ОЛИМПИАДА ШКОЛЬНИКОВ «ШАГ В БУДУЩЕЕ»

НАУЧНО-ОБРАЗОВАТЕЛЬНОЕ СОРЕВНОВАНИЕ «ШАГ В БУДУЩЕЕ, МОСКВА»

ШМ0233

регистрационный номер

Информатика и системы управления
название факультета

Программное обеспечение ЭВМ и информационные технологии
название кафедры

Тепловая карта предложений жилья
название работы

Автор: **Медяновский Олег Вячеславович**
фамилия, имя, отчество
ГБОУ Школа №1301 имени Е.Т. Гайдара
наименование учебного заведения, класс

Научный руководитель: **Отсутствует**
фамилия, имя, отчество
Отсутствует
место работы
Отсутствует
звание, должность

Отсутствует
подпись научного руководителя

Москва - 2018

Оглавление

Введение

Современный человек осуществляет решение многих проблем, в процессе которого ему приходится иметь дело с большим объемом данных. Негласным стандартом для такой работы стало использование различных фильтров. Выбор жилья не стал исключением. Однако вместо того, чтобы с помощью фильтров исключать предложения, оказывается возможной и достаточной замена способа их представления на более удобный с точки зрения восприятия информации. Он позволяет отображать одновременно значительно больший объем информации на экране, благодаря чему пользователь сможет принимать решение на основании большего числа факторов, что сделает решение более взвешенным. Кроме информации о самой квартире, имеется возможность отобразить информацию об её окружении, транспортной доступности, инфраструктуре. Выбор методов отображения и их реализация описана в настоящей работе.

Функционал проекта

1. Отображение предложений

Для предложений я использую вместо классических маркеров заливку зданий определенным цветом. Цвет определяется на основании отличия реальной цены от заданной пользователем.

Рисунок 1.

Для этого отлично подойдет HSB-градиент (Рисунок 1), ограниченный снизу синим цветом, то есть градиент с параметром hue (оттенок) от 0 до 240. Так же он должен быть симметричен, то есть $P_{\text{макс}} - P_{\text{указ}} = P_{\text{указ}} - P_{\text{мин}}$ где P — цена.

Благодаря интуитивной ассоциации красного цвета с чем-то выше нормы, а синего, наоборот, с чем-то ниже нормы, пользователю будет легко привыкнуть к значениям цветов. Однако при построении градиента на всем диапазоне цен, представленных на карте, может возникнуть проблема. Она заключается в том, что градиент может стать слишком широким, из-за чего он перестанет отображать разницу в ценах.

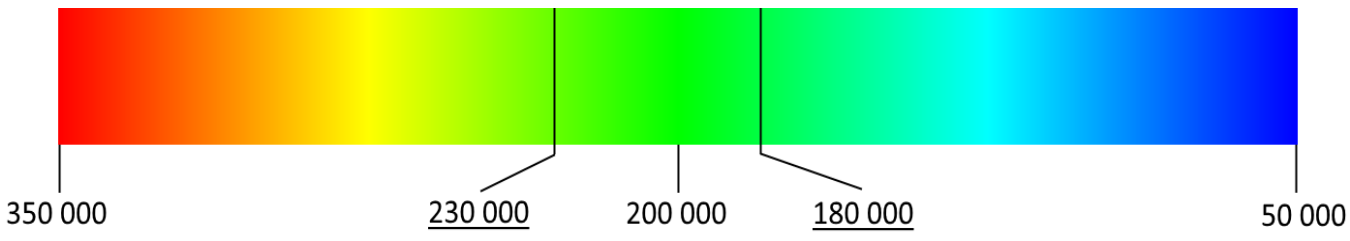


Рисунок 2.

На рисунке 2 показан градиент (все цены в рублях за метр квадратный), отражающий диапазон цен от 50000 до 350000, с указанной ценой в 200000. Также отмечены цены 230000 и 180000. Видно, что несмотря на отличие

цены на 30000 и 20000 соответственно, их цвета едва различимы. Для решения этой проблемы я использую настраиваемый диапазон цен. Пользователь будет выбирать не только целевую цену, но и диапазон цен.

Полученный градиент я использую для отрисовки домов, причем сама отрисовка происходит на прозрачном фоне, чтобы дома можно было наложить на карту. Пример представлен на Рисунке 3.

Рисунок 3.

2. Транспортная доступность

Для отображения транспортной доступности я окрашиваю дороги в цвета, зависящие от того, за какое время можно добраться до той или иной её части. Для этого, имея дорожный граф и начальную точку, я нахожу значения для каждой вершины: они и являются минимально возможными значениями времени, за которое можно добраться из начальной вершины в данную. На основании этих значений и максимального времени граф окрашивается с помощью градиента, приведенного на рисунке 4.

Рисунок 4.

Согласно рисунку 4, недоступные части графа окрашены в темно-красный цвет (находится за отметкой «макс. время»). Кроме автомобильного режима, использующего обычные автомобильные дороги, я добавил пешеходный режим, учитывающий пешеходные улицы и общественный транспорт. Этот граф должен быть отрисован также на прозрачном фоне, чтобы после наложения на карту можно было увидеть названия улиц.

3. Доступность инфраструктуры

При выборе дома я показываю список объектов инфраструктуры поблизости: магазины, кафе, госучреждения.

Существующие альтернативы

Отображение цен с помощью тепловых карт уже было реализовано, например, здесь: https://квартиры-домики.рф/Карта-цен/Москва/ (статья автора сайта: https://habrahabr.ru/post/335638/). Но метод отображения, реализованный на этом сайте, не показывает данные об отдельных домах. К тому же из-за погрешностей тепловой карты нельзя точно сказать о ценах в том или ином доме. К сожалению, во время работы над проектом я не знал об этом сайте, потому не смог заимствовать некоторые методы, описанные в указанной статье.

Отображение транспортной доступности имеется у сервиса https://maps.openrouteservice.org, но моё отображение отличается тем, что поддерживает общественный транспорт и методом отрисовки. У данного сервиса зоны закрашиваются сплошной заливкой, в то время как у меня закрашиваются только дороги, и потому пользователю легче ориентироваться на карте.

Ближайшую инфраструктуру показывают сервисы популярных агрегаторов жилья, например https://cian.ru. На данном сайте объекты инфраструктуры представляются на карте, что позволяет посетителю самостоятельно оценить её доступность. У меня же пользователь информируется о примерном времени, за которое реально добраться до тоги или иного объекта.

