Detektiranje i mapiranje reljefa površine kolnika

Milić, Matej

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij računarstva

DETEKTIRANJE I MAPIRANJE RELJEFA POVRŠINE KOLNIKA

Diplomski rad

Matej Milić

Osijek, 2020.

SADRŽAJ

1 UVOD 1

2 PREGLED PODRUČJA TEME RADA 2

3 KORIŠTENE TEHNOLOGIJE 5

3.1 Android platforma 5

3.1.1 Arhitektura Android platforme 6

3.1.2 Senzori 6

3.1.3 Android Studio 7

3.2 Kotlin programski jezik 8

3.3 Firebase 9

3.4 Koin dependency injection 10

4 RAZVOJ APLIKACIJE ZA DETEKTIRANJE I MAPIRANJE RELJEFA POVRŠINE KOLNIKA 11

4.1 Korisnički zahtjevi 11

4.2 Izrada modela korisničkog sučelja 12

4.3 Arhitektualni dizajn sustava 14

4.4 Životni krugovi korisnika i očitanih podataka 15

4.5 Dodavanje ovisnosti 17

4.6 Izrada korisničkog sučelja 18

4.7 Prijava i registracija korisnika 24

4.8 Spremanje i rad sa podacima 27

5 RAZVOJ ALGORITMA ZA DETEKCIJU I KLASIFIKACIJU NEPRAVILNOSTI 30

5.1 Postavljanje akcelerometra i lokacijske usluge 30

5.2 Izrada i implementacija algoritma 33

5.3 Mapiranje podataka 36

6 KORIŠTENJE I TESTIRANJE APLIKACIJE 38

7 ZAKLJUČAK 43

LITERATURA 44

SAŽETAK 46

ABSTRACT 47

ŽIVOTOPIS 48

1 UVOD

Održavanje kvalitete prometnih površina jedan je od bitnijih aspekata cestovnog prometa. Postoje razne tehnike i načini izrade prometnica, no zbog raznih vremenskih nepogoda te velikog broja vozila koje iste moraju trpjeti nakon određenog vremena dolazi do stvaranja oštećenja i rupa. Kako bi se smanjio rizik od prometnih nesreća potrebno je pronaći i popraviti oštećenja što može biti veoma dugotrajan i skup proces. U svrhu olakšanja pronalaska raznih udubina i rupa na cestama osmišljena je mobilna aplikacija koja bi korištenjem senzora detektirala nepravilnosti i bilježila ih na karti. Postoje razne varijable koje treba uzeti u obzir, kao što su veličina nepravilnosti na cesti i hrapavost podloge. Cilj rada je istražiti i utvrditi najefikasniji način detektiranja nepravilnosti, spremanje i rad sa dobivenim podacima te stvaranje reljefne karte koja će zorno prikazati lokaciju i klasu detektirane nepravilnosti.

Većina današnjih mobilnih uređaja dolazi sa ugrađenim akcelerometrom koji omogućuje detektiranje pomicanja uređaja po x i y osi. Aplikacija će koristiti tu mogućnost da prilikom vožnje detektira razna poskakivanja automobila kako bi algoritam mogao utvrditi o kakvoj se nepravilnosti radi te označiti istu na karti. Korisnici aplikacije imati će mogućnost:

- Prijave i registracije
- Pokretanje detektiranja i mapiranja
- Prikaz trenutne akceleracije i očitanih podataka prilikom mjerenja
- Prikaz i upravljanje detektiranim podacima i vrijednostima
- Pregled i upravljanje označenim područjima na karti

U drugom poglavlju rada predstavljena su slična rješenja koja se bave sličnom tematikom. U trećem poglavlju opisane se korištene tehnologije. Navedene su osnove arhitekture Android platforme, te je opisan Kotlin programski jezik u kojemu je aplikacija izrađena. Osim toga opisana je Firebase platforma koja je korištena za spremanje podataka te ovjeravanje autentičnosti korisnika. U četvrtom poglavlju opisan je razvoj aplikacije te općenita arhitektura sustava. Osim toga opisana je izrada korisničkog sučelja, rad sa korisnicima i spremanje podataka. Unutar petog poglavlja opisan je rad sa senzorima, razvoj algoritma te mapiranje podataka. U šestome poglavlju prikazano je testiranje aplikacije te analiza dobivenih rezultata. Sedmo poglavlje zaključuje rad.

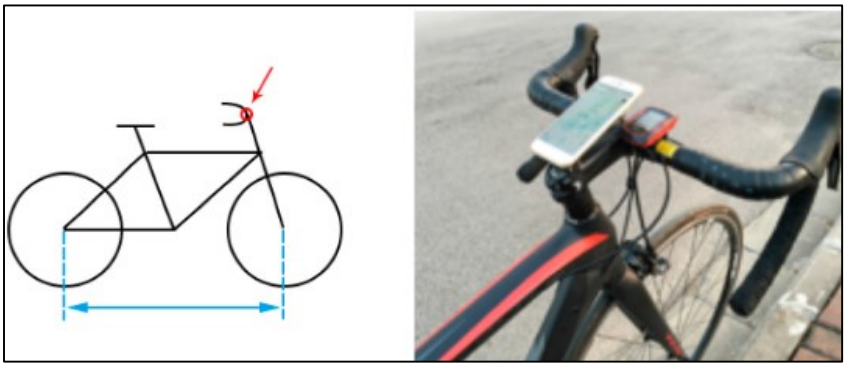
2 PREGLED PODRUČJA TEME RADA

Rupe i udubine na cesti velik su problem u cestovnom prometu i mogu dovesti do prometnih nesreća ukoliko se ne vodi briga o njima. Jedan od osnovnih problema je količina cestovnih prometnica koje treba kontrolirati što oduzima puno vremena, novaca i zahtjeva puno uloženih ljudskih sati. Potrebno je detektirati sva oštećenja što brže nakon njihovog stvaranja kako bi se mogli kloniti u što kraćem vremenskom roku. Kontroliranje cesta vožnjom i manualnim pregledom vrlo je sporo i nije dovoljno efikasno kako bi se osigurala što veća kvaliteta i sigurnost prometnica.

Uvođenjem uređaja koji bi mjerili oštećenja pregled prometnica bi se ubrzao, a koristeći Android uređaje osigurao bi se vrlo velik broj jeftinih mjernih uređaja. Osnovni problem ovog rada je izrada efikasne aplikacije koja bi mogla osigurati pristupačno i pouzdano mjerenje uvjeta na cesti koristeći različite tipove vozila u različitim uvjetima. Uz to uređaji bi se mogli postaviti na privatna vozila, taksije, autobuse ili ostale vrste javnog prijevoza.

Ova ideja postoji već duže vremena te je već izrađeno nekoliko aplikacija i znanstvenih radova na ovu temu. Aplikacije koje se bave ovom temom su prikazane u [1] i [2], dok su znanstveni radovi koji opisuju izradu, testiranje i opisani u [3], [4] i [5]. U svakom od istraživanja postignuti su dobri rezultati i vidljivo je da postoji veliki potencijal u izradi ovakve aplikacije.

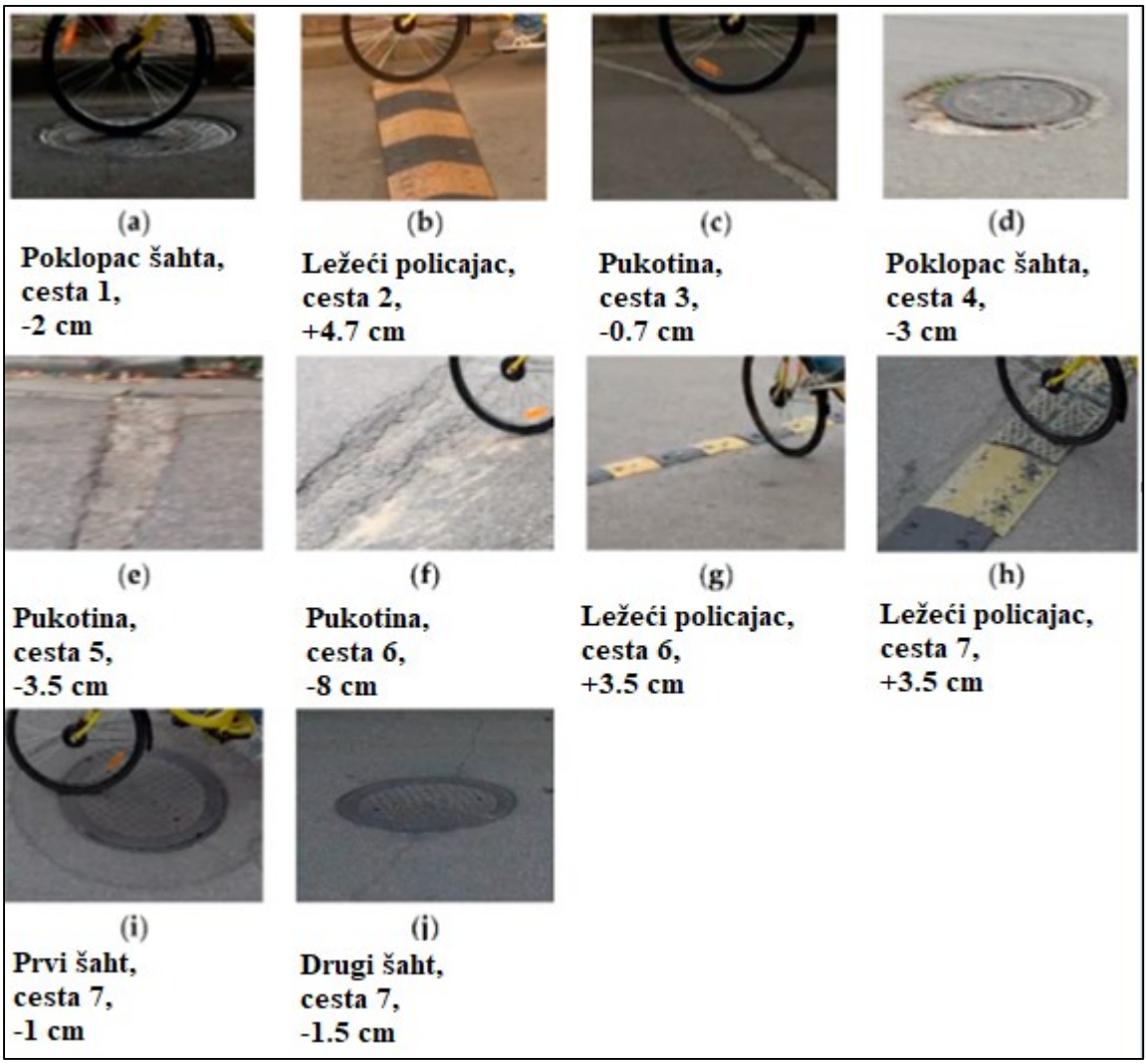
Izrada aplikacije za Android uređaj koji se postavlja na bicikl i prilikom vožnje mjeri podrhtavanja i različite tipove rupa, opisano je u [4]. Opisan je problem detekcije rupa, predstavljen je algoritam te je obavljeno testiranje aplikacije u stvarnim uvjetima. Postavljanje uređaja na bicikl prikazano je na slici 2.1.



Slika 2.1. – Postavljanje uređaja na bicikl [4]

Aplikacija pomoću algoritma određuje različite tipove nepravilnosti koje se nalaze na prometnicama. Tipovi nepravilnosti po kojima je aplikacija testirana vidljivi su na slici 2.2.

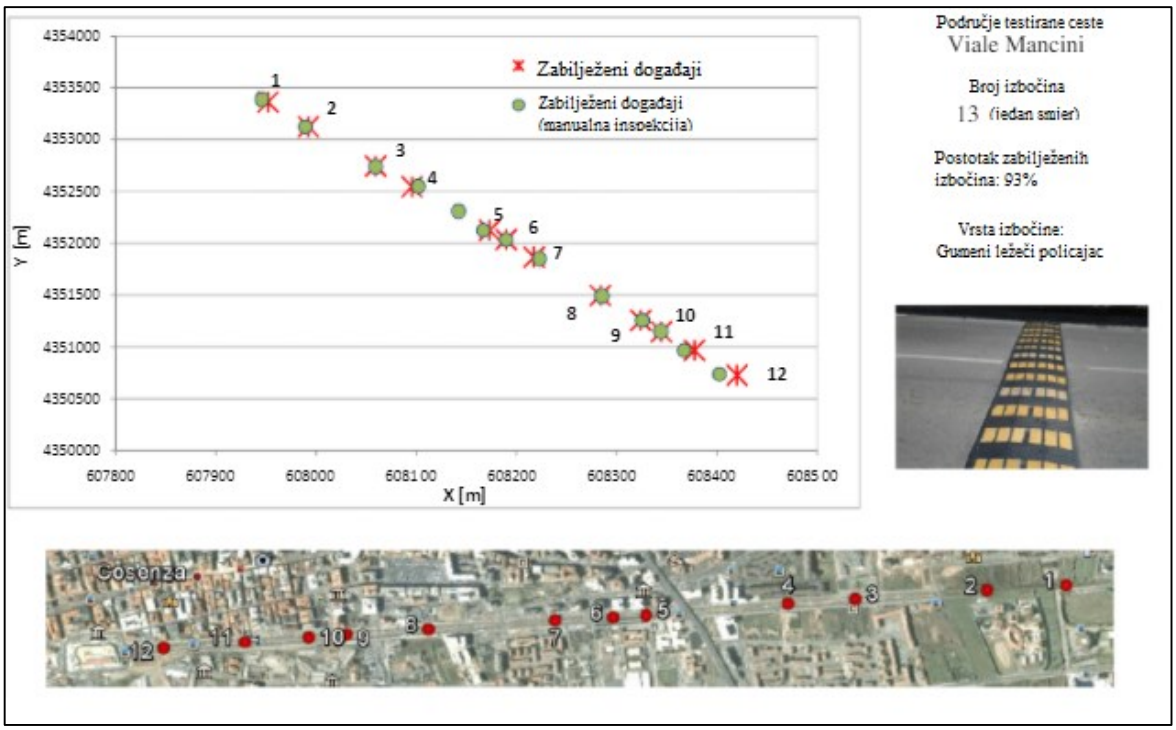
Vidljivo je da postoje različiti tipovi nepravilnosti na sedam različitih sekcija ceste na koje treba obraćati pozornost. Svaka nepravilnost ima udubljenje ili izbočinu izmjerenu u centimetrima.



Slika 2.2. – Tipovi udubina [4]

Zaključeno je da pametni uređaju postavljeni na bicikle mogu biti korišteni za mjerenje nepravilnosti na cesti te također identificiranje rupa i izbočina na cestama. Ovaj pristup posebno je koristan za one ceste koje nisu pristupačne za profesionalne instrumente (koji se često postavljaju na motorna vozila poput automobila), kao što su biciklističke staze i staze za hodanje. Rezultat rada je pokazao da vrijednosti očitane pomoću ovakve metode jako i pozitivno koreliraju sa podacima koji su očitani pomoću profesionalnih uređaja [4].

