TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Võrgutarkvara õppetool

# Agendi teekonna ja tegevuste planeerimine strateegiamängus

bakalaureusetöö

Üliõpilane: Markus Rondo
Üliõpilaskood: 112357
Juhendaja: Jaagup Irve

Tallinn
2016

# 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.

-------------------------------------------- --------------------------------------------

(*kuupäev*) (*allkiri*)

# Annotatsioon

Töö eesmärkideks on hinnata ja võrrelda erinevaid agendipõhiseid tehisintellekti lahendusi ning luua tulemuste põhjal strateegiamängu prototüüp. Lisaeesmärgiks on tehisintellekti lahenduste mõõtmismeetodi leidmine.

Mängu agendi tehisintellekti realiseerimine võib olla keeruline. Meetodeid, implementatsioone ning rakendusi on palju ja nende seast mõistlikumate lahenduste leidmine on probleemiks. Käesoleva töö prototüübi loomiseks tuleb leida sobivaimad strateegiamängu agendi tehisintellekti loomise meetodid. Tehisintellekti erinevateks elementideks on käesolevas töös teekonna leidmine, nägemine ja otsuste langetamine.

Paljud agendi teekonna leidmiseks mõeldud algoritmide variatsioonid on omavahel sarnased ja neid on lihtne objektiivselt võrrelda. Samuti on nägemise jaoks mõeldud algoritmidega. Otsuste langetamise algoritmid võivad aga olla põhimõttelt väga erinevad ning nende mõõtmine võib osutuda väga subjektiivseks.

Tulemusena sai loodud C++ programmeerimiskeeles strateegiamängu agendi tehisintellekti prototüüp. Prototüübi graafikamootor on loodud SDL abiteegiga. Sobivamate tehisintellekti loomise meetodite valimiseks on leitud aspektid, millega neid mõõta. Prototüüpi on implementeeritud sobivaimad teekonna leidmise ja nägemise algoritmid. Agendi otsuste langetamise jaoks arendas töö autor kohandatud erilaadseid meetodeid kasutava lahenduse.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 46 leheküljel, 5 peatükki ja 16 joonist.

# Abstract

The aim of this thesis is to evaluate and compare different agent based artifical intelligence approaches and create a prototype of a strategy game based on the results. An additional aim is to find and formulate a method of measuring different artificial intelligence methods.

Realization of an agents artificial intelligence in a game can be difficult. There are many methods, implementations and applications. Finding the more useful ones for a project is a problem. To create this thesis' prototype it is necessary to find the most suitable artificial intelligence methods for a strategy game. The artificial intelligence is composed of pathfinding, simulating vision and decision-making.

Several pathfinding algorithms are similar in nature and can be objectively compared. It is similar with the vision simulating algorithms. The decision-making algorithms can have very different principles and evaluating can become very subjective.

As a result a prototype of a strategy game artifical intelligence was created in the C++ programming language. The rendering motor of the prototype is created using the SDL library. Aspects were determined and formulated to find the better and more suitable artificial intelligence methods. The most suitable pathfinding and vision simulation algorithms are implemented into the prototype. The author of this thesis created a unique customized method for the decision-making approach of the prototype.

The thesis is in Estonian and contains 46 pages of text, 5 chapters and 16 figures.

# Lühendite ja mõistete sõnastik

| | |
|---|---|
| **Smoothing** | ***Smoothing***<br>Protsess, mis eemaldab liigsed nurgad agendi teekonnast. |
| **AA** | ***Any-angle***<br>Igasuunalised algoritmid leiavad teekondi, mille rajad on 360 kraadi ulatuses. |
| **Expert system** | ***Expert system***<br>Tehisintellekti loomise meetod, mis jäljendab kogenud asjatundja inimese käitumist ja valikuid mingis kindlas valdkonnas. |
| **Grid** | ***Grid***<br>Kahemõõtmeline võrgustik, mida mööda agent liigub. |
| **RTS** | ***Real-time strategy***<br>On strateegiamängu sort, kus juhitakse agente reaalajas. |

# Jooniste nimekiri

Joonis 1 - Dijkstra algoritmi progressi järk-järguline kujutus [1] ............ 15
Joonis 2 - A* algoritmi ühest nurgast teise teekonna leidmisel avatud ja suletud tipud heuristilise eukleidilise väärtusega (vasakul) ja 0 väärtusega (paremal) [1] ............ 16
Joonis 3 - A* algoritmi (vasakul) ja *any-angle* algoritmi (paremal) leitud teekonnad [20] ..... 18
Joonis 4 - Theta* (vasakul) ja Lazy Theta* (paremal) leitud teekonnad [20] ............ 19
Joonis 5 - Lihtne otsusepuu [1] ............ 26
Joonis 6 - Lihtne olekumasina UML diagramm [1] ............ 27
Joonis 7 - Keerulisem käitumispuu [1] ............ 28
Joonis 8 - Kolme *fuzzy logic* väärtuse tõusmine ja langemine temperatuuri muutudes [41] ... 29
Joonis 9 - Agendi võimalikud eesmärgid ning tegevused eesmärgipõhise käitumisega lahenduses. Võib juhtuda, et agent otsustab juua, kuigi see tekitaks kriitilise olukorra [1] ..... 30
Joonis 10 - Näide *shadowcasting* algoritmi poolt loodud varjust [49] ............ 34
Joonis 11 - Oktiilid, mille järgi *shadowcasting* algoritm ruute läbi töötab [49] ............ 34
Joonis 12 - Agendi põgenemisest tulenev tegevuste muutumine ............ 36
Joonis 13 - Toidu ja ohu situatsioon prototüübis ............ 37
Joonis 14 - Sõjaolukorra situatsioon prototüübis ............ 37
Joonis 15 - Maailma kujutus karakteritena ............ 45
Joonis 16 - Maailma graafiline kujutus prototüübis ............ 46

# Sisukord

Sissejuhatus ..... 9
1. Mängu agendi tehisintellekti mõõtmine ..... 10
  1.1 Kiirus ja mälu ..... 10
  1.2 Sobimus ..... 11
    1.2.1 Esilekerkiv käitumine ..... 11
    1.2.2 Oskuslikkus ..... 12
  1.3 Implementatsiooni keerulisus ..... 13
2. Liikumine ..... 14
  2.1 Dijkstra algoritm ..... 14
  2.2 A* algoritm ..... 15
  *2.3 Any angle* algoritmid ..... 18
3. Nägemine ..... 22
4. Otsuste tegemine ..... 24
  4.1 Strateegiamängu agendi struktuur ..... 24
  4.2 Probleemsed situatsioonid ..... 25
  4.3 Levinuimad agendi otsuste langetamise põhimõtted ..... 26
    4.3.1 Otsusepuu ..... 26
    4.3.2 Olekumasin ..... 26
    4.3.3 Käitumispuu ..... 27
    4.3.4 Määramatus ..... 28
    4.3.5 Eesmärgipõhine käitumine ..... 29
  4.4 Kohandatud otsuste langetamise algoritm ..... 31
5. Prototüübi implementatsioon ..... 32
  5.1 Üldine ..... 32
  5.2 Graafikamootor ..... 32
  5.3 Liikumine, Block A* ..... 32
  5.4 Shadowcasting ..... 33
  5.5 Kohandatud otsuste langetamise algoritmi implementatsioon ..... 35
Kokkuvõte ..... 39
Summary ..... 41
Kasutatud kirjandus ..... 42
Lisa 1 ..... 45

Lisa 2 ........................................................................................................................ 46

# Sissejuhatus

Mängude loomise puhul on üks keerulisemaid ülesandeid teha toimiv tehisintellekt. Tihti tuleb selleks palgata spetsialist või siis jätta see tegemata. Probleem on kriitika tulvas, mis tehisintellekti tabab ja tihti määrab mängu staatuse turul. Et luua edukat mängu on igasugune arendus sellel alal väga teretulnud. See on huvitav teema kõigile mänguloojatele, kelle töös on tehisintellekti vaja kasutada.

Luuakse agendipõhine strateegiamängu tehisintellekti prototüüp. Selle jaoks tuleb teha erinevates strateegiamängudes esinevate tehisintellektide analüüs ja meetodite võrdlus, et teada milliseid lahendusi rakendada. Sellest tulenevalt on vaja leida meetodid, millega oleks neid võimalik mõõta. Strateegiamängu prototüüp töötab C++ põhjal SDL abiteeki kasutades. Agendi töötamise jaoks on implementeeritud erinevate funktsioonidega algoritmid.

Kõigepealt määratakse ja defineeritakse atribuudid, mille põhjal erinevaid tehisintellekti lahendusi hinnatakse. Peale seda hinnatakse ja analüüsitakse agendi liikumiseks vajalikke teekonna leidmise algoritme ja agendile sisendi tekitamiseks tarvilikke nägemise algoritme. Siis määratakse ära töö prototüübi agendi olemus ja leitakse tema toimimisega seotud mõned võimalikud probleemid. Seejärel analüüsitakse populaarsemaid *decision-making* ehk otsuste langetamise algoritme. Nende lahenduste ja eelnevalt leitud võimalike kitsaskohtade põhjal luuakse kohandatud *decision-making* algoritm. Lõpuks on prototüübi implementatsiooni kirjeldus ning töö tulemused.

# 1. Mängu agendi tehisintellekti mõõtmine

Prototüübi agendi tegutsemist määravaid lahendusi on erinevaid. Optimaalse tulemuse saavutamiseks peaks eelistama paremaid ja sobivamaid tehisintellekti komponente. Järgnevas töös leiame aspekte, mille järgi on võimalik mõõta erinevaid tehisintellekti lahendusi.

## 1.1 Kiirus ja mälu

Mängudes kasutatava agendi tehisintellekt koosneb suurel määral erinevatest algoritmidest. Seetõttu on paljude lahenduste sobivuse mõõtmiseks võimalik kasutada samu vahendeid nagu enamike algoritmide puhul. Väga oluliseks faktoriks on paljudes mängudes agendi algoritmide kiirus. Tihti on vaja, et agent reageeriks mängija tegevusele koheselt. Kui reaalajas toimivates mängudes ei tööta tehisintellekt piisavalt kiiresti, siis olenevalt implementatsioonist võib see tekitada kas ebausutavat käitumist või mängu aeglustumist. [1]

Mõnedes mängudes on tehisintellekti toimimiseks tarvis jooksutada keerukamaid ja rohkem arvuti ressursse nõudvaid algoritme. Mängumootorid arvutavad mitu kaadrit sekundis, see mitu kaadrit mängija sekundis näeb sõltub monitori kaadrisagedusest, riistvara suutlikkusest ja mängu optimeeritusest. Pikemaid protsesse saab panna toimima üle mitme kaadri, aga see pole igas olukorras võimalik ning kui ühe kaadri ajal toimub liiga palju arvutusi, tekib viivitus ja mängu sujuvus saab kannatada. [2]

Teine oluline piirang tehisintellekti algoritmide puhul on mälu kasutus. Nii mõnedki kõige uuemad ja tehnoloogiliselt värskeimad tehisintellekti lahendused ei ole mängudes kasutatavad, sest need vajavad informatsiooni talletamiseks liiga palju ressursse. Populaarsemad kasutusel olevad lahendused kasutavad mälu mõistlikult. [1]

Mängu agendi tehisintellektis toimivate algoritmide kiiruse ja mälu kasutuse keerukust saab üsnagi objektiivselt ja täpselt mõõta suure O-tähistuse abil [3]. Mängude algoritmide kirjelduste puhul on O-tähistusele tuginevad keerukusklassid üsnagi üldised ja lihtsustatud, sest praktilise implementatsiooni jaoks pole liiga detailidesse vaja laskuda. Mängude puhul pole ka enamasti halvima juhtumi keerukus (*worst-case complexity)* oluline, sest mäng luuakse nii, et sellist keerukust nõudvat olukorda ei tekiks. Mõne keerulisema või mitme omavahel sarnase algoritmi puhul ei ole mõõtmine ainult O-notatsioonis praktiline. Seega vajaduse korral on ka käesolevas töös kasutatud algoritmide mõõtmiseks lisaks O-tähistusele ka katsete tulemusi ning muid struktuurseid omadusi. [1]

## 1.2 Sobimus

Eksisteerib palju erinevaid mänguliike ning seetõttu ei saa olla ühte kõige paremat tehisintellekti teostusviisi. Mõned agendi käitumise omadused on teatud mänguliikide puhul palju olulisemad kui teised ning probleemid mida tehisintellekt lahendama peab võivad väga palju erineda [4]. Seetõttu tuleb iga tehisintellekti lahenduse mõõtmisel lähtuda sellest, kuidas seda rakendatakse.

