Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»**
(МИ ВлГУ)

Факультет ИТР

Кафедра ИС

УТВЕРЖДАЮ

Заведующий кафедрой

_____ Д.Е. Андрианов
(подпись)

«_____» _____ 2022 г.

БАКАЛАВРСКАЯ РАБОТА

Тема Разработка и исследование алгоритма для комплексирования векторных данных с сохранением топологии

МИВУ.09.03.02-02.000 БР

Руководитель

Еремеев С.В.
(фамилия, инициалы)

_____
(подпись) (дата)

Студент ИС-118
(группа)

Кашин Н.П.
(фамилия, инициалы)

_____
(подпись) (дата)

Муром 2022

Бланк задания

В данной бакалаврской работе выполнены реализация и иследование алгоритма комплексирования векторных данных с сохранением топологии в среде QGIS на языке PyQGIS. Произведён поиск и анализ материалов по теме работы. Описана реализация основных функций разработанного алгоритма.

Табл. 2. Ил. 82. Библ. 8.

In this bachelor's work, the implementation and study of the algorithm for integrating vector data with the preservation of topologies in the QGIS environment in the PyQGIS language are carried out. The search and analysis of materials on the topic of the work was carried out. The implementation of the main functions of the developed algorithm is described.

Tabl. 2. Fig. 82. Bibl. 8.

Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»**
(МИ ВлГУ)

Факультет ИТР

Кафедра ИС

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Тема: Разработка и исследование алгоритма для комплексирования векторных данных с сохранением топологии

МИВУ.09.03.02-02.000 ПЗ

Муром 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Во второй половине XX века появилось множество инновационных технологий, которые повлекли развитие методов пространственной оценки. Достижения в области компьютерных технологий, информатики и компьютерной графики, опыт топографического и тематического картографирования, а также успешные попытки автоматизирования процесса создания и изменения карт, привели к развитию географических информационных систем.

Геоинформационная система (географическая информационная система, ГИС) — это система для сбора, хранения, анализа и графической визуализации пространственных (географических) данных и связанной с ними информации о необходимых объектах.

Понятие геоинформационной системы используется также в более узком смысле — как инструмент (программный продукт), позволяющий искать, анализировать и редактировать как цифровую карту местности, так и дополнительную информацию об объектах.

Геоинформационная система может включать в себя пространственные базы данных, редакторы растровой и векторной графики, а также различные средства анализа пространственных данных. Они используются в картографии, геологии, метеорологии, землеустройстве, экологии, муниципальном управлении, транспорте, экономике, обороне и многих других областях.

Топология описывает пространственные отношения между соединенными или прилегающими векторными объектами (точками, линия и полигонами) в ГИС. При комплексировании векторных данных теряются топологические связи между объектами, находящимися на карте.

В связи с этим возникает актуальная задача разработки алгоритма комплексирования векторных данных. Решению этой задачи и посвящена бакалаврская работа.

1 Анализ технического задания

В данной бакалаврской работе была поставлена задача разработать и исследовать алгоритм комплексирования векторных данных.

Исходными данными для работоспособности алгоритма являются две карты одной и той же местности, но сделанные в разный период времени и/или с разным масштабом.

Под комплексированием понимается объединение, сочетание, создание комплекса или комплексов.

Алгоритм, разработанный в ходе исследовательской работы, будет объединять некоторый набор векторных данных на одной карте с другой картой. То есть копировать набор данных с одной карты на другую. После комплексирования векторных данных алгоритм проведёт анализ топологических отношений и исправления данных, в случае их неправильного расположения.

В качестве среды разработки будет использоваться географическая информационная система QGIS.

1.1 Географическая информационная система QGIS

QGIS — это бесплатная кроссплатформенная географическая информационная система (ГИС) с открытым исходным кодом, которая поддерживает просмотр, редактирование, печать и анализ геопространственных данных. QGIS позволяет пользователям анализировать и редактировать пространственную информацию, а также составлять и экспортировать графические карты. QGIS поддерживает растровые, векторные и сетчатые слои. Векторные данные хранятся в виде точечных, линейных или полигональных объектов. Поддерживаются несколько форматов растровых изображений.

QGIS поддерживает шейп-файлы, персональные базы геоданных, dxf, MapInfo, PostGIS и другие стандартные отраслевые форматы. Веб-службы, в том

числе служба веб-карт и служба веб-объектов, также поддерживаются, позволяя использовать данные из внешних источников.

QGIS интегрируется с другими ГИС-пакетами с открытым исходным кодом, включая PostGIS, GRASS GIS и MapServer. Плагины, написанные на Python или C++, расширяют возможности QGIS. Плагины могут выполнять геокодирование с помощью Google Geocoding API, выполнять функции геообработки, аналогичные функциям стандартных инструментов ArcGIS, и взаимодействовать с базами данных PostgreSQL/PostGIS, SpatiaLite и MySQL.

QGIS может отображать несколько слоев, содержащих различные источники или их изображения.

Как бесплатное программное приложение под GNU GPLv2, QGIS можно свободно модифицировать для выполнения других или более специализированных задач. Двумя примерами являются приложения QGIS Browser и QGIS Server, которые используют один и тот же код для доступа к данным и визуализации, но имеют разные внешние интерфейсы.

Для реализации алгоритма будет использоваться язык программирования PyQGIS. Данный язык представляет собой язык программирования Python с доступом к API функциям QGIS. Он позволяет запускать созданный скрипт в консоли системы QGIS, что ускоряет и упрощает процесс реализации алгоритма.

1.2 Обзор существующих подходов

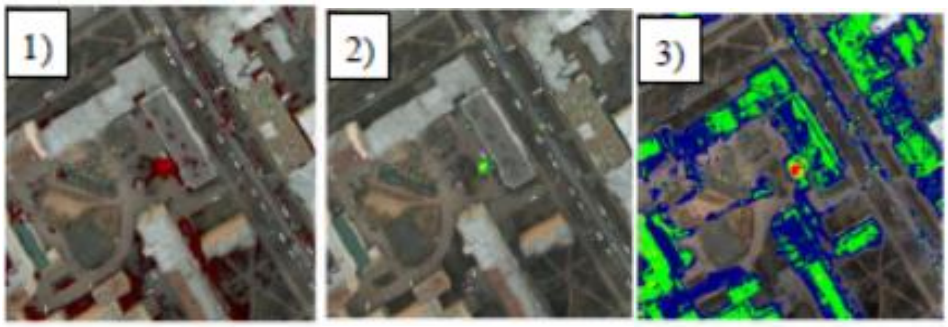
Статья «Методы комплексирования изображений в многоспектральных оптико-электронных системах» А.С. Васильева и А.В. Трушкиной посвящена рассмотрению вопросов комплексирования изображений многоспектральных оптикоэлектронных систем [1]. Рассмотрены принципы формирования и методы комплексирования изображений. Оценка качества результирующего изображения выполнялась на основе расчета значения перекрестной энтропии, структурной схожести и контраста, предложены критерии объективной оценки

качества комплексированного изображения. Исследования методов комплексирования проводились по изображениям, полученным в видимом и дальнем инфракрасном спектральных диапазонах при обследовании тепловых сетей г. Санкт-Петербург.

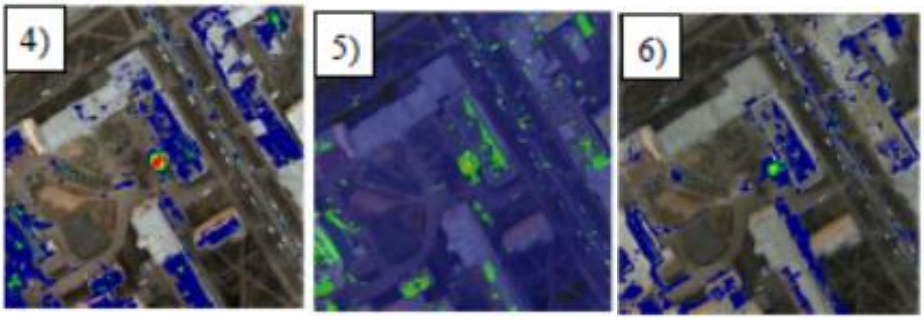
Авторами были изучены следующие алгоритмы комплексирования векторных данных:

- метод максимума;
- метод маски;
- метод усреднения;
- метод степенного преобразования;
- метод чересстрочного комплексирования;
- метод весовой функции.

Эффективность применяемых методов предлагается оценивать с помощью информационной энтропии от контуров изображений. Результаты исследований показали, что по субъективным и объективным критериям оценки качества изображения, методы степенного преобразования и весовой функции обладают лучшими характеристиками. Результаты реализации методов представлены на рисунке 1. Как видно из представленных результатов, для методов 3 и 5 на комплексированном изображении присутствует много лишних деталей, для методов 1 и 2 произошла существенная потеря информации. Методы 4 и 6 показали лучшие результаты, сохранив необходимые информативные признаки двух изображений. Для объективной оценки качества методов комплексирования необходимо применение критериев оценки определения эффективности преобразований, включающих оценку информационной составляющей на результирующем изображении и качественную оценку его восприятия.



а) б) в)



г) д) е)

Рисунок 1 - Результаты комплексирования изображений: а – метод максимума; б – метод маски; в – метод усреднения; г – метод степенного преобразования; д – метод чересстрочного комплексирования; е – метод весовой функции

Д. С. Шарак, А .В. Хижняк и А.С. Мамченко изучили вопрос применения комплексирования изображений для повышения эффективности работы корреляционного алгоритма сопровождения в условиях сложной фоно-целевой обстановки и наличия преднамеренных помех при относительно невысоком контрасте объекта интереса [2]. Применение типового корреляционного алгоритма показало неплохие результаты по сопровождению объектов, в том числе для таких сложных условий, как сопровождение целей с подвижной платформы при невысоком её контрасте относительно сложного фона. В то же время, при уменьшении контраста наблюдались срывы сопровождения.

Для выявления условий возникновения срывов при сопровождении объектов типовым корреляционным алгоритмом был проведён эксперимент. Смоделированные условия показаны на рисунке 2.

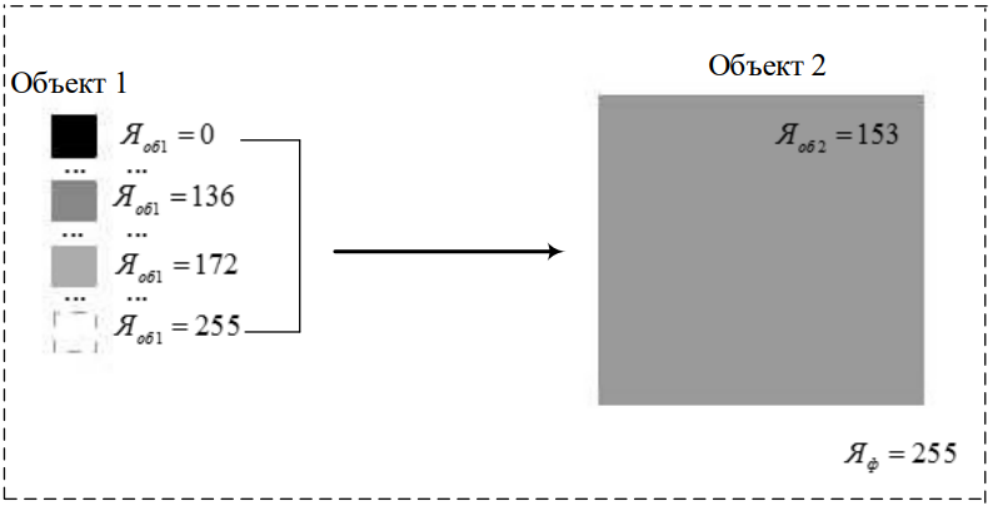


Рисунок 2 – Определение условий возникновения срывов сопровождения при работе типового корреляционного алгоритма

Объект интереса (объект 1 на рисунке 2) движется на равномерном фоне с яркостью пикселей равной 255 слева направо. В определенный момент времени объект попадает в область 2 (объект 2 на рисунке 2) с яркостью пикселей равной 153. Диапазон значений изменения яркости пикселей изображений изменяется в интервале от 0 до 255. С момента начала движения объект интереса захватывается на сопровождение типовым корреляционным алгоритмом. Значение яркости объекта в начальный момент времени равен 0 и увеличивается с каждым экспериментом на 1. На каждом эксперименте фиксируется наличие срыва сопровождения.

Проведенное моделирование работы корреляционного алгоритма показало, что срывы сопровождения наблюдаются в случаях разности в яркостях пикселей объекта и фона менее 11%.

С другой стороны, отличия в яркости пикселей объекта и фона менее 11% можно рассматривать как ситуации постановки различного рода помех и сложной фоно-целевой обстановки. Подобные ситуации снижают эффективность работы корреляционного алгоритма и часто имеют место в реальной обстановке. Для этого путем применения алгоритмов комплексирования необходимо добиться превышения разности в яркостях пикселей объекта и фона более 11%.

Были получены изображения, комплексированные по шести алгоритмам, а именно сложение яркости пикселей двух изображений, метод усреднения, метод максимума, комплексирование по квадратному корню, критериальное суммирование и попеременная запись строк.

На следующем этапе в эксперименте были использованы результаты комплексирования при работе корреляционного алгоритма. Анализ работы корреляционного алгоритма с вновь полученными комплексированными изображениями показал, что срывов сопровождения не наблюдалось.

Общий вывод по результатам моделирования заключается в том, что результат комплексирования имеет более высокий контраст по отношению к фону, чем каждое из исходных изображений в отдельности.

Таким образом, в условиях сложной фоно-целевой обстановки, когда уровень контраста объекта интереса в инфракрасном и видимом диапазонах незначителен, целесообразно применение алгоритмов комплексирования изображений для повышения устойчивости работы корреляционного алгоритма автоматического сопровождения.

Praveen Rao в статье “Efficient processing of raster and vector data” предложил структуру для хранения пространственных данных, которая включает в себя новые эффективные алгоритмы для выполнения операций, принимающих в качестве входных данных набор растровых и векторных данных [3].

В предложенном фреймворке векторные наборы данных хранятся с использованием традиционной настройки, индексируемой с помощью R-деревьев, а растровые наборы данных хранятся и индексируются с помощью k2-растров. Выбор R-дерева для индексации векторного набора данных является прагматичным выбором, поскольку оно является стандартом для этого типа данных. Использование k2-растра представляло собой сложную задачу. Он был разработан для использования в основной памяти без предварительной распаковки всего набора данных, но для этого требуется сложное расположение данных (что-то аналогичное большинству компактных структур данных). Это

означает, что процессы распаковки, применяемые к небольшим частям по требованию, и управление содержащимися в них индексами могут снизить потребление основной памяти и повлиять на время обработки.

Автор так же представил эффективное пространственное соединение между растровыми и векторными наборами данных. Алгоритм заключается в вычислении соединения между растровым и векторным набором данных, накладывающий ограничение диапазона на значения растра. Следовательно, алгоритм возвращает элементы векторного набора данных (полигоны, линии или точки) и положение ячеек растрового набора данных, которые перекрывают друг друга, так что ячейки имеют значения в заданном диапазоне. Наиболее важной операцией является проверка того, перекрывает ли элемент векторного набора данных область набора растровых данных, имеющую значения в пределах запрошенного диапазона.

В ходе исследовательской части, проанализировав литературные материалы по теме работы, было выявлено, что существует достаточное количество статей, описывающих различные методы комплексирования изображений, а для векторных данных мало информации и нужны новые подходы.

2 Разработка алгоритмов

Разработка алгоритма была разделена на два этапа. Первый этап состоит из разработки комплексирования векторных данных, в ходе этого этапа был создан скрипт barycentric_coor.py, были разработаны основные концепции. Во время второго этапа были произведены улучшения комплексирования векторных данных, а также был разработан алгоритм построения пространственного индекса для сохранения топологических отношений. В ходе второго этапа был создан скрипт indexing_of_elements.py.

2.1 Разработка алгоритма комплексирования векторных данных

2.1.1 Словесное описание алгоритма.

Алгоритм основывается на принципе вычисления барицентрических координат.

Рассмотрим частный случай барицентрических координат с использованием треугольников. Допустим, имеется определенная точка *P* с координатами *x* и *y*. А также два треугольника с вершинами P_1, P_2, P_3 и P'_1, P'_2, P'_3 соответственно. Треугольники разные по размеру, но вершины треугольников соответствуют друг другу. Точка *P* расположена внутри первого треугольника. Необходимо перенести точку из одного треугольника в другой с учетом её расположения относительно вершин и сторон исходного треугольника. Рассмотрим алгоритм для расчета барицентрических координат.

Найдём площадь S треугольника с точкой P. Проведем от точки P линии к вершинам треугольника. Найдем площади S_1, S_2, S_3 получившихся треугольников. Произведем вычисления по формулам 1, 2 и 3

$$u = \frac{S_1}{S} \quad (1)$$

$$v = \frac{S_2}{S} \quad (2)$$

$$w = \frac{S_3}{S} \quad (3)$$

где $u, v, w,$ - барицентрические координаты, причем $u + v + w = 1$.

Вычислим расположение точки P` по формуле 4.

$$P` = uP_1` + vP_2` + wP_3` \quad (4)$$

С помощью барицентрических координат точка была скопирована. На рисунке 3 отображен данный процесс.

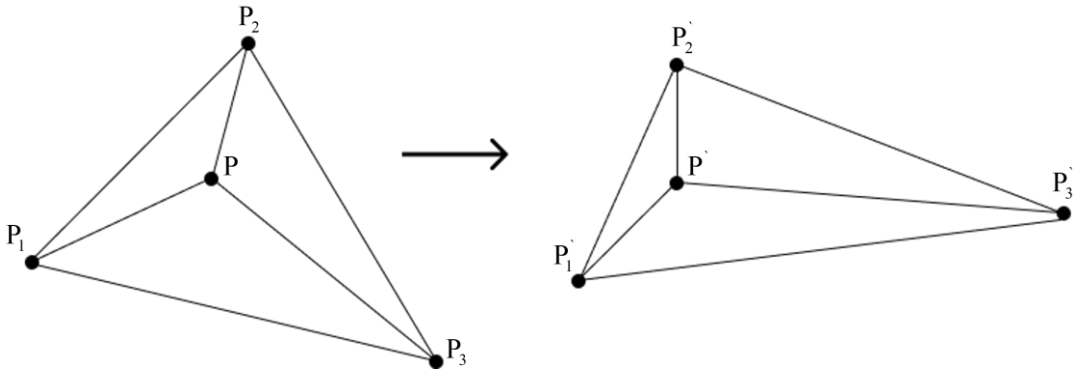


Рисунок 3 – Пример работы барицентрических координат на одном треугольнике

Следовательно, барицентрический координаты - три числа, в сумме равные единице, определяющие положение точки в треугольнике, равные массам, которые следует поместить в вершинах треугольника так, чтобы определяемая точка сделалась центром тяжести этих масс.

Алгоритм так же должен работать на больших объемах данных с большим количеством точек, а для этого требуется больше одного треугольника.

Пример работы барицентрических координат на больших данных представлен на рисунках 4 - 5. На рисунке 4 представлены две исходные карты с базовыми точками для построения треугольников. На рисунке 5 представлены эти же две карты после преобразования.

Для автоматического построения треугольников на базовых точках была использована триангуляция Делоне [7]. На множестве точек на плоскости задана триангуляция, если некоторые пары точек соединены ребром, любая конечная грань в получившемся графе образует треугольник, ребра не пересекаются, и граф максимален по количеству ребер. Триангуляцией Делоне называется такая триангуляция, в которой для любого треугольника верно, что внутри описанной около него окружности не находится точек из исходного множества. На рисунке 6 представлена обычная триангуляция, на рисунке 7 представлена триангуляция Делоне. Изображения взяты из открытых источников [7].

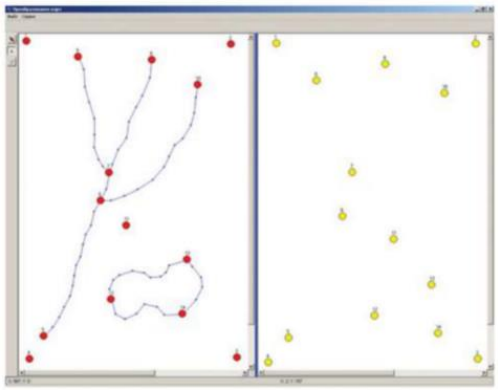


Рисунок 4 – Исходные данные

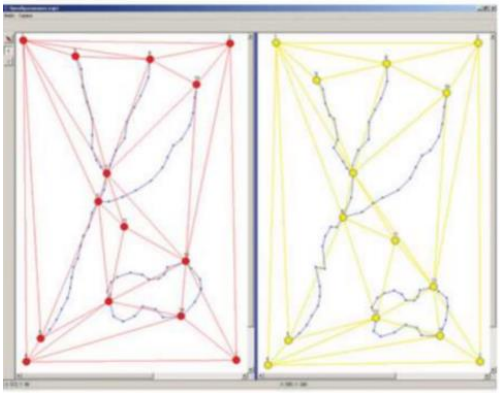


Рисунок 5 – После преобразования

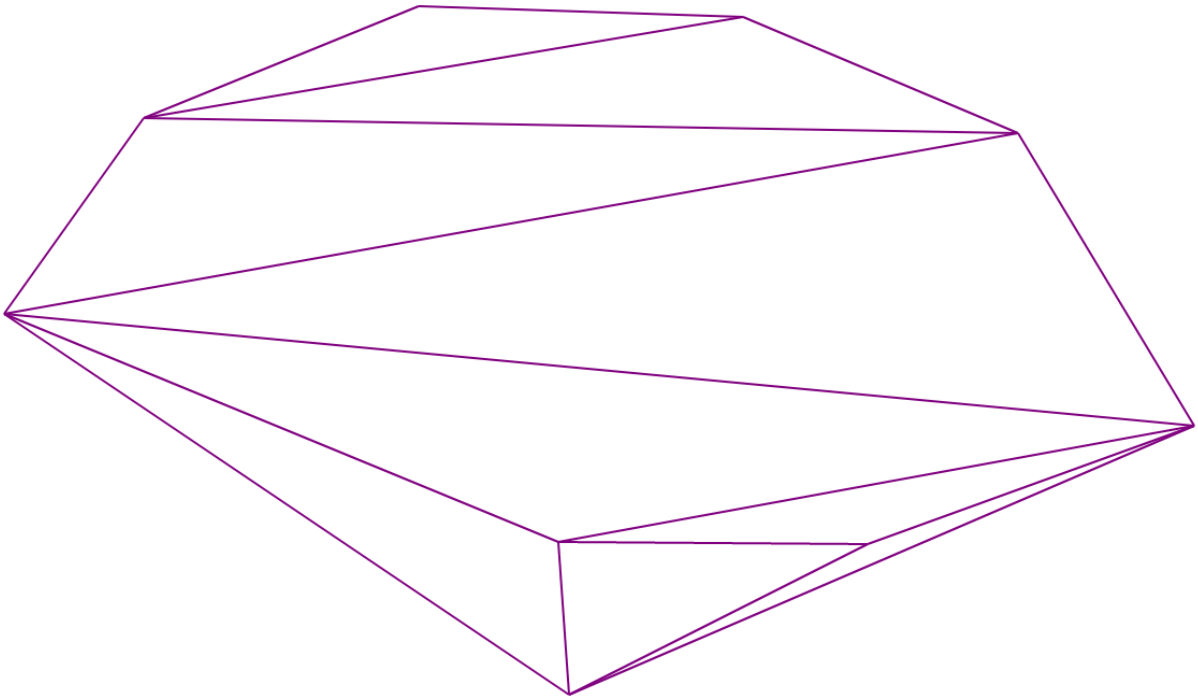


Рисунок 6 – Триангуляция

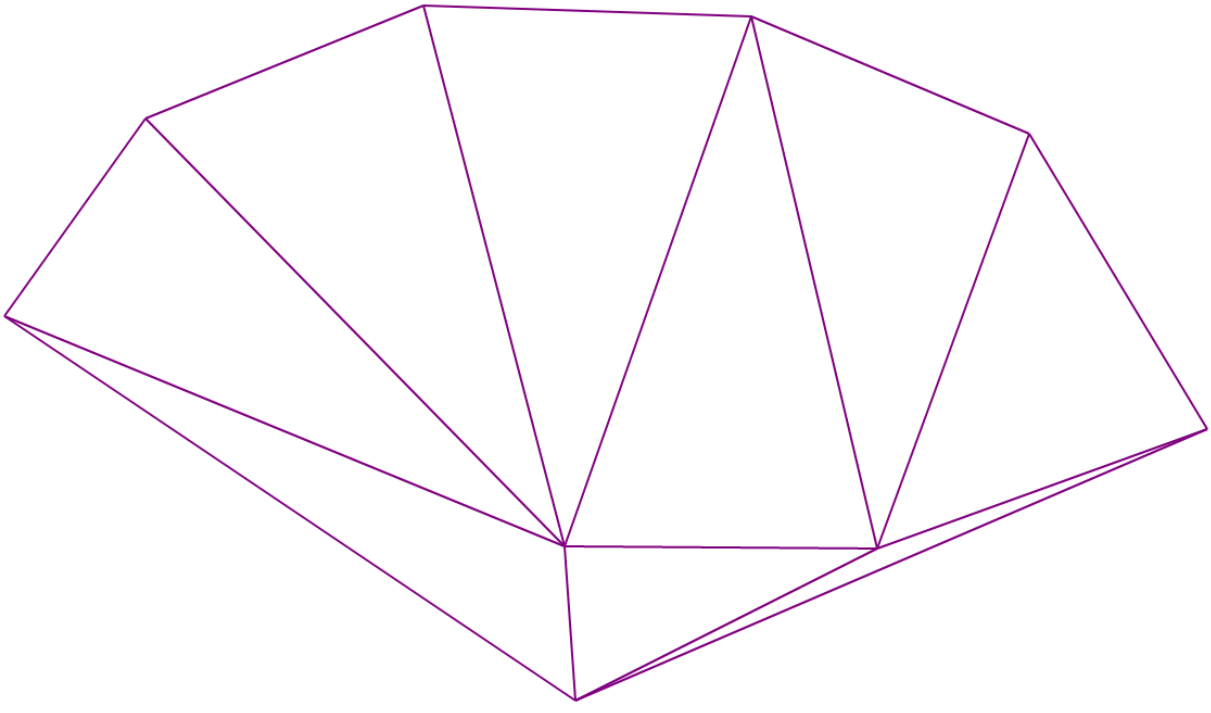


Рисунок 7 – Триангуляция Делоне

Алгоритм триангуляции Делоне основан на стандартной для многих алгоритмов методике сведения сложной задачи к более простым, в которых решение очевидно. Сам алгоритм состоит из 2 шагов.

