МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Промежуточный отчет по курсовой работе

по дисциплине: «Основы программирования»

по теме: **«Автоматическое составление схемы для вышивки крестом по заданному изображению»**

Выполнил/а: ст. группы ПВ-231
Чупахина София Александровна
Проверил:
Лукьянов Александр Михайлович

Белгород, 2024

Содержание

Введение:

Для выполнения стандартных действий при работе с изображениями: изменения пропорций, смещения цветовой гаммы, пикселизации или размытия изображения — существуют готовые алгоритмы решения. Проблемы, с которыми приходится сталкиваться в реальной жизни, часто содержат их в себе в качестве подзадач. Однако бывает и так, что подзадача оказывается модифицированным вариантом одной из типовых задач такого плана, и применить стандартные алгоритмы не получится.

Рассмотрим пример из жизни. Среди любителей рукоделия популярна вышивка крестом — в немалой степени из-за простоты этого вида вышивки. Что имеется в виду под «простотой»? Изображение в вышивке крестом (по большей части) составляется из отдельных крестиков разных цветов, расположенных вдоль прямоугольной сетки с постоянным размером и квадратной формой ячейки. То есть этот способ составления изображения схож с растровым типом графики. Это сходство само по себе наводит на мысль, что растровое изображение можно легко перевести в схему для вышивки крестом и так же легко автоматизировать этот процесс составления схем.

Определим **цель работы**: написать программу, которая составляет схему для вышивки крестом по заданному изображению и дополнительным параметрам.

Решается ли поставленная цель какой-либо стандартной функцией? Какого плана обработка требуется растровому изображению, чтобы оно могло стать схемой для вышивки? Начать стоит с уменьшения количества пикселей. Изображение с самым низким (из стандартных) разрешений, 240*320 пикселей, при переводе 1 пикселя в 1 крестик образует схему для весьма внушительного панно — стоит ли говорить о разрешениях, где общее количество точек измеряется в мегапикселях?.. Стандартная процедура пикселизации вполне себе решает эту задачу, существенно уменьшая разрешение — объединяя соседние пиксели в квадратные области одинакового цвета вдоль сетки, укрупненной относительно изначальной.

Но после этого шага мы столкнемся с потребностью сократить количество цветов. Оставлять изображение в прежней палитре (а по умолчанию в форматах .png и .jpg доступно более 16000000 цветов), бессмысленно, потому что пользователь никогда не подберет нитки для воспроизведения этих цветов с такой точностью. Все это многообразие должно быть сведено к нескольким десяткам цветов. А пикселизация с фиксированным количеством цветов — уже не такая тривиальная задача.

Учитывая сказанное выше, подытожим: программа должна принимать на вход графический растровый файл и параметры схемы (требуемые разрешение схемы в крестиках и количество цветов), а возвращать растровое изображение, разбитое на пиксели заданного количества цветов в соответствии с указанными размерами; плюс измененное в целях читаемости схемы (что такое читаемость схемы, рассмотрим позднее).

Задача, поставленная нами, весьма конкретна, и можно составить примерное словесное описание алгоритма, создающего схему:

- Принимает на вход графический растровый файл и введённые пользователем данные:
  - Требуемая длина схемы в крестиках (вводится только длина или только ширина, а оставшийся параметр рассчитывается из пропорций самого изображения, либо вво-

дятся оба параметра, и изображение обрезается, чтобы его соотношение длины к ширине было равно соответствию введенных длины и ширины);

  - Количество цветов, из которых вышивка будет состоять;

- Выделяет в данном изображении n цветов, условно говоря, встречающихся наиболее часто. Всегда ли цветами, которые стоит выделить, будут именно самые частые цвета, рассмотрим ниже;

- Разбивает изображение на квадраты-пиксели в соответствии с заданной длиной/шириной и присваивает каждому квадрату один из выбранных шагом ранее основных цветов, в зависимости от цветов оригинального изображения, используемых в этом квадрате, и их близости к выбранным;

- Сохраняет как отдельные растровые файлы один из вариантов либо два варианта схемы:

  - Изображение, разбитое на различимые квадраты-пиксели (разделенные относительно тонкой сеткой) тех цветов, которые наиболее часто встречаются в исходном изображении и должны будут использоваться в вышивке;

