

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка інструменту 3D-ігрового рушія для гнучкої генерації
воксельних ландшафтів із використанням алгоритму Surface Nets

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших
авторів без відповідних
посилань.

Студент гр. ІПЗ-22-1

_____ / Д. Д. Аннаєв /

Керівник

кваліфікаційної роботи

/ Д. В. Швець /

Завідувач кафедри

/ А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій
Кафедра: Моделювання та програмного забезпечення
Ступінь вищої освіти: бакалавр
Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри
А. М. Стрюк
«__» _____ 20__ р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ПЗ-22-1 Аннаєву Даніілу Дмитровичу

1. Тема: Розробка інструменту 3D-ігрового рушія для гнучкої генерації воксельних ландшафтів із використанням алгоритму Surface Nets
затверджено наказом по КНУ № 284с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «10» червня 2026 р.
3. Вихідні дані: розроблена система повинна створювати, відображати та локально редагувати процедурний воксельний ландшафт у Unity 6000.3.2f1, підтримувати різні режими генерації, будувати поверхню методом Surface Nets, використовувати тайлову організацію з quadtree-стримінгом.
4. Зміст пояснювальної записки: виконати аналіз підходів до процедурної генерації ландшафту, подання воксельних поверхонь і просторової організації великих сцен; спроектувати архітектуру програмної системи; реалізувати генератор густини, Surface Nets мешер, quadtree-стримінг, інструменти редагування, візуалізацію води і матеріалів; виконати тестування режимів генерації, стримінгу та редагування.
5. Перелік ілюстративного матеріалу: загальна архітектура системи, функціональна модель з різними рівнями декомпозиції, принцип побудови вершини Surface Nets, схема поділу простору quadtree, скріншоти налаштувань системи, скріншоти роботи системи.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання
1	Огляд літератури за тематикою та збір даних	20.02.2026 – 28.02.2026
2	Проведення порівняльного аналізу підходів до генерації та побудови воксельних поверхонь	01.03.2026 – 04.03.2026
3	Підготовка матеріалів першого розділу роботи	05.03.2026 – 09.03.2026
4	Проектування архітектури та структур даних	10.03.2026 – 18.03.2026
5	Підготовка матеріалів другого розділу роботи	19.03.2026 – 24.03.2026
6	Реалізація модулів генерації, мешингу та стримінгу	25.03.2026 – 21.04.2026
7	Реалізація редактора, води, матеріалів і камери	22.04.2026 – 29.04.2026
8	Підготовка матеріалів третього розділу роботи	30.04.2026 – 11.05.2026
9	Тестування системи та аналіз результатів	12.05.2026 – 20.05.2026
10	Підготовка матеріалів четвертого розділу роботи	21.05.2026 – 26.05.2026
11	Оформлення пояснювальної записки	27.05.2026 – 05.06.2026

Дата видачі завдання: «15» лютого 2026 р.

Студент _____ / Д. Д. Аннаєв /

Керівник роботи _____ / Д. В. Швець /

РЕФЕРАТ

ВОКСЕЛЬНИЙ ЛАНДШАФТ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, SURFACE NETS, ПОЛЕ ГУСТИНИ, QUADTREE, LOD, UNITY, SCRIPTABLEOBJECT, МЕШ, РЕДАГУВАННЯ ЛАНДШАФТУ.

Пояснювальна записка: 124 с., 22 рис., 5 табл., 7 дод., 27 джерел.

Об'єктом розробки є програмне забезпечення для процедурної генерації, відображення та редагування воксельного ландшафту. Метою роботи є розроблення програмної системи для створення тривимірного ландшафту у середовищі Unity з використанням поля густини, алгоритму Surface Nets і quadtree-організації тайлів різного рівня деталізації.

Досягнення мети забезпечено аналізом підходів до процедурної генерації, проектуванням модульної архітектури, реалізацією налаштовуваного генератора висоти і густини, побудовою поверхні методом Surface Nets, організацією тайлів із LOD, асинхронною перебудовою мешів, створенням колайдерів поблизу спостерігача та реалізацією редактора ландшафту з кистю.

Розробка підтримує різні види генерації, параметризується через ScriptableObject WorldGenSettings, формує гори, рівнини, регіональні підняття і пониження, річкові долини, озера, детальний шум, водну площину, снігову маску та вершинне забарвлення поверхні. Система може застосовуватися як основа для процедурних ігрових світів, інтерактивних симуляцій, візуальних прототипів і дослідницьких інструментів процедурної генерації.

У результаті виконання роботи створено програмну систему, що дозволяє генерувати, відображати й локально редагувати воксельний ландшафт, а також розширювати реалізацію у напрямі збереження змін, складнішої гідрології, біомів, повноцінних тривимірних SDF-шарів і перенесення важких обчислень у паралельні обчислювальні механізми.

ABSTRACT

VOXEL TERRAIN, PROCEDURAL GENERATION, SURFACE NETS, DENSITY FIELD, QUADTREE, LOD, UNITY, SCRIPTABLEOBJECT, MESH, TERRAIN EDITING.

Thesis in 124 p., 22 fig., 5 tab., 7 app., 27 references.

The development objective is software for procedural generation, visualization, and editing of voxel terrain. The purpose of the work is to develop a Unity-based software system that represents terrain as a density field, reconstructs the surface using the Surface Nets algorithm, and organizes terrain tiles through a quadtree structure with several levels of detail.

The purpose is achieved through the analysis of procedural terrain generation methods, the design of a modular software architecture, the implementation of height and density generation, Surface Nets meshing, quadtree tile streaming, asynchronous tile rebuilding, collider generation near the observer, vertex-color visualization, and brush-based terrain editing.

The developed system supports the Combined, MountainsOnly, PlainsOnly, RiversOnly, LakesOnly, and NoiseOnly generation modes. It includes configurable WorldGenSettings parameters, terrain shape mixing, procedural mountains, plains, rivers, lakes, detail noise, water plane, snow mask and surface coloring. The system can serve as a basis for game worlds, simulations, visual prototypes, and interactive procedural generation tools.

As a result, the software system allows the creation, display, and local editing of procedural voxel terrain in Unity. It provides a basis for further development, including persistent edits, extended biome logic, more advanced hydrology, complete three-dimensional SDF layers, and parallel computation through Jobs, Burst, or compute shaders.

ЗМІСТ

ВСТУП.....	8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ НАПРЯМКУ РОЗРОБКИ	10
1.1 Аналіз задачі процедурної генерації ландшафту.....	10
1.2 Огляд алгоритмів побудови поверхні	11
1.3 Просторова організація великих ландшафтів	12
1.4 Вибір інструментів розробки	13
2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	15
2.1 Загальна архітектура системи.....	15
2.2 Функціональна модель програмної системи.....	16
2.3 Структура даних.....	19
2.4 Алгоритми модулів генерації ландшафту	20
2.4.1 Алгоритм генерації висоти та густини.....	20
2.4.2 Алгоритм побудови поверхні Surface Nets	21
2.4.3 Алгоритм quadtree-стримінгу та LOD.....	23
2.4.4 Усунення розривів між рівнями деталізації	24
2.4.5 Редагування ландшафту.....	25
3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ.....	26
3.1 Розробка модуля генерації ландшафту	26
3.1.1 Реалізація налаштувань генерації.....	26
3.1.2 Реалізація генератора SDF	27
3.1.3 Реалізація процедурних форм рельєфу	28
3.2 Розробка модулів мешингу та стримінгу	30
3.2.1 Реалізація Surface Nets	30
3.2.2 Реалізація тайлів ландшафту	31
3.2.3 Реалізація quadtree-стримінгу	31
3.3 Реалізація редагування ландшафту	33
3.4 Реалізація візуалізації води та матеріалів	34
3.5 Реалізація керування камерою.....	35
4 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ.....	36
4.1 Методика тестування	36
4.2 Тестування програмних модулів системи.....	36
4.2.1 Тестування режимів генерації	36
4.2.2 Тестування quadtree-стримінгу.....	40
4.2.3 Тестування редагування ландшафту	42
4.3 Аналіз отриманих результатів	44
ВИСНОВКИ	45
ПЕРЕЛІК ПОСИЛАНЬ	46
ДОДАТКИ	49

ПЕРЕЛІК СПЕЦІАЛЬНИХ ТЕРМІНІВ

API	Application Programming Interface
LOD	Level of Detail
SDF	Signed Distance Field
URP	Universal Render Pipeline
Воксель	Об'ємний елемент тривимірної сітки, що є аналогом пікселя у просторі
Тайл	Окремий квадратний фрагмент воксельного ландшафту, який генерується, оновлюється та відображається незалежно від інших частин світу.
Стримінг	Процес динамічного завантаження, створення та видалення фрагментів сцени.
Семплювання	Отримання значення функції або поля в конкретній точці простору.
Меш	Полігональна геометрична модель, що складається з вершин, ребер і трикутників та використовується для відображення поверхні.
Колайдер	Невидима поверхня, за якою фізична система визначає зіткнення об'єктів.
Рендеринг	Процес перетворення геометрії, матеріалів і освітлення на видиме зображення на екрані.
Шейдер	Програма, яка визначає, як поверхня або ефект виглядає під час рендерингу.

ВСТУП

Процедурна генерація ландшафтів є важливим напрямом розробки інтерактивних систем, оскільки вона дає змогу формувати великі просторові середовища на основі параметрів і алгоритмічних правил, а не лише шляхом ручного моделювання кожного фрагмента сцени [1]. Для ігрових світів, симуляцій і візуальних прототипів такий підхід забезпечує кероване різноманіття форм, можливість швидкого створення варіантів середовища та повторюваність результатів завдяки використанню зерна генерації.

Актуальність теми зумовлена потребою у програмних системах, які поєднують процедурне формування рельєфу, тривимірне представлення поверхні, інтерактивне редагування та потокове відображення фрагментів світу. Класичні висотні карти добре підходять для значної частини відкритих поверхонь, однак вони обмежені функціональним поданням висоти над площиною XZ і недостатньо гнучкі для редагування тривимірної геометрії.

Метою кваліфікаційної роботи є розроблення програмного забезпечення для процедурної генерації, відображення та редагування воксельного ландшафту в середовищі Unity з використанням алгоритму Surface Nets і quadtree-структури для просторової організації тайлів різного рівня деталізації.

Для досягнення поставленої мети необхідно проаналізувати підходи до процедурної генерації, розглянути методи побудови воксельних поверхонь, обґрунтувати вибір Unity, C#, Surface Nets і quadtree-стримінгу, спроектувати архітектуру системи, реалізувати модулі генерації, мешингу, стримінгу, редагування та візуалізації, а також виконати тестування працездатності розробленого програмного забезпечення.

Об'єктом розробки є програмне забезпечення для створення процедурного воксельного ландшафту. Предметом розробки є методи й програмні засоби формування поля густини, побудови поверхні методом Surface Nets, тайлового поділу світу, LOD-стримінгу та локального

редагування ландшафту. Результати роботи можуть застосовуватися як основа для процедурних ігрових світів, інтерактивних симуляцій, візуальних прототипів та інструментів для дослідження процедурної генерації.

Результатом є програмна система у Unity 6000.3.2f1, яка дозволяє налаштовувати параметри генерації, формувати різні типи рельєфу, будувати згладжену поверхню з поля густини, відображати ландшафт за допомогою тайлів різної деталізації, виконувати локальне редагування поверхні за допомогою кисті.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ НАПРЯМКУ РОЗРОБКИ

1.1 Аналіз задачі процедурної генерації ландшафту

Процедурна генерація контенту розглядається як створення ігрових або інтерактивних об'єктів шляхом застосування алгоритмів, що працюють на основі заданих параметрів, правил і випадкових або псевдовипадкових величин [1]. Для ландшафту це означає формування висоти, форми поверхні, водних об'єктів, дрібної деталізації та додаткових класифікаційних масок без необхідності ручного моделювання кожного елемента.

У процедурному моделюванні природних поверхонь широко застосовують ієрархічне поєднання базових форм, шумових функцій, масок і локальних модифікаторів, оскільки жоден окремий шумовий шар не здатний повністю описати різноманіття масштабів реального рельєфу [2]. Саме тому у розроблюваній системі рельєф формується як сума регіональних підйомів, рівнинних хвиль, гірських масивів, річкових долин, озерних улоговин та детального шуму.

Застосування псевдовипадкового зерна забезпечує отримання повторюваних результатів, що є важливим аспектом для тестування та налаштування систем. Якщо генератор побудовано на детермінованій основі, однакові параметри та ідентичне зерно повинні створювати однаковий світ, а зміни у окремих параметрах мають вплив лише на відповідні компоненти рельєфу.

У рамках аналізу питання процедурної генерації ландшафту слід розмежувати алгоритмічну складову та інженерну організацію коду. Алгоритм визначає математичні або логічні перетворення даних, тоді як програмна реалізація має забезпечувати повторюваність результату, контроль параметрів, прийнятний час перебудови та можливість подальшого розширення системи без порушення вже реалізованих модулів.

Проектне рішення базується на принципі локальності. Тайли мають формуватися незалежно, проте мають забезпечувати узгоджений результат на межах. Це вимагає застосування однакових методів семплювання густини, використання додаткової крайової області та стабільної ідентифікації фрагментів світу за допомогою координат і рівнів деталізації.

Особливу увагу приділено тому, щоб процедурна генерація ландшафту не була жорстко прив'язана до одного набору параметрів. Налаштування винесено в централізовану структуру, а функції генерації побудовано так, щоб зміна зерна, масштабу шуму, рівня води або розміру тайла призводили до передбачуваної зміни результату без необхідності переписувати логіку інших модулів.

Практична цінність такого рішення полягає у тому, що воно закладає основу для подальшого розвитку процедурних світів. На цій базі можливо додавати біоми, складніші гідрологічні правила, збереження локальних змін, додаткові шари SDF, а також переносити ресурсоємні елементи до паралельних обчислювальних механізмів.

1.2 Огляд алгоритмів побудови поверхні

Для перетворення поля густини на полігональну поверхню розглянуто методи Marching Cubes, Surface Nets та Dual Contouring. Метод Marching Cubes створює поверхню на основі локальної конфігурації знаків у вершинах комірки і є одним із найвідоміших підходів для реконструкції ізоповерхонь [8]. Surface Nets розміщує одну вершину в активній комірці, де поверхня перетинає її ребра, і з'єднує такі вершини між сусідніми комірками, що забезпечує утворення компактної згладженої сітки [9].

Dual Contouring використовує інформацію про перетини та нормалі і здатний краще зберігати гострі особливості форми, однак його повна реалізація потребує коректного збору Hermite-даних (набір даних про точку перетину поверхні з ребром комірки) і стабільного розв'язання локальних задач оптимізації [10].

Таблиця 1.1 – Порівняння алгоритмів побудови ізоповерхні

Алгоритм	Якість поверхні	Кількість вершин	Придатність для LOD
Marching Cubes	Груба, потребує менших розмірів вокселів для збереження деталізації	Висока, кількість вершин не залежить від складності поверхні.	Низька, оскільки алгоритм погано масштабується під роботу з різними розмірами комірок воксельної сітки
Surface Nets	Гладка поверхня, яка може бути утворена навіть при великому розмірі вокселів	Низька, одна вершина на активну комірку	Висока, але потребує вирішення проблем зі стикуванням тайлів різного рівня деталізації.
Dual Contouring	Гладка, гарно відображає гострі грані, але може утворювати надмірну кількість геометрії	Середня, може утворювати велику кількість вершин на переломах форми.	Висока, але так само, як і Surface Nets, потребує узгодження на межах тайлів різного рівня деталізації та вирішення проблем зі швом.

У розроблюваній системі перевагу надано Surface Nets, оскільки він забезпечує прийнятний баланс між якістю поверхні, зрозумілістю реалізації, складністю готової ізоповерхні та зручністю тайлової побудови.

1.3 Просторова організація великих ландшафтів

Великі сцени доцільно не планувати як єдине ціле, оскільки будь-яка зміна або переміщення спостерігача потребувало б суттєвого перепланування надмірної кількості даних. Виявляється ефективним поділити простір на незалежні сегменти: чанки або тайли. Для ефективного управління різними рівнями деталізації в площині XZ доцільно використовувати quadtree, який рекурсивно ділить квадратну область на чотири дочірні квадранти [11].

У рендерингу ландшафтів застосування рівня деталізації (LOD) дозволяє зменшити геометричне навантаження для віддалених ділянок сцени, зберігаючи при цьому високий рівень деталізації поблизу до камери [12]. Ідея використання вкладених або ієрархічних сіток також знаходить застосування у методах потокового рендерингу великих ландшафтів [13]. Для розроблюваної системи quadtree обрано саме для горизонтальної площини,

оскільки тайли організовані за координатами XZ, а вертикальний діапазон визначається під час побудови меша.

Поточний стан системи дозволяє здійснити подальший перехід від підходу quadtree до octree, що дасть змогу застосовувати рівні деталізації також для вертикального діапазону світу XY та ZY. Це сприятиме значному збільшенню максимальної висоти світу (у поточному стані вона обмежена 768 воксельними комірками для забезпечення прийнятної швидкості побудови поля та меша), а також покращенню загальної продуктивності системи.

1.4 Вибір інструментів розробки

Проект виконано у Unity 6000.3.2f1, оскільки Unity надає редактор сцени, компонентну модель, C#-скрипти, API для роботи з Mesh, колайдерами, матеріалами та вхідними пристроями [17]. Для реалізації логіки обрано C#, оскільки ця мова використовується у Unity як основна мова скриптів, має статичну типізацію, підтримку об'єктно-орієнтованого програмування, колекцій, структур, асинхронних задач і зручну інтеграцію з .NET [18].

Побудова геометрії виконується через Mesh API Unity, який дозволяє програмно задавати вершини, індекси трикутників, UV-координати, кольори, нормалі та межі меша [19]. Налаштування генератора винесено у

ScriptableObject, що дає змогу зберігати параметри як окремий асет (ресурс проєкту Unity, наприклад матеріал, налаштування, модель, текстура або ScriptableObject) і змінювати їх без жорсткого вшивання у код [20].

Колайдери створюються за допомогою MeshCollider, а керування камерою та редактором здійснюється за допомогою Unity Input System [21][22]. Для матеріалів і шейдерів застосовано Universal Render Pipeline, який орієнтований на сучасний рендеринг у Unity-проєктах [23].

Unity вибрано як середовище, що поєднує редактор сцени, систему компонентів, API для створення мешів, колайдерів і матеріалів, а також засоби для інтерактивного керування. Це дає можливість розробляти не лише

генератор даних, а й повноцінну систему перегляду та редагування ландшафту.

Мова С# використовується для реалізації генератора, мешера, стримера та редакторських інструментів. ScriptableObject застосовано для винесення параметрів світу в окремий ресурс, що спрощує налаштування режимів генерації без зміни вихідного коду.

URP, Mesh API, MeshCollider та Input System забезпечують практичну інтеграцію результату в сцену Unity. Завдяки цим засобам одна й та сама система може генерувати геометрію, задавати матеріали, обробляти введення користувача і створювати колізії поблизу спостерігача.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Загальна архітектура системи

Архітектура програмної системи побудована за модульним принципом. Окремі компоненти відповідають за налаштування генерації, семплування висот і густини, змішування форм рельєфу, побудову меша, просторовий поділ, стримінг, локальне редагування, візуалізацію води та матеріалів, а також керування камерою. Такий поділ відповідає загальним принципам інженерії програмного забезпечення, за якими складну систему розбивають на взаємопов'язані частини з визначеними ролями [24].

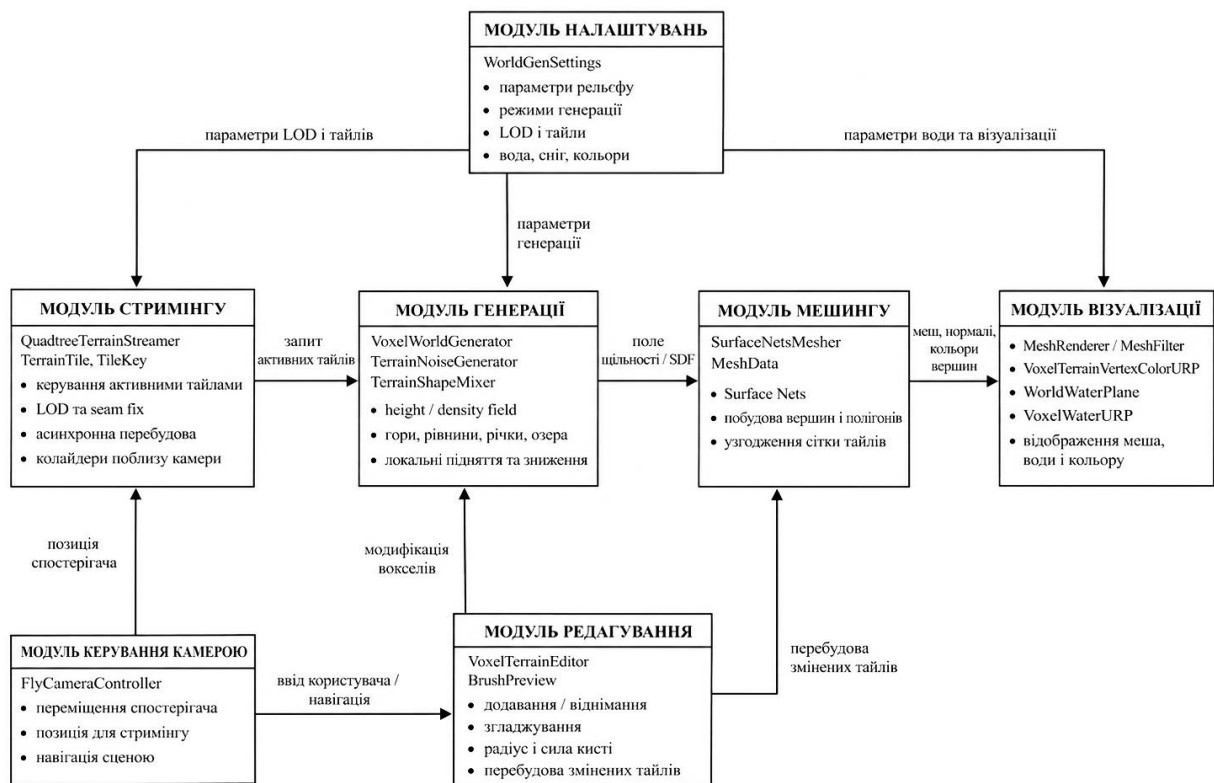


Рисунок 2.1 – Загальна архітектура програмної системи

Архітектура системи структурована на модулі з чітко визначеними зонами відповідальності. Параметри світу зберігаються окремо від алгоритмів генерації, мешинг відокремлений від стримінгу, а інструменти редагування взаємодіють із генератором через операції зміни густини. Такий поділ сприяє зменшенню зв'язності між компонентами. Наприклад, зміна параметрів

матеріалів або води не потребує перероблення мешу Surface Nets, а зміна критерію поділу quadtree не впливає на формули генерації рельєфу. У результаті система залишається відкритою для розширення: до неї можна додавати нові шари рельєфу, інші матеріали, варіанти рівнів деталізації (LOD) або способи збереження змін без руйнування базового конвеєра.

2.2 Функціональна модель програмної системи

Функціональна модель відображає загальну логіку перетворення даних у системі процедурної генерації воксельного ландшафту. Вхідними даними є параметри генерації та дії користувача. Керування процесом здійснюється через правила побудови рельєфу, правила LOD, узгодження меж, а також параметри візуалізації. Механізмами виконання є генерація рельєфу, мешинг, стримінг і редагування. Результатом роботи системи є набір фрагментів ландшафту, поверхневі меші з колайдерами та готові об'єкти сцени Unity.



Рисунок 2.2 – Функціональна модель програмної системи, рівень A0

Контекстна функціональна модель (див. рисунок 2.2) узагальнює процес генерації воксельного ландшафту як єдину функцію A0, що приймає параметри генерації та дії користувача, підпорядковується правилам побудови рельєфу, LOD, узгодження меж і візуалізації, а на виході формує фрагменти ландшафту, меші, колайдери та об'єкти сцени Unity. Такий рівень моделі показує межі системи та основні типи даних, що входять до процесу або створюються внаслідок його виконання.

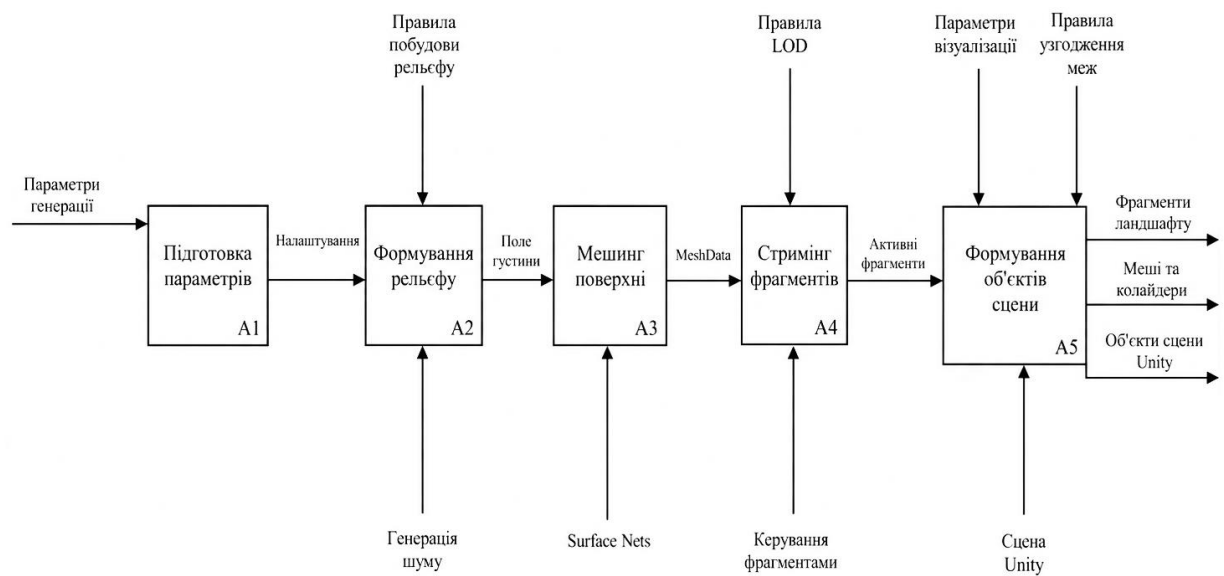


Рисунок 2.3 – Функціональна модель програмної системи, декомпозиція A0

Процес починається з підготовки параметрів, під час якого початкові налаштування трансформуються в узгоджений набір даних для подальшої генерації. Наступним кроком є формування рельєфу, у межах якого правила побудови та генерація шуму визначають поле густини. Отримане поле густини передається на етап мешингу поверхні, де алгоритм Surface Nets перетворює його на проміжне геометричне подання. На цьому етапі важливо, що система працює не з готовим візуальним об'єктом, а з даними, які описують форму майбутньої поверхні та можуть бути передані наступним модулям без прив'язки до конкретного об'єкта сцени. Після цього на основі стримінгу фрагментів визначається, які частини ландшафту мають бути активними відповідно до правил LOD. Такий підхід дозволяє не створювати весь світ

одночасно, а підтримувати лише ті фрагменти, які потрібні для поточного положення спостерігача. Додатково, правила узгодження меж зменшують ризик появи розривів між сусідніми фрагментами з різним рівнем деталізації. Завершальним етапом є формування об'єктів сцени, у рамках якого активні фрагменти перетворюються на видимі елементи Unity-сцени з мешами, колайдерами та візуальними параметрами.

