|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Лабораторная работа № 3**

|  |  |
| --- | --- |
| **Тема «Программная реализация основных алгоритмов построения отрезков и исследование их временных и визуальных характеристик»**  **Дисциплина Компьютерная графика**  **Студент Кузин Антон**  **Группа ИУ7-42Б**  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель Куров А.В.** |  |

Москва.

2020 г.

**Введение:**

К алгоритмам построения прямых есть несколько требований, такие как скорость работы, построенные отрезки должны выглядеть прямыми, совпадение конечной точки. Различные алгоритмы в разной степени удовлетворяет данным условиям,

**Цель:**

Реализация алгоритмов построения отрезков по методу цифрового дифференциального анализатора (ЦДА) и алгоритмов Брезенхема (действительного, целочисленного и с устранением ступенчатости), а также Ву и встроенного библиотечного метода и исследование их характеристик и сравнение полученных результатов.

**Техническое задание:**

Реализовать алгоритмы построения прямой на основе: метода ЦДА, алгоритма Брезенхема(действительного, целочисленного и с устранением ступенчатости), Ву, с помощью библиотечной функции.

Пользователь выбирает из списка определённый алгоритм, задаёт концы отрезка, его цвет.

Сравнение визуальных характеристик при рисовании спектра прямых.

Пользователь выбирает из списка алгоритм, задаёт цвет рисования, длину прямых и шаг изменения угла конца прямой.

**Практическая часть:**

1. **ЦДА**

int dda(const QLine &line, canvas\_t &canvas)

{

int dX = line.x2() - line.x1();

int dY = line.y2() - line.y1();

int l = qMax(abs(dX), abs(dY));

if (!l)

{

canvas.image->setPixel(line.x1(), line.y1(), canvas.color->rgb());

return 1;

}

double dx = (double) dX / l;

double dy = (double) dY / l;

double xf = line.x1();

double yf = line.y1();

int x, y;

for (int i = 0; i <= l; i++)

{

x = qRound(xf);

y = qRound(yf);

canvas.image->setPixel(x, y, canvas.color->rgb());

xf += dx;

yf += dy;

}

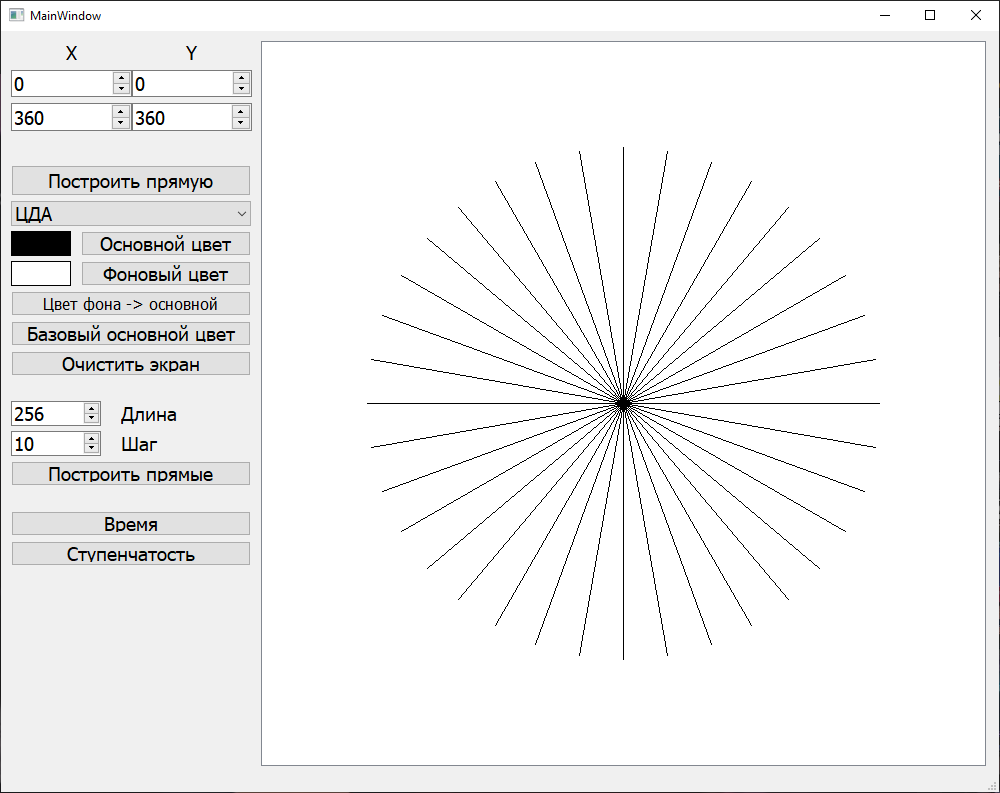
if (comparePoints(QPoint(x, y), line.p2()))

