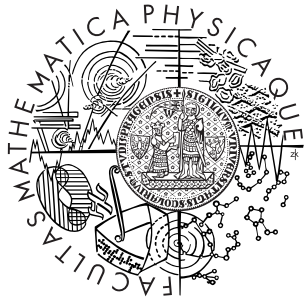
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Eva Pešková

Codewars, vojna robotov

Katedra distribuovaných a spolehlivých systémů

Vedúci bakalárskej práce: RNDr. Tomáš Poch
Studijní program: Všeobecná informatika

2010

íslování stránek

Ďakujem svojmu vedúcemu RNDr.Pochovi za vedenie práce, cenné rady a maximálnu ústretovosť a svojej rodine za morálnu podporu.

Prehlasujem, že jsem svoju bakalárskou prácu napísala samostatne a výhradne s použitím citovaných prameHov. Súhlasím so zapožičovaním práce a jej zverejňovaním.

V Praze dne 6.8.2010 Eva Pešková

Obsah

1 Úvod **5**
- 1.1 Motivácia . . . 5
- 1.2 Ciele práce . . . 7

2 Analýza **9**
- 2.1 Virtuálny svet . . . 9
  - 2.1.1 Súčasti virtuálneho sveta . . . 9
  - 2.1.2 Život v prostredí . . . 12
- 2.2 Programovateľnosť postavy . . . 15
  - 2.2.1 Rozširenie sveta vzhľadom na programovatelnosť . . . 16
  - 2.2.2 Možné prístupy k programovaniu virtuálneho sveta . . . 23
  - 2.2.3 Robotština . . . 24
- 2.3 Konkrétna realizácia jazyka . . . 28
  - 2.3.1 Možné prístupy . . . 28
  - 2.3.2 Práca s jazykom . . . 29
  - 2.3.3 Interpret medzikódu . . . 31

3 Implementácia **35**
- 3.1 Virtuálny svet . . . 35
  - 3.1.1 Interpret a jazyk . . . 39
  - 3.1.2 Generovanie bludiska . . . 40
- 3.2 Správa okien . . . 40

4 Porovnanie **42**

5 Záver **44**
  - 5.0.1 Zhodnotenie splnenia cieľov . . . 44
  - 5.0.2 Možne rozširenie CODEWARS . . . 44

Literatúra **45**

Prilohy **45**

A CD **47**
- A.1 struktura CD . . . 47

B Gramatika robotštiny **48**

Název práce: Codewars, vojna robotov
Autor: Eva Pešková
Katedra (ústav): Katedra distribuovaných a spolehlivých systémo
Vedoucí bakaláYské práce: RNDr. Tomáš Poch
e-mail vedoucího: tomas.poch@d3s.mff.cuni.cz

Abstrakt: V predloenej práci navrhujeme a implementujeme prostredie pre testovanie algoritmov. Tieto algoritmy majú odrá ae stratégiu pou ite¿nú v rozli ných bojových hrách proti umelej inteligencii. Práca analyzuje mo né podmienky pre víeazstvo a vplyvy astí kódu na vykonávanie algoritmu. Kladie sa dôraz na jednoduchose jazyka popisujúceho algoritmus, rozaírite¿nose a Klí ová slova: robot, vojna, prostredie

Title: Codewars, battle of robots
Author: Eva Peaková
Department: Katedra distribuovaných a spolehlivých systémo
Supervisor: RNDr. Tomáa Poch
Supervisor's e-mail address: tomas.poch@d3s.mff.cuni.cz

Abstract: In the present work we study ... Uvede se anglický abstrakt v rozsahu 80 a 200 slov. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut sit amet sem. Mauris nec turpis ac sem mollis pretium. Suspendisse neque massa, suscipit id, dictum in, porta at, quam. Nunc suscipit, pede vel elementum pretium, nisl urna sodales velit, sit amet auctor elit quam id tellus. Nullam sollicitudin. Donec hendrerit. Aliquam ac nibh. Vivamus mi. Sed felis. Proin pretium elit in neque. Pellentesque at turpis. Maecenas convallis. Vestibulum id lectus. Fusce dictum augue ut nibh. Etiam non urna nec mi mattis volutpat. Curabitur in tortor at magna nonummy gravida. Mauris turpis quam, volutpat quis, porttitor ut, condimentum sit amet, felis.

Keywords: robot, war, enviroment

Kapitola 1

Úvod

Bakalárska práca sa zaoberá, ako názov naznačuje, súťažením algoritmov. Súťaženie však musí prebiehať zábavnou a zrozumiteľnou formou. Vhodným suťažiacim kritériom je napísanie algoritmu chovania robota, ktorý bojuje s inými robotami o život.

Jedným z prvých programov, ktorý používal koncept programovateľného robota, je KAREL[8]. Tento program bol vytvorený pre podporu výučby programovacích jazykov. Príležitosť potrápiť svoje schopnosti zdanlivo banálnymi príkladmi jednoduchým jazykom (napr. napíšte program, po skončení ktorého bude robot v strede svojho sveta), viedla k vytvoreniu súťaží. Doteraz je populárna hra C++ robot[14], prebiehajúja pomocou mailovej konferencie. Tie rôznymi obmedzeniami, kladenými na algoritmus nútili programátora pristupovať k problému kreatívne.

Dobrým príkladom takejto súťaže je hra HERBERT[9], kde bolo cieľom prejsť v neprerušenej postupnosti všetky biele políčka v šachovnicovom svete. Zistiť postupnosť príkazov, ktoré sú riešením, bolo triviálne. Problémom však bolo zapísať ich pomocou rekurzií a s čo najmenším počtom použitých príkazov.

1.1 Motivácia

Ako sme naznačili v úvode, vymýšľanie stratégií chovania robota a následné pozorovanie výsledku je dobrým spôsobom, ako si overiť svoje schopnosti zábavnou a hravou formou. V predstavených hrách je ale jediným súperom zadaný problém. Súťažiaci tak rátajú s pevne danými dátami (rozostavenie políčiek a pod.). Na podobnom princípe je založené aj súbežne pustenie algoritmov s tým, že ich vykonávanie môže ostatným škodiť. Algoritmus súťažiaceho musí rátať s nerovnakým prostredím vo virtuálnom sveta a môže dokonca profitovať zo znalosti stratégie súperov. Preto sa v tejto bakalárskej práci zaoberáme algoritmami robotov, ktoré sa budú vykonávat paralelne v rovnakom prostredí.
Smerom, ktorým by sme sa chceli uberať, je hra ROBOCODE[2]. Užívateľ programuje v jazyku Java tanky. Tie hľadajú a ničia nepriateľské tanky. Vo virtuálnom svete sú nastavené základné obmedzenia, s ktorymi musí uživateľ počítať. Napr. čo sa stane, ak strieľa tank príliš často a pod. Predstavme si extrémy, ktoré existujú v takýchto hrách a definujme smer, v ktorom sa bude uberať bakalárska práca.

- ARES - hra dvoch hráčov. Odohráva sa na mieste simulujúcom pamäť počítača. Úlohou hráča je naprogramovať robota v tomto prostredí v strojovom kóde (assembleri). Kritériom úspešnosti je program, ktorý sa bude vykonávať (prakticky hocijaký), a hodnotiacou funkciou je čas, za ktorý program pobeží. Program skončí v okamihu, keď sa pokúsi vykonať neplatnú inštrukciu (napr. delenie nulou, skok na adresu nula, prázdnu inštrukciu). Cieľom je prepisať pamäť takým spôsobom, aby sa druhý program ukončil. Hráč dopredu nepozná ani program protihráča, ani dáta, ktorými je inicializovaná pamäť. Hráči sa však môžu dopredu dohodnúť. Oba programy sú podobne ako v reálnom svete uložené v pamäti počítača a keďže pamät je zdieľaná, môžu si navzájom prepisovať dáta alebo dokonca inštrukcie. Extrém, ktorý tento koncept prináša, je:
  - použitie jednorozmerného priestoru (pole pamäte)
  - obmedzenie na počet hráčov (2)
  - nutnosť poznať do hĺbky assembler, jazyk, v ktorom je zapísaný algoritmus
  - možnosť zásahu do algoritmu ostatných hráčov a teda jeho zmena
  - objekty vyskytujúce sa v hre sú iba dáta a inštrukcie
  - hra sa odohráva na najnižšej možnej úrovni - nie sú tu telá robotov, ale iba priamo program.
- POGAMUT je už vysoko komplexná hra niekoľkých robotov. Algoritmy sa dajú programovať v Jave, ktorý umožňuje používať špeciálne znaky jazyka, ako je napríklad preťaženie, dedičnosť, atd. Prostredie je trojrozmerné a tým majú roboti možnosť širokej škály pohybov, skákania po stene, pohľady hore a dole, strely do boku atď. Spôsob, akým sa ubližuje robotom, je intuitívny - robot vystreľuje obmedzené množstvo striel a má na výber viac zbraní, ktoré sa líšia presnosťou zásahu. Hráč má dokoca možnosť ručne riadiť vlastného robota proti naprogramovanému a tým otestovat vhodnosť jeho algoritmu.
  Extrémom v tomto kontexte sú:
  - trojrozmerný priestor, ktorý ponúka množsto možností, ako realizovať pohyb - lezenie po stenách, skákanie, gravitácia
  - množstvo objektov pôsobiacich na robota, napr. úkryty, strely, vyhýbanie sa strelám, obehnutie prekážky, hľadanie vhodných ciest
  - zobrazovanie je dobre zrozumiteľné pre pozorovateľa, používajú sa základné fyzikálne javy
  - možnosť obmedzeného videnia robotov, robot sa môže schovať alebo byť tak ďaleko, že ho iný robot nezaregistruje

Uvedené hry uspokojujúco spĺňajú základnú problematiku boja algoritmov. V ARES-e je kritériom zostať nažive podobne ako v ROBOCODE, v POGAMUT-e je naviac vopred dané kritické množstvo protivníkov. Hodnotiaca funkcia je rýchlosť, kto skôr splní cieľ, vyhráva.

1.2 Ciele práce

Cieľom bakalárskej práce je umožniť užívateľovi naprogramovať robota, ktorého algoritmus chovania bude súťažiť s ostatnými robotmi v prostredí, kde si navzájom môžu ubližovať. V tomto prostredí sa budú dať definovať aj rôzne nároky na algoritmus. Tým je myslené napríklad obmedzenie použitia niektorých prvkov, nároky na spotrebu zdrojov a pod.

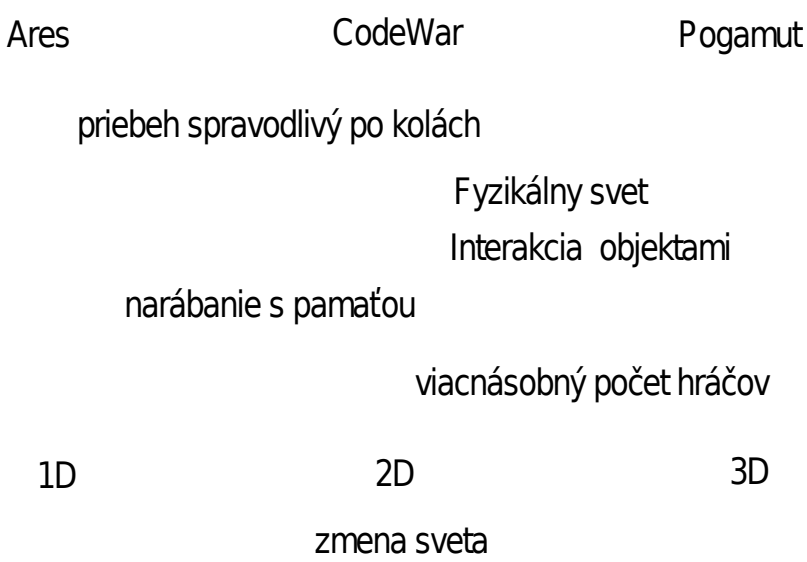
Výsledný program bakalárskej práce je nástroj, v ktorom môže užívateľ meniť virtuálny svet, kde sa odohrávaju súboje, naprogramovať robota a zistiť úspešnosť napísaného algoritmu. Program by mal byť napísaný tak, aby bol portabilný.
Predstavili sme si dve hry, ktoré istým spôsobom predstavujú extrémy v tomto obore. Práca by mala spojiť niektoré koncepty z týchto extrémov s tým vytvoriť nové súťažné prostredie.
Preto je potrebné sa zamerať najmä na:

Vytvorenie virtuálneho sveta Naprogramovaný robot bude žiť vo virtuálnom svete, v ktorom sa odrážajú fyzikálne zákony a kde je prirodzené sledovať postup algoritmu. Súčasťou sveta budú objekty, ktoré interagujú s robotmi a tak prinášajú do tvorby stratégií komplikovanejšie prvky. V ARES-e reprezentujú tieto objekty dáta uložené vo virtuálnom svete (pamäti). V prípade POGAMUT-u sú to steny, teleporty, priepasti, strely a pod. Codewars sa pokúsi spojiť obidva tieto koncepty, to znamená, že na hráča bude kladený nárok aj z h)ladiska prostredia aj z hľadiska algoritmu. Predpokladá sa, že takto vytvorený virtuálny svet bude možné upravovať a vytvárať, aby algoritmy nemuseli byť napisané "na telo" jednej mapy/počiatočnému stavu sveta. Hráč bude mať možnosť ovplyvniť/zmeniť správanie tohoto sveta.

Dynamika virtuálneho sveta Naprogramovaní roboti budú mať možnosť bojovať, t.j. si ubližovať, výsledok útoku bude známy v okamihu ublíženia. Tým sa ľahšie vyhodnotí program. To je smer, v ktorom sa budeme približovať hre POGAMUT, keďže v ARES-e hráč nevie, kedy a ako ublížil robotovi. Roboti vo svete sa budú pohybovať všetkými smermi a interagovať s ostatnými objektami vo svete. Pre ľahšiu zrozumiteľnosť vývoja virtuálneho sveta sa objekty budú riadiť základnými fyzikálnymi javmi, ako je napríklad odraz. Hráč by mal mať tiež voľbu útoku na blízko aj na diaľku kvôli lepším strategickým možnostiam.

Životný cyklus robotov Život robotov bude začínať vstupom do virtuálneho sveta a končiť jeho opustením. Možnosť nejakého znovuzrodenia ako je to v hre POGAMUT sa nebude pripúšťať, ale môže byť otázkou ďalšieho rozšírenia. Víťazný robot ostáva živý. Životný cyklus robota sa bude dať naprogramovať pomocou nejakého programovacieho jazyka, ktorý bude dostatočne zrozumiteľný aj pre laika.

Vlastnosti robotov Ani jeden z uvedených programov ale nemá možnosť špecifikovať, aké budú jednotlivé vlastnosti robotov. Či robotov skolí jedna rana



Obr. 1.1: Smer uvažovania nad prostredím

(ARES), alebo tu je aj možnosť nejakého obmedzeného znovuzrodenia (POGAMUT). V bojových hrách sa tiež ukázalo vhodné umožniť, aby si hráč pred samotným vstupom do sveta mohol tieto vlastnosti upraviť a tým ovplyvniť priebeh súboja.

Obrázok 1.1 naznačuje smer, v ktorom sa bude práca uberať. Obrázok popisuje jednotlivé koncepty, s ktorými sa môžeme v týchto hrách stretnúť. Ich umiestnenie naznačuje, kde sa tieto koncepty vyskytujú a ako súvisia s Codewars.

Kapitola 2

Analýza

Súčasťou tejto kapitoly je zdôvodnenie jednotlivých rozhodnutí, ktoré sme navrhli. Chceme vytvoriť taký virtuálny svet, v ktorom môzeme pozorovať chovanie robotov, ktorý obsahuje rôzne objekty a v ktorom je netriviálne napísať dobrý program.

2.1 Virtuálny svet

Neoddeliteľnou súčasťou hry je virtuálny svet (prostredie), v ktorom sa bude súboj odohrávať. Najskôr vysvetlíme, čo všetko svet obsahuje, ako sa v ňom žije z hľadiska robota, na čo musí dbať uživateľ (ako sa narába s objektom v programe) a ako jednotlivé pravidlá prispievajú k úspešnosti algoritmu.
Uvažovaný svet bol vybraný dvojrozmerný, pretože poskytuje dostatok možností pre dianie na ploche (smer pohybu, zrozumiteľne vykresľovanie stavu, podpora fyzikálnych zákonov a pod.) a súčasne nie je obtiažne implementovateľný. Tým sme sa vyhli obom predstaveným extrémom, a to 1D a 3D prostrediu.

2.1.1 Súčasti virtuálneho sveta

Základnou súčasťou virtuálneho sveta sú roboti, ktorí sa smú pohybovať. Bez ďalších objektov by bolo logickým vyústením hry len jednoduché nájdenie robota. Roboti sa môžu brániť iba pohybom (nič iné by vo svete nebolo). Tento prístup je zaujímavý, ale jednotvárny, ráta sa stále s tým istým stavom sveta). Preto uvažujeme aj o ďalších objektoch.