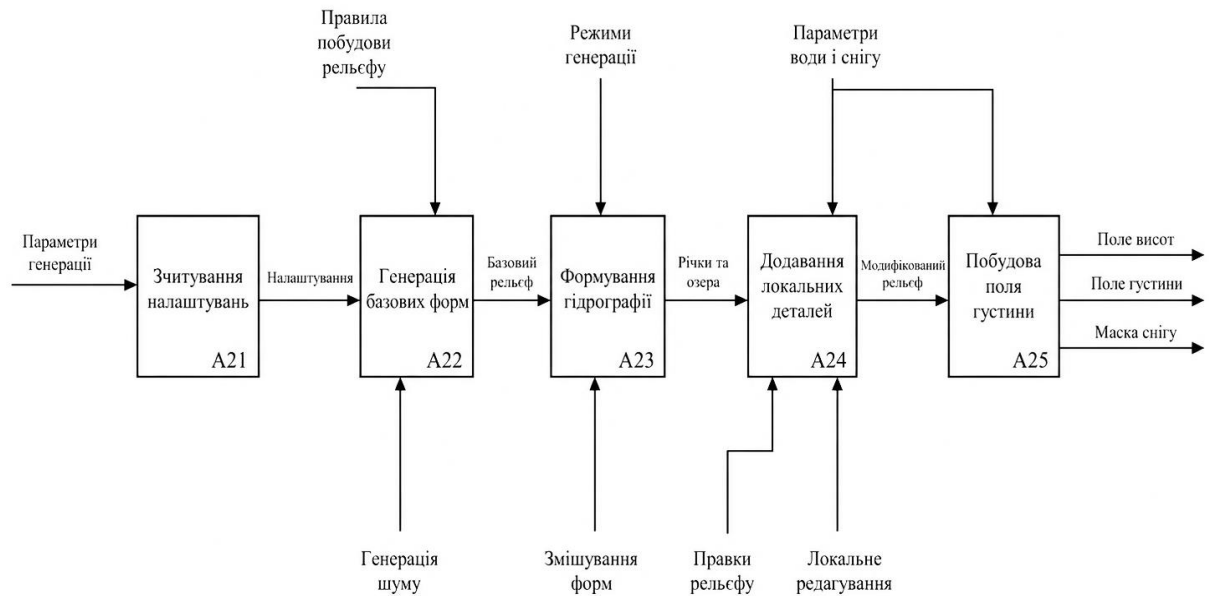


Рисунок 2.4 – Функціональна модель програмної системи, декомпозиція A2

Для побудови рельєфу система спочатку зчитує параметри генерації та переводить їх у набір налаштувань, придатних для обчислення висот. Після цього виконується генерація базових форм рельєфу, таких як рівнини, підняття, гірські масиви та інші великі структуровані елементи. Наступним кроком формується гідрографія, зокрема річки та озера, які змінюють початковий рельєф через вирізання русел, долин і водойм. Далі додаються локальні деталі та враховуються модифікації рельєфу, виконані користувачем. Завершальним етапом є побудова поля густини, яке слугує основою для формування поля висот, поля густини та маски снігу. У рамках цієї функціональної моделі параметри генерації визначають числові характеристики світу, режими генерації, рівень води та снігу, а також деталізацію й візуальне оформлення. Генерація шуму та змішування форм

створюють базову висотну структуру ландшафту, а локальні деталі уточнюють форму поверхні. Поле густини використовується як основа для мешинга: поверхня виділяється за переходом між заповненим і порожнім простором, а результатом цього етапу є проміжне геометричне подання з вершинами, трикутниками, UV-координатами та кольорами. Після побудови геометричних даних система стримінгу визначає набір активних фрагментів ландшафту, враховуючи положення спостерігача, видиму дистанцію, рівень деталізації та правила узгодження меж між сусідніми фрагментами. Активні фрагменти передаються на етап формування об'єктів сцени, де геометрію застосовують до компонентів візуалізації і, за потреби, до фізичних колайдерів.

2.3 Структура даних

У системі використовуються прості структури для ідентифікації тайлів, проміжного збереження меша, фіксації операцій редагування та організації черг побудови.

Таблиця 2.1 – Основні структури даних програмної системи

Структура	Призначення	Основні поля	Роль у системі
TileKey	Ідентифікація тайла	X, Z, Size, Lod	Ключ словників активних тайлів
MeshData	Проміжне збереження меша	Vertices, Triangles, UVs, Colors	Передача даних від мешера до Unity Mesh
BrushOperation	Опис локальної зміни густини	position, radius, strength, mode	Редагування поля густини
WorldGenSettings	Параметри генерації	worldSize, worldHeight, generationMode, seed	Керування генератором
Черги та множини стримера	Контроль побудови тайлів	desiredSet, buildQueue, queuedSet, buildingSet	Узгодження активних, бажаних і створюваних тайлів

Структури даних у системі ретельно підібрано з метою забезпечення швидкого доступу до активних тайлів, стабільної ідентифікації фрагментів світу та ефективного передавання результатів обмінів між модулями. Найважливішими компонентами є TileKey, MeshData, BrushOperation та

набори станів для черги побудови. TileKey об'єднує координати, розмір та рівень деталізації (LOD) тайла, що дозволяє використовувати його як ключ у словниках. MeshData акумулює вершини, трикутники, UV-координати та кольори до моменту формування Unity Mesh, що відділяє алгоритм Surface Nets від конкретного компонента сцени. Для редагування застосовується список BrushOperation, із якого формуються знімки операцій. Це дає змогу враховувати локальні зміни під час семплювання густини й перебудовувати лише ті області, на які вплинула кисть.

2.4 Алгоритми модулів генерації ландшафту

2.4.1 Алгоритм генерації висоти та густини

Висота ландшафту формується поетапно. Спочатку задається базова висота, після чого додаються регіональне підняття або пониження, рівнинні хвилі, гірські масиви, озерні улоговини, детальний шум і вплив річок. Після змішування шарів результат обмежується м'яким верхнім порогом, щоб уникнути різких перевищень worldHeight.

Перехід від висотної функції до поля густини виконується за формулою:

$$d(x, y, z) = h(x, z) - y + \Delta b(x, y, z), \quad (2.1)$$

де $d(x, y, z)$ – значення густини в точці простору; $h(x, z)$ – процедурно сформована висота поверхні; y – вертикальна координата; $\Delta b(x, y, z)$ – сумарний внесок операцій кисті. Додатне значення густини відповідає матеріалу, а від'ємне — порожньому простору.

Генерація розпочинається з базової висоти, до якої додаються регіональні підняття або пониження, рівнинні хвилі, гірські масиви, озерні басейни, річкові долини та детальний шум. Кожен шар має свої власні параметри масштабу й сили, що дозволяють окремо керувати характером рельєфу. Після формування висоти відбувається перехід до поля густини: точки нижче поверхні отримують позитивне значення, а точки вище —

від’ємне. Саме зміна знаку густини використовується у Surface Nets для пошуку ізоповерхні.

Операції з кистю не переписують базову карту висот, а додають локальну поправку до густини. Це забезпечує можливість накопичення редагувань поверхні та їх відображення під час наступної перебудови відповідних тайлів.

2.4.2 Алгоритм побудови поверхні Surface Nets

Побудова поверхні тайла методом Surface Nets починається з вибору області тайла, додавання gutter-комірок і оцінки вертикального діапазону. Далі відбувається семплювання густин у вузлах ґратки, пошук комірок зі зміною знака, визначення перетинів поверхні з ребрами, усереднення точок перетину, створення вершини, побудова граней і передача результату до MeshData.

Наведемо приклад програмної реалізації побудови поверхні тайла методом Surface Nets:

```

Input: generator, originX, originZ, tileSize, worldHeight, step
Add one-cell gutter around tile
Estimate yStart and yEnd for the tile
Sample density at all grid nodes
for each cell in grid:
    collect eight corner densities
    if signs do not differ: continue
    crossings = intersections on crossed edges
    vertex = average(crossings)
    store vertex index for the cell
for each axis X, Y, Z:
    if adjacent density samples cross the surface:
        collect four neighboring cell vertices
        orient quad by estimated normal
        add triangles to MeshData
return MeshData

```

Рисунок 2.5 – Побудова поверхні тайла методом Surface Nets

Алгоритм відображає загальну логіку реалізації без другорядних перевірок, потрібних у конкретному C#-кодi. У програмній системі ці кроки

доповнено нормалізацією майже нульових густин, перевіркою меж тайла та оцінюванням нормалей для коректної орієнтації граней.

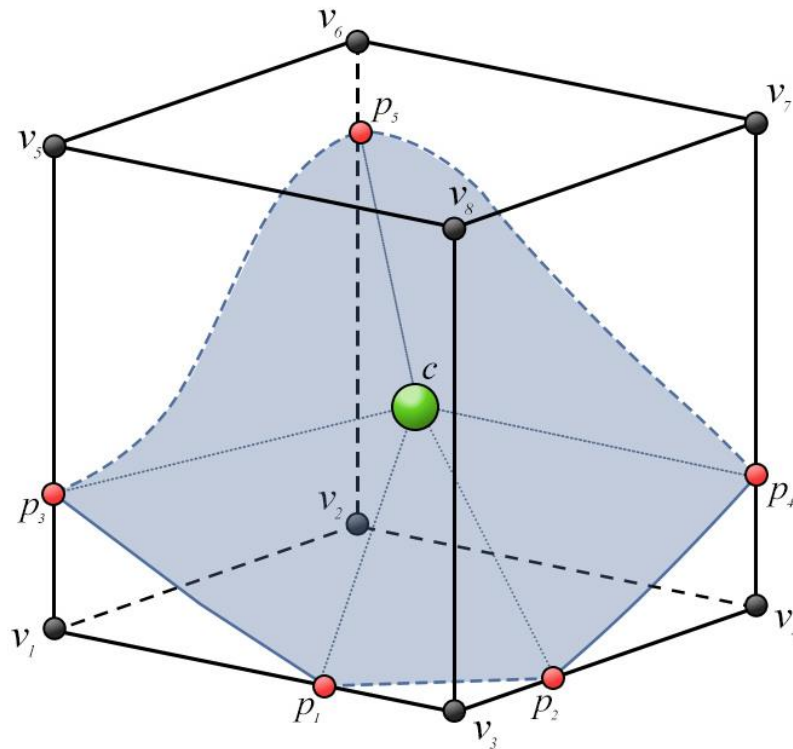


Рисунок 2.6 – Пошук вершини за алгоритмом Surface Nets

На рисунку показано одну воксельну комірку, у вершинах якої задані значення скалярного поля. Точки v_i відповідають вершинам комірки, точки p_i позначають місця перетину ізоповерхні з ребрами комірки, отримані шляхом лінійної інтерполяції між сусідніми значеннями поля з різними знаками. Напівпрозора область умовно відображає фрагмент ізоповерхні всередині комірки. Тоді алгоритм можна представити у вигляді формули (відображено пошук позиції вершини для однієї з координат X, Y або Z):

$$c = \frac{1}{n} \sum_{i=1}^n p_i \quad (2.2)$$

де n – кількість точок перетину ізоповерхнею ребер комірки вокселя, p_i – координати таких точок, c – координата вершини Surface Nets для даної комірки.

Таким чином, головною перевагою даного алгоритму є можливість отримати компактне представлення поверхні одним вузлом для кожної активної воксельної комірки, що є простішим у реалізації та створює меншу кількість надлишкової геометрії порівняно з іншими алгоритмами мешингу ізоповерхонь.

У реалізованій програмній системі алгоритм Surface Nets оперує з локальною областю тайла, до якої додаються крайові комірки. Такий запас необхідний для узгодженого пошуку перетинів на межах і для того, щоб сусідні тайли використовували спільні значення густини. Після створення вершин виконуються проходи побудови граней уздовж осей X, Y та Z. Орієнтація трикутників уточнюється за допомогою оціненої нормалі, таким чином, поверхня отримує коректну орієнтацію для освітлення та вершинного фарбування.

2.4.3 Алгоритм quadtree-стрімінгу та LOD

Quadtree-стрімінг у системі розглядає ландшафт не як суцільний меш, а як ієрархічну структуру квадратних ділянок у площині XZ. Початковим елементом цієї ієрархії є root tile, розмір якого відповідає межах усього світу, встановленим у параметрах генерації. Кожен вузол дерева описується координатами початкової точки, розміром та рівнем деталізації, що дозволяє однозначно ідентифікувати один і той самий фрагмент простору незалежно від того, чи він вже вставлений у сцену. На етапі оновлення система не створює геометрію безпосередньо, а спершу формує цільове покриття світу: перевіряє, які ділянки потрапляють до зони видимості, які з них потребують деталізації, а які можна залишити на поточному рівні поділу. Такий підхід відокремлює логіку вибору потрібних фрагментів від їх фактичної побудови, що дозволяє керувати навантаженням на кадр і поступово оновлювати сцену без повної регенерації всього ландшафту. Після завершення обходу дерева отриманий набір цільових тайлів використовується як основа для синхронізації з уже наявними об'єктами сцени.

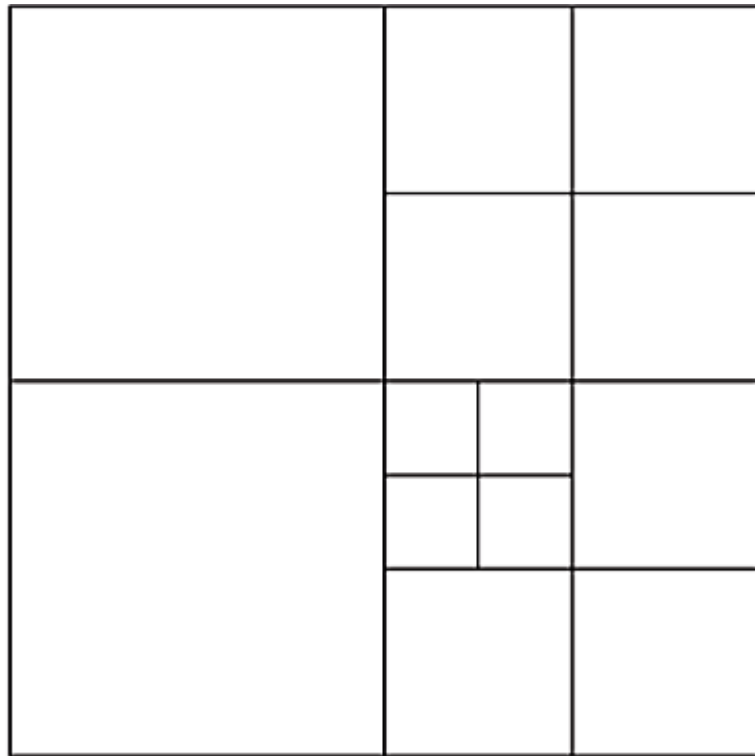


Рисунок 2.7 – Схема quadtree на площині XZ

Quadtree-стрімінг визначає, які тайли потрібні навколо камери в поточний момент. Близьні області розбиваються на менші фрагменти з нижчим кроком семплювання, а віддалені залишаються більшими, що зменшує кількість геометрії на периферії. Набір *desired tiles* обчислюється рекурсивно, після чого порівнюється з уже активними тайлами. Нові фрагменти додаються до черги побудови, а застарілі вилучаються лише після завершення актуальних задач, щоб уникнути помітних прогалин у сцені. LOD у цій реалізації безпосередньо пов'язаний із значенням кроку. Чим вищий рівень деталізації за відстанню, тим більший крок використовується під час *Surface Nets*, а отже зменшується кількість комірок і вершин.

2.4.4 Усунення розривів між рівнями деталізації

Коли сусідні тайли мають суттєво різний LOD, їхні кроки дискретизації не збігаються, що може створювати видимі щілини або різкі переходи на межах. Реалізований механізм не є універсальним математичним рішенням усіх можливих випадків, але зменшує кількість проблемних меж, обмеживши різницю LOD між сусідами. Розриви між тайлами виникають тоді, коли сусідні

фрагменти побудовано з різним кроком семплювання. Їхні вершини на межі можуть не збігатися, тому між поверхнями з'являються щілини або різкі зміщення

Практично це реалізовано через перевірку дотику тайлів і додаткове розбиття надто грубого сусіднього тайла. Якщо різниця LOD перевищує одиницю, грубіший тайл замінюється чотирма дочірніми тайлами. Такий підхід підтримує поступову зміну деталізації та знижує ризик візуальних розривів.

2.4.5 Редагування ландшафту

Редагування ландшафту виконується через raymarching у полі густини. Промінь від камери проходить через простір із заданим кроком, а момент перетину поверхні визначається за зміною знака густини.

Операції кистю записуються як BrushOperation, після чого перебудовуються лише ті тайли, що перетинають область кисті. Візуальний контур кисті формується окремим компонентом BrushPreview, що дає користувачу зворотний зв'язок щодо позиції, радіуса та сили дії.

Редагування ландшафту побудоване навколо роботи з тим самим полем густини, яке використовується для генерації поверхні. Користувач не переміщує окремі вершини меша, а додає операції, що змінюють густину у сферичній області навколо точки контакту.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Розробка модуля генерації ландшафту

3.1.1 Реалізація налаштувань генерації

Налаштування генерації реалізовано у класі WorldGenSettings, який успадковується від ScriptableObject. У цьому класі зосереджено параметри розміру світу, висоти, розміру вокселя, режиму генерації, зерна генерації, базової висоти, регіонального підняття, рівнин, гір, річок, озер, детального шуму, снігу, води, кольорів поверхні, параметрів Surface Nets і матеріалів. Фрагмент структури режимів генерації реалізовано наступним чином:

```
public enum TerrainGenerationMode
{
    Combined,
    MountainsOnly,
    PlainsOnly,
    RiversOnly,
    LakesOnly,
    NoiseOnly
}
```

Рисунок 3.1 – Структура режимів генерації

Наведений перелік режимів дає змогу використовувати один генератор у різних конфігураціях без дублювання коду. Під час тестування це дозволяє окремо перевіряти гірські форми, рівнини, річкові долини, озера, шумові форми та комбінований режим.

Таблиця 3.1 – Основні параметри генерації ландшафту

Група параметрів	Приклади полів	Призначення
World	worldSize, worldHeight, voxelSize	Визначення розміру світу і масштабу
Mode	generationMode, seed	Вибір режиму та повторюваність генерації
Regional Height	regionalScale, regionalUplift, regionalDepression	Формування великих піднять і понижень
Terrain Layers	plains, mountains, rivers, lakes, detailNoise	Змішування основних форм рельєфу
Surface	waterLevel, colors, snow	Матеріали і вода

Продовження таблиці 3.1

Група параметрів	Приклади полів	Призначення
Surface Nets	defaultTileSize, defaultStep, generateCollider	Параметри побудови тайлів і колайдерів

Метод OnValidate обмежує некоректні значення, наприклад, занадто малий розмір тайла або від’ємний масштаб шуму. Це підвищує стабільність генерації, оскільки частина помилок параметризації усувається ще до запуску сцени.

3.1.2 Реалізація генератора SDF

Клас VoxelWorldGenerator є центральною точкою доступу до висоти та густини. Він містить методи SampleHeight, SampleDensity, SampleDensityFromKnownHeight, механізм кешування висот, контроль зміни налаштувань через хеш, знімок brush operation snapshot та метод EvaluateSurfaceColor для вибору кольору поверхні.

Функція SampleDensity реалізує базове перетворення висоти у густину та додає вплив операцій кисті:

```
public float SampleDensity(float x, float y, float z)
{
    float height = SampleHeight(x, z);
    float density = height - y;

    if (enableBrushOperations)
        density += SampleBrushDensityDelta(x, y, z);

    return density;
}
public float SampleDensityFromKnownHeight(float x, float y, float z, float knownHeight)
{
    float density = knownHeight - y;

    if (enableBrushOperations)
        density += SampleBrushDensityDelta(x, y, z);

    return density;
}
```

Рисунок 3.2 – Функція SampleDensity

Застосування кешу висот зменшує кількість повторних викликів процедурних шумових функцій під час побудови сусідніх точок і тайлів. Кеш

очищується після зміни ключових налаштувань, щоб система не використовувала застарілі значення.

Операції кисті враховуються у функції густини як додаткова локальна поправка. Завдяки цьому Surface Nets отримує вже змінене поле, а редагування природньо включається у той самий механізм побудови поверхні.

Клас TerrainShapeMixer визначає, які шари рельєфу беруть участь у формуванні висоти для кожного режиму генерації. У режимі Combined змішуються рівнини, гірські масиви, озера та детальний шум, після чого додається вплив річок. У режимах MountainsOnly, PlainsOnly, RiversOnly, LakesOnly і NoiseOnly активується відповідний піднабір шарів.

Після змішування висота проходить через SoftClampHeight, який обмежує результат у межах доступного worldHeight. Це зменшує ризик того, що окремі гірські піки або додатковий шум вийдуть за межі вертикального діапазону, потрібного мешеру, що спричинило б різкі зрізи поверхні, яка виходить за цей діапазон.

3.1.3 Реалізація процедурних форм рельєфу

Клас TerrainNoiseGenerator містить допоміжні математичні функції і процедури формування окремих елементів рельєфу. Функція Fractal01 обчислює суму кількох октав Perlin Noise, Ridged01 перетворює шум у хребтову форму, SmoothMask01 створює плавну маску для змішування різних шумових функцій, а FeaturePoint генерує стабільні точки розташування річок, озер і додаткових гірських масивів.

Функція Fractal01 використовується для утворення базового рельєфу і як основа для інших видів генерації та реалізована наступним чином:

$$F(x, z) = \frac{\sum_{i=0}^{n-1} a_i \cdot N\left(\frac{x + \alpha_s + i\alpha_i}{S}, \frac{z + \beta_s - i\beta_i}{S}\right)}{\sum_{i=0}^{n-1} a_i}, \quad (3.1)$$

де $F(x, z)$ – нормалізована фрактальна сума шуму; N – базова функція Perlin Noise; S – масштаб шуму; a_i – амплітуда октави; n – кількість октав; $\alpha_s, \beta_s, \alpha_i, \beta_i$ – зміщення координат, пов'язані із seed та номером октави. У реалізації

амплітуда кожної наступної октави зменшується вдвічі, а частота збільшується вдвічі, що дозволяє поєднувати великі й дрібні форми.

Функція *Ridged01* використовується як фільтр поверх базового шуму для утворення хребтових структур, таких як гори та передгір'я та має наступний вигляд:

$$R(x, z) = \text{clamp}_{[0,1]}(1 - |2F(x, z) - 1|), \quad (3.2)$$

де $R(x, z)$ – хребтова форма шуму, отримана на основі нормалізованого значення $F(x, z)$. Таке перетворення підсилює значення поблизу середини діапазону шуму та формує вузькі гребенеподібні структури.

Функцію *SmoothMask01*, яка застосовується для утворення згладженої маски для плавного підмішування інших видів структур до базового шуму, реалізовано наступним чином:

$$M(d, r) = t^2(3 - 2t), \quad t = 1 - \text{clamp}_{[0,1]} \left(\frac{d}{r} \right) \quad (3.3)$$

де $M(d, r)$ – маска; d – відстань до центра або лінії впливу; r – радіус. Такі маски застосовуються для річкових долин, озерних улоговин, гірських хребтів і локальних операцій редагування.

Річки представлені полілініями з меандруванням, уздовж яких висота знижується до рівня води, заданого в параметрах генератора, утворюючи таким чином долини та каньйони. Озера формуються через локальні басейни з варіативним береговим контуром, що залежить від шуму.

Окремо обчислюються маски матеріалів поверхні, таких як ґрунт, каміння, пісок, сніг, які враховують висоту, нахил поверхні та шумове розсіювання, що також налаштовується параметрично в інтерфейсі рушія. Це дає змогу фарбувати вершини не лише за абсолютною висотою, а й за

придатністю поверхні до утворення того чи іншого матеріалу. Наприклад, ґрунт не може утворюватися за кута нахилу поверхні вище 45° , внаслідок чого до таких вершин застосовуватиметься камінь; пісок може утворюватися тільки біля рівня води тощо.

3.2 Розробка модулів мешингу та стримінгу

3.2.1 Реалізація Surface Nets

SurfaceNetsMesher реалізовано як статичний клас, що приймає генератор, координати тайла, його розмір, висоту світу та крок дискретизації. У класі визначено CornerOffsets і EdgeCorners, які задають локальну структуру кубічної комірки, а також методи BuildFacesX, BuildFacesY і BuildFacesZ для побудови граней уздовж трьох напрямів координатної системи.

Фрагмент перевірки зміни знака густини реалізовано таким чином:

```
private static bool CrossesSurface(float a, float b)
{
    a = NormalizeDensityForMeshing(a);
    b = NormalizeDensityForMeshing(b);

    return (a > 0f && b < 0f) || (a < 0f && b > 0f);
}
private const float DensityEpsilon = 0.0001f;

private static float NormalizeDensityForMeshing(float d)
{
    if (Mathf.Abs(d) < DensityEpsilon)
        return DensityEpsilon;

    return d;
}
```

Рисунок 3.3 – Функція перевірки знака густини

Усереднення точок перетину дозволяє розмістити вершину всередині активної комірки, а не на фіксованому ребрі. Такий підхід зменшує кількість вершин і формує більш плавну поверхню порівняно з прямим полігональним розбиттям кожної конфігурації комірки.

3.2.2 Реалізація тайлів ландшафту

TerrainTile відповідає за життєвий цикл одного тайла: ініціалізацію, синхронну або асинхронну перебудову, застосування MeshData до MeshFilter, призначення MeshRenderer, створення або видалення MeshCollider і застосування кольорів поверхні. Тайли створюються стримером, але сам процес перетворення MeshData в Unity Mesh інкапсульований саме в TerrainTile. TerrainTile є компонентом сцени, який перетворює результат алгоритму Surface Nets на видимий об'єкт Unity. Він зберігає координати тайла, його розмір, LOD, step і прапорець необхідності створення колайдера.

Асинхронна перебудова використовує Task.Run для виконання обчислювальної частини за межами основного потоку, а готовий MeshData повертається у контекст компонента Unity. Це зменшує затримки під час переміщення камери, хоча фінальне створення й призначення Unity-об'єктів все одно виконуються в середовищі Unity. Після завершення результат застосовується на головному потоці через MeshFilter, MeshRenderer і, за потреби, MeshCollider.

Після створення Mesh поверхня додатково отримує кольори вершин. Колір визначається генератором за висотою, нормаллю, близькістю до води та маскою снігу, тому один матеріал може відображати траву, камінь, пісок і сніг.

3.2.3 Реалізація quadtree-стримінгу

QuadtreeTerrainStreamer керує списком бажаних тайлів, чергою побудови, активними тайлами та наборами queuedSet і buildingSet. Метод BuildDesiredRecursive формує набір тайлів відповідно до видимої області і відстані від спостерігача, ShouldSplit визначає необхідність поділу, TileIntersectsVisibleCircle відкидає дальні тайли, ApplySimpleLodBalancing зменшує різницю деталізації між сусідами. Фрагмент критерію поділу тайла наведено на рисунку 3.4.

```
private bool ShouldSplit(int x, int z, int size, int lod)
{
    if (size ≤ minTileSize)
        return false;

    if (lod ≤ 0)
        return false;

    float distance = DistanceToTileXZ(x, z, size, viewer.position);

    if (distance < lod0Distance)
        return true;

    if (size > minTileSize * 2 && distance < lod1Distance)
        return true;

    if (size > minTileSize * 4 && distance < lod2Distance)
        return true;

    return false;
}
```

Рисунок 3.4 – Критерії поділу тайла

Крок побудови залежить від LOD: тайл LOD 0 має найменший крок, а кожен наступний рівень збільшує його вдвічі. Колайдери створюються лише для тайлів, які перебувають у межах colliderDistance від спостерігача, що зменшує навантаження для віддалених ділянок.

QuadtreeTerrainStreamer підтримує активний набір тайлів навколо спостерігача. Він обчислює бажані тайли, сортує їх за відстанню, запускає асинхронне створення та видаляє фрагменти, які більше не потрібні.

Функція ShouldSplit визначає, чи потрібно ділити вузол дерева, з урахуванням розміру тайла, залишкового LOD і відстані до камери. StepForLod перетворює рівень деталізації на крок семплювання, що безпосередньо впливає на кількість геометрії.

Додатково реалізовано обмеження кількості запусків побудови за кадр і кількості одночасних задач. Це запобігає різким навантаженням під час запуску сцени або переміщення користувача та дозволяє поступово оновлювати світ.

3.3 Реалізація редагування ландшафту

VoxelTerrainEditor реалізує взаємодію користувача з полем густини. Компонент отримує напрям променя з камери, виконує raymarching до першої зміни знака густини, уточнює точку перетину двійковим пошуком і, залежно від введення, створює операцію додавання, вилучення або згладжування матеріалу.

Після запису BrushOperation редактор викликає RebuildTilesInBounds у QuadtreeTerrainStreamer. Це дозволяє перебудовувати лише ті тайли, що перетинаються з областю впливу кисті, замість повної перебудови світу. Фрагмент коду визначення області, необхідної для перебудови:

```
private void RebuildBrushArea()
{
    if (streamer == null)
        return;

    float r = brushRadius + rebuildPadding;

    streamer.RebuildTilesInBounds(
        currentHitPoint.x - r,
        currentHitPoint.z - r,
        currentHitPoint.x + r,
        currentHitPoint.z + r
    );
}
```

Рисунок 3.5 – Визначення області тайлів, необхідної для перебудови

BrushPreview візуалізує радіус за допомогою сфери, утвореної трьома кільцями, а силу — кольоровим градієнтом від білого до червоного, що допомагає контролювати область редагування.

VoxelTerrainEditor поєднує обробку введення, пошук точки на поверхні та запуск перебудови тайлів. Він використовує нову систему Input System для читання клавіатури й миші, тому операції кисті виконуються незалежно від застарілого механізму введення Unity.

Додавання матеріалу виконується натисканням «ЛКМ», вилучення — поєднанням «Alt» і «ЛКМ», а згладжування — поєднанням «Q» і «ЛКМ». Радіус і сила кисті змінюються за допомогою клавіш «[» і «]» або колеса миші.

