Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для формування текстур з використанням фракталів»

Виконав: студент 4 курсу
групи 1ПІ-21Б спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Магуран В.С.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ПЗ Майданюк В.П.
(прізвище та ініціали)
« 06.06 » 2025 р.

Рецензент: к.т.н., доц. каф. ОТ Обертюх М.Р.
(прізвище та ініціали)
« 06.06 » 2025 р.

Допущено до захисту
Зав. кафедри ПЗ
д.т.н., проф. О.Н. Романюк
(ініціали та прізвище)
«06» 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О.Н.
«04» березня 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Магурану Володимиру Сергійовичу

1. Тема роботи – «Розробка програмного забезпечення для формування текстур з використанням фракталів»

Керівник роботи: Майданюк Володимир Павлович, к.т.н., доц. кафедри ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. № 97.

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: тип програми – десктопний застосунок; модель розробки – інкрементна; засоби організації даних програми – бібліотеки OpenGL та PyQt; середовище розробки – PyCharm; мова програмування – Python; операційна система - Windows.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз стану розвитку застосунків для формування текстур з використанням фракталів; розробка методів та алгоритмів програмного застосунку; розробка застосунку; тестування застосунку; висновки; список використаних джерел; додатки.

5. Перелік графічного матеріалу: титульний слайд; актуальність теми; мета, об'єкт та предмет дослідження; задачі дослідження, наукова новизна, практична

Зав. кафедри О.Н. Романюк

цінність одержаних результатів; порівняльний аналіз аналогів; використані технології; розробка методу адаптивного фрактального деталізування текстур; розробка методу гармонійного змішування шарів фракталів; тестування застосунку; висновки; апробація результатів роботи і публікації.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Майданюк В.П., к.т.н., доцент кафедри ПЗ	24.03.2025	02.06.2025

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз стану розвитку застосунків для формування текстур з використанням фракталів	25.03.25 - 08.04.25	Вик
2	Розробка методів та алгоритмів програмного застосунку	09.04.25 - 20.04.25	Вик
3	Варіантний аналіз та вибір мови програмування розробки	21.04.25 - 23.04.25	Вик
4	Розробка застосунку	24.04.25 - 20.05.25	Вик
5	Тестування застосунку	21.05.25 - 25.05.25	Вик
6	Оформлення матеріалів до захисту БКР	26.05.25 - 30.05.25	Вик

Студент _____ Магуран В.С.
(підпис) (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи _____ Майданюк В.П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 66 сторінок формату А4, на яких є 43 рисунки та 2 таблиці, список використаних джерел містить 29 найменувань.

Метою бакалаврської кваліфікаційної роботи є зменшення трудомісткості проектування текстурних поверхонь та узорів тканин за рахунок використання методів фрактальної геометрії і комп'ютеризованому формуванню зображень.

У роботі було проведено аналіз стану питання розробки застосунку для формування текстур з використанням фракталів, було розглянуто існуючі застосунки-аналоги. Проведено порівняльний аналіз, визначено переваги та відповідно актуальність власної розробки. Було проведено аналіз методів розв'язання задачі, обрано метод, що поєднує класичні фрактальні алгоритми з обчисленням на графічному процесорі комп'ютера.

Проведено аналіз інформаційного забезпечення застосунку, розроблено метод адаптивного фрактального деталізування текстур та метод гармонійного змішування шарів, розроблено алгоритми роботи застосунку.

З використанням мови Python розроблено головний модуль застосунку та модуль змішування шарів фракталів.

Проведено тестування розробленого застосунку. Продемонстровано його можливості, працездатність застосунку та правильність його роботи. Було доведено, що застосунок виконує всі поставлені на початку розробки задачі.

Ключові слова: зображення, фрактал, алгебраїчний фрактал, програма.

ABSTRACT

The bachelor's qualification work consists of 66 pages of A4 format, on which there are 43 figures and 2 tables, the list of used sources contains 29 names.

The purpose of the bachelor's qualification work is to reduce the complexity of designing textured surfaces and fabric patterns by using fractal geometry methods and computerized image formation.

The work analyzed the state of the issue of developing an application for forming textures using fractals, considered existing similar applications. A comparative analysis was conducted, the advantages and, accordingly, the relevance of our own development were determined. An analysis of the methods for solving the problem was conducted, a method was selected that combines classical fractal algorithms with calculations on a computer graphics processor.

An analysis of the information support of the application was conducted, a method of adaptive fractal detailing of textures and a method of harmonious mixing of layers were developed, and algorithms for the application were developed.

Using the Python language, the main module of the application and the module for mixing fractal layers were developed.

The developed application was tested. Its capabilities, the functionality of the application and the correctness of its work were demonstrated. It was proven that the application fulfills all the tasks set at the beginning of development.

Keywords: image, fractal, algebraic fractal, program.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Більшість тканин, що використовуються в легкій промисловості характеризуються тим чи іншим рисунком (узором) та кольоровою гамою. Фактично способів нанесення рисунків на тканини два:

- використання різнокольорових ниток і нанесення рисунку на етапі виготовлення (ткання) тканини;

- використання друкованих тканин – рисунок наноситься після виготовлення тканини методом друку.

Зрозуміло, що другий спосіб менш затратний та більш технологічний. Однак, при обох підходах важливою задачею є проектування самих узорів тканин. Раніше ці задачі вирішували люди – художники, які з використанням різних графічних технік та своєї фантазії створювали рисунки які наносились на тканини.

Однією з таких технік є монотипія. Монотипія (від моно. і грец. Τυπος – відбиток) [1] – вид друкарської графіки.

Техніка монотипії полягає в нанесенні фарб від руки на ідеально гладку поверхню друкарської форми з подальшим друкуванням на верстаті; отримане на папері відтиснення завжди буває єдиним, унікальним. Художник після друку вибирає ті відбитки, які задовольняють його по естетичній привабливості і сюжету. З багатьох відбитків вибираються лише деякі. Тобто монотипія досить трудомістка і вимагає великої кількості матеріалів і немало терпіння.

Ситуація змінилася з появою фрактальної геометрії [2, 3]. У сімдесяті роки минулого століття американський математик Бенуа Мандельброт написав книгу по фрактальній геометрії (Benoit Mandelbrot, The Fractal Geometry of Nature, 1983).

Слово фрактал утворено від латинського fractus (дробовий) і означає той, що складається з фрагментів. Воно було запропоноване Бенуа Мандельбротом в 1975 році для позначення нерегулярних, але самоподобніх структур, якими він займався. Тепер завдяки досягненням фрактальної геометрії можна отримувати

такі узори і малюнки, використовуючи формалізовані математичні методи з можливістю повторення малюнків повністю автоматично як результат роботи комп'ютерної програми. Це дозволить значно зменшити трудомісткість проектування узорів для тканин і гобеленів. Тому актуальною є розробка застосунку для формування текстур з використанням фракталів.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою роботи є зменшення трудомісткості проектування текстурних поверхонь та узорів тканин за рахунок використання методів фрактальної геометрії і комп’ютеризованому формуванню зображень.

Основними задачами дослідження є:

- розробити архітектуру застосунку для формування текстур з використанням фракталів;
- розробити інтерфейс користувача застосунку;
- розробити метод гармонійного змішування шарів фракталів;
- розробити метод адаптивного фрактального деталізування текстур;
- розробити алгоритми роботи застосунку;
- розробити функціонал застосунку;
- провести тестування розробленого застосунку.

Об'єкт дослідження – процес розробки програмного застосунку для формування текстур з використанням фракталів.

Предмет дослідження – методи та засоби розробки програмного застосунку для формування текстур з використанням фракталів.

Методи дослідження. У процесі досліджень використовувались: теорія алгоритмів, методи фрактальної геметрії для формування текстурних зображень; методи людино-машинної взаємодії для розробки інтерфейсу користувача; методи деталізування текстур; метод змішування шарів фракталів; методи тестування програмних застосунків.

Новизна отриманих результатів.

1. Подальшого розвитку отримав метод гармонійного змішування шарів фракталів, який, на відміну від відомих, використовує накладання шарів, враховуючи математичні принципи гармонії та візуального сприйняття, що дозволяє динамічно адаптувати рівень деталізації фракталу залежно від його просторового розташування в сцені та важливості для кінцевого зображення.

2. Подальшого розвитку отримав метод адаптивного фрактального деталізування текстур, який, на відміну від відомих, використовує комплексний підхід до визначення рівня деталізації, що дозволяє зменшити навантаження на графічний процесор.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в бакалаврській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та розроблено програмні засоби для формування текстурних зображень, що зменшують трудомісткість проектування узорів для тканин і гобеленів.

Особистий внесок здобувача. Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі, отримані автором особисто. У друкованій праці, опублікованій у співавторстві, автору належать такі результати: граф-схема алгоритму генерації зображень, тестування за стосунку [4].

Апробація результатів роботи. Основні положення бакалаврської кваліфікаційної роботи доповідалися та обговорювалися на конференції «LIV Всеукраїнська науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (2025)».

Публікації. Основні результати досліджень опубліковано в одній статті у матеріалах конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ) (Вінниця, 2025 р.).

1 АНАЛІЗ СТАНУ РОЗВИТКУ ЗАСТОСУНКІВ ДЛЯ ФОРМУВАННЯ ТЕКСТУР З ВИКОРИСТАННЯМ ФРАКТАЛІВ

1.1 Аналіз стану питання розробки застосунків для формування текстур

Розробка застосунків для формування текстур із використанням фракталів представляє собою комбінацію класичних математичних методів і сучасних обчислювальних технологій. Фрактали, завдяки своїй властивості самоподібності та нескінченної деталізації, дозволяють створювати реалістичні та деталізовані зображення, що отримало широке застосування у комп'ютерній графіці, цифровому мистецтві та ігровій індустрії.

Сучасні алгоритми генерування фрактальних текстур базуються на класичних методах, таких як побудова множин Мандельброта і Джуліа, а також на фрактальному шумовому моделюванні з використанням алгоритмів, наприклад, шуму Перліна або фрактального Brownian motion [5]. Завдяки інтеграції обчислень на графічному процесорі, і використанню спеціалізованих шейдерів, відкривається можливість досягти високої продуктивності, що відкриває можливість для швидкої генерації текстур. Це дозволяє створювати інтерактивні застосунки, які миттєво відображають зміни параметрів.

У багатьох сучасних застосунках використовуються гібридні методи, які комбінують фрактальні алгоритми з іншими технологіями, зокрема процедурною генерацією текстур та алгоритмами машинного навчання. Такий підхід дозволяє адаптувати текстури під конкретні умови, наприклад, враховуючи освітлення чи інші фактори навколишнього середовища, що сприяє досягненню більш реалістичного результату.

Ринок програмного забезпечення для генерації фрактальних текстур включає як open-source проєкти, так і комерційні рішення, розроблені на різних мовах програмування, включаючи Python, C++ та JavaScript. Сучасні застосунки часто мають зручні графічні інтерфейси, що дозволяють користувачам змінювати параметри фракталів, а також інтегрувати їх у більші проєкти, наприклад, у відеоігри чи візуальні симуляції природних процесів.

Попри перераховані переваги, розробникам доводиться вирішувати проблеми, пов’язані з оптимізацією обчислень і забезпеченням високої продуктивності при генерації текстур високої якості.

Поєднання фрактальних методів із технологіями машинного навчання та інтеграція їх, розробка більш оптимізованих та адаптивних алгоритмів формування фрактальних текстур відкривають нові можливості, але при цьому й створюють додаткові вимоги до адаптивності і масштабованості застосунків.

1.2 Порівняльний аналіз аналогів

Для дослідження вимог до застосунків для формування текстур з використанням фракталів, та визначення функціоналу власної розробки, необхідно провести аналіз аналогів.

Наразі існують три провідних застосунки для формування текстур з використанням фракталів: Ultra Fractal, Mandelbulb 3D, Apophysis.

Ultra Fractal [6] – це потужна програма для генерації фракталів, яка дозволяє користувачам створювати складні та красиві візуальні композиції. Вона відома своєю гнучкістю та широкими можливостями для налаштувань, що робить її улюбленим інструментом серед художників і дизайнерів. Програма дозволяє працювати з численними математичними формулами, які лежать в основі генерації фракталів.

Завдяки інтуїтивному інтерфейсу та детальним налаштуванням, Ultra Fractal надає можливість тонкого регулювання кольорових палітр, рівнів рекурсії та інших параметрів. Це дозволяє користувачам експериментувати з різними комбінаціями, створюючи унікальні візуальні ефекти, що неможливо досягти звичайними методами. Така гнучкість сприяє творчому пошуку та розвитку індивідуального стилю.

Програма підтримує багатошаровість, що дозволяє комбінувати різні фрактальні елементи в одному зображенні. Кожен шар можна налаштовувати окремо, змінюючи його параметри, прозорість і ефекти змішування. Це відкриває великі можливості для створення комплексних композицій з глибокою структурою

та насиченою деталізацією.

Спільнота користувачів Ultra Fractal активно обмінюється знаннями, порадами та прикладами робіт, що допомагає новачкам швидко освоїти інструмент. Регулярні оновлення та підтримка розробників гарантують, що програма залишається актуальною та відповідає сучасним вимогам цифрового мистецтва.

Інтерфейс Ultra Fractal наведено на рисунку 1.1.

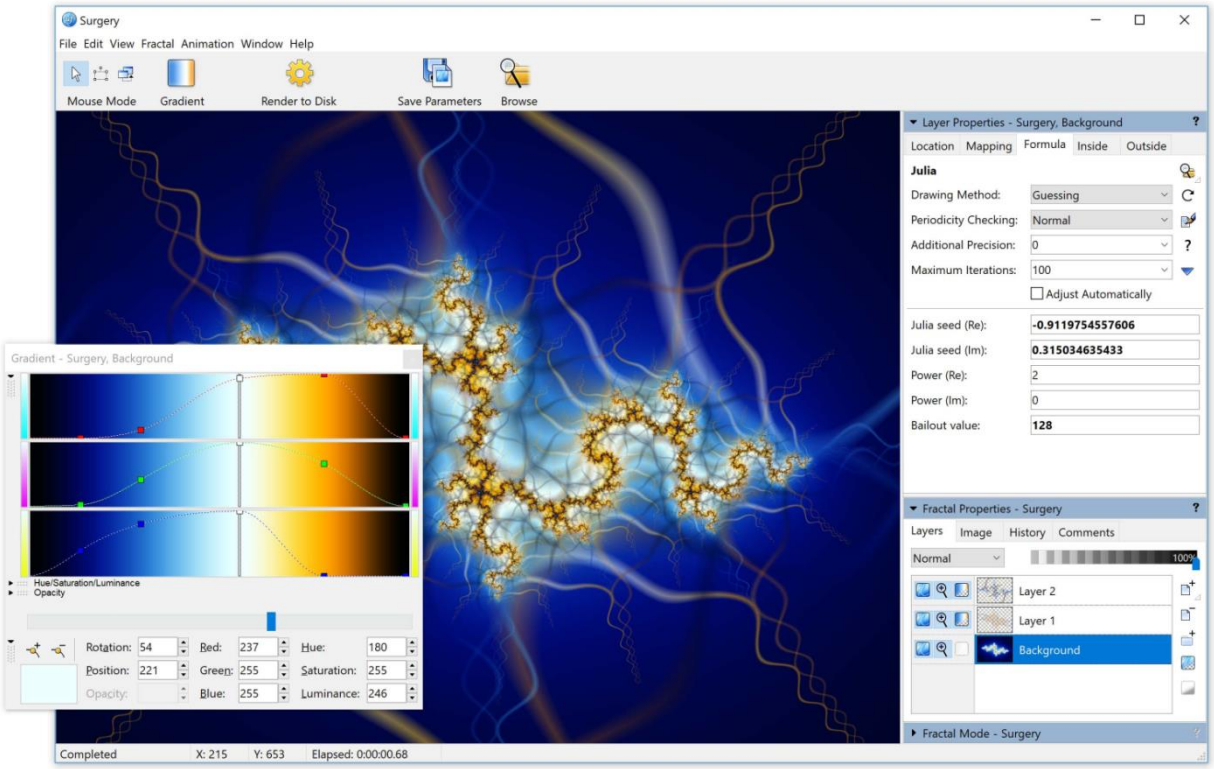


Рисунок 1.1 – Інтерфейс Ultra Fractal

Mandelbulb 3D [7] – це спеціалізований інструмент для генерації тривимірних фракталів, який дозволяє користувачам досліджувати і створювати неймовірно детальні об’ємні структури. Програма відкриває можливості для візуалізації 3D-фрактальних форм, що раніше були недоступні через обмеження класичних двовимірних підходів.

Інтерфейс Mandelbulb 3D забезпечує доступ до широкого спектру параметрів, що дозволяють детально налаштовувати геометрію, освітлення та

кольорові схеми тривимірних об’єктів. Завдяки цьому користувачі можуть отримати контроль над процесом генерації фракталів та досягти бажаного художнього ефекту, експериментуючи з різними стилями та техніками візуалізації.

Технічна сторона Mandelbulb 3D включає можливості для глибокої оптимізації та рендерингу, що дозволяє створювати зображення високої якості. Завдяки використанню потужних алгоритмів та підтримці багатопоточності, програма здатна ефективно працювати навіть з дуже складними тривимірними моделями, забезпечуючи плавну роботу та швидке відображення змін.

Програма доступна безкоштовно, що робить її популярною серед широкого кола користувачів – від ентузіастів цифрового мистецтва до професійних дизайнерів. Активна спільнота користувачів регулярно ділиться своїми роботами та налаштуваннями, що сприяє постійному розвитку інструменту та розширенню його функціональних можливостей.

Інтерфейс застосунку Mandelbulb 3D наведено на рисунку 1.2.

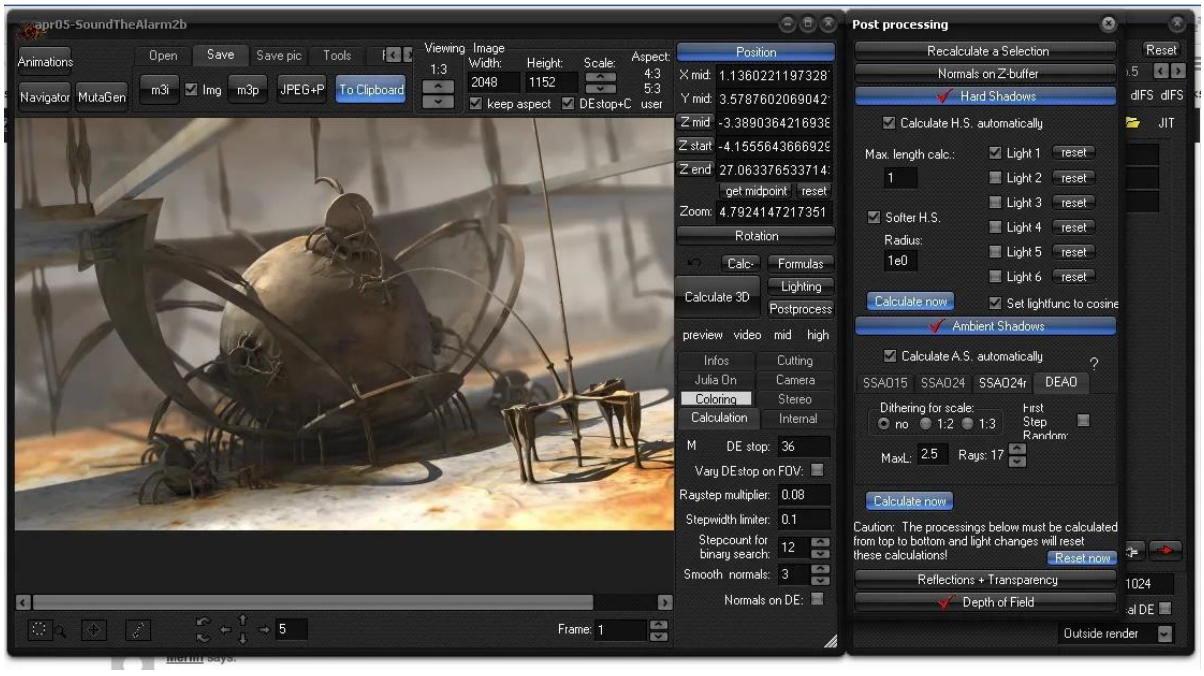


Рисунок 1.2 – Інтерфейс Mandelbulb 3D

Apophysis [8] – це програмне забезпечення, яке спеціалізується на створенні так званих фрактальних вогнів (fractal flames), пропонуючи користувачам можливість генерувати яскраві та експресивні фрактальні зображення. Цей

інструмент особливо популярний серед художників, які шукають нові форми для вираження своєї креативності через абстрактні візуальні ефекти.

Програма надає можливість налаштовувати численні параметри генерації, включаючи варіації в кольорах, геометрії та переходах між фрактальними формами. Інтуїтивний інтерфейс дозволяє навіть новачкам легко зрозуміти принципи роботи з фрактальними алгоритмами та експериментувати з різними налаштуваннями для досягнення бажаних результатів.

Інтерфейс Apophysis наведено на рисунку 1.3.

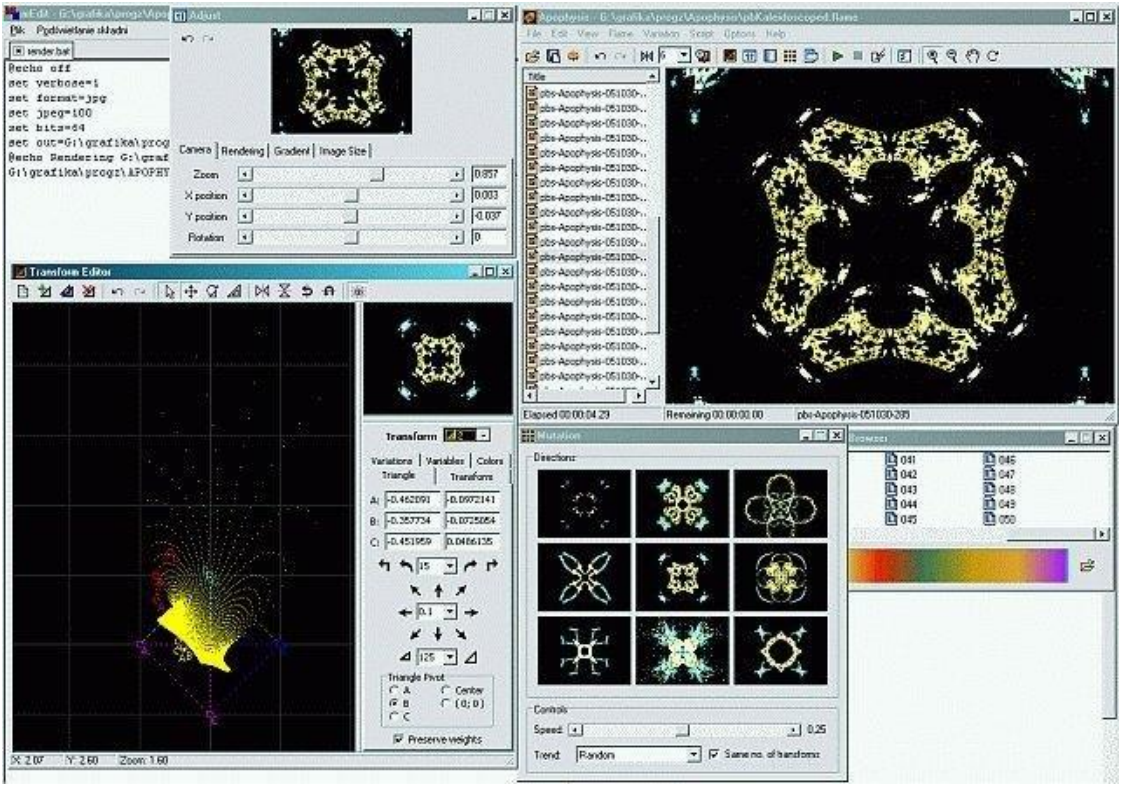


Рисунок 1.3 – Інтерфейс Apophysis

Apophysis використовує алгоритми, які створюють складні та органічні форми, що нагадують природні процеси та явища. Завдяки цьому зображення отримують динамічний вигляд і здатні передавати відчуття руху та енергії. Такий підхід дозволяє створювати не тільки статичні композиції, але й анімовані фрактальні візуалізації.

Для порівняння можливостей та функціоналу аналогів між собою, та в

власною розробкою, було розроблено таблицю 1.1.

Таблиця 1.1 – Порівняльний аналіз аналогів

Характеристика	Ultra Fractal	Mandelbulb 3D	Apophysis	Власна розробка
Підтримка 2D фракталів	+	+	+	+
Гнучкість налаштувань	+	+	+	+
Інтерактивність	+	+	+	+
Багатошаровість	+	+	+	+
Адаптивна деталізація текстури	-	-	-	+
Гармонійне змішування шарів	-	-	-	+
Загальний бал	4	4	4	6

Таким чином, власна розробка має вищий загальний бал порівняно з аналогами на 33,3% (100-4/6*100% = 33,3%). Тобто, власна розробка випереджає аналоги за можливостями та функціоналом, отже є актуальною.

1.3 Аналіз методів розв’язання задачі

Класичні фрактальні алгоритми, такі як множина Мандельброта та множина Джулії, становлять базову математичну основу для генерації фрактальних зображень [9]. Ці алгоритми дозволяють отримати високодеталізовані та повторювані патерни завдяки точному контролю параметрів формули. Перевагою даного підходу є можливість глибокої математичної моделювання, що дозволяє розробникам точно керувати формою, кольоровою палітрою та іншими характеристиками фракталів. Однак недоліком може бути висока обчислювальна складність, особливо при збільшенні рівня деталізації.

Другий підхід базується на використанні процедурної генерації текстур із

застосуванням алгоритмів фрактального Brownian motion або шуму Перліна [10]. Ці методи додають природний вигляд зображенням за рахунок інтеграції випадковості та органічних елементів. Процедурна генерація дозволяє створювати варіативні та унікальні текстури, що добре імітують природні поверхні, такі як кам’яні, водні чи рослинні текстури. Перевагою цього методу є здатність до створення великої кількості різних варіантів без необхідності ручного налаштування кожного параметра, але разом з тим контроль над кінцевим результатом може бути менш точним, ніж у чисто математичних підходах.

Сучасні технології активно використовують обчислення на графічних процесорах (GPU) для забезпечення інтерактивності та високої продуктивності застосунків [11]. Використання GPU дозволяє реалізувати паралельну обробку фрактальних алгоритмів, що значно скорочує час рендерингу складних зображень і дозволяє в режимі реального часу змінювати параметри текстур. Інтерактивний режим сприяє експериментуванню з різними ефектами та швидкому отриманню візуального зворотного зв’язку. Однак оптимізація під конкретне обладнання може вимагати додаткових зусиль з боку розробників для забезпечення сумісності та стабільної роботи застосунку на різних пристроях.

Нещодавнім трендом є застосування методів машинного навчання та нейронних мереж для вдосконалення генерації фрактальних текстур. Такі підходи дозволяють аналізувати величезні обсяги даних природних текстур і використовувати отримані закономірності для генерації нових, більш реалістичних зображень. Поєднання класичних фрактальних алгоритмів з можливостями штучного інтелекту відкриває шлях до автоматичної оптимізації параметрів і адаптації текстур під конкретні візуальні вимоги. Проте інтеграція машинного навчання часто вимагає значних ресурсів і досвіду в області AI, що може стати додатковою складністю при розробці застосунку.

Отже, розробка застосунку для формування текстур із використанням фракталів є задачею, що включає в себе класичні математичні методи, процедурні підходи, оптимізацію за допомогою сучасних обчислювальних технологій та інноваційні методи машинного навчання. Кожен із цих підходів має свої переваги

та недоліки, а вибір оптимальної стратегії залежить від конкретних цілей, вимог до продуктивності та бажаного рівня контролю над кінцевим результатом.

Оптимальним вибором буде гібридний метод, що поєднує класичні фрактальні алгоритми з обчисленнями на графічному процесорі комп'ютера. Цей підхід забезпечує математичну точність генерації фракталів та високу продуктивність застосунку.

Для реалізації двовимірних фракталів найкраще використовувати класичні алгоритми множин Мандельброта та Джуліа, тому що вони забезпечують точний контроль над параметрами і дозволяють отримати високодеталізовані зображення фракталів. Для тривимірних фракталів найкращим є алгоритм Mandelbulb з модифікаціями для покращення продуктивності.

Всі обчислення необхідно проводити за допомогою графічного процесора комп'ютера через шейдери GLSL, тому що таким чином користувачі матимуть можливість одразу змінювати параметри фракталів та швидко отримувати результат. Також це дозволяє вирішити проблему високої обчислювальної складності, яка є основним недоліком класичних алгоритмів для роботи з фракталами.

Для забезпечення природності текстур обрано алгоритми процедурної генерації, зокрема фрактальний Brownian motion, що дозволять додавати до текстур елементи випадковості та органічності; й імітувати природні поверхні.

1.4 Постановка задач дослідження

Провівши аналіз методів розв'язання поставленої задачі, аналіз стану застосунків для формування текстур з використанням фракталів, порівняння аналогів було поставлено такі задачі дослідження:

- розробити архітектуру застосунку для формування текстур з використанням фракталів;
- розробити інтерфейс користувача застосунку;
- розробити метод гармонійного змішування шарів фракталів;
- розробити метод адаптивного фрактального деталізування текстур;

- розробити алгоритми роботи застосунку;
- розробити функціонал застосунку;
- провести тестування розробленого застосунку.

Технічне завдання на розробку наведено в додатку А.

1.5 Висновки

Метою першого розділу було обґрунтування доцільності розробки програмного забезпечення для формування текстур з використанням фракталів та формулювання основних вимог до програми.

У процесі аналізу стану питання розробки застосунків для формування текстур було встановлено, що головним завданням, яке постає перед розробниками таких застосунків, є оптимізація обчислень і забезпечення високої продуктивності при генерації текстур високої якості.

Порівняльний аналіз аналогів, зокрема Ultra Fractal, Mandelbulb 3D та Apophysis, дозволив виявити наявні функціональні обмеження існуючих рішень та сформувати перелік необхідних можливостей, які доцільно реалізувати у власному продукті.

Було проаналізовано методи розв’язання задачі з розробки власної програми для формування текстур з використанням фракталів, обрано алгоритми процедурної генерації, що дозволять додавати до текстур елементи випадковості та органічності й імітувати природні поверхні.

На основі проведеного аналізу було сформульовано задачі, що включають розробку архітектури застосунку для формування текстур з використанням фракталів, розробку інтерфейсу користувача застосунку, розробку методу гармонійного змішування шарів фракталів, розробку методу адаптивного фрактального деталізування текстур, розробку алгоритмів роботи застосунку, розробку функціоналу застосунку.

Таким чином, результати першого розділу підтверджують доцільність створення власного застосунку для формування текстур з використанням фракталів

2 РОЗРОБКА МЕТОДІВ ТА АЛГОРИТМІВ ПРОГРАМНОГО ЗАСТОСУНКУ

2.1 Аналіз інформаційного забезпечення

Інформаційна архітектура застосунку для формування фрактальних текстур організована навколо трьох основних потоків даних. Вхідні дані включають параметри фракталів, які користувач налаштовує через інтерфейс, імпортовані зображення та текстури для базових шарів, конфігурації методів адаптивного фрактального деталізування текстур і гармонійного змішування шарів фракталів, а також параметри експорту, що визначають формат і якість результату [12]. Проміжні дані представлені картами важливості для методу адаптивного фрактального деталізування текстур, результатами спектрального аналізу шарів для гармонійного змішування шарів фракталів, попередніми версіями текстур для швидкого перегляду та гармонійними матрицями для кожної пари шарів. Вихідні дані формуються як фінальні текстури з оптимізованою деталізацією та гармонійним змішуванням, метадані з параметрами генерації, експортовані файли в різних форматах та збережені налаштування проєктів. Така структура потоків даних забезпечує ефективну обробку інформації на всіх етапах генерації текстур.

Для ефективної роботи з фрактальними текстурами застосунок використовує різноманітні формати даних. Внутрішні формати включають 32-бітний RGBA з плаваючою точкою для високоточного представлення кольорів текстур, одноканальний 32-бітний формат з плаваючою точкою для карт важливості та масиви 32-бітних значень для спектральних даних. Зовнішні формати охоплюють стандартні графічні формати для імпорту та експорту текстур (PNG, JPG, BMP, TGA), з додатковою підтримкою EXR для HDR-текстур, що забезпечує збереження розширеного динамічного діапазону. Для зберігання налаштувань застосовуються структуровані формати JSON або XML, що дозволяють зберігати ієрархічні дані з параметрами фракталів, шарів та методів обробки. Окремо використовується JSON-формат для збереження та обміну пресетами — наборами параметрів для створення специфічних ефектів, що

спрощує повторне використання успішних налаштувань.

Система локального зберігання даних у застосунку реалізована через кілька спеціалізованих компонентів:

1. База пресетів зберігає попередньо налаштовані параметри для різних типів фракталів та ефектів, що дозволяє користувачам швидко застосовувати перевірені конфігурації.