Hráč aktívne nevstupuje do vykonávania algoritmu, preto je nutné charakterizovať virtuálny svet z hľadiska robota, resp. uživateľovou znalosťou objektov, na ktoré môže robot reagovať.

Objekty uvažované vo svete vzhľadom na to, čo môže algoritmus využívať, sú nasledovné:

Robot ako objekt
: Základnou schopnosťou robota je, že môže ubližovať ostatným robotom. Povedali sme, že robot získava informácie o objektoch svojho sveta, teda roboti

sami musia mať vlastnosti objektu sveta. Tym sme sa úspešne vyhli konceptu ARES-a, ktorý nedokáže zistiť objekty k nemu nepatriace.

Strely

Roboti by mali vedieť útočit na diaľku. To možno docieliť viacerými spôsobmi:

- Robot zaútočí z diaľky na konkrétne miesto okamžite je známy výsledok útoku. To dáva užívateľovi jasnú spätnú väzbu. V okamihu útoku na diaľku je treba rozhodnúť, kedy smie robot zaútočiť týmto spôsobom. Ak môže robot zaútočiť na akékoľvek miesto, potom ostatní roboti nielenže nemajú možnosť sa útoku vyhnúť, ale strácaju aj informácie o tom, odkiaľ útok prišiel (na každé políčko môže byť zaútočené). Stratégie sa zredukujú na dva príncípy - náhodne útočenie z diaľky na nejaké miesta alebo na pohyb robota dovtedy, pokiaľ sa nenájde cieľ a následne na masívny útok na cieľ. Pripomína to hru typu lode, ktorá neponúka dostatočný priestor pre vymýšľanie stratégií.

- Minimálne je teda nutné obmedziť pravidlami, na ktoré miesta sa môže útočiť z diaľky. Intuitívne a ľahko zobraziteľné je vymedziť polomer zásahu, robot bude môcť útočiť len do určitej vzdialenosti. Stále je ale nemožné spozorovať útok na diaľku a tak sa mu vyhnúť. Od predchádzajúcej metódy je však minimálne zaistená väčsia možnosť taktického manévrovania. Robot si z útoku bude vediet približne odhadnúť, že niekde v okolí je nepriateľský robot. V ARES-e toto nie je možné, aj keď útok na hocijaké políčko áno.
- Ďaľšou možnosťou je vytvorenie strely. Strela je objekt, ktorého jedinou činnosťou je pohybovať sa predvídateľne vpred po dobu dopredu známeho času (v danom smere výstrelu). Dodatočne pridávame, že strela skončí svoju činnosť aj v okamihu, keď spraví útok na blízko - zasiahne robota. V tom okamihu bude robot informovaný o úspechu tým, že strela vybuchne a robot sa súčasne sa nemusí obávať toho, že strela aj jeho neskôr zasiahne. To je vlastne robotova odmena. Pre boj z diaľky bol teda vytvorený nový objekt - strela.

Robot útočí tým, že na cieľ vystrelí, strela následne spraví útok na blízko. Tento prístup poskytuje väčšiu voľnosť pri útočení, nakoľko stači rozhodnúť, v ktorom smere má strela ísť. Takto je možné približne odhadnúť smer, odkiaľ strela prišla. To je dôležité pri rozhodovaní sa, či robota napadnúť priamo (nablízko), alebo odpovedať streľbou.
Robot nebude mat nekonečnú možnosť strieľať, inak by hra nekládla nároky na zistenie, ktorým smerom je vhodné zaútočiť. Navyše ak robot strieľa neobmedzene, potom môže byt hracie pole zahltené strelami, čo výrazne spomalí simuláciu. Preto je počet striel obmedzený. Strelivo ale nie je možné dopĺňať. Preto sa strela okamžite po vykonaní útoku vráti k robotovi, ktorý ju vystrelil a ten ju môže použit znova. Takto robot okamžite zistí úspešný zásah.

Steny

Ďalším uvažovaným prvkom sú objekty, ktoré môže robot využívať na svoju obranu. Útočiť na robota môžu objekty len z blízka (robot z blízka alebo z diaľky, čo je ale vlastne strela z blízka). Vo virtuálnom svete potom potrebujeme niečo, čo obmedzí pohyb. Týmito objektami budú steny. Robot ich môže využívat ako strategický úkryt, podobne ako vojaci využívaju terénne nezrovnalosti.

Prepadliska - zakázané miesta

Zaujímavým strategickým obmedzením sú miesta, na ktoré robot za žiadnych okolnosti nesmie stúpiť. V prípade POGAMUT-u sú to priepasti, jamy, tekutá láva a pod. V ARES-e zase miesto v pamäti, ktoré obsahuje neplatnú inštrukciu. Existencia prepadlísk umožňuje plánovať stratégiu využívajúcu silný útok z diaľky a vyhne sa silným útokom zblízka.

Štartovné pozície robota

Ak roboti nemajú dopredu dané miesto na mape, potom sa ich štartovné pozície musia generovať buď náhodne alebo vypočítať tak, aby umiestnenie bolo nejakým spôsobom spravodlivé (nebol zvýhodnený ani jeden robot). Rozoberme si podrobnejšie jednotlivé ne/výhody virtuálneho sveta bez štartovných pozícií robotov:

Roboti vygenerovaní vedľa seba

Táto situácia môže nastať kedykoľvek počas behu programu, robot na ňu musí vedieť zareagovať. Preto jenáhdné generovanie vlastne spravodlivé.

Roboti rovnomerne rozhádzaní po svete

To môže zabrať netriviálne veľa času, obzvlášť ak je virtuálneho sveta zložená z veľkeho množstva stien a úzkeho priestoru pre pohyb robotov.

Kvôli zabezpečeniu spravodlivosti generovania pozícií robotov bude použitá heuristika. No je otázne čo chápať pod spravodlivosťou. Robot má za úlohu reagovať na každú situáciu, nie je preto nutné uvažovať o špeciálne vypočitaných miestach a tak pojem spravodlivosť stráca svoj význam.
Problémom zostáva množstvo robotov na mape a dlhé generovanie počiatočných pozícií. Z tohoto dôvodu sa pristúpilo k možnosti vytvorenia štartovacích políčok. To prináša okrem iného aj možnosť definovať, pre maximálne koľko robotov je mapa ideálna (ráta sa s týmto maximálnym počtom robotov). Ak ale uživateľ zadá viac robotov, simulácia sa aj potom môze uskutočniť. Je nutné len uživateľa upozorniť, že prebehol pokus umiestniť robota náhodne a či bol tento pokus úspešný. Teraz si už ale môžeme dovoliť dopredu daný počet pokusov a nepokúšať sa tak zbyttočne dlho. Po neúspechu nasleduje odstránenie robota zo simulácie. Ak aj pokus nebol úspešný, nič významné sa nedeje, pretože existuje na mape virtuálneho sveta dostatočné množstvo robotov, umiestnených na štartovacích políčkach.

2.1.2 Život v prostredí

Roboti žijú vo svete a snažia sa zničiť ostatných. Nato vplývajú nasledovné vlastnosti robotov.

- Jednou z nich je **veľkosť útoku**. To je vyjadrené celým číslom. Tak sa dá škoda zistiť presne a neobjavia sa problémy s malými číslami alebo zlomkami, ako je to v prípade reálnych čísel (v C je napríklad 0 vyjadrená ako malé nenulové číslo). Čim väčšie číslo, tým väčšia škoda sa deje robotovi.

- V prípade vzdialeného útoku je tiež dôležitou otázkou, **ako ďaleko** môže robot zaútočiť. Ak je toto čislo vopred dané, mal by o tom robot vedieť dopredu, aby mohol svoj algoritmus prispôsobit.

- Útok robotov prebieha na úrovni ich tiel a nie programu (viď hra typu ARES). Životne dôležitou otázkou je, **koľko zásahov** robot vydrží. Keďže útok je vyjadrený pomocou celých čísel, je vhodné vyjadriť celočíselne aj životnosť robotov.

- Ďalšou vhodnou vlastnosťou by mohla byť **aktívna obrana** proti útokom. Doteraz sa robot mohol brániť len dostatočným počtom životov. Môže sa tak stať, že pri malom počte životov bude stačiť jedna rana a robot zahynie. Aktívna obrana znamená, aké množstvo zranenia bude pohltené pred jeho smrťou. Výsledný efekt je ale rovnaký, akoby sa životnosť zvýšila a preto táto vlastnosť nebola použitá.

- Na algoritmus vplýva aj **rýchlosť** , s akou sa robot pohybuje. Pri vyššej rýchlosti má napríklad väčšiu šancu sa vyhnúť strele.

Život v prostredí obnáša aj pohyb. Mapa môže byť rozľahlá a roboti nemusia byť na dosah útoku. Roboti sa na výhodnejšie miesto musia dostať. Pohyb môže byť realizovaný ako jednoduchý presun z miesta A na miesto B (teleportácia), alebo ako postupný prechod na druhé miesto. Na druhej strane je ale vhodné definovať najmenšiu vzdialenosť, o ktorú sa robot zámerne pohne. Hovorme tomu **krok**.
Ideálne sa javí postupné vykonávanie pohybu (plynulý prechod na ďalšie miesto), pretože je to prirodzené a takýto pohyb poskytuje priestor pre návrh stratégií, ako napríklad zo smeru robota odhadnúť jeho ďalšiu pozíciu. Navyše, keď sa robot ocitne pred priepasťou, vie, že robot za ním pravdepodobne touto cestou nepríde, pretože by mu bolo ublížené. Otázkou je, ako sa bude tento algoritmus vykonávať, akým spôsobom sa robot dostane do zadaného miesta.

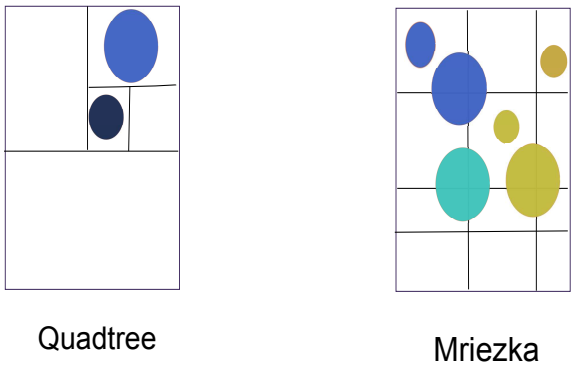
- Jedným zo spôsobov je, že samotná mapa bude ponúkať možnosť navigovať robota. Tento koncept zjednodušuje uživateľovi písanie algoritmov. Následkom tohoto rozhodutia ale nebude mať užívateľ možnosť zistiť, ako dlho bude trvať robotovi cesta na dané miesto a teda nebude mu umožnené ani predpokladať,

ako sa za ten čas zmenil virtuálny svet. To výrazne obmedzuje vymýšľanie stratégií.

- Pohyb sa bude bez ohľadu na dostupnosť miesta realizovať po priamke. Pohyb sa bude realizovať po priamke bez ohľadu na dostupnosť miesta. Potrebujeme kontrolovať, kedy sa má robot prestať hýbať a pokračovať vo vykonávaní algoritmu. To ale nemôžeme vzhľadom na pozíciu, na ktorú má dôjsť, pretože tá nemusí byť dosiahnuteľná. Preto pohyb obmedzíme počtom krokov. Nakoľko hovoríme o priamke, je jednoduché vypočítať, koľko krokov robota by bolo treba vykonať. Preto namiesto kontrolovania, či robot skončil na žiadanom mieste, stačí skontrolovať, či robot vykonal dostatočné množstvo krokov a potom pohyb ukončiť.

Pohyb, ktorý môže urobiť robot, je pohyb v ľubovoľnom smere. Tymto spôsobom dosiahneme, že svet sa bude zdať reálnejší a budú sa dať použiť základné fyzikálne javy. Smer je určený vektorom $[x, y], x, y \in N$, takže sa ním dá vyjadriť presne smer pohybu. Konkrétny pohyb je potom aproximovaný vektorom a súradnice sa budú vypočítavať v závislosti od čase.
V súvislosti s pohybom je možné uvažovať o rozšírení virtuálneho sveta nad rámec popisu v 2.1.1, a to konkrétne o posuvné steny, ktoré budú chápané ako zvláštny druh stien. Steny zabraňujú pohybu robota a striel, preto prínos posuvných sten je v tom, že robot bude môct zmeniť prostredie sveta tak, aby zodpovedal jeho konkrétnemu algoritmu. Vie posunúť stenu, aby ho chránila pred streľbou, alebo posunúť stenu tak, aby zabránila postupu robota na nejake miesto. Spôsob s posuvnými stenami ma oproti využívaniu konkrétneho prostredia sveta naviac tú výhodu, že algoritmus musí počitať so zmeneným svetom a tým sa zvyšuje náročnosť hry. Z tohoto dôvodu obsahuje virtuálny svet aj posuvné steny. **Kolízie** V súvislosti s pohybom nastáva je nutné si položiť otázku, kedy a ako budú objekty navzájom interagovať. Kolízia nastane vtedy, keď obrazy reprezentujúce objektov majú spoločný neprázdny prienik. To kladie nemalé nároky na štruktúru virtuálneho sveta, ale súčasne to má tiež vedľajší efekt. Čim väčší obrázok bude symbolizovať objekt, tým väčšia je možnosť kolízie. To môže byť trochu nepríjemné, ale poskytuje to možnosť pre ďalšie rozšírenia, keď napr. silnejší robot (viac života, väčší útok) bude mať povinne väčší aj zodpovedajúci obraz.

Kolízia môže nastať prakticky pri akomkoľvek malom pohybe. Na samotné ukladanie objektov do mapy existuje jednoduchý trik, rozdeliť mapu na malé políčka a každé políčko obsadiť práve jedným objektom. To prináša väčšie nároky na pamäť (obzvlášť, ak je veľká mapa a políčka malé), ale zato kolíziu vieme určiť okamžite. Stači zistiť, či v danom políčku, kde leži výsledok pohybu, je objekt jediný. Tento spôsob sa dosť často používa v bludiskách, kde sú objekty rovnako veľké (každé zaberá práve jedno políčko). Nazvime ich **diskrétne bludiská**. Okrem veľkosti políčka má tento prístup ale problém aj s rozhodnutím, kde objekt patrí. Ak sa v mape pohne len o niekoľko pixelov, bude patriť stále do toho istého políčka, pretože mapa je rozdelená staticky (políčka sa nepohybujú s objektom). Kúsok objektu teda môže presahovať nad rámec políčka. Potom takto implementovaná kolízia nemusí presne zodpovedat tomu, ako je zobrazená. Preto boli uvažované nasledujúce algoritmy bezpečnejšiu detekciu kolízie, kde sa používa iba veľkost objektu:



Obr. 2.1: Porovnanie quadtree a mriežky

Quadtree Princípom quadtree[10] je delenie priestoru na stále menšie časti v závislosti na tom, koľko je v danej oblasti objektov. Práve v tom tkvie jeho výho, nerozdeľuje priestor rovnomerne, čím efektívne šetrí pamäť a procesorový čas.

Mriežková metóda Mriezková metóda je jedna z najstarších a pozostáva podobne ako distrétne bludisko z množstva disjunktných políčok. Na rozdiel od diskrétneho bludiska môže každá táto oblasť obsahovať niekoľko objektov. Sprvu sa to zdá ako pohoršenie, ale týmto spôsobom sa iba vymedzujú možné kolízne objekty. Systém práce algoritmov je ukázaný na 2.1. Objekty s rovnakou farbou patria do rovnakej obálky.

Pre lepšiu implementovateľnosť bola zvolená mriežková metóda.
Keď máme vyriešený spôsob, akým sa rieši kolízia, je treba určiť, ako na ňu jednotlivé objekty reagujú. Rozhodli sme sa takto:

Robot vs. strela Strela ukončí svoj život výbuchom a uštedrí robotovi náležité zranenie. Robotovi sa v tomto okamihu ako dôsledok preruší akákoľvek činnosť, ktorú predtým vyvíjal, napríklad pohyb. To čiastočne zodpovedá nabúraniu programu, ako je tomu u ARES-a. Strela potom už z hroe uvedených príčin nepokračuje. Je však na ďalšom rozšírení programu, ako sa strela po zásahu bude chovať. Strela môže zasiahnúť aj toho, kto ju vystrelil. Z toho dôvodu, aby robot nemal tendenciu strieľať všade, ale aby bol prinútenú rozmýšľať, či to neublíži aj jemu. Z toho dôvodu je rychlosť striel vždy vyššia ako rýchlosť robota.

Robot vs. Robot Roboti nemôžu navzájom interagovať inak ako ublížením a pri kolízií sa teda aplikuje útok na blízko. Tento spôsob zľahčuje písanie algoritmu, nakoľko sa nemusí explicitne deklarovať útok nablízko (útok na diaľku algoritmus musí explicitne vyjadriť, útok na blizko je automatický). Útok na blízko bude prevedený iba robotom, ktorý spôsobil kolíziu.