Після кожної операції редактор визначає прямокутну область XZ, яка охоплює радіус кисті з додатковим відступом, і передає її стримеру. Це дозволяє перебудувати лише зачеплені тайли.

3.4 Реалізація візуалізації води та матеріалів

Візуалізація води реалізована через WorldWaterPlane, який створює прямокутну площину розміром із світ і розміщує її на висоті waterLevel. Матеріал води може бути заданий у налаштуваннях. Колір і прозорість беруться з WorldGenSettings, тому візуальний стан води узгоджений з параметрами генерації річок, озер і берегових зон. Для написання шейдерів води та ландшафту використовувалася мова HLSL.

Для ландшафту використовується вершинне фарбування. Фрагмент коду, який відповідає за кольорові варіації вершин:

```
half4 frag(Varyings input) : SV_Target
{
    float3 normalWS = normalize(input.normalWS);

    Light mainLight = GetMainLight();

    float ndotl = saturate(dot(normalWS, mainLight.direction));
    float3 ambient = SampleSH(normalWS);

    float largeNoise = SmoothNoise(input.positionWS.xz * _VariationScale);
    float fineNoise = SmoothNoise(input.positionWS.xz * _FineVariationScale + 31.7);

    float mixed = (largeNoise - 0.5) * 0.75 + (fineNoise - 0.5) * 0.25;

    float3 baseColor = input.color.rgb;
    float3 warmer = baseColor * float3(1.05, 0.98, 0.88);
    float3 cooler = baseColor * float3(0.86, 0.95, 1.02);
    float3 darker = baseColor * 0.72;

    float warmMask = saturate(0.5 + mixed * 2.0);
    float darkMask = saturate(abs(mixed) * 1.35) * _VariationStrength;

    float3 varied = lerp(cooler, warmer, warmMask);
    baseColor = lerp(baseColor, varied, _VariationStrength);
    baseColor = lerp(baseColor, darker, darkMask * 0.55);

    float slopeFactor = saturate(normalWS.y);
    float slopeDarken = lerp(1.0 - _SlopeDarken, 1.0, slopeFactor);
    baseColor *= slopeDarken;

    baseColor = saturate(baseColor);

    float3 lighting = ambient + mainLight.color * ndotl;
    float3 finalColor = baseColor * lighting * _Brightness;

    return half4(finalColor, 1.0);
}
```

Рисунок 3.6 – Варіації кольору вершин

Колір вершини обчислюється за висотою, нахилом нормалі, наближенням до рівня води та сніговою маскою, що дозволяє отримувати різні поверхневі зони без окремих текстурних карт.

3.5 Реалізація керування камерою

FlyCameraController забезпечує вільне переміщення спостерігача у просторі. Користувач може пересуватися по всіх трьох координатних осях, прискорювати або сповільнювати рух та керувати напрямом погляду. Такий режим камери зручний для огляду ландшафтів, перевірки переходів LOD і тестування редактора. Фрагмент коду, що відповідає за рух камери:

```
private void HandleMovement()
{
    Vector3 input = Vector3.zero;

    if (Keyboard.wKey.isPressed) input += Vector3.forward;
    if (Keyboard.sKey.isPressed) input += Vector3.back;
    if (Keyboard.dKey.isPressed) input += Vector3.right;
    if (Keyboard.aKey.isPressed) input += Vector3.left;

    if (IsKeyPressed(upKey)) input += Vector3.up;
    if (IsKeyPressed(downKey)) input += Vector3.down;

    float speed = moveSpeed;

    Vector3 move =
        transform.forward * input.z +
        transform.right * input.x +
        Vector3.up * input.y;

    if (move.sqrMagnitude > 1f)
        move.Normalize();

    transform.position += move * speed * Time.deltaTime;
}
```

Рисунок 3.7 – Контролер руху камери

Камера також виконує роль джерела позиції для quadtree-стрімера і редактора. Від її координат залежить вибір тайлів, які мають бути побудовані з високою деталізацією, а її напрям використовується для пошуку точки редагування поверхні.

4 ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Методика тестування

Тестування розробленої системи виконувалося за функціональними сценаріями, що перевіряють генерацію різних режимів, коректність побудови Surface Nets, створення тайлів, роботу балансування та стикування LOD, редагування поверхні, перебудову тайлів після кисті, коректність колайдерів, відображення води й снігу та керування камерою. Тестування ПЗ має бути спрямоване на виявлення помилок, а не лише на підтвердження очікуваної поведінки [25].

Для кожної групи функцій було визначено вхідні умови, очікуваний результат, фактичне спостереження та статус. Такий підхід дозволяє фіксувати працездатність основних модулів і виявляти напрями подальшого покращення без тверджень про повну відсутність дефектів.

Тестування має практичний характер: основна увага приділяється стабільності роботи, відсутності критичних розривів, коректній перебудові тайлів і передбачуваній реакції системи на дії користувача.

4.2 Тестування програмних модулів системи

4.2.1 Тестування режимів генерації

Перевірка режимів генерації виконувалася шляхом перемикання `generationMode` у `WorldGenSettings` і перебудови світу. Очікувані результати визначалися логікою `TerrainShapeMixer`: комбінований режим має поєднувати основні шари, `MountainsOnly` — акцентувати гори, `PlainsOnly` — хвилясті рівнини, `RiversOnly` — річкові долини, `LakesOnly` — озерні улоговини, `NoiseOnly` — шумову деталізацію. Тестування проводилося з використанням однакового зерна для отримання повторюваних результатів. Розмір області для тестування був обмежений 4096×4096 вокселів по площині XZ, вертикальний діапазон Y був заданий у проміжку 0-768 вокселів. Повний набір налаштувань стримера та генератора, заданих для тестування

продемонстровано на рисунку А.1. Результати перевірки режимів генерації наведено у таблиці 4.1.

Таблиця 4.1 – Результати перевірки режимів генерації

Режим	Очікуваний результат	Отриманий результат
Combined	Поєднання гір, рівнин, річок, озер і шуму	Формується комплексний ландшафт із різними зонами
MountainsOnly	Домінування гірського масиву	Гірські форми виражені сильніше за інші шари
PlainsOnly	Переважаання хвилястих рівнин	Рельєф має помірні перепади висоти
RiversOnly	Річкова долина з руслом	Висота знижується вздовж річкового шляху
LakesOnly	Озерні улоговини	Формуються локальні зниження під водойми
NoiseOnly	Шумова поверхня без складних форм	Відображаються детальні шумові нерівності

Тестування режимів генерації було розпочато з перевірки режиму генерації PlainOnly:

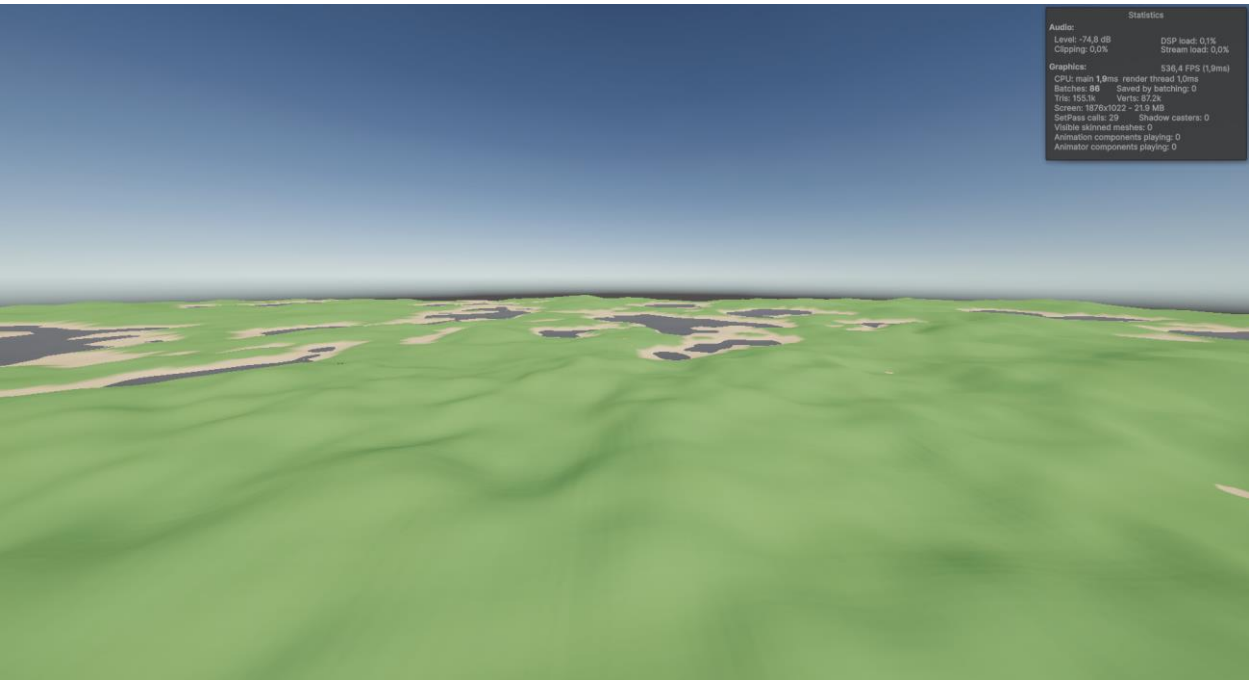


Рисунок 4.1 – Результат генерації у режимі PlainsOnly

У цьому режимі генерації добре видно окремі шари. Основу поверхні задано шумовою функцією Perlin Noise у 3 різних масштабах, щоб отримати

пагорби різних розмірів. Поверх накладаються регіональні підвищення та заниження рель'єфу. Для наочності цього фільтра різниця між рівнем води та базовим підйомом світу була обрана невелика, внаслідок чого утворилися «озера» — місця, де ландшафт уходить нижче рівня води.

Наступним протестованим режимом був Lakes Only:

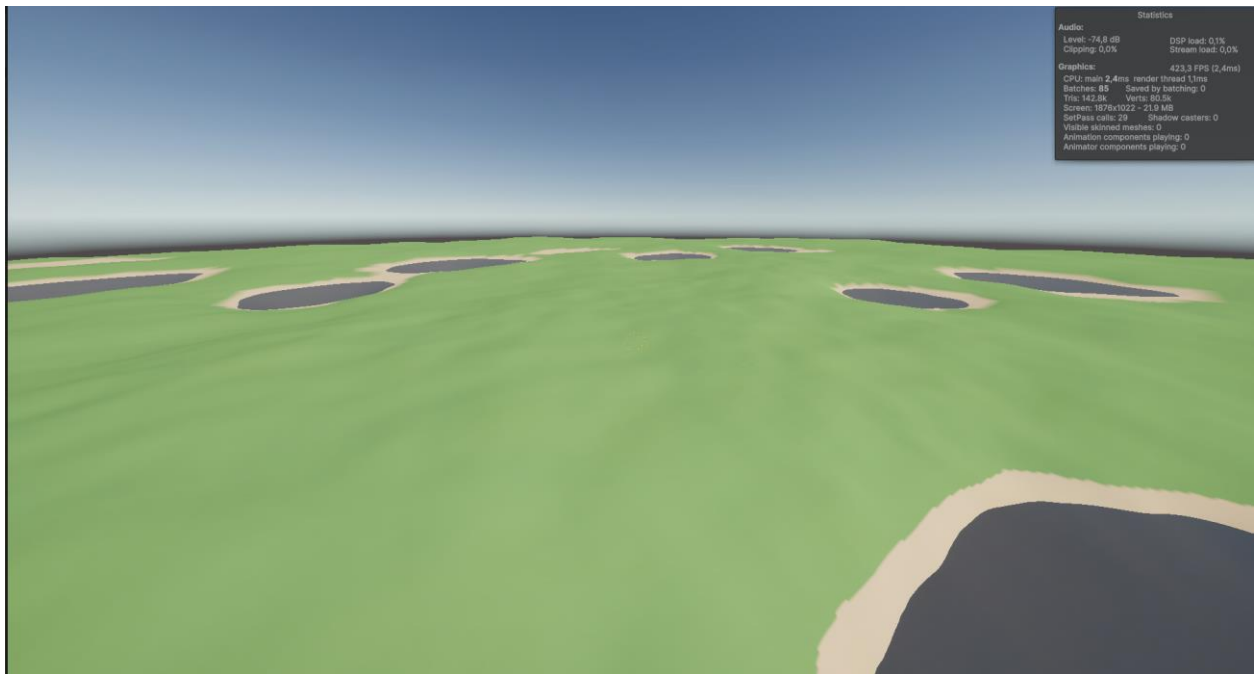


Рисунок 4.2 – Результат генерації у режимі LakesOnly

Даний режим надає майже пласку поверхню з локальними улоговинами правильної круглої форми, причому незалежно від базового підйому світу, шумова функція озер завжди буде опускати ці регіони нижче рівня води. Оскільки обране зерно генерації не давало озер в обраній області для перегляду, для тестування цього режиму генерації було збільшено їхню кількість у налаштуваннях генератора. У режимі Combined це значення було повернуто до попереднього. Такий результат підтверджує, що модуль озер може використовуватися як самостійний елемент генерації та як частина комбінованого режиму, де він доповнює інші форми ландшафту.

Далі було протестовано режим RiversOnly:

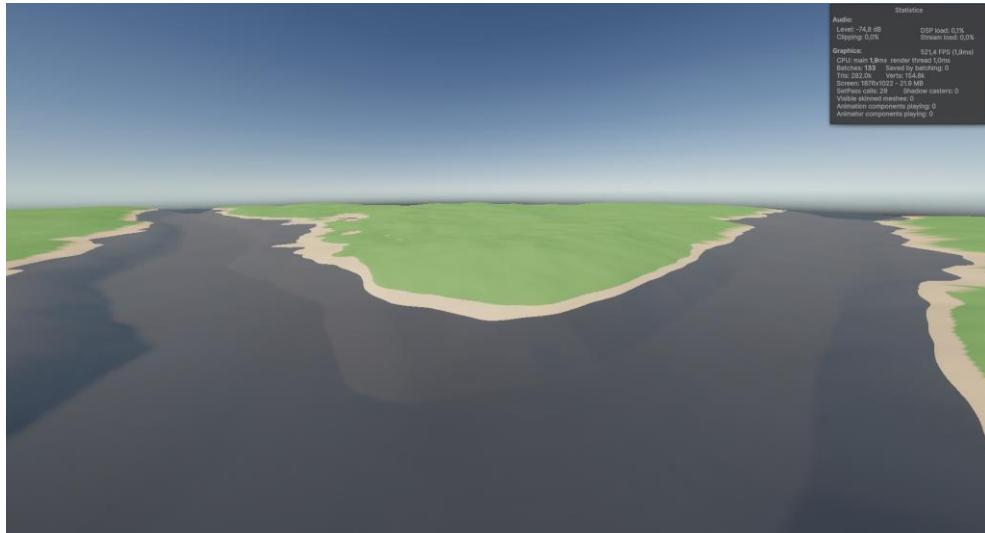


Рисунок 4.3 – Результат генерації у режимі RiversOnly

На зображенні видно, як працює функція генерації річкових долин. Унаслідок обраних базової висоти світа та висоти рівня води дно стає плоским, оскільки досягає координати $Y = 0$. Це також дозволяє перевірити коректність роботи мешера на абсолютно плоских поверхнях, коли значення густини може дорівнювати 0.

Останнім з ізольованих режимів генерації було протестовано MountainsOnly:

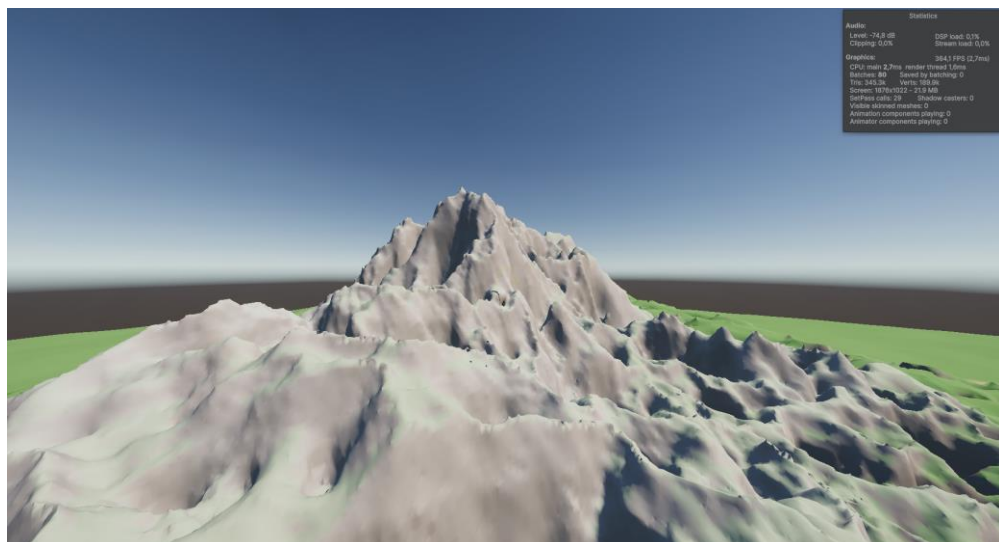


Рисунок 4.4 – Результат генерації у режимі MountainsOnly

У цьому режимі (див. рисунок 4.4) видно, що гора має хребтову структуру з підвищенням до центру хребта. Висоту можна оцінити за утворенням снігу. Самі ж налаштування шумових функцій у даному режимі зроблені таким чином, щоб утворювати гілкуваті скелі з різким підйомом, а до вершин застосовувався матеріал каменю.

На останок було протестовано режим Combined, де всі попередні режими змішуються у заданих пропорціях. За це відповідає параметр Weight у налаштуваннях кожного окремого режиму (див. рис. А.1):

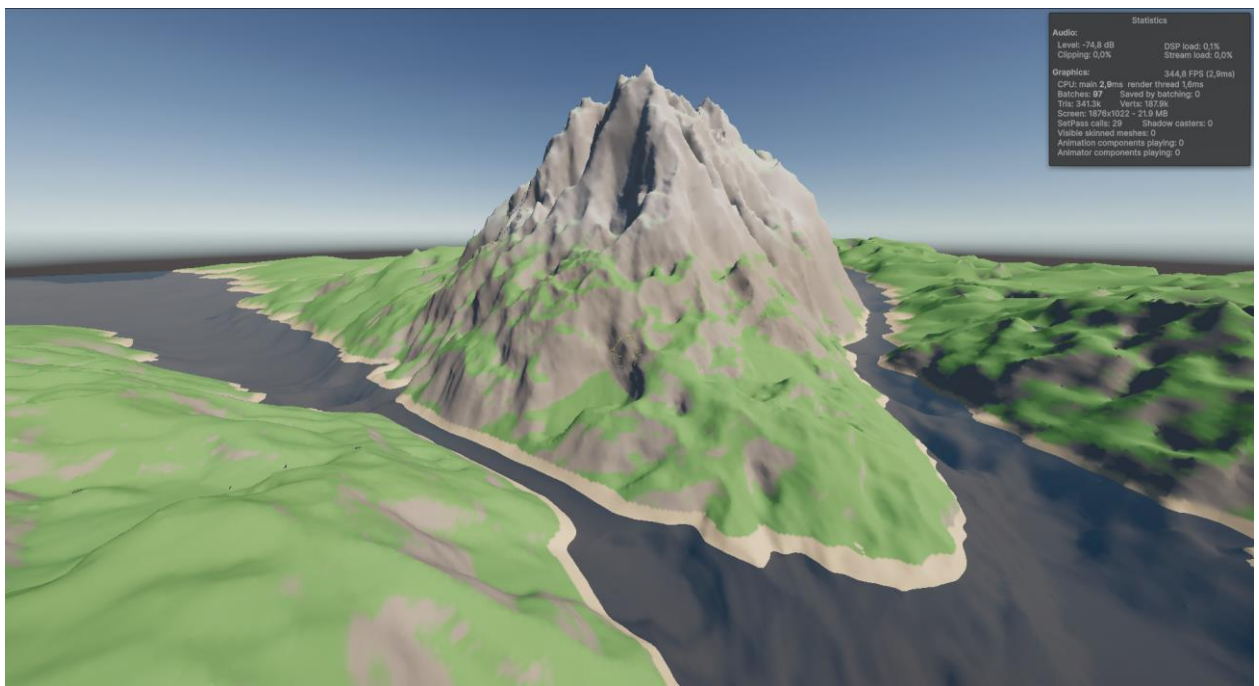


Рисунок 4.5 – Результат генерації у режимі Combined

Як можна бачити, в обраній області тестування утворилися усі очікувані структури.

4.2.2 Тестування quadtree-стрімінгу

Під час тестування quadtree-стрімінгу перевірялося, що тайли поблизу камери будуються з більшою деталізацією, дальні тайли використовують більший step, набір активних тайлів змінюється під час руху, неактуальні тайли видаляються після завершення черги побудови, а колайдери створюються тільки в межах заданої відстані.

Результати перевірки роботи quadtree-стрімінгу наведено у таблиці 4.2.

Таблиця 4.2 – Перевірка роботи quadtree-стримінгу

Перевірка	Очікуваний результат	Отриманий результат	Статус
LOD біля камери	Малі тайли з малим step	Біля камери створюються детальніші тайли	Виконано
Дальній LOD	Більший step для віддалених тайлів	Дальні тайли мають нижчу деталізацію	Виконано
Оновлення при русі	Зміна desired tiles	Набір тайлів оновлюється після переміщення	Виконано
Очищення	Видалення неактуальних тайлів	Старі тайли видаляються після завершення побудови	Виконано
Колайдери	Тільки поблизу спостерігача	Колайдери створюються в межах colliderDistance	Виконано

Перевірка демонструє, що стример реагує на переміщення камери, змінюючи активний набір тайлів і рівень деталізації. Окреме створення колайдерів поблизу спостерігача зменшує зайве навантаження та залишає фізичну взаємодію лише там, де вона потрібна.

Під час тестування quadtree-стримінгу камера переміщувалася між різними ділянками світу, а активний набір тайлів перевірявся за візуальним результатом і вікном відладки. Очікувалося, що ближні тайли отримують більшу деталізацію, а дальні залишаються грубішими. Додатково перевірялося, чи не виникають розриви між тайлами різного рівня деталізації під час оновлення сцени.

Перевірялася також робота черги побудови: нові тайли мають з’являтися поступово, без блокування сцени, а неактуальні фрагменти — видалятися після завершення актуальних задач. Це важливо для уникнення помітних провалів під час руху. Такий підхід дозволяє зберігати вже побудовану геометрію до моменту, коли новий набір тайлів буде готовий до відображення.

Окремий критерій стосувався колайдерів. Вони повинні створюватися лише в зоні поблизу спостерігача, оскільки колізії на віддалених тайлах не потрібні для поточної взаємодії й лише збільшували б навантаження.

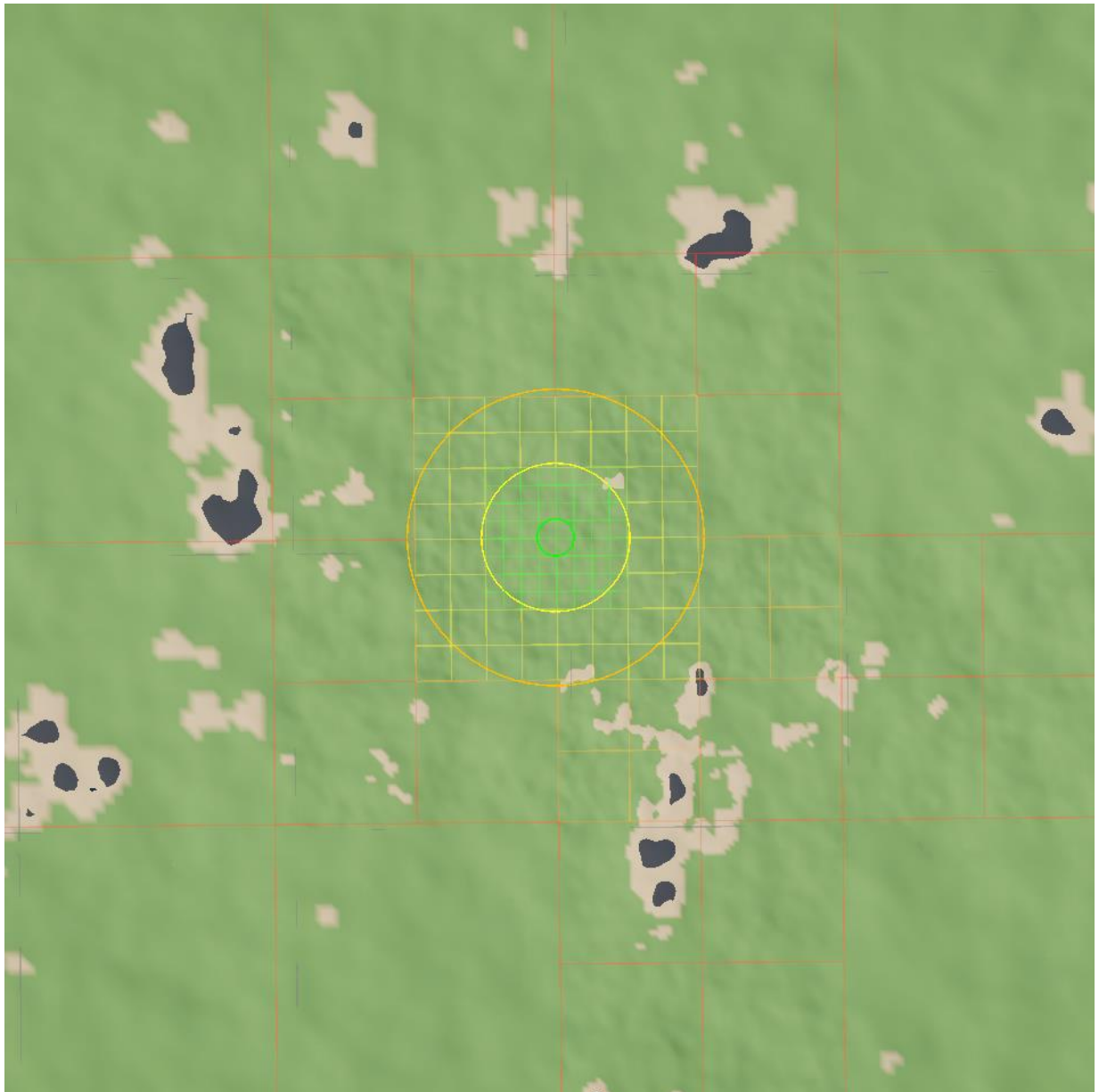


Рисунок 4.6 – Поділ світу на тайли з різним рівнем LOD

Кільцями на рисунку показано радіуси відстаней до спостерігача, для яких мають застосовуватись різні рівні LOD. Сітка показує кордони тайлів, завдяки чому наочно видно роботу quadtree-поділу. Колайдери створюються лише в області внутрішнього кола, де значення *step* є найменшим.

4.2.3 Тестування редагування ландшафту

Редагування перевірялося в трьох режимах: додавання матеріалу, вилучення матеріалу та згладжування. Також перевірялися зміна радіуса, зміна сили кисті, наявність візуального предпросмотру та перебудова тайлів, що перетинають область впливу.

Результати перевірки інструментів редагування наведено у таблиці 4.3.

Таблиця 4.3 – Перевірка інструментів редагування ландшафту

Операція	Очікуваний результат	Отриманий результат	Статус
Add	Локальне додавання матеріалу	Поверхня піднімається в області кисті	Виконано
Subtract	Локальне вилучення матеріалу	Формується заглиблення або виріз	Виконано
Smooth	Згладжування внеску кисті	Зменшується різкість локальної зміни	Виконано
Зміна радіуса	Зміна зони впливу	BrushPreview і rebuild bounds змінюються	Виконано
Зміна сили	Зміна інтенсивності дії	Вплив операції стає сильнішим або слабшим	Виконано
Перебудова	Оновлення затронутих тайлів	Викликається RebuildTilesInBounds	Виконано

Наведені результати показують, що всі базові режими кисті виконують різні дії над полем густини. Зміна радіуса, сили та подальша перебудова тайлів підтверджують, що редактор впливає не на тимчасове відображення меша, а на джерельні дані поверхні.

Редагування перевірялося послідовним застосуванням трьох типів кисті: додавання, вилучення та згладжування. Для кожного режиму оцінювалися наявність зміни форми, стабільність позиції попадання та перебудова саме тих тайлів, що перетинають область впливу.

Також перевірялися зміна радіуса й сили кисті. Збільшення радіуса має розширювати область впливу, а зміна сили — впливати на інтенсивність додавання або вилучення матеріалу, без зміни логіки пошуку поверхні.

