

Kapitola 1

Uvod

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

Jedným z prvých programov, ktorý využíval koncept programovateľného robota, je KAREL. Tento program bol vytvorený pre podporu výučby programovacích jazykov. Jednoduchosť jazyka, možnosť pozorovať jednotlivé kroky algoritmu. Je tu príležitosť potrať svoje schopnosti zdánlivo banálnymi príkladmi (napr. Napíšte program, po skončení ktorého bude robot v strede svojho sveta) viedla k vytvoreniu súťaží, ktoré rôznymi obmedzeniami kladenými na algoritmus nútili programátora pristupovať k problému kreatívne.

Dobrou príkladom takejto súťaže je hra HERBERTt??, 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.

1.1 Motivacia

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 rátaju s pevne danými dátami (rozostavenie políček a podobne). Na podobnom princípe je založené aj súbežne pustenie algoritmov s tým, že ich vykonávanie môže ostatným škodiť. Algoritmus súťažiaceho musí rátať s nerovnakým prostredím sveta a môže dokonca profitovať zo znalosti stratégie súperov. Preto sa v tejto bakalárskej práci zaoberáme algoritmi robotov, ktoré sa budú vykonávať paralelne v rovnakom prostredí. Toto prostredie sa však môže chovaním algoritmov zmeniť, s čím musí algoritmus počítať.

Najbližším príkladom toho, čo by sme chceli docieľiť je hra ROBOCODE??, kde užívateľ programuje v jazyku Java tanky. Tanky hľadajú a ničia nepriateľské tanky. Vo virtuálnom svete sú nastavené základné obmedzenia, s ktorými musí užívateľ počítať. Napr. čo sa stane, ak strieľa tank príliš často a pod. Toto riešenie je strednou cestou medzi extrémami v podobných programoch, ako je sú:

- ARES - hra dvoch hráčov. Odohráva sa na mieste simulujúcom pamäť počítača. Úlohou hráča je naprogramovať robota v tomto prostredí v strojovom kóde (asembleri). Kritériom úspešnosti je program, ktorý sa bude vykonávať (prakticky hociaký) 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 prepísať pamäť takým spôsobom, aby sa druhý program ukončil. Hráč dopredu nepozná ani program protivníka ani dáta, ktorými je inicializovaná pamäť, ak sa hráči dopredu nedohodnú. Oba programy su podobne ako v reálnom svete uložené v pamäti počítača a keďže pamäť 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 teda roboty.

- POGAMUT je už vysoko komplexná hra na roboty. Algoritmy sa dajú programovať v Jave, čím je dovolené používať špeciálne znaky jazyka, ako je napríklad preťaženie, dedičnosť, atd. Prostredie je trojrozmerné a tým majú roboti možnosť širokej škály hybov, skákania po stene, sklony, pohľady hore a dole. Spôsob, akým sa ubližuje ďalším robotom, je, že intuitívnejší robot vystreľuje obmedzené množstvo striel a má na výber viac zbraní, ktoré sa líšia presnosťou zásahu. Hráč má dokopy možnosť ručne riadiť vlastného robota proti naprogramovanému a tým otestovať vhodnosť jeho algoritmu.

Extrémom v tomto kontexte sú:

- trojrozmerný priestor, ktorý ponúka kvantum možností, ako realizovať pohyb - lezenie po stenách, skákanie, graviácia item množstvo objektov pôsobiach na robota, napr. úkryty, strely, vyháňanie sa strelám, obehnutie prekážky, aplikovanie pathfindingu, obmedzenie
- zobrazovanie dobre zrozumiteľne pre pozorovateľa
- možnosť obmedzeného videnia robotov, robot sa môže schovať alebo byť tak ďaleko, že ho algoritmus nezaregistruje

Uvedené hry uspokojujúco spĺňajú základnú problematiku boja algoritmov. V ARES-e je kritériom zostať nažive podobne ako v ROBOCODE, v POGAMUTe je navyše vopred dane kritické množstvo protivníkov. Hodnotiacou funkciou je rýchlosť, kto skôr splní cieľ, vyhráva.