Robot vs. stena Robot sa vzhľadom na funkciu steny, tak, ako bola definovaná, zastaví

Robot vs. 5osuvná stena Posuvná stena má schopnosť meniť svoje miesto. Robot ju má možnosť posunúť (musí byť pri nej a pohybovať sa). Stena by sa

mala hýbať len s robotom, keďže cieľom je, aby mu sústavne poskytovala úkryt. Stena sa tak pri kontakte s robotom posunie v smere, v akom ide robot, v súlade s fyzikálnymi zákonmi. Ak sa snažia stenu posúvať dvaja roboti a smer ich pohybu je vyjadrený dojrozmernym vektorom, potom sa stena pohybuje v smere vektoroveho súčtu týchto dvoch smerov.

Robot vs. prepadlisko Robot by na prepadlisko nemal stúpať. Z definície musí byť robot potrestaný, keď vstúpi na toto políčko. Ako najjednoduchší spôsob sa ponúka strata životov a následne zastavenie alebo prejdenie prepadliska za cenu niekoľkonásobnej straty životov. Bol implementovaný druhý spôsob. Pri prejdeni prepadliska aj za cenu toho, že bude robot polomŕtvy, sa pri dostatočnom množsve sa života môže ešte podieľať na simulácii.

Stena vs. strela Strela sa bude môcť od obyčajnej steny odrážať. Stena nie je nikdy primárnym cieľom. Preto nemá zmysel, aby strela ukončila svoju dráhu pri narazeni na stenu. Jediný dôsledok by bol, že robot môže strelu vidieť a tým by bola prezradena pozícia strelca. Ak sa strela bude ďalej pohybovať, aj keď iným smerom, bude možné strieľať aj za "rohom", umýselne mýliť protivníka vyslanim strely tak, aby sa odrazila, zaútočiť na robota a pritom nedať robotovi žiadnu informáciu o svojej pozícií, zasiahnuť robota, aj ked sa schovava (aby neexistovalo nič ako dokonalý úkryt). Preto bolo rozhodnuté o odrazení strely od steny.

Stena vs. posuvná stena Stena z definície bráni pohybu. Teda logicky bráni pohybu aj posuvnej stene a následne robotovi za ňou.

Posuvná stena vs. posuvná stena Posuvná stena má ako hlavný cieľ poskytknúť úkryt, preto nemá zmysel, aby vyvolala pohyb druhej stenou, keď to robot nečaká. Nič to neprinesie. Na druhej strane je zbytočné komplikovať si implenetáciu tým, že sa špeciálne pre istý ibjekt sa stena zastaví. Preto stenou môže pohnúť každý objekt

Strela vs. posuvná stena Otázkou je, či by sa mala posuvná stena hýbať aj pri kontakte so strelou. Keďže strela vlastne vykonáva útok robota na diaľku, je možné stenou posunúť a to v smere strely. Strela sa súčastne odrazí.

Strela vs Strela
Strela nijak neprofituje so zrážky s inou strelou, takže sa nič nestane. Bolo by možné síce strelu ďalšou strelou odchýliť, ale to znamená pre robota zložité vypočítavanie, kde je strela, kde strela bude a v akom smere má vystrediť. Výsledkom je vyhnutie sa strele, čo zvládne obyčajný pohyb rýchlejšie.

2.2 Programovateľnosť postavy

Chovanie robota na mape bude mať užívateľ možnosť naprogramovať v nejakom jednoduchom jazyku. Z toho vyplývaju ďalšie nároky na svet ako aj nároky na programovací jazyk robota.

2.2.1 Rozširenie sveta vzhľadom na programovatelnosť

Získavanie informácii o svete

Užívateľ ma napísat algoritmus, ktorým by sa robot riadil. Preto je nutné vhodným spôsobom získavat informácie o okolí robota, na ktorý má robot reagovať.

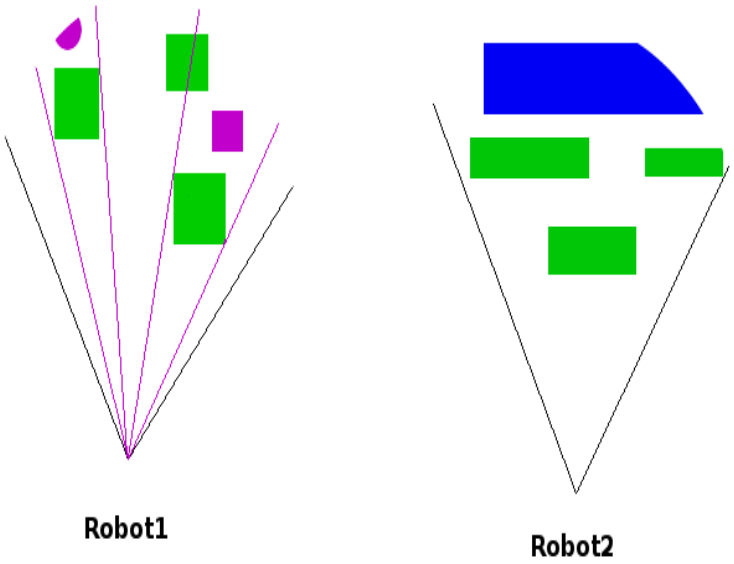
Virtuálny svet je dvojrozmerný, ako bolo povedané a tak si užívateľ môže získať prehľad o tom, čo je vo svete pozrením do mapy (vizuálne). Hneď bude vedieť, s čím môže počitať na danom mieste, ak sa tam robot ocitne. Ak by robot získaval informácie inak, ako to vidí uživateľ, nastane nekompatibilita v chápaní toho, čo práve robot vidí. Výsledkom môže byť nezrozumiteľnosť vykonávania algoritmu. Preto teda bude robot objekty tiež vidieť.
Videnie robota zahŕňa ešte ďalšie problémy, na ktoré sa treba zamerať, a to, ako ďaleko a akým spôsobom bude robot vidieť. Ako ďaleko bude robot vidieť, je pre algorimus dôležité, pretože čím ďalej robot vidí, tým ucelenejšiu predstavu o svete bude mať a tým je pravdepodobnejšie, že nájde nepriateľského robota a bude mu môcť škodiť. To môže byť jednoducho vyriešené nastavením pevnej vzdialenosti alebo inom riadení robota v závislosti na užívateľovej ľubovôli.
Ďalšou otázkou je, akým spôsobom robot vidí, ako presne určiť, že objekt na danom mieste je robotom viditeľný. Najskôr začnime vymedzením priestoru, v aktorom by mal robot vidieť. Sme v dvojrozmernom priestore a rozhodli sme sa pre plynulý pohyb, preto je dôvod predpokladať, že aj oblasť viditeľnosti robota bude súvislá. To znamená, že ak v robotovej oblasti viditeľnosti je objekt A a objekt B, potom je v tejto oblasti aj objekt C, ležiaci na spojnici A a B. Z toho vyplýva, že uvažované objekty sa budú dať popísať nejakým dvojrozmerným útvarom.
Najvhodnejšia sa javí kruhová výseč, podobná tej, ako vrhá baterka v tme. Tento spôsob je pre ľudské oko prirodzený a preto bol aplikovaný.
Viditeľnosť robota tak bude znamenať, že sa budú brať do úvahy všetky objekty, ktoré sa nachádzaju v kruhovej výseči, definovanej polomerom a uhlom. Ak by všetky tieto objekty boli viditeľné, nemala by zmysel stena, ktorá má poskytovať úkryt. Preto z týchto objektov, ktoré môžu byť viditeľné robotom, sa musia vybrať len tie, ktoré neblokujú výhľad.
Úkryt môžu používať všetky objekty, nielen roboti. Inak by sa obtiažne implementovala viditeľnosť, závislá na tom, či v určitej vzdialenosti stojí robot. Tiež nie je logický dôvod, aby sa viditeľnosť správala rozdielne pre rôzne typy objetov. Preto je u každého objektu okamžite jasné, či je priehľadný alebo nie.

Priehľadnosť objektov bola vybraná následovne:

- Strela by nemala poskytovať úkryt a teda blokovať viditeľnosť. Je to len nástroj k prevedeniu útoku na diaľku a ako taký by nemal tak výrazne ovplyvňovať svet, ako je poskytovanie úkrytu. V porovnaní s reálnym životom sa tiež obvykle nestáva, aby agresor hádzal po obeti ľadničku.

- Stena poskytuje úkryt a z toho dôvodu blokuje viditeľnost. Z rovnakého dôvodu to plati pre posuvnú stenu.

- Prepadlisko neblokuje viditeľnosť. Z definície vieme, že prepadlisko neblokuje



Obr. 2.2: Príklady kruhovej výseče

pohyb, iba ho sťažuje. Preto je rozumné, aby robot vedel, že cez neho môže prejsť. To znamená, že musí cez prepadlisko vidieť.

- Samotný robot blokuje viditeľnosť. Toto rozhodnutie vyplýva z toho, že všetky objekty, ktoré sa pohybujú smerom k robotovi, sa v pripade kolízie zastavia. Preto sa da čakať, že robotom neprejde ani lúč, určujúci viditeľnosť.

Navrhovaným algoritmom bolo sprvu vygenerovanie bitových masiek:
Algoritmus ráta s tým, že výseč sa pokryje najmenším obdĺžnikom a s ním sa potom pracuje. Princípom je rozdelenie tohoto obdĺžnika na niekoľko častí a potom predgenerovanie viditeľnosti políčok. Kazdé políčko potom bude charakretizované bitovou sekvenciou, kde jednotky značia, že políčko musí byť bezpodmienečne vidieť. Potom samotná viditeľnosť znamená lineárne prejsť objekty, nachádzajúce sa v tomto políčku a takto získanú bitovú masku aplikovať na našu vygenerovanú. Potom je jednoduchou kontrolou zistené, ktoré bitové masky zostali nezmenené.
Výhodou tejto metódy je lineárna časová zložitosť vzhľadom na veľkost pokrytej plochy. Nevýhodou je nutnosť rozdeliť výseč na rovnaké políčka a tým buď nerešpektovat veľkosť objektov alebo rátať s objektami vyskytujúcimi sa na viacerých políčkach.
Vzhľadom na uvedené nevýhody bola navrhnutá a použitá výsečová metóda Tá pozostáva z vytvorenia výseče pre každý objekt, ktorý môže robot vidieť. Následne sa zistí pokrytie, viď 2.2 Pre každý objekt najskôr zistíme, či sa nachádza v pásme viditeľnosti (na obrázku naznačený silnými čiarami). Pre každý objekt, ktorý blokuje viditeľnosť, vytvoríme ďalšiu výseč s počiatkom v mieste robota (na obrázku čiarkované). Potom je viditeľný každý objekt, ktorého ľubovoľná časť je mimo zjed-

notenia týchto výsečov. V praxi to znamená, že u každého objektu si zapamätáme, akú oblasť by tieto objekty zneviditeľnili, keby blokovali výhľad. Zoradíme ich podľa vzdialenosti od umiestnenia robota, aby orezávali výseče len tých objektov, ktoré sú vzdialenejšie a teda ich môžu pokryť. Každý blokujúci objekt potom znižuje uhol, pod ktorým je vzdialenejší objekt vidieť. Avšak existuje situácia, keď tento algoritmus zlyhá, ako je vidieť na 2.2 pri robotovi2. Na tomto obrázku sú červenou vynačené objekty, ktoré robot nevidí, zelené vidí. Avšak robot2 by veľkú prekážku vidieť nemal. Riešením je v takejto situácii veľký objekt rozdeliť na dva. Táto metóda je ale inak veľmi presná. Počita s rozdielnými veľkosťami objektov, je však K-krát pomalšia než vyššie uvedená metóda ($K \approx 10$).

Ciele algoritmu

Popísaný svet je svetom robotov, ktorí chcú vyhrať a dosahujú to tým, že ostatným robotom znemožňujú naplniť ich cieľ (uhýbajú strelám, škodia im útokmi). Naším cieľom je upraviť tento virtuálny svet tak, aby bolo pre uživateľa podnetné napisať algoritmus. To znamená použiť to, že robot vidí, na netradičné ciele nereprezentované v hrách ARES a POGAMUT. K tomu sa viažu podmienky úspešnosti algoritmu. Ponúkajú sa nasledovné ciele robotov:

Zostať na bojisku posledný
: Úspešným algoritmom je taký, ktorého výsledkom je, že robot, ktorý sa ním riadi, zostane na bojisku posledný. Spôsob, akým možno dosiahnúť tento cieľ, je iba boj so súperiacimi robotmi, útočenie na ich reprezentáciu vo svete. Tento cieľ spĺňa požiadavky na súboj algoritmov, ako je definovaný v ARES-ovi. Tento spôsob bol použitý, lebo je intuitívny.

Obranný robot
: Pouvažujme nas tým, či robot môže zvíťaziť aj inak než útočením na súperov. Veľmi sofistikovane vyzerá robot, ktorý zmätie svojho protivníka natoľko, že začne bojovať s inými robotmi. Nazvime tohoto robota obranným a robota, ktorý má za úlohu ostatných zničiť, robot-agresor.
Otázkou je, či svet, ktorý sme vytvorili, je vhodný aj pre obranných robotov. Je, pretože všetko, čo využiva agresor, používa aj tento obranca. Streľbou a odrazmi od stien sa snaží nalákať agresora na iný objekt, posuvnými stenami sa môže priblížiť na miesto, kde by ho inak zbadali a vystreliť do miest, kde by jeho strela inak nedosiahla. Môže naviesť súpera na prepadlisko a nechať ho tam biedne zhynúť, t.j manipulátor sa snaži zostaž nažive s minimom zabitých nepriateľov na konte.
Ak robot nebude mať za úlohu zničiť nejakého súpera, nebude mať ani motiváciu aktívne ich vyhľadávať. Takto môže algoritmus zdegenerovať na úroveň čakania na to, kým sa objaví nepriateľ, pred ktorým by sa dalo utekať. Tento koncept sám o sebe je zaujimavy, ale je dosť ťažko zhodnotiteľný. V práci obranného robota preto nepodporujeme samostatne.

Zničenie konkretného robota
: Aby sa roboti jednotlivých hráčov dali rozlíšiť, sú pomenovaní unikátnymi menami. Ďalšou možnosťou je tak zničenie konkrétneho robota. Robot, ktorý

má takýto cieľ, vyhrá v okamžiku, keď tohoto cieľového robota zničí práve jeho útok. Ak je kritériom samotný tento cieľ, znamená to iba, že tento robot je bojový robot, ktorý môže skončiť svoju misiu skôr za predpokladu, že zničí toho správneho protivníka. Potom sa len potuluje po svete ako potenciálna obeť a nebojuje. Takýto koncept opat nie je zaujímavý samostatne.

Navštívenie miesta Problémom algoritmu obranného robota je to, že nie je primerane akčný. Rozhýbať ho môže prinútiť dodatočná podmienka, napríklad nájdenie nejakého objektu. Potom sa pre viťazstvo musí robot skutočne premiestňovať. Takýmto bjektom ale môžu byt len roboti, (čo v podstate splýva s predchádzajúcim tvrdením). Steny, prepadliská a pod., sa ako objekt túžby nehodia (v prepadlisku príde o život, na miesto steny sa robot nikdy nedostane, stretnutie strelou tiež končí fatálne). Kvôli princípu čisto obranného robota je nutné definovať špeciálne miesta, ktoré nebudú objektami a nebudú mať význam pre nikoho iného, len pre robota, ktorý o toto miesto požiadal. Toto miesto by malo byť súčasťou mapy, aby užívateľ videl, čo presne po robotovi chce. Tieto miesta sa budú číslovať a tým budú prenositeľné aj do ďalších máp. Aby to robot nemal také ľahké a aby užívateľ mohol testovať algoritmus, možno ako špeciálne miesto označiť plánované štartovné pozície robotov.

Obmedzenie počtu nepriateľov

Nájdenie miesta, ktoré si robot vybral, môže vyzerať aj tak, že robot bude mať bojový algoritmus, s ktorým prejde celú mapu a raz na to svoje pole dorazí. Použije stratégiu "dôjdem tam a kto sa proti mne postavi, je mŕtvy robot". Tým sa robot nebude nijak líšiť od obyčajného agresora. Preto sa hodí kombinácia hľadania miesta spojiť s obmezením počtu robotov, ktoré môže robot zabiť pred dosiahnutím cieľa. Pre praktické účely by bolo vhodné preddefinovať maximálny počet zabitých nepriateľov rovný $pocetrobotovvmape - 2$ (robot a aspoň jeden protivnik).

Podobne je možné použiť takéto obmedzenie zhora na zadanie maximálneho počtu súbojov pred dosiahnutim miesta, kde ma robot doraziť. Robot tu už musí mať za sebou nejaké súbojové skúseností.

Kombinácie Kombinácie vyššie uvedených cieľov zvyšujú zložitosť vytváraných algoritmov, čím spĺňajú základný cieľ, ktorým je výzva naprogramovať algoritmus. Preto sú kombinácie uvedených cieľov algoritmov nielen možné, ale dokonca žiadúce.

