Pronalaženje točaka interesa uz pomoć proširene stvarnosti

Matković, Ivan

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij računarstva

PRONALAŽENJE TOČAKA INTERESA UZ POMOĆ PROŠIRENE STVARNOSTI

Diplomski rad

Ivan Matković

Osijek, 2017.

Sadržaj

1. UVOD

Razvojem novih tehnologija otvaraju se vrata mogućnostima za nove aplikacije. Samim time što se sukladno tehnologijama razvijaju i alati za korištenje istih, trenutno je puno jednostavnije i atraktivnije izrađivati, eksperimentirati i testirati s novim idejama i tehnologijama. Razvojem tehnologije virtualne stvarnosti (engl. augmented reality) stvaraju se nove mogućnosti za aplikacije kakve prije nisu bile moguće. Mnogo je grana industrije koje mogu iskoristiti tehnologiju proširene stvarnosti. Učenicima bi se za edukaciju moglo puno vjernije prikazati događaji i modeli. Industrija zabave bi također vrlo lako iskoristila prednosti tehnologije, kao što je viđeno na primjeru igre Pokemon GO. Cilj ovog rada je napraviti iOS aplikaciju koja će spojiti prednosti novog modernog programskog jezika Swift i nove tehnologije proširene stvarnosti. Koristeći globalni pozicijski sustav mobilnog uređaja i mrežne povezanosti, cilj je dobiti trenutnu lokaciju i točke interesa koristeći aplikacijsko programsko sučelje. Pomoću tehnologije proširene stvarnosti cilj je prikazati dobivene točke interesa na zaslon mobilnog uređaja.

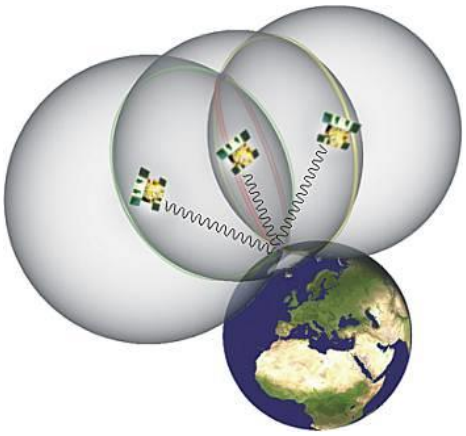
Drugo poglavlje opisuje problematiku pozicioniranja i orijentacije. Zatim opširnije opisuje ideju u radu i prikazuje koncept idejnog rješenja. Treće poglavlje opisuje tehnologije koje se koriste, alate, programske jezike, korištene API-e i pomoćne biblioteke. Četvrto poglavlje prikazuje programsko rješenje i načine na koje su korištene tehnologije iz drugog poglavlja i alati iz trećeg. Opisuje način izrade aplikacije i prikazuje programski kod te ga opisuje. Peto poglavlje opisuje rezultat rada, aplikaciju. Opisana je uporaba aplikacije od strane korisnika s pripadajućim grafičkim prikazima zaslona aplikacije. Opisani su automatski testovi, ručni testovi i rukovanje greškama. Aplikacija je analizirana i testirana.

2. KORIŠTENJE PROŠIRENE STVARNOSTI ZA ORIJENTACIJU UZ POMOĆ MOBILNIH TEHNOLOGIJA

Ovo poglavlje opisuje korištenje proširene stvarnosti uz pomoć mobilnih tehnologija. Također, opisuje problematiku orijentacije kod klasičnih navigacija i glavnu ideju rada.

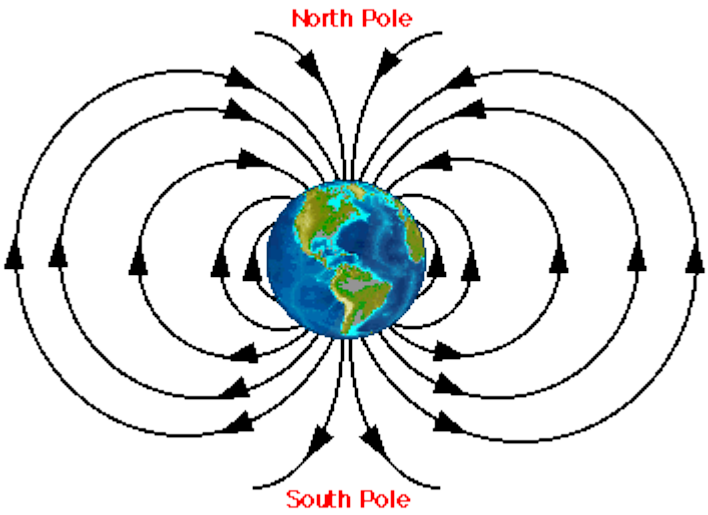
2.1. Uvod u problematiku

Od davnina čovjek je imao potrebu da se pozicionira u prostoru oko sebe. Tada bi znao gdje i kako pronaći resurse, ali također gdje se vratiti u sigurnost svoga doma. Čak i nakon tisuće godina čovjek i dalje ima istu potrebu, no uz stare razloge kao što su resursi i dom, javili su se i novi. Postoji nekoliko bitnih tehnologija koje su olakšale čovjeku pozicioniranje u prostoru. Globalni pozicijski sustav ili skraćeno GPS je najpopularniji sustav za trenutno pozicioniranje u prostoru. Prema [1], GPS je mreža satelita koja kontinuirano odašilje kodirane informacije, s pomoću kojih je omogućeno precizno određivanje položaja na Zemlji. GPS je prvotno razvijen u vojne svrhe no američki predsjednik Ronald Reagan objavio je direktivu 1983. godine gdje je učinio GPS dostupnim za civilnu uporabu i javno dobro. GPS je mreža od 30-tak satelita koji kruže zemaljskom orbitom na visini od 20 000 km i u svakom trenu su „vidljiva“ barem četiri satelita sa svake zemaljske točke. Kada se osoba pokuša pozicionirati, GPS uređaj započinje primanje signala koji putuju brzinom svjetlosti iz „vidljivih“ satelita te uređaj izračunava koliko su daleko sateliti s obzirom na vrijeme primanja poruke. Nakon što se dobije informacija od barem tri satelita GPS izračunava gdje se osoba nalazi s obzirom na prijesjek radijusa od satelita (Sl.2.1). Što više satelita je „vidljivo“, to je pozicija točnija. Danas svaki mobilni uređaj u sebi ima GPS prijamnik.



Sl. 2.1. *Princip rada GPS sustava* [22]

Zahvaljujuci GPS prijamniku u iPhone uređaju pronalaženje točaka interesa je moguće, i sama izvedba ovog rada. Jer kako bi se pronašle točke interesa potrebno je saznati referentnu lokaciju odnosno lokaciju korisnika.No, samo pozicioniranje nije dovoljno za navigaciju. Potrebna je i orijentacija kako bi se odredio smjer. Riječ orijentacija dolazi od francuske riječi „orienter“ što znači smjer prema istoku ili izlazak sunca, no sama riječ orijentacija znači snalaženje u prostoru odnosno određivanje mjesta na površini zemlje na kojem se nalazimo. U povijesti su zemljovidi i karte moreplovaca bili usmjereni prema istoku. Postoji nekoliko metoda zemljopisne orijentacije kao što su orijentacija pomoću Sunca i sata, orijentacija pomoću zvijezda, orijentacija pomoću mjeseca i druge metode koje su dosta nepouzdane kao što je rast mahovine na sjevernoj strani ili činjenica da su mravinjaci obično s južne strane. Najpopularnija metoda određivanja orijentacije je pomoću kompasa. Kompas je naprava koja služi za određivanje smjera na zemljinoj površini u odnosu na sjeverni i južni pol.Sastoji se od malene magnetske igle koja pokazuje sjever. Razlog pokazivanja sjevera je što Zemlja ima magnetska polje (Sl. 2.2) koje utječe na drugi magnet, u ovom slučaju iglu kompasa. Trenutno velika većina mobilnih uređaja koristi senzor magnetometar kako bi odredila orijentaciju uređaja.



Sl. 2.2. *Magnetski polovi Zemlje* [*23*]

Tehnologije za pozicioniranje (GPS) i orijentaciju (magnetometar) omogućile su rad aplikacija za navigaciju. No, pogledom na kartu korisnik dobije uvid u dvodimenzionalnom prostoru gdje se on nalazi i u kojem smjeru je orijentiran. Međutim, korištenjem proširene stvarnosti i drugih mobilnih senzora kao što su kamera i žiroskop moguće je korisniku pokazati na trodimenzionalnom prostoru točke interesa koje ga zanimaju na prirodniji način kako se ne bi morao orijentirati preko dvodimenzionalne karte koje prikazuje zaslon mobitela i tako bi riješili riješiti problem orijentacije na mapi koristeći relativno nove tehnologije proširene stvarnosti. Poglavlje 3.3 detaljnije opisuje tehnologiju proširene stvarnosti korištene u izvedbi ovog rada.

2.2. Ideja u radu

Glavna ideja diplomskog rada je napraviti aplikaciju za pronalaženje točaka interesa uz pomoć proširene stvarnosti koristeći programski jezik Swift za iOS platformu koristeći metodologiju agilnog razvoja programske podrške Kanban. Aplikacija bi trebala korisnika prikazati na 2D karti s točkama interesa oko njega. Nakon toga, korisnik je u mogućnosti upaliti kameru i pomoću proširene stvarnosti imati pregled svih točaka interesa na zaslonu mobitela oko njega u 3D prostoru, odnosno u stvarnom svijetu.

2.2.1. Postojeća rješenja

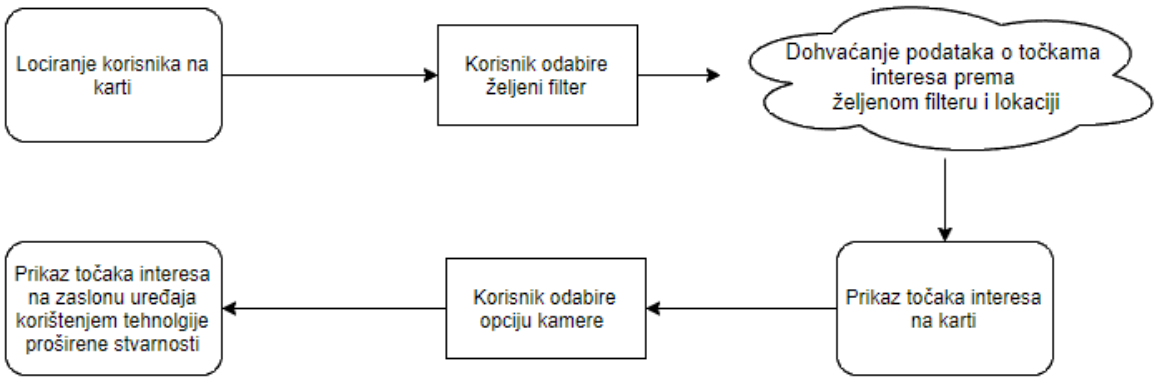
Kako tehnologija proširene stvarnosti sazrijeva, sve više je aplikacija koje koriste mogućnosti tih tehnologija no trenutno ne postoji aplikacija koja je spojila tehnologiju proširene stvarnosti s tehnologijom navigacije. Tehnologija proširene stvarnosti doživjela je masovni uspjeh s igrom za mobilne uređaje Pokemon Go. Prema [2], u prvom tjednu igra je brojila preko 15 milijuna preuzimanja dok je do kraja mjeseca taj broj dostigao 100 milijuna preuzimanja. Naravno, za uspjeh igre Pokemon Go uglavnom je zaslužan brand Pokemon, no igra je ponudila i vrlo zabavan i atraktivan način korištenja tehnologije proširene stvarnosti što je dodatno doprinijelo njenoj popularnosti.

