**УТВЕРЖДАЮ**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.В.Старостин**

**«\_\_\_» \_\_\_\_\_\_\_\_\_ 2022 г.**

**Научно-технический отчет**

**на опытно-конструкторскую работу**

**«Разработка базового функционала библиотеки 2D геометрического ядра»**

**№ 02068143.00221 90**

г. Н. Новгород, 2022

**РЕФЕРАТ**

Рассматривается задача разработки части функционала библиотеки, предназначенной для проверки норм конструктивно-технологических ограничений. Целью работы является разработка программного средства для решения задачи анализа областей на плоскости, возникающей при проектировании печатных плат на этапе проверки соответствия изделия заданным конструктивно-технологическим ограничениям.

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 4](#_Toc126020044)

[1. Основные структуры данных библиотеки 5](#_Toc126020045)

[2. Структура RBTree 5](#_Toc126020046)

[2.1 Постановка задачи 5](#_Toc126020047)

[2.2 Реализация структуры 6](#_Toc126020048)

[2.3 Производительность RBTree 6](#_Toc126020049)

[3. Алгоритм поиска пересечений двух отрезков 6](#_Toc126020050)

[3.1 Постановка задачи 6](#_Toc126020051)

[3.2 Описание подхода 6](#_Toc126020052)

[3.3 Программные интерфейсы, входные и выходные структуры данных 6](#_Toc126020053)

[4. Алгоритм поиска пересечений множества отрезков 7](#_Toc126020054)

[4.1 Описание подхода 7](#_Toc126020055)

[4.2 Программные интерфейсы, входные и выходные структуры данных 8](#_Toc126020056)

[4.3 Тестирование алгоритма 8](#_Toc126020057)

[4.4 Тестирование производительности 9](#_Toc126020058)

[5. Алгоритмы преобразования контейнеров 10](#_Toc126020059)

[5.1 Постановка задачи 10](#_Toc126020060)

[5.1 Описание подхода 10](#_Toc126020061)

[5.2 Программные интерфейсы, входные и выходные структуры данных 10](#_Toc126020062)

[5.3 Тестирование алгоритмов 10](#_Toc126020063)

[6. Цепочка операций над слоями и наборами отрезков и контуров 11](#_Toc126020064)

[6.1 Постановка задачи 11](#_Toc126020065)

[6.2 Описание подхода 12](#_Toc126020066)

[6.3 Реализация подхода 13](#_Toc126020067)

[6.3 Программные интерфейсы, входные и выходные структуры данных 14](#_Toc126020068)

[6.4 Тестирование алгоритма 14](#_Toc126020069)

[6.5 Тестирование производительности 14](#_Toc126020070)

[7. Демонстрационный пример 16](#_Toc126020071)

[ЗАКЛЮЧЕНИЕ 19](#_Toc126020072)

[СПИСОК ССЫЛОЧНЫХ ДОКУМЕНТОВ И ЭЛЕКТРОННЫХ ИСТОЧНИКОВ 20](#_Toc126020073)

[ПРИЛОЖЕНИЕ 21](#_Toc126020074)

### ВВЕДЕНИЕ

Рассматривается задача разработки программного обеспечения для решения задачи анализа областей на плоскости.

В рамках данной работы была поставлена задача разработать программную библиотеку, содержащую следующие структуры и алгоритмы:

1. Структуры данных «Отрезок», «Точка».
2. Контейнеры:
   * контур;
   * набор отрезков;
   * набор контуров;
   * слой отрезков;
   * слой контуров.
3. Структура, выполняющая операции:
   * добавление элемента;
   * поиск минимального/максимального элемента;
   * удаление элемента;
   * поиск следующего элемента для заданного;
   * поиск предыдущего элемента для заданного;
   * доступ к элементу структуры на заданной позиции.
4. Алгоритмы поиска пересечений двух отрезков и множества отрезков.
5. Алгоритмы преобразования контейнеров.
6. Цепочка операций над слоями и наборами отрезков и контуров.

Помимо разработки библиотеки, стояла задача реализовать один пример, демонстрирующий работу цепочки операций.

# 1. Основные структуры данных библиотеки

Основными структурами данных библиотеки «gkernel2d» являются Point и Segment, которые описывают геометрические объекты точка и отрезок на плоскости, а также перечень контейнеров: контур, набор отрезков, набор контуров, слой отрезков, слой контуров. Соответствие структур данных с их названиями в библиотеке представлено в таблице 1.

