TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Oliver Paljak 206640IACB

Pac-Mani laadse mängu loomine robotitega

Bakalaureusetöö

Juhendaja: Priit Ruberg
PhD
Kaasjuhendaja: Erki Meinberg
MSc

Tallinn 2023

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Oliver Paljak

15.05.2023

Annotatsioon

Käesolev bakalaureusetöö annab ülevaate TTÜ Robotiklubile loodud esindusprojektist, mida on võimalik kasutada erinevatel konverentsidel TTÜ Robotiklubi tutvustamiseks huvilistele ja potentsiaalsetele uutele liikmetele – täpsemalt seati põhieesmärgiks luua Pac-Mani laadne mäng robotitega. Töös tuuakse välja üleüldised tingimused sellise mängu loomiseks ning kuidas mängus roboteid tuvastada ja raadioside abil juhtida. Lisaks, antakse ülevaade mänguväljakul asuvate takistuste tuvastusest ning nende vältimisest robotite juhtimisel. Töö käigus valmis programm, mida on võimalik TTÜ Robotiklubil esindusprojektina kasutada, kusjuures töö põhieesmärk koos seatud vaheülesannetega – luua Pac-Mani laadne mäng robotitega – saavutati.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 8 peatükki, 25 joonist, 1 tabelit.

Abstract

Creation of a Pac-Man Like Game with Robots

The aim of this thesis is to create a demo project for TUT Robotics Club that could be used for introducing the Robotics Club activities to people outside the robotics club including also to potential new members – in fact, the project is planned to be used in robotics related festivals (such as Robotex, LIT festival and so on). The idea of Pac-Man like game with robots was proposed by a former chairman and long term member of TUT Robotics club Erki Meinberg who is also the project lead and one of the supervisors of this thesis.

The thesis gives an overview of what is required to create such a game and also explains some of the reasoning behind certain design decisions. The work consists of 5 main chapters (8 in total) which should give the reader an idea on how create a Pac-Man like game with robots: first, an overview is given of what base systems should be in place before „real" development could begin. Secondly, a robot detection system with ArUcos is introduced. Thirdly, an overview is given about the control of robots using PID control and also specially designed radio communcations protocol for sending orders to the robots. Fourthly, a wall detection system is introduced that uses a grid, LSD and an algorithm that will detect walls in the grid using the lines detected from the LSD. Finally, a pathfinding system will be implemented that is used for controlling the robots so that they will avoid the detected obstacles in the grid.

It should be noted, that all the goals that were set for the thesis, were achieved – all systems that were implemented were tested in real life and worked accordingly as outlined in the thesis. The thesis author who is also a member of TUT Robotics Club was helped by other club members to create this work – the author wants to thank Märtin Laanemets for the help in testing the game and also the project lead Erki Meinberg for the vision he had for the game and supervising this thesis and project.

The thesis is in estonian and contains 40 pages of text, 8 chapters, 25 figures, 1 table.

Lühendite ja mõistete sõnastik

TTÜ	Tallinna Tehnikaülikool
TUT	*Tallinn University of Technology*
MP	Megapiksel
USB	*Universal Serial Bus*; Universaalne jadasiin
CPU	*Central Processing Unit*; Keskprotsessor
GUI	*Graphical User Interface*; Graafiline kasutajaliides
OpenCV	*Open Source Computer Vision Library*
PID	*Proportional, Integral, and Derative*; Proportsionaalne, Integraalne ja Tuletuslik juhtimine
AES	*Advanced Encryption Standard*; Täiustatud krüpteerimisstandard
UDP	*User Datagram Protocol*; Kasutajadatagrammi protokoll
ASCII	*American Standard Code for Information Interchange*
LSD	*Line Segment Detector*; Lõigu tuvastussüsteem
DFS	*Depth First Search*; Sügavuti otsing
BFS	*Breadth First Search*; Laiuti otsing

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Käesoleva bakalaureusetöö põhieesmärgiks on luua Pac-Mani sarnane mäng koos robotitega. Loodavat mängu on kavas kasutada TTÜ Robotiklubi tutvustamiseks läbi põneva ja kaasahaarava tegevuse erinevatel tehnikakonverentsidel (näiteks Robotex, LIT Festival jne). Antud idee käis välja TTÜ Robotiklubi endine esimees Erki Meinberg, kes on ka käesoleva projekti juht ja ka lõputöö üks juhendajatest. Siinkohal tahaks töö autor tänada nii Erki Meinbergi, Märtin Laanemetsa kui ka varem projektiga seotud olnud meeskonnaliikmeid, kes suuremal või vähemal määral on aidanud töö autorit projekti tegemisel – loomulikult, kui töös peaks leiduma vigu, siis need on täielikult omistatavad töö autorile.

Töö alameesmärkideks seati järgnevad ülesanded:

1) selgitada välja, mida on vaja Pac-Mani sarnase mängu loomiseks robotitega;
2) tuvastada mängus kasutatavaid roboteid nii, et tuvastusinfo sisaldaks kõike, mida on vaja mängu ülesehitamiseks;
3) luua üldine viis robotite juhtimiseks;
4) tuvastada takistusi mänguväljakul;
5) kuidas roboteid juhtida nii, et need väldiksid tuvastatud takistusi.

Vastavalt töö eesmärkidele ehitati töö ka üles. Peatükis „2 Mängusüsteemi ülevaade” selgitatakse, mida on üleüldises plaanis vaja Pac-Mani sarnase mängu loomiseks robotitega, milliseid vahendeid mängusüsteemis kasutatakse ja tutvustatakse lugejale ka loodud mänguprogrammi. Peatükis „3 Robotite tuvastus” tuuakse välja üldlevinenumad viisid objektide tuvastamiseks masinnägemisel ja ka selgitatakse, mida täpsemalt on vaja robotite tuvastamisinfos ja ka kuidas on robotite tuvastus mängus implementeeritud. Peatükk „4 Robotite juhtimine” annab ülevaate robotite baasjuhtimissüsteemist ja ka koostatud raadioside protokollist, mille abil juhtimine käib.

Peatükis „5 Takistuste tuvastus mänguväljal” tuuakse välja, kuidas toimub mänguprogrammis takistuste tuvastus. Peatükk „6 Rajaleidmine” annab ülevaate erinevatest rajaleidmisalgoritmidest (*pathfinding algorithms*) ja sellest, kuidas toimub robotite juhtimine nii, et need väldiksid tuvastatud takistusi. Viimases peatükis „7 Ideid edasiarenduseks” tuuakse välja ideed, mida saaks teha, et loodud programmi veel paremaks muuta.

2 Mängusüsteemi ülevaade

Projekti alusideeks on 1980. aastal Jaapani firma Bandai Namco Entertainment poolt loodud populaarne konsoolimäng Pac-Man (originaalselt planeeritud nimega Puck-Man), kus mängija ehk Pac-Man peab labürindis koguma punkte samal ajal hoidudes vastastest ehk kummitustest. [1]

Joonis 1: Pac-Mani (varasema nimetusega Puck-Man) labürint koos originaalse mängukonsooliga [1]

2.1 Pac-Man robotitega

Pac-Mani mängu loomiseks päris robotitega on mitmeid viise ehk antud ülesandele on võimalik esitada mitu küsimust/probleemi:

1) Kuidas tõlgendada originaalses mängus kasutatav labürint (Joonis 1) või vähemalt selle sarnane takistusrada päris maailma – kas selleks peakski olema näiteks reaalsed seinad, millest mängija ja vastased ei saa läbi sõita või piisab lihtsalt näiteks seinte maha märkimisest mingi markeriga (näiteks kleeplindiga) mänguväljakule? Või on näiteks võimalik labürint luua kuidagi virtuaalselt – näiteks projekteerida mänguprogrammi mälus olev labürint läbi projektori mänguväljakule?

2) Mida peavad mängus kasutatavad robotid suutma teha – näiteks kas robotid peavad suutma tuvastada ise takistusi või on võimalik mäng implementeerida ka nii, et robotid ise ei pea midagi tuvastama?

3) Kuidas toimub robotite juhtimine – kas robotid juhivad ennast ise ehk võtavad mängu seisukohast otsuseid vastu autonoomselt või on mängul näiteks mingi keskne „aju", mis ülejäänud süsteeme kordineerib.

4) Kuidas toimub mänguoleku (*game state*) juhtimine – millal mäng läbi saab, kuidas loetakse mängus mängija skoori jne.

5) Muud otseselt loodavast Pac-Mani mängust mitte sõltuvad lisatingimused ja -piirangud – kui palju ajalisi ja rahalisi ressursse saab kulutada antud mängu loomise peale; kas loodav lahendus peaks olema ainult Pac-Man või oleks hea, kui mängus loodavaid süsteeme saaks kasutada ka näiteks teiste mängude loomiseks jne.

Nagu võib aru saada, siis ühele küsimusele vastamine kitsendab ka teiste küsimuste vastuseid – näiteks labürindi tegemise meetodi valimisel saab paika panna robotitele seatavad piirangud ja ka vastupidi. Kuigi eelmises loetelus väljatoodud viies küsimus puudutab just neid aspekte, mis Pac-Mani mängu loomist otseselt ei mõjuta, siis antud projekti puhul oli see tegelikult kõige olulisem küsimus – põhjuses mängivad rolli kaks järgmist aspekti:

1) vastavalt projektijuhi ideele kasutatakse projekti TTÜ Robotiklubi tutvustamiseks välismaailmale ehk oleks väga hea, kui tulevikus oleks võimalik samade süsteemide alusel luua ka teisi mänge või demonstratsioonprojekte;

2) TTÜ Robotiklubil on mitmed seadmeid (üldotstarbelised arvutid, universaalne robotiplatvrom PisiBot, XIMEA kaamera), mida saaks sellise projekti loomiseks ära kasutada – see kõik tähendab, et ei pea kulutama aega ja raha uute süsteemide loomiseks, et antud projekti teostada;

Antud kaks tingimust muudab Pac-Mani mängu loomise robotitega palju kitsamaks:

1) Robotid ei tohiks olla autonoomsed ehk neid peaks juhtima keskne „aju”, kuna soov on luua projekt nii, et Pac-Man robotitega ei oleks tulevikus ainukene mäng/demonstratsioon, siis iga roboti ümberprogrammeerimine iga eraldi demonstratsiooni jaoks on väga ebamugav. Kui on olemas keskne juhtimissüsteem, siis saab teha muudatused seal ja need kehtivad kogu projektile, seal hulgas igale robotile. Muidugi tähendab see seda, et antud keskne juhtimissüsteem tuleb luua.

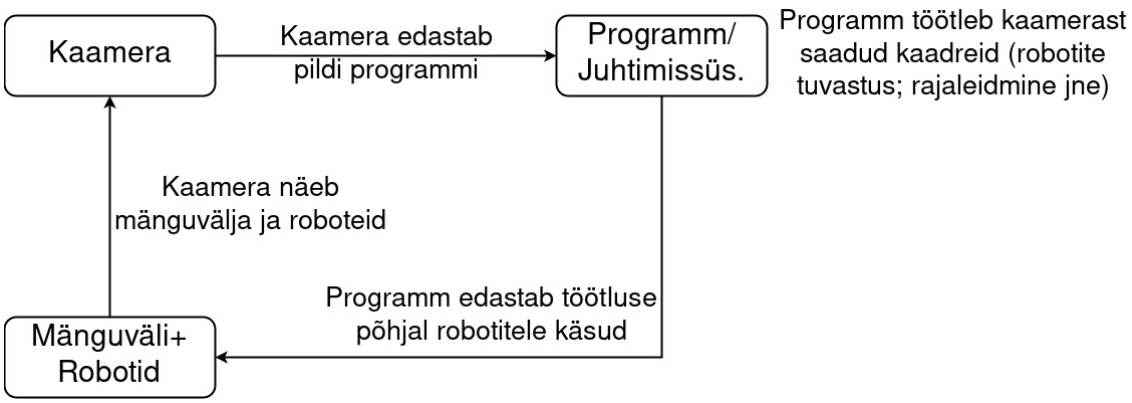
2) Seinad võiksid olla eelkõige markeritega märgitud, kuna kindlasti on võimalus luua mänge/demonstratsiooniprojekte, kus seinu/takistusi ei ole vaja kasutada. Takistuste märkimist virtuaalselt oleks võimalik implementeerida kesksesse juhtimissüsteemi vastavalt vajadusele.

3) Kuna on olemas keskne juhtimissüsteem, siis ei pea robotid olema tehniliselt väga võimekad ehk robotid peaksid ainult suutma vastuvõtta juhtimissüsteemilt käske ja neid vastavalt täitma.

4) Mänguoleku juhtimist saab implementeerida kesksesse juhtimissüsteemi.

5) Projekt peaks kasutama juba TTÜ Robotiklubis olevaid seadmeid (üldotstarbelised arvutid, universaalne robotiplatvrom PisiBot, XIMEA kaamera).

Antud kriteeriumitest on võimalik tuletada järgnev süsteem: XIMEA kaamera näeb ülevalt alla vaadates mänguvälja, kus asuvad PisiBotid. Läbi kaamera on võimalik kesksel juhtimissüsteemil tuvastada mänguks vajalik informatsioon (takistused, robotite asukohad jne) ning selle informatsiooni põhjal on võimalik roboteid vastavalt juhtida (Joonis 2), kusjuures robotitele käskude saatmine käib läbi raadioside. Niinimetatud keskne juhtimissüsteem ei ole mitte midagi muud kui töö käigus loodud C++ programmeerimiskeeles arvutiprogramm, mida on võimalik põhimõtteliselt igal tänapäevasel arvutil täita. Just loodud arvutiprogrammile lõputöö keskendubki, kusjuures vastavalt töö teemale on fookuses Pac-Mani mänguga seotud loogika.

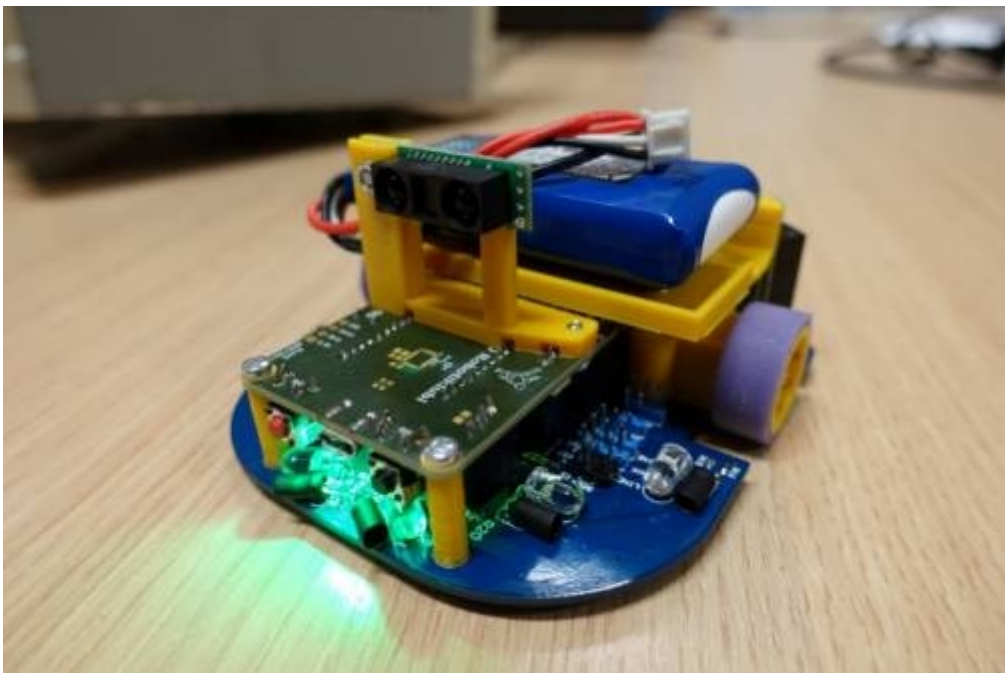


Joonis 2: Mängusüsteemi plokkskeem

2.2 Töös kasutatavad seadmed

Kõik seadmed, mida töös kasutatakse valiti eelkõige selletõttu, et need olid juba TTÜ Robotiklubil olemas, kuid ei saa ka eitada fakti, et kasutatud seadmed on piisavalt võimekad, et täita mängu jaoks vajalikke ülesandeid. Järgnevalt antakse ülevaade kasutatud seadmetest.

