|  |  |
| --- | --- |
| **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 № \_5\_**

|  |  |
| --- | --- |
| **Тема \_ РЕАЛИЗАЦИЯ И ИССЛЕДОВАНИЕ АЛГОРИТМОВ РАСТРОВОГО ЗАПОЛНЕНИЯ СПЛОШНЫХ ОБЛАСТЕЙ\_**  **Студент \_Чаушев Александър Красимиров\_\_\_\_\_\_\_\_**  **Группа \_ИУ7-46Б\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель \_Куров А. В.\_\_\_\_** |  |

Москва.

2020 г.

**Цель работы:** Реализация и исследование одного из алгоритмов( алгоритм заполнения по ребрам) растрового заполнения области .

**Входные данные:** Координаты вершин многоугольника, цвет заполнения, выбор режима — с задержкой и без.

**Выходные данные:** Конечное изображение

**Ошибочные ситуации:** Программа прекращается, если хотя бы один из входных данных не корректен.

**Теоретическая часть:**

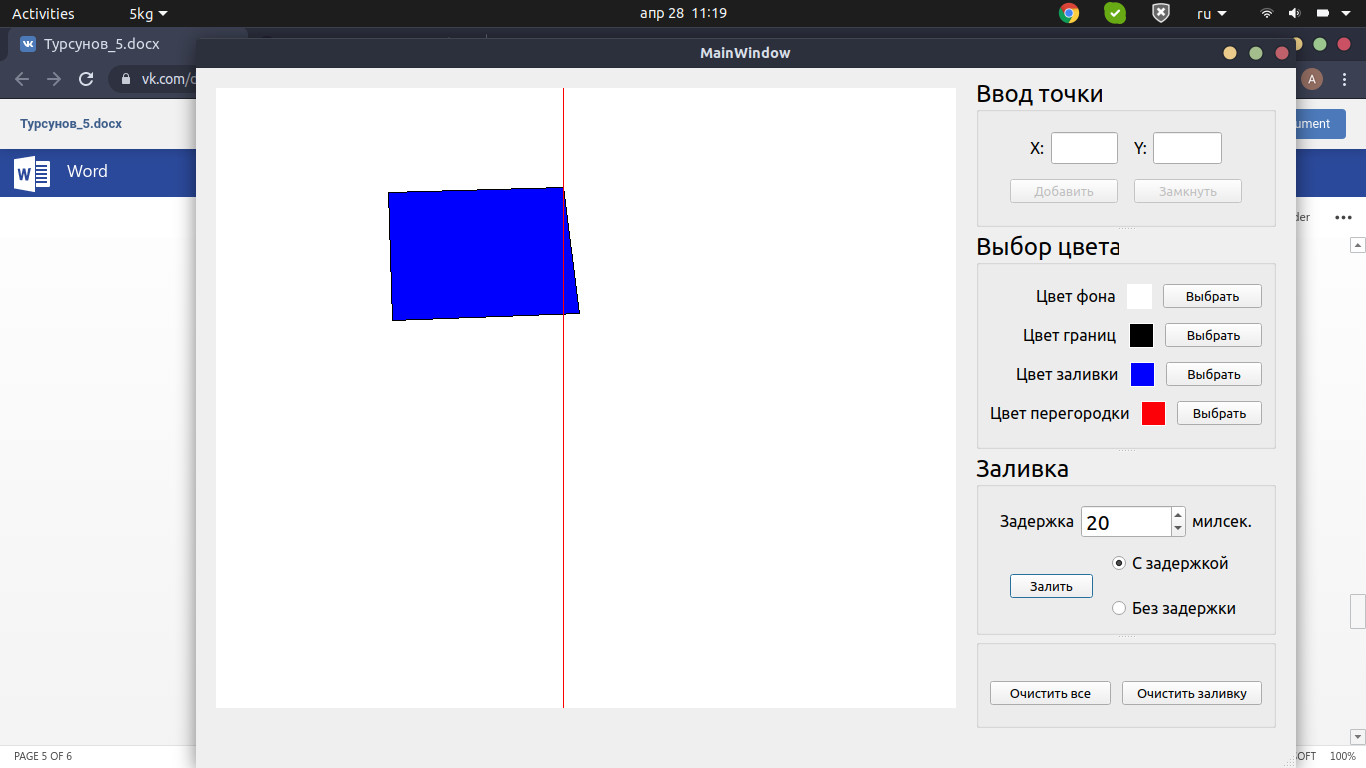
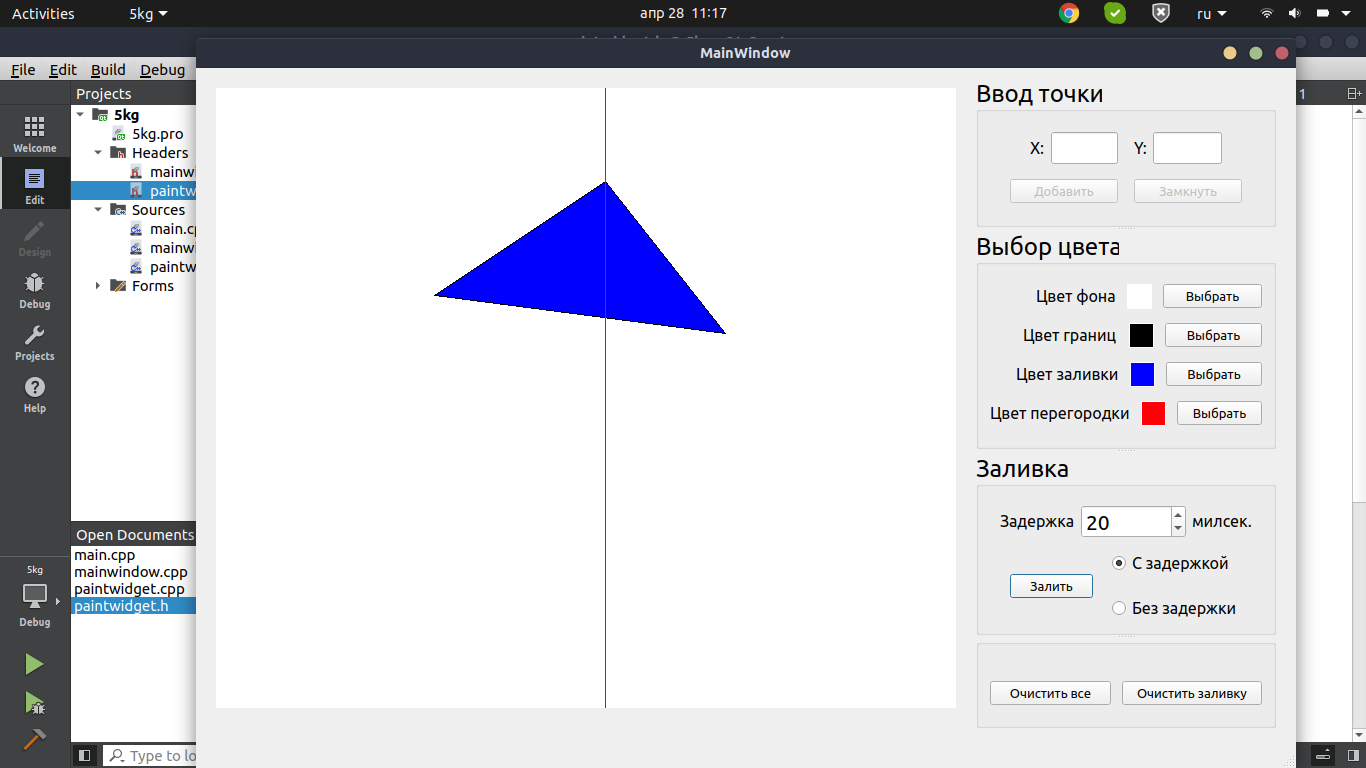
Данный алгоритм весьма прост. Суть его можно сформулировать следующим образом: Для каждой сканирующей строки, пересекающей ребро многоугольника в точке (x1,y1) дополнить все пикселы, лежащие справа от этой точки (x1,y1). Под дополнением здесь понимается операция изменения цвета пиксела: если цвет пиксела был фоновым, то он меняется на цвет закраски и наоборот, если цвет пиксела соответствовал цвету закраски, то он меняется на фоновый цвет. Другими словами, происходит инвертирование цвета пиксела. К каждому ребру многоугольника алгоритм применяется индивидуально. Пересечения строк сканирования осуществляется аналогично алгоритму с упорядоченным списком ребер. Рекомендуется использовать этот алгоритм совместно с дисплейным файлом, что позволяет выбирать и обрабатывать ребра в любом порядке. Ибо тогда при обработке ребра многоугольника обрабатываются пикселы из дисплейного файла, соответствующие точкам пересечения ребра со строкой сканирования. После обработки всех ребер многоугольника, ограничивающего закрашиваемую область, дисплейный файл выводится в порядке сканирования на экран графического дисплея. Данный алгоритм характеризуется существенными недостатками: многократной обработкой одного и того же пиксела в случае областей закрашивания сложной формы; зависимостью алгоритма от операций ввода/вывода.