Таблица 1. Соответствие структур данных с их названиями в «gkernel2d».

|  |  |
| --- | --- |
| **Структура данных** | **Название в «gkernel2d»** |
| Точка | Point |
| Отрезок | Segment |
| Контур | Circuit |
| Набор отрезков | SegmentsSet |
| Набор контуров | CircuitsSet |
| Слой отрезков | SegmentsLayer |
| Слой контуров | CircuitsLayer |

Данные структуры реализуют функционал, описанный в техническом задании к опытно-конструкторской работе «Разработка базового функционала библиотеки 2D геометрического ядра» [1] в пунктах 5.1-5.3, 5.3.1-5.3.6. Интерфейсы представлены в руководстве программиста к библиотеке «gkernel2d» [2].

# 2. Структура RBTree

## 2.1 Постановка задачи

Необходимо реализовать структуру данных, в которой следующие операции имеют сложность не выше O(n\*logn):

* добавление элемента;
* поиск минимального/максимального элемента;
* удаление элемента;
* поиск следующего элемента для заданного;
* поиск предыдущего элемента для заданного;
* доступ к элементу структуры на заданной позиции.

## 2.2 Реализация структуры

Данная структура реализована в виде обёртки над контейнером std::set библиотеки STL. Структура в библиотеке «gkernel2d» представлена в виде шаблонного класса RBTree.

## 2.3 Производительность RBTree

Операции вставки, поиска и удаления в RBTree осуществляют вызов аналогичных функций в std::set, которые имеют логарифмическую сложность [4], поэтому операции RBTree также имеют логарифмическую сложность, что соответствует ограничениям, предъявляемым к структуре.

# 3. Алгоритм поиска пересечений двух отрезков

## 3.1 Постановка задачи

Необходимо реализовать функцию поиска точки пересечения двух отрезков.

## 3.2 Описание подхода

Реализован алгоритм, использующий ориентированную площадь треугольника [3].

## 3.3 Программные интерфейсы, входные и выходные структуры данных

Алгоритм реализован в функции Intersection::intersectSegments, которая имеет следующий вид:

static Point intersectSegments(const Segment& first, const Segment& second)

Аргументами функции являются: first, second — отрезки, для которых ищется общая точка.

В результате функция возвращает объект Point, содержащий координаты точки пересечения отрезков.

**3.4** Тестирование алгоритма

Тестирование проводится с помощью запуска автоматических тестов. Каждый автоматический тест создаёт два объекта класса Segment и передаёт их функции Intersection::intersectSegments. Полученный результат сравнивается с ожидаемым. Набор валидационных тестов представлен в исходных кодах программы.

# 4. Алгоритм поиска пересечений множества отрезков

Необходимо реализовать функцию поиска точек пересечений множества отрезков.

## 4.1 Описание подхода

Алгоритм основывается на методе заметающей прямой. Данный алгоритм использует структуру RBTree.

Вход: набор отрезков

**Шаг 1.** Формирование очереди событий. Событиями являются: начало отрезка, конец отрезка, вертикальный отрезок, пересечение справа, пересечение слева. При первом построении очереди могут встречаться только начало отрезка (концевая точка отрезка с наименьшей координатой x), конец отрезка (концевая точка отрезка с наибольшей координатой x) и вертикальный отрезок. События сортируются по возрастанию координаты x.

**Шаг 2.** Итерация по событиям в очереди.

**Если событие является пересечением справа,** то перестраиваем RBTree. Удаляем отрезки из RBTree, для которых будет нарушен порядок при перемещении заметающей прямой. Эти же отрезки добавляются во временный буфер. Перемещаем заметающую прямую на координату x текущего события. Возвращаем отрезки из временного буфера обратно в RBTree.

**Если событие является вертикальным отрезком,** то просматриваем все отрезки, которые лежат в RBTree (отрезки, для которых ещё не достигнут конец) и ищем пересечения с вертикальным отрезком. Найденные пересечения добавляются в результирующий список. Переход к следующему событию.

**Если событие не является вертикальным отрезком**, то добавляем отрезок, которому принадлежит текущее событие в RBTree. Ищем точку пересечения текущего отрезка с его предшественником в RBTree и со следующим после него. Для каждого пересечения создаются новые события, чуть левее точки пересечения и чуть правее, которые добавляются в очередь. Точки пересечения добавляются в результирующий список.