Mängus kasutatav Pisi-Bot (Joonis 3) on umbes käelaba suurune multifunktsionaalne robot, mis loodi eelkõige selleks, et see suudaks lahendada robootikavõistluste tüüpülesandeid (joonejärgimine, *folkrace* ja labürindi lahendamine). [2] Tänu oma multifunktsionaalsusele on võimalik Pisi-Boti käesolevas projektis edukalt kasutada. Töö käigus arendati autori poolt Pisi-Botidele ka mängu jaoks vastav tarkvara vahekiht, aga kuna sama esindusprojekti osana arendatakse (teise meeskonnaliikme poolt) Pisi-Botidele modernsemat trükkplaati, mille tulemusena loodud tarkvara muutub irrelevantseks, siis antud lõputöös vahekihti põhjalikult ei käsitleta. Ainukene asi, mis on antud vahekihi juures oluline on, et see võimaldab Pisi-Botil raadiokanalist tulnud käskudest aru saada ja ka nendele vastavalt reageerida (käsku täita).



Joonis 3: Pisi-Bot – TTÜ Robotiklubi mitmeotstarbeline robotiplatvorm [2]

Kaameraks, mille vahendusel käib robotite ja ka takistuste tuvastamine, on XIMEA xIQ MQ013CG-E2 (Joonis 4). Kaamera spetsifikatsioonid on järgnevad: resolutsioon 1.3 MP, 1280x1024; kaadrisageduseks on 60 kaadrit sekundis; USB3 ühendus. [3]

Joonis 4: XIMEA xIQ MQ013CG-E2 [3]

Keskne juhtimissüsteem kasutab infovahetuseks kahte viisi: suhtlus kaameraga toimub läbi USB ja suhtlus robotitega toimub läbi raadioside kasutades Digi Xbee 3 Zigbee 3

raadiomooduleid [4] – PisiBotid toetavad Xbeed [2] , kusjuures mängu jaoks sai loodud spetsiaalne rakenduskihi protokoll.

2.3 Keskne juhtimissüsteem

Projekti keskseks juhtimisüsteemiks loodi töö käigus programmeerimiskeeles C++ programm, mille ülesandeks ongi kogu mängusüsteemi hallata:

1) tuvastada kaamera vahendusel mänguväljakul olevad takistused (seinad);
2) tuvastada kaamerast saadud kaadrite põhjal robotid ja nende asukohad;
3) juhtida vastasroboteid läbi raadioside mängija robotini (vältides väljakul olevaid takistusi);
4) reageerida kasutaja sisenditele ehk mängija roboti juhtimine;
5) mänguoleku juhtimine ehk mis olekus on mäng praegusel ajahetkel (näiteks, kas vastasrobotid on mängija kinni püüdnud ehk kas mäng on läbi või ei).

Programmi on praktiliselt võimalik täita igal tänapäevasel arvutil, kuid töös on kasutusel olnud eelkõige Linux (Debian), mis tegi programmi arendamise lihtsamaks, kuna Debianil oli lihtsam võrreldes näiteks Windowsiga arenduseks vajalikke tööriistu üles seada.

Programmi jõudluse ja kiiruse tõstmiseks on programmis kasutatud mitmelõimelisust (*multithreading*). See tähendab, et arvuti CPU täidab programmi poolt antud ülesandeid ühel ja samal ajahetkel mitmes lõimes korraga ehk näiteks kui programmi pealõim saab kaameralõimest töödeldud (see tähendab robotite asukohtadega) pildi ja hakkab tegelema näiteks robotitele rajaleidmisega (*pathfinding*), siis samal ajal saab juba kaameralõim hakata tuvastama robotite asukohti järgmisel kaadril, mis omakorda tähendab seda, et kui pealõim peaks uuesti küsima kaameralõimest töödeldud pilti, siis on see juba olemas ja selle järel ei pea (üldjuhul) pealõim ootama. Kui mitmelõimelisust ei oleks programmis kasutatud, siis tekiks alati olukord, kus pealõim peaks pilditöötluse taga ootama.

Kokku on programmis neli suuremat lõime (Joonis 5):

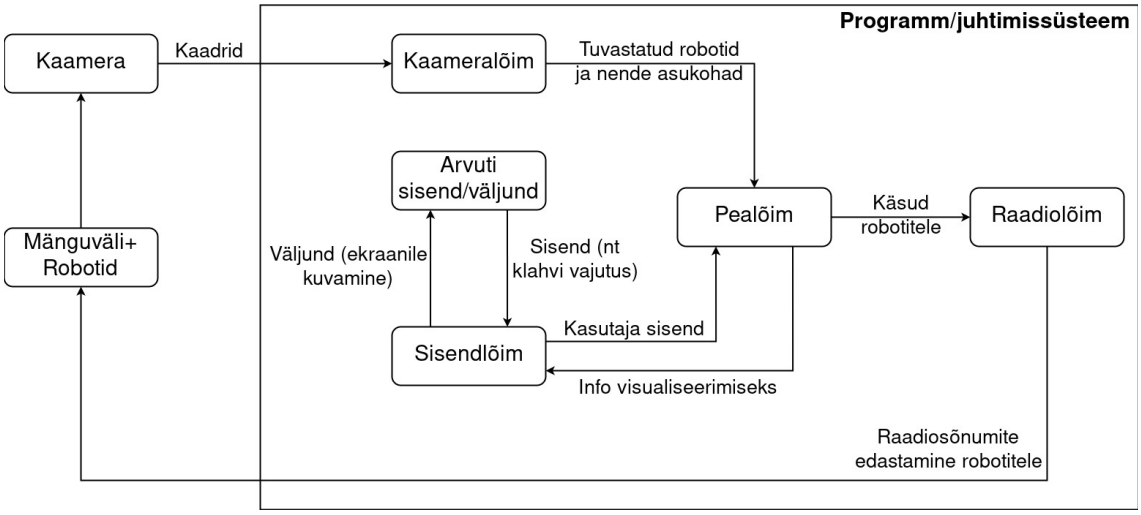
1) pealõim – siin asub eelkõige mänguspetsiifiline loogika (kasutajalt sisendi saamine; seinade tuvastus; rajaleidmine; mänguoleku juhtimine jne);

2) kaameralõim (koos tuvastuslõimedega) – antud lõim vastutab kaamerast kaadrite saamise ja ka mänguväljakul asuvate robotite tuvastamise eest;

3) sisendlõim – vastutab GUI ja kasutaja sisendite lugemise eest;

4) raadiolõim – vastutab raadioside eest.

Programm on disainitud nii, et programmi on võimalik juurde kirjutada erinevaid mänge – see tähendab, et Pac-Man on ainult üks võimalik mäng terve programmi kasutusvõimalustest. Näiteks juba praegu on programmis tegelikult kaks mängu – üks on siis Pac-Man, teine on lihtne tagaajamismäng (praktiliselt Pac-Man ilma seinteta). Tehniliselt tähendab see seda, et pealõim on see, mis otsustab, mida teistest lõimedest saadud infoga peale hakata ja mida selle põhjal mänguväljakul asuvatele robotitele läbi raadiolõime saata. Vastavalt teemale keskendub töö edaspidi just nendele lahendustele, mida oli vaja Pac-Mani laadse mängu loomiseks.



Joonis 5: Programmi plokkskeem lõimedest Pac-Mani mängu põhjal

2.4 Peatüki kokkuvõte

Peatükis anti ülevaade:

- projekti alusideest ehk originaalsest Pac-Man mängust (peatükk „2 Mängusüsteemi ülevaade”);

- kuidas kavatsetatakse originaalne konsoolimäng üle tuua päris maailma koos robotitega (peatükk „2.1 Pac-Man robotitega”);
- milliseid füüsili seadmeid kasutatakse Pac-Mani realiseerimiseks robotitega (peatükk „2.2 Töös kasutatavad seadmed”);
- mida täpsemalt kujutab endast loodud keskne juhtimissüsteem (peatükk „2.3 Keskne juhtimissüsteem”).

3 Robotite tuvastus

Robotite tuvastamisel mängus on olulised kolm aspekti:

1) robotite asukohad/koordinaadid mänguväljal – vajalik mänguoleku (*game state*) juhtimisel ja ka rajaleidmisel (*pathfinding*);

2) robotite indentifitseerimine – oluline, et robotitele saaks omandada erinevaid rolle (näiteks mängija poolt kontrollitud robot ja arvuti poolt kontrollitud vastasrobotid);

3) roboti pöördenurk sihtmärgi suhtes – vajalik roboti juhtimisel PID kontrolleriga.

Programmis implementeeritud tuvastussüsteem peab suutma kindlaks määrata kõik kolm eelmainitud aspekti.

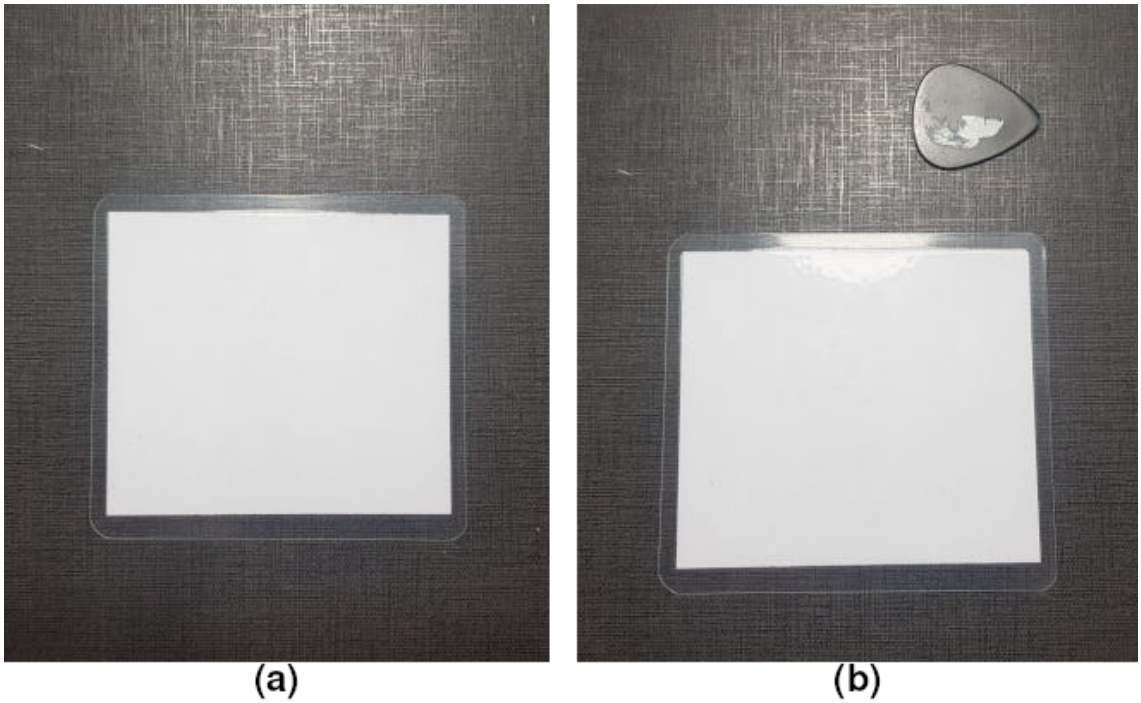
3.1 Võimalusi objektide tuvastamiseks masinnägemises

Peamiselt on masinnägemisel võimalik erinevaid objekte ära tuvastada kahel viisil: kas masinõppe abiga või puhtalt pilditöötluse abiga. [5] Masinõppe puhul treenitakse närvivõrgustik, mis suudab objektid ära tuvastada/klassifikatseerida. Pilditöötluses aga muundatakse/võetakse pildilt välja informatsioon, et selle põhjal matemaatiliselt ja algoritmiliselt objekte tuvastada. Oluline on võib olla ka mainida, et masinõppe puhul kasutatakse ka tihti pilditöötluse abi, kuid tuvastamine siiski toimub närvivõrgustikus. [6]

Kui masinõppe abiga oleks võimalik Pisi-Botide asukohta ja ka nurka määrata, siis keeruliseks muutub eelkõige robotite identifitseerimine ja seda selletõttu, et Pisi-Botid näevad väga sarnased välja. Lisaks, oleks vaja masinõppe puhul programmis treenida enda närvivõrgustik, mis oleks üsna aega ja ressurssi nõudev tegevus.

Järelikult, et vältida töös liigset keerukust, otsustati pilditöötluse kasuks. Kuigi pilditöötlusega võib objekti enda tuvastamine muutuda keeruliseks [6] , siis siiski on

võimalik kasutada pilditöötlust, et määrata mängus robotite tuvastamisel olulised aspektid (asukoht, identifitseerimine ja nurga määramine – peatükk „3 Robotite tuvastus”). Üheks selliseks viisiks oleks kasutada piltkoode (*visual marker*). Piltkoodid on masinnägemises kasutatavad kui referentsväärtused (*fiducial marker*), mille järgi on võimalik määrata nii vahemaid, nurki ja palju muud. [7]



Joonis 6: (a) Ilma referentsita pilt; (b) Referentsiga pilt [7]

Jooniselt 6.a võib näha valget ruutu, kuid selle mõõtmeid on väga raske hinnata – antud ruut võib olla millimeetrites, sentimeetrites või isegi meetrites. Kui aga lisada juurde referents (ehk midagi, mille mõõtmed on kas täpselt või umbkaudu teada), siis muutub hinnangu andmine kordades lihtsamaks (Joonis 6.b). Täpselt samal eesmärgil (hinnangu andmisel) on ka piltkoodid masinnägemisel kasutusel. [7]

3.1.1 Piltkoodid

Kuna piltkoode kasutatakse referentsväärtusteks, siis nende tuvastamine peab olema tehniliselt lihtne, kiire, veakindel ja sisaldama just hinnangu andmiseks vajaliku informatsiooni. Kuigi piltkoode võib teha värvilisi, siis enamus nendest on siiski must-valged, kuna must-valge pildi tuvastamine on värvilisest kiirem ja veakindlam, kuna must-valge pilt sisaldab endas võrreldes värvilise pildiga vähem informatsiooni [8] [9] . Oma (tehnilise) lihtsuse tõttu saab kasutada nende tuvastuseks ainult pilditöötlust.

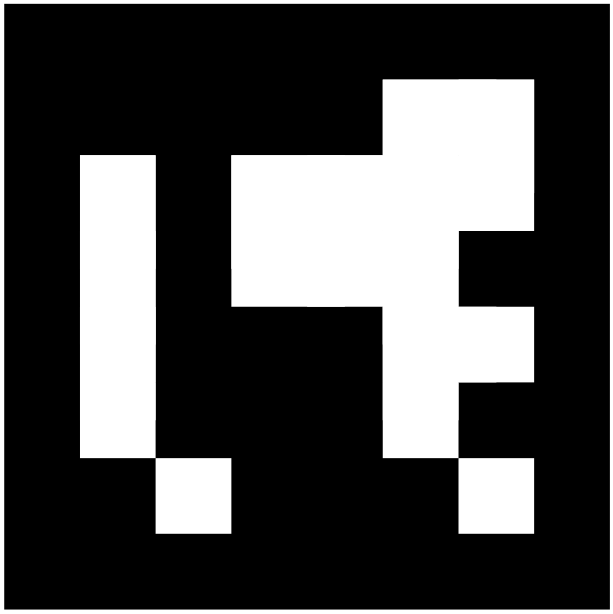
Piltkoodid aitavad lahendavad järgmisi probleeme:

- suuruste hindamine – teades piltkoodi füüsilist suurust, saame hinnata teiste objektide suurusi masinnägemisel;
- orientatsioon (või ka *pose estimation*) – kuidas objekt paikneb kaamera suhtes;
- kalibreerimine – näiteks kaamerapildis võib sirge joon paista kõverdatuna, piltkood aitab seda tuvastada;
- identifikatsioon – objektide identifitseerimine ja ka vahet tegemine sarnastel objektidel. [7]

Nagu võib aru saada, siis piltkoodid lahendavad just neid probleeme, mida mängus on vaja lahendada (asukoht, identifitseerimine ja nurga määramine – peatükk „3 Robotite tuvastus”). Kuigi piltkoodi meetodeid on palju (ARTag, Intersense, Matrix, CyberCode, VisualCode ja teised [10]), siis edaspidi keskendub töö ArUco piltkoodile, kuna programmis kasutatud teek OpenCV toetab just ArUco piltkoode.