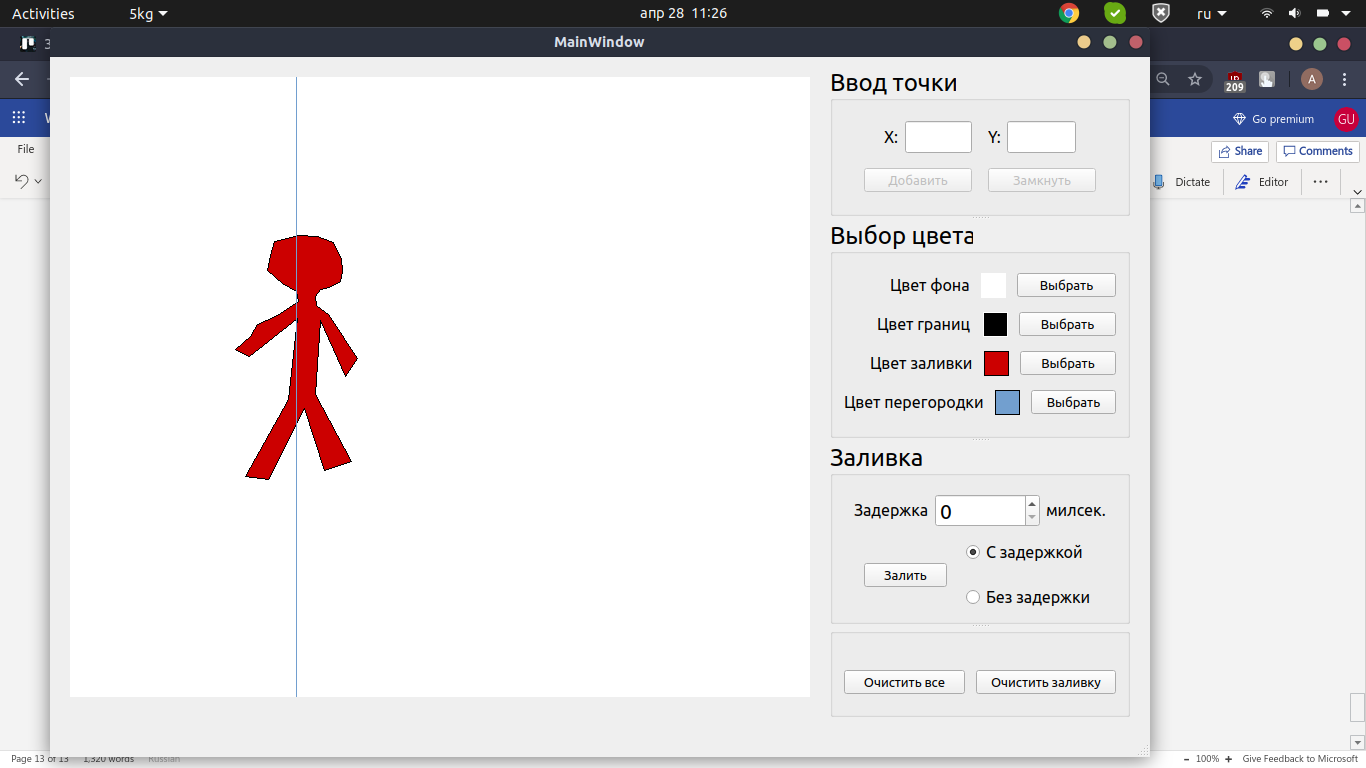
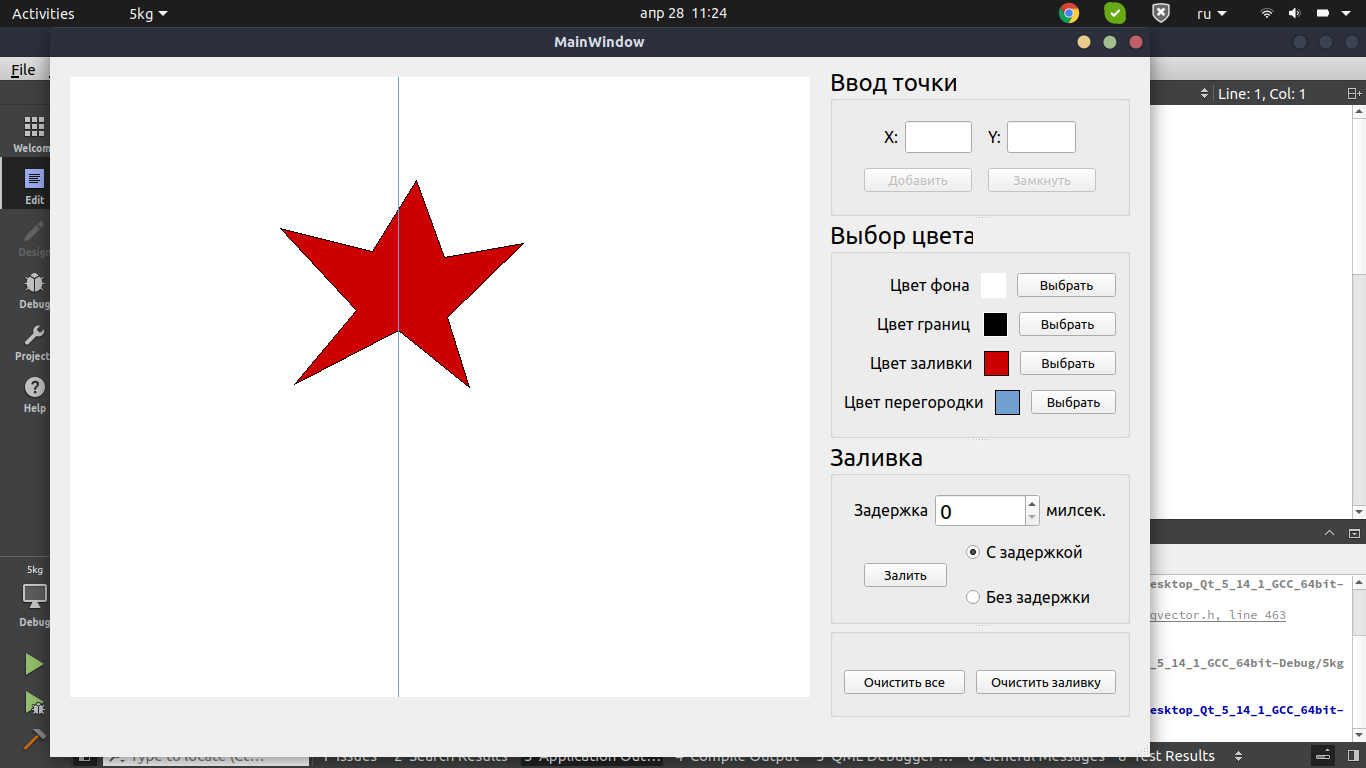
Для сокращения числа обрабатываемых пикселов используется “перегородка”. В этом случае алгоритм модифицируется следующим образом: анализируется каждая точка пересечения каждого ребра многоугольника со строкой сканирования. Если текущая точка пересечения находится слева от перегородки, то активизируются все пикселы, центры которых лежат справа от точки пересечения ребра со строкой сканирования и слева от перегородки. Если же точка пересечения находится справа от перегородки, то активизируются пикселы, центры которых расположены слева или на пересечении строки сканирования с ребром многоугольника и справа от перегородки Рекомендуется проводить перегородку через одну из вершин многоугольника и реализовывать данный алгоритм совместно с использованием дисплейного файла. Недостатком алгоритма заполнения с перегородкой все же остается неоднократная обработка части пикселов.

