ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА
(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп’ютерних систем
(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття рівня вищої освіти «бакалавр»
(рівень вищої освіти)

на тему Застосування двовимірних фракталів в якості динамічних текстур у комп'ютерній графіці
The applying of two-dimensional fractals as dynamic textures in computer graphics

Виконав: студент денної форми навчання
спеціальності 126 – Інформаційні системи та технології .
(шифр і назва напряму підготовки, спеціальності)
Освітня програма «Інформаційні системи та технології»
(назва освітньої програми)
Сокур Олексій Максимович
(прізвище, ім’я, по-батькові)

Керівник к.ф.-м.н., доц. Петрушина Т.І. ________
(науковий ступінь, вчене звання, прізвище та ініціали, **підпис**)

Рецензент д.т.н., проф. Малахов Є.В. ________
(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:
Протокол засідання кафедри
№___ від «___»_____ 2024 р.
Завідувач кафедри
____________ Євгеній МАЛАХОВ
(підпис) (ім’я, прізвище)

Захищено на засіданні ЕК № ___
протокол № __ від «___» ____ 2024 р.
Оцінка _____________/______/_____
(за національною шкалою, шкалою ECTS, бали)
Голова ЕК
_______ Володимир ВИЧУЖАНІН
(підпис) (ім’я, прізвище)

Одеса - 2024

АНОТАЦІЯ

У даній кваліфікаційній роботі розглядається тема «Застосування двовимірних фракталів в якості динамічних текстур у комп'ютерній графіці».

Метою даної роботи є спрощення і пришвидшення процесу розробки динамічних процедурних текстур на основі фракталів та просування ідеї використання таких текстур замість растрових в комп’ютерній графіці. Мету планується досягти шляхом розробки спеціалізованого програмного забезпечення для PC, що спростить редагування таких текстур, та дослідження його можливих застосувань.

В роботі розглядаються нюанси створення та застосування динамічних текстур в комп’ютерній графіці, особлива увага приділяється фрактальним текстурам.

В результаті аналізу аналогів та дослідження предметної області спроектовано та розроблено редактор динамічних текстур, що добре оптимізований та має наступні характерні можливості:

1) створення текстури на основі HLSL коду з гнучким налаштуванням відображення;

2) контроль над параметрами за допомогою графічного інтерфейсу або програмного коду;

3) автоматичне оновлення відображення текстури при зміні коду чи параметрів з мінімальною затримкою;

4) експорт текстури у вигляді вихідного коду шейдеру HLSL чи GLSL в формі функції, готової для застосування, або ж збереження як звичайне зображення;

5) одночасна робота над декількома текстурами із швидким перемиканням між ними;

6) підтримка Windows, Linux та MAC.

З метою демонстрації можливостей та тестування, за допомогою редактора розроблено ряд фрактальних 2D та 3D текстур. Також продемонстровано експортування текстур до графічних рушіїв UE 5 та Unity.

ABSTRACT

The topic of this theses is “The applying of two-dimensional fractals as dynamic textures in computer graphics”.

The purpose of this work is to simplify and speed up the process of developing dynamic procedural textures based on fractals and advance an idea of application of such textures instead of raster ones in computer graphics. The goal is planned to be achieved by developing specialized software for the PC, which will simplify the editing of such textures, and further research on its possible applications.

The work examines the creation and application of dynamic textures in computer graphics, with special attention paid to fractal textures.

As a result, after the analysis of analogues and research of the subject area, a dynamic texture editor was designed and developed, it is well optimized and has the following characteristic capabilities:

1) creation of textures based on HLSL code with flexible preview settings;

2) control over parameters via a graphical interface or program code;

3) automatic updating of the texture preview when changing the code or parameters with minimal delay;

4) export the texture to the HLSL or GLSL shader code in the form of a ready-to-use function, or saving the texture as an image;

5) simultaneous work on multiple textures with rapid switching between them;

6) supports Windows, Linux and MAC.

In order to demonstrate the capabilities and test the editor, a number of fractals 2D and 3D textures were developed using the editor. Additionally, the export of the created textures to the UE 5 and Unity graphics engines was demonstrated.

ЗМІСТ

ВСТУП 9

1 ОГЛЯД МЕТОДІВ ГЕНЕРАЦІЇ ТЕКСТУР 12

1.1 Сканування матеріалів 12

1.2 Цифрові інструменти (растрові графічні редактори) 12

1.3 Генерація за допомогою штучного інтелекту 13

1.4 Процедурна генерація 13

1.5 Висновок 14

2 ВІДОБРАЖЕННЯ ТЕКСТУР ЗА ДОПОМОГОЮ ГРАФІЧНОГО ПРОЦЕСОРА 15

2.1 Графічний конвеєр 15

2.2 Шейдер 16

2.3 Способи відображення 17

3 ФРАКТАЛЬНІ ТЕКСТУРИ 19

4 ПОСТАНОВКА ЗАДАЧІ 22

5 РОЗРОБКА РЕДАКТОРА ДИНАМІЧНИХ ПРОЦЕДУРНИХ ТЕКСТУР 23

5.1 Існуючі рішення 23

5.2 Проектування 27

5.2.1 Компіляція вихідного коду текстури для відображення 27

5.2.2 Параметри 28

5.2.3 Експорт 29

5.3 Вибір програмного забезпечення 31

5.4 Програмна Реалізація 32

5.5 Інструкція з використання 34

5.5.1 Загальний інтерфейс 34
5.5.2 Редагування текстури 36
5.5.3 Динамічні та статичні параметри. 42
5.5.4 Експортування текстури 43
6 СТВОРЕННЯ ДИНАМІЧНИХ ФРАКТАЛЬНИХ ПРОЦЕДУРНИХ ТЕКСТУР ЗА ДОПОМОГОЮ РЕДАКТОРА 46
6.1 Реалізація відомих фрактальних множин 46
6.1.1 Множина Мандельброта 46
6.1.2 Множина Жуліа 52
6.1.3 Фрактал Ньютона 57
6.1.4 Палаючий корабель 59
6.1.5 Фрактали Ляпунова 63
6.2 Створення нових різновидів фракталів 64
6.2.1 Модифікації Множини Мандельброта та відповідної множини Жуліа. 64
6.3 Фрактальний шум 66
6.3.1 Шум Перліна. 67
6.3.2 Фрактальний броунівський рух 68
6.3.3 Використання фрактального шуму для викривлення простору 69
6.4 Анімація фрактальних текстур 74
6.5 Точність обчислення алгоритмів 77
6.6 Робота з тривимірними текстурами 79
7 СТВОРЕННЯ ДЕМОНСТАЦІЙНОЇ СЦЕНИ З ВИКОРИСТАННЯМ ФРАКТАЛІВ В ЯКОСТІ ТЕКСТУР 81
7.1 Експорт текстури в Unreal Engine 81

7.1 Експорт текстури в Unity 82
ВИСНОВКИ 84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 87
ДОДАТОК А Коди клавіш, що доступні для використання в редакторі 91
ДОДАТОК Б Повний код розроблених тексту 92
ДОДАТОК В Варіації текстур на основі множини Мандельброта 113

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

Скорочення:

API – An application programming interface (інтерфейс прикладного програмування).

GPU – graphics processing unit (графічний процесор).

SIMD – single instruction, multiple data (одиночний потік команд, множинний потік даних).

Терміни:

Во́ксель (від англ. Volume та англ. pixel) — елемент простору, позначає значення певної величини в клітинках рівномірної просторової ґратки. Аналогічний пікселю, у двовимірних зображеннях (значенням величини в клітинах ґратки виступає колір).

Графічний рушій (англ. graphics engine; іноді «рендерер» або «візуалізатор») — підпрограмне забезпечення (англ. middleware), програмний редактор, основним завданням якого є візуалізація (рендеринг) двовимірної або тривимірної комп'ютерної графіки. Може існувати як окремий продукт або в складі ігрового рушія.

Динамічна текстура – текстура обчислена під час візуалізації в шейдері на GPU і така що зберігаються у вигляді програмного коду, а не растрового зображення.

Растрове зображення – зображення, яке являє собою сітку (растр), зазвичай прямокутну, пікселів відображених на моніторі, папері та інших відображальних пристроях і матеріалах.

Тексел (англ. texel, скорочення від англ. Texture element) – є фундаментальною одиницею текстурного простору, використовується у комп'ютерній графіці. Текстури представлені масивами текселів, так само, як фотографії представлені масивами пікселів.

Фракта́л (лат. fractus — подрібнений, зламаний, розбитий) — множина, що має властивість самоподібності (об'єкт, що точно або приблизно збігається з частиною себе самого, тобто ціле має ту саму форму, що й одна або більше частин). У математиці під фракталами розуміють безліч точок в евклідовому просторі, що мають дробову метричну розмірність (у сенсі Мінковського або Хаусдорфа), або метричну розмірність, відмінну від топологічної, тому їх слід відрізняти від інших геометричних фігур, обмежених кінцевим числом ланок.

Шейдер — це визначена користувачем програма, призначена для роботи на певному етапі графічного конвеєра.

ВСТУП

Сьогодні комп’ютерна графіка швидко прогресує через застосування в багатьох промислових сферах. До таких сфер можна віднести сферу розваг, куди входять кінематографія, де за допомогою комп’ютерної графіки створюються ефекти, анімації чи навіть фільми цілком, та розробка комп’ютерних ігор, де вирішується більш складна задача відображення віртуального світу в реальному часі. Крім цього, комп’ютерна графіка знаходить широке застосування в дизайні, мистецтві, навчанні, симуляції, архітектурі, наукових дослідженнях та багатьох інших галузях. Серед нових та швидко прогресуючих технологій, існування яких неможливо без комп’ютерної графіки, можна відзначити VR (Virtual reality) та AR (Augmented reality), або разом XR (Extended reality), що дозволяють глибше зануритись у віртуальний світ, та мають безліч можливих застосувань.

У комп’ютерній графіці важливим аспектом є надання об’єктам певного зовнішнього кольору та властивостей матеріалу поверхні. Зазвичай це вирішується за допомогою текстур. Для визначення кольору поверхні, під час візуалізації, тривимірних та, тим паче, двовимірних об'єктів досить зручно використовувати двовимірні текстури. Це зумовлено тим, що двовимірний екран здатен відображати лише проекцію тривимірних об'єктів, та оскільки при проекції можна побачити лише поверхню, то, як правило, лише поверхня і моделюється при відображені, а поверхню ж зручно описувати двовимірним зображенням. Зазвичай, в якості таких текстур використовуються двовимірні растрові зображення, що потім накладаються на спроектовану на екран поверхню методом семплювання (вибірки текселя текстури на основі координат пікселя у просторі поверхні об’єкта або, в залежності від налаштувань семплера, фільтрація на основі декількох текселів)[1]. Однак, використання растрових зображень в якості текстур має декілька значних недоліків. По-перше, для якісних зображень необхідно досить багато пам'яті, як на фізичному накопичувачі так і в оперативній пам'яті, що

використовується під час процесу візуалізації. По-друге, точність растрових текстур обмежена розміром зображення, при масштабуванні, якість значно погіршується. Іншим недоліком є статичність зображення, для реалізації певних видів анімації необхідно використовувати набір растрових зображень з їх швидкою підміною, і це коштує значно більше пам'яті.

Майже всі сучасні графічні процесори підтримують можливість керувати певними стадіями графічного конвеєру. Однією з таких стадій є піксельний або фрагментний шейдер. Фрагментний шейдер – це частина графічного конвеєру, що може бути запрограмована (на відмінно від багатьох інших стадій, що виконуються апаратно та реалізація яких скрита) що дозволяє керувати процесом вибірки текстури і затіненням, фактично відповідає за те, який колір буде призначено пікселю екрана. Замість того щоб семплювати растрове зображення у фрагментному шейдері (що виконується у більшості випадків), тут можна напряму обчислити це зображення за певним математичним правилом, керуючись текстурними координатами фрагмента (пікселя). Ці обчислення відбуваються динамічно під час візуалізації кожного з кадрів безпосередньо на GPU і не зберігаються в пам'яті. Крім того, такий підхід, в залежності від складності алгоритму, може бути швидшим за семплювання растрового зображення через досить дорогу ціну довільного доступу до пам'яті на GPU. Також, за потреби, досить легко ввести параметри й анімувати текстуру.

Динамічними, в рамках цієї роботи, будуть вважатися текстури отримані шляхом описаним вище, тобто обчисленні під час візуалізації в шейдері на GPU і що зберігаються у вигляді програмного коду, а не растрового зображення.

Хоча найбільш очевидним застосуванням динамічної текстури є визначення кольору поверхні, її також можна використовувати і для інших цілей. Наприклад, за допомогою теселяційного шейдеру [2] можна генерувати полігони на льоту, таким чином, що поверхня буде визначатися текстурою. В цьому використання процедурних текстур також має сенс так як вони мають

дуже велику точність, а рівень деталізації можна змінювати в залежності від відстані до об’єкта.

Проблемою ж при динамічній генерації текстур стає знаходження певного математичного правила, що дозволить отримати текстуру яка підходить до конкретної задачі.

Існує багато різних способів процедурного генерування текстур, але не всі вони доступні для реалізації (або складно чи неефективно реалізуємі) у шейдері, враховуючи обмеження архітектури (більш детально у підрозділі 2.2). Одним з підходів є створення простих геометричних фігур за певними математичними правилами, або комбінації таких фігур. Це можуть бути лінії, еліпси, прямокутники, та майже будь-які фігури що описуються математично на площині. Взагалі можна використовувати будь-які функції двох змінних. У багатьох випадках цього буде достатньо, але при моделюванні реалістичних об’єктів, особливо природніх, добре визначені прості фігури та функції вже не підходять. Для таких текстур, як правило, бажана більш хаотична структура.

Найбільш цікавими з точки зору складності, унікальності та хаотичності зображення є фрактали. Однією з важливих властивостей фракталів є самоподібність, що також вирішує проблему якості при масштабуванні.

При розробці динамічних текстур, особливо фрактальних, постає необхідність в певній середі, що дозволить зручно це зробити. Як виявилось створення такого роду текстур здебільшого виконувалось прямим написанням шейдерного коду. Для того щоб покращити процес створення та дослідження динамічних процедурних текстур є сенс розробки спеціалізованої програми, що дозволить підвищити ефективність розробки процедурних динамічних текстур.

1 ОГЛЯД МЕТОДІВ ГЕНЕРАЦІЇ ТЕКСТУР

1.1 Сканування матеріалів

Цей метод передбачає захоплення матеріалів реального світу за допомогою спеціального обладнання, наприклад 3D-сканерів або камер високої роздільної здатності.

Перевагами методу є:

1) низькі витрати часу;

2) реалістичність отриманої текстури. Текстура більш точно відповідає реальному світу якщо є така необхідність.

Недоліки:

1) текстури, що можуть бути отримані таким способом, обмежені існуючими об'єктами;

2) для створення такої текстури необхідне спеціальне обладнання та зразок матеріалу, що не завжди легко отримати;

3) лише растрова текстура може бути згенерована, а точність залежить від якості обладнання;

1.2 Цифрові інструменти (растрові графічні редактори)

Такі програми, як Photoshop [3], GIMP [4], Microsoft Paint [5], пропонують пензлі, фільтри та ефекти для створення растрових текстур на цифровому полотні.

Основною перевагою є майже безмежні можливості для створення текстури. Важливими недоліками ж те, що це потребує людських ресурсів та багато часу. Знову ж таки може бути отримана лише растрова текстура.

1.3 Генерація за допомогою штучного інтелекту

Сьогодні існують згорточні нейронні мережі здатні генерувати досить складні та якісні зображення. Серед таких популярними є DALL·E 3 [6], що розроблена OpenAI та Stable Diffusion від Stability AI[7]. Основною перевагою є незначні витрати часу для генерації.

До недоліків входять:

1) обмежений розмір зображення;

2) генерується лише растрове зображення;

3) зображення може бути з дефектами та потребувати доопрацювання;

4) необхідне дуже потужне обладнання.

1.4 Процедурна генерація

Цей метод передбачає використання алгоритмів для математичної генерації текстур [8].

Переваги:

1) математичне описання дозволяє не зберігати растрове зображення в пам'яті комп'ютера;

2) не втрачає якості при масштабуванні та інших математичних перетвореннях простору;

3) зображення може бути легко модифіковане зміною параметрів в реальному часі;

4) зберігаються інформація про всі етапи створення текстури;

5) може використовуватися для генерації динамічної текстури.

Недоліком є складність підібрати алгоритм для бажаної текстури.

1.5 Висновок

Розглядаючи проблему динамічної генерації текстур в реальному часі лише процедурна генерація є можливою опцією.

Теоретично генерація за допомогою штучного інтелекту також може бути застосована при динамічний генерації текстур і навіть існують певні алгоритми що виконують це. Як приклад, можна навести технологію DLSS (Deep Learning Super Sampling) [9] компанії Nvidia, що дозволяє підвищити якість кінцевого зображення в реальному часі. Але тут алгоритм застосовується не до текстури, а вже до кінцевого зображення і лише один раз на кадр. Використання технології для генерації багатьох текстур буде дуже неефективне так як алгоритм приймає ціле зображення і повертає ціле зображення (вже більшої роздільної здатності). На даний момент потужності сучасних GPU недостатньо для практичного застосування штучного інтелекту в реальному часі для ефективної динамічної генерації багатьох текстур, особливо на GPU з середньої цінової категорії чи мобільних пристроях.

2 ВІДОБРАЖЕННЯ ТЕКСТУР ЗА ДОПОМОГОЮ ГРАФІЧНОГО ПРОЦЕСОРА

2.1 Графічний конвеєр

Графічний конвеєр – це фреймворк для візуалізації 3D (чи 2D) сцен на екрані. Графічний конвеєр представляє собою певний ланцюг послідовно виконуваних операцій над даними для перетворення їх у кінцеве 2D зображення. Здебільшого операції графічного конвеєра виконуються на GPU на апаратному рівні або на рівні драйвера. Графічні API, такі як Direct3D [10], OpenGL [11], Vulkan [12], Metal [13], представляють певний стандартизований інтерфейс для керування конвеєром. Етапи графічного конвеєра залежать від архітектури GPU та графічного API, але здебільшого можна виділити такі основні етапи (рис. 2.1 [14]), серед яких сині блоки програмовані шейдери.

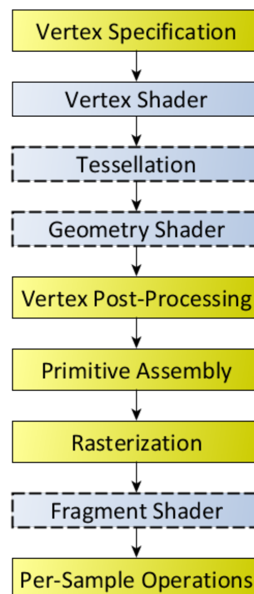

Рисунок 2.1 – Основні етапи графічного конвеєра

Окрім звичайного графічного конвеєру також існує обчислювальний конвеєр та конвеєр трасування променів, де також застосовуються шейдери.

2.2 Шейдер

Згідно із специфікацією OpenGL [15] шейдер – це визначена користувачем програма, призначена для роботи на певному етапі графічного конвеєра. Шейдери надають код для певних програмованих етапів конвеєра візуалізації. Їх також можна використовувати в дещо більш обмеженій формі для загальних обчислень на GPU.

Шейдери виникли з метою реалізації затінення та освітлення. Звідси також і назва *shader*. На початку підтримувався лише фрагментний шейдер.

Шейдером також називають і вихідний код такої програми. Наразі в широкому застосуванні є 3 високорівневі мови програмування шейдерів:

1) GLSL (OpenGL Shading Language) [16] – мова програмування шейдерів першочергово розроблена для OpenGL, яка має синтаксис, що схожий на мову C;

2) HLSL (High-level shader language) [17] – розроблена Microsoft і використовується для програмування шейдерів в DirectX. Має більш високий рівень за GLSL, хоча різниця дуже незначна;

3) MSL (Metal Shading Language) [18] – мова затінення, що базується на C++ та використовується на пристроях Apple при застосуванні графічного API Metal.

Найпопулярнішою серед цих мов є HLSL, використовується в багатьох графічних рушіях, серед яких Unity [19] , Unreal Engine [20] . Так як всі ці мови схожі по синтаксису то, як правило, автоматичний переклад між ними не є проблемою.

Основною різницею програмування шейдерів порівняно з програмуванням звичайного коду для CPU є те, що шейдери виконуються паралельно великими групами. Кількість шейдерних ядер може сягати десятків тисяч. Наприклад GeForce RTX 4090 має 16384 ядер [21]. Велика

кількість ядер також має певні обмеження що до структури коду та доступу до пам'яті. По-перше, це відсутність кучі й стеку та відповідно динамічного виділення пам'яті і рекурсії. Всі змінні в межах шейдеру мають бути статично об'явлені та їх розмір відомий під час компіляції. Є можливість доступу до буферів і текстур, але створення самих буферів і текстур і їх відповідне знищення виконується за межами шейдерного коду, як правило, керується на стороні CPU. Виконання шейдерів виконується по групам, ядра в одній групі поділяють між собою певні ресурси. В межах однієї групи використовується модель SIMD (Single Instruction/Multiple Data) [22], тобто лише один потік інструкцій на групу паралельних ядер. Як результат, будь-які розгалуження (оператор `if` або цикли, що мають різну кількість ітерацій) імітуються і фактично виконуються всі гілки програми. В більш сучасних версіях є можливість динамічного розгалуження, але вона коштує багато ресурсів [23] .

2.3 Способи відображення

На всіх етапах графічного конвеєру, що визначені як програмовані, можна використовувати шейдерні програми. Тут при розрахунку будь то кольору, нормалі, коефіцієнту віддзеркалення, матовості чи зміщення вершин майже не обходиться без використання текстур, що характеризують поверхню чи об'єм. Одним із найпоширеніших підходів є семплювання растрового зображення.

Перевагами є:

1) зручність з точки зору розробника;

2) відсутні обмеження на вміст текстури;

3) висока швидкість для невеликих зображень.

Недоліками:

1) необхідно багато пам'яті для зберігання якісного зображення;

2) погана якість при збільшенні зменшені чи відображенні під кутом;

3) відносно висока затримка при читані з пам’яті;

4) зображення статичне.

Іншим підходом є обчислення властивостей поверхні під час виконання, на основі текстурних координат точки. Підхід має такі переваги:

1) якість не залежить від масштабування (якщо не враховувати обмеження точності типів даних);

2) можливість керування текстурою динамічно за допомогою зміни параметрів;

3) майже не потребує пам'яті, так як текстура повністю описується шейдерним кодом;

4) може використовуватись для описання дуже великих поверхонь.

Серед недоліків:

1) час відображення залежить від складності алгоритму;

2) важко знайти інструменти, що допоможуть створювати код таких текстур.

Ще одним варіантом може бути поєднання обох методів. Наприклад, можна обрати растрове зображення та надати йому певної анімації на основі додаткових обчислень чи перетворень простору.

3 ФРАКТАЛЬНІ ТЕКСТУРИ

Більшість цікавих процедурних текстур базуються саме на фракталах.

При досліджені динамічних процедурних текстур знайдено наступні реалізації, що добре демонструють можливості використання фракталів в реальному часі для моделювання природніх об’єктів.

На рисунку 3.1 анімована текстура органічного матеріалу[24].

Рисунок 3.1 – Фрактальна динамічна текстура органічного матеріалу

Інший приклад застосування фракталів для моделювання хвиль (рис. 3.2) [25].

На рисунку 3.3 [26] зображено вже тривимірні текстури що представляють фрактальний ландшафт.

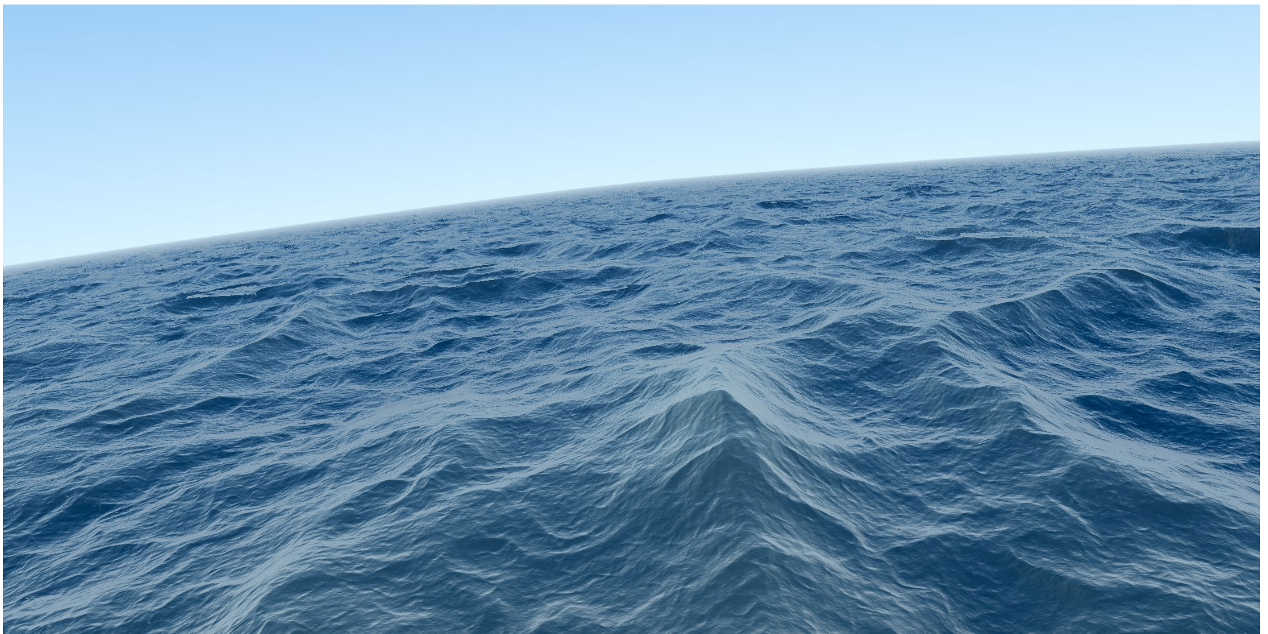

Рисунок 3.2 – Фрактальна динамічна текстура хвиль

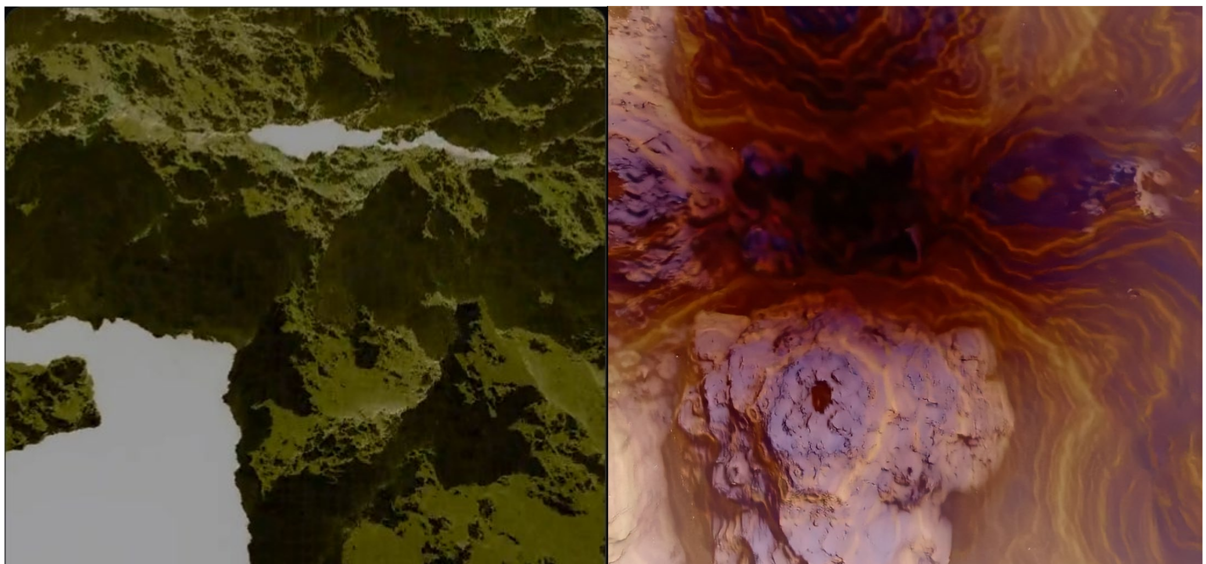

Рисунок 3.3 – Приклади фрактального ландшафту

Ще один приклад (рис. 3.4) [27], де ціла сцена генерується в реальному часі за допомогою фракталів. Тут фрактальні текстури використовуються для генерації ландшафту, дерев та хмар.

Рисунок 3.4 – Відображення динамічної сцени, побудованої з фракталів, в реальному часі

4 ПОСТАНОВКА ЗАДАЧІ

Метою даної роботи є спрощення і пришвидшення процесу розробки динамічних процедурних текстур на основі фракталів та просування ідеї використання таких текстур замість растрових в комп'ютерній графіці. Мету планується досягти шляхом розробки спеціалізованого програмного забезпечення для PC, що спростить редагування таких текстур, та дослідження його можливих застосувань.

Для досягнення мети необхідно:

1) проаналізувати предметну область та існуючі інструменти для створення процедурних текстур;

2) сформувати вимоги до програмного забезпечення враховуючи досвід аналогів;

3) розробити спеціалізований редактор процедурних текстур які будуть описуються програмним кодом з можливістю керування параметрами та зручним експортом результуючого коду текстури;

4) виконати реалізацію відомих фракталів та фрактального шуму за допомогою розробленого редактора. Дослідити інші варіації фракталів;

5) створити демонстраційну 3D сцену за допомогою графічного рушія з використанням створених шаблонів;

6) сформувати висновки щодо використання динамічних текстур на основі фракталів в комп'ютерній графіці.

5 РОЗРОБКА РЕДАКТОРА ДИНАМІЧНИХ ПРОЦЕДУРНИХ ТЕКСТУР

5.1 Існуючі рішення

Для процедурної генерації текстур існує багато інструментів. Але серед них немає зручних для роботи саме з фракталами, оскільки для фракталів необхідно підбирати багато параметрів, невелика зміна яких може призводити до дуже значних змін кінцевого зображення, звідси випливає необхідність в гнучкому керуванні параметрами та, бажано, з можливістю керування ними за допомогою клавіатури та миші, замість прямої зміни числових значень. Окрім цього, зовсім не всі редактори текстур дозволяють генерувати кінцевий код шейдеру, а лише створюють растрові бітмапи певного розміру, що відповідно не підходить якщо ціллю є створення динамічної текстури.

Далі розглянуті деякі існуючі рішення.

Adobe Substance Designer [28]**.** Серед переваг має гарну підтримку та регулярні оновлення. Є зручна інтеграція з графічними рушіями, що дозволяє працювати над текстурою та автоматично оновлювати її для графічного рушія. Всі зміни можна бачити в реальному часі. Текстура будується за допомогою графічного інтерфейсу з застосуванням графу вузлів. З одно боку цей підхід може бути простішим для освоєння та дозволяє бачити всі проміжні стани текстури, з іншого боку деякі прості операції, що легко та компактно описуються кодом (наприклад цикли чи виклик користувацької функції) може бути складно та громіздко втілити за допомогою вузлів.

До недоліків треба віднести те, що Substance Designer генерує текстуру в якості растрового зображення фіксованого розміру. Під час розробки відсутні можливості для масштабування певних регіонів текстури щоб детально дослідити цікаві частини згенерованого зображення. Використання

інструменту платне. Неможливо створити анімовані текстури з динамічними параметрами.

Blender's node-based shader editor [29]. Аналогічно до Substance Designer, дозволяє будувати текстури за допомогою зручного графічного інтерфейсу з застосуванням вузлів без коду. Але тут результатом вже є шейдерний код, а на растрове зображення. Можливість одразу бачити результат на 3D моделі. Дозволяє працювати над різними каналами матеріалу одночасно (колір, нормалі, карта висот, металевість). Безкоштовний, з відкритим вихідним кодом. Є вбудовані функції як, наприклад, генерація шуму.

Незручно розглядати деталі зображення та змінювати параметри для перегляду результату. Неможливо використовувати код замість графу, що може бути зручнішим для складних алгоритмів. Необхідно встановлювати Blender для використання.

Створювати процедурну текстуру можна і без використання спеціалізованих інструментів. В залежності від графічного API або рушія, код можна писати в якості звичайного шейдера. Звичайно, в такому випадку немає можливості зручного керування параметрами (їх доведеться задавати вручну в якості констант), а також відсутнє оновлення в реальному часі. Крім того, при розробці текстури бажано б мати зручний інтерфейс, що дозволить в деталях дослідити зображення.

Деякі графічні рушії дозволяють створювати матеріали за допомогою графічного інтерфейсу, аналогічному Blender's node-based shader editor та Substance Designer. Цей варіант кращий за пряме написання коду, але має високу затримку при оновлені результату та не підходить для цілей дослідження текстур, особливо фрактальних, так як важко виконати необхідне перетворення параметрів, як, наприклад масштабування та переміщення.

ShaderToy. ShaderToy [30] не є спеціалізованим інструментом для розробки текстур, а представляє редактор фрагментних GLSL шейдерів. Також може бути застосований для створення динамічних текстур.

Превагами ShaderToy є веб-інтерфейс і можливість ділитися своїми розробками. Має велику базу різноманітних процедурних зображень користувачів.

Керування параметрами обмежене константами і координатами курсора, що незручно при створені складних текстур.

Отже, проаналізувавши аналоги, при створенні редактора необхідно врахувати такі властивості:

1) більшість аналогів дозволяє бачити результат в реальному часі, це дуже спрощує створення текстури. Тому розробляючи власний редактор також важливо мінімізувати час оновлення зображення;

2) серед розглянутих рішень лише ShaderToy дозволяє писати код напряму, без використання графічного інтерфейсу за допомогою вузлів. Так як цей підхід більш універсальний та легший для реалізації, початкова версія редактора має базуватися саме на вихідному коді, тоді як візуальний інтерфейс редагування можна буде додати в подальшому;

3) головним недоліком всіх розглянутих аналогів є дуже обмежене керування параметрами. Це робить процес створення текстури довшим та незручним, особливо якщо це фрактал. При створені власного редактора необхідно дозволити керувати параметрами як за допомогою графічного інтерфейсу так і за допомогою програмного коду, що наддасть максимальної гнучкості та повний контроль з можливістю реалізації будь-якої поведінки. Крім того, керування за допомогою коду має дозволити прив’язати зміну параметрів до клавіатури, що неможливо в аналогах;

4) при розгляданні аналогічних рішень було дуже важко знайти такі, що дозволяють створювати динамічні текстури, а не растрові зображення.