2. Система кешування текстур зберігає проміжні результати обчислень для підвищення продуктивності при повторному застосуванні схожих параметрів або при роботі з багатошаровими композиціями.

3. Бібліотека кольорових палітр містить попередньо налаштовані гармонійні комбінації кольорів, оптимізовані для фрактальних текстур різних типів.

Управління ресурсами здійснюється через спеціалізовану систему оптимізації використання відеопам'яті, яка динамічно розподіляє ресурси графічного процесора між різними процесами генерації текстур, та менеджер буферів, що забезпечує ефективне перевикористання тимчасових обчислювальних буферів, мінімізуючи операції виділення та звільнення пам'яті.

Для досягнення високої продуктивності при роботі з великими текстурами застосунок використовує комплексні стратегії оптимізації даних:

1. Адаптивне стиснення текстур, яке аналізує вміст зображення і застосовує різні алгоритми стиснення залежно від характеристик контенту, що дозволяє знизити вимоги до пам'яті без помітної втрати якості.

2. Ієрархічне представлення даних через mip-mapping, яке забезпечує швидкий доступ до текстур на різних рівнях деталізації, що особливо важливо для методу адаптивного фрактального деталізування текстур.

3. Розділення каналів, яке дозволяє окремо обробляти різні компоненти зображення, наприклад, яскравість та колір, що оптимізує обчислювальні процеси.

Для прискорення доступу до даних використовуються просторові структури quadtree або octree, попередні обчислення важливих метрик та локально-адаптивні кеші для частин текстур, що найчастіше використовуються.

Такий багаторівневий підхід до оптимізації дозволяє суттєво підвищити

продуктивність застосунку навіть при обробці надзвичайно детальних фрактальних текстур.

Для підтримки цілісності даних та захисту результатів роботи користувача застосунок впроваджує комплексні механізми контролю:

1. Система версій параметрів проєкту забезпечує можливість відстеження змін та повернення до попередніх станів при необхідності.

2. Історія змін зберігає всю послідовність модифікацій з детальними метаданими, включаючи час, автора та опис змін.

3. Функція автоматичного збереження регулярно створює резервні копії поточного стану проєкту, а система відновлення після збоїв здатна зберегти результати навіть тривалих обчислень у випадку непередбачених проблем.

4. Інкрементальне резервне копіювання мінімізує обсяг даних, що зберігаються при кожній ітерації, зберігаючи лише змінені елементи.

Описані механізми забезпечують надійний захист даних без ускладнення робочого процесу.

Інформаційне забезпечення розробленого застосунку для формування текстур з використанням фракталів представляє собою збалансовану систему, що поєднує високу продуктивність, гнучкість та надійність. Ефективне представлення та зберігання даних досягається через оптимізовані внутрішні формати з плаваючою точкою, що забезпечують високу точність при збереженні розумних вимог до пам'яті. Гнучкі структури даних підтримують складні алгоритми фрактального аналізу та синтезу, дозволяючи реалізувати інноваційні методи адаптивного фрактального деталізування текстур та гармонійного змішування шарів фракталів з максимальною ефективністю.

2.2 Розробка моделі застосунку

Модель системи характеризується чітким розподілом функціональності між логічними блоками, що взаємодіють через визначені залежності.

Центральним елементом архітектури виступає компонент "Генератор фрактальних текстур", який виконує роль керуючого модуля системи. Даний

компонент реалізує основну бізнес-логіку застосунку та має двосторонню залежність із компонентом "Головний інтерфейс". Така взаємодія забезпечує передачу команд користувача до генератора та відображення результатів його роботи в інтерфейсі. Зв'язок типу "depends" вказує на те, що "Головний інтерфейс" використовує функціональні можливості "Генератора фрактальних текстур" для виконання операцій зі створення фрактальних зображень. "Генератор фрактальних текстур" має залежність по відношенню до "Сервісу обчислень GPU", що свідчить про передачу обчислювальних завдань графічному процесору для прискорення генерації фракталів. Дана взаємодія реалізується шляхом формування генератором відповідних завдань та їх передачі до сервісу обчислень, який, у свою чергу, використовує потужності графічного процесора для паралельних обчислень.

"Сервіс рендерингу OpenGL" займає ключову позицію у структурі візуалізації результатів, маючи залежності з кількома компонентами. Цей сервіс отримує оброблені дані від "Сервісу обчислень GPU" для їх подальшої візуалізації. Водночас компонент "OpenGL віджет" залежить від "Сервісу рендерингу OpenGL", використовуючи його можливості для відображення згенерованих фрактальних текстур у графічному інтерфейсі. Дана взаємодія забезпечує ефективне виведення графічної інформації з використанням апаратного прискорення.

Компонент "Експорт текстур" характеризується залежністю від "Сервісу рендерингу OpenGL", що вказує на використання можливостей рендерингу для отримання остаточного зображення та його збереження у зовнішніх форматах. Процес експорту полягає в отриманні візуалізованого зображення від сервісу рендерингу, його перетворенні у відповідний формат та збереженні у файловій системі.

"Адаптивний деталізатор" є спеціалізованим компонентом, який має залежність від "Панелі налаштувань". Даний взаємозв'язок відображає механізм конфігурації параметрів адаптивної деталізації через користувацький інтерфейс. "Панель налаштувань" передає параметри деталізації до "Адаптивного

деталізатора", який, у свою чергу, застосовує ці параметри для оптимізації процесу генерації фрактальних текстур з врахуванням значущості різних областей зображення.

Модель системи наведено на рисунку 2.1.

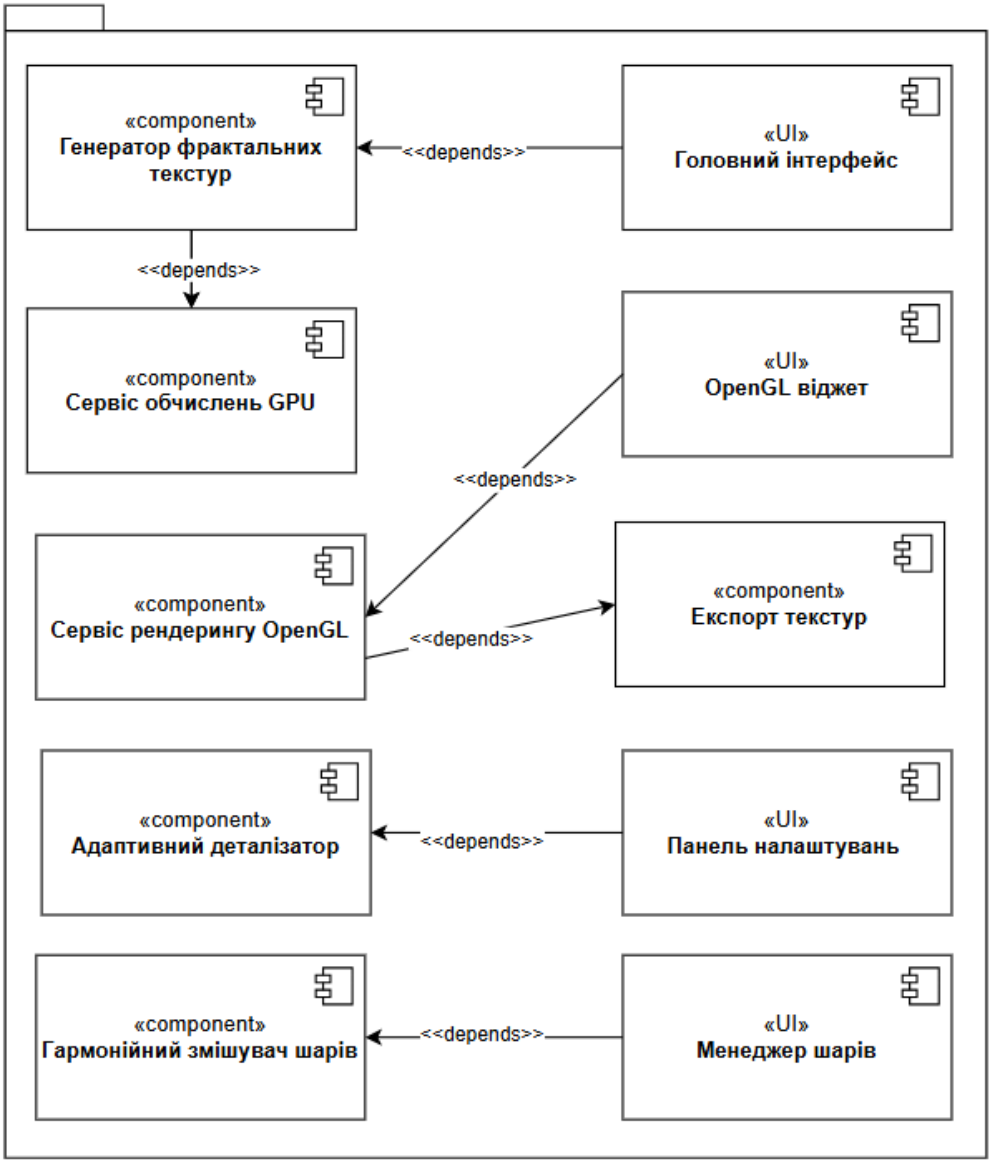


Рисунок 2.1 – Модель системи

"Гармонійний змішувач шарів" взаємодіє з компонентом "Менеджер шарів" за принципом залежності. "Менеджер шарів", будучи інтерфейсним компонентом, надає користувачеві можливість керувати шарами фрактальних текстур, а "Гармонійний змішувач шарів" використовує цю інформацію для об'єднання шарів

з урахуванням їх гармонічних характеристик. Процес змішування включає аналіз спектральних характеристик кожного шару та їх комбінування відповідно до параметрів, визначених через "Менеджер шарів".

Загальна структура діаграми демонструє чітке розмежування між компонентами користувацького інтерфейсу (позначені як "UI") та функціональними компонентами системи (позначені як "component"). Така організація забезпечує високу модульність системи, дозволяючи модифікувати окремі компоненти без суттєвого впливу на інші частини застосунку. Система залежностей між компонентами формує логічний потік даних від користувацького вводу через обчислювальні сервіси до візуалізації та експорту результатів, забезпечуючи цілісність функціонування застосунку для генерації фрактальних текстур.

2.3 Розробка архітектури застосунку

Застосунок для формування текстур з використанням фракталів являє собою інструмент для створення й відображення фрактальних текстур із застосуванням різноманітних алгоритмів та методів оптимізації.

Архітектура застосунку реалізована за патерном MVC (Model-View-Controller) та містить в своїй основі класи Texture, FractalParams, FractalLayer як моделями, GLWidget для OpenGL-рендерингу як виглядом і головним класом FractalTextureGenerator як контролером. Патерн Worker-Thread реалізований через класи FractalComputeThread і TextureGeneratorWorker, які виконують важкі обчислення в окремих потоках і спілкуються через сигнали та слоти. A Strategy патерн дозволяє легко перемикатися між різними алгоритмами генерації фракталів і вибирати між центральним та графічним процесорами реалізаціями залежно від ситуації.

Для управління текстурними даними відповідає клас Texture, який дозволяє встановлювати пікселі, розраховувати контраст і коригувати насиченість зображення.

Діаграма класу Texture наведена на рисунку 2.2.

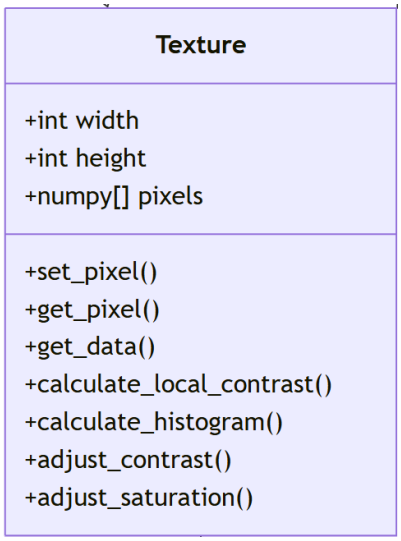


Рисунок 2.2 – Діаграма класу Texture

Поля класу Texture:

- width (int) – ширина текстури в пікселях;
- height (int) – висота текстури в пікселях;
- pixels (numpy.ndarray) – масив пікселів зображення формату RGBA у вигляді масиву NumPy з розмірами (height, width, 4).

Функції:

- set_pixel(x, y, color) – встановлює колір для пікселя з координатами (x, y);
- get_pixel(x, y) – повертає колір пікселя з координатами (x, y);
- get_data()– повертає всі дані текстури як масив NumPy;
- calculate_local_contrast(x, y, radius=3) – обчислює локальний контраст навколо точки (x, y) з заданим радіусом;
- calculate_histogram() – обчислює гістограму яскравості текстури;
- adjust_contrast(min_value, max_value) – регулює контраст зображення на основі заданих мінімального та максимального значень;
- adjust_saturation(factor) – регулює насиченість кольорів зображення з заданим фактором;
- calculate_average_saturation() – обчислює середню насиченість для всього зображення.

Клас Color відповідає за роботу з кольорами у форматі RGBA з підтримкою всіх необхідних операцій.

Клас Color наведено на рисунку 2.3.

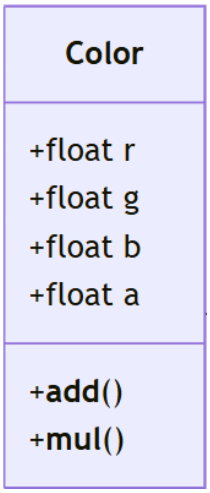


Рисунок 2.3 – Клас Color

Поля класу Color:

- r (float) – червоний компонент кольору (діапазон 0.0-1.0);
- g (float) – зелений компонент кольору (діапазон 0.0-1.0);
- b (float) – синій компонент кольору (діапазон 0.0-1.0);
- a (float) – альфа-канал (прозорість, діапазон 0.0-1.0).

Функції:

- __add__(other) – перевантажений оператор додавання для кольорів, дозволяє додавати два кольори;
- __mul__(scalar) – перевантажений оператор множення, дозволяє множити колір на скаляр.

Клас FractalParams містить усі параметри генерації фракталів – тип, центр, масштаб, кількість ітерацій – зберігаються у класі FractalParams.

Поля класу FractalParams:

- type (int) – тип фракталу (0: Мандельброт, 1: Джулія, 2: Mandelbulb 3D);
- center (tuple) – центр області перегляду фракталу на комплексній площині, представлений як (x, y);

- zoom (float) – рівень масштабування фракталу;
- baseIterations (int) – базова кількість ітерацій для обчислення фракталу;
- juliaConstant (complex) – комплексна константа для множини Джулії;
- color1 (Color) – перший колір для градієнтного зафарбовування фракталу;
- color2 (Color) – другий колір для градієнтного зафарбовування фракталу.

Клас FractalParams наведено на рисунку 2.4.

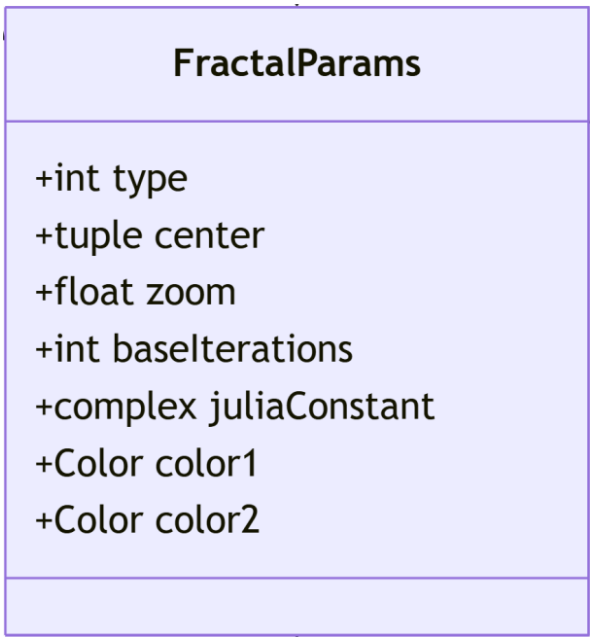


Рисунок 2.4 – Клас FractalParams

Кожен шар фрактальної текстури представлений класом FractalLayer з власними параметрами та візуальною значимістю. А для збереження контексту сцени, включаючи точку фокусу та радіус видимості, ми використовуємо клас Scene.

Поля класу FractalLayer:

- texture (Texture) – об'єкт текстури, що містить дані зображення шару;
- params (FractalParams) – параметри фракталу, використані для генерації шару;
- visualImportance (float) – коефіцієнт важливості шару при змішуванні (діапазон 0.0-1.0).

Функції:

- get_width()– повертає ширину текстури шару;
- get_height()– повертає висоту текстури шару;
- get_color_at(x, y) – повертає колір у заданій точці (x, y);
- get_visual_importance()– повертає значення візуальної важливості шару;

get_average_contrast()–обчислює середній контраст для шару.

Клас FractalLayer наведено на рисунку 2.5.

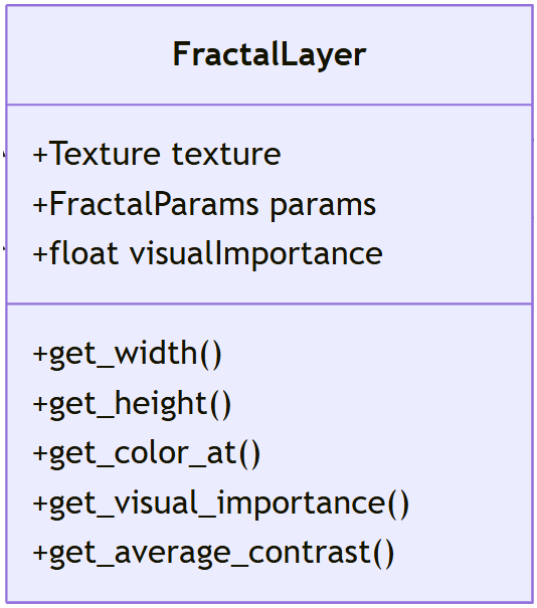


Рисунок 2.5 – Клас FractalLayer

Одна з головних особливостей системи – механізм адаптивної деталізації, втілений у класі AdaptiveDetailingEngine. Цей розумний механізм збільшує кількість ітерацій у "цікавих" зонах фракталу, що дає більш детальне відображення важливих частин зображення. Система визначає "цікавість" області на основі трьох факторів: наскільки добре її видно, як далеко вона від точки фокусу та який там локальний контраст. Користувач може налаштовувати вагові коефіцієнти для кожного з цих факторів через інтерфейс – це дає високу гнучкість у роботі. Клас AdaptiveDetailingEngine наведено на рисунку 2.6.

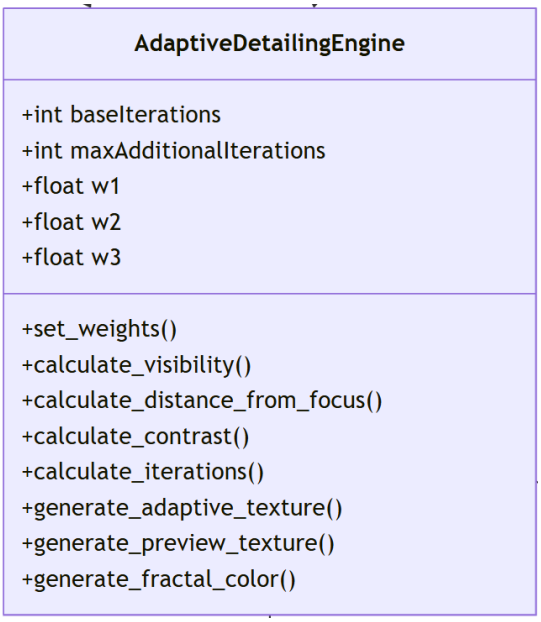


Рисунок 2.6 – Клас AdaptiveDetailingEngine

Поля класу AdaptiveDetailingEngine:

- baseIterations (int) – базова кількість ітерацій для всіх пікселів;
- maxAdditionalIterations (int) – максимальна додаткова кількість ітерацій для важливих областей;
- w1 (float) – вага фактору видимості (weight for visibility);
- w2 (float) – вага фактору відстані (weight for distance);
- w3 (float) – вага фактору контрасту (weight for contrast).

Функції:

- set_weights(w1, w2, w3) – встановлює вагові коефіцієнти для факторів адаптивності;
- calculate_visibility(x, y, scene) – обчислює фактор видимості для точки (x, y);
- calculate_distance_from_focus(x, y, scene) – обчислює фактор відстані від точки фокусу;
- calculate_contrast(x, y, current_texture) – обчислює фактор локального контрасту;
- calculate_iterations(x, y, scene, current_texture) – обчислює оптимальну кількість ітерацій для точки;

•generate_adaptive_texture(width, height, params, scene) – генерує текстуру з адаптивною деталізацією;

•generate_preview_texture(width, height, params) – генерує спрощену версію текстури для попереднього аналізу;

•generate_fractal_color(x, y, params, iterations) – генерує колір фракталу для заданої точки з вказаною кількістю ітерацій.

Для створення складних багатошарових композицій було розроблено HarmonicLayerBlendingEngine, який змішує кілька фрактальних шарів з урахуванням їхніх гармонічних частот і візуальної значущості. Процес включає аналіз спектру кожного шару, розрахунок матриць гармонійного перетворення, змішування з урахуванням "важливості" шарів та балансування контрасту й насиченості кінцевої текстури. Завдяки цьому підходу можна створювати композиції з різних типів фракталів. Клас HarmonicLayerBlendingEngine наведено на рисунку 2.7.

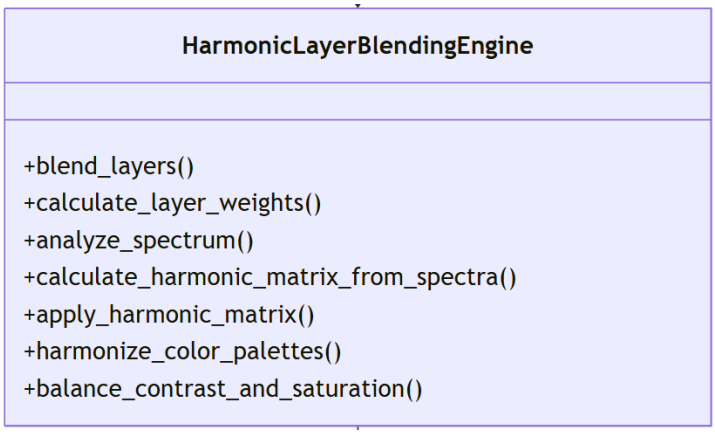


Рисунок 2.7 – Клас HarmonicLayerBlendingEngine

Функції класу HarmonicLayerBlendingEngine:

•blend_layers(layers) – змішує кілька шарів фракталів з урахуванням їх гармонічних властивостей;

•calculate_layer_weights(layers) – обчислює вагові коефіцієнти для шарів на основі їх важливості;

•analyze_spectrum(layer) – аналізує спектр текстури шару для визначення

домінантних частот;

•calculate_harmonic_matrix_from_spectra(spec1, spec2) – обчислює матрицю гармонічного перетворення на основі спектрів;

•apply_harmonic_matrix(color_vector, matrix) – застосовує матрицю гармонічного перетворення до вектора кольору;

•harmonize_color_palettes(layers) – гармонізує кольорові палітри різних шарів;

•balance_contrast_and_saturation(result) – балансує контраст і насиченість результуючої текстури після змішування.

Щоб забезпечити високу швидкість роботи, передбачено використання обчислювальних потужностей як центрального, так і графічного процесора. Для GPU-обчислень (тобто з використанням графічного процесора) використовується OpenGL compute shaders, які забезпечують високу швидкість генерації текстур. Система сама визначає, чи доступне GPU-прискорення, і автоматично використовує його за можливості.

Клас FractalComputeThread наведено на рисунку 2.8.

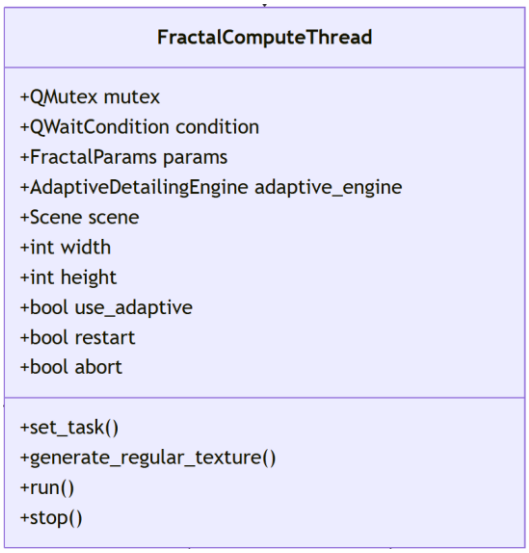


Рисунок 2.8 – Клас FractalComputeThread

Поля FractalComputeThread:

•mutex (QMutex) – м'ютекс для синхронізації доступу до спільних ресурсів;

- condition (QWaitCondition) – умова очікування для керування потоком;
- params (FractalParams) – параметри фракталу для обчислення;
- adaptive_engine (AdaptiveDetailingEngine) – посилання на двигун адаптивної деталізації;
- scene (Scene) – інформація про сцену;
- width (int) – ширина текстури, що генерується;
- height (int) – висота текстури, що генерується;
- use_adaptive (bool) – прапорець використання адаптивної деталізації;
- restart (bool) – прапорець для перезапуску обчислень;
- abort (bool) – прапорець для переривання обчислень.

Функції:

- set_task(adaptive_engine, params, scene, use_adaptive, width, height) – встановлює параметри задачі для обчислення;
- generate_regular_texture(width, height, params) – генерує звичайну (не адаптивну) текстуру;
- run() – метод, що виконується в окремому потоці, який здійснює обчислення текстури;
- stop()– зупиняє роботу потоку.

Графічний інтерфейс містить головне вікно з розділеним на зони екраном – OpenGL-віджет для відображення результатів і панель керування з усіма налаштуваннями. На панелі розміщені групи налаштувань для різних типів фракталів, елементи керування для адаптивної деталізації, список шарів і параметри гармонійного змішування.

Клас GLWidget, який реалізує OpenGL-віджет наведено на рисунку 2.9.

Поля класу GLWidget:

- current_texture (Texture) – поточна текстура для відображення;
- texture_id (GLuint) – ідентифікатор текстури в OpenGL;
- vao (GLuint) – ідентифікатор Vertex Array Object;
- vbo (GLuint) – ідентифікатор Vertex Buffer Object;

- ebo (GLuint) – ідентифікатор Element Buffer Object;
- shader_program (GLuint) – ідентифікатор програми шейдерів.

Методи:

- initializeGL()– ініціалізує OpenGL контекст;
- create_shader_program()– створює та компілює програму шейдерів;
- create_geometry()– створює геометрію для відображення текстури;
- paintGL()– виконує рендеринг сцени;
- resizeGL(w, h) – обробляє зміну розміру віджета;
- set_texture(texture) – встановлює нову текстуру для відображення;
- grab_framebuffer() – захоплює вміст буфера кадру для експорту зображення.

Рисунок 2.9 – Клас GLWidget

Архітектура застосунку наведена на рисунку 2.10

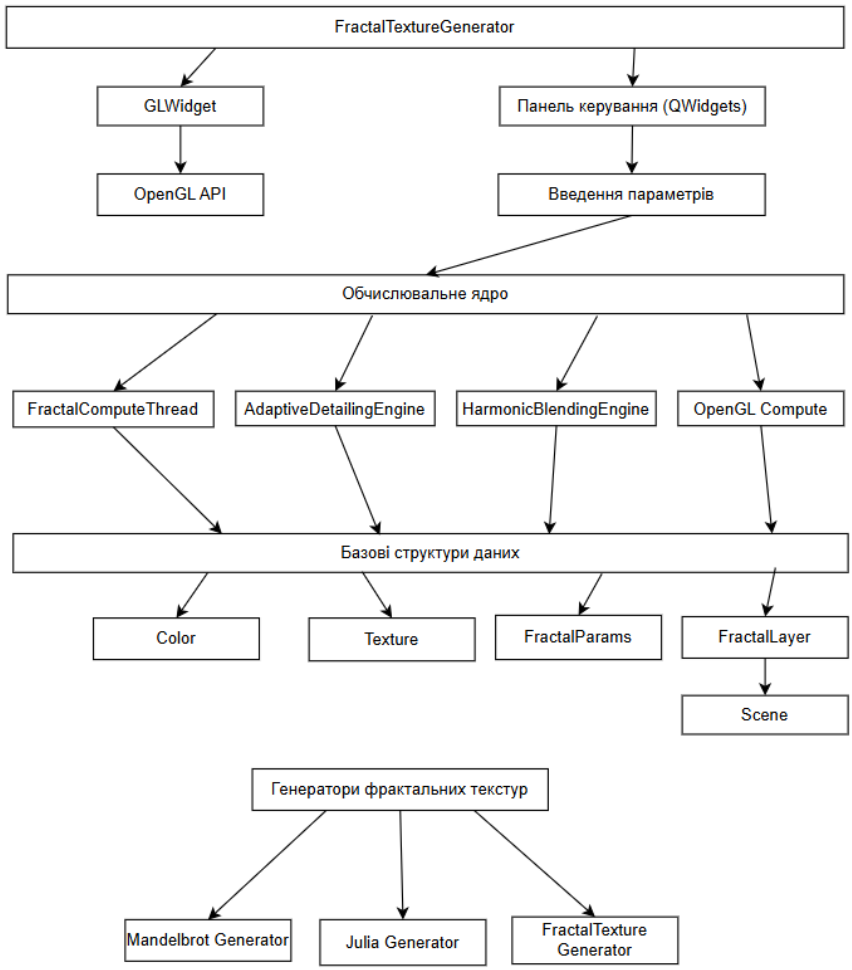


Рисунок 2.10 – Архітектура застосунку

Отже, архітектура застосунку забезпечує високу гнучкість для генерації різноманітних фрактальних текстур, використовує найсучасніші технології для максимальної швидкості і відкриває широкі можливості для експериментів. Модульна структура з чітким розділенням логіки робить систему легко розширюваною, а ефективне використання багатопотоковості та GPU-прискорення демонструє сучасний підхід до розробки програм для графічної обробки.

2.4 Розробка методу адаптивного фрактального деталізування текстур

Метод адаптивного фрактального деталізування текстур є підходом до генерації фрактальних текстур, який дозволяє динамічно адаптувати рівень

деталізації фракталу залежно від його просторового розташування в сцені та важливості для кінцевого зображення.

Метод володіє багаторівневою системою деталізації, де кожен рівень має свою глибину ітерацій фрактального алгоритму. Метод використовує просторову локалізацію, де центральні або найбільш помітні частини текстури генеруються з вищою деталізацією, тоді як периферійні або менш важливі області – з нижчою.

Для кожної точки текстури (x, y) визначається коефіцієнт важливості I(x, y) за формулою:

$$I(x, y) = W_1 \cdot V(x, y) + W_2 \cdot D(x, y) + W_3 \cdot C(x, y)$$ [13],

де:

- V(x, y) – коефіцієнт видимості точки в сцені;
- D(x, y) – відстань від центру уваги;
- C(x, y) – контрастність області;
- W1 — коефіцієнт, що визначає вагу фактора видимості при обчисленні важливості пікселя. Збільшення значення W1 призводить до рівномірнішого розподілу деталізації по всьому зображенню, оскільки надає більшу вагу фактору з найбільш рівномірним розподілом.
- W2 — коефіцієнт відстані від фокусу. W2 контролює вплив відстані пікселя від фокусної точки на його важливість. Фактор відстані обчислюється як:

distance_factor(p) = 1.0 - min(1.0, distance(p, focus_point) / focus_radius).

Збільшення W2 призводить до концентрації обчислювальних ресурсів у фокусній області зображення. Це створює ефект "лупи", де центральна частина зображення отримує більшу деталізацію порівняно з периферійними областями.

- W3 — коефіцієнт контрасту. W3 визначає вагу фактора локального контрасту, який вказує на наявність структурно важливих елементів. Фактор контрасту обчислюється на основі варіації значень яскравості в околі пікселя. Збільшення W3 призводить до концентрації обчислювальних ресурсів у областях з високим контрастом, таких як границі і складні структури фракталів. Це дозволяє більш детально відображати критично важливі візуальні елементи, зберігаючи ресурси на однорідних ділянках.

Кількість ітерацій для конкретної точки обчислюється як:

$$Iterations(x, y) = BaseIterations + I(x, y) \cdot AdditionalIterations$$

Метод адаптивного фрактального деталізування текстур має такі особливості:

1. Значне підвищення продуктивності без помітної втрати якості.
2. Можливість генерувати надзвичайно детальні текстури в ключових зонах.
3. Динамічна адаптація до умов рендерингу.
4. Зменшення навантаження на GPU.

Цей метод відрізняється від існуючих, які зазвичай використовують фіксований рівень деталізації для всього зображення або простий LOD (Level of Detail) на основі відстані.

