**Минобрнауки России**

**Федеральное государственное образовательное учреждение высшего образования «Юго-Западный государственный университет»**

**Факультет фундаментальной и прикладной информатики**

**кафедра Программной инженерии**

По дисциплине «Компьютерная графика»

**ЛАБОРАТОРНАЯ РАБОТА №3**

КОМПЬЮТЕРНАЯ ГРАФИКА: Растеризация отрезков прямых

Выполнили: ст. гр. ПО-33б

Лукьянчиков А.Е.

Проверил: Ефремов В. В.

Курск, 2025

# Вариант 6

Разработать программу, реализующую алгоритмы растериза ции отрезков с их последующей прорисовкой. Вывести рисунок в со ответствии с вариантом. Для вывода отрезков использовать: − алгоритм ЦДА; − алгоритм Брезенхема; − целочисленный алгоритм Брезенхема;

6. Вывести полилинию по заданным координатам четырёх вер шин.

Ссылка на открытый гит репозиторий https://github.com/xxyofu/GraphicsLaba3

# 2. Программа

# #include <iostream>

# #include <fstream>

# #include <vector>

# #include <string>

# #include <sstream>

# #include <cmath>

# #define PI 3.14159265358979323846

# using namespace std;

# struct RGB {

# unsigned char r, g, b;

# };

# struct Point {

# int x, y;

# };

# class Image1

# {

# public:

# int width, height;

# std::vector<RGB> pixels;

# Image1(const string& filepath)

# {

# ifstream stream(filepath, ios::binary);

# if (!stream) {

# cout << "Cant open file" << endl;

# return;

# }

# stream.ignore(100, '\n');

# stream >> width >> height;

# stream.ignore(100, '\n');

# stream.ignore(100, '\n');

# pixels.resize(width \* height);

# stream.read(reinterpret\_cast<char\*>(pixels.data()), pixels.size() \* sizeof(RGB));

# }

# void Replace\_Pixel(int x, int y, unsigned char r, unsigned char g, unsigned char b)

# {

# pixels[y \* width + x] = { r, g, b };

# }

# bool Save\_image(const string& filepath)

# {

# ofstream stream(filepath, ios::binary);

# if (!stream) {

# cout << "Cant make file" << endl;

# return false;

# }

# stream << "P6\n" << width << " " << height << "\n255\n";

# stream.write(reinterpret\_cast<const char\*>(pixels.data()), pixels.size() \* sizeof(RGB));

# return true;

# }

# };

# double Sign(double x){

# if(x>0)

# return 1.0;

# else if(x==0)

# return 0;

# else

# return -1.0;

# }

# class Canvas1 {

# public:

# int width, height;

# std::vector<RGB> pixels;

# Canvas1(const int& w, const int& h)

# {

# width = w;

# height = h;

# pixels.assign(width \* height,{255,255,255});

# }

# void Replace\_Pixel(int x, int y, RGB& color)

# {

# pixels[y \* width + x] = color;

# }

# void DrawLine(Point p1, Point p2, RGB& color)

# {

# int stepx = abs(p2.x - p1.x);

# int stepy = abs(p2.y - p1.y);

# int d = (stepx > stepy) ? stepx : stepy;

# int sx = (p1.x < p2.x) ? 1 : -1;

# int sy = (p1.y < p2.y) ? 1 : -1;

# int bx = 0;

# int by = 0;

# while (true) {

# if (p1.x == p2.x && p1.y == p2.y) break;

# Replace\_Pixel(p1.x, p1.y, color);

# bx += stepx;

# by += stepy;

# p1.x += sx\*(bx / d);

# p1.y += sy \* (by / d);

# bx %= d;

# by %= d;

# }

# 

# }

# void CopyLine(Point p1, Point p2, Point i1, Point i2, Image1 image)

# {

# int stepx = abs(p2.x - p1.x);

# int stepy = abs(p2.y - p1.y);

# int d = (stepx > stepy) ? stepx : stepy;

# int sx = (p1.x < p2.x) ? 