Također, Apple je u lipnju 2016 godine objavio svoj inovativni programski okvir za izradu aplikacija (eng. *framework*) s tehnologijom proširene stvarnosti. Nedugo zatim Google je ponudio svojim razvojnim inženjerima slično rješenje namijenjeno za Android platformu. Razvoj sklopovlja u mobilnim uređajima omogućio je razvoj kompleksnijih programskih rješenja te će zasigurno upotreba tehnologije proširene stvarnosti imati puno širu i popularniju primjenu u programskim rješenjima. Odabir tehnologije proširene stvarnosti za ovaj rad inspirirala je atraktivnost tehnologije, dostupnost sklopovlja i činjenica da je tehnologija relativno nova.

2.3. Prikaz koncepta idejnog rješenja

Nakon što korisnik uđe u aplikaciju, treba mu se prikazati karta. Preko GPS-a mobilnog uređaja, korisnik dobiva informaciju o trenutnoj lokaciji na kojoj se nalazi. Zatim se na aplikacijsko programsko sučelje (engl. Application Programming Interface, API) šalje trenutna lokacija i radijus koji vraća općenite točke interesa oko korisnika i prikazuje ih na karti. Korisnik ima opciju specificirati filtar prema tipu točke interesa te se šalje novi zahtjev na API gdje se postojeće točke interesa zamjenjuju novima. Ako korisnik želi koristiti tehnologiju proširene stvarnosti, može pritisnuti na gumb koji označava kameru te se zatim podaci o točkama interesa

prikazuju koristeći tehnologiju proširene stvarnosti. Koncept idejnog rješenja prikazan je na slici 2.3.



Sl. 2.3. *Koncept idejnog rješenja*

Kako bi se točke interesa prikazale na zaslonu uređaja, potrebno je dobiti podatke o nagibu uređaja, visini i orijentaciji. Nakon što se podaci dobiju, obrađuju se te se određuje na kojem mjestu je potrebno prikazati karticu o informacijama točaka interesa. Također, korisnik ima mogućnost klikom na karticu koja prikazuje određenu točku interesa dobiti detaljnije informacije kao što su adresa, broj telefona i mrežna stranica.

3. PRIJEDLOG PROGRAMSKE ARHITEKTURE I KORIŠTENI PROGRAMSKI ALATI

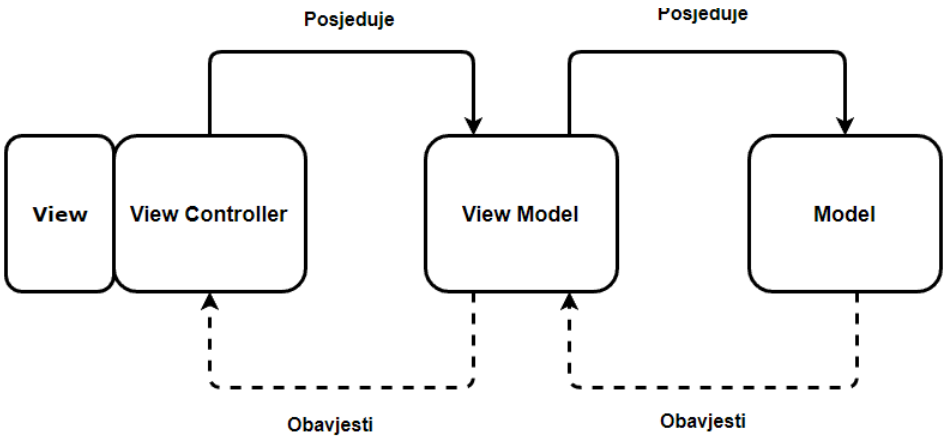
U ovom poglavlju opisana je arhitektura programskog rješenja MVVM. Opisan je agilan pristup izradi rada. Također, opisana je glavna tehnologija korištena u radu, alati, programski jezik i sučelja za programiranje aplikacija, API.

3.1. Arhitektura programskog rješenja

Kako bi se izradilo programsko rješenje korištena je programska arhitektura Model-View-ViewModel ili skraćeno MVVM.

3.1.1. MVVM arhitektura

Prema [3], korištenje programskih arhitektura aplikacija (eng. *design patterns*) pri razvoju programa postoji kako bi se razvoj učinio manje kompleksan, programski kod bio pogodniji za održavanje i za testiranje. Postoje mnoge programske arhitekture za različite platforme i niti jedna nije najbolja za sve projekte, već se odabire s obzirom na zahtjeve projekta. Model-View-ViewModel je programska arhitektura koja potječe iz Microsofta, no nameće se kao najpopularnija arhitektura za razvoj iOS aplikacija. Sastoji se od tri glavne komponente: View, ViewModela i Modela. Iako zvuči komplicirano, programska arhitektura MVVM je vrlo slična vrlo popularnoj MVC odnosno Model-View-Controller arhitekturi. Slika 3.1 prikazuje MVVM arhitekturu.



Sl. 3.1. *Prikaz MVVM arhitekture*

Prema [4], komponenta Model sadrži reprezentaciju podataka s kojim program radi. ViewModel ima instancu, odnosno posjeduje Model te na njegove promjene reagira, jer ga Model obavijesti kada se nešto promijeni od podataka. Cilj ViewModela je da sadrži svu logiku za manipulaciju s podacima i da ih pripremi za prikaz odnosno predaju ViewControleru koji ima njegovu instancu.

ViewController u iOS sustavu je usko povezan s View-om te ga se nastoji što manje opteretiti manipulacijom podataka. ViewController reagira i upravlja podacima koje mu ViewModel priredi. Glavne prednosti programske arhitekture MVVM su ispitljivost, jer se može testirati poslovna logika koja je smještena u ViewModel i koja je potpuno odvojena od sučelja (View-a). Zatim je tu i jednostavnost primjene, jer u prosjeku MVVM ima puno manje koda od ModelViewController arhitekture.

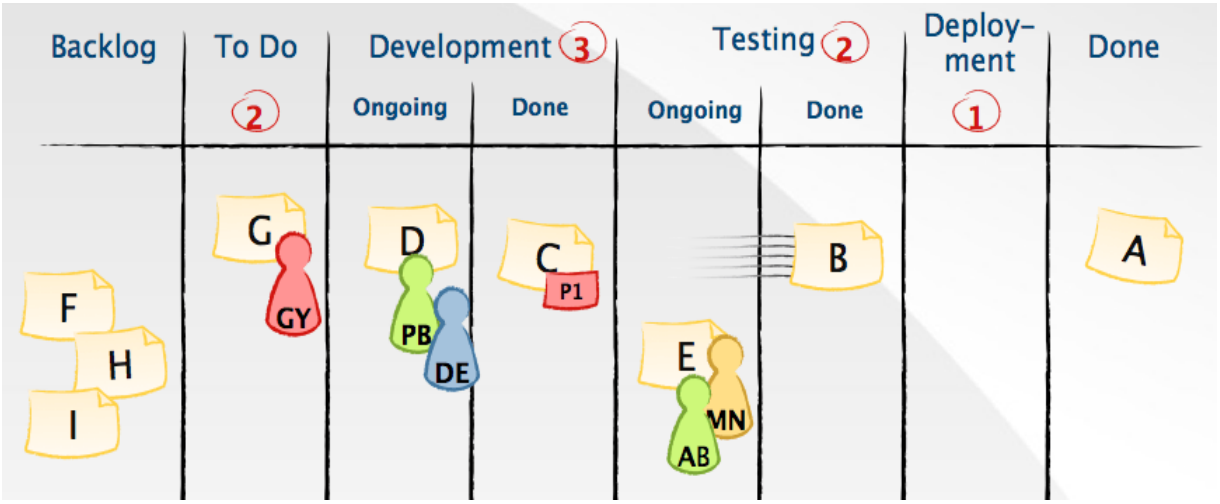
3.2. Agilni pristup izradi

Prema [5], agilne metode imaju korijene u japanskoj poslovnoj filozofiji (kanban) nastaloj prije više od 20 godina, no danas su postale vrlo popularne u tvrtkama za razvoj programskih rješenja. Za razliku od modela slapa (engl. waterfall) u kojem se unaprijed odrede specifikacije i zadaci, u agilnim metodologijama se rješavanju zadataka pristupa agilno te su moguće promjene i korekcije za vrijeme izrade projekta.

3.2.1. Agilna metoda Kanban

Kanban je danas jedna od vrlo popularnih i često korištenih metodologija agilnog razvoja programske podrške, a glavni cilj je fleksibilnost i kontinuirane promjene tijekom razvoja. Kroz Kanban načela koji nisu kompliciran, razvojni tim je u mogućnost lakše upravljati svojim resursima. Kanban se zasniva na ploči koja je podijeljena u šest vertikalnih sekcija no broj sekcija nije striktan i može se mijenjati i prilagođavati timu. Svaka sekcija predstavlja stanje jednog razvojnog zadatka, a ideja je da zadatak kako se izvršava tako prelazi iz jedne sekcije u drugu. Prema slici 3.2, tipična Kanban ploča sadrži sljedeće sekcije:

- **Backlog –** S backlogom upravlja naručitelj projekta (engl. project owner). On definira što želi ostvariti, odnosno zadatke.
- **Ready –** Nakon dogovora s razvojnim timom oko detalja, zadaci su spremni za rješavanje.
- **Coding –** Zadaci koji se trenutno rješavaju.
- **Testing -** Zadaci koji su završeni i i trenutno se testiraju.
- **Approval –** Zadaci koji su uspješno testirani i čekaju na odobrenje za puštanje u produkciju.
- **Done –** Završeni zadaci koji su sve prijašnje sekcije prošli uspješno. Predstavlja kraj svakog zadatka.



Sl. 3.2. *Prikaz Kanban ploče [23]*

Ideja Kanbana je da se ograniči broj zadataka i da se smanji višezadaćnost (engl.multitasking), jer se tako puno lakše razvojni tim fokusira na trenutni zadatak bez razmišljanja o ostalima. Također, kako postoji ograničen broj zadatka u svakoj sekciji, može se dogoditi da dođe do limita sekcije i tada zadaci iz prethodnih sekcija ne prelaze dalje već se čeka da se otkloni problem. To je vrlo pozitivno, jer je potrebno učiniti dodatan napor oko blokirajućih zadataka kako bi se projekt nastavio, u suprotnom bi se nastavio rad na drugom mjestu a projekt bi se sve više blokirao.

Upotrebom Kanban metodologije može se doći do vrlo bitnih statističkih podataka i spoznaja koje se mogu iskoristiti kako bi se unaprijedila učinkovitost. Praćenje vremena potrebnog za svaki zadatak i praćenje vremena zadataka u svakoj od sekcija može se vidjeti gdje je najproblematičnije područje i što usporava rad. S tim informacijama može se prilagoditi ograničenje za svaku sekciju čime će se razvojni proces izmijeniti, a vrijeme izrade poboljšati.

3.3. Korištene tehnologije

Tehnologija je razvoj i primjena alata, strojeva, materijala i postupaka za izradu nekog proizvoda. Za ostvarivanje programskog riješenja korištena je tehnologija proširene stvarnosti, mobilni telefon iPhone i operacijski sustav iOS. Alati i programski jezik korišteni za izradu su usko vezani uz tvrtku Apple odnosno uz iOS i MacOS sustav.

3.3.1. Proširena stvarnost

Prema [6], proširena stvarnost (engl. *augmented reality, AR)* je rastuće područje istraživanja i primjene na području virtualne stvarnosti. Ljudi percipiraju veliku količinu podataka iz stvarnog okoliša i reprodukcija takvog stvarnog okoliša u digitalnom obliku vrlo često je nemoguća.