Враховуючи що метою є створення саме динамічних текстур, то результатом розробленої текстури має бути саме код шейдеру у формі функції. Але, при цьому, експорт у формі звичайного растрового зображення також має бути присутній. Іноді застосування растрового зображення може бути зручнішим, а також це необхідно для порівняння роботи растрової та динамічної текстури з однаковим зображенням.

Отже, необхідно щоб редактор відповідав таким потребам:

1) редактор має бути реалізовано у вигляді десктопного застосунку з графічним інтерфейсом, оскільки нативний застосунок має більше можливостей при роботі з графікою;

2) редактор має підтримувати декілька платформ, принаймні Windows, Linux та MAC щоб відповідати потребам різних користувачів;

3) текстура описується кодом на мові HLSL;

4) можливість бачити поточний стан текстури під час розробки;

5) керування параметрами за допомогою коду;

6) можливість експортувати текстуру в якості шейдерного коду на мові HLSL та GLSL або ж зберегти як звичайне растрове зображення;

7) можливість роботи над декількома текстурами одночасно із швидким перемиканням між ними;

8) інтерактивність та мінімальна затримка при редагуванні: миттєве оновлення відображення після змін в коді або параметрів;

9) відображення допоміжного користувацького контенту, окрім самої текстури, який також може бути запрограмований, але не буде входити в експортоване зображення (наприклад дозволити відображати координатну сітку або виділити певний регіон). Такий підхід наддасть можливості створити необхідні допоміжні візуальні елементи специфічні для конкретного виду текстури (наприклад двовимірної, одновимірної чи тривимірної);

10) оцінка часу візуалізації процедурної текстури.

5.2 Проектування

5.2.1 Компіляція вихідного коду текстури для відображення

На рисунку 5.1 визначена послідовність операцій, що виконується при зміні вихідного коду для оновлення параметрів та зображення текстури. Під оновленням параметрів тут мається на увазі їх структура: послідовність, імена, тип даних, розмір, а не конкретні значення.

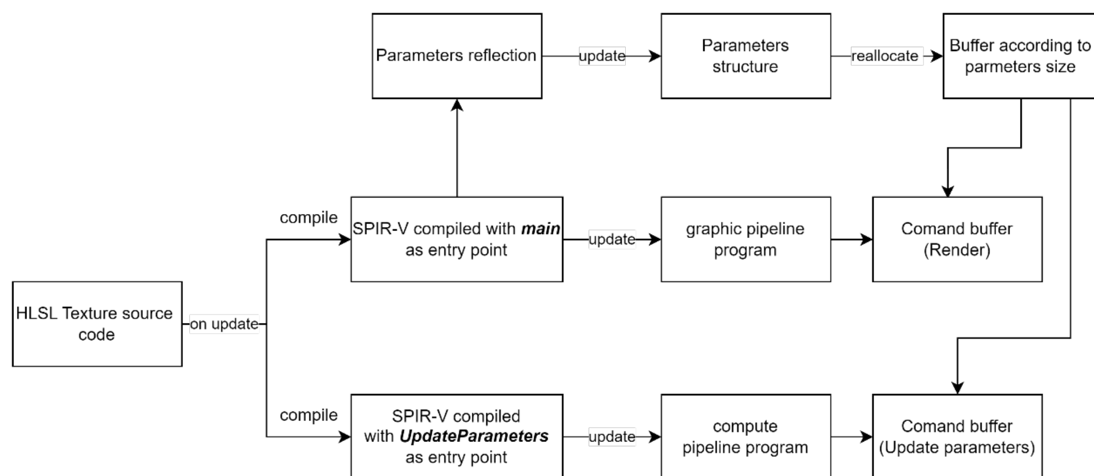

Рисунок 5.1 – Оновлення при зміні вихідного коду

Спочатку вихідний код, описуючий текстуру, компілюється з врахуванням вхідної точки `main`. Далі з бінарного коду отримується інформація про параметри, після чого оновлюються параметри і відповідний їм буфер (якщо розмір структури параметрів змінився). На основі бінарного коду створюється графічний конвеєр.

Наступним кроком, вихідний код знову компілюється, але вже з іншою точкою входу – `UpdateParameters`, на основі скомпільованого коду створюється відповідний обчислювальний конвеєр.

Далі перезаписуються командні буфери, що описують операції виконувані на GPU. При записі враховується буфер параметрів, що підключається як до графічного так і обчислювального конвеєру.

Така архітектура дозволить розділити функціонал оновлення параметрів і відображення, при цьому повністю описувати текстуру в єдиному файлі, що зручно, так як всі допоміжні функції доступні як при відображені так і при оновлені параметрів.

5.2.2 Параметри

В попередньому підрозділі (5.2.1) описувалось оновлення структури параметрів при оновлені вихідного коду (назви змінних, типи змінних, їх послідовність та розмір). На рисунку 5.2 зображено процес оновлення значень параметрів у межах однієї ітерації програми (одного кадру).

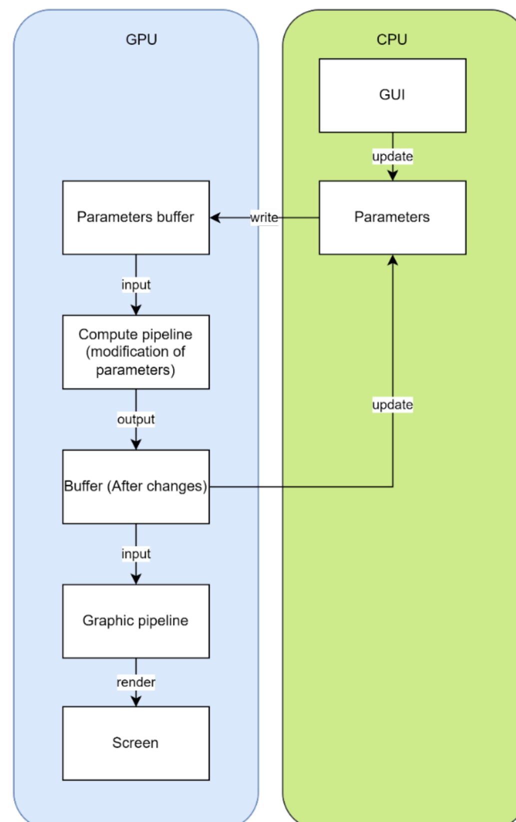

Рисунок 5.2 – Оновлення параметрів (значень) в межах одного кадру

Спершу, за допомогою графічного інтерфейсу виконується зміна параметрів на стороні CPU. Далі змінені параметри копіюються в буфер, що доступний на GPU. Тут виконується обчислювальний шейдер, для оновлення параметрів (що описано функцією `UpdateParameters`), на вхід до якого, поступає буфер параметрів. Після змін, буфер копіюється назад до параметрів на стороні CPU, а сам поступає на вхід графічного конвеєру (тільки для читання) для відображення текстури. Графічний конвеєр же на основі параметрів відображає текстуру на екрані.

5.2.3 Експорт

При експорті бажано отримати єдину функцію, незалежну від зовнішніх функцій та змінних. Процес експорту зображено на рисунку 5.3.

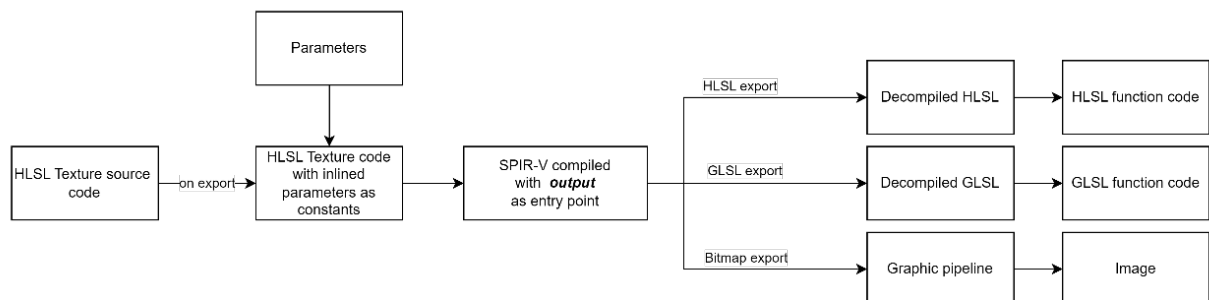

Рисунок 5.3 – Процес експортування текстури

Першим кроком є вбудовування всіх параметрів в код функції, шляхом об’явлення структури типу параметрів та присвоєння кожному з членів значення, що наразі відповідає параметру.

Далі необхідно позбавитись зв’язків від зовнішніх функцій. Найпростішим підходом є компіляція вихідного коду з точкою входу `output` в SPIR-V асемблер з оптимізацією, після чого, декомпіляція назад в HLSL або GLSL (в залежності від обраної опції). При компіляції всі функції неявно отримують модифікатор `inline`, як це визначено в документації HLSL [31].

Це зумовлено тим, що на GPU шейдерні ядра не мають стеку, що притаманний CPU, тому, фактично немає сенсу розділяти код на функції на бінарному рівні. Підхід з компіляцією та декомпіляцією значно спрощує результуючий код та видаляє невикористані фрагменти (див. лістинги 5.1 та 5.2): наприклад, маючи тернарний умовний оператор, умова якого визначається параметром, він буде замінений лише гілкою що має виконуватись, або, наприклад, при обчислені синусу від певного константного значення параметра оператор заміниться магічним числом. Більше того, це дає можливість декомпілювати не лише в HLSL, а й в інші високорівневі мови як GLSL або MSL.

```
float sin2(float val)
{
    return sin(val)*sin(val);
}


float output(float2 uv)
{
    if(p.a>0)
        return sin2(p.a);
    else
         return 0;
}
```

Лістинг 5.1 – Приклад вихідного коду HLSL

```
float generated()
{
    return 0.2383170127868652343 75f;
}
```

Лістинг 5.2 – Приклад коду після компіляції в SPIR-V та декомпіляції в HLSL

Після декомпіляції виконується додаткові модифікації щоб отримати лише код функції і відкинути зайве.

При експортуванні в бітмап декомпіляція вже не відбувається. На основі бінарного шейдерного коду SPIR-V створюється графічний конвеєр та виконується рендеринг в бітмап.

Недолік підходу з компіляцією та декомпіляцією виражений тим, що внутрішні назви змінних, структура коду та коментарі губляться і складний код стає майже нечитабельний, але реалізація зі збереженням імен змінних та структури коду потребує модифікації коду компілятора чи розробки власного інструменту розбору та вбудовування коду функцій. В обох випадках це складна задача в якій необхідно враховувати багато деталей мови HLSL та, при зміні версії мови, замість простої заміни версії компілятора, цей модуль необхідно буде переписувати.

5.3 Вибір програмного забезпечення

Враховуючи необхідність високої оптимізації, необхідно обрати мову програмування, яка дозволить контролювати ресурси комп'ютера та працювати з графічним API на низькому рівні. На сьогоднішній день це C, C++ та Rust. Враховуючи, що C++ має велику кількість бібліотек, серед яких є компілятори шейдерів, та має значно вищий рівень абстракції за C, то їй було надано перевагу.

В якості графічного API обрано Vulkan [6], перевагами якого є кросплатформність та низький і детальний контроль над усіма ресурсами, що дозволяє більш детально описати роботу спеціалізованої програми.

В якості віконної системи обрано SDL [32], через підтримку багатьох платформ.

Інтерфейс реалізовано з використанням бібліотеки Dear ImGui [33], що відносно легко інтегрується з графічним API.

Для компіляції шейдерного коду використовується бібліотека shaderc [34], тоді як бібліотека SPIRV-Cross [35] застосовується для декомпіляції та отримання рефлексії з бінарного коду.

Допоміжні бібліотеки:

1) nlohmann [36] – використовується для роботи з json файлами, необхідна для збереження текстур та поточного стану програми;

2) stb [37] – для збереження растрових зображень різних форматів;

3) nativefiledialog-extended [38] – кросплатформовий доступ до нативного файлового діалогу, бібліотека необхідна для відкриття та збереження текстур і зображень;

4) ImGuiColorTextEdit [39] – редактор тексту з підсвіткою, що базується на Dear ImGui.

Для автоматизації збірки використовується CMake [40]. CMake – це інструмент із відкритим вихідним кодом, який використовує незалежні від компілятора та платформи конфігураційні файли для створення власних файлів інструментів збірки, специфічних для компілятора та платформи. CMake використовується для керування процесом компіляції програмного забезпечення за допомогою простих конфігураційних файлів, незалежних від платформи та компілятора.

5.4 Програмна Реалізація

На рисунку 5.4 зображено взаємозв’язок розроблених програмних модулів та зовнішніх бібліотек.

Основний функціонал роботи з графічним API винесено в бібліотеку vkbase. Тут реалізовано керування об’єктами Vulkan, необхідними для роботи програми, а також керування процесом рендерингу з можливістю вставляти користувацькі події. Наразі в якості віконної бібліотеки vkbase спирається на SDL, але функціонал роботи з SDL винесено в окремий файл і легко може бути замінений іншою бібліотекою.

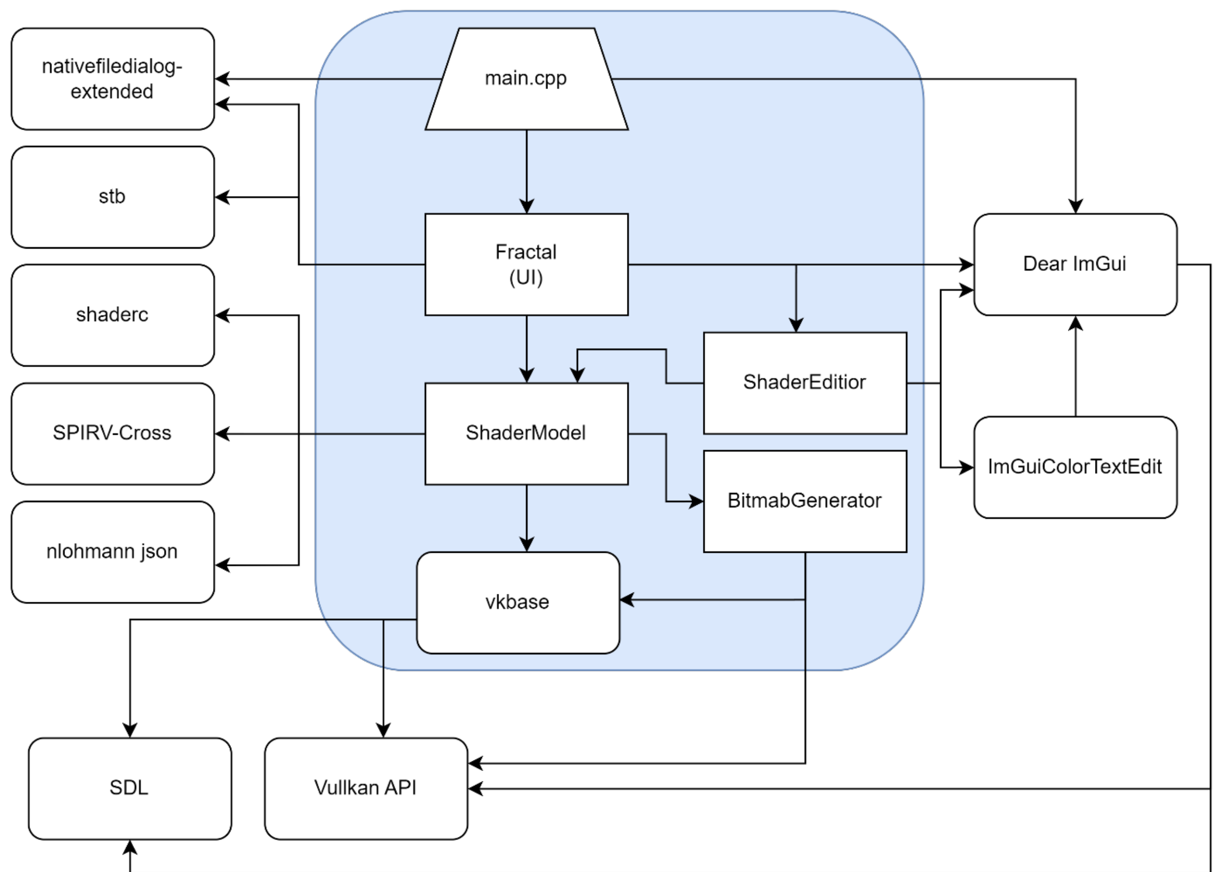

Рисунок 5.4 – Взаємозв’язок основних програмних модулів (на синьому фоні розроблені компоненти, за межами – сторонні бібліотеки)

Клас ShaderModel реалізує майже весь функціонал роботи з текстурою. Тут реалізовано компіляція коду, експортування текстури, оновлення параметрів, візуалізація текстури за допомогою Vulkan, серіалізація та збереження текстури у json. Функціонал генерації бітмабу для зручності винесено в клас BitmabGenerator.

З метою розділення інтерфейсу і основної логіки, інтерфейс для редагування текстури винесено до класу Fractal, що є огорткою над класом ShaderModel і утворює композиційний зв’язок. Fractal також передає введення з клавіатури до ShaderModel для подальшого доступу з шейдеру.

Файл main.cpp містить точку входу програми. Тут виконується ініціалізація всіх ресурсів. Також відповідає за спільний інтерфейс: меню,

роботу вкладок та перемикання між робочими текстурами, керування відкриттям текстур з файлової системи і створенням нових.

5.5 Інструкція з використання

5.5.1 Загальний інтерфейс

Графічний інтерфейс складається з меню та робочого простору, який в свою чергу поділяється за допомогою вкладок, кожна з яких відповідає одній з відкритих в даний момент текстур. Весь простір вікна призначено для відображення текстури, поверх якого знаходяться інтерфейс (рис. 5.5).

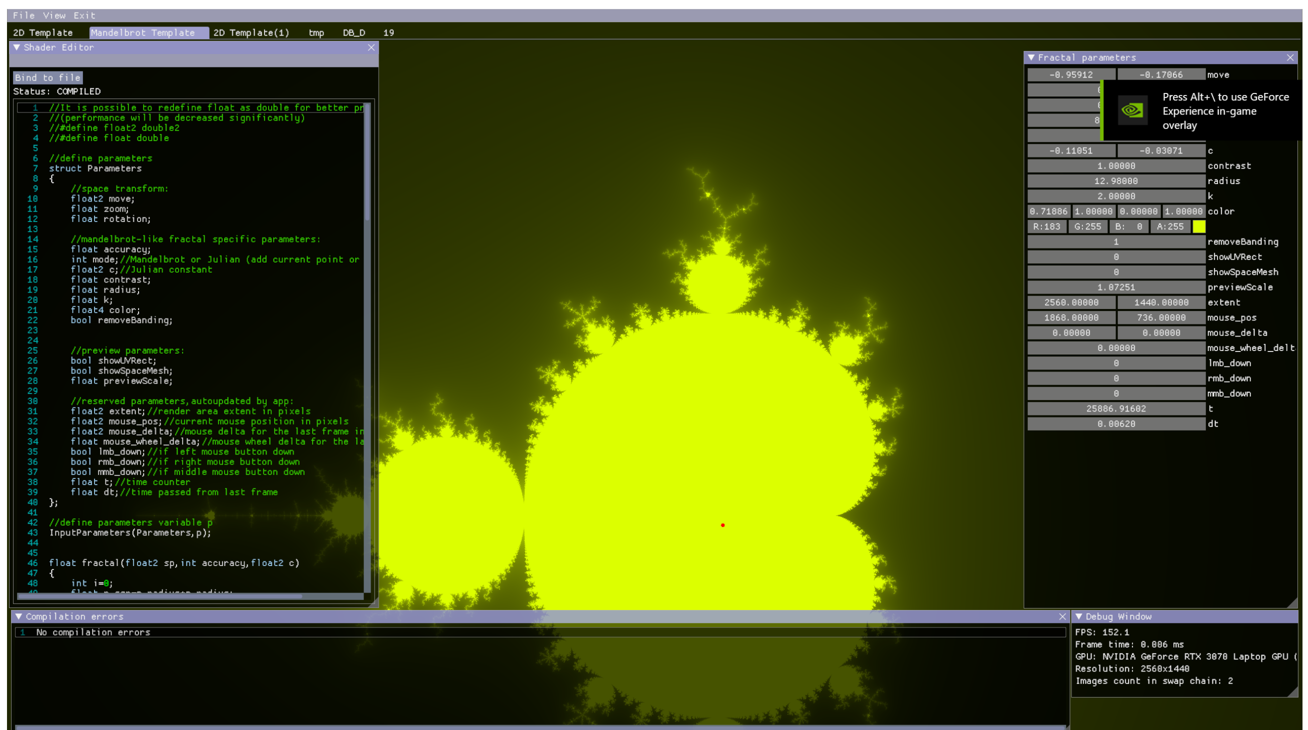

Рисунок 5.5 – Загальний інтерфейс редактора

За допомогою меню можна керувати створенням, збереженням, відкриттям та закриттям текстур.

Для створення нової текстури необхідно обрати один із шаблонів з певної групи (рис. 5.6).

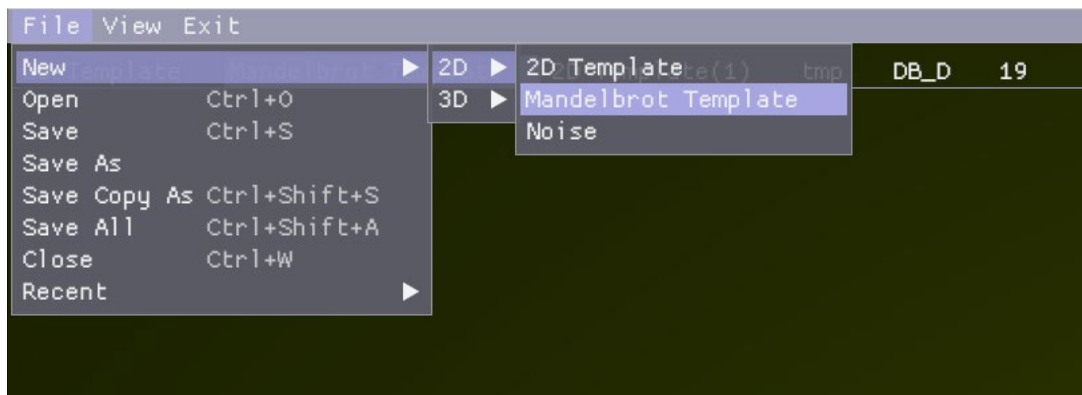

Рисунок 5.6 – Підменю «File/New»

Через підменю «Recent» можна відкрити один з нещодавніх файлів (рис. 5.7).

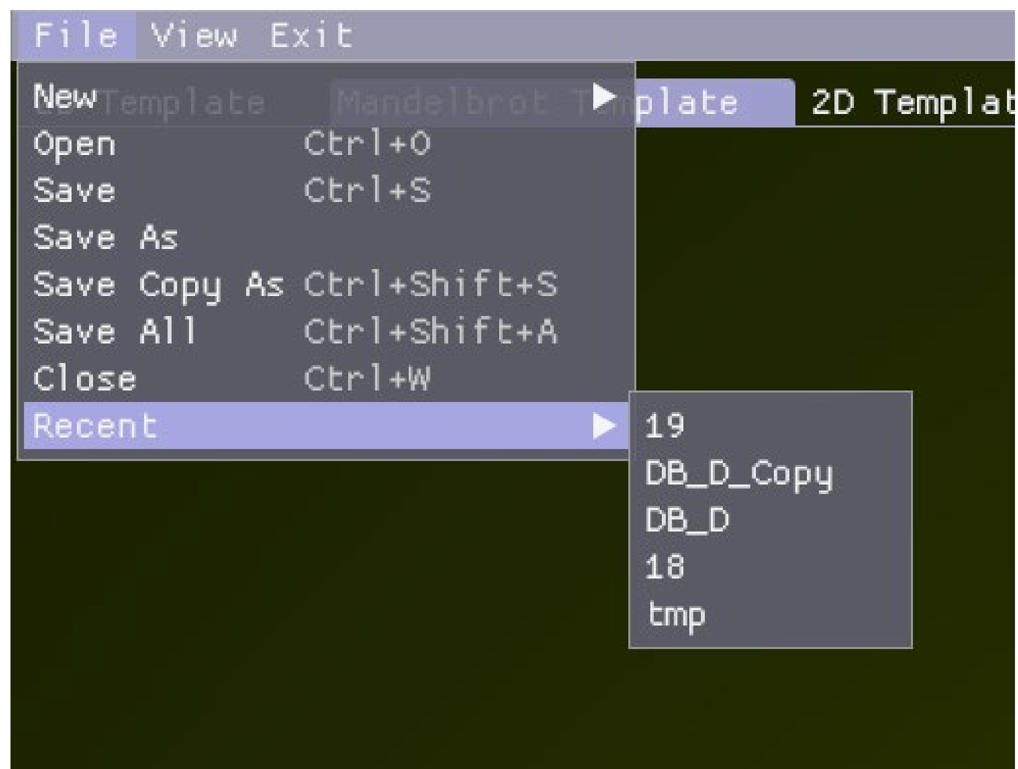

Рисунок 5.7 – Підменю «File/Recent»

Також за допомогою меню можна керувати відображенням дочірніх вікон, скривати весь інтерфейс та переключатися між віконним та повноекранним режимами (рис. 5.8).

Рисунок 5.8 – Підменю «View»

Інтерфейс складається з таких основних дочірніх вікон:

1) редактор коду;

2) редактор параметрів;

3) вікно з інформацією про статус компіляції та можливі помилки;

4) вікно експорту.

5.5.2 Редагування текстури

Редактор коду (рис. 5.9) представляє собою доволі простий вбудований редактор вихідного коду текстури (наразі тільки на мові HLSL). Зверху відображається статус компіляції («COMPILING…» , «COMPILED», «ERROR»). Так як функціонал вбудованого текстового редактору обмежений, за необхідності, можна прив'язати вихідний код до файлу і редагувати його в зручній середі. Для цього необхідно натиснути кнопку «Bind to file» та обрати місце у файловій системі для створення файлу. Оновлення коду виконується автоматично при збереженні файлу, як і всі зміни у вбудованому редакторі відображатимуться у файлі.

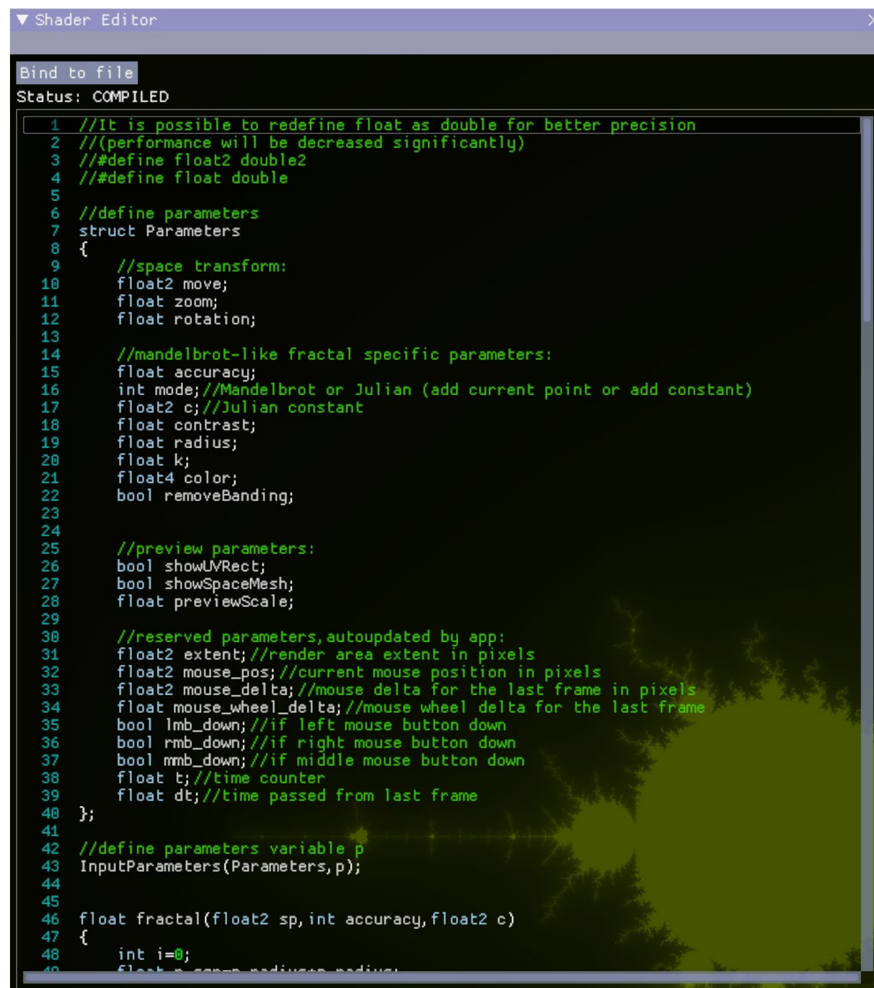

Рисунок 5.9 – Вікно редактора коду

Оновлення графічного відображення текстури відбувається одразу при зміні вихідного коду і займає в середньому до пів секунди. Помилки в коді підсвічуються червоним у редакторі (рис. 5.10), а також виводяться у окремому вікні (рис. 5.11).

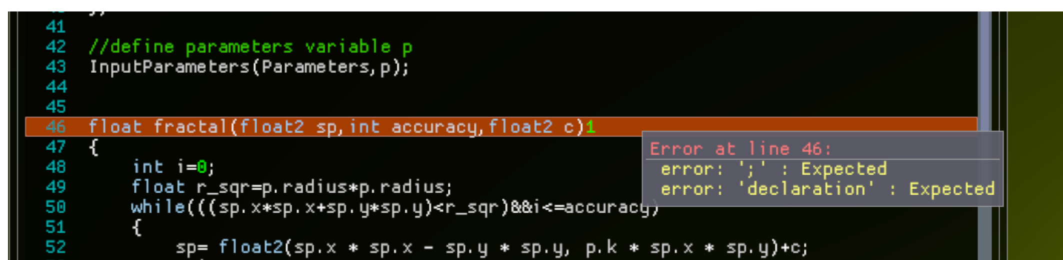

Рисунок 5.10 – Підсвічення помилки у редакторі

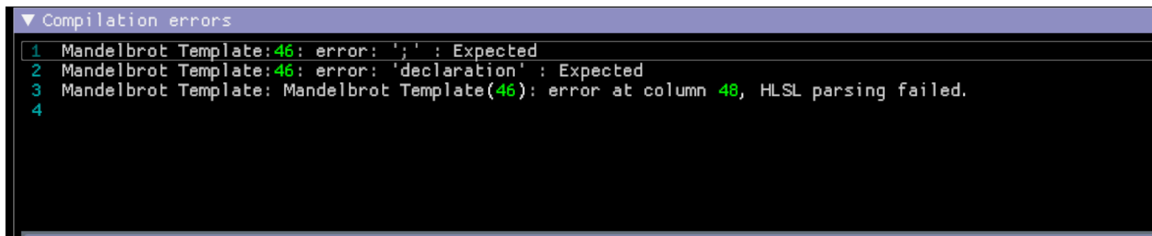

Рисунок 5.11 – Виведення повної інформації про помилку у окремому вікні

Код текстури базується на стандарті HLSL з незначними доповненнями. Код, що описує текстуру має складатися з трьох точок входу (основних функцій) та структури, що представляє спільні параметри.

Для об’явлення спільних параметрів використовується директива `InputParameters`, яка приймає тип структури та назву змінної (лістинг 5.3).

Всі параметри об’явлені в якості членів структури будуть автоматично відображені у вікні редактора параметрів, де їх можна редагувати (рис. 5.12). Наразі в якості параметрів підтримуються наступні типи: `int`, `uint`, `float`, `double`, `bool`, `int2`, `int3`, `int4`, `uint2`, `uint3`, `uint4`, `float2`, `float3`, `float4`, `double2`, `double3`, `double4`, `bool2`, `bool3`, `bool4`.

Параметри з наступними назвами будуть оновлюватись автоматично:

1) `extent` – розмір доступної області;

2) `mouse_pos` – координати курсора;

3) `mouse_delta` – зміна позиції курсора з часу попереднього оновлення;

4) `mouse_wheel_delta` – зміна прокрутки колеса миші з часу попереднього оновлення;

5) `lmb_down` – чи ліва кнопка миші натиснута;

6) `rmb_down` – чи права кнопка миші натиснута;

7) `mmb_down` – чи середня кнопка миші натиснута;

8) `t` – лічильник часу в секундах;

9) `dt` – час з попереднього оновлення в секундах.

```
struct Parameters
{
	//space transform:
	float2 move;
	float zoom;
	float rotation;


	//mandelbrot-like fractal specific parameters:
	float accuracy;
	int mode;
	float2 c;// constant
	float contrast;
	float radius;
	float k;
	float4 color;
	bool removeBanding;


	//preview parameters:
	bool showUVRect;
	bool showSpaceMesh;
	float previewScale;


	//reserved parameters,autoupdated by app:
	float2 extent;//render area extent in pixels
	float2 mouse_pos;//current mouse position in pixels
	float2 mouse_delta;//mouse delta for the last frame in
pixels
	float mouse_wheel_delta;//mouse wheel delta for the
last frame
	bool lmb_down;//if left mouse button down
	bool rmb_down;//if right mouse button down
	bool mmb_down;//if middle mouse button down
	float t;//time counter
	float dt;//time passed from last frame
};

//define parameters variable p
InputParameters(Parameters,p);
```

Лістинг 5.3 – Приклад об’явлення структури параметрів

При збереженні, значення параметрів також зберігаються разом з кодом текстури.