3.2 ArUco piltkood ja OpenCV

ArUco on binaarne piltkood (Joonis 7), mida on oma lihtsuse tõttu võimalik väga edukalt ära kasutada masinnägemises. Eelkõige on ArUcod leidnud kasutust probleemides, kus on oluline tuvastada objekti ja kaamera asukoha suhe (*pose estimation*). Kuna ArUco kasutab värvidest ainult musta ja valget, siis on selle tuvastamine väga kiire ja ka veakindel – ArUco (sisemine) valge osa on binaarne maatriks, mis omastab ArUcole kindla arvu (ehk ID). Antud binaarne maatriks võib olla erineva suurusega – näiteks eksisteerib 4x4 (ehk 16-bitise) maatriksiga ArUcosid, kuid on olemas ka 6x6 (ehk 36-bitiseid) ArUcosid jne. [10] [11]



Joonis 7: Näide ArUco piltkoodist, mille maatriksi mõõtmed on 6x6 ja ID 1 [12]

OpenCV on üks populaarsemaid masinnägemise ja masinõppe teeke maailmas, mis sisaldab endas rohkem kui 2500 optimiseeritud algoritmi. OpenCV suudab tuvastada inimnägusid, objekte, inimtegevusi, jälgida tuvastatud objekte, luua piltidest 3D mudeleid jne. OpenCV on avatud lähtekoodiga ja seda on võimalik kasutada nii Windowsis, Linuxis, Androidis kui ka Mac OS-il. Lisaks, kasutavad OpenCV-d tuntud tehnoloogiafirmad nagu Google, Yahoo, Microsoft, Intel ja teised. [13] OpenCV-d otsustati kasutada loodud programmis just sellepärast, et selles sisaldub palju algoritme, mis muudab mängus masinnägemisega seotud ülesanded ja probleemid kergemini lahendatavaks ja ka sellepärast, et antud teek on reaalaja masinnägemise üks standardteekidest. Lisaks, toetab mängus kasutatav kaamera (XIMEA xIQ MQ013CG-E2) ametlikult OpenCV teeki [14] .

3.3 Robotite tuvastus mängus

Mängus on kaamera asetatud nii, et see vaatab mänguväljale otse ülevalt peale, mis muudab nii robotite tuvastamise kui ka ülejäänud mänguloogika lihtsamaks, kuna praktiliselt kaob ära kolmas mõõde. Robotite tuvastuseks kleebiti mängus kasutatavatele Pisi-Botidele peale ArUco, mille järel oli võimalik OpenCV abiga programmis iga roboti asukoht koos ID-ga kindlaks teha (Joonis 8). Kuna ArUcosid kasutatakse tihti kaamera ja ArUco vahelise suhte määramiseks (*pose estimation*), mida küll mängus vaja

ei ole, siis sellest tulenevalt on väga lihtne ka määrata iga roboti (ehk ArUco) nurka sihtmärgi suhtes, mis mängib robotite juhtimises väga olulist rolli, ja ka määrata robotite vahelisi kaugusi sentimeetrites.

Joonis 8: Pisi-Botide tuvastus programmis

Kuigi OpenCV on optimiseeritud teek, siis kahjuks jäi OpenCV ArUco tuvastamiskiirus mängu jaoks aeglaseks. Antud probleemi lahendamiseks otsustati tuvastamine viia kaameralõimes veel kolme eraldi lõime ja lisada ArUco tuvastamise algoritmile kindel sagedus, millal tuvastamine toimub. Tehniliselt näeb protsess välja järgnev (Joonis 9):

1) kaameralõim paneb kolm tuvastuslõime korraga tööle;

2) kaameralõim pärib XIMEA kaamerast (kõige hilisema) kaadri;

3) kaameralõim lisab päritud (pärast teatud aja mõõdumist) kaadri ühte vabasse tuvastuslõime;

4) koheselt pärast kaadri lisamist ühte tuvastuslõime, kontrollib kaameralõim, kas mõni varasemalt lisatud kaader on ära tuvastatud – kui on, siis kaameralõim annab selle koos tuvastusinfoga edasi pealõimele.

Oluline on rõhutada, et info, mille kaameralõim edastab pealõimele on alati kõige värskem. See tähendab, et kui näiteks ühe kaadri tuvastamine võtab nii kaua aega, et teine tuvastuslõim suudab järgmise kaadri samal ajal ära tuvastada, siis esimese kaadri

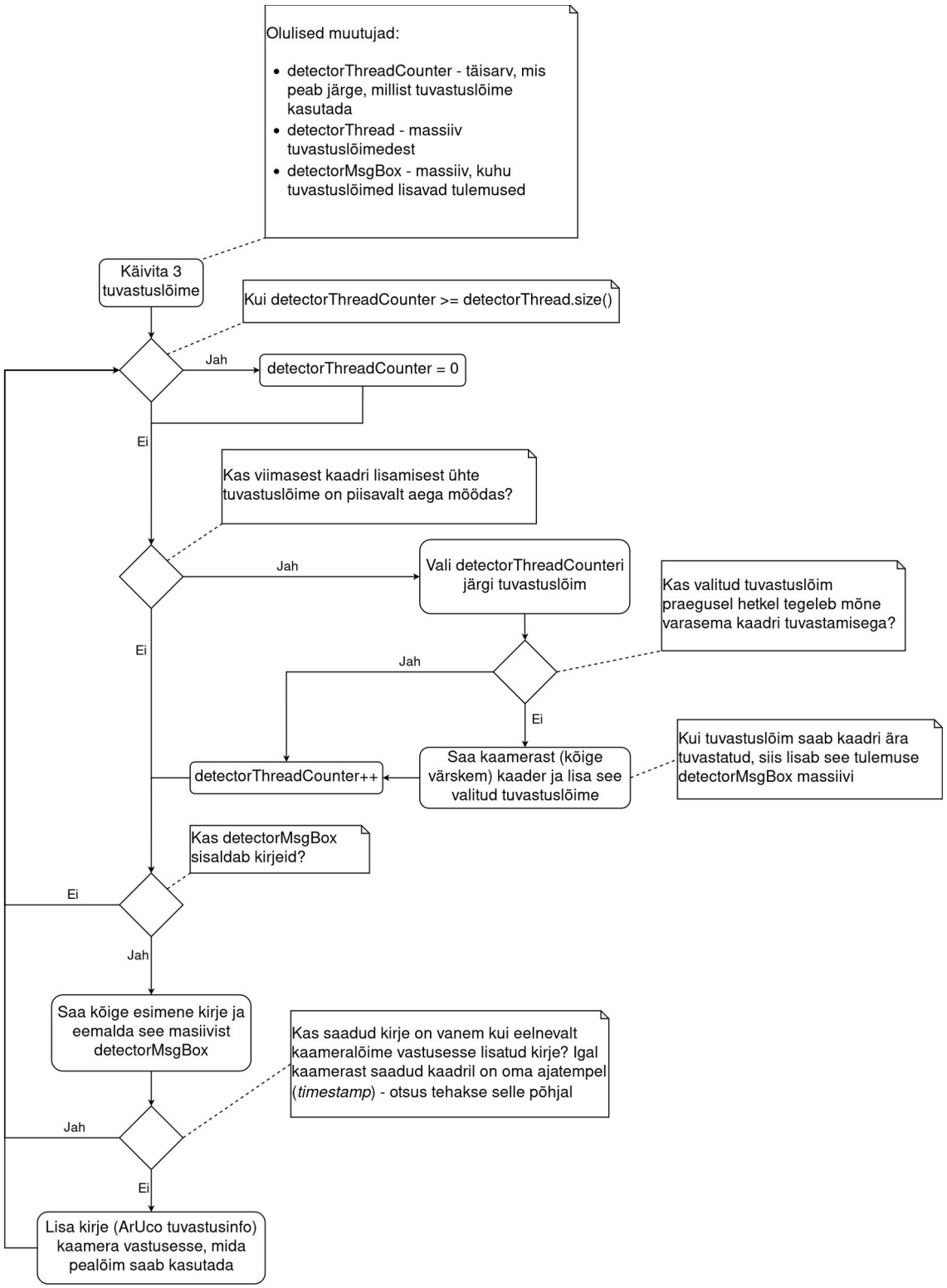
infot ei edastata pealõime. Antud teguviis tagab, et pealõim saab alati kasutada töötluseks infot, mis kajastab mänguväljal toimuvat kõige täpsemalt ja võimalikult lähedalt reaalajale.

Pealõimes kasutatakse tuvastusinfot, et omistada robotitele identifikatsiooni põhjal erinevad rollid ehk tehakse vahet vastasrobotitel, mida kontrollib arvuti, ja mängija robotil, mida kontrollib inimene. Lisaks, tuvastusinfo sisaldab endas robotite asukohti, mis on väga oluline nii mänguoleku kontrollimises, skoori lugemisel kui ka rajaleidmisel. Robotite asukohtade infost on võimalik tuletada robotite pöördenurgad, mis on oluline robotite juhtimisel PID kontrolleriga.

3.4 Peatüki kokkuvõte

Peatükk andis ülevaate:

- millised on mängus robotite tuvastmisel olulised aspektid (peatükk „3 Robotite tuvastus”);
- kuidas üleüldse on võimalik objekte masinnägemises tuvastada (peatükk „3.1 Võimalusi objektide tuvastamiseks masinnägemises”) ja mida täpsemalt endast kujutavad piltkoodid ning miks neid masinnägemisel kasutatakse (peatükk „3.1.1 Piltkoodid”);
- mängus kasutatavast piltkoodist ArUco ja ka masinnägemise teegist OpenCV (peatükk „3.2 ArUco piltkood ja OpenCV”);
- kuidas täpsemalt käib mängus robotite tuvastus kasutades ArUco piltkoode ja teeki OpenCV (peatükk „3.3 Robotite tuvastus mängus”).



Joonis 9: Vooskeem ArUco tuvastusest

4 Robotite juhtimine

Robotite juhtimine toimub programmis raadioside abiga, kusjuures raadiosides kasutatakse spetsiaalselt disainitud rakenduskihi protokolli. Robotid saavad aru neljast käsust:

1) „stop” – käsk, mis katkestab kõik teised käsud ja jätab roboti seisma;
2) „keera” – käsk, millega robot pöörab ümber oma telje käsuga etteantud nurga;
3) „sõida otsejoones” – käsk, millega robot sõidab otsejoones käsuga etteantud vahemaa;
4) „mootorite kiirus” – käsk, millega saab seada nii vasaku kui ka parema mootori kiirust ja suunda.

Pac-Manis kasutatakse kolme käsku: „stop”, „keera” ja „mootorite kiirus”. „Stop” käsku kasutatakse, kui näiteks vastasrobotil rajaleidmine ebaõnnestub või kui mäng on sellises olekus, kus robotid ei pea liikuma (näiteks pausil või kui mäng saab läbi ehk kui vastasrobotid on mängija roboti „kinni püüdnud”). „Keera” käsku kasutatakse siis, kui punkt, kuhu robot sõitma peab, on tema vaateväljast väljas (roboti vaatevälja moodustavad kaks kiirt, mis algavad roboti keskpunktist ja need on üksteise suhtes programmis defineeritud nurga all ehk roboti vaateväli on lõpmatu sektor roboti keskpunktist roboti edasi liikumise suunas). Kuna „keera” käsuga saab ette anda ka nurga, siis saab robotit keerata just täpselt sihtpunkti suunas. Kui aga sihtpunkt on roboti vaateväljas, siis kasutatakse käsku „mootorite kiirus”, kusjuures mootorite kiiruste arvutamiseks kasutatakse PID kontrollerit.

4.1 PID kontroller mängus

Mängus kasutatakse PID kontrollerist ainult kahte osa – proportsionaalset ja tuletuslikku osa. Antud lahendusi jõuti läbi katsetamise. Oluline on võib olla ka

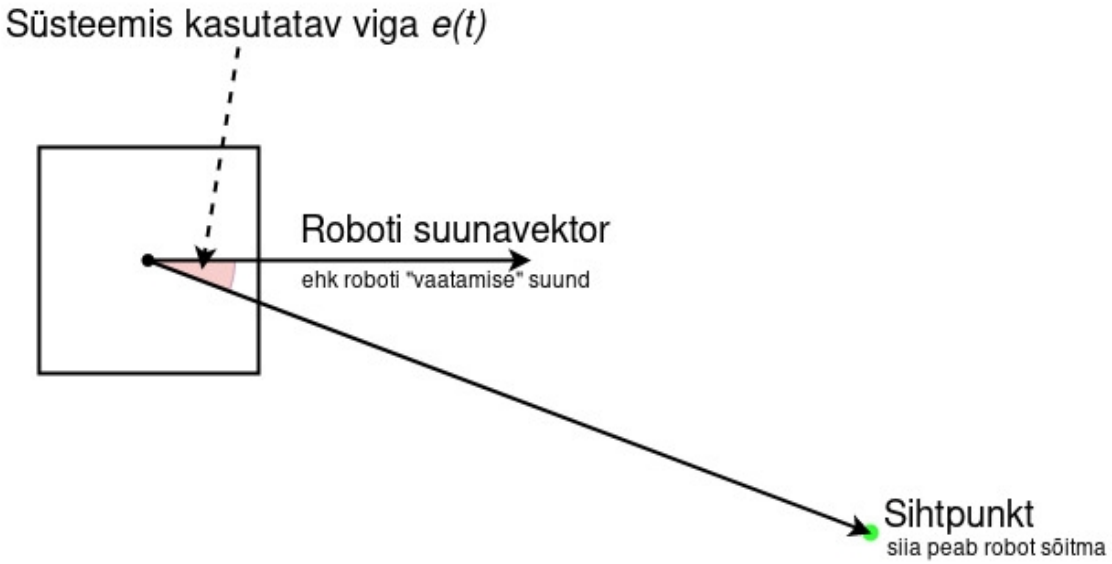
mainida, et süsteemi veaks ei ole mitte roboti kaugus sihtmärgist, vaid nurk roboti suunavektori ja vektori vahel, mille üheks otspunktiks on roboti enda keskpunkt ja mille teiseks otspunktiks on sihtpunkt, kuhu robot sõitma peab (Joonis 10) – kaugus sihtpunktist mängib eelkõige rolli leitud raja (*path*) järgmisel ja mänguoleku seadmisel (kas vastasrobotid on mängija robotile piisavalt lähedal ehk mängija roboti „kinni püüdnud").

Selleks, et muuta PID kontrolleri kasutamine mängus arusaadavamaks, on arvatavasti mõistlik esitada implementeeritud kontroller matemaatilisel kujul. Vea valem on järgnev: $e(t)=r(t)-s(t)$, kus

- $r(t)$ on roboti pöördenurk kraadides ühikvektor $\vec{j}=(0;1)$ suhtes (ajahetkel t);
- $s(t)$ on nurk kraadides (ajahetkel t), mis jääb ühikvektori $\vec{j}=(0;1)$ ja vektori vahele, mille alguspunktiks on antud roboti keskpunkt ja lõpp-punktiks sihtpunkt/sihtmärk;
- $e(t)$ on nende kahe nurga vahe ehk süsteemis kasutatav viga (Joonis 10).

