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

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.4'24

«До захисту допущено»

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інженерія програмного забезпечення інформаційних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Методи та програмні засоби конвертації зображень у DDS текстури»

Виконав:
студент ІІ курсу, групи ІП-11мн
Нестеренко Костянтин Павлович _____

Керівник:
д.т.н., проф.
Стеценко Інна Вячеславівна _____

Рецензент:
д.ф.-м.наук, проф.
Гордієнко Юрій Григорович _____

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

Київ – 2023 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2023р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Нестеренко Костянтин Павлович

1. Тема дисертації «Методи та програмні засоби конвертації зображень у DDS текстури», науковий керівник дисертації Стеценко Інна Вячеславівна, д.т.н., проф., затверджені наказом по університету від «23» березня 2023 р. № 1275с

2. Термін подання студентом дисертації «16» травня 2023 р.

3. Об’єкт дослідження – програмне забезпечення для конвертації зображень у DDS текстури.

4. Предмет дослідження – методи розробки програмного забезпечення для конвертації зображень у DDS текстури.

5. Перелік завдань, які потрібно розробити – аналіз проблеми та існуючих рішень для конвертації зображень у DDS текстури; розробка методу для написання ефективного та швидкого багатопоточного програмного коду; реалізація програмного засобу для конвертації зображень у DDS текстури на основі розробленого методу; дослідження ефективності розробленого програмного забезпечення.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 1 плакат

7. Орієнтовний перелік публікацій – одна публікація

8.Дата видачі завдання «5» лютого 2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Аналіз предметної області	5.02.2023	
2	Робота з літературою по темі роботи	20.02.2023	
3	Аналіз існуючих рішень	8.03.2023	
4	Розробка методу автоматизації розробки багатопоточної програми	24.03.2023	
5	Розробка архітектури програмного засобу на основі запропонованого методу	6.04.2023	
6	Реалізація програмного засобу	24.04.2023	
7	Виконання експериментальних досліджень	28.04.2023	
8	Оформлення пояснювальної записки	6.05.2023	
9	Подання дисертації на попередній захист	10.05.2023	
10	Подання дисертації на захист	16.05.2023	

Студент Костянтин НЕСТЕРЕНКО

Науковий керівник Інна СТЕЦЕНКО

РЕФЕРАТ

Розмір пояснювальної записки – 70 аркушів, містить 27 ілюстрацій, 3 додатки, 25 посилань на джерела.

Актуальність теми. На сьогоднішній день, DDS є найбільш розповсюдженим форматом текстур для будь-якого програмного забезпечення з використанням двовимірної або тривимірної графіки. Проте конвертація зображення у DDS текстуру це досить вибаглива за часом операція. Оскільки з часом вимоги до якості комп'ютерної графіки постійно зростали, оригінальні зображення можуть мати досить великий розмір. І коли мова йде про сотні а часто й тисячі таких зображень, швидкодія стає однією з основних вимог до програмних засобів для конвертації зображень у DDS текстури. Тому вдосконалення програмних засобів та методів конвертації зображень у DDS текстури є актуальною науково-прикладною проблемою.

Мета дослідження. Основною метою є покращення програмного засобу для конвертації зображень у текстури для забезпечення кращих показників швидкодії відносно існуючих рішень.

Об'єкт дослідження: програмне забезпечення для конвертації зображень у DDS текстури.

Предмет дослідження: методи розробки програмного забезпечення для конвертації зображень у DDS текстури.

Для реалізації поставленої мети **сформульовані наступні завдання**:

- аналіз проблеми та існуючих рішень для конвертації зображень у DDS текстури;
- розробка методу для написання ефективного та швидкого багатопоточного програмного коду;
- реалізація програмного засобу для конвертації зображень у DDS текстури на основі розробленого методу;

- дослідження ефективності розробленого програмного забезпечення.

Наукова новизна результатів магістерської дисертації полягає наступних результатах:

- вперше розроблено метод автоматизації розробки багатопоточного програмного коду мовою C++, який за допомогою високорівневих абстракцій дає змогу розбити процес на окремі задачі та виконати їх асинхронно з урахуванням встановлених залежностей, ефективно використовуючи при цьому наявні ресурси програми;
- вдосконалено програмний засіб для конвертації зображень у текстури за рахунок використання запропонованого методу автоматизації розробки багатопоточного програмного забзпечення, що забезпечує кращі показники швидкодії відносно існуючих рішень.

Практичне значення отриманих результатів полягає в розробці швидкого та ефективного програмного засобу для конвертації зображень у текстури, що дозволяє суттєво скорити витрати часу кінцевого користувача при використанні аналогічних програмних засобів.

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Апробація. Наукові положення дисертації пройшли апробацію на IV міжнародній конференції «Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2023)» – м. Київ.

Публікації. Наукові положення дисертації опубліковані в:

1) Нестеренко К.П., Стеценко І.В. Метод автоматизації розробки багатопоточної програми мовою C++ на прикладі конвертації зображень у

DDS текстури. // Адаптивні систем автоматичного управління. – Київ, 2023. - №1(42) – С. 160 –170. (категорія «Б»).

Ключові слова: БАГАТОПОТОЧНІСТЬ, ШВИДКОДІЯ, ТЕКСТУРА, DDS, ВС, ПРОГРАМНИЙ ЗАСІБ, КОНВЕРТАЦІЯ ЗОБРАЖЕНЬ.

ABSTRACT

Explanatory note size – 70 pages, contains 27 illustrations, 3 applications, 25 references.

Topicality. As of today, DDS is the most widely used texture format for any software that utilizes 2D or 3D graphics. However, converting an image to a DDS texture is a time-consuming operation. As the requirements for computer graphics quality continue to grow over time, original images mages can be a quite large. When it comes to hundreds or often thousands of such images, time performance becomes one of the main requirements for image-to-DDS texture conversion software. Therefore, the improvement of software tools and methods for converting images to DDS textures is an important scientific and applied problem.

The aim of the study. The aim is to improve the software tool for converting images to textures to provide better performance metrics compared to existing solutions.

The object of the research: software for converting images to DDS textures.

The subject of the research: techniques for software development for converting images to DDS textures.

To achieve this goal, the **following tasks** were formulated:

- analysis of the problem and existing solutions for converting images to DDS textures;
- development of a method for writing efficient and fast multithreaded software code;
- implementation of software tools for converting images to DDS textures based on the developed method;
- investigation of the efficiency of the developed software.

The scientific novelty of the results of the master's dissertation consists of the following results:

- for the first time, a technique for automating the development of multithreaded software code using C++ has been developed, which utilizes high-level abstractions to break down the process into individual tasks and execute them asynchronously, taking into account established dependencies and efficiently utilizing program resources;
- the software tool for image-to-texture conversion has been improved by utilizing the proposed method for automating the development of multithreaded software, which provides better performance compared to existing solutions.

The practical value of the obtained results is the development of a fast and efficient software tool for converting images into textures, which significantly reduces the time and effort required by end-users compared to similar software tools.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Kyiv Polytechnic Institute. Igor Sikorsky».

Approbation. The scientific provisions of the thesis were tested at the IV International Conference "Software Engineering and Advanced Information Technologies (SoftTech-2023)" – Kyiv.

Publications. The scientific provisions of the thesis were published in:

1) Nesterenko K.P., Stetsenko I.V. Method of automating the development of a multithreaded program in C++ on the example of conversion of images into the DDS texture // Adaptive Systems of Automatic Control. – Kyiv, 2023. - №1(42) – p. 160 –170. (category "B").

Keywords: MULTITHREADING, PERFORMANCE, TEXTURE, DDS, BC, SOFTWARE TOOL, IMAGE CONVERSION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

JPEG (Joint Photographic Experts Group) – растровий формат зберігання графічної інформації, що використовує стиснення з втратами якості зображення.

PNG (Portable Network Graphics) – растровий формат збереження графічної інформації, що використовує стиснення без втрат.

BMP (Bitmap) – формат файлу зображень растрової графіки, де зображення зберігається у вигляді двовимірного масиву з пікселів, кожен піксель має власний колір.

DDS (DirectDraw Surface) – компресований формат зберігання даних розроблений корпорацією Microsoft, з метою зменшення розміру файлів та меншого навантаження на графічний процесор.

BC (Block Compression) – сімейство алгоритмів стиснення з втратами, які використовуються під час конвертації зображень у текстури в форматі DDS.

API (Application Programming Interface) – набір чітко визначених методів для взаємодії різних компонентів.

UML (Unified Modeling Language) – уніфікована мова моделювання, що використовується у парадигмі об'єктно-орієнтованого програмування.

RGBA (Red Green Blue Alpha) – чотирьох канальний спосіб кодування кольорів, який окрім стандартних трьох каналів містить додатковий канал для збереження значення прозорості кольору.

ВСТУП

З розвитком технологій поступово все більше і більше різноманітні гаджети та програмне забезпечення інтегрувалися і різні сфери життєдіяльності людини та на даний час вже можна точно стверджувати, що вони безповоротно стали невід’ємною їх складовою.

І оскільки за своєю природою людина звична до сприйняття інформації візуально, однією з таких категорій програмного забезпечення є застосунки з використання двовимірної та тривимірної графіки. На початку воно використовувалося здебільшого в сфері розваг, наприклад в відеоіграх або кінематографі. Проте з розвитком технологій це програмне забезпечення ставало все більш привабливим для інших, більш серйозних та вибагливих галузей, де помилка програмного забезпечення буквально може коштувати людині життя.

Тим не менш зараз подібне програмне забезпечення використовується без перебільшення усюди. Починаючи від медицини і закінчуючи військово-промисловим комплексом, програмне забезпечення з використанням двовимірної та тривимірної графіки стало невід’ємною складовою діяльності сучасного спеціаліста. Хірург може буквально спланувати та симулювати перебіг операції до її початку. Інженер може перевірити роботу механізму без необхідності витрачати ресурси на побудову прототипу. Архітектор може перевірити як буде поводити себе будівля в екстремальних умовах або під час природних катаклізмів. І цей список можна продовжувати майже нескінченно.

Проте, усе це програмне забезпечення в своїй роботі спирається на використання графічного процесору, який свою чергу є пристроєм оптимізованим для роботи з певними форматами даних. І якщо ми кажемо про зображення, то таким форматом будуть DDS текстури. Тобто будь-яке

програмне забезпечення з двовимірною або тривимірною графікою так чи інакше потребує конвертації з зображень у форматах, що зазвичай використовуються для зберігання зображень, таких як JPEG, PNG та BMP, у формат даних DDS.

Конвертація зображення у DDS текстуру це досить вибаглива за часом операція, оскільки вона вимагає великої кількості математичних операцій. А оскільки з часом вимоги до якості комп'ютерної графіки постійно зростали, оригінальні зображення можуть мати досить великий розмір. І коли мова йде про сотні а часто й тисячі таких зображень, швидкодія стає однією з основних вимог до програмних засобів для конвертації зображень у DDS текстури.

Метою даної дисертації є дослідження програмних засобів та методів конвертації зображень у DDS текстури. В рамках дослідження будуть проаналізовані вже існуючі рішення, оцінена їх швидкодія та придатність до використання з великою кількістю вхідних зображень. За результатами цього аналізу буде розроблено програмний засіб для конвертації зображень у DDS текстури, метою якого буде покращити швидкодію відносно існуючих засобів.

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Програмні засоби для конвертації зображень у DDS текстури можна розділити на наступні категорії за способом їх використання: плагін для графічного редактору, окрема утиліта та як окремий модуль програмного забезпечення, що використовує двовимірну або тривимірну графіку, наприклад ігровий рушій або програмне забезпечення для тривимірного моделювання. Кожен з перелічених видів має свої переваги та недоліки, які впливають як на швидкодію програмного забезпечення так і на те для яких потреб його використання є доцільним. Пропонується розглянути їх на прикладі найбільш популярних та просунутих реалізацій.

Для порівняння швидкодії програмного забезпечення був створений тестовий набір даних з десяти зображень збережених у форматі PNG розміром 4096 на 4096 пікселі. Варто зазначити, що вміст зображення не впливає на процес конвертації. Для конвертації буде використаний алгоритм конвертації BC1 [1]. Усі заміри будуть виконуватися в однакових умовах з однаковою конфігурацією персонального комп’ютера. У випадку якщо програмне забезпечення не надає можливості точно виміряти час виконання або обробити декілька зображень одночасно, час виконання буде визначено приблизно.

1.1 Графічний редактор Paint.NET

Paint.NET [2] - це безкоштовний графічний редактор, який був розроблений для платформи Windows. Він пропонує ряд функцій, які забезпечують його функціональність для створення, редагування та покращення зображень.

Paint.NET має простий інтерфейс (рис. 1.1), що дозволяє користувачам швидко і легко виконувати операції з редагування зображень, такі як

розміщення тексту, видалення об'єктів, ретушування фотографій, корекція кольору та багато іншого. Крім того, він підтримує шари, що дозволяє користувачам працювати з окремими елементами зображення, щоб забезпечити більшу гнучкість в процесі редагування.

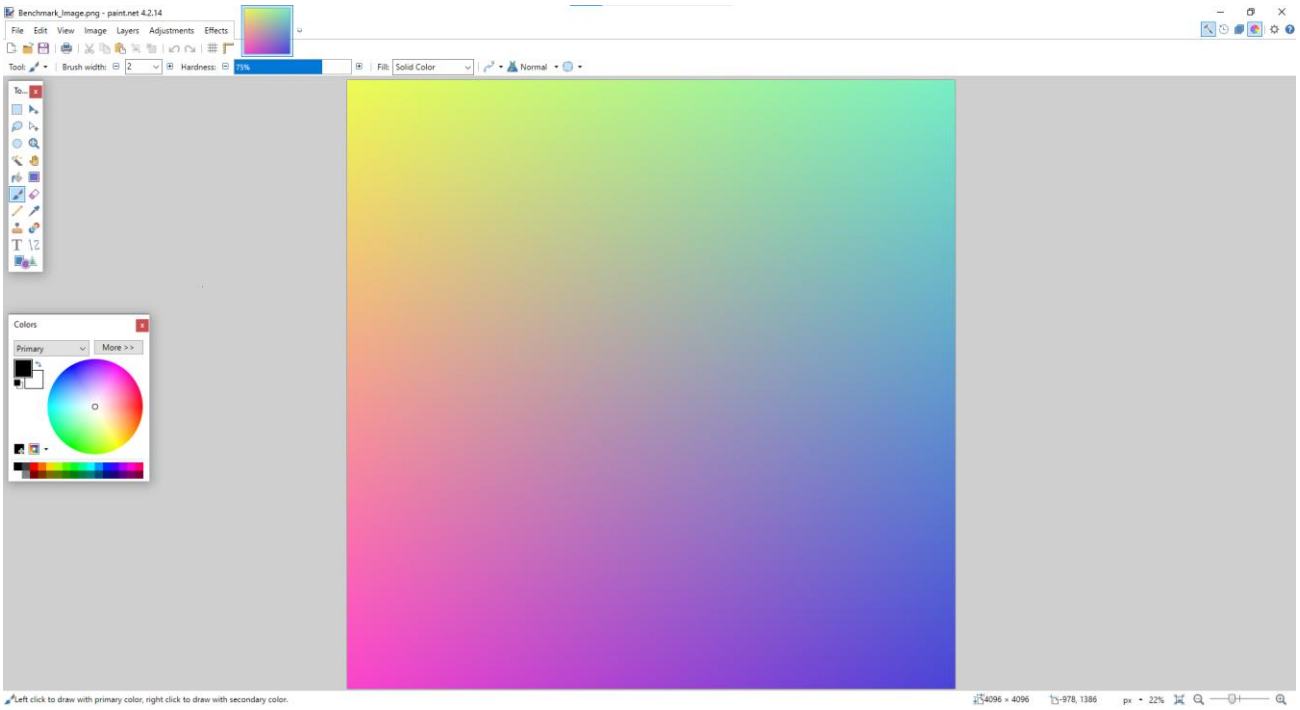


Рисунок 1.1 – Інтерфейс графічного редактору Paint.NET

Основні можливості Paint.NET включають роботу з растровими зображеннями, підтримку багатьох форматів файлів, включаючи BMP, JPEG, PNG, TIFF та GIF, можливість зберігати проекти для подальшого редагування, можливість масштабування та обрізання зображень, корекція кольору, функції ретушування, та багато іншого. Він також дозволяє використовувати клавіатурні скорочення, щоб забезпечити більшу продуктивність при роботі з програмою.

Додатковою функцією Paint.NET є наявність плагінів, що дозволяють розширити функціональність редактора, так що користувачі можуть

використовувати його для створення складних ефектів та фільтрів. На офіційному сайті Paint.NET є велика кількість плагінів, які можна безкоштовно завантажити та використовувати.

Один з таких плагінів, який по замовчуванню встановлюється разом з графічним редактором, є плагін для роботи з DDS текстурами. Він дозволяє користувачу переглядати та редагувати DDS текстури, а також зберігати зображення таких форматів як JPEG, PNG та BMP у форматі DDS текстур (рис. 1.2). В налаштуваннях доступні найбільш вживані алгоритми компресії такі як BC1, BC2 та BC3 [1].

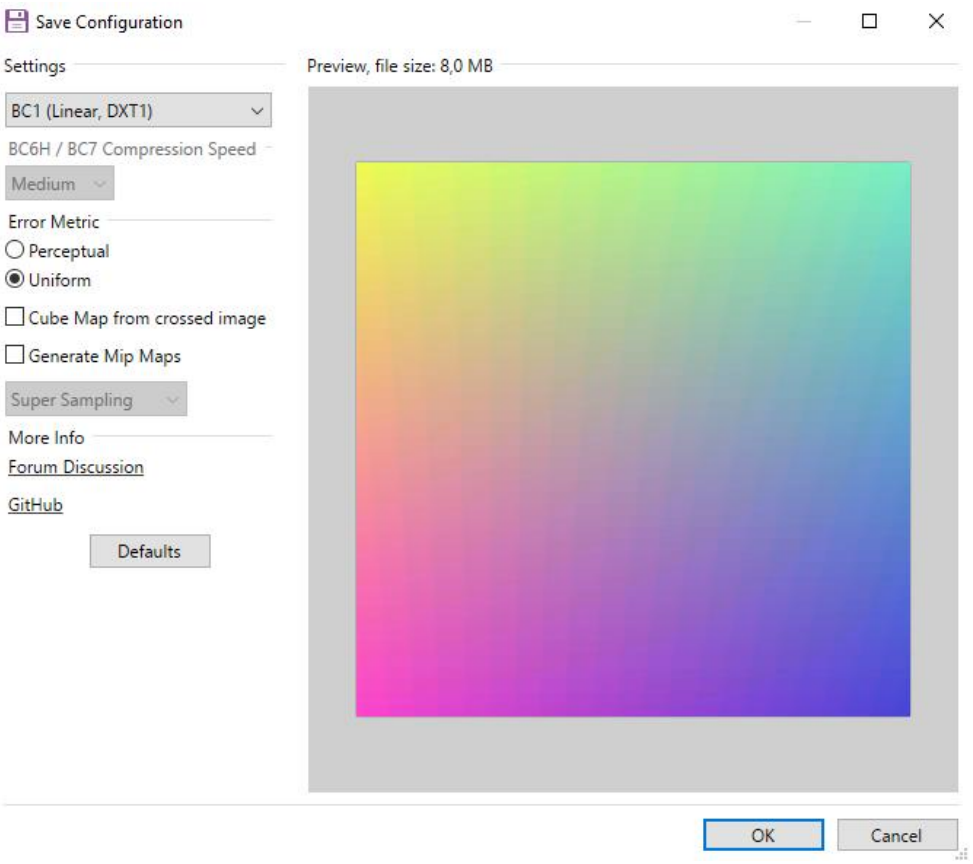


Рисунок 1.2 – Інтерфейс конвертації зображень у DDS текстури у графічному редакторі Paint.NET

1.1.1 Переваги програмного засобу

Основна перевага реалізації програмного засобу для конвертації зображень у DDS текстури у вигляді плагіну для графічного редактору полягає в тому, що користувачу не потрібно використовувати декілька інструментів для роботи. Такі редактори як Paint.NET є досить поширеними, особливо якщо у користувача немає можливості або необхідності у придбанні більш просунутих інструментів таких як Adobe Photoshop, комерційна ліцензія для яких є достатньо дорогою. Фактично користувач поєднує в одному програмному засобі інструменти для роботи та редагування зображень та конвертації зображень у DDS текстури. З точки зору бізнесу це є хорошим рішенням, оскільки не потрібно витрачати ресурси на підримку більшої кількості програмного забезпечення, та спрощується процес навчання нових співробітників.

Також користуючись можливостями редактору користувач може одразу переглянути результат конвертації у текстуру. Це дозволяє візуально оцінити наявність дефектів та у разі потреби порівняти результати виконання різних алгоритмів конвертації для того щоб обрати найбільш доцільний за співвідношенням якості до фізичного розміру вихідної текстури.

1.1.2 Недоліки програмного засобу

Основним недоліком реалізації програмного засобу для конвертації зображень у DDS текстури у вигляді плагіну для графічного редактору є те, що таке рішення погано масштабується та не підходить для роботи з великою кількістю текстур. Програмне забезпечення такого типу в першу чергу залишається графічним редактором, а тому як наслідок воно розраховане на роботу з невеликою кількістю зображень одночасно. Можливість конвертувати декілька зображень одночасно, Paint.NET та його аналоги не надають.

Відповідно конвертація навіть кількох десятків, не кажучи вже про сотні і тисячі зображень у DDS текстури буде вимагати великих витрат часу спеціаліста на виконання примітивної та рутинної роботи.

Також не для всіх проєктів з використанням двовимірної або тривимірної графіки графічний редактор є обов'язком інструментом для кожного співробітника. У такому випадку окремий інструмент, який виконує конкретну задачу з конвертації зображень у DDS текстури, був би більш прийнятним. Тому графічний редактор не є універсальним рішенням, яке підходить для будь-якого проєкту.

1.1.3 Аналіз швидкодії програмного засобу

Графічний редактор Paint.NET не надає інформації про те, скільки саме часу зайняв процес конвертації зображення у DDS текстуру. Також оскільки він не дає можливості конвертувати кілька зображень одночасно то заміряти можливо лише час витрачений на конвертацію одного зображення. Цей час становив приблизно 1.5 секунди для зображення визначеного для порівняння програмних засобів між собою (розділ 1). Проте варто зазначити, що даний процес не є автоматизованим і має виконуватися користувачем в ручну з вибором налаштувань для кожного окремого зображення. Відповідно реальні витрати часу користувача будуть набагато перевищувати час роботи алгоритму конвертації, що також варто враховувати при оцінці ефективності подібного програмного рішення.

1.2 Ігровий рушій Open 3D Engine

Open 3D Engine (далі O3DE) [3] (рис. 1.3) - це безкоштовний ігровий рушій з відкритим вихідним кодом, розроблений групою Amazon Lumberyard.

Він був випущений у вересні 2021 року з метою надання інструментів розробникам ігор, щоб допомогти їм створювати ігри високої якості.

O3DE побудований на основі ряду відкритих технологій, таких як Amazon Lumberyard, CryEngine, Autodesk, NVIDIA і AWS. Рушій має повний набір функцій, який дозволяє розробникам створювати ігри для різних платформ, включаючи Windows, Linux, macOS, Android та iOS.

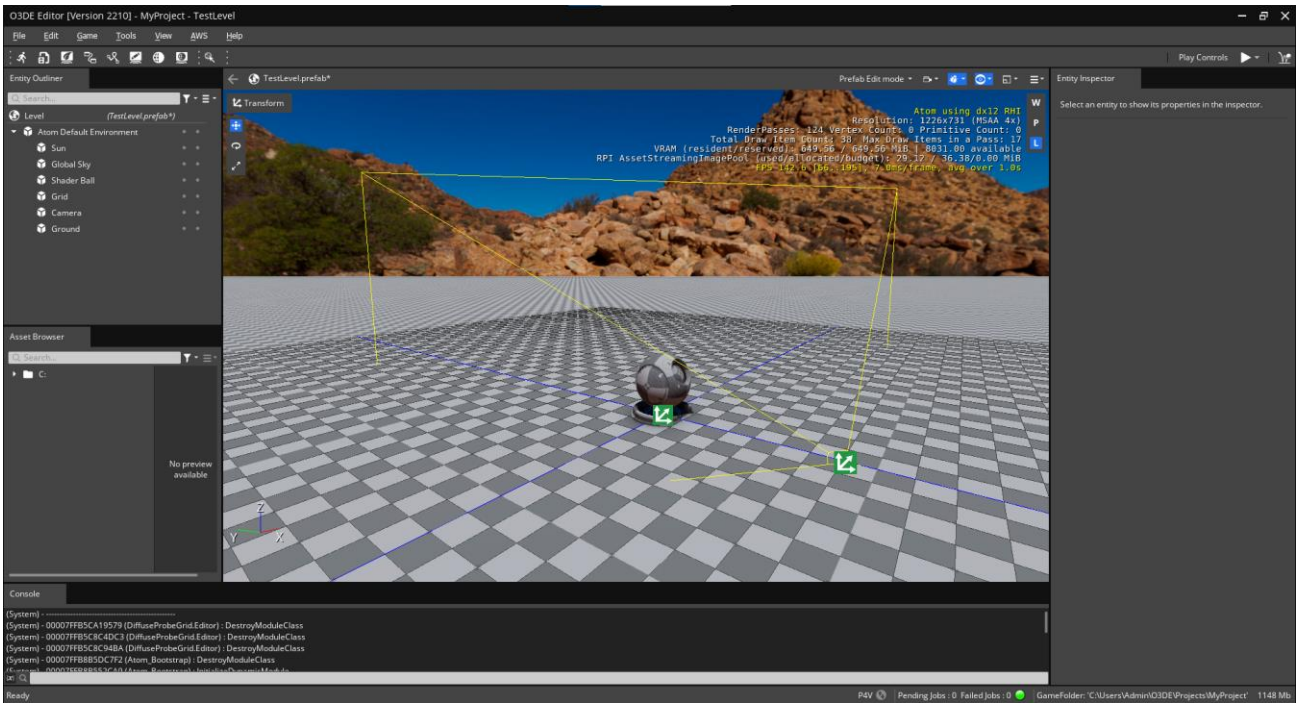


Рисунок 1.3 – Інтерфейс редактору O3DE

Серед ключових особливостей O3DE можна виділити широку підтримку фізики, включаючи розрахунки колізій, силу тяжіння, динамічні тіла та воду. Рушій також має вбудовану підтримку штучного інтелекту, зокрема системи штучного інтелекту з розподіленою обробкою.

Крім того, O3DE має готові рішення для різних аспектів гри, таких як управління персонажем, анімація, звукові ефекти та інші. Рушій також

підтримує різні формати файлів, що дозволяє розробникам ігор інтегрувати різноманітні ресурси, такі як моделі персонажів, текстури, звукові ефекти та інші.

За обробку та конвертацію цих ресурсів відповідає модуль під назвою AssetProcessor (рис. 1.4). Його основна задача це конвертація вхідних файлів у готові ресурси з урахуванням налаштувань, які можливо зберегти окремо для кожного з вхідних файлів.