Každému robotovi môžeme prideliť špeciálny cieľ. Tým sme definovali definovali novú koncept, ktorý v sebe zahŕňa ciele, ako sú definované v POGAMUT-ovi (obmedzenie na počet nepriatelov, hladanie miesta) a ARESOVi (znemožnenie protivníkovi prežiť dlhšie). Tieto ciele sa ale nie vždy musia podariť splniť. Napríklad robot, ktorý bol cieľom, zahynie pri pokuse o prechod prepadliskom. Preto je nutné zaviesť takzvaný supercieľ (cieľ, ktorý je možné splniť kedykoľvek) na zakončenie simulácie. Tiež je možné odstrániť robota, ktorý nesplnil cieľ. To ale môže vyvolať reťazovú reakciu u ďalšieho robota, ktorý ale stále mal možnosť zviťaziť. Nebolo by spravodlivé, aby bol odstránený len preto, že zlyhala misia iného robota. Takto sa

dá odôvodniť existencia supercieľa. Nech supercieľom bude zničiť ostatných robotov na bojisku v zmysle zachovania principu "keď už som zlyhal, nech to aspoň niekto neprežije!". Tento cieľ je vždy dosiahnuteľný a obtiažnosť napísania algoritmu zodpovedá požadovanej úrovni. Bude na samotnom robotovi, kedy začne plniť tento cieľ, pretože napríklad nemôže vedieť, ze mu zomrela obeť.

Robot môže mať rôzne ciele a je na hráčovi, ako si ich definuje. Rozdielne ciele ale prirodzene vyžadujú od robota rozdielne chovanie. Robot, ktorý nemieni zničiť ostatných robotov, by mal mať možnosť len zastrašovacieho útoku, aby sa mu náhodou nestalo, že niekoho trafí. Podobne rozsah toho, čo vidí, by mal byť väčší, aby si mohol lepšie naplánovať trasu pohybu. Preto bol zvolený spôsob modifikovania vlastnosti.

Vlastnosti, ktoré robot má a ktoré sme už uviedli vyššie, sú útok na blízko, na diaľku, počet životov, definovanie, ako ďaleko robot vidí, počet striel, rýchlosť. Na to, aby si robot zisťoval informácie zo sveta a následne podľa nich vyhodnocoval stratégiu, si potrebuje robot niekde uchovávať informácie. Nazvime to **úložisko**. Každý robot bude mať vlastné úložisko, aby mohol súťažiť na úrovni virtuálneho sveta a nie prepisovať algoritmus nepriateľov. V tomto sa síce líši od ARESa. ale súčasne mu je dovolené pracovať na takej nízkej úrovni, ako je pamäť a zlyhať tak na inom mieste.
V tomto úložisku bude pre jednoduchosť akákoľvek informácia zaberať práve jedno miesto. Veľkost úložiska potom ovplyvňuje znalosti o svete a tým aj algoritmus, ktorý znalosti využíva. Preto by malo byt jednou z ďalších voliteľných vlastností veľkosť úložiska (pamäte)

Ďalšia vlastnosťou, o ktorej je možné uvažovať pre podporu algorimu, je rýchlosť robotov. Čím vyššia rýchlosť pohybu, tým viac je pravdepodobné, že robot nájde súpera, poprípade ho dobehne a zaútočí. Rýchlosť je ale potrebné obmedziť. Je povolený pohyb všetkými smermi a tak je pri veľkej rýchlosti robota nutne kontrolovať celú možnú trasu, aby sme zistili, kde sa robot zastaví, či tam nie je náhodou kolízia.

Väčším problémom je ale kontrola cesty strely. Strela sa na rozdiel od robota môže odrážať od stien a navyše musí byť aspoň tak rýchla ako robot, aby do nej po vystreleni nenarazil a neumrel z toho. Je pomerne náročné vypočítávať všetky odrazy striel, ktorých môže byť príliš mnoho. Celkové zobrazovanie simulácie sa tam môže dosť spomaliť. Rýchlosť bude teda pevne vymedzená. Ďalej je nutné zadať hornú hranicu pre každú vlastnosť, aby niektorá z vlastností nemohla byť aj blízko nekonečna. To sa môže stať, pretože robot, ktorý má maximálny počet životov, maximálnu rýchlosť. a pod. je najlepši, akého môžeme vytvoriť. Nastavenie vlastnosti menšej, ako je maximum, by bol zámerny handicap, čo je v rozpore s tým, že robot chce vyhrať. Potom nemá zmysel hovoriť o vlastnostiach ako počet životov, pretože by boli pevne dané. My však chceme, aby to boli vlastnosti voliteľné, pretože algoritmy sú na týchto vlastnostiach závislé.
To nás priviedlo k bodovaciemu systému. Najskôr si však musíme vysvetliť základne pravidla vlastností.
Viditeľnost bude obmedzená intervalom (0-180), čo sú stupne, pod akými ešte robot vidí. Stupne udávajú, pod akým uhlom môže robot vidieť doľava, rovnaký stupeň je potom použitý doprava. Tento koncept sa javí ako najprirodzenejší, nič však nebráni

asymetrickému pohľadu. Robot tak môže pokryť celých 360 stupňov.
Mapou je definovaná maximálna vzdialenosť v pixeloch, na akú je možné dovidieť. Zodpovedá to viditeľnosti, ako je známa v reálnom živote (hmla, čistá voda a pod.)

Uhol viditeľnosť je obmedzený určitým číslom, čo ju odlišuje od ostatných vlastností. Je možné k viditeľnosti pristupovať i tak, že si užívateľ presne definuje, v akom polomere bude robot vidieť, alebo, aby bol zvýhodnený hráč, ktorý vidí užší kužeľ, viditeľnosť sa bude zvyšovať s menším uhlom. Vlastnosti, ktoré si užívateľ bude môcť nastaviť, potom budú:

- veľkost úložiska
- životnosť
- útok strely
- životnosť strely
- útok zblízka
- počet striel
- uhol (0-180)

Z tohto výpočtu bola vynechaná vlastnosť rýchlosť. Je síce dôležitá, ale z časových dôvodov nebola použitá. Je predmetom ďalšieho rozšírenia.
Bodovaci systém znamená, že dostaneme číslo, ktoré pre nás predstavujú počet bodov. Tieto body prerozdelíme medzi jednotlivé vlastnosti.
Veľkosť bodov sa zdá byť obmedzená nepriamo, iba technickými parametrami (napr. veľkost RAM pamäte), ktorými nie je vhodne obťažovať čitateľa ani hráča, preto bude počet bodov obmedzený vhodne veľkým číslom, bolo vybrané číslo 1000 ako dostatočne veľké.

Užívateľ môže ale nechtiac prekročiť zadané sumárne čislo pri zadávaní vlastností. To je ale zaväzné pre všetkých robotov, aby ani jeden nebol zvýhodnený. Potom je potreba zadané hodnoty upraviť. Rozumným spôsobom sa zdá škálovanie hodnôt zo súčasného súčtu na deklarovaný, pretože zachováva istým spôsobom základnú myšlienku algoritmu (útok na blízko veľký, útok na ďaleko malý, pretože nebude zbytočne strieľať na diaľku).
Rovnako môže užívateľ podceniť rozdelenie bodov. Vtedy nastávaju dve možnosti, buď to bolo zámerné (zámerne slabý robot so stratégiou "aj tak na nezničíte"), alebo robot, ktorý bol vytvorený pri znalosti iného rozdelenia bodov. V tom prípade sa dá uplatniť tiež škálovanie. Bolo rozhodnuté použiť ho, keďže škálovanie zachováva smer algoritmu a pri zvýšení počtu bodov sa dá predpokladať, že sa algoritmus, spoliehajúci povedzme na malý útok na blízko, bude chovať podobne.

Algoritmus je potrebné napísať pomocou príkazov jazyka robotov. V záujme zachovania spravodlivosti by sa roboti mali chovať tak, aby ani jeden z nich nebol zvýhodnený, nevykonal väčšiu časť kódu ako ostatní roboti. To znamená, že v prípade, že roboti majú rovnaký algoritmus, tak po čase X všetci vykonávajú príkaz na riadku T $\forall X, T \in N$. Ideálne by bolo, ak by svoje algoritmy vykonávali súbežne a

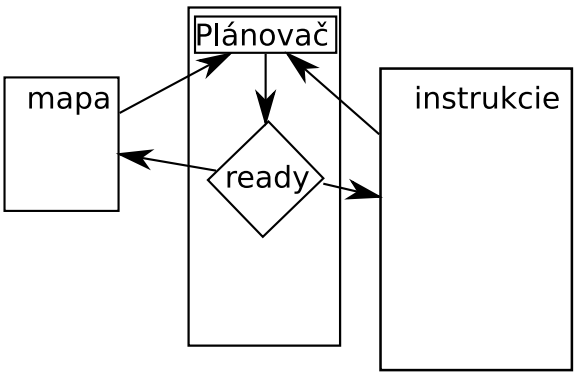
nemuseli by sme vykonaných časti kódu nijak kontorolovať. Pre každý algoritmus by sa dali využiť paralelne vlákna, takže paralelizácia by prebehla na najnižšej úrovni. Takéto rozhodnutie obsahuje nekoľko problémov. Pri jednojadrových procesoroch sa vlákna striedajú na základe prideleného časového kvanta a tak by sa algoritmy tak či tak realizovali sekvenčne. Tento spôsob má navyše tú nevýhodu, že čas, ktorý jednotlivé vlákno reálne dostane, závisí na výťaženosti procesora, softwarových a hardwarových prerušení a pod., takže nemôžeme zaručiť spravodlivosť.
Použitie multiprocesora princíp spravodlivosti mierne vylepší. Vzniká tu ale zásadný problém v tom, že mapa je len jediná a tak je potrebné naimplementovať ochrany proti prístupu na jedno pamätové miesto naraz. Tento koncept má však kvôli prerušeniam stále ešte malý problém so spravodlivosťou, aby každý robot vykonal približne rovnakú časť kódu algoritmu. Nechceme ale aplikáciu obmedziť len na použitie na počítačoch využívajúcich multiprocesory.
Z týchto dôvodov je najvhodnejšie zaistiť spravodlivosť na úrovni software. To znamená, že program sám bude kontrolovať poradie robotov a ich vykonávaný algoritmus. Doba, ktorá bude pridelená jednotlivým robotom, aby vykonali časť svojho programu a potom následne prenechali vykonávanie algoporitmu, nazvime časove kvantum. Je pre všetkych rovnaké. Proces, keď roboti toto kvantum využívajú, nazvime kolo. Kolá sa periodicky musia opakovať, aby mohol prebehnúť celý algoritmus. V opačnom pripade by sa simulácia zasekla už po jednom kole.
Vidíme, že je nutné implementovať mechanizmus, ktorý by kontroloval, aká časť kódu sa vykonala a následne prípadne odobral robotom aktivitu. Nazvime ho plánovač. Tento plánovač je kvôli spravodlivosti globálny, t.j. musí ho využívať každý robot.Plánovač je ale možné implementovať dvoma spôsobmi:

Obmedzenie kvantitou
: Plánovač tohoto typu nikdy nedovolí vykonávanie veľkých časti algoritmu naraz. Robot teda striktne vykoná práve jednu, rovnako veľkú časť svojho kódu a predá slovo ďalšim robotom. Tento spôsob prináša rýchlejšie striedanie robotov.

Obmedzenie časom
: Plánovač tohoto typu priradí každému robotovi čas, za ktorý môže vykonávať svoj program. To znamená, že na rozdiel od predchádzajúceho typu je možné postúpiť v programe omnoho ďalej v jednom kole.

Obrázok 2.3 naznačuje spôsob práce. Mapa sa v nejakom okamihu bude pýtať robota na aktualizovanie činnosti. Zavolá sa teda plánovač a ak je pripravený alebo nie, sa vykoná inštrukcia. Či je pripravený záleží na jeho vnútornej logike, ako bola popísaná v dvoch prípadoch. Robot by mal vykonávať svoj algorimus dovtedy, pokiaľ ho niekto externe nezničí. To znamená, že algoritmus sa musí opakovane vykonávať potenciálne do nekonečna. No nie je vhodné, aby na to dával pozor sám hráč, Jednak by to musel opakovane deklarovať u každého svojho naprogramovaného robota a jednak táto deklarácia je iba manuálny zapis, ktorý nemá vplyv na použitú stratégiu. Preto sa v práci dbá na to, aby v okamihu, keď by mal robot skončiť jeho algoritmus, spustil odznova v nekonečnom cykle.

Doteraz sme mlčky predpokladali, že každá časť algoritmu má rovnakú váhu. Tým, že niektoré časti algoritmu vyhlásime za ťažšie spraviteľné, potom vytvoríme



Obr. 2.3: Práca plánovača

novú inštanciu sveta. Tam sa nemení obsah, ale spôsob narábania s algoritmom. Hráč potom musí vymyslieť taký algoritmus, ktorý počíta s daným nastavením.
Potrebujeme ale vedieť, že to skutočne prispeje k atraktívnosti hľadania vhodnej stratégie. Rozoberme si teda, ako to bude vplývat na algoritmus, ak by jednotlivé časti trvali iný počet kôl. Potom by bol užívateľ nútený použiť taký algorimus, ktorý dlho trvajúce časti používa minimálne. Napriklad, ak by robotovi trvalo štyri kola na to, aby sa otočil, potom hráč sa pravdepodobne bude snažiť použiť minimálny počet otočení. Tým spôsobom sa môže vymýšlanie stratégií posunúť na hlbšiu úroveň, čo vyhovuje nároku na prácu.
Výsledne plánovače potom dostanú iné vlastnosti.

2.2.2 Možné prístupy k programovaniu virtuálneho sveta

Dôležitou časťou práce je predstaviť spôsob, akým bude užívateľ zapisovať vymyslený algoritmus. Uvažujeme nasledujúce jazyky:

Grafický jazyk

Pod grafickým jazykom rozumieme jednoduché grafické zobrazenie zápisu algoritmov. Tento spôsob sa najskôr využival pri výuke programovacich jazykov ako ľahký a zrozumiteľný spôsob písania algoritmu. Jednoduchým sledovaním šipiek v zápise sa dá zistiť, v akom stave sa program nachádza pri počiatočnych podmienkách. Implementovanie vlastného grafického jazyka, kde by boli príkazy len pre potrebu programátora, by bola dosť náročná. Jediný nástroj, ktorý splňoval základné predstavy, bol ale Microsoft Visual Language[5], ktorý je pre nekomerčné využitie síce zdarma, bohužiaľ je závislý na operačnom systéme, takže sa ukázal ako nevhodný

scriptovací jazyk LUA

LUA[7] je scriptovací jazyk, ktorý sa využíva práve na programovanie problémov umelej inteligencie, čo je aj náš prípad. Jeho výhodou je, že tento jazyk je jednoduchý, voľne širiteľný a prenositeľný.
LUA je však konštruovaná tak, že kompletne prevedie daný algoritmu, takže by sme nemali priamu kontrolu nad jednotlivými časťami kódu. Napríklad kontrola obsadeného úložisťa, ktoré môže robot používať, by sa skomplikovala.

Rovnako rozhodnutie, kedy algoritmus vykonal dosť príkazov a je nutne ho pozastaviť. Preto nebol použitý jazyk LUA.

Vlastný jazyk
Ďalšou možnosťou je vytvoriť vlastný jednorázový Domain Specific Language, ktorý bude použiteľný len na úlohy typu Codewars. Ponúka kompletný dohľad nad kódom, rozhodli sme sa preto pre tento koncept.

Vyšší programovací jazyk

2.2.3 Robotština

Algoritmus bude teda zapísaný vo vlastnom jazyku - robotštine. To naň kladie netriviálne nároky.

Minimálne schopnosti

Na začiatku je dôležité vedieť, čo všetko by mal jazyk ponúkať, aby sa pomocou neho dal napísať plnohodnotný algoritmus chovania robota v popísanom svete.
Robot musí pomocou popísaného algoritmu ovládať akcie, ktoré smie robiť, aby vôbec vo svete niečo vykonal. Jazyk preto bude obsahovať príkazy na

- pohyb
- čakanie (ako opak pohybu - ničnerobenie na istý počet kôl)
- útok
- otočenie sa
- získavanie informácií o objektoch z okolia - viditeľnosť

Tieto príkazy podmieňujú vznik operacií s objektami.

Pohyb znamená, že robot sa pohne daným smerom niekoľko krokov. Počet krokov je intuitívne vyjadrený celým číslom. V okamihu vykonávania príkazu nemusíme ešte presne vedieť, o koľko krokov sa robot pohne, preto je potrebné zaviesť premenné.