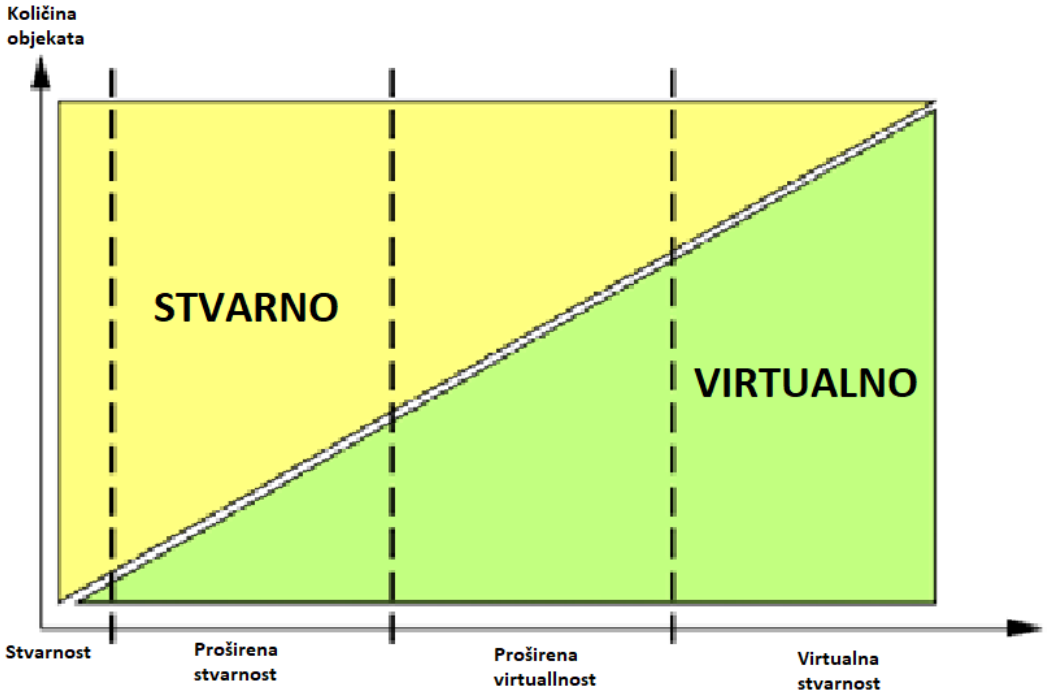
Gledajući na današnje virtualne okoline kao što su simulatori letenja, zbog velike potrebe za procesorskom i grafičkom moći vrlo su nedostupne i skupe. Proširena stvarnost generira zbirni prikaz za korisnika kao što je kombinacija krajolika kojeg korisnik zapravo vidi i komponenti koje se digitalno kreiraju. Takvim načinom, korisniku se povećava percepcija i razumijevanje stvarnog svijeta, naravno u ovisnosti o aplikaciji koja generira prikaz. Cilj proširene stvarnosti je da korisnik više ne može razlikovati objekte iz stvarnog svijeta i objekte dodane virtualnim putem. Proširena stvarnost se odnosi samo na vizualni kanal korisnika. Proširena stvarnost je toliko popularna da je Apple, prema [7], na Worldwide Developers Conference 2017. godine predstavio ARKit kako bi omogućio razvojnim inženjerima da što lakše rade aplikacije s navedenom tehnologijom.

3.3.2. Virtualna stvarnost

Povećavanjem snage procesorske i grafičke moći došli smo do točke gdje je često teško prepoznati radi li se o virtualnoj ili stvarnoj slici. No, računalno generirane slike koje često vidimo u filmovima, igrama i ostalim medijima su odvojene od fizičkog okruženja stvarnog svijeta. U ovakvom, virtualnom svijetu sve je moguće ali to je i ograničenje, jer glavni interesi za korisnika dolaze upravo iz stvarnog svijeta.

3.3.3. Razlika između virtualne stvarnosti i proširene stvarnosti

Pametni telefoni i drugi mobilni uređaji imaju pristup velikom broju informacija iz stvarnog svijeta koji nas okružuje. Međutim, ove informacije su uglavnom odvojene od stvarnog svijeta. Proširena stvarnost cilja na prezentaciju informacija koje su direktno vezane za fizičko okruženje korisnika. Proširena stvarnost nadilazi mobilno računalstvo i smanjuje prazninu između virtualnog svijeta i pravog, stavrnog svijeta. S proširenom stvarnosti, digitalna informacija postaje dio stvarnog svijeta, bar u korisnikovoj percepciji, dok se virtualna stvarnost se nalazi u korisnikovom kompletno računalno generiranom okruženju. Prema [8], Paul Milgram 1994. godine uvodi pojam kontinuum stvarnosti i virtualnosti gdje opisuje razlike između virtualne stvarnosti i proširene stvarnosti. Također, spominje i proširenu virtualnost. Slika 3.3 prikazuje Miligramov koncept mješovite stvarnosti.



Sl. 3.3. *Miligramov koncept mješovite stvarnosti*

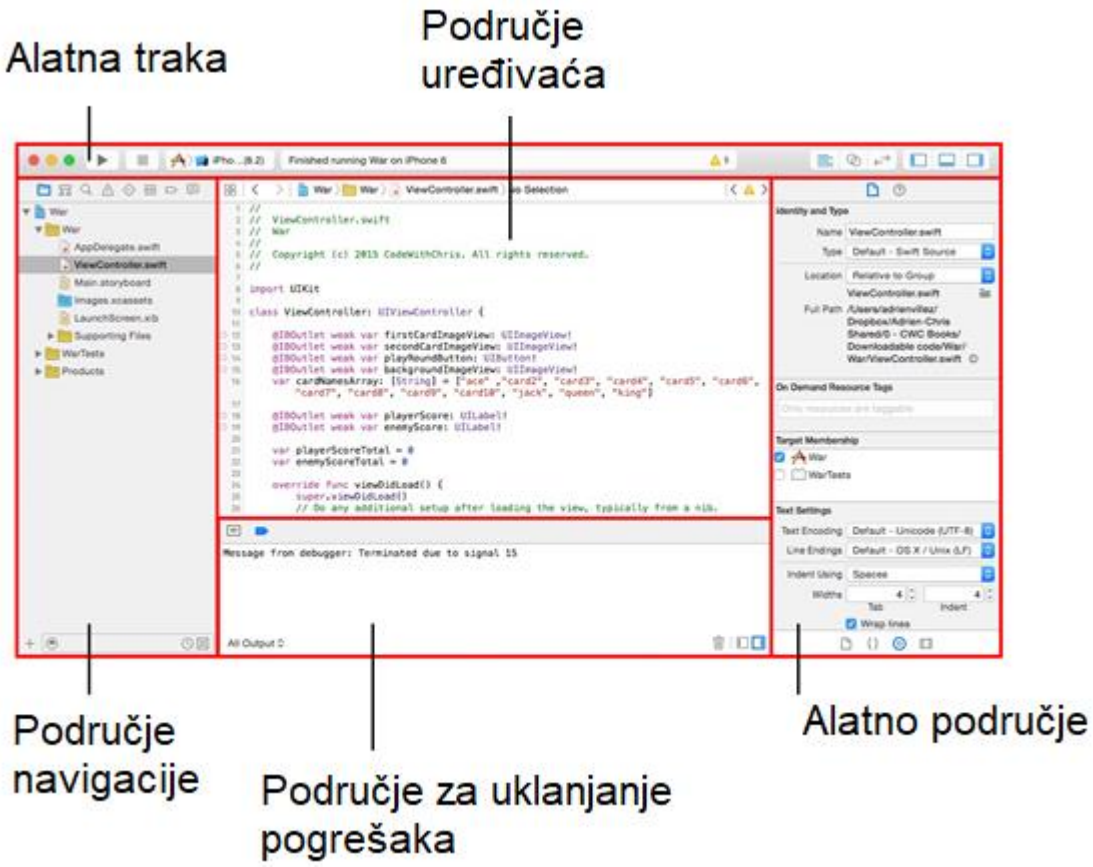
3.4. Razvojna okolina i alati

3.4.1. Računalo Mac s operacijskim sustavom OS X

Prema [9], Mac OS X je prvi put spomenut na Worldwide Developers Conference 1998 godine. Temelji se na Unixu. OS X je deseta inačica Appleovih operacijskih sustava za Mac računala. Zanimljivo je da su prethodne inačice imenovane arapskim brojevima kao npr. OS 9, no kod OS X-a slovo X označava rimski broj 10. Kao glavne prednosti sustava mnogi spominju visoku razinu stabilnosti, sigurnost i jednostavnost korištenja. U trenutku pisanja ovog rada, za razvoj iPhone aplikacija Apple nije službeno dao rješenje kako bi omogućio razvoj aplikacija na računalima bez OS X operacijskog sustava.

3.4.2. Razvojna okolina Xcode

Xcode je razvojna okolina koja se koristi za razvoj iOS i MacOS aplikacija [10]. Sadrži uređivač izvornog koda (engl. source editor), prevoditelj, emulator, razvojne okvire i mnoge elemenate koji su neophodni za razvoj aplikacija. Također, sadrži i simulator kako bi se bez fizičkog uređaja mogle testirati aplikacije u razvoju. Postoje i drugi alati za razvoj, no Xcode je najpopularniji, jer je podržan od strane Applea. Na slici 3.4 prikazan je glavni korisnički zaslon Xcodea.



Sl. 3.4. *Prikaz glavnog korisničkog zaslona Xcodea*

3.5. Korišteni programski jezici

3.5.1. Programski jezik Swift

Prema [11], godine 2014. na Apple Worldwide Developers Conference, Apple je iznenadio sve u trenutku kada je predstavio novi programski jezik Swift. Swift je predstavljen kao „Objective-C bez C“ i godinu dana poslije kao protokolno orijentiran programski jezik. Swift je jezik kreiran za iOS, OSX, WatchOS, TvOS i Linux platforme. Radi s Appleovim Cocoa i Cocoa Touch okvirima i namjera je da bude puno sigurniji od programskog jezika Objective-C koji je bio glavni jezik za razvoj aplikacija na Apple platformama. Razvoj Swifta počeo je 2010. godine. Napravljen je sa LLVM okvirom prevođenja (engl. compiler framework) i koristi Objective-C runtime što omogućuje Objective-C, C, C++ i Swift kodu da se pokrene u jednom programu. Izlaskom inačice 2.2, Swift je napravljen open-source pod Apache 2.0 licencom u 2015. što je odmak od tradicije koju Apple ima da „zaključa“ jezik u svoj ekosustav. Nedugo nakon toga IBM je dao potporu poslužiteljskoj strani Swiftu i izdao web framework Kitura. Swift kao moderan jezik ima mnogo prednosti a neke od njih su čitkost jezika (engl. readability). Programski kod 3.1 prikazuje usporedbu Swifta s Objective-C programskim jezikom koji se

primarno koristio za razvoj iOS aplikacija prije dolaska Swifta, no i danas se koristi.

```
// Objective-C
NSString *name = @"Nate";
NSString *str = [NSString stringWithFormat:@"Hello %@, how are you today?", name];
// Swift
let name = "Nate"
let str = "Hello \(name), how are you today?"
```

Programski kod 3.1. Usporedba Objective-C i Swift koda

Također, velika prednost Swifta nad programskim jezikom Objective-C je sigurnost. Swift uvodi opcionalan (engl. *optional*) tip podatka koji sa sigurnosti rukuje nepostojanje vrijednosti podatka. Tjera nas da eksplicitno rukujemo s *nil* vrijednosti kako ne bi imali krivih predodžbi što neka varijabla sadrži. *Nil* predstavlja nepostojanje vrijednosti. Npr. ako kažemo da je neka varijabla *strOptional* (opcionalan string), znamo da je ta varijabla String ili da ta varijabla nema nikakvu vrijednost. Tako swift uvodi sigurno odmatanje (engl. *unwrapping*) vidljiv u primjeru programskog koda 3.2.

```
guard let str = strOptional else {
    return
}
//Poslije guard-a uvjereni smo da je str varijabla sa nekom vrijednosti tipa String, ukoliko nije izvrsiti ce se return blok

if let str = strOptional {
   print("This is a string: \(str)")
}
//Slična stvar kao kod guard leta no ovdje ce se ukoliko je vrijednost String u str varijabli izvšiti blok sa print naredbom
```

Programski kod 3.2. Rad sa opcinalnim tipom podataka

Iako Swift nije čisto funkcionalan programski jezik, obuhvaća funkcije visokog reda kao što su map, filter, reduce i flatMap. Također, u Swiftu su funkcije „građani prvog reda“ što znači da funkcija može da prihvati neku drugu funkciju kao svoj argument. Samim time sa Swiftom se može funkcionalno programirati. Prema [12], funkcijsko programiranje je proces izrade programske podrške koristeći samo funkcije, izbjegavajući dijeljena stanja i promjenjive podatke. Samim time, smanjuje se broj programskih grešaka, jer ne postoje djeljanja stanja i promjenjivi podaci.U aplikaciji je korišten Swift 3.2, a uskoro izlazi 4.0 koja donosi mnoštvo atraktivnih novosti i poboljšanja među kojima je i puno bolje raščlanjivanje JSON datoteka.