return 1;

else

return 0;

}



Этот метод является алгоритмически наиболее простым из всех, однако обладает значительными минусами, например работа с действительными величинами и округление их в дальнейшем, что увеличивает время работы алгоритма, а также заметна ступенчатость в связи с отсутствием какого-либо сглаживания.

1. **Брезенхем (действительный)**

int brezenhem(const QLine &line, canvas\_t &canvas)

{

if (line.p1() == line.p2())

{

canvas.image->setPixel(line.x1(), line.y1(), canvas.color->rgb());

return 1;

}

double x = line.x1(), y = line.y1();

int dx = line.x2() - line.x1();

int dy = line.y2() - line.y1();

int sx = sign(dx);

int sy = sign(dy);

dx = abs(dx);

dy = abs(dy);

bool swap = dy > dx;

if (swap)

qSwap(dx, dy);

double m = double(dy) / dx;

double e = m - 0.5;

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

{

canvas.image->setPixel(x, y, canvas.color->rgb());

if (e >= 0.0)

{

if (swap)

x += sx;

else

y += sy;

e--;

}

if (e < 0)

{

if (swap)

y += sy;

else

x += sx;

e += m;

}

}

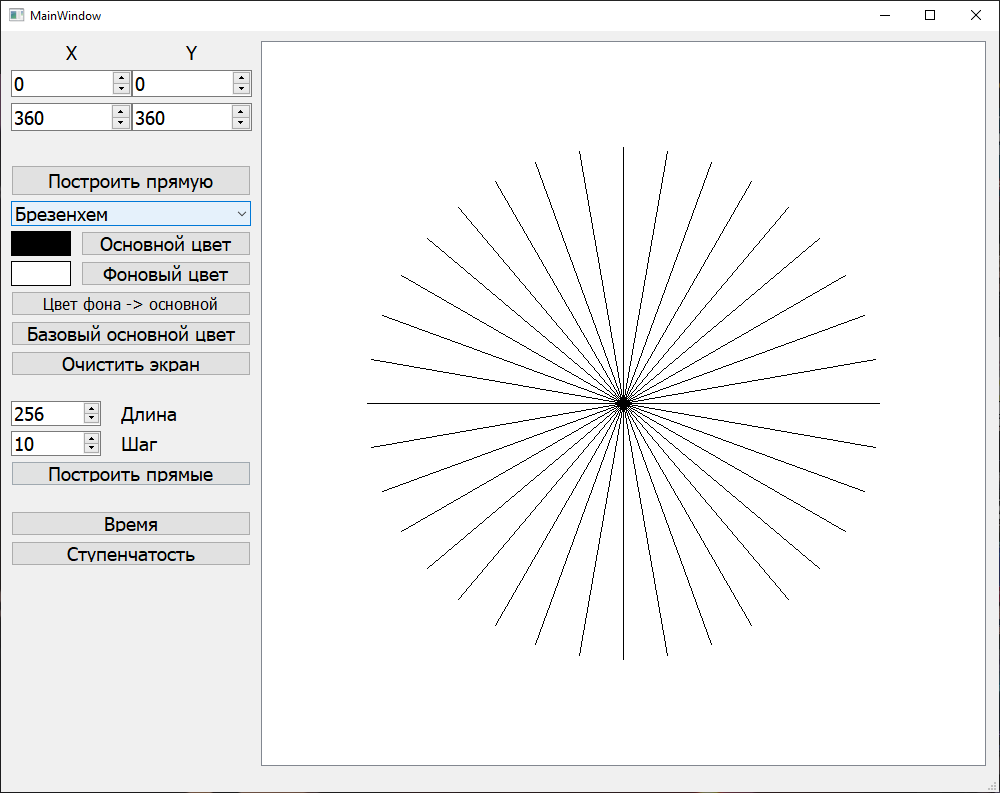
if (comparePoints(QPoint(x, y), line.p2()))