  - Изображение, разбитое на различимые квадраты-пиксели, цвета которых могут быть изменены с целью "читаемости"(если в схеме используются два близких оттенка одного цвета, на втором варианте схемы один из них может быть смещен в какой-то другой цвет, чтобы пользователь в дальнейшем их не путал) и с наложенным поверх каждого квадрата значком, отдельным для каждого цвета (точка, меньший квадрат, треугольник, пара полос и т. д.) с той же целью;

- Сохраняет в отдельный файл или выводит на экран нужную для использования схемы информацию: соответствие реальных цветов с их обозначениями на второй схеме, названия/номера используемых цветов в одной или нескольких существующих палитрах ниток.

Опираясь на этот план, можно выделить наиболее объемные и проблематичные подзадачи, хотя не факт, что такое деление сохранится при переходе к практической части.

✓Проблема выбора основных цветов.

Может показаться, что достаточно отсортировать цвета по количеству пикселей, приходящихся на них в изображении, и взять несколько первых позиций. Но всегда ли приведет ли к желаемому результату такой подход? Допустим, если изображение представляет собой относительно небольшие пятна разных групп цветов на однотонном фоне, то может оказаться, что наиболее часто встречающимися цветами будут близкие оттенки цвета фона, и ни один из цветов самих объектов учтен не будет. Существует ли готовое решение для этой проблемы? Возможно, стоит ввести какое-то минимальное расстояние между выбираемыми цветами. Возможно, стоит анализировать не только количество пикселей определенного цвета, но и то, является ли это количество максимальным в каком-то диапазоне близких к рассматриваемому цветов, и искать таким образом кластеры цветов. *Последний подход, если окажется верным, может оказаться базой для введения более сложных функций: например, поиска кластеров, областей цветов уже в самом изображении, а не в распределении цветов по количеству пикселей, и настройка пользователем «цельности» этих кластеров в формируемой схеме — либо же настройка чувствительности в обнаружении «кластеров по частоте» для получения схем с разными палитрами.*

✓Проблема работы с большими массивами.

В ряде библиотек для работы с растровыми графическими файлами они преобразуются в массив, где каждый пиксель — числовое значение, кодирующее цвет этого пикселя. Тогда для фотографии разрешением, к примеру, 960*1280 количество элементов в таком массиве будет превышать миллион. Не станет ли такое количество элементов препятствием для быстрого выполнения программы, особенно при использовании более сложных схем выбора основных цветов? Если да, то существуют ли готовые алгоритмы, ускоряющие работу над массивами, представляющими изображение, в целом или выполнение функций, которые мы в общих чертах описывали выше?

✓Проблема создания понятного интерфейса

Напрямую эта подзадача не относится к основной; условно можно назвать ее проблемой «фронтенда», в то время как рассматриваемые выше пункты — проблемы «бэкенда». Программы и библиотеки, создаваемые нами в рамках предмета ОП на первом курсе, были больше похожи на черные ящики: единственным элементом интерфейса была консоль, отображающая стандартные потоки ввода-вывода; более сложные типы данных, чем отдельные числа и строки, приходилось вписывать непосредственно в файл программы, а если программа предусматривала создание и редактирование файлов, оно производилось в автоматическом режиме. Для результата курсовой работы хотелось бы иметь более удобный графический интерфейс: отдельное окно, в котором отображались бы исходный и результирующий файл, панель с настройкой основных параметров схемы с помощью ввода чисел или передвижения ползунков, меню с основными функциями программы (добавление нового файла, преобразование текущего, сохранение результата, возможно, открытие предыдущих файлов и так далее).

Можно ли назвать тему этой работы **актуальной**? С одной стороны, для составления схем вышивки крестом существует весьма большое количество готовых программ. С учетом этого тема может показаться исчерпанной, а решение, написанное в рамках данной курсовой — вторичным. Однако можно попытаться, помимо основного функционала, добавить более сложные настройки (примеры мы приводили, когда говорили о проблеме выбора цветов), посмотрев, таким образом, с новой стороны даже на относительно простую проблему. Ну и нельзя не отметить, что решение этой задачи даёт набор навыков и инструментов для решения других задач программной обработки изображения похожей сложности, открывает путь к использованию этих решений в составе более сложных проектов. Актуальность этих навыков не утратится со временем, что дает нам право назвать в каком-то роде актуальной и выбранную тему.

