Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Кафедра ПМиК

Расчетно-графическое задание по дисциплине

«Представление графической информации»

Вариант 7

Выполнил:   
студент 4 курса ИВТ,

гр. ИП-013

Копытина Т.А.

Проверил:

Доцент кафедры ПМиК

Перцев И. В.

Новосибирск, 2024 г.

**Оглавление**

[Задание 3](#_Toc160383872)

[Описание алгоритма 4](#_Toc160383873)

[Скриншоты работы программы 6](#_Toc160383874)

[Листинг 7](#_Toc160383875)

# **Задание**

Написать программу-конвертор количества цветов в изображении. Предлагаемый алгоритм. Для уменьшения количества цветов выбираются наиболее часто встречаемые цвета в исходном изображении. Причем эти цвета не должны быть слишком похожими друг на друга. Для сравнения цветов вычисляются разности между RGB составляющими.



После формирования новой палитры цвета заменяются на наиболее похожие из записанных в палитру.

Вариант 7:

Преобразовать 256-цветный PCX файл в 16-цветный BMP файл.

# **Описание алгоритма**

Конструктор экземпляра класса PcxFile принимает количество цветов выходного файла и имя оригинального файла, после чего извлекает информацию из файла – ширину, высоту и цветовую палитру. Затем происходит декодирование изображения формата PCX, и пиксели изображения в формате RGB заносятся в матрицу размером (ширина \* высота \* 3).

Метод Convert выполняет преобразование, следующие шаги выполняются последовательно:

* Вычисление новой палитры цветов путем вызова метода GenerateNewPalette, в котором определяется количество использований каждого цвета в изображении, предварительно каждый цвет сглаживается до 4 младших бит. Затем создается новая палитра, в которую включаются наиболее часто используемые цвета, не слишком похожие на те, которые уже включены в новую палитру. Различие цветов определяется в методе CountDelta.
* Создается массив байтов растра на основе матрицы RGB, где цвета заменяются на наиболее похожие из новой палитры. Новый цвет определяется методом GetSimilarColor, который находит наиболее подходящий цвет путем сравнения дельт из метода CountDelta.
* Сырой массив байтов растра преобразуется в массив, учитывая, что в BMP изображении с глубиной цвета = 4, в одном байте хранится 2 пикселя. Поочередно выбираются байты, для первого происходит сдвиг влево на 4, а затем второй байт дописывается в младшие биты первого, таким образом, получается 2 пикселя в одном байте, каждый из которых занимает по 4 бита.
* Новое изображение записывается в файл и отображается на экране в методе WriteOutputFile, где заполняется заголовок файла BMP, RGB палитра переводится в байты, затем в файл записываются все это и массив байтов растра. После этого на экран выводится оригинальное изображение и рядом с ним новое.

# **Скриншоты работы программы**



Рис. 1-2 Отрисовка 256-цветного PCX изображения и 16-цветного BMP изображения тигра

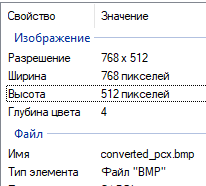


Рис. 3 Информация о сконвертированном BMP файле

# **Листинг**

import random

import math

import numpy as np

import matplotlib.pyplot as plt

BMP\_HEADER\_BSIZE = 14

BMP\_INFO\_HEADER\_BSIZE = 40

class BmpFile:

def \_\_init\_\_(self, name):

self.name = name

self.fileObj = None

self.header = None

self.infoHeader = None

self.palette = None

self.paletteSize = None

self.colorCount = None

self.bpp = None

self.padding = None

self.type = None

self.size = None

self.reserved = None

self.offset = None

self.infoHeaderSize = None

self.width = None

self.height = None

self.planes = None

self.depthColor = None

self.compression = None

self.compressedSize = None

self.xPixPM = None

self.yPixPM = None

self.usedColors = None

self.importantColors = None

**«…»**

class BmpFileReader:

def \_\_init\_\_(self, fileName):

self.bmpObj = BmpFile(fileName)

def Read(self):

self.bmpObj.fileObj = open(self.bmpObj.name, 'rb')

self.bmpObj.header = self.bmpObj.fileObj.read(BMP\_HEADER\_BSIZE)