Strateegiliste mängude jaoks loodud tehisintellektid võivad kasutada *expert system* lahendust, mille puhul on kõikide olukordade jaoks, millesse agent võib sattuda, sisestatud täpsed juhised, mille järgi käituda. Selle meetodiga on võimalik teha nii, et agent tegutseb alati, nagu käituks osav inimmängija. Mängudes esinevate olukordade hulk võib aga olla väga suur ning paljudes mängudes ei kehti agendile samad reeglid, mis mängijale ning *expert system* ei ole hea valik. [5]

Seda kui hästi tehisintellekti lahendus mängu sobib, saab hinnata kasutades näiteks eelnevalt kirjeldatud kiirust ja mälu kasutatavust. Kui mängus on palju agente, mis peavad koheselt ja pidevalt mängija tegevusele reageerima, on kiiremad algoritmid paremad. Samas mängude jaoks, mille maailmaoleku moodustab suur kogus andmeid, on olulisem pigem mälu mõistlik kasutamine. Analoogselt kiirusele ja mälu kasutamisele on iga eripära sobimuse hindamisel oluline. Käesolevas töös on erinevate tehisintellekti meetodite headuse hindamisel arvesse võetud nende sobimust prototüübiga. [1]

### 1.2.1 Esilekerkiv käitumine

Esilekerkiv käitumine (*emergent behavior*) on üks nendest tehisintellekti aspektidest, mis paneb agendi mängu mõistes huvitavalt käituma. Esilekerkiv käitumine on märkimisväärne olukorras, kus on palju erinevaid määravaid faktoreid. Kui agenti ähvardab mingisugune oht ja ainuke võimalik vastus sellele on ründamine, pole see üllatav kui agent ohtu ründab. Kui aga mängumaailm on realiseeritud nii, et

- agendil on numbrilised väärtused, mis määravad ära kui julge või tugev ta on;
- ohuallikaid on mitu ja neil on erinevad tasemed;
- agent oskab ära joosta ja rünnata ning valida keda ta ründab;
- agendil on funktsioon, mis arvutab nende väärtuste põhjal ühe võimalikest tegevustest;

esilekerkiv käitumine ongi see mida agent teeb, siis mida mõistlikum see käitumine mängureeglite järgi on, seda põnevam on seda jälgida. [6]

*Emergent behavior'i* on kõige kasulikum rakendada mängudes, kus on palju väikeseid enam-vähem sarnaseid olukordi, mis mängija ja agendi vahel tekib, aga spetsiaalselt iga variatsiooni jaoks vastava koodi kirjutamise asemel oleks palju parem, kui agent kohaneks olukorra erinevustele ise ning teeks mõistlikke otsuseid dünaamiliselt ja mängulooja ei peaks absoluutselt iga olukorda ette nägema [7]. Samuti on esilekerkiv käitumine oluline mängudes, kus on väga suur rõhk tehisintellekti tegevuse jälgimisel ja suunamisel nagu näiteks Black & White [8] või *The Sims* [9, 10].

Agendi esilekerkiva käitumise mõistlikkus ja loogilisus sõltub sellest, kui hästi tehisintellekti reeglistik on tehtud.

### 1.2.2 Oskuslikkus

Mängudes on agendi osavuse tase väga oluline. Kui agent käitub mängija vastasena kõikides olukordades liiga targalt ja teeb alati objektiivselt parima võimaliku otsuse, ei ole mängijal enam lõbus. Seega peaks agent tegema vahel vigu. Samas kui viga on ebaloomulik, jätab agent masinliku ja tehisliku mulje ning see teeb mängule üldjuhul ainult kahju. Eelistatav on pigem selline tehisintellekt, mis on keskmisest targem, kuid mitte täiuslik. [11]

Kui tegu on mänguga, milles tehisintellekt peab oma käitumisega jäljendama inimmängijat, on jälgimisega võimalik kindlaks teha milliseid vigu erinevate tasemete mängijad teevad ning agentdil neid jäljendada lasta. Selliseid vigu sisestades peab olema väga täpne, sest kui agent teeb vea vales kohas või valel ajal, mõjub see mängus halvasti. Ka väga keerukas mängu tehisintellekt ei ole kaitstud tobedate vigade eest. [12]

Mängudes, kus agent ei tegutse päris samade reeglite järgi kui mängija, on loomulikkus ehk ühtlaselt keskmisest mõistlikum käitumine veelgi olulisem [11]. Agendi eesmärk ei ole sellisel juhul mitte mängu võita, vaid hoopis mängumaailma olekut enda jaoks paremaks muuta. Prototüüp toimib sellisel põhimõttel, et tarkus või rumalus lähtub pigem mitte mängija perspektiivist, vaid agendi perspektiivist mängumaailmas.

Mõnikord on mõistlikum reaalse võimekuse asemel tekitada tarkuse illusioon. Selle saavutamiseks antakse tehisintellektile palju rohkem informatsiooni kui mängija eeldab.

Näiteks *RTS* mängudes, kus mitteokupeeritud kohtadesse nägemine on mängija eest uduga varjatud, aga agent näeb kõike mis toimub, kuid teeskleb otsustades, et ei näe [13].

## 1.3 Implementatsiooni keerulisus

Iga erineva mängulooja implementatsioon samast tehisintellekti meetodist võib teiste omast tundmatuseni erineda [1].

Algoritmide implementatsioonide keerulisust saab hinnata objektiivselt selle järgi kui mitmest elemendist need koosnevad ja kui keeruline on põhimõte, mille järgi nad toimivad. Samas on ka oluline algoritmi implementeerimise keerulisuse subjektiivne pool ja kuidas erinevate oskustasemetega programmeerijate jaoks see keerulisus skalaarub. Seetõttu on käesoleva töö raames selle aspekti hindamisel arvesse võetud nii professionaalide kui ka autori isiklikku suutlikkust ja arvamust.

# 2. Liikumine

Mängu agendi üheks tegevustest mängumaailmas on liikumine. Olenevalt vajadusest võtab agent liikumisel arvesse erinevaid andmeid, näiteks kui agent liigub millegi poole, arvestab ta koordinaatidega, kus see objekt maailmaruumis on. Kui agendi ümbruses on ohte või takistusi, peaks ta nendest teadlik olema ning oma liikumist sellele vastavalt korrigeerima.

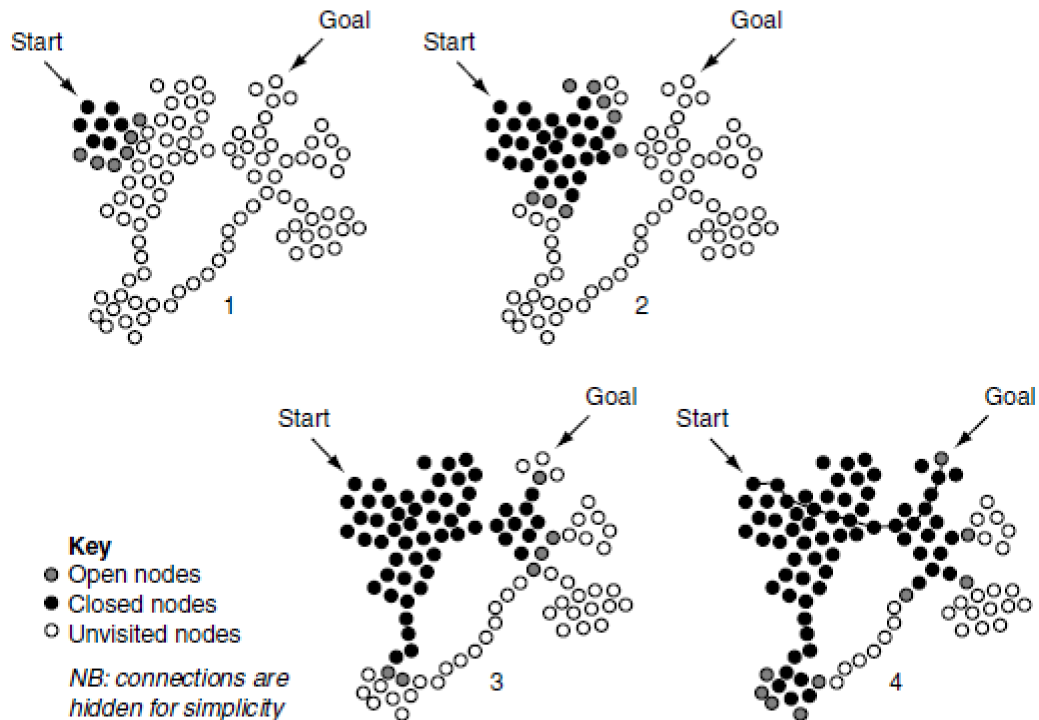
## 2.1 Dijkstra algoritm

Edsger Dijkstra [1, 14] algoritm leiab kõige lühema tee sidusas graafis algtipust kõikidesse teistesse tippudesse. Lühimad teed võtavad arvesse graafi tippude kaalutud pikkuseid. Algoritm loob iga tipu naabertippudega ühenduse ja talletab naabertippudesse ka senise maksumuse (*cost*), milleks on hetkel aktiivse tipuni jõudmise maksumusele liidetud naabertipu ühenduse maksumus. Algoritm töötab iteratiivselt ja esimese ning sellele järgnevate iteratsioonide vahel ei ole erinevust. Esimese tipu ehk alguse maksumus on 0. Itereeritava tipu valib Dijkstra sellise, millel oleks kõige väiksem senine maksumus.

Algoritm peab erinevate tippude suhtes järge loendite abil. Üks loend on nimega suletud *(closed)*, millesse lähevad kõik tipud, millega on algoritm juba iteratsiooni läbi viinud ja teine loend on avatud *(open)*, mis on olnud mõne teise tipu naabertipuks, aga millega algoritm pole veel omakorda iteratsiooni teinud. Kui mõne tipu naabertipuks osutub tipp, mis juba asub avatud loendis (suletud loendis olevat tippu ei ole võimalik teist teed pidi Dijkstral leida, sest suletud tipu puhul on kõik selle ühendused juba läbi vaadatud) ning millel on juba senine maksumus olemas, peab algoritm käituma sellevõrra teistmoodi, et kui uus senine maksumus on väiksem kui vana, asendatakse vana uuega. Vastasel juhul ei muutuks midagi.

Kõige geneerilisem Dijkstra algoritm lõpetab töö siis, kui *avatud* loend saab tühjaks. Seetõttu on tulemuseks alati teekond, mis on tõepoolest kõige lühem. Praktikas katkestatakse tihti algoritm kohe, kui esimene teekond otsitava lõpptipuni on leitud, sest järgnevad teekonnad on harva lühemad ja väiksema maksumusega teekonna leidmise korral ei ole võit esimese leitud teekonnaga võrreldes tavaliselt eriti suur.

Teekonna leidmiseks tuleb alustada eesmärgiks oleva tipuga ning liikuda mööda tippudesse talletatud ühendusi, mis viivad alguseni. Saadud nimekiri tippudest tuleb veel tagurpidi pöörata ning tulemuseks ongi täielik teekond.

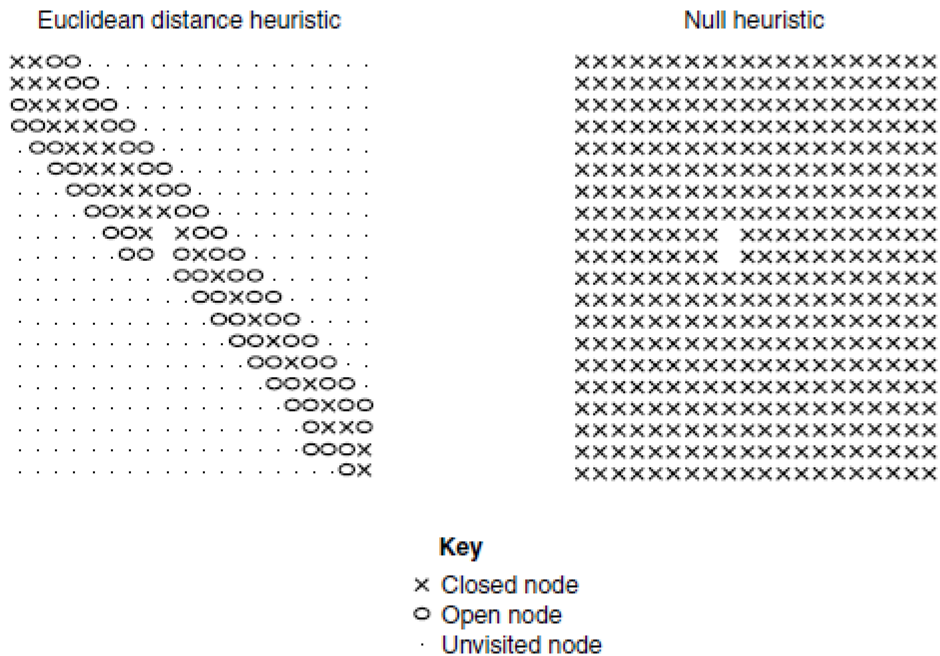



**Joonis 1 - Dijkstra algoritmi progressi järk-järguline kujutus [1]**

Algoritm leiaks mängus kasutust nii, et graafi tipud on punktid kuhu mängu agent saab liikuda. Juhul kui aga agent liigub ühest punktist teise, leiab algoritm talle küll õige teekonna, kuid agendil ei ole samas kasulik omada informatsiooni kõikide lühimate teede kohta. Algoritmi keerukus on tihedalt ühendatud graafi puhul $O(n^2)$ kui *n* on tippude arv graafis. Mängus liikuva agendi lühima tee leidmiseks kohandati algoritm optimaalsemaks ja tulemuseks on A* algoritm.

## 2.2 A* algoritm

A* (*A-star*) [1] on struktuuri poolest Dijkstra algoritmiga sarnane, aga kasutab järgmise itereeritava tipu valimiseks heuristikat. Seega iga tipu peal valib algoritm järgnevaks tipuks selle, mis viib heuristika järgi kõige tõenäolisemalt sihtkohani. Lisaks senisele maksumusele võtab A* algoritm arvesse ka heuristilist väärtust, milleks on ennustatav maksumus tipust eesmärgini. Selle väärtuse genereerib eraldi kood valitud meetodi abil ja A* algoritm seda ei tekita. Dijkstra on põhimõtteliselt A*, mille heuristiline väärtus on alati 0.