Шаг №1. Разбиение исходного множества на более мелкие множества. Для этого мы проводим вертикальные или горизонтальные прямые в середине множества и уже относительно этих прямых разделяем точки на две части. Далее для каждой группы точек рекурсивно запускаем процесс деления.

Шаг №2. Объединение оптимальных триангуляций. Сначала находятся две пары точек, отрезки которых образуют в совокупности с построенными триангуляциями выпуклую фигуру. Они соединяются отрезками, и один из полученных отрезков выбирается как начало для последующего обхода. Обход заключается в следующем: на этом отрезке мы как будто «надуваем пузырь» внутрь до первой точки, которую достигнет раздувающаяся окружность «пузыря». С найденной точкой соединяется та точка отрезка, которая не была с ней соединена. Полученный отрезок проверяется на пересечение с уже существующими отрезками триангуляции, и в случае пересечения они удаляются из триангуляции. После этого новый отрезок принимается за начало для нового «пузыря». Цикл повторяется до тех пор, пока начало не совпадёт со вторым отрезком выпуклой оболочки.

Для реализации алгоритма был создан скрипт barycentric_coor.py, написанный на языке программирования Python и использующий возможности QGIS Python API [6].

2.1.2 Структура файла

Скрипт состоит из класса Moved, создания объекта данного класса и запуска методов преобразования.

В свою очередь класс Moved состоит из нескольких функций. Ниже будут рассмотрены основные функции класса, использующиеся для вычислений. Некоторые функции не будут упомянуты, так как они реализованы для удобства разработки алгоритма и программирования в целом.

Функция _init__ - функция инициализаций, запускается при создании объекта класса Moved. В данной функции задаются сконстанты, использующиеся в дальнейшем во всех функциях класса, а именно:

– vertex_point_in – имя слоя с базовыми точками первой карты;

– vertex_point_out – имя слоя с базовыми точками второй карты;

– move_layer – имя слоя с объектами для переноса;

– type_of_geom – тип геометрии исходного слоя;

– coordinate_system – используемая координатная система в проекте.

Функция is_in_triangle выполняет проверку находится ли точка внутри треугольника. Для того чтобы определить лежит ли точка P внутри треугольника ABC необходимо вычислить 3 векторных произведения: ABxAP, BCxBP и CAxCP. Получив результаты по трем векторным произведениям, остается их проанализировать, чтобы понять лежит ли точка внутри треугольника. Если мы имеем и положительные и отрицательные результаты, точка лежит вне треугольника, если результаты только положительные или только отрицательные, точка - внутри.

Функция draw_triangles выполняет отрисовку треугольников на карте. Происходит это следующим образом:

– получаем базовые точки двух карты;

– выполняем триангуляцию Делоне на базовых точках первой карты;

– основываясь на полученных треугольников первой карты, нарисуем треугольники второй карты;

– с помощью средств QGIS обрисовываем треугольники.

Функции barycentric_out вычисляет барицентрические координаты точки относительно треугольника. А функция barycentric_in вычисляет координаты точки относительно ее барицентрических координат.

Функция run основная функция класса, запускающая другие функции. А также проводит проверку входных данных, типов геометрии и отрисовывает комплексированный слой.

2.1.3 Порядок работы данного алгоритма.

Загружаем в QGIS исходные данные в виде векторных слоёв. На рисунках 8 и 9 представлены исходные карты. Так как исходные карты загружены в виде слоёв можно рассмотреть сходства и различия между ними. На рисунке 10 представлены сходства и различия исходных карт.

Рисунок 8 – Исходная карта 1

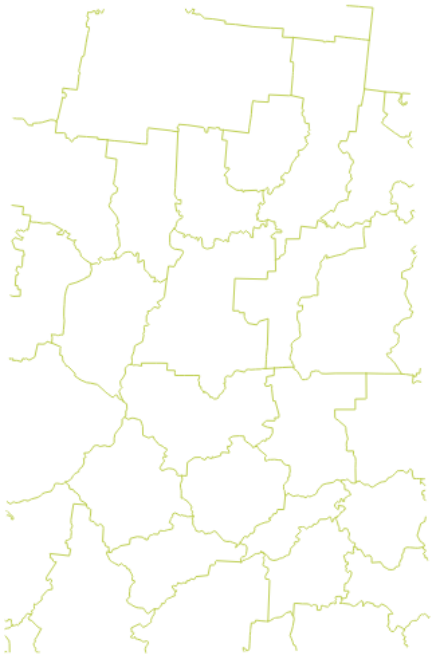


Рисунок 9 – Исходная карта 2

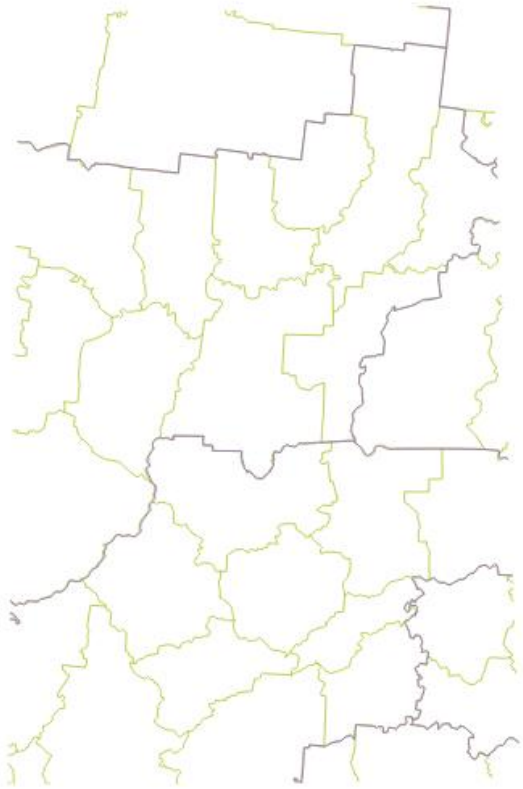
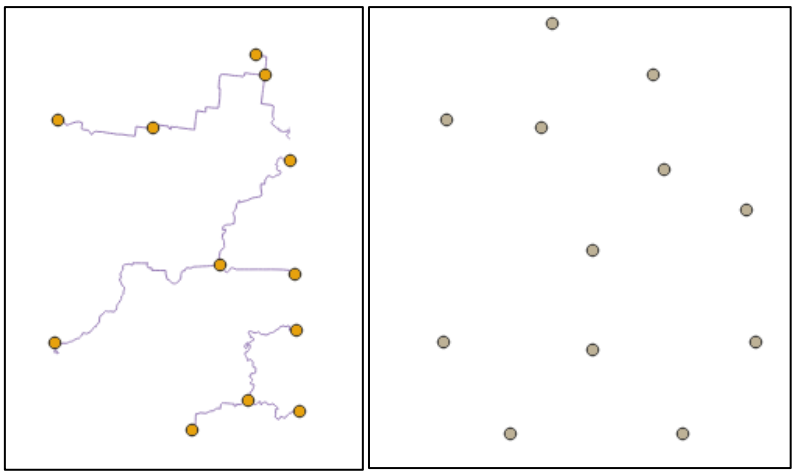


Рисунок 10 – Сходства и различия

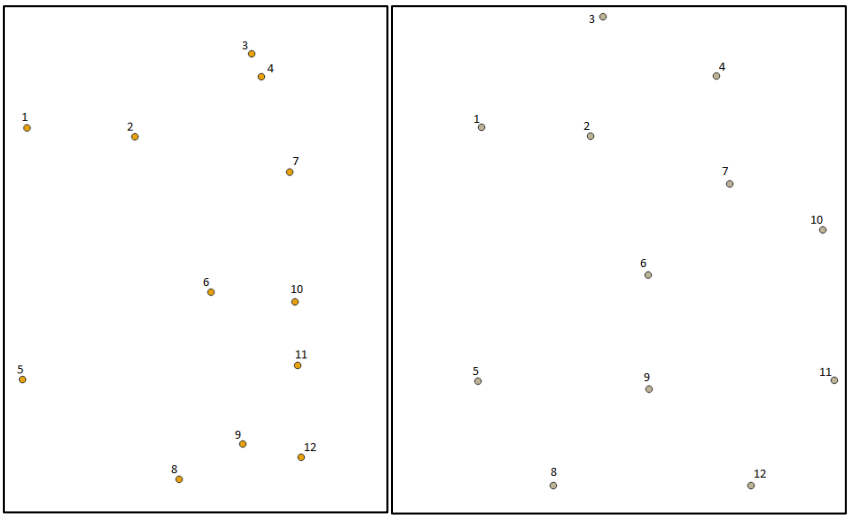
Расставим базовые точки, которые будут являться вершинами будущих треугольников. Для наглядности можно немного перенести базовые точки второй карты. Базовые точки двух карт представлены на рисунке 11.



а) б)

Рисунок 11 – Базовые точки: а – точки первой карте; б – точки второй карты

Необходимо сделать так, чтобы определённая точка первой карты соответствовала определённой точке второй карты. Для этого назначим им идентификаторы. Идентификаторы базовых точек представлены на рисунке 12.



а) б)

Рисунок 12 – Идентификаторы: а – точки первой карты; б – точки второй карты

Добавим на первую карту некоторые объекты, которые будем переносить на вторую. Добавленные объекты представлены на рисунке 13.

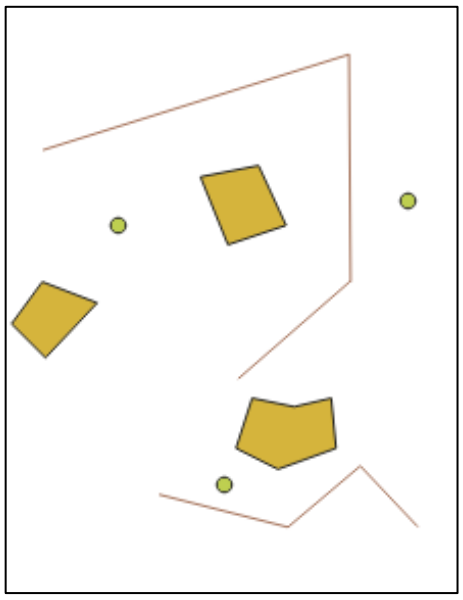


Рисунок 13 – Объекты

Произведем триангуляцию Делоне на первой карте. Результат триангуляции Делоне представлен на рисунке 14. Так как точки на второй карте расположены немного по-другому относительно точек на второй карте, триангуляцию Делоне использовать нельзя, потому что могут появится нежелательные треугольники. На основе идентификаторов точек, расставленных в пункте 3, построим треугольники на второй карте, так чтобы треугольники обеих карт соответствовали друг другу. Треугольники второй карты представлены на рисунке 15. С помощью барицентрических координат перенесём объекты с первой карты на вторую. Результаты комплексирования объектов представлены на рисунках 16 – 18.

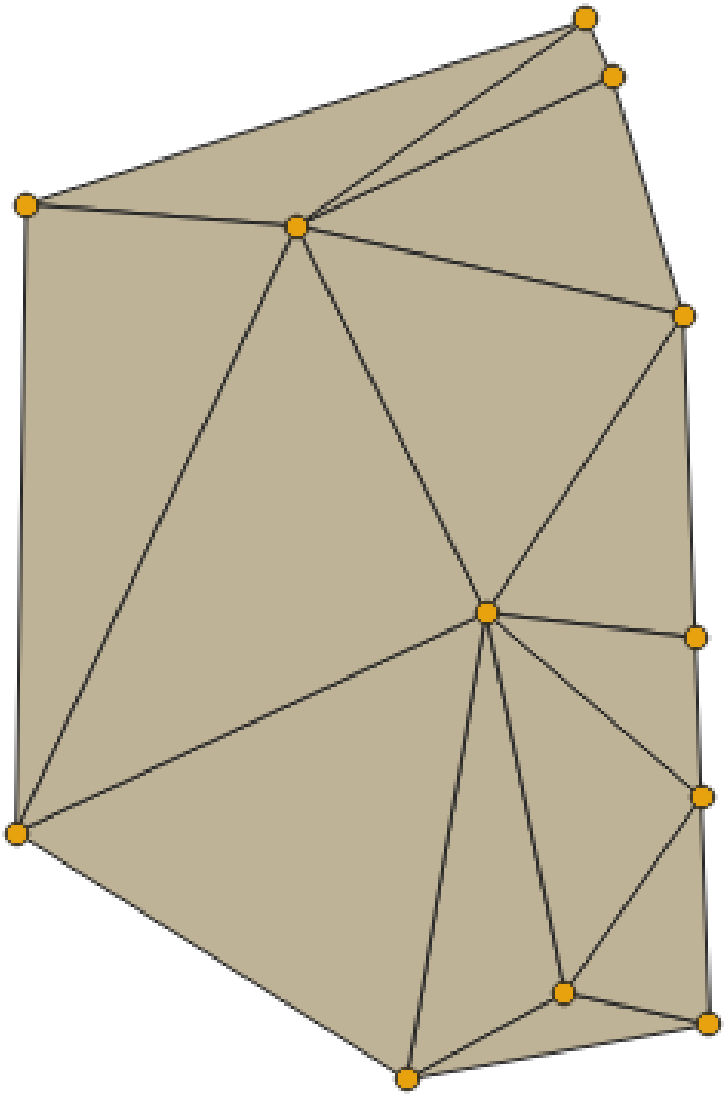


Рисунок 14 – Триангуляция

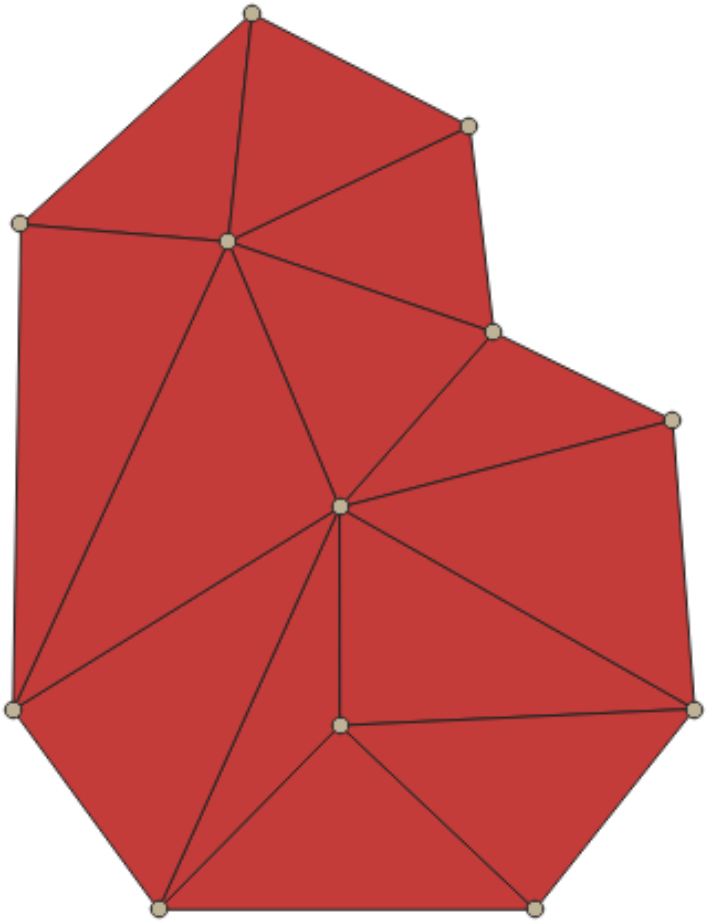
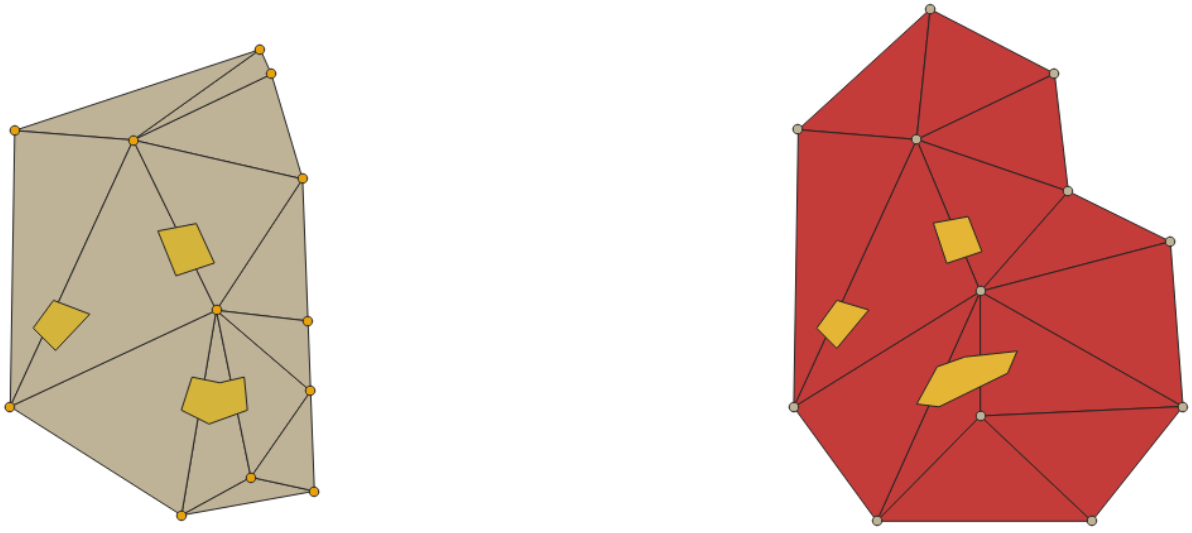
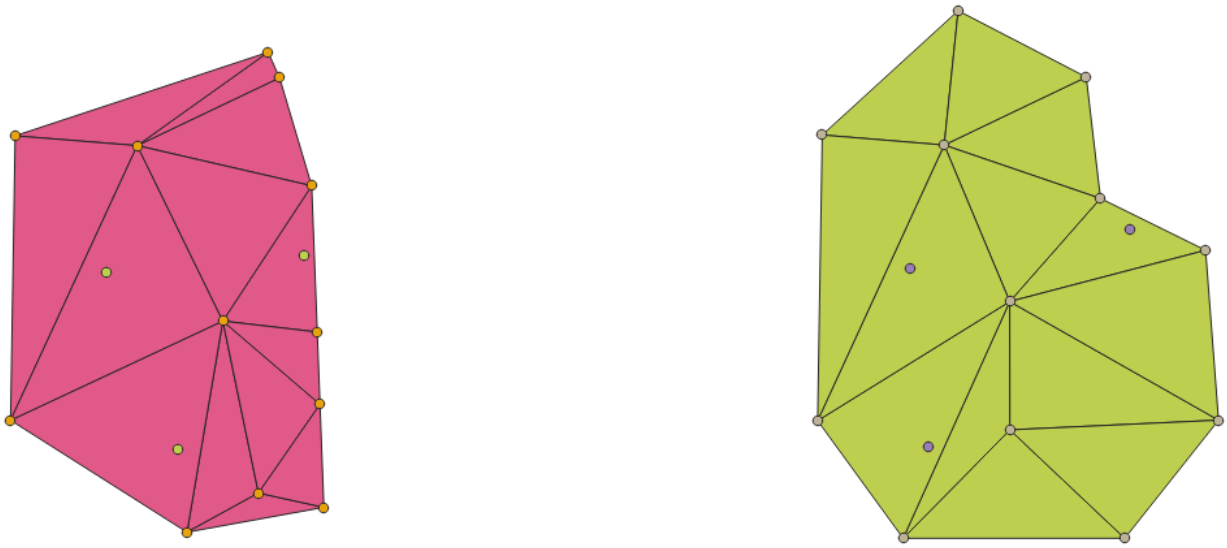


Рисунок 15 – Триангуляция на второй карте



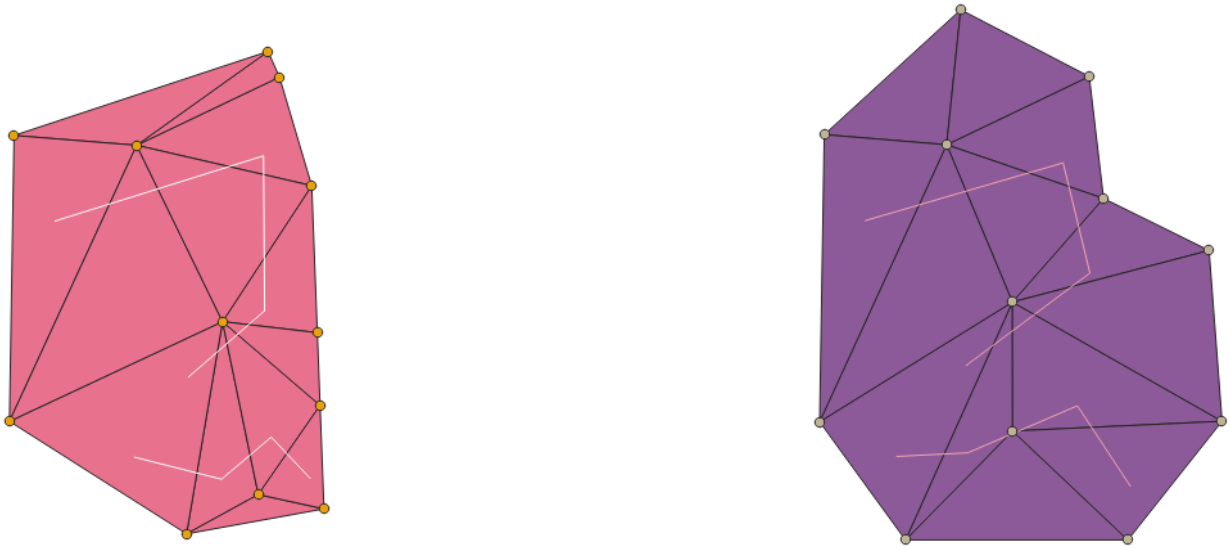
а) б)

Рисунок 16 – Перенос полигональных объектов: а – исходные объекты; б – комплексированные объекты



а) б)

Рисунок 17 – Перенос точечных объектов: а – исходные объекты; б – комплексированные объекты



а) б)

Рисунок 18 – Перенос линейных объектов: а – исходные объекты; б – комплексированные объекты

На рисунке 18 можно заметить, что на второй карте нижний линейный объект проходит через базовую точку. Это происходит, потому что в QGIS для построения линейных объектов требуется указать только массив точек, между

которыми автоматически построятся отрезки. Каждая точка этого массива находится в своём треугольнике и привязана к нему. Поэтому при переносе учитывается только расположение конкретной точки внутри треугольника. Решение данной проблемы будет описано ниже.

2.2 Улучшения алгоритма комплексирования данных

Первым делом необходимо было улучшить перенос линейных и полигональных объектов. Полигональный объект состоит из нескольких зацикленных линейных объектов. Поэтому было принято решение что каждый линейный объект будет делиться на несколько промежуточных объектов – отрезков. Таким образом у каждого линейного объекта появляется большее количество точек, что улучшает точность переноса объекта на другую карту.

На рисунке 19 представлена исходная линия. На рисунке 20 представлен результат переноса линии без деления ее на несколько отрезков. На рисунке 21 представлен результат переноса линии с делением.

Такой же эксперимент был проведён и над полигональными объектами. На рисунке 22 представлены исходные полигоны. На рисунке 23 представлены полигоны без деления линий, из которых состоят эти полигоны. На рисунке 24 представлены полигоны с делением линий.