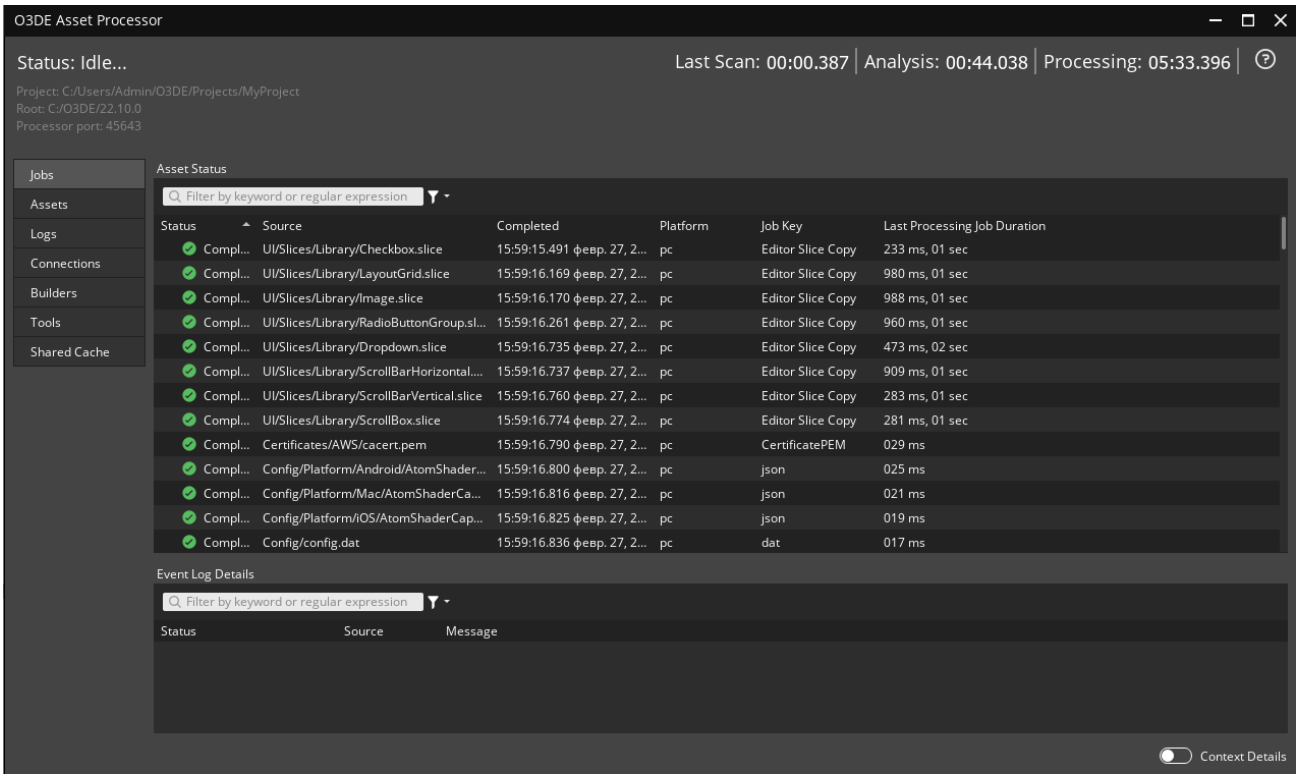


Рисунок 1.4 – Інтерфейс Asset Processor

Відповідно до цих ресурсів належать і DDS текстури. Для кожного зображення можна обрати один з готових пресетів для конвертації, або за потреби створити власний. Пресет (рис. 1.5) дозволяє налаштувати різні параметри конвертації для різних платформ, що суттєво розширює можливості для міжплатформної розробки.

При цьому варто зазначити, що AssetProcessor постійно спостерігає чи не змінився вхідний файл для одного з його ресурсів. У випадку якщо зміна зафіксована, він автоматично виконає конвертацію використавши обраний до цього присет. Це суттєво пришвидшує та автоматизує роботу користувача.

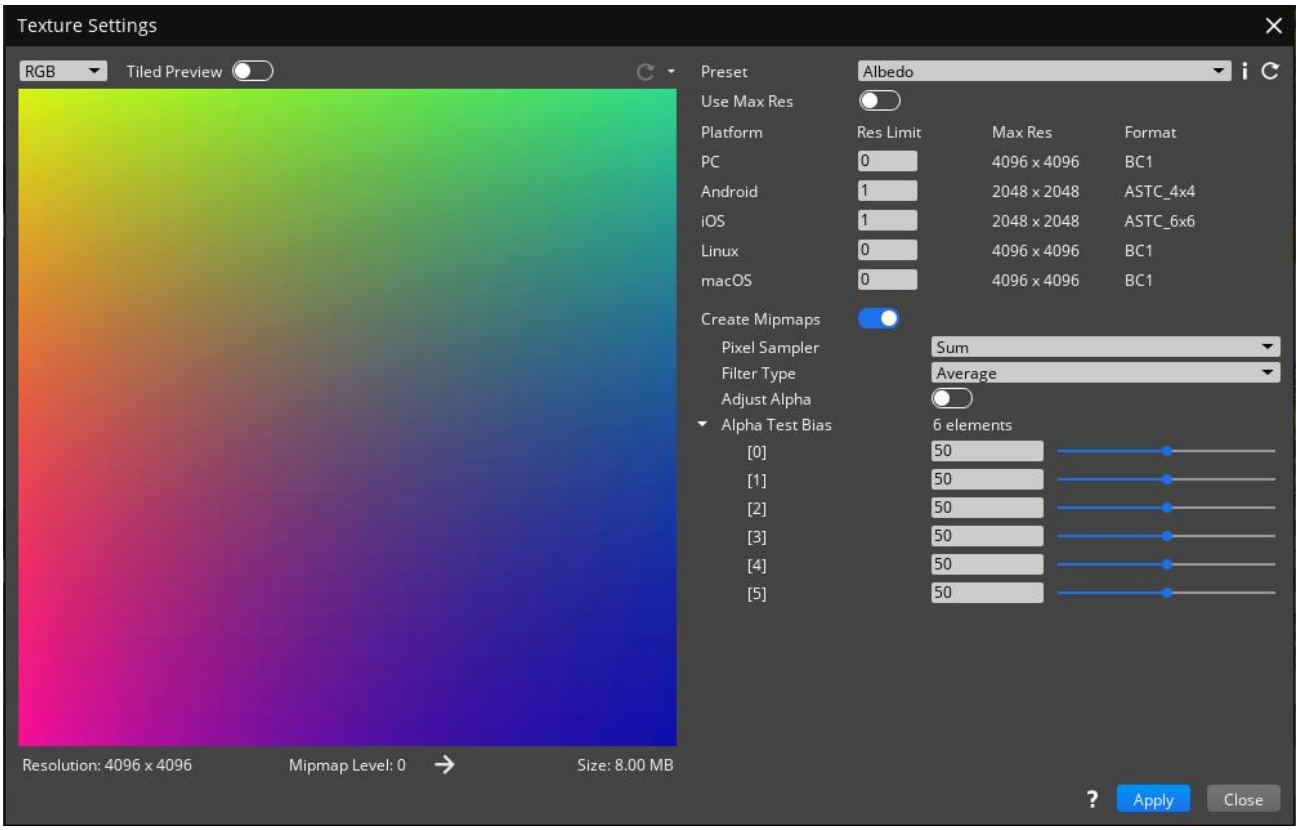


Рисунок 1.5 – Інтерфейс налаштування пресета для текстури

1.2.1 Переваги програмного засобу

Основна перевага використання AssetProcessor як складову частину O3DE, це високий рівень автоматизації процесів конвертації та інтеграції цих процесів у розробку. Користувач не є обмеженим у засобах якими він працює з зображеннями, при цьому результат його роботи буде автоматично конвертовано у текстуру без потреби встановлювати додаткове програмне

забезпечення, налаштовувати плагіни тощо. Більш того, якщо користувач безпосередньо не працює з зображеннями а лише отримує їх наприклад з системи контроля версій, то йому взагалі не є потрібним наявність жодного графічного редактору чи подібного програмного забезпечення на його робочому персональному комп'ютері. Після збереження файлів на диск, AssetProcessor автоматично їх конвертує і збереже результат.

1.2.2 Недоліки програмного засобу

Основний недолік даного програмного засобу полягає в тому, що користувач не може використовувати його модулі окремо. Тобто, якщо користувач зацікавлений у використані AssetProcessor для свого проєкту, це не є можливим якщо весь проєкт не розробляється використовуючи ігровий рушій Open 3D Engine. Це ж твердження буде справедливим і для решти конкурентів Open 3D Engine, або подібного програмного забезпечення направленого на вирішення інших задач.

Тобто не зважаючи на всі його суттєві переваги, використання AssetProcessor накладає суттєві обмеження на те для яких проєктів підходить це рішення. І в більшості випадків, користувачам доведеться шукати більш універсальне рішення, інтеграція якого буде можлива саме для їх проєкту.

Також варто зазначити, що AssetProcessor розрахований для конвертації великої кількості вхідних файлів різних типів у відповідні їм ресурси. Універсальність цього інструменту призводить до того, що впровадження конкретних оптимізацій для конвертації зображень у DDS текстури не є доцільним, оскільки такі оптимізації можуть негативно впливати на конвертацію ресурсів інших типів.

1.2.3 Аналіз швидкодії програмного засобу

Програмний засіб AssetProcessor надає досить детальну статистику (рис. 1.6) стосовно затраченого часу на конвертацію ресурсів, тому ми можемо отримати конкретні значення для тестового набору даних (розділ 1).

Status	Source	Completed	Platform	Job Key	Last Processing Job Duration
Completed	Folder_1/Benchmark_Image_6.png	17:51:23.559 февр. 27, 2023	pc	Image Compile: PNG	723 ms, 12 sec
Completed	Folder_1/Benchmark_Image_3.png	17:51:23.543 февр. 27, 2023	pc	Image Compile: PNG	718 ms, 12 sec
Completed	Folder_1/Benchmark_Image_8.png	17:51:23.487 февр. 27, 2023	pc	Image Compile: PNG	656 ms, 12 sec
Completed	Folder_1/Benchmark_Image_4.png	17:51:23.116 февр. 27, 2023	pc	Image Compile: PNG	283 ms, 12 sec
Completed	Folder_1/Benchmark_Image_5.png	17:51:23.053 февр. 27, 2023	pc	Image Compile: PNG	217 ms, 12 sec
Completed	Folder_1/Benchmark_Image_2.png	17:51:22.930 февр. 27, 2023	pc	Image Compile: PNG	102 ms, 12 sec
Completed	Folder_1/Benchmark_Image_1.png	17:51:22.637 февр. 27, 2023	pc	Image Compile: PNG	803 ms, 11 sec
Completed	Folder_1/Benchmark_Image_0.png	17:51:22.615 февр. 27, 2023	pc	Image Compile: PNG	802 ms, 11 sec
Completed	Folder_1/Benchmark_Image_7.png	17:51:22.381 февр. 27, 2023	pc	Image Compile: PNG	543 ms, 11 sec
Completed	Folder_1/Benchmark_Image_9.png	17:51:22.211 февр. 27, 2023	pc	Image Compile: PNG	398 ms, 11 sec

Рисунок 1.6 – Результат конвертації тестового набору даних

Колонка Last Processing Job Duration (рис. 1.6) відображає скільки часу пройшло з моменту як прийшов запит на конвертацію зображення у текстуру до моменту як конвертація була успішно виконана. Тобто час конвертації усього набору тестових даних це найбільший час виконання одної такої конвертації, тобто 12.723 секунди.

Варто зазначити, що під час експерименту AssetProcessor використовував стандартні налаштування, тобто використовував кількість потоків, що співпадає з кількістю логічних ядер центрального процесора.

1.3 NVIDIA Texture Tools Exporter

NVIDIA Texture Tools Exporter [4] (рис. 1.7) - це додаток який дозволяє розробникам відеоігор та інших графічних додатків оптимізувати та обробляти текстури, що використовуються в їх проектах.

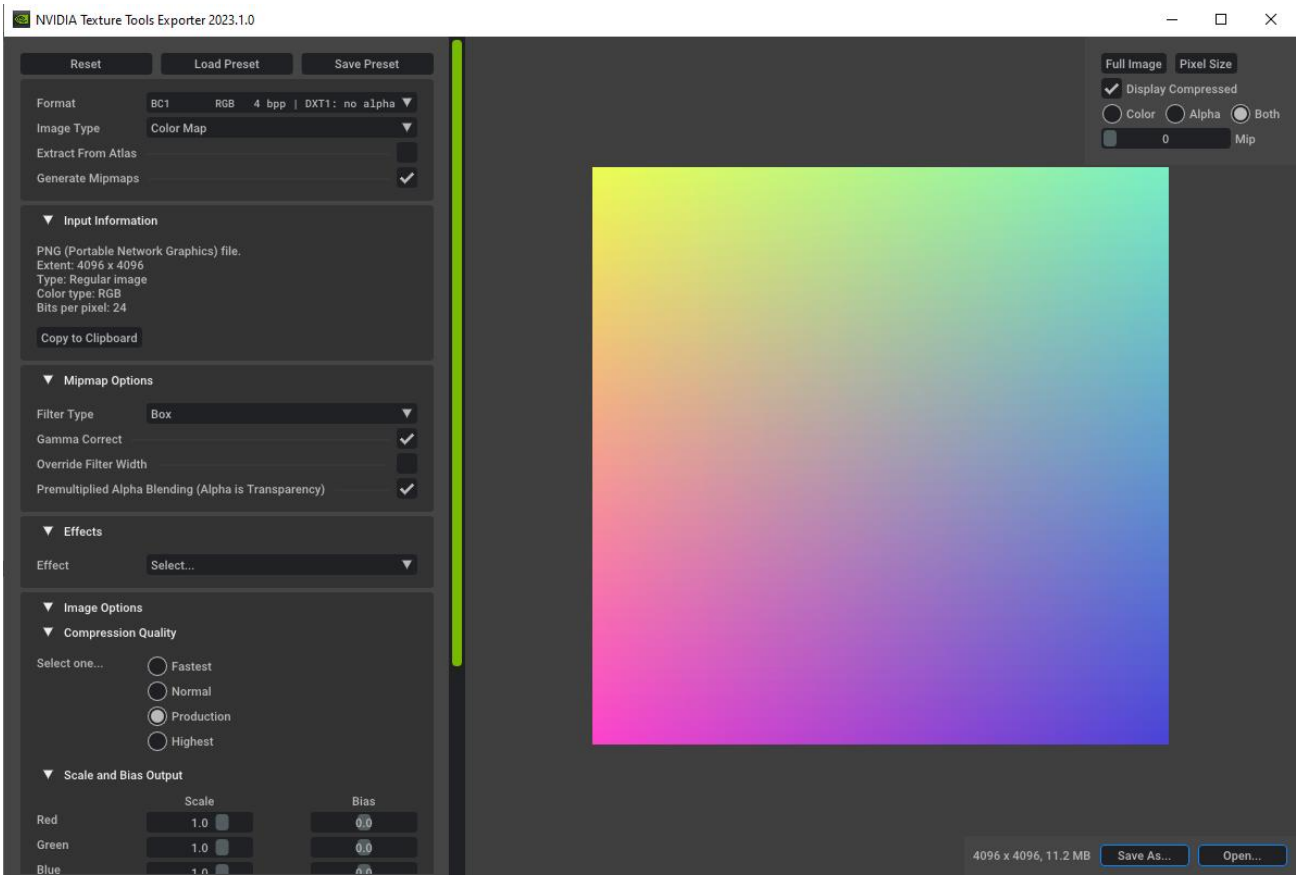


Рисунок 1.7 – Інтерфейс NVIDIA Texture Tools Exporter

Основні можливості NVIDIA Texture Tools Exporter включають:

– обробка текстур - програма дозволяє обробляти текстури з різними форматами, включаючи PNG, JPEG, TIFF, TGA, PSD та інші. Завдяки цьому, розробники можуть збільшувати роздільну здатність та розмір текстур, оптимізувати їх для конкретних графічних рушіїв, видаляти шум та інші дефекти;

– стиснення текстур - програма підтримує різні алгоритми стиснення, такі як BC1, BC2, BC3 та інші, що дозволяє зменшити розмір текстур без видимої втрати якості зображення;

– конвертація форматів - додаток підтримує конвертацію текстур з одного формату в інший, що дозволяє розробникам працювати з різними форматами в одному проєкті;
– масштабування текстур - програма дозволяє масштабувати текстури, збільшувати або зменшувати їх роздільну здатність.

NVIDIA Texture Tools Exporter є потужним інструментом для оптимізації та обробки текстур, що використовуються в різних графічних проектах. Додаток дозволяє розробникам швидко та ефективно працювати з текстурами, зменшуючи час, необхідний для їх розробки та оптимізації.

Також програмний засіб дозволяє зберігати налаштування для текстур у пресети і повторно їх використовувати для багатьох зображень. Це дозволяє оптимізувати роботу над однотипними зображеннями оскільки користувачу буде достатньо один раз знайти той набір налаштувань, який найкраще вирішує поставлену перед ним задачу.

Ще одна особливість NVIDIA Texture Tools Exporter це використання графічного процесору для оптимізації конвертації зображень за допомогою набору інструментів NVIDIA CUDA Toolkit [5]. Це дозволяє суттєво пришвидшити час необхідний на конвертацію зображень порівняно з її виконанням на центральному процесорі.

Якщо користувачу потрібно конвертувати одночасно декілька зображень, для цього існує консольна утиліта nvbatchcompress (рис. 1.8), яка інсталюється разом з NVIDIA Texture Tools Exporter.

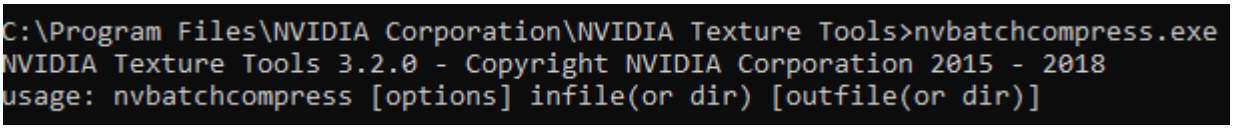


Рисунок 1.8 – Опис консольної утиліти nvbatchcompress

Консольна утиліта має усі ті самі налаштування, що і версія з користувацьким графічним інтерфейсом. Вона може приймати як аргумент шлях як до конкретного зображення так і до каталогу з зображеннями. Проте варто зауважити, що при конвертації зображень, до кожного з них будуть застосовуватися одні й ті самі налаштування.

1.3.1 Переваги програмного засобу

Основна перевага даного програмного засобу це велика кількість налаштувань, яку він пропонує користувачу. Це дозволяє оптимізувати текстури під вимоги конкретного проєкту, та підібрати оптимальне співвідношення між якістю текстури та її фізичним розміром.

Також за допомогою утиліти nvbatchcompress програмний засіб пропонує зручний спосіб конвертувати одночасно велику кількість зображень. Більш того те, що ця утиліта працює в консольному режимі, дає можливість користувачу використати її у власних сценаріях виконання, для ще більшої автоматизації власної роботи, а відповідно зменшенню витрат часу на рутинні задачі.

Ще однією перевагою є використання графічного процесору для конвертації зображень у текстури. Обчислювальні можливості сучасних графічних процесорів набагато перевищують центральні процесори, особливо в задачах з великою кількістю однотипних математичних операцій. Також при використанні графічного процесору, інші фонові процеси користувача можуть продовжувати своє виконання на центральному процесорі, а отже можна стверджувати, що ресурси системи в цілому використовуються більш ефективно.

1.3.2 Недоліки програмного засобу

Основний недолік даного програмного засобу полягає у його системних вимогах. Для інсталяції даного програмного забезпечення, користувач має використовувати 64 бітну операційну систему Windows 10 або Windows 11 [4]. Окрім цього для використання прискорення конвертації за допомогою графічного процесору, користувач повинен мати у своєму розпоряджені графічний процесор виробництва компанії NVIDIA, який має підтримувати актуальну версію Cuda. Подібні графічні процесори є досить дорогим обладнанням, а тому кількість користувачів та проєктів які могли б скористатися цими можливостями є досить обмеженою.

Окрім цього, для отримання доступу до NVIDIA Texture Tools Exporter, користувач має зареєструватися у програмі NVIDIA Developer Program [6]. В свою чергу це може накладати певні юридичні обмеження на використання даного програмного забезпечення в комерційних цілях, що робить його непридатним для використання в деяких проєктах.

1.3.3 Аналіз швидкодії програмного засобу

При використанні утиліти nvbatchcompress для обробки кількох зображень одночасно, користувач в результаті отримує інформацію про сумарний час витрачений на конвертацію зображень у тектури. Це дозволяє отримати конкретні значення швидкодії програмного засобу для тестового набору даних (розділ 1).

При використанні прискорення конвертації за допомогою графічного процесору, час конвертації для тестового набору даних склав 2.946 секунди (рис. 1.9).

```
NVIDIA Texture Tools 3.2.0 - Copyright NVIDIA Corporation 2015 - 2018
CUDA acceleration ENABLED

Compressing the following files:
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_0.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_1.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_2.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_3.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_4.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_5.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_6.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_7.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_8.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_9.png

time taken: 2.946 seconds
```

Рисунок 1.9 – Результат роботи утиліти nvbatchcompress з використанням графічного процесору

Проте, оскільки для великої кількості користувачів та проєктів дана можливість відсутня, більш актуальним є час конвертації тестового набору даних лише за допомогою центрального процесору. В такому випадку час конвертації склав 11.400 секунди (рис. 1.10).

```
NVIDIA Texture Tools 3.2.0 - Copyright NVIDIA Corporation 2015 - 2018
CUDA acceleration DISABLED

Compressing the following files:
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_0.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_1.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_2.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_3.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_4.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_5.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_6.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_7.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_8.png
C:\Users\Admin\Desktop\Image_Benchmark\Folder_1\Benchmark_Image_9.png

time taken: 11.400 seconds
```

Рисунок 1.10 – Результат роботи утиліти nvbatchcompress без використання графічного процесору

1.4 Постановка задачі

Реалізація у вигляді плагіну для графічного редактору (розділ 1.1) не підходить для проєктів для яких важлива швидкість конвертації зображень, а також можливість конвертувати їх певну кількість одночасно. Використання даного рішення є виправданим лише у випадку коли графічний редактор є основним інструментом роботи спеціаліста.

Реалізація у вигляді окремого модулю ігрового рушія (розділ 1.2) продемонструвала себе найкраще з точки зору автоматизації роботи користувача. Проте оскільки неможливо використовувати лише один з модулів програмного забезпечення подібного типу, це рішення можливо застосувати лише до обмеженої кількості проєктів.

Реалізація у вигляді окремої утиліти (розділ 1.3) є найбільш універсальним варіантом реалізації програмного засобу для конвертації зображень у DDS текстури. Подібне рішення є простим у використанні, та дозволяє у зручній формі виконувати конвертацію великої кількості зображень одночасно.

Аналізуючи результати швидкодії програмних засобів, можна зазначити, що AssetProcessor (розділ 1.2.3) та утиліта nvbatchcompress (розділ 1.3.3) продемонстрували подібні результати, проте nvbatchcompress все ж виявився більш ефективним. Звичайно nvbatchcompress з використанням графічного процесору показав набагато кращий результат, проте через обмежений доступ користувачів до графічних процесорів, які задовольняють вимоги даної утиліти, це можна розглядати лише як допоміжний інструмент.

Метою роботи та подальшого дослідження є підвищення ефективності програмного забезпечення для конвертації зображення у DDS текстури шляхом

покращення його швидкодії. Для досягнення мети необхідно вирішити наступні задачі:

- аналіз існуючих засобів для написання ефективного багатопоточного коду;
- розробка методу для написання ефективного багатопоточного коду;
- реалізація програмного засобу для конвертації зображень у текстури на основі розробленого методу;
- оцінка ефективності запропонованого рішення.

Розроблений програмний засіб повинен відповідати наступним вимогам:

- програмний засіб повинен мати кращі показники швидкодії за існуючі методи;
- програмний засіб повинен ефективно оброблювати велику кількість вхідних зображень одночасно;
- програмний засіб повинен підтримувати найбільш вживані алгоритми конвертації зображень у текстури та мати відповідні налаштування;
- програмний засіб має надавати можливість переглянути отриманий результат без застосування стороннього програмного забезпечення.

Висновки до розділу

У даному розділі був проведений аналіз існуючих рішень для вирішення задачі конвертація зображень у DDS текстури. Було виділено наступні категорії програмних засобів згідно способу їх використання: плагін для графічного редактору, окрема утиліта та як окремий модуль програмного забезпечення, що використовує двовимірну або тривимірну графіку, наприклад ігровий рушій або програмне забезпечення для тривимірного моделювання.

Був проведений аналіз програмних засобів, кожен з яких є одним з найбільш просунутих прикладів програмних засобів зазначених вище категорій. В результаті аналізу переваг та недоліків існуючих рішень, були сформульовані постановка задачі та вимоги до програмного засобу.

2 МЕТОД АВТОМАТИЗАЦІЇ РОЗРОБКИ БАГАТОПОТОЧНОЇ ПРОГРАМИ

Використання механізму багатопочності є одним з основних методів оптимізації програмного забезпечення. Розглянуті у попередньому розділі рішення (розділ 1) використовують механізм багатопоточності щоб максимально пришвидшити процес конвертації зображень у текстури. При цьому по замовчуванню використовуючи усі наявні в системі ресурси. Проте ефективність конкретної реалізації може залежати від багатьох факторів, таких як управління ресурсами всередині програмного засобу, використання різних засобів синхронізації між потоками, рівномірність розподілу навантаження між потоками тощо.

Оскільки метою дослідження є покращення швидкодії відносно існуючих засобів конвертації зображень у DDS текстури, доцільним є розробка методу який дозволить автоматизувати написання багатопоточного коду [7]. Це дозволить як розробити програмний засіб, який зможе адаптуватися під різні набори вхідних даних, так і з мінімальними витратами часу експериментальним шляхом перевіряти гіпотези, стосовно оптимізації часу на виконання конвертації.

2.1 Обґрунтування вибору мови програмування

Для розробки та реалізації методу була обрана мова програмування C++. Мова програмування C++ є стандартним вибором для програмного забезпечення націленого на швидкодію. C++ має низький рівень абстракції і може працювати безпосередньо з пам'яттю та процесором, що дозволяє досить точно контролювати розподілення ресурсів та оптимізувати код для досягнення максимальної продуктивності. Також C++ підтримує безпосереднє створення та керування багатьма потоками. Крім того, C++ надає можливість створювати

потокобезпечний код, використовуючи механізми, такі як блокування даних та атомарні операції. Окрім цього, мова програмування C++ дуже розповсюджена для програмного забезпечення з використанням двовимірної та тривимірної графіки. З точки зору бізнесу простіше та дешевше підтримувати програмні засоби написані на одній і тій самій мові програмування, оскільки це не вимагатиме залучення додаткових спеціалістів.

2.2 Огляд засобів для багатопоточної розробки мовою C++

2.2.1 C++ Standard Library

Сучасні мови програмування пропонують різноманітні засоби для написання багатопоточного коду і мова програмування C++ не є виключенням. Інженер має в своєму розпорядженні усі стандартні примітиви синхронізації такі як mutex, semaphore, condition variables та atomic типи даних [8]. Також наявні інтерфейс для роботи з потоками та алгоритми паралельних обчислень [9].

Основною перевагою використання стандартної бібліотеки є те, що вона надає широкі можливості розробити код, який найкраще підходить для вирішення задачі, поставленої перед інженером програмного забезпечення. Для цього інженеру потрібно провести детальний аналіз програмного забезпечення, виявити місця програми, виконання яких в кількох потоках дасть найбільший приріст швидкодії, та впровадити оптимізації, які будуть найбільш ефективними для розроблюваного програмного продукту.