return 1;

else

return 0;

}



Данный алгоритм всё еще работает с действительными числами, что остается минусом, однако несмотря на это, как будет виднее далее, он работает быстрее метода ЦДА. Также стоит отметить, что метод Брезенхема выбирает пиксели аналогичные методу ЦДА.

1. **Брезенхем (целочисленный)**

int brezenhemInt(const QLine &line, canvas\_t &canvas)

{

if (line.p1() == line.p2())

{

canvas.image->setPixel(line.x1(), line.y1(), canvas.color->rgb());

return 1;

}

float x = line.x1(), y = line.y1();

int dx = line.x2() - line.x1();

int dy = line.y2() - line.y1();

int sx = sign(dx);

int sy = sign(dy);

dx = abs(dx);

dy = abs(dy);

int swap = dy > dx;

if (swap)

qSwap(dx, dy);

int e = dy \* 2 - dx;

for (int i = 0; i < dx + 1; i++)

{

canvas.image->setPixel(x, y, canvas.color->rgb());

if (e >= 0)

{

if (swap)

x += sx;

else

y += sy;

e -= 2 \* dx;

}

if (e < 0)

{

if (swap)

y += sy;

else

x += sx;

e += 2 \* dy;

}

}

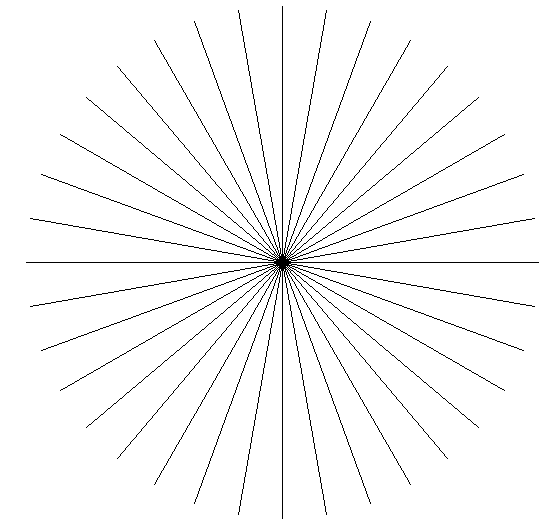
if (comparePoints(QPoint(x, y), line.p2()))

return 1;

else

return 0;

}



Это модификация действительного алгоритма Брезенхема, который избавлен от работы с действительными числами, что увеличивает скорость алгоритма, не влияя на результат его работы.

1. **Брезенхем (с устранением ступенчатости)**

int brezenhemAntialized(const QLine &line, canvas\_t &canvas)

{

if (line.p1() == line.p2())

{

canvas.image->setPixel(line.x1(), line.y1(), canvas.color->rgb());

return 1;

}

int i\_max = 255;

int dx = line.x2() - line.x1();

int dy = line.y2() - line.y1();

int sx = sign(dx);

int sy = sign(dy);

dx = abs(dx);

dy = abs(dy);

double x = line.x1();

double y = line.y1();

bool swapped = dy > dx;

if (swapped)

qSwap(dx, dy);

double m = 0;

if (dy)

m = double(i\_max \* dy) / dx;

double e = i\_max / 2.0;

double w = i\_max - m;

QColor color(canvas.color->rgba());