3.6. Korišteni API-ji

Sučelje za programiranje aplikacija, API predstavlja skup određenih pravila i specifikacija koje programeri slijede tako da se mogu služiti uslugama ili resursima nekog drugog složenog programa kao standardne biblioteke funkcija odnosno metoda, struktura podataka, objekata i protokola.

3.6.1. Google Places API

Kako bi se dobile informacije o entitetima koje zanimaju korisnika, u ovom slučaju točke interesa koristi se Google Places API. Zahtjevom na poslužitelj u kojem je potrebno poslati korisnikovu lokaciju i radijus dobiva se povratna informacija o točkama interesa kao što su geografska lokacija, ime i ostale informacije. Temeljem tih informacija, moguće je mapirati točke interesa na kartu prikazanu na zaslonu korisnikovog uređaja. Programski kod 3.3 prikazuje primjer adrese za Google Places API.

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-33.8670522,151.1957362&radius=500&type=restaurant&keyword=cruise&key=YOUR_API_KEY
```

Sl. 3.3. Primjer adrese za zahtjev Google Places

Slanjem parametara lokacije, radijusa, tipa zgrade, ključne riječi i našeg privatnog API ključa dobijemo odgovor koji sadrži JSON datoteku koja ima informacije o restoranima s ključnom riječi „cruise“ koji su udaljeni 500 metara od lokacije koja je poslana. Dio JSON odgovora može se vidjeti u programskom kodu 3.4.

```
{
   "html_attributions" : [],
   "next_page_token" : "qwertzy12345",
   "results" : [
      {
         "geometry" : {
            "location" : {
               "lat" : -33.867217,
               "lng" : 151.195939
            }
         },
         "icon" :
"http://maps.gstatic.com/mapfiles/place_api/icons/cafe-71.png",
         "id" : "7eaf747a3f6dc078868cd65efc8d3bc62fff77d7",
         "name" : "Biaggio Cafe - Pyrmont",
         "opening_hours" : {
            "open_now" : true
         },
         "photos" : [
            {
               "height" : 600,
               "html_attributions" : [],
               "photo_reference" : "easdvdfs",
               "width" : 900
            }
        "types" : [ "cafe", "bar", "restaurant", "food", "establishment"
],
         "vicinity" : "48 Pirrama Rd, Pyrmont"
      },
```

Programski kod 3.4. Dio odgovora dobivenog preko Google Places API

Vidljivo je da se dobivaju mnogi atributi, no koriste se samo oni koji nam trebaju. Također je vidljivo da jedna zgrada može imati više tipova kao što su kafić, restoran, hrana. Primjer zahtjeva i odgovora je dostupan na stranici dokumentacije Google Places API-a, prema [13].

3.7. Pomoćne biblioteke

U ovom poglavlju bit će nabrojene i opisane pomoćne biblioteke (engl. *library)* koje su korištene u izradi aplikacije. Prema [14], programska biblioteka je skup podataka i programskog koda koji se koristi za razvoj aplikacija. Osmišljena je kako bi se pomoglo programeru i prevoditelju programskog jezika u izradi i izvođenju programa.

3.7.1. Mrežna bibliotekaAlamofire

Prema [15], Alamofire je HTTP mrežna biblioteka napisana u Swifu. Pruža elegantno sučelje preko Appleovog temeljnog mrežnog stoga i pojednostavljuje mnogo mrežnih zadataka. Alamofireova elegantnost dolazi od činjenice da je iz temelja napisan u Swiftu i ne nasljeđuje ništa iz Objective-C-ovog AFNetworkinga. Između ostalog, Alamofire pruža mogućnost lančanih mrežnih zahtjeva i odgovora, autentifikaciju, prijenos podataka na mrežu i mnogo toga. Alamofire je biblioteka otvorenog koda (engl. *open source*).

3.7.2. HDAugmentedReality

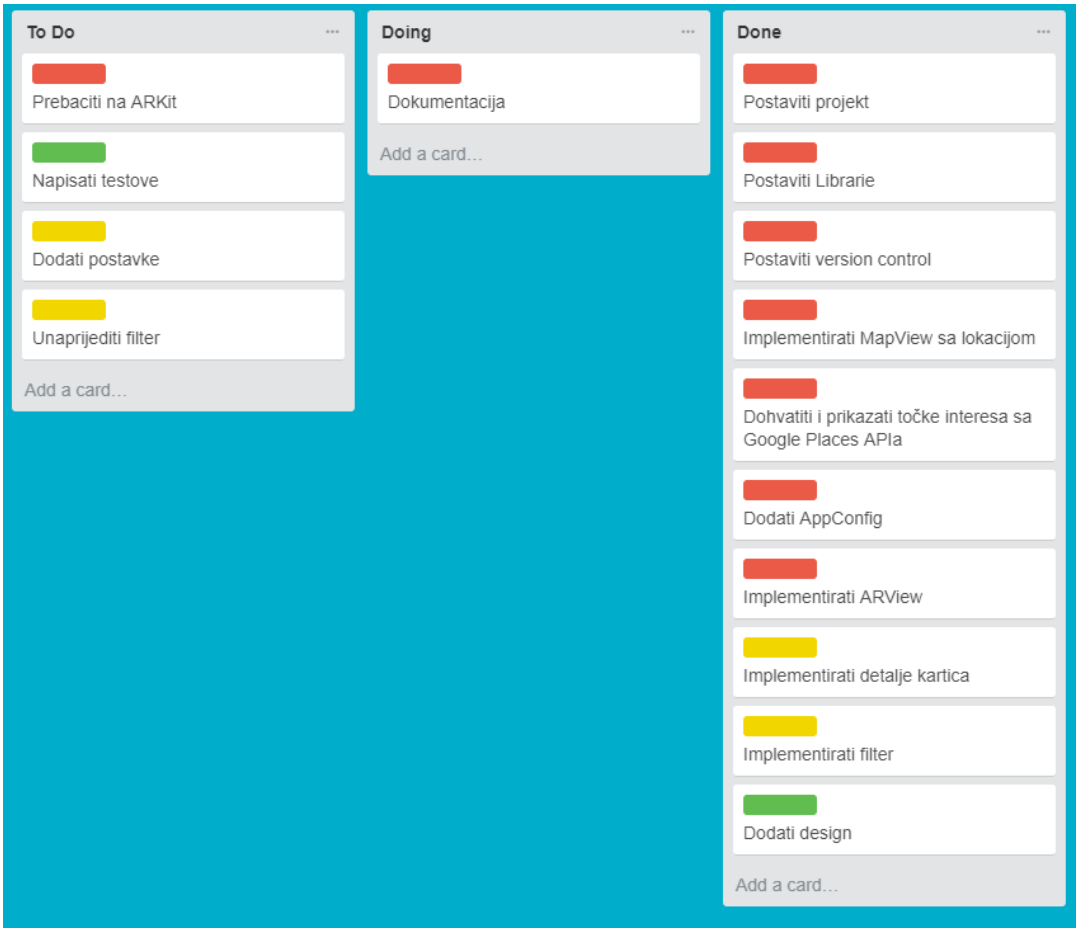
HDAugmentedReality služi kako bi se prikazala proširena stvarnost na zaslonu mobilnog uređaja. Omogućuje stavljanja anotacija preko zaslona, dok je aktivna kamera kako bi se dobio dojam proširene stvarnosti. HDAugmentedReality je biblioteka otvorenog koda napisana u programskom jeziku Swift. Koristi temeljnje razvojne okvire iz Swift programskog jezika kao što su *CoreLocation.Framework* i *CoreMotion.Framework*. Ima podršku za pronalaženje grešaka (engl. *debugging*).

4. PROGRAMSKO RJEŠENJE

U sljedećem poglavlju opisano je programsko rješenje koje je ostvareno pomoću arhitekture i tehnologija opisanih u poglavlju 3. Kao što je spomenuto, zasniva se na MVVM arhitekturi i sastoji se od tehnologije proširene stvarnosti.

4.1. Planiranje i definiranje projektnih zadataka

Uz pomoć besplatne aplikacije u nekomercijalne svrhe Trello [16], kreirana je Kanban ploča prikazana na slici 4.1.



Sl. 4.1. *Kanban ploča*

Kanban ploča ima tri sekcije s definiranim zadacima. Prva sekcija je *To Do* gdje su definirani zadaci koji su spremni za izradu. Druga sekcija *Doing* označava zadaće koji se trenutno izvode. Treća sekcija *Done* označava zadatke koji su riješeni. Kao što je vidljivo na slici 4.1, svaki do zadataka ima i određenu boju koja označava prioritete zadataka. Crvena boja označava zadatke sa najvećim prioritetom. Zelena boja označava zadatke s najmanjim prioritetom dok žuta boja označava zadatke sa srednjim prioritetom. Svaki zadatak je detaljnije definiran klikom na

karticu. Agilna metoda pristupa zadatku se pokazala vrlo uspješna, jer su tijekom izrade projekta ciljevi bili jasno postavljeni i definirani.

4.2. Postavljanje pomoćnih biblioteka

Za dodavanje pomoćnih biblioteka u projekt korišten je CocoaPods. Prema [17], CocoaPods je upravitelj bibliotekama (engl. *dependency manager*) za Swift i Objective-C Cocoa projekte. Ima više od 37 000 pomoćnih biblioteka i korišten je u preko 2.5 milijuna aplikacija. Kako bi se koristili CocoaPodsi dovoljno ih je samo instalirati na računalo s Mac OS sustavom uz pomoć naredbe prikazane u programskom kodu 4.1

```
$ sudo gem install cocoapods
```

Programski kod 4.1. Instaliranje CocoaPods

Zatim je potrebno napraviti Podfile koji ima jasno definirane biblioteke koje smo spremni koristiti. Tijekom izrade projekta, moguće je uređivati Podfile kako bi se dodale ili zamijenile postojeće biblioteke. Podfile korišten u projektu prikazan je u programskom kodu 4.2.

```
platform :ios, '8.0'
use_frameworks!

target “ARPlaces” do
pod 'HDAugmentedReality', '~> 2.2'
pod 'Alamofire', '~> 4.5'
end
```

Programski kod 4.2. Podfile

Vidljivo je korištenje pomoćnih biblioteka navedenih i opisanih u poglavlju 3. Nakon što se kreira Podfile dovoljno je otiči na mjesto projekta i pokrenuti naredbu prikazanu u programskom kodu 4.3.

```
$ pod install
```

Programski kod 4.2. Naredba za instaliranje pomočnih biblioteka

Nakon toga, CocoaPods će pokušati pronaći pomoćne biblioteke u svojoj bazi te ih skinuti i staviti automatski u projekt. Nakon instalacije, pomoćne biblioteke su spremne za korištenje. Time su stvoreni preduvjeti za korištenje Alamofire i HDAugmentedReality pomoćnih biblioteka opisanih u poglavlju 3.7.

4.3. Implementacija trenutne lokacije

Koristeći osnovne iOS komponente, kao što je *MKMapView,* vrlo je jednostavno bilo prikazati mapu na zaslonu uređaja no kako bi se pokazala korisnikova lokacija bilo je potrebno prilagoditi

se protokolu *CLLocationManagerDelegate* u kojem je potrebno definirati metodu vidljivu u programskom primjeru 4.4.