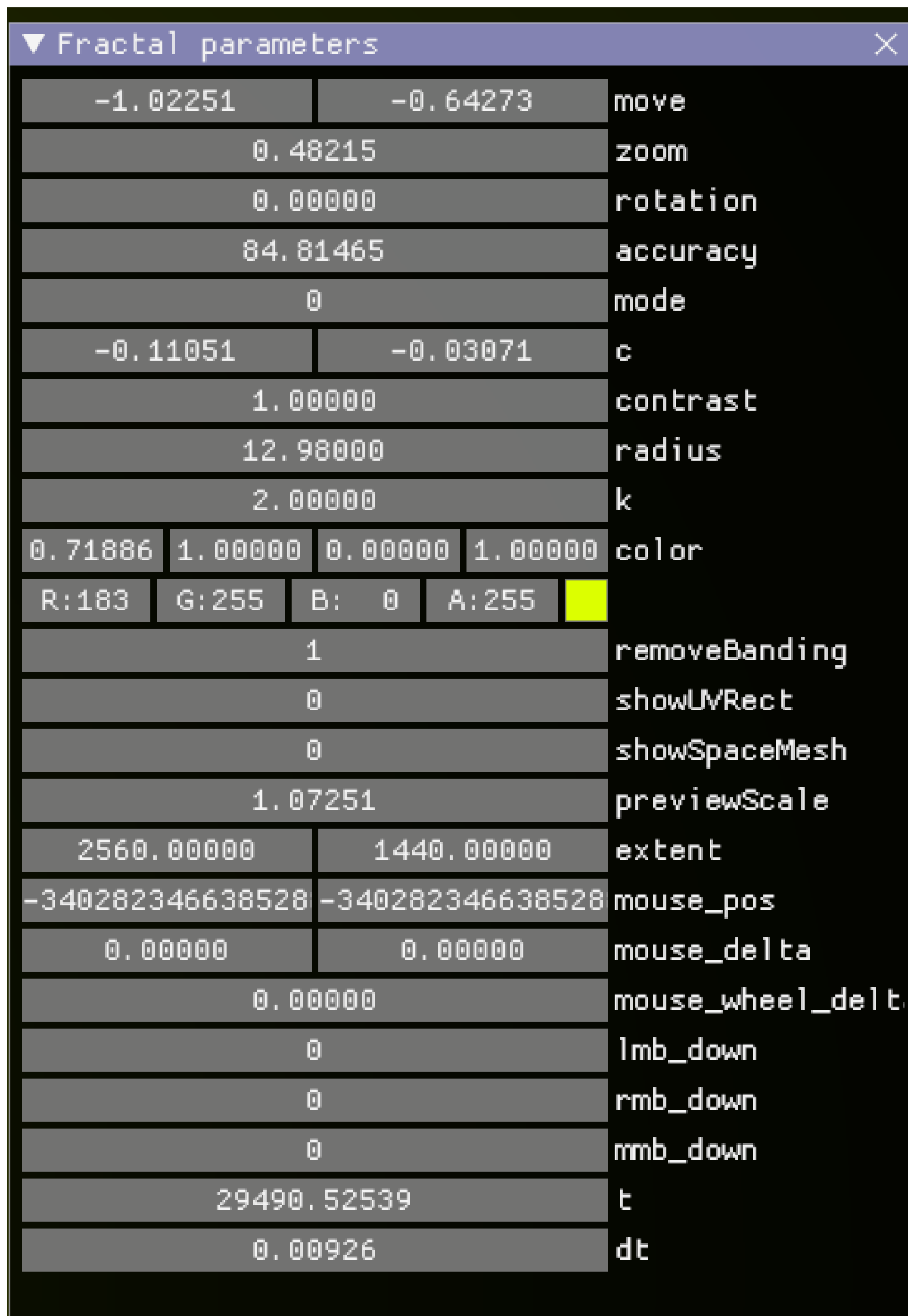

Рисунок 5.12 – Вікно для керування параметрами

Далі перераховано точки входу, що повинні бути присутні в коді:

`output` – функція, що буде використовуватись при експортуванні текстури (фактично описує саму текстуру), хоча може описувати не лише

текстуру, а будь-яку функцію. Може приймати довільні вхідні параметри довільного типу так як і повертати параметр довільного типу.

main – точка входу, що використовується для перегляду output у редакторі під час розробки текстури. Викликається для кожного пікселя з регіону де відображається текстура. На вхід приймає координати точки з регіону типу float2 у діапазоні [-1,1] і повертає вектор кольору (float4). main необхідна для «представлення» output на екрані. Таке розділення створено з метою надання можливості гнучкого відображення. Наприклад, тут можна домалювати допоміжний інтерфейс: координатну сітку, виділення регіону, фон. Також є можливість виконати додаткове перетворення координат з врахуванням співвідношення сторін та масштабування. Якщо функція output повертає не колір або приймає не двовимірний вектор, то задача main перетворити параметри, таким чином, щоб відобразити їх на екрані. Це може бути корисним при створенні тривимірних текстур або функцій однієї змінної. Також у main результат функції може бути модифіковано з метою представлення результату за допомогою кольору, якщо output повертає скалярне значення. Функція main жодним чином не впливає на результат експорту, і необхідна лише для зручного відображення результату.

UpdateParameters викликається один раз перед візуалізацією кожного з кадрів та дозволяє програмно змінювати параметри. Тут доступні три додаткові функції, реалізація яких скрита та які дозволяють відстежувати натискання клавіш:

1) bool isKeyPressed(int key) – перевіряє чи була клавіша натиснута з останнього оновлення;

2) bool isKeyReleased(int key) – перевіряє чи була клавіша відпущена з останнього оновлення;

3) bool isKeyDown(int key) – перевіряє чи натиснута клавіша зараз.

Об’явлення всіх доступних кодів клавіш наведено в додатку А.

5.5.3 Динамічні та статичні параметри.

При створенні текстури важливо розрізняти динамічні та статичні параметри. Статичні параметри при експортуванні замінюються константними значеннями, тоді як динамічні мають залишитися в якості змінних щоб текстурою можна було керувати динамічно.

Статичними вважаються параметри, що напряму представлені структурою, яка об’явлена в якості спільних параметрів (за допомогою директиви `InputParameters`). Статичні параметри будуть замінені константними значеннями. Якщо необхідно щоб певний параметр був динамічним, його необхідно передати через вхідний параметр функції `output`.

Щоб керувати динамічним параметром під час редагування текстури в редакторі необхідно додати змінну до загальних параметрів та передати її до функції `output` через `main`.

Наприклад, якщо змінна параметрів об’явлена як `p`, тоді для наступного коду (лістинг 5.4) `p.t` є статичним параметром, а `d_t` – динамічним. Результат експорту в HLSL виглядатиме як у лістингу 5.5.

```
struct Parameters
{
    float t;
}

InputParameters(Parameters, p);

float4 output(float2 uv, float d_t)
{
     return float4(sin(p.t),sin(d_t),0,1);
}

float4 main(float2 position)
{
     return output(position,p.t);
}
```

Лістинг 5.4 – Статичний та динамічний параметр

```
    float4 generated(float d_t)
    {
        return float4(0.849223256111450195312 5f, sin(d_t),
0.0f, 1.0f);
    }
```

Лістинг 5.5 – Експорт з динамічним параметром `d_t`

У прикладі динамічний параметр `uv` не використовується функцією `output`, тому він був втрачений. До речі, це важливо, щоб координата точки з простору текстури – `uv` була динамічним параметром, оскільки результат від функції текстури має залежати від координат точки, інакше вся текстура буде «залита» одним кольором (або значенням іншого типу, в залежності від типу текстури).

5.5.4 Експортування текстури

Експортування виконується за допомогою вікна «Export» (рис. 5.13), яке можна відкрити через меню або за допомогою комбінації клавіш Ctrl+Q.

Для експорту доступні три варіанти:

1) HLSL;
2) GLSL;
3) Bitmap.

Для генерації коду необхідно обрати HLSL або GLSL та натиснути «Export». Після натискання буде згенеровано відповідний код у вигляді функції, який тепер можна виділити та скопіювати або скористатися кнопкою «Copy to clipboard». Маркер «only body» дозволяє згенерувати лише тіло функції.

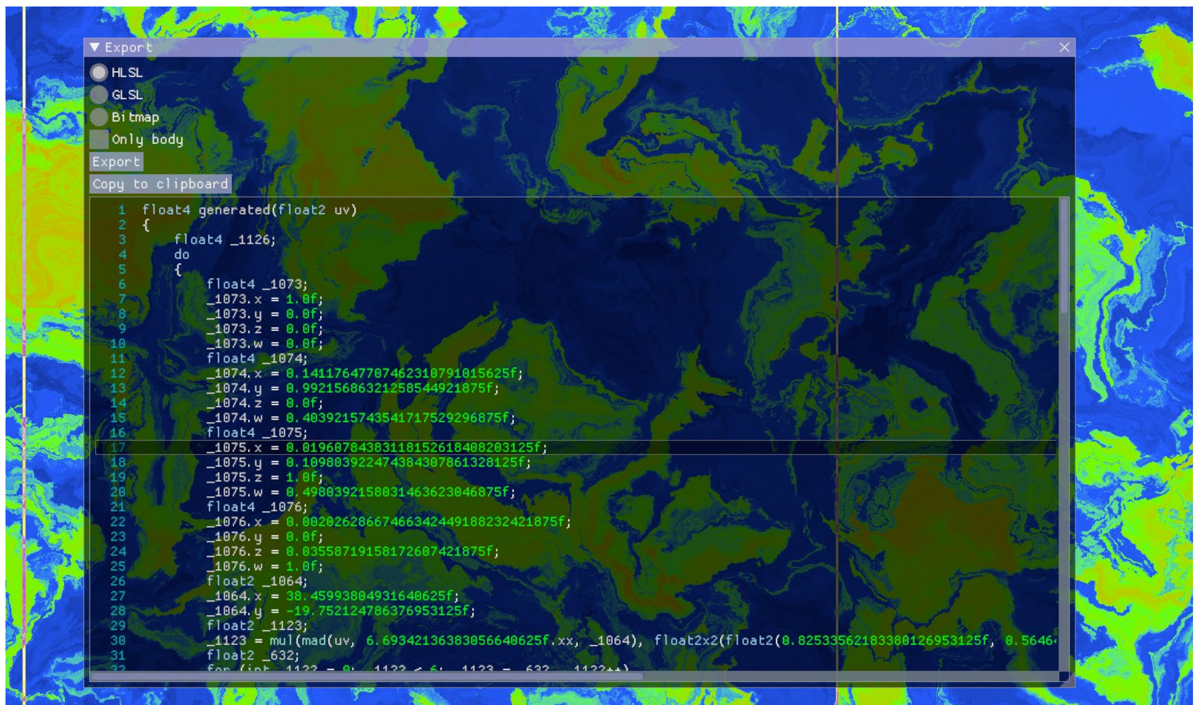

Рисунок 5.13 – Вікно експорту текстури

Для генерації растрового зображення необхідно обрати «Bitmap», ввести розмір зображення та виклик функції (рис. 5.14). Виклик функції представляє HLSL вираз, результат якого (має бути `float4`) записується у відповідний піксель зображення. Функція, що представляє текстуру має назву `generated` та відповідає функції згенерованій при HLSL експорті. Під час виклику функції доступний параметр `uv`, який представляє відносне положення пікселя на зображенні типу `float2` в діапазоні 0-1. Під час виклику можна зробити додаткове розтягування та встановити інші динамічні параметри.

Зображення генерується після натискання «Generate». Після генерації (рис. 5.15) його можна або зберегти або скопіювати в буфер обміну. Також після генерації відображається час рендерингу, що фактично виражає скільки часу займав безпосередньо процес обчислення для визначеного розміру текстури (без врахування часу підготовки), тобто це той час, що займатиме графічний процесор відобразити цю текстуру у зазначеному розмірі в умовах використання текстури як динамічної.

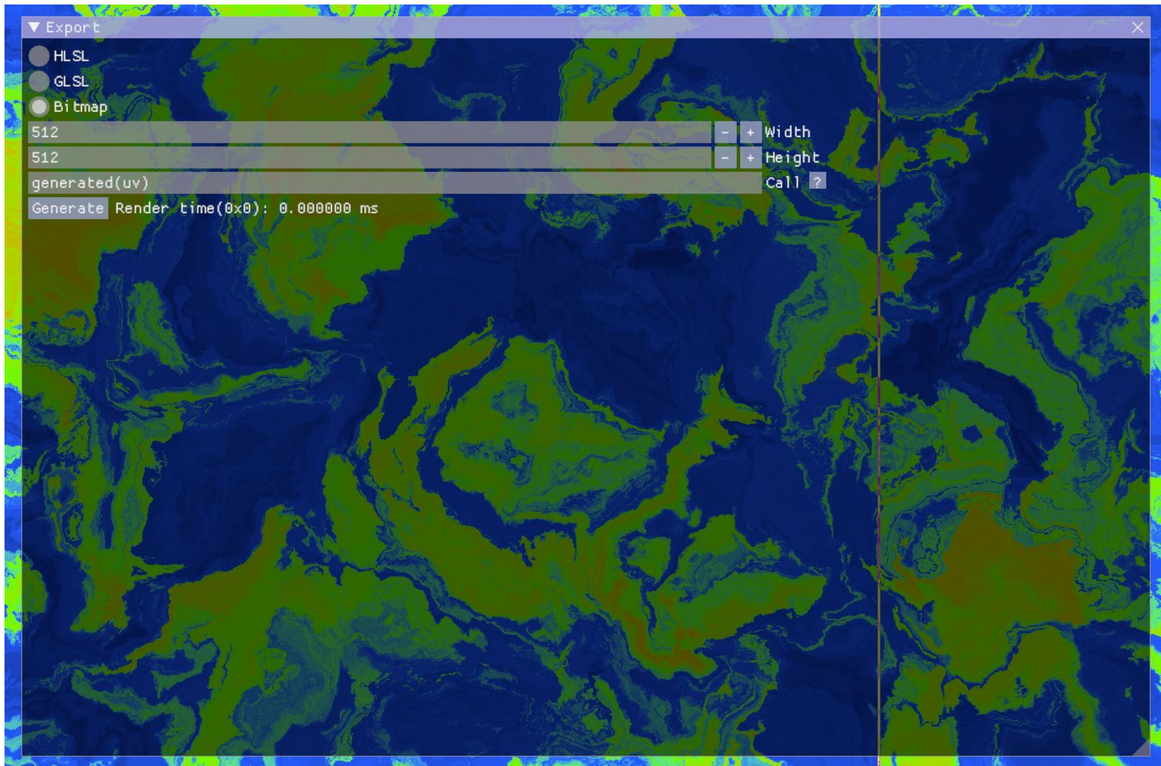

Рисунок 5.14 – Експорт в растрове зображення

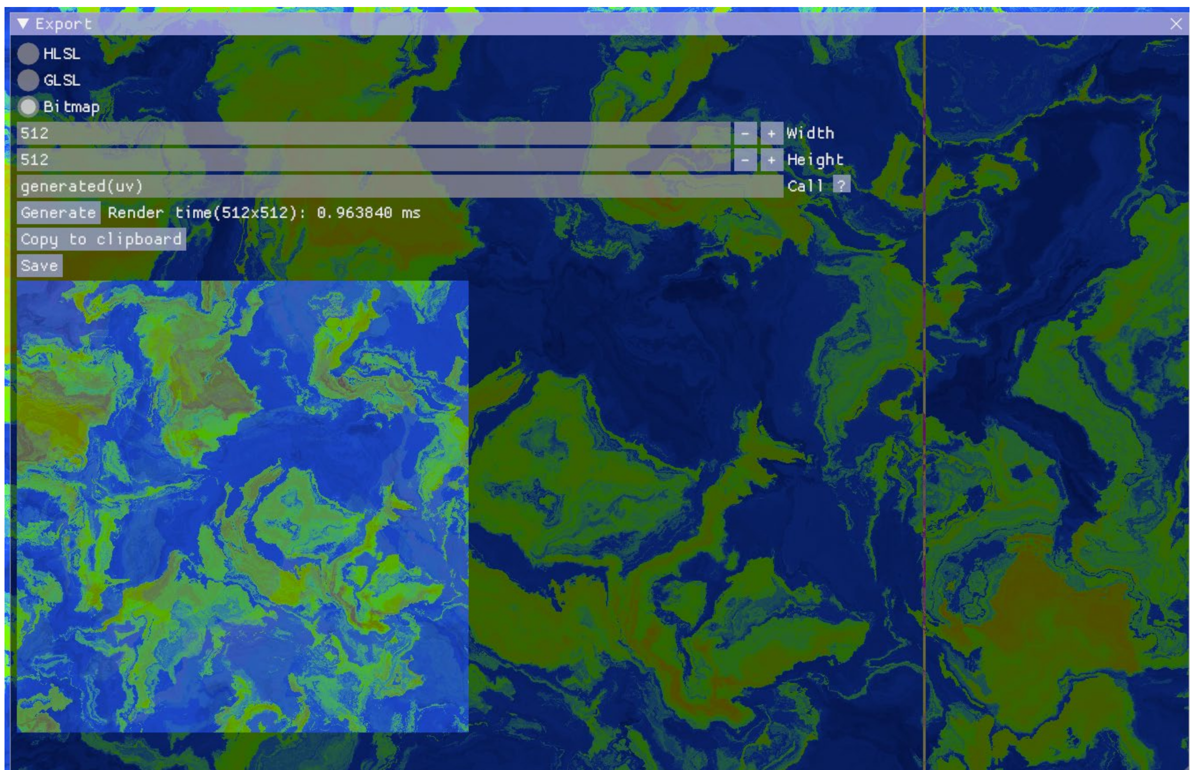

Рисунок 5.15 – Згенероване растрове зображення

6 СТВОРЕННЯ ДИНАМІЧНИХ ФРАКТАЛЬНИХ ПРОЦЕДУРНИХ ТЕКСТУР ЗА ДОПОМОГОЮ РЕДАКТОРА

6.1 Реалізація відомих фрактальних множин

Основним критерієм до фракталів, що можуть бути використанні в якості динамічної текстури є можливість отримати результат (належність до множини, неналежність, або деяке проміжне значення) за координатою точки у просторі. Серед фракталів, алгоритм обчислення яких базується саме на знаходженні належності точки до множини, добре відомими є фрактали (або сімейства фракталів): Множина Мандельброта, Множина Жуліа, Палаючий корабель, Фрактали Ляпунова, Фрактал Ньютона.

Треба відмітити, що обчислення фракталів (зокрема перерахованих вище) є процесом нескінченним, та визначається рекурентним відношенням, тому при програмній реалізації знаходиться лише наближення з певною мірою точності до множини фракталу, як правило, досягається перериванням рекурсивного обчислення після визначеної кількості ітерації.

6.1.1 Множина Мандельброта

Множина Мандельброта визначається за наступним правилом: до множини належать такі точки x, що для послідовності комплексних чисел z:

$$z_0 = x$$

$$z_{n+1} = z_n^2 + x,$$

$$\forall\, n \text{ виконується умова: } |z_n| < 2$$

Множина знаходиться в межах кола радіусом 2.

Реалізація у редакторі наведена в лістингу 6.1. Результуюче зображення на рисунку 6.1.

```
float fractal(float2 x)
{
    int i=0;
    float2 z=x;

    while(((z.x*z.x+z.y*z.y)<4)&&i<=p.accuracy)
    {
        z= float2(z.x * z.x - z.y * z.y, 2 * z.x * z.y)+x;
        ++i;
    }

    if(i>=p.accuracy)
        return 1;

    return 0;
}
```

Лістинг 6.1 – код HLSL функції для обчислення множини Мандельброта

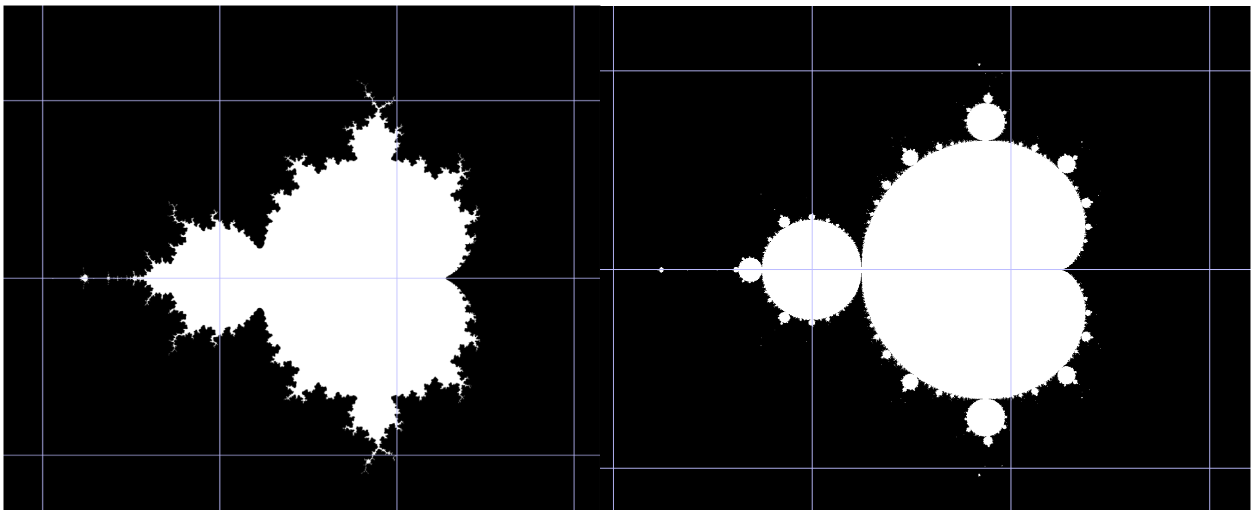

Рисунок 6.1 – Множина Мандельброта, точність 16 (зліва) та 200 (справа)

При генерації зображення також можна відображати точки, що не належать до множини, але близькі до неї у градації, що буде відповідати тому як швидко точка виходить за межі кола радіусом 2. Для цього достатньо визначати скільки ітерацій, що знадобилась для виходу точки за межі кола відносно загальної кількості ітерацій (рис. 6.2).

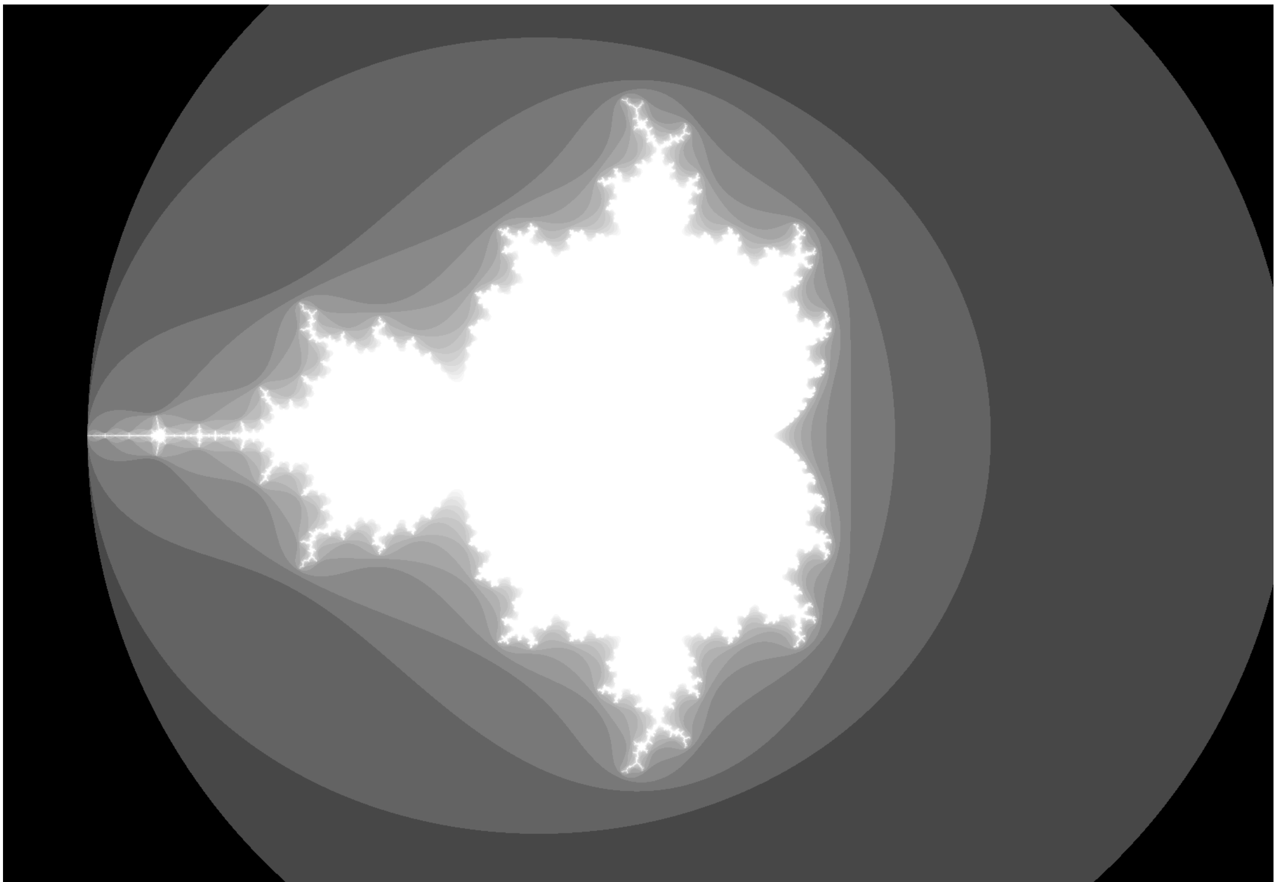

Рисунок 6.2 – Множина Мандельброта з підсвіченням зовнішніх шарів

Так як кількість ітерацій `i` – ціле значення, то регіони з різною кількістю ітерацій чітко виражені. Цього можна позбавитись якщо ввести корекцію [41], що враховуватиме як далеко послідовність z вийшла за межі:

$$i = i + 1 - \log_2 \log(|z|)$$

Для цього додано логічний параметр `removeBanding` щоб включати і виключати корекцію. Точність цієї корекції залежить від того, який радіус кола, вихід за який перериває цикл. Тому значення квадрату радіуса підвищено до 20 (лістинг 6.2). Результат після корекції на рисунку 6.3.

```
float fractal(float2 x)
{
    int i=0;
    float2 z=x;

    while(((z.x*z.x+z.y*z.y)<20)&&i<=p.accuracy)
    {
        z= float2(z.x * z.x - z.y * z.y, 2 * z.x * z.y)+x;
        ++i;
    }

    if(i>=p.accuracy)
      return 1;

    float val=i;
    if(p.removeBanding)
        val=clamp(val-log(log(length(z)))/log(2)+1,0,
p.accuracy);
     return val/p.accuracy;
}
```

Лістинг 6.2 – Алгоритм з корекцією для виключення чітких переходів

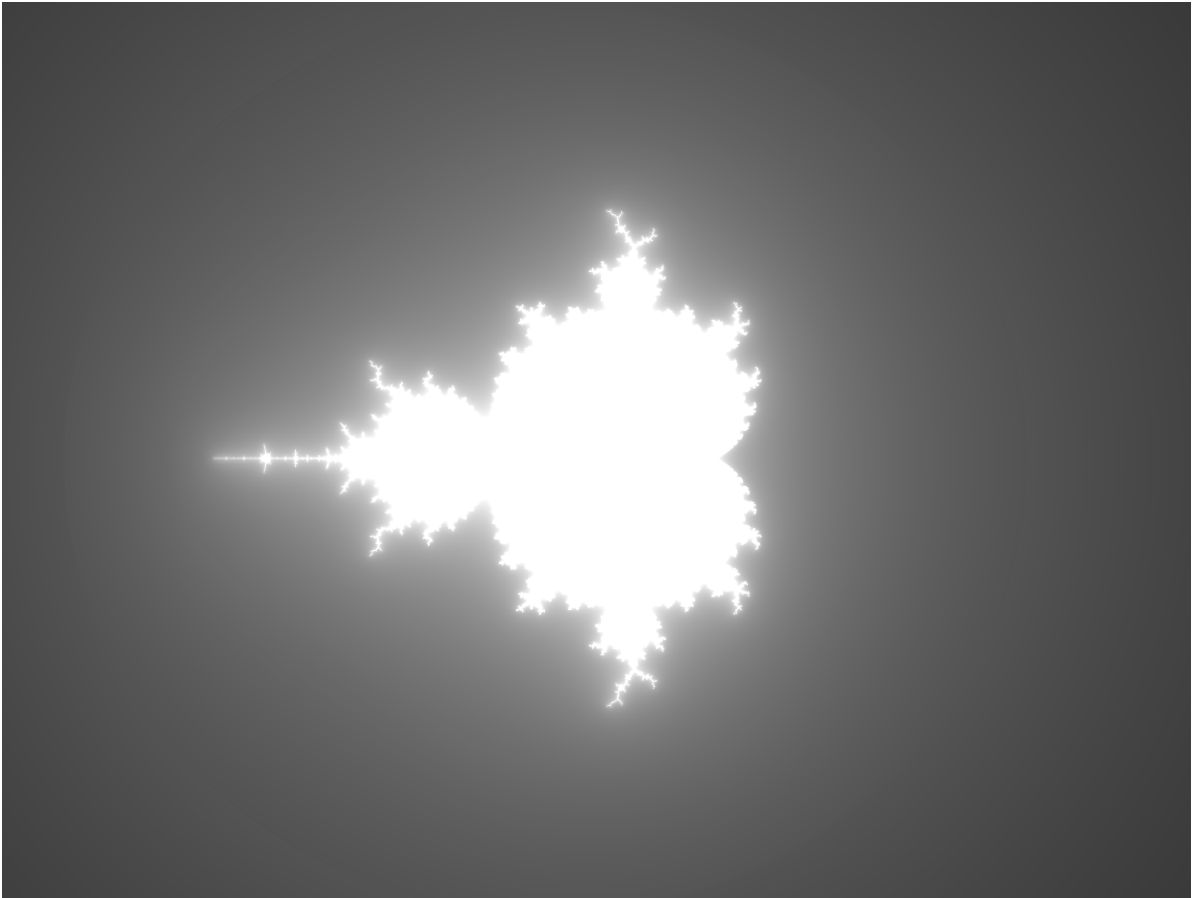

Рисунок 6.3 – Увімкнена корекція (Точність 16)

Для керування контрастом додано параметр `float contrast`, що додає нелінійне перетворення у вигляді степеневої функції, що в коді виглядає так:

```
return pow(val/p.accuracy, p.contrast);
```

Деякі фрагменти згенерованого фракталу зображено на рисунках (рис. 6.4, 6.5, 6.6), до яких включено вікно з поточними параметрами. Важливими параметрами є `move` і `zoom`, що вказують зміщення центу у просторі фракталу та коефіцієнт масштабування відповідно.

