УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Душан Петровић

ИСЦРТАВАЊЕ ГЕОПРОСТОРНИХ ПОДАТАКА НА УРЕЂАЈИМА СА ОПЕРАТИВНИМ СИСТЕМОМ АНДРОИД

мастер рад

Београд, 2022.

Ментор:

др Весна Маринковић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Исцртавање геопросторних података на уређајима са оперативним системом Андроид

Резиме: Рад се бави приказивањем геопросторних података на уређајима са оперативним системом Андроид. За потребе рада имплементирана је апликација *OSMRenderer*. Помоћу апликације, корисници имају могућност да преузму геопросторне податке неког мањег подручја као и да прикажу те податке на мапи. Апликација ради и без приступа интернету и може, такође да претражује најближе објекте од неке задате локације, као што су школе и болнице.

У првом делу рада говори се о подацима који су коришћени за исцртавање мапе и о пројекту *OpenStreetMap*. Други део рада је посвећен просторним базама података. Трећи део рада бави се самим исцртавањем података из просторне базе на екран уређаја. Такође у том делу биће приказани неки имплементациони детаљи апликације.

Кључне речи: Андроид, SQLite, OpenGL ES, OpenStreetMap, P стабло

Садржај

Глава 1

Увод

Геопросторни подаци представљају информације о локацији и карактеристикама географских објеката на површини Земље. На пример, геопросторни податак би могао да се односи на неку зграду, за коју би се чувао подтак о њеној локацији, као и њене карактеристике као што су: број спратова, број станара, боја зграде, година изградње итд. Једна од најзначајнијих примена геопросторних података је за потребе визуелизације, тј. за креирање географских мапа. Геопросторни подаци имају широку примену и у разним другим областима: у алатима за навигацију, просторном планирању, управљању животном средином и реаговању на катастрофе. На пример, у контексту просторног плана неке општине, геопростроним подацима би била представљена намена површина, путеви и улице, јавне службе и инфраструктурни системи.

Геопросторни подаци су некада били доступни само великим организацијама као што су војска и државне управе. Данас они налазе примену у готово свим сферама живота и велика количина геопросторних података је јавно доступна. Најзначајнији пројекат који се бави заједничким стварањем геопросторних података је *OpenStreetMap* [3]. Развојем мобилних уређаја и оперативног система Андроид као водеће платформе са преко 3 милијарде коришћених уређаја, популаризовала се и употреба дигиталних мапа. Корисници могу да преузму мапу, што им омогућава навигацију и приступ подацима без приступа интернету.

У овом раду биће описан процес исцртавања геопросторних података на уређајима са оперативним системом Андроид помоћу библиотеке *OpenGL ES* [4], као и принцип рада просторних база података. За демонстрацију рада имплементирана је апликација за Андроид *OSMRenderer* [10] у програмском

језику Котлин. Апликација ради у офлајн режиму рада и исцртава мапу на екран уређаја, кроз коју се корисник може кретати. За неку задату тачку са мапе могуће је поставити упит којим се добија k њој најближих објеката одређеног типа. Подаци коришћени у раду биће преузети са пројекта *OpenStreetMap* и чуваће се у *SQLite* [7] бази података која је за ове потребе проширена модулом за R стабло [5] како би се убрзали упити дохватања геопросторних података.

Глава 2

Подаци

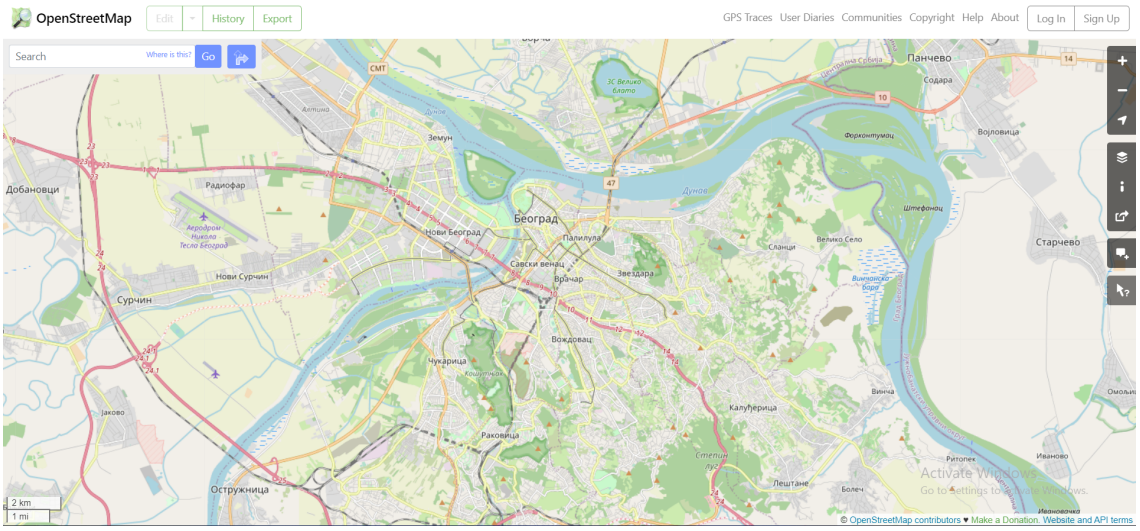
2.1 OpenStreetMap пројекат

OpenStreetMap [3] је пројекат који се бави заједничким одржавањем геопросторних података. Подаци су отворени, слободни и бесплатни за употребу без правних и техничких ограничења. Податке одржава заједница људи који добровољно раде на прикупљању података користећи ГПС уређаје, фотографије из ваздуха и друге бесплатне изворе. Након што се подаци прикупе, они се уносе у базу података коришћењем различитих софтверских алата попут веб едитора *iD* или десктоп апликације *JOSM*.

Пројекат *OpenStreetMap* је покренуо Стив Коуст 2004. године. Инспирисан успехом Википедије, као енциклопедијског пројекта слободног садржаја који сви корисници могу да допуњују и надограђују, пожелео је да направи одговарајући картографски пројекат. Пројекат се иницијално фокусирао на геопросторно мапирање Велике Британије, а касније је проширен на цео свет. У априлу 2006. године основана је фондација *OpenStreetMap* са циљем да подстакне развој и дистрибуцију бесплатних просторних података и обезбеди просторне податаке доступне свим заинтересованим корисницима.

Подаци расположиви унутар пројекта *OpenStreetMa* могу се користити за исцртавање географских мапа (пример на слици 2.1), за навигацију и за геокодирање[1]. Овај пројекат данас има преко 2 милиона корисника, а неки од најзначајнијих су компаније попут Епла, Убера, Мајкрософта и Мете.

[1]Геокодирање је поступак претварања улазног текста, који представља адресу, у географске координате.



Слика 2.1: Веб клијент пројекта *OpenStreetMap*.

2.2 OSM структура података

Пројекат *OpenStreetMap* за чување података користи једноставну структуру података, која се састоји од четири врсте елемената: чворова, путања, веза и ознака. Структура података представља колекцију, у којој су прво наведени сви чворови, па све путање и на крају све везе.

Чворови (енг. Nodes) представљају тачке са географским координатама по стандарду *WGS 84*[2]. Могу да представљају делове путања или независне географске тачке на карти попут локација ресторана или врхова планина.

Путање (енг. Ways) представљају изломљене линије на мапи и састоје од листе чворова. Сваки чвор представља тачку изломљене линије и ако су први и последњи чвор у листи једнаки онда је путања затворена изломљена линија, односно полигон. На пример, на карти путање могу означавати линијске податке као што су улице или површинске као што су шуме или језера.

Везе (енг. Relations) служе за дефинисање релација између два или више елемената и састоје од листе чворова, путања и других веза који се једним именом називају *чланови* (енг. members). Сваки члан везе има своју *улогу* (енг. role). Пример везе би било представљање трибина фудбалског стадиона у којој би један члан била путања са улогом спољашњег полигона а други члан путања са улогом унутрашњег полигона.

[2] *WGS 84* стандард дефинише елипсоид и његове координате којим се апроксимира облик Земље.

Ознаке (енг. Tags) су парови кључева и вредности. Они ближе одређују шта тачно представља неки елемент. На пример болница на мапи би имала ознаку са кључем `building` и вредношћу `hospital`.

Комплетни подаци целог света пројекта *OpenStreetMap* доступни су на адреси https://planet.openstreetmap.org. Податке је могуће преузети за неки одређен регион, омеђен минималном и максималном географском ширином и минималном и максималном географском дужином, коришћењем различитих алата. Један од таквих алата је *Overpass API*[3]. *OpenStreetMap* подаци се могу представити у два различита формата: у текстуалном формату *XML* и у бинарном формату *PBF*. Предност формата *PBF* је у томе што заузима мање меморије, док је предност формата *XML* то што је читљивији. Приликом развоја апликације *OSMRenderer* коришћен је формат *XML* података. У наставку текста дат је пример геопросторних података у формат *XML* који се односе на Нови Београд. Веза у примеру представља границе месне заједнице Младост, путања представља Гандијеву улицу, први чвор је један од чворова путање, док други чвор представља аутобуску станицу:

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap␣0.0.2">
 <bounds
    minlat="44.8088900"
    minlon="20.3809400"
    maxlat="44.8105500"
    maxlon="20.3851500"
/>
 <node
    id="2172227517"
    visible="true"
    version="4"
    changeset="110750507"
    timestamp="2021-09-05T13:50:04Z"
    user="TXBG"
    uid="5569366"
    lat="44.8086784"
    lon="20.3860018"
/>
 <node
```

[3]Алат *Overpass API* доступан је са адресе https://www.openstreetmap.org/export.

```
    id="10226427594"
    visible="true"
    version="2"
    changeset="129716901"
    timestamp="2022-12-04T20:11:03Z"
    user="Branko␣Kokanovic"
    uid="95504"
    lat="44.8098425"
    lon="20.3829361">
        <tag k="bus" v="yes"/>
        <tag k="name" v="Vojvođanska"/>
        <tag k="network" v="BG␣Prevoz"/>
        <tag k="operator" v="GSP␣Beograd"/>
        <tag k="public_transport" v="stop_position"/>
</node>
...
<way
    id="207097162"
    visible="true"
    version="6"
    changeset="110750507"
    timestamp="2021-09-05T13:50:05Z"
    user="TXBG"
    uid="5569366">
        <nd ref="2172227517"/>
        <nd ref="2172227521"/>
        <tag k="highway" v="service"/>
        <tag k="name" v="Gandijeva"/>
        <tag k="name:en" v="Gandijeva"/>
        <tag k="name:etymology:wikidata" v="Q1001"/>
        <tag k="surface" v="asphalt"/>
</way>
...
<relation
    id="9426445"
    visible="true"
    version="6"
    changeset="116498961"
    timestamp="2022-01-23T13:22:53Z"
    user="Srdjan02131"
    uid="13595876">
        <member type="way" ref="678924623" role="outer"/>
```

