

Univerza v Ljubljani

Fakulteta za elektrotehniko

Aljaž Dolinar

Simulator mobilnega robota v skladiščnem okolju, zasnovan v okoljih Unity in Matlab

Magistrsko delo

Magistrski študijski program druge stopnje Elektrotehnika

Mentor: izr. prof. dr. Gregor Klančar, univ. dipl. inž. el.

Ljubljana, 2023

Zahvala

Iskreno bi se rad zahvalil mentorju, izr. prof. dr. Gregor Klančar, univ. dipl. inž. el. za njegovo svetovanje, prizadevanje in usmerjanje pri izvedbi magistrskega dela.

Posebna zahvala gre tudi staršem, Mojci in Edvardu za njuno dolgotrajno vzpodbudo in pomoč skozi celotno zgodbo moje izobrazbe. Hvala tudi prijateljem in vsem, ki so me spodbujali pri nastanku magistrskega dela.

Povzetek

V okviru magistrskega dela smo ustvarili orodje za simulacijo kolesnih mobilnih robotov v skladiščnih okoljih. Orodje sestavlja dva ključna gradnike: simulator mobilnega robota in sistem vodenja. Simulacija je realizirana v razvojnem okolju Unity, vodenje pa v okolju Matlab-Simulink. Med seboj komunicirata preko internetnega komunikacijskega protokola UDP. Simulacija je namenjena predvsem, da služi kot učni pripomoček študentom robotike. Dodatno smo implementirali še nekaj pogostih algoritmov vodenja, ki služijo kot primeri uporabe razvite rešitve.

V uvodnih dveh poglavjih magistrske naloge je predstavljena tematika kolesnih mobilnih robotov: njihova umestitev v robotiki in industriji, njihova tipična okolica in senzorika. Poudarek je na razumevanju elementov mobilnih robotov, ki bodo relevantni v kasnejših poglavjih kot del simulacije.

V naslednjem poglavju je predstavljena simulacija, ki smo jo ustvarili v okviru tega dela. Za pomoč pri razumevanju so na kratko predstavljene glavne uporabljene komponente in pomembni pojmi okolja Unity. Razloženi so vsi ključni elementi programa: kontrolnik kamere, kontrolnik grajenja, sistem shranjevanja in nalaganja, simulacija samega robota in implementacija UDP komunikacije z okoljem Matlab-Simulink. Predstavljene so tudi implementacije simuliranih senzorjev: svetlobni merilnik razdalje Lidar, senzor za sledenje črti in senzor za zaznavo NFC značk.

V nadaljevanju je predstavljenih nekaj pogostih rešitev iskanja optimalne poti za mobilne robote: algoritmom Dijkstra, algoritmom A* ter iskanje v širino in iskanje v globino. Podrobno je predstavljena tudi izvedba algoritma A*, ki je bila v okviru tega magistrskega dela implementirana v okolju Matlab.

V sklepnem delu je predstavljenih še nekaj najpogostejših algoritmov vodenja, ki smo jih kot primer uporabe za končnega uporabnika simulacije razvili v okolju Matlab-Simulink. Gre za algoritme vodenja v točko, vodenja po črti in izogibanja trkov.

Ključne besede: mobilni robot, simulacija, vodenje, PID-regulator, iskanje poti, Unity, Matlab, Simulink, UDP.

Abstract

Within the scope of this thesis, we created a tool for simulating wheeled mobile robots in warehouse-like environments. It contains two key elements: a mobile robot simulator and its control system. The simulation was created in Unity development platform and the control system was made in Matlab-Simulink environment. The two elements communicate via the UDP internet communication protocol. The simulation is intended to serve as a learning assist to robotics students. Additionally, we implemented a few common control algorithms, which serve as usage examples of the developed solution.

In the opening two chapters of the thesis, the topic of wheeled mobile robotics is presented: its place in robotics and industrial applications, their typical environment and sensorics. An emphasis is placed on understanding those elements of mobile robots, that will be relevant in the upcoming chapters as parts of the simulation.

Next chapter presents the simulation, that was created in the scope of this thesis. The most important used components and concepts of Unity environment are briefly explained to aid the reader's understanding. Key elements of the program are explained: a camera controller, a building controller, a save and load system, the robot simulation itself and the implementation of communication between the two used environments via UDP. Additionally, implementations of simulated sensors are also presented: Light ranging and detection sensor (Lidar), Line following sensor and NFC tag reader.

Afterwards, a few common optimal path finding solutions are presented: Dijkstra's Algorithm, Algorithm A*, Breadth-first Search and Depth-first Search. The chapter also contains a detailed explanation of the implemented A* algorithm.

A few of the most common control algorithms are presented in the closing chapter. We developed the following control algorithms in the Matlab-Simulink environment: control to point, line following control and obstacle avoidance. These serve as usage examples for the end user of the developed simulation.

Key words: mobile robot, simulation, control, PID control, pathfinding, Unity, Matlab, Simulink, UDP.

Vsebina

1	Uvod	1
1.1	Namen magistrske naloge	2
1.2	Uporabljena metodologija	2
2	Mobilni roboti	5
2.1	Delovno območje	6
2.2	Senzorika	7
2.2.1	Propriocepcijski senzorji	8
2.2.2	Ekstrocepcijski senzorji	8
2.2.3	Uporabljeni senzorji	8
3	Simulacija	11
3.1	Programsko okolje Unity	12
3.2	Vzpostavitev razvojnega okolja	14
3.3	Kontrolnik kamere	14
3.4	Kontrolnik grajenja	16

3.5	Sistem shranjevanja in nalaganja	19
3.6	Simulacija robota	20
3.6.1	Regulacija motorja in zavoja	21
3.6.2	Svetlobni merilnik razdalje	23
3.6.3	Senzor za sledenje črti	25
3.6.4	Senzor za zaznavo markerjev	25
3.7	UDP komunikacija	27
3.8	Programsko okolje Matlab-Simulink	28
4	Iskanje poti	31
4.1	Algoritem Dijkstra	33
4.2	Algoritem A*	34
4.3	Algoritma iskanje v širino in iskanje v globino	35
4.4	Implementacija algoritma A*	36
5	Vodenje	41
5.1	Vodenje v točko	41
5.2	Vodenje po črti	43
5.3	Izogibanje trkov	47
6	Zaključek	51
Literatura		53

Seznam slik

2.1	Primer avtonomnega mobilnega robota.	5
2.2	Izgled referenčnega robota.	6
3.1	Simulacija mobilnega robota.	11
3.2	Poenostavljen diagram poteka simulacije robota.	12
3.3	Zaslonska slika razvojnega okolja.	13
3.4	Perspektivna projekcija kamere z ozkim vidnim poljem.	15
3.5	Prikazan sistem treh objektov kamere.	16
3.6	Prikaz duha grajenja ob že postavljenih objektih istega načrta. . .	17
3.7	Prikaz označbe za brisanje objekta.	18
3.8	Prikaz skladišča po meri, ki je bil ustvarjen z uporabo kontrolnika grajenja, sestavljen iz treh zaslonskih slik.	19
3.9	Prikaz razlike med modelom trka robota in vizualnim modelom robota.	20
3.10	Robot med zavijanjem in viden indikator zavijanja.	23
3.11	Lidar pogled iz ptičje in stranske perspektive.	24
3.12	Vizualizacija senzorja za sledenje črti.	26

3.13	Primer zemljevida skladišča, ki ga Matlab ustvari ob zagonu.	28
3.14	Diagram poteka glavne zanke osnovnega Simulink modela.	29
4.1	Primer strukture grafa v sistemu z mobilnimi roboti za namene iskanja poti.	32
4.2	Diagram poteka implementacije algoritma A* v sklopu dela.	37
4.3	Rezultat A* algoritma, prikazan na zemljevidu skladišča.	40
5.1	Skica veličin pri vodenju robota v točko.	43
5.2	Prikaz vedenja robota pri izvedbi algoritma za vodenje po črti (di- menzije so večje od dejanskih za boljšo preglednost).	44
5.3	Prikaz izbire načina sledenja črti v skladu z setom akcij, ki jih je vrnil algoritem A*.	47
5.4	Prikaz vedenja algoritma za izogibanje oviram.	48

1 Uvod

Mobilna robotika postaja z vsakim dnem pomembnejša veja robotike, saj njena glavna prednost pred klasično industrijsko robotiko, zmožnost avtonomne mobilnosti, znatno poveča vsestranskoščnost sistemov. Mobilne robote najdemo v mnogo oblikah: od humanoidnih robotov, brezpilotnih letečih robotov in do skladiščnih robotov. Služijo različnim namenom: humanoidni roboti imitirajo ljudi, lahko pozdravljam in vodijo obiskovalce, služijo kot pomagalo pri učenju, kot zabava in da pritegnejo pozornost na razstavnih dogodkih; Brezpilotna letala so popularna za snemanje dogodkov na prostem, kot hobi za navdušence letenja, ter za vojaške namene. V tem delu pa se bomo predvsem posvetili avtonomnim kolesnim mobilnim robotom, pogosto okrajšani s kratico AGV (Avtonomno vodenno vozilo, ang. Autonomous guided vehicle) oziroma AMR (Avtonomen mobilni robot, ang. Autonomous mobile robot). Pogosto služijo za transport surovin, polizdelkov in izdelkov znotraj skladišč oz. tovarniških poslopij.

Pri implementaciji vodenja se močno opiramo na dejstvo, da je območje gibanja robota vnaprej znano: Poznamo oblike prostorov, postavitve in povezave med njimi. Pomemben aspekt vodenja je tudi upoštevanje dejstva, da si robot deli prostor z ljudmi, ljudje pa imamo lastnost, da smo robotom nepredvidljivi. Lahko stopimo pred robota, ki ga ne vidimo, lahko odložimo kakšno oviro na robotovo običajno pot, lahko mu zastremo značilke, po katerih se orientira in podobno. Vedenje robota za primere najpogostejših človeških nepredvidljivosti je potrebno implementirati tako, da je robot varen za ljudi v svoji okolici, da ne povzroča nepotrebne materialne škode in da je pri opravljanju svojega dela kar se da avtonomen.

1.1 Namen magistrske naloge

Zastavljen cilj magistrske naloge je: zasnovati programsko aplikacijo oz. simulator, ki bo simuliral AGV robota v uporabniško definiranih skladiščih. Simulacija in vizualizacija bo realizirana v programskem okolju razvojne platforme v realnem času Unity [1] (ang. Unity Real-Time Development Platform, v nadaljevanju Unity). Vodenje bo realizirano v programskih okoljih Matlab in Simulink [2]. Za ustvarjanje 3D modelov se je uporabilo tudi programsko orodje za 3D modeliranje Blender [3].

Končni rezultat bo uporaben tudi kot učni pripomoček za nove generacije študentov, katerih program vsebuje predmet Avtonomni mobilni sistemi. Trenutno se vaje predmeta izvajajo v laboratoriju na fizičnih robotih, ki se premikajo znotraj pripravljenega poligona, s pomočjo Robotskega operacijskega sistema [4] (ang. Robot Operating System), bolj znanega s kratico ROS. Vaje v laboratoriju dobro prikažejo uporabo in programiranje mobilnih robotov, vendar pa se je z epidemijo leta 2020 izkazalo, da je pametno študentom omogočiti preizkušanje algoritmov tudi na domačem računalniku.

1.2 Uporabljena metodologija

Na začetku smo analizirali nekaj primerov učnih mobilnih robotov in izbrali katerega bomo uporabili v simulaciji. Naredili smo 3D model robota in modele predmetov v njegovem okolju: stene, ograje, talne oznake in ovire. Nekatere smo po potrebi dodal naknadno.

Nato smo zasnovali simulacijo v okolju Unity. Ustvarili smo prazen projekt, in v njem vzpostavili osnove: kamero in možnost premikanja, globalno osvetlitev, osnove okolja in hierarhijo objektov v projektu. Dodali smo objekte okolja, jih skonfigurirali, in naredili sistem za uporabniško grajenje lastnih skladišč s temi objekti. Potem smo dodali objekt robota, ga skonfigurirali in napisali kodo, ki je odgovorna za vedenje robota: regulacija motorjev, zaznavanje okolja, in prikazovanje povratnih informacij uporabniku.

V naslednjem koraku smo vzpostavili komunikacijo med nastalo simulacijo in vodenjem v okoljih Matlab in Simulink. Robot sporoča podatke s svojih senzorjev, vodenje pa mu nazaj sporoča želene hitrosti in kote zavijanja. Zadnji korak je bila implementacija vodenja: vodenje v točko, sledenje črti, iskanje poti (ang. Path finding) in izogibanje ovir.

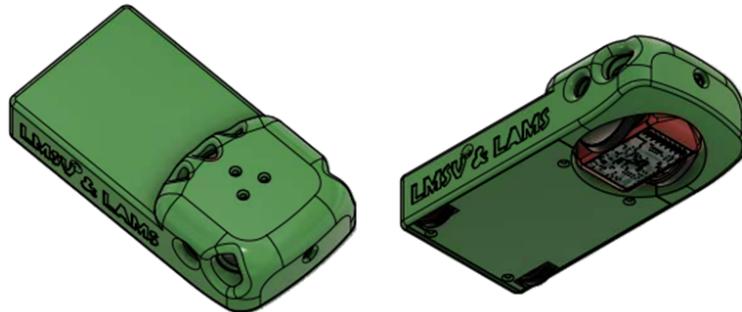
2 Mobilni roboti

Humanoidni roboti, brezpilotni roverji, robotski hišni ljubljenčki in droni so odlični primeri mobilnih robotov. Od ostalih robotov se razlikujejo po svoji zmožnosti avtonomnega premikanja z zadostno inteligenco, da se odzovejo in odločajo na podlagi podatkov, ki jih zaznavajo v svojem okolju. Mobilni roboti potrebujejo vir vhodnih signalov, sposobnost njihove interpretacije, in zmožnost izvajanja različnih dejanj (vključno s svojo lokomocijo) kot odziv na spremnajočo se okolico. Potreba po zaznavi in prilaganju neznanemu okolju zahteva zmogljiv kognitivni sistem [5]. V tem poglavju so opisane glavne lastnosti mobilnih robotov in njihove senzorike.



Slika 2.1: Primer avtonomnega mobilnega robota.

V delu obravnavam predvsem mobilne kolesne robote, znane pod predhodno omenjeno okrajšavo AGV. Primer tipičnega AGV robota je viden na sliki 2.1. Od ostalih mobilnih robotov se razlikujejo po tem, da se gibljejo po kopnem z vsaj dvemi (diferencialni pogon), tipično pa s štirimi kolesi (npr. Ackermannov pogon). Posebnost pri štiri kolesnih pogonih, ki je vredna omembe, je tako imenovan vsesmerni pogon (ang. Omnidirectional drive), kjer ima robot posebej zasnovana kolesa, ki mu omogočajo premikanje tudi v lateralni smeri in rotacijo robota na mestu. Za robota v simulaciji smo za grobo referenco izbrali robota Laboratorija za avtomatiko in kibernetiko na Fakulteti za Elektrotehniko, ki smo ga uporabljali na laboratorijskih vajah pri študiju. Končna verzija vsebuje znatno manj podrobnosti v geometriji, z namenom optimizacije programa. Na sliki je vidna tudi posebna oblika pogona, kjer prednji kolesi zavijata tako, da se rotirata okoli skupne navpične osi. V nalogi uporabljam Ackermannov pogon, vendar z enakim kotom zavoja na obeh prednjih kolesih. Zunanje kolo ob zavojih posledično rahlo drsi lateralno, kar natančen fizikalni model simulacije podpira.