Метод фрактального деталізування структур функціонує по принципу змінного розподілу деталізації по фрактальному зображенню відповідно до заздалегідь визначених параметрів значущості. Цей підхід відрізняється від традиційних методів рівномірної обробки, досягаючи вищої візуальної точності при збереженні обчислювальної ефективності.

Основна математична формула для визначення важливості пікселя виражається як:

$$I(p) = w_1V(p) + w_2D(p) + w_3C(p),$$

де:

- $I(p)$ – розрахунковий показник важливості для пікселя p;
- w_1, w_2, w_3 - вагові коефіцієнти;
- $V(p)$ – фактор видимості;
- $D(p)$ – фактор відстані від фокусної точки;
- $C(p)$ – фактор контрасту в околиці пікселя.

Фактор відстані обчислюється за формулою:

$$D(p) = 1.0 - \min(1.0, d(p,f)/r) \text{ [14]},$$

де:

- $d(p,f)$ – евклідова відстань між пікселем p і фокусною точкою f;

- r – параметр фокусного радіуса.

Фактор контрасту визначається через аналіз стандартного відхилення значень яскравості в межах визначеного околу пікселя:

$$C(p) = \sigma(L(N(p))) \text{ [15]},$$

де:

- σ – операція стандартного відхилення;
- L – перетворення яскравості (0.299R + 0.587G + 0.114B);
- N(p) вказує на матрицю околу навколо пікселя p.

Реалізація методу відбувається згідно з наступним процесом:

1. Встановлення базової кількості ітерацій, максимальної додаткової кількості ітерацій, вагових коефіцієнтів та фокусних параметрів.
2. Генерація текстури попереднього перегляду зі зниженою роздільною здатністю для початкової оцінки контрасту, коли це необхідно.
3. Розрахунок фактора важливості для кожного пікселя через складену формулу з використанням факторів видимості, відстані та контрасту.
4. Визначення оптимальної кількості ітерацій на основі формули важливості:

$$\text{iterations}(p) = \text{baseIterations} + [\text{importance}(p) \times \text{maxAdditionalIterations}]$$

5. Застосування балансування контрасту та коригування насиченості для оптимізації візуального представлення.

Метод використовує обчислювальні шейдери для паралельної обробки, що призводить до суттєвого покращення продуктивності. Обчислювальний шейдер використовує структуру робочої групи, яка ефективно розподіляє обчислення між доступними обчислювальними блоками.

Метод адаптивного фрактального деталізування текстур має кілька переваг:

1. Концентруючи обчислювальні ресурси на візуально значущих ділянках, метод досягає оптимального використання ресурсів.
2. Метод забезпечує вищу роздільну здатність деталей у фокусних областях та регіонах високого контрасту, що призводить до кращого представлення складних фрактальних структур.

3. Динамічно реагує на фокус користувача та характеристики сцени, відповідно коригуючи рівні деталізації.

4. Регульовані вагові коефіцієнти дозволяють налаштування для різних типів фракталів та сценаріїв застосування.

Порівняно з рівномірним розподілом деталей, метод адаптивного фрактального деталізування текстур реалізує нерівномірний розподіл ресурсів на основі візуальної значущості, на відміну від підходів рівномірної обробки. Це призводить до ефективнішого використання обчислювальних ресурсів та підвищеної візуальної якості на важливих ділянках.

Порівняно з альтернативними адаптивними методами, метод вирізняється завдяки інтеграції кількох факторів важливості (видимість, відстань, контраст), включенню фокусних механізмів, узгоджених із принципами людського сприйняття, спеціалізованій оптимізації для фрактальних структурних характеристик.

Реалізація передбачає спеціалізовані адаптації для різних типів фракталів:

1. Застосування для множини Мандельброта. Для генерації множини Мандельброта метод динамічно коригує кількість ітерацій формули $z = z^2 + c$ [16], забезпечуючи точніше відтворення складних граничних ділянок та мініатюрних реплікацій основної структури.

2. Застосування для множини Жюліа. При застосуванні до множин Жюліа алгоритм адаптує кількість ітерацій, враховуючи характерні ниткоподібні структури, часто присутні в цьому типі фракталів.

3. Застосування для процедурних текстур. Для процедурних фрактальних текстур метод модифікує не лише кількість ітерацій, а й параметри текстури (a_param, b_param, k_param), вносячи додаткову варіацію та деталізацію в значущих ділянках.

Метод адаптивної фрактальної деталізації текстур представляє собою складний підхід до генерації фрактальних зображень, який ефективно балансує використання обчислювальних ресурсів із візуальною якістю. Селективно розподіляючи обчислювальні ресурси відповідно до перцептивної важливості,

алгоритм досягає кращих результатів порівняно з традиційними методами рівномірної обробки.

2.5 Розробка методу гармонійного змішування шарів фракталів

Метод гармонійного змішування шарів фракталів представляє собою алгоритм для створення складних багатошарових фрактальних текстур, що базується на принципах теорії кольорових гармоній та математичної оптимізації переходів між шарами.

Алгоритм використовує спеціальний підхід до змішування різних типів фракталів в одній текстурі, де кожен шар взаємодіє з іншими за допомогою математично визначених гармонійних співвідношень. Метод дозволяє автоматично обирати оптимальні параметри змішування на основі аналізу спектральних характеристик кожного шару.

Для роботи методу проводяться такі операції:

1. Спектральний аналіз кожного фрактального шару для визначення його домінантних частот.

2. Алгоритм узгодження фаз для забезпечення плавних переходів між шарами.

3. Автоматичне балансування контрасту та насиченості.

4. Гармонізація кольорових палітр з використанням теорії кольорових гармоній.

Для кожної пари шарів L_1 і L_2 обчислюється матриця гармонічного змішування H:

$$H = [\cos(\theta) \ -\sin(\theta); \ \sin(\theta) \ \cos(\theta)] \cdot [a \ 0; \ 0 \ b] \ [17],$$

де:

θ – кут гармонійного зміщення, обчислений на основі спектрального аналізу

a, b – параметри масштабування, що визначають силу взаємодії між шарами

Результуюча текстура R обчислюється як:

$$R = \sum(w_i \cdot T_i \cdot \prod(H\_ij)), \text{ де } i \neq j$$

де:

w_i – ваговий коефіцієнт шару

T_i – текстура і-го шару

H_ij – матриця гармонійного змішування між шарами і та j

Цей метод дозволяє автоматично узгоджувати параметри між різними типами фракталів, та створювати плавні переходи між різними станами текстури.

Метод гармонійного змішування шарів фракталів (ГЗШФ) представляє собою підхід до композиції декількох фрактальних текстур в єдине зображення з покращеними візуальними характеристиками. На відміну від простого накладання шарів із використанням прозорості, ГЗШФ враховує спектральні, структурні та кольорові особливості кожного шару, створюючи естетично привабливий і візуально цілісний результат.

Гармонійне змішування шарів фракталів ґрунтується на аналізі частотних та структурних характеристик фрактальних зображень і застосуванні принципів теорії гармонії до їх комбінування. Ключовою ідеєю методу є встановлення математично обґрунтованих взаємозв'язків між різними фрактальними структурами для досягнення природного та збалансованого результату змішування.

Нехай маємо n фрактальних шарів, представлених як функції:

$$F_1(x,y),\ F_2(x,y),\ ...,\ F_n(x,y),$$

де кожна функція визначає колір у точці (x,y). Гармонійне змішування здійснюється за формулою:

$$R(x,y) = \Sigma\ w_i \cdot T(F_i(x,y),\ \{F_1...F_n\}),$$

де:

- R(x,y) – результуючий колір у точці (x,y)
- w_i – ваговий коефіцієнт (важливість) і-го шару
- T() – функція гармонійного перетворення, яка адаптує колір шару з урахуванням властивостей усіх інших шарів.

Функція гармонійного перетворення T визначається як:

$$T(F_i(x,y),\ \{F_1...F_n\}) = F_i(x,y) + \Sigma_{j=1}^{n}\ (j \neq i)\ H_{ij}(F_j(x,y)),$$

де H_{ij} – гармонійна матриця перетворення між шарами i та j.

Алгоритм методу гармонійного зглажування шарфів фракталів складається з наступних основних етапів:

1. Підготовка шарів, де кожен фрактальний шар генерується окремо з власними параметрами.

2. Аналіз та адаптація колірних схем різних шарів для забезпечення гармонійного комбінування.

3. Визначення важливості кожного шару на основі його контрасту, складності та візуальної значущості.

4. Обчислення домінантних частот, амплітуд та фаз для кожного шару через перетворення Фур'є.

5. Для кожної пари шарів створюється матриця перетворення, яка визначає спосіб їх взаємодії.

6. Поетапне змішування всіх шарів з застосуванням обчислених перетворень.

7. Кінцева оптимізація загального контрасту та насиченості для цілісного сприйняття.

Для кожного шару виконується двомірне перетворення Фур'є:

$$F(u,v) = \sum_x \sum_\gamma f(x,y) \cdot e^{(-j2\pi(ux/M + vy/N))} \text{ [18]},$$

де:

- F(u,v) – компонент Фур'є на частотах u і v;
- f(x,y) – значення яскравості пікселя в точці (x,y);
- M, N – розміри зображення.

Домінантні частоти визначаються за амплітудним спектром:

$$|F(u,v)| = \sqrt{(Re(F(u,v))^2 + Im(F(u,v))^2)} \text{ [19]},$$

Для кожної пари шарів (i, j) обчислюється гармонійна матриця H_{ij}:

H_{ij} = {

a: $\sqrt{(contrast_i / contrast_j)}$,

b: 1 / a,

θ: atan2(sin(π·$freq_i$/$freq_j$), cos(π·$freq_i$/$freq_j$))

},

де:

- a, b – коефіцієнти масштабування;
- θ – кут повороту в просторі ознак;
- $contrast_i$ – середній контраст і-го шару;
- $freq_i$ – домінантна частота і-го шару.

Для кожного пікселя застосовується перетворення кольору через гармонійні матриці. У двовимірному колірному просторі (наприклад, проекція RGB на площину) це виглядає так:

$$[x\_new, y\_new] = [a \cdot \cos(\theta) \cdot x - b \cdot \sin(\theta) \cdot y, a \cdot \sin(\theta) \cdot x + b \cdot \cos(\theta) \cdot y],$$

де [x, y] — початкові колірні координати, [x_new, y_new] — перетворені координати.

Після змішування шарів виконується оптимізація глобального контрасту:

$$I\_adjusted = (I - Imin) / (Imax — Imin),$$

де Imin і Imax визначаються з гістограми з обрізанням найнижчих і найвищих 5% значень.

Насиченість оптимізується через коригування колірних компонент в просторі HSV:

$$S\_adjusted = S \cdot (targetSaturation / avgSaturation).$$

Метод гармонійного змішування шарів фракталів надає ряд переваг порівняно з традиційними підходами до комбінування зображень:

1. Результуючі зображення мають високу привабливість завдяки взаємодії шарів на рівні частотних характеристик.
2. Метод зберігає та підкреслює ключові структурні елементи кожного шару, не втрачаючи деталей.
3. Адаптивна гармонізація колірних схем забезпечує природне поєднання різних палітр.
4. Результат має високу просторову зв'язність, уникаючи артефактів на

межах різних текстурних зон.

5. Можливість точного контролю через зміну ваг шарів та параметрів гармонійних перетворень.

Метод гармонійного змішування шарів фракталів дозволяє створювати композитні фрактальні зображення з високими естетичними якостями. Завдяки врахуванню частотних характеристик та структурних особливостей кожного шару, метод гармонійного змішування шарів фракталів забезпечує природне і гармонійне поєднання різних фрактальних текстур.

2.6 Розробка алгоритмів роботи застосунку

На рисунку 2.11 наведено блок-схему алгоритму адаптивної деталізації фракталу.

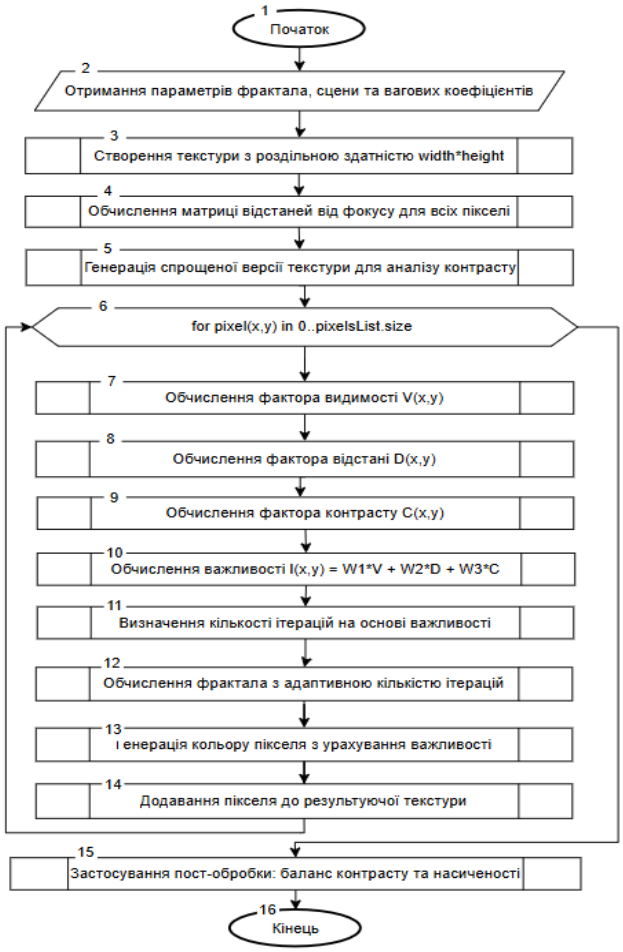


Рисунок 2.11 – Блок-схема алгоритму адаптивної деталізації фрактальної текстури

Алгоритм адаптивної фрактальної деталізації оптимізує обчислювальні ресурси при генерації фрактальних зображень шляхом диференційованого розподілу ітерацій. Замість того, щоб застосовувати однакову кількість ітерацій до кожного пікселя, метод аналізує важливість кожної ділянки зображення на основі трьох ключових факторів: видимості, відстані від фокусної точки та контрасту[20].

Алгоритм гармонійного змішування шарів фракталів забезпечує естетично привабливе поєднання кількох фрактальних текстур в єдине зображення.

Блок-схема цього алгоритму наведена на рисунку 2.12.

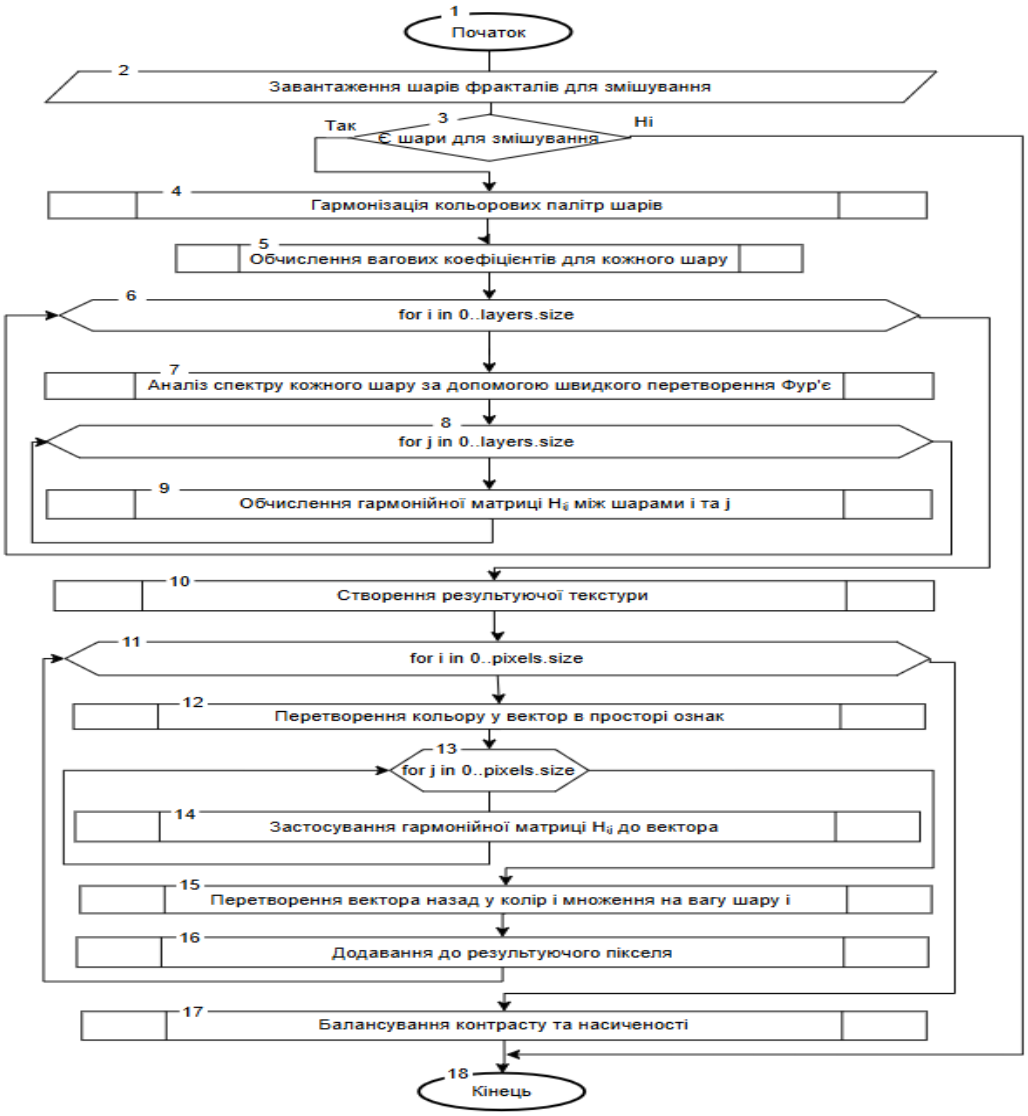


Рисунок 2.12 – Блок-схема алгоритму гармонійного змішування шарів фракталів

Для кожного пікселя обчислюється композитний показник важливості, який

визначає оптимальну кількість ітерацій фрактальної формули. Ділянки з вищою важливістю (наприклад, ближчі до фокусної точки або з високим контрастом) обробляються з більшою кількістю ітерацій, що забезпечує вищу деталізацію. Водночас, менш важливі ділянки обраховуються з меншою кількістю ітерацій, що зберігає обчислювальні ресурси без суттєвої втрати візуальної якості.

Алгоритм також застосовує спеціальні колірні перетворення, які забезпечують візуальну узгодженість між ділянками з різним рівнем деталізації, підкреслюючи важливі структурні елементи фрактала.

На відміну від простого накладання з прозорістю, алгоритм гармонійного змішування шарів фракталів аналізує спектральні та структурні характеристики кожного шару та застосовує математично обґрунтовані гармонійні перетворення.

Процес починається зі спектрального аналізу кожного шару через двовимірне перетворення Фур'є, що дозволяє виявити домінантні частоти, амплітуди та фази. На основі цього аналізу для кожної пари шарів обчислюється гармонійна матриця, яка визначає оптимальне перетворення кольору для забезпечення візуальної когерентності.

При змішуванні шарів кожен піксель перетворюється з урахуванням гармонійних матриць, що зберігає структурні особливості шарів, але забезпечує їхнє гармонійне поєднання. Завершальна стадія включає балансування контрасту та насиченості для оптимізації загального візуального сприйняття.

Алгоритм генерації фракталів на графічному процесорі з використанням compute shaders використовує паралельні обчислювальні можливості графічного процесора для суттєвого прискорення генерації фрактальних зображень. Центральним елементом виступають обчислювальні шейдери OpenGL, які забезпечують масове паралельне виконання ітеративних фрактальних формул.

Процес починається з підготовки шейдерного коду, специфічного для типу фрактала (Мандельброт, Жюліа або фрактальна текстура), та його компіляції. Обчислювальне завдання розбивається на робочі групи розміром 16×16 пікселів, що оптимізує використання кеш-пам'яті GPU. Шейдер виконує ітеративні обчислення фрактальної формули для кожного пікселя паралельно, застосовуючи

відповідні перетворення кольору.

Блок-схема цього алгоритму наведена на рисунку 2.13.

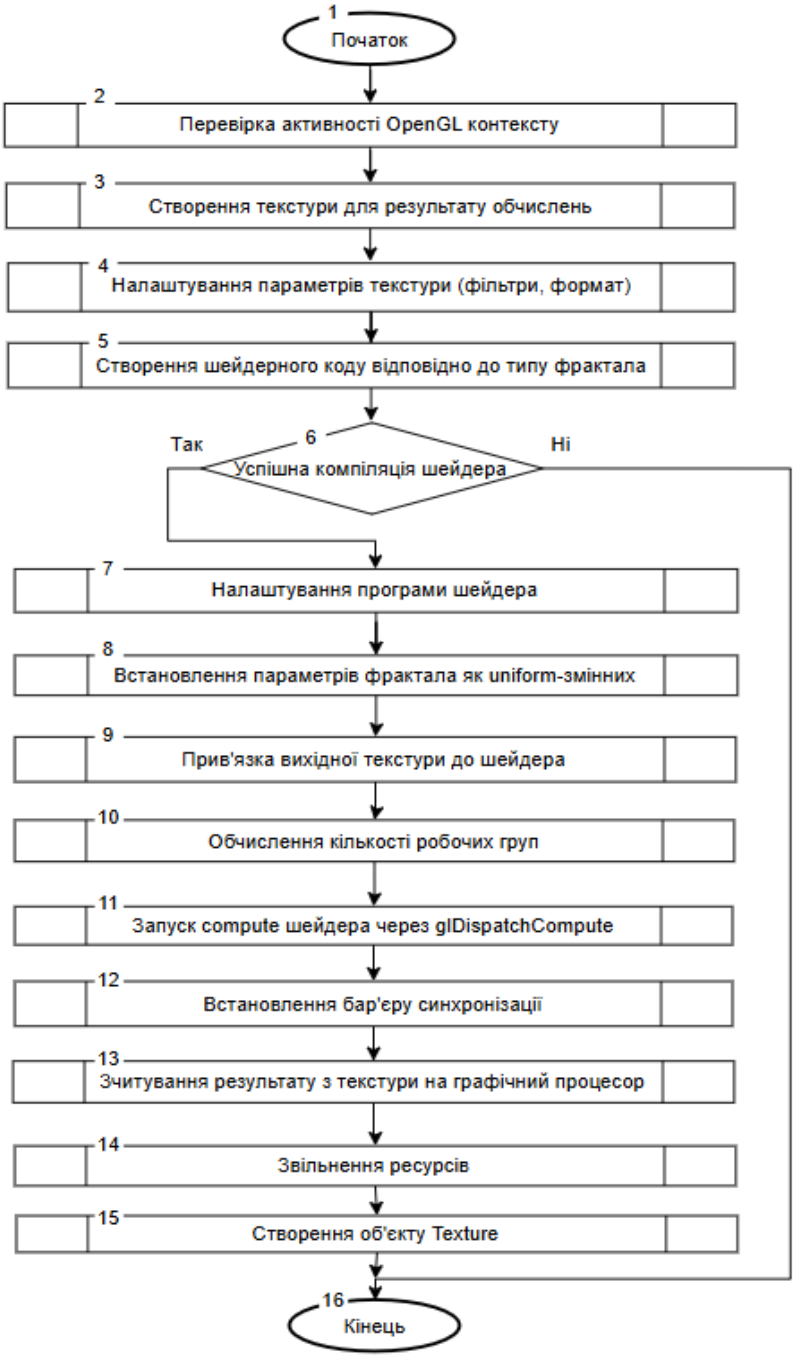


Рисунок 2.13 – Блок-схема алгоритму генерації фракталів на графічному процесорі з використанням compute shaders

Алгоритм оптимізований для мінімізації передачі даних між центральним процесором та графічним процесором, оскільки всі обчислення виконуються безпосередньо на графічному процесорі, а результати зберігаються в текстурі, що

згодом зчитується для формування кінцевого зображення.

Алгоритм рендерингу фракталів через OpenGL забезпечує ефективне відображення обчислених текстур на екрані з використанням сучасного конвеєра OpenGL. Він інкапсульований у класі GLWidget, що інтегрується з Qt-інтерфейсом.

Блок-схема цього алгоритму наведена на рисунку 2.14.

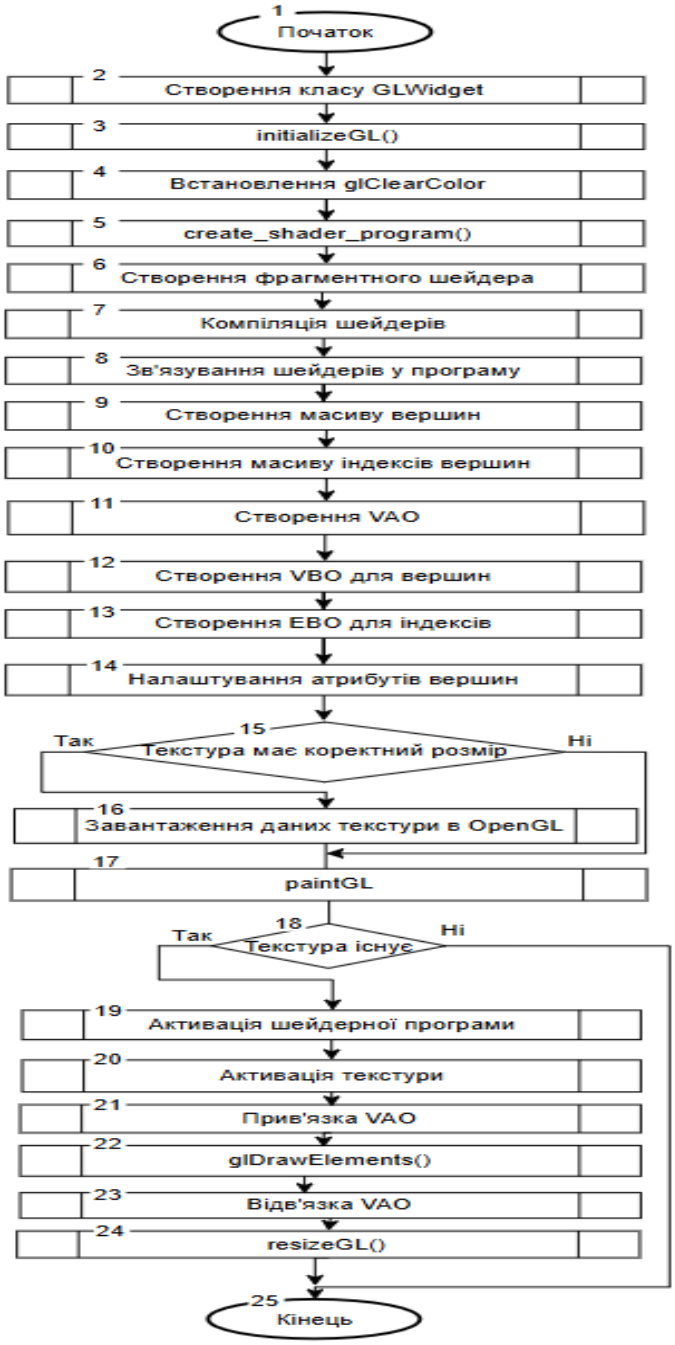


Рисунок 2.14 – Блок-схема алгоритму рендерингу фракталів через OpenGL

Процес рендерингу включає три основні етапи: ініціалізацію OpenGL-контексту, налаштування графічного конвеєра та безпосередню візуалізацію. Під час ініціалізації створюються та компілюються вершинний і фрагментний шейдери, що утворюють шейдерну програму, а також формується геометрія (квадрат), на яку накладається текстура.

Для відображення фрактальних текстур на екрані використовується оптимізований підхід з одним квадратом на весь екран, покритим фрактальною текстурою. Це мінімізує кількість вершин та спрощує процес рендерингу, дозволяючи зосередити обчислювальні ресурси на генерації текстури, а не на її відображенні.

Алгоритм також забезпечує ефективне оновлення відображення при зміні текстури та коректне масштабування при зміні розміру вікна, що гарантує стабільну взаємодію з користувачем.

Описані алгоритми реалізують основний функціонал застосунку для формування текстур з використанням фракталів.

2.7 Висновки

У цьому розділі було проведено аналіз інформаційного забезпечення застосунку, описано усі вхідні та вихідні дані, що забезпечують роботу застосунку. Описано механізм роботи адаптивної деталізації фрактальної текстури, алгоритм генерації фракталів на графічному процесорі з використанням compute shaders, алгоритму рендерингу фракталів через OpenGL, алгоритм гармонійного змішування шарів фракталів. Розроблено модель системи у вигляді діаграми компонентів, яка демонструє взаємозв’язок між основними компонентами застосунку.

3 РОЗРОБКА ЗАСТОСУНКУ

3.1 Варіантний аналіз та вибір мови програмування розробки

Для аналізу та вибору мови програмування розробки розглянемо такі мови, як: Python, C++, Java.

Python [21] – інтерпретована мова програмування високого рівня з динамічною типізацією, що надає перевагу читабельності коду та продуктивності програміста. Його стандартна бібліотека та розширена екосистема наукових і графічних пакетів, таких як NumPy, PyQt та прив'язки OpenGL, роблять його придатним для розробки спеціалізованих застосунків, як-от генератори фрактальних текстур. Спрощений синтаксис Python та об'єктно-орієнтована парадигма забезпечують швидкі цикли розробки та полегшують реалізацію складних математичних алгоритмів, необхідних для генерації фракталів, тоді як його кросплатформна сумісність гарантує доступність на різних операційних системах.

C++ [22] – компільована, статично типізована мова програмування, що пропонує високу продуктивність для обчислювальновибагливих застосунків. Ця мова забезпечує прямий доступ до апаратного забезпечення та можливості для управління пам'яттю, які можуть значно прискорити процеси рендерингу фракталів. C++ підтримує принципи об'єктно-орієнтованого проєктування, що дозволяє створювати комплексні ієрархії класів для представлення фрактальних алгоритмів та компонентів візуалізації. Екосистема C++ включає бібліотеки, такі як Qt для розробки інтерфейсу та різні реалізації OpenGL для обробки графіки, хоча реалізація зазвичай вимагає більш тривалого часу розробки порівняно з інтерпретованими мовами.

Java [23] – строго типізована об'єктно-орієнтована мова програмування, основний принцип якої є реалізація застосунків з платформною незалежністю. Віртуальна машина JVM дозволяє застосункам працювати послідовно в різних обчислювальних середовищах. Java надає суттєву вбудовану функціональність для розробки графічного інтерфейсу користувача через такі фреймворки, як JavaFX та

Swing. Java пропонує хороші характеристики продуктивності після компіляції JIT, в той же час зазвичай демонструє дещо вищі вимоги до пам'яті порівняно з нативними рішеннями. Комплексна модель багатопотоковості Java підтримує можливості паралельної обробки, корисні для генерації фракталів, хоча може вимагати більш громіздких кодових шаблонів реалізації.

Для порівняння можливостей описаних мов програмування, було сформовано таблицю 3.1.

Таблиця 3.1 – Порівняльний аналіз мов програмування

Характеристика	Python	C++	Java
Продуктивність для фрактальних алгоритмів	+	+	-
Можливості інтеграції з графічним процесором	+	+	+
Наукові обчислювальні бібліотеки	+	-	-
Підтримка спільноти для графіки	+	+	-
Інтеграція з фреймворками графічного інтерфейсу	+	+	+
Реалізація паралельної обробки	+	+	+
Простота управління пам'яттю	+	-	+
Сумарний бал	7	5	4

В результаті порівняльного аналізу було визначено, що Python має вищий сумарний бал порівняно з C++ та Java, має кращі характеристики для розробки застосунку для роботи з фрактальними текстурами, тому було обрано Python для розробки застосунку.

3.2 Розробка головного модуля застосунку

Головний модуль застосунку являє собою вікно, в якому знаходиться область відображення фрактальних текстур, та меню їх налаштування. Він дозволяє генерувати різного роду текстури, змінювати їх параметри, активувати гармонійне змішування шарів та адаптивну деталізацію.

Розроблено структурну схему вікна головного модуля застосунку, яка наведена на рисунку 3.1.

Рисунок 3.1 – Структурна схема вікна головного модуля застосунку

Складові елементи інтерфейсу головного вікна застосунку:

1. Назва застосунку.
2. Кнопка згортання вікна.
3. Кнопка розкриття вікна.
4. Кнопку закриття програми.
5. Область відображення текстури.
6. Меню налаштування текстури.

Розроблено меню налаштування текстури, яке дозволяє задавати параметри текстур, активувати режим адаптивної деталізації, обирати вид текстур, змішувати

шари текстур, експортувати зображення текстури. Структурна схема інтерфейсу меню налаштування текстур наведено на рисунку 3.2.