```
        <member type="way" ref="758447058" role="outer"/>
        <member type="way" ref="678924607" role="outer"/>
        <member type="way" ref="678924590" role="outer"/>
        <member type="way" ref="678924588" role="outer"/>
        <member type="way" ref="678924587" role="outer"/>
        <tag k="admin_level" v="10"/>
        <tag k="boundary" v="administrative"/>
        <tag k="name" v="MZ␣Mladost"/>
        <tag k="name:en" v="Mladost"/>
        <tag k="ref:RS:mesna_zajednica" v="737950"/>
        <tag k="type" v="boundary"/>
 </relation>
 ...
</osm>
```

2.3 SQLite

SQLite [7] је библиотека која пружа могућност организације локалних података у облику релационе базе података и њихово коришћење путем елементарних *SQL* наредби. Изворни код система *SQLite* је у јавном власништву и написан је у програмском језику *C*.

SQLite је један од најкоришћенијих система за управљање базама података јер је уграђен у велики број популарних оперативних система, прегледача за веб и осталих уграђених система. На пример, уграђен је у оперативне системе Андроид, *iOS* и *Windows 10*, и у прегледаче *Google Chrome* и *Mozilla Firefox*. Доступност овог система за управљање базама података у наведеним системима у великој мери утиче на широку распрострањеност коришћења овог система.

Разлог популарности СУБП *SQLite* лежи у његовом дизајну. Он је конципиран тако да програм може да га користи и без инсталирања СУБП. За разлику од клијент-сервер система за управљање базама података, језгро СУБП *SQLite* није самосталан процес са којим апликација комуницира. Уместо тога, *SQLite* библиотека је увезана и саставни је део апликације. Апликација користи функционалност рада са базама података кроз просте функцијске позиве. Они повећавају брзину приступа базама података јер су функцијски позиви унутар једног процеса ефикаснији од међупроцесне комуникације.

Комплетна база података (њена дефиниција, табеле, индекси и сами подаци) је сачувана као једна вишеплатформна датотека на матичној машини, што омогућава да више процеса и нити приступе бази у исто време. Овакав дизајн је постигнут закључавањем комплете датотеке базе података за писање на почетку трансакције.

SQLite имплементира већину елемената препоручених *SQL-92* стандардом. Неке од функционалности предвиђене стандардом које *SQLite* не подржава су подршка за рекурзивне окидаче и могућност писања у погледима. *SQLite* прати *PostgreSQL* синтаксу. Једина битнија разлика је та што *PostgreSQL* користи статичке типове, док *SQLite* користи динамичке типове.

2.4 Претпроцесирање података

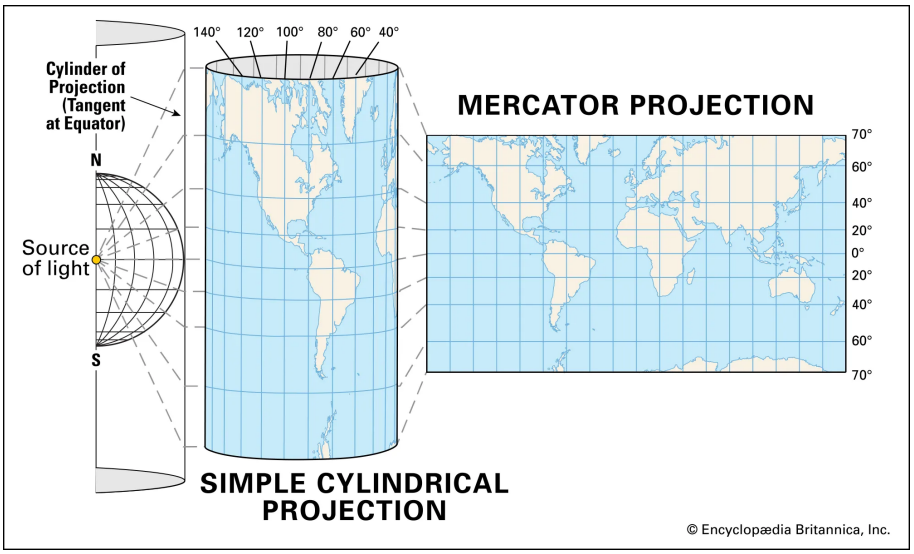
Како се на оперативном систему Андроид подаци чувају у *SQLite* бази података, податке је потребно конвертовати из формата *XML* у табеле *SQLite* базе података. За тај процес може се искористити *Python* скрипт osm2sqlite [9] који је прилагођен за потребе овог мастер рада. Скрипт функционише тако што најпре креира базу података и све потребне табеле, а потом парсира сваку *XML* ознаку и попуњава ред у одређеној табели. На крају се бришу непотребни подаци из одређених табела као што су чворови који немају ознаку и креирају се табеле са просторним индексима. База података која се креира се састоји од следећих табела:

1. Табела `nodes` која садржи идентификатор чвора и његове координате у *Меркаторовој пројекцији* (енг. Mercator projection). Меркаторова пројекција је цилиндрична картографска пројекција која трансформише сферне координате (координате које су записане у *WGS 84* стандарду) тачке са површине Земље у Декартов координатни систем, како би се лакше приказале на екрану уређаја (слика 2.2). Формуле којима се врши пројекција су [12]:

$$x = \lambda \cdot R$$

$$y = \ln(\tan(\frac{\pi}{4} + \frac{\varphi}{2})) \cdot R$$

где су са φ и λ означене редом географска ширина и географска дужина тачке са сфере којом је апроксимирана површина Земље, а са $R = 6371\text{km}$ полупречник Земље.



Слика 2.2: Приказ пројектовања сфере у омотач ваљка Меркаторове пројекције.

node_id	lat	lon
25264038	2281258.3134535	5595513.79108356
25264039	2281206.98403629	5595117.07200944
25264052	2280922.26217469	5593101.51871232

Табела 2.1: Прва три реда табеле `nodes`, где је са `lon` означена x координата, а са `lat` y координата у Меркаторовој пројекцији.

2. Табела `way_nodes` везује идентификатор путање са свим чворовима које он садржи. Редослед чворова у путањи је у истом поретку као у табели.

way_id	node_id
5210018	3936424954
5210018	3936424955
5210018	1858130583

Табела 2.2: Прва три реда табеле `way_nodes`, која садрже информације о путањи која садржи три дата чвора.

3. Табела `way_tags` везује идентификатор путање са свим паровима кључа и вредности ознака које припадају тој путањи.

way_id	key	value
5210018	highway	tertiary
23070200	building	office
23073175	leisure	park

Табела 2.3: Прва три реда табеле `way_tags` која садрже информације о ознакама путања. На пример други ред нам говори да путања представља зграду која се користи као канцеларијски простор.

4. Табела `relation_members` у сваком реду садржи информације о неком члану везе. Члан везе је дефинисан идентификатором везе, типом, улогом и референцом на елемент који може бити чвор или путања.

relation_id	type	ref	role
3633	way	434687241	outer
3633	way	434604431	outer
3633	way	295468574	inner

Табела 2.4: Прва три реда табеле `relation_members` која садрже информације о члановима везе.

5. Табела `relation_tags` везује идентификатор везе са свим паровима кључа и вредности ознака које припадају тој вези.

relation_id	key	value
3633	natural	water
138185	building	stadium
2539464	landuse	forest

Табела 2.5: Прва три реда табеле `relation_tags` која садрже информације о ознакама везе. У првом реду приказана је веза која представља водену површину, у другом реду је веза која представља стадион, док је у трећем реду веза која представља шуму.

6. Табеле `rtree_way1`, `rtree_way2`, `rtree_relation1` и `rtree_relation2` представљају просторне табеле (о којима ће бити више речи у поглављу 3) које садрже идентификатор путање и њену минималну и максималну

географску ширину и минималну и максималну географску дужину. Просторне табеле омогућавују брз проназак свих елемената чије геометрије имају пресек са неким задатим граничним оквиром. Геометрије елемената су дефинисане као правоугаоници одређени минималним и максималним вредностима географске дужине и географске ширине елемента. *Гранични оквир* (енг. bounding box) је такође правоугаоник који се задаје помоћу минималне и максималне географске ширине и минималне и максималне географске дужине приликом упита, како би се пронашли елементи који имају пресек са њим. Табеле чувају путање или везе и чувају различите податке за различите размере карте. На пример, на размерама које су мање од 1:8000, желимо да прикажемо више података како би приказ био детаљнији, док на размерама већим од 1:8000 немамо потребу да исцртамо све податке, као што су неке мање улице или појединачне зграде. Подаци на мањој размери се чувају у табелама `rtree_way1` и `rtree_relation1`, док се подаци у случају већих размера чувају у табелама `rtree_way2` и `rtree_relation2`. Гранични оквир је већи на већој размери па упит за тај оквир проналази више података и стога се спорије извршава, што је још један добар разлог да се на већим размерама изоставе неки подаци.

way_id	min_lat	max_lat	min_lon	max_lon
3444666320	5580380.5	5582265.5	2267836.25	2270563.25
689922776	5580321.5	5581440.5	2268103.75	2269609.0
444832544	5581423.0	5581926.5	2268862.25	2269182.25

Табела 2.6: Прва три реда табеле `rtree_way1`, у којој се чувају информације о путањама на мањој размери карте са њиховим минималним и максималним географским ширинама (`min_lat` и `max_lat`) и минималним и максималним географским дужинама (`min_lon` и `max_lon`).

Глава 3

Просторне базе података

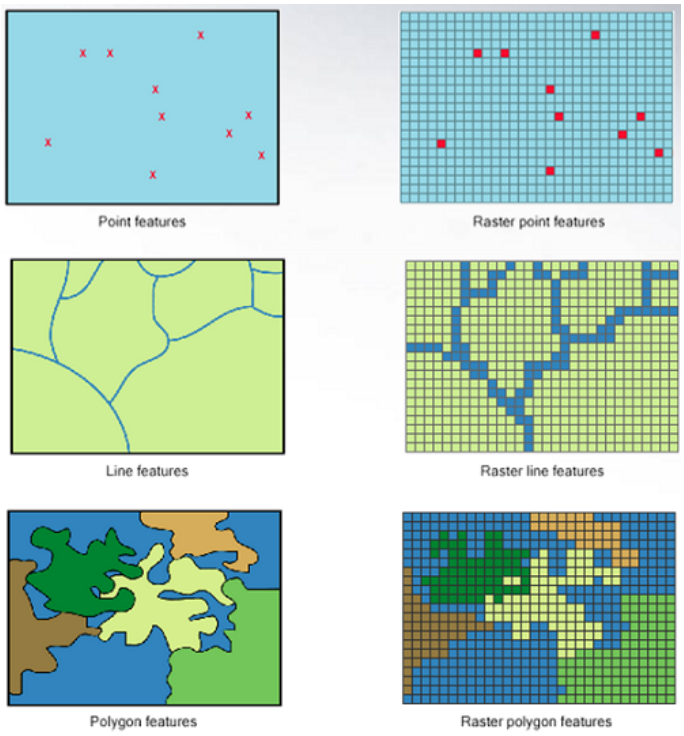
Просторне базе података (енг. spatial databases) [11] су базе података које су проширене геометријским атрибутом, којим се задаје информација о томе где се податак налази у простору. Такве базе подржавају све класичне упите, као што су `SELECT`, `INSERT`, `DELETE` и `UPDATE`, али поред њих подржавају и *просторне упите* (енг. spatial query), као и *просторне индексе* (енг. spatial index). Просторни упити омогућавају претрагу по геометријском атрибуту, а просторни индекси служе за организацију података тако да се просторни упити брже извршавају. Просторни подаци углавном представљају објекте који се налазе негде на Земљиној површини, и они су приказани помоћу координата у некој картографској пројекцији како би их било могуће приказати на мапи (табела 3.1). Они се могу приказати на географској карти и анализирати помоћу бројних алата.

Просторни подаци могу бити растерски или векторски (слика 3.1). Растерски подаци представљају површине, чији је основни геометријски елемент пиксел. Другим речима, они представљају мрежу чије су ћелије квадрати једнаке величине и којима је додељена вредност боје. Векторски подаци се задају као колекција тачака, линија и полигона и њихових комбинација. У овом раду су коришћени векторски подаци јер су подаци пројекта *OpenStreetMap* у том облику.

Неки од примера просторних база података су: *PostgreSQL* са просторном екстензијом *PostGIS*, *Oracle Spatial*, *MySQL* са геометријским типом података, са проширењем за P стабло.

Табела 3.1: Пример просторне табеле у којој су приказани градови Србије заједно са информацијом о броју становника и координатама у *WGS 84* координатном систему, које представљају геометријски податак табеле

Град	Геометрија	Број становника
Београд	POINT(44.8125, 20.4612)	1166000
Нови Сад	POINT(45.2396, 19.8227)	289000
Ниш	POINT(43.3209, 21.8954)	185000
Крагујевац	POINT(44.0128, 20.9114)	150000
