МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Отчет по лабораторной работе №3

По дисциплине «Компьютерная геометрия и графика»

Изучение алгоритмов псевдотонирования изображений

Выполнил студент группы №M3101:

Пантелеев Ярослав Кириллович

Преподаватель:

Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ

2020

Цель работы:

Изучить алгоритмы и реализовать программу, применяющий алгоритм дизеринга к изображению в формате PGM (P5) с учетом гамма-коррекции.

Описание работы:

Программа должна поддерживать серые изображения (PNM P5), самостоятельно определяя формат по содержимому, быть написана на языке C/C++ и не использовать внешние библиотеки.

Аргументы программе передаются через командную строку:

program.exe <имя_входного_файла> <имя_выходного_файла> <градиент> <дизеринг> <битность> <гамма>

где:

- <градиент>: 0 - используем входную картинку, 1 - рисуем горизонтальный градиент (0-255) (ширина и высота берутся из <имя_входного_файла>);

- <дизеринг> - алгоритм дизеринга:
  - 0 – Нет дизеринга;
  - 1 – Ordered (8x8);
  - 2 – Random;
  - 3 – Floyd–Steinberg;
  - 4 – Jarvis, Judice, Ninke;
  - 5 - Sierra (Sierra-3);
  - 6 - Atkinson;
  - 7 - Halftone (4x4, orthogonal);

- <битность>: битность результата дизеринга (1..8);

- <гамма>: (optional)положительное вещественное число: гамма-коррекция с введенным значением в качестве гаммы. При его отсутствии используется sRGB.

Теоретическая часть:

Дизеринг

- при обработке цифровых сигналов представляет собой подмешивание в первичный сигнал псевдослучайного шума со специально подобранным спектром. Применяется при обработке цифрового звука, видео и графической информации для уменьшения негативного эффекта от квантования.

В компьютерной графике дизеринг используется для создания иллюзии глубины цвета для изображений с относительно небольшим количеством цветов в палитре. Отсутствующие цвета составляются из имеющихся путем их «перемешивания».

Определение пороговых цветов для битностей

для округления текущего значения цвета до ближайшего, который можно отобразить в задаваемой битности B, из целочисленного значения цвета берутся B старших бит и дублируются сдвигами по B бит в текущее значение цвета.

Ordered dithering

Алгоритм уменьшает количество цветов, применяя карту порогов M (другое обозначение: Bayer matrix) к отображаемым пикселям, в результате чего некоторые пиксели меняют цвет в зависимости от расстояния исходного цвета от доступных записей цветов в уменьшенной палитре. Причем, все элементы матрицы должны быть представлены в диапазоне (0, 1].

Алгоритм смещает для каждого пикселя его значение цвета на соответствующее значение из карты порогов M в соответствии с его местоположением, в результате чего значение пикселя квантуется на другой цвет, если оно превышает пороговое значение.

Для большинства случаев сглаживания достаточно просто добавить пороговое значение к каждому пикселю или эквивалентно сравнить значение этого пикселя с порогом: если значение пикселя меньше, чем число в соответствующей ячейке матрицы, записать в пиксель черный цвет, в противном случае, белый в случае битности 1.

Алгоритм выполняет следующее преобразование для каждого цвета с каждого пикселя:

color'=findNearestPaletteColor(color + resizer*M(x%n,y%n)),

где:

- color - старый цвет пикселя
- M(x%n,y%n) - элемент карты порогов
- findNearestPaletteColor - функция, возвращающая ближайший цвет к подаваемому, который можно отобразить на текущей палитре
- color' - новый цвет пикселя в текущей палитре
- resizer – коэффициент цветности(1/битность)

Halftone

Полутонирование - создание изображения со многими уровнями серого или цвета (т.е. слитный тон) на аппарате с меньшим количеством тонов, обычно чёрно-белый принтер. В принципе, задача в том чтобы уменьшить разрешение, увеличивая видимую глубину тона (так называемое пространственное полутонирование).

Полутона широко используются для печати цветных изображений. Общая идея: изменяя плотность четырех вторичных цветов печати: голубого, пурпурного, желтого и черного (сокращение CMYK), можно воспроизвести любой конкретный оттенок.

В нашей лабораторной полутонирование максимально схоже с Ordered dithering и на него просто применяется иного матрица.

Error diffusion algorithms

Алгоритмы с распространением (рассеиванием) ошибок распределяют остаток квантования по соседним пикселям, которые еще не были обработаны.

В отличие от предыдущих алгоритмов, данные алгоритмы работают с некоторой окрестностью пикселя: то, что алгоритм делает в одном месте, влияет на то, что происходит в других местах. Это означает, что требуется буферизация и усложняет параллельную обработку.

