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: Блохин Д.М.

Группа: ИУ7-42Б

Оценка (баллы): _____

Преподаватель: Куров А.В.

Москва.
2020 г.

Цель работы: реализация и исследование одного из алгоритмов растрового заполнения области. Персональное задание 3 вариант – реализация и исследования ***алгоритма заполнения с перегородкой***

При выполнении этой лабораторной работы необходимо:

- Обеспечить ввод произвольной многоугольной области, содержащей произвольное количество отверстий. Ввод (вершин многоугольника) производить с помощью мыши, при этом для удобства пользователя должны отображаться ребра, соединяющие вводимые вершины. Предусмотреть ввод горизонтальных и вертикальных ребер.

- Предоставить пользователю возможность выбора цвета заполнения

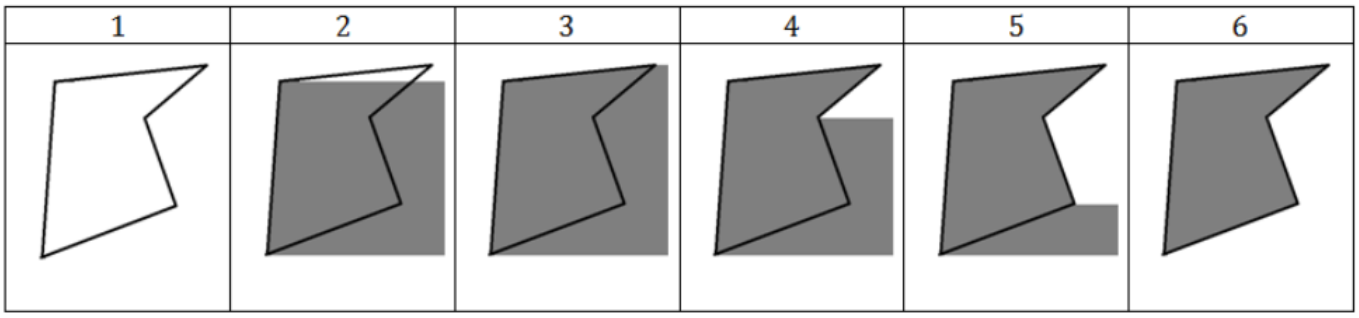
- Предусмотреть два режима – с задержкой (для отслеживания промежуточных состояний выполнения алгоритма) и без задержки.

- Обеспечить замер времени выполнения алгоритма (без задержки, с выводом на экран только окончательного результата).

Теория

Алгоритм растрового заполнения с перегородкой является модификацией ***алгоритма заполнения по ребрам***. Поэтому, думаю, следует описать второй алгоритм вначале.

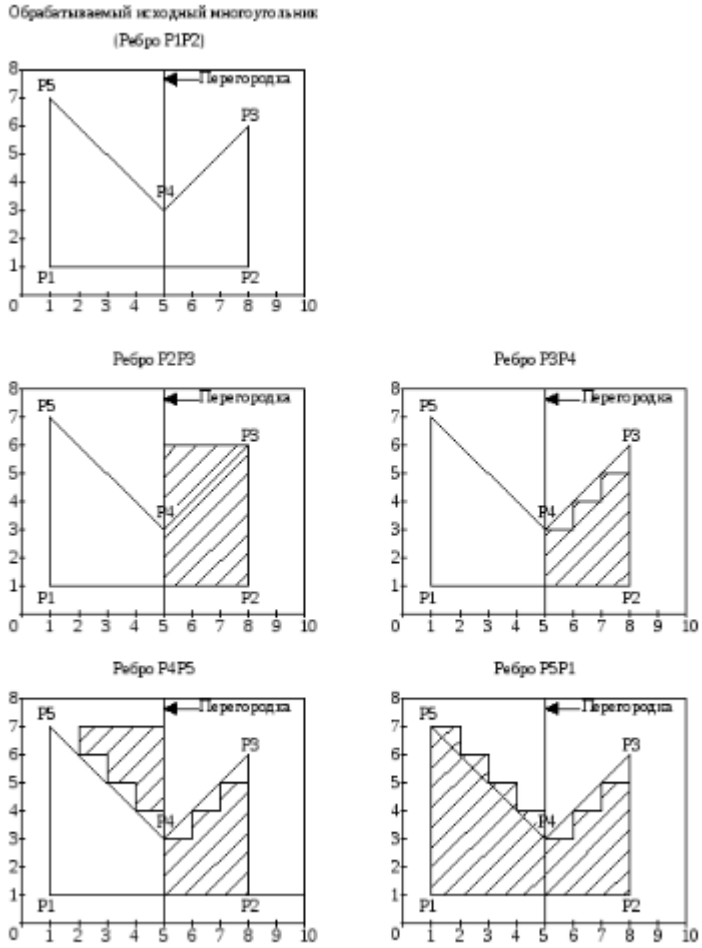
Алгоритм растрового заполнения по ребрам – алгоритм, где для каждой сканирующей строки, пересекающей ребро многоугольника, необходимо дополнить все пиксели, которые лежат правее пересечения строки и ребра многоугольника. Причем данный алгоритм не основывается на порядке обработки ребер: порядок ребер для алгоритма не важен! Пример заполнения по ребрам:



Основной и большой недостаток данного алгоритма – многократная обработка одного и того же пикселя в случае областей закрашивания сложной формы. Учитывая, что для каждого пикселя мы узнаем его цвет, анализируем его, а затем инвертируем и задаем его пикселю, то данная проблема приводит к существенным временным затратам

Алгоритм заполнения по ребрам можно оптимизировать, что позволит частично уменьшить недостаток этого алгоритма: *вводится специальная перегородка*, и для каждой сканирующей строки, пересекающей ребро МУ:
Если пересечение находится слева от перегородки, то перекрасить все пиксели, центры которых лежат справа от пересечения сканирующей строки с ребром и слева от перегородки. Если пересечение находится справа от перегородки, то перекрасить все пиксели, центры которых лежат слева или на пересечении сканирующей строки с ребром и справа от перегородки.

Пример работы алгоритма:



Таким образом, удачно выбрав положение перегородки, можно значительно сократить число обрабатываемых пикселей. Однако отсюда следует проблема: временные затраты алгоритма зависят от положения перегородки, а найти идеальное положение перегородки для сложных фигур весьма сложная задача. И также остается проблема многократной обработки одного и того же пикселя в случае областей закрашивания сложной формы.

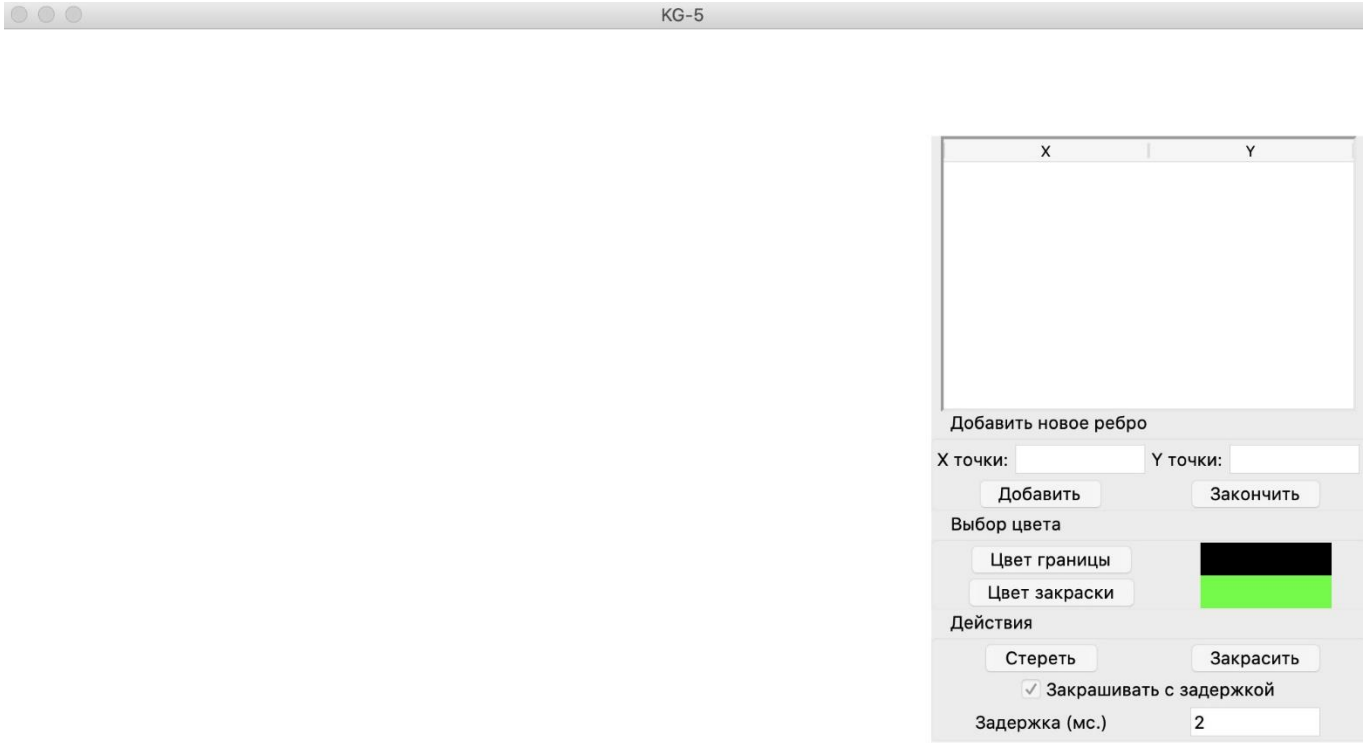
Код:

```
#include "time.h"
void MainWindow::on_pushButton_3_clicked()
{
    auto mas = dots;
    int border = 0;
    using pair_t = std::pair<int, int>;
    double beg = clock();
    // В данный момент в коде представлено положение перегородки по центру фигуры.

    int max = mas[0].first;
    int min = mas[0].first;
    for (const auto& el : mas) {
        if (el.first == -1)
            continue;
        if (el.first > max)
            max = el.first;
        if (el.first < min)
            min = el.first;
    }

    border = (max + minimal) / 2;
    auto pixels = pix->toImage();
    QColor colorAt;
    for (auto i = 0; i < mas.size() - 1; ++i) {
        int dot_1_x = mas[i].first;
        int dot_1_y = mas[i].second;
        int dot_2_x = mas[i + 1].first;
        int dot_2_y = mas[i + 1].second;

        if (dot_1_x == -1 or dot_2_x == -1)
            continue;

        if (dot_1_y == dot_2_y) // обработка горизонтальной линии
            continue;
    int FLAG = 0;
        FLAG = calcEx(mas, i);