**Joonis 2 - A* algoritmi ühest nurgast teise teekonna leidmisel avatud ja suletud tipud heuristilise eukleidilise väärtusega (vasakul) ja 0 väärtusega (paremal) [1]**

Sellise tippude eelistamise tagajärjel itereerib algoritm alati eelisjärjekorras paljulubavaid tippe ning teeb tõenäoliselt alati vähem või kõige rohkem sama palju tsükleid kui Dijkstra. A* peab tegutsema erinevalt sellises olukorras, kus mõne tipu naabertipp on suletud nimekirjas, kuna suletud loendis olevatel tippudel ei ole kõiki naabertippe läbi vaadatud. Kui suletud tipuni leitud uus teekond on eelnevast parem, tuleb see tipp uuesti avatud tippude loendisse panna, et selle naapertipud saaksid ka uue hinnangu.

Olenevalt heuristilise väärtuse leidmise meetodist ja algoritmi peatamise tingimusest sõltub algoritmi potentsiaalne eelis. Algoritm võib leida näiteks väga tõenäolise lühima teekonna ilma kogu graafi läbi vaatamata või teha võimalikult vähe iteratsioone, aga siiski leida adekvaatse teekonna. Heuristilise väärtuse leidmine võib üldise keerukustaseme märksa kõrgemale viia, aga on üldjuhul siiski O(1) ning ei muuda A* keerukustaset. Heuristline väärtus peaks täpsemate A* algoritmide puhul tõenäolist maksumust eesmärgini alahindama, et iteratsiooni läbiks rohkem tippe ning kiiremate algoritmide puhul maksumust ülehindama.

Üheks võimalikuks heuristikaks on kahe punkti vaheline kaugus kahemõõtmelises ruumis (*euclidean distance)*. Selle kauguse arvutamisel ei võeta arvesse mängus olevaid takistusi. See heuristika on takistuste puudumisel täpne ja muudes olukordades alahindav. Ennustavaks väärtuseks võib olla ka näiteks klaster heuristika. Juhul kui agent liigub mängus siseruumides

ehk erinevad maailma osad on eraldatud seintega, paigutatakse graafi tipud ühte klastrisse kui nad on samas ruumis. Heuristilised väärtused on määratud ruumi ehk klastri kaupa ning on sellisel juhul iga maailma jaoks eelnevalt tabelisse käsitsi ära määratud, ehk eeltööd tuleb rohkem, aga algoritm peab ise tegema vähem tööd.

A* algoritmist on palju erinevaid variante. Iga erinev A* algoritmi tuletis püüab parandada või parendada mingisugust A* aspekti. Järgnevalt on kirjeldatud mõningad näited sellistest arendustest.

1. D* algoritm [15] toimib nii, et ta ei ole kogu maailmast teadlik ning oletab, et kõik mida ei ole näha on ilma takistuseta ning tegeleb seintega alles siis kui need on nähtaval. See algoritm leiab rakendust robootikas. Tänapäeval eelistatakse teistele D* versioonidele pigem D* Lite algoritmi. D* Lite on tehtud LPA* ehk Lifelong Planning A* põhjal. LPA* on inkrementaalne versioon A*'st ja on kasulik olukorras, kus maailmaruum on muutlik (näiteks seinad liiguvad) ning on vaja kiiresti leida uus teekond olukorras, kui algus ja lõpp on samad, kuid ülejäänud graafi otsad muutuvad.

2. IDA* ehk *iterative deepening A** [16] on tavalisest A*'st aeglasem, aga vajab vähem mälu. Algoritm ei jäta *open* ja *closed* tippe meelde ning käib lihtsalt kõik teekonnad lõpuni. Sellest on tuletatud Fringe, mis on kasutab põhimõtteid A*'st ja IDA*'st. Fringe käib teekonnad läbi sarnaselt nagu IDA*, kuid teekonna jooksul jätab meelde naabertipud, et nendest hiljem samasuguseid otsinguid sooritada.

3. JPS ehk *jump point search* [17] algoritm „hüppab" rekursiivselt tippudest üle ning jätab kõik vahepealsed tipud avamata ja läbi töötamata. See on paljudes olukordades üle kümne korra kiirem kui A*. JPS toimimiseks peab maailmaruum olema *grid* ja graafi tippude maksumused peavad olema binaarsed (kas on takistus või mitte).

Need demonstreerivad selliste algoritmide üldist arengusuunda ja kuidas mõned muudatused teevad algoritmi objektiivselt kiiremaks ja tulemused optimaalsemaks, aga teised parendused tulevad muude valdkondade arvelt. Erinevate rakenduste jaoks on erisugused algoritmi omadused olulisemad ning on aksepteeritav ohverdada suutlikkust ühes valdkonnas, et parandada seda teises.

Mängude jaoks on olulisi faktoreid veel, mida on raskem mõõta. A* algoritmi üks põhilisi probleeme on selle n-ö masinlikkus. Tulemuseks saadav teekond koosneb suundadest. Kui

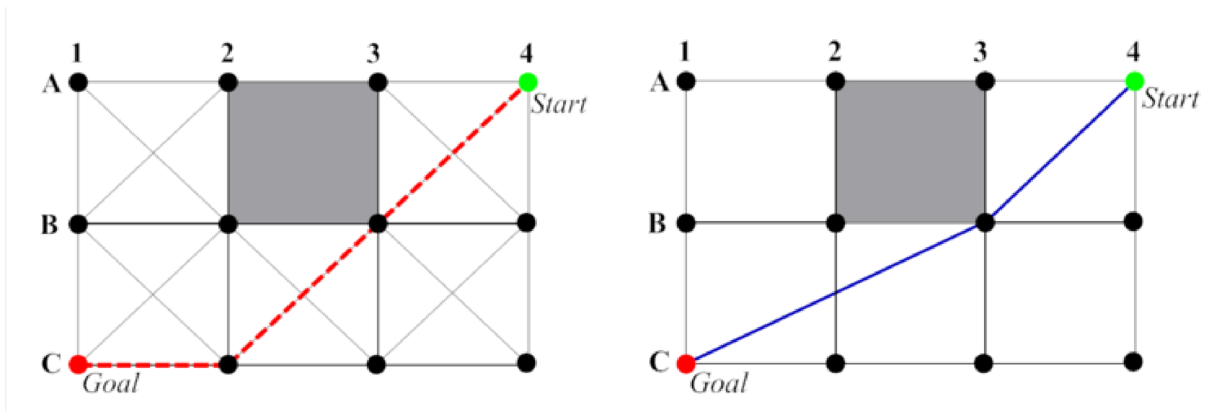
maailm koosneb kindlatest radadest või ruudustikust, jälgib tegelane masinlikult kas radu või ruudustiku külgi ja diagonaale. Selle vältimiseks on tuletatud erinevaid lahendusi. [1]

## *2.3 Any angle* algoritmid

Et saavutada mängus sirgemaid ja loomulikumaid teekondi, on üheks lahenduseks algoritmid, mis liiguvad igas suunas, mitte ainult diagonaalis, horisontaalis ja vertikaalis.

*Any angle* algoritmid leiavad pikemate sirgete radadega teekondi ja seetõttu võivad igasuunalised algoritmid leida mängukeskkonnas optimaalsemaid teekondi kui A* algoritmid [18]. A* algoritmid garanteerivad paljudel juhtudel lühima teekonna graafil, kuid see võib erineda tegelikest lühimatest teekondadest mängumaailmas.

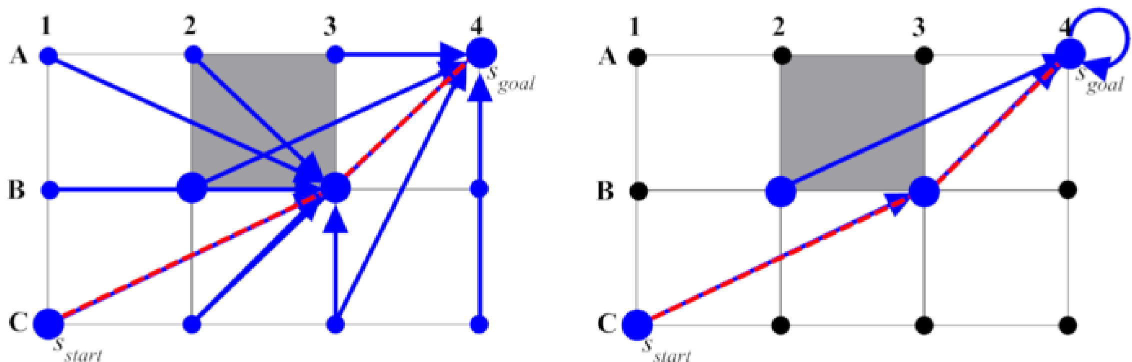
Käsitleme esimesena algoritmi Theta* [19], sest see on *AA* algoritmide seas kõige analoogsem eelnevalt käsitletud A* algoritmiga. Üldiselt A* algoritmide puhul peab iga tipu vanemaks (*parent*) olema tipp, mis on nende vahetus läheduses. Theta* puhul võib vanemaks olla ükskõik milline tipp. Igat naabertippu uurides vaatab algoritm ka seda, kas tipu vanemaga on võimalik tekitada otsene joon ilma, et mõni takistus ette jääks. Selle tagajärjel tekib teekond, mille pöördekohad on takistuste nurkades. Sarnane teekond tekib ka siis, kui A* algoritm läbib *smoothing* protsessi.




**Joonis 3 - A* algoritmi (vasakul) ja *any-angle* algoritmi (paremal) leitud teekonnad [20]**

Sellest algoritmist on tuletatud Lazy Theta* [21]. Tavalise Theta* puhul teeb algoritm väga palju *line of sight* kontrolle ning paljud neist on tegelikult mittevajalikud. Tavaline Theta* kontrollib iga naabertipu *line of sight'i*, aga Lazy Theta* oletab tippude avamisel, et hetkel avatud tipu naabertipp on selle tipu vanema vaateväljas. Alles siis teeb algoritm vaatevälja kontrolli ja *line of sight* puudub, määrab ta naabertipu vanemaks avatud tipu, aga järgmise

sammuna algoritm avab juba järgmise tipu. Tulemusena teeb Lazy Theta* märkimisväärselt vähem vaateväljakontrolle.




**Joonis 4 - Theta* (vasakul) ja Lazy Theta* (paremal) leitud teekonnad [20]**

Nash ja Koenig on püüdnud Lazy Theta*'t optimeerida [21]. Näiteks Lazy Theta* -R jätab naabertipu avades hetkelise tipu avatud nimekirja, et algoritm saaks selle sama tipuni algusest hiljem lühemaid teid leida. Samuti on olemas pessimistliku oletusega Lazy Theta* -P, mis teeb iga sammuga eelduse, et naabertipul puudub vaateväli hetkelise tipu vanemani. Sellega säilitab algoritm omaduse, et iga avatud loendis olev tipp on kindlasti oma vanema vaateväljas. Selle oletusega saab algoritmi edasi arendada Incremental Phi* algoritmiks, mis on põhimõtteliselt eelnevalt nimetatud A* variatsiooni D* *any angle* vaste.

Lazy Theta* algoritmi heuristika on võimalik sättida nii, et alguses oleks algoritmi jaoks olulisem, et ta liigub mingis suunas ja ei uuri paljusid ümbritsevaid tippe ning otsingu lõpus on heuristika vähem oluline ja lõpu leidmine tähtsam. [22]

$$h(s) = w * c(s, s_{goal})$$

See variant avab üldjuhul kõige vähem tippe ning teeb seega kõige vähem vaatevälja kontrolle [22].

Tavalise Theta* algoritmiga võrreldes võib A* algoritm koos hilisema *smoothing'uga* olla kuni kaks korda kiirem, aga Lazy Theta* ja optimeeritud Lazy Theta* on paljudes olukordades kiiremad kui A*. Kusjuures optimeeritud Lazy Theta* on kiirem, aga võrdeliselt suurem on ka risk, et leitud teekond on pikem [21].

Siinkohal on võrdluses Theta* algoritmid teiste igasuunaliste algoritmidega. Põhimõtted on neil erinevad. Oluline on ära määrata, mis nende mõõdetavate omaduste kasulikkus reaalsel rakendamisel on ning mida tuleks käesoleva töö prototüübi puhul arvestada.

Tansel Uras ja Sven Koenig [18] võrdlesid erinevaid igasuunalisi algoritme ja A* algoritmi. Implementatsioonid olid sarnase põhjaga ja seetõttu on tulemused kasulikud. *Any angle* algoritme on varem ka võrreldud, aga ilma eksperimentaalsete tulemusteta või vähesemate algoritmide vahel. Kõikide erinevate tööde tulemusi omavahel võrrelda on ka keeruline, sest erinevad teaduslikud tööd kasutavad erisuguseid implementatsioone ning katsetingimusi [18]. Tulemustest on näha kuidas algoritmid reaalse rakenduse korral töötaksid.

Kirjeldamata on veel mõned algoritmid mida võrreldi töös [18].