Рассеивание ошибок следует более умному подходу к проблеме. Как вы могли предположить, рассеивание ошибок работает путем «рассеивания» (распространения) ошибки каждого вычисления в соседние пиксели. Если алгоритм находит серый пиксель со значением 96, он также определяет, что 96 ближе к 85, и поэтому делает пиксель темно-серым. Но тогда алгоритм учитывает «ошибку» в его преобразовании. В частности, ошибку в том, что преобразованный пиксель на самом деле был на расстоянии в 11 шагов от темно-серого.

Когда мы перемещаемся к следующему пикселю, алгоритм рассеивания ошибок добавляет ошибку предыдущего пикселя к текущему пикселю. Если следующий пиксель имеет серый цвет 118, вместо того, чтобы сделать его темно-серым, алгоритм добавляет ошибку 11 из предыдущего пикселя. Это приводит к значению 129, которое на самом деле ближе к 170. Таким образом, алгоритм делает этот пиксель светло-серым и снова учитывает ошибку. В этом случае ошибка составляет −41, потому что 129 на 41 меньше, чем 170 — то значение, на которое этот пиксель поменяли.

Это куда лучше, чем окрашивать все пиксели подряд чёрным. Как правило, когда мы заканчиваем обработку строки изображения, мы отбрасываем значение ошибки, которое мы отслеживали, и начинаем заново с ошибкой «0» со следующей строки изображения.

Для простоты расчетов все стандартные формулы дизеринга продвигают ошибку только вперед. Если обойти изображение попиксельно, начиная с верхнего левого угла и двигаясь вправо, учитывать ошибки назад (например, влево и/или вверх) необходимости не будет.

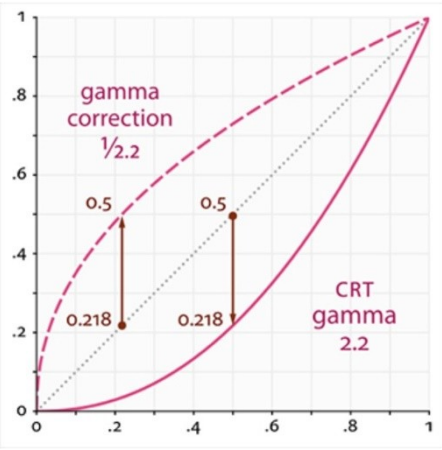
Таким образом, для стандартного поведения цикла (начиная с верхнего левого угла изображения и двигаясь вправо) мы хотим, чтобы движение пикселей шло только вправо и вниз.

Таким образом, алгоритмы с рассеянием ошибки действуют следующим образом:

1. Обход по пикселям совершается слева направо, сверху вниз
2. При анализе текущего пикселя (color):
    1. определяется его новый цвет color' через findNearestPaletteColor
    2. рассчитывается ошибка как color-color'
    3. затем ошибка распространяется согласно используемому паттерну

Гамма-коррекция

Гамма-коррекция (иногда — гамма) — предыскажения яркости чёрно-белого или цветоделённых составляющих цветного изображения при его записи. В качестве передаточной функции при гамма-коррекции чаще всего используется степенная в виде $V_{out} = V_{i}^{\gamma}$. Идея гамма-коррекции заключается в том, чтобы применить инверсию гаммы монитора к окончательному цвету перед выводом на монитор (записью в файл), чтобы картинка представлялась пользователю в корректном виде.



Здесь:

- Gamma correction – записанные в файле данные
- CRT gamma – гамма монитора – искажения им накладываемые
- Пунктирная линия – то, что увидит пользователь

sRGB является стандартом представления цветового спектра с использованием модели RGB. sRGB создан для унификации использования модели RGB в мониторах, принтерах и Интернет-сайтах.

sRGB использует основные цвета, описанные стандартом BT.709, аналогично студийным мониторам и HD-телевидению, а также гамма-коррекцию, аналогично мониторам с электронно-лучевой трубкой. Такая спецификация позволила sRGB в точности отображаться на обычных CRT-мониторах и телевизорах, что стало в своё время основным фактором, повлиявшим на принятие sRGB в качестве стандарта.

Алгоритмы дизеринга

Для Halftone и Ordered – матрицы находятся в коде. Оба алгоритма в нашем случае являются случаями Ordered Dithering'a, описанного выше.

Для отсутствия дизеринга для каждого пикселя просто определяется ближайший цвет(thresholding).

Для случайного дизеринга к каждому пикселю добавляется случайное значение от 0 до 1, аналогично для Ordered dithering умножнное на resizer.