Slika 2.2: Izgled referenčnega robota.

2.1 Delovno območje

Delovna območja skladiščnih AGV robotov potrebujejo pravilno zastavljenou infrastrukturo. Pogosto spominjajo na svoj lasten cestni sistem z jasno definiranimi progami, pasovi, območji za pešce (oz. delavce), polnilnimi postajami, ipd. Ključni element za navigacijo robota po takšnem področju so talne črte,

ki označujejo predvidene poti robotov. Pogosto so realizirane s talno barvo, namenskim lepilnim trakom ali pa narejene iz kovine in pritrjene. Zadnja opcija je trajnejša in zanesljivejša od prvih dveh, posledica tega pa je dražja cena začetne investicije za vzpostavitev sistema. Roboti imajo za ta namen poseben senzor na svetlobni osnovi, ki pod robotom išče takšno črto, algoritmom vodenja robota pa z njegovo pomočjo skrbi, da se robot skozi celotno vožnjo centrira na zaznano črto. Predvidene poti se zastavi tako, da so najpomembnejše relacije optimalne, tveganje za trke pa najmanjše.

V kombinaciji z sledenjem črt je smiselno uporabljati tudi markerje za lokализacijo. Z njihovo pomočjo definiramo odseke poti saj je vsak marker unikaten. Ko robot zazna marker, lahko preprosto ugotovimo, kje se robot nahaja, saj je lokacija markerja vnaprej znana in konstantna. Predvsem jih postavljamo na začetke križišč, da se ob njihovi zaznavi lahko odločimo, katero pot bo robot ubral. Ker robot v križiščih zaznava več črt, je potrebno prilagoditi logiko sledenja. Ena od možnih rešitev, ki je opisana tudi v kasnejšem poglavju, je sledenje levemu oz. desnemu robu črte, odvisno seveda od želene smeri zavoja. Ker začasno zaznavamo le rob črte, nam prisotnost večih črt ne zmede algoritma vodenja, saj je potrebno le najti najbolj levo oz. desno zaznavo črte na senzorju in regulatorju podajati to vrednost (namesto sredine črte).

2.2 Senzorika

Senzorji avtonomnim mobilnim robotom omogočajo zaznavanje okolice kot tudi svojega delovanja, podobno kot čutila živih bitij (npr. oči, ušesa, ipd.). Ko so združeni s programsko opremo senzorjev, robotu omogočajo razumevanje okolja in navigacijo v njem, zaznavo in izogibanje trkov z objekti, in podajanje informacij o lokaciji robota [6]. Predvsem lokalizacija, torej ugotavljanje točne pozicije mobilnega robota v svoji okolici, je temeljen problem mobilne robotike [7]. Vrste senzorjev mobilnih robotov pogosto delimo na propriocepcijske in eksterocepcijiske senzorje.

2.2.1 Propriocepcijski senzorji

Propriocepcijski senzorji zaznavajo podatke o samem robotu. Na njih se zanašamo predvsem za regulacijo motorjev robota. Na kolesih se nahajajo enkoderji, ki nam za izbrano kolo z vsakim zajetim vzorcem podajo absolutno rotacijo glede na referenčno lego kolesa, ali pa relativno rotacijo glede na lego kolesa v prejšnjem vzorcu.

2.2.2 Eksterocepcijski senzorji

Eksterocepcijski senzorji zaznavajo okolje, v katerem se robot nahaja. Primeri pogosto uporabljenih senzorjev so: kamere, LiDAR, radar in GPS. Te senzorje najpogosteje uporabljamo za lokalizacijo robota, za izogibanje trkov in višenivojsko odločanje. Senzorji za lokalizacijo robota so lahko postavljeni tudi v okolju, pogost primer je sistem kamer, ki s prepoznavo slik določi lokacijo in orientacijo robota, to pa nato posreduje algoritmu vodenja robota.

2.2.3 Uporabljeni senzorji

V tem podoglavlju opisujem senzorje, ki jih v simulaciji uporabljamo. Tu so predstavljene resnične izvedbe, v kasnejših poglavijih pa so opisane njihove pripadajoče simulirane enačice. Prva dva predstavljeni senzorji sta propriocepcijska senzorja, zadnji trije pa eksterocepcijski.

- Pozicijo in orientacijo tipično zaznavamo s pomočjo sistema kamer, postavljenih na višjih točkah v delovnem območju, in značilk na robotu. Tipične značilke so QR-kode, razporejene na vidnih mestih robota. Med seboj so si različne, v sistemu za lokalizacijo pa hranimo podatke, ki nam za vsako QR-kodo povedo na katerem robotu in kje na robotu je nameščena. Ko algoritem prepozna na vsaj eni kameri značilko, lahko njeni pozicijo na sliki kamere preračuna v realno pozicijo. Pri izračunu je potrebno izvesti perspektivistično transformacijo, ki upošteva lastnosti kamere (npr. goriščna razdalja in parametri popačenja). Tipično vsaj ena kamera zaznava več

značilk na robotu, zato je možno ugotoviti tudi orientacijo robota. Večje število zaznanih značilk na robotu se prevede v večjo natančnost končnega rezultata. Alternativna opcija sistemu kamer je odometrija. Robot si zapomne zadnjo znano lego (npr. s pomočjo markerja), nato pa od trenutka dalje integrira hitrost (v 3D prostoru), da dobiva novo pozicijo. To storí s pomočjo enkoderjev in matematičnega modela, ki zaznane premike na posameznih kolesih preračuna v spremembo orientacije in pozicije robota.

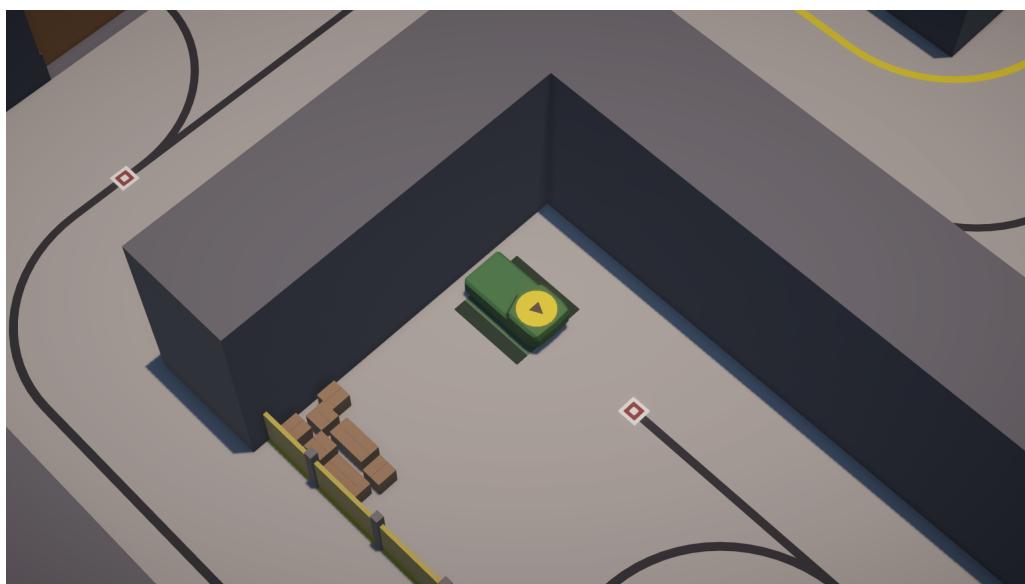
- Hitrost na mobilnih robotih zaznavamo s pomočjo enkoderjev, ki je tudi splošno pogosta metoda merjenja hitrosti. Gre za senzor, ki se namesti na vsako kolo robota, ki meri absolutni ali inkrementalni zasuk kolesa. V obeh primerih so realizirani z virom svetlobe in senzorjem svetlobe (oba pritjena na robota, ne kolo), med njima pa je disk z vrsto rež (disk se vrta skupaj s kolesom). Ta med vrtenjem prekinja svetlobnega žarek med virom in senzorjem. Vrednost svetlobnega senzorja periodično zajemamo z mikro-krmilnikom, nato pa lahko iz časov zaznanih prekinitev svetlobnega žarka izračunamo hitrost, saj je zasuk kolesa med dvema prekinitvama znan, s tem pa tudi razdalja, ki jo je kolo prepotovalo. Da lahko izmerimo smer vrtenja dodamo še eno, rahlo zamaknjeno vrsto rež in nov set vira in senzorja svetlobe, iz zaporedja obeh zaznav pa lahko ugotovimo smer vrtenja. Če želimo vedeti absolutni zasuk, ne pa inkrementalni, dodamo še več setov rež (s posebno postavitvijo zasukov in različnih dimenziij rež), virov svetlobe in senzorjev, nato pa prilagodimo izračun. Za meritev hitrosti zadošča že inkrementalni enkoder.
- Lidar (zaznava svetlobe in dometa, ang. Light detection and ranging) je merilnik razdalje, ki deluje na osnovi svetlobnih žarkov. V osnovi deluje tako, da pošlje svetlobni žarek proti objektu, do katerega želimo izmeriti razdaljo. Nato s svetlobnim senzorjem preverja, kdaj se odboj poslanega žarka vrne nazaj. Ker so nam znani čas, ko je bil žarek poslan, čas ko se je žarek vrnil in svetlobna hitrost, lahko preprosto izračunamo razdaljo, ki jo svetloba dvakrat prepotuje v izmerjenem času.
- Senzor za sledenje črti zaznava prisotnost in lateralni zamik črte in njenih robov glede na senzor. Tipično so nameščeni na spodnji strani prednjega dela robota. Obstaja več možnih implementacij: z virom svetlobe (vidne ali infrardeče) in merilniki odbite svetlobe ter s kamero. V delu simuliram

prvi tip. Deluje tako, da senzorji svetlobe, razporejeni vzdolž lateralne osi robota, zaznavajo odbito svetlobo od svetlobnega vira senzorja. Tako se lahko ugotovi, kateri senzorji vidijo črto (tipično temnejša, torej zaznajo manj odbite svetlobe) in kateri je ne. Na podlagi teh podatkov se nato izračunajo zamiki sredine, desnega ali levega roba črte od središča senzorja.

- Zadnji implementirani senzor je bralnik NFC značk (komunikacija bližnjega polja, ang. Near Field Communication). Alternativno bi lahko uporabljali bralnik RFID značk (identifikacija s pomočjo radijske frekvence, ang. Radio Frequency Identification), da bi dosegli primerljiv rezultat. NFC je ponavadi omejen na domet nekaj centimetrov, zato je izjemno pomembno, da se bralnik na robottu zadostno poravna z značko na tleh, čez katero robot zapelje. NFC bralnik oddaja brezžični signal, ki sproži odziv NFC značke, če je ta dovolj blizu. NFC značka nato odda svoj signal, v katerem sporoči (v našem primeru) svojo identifikacijsko številko (ID). Posebnost tu je, da magnetno polje NFC bralnika napaja vezje v NFC znački, da ta odda svoj signal. NFC značka v mnogih izvedbah ne potrebuje lastnega napajanja, kar znatno olajša namestitev in vzdrževanje sistema.

3 Simulacija

V okviru magistrske naloge je bila ustvarjena simulacija mobilnega robota s pomočjo štirih programskih okolij: Unity, Blender, Matlab in Simulink. V okolju Unity se izvaja sama simulacija, vidna na sliki 3.1, ki nato komunicira z algoritmom vodenja, ki je realiziran v okolju Simulink, skupaj s podporo okolja Matlab za inicializacijo potrebnih spremenljivk. Komunikacija med okolji je realizirana s pomočjo UDP (uporabniški datagramskega protokola, ang. User Datagram Protocol). Programsko okolje Blender pa je bilo uporabljen za ustvarjanje 3D modelov, uporabljenih v Unity simulaciji.



Slika 3.1: Simulacija mobilnega robota.

V poglavju se bomo predvsem osredotočili na simulacijo robota, realizirano v okolju Unity. Poenostavljen diagram poteka je viden na sliki 3.2. Glavna

zanka se izvaja s fiksno frekvenco, vzporedno se ob zaznavi uporabniških vhodov vpletejo še kontrolnik grajenja, kontrolnik kamere oz. sistem za shranjevanje in nalaganje. Vodenje v okolju Simulink in inicializacija v okolju Matlab bosta opisana v kasnejših poglavjih, na tem diagramu sta prikazana z namenom, da ta prikaže celotno sliko.



Slika 3.2: Poenostavljen diagram poteka simulacije robota.

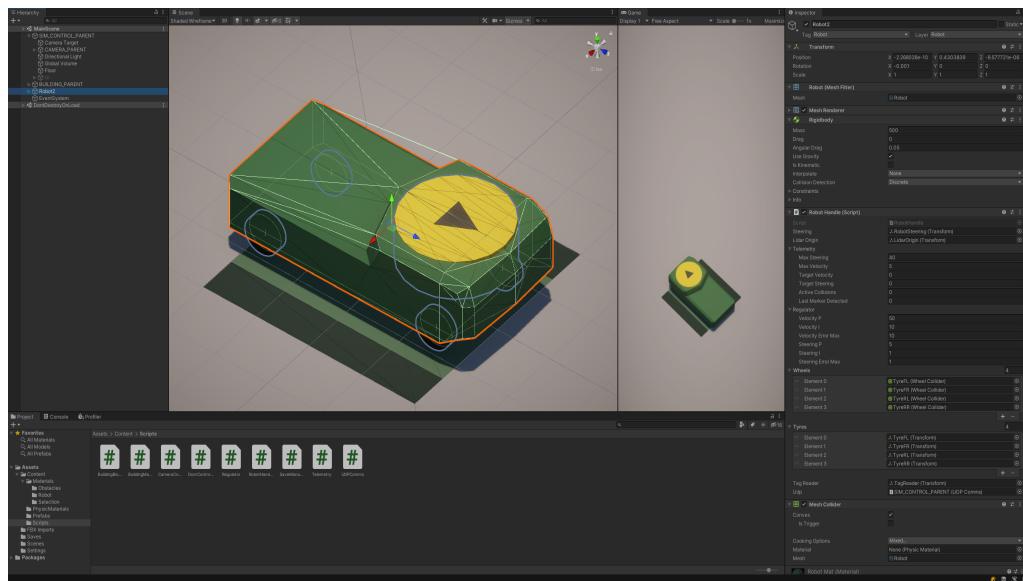
3.1 Programsko okolje Unity

Okolje Unity je priljubljeno programsko okolje, sicer primarno namenjeno razvoju računalniških iger, vendar se pogosto uporabi tudi v robotiki za vizualizacije in simulacije.

V okolje uvozimo 3D modele, v tem primeru narejene s programom za 3D modeliranje Blender, ki jim potem dodamo skripte, komponente in lastnosti, da se vedejo kot zaželeno. Unity te objekte, ki so temeljni gradnik okolja, imenuje igralni objekt (ang. Game Object, v nadaljevanju imenovan le objekt). Vsa koda je napisana v jeziku C#, v urejevalniku Microsoft Visual Studio.

Na sliki 3.3 je prikazano razvojno okolje Unity: Na sredini imamo pogled scene

na levi (ang. Scene View), kjer urejamo začetne lege objektov, in pogled igre na desni (ang. Game View), kjer je prikazan predogled kamere, ki ga ob zagonu simulacije potem tudi vidimo. Levo je prikazan seznam vseh objektov v simulaciji. Na skrajni desni imamo t.i. inšpektorja (ang. Inspector), ki nam omogoča pregled in spremjanje lastnosti, komponent in skript izbranega objekta. Na dnu pa je viden pregled nad datotekami v projektu: napisani skripti, uvoženi 3D modeli, uporabljeni materiali, ki določajo izgled površin objektov, ipd.