BrushPreview використовувався як допоміжний індикатор для контролю положення та розміру кисті. Його відображення дозволяє до застосування операції оцінити, яка частина ландшафту буде змінена. Результат тестування інструментів редагування ландшафту показано на рисунку 4.7

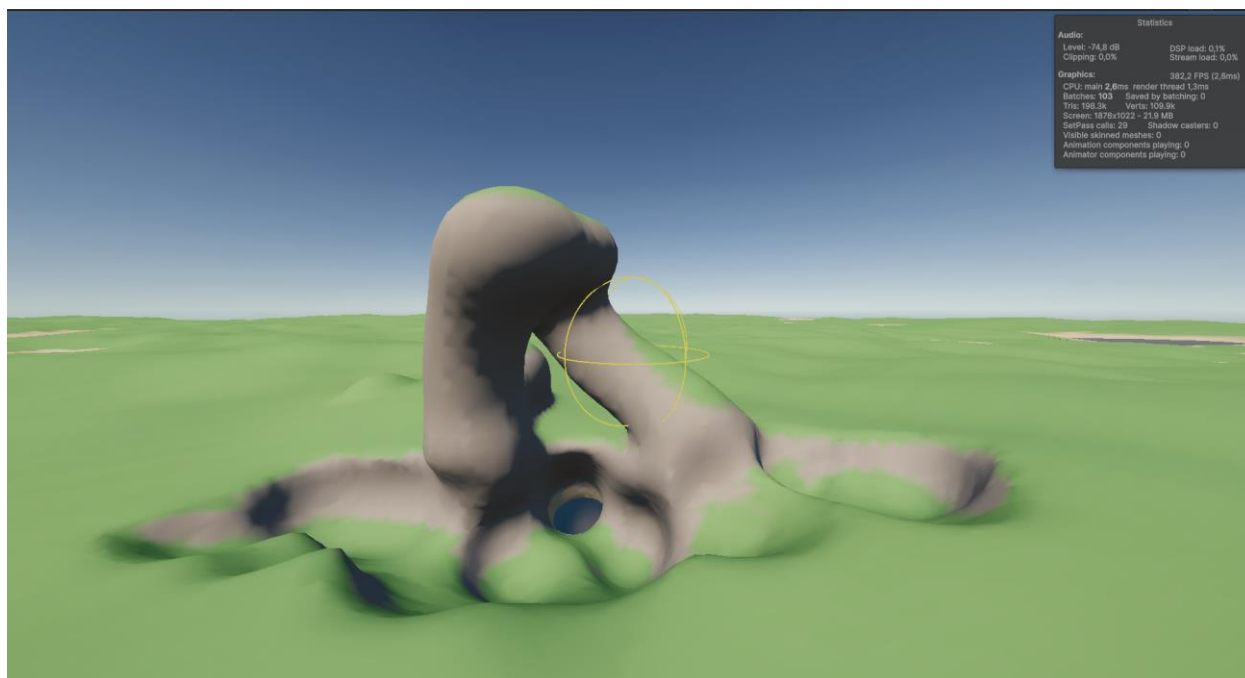


Рисунок 4.7 – Тестування інструментів редагування ландшафту

На рисунку показано роботу операцій додавання, віднімання та згладжування поверхні.

4.3 Аналіз отриманих результатів

У результаті реалізації отримано програмну систему для процедурної генерації, відображення та локального редагування воксельного ландшафту. Модульна структура дозволила розділити налаштування, формування SDF-поля, побудову поверхні, стрімінг, редагування та візуалізацію.

Surface Nets дозволив отримати згладжену поверхню з поля густини, а quadtree-організація дала змогу керувати рівнем деталізації залежно від відстані до спостерігача. Редактор кисті забезпечив локальну зміну поля густини. Вершинне забарвлення дозволило отримати базову диференціацію матеріалів поверхні без окремих текстурних карт для кожного тайла.

Розробка може бути розвинена у напрямі оптимізації обчислень, покращення LOD-переходів, збереження змін ландшафту, розширення біомів, додавання печер і нависаючих форм через повноцінні 3D SDF-шари, складнішої гідрології та перенесення ресурсоемних операцій у Jobs або Burst.

ВИСНОВКИ

На основі аналізу методів процедурної генерації, представлення тривимірних поверхонь і просторової організації великих сцен обґрунтовано вибір поля густини, алгоритму Surface Nets і quadtree-структури для побудови редагованого воксельного ландшафту.

Розроблено архітектуру програмного забезпечення, що включає модулі налаштувань генерації, генератора висоти і густини, змішування форм рельєфу, побудови меша, тайлового стримінгу, редагування, візуалізації води й матеріалів та керування камерою.

Реалізовано програмні модулі для налаштування генерації, генератора поля SDF, генератора шумових карт, змішувача шумових карт, мешера ізоперхні за алгоритмом Surface Nets, quadtree-стрімера, шейдери для ландшафту та води, модулі редагування ландшафту та контролера користувача.

Виконано тестування режимів генерації, Surface Nets мешингу, quadtree-стрімінгу, балансування та стикування LOD, локального редагування, перебудови тайлів, колайдерів, відображення води та матеріалів, роботи камери.

Розробка дозволяє створювати, відображати та локально редагувати процедурний воксельний ландшафт у Unity, а також слугує основою для подальшого розвитку інструментів для процедурного світу в іграх, симуляціях, візуальних прототипах і дослідницьких системах.

ПЕРЕЛІК ПОСИЛАНЬ

1. Shaker N. Procedural Content Generation in Games: A Textbook and an Overview of Current Research / N. Shaker, J. Togelius, M. J. Nelson. – Cham : Springer, 2016. – 237 с. – DOI: 10.1007/978-3-319-42716-4.
2. Texturing and Modeling: A Procedural Approach / D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, S. Worley. – 3-тє вид. – San Francisco : Morgan Kaufmann, 2002. – 712 с.
3. Perlin K. An Image Synthesizer / K. Perlin // Computer Graphics. – 1985. – Т. 19, № 3. – С. 287–296. – DOI: 10.1145/325165.325247.
4. Perlin K. Improving Noise / K. Perlin // ACM Transactions on Graphics. – 2002. – Т. 21, № 3. – С. 681–682. – DOI: 10.1145/566570.566636.
5. Lagae A. A Survey of Procedural Noise Functions / A. Lagae, S. Lefebvre, G. Drettakis, P. Dutré // Computer Graphics Forum. – 2010. – Т. 29, № 8. – С. 2579–2600. – DOI: 10.1111/j.1467-8659.2010.01827.x.
6. Worley S. A Cellular Texture Basis Function / S. Worley // Proceedings of SIGGRAPH 1996. – New Orleans : ACM, 1996. – С. 291–294. – DOI: 10.1145/237170.237267.
7. Frisken S. F. Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics / S. F. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones // Proceedings of SIGGRAPH 2000. – New Orleans : ACM, 2000. – С. 249–254. – DOI: 10.1145/344779.344899.
8. Lorensen W. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm / W. E. Lorensen, H. E. Cline // Computer Graphics. – 1987. – Т. 21, № 4. – С. 163–169. – DOI: 10.1145/37402.37422.
9. Gibson S. F. F. Constrained Elastic Surface Nets: Generating Smooth Surfaces from Binary Segmented Data / S. F. F. Gibson // Medical Image Computing and Computer-Assisted Intervention – MICCAI'98. – Berlin : Springer, 1998. – С. 888–898. – DOI: 10.1007/BFb0056307.

10. Ju T. Dual Contouring of Hermite Data / T. Ju, F. Losasso, S. Schaefer, J. Warren // ACM Transactions on Graphics. – 2002. – Т. 21, № 3. – С. 339–346. – DOI: 10.1145/566570.566586.
11. Samet H. Foundations of Multidimensional and Metric Data Structures / H. Samet. – San Francisco : Morgan Kaufmann, 2006. – 1024 с.
12. Ulrich T. Rendering Massive Terrains Using Chunked Level of Detail Control / T. Ulrich // SIGGRAPH 2002 Course Notes. – San Antonio : ACM, 2002. – С. 1–12.
13. Losasso F. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids / F. Losasso, H. Hoppe // ACM Transactions on Graphics. – 2004. – Т. 23, № 3. – С. 769–776. – DOI: 10.1145/1015706.1015799.
14. Pajarola R. Survey of Semi-Regular Multiresolution Models for Interactive Terrain Rendering / R. Pajarola, E. Gobbetti // The Visual Computer. – 2007. – Т. 23. – С. 583–605. – DOI: 10.1007/s00371-007-0163-2.
15. Akenine-Möller T. Real-Time Rendering / T. Akenine-Möller, E. Haines, N. Hoffman. – 4-те вид. – Boca Raton : CRC Press, 2018. – 1198 с.
16. Gregory J. Game Engine Architecture / J. Gregory. – 3-тє вид. – Boca Raton : CRC Press, 2018. – 1240 с.
17. Unity 6.3 User Manual [Электронний ресурс] / Unity Technologies. – Режим доступу : <https://docs.unity3d.com/6000.3/Documentation/Manual/>.
18. C# Guide [Электронний ресурс] / Microsoft. – Режим доступу : <https://learn.microsoft.com/en-us/dotnet/csharp/>.
19. Mesh Scripting API [Электронний ресурс] / Unity Technologies. – Режим доступу : <https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Mesh.html>.
20. ScriptableObject Scripting API [Электронний ресурс] / Unity Technologies. – Режим доступу : <https://docs.unity3d.com/6000.3/Documentation/ScriptReference/ScriptableObject.html>.

21. Mesh Collider component reference [Электронный ресурс] / Unity Technologies. – Режим доступа : <https://docs.unity3d.com/6000.3/Documentation/Manual/class-MeshCollider.html>.

22. Input in Unity [Электронный ресурс] / Unity Technologies. – Режим доступа : <https://docs.unity3d.com/6000.3/Documentation/Manual/Input.html>.

23. Universal Render Pipeline documentation [Электронный ресурс] / Unity Technologies. – Режим доступа : <https://docs.unity3d.com/6000.6/Documentation/Manual/urp/urp-introduction.html>.

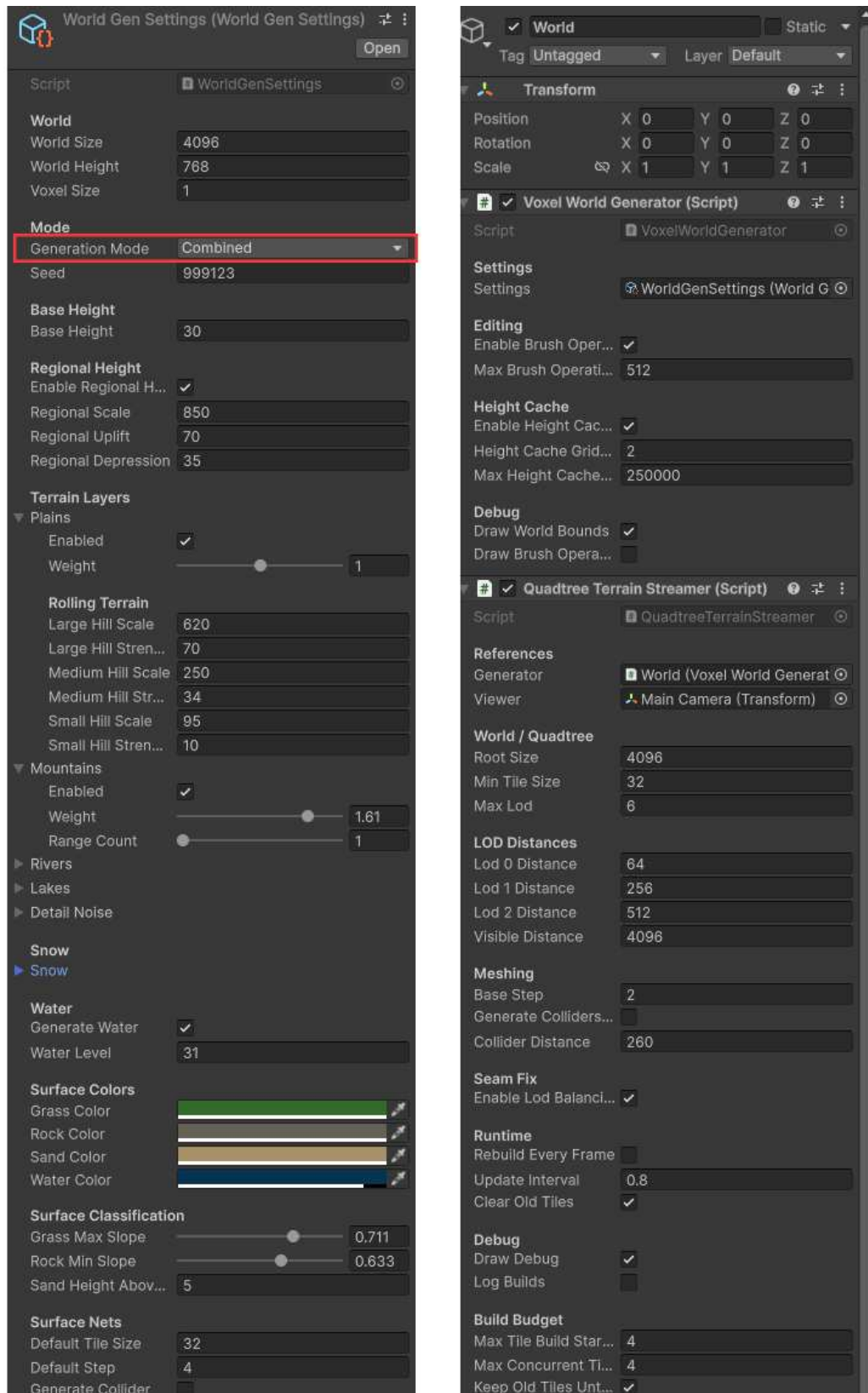
24. Sommerville I. Software Engineering / I. Sommerville. – 10-е вид. – Boston : Pearson, 2016. – 816 с.

25. Myers G. J. The Art of Software Testing / G. J. Myers, C. Sandler, T. Badgett. – 3-е вид. – Hoboken : Wiley, 2011. – 240 с.

ДОДАТКИ

Додаток А

Рисунок А.1 – Налаштування стримера та генератора



Додаток Б

Код програмного модуля налаштувань генерації

Скрипт WorldGenSettings.cs

```

using System;
using UnityEngine;

namespace VoxelWorldGenerator.Generation
{
    public enum TerrainGenerationMode
    {
        Combined,
        MountainsOnly,
        PlainsOnly,
        RiversOnly,
        LakesOnly,
        NoiseOnly
    }

    public enum BrushOperationMode
    {
        Add,
        Subtract,
        Smooth
    }

    [Serializable]
    public class MountainSettings
    {
        public bool enabled = true;

        [Range(0f, 2f)]
        public float weight = 1f;

        [Range(1, 4)]
        public int rangeCount = 1;
    }

    [Serializable]
    public class PlainsSettings
    {
        public bool enabled = true;

        [Range(0f, 2f)]
        public float weight = 1f;

        [Header("Rolling Terrain")]
        public float largeHillScale = 620f;
        public float largeHillStrength = 70f;

        public float mediumHillScale = 250f;
        public float mediumHillStrength = 34f;

        public float smallHillScale = 95f;
        public float smallHillStrength = 10f;
    }

    [Serializable]
    public class RiverSettings
    {
        public bool enabled = true;

        [Range(0f, 2f)]
        public float weight = 1f;
    }
}

```



```

[Header("Continuous Rivers")]
public int riverCount = 2;
public int segmentsPerRiver = 16;
public float riverLength = 3200f;

[Header("Valley / Canyon")]
public float valleyWidth = 300f;
public float canyonWidth = 135f;
public float coreWidth = 36f;

public float valleyDepth = 26f;
public float canyonDepth = 46f;
public float riverBedBelowWater = 3f;
public float valleyShoulderHeight = 18f;

[Header("Meander")]
public float meanderScalePrimary = 700f;
public float meanderAmplitudePrimary = 190f;

public float meanderScaleSecondary = 240f;
public float meanderAmplitudeSecondary = 60f;
}

[Serializable]
public class LakeSettings
{
    public bool enabled = true;

    [Range(0f, 2f)]
    public float weight = 0.55f;

    public int lakeCount = 2;
    public float radius = 140f;
    public float basinRadiusMultiplier = 1.8f;
    public float basinDepth = 16f;
    public float lakeDepth = 22f;

    public float shoreNoiseScale = 120f;

    [Range(0f, 0.75f)]
    public float shoreNoiseStrength = 0.22f;
}

[Serializable]
public class DetailNoiseSettings
{
    public bool enabled = true;

    [Range(0f, 2f)]
    public float weight = 0.7f;

    public float broadScale = 620f;
    public float broadStrength = 28f;

    public float mediumScale = 210f;
    public float mediumStrength = 14f;

    public float smallScale = 70f;
    public float smallStrength = 4f;
}

[Serializable]
public class SnowSettings

```

```

{
    public bool enabled = true;

    public float snowHeightStart = 260f;
    public float snowHeightFade = 90f;

    [Range(0f, 90f)]
    public float maxSnowSlopeAngle = 38f;

    public float snowNoiseScale = 160f;

    [Range(0f, 1f)]
    public float snowNoiseStrength = 0.25f;

    public Color snowColor = new Color(0.88f, 0.92f, 0.96f, 1f);
}

[CreateAssetMenu(menuName = "Voxel Diploma/World Gen Settings")]
public class WorldGenSettings : ScriptableObject
{
    [Header("World")]
    public int worldSize = 2048;
    public int worldHeight = 768;
    public float voxelSize = 1f;

    [Header("Mode")]
    public TerrainGenerationMode generationMode = TerrainGenerationMode.Combined;
    public int seed = 12345;

    [Header("Base Height")]
    public float baseHeight = 110f;

    [Header("Regional Height")]
    public bool enableRegionalHeight = true;
    public float regionalScale = 850f;
    public float regionalUplift = 80f;
    public float regionalDepression = 42f;

    [Header("Terrain Layers")]
    public PlainsSettings plains = new PlainsSettings();
    public MountainSettings mountains = new MountainSettings();
    public RiverSettings rivers = new RiverSettings();
    public LakeSettings lakes = new LakeSettings();
    public DetailNoiseSettings detailNoise = new DetailNoiseSettings();

    [Header("Snow")]
    public SnowSettings snow = new SnowSettings();

    [Header("Water")]
    public bool generateWater = true;
    public float waterLevel = 42f;

    [Header("Surface Colors")]
    public Color grassColor = new Color(0.20f, 0.42f, 0.16f, 1f);
    public Color rockColor = new Color(0.40f, 0.38f, 0.34f, 1f);
    public Color sandColor = new Color(0.66f, 0.57f, 0.40f, 1f);
    public Color waterColor = new Color(0.02f, 0.16f, 0.34f, 0.88f);

    [Header("Surface Classification")]
    [Range(0f, 1f)]
    public float grassMaxSlope = 0.48f;

    [Range(0f, 1f)]
    public float rockMinSlope = 0.22f;
}

```

```

public float sandHeightAboveWater = 5f;

[Header("Surface Nets")]
public int defaultTileSize = 32;
public int defaultStep = 4;
public bool generateCollider = false;

[Header("Optional Materials")]
public Material terrainMaterial;
public Material waterMaterial;

private void OnValidate()
{
    worldSize = Mathf.Max(128, worldSize);
    worldHeight = Mathf.Max(32, worldHeight);
    voxelSize = Mathf.Max(0.01f, voxelSize);

    baseHeight = Mathf.Clamp(baseHeight, 1f, worldHeight - 2f);

    regionalScale = Mathf.Max(1f, regionalScale);
    regionalUplift = Mathf.Max(0f, regionalUplift);
    regionalDepression = Mathf.Max(0f, regionalDepression);

    waterLevel = Mathf.Clamp(waterLevel, 0f, worldHeight - 1f);
    sandHeightAboveWater = Mathf.Max(0f, sandHeightAboveWater);

    defaultTileSize = Mathf.Max(16, defaultTileSize);
    defaultStep = Mathf.Max(1, defaultStep);

    plains ??= new PlainsSettings();
    mountains ??= new MountainSettings();
    rivers ??= new RiverSettings();
    lakes ??= new LakeSettings();
    detailNoise ??= new DetailNoiseSettings();

    mountains.rangeCount = Mathf.Clamp(mountains.rangeCount, 1, 4);

    plains.largeHillScale = Mathf.Max(1f, plains.largeHillScale);
    plains.mediumHillScale = Mathf.Max(1f, plains.mediumHillScale);
    plains.smallHillScale = Mathf.Max(1f, plains.smallHillScale);

    rivers.riverCount = Mathf.Max(0, rivers.riverCount);
    rivers.segmentsPerRiver = Mathf.Max(4, rivers.segmentsPerRiver);
    rivers.riverLength = Mathf.Max(1f, rivers.riverLength);
    rivers.valleyWidth = Mathf.Max(1f, rivers.valleyWidth);
    rivers.canyonWidth = Mathf.Max(1f, rivers.canyonWidth);
    rivers.coreWidth = Mathf.Max(1f, rivers.coreWidth);
    rivers.meanderScalePrimary = Mathf.Max(1f, rivers.meanderScalePrimary);
    rivers.meanderScaleSecondary = Mathf.Max(1f, rivers.meanderScaleSecondary);

    lakes.lakeCount = Mathf.Max(0, lakes.lakeCount);
    lakes.radius = Mathf.Max(1f, lakes.radius);
    lakes.basinRadiusMultiplier = Mathf.Max(1f, lakes.basinRadiusMultiplier);
    lakes.shoreNoiseScale = Mathf.Max(1f, lakes.shoreNoiseScale);

    detailNoise.broadScale = Mathf.Max(1f, detailNoise.broadScale);
    detailNoise.mediumScale = Mathf.Max(1f, detailNoise.mediumScale);
    detailNoise.smallScale = Mathf.Max(1f, detailNoise.smallScale);

    snow.snowHeightFade = Mathf.Max(0.01f, snow.snowHeightFade);
    snow.snowNoiseScale = Mathf.Max(1f, snow.snowNoiseScale);
}
}
}

```

Додаток В

Код програмного модуля генерації густини

Скрипт VoxelWorldGenerator.cs

```

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

namespace VoxelWorldGenerator.Generation
{
    [Serializable]
    public struct BrushOperation
    {
        public Vector3 position;
        public float radius;
        public float strength;
        public bool addTerrain;
        public BrushOperationMode mode;

        public BrushOperation(Vector3 position, float radius, float strength, bool
addTerrain)
        {
            this.position = position;
            this.radius = radius;
            this.strength = strength;
            this.addTerrain = addTerrain;
            this.mode = addTerrain ? BrushOperationMode.Add : BrushOperationMode.Subtract;
        }

        public BrushOperation(Vector3 position, float radius, float strength,
BrushOperationMode mode)
        {
            this.position = position;
            this.radius = radius;
            this.strength = strength;
            this.mode = mode;
            this.addTerrain = mode != BrushOperationMode.Subtract;
        }
    }

    public class VoxelWorldGenerator : MonoBehaviour
    {
        [Header("Settings")]
        public WorldGenSettings settings;

        [Header("Editing")]
        public bool enableBrushOperations = true;
        public int maxBrushOperations = 512;

        [Header("Height Cache")]
        public bool enableHeightCache = true;
        public float heightCacheGridSize = 2f;
        public int maxHeightCacheEntries = 250000;

        [Header("Debug")]
        public bool drawWorldBounds = true;
        public bool drawBrushOperations = false;

        private readonly List<BrushOperation> brushOperationsList = new
List<BrushOperation>();
        private BrushOperation[] brushOperationsSnapshot = Array.Empty<BrushOperation>();

        private readonly ConcurrentDictionary<long, float> heightCache = new
ConcurrentDictionary<long, float>();

```

```

private int lastSettingsHash;

public int WorldSize => settings != null ? settings.worldSize : 2048;
public int WorldHeight => settings != null ? settings.worldHeight : 768;
public int BrushOperationCount => brushOperationsSnapshot.Length;
public int HeightCacheCount => heightCache.Count;

private void Reset()
{
    name = "VoxelWorldGenerator";
}

private void OnValidate()
{
    heightCacheGridSize = Mathf.Max(0.25f, heightCacheGridSize);
    maxHeightCacheEntries = Mathf.Max(1000, maxHeightCacheEntries);
}

[ContextMenu("Clear Height Cache")]
public void ClearHeightCache()
{
    heightCache.Clear();
}

public BrushOperation[] GetBrushOperationsSnapshot()
{
    return brushOperationsSnapshot;
}

public void AddBrushOperation(Vector3 position, float radius, float strength, bool
addTerrain)
{
    if (!enableBrushOperations)
        return;

    BrushOperation op = new BrushOperation(
        position,
        Mathf.Max(0.1f, radius),
        Mathf.Max(0.01f, strength),
        addTerrain
    );

    AddBrushOperationInternal(op);
}

public void AddSmoothBrushOperation(Vector3 position, float radius, float strength)
{
    if (!enableBrushOperations)
        return;

    BrushOperation op = new BrushOperation(
        position,
        Mathf.Max(0.1f, radius),
        Mathf.Max(0.01f, strength),
        BrushOperationMode.Smooth
    );

    AddBrushOperationInternal(op);
}

private void AddBrushOperationInternal(BrushOperation op)
{
    brushOperationsList.Add(op);

    while (brushOperationsList.Count > maxBrushOperations)

```

```

        brushOperationsList.RemoveAt(0);

        brushOperationsSnapshot = brushOperationsList.ToArray();
    }

    public void ClearBrushOperations()
    {
        brushOperationsList.Clear();
        brushOperationsSnapshot = Array.Empty<BrushOperation>();
    }

    public float SampleDensity(Vector3 worldPosition)
    {
        return SampleDensity(worldPosition.x, worldPosition.y, worldPosition.z);
    }

    public float SampleDensity(float x, float y, float z)
    {
        float height = SampleHeight(x, z);
        float density = height - y;

        if (enableBrushOperations)
            density += SampleBrushDensityDelta(x, y, z);

        return density;
    }

    public float SampleDensityFromKnownHeight(float x, float y, float z, float
knownHeight)
    {
        float density = knownHeight - y;

        if (enableBrushOperations)
            density += SampleBrushDensityDelta(x, y, z);

        return density;
    }

    public float SampleHeight(float x, float z)
    {
        if (settings == null)
            return 32f;

        CheckSettingsHash();

        if (!enableHeightCache)
            return TerrainShapeMixer.SampleHeight(settings, x, z);

        long key = MakeHeightCacheKey(x, z);

        if (heightCache.TryGetValue(key, out float cached))
            return cached;

        float h = TerrainShapeMixer.SampleHeight(settings, x, z);

        if (heightCache.Count < maxHeightCacheEntries)
            heightCache.TryAdd(key, h);

        return h;
    }

    private void CheckSettingsHash()
    {
        int hash = CalculateSettingsHash();

        if (hash == lastSettingsHash)
            return;
    }

```

```

        lastSettingsHash = hash;
        heightCache.Clear();
    }

private int CalculateSettingsHash()
{
    if (settings == null)
        return 0;

    unchecked
    {
        int hash = 17;

        hash = hash * 31 + settings.seed;
        hash = hash * 31 + (int)settings.generationMode;
        hash = hash * 31 + Mathf.RoundToInt(settings.baseHeight * 10f);
        hash = hash * 31 + Mathf.RoundToInt(settings.regionalScale * 10f);
        hash = hash * 31 + Mathf.RoundToInt(settings.regionalUplift * 10f);
        hash = hash * 31 + Mathf.RoundToInt(settings.regionalDepression * 10f);
        hash = hash * 31 + Mathf.RoundToInt(settings.waterLevel * 10f);

        if (settings.mountains != null)
        {
            hash = hash * 31 + (settings.mountains.enabled ? 1 : 0);
            hash = hash * 31 + Mathf.RoundToInt(settings.mountains.weight * 100f);
            hash = hash * 31 + settings.mountains.rangeCount;
        }

        if (settings.plains != null)
        {
            hash = hash * 31 + (settings.plains.enabled ? 1 : 0);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.weight * 100f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.largeHillScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.largeHillStrength *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.mediumHillScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.mediumHillStrength
* 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.smallHillScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.plains.smallHillStrength *
10f);
        }

        if (settings.rivers != null)
        {
            hash = hash * 31 + (settings.rivers.enabled ? 1 : 0);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.weight * 100f);
            hash = hash * 31 + settings.rivers.riverCount;
            hash = hash * 31 + settings.rivers.segmentsPerRiver;
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.riverLength * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.valleyWidth * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.canyonWidth * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.coreWidth * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.valleyDepth * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.rivers.canyonDepth * 10f);
        }

        if (settings.lakes != null)
        {
            hash = hash * 31 + (settings.lakes.enabled ? 1 : 0);
            hash = hash * 31 + Mathf.RoundToInt(settings.lakes.weight * 100f);
            hash = hash * 31 + settings.lakes.lakeCount;
            hash = hash * 31 + Mathf.RoundToInt(settings.lakes.radius * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.lakes.basinDepth * 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.lakes.lakeDepth * 10f);
        }
    }
}