1 : -1;

# int sy = (p1.y < p2.y) ? 1 : -1;

# int bx = 0;

# int by = 0;

# int istepx = abs(i2.x - i1.x);

# int istepy = abs(i2.y - i1.y);

# int id = (istepx > istepy) ? istepx : istepy;

# int isx = (i1.x < i2.x) ? 1 : -1;

# int isy = (i1.y < i2.y) ? 1 : -1;

# int ibx = 0;

# int iby = 0;

# while (true) {

# if (p1.x == p2.x && p1.y == p2.y) break;

# Replace\_Pixel(p1.x, p1.y, image.pixels[i1.y\*image.width+i1.x]);

# bx += stepx;

# by += stepy;

# p1.x += sx \* (bx / d);

# p1.y += sy \* (by / d);

# bx %= d;

# by %= d;

# ibx += istepx;

# iby += istepy;

# i1.x += isx \* (ibx / id);

# i1.y += isy \* (iby / id);

# ibx %= id;

# iby %= id;

# }

# }

# void CDA(Point start, Point end,RGB color)

# {

# double L;

# if (abs(start.x - end.x) >= abs(start.y - end.y)) {

# L = abs(start.x - end.x);

# }

# else {

# L = abs(start.y - end.y);

# }

# double dx = (double)(end.x - start.x) / L;

# double dy = (double)(end.y - start.y) / L;

# double x = start.x + 0.5 \* Sign(dx);

# double y = start.y + 0.5 \* Sign(dy);

# for (int i = 1; i < L + 1; i++) {

# Replace\_Pixel(floor(x),floor(y), color);

# x += dx;

# y += dy;

# }

# }

# void Brezenhem(Point start, Point end, RGB color)

# {

# double dx = (double)(end.x - start.x);

# double dy = (double)(end.y - start.y);

# double sx = Sign(dx);

# double sy = Sign(dy);

# dx = abs(dx);

# dy = abs(dy);

# int flag = 0;

# if (dx < dy) {

# double t = dx;

# dx = dy;

# dy = t;

# flag = 1;

# }

# double f = dy / dx - 0.5;

# double x = start.x;

# double y = start.y;

# for(int i =0;i<dx;i++)

# {

# Replace\_Pixel(floor(x), floor(y), color);

# if (f >= 0) {

# if (flag == 1) {

# x = x + sx;

# }

# else {

# y = y + sy;

# }

# f -= 1;

# }

# if (flag == 1) {

# y = y + sy;

# }

# else {

# x = x + sx;

# }

# f = f + dy / dx;

# };

# }

# void BrezenhemC(Point start, Point end, RGB color)

# {

# int dx = (end.x - start.x);

# int dy = (end.y - start.y);

# int sx = Sign(dx);

# int sy = Sign(dy);

# dx = abs(dx);

# dy = abs(dy);

# int flag = 0;

# if (dx < dy) {

# int t = dx;

# dx = dy;

# dy = t;

# flag = 1;

# }

# int f = 2\*dy - dx;

# int x = start.x;

# int y = start.y;

# for (int i = 0; i < dx; i++)

# {

# Replace\_Pixel(floor(x), floor(y), color);

# if (f >= 0) {

# if (flag == 1) {

# x = x + sx;

# }

# else {

# y = y + sy;

# }

# f -= 2 \* dx;

# }

# if (flag == 1) {

# y = y + sy;

# }

# else {

# x = x + sx;

# }

# f = f + 2\*dy;

# };

# }

# void CopyPart(Image1& image, Point destcircle\_center, Point circle\_center, int radius)

# {

# //Point e{ place.x + end.x - start.x, place.y + end.y - start.y };

# //(i-radius)/radius

# // for (int i = 0; i < radius\*2; i++)

# // {

# //     CopyLine({destcircle\_center.x+(i-radius),destcircle\_center.y+int(acos((i-radius)/radius)\*radius/PI/2)}, {destcircle\_center.x+(i-radius),destcircle\_center.y-int(acos((i-radius)/radius)\*radius/PI/2)}, {circle\_center.x+(i-radius),circle\_center.y+int(acos((i-radius)/radius)\*radius/PI/2)}, {circle\_center.x+(i-radius),circle\_center.y-int(acos((i-radius)/radius)\*radius/PI/2)}, image);