        if (dot_2_y < dot_1_y) {
            int temp = dot_2_y;
            dot_2_y = dot_1_y;
            dot_1_y = temp;
            temp = dot_2_x;
            dot_2_x = dot_1_x;
            dot_1_x = temp;
        }

        double dx = (dot_2_x - dot_1_x) / double(dot_2_y - dot_1_y);
        double beg = dot_1_x;
```

```
int temp = FLAG;
        for (auto y = dot_1_y; y < dot_2_y + temp; ++y) {
            if (i > 0)
                if (FLAG == 0) {
                    FLAG = 1;
                    continue;
                }
        for (auto y = dot_1_y; y < dot_2_y; ++y) {
            int x = beg + dx;
            int flag = 0;

            while (abs(x - border)){ //пока не достигли перегородки
                if (x < border) { //если пересечение слева
                    flag = 1;
                    colorAt = pixels.pixelColor(x, y);
                    if (colorAt == bg_color)
                        colorAt = fill_color;
                    else
                        colorAt = bg_color;
                    pixels.setPixelColor(x, y, colorAt);
                    x ++;
                }
                else if (x > border) { //если пересечение справа
                    colorAt = pixels.pixelColor(x, y);
                    if (colorAt == bg_color)
                        colorAt = fill_color;
                    else
                        colorAt = bg_color;
                    pixels.setPixelColor(x, y, colorAt);
                    x --;
                }

                if (flag == 1) {
                    if (x == border) {

                        colorAt = pixels.pixelColor(x, y);
                        if (colorAt == bg_color)
                            colorAt = fill_color;
                        else
                            colorAt = bg_color;
                        pixels.setPixelColor(x, y, colorAt);
                    }
                }
            }

            //условие для задержки
            if (ui->radioButton_2->isChecked()) {
                delay();
                pix->convertFromImage(pixels);
                scene->addPixmap(*pix);
        }

            beg += dx;
        }
    if (!ui->radioButton_2->isChecked()) {
            pix->convertFromImage(pixels);
            scene->addPixmap(*pix);
    }
    }
  draw_edges();

  double end = clock();

  qDebug() << (end - beg) / CLOCKS_PER_SEC;

}
```