Zostrojenie takejto hry ale vyžaduje podrobnejší prehľad o nárokoch na jazyk, svet a samotných robotov. Preto sa zameriame na nasledovné charakteristiky týchto hier:

Priestor hier Kým v ARES-e ide o 1D priestor (jedna veľka pamäť - pole), boj v POGAMUT-e sa odohráva v 3D priestore. S tým súvisí pohyb po ireálnom svete. 3D priestor má omnoho viac možností, ako realizovať pohyb. Je nutné zväziť, či bude povolené lietanie, padanie, pohľad zhora, zdola, vrhanie zbraní z boku, a v akých smeroch sa objekty sveta odrážajú a podobne. V ARES-e sa o pohybe, ako ho pozname (plynulý prechod z miesta A na miesto B) nedá ani hovoriť, pretože všetky akcie súviace so svetom sú inštrukcie a zmeny v pamäti.

Kritéria úspešnosti V ARES-e je jasné, že hra skončí, keď hrač nedokáže naďalej vykonávať svoj program. V POGAMUT-e je situácia o poznanie horšia: Pri programovaní robota sleduje programátor (hráč dva ciele: (a) buď naprogramovať takého robota, ktorého zložiť bude výzva, alebo (b) naopak takého robota, o ktorom sa všeobecne vie, že síce bude poraziteľný, ale nie je ľahké ho obísť. To znamená naprogramovať takého robota, ktorého zložiť bude výzvou, ale ktorý bude mať chyby, ktorých sa da využiť.

Spôsoby boja POGAMUT na rozdiel navyše od ARES-a implementuje omnoho viac spôsobov, ako ublížiť robotovi, od rôznych zbraní po odrazenie guľiek, rýchleho spadnutia na zem, atď. Jediným spôsobom, ako je možné v ARES-e poškodiť protivníkovi, je prepísať mu tú časť pamäte, o ktorej je dôvod predpokladať, že ju bude v dohľadnej dobe potrebovať. Vyhodnotenie algoritmu nie je tak možné po častiach, ale až po skončení celej simulácie. V POGAMUT-e je možné rozlíšiť už v priebehu simulácie, ako a či robot zasiahol protivníka.

Vykonávanie programu Ďalším prístupom, ktorý je v vytváraní hry dôležitý, je aj spôsob, v akom poradí sú akcie hráčov interpretované. Aby ostatní hráči neboli znevýhodnení, je vhodné vykonávať jednotlivé časti nezávisle od na ostatných robotov podľa jednotných pravidiel pre všetkých. V ARES-e je to jednoduché, hra prebieha po kolách. Každé kolo znamená vykonanie aktuálnej inštrukcie, čo je to spravodlivé pre všetkých hráčov. V POGAMUT-e je nutné zaistiť paralelizáciu, aby robot nebol závislý na vykonávaní programu ostatných robotov.

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 roboti môžu ublížovať.

Bakalárska práca sa zaoberá vytvorením vhodného nástroja, v ktorom môže užívateľ meniť svet, odohrávajú sa súboje, naprogramovať robota a zistiť úspešnosť napísaného algoritmu. Program by mal byť napísaný tak, aby bol portabilný.

Preto je potrebné sa zamerať najmä na:

Vytvorenie virtuálneho sveta Naprogramovaný robot by mal žiť vo virtuálnom isvete, kde je jednoduché sledovať postup vykonávania jeho algoritmu. Z toho

vyplýva nárok na prostredie, v ktorom sa bude súboj robotov odohrávať. Súčasťou sveta budú objekty, ktoré interagujú s robotmi a prinášajú tak do vymýšľania stratégií komplikovanejšie prvky. V ARES-e reprezentujú tieto objekty dáta uložené vo virtuálnom svete (pamäti), v prípade POGAMUT-a sú to steny, teleporty, priepasti, strely a pod. Treba tiež vymedziť a implementovať také objekty do sveta, ktoré prispievajú k vymýšľaniu sofistikovanejších stratégií. To zahŕňa steny a ich vlastnosti, napr. priehľadnosť, existencia predmetov na dobíjanie zdravia, streliva a pod. Predpokladá sa, že tak vytvorený virtuálny svet bude možné upravovať a vytvárať, aby algoritmy boli napísané "na telo" jednej mapy/počiatočnému stavu sveta. Hráč bude mať možnosť ovplyvniť/zmeniť správanie virtuálneho sveta.