Суботица	POINT(46.0970, 19.6576)	103000



Слика 3.1: Приказ векторских и растерских података

3.1 Просторни индекси

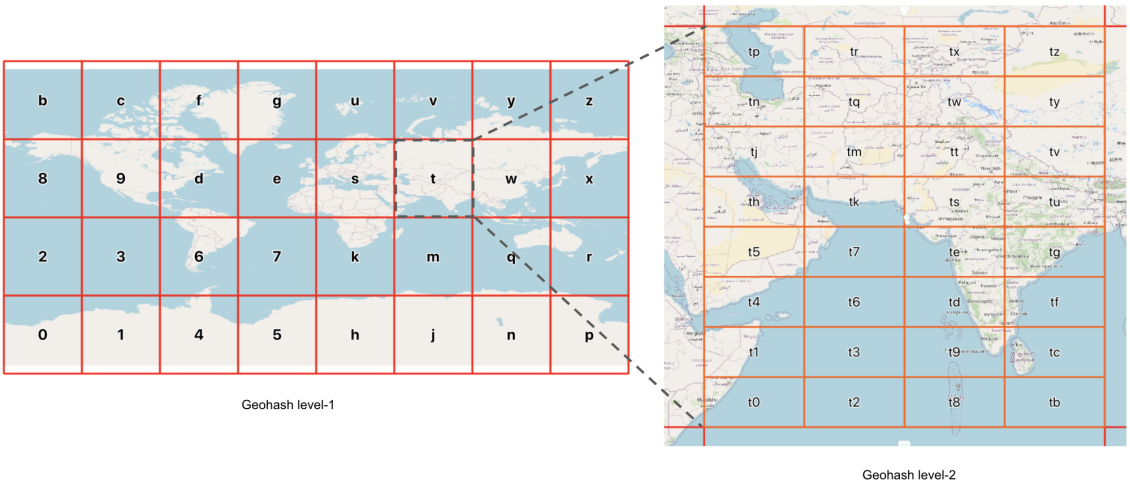
Просторни индекси представљају структуре података помоћу којих се индексирају просторни подаци у бази. Они се користе да би се оптимизовали просторни упити. У наставку ће бити описани неки од најкоришћенијих просторних индекса.

Геохеш

Геохеш [8] је систем енкодирања географских површина у ниске слова и бројева. Технички гледано, геохеш је хијерархијска структура података која подразумева поделу мапе света на мрежу од 32 ћелије једнаке величине, у 4 реда и 8 колона, тако да свака ћелија представља један карактер у нисци. Свака ћелија се може даље поделити на 32 нове ћелије, сада у 8 редова и 4 колоне, чији се карактер надовезује на претходне карактере како би се генерисала ниска. Поделе се настављају са наизменичним бројем колона и редова све док се не дође до довољно мале површине, што се обично дешава након десетак подела. Геохеш подела је илустрована на слици 3.2. За карактере којима се кодирају ћелије користе се све цифре и сва слова енглеске абецеде осим слова a, i, o и l. Наиме, пошто вршимо поделу на 32 ћелије, а укупно постоји 36 цифара и слова енглеске абецеде, било је потребно избацити нека 4 карактера.

Геохеш ниске имају својство да што је већи њихов заједнички префикс, то су локације које оне представљају и ближе у простору. Наиме, пошто геохеш ниске представљају географске површине, што је већи број карактера у нисци, то је површина коју ниска кодира мања. То значи да ће се две површине кодиране геохеш нискама са већим заједничким префиксом налазити у истој мањој ћелији и самим тим биће ближе једна другој. Међутим, обрнуто не важи: може се десити да се површине кодиране геохеш нискама без заједничког префикса налазе једна поред друге. На пример, на слици 3.2 површине кодиране нискама tp и sz су једна поред друге али немају заједнички префикс.

Пошто су геохеш ниске једнодимензионални подаци, а површине које оне представљају дводимензионалне, потребно је извршити декодорање како би се за дату ниску пронашла површина коју она представља. То се може постићи помоћу *З-кривe* (енг. z-ordered curve). З-крива је функција која пресликава бинарне бројеве у дводимензионалне податке, тако што битови на непарним

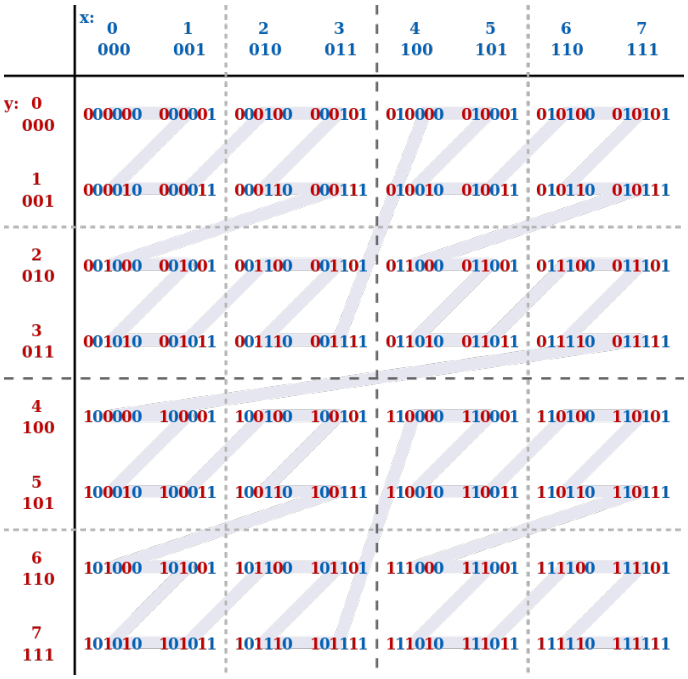


Слика 3.2: Геохеш подела света на ћелије

позицијама одређују y координату, док битови на парним позицијама одређују x координату. На слици 3.3 приказан је поступак обиласка криве. Обилазак креће од крајњег левог бита и сваки бит дели простор на два дела. На пример ако је први бит 0 онда је површина у горњој половини простора, а ако је 1 онда је површина у доњој половини простора. Сада смо добили нови простор и прелазимо на следећи бит. Ако је други бит 0 онда је површина у левој половини простора, а ако је 1 онда је површина у десној половини простора. Пошто је азбука којом се кодирају ниске величине 32, сваки карактер ниске може се кодирати помоћу петобитног бинарног броја. Надовезивањем карактера у нисци у бинарном запису добија се улазни податак за З-криву.

kd-стабло

kd-стабло [1] је структура података којом се врши подела k-димензионог простора на начин који омогућава организацију података у k-димензионом простору. У случају геопросторних података тај простор је дводимензионалан. Можемо га посматрати као бинарно стабло чији су чворови геопросторни подаци, а деца неког чвора се налазе са супротних страна праве паралелне са неком координатном осом која пролази кроз тачку тог чвора. На пример, деца e и b чвора f са слике 3.4 налазе се са различитих страна праве $x = x_f$,



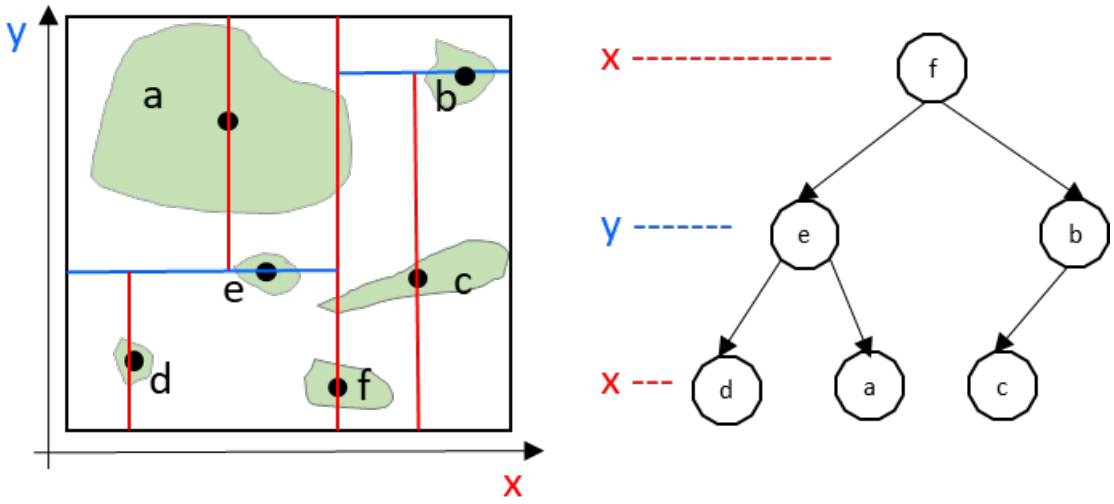
Слика 3.3: Обилазак Z-криве

а d и а са различитих страна праве $y = y_e$.

kd-стабло димензије 2 се конструише узастопном поделом скупа тачака на два подскупа према медијани по x, односно y оси, наизменично. Најпре се проналази тачка која представља медијану скупа тачака по x оси, и она представља корен стабла. Даље се подаци деле на два подскупа, на оне чија је x координата мања или једнака x координати тачке у корену и на оне чија је x координата већа, и за сваки од подскупова се проналази тачка која представља медијану по y оси. Те тачке представљају чворове који су деца корена *kd*-стабла. Добијени подскупови се даље деле на два подскупа, на тачке чија је y координата мања или једнака и на тачке чија је је y координата већа од y координате тачке новог чвора, па се опет проналази медијана сваког од подскупова по x оси, и процес поделе се понавља све док су добијени подскупови тачака непразни. На крају ће се у левом подстаблу произвољног чвора наћи тачке чија је x, односно y координата мања или једнака од његове x, односно y координате, а у десном подстаблу тачке чија је x, односно y координата већа. Како се у сваком кораку врши подела по медијани, за произвољни чвор стабла важи да има једнак број потомака у левом и у десном подстаблу, тако да ћемо на крају добити балансирано стабло.

Додавање и брисање елемената из *kd*-стабла се врши на исти начин као код

уређених бинарних стабала претраге, с тим да треба водити рачуна о томе да стабло после промена остане балансирано.

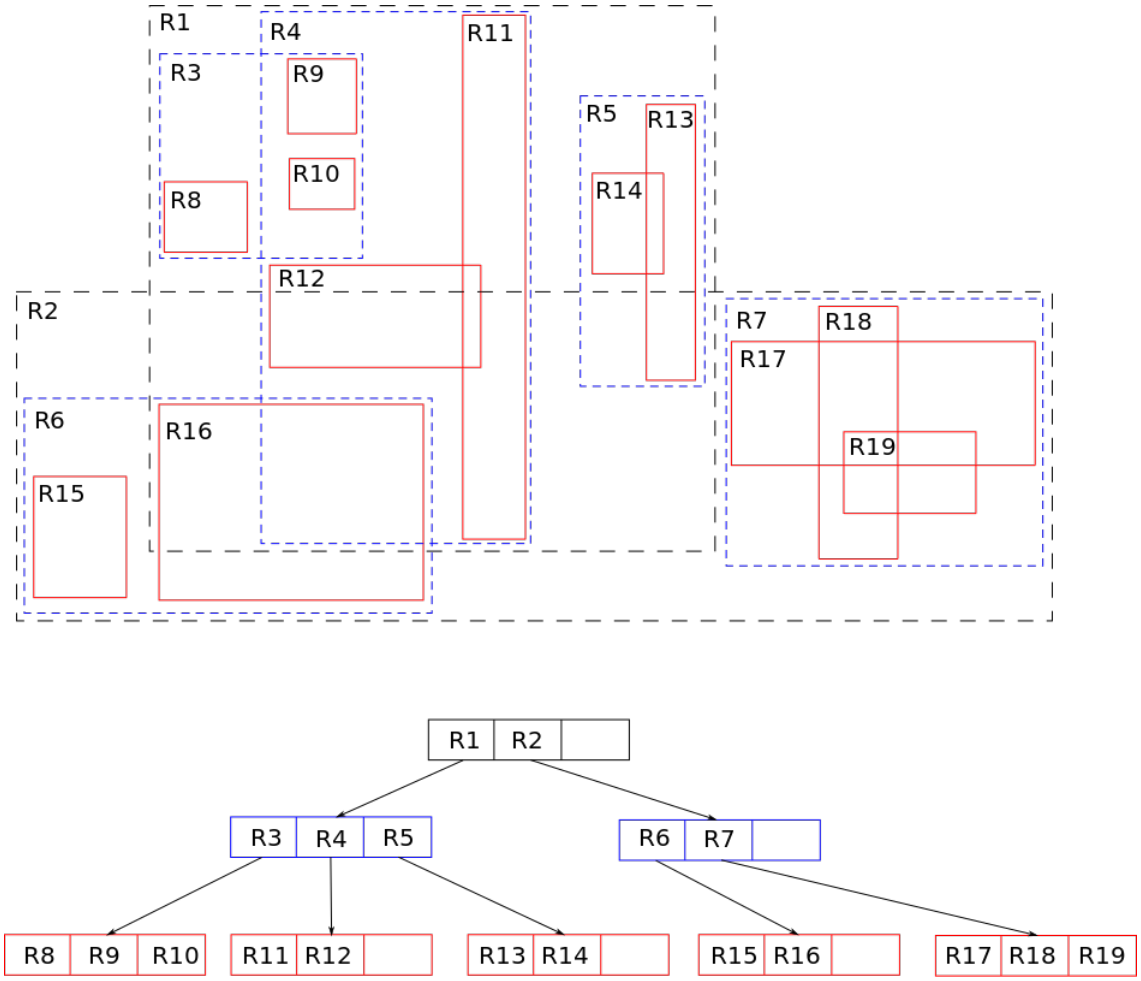


Слика 3.4: kd-стабло

P Стабло

P стабло [5] је дрволика структура података која се користи за просторно претаживање података. Добила је назив по енглеској речи за правоугаоник (енг. rectangle) јер је сваки чвор у стаблу представљен као минимални гранични оквир података који се налазе у потомцима тог чвора. Просторни подаци се налазе у листовима стабла, а листови се групишу тако што се одаберу просторно блиски елементи и око њих се направи минимални гранични оквир. Такав оквир одговара унутрашњем чвору у стаблу и он ће бити родитељски чвор листовима чијим је груписањем настао. Процес груписања се наставља: суседни унутрашњи чворови се групишу и њихов гранични оквир постаје њихов родитељски чвор. Овај поступак се завршава када сви елементи буду обухваћени једним чвором и тај чвор биће корен P стабла. Чворови P стабла могу међусобно да се преклапају. Минимални и максимални број деце који сваки чвор може да има је унапред одређен. На слици 3.5 приказано је једно P стабло.

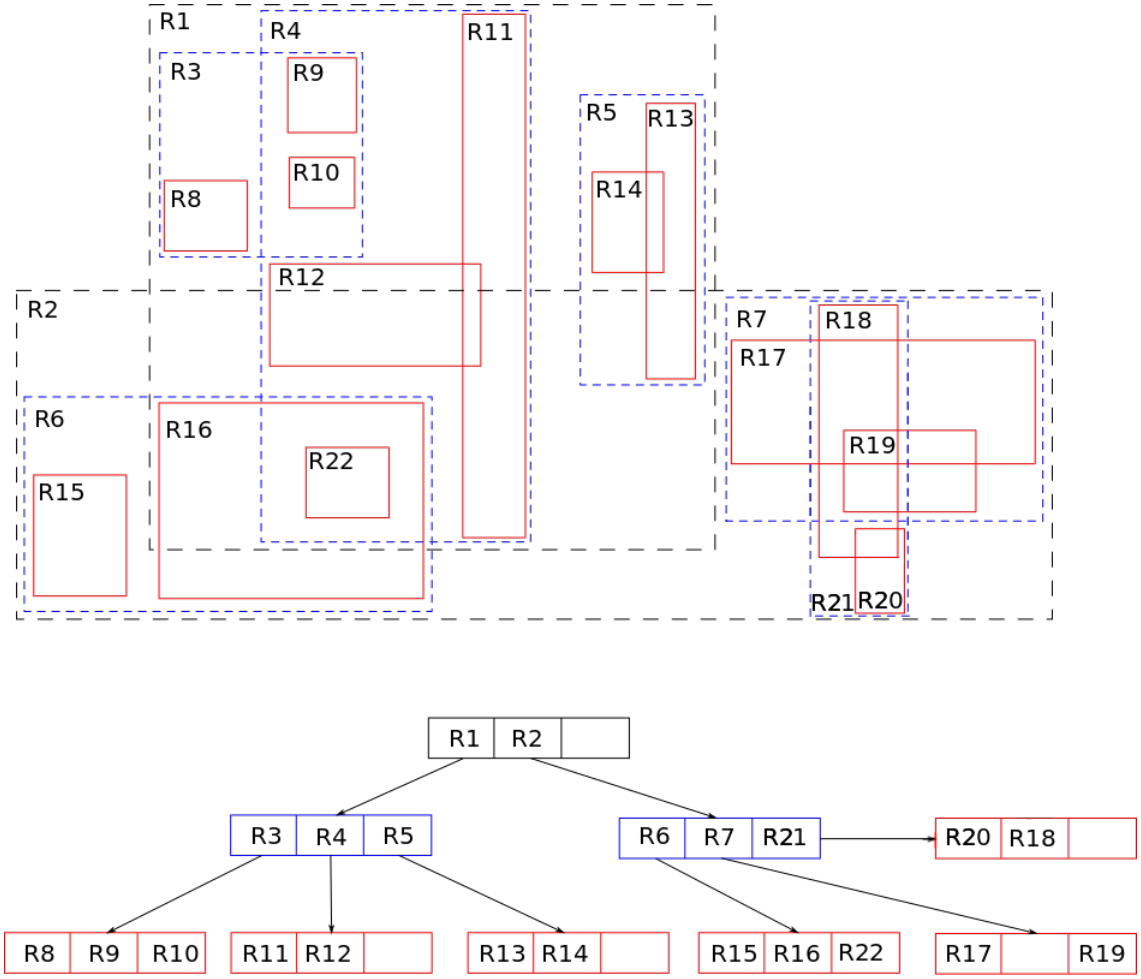
Да би се у P стабло додао нови елемент потребно је рекурзивно обићи стабло од корена до одговарајућег листа. У сваком кораку разматрају се сва деца текућег чвора као кандидати у чије подстабло би могао да се убаци нови



Слика 3.5: Пример R стабла, где су црвеним правоугаоницима обележени просторни подаци, док су плавим и црним правоугаоницима означени њихови гранични опсези који одговарају унутрашњим чворовима R стабла. За сваки чвор овог R стабла важи да може да има максимално три детета, а минимално два детета.

елемент. Бира се чвор за који важи да је додавањем овог елемента у његово подстабло његов минимални гранични оквир потребно најмање проширити. Када се током обиласка стабла стигне до последњег чвора који није лист, ако он нема максималан дозвољен број деце, онда се нови елемент умеће као лист са тим чвором као родитељем. У случају да чвор већ има максималан број деце онда се он мора партиционисати, тј. поделити на два чвора. Поступак партиционисања се рекурзивно понавља за све чворове стабла који се налазе на путањи од корена до новог елемента и који имају више од максималног дозвољеног броја деце. На слици 3.6 је илустрован поступак додавања нових

чворова у P стабло са слике 3.5.



Слика 3.6: Пример додавања нових чворова у P стабло са слике 3.5. Додавањем чвора R20 потребно је партиционисати чвор R7 на R7 и R21. Приликом додавања чвора R22 није потребно вршити партиционисање.

Када се брише елемент у листу стабла потребно је проверити да ли након његовог брисања његов родитељ остаје са мање деце од унапред задатог минималног броја деце сваког чвора. Ако је то случај онда се бришу чворови на путањи од корена до тог чвора све док постоје чворови са мање деце од тог унапред задатог броја. Када се ти чворови обришу потребно је листове обрисаних чворова опет уметнути у стабло. То се ради претходно описаним поступком убацивања нових елемената у P стабло.

Геохеш, *kd*-стабло и P стабло су једни од најчешће коришћених просторних индекса, а у овом раду за имплементацију просторних индекса су коришћена су P стабла јер су у пракси најкоришћенија и *SQLite* има подршку само за њих.

На слици 3.7 су приказани резултати извршавања два упита који проналазе све геопросторне елементе унутар једног граничног оквира. Видимо да је упит са просторним индексом много бржи и он се извршава за 18 милисекунди, док се упит без просторног индекса извршава за 1436 милисекунди.

3.2 Просторни упити

Поред чувања просторних података, главна одлика просторних база података су и просторни упити. Они нам омогућавају да користимо геометријске функције, као што је испитивање пресека и рачунање раздаљине, за одговарање на питања о простору као и о објектима у простору. Три основне врсте упита су:

1. проналазак свих елемената у неком правоугаонику претраге,
2. проналазак k најближих суседа за неку задату тачку,
3. одређивање просторног односа између два геопросторна податка (пресек, разлика, итд.).

Поред горе наведених упита, просторне базе података подржавају и друге геометријске упите као што су проналазак удаљености између два геопросторна елемента, одређивање површине полигона као и трансформације између два координатна система. Просторни упити се, такође, могу комбиновати са класичним упитима. Тако, на пример, можемо да пронађемо три нама најближа ресторана који имају оцену већу од 4,5.

У овој секцији биће описана реализација неких просторних упита над подацима чуваним у P стаблу.

SQL 1

