile> { tileToCollapse };
    AddNeighborsToGenerationQueue(position);
}
```



МЕТОД ГЕНЕРАЦІЇ НА ОСНОВІ WFC (5)

Ітераційний процес продовжується, поки у всіх елементах матриці не залишиться по одному тайлу, або кількість ітерацій перевищить допустиме значення.



ЗАПРОПОНОВАНИЙ МЕТОД

Етап 1. Отримання даних тайлсету

- Вхідні дані: Тайлсет (3D-моделі, колірний атлас)
- Результат: Сформована структура даних на основі вхідних даних

Етап 2. Валідація отриманих даних

- Вхідні дані: Наданий тайлсет
- Результат: Провалідований тайлсет – перевірені чи вірні елементи тайлсету було подано на вхід, розмір та кольори, тощо.

Етап 3. Оброблення отриманих даних

- Вхідні дані: Валідний тайлсет
- Результат: Закешований та розширений двосторонніми та чотиристоронніми тайлами тайлсет

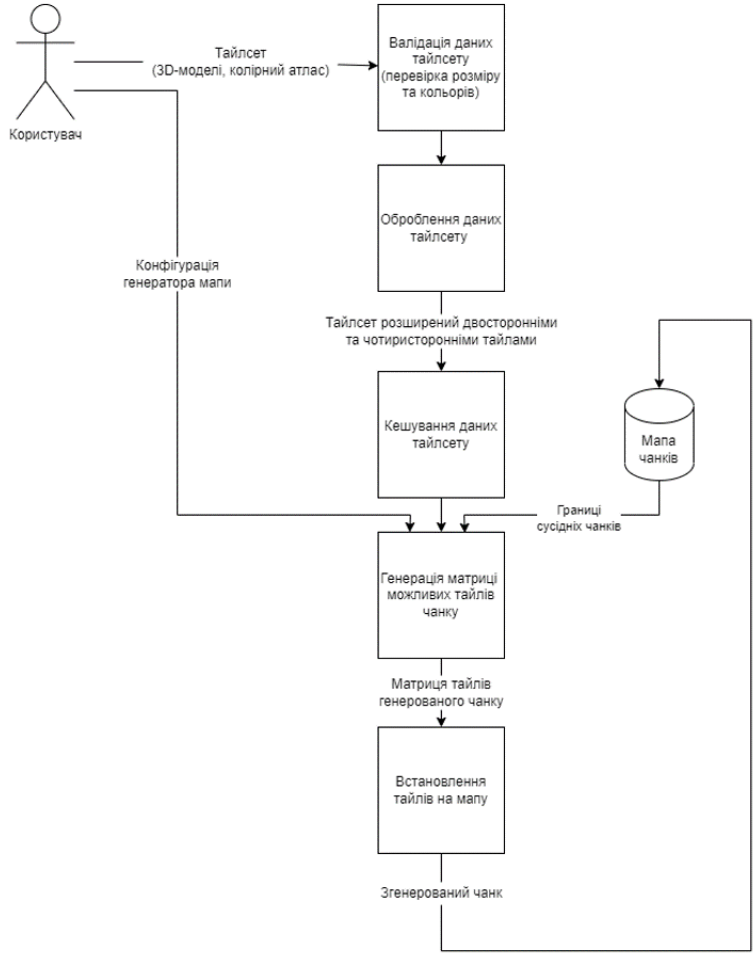
Етап 4. Генерація чанку комбінованим методом

- Вхідні дані: Оброблений тайлсет, границі сусідніх чанків
- Результат: Згенерована матриця тайлів чанку

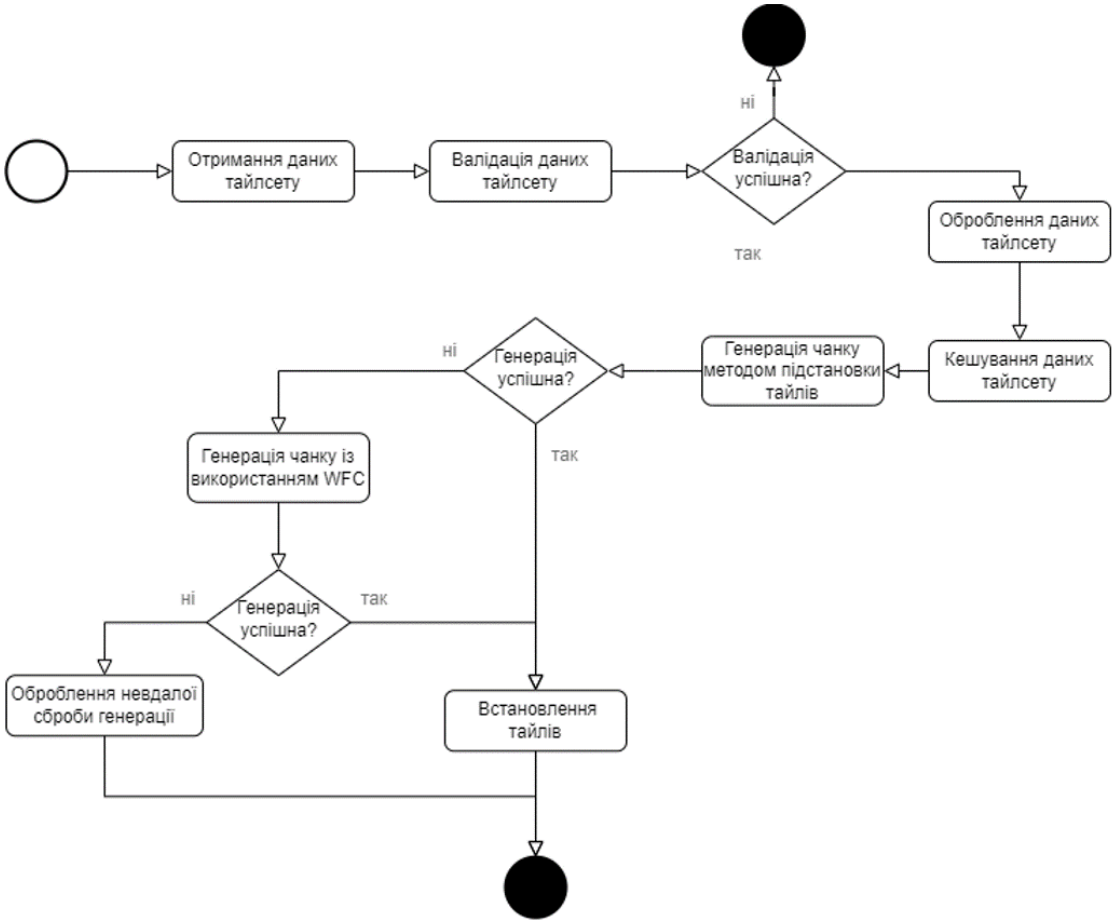
Етап 5. Встановлення тайлів

- Вхідні дані: Матриця тайлів генерованого чанку
- Результат: Згенерований чанк

ДІАГРАМА ПОТОКУ ДАНИХ



АЛГОРИТМ ПОБУДОВИ ВОКСЕЛЬНОГО ЛАНДШФАТУ

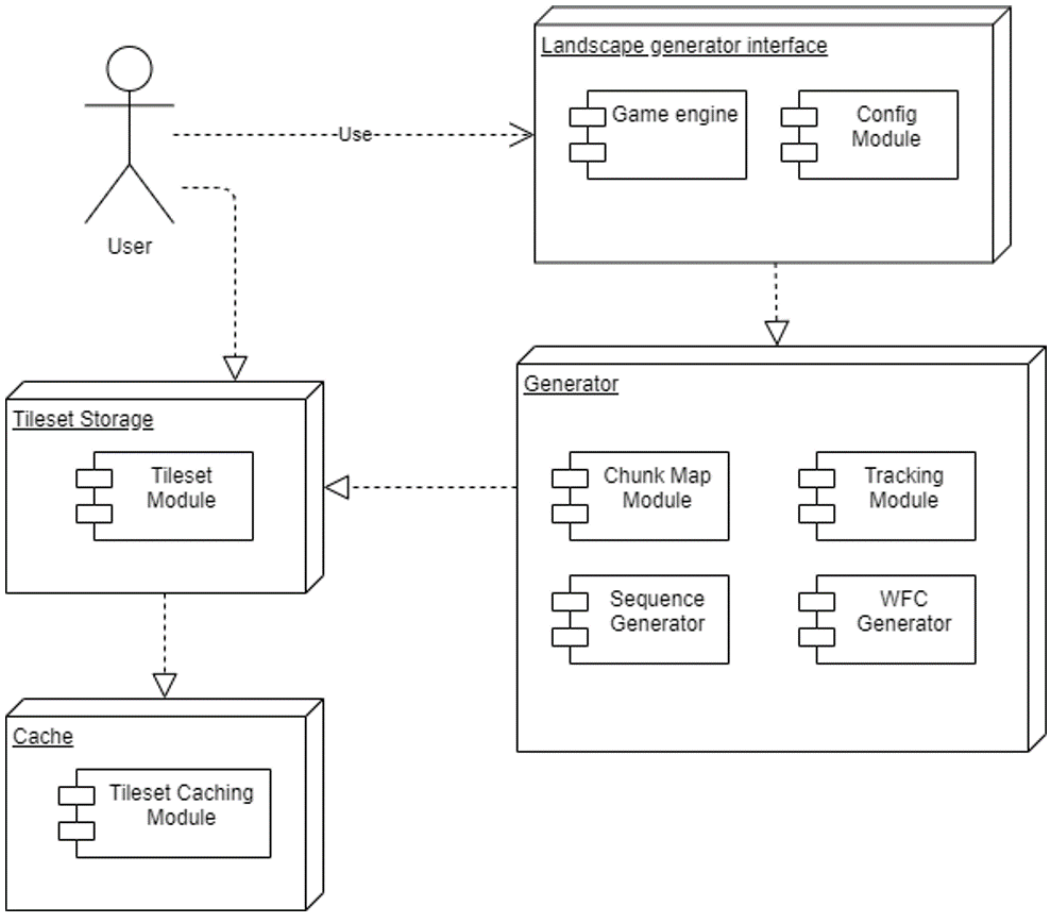


ОСОБЛИВОСТІ ЗАПРОПОНОВАНОГО МЕТОДУ

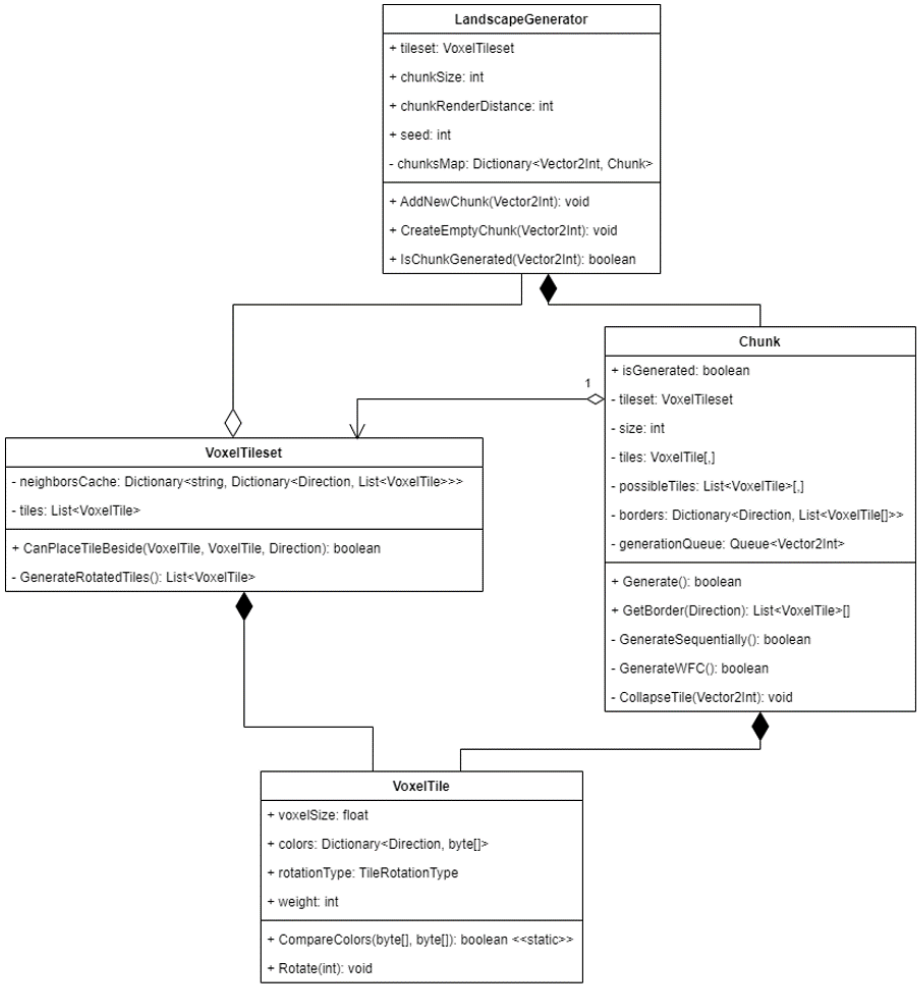
1. Метод використовує поєднання генерації за допомогою методу послідовної підстановки тайлів та методу із використанням колапсу хвильової функції.

2. Перша спроба генерації надається методу послідовної підстановки тайлів, якщо спроба генерації видалася невдалою, генерація мапи виконується за допомогою методу із використанням WFC.