# HEADER

self.bmpObj.type = self.bmpObj.header[:2].decode('utf-8')

self.bmpObj.size = int.from\_bytes(self.bmpObj.header[2:6], 'little')

self.bmpObj.reserved = int.from\_bytes(self.bmpObj.header[6:10], 'little')

self.bmpObj.offset = int.from\_bytes(self.bmpObj.header[10:14], 'little')

# HEADER #

self.bmpObj.infoHeader = self.bmpObj.fileObj.read(BMP\_INFO\_HEADER\_BSIZE)

# INFO HEADER

self.bmpObj.infoHeaderSize = int.from\_bytes(self.bmpObj.infoHeader[:4], 'little')

self.bmpObj.width = int.from\_bytes(self.bmpObj.infoHeader[4:8], 'little')

self.bmpObj.height = int.from\_bytes(self.bmpObj.infoHeader[8:12], 'little')

self.bmpObj.planes = int.from\_bytes(self.bmpObj.infoHeader[12:14], 'little')

self.bmpObj.depthColor = int.from\_bytes(self.bmpObj.infoHeader[14:16], 'little')

self.bmpObj.compression = int.from\_bytes(self.bmpObj.infoHeader[16:20], 'little')

self.bmpObj.compressedSize = int.from\_bytes(self.bmpObj.infoHeader[20:24], 'little')

self.bmpObj.xPixPM = int.from\_bytes(self.bmpObj.infoHeader[24:28], 'little')

self.bmpObj.yPixPM = int.from\_bytes(self.bmpObj.infoHeader[28:32], 'little')

self.bmpObj.usedColors = int.from\_bytes(self.bmpObj.infoHeader[32:36], 'little')

self.bmpObj.importantColors = int.from\_bytes(self.bmpObj.infoHeader[36:40], 'little')

# INFO HEADER #

self.bmpObj.colorCount = pow(2, self.bmpObj.depthColor)

self.bmpObj.paletteSize = self.bmpObj.colorCount \* 4

# self.bmpObj.palette = self.bmpObj.fileObj.read(self.bmpObj.paletteSize)

self.bmpObj.bpp = self.bmpObj.depthColor // 8

self.bmpObj.padding = (4 - (self.bmpObj.width \* self.bmpObj.bpp) % 4) % 4

return self.bmpObj.fileObj

def Rewrite(self, newName):

with open(newName, 'wb') as newFileObj:

newFileObj.write(self.bmpObj.header)

newFileObj.write(self.bmpObj.infoHeader)

newFileObj.write(self.bmpObj.palette)

newFileObj.write(self.bmpObj.fileObj.read())

newFileObj.close()

class PcxFile:

def \_\_init\_\_(self, outputColorNum, filename):

self.outputColorNum = outputColorNum

with open(filename, 'rb') as file:

self.header = file.read(128)

self.depth = int.from\_bytes(self.header[3:4], 'little')

self.width = self.header[8] + (self.header[9] << 8) - self.header[4] - (self.header[5] << 8) + 1

self.height = self.header[10] + (self.header[11] << 8) - self.header[6] - (self.header[7] << 8) + 1

file.seek(-768, 2)

self.palette = np.frombuffer(file.read(), dtype=np.uint8).reshape((256, 3))

self.graphImg = np.zeros((self.height, self.width, 3), dtype=np.uint8)

file.seek(128)

x, y = 0, 0

while y < self.height:

x = 0

while x < self.width:

binByte = file.read(1)

byte = int.from\_bytes(binByte, 'little')

if byte < 192:

self.graphImg[y, x] = self.palette[byte]

x += 1

else:

count = byte - 192

binRepeatedByte = file.read(1)

repeatedByte = int.from\_bytes(binRepeatedByte, 'little')