```
1   SELECT w.way_id FROM rtree_way1 r
2   JOIN way_nodes w ON r.way_id = w.way_id
3   JOIN nodes n ON w.node_id = n.node_id
4   JOIN way_tags t ON t.way_id = w.way_id
5   WHERE min_lon < 2277337.5 AND max_lon > 2276721.5
6   AND min_lat < 5591241.0 AND max_lat > 5589947.0
```

```
Execution finished without errors.
Result: 2920 rows returned in 18ms
At line 1:
SELECT w.way_id FROM rtree_way1 r
JOIN way_nodes w ON r.way_id = w.way_id
JOIN nodes n ON w.node_id = n.node_id
JOIN way_tags t ON t.way_id = w.way_id
WHERE min_lon < 2277337.5 AND max_lon > 2276721.5
AND min_lat < 5591241.0 AND max_lat > 5589947.0
```

(a) извршавање упита са просторним индексом

SQL 1

```
1   SELECT w.way_id FROM way_nodes w
2   JOIN nodes n ON w.node_id = n.node_id
3   JOIN way_tags t ON t.way_id = w.way_id
4   GROUP BY w.way_id
5   HAVING min(n.lon) < 2277337.5 AND max(n.lon) > 2276721.5
6   AND min(n.lat) < 5591241.0 AND max(n.lat) > 5589947.0
```

```
Execution finished without errors.
Result: 2920 rows returned in 1436ms
At line 1:
SELECT w.way_id FROM way_nodes w
JOIN nodes n ON w.node_id = n.node_id
JOIN way_tags t ON t.way_id = w.way_id
GROUP BY w.way_id
HAVING min(n.lon) < 2277337.5 AND max(n.lon) > 2276721.5
AND min(n.lat) < 5591241.0 AND max(n.lat) > 5589947.0
```

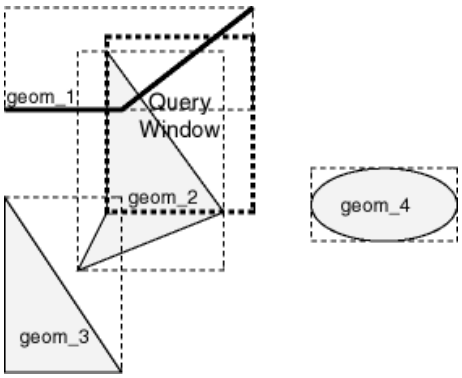
(b) извршавање упита без просторног индекса

Слика 3.7: Поређење времена извршавања упита за проналазак свих геопросторних елемената унутар неког граничног оквира.

Интервалски упити

Под *интервалским упитима* (енг. range query) се подразумева проналазак свих елемената у неком правоугаонику претраге или унутар неке геометријске фигуре као што је полигон. Можемо, на пример, пронаћи све школе у неком граду или локације свих саобраћајних незгода на неком путу.

Интервалски упит се реализује на следећи начин: за задату фигуру претраге, рекурзивно се пролази кроз R стабло почев од корена и испитује да ли се та фигура сече са минималним граничним оквиром текућег чвора стабла. Ако је то случај, испитују се и деца тог чвора, а ако није врши се одсецање претраге у том чвору. Поступак се понавља све до листова стабла, где се врши провера да ли има преклапања геопросторног елемента са фигуром претраге, и ако преклапање постоји тај елемент се додаје у скуп резултата претраге. Сложеност оваквог поступка је у најгорем случају $O(n)$, где је n број геопросторних елемената, али је у пракси сложеност мања због великог броја одсецања претраге. На слици 3.8 илустрован је пример интервалског упита.



Слика 3.8: Интервалски упит, где се у правоуганику претраге налазе објекти `geom_1` и `geom_2`. Минималан гранични оквир објекта `geom_3` има пресек са правоугаоником претраге али сама геометрија нема пресек па се зато она не налази у скупу резултата, док минимални гранични оквир објекта `geom_4` нема пресека са правоугаоником претраге, па ни сама геометрија нема пресек.

Упити по блискости

Упити по блискости (енг. nearness query) представљају проналазак k најближих суседа за неку задату тачку. Тако, на пример, можемо да одредимо

најближе хотеле од неке конференцијске сале, или нама најближу аутобуску станицу.

Имплементација алгоритма проналаска k најближих суседа, када су подаци смештени у R стаблу, се своди на додавање чворова у иницијално празан хип, тако да на врху хипа буде чвор чији је минимални гранични оквир најближи траженој тачки. Раздаљина између тачке и минималног граничног оквира има вредност 0 ако је тачка у оквиру а ако је тачка ван граничног оквира онда је једнака растојању тачке до најближе ивице оквира. На почетку се у празан хип додаје корен стабла и у сваком кораку алгоритма се избацује чвор са врха хипа и у хип се убацују сва његова деца. Овако дефинисан хип има својство да се на његовом врху увек налази најближи минимални гранични оквир из хипа. Због тога, када се по први пут на врху хипа нађе лист R стабла, тј. тачка која представља суседа, он ће представљати први најближи елемент задатој тачки. Ако је потребно пронаћи више од једног најближег суседа поступак се понавља све док се k пута на врху хипа не нађе лист R стабла.

Сложеност алгоритма у најгорем случају је $O((\frac{n}{m-1} + k) \cdot \log(\frac{n}{m-1} + k))$, где је n број геопросторних елемената, а m је максимални број деце сваког чвора у R стаблу. Међутим, у пракси скоро никада не долази до најгорег случаја, јер је за њега потребно обићи све чворове који нису листови пре него што се дође до првих k листова што се обично не дешава са добро уређеним R стаблом. Пример резултата упита по блискости на примеру тражења k најближих аеродрома је дат на на слици 3.9.

Просторно придруживање

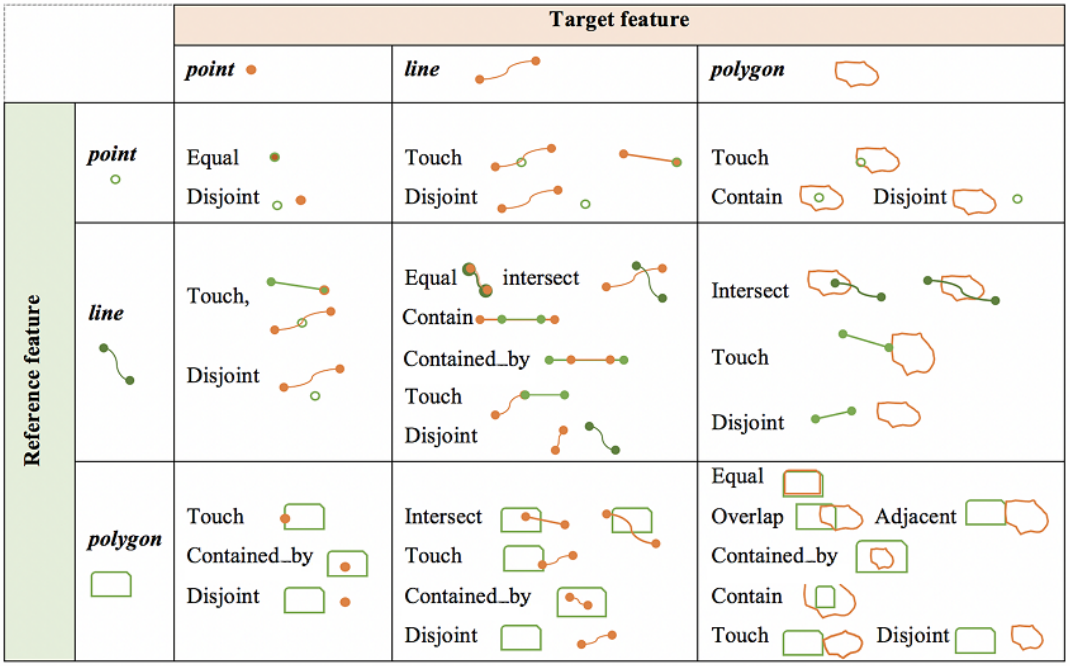
Просторна релација (енг. spatial relation) представља тополошки однос између два објекта у простору, и она зависи од врсте њихових геометрија (тачка, линија или полигон). Неке од просторних релација су једнакост, пресек, подскуп, дисјунктност и додиривање. Слика 3.10 приказује детаљније тополошке односе за дате геометрије два елемента.

Утврђивање односа два елемента у R стаблу се обично своди на проверу односа њихових минималних граничних оквира, како би се процес провере убрзао, па се, тек уколико има потребе, пореде и сами геометријски елементи.

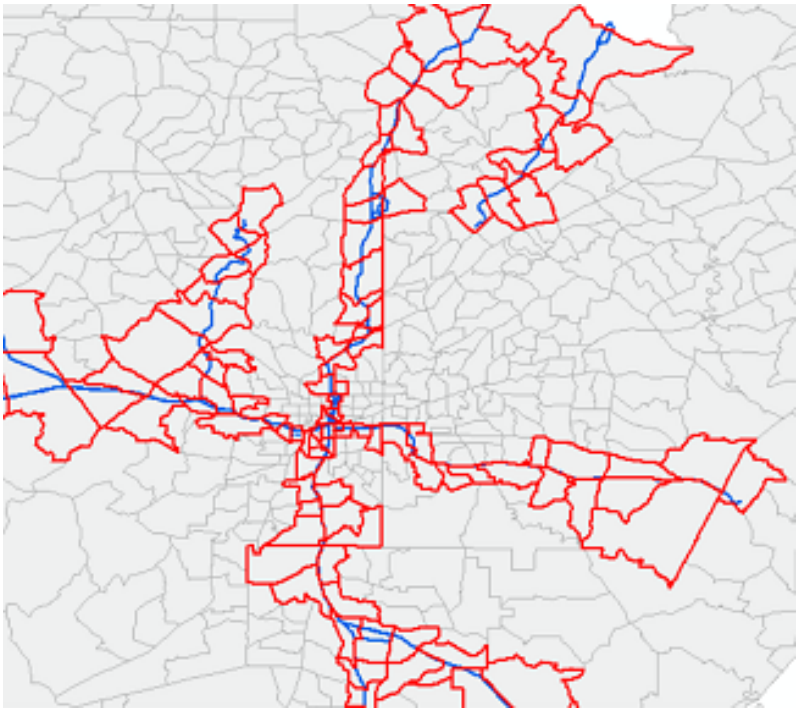
Просторно придруживање (енг. spatial join) је операција којом се две табеле придружују на основу њихових геометријских података. Претпоставимо

Слика 3.9: Пет најближих аеродрома од дате локације авиона.

да имамо табелу која садржи информације о линијама метроа и другу табелу која садржи списак градских општина. Ове две табеле можемо просторно да придружимо тако што ћемо издвојити општине у граду кроз које пролази нека од линија метроа (INTERSECT). Овај пример илустрован је на слици 3.11.



Слика 3.10: Детаљан преглед просторних релација у зависности од типа геометријских објеката. Преузето са https://gistbok.ucgis.org/bok-topics/spatial-queries.



Слика 3.11: Резултат просторног упита којим се издвајају општине кроз које пролази нека линија метроа. Плавом бојом обележене су линије метроа, а црвеном бојом уоквирене општине кроз које пролази нека од линија метроа.

Глава 4

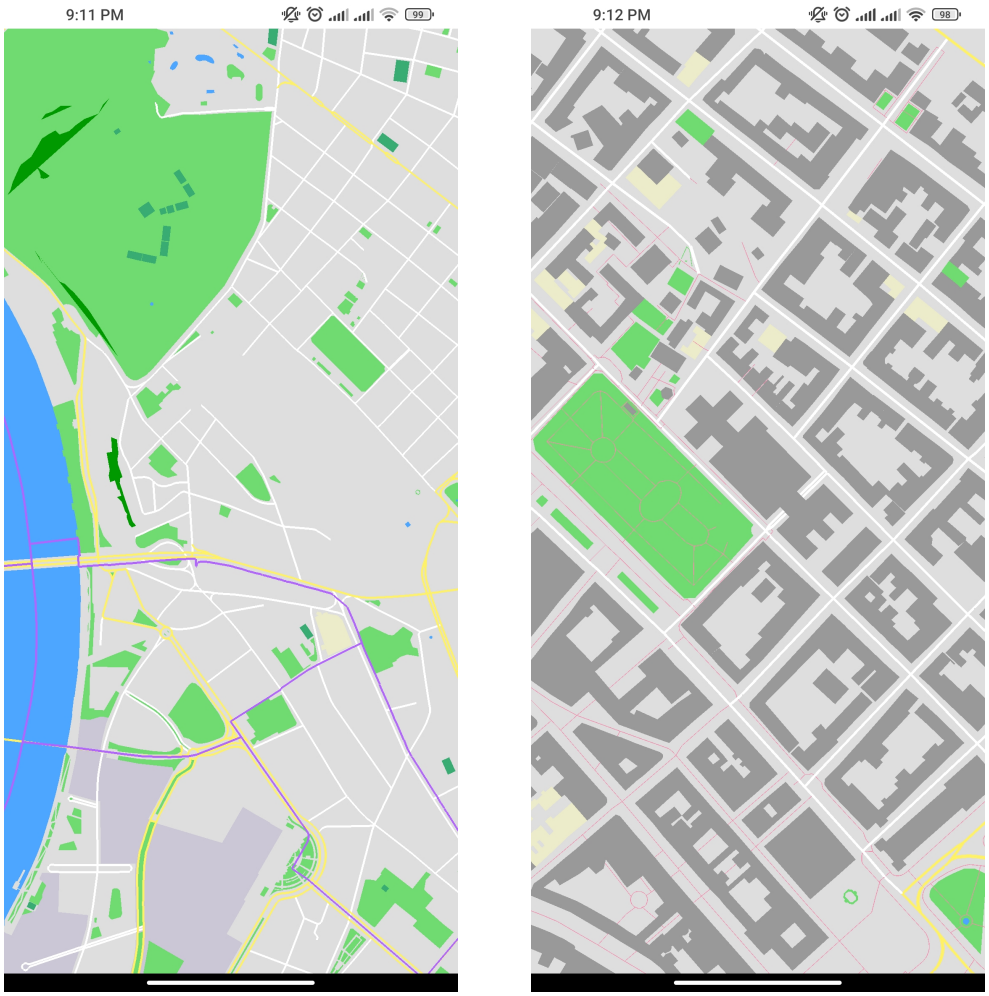
Приказ рада апликације за Андроид

У овом поглављу биће речи о исцртавању геопросторних података на екран уређаја и биће дат приказ реализације одговарајуће апликације за Андроид.

Апликација *OSMRenderer* приказује мапу на основу података које је корисник преузео са адресе https://planet.openstreetmap.org. Изворни код апликације доступан је на адреси: https://github.com/dpns98/OSMRenderer. Приказивање мапе се остварује исцртавањем геопросторних података који су видљиви са тренутне позиције камере. Корисник може да помера мапу додиривањем екрана уређаја. На тај начин он може да помера камеру и да приближи, односно удаљи приказ. На слици 4.1 дата су два различита приказа апликације, у двема различитим размерама.

Корисник, такође, има могућност да претражује објекте на мапи. Одабиром броја k и врсте објеката којег жели да пронађе, на мапи ће се обележити k објекта те врсте који су најближи централној тачки екрана. Ова функционалност илустрована је на слици 4.2.

У поглављу 2 је дефинисан модел података над којима ради апликација. Модел је добијен претпроцесирањем података преузетих са пројекта *OpenStreetMap* и он се чува у самој апликацији као база података. База података има проширење за P стабло, како би се убрзали просторни упити, илустровани у поглављу 3. За издвајање података који ће бити приказани на екрану уређаја биће коришћен интервалски упит који за дату позицију камере, која гледа на мапу, дохвата све геопросторне податке из базе који улазе у тренутни оквир екрана уређаја. Ти елементи се потом исцртавају на



(a) приказ са већом размером (b) приказ са мањом размером

Слика 4.1: Приказ дела Београда коришћењем апликације *OSMRenderer* у две различите размере.

екран користећи *OpenGL ES* [4] графички програмски интерфејс апликације (API). Сваким следећим померањем камере и приближавањем или удаљавањем приказа мапе, поново се извршава интервалски упит и добијени подаци се исцртавају на мапи.

Слика 4.2: Означавање три најближе школе за дату позицију која одговара централној тачки екрана.

4.1 Програмски језик Котлин

Котлин је строго типизиран вишеплатформски програмски језик опште намене. Њега је дизајнирала компанија *JetBrains*, тако да буде потпуно интероперабилан са програмским језиком Јава. То значи да се у апликацији која је писана у Котлину могу користити библиотеке и класе писане у Јави. 2019. године Гугл је објавио да је Котлин постао преферирани програмски језик за развој Андроид апликација. Главна предност програмског језика Котлин у односу на програмски језик Јава је у једноставнијој синтакси, што омогућава програмерима да за мање времена напишу више кода.

Промењиве

Котлин користи две кључне речи за декларисање променљивих: *val* за променљиве чија се вредност не мења и *var* за променљиве чија се вредност може променити. Променљиве се декларишу и дефинишу на следећи начин:

```
var count: Int = 10
```

После назива променљиве се може додати и њен тип, али то није обавезно ако се приликом декларације одмах и дефинише њена вредност, јер ће се на основу додељене вредности аутоматски препознати тип променљиве. На пример, наредбом:

```
val languageName = "Kotlin"
```

променљивој `languageName` се додељује тип `String`, јер је вредност `''Kotlin''` типа `String`.

Променљиве у програмском језику Котлин подразумевано не могу да имају вредност *null*. Да би се омогућило променљивој да има вредност *null*, потребно је приликом декларације да се поред њеног типа дода ?.

```
var languageName: String? = null
```

Функције

Функције се у програмском језику Котлин декларишу кључном речју `fun`. Приликом декларације потребно је навести назив функције, њене параметре, као и тип повратне вредности. У телу функције је потребно навести наредбе функције као и наредбу `return` којом се означава повратна вредност функције.

```
fun isNumberBiggerThan100(number: Int): Boolean {
    return if (number > 100) {
        true
    } else {
        false
    }
}
```

Котлин такође подржава и анонимне функције. Иако су оне анонимне, можемо сачувати њихову референцу и даље их користити у програму:

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}

val stringLength: Int = stringLengthFunc("Android")
```

Оне се могу користити и као параметри других функција:

```
fun stringMapper(str: String, mapper: (String) -> Int): Int {
    return mapper(str)
}

stringMapper("Android", { input ->
    input.length
})
```

Уколико је анонимна функција последњи параметар функције, функција се може позвати и на следећи начин:

```
stringMapper("Android") { input ->
    input.length
}
```

4.2 OpenGL ES

OpenGL ES (OpenGL for Embedded Systems)[1] је вишеплатформски подскуп *OpenGL API*-ја за исцртавање дводимензионалне и тродимензионалне графике, који је обично хардверски убрзан коришћењем графичке картице. Дизајниран је за уграђене уређаје као што су паметни телефони, таблети и конзоле за видео игрице. *OpenGL ES* је историјски гледано најчешће коришћен тродимензионални графички интерфејс.

Оперативни систем Андроид има нативну подршку за *OpenGL ES* [2]. У Андроид окружењу за креирање и руковање графиком коришћењем про-

[1]Званична документација се може пронаћи на адреси: https://registry.khronos.org/OpenGL-Refpages/es2.0.

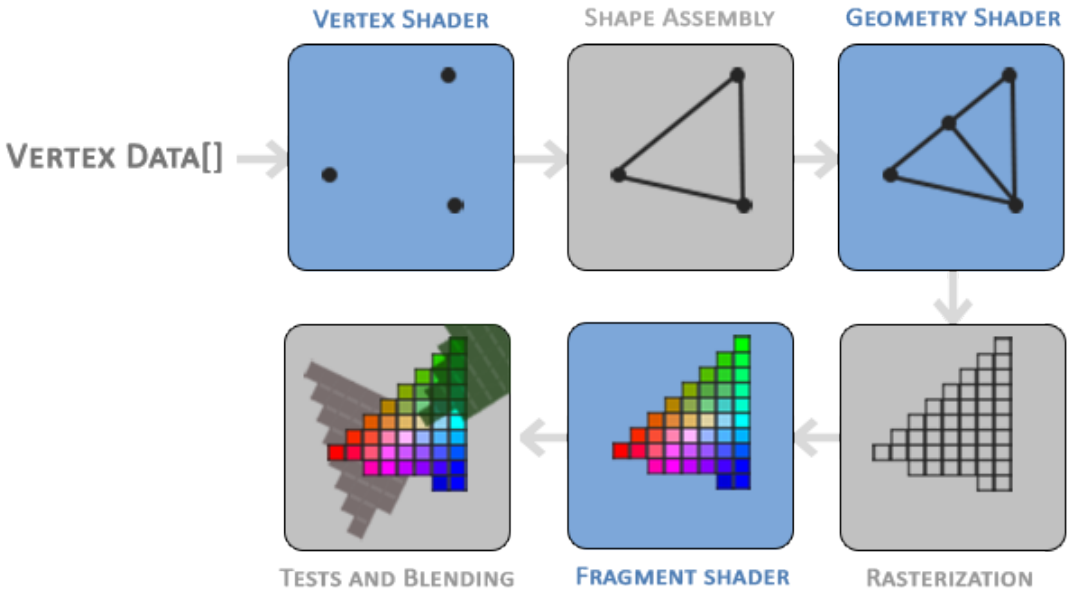
грамског интерфејса *OpenGL ES* користе се класа `GLSurfaceView` и интерфејс `GLSurfaceView.Renderer`.

- `GLSurfaceView` је класа која наслеђује класу `View` помоћу које се исцртавају сви елементи корисничког интерфејса и графика на екран Андроид уређаја. Она обезбеђује посебну нит за *OpenGL* функције, како се главна нит не би преоптеретила. Помоћу интерфејса `GLSurfaceView.Renderer` исцртавају се графички елементи. Класа `GLSurfaceView` може, такође, да имплементира метод `onTouchEvent` како би се ослушкивали догађаји додира екрана.

- `GLSurfaceView.Renderer` дефинише методе потребне за исцртавање графике у `GLSurfaceView`. Ти методи су:

  - `onSurfaceCreated`: Систем позива овај метод само једном приликом креирања објекта класе `GLSurfaceView`. Користи се да би извршио радње које се морају десити само једном, као што је подешавање параметара *OpenGL* окружења или иницијализација *OpenGL* графичких објеката.
  - `onDrawFrame`: Систем позива овај метод приликом исцртавања сваког фрејма. Користи се за цртање графичких објеката.
  - `onSurfaceChanged`: Систем позива овај метод приликом промене величине или оријентације екрана уређаја. На пример, систем позива овај метод када се уређај промени из усправне (енг. portrait) у положену (енг. landscape) оријентацију.

Са иницијализованим објектом класе `GLSurfaceView`, добија се површина на којој могу да се исцртавају графички елементи. Процес трансформације координата графичких елемената у обојене пикселе екрана се зове *графичка проточна обрада* (енг. graphics pipeline) [6]. Графичка проточна обрада се може поделити на неколико мањих фаза, где свака фаза као улаз очекује излаз претходне фазе. Свака фаза се може извршавати паралелно на графичкој картици помоћу програма који се зове *шејдер* (енг. shader). Шејдер је програм који дефинише како се исцртава сваки пиксел. У зависности од фазе он може да рачуна на којим пикселима се налазе објекти који се исцртавају или којом се бојом боји сваки пиксел. На слици 4.3 су приказане фазе графичке проточне обраде.



Слика 4.3: Илустрација фаза графичке проточне обраде. Преузето са https://learnopengl.com/Getting-started/Hello-Triangle.

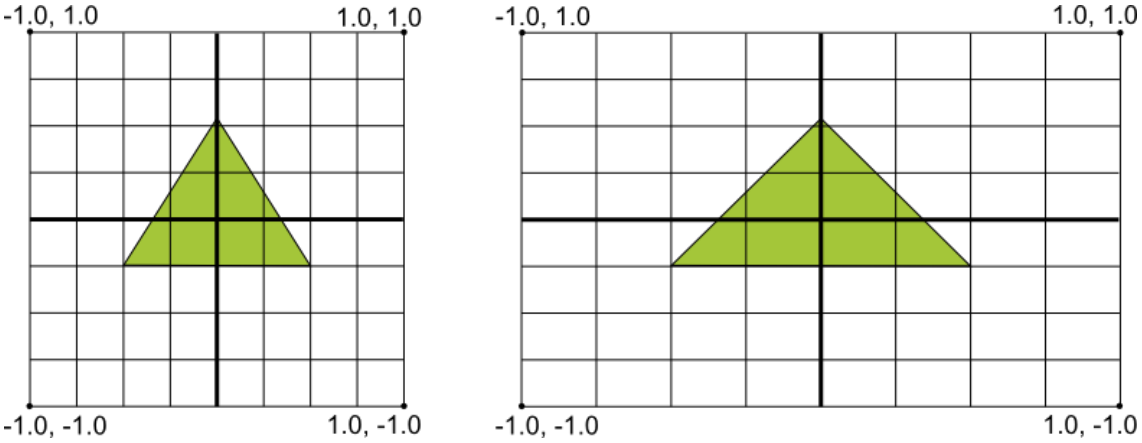
На улазу у графичку проточну обраду имамо податке о координатама облика који треба да се исцрта које се прослеђују *вертекс шејдеру* (енг. vertex shader). Његова улога је да те координате трансформише у пикселе помоћу матрица пројекције и погледа камере, о којима ће бити више речи у поглављу 4.3. У зависности од одабране *OpenGL* примитиве (троугао, линија, ...), формира облик спајањем координата. Потом се врши растеризација добијеног облика на пикселе. *Фрагмент шејдер* (енг. fragment shader) сваком пикселу додељује боју, на основу улазних параметара и осталих *OpenGL* ефеката. На крају се врши провера прозирности, и у зависности од улазне алфа вредности, која одређује транспарентност објекта, добија се коначна боја пиксела.

OpenGL ES захтева да се дефинишу барем вертекс и фрагмент шејдер. Они се програмирају у *OpenGL Shading Language* (GLSL) језику.

4.3 Пројекција

Један од основних проблема у приказивању графике на Андроид уређајима је тај што екрани уређаја могу да варирају по величини и облику. *OpenGL* претпоставља да је сваки екран квадратног облика, што наравно често није

случај. У ситуацији када екран уређаја није квадратног облика, облици се на таквом екрану исцртавају развучено.



Слика 4.4: *OpenGL* пресликавање координата. Преузето са https://developer.android.com/develop/ui/views/graphics/opengl/about-opengl.

На слици 4.4 лево приказан је униформни координатни систем, код кога се вредности по `x` и `y` оси крећу од -1 до 1 и како се ове координате заправо пресликавају на типичан екран уређаја у положеној оријентацији (десно). Ово пресликавање се може извести применом *OpenGL* матрица пројекције и погледа чиме се постиже да графички објекти имају исправне пропорције на произвољном екрану.