**Если текущее событие является концом отрезка**, то проверяем предыдущий и следующий отрезок в RBTree на пересечение. Если они пересекаются, то точка пересечения добавляется в результирующий список, а новые чуть левее точки пересечения и чуть правее добавляются в очередь. Удаляем отрезок из RBTree. Переход к следующему событию.

Выход: список точек пересечения множества отрезков.

## 4.2 Программные интерфейсы, входные и выходные структуры данных

Алгоритм реализован в функции Intersection::intersectSetSegments, которая имеет следующий вид:

std::vector<IntersectionPoint> Intersection::intersectSetSegments(const SegmentsSet& segments).

Аргументом функции является объект SegmentsSet (набор отрезков), на этом наборе будет осуществляться поиск.

Результатом работы алгоритма является вектор структур IntersectionPoint.

struct IntersectionPoint {

Point \_point;

segment\_id \_first\_segment\_id;

segment\_id \_second\_segment\_id;

}

Структура IntersectionPoint содержит следующие поля:

* point — точка пересечения отрезков;
* first\_segment\_id — id первого пересекаемого отрезка в segments;
* second\_segment\_id — id второго пересекаемого отрезка в segments.

first\_segment\_id, second\_segment\_id имеют тип segment\_id — беззнаковый int64.

## 4.3 Тестирование алгоритма

Тестирование проводится с помощью запуска автоматических тестов. Каждый автоматический тест создаёт объект SegmensSet и передаёт его на вход intersectSetSegments. Результат работы функции сравнивается со списком ожидаемых точек пересечения. Набор валидационных тестов представлен в исходных кодах программы.

## 4.4 Тестирование производительности

В таблице 2 приведены результаты тестов производительности алгоритма. Тестирование проводилось на сгенерированных задачах, принимающих в качестве аргумента число отрезков. На графике 1 отображена зависимость времени выполнения от количества точек пересечения.

Таблица 2. Результаты тестов производительности

|  |  |  |  |
| --- | --- | --- | --- |
| **№ теста** | **Кол-во отрезков** | **Кол-во пересечений** | **Время, мс** |
| 1 | 100 | 0 | 0.045 |
| 2 | 1000 | 25 | 0.579 |
| 3 | 3000 | 124 | 2.00 |
| 4 | 5000 | 315 | 3.64 |
| 5 | 10 000 | 787 | 8.30 |
| 6 | 20 000 | 3248 | 20.0 |
| 7 | 100 000 | 75871 | 194 |
| 8 | 250 000 | 483311 | 846 |

График 1. Зависимость времени выполнения от количества точек пересечения

# 5. Алгоритмы преобразования контейнеров

## 5.1 Постановка задачи

Необходимо реализовать следующие функции преобразования контейнеров:

1. Из набора отрезков в слой отрезков.
2. Из слоя отрезков в набор отрезков.
3. Из слоя контуров в набор контуров.

## 5.1 Описание подхода

*1. Из набора отрезков в слой отрезков*

Осуществляется поиск точек пересечений входного набора отрезков с помощью алгоритма поиска пересечений множества отрезков. Отрезки разбиваются в точках пересечения с сохранением меток исходных отрезков.

*2. Из слоя отрезков в набор отрезков*

Слой отрезков уже является набором отрезков. Дополнительные преобразования не требуются.

*3. Из слоя контуров в набор контуров*

Слой контуров уже является набором контуров. Дополнительные преобразования не требуются.

## 5.2 Программные интерфейсы, входные и выходные структуры данных

*1. Из набора отрезков в слой отрезков*

static SegmentsLayer convertToSegmentsLayer(const SegmentsSet& segments)

*2. Из слоя отрезков в набор отрезков*

static SegmentsSet convertToSegmentsSet(const SegmentsLayer& segments)

*3. Из слоя контуров в набор контуров*

static CircuitsSet convertToCircuitsSet(const CircuitsLayer& circuits)

Программные интерфейсы, входные и выходные данные описаны в руководстве программиста.

## 5.3 Тестирование алгоритмов

Тестирование проводится с помощью запуска автоматических тестов. Набор валидационных тестов представлен в исходных кодах программы.