Kuna mängus kasutatakse tegelikult PID kontrollerist proportsionaalset ja tuletuslikku osa, siis saab mängus kasutatavat kontrollerit matemaatiliselt avaldada järgnevalt: $u(t)=K_P\cdot e(t)+K_D\cdot\frac{de(t)}{dt}$ (võrreldes PID kontrolleri täisvalemiga: $u(t)=K_P\cdot e(t)+\int_0^t e(\tau)d\,\tau+K_D\cdot\frac{de(t)}{dt}$ [15]). Kuigi võiks arvata, et tuletise rakendamine programmis on keeruline, siis tegelikult saab ära kasutada tuletise definitsiooni: $\frac{de(t)}{dt}=\lim_{\Delta t\to 0}\frac{e(t+\Delta t)-e(t)}{\Delta t}$. Definitsioonis $e(t+\Delta t)$ tähistab tuleviku viga või matemaatiliselt täpsemalt väljendades järgmise ajahetke viga, kuid tulevikku ei ole võimalik reaalses maailmas ette ennustada, seega peab aluseks võtma mineviku ja kuna ajahetke määramine on suhteline, siis on võimalik tuletis defineerida järgnevalt: $\frac{de(t)}{dt}=\lim_{\Delta t\to 0}\frac{e(t-\Delta t)-e(t)}{\Delta t}$, kus $e(t-\Delta t)$ on siis mineviku või ka eelneva ajahetke viga. PID kontrolleri arvutamise sagedus on programmis väga sage (igas mängutsüklis tehakse PID kontrolleri arvutus) – järelikult ajavahemik Δt ongi võimalikult lähedal nullile ja samal ajal praktiliselt konstantne (iga mängutsükkel võtab enam-vähem sama

palju aega). Seega tuletuslikku osa implementeerimisel programmis võib jagamise Δt ära jätta (õigemini saab murru $\frac{1}{\Delta t}$ tuua ette kordajaks, aga kuna see muutub konstandi K_D paika panemisel irrelevantseks, siis võib selle ära jätta) ning võttes arvesse ajahetke määramise suhtelisuse, jääb alles ainult lahutamine ehk $\frac{de(t)}{dt} \approx e(t-\Delta t)-e(t)$, mis praktiliselt tähendab eelmise vea väärtuse ja praeguse vea väärtuse vahet. Lõplikult on programmis kasutusel järgnev valem: $u(t)=5\cdot e(t)+1\cdot[e(t-\Delta t)-e(t)]$, kusjuures kordajate väärtused ($K_P=5; K_D=1$) leiti katsetamise teel. Juhtsignaali väärtust $u(t)$ kasutatakse mootorite kiiruste määramiseks.



Joonis 10: Illustratsioon mängu süsteemiveast

Siinkohal on oluline rõhutada, et eelnenud kirjeldus käis vastasrobotite ehk nende robotite kohta, mis ajavad mängija poolt kontrollitud robotit taga. Mängija kontrollib oma robotit ikkagi ise klaviatuuri abiga ning juhtimise aluseks on täpsed samad raadiokäsud, mis sai peatüki „4 Robotite juhtimine” alguses välja toodud.

4.2 Koostatud protokoll raadiosides

Loodud rakenduskihi protokoll (edaspidi suhtlusprotokoll) toetub Zigbee 3 protokolli peale. Jällegi, Zigbee protokolli kasuks otsustati eelkõige selletõttu, et robotiklubis olid juba Zigbee raadiomoodulid olemas, aga Zigbee on ka piisavalt võimekas, et see sobis raadioside aluseks. Mõned Zigbee protokolli võimalused ja omadused:

- toetus mitmele erinevale raadioside topoloogiale (*point-to-point*; *point-to-multipoint* jne);
- vähene viiteaeg (*latency*);
- piisav sõlmede (*node*) arv ühes raadiovõrgus;
- 128-bitine AES krüpteerimisvõimalus;
- põrgete vältimine (*collision avoidance)*, uuesti saatmised ja sõnumi vastuvõtmise kinnitamine *(acknowledgments*). [16]

Zigbee 3 ja ka loodud suhtlusprotokoll tõestasid ennast, kui robotite raadiomoodulitel ja arvuti raadiomoodulil olid erinevad baudid (vastavalt 115200 ja 9600) – mängu sai sellise konfiguratsiooniga mängida kuni kolme robotiga! Muidugi vea avastamisel viidi ka arvuti raadiomooduli baud 115200 peale.

Loodud protokoll on orienteeritud pidevale andmevoole, kuna mäng peaks olema võimalikult kiire ja ilma viidedeta (*responsive*). Lisaks, mängus kasutatav peamine juhtimissüsteem PID (peatükk „4.1 PID kontroller mängus") töötab samuti kõige paremini, kui juhtsignaali saaks viia juhitavasse süsteemi võimalikult kiiresti. Järelikult mängib raadiokommunikatsioonis suurt rolli vähene viiteaeg ja oluline ei ole iga sõnumi jõudmine robotiteni – kui mõni sõnum peakski ära kaduma, siis saab alati saata järgmise, mis võtab arvesse kõige värskemat olukorda mänguplatsil. Antud tõsiasjadest saab jõuda järelduseni, et mõistlik on lehvikside (*point-to-multipoint*) topoloogia, kus siis üks peasõlm (ehk arvuti) saadab ülejäänud sõlmedele (ehk robotitele) sõnumeid. Robotid ise ei peaks sõnumeid (seal hulgas kinnitussõnumeid ehk ACK/NACK) arvutile saatma, kuna see suurendaks viidet – oluline on ka siinkohal rõhutada, et arvuti saab teatud mõttes aru, kui sõnum robotini ei jõudnud, kuna arvuti näeb kõike mänguväljakul toimuvat läbi kaamera (näiteks roboti mitte/vale liikumise korral, saadab arvuti uue, nüüd juba praegusele olukorrale paremini vastava raadiosõnumi). Kui tuua paralleele interneti maailmast, siis loodud protokoll sarnaneb eelkõige UDP-ga, kus oluline ei ole mitte iga biti ja baidi jõudmine sihtkohta, vaid pidev andmevoog.

Selleks, et optimiseerida raadiosidet veelgi on programmis rakendatud erinevaid tingimusi, et mitte saata samu või ülisarnaseid käske mitu korda. Näiteks, kui robotile

on saadetud käsk keerata 90 kraadi vastupäeva (praegusest positsioonist) ja oletame, et robot on jõudnud kahe tsükli vahel keerata antud 90 kraadist viis kraadi, siis sooviks programm saata talle talle uue keeramiskäsu argumendiga 85 kraadi – ilmselgelt oleks see mõttetu ja raadioressurssi raiskav. Sama kehtib ka näiteks „mootori kiiruste" käsu puhul – kui uue käsu mootori kiirused erinevad vana käsu seatud kiirustest piisavalt vähe, siis ei ole ka mõtet sellist käsku saata. Seega, programm peab arvet, mis käsk on mingile robotile saadetud ja üritab vähendada topeltkäskude saatmist.

Üldiselt saab suhtlusprotokolli kirja panna järgmiselt `AAITLdataC`:

Tabel 1: Suhtlusprotokolli ülevaade

	1. kirje	**2. kirje**	**3. kirje**	**4. kirje**	**5. kirje**	**6. kirje**	**7. kirje**
Tähistus	A	A	I	T	L	data	C
Tähendus	Eel-kõnelus	Eel-kõnelus	ID	Käsu-tüüp	Andmete pikkus	Andmed	Kontroll-summa
Pikkus tähtedes	2	2	2	2	2	1...255	2
Lubatud väärtused	00	00	01...FF	00...03	01...FF	*Oleneb käsust*	00...FF

kus iga sümbol (välja arvatud `data`), tähistab ühte baiti (ehk siis väärtust 0-255), kusjuures antud bait peab olema väljendatud kahe numbrikohaga 16-süsteemis (ehk 00-FF). Sümbolid suhtlusprotokollis tähendavad järgmist:

- `A` on eelkõnelus (*preamble*) – sümbol, mis tähistab raadiosõnumi algust (selleks on väärtus 00);
- `I` on aadress ehk roboti/ArUco ID, millele sõnum saadeti, kusjuures väärtuseks ei tohi olla 00 ja väärtus FF edastab sõnumi kõikidele robotitele (ehk tegemist on *broadcast* aadressiga);
- `T` on käsu tüüp ehk mida peab robot tegema – praegusel hetkel on defineeritud neli käsu tüüpi („stop", „sõida otsejoones", „keera" ja „mootorite kiirus");

- `L` on sõnumi andmete (`data`) pikkus – vastavalt suhtlusprotokolli piirangutele, saab andmete pikkuseks olla 255 tähemärki/sümbolit ja minimaalseks pikkuseks on 1 tähemärk;

- `data` on raadiosõnumi/käsu reaalne sisu ehk näiteks „keera” käsu korral sisaldab see suunda, kuhu poole peab robot keerama, ja ka nurka, kui palju robot keerama peab, kusjuures käskude korral, millel on mitu parameetrit (nt „mootorite kiirus”) saab eraldada need üksteisest komaga;

- `C` on kontrollsumma, mis saadakse liites kõik sõnumi märkide (välja arvatud eelkõneluste) ASCII tabeli väärtused kokku (siinkohal on oluline rõhutada, et kontrollsumma arvutamisel ei kohelda sõnumi osasid kui baite, vaid lihtsalt kui sõne, mis koosneb ASCII tähemärkidest) ning antud summa jääk jagatakse (*modulo*) arvuga 255 – nii saame kontrollsumma, mida saab edukalt esitada ühe baidina. Valemi kujul on kontrollsumma leitav järgmiselt: $C = \left(\sum_{i=0}^{n-1} a_i \right) \% 255$, kus n on sõnumi sõne pikkus (ilma eelkõnelusteta), a_i on sõnumi sõne ühe tähemärgi ASCII väärtus.

Võib olla on oluline ka mainida, et suunda ei kohelda otseselt eraldi parameetrina, vaid suunda on võimalik arv väärtuse sisse kodeerida kasutades miinusmärki. Suund on muidugi kokkuleppiline ja sõltub eelkõige käsu tüübist, kuid kasutusel olevatest käskudest, kus on võimalik suunda määrata („keera”, „sõida otsejoones” ja „mootorite kiirus”) on positiivseks suunaks kas päripäeva liikumine (käsu „keera” puhul) või liikumine selliselt, et mootorite pöörlemise suund viiks roboti edasi (käskude „sõida otsejoones” ja „mootorite kiirus” puhul). Seega, selle asemel, et raadiosõnumis kirjeldada roboti keeramist 90 kraadi vastupäeva kahe parameetrina (näiteks data=v,5A; $5A_{16} = 90_{10}$), saab seda teha lihtsalt data=-5A.

Selleks, et paremini mõista loodud suhtlusprotokolli on võib olla mõistlik tuua näide, milline võib olla päris kirje raadiokanalis ja kuidas seda lahti mõtestada. Oletame, et mänguplatsil on kolm robotit, mille ID arvudeks on 1, 2 ja 10 ning raadiokanalis on rakenduskihi tasemel järgnev kirje (*buffer*): 0000010001053000002030**5C8,C8**4E00000A0206**-5A,C885**, kus:

- punasega märgitu tähistab iga üksiku sõnumi algust (eelkõnelust) – alati kaks baiti väärtusega 0 (peab olema alati kirjes eristatav, see tähendab, et parser peab arusaama, kus algab järgmine sõnum);
- rohelisega tähistatud osa/bait on aadressat ehk roboti ID – lubatud väärtusvahemikuks on 01 kuni FF;
- sinisega märgitud bait on käsutüüp – praegusel ajahetkel on defineeritud 4 käsku, seega lubatud väärtusteks on 00 kuni 03;
- kollasega märgitu tähistab andmeosa pikkust – lubatud väärtusvahemik on 01 kuni FF;
- **mustaga** on märgitud andmed ise – lubatud väärtused ja ka tähendused sõltuvad käsust;
- oranžiga on tähistaud kontrollsumma – sellega on võimalik kontrollida, kas see sõnum on ikka see, mida saatja tahtis vastuvõtjale saata.

Eelkõnelus on valitud nii, et see oleks raadiokanalist lihtsasti eristatav – selletõttu ei tohi kasutada mängus robotit, mille ID-ks oleks null, ja ka sõnumi andmete pikkus peab olema vähemalt üks, kuna vastasel juhul võib tekkida olukord, kus ühe sõnumi sisu ei ole võimalik eristada teise sõnumi algusest. Siit tuleb välja ka see tõsiasi, et kuigi „stop” käsk tegelikkuses parameetrit ei vaja, aga vastavalt suhtlusprotokolli nõuetele peab olema andmeosa pikkuse bait vahemikus 01-FF – järelikult peab siiski „stop” käsus sisalduma andmeid vähemalt ühe tähemärgi jagu.

Näites toodud kirje võib võtta kokku järgnevalt:

- 00 00 01 00 01 **0** 53 – „stop” käsk robotile 1;
- 00 00 02 03 05 **C8,C8** 4E – „mootorite kiirus” käsk robotile 2, kusjuures mootorite kiiruseks on antud juhul seatud nii vasaku kui ka parema mootori jaoks 200 ($C8_{16} = 200_{10}$), kusjuures mõlemad mootorid peaksid pöörlema nii, et need viiksid robotid edasi ehk pöörleksid päripäeva (kui oleks soov mootoreid vastupäeva pöörlema panna tuleks kasutada kiiruse ees miinus märki): vastavalt protokollile on esimene argument vasaku mootori kiirus ja teine argument

parema mootori kiirus – oluline on ka rõhutada, et sõna „kiiruse” all ei mõelda siin otseselt m/s või muud sellesarnast, vaid tegemist on suhtelise ühikuga, kus väärtus 0 tähistab mootori seismist ja väärtus 1000 mootori maksimaalset kiirust;

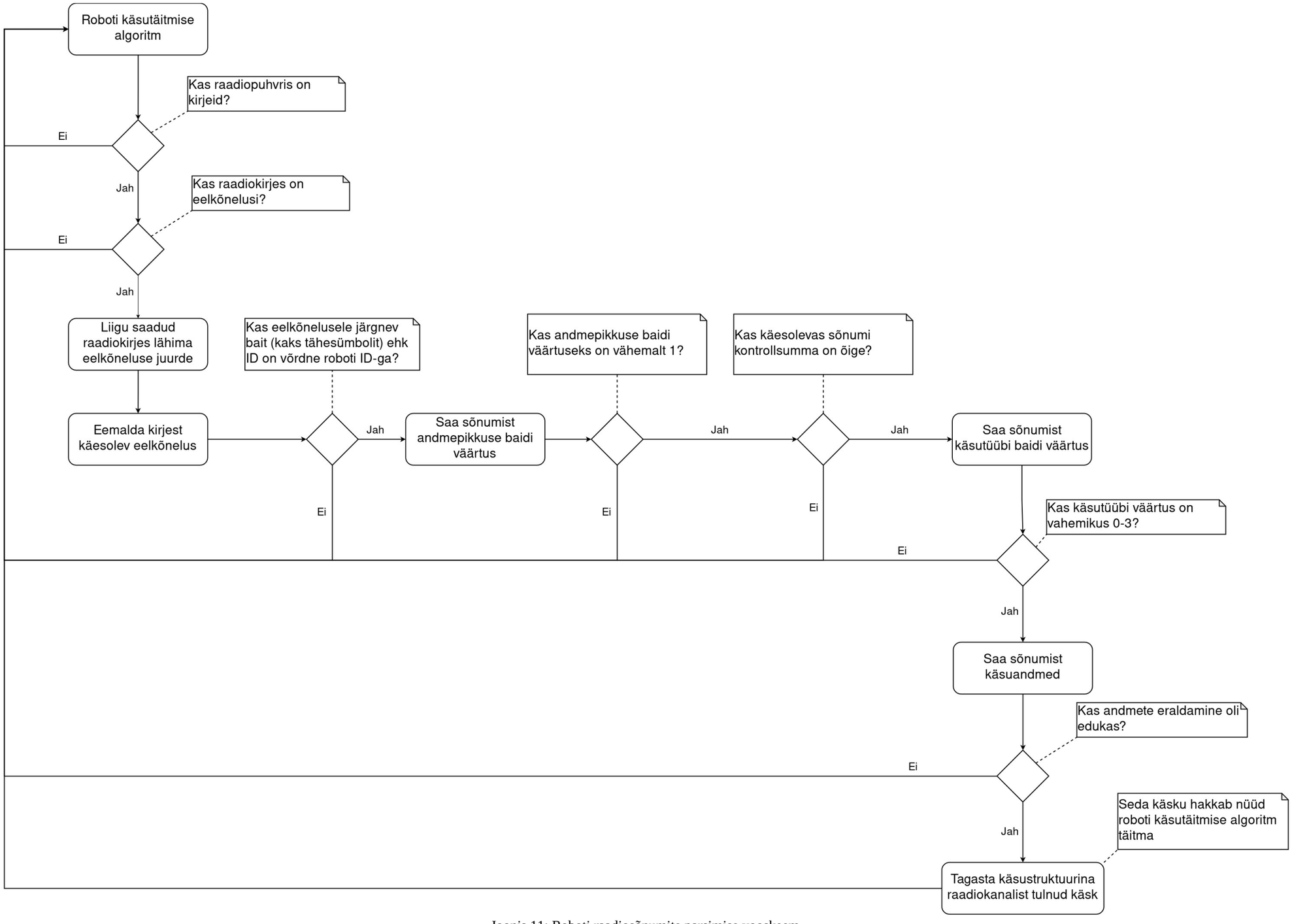
- 00 00 0A 02 03 **-5A,C8** 85 – „keera” käsk robotile 10 ($0A_{16} = 10_{10}$), kus robot peab keerama ennast praegusest olevast positsioonist 90 ($5A_{16} = 90_{10}$) kraadi vastupäeva (miinusmärk tähistab vastupäeva pööramist), kusjuures mootorite kiirus pööramisel peab olema 200 ($C8_{16} = 200_{10}$).