```

```

        if (settings.detailNoise != null)
        {
            hash = hash * 31 + (settings.detailNoise.enabled ? 1 : 0);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.weight *
100f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.broadScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.broadStrength
* 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.mediumScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.mediumStrength
* 10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.smallScale *
10f);
            hash = hash * 31 + Mathf.RoundToInt(settings.detailNoise.smallStrength
* 10f);
        }

        return hash;
    }
}
private long MakeHeightCacheKey(float x, float z)
{
    float grid = Mathf.Max(0.25f, heightCacheGridSize);

    int ix = Mathf.RoundToInt(x / grid);
    int iz = Mathf.RoundToInt(z / grid);

    unchecked
    {
        return ((long)ix << 32) ^ (uint)iz;
    }
}

private float SampleBrushDensityDelta(float x, float y, float z)
{
    BrushOperation[] ops = brushOperationsSnapshot;

    if (ops == null || ops.Length == 0)
        return 0f;

    Vector3 p = new Vector3(x, y, z);
    float delta = 0f;

    for (int i = 0; i < ops.Length; i++)
    {
        BrushOperation op = ops[i];

        float r = Mathf.Max(0.001f, op.radius);
        float dist = Vector3.Distance(p, op.position);

        if (dist > r)
            continue;

        float t = 1f - dist / r;
        t = t * t * (3f - 2f * t);

        switch (op.mode)
        {
            case BrushOperationMode.Add:
                delta += op.strength * t;
                break;

            case BrushOperationMode.Subtract:

```



```

        delta -= op.strength * t;
        break;

    case BrushOperationMode.Smooth:
    {
        float smoothPower = Mathf.Clamp01(op.strength / 40f);
        float blend = smoothPower * t;
        delta = Mathf.Lerp(delta, 0f, blend);
        break;
    }
}

return delta;
}

public Color EvaluateSurfaceColor(Vector3 worldPosition, Vector3 normal)
{
    if (settings == null)
        return Color.white;

    float y = worldPosition.y;
    float normalY = Mathf.Clamp01(normal.normalized.y);

    bool nearWater = y <= settings.waterLevel + settings.sandHeightAboveWater;
    bool steep = normalY < settings.rockMinSlope;
    bool flatEnoughForGrass = normalY >= settings.grassMaxSlope;

    Color baseColor;

    if (nearWater)
    {
        baseColor = settings.sandColor;
    }
    else if (steep)
    {
        baseColor = settings.rockColor;
    }
    else if (flatEnoughForGrass)
    {
        baseColor = settings.grassColor;
    }
    else
    {
        float rockBlend = Mathf.InverseLerp(settings.grassMaxSlope,
settings.rockMinSlope, normalY);
        baseColor = Color.Lerp(settings.grassColor, settings.rockColor, rockBlend);
    }

    float snowMask = TerrainNoiseGenerator.EvaluateSnowMask(settings,
worldPosition, normal);
    return Color.Lerp(baseColor, settings.snow.snowColor, snowMask);
}

private void OnDrawGizmosSelected()
{
    if (settings != null && drawWorldBounds)
    {
        Gizmos.color = Color.yellow;

        Vector3 center = new Vector3(
            settings.worldSize * 0.5f,
            settings.worldHeight * 0.5f,
            settings.worldSize * 0.5f
        );

        Vector3 size = new Vector3(
            settings.worldSize,

```

```

        settings.worldHeight,
        settings.worldSize
    );

    Gizmos.DrawWireCube(center, size);
}

if (!drawBrushOperations)
    return;

BrushOperation[] ops = brushOperationsSnapshot;

if (ops == null)
    return;

for (int i = 0; i < ops.Length; i++)
{
    if (ops[i].mode == BrushOperationMode.Smooth)
        Gizmos.color = Color.yellow;
    else
        Gizmos.color = ops[i].addTerrain ? Color.green : Color.red;

    Gizmos.DrawWireSphere(ops[i].position, ops[i].radius);
}
}
}
}

```

Скрипт TerrainShapeMixer.cs

```

using UnityEngine;

namespace VoxelWorldGenerator.Generation
{
    public static class TerrainShapeMixer
    {
        public static float SampleHeight(WorldGenSettings settings, float x, float z)
        {
            if (settings == null)
                return 32f;

            TerrainSampleContext ctx = new TerrainSampleContext(settings, x, z);

            float h = settings.baseHeight;
            h += TerrainNoiseGenerator.EvaluateRegionalHeight(ctx);

            switch (settings.generationMode)
            {
                case TerrainGenerationMode.MountainsOnly:
                {
                    h += TerrainNoiseGenerator.EvaluatePlains(ctx) * 0.20f;
                    h += TerrainNoiseGenerator.EvaluateMountains(ctx) * 1.30f;
                    h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx) * 0.16f;
                    break;
                }

                case TerrainGenerationMode.PlainsOnly:
                {
                    h += TerrainNoiseGenerator.EvaluatePlains(ctx);
                    h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx) * 0.35f;
                    break;
                }

                case TerrainGenerationMode.RiversOnly:
                {
                    h += TerrainNoiseGenerator.EvaluatePlains(ctx);
                    h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx) * 0.25f;
                    h = TerrainNoiseGenerator.ApplyRiversToHeight(ctx, h);
                }
            }
        }
    }
}

```

```

        break;
    }

    case TerrainGenerationMode.LakesOnly:
    {
        h += TerrainNoiseGenerator.EvaluatePlains(ctx);
        h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx) * 0.20f;
        h += TerrainNoiseGenerator.EvaluateLakes(ctx);
        break;
    }

    case TerrainGenerationMode.NoiseOnly:
    {
        h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx);
        break;
    }

    case TerrainGenerationMode.Combined:
    default:
    {
        h += TerrainNoiseGenerator.EvaluatePlains(ctx) * 0.95f;
        h += TerrainNoiseGenerator.EvaluateMountains(ctx) * 1.10f;
        h += TerrainNoiseGenerator.EvaluateLakes(ctx) * 0.70f;
        h += TerrainNoiseGenerator.EvaluateDetailNoise(ctx) * 0.50f;

        h = TerrainNoiseGenerator.ApplyRiversToHeight(ctx, h);
        break;
    }
}

return SoftClampHeight(h, settings);
}

private static float SoftClampHeight(float height, WorldGenSettings settings)
{
    float minHeight = 2f;
    float maxHeight = settings.worldHeight - 2f;

    if (height < minHeight)
        return minHeight;

    float softLimitStart = settings.worldHeight * 0.84f;

    if (height <= softLimitStart)
        return height;

    float excess = height - softLimitStart;
    float compressed = Mathf.Log(1f + excess) * 24f;

    return Mathf.Min(softLimitStart + compressed, maxHeight);
}
}
}

```

Скрипт TerrainNoiseGenerator.cs

```

using UnityEngine;

namespace VoxelWorldGenerator.Generation
{
    public struct TerrainSampleContext
    {
        public WorldGenSettings settings;
        public float x;
        public float z;
        public int seed;
    }
}

```

```

public TerrainSampleContext(WorldGenSettings settings, float x, float z)
{
    this.settings = settings;
    this.x = x;
    this.z = z;
    this.seed = settings != null ? settings.seed : 0;
}

public static class TerrainNoiseGenerator
{
    #region Common Math

    public static float Hash01(int seed, int index, int salt)
    {
        float v = Mathf.Sin(seed * 12.9898f + index * 78.233f + salt * 37.719f) *
43758.5453f;
        return v - Mathf.Floor(v);
    }

    public static Vector2 FeaturePoint(int seed, int index, int salt, float worldSize)
    {
        float nx = Hash01(seed, index, salt + 13);
        float nz = Hash01(seed, index, salt + 71);

        nx = Mathf.Lerp(0.16f, 0.84f, nx);
        nz = Mathf.Lerp(0.16f, 0.84f, nz);

        return new Vector2(nx * worldSize, nz * worldSize);
    }

    public static float Fractal01(float x, float z, float scale, int octaves, int seed)
    {
        scale = Mathf.Max(1f, scale);
        octaves = Mathf.Max(1, octaves);

        float total = 0f;
        float amplitude = 1f;
        float frequency = 1f;
        float amplitudeSum = 0f;

        for (int i = 0; i < octaves; i++)
        {
            float sx = (x + seed * 17.31f + i * 91.7f) / scale * frequency;
            float sz = (z - seed * 11.17f - i * 43.3f) / scale * frequency;

            float n = Mathf.PerlinNoise(sx, sz);

            total += n * amplitude;
            amplitudeSum += amplitude;

            amplitude *= 0.5f;
            frequency *= 2f;
        }

        return total / Mathf.Max(0.0001f, amplitudeSum);
    }

    public static float Ridged01(float x, float z, float scale, int octaves, int seed)
    {
        float n = Fractal01(x, z, scale, octaves, seed);
        return Mathf.Clamp01(1f - Mathf.Abs(n * 2f - 1f));
    }

    public static float SmoothMask01(float distance, float radius)
    {

```

```

        radius = Mathf.Max(0.0001f, radius);

        float t = 1f - Mathf.Clamp01(distance / radius);
        return t * t * (3f - 2f * t);
    }

    public static float DistancePointSegment(Vector2 p, Vector2 a, Vector2 b)
    {
        Vector2 ab = b - a;
        float abLenSq = Vector2.Dot(ab, ab);

        if (abLenSq < 0.0001f)
            return Vector2.Distance(p, a);

        float t = Vector2.Dot(p - a, ab) / abLenSq;
        t = Mathf.Clamp01(t);

        Vector2 closest = a + ab * t;
        return Vector2.Distance(p, closest);
    }

    public static float SegmentAlong01(Vector2 p, Vector2 a, Vector2 b)
    {
        Vector2 ab = b - a;
        float abLenSq = Vector2.Dot(ab, ab);

        if (abLenSq < 0.0001f)
            return 0f;

        return Mathf.Clamp01(Vector2.Dot(p - a, ab) / abLenSq);
    }

    private static Vector2 Rotate(Vector2 v, float radians)
    {
        float c = Mathf.Cos(radians);
        float s = Mathf.Sin(radians);

        return new Vector2(
            v.x * c - v.y * s,
            v.x * s + v.y * c
        );
    }

    private static float TaperedLineMask(Vector2 p, Vector2 a, Vector2 b, float
startWidth, float endWidth)
    {
        float t = SegmentAlong01(p, a, b);
        float width = Mathf.Lerp(startWidth, endWidth, t);
        float dist = DistancePointSegment(p, a, b);
        return SmoothMask01(dist, width);
    }

    private static float LocalHash(int seed, int index, int salt)
    {
        return Hash01(seed, index, salt);
    }

#endregion

#region Regional Height

    public static float EvaluateRegionalHeight(TerrainSampleContext ctx)
    {
        WorldGenSettings s = ctx.settings;

```

```

    if (!s.enableRegionalHeight)
        return 0f;

    float n = Fractal01(ctx.x, ctx.z, s.regionalScale, 3, ctx.seed + 50);

    float upliftMask = Mathf.SmoothStep(0.48f, 0.86f, n);
    float depressionMask = 1f - Mathf.SmoothStep(0.12f, 0.44f, n);

    return upliftMask * s.regionalUplift - depressionMask * s.regionalDepression;
}

#endregion

#region Plains / Hills

public static float EvaluatePlains(TerrainSampleContext ctx)
{
    PlainsSettings p = ctx.settings.plains;

    if (!p.enabled)
        return 0f;

    float large = Fractal01(ctx.x, ctx.z, p.largeHillScale, 4, ctx.seed + 100) -
0.5f;
    float medium = Fractal01(ctx.x, ctx.z, p.mediumHillScale, 3, ctx.seed + 101) -
0.5f;
    float small = Fractal01(ctx.x, ctx.z, p.smallHillScale, 2, ctx.seed + 102) -
0.5f;

    float h =
        large * p.largeHillStrength +
        medium * p.mediumHillStrength +
        small * p.smallHillStrength;

    return h * p.weight;
}

#endregion

#region Mountains

private struct MountainPreset
{
    public float rangeLength;
    public float rangeWidth;
    public float coreWidth;
    public float crestWidth;

    public float baseHeight;
    public float coreHeight;
    public float crestHeight;
    public float peakHeight;

    public float centralRidgeWidth;
    public float centralRidgeHeight;

    public float domainWarpScale;
    public float domainWarpStrength;

    public float continentScale;
    public float continentStrength;

```

```

    public float erosionControlScale;
    public float weirdnessScale;
    public float ridgeControlScale;

    public float macroScale;
    public float macroStrength;

    public float ridgeScale;
    public float ridgeStrength;

    public float detailScale;
    public float detailStrength;

    public int valleyCount;
    public float valleyLengthMin;
    public float valleyLengthMax;
    public float valleyWidthTop;
    public float valleyWidthBottom;
    public float valleyDepth;

    public int spurCount;
    public float spurLengthMin;
    public float spurLengthMax;
    public float spurWidth;
    public float spurHeight;

    public int branchRidgeCount;
    public float branchRidgeLengthMin;
    public float branchRidgeLengthMax;
    public float branchRidgeWidth;
    public float branchRidgeHeight;
}

private static MountainPreset GetMountainPreset(WorldGenSettings settings)
{
    float w = settings != null ? settings.worldSize : 2048f;

    return new MountainPreset
    {
        rangeLength = w * 1.00f,
        rangeWidth = w * 0.24f,
        coreWidth = w * 0.32f,
        crestWidth = w * 0.15f,

        baseHeight = 260f,
        coreHeight = 520f,
        crestHeight = 560f,
        peakHeight = 420f,

        centralRidgeWidth = w * 0.060f,
        centralRidgeHeight = 620f,

        domainWarpScale = w * 0.34f,
        domainWarpStrength = w * 0.075f,

        continentScale = w * 0.95f,
        continentStrength = 170f,

        erosionControlScale = w * 0.42f,
        weirdnessScale = w * 0.30f,
        ridgeControlScale = w * 0.20f,

        macroScale = w * 0.16f,
        macroStrength = 195f,
    }
}

```

```

        ridgeScale = w * 0.055f,
        ridgeStrength = 145f,

        detailScale = w * 0.024f,
        detailStrength = 22f,

        valleyCount = 16,
        valleyLengthMin = w * 0.24f,
        valleyLengthMax = w * 0.56f,
        valleyWidthTop = w * 0.026f,
        valleyWidthBottom = w * 0.145f,
        valleyDepth = 245f,

        spurCount = 36,
        spurLengthMin = w * 0.16f,
        spurLengthMax = w * 0.46f,
        spurWidth = w * 0.022f,
        spurHeight = 135f,

        branchRidgeCount = 26,
        branchRidgeLengthMin = w * 0.22f,
        branchRidgeLengthMax = w * 0.52f,
        branchRidgeWidth = w * 0.020f,
        branchRidgeHeight = 175f
    };
}

public static float EvaluateMountains(TerrainSampleContext ctx)
{
    MountainSettings m = ctx.settings.mountains;

    if (!m.enabled)
        return 0f;

    MountainPreset preset = GetMountainPreset(ctx.settings);

    Vector2 p = new Vector2(ctx.x, ctx.z);

    float height = EvaluateMountainRange(ctx, p, preset, 0, true);

    int extra = Mathf.Max(0, m.rangeCount - 1);

    for (int i = 0; i < extra; i++)
        height += EvaluateMountainRange(ctx, p, preset, i + 1, false) * 0.55f;

    return height * m.weight;
}

private static float EvaluateMountainRange(
    TerrainSampleContext ctx,
    Vector2 world,
    MountainPreset preset,
    int rangeIndex,
    bool mainRange)
{
    float worldSize = ctx.settings.worldSize;

    Vector2 center = mainRange
        ? new Vector2(worldSize * 0.52f, worldSize * 0.52f)
        : FeaturePoint(ctx.seed, rangeIndex, 9100, worldSize);

    float angle = mainRange

```



```

        ? Mathf.Lerp(0.18f * Mathf.PI, 0.34f * Mathf.PI, LocalHash(ctx.seed,
rangeIndex, 9101))
        : LocalHash(ctx.seed, rangeIndex, 9102) * Mathf.PI * 2f;

Vector2 dir = new Vector2(Mathf.Cos(angle), Mathf.Sin(angle)).normalized;
Vector2 side = new Vector2(-dir.y, dir.x);

Vector2 p = DomainWarp(
    world,
    preset.domainWarpScale,
    preset.domainWarpStrength,
    ctx.seed + rangeIndex * 97
);

Vector2 d = p - center;

float along = Vector2.Dot(d, dir);
float cross = Vector2.Dot(d, side);

float halfLength = preset.rangeLength * 0.5f;

float along01 = Mathf.Clamp01((along + halfLength) / Mathf.Max(1f,
preset.rangeLength));
float endFade =
    Mathf.SmoothStep(0f, 0.05f, along01) *
    Mathf.SmoothStep(1f, 0.95f, along01);

float absCross = Mathf.Abs(cross);

float rangeMask = SmoothMask01(absCross, preset.rangeWidth) * endFade;
float coreMask = SmoothMask01(absCross, preset.coreWidth) * endFade;
float crestMask = SmoothMask01(absCross, preset.crestWidth) * endFade;

if (rangeMask <= 0.0001f)
    return 0f;

float continent = Fractal01(
    p.x,
    p.y,
    preset.continentScale,
    4,
    ctx.seed + 9200 + rangeIndex * 19
);

float erosion = Fractal01(
    p.x + 133.7f,
    p.y - 281.5f,
    preset.erosionControlScale,
    4,
    ctx.seed + 9210 + rangeIndex * 19
);

float weirdness = Fractal01(
    p.x - 411.2f,
    p.y + 93.4f,
    preset.weirdnessScale,
    4,
    ctx.seed + 9220 + rangeIndex * 19
);

float ridgeControl = Ridged01(
    p.x + 57.1f,
    p.y - 119.7f,
    preset.ridgeControlScale,
    4,
    ctx.seed + 9230 + rangeIndex * 19
);

```

```

);

continent = Mathf.SmoothStep(0.18f, 0.92f, continent);
erosion = Mathf.SmoothStep(0.10f, 0.95f, erosion);
weirdness = weirdness * 2f - 1f;
ridgeControl = Mathf.Pow(ridgeControl, 1.35f);

float broadBody =
    preset.continentStrength * continent * rangeMask +
    preset.baseHeight * Mathf.Pow(rangeMask, 1.02f) +
    preset.coreHeight * Mathf.Pow(coreMask, 1.05f) +
    preset.crestHeight * Mathf.Pow(crestMask, 1.01f);

float macro = Ridged01(
    p.x + 211.4f,
    p.y - 71.9f,
    preset.macroScale,
    5,
    ctx.seed + 9300 + rangeIndex * 31
);

macro = Mathf.Pow(macro, 1.40f);

float u = along;
float v = cross;

float ridgeA = Ridged01(
    u,
    v * 2.10f,
    preset.ridgeScale,
    5,
    ctx.seed + 9310 + rangeIndex * 31
);

float ridgeB = Ridged01(
    u * 1.55f + 317.2f,
    v * 3.60f - 81.7f,
    preset.ridgeScale * 0.52f,
    4,
    ctx.seed + 9320 + rangeIndex * 31
);

float ridgeC = Ridged01(
    u * 2.30f - 113.6f,
    v * 5.20f + 219.2f,
    preset.ridgeScale * 0.27f,
    3,
    ctx.seed + 9330 + rangeIndex * 31
);

ridgeA = Mathf.Pow(ridgeA, 1.35f);
ridgeB = Mathf.Pow(ridgeB, 1.75f);
ridgeC = Mathf.Pow(ridgeC, 2.10f);

float highMountainMask = Mathf.Clamp01(coreMask * 0.75f + crestMask * 1.55f);
float highDetailMask = Mathf.SmoothStep(0.18f, 0.82f, highMountainMask);

float ridgeField =
    ridgeA * preset.ridgeStrength * 0.46f +
    ridgeB * preset.ridgeStrength * 0.27f +
    ridgeC * preset.ridgeStrength * 0.12f;

float ridgeMask = Mathf.Clamp01(
    coreMask * 0.72f +
    crestMask * 1.55f +
    ridgeControl * rangeMask * 0.32f
);

```

```

ridgeField *= Mathf.Lerp(0.30f, 1.0f, highDetailMask);

float peakNoise = Ridged01(
    p.x - 719.1f,
    p.y + 148.5f,
    preset.ridgeScale * 1.20f,
    5,
    ctx.seed + 9340 + rangeIndex * 31
);

float peakMask = Mathf.Pow(Mathf.Clamp01(peakNoise * ridgeControl), 2.8f);
float peaks = peakMask * preset.peakHeight * Mathf.Clamp01(coreMask + crestMask
* 1.8f);

float centralRidge = EvaluateCentralRidge(
    ctx,
    p,
    center,
    dir,
    side,
    preset,
    rangeIndex,
    along,
    cross,
    coreMask,
    crestMask,
    ridgeControl
);

float branchRidges = EvaluateBranchRidges(
    ctx,
    p,
    center,
    dir,
    side,
    preset,
    rangeIndex,
    rangeMask,
    coreMask,
    ridgeControl
);

float spurs = EvaluateSpurs(ctx, p, center, dir, side, preset, rangeIndex,
rangeMask, ridgeMask);
float valleys = EvaluateErosionValleys(ctx, p, center, dir, side, preset,
rangeIndex, rangeMask, coreMask);

float interRidgeValleys = EvaluateInterRidgeValleys(
    ctx,
    p,
    center,
    dir,
    side,
    preset,
    rangeIndex,
    rangeMask,
    coreMask,
    crestMask
);

float drainageA = Ridged01(
    p.x + 17.9f,
    p.y - 303.1f,
    preset.ridgeScale * 0.72f,
    5,
    ctx.seed + 9400 + rangeIndex * 31
);

```

```

float drainageB = Ridged01(
    p.x - 241.3f,
    p.y + 89.8f,
    preset.ridgeScale * 0.33f,
    4,
    ctx.seed + 9410 + rangeIndex * 31
);

float drainage = Mathf.Clamp01(drainageA * 0.65f + drainageB * 0.35f);
drainage = Mathf.Pow(drainage, 2.25f);

float drainageCut =
    drainage *
    Mathf.Clamp01(rangeMask - crestMask * 0.18f) *
    Mathf.Lerp(24f, 105f, erosion) *
    Mathf.Clamp01(ridgeControl * 0.55f + coreMask * 1.15f) *
    Mathf.Lerp(0.35f, 1.0f, highDetailMask);

float cliffNoise = Ridged01(
    p.x + 811.5f,
    p.y - 512.4f,
    preset.ridgeScale * 0.40f,
    4,
    ctx.seed + 9500 + rangeIndex * 37
);

cliffNoise = Mathf.Pow(cliffNoise, 2.0f);

float cliffMask = Mathf.Clamp01(coreMask * 1.00f + crestMask * 1.80f +
ridgeControl * 0.45f);
float cliffLift = Mathf.Max(0f, cliffNoise - 0.30f) * 130f * cliffMask;
float cliffCut = Mathf.Max(0f, cliffNoise - 0.62f) * 82f * cliffMask;

float detail = (Fractal01(
    p.x + 19.4f,
    p.y - 77.2f,
    preset.detailScale,
    3,
    ctx.seed + 9600 + rangeIndex * 31
) - 0.5f) * preset.detailStrength * ridgeMask * highDetailMask;

float erosionFlatten = Mathf.Lerp(1.10f, 0.78f, erosion * (1f - ridgeControl *
0.40f));

float height =
    broadBody * erosionFlatten +
    macro * preset.macroStrength * rangeMask +
    ridgeField * ridgeMask +
    centralRidge +
    branchRidges +
    peaks +
    spurs +
    cliffLift +
    detail -
    valleys -
    interRidgeValleys -
    drainageCut -
    cliffCut;

height += weirdness * 54f * rangeMask * Mathf.Clamp01(1f - crestMask * 0.18f);

if (mainRange)
    height *= 1.26f;
else
    height *= 0.98f;

return Mathf.Max(0f, height);
}

```

```

private static Vector2 DomainWarp(Vector2 p, float scale, float strength, int seed)
{
    float wx1 = Fractal01(p.x, p.y, scale, 4, seed + 1) - 0.5f;
    float wy1 = Fractal01(p.x + 173.2f, p.y - 91.7f, scale, 4, seed + 2) - 0.5f;

    Vector2 p2 = p + new Vector2(wx1, wy1) * strength;

    float wx2 = Fractal01(p2.x - 311.3f, p2.y + 97.5f, scale * 0.45f, 3, seed + 3)
- 0.5f;
    float wy2 = Fractal01(p2.x + 41.9f, p2.y - 229.4f, scale * 0.45f, 3, seed + 4)
- 0.5f;

    return p2 + new Vector2(wx2, wy2) * strength * 0.45f;
}

private static float EvaluateCentralRidge(
    TerrainSampleContext ctx,
    Vector2 p,
    Vector2 center,
    Vector2 dir,
    Vector2 side,
    MountainPreset preset,
    int rangeIndex,
    float along,
    float cross,
    float coreMask,
    float crestMask,
    float ridgeControl)
{
    float halfLength = preset.rangeLength * 0.5f;

    Vector2 a = center - dir * halfLength;
    Vector2 b = center + dir * halfLength;

    float mainLine = TaperedLineMask(
        p,
        a,
        b,
        preset.centralRidgeWidth * 0.95f,
        preset.centralRidgeWidth * 1.65f
    );

    mainLine = Mathf.Pow(mainLine, 1.18f);

    float along01 = Mathf.Clamp01((along + halfLength) / Mathf.Max(1f,
preset.rangeLength));

    float endFade =
        Mathf.SmoothStep(0f, 0.05f, along01) *
        Mathf.SmoothStep(1f, 0.95f, along01);

    float wave = Ridged01(
        along,
        cross * 1.7f,
        preset.ridgeScale * 1.35f,
        4,
        ctx.seed + 9900 + rangeIndex * 31
    );

    wave = Mathf.Pow(wave, 1.28f);

    float segmentedPeaks = 0f;
    int peakCount = 8;

    for (int i = 0; i < peakCount; i++)

```

```

    {
        float t = Mathf.Lerp(0.06f, 0.94f, (i + 0.5f) / peakCount);
        t += (LocalHash(ctx.seed, rangeIndex * 700 + i, 9901) - 0.5f) * 0.045f;
        t = Mathf.Clamp01(t);

        float distAlong = Mathf.Abs(along01 - t);
        float peakMask = SmoothMask01(distAlong, 0.10f);
        peakMask = Mathf.Pow(peakMask, 1.55f);

        segmentedPeaks += peakMask * Mathf.Lerp(70f, 150f, LocalHash(ctx.seed,
rangeIndex * 700 + i, 9902));
    }

    float ridge =
        mainLine *
        (preset.centralRidgeHeight + segmentedPeaks) *
        Mathf.Lerp(0.95f, 1.50f, ridgeControl) *
        endFade;

    ridge += crestMask * wave * 170f * endFade;
    ridge += coreMask * wave * 120f * endFade;

    return Mathf.Clamp(ridge, 0f, 520f);
}

private static float EvaluateBranchRidges(
    TerrainSampleContext ctx,
    Vector2 p,
    Vector2 center,
    Vector2 dir,
    Vector2 side,
    MountainPreset preset,
    int rangeIndex,
    float rangeMask,
    float coreMask,
    float ridgeControl)
{
    float result = 0f;
    float halfLength = preset.rangeLength * 0.5f;

    for (int i = 0; i < preset.branchRidgeCount; i++)
    {
        float t = Mathf.Lerp(0.05f, 0.95f, (i + 0.5f) / preset.branchRidgeCount);
        t += (LocalHash(ctx.seed, rangeIndex * 800 + i, 9920) - 0.5f) * 0.060f;
        t = Mathf.Clamp01(t);

        Vector2 root = center + dir * Mathf.Lerp(-halfLength, halfLength, t);

        int sign = i % 2 == 0 ? 1 : -1;

        float angle = Mathf.Lerp(
            0.48f,
            1.15f,
            LocalHash(ctx.seed, rangeIndex * 800 + i, 9921)
        );

        Vector2 branchDir = Rotate(dir, sign * angle).normalized;

        float length = Mathf.Lerp(
            preset.branchRidgeLengthMin,
            preset.branchRidgeLengthMax,
            LocalHash(ctx.seed, rangeIndex * 800 + i, 9922)
        );

        Vector2 end = root + branchDir * length;
    }
}