1. Block A* [23] jaotab *grid'i* ühesuurusteks eelnevatest suuremateks ruutudeks ning teostab graafi läbimist nende ruutude kaudu, mitte individuaalsete tippude kaudu. Antud katses on kasutusel 5x5 suurused ruudud, ehk iga ruut koosneb 25 graafi tipust.

2. Field D* [24] on modifitseeritud versioon eelnevalt mainitud D* Lite'st. Tippe avab see algoritm nagu A* ning otsing toimub õigetpidi, mitte tagurpidi nagu D* Lite puhul.

3. ANYA [17] avab hoopis sirgjooni võrgustikul asuvate nurkadeni. Igal olekul on rea number ja nurk millel nad asuvad. Toimib sarnaselt nagu JPS.

4. Sub-2 [25] kombineerib erinevad graafi tipud võrgustikus olevate takistuste järgi. Põhimõtteliselt püüab Sub-2 luua takistuste vahele loogilised ruumid ehk alameesmärgid. Teekonna arvutab algoritm seejärel erinevate leitud ruumide vahel.

Võrdluses jooksutati algoritme erinevate kaartide peal. Osad kaardid on moodustatud eksisteerivate mängumaailmade järgi ja mõned on juhuslikult genereeritud. Mängukaartidel on võrdluse järgi kõige kiirem Sub-2 (17.9 korda kiirem kui A*-Oct algoritm, mis kasutab oktiilse kauguse heuristikat). Teiseks kõige kiirem on Block A* (2.52 korda kiirem kui A*-Oct). ANYA kiirus on ebastabiilne ja on ühel kaardil kõige kiirem ning enamikel kõige aeglasem.

Mida keerulisem on kaart, seda pikemaks Lazy Theta* leitavad teed lähevad. Kõige halvemas olukorras on L-Theta* teekond 0.6% optimaalsest teest pikem. Teine algoritm mille pikkused on keerulisematel kaartidel ebaoptimaalsed on Sub-2. A*-Euc ja A*-Oct algoritmidel on tee

pikkus keskmiselt 4-5% optimaalsest pikem. Block A* ja Field A* leiavad praktiliselt igas olukorras pisut lühema teekonna kui Theta*.

Pöördekohti on tavalistel kaartidel kõige vähem Block A*'l ja juhuslikult genereeritud maailmades tavalistel A* algoritmidel. Lazy Theta*'l on enne *smoothing'ut* rohkem pöördekohti kui Theta*'l, aga pärast *smoothing'ut* vähem. See iseloomustab neid algoritme hästi. Pöörete arv ei ole seotud teekonna pikkusega, vaid pigem iseloomustab milliseid teekondi algoritm eelistab.

Vaadeldes teiste katsete [26, 23] tulemusi, kus olid mõned implementatsiooni erinevused. Uras'i ja Nash'i Block A* oli 1.17 korda aeglasem ning Theta* 2.84 korda aeglasem kui Uras'i ja Koenig'i implementatsioonid [18]. Sellegi poolest kinnitavad ka need katsetulemused, et Block A* on Theta*'st igas olukorras kiirem.

Uras ja Koenig toovad veel välja, et sõltumata mõõtetulemustest on algoritmidel mõned omadused, mis võivad teatud rakenduste jaoks väga olulised olla. Theta* algoritmi on võrreldes teistega kõige lihtsam implementeerida (peale A* algoritmi) ning seda on võimalik rakendada muudes 2D ja 3D keskkondades, mis ei ole *grid*. Field A* saab kasutada inkrementaalseks teekonna leidmiseks. Sub-2 on mängukaartidel palju kiirem kui teised algoritmid, aga Sub-2 vajab erinevalt teistele algoritmidele eelprotsesseerimiseks võrdlemisi pikka aega (kuni 35 sekundit).

Selgub, et igal algoritmil on omad tugevused ja seega ka põhjused, et neid kasutada. Theta* ja Lazy Theta* on lihtsa implementatsiooni ja universaalsuse tõttu paljudes olukordades väga hea valik, vaatamata sellele, et need algoritmid jäid teistele paljudes olukordades alla. Pika eelprotsessi ajaga Sub-2 algoritm on väärtuslik väga paljude mängijatega suurel alal. Sellises olukorras avab Sub-2 kasutamine väga palju ressursse muude protsesside jooksutamiseks. Antud töö raames on kasutusel Block A* algoritm, sest selle toimimine on iga aspekti suhtes teistega võrreldes tugev ning selle omadused on prototüübiga igati sobilikud.

# 3. Nägemine

Selleks et tehisintellektil oleks võimalik otsuseid langetada ja nende alusel tegutseda, on agendil vaja andmeid. Selleks peab otsustama, mis sisendit kaudu agent andmeid saab. Üks variant on kõik teadmised agendi käivitamisel teatavaks teha [13]. Teiseks variandiks on jäljendada meeli nagu näiteks nägemine ja kuulmine.

Nägemiseks mõeldud algoritme kasutatakse olukorras, kus on vaja määrata, mida tegelane mängualast näeb. See sobib olukorras, kus on vaja selgeks teha, kus on varjud ja milliseid alasid tegelane igal ajahetkel näeb ning milliseid ei näe. Käesolevas töös on käsitletud graafipõhiseid nägemisalgoritme, mis töötavad sarnase kaardiga, mida kasutavad *pathfinding* algoritmid [27]. Järgnevalt on lühidalt kirjeldatud mõned nägemisalgoritmid.

1. *Raycasting* [28, 29] kasutab põhjaks *line of sight* algoritmi, mida kasutavad Theta* algoritmid ja *smoothing* algoritmid. Seda on lihtne implementeerida. Põhimõtteliselt joonistab algoritm vaataja asukohast välja mitu kiirt kuni seinteni, kasutades Bresenham'i joone algoritmi [30].

2. *Shadowcasting* [29, 31] leiab takistused ning arvutab nendest takistustest edasi varjud, mis peaksid takistuste taha langema. See algoritm on keskmise implementeerimise keerukusega.

3. *Diamond* [29] kasutab konkreetselt *line of sight* algoritmi erinevates võimalikes suundades. Algoritm kohtleb ruute kui rombe. See on põhimõttelt tavalise *raycastingu* sarnane, aga joone joonistamiseks kasutataksegi *line of sight'i*, mitte Bresenham'i joone algoritmi [30].

4. *Permissive* algoritm [29, 32] defineerib nähtavust natuke lõdvemalt kui enamik algoritme. Üks ruut on teisele nähtav juhul, kui ükskõik milline punkt ruudust on nähtav teise ruudu ükskõik mis punkti jaoks. See on üks raskemaid algoritme, mida implementeerida.

5. *Digital* [29, 33] loob nägemisala digitaalsete joonte abil. Nähtava ala loomiseks käib algoritm kõik digitaalsed jooned läbi, kuni jõuab takistuseni. Ruutude nähtavus on analoogne *permissive* algoritmiga ning ruute kohtleb *digital* samuti rombidena nagu *diamond* algoritm.

Neid algoritme kompareerivas uuringus [27] võrreldi nende praktilist väärtust mängus ja kiirust välis- ja sisekaartidel. Tulemusena leiti, et digitaalne algoritm on palju aeglasem kui teised. Kõige tavalisem *raycasting* on suurte väliskaartide puhul kiireim. Teiste olukordade jaoks on kõige kiirem *shadowcasting*.

Mängudes kasutamiseks võrreldi erinevates olukordades algoritmide moodustatud varje ning sümmeetriat. Varjude puhul on oluline reaalsus ning sümmeetrilisuse määrab ära see, kas ühe agendi poolt nähtavate ruutude peal kõik teised agendid ka teda omakorda vastu näevad. *Permissive* algoritm lubavuse astmel 8 ja digitaalne algoritm on loomult sümmeetrilised algoritmid ja nad on tulemustes samuti ilma vigadeta sümmeetrilised. Mida väiksem on *permissive* algoritmi lubavuse aste, seda vähem sümmeetriline see on. Muude algoritmide sümmeetrilisuse protsentuaalne viga on siseruumides sarnane ja üldiselt ei tohiks probleem olla, aga välisruumides on viga liiga suur ning võib tekitada mängus ebaausaid olukordi.

Osade (eriti sümmeetriliste) algoritmide loodud varjud on väga ebaloomulikud või ebamõistlikud. Kõik teised algoritmid loovad enamikes olukordadest mängu suhtes loogilisi varje, aga mitte ükski neist ei ole täiuslik ja igal algoritmil on vähemalt üks natuke vigane olukord. Nagu ka katse tegija täheldab, tuleb lahenduse valikul mängu konkreetset iseloomu ja vajadusi silmas pidada. Käesoleva töö prototüübi loomisel on kasutusel *shadowcasting* algoritm, sest see töötab kiiresti, seda on lihtne vajaduse järgi kohandada ning olukordi ebaloomulike vigade tekkimiseks prototüübis ei esine.

# 4. Otsuste tegemine

Otsuste tegemise puhul muudab agent sisendi, ehk selle mis mängumaailmas toimub väljundiks, ehk kuidas peaks sellises mängu situatsioonis käituma.

Käesoleva töö prototüübi olukorrad on põhimõtteliselt simulatsioonid. Enamik *decision-making* lahendusi toimivad oma tavalisel kujul kõige paremini lihtsates olukordades, kus on vaja täita otsekohest mängukäiku edendavat rolli [1]. Sellest tulenevalt populaarseimad ja tunnustatuimad üldkasutuses olevad *decision-making* lahendused käesoleva töö prototüübi jaoks ei toimi. Selgus, et kui agendi tehisintellekt ei käitu päris nii nagu eksisteerivates populaarsemates mängudes ning kui mängu olemusest tulenevat piirangut agendi käitumise suhtes ei teki, tundub parim lahendus uue otsuste tegija loomine, mille struktuur oleks kohandatud rakenduse järgi.

Siit edasi on formuleeritud käesoleva töö strateegiamängu agendi tehisintellekti olemus, sellest tulenev struktuurivajadus ja probleemsed kohad.

## 4.1 Strateegiamängu agendi struktuur

Käesoleva töö strateegiamängu agent simuleerib elu, kus ta täidab oma vajadusi, tegeleb ohtudega ja püüab paremaks saada. Need probleemid peegeldavad mitmeid muid mängutüüpe [1]. Seega nende probleemide lahendamine võib pakkuda üldiseid lahendusi erinevate mängutüüpide jaoks.

Mudel, mis tegeleb ohuga, vajaduste rahuldamisega ja parendamisega on näiteks Maslow püramiid [34]. Seda on mõned korrad tehisintellekti formuleerimisel kasutatud [35, 36]. Seda põhimõtet kasutab ka tuntud agendipõhisele tehisintellektile toetuv mäng *The Sims* [10]. Tavaliselt lahendavad mängudes olevad agendid ühte või kahte püramiidiastet [1]. Mida enamate püramiidiastmetega agent tegeleks, seda huvitavam on agenti jälgida.

Agendi vajaduste väljendamiseks on vaja igale agendile mingisuguseid väärtusi, mis kirjeldavad nende olukorda maailmas. Need omadused võimaldaksid erinevatel agentidel teha samades sitatsioonides erisuguseid otsuseid. Sellest tulenevalt tekiks ka esilekerkiv käitumine ja oleks võimalik simuleerida loomulikku rumalust või tarkust. [9]

Agendil on tarvis nende vajaduste ja sisenditega tegeleda. Otsuseid langetav funktsioon peaks tegema kindlaks milline lahendatav probleem on olulisem kui teised ja samuti millised objektid

on eesmärkidega seoses kõige kasulikumad/kahjulikumad. Kuna agent peab sisendile ja vajadustele reageerima koheselt, on kiirus oluline. Samas on vaja teatud ulatuses tegevusi planeerida, et kohene reaktsioon oleks mõistlik ja tulevikuga arvestav. [37]

Käesoleva töö autor tuvastas lisaks mõned keerukamad situatsioonid, mis võivad sellisel agendil tõenäoliselt juhtuda. Selle asemel, et iga sellise kitsaskoha jaoks erandit luua, võiks see tehisintellekt oma struktuurilt neid esilekerkiva käitumise abil lahendada. Järgnevalt on esitatud situatsioonikirjeldused, mis aitavad moodustada käesoleva töö prototüübi kohandatud otsuste langetamise funktsiooni struktuuri.

## 4.2 Probleemsed situatsioonid

1. Agendil on janu ja nälja tekkimise võimalus. Nälg suureneb kiiremini kui janu. Agendil tekib janu ja ta otsustab sellega tegeleda. Lähim ja parim joogiallikas on aja poolest kulukas ning nälg, mis veel kriitiline ei ole, läheks selle joogiallika kasutamisel kriitiliseks. Kõrval on aga toiduallikas, mille tarbimine ei võta kaua aega. Agendil peaks olema piisavalt ettenägelikkust, et otsustada enne januga tegelemist toidu tarbimise kasuks.

2. Agendil tekib nälg. Parima ja lähima toiduallika ja agendi vahel on ohuolukord. Oht on liiga suur, et sellega tegeleda ja agent otsustab selle eest põgeneda. Kui ta on edukalt ohust eemale saanud, on see toiduallikas endiselt parim variant. Et agent ei jääks edasi-tagasi jooksma, peaks tal olema oskus parima toiduallika leidmisel ohtu arvesse võtta ja alternatiivse toiduallika puudumisel teha püsiv otsus, kas pigem riskida ohuga, jääda nälgima või minna otsima muid toiduallikaid.