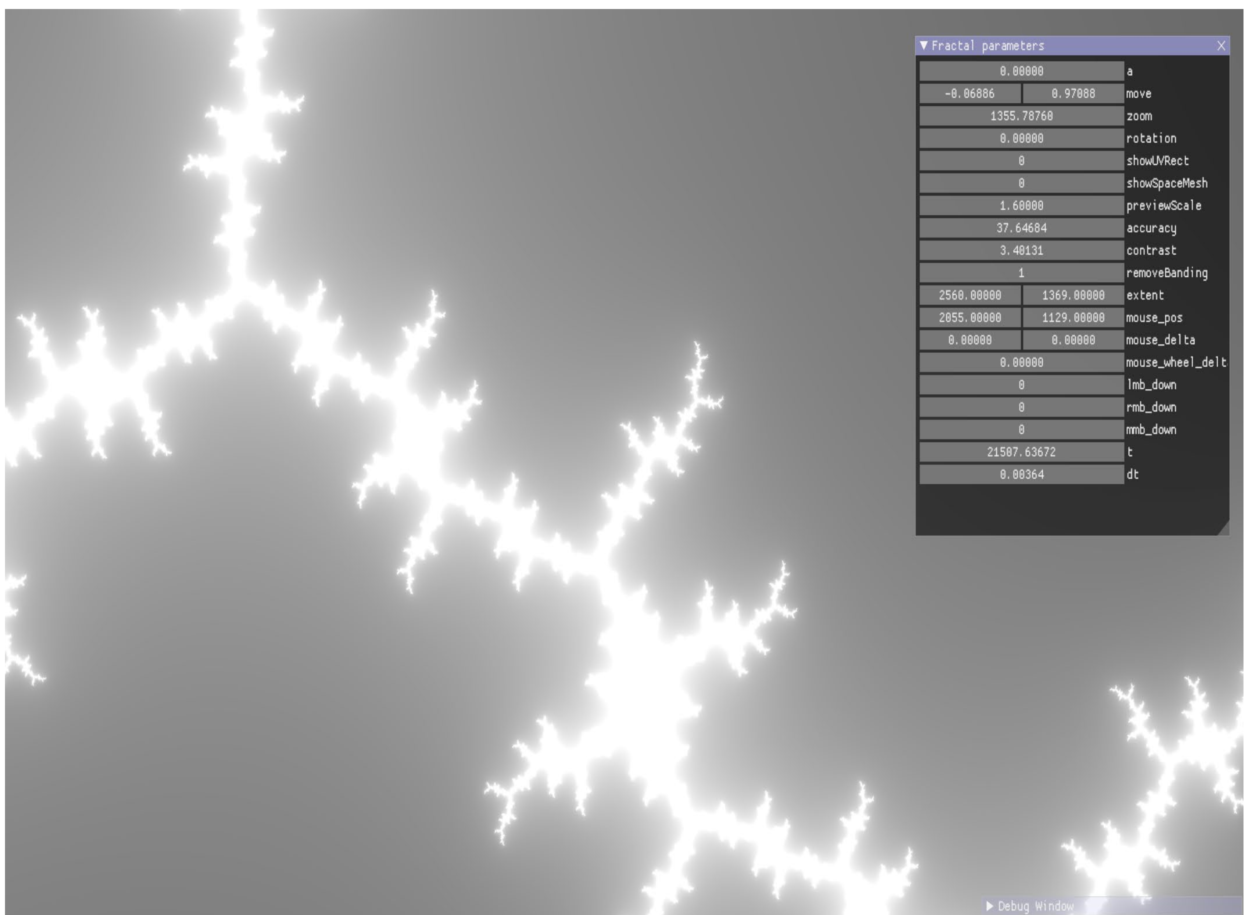

Рисунок 6.4 – Фрагмент множини Мандельброта

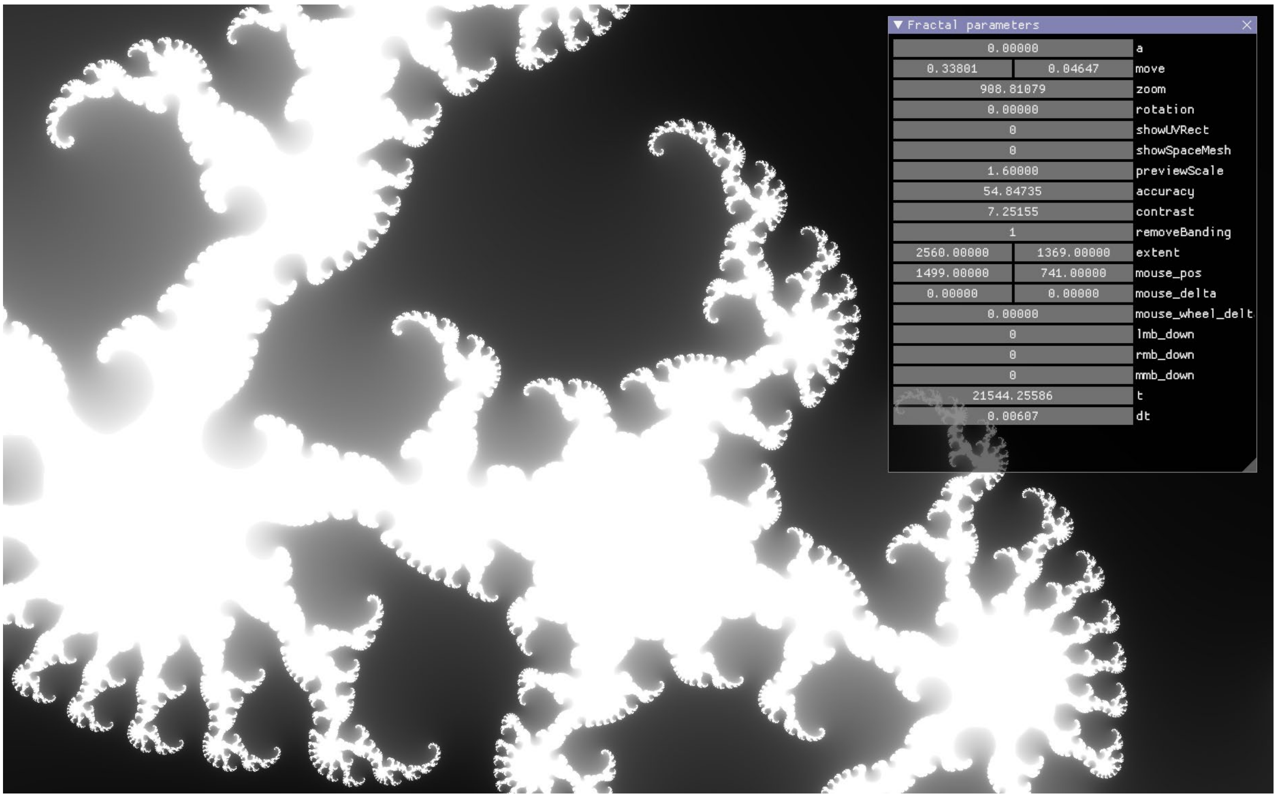

Рисунок 6.5 – Фрагмент множини Мандельброта

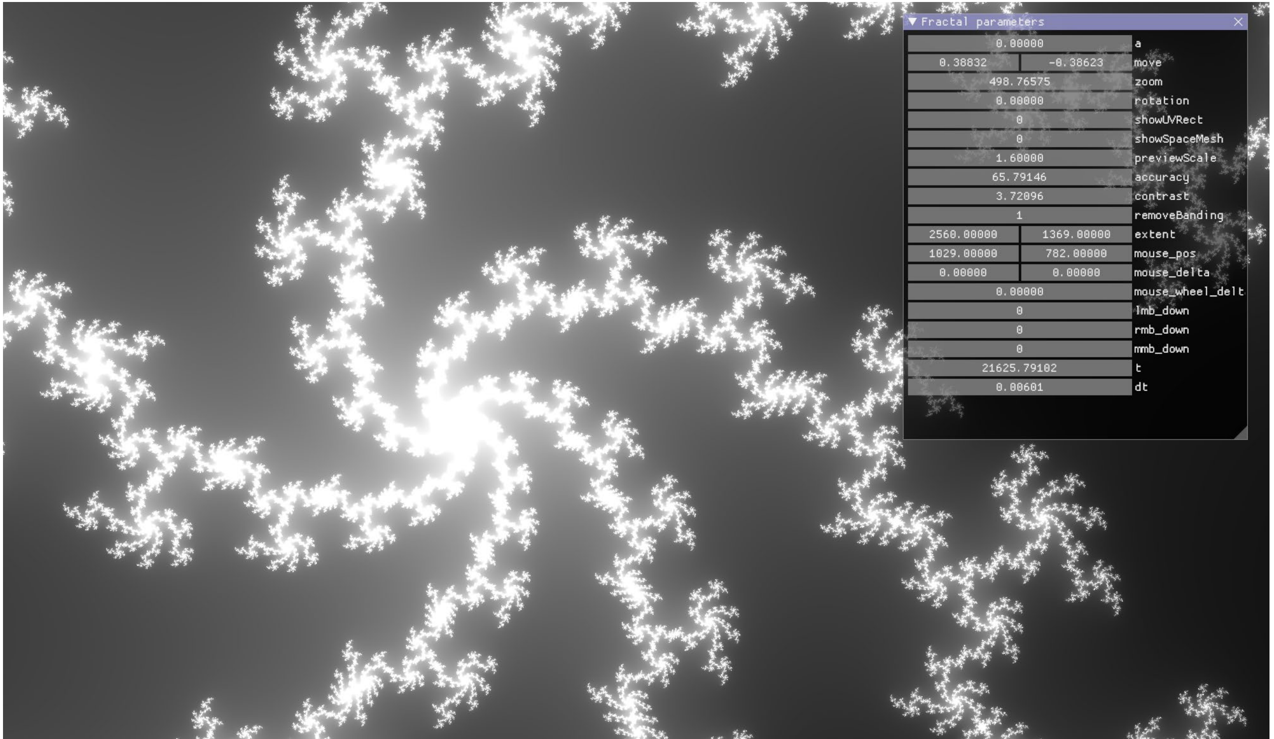

Рисунок 6.6 – Фрагмент множини Мандельброта

6.1.2 Множина Жуліа

Множина Жуліа визначається за правилом дуже схожим на множину Мандельброта:

$$z_0 = x$$
$$z_{n+1} = z_n^2 + c,$$
$$\forall\, n \text{ виконується умова: } |z_n| < 2$$

Замість додавання самої точки x , на кожній ітерації додається константне значення c.

Отже, Множина Мандельброта складається з таких точок x , що належать Множині Жуліа з константою $c = x$.

Для точок, що знаходяться близько до точки c, множина Мандельброта має схожу структуру до множини Жуліа побудованої за цією константою c. Чим менше різниця $x - c$ тим більша подібність. Це дуже корисно при створенні текстур, так як можна знайти необхідний фрагмент в множині Мандельброта та, на основі точки цього фрагменту, згенерувати відповідну множину Жуліа, що буде цілком схожа на цей фрагмент. Це корисно тому, що замість того щоб використовувати фрагмент Мандельброта в масштабі можна використовувати цілий фрактал Жуліа без масштабування, що дає більшої точності (враховуючи обмеження типу `float`).

Для реалізації можна модифікувати вже створену функцію `fractal` так щоб вона приймала константне значення в якості параметра та налаштувати перемикання між ними у `output`.

Далі додано параметри `c`, `mode` та два кольори `julia_color`, `mandelbrot_color`. За допомогою `mode` буде визначатися який тип фракталу відображати: Жуліа, Мандельброта чи обидва. Для чіткого розрізнення фракталів при одночасному відображенні використовуються різні кольори. Точка `c` встановлюється за допомогою натискання правої клавіші миші на

позицію курсора, тоді як для перемиканням між режимами треба натиснути клавішу F. Якщо F натиснута то обидва типи відображаються одночасно, що зручно для пошуку правильної константи множини Жуліа. Точка *c* відображається як червона точка, що домальовується в `main`.

На рисунках (6.7, 6.8, 6.9) зображено обидва фрактали (червоною точкою виділена константа Жуліа), фрагмент збільшений приблизно в 13000 раз. Досить очевидно, що обидва фрактали мають майже однакову структуру в даній точці.

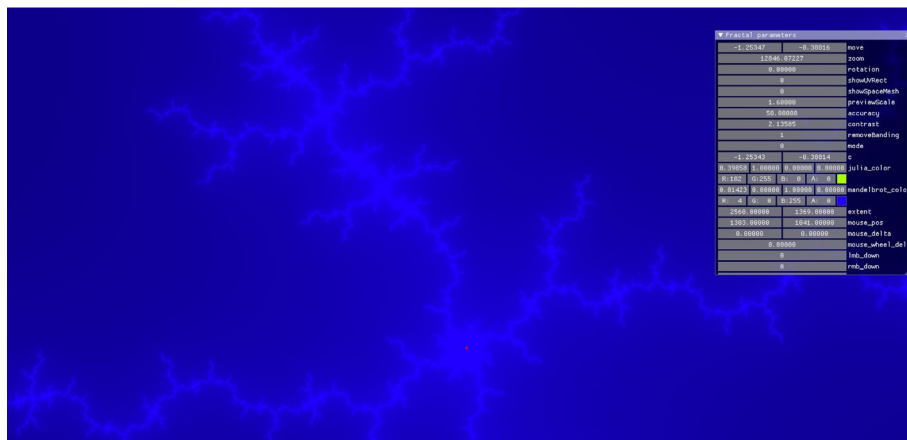

Рисунок 6.7 – Фрактал Мандельброта біля точки c = (-1.25347,-0.38816) при збільшенні в 12846 раз

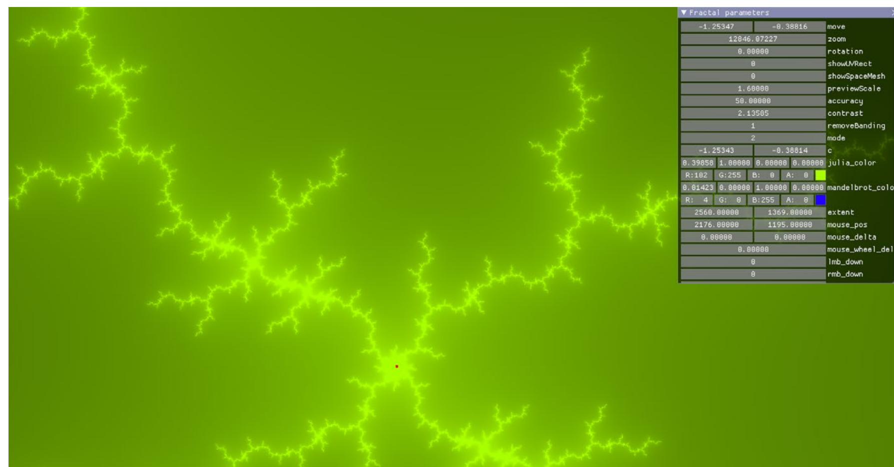

Рисунок 6.8 – Фрактал Жуліа біля точки c= (-1.25347,-0.38816) при збільшенні в 12486 раз

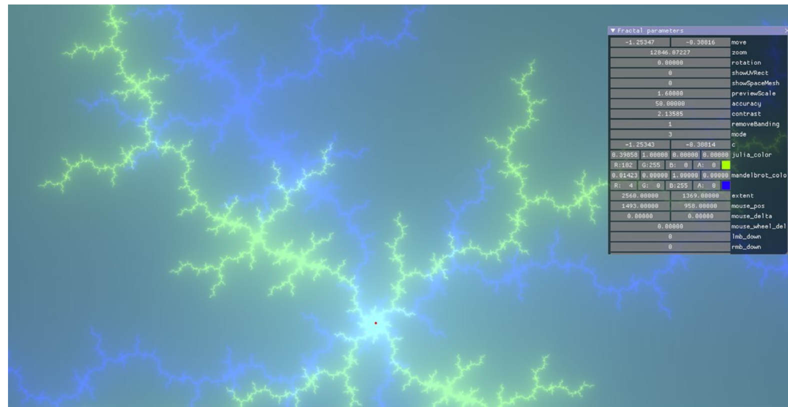

Рисунок 6.9 – Обидва фрактали (Мандельброта – синій, Жуліа – жовтий) біля точки c= (-1.25347,-0.38816) при збільшенні в 12486 раз

Для точок, що знаходяться всередині множини Мандельброта (рис. 6.10) множина Жуліа є зв’язаною і навпаки для точок що за межами множини Мандельброта (рис. 6.11) множина складається з окремих точок.

Найбільш цікаві та складні візерунки (рис. 6.12, 6.13, 6.14, 6.15) виникають для множин з константою, що лежить близько межі множини Мандельброта, де відбувається перехід від зв’язаної структури до незв’язаної.

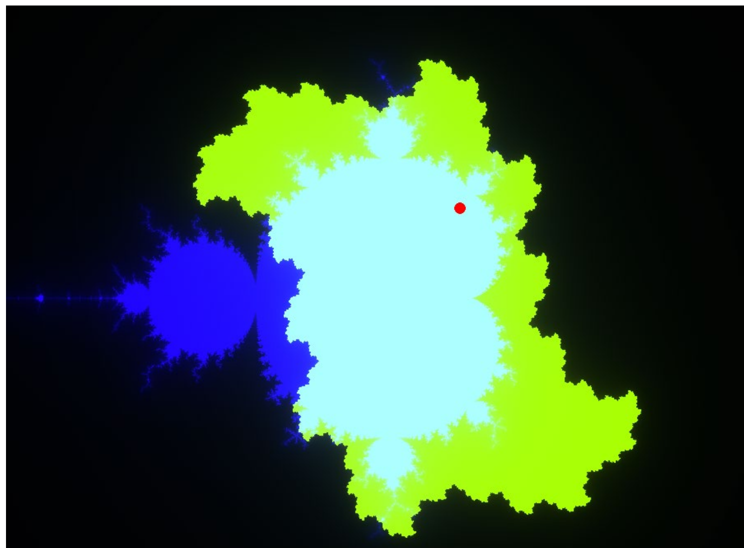

Рисунок 6.10 – Множина Жуліа для константи c всередині множини Мандельброта

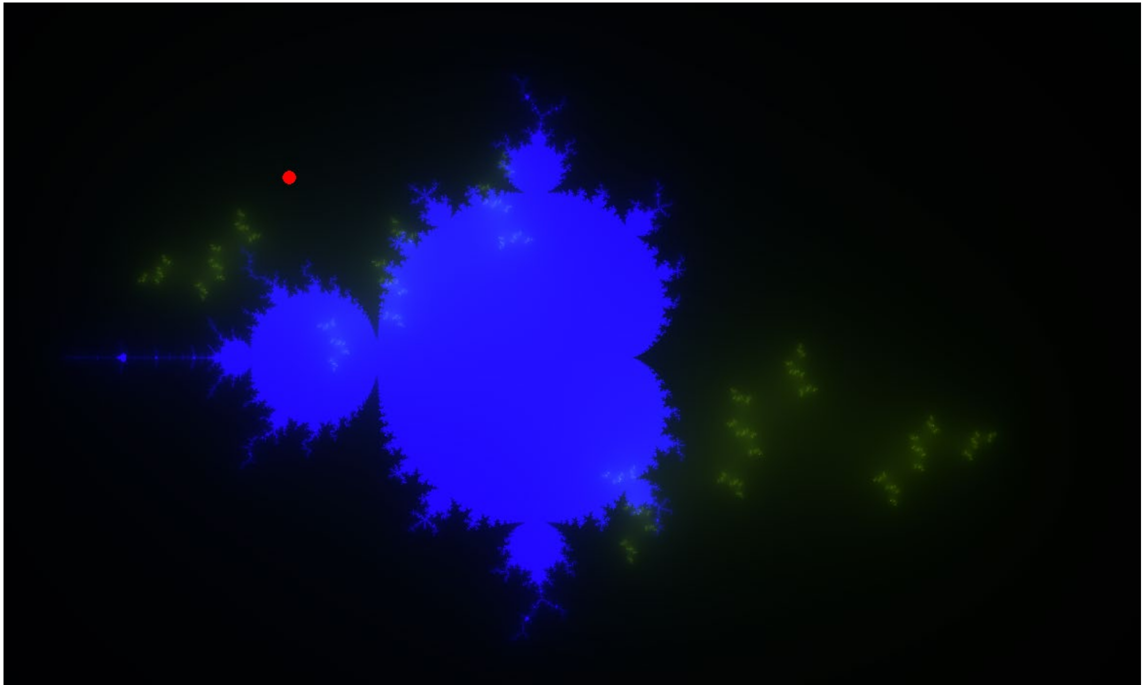

Рисунок 6.11 – Множина Жуліа для константи c за межами множини Мандельброта

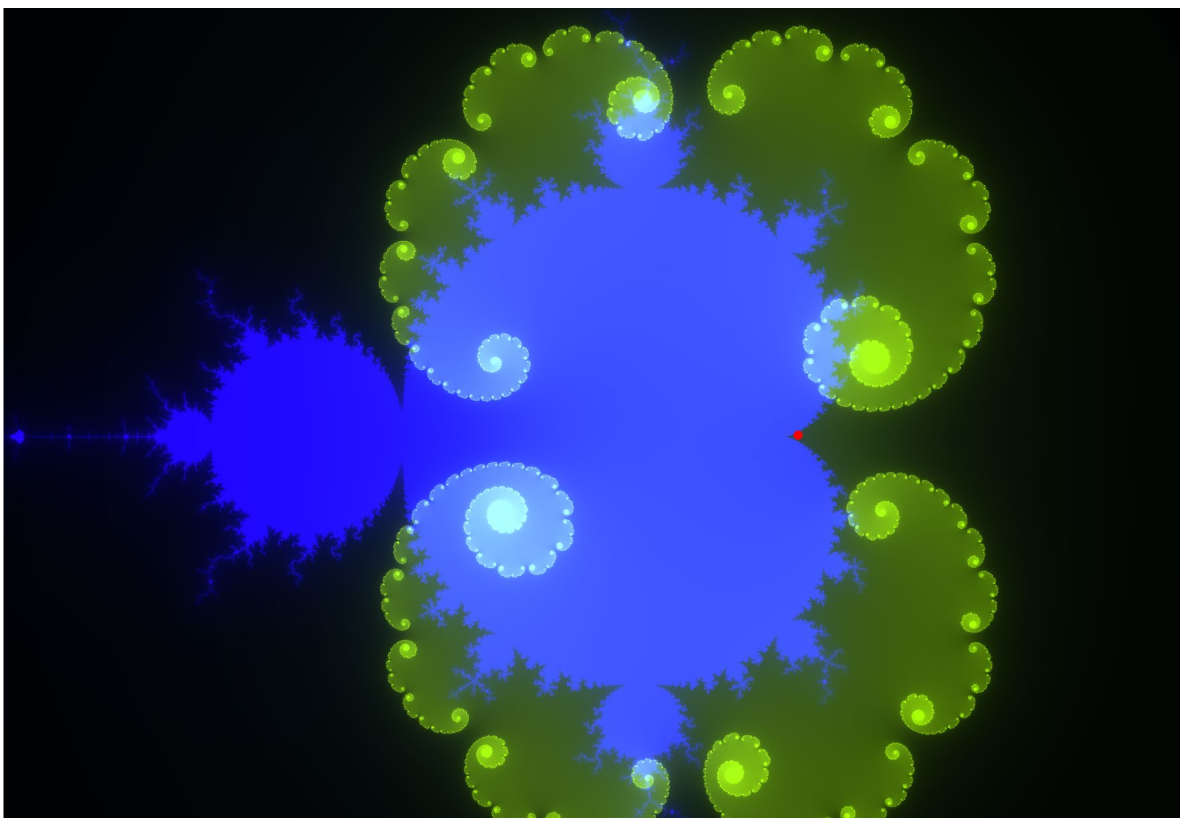

Рисунок 6.12 – Множина Жуліа для точки c на межі множини Мандельброта

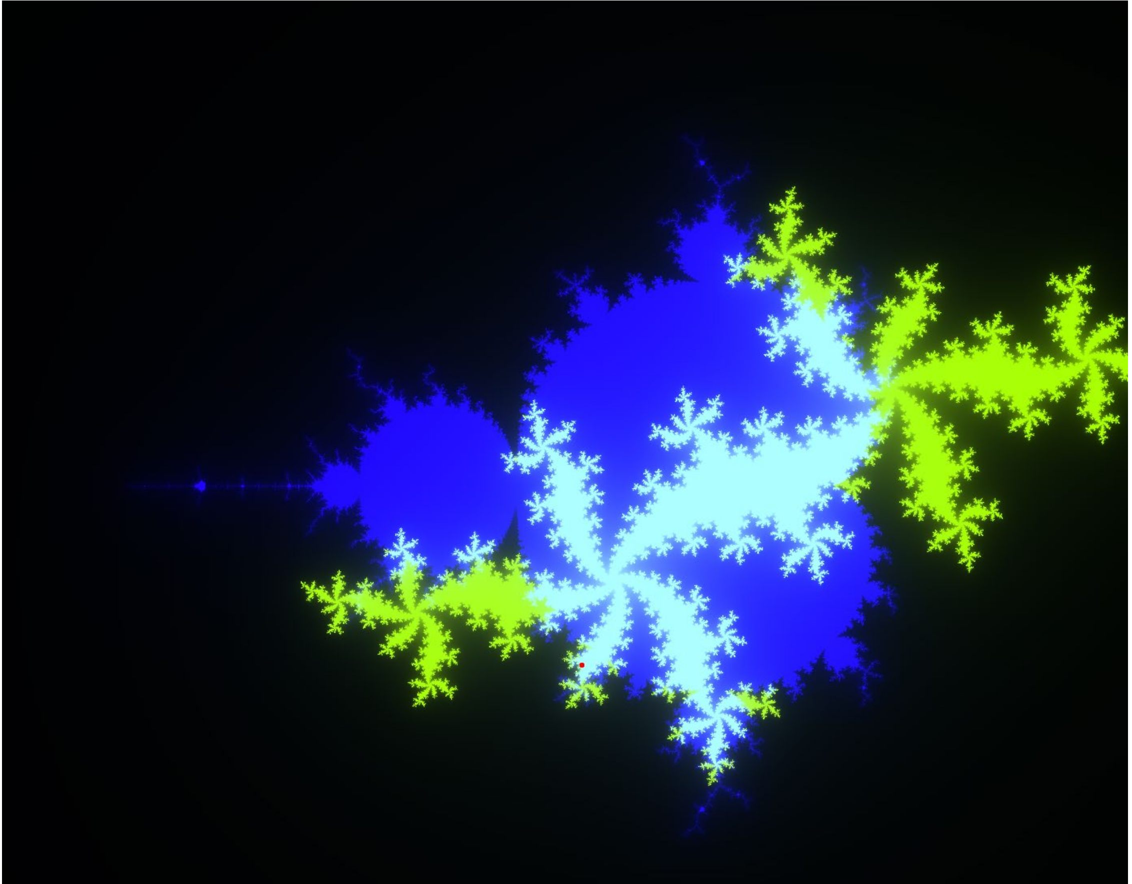

Рисунок 6.13 – Множина Жуліа для точки c на межі множини Мандельброта

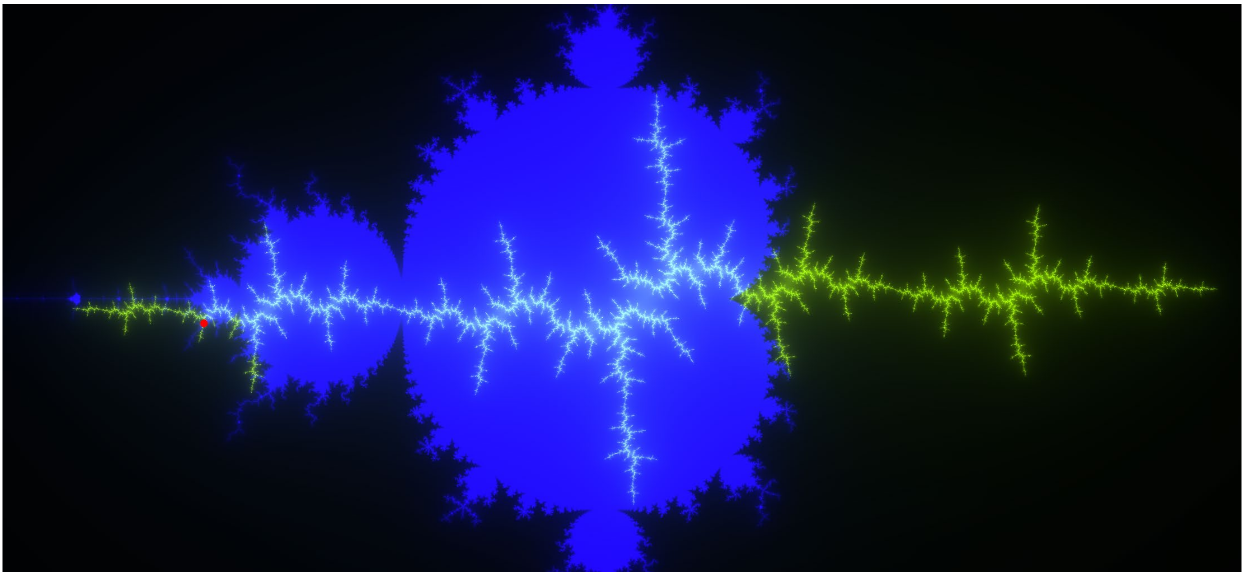

Рисунок 6.14 – Множина Жуліа для точки c на межі множини Мандельброта

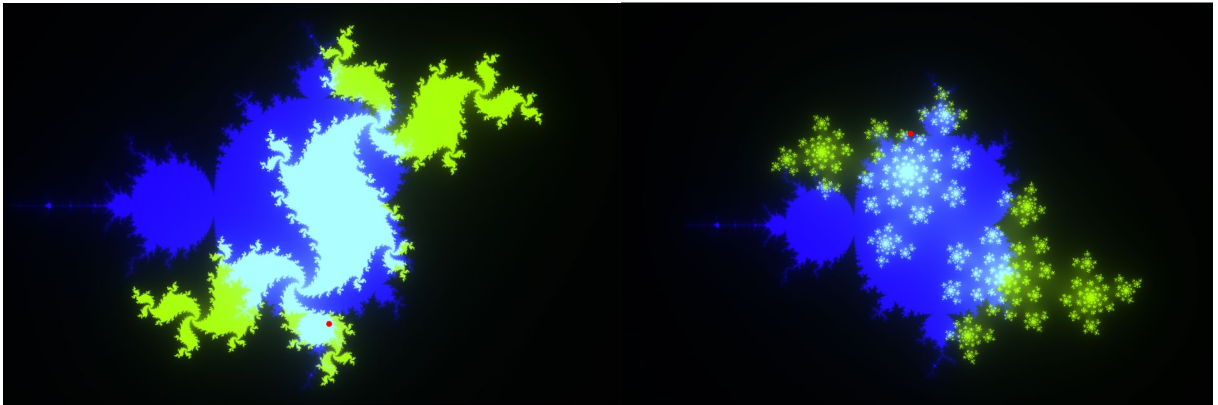

Рисунок 6.15 – Множина Жуліа для точки с на межі множини Мандельброта

Повний код створеного фракталу з керуванням параметрами та перетвореннями простору наведено в додатку Б.

6.1.3 Фрактал Ньютона

Фрактали Ньютона (або Басейни Ньютона) виникають при обчисленні коренів нелінійного комплексного рівняння алгоритмом Ньютона. Для таких фракталів зазвичай точка висвічується кольором, в залежності від кореня до якого призвело вирішення.

Фрактал Нова є узагальненням фрактала Ньютона, де на кожному кроці додається константа с:

$$z_{n+1} = z_n - a\frac{p(z_n)}{p'(z_n)} + c,$$

де p(z) – поліном від z.

Найпоширенішим для створення фракталів є поліном $z^3 - 1$ (рис. 6.16). Код реалізації якого наведено в додатку Б.

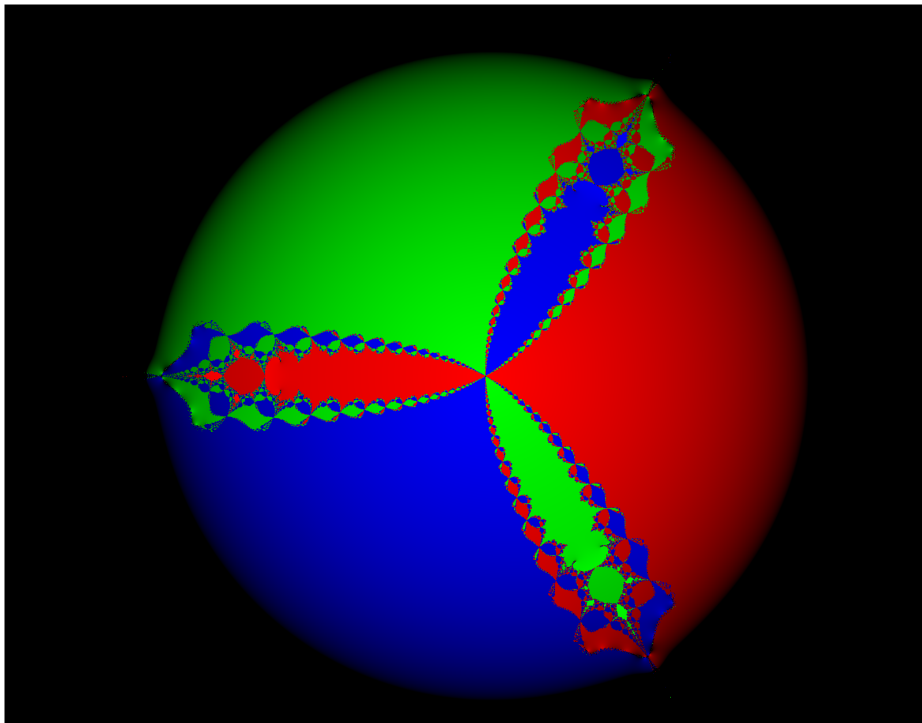

Рисунок 6.16 – Фрактал Ньютона $f(z) = z^3 - 1, a = 1$

Фрактал також має аналог до множини Жуліа (рис. 6.17).

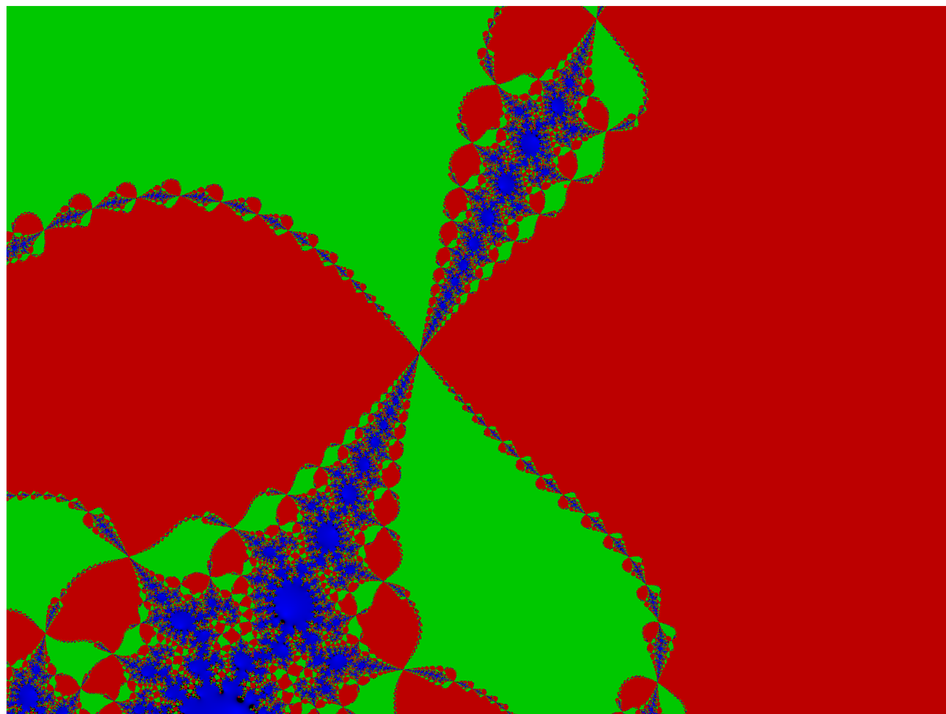

Рисунок 6.17 – Аналог множини Жуліа для фракталу Ньютона

6.1.4 Палаючий корабель

Алгоритм обчислення фракталу Палаючий корабель має лише незначну модифікацію, яка відрізняє його від алгоритму обчислення Мандельброта. Основна різниця в тому, що на кожній ітерації значення дійсної та уявної частини беруться за модулем:

$$z_0 = x$$
$$z_{n+1} = (|Re(z_n)| + i|Im(z_n)|)^2 + x,$$
$$\forall\, n \text{ виконується умова: } |z_n| < 2$$

Для того щоб «корабель» був розташований «щоглою» вверх також можна відразити зображення, найпростіше це зробити змінивши знак перед доданком x:

$$z_{n+1} = (|Re(z_n)| + i|Im(z_n)|)^2 - x$$

Отже, замінивши фрагмент коду для Мандельброта отримано фрактал Палаючий Корабель (лістинг 6.3).

```
float fractal(float2 x, float2 c)
{
    int i=0;
     float2 z=x;
    while(((z.x*z.x+z.y*z.y)<20)&&i<=p.accuracy)
    {
        z=abs(z);
        z= (float2(z.x * z.x - z.y * z.y, 2 * z.x * z.y)-c);
        ++i;
    }
    if(i>=p.accuracy)
     return 1;

    float val=i;
    if(p.removeBanding)
    val=clamp(val-log(log(length(z)))/log(2)+1,0,
p.accuracy);
     return pow(val/p.accuracy,p.contrast);
}
```

Лістинг 6.3 – Модифікація алгоритму множини Мандельброта для отримання множини «Палаючий корабель»

Сам «Корабель» на зображенні (рис. 6.18) знаходиться справа вздовж осі координат x (рис. 6.19) .