Kuna iga robot saab eelpool mainitud pika kirje raadiokanalist, siis on parsimine tehtud just nii, et robot käib sõnumi läbi kasutades ära eelkõnelusi. See tähendab, et parser „hüppab” kirjes alati iga eelkõneluse juurde ja kontrollib, kas sellele järgneb tema ID (või üleüldine aadress FF) – kui järgneb, siis on sõnum mõeldud just antud robotile ja parsimine jätkub kasutades ära suhtlusprotokolli reegleid ja ka sõnumis olevat informatsiooni (näiteks andmeosa pikkus). Kui eelkõnelusele ei järgne roboti ID-d, siis järgmisel roboti käsutäitmise tsüklil „hüpatakse” järgmise eelkõneluse juurde ja sama protsess kordub (Joonis 11). Kui saadud sõnum on millegipärast vigane või kui parsimine millegipärast ebaõnnestub jätkab robot viimasena antud käsu täitmist (kui just viimane käsk teatud käskude tüüpide puhul juba täidetud ei ole). Lisaks, on robotil ka turvameede – kui robot ei ole teatud aja jooksul mingit käsku saanud, siis katkestab ta käesoleva käsu täitmise ning jääb ootama uut käsku (Joonis 12).

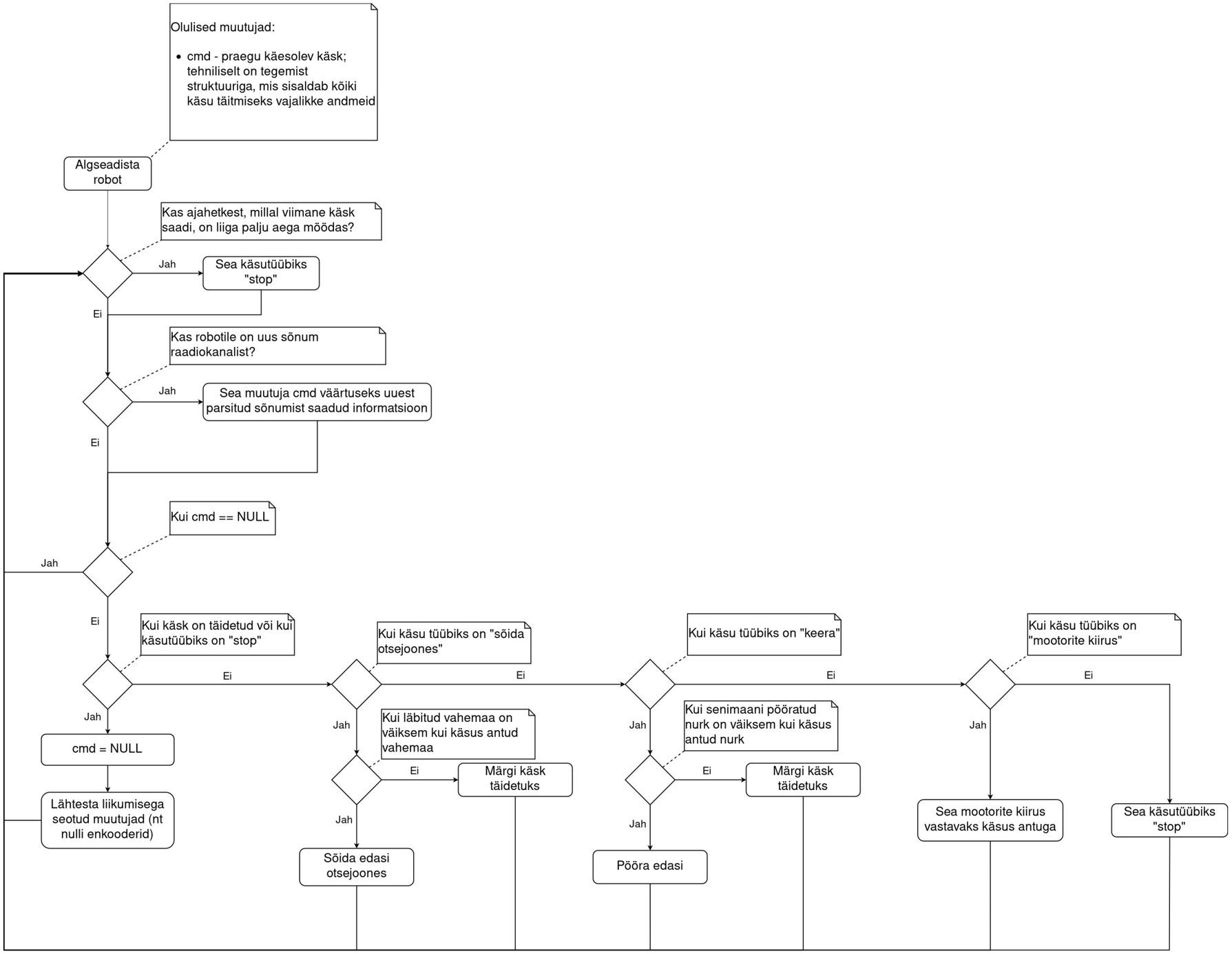
4.3 Peatüki kokkuvõte

Peatükis anti ülevaade:

- milliste raadiokäskude alusel toimub robotite juhtimine (peatükk „4 Robotite juhtimine”);
- kuidas toimib robotite juhtimine PID kontrolleri alusel (peatükk „4.1 PID kontroller mängus”);
- kuidas on rakenduskihi suhtlusprotokoll disainitud (peatükk „4.2 Koostatud protokoll raadiosides”).



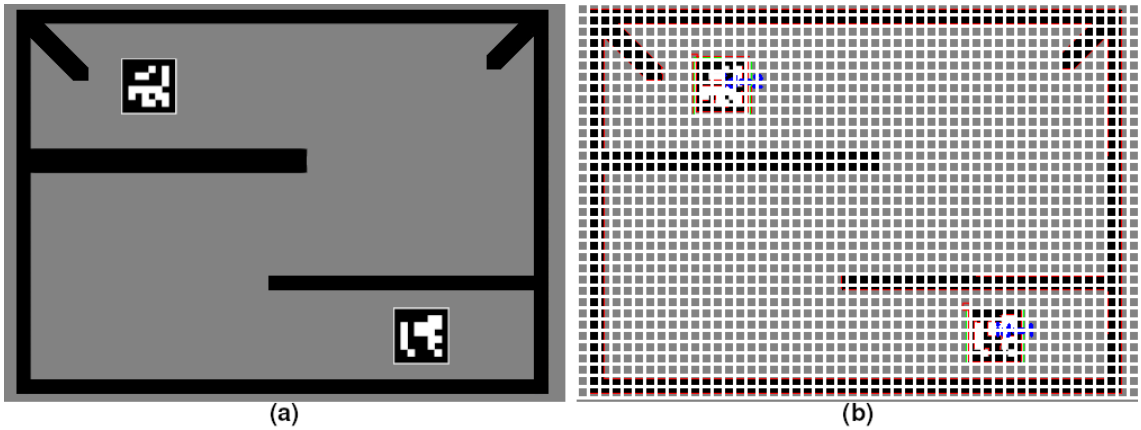
Joonis 11: Roboti raadiosõnumite parsimise vooskeem



Joonis 12: Roboti käsutäitmise algoritmi vooskeem

5 Takistuste tuvastus mänguväljal

Takistuste (eelkõige seinade) tuvastamine on tihedalt seotud tegelikult rajaleidmisega (*pathfinding*) – nimelt takistuste vältimiseks ongi rajaleidmist vaja teha. Järelikult peavad takistused olema kujul, millest mängus kasutatav rajaleidmisalgoritm A* (*A-star*) aru saaks. Takistuste tuvastamine mängus peaks olema ka dünaamiline – see tähendab, et kui teha mänguväljakul takistuste osas muudatusi, siis oleks võimalik need ilma suurema vaevata ära tuvastada. Lisaks, kuna tegemist on ikkagi mänguga, mille osad eksisteerivad päris maailmas, siis võib ikka juhtuda, et keegi näiteks läheb kaamera vastu ja nihutab selle paigast – järelikult on dünaamiline takistuste tuvastamine oluline ka projekti reaalsel kasutamisel. Kuna A* algoritm suudab üsna edukalt töötada koordinaadistikus (*grid*) [17] [18] ja kuna koordinaadistikus on võimalik ka ära implementeerida nõutud dünaamilisus, siis otsustatigi mängu luua kordinaatvõrgustik (Joonis 13). Kuigi võrgustiku osadeks võivad olla erinevad kujundid (näiteks kolmnurgad, kuusnurgad jne), siis kõige lihtsam on kasutada ruute, kuna ruudustik on intuitiivne, mängude programmeerimises kõige enam kasutust leidnud ja ka ruudustiku kohta on olemas teada tuntud valemeid nagu näiteks Manhattani kaugusvalem [19] .



Joonis 13: Seinade tuvastuse ja rajaleidmise prototüüpimiseks jaoks kasutatud pilt (a) ja sama pilt koos välja joonistatud ruudustikuga (b)

5.1 Implementeeritud takistuste tuvastus

Takistuste tuvastamise arendamisel kaaluti kahte võimalust:

1) kas luua kaamerast saadud pildi põhjal binaarpilt (pilt, kus on ainult mustad ja valged pikslid), kus siis näiteks mustad alad tähistaksid takistusi ja valged alad neid osi mänguväljast, kus robot liikuda võib või

2) kasutada ära OpenCV LSD algoritmi [20] , kus takistuste ääred oleksid määratud tuvastatud lõikudega.

Esimese variandi puhul oleks ruudustikus takistusi märgitud nii, et kui ruut ehk sõlm (*node*) koordinaadistikus sisaldab piisavalt palju musti piksleid (oletades, et takistuse värviks oleks valitud must, mitte valge), siis on tegemist ruuduga, mis on takistus. Teise variandi puhul oleks läbi LSD kontrollitud, kas tuvastatud lõigud läbivad kontrollitavat ruutu – kui vähemalt üks lõik läbib antud ruutu (või on ruudu sees), siis on käesolev sõlm takistus.

Pärast mõnda kaalumist, otsustati LSD kasuks (Joonis 14.b), kuna LSD ei nõua suuremat kalibreerimist – piisab, kui mänguväljakul märgitud takistused oleksid piisavalt tagataustast erinevad. Binaarpildi puhul oleks olnud vaja paika panna lävend (*threshold*), mille järgi otsustatakse, kas mingi piksel on valge või must. Antud lävend suure tõenäosusega sõltub valgustusest ja kuna mängu kasutatakse kohtades, kus valgustus võib väga palju erineda, siis see oleks nõudnud kalibreerimist. Lisaks, oleks nõudnud ka kalibreerimist parameeter, mille järgi ruudustikus märgitakse, kas tegemist on takistusega või mitte, kuna ühe ruudu suurust koordinaadistikus on võimalik muuta ning seda isegi siis, kui kasutada ruudu takistuse määramiseks mingisugust suhtarvu (näiteks protsenti), mis ei sõltu ruudu enda mõõtmetest (kuna kui valgustus erineb, siis erineb ka määratav suhtarv).

Muidugi on LSD algoritmi kasutamisel oma negatiivne pool. Nimelt, kui binaarpildis oleks ruudu määramine takistuseks teoreetiliselt üsna lihtne (kui ruudus olevate mustade pikslite arv on määratud piirist kas suurem või väiksem), siis LSD algoritmi puhul peab kindlaks tegema, kas tuvastatud takistuste lõigud läbivad koordinaadistikus olevaid ruute, mis on programmeerimise seisukohalt palju keerulisem. Selleks kasutati Martin Thoma poolt välja arendatud algoritmi [21] .

5.1.1 Martin Thoma poolt välja arendatud algoritm

Martin Thoma poolt välja arendatud algoritm tegelikult ei kontrolli, kas lõik läbib ruutu, vaid hoopis kontrollib, kas kaks lõiku lõikuvad üksteisega või ei ja kuna ruut koosneb neljast võrdset küljest/lõigust, siis on võimalik seda ka ruutude peal ära kasutada. Martin Thoma algoritmi tuumaks on kolm tingimust:

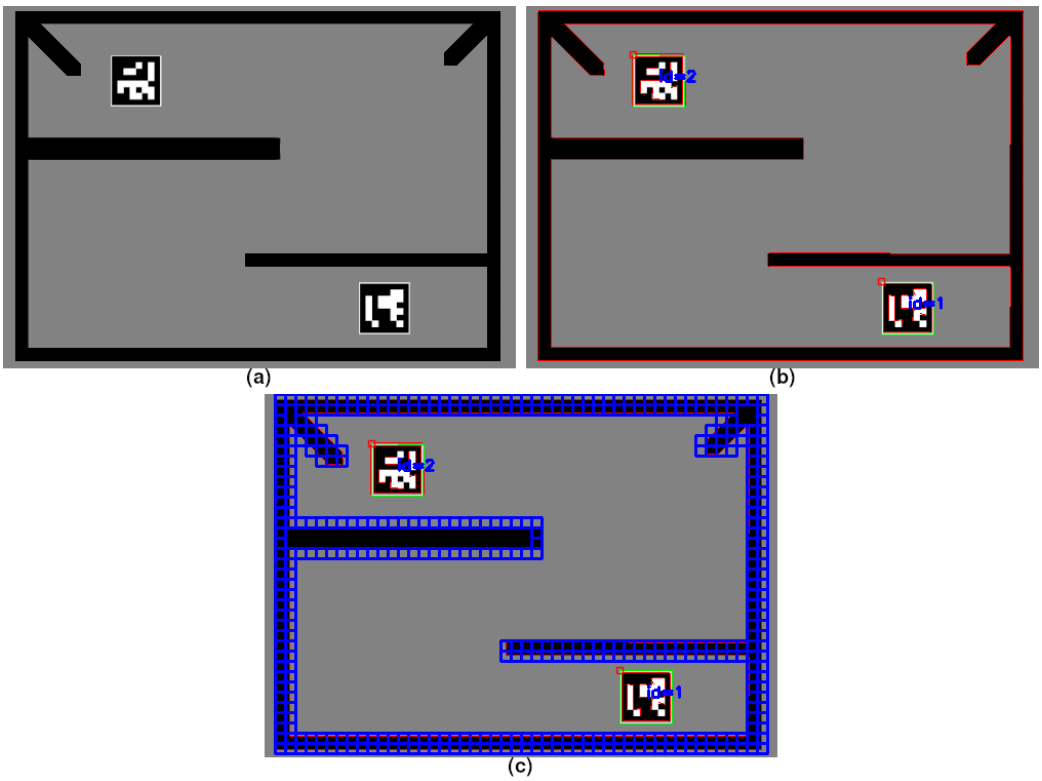
1) kas lõikude piirkastid (*bounding box*; tegemist on siis mõttelise riskülikuga, mille diagonaaliks on antud lõik) „lõikuvad" üksteisega – tegemist on üsna lihtsa tingimusega (olgu A ja B kaks erinevat ristkülikut, kusjuures A diagonaaliks on LSD algoritmi poolt tuvastatud lõik ja B diagonaaliks on kontrollitava ruudu üks külg ning ristkülikutel olgu defineeritud ülemine vasak punkt kui $(x_1; y_1)$ ja alumine parem punkt olgu defineeritud kui $(x_2; y_2)$): $(A.x_1 < B.x_2) \wedge (A.x_2 > B.x_1) \wedge (A.y_1 < B.y_2) \wedge (A.y_2 > B.y_1)$ [22] ;