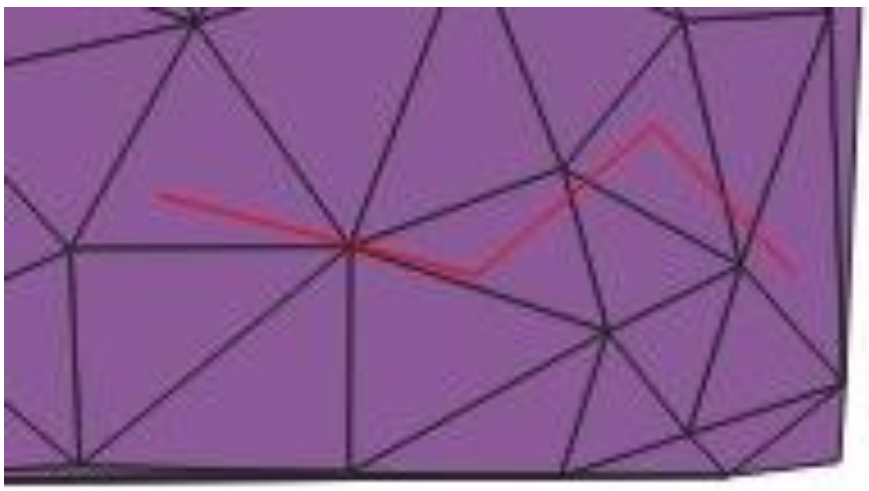


Рисунок 19 – Исходная линия

Рисунок 20 – Линия без деления

Рисунок 21 – Линия с делением

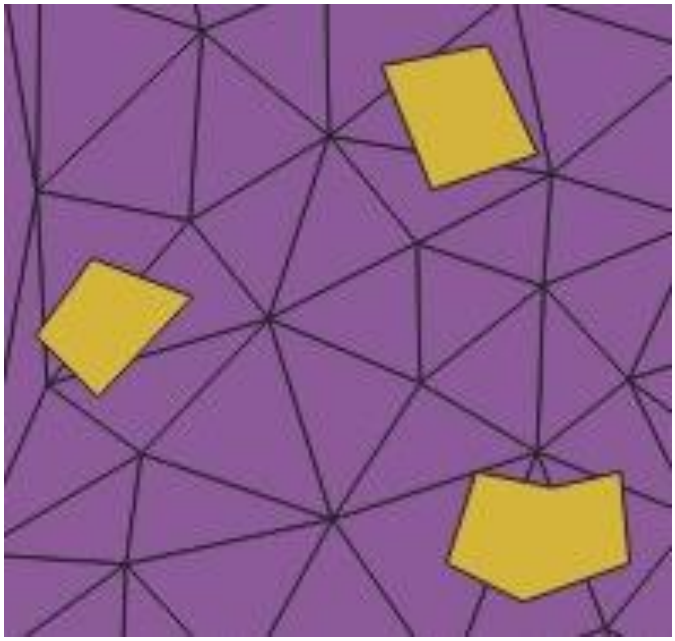


Рисунок 22 – Исходные полигоны

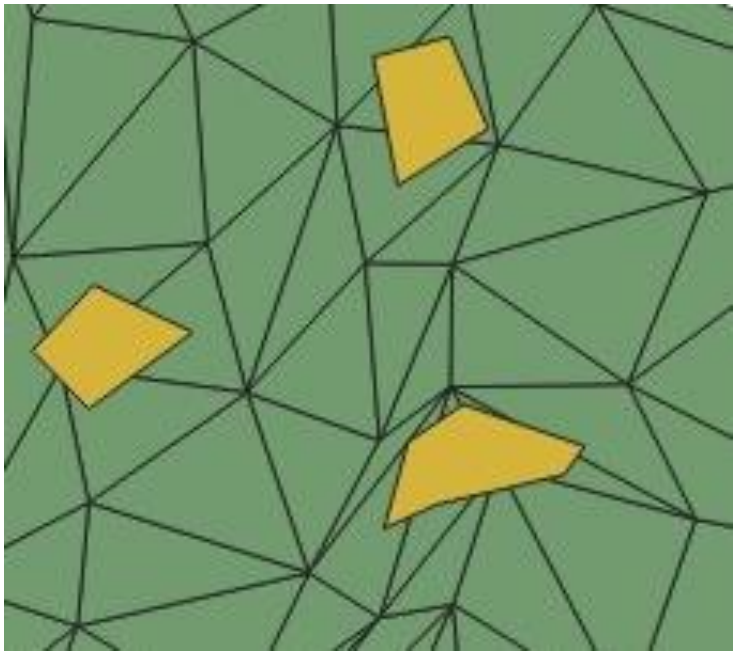


Рисунок 23 – Полигоны без деления

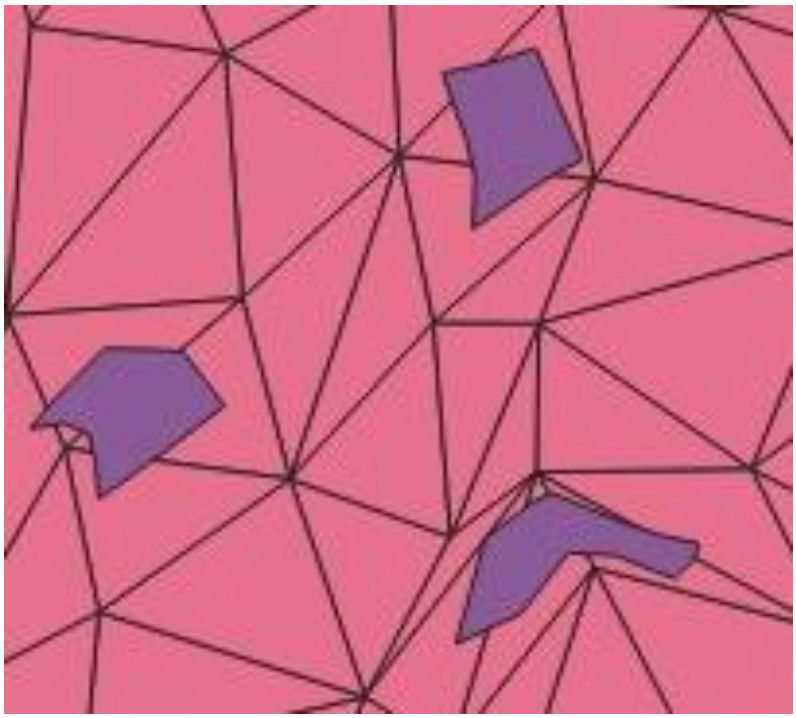


Рисунок 24 – Полигоны с делением

Одной из проблем появившейся в процессе первого этапа разработки, является необходимость расставлять базовые точки вручную. Решение нашлось в использовании алгоритма SIFT [8]. Данный алгоритм способен находить одинаковые точки на двух растровых изображениях. Чтобы данный алгоритм

работал вместе с разрабатываемым, необходимо преобразовать исходные векторные карты в растровый формат, то есть растеризация. Данный процесс можно выполнить с помощью встроенного средства QGIS. На рисунке 25 представлен пример запуска функции растеризации.

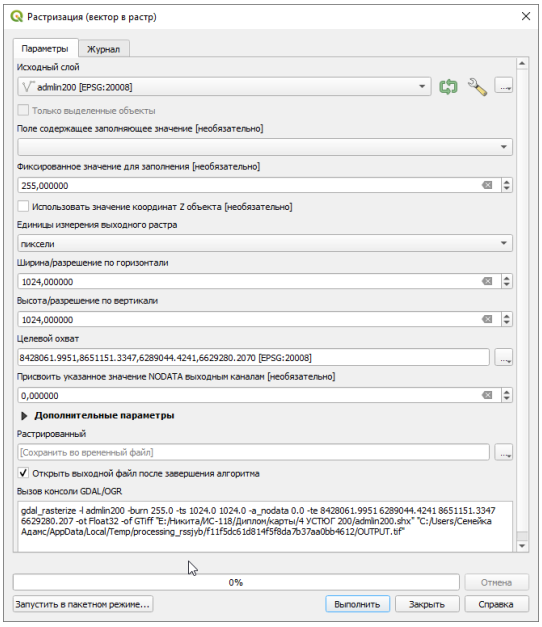


Рисунок 25 – Растеризация векторного слоя

На языке программирования Python SIFT рассчитывается приведенным ниже скриптом.

```
import cv2

sift = cv2.xfeatures2d.SIFT_create()
psd_kp1, psd_des1 = sift.detectAndCompute(npKernel_eroded1,
None)
psd_kp2, psd_des2 = sift.detectAndCompute(npKernel_eroded2,
None)

FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(psd_des1, psd_des2, k=2)
goodMatch = []
for m, n in matches:
    if m.distance < 0.50*n.distance:
        goodMatch.append(m)
```

Переменные npKernel_eroded1 и npKernel_eroded2 хранят изображения для обработки алгоритмом SIFT. Список goodMatch это отфильтрованная высококачественная пара.

На рисунке 26 представлен результат работы алгоритма SIFT.

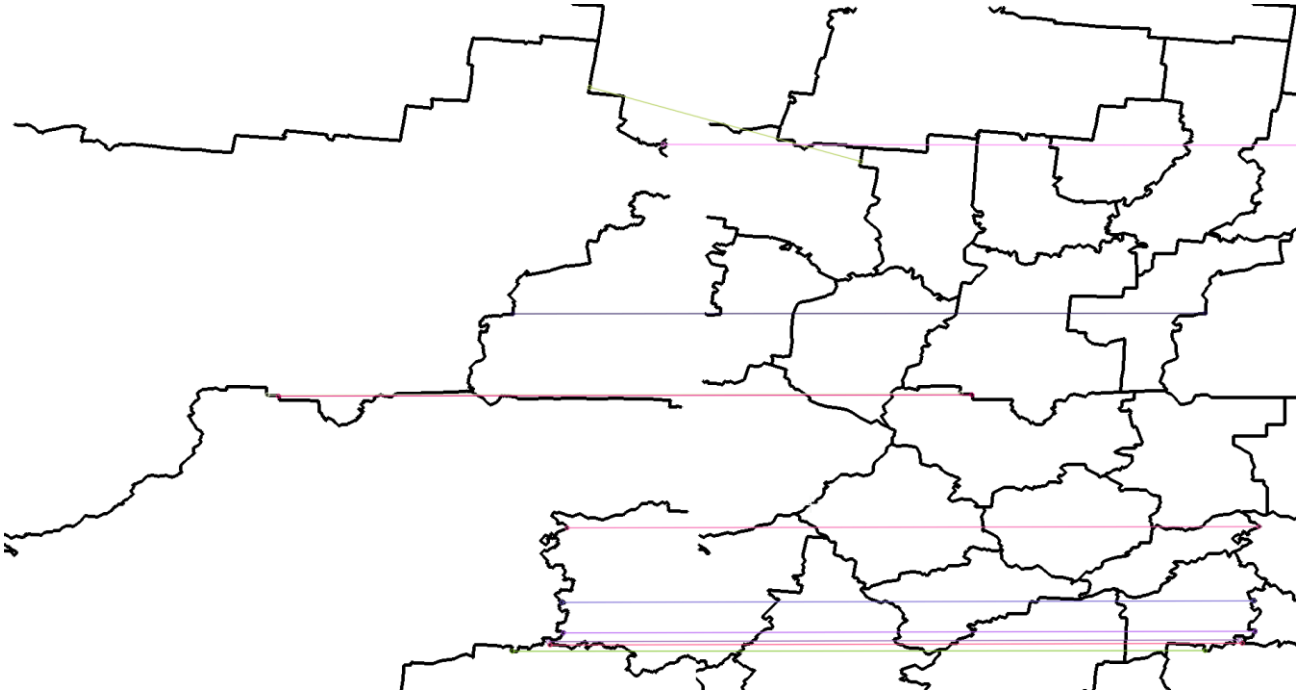


Рисунок 26 - Результат работы алгоритма SIFT.

Полученные базовые точки были перенесены в векторный формат и уже на их основе проводилась дальнейшая работа алгоритма.

На рисунке 27 представлена исходная карта с тестовыми данными и, построенными на основе полученных базовых точек, треугольниками. На рисунке 28 представлена уже преобразованная карта.

Как можно заметить переместились объекты, расположенные только внутри треугольников, поэтому было решение добавить на каждый угол карты еще по одной точке. На рисунке 29 представлены треугольники первой карты, покрывающие всю область карты. На рисунке 30 представлена преобразованная карта с добавленными по краям базовыми точками.

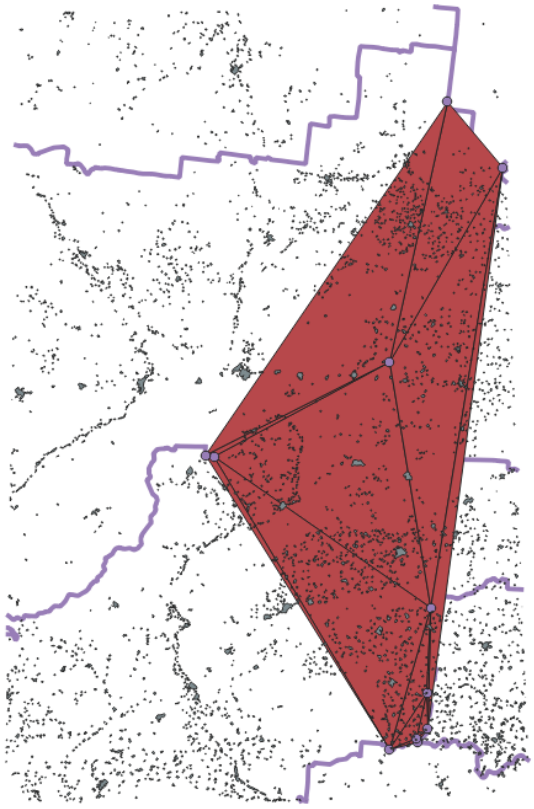


Рисунок 27 – Исходная карта

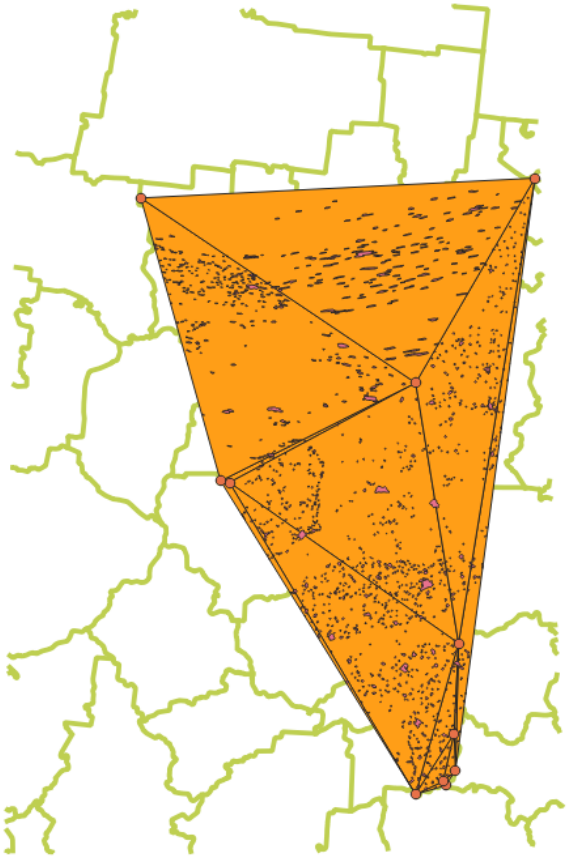


Рисунок 28 – Преобразованная карта

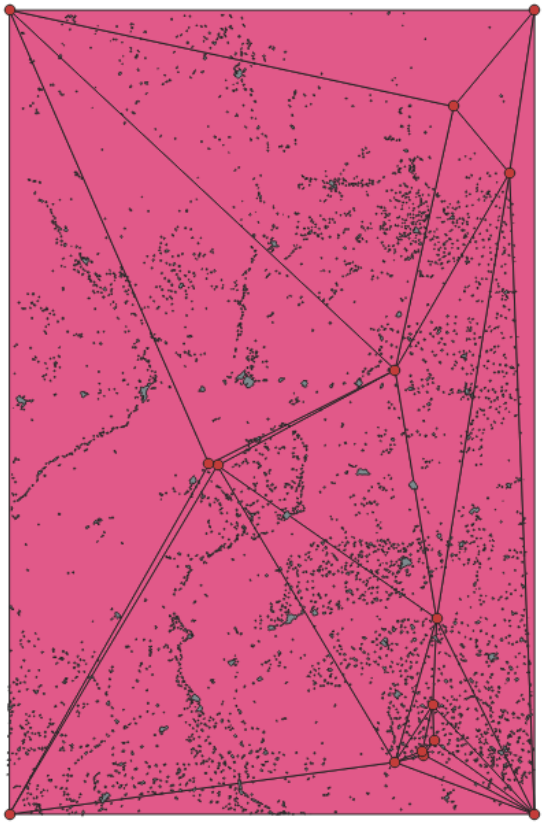


Рисунок 29 – Добавленные точки по углам карты

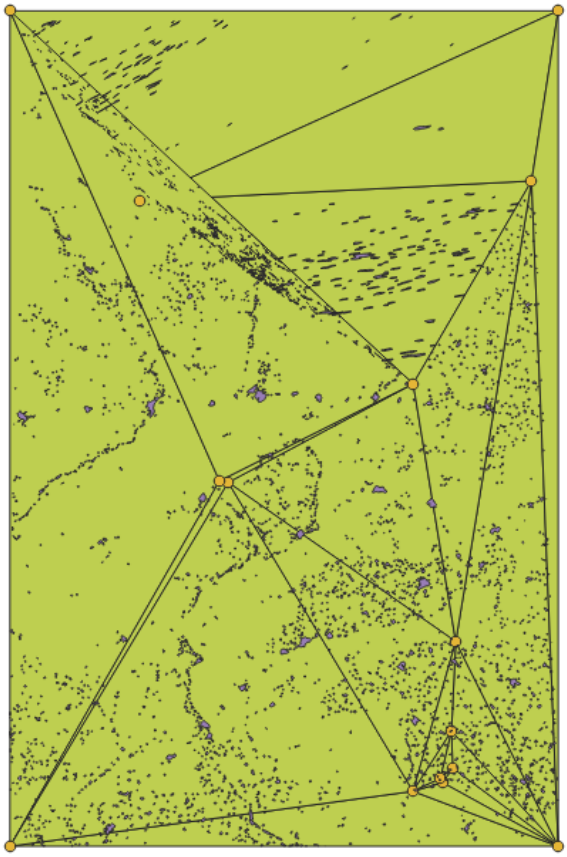


Рисунок 30 – Преобразованная карта

Основной концепцией разрабатываемого алгоритма является перемещение объектов с сохранением топологических связей. Для этого было принято решение использовать алгоритм построения пространственного индекса. С помощью него мы можем отслеживать расположение каждого перемещаемого объекта на слое и его топологические связи. Это было реализовано в файле indexing_of_elements.py. Скрипт был написан на языке программирования PyQGIS.

2.3 Разработка алгоритма индексирования

2.3.1 Словесное описание алгоритма

Файл indexing_of_elements.py включает в себя класс Index_of_elements, включающий в себя 11 основных функций. Теперь подробно разберем алгоритм индексации.

После запуска программы инициализируются 3 глобальные переменные для хранения: объекта класса QgsProject, списка названий всех слоев по выбранному проекту и пустого списка для индексирования временных слоев (прямоугольников). Кроме этого, запускается функция удаления, которая при наличии временных слоев удаляет их. Причем только те слои, в названиях которых содержится имя «rectangle», чтобы избежать удаления важных слоев.

Запускается функция для нахождения крайних точек карты. Экстремумы формируются после сравнения координат всех точек объектов в каждом слое. По этим точкам строится основной прямоугольник белого цвета в виде временного слоя. Данный слой и все созданные в дальнейшем прямоугольники будут иметь тип MultiPolygon.

Далее выполняется деление главного прямоугольника на 4 основных. На данном этапе сначала математически находятся точки, по которым будут строиться временные слои. Для этого определяются середины каждых сторон главного прямоугольника и его центр. По найденным точкам строятся 4

прямоугольника белого цвета. Белый цвет будет указывать на отсутствие в нём объектов. По завершению построения главный слой удаляется, так как в дальнейшем он будет мешать отображению внутренних прямоугольников.

После этого запускается рекурсия по 4 основным слоям. Сущность рекурсии заключается в следующем:

– если в прямоугольнике находится более 1 объекта вызвать функцию разделения и выполнить повтор, на рисунке 31 представлена данная ситуация;

– если в прямоугольнике находится 1 объект, то не выполнять разделение и перейти к следующему прямоугольнику, на рисунке 32 представлена данная ситуация;

– если в прямоугольнике несколько объектов, но они имеют общую точку пересечения, то не выполнять разделение и перейти к следующему прямоугольнику, на рисунке 33 представлена данная ситуация;

– если в каждом прямоугольнике по 1 объекту или несколько объектов с общей точкой, то остановить рекурсию, это представлено на рисунке 34.

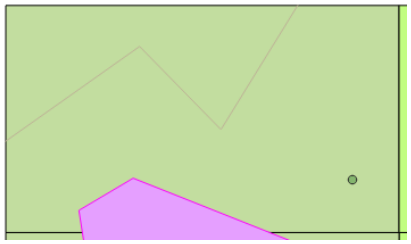


Рисунок 31 – Прямоугольник с 3-мя объектами, в котором необходимо сделать разделение

Рисунок 32 – Прямоугольник с 1-м объектом, в котором не нужно выполнять разделение

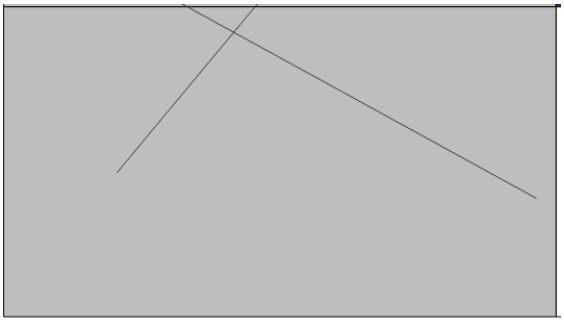


Рисунок 33 – Прямоугольник с общей точкой пересечения, в котором не нужно выполнять разделение

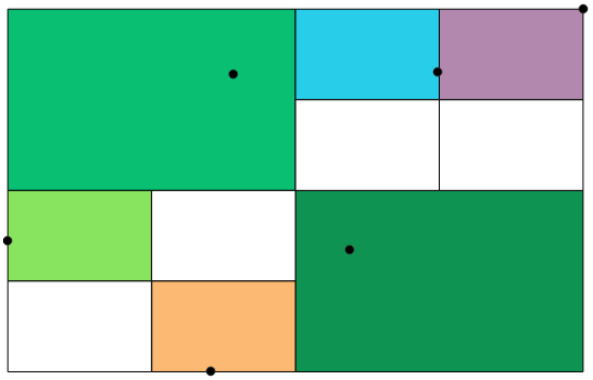


Рисунок 34 – Пример завершения рекурсии

В результате получается карта, разделенная на прямоугольники, включающие в себя один или несколько объектов. На рисунках 35 – 38 приведён пример итерационного процесса формирования прямоугольников.

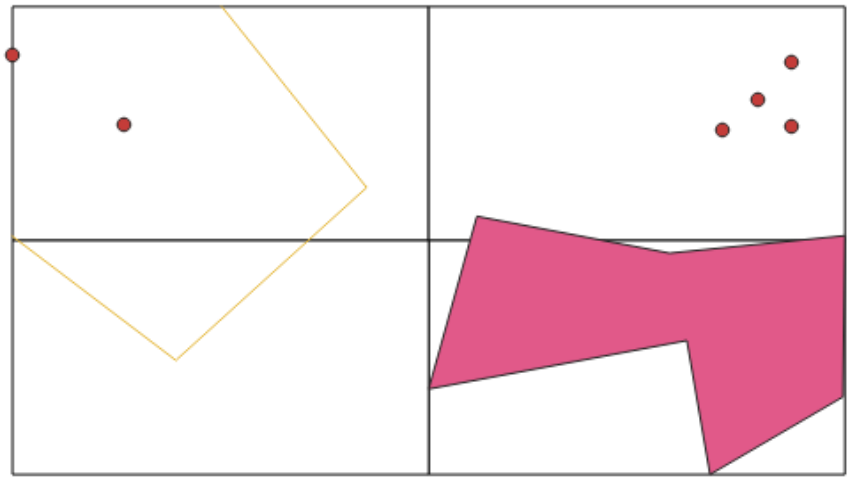


Рисунок 35 – Итерационный процесс формирования прямоугольников

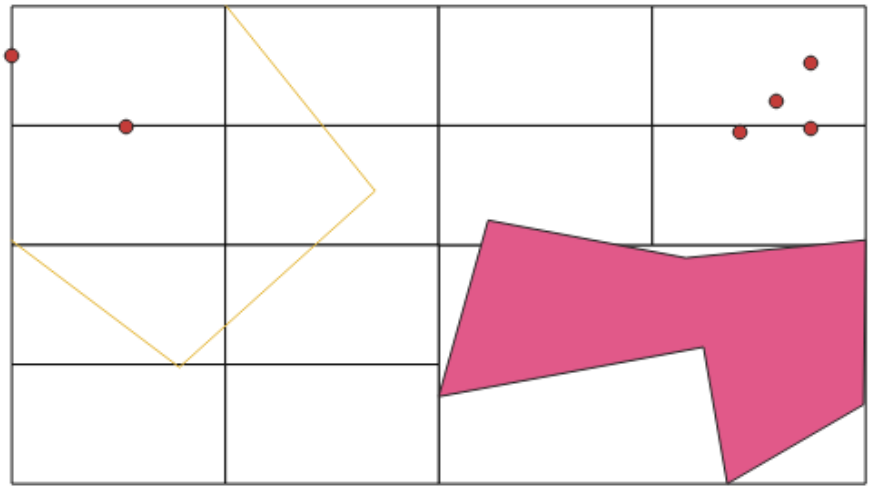


Рисунок 36 – Итерационный процесс формирования прямоугольников

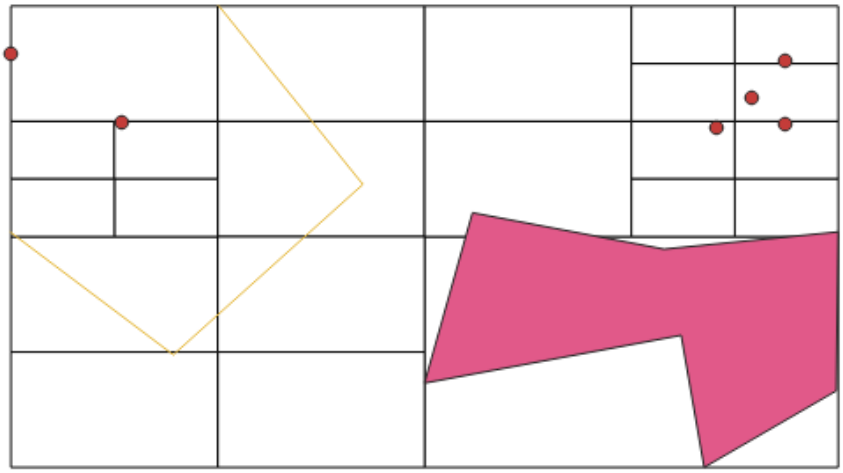


Рисунок 37 – Итерационный процесс формирования прямоугольников

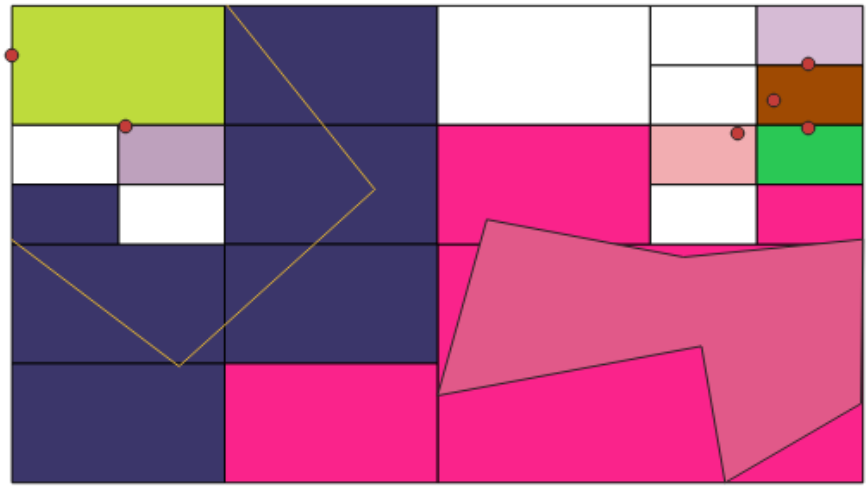


Рисунок 38 – Итерационный процесс формирования прямоугольников

Вызывается функция индексации. Запускается итерация по всем объектам и прямоугольникам. Затем проверяется условие на наличие пересечения объекта с прямоугольником. Если имеется пересечение, то сохраняется его индекс. Примерный вид индекса – «2.1.3.4». Данная запись означает, что объект пересекает 2-ой прямоугольник после 1-го разделения, 1-ый прямоугольник после 2-го разделения, 3-ий прямоугольник 3-го разделения и 4-ый прямоугольник 4-го разделения. Иными словами – это путь разделения до выбранного прямоугольника.