Основний недолік бібліотеки є наслідком з її переваги. Якщо створюється унікальне рішення і фактично з нуля, то цілком ймовірно, що це буде найбільш витратне по ресурсам рішення і найдорожче з позиції бізнесу. До недоліку можна віднести також те, що можливості використання багатопоточності у C++

Standard Library з’явились лише у стандарті C++11 [10], тому, якщо в проєкті присутні модулі, які не підтримують цей стандарт, то використання C++ Standard Library не є можливим.

2.2.2 Boost

Boost – це, напевно, найпопулярніша колекція бібліотек для багатопоточного програмування мовою C++. Її виникнення пов'язане з тим, що до стандарту C++ були включені не всі бібліотеки, які вважалися багатьма розробника необхідними. Зараз бібліотеки Boost позиціонуються як розширення до стандартної бібліотеки і залишаються актуальними не зважаючи на постійні оновлення стандарту C++.

Перевага Boost полягає в тому, що деякі проблеми, які виникають під час розробки багатопоточних програм, цими бібліотеками вже вирішені, на відміну від стандартної бібліотеки, й інженеру не доведеться витрачати на їх вирішення час. Для прикладу, функції стандартної бібліотеки rand або ctime, не є безпечними для використання у багатопоточному середовищі, оскільки вони зберігають внутрішній стан що не синхронізується між потоками. Водночас, бібліотеки Boost гарантують безпечність їх використання [11]. До переваг Boost можна віднести також те, що вони підтримуються навіть досить ранніми версіями компіляторів. Популярність цієї колекції бібліотек обумовлює також високу ймовірність того, що він може бути інтегрований у проєкт, що розробляється.

Основний недолік використання бібліотеки Boost є аналогічним стандартній бібліотеці. Не зважаючи на те, що певні проблеми, які постануть перед розробником в ньому вже вирішені, ця бібліотека орієнтована на розробку програми з нуля і розробка все ще буде достатньо дорогою з позиції

бізнесу. Також варто зазначити, що у проєктах використовується різна політика стосовно додавання сторонніх бібліотек, тому, не зважаючи на популярність, багато проєктів принципово відмовляються від використання Boost на користь використання виключно стандартної бібліотеки.

2.2.3 Concurrencpp

Concurrencpp - бібліотека для розробки багатопоточних програм мовою C++ з відкритим вихідним кодом [12], що дає змогу писати код без використання низькорівневих примітивів таких, як lock або conditional variable, замінюючи їх високорівневими абстракціями. Бібліотека має багато переваг. Фактично, вона пропонує готове рішення для написання багатопоточного коду, беручи на себе функції синхронізації даних та безпечного виконання коду у потоках. Використання цієї бібліотеки здатне суттєво підвищити швидкість розробки і відповідно зробити її більш дешевою, оскільки інженеру програмного забезпечення не потрібно буде вирішувати низькорівневі проблеми, а достатньо сфокусуватися на високорівневій архітектурі коду. Окрім того результуючий код буде простіше підтримувати і він буде містити менше дефектів.

Проте бібліотека Concurrencpp не позбавлена недоліків. Вона має досить обмежені можливості стосовно її модифікації під потреби розробника, що може призвести до потреби у написанні додаткових обгорток або проксі аби застосувати дану бібліотеку для конкретного програмного забезпечення.. Також бібліотека пропонує лише досить примітивний механізм встановлення залежностей між задачами, які виконуються в потоках, і в досить обмеженому вигляді дозволяє обмін даними між залежними задачами. До того ж бібліотека

має досить значні вимоги до версії компіляторів та стандарту C++, що робить її інтеграцію неможливою для великої кількості проєктів.

2.3 Формування вимог до методу

Спираючись на огляд існуючих рішень, на їх переваги і недоліки (розділ 2.2), можна сформувати наступні функціональні та нефункціональні вимоги до методу автоматизації розробки багатопоточної програми. Основна функціональна вимога - ефективність використання наявних обчислювальних ресурсів. Тобто, має бути забезпечено ефективне використання таких ресурсів як потоки, при цьому час простою системи має бути мінімальним [13]. На додачу до цього метод повинен, за потреби, надати розробнику можливість гарантувати порядок виконання задач відносно одна одної, тобто встановити між ними залежності. Виконання цієї умови дасть змогу застосувати даний метод автоматизації для більшої кількості проєктів, для яких не достатньо рішень запропонованих класичними шаблонами проєктування.

Серед нефункціональних вимог можна виділити наступне. По-перше, реалізація методу має базуватися на стандартній бібліотеці C++. Не зважаючи на те, що для доступу до компонентів для багатопоточної розробки необхідна підтримка стандарту C++11, ця вимога виконується для набагато більшої кількості проєктів, ніж вимоги для Concurrencpp (розділ 2.2.3) і взагалі не буде потребувати інтеграції сторонніх бібліотек. По-друге, метод має інкапсулювати всередині себе низькорівневий код, включаючи примітиви синхронізації, та надавати розробнику можливість використовувати його інтерфейс як високорівневу абстракцію [14]. При цьому має бути підримка кастомізації під потреби розробника, а також механізми, які дають змогу налаштувати залежності між задачами, що виконуються, а також обмін даними між ними.

2.4 Реалізація методу

Для початку введемо наступні абстракції. Абстрактний клас Job (рис. 1) буде замінювати собою об’єкт класу function, який зазвичай передається в якості аргументу для виконання в потоці. Це дає можливість розробнику застосовувати принципи об’єктно-орієнтованого програмування для побудови більш складної логіки виконання задач, а також реалізації механізму залежностей між екземплярами класу Job, або його нащадками. Абстрактний клас JobData агрегується всередині класу Job та зберігає стан, який може бути спільним для кількох задач, а також бути використаним для отримання результату після виконання задач. Лістинг абстрактного класу Job приведений нижче.

```
    class Job
    {
    public:
        virtual ~Job() = default;
        void Start();
        void  AddJobData(const  std::shared_ptr<JobData>&
jobData);
        JobData* GetJobData();
        virtual ThreadAffinity GetAffinity();
        void AddDependency(Job* pJob);
        bool IsReady();
        bool IsSuccessful();

    protected:
        virtual void Execute() = 0;

    private:
        void AddDependentJob(Job* pJob);
        void OnFinish();

    private:
        bool m_bSuccessful { false };
        std::atomic_int m_dependeciesCounter { 0 };
```

```
        std::vector<Job*> m_dependentJobs;
        std::shared_ptr<JobData> m_jobData;
    };
```

Механізм залежностей дозволяє розробнику задати відносний порядок виконання задач. Цей механізм необхідний у випадках, коли виконання одної з задач має сенс лише після виконання певного набору інших задач. Тобто, коли говоримо, що задача Job A має залежність на задачу Job B, то мається на увазі, що Job A може бути виконана тоді і тільки тоді, коли Job B успішно завершила своє виконання. Тобто метод додає поведінку, якої можна було б досягнути, застосувавши механізм join. Проте суттєва відмінність від join полягає в тому, що дана реалізація не є блокуючою, тобто вона не блокує поточний потік в очікуванні завершення виконання іншого потоку. Більш того, якщо потік повністю завершив свою роботу, що є вимогою для спрацювання join, він знищується і не може бути використаний повторно. Запропонований метод натомість буде шукати альтернативні задачі, які можна виконувати в даний час, при цьому повторно використовуючи вже створений потік, що дозволяє більш ефективно використовувати ресурси та збільшити швидкодію програмного забезпечення.

Для візуалізації механізму залежностей використаємо граф. Припустимо розробник виділив шість різних задач: Job A, Job B, Job C, Job D, Job E, Job F. Задачі мають наступні залежності: Job B та Job C залежать від Job A; Job D залежить від Job B; Job E залежить від Job C; Job F залежить від Job D та Job E (рис. 2.1). Дуги графу відображають порядок, в якому мають бути виконані задачі. Не можливо гарантувати, яка з задач Job D чи Job E виконається раніше (оскільки вони виконуються паралельно), проте це і не важливо, оскільки завдання механізму залежностей – гарантувати, що Job F почне своє виконання лише після них обох.

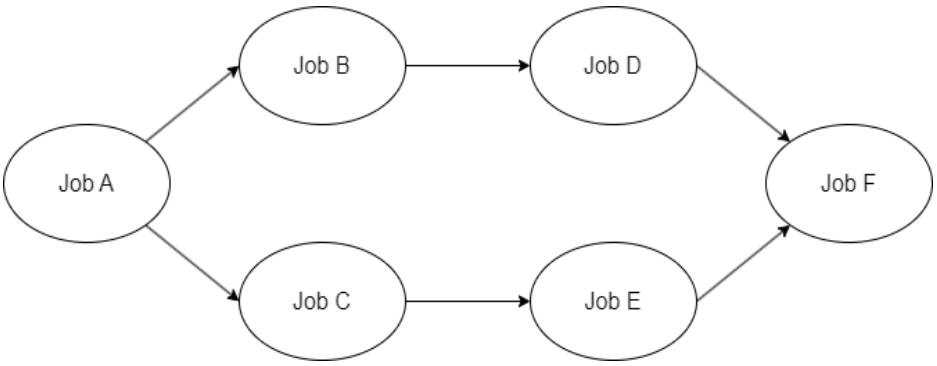


Рисунок 2.1 – Граф залежностей між задачами

Наступним кроком реалізації методу є додавання механізму управління потоками та розподілу задач. Згідно сформованих вимог (розділ 2.3) цей механізм має інкапсулювати в собі низькорівневі механізми синхронізації, та надавати розробнику високорівневий інтерфейс для взаємодії. Основою розробки є шаблон проєктування Thread Pool [15], перевагою якого є те, що потоки створюються один раз на початку створення пулу і знищуються після завершення його роботи. Це дозволяє оптимізувати використання ресурсів у випадку коли за один запуск програмне забезпечення виконує велику кількість задач.

Проте залишається нерозв’язаною проблема, коли потрібно налаштувати середовище таким чином, щоб не кожен потік міг виконувати будь-яку задача. Наприклад є потреба постійно виконувати операції запису на диск і для уникнення проблем синхронізації є необхідність зробити так, щоб ці задача міг виконувати лише один конкретний потік. Для вирішення цієї проблеми реалізується механізм «схильності» (англ. Affinity) потоків і задач.

Ідея цього механізму полягає в тому, що інженер програмного забезпечення на етапі ініціалізації пулу потоків має можливість створити потік, який буде виконувати або виключно задачі певного типу, або надавати їм

пріоритет у виконанні, але за їх відсутності продовжувати виконувати інші задачі. Кожен клас задач має можливість задати до якого типу вони належать, а також чи можливо виконувати цю задачу за відсутності спеціалізованого потоку.

Нижче представлений лістинг основної функції, яка виконується в кожному з потоків.

```
void JobManager::ThreadLoop(const ThreadConfiguration
&configuration)
{
    while (true)
    {
        std::shared_ptr<Job> job = nullptr;
        {
            std::unique_lock<std::mutex>
lock(m_queueMutex);
            job = GetRunnableJob(configuration);

            if (!job && m_bShouldTerminate && m_jobsCount
== 0)
            {
                return;
            }
        }

        if (job)
        {
            job->Start();
        }
        else
        {
            std::this_thread::yield();
        }
    }
}
```

Враховуючи механізми описані до цього, класичний спосіб упорядкування задач в черзі пулу потоків за принципом «перший увійшов, першим вийшов», має бути модифікований, щоб враховувати механізм залежностей між задачами та їх схильностей. Функція GetRunnableJob знаходить задачу, для якої на момент перевірки вже були виконані усі її залежності, і яку може виконувати вільний потік з урахуванням схильності як задачі так і самого потоку. Якщо таких задач є декілька, виконається та, яка була додана у чергу раніше.

Доповнимо схему залежностей між задачами (рис. 2.1), додавши візуалізацію роботи цього механізму в певний момент часу. Припустимо метод використовує два потоки, і задачі Job A, Job C вже були виконані, а задача Job B виконується зараз в потоці Thread 1. Тоді стан задач матиме наступний вигляд (рис. 2.2).

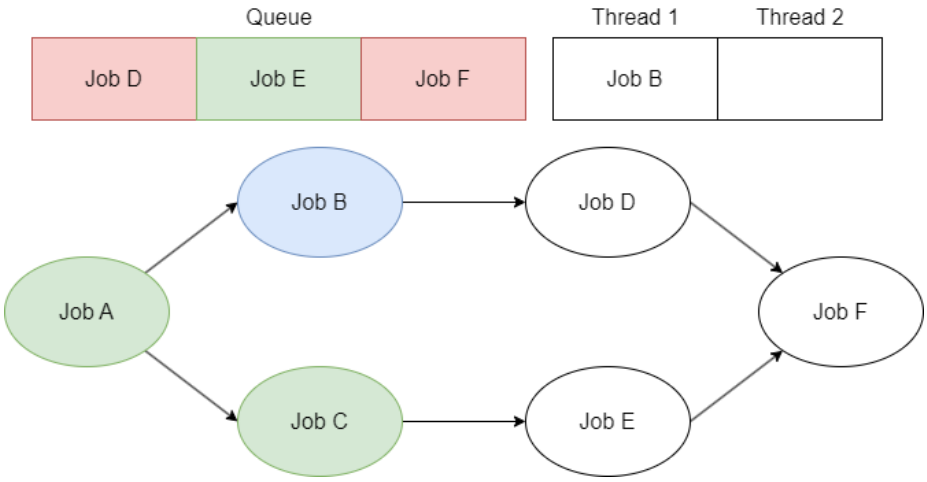


Рисунок 2.2 – Візуальне представлення стану системи

На схемі (рис. 2.2) зеленим кольором позначені задачі, які вже були виконані, а синім – ті, що виконуються в даний момент. У черзі червоним позначені задачі, залежності яких все ще не виконані і їх неможливо запустити в цей момент часу, а зеленим - відповідно ті, які готові до запуску. Як ми можемо

побачити, для такого стану, не зважаючи на те, що задача Job D була додана в чергу раніше, через те, що її залежність все ще в процесі, ми не можемо почати її виконувати, тому буде обрано задачу Job E для виконання в потоці Thread 2.

Висновки до розділу

Розроблено метод автоматизації розробки багатопоточного програмного забезпечення мовою C++, який найкраще демонструє себе у проєктах, де необхідно мати можливість виконувати задачі в певній послідовності, при цьому оптимально використовуючи наявні ресурси програми. Для розробки методу були досліджені існуючі рішення, виділені їх основні переваги та недоліки.

Запропонований метод автоматизації розробки багатопоточного програмного коду мовою C++ за допомогою високорівневих абстракцій дає змогу розбити процес на окремі задачі та виконати їх асинхронно з урахуванням встановлених залежностей. При цьому для роботи методу достатньо підтримки проєктом стандарту C++11 і немає потреби в інтеграції інших сторонніх бібліотек.

На основі даного методу буде розроблено програмний засіб з метою покращення швидкодії конвертації зображень у DDS текстури порівняно з існуючими рішеннями.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір середовища розробки

Для розробки програмного забезпечення був обраний фреймворк Qt [16] та інтегроване середовище розробки Qt Creator (рис. 3.1). Qt - це потужний фреймворк, який був створений для розробки кросплатформних додатків з використанням мови програмування C++. Qt забезпечує розширені можливості для розробки графічних інтерфейсів користувача, мережевого програмування, роботи з базами даних, роботи зі звуком і відео, а також забезпечує безпеку та захист даних.

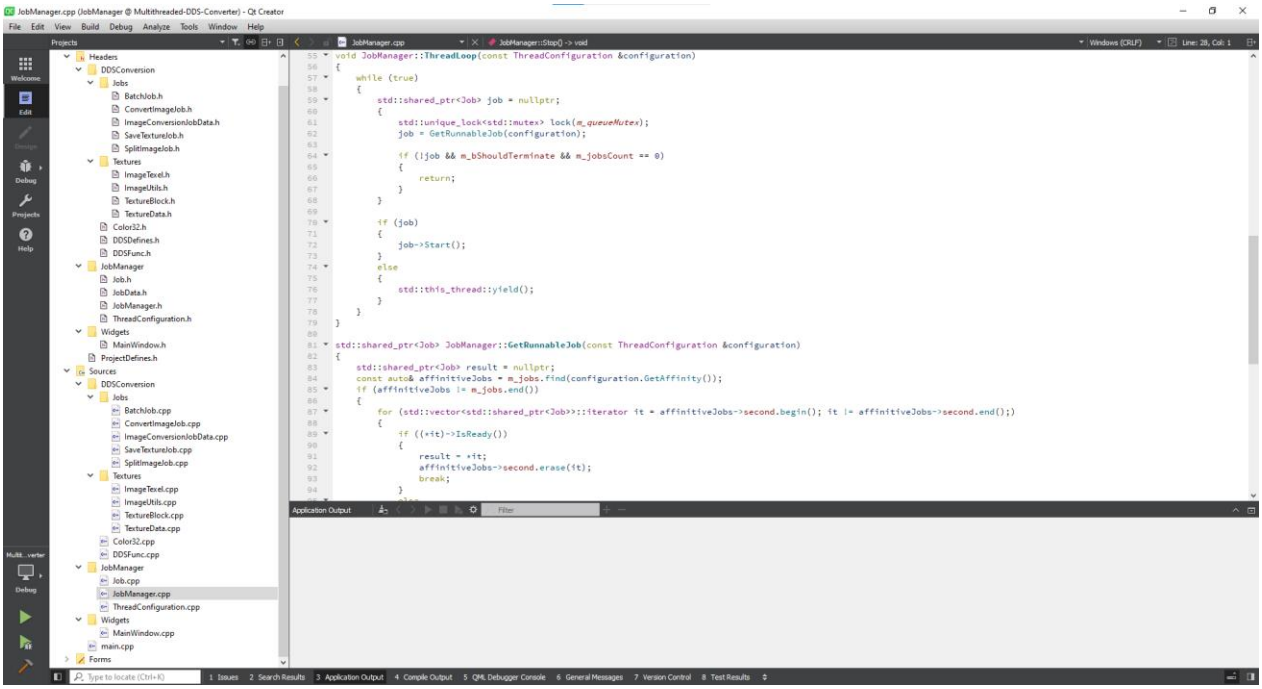


Рисунок 3.1 – Інтегроване середовище розробки Qt Creator

Qt є одним з найбільш популярних фреймворків для розробки додатків на C++. Серед його основних переваг можна виділити наступне:

- кросплатформність: Qt може бути використаний для розробки програм на різних платформах, таких як Windows, MacOS, Linux та інші;
- готовий інструментарій: Qt поставляється з готовим інструментарієм, який включає в себе інструменти для розробки графічних інтерфейсів, роботи з мережами та базами даних, що дозволяє розробникам зосередитись на розробці функціоналу програми, а не на написанні власних інструментів;
- простота використання: Qt має простий і легкий в освоєнні API, що дозволяє розробникам швидко створювати додатки з графічним інтерфейсом та іншими функціями;
- велика спільнота: Qt має велику спільноту розробників, яка постійно підтримує і оновлює фреймворк, що забезпечує швидкий розвиток та підтримку для нових платформ та технологій;
- можливості розширення: Qt має великий набір розширень та модулів, що дозволяє розробникам використовувати його для різних типів додатків, включаючи настільні програми, мобільні додатки тощо.

Також варто зазначити, що Qt має вбудовану підтримку графічного API OpenGL, що дає можливість реалізувати перегляд текстур за допомогою їх відмалювання графічним процесором, тобто в саме такому вигляді як би їх побачив кінцевий користувач. Також це важливо для перевірки правильності роботи алгоритмів конвертації, оскільки якщо при реалізації алгоритмів були допущені помилки, вичитування текстури і відображення у вигляді масиву пікселів може їх приховати.

3.2 Розробка архітектури програмного забезпечення

Для того щоб ефективно застосувати розроблений метод для побудови багатопоточноного програмного забезпечення (розділ 2.4) необхідно провести аналіз процесу конвертації зображення у текстури.

Весь процес можна розділити на три основні етапи (рис. 3.2):

– попередня обробка зображення;
– застосування алгоритму конвертації;
– збереження результату у вигляді DDS текстури.

Рисунок 3.2 – UML діаграма діяльності загального алгоритму конвертації зображення у текстуру

Очевидним є твердження, що в процесі конвертації одного зображення, виділені етапи мають виконуватися послідовно. Окрім цього кожен наступний етап повинен мати доступ до результату виконання попереднього. При цьому різні вхідні зображення можуть бути опрацьовані паралельно, незалежно один від одного.

В контексті розробленого раніше методу (розділ 2.4) кожен з цих етапів може бути представленим у вигляді певного класу-нащадка абстрактного класу Job. Реалізований механізм залежностей між задачами дозволить гарантувати те, що етапи обробки одного вхідного зображення будуть виконані послідовно, при цьому дозволяючи конвертувати одночасно декілька зображень. В той же час за допомогою абстрактного класу JobData, задачі в рамках одного вхідного зображення можуть мати спільний стан, без потреби копіювання даних між етапами.

3.3 Попередня обробка зображень

Попередня обробка зображення є першим етапом конвертації зображення у текстуру. Цей етап є необхідним тому, що вхідне зображення необхідно перетворити у масив даних, який далі буде сумісним з алгоритмами блочного стиснення текстур. Зазвичай вхідне зображення зберігається у певному форматі, найрозповсюдженіші з яких це PNG, TIFF, JPEG тощо. Фреймворк Qt (розділ 3.1) має вбудовану підтримку усіх найбільш вживаних форматів зображень, та за допомогою класу QImage [17] дозволяє представити їх у вигляді масиву пікселів.

Проте, алгоритми блочного стистення текстур (англ. Block Compression Algorithms, BC) працюють не з окремими пікселями з блоками пікселів розміром чотири на чотири. Такий блок має назву тексель (рис. 3.3). Тобто перш

ніж приступити до виконання відповідного алгоритму, масив пікселів необхідно перетворити на масив текселів.

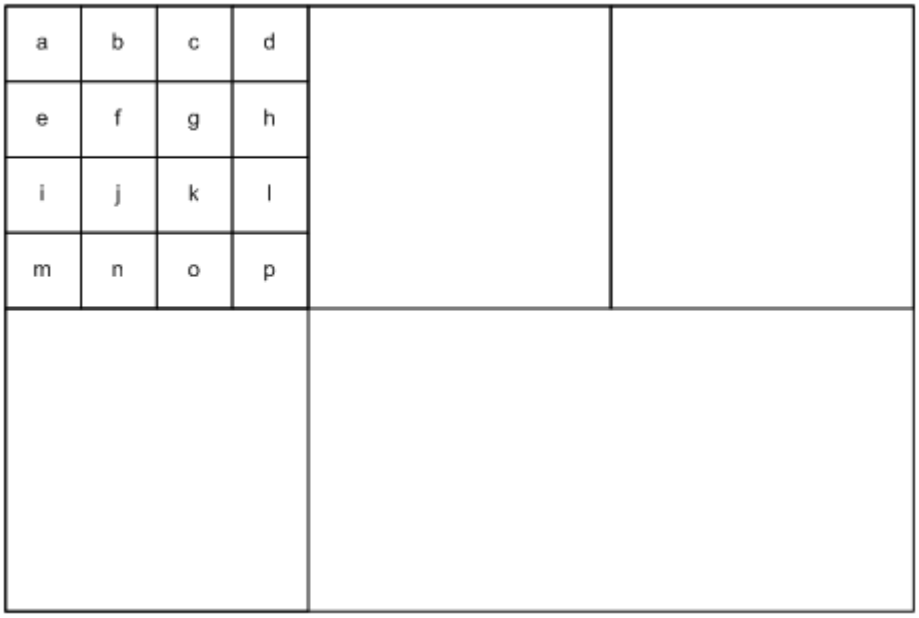


Рисунок 3.3 – Розбиття зображення на текселі [1]

Але, оскільки алгоритми працюють з блоками розміром чотири на чотири пікселі, очевидно це накладає певні обмеження на розмір вхідного зображення. Можна виділити дві основні стратегії поведінки у випадку коли розміри вхідного зображення не задовольняють вимоги алгоритмів. Перша, найбільш очевидна, це просто видавати користувачу відповідну помилку і зупиняти процес конвертації для конкретного зображення. З точки зору конвертації текстур цей підхід є найбільш коректним, проте насправді у багатьох випадках користувачу доведеться витрачати час на зміну розміру зображення буквально на кілька пікселів. Більш того часто край зображення може бути повністю прозорим, тобто значення альфа каналу пікселя в ньому дорівнює 0, і відповідно ця зміна не впливає на загальний вигляд зображення. Тому існує альтернативна стратегія, яка полягає у використанні «підкладки» (англ. Padding) (рис. 3.4).

Суть цієї «підкладки» полягає в тому, що замість того щоб змушувати користувача змінювати фізичний розмір зображення, ми змінимо його віртуальний розмір в процесі розбиття зображення на текселі. Тобто ми будемо додавати віртуальні пікселі до зображення доки його розміри не почнуть задовільняти вимоги алгоритмів блочного стиснення текстур. Всі додані таким чином пікселі будуть чорними повністю прозорими пікселями, тобто у форматі RGBA (red green blue alpha) матимуть значення (0, 0, 0, 0).

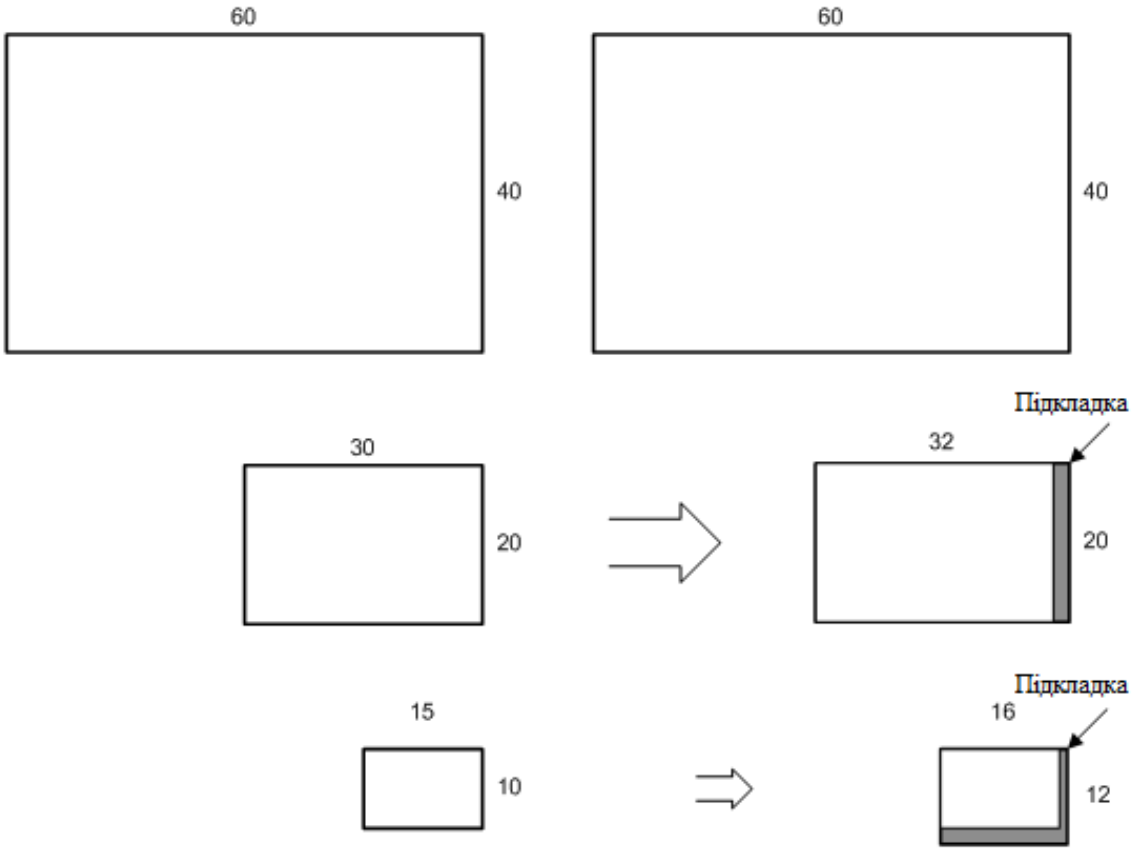


Рисунок 3.4 – Використання підкладки для зображення

Як було зазначено раніше (розділ 3.2), кожен етап конвертації зображення можливо представити у якості окремої задачі та описати за допомогою класу-нащадка абстрактного класу Job.