Реализация

1. Структура проекта и инструменты

1.1 Структура

Структура изображена на схеме в Приложение 1. Клиент отправляет запрос к серверу (сервер nginx на порте 80 — стандартный порт для http запросов). В зависимости от запроса nginx либо отдает тайл (изначально хранящийся как дисковый файл), либо перенаправляет запрос к серверу, написанному на java с использованием netty. После PathRouter в зависимости от uri запроса направляет его к одному из handler'ов.

PriceTileHandler — отдает тайлы, на которых нарисованы дома, окрашенные в определенный цвет (согласно запросу) и которые впоследствии накладываются на карту.

RoadGraphTileHandler — отдает тайлы, на которых нарисованы дороги, окрашенные в определенный цвет (согласно запросу) и которые впоследствии накладываются на карту.

MapPointSearchHandler — производит поиск по точке, координаты которой отправлены в запросе и, если объект найден, возвращает его данные в формате JSON.

StringSearchHandler — производит поиск по данной в запросе строке среди названий улиц, и при полном совпадении названий возвращает данные о найденном объекте.

StringPredictHandler — производит составление подсказок по префиксу, указанному в запросе, и возвращает их.

CloseObjectHandler — производит поиск объектов инфраструктуры, и возвращает их.

1.2 Инструменты

Сервер nginx был выбран по причине высокой производительности и простоты настройки. Его используют очень крупные сайты и порталы (vk.com,

yandex.ru — для некоторой части контента, и др.), что говорит о его высокой надежности и степени доверия к нему.

В качестве языка программирования был выбран Java как хорошо знакомый мне и зарекомендовавший себя при написании серверов. Благодаря такой направленности у этого языка есть много библиотек, сильно упрощающих процесс написания сервера.

Netty — это веб фреймворк, выбранный мною для написания сервера на java. По причине того, что я мало сталкивался с веб-фреймворками, он был выбран на основании своей популярности и того факта, что он используется крупными компаниями. Кроме того, в сети имеется крупное сообщество, в котором можно найти рекомендации по работе с этим продуктом.

2. Сервер:

Все данные карты, цены и методы работы с ними я объединил в классе PropertyMap. Следовательно при обработке запросов к серверу обработчик обращается только к объекту PropertyMap, а все обращения к внутренним объектам производятся только этим классом. Такое разделение позволит при необходимости держать несколько карт в одной виртуальной машине jvm с минимальными изменениями кода.

2.1 Данные

2.1.1 Геоданные

Я искал источник геоданных, и мой выбор пал на openstreetmap.org. Данные этого сервиса открыты и регулярно обновляются сообществом. После было необходимо создать парсер для этих данных. Данные openstreetmap.org распространяются в разных форматах, и я выбрал формат osm, поскольку очевидна необходимость использования overpass api[1]. Формат osm

[1] Overpass api – позволяет скачивать произвольную, ограниченную прямоугольником область карты. Поддерживает только формат osm.

предоставляет данные об объектах, организованные с помощью xml. Есть три основных элемента таких данных: Node, Way и Relation.

Node — это обычная точка. В качестве своих xml-атрибутов этот элемент имеет идентификатор id, географические координаты и информацию о пользователе, который её добавил или изменил. Также в него вложены элементы типа tag, которые содержат информацию о точке в формате ключ–значение. Например, автобусная остановка описывается как public_transport–platform.

Way — это элемент, описывающий полигоны и линии на карте. Он содержит те же атрибуты, что и Node. В него вложены элементы nd и tag. Элементы tag также содержат информацию о Way, nd содержат id элементов Node, из которых он состоит. Соответственно, соединив все элементы Node, мы получим представление Way.

Relation — это элемент, группирующий элементы Node, Way и другие Relation. Обычно элементы Relation являются маршрутами общественного транспорта или какими-либо сетями.

Для работы над этими данными созданы классы, предоставляющие геоданные. Они показаны в Приложение 2.

Выбор именно такой схемы классов показан ниже.

Поскольку osm-файлы обладают внушительным размером (карта Москвы занимает примерно 1,5 гигабайта), обычные xml парсеры, строящие DOM-дерево в памяти, не подходят. Для парсинга больших xml-файлов используются потоковые парсеры (stream parsers), которые не строят DOM-дерево в памяти, а всего лишь сообщают о типе элемента и его атрибутах. С такими парсерами вся забота о правильной иерархии элементов ложится на плечи разработчика. Поскольку иерархия элементов у формата osm очень проста[2], это не является проблемой. Java включает в себя StAX парсер.

[2] Вложенность элементов не превышает 2 уровней, если считать корневой элемент уровнем 0.

Загрузка происходит по схеме, представленной в Приложение 3. Карта — это фрагмент данных в формате osm, полученный с помощью overpass api. В самом начале читаются границы карты и записываются в соответствующие поля: это необходимо для того, чтобы при последующей конвертации географических координат в координаты на плоскости можно было получать не абсолютные их значения, а заданные относительно верхнего левого угла карты (точки с минимальной долготой и максимальной широтой). Это сделано для упрощения передачи координат от клиента к серверу.

Затем метод load() производит чтение файла и создает элементы Node, Way и Relation, которые сохраняются внутри загрузчика, а потом забираются из него с помощью методов getNodes(), getWays(), getRelations(), getSimpleNodes(). Благодаря тому, что все реализовано через интерфейс MapLoader, можно в дальнейшем создать другие загрузчики, реализующие этот интерфейс и читающие данные другого формата, получив тем самым поддержку новых форматов. Также этот метод создаёт дорожный граф, подробнее об этом в разделе 2.3 Дорожный граф.

2.1.2 Данные о предложениях

Я начал работать над проектом в прошлом году, поэтому использовал данные, вероятно, несколько устаревшие к настоящему времени. Я выполнил парсинг ресурса cian.ru и извлек данные о предложениях, после чего сохранил их в формате JSON. Из соответствующего файла сервер и читает эти данные. Все они используются исключительно для демонстрации возможностей программы.

2.2 Квадродерево

Для выбора и отрисовки домов, поиска близкой к ним инфраструктуры и составления дорожного графа необходим быстрый поиск по данным карты. Для этого существует множество методов организации данных на плоскости. Из них мною было выбрано квадродерево (Quadtree) за его простоту в реализации.