Mobilna aplikacija za testiranje kontrole kvalitete cestovnih površina, prikazana u [5] opisuje izradu aplikacije za detekciju ležećih policajaca i šahtova na cesti. Koristeći akcelerometar i GPS u mobilnim uređajima izrađena je aplikacija koja u stvarnom vremenu prikazuje lokaciju vozila i anomalija na cesti. Algoritam detektira ležeće policajce i šahtove analizom akceleracijskog signala pri događajima sa visokom energijom. Aplikacija je testirana na pet različitih uređaja koji su bili postavljani na tri različita mjesta u automobilu. U testiranju ceste postotak otkrivenih ležećih policajaca bio je oko 90%, dok je postotak detektiranih šahtova bio oko 65%. Primjer rezultata vidljiv je na slici 2.3. gdje su prikazani dobiveni rezultati za gumene ležeće policajce.

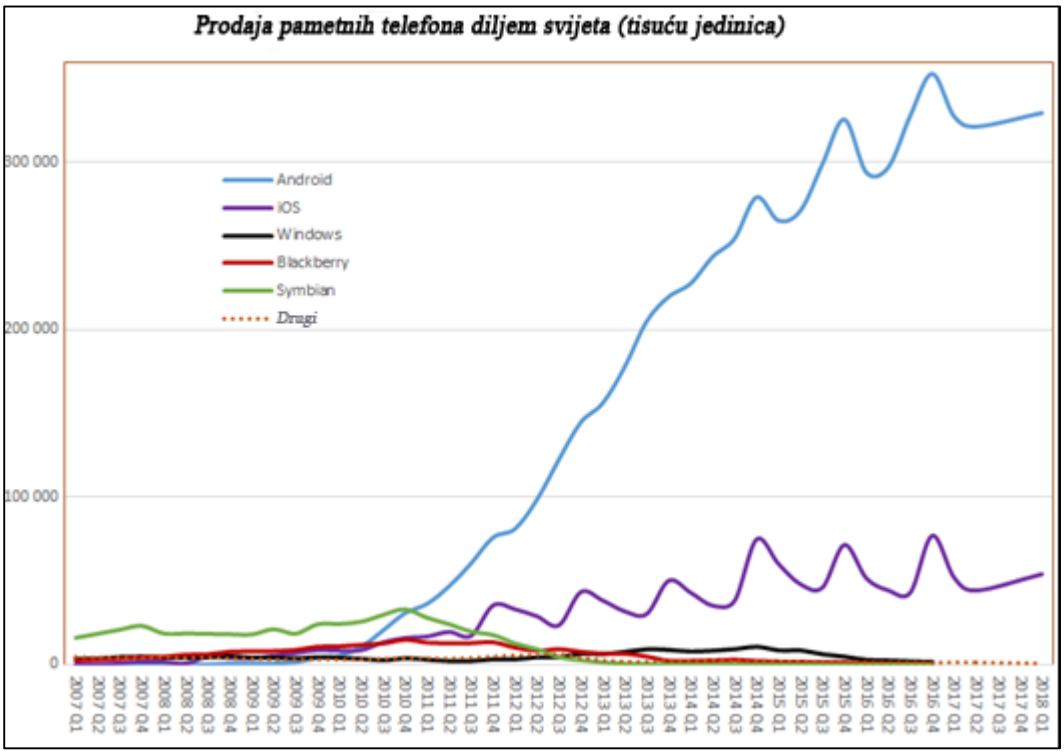


Slika 2.3. – Dobiveni rezultati za gumene ležeće policajce

3 KORIŠTENE TEHNOLOGIJE

3.1 Android platforma

Android je otvoreni operacijski sustav za mobilne uređaje, kao što su pametni telefoni i tablet računala, američke tvrtke *Google* temeljen na jezgri Linux i drugom softveru otvorenog koda. Nadalje, *Google* je razvio i Android TV za pametne televizore, Android Auto za automobile i Wear OS za pametne satove [6]. Činjenica da je Android operacijski sustav otvorenog koda omogućuje veliku raspodijeljenost na raznim uređajima pa je zbog toga Android najrašireniji operacijski sustav za pametne telefone što je vidljivo na slici 3.1.



Slika 3.1. – Prodaja pametnih telefona u svijetu [7]

Android je veoma opsežna platforma te je zbog svoje portabilnosti i otvorenosti savršena za korištenje na mobilnim uređajima. Za razvojne inženjere, Android omogućuje sve alate i okvire za razvoj mobilnih aplikacija brzo i efikasno. Android SDK je sve što je potrebno za početak razvoja na Androidu. Za korisnike, Android radi „ravno iz kutije“, a korisnici mogu bitno prilagoditi doživljaj svog mobilnog uređaja [8].

3.1.1 Arhitektura Android platforme

Android je građen na Linux operacijskom sustavu što mu daje razne prednosti, a osnovne su portabilnost i sigurnost. Osim toga Linux omogućuje veći nivo apstrakcije. Android operacijski sustav sastoji se od nekoliko slojeva, od kojih svaki ima različite karakteristike i svrhu. Podjela i osnovne karakteristike Android operacijskog sustava su:

- Linux kernel - temelj Android platforme, omogućuje proizvođačima razvijanje *driver-a* za *hardware*, korištenje sigurnosnih značajki Linux, upravljanje memorijom i nitima
- HAL (*engl. Hardware Abstraction Layer*) - definira standardno sučelje koje prodavači *hardware-a* implementiraju. Korištenje HAL-a omogućuje implementiranje funkcionalnosti bez mijenjanja ili utjecanja na sistem sa višeg nivoa [9]
- Android *runtime* - svaka aplikacija se pokreće kao samostalni proces sa vlastitom instancom Android *Runtime* (ART) [10]. Također sadrži osnovne biblioteke za pokretanje koje sadrže neke osnovne funkcionalnosti Java programskog jezika
- Nativne C++ biblioteke - pružaju nužne servise za aplikacijski sloj
- Java API okvir - sve mogućnosti i značajke Androida operacijskog sustava su dostupne kroz API-e pisane u Java programskom jeziku
- Sistemske aplikacije - aplikacije koje korisnici koriste. Mogu biti unaprijed instalirane ili se mogu instalirati kroz neki od Android marketa

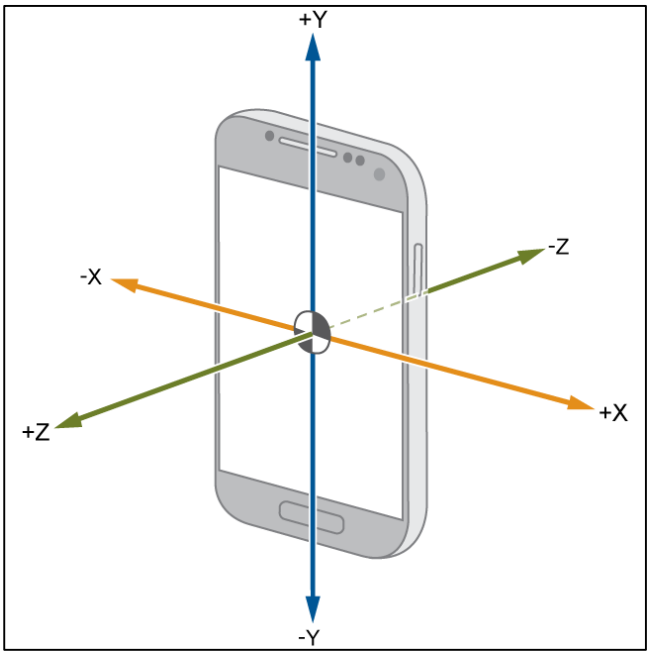
3.1.2 Senzori

Većina Android uređaja ima ugrađeno više vrsta senzora za određivanje položaja, orijentacije i raznih okolnih uvjeta. Moguće je dobiti podatke o trenutnoj lokaciji, brzini kretanja, ubrzanju promatrajući trodimenzionalne pokrete uređaja kao što su drmanje, zakrivljenost ili ljuljanje. Neke od tih mogućnosti biti će korištene pri određivanju udubina na cestama u ovom radu. Android platforma koristi tri kategorije senzora: [11]

- Senzori pokreta - ovi senzori mjere akceleracijske i rotacijske sile na sve tri osi koordinatnog sustava. Ova kategorija uključuje akcelerometar, senzore za gravitaciju, žiroskop i senzor rotacijskih vektora
- Senzori okoline - ovi senzori mjere razne parametre iz okoline uređaja kao što su temperatura okolnog zraka, pritisak, osvijetljenost i vlažnost. Ova kategorija uključuje barometar, fotometar i termometar

- Senzor pozicije - Android koristi različite metode dostavljanja informacije o lokaciji aplikaciji. U Androidu takve metode zovu se *location providers* i svaki ima unikatne karakteristike koje se mogu koristiti u različitim situacijama na različite načine. Primjeri dostavljača informacija su GPS i mrežni dostavljači [12].

Na slici 3.2. prikazan je akcelerometar.



Slika 3.2. – Prikaz akcelerometra na Android uređaju [13]

3.1.3 Android Studio

Android Studio je službeni IDE (*engl. Integrated Development Enviroment*) koji se koristi pri razvoju Android aplikacija. Dostupan je na Windows, MacOS i Linux operativnim sustavima, a zamijenio je Eclipse Android Development Tools kao primarno razvojno okruženje. Android studio omogućuje razne značajke poput: [14]

- Fleksibilni sustav gradnje zasnovan na Gradle-u
- Brzi emulator sa mnoštvom mogućnosti
- Jedinstveno okruženje za razvoj na svim Android uređajima
- Apliciranje svih promjena u kodu na aplikaciju koja je već pokrenuta, bez ponovnog pokretanja
- Integracija sa GitHub-om

- Opsežni okviri i alati za testiranje
- C++ i NDK podrška
- Ugrađena podrška za *Google* Cloud Platform, što omogućuje laku integraciju različitih *Google* servisa

Tri osnovna dijela programskog okruženja su navigacijski prozor koji prikazuje strukturu projekta, prozor sa alatima za upravljanje rada aplikacije (debugger, logiranje podataka, kontrola verzije, terminal) i prozor za pisanje i uređivanje koda. Osim toga moguć je prikaz izgleda aplikacije prilikom pisanja XML koda, te upravljanje raznim postavkama okruženja.

Struktura projekta u zadanom prikazu podijeljena je na dva dijela, a to su aplikacijski modul koji sadrži sav kod i resurse aplikacije te Gradle skripti. Aplikacijski modul sastoji se od slijedećih direktorija:

- *manifests* direktorij koji sadrži postavke i sadržaj aplikacije
- *java* direktorij koji sadrži kod i testove
- *res* direktorij koji sadrži XML kod te resurse poput slika, dimenzija, boja i ostalih vrijednosti koje se koriste unutar aplikacije

3.2 Kotlin programski jezik

Kotlin je programski jezik koji kombinira mogućnosti objektno orijentiranog i funkcionalnog programiranja. Kotlin je izrađen od strane JetBrains kompanije koja je htjela izraditi novi programski jezik za JVM (*Java Virtual Machine*). Kotlin primarno cilja na JVM, no može se kompajlirati kao JavaScript ili strojni kod [15]. Kotlin jezik prisutan je od 2011. godine, ali postao je popularan 2017. kada je *Google* objavio uvrštavanje Kotlina kao prvoklasnog jezika u Android razvoj [16].

Najveća razlika u odnosu na Javu je u tome što je moguće pisati funkcije izvan klasa, odnosno nije potrebna klasa kako bi se obavio neki zadatak. Osim toga osnovne mogućnosti Kotlin programskog jezika su:

- Zaključivanje tipa podataka (engl. *Type Inference*) - prilikom deklariranja nove varijable nije potrebno deklarirati tip podatka, nego će on biti pretpostavljen od strane programskog jezika
- Funkcije pridruživanja (engl. *Extension Function*) - funkcije koje se pišu izvan klase, a omogućuju „produživanje“ već napisane klase novom metodom

- Primarni i sekundarni konstruktor - primarni konstruktor služi za postavljanje osnovnih svojstava klase, dok se sekundarni može koristiti za postavljanje ostalih atributa
- Podatkovne klase (engl. *Data Class*) - klase koje služe samo za pohranu svojstava. Nemaju standardno tijelo klase i pripadajuće metode, a same po sebi imaju definirane *set* i *get* metode

Kotlin je napravljen da bude potpuno interoperabilan sa Javom i koristi standardne Java biblioteke, a uz to još neke dodatne mogućnosti poput Kotlin Korutina (engl. *Kotlin Coroutines*). Glavne beneficije rada u Kotlin programskom jeziku su [17]:

- Manje koda i lakša čitljivost
- Zreo jezik i okruženje
- Kotlin podrška u Android Jetpack i ostalim bibliotekama
- Interoperabilnost sa Java programskim jezikom
- Podrška za više platformi

3.3 Firebase

Firebase je *Google*-ova platforma za razvoj mobilnih i web aplikacija koja je započela 2011. kao *startup* kompanija, dok je *Google* nije otkupio 2014. godine. U načelu Firebase je *backend-as-a-service* (skraćeno BAAS) što znači da razvojnom inženjeru omogućuje razvijanje klijentske strane i integraciju usluga unutar aplikacije. Oslobađa razvojne inženjere rada i upravljanja sa serverom, pisanja API-a (engl. *Application Program Interface*) i brine se za spremanje podataka. Osnovne mogućnosti Firebase platforme su:

- Spremanje podataka na oblaku
- Autentikacija korisnika
- Baza podataka u stvarnom vremenu
- Analitike za poboljšanje kvalitete aplikacije

Firebase autentikacija koristi se za potvrđivanje korisnika u aplikaciji. Omogućuje *backend* servise, skup SDK (engl. *Software Development Kit*) koji se lako koriste, i gotove UI (engl. *User Interface*) biblioteke za autentifikaciju korisnika unutar aplikacije. Podržava autentifikaciju koristeći zaporku, broj mobitela, popularne usluge identifikacije poput *Google-a*, Facebook-a, Twitter-a i sl. [18].

Cloud Firestore je fleksibilna, skalabilna baza podataka za mobilno, web i server razvijanje aplikacija sa Firebase i *Google* Cloud platforme [19]. Koristi NoSQL bazu podataka kako bi spremila velike količine podataka razne vrste. NoSQL je alternativa tradicionalnim bazama podataka gdje se podaci pohranjuju u tablice, a shema podataka se pažljivo izrađuje prije izrade same baze podataka.

