Постановка проблемы

Некоторая организация использует для решения своих задач старые, но проверенные временем компьютеры, замена которых экономически необоснованна и влечет за собой риски сбоев в производстве. Данные машины имеют ограниченные графические возможности, и способны отображать на мониторе лишь несколько цветов из фиксированной палитры. Необходимо написать программу, которая позволила бы конвертировать изображения из полноцветного формата в формат с заданным набором цветов и, таким образом, дала возможность просматривать их на устаревшем оборудовании. Преобразование будет выполняться на современном сервере с большим количеством ядер, поэтому программа должна поддерживать многопоточное исполнение.

Выбор способа решения

Формализуем задачу. На вход программы подается некое полноцветное изображение A, которое представляет собой матрицу, компонентами которой являются цвета пикселей. Цвет каждого пикселя a_{ij} описывается вектором, состоящим из трех чисел $a_{i,j} = [red_{i,j}, green_{i,j}, blue_{i,j}]$. Также имеется палитра $P = \{p_1, \dots p_n\} = \{[red_k, green_k, blue_k]\}, k = 1 \dots n$ из нескольких фиксированных цветов, которые могут использоваться в изображении-результате B. Таким образом, для каждого пикселя a_{ij} необходимо подобрать оптимальный цвет из палитры P и записать этот цвет в соответствующий пиксель изображения B. Этот процесс называется квантованием.

Нужно выбрать способ поиска в палитре P оптимального цвета, который заменил бы собой оригинальный цвет пикселя $a_{i,j}$ с цветом. Рассмотрим несколько популярных методов. Во-первых, можно найти для пикселя $a_{i,j}$ ближайший в геометрическом смысле цвет из палитры P, то есть цвет p_i такой, что расстояние

$$distance_{k^*} = \sqrt{\left(red_{k^*} - red_{i,j}\right)^2 + \left(green_{k^*} - green_{i,j}\right)^2 + \left(blue_{k^*} - blue_{i,j}\right)^2}$$

будет минимальным среди всех $k = 1 \dots n$. Однако, такой подход дает плохие результаты. Приведем пример конвертации этим способом полноцветного изображения в черно-белое (бинаризация)

Можно видеть, что в изображении-результате B отсутствуют полутона и значительная часть деталей, которые были в исходном изображении. Таким образом, данный метод нам не подходит.

Рассмотрим теперь другой вариант решения проблемы – дизеринг шумом. Согласно этому подходу, необходимо сгенерировать шум N и подмешать его к изображению A, чтобы внести в квантование небольшие ошибки. Существует множество видов шума, подходящих для дизеринга. Составим таблицу, чтобы сравнить их между собой.

Название	Способ генерации	Вычислительная сложность генерации	Возможна ли многопоточная генерация?	Качество результата
Белый шум	С помощью датчика случайных чисел	Низкая	Да	Низкое
Шум Байера	С помощью матрицы Байера. Это рекурсивный процесс	Средняя	Нет	Среднее
Синий шум	С помощью метода пустот и кластеров и размытия полученной матрицы методом Гаусса	Высокая	Нет	Высокое

Можно видеть, что лишь генерация белого шума поддается разделению между несколькими потоками, однако она не обеспечивает должного качества изображения-результата. Если же требуется получить на выходе достойный результат, придется задействовать лишь одно ядро процессора. Значит, и этот подход в нашем случае неприменим.

Наконец, обратимся к дизерингу с рассеянием ошибки. В этом случае оптимальный цвет для очередного пикселя a_{ij} вычисляется, как и в первом случае – поиском ближайшего цвета из палитры P. Однако, после того, как оптимальный цвет найден, вычисляется ошибка бинаризации

$$mistake_{i,j} = [red_{mistake}, green_{mistake}, blue_{mistake}] = \\ = \left[red_{k^*} - red_{i,j}, green_{k^*} - green_{i,j}, blue_{k^*} - blue_{i,j}\right],$$