Dynamika sveta Naprogramovani roboti budú mať možnosť bojovať, t.j. si ubližovať a výsledok útoku bude známy v okamihu ublíženia pre ľahšie vyhodnotenie programu. Roboti vo svete sa budú pohybovať všetkými smermi a interagovať s ostatnými objektami vo svete (OK rýchlosť je už vlastnosť, to by som riešila osobitne JJ. Hráč by mal mať tiež voľbu útoku robotov na blízko aj na diaľku pre lepšie strategické možnosti, inak by sa hra zvrhla na "najdi robota a kopni ho").

Životný cyklus robotov Život robotov bude začínať vstupom do sveta a končiť opustením sveta. Možnosť nejakého znovuzrodenia ako v hre POGAMUT sa nebude pripúšťať, ale bude 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 (tinka)

Vlastnosti robotov Ani jeden z uvedených programov ale nemá možnosť špecifikovať, ako budú jednotlivé vlastnosti robotov. Či robotov skolí jedna rana (POGAMUT), alebo tu je aj možnosť nejakého obmedzeného znovuzrodenia (ARES). V bojových hrách sa tiež ukázalo vhodne umožniť, aby si hráč pred samotným vstupom do sveta mohol tieto vlastnosti upraviť a tým ovplyvniť priebeh súboja.

Obrazok na (TODO) naznačuje smer, v ktorom sa bude práca uberať.

Kapitola 2

Analýza

Súčasťou tejto kapitoly je zdôvodnenie jednotlivých rozhodnutí, ktoré sme navrhli. Chceme vytvoriť taký virtuálny svet, v ktorom môžeme pozorovať chovanie robotov, pričom kritériom je zostať naživo a hodnotiacou funkciou je maximálne predĺžiť čas života robota.

2.1 Virtuálny svet

Neoddeliteľnou súčasťou hry je virtuálny svet (prostredie), v ktorom sa bude suboj odohrávať. Najskôr vysvetlíme, čo všetko svet obsahuje, ako sa v ňom žije z hľadiska robota i z hľadiska užívateľa (narábanie s objektom v programe) a ako to prispieva k úspešnosti algoritmu.

Uvažovaný svet bol vybraný dvojrozmerný, pretože poskytuje dostatok možnosti pre dianie na ploche (smer pohybu, zrozumiteľne vykresľovanie stavu a pod.) a súčasne nie je obtiažne implementovateľný.

2.1.1 Súčasti virtuálneho sveta

Robot žije v prostredí a môže ovplyvňovať (útočiť) ďalších robotov .na ďalšie roboty, teda sám je objektom sveta. Roboti sa smú pohybovať všetkými smermi a tak by sa bez ďalších objektov jednalo len o nájdenie robota, ktorý sa potom môže brániť pohybom (nic iné by vo svete nebolo). Tento prístup je tiež zaujímavý, ale jednotvárny, ráta sa stále s tým istým stavom sveta). Preto uvažujeme aj o ďalších objektoch ďalšie objekty.

Nakoľko háč aktívne nevstupuje do vykonávania algoritmu, je nutné popísať virtuálny svet z hľadiska robota, resp. užívateľovou znalosťou objektov, na ktoré môže robot reagovať.

V ARES-ovi sa robot orientuje podľa dát v pamäti, číta, porovnáva a prepisuje. V POGAMUT-e reaguje robot na vizuálne podnety, ktoré vidí aj pozorovateľ. Hráč ma naprogramovať algoritmus chovania robota a tak je prirodzenejšie použiť podobný princíp ako v POGAMUT-e t.j. zariadiť robotovi možnosť získať informáciu zo sveta v nejakom obmedzenom okolí. Objekty uvažované vo svete vzhľadom na to, čo môže algoritmus využívať, sú nasledovné:

Robot ako objekt

Základnou vlastnosťou robota je, že môže ubližovať ostatným robotom, musí ich najskôr lokalizovať. Teda roboti sami musia mať vlastnosti objektu sveta (to znamená na ne reagovať)

Ako veľmi moc robotovi tieto útoky uškodia, je vyjadrené celým číslom. Tak sa da 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). Čím väčšie číslo, tým väčšia škoda sa deje robotovi. V prípade vzdialeného útoku je tiež dôležitou otázkou, ako ďaleko môže robot zaútočiť. Ak je toto číslo vopred dané, mal by o tom robot vedieť dopredu, aby mohol svoj algoritmus prispôbiť.