Для всех остальных алгоритмов, являющихся алгоритмами исправления ошибки, применяются матрицы, которые тоже в явном виде описаны в коде

Экспериментальная часть

Язык выполнения работы: C++17, компилятор Microsoft Visual C++.

Этапы работы программы

1) Чтение файла картинки (хедера и информации о цвете каждого пикселя) с обработкой ошибок чтения и применением обратной гамма-коррекции
2) Вызываем метод dithering у объекта изображения, в котором сжимается палитра цветов и изменяются цвета определенных пикселей (есть возможность рассеивать ошибкик)
3) Запись полученного изображения в выходной файл с применением гамма-коррекции

Вывод:

Выполнение данной лабораторной работы позволило изучить алгоритмы псевдотонирования изображений как с упорядоченным распределением ошибок, так и с рассеиванием ошибок. Были реализованы следующие алгоритмы для псевдотонирования изображений:

1. Ordered (8x8);
2. Random;
3. Floyd–Steinberg;
4. Jarvis, Judice, Ninke;
5. Sierra (Sierra-3);
6. Atkinson;
7. Halftone (4x4, orthogonal);

При реализации чтения и записи изображения была изучена гамма-коррекция значений пикселей изображения. Реализована гамма-коррекция для вещественного значения гаммы, а также sRGB гамма-коррекция.

Листинг:

main.cpp

```cpp
#include <iostream>
#include <string>
#include "Image.h"

int main(int argc, char *argv[]) {
    if(argc != 7) { // àðãóìåíòîâ íå 7
        std::cerr << "Invalid command line arguments!" << "\n";
        return 1;
    }

        const std::string fin = std::string(argv[1]);
        const std::string fout = std::string(argv[2]);
        bool gradient;
        int type, bit;
        bool SRGB = false;
        double gamma;

        try {
            gradient = (std::string(argv[3]) == "1");
            type = atoi(argv[4]);
            bit = atoi(argv[5]);
            if(std::string(argv[6]) != "0" && std::string(argv[6]) != "0.0") {
                gamma = std::stold(argv[6]);
            }
            else {
                gamma = 2.4;
                SRGB = true;
            }
        }
        catch (const std::exception& error) {
            std::cerr << error.what() << "\n";
            return 1;
        }

        Image* image;
        try {
            image = new Image(fin, gradient, gamma, SRGB);
        }
        catch (const std::exception& error) {
            std::cerr << error.what() << "\n";
            return 1;
        }

        image->dithering(bit, type);
        try {
            image->write(fout, gamma, SRGB, bit);
        }
        catch (const std::exception& error) {
            std::cerr << error.what() << "\n";
            delete image;
            return 1;
        }

        delete image;
        return 0;
    }
```

Image.h

```
#pragma once
#include <vector>
#include <string>

class Point {
public:
    double x, y;
    Point(double, double);
};

      class Image {
      public:
          Image(const std::string&, bool, double, bool);
          void write(const std::string&, double, bool, int);
          void dithering(int, int);
      private:
          int width;
          int height;
          int color_depth;
          std::vector <std::vector <double>> image;
          std::vector <std::vector <double>> errors;
      };
```