```
    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations:
[CLLocation]) {

        guard locations.count > 0,  let location = locations.last,
            location.horizontalAccuracy < 100 else {
            showAlert(title: Localizable.errorTitle,
                    message: Localizable.generalNoLocationError)
           return
        }

        manager.stopUpdatingLocation()
        let span = MKCoordinateSpan(latitudeDelta: 0.014, longitudeDelta: 0.014)
        let region = MKCoordinateRegion(center: location.coordinate, span: span)
        mapView.region = region
        mapView.removeActiveAnotations()
        viewModel.loadPOIS(location: location)
    }
```

Programski kod 4.4. Prilagodba CLLOcationManagerDelegate protokolu

Iz programskog koda 4.4 vidljivo je korištenje naredbe *guard* koja u Swiftu doprinosi sigurnosti i jedna je od glavnih "krivaca" zašto je Swift siguran jezik. Naredbom *guard* odmah je u početku provjereno dali je uređaj detektirao lokaciju, zatim sigurno odmotava tu lokaciju. Ukoliko se dogodi pogreška, program automatski napušta metodu i poziva funkciju *showAlert* koja ispisuje na zaslonu uređaja informaciju o greški, u ovom slučaju informaciju da nije moguće dobiti trenutnu lokaciju. Ostatak koda se ne izvršava ako metoda ne prođe *guard* uvjete. Ako prođe, od lokacijskog upravitelja se zahtijeva da stane s daljnjim dobivanjem lokacije i prikazuje se korisnikovu lokaciju na mapi. Zatim se iz viewModel-a koji je dio arhitekture opisan u poglavlju 3 šalje zahtjev na Google Places API kako bi se dobile točke interesa u blizini trenutne lokacije.

4.4. Dohvaćanje točaka interesa

Kako bi se dobile točke interesa, potrebno je koristiti API opisan detaljnije u poglavlju 3.6. Implementacija zahtjeva za dohvaćanje točaka interesa opisana je u programskom primjeru 4.5

```
func loadPOIS(location: CLLocation, radius: Int = 1500, filters: [Filter] = []) {

      let filter = self.getActiveFilters(filters: filters)

      Alamofire.request(ApiPath.getPlace(location: location, radius: radius, filter: filter)).responseJSON
{ response in
         switch response.result {
         case .success(let result):
            self.parseDataAndGetAnnotation(data: result)
         case .failure:
            self.onError?(Localizable.generalServerError)
         }
      }
   }
```

Programski kod 4.5. LoadPOIS funkcija

Funkcija *LoadPOIS* prima lokaciju, radijus i listu filtara kako bi mogla napraviti zahtjev za točke interesa u blizini. Međutim, vidljivo je da varijabla *radius* ima zadanu vrijednost od 1500 (metara), a varijabla filters zadanu vrijednost praznog niza []. Razlog tome je što trenutno nije bitan polumjer no ako se odlučimo dodati u aplikaciju postavke mijenjanja polumjera lako ćemo koristiti postojeću metodu. Također, u trenutnoj aplikaciji može se tražiti točke interesa s filtrom ili bez njega. Stavljanjem zadanih vrijednosti na neke parametre, preopterećena je funkcija (engl. overload function) i nije se trebalo napisati više implementacija za skoro istu stvar. Nakon što funkcija *LoadPOIS* primi podatke, stvara se varijabla *filter* koja filtrira sve trenutno uključene filtre. Implementacija funkcije *getActiveFilters* opisana je u programskom kodu 4.6. Preko pomoćne biblioteke Alamofire uspostavlja se zahtjev na URL opisan u strukturi *ApiPath.getPlace* koja će biti opisana zajedno s ostalim konfiguracijskim strukturama u potpoglavlju 4.11. Nakon što se dobije odgovor, provjerava se je li taj odgovor uspješan i ako jest, dobiveni rezultat šalje se u metodu *parseDataAndGetAnnotation* opisanu u programskom kodu 4.7. Ako je odgovor neuspješan, rukuje se pogreškom s funkcijom *onError* koja će biti opisana u poglavlju 5, u potpoglavlju rukovanje greškama.

```
 private func getActiveFilters(filters: [Filter]) -> Filter? {
     return filters.first(where: { $0.isActive })
  }
```

Programski kod 4.6. Funkcija getActiveFilters

Funkcija *GetActiveFilters* je vrlo jednostavna funkcija koja prima listu filtara i vraća filtar koji je opcionalan, što znači da može biti *nil* iz jednostavnog razloga što postoji mogućnost da ne vrati niti jedan aktivan filter. Funkcija koristi funkcije višeg reda kako bi pronašla prvi element koji ima varijablu *isActive* postavljenu na istinitu vrijednost. Vrlo je jednostavna i ovdje pokazuje zašto se Swift smatra čitljivim jezikom, jer i netko bez vještina u Swiftu bi mogao iz same

implementacije shvatiti što se događa.

4.5. Raščlanjivanje odgovora

Raščlanjivanje odgovora engl. (response parsing) služi kako bi se odgovor dobiven s poslužitelja pretvorio u informacije koje su potrebne za rad aplikacije.

```
private func parseDataAndGetAnnotation(data: Any) {

        guard let responseDict = data as? NSDictionary, let placesArray = responseDict.object(forKey:
"results") as? [NSDictionary] else {
            self.onError?(Localizable.generalServerError)
            return
        }
        self.places = []
        for placeDict in placesArray {
            let latitude = placeDict.value(forKeyPath: "geometry.location.lat") as! CLLocationDegrees
            let longitude = placeDict.value(forKeyPath: "geometry.location.lng") as! CLLocationDegrees
            let reference = placeDict.object(forKey: "reference") as! String
            let name = placeDict.object(forKey: "name") as! String
            let address = placeDict.object(forKey: "vicinity") as! String
            let location = CLLocation(latitude: latitude, longitude: longitude)

            guard let place = Place(location: location, reference: reference, placeName: name, address:
address)
            else {
                self.onError?(Localizable.generalServerError)
                return
            }

            self.places.append(place)
            let annotation = PlaceAnnotation(location: place.location.coordinate, title: place.placeName)
            onGotAnnotation?(annotation)
        }
        self.onSuccess?()
    }
```

Programski kod 4.7. Funkcija parseDataAndGetAnnotation

Funkcija *parseDataAndGetAnnotation* prima podataka data koji je tipa *Any*. *Any* tip je nepoznat tip i kao što mu ime sugerira može biti bilo što. Pod pretpostavkom da se radi o JSON datoteci koja je ujedno i rječnik (engl. dictionary) pokušavamo podatak pretvoriti u NSDictionary i zatim odmah dohvatiti rezultate za kljuc "results" koji je opisan u dokumentaciji API-ja koji se koristi. Ako nešto kod pretvaranja u rječnik „pođe po zlu", program poziva funkciju *onError* i zatim izlazi iz funkcije. Ostatak koda se ne izvodi. Ako sve prođe u redu, postavlja se varijabla *places* koja sadrži niz točaka interesa na prazan niz kako bi se pri svakom novom zahtjevu poništila na praznu vrijednost i tako spriječila gomilanje podataka. Zatim se raščlanjuju ostali elementi

rječnika s pripadajućim ključevima, spremaju se u varijable i instancira se klasa *Place* s pripadajućim vrijednostima koje se pridružuju na varijablu *places*. Također se kreira i klasa *PlaceAnnotation* koja se predaje funkciji *onGotAnnotation*. Nakon što funkcija završi s raščlanjivanjem, poziva se varijabla funkcija (objasnjeno u nastavku) *onSuccess* koja ima implementaciju u ViewControlleru opisana u programskom kodu 4.9

```
typealias EmptyCallback = (() -> Void)
   var onGotAnnotation: ((PlaceAnnotation) -> Void)?
   var onSuccess: EmptyCallback?
   var onError: ((String) -> Void)?
   var onSuccessDetails: ((Place) -> Void)?
```

Programski kod 4.8. Definicije varijabli funckija korištenih u ViewModel

Swift omogućuje definiranje varijable koje su funkcije. Kao što je vidljivo u programskom kodu 4.8, postoji nekoliko definiranih varijabli koje primaju različite tipove podataka i imaju različite implementacije. Varijable funkcije mogu primiti ništa ili više ulaznih podatak i isto tako mogu vratiti ništa (*void)* ili nekakav određeni tip podatka. Vrlo su korisne u asinkronom izvršavanju zadataka. Prema arhitekturi opisanoj u poglavlju 3.1, ViewController ima referencu na viewModelu i samim time može pristupiti njegovim varijablama. Implementacije na ViewControlleru varijabli funkcija navedenih u programskom kodu 4.8 vidljive su u programskom kodu 4.9.

```
    viewModel.onGotAnnotation = { newAnnotations in
        self.mapView.addAnnotation(newAnnotations)
    }

    viewModel.onSuccess = {
        self.places = self.viewModel.places
    }

    viewModel.onSuccessDetails = { [weak self] place in
        self?.showAlert(title: place.placeName, message: place.infoText)
    }

    viewModel.onError = { [weak self] error in
        self?.showAlert(message: error)
    }
```

Programski kod 4.9. Implementacija Varijable funkcija u ViewControlleru

Iz programskog koda 4.9 vidljivo je da onGotAnnotation predaje varijablu *newAnnotations* u ugrađenu funkciju MapView-a koji je dodan u potpoglavlju 4.3. Implementacija varijable *onSuccess* u lokalnu varijablu stavlja vrijednosti iz varijable ViewModela. Implementacije varijabli *onSuccessDetails* i *onError* koriste istu funkciju, ali na različite načine. Funkcija

showAlert vidljiva je u programskom kodu 4.10. Služi kako bi se prikazala obavijest korisniku o mogućoj grešci ili neka druga informacija na zaslon mobilnog uređaja.

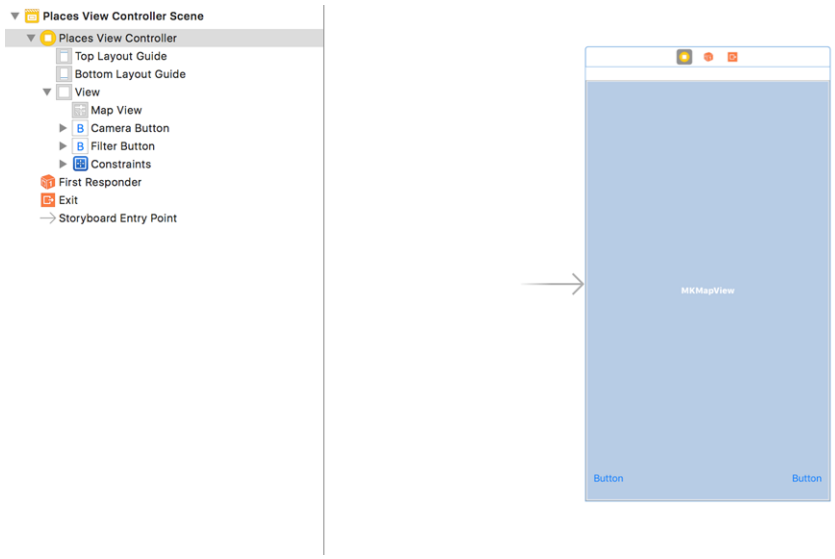
```
func showAlert(title: String = Localizable.errorTitle, message: String) {
        let alert = UIAlertController(title: title, message: message, preferredStyle:
UIAlertControllerStyle.alert)
        alert.addAction(UIAlertAction(title: "OK", style: UIAlertActionStyle.default, handler: nil))
        arViewController.present(alert, animated: true, completion: nil)
    }
```

Programski kod 4.10. Implementacija funkcije showAlert

Funkcija *showAlert* prima naslov alerta i poruku koja se prikazuje. Koristi nativan konstruktor temeljne *UIKit* komponente *UIAlertController*. Nakon konstruiranja i postavljanja akcije s naslovom "OK", prikazuje se na zaslonu. Zanimljivo je da ista funkcija može služiti i za prikaz greške kao i za prikaz detalja u aplikaciji.

4.6. Storyboard i priključci

Prema [17], priključci (engl. *outlets*) su gradivne komponente i služe kako bi se spojile komponente storyboarda. Svaki priključak, kao i varijabla mora imati jedinstveno ime. Također svaki priključak ima svoj tip. Priključci su poveznice između programskog koda i programskog dizajna. Storyboard je vizualna reprezentacija dizajna aplikacije. Na storyboardu se slažu komponente. Na slici 4.2 vidljiv je MKMapView i dva gumba. MKMapView predstavlja kartu koju smo definirali na početku poglavlja 4. Buttoni predstavljaju akcije koje ćemo koristiti. U ovom slučaju gumb filtera i gumb kamere. Storyboard je povezan s kodom preko priključaka.



Sl. 4.2. *Storyboard*

Programski kod 4.11 prikazuje priključke za komponente koje su dodane u Storyboard. Odmah je vidljivo da za tri komponente postoje tri priključka. Kada su priključci dodani u klasu, njih je moguće koristiti u kodu.