Відповідно для цього етапу буде створений клас SplitImageJob. Лістинг оголошення цього класу наведений нижче:

```
class SplitImageJob
    : public Job
{
public:
    SplitImageJob(std::string              sourceImage,
std::shared_ptr<ImageConversionJobData>& jobData);
    ~SplitImageJob();

private:
    virtual void Execute() override;

private:
    QImage* m_sourceImage { nullptr };
};
```

Завдяки розробленому методу автоматизації розробки багатопоточного програмного забезпечення, достатньо лише визначити конструктор цього класу та перевизначити функцію Execute, яка містить в собі усю описану раніше логіку виконання.

В результаті на початку виконання SplitImageJob завантажує зображення, за потреби змінює його віртуальний розмір та в результаті створює масив текселів на які було розбите вхідне зображення зображення.

1.4 Імплементація алгоритмів блочного стиснення

Наступним етапом, який необхідно імплементувати, є безпосередньо конвертація отриманих на попередньому етапі текселів у формат визначений конкретним алгоритмом блочного стиснення текстур.

Аналогічно попередньому етапу, був створений клас ConvertImageJob. Лістинг оголошення цього класу наведений нижче:

```
class ConvertImageJob
    : public Job
{
public:

ConvertImageJob(std::shared_ptr<ImageConversionJobData>&
jobData);

ConvertImageJob(std::shared_ptr<ImageConversionJobData>&
jobData, int texelBegin, int texelEnd);

private:
    virtual void Execute() override;

private:
    int m_texelBegin { 0 };
    int m_texelEnd { 0 };
};
```

Функція Execute під час виконання задачі конвертує отримані на попередньому етапі текселі у формат згідно заданого алгоритму конвертації. Також є можливість конвертувати не весь масив текселів, а тільки його частину. Ефективність такого рішення в залежності від вхідних даних буде оцінена у наступному розділі (розділ 4.4.1).

Можна виділити три основні, найбільш вживані алгоритми конвертації зображень у текстури: BC1, BC2 та BC3. Варто зазначити, що усі алгоритми BC

є алгоритмами з постійним коефіцієнтом стиснення, тобто розмір результуючої текстури залежить лише від розміру вхідного зображення та обраного алгоритму стиснення. Також варто зазначити, що при порівнянні з такими форматами як PNG та JPEG, DDS текстури займають більше місця на накопичувальному пристрої. Проте, цей формат є нативним для графічного процесору і при роботі з ним не виконується додаткове декодування даних, при цьому така текстура завантажена у пам’ять має набагато менший розмір ніж аналогічний їй масив пікселей у форматі RGB або RGBA.

3.4.1 BC1

Алгоритми BC1 можна назвати найпростішим з сімейства алгоритмів блочного стиснення текстур, проте водночас він забезпечує найбільший коефіцієнт стиснення 1:6. Основний недолік цього алгоритму це те, що він не вміє зберігати альфа-канал зображення, тобто значення прозорості вхідного зображення будуть ігноруватися.

Деяке програмне забезпечення для роботи з текстурами інколи виділяє окрему специфікація цього алгоритму, що має назву BC1a, проте часто він також використовується автоматично в залежності від вхідного зображення. Відмінність цього алгоритму полягає в тому, що він може зберігати так звану 1-бітну альфу, тобто в такому випадку піксель може бути або повністю прозорим, тобто мати значення альфа-каналу рівним 0, або повністю видимим, тобто мати значення альфа-каналу рівним 255.

Першим кроком алгоритм знаходить два основні кольори в заданому текселі. Основними називаються кольори різниця між якими є максимальною. Задамо тривимірний простір з Декартовою системою координат XYZ, де канали кольору виконують роль координатних осей (R = X, G=Y, B=Z). Якщо

представити кольори у вигляді точок у цьому тривимірному просторі, то основними будуть вважатися ті кольори відстань між якими у цьому просторі є найбільшою. Для зручності назвемо ці два кольори *color_0* та *color_1* відповідно.

Далі обчислюються два додаткові проміжні кольори *color_2* та *color_3* за наступними формулами:

$$color\_2 = 2/3 * color\_0 + 1/3 * color\_1 \quad (3.1)$$

$$color\_3 = 1/3 * color\_0 + 2/3 * color\_1 \quad (3.2)$$

Наступним кроком кожному пікселю у текселі ставиться у відповідність індекс, який залежить від того до якого з визначених чотирьох кольорів, колір поточного пікселя найближчий. Тобто аналогічному тому як визначалися основні кольори, колір пікселя порівнюється з чотирма визначеними кольорами. Індекс кольору відстань до якого була найменша ставиться у відповідність поточному пікселю. Варто зазначити, що для оптимізації математичних обчислень, немає сенсу шукати саме відстань між цими двома точками. Для порівняння достатньо обчислювати суми різниць квадратів по кожній з координат точки, без обчислення квадратного кореня з результату, оскільки це є досить затратною для процесору операцією.

В саму текстуру зберігається наступна інформація:

- два основні кольори у форматі RGB 565 (5-бітні канали R і B, та 6-бітний канал G);
- масив з 16-ти 2-бітних індексів які репрезентують колір кожного пікселя.

Якщо пронумерувати пікселі таким чином як було продемонстровано до цього (рис 3.3), то ця структура матиме в пам’яті наступний вигляд (рис 3.5).

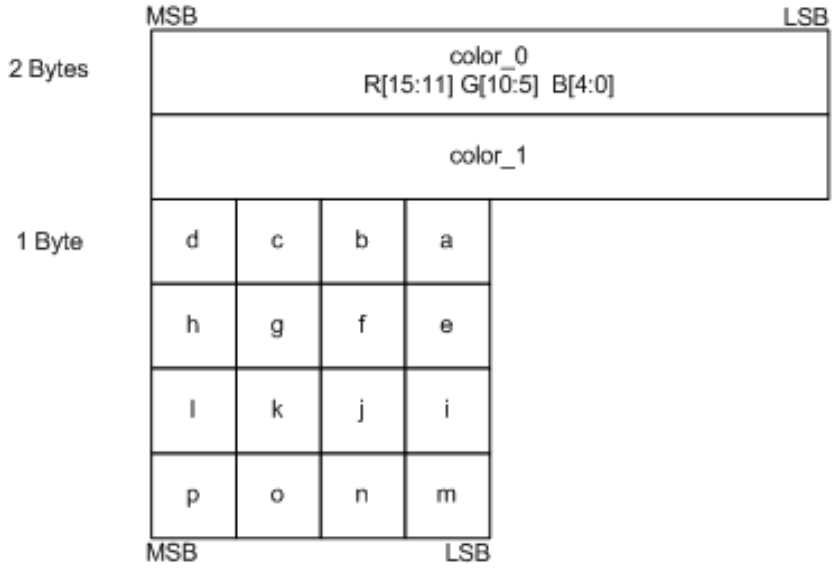


Рисунок 3.5 – Структура одного текселя у форматі алгоритму BC1 [1]

Тобто для збереження одного текселя у пам’яті необхідно всього лише 8 байт (два кольори по 2 байти, та 4 байти для індексів кольорів). Для порівняння якщо зберігати усі 16 пікселів у форматі RGB, то для цього необхідно витратити 48 байт пам’яті (16 кольорів по 3 байти). Звідси ми отримуємо коефіцієнт стиснення 1:6.

Проте, як було зазначено раніше, існує модифікація цього алгоритму яка дозволяє зберігати 1-бітний альфа канал. В такому випадку *color_3* завжди має значення RGBA (0, 0, 0, 0), тобто повністю прозорого чорного кольору. Відповідно *color_2* тепер обчислюється за наступною формулою:

$$color\_2 = 1/2 * color\_0 + 1/2 * color\_1 \quad (3.3)$$

Оскільки очевидно, що скоріш за все не в кожному текселі буде присутній прозорий піксель а відповідно необхідність зберігати 1-бітний альфа канал, як саме інтерпретувати даний тексель у збереженому форматі визначається за наступним принципом. Значення *color_0* та *color_1* відображаються у вигляді

16-бітних чисел та порівнюються між собою. Якщо *color_0* строго більший за *color_1*, тоді вважається, що в даному текселі відсутні прозорі пікселі та інтерполюються усі чотири кольори. В іншому випадку вважається, що прозорі пікселі присутні і четвертий колір резервується для відображення повністю прозорих пікселів.

Для демонстрації відмінності роботи алгоритмів, розглянемо який вони будуть генерувати результат для одного й того самого тестового зображення (рис. 3.6). Варто зазначити, що дана візуалізація має на меті виключно продемонструвати різницю в роботі алгоритмів, і подібне тестове зображення є тим прикладом коли не варто використовувати алгоритм BC1, а застосувати один з алгоритмів які підтримують кодування альфа-каналу, наприклад BC2 або BC3.

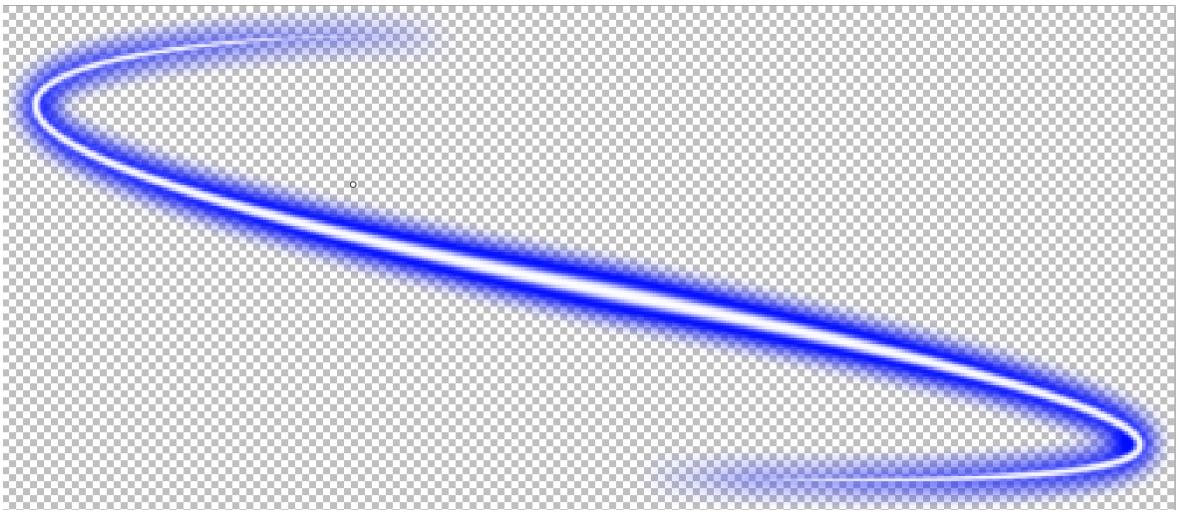


Рисунок 3.6 – Тестове зображення для демонстрації роботи алгоритмів

Результат роботи алгоритму BC1 матиме наступний вигляд (рис. 3.7). Тобто як було зазначено раніше у цьому розділі, даний алгоритм дозволяє зберегти лише 1-бітне значення альфа-каналу. Тобто повністю прозорі пікселі

такими і лишилися, але решта отримали значення альфа-каналу 255, тобто стали повністю видимими.

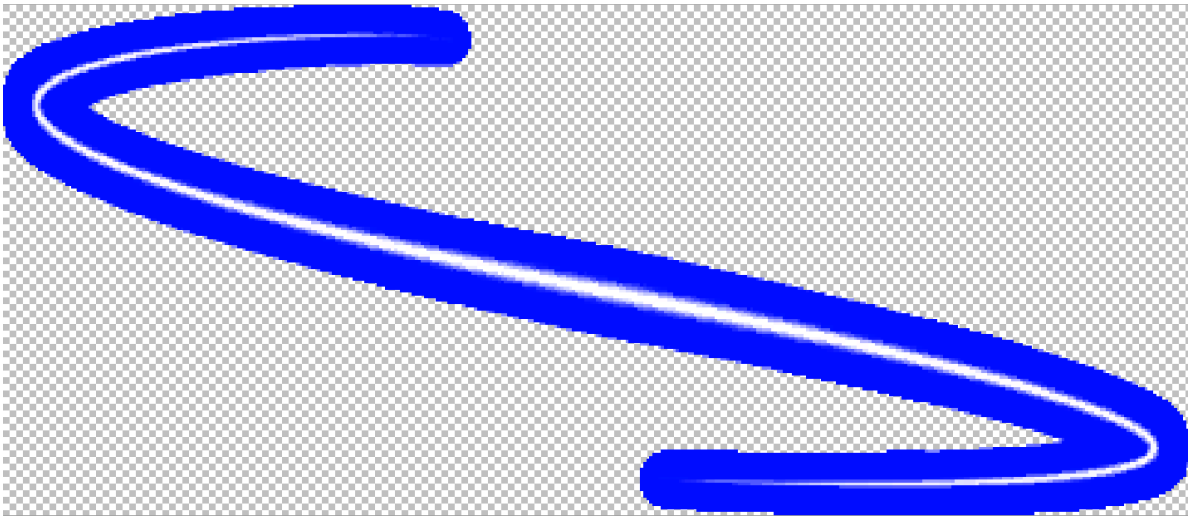


Рисунок 3.7 – Результат роботи алгоритму BC1

3.4.2 BC2

Алгоритм BC2 суттєво відрізняється від алгоритму BC1 тим, що окрім кольорів також кодує значення альфа-каналу для кожного пікселя у текселі. Це дозволяє суттєво покращити якість текстур отриманих з вхідного зображення де активно використовується зміна альфа-каналу. Проте очевидно, що через це розмір текстури також збільшується, а коефіцієнт стиснення зменшується по відношенню до алгоритму BC1. Для алгоритму BC2 коефіцієнт стиснення становить 1:4.

Для кодування альфа каналу алгоритм BC2 зберігає 4 старших біти значення альфа каналу кожного пікселя. Тобто, якщо значення альфа-каналу для конкретного пікселя становить для прикладу 127 (в двійковому записі 01111111), то даний алгоритм збереже значення 112 (в двійковому записі 01110000). Така зміна точності альфа-каналу може призводити до незначних

графічних артефактів, тобто певних спотворень зображення. Проте якщо у зображення зустрічаються різкі перепади значення альфа-каналу в межах одного текселя, вони будуть ледь помітні і загальний результат буде кращий ніж у решти алгоритмів.

Кольори кодуються повністю аналогічно алгоритму BC1 (розділ 3.4.1). Загальна структура текселя матиме наступний вигляд (рис. 3.8). Тобто для збереження одного текселя потрібно витратити 16 байтів (4 біти на кожен піксель для збереження альфа-каналу, та 8 байтів для збереження кольорів аналогічно BC1). Проте оскільки тепер альфа канал зберігається то при розрахунку коефіцієнту стиснення результуючу текстуру треба порівнювати з форматом RGBA. В такому випадку для збереження 16 пікселів було б потрібно 64 байти пам’яті (по 4 байти на кожен піксель). З цього отримується коефіцієнт стиснення 1:4.

Рисунок 3.8 – Структура одного текселя у форматі алгоритму BC2 [1]

Використовуючи тестове зображення (рис. 3.6), результат застосування алгоритму BC2 матиме наступний вигляд (рис. 3.9). Як бачимо якість отриманого результату набагато краща у порівнянні з алгоритмом BC1. Певні артефакти пов’язані з різкою зміною альфа-каналу є помітними, але у більшості випадків вони є припустимими.

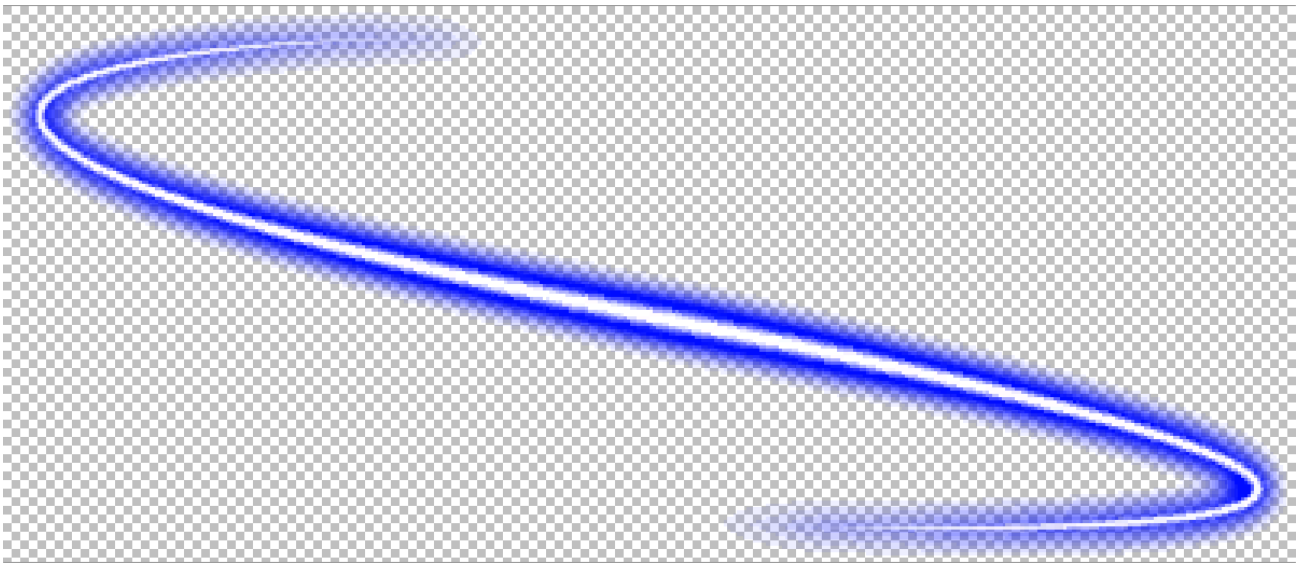


Рисунок 3.9 – Результат роботи алгоритму BC2

3.4.3 BC3

Алгоритм BC3 аналогічно алгоритму BC2 кодує альфа-канал вхідного зображення. Проте його механізм кодування дещо складніший і краще працює для випадків де в межах одного текселя відбувається плавна зміна значення альфа-каналу. При цьому коефіцієнт стиснення алгоритму BC3 такий само як і алгоритму BC2, тобто 1:4.

Принцип кодування альфа-каналу алгоритму BC3 дуже подібний до того яким чином алгоритм BC1 дозволяє кодувати 1-бітний альфа канал. Першим кроком у текселі знаходяться два основні значення альфа-каналу, тобто це два

значення різниця між якими є максимальною. Для зручності назвемо ці значення *alpha_0* та *alpha_1* відповідно. Наступним кроком інтерполюється ще шість значень альфа каналу, проте які саме це будуть значення залежить від того яке значення *alpha_0* чи *alpha_1* є більшим. Аналогічно алгоритму BC1 порівнюються їх числові значення. У випадку коли значення *alpha_0* є строго більшим за *alpha_1* будуть обчислені наступні значення:

$$alpha\_2 = 6/7 * alpha\_0 + 1/7 * alpha\_1 \quad (3.4)$$

$$alpha\_3 = 5/7 * alpha\_0 + 2/7 * alpha\_1 \quad (3.5)$$

$$alpha\_4 = 4/7 * alpha\_0 + 3/7 * alpha\_1 \quad (3.6)$$

$$alpha\_5 = 3/7 * alpha\_0 + 4/7 * alpha\_1 \quad (3.7)$$

$$alpha\_6 = 2/7 * alpha\_0 + 5/7 * alpha\_1 \quad (3.8)$$

$$alpha\_7 = 1/7 * alpha\_0 + 6/7 * alpha\_1 \quad (3.9)$$

У випадку коли значення *alpha_0* є меншим або рівним *alpha_1*:

$$alpha\_2 = 4/5 * alpha\_0 + 1/5 * alpha\_1 \quad (3.10)$$

$$alpha\_3 = 3/5 * alpha\_0 + 2/5 * alpha\_1 \quad (3.11)$$

$$alpha\_4 = 2/5 * alpha\_0 + 3/5 * alpha\_1 \quad (3.12)$$

$$alpha\_5 = 1/5 * alpha\_0 + 4/5 * alpha\_1 \quad (3.13)$$

$$alpha\_6 = 0 \quad (3.14)$$

$$alpha\_7 = 255 \quad (3.15)$$

Наступним кроком кожному пікселю у відповідність ставиться індекс відповідного значення альфа-каналу в залежності від того до якого з них воно є найближчим. Також варто зазначити, що кожен індекс буде займати 3 біти у порівнянні з кодуванням кольору де для індексу було потрібно лише 2 біти пам’яті.

Альфа-канал зберігається аналогічно тому як зберігаються кольори, тобто зберігається два основні значення альфа-каналу *alpha_0* та *alpha_1* та масив індексів визначених для кожного пікселя. Кодування кольорів при цьому не відрізняється від алгоритмів BC1 та BC2.

В результаті кодований алгоритмом BC3 тексель буде мати наступну структуру (рис. 3.10). Аналогічно алгоритму BC2, один кодований тексель займає 16 байтів пам’яті (два основних значення альфа-каналу по 1 байту, 16 індексів по 3 біти кожен, та 8 байтів для кодування кольору).

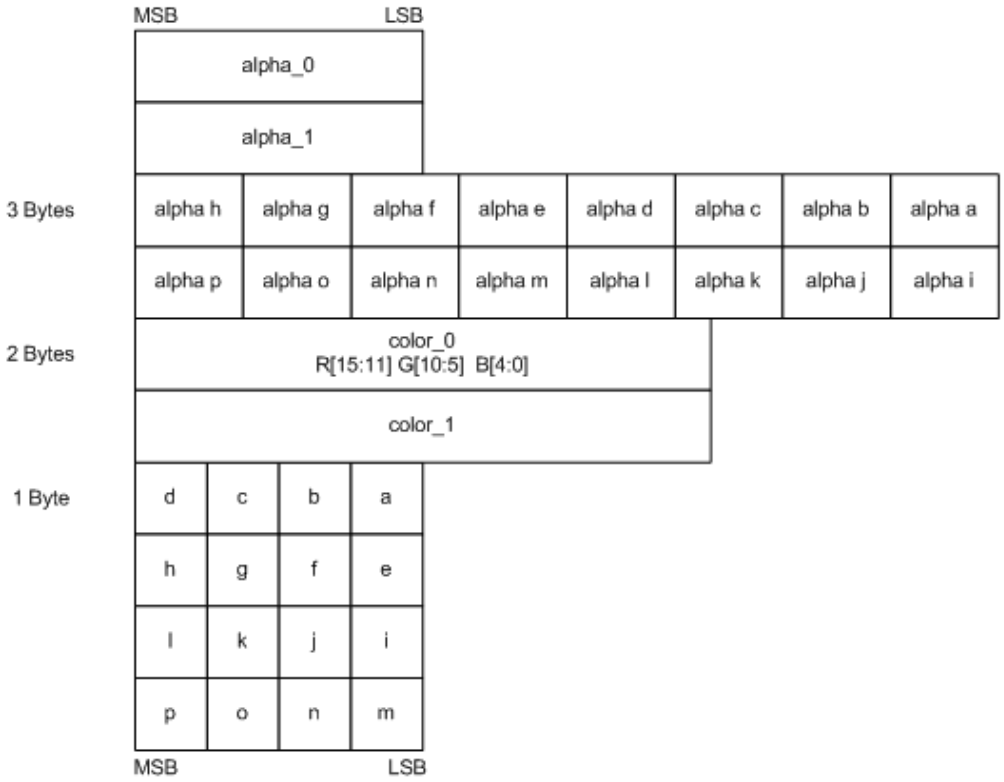


Рисунок 3.10 – Структура одного текселя у форматі алгоритму BC3 [1]

Використовуючи тестове зображення (рис. 3.6), результат застосування алгоритму BC3 матиме наступний вигляд (рис. 3.11). Як бачимо якість отриманого результату так само набагато краща у порівнянні з алгоритмом BC1, проте при такому рівні деталізації фактично не відрізняється від результату роботи алгоритму BC2.

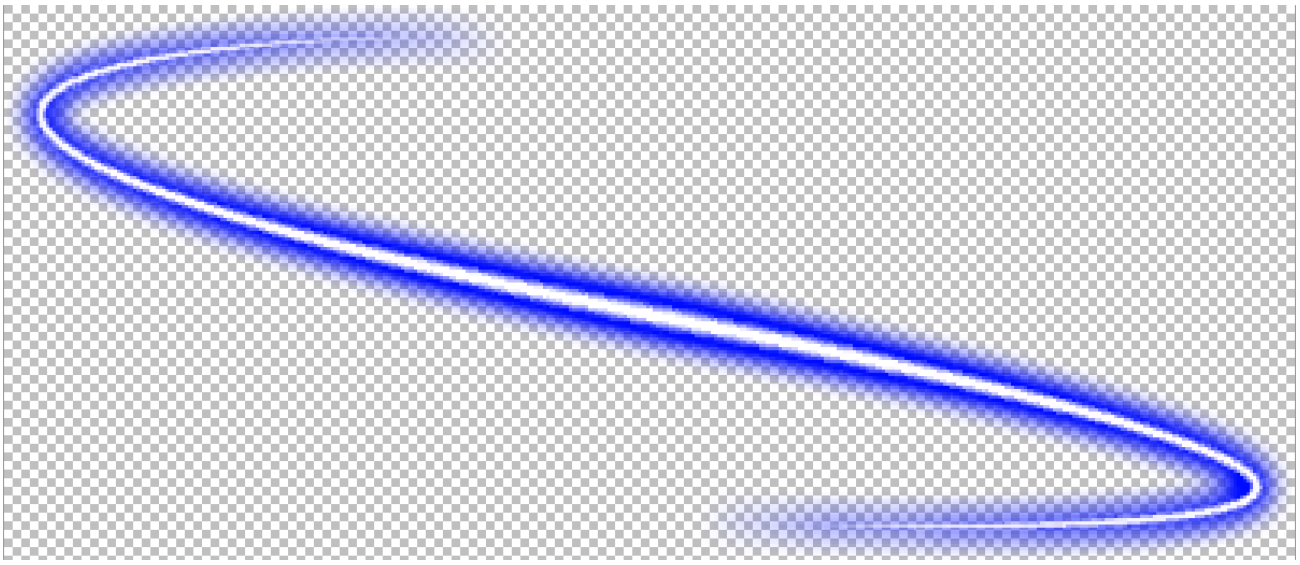


Рисунок 3.11 – Результат роботи алгоритму BC3

Проте при більшій деталізації частини зображення де є переходи значення альфа-каналу, різниця стає більш видимою (рис. 3.12). На практиці використовуються обидва алгоритми, оскільки якість отриманого результату оцінити заздалегідь неможливо. Здебільшого кінцевий користувач просто підбирає алгоритм в залежності в залежності від вхідного зображення, або частіше класу однотипних зображень, які конвертуються у текстури.