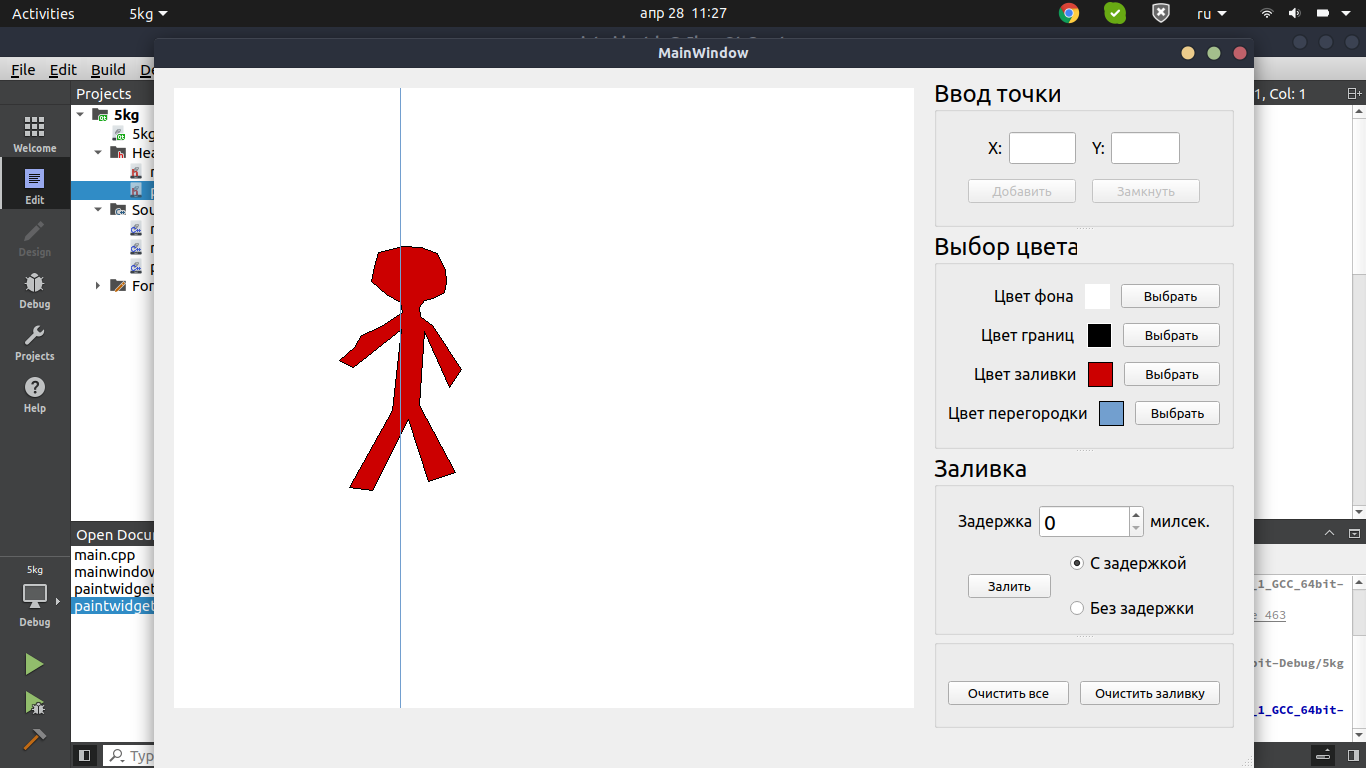
**Код программы:**

#include "paintwidget.h"  
  