```
// MARK: - Outlets
@IBOutlet weak var filterButton: UIButton!
@IBOutlet weak var mapView: MKMapView!
@IBOutlet weak var cameraButton: UIButton!
```

Programski kod 4.11. Prikaz priključaka

Programski kod 4.12 pokazuje definirane metode za izvođenje akcija kao što su klik na gumb. Klikom na gumb Filter poziva se funkcija *filterButtonPressed* koja poziva metodu *addBlur* koja je prikazana u programskom kodu 4.13 i showFilter također prikazana u kodu 4.14

```
// MARK: - UserInteractions
@IBAction func filterButtonPressed(_ sender: Any) {
    addBlur()
    showFilter()
}

@IBAction func cameraButtonTapped(_ sender: UIButton) {
    arViewController = ARViewController()
    setConfiguration(for: arViewController)
    present(arViewController, animated: true, completion: nil)
}
```

Programski kod 4.12. Akcije

```
private func addBlur() {
    let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.light)
    let blurEffectView = UIVisualEffectView(effect: blurEffect)
    blurEffectView.alpha = 0.0
    blurEffectView.tag = 1000
    blurEffectView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
    blurEffectView.frame = view.bounds
    view.addSubview(blurEffectView)

    UIView.animate(withDuration: 1) {
        blurEffectView.alpha = 0.9

    }
}
```

Programski kod 4.13. Metoda addBlur

Prema [18], efekt zamagljena (engl. *blur effect*) je komponenta koju je Apple predstavio u iOS 8.0. Kako bi se napravila glatka tranzicija, dodana je animacija u trajanju od 1 sekunde. Također i pri micanju blur efekta koristi se animacija u istom trajanju. Funkcija prikaza u programskom

kodu 4.14 pokazuje showFilter funkciju kojoj je zadaća instancirati novi ViewController s ciljem prikaza izbornika filtera.

```
  private func showFilter() {
      let storyboard = UIStoryboard(name: Constants.filterStoryboardName, bundle: nil)
      let filterVC = storyboard.instantiateViewController(withIdentifier:
Constants.filterStoryboardName) as! FilterViewController
      filterVC.modalPresentationStyle = .overCurrentContext
      filterVC.delegate = self
      present(filterVC, animated: true, completion: nil)
   }
```

Programski kod 4.14. showFilter metoda

4.7. Filter

Filter se također prema arhitekturi korištenoj u aplikaciji sastoji od *ViewModela*, *Storyboarda* (view) i *ViewControllera*. S obzirom da ima u sebi komponentu tablice, ima dodatnu klasu tipa *UITableViewCell* koja služi za definiranje izgleda ćelije korištene u tablici. Kako bi se mogla koristiti tablica, baš kao i karta u poglavlju 4.3, potrebno je implementirati specifične protokole, u slučaju tablice *UITableViewDelegate* i *UITableViewDataSource*.Programski kod 4.15 prikazuje implementaciju *UITableViewDelegate* i *UITableViewDataSource* protokola odnosno implementaciju njihovih metoda. Prije svega, tablicu zanima koliko će imati ćelija što je lako implementirati. Broj filtara koji će ViewModel imat će biti broj ćelija koje će tablica prikazati. Zatim je potrebno definirati koja vrsta ćelija će se prikazati na kojem redu. S obzirom da tablica ima ćelije koje prikazuju filtere koji svaki ima drugačiju vrijednost no ne i drugačiji dizajn za svaku ćeliju će se koristiti isti dizajn.

```
// MARK: - Delegates
extension FilterViewController: UITableViewDelegate, UITableViewDataSource {

   func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
      return viewModel.filters.count
   }

   func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

      let cell = tableView.dequeueReusableCell(withIdentifier: Constants.cellIdentifier, for: indexPath) as! FilterTableViewCell

      guard let filter = viewModel.itemAt(index: indexPath.row) else {
         cell.isHidden = true
         return cell
      }

      cell.filterLabel.text = filter.name
      cell.filterSwitch.isOn = filter.isActive
      cell.onFilterSwitch = { [weak self] isOn in
         self?.viewModel.filters[indexPath.row].isActive = isOn
      }
      return cell
   }
```

Programski kod 4.15. Implementacija UITableViewDelegate i UITableViewDataSource ViewModel se sastoji od strukture Filter prikazane u programskom kodu 4.16. Struktura filter se sastoji od varijable ime, ključa i zastavice koja pokazuje je li filter aktivan. Varijabla *ime* služi kako bi se na ćeliji prikazalo ime, varijabla ključ služi kako bi se poslao ključ sa zahtjevom na API dok varijabla zastavice služi kako bi znali je li korisnik označio taj filter. Također. struktura ima preopterećeni inicijalizator (engl. *override initializer*) gdje se zastavica ne treba staviti pri kreiranju novih filtera što puno smanjuje kod i olakšava unos novih filtara.

```
struct Filter {
    let name: String
    let key: String
    var isActive: Bool

    init(name: String, key: String, isActive: Bool = false) {
        self.name = name
        self.key = key
        self.isActive = isActive
    }
}
```

Programski kod 4.16. Struktura Filter

Filteri su definirani u nizu filter struktura prikazanog u programskom kodu 4.17. Svaki filter je jednostavno dodati, potrebno je napisati ime i ključ prema dokumentaciji [19]. Kako ništa u tablici nije čvrsto kodirano (engl. hardcoded), sve promjene u nizu filtera će se odraziti na tablicu bez dodatnog kodiranja.

```
var filters: [Filter] = [
        Filter(name: "Atm", key: "atm"),
        Filter(name: "Caffe", key: "caffe"),
        Filter(name: "Park", key: "park"),
        Filter(name: "Parking", key: "parking"),
        Filter(name: "Restaurant", key: "restaurant"),
        Filter(name: "School", key: "school"),
        Filter(name: "Store", key: "store"),
        Filter(name: "University", key: "university"),
    ]
```

Programski kod 4.17. Definicija filtera

4.8. ARViewController

Klikom na gumb kamere instancira se ARViewController klasa i postavlja se konfiguracija prikaza. Na programskom kodu 4.18 vidljiva je setConfiguration metoda koja ima razne parametre. Kao što su maksimalna udaljenost anotacija, orijentacija prikaza i dali je okomito slaganje anotacija podržano.

```
    private func setConfiguration(for arViewController: ARViewController) {
        arViewController.dataSource = self
        arViewController.presenter.distanceOffsetMode = .manual
        arViewController.presenter.maxVisibleAnnotations = 100
        arViewController.presenter.verticalStackingEnabled = true
        arViewController.uiOptions.closeButtonEnabled = true
        arViewController.uiOptions.debugLabel = false
        arViewController.uiOptions.debugMap = false
        arViewController.uiOptions.simulatorDebugging = Platform.isSimulator
        arViewController.uiOptions.setUserLocationToCenterOfAnnotations = Platform.isSimulator
        arViewController.interfaceOrientationMask = .all
        arViewController.setAnnotations(places)
        arViewController.closeButtonImage =  imageLiteral(resourceName: "icons8-Cancel")
    }
}
```

Programski kod 4.18. Postavljanje konfiguracije

ARViewController je klasa koja pokazuje sa ARPresenter pogledom iznad. Koristeći ARTrackingManager određuje položaj mobitela i položaj odgovarajućih točaka interesa koristeći različite senzore. ARPresenter prihvaća niz ARAnnotationView te ih dodaje na zaslon.

4.9. Modeli

Za spremanje mjesta koristi se Place model. Place je klasa koja nasljeđuje ARAnnotation klasu. Implementacije klase Place prikazana je u programskom kodu 4.19. Klasa Places ima varijable kao što su ime, adresa i reference. Dobivene podatke dobije se ranije opisanim postupkom raščlanjivanja podataka.

```
class Place: ARAnnotation {
    let reference: String
    let placeName: String
    let address: String
    var phoneNumber: String?
    var website: String?

    var infoText: String {
        var info = "Address: \(address)"

        if let phoneNumber = phoneNumber {
            info += "\nPhone: \(phoneNumber)"
        }

        if let website = website {
            info += "\nWebsite: \(website)"
        }
        return info
    }

    init?(location: CLLocation, reference: String, placeName: String, address: String) {
        self.placeName = placeName
        self.reference = reference
        self.address = address
        super.init(identifier: reference, title: placeName, location: location)
    }
}
```

Programski kod 4.19. Klasa Place

4.10. Konfiguracija aplikacije

Kako bi održavanje aplikacije bilo što jednostavnije, napravljena je datoteka koja ima nekoliko važnih struktura. U ovom potpoglavlju opisane su neke od njih.

4.10.1. AppConfig

Programski kod 4.20 predstavlja strukturu AppConfig. Prednost strukture iz programskog koda 4.20 je što ako se promjeni apiURL, nije potrebno kroz cijelu aplikaciju mijenjati svaki request već je dovoljno samo na jednom mjestu to napraviti. Također, ako se dobije novi ključ za autentifikaciju može se samo ovdje promijeniti.

```
struct AppConfig {

    static let apiURL = "https://maps.googleapis.com/maps/api/place/"
    static let apiKey = "yourKeyHere"

}
```

Programski kod 4.20 Struktura AppConfig

4.10.2. ApiPath

Programski kod 4.21 prikazuje ApiPath strukturu. U ApiPath strukturi opisani su svi zahtjevi koje aplikacija koristi kako bi dohvatila podatke. Prednost ovog načina je što su sve putanje na jednom mjestu i što se mogu ponovno iskoristiti ako bude potrebe kroz kod. Također, moguće je i vrlo lako i brzo pronaći zahtjevi i prilagoditi ga promjenama koje su moguće.

```
struct ApiPath {

    static func getPlace(location: CLLocation, radius: Int, filter: Filter?) -> String {

        let latitude = location.coordinate.latitude
        let longitude = location.coordinate.longitude
        var filterPath = ""
        if let filter = filter {
            filterPath = "&type=" + filter.key
        }
        return AppConfig.apiURL +
"nearbysearch/json?location=\(latitude),\(longitude)&radius=\(radius)\(filterPath)&key=\(AppConfig.apiKey)"
    }

    static func getDetails(for place: Place) -> String {
        return AppConfig.apiURL +
"details/json?reference=\(place.reference)&sensor=true&key=\(AppConfig.apiKey)"
    }
}
```

Programski kod 4.21 Struktura ApiPath

4.10.3. Localizable

U strukturu Localizable vidljivoj na programskom kodu 4.22 pohranjeni su svi tekstovi koji se koriste u aplikaciji. Ona daje mogućnost da se vrlo lako i jednostavno promjeni određeni tekst bez dodanih ulazaka u sam programski kod. Također daje mogućnost ponovnog korištenja istog teksta na više mjesta kao i mogućnost za višejezičnost.