Slika 3.3: Zaslonska slika razvojnega okolja.

Unity vsebuje napreden sistem za izračun fizike (ang. Physics Engine), ki omogoča modeliranje in simulacijo fizikalnih veličin in zakonov, na katerega se v projektu močno opiram. Ta nam z izbrano frekvenco izračunava sile in navore na objekte, zaznava trke med pravilno konfiguriranimi objekti ter njihove odzive. Iz teh podatkov potem izračuna nove lege objektov, kar se prikaže tudi na zaslonu.

V projektu uporabljamo frekvenco $f = 100Hz$ za t.i. fiksni časovni korak (ang. Fixed Timestep). Ta določa časovni razmik (čas vzorčenja) med zaporednim klicanjem metod (ang. method), ki potrebujejo konstanten časovni razmik, če je računalniški sistem tega zmožen. V tej kategoriji so vsebovani vsi izračuni fizike [8], metoda fiksne posodobitve (ang. FixedUpdate method) vsakega aktivnega objekta, in premiki objekta s kamero. V metodi fiksne posodobitve je pametno tudi posodabljati vse podatke, ki potrebujejo stalno posodabljanje, a

jim izbrana frekvenca posodabljanja zadošča. V opisani simulaciji so to npr. posodobitve stanj regulatorja robota in rotacija indikatorja kota krmiljenja koles robota (ang. robot steering angle).

Metoda posodobitve (ang. Update method) je zelo podobna predhodno omenjeni metodi fiksne posodobitve, vendar se izvede vsakič, ko se na grafični kartici računalniškega sistema izriše nova sličica (ang. frame). Frekvenca klicov te metode je torej močno odvisna od zmogljivosti in obremenitve računalniškega sistema, ki program izvaja. V tej metodi se tipično izvaja računsko nezahtevna koda, povezana z zaznavanjem vhodov uporabnika, npr. pritiskov na tipkovnici ali miški, in koda, povezana z vizualnimi efekti, npr. animiranje uporabniškega vmesnika.

3.2 Vzpostavitev razvojnega okolja

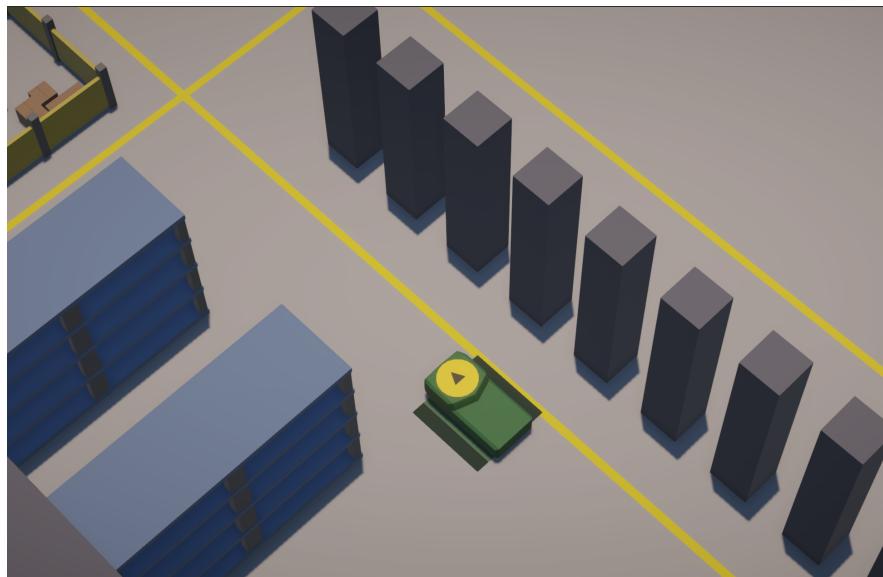
V okolju Unity uporabljam t.i. Univerzalni postopek upodabljanja (ang. Universal Render Pipeline, znano pod kratico URP), ki je eden od treh privzetih postopkov upodabljanja, ki jih okolje nudi. Prednost tega postopka je predvsem računska učinkovitost algoritma za izris 3D prizorov na grafični kartici, saj je namenjen razvoju aplikacij, ki so zmožne delovati tako na računalnikih, kot na telefonih in igralnih konzolah. Z izbiro tega postopka pa se odpovemo modernim in naprednejšim grafičnim možnostim, ki jih tipično zmorejo uporabljati predvsem dražji sodobni računalniki. Za simulacijo mobilnega robota je ta kompromis sprejemljiv, saj nam je računska učinkovitost pomembnejša. Z istim razlogom tudi uporabljam samo globalno osvetlitev, brez dodatnih virov svetlobe, saj bi z vsakim le povečali računsko zahtevnost izračuna. Okolje Unity nudi že pripravljeno rešitev za globalno osvetlitev objektov, v projekt jo dodamo z le nekaj preprostimi kliki.

3.3 Kontrolnik kamere

Kontrolnik kamere (eng. Camera Controller) omogoča uporabniku nadzor nad kamero. Uporabnik lahko kamero premika v vse smeri po površini tal in jo vrти

okoli navpične osi. Uporabnik lahko preklaplja med privzetim načinom, kjer kamera s 60° kotom gleda navzdol proti tlem, in načinom, kjer gleda navpično navzdol z 90° kotom. Uporabnik lahko tudi prikaže širše oziroma ožje področje s preklapljanjem med dvema nastavitevama širine vidnega polja (ang. Field of View). S pritiskom na robota, kamera začne slediti robotu in ga ohranja v sredini pogleda. S premikom kamere sledenje prekinemo.

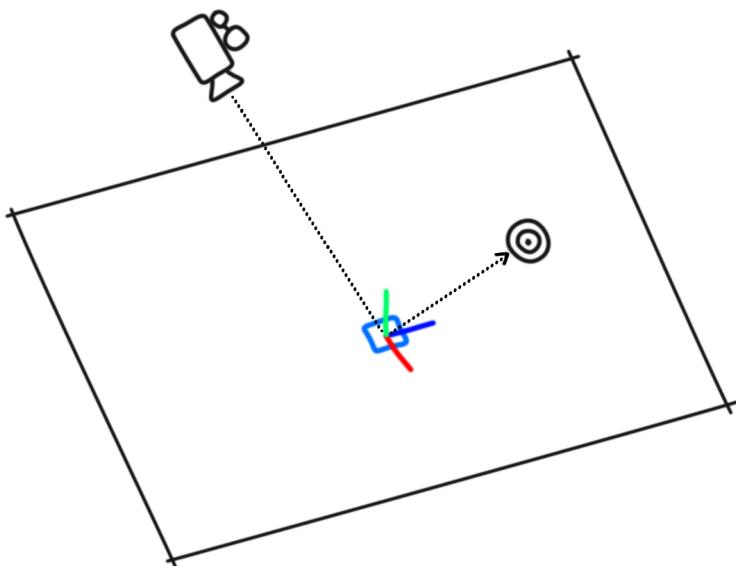
Kamera na prvi pogled spominja na ortografsko kamero, vendar lahko vidimo rahle posledice perspektivne projekcije. Uporabljamо perspektivno kamero z nizko širino 18° vidnega polja, rezultat česar je opisani učinek, prikazan tudi na sliki 3.4. Učinek je predvsem viden, ko primerjamo stebre na desni polovici slike.



Slika 3.4: Perspektivna projekcija kamere z ozkim vidnim poljem.

Da dosežemo gladko gibanje kamere uporabljam sodelovanje treh nevidnih objektov: Kamera, njen nadrejeni objekt (ang. Parent Object) in tarča (ang. Target Object). Sistem je viden na sliki 3.5. Kamera je glede na nadrejeni objekt zamaknjena za določen vektor in določeno rotacijo v koordinatnem sistemu nadrejenega objekta. Nadrejeni objekt je postavljen v središče vidnega polja kamere, kar posledično pomeni, da ob rotaciji in premikanju nadrejenega objekta ta ostaja v središču vidnega polja kamere. Nadrejenega objekta ne premikamo direktno s pritiski in držanjem tipk, ampak premikamo tarčo za premik, katerega vrednost je določena kot produkt vnaprej določene hitrosti in časa trajanja izrisa zadnje

sličice. Tako dobimo hitrost premika tarče, ki je neodvisna od trenutne hitrosti posodabljanja sličic (ang. frame rate). V metodi fiksne posodobitve pa nato premikamo nadrejeni objekt proti tarči. Za to uporabljamo zelo uporabno metodo za linearno interpolacijo med dvema vektorjema, imenovano `Vector3.Lerp`. Rezultat je postopen premik proti tarči, ki se upočasni z manjšanjem razdalje. Podobno se naredi za rotacijo tarče in širino vidnega polja, vendar uporabimo metodo za sferično interpolacijo med dvema kvaternionoma (ang. Quaternion), imenovano `Quaternion.Slerp`, oz. metodo za linearno interpolacijo med dvema decimalnima vrednostima s plavajočo vejico (ang. float), imenovano `Mathf.Lerp`.



Slika 3.5: Prikazan sistem treh objektov kamere.

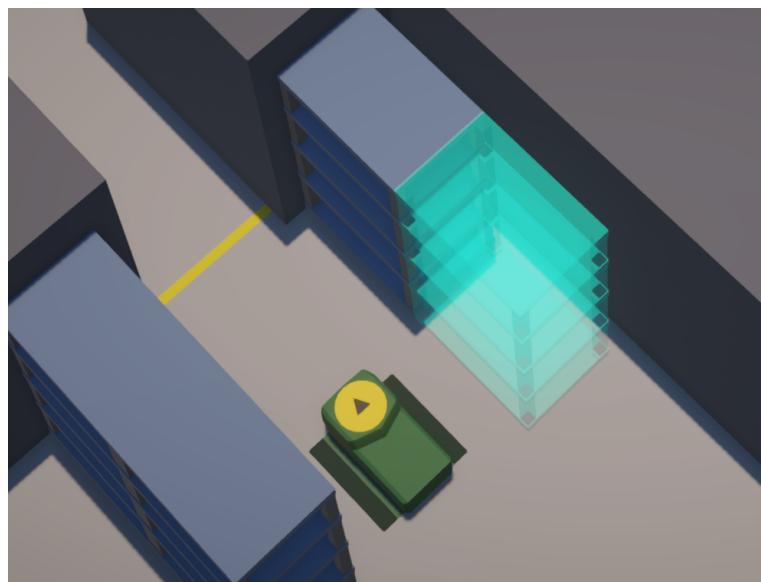
3.4 Kontrolnik grajenja

Kontrolnik grajenja (ang. Building Manager) je skript, ki uporabniku omogoča spremeniti postavitev ovir in izgled območja v simulaciji. Uporabnik vklopi oz. izklopi način grajenja s pritiskom na tipko ”B”, nato s kolescem na miški izbere objekt, ki ga želi postaviti. Prikaže se duh objekta (ang. ghost), ki prikazuje lokacijo in orientacijo bodočega objekta, če se ga uporabnik odloči tam postaviti. To je vidno na sliki 3.6 kot svetlomer delno prosojen objekt, zraven pa je izbrani objekt že postavljen za primerjavo. V tem načinu je tudi možno že postavljeni

objekte izbrisati, z dvema pritiskoma desnega miškinega gumba na izbran objekt.

Implementacija kode je sledeča: v spominu imamo shranjen izbran načrt objekta (ang. object blueprint) in njegovo rotacijo. V metodi posodobitve preverimo vsako sličico, če je kateri od relevantnih vhodov pritisnjen. Ko je način grajenja izklopljen je to le tipka "B", ki ga omogoči, drugače pa imamo definirane še vhode za: rotiranje in postavitev objektov, menjavo izbranega načrta objekta in nastavitev velikosti mreže, na kateri je objekt postavljen v center polja.

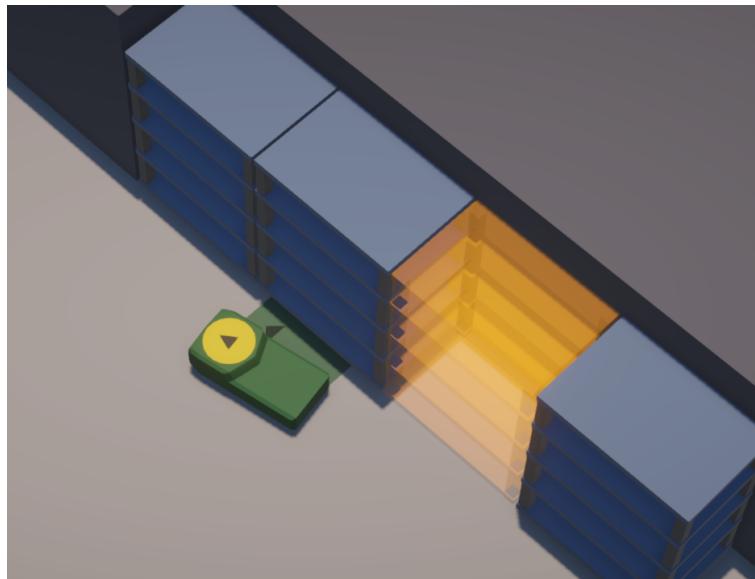
Pritiske vhodov zaznavamo v metodi posodobitve te skripte, ko uspešno zazna pritisk vhoda kliče ustrezno funkcijo. Če smo zagnali način grajenja se mora ustvariti duh objekta, ki je svoj objekt z vidika Unity-a, in nato slediti miški. Je kopija objekta iz izbranega načrta, kjer zamenjamo vse materiale objekta z materialom v transparentno-modri barvi. Ob rotaciji, se mora duh zavrteti. Ob menjavi izbranega načrta grajenja je potrebno uničiti star duh objekta in ustvariti novega. Potrebno ga je tudi uničiti če izstopamo iz načina grajenja. Ob levem miškinem kliku je potrebno ustvariti nov objekt iz trenutno izbranega načrta in mu določiti lego (enake vrednosti, kot jih ima duh objekta).



Slika 3.6: Prikaz duha grajenja ob že postavljenih objektih istega načrta.

Obstaja tudi možnost brisanja obstoječih objektov. Ko smo v načinu grajenja, lahko z desnim miškinim klikom označimo objekt. To se odraža v oranžni verziji

efekta, ki je viden na sliki 3.7 za grajenje. Če označen objekt ponovno kliknemo z desnim miškinim gumbom se objekt odstrani.