PaintWidget::PaintWidget(QWidget \*parent) : QWidget(parent)  
{  
 widget\_width = 740;  
 widget\_height = 620;  
 x\_min = 0;  
 x\_max = widget\_width;  
 bg\_color = Qt::white;  
 borders\_color = Qt::black;  
 fill\_color = Qt::blue;  
 separator\_color = Qt::red;  
 drawing\_enabled = false;  
 image = new QImage(widget\_width, widget\_height, QImage::Format\_RGB32);  
 image->fill(bg\_color);  
 setGeometry(20, 20, widget\_width, widget\_height);  
}  
  
void PaintWidget::redrawImage()  
{  
 image->fill(bg\_color);  
 QPainter painter(image);  
 painter.setPen(borders\_color);  
 for (const QPoint &point: points)  
 {  
 painter.drawPoint(point);  
 }  
 for (const auto &edge: edges)  
 {  
 drawBresenham(painter, points[edge.first], points[edge.second]);  
 }  
 findBorders();  
 setSeparator(painter);  
 update();  
}  
  
void PaintWidget::addLine(QPainter &painter, const QPoint &cur\_point)  
{  
 edges.push\_back(std::make\_pair(points.size() - 1, points.size()));  
 if (cur\_point.y() < last\_point.y())  
 qSwap(edges.back().first, edges.back().second);  
 points.push\_back(cur\_point);  
 drawBresenham(painter, last\_point, cur\_point);  
 last\_point = cur\_point;  
 update();  
 emit pointsChanged(points.size(), start\_point\_index);  
}  
  
void PaintWidget::setStartPoint(QPainter &painter, const QPoint &cur\_point)  
{  
 start\_point\_index = points.size();  
 points.push\_back(cur\_point);  
 painter.drawPoint(cur\_point);  
 last\_point = cur\_point;  
 drawing\_enabled = true;  
 update();  
}  
  
void PaintWidget::findBorders()  
{  
 if (!points.empty())  
 {  
 x\_max = points[0].x();  
 x\_min = points[0].x();  
 for (const QPoint &point: points)  
 {  
 if (point.x() > x\_max)  
 x\_max = point.x();  
 if (point.x() < x\_min)  
 x\_min = point.x();  
 }  
 }  
}  
  
void PaintWidget::setSeparator(QPainter &painter)  
{  
 painter.setPen(separator\_color);  
 double middle = (x\_max - x\_min) / 2 + x\_min;  
 if (!points.empty())  
 {  
 x\_sep = points[0].x();  
 for (const QPoint &point: points)  
 {  
 if (abs(middle - point.x()) < abs(middle - x\_sep))  
 x\_sep = point.x();  
 }  
 }  
 else  
 {  
 x\_sep = static\_cast<int>(middle);  
 }  
 painter.drawLine(x\_sep, 0, x\_sep, widget\_height);  
}  
  
void PaintWidget::drawPoint(QPainter &painter, const int &x, const int &y)  
{  
 QColor cur = image->pixelColor(x, y);  
 if (cur == fill\_color)  
 {  
 painter.setPen(bg\_color);  
 painter.drawPoint(x, y);  
 }  
 else if (cur == bg\_color)  
 {  
 painter.setPen(fill\_color);  
 painter.drawPoint(x, y);  
 }  
}  
  
void PaintWidget::drawBresenham(QPainter &painter, const QPoint &first, const QPoint &second)  
{  
 int xn = first.x();  
 int yn = first.y();  
 int xk = second.x();  
 int yk = second.y();  
 if (xn == xk && yn == yk)  
 {  
 painter.drawPoint(xk, yk);  
 }  
 else  
 {  
 int dx = xk - xn;  
 int dy = yk - yn;  
 int sx = sign(dx);  
 int sy = sign(dy);  
 dx = abs(dx);  
 dy = abs(dy);  
 bool exchange;  
 if (dy > dx)  
 {  
 qSwap(dx, dy);  
 exchange = true;  
 }  
 else  
 exchange = false;  
 int e = 2 \* dy - dx;  
 int x = xn, y = yn;  
 for (int i = 0; i < dx + 1; i++)  
 {  
 painter.drawPoint(x, y);  
 if (e >= 0)  
 {  
 if (exchange)  
 x += sx;  
 else  
 y += sy;  
 e -= 2 \* dx;  
 }  
 if (exchange)  
 y += sy;  
 else  
 x += sx;  
 e += 2 \* dy;  
 }  
 }  
}  
  
int PaintWidget::sign(int x)  
{  
 if (x > 0)  
 return 1;  
 else if (x == 0)  
 return 0;  
 return -1;  
}  
  
void PaintWidget::finishFigure(QPainter &painter)  
{  
 if (start\_point\_index + 2 < points.size())  
 {  
 edges.push\_back(std::make\_pair(points.size() - 1, start\_point\_index));  
 if (last\_point.y() > points[start\_point\_index].y())  
 qSwap(edges.back().first, edges.back().second);  
 drawBresenham(painter, last\_point, points[start\_point\_index]);  
 drawing\_enabled = false;  
 emit pointsChanged(points.size(), points.size());  
 update();  
 }  
}  
  
void PaintWidget::setBGColor(QColor &color)  
{  
 bg\_color.setRgb(color.rgb());  
 redrawImage();  
}  
  