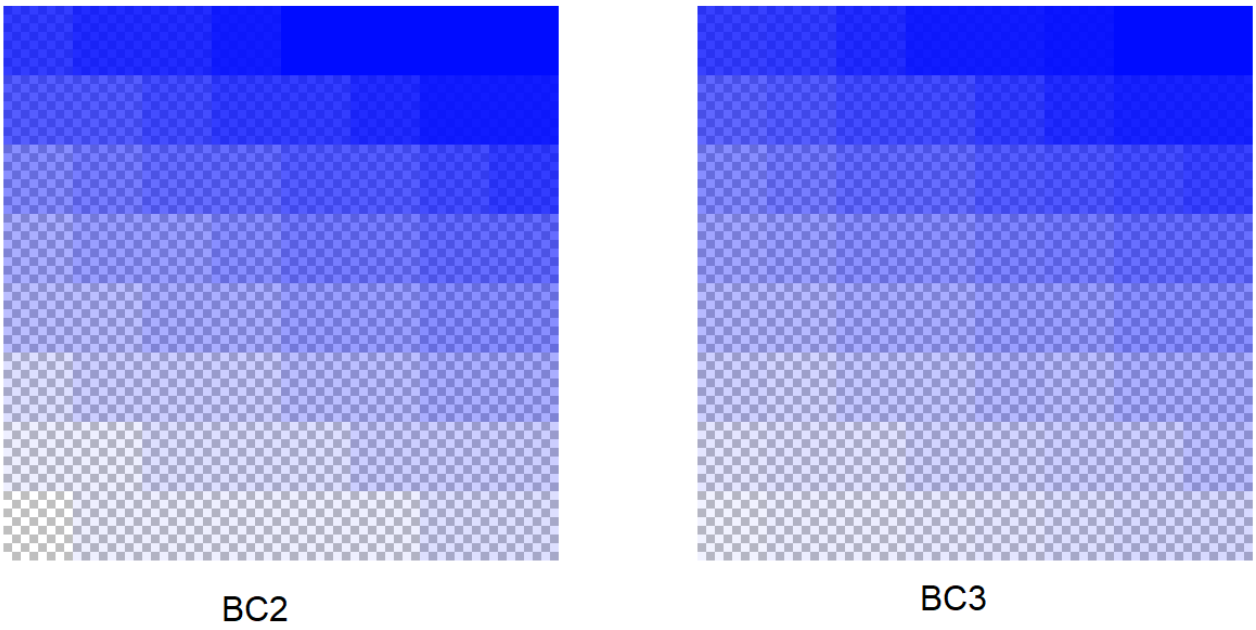


Рисунок 3.12 – Порівняння результатів роботи алгоритму BC2 та BC3

1.5 Збереження результату

Фінальним етапом конвертації зображення у текстуру є збереження отриманих у процесі конвертації даних у файл з відповідним розширенням. Важливо зазначити, що файли текстур обов’язково мають бути бінарними файлами.

Першими у файл записуються так зване «магічне число» (англ. Magic number) [18]. Магічне число є константою і відповідає бітовому запису строки «DDS » (обов’язково з пробілом у кінці). Значення цього магічного числа 0x20534444. Його призначення полягає в тому, щоб при читанні файлу можна було б одразу зробити висновок чи дійсно у ньому записана DDS текстура.

Якщо магічне число відсутнє, або не співпадає з потрібним, текстура далі не вичитується.

Наступним у файл записується заголовок текстури [19]. Це структура яка має фіксований розмір, та містить у собі інформацію, що дозволяє отримати основні дані про текстуру, такі як її розміри, яким алгоритмом вона була конвертована і тд. Лістинг цієї структури має наступний вигляд:

```
struct DDS_PIXELFORMAT
{
    DWORD dwSize{ 0 };
    DWORD dwFlags{ 0 };
    DWORD dwFourCC{ 0 };
    DWORD dwRGBBitCount{ 0 };
    DWORD dwRBitMask{ 0 };
    DWORD dwGBitMask{ 0 };
    DWORD dwBBitMask{ 0 };
    DWORD dwABitMask{ 0 };
};

struct DDS_HEADER
{
    DWORD           dwMagic{ 0 };
    DWORD           dwSize{ 0 };
    DWORD           dwFlags{ 0 };
    DWORD           dwHeight{ 0 };
    DWORD           dwWidth{ 0 };
    DWORD           dwPitchOrLinearSize{ 0 };
    DWORD           dwDepth{ 0 };
    DWORD           dwMipMapCount{ 0 };
    DWORD           dwReserved1[11];
    DDS_PIXELFORMAT ddspf;
    DWORD           dwCaps{ 0 };
    DWORD           dwCaps2{ 0 };
    DWORD           dwCaps3{ 0 };
    DWORD           dwCaps4{ 0 };
    DWORD           dwReserved2{ 0 };
};
```

Після заголовку записуються текселі в кодованому вигляді в залежності від обраного алгоритму (рис. 3.5, рис. 3.8, рис. 3.10).

Аналогічно попереднім етапам був розроблений клас-нащадок абстрактного класу Job, який під час свого виконання повинен зберегти отримані на попередньому етапі дані у відповідний файл згенерувавши відповідний заголовок з урахуванням параметрів вхідного зображення та обраного алгоритму конвертації. Лістинг оголошення класу SaveTextureJob приведений нижче:

```
class SaveTextureJob
    : public Job
{
public:

SaveTextureJob(std::shared_ptr<ImageConversionJobData>&
jobData);

    virtual ThreadAffinity GetAffinity() override;

private:
    virtual void Execute() override;
};
```

Також варто зазначити, що на відміну від попередніх етапів, у класі SaveTextureJob перевизначена функція GetAffinity(). Оскільки даний клас виконує блокуючу операцію запису у файл, використовуючи механізм схильності (розділ 2.4), є можливість виконувати цю задачу у спеціально відведеному потоці.

1.6 Організація виконання задач у потоках

Для організації виконання задач у багатопоточному середовищі буде використаний клас JobManager, який є частиною розробленого методу

автоматизації написання багатопоточного програмного забезпечення (розділ 2.4). На відміну від окремих задач, класи яких необхідно було реалізувати для кожного етапу процесу конвертації зображення у текстуру, JobManager не потребує жодних змін оскільки він не має залежностей від конкретних типів задач і працює з ними як з абстракцією.

Для класу JobManager буде використана наступна конфігурація. Кількість потоків буде дорівнювати максимальній кількості доступних потоків у системі. При цьому один з потоків матиме у своїй конфігурації схильність (розділ 2.4) для операцій запису та зчитування даних у системі. Як було зазначено раніше, SaveTextureJob є типом задач у яких буде виставлене таке саме значення схильності (розділ 3.5).

Наступним кроком є імплементація програмного коду, який отримавши вхідні дані має створити потрібну кількість задач для конвертації зображень у текстури. Вхідні дані включають в себе список файлів зображень, список файлів у які треба зберегти результуючі текстури, алгоритм конвертації який має бути застосований та під який графічний API текстури повинні мати орієнтацію (розділ 3.7.2).

Лістинг відповідного програмного коду приведений нижче:

```
    void CreateJobForImageConversion(JobManager &jobManager, std::string &inputFile, std::string &outputFile, CompressionType type, TextureOrientation textureOrientation)
    {
        std::shared_ptr<ImageConversionJobData> jobData = std::make_shared<ImageConversionJobData>(inputFile, outputFile, type, textureOrientation);

        std::shared_ptr<Job> splitJob = std::make_shared<SplitImageJob>(inputFile, jobData);
        std::shared_ptr<Job> convertJob = std::make_shared<ConvertImageJob>(jobData);
```

```
        convertJob->AddDependency(splitJob.get());
        std::shared_ptr<Job>       saveTextureJob       =
std::make_shared<SaveTextureJob>(jobData);
        saveTextureJob->AddDependency(convertJob.get());

        jobManager.QueueJob(splitJob);
        jobManager.QueueJob(convertJob);
        jobManager.QueueJob(saveTextureJob);
    }
```

За допомогою механізму залежностей (розділ 2.4) через виклик функції AddDependency() задається строгий порядок виконання задач. Тобто цей механізм гарантує правильний порядок виконання етапів конвертації зображення у текстури (розділ 3.2).

Клас ImageConversionJobData є нащадком класу JobData те реалізує механізм спільного стану (розділ 2.4) для усіх задач створених для конвертації зображення у текстури. Це дозволяє не витрачати додаткові ресурси і час виконання на копіювання даних між етапами виконання. Наприклад SplitImageJob по завершенню не повинен буде робити копію масиву текселів на які було розбите зображення та передавати їх на обробку ConvertImageJob, оскільки вони є частиною спільного стану цих двох задач. За допомогою std::shared_ptr, який є частиною стандартної бібліотеки C++, ресурси виділені для цього спільного стану будуть автоматично звільнені, після того як буде видалене останнє посилання на них. Тобто в даному випадку після того як виконається остання задача, а саме SaveTextureJob.

Дану функцію потрібно викликати для кожного запису у вхідних даних. Після того як JobManager отримує повний список задач для виконання, достатньо всього лише викликати відповідну функцію Start() для того щоб запустити процес їх виконання. Якщо програмний код повинен дочекатися виконання усіх задач перш ніж продовжувати своє виконання, можна

скористатися функцією Stop(). Вона заблокує потік з якого була викликана до моменту поки JobManager повністю не закінчить процес виконання своїх задач.

Варто звернути увагу, що у наведеному коді повністю відсутнє використання низькорівневих примітивів синхронізації, або використання багатопоточної бібліотеки як такої. Усі ці механізми інкапсульовані в середині класу JobManager, що суттєво спрощує і пришвидшує написання багатопоточного програмного коду. Також це є підтвердженням виконання нефункціональної вимоги до розробленого методу (розділ 2.3) про надання розробнику високорівневої абстракції для роботи з багатопоточним кодом, замість написання низькорівневого коду.

3.7 Розробка Qt застосунку для конвертації зображень

Для перевірки коректності роботи алгоритмів конвертації, а також для проведення замірів швидкодії даної реалізації програмного забезпечення для конвертації зображень у текстури на основі розробленого методу автоматизації написання багатопоточного програмного забезпечення (розділ 2.4), був розроблений застосунок з графічним інтерфейсом користувача з використанням фреймворку Qt (розділ 3.1).

Застосунок складається з двох віджетів [20], структурних одиниць побудови графічного інтерфейсу за допомогою фреймворку Qt. Віджет головного вікна надає доступ увесь доступний функціонал для конвертації зображень у текстури. Також у віджеті головного вікна інкапсульований програмний код (розділ 3.6) для конвертації зображень у текстури на основі розробленого раніше методу (розділ 2.4). Віджет для перегляду текстур дозволяє переглянути результат конвертації зображення у текстуру засобами графічного API OpenGL інтегрованого у фреймворк Qt.

3.7.1 Віджет головного вікна

Віджет головного вікна містить в собі усі основні налаштування та функції пов’язані з конвертацією зображення у текстури (рис. 3.13).

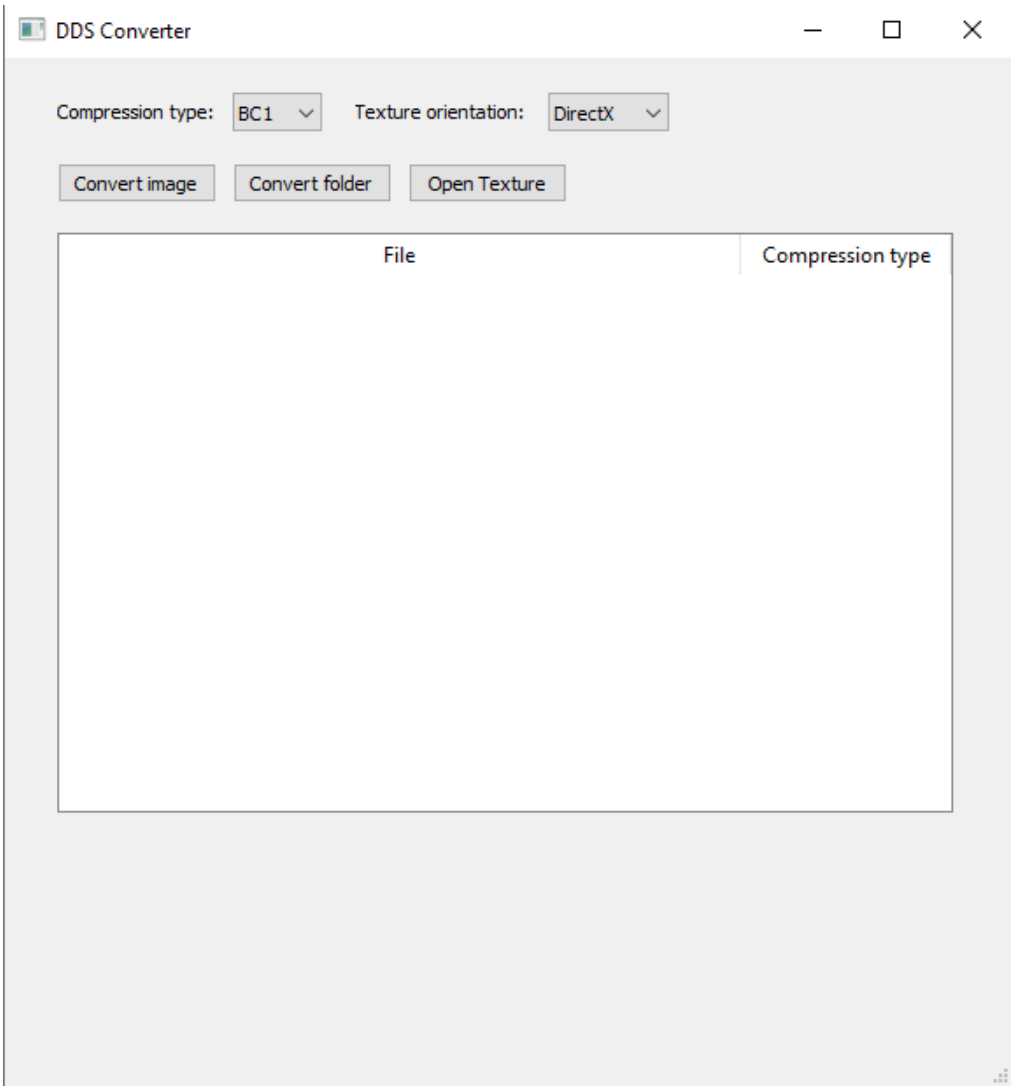


Рисунок 3.13 – Віджет головного вікна

Список доступних налаштувань:

- вибір алгоритму яким зображення конвертується у текстуру (BC1, BC2, BC3);
- орієнтація текстури в залежності від графічного API (DirectX, OpenGL) (розділ 3.7.2).

Список доступних функцій:

- конвертація обраного зображення у текстуру з поточними налаштуваннями;
- конвертація усіх зображень у папці в текстури з поточними налаштуваннями;
- перегляд обраної текстури за допомогою спеціального віджету (розділ 3.7.2);
- перегляд списку конвертованих текстур з моменту поточного запуску застосунку, з можливістю переглянути текстуру за допомогою подвійного кліку по її назві.

3.7.2 Віджет перегляду текстур

Оскільки фреймворк Qt має вбудовану підтримку графічного API OpenGL (розділ 3.1), це дозволяє з його використанням розробити віджет, який буде відмальовувати текстури саме засобами графічного процесору. Головна перевага такого підходу, що це гарантує, що результуюча текстура матиме точно такий самий вигляд, як при використанні в цільовому програмному забезпеченні. Тобто це дозволяє як оцінити якість отриманого результату, так і перевірити текстуру на наявність графічних артефактів та за потреби змінити алгоритм конвертації зображення у текстури.

Віджет (рис. 3.14) може автоматично масштабувати розмір текстури, проте гарантує збереження співвідношення її розмірів.

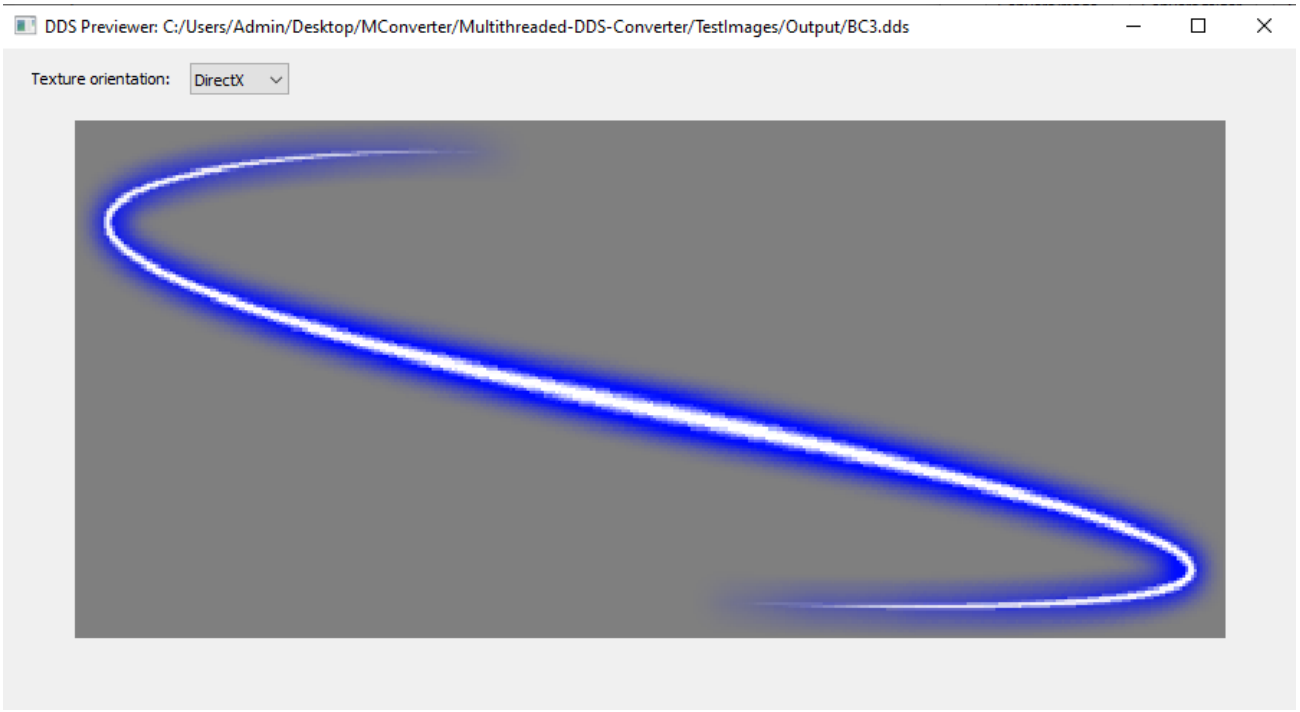


Рисунок 3.14 – Віджет перегляду текстури

Єдиним доступним налаштуванням є вибір орієнтації текстури. Цей параметр залежить від того яким графічним API користується кінцевий користувач. На вибір предсталена орієнтація текстур для графічних API DirectX та OpenGL.

DirectX та OpenGL є найбільш розповсюдженими станом на зараз графічними API. Основною перевагою OpenGL можна виділити те, що він працює на усіх сучасних операційних системах в той час як DirectX працює лише на операційних системах розроблених корпорацією Microsoft. В сучасному графічному забезпеченні часто можна зустріти можливість обирати яким саме графчіним API користуватися під час роботи, з урахуванням обмеження операційної системи.

Проте для текстур між цими графічними API є одна ключова відмінність. Вони використовують різні системи координат для роботи з текстурами (рис. 3.15). Більшість програмних застосунків які реалізують алгоритми блочного стиснення текстур ігнорують дану відмінність, що призводить до значного зменшення потенційних користувачів програмного продукту.

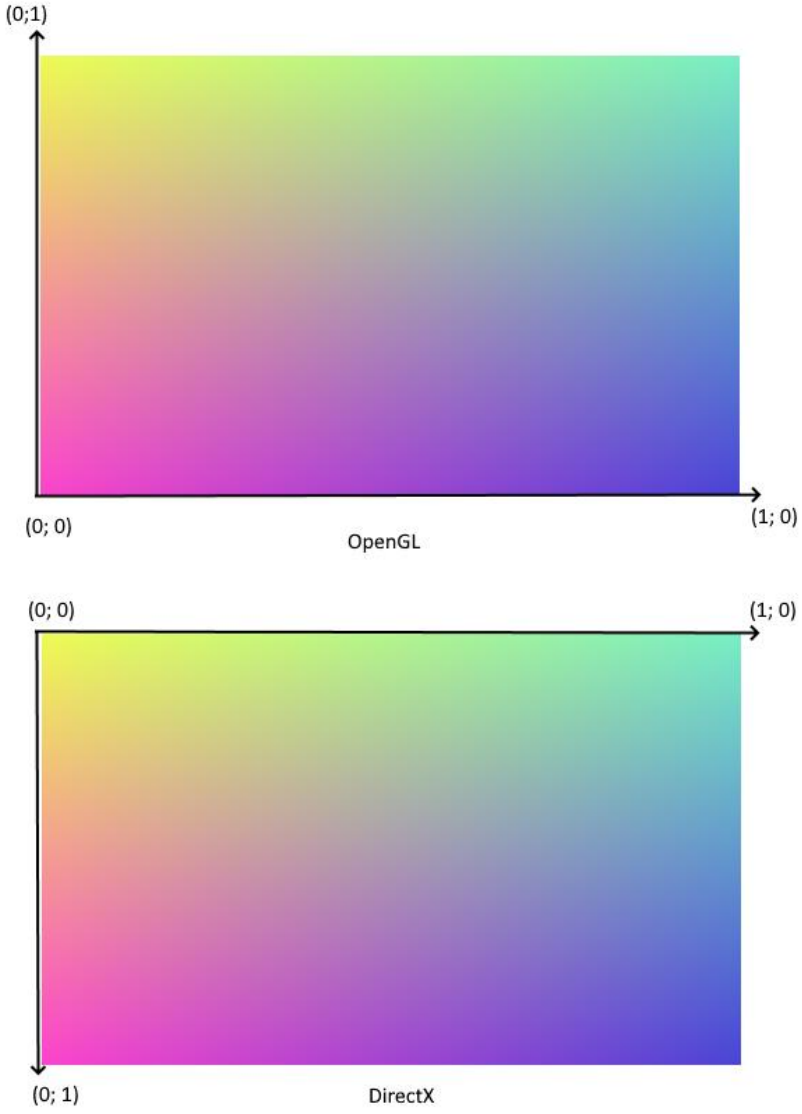


Рисунок 3.15 – Порівняння систем координат DirectX та OpenGL

Налаштування орієнтації текстур, яке присутнє як під час конвертації зображення у текстуру (розділ 3.7.1), так і під час перегляду, дозволяє вирішити цю проблему.

Це є необхідним рішенням оскільки заголовок текстури не містить інформації про те під який саме графічне API вона була створена. Фактично це означає, що в залежності від налаштування орієнтації текстури змінюється порядок в якому ітерується зображення під час розділення на текселі (розділ 3.3).

Оскільки віджет для перегляду текстур створений за допомогою графічного API OpenGL, у випадку якщо текстура була створена під графічне API DirectX, змінюється механізм ітерації текстури для того щоб компенсувати цю різницю.

Для відображення текстури були імплементовані вертексний та фрагментний шейдери. Шейдером [21] називається визначений користувачем програмний код, який виконується графічним процесором на одному з етапів відмалювання зображення засобами графічного API. Вертексний шейдер [22] відповідає за трансформацію текстури, тобто зміну її розташування на екрані з урахуванням можливості її повороту, нахилу зміни масштабу тощо. Фрагментний шейдер [23] відповідає за те, за яким механізмом колір текстури з певною текстурною координатою переноситься на екран.

Висновки до розділу

В даному розділі було детально описано етапи розробки програмного засобу для конвертації зображень у текстури за допомогою розробленого раніше методу автоматизації розробки багатопоточного програмного забезпечення.

Було проаналізовано з яких етапів складається процес конвертації зображення у текстури. Кожен виділений етап був описаний у вигляді задачі, яка може бути виконана у багатопоточному середовищі засобами раніше розробленого методу.

Були детально описані найбільш розповсюджені алгоритми блочного стиснення текстур, а саме BC1, BC2 та BC3. Були наведені приклади застосування цих алгоритмів та продемонстровано різницю в отриманому результаті.

Було розроблено застосунок з використанням фреймворку Qt згідно зі сформованими раніше вимогами (розділ 1.4).

4 ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ

4.1 Оцінка ефективності відносно існуючих методів

Під час аналізу існуючих рішень (розділ 1), програмні засоби перевірялися на невеликому тестовому наборі зображень, що складався з десяти зображень збережених у форматі PNG розміром 4096 на 4096 пікселі. Для конвертації був використаний алгоритм BC1. Були отримані наступні результати:

- Open 3D Engine Asset Processor: 12.723 секунди;
- NVIDIA Texture Tools Exporter: 11.400 секунди.

Результат отриманий для запропонованого рішення становить 7.488 секунди, що підтверджує ефективність запропонованого методу у порівнянні з існуючими засобами.

Таку різницю у швидкодії можна пояснити наступним чином.

Asset Processor, який є складовою ігрового рушія Open 3D Engine, це в першу чергу універсальний програмний засіб. Тобто його задача займатися конвертацією не лише зображень у текстури, але взагалі усіх наявних ресурсів у проекті, таких як файли тривимірної геометрії, анімації, матеріали, скелети анімованих об’єктів тощо. Тобто цей інструмент не має змоги бути оптимізованим під конкретну задачу, оскільки в першу чергу він повинен з адекватною швидкодією виконувати абсолютно усі типи задач, замість того щоб досягти найкращого результату для однієї з них. Також варто зазначити, що Asset Processor також містить певні модифікації в етапах конвертації зображення у текстури, призначення яких є оптимізацією текстур конкретно для рушія Open 3D Engine.

Порівнюючи розроблений програмний засіб з NVIDIA Texture Tools Exporter, варто зазначити наступне. Головна перевага програмного засобу від NVIDIA це можливість використання бібліотеки CUDA [5] для виконання операцій конвертації зображення у текстури за допомогою графічного процесору. Оскільки алгоритми блочного стиснення текстур використовують велику кількість математичних операцій, які виконуються над великою кількістю даних, це створює умови коли використання ресурсів графічного процесору є виправданим, та дозволяє отримувати результат за дуже невелику кількість часу (розділ 1.3.3). І даний програмний засіб був розроблений саме з акцентом на такому рішенні. Проте як вже було зазначено (розділ 1.3.2), такий підхід тягне за собою дуже жорсткі вимоги стосовно апаратного забезпечення користувача. Конвертація без використання графічного процесору для даного програмного застосунку існує скоріш у вигляді рішення для сумісності з більш простими системами. Проте навіть враховуючи все вище зазначене, дане програмне рішення є одним з найкращих серед тих, що знаходяться у вільному доступі.