```

```

float mask = TaperedLineMask(
    p,
    root,
    end,
    preset.branchRidgeWidth,
    preset.branchRidgeWidth * 3.8f
);

float along = SegmentAlong01(p, root, end);
float taper = Mathf.Lerp(1f, 0.08f, along);

float h =
    Mathf.Pow(mask, 1.35f) *
    taper *
    preset.branchRidgeHeight *
    Mathf.Clamp01(coreMask + rangeMask * 0.55f) *
    Mathf.Lerp(0.95f, 1.45f, ridgeControl);

result += h;

if (i % 2 == 0)
{
    Vector2 mid = Vector2.Lerp(root, end, Mathf.Lerp(0.34f, 0.64f,
LocalHash(ctx.seed, rangeIndex * 800 + i, 9923)));
    Vector2 subDir = Rotate(branchDir, -sign * Mathf.Lerp(0.38f, 0.78f,
LocalHash(ctx.seed, rangeIndex * 800 + i, 9924)));
    Vector2 subEnd = mid + subDir * length * Mathf.Lerp(0.30f, 0.58f,
LocalHash(ctx.seed, rangeIndex * 800 + i, 9925));

    float subMask = TaperedLineMask(
        p,
        mid,
        subEnd,
        preset.branchRidgeWidth * 0.55f,
        preset.branchRidgeWidth * 1.80f
    );

    float subAlong = SegmentAlong01(p, mid, subEnd);
    float subTaper = Mathf.Lerp(1f, 0.08f, subAlong);

    result += Mathf.Pow(subMask, 1.45f) *
        subTaper *
        preset.branchRidgeHeight *
        0.50f *
        rangeMask;
}
}

return Mathf.Clamp(result, 0f, 420f);
}

private static float EvaluateSpurs(
    TerrainSampleContext ctx,
    Vector2 p,
    Vector2 center,
    Vector2 dir,
    Vector2 side,
    MountainPreset preset,
    int rangeIndex,
    float rangeMask,
    float ridgeMask)
{
    float result = 0f;
    float halfLength = preset.rangeLength * 0.5f;

    for (int i = 0; i < preset.spurCount; i++)
    {
        float t = Mathf.Lerp(0.05f, 0.95f, (i + 0.5f) / preset.spurCount);

```

```

t += (LocalHash(ctx.seed, rangeIndex * 200 + i, 9700) - 0.5f) * 0.055f;
t = Mathf.Clamp01(t);

float along = Mathf.Lerp(-halfLength, halfLength, t);

Vector2 root = center + dir * along;

int sign = i % 2 == 0 ? 1 : -1;

float angle = Mathf.Lerp(0.42f, 1.12f, LocalHash(ctx.seed, rangeIndex * 200
+ i, 9701));
Vector2 spurDir = Rotate(dir, angle * sign).normalized;

float length = Mathf.Lerp(
    preset.spurLengthMin,
    preset.spurLengthMax,
    LocalHash(ctx.seed, rangeIndex * 200 + i, 9702)
);

Vector2 end = root + spurDir * length;

float mask = TaperedLineMask(p, root, end, preset.spurWidth,
preset.spurWidth * 3.5f);
float alongSpur = SegmentAlong01(p, root, end);
float taper = Mathf.Lerp(1f, 0.08f, alongSpur);

result += Mathf.Pow(mask, 1.45f) * taper * preset.spurHeight * ridgeMask;

if (i % 2 == 0)
{
    Vector2 mid = Vector2.Lerp(root, end, Mathf.Lerp(0.35f, 0.62f,
LocalHash(ctx.seed, rangeIndex * 200 + i, 9703)));
    Vector2 branchDir = Rotate(spurDir, -sign * Mathf.Lerp(0.42f, 0.82f,
LocalHash(ctx.seed, rangeIndex * 200 + i, 9704)));
    Vector2 branchEnd = mid + branchDir * length * Mathf.Lerp(0.28f, 0.56f,
LocalHash(ctx.seed, rangeIndex * 200 + i, 9705));

    float bMask = TaperedLineMask(p, mid, branchEnd, preset.spurWidth *
0.55f, preset.spurWidth * 1.7f);
    float bAlong = SegmentAlong01(p, mid, branchEnd);
    float bTaper = Mathf.Lerp(1f, 0.10f, bAlong);

    result += Mathf.Pow(bMask, 1.5f) * bTaper * preset.spurHeight * 0.52f *
rangeMask;
}
}

return result;
}

private static float EvaluateErosionValleys(
    TerrainSampleContext ctx,
    Vector2 p,
    Vector2 center,
    Vector2 dir,
    Vector2 side,
    MountainPreset preset,
    int rangeIndex,
    float rangeMask,
    float coreMask)
{
    float result = 0f;
    float halfLength = preset.rangeLength * 0.5f;

    for (int i = 0; i < preset.valleyCount; i++)
    {

```



```

float t = Mathf.Lerp(0.05f, 0.95f, (i + 0.5f) / preset.valleyCount);
t += (LocalHash(ctx.seed, rangeIndex * 300 + i, 9800) - 0.5f) * 0.05f;
t = Mathf.Clamp01(t);

float along = Mathf.Lerp(-halfLength, halfLength, t);
Vector2 root = center + dir * along;

int sign = i % 2 == 0 ? 1 : -1;

float angle = Mathf.Lerp(0.70f, 1.30f, LocalHash(ctx.seed, rangeIndex * 300
+ i, 9801));
Vector2 valleyDir = Rotate(dir, angle * sign).normalized;

float length = Mathf.Lerp(
    preset.valleyLengthMin,
    preset.valleyLengthMax,
    LocalHash(ctx.seed, rangeIndex * 300 + i, 9802)
);

Vector2 end = root + valleyDir * length;

Vector2 bendSide = new Vector2(-valleyDir.y, valleyDir.x);
Vector2 mid = Vector2.Lerp(root, end, 0.52f) +
    bendSide * length * Mathf.Lerp(-0.18f, 0.18f,
LocalHash(ctx.seed, rangeIndex * 300 + i, 9803));

float d1 = DistancePointSegment(p, root, mid);
float d2 = DistancePointSegment(p, mid, end);
float d = Mathf.Min(d1, d2);

float t1 = SegmentAlong01(p, root, mid) * 0.5f;
float t2 = 0.5f + SegmentAlong01(p, mid, end) * 0.5f;
float localT = Mathf.Min(t1, t2);

float width = Mathf.Lerp(preset.valleyWidthTop, preset.valleyWidthBottom,
localT);
float mask = SmoothMask01(d, width);

mask = Mathf.Pow(mask, 1.18f);

float depth = Mathf.Lerp(preset.valleyDepth * 0.75f, preset.valleyDepth *
1.25f, localT);

float cutMask = Mathf.Clamp01(rangeMask + coreMask * 0.85f);
result += mask * depth * cutMask;

for (int b = 0; b < 3; b++)
{
    float attachT = Mathf.Lerp(0.22f, 0.74f, LocalHash(ctx.seed, rangeIndex
* 400 + i * 10 + b, 9810));
    Vector2 attach = Vector2.Lerp(root, end, attachT);

    int bSign = (b % 2 == 0) ? -sign : sign;

    Vector2 bDir = Rotate(
        valleyDir,
        bSign * Mathf.Lerp(0.36f, 0.92f, LocalHash(ctx.seed, rangeIndex *
400 + i * 10 + b, 9811))
    );

    Vector2 bEnd = attach + bDir * length * Mathf.Lerp(0.26f, 0.56f,
LocalHash(ctx.seed, rangeIndex * 400 + i * 10 + b, 9812));

    float bMask = TaperedLineMask(

```

```

        p,
        attach,
        bEnd,
        preset.valleyWidthTop * 0.55f,
        preset.valleyWidthBottom * 0.55f
    );

    float bAlong = SegmentAlong01(p, attach, bEnd);

    result += Mathf.Pow(bMask, 1.30f) *
        Mathf.Lerp(preset.valleyDepth * 0.25f, preset.valleyDepth *
0.70f, bAlong) *
        rangeMask;
    }
}

return result;
}

private static float EvaluateInterRidgeValleys(
    TerrainSampleContext ctx,
    Vector2 p,
    Vector2 center,
    Vector2 dir,
    Vector2 side,
    MountainPreset preset,
    int rangeIndex,
    float rangeMask,
    float coreMask,
    float crestMask)
{
    float result = 0f;
    float halfLength = preset.rangeLength * 0.5f;

    int count = 18;

    for (int i = 0; i < count; i++)
    {
        float t = Mathf.Lerp(0.06f, 0.94f, (i + 0.5f) / count);
        t += (LocalHash(ctx.seed, rangeIndex * 910 + i, 9820) - 0.5f) * 0.055f;
        t = Mathf.Clamp01(t);

        Vector2 root = center + dir * Mathf.Lerp(-halfLength, halfLength, t);

        int sign = i % 2 == 0 ? 1 : -1;

        float angle = Mathf.Lerp(
            0.82f,
            1.34f,
            LocalHash(ctx.seed, rangeIndex * 910 + i, 9821)
        );

        Vector2 valleyDir = Rotate(dir, sign * angle).normalized;

        float length = Mathf.Lerp(
            preset.rangeWidth * 0.30f,
            preset.rangeWidth * 0.78f,
            LocalHash(ctx.seed, rangeIndex * 910 + i, 9822)
        );

        Vector2 end = root + valleyDir * length;

        Vector2 bendSide = new Vector2(-valleyDir.y, valleyDir.x);
        Vector2 mid = Vector2.Lerp(root, end, 0.55f) +
            bendSide * length * Mathf.Lerp(
                -0.16f,

```

```

        0.16f,
        LocalHash(ctx.seed, rangeIndex * 910 + i, 9823)
    );

    float d1 = DistancePointSegment(p, root, mid);
    float d2 = DistancePointSegment(p, mid, end);
    float d = Mathf.Min(d1, d2);

    float t1 = SegmentAlong01(p, root, mid) * 0.5f;
    float t2 = 0.5f + SegmentAlong01(p, mid, end) * 0.5f;
    float localT = Mathf.Min(t1, t2);

    float width = Mathf.Lerp(
        preset.valleyWidthTop * 0.85f,
        preset.valleyWidthBottom * 0.95f,
        localT
    );

    float mask = SmoothMask01(d, width);
    mask = Mathf.Pow(mask, 1.22f);

    float crestProtection = 1f - Mathf.Clamp01(crestMask * 1.25f);
    float verticalMask = Mathf.Clamp01(coreMask * 0.85f + rangeMask * 0.35f);

    float depth = Mathf.Lerp(
        preset.valleyDepth * 0.45f,
        preset.valleyDepth * 0.95f,
        localT
    );

    result += mask * depth * verticalMask * crestProtection;

    if (i % 2 == 0)
    {
        Vector2 attach = Vector2.Lerp(
            root,
            end,
            Mathf.Lerp(0.35f, 0.68f, LocalHash(ctx.seed, rangeIndex * 910 + i,
9824))
        );

        int bSign = sign > 0 ? -1 : 1;

        Vector2 branchDir = Rotate(
            valleyDir,
            bSign * Mathf.Lerp(0.38f, 0.78f, LocalHash(ctx.seed, rangeIndex *
910 + i, 9825))
        );

        Vector2 branchEnd = attach + branchDir * length * Mathf.Lerp(
            0.25f,
            0.45f,
            LocalHash(ctx.seed, rangeIndex * 910 + i, 9826)
        );

        float bMask = TaperedLineMask(
            p,
            attach,
            branchEnd,
            preset.valleyWidthTop * 0.45f,
            preset.valleyWidthBottom * 0.42f
        );

        float bAlong = SegmentAlong01(p, attach, branchEnd);

        result += Mathf.Pow(bMask, 1.35f) *

```

```

0.42f, bAlong) *
    Mathf.Lerp(preset.valleyDepth * 0.18f, preset.valleyDepth *
        verticalMask *
        crestProtection;
    }
}

return result;
}

#endregion

#region Rivers

private static Vector2 GetRiverPolylinePoint(
    TerrainSampleContext ctx,
    int riverIndex,
    RiverSettings r,
    Vector2 start,
    Vector2 end,
    float t)
{
    Vector2 basePoint = Vector2.Lerp(start, end, t);
    Vector2 dir = (end - start).normalized;
    Vector2 side = new Vector2(-dir.y, dir.x);

    float n1 = Fractal01(
        basePoint.x,
        basePoint.y,
        r.meanderScalePrimary,
        2,
        ctx.seed + 410 + riverIndex * 17) - 0.5f;

    float n2 = Fractal01(
        basePoint.x + 173.2f,
        basePoint.y - 91.4f,
        r.meanderScaleSecondary,
        2,
        ctx.seed + 411 + riverIndex * 17) - 0.5f;

    float offset =
        n1 * 2f * r.meanderAmplitudePrimary +
        n2 * 2f * r.meanderAmplitudeSecondary;

    return basePoint + side * offset;
}

#endregion

#region Lakes

public static float EvaluateLakes(TerrainSampleContext ctx)
{
    LakeSettings l = ctx.settings.lakes;

    if (!l.enabled)
        return 0f;

    Vector2 p = new Vector2(ctx.x, ctx.z);
    float cut = 0f;

    for (int i = 0; i < l.lakeCount; i++)
    {
        Vector2 center = FeaturePoint(ctx.seed, i, 500, ctx.settings.worldSize);

```

```

501));

    float baseRadius = l.radius * Mathf.Lerp(0.8f, 1.45f, Hash01(ctx.seed, i,

    float shoreNoise = Fractal01(
        ctx.x,
        ctx.z,
        l.shoreNoiseScale,
        3,
        ctx.seed + 510 + i * 29);

    float radius = baseRadius * Mathf.Lerp(
        1f - l.shoreNoiseStrength,
        1f + l.shoreNoiseStrength,
        shoreNoise);

    float dist = Vector2.Distance(p, center);

    float lakeMask = SmoothMask01(dist, radius);
    float basinMask = SmoothMask01(dist, radius * l.basinRadiusMultiplier);

    cut += l.basinDepth * basinMask;
    cut += l.lakeDepth * lakeMask;
}

    return -cut * l.weight;
}

#endregion

#region Detail Noise

public static float EvaluateDetailNoise(TerrainSampleContext ctx)
{
    DetailNoiseSettings n = ctx.settings.detailNoise;

    if (!n.enabled)
        return 0f;

    float broad = Fractal01(ctx.x, ctx.z, n.broadScale, 4, ctx.seed + 600) - 0.5f;
    float medium = Fractal01(ctx.x, ctx.z, n.mediumScale, 3, ctx.seed + 601) -
0.5f;
    float small = Fractal01(ctx.x, ctx.z, n.smallScale, 2, ctx.seed + 602) - 0.5f;

    float h =
        broad * n.broadStrength +
        medium * n.mediumStrength +
        small * n.smallStrength;

    return h * n.weight;
}

#endregion
}

```

Додаток Г

Код програмного модуля побудови поверхні

Скрипт SurfaceNetsMesher.cs

```

using VoxelWorldGenerator.Core;
using VoxelWorldGenerator.Generation;
using UnityEngine;

namespace VoxelWorldGenerator.Meshing
{
    public static class SurfaceNetsMesher
    {
        private static readonly Vector3Int[] CornerOffsets =
        {
            new Vector3Int(0, 0, 0),
            new Vector3Int(1, 0, 0),
            new Vector3Int(1, 1, 0),
            new Vector3Int(0, 1, 0),
            new Vector3Int(0, 0, 1),
            new Vector3Int(1, 0, 1),
            new Vector3Int(1, 1, 1),
            new Vector3Int(0, 1, 1)
        };

        private static readonly int[,] EdgeCorners =
        {
            {0, 1}, {1, 2}, {2, 3}, {3, 0},
            {4, 5}, {5, 6}, {6, 7}, {7, 4},
            {0, 4}, {1, 5}, {2, 6}, {3, 7}
        };

        public static MeshData Build(
            VoxelWorldGenerator generator,
            int originX,
            int originZ,
            int tileSize,
            int worldHeight,
            int step)
        {
            step = Mathf.Max(1, step);
            tileSize = Mathf.Max(step, tileSize);

            int gutterCells = 1;
            int buildOriginX = originX - gutterCells * step;
            int buildOriginZ = originZ - gutterCells * step;

            int baseCellsX = Mathf.CeilToInt(tileSize / (float)step);
            int baseCellsZ = Mathf.CeilToInt(tileSize / (float)step);

            int cellsX = baseCellsX + gutterCells * 2;
            int cellsZ = baseCellsZ + gutterCells * 2;

            EstimateVerticalRange(
                generator,
                originX,
                originZ,
                tileSize,
                worldHeight,
                step,
                out int yStart,
                out int yEnd
            );

            int verticalSize = Mathf.Max(step, yEnd - yStart);
            int cellsY = Mathf.CeilToInt(verticalSize / (float)step);

```

```

float[,] densities = new float[cellsX + 1, cellsY + 1, cellsZ + 1];
int[,] vertexIndices = new int[cellsX, cellsY, cellsZ];

float[,] sampledHeights = new float[cellsX + 1, cellsZ + 1];

for (int gx = 0; gx <= cellsX; gx++)
{
    for (int gz = 0; gz <= cellsZ; gz++)
    {
        float wx = buildOriginX + gx * step;
        float wz = buildOriginZ + gz * step;

        sampledHeights[gx, gz] = generator.SampleHeight(wx, wz);
    }
}

for (int x = 0; x < cellsX; x++)
for (int y = 0; y < cellsY; y++)
for (int z = 0; z < cellsZ; z++)
    vertexIndices[x, y, z] = -1;

for (int gx = 0; gx <= cellsX; gx++)
{
    for (int gz = 0; gz <= cellsZ; gz++)
    {
        float wx = buildOriginX + gx * step;
        float wz = buildOriginZ + gz * step;
        float knownHeight = sampledHeights[gx, gz];

        for (int gy = 0; gy <= cellsY; gy++)
        {
            float wy = yStart + gy * step;

            float d = generator.SampleDensityFromKnownHeight(
                wx,
                wy,
                wz,
                knownHeight
            );

            densities[gx, gy, gz] = NormalizeDensityForMeshing(d);
        }
    }
}

MeshData meshData = new MeshData();

for (int cx = 0; cx < cellsX; cx++)
{
    for (int cy = 0; cy < cellsY; cy++)
    {
        for (int cz = 0; cz < cellsZ; cz++)
        {
            float[] cubeDensities = new float[8];
            Vector3[] cubePositions = new Vector3[8];

            bool hasPositive = false;
            bool hasNegative = false;

            for (int i = 0; i < 8; i++)
            {
                Vector3Int o = CornerOffsets[i];

                float density = densities[cx + o.x, cy + o.y, cz + o.z];
                cubeDensities[i] = density;
            }
        }
    }
}

```

```

        if (density > 0f) hasPositive = true;
        if (density < 0f) hasNegative = true;

        cubePositions[i] = new Vector3(
            buildOriginX + (cx + o.x) * step,
            yStart + (cy + o.y) * step,
            buildOriginZ + (cz + o.z) * step
        );
    }

    if (!hasPositive || !hasNegative)
        continue;

    Vector3 average = Vector3.zero;
    int crossings = 0;

    for (int e = 0; e < 12; e++)
    {
        int a = EdgeCorners[e, 0];
        int b = EdgeCorners[e, 1];

        float da = cubeDensities[a];
        float db = cubeDensities[b];

        if (CrossesSurface(da, db))
        {
            float t = da / (da - db);
            t = Mathf.Clamp01(t);

            Vector3 p = Vector3.Lerp(cubePositions[a],
cubePositions[b], t);

            average += p;
            crossings++;
        }
    }

    if (crossings == 0)
        continue;

    Vector3 vertex = average / crossings;

    Vector2 uv = new Vector2(
        vertex.x * 0.01f,
        vertex.z * 0.01f
    );

    int index = meshData.AddVertex(vertex, uv);
    vertexIndices[cx, cy, cz] = index;
}
}

    BuildFacesX(generator, meshData, densities, vertexIndices, buildOriginX,
buildOriginZ, originX, originZ, tileSize, step, cellsX, cellsY, cellsZ, yStart);
    BuildFacesY(generator, meshData, densities, vertexIndices, buildOriginX,
buildOriginZ, originX, originZ, tileSize, step, cellsX, cellsY, cellsZ, yStart);
    BuildFacesZ(generator, meshData, densities, vertexIndices, buildOriginX,
buildOriginZ, originX, originZ, tileSize, step, cellsX, cellsY, cellsZ, yStart);

    return meshData;
}

private static void EstimateVerticalRange(
    VoxelWorldGenerator generator,
    int originX,
    int originZ,

```



```

int tileSize,
int worldHeight,
int step,
out int yStart,
out int yEnd)
{
    float minH = float.MaxValue;
    float maxH = float.MinValue;

    int sampleStep = Mathf.Max(16, step * 4);

    for (int x = originX - step; x <= originX + tileSize + step; x += sampleStep)
    {
        for (int z = originZ - step; z <= originZ + tileSize + step; z +=
sampleStep)
        {
            float h = generator.SampleHeight(x, z);
            minH = Mathf.Min(minH, h);
            maxH = Mathf.Max(maxH, h);
        }
    }

    float h00 = generator.SampleHeight(originX, originZ);
    float h10 = generator.SampleHeight(originX + tileSize, originZ);
    float h01 = generator.SampleHeight(originX, originZ + tileSize);
    float h11 = generator.SampleHeight(originX + tileSize, originZ + tileSize);

    minH = Mathf.Min(minH, h00, h10, h01, h11);
    maxH = Mathf.Max(maxH, h00, h10, h01, h11);

    BrushOperation[] ops = generator.GetBrushOperationsSnapshot();

    if (ops != null && ops.Length > 0)
    {
        float tileMinX = originX - step;
        float tileMaxX = originX + tileSize + step;
        float tileMinZ = originZ - step;
        float tileMaxZ = originZ + tileSize + step;

        for (int i = 0; i < ops.Length; i++)
        {
            BrushOperation op = ops[i];

            float r = Mathf.Max(0.1f, op.radius);

            float opMinX = op.position.x - r;
            float opMaxX = op.position.x + r;
            float opMinZ = op.position.z - r;
            float opMaxZ = op.position.z + r;

            bool overlapsTile =
                opMinX <= tileMaxX &&
                opMaxX >= tileMinX &&
                opMinZ <= tileMaxZ &&
                opMaxZ >= tileMinZ;

            if (!overlapsTile)
                continue;

            float verticalExtra = r + Mathf.Abs(op.strength) * 2f + step * 4f;

            minH = Mathf.Min(minH, op.position.y - verticalExtra);
            maxH = Mathf.Max(maxH, op.position.y + verticalExtra);
        }
    }
}

```

```

int margin = Mathf.Max(step * 4, 16);

yStart = Mathf.FloorToInt((minH - margin) / step) * step;
yEnd = Mathf.CeilToInt((maxH + margin) / step) * step;

yStart = Mathf.Clamp(yStart, 0, worldHeight - step);
yEnd = Mathf.Clamp(yEnd, yStart + step, worldHeight);
}

private static void BuildFacesX(
    VoxelWorldGenerator generator,
    MeshData meshData,
    float[, ,] densities,
    int[, ,] vertexIndices,
    int buildOriginX,
    int buildOriginZ,
    int tileOriginX,
    int tileOriginZ,
    int tileSize,
    int step,
    int cellsX,
    int cellsY,
    int cellsZ,
    int yStart)
{
    for (int x = 0; x < cellsX; x++)
    {
        for (int y = 1; y < cellsY; y++)
        {
            for (int z = 1; z < cellsZ; z++)
            {
                float a = densities[x, y, z];
                float b = densities[x + 1, y, z];

                if (!CrossesSurface(a, b))
                    continue;

                int v0 = vertexIndices[x, y - 1, z - 1];
                int v1 = vertexIndices[x, y, z - 1];
                int v2 = vertexIndices[x, y, z];
                int v3 = vertexIndices[x, y - 1, z];

                if (AnyInvalid(v0, v1, v2, v3))
                    continue;

                Vector3 edgeCenter = new Vector3(
                    buildOriginX + (x + 0.5f) * step,
                    yStart + y * step,
                    buildOriginZ + z * step
                );

                if (!IsInsideTileXZ(edgeCenter, tileOriginX, tileOriginZ,
tileSize))
                    continue;

                Vector3 normal = EstimateSurfaceNormal(generator, edgeCenter,
step);
                AddOrientedQuad(meshData, v0, v1, v2, v3, normal);
            }
        }
    }
}

private static void BuildFacesY(
    VoxelWorldGenerator generator,
    MeshData meshData,
    float[, ,] densities,
    int[, ,] vertexIndices,

```

```

int buildOriginX,
int buildOriginZ,
int tileOriginX,
int tileOriginZ,
int tileSize,
int step,
int cellsX,
int cellsY,
int cellsZ,
int yStart)
{
    for (int x = 1; x < cellsX; x++)
    {
        for (int y = 0; y < cellsY; y++)
        {
            for (int z = 1; z < cellsZ; z++)
            {
                float a = densities[x, y, z];
                float b = densities[x, y + 1, z];

                if (!CrossesSurface(a, b))
                    continue;

                int v0 = vertexIndices[x - 1, y, z - 1];
                int v1 = vertexIndices[x - 1, y, z];
                int v2 = vertexIndices[x, y, z];
                int v3 = vertexIndices[x, y, z - 1];

                if (AnyInvalid(v0, v1, v2, v3))
                    continue;

                Vector3 edgeCenter = new Vector3(
                    buildOriginX + x * step,
                    yStart + (y + 0.5f) * step,
                    buildOriginZ + z * step
                );

                if (!IsInsideTileXZ(edgeCenter, tileOriginX, tileOriginZ,
tileSize))
                    continue;

                Vector3 normal = EstimateSurfaceNormal(generator, edgeCenter,
step);
                AddOrientedQuad(meshData, v0, v1, v2, v3, normal);
            }
        }
    }

    private static void BuildFacesZ(
        VoxelWorldGenerator generator,
        MeshData meshData,
        float[, ,] densities,
        int[, ,] vertexIndices,
        int buildOriginX,
        int buildOriginZ,
        int tileOriginX,
        int tileOriginZ,
        int tileSize,
        int step,
        int cellsX,
        int cellsY,
        int cellsZ,
        int yStart)
    {
        for (int x = 1; x < cellsX; x++)
        {
            for (int y = 1; y < cellsY; y++)
            {
                for (int z = 0; z < cellsZ; z++)

```

```

        {
            float a = densities[x, y, z];
            float b = densities[x, y, z + 1];

            if (!CrossesSurface(a, b))
                continue;

            int v0 = vertexIndices[x - 1, y - 1, z];
            int v1 = vertexIndices[x, y - 1, z];
            int v2 = vertexIndices[x, y, z];
            int v3 = vertexIndices[x - 1, y, z];

            if (AnyInvalid(v0, v1, v2, v3))
                continue;

            if (!IsInsideTileXZ(edgeCenter, tileOriginX, tileOriginZ,
tileSize))
                continue;

            Vector3 normal = EstimateSurfaceNormal(generator, edgeCenter,
step);
            AddOrientedQuad(meshData, v0, v1, v2, v3, normal);
        }
    }
}

private static bool IsInsideTileXZ(Vector3 p, int originX, int originZ, int
tileSize)
{
    const float eps = 0.01f;

    return p.x >= originX - eps &&
        p.x <= originX + tileSize + eps &&
        p.z >= originZ - eps &&
        p.z <= originZ + tileSize + eps;
}

private static Vector3 EstimateHeightNormal(VoxelWorldGenerator generator, Vector3
p, int step)
{
    float e = Mathf.Max(1f, step);

    float hL = generator.SampleHeight(p.x - e, p.z);
    float hR = generator.SampleHeight(p.x + e, p.z);
    float hD = generator.SampleHeight(p.x, p.z - e);
    float hU = generator.SampleHeight(p.x, p.z + e);

    float dhdx = (hR - hL) / (2f * e);
    float dhdz = (hU - hD) / (2f * e);

    Vector3 normal = new Vector3(-dhdx, 1f, -dhdz);

    if (normal.sqrMagnitude < 0.0001f)
        return Vector3.up;

    return normal.normalized;
}

private static Vector3 EstimateDensityNormal(VoxelWorldGenerator generator, Vector3
p, int step)
{
    float e = Mathf.Max(1f, step * 0.5f);

```

```

float dx =
    generator.SampleDensity(p.x + e, p.y, p.z) -
    generator.SampleDensity(p.x - e, p.y, p.z);

float dy =
    generator.SampleDensity(p.x, p.y + e, p.z) -
    generator.SampleDensity(p.x, p.y - e, p.z);

float dz =
    generator.SampleDensity(p.x, p.y, p.z + e) -
    generator.SampleDensity(p.x, p.y, p.z - e);

Vector3 gradient = new Vector3(dx, dy, dz);

if (gradient.sqrMagnitude < 0.000001f)
    return Vector3.up;
}

private static bool IsNearBrushOperation(VoxelWorldGenerator generator, Vector3 p,
float padding)
{
    if (generator == null)
        return false;

    BrushOperation[] ops = generator.GetBrushOperationsSnapshot();

    if (ops == null || ops.Length == 0)
        return false;

    for (int i = 0; i < ops.Length; i++)
    {
        BrushOperation op = ops[i];

        float r = Mathf.Max(0.1f, op.radius) + padding;
        float rSq = r * r;

        if ((p - op.position).sqrMagnitude <= rSq)
            return true;
    }

    return false;
}

private static void AddOrientedQuad(MeshData meshData, int a, int b, int c, int d,
Vector3 desiredNormal)
{
    Vector3 va = meshData.Vertices[a];
    Vector3 vb = meshData.Vertices[b];
    Vector3 vc = meshData.Vertices[c];

    Vector3 currentNormal = Vector3.Cross(vb - va, vc - va).normalized;
    bool flip = Vector3.Dot(currentNormal, desiredNormal) < 0f;

    meshData.AddQuad(a, b, c, d, flip);
}

private static bool CrossesSurface(float a, float b)
{
    a = NormalizeDensityForMeshing(a);
    b = NormalizeDensityForMeshing(b);