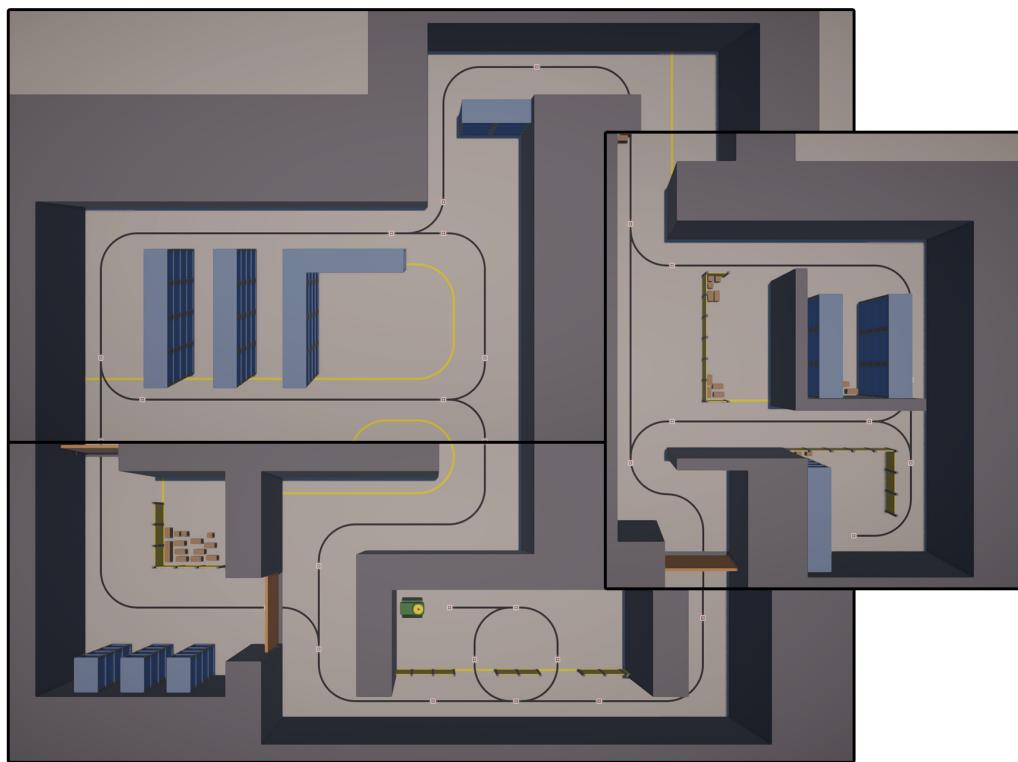


Slika 3.7: Prikaz označbe za brisanje objekta.

Z uporabo tega orodja je možno ustvariti raznolika in zanimiva skladišča po meri. Primer uporabniškega skladišča, ustvarjenega na tak način (brez uporabe Unity urejevalnika, ampak znotraj programa), je viden na sliki 3.8.

Na sliki so vidni razni možni objekti, ki jih sistem dovoli postavljati: začetno mesto robota, stene, ograje, vrata, omare, škatle, črne talne črte (za robotov senzor za sledenje črti), rumene talne črte (zgolj dekorativne, služijo le kot oznaka), in markerji.

Objekti imajo razne posebnosti. Začetno mesto robota ima to posebnost, da se lahko le prestavi, ne pa ustvari novo, saj ni smiselno imeti večih. Tla so vedno prisotna in se jih ne da urejati, saj so objekti vedno postavljeni na ta tla (objektov ni možno dvigati od tal pri postavljanju). Škatle se vedejo kot fizikalni objekt: robot lahko z njimi trči in jih odrine, te pa ga nazaj zavirajo. Vrata s frekvenco $f_v = 1Hz$ preverjajo, če je robot v bližini, in se odprejo. Vloga markerjev in črnih črt pa je povezana s senzorji na robotu in je predstavljena v poglavju 3.6.



Slika 3.8: Prikaz skladišča po meri, ki je bil ustvarjen z uporabo kontrolnika grajenja, sestavljen iz treh zaslonskih slik.

3.5 Sistem shranjevanja in nalaganja

Ko smo s pomočjo sistema iz prejšnjega poglavja naredili skladišče, je čas, da ga shranimo. To se izvede v skripti za shranjevanje (ang. Save Manager), ki preverja pritisk tipke za shranjevanje "F11". Ko se sproži, ta shrani kopijo trenutnega skladišča v tekstovno datoteko, kjer ustvari seznam vseh objektov v simulaciji, njihovih pozicij in orientacij ter njihove pripadajoče načrte objekta. Primer vnosa tega seznama je naveden spodaj, vsak vnos predstavlja en shranjen objekt.

```
i4x -14.00y0.00z31.00r0e
```

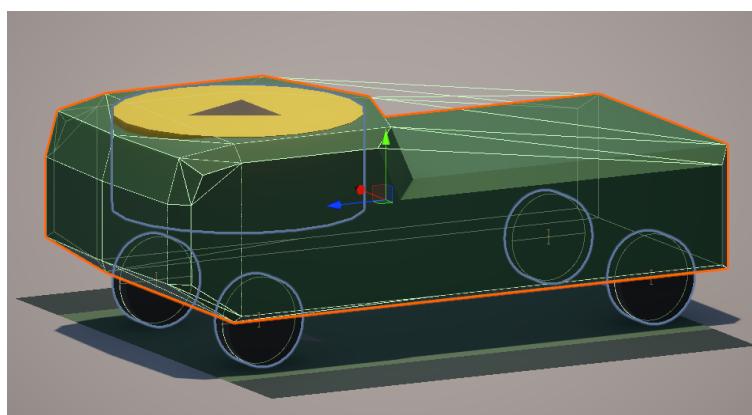
Številka za črko i nam pove zaporedno številko načrta shranjenega objekta, številke za črkami x, y in z nam povedo pozicije, številka za črko r nam pove rotacijo objekta okoli navpične osi (y os), črka e pa označuje konec vnosa. Iz tega shranjevanja so izvzeti objekti, ki so vedno prisotni, npr. kamera. Njih v vsakem

primeru naložimo ob zagonu simulacije, zato jih ni potrebno dodatno shranjevati v datoteki.

Ko želimo naložiti predhodno shranjeno datoteko, skript naloži izbrano datoteko, iterira čez celoten seznam teh objektov v tekstovni datoteki in enega za drugim ustvari s pomočjo načrtov objektov, na shranjeni legi.

3.6 Simulacija robota

Robota simuliramo s pomočjo vgrajenega sistema za izračun fizike in skripte, ki ga vodi. 3D modelu robota, ki sem ga uvozil v Unity kot nov objekt, sem dodal komponente za modele trka (ang. Collider), in komponento togega telesa (ang. Rigidbody). Skupaj te dve komponenti objektu omogočita interakcijo z drugimi fizičnimi objekti. Ker ima robot komponento togega telesa, nanj lahko vplivajo sile, in se ne vede kot del nepremakljive stene. Tu mu definiramo maso, koeficient zračnega upora, koeficient trenja površine in mnogo drugih. Na sliki 3.9 so vidni model trka robota in modeli trkov posameznih koles kot svetlo zeleni obrisi. Robot je modeliran kot konveksna aproksimacija svojega modela, kolesa pa so modelirana sicer kot cilindri, vendar kot da imajo infinitezimalno širino. Te ponostavitev so seveda storjene v imenu optimizacije izračuna, njihove posledične nenatančnosti pa so zanemarljive.



Slika 3.9: Prikaz razlike med modelom trka robota in vizualnim modelom robota.

Objektom, ki morajo biti upoštevani v izračunih trkov, dodamo komponento tipa 3D model trka. Ta ima več podvrst, ki so odvisne od oblike objekta. Nam relevantni so kvadrasti model trka (ang. Box Collider), natančni mrežni model trka (ang. Mesh Collider), viden kot svetlozelen žični okvir na sliki 3.3, in kolesni model trka (ang. Wheel Collider). Prvi je uporabljen za aproksimacijo objektov, ki imajo približno obliko kvadra. Mrežni model trka natančno sledi obliki 3D modela objekta, a znatno poveča računsko zahtevnost zaznave trkov. Kolesni model trka pa je narejen specifično za kolesa vozil, in ima temu primerno dodatne opcije. Za vsako kolo posebej lahko definiramo krivulje oprijema v smeri gibanja in lateralni smeri, izberemo vrednosti koeficientov vzmeti in dušenja ter omejitve pozicije v vzmetenju. Direktno jim lahko nastavljamo vrednosti navora motorja, navora zaviranja, in kot zavoja (ang. steering angle). To znatno olajša vključitev robota v sistem za izračun fizike.

3.6.1 Regulacija motorja in zavoja

Robot potrebuje regulator za motor, zaviranje in kot zavoja. Unity je namreč fizikalni simulator, ki gibanje objektov simulira preko sil in navorov, ki nanj delujejo. Da lahko nastavljamo želene hitrosti gibanja, moramo torej izvesti regulator, ki za želene hitrosti določi potrebne navore. Za regulacijo po hitrosti sem izbral proporcionalno integracijski regulator (tip PI), saj ostali tipi regulatorjev niso zadovoljivo sledili referenčni hitrosti. Vhod v regulator je referenčna hitrost, izhod pa je navor, in je v zveznem prostoru opisan z enačbo 3.1. Na podlagi pogreška $e(t)$. Če ima izračunan navor pozitiven predznak, bo motor pospeševal gibanje, in ga zaviral, če je izračunan navor negativen. Če je referenčna hitrost negativna, bo negativen navor dosegel vzvratno vožnjo, in zaviral s pozitivnim navorom.

$$M(t) = k_P e(t) + k_I \int_0^t e(\tau) d\tau \quad (3.1)$$

Med testiranjem se je pojavila potreba po zaščiti pred integralskim pobegom regulatorja. V primeru trka motor robota ni mogel ustvariti zadostnega navora, da bi robot dosegel referenčno hitrost. Posledica tega je tako imenovan integralski pobeg, kjer integracijski člen regulatorja akumulira čedalje večjo napako. V

opisanem primeru se to odraža v zakasnitvi med spremembo referenčne hitrosti v negativno (da se vzratno umaknemo od točke trka), in v dejanski spremembi hitrosti v negativno. Dlje kot je robot neuspešno silil v oviro, dlje časa traja, da se akumulirana napaka spusti nazaj v izhodišče, in omogoči vzvratno vožnjo. Zaščita pred integralskim pobegom regulatorja je implementirana preprosto tako, da akumulirana napaka ne more zapustiti vnaprej določenega intervala vrednosti med spodnjo in zgornjo omejitvijo akumulirane napake.

Za kot zavoja sem prav tako izbral PI regulator, vendar je že P regulator izkazoval zadovoljive rezultate, saj gre tu za integrirni proces (oz. tip I). Vhod v regulator je referenčni kot, izhod pa je kotna hitrost kota zavoja. Implementacija je enaka pravkar opisani, le z znatno manjšimi koeficienti za proporcionalni člen regulatorja k_P in za integracijski člen k_I .

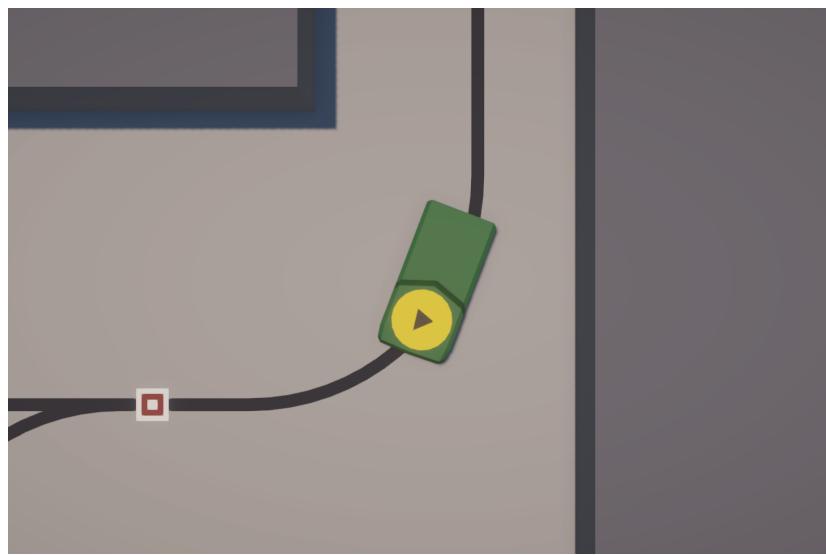
Spodaj je prikazan še izsek kode, ki to realizira v diskretnem prostoru. Ta se kliče ob vsaki izvedbi metode fiksne posodobitve. Najprej v tretji vrstici kliče metodo, ki je zaradi preglednosti prikaza ne vključujem. Ta preprosto skrbi, da se prejšnji vrednosti akumulirane napake integracijskega člena prišteje nova vrednost, nato pa vsoto omeji na interval, ki ga določa zaščita pred integralskim pobegom. Nato se izračuna nova vrednost navora, ki izhaja iz enabčbe 3.1. Napako v hitrosti pomnožimo s koeficientom proporcionalnega ojačenja, akumulirano napako pa s koeficientom integracijskega ojačenja. Njuna vsota je izhod regulatorja, ki se potem uporabi kot navor motorja. Podoben postopek ponovimo za kot zavoja. Razlika je le, da pri izračunu izhoda regulatorja kota zavoja še dodatno simuliramo vrtenje (numerično integriramo kotno hitrost, da dobimo nov kot).

```

1 void RegulateTelemetryValues()
2 {
3     regulator.AddErrors(telemetry.errorVelocity, telemetry.
4         errorSteering);
5     telemetry.realTorque = regulator.velocityP * telemetry.
6         errorVelocity + regulator.velocityI * regulator.
7         velocityErrorAccumulated;
8     telemetry.realSteering += (regulator.steeringP * telemetry.
9         errorSteering + regulator.steeringI * regulator.
10        steeringErrorAccumulated) * Time.fixedDeltaTime;
11 }
```

Koda 3.1: Odsek kode, ki skrbi za regulacijo motorja in zavoja robota.

Dodatno se na kot zavoja nastavi tudi kot rotacije indikatorja zavoja po svoji relativni y osi, viden na sliki 3.10 (puščica na prednjem delu vozila v rumenem krogu). Ta služi le enemu namenu: da uporabnik jasno vidi trenuten kot zavoja, kar olajša programiranje logike robota, saj so kolesa v pogledu z vrha skrita pod preostankom robota.

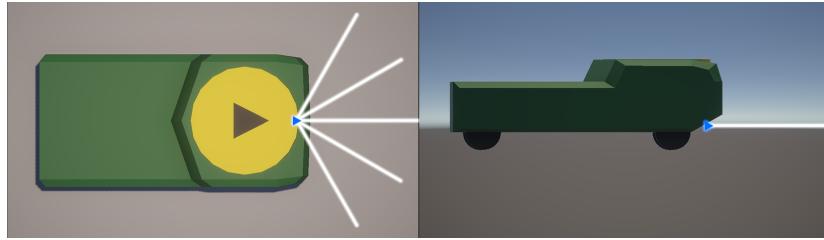


Slika 3.10: Robot med zavijanjem in viden indikator zavijanja..

3.6.2 Svetlobni merilnik razdalje

Robot je opremljen z simuliranim svetlobnim merilnikom razdalje Lidar. Sistem iz svoje izvorne točke periodično pošilja žarke in preverja, če kaj zadanejo. Izračun se ponovi ob vsaki izvedbi metode fiksne posodobitve, torej s frekvenco 100Hz . Na sliki 3.11 je prikazan izvor žarkov z modro puščico in žarki z belimi črtami. Žarki na sliki so zgolj za lažjo predstavo, v simulaciji niso vidni in imajo daljši domet. Izvor je postavljen na sredino sprednjega dela robota, na višino koles. Pošilja 5 žarkov v 30-stopinjskih inkrementih od -60° do $+60^\circ$, z dometom 20 metrov. Žarki so razporejeni po ravnini, ki je definirana z x osjo (smer desno) in z osjo (smer naprej) robota.

Za svoje delovanje uporablja funkcijo `Physics.Raycast()`, ki ji podamo izvor, smer in domet žarka. Dodatno lahko nastavimo, da izračun določene objekte



Slika 3.11: Lidar pogled iz ptičje in stranske perspektive.

prezre. Ta funkcija ustvari (matematičen, ne vizualen) žarek in preveri, če ta trči v katerega od modelov trka, aktivnih v simulaciji. Nato vrne rezultat v obliki `out RaycastHit` objekta, ki vsebuje podatke o zaznanem trku: točna pozicija in normala trka, razdalja od izvora žarka, zadeti model trka, ... Za izračun je najuporabnejša razdalja od izvora žarka, ostali našteti podatki pa so bili predvsem uporabni pri preverjanju pravilnosti delovanja kode in odpravljanju napak. Rezultate izračunanih razdalj vseh teh žarkov nato shranimo. V primeru, da žarek ne zadane ničesar, se shrani vrednost dometa (20 m). Po izračunu rezultatov vseh senzorjev te vrednosti (skupaj z rezultati ostalih senzorjev in drugih vrednosti) posredujemo modulu za vodenje v okolju Simulink.