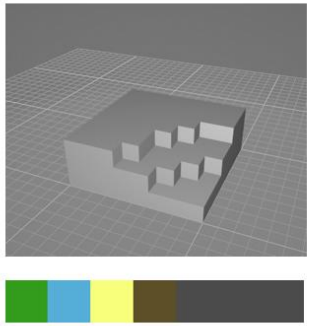
АРХІТЕКТЕРА ПРОГРАМНОГО ЗАБЕЗПЕЧАННЯ

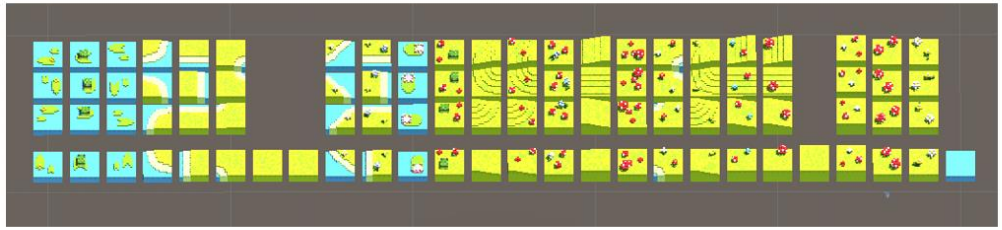


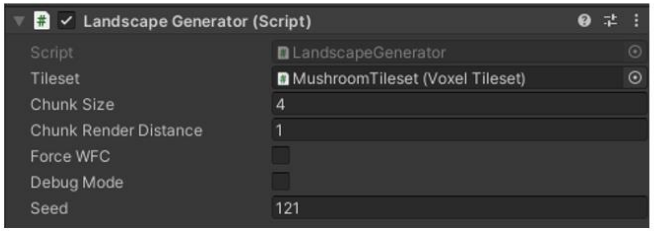
ДІАГРАМА КЛАСІВ АЛГОРИТМІЧНО-ПРОГРАМНОГО МЕТОДУ

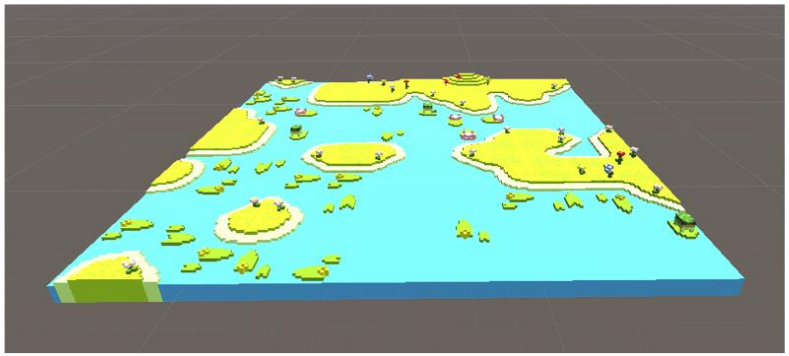


РЕЗУЛЬТАТ РОБОТИ МЕТОДУ









ПОРІВНЯЛЬНИЙ АНАЛІЗ РОЗРОБЛЕНОГО МЕТОДУ (1)

Оцінка часової ефективності методів генерації:

Номер тайлсету	Метод послідовної підстановки (без застосування WFC)	Метод із застосуванням WFC	Комбінований метод
1 (27 вокселів)	2.363мс	9.691мс	2.363мс, 0 чанків WFC
2 (512 вокселів)	-	11.559мс	4.161мс, 182 чанки WFC
3 (4,096 вокселів)	-	8440.334мс	6450.841мс, 689 чанки WFC

ПОРІВНЯЛЬНИЙ АНАЛІЗ РОЗРОБЛЕНОГО МЕТОДУ (2)