которая затем распространяется на соседние пиксели по определенному правилу. Чем на большее количество пикселей будет распространена ошибка, тем более качественное изображение мы получим на выходе. Перечислим самые популярные правила в порядке от худшего к лучшему: дизеринг Флойда-Стейнберга, дизеринг Аткинсона, дизеринг Джарвиса-Джужиса-Нинке. Очевидно, что наиболее интересным для нас является последний метод. В этом случае матрица рассеяния ошибки имеет вид

$$\frac{1}{48}\begin{pmatrix} 0 & 0 & * & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{pmatrix}.$$

То есть, найдя ошибку $mistake_{ij}$, которая возникла при бинаризации пикселя a_{ij} мы рассеем ее между соседними пикселями следующим образом

$$a_{i,j+1} = a_{i,j+1} - \frac{7}{48} mistake_{i,j},$$

$$a_{i,j+2} = a_{i,j+2} - \frac{5}{48} mistake_{i,j},$$

$$a_{i+1,j-2} = a_{i+1,j-2} - \frac{3}{48} mistake_{i,j},$$

и т.д.

Однако, пока что не ясно, как распределить эту задачу между несколькими потоками. Ведь мы не можем обрабатывать произвольные пиксели параллельно из-за зависимостей между ними. Однако, способ распараллеливания существует. Изобразим, какие пиксели будут влиять на a_{ij}

$$\begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & * & & \end{pmatrix}.$$

Далее, пусть, например, входное изображение A состоит из 6 строк и 6 столбцов. Пронумеруем пиксели этого изображения следующим образом

0	1	2	3	4	5
3	4	5	6	7	8
6	7	8	9	10	11
9	10	11	12	13	14
12	13	14	15	16	17
15	16	17	18	19	20

Можно видеть, что пиксель с номером l зависит только от пикселей с номерами, меньшими, чем l. Можно также видеть, что пиксели с одинаковыми значениями l могут обрабатываться параллельно. В этом заключается подход к многопоточному решению задачи. Для каждого $l = 0 \ldots 20$. Мы параллельно вычисляем оптимальный цвет из палитры P и вносим поправку в цвет соседних пикселей.

В конечном итоге, мы выбираем для решения нашей задачи дизеринг Джарвиса-Джужиса-Нинке.