Основная часть:

Немного о выбранных библиотеках:

Для дальнейшей работы будем использовать следующие библиотеки: Pillow и Numpy. Поговорим подробнее, почему именно на них пал выбор.

Pillow — расширенная и продолжающая обновляться версия библиотеки PIL (Python Imaging Library). Она позволяет представлять изображения с жесткого диска внутри файла Python как экземпляр класса Image, получать информацию о изображении, обращаясь к атрибутам этого класса, и изменять само изображение с помощью определенных для класса методов.

В число этих методов входят и такие, которые реализуют достаточно сложные функции — мы будем обращаться к этому функционалу нечасто, в исследовательских целях полезнее будет попробовать реализовать их самим. Однако, скорее всего, полезны будут функции, связанные с изменением размера изображения, а также действиями вроде добавления текста.

Numpy — куда более универсальный инструмент. Эта библиотека предназначена для работы с многомерными массивами. Простое объявление многомерных массивов как экземпляров класса numpy.array, встроенные методы для заполнения их значениями еще на стадии инициализации, методы для поиска максимального, минимального значения, суммы, среднего как во всем массиве, так и в его срезе либо же по строкам/столбцам, удобное взятие срезов, возможность складывать, вычитать, умножать массивы, а также получать скалярное произведение, функция переформирования многомерного массива (измерения массива меняются, при условии, что не изменяется общее его количество элементов, и как следствие, значения массива после переформирования сохраняются)... Это только часть возможностей, предоставляемых этой библиотекой, и с их помощью можно сосредоточиться на более высоком уровне алгоритма, не усложняя код многочисленными вложенными циклами.

Сам по себе многомерный массив может использоваться для представления самых разных структур: матриц, таблиц, частотных словарей для анализа текстов, звуковых и графических файлов. Как несложно догадаться, нас интересуют файлы графические. Растровое изображение состоит из множества точек-пикселей. Цвет каждой точки, в свою очередь, задается тремя компонентами различной интенсивности: красным, зеленым, синим. Интенсивность каждого компонента может меняться в диапазоне от 0 до 255. Такая цветовая модель, RGB (Red, Green, Blue) используется повсеместно; это аддитивная модель, что делает ее наиболее удобной для систем, работающих с излучаемым светом.

Иначе говоря, первым делом приходит на ум представить растровое изображение как двумерный массив точек, где одно измерение массива — ширина изображения, другое — высота; то есть по сути элементы массива — массивы, хранящие цвета каждого пикселя конкретной строки. Однако учитывая способ кодирования цвета в модели RGB, логично будет представлять изображение как трехмерный массив, где измерения — ширина изображения, его высота и количество компонент цвета (всегда равное трем). Получается, если массив img хранит в себе изображение, то обращение к элементу img[i] вернет массив, хранящий цвета для каждой точки i-ой строки, обращение к элементу img[i][j] вернет массив, хранящий компоненты цвета для пикселя в i-ой строке и j-ом столбце, а обращение к элементу img[i][j][k] вернет интенсивность k-ого компонента цвета пикселя в i-ой строке и j-ом столбце (красного цвета при $k = 0$, зеленого для $k = 1$, синего для $k = 2$).

Подключив к файлу python одновременно Pillow и Numpy, можно создать экземпляр Image на основе растрового файла, хранящегося на жестком диске (нужно только указать прямой или относительный путь к нему), а на основе этого экземпляра создать трехмерный массив, изменить его в соответствии с поставленной целью, потом на его основе создать новый экземпляр Image и сохранить изображение, которое он представляет, на жесткий диск. До стадии работы над трехмерным массивом или после нее можно применить к изображению функции, определенные в библиотеке Pillow — поскольку работа с массивами типа numpy.array накладывает определенные ограничения: к таким массивам неприменимы стандартные методы push, pop, append, и в целом изменение размера массива не всегда интуитивно.

Продемонстрируем, как это происходит, на примере выполнения какой-нибудь простой задачи. Сверившись с планом работы, представленном во введении, заметим, что в ходе составле-

ния схемы нам придется так или иначе заняться пикселизацией изображения. Напишем простую программу, решающую исключительно эту задачу.