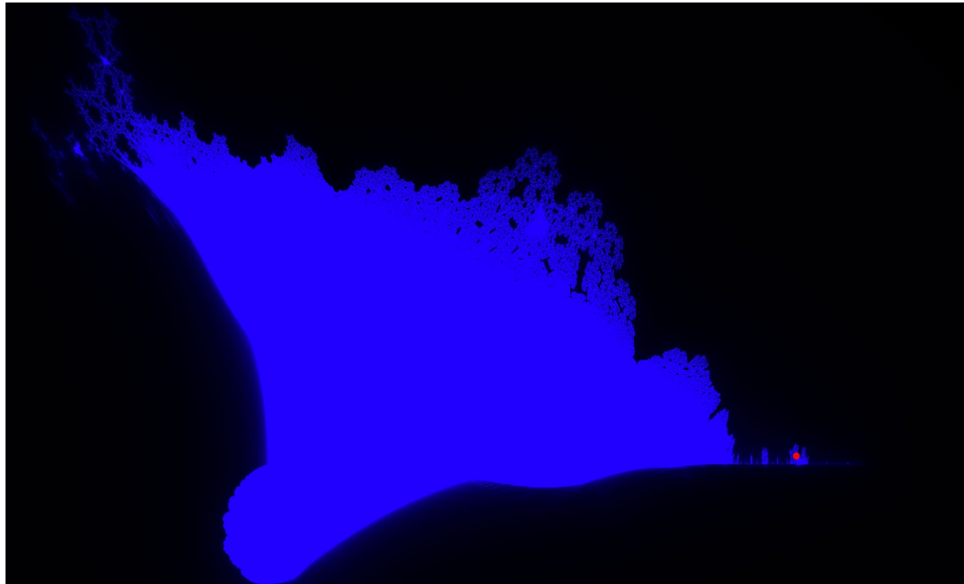

Рисунок 6.18 – Повне зображення фракталу «Палаючий корабель», сам «корабель» виділено червоною точкою

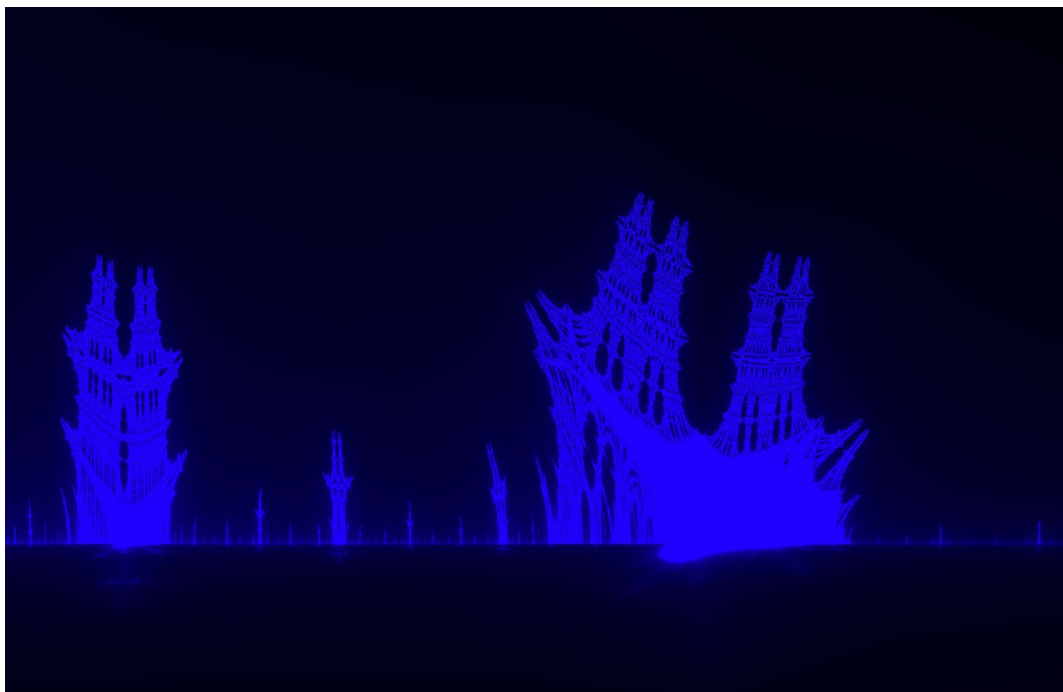

Рисунок 6.19 – «Палаючий корабель» фрагмент

За аналогією до множини Мандельброта, палаючий корабель також має аналог множини Жуліа (рис. 6.20).

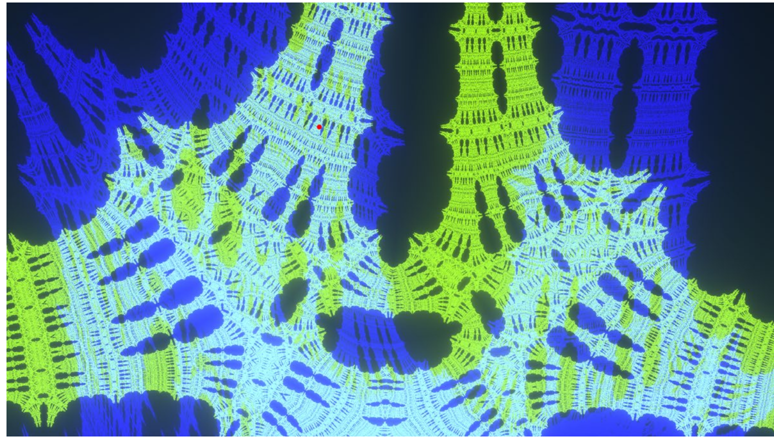

Рисунок 6.20 – Фрагмент Палаючого корабля та відповідної йому множини Жуліа (точка c виділена червоним)

При створенні текстур на основі «Палаючого корабля» має сенс обирати точки або біля «корабля» або з протилежної сторони (рис. 6.21). Причина в тому, що в середині структура дуже мілка та складається з невеликих фрагментів, це спричиняє шум (рис. 6.22), і лише досить значне приближення виключає шум. Іншим варіантом може стати зниження точності обчислення, що зменшує деталізацію (рис. 6.23).

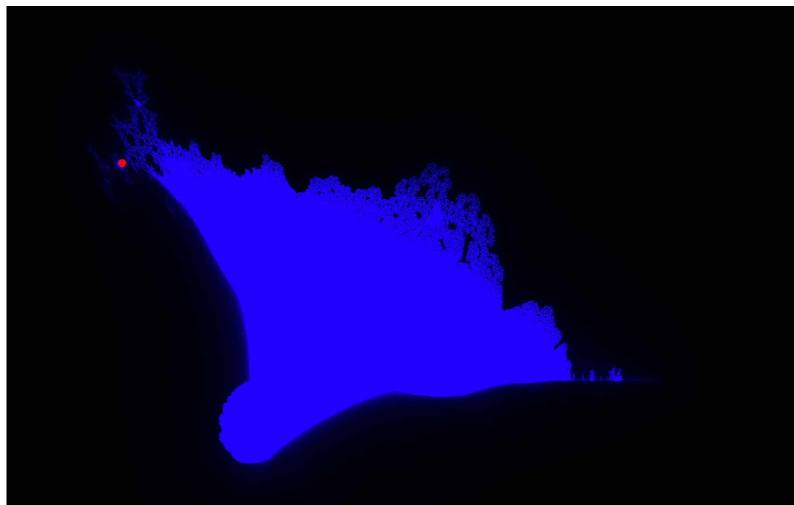

Рисунок 6.21 – Протилежна сторона від «корабля»

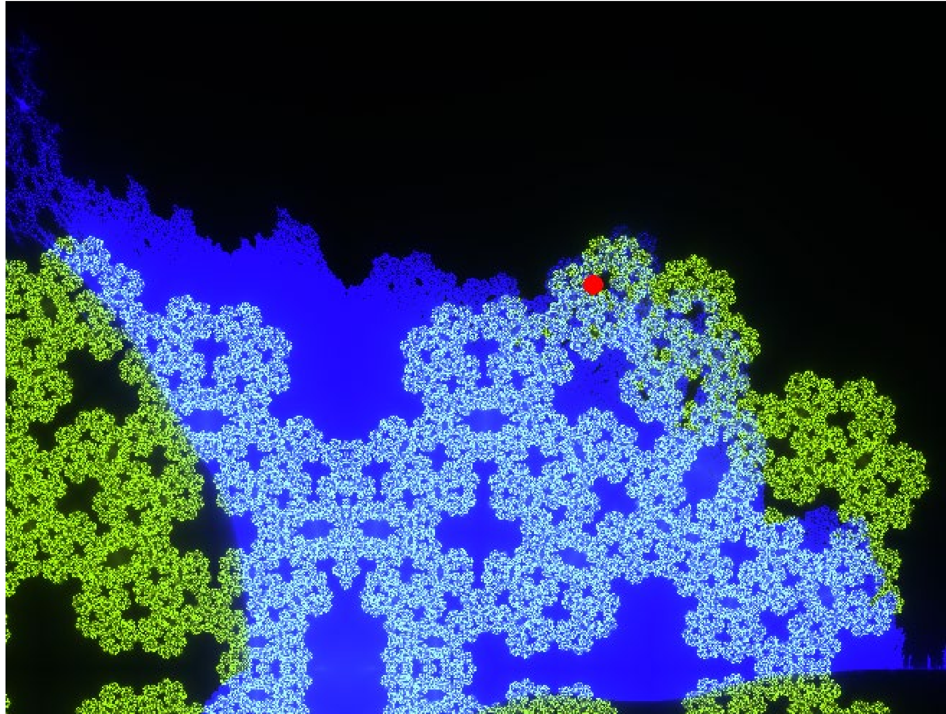

Рисунок 6.22 – «Палаючий корабель» і відповідний фрагмент Жуліа зверху по центру (точність 33 ітерації).

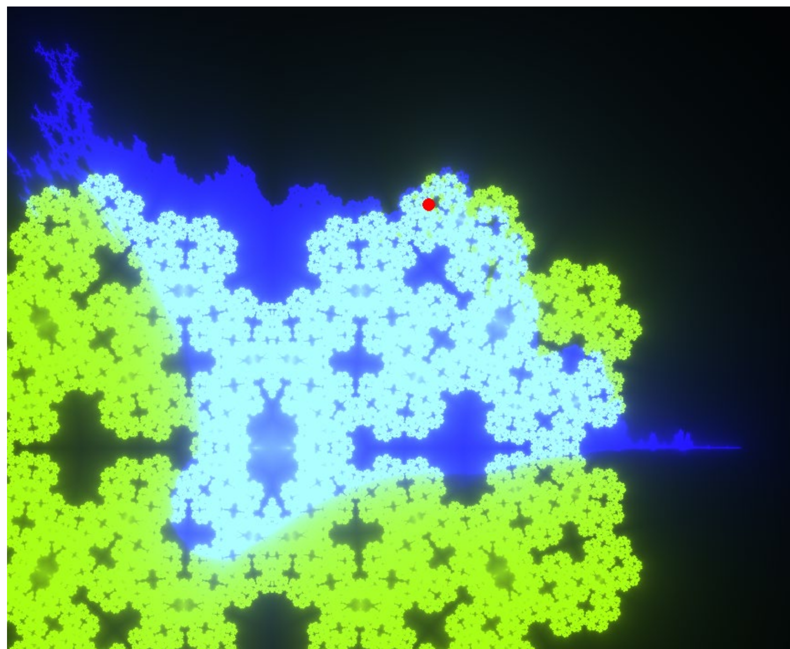

Рисунок 6.23 – «Палаючий корабель» і відповідний фрагмент Жуліа зверху по центру (точність 12 ітерацій).

6.1.5 Фрактали Ляпунова

Алгоритм Фракталів Ляпунова відрізняється від алгоритмів попередніх фракталів, хоча в основі також лежить рекурентне співвідношення. Фрактали Ляпунова будуються відображенням ділянок стабільної та хаотичної поведінки, що вимірюються показником Ляпунова λ, у площині a-b для даної періодичної послідовності a і b [42].

На рисунку 6.24 наведено приклад фракталу Ляпунова. Код реалізації фракталу з анімацією параметрів наведено в додатку Б.

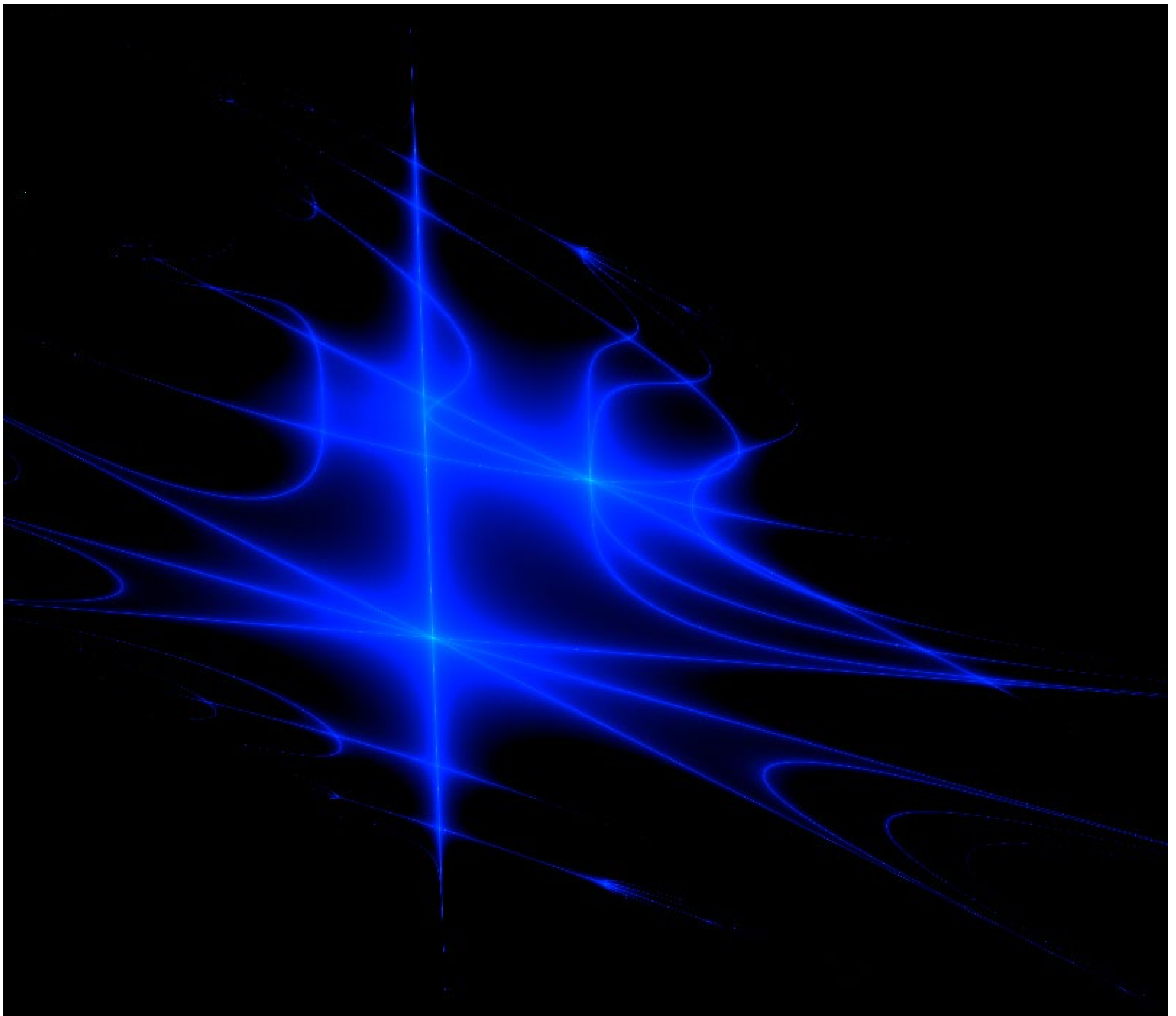

Рисунок 6.24 – Фрактал Ляпунова

6.2 Створення нових різновидів фракталів

Отже, використання різноманітних рекурентних співвідношень відносно початкової координати в багатьох випадках створюють фрактальні множини.

6.2.1 Модифікації Множини Мандельброта та відповідної множини Жуліа.

Багато нових і цікавих текстур можна отримати при зміни основної обчислювальної ітерації за алгоритмом Множини Мандельброта. До речі, одним з таких прикладів є фрактал «Палаючий корабель».

Для структурної модифікації множини Мандельбота необхідно змінити рекурентну формулу:

```
z= float2(z.x * z.x - z.y * z.y, 2 * z.x * z.y)+x;
```

Найпростішою з модифікацій буде зміна коефіцієнта 2 уявної частини. Хоча тоді вже не можна говорити про операції над комплексним числом так як це не бути відповідати z^2.

Для цього зручно ввести параметр k:

```
z= float2(z.x * z.x - z.y * z.y, p.k * z.x * z.y)+x;
```

На малюнках (рис. 6.25, 6.26) наведені можливі варіації для різних k. Слід відмітити, що алгоритм по видаленню чітких переходів не буде працювати правильно через те, що операція не відповідає комплексному квадрату і модуль $|z_n|$ не є приблизним квадратом модуля $|z_{n-1}|$.

Інші модифікації множини Мандельброта наведено у додатку В.

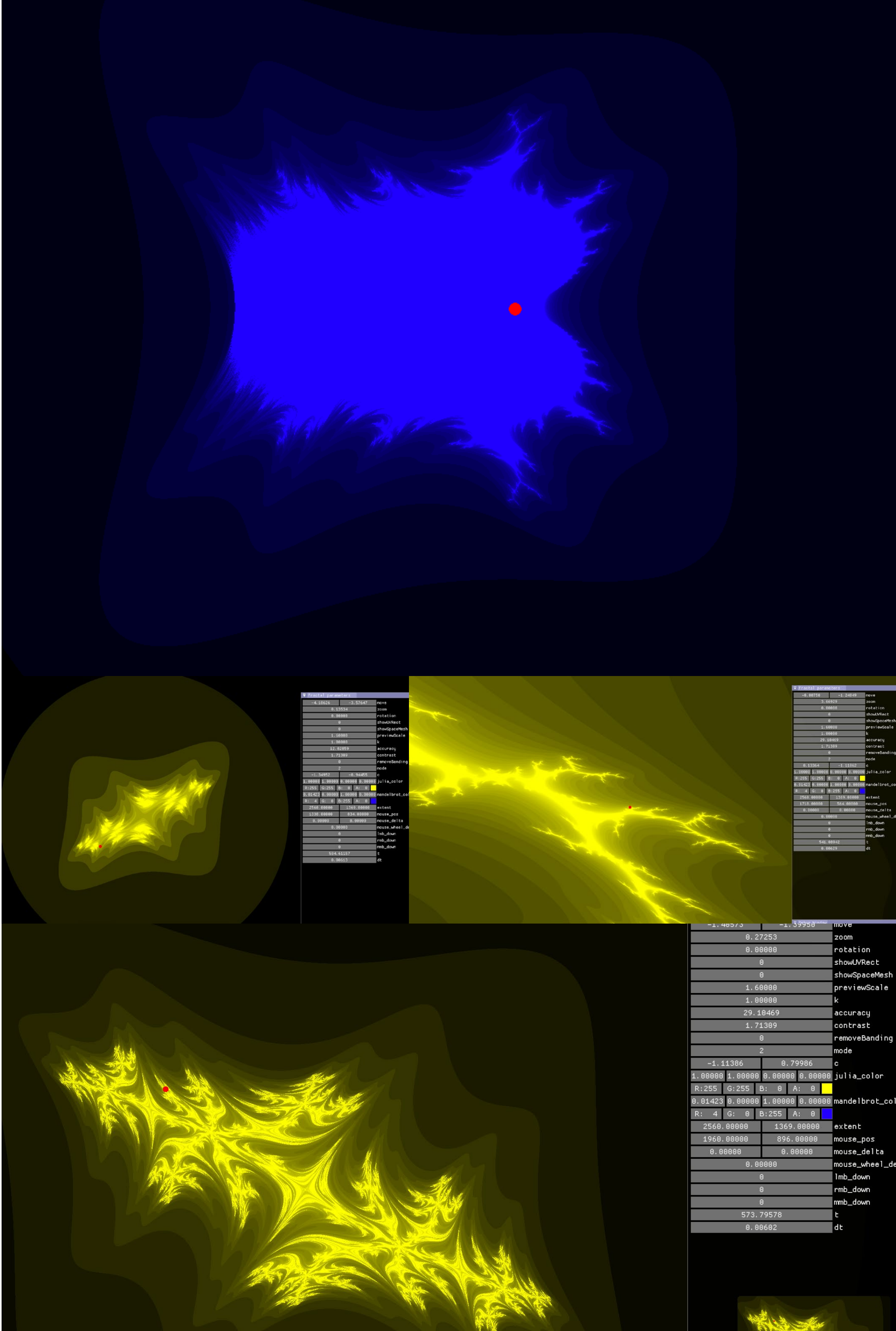

Рисунок 6.25 – Модифікація множини Мандельброта і відповідних множин Жуліа ($k = 1$)

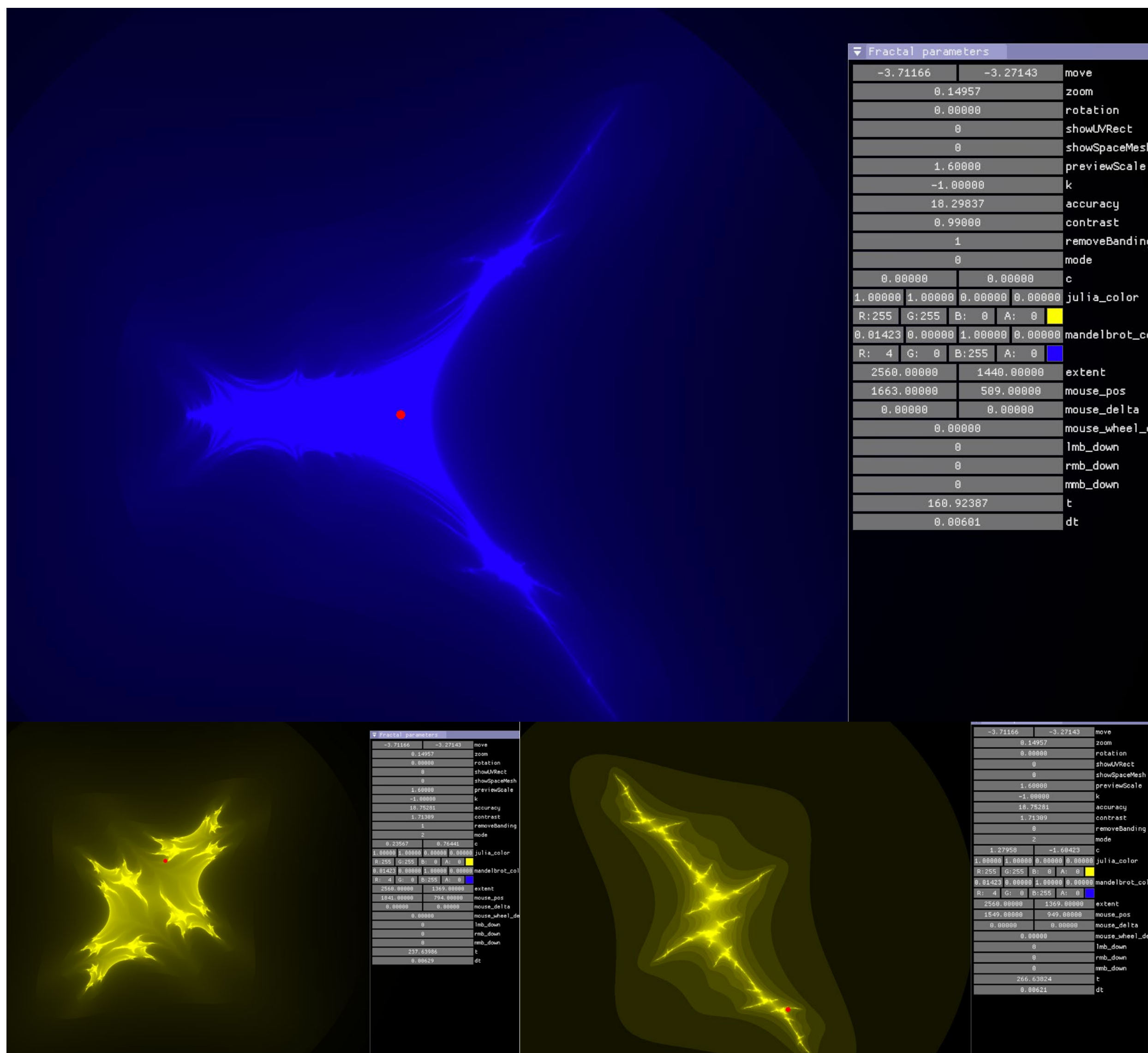

Рисунок 6.26 – Модифікація множини Мандельброта і відповідних множин Жуліа ($k = -1$)

6.3 Фрактальний шум

Окрім фрактальних множин, реалізація яких була наведена, при створенні текстур також може бути корисним застосування фрактального шуму.

Ідея фрактального шуму полягає в рекурентному додаванні шуму зменшуючи амплітуду та збільшуючи частоту.

6.3.1 Шум Перліна.

За основу взято шум Перліна, який можна обчислити за наступним алгоритмом (лістинг 6.4) [43].

```
	float random(float2 pos)
	{
		return frac(sin(dot(pos,float2(12.9898,78.233)))*
43758.5453123);
	}
	float noise(float2 pos)
	{
		float2 p0=floor(pos);
		float2 f=pos%1.0;
		float2 normal=6*f-6*f*f;
		f= f*f*(3.0-2.0*f);
		float r=random(int2(p0));
		float r_x=random(int2(p0)+int2(1,0));
		float r_y=random(int2(p0)+int2(0,1));
		float r_xy=random(int2(p0)+int2(1,1));
		float res=lerp(lerp(r,r_x,f.x),lerp(r_y,r_xy,f.x),f.y);
		return res;
	}
```

Лістинг 6.4 – Обчислення шуму Перліна на площині

Суть шуму полягає в гладкому переході між випадковими значеннями, що утворюють сітку (рис. 6.27). Це один з перших алгоритмів генерації шуму, не самий оптимальний за швидкістю та якістю, але для створення демонстраційного фрактального шуму його достатньо.

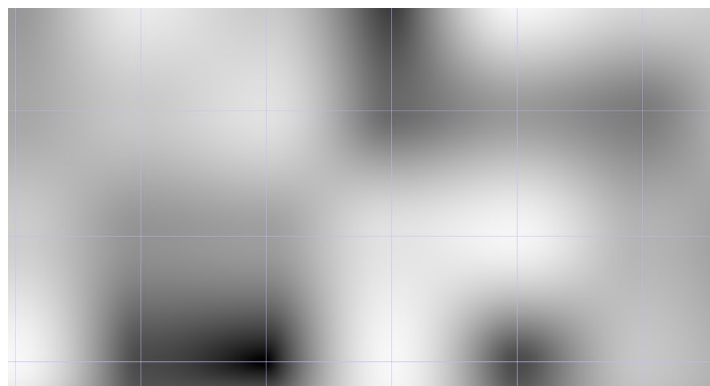

Рисунок 6.27 – Шум Перліна

6.3.2 Фрактальний броунівський рух

Фрактальний шум (або фрактальний броунівський рух, fbm) генерується ітеративним додаванням шуму зі зміною частоти та амплітуди (лістинг 6.5) [44]. У класичному підході амплітуда зменшується вдвічі, а частота збільшується вдвічі, хоча можуть бути застосовані і інші коефіцієнти. Результат для різної кількості ітерацій наведено на рисунку 6.28.

```
float fbm(float2 st)
{
	float val=0;float amplitude=0.5; float freq=1;
	for(int i=0;i<p.octaves;++i)
	{
		val+=noise(st*freq)*amplitude;
		freq*=2;
		amplitude*=0.5;
	}
	return val;
}
```

Лістинг 6.5 – Обчислення фрактального шуму

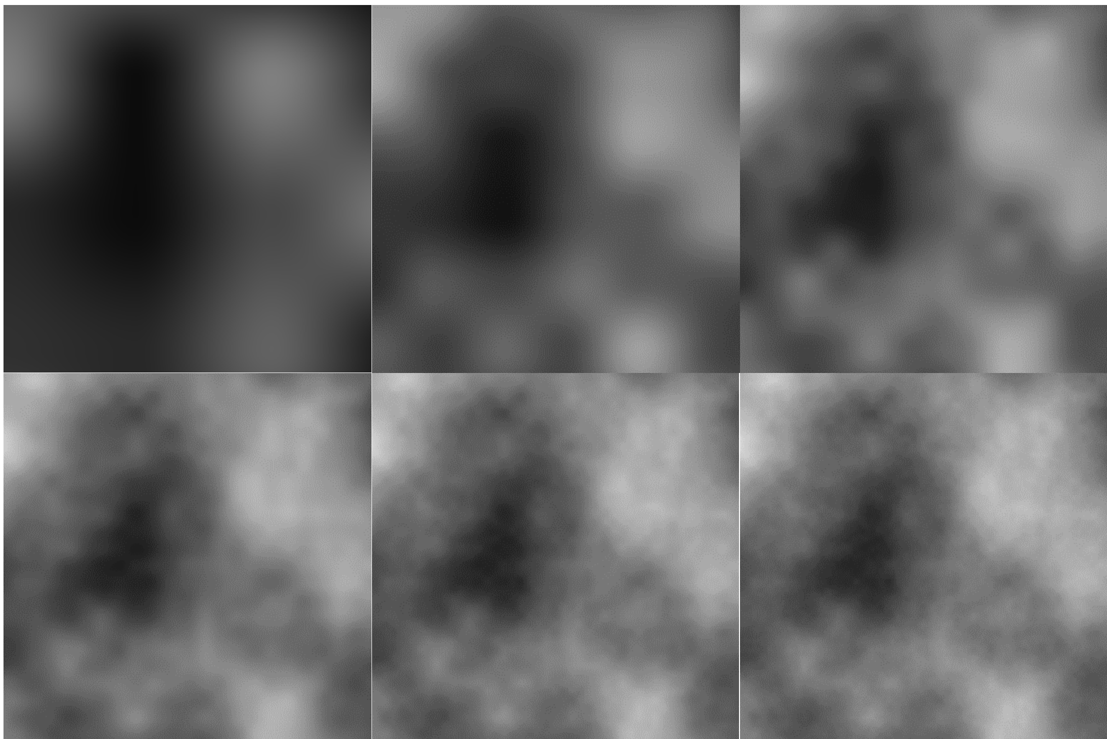

Рисунок 6.28 – Фрактальний броунівський рух з кількістю ітерацій 1 (звичайний шум), 2, 3, 4, 5 та 20

6.3.3 Використання фрактального шуму для викривлення простору

Фрактальний шум цікаво застосовувати для викривлення простору. Ідея полягає в зміщенні координатної точки, що відповідає фрагменту зображення, на значення шуму. Оскільки шум є неперервною функцією, то і таке зміщення є неперервною функцією у просторі. Таким чином, для кожної точки отримуєтеся вектор зміщення, що плавно зміються від пікселя до пікселя (рис. 6.29, 6.30, 6.31).

Цікавого ефекту можна досягти додавши викривлення простору на основі фрактального шуму саме до зображення фрактальної множини (рис. 6.32, 6.33, 6.34, 6.35, 6.36). При викривленні певні регіони значно розтягуються інші ж стискаються, але, оскільки точність базового фрактального зображення необмежена, такі викривлення не впливають на кінцеву якість.

