|  |  |
| --- | --- |
| 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 № 8**

|  |  |
| --- | --- |
| **Тема** Реализация алгоритма отсечения отрезка произвольным выпуклым отсекателем. (алгоритм Кируса-Бека)  **Студент** Склифасовский Д. О.  **Группа ИУ 7-45**  **Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** |  |

Москва.

2020 г.

**Цель работы:** изучение и программная реализация алгоритма отсечения отрезка.

**Техническое задание:** необходимо обеспечить ввод отсекателя – произвольного многоугольника. Высветить его первым цветом. Также необходимо обеспечить ввод нескольких (до десяти) различных отрезков (высветить их вторым цветом). Отрезки могут иметь произвольное расположение: горизонтальные, вертикальные, имеющие произвольный наклон.

Предусмотреть ввод отрезков, параллельных границе отсекателя.

Ввод осуществлять с помощью мыши и нажатия других клавиш.

Выполнить отсечение отрезков, показав результат третьим цветом. Исходные отрезки не удалять.

**Теоретический материал:**

Для создания надежного алгоритма отсечения нужно иметь хороший метод определения местоположения относительно окна (внутри, на границе или вне его) точки, принадлежащей отрезку. Для этой цели в алгоритме Кируса-Бека используется вектор нормали.

**Алгоритм:**

1. Ввод изначальных данных P1, P2, n вершин, массив вершин
2. D = P2 – P1
3. tLower = 0, tTop = 1
4. цикл отсечения по всем сторонам отсекателя i = 1..n
5. определяем весовой множитель удаленности W
6. определяем нормаль N
7. вычисляем скалярного произведения D и N
8. вычисляем скалярного произведения D и N
9. Если Dskalar равен 0:
   1. Если Wscalar равен 0 – конец
10. Иначе:
    1. t = -Wscalar / Dscalar
    2. Если Dscalar больше 0:
       1. Если t больше 1 – конец
       2. Иначе – tLower = max(tLower, t)
    3. Иначе если Dscalar меньше 0:
       1. Если t меньше 0 – конец
       2. Иначе – tTop = min(tTop, t)
11. Если tLower < tTop – нарисовать P(tLower) до P(tTop)

**Моя реализация алгоритма:**

    def CyrusBeck(self, currentLine, points, norm):

        # инициализируем пределы значений параметра, предполагая, что весь отрезок полностью видимый

        # максимизируем t нижнее и t верхнее, исходя из того что 0 <= t <= 1

        tLower = 0

        tTop = 1

        # вычисляем директрису directrix= p1 - p2

        directrix = QtCore.QPointF()

        directrix.setX(currentLine[1][0] - currentLine[0][0])

        directrix.setY(currentLine[1][1] - currentLine[0][1])

        # цикл по сторонам отсекателя

        for i in range(len(points)):

          # определяем нормаль

            normal = QtCore.QPointF()

            if i == len(points) - 1:

                normal.setX(-norm \* (points[0].y() - points[i].y()))

                normal.setY(norm \* (points[0].x() - points[i].x()))

            else:

                normal.setX(-norm \* (points[i + 1].y() - points[i].y()))

                normal.setY(norm \* (points[i + 1].x() - points[i].x()))

# весовой множитель удаленности гранничной точки от р1

            distanceWeight = QtCore.QPointF()

            distanceWeight.setX(currentLine[0][0] - points[i].x())

            distanceWeight.setY(currentLine[0][1] - points[i].y())

            # определяем скалярные произведения

            Dscalar = self.scalar(directrix, normal)

            Wscalar = self.scalar(distanceWeight, normal)

            if Dscalar == 0:

                # если отрезок параллелен ребру отсекателю

                if Wscalar < 0:

                    # виден ли

                    return

            else:

                # отрезок невырожден, определяем t

                t = - Wscalar / Dscalar

                if Dscalar > 0:

                    # поиск нижнего предела

                    if t > 1:

                        return

                    else:

                        tLower = max(tLower, t)

                elif Dscalar < 0:

                    # поиск верхнего предела

                    if t < 0:

                        return

                    else:

                        tTop = min(tTop, t)

            # проверка фактической видимости отрезка

        if tLower <= tTop:

            self.scene.addLine(currentLine[0][0] + (currentLine[1][0] - \

currentLine[0][0]) \* tTop,

                    currentLine[0][1] + (currentLine[1][1] - currentLine[0][1]) \* tTop,

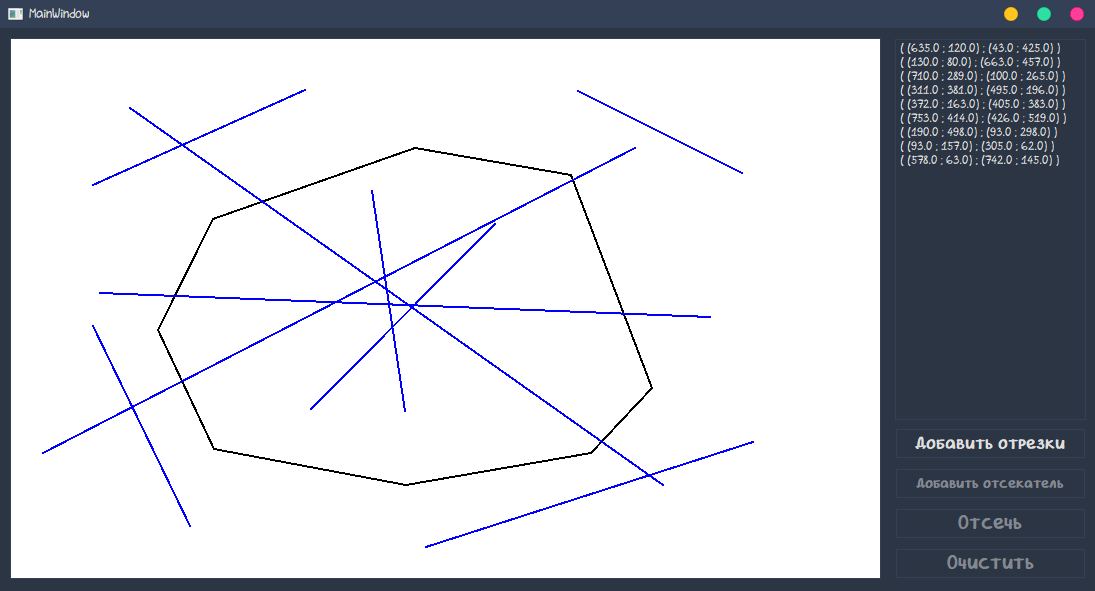
                    currentLine[0][0] + (currentLine[1][0] - currentLine[0][0]) \* tLower,

                    currentLine[0][1] + (currentLine[1][1] - currentLine[0][1]) \* tLower,

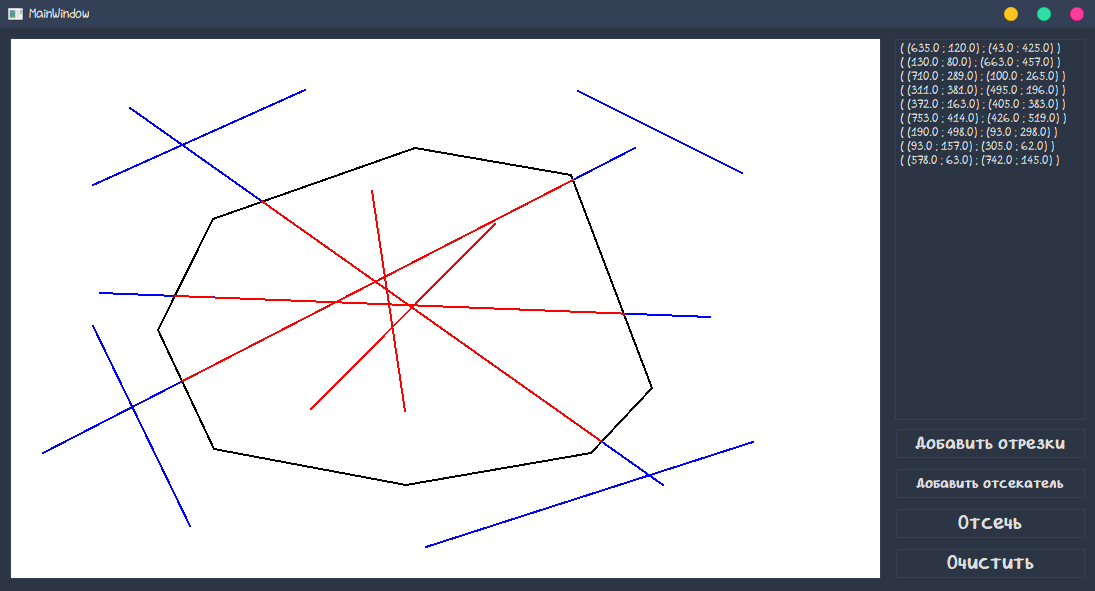
self.pen)