```
struct Localizable {
    static let generalServerError = "Something wrong happend"
    static let generalNoLocationError = "Sorry, we can't locate you"
    static let errorTitle = "Error"
    static let filterSaveButton = "Save"
    static let filterButton = "Filter"
    static let cameraButton = "Camera"
}
```

Programski kod 4.21 Struktura Localizable

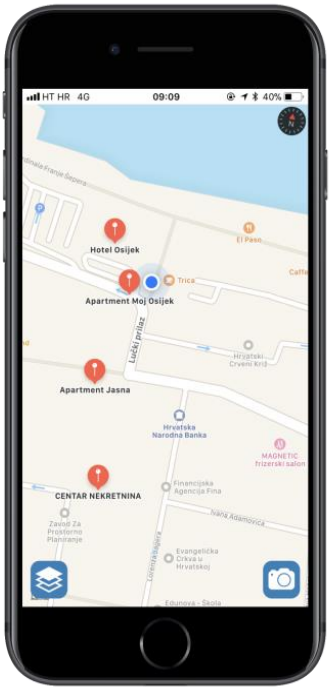
5. PRIKAZ RADA, TESTIRANJE I ANALIZA APLIKACIJE

Ovo poglavlje prikazuje rad aplikacije kroz sve zaslone kroz koje korisnik prolazi služeći se aplikacijom. Također opisuje postupak i rezultate testiranja, te analizira aplikaciju.

5.1. Prikaz rada

5.1.1. Početni zaslon

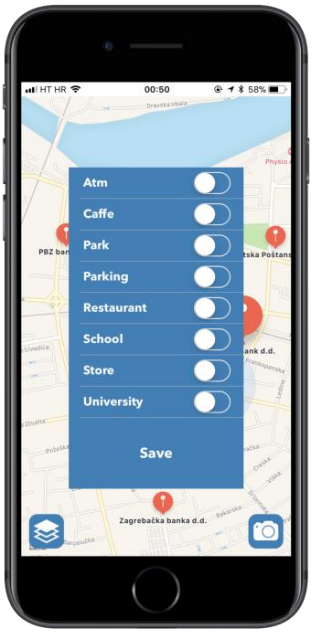
Ulaskom u aplikaciju prikazuje se karta. Nakon nekoliko sekundi prikazuje se korisnikova lokacija na karti i točke interesa u blizini korisnika. Korisnikova lokacija je označena plavom bojom, dok su točke interesa označene crvenom. Na zaslonu se nalaze i dvije tipke. Tipka filter smještena je u donjem lijevom kutu. Tipka kamera smještena je u donjem desnom kutu. Slika 5.1 prikazuje izgled početnog zaslona aplikacije.



Sl. 5.1. *Početni zaslon*

5.1.2. Zaslon filtra

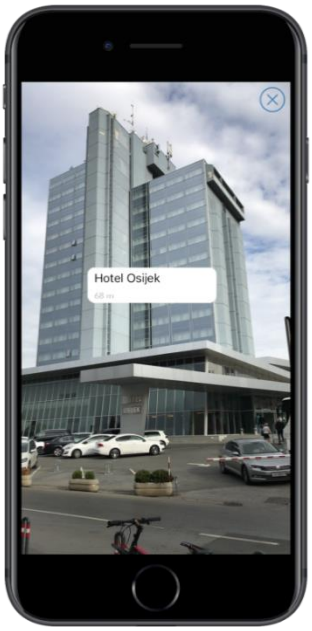
Klikom na donju desnu tipku na početnom zaslonu prikazuje se zaslon filtra. Tijekom prikazivanja, pozadina zaslona se zamagljuje kako bi korisnik bio usredotočen na izbor filtra. Odabirom željenog filtra pritisne se tipka spremi (engl. *save*) i zatim se pokazuju nove lokacije na karti. Slika 5.2 prikazuje zaslon filtara s trenutno niti jednim aktivnim filtrom.



Sl. 5.2. *Zaslon filtera*

5.1.3. Zaslon proširene stvarnosti

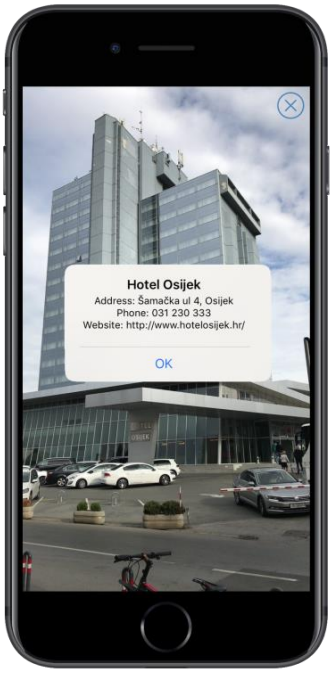
Pritiskom na tipku kamere otvara se zaslon proširene stvarnosti. Na njemu su vidljive sve informacije koje u tom trenutku kamera prima uz dodatak anotacija koje su bijele boje i prikazuju u smjeru točke interesa. U primjeru na slici 5.3 vidljiv je hotel Osijek s opisom koji prikazuje kolika je udaljenost do njega i u kojem smjeru se nalazi. Na zaslonu se nalazi i tipka u gornjem desnom kutu koja služi za izlazak iz moda proširene stvarnosti.



Sl. 5.3. *Zaslon proširene stvarnosti*

5.1.4. Zaslon detalja proširene stvarnosti

Dodirom na anotiaciju prikazuju se detaljnije informacije o točci interesa koju anotacija prikazuje. Ako neki od podataka nije dostupan, taj podatak neće biti prikazan. Detalji anotacije prikazuju ime točke interesa, adresu, broj telefona i mrežnu stranicu. Pritiskom na tipku Ok zatvara se kartica sa detaljima anotacije ali ne i proširena stvarnost. Slika 5.4 prikazuje zaslon detalja o hotelu Osijek.



Sl. 5.4. *Zaslon detalja proširene stvarnosti*

5.2. Testiranje aplikacije

5.2.1. Automatski testovi

Provjera točnosti napisanog koda često je težak posao. Kako bi se programerima olakšala provjera i kako bi se izbjegle greške u daljnjem programiranju, kreiraju se automatski testovi za ispitivanje jedinica (*unit test*). Prema [21], testovi za ispitivanje jedinica testiraju jedan modul. Ako test ne prolazi, to znači da je došlo do greške. Testovi se pišu neposredno prije ili poslije programiranja modula i definiraju specifikacije koje modul treba obaviti. U MVVM arhitekturi testira se ViewModel, jer u ViewModel je smještena logika obrade podataka. Primjeri testova gdje je testirana metoda *getActiveFilters* prikazan je na slici 5.5.