Premenné v zmysle ich definície uchovávajú určité informácie. Vzhľadom na náš virtuálny svet potrebujeme, aby uchovavali:

- celé číslo - *Integer*
- reálne čislo – spresnenie celého čísla kvôli aritmetickým operáciam, ako je ďalej uvedené - *real*
- objekt sveta *Object*
- pozíicia sveta ako dvojrozmerný vektor *Location*
- pole premenných - *int[3], real[a][3], location[4]*

- *null* pre oznámenie robotovi, že nemá uložený žiadny objekt
- *this* pre referenciu svojho vlastneho objektu

Operácie s číslami sa objavujú v súvislosti s premennými. Budú podporované iba niektoré základné operácie s reálnymi čislami, a to :

- sčítanie
- odčítanie
- násobenie
- delenie, pričom výsledok delenia celých čisel bude reálne číslo
- zvyšok po delení pre celé čisla = %

Ostatné aritmetické operácie nebudú podporované, nakoľko nie sú nevyhnutne potrebné. Aby sme docielili výsledok delenia v celom čisle, reálne a celé čísla sa navzájom automaticky konvertujú. Uživateľ toto nemusí špeciálne ošetrovať.

Operácie s objektami dávajú možnosť reagovat na svet, t.j. umožňujú zistiť:

- akým smerom je objekt otočený (vhodné pre robota, aby zistil, či nie je na muške) *getDirection (o)*
- akým smerom sa objekt pohybuje *isMoving(o)*
- či je to strela, stena alebo robot -*isMissile(m), isWall(o), isPlayer(p), isEnemy(o)*
- či bol nepriateľský robot zasiahnutý - *isHit(robot)*

Relačne operácie $>, <, =, \neq$ a ich kombinácia pre všetky typy premenných. Využijú sa napríklad pri zisťovaní najbližšieho objektu, atd. Pre objekty je nezmysel porovnavať, či je menší alebo väčši, preto u objektov je povolený iba relačný operátor = a $\neq$.

Podmienky výrazne prispievajú k eliminovaniu pre robot nepríjemných udalostí, ktoré môžu vo virtuálnom svete nastať (ak je tam mína, tak tam nesľapni). Viažu sa k nim ďalšie kódové slová TRUE (1) a FALSE (0).

Cykly zjednodušujú prácu pri vykonávaní rovnakých časti kódu. Podporované sú cykly s podmienkou na začiatku, na konci, pevný počet opakovaní a počet opakovaní v závislosti na premennej, s ktorou sa dá manipulovať.

Procedry a funkcie umožňujú sprehľadňovať a členiť kód. Prispievajú k lepšiemu kódovaniu algoritmov, využívaniu pamäte, ktorá sa po skončení procedúry uvoľní. Parametre k funkciám sa dajú predávať odkazom alebo referenciou. Definovanie predávania parametra referenciou je značené kódovým slovom *ref* pred premennou, ktorá je takto predávana. Poradie parametrov predávaných odkazom a hodnotou nehrá rolu. Parameter s preddefinovannou hodnotou nie je podporovaný

Definovanie cieľov a vlastností reprezentuje chovanie robotov.

Premenné sa ukladajú do úložiska. Ak už nie je voľné miesto v pamäti, bolo by nemilé, aby robot umrel. Rovnako je to nefér aj voči robotom, ktorý ho lovia. Namiesto toho bude premenná ukazovať na miesto v pamäti, ktorá je už obsadené. Robot tak pri nešetrnom zachádzaní s pamäťou môže poškodiť sám seba na úrovni programu, čo zahŕňa princíp škodenia, aký je v Aresovi. Algoritmus môže efektívne prepísat hodnotu, na ktorú sa spolieha, ale to zodpovedá tomu, že sa robot z nedostatku úložišťa zbláznil. Obzvlásť je to viditeľné, ak si prepísal premennú, na ktorú sa spolieha. Napríklad hodnotu TRUE bude zrazu FALSE.

Aby sa predišlo týmto nepríjemnostiam, je vhodné vedieť, kedy premenné vznikajú (v zmysle obsadzujú pamať) a kedy zanikajú (uvoľnujú pamät).
Pri deklarovaní premennej sa táto automaticky vytvorí, čo je normálne chovanie, známe takmer vo všetkých vyšších programovacích jazykoch. Pri volaní funkcie s parametrami sú tieto parametre kopírované, pokiaľ neboli definované kódovým slovom ref. To znamená, že sa vytvoria znova všetky premenné a pridá sa im hodnota, s akým je funkcia volaná. Návratova hodnota sa vytvorí v okamžiku volania *return*. V prípade, že ide o procedúru, návratová hodnota sa nevytvorí. Premenná sa môže dočasne vytvoriť aj v bloku označenom {}. Po ukončení bloku sa premenná v úložisku uvoľní. Preto by sa v nasledujúcom kóde (mimo bloku) mohla dať použiť. Premennú s rovnakým menom nie možné vzhľadom na implenentáciu vytvoriť ďalej s iným typom, než bola prvýkrát deklarovaná. Toto ale nie je závažný problém, keďže sa jedná iba o vytvorenie názvu pre premennú a rovnaké mená s rôznym typom iba mätú neskoršieho čitateľa.

Premenná obsadzuje miesto v úložisku. Vzhľadom na obmedzenie úložiska je namieste určiť, aké množstvo jednotiek úložiska jednotlivé premenné zaberajú. Premenné typu Object, Real a Integer budú zaberať jednu jednotku miesta, pretože nemá ďalej zmysel deliť ich na menšie časti. V reálnych jazykoch tomu tak pochopiteľne nie je, tu sme to obmedzili preto, aby obsadzovanie pamäte bolo jednoduché a pochopiteľné. Ďalej je u je typ Location. Ten má z princípu dve zložky. Preto v pamäti bude zaberat 3 jednotky, pre 3 premenné typu integer, ktoré sú jej zložkami a jednu pre seba. Podobne obsadzujú pamäť zložné prvky (polia). To znamená jedno pre samotnú premennú a potom súčet obsadenia všetkých jeho premenných.

Syntax robotštiny

Algoritmus je možné zapísať pomocou sekvencií príkazov. Pre správne pochopenie algoritmu je nutné definovať ich gramatiku. Tú zobrazuje obrazok ¡OBR-TODO¿ kde :

vlastnosť je jedno z:

- hipoints (počet životov)
- attack (veľkosť útoku)
- mAttack (veľkosť útoku strely)

- mHealth (dostrel/životnosť strely)
- angle (veľkosť uhla viditeľnosti)

nasledované číslom, ktoré definuje vlastnosť. V prípade, ak užívateľ počet bodov prekročí alebo podcení, vlastnosti sa budu škálovať.

Ciele algoritmu sa skladajú z:

- $visit|visitSequence([X,Y]|X)$
- $kill[a-zA-Z0-9]*$
- $killed <|>|<=|==|!=|>=$

Príkaz na robota je príkaz z množiny $step(X), wait(X), see(), eye(X), turn(X)X \in N, seeEnemy()$

Príkaz na informáciu je príkaz z množiny $(Locate(o), isWall(o), isPlayer(o), isEnemyo \in Objekt, seen[n]n \in N$

Príklady použitia

Nasledujúce príklady vysvetľujú, ako sa používajú jednotlivé príkazy, nemajú však za úlohu demonstrovať skutočný kód

```
robot R1 {
    attack = 20
    mAttack = 100
    hitpoint = 60
    memory  =10
    kill R2
    killed ! = 1
    integer l = 10;
    main()
    {
     for (integer i = 0; i < l; i++)
        {
            while (see()>0 && isPlayer[seen[0]])
                shoot(seen[0]);
            turn (30);
        }
    }
}
robot R2 {
    target ( Start[ R1 ], Start [R2])
    kill < 2
    Location was = [0,0]
    main()
    {
        Location l = [-1,-1];
```

```
        turn (getTarget());
        while (step(4) != 0)
        {
            if (was == l)
            {
                turn(15);
                continue;
                was = locate(this);
            }
            turn ( -direction(l));
        }
    }
```

2.3 Konkrétna realizácia jazyka

Máme popísaný jazyk, ktorý bude robotov ovládať. Ďalej uvedieme mechanizmus ovládania robota pomocou príkazov robotštiny.

2.3.1 Možné prístupy

Jedným z možných prístupov je interpretovať napísaný algoritmus v robotštine priamo. To znamená, vnútorne nereprezentovať jednotlivé časti algoritmu, ale opakovane prechádzať napísaný text. Nevýhodou tejto metódy je, že je veľmi pomalá. Pre každé kódové slovo treba rozlíšť, do ktorej kategórie patrí a uviesť robota do príslušného stavu, aby bol pripravený analyzovať ďalšie slovo. Napríklad pri kódovom slove *shoot* by si mal externe pamätať, že nasledujúce slovo, ktoré ma prijať, je napríklad celé číslo. Je to dosť časovo náročné a hrozí, že simulácia nebude plynulá. Navyše nie je vopred jasné, po akých častiach kódu sa roboti majú striedať, ako bolo povedané pri plánovaní. Je možné napríklad striedať sa, keď interpret prejde určitý počet slov. To ale nie je spravodlivé. Napriklad príkaz a=b, kde a,b su veľmi veľké polia, by logicky mal trvať dlhšie, než keby boli typu integer. Preto tento prístup nebol použitý.

Ďalšou možnosťou je použiť kompiler nejakého skutočného programovacieho jazyka. To znamená upraviť jazyk tak, aby sme ho prepísali do podoby vhodnej pre robotštinu. Možno tu definovat interface, s ktorým bude užívateľom generovaný program komunikovať. Príkladom je použitie programovacieho jazyka C# a následne vygenerovaný medzikód MSIL. MSIL ale podobne ako strojový kód alokuje reálnu pamäť pri vytváraní premenných zo skutočnej pamäte. Samotná aplikácia tak stráca kontrolou nad tým, kde je aká premenná uložená a kedy musí prepisovať z nedostatku voľnej pamäte. Preto bol tento koncept tiež odmietnutý.
Metódu rozkladu kódu na menšie časti nemusíme opustiť celkom. Aby sme mali úplnú kontrolu nad chovaním algoritmu, je nutné okrem vytvorenia vlastného medzikódu vytvorit aj vlastnú *virtual machine*, ktorá bude jednotlivé preložené časti v medzikóde interpretovať. Inšpiráciou pre takúto metódu bola Java virtual machine [6]. Súčasťou takéhoto virtuálneho stroja by mal byť potom aj plánovač a vlastná správa pamäte. To je presne to, čo nám vyhovuje, preto bolo implementované v

robotštine.

Niekedy sa ale preklad do medzikódu vypustí a generuje sa priamo do cieľového jazyka, jazyka, v ktorom bude implementovaný program. To znamena opäť obtiažnu správu pamäte robota.

2.3.2 Práca s jazykom

Rozhodli sme sa, že bude implementovaný vlastný jazyk s prekladom do medzikódu a interpretovaný vlastným virtuálnym strojom. Na to potrebujeme definovať, ako medzikód vyzerá, ako komunikuje s robotmi a ako sa vykonáva.

Generovaný medzikód

Algoritmus je zapísaný pomocou voľného textu, je teda nutné použiť parsovacie nástroje. Na základe predchádzajúcich skúseností boli zvolené nástroje Bison a Flex [15].
Najskôr bolo nutné si premyslieť formát medzikódovych inštrukcií. Jedným z najčastejších foriem je troj-dresný, štvor-dresný alebo formát zásobníkového čitača. Trojadresný a štvoradresný znamenajú, že medzikódové inštrukcie operujú nad dvoma alebo troma operandami a majú ešte ďalší ukazateľ na to, kde ukladá výsledok. My však budeme potrebovať aj premenný počet parametrov, preto bol použitý posledný spôsob. Inštrukcia medzikódu si tu zo zásobníka vyberie toľko operandov, koľko bude potrebovať a do zásobníka zapíše hodnotu operácie, resp. nevyberie alebo nezapíše vôbec nič.

Jednou z otázok je, čo sa do tohto zásobnika ukladá. Je možné pozerať sa na tento zásobník ako na pole prvkov neznámeho typu, kde inštrukcie predpokladajú správny typ pre svoju funkčnosť. Toto je zaistené už pri pokuse o preklad. Problém nastáva v okamihu, keď je úložisko robota už plné. Potom, ak sa program nemá zastaviť, musíme na vrchol zasobníka niečo pridať. Preťaženie pamäte sa bude prejavovať tak, že robot bude prepisovať svoje už priradené premenné. To potom znamená ale analýzu úložiska na existenciu premennej typu (Objekt, Integer, Real, Location), ktorá zodpovedá požiadavke nasledujúcej inštrukcie. To je ale dosť neefektívne. Tento spôsob má problém aj pri premenných zloženého typu. Tento nepríjemný efekt môžeme odstrániť implementovaním polymorfného zásobníka. To znamená, že sa zasobník bude chovať ku všetkým prvkom ako k pôvodnému abstraktnému prvku a abstraktný prvok bude podporovať operácie pre uloženie a vydanie podporovaných prvkov. Ak bude mať robot preplnené úložisko, je možné na zasobník pridať akýkoľvek prvok z úložiska a následne operácie budú bezpečné. Problémom je, že sa nebudú prepisovať už obsadené premenné. To by nám ale príliš nevadilo. Zápis do premennej obvykle znamená jej prepis na inú hodnotu, takže pri načitaní premennej by robot dostal inú hodnotu ako očakávanú, čo je tiež spôsob prepisu. Problémom tejto metódy je, že počas života algoritmu bude v úložisku často rôzny počet rôznych premenných (pri odchode z funkcie sa niektoré premenné typu Integer zničia a pri zavolaní následnej funkcie sa na uložisku objaví premenná typu Integer). Takto sa bude musieť

úložisko dynamicky meniť na úrovni programu. To znamená častú alokáciu a dealokáciu premenných za behu algoritmu. To je dosť neefektívne obsadzovanie pamäte, keďže vieme presne, aké veľké úložisko bude robot mať a teda by sme si ho mohli predgenerovať.

Preto bol navrhnutý nasledovný spôsob : Každá premenná bude obsahovať všetky základné typy a v prípade, že ide o pole, bude obsahovať odkaz na ďalšie takéto premenné. Tým pádom sa dá celé úložisko predgenerovať a na zásobník ukladať takúto premennú. Pri nesprávnom zápise sa informácia niekde zapíše a bude prístupná opäť len pri nesprávnom použití. Tento spôsob má tú nevýhodu, že nám prakticky znásobuje pamäťt. Výsledný efekt je ale uspokojivý.

Zásobník je na pevno obmedzený maximálnou veľkosťou 10000 prvkov, ktorá je považovaná za dostatočne veľké čislo.

Samotný medzikód bude mať formát spojového zoznamu. Tento formát sa vhodne konštruuje zo syntaktického stromu, ktorý je výsledkom syntaktickej analýzy prevádzanej pomocou nástrojov Flex a Bison [15].

V tejto fázi treba rozhodnúť, aké inštrukcie bude medzikód používať. Tie vyplývajú zo syntaxe robotštiny a ďalej sa s nimi budeme zaoberať až v sekcií o vykonávaní medzikódu.

Preklad jazyka

Sekcia popisuje, aké štrukúry sú nutné pre generovanie a uloženie jazyka. Vynecháme spôsob definovania vlastnosti a cieľov, pretože to sa priamo jazyka netýka.

Samotný medzikód sa skladá z inštrukcií. Inštrukcie sa zoskupia do jedného zoznamu. Ponuka sa koncept polymorfného poľa. Každá inštrukcia potom bude objekt odvodený od základného objektu

Jazyk podporuje premenné a tak je nutné tieto premenné ukladať. Premenné sú dvojakého typu, globálne a lokálne. Navyše je možné definovať funkcie. Premenná s rovnakým menom sa môže vyskytovať v rôznych funkciách môže byť navyše rôzneho typu. To vadí najviac, pretože premenná musí byť práve jedného typu. Preto ich treba nejak odlíšiť. Názvy premenných budeme meniť nasledujúco: v okamihu, keď je definovaná funkcia F(), všetky premenné vytvorené v tejto funkcií, sa budú ukladať pod menom F#premenná. Znak # bol zvolený, pretože sa nevyskytuje v jazyku. Oddeľuje názov funkcie/procedúry, kde bola definovaná premenná a tak je možné ľahko spätne rekonštruovať jej názov.
Ťažkosti nastáva s premennou, ktorá bola definovaná vnútri nejakého bloku. Potom sme ju uložili pod týmto menom a s už daným typom. Dalšia takáto premenná definovaná v rovnakej fukncii, ale v inom bloku, bude totiž reprezentovaná rozvnakým reťazcom.

Kvôli zisťovaniu, ako a kde sa v algoritme používajú premenné, je nutné vyhladávať podľa reťazca. Minimálne pri parsovani by bolo vhodné nájsť takú štrukúru,

ktorá nie je z najpomalším (lepšia ako lineárne prechádzanie zoznamu). Ponuka sa hash-mapa, ktorú štandartne obsahuje STL (Standard Template Library). Pretože sa predpokladalo (v záchvate paniky), že bude nutné premenné vyhľadávať aj počas vykonávania programu, bol zvolený Burst trie[4]. Toto rozhodnutie sa žiaľ neskôr ukázalo ako predčasné z dôvodov, ktoré budú jasné pri objasneni práce interpreta medzikódu. Z dôvodu fungovania tohoto kusu kódu bolo ale ponechané.