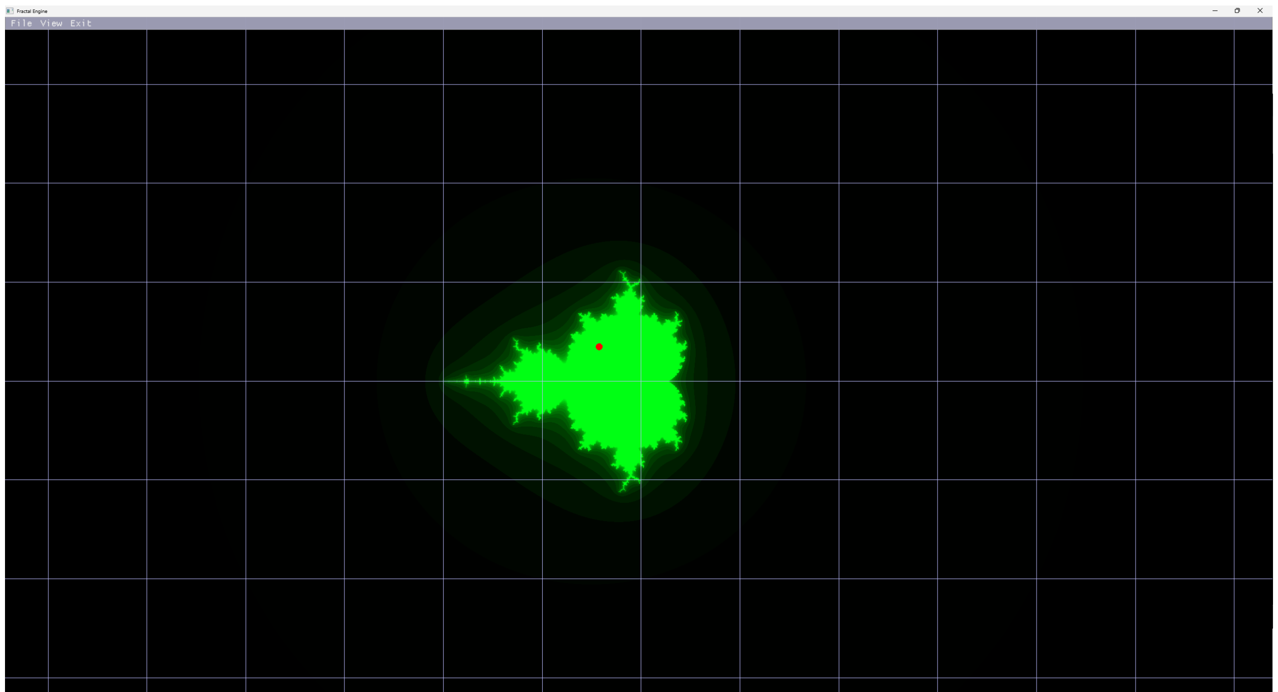

Рисунок 6.29 – Неспотворене зображення

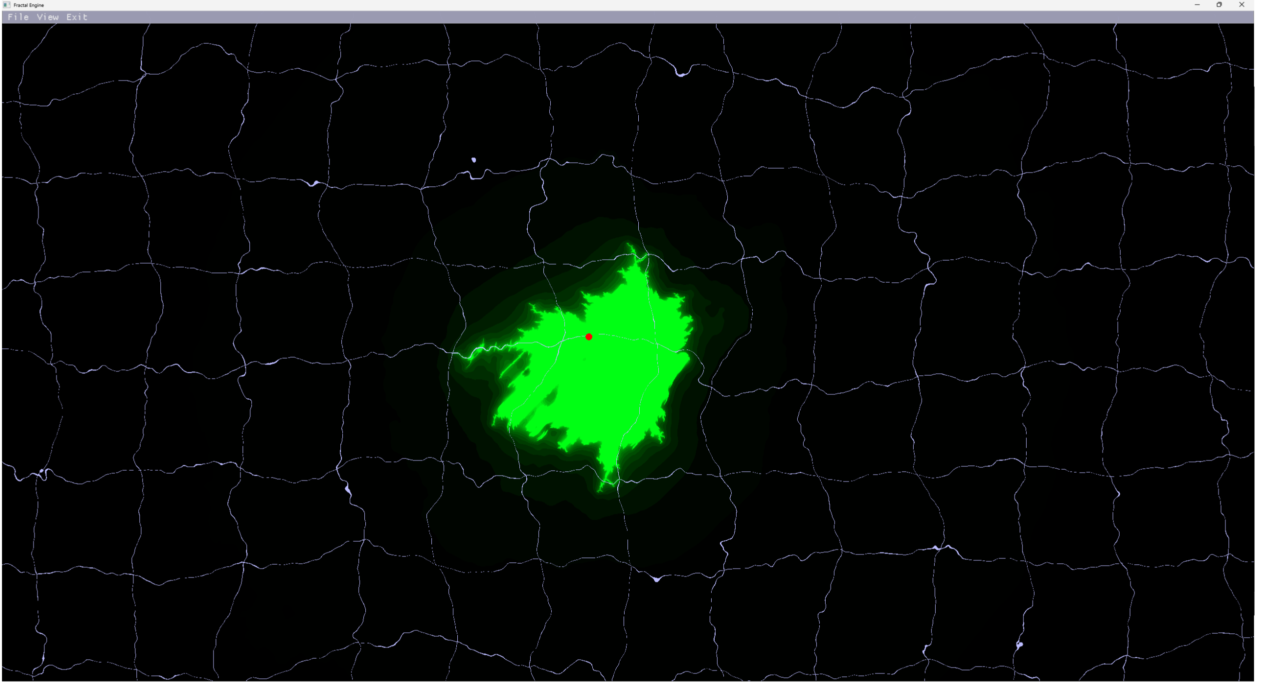

Рисунок 6.30 – Зображення при зміщення координат на основі фрактального шуму з максимальним відхиленням 0.5

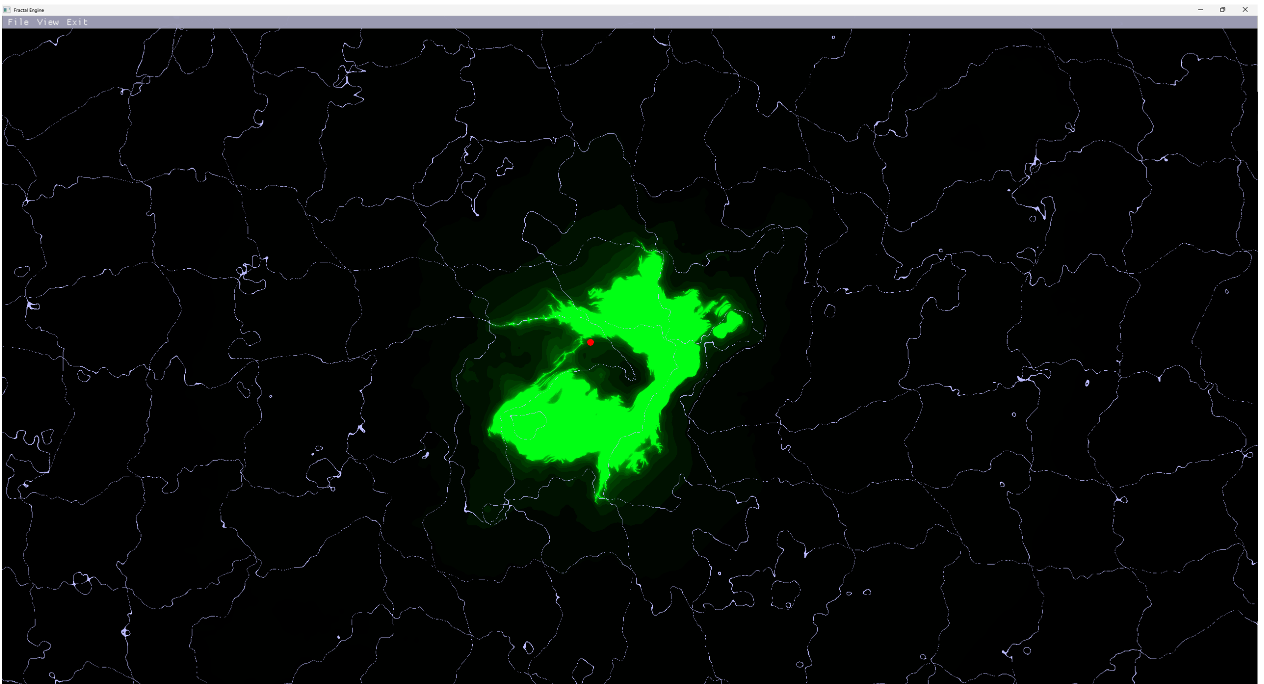

Рисунок 6.31 – Зображення при зміщення координат на основі фрактального шуму з максимальним відхиленням 1

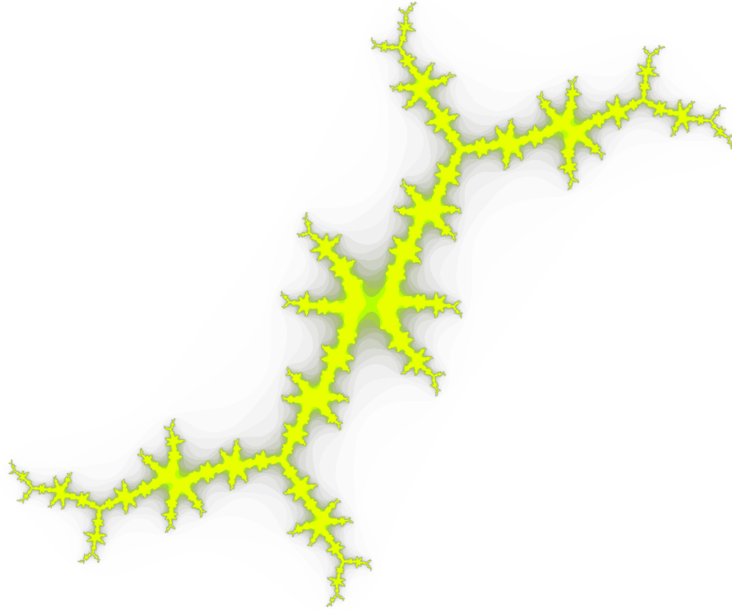

Рисунок 6.32 – Початкове зображення множини Жуліа

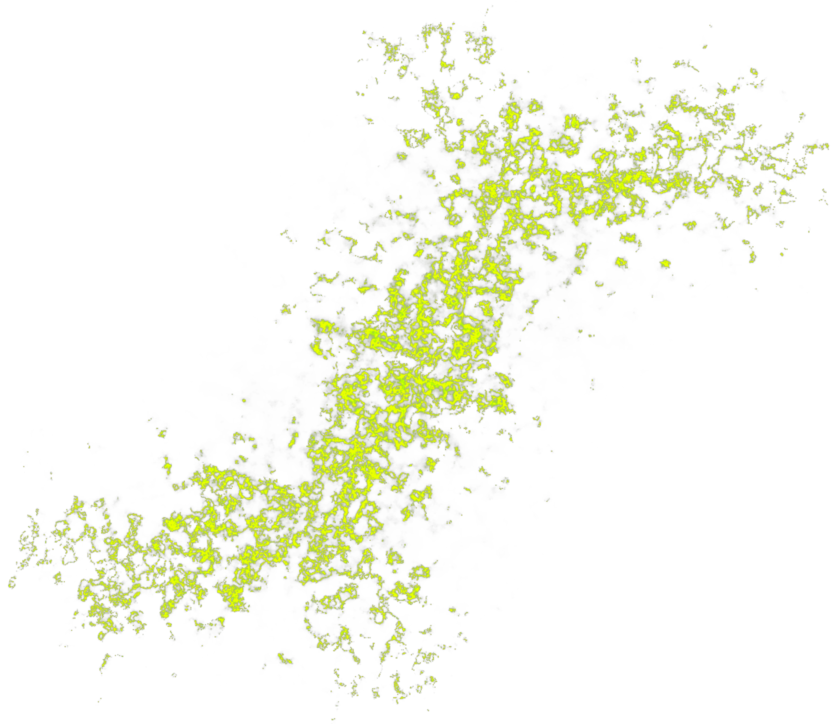

Рисунок 6.33 – Викривлення множини Жуліа за допомогою fbm, максимальне відхилення 0.5

Рисунок 6.34 – Викривлення множини Жуліа за допомогою fbm, максимальне відхилення 2

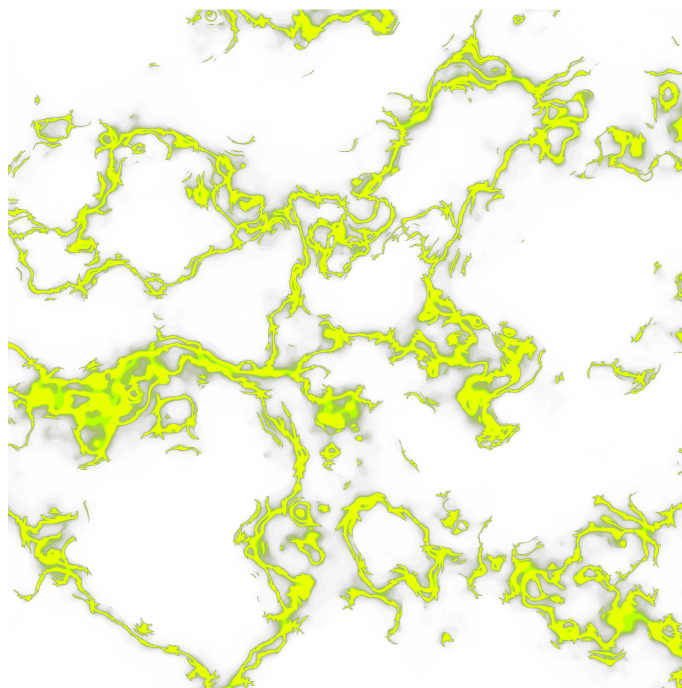

Рисунок 6.35 – Викривлення множини Жуліа за допомогою fbm (збільшене зображення), максимальне відхилення 2

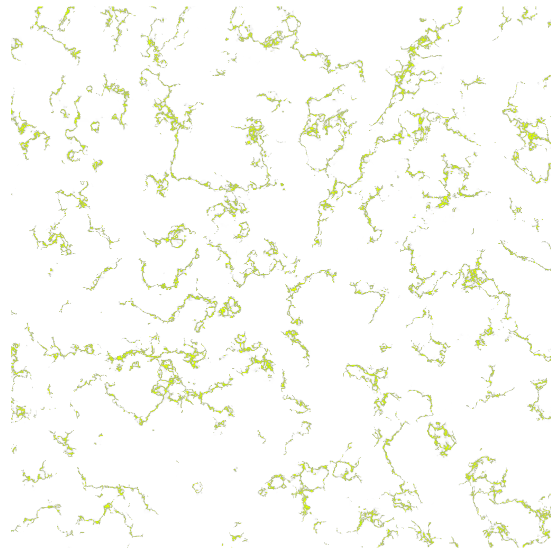

Рисунок 6.36 – Викривлення множини Жуліа за допомогою fbm, максимальне відхилення 5

Приклад того, що можна отримати якщо декілька разів послідовно викривити простір за допомогою фрактального шуму та відобразити у викривленому просторі зображення фрактального шуму з кольоровим переходом зображено на рисунку 6.37. Код даної текстури у додатку Б.

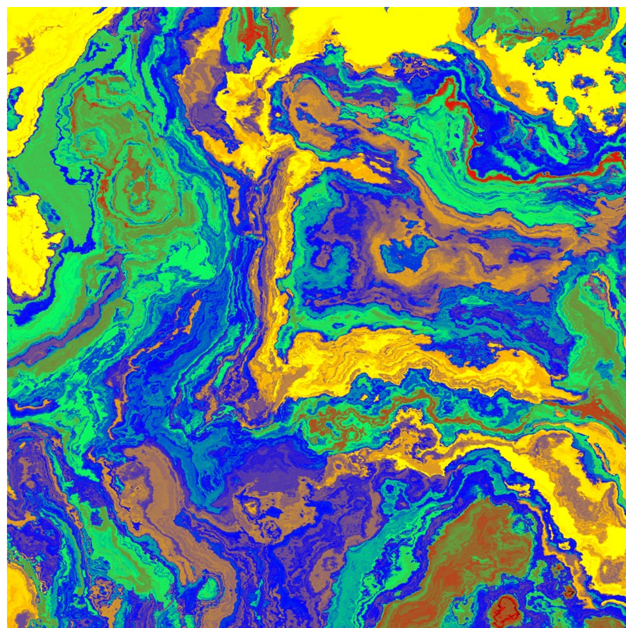

Рисунок 6.37 – Кольоровий фрактальний шум відображений у декілька разів викривленому просторі за допомогою фрактального шуму

6.4 Анімація фрактальних текстур

Анімація створюється шляхом додавання динамічних параметрів до коду текстури. Наприклад, можна додати залежність від часу, і передавати час до коду текстури.

Тут наведено основну ідею створення анімації на прикладі множини Жуліа. В цьому випадку цікаво змінювати значення константи, додаючи до неї невеликі значення у регіоні. Наприклад, це може бути замкнутий рух по еліпсу.

Для реалізації такої поведінки існує два підходи. Зручнішим, з точки зору подальшого використання, буде передати час як динамічний параметр та вже на основі часу визначити зміщення (лістинг 6.6). Все що потрібно для роботи такої текстури це надати їй інформацію про час.

```
float4 output(float2 uv, float t)
{
    float v=float2(cos(t), sin(t))*0.1;
    float2 pos = uvToSpacePoint(uv);

    float4 result=0;
    if(p.mode!=0)
           result+= fractal(pos,p.c+v)*p.julia_color;
    if(p.mode!=2)
            result+= fractal(pos,pos)*p.mandelbrot_color;
    return result;
}
```

Лістинг 6.6 – Анімація множини Жуліа з динамічним параметром t

Більш складна реалізація прийматиме значення зміщення константи на вхід, тоді як поведінка самої константи буде керуватися за межами текстури програмою чи графічним рушієм, що її використовує (лістинг 6.7). Цей підхід надає більш гнучкий контроль над текстурою, а також є більш оптимальним, так як зміщення обчислюється лише один раз, а не для кожного фрагменту текстури окремо. При роботі в створеному редакторі, для зовнішнього

керування можна створити окремий параметр та оновлювати його в функції `UpdateParameters` замість прямого обчислення цього зміщення в `output`. Звичайно при експорті такої текстури анімація буде виконуватися лише при зміні параметра середою в якій вона застосовується, це додає незручності, так як необхідно окремо синхронізувати алгоритм зміни константи для отримання поведінки ідентичної до тої що отримана в редакторі.

```
float4 output(float2 uv, float2 v)
{
    float2 pos = uvToSpacePoint(uv);
    float4 result=0;
    if(p.mode!=0)
           result+= fractal(pos,p.c+v)*p.julia_color;
    if(p.mode!=2)
            result+= fractal(pos,pos)*p.mandelbrot_color;
    return result;
}
```

Лістинг 6.7 – Анімація множини Жуліа з динамічним параметром v

На рисунку 6.38 зображено декілька кадрів зі створеної анімації.

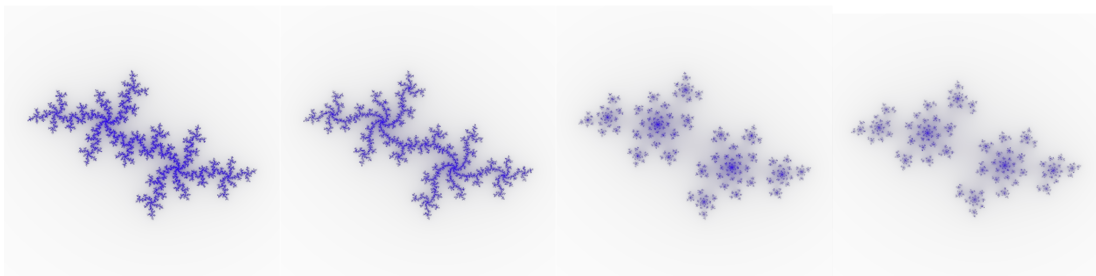

Рисунок 6.38 – Кадри анімації множини Жуліа

Іншу анімацію можна отримати якщо переміщувати зображення по простору, що було викривлено за допомого фрактального шуму. Якщо викривити координати, а потім зміщати зображення що відображається за цими координатами, можна створити анімацію текучості (рис. 6.39). Головний код анімації наведено в лістингу 6.8.

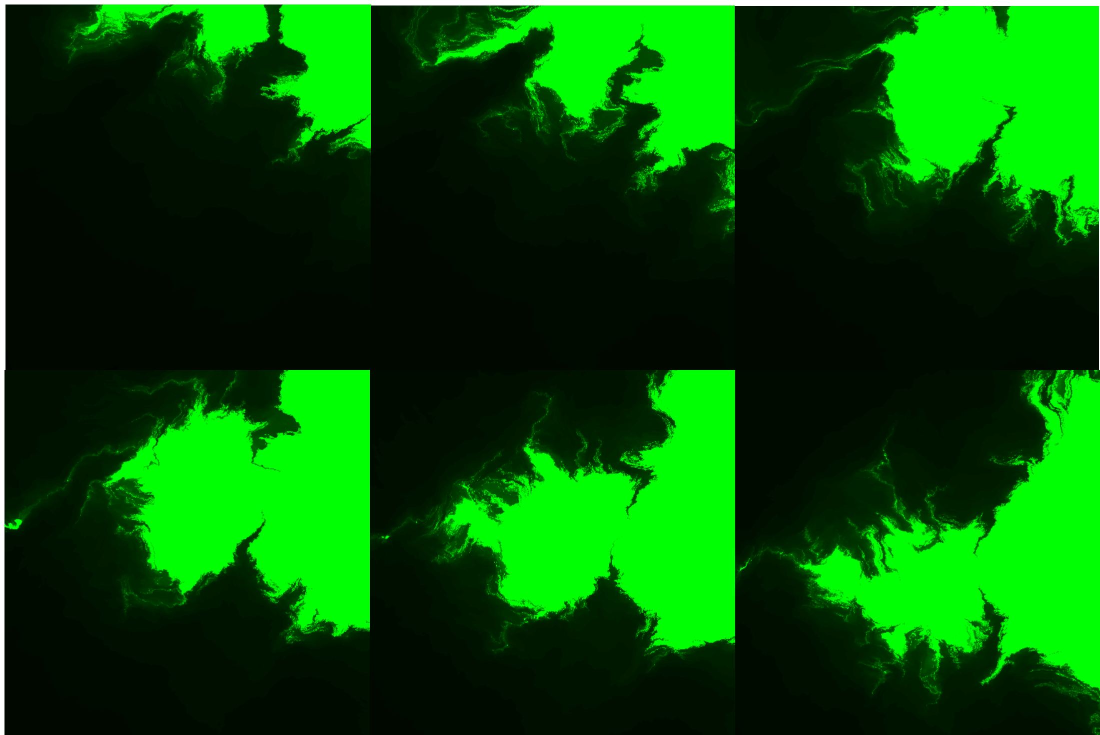

Рисунок 6.39 – Приклад анімації текучості за допомогою викривлення простору фрактальним шумом, за тестове зображення взято множину Мандельброта, що рухається зверху вниз. Інтервали часу між кадрами дорівнюють 0.1 секунда.

```
float4 output(float2 uv,float t)
{
	float2 v=float2(0,t*5);
	float2 pos = uvToSpacePoint(uv);
	for(int i =0; i<3;++i)
	{
		pos+=fbm2(pos+float2(-100,-345));
	}
	return mandelbrot((pos+v)/10%1);
}
```

Лістинг 6.8 – Анімація текучості

Ще один варіант схожої анімації наведено на рисунку 6.40. Тут вже замість переміщення зображення трохи видозмінюються зміщення точки при розрахунку викривлення на кожній ітерації (лістинг 6.9). Додаткове зміщення рухається по колу радіусом 1. За основне зображення взято фрактальний шум в кольорі. Як результат, загальне зображення вже не рухається а якби перетікає локально.

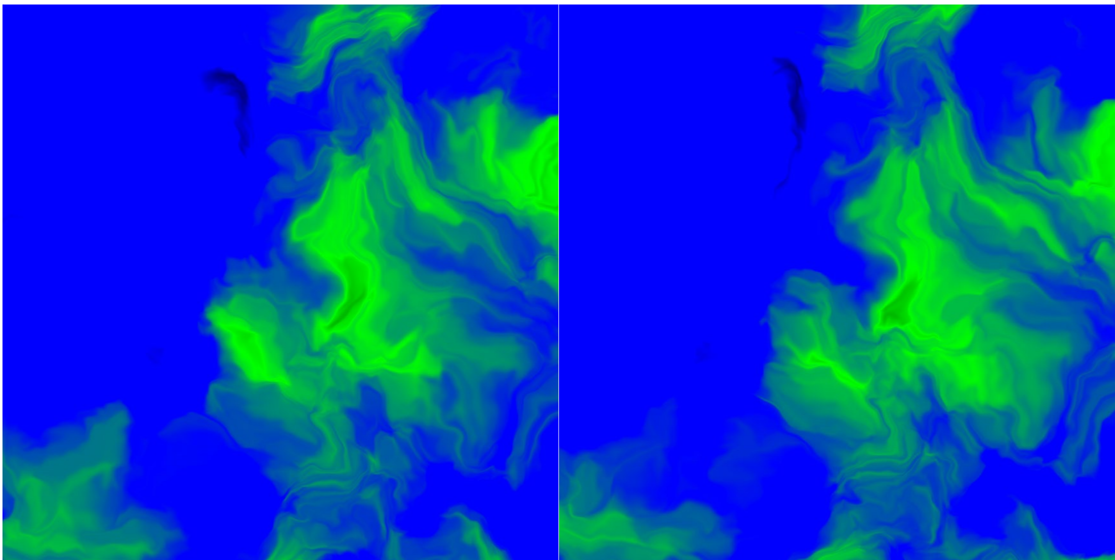

Рисунок 6.40 – Приклад анімації текучості за допомогою викривлення простору фрактальним шумом зі зміною зміщення координат при розрахунку викривлення

```
float4 output(float2 uv,float t)
{
	float2 v=float2(cos(t/10),sin(t/10));
	float2 pos = uvToSpacePoint(uv);
	for(int i =0; i<4;++i)
	{
		pos+=fbm2(pos+float2(-100,-345)+v);
		pos*=1.1;
	}
	return colored(fbm(pos));
}
```

Лістинг 6.9 – Анімація текучості (локальна)

6.5 Точність обчислення алгоритмів

В таблиці 6.1 наведена точність обчислень при розрахунку з використанням типу даних `float`.

Для обчислення максимального коефіцієнту масштабування визначається в скільки разів необхідно приблизити щоб в одиничному квадраті вмістився один прямокутник, всі точки якого визначаються як одне й

те саме значення. Для цього необхідно розглядати найвіддаленіші точки від початку координат простору фрактала, так як для них точність числа з плаваючою точкою (`float`) буде менша. Наприклад, на рисунку 6.41 показано максимальний масштаб для фрактала Мандельброта. Розмір еквівалентної текстури розраховується як добуток розміру множини та коефіцієнту максимального масштабування.

Таблиця 6.1 – Точність обчислення для різних фрактальних текстур

Фрактал	Приблизний розмір множини у просторі	Максимальний коефіцієнт масштабування	Розмір еквівалентної растрової текстури
Множина Мандельброта	2.5x2.5	7,775,432	19,438,580x19,438,580
Множина Жулія	2.5x2.5	7,775,376	19,438,440x19,438,440
Фрактал Ньютона	1x1	16,283,763	16,283,763x16,283,763
Фрактал Ляпунова	9x9	3,815,428	34,338,852x34,338,852
Фрактальний шум	100,000,000x 100,000,000	1081	$108*10^9$x$108*10^9$

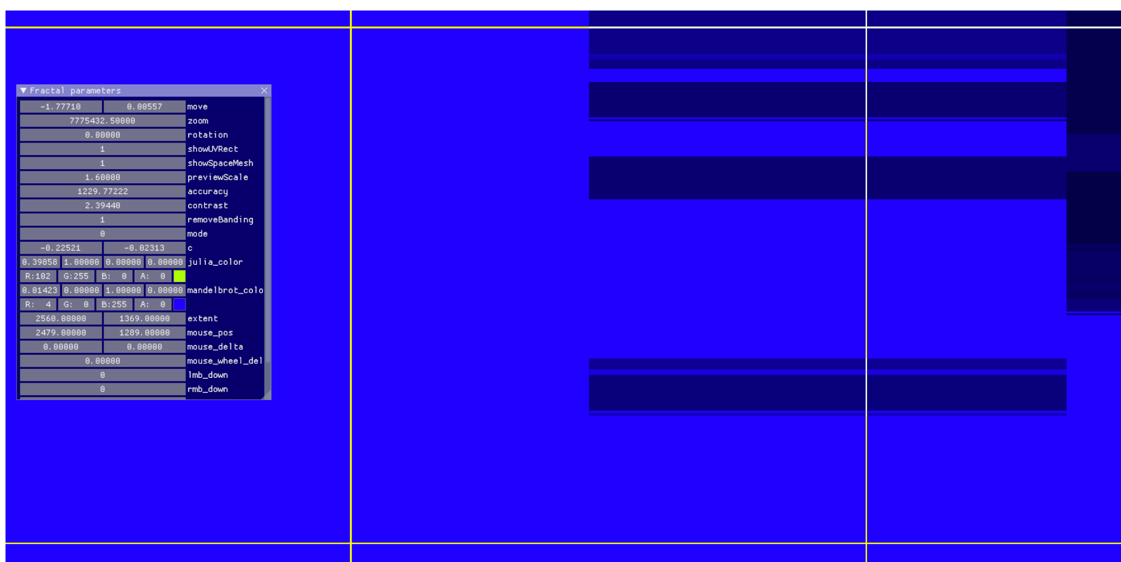

Рисунок 6.41 – Визначення максимального масштабу Мандельброта

6.6 Робота з тривимірними текстурами

Редактор також може бути використаний для розробки тривимірних текстур та поверхонь. Такі текстури можуть бути корисними для алгоритмів трасування променів, динамічній генерації ландшафту, або описані певних властивостей простору (вектор сили, вектор вітру, туманність). Для відображення тривимірної текстури нема необхідності змінювати вихідний код самого редактора, а всі необхідні налаштування можна описати у керуючому коді (`main` – відображення текстури та `UpdateParameters` – керування параметрами). Для продуктивної розробки таких текстур необхідно створити шаблони текстур, що будуть спрощувати обернення камери та відображення. Створено два демонстраційних алгоритми відображення тривимірної текстури за допомогою відстеження променів через матрицю вокселів (рис. 6.42) та відстеженням променів із фіксованим кроком (рис. 6.43). В обох випадках тестовою текстурою обрано 4D аналог множини Жуліа, що базується на кватерніонах (чотиривимірне гіперкомплексне число).

Обидва алгоритми можуть відображати будь-яку тривимірну текстуру типу кольору з прозорістю (`float4`) з певною мірою точності. Повний код прикладу наведено в додатку Б.

Це лише два з можливих підходів зобразити тривимірну текстуру, вони є відносно універсальними, через що повільні. Якщо враховувати особливості текстурної функції можливе пришвидшення обчислень. Наприклад для 4D множини Мандельброта (та Жуліа) існує алгоритм обчислення поверхні з врахуванням найкоротшої відстані до множини [45]. Якщо відома відстань до поверхні в точці то можна сміливо рухатись на цю відстань на кожній ітерації, замість переходу до сусіднього вокселя чи зміщення на фіксоване значення.

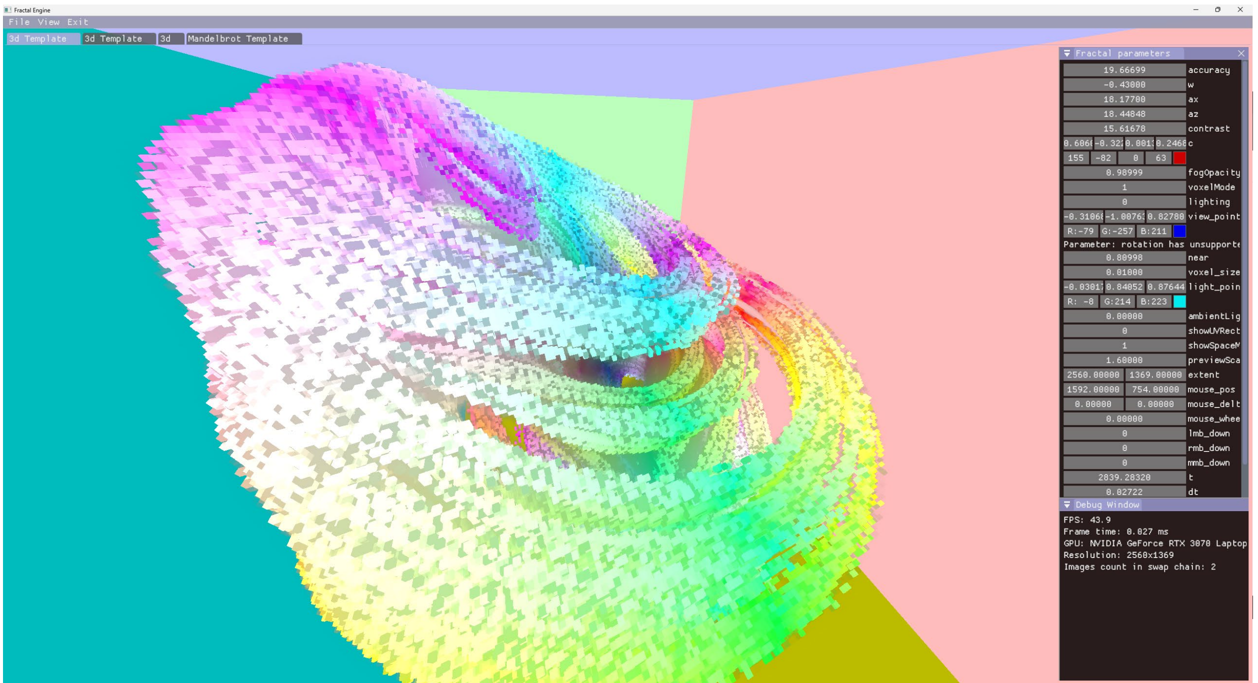

Рисунок 6.42 – Застосування редактора для відображення тривимірної проекції 4D множини Жуліа методом відстеження променів по вокселям

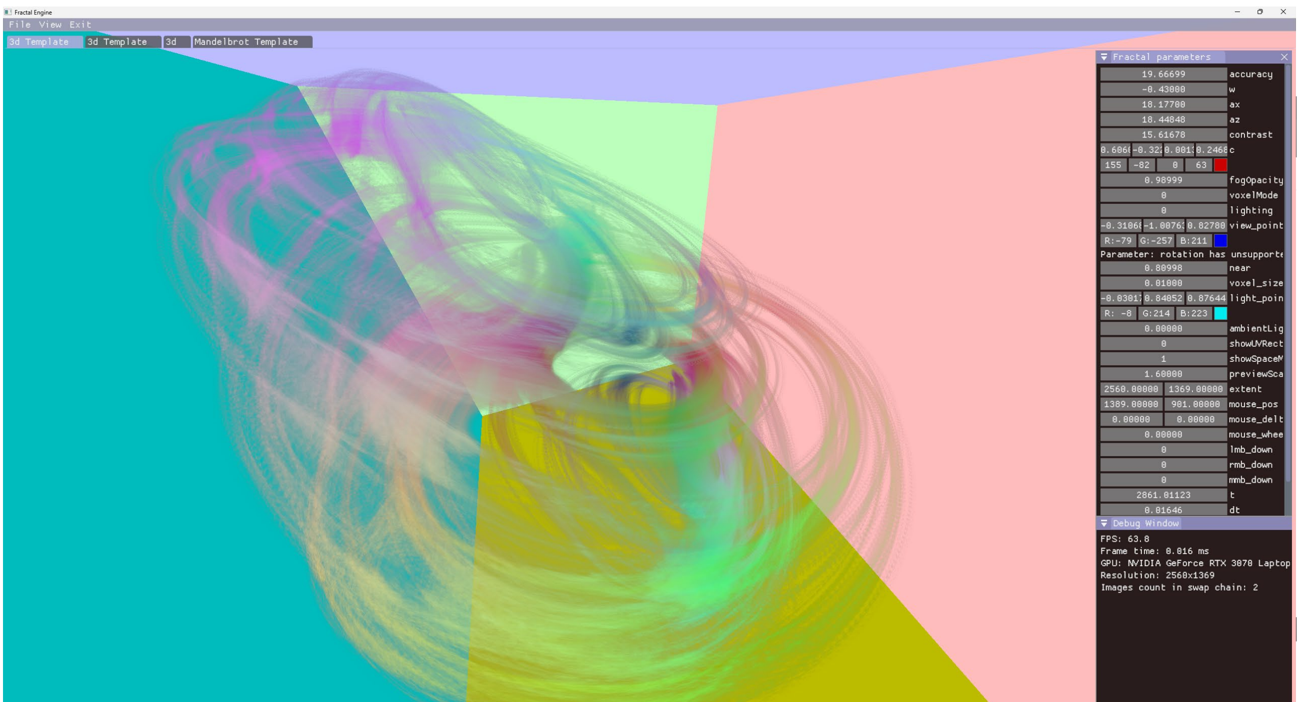

Рисунок 6.43 – Застосування редактора для відображення тривимірної проекції 4D множини Жуліа методом відстеження променів з фіксованим кроком

7 СТВОРЕННЯ ДЕМОНСТАЦІЙНОЇ СЦЕНИ З ВИКОРИСТАННЯМ ФРАКТАЛІВ В ЯКОСТІ ТЕКСТУР

Тут продемонстровано як застосовувати розроблені текстури при створенні сцен на прикладі графічних рушіїв Unreal Engine 5 та Unity.

7.1 Експорт текстури в Unreal Engine

В якості прикладу взято анімований матеріал з підрозділу 6.5. Спершу, в редакторі розроблена текстура експортована в HLSL. Тут необхідно лише тіло функції.

В UE створено новий матеріал. До графу додано вузол «Custom», що дозволяє виконувати довільний HLSL код. У поле «Code» вставляться скопійований код текстури. Далі додаються параметри, що будуть прийматися на вхід. В даному випадку це текстурні координати `uv` та час `t`. На вхід під'єднано відповідні параметри графічного рушія: час та текстурні координати. Вихід тепер можна під'єднати до вихідних параметрів матеріалу, для демонстрації обрано колір. Результат створеного графу матеріалу на рисунку 7.1.