Сваки објекат има придружене координате у односу на светски координатни систем (енг. world space). У случају апликације *OSMRenderer* координатни почетак је пресек Екватора и нултог меридијана. Координате објекта у координатном систему са координатним почетком на месту положаја камере могу се добити помоћу матрице погледа (енг. view matrix). Она трансформише светске координате у координате погледа. Матрица пројекције (енг. projection matrix) трансформише координате погледа у униформне координате тако да се графички објекти могу приказати на екрану Андроид уређаја.

Када се креирају ове матрице, потребно је матрицу композиције матрице погледа и матрице пројекције, која је означена са `uVPMatrix`, проследити у одговарајуће фазе графичке проточне обраде. Прво је потребно проследити матрицу у вертекс шејдер.

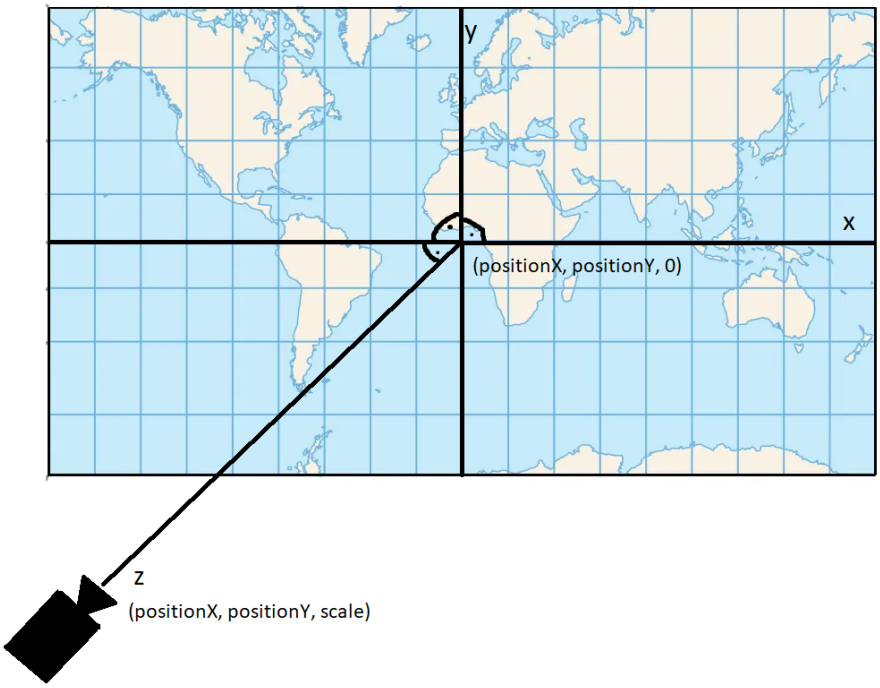
```
uniform mat4 uVPMatrix;
```

```
attribute vec4 vPosition;
void main(){
    gl_Position = uVPMatrix * vPosition;
}
```

Након прослеђивања матрице у шејдер, може се приступити промењивој матрице како би се она применила на објекте који се исцртавају.

```
GLES20.glGetUniformLocation(program, "uMVPMatrix")
```

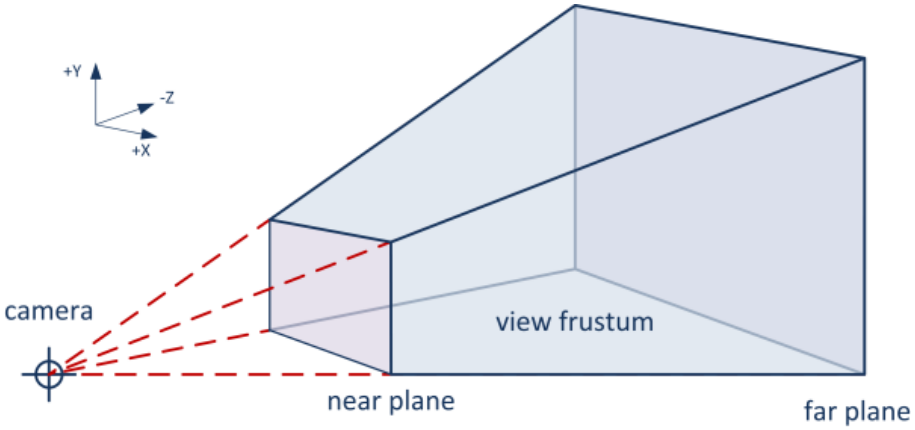
Да би се креирала матрица погледа потребно је одредити где се камера налази у светском координатном систему, ка којој тачки је усмерена и под којим углом. Пошто корисник има могућност да мења положај камере, потребно је увести промењиве `positionX` и `positionY` које представљају на којој x и y координати се налази камера, док промењива `scale` представља са које висине камера гледа, односно z координату положаја камере. z координата геопросторних података је 0, због тога ће камера увек бити усмерена ка тачки $(positionX, positionY, 0)$ под правим углом, тако да је y координата окренута ка горе. Односно камера са висине гледа доле ка мапи, којој y оса представља географску ширину, а x оса географску дужину, као на слици 4.5.



Слика 4.5: Позиција камере.

```
Matrix.setLookAtM(
    viewMatrix, 0,
    positionX, positionY, scale,
    positionX, positionY, 0f,
    0f, 1.0f, 0.0f
)
```

Матрицом пројекције пресликавамо мапу на екран уређаја. Који су објекти видљиви са тренутне позиције камере задаје се коришћењем *OpenGL фрустума* (енг. frustum). Фрустум је зарубљена четворострана пирамида, која је одређена помоћу 6 равни: ближе равни (енг. near plane), даље равни (енг. far plane), леве, десне, горње и доње равни (слика 4.6). Сви објекти који се налазе унутар фрустума или имају пресек са њим биће исцртани на екрану, тако да се исцртавају само делови објекта који се налазе унутар фрустума. Свака тачка тих објеката ће се пресликати у пресечну тачку ближе равни и зрака одређеног позицијом камере и тачком објекта.



Слика 4.6: OpenGL фрустум.

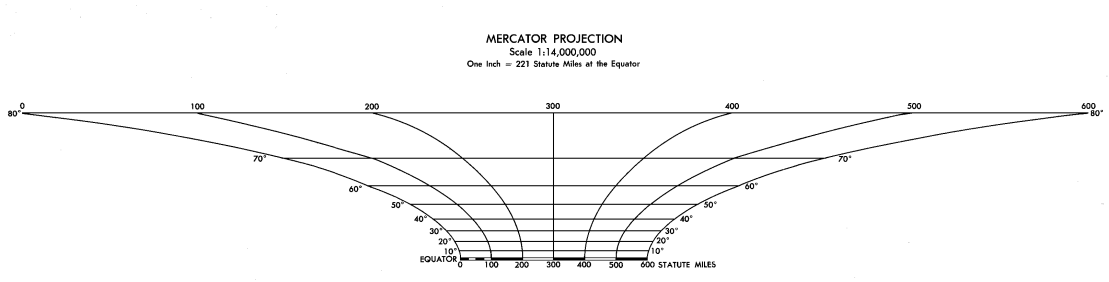
Ако бисмо фрустум проширили до позиције камере (црвени испрекидани зраци на слици 4.6), добили бисмо пирамиду. Фрустум можемо посматрати као разлику две сличне пирамиде које имају исти врх, где раван подножја мање пирамиде представља ближу раван фрустума и налази се на раздаљини 1 од врха пирамиде, док је раван подножја веће пирамиде паралелна ближој равни фрустума и налази се на раздаљини *scale* од врха пирамиде. Ако би се сви дводимензионални геопросторни подаци налазили у равни подножја веће пирамиде, онда би они били *scale* пута мањи када би се пресликали на ближу раван, тј. променљива *scale* би била размер карте. Дакле, претпоставићемо

да је ближа раван фрустума на растојању 1 од камере, да се геопросторни подаци налазе на раздаљини *scale* од камере а да је даља раван довољно велики број како се не би одсецали подаци са мапе.

Потребно је још дефинисати горњу, доњу, леву и десну раван фрустума. Како су геопросторни подаци записани у Меркаторовој пројекцији, раздаљину у метрима између две тачке на мапи које се налазе на истој географској ширини можемо да израчунамо по формули:

$$k \cdot \sqrt{(A.x - B.x)^2 + (A.y - B.y)^2} \tag{4.1}$$

где су A и B тачке на мапи а k дисторзија пројекције. *Дисторзија пројекције* (енг. scale factor) представља фактор грешке раздаљине који се јавља приликом пројектовања сфере на ваљак. Она је једнака косинусу географске ширине тачака за које меримо раздаљину. Тако, на пример, ако меримо раздаљину „дуж Екватора" дисторзија ће бити једнака 1, тј. неће бити грешке јер је кружница која представља екватор на сфери пресликана у кружницу на ваљку истог полупречника. Уколико меримо раздаљину између тачака чија је географска ширина једнака 45 степени, дисторзија је једнака 0.7, односно две тачке на тој паралели које се разликују за 1 метар у Меркаторовој пројекцији у стварности су удаљене за 70 центиметра. Грешка коришћења ове формуле за тачке које нису на истој географској ширини, али су близу, које се обрађују у овом раду је занемарљива.



Слика 4.7: Илустрација дисторзије у Меркаторовој пројекцији

Са дефинисаним раздаљинама у Меркаторовој пројекцији можемо да одредимо и ширину и дужину екрана уређаја у пројекцији, како би добили горњу, доњу, леву и десну раван фрустума. Ширина и дужина екрана се израчунавају на следећи начин: најпре се број пиксела по дужини и ширини уређаја подели густином пиксела по инчу, чиме се добија величина уређаја у инчима;

добијене вредности се конвертују у метре и помноже дисторзијом добијеном на основу географске ширине тачке која представља центар мапе. Коначно, добијене вредности делимо са 2 како бисмо добили половину дужине и ширине уређаја, чиме коначно добијамо димензије фрустума, а самим тим и матрицу пројекције.

```
Matrix.frustumM(
    projectionMatrix, 0,
    -halfScreenWidth, halfScreenWidth,
    -halfScreenHeight, halfScreenHeight,
    1f, 20000f
)
```

Како би се одредили пиксели у које се пресликавају тачке са мапе потребно је у сваком фрејму помножити матрицу погледа камере и матрицу пројекције и њихов производ проследити вертекс шејдеру. Композиција матрице пројекције и матрице погледа камере се даље прослеђује графичким објектима које *OpenGL* исцртава.

```
override fun onDrawFrame(gl: GL10) {
    ...
    Matrix.multiplyMM(vPMatrix, 0, projMatrix, 0, vMatrix, 0)
    GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, vPMatrix, 0)
    ...
}
```

4.4 Померање мапе

Корисник управља апликацијом на два начина: додиром екрана једним прстом може да помера мапу дуж x и y координатне осе, а додиром екрана са два прста и њиховим померањем може да промени размер карте, односно да помера камеру дуж z координатне осе.

Приликом померања мапе померањем прста по екрану очекује се да тачка на мапи коју је корисник додирнуо остане испод прста. Да би се тај ефекат постигао потребно је запамтити растојање које је прст прешао на екрану и прерачунати то растојање у Меркаторовој пројекцији како бисмо знали за колико пиксела треба да померимо мапу. За рачунање растојања у Меркато-

ровој пројекцији биће коришћена променљива `pixels2meters` чија се вредност добија на основу наредне формуле:

$$pixels2meters = \frac{0.0254}{k \cdot d} \tag{4.2}$$

где је 0.0254 број метра у инчу, d густина пиксела уређаја по инчу, а k фактор дисторзије. Множењем променљиве `pixels2meters` растојањем које је прст прешао на екрану израженим у пикселима, добија се растојање у Меркаторовој пројекцији у случају када је размера карте једнак 1:1. У ситуацији када размер мапе није једнак 1, претходну вредност је потребно помножити тренутним размером. Сад имамо формулу којом се одређује нова позиција центра екрана:

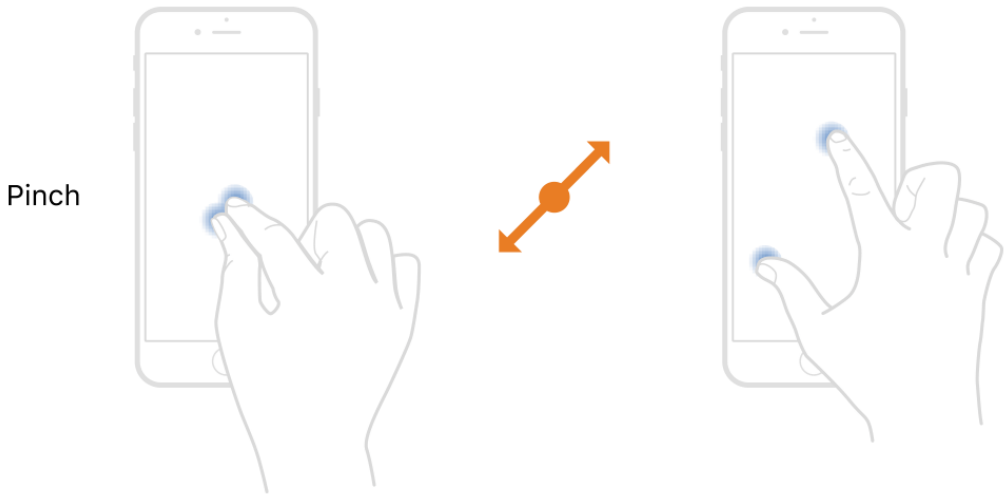
```
renderer.positionX += (prevX - x) * renderer.scale * pixels2meters
renderer.positionY += (y - prevY) * renderer.scale * pixels2meters
```

где су $(prevX, prevY)$ координате тачке екрана које је прст додирнуо, (x, y) координате тачке екрана у тренутку када је прст престао да додирује екран, а *scale* је размер карте.