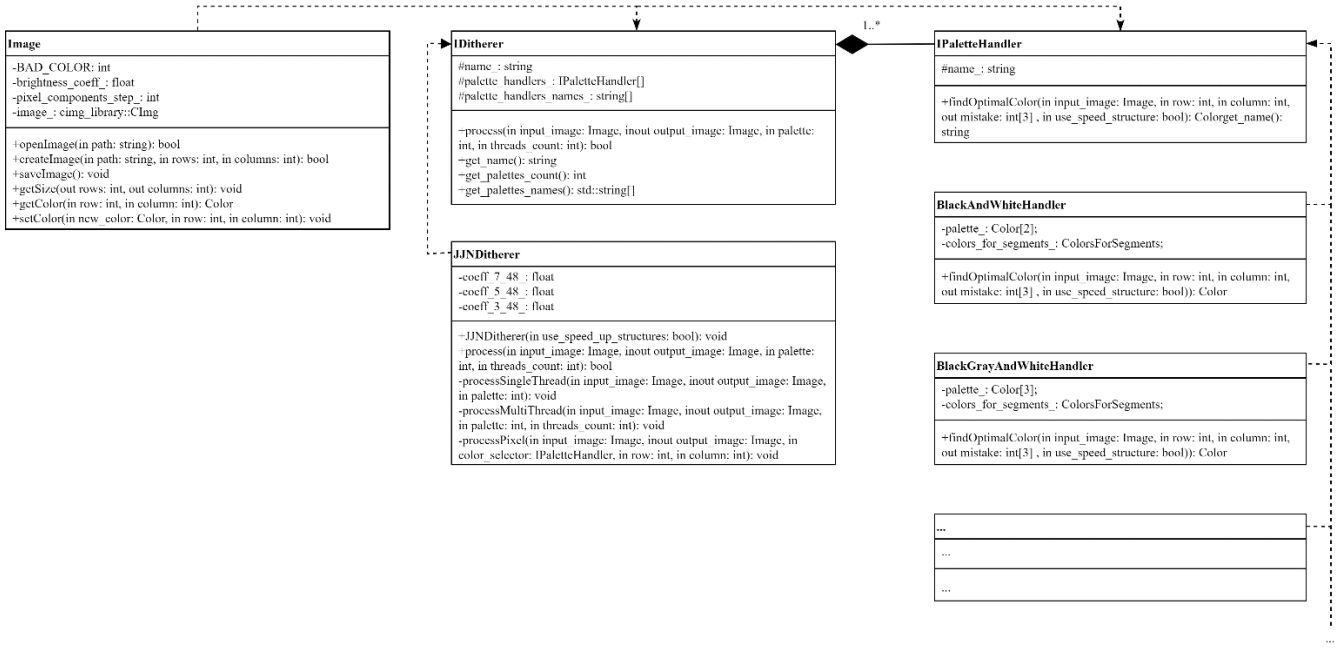
Требования к программе

Требуется написать многопоточную программу, которая позволит преобразовывать изображения в формат с малым количеством цветов с помощью дизеринга Джарвиса-Джужиса-Нинке. Приложение должно работать с файлами в форматах JPEG и PNG и иметь графический интерфейс. Предусмотреть возможность расширения программы в будущем путем добавления новых палитр и новых способов дизеринга.

Архитектура программы

Программа строится вокруг двух групп классов. Классы из первой группы описывают различные палитры и являются реализациями общего интерфейса IPaletteHandler. Эти классы инкапсулируют поиск оптимального цвета в палитре. Объявление класса IPaletteHandler приведено в приложении А. Вторая группа классов описывает различные методы дизеринга и являются реализациями общего интерфейса IDitherer. Эти классы инкапсулируют логику дизеринг. Обьявление класса IDitherer тоже приведено в приложении А. Кроме того, каждая реализация класса IDitherer хранит список палитр, которые поддерживает конкретный метод дизеринга. Кроме того, в проекте присутствует класс Image, инкапсулирующий работу с изображениями.

Приведем UML диаграмму проекта



Таким образом, расширение возможностей программы возможно путем добавления реализации интерфейса IDitherer, если нужно добавить новый способ дизеринга, либо новой реализации интерфейса IPaletteHandler, если нужно добавить новую палитру.

Использование сторонних библиотек

Перед началом разработки было принято решение использовать сторонние библиотеки для работы с графическими файлами и создания пользовательского интерфейса.

В качестве инструмента для работы с изображениями рассматривался фреймворк Qt, однако довольно быстро было решено отказаться от этого варианта в силу избыточности возможностей этой платформы для нашей задачи. После изучения ряда других библиотек, например LibGD, было решено остановиться на библиотеке CImg в силу ее простоты, минимализма и хорошей документации.

Qt рассматривался и для построения графического интерфейса, но был признан неподходящим по все тем же причинам. Взамен него была использована связка библиотек GLFW и ImGui. Первая упрощает работу с графическим стеком OpenGL, а вторая позволяет создавать базовый интерфейс для пользователя.

Каждая из выбранных библиотек имеет свои зависимости. Приведем итоговый список зависимостей проекта.

А) CImg,

Б) Glad,

В) GLFW,

Г) ImGui,

Д) libjpeg,

Е) libpng,

Ж) LibTIFF,

И) ZLib.

Отметим, что многопоточные вычисления основаны на стандарте OpenMP.

В результате, мы получили быстрое и легкое кроссплатформенное приложения. Исполнительный файл программы вместе с файлами библиотек занимает всего 2.2 Мегабайта на диске. Кроме того, будучи запущенной, программа занимает лишь 21 Мегабайт ОЗУ.