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

{

color.setAlpha(i\_max - e);

canvas.image->setPixel(x, y, color.rgba());

if (e <= w)

{

if (swapped)

y += sy;

else

x += sx;

e += m;

}

else

{

x += sx;

y += sy;

e -= w;

}

}

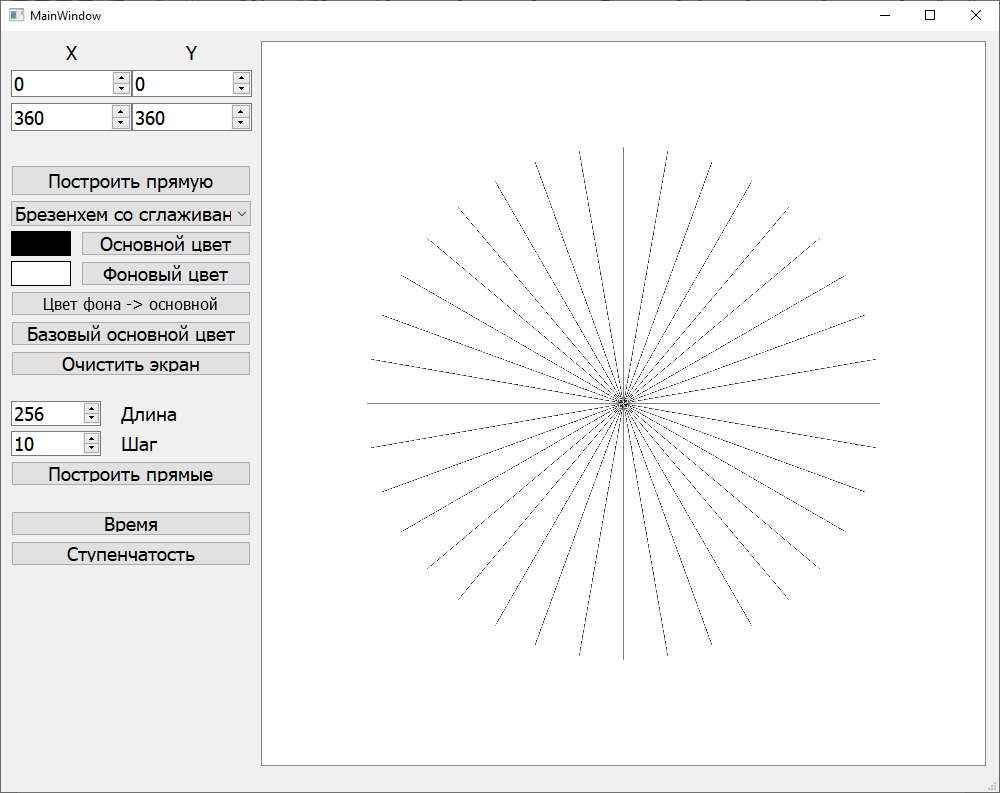
if (comparePoints(QPoint(x, y), line.p2()))

return 1;

else

return 0;

}



Данный метод является модификацией алгоритма Брезенхема, однако с устранением ступенчатости, однако в линии толщиной в один пиксель ступеньки становятся более заметны, таким образом этот метод может быть применим для построения многоугольников с закрашиванием внутренней части, или при построении отрезка двойной толщины.

1. **Библиотечный**

int defaultQt(const QLine &line, canvas\_t &canvas)

{

QPixmap pixmap = QPixmap::fromImage(\*(canvas.image));

QPainter painter(&pixmap);

painter.setPen(\*(canvas.color));

painter.drawLine(line);