# 6. Цепочка операций над слоями и наборами отрезков и контуров

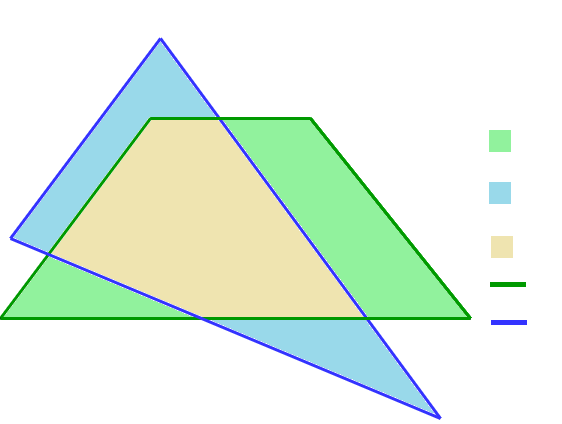
## 6.1 Постановка задачи

На плоскости заданы слои контуров. Каждый контур ограничивает часть плоскости, образуя внутреннюю область контура. Эти области могут накладываться на внутренние области других контуров. Таким образом, плоскость разбивается на множество областей, которые характеризуются количеством наложенных контуров и сочетанием слоёв.

В примере, изображённом на рисунке 1, заданы два слоя контуров, каждый состоит из одного контура: слой 1 — из зелёного, слой 2 — из синего. Плоскость состоит из участков, которые не принадлежат ни одному контуру, которые ограничены контуром только из слоя 1, которые ограничены контуром только из слоя 2 и где пересекаются слой 1 и слой 2.

Необходимо реализовать функционал, позволяющий выделить области, удовлетворяющие заданной логической функции над слоями. Функция задаётся пользователем и может включать логические операторы «И», «ИЛИ», «НЕ». В примере на рисунке 1 результатом функции «слой 1 И слой 2» является светло-жёлтая область, а результатом «слой 1 И НЕ слой 2» являются зелёные области.

Рисунок 1. Разметка областей, полученные пересечением контуров из разных слоёв.



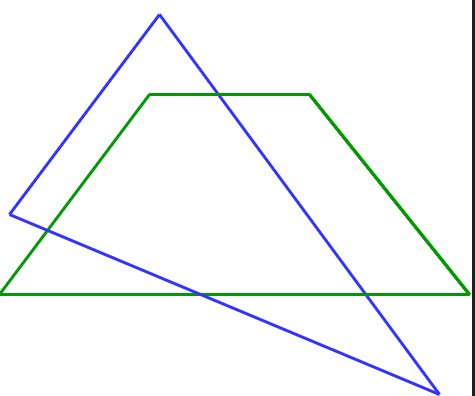
слой 1

слой 2

область, ограниченная слоем 1 и 2

область, ограниченная слоем 2

область, ограниченная слоем 1



**слой 1**

**слой 2**

**слой 1**

**слой 2**

## 6.2 Описание подхода

Идея основывается на том, что ребро каждого контура находится на границе двух областей. Установка маркеров для ребра, указывающие какие области оно разделяет позволит построить требуемый результат путём селекции рёбер по маркерам. Логическая функция над слоями преобразуется в логическую функцию над маркерами.

## 6.3 Реализация подхода

Алгоритм

Вход: 2 слоя контуров.

1. Входные слои контуров передаются на вход функции Converter::mergeCircuitsLayers, где отрезки слоёв сливаются в один набор и помечаются маркерами, указывающие какому слою они принадлежали. Результатом слияния является контейнер SegmentsSet (набор отрезков).
2. Результат функции слияния передаётся пользователем в функцию конвертации из набора отрезков в слой отрезков Converter::convertToSegmentsLayer. Результат содержит отрезки из исходных слоёв, разбитые на несколько частей в точках пересечения, где каждая часть сохраняет метку исходного ребра.
3. Результат конвертации и функция селекции по меткам передаётся пользователем в функцию markAreasAndFilter. Внутри функции происходит маркировка рёбер новым набором меток:

* метка 0 – область над данным ребром ограничена контуром слоя 1;
* метка 1 – область над данным ребром ограничена контуром слоя 2;
* метка 2 – область под данным ребром ограничена контуром слоя 1;
* метка 3 – область под данным ребром ограничена контуром слоя 2.

Метки принимают значения 0 или 1: 0 – область не ограничена сверху/снизу, 1 – область ограничена сверху/снизу;

Функция селекции по меткам состоит из логических операторов «И», «ИЛИ» для значений четырёх меток. markAreasAndFilter выбирает отрезки, чьи метки удовлетворяют переданной логической функции и добавляет их в результирующий слой отрезков. Результат возвращается пользователю.