После прохождения цикла по прямоугольникам в список заносится название слоя объекта, id объекта и строка индексов, где эти элементы разделены символом «_», а каждый индекс разделен символом «;». Далее цикл повторяется и проходит по всем объектам. На рисунке 39 представлена визуальная индексация прямоугольников.

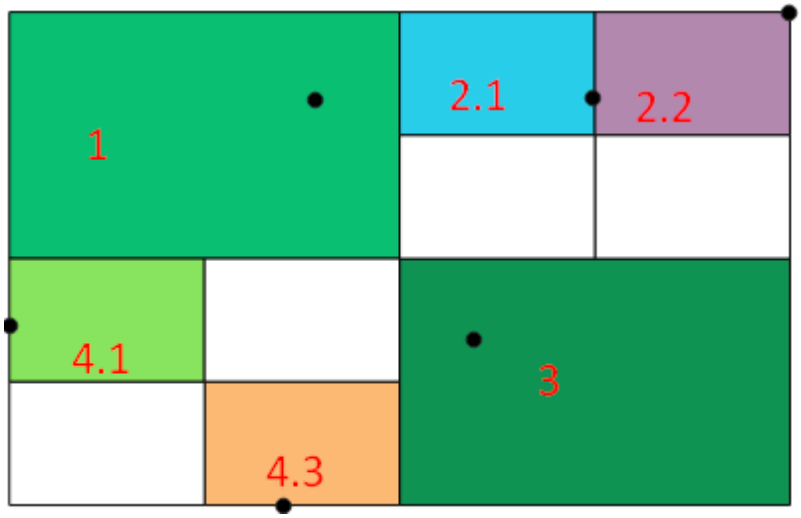


Рисунок 39 – Визуальная индексация прямоугольников

На последнем этапе вызывается функция установки цвета прямоугольников. Сущность ее заключается в следующем:

– запускаем цикл по всем объектам:
– находим его по названию слоя и id в списке;
– обрабатываем строку индексов;

– у всех прямоугольников с этими индексами случайно задаем цвет в формате rgb;

– прямоугольникам с общими точками (одинаковыми индексами) задается серый цвет.

На рисунке 40 представлена установка цветов прямоугольникам.

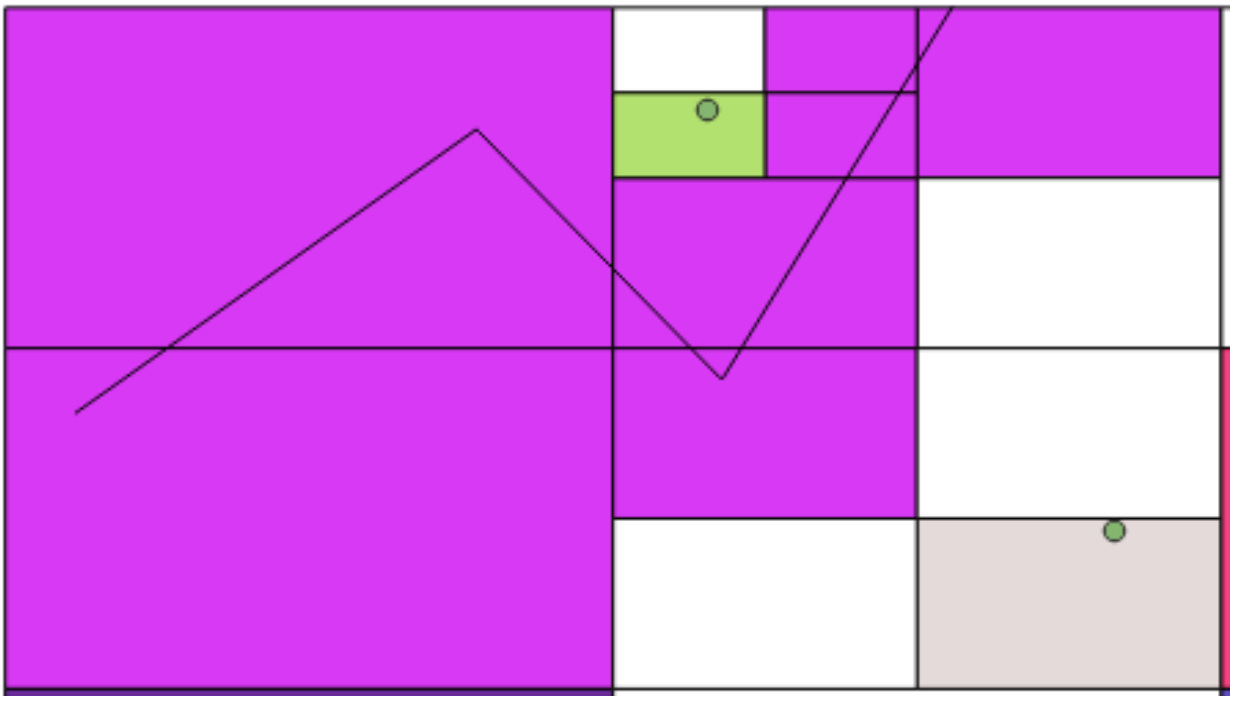


Рисунок 40 – Одноцветное выделение прямоугольников одного объекта

Следующим этапом разработки является нахождение пересекающихся объектов, расположенных на разных слоях. Можно заметить, что если объекты пересекаются, то они находятся в одном индексе, но в некоторых ситуациях объекты, расположенные на одном слое, так же находятся в одном индексе, что для нашей ситуации некорректно. Для этого необходимо правильно обработать список индексов. На рисунке 41 представлен пример списка индексов.

Рисунок 41 – Список индексов

Рассмотрим один элемент из этого списка поподробнее. На рисунке 42 представлен элемент списка индексов.

admlin1000_34_1.3.3.1.4;1.3.3.4.1;1.3.4.3;4.2.1.1;4.2.1.2;

Рисунок 42 – Элемент списка индексов.

Разделим данный элемент по знаку нижнего подчеркивания («_»). Первая часть, расположенная до первого нижнего подчеркивания, означает имя слоя. Вторая часть – между двумя нижними подчеркиваниями – id объекта в слое, указанного в первой части. Третья часть, расположенная после нижнего подчеркивания, является списком идентификаторов индексов, в которых расположен объект, id которого указан во второй части элемента.

Далее необходимо найти разные объекты, имеющие как минимум один общий индекс, но расположенные на разных слоях. Разделим список индексов на несколько других списков по имени слоя. А так как в начале работы алгоритма мы задали список слоёв, участвующих в индексировании это не составит труда. Теперь сравнив полученные списки по идентификаторам индексов, найдём id объектов, которые пересекаются.

Следующим этапом необходимо исправить данное пересечение. В процессе разработки было придумано несколько вариантов. Одним из вариантов было придумать смещение объекта на комплексированном слое. Но с этим вариантов возникло множество вопросов. В первую очередь надо было придумать в какую сторону смещать данный объект. Но даже если можно было решить данный вопрос, неизвестно на какое расстояние смещать объект. Потому что в процессе смещения сложно отследить пересекаются ли в данный момент объекты. Тогда бы каждую итерацию смещения пришлось бы заново запускать процесс индексации, что для больших карт происходит не быстро. В итоге, от концепции что-либо делать с объектом комплексированного слоя пришлось отказаться.

Было решено производить манипуляции с объектом основной карты, той карты, куда изначально переносились объекты.

Так как объекты основной карты достаточно большие относительно объектов перенесённого слоя, было принято решение делать вырезы в объектах большой карты. Эти вырезы будут незначительно влиять на общий фон карты, но будут поддерживать основную концепцию разрабатываемого алгоритма, а именно сохранять топологические отношения между объектами. На рисунке 43 представлена простейшая ситуация.

Рассмотрим процесс создания выреза в объектах. Все объекты на карте (кроме точечных) представляют собой набор линейных объектов. Поэтому были найдены точки пересечения объекта перенесённого слоя и слоя исходной карты. На рисунке 44 смоделированы точки пересечения. Найденные точки необходимо немного сместить от объекта перенесённого слоя. Эти точки понадобятся нам в дальнейшем. На рисунке 45 представлены смещенные точки. Далее мы находим все точки объекта перенесённого слоя, попадающие в плоскость объекта исходной карты. На рисунке 46 представлены эти точки. С помощью этих точек и будет производиться вырез, но с начала их необходимо сместить, а точнее сделать их копии с некоторым смещением относительно центра масс объекта

перенесённого слоя. Для этого воспользуемся методом деления отрезка в данном отношении. Суть данного метода заключается в следующем. Если известны две точки плоскости $A(x_A, y_A)$ и $B(x_B, y_B)$, то координаты точки $M(x_M, y_M)$, которая делит отрезок AB в отношении λ, выражаются формулами 5 и 6:

$$x_M = \frac{x_A + \lambda x_B}{1+\lambda}, \quad (5)$$

$$y_M = \frac{y_A + \lambda y_B}{1+\lambda}. \quad (5)$$

Подставив вместо точки A точку центра масс объекта, вместо точки M каждую точку объекта, находящуюся в плоскости большого объекта, мы сможем рассчитать точку B то есть точку куда необходимо сместить точку M. Данная ситуация смоделирована на рисунке 47. Расположим все полученные точки в нужном порядке и вставим этот список точек в объект большой карты. Произойдет изменение геометрии и в QGIS можно будет наблюдать необходимый вырез. Смоделированный вырез представлен на рисунке 48. На рисунках 43 – 48 представлен лишь макет производимого выреза, тестируемые данные могут отличаться от представленных, из-за этого вырез может быть выглядеть другим.

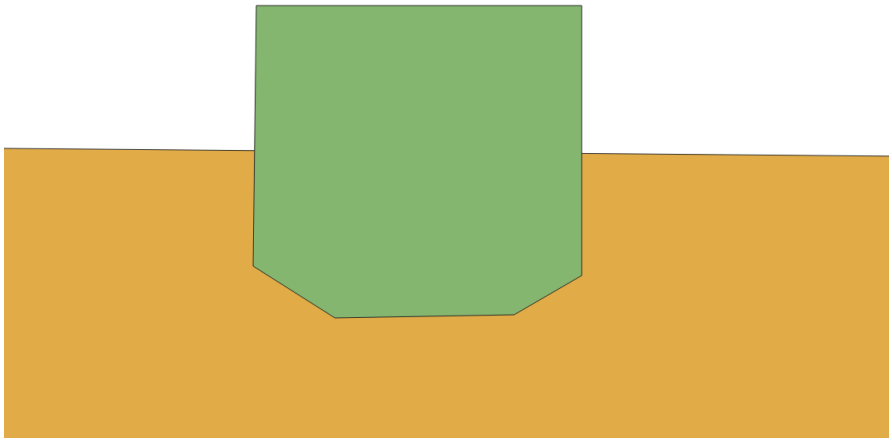


Рисунок 43 – Комплексированный объект пересекает другой объект

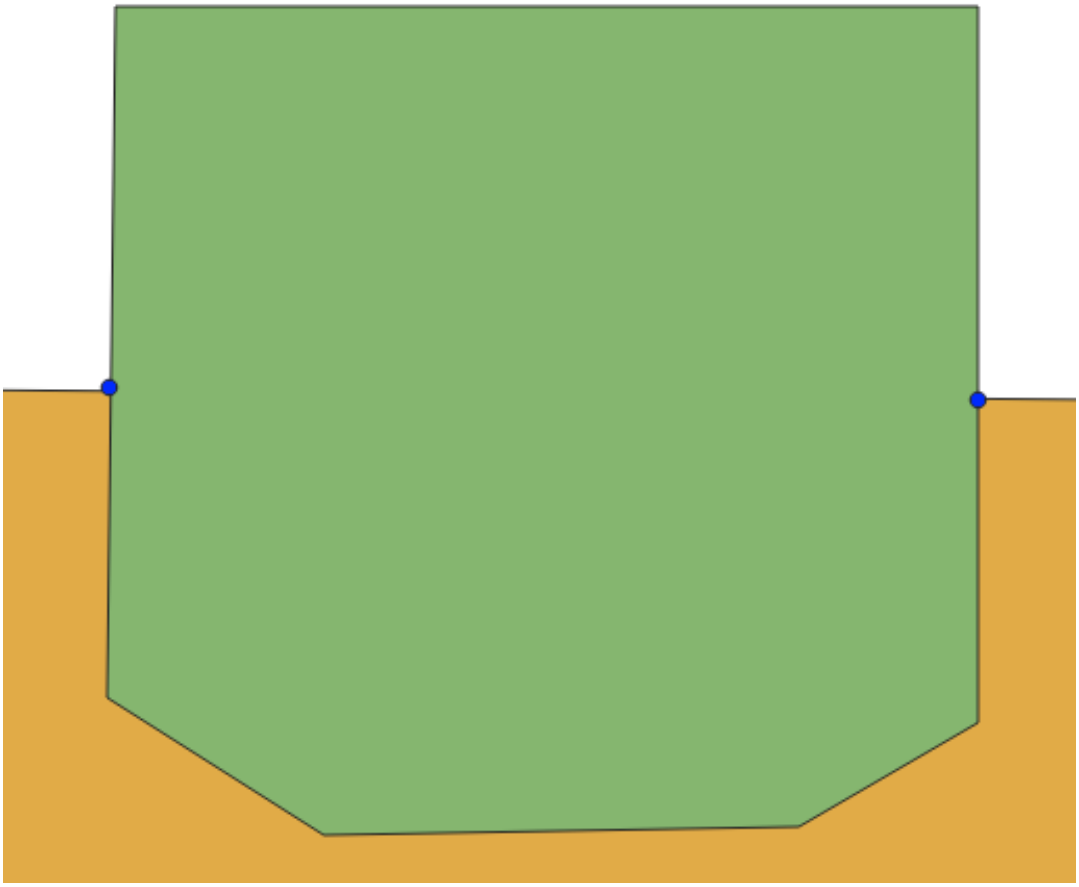


Рисунок 44 – Точки пересечения

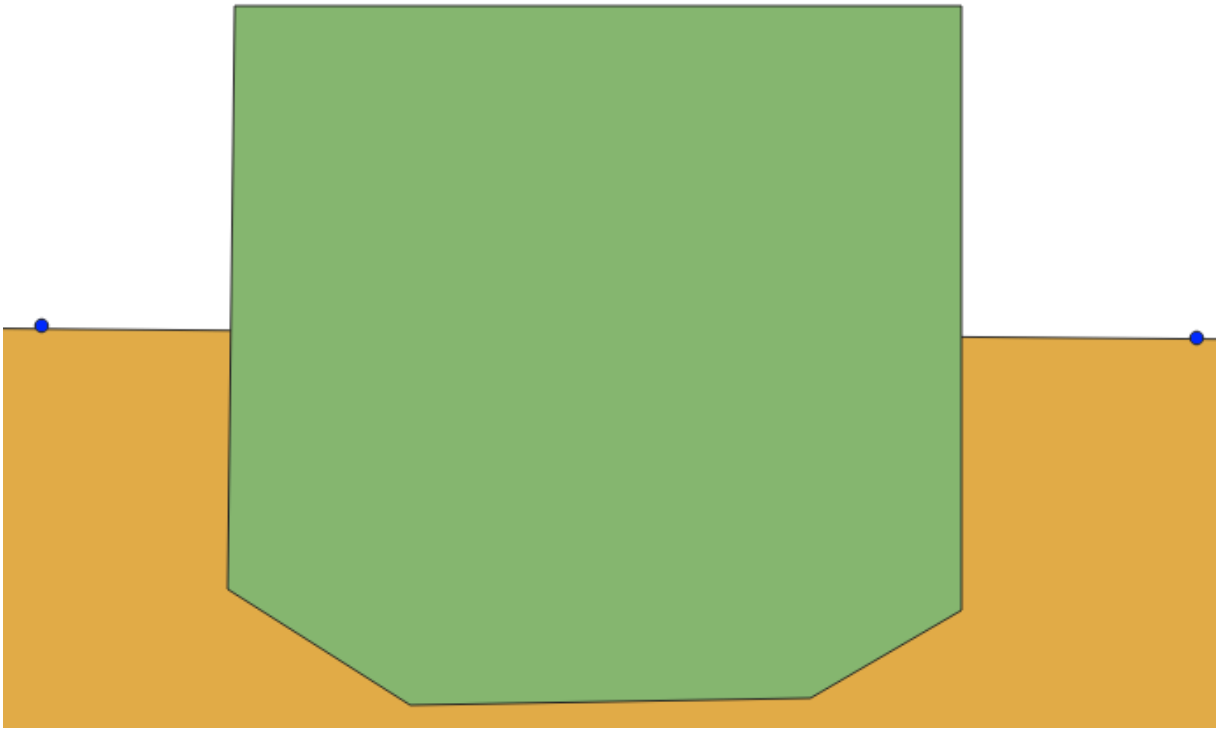


Рисунок 45 – Смещённые точки

Рисунок 46 – Точки, попадающие в плоскость объекта

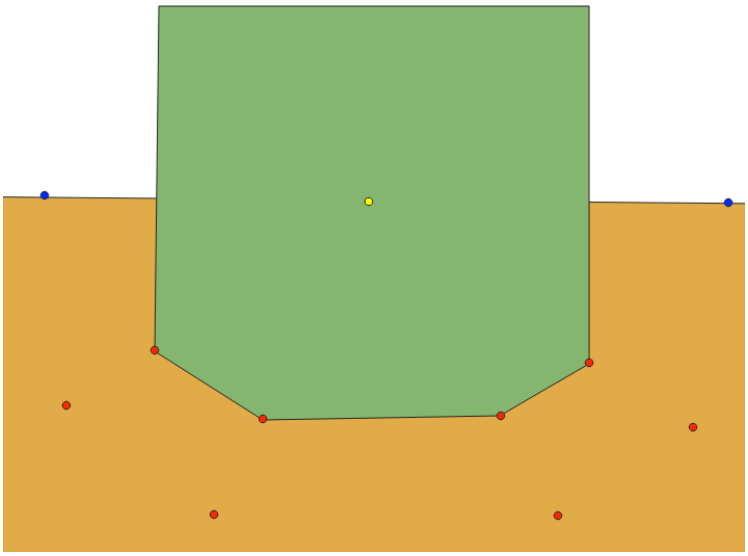


Рисунок 47 - Смещенные точки

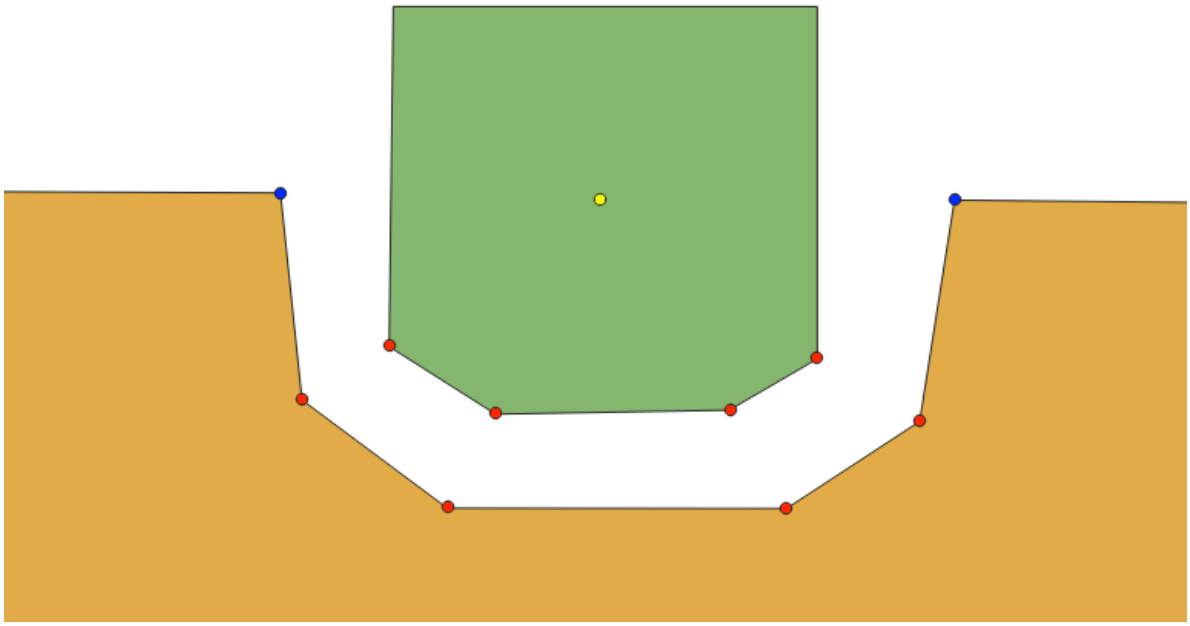


Рисунок 48 - Вырез

2.3.2 Структура файла

Файл indexing_of_elements.py содержит основной класс для работы алгоритма под названием Index_of_element. В свою очередь класс содержит несколько основных функций. Остановимся на них поподробнее.

Функция __init__ - воспроизводится во время создания экземпляра класса Index_of_element. В ней инициализируются такие переменные:

– project – содержит информацию о текущем проекте QGIS;

– layers_name – список слоёв, участвующих в индексировании;

– coordinate_system – содержит информацию о системе координат текущего проекта.

Во время работы функции get_points_main_rectangle происходит считывание границ слоёв, участвующих в индексировании. После этого формируются точки главного прямоугольника, таким образом, чтобы этот прямоугольник покрывал все объекты на слоях.

Функция add_temporary_layer создает временный слой с прямоугольником.

Функция separation_on_rect смотрит на объекты внутри текущего прямоугольника и делит его на 4 если там более одного объекта.

Функция separation – основная рекурсивная функция, выполняющая индексирование объектов на слоях.

Функция fill_index создает список индексов рассматриваемый выше.

Функция set_color_rects обрабатывает полученный список индексов и на его основе изменяет цвета прямоугольников.

Функция fix_cross устраняет пересечение объектов, делая вырез, рассмотренный ранее.

Также в классе Index_of_element имеются некоторые вспомогательные функции, не представляющие интереса. Они необходимо только для удобства работы над алгоритмом и скриптом в целом.

3 Исследование работы алгоритмов

3.1 Тестирование алгоритма комплексирования векторных данных

В ходе бакалаврской работы был протестирован данный алгоритм на реальных данных.

Разработанный алгоритм работает с любой системой координат. В качестве данных для тестирования была выбрана карта города Устюг, имеющая систему координат Гаусса-Крюгера (EPSG:20008). Она является случаем специального применения проекции Гаусса-Крюгера и используется в Евразии, включая Россию и Китай. Данная система координат делит мир на зоны шириной в шесть градусов. В каждой зоне коэффициент масштаба равен 1, а смещение по долготе равно 500 000 метрам. Загрузим слой с линиями и полигонами. Данные слои представлены на рисунке 49.

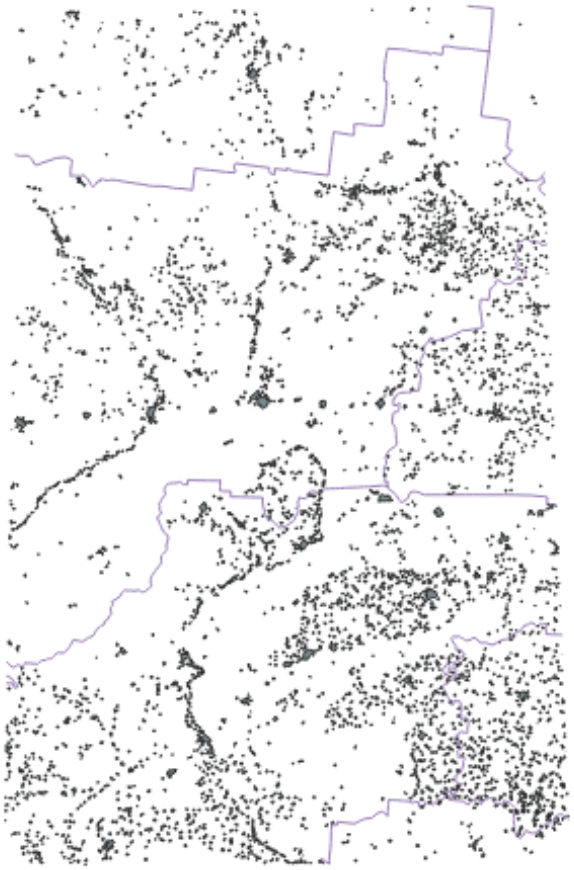


Рисунок 49 – Карта города Устюг

Добавим базовые точки для будущих треугольников. Для второй карты расположение базовых точек специально задано с различиями относительно базовых точек первой карты. Всего вершин для треугольников получилось 58 штук. Расставленные базовые точки представлены на рисунках 50 и 51. На рисунке 52 представлено сравнение расположения базовых точек.