Обеспечение стабильности программы

Перечислим параметры, которые принимает программа

А) Способ дизеринга,

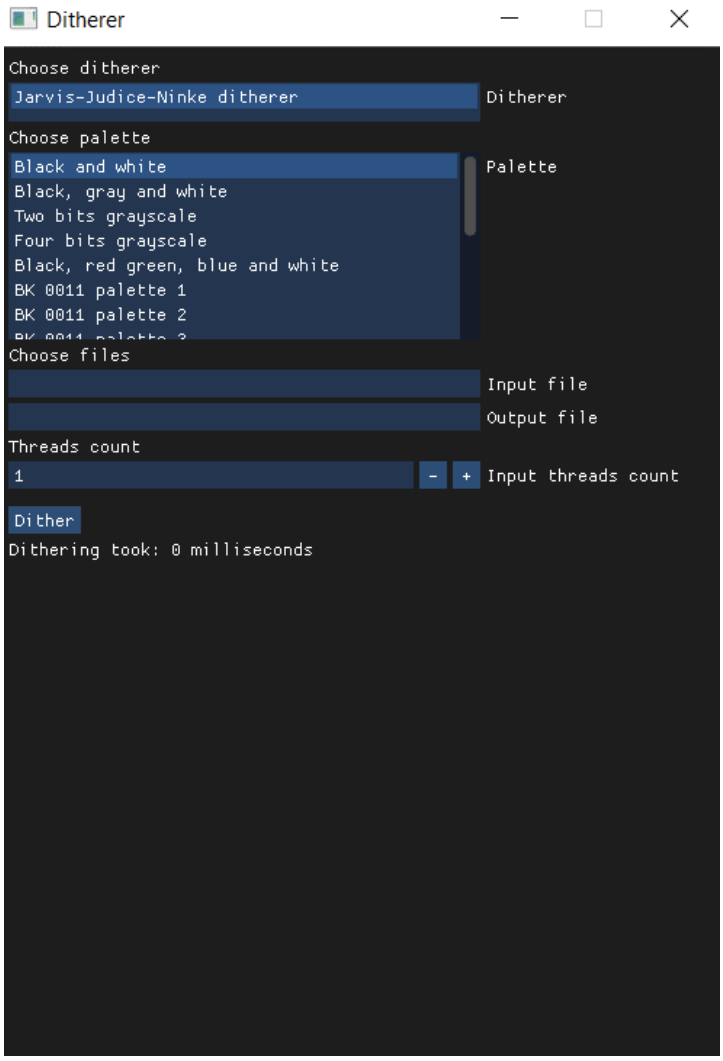
Б) Палитра,

В) Путь исходного файла,

Г) Путь файла с результатом,

Д) Количество потоков, между которыми необходимо распределить задачу.

Покажем также графический интерфейс программы



Можно видеть, что пользователь выбирает способ дизеринга и палитру из списка доступных вариантов. Таким образом, параметры А) и Б) не могут быть некорректными. Далее, предлагается указать пути к файлам. Напомним, что программа поддерживает лишь изображения в форматах

JPEG и PNG. Если же пользователь указал путь файла с другим расширением или несуществующий путь, то в библиотека CImg генерирует исключение при попытке отрыть файл, которое затем обрабатывается в программе. Данный сценарий не влечет за собой завершение программы, а лишь отменяет дизеринг. При этом, на экран выводится сообщение об ошибке. Наконец, программа контролирует значение параметра Д), позволяя вводить лишь числа от 1 до 128.

Таким образом, пользователь ограничен в выборе параметров А), Б) и Д), а некорректные значения параметров В) и Г) не влекут за собой сбоев в работе программы.

Результаты

Приведем примеры дизеринга изображения методом Джарвиса-Джужиса-Нинке с применением различных палитр.