if count == 1 and repeatedByte == 0: continue

for \_ in range(count):

if x >= self.width: break

self.graphImg[y, x] = self.palette[repeatedByte]

x += 1

y += 1

def Convert(self):

self.newPalette = self.GenerateNewPalette(self.graphImg, self.width, self.height)

dictPalette = {self.newPalette[i]: i for i in range(len(self.newPalette))}

imgBytes = []

for y in range(self.height - 1, -1, -1):

for x in range(self.width):

self.graphImg[y, x] = self.GetSimilarColor(self.newPalette, self.graphImg[y, x])

colorInd = dictPalette.get(tuple(self.graphImg[y, x]))

if colorInd == None:

colorInd = 0

imgBytes.append(colorInd)

self.imgZipBytes = []

for i in range(0, len(imgBytes), 2):

colorByte = 0

pixel1 = (colorByte | imgBytes[i]) << 4

pixel2 = pixel1 | imgBytes[i]

#pixel2 = pixel1 | imgBytes[i + 1]

self.imgZipBytes.append(pixel2)

self.WriteOutputFile()

def WriteOutputFile(self):

bmpHeader = bytearray()

headerSize = 54

paletteSize = 4 \* 16

bmpHeader.extend(b'BM')

bmpHeader.extend(int(self.width \* self.height // 2 + paletteSize + headerSize).to\_bytes(4, 'little'))

bmpHeader.extend(b'\x00\x00\x00\x00')

bmpHeader.extend(int(paletteSize + headerSize).to\_bytes(4, 'little'))

bmpHeader.extend(int(40).to\_bytes(4, 'little'))

bmpHeader.extend(int(self.width).to\_bytes(4, 'little'))

bmpHeader.extend(int(self.height).to\_bytes(4, 'little'))

# Planes

bmpHeader.extend(int(1).to\_bytes(2, 'little'))

# bit count

bmpHeader.extend(int(4).to\_bytes(2, 'little'))

# compression

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

# compressed size

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

# pix per m X

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

# pix per m Y

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

# used colors

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

# important colors

bmpHeader.extend(int(0).to\_bytes(4, 'little'))

paletteBytes = []

for color in self.newPalette:

r, g, b = color

paletteBytes.extend([b, g, r, 0])

with open('converted\_pcx.bmp', 'wb') as f:

f.write(bytes(bmpHeader))

f.write(bytes(paletteBytes))

f.write(bytes(self.imgZipBytes))

def Show(self):

plt.imshow(self.graphImg)

plt.axis('off')

plt.show()

def CountDelta(self, left, right):

return sum([(x - y) \*\* 2 for x, y in zip(left, right)])

def GenerateNewPalette(self, pixels, width, height):

colors = {}

for y in range(height):

for x in range(width):

flattenColor = (pixels[y, x][0] >> 4 << 4, pixels[y, x][1] >> 4 << 4, pixels[y, x][2] >> 4 << 4)

colors[flattenColor] = colors[flattenColor] + 1 if flattenColor in colors else 1

colors = list(colors.items())

colors.sort(key=lambda x: x[1], reverse=False)

newPalette = []

newPalette.append(colors.pop()[0])

newColorCount = 1

while newColorCount < self.outputColorNum:

newColor = colors.pop()[0]

for color in newPalette:

if 128\*128\*3 < self.CountDelta(color, newColor):

newPalette.append(newColor)

newColorCount += 1

break

return newPalette

def GetSimilarColor(self, palette, color):

similarColor = (0, 0, 0)

for paletteColor in palette:

if self.CountDelta(similarColor, color) > self.CountDelta(paletteColor, color):

similarColor = paletteColor

return similarColor

**«…»**

def ConvertScript():

#pcxFile = PcxFile(16, "200001.pcx")

pcxFile = PcxFile(16, "CAT256.pcx")

pcxFile.Convert()

pcxFile.Show()

if \_\_name\_\_ == '\_\_main\_\_':

ConvertScript()