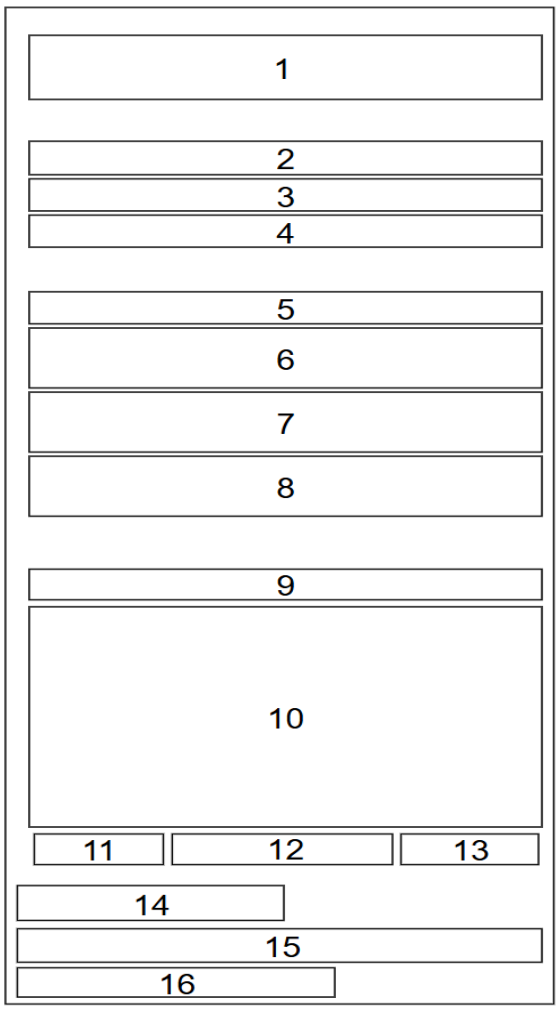


Рисунок 3.2 – Структурна схема меню налаштування текстур

Складові елементи меню налаштування текстур:

1. Випадаючий список для вибору текстури.
2. Поле для координати x.
3. Поле для координати y.
4. Поле для координати K.
5. Прапорець активації адаптивної деталізації.
6. Повзунок для ваги W1.
7. Повзунок для ваги W2.
8. Повзунок для ваги W3.
9. Прапорець активації гармонійного змішування шарів.

10. Список доданих шарів.

11. Кнопка «Додати шар».

12. Кнопка «Додати поточне як шар».

13. Кнопка «Видалити шар».

14. Кнопка «Обчислити».

15. Кнопка «Експортувати текстуру».

16. Панель статусу.

Блок-схема алгоритму роботи головного модуля застосунку наведена на рисунку 3.3.

Рисунок 3.3 – Блок-схема алгоритму роботи головного модуля застосунку

Таким чином, головний модуль застосунку надає функціонал для створення фрактальних текстур, їх налаштування за допомогою зміни параметрів x, y та K, а також шляхом налаштування адаптивної деталізації. Також міститься функціонал для гармонійного змішування текстур та експорту готових текстур у окремий файл для подальшого використання.

3.3 Розробка модуля змішування шарів фракталів

Модуль змішування шарів фракталів реалізує відповідний метод гармонійного змішування шарів фракталів, який дозволяє створювати складні текстури з різних фракталів. Блок-схема алгоритму роботи модуля наведена на рисунку 3.4.

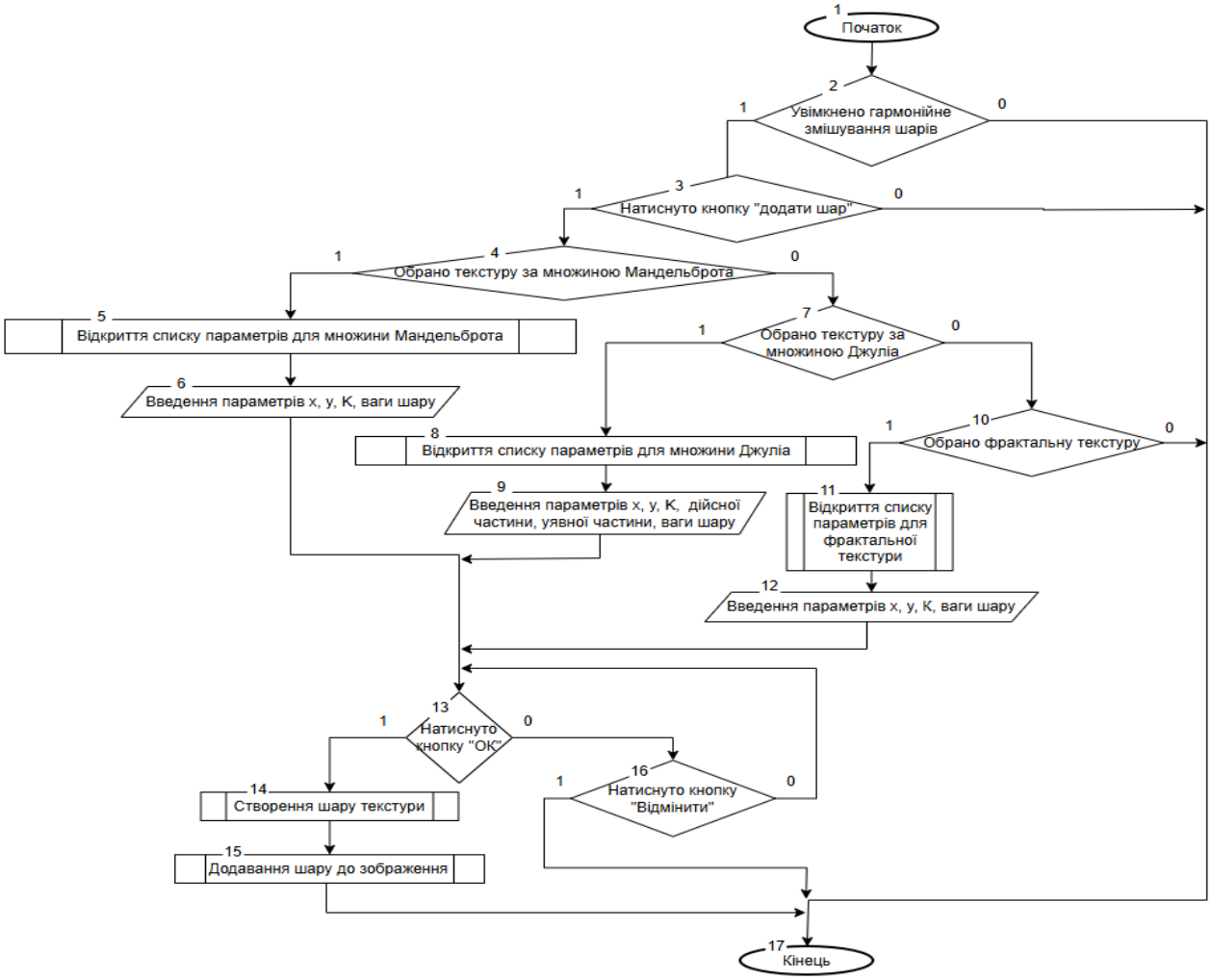


Рисунок 3.4 – Блок-схема алгоритму роботи модуля змішування шарів фрактальних текстур

Для роботи цього модуля також розроблено відповідний інтерфейс, який дозволяє налаштовувати шари фрактальних текстур для гармонійного змішування.

Розроблено вікно для налаштування шару текстури за множиною Мандельброта, структура якого наведена на рисунку 3.5.

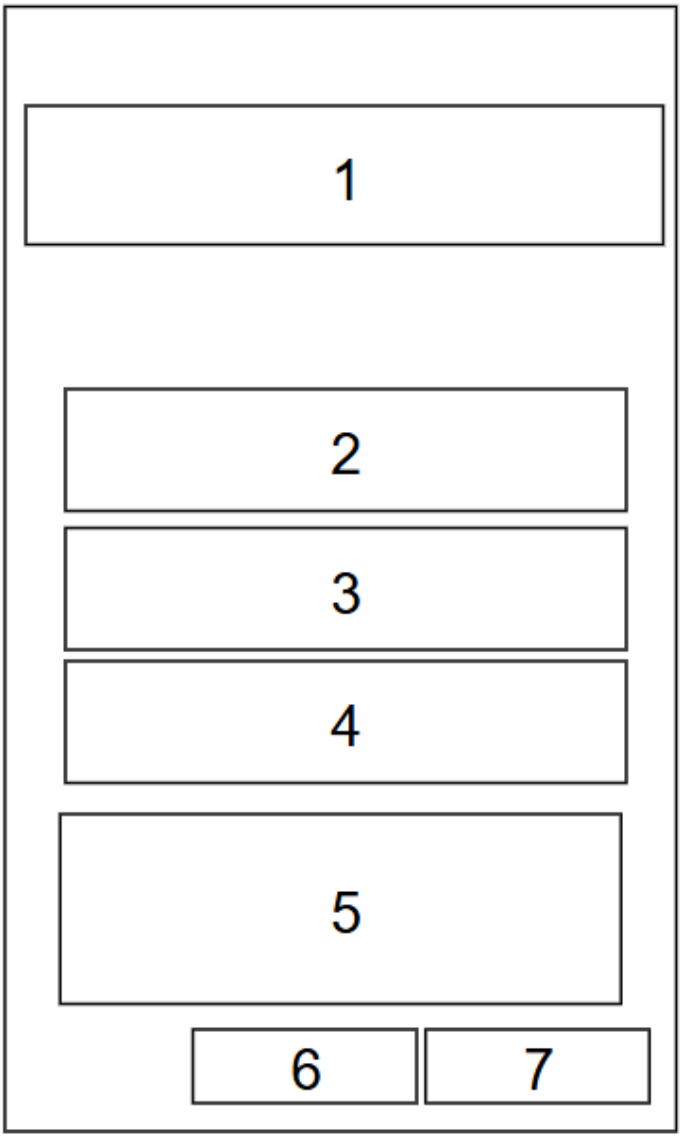


Рисунок 3.5 – Структура інтерфейсу вікна додавання текстури за множиною Мандельброта

Складові елементи інтерфейсу вікна додавання текстури за множиною Мандельброта:

1. Меню вибору фрактала.

2. Поле для параметра x.

3. Поле для параметра y.

4. Поле для параметра K.

5. Повзунок ваги шару в текстурі.

6. Кнопка «ОК».

7. Кнопка «Відмінити».

Також розроблено окреме вікно для додавання текстури за множиною Джуліа, адже цей тип фракталу має особливі параметри: дійсна частина та уявна частина.

Структурна схема вікна наведена на рисунку 3.6.

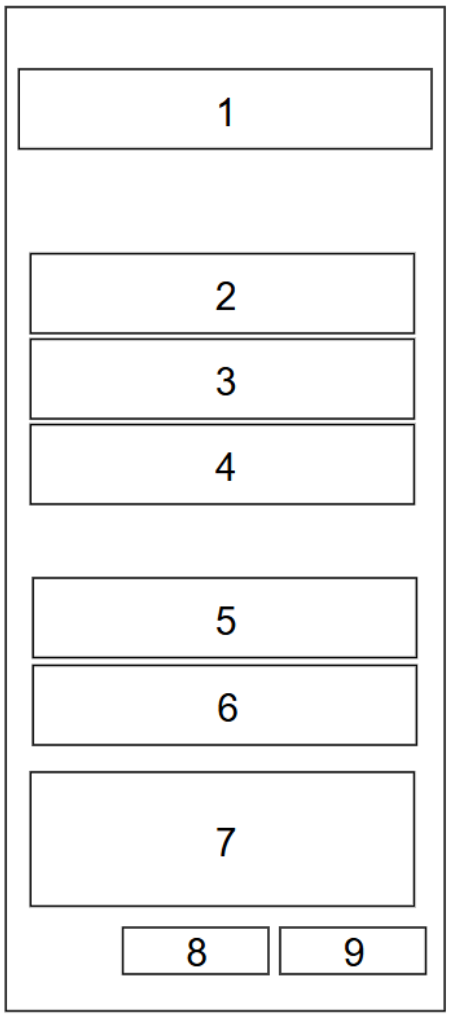


Рисунок 3.6 – Структура інтерфейсу вікна додавання текстури за множиною Джуліа

Складові елементи інтерфейсу вікна додавання шару за множиною Джуліа:

1. Меню вибору фрактала.
2. Поле для параметра x.
3. Поле для параметра y.
4. Поле для параметра K.
5. Повзунок ваги шару в текстурі.
6. Поле для дійсної частини.
7. Поле для уявної частини.
8. Кнопка «ОК».
9. Кнопка «Відмінити».

Таким чином, розроблений функціонал та інтерфейс дозволяє змішувати різні фрактальні текстури у єдине зображення, аналогічно звичайній одношаровій фрактальній текстурі.

3.4 Висновки

У цьому розділі було проведено порівняльний аналіз мов програмування, та обрано мову Python з огляду на її можливості у роботі з фракталами з використанням наявних бібліотек, які значно покращують та полегшують процес розробки, та дозволяють якісно налаштовувати процес генерації зображень з використанням ресурсів графічного процесора.

Розглянуто розробку ключових модулів:

- головний модуль застосунку, в якому налаштовуються та відображаються згенеровані текстури;

- модуль змішування шарів, який реалізує метод гармонійного змішування шарів фрактальних текстур.

Розроблені модулі дозволяють користувачам формувати власні унікальні текстури з використанням фракталів, налаштовувати їх під власні потреби та експортувати готові фрактальні текстури для подальшого використання у своїх потребах.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз методів тестування

Тестування застосунку є останнім та дуже важливим етапом розробки будь-якого програмного застосунку. Під час процесу тестування визначається відповідність роботи застосунку поставленим на початку розробки вимогам, відсутність помилок при роботі, очікувана реакція застосунку на дії користувача, видача очікуваного результату, правильність роботи алгоритмів роботи застосунку[24].

Модульне тестування представляє собою фундаментальний підхід до забезпечення якості програмного забезпечення з генерації фрактальних текстур, що передбачає ізольовану перевірку функціональності окремих компонентів системи. Для даного застосунку особливої уваги вимагають алгоритмічні модулі, зокрема класи AdaptiveDetailingEngine та HarmonicLayerBlendingEngine, функціональність яких визначає коректність математичних обчислень при генерації фракталів. Імплементація модульних тестів дозволяє верифікувати виконання алгоритмів та їх відповідність математичним моделям. Для цього можна створювати тестові випадки з вхідними параметрами за замовчуванням та розрахованими еталонними результатами, що актуально для тестування граничних умов у фрактальних обчисленнях, де помилки округлення та неточності алгоритмів можуть призводити до суттєвих відхилень у кінцевому візуальному представленні.

Інтеграційне тестування забезпечує верифікацію коректності взаємодії різних компонентів застосунку, що є особливо важливим для системи, яка поєднує алгоритми генерації фракталів з інтерфейсом користувача та механізмами візуалізації через OpenGL. Основним об'єктом тестування в цьому контексті виступають взаємодії між обчислювальними потоками FractalComputeThread та основним потоком інтерфейсу, а також передача даних між GPU-прискореними обчисленнями та системою рендерингу [25]. Необхідним є розробка тестових сценаріїв, які охоплюють повний цикл обробки даних – від налаштування

параметрів фракталу користувачем до відображення результату, контролюючи проміжні стани системи та коректність передачі даних між компонентами.

Тестування графічного інтерфейсу користувача для застосунку генерації фрактальних текстур передбачає верифікацію коректності відображення елементів керування, їх інтуїтивності та відповідності очікуваній поведінці. Методологія включає як автоматизоване тестування інтерфейсу з використанням інструментів для взаємодії з Qt-елементами, так і проведення юзабіліті-тестування з реальними користувачами. Ключовими аспектами для перевірки є коректність відображення шарів у режимі гармонійного змішування, візуалізації прогресу при тривалих обчисленнях, та стабільність інтерфейсу при інтенсивній взаємодії користувача з елементами керування [26].

Функціональне тестування застосунку має зосереджуватися на верифікації коректності реалізації ключових можливостей системи, таких як генерація різних типів фракталів, адаптивна деталізація та гармонійне змішування шарів [27]. Методологія передбачає розробку комплексних тестових сценаріїв, що охоплюють різноманітні комбінації параметрів і режимів роботи застосунку. Особливу увагу слід приділяти тестуванню граничних випадків, таких як генерація фракталів з високою деталізацією, обробка складних структур при гармонійному змішуванні багатьох шарів, та коректність збереження і завантаження створених текстур. Для кожної функціональної можливості доцільно розробити набір очікуваних результатів, з якими можна порівнювати фактичну поведінку системи, використовуючи як точні числові порівняння для математичних алгоритмів, так і метрики візуальної подібності для оцінки якості згенерованих текстур [28].

4.2 Тестування застосунку

Тестування застосунку розпочинається з його початкового стану, який користувач спостерігає при його запуску. Початковий стан застосунку передбачає наявність порожньої області, на місці якої будуть відображатися згенеровані фрактальні текстури після виконання відповідних дій користувача. У правій частині екрану застосунку розміщено панель налаштування фрактала. Початковий

стан застосунку наведено на рисунку 4.1.

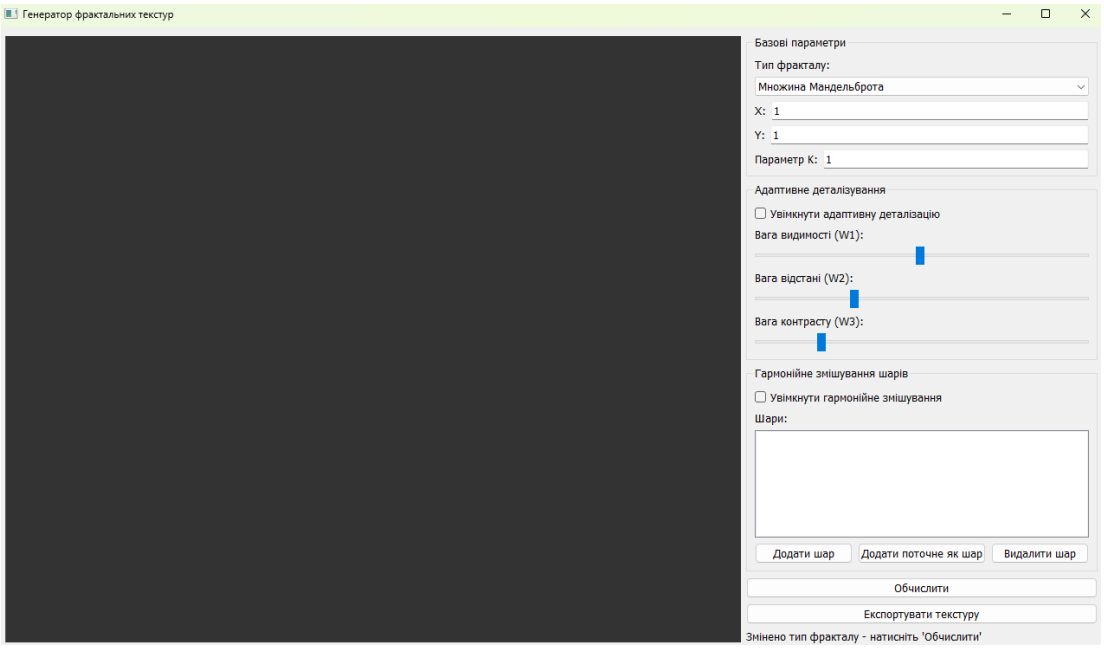


Рисунок 4.1 – Початковий стан застосунку

Згенеруємо множину Мандельброта з параметрами X=1, Y=1, K=1. Результат продемонстровано на рисунку 4.2.

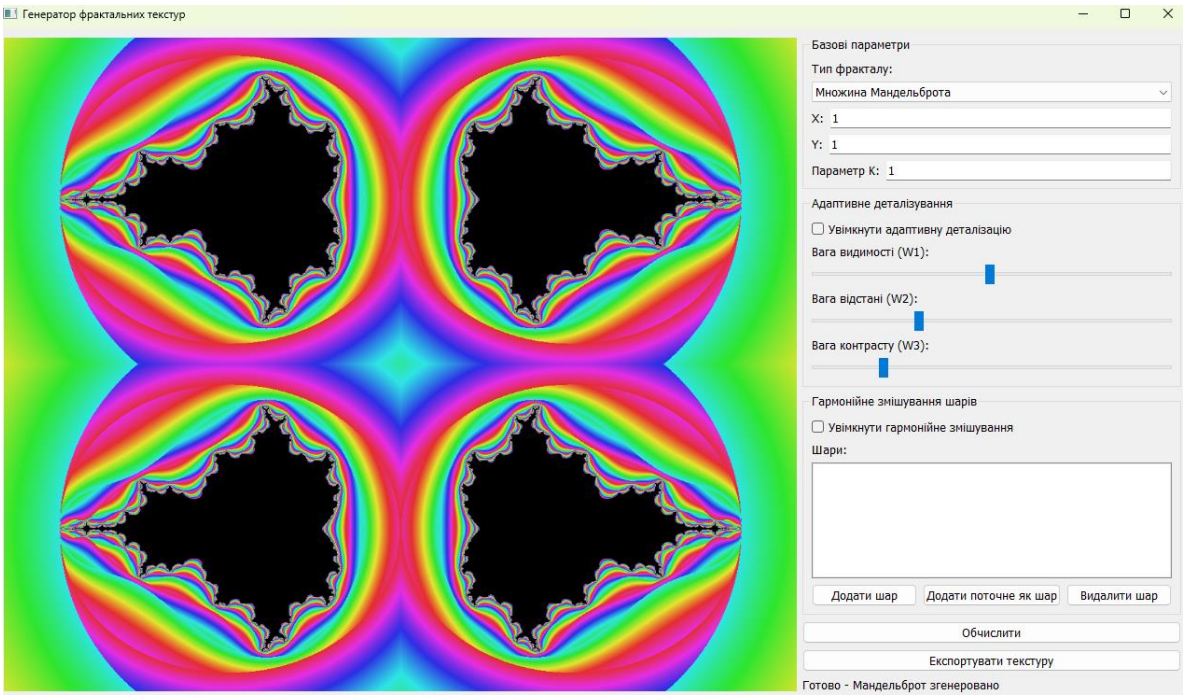


Рисунок 4.2 – Згенерована текстура за множиною Мандельброта з параметрами X=1, Y=1, K=1

Змінимо параметр K=500 та заново згенеруємо текстуру. Отриманий результат продемонстровано на рисунку 4.3.

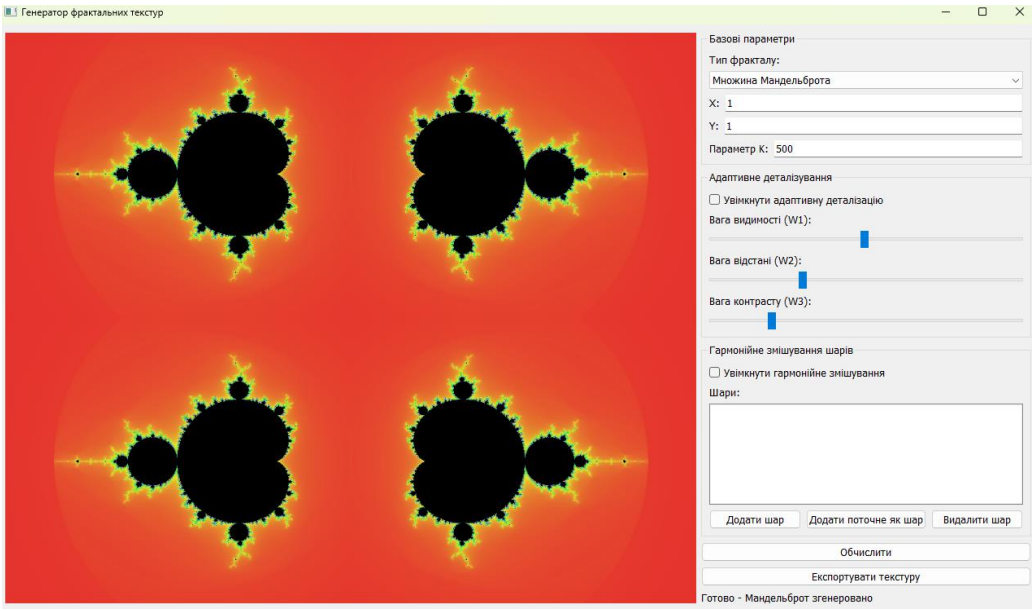


Рисунок 4.3 – Текстура за множиною Мандельброта при K=500

Змінимо координати X=50 та Y=50. Результат наведено на рисунку 4.4.

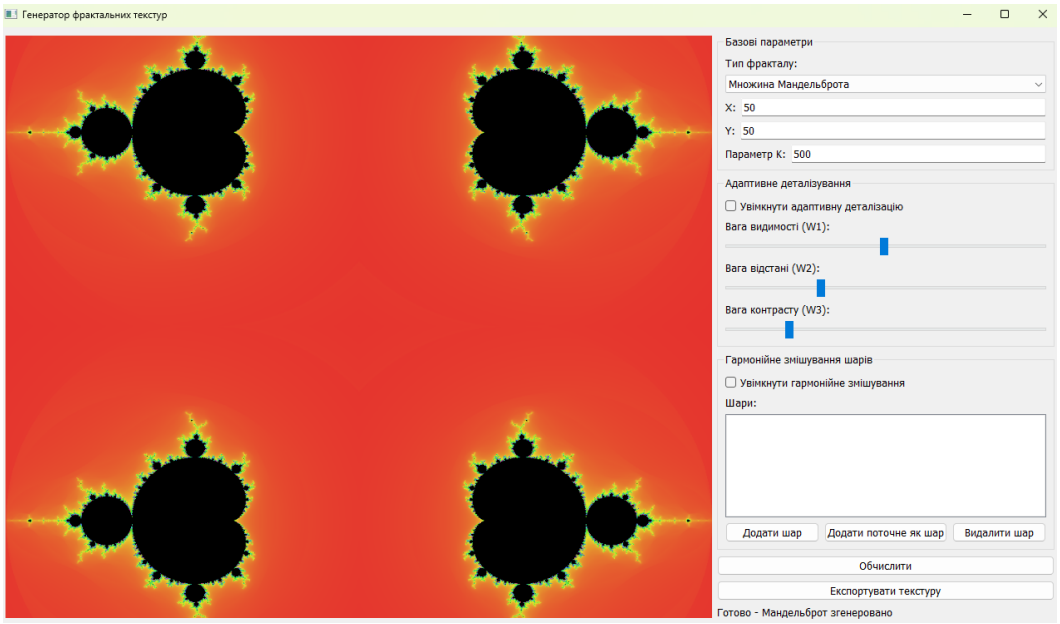


Рисунок 4.4 – Текстура за множиною Мандельброта при X=50, Y=50

Можемо побачити, що при зміні координат X та Y, зображення фракталів

Мандельброта змістилися від центру координат, які знаходяться посередині області текстури на 50 пунктів по осям X та Y.

Тепер увімкнемо адаптивну деталізацію. Результат зміненої текстури наведено на рисунку 4.5.

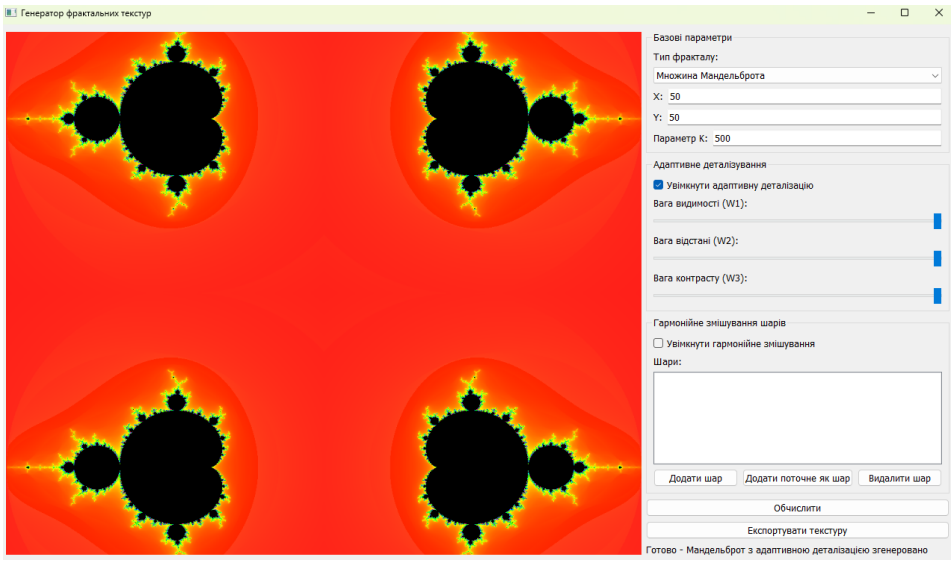


Рисунок 4.5 – Текстура з адаптивною деталізацією

Тепер згенеруємо текстуру за множиною Джуліа з параметрами X=50, Y=50, K=500 та увімкненою адаптивною деталізацією. Результат наведено на рисунку 4.6.

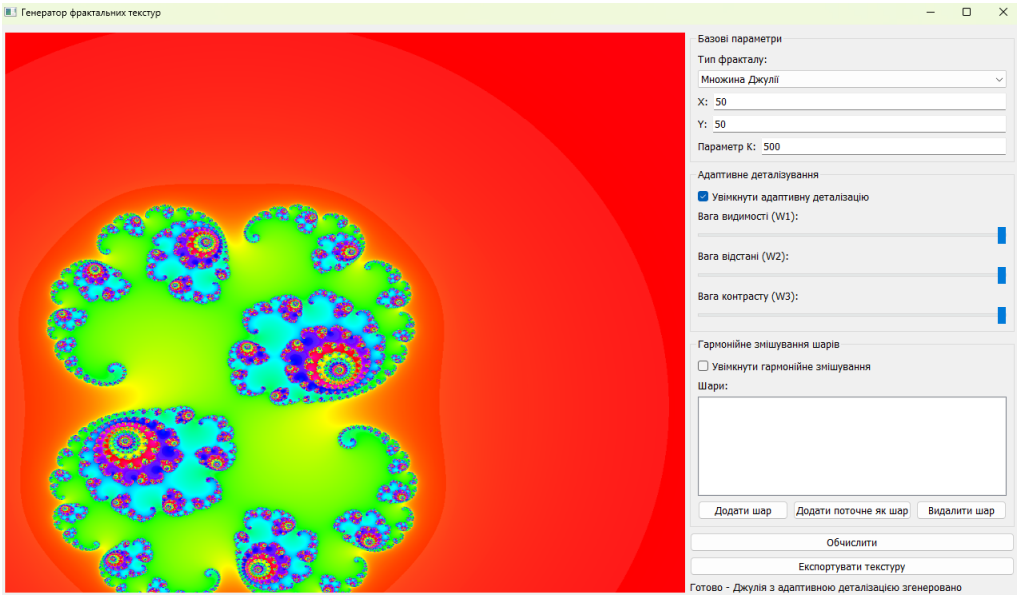


Рисунок 4.6 – Фрактальна текстура за множиною Джуліа при X=50, Y=50, K=500

Тепер вимкнено адаптивну деталізацію. Результат наведено на рисунку 4.7.

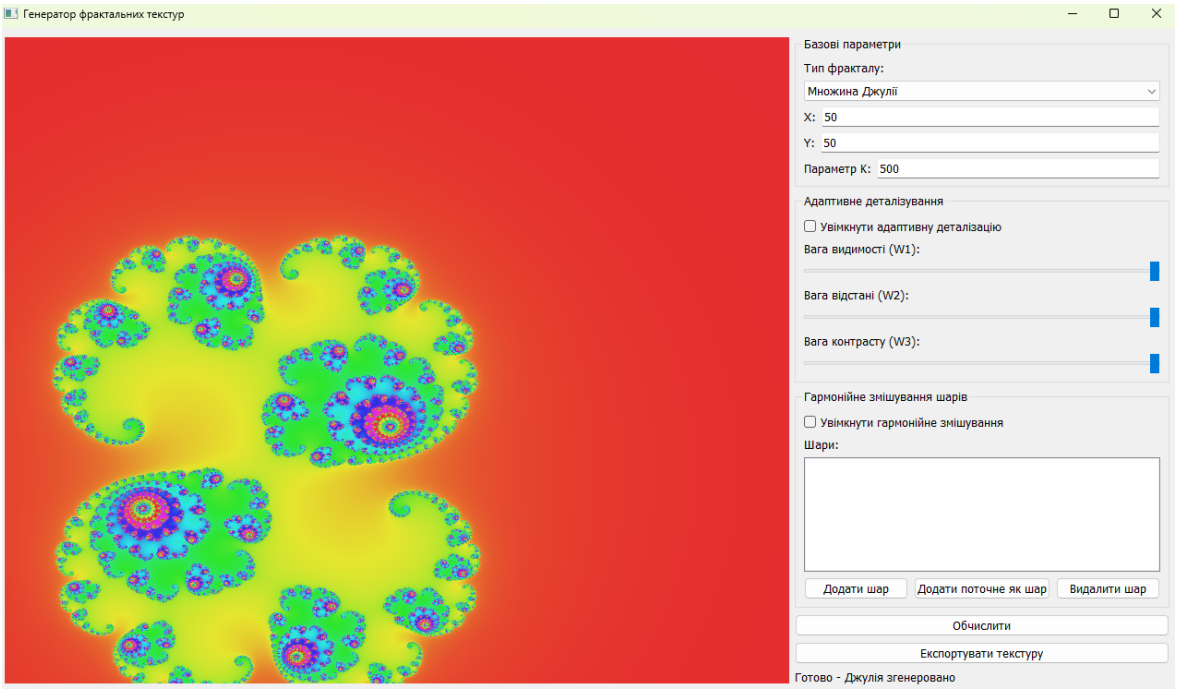


Рисунок 4.7 – Текстура за множиною Джуліа без деталізації

Змінимо параметри X = 1, Y=1, K=1. Результат наведено на рисунку 4.8.