Podla syntaxe môžeme jednotlivé typy kombinovať do zložitejších polí. Je nutné si uchovávať štruktúry týchto nových typov, keďže premenné týchto typov môžu neskôr v algoritme nadobudnúť platnosť. Týchto typov nebude veľa, keďže užívateľ nemá právo vytvárať vlastné štruktúry, preto ani nemá zmysel robiť hlbšiu analýzu alebo optimalizovať vyhľadávanie. Typy potom budú uložené v jednoduchom poli.

Otázka je, ako reprezentovať samotný typ. Jazyk rozoznáva len niekoľko typov premenných a typ zložený z nich. Potom je namieste reprezentovať typ ako spojitý zoznam.

Samotné inštrukcie budú potomkovia jednej abstraktnej triedy Instruction. To znamená, že na simulovanie chodu jednoduchého programu nám stačí interpretovať každú jednu inštrukciu, až kým sa neminú inštrukcie. Zložitejšie programy však používajú cykly, podmienky a podobne, ktoré menia poradie práve vykonávanej inštrukcie. Preto je nutné zaviesť *Program Counter (PC)* , ktorý bude ukazovať na práve vykonavanú inštrukciu.

Súčasťou medzikódu sú aj volané funkcie a procedúry. Tie pozostávajú rovnako z inštrukcií, teda ich môžeme priradiť hneď za vygenerovanou hlavnou procedúrou main. Funkcie a procedúry teda tiež menia poradie vykonávanych inštrukcií. Na rozdiel od príkazov je nutné si zapamätať, odkiaľ bol spravený skok, aby bolo možné sa vrátiť a pokračovať. Teda namiesto jedno PC budeme potrebovať opäť zásobnik PCciek.

2.3.3 Interpret medzikódu

V tejto časti budeme zaoberať problémom, na aké časti má byť algoritmus rozkúskovaný, aby plánovač, ako bol popisaný, korektne vykonal príkazy a vrátil očakávaný výsledok. Prioritou je teda rozdeliť kód na čo najmenšie časti. Tu sú nasledujúce robotické inštrukcie :

InstructionCreate pridá premennej priestor v úložisku

InstructionLoadVariable na vrchol zasobníka s hodnotami pridá hodnotu premennej

InstructionLoadElement zo zásobnika vyberie $n \in N$ a premennú typu pole. Na vrchol zásobníka pridá n-tý element tohoto poľa

InstructionConversionToInt zmení premennú z typu real na integer. Načitaná reálna reprezentácia celého čisla bude dočasne umiestnená v pamäti, pamäť sa dočasne zaplní o jedno miesto naviac

InstructionConversionToReal zmenu premennú typu Integer na Real

InstructionDuplicate zduplikujú hodnotu na zásobníku

InstructionStoreRef vezme premennú zo zásobníka a uloží odkaz na ňu do premennej na vrchole zásobníka

InstructionStoreInteger,InstructionStoreReal, InstructionStoreObject uložia načitanú prislušnú hodnotu do premennej na vrchole zásobníka.

InstructionCall spôsobí uloženie PC a založenie noveho

InstructionPop odoberie hodnotu na zásobníku

InstructionMustJump skočí na inú inštrukciu v rámci aktuálnej funkcie.

InstructionJump podľa vrchola zásobníka zmení vykonávanú inštrukciu

InstructionBreak podobne ako MustJump, ale v okamihu generovania kódu nie je známe, kam má inštrukcia skočit

InstructionContinue podobne ako break, ale v dobre generovanom preklade je jasne, kam skoči, už pri generovaní medzikódu

InstructionReturn vytvorí návratovú hodnotu

InstructionRestore obnoví stav, aký bol pred volanim funkcie

InstructionRemoveTemp uvoľní v úložisku poslednú premennú deklarovanú ako temp (obvykle výsledok operacie)

Operacie vezmú dva prvky na vrchol zásobníka vložia výsledok operácie

aritmetické operácie pre Integer a Real

binárne a logické OR a AND

InstructionNot vezme celé čislo zo zásobníka a ak je to 0, vloži tam 1, inak0

inštrukcie rovnosti a nerovnosti pre objekt vezme dva prvky zo zásobníka a vloži tam výsledok operácie. Objekty nemajú relačne operátory na nerovnosť, preto sú uvedené osobitne

Relačné operácie pre Integer a Real vezmú dva prvky zo zásobníka a vložia na vrchol zásobníka výsledok porovnánia (0 = npravdivé tvrdenie, 1 inak)

InstructionBegin nastaví príznak, že začal novy blok, kvôli premenným a ich neskoršim dealokáciam

InstructionEndBlock vyčistí pamäť od premenných definovaných v tomto bloku a zru)sňi prňiznak posledného bloku

InstructionSee naplní robotove zorné pole objektami, ktoré vidí a uloží na vrchol zásobníka počet viditeľných objektov

InstructionEye zoberie zo zásobnika celé číslo X a uloži objekt, ktorý bol videný robotom ako X-tý v poradí. Ak žiaden objekt nevidí, uloží fiktívny objekt reprezentujúci NULL

InstructionFetchState uloží na vrchol zásobníka výsledok poslednej akcie, ktorú robot vykonával (pohyb, streľba)

InstructionStep vezme zo zásobníka cele číslo a robot sa pohne prislušným smerom

InstructionWait vezme celé čislo X a nastaví robota do "čakacieho" režimu po dobu X kol

InstructionShootAngle vezme zo zásobnika celé čislo prinúti robota vystreliť v danom smere

InstructionTurn vezme zo zásobnika celé číslo a otočí sa podľa neho (kladne doprava)

InstructionTurnR robot sa otočí doprava o 90^o

InstructionTurnL robot sa otočí doľava o 90^o

InstructionHit zoberie zo zásobníka objekt a uloží na vrchol zásobníka jeho aktuálnu životnosť

InstructionLocate vyberie zo zásobnika objekt a uloží pozíciu tohoto objektu vo svete na vrchol zásobníka, ak je tento objekt robotom viditelný, inak uloží $[-1, 1]$

InstructionIsXXX kde XXX je žiadosť o detail objektu (isMoving atď.). Vyberie zo zásobníka objekt a uloží 0/1 v závislosti na vlastnosti objektu v premennej zobratej zo zásobníka. XXX môže byť missille, wall, player...

InstructionTarget dá na zásobník cieľové miesto robota, ak bolo nejaké definované. Inak uloží [-1, -1]

InstructionSaveVariable uloží premennú mimo zásobnik. Táto inštrukcia vznikla kvôli priraďovaniu zložených typov. Na vrchole zásobníka bude zložená premenná. Na to, aby sa dve zložené premenné priradili, potrebujú sa rozvinúť až na úroveň jednoduchých prvkov. Príkladom nech priraďujeme A = B, kde A, B sú typu integer[6][2]. A sa rozloží na 12 integerov v poradí integer[0][0], integer[0][1] atď., postupnosť takýcho príkazov vieme zabezpečiť už počas prekladu. Aby sa B správne priradilo A, potrebujeme B vhodne rozmiestniť medzi načítané hodnoty A. K tomu využijeme premennú, ktorú sme si odložili bokom, každý príslušný prvok generujeme z uloženej premennej znova cez všetky dimenzie.

InstructionLoadVariable načíta premennú uloženú mimo, na vrchol zásobníka

InstructionDirection zoberie zo zásobníka premennú a dá na vrchol zásobníka smer k pozícií (typ Location)

InstructionRandom uloží na zásobník náhodne celé čislo v rozsahu 1-10 000. Toto číslo bolo zvolené ako dostatočne veľké.

V prípade niektorých inštrukcií, ako napríklad InstructionBreak, nie je v čase analýzy zrejmé, kde ma algoritmus skočiť, preto na záver vygenerovania medzikódu je nutné tento kód ešte raz prejsť a doplniť chýbajúce informácie.

Informácie o premenných sú uložené v trie. Táto štruktúra už bude obsahovať všetky deklarované premenné, preto jej súčasťou je aj štruktúra, ktorú bude používať algoritmus počas behu pri čitaní hodnoty premennej. Keďže premenná je známa pod svojim menom a navyše je povolené používať rekurzívne funkcie, je možné, že rovnaká premenná bude vytvorená v rôznych hĺbkach, ale pod rovnakým menom. Táto štruktúra bude obsahovat nielen premennú známu pod konkrétnym menom, ale pole premenných, ktoré boli vytvorené v rôznych hĺbkach.
Vzhľadom na to, že všetky ostatné atribúty, ktoré premennej prislúchajú (napríklad, či bola definovaná lokálne alebo globálne), už zostávaju rovnaké, stači nám ako štruktúra opäť zásobník. Pri uvoľnovaní premennej z robotovho úložiska sa potom len odstráni vrchný prvok a požiada úložisko o uvoľnenie.

Čo sa týka obsadzovania pamäte, pri reálnych pamätiach sa dbá na to, aby jednotlivé dáta boli vedľa seba. V našom prípade nám ale nezaleží na tom, kde sú premenné uložené, nijak to neovplyvňuje výkonnosť robotov. Teda v pamäti si stači udržiavať informácie o poslednom obsadenom prvku. Nasledujúce voľné miesto sa nájde tak, že sa lineárne budú prechádzat ďalšie miesta a keď dosiahneme koniec, kontrolujeme od začiatku. Úložisko je preplnené, keď kontrolované miesto bude rovnaké ako to, odkiaľ sme začínali.

Ďalej medzikód potrebuje prístup k úložisku. V čase vykonávania tohoto kódu sa budú premenné vytvárať, to znamená uberať voľné miesto v robotej pamäti.

Kapitola 3

Implementácia

Táto kapitola pojednáva o konkrétnych implementačných technikách, ktoré vyplývajú zo záverov analýzy.
Aplikácia sa skladá z troch hlavných častí, ktoré sa dajú používať samostatne. Sú to:

- Mapa virtuálného sveta
- Preklad napísaného algoritmu do medzikódu
- Systém okien
- Modul pre generovanie diskrétného bludiska

Architektúra celej aplikácie je popísaná na obrázku 3.1. Jednotlivé časti ďalej vysvetlíme.

3.1 Virtuálny svet

Virtuálny svet pozostáva z objektov. Každý konkrétny objekt je potomkom abstraktnej triedy Object. Navzájom riešia výsledky kolízií, výsledky príkazov na zistenie pozície a pod. Mapa slúži len ako zastrešujúca štruktúra vyhľadávajúca kolízie a riadiaca vykresľovanie.
Súčasťou objektu je aj spôsob, akým sa objekt vykresľuje na plochu, čím sa zaoberá trieda ImageSkinWorker.
Obrázky reprezentujúce jednotlivé objekty sú najskôr načitané triedou Skin. Počet obrázkov je presne N, kde N je počet stavov, aké môže objekt dosiahnúť. Stavy sú nasledujúce:

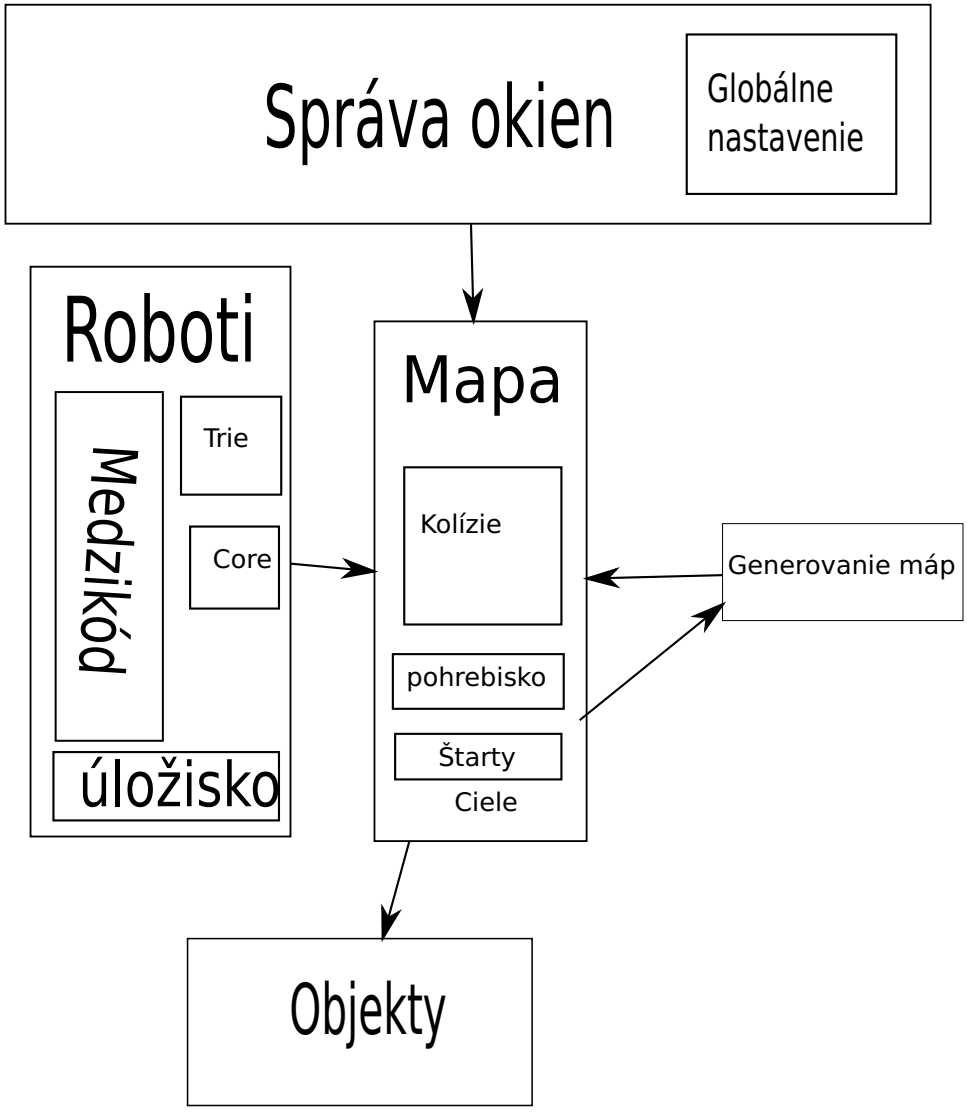
Defaultne
Stav, ktorý má objekt, keď sa nič nedeje.

- čakajúci objekt
- spiaci objekt. Tento stav sa nastaví v okamžiku, ked je objekt príliš dlho v defaultnom stave

Trvale

- pohybujúci sa

Obr. 3.1: Architektura aplikácie



- čakajúci

Dočasne
- zasiahnutý
- útočiaci

Niektoré objekty, ako je napríklad stena, nepotrebujú toľko stavov, preto rôzne objekty majú svoju vlastnú triedu, odvodenú od Skin. Trieda Skin berie do úvahy špeciálne požiadavky jednotlivých typov objektov. Každý nahrávaný obrázok ale obsahuje niekoľko častí - každý z obrázkov predstavuje objekt v inej fázi. Smerom doľava sa časť obrázka mení kvôli animácií, smerom dole sa vykresľovaný obrázok mení vzhľadom na to, ako má byť objekt otočený. Potrebujeme pritom vedieť veľkosť obrázka, ktorý sa má vykresľovať. Veľkosť obrázka nemusí súhlasiť s veľkosťou obrázka, ktorý spôsobí kolíziu. Preto je v každom adresári, odkiaľ sa nahrávajú obrázky, ešte súbor config. Tento súbor hovorí ako je vykresľovaný obrazok veľký, odkiaľ začína kolízny obdĺžnik a aké má rozmery.
Trida ImageSkinWorker potom narába s nahranými obrázkami. V jej vnútornej interpretácií sa nachádza ešte aj minizásobník, ktorý hovorí, aký zo stavov "trvalý", "dočasný", "default" je aktívny. Dočasný stav ma vždy vyššiu prioritu ako trvalý a trvalý vyššiu ako default. Stavy sa nastavujú pomocou metódy *SetState(State s, whichState w)*
Objekt je základný prvok sveta, všetky ďalšie typy vzniknú len jeho dedením.
Pohyb sa v objekte rieši nastavením počtu pixelov, ktoré má prejsť. V okamžiku, keď je nastavený nenulový počet pixelov, objekt sa pri volaní metódy *move()* pohne svojou rýchlosťou žiadaným smerom a odčíta sa príslušný počet pixelov. Po skončení pohybu sa zavolá metóda *endMove()* , ktorá odstráni permanentný pohybujúci sa stav. U robota je situácia trochu zložitejšia, pretože potrebujeme naviac vedieť, či už náhodou nestúpil na poličko, ktoré bolo deklarované ako cieľové. Preto sa v priebehu vykonávania pohybu kontroluje aj táto podmienka.
Kolízie medzi objektami sa riešia pomocou virtuálnych metód *hit(Object *)* a *hitted(Object *)*. Problémom je strela, ktorá v čase, keď do nej niečo narazí, sa musí chovať ako útočník. Ak by ale útočníkom bola opäť strela, výsledkom by bolo zacyklenie. K takej situácií ale nikdy nesmie dôjsť, čo ošetruje štruktúra zastrešujúca objekty.
Objekt, ktorý pôsobením útoku zomrie, registruje, kto smrť priamo spôsobil. Je to zabezpečené premennou "Owner" vlastníka objektu. V pripade, že je obeťou robot, tak má možnosť poslať správu útočníkovi (vlastníkovi strely alebo tela robota). Tak sa robot dozvie, že niekoho zabil a zistí, či sa tým nespnila podmienka pre ukončenie simulácie. Objekt sa následne pridá do štruktúry, ktorá zhŕňa mŕtve objekty pre neskoršiu dealokáciu.
Robot na rozdiel od obyčajného objektu má aj ďalšie metódy. Menovite je to *shoot()*, *see()*
Strieľanie robota je jednoduché, pokiaľ je známy smer, v ktorom ma robot strielať. Znamená to iba pridanie objektu na miesto, kde nebude kolidovať s robotom, ktorý ju vypustil.
Robotové videnie je podobne realizované pomocou triedy Seer. K objektom, ktoré boli ohodnotené ako viditeľné, je možno pristupovať pomocou metódy *getObject(int index)*

Vzájomné vplyvy objektov sa riešia len medzi objektami samotnými. Jediné, čo reba zistiť, je, že ktorý objekt má vplývat na ktorý. Takouto zastrešujúcou štruktúrou je mapa.
Mapa je riešená ako dvojrozmerné pole oblasti (obdĺžnikov) kvôli kolíziam, ako sme už spomínali v Kolíziách. Oblasť je ohraničená počiatočnými súradnicami, ktoré sú pravým horným rohom jej obdĺžnika a jeho výškou a šírkou.
Každá oblasť potom obsahuje zoznam objektov, ktoré do nej patria. Pozícia týchto objektov nie je nijak obmedzovaná, takže sa môžu vyskytovať kdekoľvek v oblasti. Pozícia objektov v mape je určená ich ľavým horným rohom. Pri zisťovaní koíizií to prináša nepríjemný jav, pretože je nutné kontrolovať všetky oblasti, kde by sa daný objekt mohol vyskytovať. Mapa ďalej obsahuje štartovacie políčka - užívateľom definované významné políčka, ktoré sa dajú neskôr definovať ako cieľ.