Выход: SegmentsLayer, состоящий из отрезков, удовлетворяющих логической функции.

## 6.3 Программные интерфейсы, входные и выходные структуры данных

Программные интерфейсы используемых структур и алгоритмов представлены в руководстве программиста.

## 6.4 Тестирование алгоритма

Тестирование проводится с помощью запуска автоматических тестов. Набор валидационных тестов представлен в исходных кодах программы.

## 6.5 Тестирование производительности

В таблице 3 приведены результаты тестов производительности алгоритма. Общее время складывалось из времени выполнения шагов 2 и 3. Тестирование проводилось на сгенерированных задачах, принимающих в качестве аргумента число отрезков. На графике 2 отображена зависимость времени выполнения от количества точек пересечения.

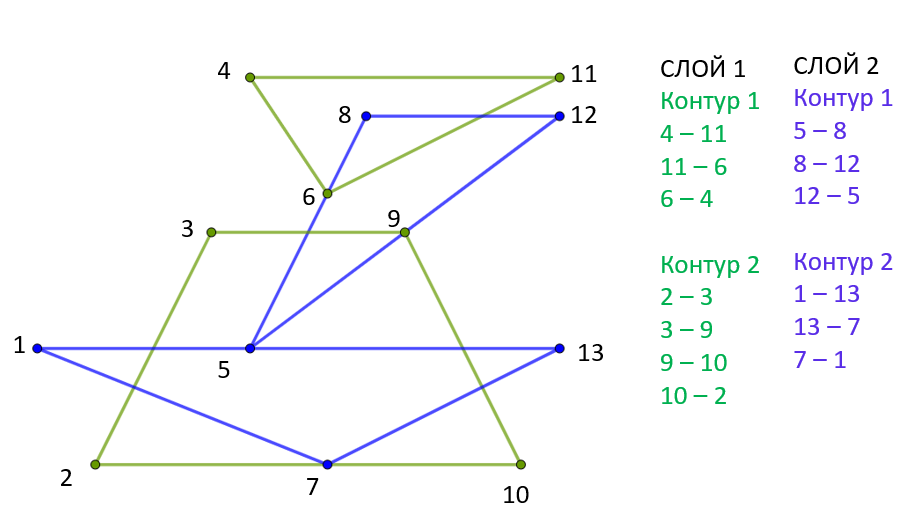
Таблица 3. Результаты тестов производительности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ теста** | **Кол-во отрезков** | **Кол-во пересечений** | **Кол-во отрезков после шага 2** | **Время, мс** |
| 1 | 100 | 0 | 100 | 0.038 |
| 2 | 1000 | 0 | 1000 | 0.708 |
| 3 | 3000 | 214 | 3042 | 2.75 |
| 4 | 5000 | 517 | 5132 | 5.45 |
| 5 | 10 000 | 1647 | 10 410 | 13.3 |
| 6 | 20 000 | 6272 | 21 590 | 35.2 |
| 7 | 100 000 | 170 709 | 143 064 | 649 |
| 8 | 250 000 | 1 064 000 | 519 325 | 6146 |

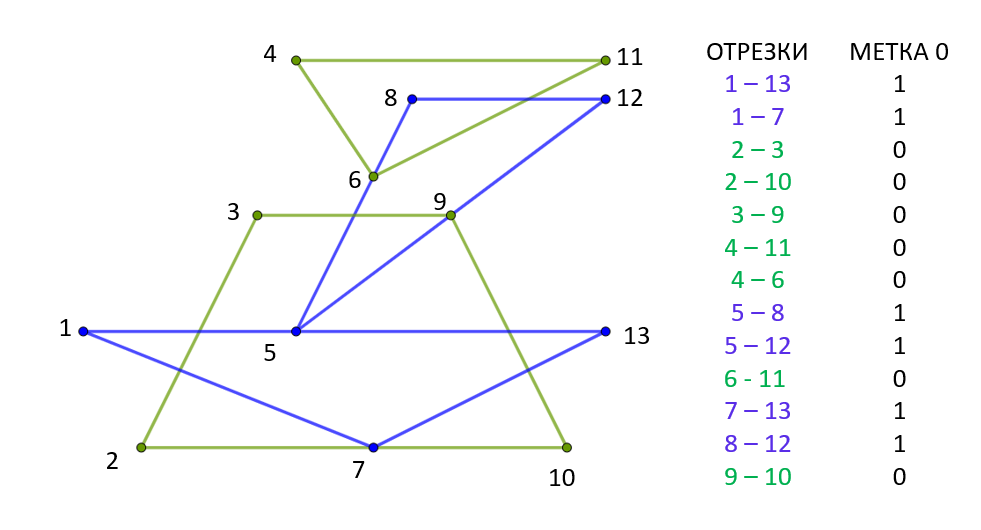
График 2. Зависимость времени выполнения от количества точек пересечения