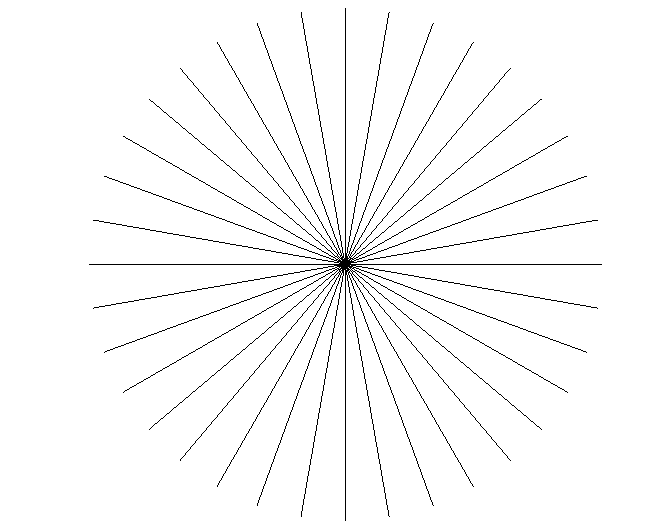
painter.end();

\*(canvas.image) = pixmap.toImage();

\*(canvas.image) = canvas.image->convertToFormat(QImage::Format\_ARGB32);

return 0;

}



1. **Ву**

int ipart(float a){

return floor(a);

}

float fpart(float a){

return a - floor(a);

}

float rpart(float a){

return 1 - fpart(a);

}

int wu(const QLine &line, canvas\_t &canvas)

{

if (line.p1() == line.p2())

{

canvas.image->setPixel(line.x1(), line.y1(), canvas.color->rgb());

return 1;

}

int x1 = line.x1(), y1 = line.y1();

int x2 = line.x2(), y2 = line.y2();

int dx = x2 - x1;

int dy = y2 - y1;

int swap = abs(dy) > abs(dx);

if (swap)

{

qSwap(x1, y1);

qSwap(x2, y2);

qSwap(dx, dy);

}

if (x2 < x1)

{

qSwap(x1, x2);

qSwap(y1, y2);

}

dx = x2 - x1;

dy = y2 - y1;

QColor color(canvas.color->rgba());

float angle = dx ? (float) dy / dx : 1;

float y = y1;

int s;

for (int x = x1; x <= x2; x++)

{

s = sign(y);

if (swap)

{

color.setAlphaF(rpart(y));

canvas.image->setPixel(ipart(y), x, color.rgba());

color.setAlphaF(fpart(y));

canvas.image->setPixel(ipart(y) + s, x, color.rgba());

}

else

{

color.setAlphaF(rpart(y));

canvas.image->setPixel(x, ipart(y), color.rgba());

color.setAlphaF(fpart(y));

canvas.image->setPixel(x, ipart(y) + s, color.rgba());

}

y += angle;

}

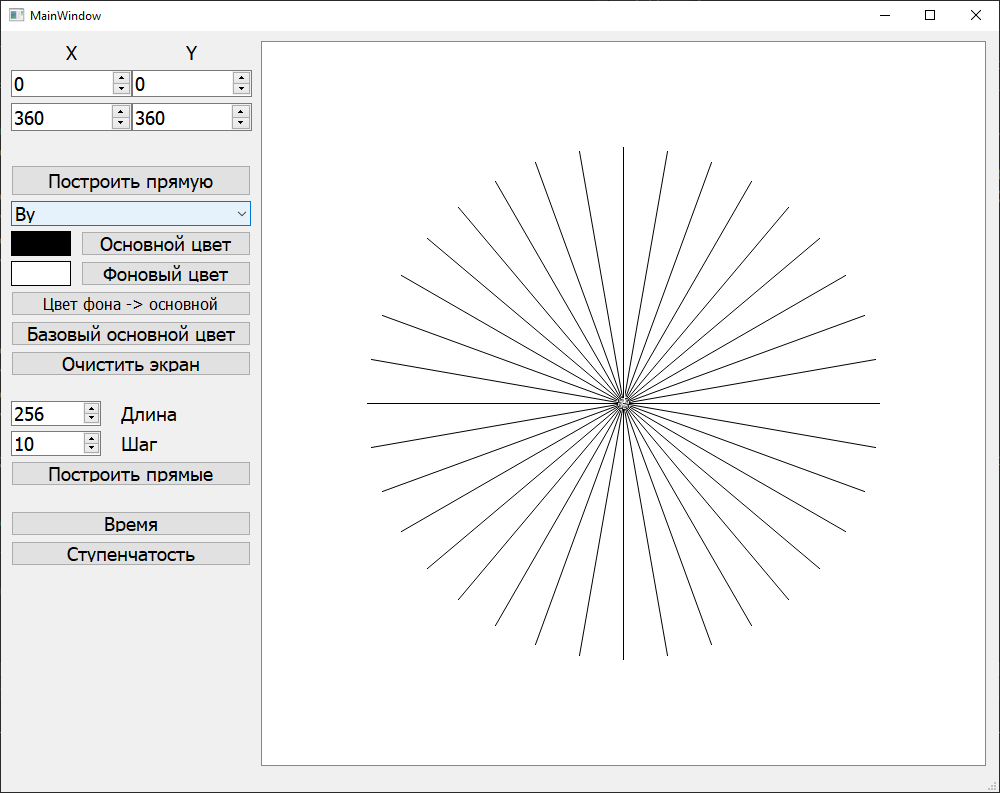
if (comparePoints(QPoint(x2, y), line.p2()))