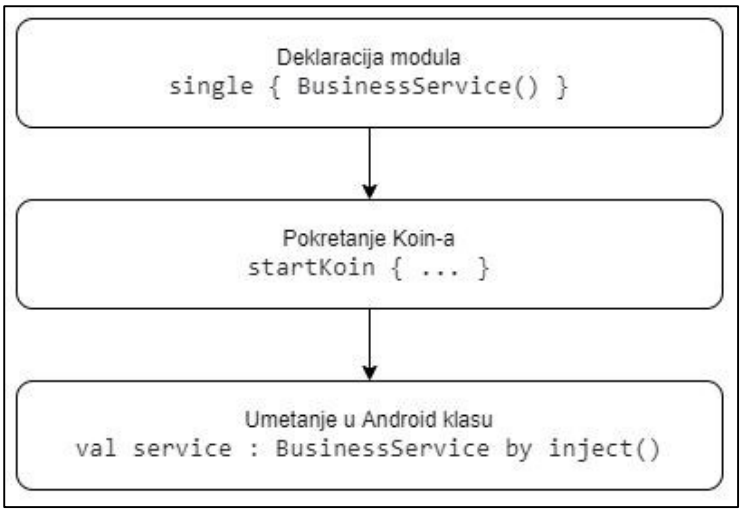
3.4 Koin dependency injection

Umetanje ovisnosti (*engl. dependency injection*) omogućava postavljanje temelja za dobru arhitekturu aplikacije. Implementacija *dependency injection-a* u aplikaciju omogućuje [20]:

- Ponovno korištenje koda (*engl. reusability*)
- Lagano refaktoriranje (*engl. refactoring*)
- Lagano testiranje

Koin je pragmatički lagani okvir (engl. *Framework*) za umetanje ovisnosti koji koriste Kotlin razvojni inženjeri. Napisan je u čistom Kotlinu bez proxy-a, generiranja novog koda i refleksija [21]. Vrlo je jednostavan za korištenje, a sastoji se od tri osnovna koraka (slika 3.3.):

- Deklariranje modula koji će se koristiti
- Pokretanje Koina prilikom kojeg se navodi lista deklariranih modula
- Umetanje u Android klasu u kojoj je to potrebno



Slika 3.3. – Koraci za korištenje Koina

4 RAZVOJ APLIKACIJE ZA DETEKTIRANJE I MAPIRANJE RELJEFA POVRŠINE KOLNIKA

4.1 Korisnički zahtjevi

Prije početka izrade aplikacije potrebno je definirati korisničke zahtjeve. Aplikacija služi za detekciju rupa i udubina na cestovnim prometnicama. Koristiti će je korisnici koji prilikom vožnje žele očitavati podatke sa akcelerometra na Android uređaju koji je postavljen u autu. Mogućnosti korisnika podijeljene su u pet grupa. Prva grupa mogućnosti je autentikacija, a korisnik će imati mogućnosti:

- Registracija - za registraciju je potrebno unijeti adresu elektroničke pošte, korisničko ime i zaporku
- Prijava - za prijavu već registriranog računa potrebno je unijeti adresu elektroničke pošte i zaporku

Druga grupa mogućnosti je očitavanje vrijednosti. Korisnik će imati mogućnost:

- Pokretanje očitavanja vrijednosti na akcelerometru i dobivanja trenutne lokacije pomoću gumba za pokretanje
- Odabir postavki
- Prikaz trenutne vrijednosti akceleracije
- Zapis očitanih vrijednosti i lokacije koristeći *RecyclerView*

Treća grupa mogućnosti uključuje pregled očitanih podataka prije dodavanja na kartu. Korisnik će imati mogućnost:

- Pregled zabilježenih podataka iz lokalne baze podataka i spremanje u online bazu podataka
- Pregled zabilježenih podataka spremljenih u online bazu podataka
- Prikaz odabrane zabilježene vrijednosti na karti
- Brisanje odabrane zabilježene vrijednosti

Četvrta grupa mogućnosti je prikaz karte sa svim zabilježenim mjestima gdje su detektirane rupe na cesti. Korisnik će imati mogućnost:

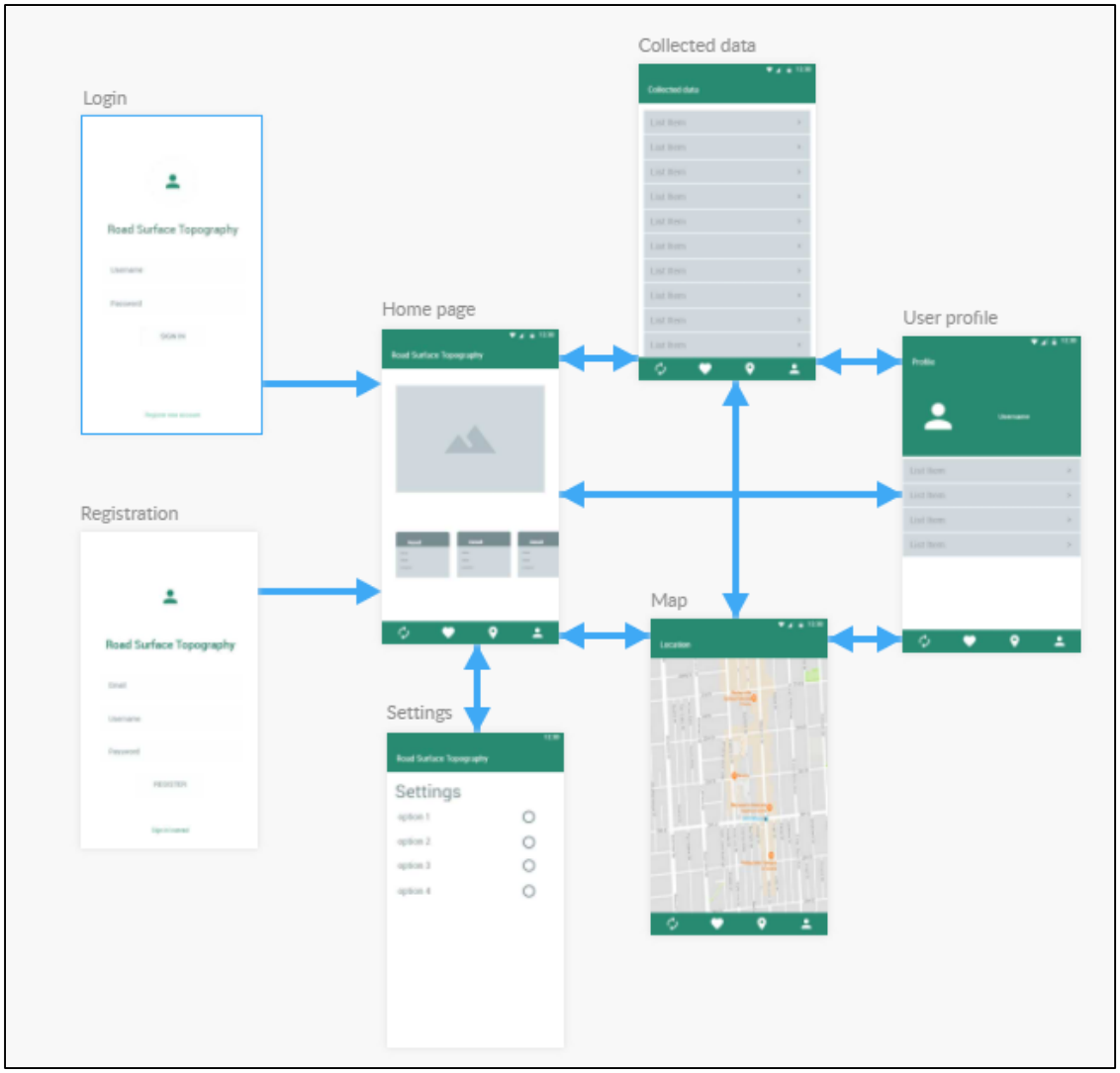
- Mapiranje podataka iz lokalne baze podataka
- Mapiranje podataka iz online baze podataka
- Odabir markera za prikaz očitane vrijednosti i otvaranje istog u *Google Maps* aplikaciji

Zadnja grupa mogućnosti je prikaz korisničkog profila na kojemu će biti moguće:

- Prikaz korisničkog imena
- Odjava iz aplikacije
- Prikaz svih zabilježenih vrijednosti
- Prikaz zabilježenih vrijednosti na karti

4.2 Izrada modela korisničkog sučelja

Kako bi se olakšala sama izrada aplikacije napravljena je maketa (engl. *mock-up*) korisničkog sučelja. Za izradu makete korišten je online alat Fluid-UI [22] koji na jednostavan način omogućuje izradu izgleda aplikacije. Maketa aplikacije vidljiva je na slici 4.1.

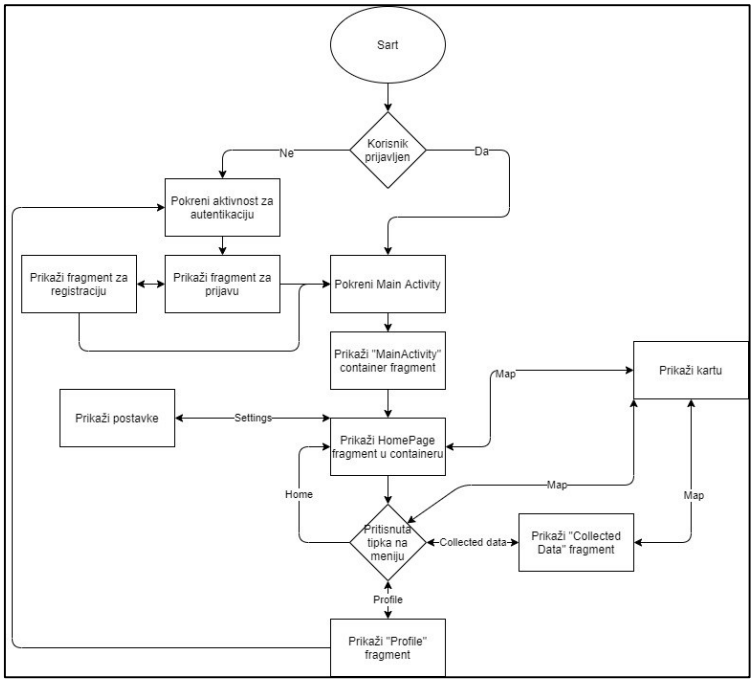


Slika 4.1. – *Mock-up* aplikacije

Na slici su prikazani svi ekrani koji će biti dostupni u aplikaciji. Uz njihov izgled prikazane su veze između pojednih dijelova aplikacije. Nakon prijave ili registracije korisnik će se u aplikaciji navigirati koristeći *BottomNavBar* pomoću kojega se može prebacivati sa jednoga ekrana na drugi. Aplikacija sadrži šest ekrana a to su:

- *Login* ekran - služi za prijavu korisnika, ukoliko je prijava uspješna korisnika se prebacuje na *Home Page* ekran
- *Register* ekran - služi za registraciju korisnika, ukoliko je registracija uspješna korisnika se prebacuje na *Home Page* ekran
- *Home Page* ekran - sadržavati će grafikon koji prikazuje trenutna očitavanja na akcelerometru, te očitane vrijednosti unutar *RecyclerView-a*
- *Settings* ekran - sadržavati će odabir za postavke aplikacije
- *Collected Data* ekran - sadržavati će sve zabilježene podatke u *RecyclerView* pregledu, odabirom podatka korisnika će se preusmjeravati na kartu
- *Map* ekran - prikazuje sva mjesta gdje su zabilježene udubine u cesti
- *User Profile* ekran - prikazuje korisničke podatke

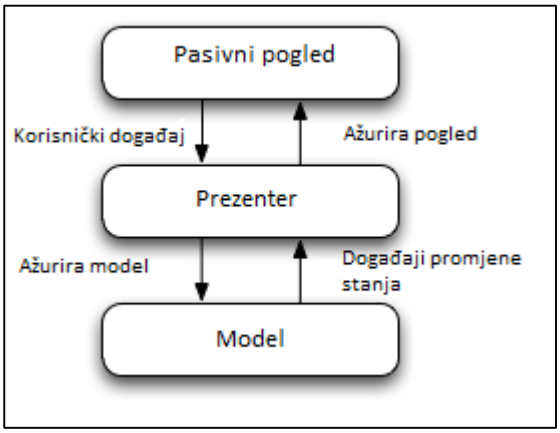
Kako bi se znao točan tijek aplikacije izrađen je dijagram toka. Dijagram pokazuje sve mogućnosti promjena ekrana od početka korištenja aplikacije i prijave u sustav, do kraja i odjave korisnika. Dijagram toka prikazan je na slici 4.2.



Slika 4.2. – Dijagram toka aplikacije

4.3 Arhitektualni dizajn sustava

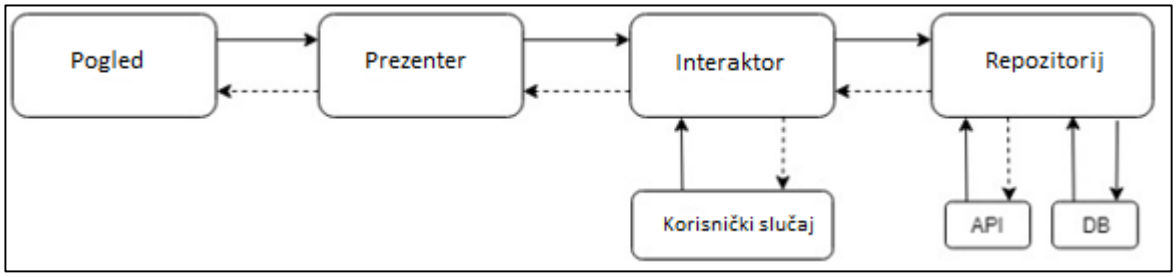
Za izradu aplikacije odabrana je *Model-View-Presenter* arhitektura. Skraćeno MVP je je derivacija *Model-View-Controler* arhitekture i koristi se za izradu korisničkih sučelja [23]. Kao što samo ime govori MVP se sastoji od tri dijela koji međusobno surađuju. *View* odnosno pogled se brine za sve što se nalazi na "vidljivoj" strani aplikacije, kao što je postavljanje korisničkog sučelja, teksta i ostalih vizualnih dijelova. Model dio arhitekture sadrži sve modele koji se koriste unutar aplikacije, kao što su podaci o korisniku i očitavanja sa akcelerometra. *Presenter* dio koji se nalazi između ostala dva dijela i pomoću podataka iz modela ažurira podatke koji se vide na zaslonu korisnika. *Model-View-Presenter* dijagram prikazan je na slici 4.3.