Mapa počíta iba s pozíciami objektov, preto o nich nič viac nepotrebuje vedieť. Každý objekt zobraziteľný na mape je teda potomkom abstraktnej triedy Object. Ten si musí pamätať svoju presnú pozíciu kvôli neskorším počitaním kolízií// Objekt sa pridá do mapy pomocou metódy *add*. Táto metóda skontroluje, či pridávaný objekt už nekoliduje s iným v mieste a ak áno, nepridá ho.

Pridávané objekty na mape sú odkazy na vytvorený objekt z dôvodu využitia dedičnosti a virtuálnych funkcií. Tieto objekty môžu byť vytvárané samotnou mapou pri nahrávaní sveta alebo užívateľom Potom pri dealokovaní mapy mapa samozrejme nemí možnosť zistiť, či tieto objekty boli skutočne dealokované, alebo či si ich inštancia vytvorila samostatne. Preto ticho predpokladá druhú možnosť a pri skončení na všetky prvky, ktoré zostali na mape, sa použije delete. Ak bol prvkom mapy objekt, s ktorým programátor ešte počita, musí ho najskôr z mapy odobrať.
Program podporuje uloženie a opätovné nahranie mapy. Keďže mapa tieto objekty iba zobrazuje, postačí pre každý objekt zistiť, čo to vlastne je. To kladie nárok na jednotlivé objekty, ktoré si musia pamätať unikátne čislo označujúce tento typ. Mapa potom pri načítani sveta podľa typu vytvorí objekt. Ďalšimi položkami, ktoré je nutné uchovať a ktoré nie sú objekty, sú štartovné políčka a miesta cieľov. Tieto sa ukladajú osobitne. Mapa musí dbať na to, aby objekty, ktoré obsahuje, neprešli mimo vykresľovanej plochy. Aby sme nemuseli kontrolovať objekt pri každom pohybe a okrem kontroly kolízií kontrolovať aj prípustnosť pozície, sú ku každej nahrávanej mape pridané steny okolo celej mapy. Po vytvorení prostredia (mapy, sveta) sa v mape nenáchadzaju roboti, pretože nie podporovaný žiadny ďalší mechanizmus na ovládanie tela robota (nic by sa nedialo).
Mapu je možné okamžite vykresliť a objekty neskôr. Mapa ďalej nijak nevyhodnocuje výsledok kolízie, slúži iba ako nástroj na ich detekciu. Detekuje kolíziu iba pre objekty, ktoré sa pohybujú, t.j pre každý pohybujúci sa objekt prejde lineárne všetky objekty v okolí, či nenastala kolízia. V okamžiku, keď sa objekty skolidujú, mapa vyberie najbližši objekt, s ktorým sa objekt skolidoval. Pod kolidáciou rozumiene prekrytie obrazov dvoch objektov. Kedže máme kombináciu dvoch objektov, ktoré spolu nekolidujú, je zaistiť riešenie aj pre takúto kombináciu. Na implementačnej úrovni to znamená, že sa buď:

- zavolá aj tak riešenie kolízií u objektu. Až objekt, ktorý bol zasiahnutý, spozná,

že ku kolízií vlastne vôbec nedošlo. To má nevýhodu, že objekt, ktorý sme kontrolovali na kolíziu, zostane na mieste, ktorý mu pohyb určil. Keďže ale kontrolujeme iba najblžši kolízny prvok, môže sa stať, že aj po vyriešeni kolízie bude prvok stále kolidovať

- nedôjde k volaniu riešenia kolízií medzi objektami. Objekt nebude uvažovaný pre kolíziu. Toto riešenie bolo zvolené, keďže má najlepšie výsledky. Pre kontrolu toho, či objekt má alebo nemá by uvažovaný pre koííziu, bol daný príznak Solid. Objekt, ktorého logicky súčet (or) zasiahnuteľnosti s uvažovaným objektom má príznak solid (aspoň jeden z objektov má solid = true), je uvažovaný pre kolíziu

Mapa ako vykresľovacia štruktúra musí poznať, kedy by bolo vhodné objekt vykresliť a kedy naopak nie. Výrazne to šetri procesorový čas. Preto bola u každého objektu implementovaná metóda *changed()*

3.1.1 Interpret a jazyk

Ako bolo zmienené, jazyk sa generuje pomocou parsovacích nástrojov Bison a Flex. Tieto nástroje umožňujú veľmi ľahko meniť syntax jazyka a sú ľahko udržovateľné.
Robotov môže byť niekoľko a vďaka cieľom, ako je KILL, je možné sa na ne odkazovať pomocou mena. Preto bolo nutné vytvoriť zastrešujúcu štruktúru. Tou je trieda Robots. Tá potom musí obsahovať:

- počet bodov, ako bol defaultne definovaný. Toto defaultne nastavenie sa tam potom dá jednoducho priradzovať vytvoreným robotom.
- pole generovaných robotov
- informácie o práve vytváranom robotovi
- zoznam nevyriešených požiadaviek na zabitie robota
- zoznam nevyriešených požiadaviek na miesto na mape

Novovytvorený robot sa skladá z týchto časti:

- názov robota pre definovanie cieľa KILL
- použitý plánovač
- prázdnu štruktúru na definované typy
- prázdnu štruktúru pre medzikód
- prázdne informácie o nájdenych chybách
- trie premenné
- pomocnú štruktúru pre spracovanie medzikódu

- chybové kódy
- štruktúra pre prácu s medzikódom (Core)

Algoritmus bude zapísany v textovom súbore. Z neho sa pomocou parsovacích nástrojov vygeneruje kód, ako je popísaný v kapitole Interpret. V prípade, že nastala chyba, bude potom tato zapísana pomocou stringu. Výsledny medzikód je potom zapísaný v XML-súbore, ktorého názov je kombinácia názvu robota a vstupného súboru. Napríklad technológia XML bola vybraná ako jedna dobre zo zrozumiteľných dostupných zobrazení.

Po dogenerovaní všetkých inštrukcií sa na záver pridá ešte jedna inštrukcia, ktorá spôsobí opakovanie celého programu. (instructionMustJump). Po správnom vygenerovaní je robot schopný vykonávať program pomocou metódy *execute()* alebo *action()*. Tu sa prejaví vplyv plánovača. Robot ma vlastnú inštanciu plánovača, každá inštrukcia si vnútorne drži čislo skupiny, do ktorej patrí. Jednolivé názvy skupín sa volaju intuitívne podla inštrukcii. Metóda '*action()* volá plánovač. V okamžiku, keď plánovač povolí vykonanie inštrukcie, do plánovača sa zanesie, ako moc je inštrukcia hodnotená a následne je táto vykonaná.,
Vykonávanie inštrukcíi pozostáva z volania metódy *execute(Core *)* , kde trieda Core poskytuje prostriedky pre interpretovanie medzikódu. Trieda Core tiež obsahuje telo robota, ktorým sa bude hýbat po mape úložiska. Aj strela sa teda bude realizovať vyvolaním príslušnej metódy robota.
Pri interpretácii algoritmu ešte ale musíme vedieť, kedy robot dosiahol splnenie všetkých cieľov alebo či zostal vo svete sám. V pripade, keď zostal sám a cieľom nebolo len zničenie ostatných,vráti metóda *action()* hodnotu false, ak hráč umrel, inak vracia true. Je vhodné ale sučasne kontrolovať splnenie podmienok. Metóda *action* nemôže vrátit viac premenných naraz, preto ma naviac parameter odovzdávany referenciou.

3.1.2 Generovanie bludiska

Generovanie bludiska Z dôvodov, vysvetliteľných v analýze, sa vygenerujú len základné steny. Vygenerujú sa z dôvodov vysvetlených v analýze len zakladne steny. Teraz potrebujeme tieto pevné steny z diskrétneho bludiska vhodne preniest do nášho sveta. Aby sme zachovali pomer rozostavenia, ako je vo vygenerovanom bludisku, steny sa rozmiestnia rovnomerne.

3.2 Správa okien

Ako grafická knižnica bola zvolená SDL[1], ktorá bola vyvinutá pre rýchle zobrazovanie a je jednou z najčastejšie používaných knižníc pri tvorbe 2D hier. Aplikácia bude používať minimálny užívateľský vstup z klávesnice. Jediný vstup rozsiahlešieho charakteru je napísanie samotného algoritmu. Pre uživateľa v Unixe, zvyknutého na svoj obľúbený editor, makra a zkratky, bola ponechaná možnosť vytvoriiťsi v ňom aj algoritmus.
Vzhľadom na to, že zvyšok okien, ktoré ma aplikácia zvládať, má tiež minimálne požiadavky z hľadiska užívateľovho vstupu, bolo v práci použité vlastné rozhranie

správ okien a nebolo nutné prikompilovať žiadnu dodatočnú knižnicu, ktorá sa zaoberá uživateľskými vstupmi.
V zdrojovom kode to bolo realizované pomocou triedy **Window a potomkov abstraktnej triedy Menu. Systém práce triedy Window je naznačený na obrázku ¡OBR¿. Ide o to, že ak sa má nastaviť v obrazovke nové menu, potom treba oknu, ktorého inštancia triedy Window reprezentuje, povedať, aby toto menu pridal do ZÁSOBNÍKA. Keďže okno je v SDL povolené maximálne jedno, inštancia triedy Window bude práve jedna. Preto ďalej v texte budeme pod označením bfWindow myslieť práve túto jednu inštanciu a nie triedu.**

Pomocou ZÁSOBNÍKA na pridávania menu je zrejmé, do ktorého menu sa program vráti po zrušeni obrazovky so základným Menu. Zrušenie obrazovky sa prevedie zavolaním metódy pop(). Táto metóda odoberie menu zo zásobníka a pomocou metody Menu::clean uvoľni zložky potrebné pre vykreslenie. Efektívne sa tak šetrí pamäť. Pre potomka triedy Menu musia byt implementované nasledovné metódy:

Init pre inicializáciu dynamických položiek nutných k správnej funkcií príslušného menu (načítanie obrázkov, vygenerovanie pisma a pod.)

Draw pre vykreslenie celého okna

Process , ktorý je volaný v nekonečnom cykle kvôli spracovaniu udalosti z klávesnice

Resume uvedie menu do pôvodnej podoby

Resize pre prípadné zmeny rozlíšenia okna na prepočítanie premenných určujúcich pozíciu

Clean pre deinicializáciu dynamických premenných pre čo najmenšie množstvo alokovaej pamäte. V tomto ohľade sa táto implementáacia, čo sa obsadenej pamäte týka, dokonca predčí implementáciu pomocou grafickej knižnice QT. QT je zameraná na uživateľskú prívetivosť. Jednotlivé formuláre sa vytváraju pomocou WIDGETOV(prvkov), ktoré su vo formulári neustále prítomné.

Trieda Window pri žiadosti o pridanie menu do ZÁSOBNÍKA inicializuje menu, ktoré sa má pridať a v prípade úspechu inicializácie ho pridá a vykreslí. Vďaka cyklickemu volaniu metódy process objektu na vrchole zásobníka je zaistené, že reakcia na udalosť (klávesnica, myš) sa bude týkať práve nasadeného menu.

Kapitola 4

Porovnanie

Existuje mnoho nástrojov, ktoré sa venujú súťažne algoritmom zameraným na programovanie robotov. Z typických zástupcov môžeme uviesť uz spomínanych RoboCode, C++ robots, ako aj MindRover, Grobots a množstvo ďalších v [12]. Väčšinou sa zameriavajú na prežitie v aréne. Podla corewar terminológie sa tento cieľ nazýva "King Of The Hill" (ďalej len KotH)
Stručne porovnáme výsledný program bakalárskej práce s uvedenými aplikáciami, zhrnieme výhody a nevýhody ich prístupov a čo nové prinásame. Obmedzíme sa len na niekoľko charakteristických nástrojov ponímania sveta podobných CODEWARS. Tie budú demonštrovať, aký koncept CODEWARS použila a ako ho zmenila/vylepšila.

C++ robots[14] je veľmi podobná hra o prežitie. Roboti sú písaní v C++ a majú k dispozícií jedinú zbraň - kanón. Svet je ale veľmi jednoduchý, je to priestor 100x100m ohraničený stenami. Celá hra je na rozdiel od CODEWARS koncipovaná ako turnaj, robot si postupne vyberá súperov. Codewars naviac ponúka možnosť pustiť všetky algoritmy naraz s tým, že hráč môže využívať aj ostatných robotov, aby za neho spravili špinavú prácu.

ARES[13] je typická obranno-útočná hra, ktorá je ale hlavne určená programátorom. Svet sa mení veľmi dynamicky a roboti nevedia vôbec nič o svete. Len sa predstavujú výsledky útoku a podľa toho reagujú. Codewars je z tohoto pohľadu úplne opačná aplikácia, roboti vedia veľmi dobre mapovať svet a spoznajú nepriateľa. Spoločný prvok tak môžu mať len v spôsobe interpretácie algoritmu po kolách. V Codewars bol tento koncept dovedený ad absurdum zavedením plánovača.
RoboCode je tiež zložitá hra, ktorá sa dlho vyvíjala. Je určená pre užívateľov, ktori začinajú s programovacím jazykom Java. Svet sa riadi svojími zvláštnymi pravidlami pre streľbu a víťazenie. Víťazný algoritmus je taký, ktorý získa najviac bodov. Codewars bolo silne inšpirované týmto programom. Líšia sa ale významne v spôsobe, akým vykonávajú algoritmus (RoboCode sa snaží o paralelizáciu) a definovaním cieľa. Codewars oproti

Tabuľka 4.1: Charakteristiky porovnavanych hier

	Svet	zbrane	ciele	obmedzenia na algoritmus
POGAMUT	3D	strely	rôzne	žiadne
ARES	1D	zdieľaná pamäť	KotH	veľkost pamäte
Codewars	2D	strely, roboti	voliteľné	voliteľné
C++Robots	2D	strely, roboti	KotH	žiadne

Tabuľka 4.2: Charakteristiky porovnavanych hier

	pohyb	objekty	jazyk
POGAMUT	komplexný	množstvo	Java
ARES	vykonanie inštrukcií	data	assembler
Codewars	jednoduchý 2D	variácie stien	jednoduchý
C++Robots	jednoduchý		C++

Robocode prináša možnosť prispôsobiť si robota pomocou vlastnosti jeho algoritmu a s možnostou napisat aj čisto mierumilovného, no chytrého robota.

Koncepty prijaté v jednotlivých hrách sú popísané v tabuľke 4.1 a v 4.2. Tabuľka obsahuje iba tie hry, ktoré sa od seba vyrazne líšia.