Што се тиче промене размера карте, корисник очекује да се трансформација скалирања врши око тачке која се налази између два прста на екрану. Да би се то постигло, потребно је извршити и скалирање и транслацију. Потребно је, такође, одредити и да ли се врши увећавање или удаљавање приказа. То се постиже рачунањем количника растојања између два прста у претходном фрејму и у тренутном фрејму: ако је вредност количника већа од 1 прсти се приближавају један другом па је потребно повећати размер карте, а ако је вредност количника мања од 1 прсти се удаљавају један од другог и потребно је смањити размер карте (слика 4.8). Померај за који се врши транслација се добија рачунањем растојања између централне тачке екрана тј. тачке ка којој камера тренутно гледа и средишње тачке дужи одређене тачкама у којима два прста додирују екран. Како желимо да раздаљина на екрану између те две тачке остане иста, потребно је израчунати растојање у Меркаторовој пројекцији на старој размери мапе, као и на новој размери и померај камере на новој размери добити одузимањем те две вредности.

```
val d1 = sqrt((prevX-prevX1) * (prevX-prevX1) +
         (prevY-prevY1) * (prevY-prevY1))
```

```
val d2 = sqrt((x-x1) * (x-x1) + (y-y1) * (y-y1))
val ratio = d1/d2

val xOffset = (metrics.widthPixels/2 - (x+x1)/2) *
              pixels2meters * renderer.scale*(1-ratio)
val yOffset = (metrics.heightPixels/2 - (y+y1)/2) *
              pixels2meters * renderer.scale*(1-ratio)

if (renderer.scale * ratio > minScale && renderer.scale * ratio < maxScale) {
    renderer.scale *= ratio
    renderer.positionX -= xOffset
    renderer.positionY += yOffset
}
```



Слика 4.8: Удаљавањем два прста смањује се размер карте

4.5 Упити над базом

Приликом сваког померања мапе потребно је одредити који се геопросторни елементи налазе у новом опсегу екрана, како бисмо могли да их исцртамо. Најпре је потребно одредити опсег екрана у Меркаторовој пројекцији. Њега одређујемо рачунањем минималне и максималне x координате и минималне и максималне y координате опсега екрана. Те координате се могу добити одузимањем и сабирањем половине ширине и половине дужине екрана, израженим у метрима помноженим дисторзијом, са тачком на коју гледа камера.

```
val extent = Extent(
    renderer.positionX - (halfScreenWidth*renderer.scale),
    renderer.positionX + (halfScreenWidth*renderer.scale),
    renderer.positionY - (halfScreenHeight*renderer.scale),
    renderer.positionY + (halfScreenHeight*renderer.scale)
)
```

Као што је било описано у поглављу 2, база података садржи четири просторне табеле: табеле са путањама на мањој и на већој размери као и табеле са везама на мањој и на већој размери. Разлог раздвајања табела по размерама је то што на већим размерама не исцртавамо све податке, као што су појединачне зграде и мање улице, јер се они не би добро видели. Додатно, пошто табела на већим размерама садржи мањи број података, упит проналажења свих геопросторних елемената из неког опсега се брже извршава. Дакле, у зависности од тренутне размере, геопросторни подаци које треба да исцртамо на екран се проналазе у одговарајућој табели. Размера 1:8000 је граница за одређивање табеле над којом ће се вршити упит.

Размотримо упит којим се проналазе све путање у задатом опсегу:

```
SELECT r.way_id, lon, lat, key, value FROM rtree_way$scale r
JOIN way_nodes w ON r.way_id = w.way_id
JOIN nodes n ON w.node_id = n.node_id
JOIN way_tags t ON t.way_id = w.way_id
WHERE min_lon < ${extent.maxX} AND max_lon > ${extent.minX}
AND min_lat < ${extent.maxY} AND max_lat > ${extent.minY}
```

Улазни параметри упита су `scale` који може имати вредност 1 и 2 у зависности од тренутне размере и `extent` који представља тренутни опсег. Упитом се издвајају тачке путања које се налазе у траженом опсегу. Сваки ред у резултату упита садржи идентификатор и ознаку путање којој тачка припада као и координате те тачке. Низ координата тачака путање је потребан како би се путања могла исцртати, док је ознака путање значајна због боје којом се путања исцртава.

Проласком кроз све редове из резултата упита у променљивој `currentId` памти се идентификатор текуће путање, а у низ `coords` се додају координате тачке која припада тој путањи. Ако се приликом преласка на следећи ред резултата промени идентификатор путање, закључујемо да смо прешли на

нову путању, па се координате и ознака текуће путање додају у низ `arrays`, а нова путања постаје текући путања.

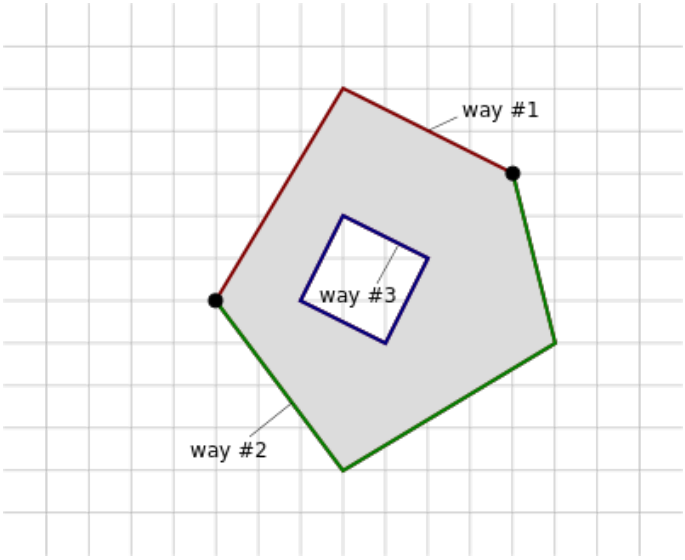
```
while (cursor.moveToNext()) {
    val id = cursor.getInt(0)
    val lon = cursor.getFloat(1)
    val lat = cursor.getFloat(2)
    var tag = cursor.getString(3)
    val value = cursor.getString(4)

    if (id != currentId || cursor.isLast) {
        arrays.add(Triple(coords.toFloatArray(), currentTag, null))
        currentId = id
        currentTag = tag
        coords.clear()
    }
    coords.add(lon)
    coords.add(lat)
}
```

У низу `arrays` се памте све информације потребне за исцртавање геопросторних елемената. Низ `arrays` садржи уређене тројке вредности: први члан је низ координата путање, други члан је њена ознака, а трећи садржи индексе у низу координата од којих почињу координате рупа у полигону. У случају путања он је `null` јер путање немају рупе у себи, док то није случај са везама. Подсетимо се да везе увек представљају полигоне са рупама (слика 4.9).

Упит којим се проналазе све везе у задатом опсегу има следећи облик:

```
SELECT r.relation_id, lon, lat, key, value, role, w.way_id
FROM rtree_relation$scale r
JOIN relation_members m ON m.relation_id = r.relation_id
JOIN way_nodes w ON m.ref = w.way_id
JOIN nodes n ON w.node_id = n.node_id
JOIN relation_tags t ON t.relation_id = r.relation_id
WHERE min_lon < ${extent.maxX} AND max_lon > ${extent.minX}
AND min_lat < ${extent.maxY} AND max_lat > ${extent.minY}
```

Упит је сличан одговарајућем упиту којим се издвајају путање у тренутном опсегу. Сваки ред у резултату упита садржи информације о тачкама везе које се налазе у тренутном опсегу: координате тачке, идентификатор путање и везе којима тачка припада, као и ознаку и улогу везе. Везе се састоје



Слика 4.9: Полигон са рупом где путање 1 и 2 одређују спољашњи део полигона, а путања 3 одређује унутрашњи део полигона

од листе путања које се називају чланови. Сваки члан је путање и има улогу унутрашњег или спољашњег полигона. Сви чланови заједно чине један полигон са рупама.

Да би се од путања саставио полигон, потребно је итерирати кроз све путање везе и надовезати координате сваке наредне путање на текућу путању која чува листу координата обиђених путања. Свака наредна путања у вези се надовезује на текућу путању тако што се упоређују први и последњи пар координата путање са првим и последњим паром координата текуће путање, па се путања у зависности од тога који су им парови координата исти надовезује на почетак или крај текуће путање, и то у једном од два могућа поретка. Када је први пар координата једнак последњем пару координата у текућој путањи онда смо добили полигон и њега чувамо у листи полигона са његовом улогом, а текућу путању празнимо.

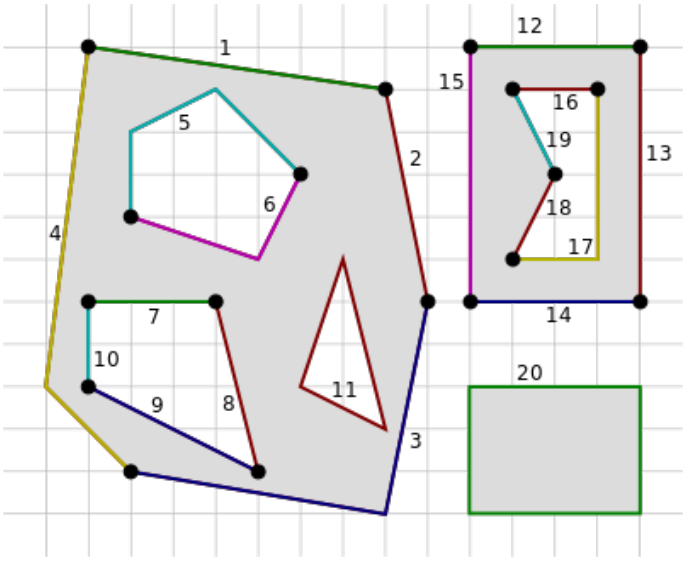
```
//za svaki put coordsWay koji je predstavljen listom parova koordinata
//vrši se njegovo nadovezivanje na tekući put currentCoordsWay
//koji predstavlja put dobijen spajanjem do sada obiđenih puteva
if (way != currentWay || cursor.isLast) {
    //ako je trenutni put prazan dodaj sve elemente niza coordsWay
    //u niz currentCoordsWay
    if (currentCoordsWay.isEmpty())
        currentCoordsWay.addAll(coordsWay)
```

```
        //ako je poslednji par koordinata trenutnog puta jednak prvom paru
        //koordinata puta coordsWay, coordsWay se nadovezuje u normalnom
        //poretku na kraj trenutnog puta
        else if (currentCoordsWay.last() == coordsWay.first())
            currentCoordsWay.addAll(
                coordsWay.subList(1, coordsWay.size)
            )
        //ako je poslednji par koordinata trenutnog puta jednak poslednjem
        //paru koordinata puta coordsWay, coordsWay se nadovezuje u obrnutom
        //poretku na kraj trenutnog puta
        else if (currentCoordsWay.last() == coordsWay.last())
            currentCoordsWay.addAll(
                coordsWay.reversed().subList(1, coordsWay.size)
            )
        //ako je prvi par koordinata trenutnog puta jednak prvom paru
        //koordinata puta coordsWay, coordsWay se nadovezuje u obrnutom
        //poretku na početak trenutnog puta
        else if (currentCoordsWay.first() == coordsWay.first())
            currentCoordsWay.addAll(
                0, coordsWay.reversed().subList(0, coordsWay.lastIndex)
            )
        //ako je prvi par koordinata trenutnog puta jednak poslednjem paru
        //koordinata puta coordsWay, coordsWay se nadovezuje u normalnom
        //poretku na početak trenutnog puta
        else if (currentCoordsWay.first() == coordsWay.last())
            currentCoordsWay.addAll(
                0, coordsWay.subList(0, coordsWay.lastIndex)
            )
        //provera da li je trenutni put zatvoren, tj. da li on poligon
        if (currentCoordsWay.first() == currentCoordsWay.last()){
            //dodavanje poligona u odgovarajući niz
            //u zavisnosti od njegove uloge
            if (currentRole == "inner")
                inner.add(currentCoordsWay.toList())
            else
                outer.add(currentCoordsWay.toList())
            currentCoordsWay.clear()
        }
        currentWay = way
        currentRole = role
        coordsWay.clear()
    }
```

Обиласком свих путања везе формирају се две листе полигона: полигони са улогом спољашњег полигона и полигони са улогом унутрашњег полигона. За креирану листу полигона потребно је за сваки спољашњи пронаћи његове унутрашње полигоне, да би се добио полигон са рупама којег веза представља (слика 4.10). То се може постићи тако што што за сваки унутрашњи полигон извршимо проверу да ли се у потпуности налази у неком спољашњем полигону. Иако овакав алгоритам није најефикаснији, у овом раду нема потребе за бољим алгоритмом јер је број полигона мали па би време извршавања било занемарљиво побољшано.

```
//za svaki spoljašnji poligon se pronalazi njemu odgovarajući
//unutrašnji poligon
if (id != currentId || cursor.isLast){
    outer.forEach { out ->
        //pronalazak minimalnog graničnog okvira spoljašnjeg poligona
        val maxX = out.map { it.first }.max()
        val maxY = out.map { it.second }.max()
        val minX = out.map { it.first }.min()
        val minY = out.map { it.second }.min()
        //niz koji označava indekse od kojih počinju
        //unutrašnji poligoni
        val holes = mutableListOf<Int>()
        inner.forEach { inn ->
            //provera da li se minimalni granični okvir unutrašnjeg
            //poligona nalazi u minimalnom graničnom okviru
            //spoljašnjeg poligona
            if (maxX >= inn.map { it.first }.max() &&
                maxY >= inn.map { it.second }.max() &&
                minX <= inn.map { it.first }.min() &&
                minY <= inn.map { it.second }.min()
            ) {
                //pamtimo kolika je dužina niza out, kako bi znali od
                //kog indeksa počinje unutrašnji poligon
                holes.add(out.size)
                //dodajemo sve elemente niza inn u niz out
                out.addAll(inn)
            }
        }

        //dodavanje novog elementa u niz arrays sa podacima o
```

```
            //poligonu sa rupama
            arrays.add(
                Triple(
                    //transformacija podataka koji su tipa
                    //Array<Pair<Float>> u tip Array<Float>
                    out.flatMap { e ->
                        listOf(e.first, e.second)
                    }.toFloatArray(),
                    currentTag,
                    if (holes.isEmpty()) null else holes.toIntArray()
                )
            )
        }
        inner.clear()
        outer.clear()
        currentCoordsWay.clear()
        currentId = id
        currentTag = tag
}
```



Слика 4.10: Једна веза може да буде представљена и већим бројем полигона, где сваки има различит број рупа. Пример би била нека шума кроз коју пролази аутопут, па је она представљена са више полигона.

4.6 Исцртавање геопросторних података

Сваки пут кад се изврши упит за проналажење геопросторних података у тренутном оквиру екрана потребно је добијене податке и исцртати. Апликација подржава исцртавање отворених и затворених путања, тј. линија и полигона. Најпре је потребно дефинисати вертекс шејдер којим се задају координате објеката које треба исцртати и фрагмент шејдер који садржи информације о томе којом бојом се боји објекат.