```

```

        return (a > 0f && b < 0f) || (a < 0f && b > 0f);
    }
    private const float DensityEpsilon = 0.0001f;

    private static float NormalizeDensityForMeshing(float d)
    {
        if (Mathf.Abs(d) < DensityEpsilon)
            return DensityEpsilon;

        return d;
    }

    private static bool AnyInvalid(int a, int b, int c, int d)
    {
        return a < 0 || b < 0 || c < 0 || d < 0;
    }
}

```

Скрипт MeshData.cs

```

using System.Collections.Generic;
using UnityEngine;

namespace VoxelWorldGenerator.Core
{
    public class MeshData
    {
        public readonly List<Vector3> Vertices = new List<Vector3>();
        public readonly List<int> Triangles = new List<int>();
        public readonly List<Vector2> UVs = new List<Vector2>();
        public readonly List<Color> Colors = new List<Color>();

        public int AddVertex(Vector3 position, Vector2 uv)
        {
            int index = Vertices.Count;

            Vertices.Add(position);
            UVs.Add(uv);
            Colors.Add(Color.white);

            return index;
        }

        public void AddTriangle(int a, int b, int c)
        {
            Triangles.Add(a);
            Triangles.Add(b);
            Triangles.Add(c);
        }

        public void AddQuad(int a, int b, int c, int d, bool flip)
        {
            if (!flip)
            {
                AddTriangle(a, b, c);
                AddTriangle(a, c, d);
            }
            else
            {
                AddTriangle(a, c, b);
                AddTriangle(a, d, c);
            }
        }

        public Mesh ToMesh(string meshName)
        {

```

```

        Mesh mesh = new Mesh();
        mesh.name = meshName;

        if (Vertices.Count > 65000)
            mesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32;

        mesh.SetVertices(Vertices);
        mesh.SetTriangles(Triangles, 0);

        if (UVs.Count == Vertices.Count)
            mesh.SetUVs(0, UVs);

        if (Colors.Count == Vertices.Count)
            mesh.SetColors(Colors);

        mesh.RecalculateNormals();
        mesh.RecalculateTangents();
        mesh.RecalculateBounds();

        return mesh;
    }
}

```

Скрипт TileKey.cs

```

using System;

namespace VoxelWorldGenerator.Core
{
    [Serializable]
    public struct TileKey : IEquatable<TileKey>
    {
        public int X;
        public int Z;
        public int Size;
        public int Lod;

        public TileKey(int x, int z, int size, int lod)
        {
            X = x;
            Z = z;
            Size = size;
            Lod = lod;
        }

        public bool Equals(TileKey other)
        {
            return X == other.X && Z == other.Z && Size == other.Size && Lod == other.Lod;
        }

        public override bool Equals(object obj)
        {
            return obj is TileKey other && Equals(other);
        }

        public override int GetHashCode()
        {
            unchecked
            {
                int hash = 17;
                hash = hash * 31 + X;
                hash = hash * 31 + Z;
                hash = hash * 31 + Size;
                hash = hash * 31 + Lod;
                return hash;
            }
        }
    }
}

```

```
public override string ToString()
{
    return $"Tile X:{X} Z:{Z} Size:{Size} LOD:{Lod}";
}
}
```


Додаток Д

Код програмного модуля quadtree-стрімінгу

Скрипт QuadtreeTerrainStreamer.cs

```

using System.Collections.Generic;
using VoxelWorldGenerator.Core;
using VoxelWorldGenerator.Generation;
using UnityEngine;

namespace VoxelWorldGenerator.Streaming
{
    public class QuadtreeTerrainStreamer : MonoBehaviour
    {
        [Header("References")]
        public VoxelWorldGenerator generator;
        public Transform viewer;

        [Header("World / Quadtree")]
        public int rootSize = 2048;
        public int minTileSize = 128;
        public int maxLod = 4;

        [Header("LOD Distances")]
        public float lod0Distance = 220f;
        public float lod1Distance = 430f;
        public float lod2Distance = 760f;
        public float visibleDistance = 1200f;

        [Header("Meshing")]
        public int baseStep = 2;
        public bool generateCollidersNearViewer = true;
        public float colliderDistance = 260f;

        [Header("Seam Fix")]
        public bool enableLodBalancing = true;

        [Header("Runtime")]
        public bool rebuildEveryFrame = false;
        public float updateInterval = 0.35f;
        public bool clearOldTiles = true;

        [Header("Debug")]
        public bool drawDebug = true;
        public bool logBuilds = false;

        private readonly Dictionary<TileKey, TerrainTile> activeTiles = new
Dictionary<TileKey, TerrainTile>();
        private readonly List<TileKey> desiredTiles = new List<TileKey>();
        private readonly HashSet<TileKey> desiredSet = new HashSet<TileKey>();
        private readonly List<TileKey> removeBuffer = new List<TileKey>();

        [Header("Build Budget")]
        public int maxTileBuildStartsPerFrame = 1;
        public int maxConcurrentTileBuilds = 1;
        public bool keepOldTilesUntilBuildQueueEmpty = true;

        private readonly Queue<TileKey> buildQueue = new Queue<TileKey>();
        private readonly HashSet<TileKey> queuedSet = new HashSet<TileKey>();
        private readonly HashSet<TileKey> buildingSet = new HashSet<TileKey>();

        private Transform tileRoot;
        private float nextUpdateTime;
        private Vector3 lastViewerPosition;
        private bool hasBuiltOnce;
    }
}

```

```

public int ActiveTileCount => activeTiles.Count;

private void Reset()
{
    generator = GetComponent<VoxelWorldGenerator>();
}

private void Start()
{
    if (generator == null)
        generator = GetComponent<VoxelWorldGenerator>();

    if (viewer == null && Camera.main != null)
        viewer = Camera.main.transform;

    if (generator != null && generator.settings != null)
    {
        rootSize = generator.settings.worldSize;
        minTileSize = generator.settings.defaultTileSize;
    }

    EnsureTileRoot();
    ForceRebuild();
}

private void Update()
{
    ProcessBuildQueue();

    if (viewer == null)
    {
        if (Camera.main != null)
            viewer = Camera.main.transform;

        if (viewer == null)
            return;
    }

    if (rebuildEveryFrame)
    {
        ForceRebuild();
        return;
    }

    if (Time.time < nextUpdateTime)
        return;

    float moved = Vector3.Distance(lastViewerPosition, viewer.position);

    if (!hasBuiltOnce || moved > minTileSize * 1.0f)
    {
        ForceRebuild();
        nextUpdateTime = Time.time + updateInterval;
    }
}

[ContextMenu("Force Rebuild")]
public void ForceRebuild()
{
    if (generator == null)
    {
        generator = GetComponent<VoxelWorldGenerator>();
        if (generator == null)
            generator = FindObjectOfType<VoxelWorldGenerator>();
    }
}

```

```

        if (generator == null || generator.settings == null)
        {
            Debug.LogError("QuadtreeTerrainStreamer: generator or settings is
missing.");
            return;
        }

        if (viewer == null && Camera.main != null)
            viewer = Camera.main.transform;

        if (viewer == null)
        {
            Debug.LogError("QuadtreeTerrainStreamer: viewer is missing.");
            return;
        }

        EnsureTileRoot();

        desiredTiles.Clear();
        desiredSet.Clear();

        int safeRootSize = Mathf.ClosestPowerOfTwo(rootSize);
        rootSize = safeRootSize;

        BuildDesiredRecursive(0, 0, rootSize, maxLod);

        if (enableLodBalancing)
            ApplySimpleLodBalancing();

        SortDesiredTilesForBuild();

        ApplyDesiredTiles();

        lastViewerPosition = viewer.position;
        hasBuiltOnce = true;
    }
    private void SortDesiredTilesForBuild()
    {
        if (viewer == null)
            return;

        desiredTiles.Sort((a, b) =>
        {
            float da = DistanceToTileXZ(a.X, a.Z, a.Size, viewer.position);
            float db = DistanceToTileXZ(b.X, b.Z, b.Size, viewer.position);

            int distanceCompare = da.CompareTo(db);

            if (distanceCompare != 0)
                return distanceCompare;

        });
    }
    private void BuildDesiredRecursive(int x, int z, int size, int lod)
    {
        if (!TileIntersectsVisibleCircle(x, z, size))
            return;

        bool shouldSplit = ShouldSplit(x, z, size, lod);

        if (shouldSplit)
        {
            int half = size / 2;

```

```

        BuildDesiredRecursive(x, z, half, lod - 1);
        BuildDesiredRecursive(x + half, z, half, lod - 1);
        BuildDesiredRecursive(x, z + half, half, lod - 1);
        BuildDesiredRecursive(x + half, z + half, half, lod - 1);
    }
    else
    {
        TileKey key = new TileKey(x, z, size, lod);

        if (!desiredSet.Contains(key))
        {
            desiredTiles.Add(key);
            desiredSet.Add(key);
        }
    }
}

private bool ShouldSplit(int x, int z, int size, int lod)
{
    if (size <= minTileSize)
        return false;

    if (lod <= 0)
        return false;

    float distance = DistanceToTileXZ(x, z, size, viewer.position);

    if (distance < lod0Distance)
        return true;

    if (size > minTileSize * 2 && distance < lod1Distance)
        return true;

    if (size > minTileSize * 4 && distance < lod2Distance)
        return true;

    return false;
}

private bool TileIntersectsVisibleCircle(int x, int z, int size)
{
    float distance = DistanceToTileXZ(x, z, size, viewer.position);
    return distance <= visibleDistance + size * 0.75f;
}

private float DistanceToTileXZ(int x, int z, int size, Vector3 position)
{
    float minX = x;
    float maxX = x + size;
    float minZ = z;
    float maxZ = z + size;

    float px = Mathf.Clamp(position.x, minX, maxX);
    float pz = Mathf.Clamp(position.z, minZ, maxZ);

    float dx = position.x - px;
    float dz = position.z - pz;

    return Mathf.Sqrt(dx * dx + dz * dz);
}

private void ApplySimpleLodBalancing()
{
    for (int pass = 0; pass < 4; pass++)

```

```

{
    bool changed = false;

    List<TileKey> current = new List<TileKey>(desiredTiles);

    for (int i = 0; i < current.Count; i++)
    {
        TileKey a = current[i];

        for (int j = 0; j < current.Count; j++)
        {
            if (i == j)
                continue;

            TileKey b = current[j];

            if (!AreTilesTouching(a, b))
                continue;

            int lodDiff = Mathf.Abs(a.Lod - b.Lod);

            if (lodDiff <= 1)
                continue;

            TileKey coarse = a.Lod > b.Lod ? a : b;

            if (coarse.Size <= minTileSize)
                continue;

            SplitDesiredTile(coarse);
            changed = true;
            break;
        }

        if (changed)
            break;
    }

    if (!changed)
        break;
}

private bool AreTilesTouching(TileKey a, TileKey b)
{
    int aMinX = a.X;
    int aMaxX = a.X + a.Size;
    int aMinZ = a.Z;
    int aMaxZ = a.Z + a.Size;

    int bMinX = b.X;
    int bMaxX = b.X + b.Size;
    int bMinZ = b.Z;
    int bMaxZ = b.Z + b.Size;

    bool overlapX = aMinX < bMaxX && aMaxX > bMinX;
    bool overlapZ = aMinZ < bMaxZ && aMaxZ > bMinZ;

    bool touchesX = aMaxX == bMinX || bMaxX == aMinX;
    bool touchesZ = aMaxZ == bMinZ || bMaxZ == aMinZ;

    return (touchesX && overlapZ) || (touchesZ && overlapX);
}

```

```

private void SplitDesiredTile(TileKey tile)
{
    if (!desiredSet.Contains(tile))
        return;

    desiredSet.Remove(tile);
    desiredTiles.Remove(tile);

    int half = tile.Size / 2;
    int childLod = tile.Lod - 1;

    AddDesiredTile(new TileKey(tile.X, tile.Z, half, childLod));
    AddDesiredTile(new TileKey(tile.X + half, tile.Z, half, childLod));
    AddDesiredTile(new TileKey(tile.X, tile.Z + half, half, childLod));
    AddDesiredTile(new TileKey(tile.X + half, tile.Z + half, half, childLod));
}

private void AddDesiredTile(TileKey key)
{
    if (desiredSet.Contains(key))
        return;

    desiredSet.Add(key);
    desiredTiles.Add(key);
}

private void ApplyDesiredTiles()
{
    for (int i = 0; i < desiredTiles.Count; i++)
    {
        TileKey key = desiredTiles[i];

        if (activeTiles.ContainsKey(key))
            continue;

        if (queuedSet.Contains(key))
            continue;

        if (buildingSet.Contains(key))
            continue;

        buildQueue.Enqueue(key);
        queuedSet.Add(key);
    }
}

private void CreateTile(TileKey key)
{
    GameObject go = new
GameObject($"Tile_X{key.X}_Z{key.Z}_S{key.Size}_LOD{key.Lod}");
    go.transform.SetParent(tileRoot, false);

    TerrainTile tile = go.AddComponent<TerrainTile>();

    int step = StepForLod(key.Lod);
    bool needCollider = ShouldGenerateCollider(key);

    tile.Initialize(
        generator,
        key,
        step,
        needCollider
    );
}

```

```

        tile.Rebuild();

        activeTiles.Add(key, tile);

        if (logBuilds)
            Debug.Log($"Built {key} step {step}, collider {needCollider}");
    }
    private void StartCreateTileAsync(TileKey key)
    {
        buildingSet.Add(key);

        GameObject go = new
        GameObject($"Tile_X{key.X}_Z{key.Z}_S{key.Size}_LOD{key.Lod}_BUILDING");
        go.transform.SetParent(tileRoot, false);

        TerrainTile tile = go.AddComponent<TerrainTile>();

        int step = StepForLod(key.Lod);
        bool needCollider = ShouldGenerateCollider(key);

        tile.Initialize(
            generator,
            key,
            step,
            needCollider
        );

        go.SetActive(false);

        tile.RebuildAsync(completedTile =>
        {
            buildingSet.Remove(key);

            if (completedTile == null)
                return;

            if (!desiredSet.Contains(key))
            {
                if (completedTile.gameObject != null)
                    Destroy(completedTile.gameObject);

                return;
            }

            completedTile.gameObject.name =
            $"Tile_X{key.X}_Z{key.Z}_S{key.Size}_LOD{key.Lod}";
            completedTile.gameObject.SetActive(true);

            if (!activeTiles.ContainsKey(key))
                activeTiles.Add(key, completedTile);

            if (logBuilds)
                Debug.Log($"Async built {key}, step {step}, collider {needCollider}");
        });
    }

    private int StepForLod(int lod)
    {
        int step = baseStep * Mathf.RoundToInt(Mathf.Pow(2, Mathf.Max(0, lod)));
        return Mathf.Max(1, step);
    }

    private bool ShouldGenerateCollider(TileKey key)

```

```

{
    if (!generateCollidersNearViewer)
        return false;

    float distance = DistanceToTileXZ(key.X, key.Z, key.Size, viewer.position);
    return distance <= colliderDistance;
}

private void EnsureTileRoot()
{
    if (tileRoot != null)
        return;

    Transform existing = transform.Find("Runtime Tiles");

    if (existing != null)
    {
        tileRoot = existing;
        return;
    }

    GameObject root = new GameObject("Runtime Tiles");
    root.transform.SetParent(transform, false);
    tileRoot = root.transform;
}

public List<TerrainTile> GetActiveTilesSnapshot()
{
    List<TerrainTile> tiles = new List<TerrainTile>();

    foreach (KeyValuePair<TileKey, TerrainTile> pair in activeTiles)
    {
        if (buildingSet.Contains(pair.Key))
            continue;

        if (!desiredSet.Contains(pair.Key))
            removeBuffer.Add(pair.Key);
    }
    return tiles;
}

public void RebuildTilesInBounds(float minX, float minZ, float maxX, float maxZ)
{
    foreach (KeyValuePair<TileKey, TerrainTile> pair in activeTiles)
    {
        TerrainTile tile = pair.Value;

        if (tile == null)
            continue;

        if (tile.IsBuilding)
            continue;

        if (tile.IntersectsXZBounds(minX, minZ, maxX, maxZ))
            tile.RebuildAsync();
    }
}

private void OnDrawGizmos()
{
    if (!drawDebug)
        return;

    Gizmos.color = Color.white;
}

```



```

Gizmos.DrawWireCube(
    new Vector3(rootSize * 0.5f, 1f, rootSize * 0.5f),
    new Vector3(rootSize, 2f, rootSize)
);

if (viewer != null)
{
    DrawCircle(viewer.position, lod0Distance, Color.green);
    DrawCircle(viewer.position, lod1Distance, Color.yellow);
    DrawCircle(viewer.position, lod2Distance, new Color(1f, 0.5f, 0f));
    DrawCircle(viewer.position, visibleDistance, Color.red);
}

if (activeTiles == null)
    return;

foreach (KeyValuePair<TileKey, TerrainTile> pair in activeTiles)
{
    TileKey key = pair.Key;

    Color c = Color.white;

    if (key.Lod == 0) c = Color.green;
    else if (key.Lod == 1) c = Color.yellow;
    else if (key.Lod == 2) c = new Color(1f, 0.5f, 0f);
    else c = Color.red;

    Gizmos.color = c;

    Vector3 center = new Vector3(
        key.X + key.Size * 0.5f,
        3f + key.Lod * 2f,
        key.Z + key.Size * 0.5f
    );

    Vector3 size = new Vector3(key.Size, 1f, key.Size);
    Gizmos.DrawWireCube(center, size);
}

private void DrawCircle(Vector3 center, float radius, Color color)
{
    Gizmos.color = color;

    const int segments = 64;
    Vector3 previous = center + new Vector3(radius, 0f, 0f);

    for (int i = 1; i <= segments; i++)
    {
        float angle = i / (float)segments * Mathf.PI * 2f;
        Vector3 next = center + new Vector3(Mathf.Cos(angle) * radius, 0f,
Mathf.Sin(angle) * radius);
        Gizmos.DrawLine(previous, next);
        previous = next;
    }
}

private void ProcessBuildQueue()
{
    int startsThisFrame = 0;

    while (buildQueue.Count > 0 &&
        startsThisFrame < maxTileBuildStartsPerFrame &&
        buildingSet.Count < maxConcurrentTileBuilds)
    {
        TileKey key = buildQueue.Dequeue();
        queuedSet.Remove(key);
    }
}

```

```

        if (!desiredSet.Contains(key))
            continue;

        if (activeTiles.ContainsKey(key))
            continue;

        if (buildingSet.Contains(key))
            continue;

        StartCreateTileAsync(key);
        startsThisFrame++;
    }

    if (buildQueue.Count == 0 &&
        buildingSet.Count == 0 &&
        clearOldTiles &&
        keepOldTilesUntilBuildQueueEmpty)
    {
        CleanupTilesNotInDesiredSet();
    }
}

private void CleanupTilesNotInDesiredSet()
{
    removeBuffer.Clear();

    foreach (KeyValuePair<TileKey, TerrainTile> pair in activeTiles)
    {
        if (!desiredSet.Contains(pair.Key))
            removeBuffer.Add(pair.Key);
    }

    for (int i = 0; i < removeBuffer.Count; i++)
    {
        TileKey key = removeBuffer[i];

        if (activeTiles.TryGetValue(key, out TerrainTile tile))
        {
            if (tile != null)
                Destroy(tile.gameObject);

            activeTiles.Remove(key);
        }
    }
}
}

```

Скрипт TerrainTile.cs

```

using System;
using System.Threading.Tasks;
using VoxelWorldGenerator.Core;
using VoxelWorldGenerator.Generation;
using VoxelWorldGenerator.Meshing;
using UnityEngine;

namespace VoxelWorldGenerator.Streaming
{
    [RequireComponent(typeof(MeshFilter))]
    [RequireComponent(typeof(MeshRenderer))]
    public class TerrainTile : MonoBehaviour
    {
        [Header("References")]
        public VoxelWorldGenerator generator;

        [Header("Tile")]
    }
}

```

```

public int originX = 0;
public int originZ = 0;
public int tileSize = 128;
public int step = 4;
public int lod = 0;

[Header("Collision")]
public bool generateCollider = false;

[Header("Debug")]
public bool buildOnStart = false;
public bool rebuildInEditor = false;
public bool drawBounds = false;

private MeshFilter meshFilter;
private MeshRenderer meshRenderer;
private MeshCollider meshCollider;

private GameObject waterObject;
private MeshFilter waterMeshFilter;
private MeshRenderer waterMeshRenderer;

private static Material fallbackTerrainMaterial;
private static Material fallbackWaterMaterial;

private int buildVersion;

public TileKey Key { get; private set; }
public bool IsBuilt { get; private set; }
public bool IsBuilding { get; private set; }
public int VertexCount { get; private set; }
public int TriangleCount { get; private set; }

private void Awake()
{
    EnsureComponents();
}

private void Start()
{
    if (buildOnStart)
        Rebuild();
}

private void OnValidate()
{
    tileSize = Mathf.Max(8, tileSize);
    step = Mathf.Max(1, step);

    if (rebuildInEditor && Application.isPlaying)
        Rebuild();
}

public void Initialize(
    VoxelWorldGenerator newGenerator,
    TileKey newKey,
    int newStep,
    bool newGenerateCollider)
{
    generator = newGenerator;

    Key = newKey;
    originX = newKey.X;
    originZ = newKey.Z;
    tileSize = newKey.Size;
    lod = newKey.Lod;
}

```

```

        step = Mathf.Max(1, newStep);
        generateCollider = newGenerateCollider;

        transform.position = Vector3.zero;
        gameObject.name =
        $"Tile_X{originX}_Z{originZ}_S{tileSize}_LOD{lod}_Step{step}";
    }

    [ContextMenu("Rebuild Tile Sync")]
    public void Rebuild()
    {
        EnsureComponents();

        IsBuilt = false;
        IsBuilding = true;
        buildVersion++;

        if (!ValidateReferences())
        {
            IsBuilding = false;
            return;
        }

        int worldHeight = generator.settings.worldHeight;

        MeshData meshData = SurfaceNetsMesher.Build(
            generator,
            originX,
            originZ,
            tileSize,
            worldHeight,
            step
        );

        ApplyMeshData(meshData);
        IsBuilding = false;
    }

    public async void RebuildAsync(Action<TerrainTile> onComplete = null)
    {
        EnsureComponents();

        IsBuilt = false;
        IsBuilding = true;

        int localVersion = ++buildVersion;

        if (!ValidateReferences())
        {
            IsBuilding = false;
            return;
        }

        VoxelWorldGenerator capturedGenerator = generator;
        int capturedOriginX = originX;
        int capturedOriginZ = originZ;
        int capturedTileSize = tileSize;
        int capturedWorldHeight = generator.settings.worldHeight;
        int capturedStep = step;

        MeshData meshData = null;
        Exception error = null;

        await Task.Run(() =>

```

```

{
    try
    {
        meshData = SurfaceNetsMesher.Build(
            capturedGenerator,
            capturedOriginX,
            capturedOriginZ,
            capturedTileSize,
            capturedWorldHeight,
            capturedStep
        );
    }
    catch (Exception e)
    {
        error = e;
    }
});

if (this == null || gameObject == null)
    return;

if (localVersion != buildVersion)
    return;

if (error != null)
{
    Debug.LogException(error);
    IsBuilding = false;
    return;
}

ApplyMeshData(meshData);

IsBuilding = false;
onComplete?.Invoke(this);
}

private void ApplyMeshData(MeshData meshData)
{
    Mesh mesh = meshData.ToMesh($"TerrainTile_{originX}_{originZ}_LOD{lod}");

    ApplySurfaceColors(mesh);

    meshFilter.sharedMesh = mesh;
    meshRenderer.sharedMaterial = GetTerrainMaterial();

    if (generateCollider)
    {
        if (meshCollider == null)
            meshCollider = gameObject.GetComponent<MeshCollider>();

        if (meshCollider == null)
            meshCollider = gameObject.AddComponent<MeshCollider>();

        meshCollider.sharedMesh = null;
        meshCollider.sharedMesh = mesh;
    }
    else
    {
        if (meshCollider != null)
        {
            if (Application.isPlaying)
                Destroy(meshCollider);
            else
                DestroyImmediate(meshCollider);

            meshCollider = null;
        }
    }
}

```

```

    }
}

DestroyWaterPlane();

VertexCount = mesh.vertexCount;
TriangleCount = mesh.triangles.Length / 3;
IsBuilt = true;
}

private bool ValidateReferences()
{
    if (generator == null)
        generator = FindObjectOfType<VoxelWorldGenerator>();

    if (generator == null)
    {
        Debug.LogError("TerrainTile: VoxelWorldGenerator not found.");
        return false;
    }

    if (generator.settings == null)
    {
        Debug.LogError("TerrainTile: generator.settings is null.");
        return false;
    }

    return true;
}

public bool ContainsXZ(Vector3 worldPosition)
{
    return worldPosition.x >= originX &&
        worldPosition.x <= originX + tileSize &&
        worldPosition.z >= originZ &&
        worldPosition.z <= originZ + tileSize;
}

public bool IntersectsXZBounds(float minX, float minZ, float maxX, float maxZ)
{
    float tileMinX = originX;
    float tileMinZ = originZ;
    float tileMaxX = originX + tileSize;
    float tileMaxZ = originZ + tileSize;

    return tileMinX <= maxX &&
        tileMaxX >= minX &&
        tileMinZ <= maxZ &&
        tileMaxZ >= minZ;
}

public Vector2 CenterXZ()
{
    return new Vector2(originX + tileSize * 0.5f, originZ + tileSize * 0.5f);
}

private void ApplySurfaceColors(Mesh mesh)
{
    Vector3[] vertices = mesh.vertices;
    Vector3[] normals = mesh.normals;

    Color[] colors = new Color[vertices.Length];

    for (int i = 0; i < vertices.Length; i++)
    {
        colors[i] = generator.EvaluateSurfaceColor(vertices[i], normals[i]);
    }
}

```

```

    }

    mesh.colors = colors;
}

private Material GetTerrainMaterial()
{
    if (generator.settings.terrainMaterial != null)
        return generator.settings.terrainMaterial;

    if (fallbackTerrainMaterial != null)
        return fallbackTerrainMaterial;

    Shader shader = Shader.Find("VoxelWorldGenerator/URP Vertex Color Terrain");

    if (shader == null)
    {
        Debug.LogWarning("Shader VoxelWorldGenerator/URP Vertex Color Terrain not found. Falling back to Universal Render Pipeline/Lit material.");
        shader = Shader.Find("Universal Render Pipeline/Lit");
    }

    if (shader == null)
        shader = Shader.Find("Standard");

    fallbackTerrainMaterial = new Material(shader);
    fallbackTerrainMaterial.name = "Runtime Voxel Terrain Material";

    if (fallbackTerrainMaterial.HasProperty("_Brightness"))
        fallbackTerrainMaterial.SetFloat("_Brightness", 1f);

    return fallbackTerrainMaterial;
}

private Material GetWaterMaterial()
{
    if (generator.settings.waterMaterial != null)
        return generator.settings.waterMaterial;

    if (fallbackWaterMaterial != null)
        return fallbackWaterMaterial;

    Shader shader = Shader.Find("VoxelWorldGenerator/URP Simple Water");

    if (shader == null)
    {
        Debug.LogWarning("Shader VoxelWorldGenerator/URP Simple Water not found. Falling back to Universal Render Pipeline/Lit material.");
        shader = Shader.Find("Universal Render Pipeline/Lit");
    }

    if (shader == null)
        shader = Shader.Find("Standard");

    fallbackWaterMaterial = new Material(shader);
    fallbackWaterMaterial.name = "Runtime Voxel Water Material";

    if (fallbackWaterMaterial.HasProperty("_Color"))
        fallbackWaterMaterial.SetColor("_Color", generator.settings.waterColor);

    if (fallbackWaterMaterial.HasProperty("_Alpha"))
        fallbackWaterMaterial.SetFloat("_Alpha", generator.settings.waterColor.a);
}