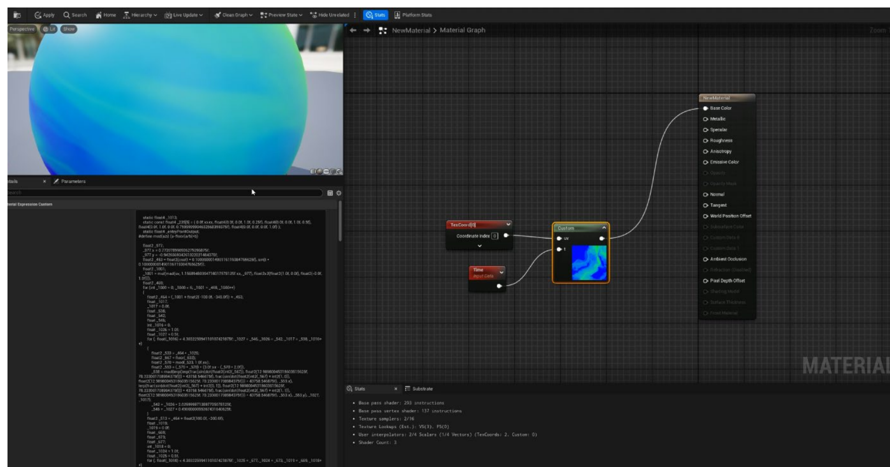

Рисунок 7.1 – Створення матеріалу в UE5 на основі експортованого HLSL коду

Тепер створений матеріал можна застосувати до моделі (рис. 7.2).

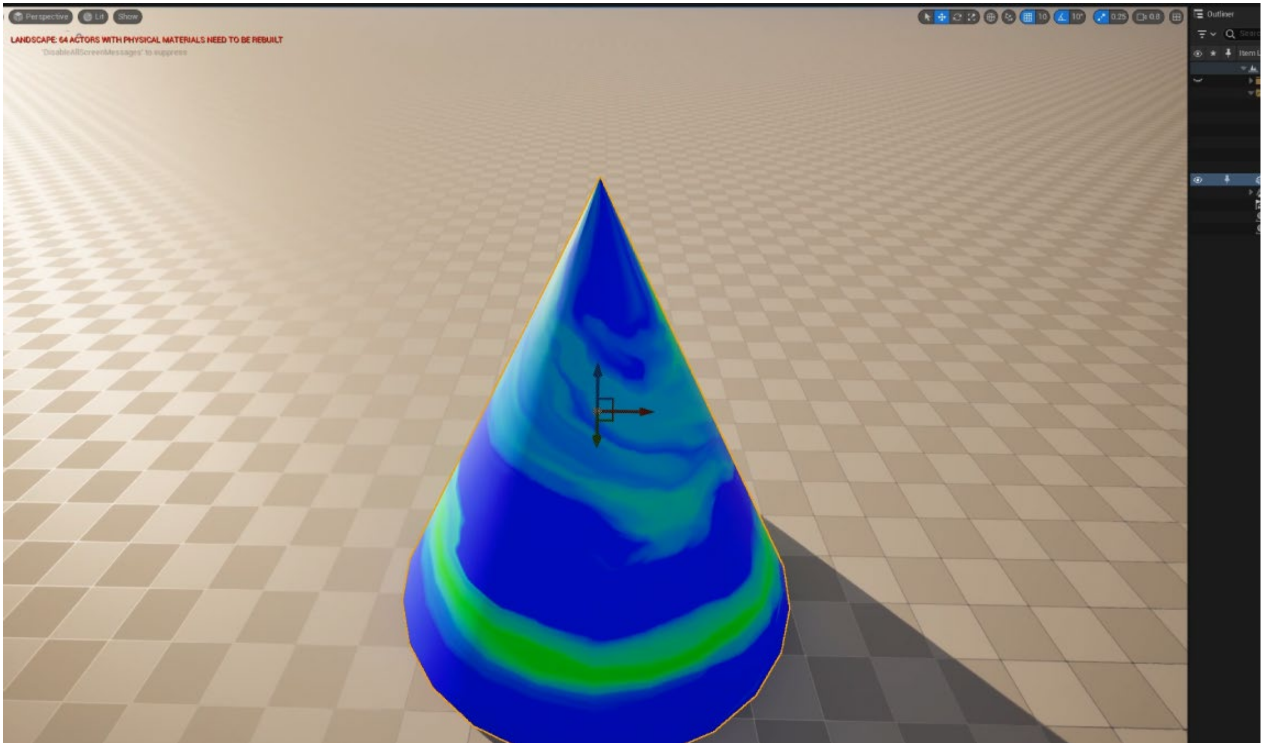

Рисунок 7.2 – Застосування матеріалу до об’єкта в UE5

Провівши додаткове тестування, виявлено, що час відображення для растрової та динамічної текстури приблизно однаковий. Для тестування матеріал було застосовано до ландшафту при вимкненому освітленні для зменшення сторонніх обчислень. В обох випадках час відображення кадра на GPU займав близько 5 мілісекунд. Результат отримано на GPU Nvidia RTX 3070 mobile, може відрізнятися на інших пристроях.

7.1 Експорт текстури в Unity

При експортуванні в Unity необхідно спочатку створити новий шейдер (рис. 7.3). В залежності від типу створеного шейдера, необхідно знайти місце де визначено фрагментний шейдер. Для випадку «Unlit Shader» це функція `frag`.

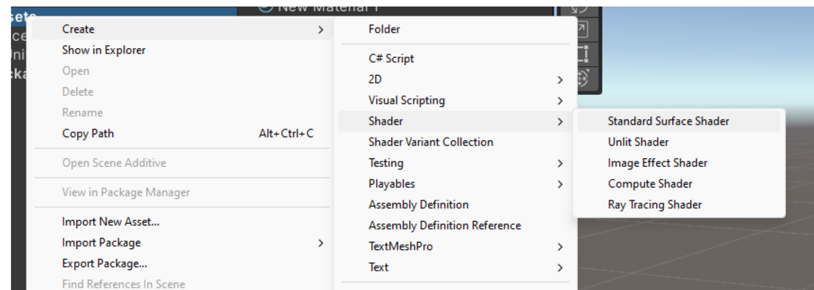

Рисунок 7.3 – Створення шейдеру

Далі необхідно додати до коду експортовану функцію текстури і потім викликати її в `frag` замість `text2D` (лістинг 7.1).

```
fixed4 frag (v2f i) : SV_Target
{
      // sample the texture
      fixed4 col = generated(i.uv);
      // apply fog
      UNITY_APPLY_FOG(i.fogCoord, col);
      return col;
}
```

Лістинг 7.1 – Модифікація шейдеру для відображення динамічної текстури

Після цього, шейдер зберігається і на його основі створюється матеріал, що може бути тепер використаний для об’єктів сцени (рис. 7.4).

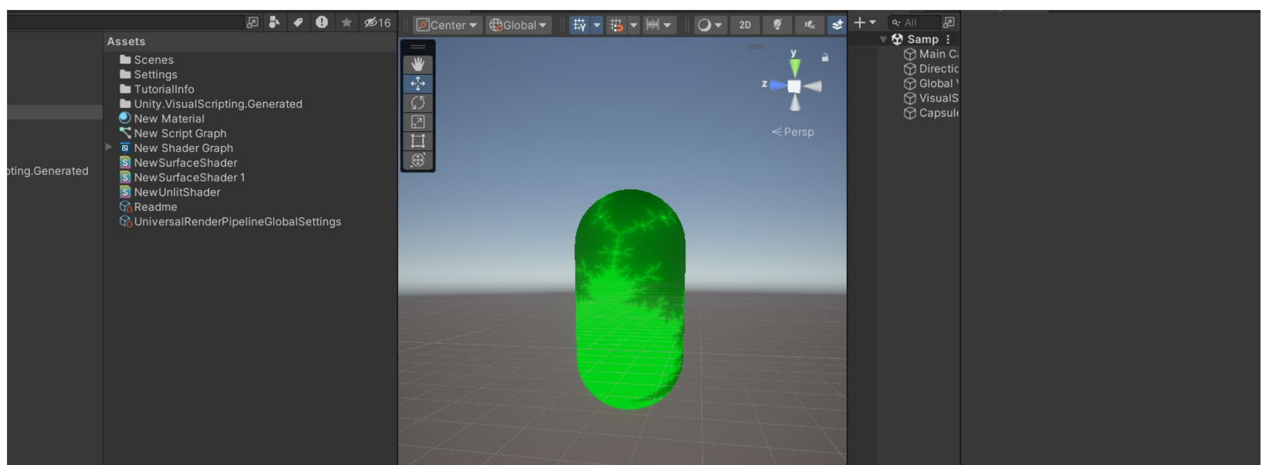

Рисунок 7.4 – Застосування матеріалу до об’єкта в Unity

ВИСНОВКИ

Розроблений редактор має багато корисних застосувань, що виходять за межі створення двовимірних фракталів. Завдяки своїй гнучкій структурі дозволяє створювати не лише двовимірні, а й тривимірні матеріали і анімації. Також, за необхідності, можна відобразити і одновимірні функції. Може бути застосований з метою прототипування нових шейдерів майже будь-якого призначення. Буде корисним для опанування програмування шейдерів, оскільки не має зайвого функціоналу, завдяки чому дозволяє сконцентруватися на коді. Низька затримка з майже миттєвим оновленням візуалізації, відповідно до змін в коді, є значною перевагою.

Деякі з функцій які ще має сенс реалізувати:

1) підтримка експорту для інших мов затінення, таких як MSL та SPIR-V асемблер;

2) інтерпретація інших мов затінення при розробці текстури на ряду з HLSL та переклад коду між ними: це дозволить користувачу розробляти текстури мовою до якої він звик та, при необхідності, полегшить перехід та опанування інших мов;

3) графічний інтерфейс побудови текстури на основі графу вузлів, який буде однозначно відповідати коду: може спростити розробку для нових користувачів;

4) можливість подавати на вхід растрову текстуру: може стати в нагоді для розробки різних ефектів постобробки зображень, або для створення текстур, що базуються на растровому зображенні з певними процедурними модифікаціями;

5) підтримка масивів та структур в якості параметрів;

6) можливість одночасної розробки багатоканального матеріалу;

7) інші формати експорту крім вихідного коду у форматі функції шейдеру і растрового зображення. Наприклад, це можуть бути файли матеріалу специфічні для деякого графічного рушія;

8) підтримка файлів-бібліотек, які можна створити або завантажити в редактор, після чого використовувати при створенні нових текстур. Розробити набір вбудованих бібліотек з часто використовуваними функціями.

Під час виконання роботи створено декілька демонстраційних текстур на базі фракталів з метою демонстрації функціоналу редактора. В подальшому редактор може бути застосований для створення більш складних текстур, в тому числі, природніх об’єктів.

Вихідний код редактора завантажено до репозиторію Git Hub [46].

Фрактали має сенс застосовувати у випадках моделювання природних об’єктів, але і не тільки. Вони мають структуру, що поєднує структурованість і хаотичність, яка часто притаманна природним об’єктам. Використання фракталів не зможе повністю замінити растрові текстури. Так як хоча структура фракталів і схожа на реальні об’єкти вона не абсолютно точно відповідає дійсним об’єктам, тоді як спроба згенерувати більш точну відповідність реальності може вилитись в дуже складні обчислення. З іншої точки зору, зовсім не завжди потрібна абсолютна відповідність реальності, а лише її апроксимація, в такому випадку використання фракталів має сенс, їх унікальна та складна структура можуть бути використані з метою художнього оформлення та принести деяку особливість до продукту. При використані графіки для створення фільмів, мультфільмів чи комп’ютерних ігор, як правило, відтворення реальності не є головною задачею, і навіть навпаки не бажана для певних жанрів. Не завжди реалістичність робить продукт краще, а іноді без потреби ускладнює та погіршує його. Якраз у таких випадках використання фракталів може бути корисним для візуалізації різних ефектів, текстур, ландшафтів, та інших абстрактних візерунків. Крім того, можна побачити, що за допомогою створеного редактора досить не складно генерувати різні текстури, що будуть відповідати певному стилю, це можна зробити трохи змінюючи декілька з параметрів, як наприклад зміщення, точність, масштаб, колір, та інші константні значення. Це може бути дуже корисним при моделюванні певної локації в одному стилі, але з різними текстурами.

Найкращого застосування динамічні текстури на основі фракталів знайдуть у розробці комп’ютерних ігор, особливо для мобільних пристроях де є обмеження на пам’ять та потужність GPU.

Ще однією з переваг текстур основаних лиже на коді є те, що вони значно швидше завантажуються в пам’ять, порівняно з растровими текстурами, де завантаження всіх текстур, в залежності від розміру сцени та потужності комп’ютера, може сягати декількох хвилин.

Також необхідно зазначити, що підхід до динамічної генерації текстур під час візуалізації має сенс здебільшого при візуалізації в реальному часі на обладнані з обмеженими ресурсами. Якщо час візуалізації та ресурси комп’ютера не є проблемою, то використання растрових зображень з дуже високою роздільною здатністю може бути зручнішими та більш гнучким варіантом.

Алгоритм динамічного відображення текстур можна модернізувати для поєднання високої точності та низької затрати пам’яті процедурних динамічних текстур із порівняно константною складністю при використані растрового зображення. Для цього необхідно замість розрахунку на кожному кадрі, виконувати періодичні оновлення растрової текстури фіксованого розміру для частини, що наразі видна користувачу або знаходиться поруч, враховуючи точність деталізації в залежності від масштабу об’єкту на екрані. Після чого, відображати вже растрову текстуру за класичним підходом. Таким чином, необхідно створити проміжний крок на якому будуть генеруватися текстури, що наразі необхідні, з необхідною точністю та необхідному розмірі зображення. В результаті загальна кількість обчислень має зменшиться. В деякому сенсі, така модернізація алгоритму має багато спільного зі алгоритмом створенням MIP карт [47], коли растрова текстура певного розміру обирається в залежності від відстані до об’єкта з метою прискорення мультисемплінгу віддалених об’єктів одночасно маючи високу точність для близьких.

Матеріали роботи доповідались на XXVII Міжнародній науково-практичній конференції «Prospects of Scientific Research in the Conditions of the Modern World», яка відбулась 12-14 червня, 2024 р. в Роттердам, Нідерланди [48].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Texture Sampling. [Електронний ресурс] – Режим доступу: https://vfxdoc.readthedocs.io/en/latest/textures/sampling/
2. Tessellation Stages. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-tessellation
3. Растровий графічний редактор Adobe Photoshop. [Електронний ресурс] – Режим доступу: https://www.adobe.com/ua/products/photoshop.html
4. Растровий графічний редактор GIMP. [Електронний ресурс] – Режим доступу: https://www.gimp.org/
5. DALL·E 3. [Електронний ресурс] – Режим доступу: https://openai.com/index/dall-e-3/
6. Stable Diffusion 3. [Електронний ресурс] – Режим доступу: https://stability.ai/news/stable-diffusion-3
7. Растровий графічний редактор Microsoft Paint. [Електронний ресурс] – Режим доступу: https://www.microsoft.com/en-us/windows/paint
8. Definition of procedural texture. [Електронний ресурс] – Режим доступу: https://www.pcmag.com/encyclopedia/term/49743/procedural-texture
9. NVIDIA DLSS 3. [Електронний ресурс] – Режим доступу: https://www.nvidia.com/en-us/geforce/technologies/dlss/
10. Direct3D 12 reference https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-reference
11. OpenGL Overview. [Електронний ресурс] – Режим доступу: https://www.khronos.org/opengl/
12. Vulkan. [Електронний ресурс] – Режим доступу: https://www.vulkan.org/
13. Metal API. [Електронний ресурс] – Режим доступу: https://developer.apple.com/metal/
14. Rendering Pipeline Overview. [Електронний ресурс] – Режим доступу: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

15. Shader. [Електронний ресурс] – Режим доступу: https://www.khronos.org/opengl/wiki/Shader
16. OpenGL Shading Language. [Електронний ресурс] – Режим доступу: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language
17. High-level shader language (HLSL). [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl
18. Metal Shading Language Specification. [Електронний ресурс] – Режим доступу: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf
19. Unity. [Електронний ресурс] – Режим доступу: https://unity.com/
20. Unreal Engine. [Електронний ресурс] – Режим доступу: https://www.unrealengine.com/
21. GeForce RTX 4090 [Електронний ресурс] – Режим доступу:https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4090/
22. All You Need To Know About SIMD – And How GPUs Employ It. [Електронний ресурс] – Режим доступу: https://acecloud.ai/resources/blog/all-about-simd-and-how-gpus-employ-it/
23. Reference for HLSL: if Statement. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-if
24. Procedural ANIMATED-ORGANIC material, 100% shader. [Електронний ресурс] – Режим доступу: https://www.reddit.com/r/Unity3D/comments/12fkdrt/procedural_animatedorganic_material_100_shader/
25. Seascape. [Електронний ресурс] – Режим доступу: https://www.shadertoy.com/view/Ms2SD1
26. 3D Текстура фрактального ландшафту. [Електронний ресурс] – Режим доступу: https://twigl.app/?ol=true&ss=-Nq5-BV8GOj1YK4902L8

27. Painting a Landscape with Maths. [Електронний ресурс] – Режим доступу: https://www.youtube.com/watch?v=BFld4EBO2RE
28. Adobe Substance Designer. [Електронний ресурс] – Режим доступу:
29. Blender 4.1 Manual: Shader Editor. [Електронний ресурс] – Режим доступу: https://docs.blender.org/manual/en/latest/editors/shader_editor.html
30. Shadertoy. [Електронний ресурс] – Режим доступу: https://www.shadertoy.com/
31. Reference for HLSL: Function Declaration Syntax. [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-function-syntax
32. About SDL. [Електронний ресурс] – Режим доступу: https://www.libsdl.org/
33. Dear ImGui. [Електронний ресурс] – Режим доступу: https://github.com/ocornut/imgui
34. Shaderc. [Електронний ресурс] – Режим доступу: https://github.com/google/shaderc
35. SPIRV-Cross. [Електронний ресурс] – Режим доступу: https://github.com/KhronosGroup/SPIRV-Cross
36. JSON for Modern C++. [Електронний ресурс] – Режим доступу: https://github.com/nlohmann/json
37. stb (single-file public domain (or MIT licensed) libraries for C/C++). [Електронний ресурс] – Режим доступу: https://github.com/nothings/stb
38. Native File Dialog Extended. [Електронний ресурс] – Режим доступу: https://github.com/btzy/nativefiledialog-extended
39. ImGuiColorTextEdit(Syntax highlighting text editor for ImGui). [Електронний ресурс] – Режим доступу: https://github.com/BalazsJako/ImGuiColorTextEdit
40. CMake: A Powerful Software Build System. [Електронний ресурс] – Режим доступу: https://cmake.org/
41. Renormalizing the Mandelbrot Escape. [Електронний ресурс] – Режим доступу: https://linas.org/art-gallery/escape/escape.html

42. Markus, Mario; Hess, Benno (1989). "Lyapunov exponents of the logistic map with periodic forcing". Computers and Graphics. 13 (4): 553–558.
43. The Book of Shaders by Patricio Gonzalez Vivo & Jen Lowe : Noise. [Електронний ресурс] – Режим доступу: https://thebookofshaders.com/11/
44. The Book of Shaders by Patricio Gonzalez Vivo & Jen Lowe : Fractal Brownian Motion. [Електронний ресурс] – Режим доступу: https://thebookofshaders.com/13/
45. Keenan Crane, Ray Tracing Quaternion Julia Sets on the GPU / University of Illinois at Urbana-Champaign, November 7, 2005
46. Git Hub репозиторій розробленого редактора. [Електронний ресурс] – Режим доступу: https://github.com/uvec3/ipte
47. Mipmaps introduction. [Електронний ресурс] – Режим доступу: https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html
48. Sokur Oleksii, The applying of fractals as dynamic textures in computer graphics / Sokur Oleksii, Petrushyna Tetiana // Collection of abstracts of the XXVII International Scientific and Practical Conference «Prospects of Scientific Research in the Conditions of the Modern World», June 12-14, 2024, Rotterdam, Netherlands – PP. 103-106. – Access Mode: https://isu-conference.com/arkhiv/prospects-of-scientific-research-in-the-conditions-of-the-modern-world/

ДОДАТОК А

Коди клавіш, що доступні для використання в редакторі

Цей список визначень неявно додається перед компіляцією до коду текстури:

```
#define KEY_Tab 1
#define KEY_LeftArrow 2
#define KEY_RightArrow 3
#define KEY_UpArrow 4
#define KEY_DownArrow 5
#define KEY_PageUp 6
#define KEY_PageDown 7
#define KEY_Home 8
#define KEY_End 9
#define KEY_Insert 10
#define KEY_Delete 11
#define KEY_Backspace 12
#define KEY_Space 13
#define KEY_Enter 14
#define KEY_Escape 15
#define KEY_LeftCtrl 16
#define KEY_LeftShift 17
#define KEY_LeftAlt 18
#define KEY_LeftSuper 19
#define KEY_RightCtrl 20
#define KEY_RightShift 21
#define KEY_RightAlt 22
#define KEY_RightSuper 23
#define KEY_Menu 24
#define KEY_0 25
#define KEY_1 26
#define KEY_2 27
#define KEY_3 28
#define KEY_4 29
#define KEY_5 30
#define KEY_6 31
#define KEY_7 32
#define KEY_8 33
#define KEY_9 34
#define KEY_A 35
#define KEY_B 36
#define KEY_C 37
#define KEY_D 38
#define KEY_E 39
#define KEY_F 40
#define KEY_G 41
#define KEY_H 42
#define KEY_I 43
#define KEY_J 44
#define KEY_K 45
#define KEY_L 46
#define KEY_M 47
#define KEY_N 48
#define KEY_O 49
#define KEY_P 50
#define KEY_Q 51
#define KEY_R 52
#define KEY_S 53
#define KEY_T 54
#define KEY_U 55
#define KEY_V 56
#define KEY_W 57
#define KEY_X 58
#define KEY_Y 59
#define KEY_Z 60
#define KEY_F1 61
#define KEY_F2 62
#define KEY_F3 63
#define KEY_F4 64
#define KEY_F5 65
#define KEY_F6 66
#define KEY_F7 67
#define KEY_F8 68
#define KEY_F9 69
#define KEY_F10 70
#define KEY_F11 71
#define KEY_F12 72
#define KEY_Apostrophe 73
#define KEY_Comma 74
#define KEY_Minus 75
#define KEY_Period 76
#define KEY_Slash 77
#define KEY_Semicolon 78
#define KEY_Equal 79
#define KEY_LeftBracket 80
#define KEY_Backslash 81
#define KEY_RightBracket 82
#define KEY_GraveAccent 83
#define KEY_CapsLock 84
#define KEY_ScrollLock 85
#define KEY_NumLock 86
#define KEY_PrintScreen 87
#define KEY_Pause 88
#define KEY_Keypad0 89
#define KEY_Keypad1 90
#define KEY_Keypad2 91
#define KEY_Keypad3 92
#define KEY_Keypad4 93
#define KEY_Keypad5 94
#define KEY_Keypad6 95
#define KEY_Keypad7 96
#define KEY_Keypad8 97
#define KEY_Keypad9 98
#define KEY_KeypadDecimal 99
#define KEY_KeypadDivide 100
#define KEY_KeypadMultiply 101
#define KEY_KeypadSubtract 102
#define KEY_KeypadAdd 103
#define KEY_KeypadEnter 104
#define KEY_KeypadEqual 105
#define KEY_MouseLeft 130
#define KEY_MouseRight 131
#define KEY_MouseMiddle 132
#define KEY_MouseX1 133
#define KEY_MouseX2 134
#define KEY_MouseWheelX 135
#define KEY_MouseWheelY 136
#define KEY_Ctrl 137
#define KEY_Shift 138
#define KEY_Alt 139
#define KEY_Super 140
```

ДОДАТОК Б

Повний код розроблених текстур

Множина Мандельброта і Жуліа:

```
struct Parameters
{
    //space transform:
    float2 move;
    float zoom;
    float rotation;

    //preview parameters:
    bool showUVRect;
    bool showSpaceMesh;
    float previewScale;

    //fractal parameters:
    float accuracy;
    float contrast;
    bool removeBanding;
    int mode;//Mandelbrot or Julian (add current point or add
constant)
    float2 c;//Julian constant
    float4 julia_color;
    float4 mandelbrot_color;

    //reserved parameters,autoupdated by app:
    float2 extent;//render area extent in pixels
    float2 mouse_pos;//current mouse position in pixels
    float2 mouse_delta;//mouse delta for the last frame in pixels
    float mouse_wheel_delta;//mouse wheel delta for the last
frame
    bool lmb_down;//if left mouse button down
    bool rmb_down;//if right mouse button down
    bool mmb_down;//if middle mouse button down
    float t;//time counter
    float dt;//time passed from last frame
};

//define parameters variable p
InputParameters(Parameters,p);

float fractal(float2 x, float2 c)
{
    int i=0;
    float2 z=x;
    while(((z.x*z.x+z.y*z.y)<20)&&i<=p.accuracy)
    {
        z= float2(z.x * z.x - z.y * z.y, 2 * z.x * z.y)+c;
        ++i;
    }
    if(i>=p.accuracy)
        return 1;
```

```
        float val=i;
        if(p.removeBanding)
            val=clamp(val-log(log(length(z)))/log(2)+1,0,p.accuracy);
        return pow(val/p.accuracy,p.contrast);
    }

    float2 uvToSpacePoint(float2 uv)
    {
        float2x2 rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return rot*(uv / p.zoom +p.move);
    }

    float2 spacePointToUv(float2 spacePoint)
    {
        float2x2 rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return  (transpose(rot)*spacePoint-p.move)*p.zoom;
    }

    //This function will be the result of export
    //Pass parameters through function arguments for dynamic
parameters
    float4 output(float2 uv)
    {
        float2 pos = uvToSpacePoint(uv);

        float4 result=0;
        if(p.mode!=0)
               result+= fractal(pos,p.c)*p.julia_color;
        if(p.mode!=2)
                result+= fractal(pos,pos)*p.mandelbrot_color;
           return result;
    }

    float2 screenToUV(float2 screenPosition)
    {
        screenPosition.y=-screenPosition.y;
        float2 identitySquare;
        if(p.extent.x>p.extent.y)
            identitySquare=float2(p.extent.y/p.extent.x,1);
        else
            identitySquare=float2(1,p.extent.x/p.extent.y);
        return
(screenPosition+identitySquare*p.previewScale*0.5)/identitySquare/p.pr
eviewScale;
    }

    //Function is called for each pixel
    //and returns color for this pixel
    //you can use it to preview output function
    //this does not affect the export result
    float4 main(float4 position) : SV_TARGET
    {
        float2 positionT=screenToUV(position);
```

```
          if (length(positionT-
spacePointToUv(p.c))*p.previewScale<0.01)
              return float4(1,0,0,1);

          if(p.showUVRect)
          {
              float w=0.005/p.previewScale;
              float2 fr= frac(positionT);
              if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                  return 1-output(positionT);
          }
          if(p.showSpaceMesh)
          {
              float w=0.005/p.previewScale;
              float2 fr= frac(uvToSpacePoint(positionT));
              if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                  return float4(0.5,0.5,1,1);
          }

             return output(positionT);
      }


      //this function called only once per frame before main
      //Allows to control parameters in one place
      void UpdateParameters(inout Parameters p)
      {
          //scale with mouse wheel
          if(!isKeyDown(KEY_Shift)&&!isKeyDown(KEY_Alt))
          {
              float2 mouseBeforeZoom=screenToUV(p.mouse_pos/p.extent*2-
1);
              float zoomScale=exp(p.mouse_wheel_delta*0.1);
              float2 mouseAfterZoom=mouseBeforeZoom*zoomScale;
              p.zoom*=zoomScale;
              p.move-=(mouseBeforeZoom-mouseAfterZoom)/p.zoom;
           }
           else if(isKeyDown(KEY_Alt))//rotate with Alt and mouse wheel
           {
               float delta=p.mouse_wheel_delta*0.1;
               p.rotation+=delta;
               float2x2 rot=float2x2(cos(delta),sin(delta),-
sin(delta),cos(delta));
               p.move=p.move*rot;
           }

          //movement with mouse
          if(p.lmb_down)
              p.move-=float2(p.mouse_delta.x,-
p.mouse_delta.y)/p.extent.y/p.zoom*2/p.previewScale;

          //scale and move with keyboard
          float scaleUpdate=exp((isKeyDown(KEY_E)-isKeyDown(KEY_Q)
)*p.dt);
          p.zoom=p.zoom*scaleUpdate;
          p.move-=(0.5-0.5*scaleUpdate)/p.zoom;
```

```
		float2 move_delta=float2(isKeyDown(KEY_D)-
isKeyDown(KEY_A),isKeyDown(KEY_W)-isKeyDown(KEY_S));
		p.move+=move_delta*p.dt/p.zoom;

		//preview parameters control
		if(isKeyPressed(KEY_1))
			p.showUVRect=!p.showUVRect;
		if(isKeyPressed(KEY_2))
			p.showSpaceMesh=!p.showSpaceMesh;
		if(isKeyDown(KEY_Shift))
			p.previewScale*=exp(p.mouse_wheel_delta*0.1);

		//limit parameters
		if(p.zoom<=0)
			p.zoom=0.1;

		p.accuracy*=exp((isKeyDown(KEY_UpArrow)-
isKeyDown(KEY_DownArrow))*p.dt);
		p.contrast*=exp((isKeyDown(KEY_RightArrow)-
isKeyDown(KEY_LeftArrow))*p.dt*5);



p.mode=(p.mode+(isKeyPressed(KEY_F)||isKeyReleased(KEY_F)))%4;
		//c point
		if(p.rmb_down)
			p.c=uvToSpacePoint(screenToUV(p.mouse_pos/p.extent*2-1));
	}
```

Фрактал Ньютона:

```
	//define parameters
	struct Parameters
	{
		//space transform:
		float2 move;
		float zoom;
		float rotation;

		//mandelbrot-like fractal specific parameters:
		float accuracy;
		int mode;//Mandelbrot or Julian (add current point or add
constant)
		float2 c;//Julian constant
		float contrast;
		float radius;
		float k;
		float4 color;
		float a;


		//preview parameters:
		bool showUVRect;
		bool showSpaceMesh;
		float previewScale;
```

```
        //reserved parameters,autoupdated by app:
        float2 extent;//render area extent in pixels
        float2 mouse_pos;//current mouse position in pixels
        float2 mouse_delta;//mouse delta for the last frame in pixels
        float mouse_wheel_delta;//mouse wheel delta for the last frame
        bool lmb_down;//if left mouse button down
        bool rmb_down;//if right mouse button down
        bool mmb_down;//if middle mouse button down
        float t;//time counter
        float dt;//time passed from last frame
    };

    //define parameters variable p
    InputParameters(Parameters,p);


    float2 cmul(float2 z1, float2 z2)
    {
        return float2(z1.x*z2.x-z1.y*z2.y,z1.x*z2.y+z1.y*z2.x);
    }

    float2 cdiv(float2 z1, float2 z2)
    {
        float2 z2c= float2(z2.x,-z2.y);
        return cmul(z1,z2c)/(z2.x*z2.x+z2.y*z2.y);
    }

    float2 cpow(float2 z, int p)
    {
        float2 result=1;
        for(int i=0;i<p;++i)
            result=cmul(result,z);
        return result;
    }

    float2 f(float2 z)
    {
        return cpow(z,3)-1;
    }

    float2 df(float2 z)
    {
        return 3*cpow(z,2);
    }

    const            float2            roots[3]={float2(1,0),float2(-
0.5,sqrt(3)/2),float2(-0.5,-sqrt(3)/2)};
    const                                                     float3
colors[3]={float3(1,0,0),float3(0,1,0),float3(0,0,1)};

    float4 color(float2 z)
    {
        float3 result=0;
        for(int i=0;i<3;++i)

                result+=(1-length(roots[i]-z))*colors[i];
        return float4(result,1);
```

```
    }

    float4 fractal(float2 sp,int accuracy,float2 c)
    {
        int i=0;
        float r_sqr=p.radius*p.radius;
        while(i<=accuracy)
        {
            sp= sp-p.a*cdiv(f(sp),df(sp))+c;
            ++i;
        }

        return color(sp);
    }


    float2 uvToSpacePoint(float2 uv)
    {
        float2x2       rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return rot*(uv / p.zoom +p.move);
    }

    float2 spacePointToUv(float2 spacePoint)
    {
        float2x2       rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return  (transpose(rot)*spacePoint-p.move)*p.zoom;
    }

    //This function will be the result of export
    //Pass parameters through function arguments for dynamic parameters
    float4 output(float2 uv, float accuracy, float4 color)
    {
        float2 pos = uvToSpacePoint(uv);
        float4 result=0;
        if(p.mode!=0)
               result+= fractal(pos,p.accuracy,p.c);
        if(p.mode!=2)
                result+= fractal(pos,p.accuracy,pos);

           if(p.mode%2==1)
               result/=2;

           return result;
    }

    float2 screenToUV(float2 screenPosition)
    {
        screenPosition.y=-screenPosition.y;
        float2 identitySquare;
        if(p.extent.x>p.extent.y)
            identitySquare=float2(p.extent.y/p.extent.x,1);
        else
            identitySquare=float2(1,p.extent.x/p.extent.y);
        return
(screenPosition+identitySquare*p.previewScale*0.5)/identitySquare/p.pr
eviewScale;
```

```
      }

      //this function is called for each pixel
      //and returns color for this pixel
      //you can use it to preview output function
      //this does not affect the export result
      float4 main(float4 position) : SV_TARGET
      {
          float2 positionT=screenToUV(position);

          if(length(positionT-
spacePointToUv(p.c))*p.previewScale<0.005)
              return float4(1,0,0,1);

          float4 result=output(positionT,p.accuracy,p.color);
          if(p.showUVRect)
          {
              float w=0.005/p.previewScale;
              float2 fr= frac(positionT);
              if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                  return 1-result;
          }
          if(p.showSpaceMesh)
          {
              float w=0.005/p.previewScale;
              float2 fr= frac(uvToSpacePoint(positionT));
              if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                  return float4(0.5,0.5,1,1);
          }

             return result;
      }


      //function called only once per frame before main
      //Allows to control parameters in one place
      void UpdateParameters(inout Parameters p)
      {
          //fractal parameters control
          //c point
          if(p.rmb_down)
              p.c=uvToSpacePoint(screenToUV(p.mouse_pos/p.extent*2-1));

          //switch mode
          if(isKeyPressed(KEY_F)||isKeyReleased(KEY_F))
              p.mode=(p.mode+1)%4;

          p.accuracy*=exp((isKeyDown(KEY_UpArrow)-
isKeyDown(KEY_DownArrow))*p.dt);
          p.contrast*=exp((isKeyDown(KEY_RightArrow)-
isKeyDown(KEY_LeftArrow))*p.dt*5);

          //scale with mouse wheel
          if(!isKeyDown(KEY_Shift)&&!isKeyDown(KEY_Alt))
          {
              float2  mouseBeforeZoom=screenToUV(p.mouse_pos/p.extent*2-
1);
              float zoomScale=exp(p.mouse_wheel_delta*0.1);
```

```
                float2 mouseAfterZoom=mouseBeforeZoom*zoomScale;
                p.zoom*=zoomScale;
                p.move-=(mouseBeforeZoom-mouseAfterZoom)/p.zoom;
            }
            else if(isKeyDown(KEY_Alt))//rotate with Alt
            {
                 float delta=p.mouse_wheel_delta*0.1;
                 p.rotation+=delta;
                 float2x2           rot=float2x2(cos(delta),sin(delta),-
sin(delta),cos(delta));
                 p.move=p.move*rot;
            }

           //movement with mouse
           if(p.lmb_down)
               p.move-=float2(p.mouse_delta.x,-
p.mouse_delta.y)/p.extent.y/p.zoom*2/p.previewScale;


           //scale and move with keyboard
           float      scaleUpdate=exp((isKeyDown(KEY_E)-isKeyDown(KEY_Q)
)*p.dt);
           p.zoom=p.zoom*scaleUpdate;
           p.move-=(0.5-0.5*scaleUpdate)/p.zoom;

           float2                    move_delta=float2(isKeyDown(KEY_D)-
isKeyDown(KEY_A),isKeyDown(KEY_W)-isKeyDown(KEY_S));
           p.move+=move_delta*p.dt/p.zoom;



           //preview parameters control
           if(isKeyPressed(KEY_1))
               p.showUVRect=!p.showUVRect;
           if(isKeyPressed(KEY_2))
               p.showSpaceMesh=!p.showSpaceMesh;
           if(isKeyDown(KEY_Shift))
               p.previewScale*=exp(p.mouse_wheel_delta*0.1);


           //limit parameters
           if(p.zoom<=0)
               p.zoom=0.1;
           if(p.radius<0)
               p.radius=0;
           if(isnan(p.move.x)||isnan(p.move.y))
               p.move=0;
           p.mode=clamp(p.mode,0,3);
       }
```