Image.cpp

```
#include "Image.h"
#include <cmath>
#include <ctime>
#include <fstream>

std::vector<std::vector <int>> ordered_dithering = {
   {0, 48, 12, 60, 3, 51, 15, 63},
   {32, 16, 44, 28, 35, 19, 47, 31},
   {8, 56, 4, 52, 11, 59, 7, 55},
          {40, 24, 36, 20, 43, 27, 39, 23},
          {2, 50, 14, 62, 1, 49, 13, 61},
          {34, 18, 46, 30, 33, 17, 45, 29},
          {10, 58, 6, 54, 9, 57, 5, 53},
          {42, 26, 38, 22, 41, 25, 37, 21},
      };
      std::vector<std::vector <int>> halftone = {
         {7, 13, 11, 4},
         {12, 16, 14, 8},
         {10, 15, 6, 2},
         {5, 9, 3, 1},
      };

      // Конструктор
      Point::Point(double x, double y) {
          this->x = x;
          this->y = y;
      }

      // Конструктор
```

```
30.       Image::Image(const std::string& filename, bool gradient, double gamma, bool SRGB)
{
31.           // Задаем seed для rand()
32.           srand(time(nullptr));
33.           // Открываем файл
34.           std::ifstream fin(filename, std::ios::binary);
35.           if (!fin.is_open()) // файл не открылся
36.               throw std::runtime_error("failed to open file");
37.
38.           // Читаем хедер
39.           char ch[2];
40.           fin >> ch[0] >> ch[1];
41.           if (ch[0] != 'P' || ch[1] != '5') // формат не поддерживается
42.               throw std::runtime_error("expected P5 format");
43.           fin >> this->width >> this->height >> this->color_depth;
44.           this->image.assign(this->height, std::vector<double>(this->width));
45.
46.           // Читаем пиксели
47.           char pixel;
48.           fin.read(&pixel, 1);
49.           for (int i = 0; i < this->height; i++) {
50.               for (int j = 0; j < this->width; j++) {
51.                   if (!gradient) {
52.                       fin.read(&pixel, sizeof(unsigned char));
53.
double old = static_cast<double>(static_cast<unsigned char>(pixel)) / this->color_depth;
54.                       if (SRGB) // SRGB гамма-коррекция
55.
old = (old < 0.04045 ? old / 12.92 : pow((old + 0.055) / 1.055, gamma));
56.                       else
57.                           old = pow(old, gamma);
58.                       this->image[i][j] = old;
59.                   }
60.                   else {
61.                       this->image[i][j] = j * (1.0 / (this->width - 1));
62.                   }
63.               }
64.           }
65.           fin.close();
66.       }
67.
68.       // Запись картинки в файл
69.       void Image::write(const std::string& filename, double gamma, bool SRGB, int bit)
{
70.           this->color_depth = (1 << bit) - 1; // 2 ** bit - 1
71.           std::ofstream fout(filename, std::ios::binary);
72.           if (!fout.is_open()) { // файл не открылся
73.               throw std::runtime_error("cannot open output file");
74.           }
75.           fout << "P5\n" << this->width << ' ' << this->height << '\n' << this-
>color_depth << '\n';
76.           for (int i = 0; i < this->height; i++) {
77.               for (int j = 0; j < this->width; j++) {
78.                   double old = this->image[i][j];
79.                   if (SRGB)
80.
old = (old <= 0.0031308 ? old * 12.92 : pow(old, 1.0 / gamma) * 1.055 - 0.055);
81.                   else
82.                       old = pow(old, 1.0 / gamma);
83.                   const int color = round(old * this->color_depth);
84.                   fout << (static_cast<unsigned char>(color));
85.               }
86.           }
```

```
87.           fout.flush();
88.           fout.close();
89.       }
90.
91.       double sum_between_0_and_1(double A, double B) {
92.           if (A + B > 1.0)
93.               return 1.0;
94.           if (A + B < 0.0)
95.               return 0.0;
96.           return A + B;
97.       }
98.
99.       // Дизеринг
100.      void Image::dithering(int bit, int type) {
101.          auto nearest_color = [bit](double pixel_color) -> double {
102.              return round(pixel_color * ( (1 << bit) - 1 )) / ( (1 << bit) - 1 );
103.          };
104.          errors.assign(height, std::vector<double>(width, 0));
105.
106.          // Без дизиринга
107.          if(type == 0) {
108.              for(int i = 0; i < height; i++)
109.                  for(int j = 0; j < width; j++)
110.                      image[i][j] = nearest_color(image[i][j]);
111.          }
112.          // Ordered (8x8)
113.          else if(type == 1) {
114.              for (int i = 0; i < height; i++)
115.                  for (int j = 0; j < width; j++)
116.                      image[i][j] = nearest_color(sum_between_0_and_1(image[i]
[j], ((ordered_dithering[i % 8][j % 8] / 64.0) - 0.5)));
117.          }
118.          // Random
119.          else if(type == 2) {
120.              for (int i = 0; i < height; i++)
121.                  for (int j = 0; j < width; j++)
122.                      image[i][j] = nearest_color(sum_between_0_and_1(image[i]
[j], (rand() * 1.0 / (RAND_MAX - 1) - 0.5)));
123.          }
124.          // Floyd-Steinberg
125.          else if(type == 3) {
126.              for (int i = 0; i < height; i++) {
127.                  for (int j = 0; j < width; j++) {
128.                      image[i][j] = sum_between_0_and_1(image[i][j], errors[i][j]);
129.                      const double color = nearest_color(image[i][j]);
130.                      const double error = (image[i][j] - color) / 16.0;
131.                      image[i][j] = color;
132.                      if (j + 1 < width)
133.                          errors[i][j + 1] += error * 7;
134.                      if (i + 1 < height) {
135.                          if (j - 1 >= 0)