Входные данные

При запуске программы пользователю будет представлен графический интерфейс, с помощью которого будет производиться работа с программой. Окно интерфейса выглядит следующим образом:

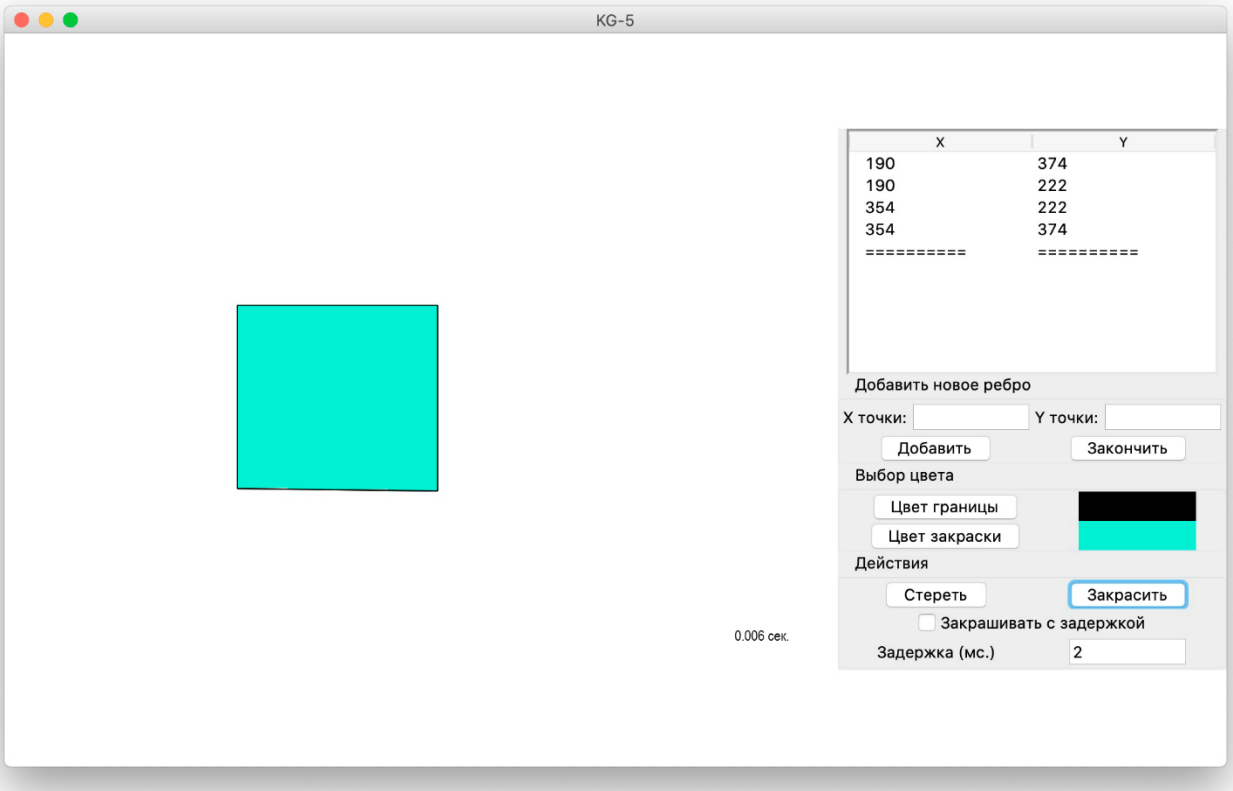


Основной способ ввода координат многоугольника – с помощью мыши, кликая на холст. Однако альтернативным способом добавления координат (для удобства