Spodaj je naveden izsek kode, ki izračuna te vrednosti. Funkcija iterira čez vse žarke, ter njihove rezultate zapiše v zbirkko (ang. Array) imenovano `lidarValues`. V tretji vrstici se pojavi `transform.rotation`, ki je kvaternion, ki opisuje rotacijo robota. Z njim v pravo smer zavrtimo enotske vektorje, ki kažejo v smereh poslanih žarkov (`lidarForwards[i]`), ko je robot v svoji osnovni orientaciji.

```

1 void HandleLidar()
2 {
3     for (int i = 0; i < lidarCount; i++)
4     {
5         if (Physics.Raycast(lidarOrigin.position, transform.rotation *
6             lidarForwards[i], out hit, lidarRange))
7             lidarValues[i] = hit.distance;
8         else
9             lidarValues[i] = lidarRange;
10    }
}
```

Koda 3.2: Odsek, ki simulira vedenje senzorja Lidar.

3.6.3 Senzor za sledenje črti

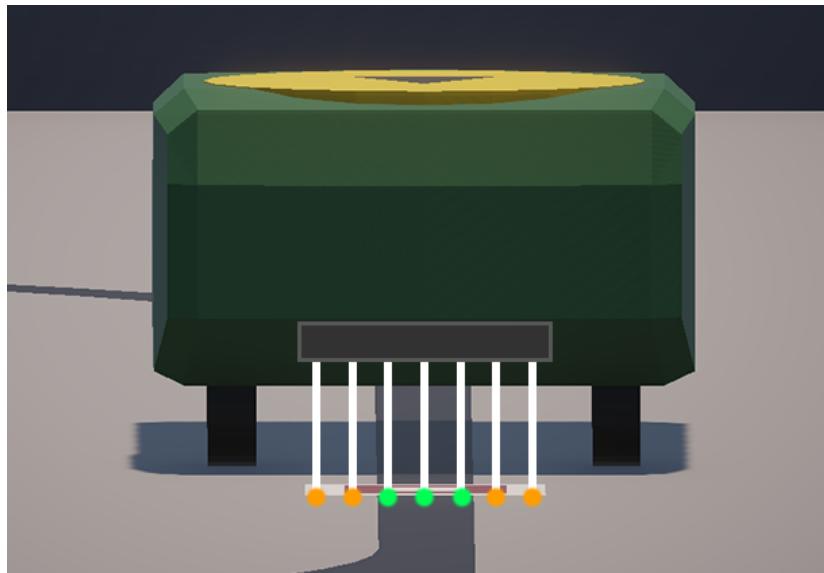
Robot simulira tudi senzor za sledenje črti. Ta je realiziran tako, da sistem na sedmih različnih točkah na isti daljici pod robotom, poravnani z x osjo robota, preveri, če je v smeri negativne y osi robota (spodaj) črta črnejše barve. Ponovno uporabljam funkcijo `Physics.Raycast()`, kot v poglavju 3.6.2, le da je tokrat dodatno definirana plast objektov (ang. Object Layer), nad katero se funkcija izvaja. Vsak od objektov v simulaciji ima definirano plast. Z njimi definiramo kombinacije katerih objektov lahko med seboj trčijo, in katere kombinacije trke prezrejo oz. jih niti ne preverjajo. V tem primeru uporabljam plasti tako, da funkcija preverja trk žarka le z objekti, ki so na pravilni plasti, imenovani sledenje črti (ang. Line Follow). Na tej plasti so vse črte, ki se jih da zaznati s tem senzorjem, torej vse črte črnejše barve. Črte rumene barve pa so, kot že omenjeno v poglavju 3.4, v programu zgolj dekorativne narave.

Na sliki 3.12 je vidna vizualizacija poslanih žarkov ene meritve. Zelene pike prikazujejo zaznano črto, oranžne pa odsotnost zaznave. Ker je pomembna le plast za sledenje črti, tudi marker na sliki ni v napoto, črta je zaznana na srednjih treh žarkih. Algoritem vodenja, razložen v kasnejšem poglavju, s pomočjo teh podatkov skrbi, da robot zavija tako, da zaznana črta ostaja na sredini (oz. robot na sredini črte).

3.6.4 Senzor za zaznavo markerjev

Robot je opremljen tudi z simuliranim senzorjem NFC markerjev. Z uporabo funkcije `Physics.OverlapSphere()` periodično preverjam prisotnost markerjev v sferičnem območju z središčem na dnu robota pod težiščem. Funkcija deluje na enak način kot predhodno omenjena funkcija `Physics.Raycast()`, z eno izjemo: za prisotnost modelov trka preverja v sferičnem območju, namesto vzdolž navigacijskega žarka. Vsi markerji so na svoji plasti, uporabljeni funkciji pa, podobno kot v poglavju 3.6.3, podamo to plast kot ciljno plast. Tako preverja le prisotnost relevantnih objektov, kar zmanjša računsko zahtevnost.

Pod tem odstavkom se nahaja odsek kode 3.3, ki zazna marker, če je le ta v dometu senzorja. V zbirko `foundTags` tipa `Collider[]` (model trka) shra-



Slika 3.12: Vizualizacija senzorja za sledenje črti.

nimo vse zaznane modele trka, nato pa izluščimo podatek, za kateri marker gre. Izluščimo njegovo zaporedno številko objekta v simulaciji (če smo korektni gre za zaporedno številko objekta znotraj hierarhije objekta, ki je starš (ang. parent) vsem objektom, ki jih uporabnik postavi v simulaciji), ter jo shranimo za uporabo pri iskanju poti v Simulinku. Številka ostane shranjena, dokler robot ne zazna novega markerja. V okolju Matlab se te vrednosti preslikajo v zaporedne št. markerjev.

```

1 void CheckForTag()
2 {
3     foundTags=Physics.OverlapSphere(tagReader.position, tagRange,
4                                     tagMask);
5     if (foundTags.Length!=0)
6         telemetry.lastMarkerDetected = foundTags[0].transform.
7             GetSiblingIndex();
}
```

Koda 3.3: Odsek kode, ki zazna marker in si ga zapomni do naslednje zaznave.

3.7 UDP komunikacija

Podatke o internih vrednostih robota, kot so pozicija, orientacija, hitrost in kota hitrost, ter rezultate opisanih senzorjev shranjujem v po meri narejenem razredu (ang. custom Class) poimenovanemu telemetrija. S tem pregledno zbiram podatke na enem mestu, iz kjer jih s funkcijo preprosto sporočam preko UDP povezave kontrolnemu algoritmu v okolju Simulink. To posredovanje informacij se, kot večina funkcij v simulaciji, izvaja periodično z $f = 100Hz$.

Za to komunikacijo skrbi samostojen skript. Na začetku definira spremenljivke, kot so zbirke za pošiljanje in prejemanje podatkov ter končne točke Internetnega protokola (ang. Internet protocol End point, v kodi okrajšano kot **IPEndPoint**), ki vsebujejo podatek o IP naslovu in vratih (ang. Port) oben členov komunikacije. Oba programa se nahajata na istem računalniku, tako da sta njuna IP naslova oba enaka **127.0.0.1** (lokalni gostitelj), razlikujeta se le v številki vrat, kjer izberemo dve nezasedeni UDP vrati. Pri izbiri vrat si lahko vedno pomagamo z ukazno vrstico, kjer z ukazom **netstat -a** pridobimo seznam vseh zasedenih UDP in TCP (Protokol prenosnega nadzora, ang. Transmission control protocol) vrat.

Nato skript inicializira dve novi programski niti (ang. Thread), eno za pošiljanje podatkov in eno za prejemanje. V prvem skriptu preprosto prebere ažurne podatke telemetrije in jih pošlje na končno točko IP, ki jo prebira Simulink. Pri prejemanju je potrebno vzpostaviti vtičnico (ang. Socket), jo vezati (ang. Bind) na končno točko IP, s katere Simulink pošilja podatke, in definirati medpomnilnik, kamor se bodo prejeti podatki zapisovali. Nato pa periodično prebiramo podatke iz definiranega medpomnilnika, kjer jih želimo uporabljati v programu.

Tu je vredno opozoriti, da z uporabo UDP ne preverjamo, če je drugi člen v komunikacijski verigi prejel poslane podatke, na to se preprosto zanašamo. V aplikacijah, kjer je pomembno, da je druga stran prejela poslane podatke, je pametnejše uporabljati predhodno omenjeni TCP, kjer je preverba integralni del uporabe protokola. Za namene simulacije, kjer v primeru odpovedi komunikacije ne bo poškodb oz. materialne škode, nam zadošča UDP. Njegova prednost in razlog za mojo izbiro protokola je predvsem hitrost.

3.8 Programsko okolje Matlab-Simulink

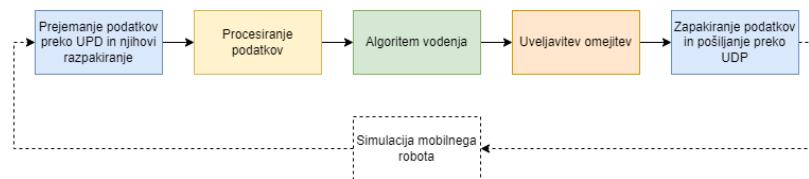
Matlab je programska in numerična računska platforma, ki je zelo razširjena med inženirji in znanstveniki. Omogoča zajemanje in analizo podatkov, modeliranje in simulacijo procesov, ter izvajanje meritev: na simuliranih veličinah in na resničnih, zajetih v realnem času. Platforma vsebuje tudi okolje za vizualno programiranje matematičnih modelov z bločnimi diagrami Simulink.

V simulaciji uporabljam Matlab za izvedbo inicializacijske kode, ki ob koncu izvedbe zažene Simulink model. Najprej inicializira spremenljivke, ki jih Simulink potrebuje za uspešno komunikacijo preko UDP, podatki o omejitvah hitrosti, omejitvah kota zavoja in podatke o postavitvi objektov v izbranem skladišču. Na sliki 3.13 je viden primer zemljevida, ki ga Matlab ustvari iz znanih podatkov o postavitvi objektov. Gre za isto skladišče po meri iz slike 3.8. Če je za zagon izbran Simulink model, kjer se uporablja iskanje poti (ang. Pathfinding), se na tem koraku tudi izračuna optimalna pot z algoritmom A zvezdica (ang. A-star, znan po oznaki A*).



Slika 3.13: Primer zemljevida skladišča, ki ga Matlab ustvari ob zagonu.

Ko je inicializacija izvedena, in simulacija mobilnega robota teče v ozadju, lahko zaženemo model. Na sliki 3.14 je prikazan poenostavljen diagram poteka glavne zanke. Podatke, ki jih prejmemo preko UDP preberemo in pretvorimo v uporabne podatke ter jih uporabimo v algoritmu vodenja. Rezultati tega algoritma (torej želena hitrost robota in kot zavoja) se po potrebi omejita na mejne dovoljene vrednosti, pretvorita v ustrezni zapis in pošljeta v simulacijo. Dodatno se pošilja tudi stanje reset gumba, ki uporabniku omogoča ponastaviti simulacijo v obeh okoljih sočasno. Ta zanka se ponavlja, dokler simulacije ne ustavimo. Algoritem vodenja bo podrobnejše razložen v kasnejših poglavjih.



Slika 3.14: Diagram poteka glavne zanke osnovnega Simulink modela.

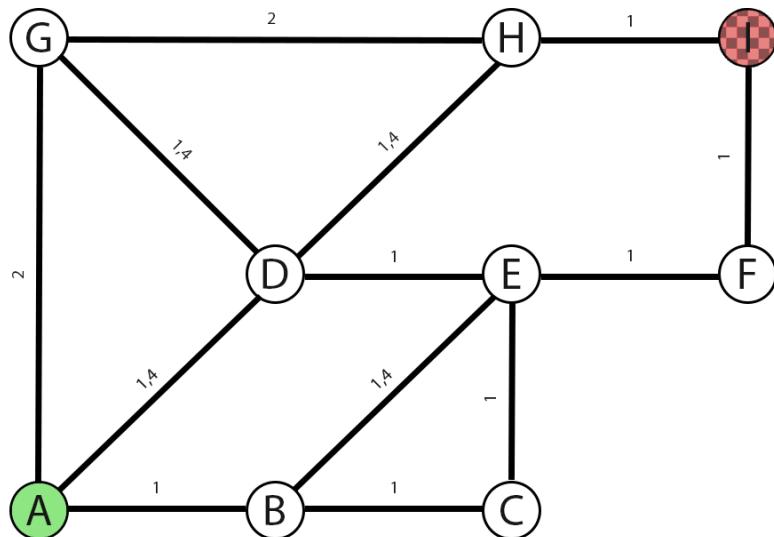
4 Iskanje poti

Algoritmi iskanja poti (ang. pathfinding algorithms) so integralen člen pri vodenju mobilnega robota. Njihov namen je najti optimalno pot med vsaj dvema točkama, nato pa to pot sporočiti algoritmu vodenja mobilnega robota. V osnovi se za optimalno pot smatra najkrajša pot, lahko pa bi uporabljali naprednejše kriterije. Pogosto je smiselno še upoštevati zasedenost poti, ki jo povzročijo drugi roboti v skupnem prostoru, preferenco za nekatere poti in izogibanje drugih (pot je lahko najkrajša skozi nek prostor, kjer ne želimo mobilnih robotov, razen nujno potrebnih).

Ne glede na izbrane kriterije, je potrebno zasnovati cenilno funkcijo, ki nam pove 'ceno' načrtovane poti. Nato poiščemo pot, kjer je rezultat cenilne funkcije najmanjši. Skupaj z izrazom cenilne funkcije je vredno omeniti še hevristične funkcije, saj gre za sorodna izraza. Hevristična funkcija je funkcija, ki poda oceno verjetnosti, da bo določeno vozlišče (ponavadi možne poti iz križišč, ang. nodes) vodilo do cilja. Tipično se tu uporabi evklidsko razdaljo med vozliščem (ponavadi pozicija njegove začetne točke) in ciljem. Poda grobo oceno, s pomočjo katere optimiziramo algoritem za iskanje poti. Če iterativno vedno preverjamo odsek poti z najnižjo hevristično vrednostjo, bo prva pot, ki jo bomo našli, tudi najbolj optimalna pot. Ko to pot najdemo, lahko prenehamo z iskanjem, in ne preverjamo vseh možnih poti. Tako je na primer zastavljen algoritem A* (prebrano kot A-zvezdica, ang. A-star), ki smo ga implementirali v sklopu tega dela.

Podatke o razporeditvi poti in povezav izbranemu algoritmu posredujemo v podatkovni strukturi grafa. Gre za zbirkovo podatkov, kjer vsak vnos vsebuje identifikacijsko številko danega vozlišča, naslednja vozlišča, in cene poti do teh vozlišč

iz pripadajočega vozlišča. Ko imamo okolje z ovirami ustrezno predstavljeno z grafom, lahko uporabimo enega izmed algoritmov, ki poišče pot od začetne do ciljne konfiguracije. V splošnem začnemo iskanje tako, da preverimo, ali je začetno vozlišče hkrati tudi ciljno vozlišče. Ponavadi to ne drži, zato razširimo iskanje na vozlišča, ki sledijo sedanjemu. Na podlagi izbranega algoritma izberemo enega od njih, ter postopek ponavljamo, dokler ne najdemos rešitve, ali dokler ne preiščemo celotnega grafa [9]. Preprost primer takega grafa je viden na sliki 4.1. S črkami so označena vozlišča, s črtami poti med njimi, s številkami pa cene pripadajočih segmentov poti med dvema vozliščema v arbitrarni enoti.