Квадродерево реализовано в классе QuadTreeNode, а методы для поиска в нем и его корень содержатся в классе QuadTree. Квадродерево может содержать в себе как точки, так и полигоны с линиями. В моей версии оно не сбалансированное.

В квадродереве у каждого узла может быть по 4 потомка, которые делят своего родителя на 4 равные части. Каждый узел в процессе создания считается конечным. Поиск необходимого узла происходит за время O($log_4 n$) (где n — количество элементов в дереве при условии, что дерево сбалансированное). Так как O($log_4 n$) пренебрежимо мало, сложность поиска в лучшем случае O(n_{min}), в худшем O(n_{max}) (где n_{min} — минимальное количество элементов в узле, n_{max} — максимальное их количество).

Добавление точки в квадродерево весьма тривиально. С линиями и полигонами уже сложнее: ведь необходимо их обрезать, чтобы они не выходили за границы узла. Для обрезки линий был создан алгоритм, реализованный в методе addRoad() класса QuadTreeNode. Для обрезки же полигонов я реализовал алгоритм Уайлера-Атертона (Weiler–Atherton). Его реализация находится в методе addPoly_double().

Поскольку дерево несбалансированное, узлы делятся лишь после того, как количество элементов в нем превосходит определенный порог. После этого текущий узел делится на 4 новых узла, которые назначаются потомками текущего. Все элементы из текущего узла перемещаются в его потомков согласно алгоритму добавления элементов в дерево. При этом текущий элемент перестаёт быть конечным.

Все алгоритмы поиска в квадродереве сводятся к одной последовательности действий: 1) поиск всех конечных узлов, которые пересекаются или содержатся в фигуре поиска, или содержат фигуру поиска; 2) обход всех найденных узлов и проверка их элементов на вхождение в фигуру поиска или вхождение фигуры поиска в них; 3) массив элементов, входящих в фигуру, и будет результатом.

Здесь есть пространство для оптимизаций. Например, проведя простую проверку на вхождение узла в фигуру поиска, при его полном вхождении мы можем добавить все элементы узла без проверки каждого из них.

Мною реализовано несколько алгоритмов поиска в дереве: по точке, окружности и прямоугольнику. Они также отличаются тем, какой именно результат выдают: Way, Node или RoadGraphNode.

2.3 Дорожный граф

Чтобы показать транспортную доступность дома, необходимо знать минимальное время, за которое можно добраться до фрагментов дорог. Для расчета этого времени необходимо объединить все возможные пути передвижения в единый граф (то есть объединять не только автомобильные дороги, но и пешеходные и маршруты общественного транспорта), и далее, имея подобный граф, провести расчет.

Вершинами графа являются объекты класса RoadGraphNode. Они хранят ссылки на другие вершины, с которыми они связаны, и время, за которое можно до них добраться. Сохраняется два разных значения времени: для пешеходного режима и для автомобильного. Время хранится не в минутах, а в сотых долях секунд (0,01 сек). Такая малая единица времени была выбрана по той причине, что на карте встречаются очень короткие участки дорог, которые, например, автомобилем, движущимся со скоростью 60 км/ч, преодолеваются за очень малое время. Тогда, если взять за минимальную единицу времени минуту, после округления выйдет, что многие участки будут преодолеваться мгновенно, следовательно, результаты вычислений окажутся некорректными.

Экземпляров RoadGraphNode достаточно много (на карте Москвы — 1716727), поэтому очень важно, чтобы они занимали минимум памяти. Это послужило причиной для отказа от динамических массивов ArrayList в пользу статических массивов java, ведь согласно описанию библиотеки jol (Java Object Layout) размер ArrayList — 24 байта (зависит от конфигурации jvm), а размер обычного

Array — 16 байт (также зависит от конфигурации jvm). Поскольку ArrayList реализован как обертка для Array, то в итоге суммарный размер ArrayList равен 40 байт. Соответственно переход от ArrayList к Array экономит 24 байта, а если учесть количество RoadGraphNode и количество массивов в них, то сохраняется минимум 4*24*1716727=164805792 байт ≈ 157 Мбайт. Поскольку в RoadGraphNode есть ещё и вложенные массивы, то реальная экономия будет еще больше.

Благодаря использованию Array я получил значительную экономию памяти. Но из-за того, что Array не может автоматически менять свой размер, код сильно усложнился. Для его упрощения мною были написаны построители (builders) для RoadGraphNode и самого дорожного графа: классы RoadGraphNodeBuilder и RoadGraphBuilder. В них реализована вся логика создания вершин и их соединения. После добавления всех узлов в RoadGraphBuilder вызывается его метод getRoadGraph(), который возвращает HashMap<Long, RoadGraphNode>, где ключ — это идентификатор узла, а значение — сам узел. Все узлы в этом HashMap связаны друг с другом и готовы к использованию.

Используя RoadGraphBuilder, MapLoader добавляет в граф данные обо всех дорогах, прочитанные им. После загрузки геоданных вызывается метод public_transport_init(), который добавляет к дорожному графу маршруты общественного транспорта.

2.4 Расчет графа

Под расчетом графа подразумевается нахождение минимально возможного времени пути из начальной точки до вершин графа. При этом предполагается, что это время меньше максимально допустимого.

При написании алгоритма для расчета графа поначалу мною был создан рекурсивный алгоритм (PropertyMap.recCalculateDistances()), очень похожий на поиск в глубину. Вместо сравнения вершины с искомой я записывал в вершину сумму весов уже пройденных ребер, если она меньше уже записанной в вершине. Перед этим во все вершины записывается максимально возможное

число (INT_MAX). Но при его использовании я заметил, что время его работы слишком велико. Тогда я реализовал расчет графа на основе поиска в ширину (PropertyMap.widthRecCalculateDistance()). Отличия в скорости работы видны из Приложение 4 и Приложение 5.

При работе с одним пользователем программа работает, как и должна, но если будет несколько пользователей, одновременно запрашивающих тайлы, то начнутся проблемы из-за того, что при смене параметров запроса (один пользователь смотрит транспортную доступность одного дома, а другой — другого) придется пересчитывать граф: даже если он был рассчитан раньше, то результаты предыдущего расчета перезаписываются новым. Для решения этой проблемы мною был реализован кэш. Вместо одной числовой переменной, в которую записывается удаленность точки, использован массив чисел, также хранящий удаленность этого узла в различных расчетах (соответственно удаленности всех узлов, рассчитанные в рамках одного запроса, хранятся под одним и тем же индексом). Чтобы определить, был ли уже произведен расчет для запрошенных параметров и, если был, то по какому индексу в массивах находятся его результаты, я создал класс CalculatedGraphCache и CalculatedGraphKey. С помощью второго выполняется проверка, являются ли запросы идентичными, а первый позволяет хранить с помощью HashMap с ключом CalculatedGraphKey индексы, по которым находятся результаты расчетов. Также экземпляр класса CalculatedGraphCache хранит список свободных для записи индексов и занимается их высвобождением при переполнении кэша.