4.2 Розробка тестового набору даних

Оскільки використаний раніше тестовий набір даних (розділ 4.1) мав на меті дати швидку оцінку швидкодії існуючих програмних засобів та дати можливість об’єктивно порівняти з ними розроблений метод, він був навмисно спрощений, для найшвидшого отримання результату.

Даний набір тестових даних мав декілька суттєвих недоліків:

- розмір тестового набору даних був замалим у порівнянні з реальною кількістю зображень з яким за раз повинно працювати подібне програмне забезпечення;

– набір тестових даних був занадто однорідний оскільки на практиці вхідні зображення можуть бути абсолютно різного розміру.

Для більш комплексної та обґрунтованої оцінки швидкодії був розроблений наступний набор тестових даних. Оскільки час конвертації зображення алгоритмами блочного стиснення текстур залежить лише від розміру зображення, а не від його вмісту, буде використане те саме тестове зображення, що і в попередньому випадку, проте воно буде також представлене з іншими розміром. Для зручності введемо наступні означення стосовно розмірності зображень. Маленьке має розмір 256 на 256 пікселів, середнє 1024 на 1024 пікселі, велике 4096 на 4096 пікселів.

Для перевірки ефективності запропонованого методу були виділені наступні набори тестових даних:

– 60 великих зображень;
– 60 середніх зображень;
– 60 маленьких зображень;
– 20 великих зображень, 20 середніх та 20 маленьких.

Оскільки під час попереднього порівняння NVIDIA Texture Tools Exporter показав результат кращий за Asset Processor, запропонований метод буде порівнюватися саме з ним на визначених наборах даних.

4.3 Отримання результатів для тестового набору даних

Застосувавши програмний засіб NVIDIA Texture Tools Exporter для розробленого тестового набору даних були отримані наступні результати:

– 60 великих зображень – 63.273 секунди;
– 60 середніх зображень – 5.184 секунди;
– 60 маленьких зображень – 0.541 секунди;

– 20 великих зображень, 20 середніх та 20 маленьких – 46.687 секунди.

Тепер отримаємо результати застосувавши розроблений програмний засіб:

– 60 великих зображень – 33.137 секунди;
– 60 середніх зображень – 2.296 секунди;
– 60 маленьких зображень – 0.229 секунди;
– 20 великих зображень, 20 середніх та 20 маленьких – 13.328 секунди.

Отриманий результат демонструє, що швидкодія запропонованого методу є вищою. Особливо велика різниця помітна в наборах тестових даних де присутня значна частина великих зображень. Це можна пояснити з наступних причин. Під час тестування була виявлена певна особливість роботи програмного засобу NVIDIA Texture Tools Exporter. Під час конвертації зображень тестових наборів 1 та 4 була задіяна майже уся оперативна пам’ять на ПК де проводилося тестування. Оскільки програмний код даного застосунку є закритим, то можна лише висунути припущення, що програмний засіб заздалегідь алокує усі потрібні йому ресурси для конвертації усіх зображень. Такий підхід є ефективним, але є дуже вибагливим до апаратних ресурсів системи. Також можна побачити, що це дуже негативно впливає на швидкодію.

З отриманих результатів можна зробити висновок, що розроблений метод є ефективним для наборів вхідних даних і не залежить від їх розміру та однорідності.

4.4 Дослідження потенційних оптимізацій розробленого методу

Не зважаючи на підтверджену ефективність запропонованого методу, існує ряд модифікацій, які спрямовані на покращення швидкодії методу для певних вхідних даних.

4.4.1 Розбиття конвертації великого зображення на частини

Очевидно, що під час конвертації великого зображення, основну частину часу займає безпосередньо виконання алгоритму блочного стиснення текстур. Що і не дивно, оскільки для зображення розміром 4096 на 4096 пікселі потрібно конвертувати 1048576 текселів. В свою чергу це призводить до явища, яке має назву перемикання контексту (context switch) [24]. Воно виникає в той момент коли системі терміново потрібно звільнити ядро процесору для виконання більш пріоритетної задачі. Для цього система зберігає контекст потоку, який виконується, та додає його в спеціальну чергу з відповідним пріоритетом. Коли з’явиться вільне ядро, цей контекст буде знову туди скопійований та продовжить виконання.

Проте подібна операція займає додатковий час під час виконання програмного забезпечення. Більш того операційна система Windows надає власним задач набагато вищий пріоритет, ніж мають потоки створені рештою програм.

Для того щоб зменшити втрати часу під час перемикання контекстів, є сенс дуже великі задачі, що виконуються в потоках розділяти на окремі підзадачі.

Для застосування алгоритмів конвертації був заздалегідь впроваджений механізм, який дозволяє конвертувати лише частину текселів (розділ 3.4).

Для тестового набору даних з 60 великих зображень було перевірено розбиття на дві та на чотири частини. Були отримані наступні результати:

- без розбиття – 33.137 секунди;
- розбиття на дві частини – 30.280 секунди;
- розбиття на чотири частини – 30.267 секунди.

Як бачимо з отриманих результатів, підхід з розбиттям задачі з застосування алгоритмів конвертації є ефективним і дає виграш у майже 3 секунди (~9%) для тестового набору даних. При цьому варто зазначити, що для даного розміру зображення особливої різниці між розбиттям на дві та чотири частини не має, тому варіант з розбиттям на дві частини є оптимальним.

4.4.2 Виконання кількох задач одним потоком

Подібно до того як занадто тривалі за часом виконання задачі можуть накладати додаткові витрати по часу виконання, те саме стосується і занадто маленьких задач. В цьому випадку час виконання коду пов’язаного з ініціалізацією задача та перемикання між різними задачами може складати помітну частину від загального часу роботи програмного забезпечення.

Для того щоб покращити швидкодію у подібному випадку, був розроблений новий тип задачі BatchJob. Лістинг оголошення цього класу наведений нижче:

```
class BatchJob
    : public Job
{
public:
    void AddJobToBatch(std::shared_ptr<Job>& job);

private:
    virtual void Execute() override;

private:
    std::vector<std::shared_ptr<Job>> m_jobs;
};
```

Функція AddJobToBatch() дозволяє додавати інші задачі у масив задач, які будуть викликані під час виконання BatchJob. Тобто фактично за один виклик у потоці будуть виконані усі задачі додані до цього списку.

Результати перевірки цієї оптимізації на тестових наборах даних з 60 середніх та 60 маленьких зображень відповідно:

- 60 середніх зображень – 2.164 секунди;
- 60 маленьких зображень – 0.189 секунди.

Можемо зробити висновок, що для середніх зображень дана оптимізація не потрібна. Проте для маленьких зображень вона дає помітний результат у 0.04 секунди (~17.5%), що є суттєвим приростом швидкодії, якщо конвертувати буде необхідно на порядок більшу кількість зображень.

Висновки до розділу

В даному розділі було виконано оцінку ефективності розробленого програмного засобу.

В результаті експериментальних досліджень для різних типів вхідних даних було доведено ефективність методу у порівнянні з існуючими рішеннями. В середньому було продемонстровано зменшення часу потрібного на конвертацію тестового набору зображень у 2.5 рази.

Також були запропоновані оптимізації, які дозволили покращити швидкодію методу при роботі з великими та маленькими зображенням. Дані оптимізації дозволили зменшити час конвертації відповідних наборів зображень на 9% для великих зображень, та на 17.5% для маленьких зображень.

ВИСНОВКИ

У ході виконання магістерської дисертації було розглянуто питання пов’язані з методами та програмними засобами конвертації зображень у DDS текстури.

На основі даних, отриманих в процесі аналізу, сформульовано задачу мета якої покращити швидкодію відносно існуючих програмних засобів.

Для розробки методу автоматизації розробки багатопоточного програмного забезпечення та розробленого на його основі програмного засобу для конвертації зображень у DDS текстури було використано мову програмування C++ та фреймворк Qt. Все перераховане є широко вживаними та безкоштовними інструментам.

Розроблено програмне забезпечення для конвертації зображень у DDS текстури. Програмний засіб містить в собі найбільш вживані алгоритми конвертації зображень у текстури, а також усі відповідні налаштування. Ефективність даного програмного рішення було доведено експериментально у порівнянні з існуючими рішенням, що знаходяться у вільному доступі.

Результати роботи над магістерською дисертацією опубліковані у науковій статті [7] та тезах конференції [25].

Наукова новизна одержаних результатів магістерської дисертації:

вперше:

– вперше розроблено метод автоматизації розробки багатопоточного програмного коду мовою C++, який за допомогою високорівневих абстракцій дає змогу розбити процес на окремі задачі та виконати їх асинхронно з урахуванням встановлених залежностей, ефективно використовуючи при цьому наявні ресурси програми.

удосконалено:

– програмний засіб для конвертації зображень у текстури за рахунок використання запропонованого методу автоматизації розробки багатопоточного програмного забзпечення, що забезпечує кращі показники швидкодії відносно існуючих рішень.

Практична значимість одержаних результатів полягає у розробці швидкого та ефективного програмного засобу для конвертації зображень у текстури, що дозволяє суттєво скорити витрати часу кінцевого користувача при використанні аналогічних програмних засобів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Learn. Block Compression [Електронний ресурс] // Microsoft. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-block-compression#bc1
2. Paint.NET [Електронний ресурс] // Paint.NET. – Режим доступу: https://www.getpaint.net/
3. Open 3D Engine [Електронний ресурс] // Open 3D Engine. – Режим доступу: https://www.o3de.org/
4. Nvidia Texture Tools [Електронний ресурс] // NVIDIA. – Режим доступу: https://developer.nvidia.com/nvidia-texture-tools-exporter
5. NVIDIA CUDA. NVIDIA Documentation Center | NVIDIA Developer [Електронний ресурс] // NVIDIA. – Режим доступу: https://docs.nvidia.com/cuda/
6. NVIDIA Developer Program [Електронний ресурс] // NVIDIA. – Режим доступу: https://developer.nvidia.com/developer-program
7. Нестеренко К.П., Стеценко І.В. Метод автоматизації розробки багатопоточної програми мовою С++ на прикладі конвертації зображень у DDS текстури. // Адаптивні систем автоматичного управління. – Київ, 2023. - №1(42) – С. 160 –170. (категорія «Б»)
8. C++ Concurrency support library [Електронний ресурс] // cppreference.com – Режим доступу: https://en.cppreference.com/w/cpp/thread
9. C++ Parallel execution [Електронний ресурс] // cppreference.com – Режим доступу: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

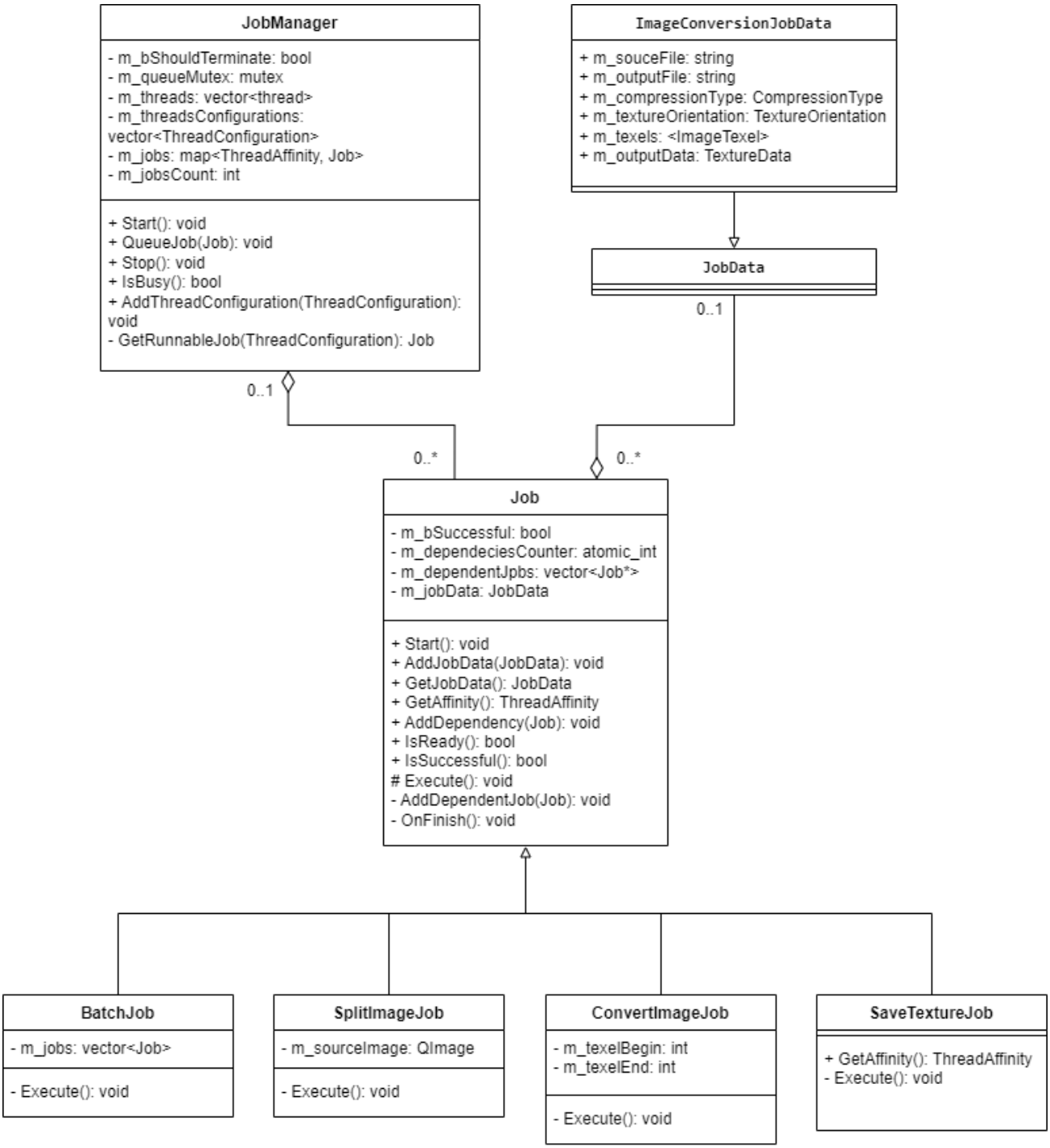
10. Modern multithreading and concurrency in C++ [Електронний ресурс] // Educative. – Режим доступу: https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp
11. Boost.Threads [Електронний ресурс] // Boost C++ Libraries. – Режим доступу: https://www.boost.org/doc/libs/1_31_0/libs/thread/doc/overview.html
12. GitHub - David-Haim/concurrencpp [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/David-Haim/concurrencpp
13. Wu, D., Wang, Y., & Huang, L. (2015). An Empirical Study on C++ Concurrency Constructs [Електронний ресурс] // IEEE Xplore. – Режим доступу: https://ieeexplore.ieee.org/document/7321187
14. Teodorescu, M. (2016). Structured Concurrency in C++ [Електронний ресурс] // ACCU. – Режим доступу: https://accu.org/journals/overload/30/168/teodorescu/
15. C++ Thread Pool [Електронний ресурс] // EDUCBA. – Режим доступу: https://www.educba.com/c-plus-plus-thread-pool/
16. Qt Framework Documentation [Електронний ресурс] // Qt. – Режим доступу: https://doc.qt.io/
17. Qt Framework QImage [Електронний ресурс] // Qt. – Режим доступу: https://doc.qt.io/qt-6/qimage.html
18. Microsoft Learn. DDS File Layout [Електронний ресурс] // Microsoft. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3ddds/dx-graphics-dds-pguide#dds-file-layout
19. Microsoft Learn. DDS_HEADER Structure [Електронний ресурс] // Microsoft. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/direct3ddds/dds-header

20. Qt Framework Widget [Електронний ресурс] // Qt. – Режим доступу: https://doc.qt.io/qt-6/qtwidgets-index.html
21. Khronos. OpenGL Shader [Електронний ресурс] // Khronos. – Режим доступу: https://www.khronos.org/opengl/wiki/Shader
22. Khronos. OpenGL Vertex Shader [Електронний ресурс] // Khronos. – Режим доступу: https://www.khronos.org/opengl/wiki/Vertex_Shader
23. Khronos. OpenGL Fragment Shader [Електронний ресурс] // Khronos. – Режим доступу: https://www.khronos.org/opengl/wiki/Fragment_Shader
24. Microsoft Learn. Windows Context Switches [Електронний ресурс] // Microsoft. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/context-switches
25. Нестеренко К.П., Стеценко І.В. Програмне забезпечення для конвертації зображень у текстури // Матеріали міжнародної конференції «Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2023)». – Київ, 2023. – [Прийнята до друку].

ДОДАТКИ

ДОДАТОК А

UML діаграма класів розробленого програмного засобу



ДОДАТОК Б

Лістинг коду

Job.h

```
#pragma once

#include <JobManager/ThreadConfiguration.h>

#include <vector>
#include <memory>
#include <atomic>

class JobData;

class Job
{
public:
    virtual ~Job() = default;

    void Start();

    void AddJobData(const std::shared_ptr<JobData>&
jobData);
    JobData* GetJobData();

    virtual ThreadAffinity GetAffinity();

    void AddDependency(Job* pJob);

    bool IsReady();
    bool IsSuccessful();
```

```
protected:
    virtual void Execute() = 0;

private:
    void AddDependentJob(Job* pJob);

    void OnFinish();

private:
    bool m_bSuccessful { false };

    std::atomic_int m_dependeciesCounter { 0 };
    std::vector<Job*> m_dependentJobs;

    std::shared_ptr<JobData> m_jobData;
};
```

Job.cpp

```
#include "Job.h"

#include <JobManager/JobData.h>

#include <algorithm>

void Job::Start()
{
    Execute();

    m_bSuccessful = true;
```

```
    OnFinish();
}

void Job::AddJobData(const std::shared_ptr<JobData>
&jobData)
{
    m_jobData = jobData;
}

bool Job::IsReady()
{
    return m_dependeciesCounter == 0;
}

bool Job::IsSuccessful()
{
    return m_bSuccessful;
}

void Job::AddDependentJob(Job *pJob)
{
    m_dependentJobs.emplace_back(pJob);
}

void Job::OnFinish()
{
    for (const auto& dependentJob : m_dependentJobs)
    {
        dependentJob->m_dependeciesCounter--;
    }
```

```
}

JobData* Job::GetJobData()
{
    return m_jobData.get();
}

ThreadAffinity Job::GetAffinity()
{
    return ThreadAffinity::Default;
}

void Job::AddDependency(Job* pJob)
{
    pJob->AddDependentJob(this);
    m_dependeciesCounter++;
}
```

JobData.h

```
#pragma once

class JobData
{
public:
    virtual ~JobData() = default;
};
```

JobManager.h

```
#pragma once

#include <JobManager/Job.h>
```

```
#include <mutex>
#include <vector>
#include <condition_variable>
#include <queue>
#include <thread>
#include <memory>
#include <map>

class JobManager
{
public:
    void Start();
    void QueueJob(const std::shared_ptr<Job>& job);
    void Stop();
    bool IsBusy();

    void AddThreadConfiguration(const
ThreadConfiguration& configuration);

    static void
SetupBasicConfigurationWithIOThread(JobManager&
jobManager);

private:
    void ThreadLoop(const ThreadConfiguration&
configuration);
    std::shared_ptr<Job> GetRunnableJob(const
ThreadConfiguration &configuration);

private:
```

```
    bool m_bShouldTerminate { false };           // Tells
threads to stop looking for jobs

    std::mutex m_queueMutex;                     // Prevents
data races to the job queue
    std::condition_variable m_mutexCondition; // Allows
threads to wait on new jobs or termination

    std::vector<std::thread> m_threads;
    std::vector<ThreadConfiguration>
m_threadsConfiguration;

    std::map<ThreadAffinity,
std::vector<std::shared_ptr<Job>>> m_jobs;
    int m_jobsCount { 0 }; // Faster to store job count
than itterate over map; Thread safe because used only
inside of critical sections
};
```

JobManager.cpp

```
#include "JobManager.h"

void JobManager::Start()
{
    const uint32_t num_threads =
std::thread::hardware_concurrency(); // Max # of threads
the system supports
    m_threads.resize(num_threads);
    for (uint32_t i = 0; i < num_threads; i++)
    {
        m_threads.at(i) =
std::thread(&JobManager::ThreadLoop, this, i <
```

```
m_threadsConfiguration.size() ? m_threadsConfiguration[i]
: ThreadConfiguration());
    }
}

void JobManager::QueueJob(const std::shared_ptr<Job>
&job)
{
    {
        std::unique_lock<std::mutex> lock(m_queueMutex);
        m_jobs[job->GetAffinity()].push_back(job);
        m_jobsCount++;
    }
}

void JobManager::Stop()
{
    {
        std::unique_lock<std::mutex> lock(m_queueMutex);
        m_bShouldTerminate = true;
    }

    for (std::thread& activeThread : m_threads) {
        activeThread.join();
    }
    m_threads.clear();
}

bool JobManager::IsBusy()
{
    bool poolbusy;
```

```
    {
        std::unique_lock<std::mutex> lock(m_queueMutex);
        poolbusy = m_jobs.empty();
    }
    return poolbusy;
}

void JobManager::AddThreadConfiguration(const
ThreadConfiguration &configuration)
{
    m_threadsConfiguration.push_back(configuration);
}

void
JobManager::SetupBasicConfigurationWithIOThread(JobManage
r &jobManager)
{

jobManager.AddThreadConfiguration(ThreadConfiguration(Thr
eadAffinity::IO, /*bStrict*/ false));
}

void JobManager::ThreadLoop(const ThreadConfiguration
&configuration)
{
    while (true)
    {
        std::shared_ptr<Job> job = nullptr;
        {
            std::unique_lock<std::mutex>
lock(m_queueMutex);
```

```
            job = GetRunnableJob(configuration);

            if (!job && m_bShouldTerminate && m_jobsCount
== 0)
            {
                return;
            }
        }

        if (job)
        {
            job->Start();
        }
        else
        {
            std::this_thread::yield();
        }
    }
}

std::shared_ptr<Job> JobManager::GetRunnableJob(const
ThreadConfiguration &configuration)
{
    std::shared_ptr<Job> result = nullptr;
    const auto& affinitiveJobs =
m_jobs.find(configuration.GetAffinity());
    if (affinitiveJobs != m_jobs.end())
    {
        for (std::vector<std::shared_ptr<Job>>::iterator
it = affinitiveJobs->second.begin(); it !=
affinitiveJobs->second.end();)
```

```
        {
            if ((*it)->IsReady())
            {
                result = *it;
                affinitiveJobs->second.erase(it);
                break;
            }
            else
            {
                it++;
            }
        }
    }

    if (!configuration.IsStrict() && !result)
    {
        for (ThreadAffinity affinity =
ThreadAffinity::Default; affinity <
ThreadAffinity::Strict; affinity =
static_cast<ThreadAffinity>(static_cast<int>(affinity) +
1))
        {
            const auto& affinitiveJobs =
m_jobs.find(affinity);
            if (affinitiveJobs != m_jobs.end())
            {
                for
(std::vector<std::shared_ptr<Job>>::iterator it =
affinitiveJobs->second.begin(); it != affinitiveJobs-
>second.end();)
                {
                    if ((*it)->IsReady())
```

```
                    {
                        result = *it;
                        affinitiveJobs->second.erase(it);
                        break;
                    }
                    else
                    {
                        it++;
                    }
                }
            }
        }

    if (result)
    {
        m_jobsCount--;
    }

    return result;
}
```

ThreadConfiguration.h

```
#pragma once

enum class ThreadAffinity
{
    Default = 0,
    Strict, // Put all affinity types that should be
executed only in specialized thread after this
```

```
    IO,
    Count // Add new affinity types before this
};

class ThreadConfiguration
{
public:
    ThreadConfiguration() = default;
    ThreadConfiguration(ThreadAffinity affinity, bool
bStrict = false);

    bool IsStrict() const;
    ThreadAffinity GetAffinity() const;

private:
    ThreadAffinity m_affinity { ThreadAffinity::Default
};
    bool m_bStrict { false };
};
```

ThreadConfiguration.cpp

```
#include "ThreadConfiguration.h"

ThreadConfiguration::ThreadConfiguration(ThreadAffinity
affinity, bool bStrict)
    : m_affinity(affinity)
    , m_bStrict(bStrict)
{
}

bool ThreadConfiguration::IsStrict() const
```

```
{
    return m_bStrict;
}

ThreadAffinity ThreadConfiguration::GetAffinity() const
{
    return m_affinity;
}
```

BatchJob.h

```
#pragma once

#include <JobManager/Job.h>

class BatchJob
    : public Job
{
public:
    void AddJob(std::shared_ptr<Job>& job);

private:
    virtual void Execute() override;

private:
    std::vector<std::shared_ptr<Job>> m_jobs;
};
```

BatchJob.cpp

```
#include "BatchJob.h"
```

```
void BatchJob::AddJob(std::shared_ptr<Job> &job)
{
    m_jobs.emplace_back(job);
}

void BatchJob::Execute()
{
    for (const std::shared_ptr<Job>& job : m_jobs)
    {
        job->Start();
    }
}
```

ConvertImageJob.h

```
#pragma once

#include <JobManager/Job.h>

class ImageConversionJobData;

class ConvertImageJob
    : public Job
{
public:

ConvertImageJob(std::shared_ptr<ImageConversionJobData>&
jobData);

ConvertImageJob(std::shared_ptr<ImageConversionJobData>&
jobData, int texelBegin, int texelEnd);
```

```
private:
    virtual void Execute() override;

private:
    int m_texelBegin { 0 };
    int m_texelEnd { 0 };
};
```

ConvertImageJob.cpp

```
#include "ConvertImageJob.h"

#include <ProjectDefines.h>

#include <DDSConversion/Jobs/ImageConversionJobData.h>

ConvertImageJob::ConvertImageJob(std::shared_ptr<ImageCon
versionJobData> &jobData)
{
    AddJobData(jobData);
}

ConvertImageJob::ConvertImageJob(std::shared_ptr<ImageCon
versionJobData> &jobData, int texelBegin, int texelEnd)
{
    AddJobData(jobData);
    m_texelBegin = texelBegin;
    m_texelEnd = texelEnd;
}

void ConvertImageJob::Execute()
{
```

```
    assertm(GetJobData(), "ImageConversionJobData has to
be set prior to this job execution");
    ImageConversionJobData* jobData =
reinterpret_cast<ImageConversionJobData*>(GetJobData());

    assertm(jobData->m_outputData, "Output data should be
created prior to this job execution");
    jobData->m_outputData->ConvertDataCPU(jobData-
>m_texels);
}
```

ImageConversionJobData.h

```
#pragma once

#include <JobManager/JobData.h>

#include <DDSConversion/DDSDefines.h>
#include <DDSConversion/Textures/ImageTexel.h>
#include <DDSConversion/Textures/TextureData.h>

#include <vector>

class ImageConversionJobData
        : public JobData
{
public:
    ~ImageConversionJobData();

    ImageConversionJobData(std::string sourceFile,
std::string outputFile, CompressionType compressionType,
TextureOrientation textureOrientation);
```

```
public:
    std::string m_sourceFile;
    std::string m_outputFile;

    CompressionType m_compressionType {
CompressionType::DXT1 };
    TextureOrientation m_textureOrientation {
TextureOrientation::DirectX };

    std::vector<ImageTexel> m_texels;
    TextureData* m_outputData { nullptr };
};
```