Útok robotov prebieha na úrovni ich tiel a nie programu (viz hra typu ARES). Ďalšou otázkou je, koľko takýchto zásahov robot vydrží. Keďže útok je vyjadrený pomocou celých čísel, je vhodné vyjadriť celočíselne aj životnosť robotov. Ďalšou vhodnou vlastnosťou je 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. Preto ďalšia vlastnosť, ktorú užívateľ môže u robota nastaviť je, aké množstvo zranenia bude pohltené pred jeho smrťou. Výsledný efekt je ale rovnaký, akoby sa životnosť zvýšila a preto táto vlastnosť nebola použitá.

Strely

Roboti by mali vedieť útočiť na diaľku. To možno doceliť viacerými spôsobmi:

- robot zaútočí z diaľky na konkrétne miesto a tam okamžite vypočíta výsledok útoku. V tom okamihu treba určiť, kedy smie robot zaútočiť týmto spôsobom. Ak môže robot zaútočiť na akékoľvek miesto, potom ostatní roboti nielenže nemajú možnosť sa útoku vyhnúť, ale strácajú sa aj informácie o tom, odkiaľ útok prišiel (na každé políčko môže byť zaútočené). Stratégie sa zredukuju na dva prístupy - náhodne útočenie z diaľky na nejaké políčko a pohyb dotedy, pokiaľ sa nenájde cieľ a na masívny útok na cieľ.
- Minimálne je teda nutné obmedziť pravidlami, na ktoré miesta sa môže útočiť z diaľky. Najviac intuitívne a ľahko zobraziteľné je vymedziť polomer zásahu. Problémom ale stále zostáva nemožnosť vyhnúť sa útoku na diaľku. Nie je tu možné zaregistrovať útok a adekvátne naň zareagovať, poprípade zareagovať na útočníka.
- Ďalšou možnosťou je vytvorenie strely. Strela je objekt, ktorého jedinou činnosťou je pohybovať sa predvídateľne vpred po dobu dopredu známeho času (v danom smere výstrelu). Strela však skončí svoju činnosť aj v okamihu, keď spravi útok na blízko - zasiahne objekt.

Ak robot útočí tým, že na cieľ vystrelí, strela spraví potom útok nablízko. Tento prístup poskytuje väčšiu voľnosť pri útočení, nakoľko stačí rozhodnúť, v ktorom smere má strela ísť. Ďalej je možné približne odhadnúť smer, odkiaľ

strela prišla. To je dôležité pri rozhodovaní sa, kam ísť, či robota napadnúť priamo (nablízko) alebo odpovedať strelbou. Strely tak boli pridané ako ďalšie objekty, ktoré zabezpečujú útok na diaľku. Strelivo ale nie je možné dopĺňať, musí byť neobmedzené pre každého robota. Ak robot ale strieľa neobmedzene, potom môže byť hracie pole zahltené strelami, čo výrazne spomalí simuláciu. Preto je počet striel obmedzený. Toto je ďalšia vlastnosť robota. Strela sa okamžite po vykonaní útoku vráti k robotovi, ktorý ju vystrelil a ten ju môže použiť znova.

Steny Ďalším uvažovaným objektom sú objekty, ktoré môže robot využívať na svoju obranu. Útočiť na robota môžu také objekty len z blízka (robot z blízka alebo z diaľky, čo je to vlastne strela z blízka. Vo virtuálnom svete potrebujeme niečo, čo zabraňuje pohybu. Týmito objektami budú steny. Robot ich môže využívať ako strategicky úkryt, podobne ako vojaci využívajú terénne nerovnosti.

Prepadliska - neprístupné miesta Strategickým obmedzením sú miesta, na ktoré robot za žiadnych okolností nesmie stúpiť. V prípade POGAMUT-a 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 vyhýbajúci sa silným útokom z blízka.

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. Náhodne vygenerované pozície:

Roboti vygenerovaní vedľa seba. Toto nám ale nevadí, pretože táto situácia môže nastať kedykoľvek počas behu programu a tak na ňu robot musí vedieť zareagovať.