Фрактал Ляпунова:

```
//define parameters
struct Parameters
{
    //space transform:
    float2 move;
       float zoom;
       float rotation;

    //mandelbrot-like fractal specific parameters:
    float accuracy;
    int mode;//Mandelbrot or Julian (add current point or add constant)
    float2 c;//Julian constant
    float contrast;
    float radius;
    float k;
    float4 color;
    bool removeBanding;
    float s0;
    float s1;
    float s2;
    float s3;
    float s4;


    //preview parameters:
    bool showUVRect;
    bool showSpaceMesh;
    float previewScale;

    //reserved parameters,autoupdated by app:
    float2 extent;//render area extent in pixels
    float2 mouse_pos;//current mouse position in pixels
    float2 mouse_delta;//mouse delta for the last frame in pixels
    float mouse_wheel_delta;//mouse wheel delta for the last frame
    bool lmb_down;//if left mouse button down
    bool rmb_down;//if right mouse button down
    bool mmb_down;//if middle mouse button down
    float t;//time counter
    float dt;//time passed from last frame

    float v;
};

//define parameters variable p
InputParameters(Parameters,p);



float r(float2 v, int n)
{
    n=n%5;
    if(n==0)
        return lerp(v[0],v[1],p.s0);
```

```
        if(n==1)
            return lerp(v[0],v[1],p.s1);
        if(n==2)
            return lerp(v[0],v[1],p.s2);
        if(n==3)
            return lerp(v[0],v[1],p.s3);
        if(n==4)
            return lerp(v[0],v[1],p.s4);
        static const float S[]={1,1,0.4,1,0};
        return lerp(v[0],v[1],S[n%5]);
    }

    float fractal(float2 sp,int accuracy,float2 c)
    {
        float x=0.5;
        float sum=0.0;
        for(int i=0;i<accuracy;++i)
        {
            x=r(sp,i)*x*(1-x);
            sum+=log(abs(r(sp,i)*(1-2*x)))/accuracy;
        }
        return pow(-sum,p.contrast);
    }


    float2 uvToSpacePoint(float2 uv)
    {
        float2x2        rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return rot*(uv / p.zoom +p.move);
    }

    float2 spacePointToUv(float2 spacePoint)
    {
        float2x2        rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
        return  (transpose(rot)*spacePoint-p.move)*p.zoom;
    }

    //This function will be the result of export
    //Pass parameters through function arguments for dynamic parameters
    float4 output(float2 uv)
    {
        float2 pos = uvToSpacePoint(uv);
        float4 result=0;
        if(p.mode!=0)
              result+= fractal(pos,p.accuracy,p.c);
        if(p.mode!=2)
               result+= fractal(pos,p.accuracy,pos);

           if(p.mode%2==1)
              result/=2;


           return result*p.color;
    }

    float2 screenToUV(float2 screenPosition)
```

```
    {
        screenPosition.y=-screenPosition.y;
        float2 identitySquare;
        if(p.extent.x>p.extent.y)
            identitySquare=float2(p.extent.y/p.extent.x,1);
        else
            identitySquare=float2(1,p.extent.x/p.extent.y);
        return
(screenPosition+identitySquare*p.previewScale*0.5)/identitySquare/p.pr
eviewScale;
    }

    //this function is called for each pixel
    //and returns color for this pixel
    //you can use it to preview output function
    //this does not affect the export result
    float4 main(float4 position) : SV_TARGET
    {
        float2 positionT=screenToUV(position);

        if(length(positionT-
spacePointToUv(p.c))*p.previewScale<0.005)
            return float4(1,0,0,1);

        if(p.showUVRect)
        {
            float w=0.005/p.previewScale;
            float2 fr= frac(positionT);
            if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                return 1-output(positionT);
        }
        if(p.showSpaceMesh)
        {
            float w=0.005/p.previewScale/p.zoom;
            float2 fr= frac(uvToSpacePoint(positionT));
            if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-step(w,fr.x))
                return float4(0.5,0.5,1,1);
        }

           return output(positionT)*p.color;
    }


    //function called only once per frame before main
    //Allows to control parameters in one place
    void UpdateParameters(inout Parameters p)
    {
        //fractal parameters control
        //c point
        if(p.rmb_down)
            p.c=uvToSpacePoint(screenToUV(p.mouse_pos/p.extent*2-1));

        //switch mode
        if(isKeyPressed(KEY_F)||isKeyReleased(KEY_F))
            p.mode=(p.mode+1)%4;

        p.c.x+=0.13*(cos(p.t+p.dt)-cos(p.t));
        p.c.y+=0.03*(sin(p.t+p.dt)-sin(p.t));
```

```
          p.accuracy*=exp((isKeyDown(KEY_UpArrow)-
isKeyDown(KEY_DownArrow))*p.dt);
          p.contrast*=exp((isKeyDown(KEY_RightArrow)-
isKeyDown(KEY_LeftArrow))*p.dt*5);

          //scale with mouse wheel
          if(!isKeyDown(KEY_Shift)&&!isKeyDown(KEY_Alt))
          {
              float2  mouseBeforeZoom=screenToUV(p.mouse_pos/p.extent*2-
1);
              float zoomScale=exp(p.mouse_wheel_delta*0.1);
              float2 mouseAfterZoom=mouseBeforeZoom*zoomScale;
              p.zoom*=zoomScale;
              p.move-=(mouseBeforeZoom-mouseAfterZoom)/p.zoom;
           }
           else if(isKeyDown(KEY_Alt))//rotate with Alt
           {
               float delta=p.mouse_wheel_delta*0.1;
               p.rotation+=delta;
               float2x2             rot=float2x2(cos(delta),sin(delta),-
sin(delta),cos(delta));
               p.move=p.move*rot;
           }

          //movement with mouse
          if(p.lmb_down)
              p.move-=float2(p.mouse_delta.x,-
p.mouse_delta.y)/p.extent.y/p.zoom*2/p.previewScale;


          //scale and move with keyboard
          float      scaleUpdate=exp((isKeyDown(KEY_E)-isKeyDown(KEY_Q)
)*p.dt);
          p.zoom=p.zoom*scaleUpdate;
          p.move-=(0.5-0.5*scaleUpdate)/p.zoom;

          float2                    move_delta=float2(isKeyDown(KEY_D)-
isKeyDown(KEY_A),isKeyDown(KEY_W)-isKeyDown(KEY_S));
          p.move+=move_delta*p.dt/p.zoom;


          //preview parameters control
          if(isKeyPressed(KEY_1))
              p.showUVRect=!p.showUVRect;
          if(isKeyPressed(KEY_2))
              p.showSpaceMesh=!p.showSpaceMesh;
          if(isKeyDown(KEY_Shift))
              p.previewScale*=exp(p.mouse_wheel_delta*0.1);


          //limit parameters
          if(p.zoom<=0)
              p.zoom=0.1;
          if(p.radius<0)
              p.radius=0;
          if(isnan(p.move.x)||isnan(p.move.y))
              p.move=0;
          p.mode=clamp(p.mode,0,3);
```

```
        if(true)
        {
        p.s0=sin(p.t/50)/3+0.5;
        p.s1=sin(p.t/25)/2+0.5;
        p.s2=sin(p.t/10)/2+0.5;
        p.s3=sin(p.t/5)/2+0.5;
        p.s4=sin(p.t/5)/2+0.5;
        }
    }
```

Кольоровий фрактальний шум відображений у декілька разів викривленому просторі за допомогою фрактального шуму:

```
    //It is possible to redefine float as double for better precision
    //(performance will be decreased significantly)
    //#define float2 double2
    //#define float double

    //define parameters
    struct Parameters
    {
        float2x2 rot;
        uint octaves;
        float distortion;
        float distortionFrequency;
        //space transform:
        float2 move;
        float zoom;
        float rotation;

        //mandelbrot-like fractal specific parameters:
        float accuracy;
        int mode;//Mandelbrot or Julian (add current point or add
constant)
        float2 c;//Julian constant
        float contrast;
        float radius;
        float k;


        float4 color0;
        float4 color1;
        float4 color2;
        float4 color3;


        //preview parameters:
        bool showUVRect;
        bool showSpaceMesh;
        float previewScale;

        //reserved parameters,autoupdated by app:
        float2 extent;//render area extent in pixels
        float2 mouse_pos;//current mouse position in pixels
```

```
    float2 mouse_delta;//mouse delta for the last frame in pixels
    float mouse_wheel_delta;//mouse wheel delta for the last frame
    bool lmb_down;//if left mouse button down
    bool rmb_down;//if right mouse button down
    bool mmb_down;//if middle mouse button down
    float t;//time counter
    float dt;//time passed from last frame
};

//define parameters variable p
InputParameters(Parameters,p);


float random (float2 st)
{
    return frac(sin(dot(st.xy,
                         float2(12.9898,78.233)))
                 * 43758.5453123);
}

float noise(float2 uv)
{
    int2 p=floor(uv);
    float2 f=frac(uv);
    f= f*f*(3.0-2.0*f);

    float r=random(p);
    float rx=random(p+int2(1,0));
    float ry=random(p+int2(0,1));
    float rxy=random(p+int2(1,1));


    float res=lerp(lerp(r,rx,f.x), lerp(ry,rxy,f.x),f.y);
    return res;
}

float2 noise2(float2 uv)
{
     return float2( noise(uv), noise(uv+float2(100,-200)));
}

float fbm(float2 st)
{
     float val=0;
     float2 amplitude=0.5;
     float2 freq=1;
     for(int i=0;i<p.octaves;++i)
     {
          float2x2 r=float2x2(cos(0.3),sin(0.3),
          -sin(0.3),cos(0.3));
          st=r*st;
          val+=noise(st*freq)*amplitude;
          freq*=2;
          amplitude*=0.5;
     }
     return val;
}
```

```
	float2 fbm2(float2 uv)
	{
		return float2( fbm(uv), fbm(uv+float2(100,-200)));
	}


	float fractal(float2 uv)
	{
		float2 sp=uv;
		for(int i=0;i<6;++i)
		{
			float2 n=fbm2(sp*p.distortionFrequency)*2-1;
		sp= sp+n*p.distortion;
		sp*=1.1;
		}
	   return fbm(sp);
		return sp.x%0.1<0.05;
	}


	float2 uvToSpacePoint(float2 uv)
	{
		float2x2      rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
		return rot*(uv / p.zoom +p.move);
	}

	float2 spacePointToUv(float2 spacePoint)
	{
		float2x2      rot=float2x2(cos(p.rotation),sin(p.rotation),-
sin(p.rotation),cos(p.rotation));
		return  (transpose(rot)*spacePoint-p.move)*p.zoom;
	}

	float4 blend(float4 c1, float4 c2, float x)
	{
		return lerp(c1,c2,(x-c1.a)/(c2.a-c1.a));
	}

	//This function will be the result of export
	//Pass parameters through function arguments for dynamic parameters
	float4 output(float2 uv)
	{
		float2 pos = uvToSpacePoint(uv);
		float result= fractal(pos);



		if(result<p.color1.a)
			return blend(p.color0,p.color1,result);
		if(result<p.color2.a)
			return blend(p.color1,p.color2,result);
		if(result<p.color3.a)
			return blend(p.color2,p.color3,result);

		return 0;
	}
```

```
      float2 screenToUV(float2 screenPosition)
      {
            screenPosition.y=-screenPosition.y;
            float2 identitySquare;
            if(p.extent.x>p.extent.y)
                  identitySquare=float2(p.extent.y/p.extent.x,1);
          else
               identitySquare=float2(1,p.extent.x/p.extent.y);
            return
(screenPosition+identitySquare*p.previewScale*0.5)/identitySquare/p.pr
eviewScale;
      }

      //this function is called for each pixel
      //and returns color for this pixel
      //you can use it to preview output function
      //this does not affect the export result
      float4 main(float4 position) : SV_TARGET
      {
            float2 positionT=screenToUV(position);
            if(length(positionT-
spacePointToUv(p.c))*p.previewScale<0.005)
                  return float4(1,0,0,1);
            if(p.showUVRect)
            {
                 float w=0.005/p.previewScale;
                 float2 fr= frac(positionT);
                 if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-
step(w,fr.x))
                  return 1-output(positionT);
            }
            if(p.showSpaceMesh)
            {
                 float w=0.005/p.previewScale;
                 float2 fr= frac(uvToSpacePoint(positionT));
                 if(step(-w,fr.y)-step(w,fr.y)+step(-w,fr.x)-
step(w,fr.x))
                  return float4(0.5,0.5,1,1);
            }

            return output(positionT);
      }


      //function called only once per frame before main
      //Allows to control parameters in one place
      void UpdateParameters(inout Parameters p)
      {
            //fractal parameters control
            //c point
            if(p.rmb_down)
                  p.c=uvToSpacePoint(screenToUV(p.mouse_pos/p.extent*2-
1));

            //switch mode
            if(isKeyPressed(KEY_F)||isKeyReleased(KEY_F))
                  p.mode=(p.mode+1)%4;
```

```
            p.accuracy*=exp((isKeyDown(KEY_UpArrow)-
isKeyDown(KEY_DownArrow))*p.dt*5);
            p.contrast*=exp((isKeyDown(KEY_RightArrow)-
isKeyDown(KEY_LeftArrow))*p.dt*5);

            //scale with mouse wheel
            if(!isKeyDown(KEY_Shift)&&!isKeyDown(KEY_Alt))
            {
                float2
mouseBeforeZoom=screenToUV(p.mouse_pos/p.extent*2-1);
                float zoomScale=exp(p.mouse_wheel_delta*0.1);
                float2 mouseAfterZoom=mouseBeforeZoom*zoomScale;
                p.zoom*=zoomScale;
                p.move-=(mouseBeforeZoom-mouseAfterZoom)/p.zoom;
             }
            else if(isKeyDown(KEY_Alt))//rotate with Alt
            {
                float delta=p.mouse_wheel_delta*0.1;
                p.rotation+=delta;
                float2x2            rot=float2x2(cos(delta),sin(delta),-
sin(delta),cos(delta));
                p.move=p.move*rot;
            }

            //movement with mouse
            if(p.lmb_down)
                 p.move-=float2(p.mouse_delta.x,-
p.mouse_delta.y)/p.extent.y/p.zoom*2/p.previewScale;
            //scale and move with keyboard
            float     scaleUpdate=exp((isKeyDown(KEY_E)-isKeyDown(KEY_Q)
)*p.dt);
            p.zoom=p.zoom*scaleUpdate;
            p.move-=(0.5-0.5*scaleUpdate)/p.zoom;

            float2                    move_delta=float2(isKeyDown(KEY_D)-
isKeyDown(KEY_A),isKeyDown(KEY_W)-isKeyDown(KEY_S));
            p.move+=move_delta*p.dt/p.zoom;
            //preview parameters control
            if(isKeyPressed(KEY_1))
                 p.showUVRect=!p.showUVRect;
            if(isKeyPressed(KEY_2))
                 p.showSpaceMesh=!p.showSpaceMesh;
            if(isKeyDown(KEY_Shift))
                 p.previewScale*=exp(p.mouse_wheel_delta*0.1);
            //limit parameters
            if(p.zoom<=0)
                 p.zoom=0.1;
            if(p.radius<0)
                 p.radius=0;
            p.move=clamp(p.move,-p.radius*100,p.radius*100);
            p.mode=clamp(p.mode,0,3);
            p.color1.a=clamp(p.color1.a,p.color0.a,1);
      }
```

Жуліа 4D (відображення тривимірної текстури):

```
struct Parameters
{
    //custom parameters:
    float accuracy;
    float w;

    float contrast;
    float4 c;

    //View parameters:
    //space transform
    float3 view_point;
    float3x3 rotation;//camera rotation matrix
    float near;//camera near plane
    float ax;//camera angle x axis
    float az;//camera angle z axis

    float voxel_size;

    bool lighting;
    float3 light_point;
    float ambientLight;
    float fogOpacity;
    bool voxelMode; //preview type

    //preview parameters:
    bool showUVRect;
    bool showSpaceMesh;
    float previewScale;

    //reserved parameters,autoupdated by app:
    float2 extent;//render area extent in pixels
    float2 mouse_pos;//current mouse position in pixels
    float2 mouse_delta;//mouse delta for the last frame in pixels
    float mouse_wheel_delta;//mouse wheel delta for the last frame
    bool lmb_down;//if left mouse button down
    bool rmb_down;//if right mouse button down
    bool mmb_down;//if middle mouse button down
    float t;//time counter
    float dt;//time passed from last frame

    float4 dbg;//debug output (used to show color under mouse cursor)
};

//define parameters variable p
InputParameters(Parameters,p);


float4 qmul(float4 q1, float4 q2)
{
    return float4(q1.w*q2.xyz + q2.w*q1.xyz + cross(q1.xyz,q2.xyz),
                  q1.w*q2.w - dot(q1.xyz,q2.xyz));
}

float fractal(float3 v)//Julia 4D fractal
```

```
	{
		float4 x= float4(v.x,v.y,p.w,v.z);
		float4 q= x;
		int i=0;
		while(length(q.xyz)<2&&i<p.accuracy)
		{
			q=qmul(q,q)+p.c;
			++i;
		}
		return pow(i/floor(p.accuracy),p.contrast);
		if(i>=p.accuracy)
			return 1;
		return 0;
	}
	float4 output(float3 v)
	{
		float f=fractal(v);
		return  float4(abs(v),f);//return  coordinates  as  color  and
fractal value as alpha
	}


	float3  side(float3  p,  float3  dir,float3  boxLowerCorner,  float
boxSize)//find which side of the box ray comes out
	{
		float3 checkDirection= sign(dir);
		float3
remainedBox=(checkDirection+1.0f)*(boxSize*0.5f)+boxLowerCorner-p;
		float3 qubeizedDir=dir/remainedBox;//can be qNaN
		float
maxCoord=max(max(qubeizedDir[0],qubeizedDir[1]),qubeizedDir[2]);//max
will discard qNaN when possible
		float3 mask=step(maxCoord, qubeizedDir);
		return mask*checkDirection;
	}

	float3x3 rotZ(float a)
	{
		return float3x3(cos(a),sin(a),0,
				-sin(a),cos(a),0,
				0,0,1);
	}

	float3x3 rotX(float a)
	{
		return float3x3(1,0,0,
				0,cos(a),sin(a),
				0,-sin(a),cos(a)
				);
	}

	float3 screenToSpace(float2 screenPoint,float near)
	{
		float3  n=float3(screenPoint.x*p.extent.x/p.extent.y,p.near,-
screenPoint.y);
		return p.rotation*n;
	}


	float3  lineFlatIntersection(float3  pos,  float3  dir,  float3
flatPos, float3 n)
```

```
	{
		float d=dot(n,flatPos);
		float t=(d-dot(n,pos))/dot(n,dir);
		return pos+t*dir;
	}

	float4 voxelViewer(in float3 pos, in float3 n)
	{
		float3
block_lower_corner=floor(pos/p.voxel_size)*p.voxel_size;

		float4 res=0;
		int maxI=4.0/p.voxel_size;
		for(int i=0;i<maxI;++i)
		{
			float3                                        step=
side(p.view_point,n,block_lower_corner,p.voxel_size);

block_lower_corner=round(block_lower_corner/p.voxel_size+step)*p.voxel
_size;

//block_lower_corner=round(block_lower_corner/p.voxel_size)*p.voxel_si
ze;
			float4 f=output(block_lower_corner+p.voxel_size/2);

			res=float4(res.xyz+f.xyz*(1-res.a)*f.a,1-(1-f.a)*(1-
res.a));

			if(1-res.a<0.0001)
			{
				res.a=1;
				break;
			}
		}
		return res;
	}

	float4 fogViewer(float3 pos, in float3 n)
	{
		n=normalize(n);
		//light loss per 1 unit
		float density=p.fogOpacity;
		const float stepLength=p.voxel_size;
		float stepLoss=1-pow(1-density,stepLength);

		float4 res=0;
		int maxI=3/stepLength;
		for(int i=0;i<maxI;++i)
		{
			float4 f=output(pos);
			f.a*=stepLoss;
			res=float4(res.xyz+f.xyz*(1-res.a)*f.a,1-(1-f.a)*(1-
res.a));
			if(1-res.a<0.0001)
			{
				res.a=1;
				break;
			}
			pos+=n*stepLength;
		}
		return res;
```

```
    }

    //Function is called for each pixel
    //and returns color for this pixel
    //you can use it to preview output function
    //this does not affect the export result
    float4 main(float2 position)
    {
        float3 n=screenToSpace(position,0.3);
        float4 bg=0.5+float4(side(p.view_point,n,-1,2),1)/2;

        float4 res;
        if(p.voxelMode)
            res=voxelViewer(p.view_point, n);
        else
            res=fogViewer(p.view_point, n);


        return float4(lerp(bg,res,res.a).xyz,res.a);
    }

    //this function called only once per frame before main
    //Allows to control parameters in one place
    void UpdateParameters(inout Parameters p)
    {
        if(p.lmb_down)
        {
            p.ax-=p.mouse_delta.y/p.extent.y*radians(360);
            p.az-=p.mouse_delta.x/p.extent.y*radians(360);
        }
        p.rotation=rotZ(p.az)*rotX(p.ax);
        p.near+=p.mouse_wheel_delta/100;

        float3              view_point_delta=float3(isKeyDown(KEY_D)-
isKeyDown(KEY_A),
        isKeyDown(KEY_W)-isKeyDown(KEY_S),
        isKeyDown(KEY_E)-isKeyDown(KEY_Q));
        p.view_point+=p.rotation*view_point_delta*p.dt;

        p.accuracy*=exp((isKeyDown(KEY_UpArrow)-
isKeyDown(KEY_DownArrow))*p.dt*0.2);
        p.contrast*=exp((isKeyDown(KEY_RightArrow)-
isKeyDown(KEY_LeftArrow))*p.dt*0.2);

        if(isKeyPressed(KEY_F))
            p.voxelMode=!p.voxelMode;
        if(isKeyPressed(KEY_L))
            p.lighting=!p.lighting;
        if(p.mmb_down)
            p.light_point=p.view_point;

        if(p.t>3600)//reset time for better precision
            p.t=0;
        if(p.voxel_size<0)
            p.voxel_size=0.001;

        //show return color under mouse cursor
        p.dbg=main(p.mouse_pos/p.extent*2-1);
    }
```

ДОДАТОК В

Варіації текстур на основі множини Мандельброта

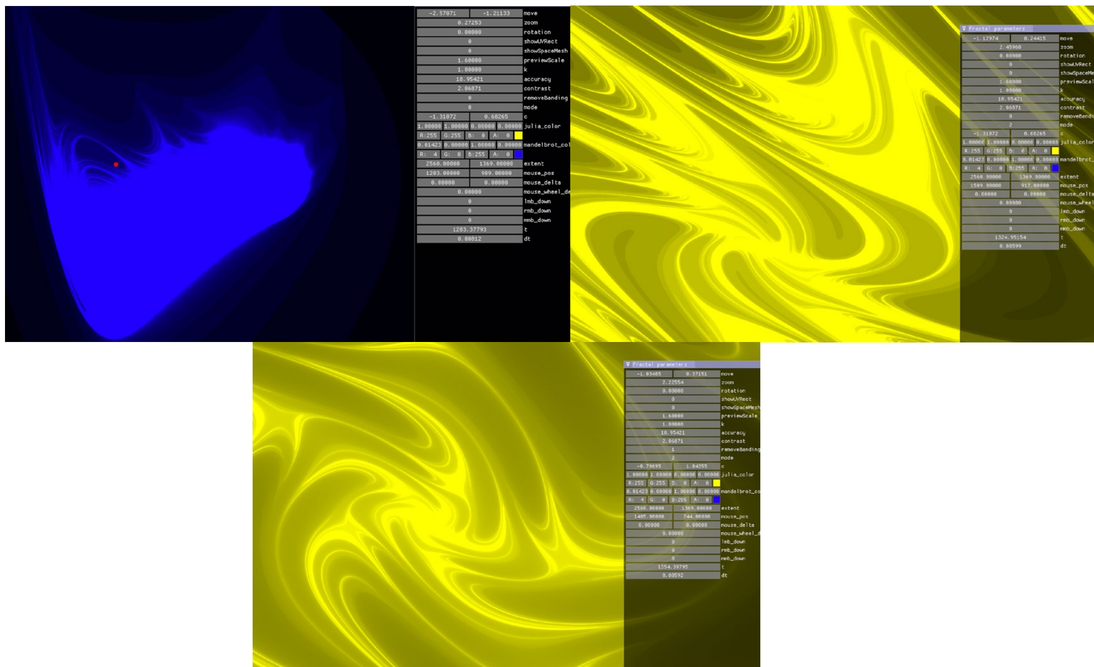

Рисунок 1 – Модифікація, що відповідає виразу:

```
z= float2(z.x * z.x -  z.y, p.k * z.x * z.y)+c;
```

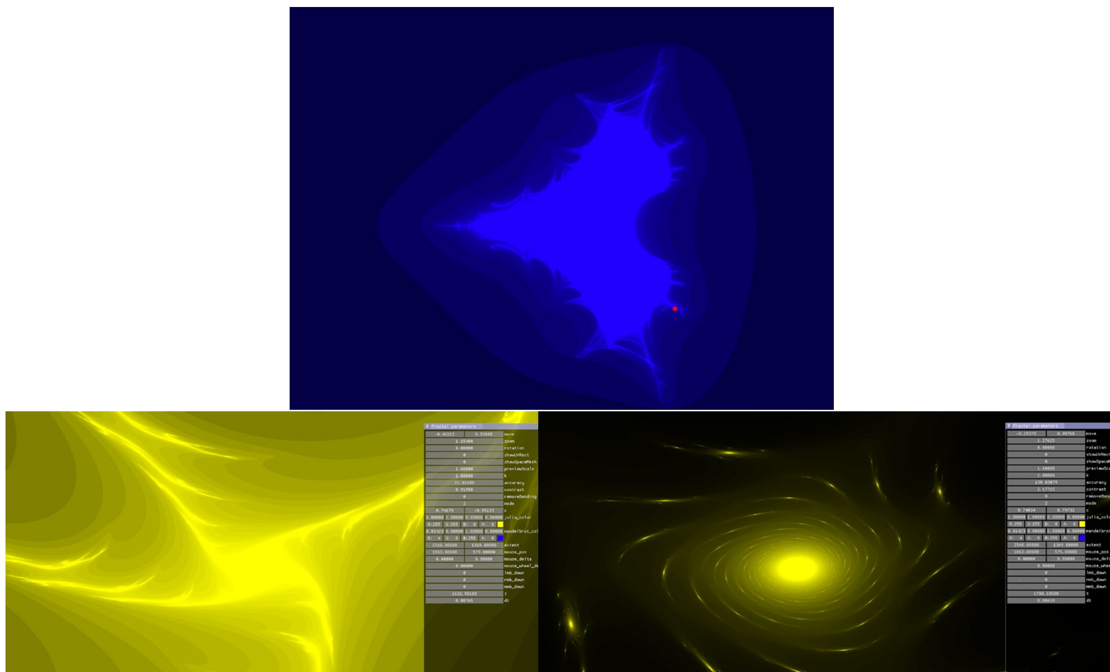

Рисунок 2 – Модифікація, що відповідає виразу:

```
z= float2(z.x * z.x -  z.y*z.y, p.k * z.x * z.y*c.x)+c;
```

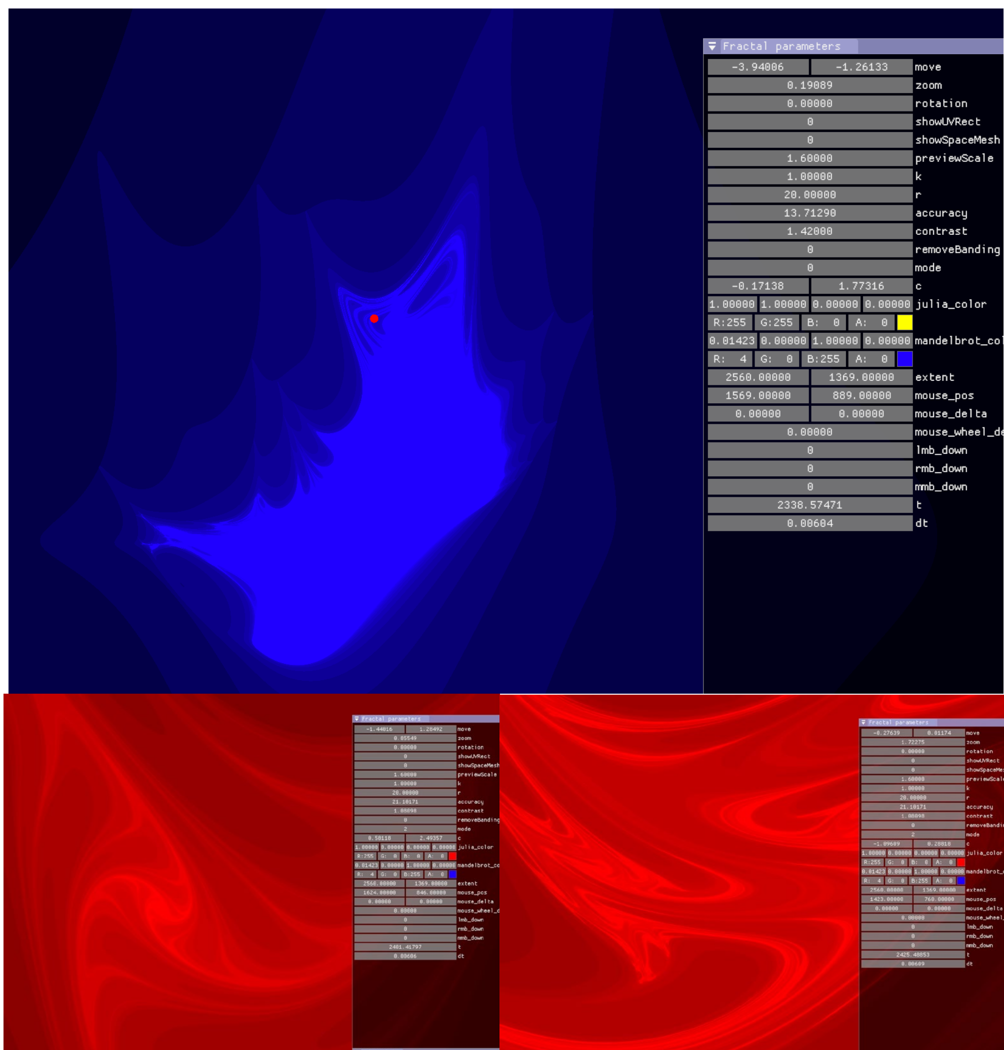

Рисунок 3 – Модифікація, що відповідає виразу:

```
z= float2(z.x * z.x -  z.y, p.k * z.x * z.y*c.x)+c;
```

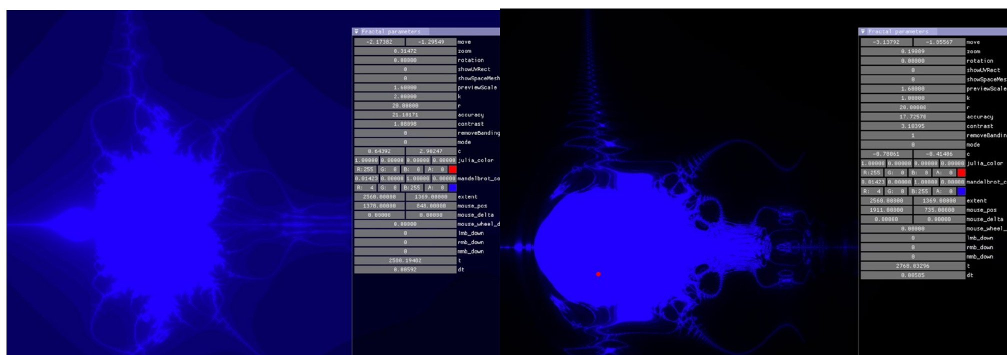

Рисунок 4 – Модифікація, що відповідає виразу:

```
z= float2(sin(z.x * z.x) - z.y*z.y, p.k * z.x * z.y)+c;
```

Рисунок 5 – Модифікація, що відповідає виразу:

```
z= float2(sin(z.x * z.x) - z.y*z.y, p.k * z.x * z.y)+c;
```