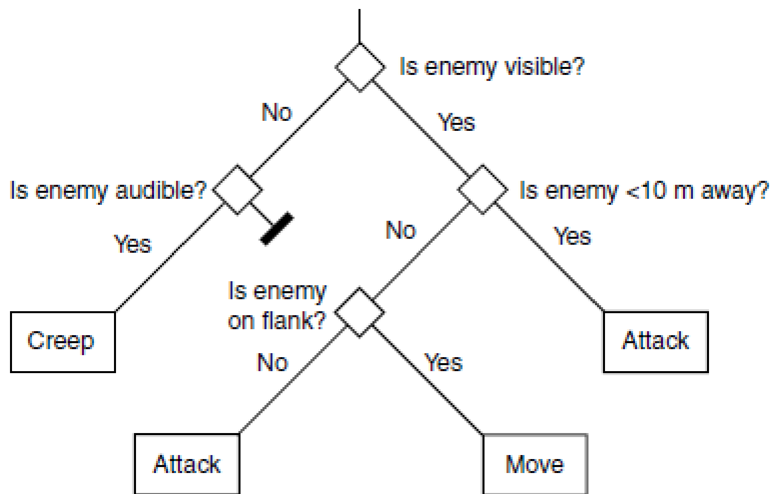
3. Kahe grupi vahel on sõjaolukord, ohte on palju. Agent on võimeline kaugelt ja lähedalt ründama ning põgenema. Maailmas on ka kohad, kus saab varjuda vastaste kaugete rünnakute eest. Agent peab otsustama millise ohuga ta peaks tegelema ja kuidas. Varjumise puhul, millist varjumiskohta ta kasutab. Agent peaks olema ka võimeline põgenema kui olukord on piisavalt ebasoodne.

Nende situatsioonide põhimõte on selles, et agent suudaks vajaduste loomisel ja lahendamisel arvesse võtta kõikvõimalikke tema maailmas esinevaid erinevaid ohte ja vajadusi ilma, et iga erisuguse olukorra jaoks oleks vaja erandit luua. Järgnevalt uurime erinevaid tehisintellekti otsuste langetamise põhimõtteid.

## 4.3 Levinuimad agendi otsuste langetamise põhimõtted

### 4.3.1 Otsusepuu

Otsusepuud on kiired, neid on lihtne implementeerida ja neist on kerge aru saada. See meetod on modulaarne ja dünaamiline ning lahendab suure hulga sisendeid mingisuguseks kindlaks tegevuseks. [1]

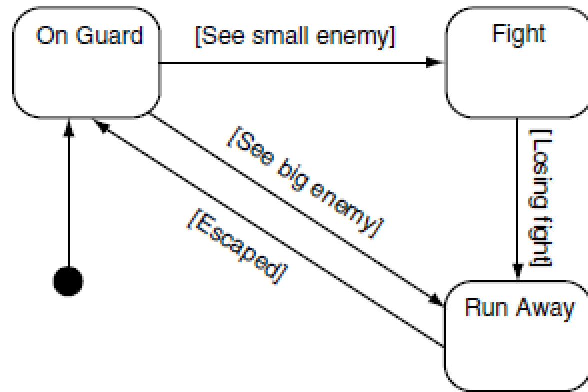



**Joonis 5 - Lihtne otsusepuu [1]**

See lahendus ei kasuta mälu ja kiiruse suutlikkus on $O(\log_2 n)$, kus $n$ on otsuste tippude hulk kui otsuste puu on tasakaalustatud. Selle lahenduse üks põhiline probleem on selles, et loodud otsused on vägagi ennustatavad. Millington ja Funge [1] pakuvad välja lahenduseks mõningad tipud ehk *juured* teha juhuslikuks. Teine probleem on see, et selle meetodi modulaarsus ja dünaamilisus tuleneb selle mitteelegantsusest. Otsusepuu on põhimõtteliselt üks suur *if else* funktsioon. See puu läheb iga uue maailmaoleku ja eseme lisamisel märkimisväärselt palju suuremaks. Käesoleva prototüübi puhul läheks probleemide lahendamine üleliia konkreetseks ja laiaks.

### 4.3.2 Olekumasin

Mõnes mängus on agendi käitumine selline, et ta sooritab mingit kindlat tegevust nii kaua kuni mingi muu sisend põhjustab teistmoodi käitumise. Sellist käitumisviisi saab tekitada otsusepuudega, aga lihtsam on seda saavutada olekumasinaga [1, 38]. Erinevad olekud on omavahel ühendatud ja ühelt olekult on üleminek võimalik ainult sellistele olekutele, mis on sellega seotud. Tihtipeale kutsutakse neid lahendusi *finite state machine'deks*.




**Joonis 6 - Lihtne olekumasina UML diagramm [1]**

UML diagrammidega on võimalik olekuid kirjeldada [39]. Otsusepuu puhul sooritab agent kindlate sisendite puhul alati samu tegevusi või juhuslikke valikuid. Olekumasinas on väga oluline iga sisendi puhul see, mis olekus agent hetkel on ja see võib omakorda agendi käitumist muuta.

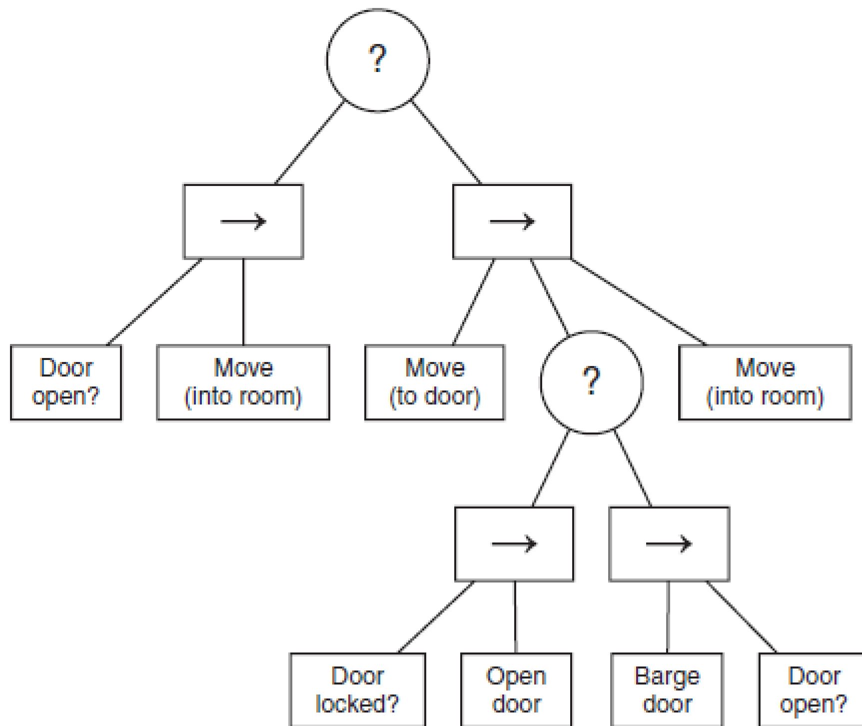
See meetod väärib mainimist, sest see on väga populaarne, aga jääks käesoleva töö raames liiga lihtsaks ning ei paku põhiliste probleemide jaoks lahendusi [1]. Kuigi Maslow püramiidi astmeid saab võtta olekutena ja sellest põhinevalt tekitada üldiseid olekuid, kus tegelane ülemiste astmetega ei tegeleks, oleks siiski avatumat süsteemi vaja, et mõistlikumalt ohtude ja vajaduste lahendamisega toime tulla. [1]

On olemas ka hierarhiline olekumasin, mis on laiendus tavalisele olekumasinale, et agent oleks võimeline tegelema alarmidega. See tähendab seda, et see lahendus koosneks ühe suure olekumasina asemel mitmest erineva osatähtsusega väikesest olekumasinast. Maslow püramiidi astmete kirjeldamiseks sobivad erinevad hierarhilised olekumasinad iseenesest paremini, kuid põhiline probleem ongi selles, et ühe vajaduse tähtsustase peab teisest olema kas rangelt kõrgem või madalam. Seega agent oma tegevuses mitme olekumasinaga ei arvesta, isegi kui see oleks mõistliku otsuse tegemiseks vajalik.

### 4.3.3 Käitumispuu

Käitumispuu ehk *behavior tree* on segu erinevatest tehisintellekti tehnikatest nagu hierarhiline olekumasin, planeerimine ja tegevuste käivitamine. Selle lahenduse tugevuseks on see, et sellega saab erinevate probleemide lahendamist teha selgelt ja konkreetselt nii, et ka mitteprogrammeerijad seda mõistaks. Isegi kui käitumispuu elemendid paistavad sobivat käesoleva töö prototüübile, on selle lahenduse struktuur ebapraktiline sellepärast, et ta on

loodud mitteprogrammeerijatele mõistetavaks. Üks põhiline element, mida käesoleva töö prototüüp peaks sooritama, on sündmustele ja sisenditele reageerimine. Käitumispuuga selle realiseerimine on võrreldes teiste lahendustega tülikas ja keeruline. [1]

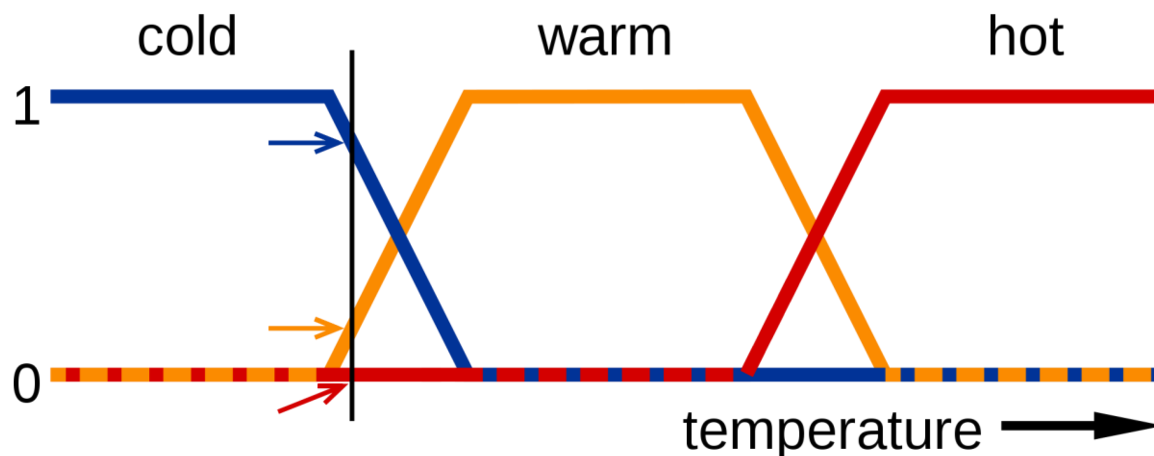



**Joonis 7 - Keerulisem käitumispuu [1]**

Vaatamata sellele kasutab käitumispuu väga efektiivset lahendust tegevuste liigitamise jaoks. Kasutusel on mõisted - tavaline tegevus, valikuline tegevus ja liittegevus. Selline lahendus pakuks käesoleva prototüübi tegevuste praktiliseks loomiseks modulaarse lahenduse.

### 4.3.4 Määramatus

Agendi otsustamise protsess ei saa olla keerulistes olukordades liiga konkreetne ja binaarne. *Fuzzy logic* tegeleb just otsustamise ebakindlusega [1]. Näiteks kui agendil on korraga päevakorras nii ohud kui ka vajadused, on vaja teha kindlaks mis on kõige olulisem ja millisel määral. Selline lähenemine lisab agendi käitumisele rohkem variatsioone, esilekerkivat käitumist ja mitmekülgsust. *Fuzzy Logic* on üldiselt väga populaarne, aga on teaduslikult diskrediteeritud. Põhjuseks see, et ebakindluse teooriaid (v.a tõenäosusteooria [1]) on võimalik liiga lihtsalt ekspluateerida. [40]




**Joonis 8 - Kolme *fuzzy logic* väärtuse tõusmine ja langemine temperatuuri muutudes [41]**

*Fuzzy logic* sai populaarseks seetõttu, et tõenäosusteooriat oli liiga raske mängudesse praktiliselt rakendada, enam see probleemiks ei ole [1]. Kuna prototüübi loomulikkuse tarbeks on agendil vaja palju erinevaid numbrilisi väärtusi, oleks tõenäosusteooriat lihtsamini rakendav lahendus vajalik.

### 4.3.5 Eesmärgipõhine käitumine

Eesmärgipõhine käitumine ehk *goal oriented behavior* erineb eelnevatest lahendustest küllaltki palju. See lahendus sobib mängudesse, kus on mingil määral agentide simulatsiooni ning agentidel on vajadused ja olek, mille parendamiseks nad püüavad sobivaid tegevusi leida. Nagu eelnevalt mainitud, viiks kõikide nende tegevuste kooslus meeletult suure otsustuspuuni. Eesmärgipõhine otsustamine pakub aga agendile kindla hulga tegevusi ja agent valib nendest sellise, mis tema olekut kõige rohkem parendab [1].