2.5 Доступность инфраструктуры

Имея рассчитанный граф, очень просто найти инфраструктуру поблизости по следующему критерию: если максимальная её удаленность не превосходит максимальной удаленности при расчете графа. Для этого необходимо обойти все объекты инфраструктуры, для каждого из них найди ближайший объект RoadGraphNode. Его удаленность, с поправкой на расстояние между ними, и будет удаленностью объекта инфраструктуры. Если

же удаленность объекта RoadGraphNode больше максимальной, то данный объект считаем недоступным. После объекты сортируются по убыванию удаленности и будут отданы клиенту.

2.6 Отрисовка

2.6.1 Предложения

Предложения отрисовываются с помощью java graphics2d api на прозрачном фоне. Процесс отрисовки выглядит следующим образом. Для здания выбирается цена, она равна ближайшей к искомой цене из всех предложений. То есть выбирается предложение, для которого значение выражения $\left|P_{\text{цел.}} - P_{\text{дан.}}\right|$ минимально ($P_{\text{цел.}}$ – целевая цена, $P_{\text{дан.}}$ – цена текущего предложения). Если для здания нет предложений, оно не рисуется. Далее на основании разницы между целевой ценой и ценой, выбранной для дома, определяется цвет, после чего на основе геоданных вычисленным цветом рисуется полигон. Процесс повторяется для всех зданий, попадающих в границы тайла. Наконец, готовая картинка отдается клиенту.

2.6.2 Транспортная доступность

Дороги поначалу тоже отрисовывались с помощью java graphics2d api, но из-за того, что меня не устраивала скорость отрисовки, я решил использовать библиотеку Cairo (через JNI[3]). Сведения о разнице в производительности приведены в Приложение 6. Видно, что Cairo работает в среднем в 10 раз быстрее по сравнению с Graphics2D.

Отрисовка проводится следующим способом. У PropertyMap запрашиваются все элементы RoadGraphNode, которые входят в границы тайла. Далее для каждого RoadGraphNode рисуются линии к узлам, с которыми он связан. Для отрисовки линии сначала с помощью метода getNodeColor() выбираются цвета для начала и конца линии, толщина линии, зависящая от типа соединения (RoadType), и она прорисовывается с помощью градиента, от цвета начала к цвету конца.

[3] JNI – Java Native Interface, позволяет из java-кода вызывать методы динамических библиотек(.dll и .so), имеющие определенную сигнатуру, описанную в java-коде.

При использовании отрисовки через JNI все происходит так же, но вместо самой отрисовки линии данные о ней записываются в массив чисел в формате $(x_1, y_1, color_1, x_2, y_2, color_2, width)$. Затем этот массив передается в native-функцию drawNativeCall(), а уже программа, написанная на C++ с использованием библиотеки Cairo, рисует все эти линии и возвращает массив байт, содержащий готовое png-изображение.

3. Клиент

Клиент является лишь средством для демонстрации работоспособности сервера, поэтому его дизайн оставляет желать лучшего. Но со своей основной задачей он справляется. Окно приложения разделено на две части: справа показана карта, а слева приведены настройки, имеется строка поиска, отображается информация о предложениях. Так, в качестве координат текущего положения карты хранится положение левого верхнего угла отображаемой части карты относительно верхнего левого угла всей карты. Это продемонстрировано в Приложение 7. Получается, для того, чтобы сообщить серверу о координатах точки, достаточно отправить лишь сумму координат карты с положением точки на экране и текущий масштаб.

Дальнейшее развитие

В дальнейшем я планирую отрисовывать плавную тепловую карту вместо отдельных домов при малых масштабах, когда отдельные дома становятся едва различимыми, наподобие https://квартиры-домики.рф/Карта-цен/Москва/. Для этого придется пересмотреть алгоритм генерирования тепловой карты, поскольку у меня нет возможности для её пререндера.

Также я собираюсь заменить клиент, созданный на Python, на веб-клиент. Он будет значительно удобнее, ведь его не понадобится устанавливать. В будущем такой веб-клиент можно слить с каким-то агрегатором жилья или же превратить его в полноценный агрегатор.

Мне также хотелось бы использовать GPU для отрисовки тайлов, ведь производительность графических процессоров на порядок выше обычных.

Заключение

В результате работы над проектом я многое узнал о геоданных и методах их обработки. Также мною были изучены алгоритмы работы с графикой.

Это был мой первый опыт написания крупных проектов на java, а также написания клиент-серверных приложений. Я получил неоценимый опыт в работе с системой контроля версий Git. И научился создавать JNI-код для java.

В итоге у меня не получилось приложение, способное заменить обычные агрегаторы жилья. Зато я создал приложение, дополняющее их ещё одним средством для поиска жилья.