Nájdenie počiatočného miesta Môže zabráť pomerne veľa času v prípadoch, keď je mapa sveta zložená z veľkého 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 vypočítaných 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 políček. To prináša okrem iného aj možnosť definovať, pre maximálne koľko robotov je mapa ideálna (ráta sa s týmto maximálnym počtom robotov). Ak ale užívateľ zadá viac robotov, simulácia sa aj potom môže uskutočniť. Nutné je len užívateľa upozorniť, že prebehol pokus umiestniť robota náhodne a či

bol tento pokus úspešný. Ak aj pokus nebol úspešný, nič významne sa nedeje, pretože existuje na mape virtuálneho sveta dostatočné množstvo robotov.

2.1.2 Život v prostredí

Roboti žijú vo svete a snažia sa zničiť ostatných. Keďže ale 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 zamerne pohne. Hovoríme tomu **krok**.

Ideálne sa javí postupné vykonávanie pohybu (plynulý prechod na ďalšie miesto), pretože je to prirodzenejšie a takýto pohyb poskytuje priestor pre návrh stratégií. Ak sa robot ocitne pred priepasťou, vie, že robot za ním touto cestou nepride, pretože by spadol. Podobne, ak je za stenou, robot pri pokuse ísť za ňu narazí a neobjaví sa za ňou?nim, takže je robot istým spôsobom chránený. Otázkou je, ako sa bude tento algoritmus vykonávať, akým spôsobom sa robot dostane do zadaneho miesta.

- jedným zo spôsobov je, že sa užívateľ nebude trápiť tým, ako sa na dané miesto dostane, pretože samotná mapa bude ponúkať možnosť navigovať robota. Tento koncept zjednodušuje písanie algoritmu. Následkom tohoto rozhodnutia ale nebude mať užívateľ možnosť zistiť, ako dlho bude trvať robotovi cesta na dane miesto a teda nemôže vedieť ako sa za ten čas zmenil svet. To ale v zásade nie je problém, iba zjednodušuje písanie algoritmov, keď sa raz užívateľ rozhodne, kam chce ísť. Problémom sa môže zdať samotná implementácia. Existujú dva spôsoby, ako pathfinding dosiahnuť:
 - výpočet za behu aplikácie. Potom toto vypočítavanie môže výrazne spomaliť simuláciu, obzvlášť ak ide o veľké mapy.
 - Obmedziť mapu na pozície, kam sa robot môže dostať a tieto si uložiť externe. Otázka, akú cestu zvoliť medzi dvoma bodmi sa potom zjednoduší na postup podľa konkrétnej úsečky. Tento spôsob však vyžaduje netriviálne množstvo pamäte najmä pre veľké mapy.
- pohyb po priamke. Pohyb sa bude bez ohľadu na dostupnosť miesta realizovať po priamke. Preto je nutné zadať podmienku, kedy sa má pohyb prestať vykonávať. Bez tejto podmienky by stačil užívateľovi jeden chybný ťah, aby zablokoval vykonávanie zvyšku algoritmu a tým znemožnil algoritmu vyhrať. Robot ale počas pohybu nebude vykonávať nič iné, a tak bude očakávať, že po nejakom čase sa na tom mieste ocitne. V reči virtuálneho sveta je týmto časom počet 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.

Čo sa samotného pohybu týka, bude zobrazovaný tak, aby ho oko vnímalo ako plynulý *TODO – nejakareferencia* Pohyb, ktorý môže urobiť robot, je pohyb v

ľubovoľnom smere. Smer je určený vektorom $[x, y]$, $x, y \in N$, takže sa ním dá vyjadriť presne smer pohybu. Konkrétny pohyb je potom aproximovaný úsečkou (objekt sa bude pohybovať po úsečke, pričom jeho súradnice budú vypočítané v závislosti na čase). Roboti sa teda môžu pohybovať všetkými smermi vyjadriteľnými celými číslami. Otázkou je, ako sa samotný pohyb týmto smerom realizuje. Robot vidí istý úsek pred sebou a tak je rozumne mu tiež dovoliť sa otáčať a tým pokryť celé svoje okolie. Pohybujúci sa robot bude môcť spraviť nasledujúce veci. Buď sa bude hýbať len smerom, ktorým je otočený alebo sa bude môcť hýbať kamkoľvek. Tieto dva pristupy majú ale rovnaku približne rovnaký efekt. Robot sa v druhom prípade iba nemusí otáčať, čo je zanedbateľná položka. Pre jednoduchosť bol implementovaný spôsob chodia len smerov, ktorým je robot otoceny. Prvý spôsob si môžeme predstaviť ako pohyb u koňa - ten kde nevidí, nevlezie. Druhý spôsob sa dá prirovnať k pohybu jeleňa - ten v prípade núdze uhýba efektne všetkými smermi. .