Оцінка часової ефективності методу генерації із застосуванням механізму кешування даних тайлсету:

Номер тайлсету	Комбінований метод без кешування даних про сусідів	Комбінований метод із застосуванням механізму кешування
1 (27 вокселів)	2.363мс	2.551мс
2 (512 вокселів)	4.161мс	4.033мс
3 (4,096 вокселів)	6450.841мс	6234.727 мс

НАУКОВА НОВИЗНА

Запропоновано алгоритмічно-програмний метод генерації воксельних ландшафтів, який відрізняється від існуючих застосуванням колапсу хвильової функції та методу послідовної генерації тайлів, що дозволяє підвищити швидкодію програмної обробки даних в середньому у 2 рази та зменшити вірогідність некоректної генерації мапи воксельного ландшафту.

АПРОБАЦІЯ

Основні положення та результати роботи були представлені на науковій конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг ПМК-2021» (Київ, 17-19 листопада 2021р.)

ПЕРЕВІРКА НА УНІКАЛЬНІСТЬ

9.5% Matches

Highest match: **4.08%** with Library source (File ID: **7843907**)

5.97% Internet sources	21	Page 51
8.29% Library sources	209	Page 51

0.48% Quotes

Quotes	1	Page 52

Exclusion of references is off

ВИСНОВКИ

1. Проведено дослідження підходів до процедурної генерації ландшафтів.
2. Проведено дослідження алгоритму колапсу хвильової функції.
3. Проаналізовано переваги та недоліки існуючих методів за визначеними критеріями.
4. Запропоновано алгоритмічно-програмний метод генерації воксельного ландшафту із 3D-тайлів.
5. Розроблено алгоритмічне та програмне забезпечення, що реалізує запропонований метод.
6. Проаналізовано ефективність запропонованого методу генерації ландшафту із 3D-тайлів.

Дякую за увагу!

Питання?