Slika 4.3. – MVP dijagram

Na slici 4.4. prikazana je struktura koda aplikacije. Sastoji se od četiri glavne skupine i podskupina koje se međusobno nadopunjuju kako bi se osigurao lakši pregled koda, odnosno kod je razdvojen na više cjelina koje imaju svoj posao. Glavne cjeline koda su:

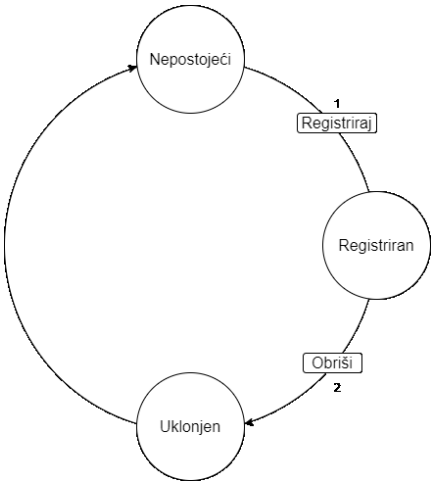
- *View* - postavlja korisničko sučelje i brine se za prikaz podataka na zaslonu
- *Presenter* - uzima i prezentira podatke *View* dijelu
- *Interactor* - interaktor pomoću kojega *Presenter* koristi napisane funkcijske dijelove koda
- *Repository* - služi za interakciju za bazom podataka i vanjskim servisima



Slika 4.4. – Struktura koda aplikacije

4.4 Životni krugovi korisnika i očitanih podataka

Kako bi mogao zabilježiti očitanja sa akcelerometra prilikom prelaska preko udubina na cesti, korisnik se mora registrirati odnosno prijaviti u sustav. Nakon prijave može koristiti sve mogućnosti aplikacije i ukoliko želi svoj korisnički račun može ukloniti. Životno krug korisničkog računa ima tri moguća stanja, a između stanja postoje određene operacije koje je potrebno izvršiti. Životni krug korisnika prikazan je na slici 4.5. dok su operacije između stanja prikazana u tablici 4.1.

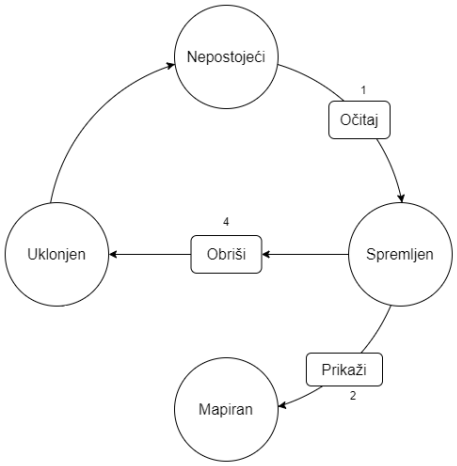


Slika 4.5. – Životni krug korisnika

Tablica 4.1. – Opis svih stanja i operacija životnog kruga

Trenutno stanje	Operacija	Opis operacije
Nepostojeći	1 (Registriraj)	Korisnikovi podaci su registrirani na Firebase servis za autentifikaciju
Registriran	2 (Obriši)	Korisnički račun se briše
Uklonjen	/	Korisnik je uklonjen sa Firebase servisa

Nakon očitavanja podatka sa akcelerometra potrebno ga je spremiti u bazu podataka i nakon toga postoji mogućnost brisanja ili mapiranja istog. Očitani podataka zbog toga ima četiri različita stanja. Životni krug očitanog podatka prikazan je na slici 4.6. dok su operacije između stanja opisane u tablici 4.2.



Slika 4.6. – Životni krug očitanih podataka

Tablica 4.2. – Opis svih stanja i operacija životnog kruga

Trenutno stanje	Operacija	Opis operacije
Nepostojeći	1 (Očitaj)	Podatak je očitan sa akcelerometra i spremljen u bazu podataka
Spremljen	2 (Prikaži)	Korisnik odabire podatak i on se prikazuje na mapi
Spremljen	4 (Obriši)	Korisnik briše podatak iz baze podataka
Uklonjen	/	Podatak je uklonjen iz baze

4.5 Dodavanje ovisnosti

Prvi korak u izradi aplikacije je dodavanje ovisnosti (engl. *dependencies*) koje će ona koristiti. Dodavanje ovisnosti omogućuje korištenje raznih biblioteka i mogućnosti koje će biti potrebne prilikom izrade. Dodane ovisnosti podijeljene su u sedam grupa (programski kod 4.1.) a to su:

- *ui* - sadrži sve ovisnosti potrebne za izradu korisničkog sučelja kao što su *RecyclerView* i *CardView* koji služe za prikaz podataka
- *firebase* - sadrži ovisnosti potrebne za korištenje Firebase servisa
- *firestore* - sadrži ovisnosti potrebne za spremanje podataka na *firestore* servis
- *di* - sadrži ovisnosti za korištenje Koin Dependency Injection-a
- *db* - sadrži ovisnosti za rad sa lokalnom bazom podataka a to je Room Persistence biblioteka koja koristi SQlite bazu podataka. Biblioteka omogućava stvaranje sučelja u kojemu se deklariraju željeni upiti (engl. Query) pomoću kojih se dohvaćaju i spremaju željeni podaci
- *map* - ovisnost potreba za rad sa *Google maps* servisima
- *location* - služi za dohvaćanje trenutne lokacije mobilnog uređaja

```
//ui
implementation 'com.google.android.material:material:1.1.0'
implementation 'androidx.recyclerview:recyclerview:1.1.0'
implementation 'androidx.cardview:cardview:1.0.0'
//firebase
implementation 'com.google.firebase:firebase-auth:19.3.1'
//firestore
implementation 'com.google.firebase:firebase-firestore-ktx:21.7.0'
//di
implementation "org.koin:koin-core:2.1.5"
implementation "org.koin:koin-android-scope:2.1.5"
implementation 'org.koin:koin-androidx-viewmodel:2.1.5'
implementation "org.koin:koin-android-ext:2.1.5"
//db
implementation "androidx.room:room-runtime:2.2.0-beta01"
kapt "android.arch.persistence.room:compiler:2.2.0-beta01"
//map
implementation 'com.google.android.gms:play-services-maps:17.0.0'
//location
implementation 'com.google.android.gms:play-services-location:17.0.0'
```

Programski kod 4.1. – Korištene ovisnosti

4.6 Izrada korisničkog sučelja

Nakon izrađene makete korisničkog sučelja potrebno ju je implementirati u aplikaciju. Aplikacija će se sastojati od dvije glavne aktivnosti a to su autentifikacija korisnika i glavna aktivnost. Aktivnost je jedinstvena, fokusirana stvar koju korisnik može raditi. Gotovo sve aktivnosti imaju neku vrstu interakcije sa korisnikom, tako da se *Activity* klasa brine za stvaranje prozora u koji će se postavljati korisničko sučelje [24]. Programski kod 4.2. prokazuje aktivnost za autentifikaciju korisnika.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.authentication.AuthenticationActivity">

    <FrameLayout
        android:id="@+id/authFragmentContainer"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

</androidx.constraintlayout.widget.ConstraintLayout>
```

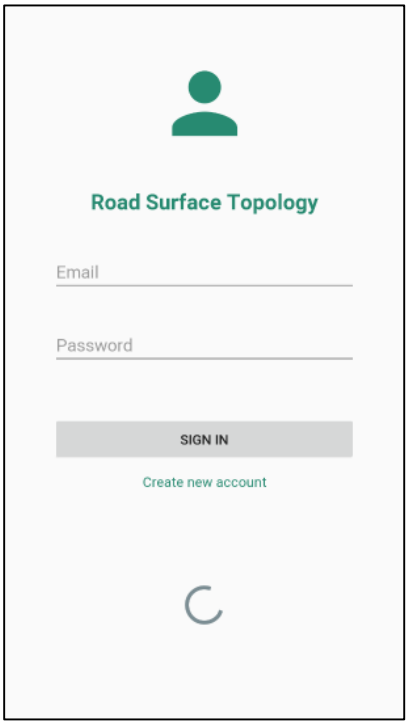
Programski kod 4.2. – XML kod aktivnosti za autentifikaciju korisnika

FrameLayout prikazan u programskom kodu 4.2. služi kao kontejner za ubacivanje fragmenta u kojemu se nalazi sami izgled ekrana. Android fragment je dio aktivnosti, poznat je i kao pod-aktivnost. Unutar jedne aktivnosti može biti više različitih fragmenata, odnosno više različitih ekrana [25]. Ovisno o potrebi fragmenti unutar aktivnosti se izmjenjuju, primjerice u aktivnosti za autentifikaciju korisnika dolaziti će do izmjene između fragmenta za prijavu odnosno registraciju korisnika. Za izmjenu fragmenta koristit će se Kotlin *Extension* funckija koja vrši transakciju specificiranog fragmenta u zadani *FrameLayout* kontejner. Funkcija *showFragment()* je prikazana u programskom kodu 4.3.

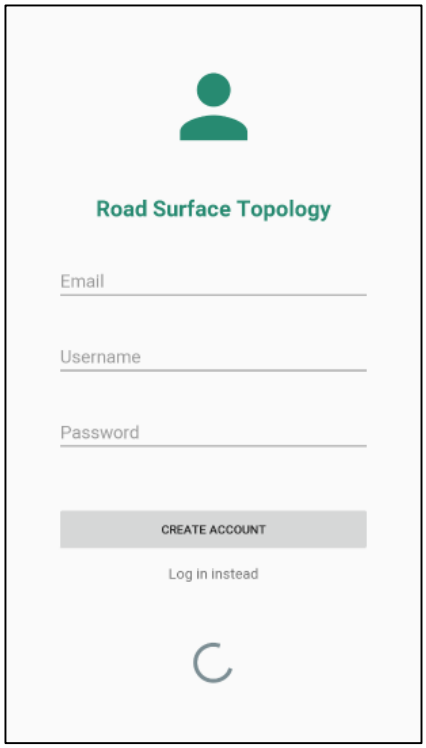
```
fun FragmentActivity.showFragment(containerId: Int, fragment: Fragment,
shouldAddToBackStack: Boolean = false, tag: String = ""){
    supportFragmentManager.beginTransaction().apply {
        if(shouldAddToBackStack){
            addToBackStack(tag)
        }
    }.replace(containerId, fragment).commitAllowingStateLoss()
}
```

Programski kod 4.3. – Funkcija *showFragment()*

Nakon toga potrebno je napraviti izgled fragmenata za prijavi i registraciju koji će se koristiti u aplikaciji. Fragment za prijavu mora sadržavati tekstualno polje za upis email adrese i zaporke te tipku za prijavu i odlazak na ekran za registraciju. Fragment za registraciju mora sadržavati tekstualno polje za upis email adrese, korisničkog imena i zaporke te tipku za registraciju i odlazak na ekran za prijavu. Izgled ekrana prikazan je na slikama 4.7. i 4.8.



Slika 4.7. – Ekran za prijavu korisnika

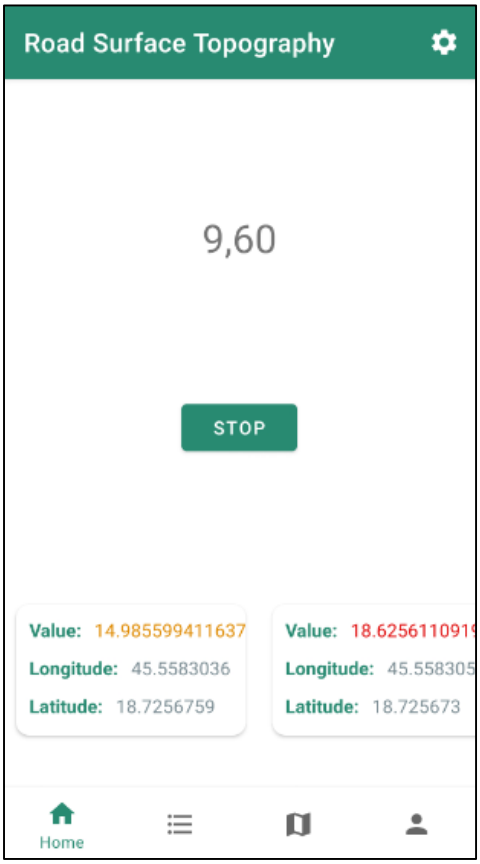


Slika 4.8. – Ekran za registraciju korisnika

Nakon što su napravljeni fragmenti za autentifikaciju, potrebno je napraviti fragmente za glavnu aktivnost u kojoj će se odvijati glavne funkcionalnosti aplikacije. U ovoj aktivnosti potrebno je 5 različitih fragmenta:

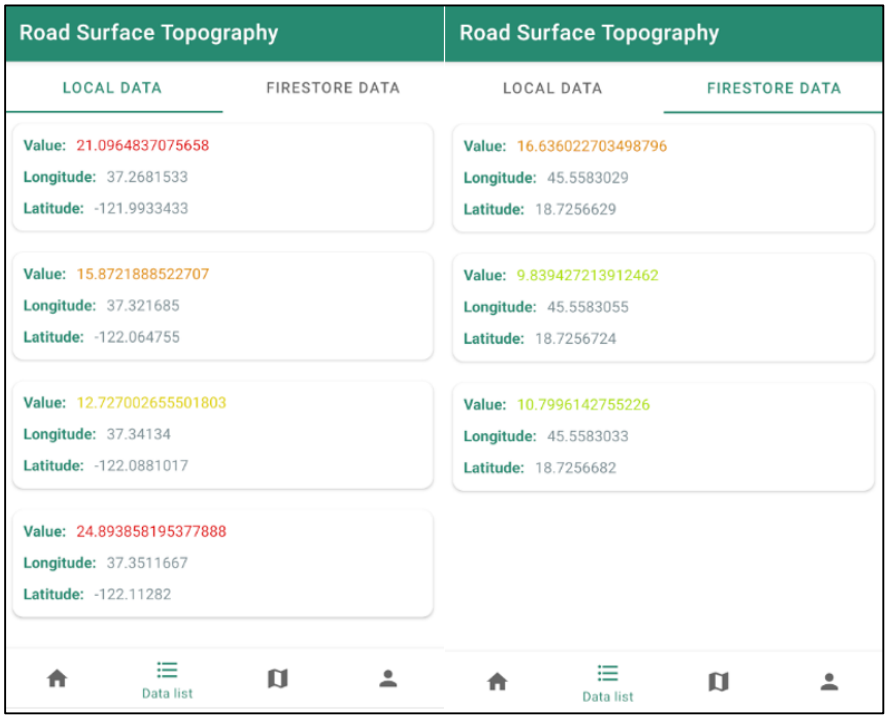
- Fragment za očitavanje vrijednosti
- Fragment za prikaz podataka iz lokalne baze podataka
- Fragment za prikaz podataka iz online baze podataka
- Fragment za prikaz podataka na karti
- Fragment za prikaz profila korisnika

Fragment za očitavanje vrijednosti je najbitniji dio aplikacije u kojemu će se vršiti prikupljanje podataka sa senzora. Također služi za prikaz trenutne akceleracije i očitanih vrijednosti sa akcelerometra i trenutne lokacije koristeći *RecyclerView*. Gumb „START“ i „STOP“ služi za pokretanje odnosno prekid očitavanja podataka i dohvaćanja lokacije. Svrha gumba u obliku kotačića je prikaz ekrana za postavke koji će biti objašnjen u slijedećem poglavlju. Izgled fragmenta prikazan je na slici 4.9.



Slika 4.9. – Ekran za očitavanje vrijednosti

Fragment za prikaz podataka iz baze podataka omogućuje prikaz podataka koji su spremljeni u lokalnu bazu podataka i podataka koji su od strane korisnika spremljeni na *Firestore* bazu podataka. Kako bi se ovo ostvarilo koristit će se *TabLayout* koji omogućuje prebacivanje iz jednog fragmenta u drugi na jednostavan način. Uz prikaz podataka u fragmentu sa lokalnim podacima biti će moguće obrisati podatak pomakom prsta u lijevo, i prebacivanje podataka na *Firestore* bazu podataka pomakom prsta u desno. U fragmentu u kojemu se prikazuju podaci iz *Firestore* baze biti će moguće obrisati podatak pomakom prsta u lijevo. Također klikom na podatak on će biti prikazan na karti. Izgled fragmenta prikazan je na slici 4.10.



Slika 4.10. – Ekran za prikaz podataka iz baze podataka

Fragment za prikaz očitanih podataka na karti prikazivati će očitane nepravilnosti na cesti sa različitim markerima u ovisnosti o njihovoj veličini. Markeri su podijeljeni u pet grupa, a svakom markeru dodijeljena je ikona u boji koja karakterizira veličinu zabilježene nepravilnosti. Mapiranje će u cijelosti biti objašnjeno u budućim poglavljima.

Prilikom učitavanja karte potrebno je prikazati odabranu lokaciju, odnosno potrebno je postaviti kameru na mjesto lokacije gdje će se prikazati marker. Stvaranjem objekta klase *CameraPosition* moguće je upravljati kamerom, animacijom i brzinom približavanja prema zadanoj lokaciji. Funkcija *setCamera()* prikazana je u programskom kodu 4.4.