Стартовые примеры:

```
from PIL import Image
import numpy as np

#Get a name of image in "images" folder and read an image
filename = input()
img = Image.open("images/" + filename)

#Convert image object into array and open it for writing
arrayImg = np.array(img)
arrayImg.setflags(write=1)

#Get a size of "pixels" on result image
cell_size = int(input())

#Iterate all possible cells
for row in range(0, arrayImg.shape[0], cell_size):
    for col in range(0, arrayImg.shape[1], cell_size):
        #Get color-array for average of all colors in cell
        cur_pixel = arrayImg[row:row+cell_size,
   col:col+cell_size].sum(axis=(0,1))
        cur_pixel = cur_pixel // cell_size**2
        #Paint all pixels of cell in average color
        arrayImg[row:row+cell_size, col:col+cell_size] = cur_pixel

#Convert array into image object and save it in same folder
img2 = Image.fromarray(arrayImg)
img2.save("images/pixel_" + filename)
```

Код выше достаточно прозрачен. Для простоты тестирования расположим в той же папке, где находится файл с данной программной, папку images со стартовым набором изображений: cats.png — фотография 1280 × 960, с двумя объектами и разнородным фоном; baloons.png — фотография 2040 × 1280, с множеством скорее однотонных объектов, но для каждого объекта тон выбирается из широкого диапазона цветов, с однородным фоном; imp.png — созданный в растровом редакторе рисунок 730 × 735, с неширокой палитрой, четкими границами и однородным фоном. Тогда при вводе имени файла будет автоматически открываться изображение в папке images с введенным именем (при условии, что оно, конечно, есть, иначе программа аварийно завершает работу) как img.

Сразу после открытия на базе этого изображения создается трехмерный массив на основе изображения img, arrayImg, и вручную меняется один из его флагов, чтобы разрешить перезапись. После этого с клавиатуры ожидается ввод cell_size — размера крупных «пикселей» (далее, чтоб не путаться, будем называть их ячейками), из которых будет состоять изображе-

ние после выполнения данного алгоритма. Затем в теле цикла перебираются значения row и col, от нуля до значения ширины и высоты изображения, которые мы получаем, обращаясь к значениям измерений массива, с шагом, равным размеру ячейки cell_size. По сути, пиксель с координатами i, j — верхний левый угол соответствующей йчейки. Для каждой новой пары значений i и j создается трехмерный массив, в котором будут храниться компоненты цвета, которым будет в будущем залита вся ячейка. Сразу при инициализации его значения определяются как сумма элементов массива arrayImg в срезе с верхним левым углом i и j и размерами cell_size по обоим измерениям. Методу .sum() передается параметр axis — кортеж (0, 1), что означает, что суммирование проводится только по двум измерениям, ширине и высоте изображения. То есть метод возвращает трехэлементный одномерный массив, элементы которого — суммы компонент Red, Green, Blue всех точек текущей ячейки. Разделив весь массив (то есть каждый его элемент) нацело на количество пикселей в ячейке, то есть на квадрат ее размера, мы и получим среднее значение цвета в ней, и можем приравнять к этому значению срез массива arrayImg, отвечающий за текущую ячейку. Как видим, использование numpy действительно облегчает работу: не пришлось в цикле перебирать все точки ячейчки начала для подсчета среднего цвета, а потом для присвоения этого цвета каждому пикселю. Наконец, массив снова преобразуется в объект-изображение и сохраняется в ту же папку, но с измененным названием.