Iga tegevus muudab agendi olekut mingil moel. See muutus võib olla kas negatiivne, positiivne või mõlemat. Tegevust valides võtab ta kõiki neid faktoreid ning oma olukorda arvesse ja teeb arvutustele vastavalt parima valiku. See lahendus on lihtne, kiire ja toimib mõistlikult, eriti kui võimalikke tegevusi ega eesmärke ei ole agendil palju. Probleem on selles, et see lahendus ei suuda tegevuse kõrvalnähte arvesse võtta ega tulevikku ette näha. [1]

Goal: Eat = 4 Goal: Bathroom = 3
Action: Drink-Soda (Eat – 2; Bathroom + 3)
Action: Visit-Bathroom (Bathroom – 4)

**Joonis 9 - Agendi võimalikud eesmärgid ning tegevused eesmärgipõhise käitumisega lahenduses. Võib juhtuda, et agent otsustab juua, kuigi see tekitaks kriitilise olukorra [1]**

Kiiruse keerukus sellel lahendusel O(n + m), kus *n* on eesmärkide kogus ja *m* on võimalike tegevuste kogus. Mälu keerukus on O(1). Kui aga lisada kõrvalnähte ja tegevuste tagajärgi arvesse võtta, on kiiruse keerukus hoopis O(nm). Suuremate eesmärkide/tegevuste koguse korral võib see algoritm väga aeglane olla, sest iga eesmärgi ja tegevuse omavaheline suhe on vaja läbida.

Sellest edasi on arendatud lahendus, mis ei võta arvesse ainult ühte järgmist tegevust, vaid püüab leida näiteks nelja järgnevat tegevust. Selline algoritm võtab erinevad tegevused ja eesmärgid, hindab nende olulisust ja millal need tuleks täita. Sellisel põhimõttel toimib Goal Oriented Action Planning ehk GOAP [37, 42]. See lahendus omab sarnasusi käesoleva töö prototüübi probleemidega, aga see lahendus on olemuselt piirav.

Parima tegevuse leidmiseks otsib see algoritm läbi kõik olemasolevad tegevused, mis on mitmekihilised. See lahendus teeb parimate tegevuste leidmise analoogseks *depth-first* graafi lahendamisega. Mälu keerukus on sellel lahendusel O(k) ja kiiruse keerukus on $O(nm^k)$, kus *k* on maksimaalne sügavus, n on eesmärkide kogus ja *m* on saadaval olevate valitavate tegevuste koguste keskmine. Teatud heuristiliste lahendustega saab mõnes olukorras seda algoritmi kiirendada [1].

Üldiselt seda algoritmi on lihtne implementeerida, aga see ei ole elegentne [1]. Erinevad parandused ja heuristikad võivad seda küll mõneti paremaks teha, aga lõpuks on ikkagi *depth-first* otsing kõige optimaalsem variant. Kui tekib liiga palju tegevusi ja eesmärke, muutub see lahendus väga aeglaseks ja on raskesti hallatav. Seda lahendust on kasutatud mängus F.E.A.R [43].

Seoses järeldusega, et iga mängu jaoks sobiks põhimõtteliselt erinevalt kohandatud otsuse-tegemise printsiip ja lahendus [1, 4], ongi käesoleva töö prototüübi agendi tehisintellektile kohandatud sobiv otsuse langetamise lahendus.

## 4.4 Kohandatud otsuste langetamise algoritm

Agendil on mingisugune kogum eesmärke. Eesmärgid vormib ta vastavalt oma olukorrale ja välistele mõjutustele. Kui mõni agendi väärtustest saavutab kriitilise olukorra või agent satub ohuolukorda, annab see agendile märku, et tuleb eesmärke uuendada. Kriitilise olukorra tekkimisel vaatab agent kogu oma olukorra ja kõik ohud üle, teeb vajaduse korral uued eesmärgid ning määrab neile olulisustaseme. Eesmärke tekitav funktsioon jälgib alarme sarnaselt nagu hierarhiline olekumasin.

Eesmärkidest ja agendi mälus olevatest objektidest moodustub jada tegevusi, mis võtavad arvesse kõiki hetkel aktiivseid eesmärke. Agent vaatab enne uue tegevuse alustamist, kas selle tegemine tekitaks kriitilise olukorra, millega ta veel ei tegele. Seejärel vana tegevuste kogum kustub ja moodustub täiesti uus tegevuste jada, mis arvestab kõiki eesmärke.

See lahendus seab eesmärgid tähtsusjärjekorda sarnaselt *goal oriented action planner'iga, s*ee algoritm teeb seda aga teisel põhimõttel. GOAP otsib eesmärkide järgi tegevuste kombinatsiooni [37] ning on olemuselt otsingualgoritm, aga käesoleva töö kohandatud *decision-making* lahendusel on iga eesmärgiga juba mingisugused tegevused seotud. Oluline on see, et eesmärkide ja objektide sortimine oleks mõistlik. Sellest täpsemalt implementatsiooni peatükis.

# 5. Prototüübi implementatsioon

## 5.1 Üldine

C++ on üks populaarsemaid programmeerimiskeeli  mängude tegemiseks [1, 44]. C++ on kiire ja dünaamiline ning võimaldab teha kiiremaid mängumootoreid. Sellel on pikk ajalugu ja palju veebiressursse. C++ on samas keeruline ja selle kasutamine mängude tegemisel nõuab aega [44]. Käesoleva töö autor soovis C++ tundma õppida just nendel põhjustel ning seetõttu on see ka selle töö prototüübi programmeerimiskeeleks.

Prototüübiks sobib kõige paremini kahedimensiooniline pealtvaates formaat, sest see on kergesti valmistatav rakendus seda sorti tehisintellekti jaoks. Põhimõtteliselt ükskõik millise strateegiamängu puhul toimib liikumine, kauguse mõõtmine ja otsustamine ikkagi kahemõõtmelise pinna põhjal [1]. Kuigi käesoleva prototüübi jaoks ei ole kulutatud lisaaega, et kolmedimensioonilist mänguruumi tekitada, ei tähenda see seda, et tulemusi ja lahendusi sellistes mängudes kasutada ei saaks.

## 5.2 Graafikamootor

Üks parimatest lahendustest prototüübi loomiseks on *Simple DirectMedia Layer* ehk SDL abiteek [45]. See kasutab pikselgraafikat ning võimaldab kõik kuvamist vajavad elemendid mängumaailma tekitada. Käesoleva töö raames on prototüübi toimima saamisel kasutatud lihtsamaid õpetusi  ja näiteid [46]. SDL teeb põhimõtteliselt rasked elemendid lihtsamaks ja on platvormide suhtes väga paindlik.

Töö käigus selgus, et mõned kindlad graafilised efektid on SDL’i abil raskemini teostatavad, samuti mõned elemendid SDL’ist on keerulisemalt kasutatavad kui mõne muu kõrgema tasme teegi puhul [47, 48]. Samas, kõik mida käesolev töö prototüübi jaoks oli vaja, on olemas ja kiiresti üles seatav.

Renderdatud mängu laius on 1280 pikslit ja kõrgus 960 pikslit.

## 5.3 Liikumine, Block A*

Selle implementatsiooni lõid Tansel Uras ja Sven Koenig [18]. See implementatsioon on loodud seoses erinevate *any angle pathfinding’u* algoritmide empiirilise võrdlemisega. Käesoleva töö

autor kohandas algoritmi prototüübi maailmaruumiga ja muutis lahendust nii, et algoritmi väljund ei oleks mitte mõõtetulemused teekonna leidmisest vaid teekond ise.

Maailmaruum on kuvatud 20 korda suuremana kui see on esitatud pathfinding algoritmi jaoks. *Pathfinding* algoritm näeb põhimõtteliselt ainult iga ruudu alumist paremat nurka ja moodustab oma teekonna nende nurkade peal. Tänu sellele, et algoritm kasutab arvutamiseks nurki, on maaimaruumi kuvamisel ja takistuste määramisel rohkem kontrolli. Maailmaruumi tekstiline kujutus on näha 1. lisas ja selle renderdatud tulemus on kujutatud 2. lisas. Graafikamootor mõistab igat tähte kui ruutu ning joonistab seinad sinna, kus on @ karakter. *Pathfinding* algoritm aga tõlgendab punkte avatud nurkadena ning kõike muud kinniste nurkadena.

Block A* algoritm moodustab 48 väikese ruudu pikkusega ja 64 väikese ruudu laiusega maailmaruumist 16 ruutu pikkusega 12 väikest ruutu ja laiusega 16 väikest ruutu. Algoritm leiab suured ruudud, mis on alguse ja lõpuga ühendatud ning nende sees olevad teekonnad. Kui teekonda ei ole silutud, on näha, et teekonnas esineb väikeseid ebaoptimaalsuseid just suurte ruutude ühenduspunktides. Nurkade silumine kasutab minimaalse koguse ressursse ja ei ole algoritmi halvendav faktor.

Et agendil oleks võimalik teekonna leidmise algoritme kasutada, on agendil vaja koordinaate tõlgendada. Agendi liikumine toimub igas kaadris mingisuguse pikslihulga võrra, liikudes suundades X ehk horisontaalselt ja Y ehk vertikaalselt. Funktsioon võtab sisendiks kaks X, Y koordinaati ning tõlgendab selle suunaks 360 kraadi ulatuses.

Seejärel arvutab funktsioon välja kui palju X ja kui palju Y suunas peaks agent liikuma, et jõuda lõppkoordinaadile. Nii tõlgendab programm kõik koordinaadid tegelikuks liikumiseks. Omavahelist kokkupõrget sellel meetodil ei ole ning selle mõistlik tekitamine nõuaks teistsugust lähenemist.

## 5.4 Shadowcasting

Nägemiseks kasutatud algoritm arvutab vaatevälja rekursiivse *shadowcasting'uga.* Meetod on algselt Björn Bergstromi [49] loodud ja see tehti algselt *roguelike* mängude jaoks - peategelase vaatevälja joonistamiseks ja vastaste ründamisala loomiseks. Kuna *roguelike* mängudes liiguvad agendid ruudult ruudule, on ka nägemine ruudukaupa. Lahenduse formaat ühtis käesoleva töö prototüübiga.

Algoritm leiab mängumaailmas olevad takistused ning arvutab nende taha varjud, mille piirkonnas olevaid objekte agent ei näe.

```
-...---  - = visible cells
 -..---  # = blocking cell
  -#---  . = cells in blocker's shadow
   ----
    ---
     --
      @
```

**Joonis 10 - Näide *shadowcasting* algoritmi poolt loodud varjust [49]**

Algoritm toimib rekursiivselt agendist väljapoole liikudes kaheksas erinevas osas.

```
           Shared
           edge by
Shared     1 & 2      Shared
edge by\      |      /edge by
1 & 8   \     |     / 2 & 3
         \1111|2222/
         8\111|222/3
         88\11|22/33
         888\1|2/333
Shared   8888\|/3333  Shared
edge by-------@-------edge by
7 & 8    7777/|\4444  3 & 4
         777/6|5\444
         77/66|55\44
         7/666|555\4
         /6666|5555\
Shared  /     |     \ Shared
edge by/      |      \edge by
6 & 7      Shared     4 & 5
           edge by
           5 & 6
```

**Joonis 11 - Oktiilid, mille järgi *shadowcasting* algoritm ruute läbi töötab [49]**

Lahendus on käesoleva töö jaoks implementeeritud nii, et moodustuks 136-kraadine ala sinna, kuhu agent on suunatud.

Sellisel meetodil, kus nägemine luuakse ruudustikuga on on palju vähem punkte vaja läbi kontrollida kui lahendustes, kus kontrollitakse iga pikslit. Ruudukaupa nurkade kaudu nägemine on prototüübi jaoks piisav ning kasutab võrdlemisi vähe ressursse. Iga piksli eraldi

kontrollimisel tuleks näiteks käesoleva töö prototüübi puhul läbi töötada 20 korda rohkem punkte.

Kui nägemisalgoritm kontrollib ruutu, millel on vastane, lisandub see agendi ohtude mälusse. Analoogselt kui algoritm leiab ruute läbides objekti, lisatakse see vastavasse mäluloendisse.

## 5.5 Kohandatud otsuste langetamise algoritmi implementatsioon

Esimene funktsioon jälgib, kas mõni uus staatus on kriitilises olukorras. Samuti jälgib see funktsioon, kas ohtude loendis on mõni uus objekt. Kui kumbki neist on tõsi, vaatab teine funktsioon kogu staatuse üle ning vormistab eesmärgid.

Vajaduste hinnangu funktsioon, mis kasutab eksponentsiaalselt tõusvat olulisuse hinnangut, määrab eesmärkidele väärtuse. *Insertion sort* funktsioon seab eesmärgid tähtsusjärjekorda. *Insertion sort* on väikeste nimekirjade sortimiseks kiire ja seda on lihtne implementeerida [50].

Agendil on ka iga objektiliigi ning ohu jaoks eraldi mäluloendid. Need loendid võivad olla võrdlemisi väikesed. Käesoleva töö prototüübi puhul on igas loendis maksimaalselt 6 objekti ja kasutusel 5 objekti. Kui mõnesse loendisse tekib kuues objekt, eemaldatakse peale *insertion sorti* kuuendale positsioonile langenud objekt loendist.

Iga uue eesmärgi tekkimisel kontrollib agent ka oma mälus olevad objektid üle. Objektide hinnangu funktsioon hindab kõiki objekte vastavalt sellele kui väärtuslikud need eesmärgi rahuldamiseks on, kui palju aega nende kasutamine võtab ning kui kaugel need objektid agendist on. Seejärel seab eelmisega analoogne *insertion sort* funktsioon need objektid tähtsusjärjekorda. Eesmärkide klass loob vastavalt saadud eesmärkidele ja agendi mälus olevatele objektidele tegevuste jada. Objektide ümberhindamine on iga uue eesmärgi tekkimisel oluline, sest agent liigub ringi ning objekti väärtus sõltub ka sellest kui kaugel see agendist on.