Kapitola 5

Záver

5.0.1 Zhodnotenie splnenia cieľov

V našom programe sa podarilo implementovať dvojrozmerné prostredie pre robotov, kde platia základné fyzikálne. Svet je navrhnutý tak, aby jeho zmena spôsobila rozdielne nároky na algoritmy a tým zvyšovala obtiažnosť hry v zmysle nároku na úspešnosť algoritmu. Toto je docielené pomocou plánovačov.
Program taktiež priniesol nový pohľad na ciele, ktoré sa dajú v dvojrozmernom svete dosiahnuť. Užívateľovi je dovolené meniť vlastnosti svojho robota v závislosti na type algoritmu.
Program je pripravený na testovanie algoritmov. Z časových dovodov sa však nepodarilo otestovať algoritmy, ktoré su považovanie v niektorých hrách za dobré v danom rozsahu.

5.0.2 Možne rozširenie CODEWARS

V tejto sekcii zhrnieme rozširenia, o ktorých už bola zmienka v texte, ich prínos a možný smer rozvoja problematiky.

Rozšírenie vzhľadom na jazyk

Jedným zo spomenutých rozšírení je navrhnutie viditeľnosti tak, aby robot mal asymetricke 'oči', (t.j priamka definovaná smerom robota), v ktorom je práve otočený a nebude rozdelovať výseč na dve rovnaké časti. Tento škuľavý robot bude do nejakej strany vidieť viac ako do druhej. Dokonca je možné pripustiť extrém, kedy by robot videl iba za seba a tým by miatol súperov. Chôdza do akejkoľvek strany je povolená a tak jediný výsledok by bol, že by nesedelo zobrazovanie. Robot nemôže spoľahnúut ani na to, ako je otočeny. Algoritmus by musel veľmi sofistikove testovať, kam robot vidi, alebo si prinajmnšom často a správne tipnút. Takto hendikepovany robot je skutočnou výzvou najmä pri hre skúseného hráča so začiatočníkom. Preto toto rozšírenie je hodné pozornosti.

Ďalšou možnosťou je nechať užívateľa definovať, aký plánovač bude konkrétny robot používať. Možno očakávať, že výsledky by mohli byť zaujímavé, nakoľko to vedie k naprogramovaniu robota, ktorý má podla všetkého rýchlejší algoritmus.

Rozširenie vzhľadom na jazyk

Jazyk robota aktuálne nepodporuje deklarovanie premenných, ktore boli vo funkcii už niekedy deklarované. Toto chovanie je síce pochopiteľné, avšak programátorsky nepríjemné. Preto by sa dalo uvažovať o primeranej náprave. Jazyk dostatočne pokrýva základné požiadavky na popísanie chovania robota. Avšak je príliš nízkoúrovňovýy, uživateľ si musi mnohé detaily ošetrťt sám. To na jednej strane môže pôsobiť blahodárne na vymyšľanie stratégie, na druhej strane môže uživateľa znechutiť napr. to, že si jeho robot zabúda všimať, že je ostreľovaný. Preto by bolo možné rozširť jazyk o funkcie, ktoré sa automaticky spustia pri vyvolaní udalosti. Toto by však vyžadovalo hlbší zasah do kódu, pretože robot ako objekt a jazyk ako hýbateľ robota sú implementované ako dve nezávislé entity. Bolo by nutne implementovať príslušné komunikačne rozhranie.
Ďalšou alternatívou je pri implementovani reakcií na udalosti nechať uživateľa vopred definovať tieto udalosti, na ktoré bude robot reagovat (napriklad "on seeEnemy()¿0"). Pri dobrom komunikačnom protokole by stačilo súcasne s vykonavanim kódu kontrolovať zoznam podmienok. Obmedzenie počtu udalosíi, na ktoré robot môže reagovať, by tiež mohlo prispieť k zaujímavým úvahám o strategiách

Literatúra

[1] http://libsdl.org

[2] http://robocode.sourceforge.net/

[3] http://ulita.ms.mff.cuni.cz/pub/predn/pp/

[4] Steffen Heinz and Justin Zobel and Hugh E. Williams, *Burst Tries: A Fast, Efficient Data Structure for String Keys*, ACM Transactions on Information Systems, 2002, volume 20, pp. 192–223.

[5] http://msdn.microsoft.com/en-us/library/bb905470.aspx

[6] Tim Lindholm, Frank Yellin :*The JavaTM Virtual Machine Specification, Second Edition*

[7] lua.org

[8] http://karel.webz.cz/

[9] http://mpherbert.codeplex.com/

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry (2nd revised ed.)* . Springer-Verlag. ISBN 3-540-65620-0. Chapter 14: Quadtrees: pp. 291–306.

[11] Použitá galéria obrázkov: http://www.bghq.com/fft

[12] http://sumost.ca/steve/games/ - rozcestník hie zaoberajúci sa algoritmamy prežitia

[13] http://harald.ist.org/ares/

[14] http://www.gamerz.net/c++robots oficiálna stránka hry C++ Robots

[15] Lex & Yacc John R. Levine, Tony Mason, Doug Brown Paperback - 366 pages 2nd/updated edition (October 1992) O'Reilly & Associates ISBN: 1565920007

Dodatok A

CD

Prilohou bakalárskej prace je aj CD so zdrojovymi subormi.

A.1 struktura CD

- **image obsahuje obrázky nutné pre vykjreslenie základného menu**
- **graphic obsahuje implenet=aciu tried Skina SkinWorker**
- **scheduller obsahuje implementáciu plánovača**
- **language obsahuje všetky štruktúry potrebné pre vygenerovanie medzikódu**
- **objekct obsahuje implementáciu všetkých objektov, ktoré sa môžu vyskytovať v mape.**

Každá táto časť ďalej obsahuje podadresáre h a cpp s hlavičkovými a zdrojovými súbormi

Dodatok B

Gramatika robotštiny

```
#include <iostream>
#include <queue>
#include "../../language/h/lval.h"
#include "../../language/h/robot.h"
#include "../../generatedFiles/h/hflex.h"
#include "../../language/h/parser_functions.h"
#include "../../add-ons/h/macros.h"

#define YYSTYPE Lval
#define YYLTYPE unsigned
#define YYLLOC_DEFAULT(cur, rhs, n) do  while(0)
#define YYERROR_VERBOSE 1

static void yyerror(YYLTYPE *line, Robots* ctx, const char *m);

%}

/* keywords */
%token TOKEN_MAIN "main function"
%token TOKEN_LOCATION "word location"
%token TOKEN_OBJECT "keyword object"
%token TOKEN_VAR_REAL "keyword real"
%token TOKEN_VAR_INT "keyword integer"
%token TOKEN_VOID "keyword void"
%token TOKEN_FUNCTION "keyword function"
%token TOKEN_IF "keyword if"
%token TOKEN_ELSE "keyword else"
%token TOKEN_WHILE "keyword while"
%token TOKEN_DO "keyword do"
%token TOKEN_FOR "keyword for"
%token TOKEN_RETURN "keyword return"
%token TOKEN_BREAK "keyword break"
%token TOKEN_REFERENCE "keyword var"
```

```
%token TOKEN_NULL "keyword null"
%token TOKEN_THIS "keyword this"
%token TOKEN_CONTINUE "keyword continue"
%token TOKEN_ROBOT "keyword robot"
%token TOKEN_RND "random function"
%token TOKEN_RET_TARGET "function get_target"
%token<op> TOKEN_OPTION "robot setings"
%token<of> TOKEN_OBJECT_FEATURE "function asking about state"

/* delimiters */
%token TOKEN_SEMICOLON ";"
%token TOKEN_DOT "."
%token TOKEN_COMMA ","
%token TOKEN_LPAR "("
%token TOKEN_RPAR ")"
%token TOKEN_LSBRA "["
%token TOKEN_RSBRA "]"
%token TOKEN_ASSIGN "="
%token TOKEN_BEGIN "{"
%token TOKEN_END "}"

/* literals */
%token<ident> TOKEN_IDENTIFIER " name of variable or function"
%token<number> TOKEN_UINT "integer number"
%token<f_number> TOKEN_REAL "real number"
%token TOKEN_SEEN "seen function"

/* target of game*/
%token TOKEN_VISIT "keyword visit"
%token TOKEN_VISIT_SEQUENCE "keyword visit_seq"
%token TOKEN_KILLED "keyword killed"
%token TOKEN_SKIN "keyword skin"
%token TOKEN_KILL "keyword kill"
%token TOKEN_START "keyword start"

/* group tokens */
%token<operation> TOKEN_OPER_REL "<, >, >=, =<"
%token<operation> TOKEN_OPER_SIGNADD "sign or additive operator"
%token<operation> TOKEN_OPER_MUL "multiplicative operator"
%token<operation> TOKEN_PLUSPLUS "++"
%token<operation> TOKEN_MINUSMINUS "--"
%token<operation> TOKEN_BOOL_AND "&&"
%token<operation> TOKEN_BOOL_OR "|| or !"

%type<ident> function_name "function name"
```

```
%type<type> simple_type "simple variable type"
%type<type> complex_type "complex variable type"
%type<type> type "variable type"
%type<type> return_type "return type"

%type<ranges> ranges "range"
%type<ident> function_header "function header"

%type<entries> parameters_empty "zero or more paramaters"
%type<entries> parameters "paramaters"

%type<instructions> names "inicializations"
%type<instructions> names_ "inicialization"
%type<instructions> cycle_for "for cycle"
%type<instructions> local_variables "local variable(s)"
%type<instructions> block_of_instructions "block of instructions"
%type<instructions> global_variables "global variable(s)"
%type<instructions> commands "block of commands"
%type<instructions> command "command"
%type<instructions> matched "if/else block"
%type<instructions> unmatched "if block"
%type<instructions> init "create or assign"
%type<instructions> assign "assign"
%type<instructions> command_var "command or vvariable declaring"
%type<instructions> simple_command "simple command"
%type<instructions> commands_and_empty "nothing or command"

%type<array_access> array_access "[number(,number)*]"

%type<defVal> values "values expression"

%type<output> array "array declaring"
%type<output> variable "variable declaring"
%type<output> variable_left "variable to be assigned to "
%type<output> number "number"
%type<output> declare_functions "declaring function(s)"
%type<output> declare_function_ "declare function"
%type<output> unary_var "unary variable"
%type<output> expression_bool "expression with boolean result"
%type<output> expression "expression"
%type<output> exps "expressions"
%type<output> expression_base "variable or constant"
%type<output> expression_mul "multiplying expression"
%type<output> expression_add "plus expression"
%type<output> expression_bool_base "variable or constant acting as
boolean"
```

```
%type<output> expression_bool_or "|| expresion"
%type<output> call_fce "calling function"
%type<output> call_parameters "parameters of function being called"
%type<places> places "defining target places"
%type<target> place "defining target place"

%start program

%error-verbose

%pure-parser

%parse-param

%lex-param

%locations

%%

program: program robot
|robot
;
define_bot:TOKEN_ROBOT TOKEN_IDENTIFIER
;
robot:  define_bot TOKEN_BEGIN options targets global_variables
declare_functions TOKEN_MAIN TOKEN_LPAR TOKEN_RPAR
block_of_instructions TOKEN_END
;
targets: /* default target */
|targets TOKEN_VISIT TOKEN_LPAR places TOKEN_RPAR
|targets TOKEN_KILL TOKEN_IDENTIFIER
|targets TOKEN_VISIT_SEQUENCE TOKEN_LPAR places TOKEN_RPAR
;
place: TOKEN_UINT
|TOKEN_LSBRA TOKEN_UINT TOKEN_COMMA TOKEN_UINT TOKEN_RSBRA
| TOKEN_START TOKEN_LSBRA TOKEN_IDENTIFIER TOKEN_RSBRA
;
places: place
| places TOKEN_COMMA place
;
options: /* nothing */
| options TOKEN_OPTION TOKEN_ASSIGN TOKEN_UINT
| options TOKEN_SKIN TOKEN_IDENTIFIER
;
global_variables: /* no parameters */
```

```
| global_variables local_variables
;
type: simple_type
| complex_type
;
local_variables: type names TOKEN_SEMICOLON
;
simple_type: TOKEN_VAR_REAL
|TOKEN_VAR_INT
|TOKEN_LOCATION
|TOKEN_OBJECT
;
complex_type: simple_type ranges
;

ranges: TOKEN_LSBRA TOKEN_UINT TOKEN_RSBRA
|ranges TOKEN_LSBRA TOKEN_UINT TOKEN_RSBRA
;

names_: TOKEN_IDENTIFIER
|TOKEN_IDENTIFIER TOKEN_ASSIGN expression
|TOKEN_IDENTIFIER TOKEN_ASSIGN begin_type values end_type
;
names: names_
|names names_
;
begin_type: TOKEN_BEGIN
;
end_type: TOKEN_END
;
values: expression
| values TOKEN_COMMA expression
| begin_type values end_type
| values TOKEN_COMMA begin_type values end_type
;
declare_functions: /* no declared functions */
|declare_function_
;

function_header:return_type function_name TOKEN_LPAR parameters_empty
TOKEN_RPAR
;
function_name: TOKEN_FUNCTION TOKEN_IDENTIFIER
;
return_type: TOKEN_VOID
|type
```

```
;

parameters_empty:
| parameters
;
parameters: type TOKEN_IDENTIFIER
| parameters TOKEN_COMMA type TOKEN_IDENTIFIER
| TOKEN_REFERENCE type TOKEN_IDENTIFIER
| parameters TOKEN_COMMA TOKEN_REFERENCE type TOKEN_IDENTIFIER
;
declare_function_: function_header block_of_instructions
|declare_function_ function_header block_of_instructions
;
number: TOKEN_OPER_SIGNADD TOKEN_REAL
|TOKEN_OPER_SIGNADD TOKEN_UINT
|TOKEN_REAL
|TOKEN_UINT
|TOKEN_RND
;

begin: TOKEN_BEGIN
;

end: TOKEN_END
;

block_of_instructions: begin commands_and_empty end
;
commands_and_empty:  /* empty */
| commands
;

commands:  TOKEN_SEMICOLON
| matched
| commands matched
| unmatched
| commands unmatched
;
cycle_for: TOKEN_FOR
;
command: cycle_for TOKEN_LPAR init expression_bool TOKEN_SEMICOLON
simple_command TOKEN_RPAR begin commands end
|TOKEN_DO begin commands end TOKEN_WHILE TOKEN_LPAR
expression_bool TOKEN_RPAR TOKEN_SEMICOLON
|TOKEN_WHILE TOKEN_LPAR expression_bool TOKEN_RPAR
begin commands end
```

```
|TOKEN_RETURN expression TOKEN_SEMICOLON
|TOKEN_RETURN TOKEN_SEMICOLON
|TOKEN_BREAK TOKEN_SEMICOLON
|TOKEN_CONTINUE TOKEN_SEMICOLON
|simple_command TOKEN_SEMICOLON
;
command_var: local_variables
| command
;
simple_command: assign
|unary_var
;
assign: variable_left TOKEN_ASSIGN expression
;
array: TOKEN_IDENTIFIER array_access
|TOKEN_SEEN TOKEN_LSBRA expression TOKEN_RSBRA
;

call_fce: TOKEN_IDENTIFIER TOKEN_LPAR call_parameters TOKEN_RPAR
|TOKEN_OBJECT_FEATURE TOKEN_LPAR call_parameters TOKEN_RPAR
;

call_parameters: expression
| /* ziadny parameter */
|call_parameters TOKEN_COMMA expression
;
matched:TOKEN_IF TOKEN_LPAR expression_bool TOKEN_RPAR matched
TOKEN_ELSE matched
| command_var
|block_of_instructions
;
unmatched: TOKEN_IF TOKEN_LPAR expression_bool TOKEN_RPAR
block_of_instructions
|TOKEN_IF TOKEN_LPAR expression_bool TOKEN_RPAR command
|TOKEN_IF TOKEN_LPAR expression_bool TOKEN_RPAR matched
TOKEN_ELSE unmatched
;
init:  local_variables
| assign TOKEN_SEMICOLON
;
unary_var: variable
|variable TOKEN_PLUSPLUS
|variable TOKEN_MINUSMINUS
;
variable_left:TOKEN_IDENTIFIER
|variable_left TOKEN_DOT TOKEN_IDENTIFIER
```

```
|array
;
variable:  TOKEN_THIS
| TOKEN_NULL
|call_fce
|variable_left
;

array_access: TOKEN_LSBRA exps TOKEN_RSBRA
;
exps:  expression
| exps TOKEN_COMMA expression
;
expression_base: unary_var
|number
|TOKEN_LPAR expression_bool TOKEN_RPAR
;
expression_mul:expression_base
|expression_mul TOKEN_OPER_MUL expression_base
;
expression_add: expression_mul
|expression_add TOKEN_OPER_SIGNADD expression_mul
;
expression: expression_add
;
expression_bool_base: expression
|expression TOKEN_OPER_REL expression
;
expression_bool_or: expression_bool_base
| expression_bool_or TOKEN_BOOL_OR expression_bool_base
;
expression_bool: expression_bool_or
| expression_bool TOKEN_BOOL_AND expression_bool_or
;
%%

static void yyerror(unsigned *line, Robots* ctx, const char *message)
{
ctx->parseError( deconvert<const char *>(message) +" at line " +
deconvert<int>(*line) +"\n");
}
```