V súvislosti s pohybom je možné uvažovať o rozšírení virtuálneho sveta nad rámec popisu v `STATICKY SVET`, a to konkrétne o posuvné steny ako i zvláštny druh stien. Steny zabráňujú pohybu robota a striel, preto prínos posuvných sten je v tom, že robot potom bude môcť zmeniť prostredie sveta tak, aby zodpovedal jeho konkrétnemu algoritmu. Vie posunúť stenu tak, aby ho chránila pred strelbou. Napríklad, ak algoritmus počíta s tým, že robot bude útočiť na isté miesto, ale cesta na to miesto prechádza miestom, kde sa neustále pohybujú strely, potom je pre robota vhodné, aby tam mal stenu, za ktorú sa môže schovať. Ak tam žiadna stena nie je, potom existencia posuvných sten dáva nádej na prechod prejdienie?????? tejto nebezpečnej zóny bez úhony.

Spôsob s posuvnými stenami má oproti využívaniu konkrétneho prostredia sveta navyše tú výhodu, že algoritmus musí počítať so zmeneným svetom a tým sa zvyšuje náročnosť hry. Preto svet obsahuje aj posuvné steny. Stále by však virtuálny svet mal možnosť mať statické prostredia (neposuvné) steny.

2.1.3 Kolízie

V súvislosti s pohybom nastáva tiež otázka, kedy a ako budú objekty navzájom interagovať. Kolízia nastane vtedy, keď sa obrazy objektov pretnú (majú spoločný neprázdny prienik). To kladie nemalé nároky na štruktúru virtuálneho sveta, ale súčasne to má tiež vedľajší efekt. Čím 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 asi väčšie nároky na pamäť (obzvlášť, ak bude veľká mapa a malé políčka), ale zato kolíziu vieme určiť okamžite. Stačí zistiť, či v danom políčku, kde leží výsledok pohybu, je objekt rôzny od uvažovaného. Tento spôsob sa dosť často používa v bludiskách, kde su objekty rovnako veľké (každé zaberá práve jedno políčko). Nazvime ich **diskrétné 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 jedného políčka, pretože mapa je rozdelená staticky (políčka sa nepo-

hybujú s objektom). Stále však môže kúsok objektu presahovať nad rámec políčka. Teda kolízia nemusí presne zodpovedať tomu, ako je zobrazená. Preto boli uvádzané nasledujúce algoritmy pre detekciu kolízie, ktoré používajú iba veľkosť objektu:

Quadtree [približný popis, výhody/nevýhody + referencie]

Mriežková metóda [približný popis, výhody/nevýhody + referencie]

Pre lepšiu implementovateľnosť bola zvolená mriežková metóda. Neboli zaznamenané výrazné rozdiely oproti quadtree.

Keď máme vyriešenú kolíziu, je treba určiť, ako jednotlivé objekty reagujú na kolíziu. Rozhodli sme sa takto:

Robot vs. strela Strela ukončí svoj život výbuchom a ušetrí robotovi náležité zranenie. Robotovi sa v tomto okamihu preruší akákoľvek činnosť, ktorú predtým vyvíjal, (napríklad pohyb). To zodpovedá tomu, že strela zasiahla cieľ, (v tomto ponímaní iba roboti) a nemá ďalej dôvod pokračovať (žiadne viacnásobné zranenia). Robot si v tomto okamihu nemusí robiť starosti, že strela sa odrazí priamo k nemu. Je však na ďalšom rozšírení, ako sa strela po zásahu bude chovať.

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

Robot vs. stena Pokiaľ robot narazí na stenu, tá ho zastaví. Toto chovanie je prirodzené, pretože stena tohoto typu predstavuje statický virtuálny svet, ktorý sa nemení.

Robot vs. posuvná stena Posuvná stena má schopnosť meniť svoje miesto. Robot ju má možnosť posunúť (musí byť pri stene 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. Ak sa snažia stenu posúvať obaja roboti a smer ich pohybu je vyjadrený dojedrozmerným vektorom, potom sa stena pohybuje v smere vektorového súčtu týchto dvoch smerov, čo dáva aj fyzikálne prijateľný zmysel.