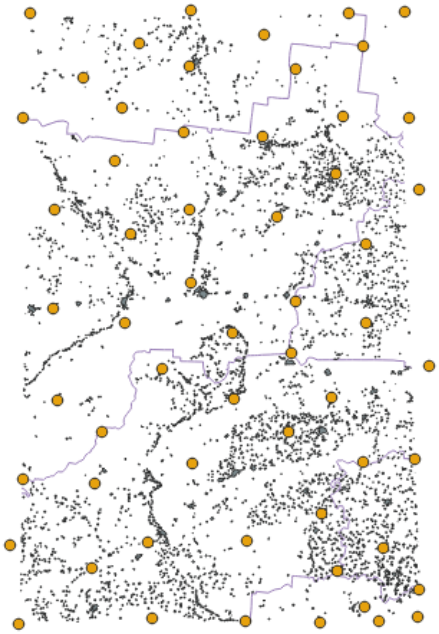


Рисунок 50 – Базовые точки первой карты

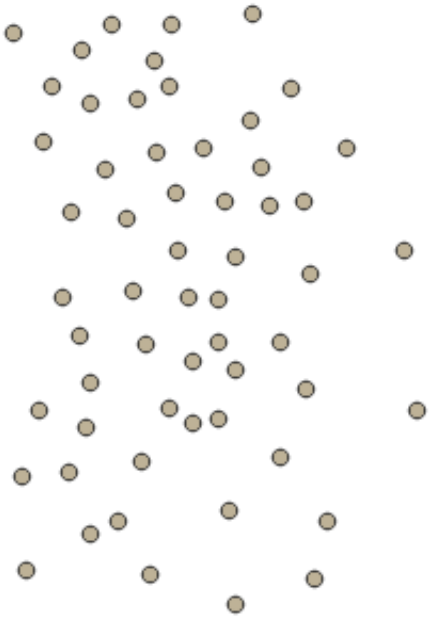
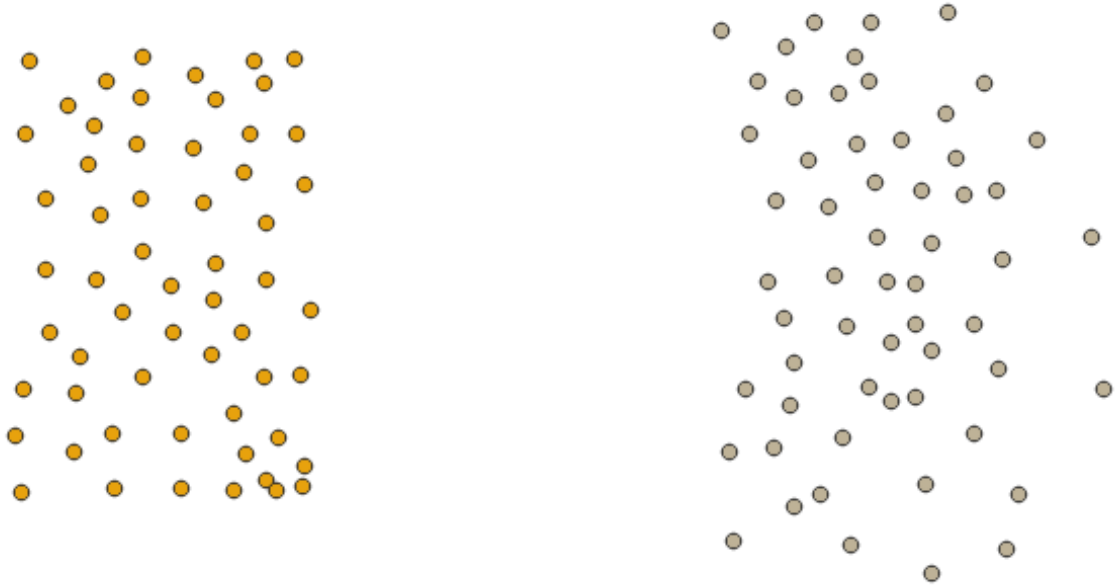


Рисунок 51 – Базовые точки второй карты



а) б)

Рисунок 52 - Сравнение базовых точек двух карт. а – базовые точки первой карты; б – базовые точки второй карты

Проведем триангуляцию и перенесём объекты с первой карты на вторую. Результат переноса представлен на рисунке 53.

а) б)

Рисунок 53 – Перенос объектов. а – первая карта с треугольниками; б – вторая карты с треугольниками

Рассмотри поближе области, выделенные красным на рисунке 53. Области представлены на рисунках 54 и 55.

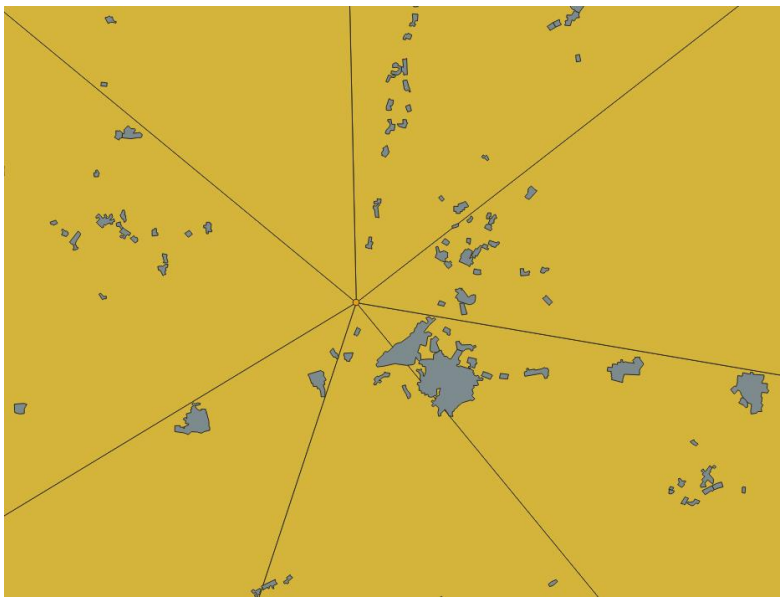


Рисунок 54 – Красная область на первой карте

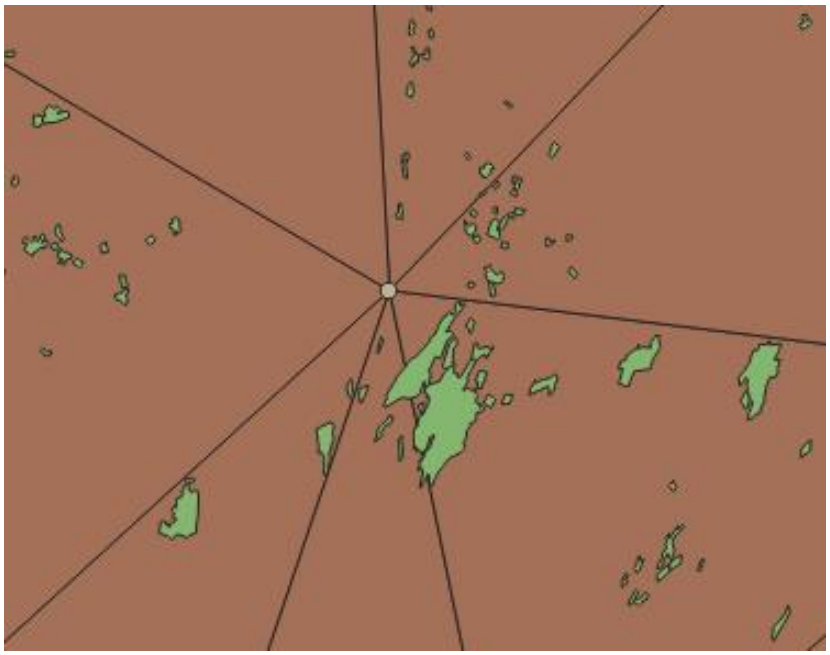


Рисунок 55 – Красная область на второй карте

По рисункам видно, что перемещенные объекты деформировались. Следовательно барицентрические координаты работают как положено, и у объектов есть привязка к треугольнику, в котором они находятся.

3.2 Тестирование алгоритма индексирования

Алгоритм построения пространственного индекса был протестирован на реальных примерах.

Разработанный алгоритм может работать на любой системе координат.

В качестве первых тестовых данных была выбрана карта города Ярославль. На карте имеются точечные и полигональные объекты. Количество объектов равняется 64, суммарное количество точек, из которых состоят эти объекты – 741. На рисунке 56 представлена карта города Ярославль.

Рисунок 56 – Исходная карта Ярославль

Далее был запущен алгоритм с глубиной рекурсии равной 10. Это значит, что разделение одного прямоугольника не произойдет более 10 раз. Результаты реализации алгоритма показаны поэтапно для лучшей видимости разделения прямоугольников. Красная рамка показывает в каких пределах будет отображен последующий рисунок с уменьшенным масштабом. На рисунках 57 – 60 представлен результат работы алгоритма с глубиной рекурсии 10.

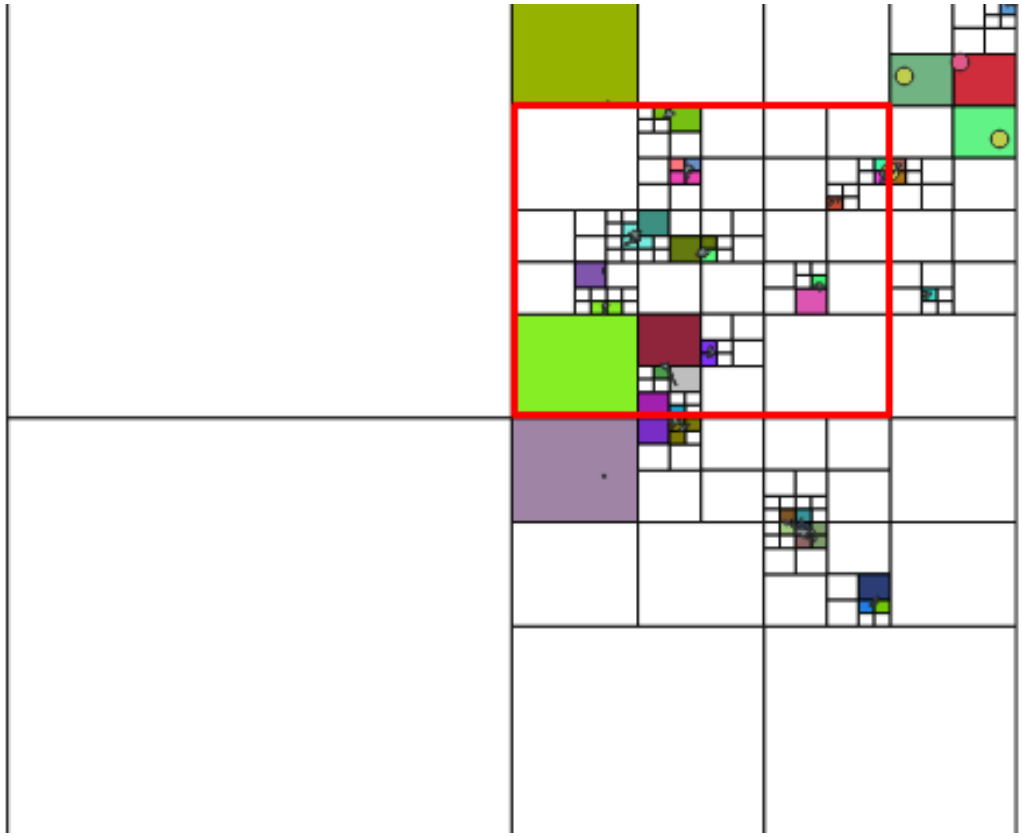


Рисунок 57 – Результат работы алгоритма с глубиной рекурсии равной 10

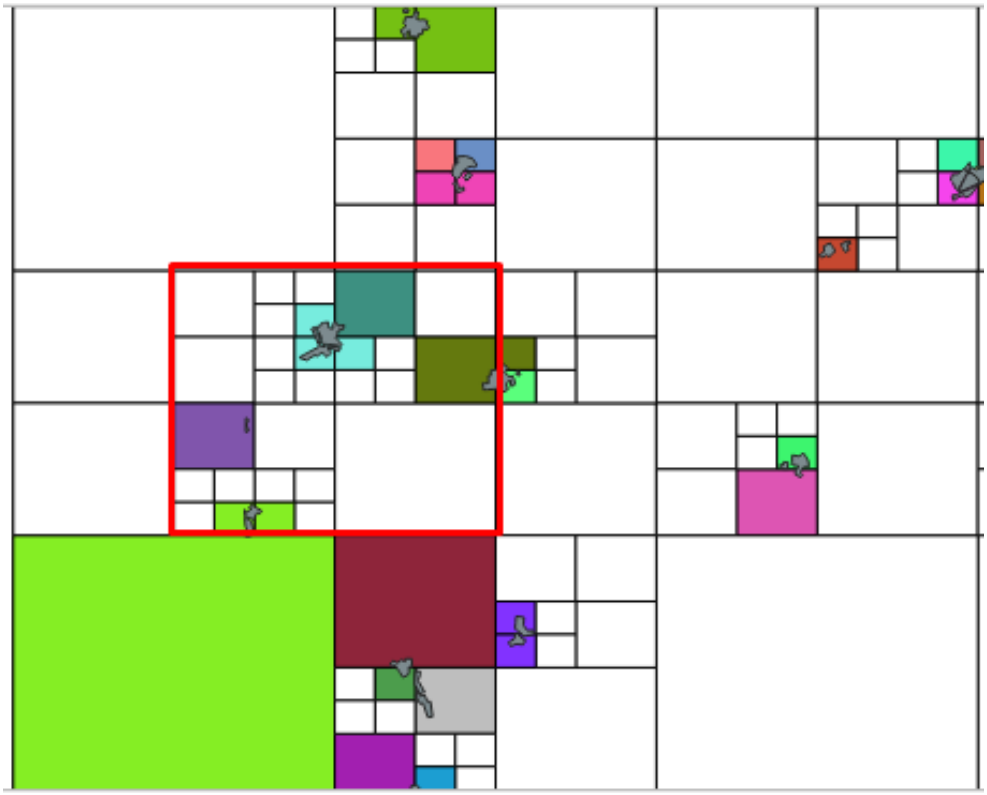


Рисунок 58 – Увеличенный Результат работы алгоритма с масштабом 1:5000000 и глубиной рекурсии равной 10

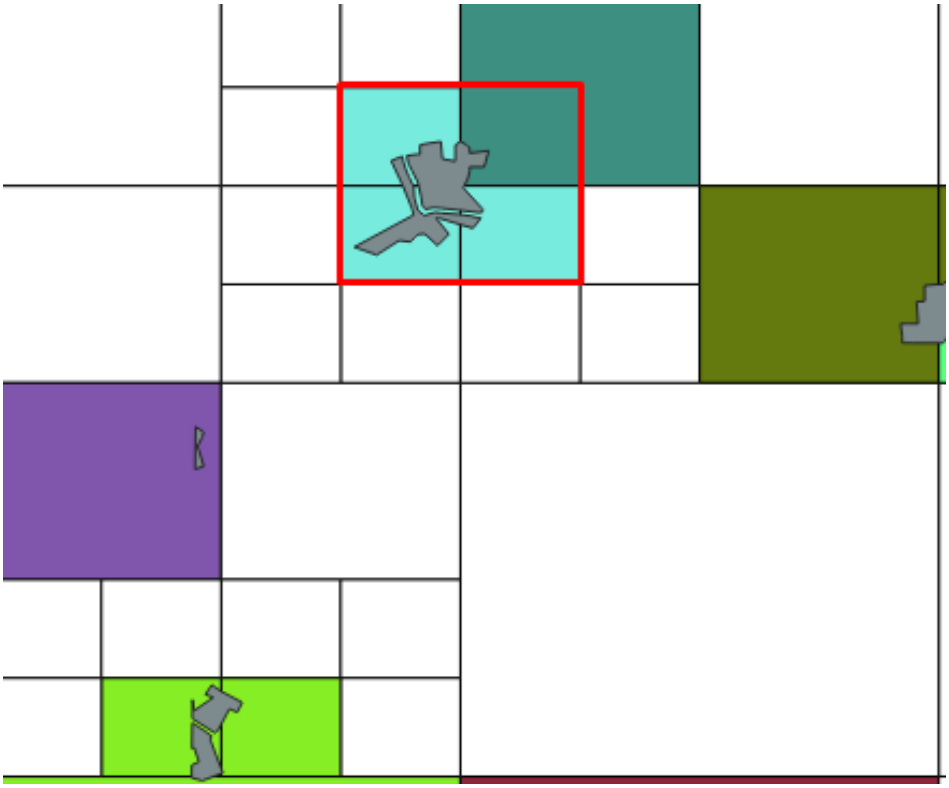


Рисунок 59 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 10

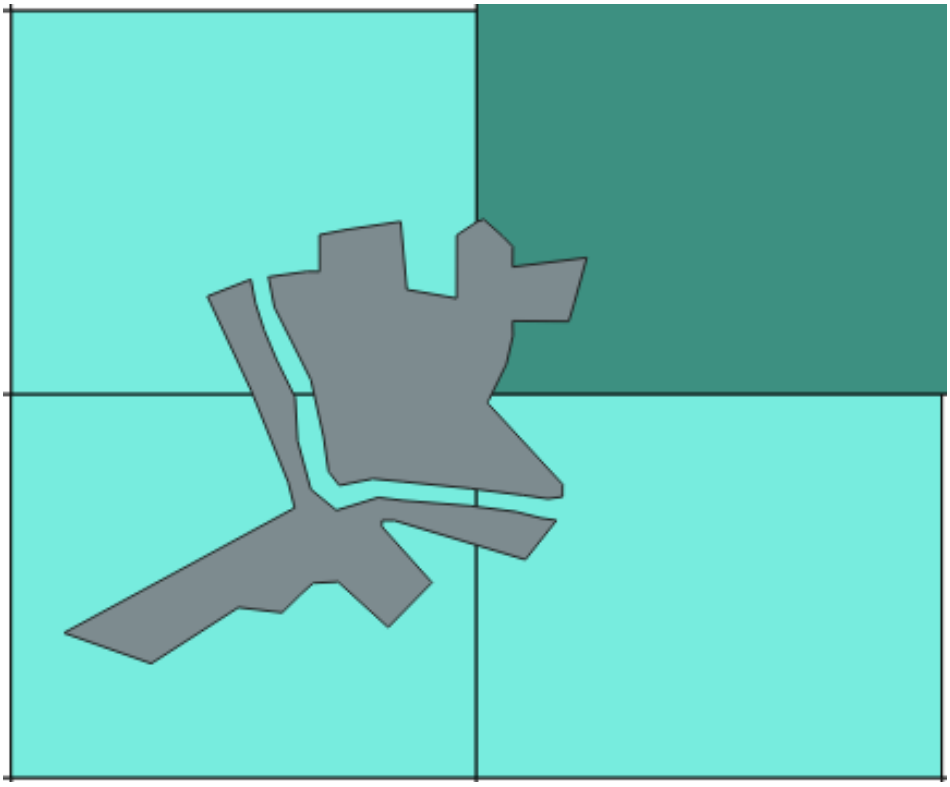


Рисунок 60 – Результат работы алгоритма с масштабом 1:100000 и глубиной рекурсии равной 10

В таблице 1 представлен пример индексации объектов с глубиной рекурсии равной 10.

Таблица 1 – Пример индексации объектов с глубиной рекурсии равной 10

ID	Список индексов
1	2.2.2.2.2.4.1.3.1
2	2.2.2.2.2.4.2.1.4.1; 2.2.2.2.2.4.1.2.3.2; 2.2.2.2.2.4.1.2.2.3
3	2.2.2.2.2.4.2.1.1; 2.2.2.2.2.4.2.1.4.1; 2.2.2.2.2.4.1.2.3.2; 2.2.2.2.2.4.1.2.2.3
4	2.2.2.2.2.4.2.1.3; 2.2.2.2.2.4.2.2.4.1; 2.2.2.2.2.4.2.2.4.4
5	2.2.2.2.2.4.2.2.4.4

На рисунке 61 имеются числовые подписи для каждого объекта, Эти числа являются идентификаторами объектов и соответствуют графе «ID» таблицы 1, которая была представлена выше.

На рисунке 61 можно увидеть, что при глубине рекурсии равной 10, алгоритм выполняется не верно. Так как объекты, к примеру 2 и 3, расположены в одном индексе.

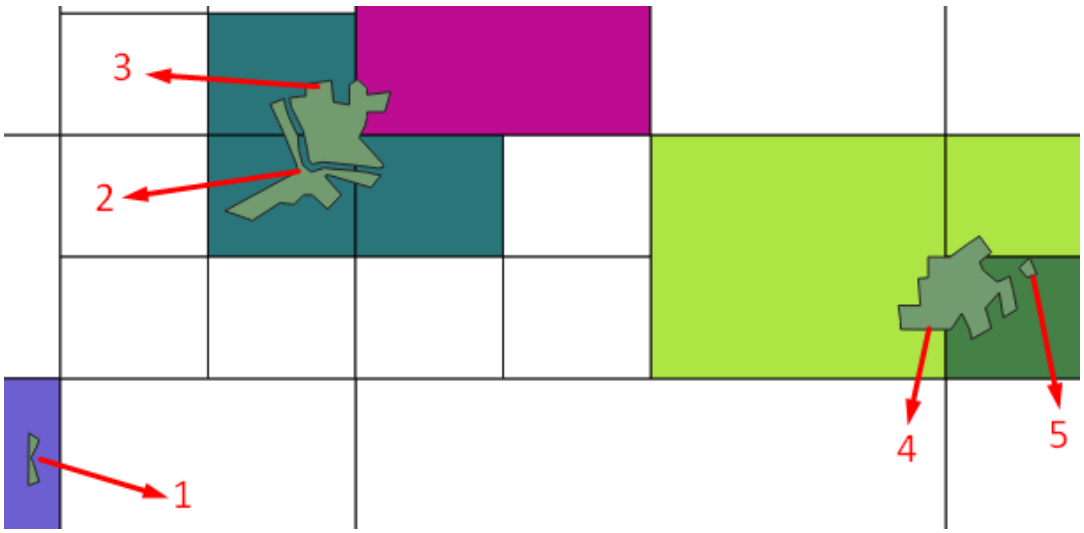


Рисунок 61 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 10

После этого реализованный алгоритм был протестирован с глубиной рекурсии равной 15. На рисунках 62 – 64 представлен результат работы алгоритма с рекурсией равной 15 в разных масштабах.

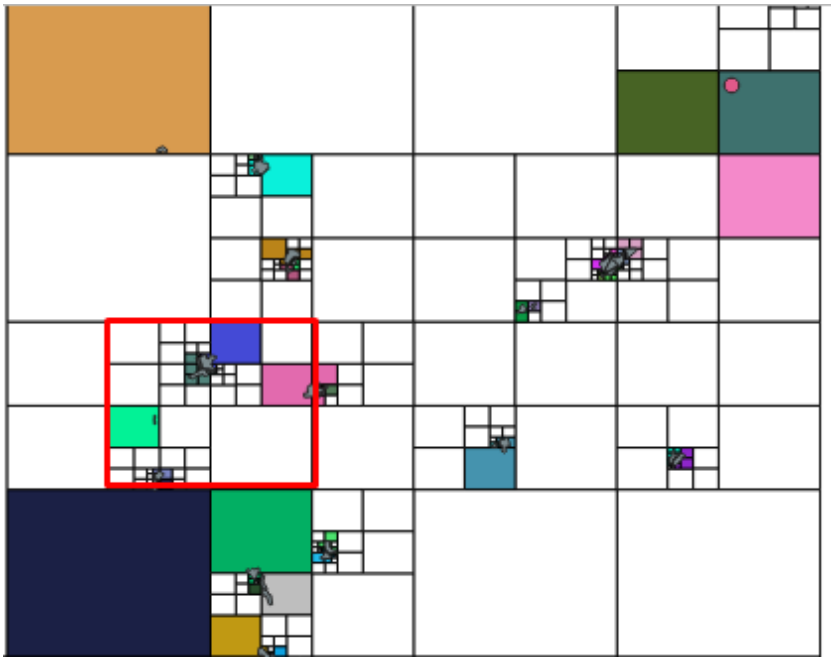


Рисунок 62 – Результат работы алгоритма с масштабом 1:5000000 и глубиной рекурсии равной 15

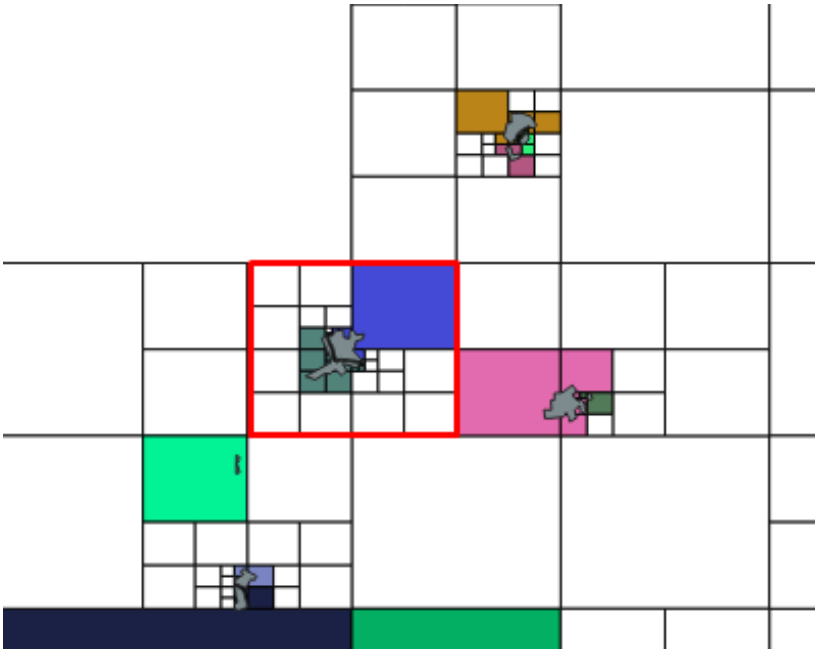


Рисунок 63 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 15

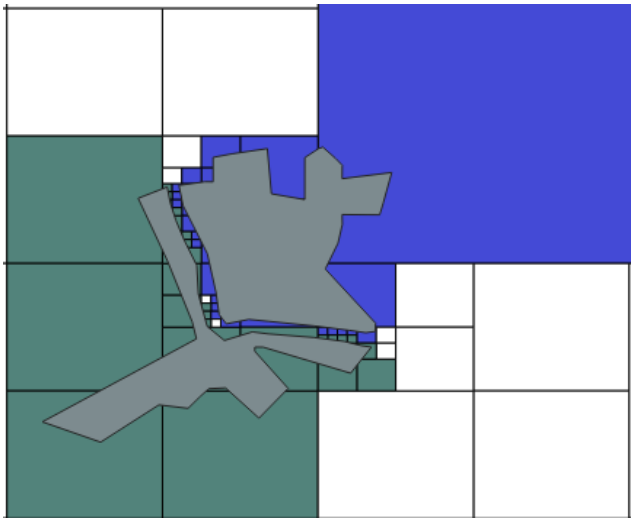


Рисунок 64 – Результат работы алгоритма с масштабом 1:100000 и глубиной рекурсии равной 15

В таблице 2 представлен пример индексации объектов с глубиной рекурсии равной 15. На рисунке 65 имеются число числовые подписи для каждого объекта, Эти числа являются id объектов и соответствуют графе «ID» таблицы 2.