void PaintWidget::setBordersColor(QColor &color)  
{  
 borders\_color.setRgb(color.rgb());  
 redrawImage();  
}  
  
void PaintWidget::setFillColor(QColor &color)  
{  
 fill\_color.setRgb(color.rgb());  
 redrawImage();  
}  
  
void PaintWidget::setSeparatorColor(QColor &color)  
{  
 separator\_color.setRgb(color.rgb());  
 redrawImage();  
}  
  
void PaintWidget::drawByButton(QPoint &cur\_point)  
{  
 QPainter painter(image);  
 painter.setPen(borders\_color);  
 if (drawing\_enabled)  
 {  
 addLine(painter, cur\_point);  
 }  
 else  
 {  
 setStartPoint(painter, cur\_point);  
 }  
}  
  
void PaintWidget::finishByButton()  
{  
 if (drawing\_enabled)  
 {  
 QPainter painter(image);  
 painter.setPen(borders\_color);  
 finishFigure(painter);  
 }  
}  
  
void PaintWidget::clearAll()  
{  
 image->fill(bg\_color);  
 if (!points.empty())  
 {  
 points.clear();  
 edges.clear();  
 drawing\_enabled = false;  
 emit pointsChanged(points.size(), points.size());  
 }  
 update();  
}  
  
void PaintWidget::clearFill()  
{  
 redrawImage();  
}  
  
void PaintWidget::fillFigure(const int &time)  
{  
 redrawImage();  
 QPainter painter(image);  
 if (drawing\_enabled)  
 {  
 painter.setPen(borders\_color);  
 finishFigure(painter);  
 }  
 if (!drawing\_enabled)  
 {  
 for (const auto &edge: edges)  
 {  
 QPoint first\_point = points[edge.first];  
 QPoint second\_point = points[edge.second];  
 int dx = first\_point.x() - second\_point.x();  
 int dy = first\_point.y() - second\_point.y();  
 double k = static\_cast<double>(dx) / dy;  
 if (first\_point.x() < x\_sep && second\_point.x() < x\_sep)  
 {  
 double x\_start = first\_point.x();  
 for (int y = first\_point.y(); y < second\_point.y(); y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x < x\_sep; x++)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 }  
 else if (first\_point.x() >= x\_sep && second\_point.x() >= x\_sep)  
 {  
 double x\_start = first\_point.x();  
 for (int y = first\_point.y(); y < second\_point.y(); y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x > x\_sep; x--)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 }  
 else if (first\_point.x() < x\_sep && second\_point.x() >= x\_sep)  
 {  
 double x\_start = first\_point.x();  
 int y\_sep = static\_cast<int>(round((x\_sep - first\_point.x()) / k) + first\_point.y());  
 for (int y = first\_point.y(); y < y\_sep; y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x < x\_sep; x++)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 for (int y = y\_sep; y < second\_point.y(); y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x > x\_sep; x--)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 }  
 else  
 {  
 double x\_start = first\_point.x();  
 int y\_sep = static\_cast<int>(round((x\_sep - first\_point.x()) / k) + first\_point.y());  
 for (int y = first\_point.y(); y < y\_sep; y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x > x\_sep; x--)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 for (int y = y\_sep; y < second\_point.y(); y++, x\_start += k)  
 {  
 for (int x = static\_cast<int>(round(x\_start)); x < x\_sep; x++)  
 {  
 drawPoint(painter, x, y);  
 }  
 sleepFeature(time);  
 update();  
 }  
 }  
 }  
 }  
}  
  
void PaintWidget::mousePressEvent(QMouseEvent \*event)  
{  
 QPainter painter(image);  
 painter.setPen(borders\_color);  
 if (drawing\_enabled)  
 {  
 if (event->button() == Qt::LeftButton)  
 {  
 QPoint cur\_point = event->pos();  
 if (event->modifiers() == Qt::ShiftModifier)  
 {  
 if (abs(last\_point.x() - cur\_point.x()) <= abs(last\_point.y() - cur\_point.y()))  
 {  
 cur\_point.setX(last\_point.x());  
 }  
 else  
 {  
 cur\_point.setY(last\_point.y());  
 }  
 }  
 addLine(painter, cur\_point);  
 }  
 else if (event->button() == Qt::RightButton)  
 {  
 finishFigure(painter);  
 }  
 }  
 else  
 {  
 if (event->button() == Qt::LeftButton)  
 {  
 QPoint cur\_point = event->pos();  
 setStartPoint(painter, cur\_point);  
 }  
 }  
}  
  
void PaintWidget::paintEvent(QPaintEvent \*event)  
{  
 QPainter painter(this);  
 painter.drawImage(0, 0, \*image);  
 Q\_UNUSED(event);  
}  
  
void PaintWidget::sleepFeature(const int &time)  
{  
 QTime end = QTime::currentTime().addMSecs(time);  
 while (QTime::currentTime() < end)  
 {  
 QCoreApplication::processEvents(QEventLoop::AllEvents, 1);  
 }  
  