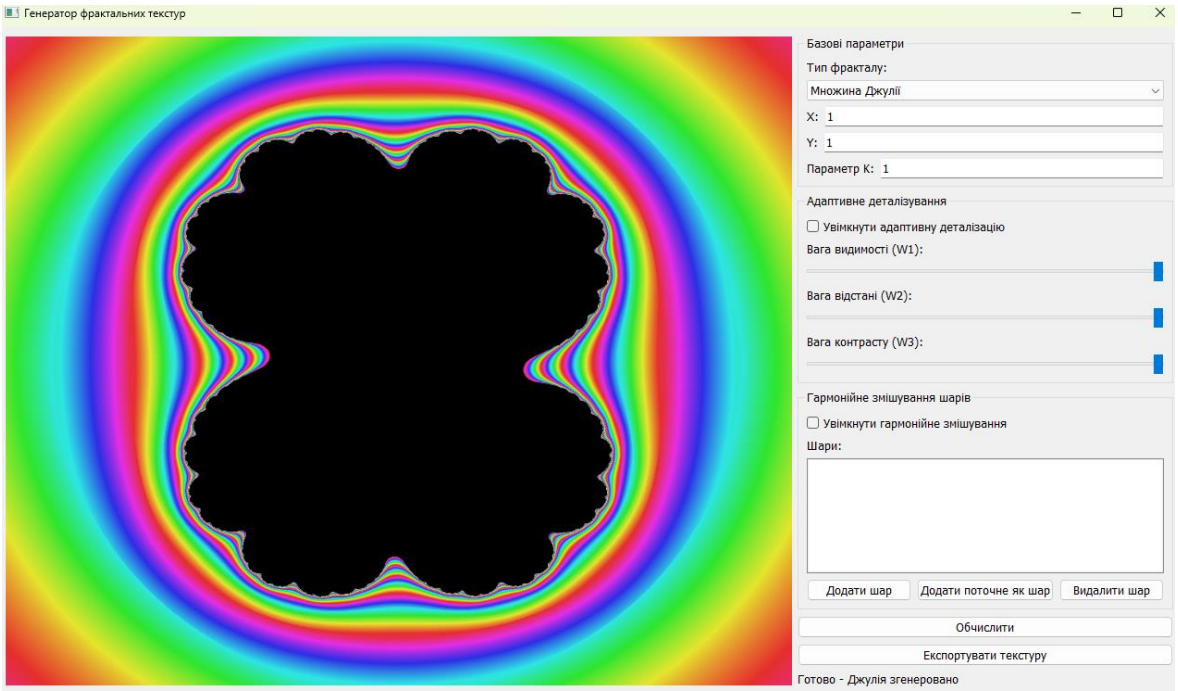


Рисунок 4.8 – Текстура за множиною Джуліа при X=1, Y=1, K=1

При зміні К=250 отримаємо зображення, як на рисунку 4.9.

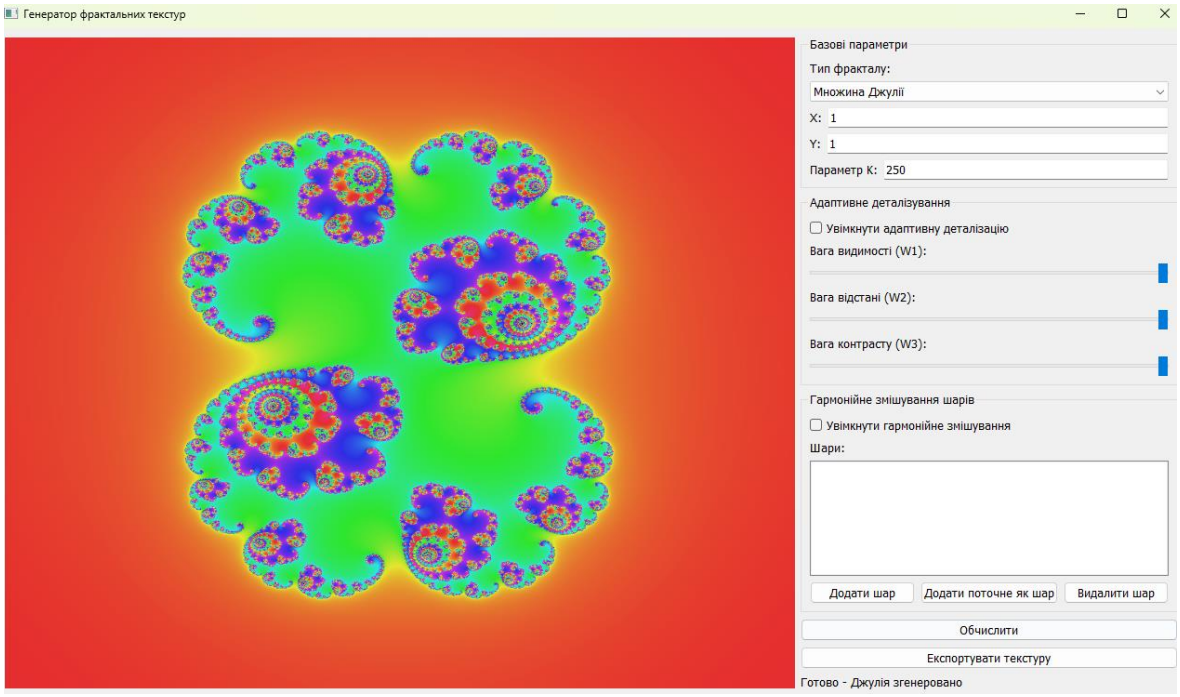


Рисунок 4.9 – Текстура за множиною Джуліа при К=250

Тепер згенеруємо фрактальну текстуру, що являє собою кола. Зображення такої текстури наведено на рисунку 4.10.

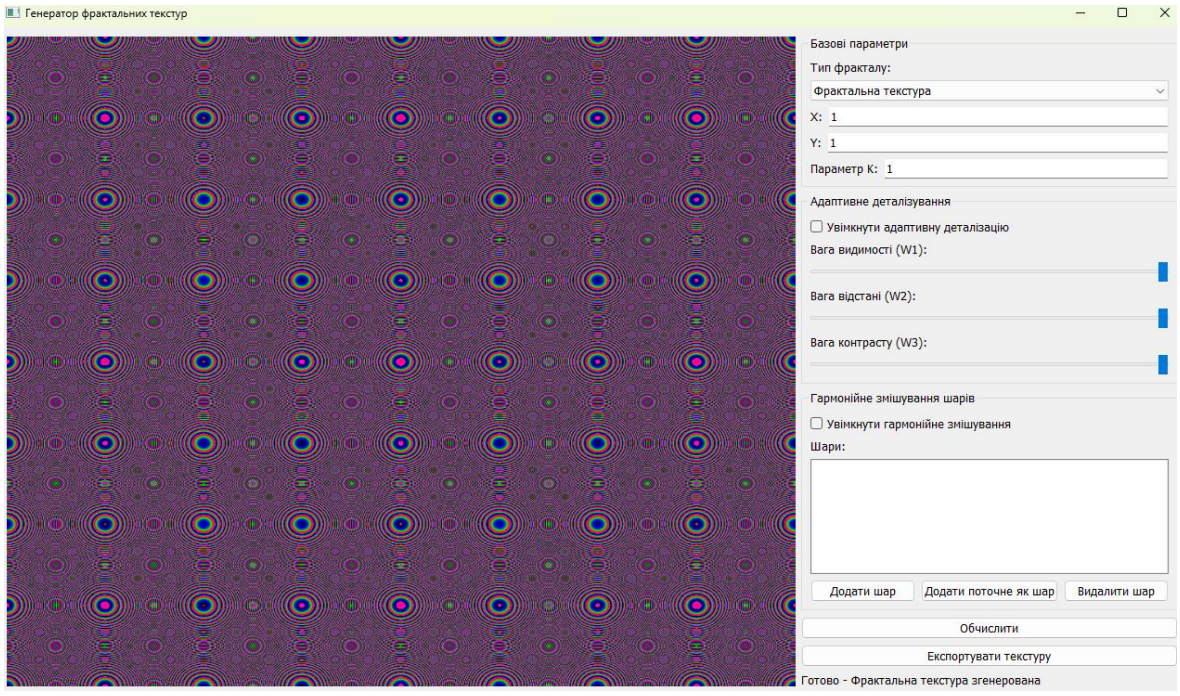


Рисунок 4.10 – Зображення фрактальної текстури

При X=50, Y=50, K=1 виходить зображення, наведене на рисунку 4.11.

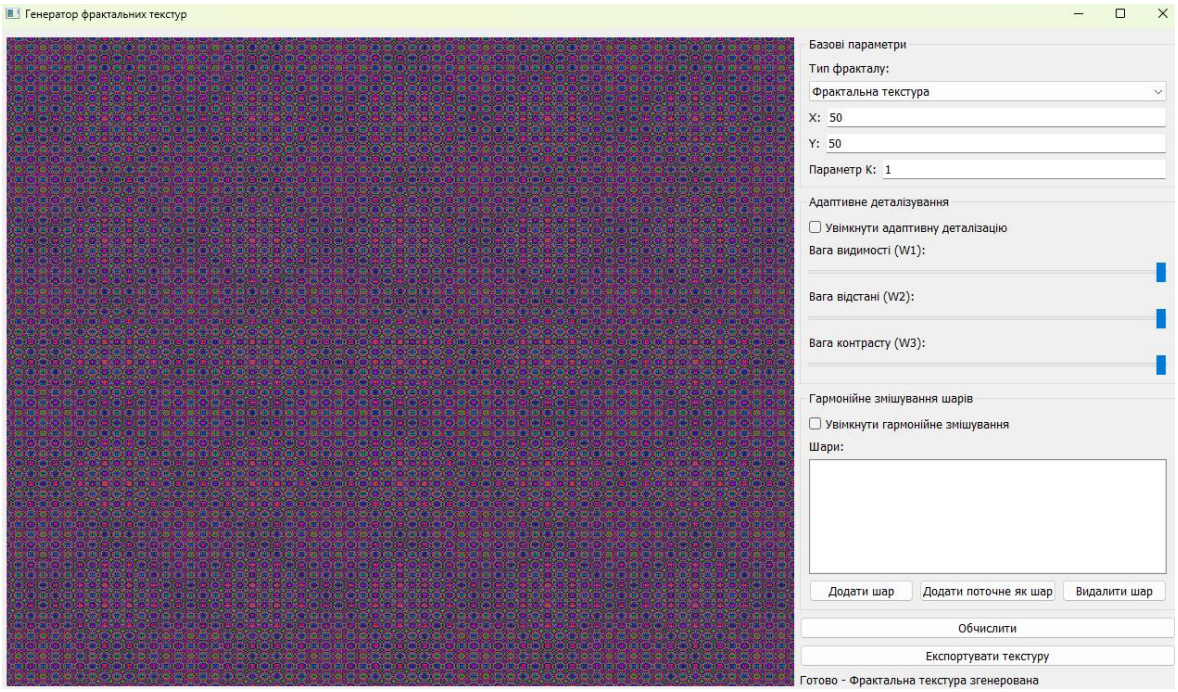


Рисунок 4.11 – Фрактальна текстура при X=50, Y=50, K=1

Застосуємо адаптивну деталізацію. Результат наведено на рисунку 4.12.

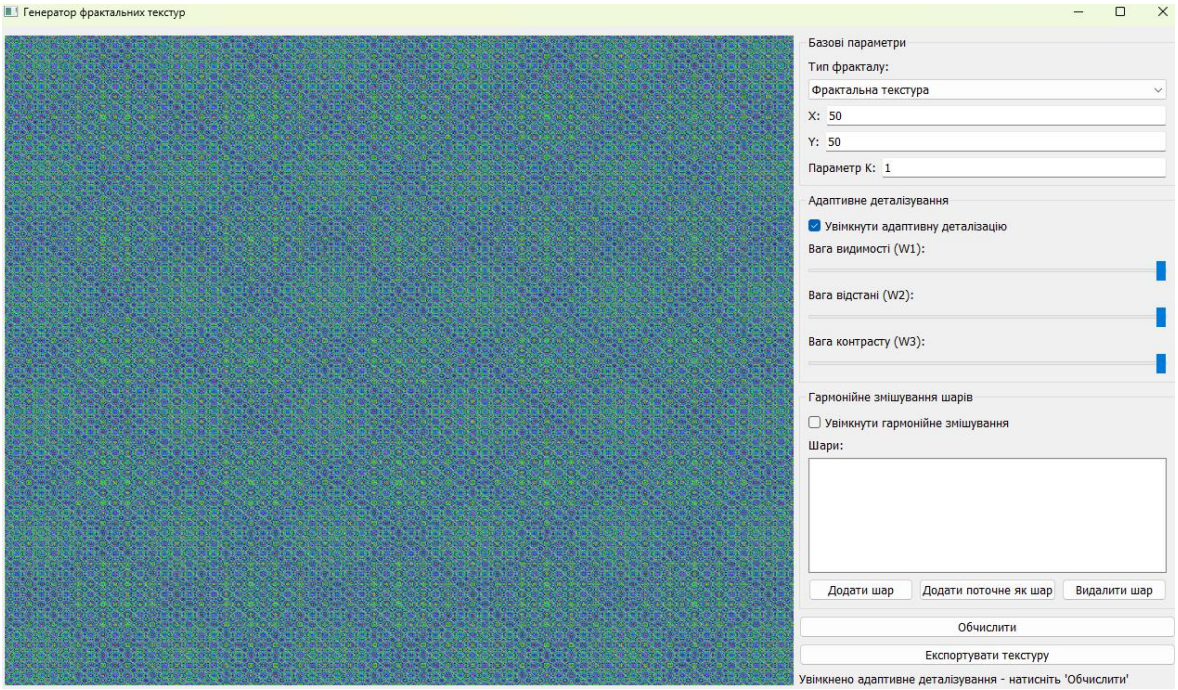


Рисунок 4.12 – Застосування адаптивної деталізації до фрактальної текстури

Змінимо ваги адаптивної деталізації. Результат наведено на рисунку 4.13.

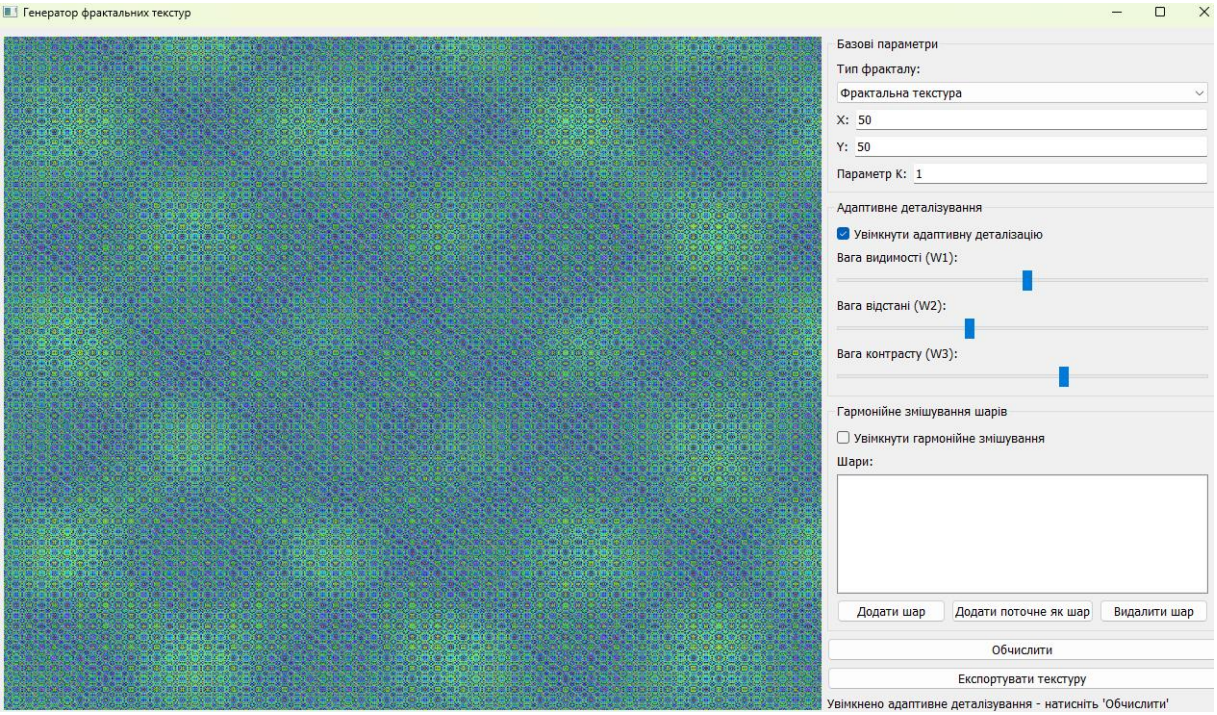


Рисунок 4.13 – Зміна зображення при зміні вагових коефіцієнтів деталізації

Перевіримо роботу гармонійного змішування шарів. Для цього активуємо відповідний прапорець, та як перший шар додамо поточний шар. Для цього натиснемо кнопку «Додати поточне як шар». Після цього з’являється діалогове вікно, яке запитує важливість шару. Цей показник впливає на видимість обраного шару на фоні інших (рисунок 4.14).

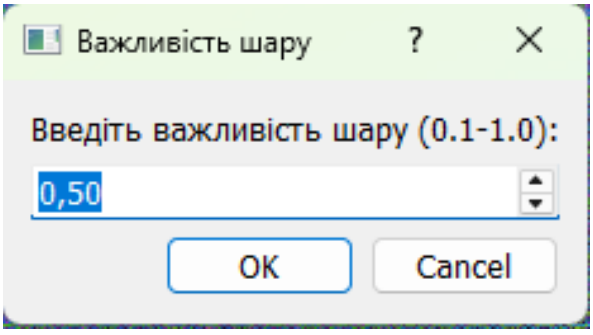


Рисунок 4.14 – Діалогове вікно вибору важливості шару

Додамо також інший шар, наприклад текстуру за множиною Мандельброта з параметрами, що наведені на рисунку 4.15.

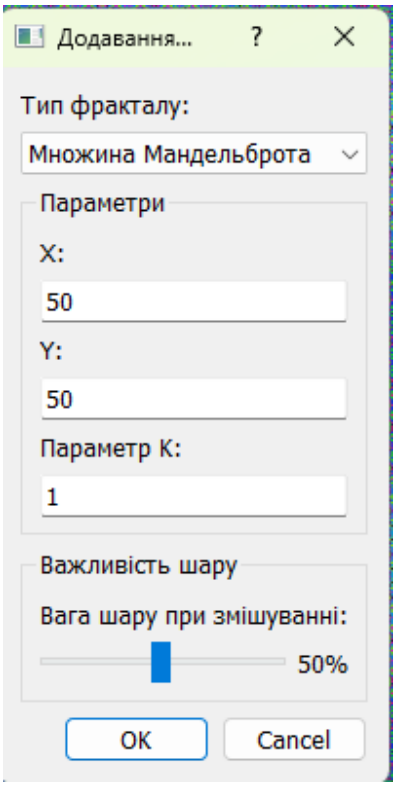


Рисунок 4.15 – Параметри для шару за множиною Мандельброта

Отримане змішане зображення наведено на рисунку 4.16.

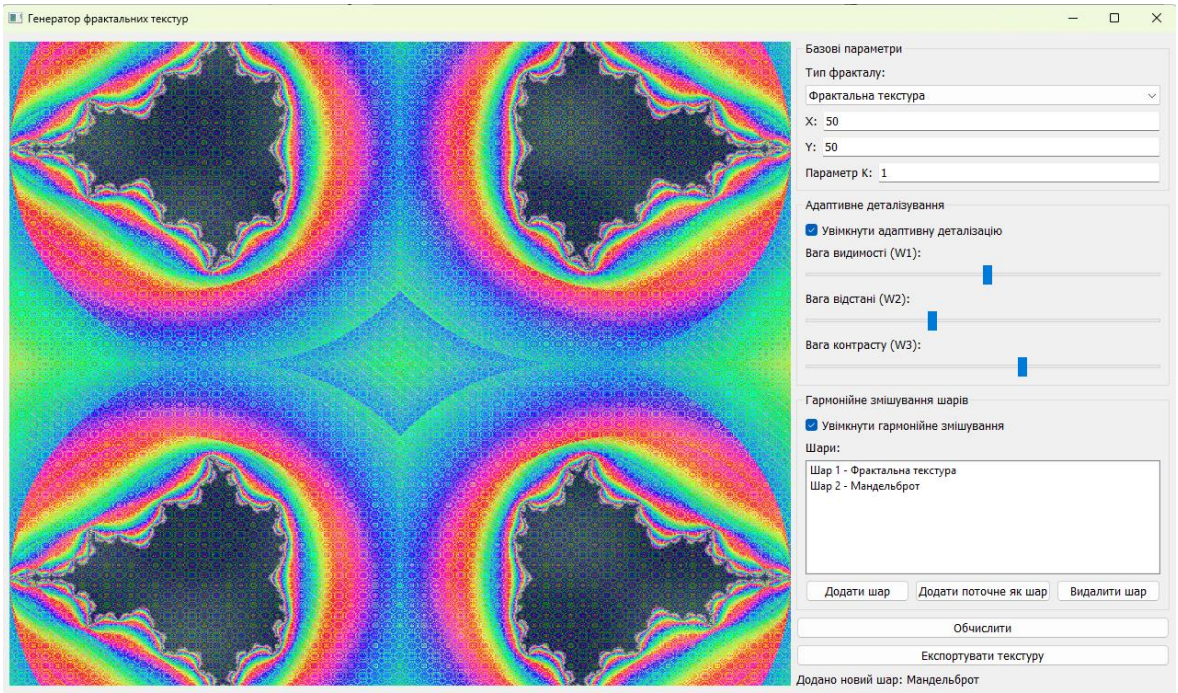


Рисунок 4.16 – Змішане зображення

Додамо аналогічним чином шар за множиною Джуліа. Діалогове вікно для введення параметрів наведено на рисунку 4.17.

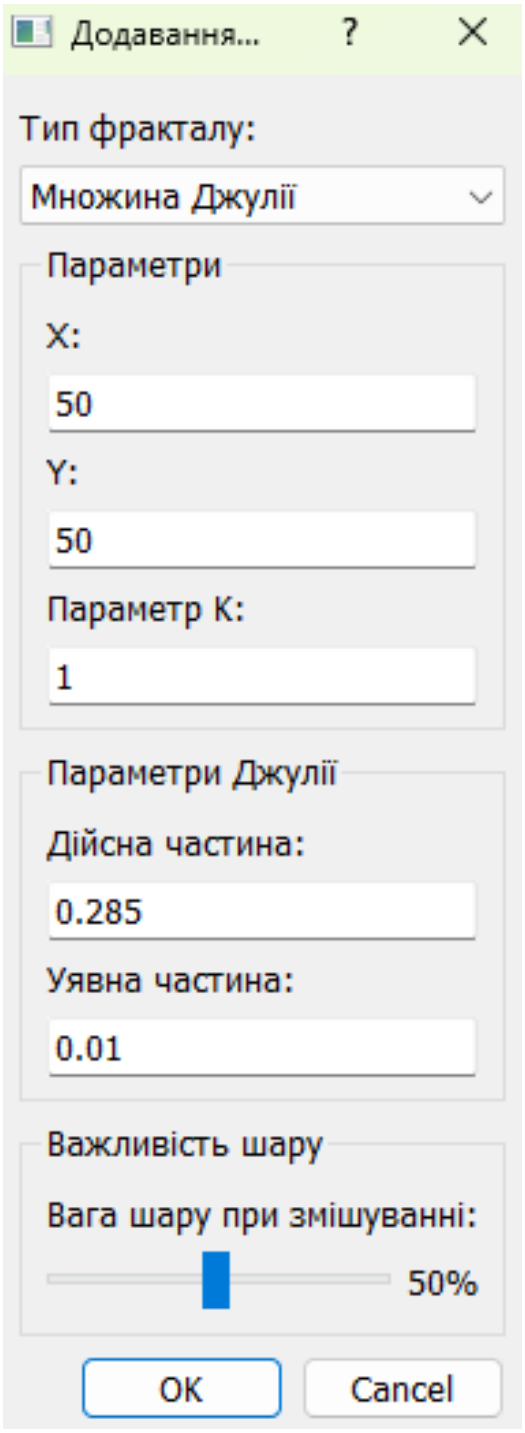


Рисунок 4.17 – Діалогове вікно додавання шару за множиною Джуліа

Як можемо побачити, це діалогове вікно відрізняється наявністю полів для введення дійсної та уявної частини.

Результат додавання шару наведено на рисунку 4.18.

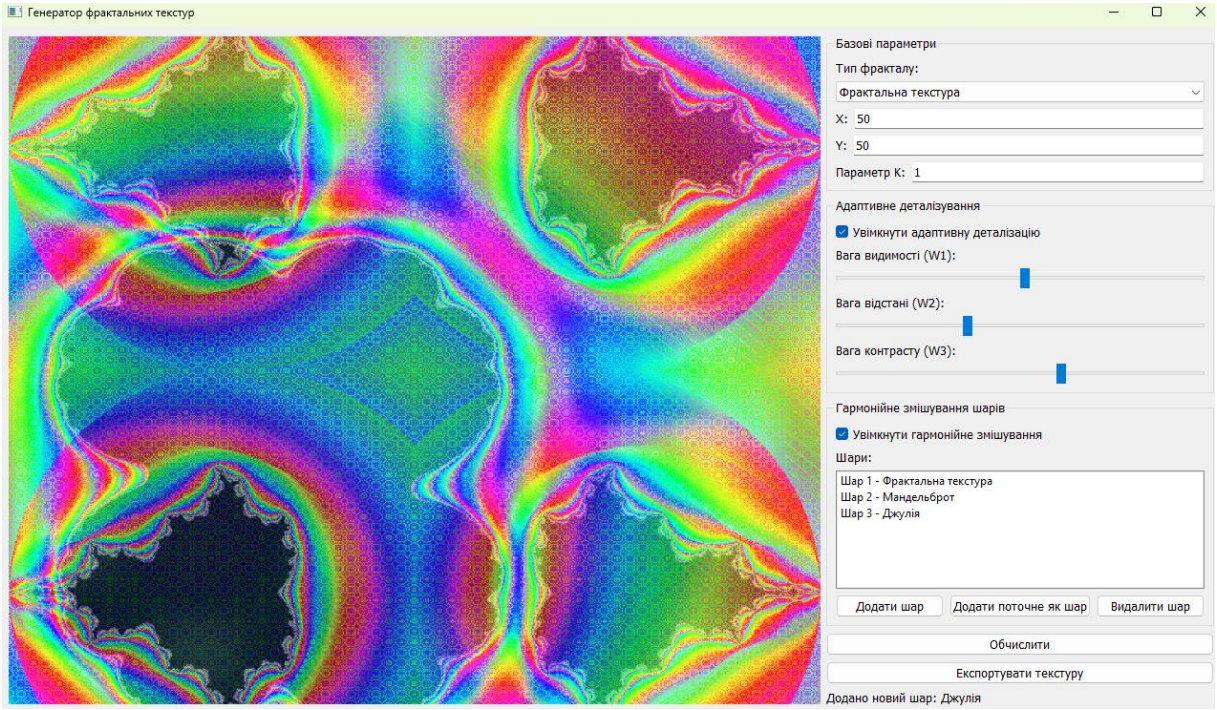


Рисунок 4.18 – Результат додавання третього шару

Спробуємо видалити шар 1. Для цього оберемо його у списку шарів та натиснемо кнопку «Видалити шар». Результат наведено на рисунку 4.19.

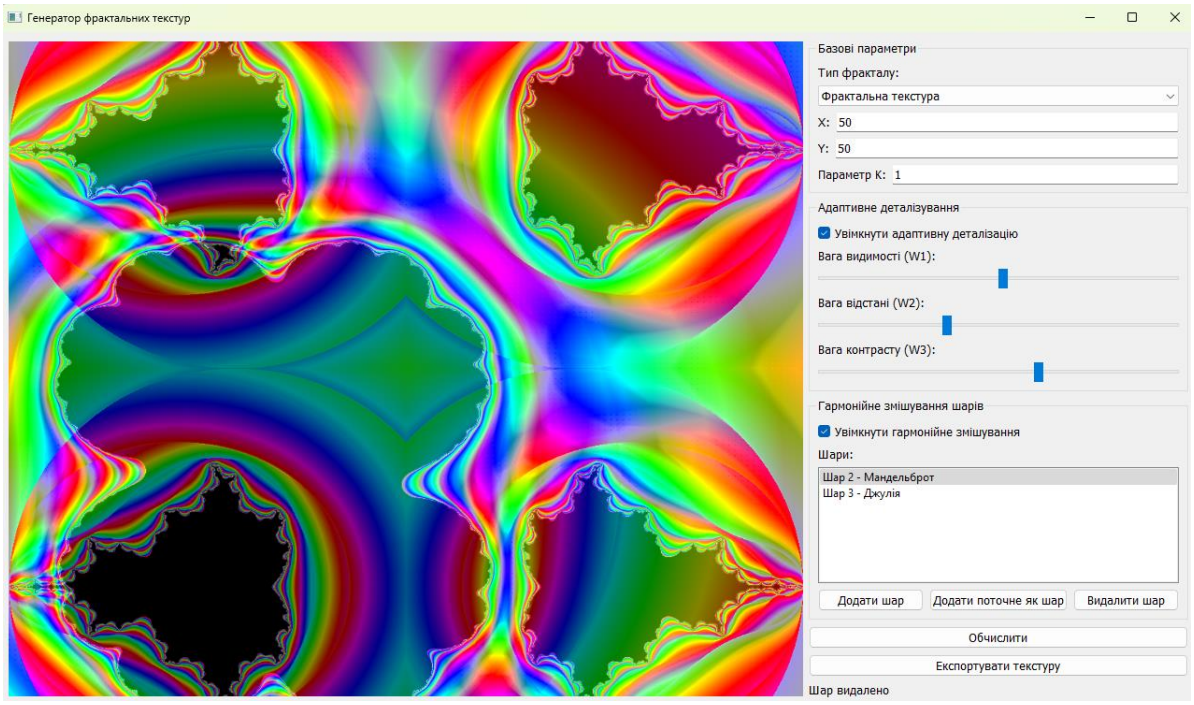


Рисунок 4.19 – Видалення шару

Спробуємо експортувати отриману текстуру. Для цього натиснемо кнопку «Експортувати текстуру», після чого оберемо папку для збереження. Отримане експортоване зображення у стандартному застосунку ОС Windows для перегляду зображень наведено на рисунку 4.20.