В следующем примере поработаем с встроенной функцией Pillow — обрезкой изображения. В текущей версии программы есть недостаток: если задать размер ячейки, которому не кратно одно из измерений изображения, то приблизившись к его краю, программа попытается взять срез несуществующей области. Вместо изменения области среза по краям (это приведет к тому, что размер ячеек по краям будет отличаться от стандартного) просто обрежем перед обработкой изображение так, чтобы этой проблемы не возникало.

```
1  from PIL import Image
2  import numpy as np
3
4  #Get a name of image in "images" folder and read an image
5  filename = input()
6  img = Image.open("images/" + filename)
7
8  #Get a size of "pixels" on result image
9  cell_size = int(input())
10
11 #Get a size of cropped image and cropped image itself
12 cropped_width = img.width - img.width % cell_size
13 cropped_height = img.height - img.height % cell_size
14 img = img.crop((img.width % cell_size // 2, img.height % cell_size //
       2, img.width % cell_size // 2 + cropped_width, img.height %
       cell_size // 2 + cropped_height))
15
16 #Convert image object into array and open it for writing
17 arrayImg = np.array(img)
18 arrayImg.setflags(write=1)
19
20 #Iterate all possible cells
21 for row in range(0, arrayImg.shape[0], cell_size):
```

```
    for col in range(0, arrayImg.shape[1], cell_size):
        #Get color-array for average of all colors in cell
        cur_pixel = arrayImg[row:row+cell_size,
    col:col+cell_size].sum(axis=(0,1))
        cur_pixel = cur_pixel // cell_size**2
        #Paint all pixels of cell in average color
        arrayImg[row:row+cell_size, col:col+cell_size] = cur_pixel

img2 = Image.fromarray(arrayImg)
img2.save("images/pixel_" + filename)
```

Стандартная программа дополняется тремя строками: вычислением ширины и высоты обрезанного изображения (ис прежних параметров вычитается их остаток от деления на размер ячейки), а потом используется метод .crop() для обрезки. В качестве аргументов передаются четыре числа — координаты верхнего левого и правого нижнего угла. Результат обрезки перезаписывается в прежний объект-изображение, и после над ним выполняются все те же операции.

Результаты пикселизации можно увидеть в секции «Приложения (Фотографии)». Говоря о времени выполнения этого алгоритма — даже для файла с наибольшим разрешением, baloons.png, время обработки составляет не больше секунды. Пока что не будем рассматривать вопросы оптимизации.

Вообще в целом уже сейчас мы можем выполнять по отдельности простые шаги плана во введении. Например, так будет выглядеть программа, разбивающая изображение на клетки серой сеткой, при этом не перекрывая ей части изображения, а смещая каждую клетку вправо от «прута» сетки (соответственно, размер изображения увеличится, и нам придется создать новый массив для работы с ним).

```
from PIL import Image
import numpy as np

#Get a name of image in "images" folder and read an image
filename = input()
img = Image.open("images/" + filename)

#Get a size of "pixels" on result image
cell_size = int(input())
grid_size = max(cell_size//10, 1)

#Get a size of cropped image and cropped image itself
cells_in_width = img.width // cell_size
cells_in_height = img.height // cell_size

cropped_width = img.width - img.width % cell_size
cropped_height = img.height - img.height % cell_size

img = img.crop((img.width % cell_size // 2, img.height % cell_size //
    2, img.width % cell_size // 2 + cropped_width, img.height %
    cell_size // 2 + cropped_height))
```

```
20
21 # convert image object into array
22 arrayImg = np.array(img)
23 arrayImg.setflags(write=1)
24
25 griddedImage = np.full((cropped_height +
     (cells_in_height-1)*grid_size, cropped_width +
     (cells_in_width-1)*grid_size, 3), 128)
26
27 for i in range(0, arrayImg.shape[0], cell_size):
28     for j in range(0, arrayImg.shape[1], cell_size):
29         cur_block = arrayImg[i:i+cell_size, j:j+cell_size]
30
     griddedImage[i+(i//cell_size)*grid_size:i+(i//cell_size)*grid_size+cell_s
     j+(j//cell_size)*grid_size:j+(j//cell_size)*grid_size+cell_size] =
     cur_block
31
32 img2 = Image.fromarray(griddedImage.astype(np.uint8))
33 img2.save("images/gridded_" + filename)
```

Результат также можно увидеть в секции «Приложения (Фотографии)»

В будущем мы перепишем данные программы, выделив в них функции, и совместим эти функции в нужном порядке. Пока попробуем перейти к более сложным задачам.