Источники

Проекция меркатора https://zenodo.org/record/35392

Все геоданные и тайлы карты https://www.openstreetmap.org

Описание формата osm https://wiki.openstreetmap.org/wiki/OSM_XML

Библиотека Cairo https://www.cairographics.org/

Библиотека и документация Netty http://netty.io/

Сервер и документация Nginx http://nginx.org/

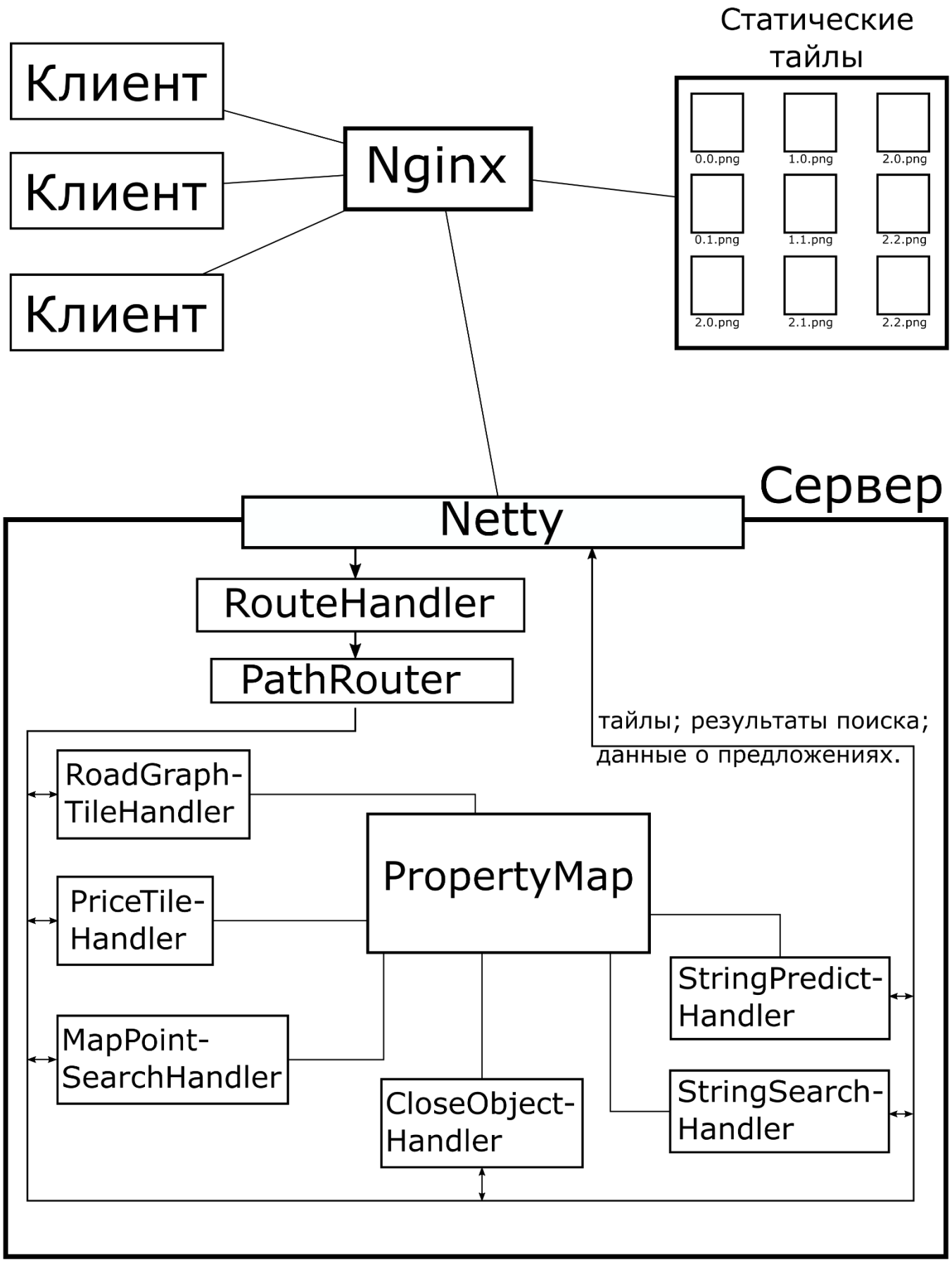
Библиотека и документация wxPython https://www.wxpython.org/

Библиотека minimal json https://eclipsesource.com/blogs/2013/04/18/minimal-json-parser-for-java/