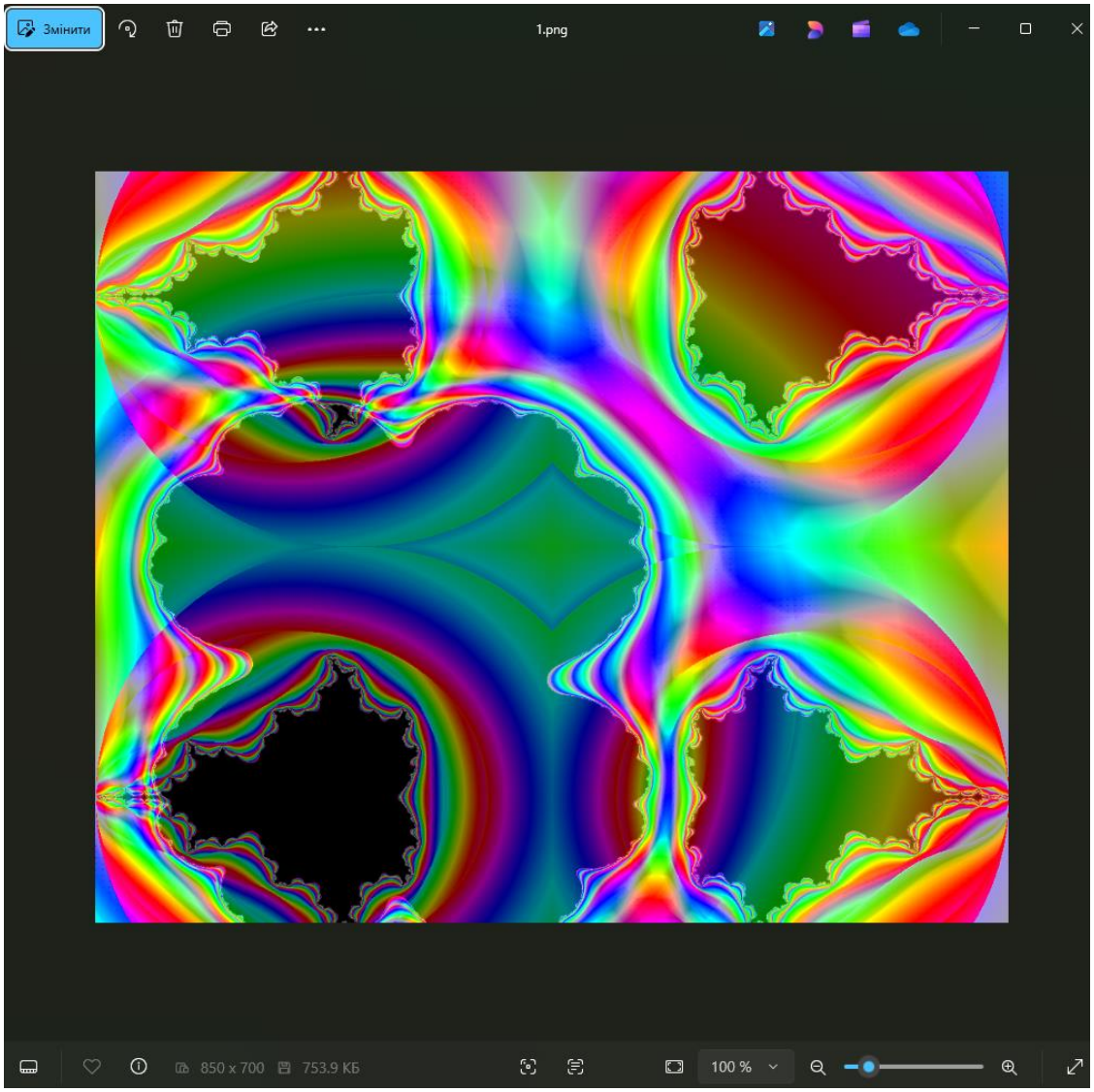


Рисунок 4.20 – Експортоване зображення

Таким чином, було проведено тестування застосунку для формування текстур з використанням фракталів. Було продемонстровано роботу усіх реалізованих функцій, та коректність їх роботи.

4.3 Висновки

У межах четвертого розділу детально розглянуто тестування застосунку для формування текстур з використанням фракталів.

У ході тестування були перевірені ключові функціональні модулі, зокрема:

- генерування текстур з використанням множини Мандельброта;
- генерування текстур з використанням множини Джуліа;
- генерування текстур з використанням фрактальної текстури:
- застосування адаптивного змішування шарів;
- застосування адаптивної деталізації фрактальних текстур;
- експорт отриманої текстури у вигляді зображення формату .png.

Тестування виконувалося на операційній системі Windows. Таким чином, можна зробити висновок, що розроблений застосунок відповідає поставленим вимогам, що були поставлені на початку розробки. Застосунок дозволяє створювати текстури з використанням фракталів, змішувати різні шари текстур, проводити деталізацію зображення залежно від потреб користувача, налаштовувати параметри текстури, експортувати отримане зображення..

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи, було розроблено програмний застосунок для формування текстур з використанням фракталів, що дозволяє створювати текстури, використовувати адаптивну деталізацію шляхом налаштування спеціальних вагових коефіцієнтів, а також формувати зображення з декількох шарів фрактальних текстур. Отримані результати можна експортувати у окремі файли для подальшого використання. Бакалаврську кваліфікаційну роботу реалізовано згідно методичних вказівок [29].

У бакалаврській кваліфікаційній роботі виконано такі завдання:

- розроблено архітектуру застосунку для формування текстур з використанням фракталів;
- розроблено інтерфейс користувача застосунку;
- розроблено метод гармонійного змішування шарів фракталів;
- розроблено метод адаптивного фрактального деталізування текстур;
- розроблено алгоритми роботи застосунку;
- розроблено функціонал застосунку;
- проведено тестування розробленого застосунку.

Для розробки було використано мову програмування Python та середовище розробки PyCharm, бібліотеки OpenGL та PyQt для реалізації інтерфейсу застосунку та реалізації шейдерів для швидкої генерації текстур.

Проведено аналіз інформаційного забезпечення застосунку, розроблено метод адаптивного фрактального деталізування текстур та метод гармонійного змішування шарів, розроблено алгоритми роботи застосунку.

Було розроблено головний модуль застосунку та модуль змішування шарів фракталів, описано алгоритми їх роботи, наведено графічні схеми інтерфейсу кожного з модулів.

Проведено тестування, продемонстровано працездатність застосунку та правильність його роботи. Було доведено, що застосунок виконує всі поставлені задачі та передбачувано реагує на дії користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Монотипія і диатипія. [Електронний ресурс]. URL: https://buki.com.ua/blogs/monotipiia-i-diatipiia-magiia-vipadkovosti-u-mistectvi/.

2. Фрактальна геометрія: навчальний посібник / Н. І. Мазуренко - Івано-Франківськ, 2010. — 65 с.

3. Ігри Хаосу. Трикутник Серпінського. Фрактали. [Електронний ресурс]. URL: mmf.lnu.edu.ua/le/ta/1966.

4. Майданюк В. П., Денисюк А. В., Магуран В.С. Формування зображень з використанням фракталів. LIV Всеукраїнська науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (2025). [Електронний ресурс]. URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025/paper/view/23513/19419

5. Perlin noise. [Електронний ресурс]. URL: https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

6. Ultra Fractal. [Електронний ресурс]. URL: https://www.ultrafractal.com/

7. Mandelbulb. [Електронний ресурс]. URL: 3D https://www.mandelbulb.com/

8. Apophysis. [Електронний ресурс]. URL: www.apophysis.org

9. Mandelbrot fractal. [Електронний ресурс]. URL: https://www.zazow.com/mandelbrot/index.php

10. Fractal Brownian Motion. [Електронний ресурс]. URL: https://thebookofshaders.com/13/

11. Fractorium. [Електронний ресурс]. URL: http://fractorium.com/

12. What is Software Development? [Електронний ресурс]. URL: https://www.ibm.com/think/topics/software-development

13. L. Wang, X. Shi and Y. Liu, "Foveated rendering: A state-of-the-art survey," in Computational Visual Media, vol. 9, no. 2, pp. 195-228, June 2023, doi: 10.1007/s41095-022-0306-4.

14. Zhou Wang, Ligang Lu and A. C. Bovik, "Foveation scalable video coding

with automatic fixation selection," in IEEE Transactions on Image Processing, vol. 12, no. 2, pp. 243-254, Feb. 2003, doi: 10.1109/TIP.2003.809015

15. Ginestet, E., Valdois, S. & Diard, J. Probabilistic modeling of orthographic learning based on visuo-attentional dynamics. Psychon Bull Rev 29, 1649–1672 (2022). https://doi.org/10.3758/s13423-021-02042-4

16. Peitgen, HO., Richter, P.H. (1986). The Mandelbrot Set. In: The Beauty of Fractals. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-61717-1_4

17. Rotation Matrix. [Електронний ресурс]. URL: https://www.cuemath.com/algebra/rotation-matrix/

18. Szaszkó, B., & Tran, U. S. (2025). The mind's abyss: exploring the adverse effects of distinct forms of meditation. Academia Mental Health and Well-Being, 2(2). https://doi.org/10.20935/MHealthWellB7666

19. Complex Modulus. [Електронний ресурс]. URL: https://mathworld.wolfram.com/ComplexModulus.html

20. George T. Heineman. Learning Algorithms: A Programmer's Guide to Writing Better Code. Фарнем: O'Reilly, 2021. 278с.

21. Python. [Електронний ресурс]. URL: https://www.python.org/

22. C++. [Електронний ресурс]. URL: https://cplusplus.com/

23. Java. [Електронний ресурс]. URL: https://www.java.com/

24. What is desktop testing? [Електронний ресурс]. URL: https://katalon.com/resources-center/blog/what-is-desktop-testing

25. Brian Okken. Python Testing with pytest: Simple, Rapid, Effective, and Scalable (2nd Edition). Нью-Йорк: The Pragmatic Bookshelf, 2022. 274с.

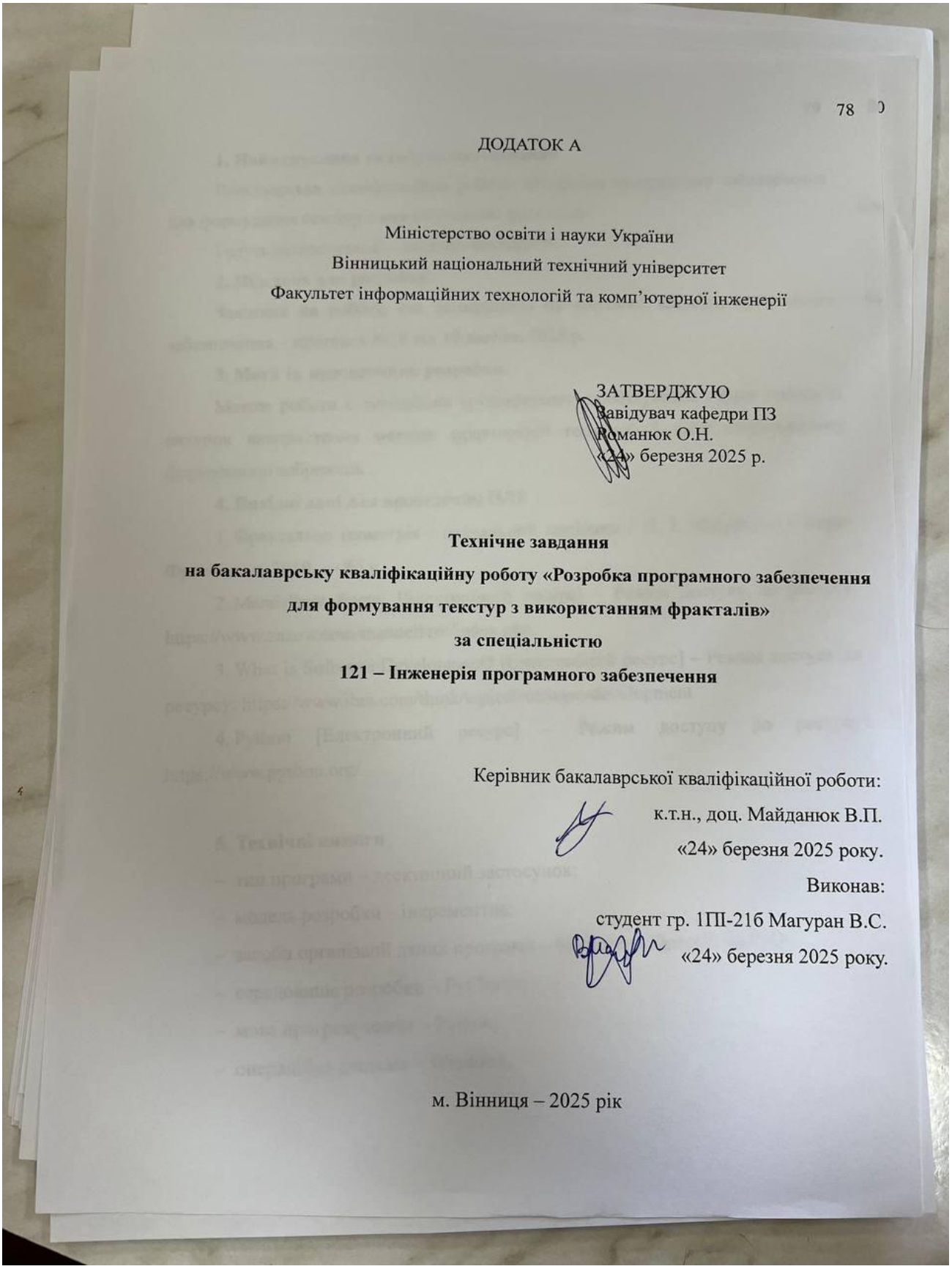
26. Sujay Raghavendra. Python Testing. Нью-Йорк: Apress, 2021. 170с.

27. Аттіла Ковач. Modern Software Testing Techniques: A Practical Guide for Developers and Testers. Нью-Йорк: Apress, 2024. 266с.

28. Анжеліна Самару. Software Testing: An ISTQB-BCS Certified Tester Foundation Level guide (CTFL v4.0) - Fifth edition. Лондон: BCS, 2024. 283с.

29. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 121 "Інженерія програмного забезпечення" / Уклад. О. Н. Романюк, Г. О. Черноволик – Вінниця: ВНТУ, 2022. – 52с.

ДОДАТКИ



78

ДОДАТОК А

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О.Н.
«24» березня 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу «Розробка програмного забезпечення для формування текстур з використанням фракталів»
за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
к.т.н., доц. Майданюк В.П.
«24» березня 2025 року.

Виконав:
студент гр. 1ПІ-21б Магуран В.С.
«24» березня 2025 року.

м. Вінниця – 2025 рік

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка програмного забезпечення для формування текстур з використанням фракталів».

Галузь застосування – десктоп-технології.

2. Підстава для розробки.

Завдання на роботу, яке затверджене на засіданні кафедри програмного забезпечення – протокол №18 від 18 лютого 2025 р.

3. Мета та призначення розробки.

Метою роботи є зменшення трудомісткості проектування узорів тканин за рахунок використання методів фрактальної геометрії і комп'ютеризованому формуванню зображень .

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Фрактальна геометрія : навчальний посібник / Н. І. Мазуренко - Івано-Франківськ, 2010. — 65 с.

2. Mandelbrot fractal [Електронний ресурс] – Режим доступу до ресурсу: https://www.zazow.com/mandelbrot/index.php

3. What is Software Development? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/think/topics/software-development

4. Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/

5. Технічні вимоги

- тип програми – десктопний застосунок;
- модель розробки – інкрементна;
- засоби організації даних програми – бібліотеки OpenGL та PyQt;
- середовище розробки – PyCharm;
- мова програмування – Python;
- операційна система – Windows.

6. Конструктивні вимоги.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до бакалаврської кваліфікаційної роботи;

– технічне завдання;

– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

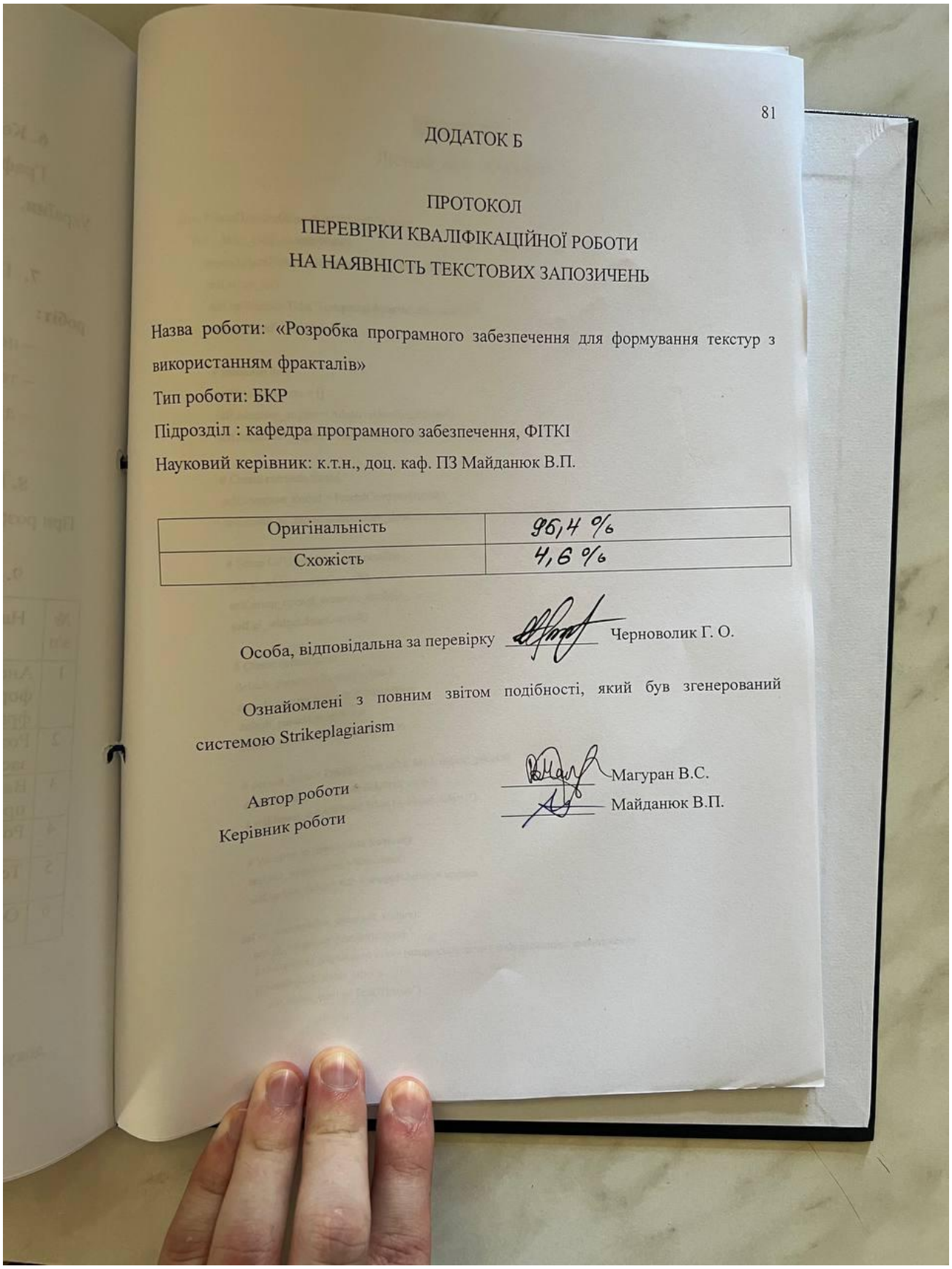
При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз стану розвитку застосунків для формування текстур з використанням фракталів	25.03.25 - 08.04.25
2	Розробка методів та алгоритмів програмного застосунку	09.04.25 - 20.04.25
3	Варіантний аналіз та вибір мови програмування розробки	21.04.25 - 23.04.25
4	Розробка застосунку	24.04.25 - 20.05.25
5	Тестування застосунку	21.05.25 - 25.05.25
6	Оформлення матеріалів до захисту БКР	26.05.25 - 30.05.25

10. Порядок контролю та прийняття.

Порядок контролю і приймання роботи регламентується відповідними документами ВНТУ і державними стандартами.



81

ДОДАТОК Б

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: «Розробка програмного забезпечення для формування текстур з використанням фракталів»

Тип роботи: БКР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: к.т.н., доц. каф. ПЗ Майданюк В.П.

Оригінальність	95,4 %
Схожість	4,6 %

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Strikeplagiarism

Автор роботи _____ Магуран В.С.

Керівник роботи _____ Майданюк В.П.