2) kui esimene lõik muuta sirgeks/kiireks, kas see lõikab teist lõiku;

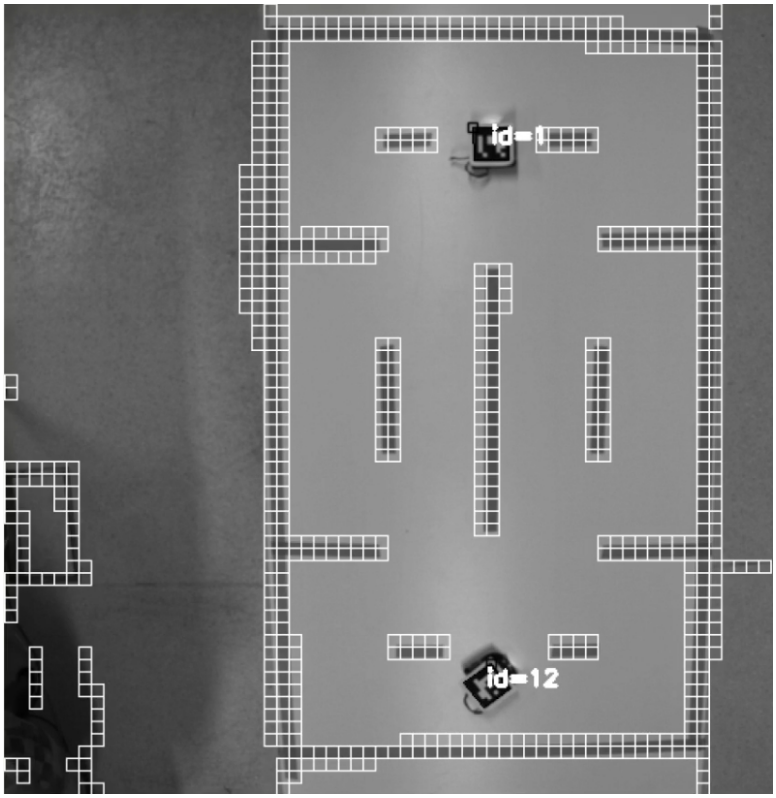
3) kui teine lõik muuta sirgeks/kiireks, kas see lõikab esimest lõiku.

Kui kõik kolm tingimust on tõesed, siis lõigud lõikuvad üksteistega – kui vähemalt üks tingimus ei ole tõene, siis lõikumist ei toimu. [21]

Programmis on eelkirjeldatud algoritm implementeeritud (Joonis 14.c ja Joonis 15) nii, et kontrollitakse kas koordinaadistiku iga ruudu iga külg lõikub iga LSD algoritmi poolt tagastatud lõiguga ning veel on juures lisatingimus, et kui peaks juhtuma, et LSD lõik asub ruudu sees, siis loetakse seda ruutu ka takistuseks. Ilmselgelt võib aru saada, et kontrollimaks, kas koordinaadistiku iga ruudu iga külg lõikub iga LSD lõiguga on üsna ajamahukas protsess – selletõttu tehakse takistuste tuvastus ühe korra mängu alguses ning takistuste asukohad jäetakse meelde. Kui on soov uuesti takistusi tuvastada, siis saab seda teha lihtsalt läbi mängu restarti.



Joonis 14: (a) Prototüüppilt; (b) LSD (punaste) lõikudega pilt; (c) Takistuste tuvastus ruudustikus



Joonis 15: Seinte ja ArUcode tuvastus programmis (seinad on märgitud sinise maalriteibiga valgele plaadile)

6 Rajaleidmine

Nagu sai punktis 5 mainitud, on rajaleidmist (*pathfinding*) mängus vaja selleks, et robotid liiguksid sihtmärgi suunas nii, et nad väldiksid mänguväljal olevaid takistusi/seinu. Rajaleidmine on üks klassikalisemaid probleeme mängude programmeerimises ning selle lahendamiseks on palju algoritme [23] .

6.1 Rajaleidmisalgoritmidest

Kuna rajaleidmisalgoritme või õigemini rajaleidmisalgoritmide variante on üsna palju (DFS, BFS, Dijkstra Algoritm, A*, D* ja lisaks masinõppe algoritmid [23]) ja kuna töös on otsustatud kasutada A* algoritmi, mis on mängude programmeerimises kõige enim kasutust leidnud algoritm [23] , siis keskendutakse eelkõige nendele algoritmidele, millele A* baseerub [24] ja loomulikult ka A* iseendale.

Oluline on ka meeles pidada, et kuigi töös on praegu ja ka varasemalt antud teemat nimetatud kui „rajaleidmiseks” (mis mängu kontekstis sobib ideaalselt), siis üldisemas pildis on antud algoritmid tegelikult osa graafiteooriast, kus sarnaselt mängude programmeerimisele üritatakse leida kõige kiirem või kõige vähem resurssi kasutav tee ühest graafipunktist teise. Koordinaatvõrgustiku (Joonis 13), mida mängus kasutatakse, võib vaadelda kui graafi erijuhtimina. [18] Lihtsuse mõttes on vaadeldud ja vaadeldakse töös edaspidi ainult mängu programmeerimise jaoks olulisi aspekte ehk eeldatakse, et algoritm töötab kui rajaleidjana kordinaatvõrgustikus.

6.1.1 BFS

BFS on algoritm, mis otsib rada sihtmärgini käies läbi iga sõlme (*node*; näiteks ruut ruudustikus) kordinaatvõrgustikus. Algoritmi ideeks on alustada valitud algussõlmest, vaadata üle algussõlme naabrid (see tähendab, et lisada need leitud sõlmede hulka, kui need juba ei ole leitud) ja siis võtta üks nendest algussõlme naabritest ning vaadata omakorda selle naabri naabreid (ehk jällegi lisada need leitud sõlmede hulka, kui neid seal juba ei ole) jne. Neid sõlmi, mis on koordinaadistikus märgitud kui „takistus”,

ignoreeritakse ehk nende naabreid ei vaadata. Oluline on ka see, et algoritmis jäetakse meelde, kuidas on sõlmed üksteisega seotud – see tähendab, et iga läbi käidud sõlme kohta on teada, millise sõlme naaber ta on. Protsess kestab, kuni kas kõik koordinaadistiku sõlmed on läbi vaadatud või kui sihtmärk on üles leitud. Kui sihtmärk leitakse üles, siis rada luuakse just selle meelde jäetud info põhjal – rada konstrueeritakse alustades sihtmärgi sõlmest ning liikudes „tagasi" algussõlmeni kasutades ära just naabrite seoseid koordinaadistikus. Kuigi BFS on hea algoritm selles mõttes, et see leiab koordinaadistikus alati kõige lühema tee sihtmärgini, siis on BFS-il ka omad miinused. Esiteks, kui algussõlme ja sihtmärgi sõlm on üksteisest kaugel, siis võib juhtuda, et BFS käib läbi terve või suurema osa koordinaadistikust, mis muidugi teeb algoritmi aeglaseks, ja teiseks, võib juhtuda, et mängus ei ole kõik sõlmed võrdsed: see tähendab, et osade sõlmede läbimiseks kulub rohkem aega või muid ressursse – seega rada, mis sõlmede arvu (ja ka distantsi) poolest on küll pikem, võib olla ajaliselt või mõne muu ressursi kohapealt parem. Kui tuua näide päris maailmast, siis näiteks otse läbi soo liikudes võib küll teekond olla vahemaa poolest lühem, kuid arvatavasti on kiiremaks ja mõistlikumaks teeks rada, mis teeb soole ringi peale. [24]

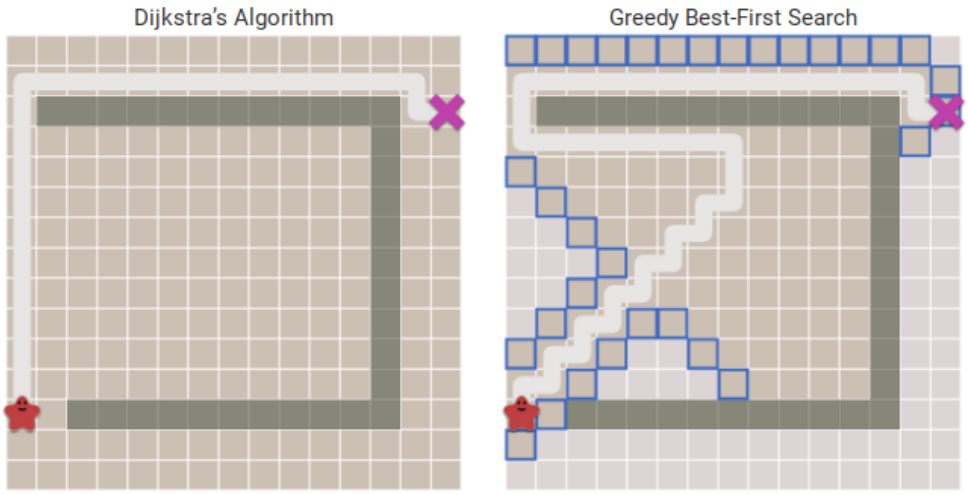
6.1.2 Dijkstra algoritm

Dijkstra algoritm (või ka *Uniform Cost Search*) on BFS-i edasiarendus just selles osas, et see arvestab, et kõik sõlmed ei ole võrdsed – see tähendab, et sõlmedele liikumine „maksab". Sõlmede „maksumus" sõltub just mängust ja selle disaini otsustest, kuid tihtipeale ongi mängudes näiteks vees liikumine aeglasem (ehk „kallim") kui maapinnal jne. Kuigi Dijkstra algoritmis toimub lõpliku raja konstrueerimine täpselt samamoodi nagu BFS-is, siis otsimisprotsess on ka peaaegu täpselt samasugune, aga nüüd sorteeritakse/valitakse naabrid selle järgi, millel on kõige väiksem maksuvus. [24]

6.1.3 Greedy Best-First Search

Dijkstra algoritm lahendab selle probleemi, et kõik sõlmed ei pruugi olla mängus võrdsed, kuid kuna Dijkstra ikkagi põhineb BFS-il, siis jääb alles probleem, et kui algussõlm ja sihtmärgi sõlm asuvad üksteisest kaugel, siis võib juhtuda, et terve koordinaadistik otsitakse läbi, mis muidugi muudab rajaleidmise ajakulukaks. *Greedy Best-First Search* on BFS-il põhinev algoritm, kus sõlmi ei prioriseerita mitte nendele liikumise „hinna" järgi, vaid hoopis selle järgi, kui kaugel on käesolev sõlm sihtsõlmest.

Mida lähemal on käesolev sõlm sihtsõlmele, seda parem ja mida kaugemal, seda halvem. See kõik tähendab seda, et *Greedy Best-First Search* on palju kiirem kui BFS (ja ka siis Dijkstra algoritm), kuna põhimõtteliselt ignoreeritakse sõlmi, mis on sihtmärgist kaugemal. Hindamaks, kui kaugel on käesolev sõlm sihtsõlmest on mitu võimalust (tegemist on hinnanguga, kuna takistuste tõttu me tegelikult ei tea, kui kaugel sihtsõlm on) – saab kasutada klassikalist kahe punkti kauguse valemit (olgu A esimese ruudu keskpunkt ja B teise ruudu keskpunkt: $d=\sqrt{(A.x-B.x)^2+(A.y-B.y)^2}$), kuid kuna (ujuvkomaarvude) ruutu võtmine ja sellest juure võtmine on arvutitehnikas ajamahukad protsessid, siis on olemas ka teisi lahendusi nagu näiteks Manhattani kaugus (olgu A esimese ruudu indeks ja B teise ruudu indeks ruudustikus): $d=|A.x-B.x|+|A.y-B.y|$. *Greedy Best-First Search* aga ei ole perfektne – esiteks, on siin ignoreeritud varem mainitud sõlmele liikumise hinda (kuid seda oleks lihtne lahendada implementeerides *Greedy Best-First Search* algoritmi koos Dijkstra algoritmiga ehk võttes arvesse nii sõlmele liikumise hinda kui ka kaugust sihtmärgist) ja teiseks, ei leia enamasti *Greedy Best-First Search* kõige lühemat teekonda, kuna kauguse hinnang omaette ei võta arvesse takistusi koordinaadistikus – olenevalt takistuse kujust võib rada muutuda võrreldes BFS-iga (või ka siis Dijkstra algoritmiga) palju pikemaks (Joonis 16 – oluline detail joonisel: tumedamad ruudud on need, mis algoritm on läbi käinud ehk *Greedy Best-First Search* on otsinud palju vähem ruute kui Dijkstra algoritm). [24]



Joonis 16: Dijkstra/BFS algoritmi (vasakul) ja *Greedy Best-First Searchi* (paremal) tulemus. Tumehallid ruudud on takistused, valge on leitud rada, täheke on otsingu algussõlm ja rist on sihtmärk. [24]