Исходное изображение

черно-белая палитра

палитра с черным, серым и белым цветами

палитра с 16 оттенками серого

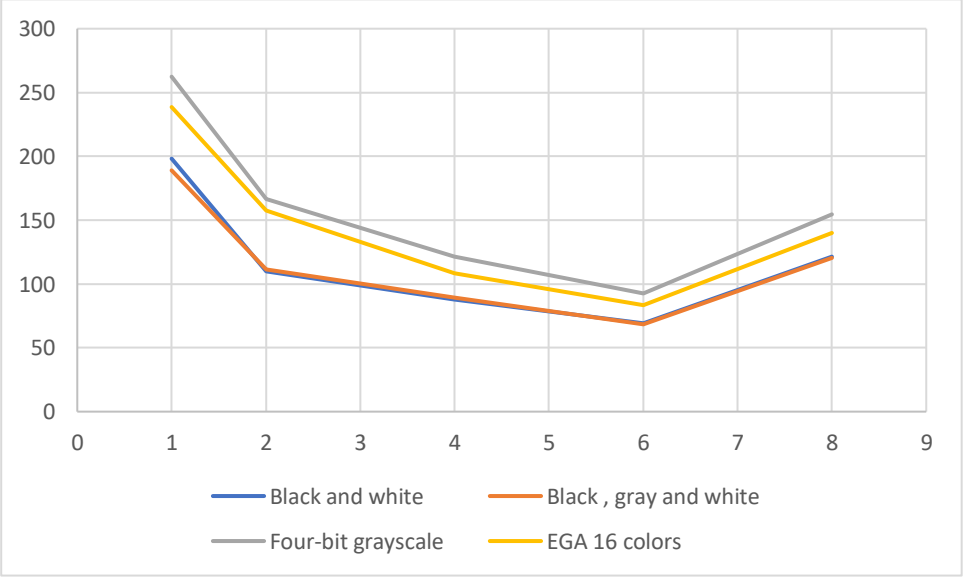
палитра стандарта EGA, 16 цветов

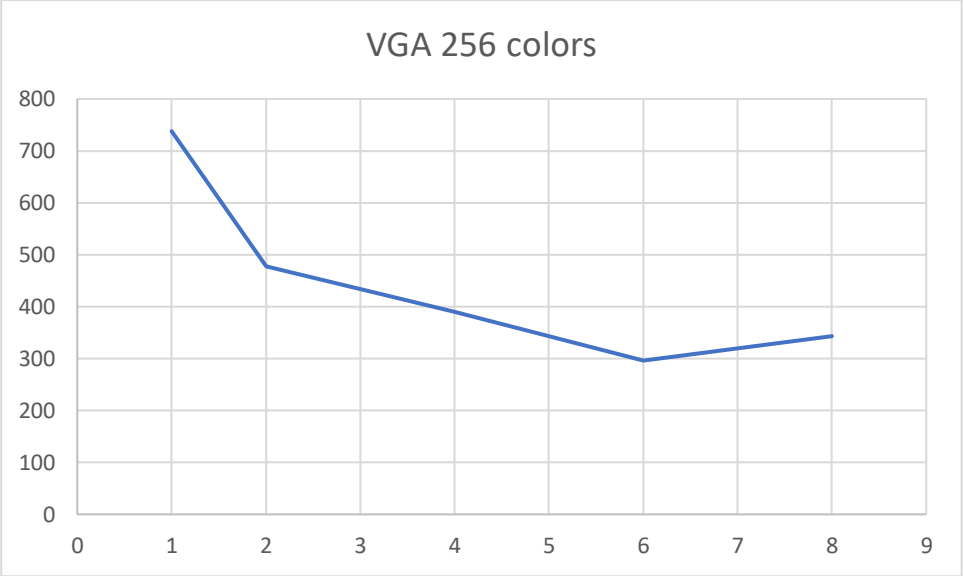
палитра стандарта VGA, 256 цветов

Приведем также замеры производительности приложения для этих же палитр. Тестирование производилось на процессоре Intel Core i7 6700k.

Количество потоков	1	2	4	6	8
Black and white	198,2	109,8	87,8	69,2	121,6
Black , gray and white	189	111,6	89,2	68,4	120,4
Four-bit grayscale	262,4	166,6	121,2	92,6	154,6
EGA 16 colors	238,6	157,6	108,4	83,4	140,2
VGA 256 colors	738	477,8	390,4	296,2	342,8

Можно видеть, что с увеличением числа потоков до 6, программа работает все быстрее, однако дальнейшее увеличение количества потоков влечет за собой ухудшение производительности. Это вызвано все возрастающими накладными расходами на организацию многопоточного исполнения.




VGA 256 colors
800
700
600
500
400
300
200
100
0
0
1
2
3
4
5
6
7
8
9