# 7. Демонстрационный пример

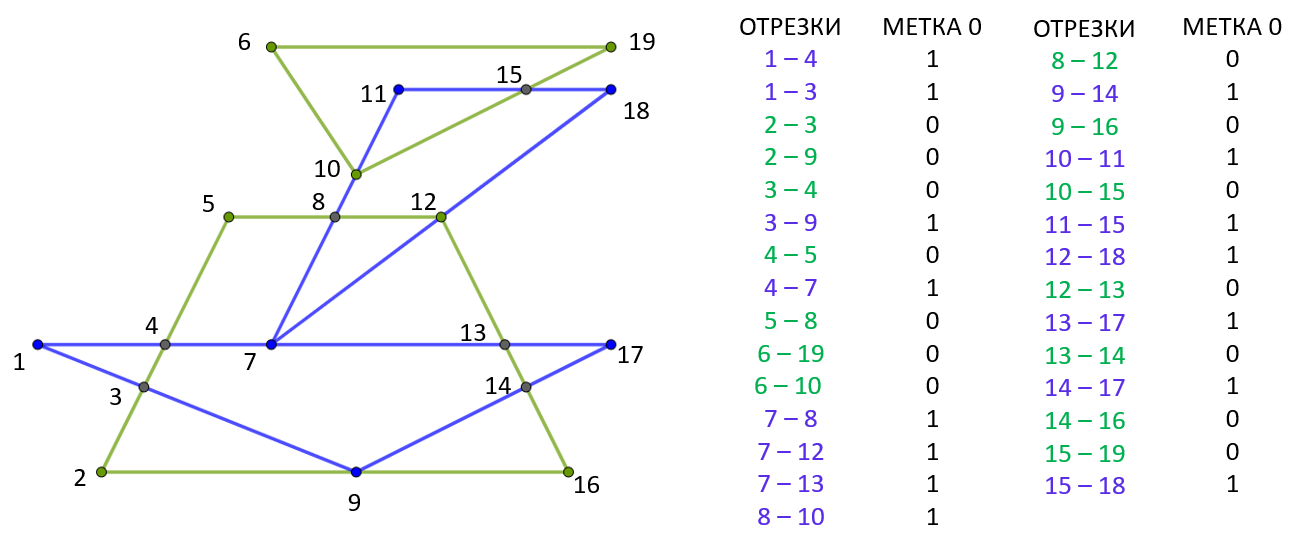
Для наглядной демонстрации работы цепочки операций был разработан пример клиентского кода. Код примера содержится в разделе «Приложение». Код осуществляет построение слоя контуров, который является результатом логической функции «И» над слоями отрезков. Рисунки ниже показывают преобразования входных данных в процессе прохождения каждого этапа алгоритма.