Slika 4.1: Primer strukture grafa v sistemu z mobilnimi roboti za namene iskanja poti.

Raziščimo zgornje koncepte s pomočjo preprostega primera: vzemimo pot, ki se začne v vozlišču A in konča v vozlišču I (na sliki sta pobrvana), želimo od našega algoritma, da najde najkrajšo pot, ki gre skozi vozlišča A, D, H in I. Kljub temu, da je možnih več poti med vozliščema nas slabše alternative (npr. pot skozi vozlišča A, G, H, I) ne zanimajo. V primeru na sliki so cene segmentov izračunane na podlagi razdalj med vozlišči.

Da ne bomo ostali na tako preprostem primeru predpostavimo, da je pot med vozlišči A in D zelo prometna, saj mnogi roboti v sistemu potujejo med temi

dvema vozliščema. Posledično pogosto prihaja do zastojev in trkov, zaradi česar želimo pot nekoliko razbremeniti. Cenilno funkcijo našega algoritma nadgradimo tako, da razdaljo pomnožimo še s koeficientom zasedenosti, ki je za povezavo med vozlišči A in D enak $k_{AD} = 2$, ostale razdalje pa je enak $k_z = 1$. Nova cena poti med vozlišči A in D je sedaj 2,8. Tako preusmerimo robote po drugih poteh v primeru, da jim ta obvoz ne predstavlja znatno daljše poti. Posledično nekoliko razbremenimo odsek poti za ostale robote. Izračunana optimalna pot med vozlišči A in D tako ostaja enaka, optimalna pot med vozlišči A in I pa sedaj potuje skozi vozlišča A, B, E, F, I in se izogne zastojem ter jih nekoliko omili. Te koeficiente zasedenosti bi lahko določali tudi dinamično, glede na trenutne aktivne poti robotov.

Od našega algoritma za iskanje optimalne poti torej želimo, da je zmožen priti do enakih ugotovitev, vendar brez naše pomoči. Pot do tega rezultata pa računalnikom ni tako samoumevna, kot je nam ljudem. Posledično so tovrstni algoritmi pogosta tema znanstvenih raziskav, ki so skozi leta pripeljala do mnogih rešitev. Nekaj pomembnih izvedb je razloženih v sledečih poglavjih.

4.1 Algoritem Dijkstra

Dijkstrov algoritem je algoritem iskanja poti, ki preišče celoten graf (ang. Graph search pathfinding algorithm). Preišče vse možne poti od izbranega začetka do konca, ter izbere optimalno. Algoritem uporablja dva seznama vozlišč: seznam raziskanih vozlišč in seznam vozlišč, ki še bodo raziskana. Takšna seznama sta tipična za mnoge tovrstne algoritme. Imenujemo jih zaprti set in odprtji set vozlišč. Ob zagonu algoritma se začetno vozlišče doda na zaprti set, vsa vozlišča, ki jih je možno od tam doseči (le sosednja, ne vsa nadaljna možna), pa se dodajo na odprtji set.

Algoritem nato ponavlja sledeče korake: Z odprtrega seta izbere vozlišče z najmanjšo ceno poti od izhodišča in ga prestavi v zaprti set. V odprtji set doda vsa sosednja vozlišča, ki jih je možno doseči iz izbranega vozlišča, če zanj še ne obstaja cenejša pot do tega vozlišča v zaprtem setu (v nasprotnem primeru posodobi star vnos). Za vsakega izračuna ceno poti od izbranega vozlišča, ji

prišteje predhodno ceno poti do izbranega vozlišča (iz izhodišča) in ta podatek shrani. Za vsako vozlišče tudi shrani vozlišče, iz katerega je bilo doseženo. Ta postopek se ponavlja, dokler v odprttem setu ne zmanjka vozlišč.

Ko jih zmanjka vemo, da smo raziskali celoten graf (oz. vsa vozlišča v grafu, ki so dosegljiva iz izhodišča). Kot rezultat algoritom vrne pot, ki ima najmanjšo skupno ceno poti. To stori tako, da iz vnosa končnega vozlišča prebere vozlišče, iz katerega je bil najden. To ponavlja za vsa nadaljna vozlišča, dokler ne pride algoritom nazaj do izhodišča. Sedaj imamo podatke o najkrajši poti, vendar v nasprotnem vrstnem redu, saj je končno vozlišče prvo na seznamu. Vrstni red preprosto obrnemo in dobimo končni rezultat algoritma.

4.2 Algoritem A*

Algoritem A* je nadgradnja Dijkstrovega algoritma. Razlikujeta se v tem, da je A* pri kompleksnejših grafih znatno bolj učinkovit, saj ne potrebuje preiskati vseh možnih poti. Prva pot, ki jo najde, je že optimalna. Na tej točki se iskanje zaključi, saj ne more najti boljše rešitve. To algoritem doseže z dodatkom hevristike, ki v posameznih vozliščih podaja oceno verjetnosti, da bo trenutno vozlišče vodilo do optimalne rešitve. Tipično se za hevristično funkcijo uporabi evklidsko razdaljo med vozliščem in ciljem.

Algoritem ponovno uporablja odprti in zaprti set vozlišč. Ponovno da prvo vozlišče v zaprti set in njegova sosednja vozlišča v odprtih set. Vsakič, ko algoritem doda vozlišče v odprtih set, izračuna tri pripadajoče cene: G cena, H cena in F cena. G cena je enaka ceni iz Dijkstrovega algoritma: gre za ceno giba od izhodišča do izbranega vozlišča. H cena je hevristična ocena cene giba od izbranega vozlišča do cilja. Kot že omenjeno, tukaj uporabljamo evklidsko razdaljo med vozliščema. F cena je skupna cena vozlišča, in je enaka vsoti njegove G cene in H cene.

Kot v poglavju 4.1 tudi tukaj algoritem iterativno preverja vozlišča v odprttem setu in jih prestavlja v zaprti set. Razlika je, da pri A* algoritmu vedno izberemo tisto vozlišče v odprttem setu, ki ima najnižjo F ceno (torej vsoto G in H cene), saj smatramo, da ima to vozlišče najboljšo možnost, da pripelje do cilja. Izbrano vozlišče torej prestavimo v zaprti set, njegove sosedne pa v odprtega in izračunamo

pripadajoče cene. Podobno kot pri Dijkstrovem algoritmu, tudi tukaj posodobimo že izračunano vozlišče, če najdemo boljšo pot do njega (z nižjo G ceno). Ko pridemo do cilja, lahko zaključimo z iskanjem, saj smo zagotovo našli optimalno pot.

Algoritom A* je med najpogosteje uporabljenimi algoritmi v mnogih aplikacijah, od robotike do računalniških iger. Smatra se za odličen algoritmom z radi svoje računske učinkovitosti in hitrosti, njegova glavna pomankljivost pa je količina spomina, ki jo izvedba potrebuje [10].

4.3 Algoritma iskanje v širino in iskanje v globino

Algoritma iskanje v širino (ang. Breadth-first Search, znan pod kratico BFS) in iskanje v globino (ang. Depth-first Search, znan pod kratico DFS) sta algoritma neinformiranega iskanja. Različni odseki poti imajo enako ceno ne glede na njihovo dolžino in druge parametre, za razliko od Dijkstrovega algoritma iz poglavja 4.1 in algoritma A* iz poglavja 4.2.

Algoritma ponovno začneta z odprtim setom in zaprtim setom. V zaprtega dodamo začetno vozlišče, v odprtega pa sosednja vozlišča, ki jih lahko iz začetnega dosežemo. Iterirata čez odprti set in prestavlja vnose v zaprtega. Algoritma se razlikujeta po metodi izbire vozlišč iz odprtega seta za preiskavo.

Iskanje v širino izbira vozlišča po principu 'prvi noter, prvi ven' (ang. first in, first out, znan pod kratico FIFO), torej razišče vozlišča v takem zaporedju, kot so bila dodana v odprti seznam. Posledično preiskuje graf v zaporedju, ki ga definira oddaljenost vozlišča od začetnega v številu korakov oddaljenosti. Najprej preišče vsa vozlišča, ki so začetnemu sosednja. Nato preišče vsa njihova sosednja vozlišča in tako dalje, dokler ne najde rešitve. Rešitev, ki jo najde, potrebuje najmanj korakov od začetnega vozlišča do končnega. Algoritom torej zagotovo najde optimalno pot le v primeru, ko imajo odseki poti med posameznimi vozlišči enako ceno.

Iskanje v globino pa izbira vozlišča z odprtega seta po principu 'zadnji noter, prvi ven' (ang. last in, first out, znan pod kratico LIFO). Vozlišča raziskuje

tako, da preizkuša celotne poti eno za drugo. Algoritem tako ne najde nujno optimalne poti, niti ne poti z najmanjšim številom korakov, niti ne garantira, da bo rešitev našel (v primeru neskončnih razvezjitev oz. cikličnih poti se lahko zatakne). Njegova glavna prednost je, da porabi znatno manj spomina, kot iskanje v širino, saj mora shranjevati le trenutno raziskano pot, in vmesne neraziskane razvezjitve.

4.4 Implementacija algoritma A*

Za uporabo skupaj s simulacijo, opisano v poglavju 3, in vodenjem, opisanem v poglavju 5, smo v okolju Matlab implementirali algoritem A*. V tem poglavju je predstavljena njegova implementacija. Na sliki 4.2 je prikazan diagram poteka implementiranega algoritma.

Algoritem je zastavljen kot funkcija, ki jo kličemo iz skripte za inicializacijo vodenja. Začnemo z definicijo funkcije in inicializacijo spremenljivk, kar je prikazano v odseku kode 4.1. Podamo ji številko začetnega in končnega vozlišča, ter seznam vseh vozlišč. Vsak vnos vozlišča vsebuje njegov ID, naslednja vozlišča in cene poti do njih (iz tega vozlišča) ter vektor pozicije. Vsebujejo tudi še neizpolnjene, vendar inicializirane podatke o G ceni, H ceni in F ceni, ter ID prejšnjega vozlišča.

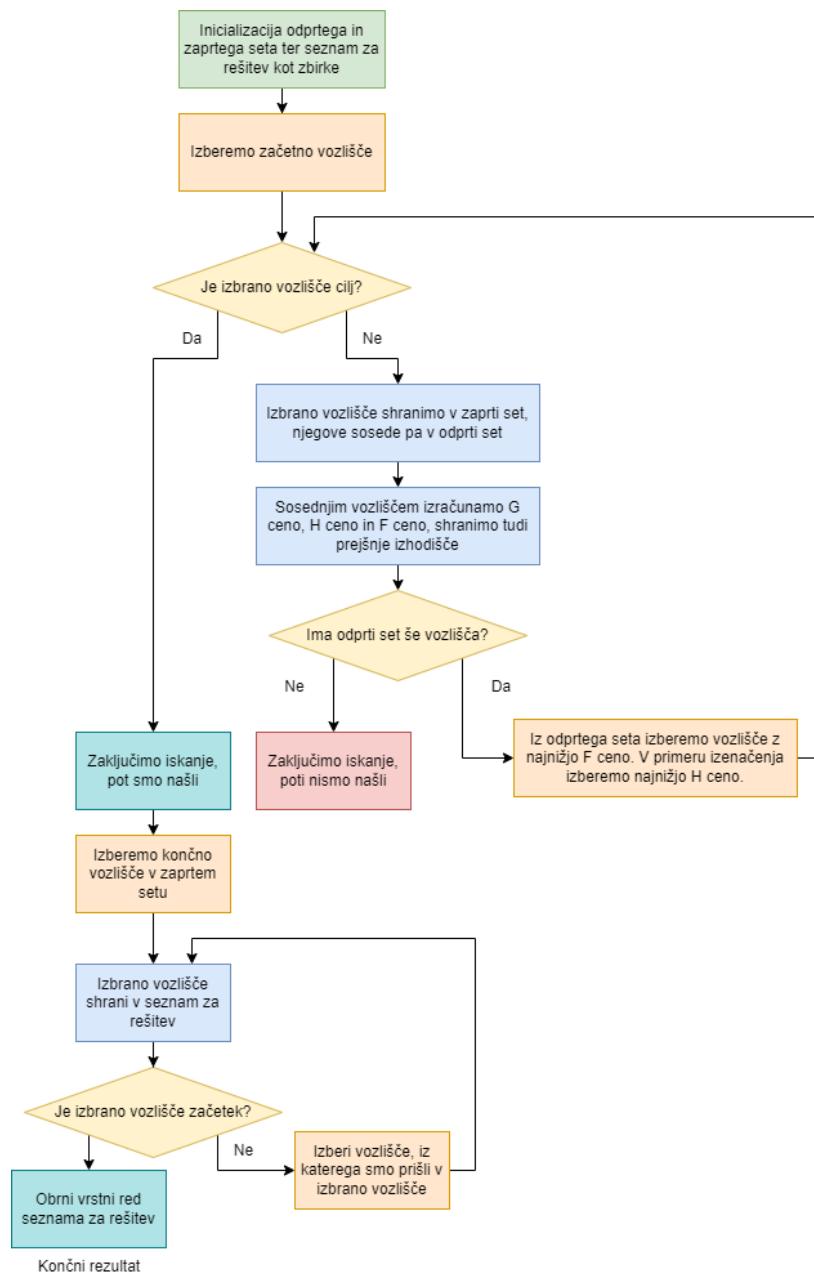
```

1 function actionSet=Astar(startTag,endTag,nodes)
2 clear openNodes;
3 clear closedNodes;
4 endNode=nodes(endTag,:);
5 startNode=nodes(startTag,:);
6 openNodes(1)=startNode;
7 closedNodes=[];

```

Koda 4.1: Inicializacija spremenljivk in zbirka, ki bodo uporabljene v algoritmu.

Nato vstopimo v glavno zanko in poiščemo vozlišče v odprtem setu z najmanjšo F ceno. Definiramo spremenljivko najnižje najdene F cene, in jo nastavimo na nedosegljivo visoko vrednost. Nato v zanki preverimo za vsako odprto vozlišče, če je njegova F cena manjša od trenutne najnižje vrednosti. Če je, posodobimo najnižjo vrednost, in si zapomnimo ID tega vozlišča.



Slika 4.2: Diagram poteka implementacije algoritma A* v sklopu dela.

Po izvedbi zanke je shranjeni ID vozlišča z najnižjo F ceno. Če najdemo več vozlišč z najnižjo F ceno potem po istem sistemu najdemo med njimi še vozlišče z najnižjo H ceno. Nato preberemo podatke o najdenem vozlišču. Spodaj je prikazan odsek kode 4.2, ki poskrbi za izbiro pravilnega vozlišča v vsaki iteraciji glavne zanke.

```

1 while (size(openNodes ,2) ~=0)
2     lowestFcost=9999999999999999;
3     lowestFIdx=1;
4     for i=1:size(openNodes ,2)
5         if (openNodes(i).Fcost<lowestFcost)
6             lowestFcost=openNodes(i).Fcost ;
7             lowestFIdx=i;
8         end
9     end
10    currentNode=openNodes(lowestFIdx);
```

Koda 4.2: Izbira pravilnega vozlišča za evalvacijo.