```
private fun setCamera(map: GoogleMap, selectedLocation: Pair<Double, Double>) {
    val location = LatLng(selectedLocation.first, selectedLocation.second)
    val cameraPosition = CameraPosition
    .Builder().target(location).zoom(17.0F).build()
    map.animateCamera(CameraUpdateFactory.newCameraPosition(cameraPosition))
}
```

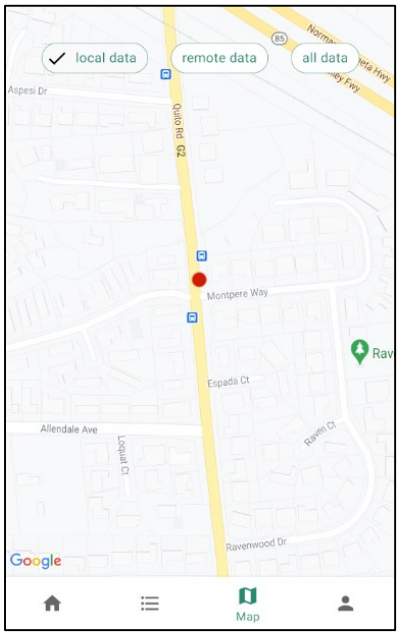
Programski kod 4.4. – Funkcija *setCamera()*

Nakon što se karta učita u fragment, korisnik ima mogućnost odabira koje podatke želi prikazati na karti. Mogućnosti prikaza su:

- *Local data* - na karti prikazuje sve korisničke podatke iz lokalne baze podataka

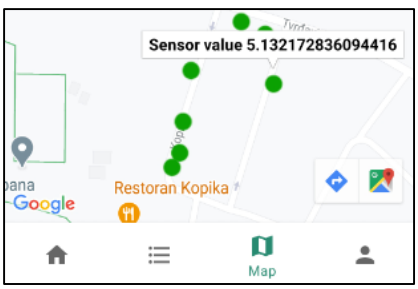
- *Remote data* - na karti prikazuje sve korisničke podatke iz *Firestore* baze podataka
- *All data* - na karti prikazuje podatke od svih korisnika

Izgled fragmenta prikazan je na slici 4.12.



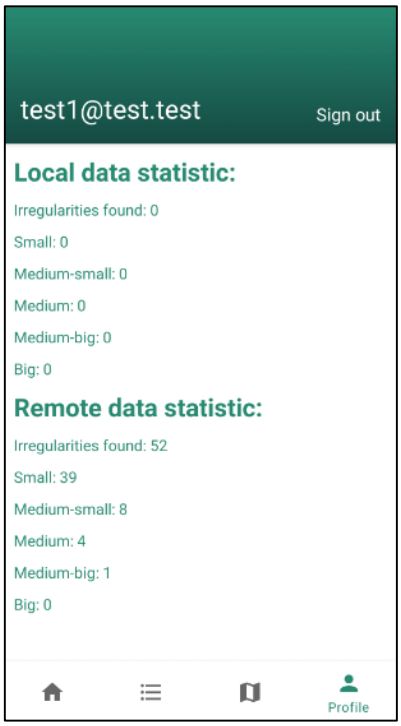
Slika 4.12. – Ekran za prikaz podataka na karti

Također, klikom na marker prikazuje se vrijednost sa senzora po kojoj je entitet spremljen u bazu podataka. Uz prikaz vrijednosti o donjem desnom kutu stvara se opcija za dobivanje uputa do dotičnog markera te otvaranje markera u *Google maps* aplikaciji u kojoj su između ostaloga može otvoriti *Google Street View* na zabilježenom mjestu. Slika 4.13. prikazuje tu funkcionalnost.



Slika 4.13. – Prikaz vrijednosti markera i daljnje opcije

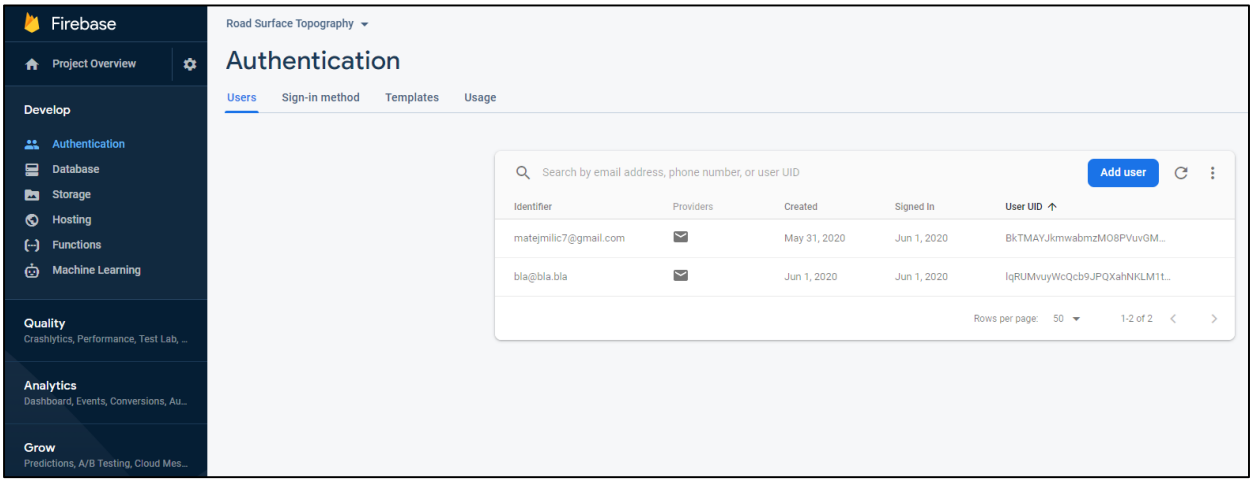
Fragment za prikaz profila korisnika služi za prikaz korisničkog imena i odjavu korisnika iz aplikacije. Uz to u fragmentu su zabilježeni ukupan broj zabilježenih nepravilnosti od strane korisnika, te udjel od svake klase nepravilnosti za lokalnu i online bazu podataka. Ekran za prikaz profila korisnika prikazan je na slici 4.14.



Slika 4.14. – Ekran za prikaz korisničkog profila

4.7 Prijava i registracija korisnika

Kako bi korisnik mogao koristiti sve mogućnosti aplikacije potrebno je izraditi prijavu i registraciju. Za autentifikaciju korisnika pomoću adrese elektroničke pošte i zaporke koristit će se Firebase servis. Za korištenje dotičnog servisa potrebno je stvoriti novi projekt u Firebase konzoli i zatim ga koristiti u aplikaciji. Stvaranje novog projekta vrlo je jednostavno koristeći Firebase konzolu nakon čega je potrebno omogućiti prijavu i registraciju korisnika koristeći email adresu. Izgled Firebase konzole prikazan je na slici 4.15. Nakon što je projekt izrađen u konzoli, dodaje se u projekt u Android studiju kao ovisnost.



Slika 4.15. – Firebase konzola

Prilikom pokretanja aplikacije korisniku će biti prikazan ekran za prijavu koji je prikazan u prethodnom poglavlju. Kako bi se korisnik prijavio u aplikaciju potrebno je upisati adresu elektroničke pošte te korisničku zaporku. Ukoliko su podaci ispravni korisnika se preusmjerava na glavnu aktivnost. U suprotnom na ekranu korisnika se ispisuje poruka da je došlo do pogreške. Kod za prijavu korisnika prikazan je u programskom kodu 4.5. dok su funkcije za uspješnu i neuspješnu prijavu prikazani u programskom kodu 4.6.

```
class LoginUseCaseImpl(private val auth: FirebaseAuth): LoginUseCase {
    override fun execute(body: UserData, onSuccess:
SuccessLambda<Task<AuthResult>>, onFailure: ErrorLambda<Task<AuthResult>>) {
        auth.signInWithEmailAndPassword(body.email,
body.password).addOnCompleteListener {
            if (it.isComplete && it.isSuccessful){
                it.run(onSuccess)
            }else{
                it.run(onFailure)
            }
        }
    }
}
```

Programski kod 4.5. – Prijava korisnika

```
override fun onLoginSuccessful() {
    startActivity(HomePageActivity::class.java)
}

override fun onLoginFailed() {
    Toast.makeText(RoadSurfaceTopography.instance, "Failed to log in",
    Toast.LENGTH_SHORT).show()
}
```

Programski kod 4.6. – Funkcije za uspješnu i neuspješnu prijavu

Ukoliko korisnik nema izrađen korisnički račun odabire opciju „*Create new account*“ i preusmjerava ga se na ekran za stvaranje novog računa. Prilikom stvaranja računa potrebno je unijeti adresu elektroničke pošte, korisničko ime i zaporku. Nakon unesenih podataka šalje se zahtjev za registraciju na *Firebase* servis. Ukoliko su podaci jedinstveni stvara se novi korisnik u bazi podataka i preusmjerava ga se na glavnu aktivnost. Ukoliko su podaci neispravni potrebno je unijeti nove podatke. Kod za registraciju korisnika prikazan je u programskom kodu 4.7. dok su funkcije za uspješnu i neuspješnu registraciju prikazane u programskom kodu 4.8.

```
class RegistrationUseCaseImpl (private val auth: FirebaseAuth):
RegistrationUseCase {
    override fun execute(body: UserData, onSuccess:
SuccessLambda<Task<AuthResult>>, onFailure: ErrorLambda<Task<AuthResult>>){
        auth.createUserWithEmailAndPassword(body.email,
body.password).addOnCompleteListener {
            if (it.isComplete && it.isSuccessful){
                auth.currentUser?.updateProfile(UserProfileChangeRequest
                    .Builder()
                    .setDisplayName(body.username)
                    .build())
                it.run(onSuccess)
            }else{
                it.run(onFailure)
            }
        }
    }
}
```

Programski kod 4.7. – Registracija korisnika

```
override fun onRegisterSuccessful() {
    startActivity(HomePageActivity::class.java)
}

override fun onRegisterFailed() {
    Toast.makeText(RoadSurfaceTopography.instance, "Failed to register",
Toast.LENGTH_SHORT).show()
}
```

Programski kod 4.7. – Funkcije za uspješnu i neuspješnu registraciju

4.8 Spremanje i rad sa podacima

Kako bi se očitani podaci mogli koristiti za analizu i mapiranje potrebno ih je spremiti u bazu podataka. Za potrebe aplikacije predviđeno je stvaranje lokalne i *online* baze. Svrha lokalne baze podataka biti će pregledavanje očitanih rezultata i njihov prikaz na karti kako bi se nepotrebni ili krivo zabilježeni podaci mogli odbaciti. Svrha *online* baze podataka je mapiranje zabilježenih podataka od svih korisnika kako bi se stvorila ispravna karta reljefa prometnica.

Za izradu lokalne baze podataka koristiti će se biblioteka *Room Persistance* koja koristi *SQLite* bazu podataka za jednostavno spremanje podataka. *Room* pruža apstrakcijski sloj iznad *SQLite* baze podataka kako bi dozvolio tečno pristupanje bazi dok u isto vrijeme iskorištava punu moć *SQLite-a* [26]. Za korištenje *Room* baze podataka potrebne su tri stvari. Prvo je potrebno napraviti instancu baze podataka preko koje će joj se pristupati bazi, nakon toga se stvara DAO (*engl. Data Access Objects*) sučelje koje sadrži metode za pristup bazi. Na kraju je potrebno napraviti entitet koji predstavlja tablicu unutar baze.

Entitet za bazu podataka napravljen je kao *data class* ispred koje se dodaje anotacija *@Entity* u kojoj se definira ime tablice. Tablica *sensorData* će sadržavati slijedeće atribute:

- *id* - jedinstveni identifikacijski broj za svaki spremljeni podatak
- *user* - korisničko ime korisnika koji sprema podatak
- *sensorValue* - razlika najveće i najmanje vrijednosti akceleracije prilikom prelaska preko nepravilnosti
- *latitude* - očitana zemljopisna širina
- *longitude* - očitana zemljopisna dužina
- *bumpType* - tip očitane nepravilnosti

Programski kod za stvaranje entiteta prikazan je u programskom kodu 4.8.