Попытка выделения основных цветов:

Неудачный способ

Еще во введении к этой работе мы говорили, что скорее всего, выбор ограниченной палитры как множества просто наиболее часто встречающихся цветов, скорее всего, приведет к нечетким результатам. Тем не менее, нельзя утверждать это, не проверив на практике.

```
1  from PIL import Image
2  import numpy as np
3
4  #Get a name of image in "images" folder and read an image
5  filename = input()
6  img = Image.open("images/" + filename)
7
8  #Convert image object into array and open it for writing
9  arrayImg = np.array(img)
10 arrayImg.setflags(write=1)
11
12 #Create an array for number of occurrences for every color
13 pixel_counting = np.zeros((256, 256, 256))
14
15 #Iterate all possible pixels
16 for row in range(0, arrayImg.shape[0]):
```

```
    for col in range(0, arrayImg.shape[1]):
        pixel_counting[arrayImg[row, col, 0], arrayImg[row, col, 1], arrayImg[row, col, 2]] += 1

#Get amount of key colors on result image and create an array for key colors and their frequences
key_colors_amount = int(input())
key_colors_frequency = np.zeros(key_colors_amount)
key_colors = np.zeros((key_colors_amount, 3))

#Iterate all posiible colors
for r in range(0, 256):
    for g in range(0, 256):
        for b in range(0, 256):
            #Checr can current color be one of more frequent
            if (pixel_counting[r, g, b] > key_colors_frequency[key_colors_amount-1]):
                #Search a place in array for this color
                for cur_c in range (0, key_colors_amount):
                    if pixel_counting[r, g, b] > key_colors_frequency[cur_c]:
                        #Add this color, shift others and come to next color

                        key_colors_frequency[cur_c+1:key_colors_amount] = key_colors_frequency[cur_c:key_colors_amount-1]
                        key_colors_frequency[cur_c] = pixel_counting[r, g, b]
                        key_colors[cur_c+1:key_colors_amount] = key_colors[cur_c:key_colors_amount-1]
                        key_colors[cur_c] = [r, g, b]
                        break

#return distance between two colors
def getColorDistance(c1, c2):
    return ((c1[0] - c2[0])**2 + (c1[1] - c2[1])**2 + (c1[2] - c2[2])**2)**0.5

#Iterate all possible pixels
for row in range(0, arrayImg.shape[0]):
    for col in range(0, arrayImg.shape[1]):
        #Find the closest color from key colors
        best_color_ind = 0
        cur_color_distance = getColorDistance(key_colors[0], arrayImg[row, col])
        for cur_c in range (1, key_colors_amount):
```

```
52              if getColorDistance(key_colors[cur_c], arrayImg[row, col])
    < cur_color_distance:
53                  cur_color_distance =
    getColorDistance(key_colors[cur_c], arrayImg[row, col])
54                  best_color_ind = cur_c
55          #Change cur pixel color on closest one
56          arrayImg[row, col] = key_colors[best_color_ind]
57
58 #Convert array into image object and save it in same folder
59 img2 = Image.fromarray(arrayImg)
60 img2.save("images/reduced_" + filename)
```

Пока будем искать основную палитру для неизменненого изображения, а не для его пикселизированного варианта. Поэтому теперь мы перебираем по отдельности каждый пиксель, значения row и col возрастают с единичным шагом. Перед началом перебора мы создали отдельный трехмерный массив размером $256 \times 256 \times 256$, в котором будут храниться количества вхождений в изображение того или иного цвета. То есть, если элемент массива pixel_count[i, j, k] равен m, то значит, в цвет с компонентами Red интенсивностью i, Green интенсивностью j и Blue интенсивностью k в изображении окрашено m пикселей. В ходе перебора мы получаем значения компонент цвета текущего пикселя и используем их как адрес, чтобы увеличить нужный элемент массива на 1.

Считывая с клавиатуры желаемое количество цветов в палитре, мы создаем 2 массива: один для хранения данных о частоте вхождения выбранных цветов, второй — для хранения самих цветов. Перебирая все возможные элементы массива pixel_count, мы отбираем те цвета, частота вхождения которых больше частоты вхождения последнего цвета в этих массивах, и ищем для них подходящее место, смещая на позицию в право все цвета с меньшим кол-вом вхождений.

После получения палитры необходимо снова пройтись по изображению и перекрасить каждый пиксель в цвет из палитры, наиболее близкий к нему. Как определить степень этой близости? Если множество цветов представимо как трехмерный массив, можно так же легко представить его как трехмерное пространство, куб, сложенный из равномерно распределенных точек разных цветов, с ребром длины 256. В зависимости от координаты точки в кубе, она имеет то или иное значение компонет Red, Green, Blue, и найти расстояние l между цветами — точками $p1$ и $p2$ ними можно по теореме Пифагора для параллелепипеда: $l^2 = (p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2$; $l = \sqrt{(p1.x - p2.x)^2 + (p1.y - p2.y)^2 + (p1.z - p2.z)^2}$. осталось лишь перебрать все пиксели изображения и найти максимально близкий цвет из падитры для каждого из них, заменив его на этот цвет.