ImageConversionJobData.cpp

```
#include "ImageConversionJobData.h"

ImageConversionJobData::~ImageConversionJobData()
{
    if (m_outputData)
    {
        delete m_outputData;
    }
}

ImageConversionJobData::ImageConversionJobData(std::strin
g sourceFile, std::string outputFile, CompressionType
compressionType, TextureOrientation textureOrientation)
    : m_sourceFile(sourceFile)
    , m_outputFile(outputFile)
    , m_compressionType(compressionType)
```

```
    , m_textureOrientation(textureOrientation)
{
}
```

SaveTextureJob.h

```
#pragma once

#include <JobManager/Job.h>
#include <DDSConversion/Jobs/ImageConversionJobData.h>

class SaveTextureJob
    : public Job
{
public:

SaveTextureJob(std::shared_ptr<ImageConversionJobData>&
jobData);

    virtual ThreadAffinity GetAffinity() override;

private:
    virtual void Execute() override;
};
```

SaveTextureJob.cpp

```
#include "SaveTextureJob.h"

SaveTextureJob::SaveTextureJob(std::shared_ptr<ImageConve
rsionJobData> &jobData)
{
```

```
    AddJobData(jobData);
}

ThreadAffinity SaveTextureJob::GetAffinity()
{
    return ThreadAffinity::IO;
}

void SaveTextureJob::Execute()
{
    assertm(GetJobData(), "ImageConversionJobData has to
be set prior to job execution");
    ImageConversionJobData* jobData =
reinterpret_cast<ImageConversionJobData*>(GetJobData());

    std::ofstream outputImage;
    outputImage.open(jobData->m_outputFile, std::ios::out
| std::ios::binary);

    jobData->m_outputData->Serialize(outputImage);
}
```

SplitImageJob.h

```
#pragma once

#include <JobManager/Job.h>

class QImage;
class ImageConversionJobData;

class SplitImageJob
```

```
    : public Job
{
public:
    SplitImageJob(std::string sourceImage,
std::shared_ptr<ImageConversionJobData>& jobData);
    ~SplitImageJob();

private:
    virtual void Execute() override;

private:
    QImage* m_sourceImage { nullptr };
};
```

SplitImageJob.cpp

```
#include "SplitImageJob.h"

#include <ProjectDefines.h>

#include <DDSConversion/Textures/ImageUtils.h>
#include <DDSConversion/Textures/TextureBlock.h>

#include <QImage>

#include "ImageConversionJobData.h"

SplitImageJob::SplitImageJob(std::string sourceImage,
std::shared_ptr<ImageConversionJobData>& jobData)
{
    m_sourceImage = new QImage(sourceImage.c_str());
    AddJobData(jobData);
```

```
}

SplitImageJob::~SplitImageJob()
{
    if (m_sourceImage)
    {
        delete m_sourceImage;
    }
}

void SplitImageJob::Execute()
{
    assertm(GetJobData(), "ImageConversionJobData has to
be set prior to job execution");
    ImageConversionJobData* jobData =
reinterpret_cast<ImageConversionJobData*>(GetJobData());

    ConvertQImageToTexelArray(*m_sourceImage, jobData-
>m_texels, jobData->m_textureOrientation);
    jobData->m_outputData = new TextureData(jobData-
>m_compressionType, m_sourceImage->height(),
m_sourceImage->width(), jobData->m_texels.size());

    delete m_sourceImage;
    m_sourceImage = nullptr;
}
```

ImageTexel.h

```
#pragma once

#include <ProjectDefines.h>
```

```
#include <DDSConversion/DDSDefines.h>
#include <DDSConversion/Color32.h>

#include <vector>


class ImageTexel
{
public:
    ImageTexel() = default;

public:
    Color32 m_pixelData[16];
};
```

ImageUtils.h

```
#pragma once

#include <QImage>
#include <QColor>

#include <DDSConversion/Textures/ImageTexel.h>

Color32 CreateColor32FromQColor(const QColor& source);
void ConvertQImageToTexelArray(const QImage& source,
std::vector<ImageTexel>& texelArray, TextureOrientation
textureOrientation);

void FillDDSHeader(DDS_HEADER& header, CompressionType
type, int width, int height);
```

ImageUtils.cpp

```
#include "ImageUtils.h"

Color32 CreateColor32FromQColor(const QColor& source)
{
    return Color32(source.red(), source.green(),
source.blue(), source.alpha());
}

void ConvertQImageToTexelArray(const QImage& source,
std::vector<ImageTexel>& texelArray, TextureOrientation
textureOrientation)
{
    if (textureOrientation ==
TextureOrientation::DirectX)
    {
        for (int i = 0; i < source.height(); i+=4)
        {
            for (int j = 0; j < source.width(); j+=4)
            {
                int pixelCount = 0;
                texelArray.push_back(ImageTexel());

                for (int y = i; y < i + 4; y++)
                {
                    for (int x = j; x < j + 4; x++)
                    {
                        if (y >= source.height() || x >=
source.width())
                        {
```

```
texelArray.back().m_pixelData[pixelCount] = Color32(0, 0,
0, 0);
                        }
                        else
                        {

texelArray.back().m_pixelData[pixelCount] =
CreateColor32FromQColor(source.pixelColor(x, y));
                        }
                        pixelCount++;
                    }
                }
            }
        }
    }
    else
    {
        for (int i = source.height() - 1; i >= 0; i-=4)
        {
            for (int j = 0; j < source.width(); j+=4)
            {
                int pixelCount = 0;
                texelArray.push_back(ImageTexel());

                for (int y = i; y > i - 4; y--)
                {
                    for (int x = j; x < j + 4; x++)
                    {
                        if (y < 0 || x >= source.width())
                        {
```

```
texelArray.back().m_pixelData[pixelCount] = Color32(0, 0,
0, 0);
                        }
                        else
                        {

texelArray.back().m_pixelData[pixelCount] =
CreateColor32FromQColor(source.pixelColor(x, y));
                        }
                        pixelCount++;
                    }
                }
            }
        }
    }
}

void FillDDSHeader(DDS_HEADER &header, CompressionType
type, int width, int height)
{
    header.dwMagic = ('D' << 0) | ('D' << 8) | ('S' <<
16) | (' ' << 24);
    header.dwSize = 124;
    header.dwFlags = (DDSD_CAPS) | (DDSD_HEIGHT) |
(DDSD_WIDTH) | (DDSD_PIXELFORMAT) | (DDSD_LINEARSIZE);
    header.dwHeight = static_cast<DWORD>(height);
    header.dwWidth = static_cast<DWORD>(width);

    switch (type)
    {
```

```
        case (CompressionType::DXT1):
        {
            header.dwPitchOrLinearSize =
static_cast<DWORD>(std::max(1, ((width + 3) / 4)) * 8);
            header.ddspf.dwFourCC = ('D' << 0) | ('X' <<
8) | ('T' << 16) | ('1' << 24);
            break;
        }
        case (CompressionType::DXT3):
        {
            header.dwPitchOrLinearSize =
static_cast<DWORD>(std::max(1, ((width + 3) / 4)) * 16);
            header.ddspf.dwFourCC = ('D' << 0) | ('X' <<
8) | ('T' << 16) | ('3' << 24);
            break;
        }
        case (CompressionType::DXT5):
        {
            header.dwPitchOrLinearSize =
static_cast<DWORD>(std::max(1, ((width + 3) / 4)) * 16);
            header.ddspf.dwFourCC = ('D' << 0) | ('X' <<
8) | ('T' << 16) | ('5' << 24);
            break;
        }
    }

    header.ddspf.dwSize = 32;
    header.ddspf.dwFlags = DDPF_FOURCC;
    header.dwCaps = DDSD_CAPS;
}
```

TextureBlock.h

```
#pragma once

#include <ProjectDefines.h>

#include <DDSConversion/Textures/ImageTexel.h>

#include <fstream>

class TextureBlock
{
public:
    virtual ~TextureBlock() = default;

    virtual void Serialize(std::ofstream& fileStream) =
0;
    virtual void ConvertDataFromTexel(const ImageTexel&
texel) = 0;

protected:
    void GetMainColors(const ImageTexel& texel, Color32&
firstColor, Color32& secondColor) const;
    uint32_t CalculateColorIndices(const ImageTexel&
texel, const Color32(&mainColors)[4]) const;

protected:
    uint16_t m_firstColor { 0 };
    uint16_t m_secondColor { 0 };
    uint32_t m_indices { 0 };
};
```

```
class BC1Block
    : public TextureBlock
{
public:
    virtual void Serialize(std::ofstream& fileStream)
override;
    virtual void ConvertDataFromTexel(const ImageTexel&
texel) override;

private:
    bool HasAlpha(const ImageTexel& texel);
    void CalculateContentIndicesBC1a(uint32_t& indices,
const ImageTexel& texel, const Color32(&mainColors)[4])
const;
};

class BC2Block
    : public TextureBlock
{
public:
    virtual void Serialize(std::ofstream& fileStream)
override;
    virtual void ConvertDataFromTexel(const ImageTexel&
texel) override;

private:
    void CalculateContentIndicesBC2(uint32_t& indices,
uint64_t& alphaContent, const ImageTexel& texel, const
Color32(&mainColors)[4]) const;

private:
    uint64_t m_alphaContent { 0 };
```

```
};

class BC3Block
    : public TextureBlock
{
public:
    virtual void Serialize(std::ofstream& fileStream)
override;
    virtual void ConvertDataFromTexel(const ImageTexel&
texel) override;

private:
    void CalculateContentIndicesBC3(uint32_t& indices,
uint64_t& alphaContent, const ImageTexel& texel, const
Color32(&mainColors)[4], const int(&mainAlpha)[8]) const;
    void GetMainColorsAndAlpha(const ImageTexel& texel,
Color32& firstColor, Color32& secondColor, int&
firstAlpha, int& secondAlpha) const;

private:
    uint64_t m_alphaContent { 0 };
};
```

TextureBlock.cpp

```
#include "TextureBlock.h"

#include "ImageTexel.h"

#include <fstream>

void BC1Block::Serialize(std::ofstream& fileStream)
```

```
{

fileStream.write(reinterpret_cast<char*>(&m_firstColor),
sizeof(m_firstColor));

fileStream.write(reinterpret_cast<char*>(&m_secondColor),
sizeof(m_secondColor));
    fileStream.write(reinterpret_cast<char*>(&m_indices),
sizeof(m_indices));
}

void BC1Block::ConvertDataFromTexel(const ImageTexel&
texel)
{
    Color32 colors[4];
    GetMainColors(texel, colors[0], colors[1]);

    m_firstColor = colors[0].GetRGB565();
    m_secondColor = colors[1].GetRGB565();

    if (HasAlpha(texel) || m_firstColor == m_secondColor)
    {
        if (m_firstColor > m_secondColor)
        {
            std::swap(m_firstColor, m_secondColor);
            std::swap(colors[0], colors[1]);
        }
        colors[2] = Color32(colors[0].red * 1 / 2 +
colors[1].red * 1 / 2, colors[0].green * 1 / 2
            + colors[1].green * 1 / 2, colors[0].blue * 1
/ 2 + colors[1].blue * 1 / 2, 255);
        colors[3] = Color32(0, 0, 0, 0);
```

```
        CalculateContentIndicesBC1a(m_indices, texel,
colors);
    }
    else
    {
        if (m_firstColor < m_secondColor)
        {
            std::swap(m_firstColor, m_secondColor);
            std::swap(colors[0], colors[1]);
        }
        colors[2] = Color32(colors[0].red * 2 / 3 +
colors[1].red * 1 / 3, colors[0].green * 2 / 3
            + colors[1].green * 1 / 3, colors[0].blue * 2
/ 3 + colors[1].blue * 1 / 3, 255);
        colors[3] = Color32(colors[0].red * 1 / 3 +
colors[1].red * 2 / 3, colors[0].green * 1 / 3
            + colors[1].green * 2 / 3, colors[0].blue * 1
/ 3 + colors[1].blue * 2 / 3, 255);

        m_indices = CalculateColorIndices(texel, colors);
    }
}

bool BC1Block::HasAlpha(const ImageTexel& texel)
{
    for (int i = 0; i < 16; i++)
    {
        if (texel.m_pixelData[i].alpha == 0)
        {
            return true;
```

```
        }
    }
    return false;
}

void BC1Block::CalculateContentIndicesBC1a(uint32_t&
indices, const ImageTexel &texel, const Color32
(&mainColors)[4]) const
{
    indices = 0;
    for (int i = 15; i >= 0; i--)
    {
        uint32_t pixelColorIndex = 0;
        if (texel.m_pixelData[i].alpha == 0)
        {
            pixelColorIndex = 3;
        }
        else
        {
            int minDistance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[0]);
            for (uint32_t colorIndex = 1; colorIndex < 3;
colorIndex++)
            {
                int distance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[colorIndex]);
                if (distance < minDistance)
                {
                    pixelColorIndex = colorIndex;
```

```
                    minDistance = distance;
                }
            }
        }

        indices = indices << 2;
        indices +=
static_cast<uint32_t>(pixelColorIndex);
    }
}

void TextureBlock::GetMainColors(const ImageTexel& texel,
Color32& firstColor, Color32& secondColor) const
{
    float max = 0;
    firstColor = texel.m_pixelData[0];
    secondColor = texel.m_pixelData[0];

    for (int i = 0; i < 16; i++)
    {
        for (int j = i + 1; j < 16; j++)
        {
            if (i != j)
            {
                float res =
texel.m_pixelData[i].CalculateColorDistanceRelative(texel
.m_pixelData[j]);
                if (res > max)
                {
                    max = res;
                    firstColor = texel.m_pixelData[i];
```

```
                    secondColor = texel.m_pixelData[j];
                }
            }
        }
    }
}

uint32_t TextureBlock::CalculateColorIndices(const
ImageTexel& texel, const Color32(&mainColors)[4]) const
{
    uint32_t indices = 0;
    for (int i = 15; i >= 0; i--)
    {
        uint32_t pixelColorIndex = 0;
        float minDistance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[0]);

        for (uint32_t colorIndex = 1; colorIndex < 4;
colorIndex++)
        {
            int distance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[colorIndex]);
            if (distance < minDistance)
            {
                pixelColorIndex = colorIndex;
                minDistance = distance;
            }
        }
```

```
        indices = indices << 2;
        indices +=
static_cast<uint32_t>(pixelColorIndex);
    }

    return indices;
}

void BC2Block::Serialize(std::ofstream &fileStream)
{

fileStream.write(reinterpret_cast<char*>(&m_alphaContent)
, sizeof(m_alphaContent));

fileStream.write(reinterpret_cast<char*>(&m_firstColor),
sizeof(m_firstColor));

fileStream.write(reinterpret_cast<char*>(&m_secondColor),
sizeof(m_secondColor));
    fileStream.write(reinterpret_cast<char*>(&m_indices),
sizeof(m_indices));
}

void BC2Block::ConvertDataFromTexel(const ImageTexel
&texel)
{
    Color32 colors[4];
    GetMainColors(texel, colors[0], colors[1]);
    colors[2] = Color32(colors[0].red * 2 / 3 +
colors[1].red * 1 / 3, colors[0].green * 2 / 3
        + colors[1].green * 1 / 3, colors[0].blue * 2 / 3
+ colors[1].blue * 1 / 3, 255);
```

```
    colors[3] = Color32(colors[0].red * 1 / 3 +
colors[1].red * 2 / 3, colors[0].green * 1 / 3
        + colors[1].green * 2 / 3, colors[0].blue * 1 / 3
+ colors[1].blue * 2 / 3, 255);

    m_firstColor = colors[0].GetRGB565();
    m_secondColor = colors[1].GetRGB565();

    CalculateContentIndicesBC2(m_indices, m_alphaContent,
texel, colors);
}

void BC2Block::CalculateContentIndicesBC2(uint32_t
&indices, uint64_t& alphaContent, const ImageTexel
&texel, const Color32 (&mainColors)[4]) const
{
    for (int i = 15; i >= 0; i--)
    {
        uint32_t pixelColorIndex = 0;
        float minDistance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[0]);

        for (uint32_t colorIndex = 1; colorIndex < 4;
colorIndex++)
        {
            int distance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[colorIndex]);
            if (distance < minDistance)
            {
                pixelColorIndex = colorIndex;
```

```
                minDistance = distance;
            }
        }

        indices = indices << 2;
        indices +=
static_cast<uint32_t>(pixelColorIndex);

        alphaContent = alphaContent << 4;
        alphaContent +=
static_cast<uint64_t>(texel.m_pixelData[i].alpha >> 4);
    }
}

void BC3Block::Serialize(std::ofstream &fileStream)
{

fileStream.write(reinterpret_cast<char*>(&m_alphaContent)
, sizeof(m_alphaContent));

fileStream.write(reinterpret_cast<char*>(&m_firstColor),
sizeof(m_firstColor));

fileStream.write(reinterpret_cast<char*>(&m_secondColor),
sizeof(m_secondColor));
    fileStream.write(reinterpret_cast<char*>(&m_indices),
sizeof(m_indices));
}

void BC3Block::ConvertDataFromTexel(const ImageTexel
&texel)
{
```

```
    Color32 colors[4];
    int alpha[8];

    GetMainColorsAndAlpha(texel, colors[0], colors[1],
alpha[0], alpha[1]);

    colors[2] = Color32(colors[0].red * 2 / 3 +
colors[1].red * 1 / 3, colors[0].green * 2 / 3
        + colors[1].green * 1 / 3, colors[0].blue * 2 / 3
+ colors[1].blue * 1 / 3, 255);
    colors[3] = Color32(colors[0].red * 1 / 3 +
colors[1].red * 2 / 3, colors[0].green * 1 / 3
        + colors[1].green * 2 / 3, colors[0].blue * 1 / 3
+ colors[1].blue * 2 / 3, 255);

    m_firstColor = colors[0].GetRGB565();
    m_secondColor = colors[1].GetRGB565();

    if (alpha[0] < alpha[1])
    {
        std::swap(alpha[0], alpha[1]);
    }

    alpha[2] = (6 * alpha[0] + 1 * alpha[1]) / 7;
    alpha[3] = (5 * alpha[0] + 2 * alpha[1]) / 7;
    alpha[4] = (4 * alpha[0] + 3 * alpha[1]) / 7;
    alpha[5] = (3 * alpha[0] + 4 * alpha[1]) / 7;
    alpha[6] = (2 * alpha[0] + 5 * alpha[1]) / 7;
    alpha[7] = (1 * alpha[0] + 6 * alpha[1]) / 7;
```

```
    CalculateContentIndicesBC3(m_indices, m_alphaContent,
texel, colors, alpha);

    m_alphaContent = m_alphaContent << 8;
    m_alphaContent += alpha[1];
    m_alphaContent = m_alphaContent << 8;
    m_alphaContent += alpha[0];
}

void BC3Block::CalculateContentIndicesBC3(uint32_t
&indices, uint64_t& alphaContent, const ImageTexel
&texel, const Color32 (&mainColors)[4], const int
(&mainAlpha)[8]) const
{
    for (int i = 15; i >= 0; i--)
    {
        uint32_t pixelColorIndex = 0;
        float minColorDistance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[0]);

        for (uint32_t colorIndex = 1; colorIndex < 4;
colorIndex++)
        {
            int distance =
texel.m_pixelData[i].CalculateColorDistanceRelative(mainC
olors[colorIndex]);
            if (distance < minColorDistance)
            {
                pixelColorIndex = colorIndex;
                minColorDistance = distance;
            }
```

```
        }

        indices = indices << 2;
        indices +=
static_cast<uint32_t>(pixelColorIndex);

        uint64_t pixelAlphaIndex = 0;
        int minAlphaDistance =
std::abs(texel.m_pixelData[i].alpha - mainAlpha[0]);

        for (uint64_t alphaIndex = 1; alphaIndex < 8;
alphaIndex++)
        {
            int distance =
std::abs(texel.m_pixelData[i].alpha -
mainAlpha[alphaIndex]);
            if (distance < minAlphaDistance)
            {
                pixelAlphaIndex = alphaIndex;
                minAlphaDistance = distance;
            }
        }

        alphaContent = alphaContent << 3;
        alphaContent +=
static_cast<uint64_t>(pixelAlphaIndex);
    }
}

void BC3Block::GetMainColorsAndAlpha(const ImageTexel
&texel, Color32 &firstColor, Color32 &secondColor, int
&firstAlpha, int &secondAlpha) const
```

```
{
    float maxColorDif = 0.0f;
    firstColor = texel.m_pixelData[0];
    secondColor = texel.m_pixelData[0];

    int maxAlphaDif = 0;
    firstAlpha = texel.m_pixelData[0].alpha;
    secondAlpha = texel.m_pixelData[0].alpha;

    for (int i = 0; i < 16; i++)
    {
        for (int j = i + 1; j < 16; j++)
        {
            if (i != j)
            {
                float colorDif =
texel.m_pixelData[i].CalculateColorDistanceRelative(texel
.m_pixelData[j]);
                if (colorDif > maxColorDif)
                {
                    maxColorDif = colorDif;
                    firstColor = texel.m_pixelData[i];
                    secondColor = texel.m_pixelData[j];
                }

                int alphaDif =
std::abs(texel.m_pixelData[i].alpha -
texel.m_pixelData[j].alpha);
                if (alphaDif > maxAlphaDif)
                {
                    maxAlphaDif = alphaDif;
```

```
                    firstAlpha =
texel.m_pixelData[i].alpha;
                    secondAlpha =
texel.m_pixelData[j].alpha;
                }
            }
        }
    }
}
```

TextureData.h

```
#pragma once

#include <ProjectDefines.h>

#include <DDSConversion/Textures/TextureBlock.h>
#include <atomic>

class TextureData
{
public:
    TextureData(CompressionType type,  int height, int
width, int texelCount);
    ~TextureData();

    void ConvertDataCPU(std::vector<ImageTexel>&
texelArray);
    void ConvertDataCPU(std::vector<ImageTexel>&
texelArray, int texelBegin, int texelEnd);
    void Serialize(std::ofstream& fileStream);
```

```
private:
    std::vector<TextureBlock*> m_blockData;

    // Required to generate DDS header
    CompressionType m_compressionType { CompressionType::DXT1 };
    int m_height { 0 };
    int m_width { 0 };
};
```

TextureData.cpp

```
#include "TextureData.h"

#include <DDSConversion/Textures/ImageUtils.h>

TextureData::TextureData(CompressionType type, int height, int width, int texelCount)
    : m_compressionType(type)
    , m_height(height)
    , m_width(width)
{
    m_blockData.resize(texelCount);

    for (int index = 0; index < texelCount; index++)
    {
        switch (m_compressionType)
        {
        case CompressionType::DXT1:
            m_blockData[index] = new BC1Block();
            break;
        case CompressionType::DXT3:
```

```
            m_blockData[index] = new BC2Block();
            break;
        case CompressionType::DXT5:
            m_blockData[index] = new BC3Block();
            break;
        default:
            break;
        }
    }
}

TextureData::~TextureData()
{
    for (TextureBlock* pBlock : m_blockData)
    {
        delete pBlock;
    }
}

void TextureData::ConvertDataCPU(std::vector<ImageTexel>&
texelArray)
{
    for (size_t i = 0; i < texelArray.size(); i++)
    {
        m_blockData[i]-
>ConvertDataFromTexel(texelArray[i]);
    }
}

void TextureData::ConvertDataCPU(std::vector<ImageTexel>
&texelArray, int texelBegin, int texelEnd)
```

```
{
    for (int i = texelBegin; i < texelEnd; i++)
    {
        m_blockData[i]-
>ConvertDataFromTexel(texelArray[i]);
    }
}

void TextureData::Serialize(std::ofstream& fileStream)
{
    DDS_HEADER header;
    FillDDSHeader(header, m_compressionType, m_width,
m_height);
    fileStream.write(reinterpret_cast<char*>(&header),
sizeof(DDS_HEADER));

    for (TextureBlock* pTextureBlock : m_blockData)
    {
        pTextureBlock->Serialize(fileStream);
    }
}
```

Color32.h

```
#pragma once

#include <stdint.h>

class Color32
{
public:
    Color32() = default;
```

```
    Color32(int red, int green, int blue);
    Color32(int red, int green, int blue, int alpha);

    float CalculateColorDistance(const Color32
otherColor) const;
    float CalculateColorDistanceRelative(const Color32
otherColor) const; // Faster but can be used only when
comparing two colors
    uint16_t GetRGB565() const;

public:
    int red { 0 };
    int green { 0 };
    int blue { 0 };
    int alpha { 0 };
};

inline bool operator< (const Color32& lhs, const Color32&
rhs)
{
    return (lhs.red * lhs.red + lhs.green * lhs.green +
lhs.blue * lhs.blue) <
        (rhs.red * rhs.red + rhs.green * rhs.green +
rhs.blue * rhs.blue);
}
inline bool operator> (const Color32& lhs, const Color32&
rhs) { return rhs < lhs; }
inline bool operator<=(const Color32& lhs, const Color32&
rhs) { return !(lhs > rhs); }
inline bool operator>=(const Color32& lhs, const Color32&
rhs) { return !(lhs < rhs); }
```

```
inline bool operator==(const Color32& lhs, const Color32&
rhs)
{
    return (lhs.red * lhs.red + lhs.green * lhs.green +
lhs.blue * lhs.blue) ==
        (rhs.red * rhs.red + rhs.green * rhs.green +
rhs.blue * rhs.blue);
}
inline bool operator!=(const Color32& lhs, const Color32&
rhs) { return !(lhs == rhs); }
```

Color32.cpp

```
#include "Color32.h"

#include <math.h>

Color32::Color32(int red, int green, int blue)
{
    this->red = red;
    this->green = green;
    this->blue = blue;
}

Color32::Color32(int red, int green, int blue, int alpha)
{
    this->red = red;
    this->green = green;
    this->blue = blue;
    this->alpha = alpha;
}
```

```
float Color32::CalculateColorDistance(const Color32
otherColor) const
{
    return sqrtf(
        (red - otherColor.red) * (red - otherColor.red) +
        (green - otherColor.green) * (green -
otherColor.green) +
        (blue - otherColor.blue) * (blue -
otherColor.blue)
    );
}

float Color32::CalculateColorDistanceRelative(const
Color32 otherColor) const
{
    return
        (red - otherColor.red) * (red - otherColor.red) +
        (green - otherColor.green) * (green -
otherColor.green) +
        (blue - otherColor.blue) * (blue -
otherColor.blue)
    ;
}

unsigned short Color32::GetRGB565() const
{
    unsigned short B = (blue >> 3);
    unsigned short G = ((green >> 2) << 5);
    unsigned short R = ((red >> 3) << 11);

    return static_cast<unsigned short int> (R | G | B);
```

```
}
```