```
@Entity(tableName = "sensorData")
data class SensorDataDb (
    @PrimaryKey(autoGenerate = true) var id: Long = 0,
    @ColumnInfo(name = "user") val user: String = "",
    @ColumnInfo(name = "sensor_value") val sensorValue: Double = 0.00,
    @ColumnInfo(name = "locationX") val locationX: Double = 0.00,
    @ColumnInfo(name = "LocationY") val locationY: Double = 0.00,
    @ColumnInfo(name = "bump_type") val bumpType: Int = 0
)
```

Programski kod 4.8. – Programski kod za stvaranje entiteta

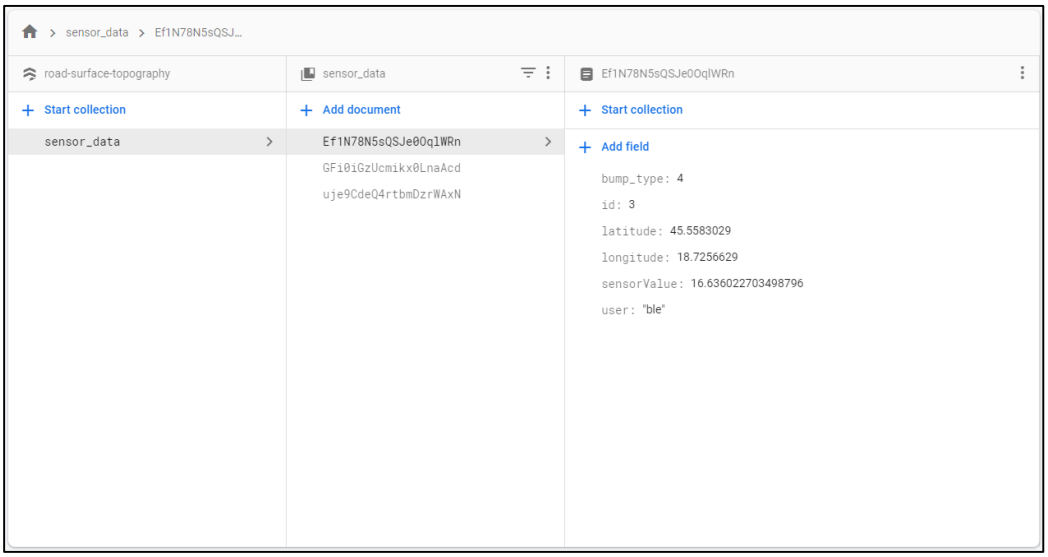
Nakon što je napravljena tablica potrebno je napraviti DAO sučelje. Sučelje će sadržavati metode i upite na bazu podataka koji su potrebni za spremanje i brisanje istih. Kod za DAO sučelje prikazan je u programskom kodu 4.9.

```
@Dao
interface SensorDataDao {
    @Insert
    fun insertSensorData(sensorData: SensorDataDb)

    @Query("SELECT * FROM sensorData WHERE user = :user ORDER BY id DESC ")
    fun getAllSensorData(user: String): List<SensorDataDb>

    @Query("DELETE FROM sensorData WHERE id = :id")
    fun removeSensorData(id: Long)
}
```

Programski kod 4.9. – DAO sučelje

Nakon što je izrađena lokalna baza potrebno je postaviti *online* bazu podataka. Kao što je objašnjeno u prethodnim poglavljima za tu svrhu koristiti će se *Firebase Firestore* servis za spremanje podataka. Pošto je riječ o *NoSQL* bazi podataka, podaci se spremaju u kolekcije koje sadrže dokumente u kojima se nalaze polja sa vrijednostima. Kako bi se omogućilo korištenje *Firestore* servisa osim dodavanja ovisnosti potrebno je postaviti projekt na *Firebase* konzoli, a nakon toga moguće je manualno stvaranje kolekcije i postavljanje polja unutar dokumenta. Osim manualnog stvaranja moguće je stvaranje kolekcije kroz programski kod, gdje se pri prvome spremanju podataka kolekcija automatski stvori u *Firestore* bazi podataka. Izgled kolekcije

sensor_data prikazan je na slici 4.16. i kao što je vidljivo sadrži sve iste vrijednosti kao i lokalna baza podataka.



Slika 4.16. – Izgled *Firestore* kolekcije u *Firebase* konzoli

Dohvaćanje podataka vrši se pomoću jednostavnih upita. *Firestore* omogućuje stvaranje učinkovitih i fleksibilnih upita, filtriranje, sortiranje i ograničavanje upita za dohvaćanje rezultata bez potrebe za dohvaćanjem cijele kolekcije ukoliko je to potrebno. U svrhu aplikacije potrebno je izraditi upite za dohvaćanje podataka po korisničkom imenu kako bi se korisniku mogli prikazati njegovi podaci, te dohvaćanje podataka od svih korisnika u svrhu prikaza reljefa prometnica. Dohvaćanje podataka iz kolekcije *sensor_data* prikazano je u programskom kodu 4.10.

```
override fun getData(user: String, onGetDataSuccessful: onGetDataSuccessful,
onGetDataFailed: onGetDataFailed) {
    firebaseRepository.getFireStore().collection("sensor_data")
        .whereEqualTo("user", user)
        .get()
        .addOnSuccessListener { run { onGetDataSuccessful(it) } }
        .addOnFailureListener { run(onGetDataFailed) }
}

override fun getAllData(onGetDataSuccessful: onGetDataSuccessful, onGetDataFailed:
onGetDataFailed) {
    firebaseRepository.getFireStore().collection("sensor_data")
        .get()
        .addOnSuccessListener { run { onGetDataSuccessful(it) } }
        .addOnFailureListener { run(onGetDataFailed) }
}
```

Programski kod 4.10. – Dohvaćanje podataka sa *Firestore* servisa

5 RAZVOJ ALGORITMA ZA DETEKCIJU I KLASIFIKACIJU NEPRAVILNOSTI

5.1 Postavljanje akcelerometra i lokacijske usluge

Prije izrade i implementacije algoritma potrebno je postaviti potrebne senzore. Za potrebu detekcija nepravilnosti i njihovog mapiranja koristit će se akcelerometar i usluga za detekciju korisnikove lokacije.

Pri postavljanju akcelerometra prvo je potrebno definirati tip senzora koji će se koristiti. Postoji više vrsta rada akcelerometra, a za potrebe aplikacije odabran je *TYPE_ACCELEROMETER*. Ovaj tip senzora aplikaciji pruža podatke o „sirovim“ podacima o linearnim i rotacijskim silama koje djeluju na mobilni uređaj, uključujući i gravitacijsku silu[27]. Pošto je uključena i gravitacijska sila u stanju mirovanja vrijednost akceleracije je 9.81, a ako se uređaj nalazi u stanju „padanja“ vrijednost će biti 0. Postavljanje akcelerometra prikazano je u programskom kodu 5.1.

```
sensorManager = getActivity()?.getSystemService(Context.SENSOR_SERVICE) as
SensorManager
accelerationSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
```

Programski kod 5.1. – Postavljanje tipa akcelerometra

Nakon postavljanja tipa senzora, potrebno je registrirati slušatelja (engl. *Listener*) za promjene na senzoru. Za to se koristi funkcija *registerListener()* kojoj je potrebno predati slijedeće parametre:

- *sensorListener* - slušatelj promjena u kojemu će se implementirati algoritam za filtriranje i detekciju anomalija prilikom promjene akceleracije
- *accelerationSensor* - senzor koji će se koristiti i unaprijed definirani tip senzora koji je vidljiv u programskom kodu 5.1.
- *samplingPeriod* - period uzimanja uzoraka sa akcelerometra, postavljen na vrijednost *SENSOR_DELAY_FASTEST* koja uzima najviše moguće uzoraka. Ova vrijednost je odabrana kako bi se što preciznije detektirala promjena akceleracije

Slušatelj će se registrirati prilikom pritiska tipke „START“ koja započinje mjerenje, a kod je prikazan u programskom kodu 5.2.

```
private fun onButtonClicked() {
    sensorManager.registerListener(sensorListener, accelerationSensor,
    SENSOR_DELAY_FASTEST)
}
```

Programski kod 5.2. – Registracija slušatelja za akcelerometar

Prilikom očitavanja nepravilnosti na akcelerometru potrebno je znati trenutno lokaciju korisnika. Za to je potrebno dobivati konstanta ažuriranja trenutne lokacije, što je moguće pomoću *Google* API-a. Kako bi se to ostvarilo, potrebno je dobiti dopuštenje korisnika za dohvaćanje trenutne lokacije. Programski kod 5.3. pokazuje postavljanje dopuštenja (engl. *Permission*) u *Manifest* datoteci. Za potrebe aplikacije odabrana je vrijednost *ACCESS_FINE_LOCATION* koja omogućuje najpreciznije ažuriranje trenutne lokacije. Prilikom prvog pokretanja aplikacije korisnika će se tražiti dopuštenje za pristup lokaciji, nakon što ga aplikacija dobije moguće je pokrenuti dohvaćanje lokacije u zadanom vremenskom intervalu.

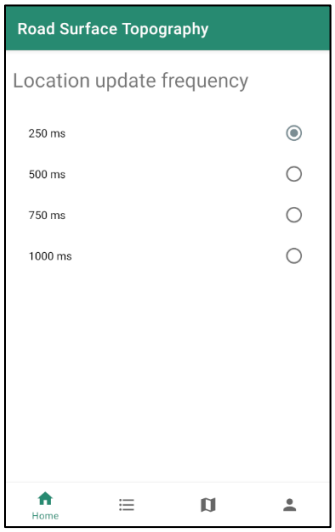
```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Programski kod 5.3. – Dopuštenja u *Manifest* datoteci

Postavljanjem vremenskog intervala definira se interval u kojemu će se primati ažuriranja o trenutnoj lokaciji mobilnog uređaja. U svrhu aplikacije u postavkama će se moći odabrati vrijednost željenog intervala u kojemu će se uzimati vrijednosti sa akcelerometra nakon detektirane nepravilnosti što će biti objašnjeno u slijedećem poglavlju. Moguće je odabrati slijedeće vrijednosti vremenskog intervala:

- 250 milisekundi
- 500 milisekundi
- 750 milisekundi
- 1000 milisekundi

Ekran za odabiranje vrijednosti vremenskog intervala prikazan je na slici 5.1.



Slika 5.1. – Ekran za odabir vremenskog intervala

Nakon što je definiran vremenski interval, potrebno je stvoriti zahtjev za lokaciju (engl. *Location Request*) u kojemu će se uz već definirani interval postaviti i prioritet dohvaćanja lokacije na vrijednost *PRIORITY_HIGH_ACCURACY* koji dohvaća najtočniju dostupnu lokaciju. Zahtjev za lokaciju prikazan je u programskom kodu 5.4.

```
private fun createLocationRequest() {
    locationRequest = LocationRequest.create().apply {
        interval = 500
        priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }
}
```

Programski kod 5.4. – Zahtjev za lokaciju

Uz zahtjev za lokaciju potrebno je napraviti i *callback* funkciju koja će se pozivati ovisno o zadanom vremenskom intervalu. Unutar funkcije uzimati će se trenutna geografska dužina (engl. *Latitude*) i geografska širina (engl. *Longitude*) koje će se spremati u varijable *latitude* i *longitude* kako bi se prilikom detekcije nepravilnosti na akcelerometru trenutna lokacija mogla spremiti u bazu podataka za daljnje korištenje. Programski kod funkcije prikazan je u programskom kodu 5.5.

```
private fun createLocationCallback() {
    locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult?) {
            locationResult ?: return
            for (location in locationResult.locations){
                latitude = location.latitude
                longitude = location.longitude
            }
        }
    }
}
```

Programski kod 5.5. – *Callback* funkcija za dohvaćanje lokacije

Nakon što su napravljeni zahtjev za lokaciju i *callback* funkcija za dohvaćanje lokacije potrebno je još započeti ažuriranja lokacije. Kako bi se izbjeglo preopterećivanje glavne niti, za ažuriranje trenutne lokacije stvoriti će se nova nit imena *location_thread* koja će biti zadužena samo za taj zadatak. Programski kod 5.6. prikazuje stvaranje niti i stvaranje zahtjeva za početak ažuriranja trenutne lokacije.

```
private fun startLocationUpdates() {
    val sensorThread = HandlerThread("location_thread")
    sensorThread.start()
    val looper = sensorThread.looper
    fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback,
    looper)
}
```

Programski kod 5.6. – Funkcija *startLocationUpdates()*

5.2 Izrada i implementacija algoritma

Kako bi se vrijednosti očitane sa implementiranih senzora mogle ispravno koristiti za mapiranje potrebno je izraditi algoritam koji će vršiti selekciju i klasifikaciju dobivenih podataka sa akcelerometra. Osnovna funkcionalnost aplikacije je stvaranje reljefne karte po kojoj će se vidjeti na kojim mjestima na prometnicama se nalaze nepravilnosti i koliko su one velike. Kako bi se to omogućilo prvo je potrebno je postaviti granicu na kojoj će se algoritam aktivirati. Pošto je vrijednost akceleracije u stanju mirovanja 9.81 potrebno je postaviti donju i gornju granicu iz razloga što će se prilikom prelaska preko nepravilnosti akceleracija povećati odnosno smanjiti u ovisnosti o tome da li je auto prešao preko udubine ili izbočine na cesti. Primjerice ukoliko auto

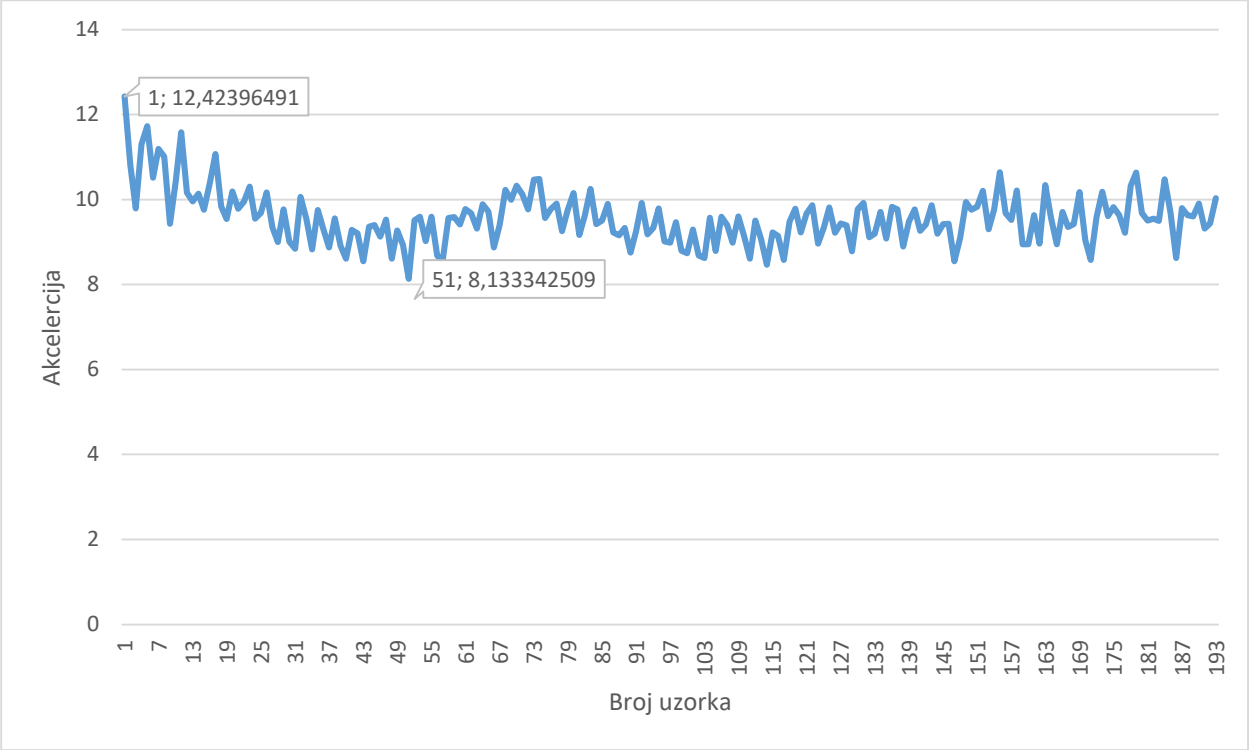
prelazi preko udubine akceleracija će se prvo naglo povećati do maksimalne vrijednosti, a zatim će se smanjivati do najniže vrijednosti u trenutku kada kotač automobila izađe iz udubine.