**Примеры работы:**

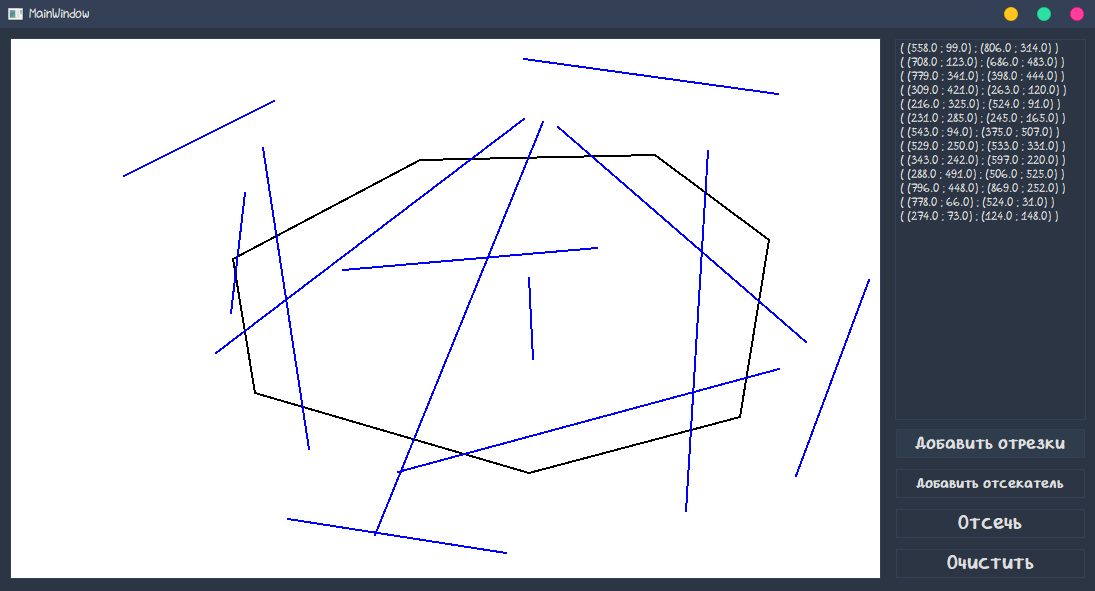
1. **Задаем отрезки и отсекатель:**



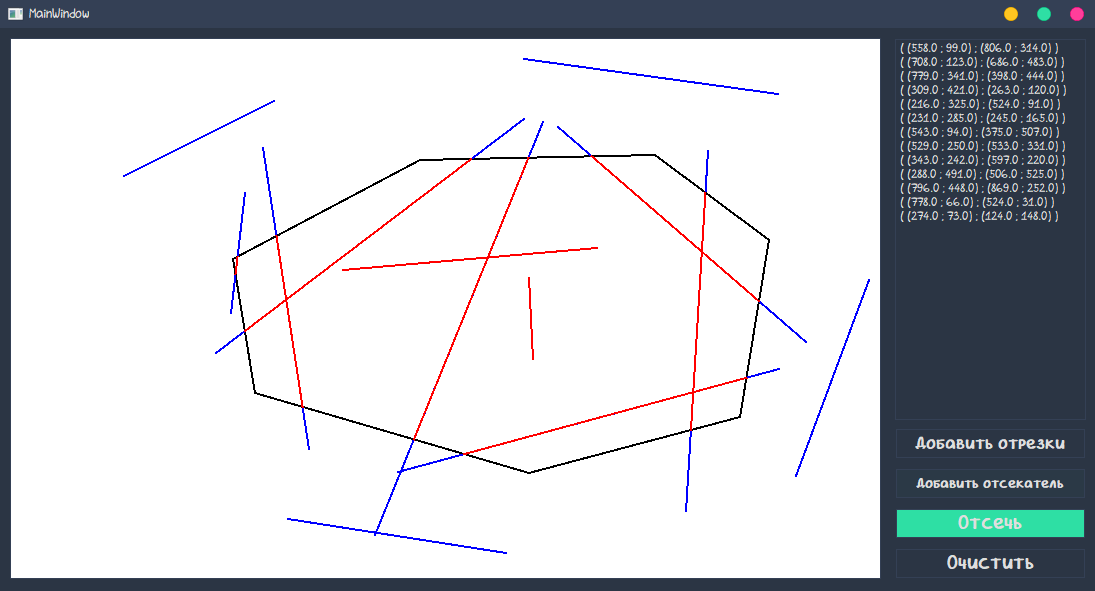
**Результат:**



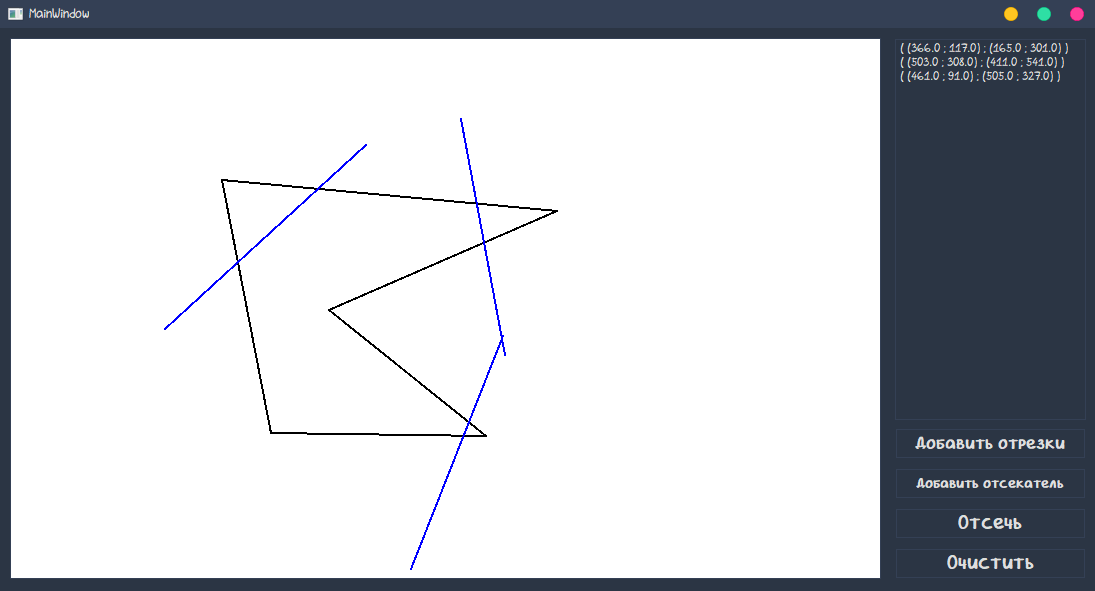
1. **Задаем отрезки и отсекатель:**



**Результат:**



1. **Задаем отрезки и отсекатель(не выпуклый, должна сработать ошибка):**



**Результат:**