добавления вертикальных и горизонтальных ребер) является ввод координат с клавиатуры.

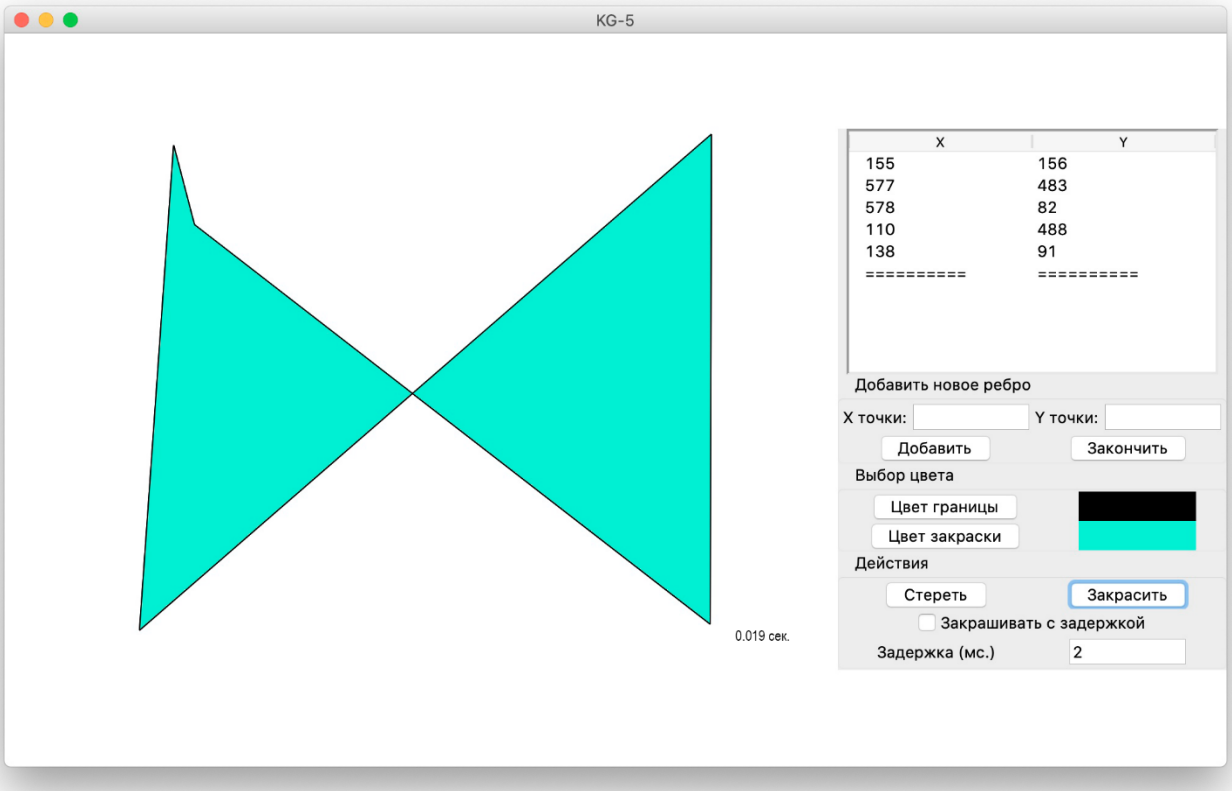
Также предоставляется возможность динамической смены цвета закраски, цвета окантовки и цвета фона. Также можно стереть нарисованную фигуру благодаря кнопке “Стереть”. Чтобы завершить добавление точек и замкнуть фигуру, необходимо нажать кнопку ”Закончить”.