Таблица 2 – Пример индексации объектов с глубиной рекурсии равной 15

ID	Список индексов
1	2
1	2.2.2.2.2.4.1.3.1
2	2.2.2.2.3.2.4.1.2.1.3.3.2.4.4; 2.2.2.2.3.2.4.1.2.1.3.3.2.4.3; 2.2.2.2.3.2.4.1.2.1.3.3.2.4.2; 2.2.2.2.3.2.4.1.2.1.3.3.2.4.1; 2.2.2.2.3.2.4.1.2.1.3.3.2.1.4; 2.2.2.2.3.2.4.1.2.1.3.3.2.1.3; 2.2.2.2.3.2.4.1.2.1.3.3.2.1.2; 2.2.2.2.3.2.4.1.2.1.3.3.2.1.1; 2.2.2.2.3.2.4.1.2.1.3.2.4.2.4; 2.2.2.2.3.2.4.1.2.1.3.2.4.2.3; 2.2.2.2.3.2.4.1.2.1.3.2.4.2.2; 2.2.2.2.3.2.4.1.2.1.3.2.4.2.1; 2.2.2.2.3.2.4.1.2.1.3.2.3.4.4; 2.2.2.2.3.2.4.1.2.1.3.2.3.4.3; 2.2.2.2.3.2.4.1.2.1.3.2.3.4.2; 2.2.2.2.3.2.4.1.2.1.3.2.3.4.1; 2.2.2.2.3.2.4.1.2.1.3.2.2.1.4; 2.2.2.2.3.2.4.1.2.1.3.2.2.1.3;

Продолжение таблицы 2

1	2
	2.2.2.2.3.2.4.1.2.1.3.2.2.1.2; 2.2.2.2.3.2.4.1.2.1.3.2.2.1.1; 2.2.2.2.3.2.4.1.2.1.3.2.1.4.4
3	2.2.2.2.2.3.2.4.3.1.1.3.4.3.2; 2.2.2.2.2.3.2.4.3.1.1.3.4.3.1; 2.2.2.2.2.3.2.4.3.1.1.3.3.1.4; 2.2.2.2.2.3.2.4.3.1.1.3.3.1.1; 2.2.2.2.2.3.2.4.3.1.1.3.2.3.1; 2.2.2.2.2.3.2.4.3.1.1.3.2.2.4; 2.2.2.2.2.3.2.4.3.1.1.3.2.2.1; 2.2.2.2.2.4.2.1.4.1.1.4.2.1; 2.2.2.2.2.3.2.4.3.1.4.4.2.1; 2.2.2.2.2.3.2.4.3.1.4.1.3.4; 2.2.2.2.2.3.2.4.3.1.4.1.3.1; 2.2.2.2.2.3.2.4.3.1.4.1.2.4; 2.2.2.2.2.3.2.4.3.1.4.1.2.2; 2.2.2.2.2.3.2.4.3.1.4.1.2.1; 2.2.2.2.2.3.2.4.3.1.1.3.4.4; 2.2.2.2.2.3.2.4.3.1.1.3.4.2; 2.2.2.2.2.3.2.4.3.1.1.3.4.1; 2.2.2.2.2.3.2.4.3.1.1.3.2.4; 2.2.2.2.2.3.2.4.3.1.1.3.2.1; 2.2.2.2.2.4.1.2.3.2.2.1.2; 2.2.2.2.2.4.1.2.2.3.3.4.3; 2.2.2.2.2.4.1.2.2.3.3.4.2; 2.2.2.2.2.4.1.2.2.3.3.1.3; 2.2.2.2.2.4.1.2.3.2.2.2; 2.2.2.2.2.4.1.2.2.3.3.3
4	2.2.2.2.2.4.2.1.3; 2.2.2.2.2.4.2.2.4.1; 2.2.2.2.2.4.2.2.2.4.4; 2.2.2.2.2.4.2.2.4.4.1.2.3; 2.2.2.2.2.4.2.2.4.4.1.2.4; 2.2.2.2.2.4.2.2.4.4.1.2.1
5	2.2.2.2.2.4.2.2.4.4.2; 2.2.2.2.2.4.2.2.4.4.1.2.2

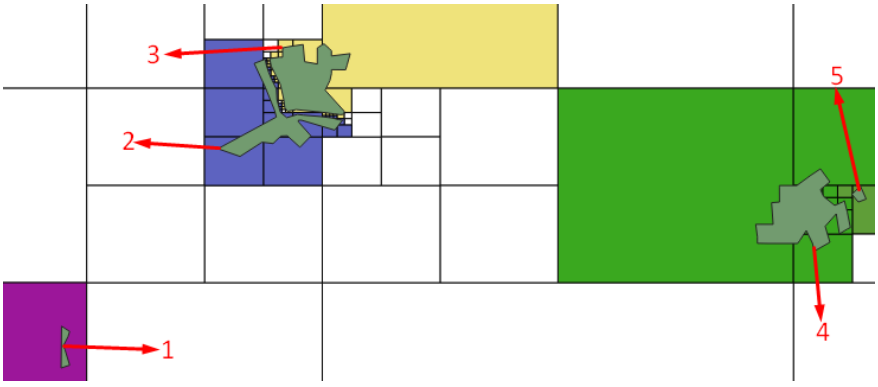


Рисунок 65 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 10

Как мы видим по результатам тестирования большая глубина рекурсии привела к верному результату, но время работы алгоритма значительно увеличивается. При глубине рекурсии равной 10 алгоритм выполняется за 38 секунд, при глубине рекурсии равной 15 – 16 минут 13 секунд.

Во втором примере представлены только линейные объекты. На карте отображены реки и дороги города Ярославль. Количество объектов равняется 44, суммарное количество точек, из которых состоят эти объекты – 423. На рисунке 66 представлена исходная карта. Ниже на рисунках 67 – 69 приведены результаты тестирования данной карты с глубиной тестирования равно 10 в разных масштабах.

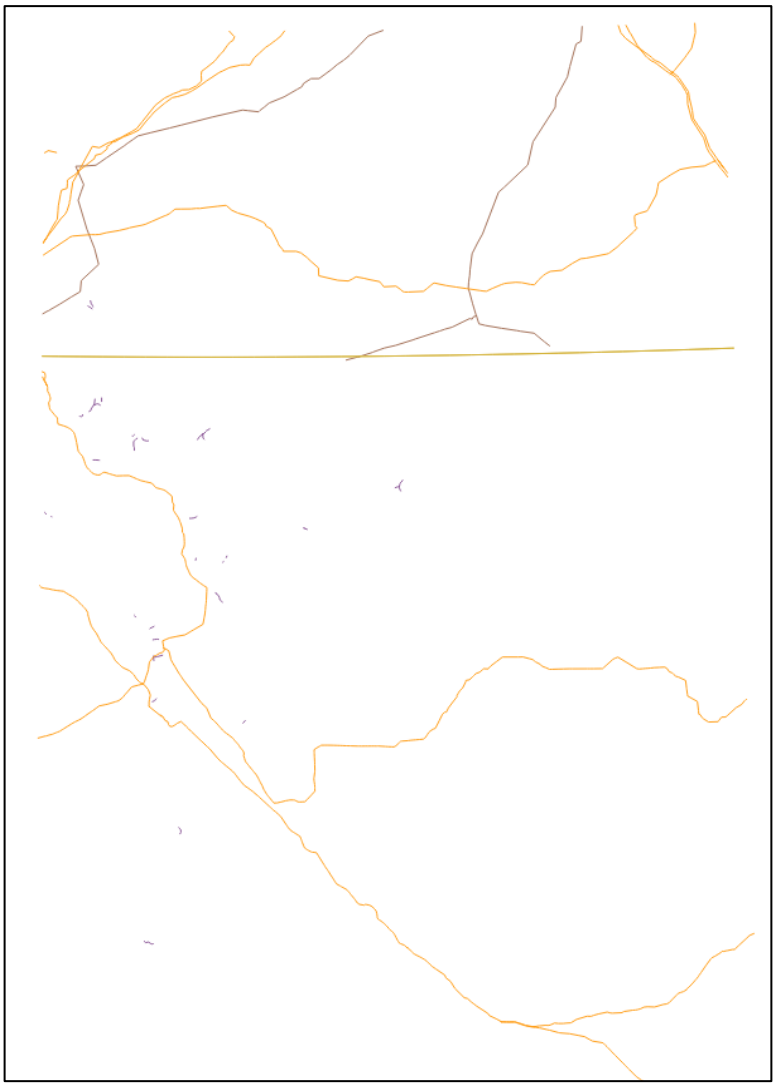


Рисунок 66 – Исходная карта второго тестирования с масштабом 1:1000000

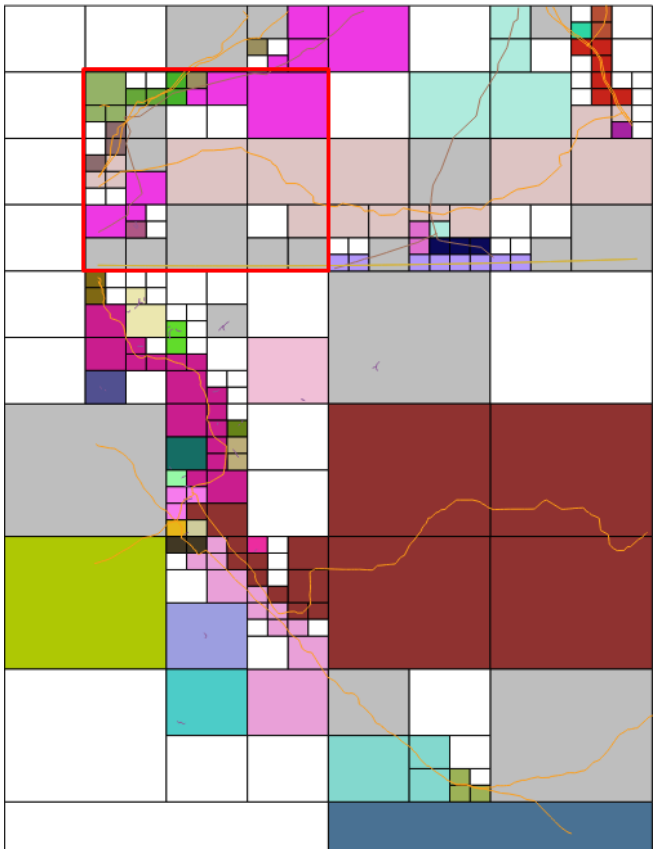


Рисунок 67 – Результат работы алгоритма с масштабом 1:500000 и глубиной рекурсии равной 10

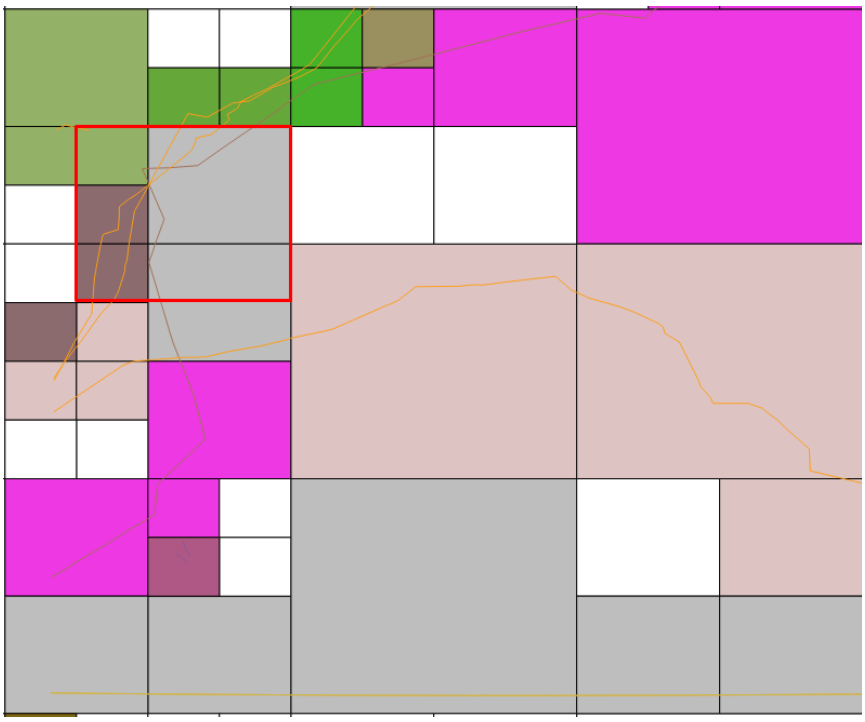


Рисунок 68 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 10

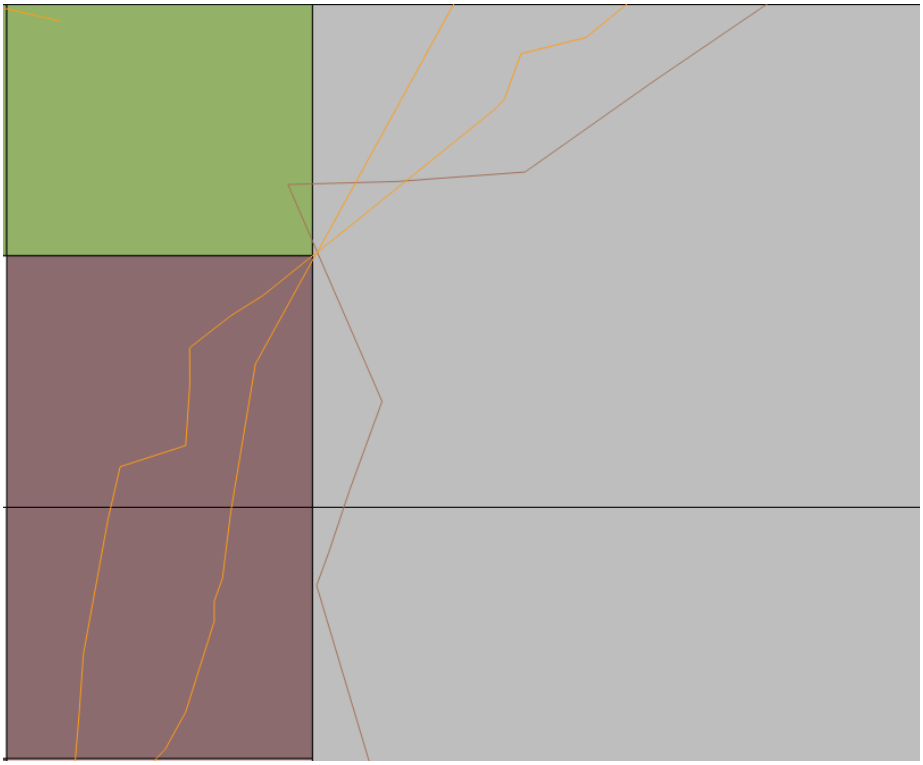


Рисунок 69 – Результат работы алгоритма с масштабом 1:100000 и глубиной рекурсии равной 10

Во втором тестировании также можно заметить, что глубины рекурсии равной 10 недостаточно для верного результата. На рисунках 70 - 72 показан результат работы алгоритма с глубиной рекурсии равной 15.

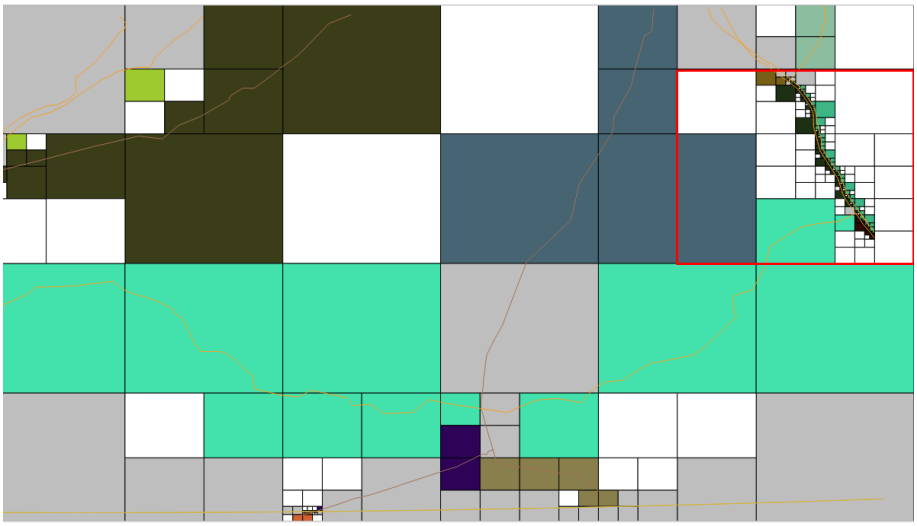


Рисунок 70 – Результат работы алгоритма с масштабом 1:500000 и глубиной рекурсии равной 15

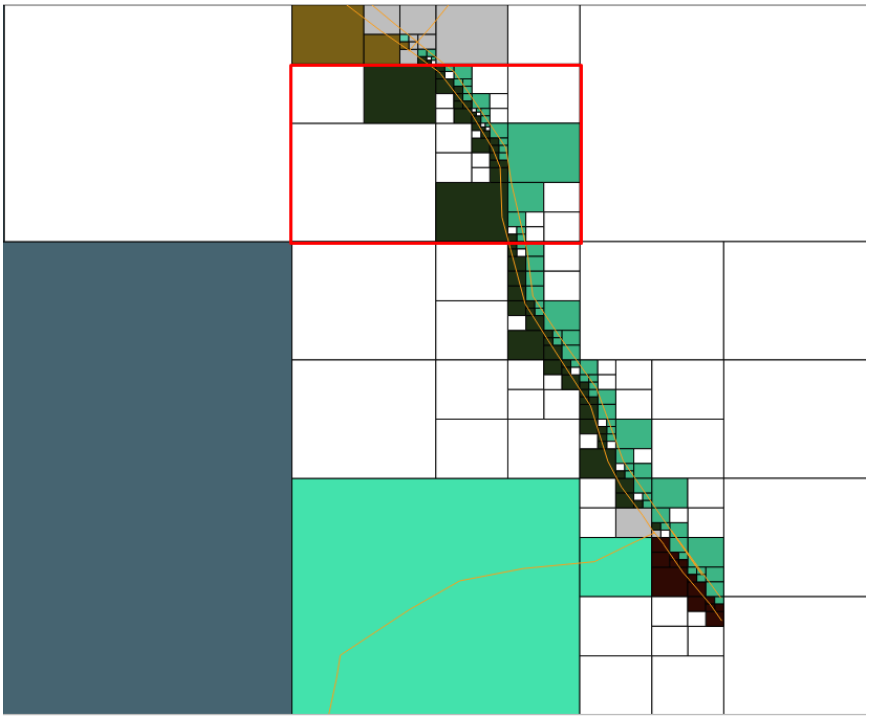


Рисунок 71 – Результат работы алгоритма с масштабом 1:250000 и глубиной рекурсии равной 15

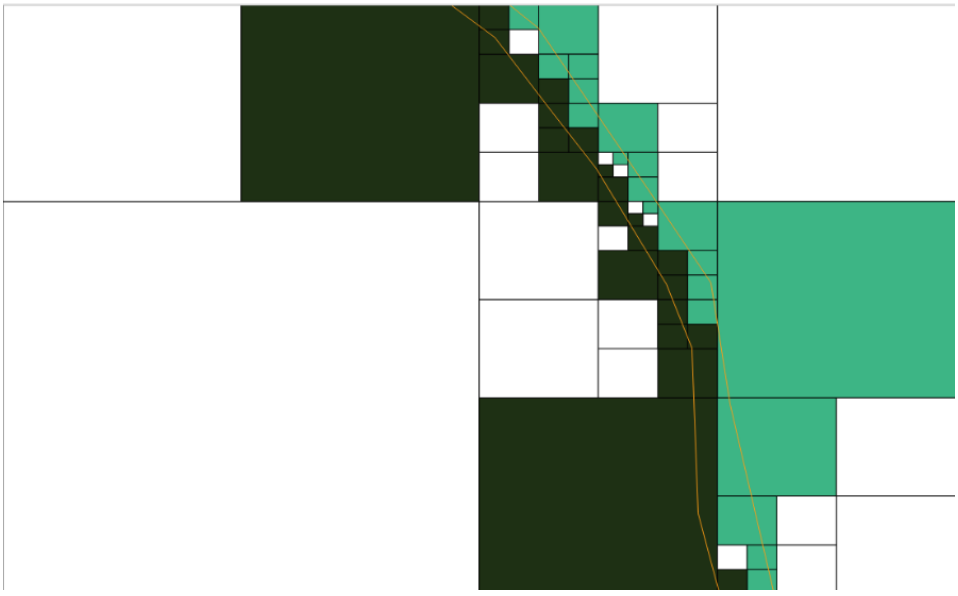


Рисунок 72 – Результат работы алгоритма с масштабом 1:100000 и глубиной рекурсии равной 15

Как видно разделение объектов выполнено правильно. Время работы реализованного алгоритма составило 8 минут 15 секунд. Для сравнения, время работы алгоритма с глубиной рекурсии равной 10 – 1 минута 28 секунд.

3.3 Тестирование алгоритма создания выреза

Для тестирования работоспособности создания выреза на объекте не нашлось реальных данных, поэтому пришлось создать свои специальные тестовые данные.

Первым делом необходимо было придумать ситуацию, в которой будет работать тестируемый алгоритм. Нарисуем две карты, допустим озера. Одна будет со множеством деталей, вторая с наименьшим количеством деталей и будет примерно показывать форму объекта. На рисунке 73 представлена детализированная карта озера. На рисунке 74 представлена карта с меньшим количеством деталей.

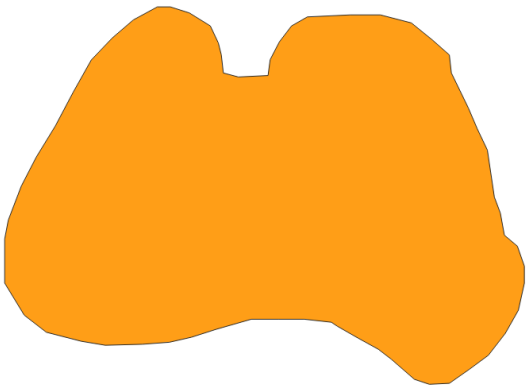


Рисунок 73 – Детализированная карта озера

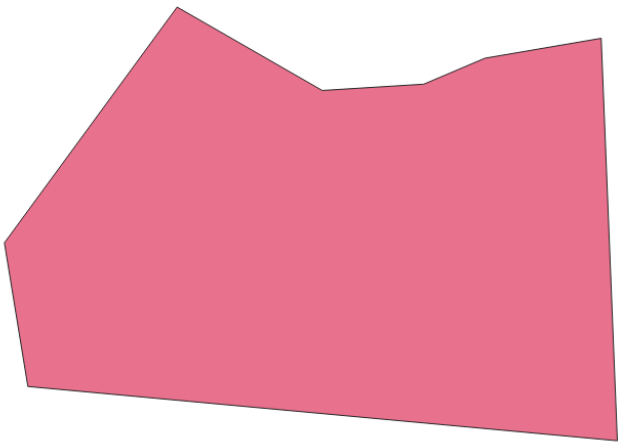


Рисунок 74 – Минималистичная карта озера

Теперь на первую карту необходимо добавить объект, который мы будем комплексировать на другую карту. На рисунке 75 представлена карта с добавленным объектом для переноса.