Nakon što se algoritam aktivira, odnosno nakon što se detektira nepravilnost na cesti sve očitane vrijednosti na akcelerometru spremaju se u listu sve dok se ne promjeni trenutna lokacija. Iz tog razloga omogućeno je postavljanje vremenskog intervala dobivanja trenutne lokacije uređaja kako bi se ostvarilo optimalno podešavanje ovisno o potrebama. Kada dođe do promjene lokacije lista sa prikupljenim podacima šalje se dalje na obradu u drugi dio algoritma. Implementacija prvog dijela algoritma prikazana je u programskom kodu 5.7.

```
override fun onSensorChanged(event: SensorEvent?) {
    val values = event?.values ?: floatArrayOf(0.0f, 0.0f, 0.0f)
    val value = Math.sqrt(values.map { x -> x*x }.sum().toDouble())
    updateAccelerationText(value)
    if(recordedData.isEmpty()){
        if (value > LIMIT_TOP || value < LIMIT_BOTTOM){
            currentLat = latitude
            currentLong = longitude
            recordedData.add(value)
        }
    }
    else{
        if(currentLat == latitude && currentLong == longitude){
            recordedData.add(value)
        }
        else{
            updateUi(recordedData)
            recordedData.clear()
        }
    }
}
```

Programski kod 5.7. – Implementacija algoritma

Prilikom prelaska preko nepravilnosti na cesti, dolazi do nagle promjene akceleracije po y-osi što stvara minimalnu i maksimalnu vršnu vrijednost. Kako bi se odredila veličina nepravilnosti drugi dio algoritma prilikom primanja liste sa očitanim podacima uzima minimalnu i maksimalnu vrijednost i računa njihovu razliku. Na grafikonu 5.1. prikazana je jedna zabilježena nepravilnost i njena maksimalna i minimalna vrijednost. Vidljivo je da je algoritam aktiviran skokom na maksimalnu vrijednost i nakon toga se postepeno smanjuje dok ne dosegne minimalnu vrijednost.



Grafikon 5.1. – Primjer zabilježene nepravilnosti

Što je razlika između vršnih vrijednosti veća, veća je i nepravilnost na cesti. Klasifikacija nepravilnosti vršiti će se po veličini razlike između vršnih vrijednosti, a određeno je 5 intervala unutar kojih će se vršiti klasifikacija:

- $X \in [\, 5, 8 >$ - malena nepravilnost
- $X \in [\, 8, 11 >$ - srednje - malena nepravilnost
- $X \in [\, 11, 14 >$ - srednja nepravilnost
- $X \in [\, 14, 17 >$ - srednje - velika nepravilnost
- $X \in [\, 17, \infty >$ - velika nepravilnost

Nakon što je odrađena klasifikacija po razlici vršnih vrijednosti podaci se spremaju u bazu podataka kako bi se mogli koristiti za mapiranje i ocrtavanje reljefa cesti. Uz spremanje u bazu korisniku se na ekranu pomoću *RecyclerView-a* prikazuje očitana vrijednost i trenutna lokacija. Programski kod algoritma i spremanje podataka prikazano je u programskom kodu 5.8.

```
private fun updateUi(recordedData: List<Double>) {
    val max = recordedData.max()!!.toDouble()
    val min = recordedData.min()!!.toDouble()
    val difference = max - min
    when{
        difference >= SMALL_BUMP_LOWER && difference < SMALL_BUMP_HIGHER
        -> saveData(difference, SMALL_BUMP)
        difference >= MED_SMALL_BUMP_LOWER && difference < MED_SMALL_BUMP_HIGHER
        -> saveData(difference, MEDIUM_SMALL_BUMP)
        difference >= MED_BUMP_LOWER && difference < MED_BUMP_HIGHER
        -> saveData(difference, MEDIUM_BUMP)
        difference >= MED_BIG_BUMP_LOWER && difference < MED_BIG_BUMP_HIGHER
        -> saveData(difference, MEDIUM_BIG_BUMP)
        difference >= BIG_BUMP_LOWER -> saveData(difference, BIG_BUMP)
    }
}

private fun saveData(difference: Double, bumpType: Int){
    adapter.addData(SensorData(difference, locationX = latitude, locationY =
    longitude, type = bumpType))
    presenter.saveSensorData(SensorDataDb(user = user, sensorValue = difference,
    locationX = latitude, locationY = longitude, bumpType = bumpType))
}
```

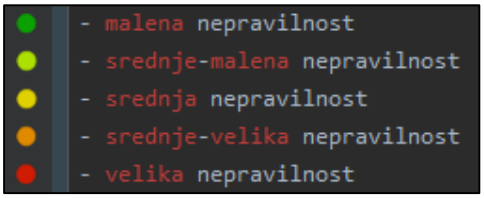
Programski kod 5.8. – Implementacija algoritma i spremanje podataka

5.3 Mapiranje podataka

Postoje razne metode mapiranja podataka i kroz povijest su se razvijali različiti načini najjasnijeg prikaza potrebnih elemenata. Reljef je kao kontinuirani trodimenzionalni element najteže prikazati i postoje razni postupci njegovog prikazivanja na kartama. Za potrebe rada odabrana je hipsometrijska metoda u kojoj se prikaz visinskih odnosa postiže bojama, a boje se odabiru po određenim principima. Primjerice najčešće primjenjivana skala boja u hipsometrijskoj metodi je [28]:

- Plavozelena - 0 - 100 metara
- Žutozelena - 100 - 200 metara
- Žuta - 200 - 500 metra
- Svjetlo smeđa - 500 - 1000 metara
- Crveno smeđa - 2000 - 4000 metara
- Smeđe crvena - iznad 4000

U kontekstu aplikacije ova metoda će se primjenjivati na način da će najmanja nepravilnost biti označena svijetlo zelenom, dok će najveća nepravilnost imati crvenu boju. Prikaz skale boja za mapiranje reljefa unutar aplikacije prikazana je na slici 5.2.



Slika 5.2. – Skala boja za mapiranje

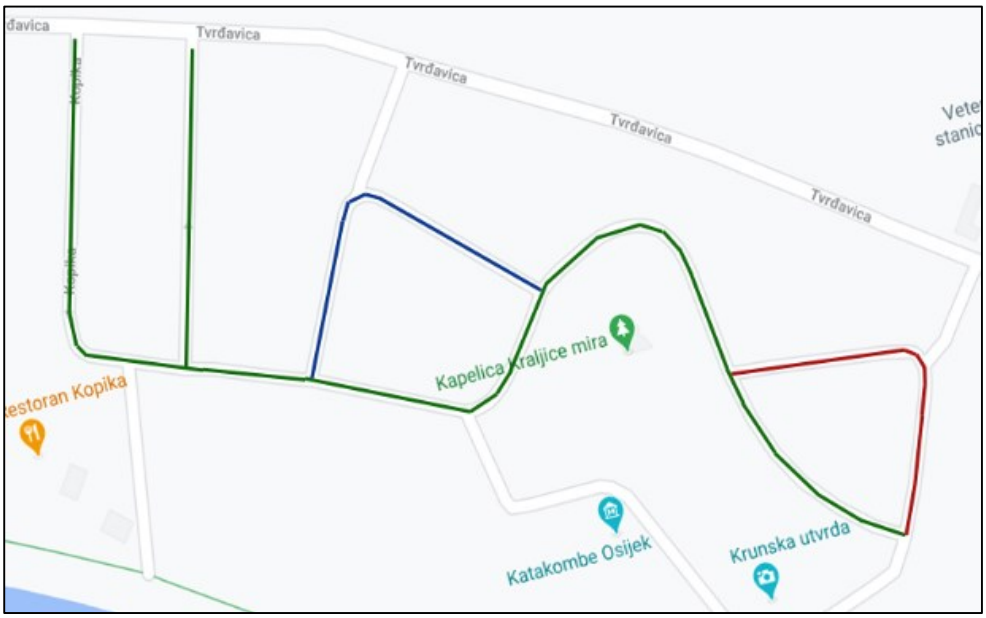
Koristeći zabilježene podatke na mapi će se prikazati markeri na spremljenim lokacijama. Kao što je spomenuto u prijašnjim poglavljima postoje tri mogućnosti prikaza podataka ovisno o korisnikovom odabiru. Programski kod 5.9. prikazuje dodavanje markera na kartu prilikom korisnikovog odabira. Nakon dodavanja markera a mapi že se jasno vidjeti reljef prometnica.

```
private fun showLocations(map: GoogleMap, locations: List<SensorDataDb>) {
    if(locations.isEmpty()) return
    setCamera(map, Pair(locations.last().locationX, locations.last().locationY))
    for (location in locations){
        when(location.bumpType){
            SMALL_BUMP -> addMarker(map, R.drawable.ic_small_bump_marker,
             location.locationX, location.locationY, location.sensorValue)
            MEDIUM_SMALL_BUMP -> addMarker(map,
            R.drawable.ic_medium_small_bump_marker, location.locationX,
            location.locationY, location.sensorValue)
            MEDIUM_BUMP -> addMarker(map, R.drawable.ic_medium_bump_marker,
            location.locationX, location.locationY, location.sensorValue)
            MEDIUM_BIG_BUMP -> addMarker(map,
            R.drawable.ic_medium_big_bump_marker, location.locationX,
            location.locationY, location.sensorValue)
            BIG_BUMP -> addMarker(map, R.drawable.ic_big_bump_marker,
            location.locationX, location.locationY, location.sensorValue)
        }
    }
}
```

Programski kod 5.9. – Dodavanje markera na kartu

6 KORIŠTENJE I TESTIRANJE APLIKACIJE

Nakon što je aplikacija izrađena potrebno ju je testirati u stvarnim uvjetima. Kako bi testiranje bilo potpuno potrebni su različiti tipovi podloge, i različite vrste nepravilnosti. Također potrebno je odabrati mjesto na kojemu se neće ometati promet. U tu svrhu odabran je maleni poligon koji će služiti kao testno okružje za aplikaciju. Slika 6.1. prikazuje kartu sa označenom putanjom kojom će se auto kretati prilikom testiranja.



Slika 6.1. – Karta poligona sa iscrtanom putanjom

Na slici je putanja označena u tri boje, a svaka boja predstavlja različiti tip podloge. Tipovi podloge po bojama su:

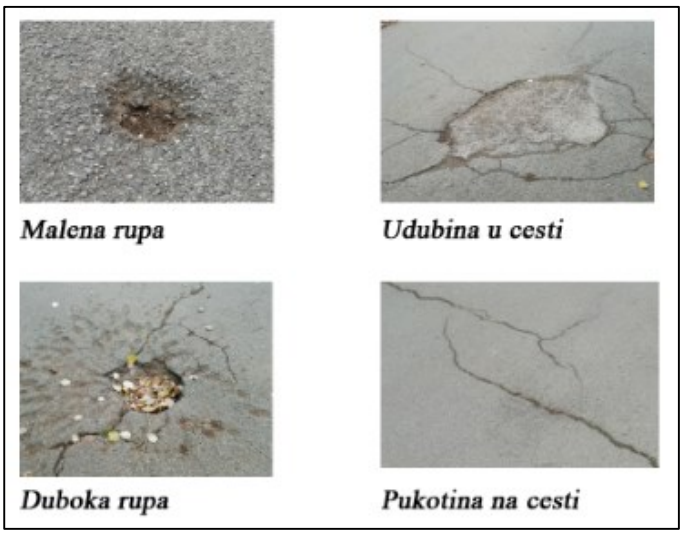
- Zelena - predstavlja normalnu cestovnu površinu
- Crvena - predstavlja hrapavu podlogu koja se sastoji od šupljih ploča
- Plava - predstavlja hrapavu podlogu koja se sastoji od punih ploča

Slika 6.2. prikazuje hrapave podloge po kojima će se testirati aplikacija.

Slika 6.2. – Hrapave podloge

Prije početka testiranja mobitel je postavljen u okomiti položaj koristeći držač za mobitele. Držač je postavljen ispod vjetrobranskog stakla i centriran između dva kotača automobila. Za testiranje je korišten automobil Ford Focus, 2008. godište te Huawei P-Smart mobilni uređaj.

Testiranjem je potrebno utvrditi postoji li razlika između korištenja različitih vremenskih intervala za dobivanje lokacije uređaja. Kako bi se to utvrdilo za svaki vremenski interval autom će se proći po poligonu te će se nakon tog usporediti dobiveni rezultati. Također potrebno je utvrditi da li algoritam dovoljno dobro detektira i klasificira nepravilnosti, te hoće li detektirati sve nepravilnosti. Primjeri nepravilnosti na poligonu prikazani su na slici 6.3.

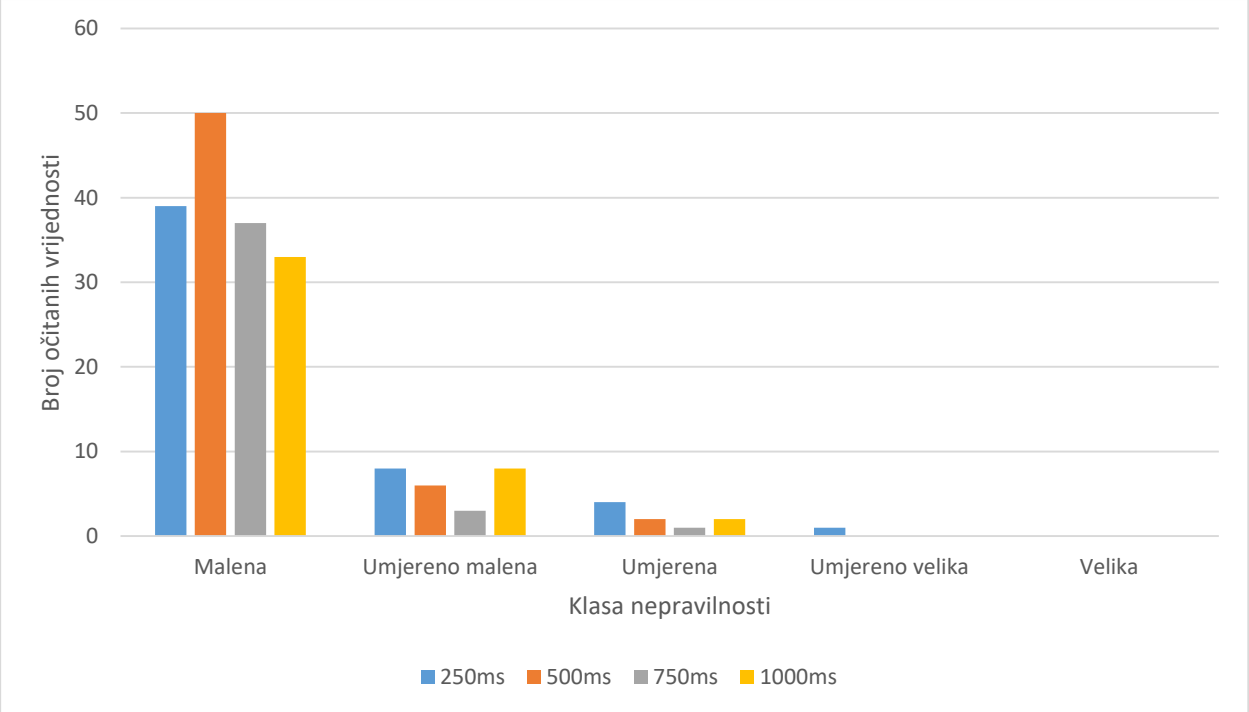


Slika 6.3. – Primjeri nepravilnosti

Testiranjem su dobiveni slijedeći rezultati:

- 250ms - 52 očitane nepravilnosti
- 500ms - 58 očitanih nepravilnosti
- 750ms - 41 očitana nepravilnost
- 1000ms - 43 očitanih nepravilnosti

Iz rezultata je vidljivo da je pri 750ms i 1000ms zabilježeno manje vrijednosti nego za 500ms i 250ms. Pretpostavka je bila da će se broj očitanja smanjiti sa duljinom intervala, no vidljivo je da je najviše zabilježenih vrijednosti na 500ms. Razlog tome je način prelaska kotača preko udubina. Naime vrlo je teško preći preko udubine na jednak način u svakom prolasku, pa se u nekim slučajevima udubina ne zabilježi na željeni način. Drugo obrazloženje je ne savršenstvo lokacijske usluge koja ne garantira dobivanje trenutne lokacije u zadanom intervalu nego kada je lokacija dostupna.