И, наконец, есть выбор, между рисованием с задержкой или без, а также время задержки.

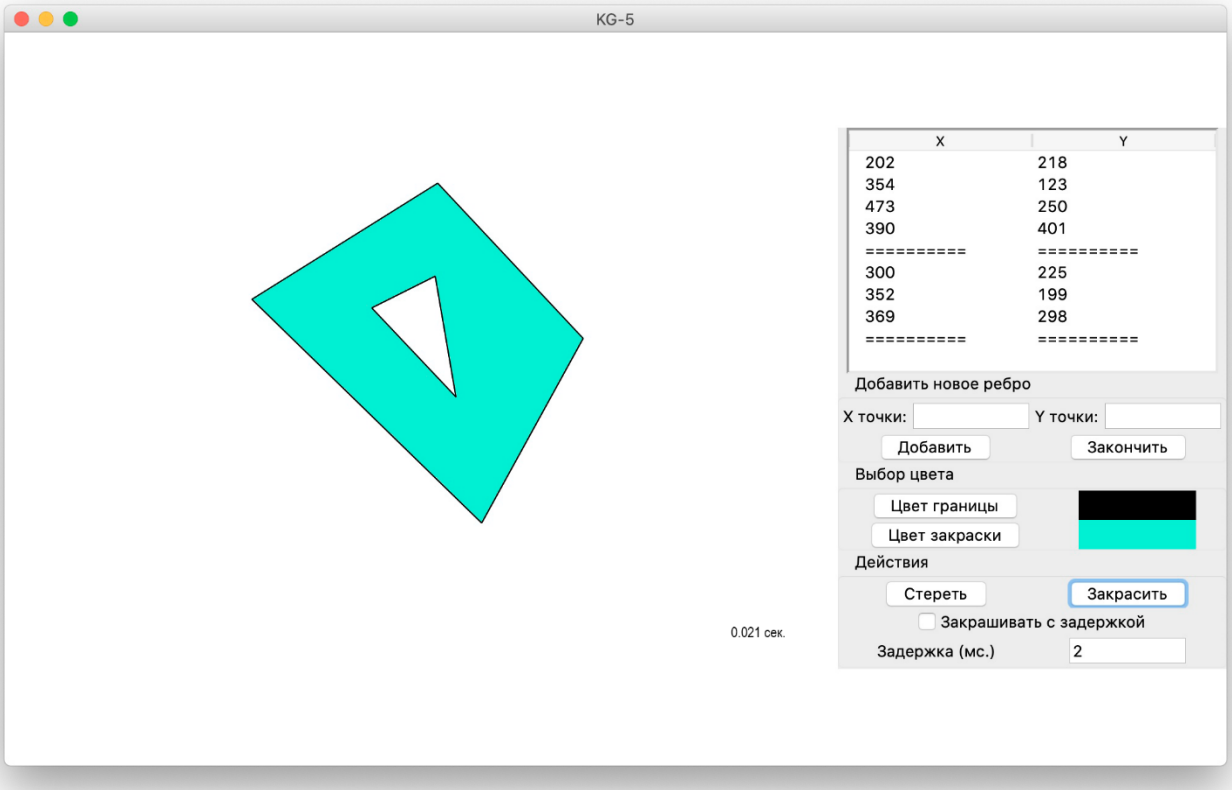
Пример работы



Время заполнения – 0.006 секунд