Igale eesmärgile vastab teatud hulk tegevusi, mis aitavad eesmärgiga seotud vajadust rahuldada. Tegevused võivad olla antud juhul kolme erinevat tüüpi.

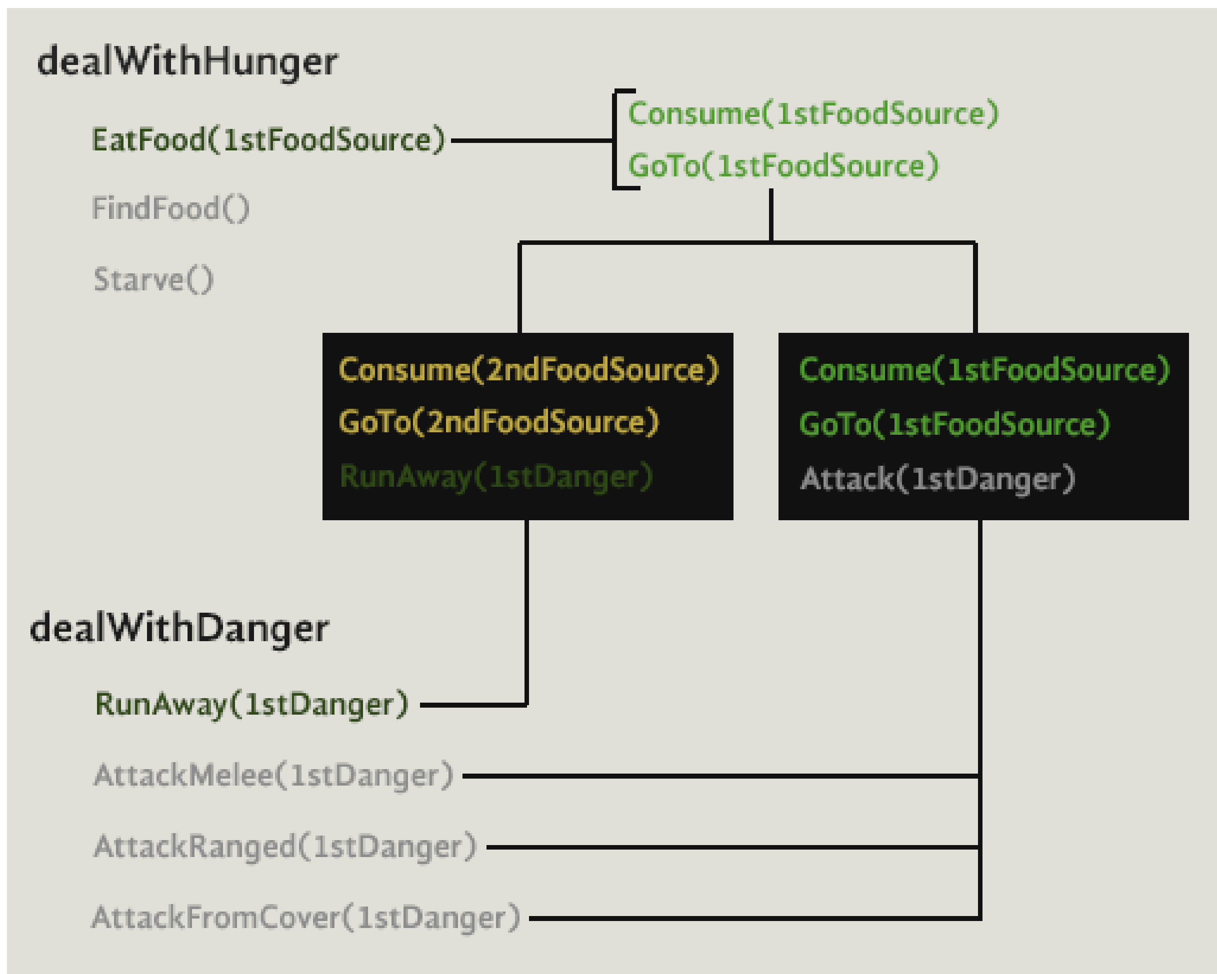
1. Lihtne tegevus on omaette ning agent oskab selle järgi tegutseda (näiteks *GoTo*).

2. Kontekstuaalne tegevus on kogumik tegevusi, millest funktsioon valib kõikide teiste aktiivsete eesmärkide hulgast parima (näiteks *RunAway, AttackMelee, AttackRanged*).

3. Liittegevus koosneb tegelikult mitmest väiksemast lihtsast tegevusest (näiteks *Eat*, mis koosneb tegevustest *GoTo* ja *Consume*) .

Tegevustel on veel selline omadus, et nad võivad olla kas objektiga seotud või mitte. Näiteks kui tegelasel on teada kus söök asub, liigub ta selle juurde tegevustega, mis on otseselt selle söögiobjektiga seotud. *GoTo* liigutab agendi sinna kus objekt on ning *Consume* teisendab selle sama objekti kasulikkuse ja kahjulikkuse väärtused agendi olekusse. Samas kui objekti ei ole, võib agent valida hoopis *Starve* (ootamine ja nälgimine) või *FindFood* (ringi liikumine ja toidu otsimine) tegevuse. Kummagi jaoks pole objekti vaja.

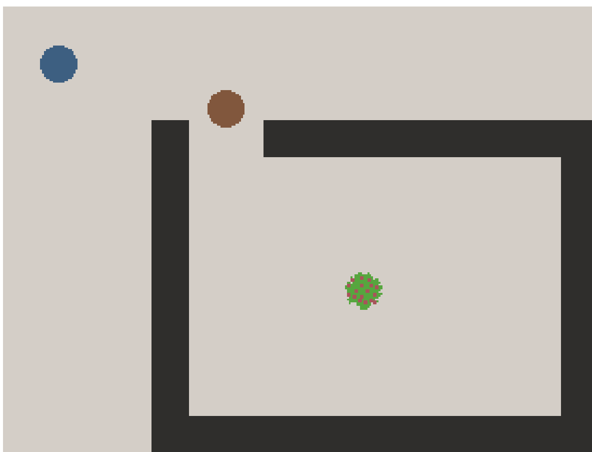
Teine mitmekülgsust lisav element on see, et iga tegevusega võivad seotud olla mingisugused tingimused või reeglid, mis muid tegevusi mõjutavad. Näiteks kui eesmärkide jada on selline nagu joonisel 12.



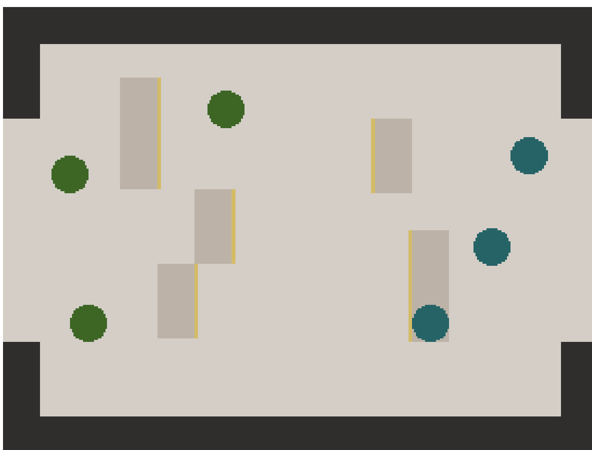

**Joonis 12 - Agendi põgenemisest tulenev tegevuste muutumine**

Sellises olukorras saab määrata tegevusele *RunAway* tingimuse. Kui see valitakse, muutuvad kõikide ülemiste tegevustega seotud objektide hinnangud kallimaks. Sellises olukorras saab funktsioon järgmist parimat objekti kasutada, sest see ei ole nii kallis kui eelnevalt parim variant. Kui teiseks või kolmandaks parimaid variante ei ole, võib piisavalt suure nälja ja piisavalt väikese ohuhinnangu korral agent otsustada, et on parem riskida ja püüda toiduni jõuda. Vastasel juhul võivad tegevused *Starve* ja *FindFood* osutuda paremaks variandiks kui ohuga riskida, seda juhul kui ohuhinnang on piisavalt suur.

Selliste tingimustega saab tekitada üldiseid seoseid erinevate tegevuste vahel ja muuta käitumist loomulikumaks. Antud juhul aitab see lahendus agendil oma mälus olevad objektid mõistlikult ümber hinnata ning eelnevalt esitatud ohu ja toiduallika situatsiooni lahendada.



**Joonis 13 - Toidu ja ohu situatsioon prototüübis**



**Joonis 14 - Sõjaolukorra situatsioon prototüübis**

Rohkemate agentidega sõjaolukorras kasutavad agendid kaitseks kattega ruute. Kohad, kus seda teha saab on esindatud objektidena. Tänu sellele on programmil võimalik hinnata katteid samuti nagu toidu- ja joogiallikaid. Agent hindab kõiki ohte ja otsustab milline neist on kõige suurem. Seejärel otsustab ta selle ohu suhtes, kas on mõistlik rünnata lähedalt, kaugelt või ära joosta. Kaugelt rünnaku kasuks otsustanud agent võtab arvesse oma tugevuse ja vastase ohtlikkuse ning valib, kas kasutada katet või mitte. Parima katte valib agent just kõige ohtlikuma vastase suhtes.

Agent võtab seejärel tegevuste jada ja sooritab tegevusi järjestikku. Enne iga tegevuse täitmist kontrollib ennatlik vajaduste funktsioon, ega tegevuse ajakulu ei põhjusta agendi staatusele uut kriitilist olukorda. Seda teeb programm nii, et vaatab olenevalt tegevuse ajakulust agendi staatust mitme kaadri võrra ette. Vajadusel funtksioon käivitab uue eesmärkide loomise jada, mille puhul agent avastab, et eelnevalt parim lahendus, mis oli näiteks toore toidu kogumine ja selle küpsetamine on liiga ajakulukas ning tal jõuab selle ajaga tekkida kriitiline janu. Paremaks variandiks saab vähem aega võtva toidu tarbimine ning kohe peale seda janu kustutamine.

# Kokkuvõte

Töö eesmärkideks oli hinnata ja võrrelda erinevaid agendipõhiseid tehisintellekti lahendusi ning luua tulemuste põhjal strateegiamängu prototüüp.

Et oleks võimalik erinevaid tehisintellekti meetodeid võrrelda, tuli teha kindlaks, mis atribuutide järgi neid oleks üldse võimalik mõõta. Töö raames sai loodud agendipõhine strateegiamängu prototüüp, mis lahendab kõiki töö käigus autori poolt tuvastatud probleemseid situatsioone. Agendi implementatsioonis on rakendatud *pathfinding'u* ja nägemise algoritmi, mis on mõlemad *grid'il* põhineva siseruumides toimuva mängu jaoks kõige optimaalsemad lahendused. Tehisintellekti otsuse langetamise lahenduseks arendas töö autor erilaadseid lahendusi kasutava meetodi.

Mõned tehisintellekti lahendused on teistest objektiivselt paremad, aga üldjuhul tuleb mänguloojal siiski silmas pidada lahenduste praktilisust, ehk milliseid probleeme tuleks nende projektis oleval agendil lahendada. Natuke keerulisema agendi käitumise jaoks tuntumaid otsuse langetamise meetodeid enam kasutada ei saa ning on vaja kasutada modifitseeritud lahendusi. C++ koos SDL abiteegiga on väga hea vahend lihtsama mänguprototüübi loomiseks.

Kuna käesoleva töö otsuste langetamise algoritm on modulaarne ja uute objektide lisamisel kiiruse keerukus märkimisväärselt ei tõuse, oleks agendile võimalik lisada eesmärke juurde. Eesmärgid võiksid näiteks tegeleda kõrgemate Maslow püramiidi astmetega. Võimalik oleks agendile rohkem parameetreid lisada, mille järgi ta otsustab, et saavutada keerukamat esilekerkivat käitumist.

Otsuste langetamise algoritmile oleks võimalik strateegilise käitumise arendamiseks integreerida ka muid lahendusi. Näiteks taktikaline käitumine, mille puhul üks agent on teiste rühmajuht ning võib edastada enda eesmärke ja objekte käsklusena teistele agentidele. Populaarseks uurimisteemaks on õppivad ja kohanemisvõimelised otsustusalgoritmid. Hetkel on need veel mängude tarbeks ebapraktilised. Käesoleva töö otsuste langetamise algoritm on piisavalt modulaarne, et selle põhjal võiks potentsiaalselt kohandada ka õppiva otsustusalgoritmi.

Töö raames loodud prototüübi põhjal oleks võimalik luua mäng. Selle töö agendi tehisintellekt ei seaks mängule olulisi ressursipiiranguid. Otsuste langetamise algoritm on prototüüp, nägemise ja teekonna leidmise algoritmid on valmis arendatud meetodite kohandatud

implementatsioonid. Töö alguses leitud mõõtmismeetoditega on võimalik erinevaid tehisintellektide loomise meetodeid edukalt hinnata.

# Summary

The aim of this thesis was to evaluate and compare different agent based artifical intelligence approaches and create a prototype of a strategy game based on the results. Aspects and measuring methods had to be determined and formulated to be able to compare different artificial intelligence methods.