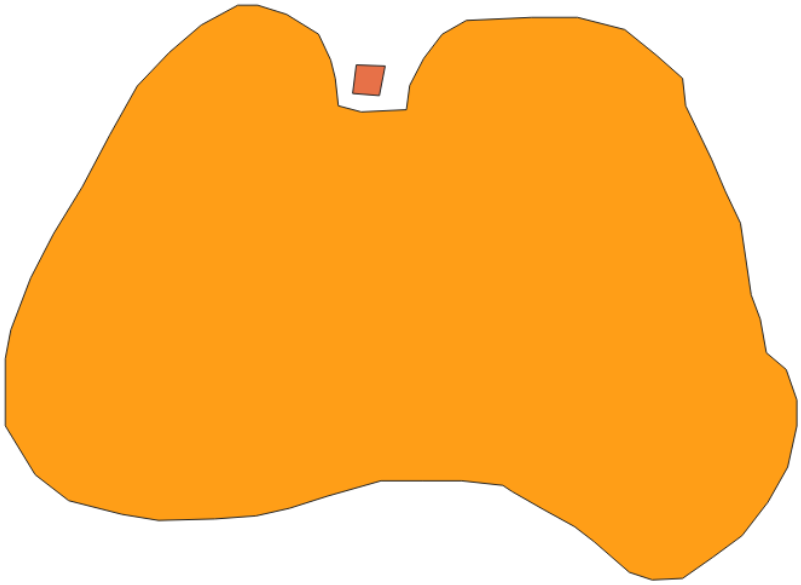


Рисунок 76 – Объект для переноса на другую карту

Для работоспособности алгоритма добавим на обе карты базовые точки для дальнейшей триангуляции на их основе. На рисунка 77 и 78 представлены базовые точки первой и второй карты, соответственно.

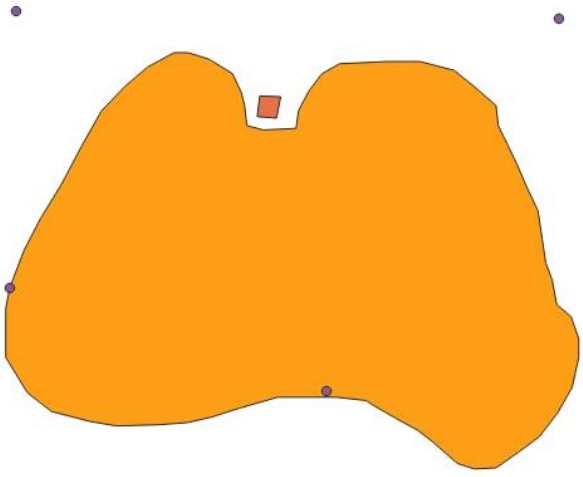


Рисунок 77 – Базовые точки первой карты

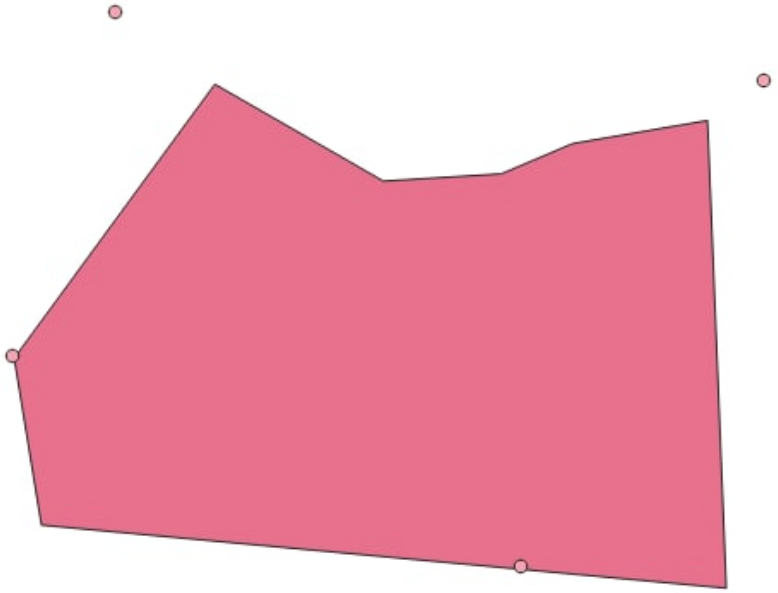


Рисунок 78 – Базовые точки второй карты

Проведем комплексирование объекта, расположенного на первой карте. На рисунке 79 представлены построенные треугольники на первой карте. На рисунке 80 представлены построенные треугольники на второй карте. На рисунке 81 представлен результат работы комплексирования объекта на первой карте. Как мы видим объект частично расположен внутри объекта исходной карты. Так быть не должно, так как на первой исходной карте два объекта не соприкасались. Поэтому проведем индексирование и сделаем необходимый вырез на второй карте. На рисунке 82 представлен полученный вырез.

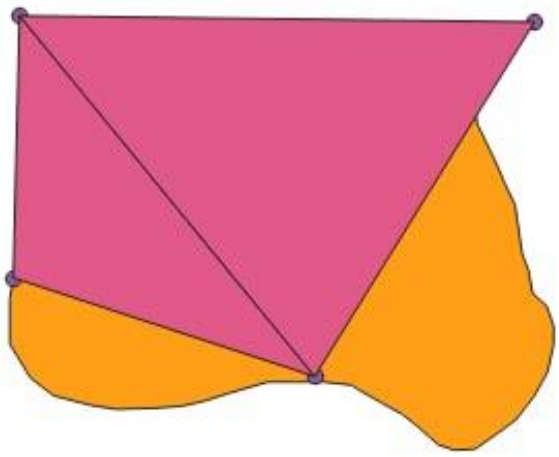


Рисунок 79 – Треугольники первой карты

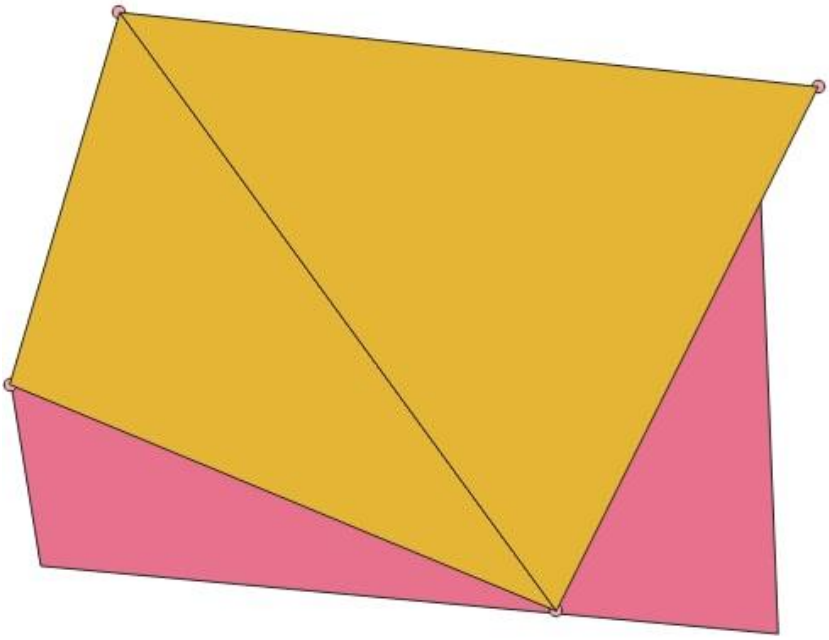


Рисунок 80 – Треугольники второй карты.

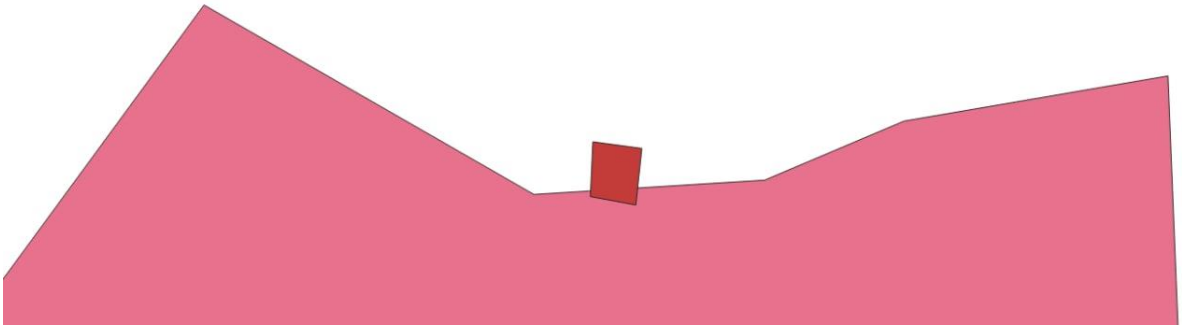


Рисунок 81 – Комплексированный объект

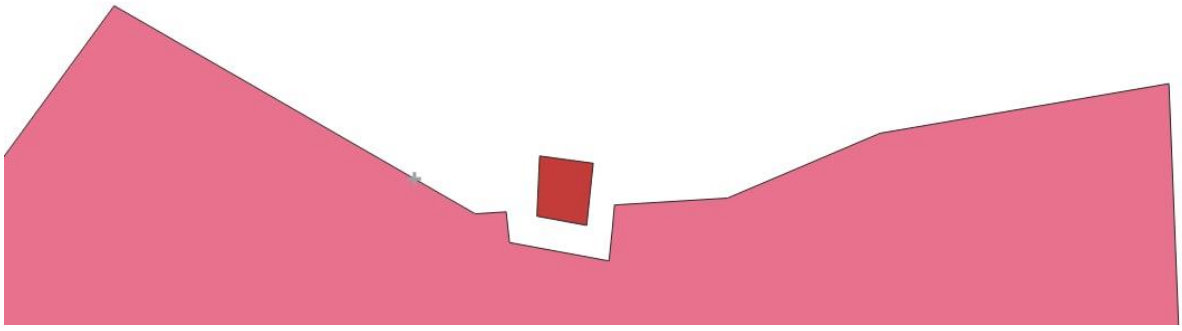


Рисунок 82 – Полученный необходимый вырез

ЗАКЛЮЧЕНИЕ

В ходе выполнения бакалаврской работы был разработан алгоритм комплексирования векторных данных с сохранением топологий в среде QGIS на языке PyQGIS. Были изучены и проанализированы литературные материалы по теме бакалаврской работы.

На основе данных, полученных в ходе анализа литературных материалов по теме работы, были определены цели и задачи выпускной квалификационной работы, а также была выбрана среда разработки.

В ходе выполнения бакалаврской работы были достигнуты все поставленные цели, а именно:

– изучены и проанализированы материалы по теме работы;

– выбрана среда разработки алгоритма;

– разработан алгоритм комплексирования векторных данных с сохранением топологий;

– произведено тестирование разработанного алгоритма.

Разработанный алгоритм состоит из следующих этапов:

– получение исходных данных;

– определение базовых точек;

– построение треугольников;

– расчет барицентрических координат;

– комплексирование векторных данных;

– анализ получившихся топологий;

– исправление объектов в случае нарушения топологических связей.

В ходе тестирования разработанного алгоритма были выявлены недостатки, которые могут быть исправлены в дальнейшем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Методы комплексирования изображений в многоспектральных оптико-электронных системах / Васильев А.С., Трушкина А.В. В сборнике: ГРАФИКОН'2016. Труды 26-й Международной научной конференции. 2016. С. 314-318.

2 Д. С. Шарак, А .В. Хижняк, А.С. Мамченко. К вопросу об обосновании применения комплексирования изображений для систем автоматического сопровождения в условиях сложного фона. Журнал радиоэлектроники [электронный журнал]. 2019. № 5. Режим доступа: http://jre.cplire.ru/jre/may19/2/text.pdf DOI 10.30898/1684-1719.2019.5.2 (дата обращения: 18.02.2022).

3 Silva-Coira F, Parama JR, Ladra S, Lopez JR, Gutierrez G (2020) Efficient processing of raster and vector data. PLoS ONE 15(1): e0226943.

4 Основы геоинформатики: В 2 кн. Кн. 1: Учеб. пособие для 0-75 студ. вузов / Е.Г.Капралов, А.В.Кошкарев, В.С.Тикунов и др.; Под ред. В.С.Тикунова. — М.: Издательский центр «Академия», 2004. — 352 с.

5 Документация QGIS [Электронный ресурс] // URL: https://qgis.org/ru/docs/index.html (дата обращения: 10.02.2022).

6 Документация PyQGIS [Электронный ресурс] // URL: https://qgis.org/pyqgis/ (дата обращения: 14.02.2022).

7 Алгоритм триангуляции Делоне методом заметающей прямой [Электронный ресурс] // URL: https://habr.com/ru/post/445048/ (дата обращения: 16.02.2022).

8 David G. Lowe. Object recognition from local scale-invariant features // Proceedings of the International Conference on Computer Vision. — 1999. — Т. 2. — С. 1150—1157.

ПРИЛОЖЕНИЕ А

(справочное)

Файл barycentric_coor.py

```
from posixpath import split
from random import triangular
from scipy.spatial import Delaunay
from osgeo import gdal
import numpy as np
import cv2
import os
from PIL import Image

class Moved:

    #функция инизиализации запускается при создании объекта класса Moved
    def __init__(self, move_layer):
        self.vertex_point_in = "ptI"
        self.vertex_point_out = "ptII"
        self.move_layer = move_layer
        self.type_of_geom                                                    =
QgsProject.instance().mapLayersByName(move_layer)[0].geometryType()
        self.dict_points200 = {}
        self.dict_points1000 = {}
        self.coordinate_system = QgsProject.instance().crs().authid()

    #функция выполняющая проверку находится ли точка внутри треугольника
    def is_in_triangle(self, pointXY, triangle):
        pointX = pointXY[0]
        pointY = pointXY[1]
        pointX1 = triangle[0][0]
        pointY1 = triangle[0][1]
        pointX2 = triangle[1][0]
        pointY2 = triangle[1][1]
        pointX3 = triangle[2][0]
        pointY3 = triangle[2][1]
        v1 = ((pointX1 - pointX) * (pointY2 - pointY1)) - ((pointX2 -
pointX1)*(pointY1 - pointY))
        v2 = ((pointX2 - pointX) * (pointY3 - pointY2)) - ((pointX3 -
pointX2)*(pointY2 - pointY))
        v3 = ((pointX3 - pointX) * (pointY1 - pointY3)) - ((pointX1 -
pointX3)*(pointY3 - pointY))
        if (v1 >= 0 and v2 >= 0 and v3 >= 0) or (v1 < 0 and v2 < 0 and v3 < 0):
            return True
        else:
            return False

    #функция отрисовки треугольника
    def draw_triangles(self, vertex_point_in, vertex_point_out):
        def toFixed(numObj, digits=0):
            return f"{numObj:.{digits}f}"

        #получения списка базовых точек первой карты
        list_layers = QgsProject.instance().mapLayersByName(vertex_point_in)
        layer_name = list_layers[0]
        dict_points_in = {}
        point_vertex = []
        point_vertex_wrong = []
```

```
        features = layer_name.getFeatures()
        for feature in features:
            geom = feature.geometry()
            list_points = geom.asMultiPoint()
            pointXY = [list_points[0].x(), list_points[0].y()]
            pointXY_wrong       =       [toFixed(list_points[0].x(),       3),
toFixed(list_points[0].y(), 3)]
            s = str(pointXY_wrong[0]), str(pointXY_wrong[1])
            dict_points_in[s] = feature['id']
            point_vertex.append(pointXY)
            point_vertex_wrong.append(pointXY_wrong)

        #выполнение треангуляции Делоне
        points = np.array(point_vertex)
        tri = Delaunay(points)
        triangleXY_in = points[tri.simplices]

        points = np.array(point_vertex_wrong)
        tri = Delaunay(points)
        triangleXY_in_wrong = points[tri.simplices]

        #формирование временного списка, состоящий из id точек для соответствия
треугольников разных карт
        triangle_id = []
        for triangle in triangleXY_in_wrong:
            one_triangle_id = [dict_points_in[tuple(triangle[0])],
            dict_points_in[tuple(triangle[1])],
            dict_points_in[tuple(triangle[2])]]
            triangle_id.append(one_triangle_id)

        #получения списка базовых точек второй карты
        list_layers = QgsProject.instance().mapLayersByName(vertex_point_out)
        layer_name = list_layers[0]
        dict_points_out = {}
        point_vertex = []
        features = layer_name.getFeatures()
        for feature in features:
            geom = feature.geometry()
            list_points = geom.asMultiPoint()
            pointXY = [list_points[0].x(), list_points[0].y()]
            s = str(pointXY[0]), str(pointXY[1])
            dict_points_out[feature['id']] = pointXY
            point_vertex.append(pointXY)

        triangleXY = []
        for id in triangle_id:
            triangle   =   [dict_points_out[id[0]],   dict_points_out[id[1]],
dict_points_out[id[2]]]
            triangleXY.append(triangle)

        triangleXY_out = np.array(triangleXY)



        # list_net200 = QgsProject.instance().mapLayersByName("net200")
        # net200 = list_net200[0]
        # list_net1000 = QgsProject.instance().mapLayersByName("net1000")
        # net1000 = list_net1000[0]

        # triangleXY_in = self.fromSquarToTriangle(net200)
        # triangleXY_out = self.fromSquarToTriangle(net1000)
```

```
        #Создание и загрузка в проект векторного слоя с тругольниками первой
карты
        suri = "MultiPolygon?crs=" + self.coordinate_system + "&index=yes"
        tr_name = "triangle" + vertex_point_in
        vl = QgsVectorLayer(suri, tr_name, "memory")
        pr = vl.dataProvider()
        vl.updateExtents()

        fet = QgsFeature()
        for triangl in triangleXY_in:

fet.setGeometry(QgsGeometry.fromPolygonXY([[QgsPointXY(triangl[0][0],
triangl[0][1]),           QgsPointXY(triangl[1][0],           triangl[1][1]),
QgsPointXY(triangl[2][0],      triangl[2][1]),      QgsPointXY(triangl[0][0],
triangl[0][1])]]))
            pr.addFeatures([fet])
            vl.updateExtents()

        vl.updateExtents()
        if not vl.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vl)

        #Создание и загрузка в проект векторного слоя с тругольниками второй
карты
        suri = "MultiPolygon?crs=" + self.coordinate_system + "&index=yes"
        tr_name = "triangle" + vertex_point_out
        vl = QgsVectorLayer(suri, tr_name, "memory")
        pr = vl.dataProvider()
        vl.updateExtents()

        fet = QgsFeature()
        for triangl in triangleXY_out:

fet.setGeometry(QgsGeometry.fromPolygonXY([[QgsPointXY(triangl[0][0],
triangl[0][1]),           QgsPointXY(triangl[1][0],           triangl[1][1]),
QgsPointXY(triangl[2][0],      triangl[2][1]),      QgsPointXY(triangl[0][0],
triangl[0][1])]]))
            pr.addFeatures([fet])
            vl.updateExtents()

        vl.updateExtents()
        if not vl.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vl)

        return triangleXY_in, triangleXY_out

    #функция вычисления барицентрических координат точки относительно
треугольника
    def barycentric_out(self, pointXY, triangle):
        pointX = pointXY[0]
        pointY = pointXY[1]
        point1X = triangle[0][0]
        point1Y = triangle[0][1]
        point2X = triangle[1][0]
        point2Y = triangle[1][1]
        point3X = triangle[2][0]
        point3Y = triangle[2][1]
        s  =  ((point2X  -  point1X)*(point3Y  -  point1Y)  -  (point3X  -
point1X)*(point2Y - point1Y))/2
```

```
        s1 = ((point2X - point1X)*(pointY - point1Y) - (pointX -
point1X)*(point2Y - point1Y))/2
        s2 = ((pointX - point1X)*(point3Y - point1Y) - (point3X -
point1X)*(pointY - point1Y))/2
        s3 = ((point2X - pointX)*(point3Y - pointY) - (point3X -
pointX)*(point2Y - pointY))/2
        u = s1/s
        v = s2/s
        w = s3/s
        coor = [u, v, w]
        return coor

    #функция вычисления координат точки относительно ее бариоцентрических
координат
    def barycentric_in(self, coor, triangle):
        (u, v, w) = coor
        point1X = triangle[0][0]
        point1Y = triangle[0][1]
        point2X = triangle[1][0]
        point2Y = triangle[1][1]
        point3X = triangle[2][0]
        point3Y = triangle[2][1]
        pointX = u*point3X + v*point2X + w*point1X
        pointY = u*point3Y + v*point2Y + w*point1Y
        pointXY = [pointX, pointY]
        return pointXY

    def sift_create(self):
        def get_extent(lname: str) -> dict:
            print(lname)
            extent200                                                    =
QgsProject.instance().mapLayersByName(lname)[0].dataProvider().extent()
            return                              dict(x_min=extent200.xMinimum(),
x_max=extent200.xMaximum(),
                        y_min=extent200.yMinimum(),
y_max=extent200.yMaximum())

#            img_1 = np.array(Image.open("C:/Users/kashi/Documents/Никита/ИС-
118/Диплом/VKR/rastr_admline200.tif").convert('L'))
#            img_2 = np.array(Image.open("C:/Users/kashi/Documents/Никита/ИС-
118/Диплом/VKR/rastr_admline1000.tif").convert('L'))
        img_1               =               np.array(Image.open("E:/Никита/ИС-
118/Диплом/VKR/images/rastr_admline200.tif").convert('L'))
        img_2               =               np.array(Image.open("E:/Никита/ИС-
118/Диплом/VKR/images/rastr_admline1000.tif").convert('L'))
        #img_1 = np.where(img_1==255, 0, 255)

        temp1 = img_1
        temp2 = img_2
        for i in range(1024):
            for j in range(1024):
                if img_1[i][j] == 255:
                    temp1[i][j] = 0
                else:
                    temp1[i][j] = 255

        for i in range(1024):
            for j in range(1024):
                if img_2[i][j] == 255:
                    temp2[i][j] = 0
                else:
                    temp2[i][j] = 255
```

```
        npKernel = np.uint8(np.zeros((5,5)))
        for i in range(5):
            npKernel[2][i] = 1
            npKernel[i][2] = 1

        npKernel_eroded1 = cv2.erode(temp1, npKernel)
        npKernel_eroded2 = cv2.erode(temp2, npKernel)
        #cv2.imshow('img', npKernel_eroded)

        #Расчет функции SIFT
        sift = cv2.xfeatures2d.SIFT_create()

        psd_kp1, psd_des1 = sift.detectAndCompute(npKernel_eroded1, None)
        psd_kp2, psd_des2 = sift.detectAndCompute(npKernel_eroded2, None)

        # 4) Сопоставление признаков фланна
        FLANN_INDEX_KDTREE = 1
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)

        flann = cv2.FlannBasedMatcher(index_params, search_params)
        matches = flann.knnMatch(psd_des1, psd_des2, k=2)
        goodMatch = []
        for m, n in matches:
            # goodMatch - это отфильтрованная высококачественная пара.
            #Если расстояние до первого совпадения в двух парах меньше 1/2
расстояния до второго совпадения, это может указывать на то, что первая пара
является уникальной и неповторяющейся характерной точкой на двух изображениях.
, Можно сохранить.
            if m.distance < 0.50*n.distance:
                goodMatch.append(m)
        # Добавить измерение
        goodMatch = np.expand_dims(goodMatch, 1)

        list_points200 = []
        list_points1000 = []

        extent200 = get_extent("admlin200")
        extent1000 = get_extent("admlin1000")
        cfx_200 = (extent200["x_max"] - extent200["x_min"]) / 1024.0
        cfy_200 = (extent200["y_max"] - extent200["y_min"]) / 1024.0

        cfx_1000 = (extent1000["x_max"] - extent1000["x_min"]) / 1024.0
        cfy_1000 = (extent1000["y_max"] - extent1000["y_min"]) / 1024.0

        count = 0
        for match in goodMatch:
            pt1 = psd_kp1[match[0].queryIdx].pt
            pt2 = psd_kp2[match[0].trainIdx].pt
            list_points200.append([count,         (extent200["x_min"]         +
pt1[0]*cfx_200, extent200["y_max"] - pt1[1]*cfy_200)])
            list_points1000.append([count,     (extent1000["x_min"]          +
pt2[0]*cfx_1000, extent1000["y_max"] - pt2[1]*cfy_1000)])
            count += 1
        list_points200.append([count,                      (extent200["x_min"],
extent200["y_min"])])
        list_points1000.append([count,                    (extent1000["x_min"],
extent1000["y_min"])])
        count += 1
        list_points200.append([count,                      (extent200["x_min"],
extent200["y_max"])])
        list_points1000.append([count,                    (extent1000["x_min"],
extent1000["y_max"])])
```

```
        count += 1
        list_points200.append([count,                          (extent200["x_max"],
extent200["y_min"])])
        list_points1000.append([count,                        (extent1000["x_max"],
extent1000["y_min"])])
        count += 1
        list_points200.append([count,                          (extent200["x_max"],
extent200["y_max"])])
        list_points1000.append([count,                        (extent1000["x_max"],
extent1000["y_max"])])

        suri  =  "MultiPoint?crs="  +  QgsProject.instance().crs().authid()  +
"&index=yes"
        tr_name = "pt200"
        vl = QgsVectorLayer(suri, tr_name, "memory")
        pr = vl.dataProvider()
        pr.addAttributes([QgsField("id",  QVariant.Int)])
        vl.updateFields()
        vl.updateExtents()
        fet = QgsFeature()
        for pt in list_points200:
            fet.setGeometry(QgsGeometry.fromMultiPointXY([QgsPointXY(pt[1][0],
pt[1][1])]))
            fet.setAttributes([pt[0]])
            pr.addFeatures([fet])
            vl.updateExtents()

        vl.updateExtents()
        if not vl.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vl)

        suri  =  "MultiPoint?crs="  +  QgsProject.instance().crs().authid()  +
"&index=yes"
        tr_name = "pt1000"
        vl = QgsVectorLayer(suri, tr_name, "memory")
        pr = vl.dataProvider()
        pr.addAttributes([QgsField("id",  QVariant.Int)])
        vl.updateFields()
        vl.updateExtents()
        fet = QgsFeature()
        for pt in list_points1000:
            fet.setGeometry(QgsGeometry.fromMultiPointXY([QgsPointXY(pt[1][0],
pt[1][1])]))
            fet.setAttributes([pt[0]])
            pr.addFeatures([fet])
            vl.updateExtents()

        vl.updateExtents()
        if not vl.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vl)

    def split_line(self, points):
        otrs = []
        for i in range(len(points) - 1):
            line = [points[i], points[i+1]]
            xm = (line[0][0] + line[1][0]) / 2
            ym = (line[0][1] + line[1][1]) / 2

            xm1 = (line[0][0] + xm) / 2
```

```
            ym1 = (line[0][1] + ym) / 2

            xm2 = (xm + line[1][0]) / 2
            ym2 = (ym + line[1][1]) / 2

            otrs.append(points[i])
            otrs.append([xm1, ym1])
            otrs.append([xm, ym])
            otrs.append([xm2, ym2])
        otrs.append(points[-1])
        return otrs

    #основная функция запускаемая пользователем

    def run(self):
        project = QgsProject.instance()
        #удаление слоёв
        for layer in project.mapLayers().values():
            if             layer.name().startswith("triangle")             or
layer.name().startswith("moved") or layer.name().startswith("pt"):
                project.removeMapLayer(layer.id())

        #self.sift_create()

        ls_1 = project.mapLayersByName(self.vertex_point_in)
        ls_2 = project.mapLayersByName(self.vertex_point_out)
        ls_3 = project.mapLayersByName(self.move_layer)

        if not ls_1 or not ls_2 or not ls_3:
            print("Указанного слоя с таким именем не существует")
            return


        #получение списков и отрисовка треугольников
        (triangleXY_in,                    triangleXY_out)                    =
self.draw_triangles(self.vertex_point_in, self.vertex_point_out)

        #индексация треугольников
        dict_triangleXY_in = {}
        count = 0
        for tr in triangleXY_in:
            dict_triangleXY_in[str(tr)] = count
            count += 1

        dict_triangleXY_out = {}
        count = 0
        for tr in triangleXY_out:
            dict_triangleXY_out[count] = tr
            count += 1

        list_layers = project.mapLayersByName(self.move_layer)
        layer_name = list_layers[0]
        features = layer_name.getFeatures()
        #получение объектов на первой карте и их построение на второй
относительно геометрии
        #тип геометрии: точки
        if self.type_of_geom == 0:
            points = []
            for feature in features:
                geom = feature.geometry()
                attr_list = feature.attributes()
                list_points = geom.asMultiPoint()
                pointXY = [list_points[0].x(), list_points[0].y()]
```

```
                points.append(pointXY)

            coors = []
            for point in points:
                for triangle in triangleXY_in:
                    if(self.is_in_triangle(point,triangle)):
                        coors.append([dict_triangleXY_in[str(triangle)],
self.barycentric_out(point, triangle)])
                        break


            suri = "MultiPoint?crs=" + self.coordinate_system + "&index=yes"
            name = "movedLayer"
            vl = QgsVectorLayer(suri, name, "memory")
            pr = vl.dataProvider()
            vl.updateExtents()
            fet = QgsFeature()

            for coor in coors:
                pointXY             =             self.barycentric_in(coor[1],
dict_triangleXY_out[coor[0]])
                fet.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(pointXY[0],
pointXY[1])))
                pr.addFeatures([fet])
                vl.updateExtents()

        #тип геометрии: линии
        if self.type_of_geom == 1:
            feature_XY =[]
            for feature in features:
                points = []
                geom = feature.geometry()
                attr_list = feature.attributes()
                list_points = geom.asMultiPolyline()
                for point in list_points[0]:
                    pointXY = [point.x(), point.y()]
                    points.append(pointXY)
#                feature_XY.append(self.split_line(points))
                feature_XY.append(points)

            #feature_XY_split = self.split_line(feature_XY)
            feature_coors = []
            for points in feature_XY:
                coors = []
                for point in points:
                    for triangle in triangleXY_in:
                        if(self.is_in_triangle(point,triangle)):
                            coors.append([dict_triangleXY_in[str(triangle)],
self.barycentric_out(point, triangle)])
                            break
                feature_coors.append(coors)


            suri   =   "MultiLineString?crs="   +   self.coordinate_system   +
"&index=yes"
            name = "movedLayer"
            vl = QgsVectorLayer(suri, name, "memory")
            pr = vl.dataProvider()
            vl.updateExtents()
            fet = QgsFeature()

            for coors in feature_coors:
                list_pointXY = []
```

```
                for coor in coors:
                    pointXY           =           self.barycentric_in(coor[1],
dict_triangleXY_out[coor[0]])
                    list_pointXY.append(QgsPointXY(pointXY[0], pointXY[1]))
                fet.setGeometry(QgsGeometry.fromPolylineXY(list_pointXY))
                pr.addFeatures([fet])
                vl.updateExtents()

        #тип геометрии: полигоны
        if self.type_of_geom == 2:

            feature_XY =[]
            for feature in features:
                points = []
                geom = feature.geometry()
                attr_list = feature.attributes()
                list_points = geom.asMultiPolygon()
                for point in list_points[0][0]:
                    pointXY = [point.x(), point.y()]
                    points.append(pointXY)
                feature_XY.append(self.split_line(points))
#                feature_XY.append(points)

            #feature_XY_split = self.split_line(feature_XY)
            feature_coors = []
            for points in feature_XY:
                coors = []
                for point in points:
                    for triangle in triangleXY_in:
                        if(self.is_in_triangle(point,triangle)):
                            coors.append([dict_triangleXY_in[str(triangle)],
self.barycentric_out(point, triangle)])
                            break
                feature_coors.append(coors)


            suri = "MultiPolygon?crs=" + self.coordinate_system + "&index=yes"
            name = "movedLayer"
            vl = QgsVectorLayer(suri, name, "memory")
            pr = vl.dataProvider()
            vl.updateExtents()
            fet = QgsFeature()

            for coors in feature_coors:
                list_pointXY = []
                for coor in coors:
                    pointXY           =           self.barycentric_in(coor[1],
dict_triangleXY_out[coor[0]])
                    list_pointXY.append(QgsPointXY(pointXY[0], pointXY[1]))
                fet.setGeometry(QgsGeometry.fromPolygonXY([list_pointXY]))
                pr.addFeatures([fet])
                vl.updateExtents()

        #добавление слоя на карту
        if not vl.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vl)

#создание объекта класса Moved: указание имён слоёв с базовыми точками двух
карт и переносимых объектов, а также указание типа геометрии
mv = Moved("homeOne")
mv.run()
```