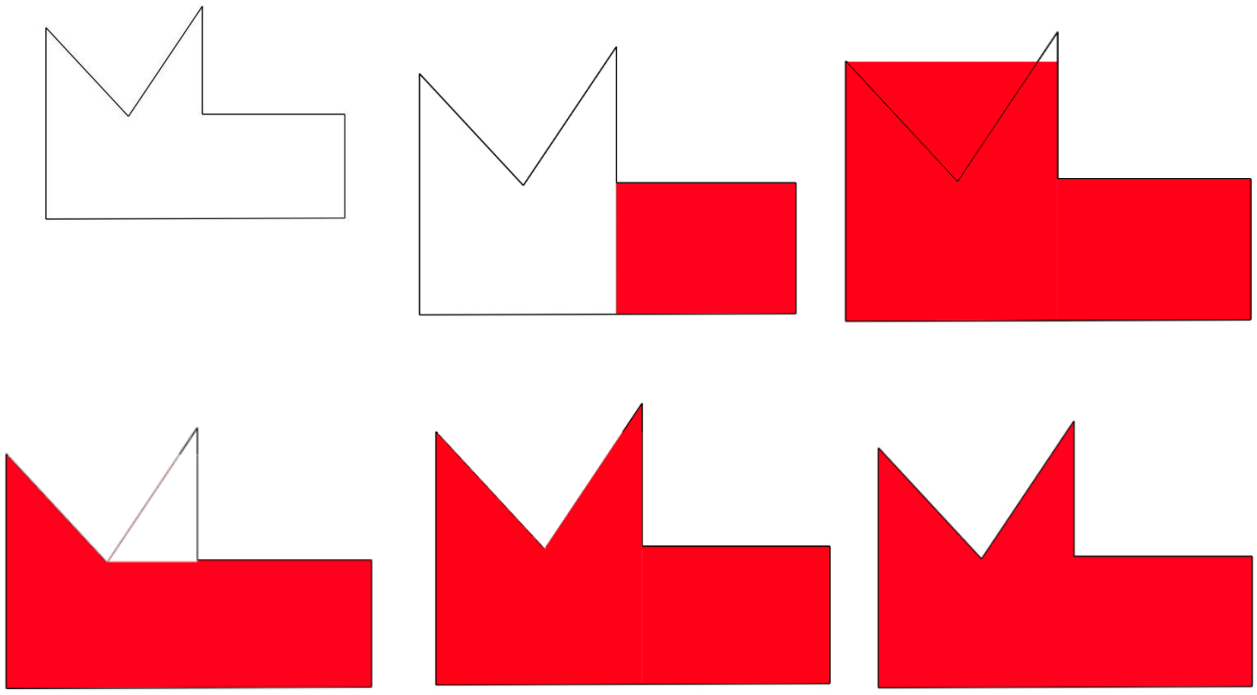
Время заполнения – 0.019 секунд



Время заполнения – 0.021 секунд

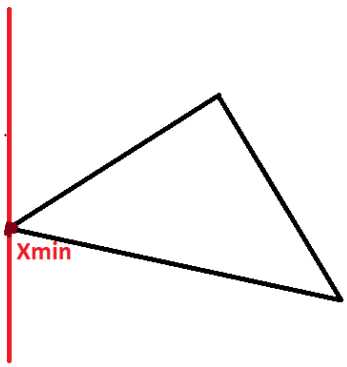
Выполнение закрашивания с задержкой:

Перегородка находится в центре

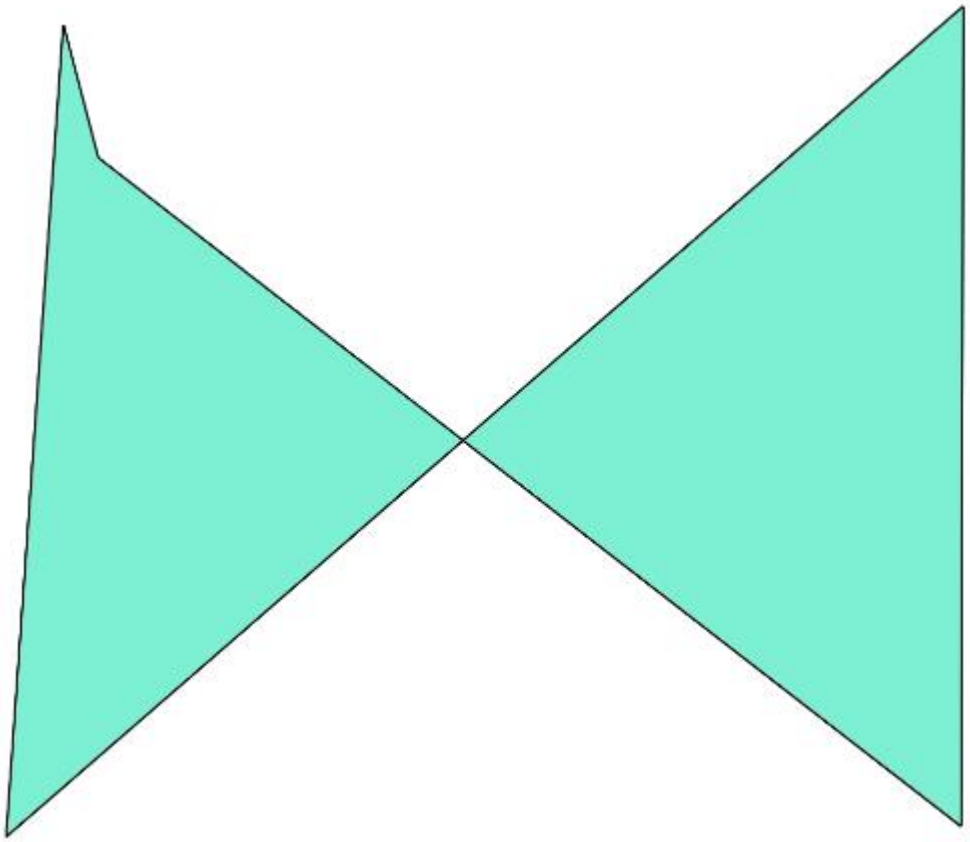


В отличие от других алгоритмов, в данном алгоритме существенную роль занимает положение перегородки относительно фигуры. Строгих правил выбора положения перегородки – нет, но рекомендуется выбирать ее положение так, чтоб она проходила через вершину. Однако все зависит от конкретной фигуры, и гарантировать, что, найдя лучшее положение перегородки для одной фигуры, это положение окажется лучшим для другой – совершенно нельзя.

Рассмотрим положение перегородки, находящейся в крайнем Xmin, то есть с краю - левее всей фигуры



Если замрить время выполнения в первом примере (квадрат) – **0.013**. То есть по сравнению с положением перегородки в центре данное положение проигрывает почти в 2 раза по времени.



Время выполнения – **0.031**. В то время как при положении перегородки в центре, время составляет **0.021**, что на одну сотую быстрее положения перегородки скраю. Исходя из результатов можно сделать вывод

- Положение перегородки ***сильно*** влияет на время выполнения алгоритма. При заполнении простых фигур, разница во времени незначительна , однако с возрастанием сложности фигуры, затрачиваемое время заметно возрастает

Однако, проведя еще ряд замеров и каждый раз меняя позицию перегородки выяснилось, что для более сложных фигур положение перегородки по центру не всегда является идеальным, и поэтому для достижения лучших временных результатов можно находить среднее значения X из всех вершин и ставить перегородку там.

Из всех проедалнных опытов можно сделать выводы:

- Для простых фигур способ выбора перегородки посередине фигуры – весьма простой и достаточно эффективный по времени. Но для сложных фигур, центр далеко не всегда является идеальным положением перегородки

- Способ выбора перегородки по среднему арифм. X из всех вершин хорош как для простых фигур, так и для сложных, так как помогает точнее определить положение перегородки, близкое к идеальному