Однако, если мы посмотрим на результаты работы этого алгоритма, то будем неприятно удивлены. Как мы можем заметить, основные цвета были выделены неправильно не только в случае «небольшие разнородные объекты — однотонный фон» (как в файле baloons.png), но и в остальных случаях. Получилось так, что доминирующими цветами являются едва различимые оттенки либо фона, либо одного из объектов (черного кота на изображении cats.png).

Примечание: этот алгоритм также выполняется весьма долго: от 26 секунд для файла imp.png до 1 минуты 52 секунд для файла baloons.png. То есть время обработки зависит в основном от размера файла, и тормозит программу необходимость перебирать для каждого

пискеля все цвета сформированной палитры. Эта проблема должна нивелироваться, если искать ближайший цвет не для пикселя, а для ячейки.

Приложения (Фотографии):

Рис. 1: Изображение по умолчанию cats.png

Рис. 2: Изображение по умолчанию baloons.png

Рис. 3: Изображение по умолчанию imp.png

Рис. 4: Пикзелизированное изображение cats.png с размером ячейки 10

Рис. 5: Пикзелизированное изображение baloons.png с размером ячейки 15 (размеры изображения не кратны ему)

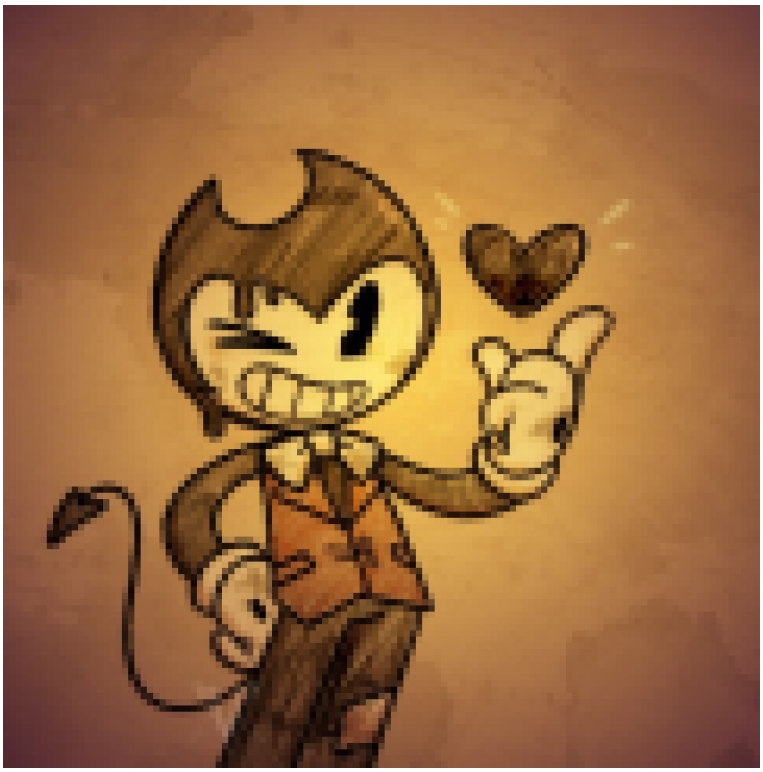

Рис. 6: Пикзелизированное изображение imp.png с размером ячейки 6 (размеры изображения не кратны ему)