ДОДАТОК В

Лістинг коду програми

```
class FractalTextureGenerator(QMainWindow):
    def __init__(self, parent=None):
        super(FractalTextureGenerator, self).__init__(parent)
        self.setup_ui()
        self.setWindowTitle("Генератор фрактальних текстур")
        self.setFixedSize(1280, 720)

        # Initialize data
        self.fractal_layers = []
        self.adaptive_engine = AdaptiveDetailingEngine()
        self.harmonic_engine = HarmonicLayerBlendingEngine()

        # Create compute thread
        self.compute_thread = FractalComputeThread()
        self.compute_thread.computation_done.connect(self.on_computation_done)

        # Setup GPU acceleration if possible
        self.gl_widget.makeCurrent()
        self.setup_opengl_compute_shader()
        self.gl_widget.doneCurrent()

        # Create default layer
        default_params = FractalParams()
        default_params.type = 0  # Mandelbrot
        default_params.zoom = 1.0
        default_params.baseIterations = 100

        # default_layer = FractalLayer(1024, 1024, default_params)
        # self.fractal_layers.append(default_layer)
        # self.layers_list.addItem("Шар 1 - Мандельброт")

        # Variables to limit update frequency
        self.last_update_time = time.time()
        self.update_delay = 0.2  # seconds between updates

    def on_computation_done(self, texture):
        self.gl_widget.set_texture(texture)
        # Оновлення візуального стану (наприклад, деактивація індикатора завантаження)
        if hasattr(self, 'status_label'):
            self.status_label.setText("Готово")
```

```
def on_slider_changed(self):
    # Встановлюємо статус "Змінено"
    if hasattr(self, 'status_label'):
        self.status_label.setText("Змінено - відпустіть для оновлення")

def closeEvent(self, event):
    # Спочатку зупиняємо потік
    self.compute_thread.stop()

    # Вивільняємо ресурси OpenGL, якщо необхідно
    if hasattr(self, 'gl_widget') and self.gl_widget:
        self.gl_widget.makeCurrent()
        # Тут можна додати код для вивільнення ресурсів OpenGL
        self.gl_widget.doneCurrent()

    event.accept()

def setup_ui(self):
    # Основний віджет
    central_widget = QWidget(self)
    self.setCentralWidget(central_widget)

    # Головний layout
    main_layout = QHBoxLayout(central_widget)

    # OpenGL віджет
    self.gl_widget = GLWidget(self)
    main_layout.addWidget(self.gl_widget, 3)

    # Панель керування
    control_layout = QVBoxLayout()
    main_layout.addLayout(control_layout, 1)

    self.setFixedSize(1280, 720)  # або будь-який інший бажаний розмір
    self.gl_widget.setMinimumSize(850, 700)
    self.gl_widget.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)

    # Група базових параметрів
    basic_group = QGroupBox("Базові параметри")
    basic_layout = QVBoxLayout(basic_group)

    self.fractal_type_combo = QComboBox()
    self.fractal_type_combo.addItem("Множина Мандельброта")
    self.fractal_type_combo.addItem("Множина Джулії")
    self.fractal_type_combo.addItem("Фрактальна текстура")
    basic_layout.addWidget(QLabel("Тип фракталу:"))
```

```
basic_layout.addWidget(self.fractal_type_combo)

# Параметри для фрактальної текстури
a_layout = QHBoxLayout()
a_layout.addWidget(QLabel("Параметр A:"))
self.a_param_edit = QLineEdit("1")
a_layout.addWidget(self.a_param_edit)
basic_layout.addLayout(a_layout)

b_layout = QHBoxLayout()
b_layout.addWidget(QLabel("Параметр B:"))
self.b_param_edit = QLineEdit("1")
b_layout.addWidget(self.b_param_edit)
basic_layout.addLayout(b_layout)

k_layout = QHBoxLayout()
k_layout.addWidget(QLabel("Параметр K:"))
self.k_param_edit = QLineEdit("1")
k_layout.addWidget(self.k_param_edit)
basic_layout.addLayout(k_layout)

control_layout.addWidget(basic_group)

# Група для методу АФДТ
adaptive_group = QGroupBox("Адаптивне деталізування")
adaptive_layout = QVBoxLayout(adaptive_group)

self.adaptive_detailing_checkbox = QCheckBox("Увімкнути адаптивну деталізацію")
adaptive_layout.addWidget(self.adaptive_detailing_checkbox)

self.w1_slider = QSlider(Qt.Horizontal)
self.w1_slider.setRange(1, 100)
self.w1_slider.setValue(50)
adaptive_layout.addWidget(QLabel("Вага видимості (W1):"))
adaptive_layout.addWidget(self.w1_slider)

self.w2_slider = QSlider(Qt.Horizontal)
self.w2_slider.setRange(1, 100)
self.w2_slider.setValue(30)
adaptive_layout.addWidget(QLabel("Вага відстані (W2):"))
adaptive_layout.addWidget(self.w2_slider)

self.w3_slider = QSlider(Qt.Horizontal)
self.w3_slider.setRange(1, 100)
self.w3_slider.setValue(20)
adaptive_layout.addWidget(QLabel("Вага контрасту (W3):"))
```

```
adaptive_layout.addWidget(self.w3_slider)

control_layout.addWidget(adaptive_group)

# Група для методу ГЗШФ
harmonic_group = QGroupBox("Гармонійне змішування шарів")
harmonic_layout = QVBoxLayout(harmonic_group)

self.harmonic_blending_checkbox = QCheckBox("Увімкнути гармонійне змішування")
harmonic_layout.addWidget(self.harmonic_blending_checkbox)

self.layers_list = QListWidget()
harmonic_layout.addWidget(QLabel("Шари:"))
harmonic_layout.addWidget(self.layers_list)

layer_buttons_layout = QHBoxLayout()
self.add_layer_button = QPushButton("Додати шар")
self.add_current_as_layer_button = QPushButton("Додати поточне як шар")
self.remove_layer_button = QPushButton("Видалити шар")
layer_buttons_layout.addWidget(self.add_layer_button)
layer_buttons_layout.addWidget(self.add_current_as_layer_button)
layer_buttons_layout.addWidget(self.remove_layer_button)
harmonic_layout.addLayout(layer_buttons_layout)

control_layout.addWidget(harmonic_group)

# Кнопка для запуску обчислень
calculate_layout = QHBoxLayout()
self.calculate_button = QPushButton("Обчислити")
self.calculate_button.setIcon(QIcon.fromTheme("system-run"))
self.calculate_button.setIconSize(QSize(24, 24))
calculate_layout.addWidget(self.calculate_button)

# Прапорець для перерахунку без кешу
self.force_recalculate_checkbox = QCheckBox("Ігнорувати кеш")
calculate_layout.addWidget(self.force_recalculate_checkbox)
control_layout.addLayout(calculate_layout)

# Експорт
export_layout = QHBoxLayout()
self.export_button = QPushButton("Експортувати текстуру")
self.export_button.setIcon(QIcon.fromTheme("document-save"))
self.export_button.setIconSize(QSize(24, 24))
export_layout.addWidget(self.export_button)
control_layout.addLayout(export_layout)
```

```
# Індикатор статусу
self.status_label = QLabel("Готово")
control_layout.addWidget(self.status_label)

# Порожній простір унизу
control_layout.addStretch()

# Підключення сигналів
self.fractal_type_combo.currentIndexChanged.connect(self.on_fractal_type_changed)
self.adaptive_detailing_checkbox.toggled.connect(self.on_adaptive_detailing_toggled)
self.harmonic_blending_checkbox.toggled.connect(self.on_harmonic_blending_toggled)

# В методі setup_ui, після додавання кнопок:
self.add_layer_button.clicked.connect(self.add_layer)
self.add_current_as_layer_button.clicked.connect(self.add_current_texture_as_layer)  # Якщо ви додали цю кнопку
self.remove_layer_button.clicked.connect(self.remove_layer)

# Remove these connections that trigger automatic updates
# self.w1_slider.valueChanged.connect(self.update_weights)
# self.w2_slider.valueChanged.connect(self.update_weights)
# self.w3_slider.valueChanged.connect(self.update_weights)

# Replace with connections that only update the status
self.w1_slider.valueChanged.connect(
    lambda: self.status_label.setText("Змінено вагові коефіцієнти - натисніть 'Обчислити'"))
self.w2_slider.valueChanged.connect(
    lambda: self.status_label.setText("Змінено вагові коефіцієнти - натисніть 'Обчислити'"))
self.w3_slider.valueChanged.connect(
    lambda: self.status_label.setText("Змінено вагові коефіцієнти - натисніть 'Обчислити'"))

# Also adjust iteration slider connections
# self.iterations_slider.valueChanged.connect(self.update_slider_labels)  # Keep this for updating labels
# self.iterations_slider.valueChanged.connect(
#     lambda: self.status_label.setText("Змінено ітерації - натисніть 'Обчислити'"))

# Слайдери з обробкою завершення перетягування
# self.iterations_slider.sliderReleased.connect(self.on_slider_released)
# self.iterations_slider.valueChanged.connect(self.update_slider_labels)

# Обробники змін для відображення статусу
# self.iterations_slider.valueChanged.connect(self.on_slider_changed)

# Параметри фрактальної текстури
self.a_param_edit.textChanged.connect(self.on_fractal_texture_params_changed)
self.b_param_edit.textChanged.connect(self.on_fractal_texture_params_changed)
self.k_param_edit.textChanged.connect(self.on_fractal_texture_params_changed)
```

```
    # Підключення кнопок
    # self.add_layer_button.clicked.connect(self.add_layer)
    # self.remove_layer_button.clicked.connect(self.remove_layer)
    self.calculate_button.clicked.connect(self.on_calculate_button_clicked)
    self.export_button.clicked.connect(self.export_texture)

    # Початкове оновлення інтерфейсу на основі обраного типу фракталу
    self.on_fractal_type_changed()

# 2. Add a GPU accelerated version of the fractal generation using OpenGL shaders

def setup_opengl_compute_shader(self):
    """
    Setup OpenGL compute shader for GPU-accelerated fractal generation
    """
    # Compute shader for Mandelbrot/Julia set generation
    compute_shader_source = """
    #version 430
    layout(local_size_x = 16, local_size_y = 16) in;
    layout(rgba32f, binding = 0) uniform image2D outputTexture;

    uniform int fractalType;      // 0 for Mandelbrot, 1 for Julia
    uniform float zoom;
    uniform int maxIterations;
    uniform vec2 center;
    uniform vec2 juliaConstant;
    uniform vec4 color1;
    uniform vec4 color2;
    uniform float adaptiveWeight; // 0.0 for no adaptive detailing, > 0.0 for adaptive

    // For adaptive detailing
    uniform vec2 focusPoint;
    uniform float focusRadius;
    uniform float visibilityWeight;
    uniform float distanceWeight;
    uniform float contrastWeight;

    void main() {
        // Get pixel coordinates
        ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
        ivec2 imageSize = imageSize(outputTexture);

        if (pixelCoords.x >= imageSize.x || pixelCoords.y >= imageSize.y) {
            return;
        }
```

```
// Normalize coordinates
vec2 normalizedCoords = vec2(float(pixelCoords.x) / float(imageSize.x),
                             float(pixelCoords.y) / float(imageSize.y));

// Map to complex plane
vec2 scaledCoords = (normalizedCoords - 0.5) * 4.0 / zoom + center;

// Initialize variables for iteration
vec2 z = vec2(0.0, 0.0);
vec2 c;

// Set c based on fractal type
if (fractalType == 0) { // Mandelbrot
    c = scaledCoords;
} else { // Julia
    z = scaledCoords;
    c = juliaConstant;
}

// Determine iterations based on adaptive detailing if enabled
int iterations = maxIterations;

if (adaptiveWeight > 0.0) {
    // Calculate distance from focus point
    float dist = distance(normalizedCoords, focusPoint);
    float normalizedDist = min(1.0, dist / focusRadius);
    float distFactor = 1.0 - normalizedDist;

    // Simple visibility factor (1.0 by default)
    float visibilityFactor = 1.0;

    // Use a fixed contrast factor for now (would need a pre-pass for real contrast)
    float contrastFactor = 0.5;

    // Calculate importance
    float importance = visibilityWeight * visibilityFactor +
                       distanceWeight * distFactor +
                       contrastWeight * contrastFactor;

    // Clamp importance
    importance = clamp(importance, 0.0, 1.0);

    // Adjust iterations based on importance
    // Lower importance = fewer iterations to save computation
    iterations = int(float(maxIterations) * (0.2 + 0.8 * importance));
```

```
    }

    // Iterate fractal formula
    int i = 0;
    float zMagSq = 0.0;

    for(i = 0; i < iterations && zMagSq < 4.0; i++) {
        // z = z^2 + c
        float zr = z.x * z.x - z.y * z.y + c.x;
        float zi = 2.0 * z.x * z.y + c.y;
        z = vec2(zr, zi);
        zMagSq = z.x * z.x + z.y * z.y;
    }

    // Color mapping
    vec4 pixelColor;

    if (i == iterations) {
        // Point is in the set
        pixelColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        // Smooth coloring
        float smoothColor = float(i) - log2(log2(zMagSq)) + 4.0;
        smoothColor = smoothColor / float(iterations);

        // Mix colors
        pixelColor = mix(color1, color2, smoothColor);
    }

    // Write to output texture
    imageStore(outputTexture, pixelCoords, pixelColor);
}
"""

# Compile shader
try:
    self.compute_shader = shaders.compileShader(compute_shader_source, GL_COMPUTE_SHADER)
    self.compute_program = shaders.compileProgram(self.compute_shader)

    # Create output texture for compute shader
    self.compute_texture = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, self.compute_texture)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, 1024, 1024, 0, GL_RGBA, GL_FLOAT, None)
    glBindTexture(GL_TEXTURE_2D, 0)
```

```
            # Flag that GPU acceleration is available
            self.gpu_acceleration_available = True
        except Exception as e:
            print(f"Failed to setup GPU acceleration: {str(e)}")
            self.gpu_acceleration_available = False

    def generate_fractal_gpu(self, width, height, params, use_adaptive):
        """
        Generate fractal using GPU acceleration
        """
        if not self.gpu_acceleration_available:
            # Fall back to CPU implementation if GPU is not available
            if use_adaptive:
                return self.generate_optimized_adaptive_texture(width, height, params, Scene())
            else:
                return self.generate_regular_texture(width, height, params)

        # Create result texture
        result = Texture(width, height)

        # Prepare for compute shader
        glUseProgram(self.compute_program)

        # Set uniforms
        glUniform1i(glGetUniformLocation(self.compute_program, "fractalType"), params.type)
        glUniform1f(glGetUniformLocation(self.compute_program, "zoom"), params.zoom)
        glUniform1i(glGetUniformLocation(self.compute_program, "maxIterations"), params.baseIterations)
        glUniform2f(glGetUniformLocation(self.compute_program, "center"), params.center[0], params.center[1])
        glUniform2f(glGetUniformLocation(self.compute_program, "juliaConstant"),
                    params.juliaConstant.real, params.juliaConstant.imag)

        # Set colors
        glUniform4f(glGetUniformLocation(self.compute_program, "color1"),
                    params.color1.r, params.color1.g, params.color1.b, params.color1.a)
        glUniform4f(glGetUniformLocation(self.compute_program, "color2"),
                    params.color2.r, params.color2.g, params.color2.b, params.color2.a)

        # Set adaptive detailing parameters
        if use_adaptive:
            glUniform1f(glGetUniformLocation(self.compute_program, "adaptiveWeight"), 1.0)

            # Get scene details
            scene = Scene()
            focus_x, focus_y = scene.get_focus_point()
            focus_radius = scene.get_focus_radius()
```

```
        glUniform2f(glGetUniformLocation(self.compute_program, "focusPoint"), focus_x, focus_y)
        glUniform1f(glGetUniformLocation(self.compute_program, "focusRadius"), focus_radius)

        # Get weights from sliders
        w1 = self.w1_slider.value() / 100.0
        w2 = self.w2_slider.value() / 100.0
        w3 = self.w3_slider.value() / 100.0

        # Normalize weights
        total = w1 + w2 + w3
        w1 /= total
        w2 /= total
        w3 /= total

        glUniform1f(glGetUniformLocation(self.compute_program, "visibilityWeight"), w1)
        glUniform1f(glGetUniformLocation(self.compute_program, "distanceWeight"), w2)
        glUniform1f(glGetUniformLocation(self.compute_program, "contrastWeight"), w3)
    else:
        glUniform1f(glGetUniformLocation(self.compute_program, "adaptiveWeight"), 0.0)

    # Bind output texture
    glBindImageTexture(0, self.compute_texture, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F)

    # Show progress dialog
    progress = QProgressDialog("Генерація фракталу на GPU...", "Скасувати", 0, 100, self)
    progress.setWindowTitle("Обчислення")
    progress.setWindowModality(Qt.WindowModal)
    progress.show()

    # Dispatch compute shader
    work_group_size = 16
    num_work_groups_x = (width + work_group_size - 1) // work_group_size
    num_work_groups_y = (height + work_group_size - 1) // work_group_size

    # Set progress to 50%
    progress.setValue(50)
    QApplication.processEvents()

    # Run compute shader
    glDispatchCompute(num_work_groups_x, num_work_groups_y, 1)

    # Make sure the computation is finished before reading results
    glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT)

    # Read back the computed texture
```

```
    glBindTexture(GL_TEXTURE_2D, self.compute_texture)
    data = glGetTexImage(GL_TEXTURE_2D, 0, GL_RGBA, GL_FLOAT)

    # Set progress to 90%
    progress.setValue(90)
    QApplication.processEvents()

    # Copy data to result texture
    result.pixels = np.array(data).reshape(height, width, 4)

    # Clean up
    glBindTexture(GL_TEXTURE_2D, 0)
    glUseProgram(0)

    # Complete progress
    progress.setValue(100)

    return result

def update_slider_labels(self):
    # Оновлення міток для слайдерів
    # self.zoom_label.setText(f"{self.zoom_slider.value() / 10.0:.1f}")
    self.iterations_label.setText(str(self.iterations_slider.value()))

def on_fractal_type_changed(self):
    fractal_type = self.fractal_type_combo.currentIndex()

    # Показуємо/приховуємо відповідні елементи інтерфейсу
    self.a_param_edit.setVisible(fractal_type == 2)
    self.b_param_edit.setVisible(fractal_type == 2)
    self.k_param_edit.setVisible(fractal_type == 2)

    # Показуємо/приховуємо елементи для звичайних фракталів
    iterations_visible = fractal_type < 2
    self.iterations_slider.setVisible(iterations_visible)
    if hasattr(self, 'iterations_label'):
        self.iterations_label.setVisible(iterations_visible)

    # Оновлюємо статус
    if hasattr(self, 'status_label'):
        if fractal_type == 0:
            self.status_label.setText("Тип фракталу: Множина Мандельброта - натисніть 'Обчислити'")
        elif fractal_type == 1:
            self.status_label.setText("Тип фракталу: Множина Джулії - натисніть 'Обчислити'")
        else:
            self.status_label.setText("Тип фракталу: Фрактальна текстура - натисніть 'Обчислити'")
```

```
    # Видаляємо автоматичне оновлення
    # self.update_fractal()

def on_adaptive_detailing_toggled(self, checked):
    # Оновлюємо статус для користувача
    if hasattr(self, 'status_label'):
        self.status_label.setText(
            f"{'Увімкнено' if checked else 'Вимкнено'} адаптивне деталізування - натисніть 'Обчислити'")

    # Активуємо або деактивуємо повзунки для вагових коефіцієнтів
    self.w1_slider.setEnabled(checked)
    self.w2_slider.setEnabled(checked)
    self.w3_slider.setEnabled(checked)

def on_harmonic_blending_toggled(self, checked):
    """
    Обробник зміни стану прапорця гармонійного змішування.
    """
    # Активуємо/деактивуємо елементи інтерфейсу
    self.layers_list.setEnabled(checked)
    self.add_layer_button.setEnabled(checked)
    self.remove_layer_button.setEnabled(checked)

    if checked and self.fractal_layers:
        # Якщо увімкнено і є шари - застосовуємо змішування
        self.update_blended_texture()
        self.status_label.setText("Гармонійне змішування увімкнено")
    elif checked:
        self.status_label.setText("Гармонійне змішування увімкнено. Додайте шари для змішування.")
    else:
        # Показуємо оригінальне зображення (відтворюємо останній фрактал)
        self.on_calculate_button_clicked()
        self.status_label.setText("Гармонійне змішування вимкнено")

def on_mode_changed(self):
    # Оновлюємо видимість елементів залежно від обраного режиму
    fractal_mode = self.fractal_mode_radio.isChecked()
    texture_mode = self.texture_mode_radio.isChecked()
    fractal_texture_mode = self.fractal_texture_radio.isChecked()

    # Загальні елементи
    self.export_button.setEnabled(True)

    # Елементи для режиму фракталу
    self.fractal_type_combo.setEnabled(fractal_mode)
```

```
    # self.zoom_slider.setEnabled(fractal_mode)
    self.iterations_slider.setEnabled(fractal_mode)
    self.adaptive_detailing_checkbox.setEnabled(fractal_mode)
    self.w1_slider.setEnabled(fractal_mode and self.adaptive_detailing_checkbox.isChecked())
    self.w2_slider.setEnabled(fractal_mode and self.adaptive_detailing_checkbox.isChecked())
    self.w3_slider.setEnabled(fractal_mode and self.adaptive_detailing_checkbox.isChecked())
    self.harmonic_blending_checkbox.setEnabled(fractal_mode)
    self.layers_list.setEnabled(fractal_mode and self.harmonic_blending_checkbox.isChecked())
    self.add_layer_button.setEnabled(fractal_mode and self.harmonic_blending_checkbox.isChecked())
    self.remove_layer_button.setEnabled(fractal_mode and self.harmonic_blending_checkbox.isChecked())

    # Елементи для режиму текстури
    self.start_x_edit.setEnabled(texture_mode)
    self.start_y_edit.setEnabled(texture_mode)
    self.x_value_edit.setEnabled(texture_mode)
    self.y_value_edit.setEnabled(texture_mode)
    self.coef_edit.setEnabled(texture_mode)
    self.tiling_checkbox.setEnabled(texture_mode)
    self.tile_size_edit.setEnabled(texture_mode and self.tiling_checkbox.isChecked())

    # Елементи для режиму фрактальної текстури
    self.a_param_edit.setEnabled(fractal_texture_mode)
    self.b_param_edit.setEnabled(fractal_texture_mode)
    self.k_param_edit.setEnabled(fractal_texture_mode)

    if hasattr(self, 'status_label'):
        if fractal_mode:
            self.status_label.setText("Режим фракталу - натисніть 'Обчислити'")
        elif texture_mode:
            self.status_label.setText("Режим текстури - натисніть 'Обчислити'")
        else:
            self.status_label.setText("Режим фрактальної текстури - натисніть 'Обчислити'")

# def on_slider_changed(self):
#     if hasattr(self, 'status_label'):
#         self.status_label.setText("Змінено параметри - відпустіть для оновлення або натисніть 'Обчислити'")

def on_slider_released(self):
    if hasattr(self, 'status_label'):
        self.status_label.setText("Змінено параметри - натисніть 'Обчислити'")

def on_texture_params_changed(self):
    # Оновлюємо відповідні параметри
    if hasattr(self, 'tiling_checkbox') and hasattr(self, 'tile_size_edit'):
        self.tile_size_edit.setEnabled(self.tiling_checkbox.isChecked() and self.texture_mode_radio.isChecked())
```

```
    if hasattr(self, 'status_label'):
        self.status_label.setText("Змінено параметри текстури - натисніть 'Обчислити'")

def on_fractal_texture_params_changed(self):
    if hasattr(self, 'status_label'):
        self.status_label.setText("Змінено параметри фрактальної текстури - натисніть 'Обчислити'")

def get_fractal_texture_params(self):
    """Отримати параметри фрактальної текстури з інтерфейсу"""
    try:
        return {
            'a_param': int(self.a_param_edit.text()),
            'b_param': int(self.b_param_edit.text()),
            'k_param': int(self.k_param_edit.text())
        }
    except ValueError:
        QMessageBox.warning(self, "Помилка",
                            "Некоректні параметри фрактальної текстури. Використовуємо значення за замовчуванням.")
        return {
            'a_param': 1,
            'b_param': 1,
            'k_param': 1
        }

# def on_texture_params_changed(self):
#     # Не запускаємо перерахунок автоматично, щоб не перевантажувати систему
#     # Користувач може натиснути кнопку "Обчислити" коли буде готовий
#     if hasattr(self, 'status_label'):
#         self.status_label.setText("Змінено параметри - натисніть 'Обчислити'")

def show_progress_indicator(self):
    """Показує індикатор прогресу для тривалих операцій"""
    # Створення діалогу з прогрес-баром
    if not hasattr(self, 'progress_dialog'):
        self.progress_dialog = QProgressDialog("Генерація текстури...", "Скасувати", 0, 100, self)
        self.progress_dialog.setWindowTitle("Обчислення")
        self.progress_dialog.setWindowModality(Qt.WindowModal)
        self.progress_dialog.setMinimumDuration(500)

    self.progress_dialog.reset()
    self.progress_dialog.setValue(0)
    self.progress_dialog.show()

    # Підключення скасування до зупинки обчислень
    self.progress_dialog.canceled.connect(self.cancel_computation)
```

```
def cancel_computation(self):
    """Обробник для скасування обчислень"""
    # Зупинка потоку генерації текстури
    if hasattr(self, 'texture_worker') and hasattr(self, 'texture_thread') and self.texture_thread.isRunning():
        self.texture_worker.stop()  # Встановлюємо прапорець abort
        self.texture_thread.quit()
        self.texture_thread.wait(1000)  # Чекаємо максимум 1 секунду

        # Якщо потік не завершився після очікування, завершуємо його примусово
        if self.texture_thread.isRunning():
            self.texture_thread.terminate()

    # Оновлюємо статус
    if hasattr(self, 'status_label'):
        self.status_label.setText("Генерацію скасовано")

def get_texture_params(self):
    """Отримати параметри текстури з інтерфейсу"""
    try:
        return {
            'start_x': int(self.start_x_edit.text()),
            'start_y': int(self.start_y_edit.text()),
            'x_value': int(self.x_value_edit.text()),
            'y_value': int(self.y_value_edit.text()),
            'coefficient': float(self.coef_edit.text()),
            'tiling': self.tiling_checkbox.isChecked(),
            'tile_size': int(self.tile_size_edit.text())
        }
    except ValueError:
        # Якщо виникла помилка конвертації, використовуємо значення за замовчуванням
        QMessageBox.warning(self, "Помилка",
                            "Некоректні параметри текстури. Використовуємо значення за замовчуванням.")
        return {
            'start_x': 1,
            'start_y': 1,
            'x_value': 1,
            'y_value': 1,
            'coefficient': 1.0,
            'tiling': True,
            'tile_size': 64
        }

def on_calculate_button_clicked(self):
    self.force_recalculate = True
    fractal_type = self.fractal_type_combo.currentIndex()
    use_adaptive = self.adaptive_detailing_checkbox.isChecked()
```

```
if hasattr(self, 'status_label'):
    self.status_label.setText("Обчислення...")

try:
    # Підготовка спільних параметрів для всіх типів фракталів
    try:
        a_param = int(self.a_param_edit.text())
        b_param = int(self.b_param_edit.text())
        k_param = int(self.k_param_edit.text())
    except ValueError:
        QMessageBox.warning(self, "Помилка",
                            "Некоректні параметри фрактальної текстури. Використовуємо значення за замовчуванням.")
        a_param, b_param, k_param = 1, 1, 1

    if fractal_type == 0:  # Мандельброт
        if use_adaptive:
            # Версія з адаптивною деталізацією
            texture = self.generate_mandelbrot_gpu_with_adaptive(
                1024, 1024,
                tileCount=2,  # 2x2 = 4 фрактали Мандельброта
                max_iterations=100,
                a_param=a_param,
                b_param=b_param,
                k_param=k_param,
                w1=self.w1_slider.value() / 100.0,
                w2=self.w2_slider.value() / 100.0,
                w3=self.w3_slider.value() / 100.0
            )
            self.status_label.setText("Готово - Мандельброт з адаптивною деталізацією згенеровано")
        else:
            # Стандартна версія без адаптивної деталізації
            texture = self.generate_mandelbrot_gpu(
                1024, 1024,
                tileCount=2,
                max_iterations=100,
                a_param=a_param,
                b_param=b_param,
                k_param=k_param
            )
            self.status_label.setText("Готово - Мандельброт згенеровано")

    elif fractal_type == 1:  # Джулія
        # Параметри для Джулії
        julia_real = 0.285
        julia_imag = 0.01
```

```
        if use_adaptive:
            # Версія з адаптивною деталізацією
            texture = self.generate_julia_gpu_with_adaptive(
                1024, 1024,
                center_x=0.0, center_y=0.0,
                view_width=3.0,
                max_iterations=100,
                a_param=a_param,
                b_param=b_param,
                k_param=k_param,
                julia_real=julia_real,
                julia_imag=julia_imag,
                w1=self.w1_slider.value() / 100.0,
                w2=self.w2_slider.value() / 100.0,
                w3=self.w3_slider.value() / 100.0
            )
            self.status_label.setText("Готово - Джулія з адаптивною деталізацією згенеровано")
        else:
            # Стандартна версія без адаптивної деталізації
            texture = self.generate_julia_gpu(
                1024, 1024,
                center_x=0.0, center_y=0.0,
                view_width=3.0,
                max_iterations=100,
                a_param=a_param,
                b_param=b_param,
                k_param=k_param,
                julia_real=julia_real,
                julia_imag=julia_imag
            )
            self.status_label.setText("Готово - Джулія згенеровано")

    elif fractal_type == 2:  # Фрактальна текстура
        if use_adaptive:
            # Версія з адаптивною деталізацією
            texture = self.generate_simple_texture_gpu_with_adaptive(
                1024, 1024,
                a_param, b_param, k_param,
                w1=self.w1_slider.value() / 100.0,
                w2=self.w2_slider.value() / 100.0,
                w3=self.w3_slider.value() / 100.0
            )
            self.status_label.setText("Готово - Фрактальна текстура з адаптивною деталізацією згенерована")
        else:
            # Стандартна версія без адаптивної деталізації
```

```
            texture = self.generate_simple_texture_gpu(
                1024, 1024,
                a_param, b_param, k_param
            )
            self.status_label.setText("Готово - Фрактальна текстура згенерована")

        # ВАЖЛИВА ЗМІНА: перевіряємо, чи увімкнено гармонійне змішування,
        # але НЕ ЗМІШУЄМО поточне зображення з шарами, а показуємо тільки результат змішування шарів
        if self.harmonic_blending_checkbox.isChecked() and self.fractal_layers:
            # Змішуємо тільки шари зі списку "Шари:" (без поточного зображення)
            self.update_blended_texture()
        else:
            # Відображаємо тільки поточне згенероване зображення
            self.gl_widget.set_texture(texture)

    except Exception as e:
        import traceback
        print(f"Помилка при обчисленні фракталу: {str(e)}")
        traceback.print_exc()
        QMessageBox.critical(self, "Помилка", f"Виникла помилка при обчисленні: {str(e)}")
        if hasattr(self, 'status_label'):
            self.status_label.setText("Помилка при обчисленні")
    finally:
        self.force_recalculate = False
```

ДОДАТОК Г

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ФОРМУВАННЯ ТЕКСТУР З ВИКОРИСТАННЯМ ФРАКТАЛІВ

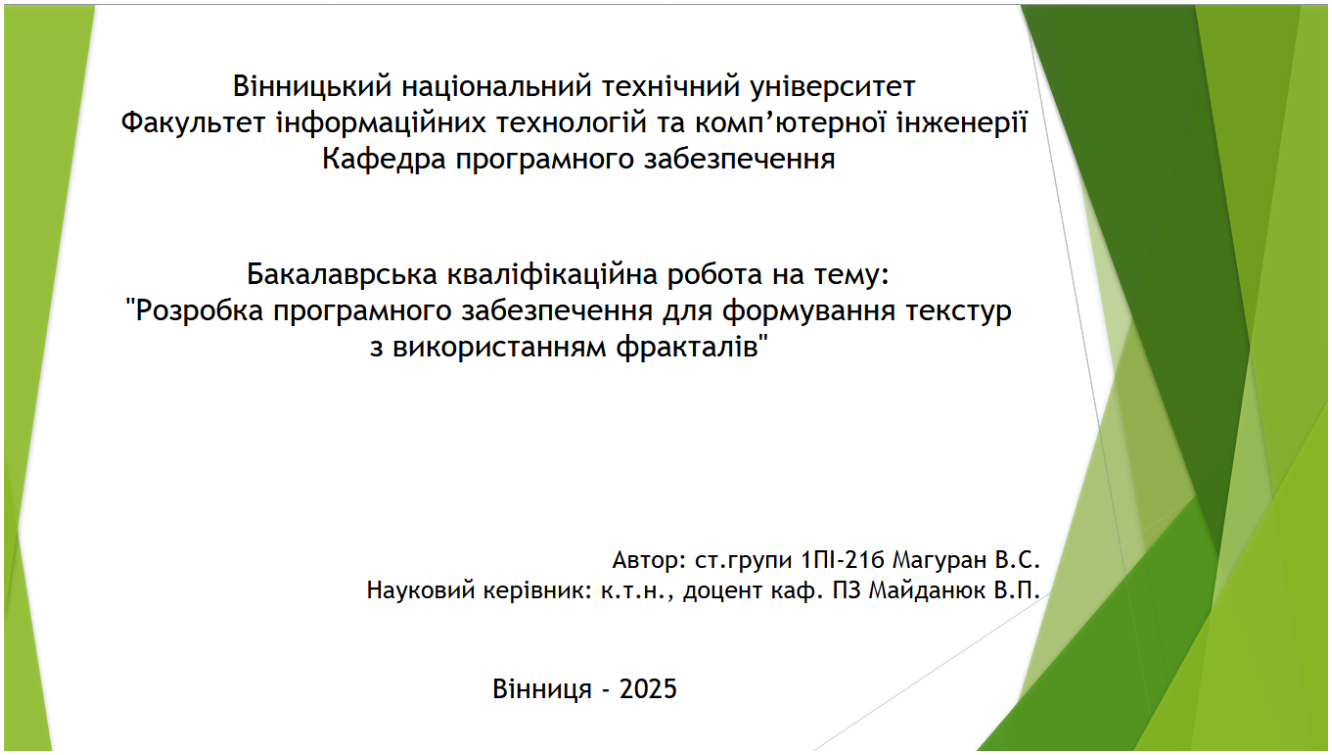


Рисунок Г.1 – Титульний слайд

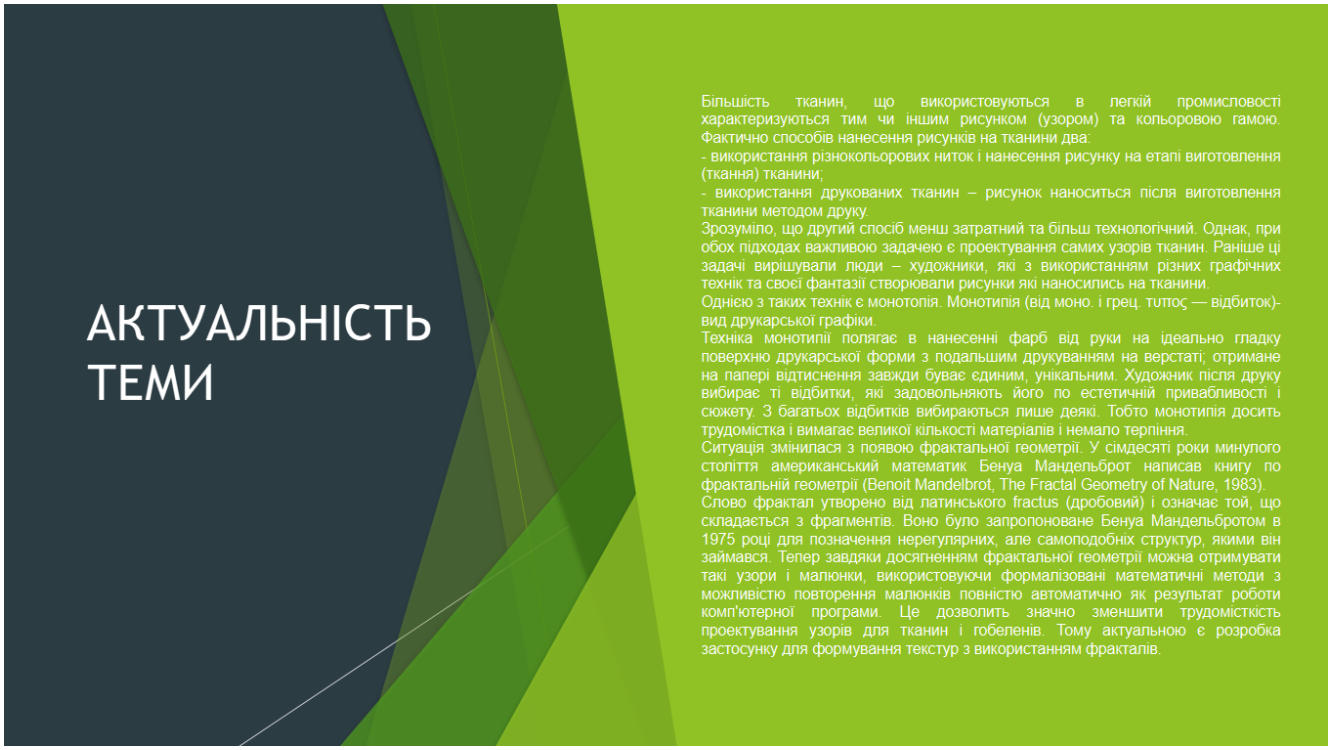


Рисунок Г.2 — Актуальність теми

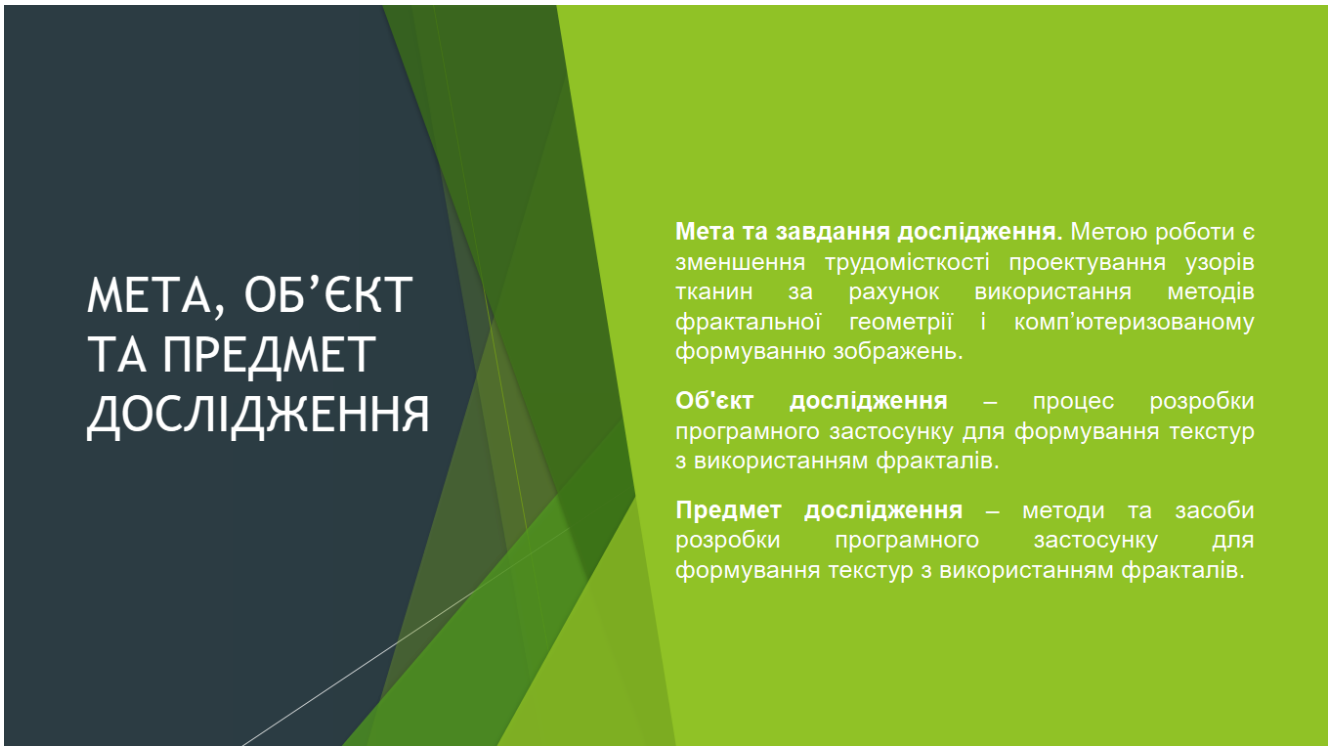


Рисунок Г.3 — Мета, об'єкт та предмет дослідження

Рисунок Г.4 — Завдання дослідження

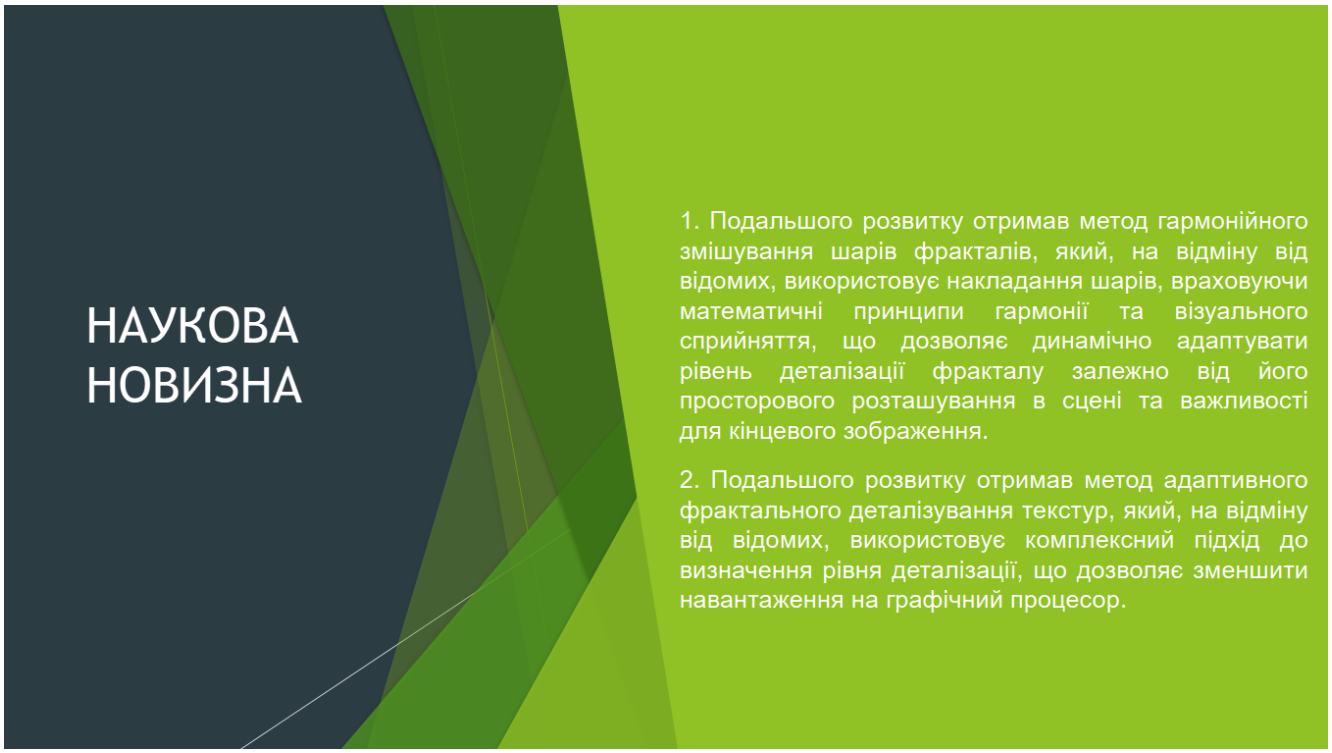


Рисунок Г.5 — Наукова новизна

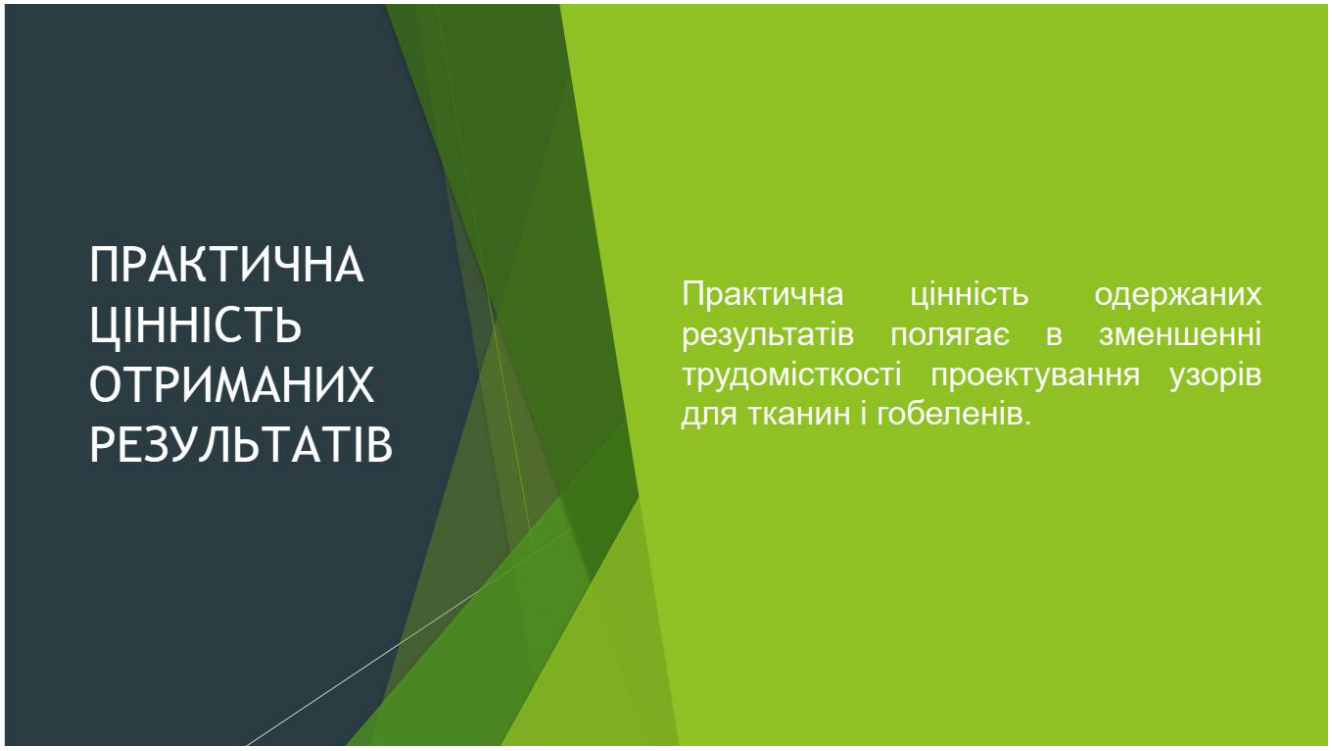


Рисунок Г.6 — Практична цінність

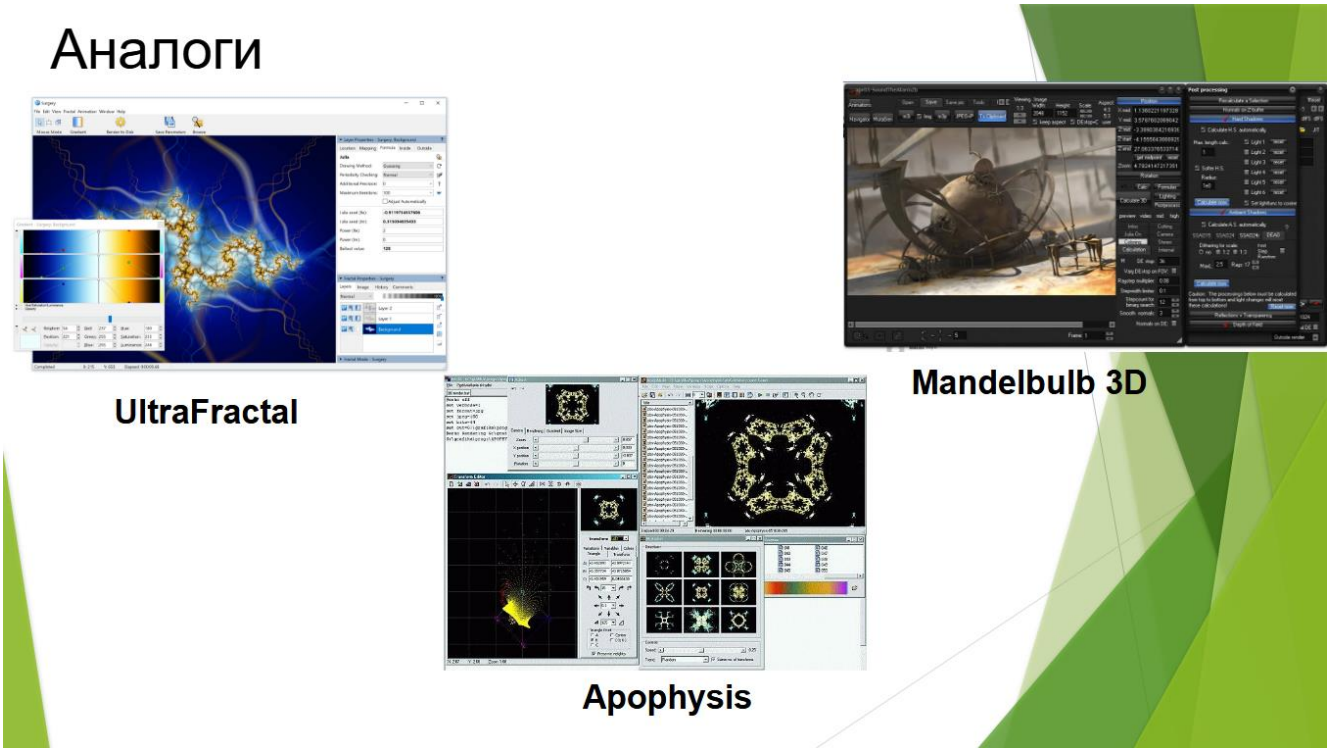


Рисунок Г.7 — Аналоги

Порівняльний аналіз аналогів

Характеристика	Ultra Fractal	Mandelbulb 3D	Apophysis	Власна розробка
Підтримка 2D фракталів	+	+	+	+
Гнучкість налаштувань	+	+	+	+
Інтерактивність	+	+	+	+
Багатошаровість	+	+	+	+
Адаптивна деталізація текстури	-	-	-	+
Гармонійне змішування шарів	-	-	-	+
Загальний бал	4	3	4	6