# // }

# for (int y = -radius; y <= radius; y++)

# {

# for (int x = -radius; x <= radius; x++)

# {

# 

# if (x\*x + y\*y <= radius\*radius)

# {

# 

# int src\_x = circle\_center.x + x;

# int src\_y = circle\_center.y + y;

# int dest\_x = destcircle\_center.x + x;

# int dest\_y = destcircle\_center.y + y;

# 

# 

# if (src\_x >= 0 && src\_x < image.width &&

# src\_y >= 0 && src\_y < image.height &&

# dest\_x >= 0 && dest\_x < image.width &&

# dest\_y >= 0 && dest\_y < image.height)

# {

# Replace\_Pixel(dest\_x, dest\_y, image.pixels[src\_y \* image.width + src\_x]);

# }

# }

# }

# }

# }

# void DrawFunk(Point zero, int l, RGB color)

# {

# for (int i = 1; i <= l; i += 1)

# {

# //cout << zero.x + i << ' '<< sin(2 \* PI \* ((double)i / step)) << '\n';

# DrawLine({zero.x+i,zero.y-int(log(double(i)/100.0))}, {zero.x+i+1,zero.y-int(log(double(i+1)/100.0))}, color);

# }

# }

# bool Save\_Canvas(const string& filepath)

# {

# ofstream stream(filepath, ios::binary);

# if (!stream) {

# cout << "Cant make file" << endl;

# return false;

# }

# stream << "P6\n" << width << " " << height << "\n255\n";

# stream.write(reinterpret\_cast<const char\*>(pixels.data()), pixels.size() \* sizeof(RGB));

# return true;

# }

# };

# int main(int argc, char \*argv[])

# {

# //string input = "input.ppm";

# string output = "output";

# if(argc==2){

# output = argv[1];

# }

# // }else if(argc==3){

# //     input = argv[1];

# //     output = argv[2];

# // }

# RGB color = { 0,0,0 };

# Canvas1 cda(100, 100);

# Canvas1 brezenhem(100,100);

# Canvas1 int\_brezenhem(100,100);

# // cda.CDA({40, 40},{50,50},color);

# std::vector<Point> polyline;

# polyline.push\_back({10, 20});

# polyline.push\_back({30, 20});

# polyline.push\_back({10, 40});

# polyline.push\_back({30, 40});

# for(int i = 0; i<polyline.size()-1; i++){

# cda.CDA(polyline[i], polyline[i+1], color);

# brezenhem.Brezenhem(polyline[i], polyline[i+1], color);

# int\_brezenhem.BrezenhemC(polyline[i], polyline[i+1], color);

# }

# cda.Save\_Canvas(output+".ppm");

# brezenhem.Save\_Canvas(output+"(1).ppm");

# int\_brezenhem.Save\_Canvas(output+"(2).ppm");

# }

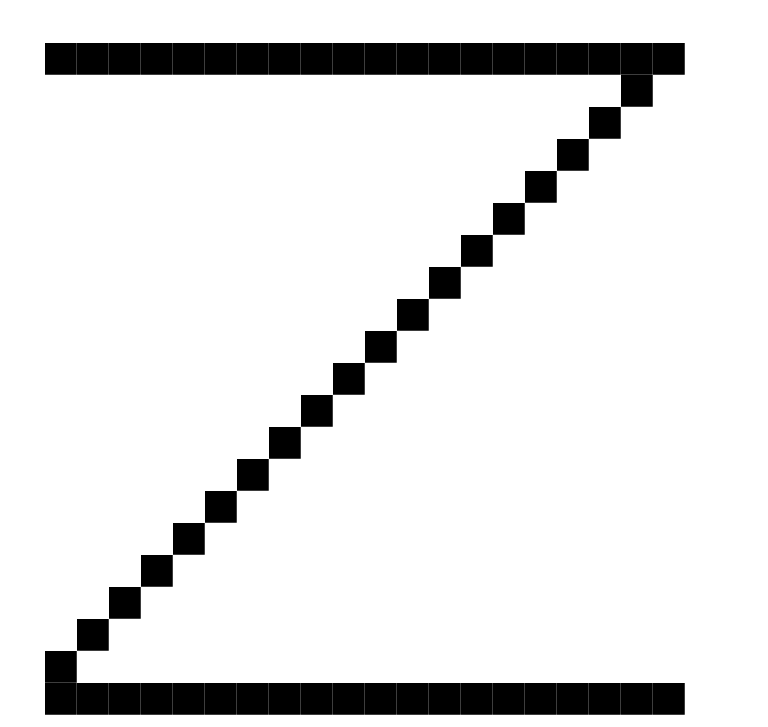
# 3. Форма программы

Программа выполнена на языке C++ и запускается в консоли. Вызов осуществляется из командной строки:

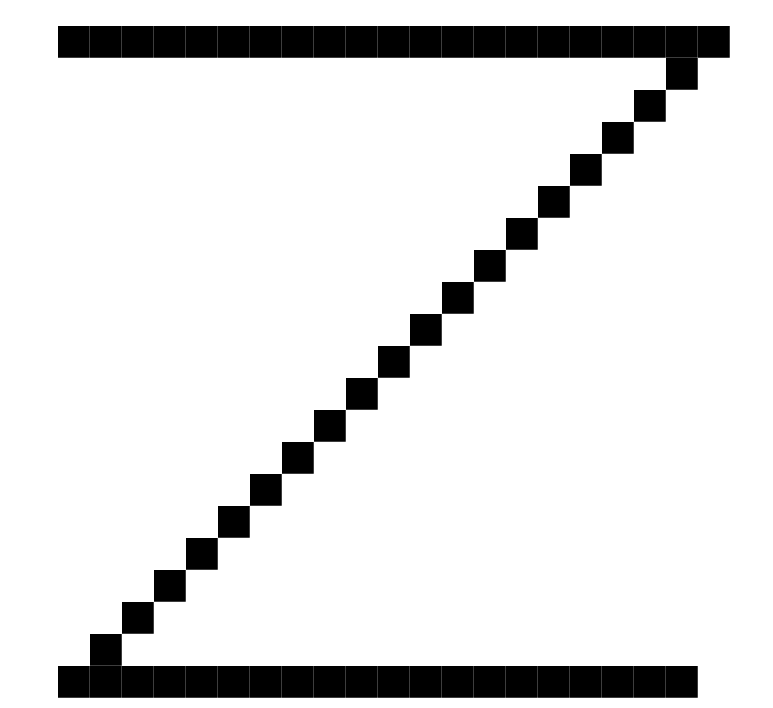


4. Созданные изображения

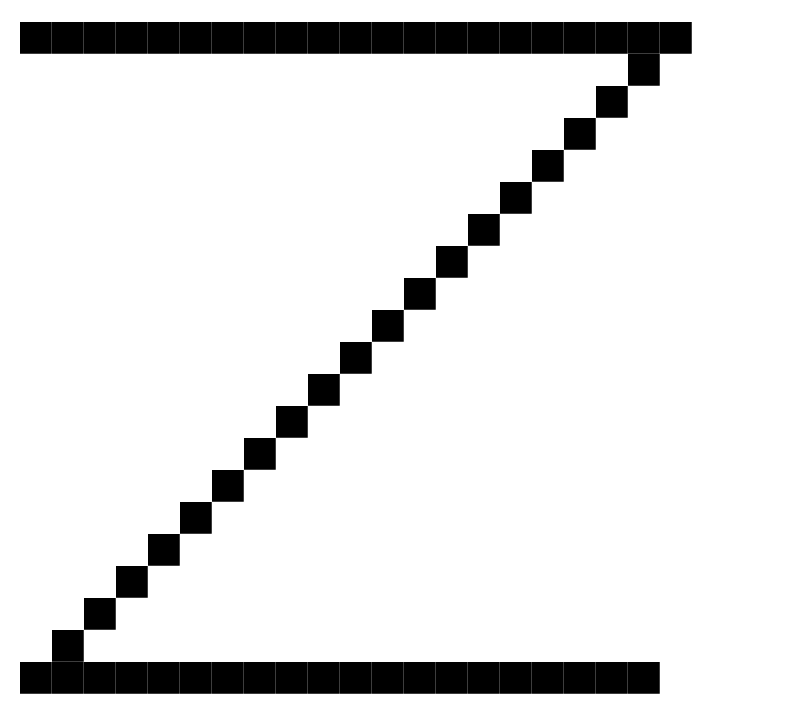
CDA



Brezenhem



Brezenhem(integer)



Между алгоритмом CDA и Брезенхейма видна небольшая разница в основании буквы.

# 5. Ответы на контрольные вопросы

**1. Что такое пиксель?**

**Пиксель** (от английского *picture element* — элемент изображения) — это наименьший логический и адресуемый элемент растрового изображения.

Если представить себе изображение как мозаику, то пиксель — это одна цветная плиточка в этой мозаике.

* **Физически** (на экране монитора, телефона, телевизора) — это крошечный светящийся элемент (субпиксели красного, зеленого и синего цвета), который вместе с другими формирует картинку.
* **Логически** (в памяти компьютера, в файле изображения) — это точка с определенными координатами (x, y) и цветом, закодированным в виде числового значения.