```
25      private func mockFilters() -> [Filter] {
26          return [Filter(name: "One", key: "one"),
27                  Filter(name: "Two", key: "two"),
28                  Filter(name: "Three", key: "three", isActive: true)]
29      }
30
32      func testActiveFiltersIsSuccess() {
33          let filters = mockFilters()
34          XCTAssertEqual(viewModel.getActiveFilters(filters: filters), filters.last)
35      }
37      func testActiveFiltersIsFail() {
38          let filters = mockFilters()
39          XCTAssertNotEqual(viewModel.getActiveFilters(filters: filters), filters.first)
      }
```

Sl. 5.5. *Unit test u razvojnom okruženju Xcode*

Test započinje s ključnom riječi test kako bi sustav prepoznao da se radi o metodi koja će imati implementaciju testa. *MockFilters* metoda služi kako bi se stvorili podaci potrebni za testiranje. *MockFilters* stvara niz filtera kako bi ga predao funckiji *getActiveFilters*. Postoji mnogo različitih funkcija usporedbe (engl. assert function) kako bi se provjerila ispravnost koda. Funkcije usporedbe su dio paketa *XCTest* koji služi za provođenje testova jedinica u Xcode razvojnom okruženju. U primjeru na slici 5.5 korištena je u prvoj funkciji *testActiveFiltersIsSuccess XCTAssertEqual* metoda koja prima dva podatka i uspoređuje dali su jednaki. Test je implementiran tako da se iz podataka koji uđu u tu funkciju očekuje povrat prvog filtera koji je aktivan, što je u ovom slučaju *Filter Three*. Druga funkcija koristi *XCTAsserrtNotEqual* kako bi provjerila da podaci nisu jednaki.

5.2.2. Ručno testiranje

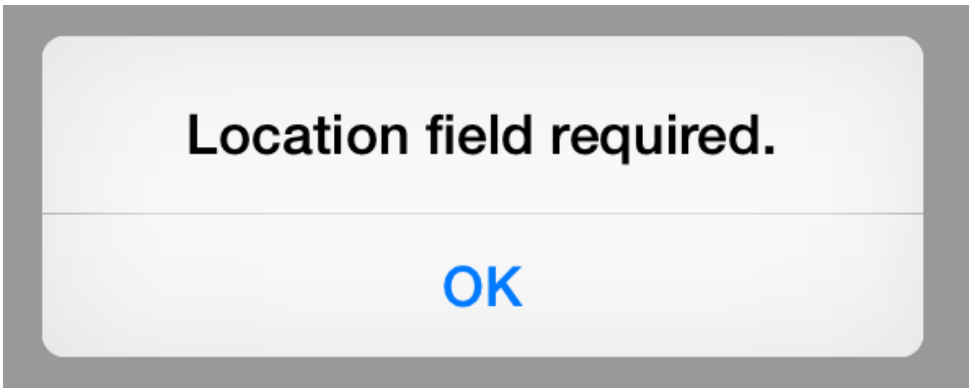
Prema [22], ručno testiranje je proces testiranja aplikacije gdje se osoba koja testira stavlja u ulogu korisnika i pokušava proći kroz sve stadije aplikacije. Kako bi proces testiranja bio potpun, osoba koja testira često zapisuje plan testiranja i važne test slučajeve (engl. test cases). Tester pokušava testirati sva svojstva aplikacije i krajnje slučajeve (engl. edge cases) kako bi uspješno testirao aplikaciju.

Testirajući aplikaciju za prepoznavanje točaka interesa koristeći tehnologiju proširene stvarnosti uvidjeli su se mnogi propusti kao što su nedovoljno točna lokacija i optimizacije zaslona proširenog zaslona gdje anotacije nisu bile dovoljno vidljive.

5.3. Upravljanje greškama

Upravljanje greškama (engl. *error handling*) služi kako bi se obavijestilo korisnika o potencijalnim greškama kroz korištenje aplikacije. Kako bi se obavijestilo korisnika, na svim

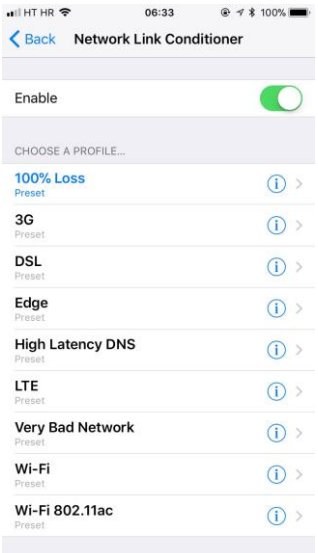
mjestima gdje bi moglo doći do problema korištena je onError varijabla funkcija čija implementacija prikazuje obavijest korisniku kao što je vidljivo u programskom kodu 5.6. OnError prima String koji je opis greške i šalje ga na obavijest zaslona. Takvim pristupom moguće je detaljnije opisati grešku korisniku.



Sl. 5.6. *Obavijest o grešci*

5.4.Analiza aplikacije

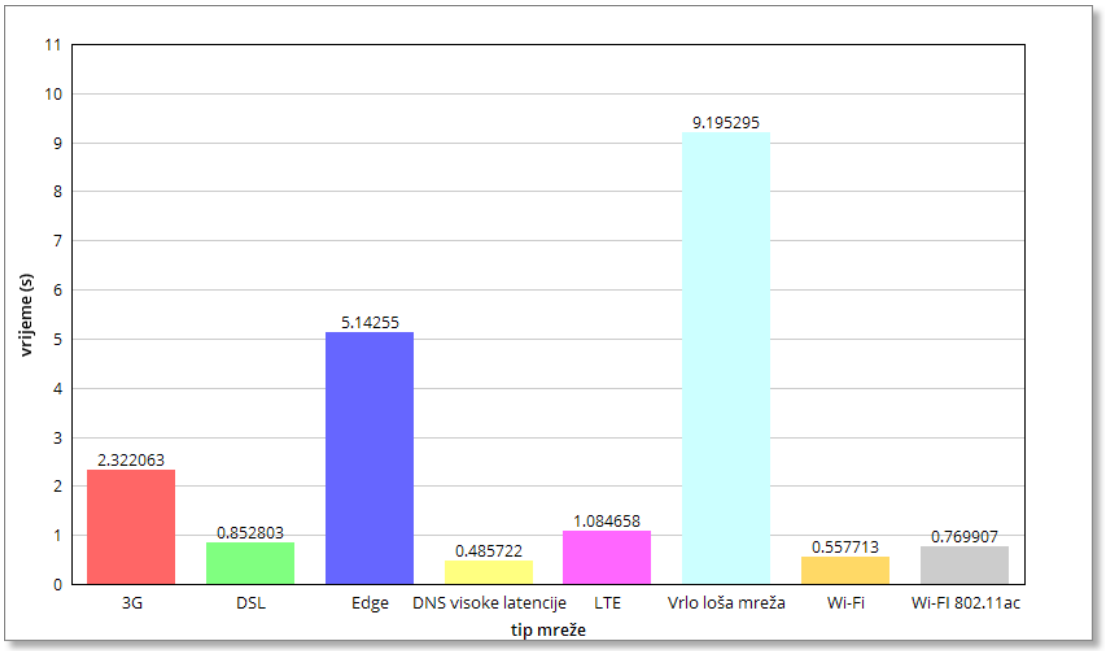
Brzina aplikacije je analizirana programski. Na ulazu u metodu postavljeno je brojilo kao i na izlazu iz metode za dobivanje točaka interesa. Od vremena brojila postavljenog na izlaz oduzeto vrijeme brojila postavljenog na ulaz čini ukupno vrijeme trajanja zahtjeva. Rad aplikacije, odnosno trajanje zahtjeva za slanje lokacije i dobijanja točaka interesa uvelike ovisi o brzini interneta na mobilnom uređaju. Putem programerske konzole na iPhone uređaju i svojstva za kontrolu mrežne veze (engl. *network link conditioner*) (sl. 5.7) moguće je postaviti željene uvjete za testiranje mreže. Rezultati su vidljivi na tablici 5.1 i grafu 5.9



Sl. 5.6. *Kontrola mrežne veze na iPhone uređaju*

Tip mreže	Brzina (s)
100% Gubitak	∞
3G	2.322062541
DSL	0.852802792
Edge	5.142549709
DNS visoke latencije	0.4857215
LTE	1.08465825
Vrlo loša mreža	9.195294958
Wi-Fi	0.557712833
Wi-FI 802.11ac	0.76990675

Sl. 5.6. *Rezultati analize*



Sl. 5.7. *Grafički prikaz analize*

Vidljivo je da se aplikacija uspori i do dvadeset puta ukoliko uređaj ima problema sa sporom mrežnom vezom. Pokušaj analiziranja dobijanja lokacije od GPS-a sličnom metodom nije moguć zbog ograničenih svojstava alata. No, ukoliko bi se i na tom polju dogodio problem, aplikacija bi dodatno usporila.

6. ZAKLJUČAK

U ovom diplomskom radu opisan je postupak izrade aplikacije za mobilni uređaj iPhone koristeći Swift programski jezik i alate koji su usko povezani za rad s iOS platformom. Koristeći spomenute alate napravljena je aplikacija koja pruža korisniku mogućnost da se lakše orijentira u prostoru, koristeći tehnologiju globalnog pozicijskog sustava, API Google Placesa i tehnologiju proširene stvarnosti. Svojstva aplikacije su planirana agilnom metodom razvoja Kanban. Koristeći agilnu metodologiju, izvedba projekta je bila definiranija i jednostavnija. S navedenim tehnologijama postignuto je vrlo vjerno rješenje koje spaja virtualne objekte, u ovom slučaju anotacije mjesta, i scene stvarnog svijeta uhvaćene kamerom mobilnog uređaja. Na taj način, korisnik puno lakše i brže pronalazi svoju poziciju na karti i u prostoru.

Aplikacija je testirana automatski i ručnim putem, rukuje greškama i analizirana je koristeći alate namijenjene programerima mobilnog uređaja iPhone. Aplikacija ima puno potencijala, posebno kada bi se prepisala na ARKit platformu koja je puno stabilnija i u boljem načinu koristi senzore. Također, aplikacija ima filtar koji je vrlo lagano primijeniti. Aplikacija je izrađena prema arhitekturi MVVM i vrlo je skalabilna i laka za održavanje. Ostavlja se mogućnost za dodatno poboljšavanje rada.

LITERATURA

[1] Miljenko Lapaine, Miroslava Lapaine, Dražen Tutić, GPS za početnike, http://www.kartografija.hr/old_hkd/obrazovanje/prirucnici/gpspoc/gpspoc.htm/, pristupljeno: lipanj, 2017.

[2] Joel Windels, Pokémon Go Figure – A data analysis of the most popular game of all time, https://www.wandera.com/blog/pokemon-go-data-analysis-popular-game/, pristupljeno: lipanj, 2017.

[3] Jason McC. Smith, Why Programmers Need Design Patterns to Communicate Effectively, http://www.informit.com/articles/article.aspx?p=2044336, pristupljeno: lipanj, 2017.

[4] Ash Furrow, MVVM in Swift, http://artsy.github.io/blog/2015/09/24/mvvm-in-swift/ , pristupljeno: lipanj, 2017.

[5] Dan Radigan, Kanban, https://www.atlassian.com/agile/kanban, pristupljeno: srpanj, 2017.

[6] Kevin Mise, Big AR: The Future of Augmented Reality, https://hackernoon.com/big-ar-the-future-of-augmented-reality-c91c267b3d9f, pristupljeno: rujan, 2017.

[7] Apple, Introducing ARKit, https://developer.apple.com/arkit/ , pristupljeno: srpanj, 2017.

[8] Paul Milgram, Haruo Takemura, Akira Utsumi, Fumio Kishino, Augmented Reality: A class of displays on the reality-virtuality continuum, Proceedings of Telemanipulator and Telepresence Technologies,1994

[9] Christian Cawley, The History of Mac OS X, http://www.brighthub.com/computing/mac-platform/articles/115648.aspx, pristupljeno: srpanj, 2017.

[10] Apple, Xcode, https://developer.apple.com/xcode/ , pristupljeno: srpanj, 2017.

[11] Frederic Lardinois, Apple Launches Swift, A New Programming Language For Writing iOS And OS X Apps, https://techcrunch.com/2014/06/02/apple-launches-swift-a-new-programming-language-for-writing-ios-and-os-x-apps/ , pristupljeno: srpanj, 2017.

[12] Eric Elliott, Master the JavaScript Interview: What is Functional Programming?, https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0, pristupljeno: lipanj, 2017.

[13] Google, Places API Web Service, https://developers.google.com/places/web-service/search, pristupljeno: srpanj, 2017.

[14] Techopedia, Software Library, https://developers.google.com/places/web-service/search, pristupljeno: srpanj, 2017.

[15] Alamofire, Alamofire, https://github.com/Alamofire/Alamofire , pristupljeno: srpanj, 2017.

[16] https://trello.com/ , pristupljeno: srpanj, 2017.

[17] Apple, Connect the UI to Code thttps://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/ConnectTheUIToCode.html , pristupljeno: srpanj, 2017.

[18] CocoaPods, CocoaPods, https://cocoapods.org/about, pristupljeno: srpanj, 2017.

[19] Apple, UIBlurEffect, https://developer.apple.com/documentation/uikit/uiblureffect, pristupljeno: srpanj, 2017.

[20] Google , Suported types, https://developers.google.com/places/web-service/supported_types , pristupljeno: srpanj, 2017

[21] programabilan.wordpress.com,Unit testing – SPA, https://programabilan.wordpress.com/2011/11/27/unit-testing-spa/ , pristupljeno: srpanj, 2017

[22] http://www.kartografija.hr/old_hkd/obrazovanje/prirucnici/gpspoc/gpspocslike/trisfere.jpg , pristupljeno: lipanj, 2017.

[23] http://www.unc.edu/depts/oceanweb/turtles/geomag.gif, pristupljeno: lipanj, 2017.

[24] https://i1.wp.com/www.everydaykanban.com/wp-content/uploads/2012/03/kanban-board.png, pristupljeno: lipanj, 2017.

SAŽETAK

U ovom radu opisano je korištenje tehnologija mobilnog uređaja i tehnologije proširene stvarnosti za orijentaciju i navođenje u prostoru s posebnim naglaskom na pronalaženje točaka interesa. Opisana je problematika konvencionalnih rješenja i predstavljena je ideja u radu. Nakon kratkog osvrta o postojećim rješenjima prikazan je koncept idejnog rješenja. Nakon koncepta opisan je prijedlog programske arhitekture i opisani su korišteni programski alati. Koristeći programski jezik Swift i agilnu metodologiju razvoja programske podrške Kanban predstavljeno je i opisano programsko rješenje s dijelovima programskog koda. Opisan je prikaz rada aplikacije iz perspektive korisnika. Rad aplikacije zasniva se na senzorima za orijentaciju, lokaciju i dobijene informacije o točkama interesa. Koristeći filtar korisnik je u mogućosti filtrirati sadržaje a opcijom proširene stvarnosti u mogućnosti je vidjeti detalje točaka interesa na zaslonu uz sliku sa kamere, te je na taj način omogućena bolja orijentacija u prostoru. Aplikacija je testirana i prikazani su primjeri testova kao i rezultati analize aplikacije.

Ključne riječi: iOS aplikacija, Kanban, navigacija, proširena stvarnost, Swift, točke interesa

ABSTRACT

POINT OF INTEREST SEARCH BASED ON AUGMENTED REALITY

This paper describes the use of mobile technology and extending reality technology for orientation and guidance in space with a special emphasis on finding points of interest. The problem of conventional solutions is described and the idea is presented in the work. After a short review of the existing solutions, the concept of conceptual solution is presented. Following the concept, the program architecture proposal is described with software tools used. Using the Swift programming language and the agile methodology for sofware development Kanban, a software solution with the parts of the program code is presented and described. A description of the application use from the perspective of the user is described. The application's work is based on orientation sensors, location, and information about points of interest. Using the filter the user is able to filter the content and the augmented reality option is able to see the details of the interest points on the screen beside the image from the camera and thus provide better orientation in the space. The application is tested and displayed examples of tests as well as the results of the application analysis.

Keywords: augmented reality, iOS application, Kanban, navigation, points of interest, Swift

ŽIVOTOPIS

Ivan Matković rođen je 19. svibnja 1992. godine u Našicama. Osnovnu školu upisuje u OŠ „Josip Kozarac“ u Punitovcima. Nakon osnove škole upisao je „Elektrotehničku i prometnu školu Osijek“ u Osijeku, smjer Tehničar za računalstvo. Nakon završene srednje škole 2011. godine, upisao je Sveučilišni preddiplomski studij računarstva na „Fakultetu elektrotehnike, računarstva i informacijskih tehnologija“ u Osijeku. Na fakultetu je 2014. godine. stekao zvanje sveučilišnog prvostupnika (lat. *baccalaureus*) inženjera računarstva. Iste godine upisao je Sveučilišni diplomski studij procesnog računarstva na „Fakultetu elektrotehnike, računarstva i informacijskih tehnologija“ u Osijeku. U lipnju 2015. godine zapošljava se kao Java developer u tvrtki Adcon d.o.o. gdje radi na razvoju telekomunikacijskih sustava. U rujnu 2016. godine prelazi u tvrtku COBE d.o.o gdje se zapošljava kao iOS developer.

PRILOZI

Prilog 1. Završni rad u .pdf formatu

Prilog 2. Završni rad u .docx formatu

Prilog 3. Projekt (programski kodovi)