136.                              errors[i + 1][j - 1] += error * 3;
137.                          errors[i + 1][j] += error * 5;
138.                          if (j + 1 < width)
139.                              errors[i + 1][j + 1] += error;
140.                      }
141.                  }
142.              }
143.          }
144.          // Jarvis, Judice, Ninke
145.          else if (type == 4) {
146.              for (int i = 0; i < height; i++) {
147.                  for (int j = 0; j < width; j++) {
```

```
148.                     image[i][j] = sum_between_0_and_1(image[i][j], errors[i][j]);
149.                     const double color = nearest_color(image[i][j]);
150.                     const double error = (image[i][j] - color) / 48.0;
151.                     image[i][j] = color;
152.                     if (j + 1 < width)
153.                         errors[i][j + 1] += error * 7;
154.                     if (j + 2 < width)
155.                         errors[i][j + 2] += error * 5;
156.                     if (i + 1 < height) {
157.                         if (j - 2 >= 0)
158.                             errors[i + 1][j - 2] += error * 3;
159.                         if (j - 1 >= 0)
160.                             errors[i + 1][j - 1] += error * 5;
161.                         errors[i + 1][j] += error * 7;
162.                         if (j + 1 < width)
163.                             errors[i + 1][j + 1] += (error * 5);
164.                         if (j + 2 < width)
165.                             errors[i + 1][j + 2] += (error * 3);
166.                     }
167.                     if (i + 2 < height) {
168.                         if (j - 2 >= 0)
169.                             errors[i + 2][j - 2] += (error * 1);
170.                         if (j - 1 >= 0)
171.                             errors[i + 2][j - 1] += (error * 3);
172.                         errors[i + 2][j] += (error * 5);
173.                         if (j + 1 < width)
174.                             errors[i + 2][j + 1] += (error * 3);
175.                         if (j + 2 < width)
176.                             errors[i + 2][j + 2] += (error * 1);
177.                     }
178.                 }
179.             }
180.         }
181.         // Sierra (Sierra-3)
182.         else if(type == 5) {
183.             for (int i = 0; i < height; i++) {
184.                 for (int j = 0; j < width; j++) {
185.                     image[i][j] = sum_between_0_and_1(image[i][j], errors[i][j]);
186.                     const double color = nearest_color(image[i][j]);
187.                     const double error = (image[i][j] - color) / 32.0;
188.                     image[i][j] = color;
189.                     if (j + 1 < width)
190.                         errors[i][j + 1] += (error * 5);
191.                     if (j + 2 < width)
192.                         errors[i][j + 2] += (error * 3);
193.                     if (i + 1 < height) {
194.                         if (j - 2 >= 0)
195.                             errors[i + 1][j - 2] += (error * 2);
196.                         if (j - 1 >= 0)
197.                             errors[i + 1][j - 1] += (error * 4);
198.                         errors[i + 1][j] += (error * 5);
199.                         if (j + 1 < width)
200.                             errors[i + 1][j + 1] += (error * 4);
201.                         if (j + 2 < width)
202.                             errors[i + 1][j + 2] += (error * 2);
203.                     }
204.                     if (i + 2 < height)
205.                     {
206.                         if (j - 1 >= 0)
207.                             errors[i + 2][j - 1] += (error * 2);
208.                         errors[i + 2][j] += (error * 3);
209.                         if (j + 1 < width)
210.                             errors[i + 2][j + 1] += (error * 2);
```

```
211.                         }
212.                     }
213.                 }
214.             }
215.             // Atkinson
216.             else if(type == 6) {
217.                 for (int i = 0; i < height; i++) {
218.                     for (int j = 0; j < width; j++) {
219.                         image[i][j] = sum_between_0_and_1(image[i][j], errors[i][j]);
220.                         const double color = nearest_color(image[i][j]);
221.                         const double error = (image[i][j] - color) / 8.0;
222.                         image[i][j] = color;
223.                         if (j + 1 < width)
224.                             errors[i][j + 1] += error;
225.                         if (j + 2 < width)
226.                             errors[i][j + 2] += error;
227.                         if (i + 1 < height)
228.                         {
229.                             if (j - 1 >= 0)
230.                                 errors[i + 1][j - 1] += error;
231.                             errors[i + 1][j] += error;
232.                             if (j + 1 < width)
233.                                 errors[i + 1][j + 1] += error;
234.                         }
235.                         if (i + 2 < height) {
236.                             errors[i + 2][j] += error;
237.                         }
238.                     }
239.                 }
240.             }
241.             // Halftone (4x4, orthogonal)
242.             else if (type == 7) {
243.                 for (int i = 0; i < height; i++)
244.                     for (int j = 0; j < width; j++)
245.                         image[i][j] = nearest_color(sum_between_0_and_1(image[i]
[j], (halftone[i % 4][j % 4] / 16.0 - 0.5)));
246.             }
247.         }
```