Библиотека jol http://openjdk.java.net/projects/code-tools/jol/

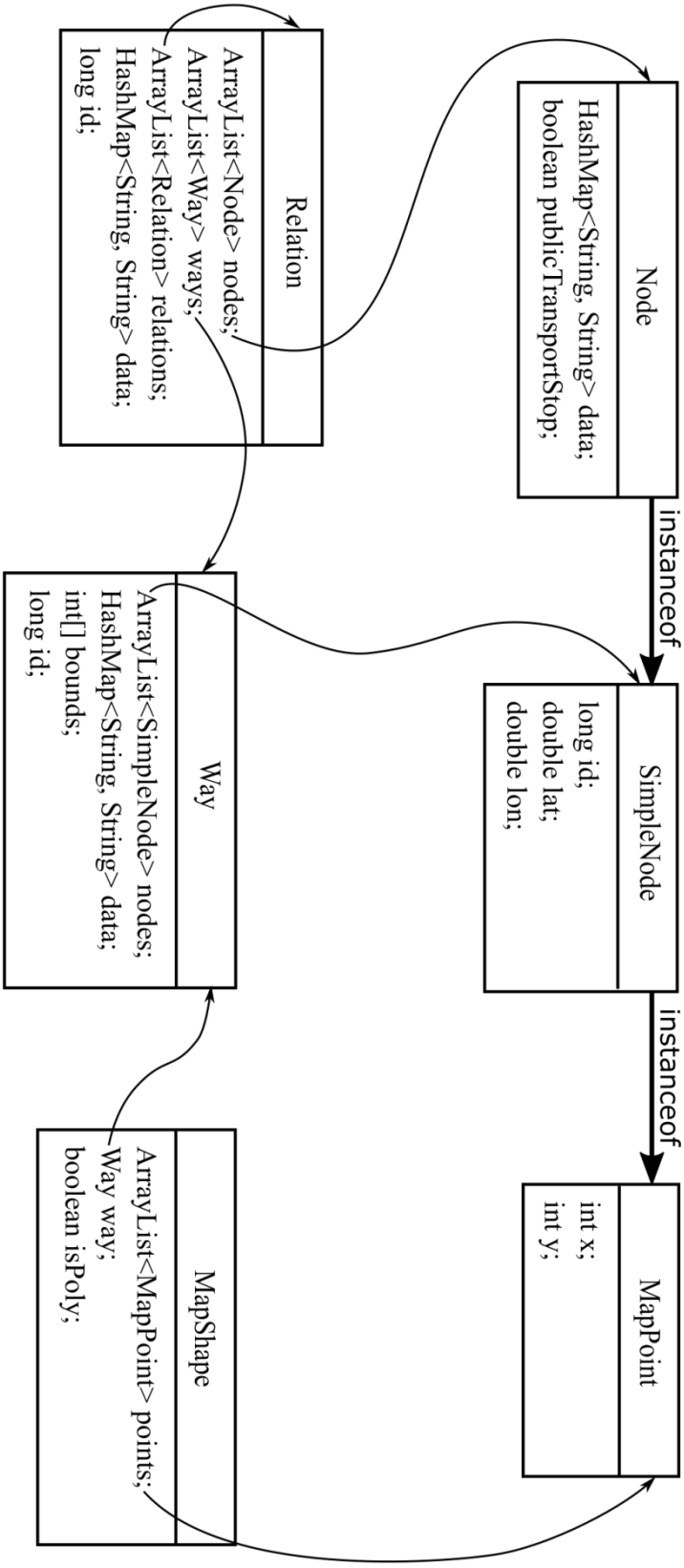
Приложения

Структура проекта



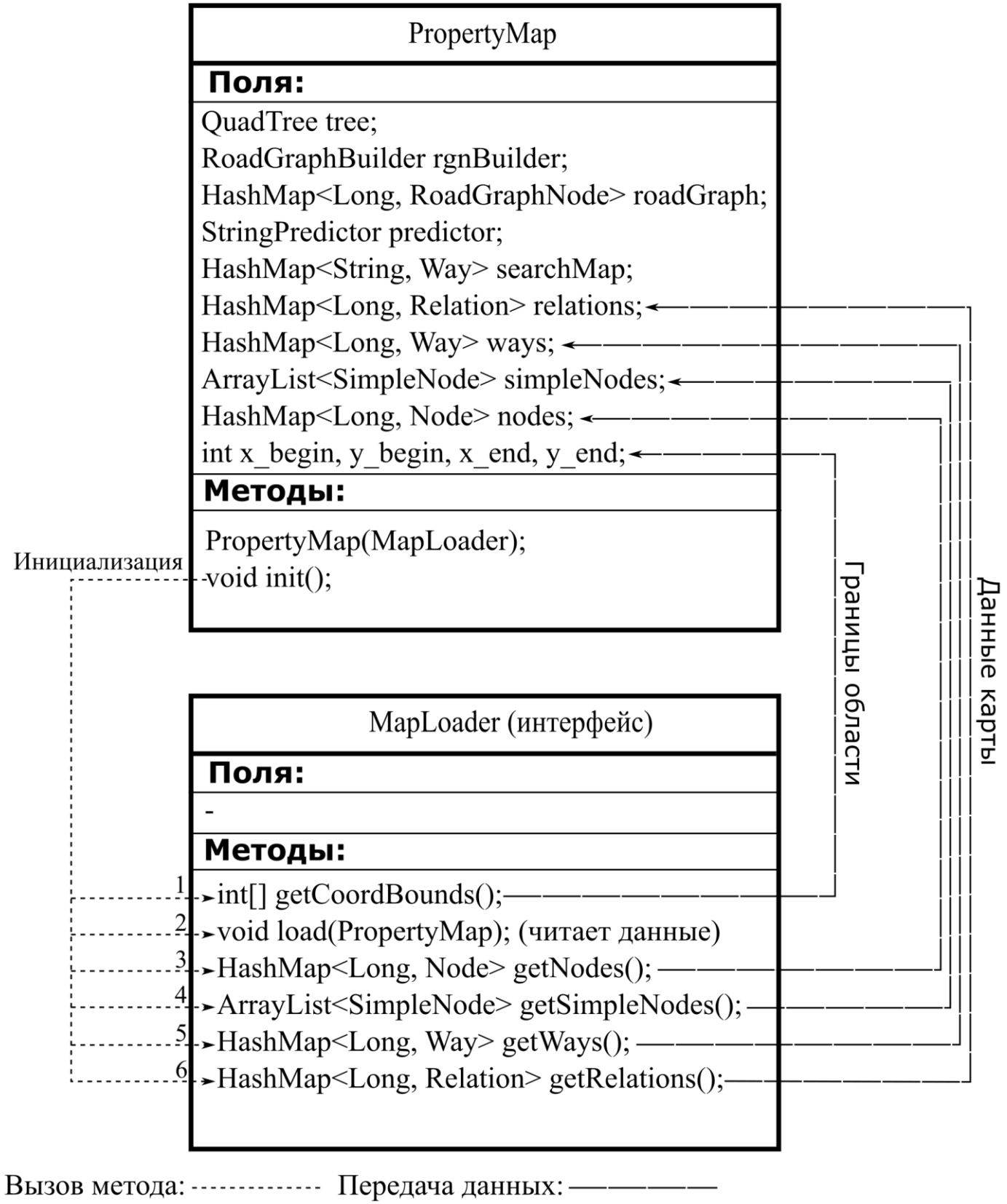
Приложение 1

Классы, представляющие геоданные.