Вертекс шејдер одређује позицију сваке тачке на екрану множењем њене позиције у светском координатном систему у Меркаторовој пројекцији `vPosition` композицијом матрице погледа камере и матрице пројекције `uVPMatrix`:

```
uniform mat4 uVPMatrix;
attribute vec4 vPosition;
void main(){
    gl_Position = uVPMatrix * vPosition;
}
```

Фрагмент шејдер боји сваки пиксел објекта прослеђеном бојом `vColor` за тај објекат:

```
uniform vec4 vColor;
void main() {
  gl_FragColor = vColor;
}
```

За исцртавање изломљених линија потребне су координате тачака дуж линије и њена ознака како би се обојила одговарајућом бојом. Како је *OSMRenderer* апликација написана у Котлину, а *OpenGL* је имплементиран у програмском језику C, потребно је извршити конверзију података. За запис координата се користи *ByteBuffer* класа, која нам омогућава да податке чувамо у истом поретку како се чувају у нативном систему, а да се потом помоћу вертекс бафер објекта (енг. Vertex Buffer Object, скраћено VBO) проследе меморији на графичкој картици. Вертекс бафер објекат је низ који садржи податке о координатама тачака као и друге атрибуте као што је боја објекта, и прослеђује их меморији графичке картице, како би се брже исцртали.

```
//kopiranje koordinata tačaka linije u ByteBuffer u nativnom poretku
var vertexBuffer: FloatBuffer? =
```

```
    ByteBuffer.allocateDirect(coords.size * FLOAT_SIZE).run {
        order(ByteOrder.nativeOrder())
        asFloatBuffer().apply {
            put(coords)
            position(0)
        }
    }
//generisanje VBO i kopiranje podataka u niz buffer
//iz promenljive vertexBuffer
GLES20.glGenBuffers(1, buffer, 0)
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, buffer[0])
GLES20.glBufferData(
    GLES20.GL_ARRAY_BUFFER,
    vertexBuffer!!.capacity() * FLOAT_SIZE,
    vertexBuffer,
    GLES20.GL_STATIC_DRAW
)
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0)
```

Потребно је још навести на који начин *OpenGL* треба да интерпретира податке из бафера, поставити боју за бојење и помножити координате објекта композицијом матрице пројекције и матрице погледа. Линије се исцртавају коришћењем *OpenGL* примитиве `GL_LINE_STRIP`. Примитива `GL_LINE_STRIP` одговара изломљеној линији састављеној од дужи чије су крајње тачке добијене из вертекс бафер објекта.

```
//dodeljivanje vrednosti za promenljive iz šejdera
//vColor = color
colorHandle = GLES20.glGetUniformLocation(program, "vColor")
GLES20.glUniform4fv(colorHandle, 1, color, 0)
//uVPMatrix = vpMatrix
vPMatrixHandle = GLES20.glGetUniformLocation(program, "uVPMatrix")
GLES20.glUniformMatrix4fv(vPMatrixHandle, 1, false, vpMatrix, 0)
//vPosition dobija vrednosti iz niza buffer
positionHandle = GLES20.glGetAttribLocation(program, "vPosition")
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, buffer[0])
GLES20.glVertexAttribPointer(positionHandle,2,GLES20.GL_FLOAT,false,0,0)
GLES20.glEnableVertexAttribArray(positionHandle)
//crtanje izlomljene linije
GLES20.glDrawArrays(GLES20.GL_LINE_STRIP, 0, coords.size/2)
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0)
```

За исцртавање полигона користи се примитива `GL_TRIANGLES`, која на основу тачака из вертекс бафер објекта формира низ троуглова: први је одређен теменима са индексима 0, 1 и 2, други теменима са индексима 3, 4, 5 итд. Пошто примитива `GL_TRIANGLES` исцртава троуглове, потребно је извршити триангулацију полигона. За то је коришћена функција *earcut* из библиотеке `earcut4j`[2], која за дату листу координата спољашњих и унутрашњих темена полигона враћа листу индекса координата која представља редослед у којем је потребно цртати троуглове да би се нацртао одговарајући полигон. За разлику од изломљених линија, код полигона можемо да уштедимо меморију тако што ће се користити индексни бафер објекат (енг. Element Buffer Object, скраћено EBO). Наиме, уместо да се у меморију графичке картице шаљу све координате темена полигона у редоследу исцртавања као бројеви у покретном зарезу (типа float) који заузимају 4 бајта, може се у вертекс бафер објекту свако од темена полигона записати тачно једном, а у индексном бафер објекту само индекси темена у редоследу за исцртавање као кратки целобројни типови (типа short) који заузимају 1 бајт.

```
//dobijanje poretka temena trouglova prilikom triangulacije poligona
drawOrder = Triangulation.earcut(coords, holes, 2)
//kopiranje indeksa temena poligona u ByteBuffer u nativnom poretku
var drawListBuffer: ShortBuffer? =
ByteBuffer.allocateDirect(drawOrder.size * 2).run {
    order(ByteOrder.nativeOrder())
    asShortBuffer().apply {
        put(drawOrder)
        position(0)
    }
}
//generisanje EBO i kopiranje indeksa u niz ebo
//iz promenljive drawListBuffer
GLES20.glGenBuffers(1, ebo, 0)
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, ebo[0])
GLES20.glBufferData(
    GLES20.GL_ELEMENT_ARRAY_BUFFER,
    drawListBuffer!!.capacity()*2,
    drawListBuffer,
    GLES20.GL_STATIC_DRAW
)
```

[2]Документација се може пронаћи на адреси: https://github.com/earcut4j/earcut4j.

```
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, 0)
```

У сваком позиву **onDrawFrame** методе када се помери мапа потребно је исцртати геопросторне податке, али је пре тога потребно испразнити бафере како не би дошло до прекорачења меморије.

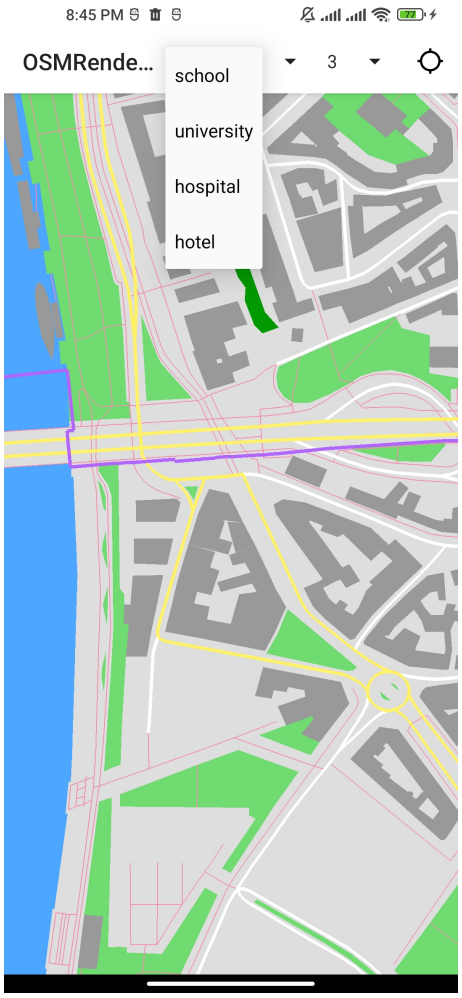
```
//brisanje bafera
geometries.forEach {
    it.release()
}
//kreiranje geometrija iz upita
geometries = createGeometries()
//iscrtavanje geometrija
geometries.forEach{
    it.draw(vPMatrix, mProgram)
}
```

4.7 k најближих суседа

Апликација *OSMRenderer* има могућност одређивања k најближих објеката неког типа датој тачки претраге. Корисник има могућност да изабере број k од 1 до 5 и врсту објекта коју претражује. Објекти могу бити: школа, факултет, болница или хотел. Кликом на дугме за претрагу обележиће се k објеката изабраног типа који су најближи централној тачки екрана. Одабир типа објекта од интереса илустрован је на слици 4.11.

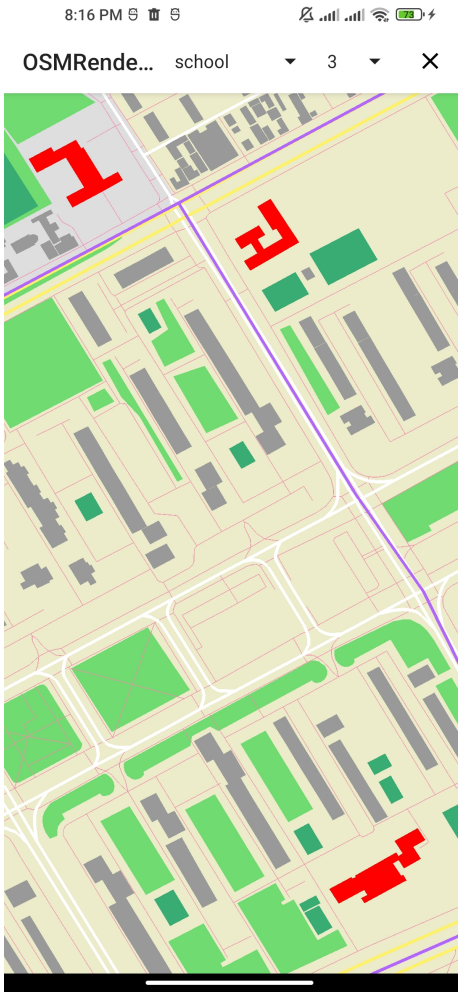
Када се кликне на дугме за претрагу у позадини се извршава упит који проналази дате објекте. Сваки ред у резултату упита представља тачку путање која обележава пронађен објекат. Ред садржи идентификатор путање као и координате тачке.

```
SELECT way_id, lon, lat
FROM way_nodes w JOIN nodes n ON w.node_id = n.node_id JOIN (
SELECT r.way_id AS id,
min(((max_lon+min_lon)/2-$lon)*((max_lon+min_lon)/2-$lon)
((max_lat+min_lat)/2-$lat)*((max_lat+min_lat)/2-$lat)) AS distance
FROM rtree_way1 r JOIN way_tags t ON t.way_id = r.way_id
WHERE value = '$value'
GROUP BY r.way_id
ORDER BY distance LIMIT $k) knn ON knn.id = way_id
```



Слика 4.11: Одабир врсте објекта за претраживање

Даље се за сваку путању посебно њен скуп координата додаје у низ `arrays`, на сличан начин као код упита за одређивање геопросторних елемената у оквиру екрана. Добијени објекти се црвеном бојом исцртавају на екрану. На слици 4.12 је приказан резултат претраге 3 најближе школе од централне тачке екрана.



Слика 4.12: Проналазак 3 најближе школе од централне тачке екрана

Глава 5

Закључак

У овом раду представљена је апликација за Андроид *OSMRenderer* за исцртавање геопросторних података. Апликација се може користити за преглед мањих географских површина као што су градови. Она, такође, ради и без приступа интернету јер се сви подаци чувају у локалној меморији. Апликација подржава и задавање упита по блискости, којим се налазе k најближих суседа за неку тачку са мапе. Од просторних упита су имплементирани проналазак свих геопросторних елемената унутар неког правоугаоника претраге као и одређивање k најближих објеката истог типа од неке задате тачке.

У оквиру рада изложене су основне идеје пројекта *OpenStreetMap* чији су подаци коришћени у развоју апликације као и модел података који он користи. Укратко је описан систем за управљање базама података , који је коришћен у развоју апликације. Приказан је процес трансформације података из формата XML у базу података. Потом је објашњен принцип рада просторних база података и просторних упита. Приказана је и структура података P стабло, која је коришћена као просторни индекс. На крају рада приказани су имплементациони детаљи апликације, као и основе развоја апликација помоћу *OpenGL ES* интерфејса у Андроид окружењу.

OSMRenderer је рађен по угледу на остале картографске апликације за Андроид, од којих је најпознатија апликација *Google Maps*. Наравно, *OSMRenderer* је доста мањег обима, али су имплементиране све основне карактеристике картографских апликација: приказивање геопросторних података, померање мапе и коришћење просторних база података.

По узору на остале апликације, *OSMRenderer* би могао да се унапредити додавањем лабела, тј. текстуалних назива геопросторних објеката на мапи,

као што су називи улица, школа, хотела и других објеката. Проблем додавања лабела је прилично сложен и он обухвата проналазак највећег броја геопросторних елемената, таквих да се њихове лабеле могу исцртати на мапи а да се не преклапају међусобно. Могло би се, такође, убрзати извршавање упита над базом, тако што би се један просторни упит над целим екраном поделио на више мањих паралелних упита, тако да унија простора који обухвата сваки од тих мањих упита буде једнака простору који обухвата полазни већи упит, тј цео екран. Тиме би се побољшале перформансе апликације. Такође, апликација би се могла проширити могућношћу извршавања још неких просторних упита, попут утврђивања просторног односа између два просторна податка.

Библиографија

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM.*, 18:509–517, 1975.

[2] Kevin Brothaler. *OpenGL ES 2 for Android: A Quick-Start Guide (Pragmatic Programmers)*. Pragmatic Bookshelf, 2013.

[3] OpenStreetMap Foundation. OpenStreetMap, 2006. online at: `https://wiki.openstreetmap.org/`.

[4] Khronos Group. OpenGl ES Overview. *3D Graphics API State of the Union: SIGGRAPH 2015*, 2011.

[5] Antonin Guttman. R trees: A dynamic index structure for spatial searching. *Sigmod Record*, 14:47–57, 1984.

[6] Frahaan Hussain. *Learn OpenGL*. Packt Publishing, 2018.

[7] Jay A. Kreibich. *Using SQLite*. O'Reilly Media, 2010.

[8] Gustavo Niemeyer. Geohash, 2008. online at: `https://web.archive.org/web/20080305223755/http://blog.labix.org/#post-85`.

[9] osmzoso. osm2sqlite, 2022. online at: `https://github.com/dpns98/OSMRenderer`.

[10] Dušan Petrović. OSMRenderer, 2022. online at: `https://github.com/dpns98/OSMRenderer`.

[11] Philippe Rigaux. *Spatial Databases: With Application to GIS*. Morgan Kaufmann, 2001.

[12] John P. Snyder. Map Projections – A Working Manual. U.S. Geological Survey Professional Paper 1395. *United States Government Printing Office*, page 38, 1987.