Рисунок Г.8 — Порівняльний аналіз аналогів

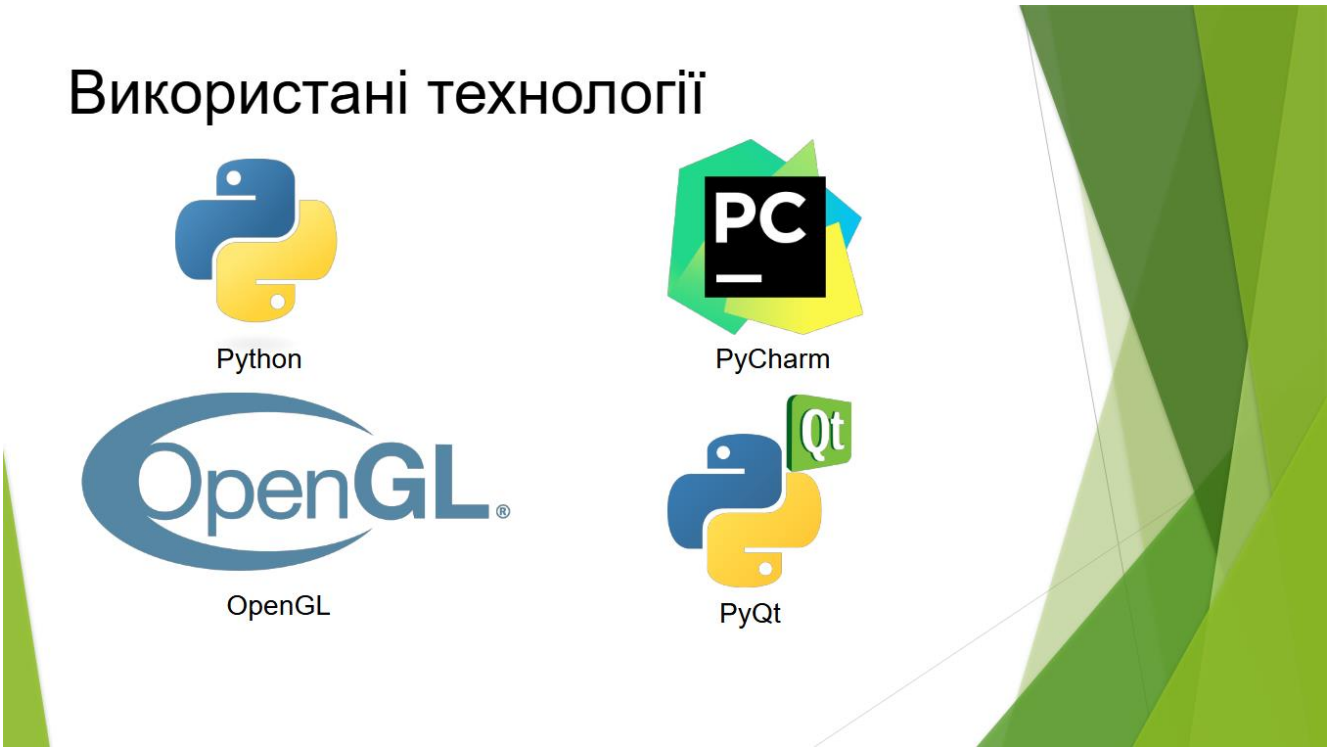


Рисунок Г.9 — Використані технології

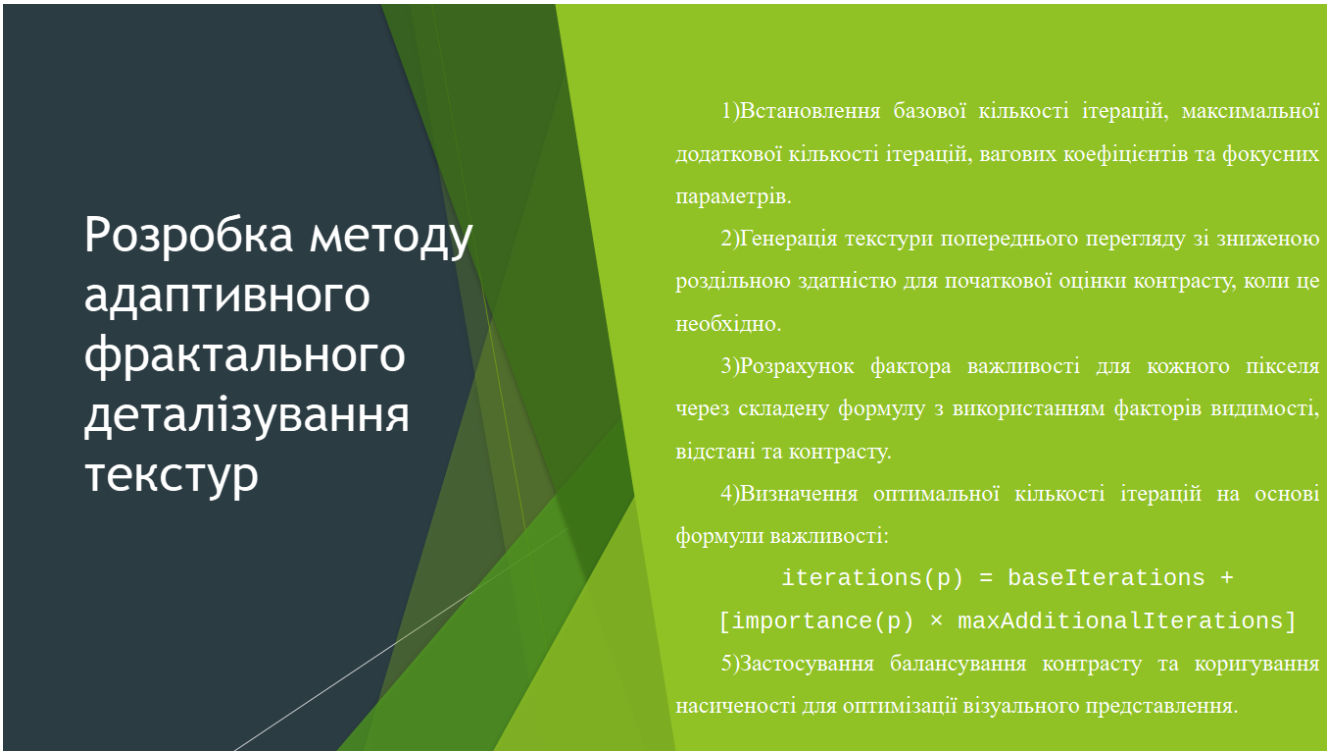


Рисунок Г.10 — Розробка методу адаптивного фрактального деталізування текстур

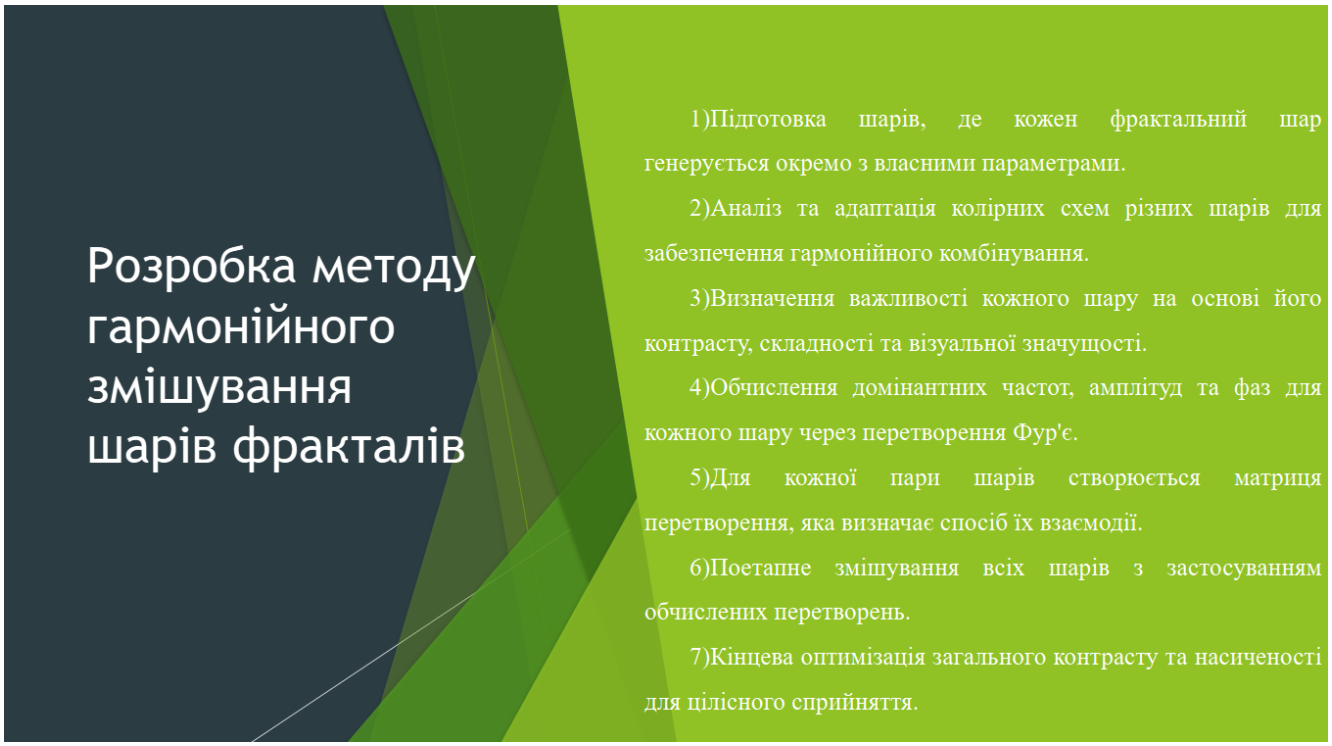


Рисунок Г.11 — Розробка методу гармонійного змішування шарів фракталів

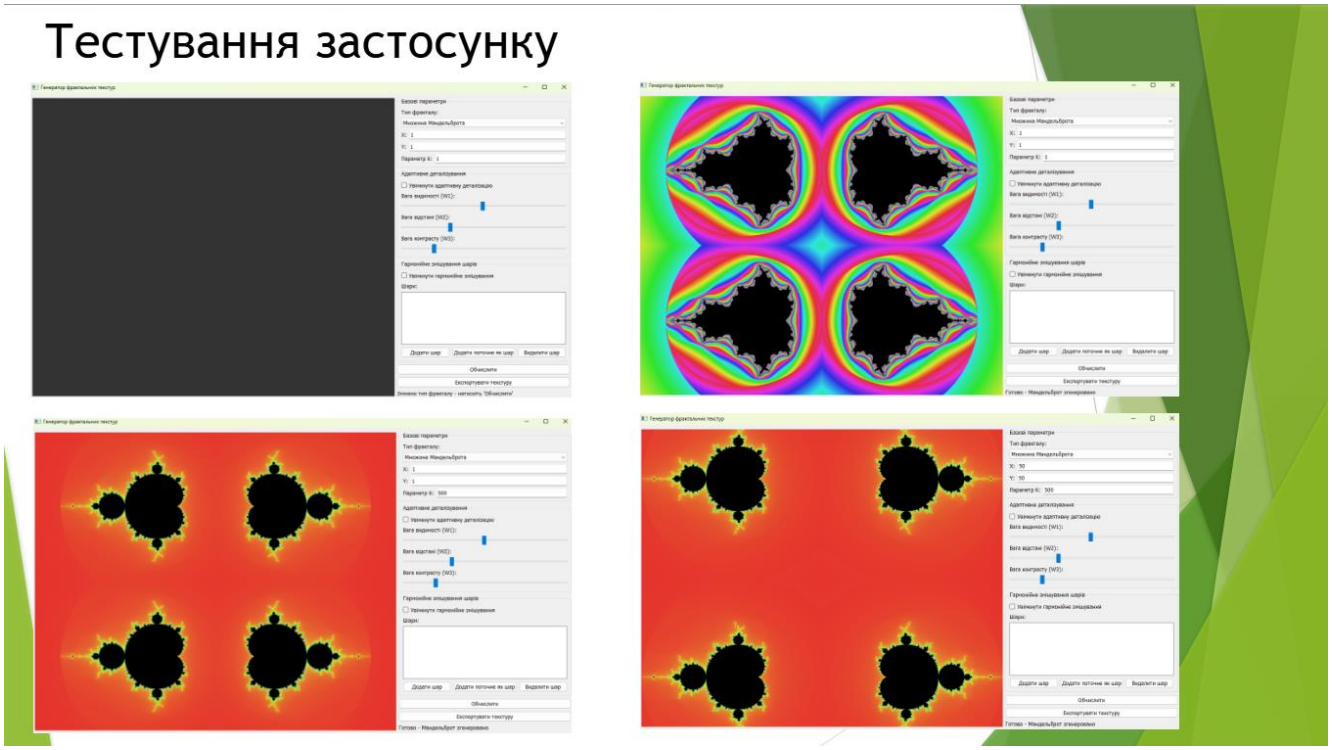


Рисунок Г.12 — Тестування формування текстури з фракталом множини Мандельброта

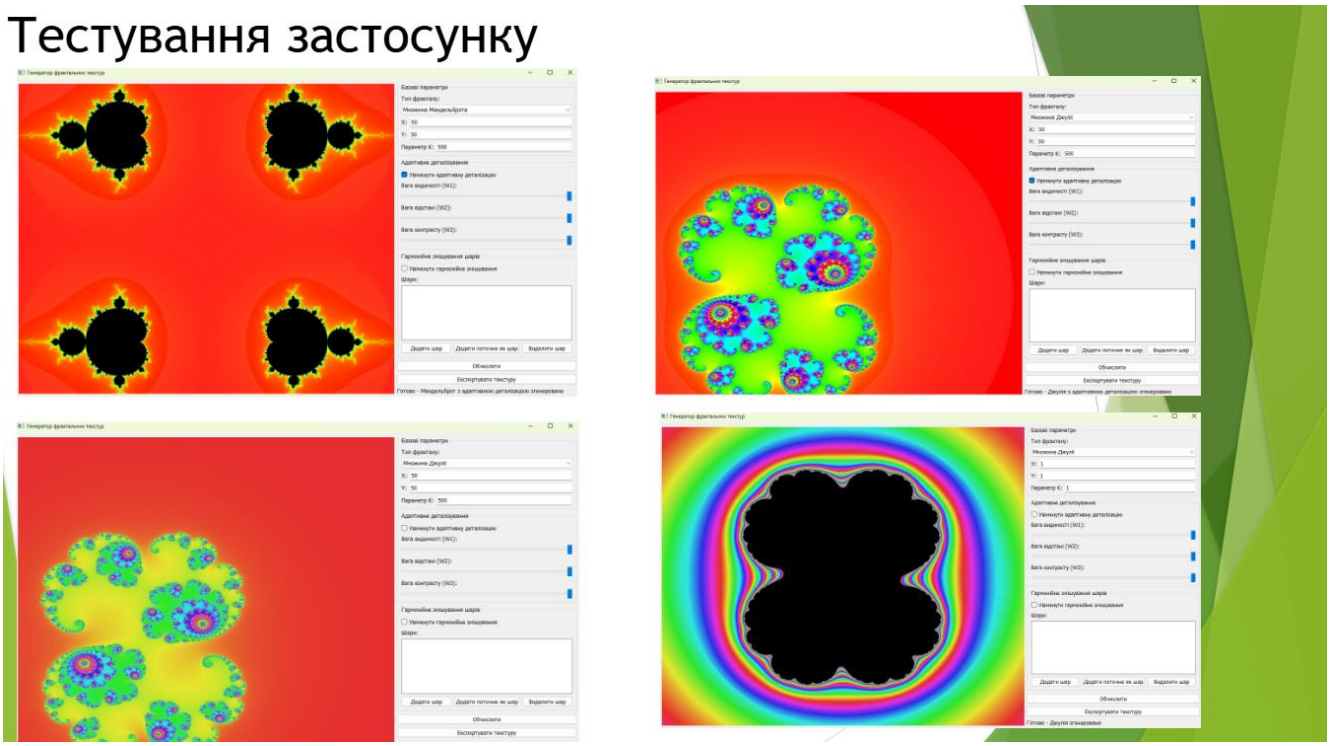


Рисунок Г.13 — Тестування формування текстури з фракталом множини Джуліа

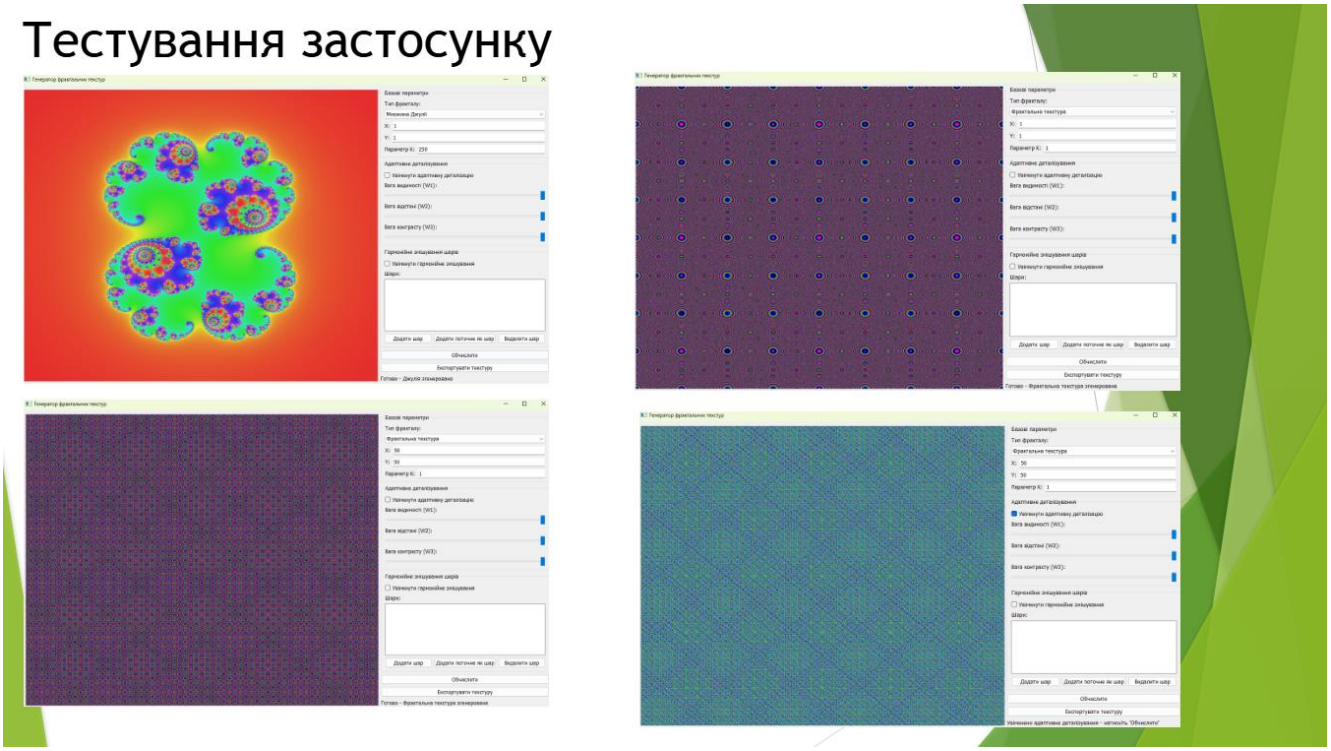


Рисунок Г.14 — Тестування фрактальної текстури

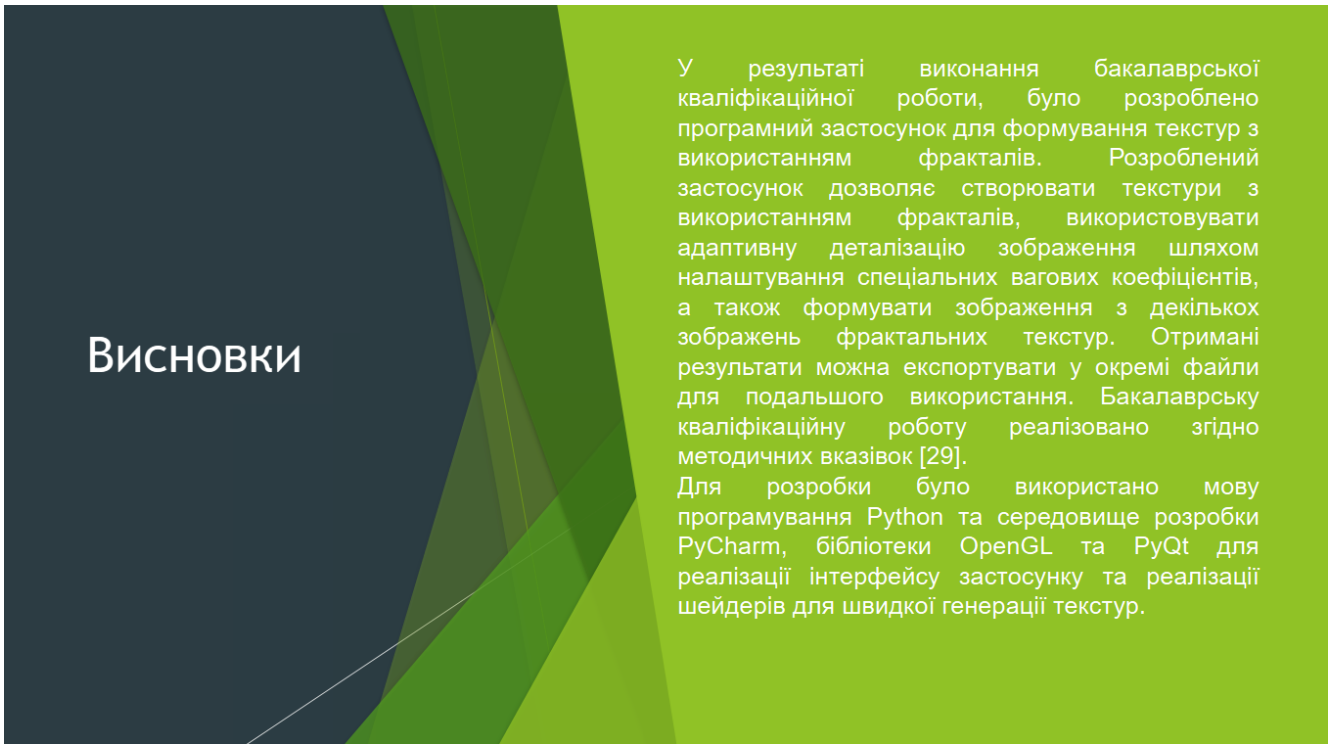


Рисунок Г.15 — Тестування (частина 1)

Рисунок Г.16 — Тестування (частина 2)

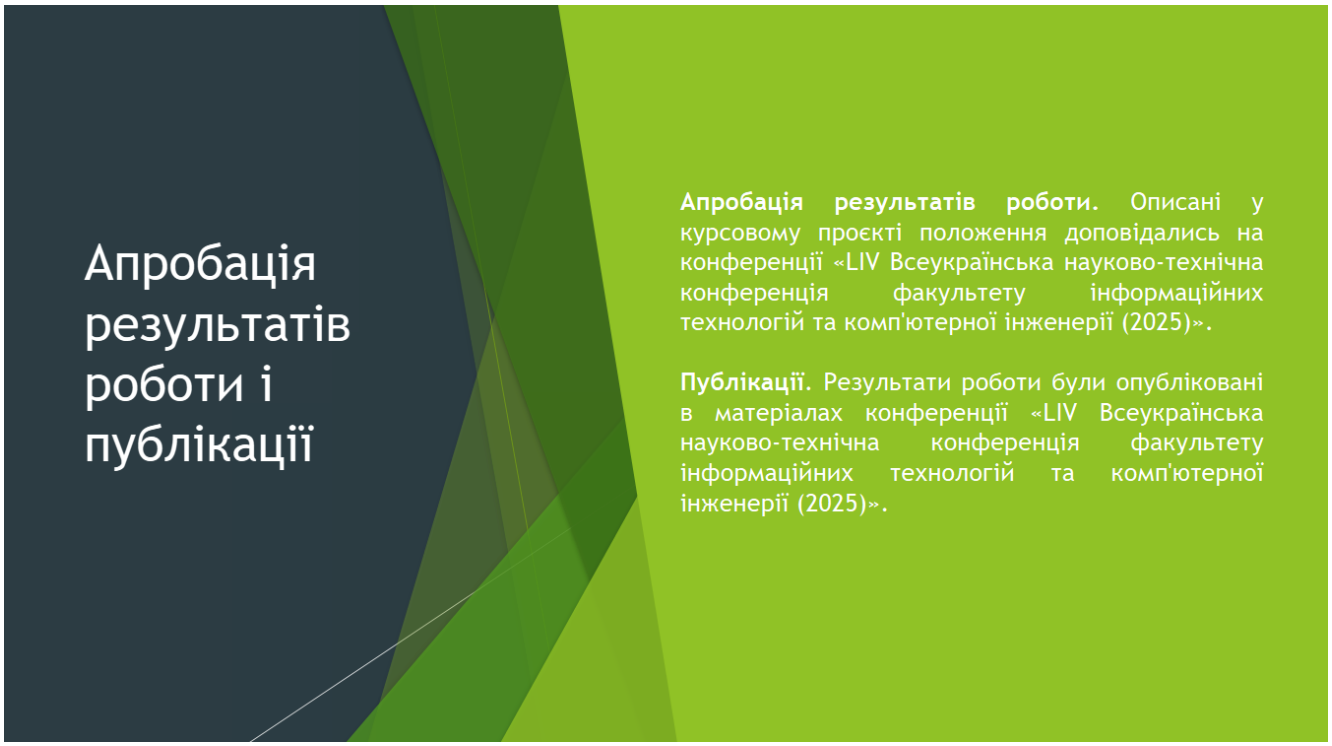


Рисунок Г.17 — Апробація результатів роботи і публікації

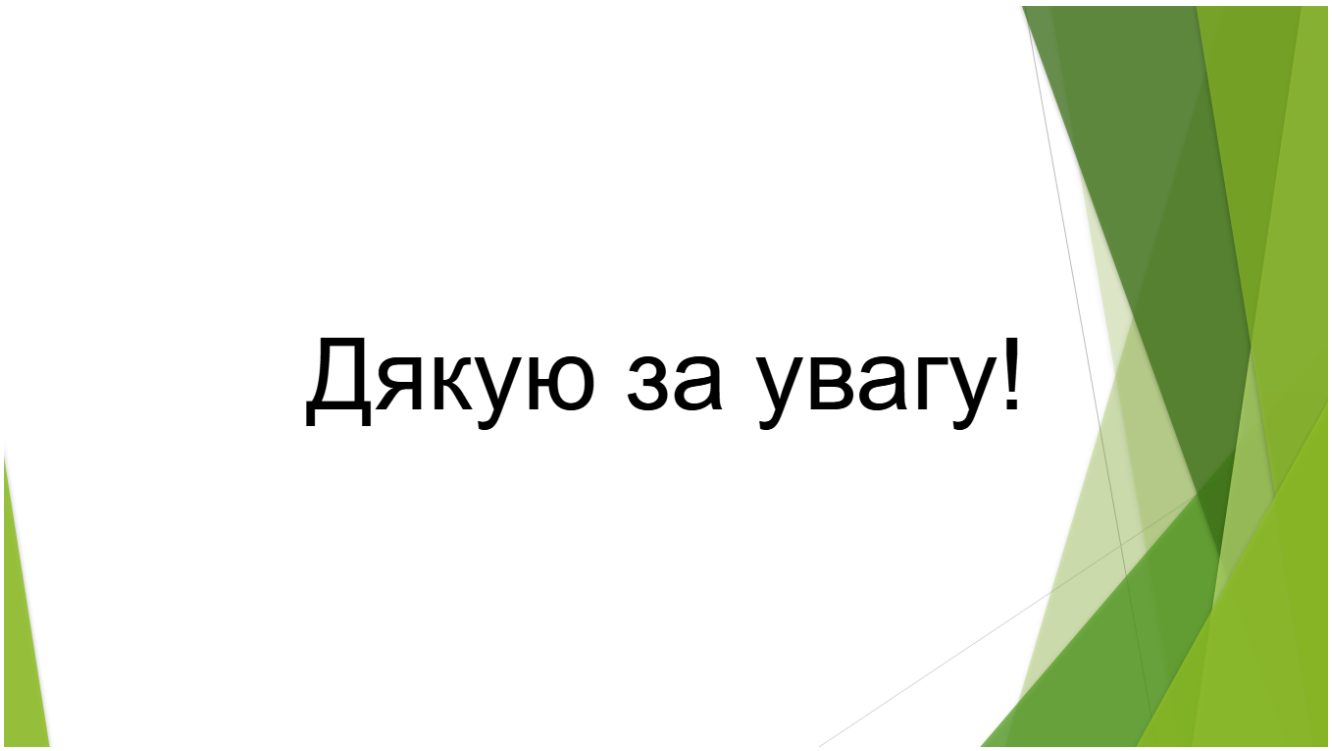


Рисунок Г.18 — Фінальний слайд