6.1.4 A* algoritm

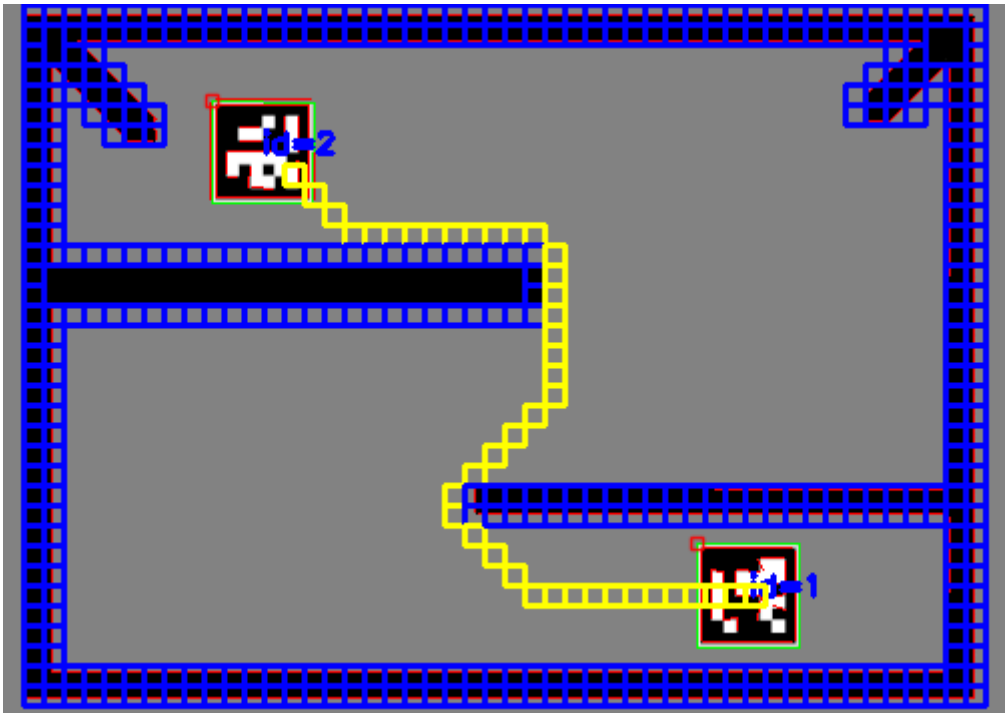
Dijkstra algoritm leiab üles kõige lühema raja sihtmärgini, kuid selle leidmiseks kulutab aega teekondadele, mis ei ole kõige optimaalsemad. *Greedy Best-First Search* leiab teekonna sihtmärgini kiirelt ehk otsides ainult nendes suundades, mis tegelikult ka lähenevad sihtmärgini, kuid enamasti ei ole leitud rada kõige lühem. A* algoritm üritab aga teha mõlemat – leida kõige lühem teekond sihtmärgini kõige väiksema ajakuluga (Joonis 17 – tumedamad ja numbritega ruudud on need, mille vastav algoritm on läbi käinud). A* saavutab selle hinnates nii kaugust sihtmärgist (nagu teeb seda *Greedy Best-First Search*), kuid samal ajal võtab arvesse ka kaugust algussõlmest. [24] Lisaks, on võimalik lisada A* hinnagusse ka sõlmele liikumise hind, nagu teeb seda Dijkstra algoritm. Kõiki hinnanguid on tegelikult väga lihtne arvesse võtta – piisab nende summast. Oletame, et sõlmel on kaks naabrit: ühe naabri kaugus algussõlmest on 20 ja kaugus sihtsõlmest on 14 ja teise naabri kaugus algussõlmest on 20 ja kaugus sihtsõlmest on ka 20 – on ilmselge, et parem naaber on esimene naabersõlm. [25] Oluline on muidugi see, et sihtsõlme kauguse hinnang oleks enam-vähem täpne (kaugust algussõlmest on täielikult teada) – kui see hinnang on vale, siis A* ei anna tulemuseks kõige optimaalsemat rada. [24]

Joonis 17: Dijkstra algoritm (vasakul); Greedy Best-First Search (keskel); A* algoritm (paremal) [24]

6.2 A* algoritmi implementeerimisest programmis

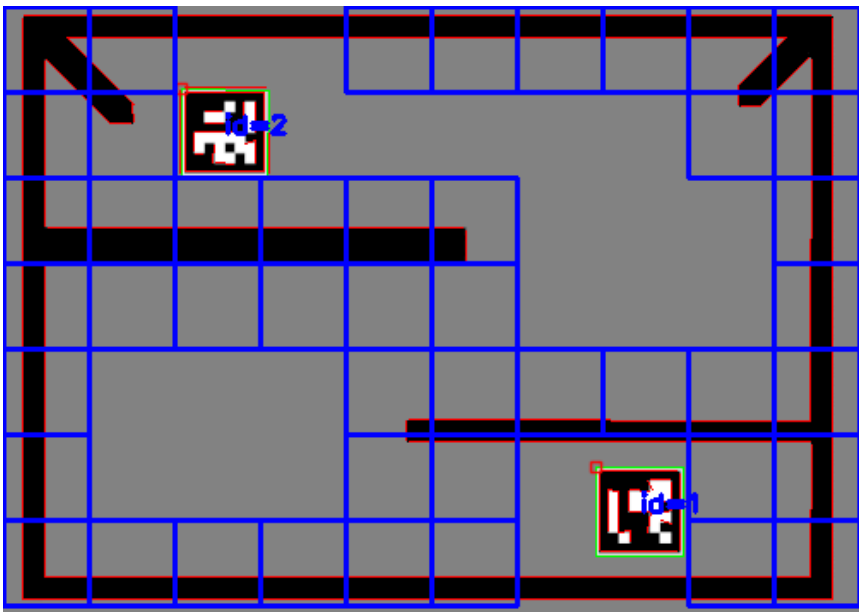
A* otsustati mängus just kasutada selletõttu, et ta on üks optimaalsemaid rajaleidmisalgoritme. Lisaks, oli A* implementeerimise kasuks ka see fakt, et soovi korral on võimalik lisada üsna lihtsalt erinevaid hinnanguid algoritmi. Kui programmi takistuste tuvastus koos rajaleidmisega oli prototüüpimisfaasis, siis avastati probleem, mis tavaliselt mängude rajaleidmisel probleemiks ei ole – nimelt oli näha, et lihtsalt A*

implementeerimisel hakkasid robotid seinu lõikama (Joonis 18), seda eriti, kuna kõige mugavam oleks rajaleidmise abil robotit juhtida, kui järgitav rada jääks roboti keskele.



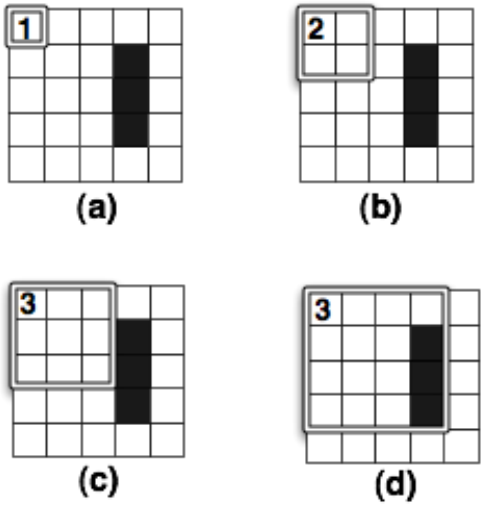
Joonis 18: Tavaline A* algoritm implementeeritud prototüüpimisel (rada genereeritakse ülemisest ArUcost alumisele ehk alumine ArUco on sihtmärk)

Antud probleemi vältimiseks tehakse enamasti mängudes ruudustik nii suureks, kui on selles liikuvad objektid. Kui mängus eksisteerivad erineva suurusega objektid, siis kasutatakse ühe ja sama mänguvälja kohta mitut erineva suurusega ruudustikku – ühe ruudustiku põhjal liiguvad siis näiteks väiksema suurusega objektid ja teise ruudustiku põhjal liiguvad suuremad objektid. Loodavas Pac-Man mängus aga ei saa teha ruudustiku ruute nii suureks kui on robotid, kuna siis muutuks seinade tuvastus väga ebatäpseks (Joonis 19). Mida suuremaks teha ühe ruudu suurus, seda ebatäpsem on takistuste tuvastus (kuid seda kiirem on rajaleidmine); mida väiksemaks teha ühe ruudu suurus, seda täpsem on takistuste tuvastus (kuid seda aeglasem on rajaleidmine, sest läbi käidavaid ruute on rohkem). Lisaks, ei ole võimalik kasutada mitme suurusega ruudustikuga lahendust, kuna liikuvaid objekte (ehk roboteid) on ainult ühe suurusega.

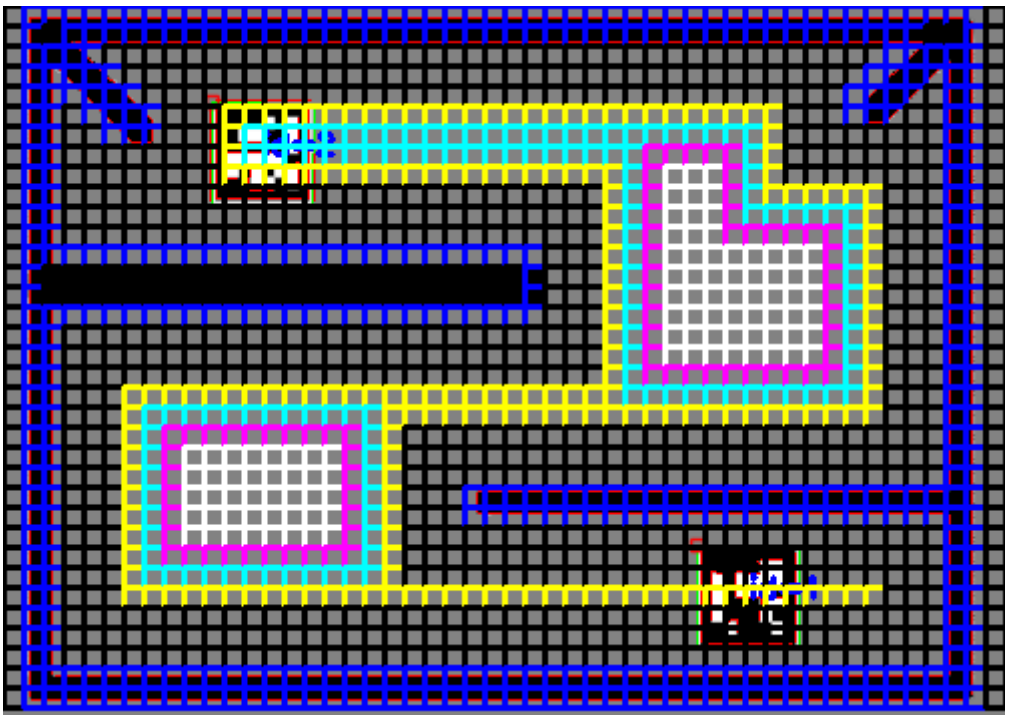


Joonis 19: Ebatäpne takistuste tuvastus (ruudustik on liiga suur) - antud juhul rada ülemisest robotist alumisele ei eksisteeri

Antud probleemi lahenduseks on kasutada kliirensiga (*clearance based*) A* rajaleidmisalgoritmi. Nimelt, kliirensiga A* töötab nii, et igale ruudule koordinaadistikus antakse uus lisaparameeter – kui suure objekti suudab ruut endasse tegelikult mahutada. Antud parameetrit võib vaadelda ka kui kaugust lähimast takistusest. Mõlemal juhul on oluline meeles pidada, et mahutava objekti „suurus" või „kaugus" lähimast takistusest ei ole mitte otseselt füüsikaline suurus, vaid on pigem väljendatud ruudustiku põhiselt – näiteks ruudule lähim takistus on kolme ruudu kaugusel, seega on antud ruudu kliirensiks kolm (Joonis 20 ja Joonis 21). [26]

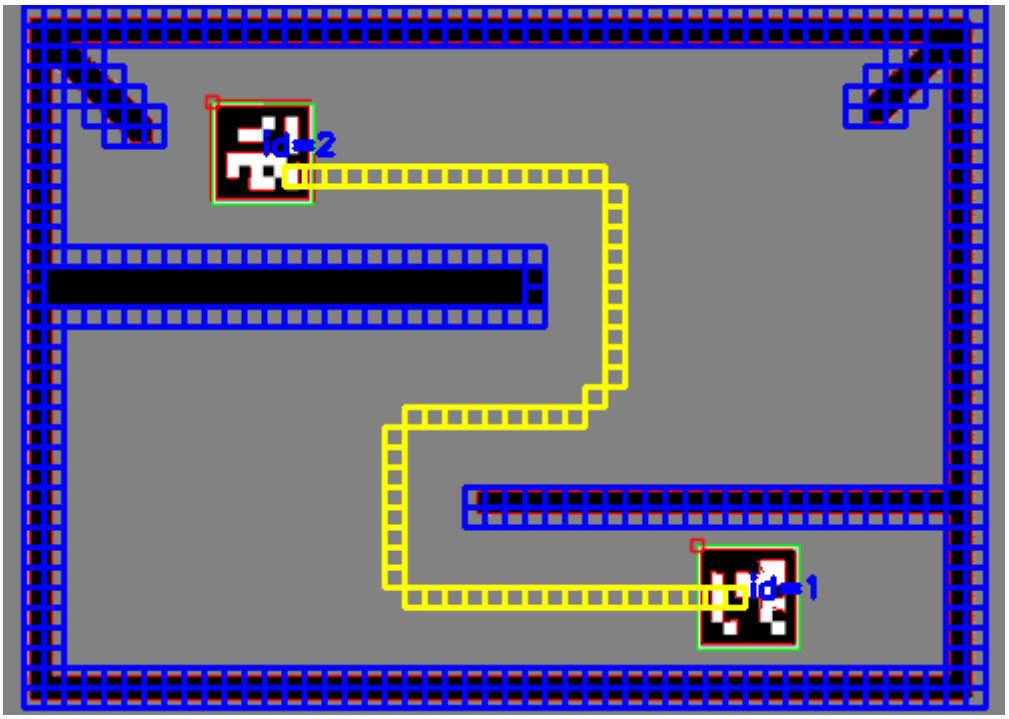


Joonis 20: Sõlme kliirensi leidmine [26]



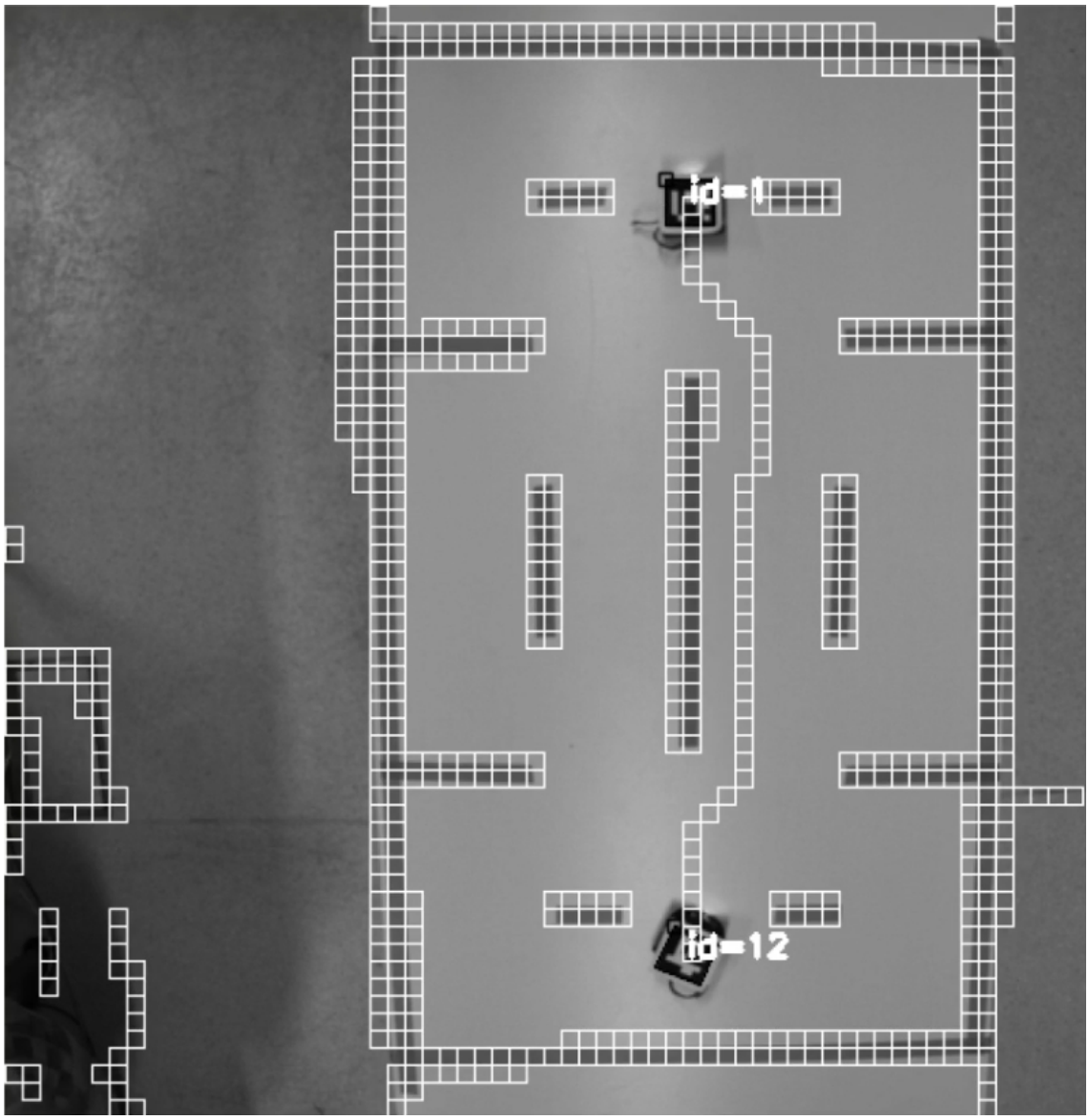
Joonis 21: Ruutude kliirensid prototüübis (mustad ruudud on alla 3 kliirensiga; kollased ruudud on kliirensiga 3; helesinised 4; lillad 5; valged 6 ja enam)

Lõpuks, kui võtta kliirens arvesse A* algoritmis, siis on võimalik roboteid rajaleidmise abil robotit juhtida nii, et järgitav rada jääb roboti keskele ilma, et robot sõidaks pooleldi seinadesse sisse (Joonis 22 ja Joonis 23).