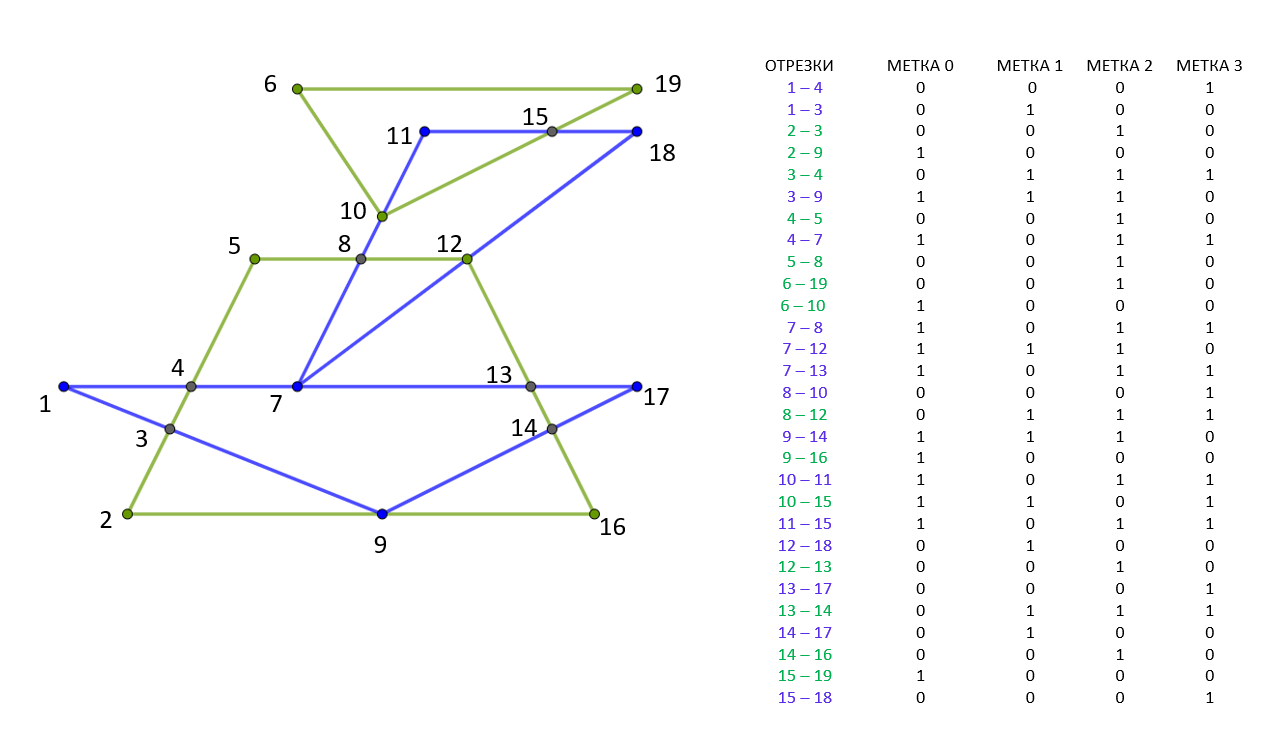
Входные данные



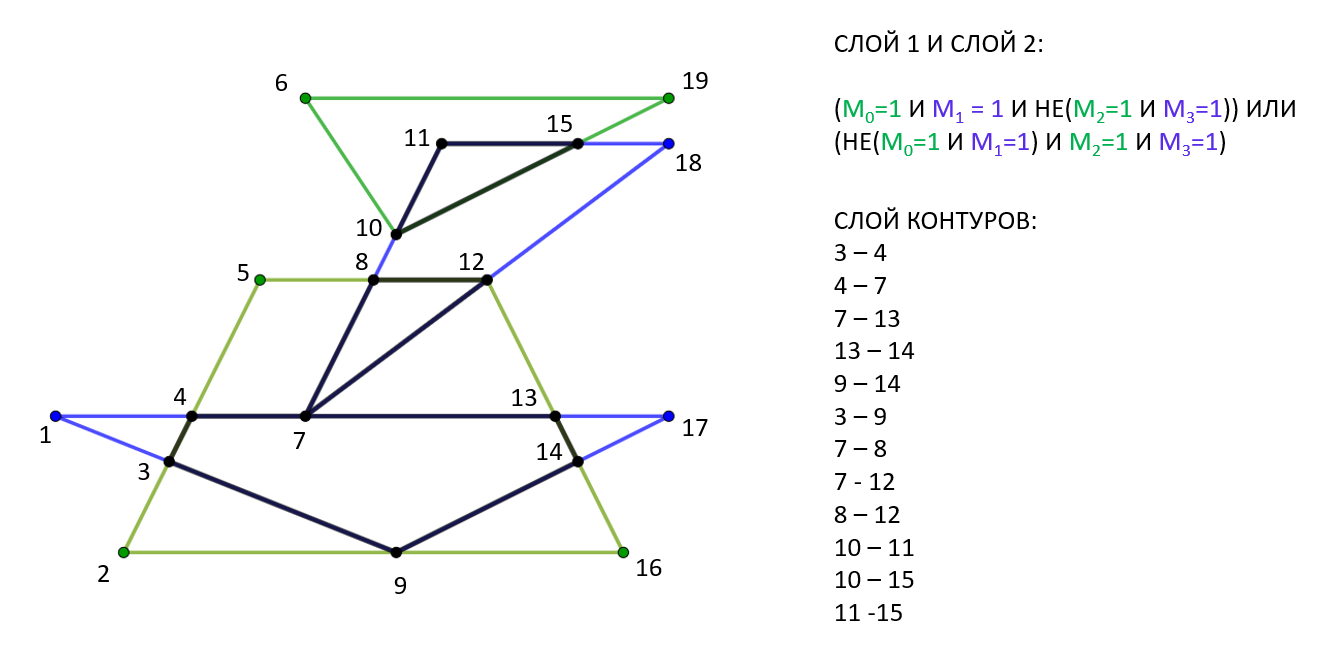
Шаг 1. Слияние слоёв и расстановка меток слоёв