Na tej točki preverimo, če se nahajamo v končnem vozlišču. Če se, lahko izstopimo iz zanke, drugače pa preberemo nadaljna vozlišča, ki jih je možno doseči iz trenutno izbranega vozlišča. Izračunamo vse tri cene vsakega novega vozlišča in jih skupaj s podatki o vozlišču ter ID številko prejšnjega vozlišča shranimo v odprt set. Odsek kode 4.3 to storí za vozlišče, ki ga dosežemo, če se v trenutno izbranem vozlišču držimo levega roba črte. Enako storimo še za vozlišče po desnem robu črte, in vozlišče, ki ga dosežemo, če se držimo sredine črte (to uporabljamo za odseke brez križišč, saj robot lažje sledi sredini črte, kot njenemu robu). Ker so odseki kode med seboj praktično enaki, je prikazan le eden.

```

1 if (currentNode.tag ~=endTag)
2     if (currentNode.nextL ~= -1)
3         nextNode=nodes(currentNode.nextL);
4         nextNode.Gcost=currentNode.Gcost+currentNode.weightL;
5         nextNode.Hcost=GetNodeDistance(nextNode ,endNode );
6         nextNode.Fcost=nextNode.Gcost+nextNode.Hcost ;
7         nextNode.from=currentNode.tag;
8         openNodes=[openNodes ,nextNode];
9     end
10 end
```

Koda 4.3: Evalvacija vozlišča.

Nato prestavimo trenutno obravnavano vozlišče iz odprtega seta v zaprti set, za kar poskrbi kratki odsek kode 4.4. Dodatno zaključimo še glavno zanko, ki smo jo začeli dva odseka kode nazaj.

```

1 openNodes=openNodes([1:lowestFIdx-1 lowestFIdx+1:end]);
2 closedNodes=[closedNodes , currentNode];
3 end

```

Koda 4.4: Prestavitev vozlišča v zaprti set.

Sedaj moramo le še najti pot nazaj do začetka skozi zaprti set, z uporabo podatka o vozlišču, iz katerega smo dosegli drugo vozlišče. Definiramo nov seznam, in vanj shranimo zadnje vozlišče. Nato preberiramo prejšnja vozlišča, dokler ne pridemo do izhodiščnega vozlišča, te podatke pa shranujemo v seznam. Spodnji odsek kode 4.5 doseže opisano v tem odstavku.

```

1 clear pathBackwards
2 pathBackwards(1)=[currentNode];
3 searchableClosed=transpose(squeeze(cell2mat(struct2cell(
    closedNodes))));
4 fromIdx=find(searchableClosed(:,2)==currentNode.from);
5 fromNode=closedNodes(fromIdx(1));
6 pathBackwards=[pathBackwards , fromNode];
7 while (fromNode.from ~= -1)
8     fromIdx=find(searchableClosed(:,2)==fromNode.from);
9     fromNode=closedNodes(fromIdx(1));
10    pathBackwards=[pathBackwards , fromNode];
11 end

```

Koda 4.5: Odsek kode, ki najde pot od cilja nazaj do izhodišča po raziskanih vozliščih.

Te podatke nato zapakiramo v set akcij, ki vsebujejo le podatek o vozlišču in načinu sledenja črti (sledenje sredini, levemu ali pa desnemu robu), ki ga mora v tem vozlišču robot uporabljati, da pride do naslednjega. Spodaj se nahaja zadnji odsek kode 4.6 v tem poglavju, ki to doseže. Iskanje poti je na tej točki zaključeno. Set akcij nato posredujemo algoritmu vodenja v okolju Matlab-Simulink, ki jih uporabi pri izbiri načina sledenja črti.

```

1 actionSet=[];
2 totalActions=size(pathBackwards ,2);
3 for i=1:totalActions

```

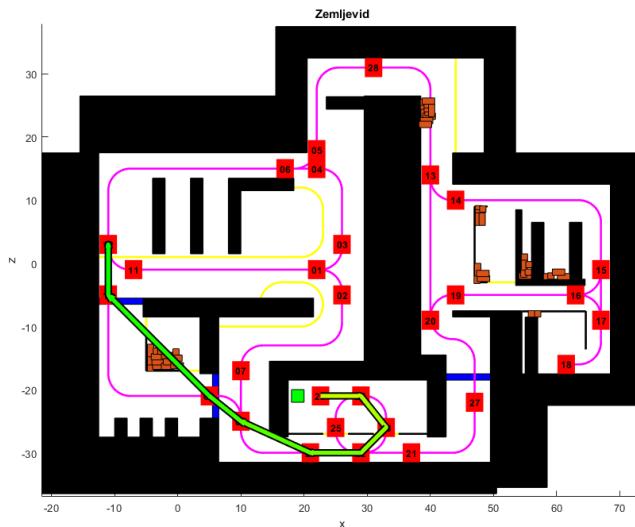
```

4     currentNode=pathBackwards(totalActions-i+1);
5     instruction=-1;
6     if (i<totalActions)
7         nextNode=pathBackwards(totalActions-i);
8         switch nextNode.tag
9             case currentNode.nextL
10                instruction=1;
11            case currentNode.nextM
12                instruction=2;
13            case currentNode.nextR
14                instruction=3;
15            otherwise
16                instruction=-1;
17        end
18    end
19    actionSet=[actionSet;currentNode.tag,instruction];
20 end

```

Koda 4.6: Tvorjenje seta akcij za algoritmom vodenja.

Na sliki 3.13 iz poglavja 3.8 so vozlišča označena z rdečimi kvadrati. Če za primer izračunamo pot z izhodiščem v vozlišču z ID številko 29 in s koncem v vozlišču z ID številko 12, se izračuna pot, ki je na sliki 4.3 označena z zeleno barvo. Puščice le povezujejo vpletena vozlišča in ne prikazujejo dejanske trajektorije.



Slika 4.3: Rezultat A* algoritma, prikazan na zemljevidu skladišča.

5 Vodenje

Vodenje sistemov je odprtozančno ali zaprtozančno vplivanje na (realni) proces z namenom, da dosežemo želene cilje oz. želeno vedenje procesa [11]. Je ključen element v vsakem postopku avtomatizacije sistema. Vodene sisteme obravnavamo tako, da skušamo, če je le možno, razdvojiti mehanizme delovanja realnega sistema in sistema vodenja. Osnovni sistem se vedno nahaja v okolju. Okolje deluje na sistem in obratno. Sistem vodenja dobiva informacije o sistemu in okolju ter iz teh informacij določa krmilne ali regulirne signale [12].

V poglavju so predstavljene nekatere temeljne naloge vodenja mobilnih robotov in njihove implementacije v sklopu magistrskega dela. Kot že omenjeno v prejšnjih poglavjih, je vodenje implementirano v okolju Matlab-Simulink. Osnovna logika modela je bila že prikazana v poglavju 3.8 in prikazana z diagramom poteka na sliki 3.14. Sledče implementacije torej vstavljam v blok za vodenje, kar nato vpliva na vedenje mobilnega robota v simulaciji. Služijo bolj kot primeri posameznih implementacij, ne pa kot celota z možnostjo menjave med različnimi algoritmi.

5.1 Vodenje v točko

Med najosnovnejšimi algoritmi v mobilni robotiki je vodenje robota v točko. Edina zahteva je, da robot pride v referenčno pozicijo, orientacija robota pa ni predpisana. Da zadostimo zahtevi, moramo smer gibanja voditi v smer proti cilju, hitrost gibanja pa s približevanjem cilju zmanjševati. Zato je potrebno izračunati napako v smeri gibanja robota in razdaljo do cilja. Napako smeri uporabimo nato za referenčni kot zavoja za robotov regulator, ki se izvaja v sami simulaciji,

razdaljo do objekta pa uporabljamo za določitev reference hitrosti za robotov hitrostni regulator.

Napako smeri izračunamo iz orientacije robota po y osi (naj opomnim, da je y os v okolju Unity navpična, robot pa se vozi po ravnini, ki jo definirata osi x in z) in razlike v pozicijah med robotom in izbrano ciljno točko. Podatke o orientaciji in poziciji robota prejmemmo iz simulacije, pozicijo cilja pa imamo definirano v modelu vodenja v okolju Matlab-Simulink. Na sliki 5.1 je vidna skica za lažjo vizualizacijo pri razlagi.

Najprej izračunamo vektor razlike \vec{d} v poziciji med ciljem in robotom. Nato lahko izračunamo kot φ_t med x osjo simulacije in vektorjem \vec{d} (njegove komponente označujem z Δx in Δz) po enačbi: $\varphi_t = \text{atan2}(\Delta z / \Delta x)$. φ_r je kot med lokalno z osjo objekta robota (na skici os modre barve, vedno kaže naravnost naprej) in x osjo simulacije, torej y rotacija robota, ki jo preko UDP že dobimo iz simulacije. Napako kota zavoja izračunamo kot razliko teh dveh kotonov: $\varphi_e = \varphi_t - \varphi_r$. Podatek je potrebno iz intervala $(-360^\circ, +360^\circ)$ preslikati na ekvivalenten kot na intervalu $(-180^\circ, +180^\circ)$, saj bodo koti, ki presegajo 180° v pozitivno ali negativno smer, v regulatorju napačno interpretirani, kar bo povzročilo nesmiselno vedenje robota. Za to poskrbi odsek kode 5.1. Ta podatek nato omejimo na interval $(\varphi_{min}, \varphi_{max})$ ter ga pošljemo v simulacijo.

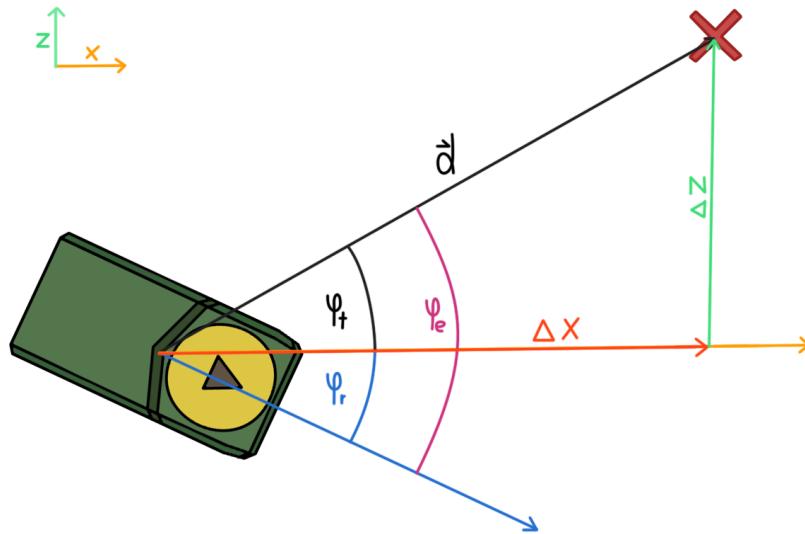
```

1 function angle=Handle180Transition(angleIn)
2
3 angle=angleIn;
4 if (angleIn>180)
5     angle=angleIn-360;
6 end
7 if (angleIn<-180)
8     angle=angleIn+360;
9 end
10 end

```

Koda 5.1: Odsek Matlab kode v funkciskem bloku, ki poskrbi za pravilen preskok med -180° in $+180^\circ$

Glede na dolžino vektorja razlike $|\vec{d}|$ v poziciji med ciljem in robotom zmanjšujemo hitrost robota, ko se približuje cilju. To vrednost lahko preprosto podamo regulatorju in robot bo vozil hitreje, ko bo daleč od cilja, in pospeševal,



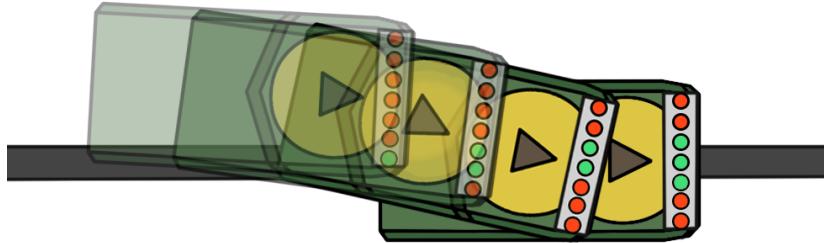
Slika 5.1: Skica veličin pri vodenju robota v točko.

ko se mu bo približeval. To nadgradimo tako, da razdaljo pomnožimo z izbranim koeficientom, in rezultat omejimo na poljuben interval. Namesto razdalje bi lahko uporabljali njen kvadrat ali drugo potenco, da dosežemo želeno vedenje robota.

Potrebeno je definirati še stopnjo natančnosti d_a : razdaljo do cilja, pod katero smatramo, da je bil cilj dosežen. Če bo ta razdalja prevelika, se bo robot ustavil pred ciljem. Če pa bo premajhna, robot ne bo nikoli zares dosegel svojega cilja. Ko torej velja neenakost $|\vec{d}| < d_a$, hitrost preprosto nastavimo na 0, ne glede na prejšnji izračun.

5.2 Vodenje po črti

Algoritem vodenja po črti oz. sledenja črti mora 7 (število žarkov senzorja) binarnih podatkov (žarek je zaznal oz. ni zaznal črte), ki jih vrne senzor za sledenje črti, opisan v poglavju 3.6.3, pretvoriti v kot zavoja robota in po potrebi vplivati na njegovo hitrost. Če črta ni zaznana na pravem mestu (tipično sredina senzorja) mora algoritem ustreznno popraviti smer gibanja robota. Ob večjih odstopanjih dodatno še zmanjšamo hitrost, da se robot lažje poravnava nazaj na črto. Na sliki 5.2 je viden prikaz premikanja robota pri izvedbi algoritma s statično sliko.



Slika 5.2: Prikaz vedenja robota pri izvedbi algoritma za vodenje po črti (dimenzije so večje od dejanskih za boljšo preglednost).

V kodi 5.2 na začetku inicializiramo vse potrebne spremenljivke in pretvorimo podatek, ki ga prejmemmo od senzorja, v zbirko sedmih binarnih vrednosti. Podatek je iz simulacije poslan v obliki zapisa decimalnega števila s plavajočo vejico (ang. float), združen v eno samo število. Ta sistem uporabljamo, ker večina podatkov, ki se posiljajo preko UDP, potrebujejo to obliko zapisa, zato nismo implementirali ločene rešitve za zbirko binarnih vrednosti. Zapisani so kot 7 mestno število oblike $a_0a_1a_2a_3a_4a_5a_6$, kjer vsak a_i predstavlja vrednost 0 ali 1 zaznave pripadajočega poslanega žarka. Torej, da izluščimo posamezni podatek, je potrebno prebrati posamezno števko iz tega števila, za kar poskrbijo vrstice 4-9 v spodnjem odseku kode.

```

1 function [r,pos,notFound,leftEdge,rightEdge] = fcn(raw_scan,
2           rayDistance)
3 leftEdge=0;
4 rightEdge=0;
5 raw_scan=round(raw_scan);
6 result=arrayfun(@(x) mod(floor(raw_scan/10^x),10),floor(log10(
7           raw_scan))-1:0);
8 if length(result)<7
9     tempZeros=zeros(1,7-length(result));
10    result=[tempZeros,result];
11 end
12 r=zeros(1,7);
13 pos=0;
14 count=0;
```

Koda 5.2: Odsek Matlab kode v funkcijskem bloku (1. del), ki poskrbi

za inicializacijo potrebnih spremenljivk, in pretvorbo podatka senzorja v individualne rezultate žarkov.