A prototype of an agent based strategy game artificial intelligence was created. The approach solves the problematic situations that the author ascertained the agent would face in the prototype. The agent is composed of implementations of pathfinding and vision simulation which are both the most optimal approaches when the games map is grid based and indoors. The more suitable the game developer wants the decision-making to be, the more customization has to be done. For the prototypes agents decision-making approach the author created a unique customized method.

Some artificial intelligence solutions are objectively better than others, but generally the game developer has to recognize the practicality and usefulness of the different approaches. Basically they have to consider what problems the agent in their game has to solve. C++ with SDL is a very useful tool for creating simpler prototypes and also for creating a full game.

# Kasutatud kirjandus


[1] I. Millington ja J. Funge, Artificial Intelligence for Games - 2nd Ed., Elsevier, 2009.

[2] M. DeLoura, Game Programming Gems 2, Publishers' Design and Production Services, 2001.

[3] D. E. Knuth, „Big Omicron and Big Omega and Big Theta,“ *SIGACT News,* kd. 8, nr 2, pp. 18-24, 1976.

[4] S. Rabin, Game AI Pro 2, CRC Press, 2015.

[5] J. Copeland, „What is Artificial Intelligence?,“ May 2000. [Võrgumaterjal]. Available: http://www.alanturing.net/turing_archive/pages/reference%20articles/what_is_AI/What%20is%20AI07.html. [Kasutatud 29 12 2015].

[6] B. Scott, „The Illusion of Intelligence,“ %1 *AI Game Programming Wisdom*, Charles River Media, 2013.

[7] N. Kirby, Introduction to Game AI, Course Technology, 2011.

[8] Lionhead Studios, „Black & White,“ EA Games, Feral Interactive, 2001.

[9] D. Johnson ja J. Wiles, „Computer Games with Intelligence,“ %1 *FUZZ-IEEE*, 2001.

[10] Maxis Software, Inc., „The Sims,“ Electronic Arts, Inc., 2000.

[11] S. Cass, „Mind games [computer game AI].,“ *Spectrum, IEEE,* kd. 39, nr 12, pp. 40-44, 2002.

[12] P. H. M. Spronck, „Adaptive game AI,“ Universitaire Pers Maastricht, 2005.

[13] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill ja M. Preuss, „RTS AI: Problems and Techniques,“ 2014.

[14] E. W. Dijkstra, „A Note on Two Problems in Connexion with Graphs,“ *Numerische Mathematik,* kd. 1, pp. 269-271, 1959.

[15] S. Koenig ja M. Likhachev, „D* Lite,“ %1 *AAAI Conference of Artificial Intelligence*, 2002.

[16] R. E. Korf, „Depth-first iterative-deepening,“ *Artificial Intelligence,* kd. 27, nr 1, pp. 97-109, 1985.

[17] D. Harabor ja A. Grastien, „The JPS Pathfinding System,“ %1 *26th National Conference on Artificial Intelligence*, 2012.

[18] T. Uras ja S. Koenig, „An Empirical Comparison of Any-Angle Path-Planning Algorithms,“ %1 *Symposium on Combinatorial Search*, 2015.

[19] K. Daniel, A. Nash, S. Koenig ja A. Felner, „Theta*: Any-Angle Path Planning on Grids,“ *Journal of Artificial Intelligence Research,* kd. 39, pp. 533-579, 2010.

[20] A. Nash, „Lazy Theta*: Faster Any-Angle Path Planning,“ 16 juuli 2013. [Võrgumaterjal]. Available: http://aigamedev.com/open/tutorial/lazy-theta-star/. [Kasutatud 29 detsember 2015].

[21] A. Nash, S. Koenig ja C. Tovey, „Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D,“ %1 *AAAI Conference on Artificial Intelligence*, 2010.

[22] A. Nash ja S. Koenig, „Any-Angle Path Planning,“ *Artificial Intelligence Magazine,* kd. 34, nr 4, pp. 85-107, 2013.

[23] P. Yap, N. Burch, R. C. Holte ja J. Schaeffer, „Block A* and Any-Angle Path-Planning,“ %1 *The Fourth International Symposium on Combinatorial Search*, 2011b.

[24] D. Ferguson ja A. Stentz, „Using interpolation to improve path planning: The Field D* algorithm,“ *Journal of Field Robotics,* kd. 23, nr 2, pp. 79-101, 2006.

[25] T. Uras ja S. Koenig, „Speeding-up any-angle path-planning on grids,“ %1 *International Conference on Automated Planning and Scheduling*, 2015.

[26] P. Yap, N. Burch, R. Holte ja J. Schaeffer, „Any-angle path planning for computer games,“ %1 *Artificial Intelligence and Interactive Digital Entertainment*, 2011a.

[27] J. C. Wilk, „Comparative study of field of view algorithms for 2D grid based worlds,“ 2009.

[28] Rogue Basin, „Rogue Basin Ray Casting,“ 6 December 2015. [Võrgumaterjal]. Available: http://www.roguebasin.com/index.php?title=Ray_casting. [Kasutatud 30 December 2015].

[29] A. Milazzo, „adammil,“ 8 oktoober 2014. [Võrgumaterjal]. Available: http://www.adammil.net/blog/v125_Roguelike_Vision_Algorithms.html. [Kasutatud 30 detsember 2015].

[30] J. E. Bresenham, „Algorithm for computer control of a digital plotter,“ *IBM Systems journal,* kd. 4, nr 1, pp. 25-30, 1965.

[31] Rogue Basin, „Rogue Basin Shadow Casting,“ 22 detsember 2015. [Võrgumaterjal]. Available: http://www.roguebasin.com/index.php?title=Shadow_casting. [Kasutatud 30 detsember 2015].

[32] Rogue Basin, „Rogue Basin Permissive FOV,“ 12 juuni 2014. [Võrgumaterjal]. Available: http://www.roguebasin.com/index.php?title=Permissive_Field_of_View. [Kasutatud 30 detsember 2015].

[33] Rogue Basin, „Rogue Basin Digital FOV,“ 3 jaanuar 2012. [Võrgumaterjal]. Available: http://www.roguebasin.com/index.php?title=Digital_field_of_view. [Kasutatud 30 detsember 2015].

[34] A. H. Maslow, „A theory of human motivation,“ *Psychological Review,* kd. 50, nr 4, pp. 370-396, 1943.

[35] I. Alatalo, *Need-based artificial intelligence for a computer game: Case Spaceship Captain - prototype,* Oulun ammattikorkeakoulu, 2014.

[36] A. Repenning, „Excuse me, I need better AI! Employing Collaborative Diffusion to make Game AI Child's Play,“ %1 *ACM SIGGRAPH Symposium on Videogames*, 2006.

[37] J. Orkin, „Applying Goal-Oriented Action Planning to Games,“ %1 *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[38] M. Buckland, Programming Game AI by Example, Wordware Publishing, 2005.

[39] R. C. Martin, „UML Tutorial: Finite State Machines,“ *Engineering Notebook Column,* 1998.

[40] S. Russell ja P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2002.

[41] Wikipedia, „Fuzzy logic temperature,“ 24 detsember 2015. [Võrgumaterjal]. Available: https://en.wikipedia.org/wiki/Fuzzy_logic. [Kasutatud 4 jaanuar 2016].

[42] D. L. Pittman, „Practical Development of Goal-Oriented Action Planning AI,“ The Guildhall at Southern Methodist University, 2007.

[43] Monolith Productions, „F.E.A.R. First Encounter Assault Recon,“ Sierra Entertainment, 2005.

[44] M. Dawson, Beginning C++ Through Game Programming, Third Edition, Course Technology, 2011.

[45] „Simple DirectMedia Layer,“ [Võrgumaterjal]. Available: https://www.libsdl.org/index.php.

[46] Lazy Foo' Productions, „Tiling SDL2," [Võrgumaterjal]. Available: http://lazyfoo.net/tutorials/SDL/39_tiling/index.php.

[47] „Allegro," [Võrgumaterjal]. Available: http://liballeg.org/.

[48] „SFML," [Võrgumaterjal]. Available: http://www.sfml-dev.org/.

[49] B. Bergström, „FOV using recursive shadowcasting," 27 aprill 2014. [Võrgumaterjal]. Available: http://www.roguebasin.com/index.php?title=FOV_using_recursive_shadowcasting. [Kasutatud 30 november 2015].

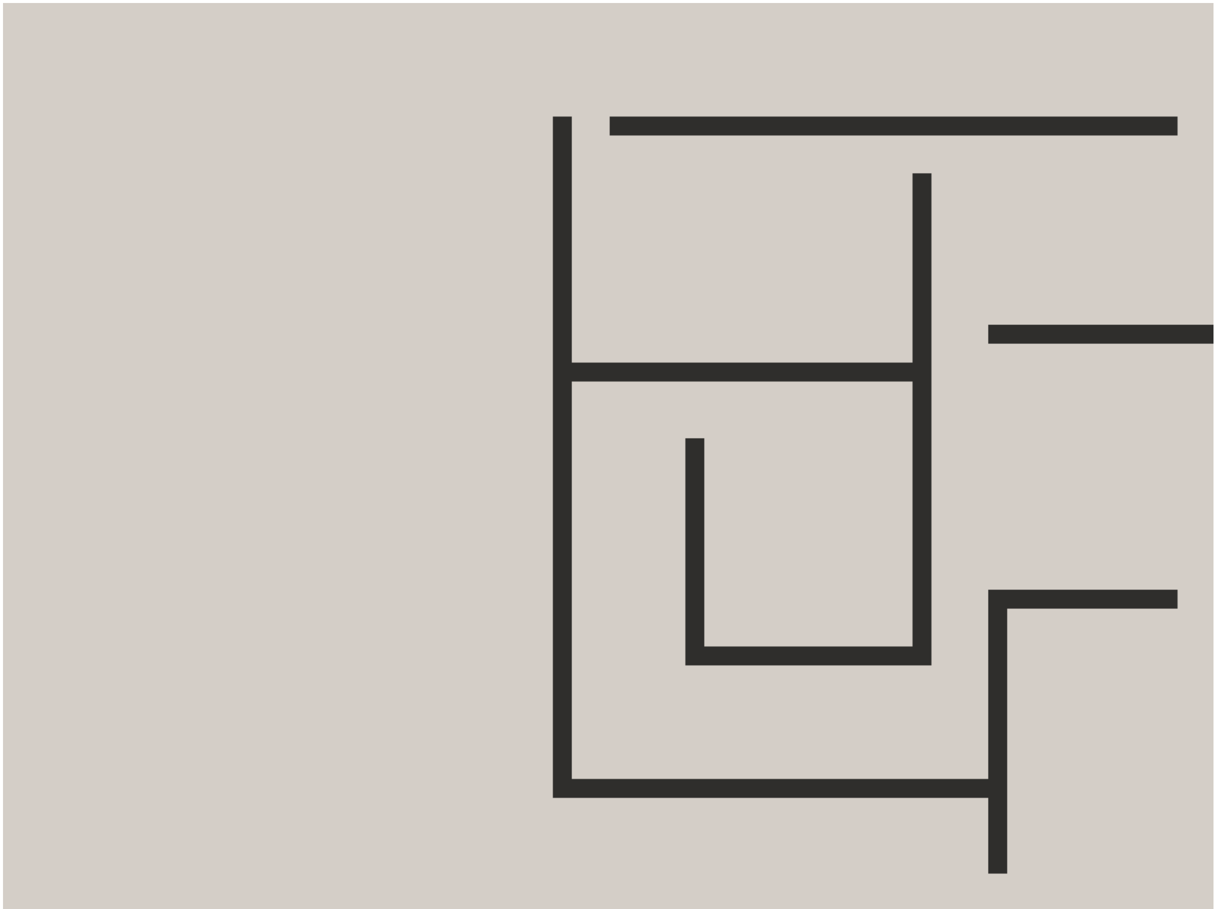
[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest ja C. Stein, „Insertion Sort," %1 *Introduction to Algorithms - 2nd ed.*, MIT Press ja McGraw-Hill, 2001, pp. 15-21.

# Lisa 1

```
type octile
height 48
width 64
map
................................................................
................................................................
................................................................
................................................................
................................................................
............................TT.TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT..
............................T@.T@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@..
............................T@..................................
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@..TTTTTTTTTTTTT
............................T@TTTTTTTTTTTTTTTTTT@..T@@@@@@@@@@@@
............................T@@@@@@@@@@@@@@@@@@@@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.................T@...............
............................T@.....TT..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@...............
............................T@.....T@..........T@..TTTTTTTTTTT..
............................T@.....T@..........T@..T@@@@@@@@@@..
............................T@.....T@..........T@..T@...........
............................T@.....T@TTTTTTTTTTT@..T@...........
............................T@.....T@@@@@@@@@@@@@..T@...........
............................T@.....................T@...........
............................T@.....................T@...........
............................T@.....................T@...........
............................T@.....................T@...........
............................T@.....................T@...........
............................T@TTTTTTTTTTTTTTTTTTTTTT@...........
............................T@@@@@@@@@@@@@@@@@@@@@@@@...........
...................................................T@...........
...................................................T@...........
...................................................T@...........
...................................................T@...........
................................................................
................................................................
```

**Joonis 15 - Maailma kujutus karakteritena**

# Lisa 2



**Joonis 16 - Maailma graafiline kujutus prototüübis**