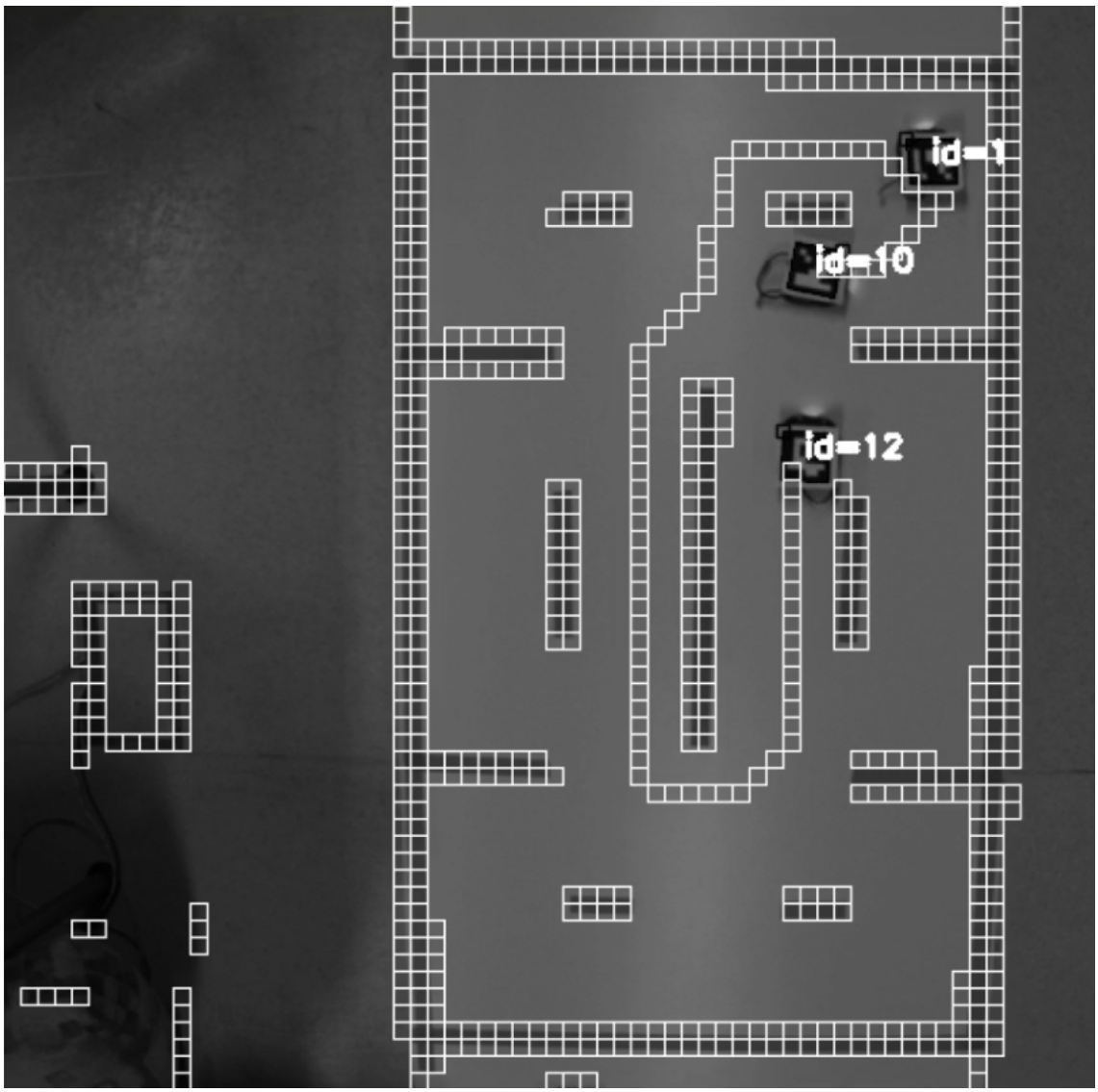
Joonis 22: A* rajaleidmisalgoritm prototüübis koos kliirensiga (vähim lubatud kliirens on 3)

Päris programmis näeb kliirensiga A* algoritm välja järgnev:



Joonis 23: Programmis kliirensiga rajaleidmine (robotite/ArUcode keskelt minevad valged ruudud on leitud rada; ArUco ID-ga 1 on sihtmärk ehk mängija poolt kontrollitud robot)

Kliirensi kasutamine A* algoritmis lahendab seinade lõikamise probleemi, kuid mängus kasutatakse ühe vastasroboti asemel mitut vastasrobotit. Seega peab rajaleidmisel arvestama ka sellega, et teised robotid (välja arvatud mängija robot) on takistused. Õnneks on ka seda üsna lihtne A* algoritmi lisada – neid sõlmesid, mis on tuvastatud ArUcodest teatud raadiuse kaugusel ignoreeritakse rajaleidmisel nagu ka teisi takistusi ehk seinu (Joonis 24).



Joonis 24: Rajaleidmine koos mitme robotiga

Oluline detail veel implementeeritud rajaleidmisel on see, et kuigi A* on üsna optimiseeritud algoritm, siis siiski tõuseb rajaleidmisele kulutatud aeg iga lisa robotiga eksponentsiaalselt, aga kuna rajaleidmist peab tegema mängu kiiruste suhtes väga harva (näiteks praegu on rajaleidmise sagedus umbes iga poole sekundi tagant), siis see ei mõjuta reaalsuses mängu jõudlust.

Lisaks, vajab mainimast ka viis, kuidas robotid rada jälgivad – rada jälgitakse põhimõtteliselt sõlm-sõlme haaval. Igale (vastas)robotile on võimalik mängus ette anda sihtmärk, kuhu robot sõitma peab. Rajaleidmisel uuendatakse seda sihtmärki nii, et kui robot on rajal mingi sõlmeni jõudnud, siis seatakse antud robotile sihtmärgiks järgmine sõlm ja nii edasi, kuni lõpuks mängijarobot „kinni püütakse”.

6.3 Peatüki kokkuvõte

Peatükis anti ülevaade:

- miks rajaleidmist on mängus vaja (peatükk „6 Rajaleidmine”);
- milliseid rajaleidmisealgoritmid on olemas ja tehti ülevaade nendest algoritmidest, millele baseerub A* rajaleidmisalgoritm (peatükk „6.1 Rajaleidmisalgoritmidest”);
- kuidas A* algoritm programmis implementeeriti (peatükk „6.2 A* algoritmi implementeerimisest programmis”).

7 Ideid edasiarenduseks

Kuigi valminud mängusüsteemi on võimalik erinevatel festivalidel edukalt kasutada, siis on siiski mängus aspekte, mis vajaks täiustamist. Esimeseks aspektiks on graafilise kasutajaliidese edasiarendus – seda nii kasutajasõbralikuse kui ka mänguvõimaluste poolest. Praegusel ajahetkel on programmis kaks akent – üks on mõeldud kasutaja sisendite lugemiseks ja teine näitab, mis mänguväljakul toimub (Joonis 25).

Joonis 25: Programmi kaks GUI akent ("botswarm" ja "out")

Käesolev kahe akna süsteem on segadust tekitav – seda eriti esmakordsetele kasutajatele. Lisaks, mänguvõimaluste poole pealt, võiks GUI-sse lisada punktipallid (nagu on päris Pac-Manis), mida mängijarobot saaks koguda, et tõsta enda skoori mängus. Praegusel hetkel on ka mängija skoori salvestamine keeruline GUI puuduste tõttu.

Teiseks aspektiks oleks lisada programmi rohkem mänge – kui programmis oleks rohkem mänge, siis see muudaks programmi veel atraktiivsemaks esindusprojektiks. Praegusel hetkel on võimalik mängida programmis kahte mängu – lihtne tagaajamise mäng (ehk põhimõtteliselt Pac-Man ilma takistuseta) ja siis loodud Pac-Mani laadne mäng.

8 Kokkuvõte

Töö põhieesmärgiks seati luua TTÜ Robotiklubile esindusprojekt, mida oleks võimalik kasutada erinevatel konverentsidel. Selle eesmärgi täitmiseks seati ka vaheülesanded:

1) selgitada välja, mida on vaja Pac-Mani sarnase mängu loomiseks robotitega;
2) tuvastada mängus kasutatavaid roboteid nii, et tuvastusinfo sisaldaks kõike, mida on vaja mängu ülesehitamiseks;
3) luua üldine viis robotite juhtimiseks;
4) tuvastada takistusi mänguväljakul;
5) kuidas roboteid juhtida nii, et need väldiksid tuvastatud takistusi.

Kõik seatud eesmärgid ja ülesanded saavutati. Esiteks, toodi välja, mida on vaja Pac-Mani sarnase mängu loomiseks robotitega – mängusüsteemis kasutati robotiteks TTÜ Robotiklubi enda universaalselt robotiplatvormi Pisi-Bot, robotite ja takistuste tuvastus käis kaamera XIMEA xIQ MQ013CG-E2 vahendusel, mänguajuks said loodud programm ja kommunikatsioonikanaliteks sai nii USB kui ka raadioside. Teiseks, selgitati, kuidas üldisemalt käib objektide tuvastus masinnägemisel ja valiti välja toodud meetoditest parim mängusüsteemi jaoks (ArUco piltkood). Kolmandaks, robotite juhtimiseks kasutati PID kontrollerit koos spetsiaalselt disainitud suhtlusprotokolliga. Neljandaks, takistusi tuvastati mänguväljakul kasutades ära ruudustiku, OpenCV LSD tuvastussüsteemi ja Martin Thoma poolt välja arendatud algoritmi abiga. Viiendaks, toodi välja takistuste vältimiseks erinevad rajaleidmisalgoritmid, nende erinevused ja kuidas A* algoritmi programmis implementeeriti.

Kasutatud kirjandus

[1] Pac-Man – Official Website. *History*. [www]. https://www.pacman.com/en/history/. Kasutatud: 08.04.2023.

[2] TTÜ Robotiklubi. *Pisi-Bot*. [www]. https://wiki.robotiklubi.ee/projektid/robotid/pisi-bot. Kasutatud: 08.04.2023.

[3] XIMEA. *MQ013CG-E2*. [www]. https://www.ximea.com/en/products/usb3-vision-cameras-xiq-line/mq013cg-e2. Kasutatud: 08.04.2023.

[4] Digi. *Digi XBee 3 Zigbee 3 RF Module*. [www]. https://www.digi.com/products/embedded-systems/digi-xbee/rf-modules/2-4-ghz-rf-modules/xbee3-zigbee-3. Kasutatud: 08.04.2023.

[5] Viso. *Object Detection in 2023: The Definitive Guide*. [www]. https://viso.ai/deep-learning/object-detection/ Kasutatud: 15.04.2023.

[6] Viso. *Image Recognition: The Basics and Use Cases (2023 Guide)*. [www]. https://viso.ai/computer-vision/image-recognition/ Kasutatud: 15.04.2023.

[7] Vacatronics – Medium. *What Is a Fiducial Marker*. [www]. https://medium.com/vacatronics/what-is-a-fiducial-marker-865799bc7266 Kasutatud: 15.04.2023.

[8] IBM Developer. *The basics of image processing and OpenCV*. [www]. https://developer.ibm.com/articles/learn-the-basics-of-computer-vision-and-object-detection/ Kasutatud: 14.05.2023.

[9] S. K. Nayar, *Binary Images*, [www]. https://cave.cs.columbia.edu/Statics/monographs/Binary%20Images%20FPCV-1-3.pdf. Kasutatud: 14.05.2023.

[10] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas ja M.J. Marín-Jiménez. "Automatic generation and detection of highly reliable fiducial markers under occlusion", 2014. Loetud aadressil: https://www.researchgate.net/publication/260251570_Automatic_generation_and_detection_of_highly_reliable_fiducial_markers_under_occlusion/link/59dda6daaca272b698e65055/download. Kasutatud: 15.04.2023.

[11] OpenCV Documentation. *Detection of ArUco Markers*. [www]. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html. Kasutatud: 15.04.2023.

[12] Oleg Kalachev. *Online ArUco Markers Generator*. [www]. https://chev.me/arucogen/. Kasutatud: 15.04.2023.

[13] OpenCV. *About*. [www]. https://opencv.org/about/. Kasutatud: 15.04.2023.

[14] XIMEA. *OpenCV – Vision Libraries – XIMEA Support*. [www]. https://www.ximea.com/support/wiki/vision-libraries/OpenCV. Kasutatud: 15.04.2023.

[15] K. J. Åstrom and R. M. Murray, *Feedback Systems – An Introduction for Scientists and Engineers*, 2nd ed. New Jersey: Princeton University Press, 2021.

[16] Digi. *Discover Zigbee Protocol 3.0*. [www]. https://www.digi.com/solutions/by-technology/zigbee-wireless-standard. Kasutatud: 18.04.2023.

[17] Amit Patel. *Amit's Thoughts on Pathfinding – Map representations*. [www]. https://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html. Kasutatud: 20.04.2023.

[18] Amit Patel. *Grids and Graphs*. [www]. https://www.redblobgames.com/pathfinding/grids/graphs.html. Kasutatud: 20.04.2023.

[19] Amit Patel. *Amit's Thoughts on Grids*. [www]. http://www-cs-students.stanford.edu/~amitp/game-programming/grids/. Kasutatud: 20.04.2023.

[20] OpenCV Documentation. *cv::LineSegmentDetector Class Reference*. [www]. https://docs.opencv.org/4.x/db/d73/classcv_1_1LineSegmentDetector.html Kasutatud: 20.04.2023.

[21] Martin Thoma. *How to check if two line segments intersect*. [www]. https://martin-thoma.com/how-to-check-if-two-line-segments-intersect/. Kasutatud: 20.04.2023.

[22] Silent Matt. *Rectangle Intersection Demonstration*. [www]. https://silentmatt.com/rectangle-intersection/. Kasutatud: 20.04.2023.

[23] R. Graham, Hugh McCabe and Stephen Sheridan, "Pathfinding in Computer GamesPathfinding in Computer Games", *The ITB Journal*, vol. 4, no. 2, 2003, doi: 10.21427/D7ZQ9J. Kasutatud: 21.04.2023.

[24] Amit Patel. *Introduction to the A* Algorithm*. [www]. https://www.redblobgames.com/pathfinding/a-star/introduction.html. Kasutatud: 21.04.2023.

[25] P. Lester, *A* Pathfinding for Beginners*, [www]. https://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf. Kasutatud: 21.04.2023.

[26] Shortest Path. *Clearance-based Pathfinding*. [www]. https://harablog.wordpress.com/2009/01/29/clearance-based-pathfinding/. Kasutatud: 21.04.2023.

Lisa 1– Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Oliver Paljak,

1 Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Pac-Mani laadse mängu loomine robotitega", mille juhendajad on Priit Ruberg ja Erki Meinberg

1.1 reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2 Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.

3 Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

15.05.2023

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Programmi lähtekood

Töökäigus loodud programmi lähtekoodi on võimalik vaadata TTÜ Robotiklubi GitHubist: https://github.com/TalTechRobotiklubi/Pacman.