Sedaj, ko imamo pripravljene podatke, lahko v zanki na začetku odseka kode 5.3 seštejemo vse odmike žarkov od središča senzorja, ki so zaznali črto, in jih prestejemo. Dodatno sledimo temu, kateri žarek je prvi, ki je zaznal črto, in kateri žarek je zadnji. Odmik prvega žarka, ki je zaznal črto, je enak odmiku levega roba zaznane črte, odmik zadnjega pa je enak odmiku desnega roba. Ko izstopimo iz zanke, vsoto odmikov zaznav delimo s številom zaznav, da dobimo povprečni odmik zaznav. Ta vrednost je zaznana sredina črte. Dodatno preverimo še, če je vsaj en žarek zaznal črto. Glede na ta podatek lahko prilagodimo vodenje v primeru, da popolnoma izgubimo črto pod senzorjem.

```

1 for ii=1:7
2     r(ii)=result(ii);
3     currPos=(-4+ii)*rayDistance;
4     pos=pos+r(ii)*currPos;
5     count=count+r(ii);
6     if (r(ii) ~=0)
7         if (count==1)
8             leftEdge=currPos;
9         end
10        rightEdge=currPos;
11    end
12 end
13 if count>0
14     pos=pos/count;
15 end
16 if count==0
17     notFound=1;
18 else
19     notFound=0;
20 end
```

Koda 5.3: Odsek Matlab kode v funkcijskem bloku, ki izračuna vrednosti za regulacijo sledenja črti: individualne rezultate posłanih žarkov, izračunana sredina zaznane črte, podatek o tem, če je bila črta sploh zaznana, in zaznani odmiki levega in desnega roba.

Robot lahko v danem trenutku sledi črti na enega od treh načinov: lahko sledi levemu robu črte, njeni sredini, ali pa desnemu robu črte. Odvisno od izbire izberemo pripadajoči odmik, izračunan v kodi 5.3. Če ne gre za sredinsko poravnavo dodatno prištejemo oz. odštejemo vrednost d_r , kjer je d_r razdalja med dvema sosednjima žarkoma, tako, da zamaknemo center zaznave proti izbrani smeri. To storimo, da kompenziramo za debelino črte, ki je ob popolnem centriranju zaznana na srednjih treh poslanih žarkih. Ob sredinski poravnavi to ni potrebno, saj jemljemo povprečje zaznanih odmikov, pri poravnavi ob rob črte pa bi robot brez tega popravka sledil sredini črte, zamaknjeni za razdaljo d_r . Ta vrednost se nato ojača in pošlje kot referenčni kot zavoja regulatorju v simulaciji.

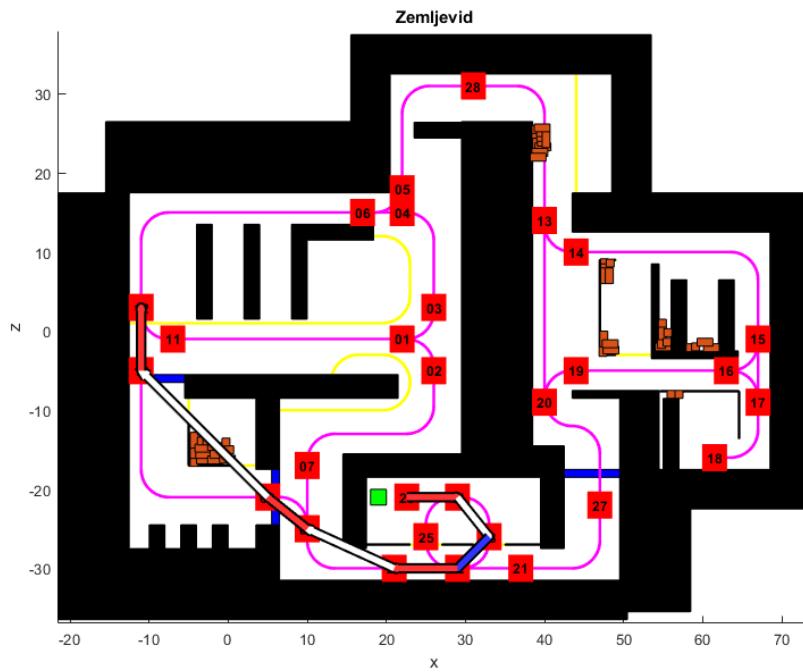
Glede na absolutno vrednost odmika se tudi zmanjša referenčna hitrost, ki je ob popolni poravnavi enaka v_1 , ob zamiku o pa je enaka $v_1(1 - |o|k_s)$, kjer je k_s poljubno določen koeficient upočasnitve. V opisanem primeru velja $k_s = 1/(6d_r)$. Tako kot pri kotu zavoja, tudi to vrednost pošljemo regulatorju v simulaciji.

Ta algoritem je namenjen uporabi v kombinaciji z algoritmom A* za iskanje poti, ki je predstavljen v poglavju 4.4. Končni rezultat algoritma je set akcij. To je zbirka vnosov, kjer vsak vsebuje podatek o ID številu vozlišča in načinu sledenja črti (levi rob, sredina ali desni rob) za to vozlišče. Ko robot zazna marker, lahko iz seta akcij poišče pripadajoče vozlišče in način sledenja. Da robot izbere najbolj levo pot skozi križišče izberemo sledenje levemu robu in da robot izbere najbolj desno pot skozi križišče izberemo sledenje desnemu robu. Ko odsek ne vsebuje križišč izberemo sredinsko poravnavo, saj robot najbolj gladko sledi črti.

Tako za pot, ki je izračunana na koncu poglavja 4.4 med vozlišči 29 in 12, lahko razberemo vedenje, ki je vizualizirano na sliki 5.3. Rdeča barva puščice med dvema vozliščema predstavlja sledenje levemu robu črte na tem odseku, bela barva predstavlja sledenje sredini črte in modra barva predstavlja sledenje desnemu robu.

V vozlišču 29 bo torej robot sledil levemu robu črte, nato bo v zavoju med vozlišči 24 in 23 sledil sredini črte. V vozlišču 23 bo začel slediti desnemu robu zaradi bližajočega se križišča. V križišču v vozlišču 22 bo robot preklopil na sledenje levemu robu črte. V vozlišču 22 bo za sledeči odsek sledil sredini črte, in v naslednjem križišču (vozlišče 8) ponovno preklopil na sledenje levemu robu.

Nato lahko sledi sredini črte, in v vozlišču 10 ponovno preklopi na levi rob zaradi bližajočega križišča. Ko robot doseže vozlišče 12 se ustavi, saj je dosegel izbrani cilj.



Slika 5.3: Prikaz izbire načina sledenja črti v skladu z setom akcij, ki jih je vrnil algoritem A*.

5.3 Izogibanje trkov

Algoritem za izogibanje trkov služi kot pomožen algoritem pri vodenju mobilnega robota. Primarni algoritem robota skrbi za opravljanje določene naloge (npr. vodenje v točko), algoritem za izogibanje trkov pa le skrbi, da ob opravljanju primarne naloge ne pride do trkov. Ima dva ključna dela: zaznava morebitnega trka in odziv na to zaznavo. Robot si konstantno "ogleduje" svojo okolico in, ko zazna na svoji poti oviro, primerno ravna, da se izogne trku. To storimo s pomočjo simuliranega senzorja Lidar, opisanega v poglavju 3.6.2. Iz razdalj do zadetka posameznega žarka in njegove smeri lahko sklepamo na morebitne ovire. Tu predpostavljamo, da se ovira giblje relativno počasi oz. da miruje. Ko je ovira zaznana, mora algoritem ugotoviti, kako se trku izogniti. Rezultat

algoritma je torej popravek referenčnega kota zavoja, glede na trenutnega, ki ga določa primarni algoritem vodenja.

Na sliki 5.4 je viden prikaz želenega vedenja robota ob zaznani oviri. Predstavljeni implementirani algoritem je dokaj osnoven: deluje dobro pri nizkem številu ovir, vendar lahko odpove pri velikem številu ovir in v ozkih prostorih, kjer nima zadostnega prostora za obvoz ovir. Možna nadgradnja bi bila v obliki načrtovanja trajektorije okoli zaznanih ovir in vodenja robota vzdolž te trajektorije. Za to bi bil potreben tudi natančnejši senzor Lidar: trenutni periodično pošilja le 5 žarkov, da prihranimo na računski zahtevnosti simulacije, kar je znatno premalo za natančno mapiranje prostora, ki bi bilo potrebno za to metodo.



Slika 5.4: Prikaz vedenja algoritma za izogibanje oviram.

Za vsak žarek je določen prag zaznane razdalje, pod katerim smatramo, da je na poti ovira, ki se ji je potrebno izogniti. Za sredinski žarek je prag znatno večji od stranskih. Uporabljene vrednosti so: $5m$ za stranske žarke pod kotom $\pm 60^\circ$, $6m$ za stranske žarke pod kotom $\pm 30^\circ$, in $9m$ za sredinski žarek. Če je vsaj ena zaznana razdalja pod pripadajočim pragom, potem smatramo, da je potrebno smer gibanja prilagoditi, da se izognemo trku. V vsaki iteraciji Simulink modela zanka za vsak žarek preveri, če je razdalja pod pripadajočim pragom. Če je, zanj izračuna vrednost, ki bo na koncu prispevala k izhodnemu rezultatu. Ta vrednost je produkt uteži in pragu pripadajočega žarka, deljen z zaznano razdaljo žarka, opisano s sledečo enačbo: $v_i = w_i t_i / d_i$. v_i predstavlja izračunano vrednost žarka i , w_i utež za žarek i , t_i prag za žarek i in d_i zaznano razdaljo žarka i . Uteži žarka w_i so vrednosti, ki določajo, koliko posamezen žarek prispeva h končnemu rezultatu, in njegovo smer (negativno za popravek v levo, pozitivno za popravek v desno).

Ko so izračunane vrednosti za vseh 5 žarkov (kjer je to potrebno), njihovo vsoto uporabimo za izhod algoritma. Če ni nobena zaznana razdalja žarka pod pripadajočim pragom, je izhod algoritma preprosto enak 0. Izhodno vrednost še ojačamo in filtriramo z vgrajenim Simulink blokom za diskretno filtriranje, da je odziv robota nekoliko bolj postopen. To vrednost nato prištejemo trenutni vrednosti referenčnega kota primarnega algoritma. Če je primarni algoritem vodenje po črti, je potrebno robotu pomagati ponovno najti črto. To storimo tako, da ob koncu izogibanja oviri ne prepustimo vodenje nazaj algoritmu za sledenje črti. Namesto tega začasno aktiviramo vodenje v točko, kjer za cilj uporabimo lokacijo naslednjega markerja na poti. Ko robot ponovno zazna črto, se lahko nazaj aktivira algoritmom za sledenje črti.

6 Zaključek

V magistrskem delu smo obravnavali tematiko mobilne robotike, bolj natančno simulacijo avtonomnega kolesnega mobilnega robota v skladiščnem okolju. V grobem smo predstavili tematiko skladiščnih mobilnih robotov in pogosto uporabljenih senzorjev, predvsem pa smo želeli ustvariti simulacijsko okolje, s pomočjo katerega bi študentom omogočili preiskušanje algoritmov vodenja mobilnih robotov tudi na daljavo. Simulacijsko okolje vsebuje dva glavna gradnika: simulacijo in vodenje.

Simulacijo mobilnega robota smo implementirali v razvojnem okolju Unity 3D, s pomočjo programa Blender za ustvarjanje 3D modelov, ki so uporabljeni v simulaciji. Simulacijo vsebuje fizikalno simulacijo robota in njegove interakcije s svojim okoljem, regulacijo hitrosti in kota zavoja, simulacijo senzorja Lidar, senzorja za sledenje črti in senzor za zaznavo markerjev. Dodatno vsebuje način za grajenje skladišč po meri in kontrolnik kamere. Za dvosmerno komunikacijo z algoritmom vodenja v okolju Matlab-Simulink uporablja po meri narejen skript za UDP komunikacijo.

Vodenje smo implementirali v razvojnem okolju Matlab-Simulink. Služi predvsem kot demonstracija uporabe implementirane simulacije in je sestavljen iz večih individualnih primerov. Implementirali smo vodenje v točko, vodenje po črti in algoritmom za izogibanje trkov. Vodenje po črti uporabljamo v tandemu z algoritmom za iskanje poti A*, ki izračuna optimalno pot ob zagonu vodenja. Dodatno smo podali še model, ki služi kot predloga. Predloga vsebuje le konfigurirano UDP komunikacijo in pretvorbo podatkov ter ne vsebuje kakršnegakoli vodenja.

Izdelana rešitev seveda ni brez možnih dodatkov: kot nadaljno delo bi predlagal implementacijo dodatnih senzorjev, ter možnost konfiguracije aktivnih senzorjev na robotu. Dobrodošla bi bila tudi nadgradnja grafičnega uporabniškega vmesnika. Dodali bi več možnih ovir, ki jih uporabnik lahko vnesti v okolje, ter možnost dodatnih robotov. Dodatni roboti bi lahko vozili po vnaprej določenih poteh, ali pa bili enakovredni trenutnemu robotu, torej bi imeli lastno vodenje v okolju Matlab-Simulink. Te spremembe bi seveda znatno vplivale na računsko učinkovitost sistema, zato jih je potrebno dodajati premišljeno.

Cilj magistrske naloge je bil dosežen: uspešno smo razvili orodje za simulacijo, ki bo pomagalo študentom preiskušati algoritme vodenja kolesnih mobilnih robotov na daljavo. Prav tako lahko služi za demonstracijo na predavanjih oz. avditorskih vajah.

Literatura

- [1] “Unity [Online].” Dosegljivo: <https://unity.com/>. [Dostopano: 18. 7. 2022].
- [2] “Matlab-simulink [Online].” Dosegljivo: <https://www.mathworks.com/products/simulink.html>. [Dostopano: 4. 8. 2022].
- [3] “Blender [Online].” Dosegljivo: <https://www.blender.org/>. [Dostopano: 18. 7. 2022].
- [4] “Ros - robot operating system [Online].” Dosegljivo: <https://www.ros.org/>. [Dostopano: 4. 8. 2022].
- [5] R. F., V. F. in L.-A. C., “A review of mobile robots: Concepts, methods, theoretical framework, and applications,” *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, str. 1–22, 2019.
- [6] S. Kurlekar, “Sensors in autonomous mobile robots for localization and navigation.” Dosegljivo: <https://www.techtarget.com/iotagenda/blog/IoT-Agenda/Sensors-in-autonomous-mobile-robots-for-localization-and-navigation#:~:text=Types%20of%20sensors,-Exteroceptive%20sensors%20discern&text=Proprioceptive%20sensors%20deal%20with%20robot,as%20active%20or%20passive%20sensors>. [Dostopano: 13. 9. 2022].
- [7] J. Borenstein, H. R. Everett, L. Feng in D. K. Wehe, “Mobile robot positioning: Sensors and techniques,” *J. Field Robotics*, vol. 14, str. 231–249, 1997.

- [8] “Unity dokumentacija [Online].” Dosegljivo: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>. [Dostopano: 22. 8. 2022].
- [9] G. Klančar, A. Zdešar, S. Blažič in I. Škrnjanc, *Wheeled mobile robotics: from fundamentals towards autonomous systems*. Ljubljana: Elsevier, 2017.
- [10] S. Permana, K. Bintoro, B. Arifitama in A. Syahputra, “Comparative analysis of pathfinding algorithms a *, dijkstra, and bfs on maze runner game,” *IJISTECH (International Journal Of Information System & Technology)*, vol. 1, str. 1, 05 2018.
- [11] B. Zupančič, *Zvezni regulacijski sistemi I. del.* Ljubljana: Fakulteta za elektrotehniko, 2010.
- [12] B. Zupančič, *Avtomatsko vodenje sistemov*. Ljubljana: Fakulteta za elektrotehniko, 2011.