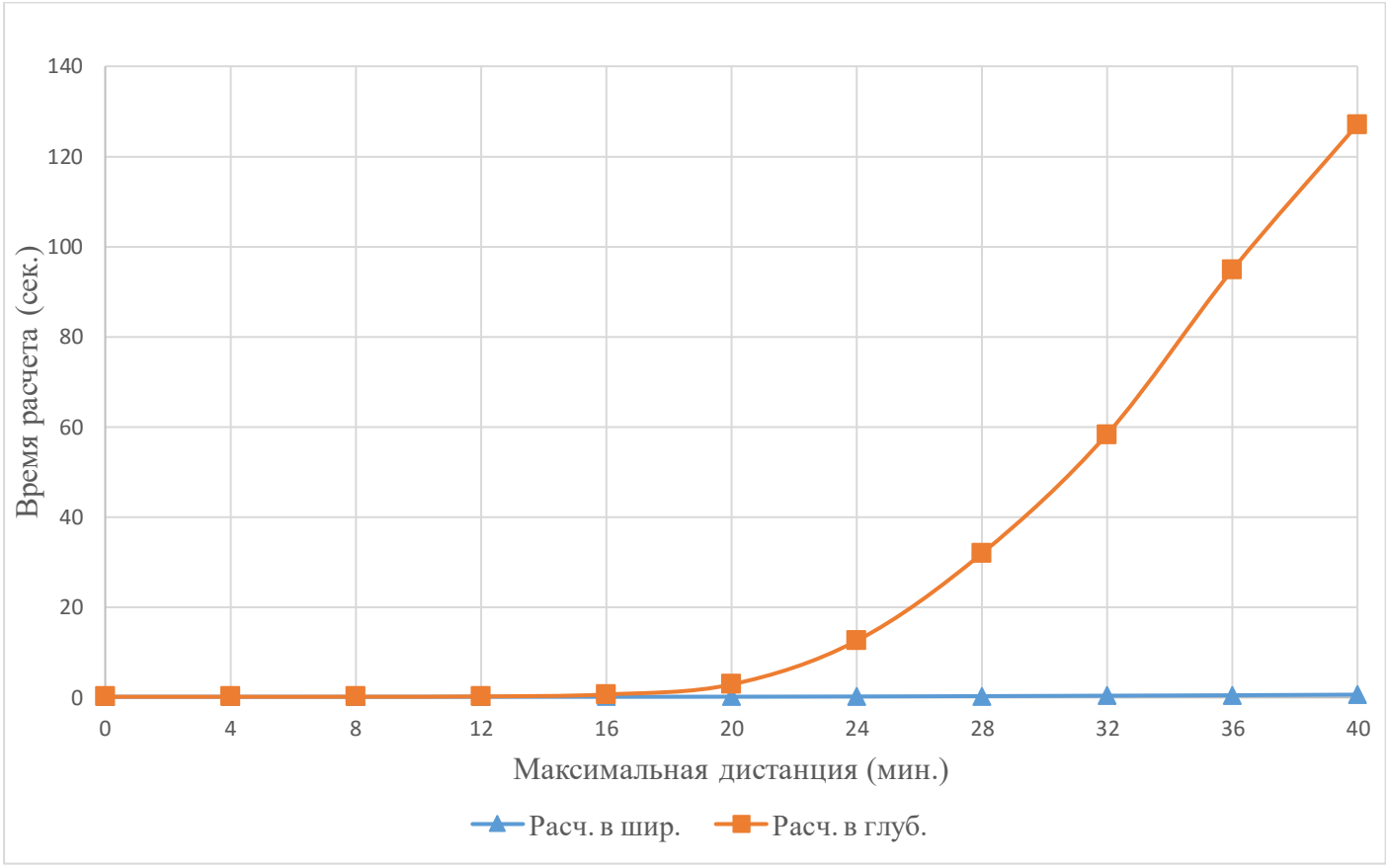
Приложение 2

Процесс загрузки геоданных из файла.



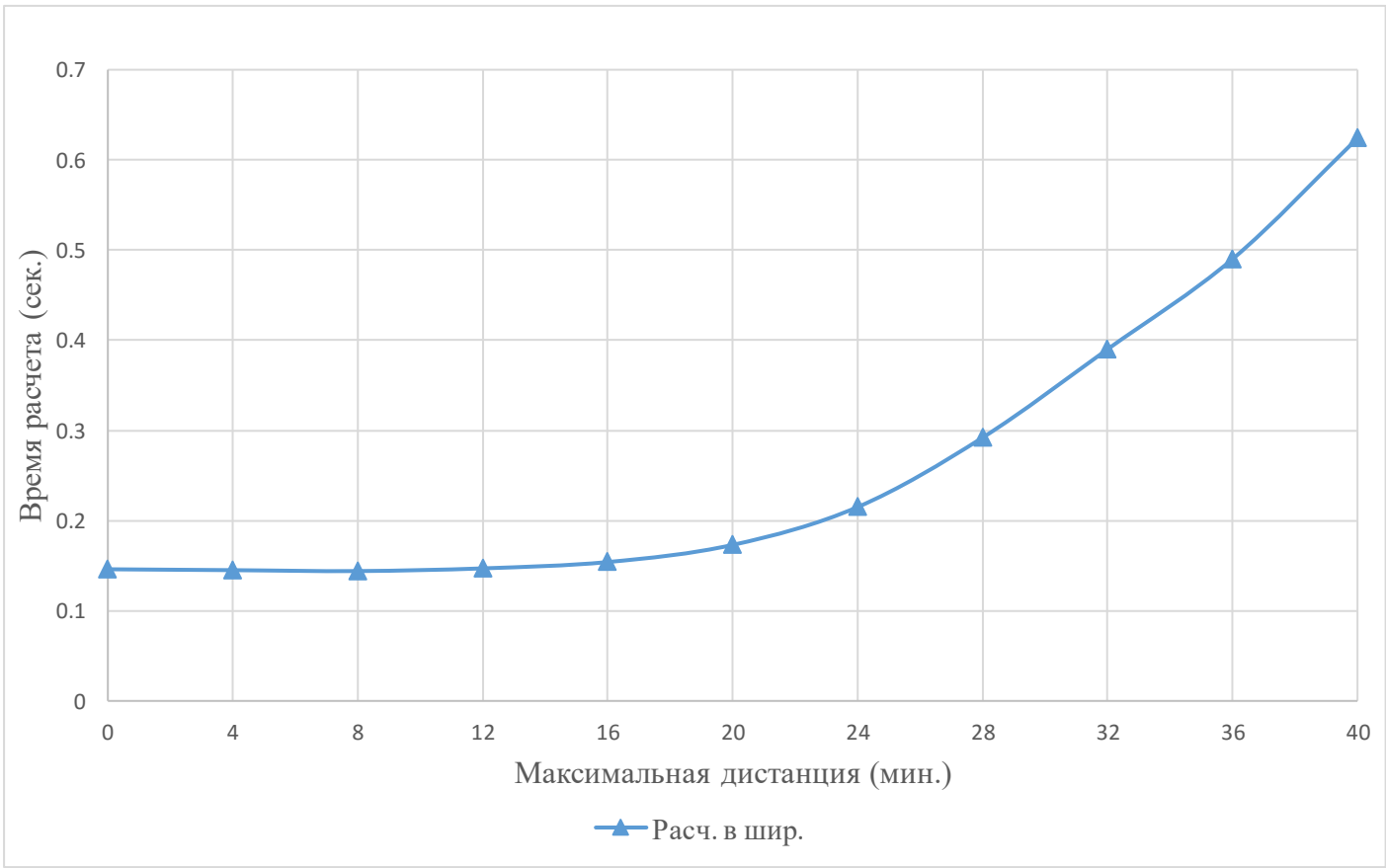
Приложение 3

Зависимость времени расчета от максимальной удаленности.