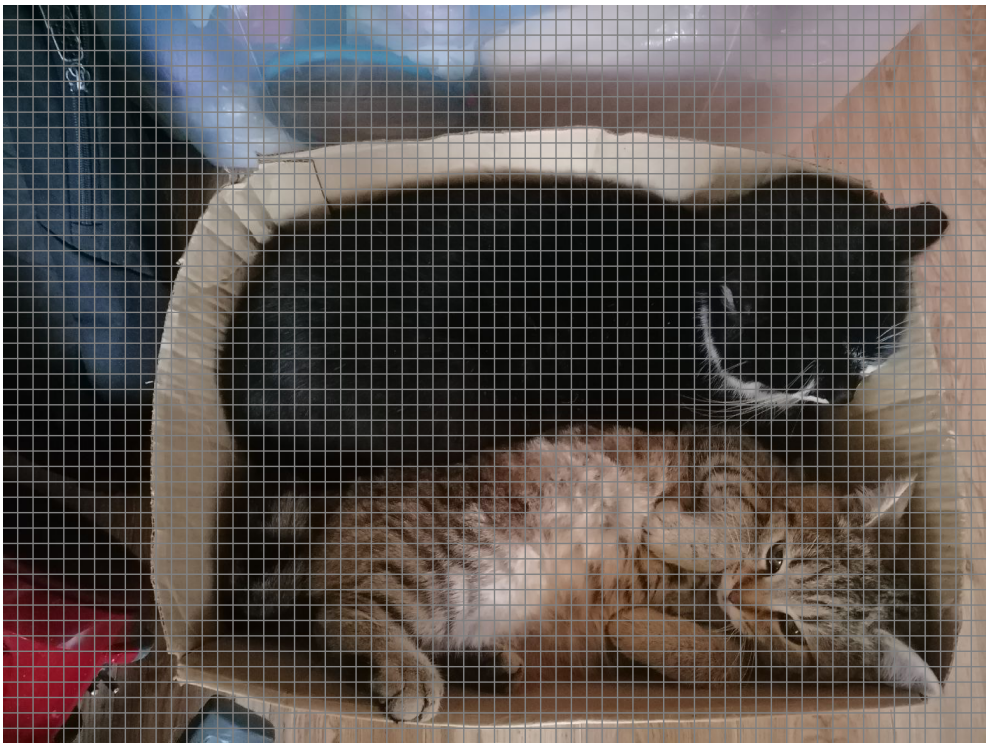

Рис. 7: Покрытое сеткой изображение cats.png с размером ячейки 20 (размер прутьев сетки = $20/2 = 2$)

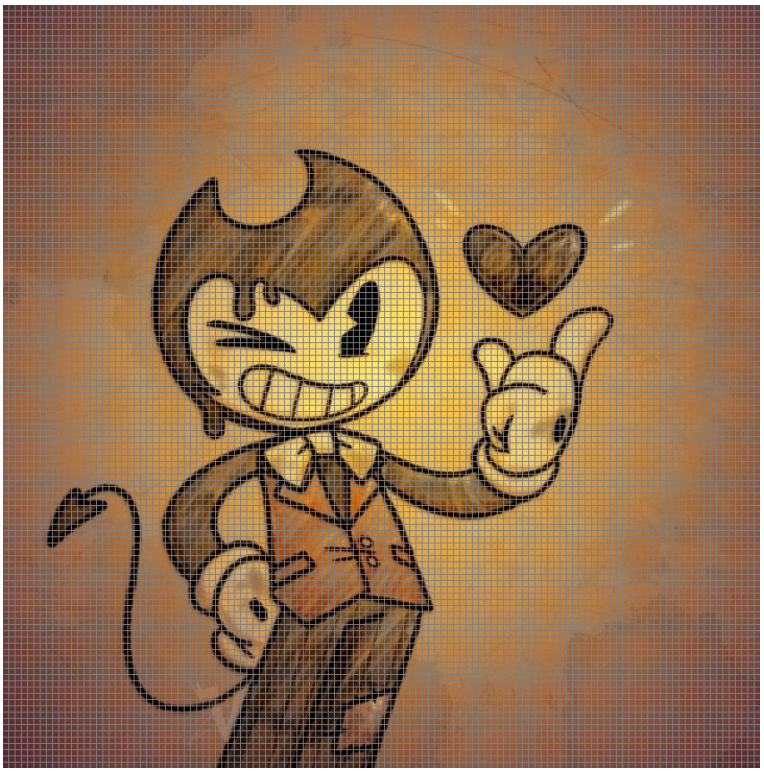

Рис. 8: Покрытое сеткой изображение imp.png с размером ячейки 6 (размер прутьев сетки 1)

Рис. 9: Изображение cats.png с сокращенной до 20 цветов палитрой

Рис. 10: Изображение baloons.png с сокращенной до 20 цветов палитрой

Рис. 11: Изображение imp.png с сокращенной до 20 цветов палитрой