return 1;

else

return 0;

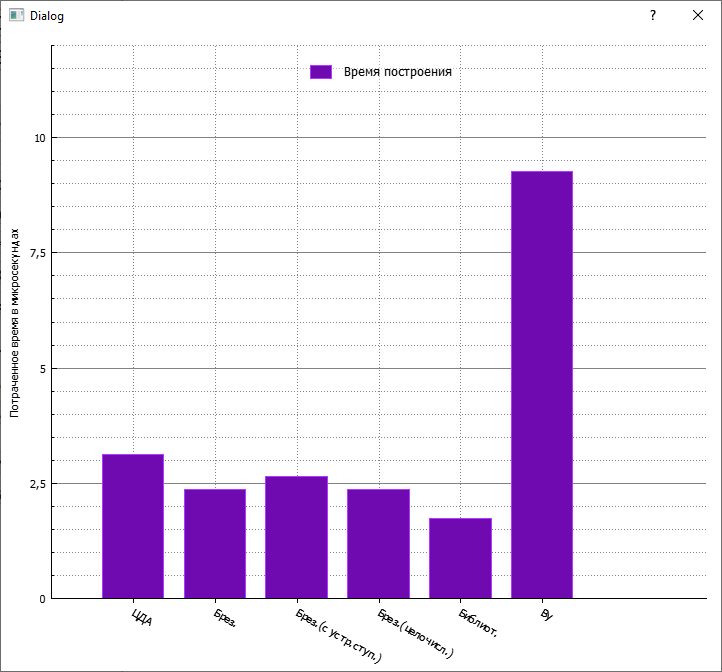
}



Отрезки, построенные с применением метода Ву выглядят наиболее прямыми, однако это требует несколько большего времени построения в сравнении, например с методом целочисленным Брезенхема в связи с необходимостью менять яркость пикселя, также он работает с действительными значениями, что также замедляет работу.

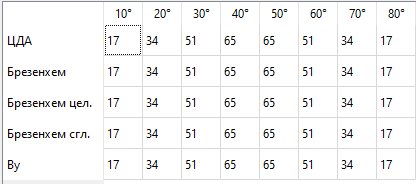
**Исследование временных характеристик**

Приведенные данные – среднее значение при построении отрезков длиной 180 единиц, с шагом 2 градуса от 0 до 180 градусов.



**Исследование ступенчатости**

Данные приведены для отрезков длиной 100 единиц.



**Вывод**

Подводя итог, следует отметить, что алгоритм ЦДА и метод Брезенхема выбирают идентичные пиксели, однако метод Брезенхема работает быстрее. Прямые, полученные методом Ву выглядят наиболее прямыми, однако требуют больше всего времени. Исходя из всего вышеперечисленного, становится ясно, что при выборе метода построения прямой необходимо выбирать между качеством получаемого изображения и скоростью.