Количество пикселей на единицу площади определяет **разрешение** изображения. Чем больше пикселей, тем более детализированным и четким выглядит изображение.

**2. Что такое растровое изображение?**

**Растровое изображение** — это изображение, представленное в виде прямоугольной сетки пикселей (растра), каждый из которых имеет свой цвет.

Проще говоря, это "цифровая фотография". Файлы таких изображений хранят информацию о цвете каждого пикселя и общем размере сетки (например, 1920x1080 пикселей).

**Ключевые характеристики:**

* **Размер в пикселях:** Ширина и высота изображения (например, 800x600).
* **Цветовая глубина:** Количество бит, отводимое на кодирование цвета одного пикселя (например, 8 бит для черно-белого, 24 бита для TrueColor, что дает более 16 миллионов оттенков).
* **Разрешение:** Количество пикселей на дюйм (PPI - Pixels Per Inch). Влияет на физический размер при печати.

**Преимущества:**

* Идеальная передача плавных цветовых переходов и сложных эффектов (например, фотографии, реалистичные рисунки).

**Недостатки:**

* **Потеря качества при масштабировании:** При сильном увеличении изображение становится размытым и появляется "пикселизация" (становятся видны отдельные квадратные пиксели).
* Большой размер файла для изображений высокого разрешения.

**Примеры форматов:** JPEG, PNG, GIF, BMP, TIFF.

**3. Для чего нужны алгоритмы растеризации отрезков?**

**Алгоритмы растеризации отрезков** нужны для того, чтобы перевести математический, "идеальный" отрезок (заданный двумя конечными точками с вещественными координатами) в набор пикселей на растровом устройстве (экране, принтере).

Поскольку экран состоит из дискретных пикселей, а координаты отрезка непрерывны, компьютер должен решить, **какие именно пиксели нужно "зажечь"**, чтобы наилучшим образом аппроксимировать этот отрезок.

**Основные цели этих алгоритмов:**

1. **Визуализация:** Нарисовать отрезок на экране так, чтобы он выглядел прямым и ровным для человеческого глаза.
2. **Эффективность:** Сделать этот процесс максимально быстрым, так как в сложных сценах (например, в 3D-играх) нужно рисовать тысячи линий за доли секунды.
3. **Качество:** Обеспечить равномерную толщину и плотность линии, избежать "лестничного эффекта" (ступенчатости) или минимизировать его.

**4. Какие ещё алгоритмы растеризации отрезков существуют?**

Помимо простейшего алгоритма, основанного на уравнении прямой y = kx + b (который медленный и неэффективный из-за использования вещественной арифметики и округлений), существуют более совершенные алгоритмы. Самый известный и фундаментальный из них — **Алгоритм Брезенхема**.

**Алгоритм Брезенхема для отрезков (1962 год)**

Этот алгоритм использует только **целочисленную арифметику** (сложение, вычитание и умножение на 2), что делает его очень быстрым.

**Принцип работы (для отрезков с углом наклона меньше 45°):**  
На каждом шаге по оси x алгоритм выбирает, какой пиксель по оси y зажечь: текущий (y) или следующий (y+1). Выбор делается на основе анализа **ошибки** — расстояния между идеальным положением отрезка и центрами пикселей. Алгоритм оперирует целочисленной переменной решения, которая обновляется на каждом шаге.

**Преимущества:**

* Высокая скорость.
* Отсутствие операций с вещественными числами.
* Простота реализации.

**Алгоритм ЦДА (Цифровой Дифференциальный Анализатор)**

Этот алгоритм работает по принципу приращений.

**Принцип работы:**  
Вычисляются приращения dx и dy. Из них выбирается большее по модулю, и координаты отрезка на каждом шаге увеличиваются на dx / Steps и dy / Steps, где Steps = max(|dx|, |dy|). Это позволяет пройти по всем основным точкам отрезка.

**Преимущества:**

* Проще для понимания, чем Брезенхем.
* Работает с вещественными числами, но все же быстрее наивного метода.

**Недостатки по сравнению с Брезенхемом:**

* Использует вещественную арифметику, что медленнее целочисленной.
* Может накапливать ошибку округления.

**Итог:** **Алгоритм Брезенхема** является де-факто стандартом для растеризации отрезков в большинстве графических библиотек и аппаратного обеспечения из-за своей исключительной эффективности.