 return;  
}

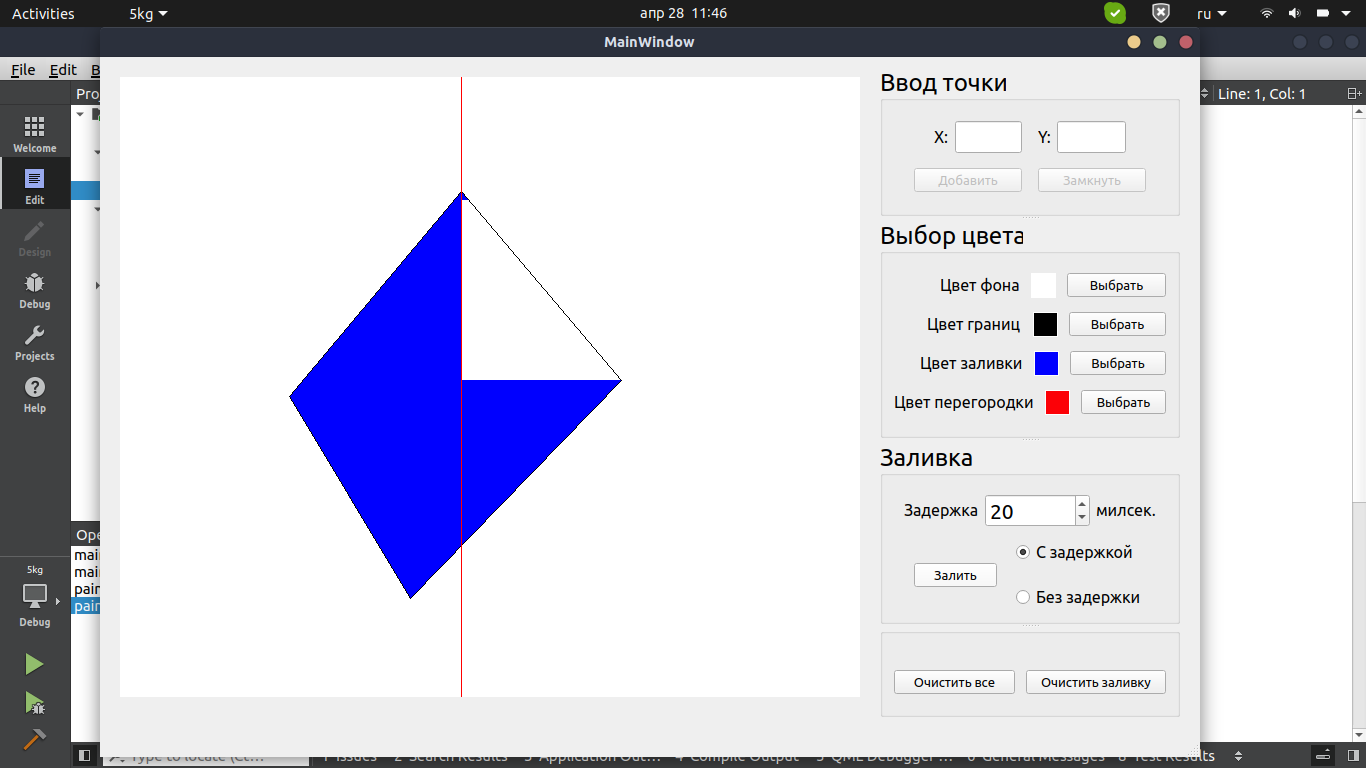
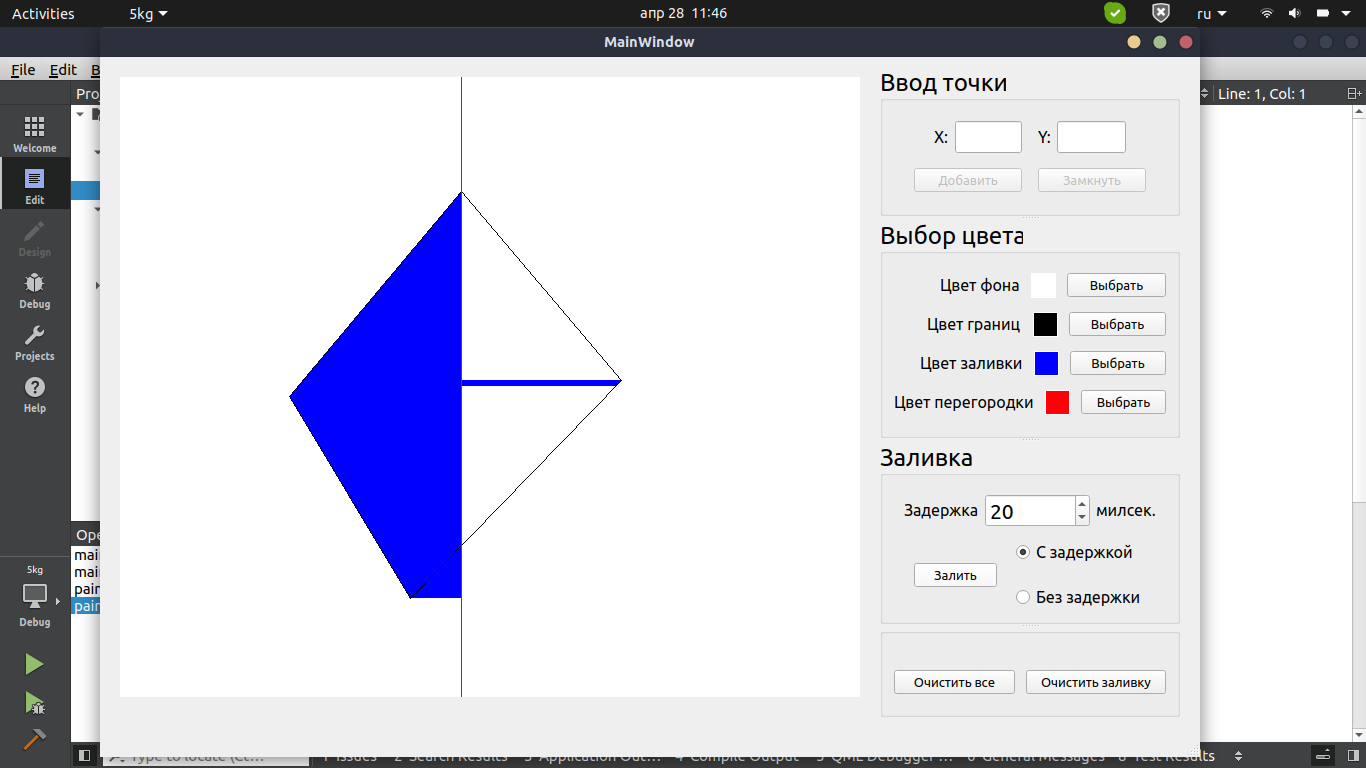
**Пример работы:**