Шаг 2. Конвертация из набора отрезков в слой отрезков



Шаг 3. Маркировка отрезков



Отбор отрезков

### ЗАКЛЮЧЕНИЕ

В результате проведенной работы была разработана библиотека, содержащая все требуемые структуры и алгоритмы, описанные в техническом задании, определены программные интерфейсы библиотеки, создана инфраструктура автоматического unit-тестирования и тестирования производительности, составлена методика испытаний библиотеки и руководство программиста. Результаты проведённых тестов производительности соответствуют всем заявленным требованиям, предъявляемым к алгоритмам. Также был разработан демонстрационный пример клиентского кода, показывающий последовательность шагов для запуска алгоритма анализа областей.

### СПИСОК ССЫЛОЧНЫХ ДОКУМЕНТОВ И ЭЛЕКТРОННЫХ ИСТОЧНИКОВ

1. Техническое задание на опытно-конструкторскую работу «Разработка базового функционала библиотеки 2D геометрического ядра».
2. Руководство программиста библиотеки gkernel2d.
3. Алгоритм поиска точки пересечения двух прямых: https://e-maxx.ru/algo/segments\_intersection\_checking
4. Сложность операций в std::set: https://en.cppreference.com/w/cpp/container/set

### ПРИЛОЖЕНИЕ

Клиентский код демонстрационного примера.

#include "gkernel/objects.hpp"

#include "gkernel/containers.hpp"

#include "gkernel/converter.hpp"

#include "gkernel/area\_analyzer.hpp"

using namespace gkernel;

int main() {

// конструирование контуров

Circuit first\_circuit = {{

{{8, 13}, {16, 13}},

{{16, 13}, {10, 10}},

{{10, 10}, {8, 13}}

}};

Circuit second\_circuit = {{

{{8, 6}, {11, 12}},

{{11, 12}, {16, 12}},

{{16, 12}, {8, 6}}

}};

Circuit third\_circuit = {{

{{4, 3}, {7, 9}},

{{7, 9}, {12, 9}},

{{12, 9}, {15, 3}},

{{15, 3}, {4, 3}}

}};

Circuit fourth\_circuit = {{

{{2.5, 6}, {16, 6}},

{{16, 6}, {10, 3}},

{{10, 3}, {2.5, 6}}

}};

// конструирование слоёв контуров

CircuitsLayer first\_layer = {{ first\_circuit, third\_circuit }};

CircuitsLayer second\_layer = {{ second\_circuit, fourth\_circuit }};

// слияние отрезков из слоёв в один набор отрезков

auto merged\_layers = Converter::mergeCircuitsLayers(first\_layer, second\_layer);

// конвертация набора отрезков в слой отрезков

auto segments\_layer = Converter::convertToSegmentsLayer(merged\_layers);

// запуск логической функции «И» над слоями

SegmentsLayer filtered = AreaAnalyzer::markAreasAndFilter(segments\_layer, [](const SegmentsLayer& segments, const Segment& segment) {

return segments.get\_label\_value(0, segment) == 1 && segments.get\_label\_value(1, segment) == 1 &&

!(segments.get\_label\_value(2, segment) == 1 && segments.get\_label\_value(3, segment) == 1) ||

!(segments.get\_label\_value(0, segment) == 1 && segments.get\_label\_value(1, segment) == 1) &&

segments.get\_label\_value(2, segment) == 1 && segments.get\_label\_value(3, segment) == 1;

});

// вывод результата

for (std::size\_t idx = 0; idx < filtered.size(); ++idx) {

std::cout << filtered[idx] << std::endl;

}

}