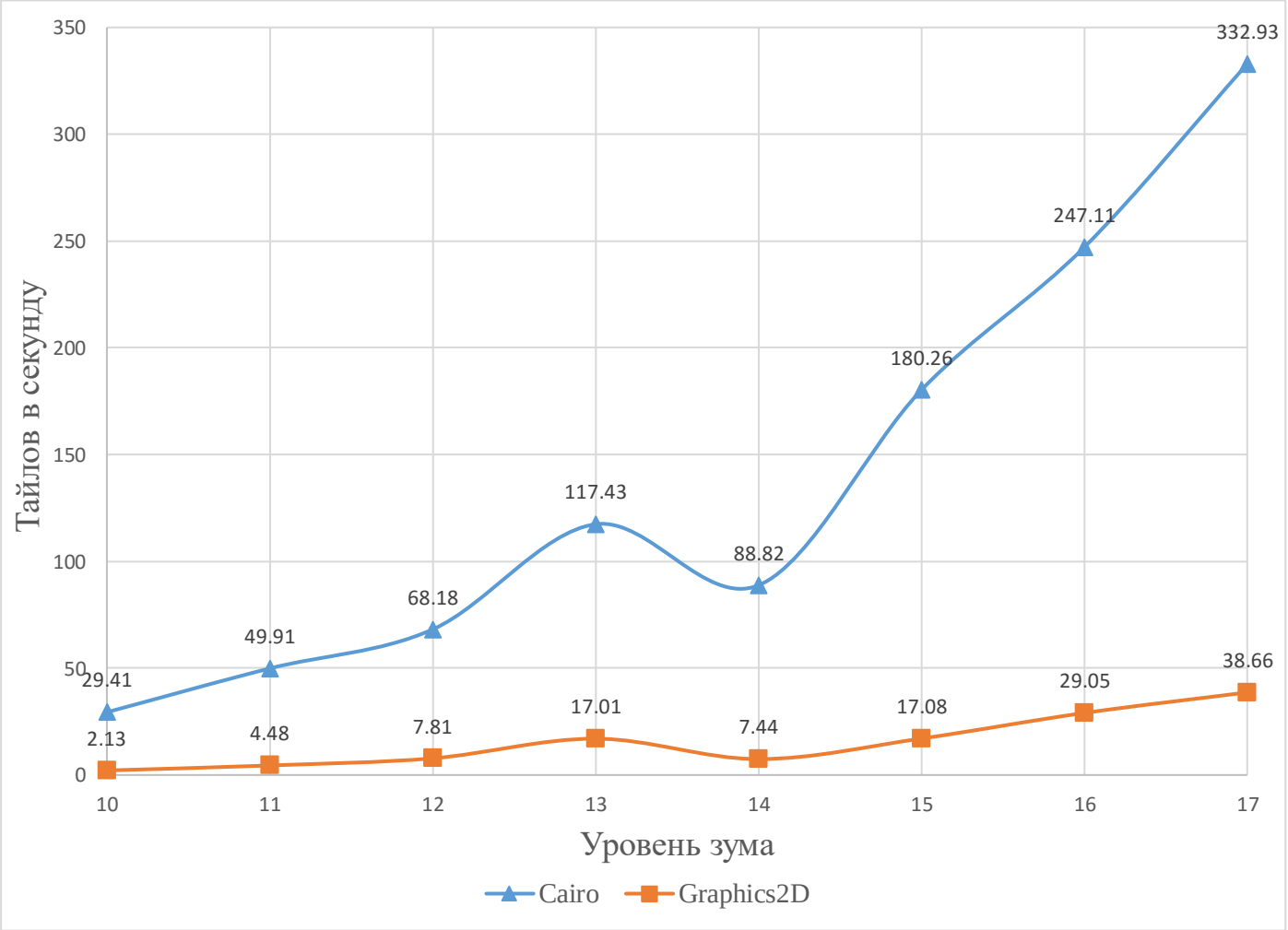
Приложение 4

График выше, без расчета в глубину.



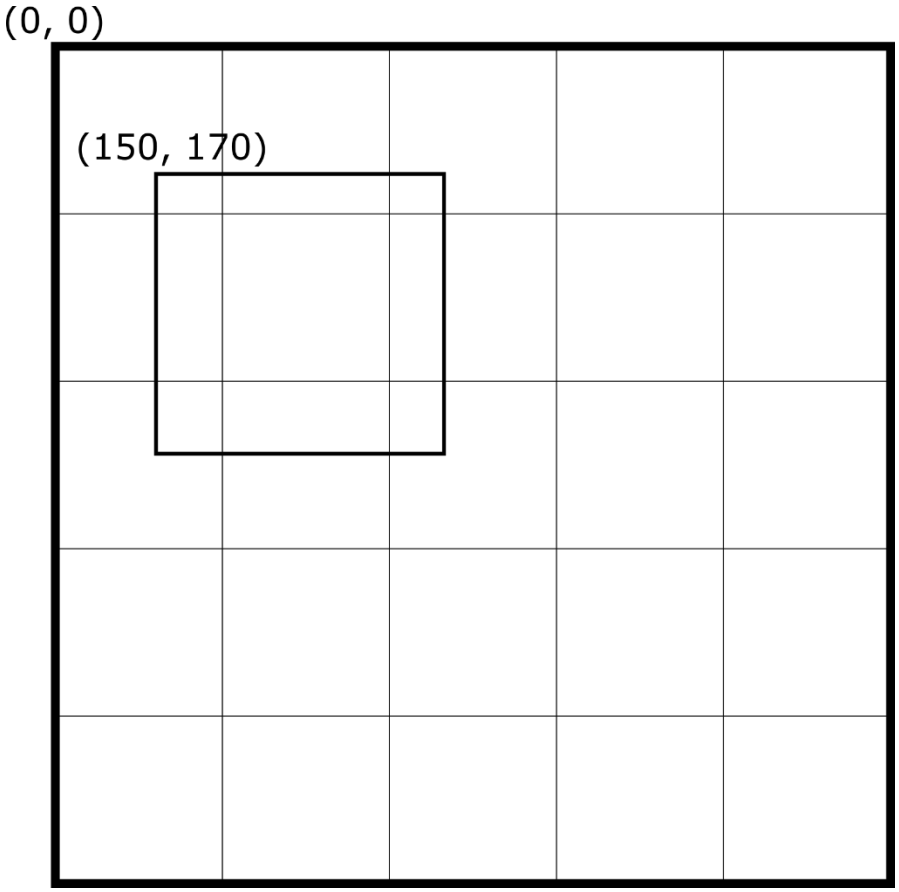
Приложение 5

Скорость генерации тайлов у Cairo и Graphics2D (8 потоков, на каждом уровне зума отрисовывались одни и те же тайлы)



Приложение 6

Координаты в клиенте



Приложение 7