ПРИЛОЖЕНИЕ Б

(справочное)

Файл indexing_of_elements.py

```
from pyexpat import features
import random

class Index_of_element(object):

    def __init__(self, layers_name):

        self.project = QgsProject.instance()
        self.del_temp_layers()
        self.layers_name = layers_name
        self.layers_name.append("movedLayer")
        self.list_number_rect = []
        self.numbers_seperation_rects = []
        self.coordinate_system = self.project.crs().authid()

    # Получение эксремумов главного прямоугольника
    def get_points_main_rectangle(self):
        def get_extent(lname: str) -> dict:
            extent200                                                    =
self.project.mapLayersByName(lname)[0].dataProvider().extent()
            return                              dict(x_min=extent200.xMinimum(),
x_max=extent200.xMaximum(),
                        y_min=extent200.yMinimum(),
y_max=extent200.yMaximum())

        extent = get_extent("lakeII")

        left_top = QgsPointXY(extent["x_min"], extent["y_max"])
        right_top = QgsPointXY(extent["x_max"], extent["y_max"])
        right_down = QgsPointXY(extent["x_max"], extent["y_min"])
        left_down = QgsPointXY(extent["x_min"], extent["y_min"])

        return [left_top,right_top, right_down, left_down]


    # Создание временного слоя
    def add_temporary_layer(self):
        suri = "MultiPolygon?crs=" + self.coordinate_system + "&index=yes"
        vector_layer        =        QgsVectorLayer(suri,        "rectangle",
"memory")#MultiLineString
        pr = vector_layer.dataProvider()
        #vector_layer.setStyleSheet("background: black;")
        vector_layer.updateExtents()

        # Добавление объекта (прямоугольника) на векторнй слой
        triangleXY = self.get_points_main_rectangle()
        fet = QgsFeature()
        fet.setGeometry(QgsGeometry.fromPolygonXY([[#fromPolylineXY
        triangleXY[0],
        triangleXY[1],
        triangleXY[2],
        triangleXY[3],
        triangleXY[0]]]))
        pr.addFeatures([fet])
```

```
        # Присваиваем чёрный цвет
        properties = {'color': 'white', 'outline_color': 'black'}
        symbol = QgsFillSymbol.createSimple(properties)
        renderer = QgsSingleSymbolRenderer(symbol)
        vector_layer.setRenderer(renderer)

        vector_layer.updateExtents()
        if not vector_layer.isValid():
            print("Layer failed to load!")
        else:
            QgsProject.instance().addMapLayer(vector_layer)

        return "rectangle"


    # Получение всех обектов внутри прямоугольника
    def separation_on_rect(self, name_layer):
        #Получение слоя по имени
        vecotr_layer = self.project.mapLayersByName(name_layer)[0]
        features = vecotr_layer.getFeatures()

        # Получение геометрии
        for feature in features:
            rect_points = feature.geometry().asPolygon()[0]

        # Деление на 4 прямоугольника
        for i in range (1, 5):
            #number_rect += 1
            if( "_" in name_layer):
                new_number_rect = name_layer.split("_")[1] + "." + str(i)
            else:
                new_number_rect = str(i)
            suri = "MultiPolygon?crs=" + self.coordinate_system + "&index=yes"
            vl = QgsVectorLayer(suri, "rectangle_" + new_number_rect, "memory")
            if(len(new_number_rect.split(".")) < 15):
                self.list_number_rect.append(new_number_rect)
            pr = vl.dataProvider()
            vl.updateExtents()

            fet = QgsFeature()
            if(i==1):
                fet.setGeometry(QgsGeometry.fromPolygonXY([[
                rect_points[0],
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],rect_points[1][1]),
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],                    (rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
                QgsPointXY(rect_points[3][0],(rect_points[0][1]-
rect_points[3][1])/2 + rect_points[3][1]),
                rect_points[0]]]))
                pr.addFeatures([fet])

            elif(i==2):
                fet.setGeometry(QgsGeometry.fromPolygonXY([[
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],rect_points[1][1]),
                rect_points[1],
                QgsPointXY(rect_points[2][0],(rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
```

```
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],                    (rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],rect_points[1][1])]]))
                pr.addFeatures([fet])

            elif(i==3):
                fet.setGeometry(QgsGeometry.fromPolygonXY([[
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],                    (rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
                QgsPointXY(rect_points[2][0],(rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
                rect_points[2],
                QgsPointXY((rect_points[2][0]-
rect_points[3][0])/2+rect_points[3][0],rect_points[2][1]),
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],                    (rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1])]]))
                pr.addFeatures([fet])

            elif(i==4):
                fet.setGeometry(QgsGeometry.fromPolygonXY([[
                QgsPointXY(rect_points[3][0],(rect_points[0][1]-
rect_points[3][1])/2 + rect_points[3][1]),
                QgsPointXY((rect_points[1][0]-
rect_points[0][0])/2+rect_points[0][0],                    (rect_points[1][1]-
rect_points[2][1])/2 + rect_points[2][1]),
                QgsPointXY((rect_points[2][0]-
rect_points[3][0])/2+rect_points[3][0],rect_points[2][1]),
                rect_points[3],
                QgsPointXY(rect_points[3][0],(rect_points[0][1]-
rect_points[3][1])/2 + rect_points[3][1])]]))
                pr.addFeatures([fet])

            # Присваиваем чёрный цвет
            properties = {'color': 'white', 'outline_color': 'black'}
            symbol = QgsFillSymbol.createSimple(properties)
            renderer = QgsSingleSymbolRenderer(symbol)
            vl.setRenderer(renderer)

            vl.updateExtents()
            if not vl.isValid():
                print("Layer failed to load!")
            else:
                QgsProject.instance().addMapLayer(vl)

        # Удаление разделённого вектора
        self.project.removeMapLayer(vecotr_layer.id())


    # Рекурсия на разделение по прямоугольникам
    def separation(self):
        # Получение названий всех прямоугольников
        current_rects = []
        for number in self.list_number_rect:
            current_rects.append("rectangle_" + number)
        for rect in current_rects:
            vecotr_layer = self.project.mapLayersByName(rect)[0]
            features = vecotr_layer.getFeatures()
            # Получение геометрии прямоугольника
            for feature in features:
```

```
                rectangle = feature.geometry()

            # Находим количество пересечений со всеми объектами слоя в текущем
прямоугольнике
            list_intersects = []
            count_intersects = 0
            inner_intersects = True
            for current_layer in self.layers_name:
                layer = self.project.mapLayersByName(current_layer)[0]
                feats = layer.getFeatures()
                for feature in feats:
                    # Проверка на наличии объектов в прямоугольнике
                    if (rectangle.intersects(feature.geometry())):
                        count_intersects += 1
                        list_intersects.append(feature.geometry())
                        #list_not_intersects.append(feature.geometry())

            # Обнуление точек пересечения если есть непересекающийся объект
            inner_intersects                                                =
self.zero_count_inner_intersects(list_intersects)

            # Заполнение списка прямоугольников с пересечениями
            if (count_intersects > 1 and inner_intersects == True):
                self.numbers_seperation_rects.append(rect.split('_')[1])
                self.numbers_seperation_rects                               =
list(set(self.numbers_seperation_rects))

            if (count_intersects > 1 and inner_intersects == False):
                # Делим прямоугольник на 4
                self.separation_on_rect(rect)
                self.list_number_rect.remove(rect.split("_")[1])
                return self.separation()


    # Получение итогового списка индексом элемента
    def fill_index(self):
        total_list = []
        layers_rect_name = []
        layers_main_name = self.layers_name
        for     current_layer     in     [layer.name()     for     layer     in
QgsProject.instance().mapLayers().values()]:
            if "rectangle_" in current_layer:
                layers_rect_name.append(current_layer)

        for current_layer in layers_main_name:
            feats                                                           =
self.project.mapLayersByName(current_layer)[0].getFeatures()
            for feat in feats:
                str_feature = ""
                str_index = ""
                for current_rect in layers_rect_name:
                    feats_rect                                              =
self.project.mapLayersByName(current_rect)[0].getFeatures()
                    for feat_rect in feats_rect:
                        if(feat.geometry().intersects(feat_rect.geometry())):
                                str_index += current_rect.split("_")[1] + ";"

                str_feature = current_layer + "_" + str(feat.id()) + "_" +
str_index
                total_list.append(str_feature)

        print(total_list)
        return total_list
```

```
    # Обнуление точек пересечения если есть непересекающийся объект
    def zero_count_inner_intersects(self, list_intersects):
        for geom_in in list_intersects:
            for geom_out in list_intersects:
                if (geom_in != geom_out):
                    if not(geom_in.intersects(geom_out)):
                        return False
        return True


    # Установка цвета прямоугольнико для каждого объекта
    def set_color_rects(self, list_indexs):
        for obj in list_indexs:
            name_layer = obj.split("_")[0]
            id_feature = obj.split("_")[1]
            list_number_rects = obj.split("_")[2]

            vecotr_layer = self.project.mapLayersByName(name_layer)[0]

            for feat in vecotr_layer.getFeatures():
                if feat.id() == int(id_feature):
                    list_number_rects =  list_number_rects.split(";")
                    # Присваиваем цвет каждому объекту
                    r = str(random.randint(0, 255))
                    g = str(random.randint(0, 255))
                    b = str(random.randint(0, 255))
                    properties  =  {'color':  r+','+g+','+b  ,'outline_color':
'black'}
                    for number in list_number_rects:
                        if number != "" :
                            current_rect = "rectangle_" + number
                            rect_layer                                          =
self.project.mapLayersByName(current_rect)[0]
                            # Присвоение цвета
                            symbol = QgsFillSymbol.createSimple(properties)
                            renderer = QgsSingleSymbolRenderer(symbol)
                            rect_layer.setRenderer(renderer)
                    break

        # Присваиваем серый цвет у пересекающихся прямоугольников
        for number in self.numbers_seperation_rects:
            vecotr_layer    =    self.project.mapLayersByName('rectangle_'    +
number)[0]

            properties = {'color': '190, 190, 190', 'outline_color': 'black'}
            symbol = QgsFillSymbol.createSimple(properties)
            renderer = QgsSingleSymbolRenderer(symbol)
            vecotr_layer.setRenderer(renderer)


    # Запуска скрипта
    def run(self):
        # Загрузка первого слоя и его разделение
        self.separation_on_rect(self.add_temporary_layer())
        # Рекурсия распределения прямоугольников
        self.separation()
        # Получение списка
        list_index = self.fill_index()
        return  list_index
```

```
    # Удаление временных слоёв
    def del_temp_layers(self):
        layers_name = [layer.name() for layer in
QgsProject.instance().mapLayers().values()]
        for current_layer in layers_name:
            if "rectangle" in current_layer:
                layer = self.project.mapLayersByName(current_layer)[0]
                self.project.removeMapLayer(layer.id())

    def fix_cross(self, indexes):
        class Index:
            def __init__(self, layer_name, id_f, str_indexes):
                self.layer_name = layer_name
                self.id_f = id_f
                list_of_indexes = str_indexes.split(";")
                list_of_indexes.pop()
                self.list_of_indexes = list_of_indexes

        def cross_point(line1, line2):
            x1 = line1[0].x()
            y1 = line1[0].y()
            x2 = line1[1].x()
            y2 = line1[1].y()
            x3 = line2[0].x()
            y3 = line2[0].y()
            x4 = line2[1].x()
            y4 = line2[1].y()
            k1 = (y1 - y2) / (x1 - x2)
            b1 = (x2 * y1 - x1 * y2) / (x2 - x1)
            k2 = (y3 - y4) / (x3 - x4)
            b2 = (x4 * y3 - x3 * y4) / (x4 - x3)
            if k1 != k2:
                x = (b2 - b1) / (k1 - k2)
                y = (k2 * b1 - k1 * b2) / (k2 - k1)
                if (min(x1, x2) <= x <= max(x1, x2)) and (min(x3, x4) <= x <=
max(x3, x4)) and (min(y1, y2) <= y <= max(y1, y2)) and (min(y3, y4) <= y <=
max(y3, y4)):
                    cross_point = QgsPointXY(x,y)
                else:
                    cross_point = 0
            else:
                cross_point = 0
            return cross_point

        def split_line(line):
            x1 = line[0].x()
            y1 = line[0].y()
            x2 = line[1].x()
            y2 = line[1].y()
            xm = (x1 + x2) / 2.0
            ym = (y1 + y2) / 2.0
            return QgsPointXY(xm, ym)

        def is_in_obj(point, obj):
            pol_str = ""
            for el in obj:
                pol_str += str(el.x()) + ' ' + str(el.y()) + ','
            pl_str = 'Polygon((' + pol_str[:-1] + '))'
            polygon_geometry = QgsGeometry.fromWkt(pl_str)
            point_geometry = QgsGeometry.fromWkt('Point ((' + str(point.x()) +
' ' + str(point.y()) + '))')
            polygon_geometry_engine =
QgsGeometry.createGeometryEngine(polygon_geometry.constGet())
```

```
            polygon_geometry_engine.prepareGeometry()
            if polygon_geometry_engine.intersects(point_geometry.constGet()):
                return True
            else:
                return False

        def e_distance(pt1, pt2):
            x1 = pt1.x()
            y1 = pt1.y()
            x2 = pt2.x()
            y2 = pt2.y()
            distance = ((x2 - x1)**2 + (y2 - y1)**2)**(1/2)
            return distance

        def move_pt(centr, list_pt_for_mv):
            list_pt_mvd = []
            for pt in list_pt_for_mv:
                x_new = (2.0 * pt.x()) - centr.x()
                y_new = (2.0 * pt.y()) - centr.y()
                list_pt_mvd.append(QgsPointXY(x_new, y_new))
            return list_pt_mvd

        def get_min_max_ind(list_pt_with_i, list_points1):
            list_i = [x for x, y in list_pt_with_i]
            if len(list_i) == len(set(list_i)):
                min_el = min(list_pt_with_i, key=lambda x: x[0])[1]
                min_id = min(list_pt_with_i, key=lambda x: x[0])[0]
                max_el = max(list_pt_with_i, key=lambda x: x[0])[1]
                max_id = max(list_pt_with_i, key=lambda x: x[0])[0] + 1
            else:
                min_id = min(list_pt_with_i, key=lambda x: x[0])[0]
                max_id = max(list_pt_with_i, key=lambda x: x[0])[0]
                pt = list_points1[min_id]
                if min == max:
                    distances = []
                    for k, p in list_pt_with_i:
                        distances.append(e_distance(pt, p))
                    min_el                                                  =
list_pt_with_i[distances.index(min(distances))][1]
                    max_el                                                  =
list_pt_with_i[distances.index(max(distances))][1]
                else:
                    min_l = []
                    max_l = []
                    for k, p in list_pt_with_i:
                        if k == min_id: min_l.append((k,p))
                        if k == max_id: max_l.append((k,p))
                    distances = []
                    for k, p in min_l:
                        distances.append(e_distance(pt, p))
                    min_el                                                  =
list_pt_with_i[distances.index(min(distances))][1]
                    distances = []
                    for k, p in max_l:
                        distances.append(e_distance(pt, p))
                    max_el= list_pt_with_i[distances.index(max(distances))][1]
                max_id += 1
            return [min_el, min_id, max_el, max_id]


        list_of_classes_index = []
        for index in indexes:
            data = index.split("_")
```

```
            list_of_classes_index.append(Index(data[0], data[1], data[2]))

        split_list_of_classes_index = []
        for layer_name in self.layers_name:
            list_indexes = []
            for index in list_of_classes_index:
                if index.layer_name == layer_name:
                    list_indexes.append(index)
            split_list_of_classes_index.append(list_indexes)


        list_crossing_object = []
        for i in range(len(split_list_of_classes_index) - 1):
            for el1 in split_list_of_classes_index[i]:
                for el2 in split_list_of_classes_index[-1]:
                    cross       =       list(set(el1.list_of_indexes)       &
set(el2.list_of_indexes))
                    if cross:
                        list_crossing_object.append((el1, el2))

        list_cross_point = []

        for (el1, el2) in list_crossing_object:
            layer1_name = self.project.mapLayersByName(el1.layer_name)[0]
            layer2_name = self.project.mapLayersByName(el2.layer_name)[0]
            feature1 = layer1_name.getFeature(int(el1.id_f))
            feature2 = layer2_name.getFeature(int(el2.id_f))
            list_pt_with_i = []
            list_points1 = feature1.geometry().asMultiPolygon()[0][0]
            list_points2 = feature2.geometry().asPolygon()[0]
            centr = feature2.geometry().centroid().asPoint()
            for i in range(len(list_points1) - 1):
                line1 = (list_points1[i], list_points1[i + 1])
                for j in range(len(list_points2) - 1):
                    line2 = (list_points2[j], list_points2[j + 1])
                    cross_pt = cross_point(line1, line2)
                    if cross_pt != 0:
                        moved_pt = split_line((line1[0],cross_pt))
                        if is_in_obj(moved_pt, list_points2):
                            moved_pt = split_line((cross_pt, line1[1]))
                        list_pt_with_i.append((i, moved_pt))

            list_pt_in_big_obj = []
            for pt in list_points2:
                if is_in_obj(pt, list_points1):
                    list_pt_in_big_obj.append(pt)

            # min_el = min(list_pt_with_i, key=lambda x: x[0])[1]
            # min_id = min(list_pt_with_i, key=lambda x: x[0])[0]
            # max_el = max(list_pt_with_i, key=lambda x: x[0])[1]
            # max_id = max(list_pt_with_i, key=lambda x: x[0])[0] + 1
            min_el, min_id, max_el, max_id = get_min_max_ind(list_pt_with_i,
list_points1)
            curent_pt = min_el
            list_pr_right_pos = []
            while(list_pt_in_big_obj):
                list_d = []
                for pt in list_pt_in_big_obj:
                    distance = e_distance(curent_pt, pt)
                    list_d.append(distance)
                min_d = min(list_d)
                min_i = list_d.index(min_d)
                curent_pt = list_pt_in_big_obj[min_i]
```

```
                list_pr_right_pos.append(curent_pt)
                list_pt_in_big_obj.pop(min_i)


            list_pt_mvd = move_pt(centr, list_pr_right_pos)

            l1 = list_points1[:min_id+1]
            l2 = list_points1[max_id:len(list_points1)]
            new_list_for_geom = l1 + [min_el] + list_pt_mvd + [max_el] + l2

            geom = QgsGeometry.fromPolygonXY([new_list_for_geom])
            layer1_name.dataProvider().changeGeometryValues({  int(el1.id_f)  :
geom })
            layer1_name.updateExtents()


            # suri = "MultiPoint?crs=" + self.coordinate_system + "&index=yes"
            # tr_name = "prprpr"
            # vl = QgsVectorLayer(suri, tr_name, "memory")
            # pr = vl.dataProvider()
            # vl.updateExtents()
            # fet = QgsFeature()
            # for pt in list_pt_mvd:
            #     fet.setGeometry(QgsGeometry.fromPointXY(pt))
            #     pr.addFeatures([fet])
            #     vl.updateExtents()

            # vl.updateExtents()
            # if not vl.isValid():
            #     print("Layer failed to load!")
            # else:
            #     QgsProject.instance().addMapLayer(vl)


obj = Index_of_element(["lakeII"])
indexes = obj.run()
obj.set_color_rects(indexes)
obj.fix_cross(indexes)
```