```

```

        return fallbackWaterMaterial;
    }

private void BuildWaterPlane()
{
    EnsureWaterObject();

    float y = generator.settings.waterLevel;

    Mesh mesh = new Mesh();
    mesh.name = $"Water_{originX}_{originZ}_{tileSize}";

    Vector3[] vertices =
    {
        new Vector3(originX, y, originZ),
        new Vector3(originX + tileSize, y, originZ),
        new Vector3(originX + tileSize, y, originZ + tileSize),
        new Vector3(originX, y, originZ + tileSize)
    };

    int[] triangles =
    {
        0, 2, 1,
        0, 3, 2
    };

    Vector2[] uvs =
    {
        new Vector2(0f, 0f),
        new Vector2(1f, 0f),
        new Vector2(1f, 1f),
        new Vector2(0f, 1f)
    };

    mesh.vertices = vertices;
    mesh.triangles = triangles;
    mesh.uv = uvs;
    mesh.RecalculateNormals();
    mesh.RecalculateBounds();

    waterMeshFilter.sharedMesh = mesh;
    waterMeshRenderer.sharedMaterial = GetWaterMaterial();
}

private void DestroyWaterPlane()
{
    if (waterObject == null)
        return;

    if (Application.isPlaying)
        Destroy(waterObject);
    else
        DestroyImmediate(waterObject);

    waterObject = null;
    waterMeshFilter = null;
    waterMeshRenderer = null;
}

private void EnsureWaterObject()
{
    if (waterObject != null)
        return;

    Transform existing = transform.Find("Water");
    if (existing != null)

```



```

        {
            waterObject = existing.gameObject;
        }
        else
        {
            waterObject = new GameObject("Water");
            waterObject.transform.SetParent(transform, false);
        }

        waterMeshFilter = waterObject.GetComponent<MeshFilter>();
        if (waterMeshFilter == null)
            waterMeshFilter = waterObject.AddComponent<MeshFilter>();

        waterMeshRenderer = waterObject.GetComponent<MeshRenderer>();
        if (waterMeshRenderer == null)
            waterMeshRenderer = waterObject.AddComponent<MeshRenderer>();
    }

    private void EnsureComponents()
    {
        if (meshFilter == null)
            meshFilter = GetComponent<MeshFilter>();

        if (meshRenderer == null)
            meshRenderer = GetComponent<MeshRenderer>();
    }

    private void OnDrawGizmosSelected()
    {
        if (!drawBounds)
            return;

        Gizmos.color = Color.cyan;

        Vector3 center = new Vector3(
            originX + tileSize * 0.5f,
            64f,
            originZ + tileSize * 0.5f
        );

        Vector3 size = new Vector3(tileSize, 128f, tileSize);
        Gizmos.DrawWireCube(center, size);
    }
}

```

Додаток Е

Код програмного модуля редагування ландшафту

Скрипт VoxelTerrainEditor.cs

```

using VoxelWorldGenerator.Generation;
using VoxelWorldGenerator.Streaming;
using UnityEngine;
using UnityEngine.InputSystem;

namespace VoxelWorldGenerator.Editing
{
    public class VoxelTerrainEditor : MonoBehaviour
    {
        [Header("References")]
        public Camera targetCamera;
        public VoxelWorldGenerator generator;
        public QuadtreeTerrainStreamer streamer;
        public BrushPreview brushPreview;

        [Header("Brush")]
        public float brushRadius = 12f;
        public float brushStrength = 8f;
        public float minBrushRadius = 2f;
        public float maxBrushRadius = 80f;
        public float minBrushStrength = 1f;
        public float maxBrushStrength = 80f;

        [Header("Input")]
        public bool editingEnabled = true;
        public float brushRadiusStep = 2f;
        public float brushStrengthStep = 2f;

        [Header("Raymarch")]
        public float maxRayDistance = 1500f;
        public float rayStep = 4f;
        public int binarySearchSteps = 8;

        [Header("Rebuild")]
        public float rebuildPadding = 8f;
        public float editCooldown = 0.08f;

        private Vector3 currentHitPoint;
        private bool hasHit;
        private float nextEditTime;

        public Vector3 CurrentHitPoint => currentHitPoint;
        public bool HasHit => hasHit;

        private Keyboard Keyboard => Keyboard.current;
        private Mouse Mouse => Mouse.current;

        private void Start()
        {
            if (targetCamera == null && Camera.main != null)
                targetCamera = Camera.main;

            if (generator == null)
                generator = FindObjectOfType<VoxelWorldGenerator>();

            if (streamer == null)
                streamer = FindObjectOfType<QuadtreeTerrainStreamer>();

            if (brushPreview == null)
                brushPreview = FindObjectOfType<BrushPreview>();
        }
    }
}

```

```

    }

    private void Update()
    {
        if (!editingEnabled)
            return;

        if (Keyboard == null || Mouse == null)
            return;

        HandleBrushSettings();
        UpdateHitPoint();
        UpdatePreview();
        HandleEditing();
    }

    private void HandleBrushSettings()
    {
        if (Keyboard.leftBracketKey.wasPressedThisFrame)
            brushRadius -= brushRadiusStep;

        if (Keyboard.rightBracketKey.wasPressedThisFrame)
            brushRadius += brushRadiusStep;

        Vector2 scroll = Mouse.scroll.ReadValue();

        if (Mathf.Abs(scroll.y) > 0.01f)
            brushStrength += Mathf.Sign(scroll.y) * brushStrengthStep;

        if (Keyboard.minusKey.wasPressedThisFrame)
            brushStrength -= brushStrengthStep;

        if (Keyboard.equalsKey.wasPressedThisFrame)
            brushStrength += brushStrengthStep;

        brushRadius = Mathf.Clamp(brushRadius, minBrushRadius, maxBrushRadius);
        brushStrength = Mathf.Clamp(brushStrength, minBrushStrength, maxBrushStrength);
    }

    private void UpdateHitPoint()
    {
        hasHit = false;

        if (targetCamera == null || generator == null)
            return;

        Ray ray = new Ray(targetCamera.transform.position,
targetCamera.transform.forward);

        if (TryRaymarchDensity(ray, out Vector3 hit))
        {
            currentHitPoint = hit;
            hasHit = true;
        }
    }

    private bool TryRaymarchDensity(Ray ray, out Vector3 hitPoint)
    {
        hitPoint = Vector3.zero;

        float previousDistance = 0f;
        Vector3 previousPoint = ray.origin;
        float previousDensity = generator.SampleDensity(previousPoint);
    }

```

```

for (float d = rayStep; d <= maxRayDistance; d += rayStep)
{
    Vector3 p = ray.origin + ray.direction * d;
    float density = generator.SampleDensity(p);

    bool crossed =
        previousDensity < 0f && density >= 0f ||
        previousDensity > 0f && density <= 0f;

    if (crossed)
    {
        Vector3 a = previousPoint;
        Vector3 b = p;

        for (int i = 0; i < binarySearchSteps; i++)
        {
            Vector3 mid = (a + b) * 0.5f;
            float midDensity = generator.SampleDensity(mid);

            if ((previousDensity < 0f && midDensity < 0f) ||
                (previousDensity > 0f && midDensity > 0f))
            {
                a = mid;
            }
            else
            {
                b = mid;
            }
        }

        hitPoint = (a + b) * 0.5f;
        return true;
    }

    previousDistance = d;
    previousPoint = p;
    previousDensity = density;
}

return false;
}

private void UpdatePreview()
{
    if (brushPreview == null)
        return;

    if (hasHit)
    {
        float previewMaxStrength = Mathf.Min(maxBrushStrength, 30f);
        float intensity01 = Mathf.InverseLerp(minBrushStrength, previewMaxStrength,
brushStrength);
        brushPreview.Show(currentHitPoint, brushRadius, intensity01);
    }
    else
    {
        brushPreview.Hide();
    }
}

private void HandleEditing()
{
    if (!hasHit)
        return;

    if (Time.time < nextEditTime)
        return;
}

```

```

    bool altPressed =
        Keyboard.leftAltKey.isPressed ||
        Keyboard.rightAltKey.isPressed;

    bool qPressed = Keyboard.qKey.isPressed;
    bool leftMousePressed = Mouse.leftButton.isPressed;

    bool smooth = qPressed && leftMousePressed;
    bool subtract = altPressed && leftMousePressed && !smooth;
    bool add = leftMousePressed && !altPressed && !smooth;

    if (!add && !subtract && !smooth)
        return;

    if (smooth)
        ApplySmoothBrush();
    else
        ApplyBrush(add);

    nextEditTime = Time.time + editCooldown;
}

private void ApplyBrush(bool addTerrain)
{
    if (generator == null)
        return;

    generator.AddBrushOperation(
        currentHitPoint,
        brushRadius,
        brushStrength,
        addTerrain
    );

    RebuildBrushArea();
}

private void ApplySmoothBrush()
{
    if (generator == null)
        return;

    generator.AddSmoothBrushOperation(
        currentHitPoint,
        brushRadius,
        brushStrength
    );

    RebuildBrushArea();
}

private void RebuildBrushArea()
{
    if (streamer == null)
        return;

    float r = brushRadius + rebuildPadding;

    streamer.RebuildTilesInBounds(
        currentHitPoint.x - r,
        currentHitPoint.z - r,
        currentHitPoint.x + r,
        currentHitPoint.z + r
    );
}
}
}

```

Скрипт BrushPreview.cs

```

using UnityEngine;

namespace VoxelWorldGenerator.Editing
{
    public class BrushPreview : MonoBehaviour
    {
        [Header("Visual")]
        public int segments = 96;
        public float lineWidth = 0.18f;
        public bool drawHorizontalRing = true;
        public bool drawVerticalRingX = true;
        public bool drawVerticalRingZ = true;

        [Header("Intensity")]
        public Color weakColor = Color.white;
        public Color mediumColor = Color.yellow;
        public Color strongColor = Color.red;

        [Header("Debug")]
        public bool visible;
        [Range(0f, 1f)] public float debugIntensity01;

        private Vector3 position;
        private float radius;
        private float intensity01;

        private LineRenderer horizontalRing;
        private LineRenderer verticalRingX;
        private LineRenderer verticalRingZ;

        private Material horizontalMaterial;
        private Material verticalXMaterial;
        private Material verticalZMaterial;

        private void Awake()
        {
            EnsureRenderers();
            Hide();
        }

        public void Show(Vector3 worldPosition, float brushRadius, float brushIntensity01)
        {
            EnsureRenderers();

            visible = true;
            position = worldPosition;
            radius = Mathf.Max(0.1f, brushRadius);
            intensity01 = Mathf.Clamp01(brushIntensity01);
            debugIntensity01 = intensity01;

            Color color = EvaluateColor(intensity01);
            color.a = 1f;

            UpdateMaterialColor(horizontalMaterial, color);
            UpdateMaterialColor(verticalXMaterial, color);
            UpdateMaterialColor(verticalZMaterial, color);

            UpdateRing(horizontalRing, RingPlane.XZ, drawHorizontalRing, color);
            UpdateRing(verticalRingX, RingPlane.YZ, drawVerticalRingX, color);
            UpdateRing(verticalRingZ, RingPlane.XY, drawVerticalRingZ, color);
        }

        public void Hide()
        {

```

```

        visible = false;

        if (horizontalRing != null) horizontalRing.enabled = false;
        if (verticalRingX != null) verticalRingX.enabled = false;
        if (verticalRingZ != null) verticalRingZ.enabled = false;
    }

    private void EnsureRenderers()
    {
        if (horizontalRing == null)
        {
            horizontalMaterial = CreateLineMaterial("Brush Preview Horizontal
Material");
            horizontalRing = CreateRingRenderer("Brush Preview Horizontal Ring",
horizontalMaterial);
        }

        if (verticalRingX == null)
        {
            verticalXMaterial = CreateLineMaterial("Brush Preview Vertical X
Material");
            verticalRingX = CreateRingRenderer("Brush Preview Vertical Ring X",
verticalXMaterial);
        }

        if (verticalRingZ == null)
        {
            verticalZMaterial = CreateLineMaterial("Brush Preview Vertical Z
Material");
            verticalRingZ = CreateRingRenderer("Brush Preview Vertical Ring Z",
verticalZMaterial);
        }
    }

    private LineRenderer CreateRingRenderer(string objectName, Material material)
    {
        GameObject go = new GameObject(objectName);
        go.transform.SetParent(transform, false);

        LineRenderer lr = go.AddComponent<LineRenderer>();

        lr.useWorldSpace = true;
        lr.loop = true;
        lr.positionCount = Mathf.Max(8, segments);
        lr.widthMultiplier = lineWidth;
        lr.material = material;
        lr.shadowCastingMode = UnityEngine.Rendering.ShadowCastingMode.Off;
        lr.receiveShadows = false;
        lr.enabled = false;

        return lr;
    }

    private Material CreateLineMaterial(string materialName)
    {
        Shader shader = Shader.Find("Sprites/Default");

        if (shader == null)
            shader = Shader.Find("Universal Render Pipeline/Unlit");

        if (shader == null)
            shader = Shader.Find("Unlit/Color");

        if (shader == null)
            shader = Shader.Find("Standard");
    }

```

```

        Material mat = new Material(shader);
        mat.name = materialName;

        UpdateMaterialColor(mat, Color.white);

        return mat;
    }

    private void UpdateMaterialColor(Material material, Color color)
    {
        if (material == null)
            return;

        if (material.HasProperty("_BaseColor"))
            material.SetColor("_BaseColor", color);

        if (material.HasProperty("_Color"))
            material.SetColor("_Color", color);
    }

    private enum RingPlane
    {
        XZ,
        XY,
        YZ
    }

    private void UpdateRing(LineRenderer lr, RingPlane plane, bool shouldDraw, Color
color)
    {
        if (lr == null)
            return;

        lr.enabled = shouldDraw && visible;

        if (!lr.enabled)
            return;

        int count = Mathf.Max(8, segments);

        if (lr.positionCount != count)
            lr.positionCount = count;

        lr.widthMultiplier = lineWidth;
        lr.startColor = color;
        lr.endColor = color;

        for (int i = 0; i < count; i++)
        {
            float t = i / (float)count;
            float angle = t * Mathf.PI * 2f;

            float a = Mathf.Cos(angle) * radius;
            float b = Mathf.Sin(angle) * radius;

            Vector3 p = position;

            switch (plane)
            {
                case RingPlane.XZ:
                    p += new Vector3(a, 0f, b);
                    break;

                case RingPlane.XY:

```



```

        p += new Vector3(a, b, 0f);
        break;

        case RingPlane.YZ:
            p += new Vector3(0f, a, b);
            break;
    }

    lr.SetPosition(i, p);
}

private Color EvaluateColor(float t)
{
    t = Mathf.Clamp01(t);

    if (t <= 0.5f)
    {
        float k = t / 0.5f;
        return Color.Lerp(weakColor, mediumColor, k);
    }
    else
    {
        float k = (t - 0.5f) / 0.5f;
        return Color.Lerp(mediumColor, strongColor, k);
    }
}
}
}

```

Додаток Є

Код допоміжних модулів: камери, води та шейдерів

Скрипт WorldWaterPlane.cs

```

using UnityEngine;

namespace VoxelWorldGenerator.Generation
{
    [RequireComponent(typeof(MeshFilter))]
    [RequireComponent(typeof(MeshRenderer))]
    public class WorldWaterPlane : MonoBehaviour
    {

        private MeshFilter meshFilter;
        private MeshRenderer meshRenderer;
        private Material runtimeMaterial;

        private void Awake()
        {
            meshFilter = GetComponent<MeshFilter>();
            meshRenderer = GetComponent<MeshRenderer>();
        }

        private void Start()
        {
            if (buildOnStart)
                Rebuild();
        }

        private void Update()
        {
            if (rebuildEveryFrame)
                Rebuild();
        }

        [ContextMenu("Rebuild Water Plane")]
        public void Rebuild()
        {
            if (generator == null)
                generator = FindObjectOfType<VoxelWorldGenerator>();

            if (generator == null || generator.settings == null)
                return;

            WorldGenSettings s = generator.settings;

            if (!s.generateWater)
            {
                meshFilter.sharedMesh = null;
                return;
            }

            if (forceZeroRotationAndScale)
            {
                transform.rotation = Quaternion.identity;
                transform.localScale = Vector3.one;
            }

            float size = s.worldSize;
            float half = size * 0.5f;
            transform.position = new Vector3(half, s.waterLevel, half);

            Mesh mesh = new Mesh();
            mesh.name = "World Water Plane";
        }
    }
}

```

```

Vector3[] vertices =
{
    new Vector3(-half, 0f, -half),
    new Vector3( half, 0f, -half),
    new Vector3( half, 0f,  half),
    new Vector3(-half, 0f,  half)
};

int[] triangles =
{
    0, 2, 1,
    0, 3, 2
};

float uvScale = Mathf.Max(1f, size / 64f);

Vector2[] uvs =
{
    new Vector2(0f, 0f),
    new Vector2(uvScale, 0f),
    new Vector2(uvScale, uvScale),
    new Vector2(0f, uvScale)
};

mesh.vertices = vertices;
mesh.triangles = triangles;
mesh.uv = uvs;
mesh.RecalculateNormals();
mesh.RecalculateBounds();

meshFilter.sharedMesh = mesh;
meshRenderer.sharedMaterial = GetWaterMaterial(s);
}

private Material GetWaterMaterial(WorldGenSettings settings)
{
    if (settings.waterMaterial != null)
        return settings.waterMaterial;

    if (runtimeMaterial == null)
    {
        Shader shader = Shader.Find("VoxelWorldGenerator/URP Simple Water");

        if (shader == null)
            shader = Shader.Find("Universal Render Pipeline/Unlit");

        runtimeMaterial = new Material(shader);
        runtimeMaterial.name = "Runtime World Water Material";
    }

    ApplyMaterialSettings(runtimeMaterial, settings);
    return runtimeMaterial;
}

private void ApplyMaterialSettings(Material material, WorldGenSettings settings)
{
    if (material == null)
        return;

    Color c = settings.waterColor;

    if (material.HasProperty("_Color"))
        material.SetColor("_Color", c);
}

```

```

        if (material.HasProperty("_BaseColor"))
            material.SetColor("_BaseColor", c);

        if (material.HasProperty("_Alpha"))
            material.SetFloat("_Alpha", c.a);
    }
}

```

Скрипт FlyCameraController.cs

```

using UnityEngine;
using UnityEngine.InputSystem;

namespace VoxelWorldGenerator.Player
{
    public class FlyCameraController : MonoBehaviour
    {
        [Header("Movement")]
        public float moveSpeed = 35f;
        public float fastMoveMultiplier = 4f;
        public float slowMoveMultiplier = 0.25f;

        [Header("Look")]
        public float mouseSensitivity = 0.12f;
        public bool lockCursorOnPlay = true;

        [Header("Keys")]
        public Key upKey = Key.E;
        public Key downKey = Key.Q;
        public Key fastKey = Key.LeftShift;
        public Key slowKey = Key.LeftCtrl;

        private float yaw;
        private float pitch;

        private Keyboard Keyboard => Keyboard.current;
        private Mouse Mouse => Mouse.current;

        private void Start()
        {
            Vector3 euler = transform.eulerAngles;
            yaw = euler.y;
            pitch = NormalizePitch(euler.x);

            if (lockCursorOnPlay)
                LockCursor();
        }

        private void Update()
        {
            if (Keyboard == null || Mouse == null)
                return;

            HandleCursor();
            HandleLook();
            HandleMovement();
        }

        private void HandleCursor()
        {
            if (Keyboard.escapeKey.wasPressedThisFrame)
                UnlockCursor();

            if (Mouse.rightButton.wasPressedThisFrame)
                LockCursor();
        }
    }
}

```

```

private void HandleLook()
{
    if (Cursor.lockState != CursorLockMode.Locked)
        return;

    Vector2 mouseDelta = Mouse.delta.ReadValue();

    yaw += mouseDelta.x * mouseSensitivity;
    pitch -= mouseDelta.y * mouseSensitivity;
    pitch = Mathf.Clamp(pitch, -89f, 89f);

    transform.rotation = Quaternion.Euler(pitch, yaw, 0f);
}

private void HandleMovement()
{
    Vector3 input = Vector3.zero;

    if (Keyboard.wKey.isPressed) input += Vector3.forward;
    if (Keyboard.sKey.isPressed) input += Vector3.back;
    if (Keyboard.dKey.isPressed) input += Vector3.right;
    if (Keyboard.aKey.isPressed) input += Vector3.left;

    if (IsKeyPressed(upKey)) input += Vector3.up;
    if (IsKeyPressed(downKey)) input += Vector3.down;

    float speed = moveSpeed;

    if (IsKeyPressed(fastKey))
        speed *= fastMoveMultiplier;

    if (IsKeyPressed(slowKey))
        speed *= slowMoveMultiplier;

    Vector3 move =
        transform.forward * input.z +
        transform.right * input.x +
        Vector3.up * input.y;

    if (move.sqrMagnitude > 1f)
        move.Normalize();

    transform.position += move * speed * Time.deltaTime;
}

private bool IsKeyPressed(Key key)
{
    if (Keyboard == null)
        return false;

    return Keyboard[key].isPressed;
}

private void LockCursor()
{
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
}

private void UnlockCursor()
{
    Cursor.lockState = CursorLockMode.None;
    Cursor.visible = true;
}

```

```

    }

    private float NormalizePitch(float angle)
    {
        if (angle > 180f)
            angle -= 360f;

        return angle;
    }
}

```

Скрипт VoxelTerrainVertexColor.shader

```

Shader "VoxelWorldGenerator/Vertex Color Terrain"
{
    Properties
    {
        _Brightness ("Brightness", Range(0.2, 2.0)) = 0.96
        _VariationStrength ("Variation Strength", Range(0.0, 0.45)) = 0.18
        _VariationScale ("Variation Scale", Float) = 0.022
        _FineVariationScale ("Fine Variation Scale", Float) = 0.085
        _SlopeDarken ("Slope Darken", Range(0.0, 0.5)) = 0.14
    }

    SubShader
    {
        Tags
        {
            "RenderPipeline"="UniversalPipeline"
            "RenderType"="Opaque"
            "Queue"="Geometry"
        }

        Pass
        {
            Name "ForwardLit"
            Tags { "LightMode"="UniversalForward" }

            Cull Back
            ZWrite On
            ZTest LEqual

            HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS
            #pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
            #pragma multi_compile _ _ADDITIONAL_LIGHTS
            #pragma multi_compile _ _SHADOWS_SOFT

            #include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Core.hlsl"
            #include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Lighting.hlsl"

            float _Brightness;
            float _VariationStrength;
            float _VariationScale;
            float _FineVariationScale;
            float _SlopeDarken;

            struct Attributes
            {
                float4 positionOS : POSITION;
                float3 normalOS : NORMAL;
                float4 color : COLOR;
            }

```

```

};

struct Varyings
{
    float4 positionHCS : SV_POSITION;
    float3 positionWS : TEXCOORD0;
    float3 normalWS : TEXCOORD1;
    float4 color : COLOR;
};

float Hash21(float2 p)
{
    p = frac(p * float2(123.34, 456.21));
    p += dot(p, p + 45.32);
    return frac(p.x * p.y);
}

float SmoothNoise(float2 p)
{
    float2 i = floor(p);
    float2 f = frac(p);

    float a = Hash21(i);
    float b = Hash21(i + float2(1, 0));
    float c = Hash21(i + float2(0, 1));
    float d = Hash21(i + float2(1, 1));

    float2 u = f * f * (3.0 - 2.0 * f);

    return lerp(lerp(a, b, u.x), lerp(c, d, u.x), u.y);
}

Varyings vert(Attributes input)
{
    Varyings output;

    VertexPositionInputs posInputs =
GetVertexPositionInputs(input.positionOS.xyz);
    VertexNormalInputs normalInputs = GetVertexNormalInputs(input.normalOS);

    output.positionHCS = posInputs.positionCS;
    output.positionWS = posInputs.positionWS;
    output.normalWS = normalize(normalInputs.normalWS);
    output.color = input.color;

    return output;
}

half4 frag(Varyings input) : SV_Target
{
    float3 normalWS = normalize(input.normalWS);

    Light mainLight = GetMainLight();

    float ndotl = saturate(dot(normalWS, mainLight.direction));
    float3 ambient = SampleSH(normalWS);

    float largeNoise = SmoothNoise(input.positionWS.xz * _VariationScale);
    float fineNoise = SmoothNoise(input.positionWS.xz * _FineVariationScale +
31.7);

    float mixed = (largeNoise - 0.5) * 0.75 + (fineNoise - 0.5) * 0.25;

    float3 baseColor = input.color.rgb;

```

```

float3 warmer = baseColor * float3(1.05, 0.98, 0.88);
float3 cooler = baseColor * float3(0.86, 0.95, 1.02);
float3 darker = baseColor * 0.72;

float warmMask = saturate(0.5 + mixed * 2.0);
float darkMask = saturate(abs(mixed) * 1.35) * _VariationStrength;

float3 varied = lerp(cooler, warmer, warmMask);
baseColor = lerp(baseColor, varied, _VariationStrength);
baseColor = lerp(baseColor, darker, darkMask * 0.55);

float slopeFactor = saturate(normalWS.y);
float slopeDarken = lerp(1.0 - _SlopeDarken, 1.0, slopeFactor);
baseColor *= slopeDarken;

baseColor = saturate(baseColor);

float3 lighting = ambient + mainLight.color * ndotl;
float3 finalColor = baseColor * lighting * _Brightness;

return half4(finalColor, 1.0);
}

ENDHLSL
}
}
}

```

Скрипт VoxelWaterURP.shader

```

Shader "VoxelWorldGenerator/Water"
{
    Properties
    {
        _Color ("Water Color", Color) = (0.02, 0.16, 0.34, 0.88)
        _Alpha ("Alpha", Range(0.0, 1.0)) = 0.88
        _ColorVariation ("Color Variation", Range(0.0, 0.35)) = 0.10
        _RippleScale ("Ripple Scale", Float) = 0.035
        _RippleSpeed ("Ripple Speed", Float) = 0.45
        _SpecularStrength ("Specular Strength", Range(0.0, 1.5)) = 0.35
        _FresnelStrength ("Fresnel Strength", Range(0.0, 1.0)) = 0.18
    }

    SubShader
    {
        Tags
        {
            "RenderPipeline"="UniversalPipeline"
            "RenderType"="Transparent"
            "Queue"="Transparent"
        }

        Pass
        {
            Name "Water"
            Tags { "LightMode"="UniversalForward" }

            Blend SrcAlpha OneMinusSrcAlpha
            ZWrite Off
            ZTest LEqual
            Cull Off

            HLSLPROGRAM

            #pragma vertex vert
            #pragma fragment frag

```



```

#include "Packages/com.unity.render-
pipelines.universal/ShaderLibrary/Core.hlsl"

float4 _Color;
float _Alpha;
float _ColorVariation;
float _RippleScale;
float _RippleSpeed;
float _SpecularStrength;
float _FresnelStrength;

struct Attributes
{
    float4 positionOS : POSITION;
    float2 uv : TEXCOORD0;
};

struct Varyings
{
    float4 positionHCS : SV_POSITION;
    float3 positionWS : TEXCOORD0;
    float2 uv : TEXCOORD1;
};

float SmoothWave(float2 p, float t)
{
    float a = sin(p.x * 1.00 + t * 1.20);
    float b = sin(p.y * 1.37 - t * 0.95);
    float c = sin((p.x + p.y) * 0.72 + t * 0.65);
    float d = cos((p.x * 0.63 - p.y * 0.81) - t * 0.50);

    return (a + b + c + d) * 0.25;
}

Varyings vert(Attributes input)
{
    Varyings output;

    VertexPositionInputs posInputs =
GetVertexPositionInputs(input.positionOS.xyz);

    output.positionHCS = posInputs.positionCS;
    output.positionWS = posInputs.positionWS;
    output.uv = input.uv;

    return output;
}

half4 frag(Varyings input) : SV_Target
{
    float t = _Time.y * _RippleSpeed;

    float2 p1 = input.positionWS.xz * _RippleScale;
    float2 p2 = input.positionWS.xz * _RippleScale * 2.15 + float2(17.3, -9.1);

    float waveA = SmoothWave(p1, t);
    float waveB = SmoothWave(p2, t * 1.37);

    float wave = waveA * 0.7 + waveB * 0.3;

    float3 baseColor = _Color.rgb;

    float3 darkWater = baseColor * 0.72;
    float3 lightWater = saturate(baseColor + float3(0.025, 0.065, 0.095));

```

```

float variation = saturate(0.5 + wave * _ColorVariation);
float3 color = lerp(darkWater, lightWater, variation);

float3 viewDir = normalize(_WorldSpaceCameraPos.xyz - input.positionWS);
float fresnel = pow(1.0 - saturate(dot(viewDir, float3(0, 1, 0))), 3.0);

float spec = pow(saturate(wave * 0.5 + 0.5), 18.0) * _SpecularStrength;

color += fresnel * _FresnelStrength;
color += spec * 0.10;

return half4(color, _Alpha);
}

ENDHLSL
}
}
}

```