Grafikon 6.1. – Broj očitanih vrijednosti po klasi nepravilnosti

Grafikon 6.1. prikazuje broj očitanih vrijednosti po klasi nepravilnosti. Vidljivo je da je najviše zabilježenih malenih nepravilnosti dok nije zabilježena niti jedna velika nepravilnost. U malene nepravilnosti spadaju malene rupe i pukotine na cesti koje su prikazane na slici 6.3. Isto tako u jednom prelasku zabilježena je umjereno velika nepravilnost, dok u ostala tri prelaska nisu

što znači da su nepravilnosti bile na granici između umjerene i umjereno velike, ili zbog malog promjera rupe postoji mogućnost da je pređeno preko ruba udubine pa nije zabilježena kao veća nepravilnost. Umjereno velika nepravilnost detektirana je na dubokoj rupi prikazanoj na slici 6.3.

Slika 6.4. – Dobivena karta za 250ms i 500ms

Slika 6.4 prikazuje mapirane podatke za testiranje pri 250ms i 500ms. Odmah je vidljivo da se najviše nepravilnosti nalazi na dva dijela poligona koji su prije testa okarakterizirani kao hrapave podloge. Najviše očitanja nalazi se na hrapavoj podlozi prekrivenoj šupljim pločama. Na normalnom dijelu ceste nema toliko puno zabilježenih vrijednosti, a vidljivo je da se sve nalaze na približno jednakim lokacijama u oba dva slučaja što znači da je algoritam uspješno detektirao nepravilnosti. Također je vidljivo da se neki markeri nalaze pokraj ceste iz razloga što lokacijska usluga najviše preciznosti garantira preciznost očitanja unutar nekoliko metara, pa se ponekad zabilježi lokacija pokraj ceste. Osim toga na dijelu poligona prekrivenim sa ravnim pločama prilikom testa voženo je po drugom dijelu ceste koji nije zabilježen na *Google* kartama pa su markeri postavljeni pokraj kao što je vidljivo na dobivenom prikazu.

Slika 6.5. – Dobivena karta za 750ms i 1000ms

Slika 6.5 prikazuje mapirane podatke za testiranje pri 750ms i 1000ms. Vidljivo je da ima manje zabilježenih vrijednosti, ali se markeri nalaze na sličnim mjestima kao i na prethodne dvije slike. Također na hrapavim podlogama ponovno je zabilježeno najviše vrijednosti. Analizirajući mapirane podatke i mjesta gdje se nalaze nepravilnosti na poligonu zaključeno je da je pri vrijednosti od 750ms i 1000ms napravljen najvjerniji prikaz. Također se može zaključiti da je algoritam detektirao gotovo sve nepravilnosti koje postoje na poligonu osim nekoliko pukotina na cesti koje su premalene da bi aktivirale graničnu vrijednost algoritma. Ovo bi se moglo popraviti dodatnim postavljanjem granica detekcije i intervala za klasifikaciju.

7 ZAKLJUČAK

Detekcija nepravilnosti na cesti i njihovo mapiranje relativno je neistraženo područje. Razne udubine, izbočine i pukotine na cesti jedan su od faktora koji može dovesti do prometnih nesreća, ili oštećenja na vozilu. Kako bi se to spriječilo potrebno je održavanje velike površine cestovnih prometnica što je veoma skupo i oduzima enormnu količinu vremena. Također manualni pregled zahtjeva radnu snagu koja bi se mogla upotrijebiti na efikasniji način.

Uz sveprisutne pametne mobitele koje sadrže potrebne senzore kao što su akcelerometar i GPS mogućnosti za rješavanje ove problematike su velike, ali nedovoljno istražene. U radu su prikazane aplikacije koje su se bavile ovom tematikom, a dobiveni su obećavajući rezultati. Izrađena aplikacija imala je cilj istražiti mogućnost i efikasnost detekcije i mapiranja reljefa površine kolnika. Koristeći spomenuti akcelerometar i GPS izrađen je algoritam za detekciju koji prilikom prelaska preko nepravilnosti detektira očitane vrijednosti, te koristeći razliku između maksimalne i minimalne vrijednosti klasificira veličinu nepravilnosti. Nakon detekcije i klasifikacije podaci se mapiraju koristeći hipsometrijsku metodu mapiranja reljefa koja koristeći skalu boja omogućuje precizno ocrtavanje reljefa. U svrhu testiranja aplikacije obavljeno je istraživanje koje je pokazalo da moguća vrlo efikasna detekcija nepravilnosti i njihovo mapiranje. Utvrđeno je da algoritam detektira većinu nepravilnosti i hrapavih površina koje su bile prisutne na testnom poligonu. Nije detektirano nekolicina pukotina na cesti koje su premale da bi aktivirale granice algoritma, koje bi se po potrebi mogle podesiti.

Aplikacija također sadrži prijavu i registraciju korisnika koji nakon uspješne autentifikacije imaju mogućnost pokretanja detekcije. Prilikom vožnje korisnici imaju uvid u trenutnu vrijednost akceleracije i prikaz očitanih vrijednosti. Nakon što su podaci detektirani korisnik im mogućnost upravljanja s njima. Moguće je brisanje, prikaz na mapi te postavljanje podataka u online bazu podataka. Na karti je moguće prikazati sve korisnikove podatke iz lokalne i online baze podataka, te podatke od svih korisnika.

U radu su opisane korištene tehnologije u koje spadaju Android platforma, Android studio, Kotlin programski jezik, Firebase servis i Koin. Nakon opisanih tehnologija prikazani su maketa aplikacije, životni krugovi, dijagram toka i korištena MVP arhitektura. Također je prikazano korisničko sučelje i rad sa podacima. U zadnja dva poglavlja opisani su senzori te njihovo korištenje unutar opisanog algoritma. Rad završava sa testiranjem i analizom rezultata.

LITERATURA

[1] RoadBounce, RoadBounce application, dopstupno na: https://www.roadbounce.com/ [srpanj 2020.]

[2] SmartRoadSense, SmartRoadSense application, dopstupno na: https://smartroadsense.it/ [srpanj 2020.]

[3] A. Giacomo, L. C. Klopfenstein, S. Delpriori, M. Dromedari, G. Luchetti, B. D. Paolini, A. Seraghiti, E. Lattanzi, V. Freschi, A. Carini, A. Bogliolo, SmartRoadSense: Collaborative Road Surface Condition Monitoring, UBICOMM 2014, The Eighth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, stranice: 210-215, godina izdavanja 2014.

[4] K. Zeng, J. Shen, H. Huang, M. Wan, J. Shi, Assessing and Mapping of Road Surface Roughness based on GPS and Accelerometer Sensors on Bicycle-Mounted Smartphones, Sensors 2018, broj časopisa: 3, svezak: 18, broj članka: 914, ožujak 2018.

[5] V. Astarita, M. V. Caruso, G. Danieli, D. C. Festa, V. Giofré, T. Iuele, R. Vaiana, A Mobile Application for Road Surface Quality Control: UNIquALroad, PROCEDIA: SOCIAL & BEHAVIORAL SCIENCES, svezak: 54, stranice: 1135-1144, listopad 2012.

[6] Wikipedia, Android OS, dopstupno na: https://hr.wikipedia.org/wiki/Android_(operacijski_sustav) [srpanj 2020.]

[7] Wikipedia, SmartphoneSales, dopstupno na: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#/media/File:World_Wide_Smartphone_Sales.png [srpanj 2020.]

[8] M. Gargenta, Learning Android, O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, 2011.

[9] Android, Android Architecture, dopstupno na: https://source.android.com/devices/architecture [srpanj 2020.]

[10] Android Developer, Platform Architecture, dopstupno na: https://developer.android.com/guide/platform [srpanj 2020.]

[11] Android Developer,Sensors Overview, dopstupno na: https://developer.android.com/guide/topics/sensors/sensors_overview [srpanj 2020.]

[12] G. Milette, A. Stroud, Professional Android Sensor Programming, John Wiley & Sons, 10475 Crosspoint Boulevard Indianapolis, 2012.

[13] Mathworks, Android accelerometer, dopstupno na: https://nl.mathworks.com/help/supportpkg/android/ref/simulinkandroidsupportpackage_galaxys4_accelerometer.png [srpanj 2020.]

[14] Android Developer, Android Studio, dopstupno na: https://developer.android.com/studio/intro [srpanj 2020.]

[15] Wikipedia, Kotlin, dopstupno na: https://en.wikipedia.org/wiki/Kotlin_(programming_language) [srpanj 2020.]

[16] I. Galata, J. Howard, R. Lucas, E. Shapiro, Kotlin Apreantice First Edition, Razeware LLC, 2018.

[17] Kotlinlang, Kotlin, dopstupno na: https://kotlinlang.org/docs/reference/android-overview.html [srpanj 2020.]

[18] Firebase, Firebase authentication, dopstupno na: https://firebase.google.com/docs/auth [srpanj 2020.]

[19] Firebase, Cloud Firestore, dopstupno na: https://firebase.google.com/docs/firestore [srpanj 2020.]

[20] Android Developer, Dependency Injection, dopstupno na: https://developer.android.com/training/dependency-injection [srpanj 2020.]

[21] Insert-koin, Koin Dependency Injection, dopstupno na: https://insert-koin.io/ [srpanj 2020.]

[22] Fluid UI, Fluid UI, dopstupno na: https://www.fluidui.com/ [srpanj 2020.]

[23] Wikipedia, MVP, dopstupno na: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter [srpanj 2020.]

[24] Android Developer, Activity, dopstupno na: https://developer.android.com/reference/android/app/Activity [listopad 2020]

[25] Javatpoint, Android Fragments, dopstupno na: https://www.javatpoint.com/android-fragments, listopad 2020.

[26] Android4dev, Room Persistence Library, dopstupno na: https://www.android4dev.com/android-architecture-components-room-persistence-library/ [studeni 2020]

[27] Android Developer, Motion Sensors, dopstupno na: https://developer.android.com/guide/topics/sensors/sensors_motion [studeni 2020]

[28] PMF, Prikazivanje reljefa na topografskim kartama, dopstupno na: https://www.pmf.unizg.hr/_download/repository/KOG_9_2020.pdf?fbclid=IwAR1tFFKB6lzwHuJxvnfBN0DT_VXceYbyGu6EADxjNEZXMJx0Acp_nT_g0_U [studeni 2020]

SAŽETAK

Diplomski rad opisuje i istražuje probleme detekcije i mapiranja nepravilnosti na cestama koristeći dostupne senzore na Android pametnim uređajima. Problem detekcije nepravilnosti je precizno očitavanje vrijednost sa akcelerometra i klasifikacija očitanih podataka kako bi se mogla stvoriti reljefna karta prometnica. U svrhu rješavanja tog problema razvijena je aplikacija za detekciju i mapiranje reljefa površine kolnika. Aplikacija koristi algoritam koji tokom vožnje detektira nepravilnosti i na temelju razlike između maksimalne i minimalne vrijednosti očitanih podataka klasificira veličinu nepravilnosti. Koristeći GPS uzima se lokacija koja se uz klasu nepravilnosti koristi za stvaranje zornog prikaza reljefa prometnica. Uz algoritam aplikacija omogućuje prijavu i registraciju korisnika, namještanje postavki aplikacije, spremanje podataka u lokalnu i online bazu te različite prikaze podataka na karti. U radu je opisano i testiranje aplikacije u stvarnim uvjetima, gdje je aplikacija podvrgnuta raznim nepravilnostima i tipovima cesta. Testiranje je pokazalo da aplikacija uspješno detektira većinu nepravilnosti.

Ključne riječi: akcelerometar, detekcija nepravilnosti ceste, GPS, mapiranje reljefa, mobilna aplikacija, pametni mobilni uređaji

ABSTRACT

Detection and mapping of road surface relief

Master thesis describes and explores the problems of detecting and mapping irregularity of road surfaces using available sensors on Android smartphones. Problem of detecting irregularities is precise reading of accelerometer values and classification of read data in purpose of creating terrain map of road surfaces. In purpose of solving this problem application for detection and mapping of road surface terrain was created. Application is using an algorithm that detects irregularities while driving and classifies said irregularities on the base of difference between maximum and minimum values from read data. User location is determined using GPS, which alongside classified irregularities is used for creating visual representation of road surface terrain. Alongside the algorithm application enables user login and registration, adjusting of application settings, data storage in local and online database and different options of showing recorded data on the map. Thesis also describes application testing in real life conditions, where application was subjected to different kind of irregularities and road surfaces. Testing showed that application can detect most of irregularities.

Key words: accelerometer, GPS, mobile application, road irregularity detection, smartphones, terrain mapping

ŽIVOTOPIS

Matej Milić rođen je 12. siječnja 1994. godine u Osijeku. Od 2000. do 2008. pohađa Osnovnu školu Ljudevita Gaja u Osijeku. Godine 2008. upisuje Elektrotehničku i prometnu školu Osijek koju završava 2012. obranom završnog rada i polaganjem ispita državne mature. Godine 2014. upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija na Sveučilištu Josipa Juraja Strossmayera u Osijeku na stručni studij informatike koji završava 2017. Nakon završene razlikovne godine, 2018. upisuje diplomski studij računarstva, smjer programsko inženjerstvo.

Matej Milić