DDSDefines.h

```
#pragma once

#include <windows.h>

#define DDSD_CAPS           0x1
#define DDSD_HEIGHT         0x2
#define DDSD_WIDTH          0x4
#define DDSD_PIXELFORMAT    0x1000
#define DDSD_LINEARSIZE     0x80000
#define DDSCAPS_TEXTURE     0x1000
#define DDPF_FOURCC         0x4

enum class TextureOrientation
{
    DirectX,
    OpenGL
};

enum class CompressionType
{
    DXT1,
    DXT3,
    DXT5
};

struct DDS_PIXELFORMAT
{
```

```
    DWORD dwSize{ 0 };
    DWORD dwFlags{ 0 };
    DWORD dwFourCC{ 0 };
    DWORD dwRGBBitCount{ 0 };
    DWORD dwRBitMask{ 0 };
    DWORD dwGBitMask{ 0 };
    DWORD dwBBitMask{ 0 };
    DWORD dwABitMask{ 0 };
};

struct DDS_HEADER
{
    DWORD           dwMagic{ 0 };
    DWORD           dwSize{ 0 };
    DWORD           dwFlags{ 0 };
    DWORD           dwHeight{ 0 };
    DWORD           dwWidth{ 0 };
    DWORD           dwPitchOrLinearSize{ 0 };
    DWORD           dwDepth{ 0 };
    DWORD           dwMipMapCount{ 0 };
    DWORD           dwReserved1[11];
    DDS_PIXELFORMAT ddspf;
    DWORD           dwCaps{ 0 };
    DWORD           dwCaps2{ 0 };
    DWORD           dwCaps3{ 0 };
    DWORD           dwCaps4{ 0 };
    DWORD           dwReserved2{ 0 };
};

struct Texture
```

```
{
    ~Texture()
    {
        if (m_data)
        {
            delete[] m_data;
        }
    }

    DDS_HEADER m_header;
    char* m_data { nullptr };
};
```

DDSFunc.h

```
#pragma once

#include <string>
#include <vector>

#include <DDSConversion/DDSDefines.h>

class JobManager;

void
ConvertImagesIntoDDS(std::vector<std::pair<std::string,
std::string>> imageFiles, CompressionType type,
TextureOrientation textureOrientation);
void CreateJobForImageConversion(JobManager& jobManager,
std::string& inputFile, std::string& outputFile,
CompressionType type, TextureOrientation
textureOrientation);
```

```
void ReadDDSTexture(std::string filePath, Texture&
outTexture);
```

DDSFunc.cpp

```
#include "DDSFunc.h"

#include <JobManager/JobManager.h>

#include <DDSConversion/Jobs/ConvertImageJob.h>
#include <DDSConversion/Jobs/SplitImageJob.h>
#include <DDSConversion/Jobs/SaveTextureJob.h>
#include <DDSConversion/Jobs/ImageConversionJobData.h>

#include <iostream>

void
ConvertImagesIntoDDS(std::vector<std::pair<std::string,
std::string>> imageFiles, CompressionType type,
TextureOrientation textureOrientation)
{
    auto start = std::chrono::steady_clock::now();

    JobManager jobManager;

JobManager::SetupBasicConfigurationWithIOThread(jobManage
r);

    for (auto& pair : imageFiles)
    {
```

```
        CreateJobForImageConversion(jobManager,
pair.first, pair.second, type, textureOrientation);
    }

    jobManager.Start();
    jobManager.Stop();

    auto end = std::chrono::steady_clock::now();
    auto diff = end - start;
    std::cout << "Job manager: " <<
std::chrono::duration<double, std::milli>(diff).count()
<< " ms" << std::endl;
}

void CreateJobForImageConversion(JobManager &jobManager,
std::string &inputFile, std::string &outputFile,
CompressionType type, TextureOrientation
textureOrientation)
{
    std::shared_ptr<ImageConversionJobData> jobData =
std::make_shared<ImageConversionJobData>(inputFile,
outputFile, type, textureOrientation);

    std::shared_ptr<Job> splitJob =
std::make_shared<SplitImageJob>(inputFile, jobData);

    std::shared_ptr<Job> convertJob =
std::make_shared<ConvertImageJob>(jobData);
    convertJob->AddDependency(splitJob.get());

    std::shared_ptr<Job> saveTextureJob =
std::make_shared<SaveTextureJob>(jobData);
```

```
    saveTextureJob->AddDependency(convertJob.get());

    jobManager.QueueJob(splitJob);
    jobManager.QueueJob(convertJob);
    jobManager.QueueJob(saveTextureJob);
}

void ReadDDSTexture(std::string filePath, Texture&
outTexture)
{
    std::ifstream file(filePath, std::ios::in |
std::ios::binary);

    char* buffer = new char[sizeof(DDS_HEADER)];

    file.read(buffer, sizeof(DDS_HEADER));

    DDS_HEADER* header =
reinterpret_cast<DDS_HEADER*>(buffer);
    outTexture.m_header = *header;

    int dataSizeMultiplier = 0;

    const DWORD dxt1 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('1' << 24);
    const DWORD dxt3 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('3' << 24);
    const DWORD dxt5 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('5' << 24);

    switch (header->ddspf.dwFourCC)
```

```
    {
        case (dxt1):
        {
            dataSizeMultiplier = 8;
            break;
        }
        case(dxt3):
        {
            dataSizeMultiplier = 16;
            break;
        }
        case(dxt5):
        {
            dataSizeMultiplier = 16;
            break;
        }
    }

    unsigned long long dataSize = (header->dwWidth + (4 -
header->dwWidth % 4) % 4) * (header->dwHeight + (4 -
header->dwHeight % 4) % 4) / 16 * dataSizeMultiplier;

    if (dataSize > 0)
    {
        if (outTexture.m_data)
        {
            delete[] outTexture.m_data;
        }

        outTexture.m_data = new char[dataSize];
        file.read(outTexture.m_data, dataSize);
```

```
    }

    delete[] buffer;
    file.close();
}
```

DDSPreviewerWidget.h

```
#ifndef DDSPREVIEWERWIDGET_H
#define DDSPREVIEWERWIDGET_H

#include <Widgets/OpenGLWidget.h>
#include <DDSConversion/DDSDefines.h>

#include <QWidget>

namespace Ui {
class DDSPreviewerWidget;
}

class DDSPreviewerWidget : public QWidget
{
    Q_OBJECT

public:
    explicit DDSPreviewerWidget(QWidget *parent =
nullptr);
    ~DDSPreviewerWidget();

    void SetTexture(QString& texturePath,
TextureOrientation textureOrientation);
```

```
private slots:
    void
on_textureOrientationComboBox_currentIndexChanged(int
index);

private:
    void CreateOpenGLWidget();

private:
    Ui::DDSPreviewerWidget *ui;

    Texture m_texture;
    TextureOrientation m_textureOrientation {
TextureOrientation::DirectX };
    OpenGLWidget* m_pOpenGLWidget { nullptr };
};

#endif // DDSPREVIEWERWIDGET_H
```

DDSPreviewerWidget.cpp

```
#include "DDSPreviewerWidget.h"
#include "ui_DDSPreviewerWidget.h"

#include <DDSConversion/DDSFunc.h>

#include <math.h>

namespace WidgetSizeDefs
{
    const int maxW = 1280;
    const int maxH = 720;
```

```
    const int minW = 640;
    const int minH = 360;

    const int xOffset = 50;
    const int yOffset = 50;
}

DDSPreviewerWidget::DDSPreviewerWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::DDSPreviewerWidget)
{
    ui->setupUi(this);
}

DDSPreviewerWidget::~DDSPreviewerWidget()
{
    delete m_pOpenGLWidget;

    delete ui;
}

void DDSPreviewerWidget::SetTexture(QString& texturePath,
TextureOrientation textureOrientation)
{
    ReadDDSTexture(texturePath.toStdString(), m_texture);
    m_textureOrientation = textureOrientation;

    ui->textureOrientationComboBox-
>setCurrentIndex(static_cast<int>(m_textureOrientation));

    CreateOpenGLWidget();
```

```
}

void DDSPreviewerWidget::CreateOpenGLWidget()
{
    if (m_pOpenGLWidget)
    {
        delete m_pOpenGLWidget;
    }

    m_pOpenGLWidget = new OpenGLWidget(this);

    int widgetH = m_texture.m_header.dwHeight;
    int widgetW = m_texture.m_header.dwWidth;

    if (widgetH > WidgetSizeDefs::maxH)
    {
        float multiplier = widgetH /
WidgetSizeDefs::maxH;
        widgetH = WidgetSizeDefs::maxH;
        widgetW = round(widgetW / multiplier);
    }

    if (widgetW > WidgetSizeDefs::maxW)
    {
        float multiplier = widgetW /
WidgetSizeDefs::maxW;
        widgetW = WidgetSizeDefs::maxW;
        widgetH = round(widgetH / multiplier);
    }

    if (widgetH < WidgetSizeDefs::minH)
```

```
    {
        float multiplier = WidgetSizeDefs::minH /
widgetH;
        widgetH = WidgetSizeDefs::minH;
        widgetW = round(widgetW * multiplier);
    }

    if (widgetW < WidgetSizeDefs::minW)
    {
        float multiplier = WidgetSizeDefs::minW /
widgetW;
        widgetW = WidgetSizeDefs::minW;
        widgetH = round(widgetH * multiplier);
    }

    this->resize(widgetW + WidgetSizeDefs::xOffset * 2,
widgetH + WidgetSizeDefs::yOffset * 2);

    m_pOpenGLWidget->resize(widgetW,  widgetH);
    m_pOpenGLWidget->move(WidgetSizeDefs::xOffset,
WidgetSizeDefs::yOffset);
    m_pOpenGLWidget->SetTexture(&m_texture);
    m_pOpenGLWidget-
>SetTextureOrientation(m_textureOrientation);
}

void
DDSPreviewerWidget::on_textureOrientationComboBox_current
IndexChanged(int index)
{
    TextureOrientation newOrientation =
static_cast<TextureOrientation>(index);
```

```
    if (m_textureOrientation != newOrientation)
    {
        m_textureOrientation = newOrientation;
        CreateOpenGLWidget();
        /*
        if (m_pOpenGLWidget)
        {
            m_pOpenGLWidget-
>SetTextureOrientation(m_textureOrientation);
        }
        */
    }
}
```

MainWindow.h

```
#pragma once

#include <QMainWindow>
#include <DDSConversion/DDSDefines.h>

class QTableWidgetItem;

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```
public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private:
    CompressionType GetCurrentCompressionType();
    TextureOrientation GetCurrentTextureOrientation();

    void
AddProcessedTextures(std::vector<std::pair<std::string,
std::string>> imageFiles);

    void DisplayPreviewerWidget(QString filePath);

private slots:
    void on_runBenchmarkButton_clicked();

    void on_convertImageButton_clicked();

    void on_convertFolderButton_clicked();

    void on_processedTexturesTable_cellDoubleClicked(int
row, int column);

    void on_openTextureButton_clicked();

private:
    Ui::MainWindow *ui;
};
```

MainWindow.cpp

```
#include "MainWindow.h"
#include "ui_MainWindow.h"

#include <DDSConversion/DDSFunc.h>
#include <Widgets/DDSPreviewerWidget.h>

#include <QDir>
#include <QFileDialog>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    ui->processingProgressBar->setVisible(false);
    ui->runBenchmarkButton->setVisible(false);
    ui->processedTexturesTable->horizontalHeader()-
>setSectionResizeMode(0, QHeaderView::Stretch);
}

MainWindow::~MainWindow()
{
    delete ui;
}

CompressionType MainWindow::GetCurrentCompressionType()
{
    return static_cast<CompressionType>(ui-
>conversionModeComboBox->currentIndex());
}
```

```
TextureOrientation
MainWindow::GetCurrentTextureOrientation()
{
    return static_cast<TextureOrientation>(ui-
>textureOrientationComboBox->currentIndex());
}

void
MainWindow::AddProcessedTextures(std::vector<std::pair<st
d::string, std::string>> imageFiles)
{
    for (const auto& entry : imageFiles)
    {
        QTableWidgetItem* pItemAssetPath = new
QTableWidgetItem(entry.second.c_str());
        QTableWidgetItem* pItemCompressionType = new
QTableWidgetItem(ui->conversionModeComboBox-
>currentText());

        ui->processedTexturesTable->insertRow(this->ui-
>processedTexturesTable->rowCount());

        ui->processedTexturesTable->setItem(this->ui-
>processedTexturesTable->rowCount() - 1, 0,
pItemAssetPath);
        ui->processedTexturesTable->setItem(this->ui-
>processedTexturesTable->rowCount() - 1, 1,
pItemCompressionType);
    }
}
```

```
void MainWindow::DisplayPreviewerWidget(QString filePath)
{
    DDSPreviewerWidget* widget = new
DDSPreviewerWidget();
    widget->setAttribute(Qt::WA_DeleteOnClose);
    widget->SetTexture(filePath,
GetCurrentTextureOrientation());
    widget->setWindowTitle("DDS Previewer: " +
QString(filePath));
    widget->show();
}

void MainWindow::on_runBenchmarkButton_clicked()
{
    QString benchmarkInputFolder =
"C:/Users/Admin/Desktop/MConverter/Multithreaded-DDS-
Converter/BenchmarkImages/Folder_1";
    QString benchmarkOutputFolder =
"C:/Users/Admin/Desktop/MConverter/Multithreaded-DDS-
Converter/BenchmarkImages/Output";

    QDir inputFolder(benchmarkInputFolder);

    if(!inputFolder.entryList().isEmpty())
    {
        QStringList images =
inputFolder.entryList(QStringList() << "*.jpg" << "*.png"
<< "*.jpg" << "*.jpeg" << "*.bmp" << "*.tif" << "*.tiff",
QDir::Files);

        std::vector<std::pair<std::string, std::string>>
imagesPath;
```

```
        for (const auto& image : images)
        {
            QStringList pieces = image.split(".");
            QString fileName = pieces.value(0);
            QString outputFileName =
benchmarkOutputFolder + "/" + fileName + ".dds";


imagesPath.push_back({inputFolder.absoluteFilePath(image)
.toStdString(), outputFileName.toStdString()});
        }

        ConvertImagesIntoDDS(imagesPath,
GetCurrentCompressionType(),
GetCurrentTextureOrientation());

        AddProcessedTextures(imagesPath);
    }
}

void MainWindow::on_convertImageButton_clicked()
{
    QString inputImageName =
QFileDialog::getOpenFileName(this, tr("Open Image"), "",
tr("Image Files (*.png *.jpg *.jpeg *.bmp *.tif
*.tiff)"));

    if(!inputImageName.isEmpty() &&
!inputImageName.isNull())
    {
```

```
        QString outputTextureName =
QFileDialog::getSaveFileName(this, tr("Save Texture"),
"", tr("Texture File (*.dds)"));

        if(!outputTextureName.isEmpty()&&
!outputTextureName.isNull())
        {

ConvertImagesIntoDDS({{inputImageName.toStdString(),
outputTextureName.toStdString()}},
GetCurrentCompressionType(),
GetCurrentTextureOrientation());



AddProcessedTextures({{inputImageName.toStdString(),
outputTextureName.toStdString()}});
        }
    }
}

void MainWindow::on_convertFolderButton_clicked()
{
    QDir inputFolder =
QFileDialog::getExistingDirectory(this, tr("Open
Folder"), "");

    if(!inputFolder.entryList().isEmpty())
    {
        QString outputFolder =
QFileDialog::getExistingDirectory(this, tr("Save
Folder"), "");
```

```
        if(!outputFolder.isEmpty() &&
!outputFolder.isNull())
        {
            QStringList images =
inputFolder.entryList(QStringList() << "*.jpg" << "*.png"
<< "*.jpg" << "*.jpeg" << "*.bmp" << "*.tif" << "*.tiff",
QDir::Files);

            std::vector<std::pair<std::string,
std::string>> imagesPath;

            for (const auto& image : images)
            {
                QStringList pieces = image.split(".");
                QString fileName = pieces.value(0);
                QString outputFileName = outputFolder +
"/" + fileName + ".dds";


imagesPath.push_back({inputFolder.absoluteFilePath(image)
.toStdString(), outputFileName.toStdString()});
            }

            ConvertImagesIntoDDS(imagesPath,
GetCurrentCompressionType(),
GetCurrentTextureOrientation());

            AddProcessedTextures(imagesPath);
        }
    }
}
```

```
void
MainWindow::on_processedTexturesTable_cellDoubleClicked(i
nt row, int column)
{
    if (column == 0) // only handle if double clicked on
asset path column
    {
        QTableWidgetItem* pItem = ui-
>processedTexturesTable->item(row, column);
        DisplayPreviewerWidget(pItem->text());
    }
}

void MainWindow::on_openTextureButton_clicked()
{
    QString inputImageName =
QFileDialog::getOpenFileName(this, tr("Open Texture"),
"", tr("Texture (*.dds)"));

    if(!inputImageName.isEmpty() &&
!inputImageName.isNull())
    {
        DisplayPreviewerWidget(inputImageName);
    }
}
```

OpenGLWidget.h

```
#pragma once

#include <DDSConversion/DDSDefines.h>
```

```
#include <QtWidgets>
#include <QOpenGLFunctions_4_0_Core>

#include <QDebug>

struct Vertex
{
   Vertex(qreal x, qreal y, qreal z, qreal s, qreal t):
     coord(x, y, z), texCoord(s, t)
   {

   }

   QVector3D coord;
   QVector2D texCoord;
};

class OpenGLWidget
        : public QOpenGLWidget
        , public QOpenGLFunctions_4_0_Core
{
public:
    explicit OpenGLWidget(QWidget* pParent):
      QOpenGLWidget(pParent),
      _qGLBufArray(QOpenGLBuffer::VertexBuffer),
      _qGLBufIndex(QOpenGLBuffer::IndexBuffer),
      _pQGLTex(nullptr)
    { }

    virtual ~OpenGLWidget()
```

```
    {
      makeCurrent();
      delete _pQGLTex;
      _qGLBufArray.destroy();
      _qGLBufIndex.destroy();
      doneCurrent();
    }

    OpenGLWidget(const OpenGLWidget&) = delete;
    OpenGLWidget& operator=(const OpenGLWidget&) =
delete;

    void SetTexture(Texture* pTexture);
    void SetTextureOrientation(TextureOrientation
textureOrientation);

protected:
    virtual void initializeGL() override;
    virtual void paintGL() override;

private:
    void initShaders();
    void initGeometry();

    void initTexture();

private:

    Texture* m_pTexture { nullptr };
    TextureOrientation m_textureOrientation {
TextureOrientation::DirectX };
```

```
    QOpenGLShaderProgram _qGLSProg;
    QOpenGLBuffer _qGLBufArray;
    QOpenGLBuffer _qGLBufIndex;
    QOpenGLTexture *_pQGLTex;
};
```

OpenGLWidget.cpp

```
#include "OpenGLWidget.h"

#include <DDSConversion/DDSDefines.h>

#include <QtWidgets>
#include <QOpenGLFunctions_4_0_Core>

#include <QDebug>

namespace Shaders
{
    const char *VertexShader =
      "#ifdef GL_ES\n"
      "// Set default precision to medium\n"
      "precision mediump int;\n"
      "precision mediump float;\n"
      "#endif\n"
      "\n"
      "uniform mat4 mvp_matrix;\n"
      "\n"
      "attribute vec4 a_position;\n"
      "attribute vec2 a_texcoord;\n"
```

```
      "\n"
      "varying vec2 v_texcoord;\n"
      "\n"
      "void main()\n"
      "{\n"
      "  // Calculate vertex position in screen space\n"
      "  gl_Position = mvp_matrix * a_position;\n"
      "\n"
      "  // Pass texture coordinate to fragment shader\n"
      "  // Value will be automatically interpolated to
fragments inside polygon faces\n"
      "  v_texcoord = a_texcoord;\n"
      "}\n";

    const char *DirectXFragmentShader =
      "#ifdef GL_ES\n"
      "// Set default precision to medium\n"
      "precision mediump int;\n"
      "precision mediump float;\n"
      "#endif\n"
      "\n"
      "uniform sampler2D texture;\n"
      "\n"
      "varying vec2 v_texcoord;\n"
      "\n"
      "void main()\n"
      "{\n"
      "  // Set fragment color from texture\n"
      "#if 0 // test check tex coords\n"
      "  gl_FragColor = vec4(1, v_texcoord.x,
v_texcoord.y, 1);\n"
```

```
      "#else // (not) 0;\n"
      "  vec4 color = texture2D(texture,
vec2(v_texcoord.x, 1 - v_texcoord.y));\n"
      "  //if (color.a < 0.01) discard;\n"
      "  gl_FragColor = color;\n"
      "#endif // 0\n"
      "}\n";

    const char *OpenGLFragmentShader =
      "#ifdef GL_ES\n"
      "// Set default precision to medium\n"
      "precision mediump int;\n"
      "precision mediump float;\n"
      "#endif\n"
      "\n"
      "uniform sampler2D texture;\n"
      "\n"
      "varying vec2 v_texcoord;\n"
      "\n"
      "void main()\n"
      "{\n"
      "  // Set fragment color from texture\n"
      "#if 0 // test check tex coords\n"
      "  gl_FragColor = vec4(1, v_texcoord.x,
v_texcoord.y, 1);\n"
      "#else // (not) 0;\n"
      "  vec4 color = texture2D(texture,
vec2(v_texcoord.x, v_texcoord.y));\n"
      "  //if (color.a < 0.01) discard;\n"
      "  gl_FragColor = color;\n"
      "#endif // 0\n"
```

```
      "}\n";
}

void OpenGLWidget::SetTexture(Texture* pTexture)
{
    m_pTexture = pTexture;
}

void
OpenGLWidget::SetTextureOrientation(TextureOrientation
textureOrientation)
{
    m_textureOrientation = textureOrientation;
    hide();
    show();
}

void OpenGLWidget::initializeGL()
{
    initializeOpenGLFunctions();

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glClearColor(0.5, 0.5, 0.5, 0);

    initShaders();
    initGeometry();
    initTexture();
}
void OpenGLWidget::paintGL()
```

```
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    _pQGLTex->bind();
    QMatrix4x4 mat; mat.setToIdentity();
    _qGLSProg.setUniformValue("mvp_matrix", mat);
    _qGLSProg.setUniformValue("texture", 0);
    // draw geometry
    _qGLBufArray.bind();
    _qGLBufIndex.bind();
    quintptr offset = 0;
    int coordLocation =
_qGLSProg.attributeLocation("a_position");
    _qGLSProg.enableAttributeArray(coordLocation);
    _qGLSProg.setAttributeBuffer(coordLocation, GL_FLOAT,
offset, 3, sizeof(Vertex));
    offset += sizeof(QVector3D);
    int texCoordLocation =
_qGLSProg.attributeLocation("a_texcoord");
    _qGLSProg.enableAttributeArray(texCoordLocation);
    _qGLSProg.setAttributeBuffer(texCoordLocation,
GL_FLOAT, offset, 2, sizeof(Vertex));
    glDrawElements(GL_TRIANGLE_STRIP, 4,
GL_UNSIGNED_SHORT, 0);
}

void OpenGLWidget::initShaders()
{
    if
(!_qGLSProg.addShaderFromSourceCode(QOpenGLShader::Vertex
, QString::fromLatin1(Shaders::VertexShader)))
    {
```

```
      parentWidget()->hide();
    }

    if (m_textureOrientation ==
TextureOrientation::OpenGL)
    {
      if
(!_qGLSProg.addShaderFromSourceCode(QOpenGLShader::Fragme
nt, QString::fromLatin1(Shaders::OpenGLFragmentShader)))
      {
          parentWidget()->hide();
      }
    }
    else
    {
      if
(!_qGLSProg.addShaderFromSourceCode(QOpenGLShader::Fragme
nt, QString::fromLatin1(Shaders::DirectXFragmentShader)))
      {
          parentWidget()->hide();
      }
    }

    if (!_qGLSProg.link()) parentWidget()->hide();
    if (!_qGLSProg.bind()) parentWidget()->hide();
}

void OpenGLWidget::initGeometry()
{
    Vertex vertices[] = {
        //  x      y      z     s     t
```

```
        { -1.0f, -1.0f, 0.0f, 0.0f, 0.0f },
        { +1.0f, -1.0f, 0.0f, 1.0f, 0.0f },
        { +1.0f, +1.0f, 0.0f, 1.0f, 1.0f },
        { -1.0f, +1.0f, 0.0f, 0.0f, 1.0f }
    };
    enum { nVtcs = sizeof vertices / sizeof *vertices };
    GLushort indices[] = { 3, 0, 2, 1 };
    enum { nIdcs = sizeof indices / sizeof *indices };
    _qGLBufArray.create();
    _qGLBufArray.bind();
    _qGLBufArray.allocate(vertices, nVtcs * sizeof
(Vertex));
    _qGLBufIndex.create();
    _qGLBufIndex.bind();
    _qGLBufIndex.allocate(indices, nIdcs * sizeof
(GLushort));
}

void OpenGLWidget::initTexture()
{
    _pQGLTex = new
QOpenGLTexture(QOpenGLTexture::Target2D);


    int dataSizeMultiplier = 0;
    const DWORD dxt1 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('1' << 24);
    const DWORD dxt3 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('3' << 24);
    const DWORD dxt5 = ('D' << 0) | ('X' << 8) | ('T' <<
16) | ('5' << 24);
```

```
    _pQGLTex->setSize(m_pTexture->m_header.dwWidth,
m_pTexture->m_header.dwHeight);


    switch (m_pTexture->m_header.ddspf.dwFourCC)
    {
        case (dxt1):
        {
            dataSizeMultiplier = 8;
            _pQGLTex-
>setFormat(QOpenGLTexture::RGBA_DXT1);
            _pQGLTex-
>allocateStorage(QOpenGLTexture::RGBA,
QOpenGLTexture::UInt16_RGB5A1);
            break;
        }
        case(dxt3):
        {
            dataSizeMultiplier = 16;
            _pQGLTex-
>setFormat(QOpenGLTexture::RGBA_DXT3);
            _pQGLTex-
>allocateStorage(QOpenGLTexture::RGBA,
QOpenGLTexture::UInt32);
            break;
        }
        case(dxt5):
        {
            dataSizeMultiplier = 16;
            _pQGLTex-
>setFormat(QOpenGLTexture::RGBA_DXT5);
```

```
            _pQGLTex-
>allocateStorage(QOpenGLTexture::RGBA,
QOpenGLTexture::UInt32);
            break;
        }
    }

    int dataSize = (m_pTexture->m_header.dwWidth + (4 -
m_pTexture->m_header.dwWidth % 4) % 4) * (m_pTexture-
>m_header.dwHeight + (4 - m_pTexture->m_header.dwHeight %
4) % 4) / 16 * dataSizeMultiplier;
    _pQGLTex->setCompressedData(dataSize,
reinterpret_cast<const void*>(m_pTexture->m_data));

    _pQGLTex-
>setMinificationFilter(QOpenGLTexture::Nearest);
    _pQGLTex-
>setMagnificationFilter(QOpenGLTexture::Nearest);
    _pQGLTex->setWrapMode(QOpenGLTexture::ClampToEdge);
}
```

ProjectDefines.h

```
#pragma once

#include <assert.h>

#define assertm(exp, msg) assert(((void)msg, exp))
```

main.cpp

```
#include "Widgets/MainWindow.h"

#include <QApplication>
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

ДОДАТОК В

Результати перевірки роботи на співпадіння

Имя пользователя:
Лісовиченко Олег Іванович

ID проверки:
1014952391

Дата проверки:
07.05.2023 08:00:20 EEST

Тип проверки:
Doc vs Internet + Library

Дата отчета:
07.05.2023 08:02:30 EEST

ID пользователя:
76913

Название файла: **ІП-11мн_Нестеренко_ПЗ**

Количество страниц: **68** Количество слов: **10930** Количество символов: **80775** Размер файла: **3.00 MB** ID файла: **1014645353**

2.34% Совпадения

Наибольшее совпадение: **1.72%** с источником из Библиотеки (ID файла: **1008193873**)

1.29% Источники из Интернета	16	Страница 70
2.26% Источники из Библиотеки	72	Страница 70

0% Цитат

Исключение цитат выключено

Исключение списка библиографических ссылок выключено

0% Исключений

Нет исключенных источников