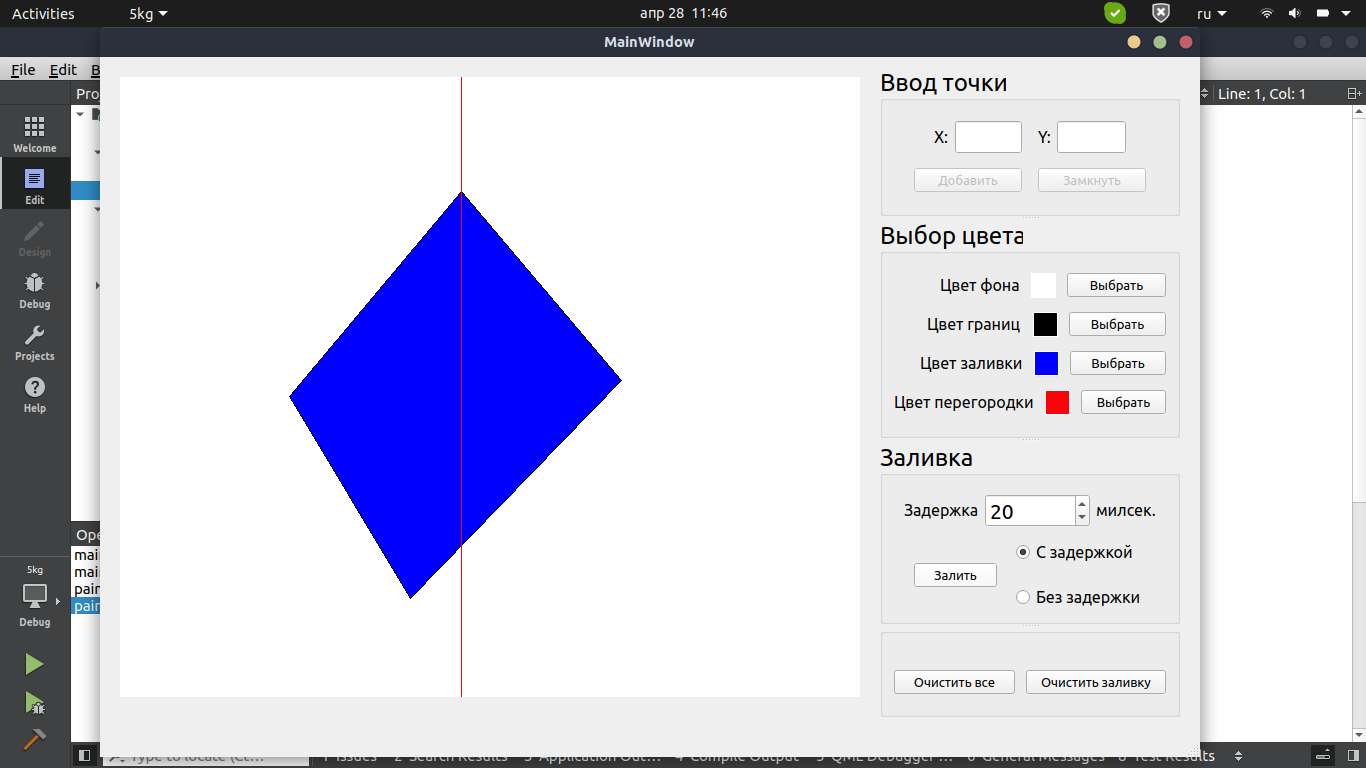






**Задержка**





**Замер Времени:**