Robot vs. prepadlisko Robot by na prepadlisko nemal stúpať. Ak by sme sa obmedzili iba na tento princíp, stačí nám stena. Preto robot musí byť potrestaný vstupom na toto políčko. Ako najjednoduchší spôsob sa ponúka strata životov a následne zastavenie alebo prejdienie prepadliska za cenu niekoľkonásobnej straty životov. Bol implementovaný druhý spôsob. Pri prejdieni prepadliska aj za cenu toho, že bude robot polomŕtvy, sa pri dostatočnom množstve života môže ešte podieľať na simulácii, no algoritmus, ktorý sa bude stále pokúšať prestúpiť prepadlisko isto zahynie.

Robot vs. strela Strela pri kontakte s robotom zaútočí svojím ôtokom na blízko. Následne sa robotovi na základe tohoto útoku zníži životnosť. Strela následne zmizne z hracieho pola, pretože zasiahla cieľ. Strela môže zasiahnuť aj toho, kto ju vystrelil. Je to rozumné z toho dodu, aby robot nemal tendenciu strieľať všade, ale rozmýšľal, či to neublíži aj jemu. Strela by preto mala mať väčšiu rýchlosť ako robot. Inak sa môže stať, že po vystrelení správnym smerom sa robot tým smerom pohne (aby mohol sledovať obeť) a zasiahne ho vlastná strela. Potom ale nemá zmysel používať strelu na diaľku, pretože rýchlejšie by bolo možné použiť útok na blízko. Strelu by sa dalo použiť iba v prípade prepadliska. Ak strela bude ale dostatočne pomalá, robot sa jej ľahko vyhne a tým sa opäť stráca význam útoku na diaľku.

Stena vs. strela Strela sa bude môcť od obvyčajnej steny odrážať. Stena nie je nikdy primárnym cieľom. Preto nemá zmysel, aby strela ukončila svoju dráhu pri narazení na stenu. Jediný dôsledok by bol, že robot môže strelu vidieť a tým by bola prezradená pozícia strelca. Napriek tomu, že strela ukončí svoju dráhu len v okamihu, keď zasiahne robota, bude možné pomocou vlastností strely ako dostrel (doba životnosti strely) čistiť cestu aj za "rohom". Možné je i umýselne mýliť miest protivníka vyslaním strely tak, aby sa odrazila. Preto bolo rozhodnute o odrazení strely od steny.

Stena vs. posuvná stena Robota normálna stena zastavuje a posuvná stena sa hýbe robotovým pričinením a poskytuje mu kryt. Je preto logické, aby normálna stena zabráňovala pohybu aj posuvnej stene. Posúvna stena teda na rozdiel od normálnej steny strely neodráža, ale zastavuje.

Strela vs. posuvna stena Otázkou je, či by sa mala posuvná stena hýbať aj pri kontakte so strelou. Posuvná stena je však primárne určená na úkryt robota a jediné, čo by mohla strela urobiť je, vziať zaseúkryt robotovi, keď vystrelí strely. robotovi zase ukryt zobrať keď vystrelí strely. Ak by však tento úkryt mal zmiznúť po vystrelení strely, smer strely by musel so smerom tejto posuvnej steny zviazať tupý uhol. ¡TODO OBR.¿ To znamená, že strela by sa aj tak odrazila, pretože posuvná stena je len druh steny. Výsledok by v najhoršom prípade nemusel byť okamžitý - robot by steny stále posúval tým smerom a tak čiastočne anuloval výsledky streľby. Výsledok by tam bol prinajmenšom neistý a ťažko využiteľný. Preto sa strela od posuvnej steny iba odráža, namiesto toho, aby ju aj posúvala.

Strela vs Strela

Strela sa nijak neprofituje so zrážky s inou strelou, takže sa nič nestane.

2.1.4 Vytváranie máp

Súčasťou práce bolo aj editácia a generovanie máp, nakoľko cieľom je odskúšanie algoritmu v rôznych svetoch. Generovanie mapy pozostáva z určenia prvkov, ktoré sa budú generovať, určenie veľkosti a samotným algoritmom na umiestnenie týchto objektov. Užívateľ bude mať možnosť upraviť vygenerovanú mapu, ak zodpovedá jeho predstavám.

Za generovateľné objekty boli vybrané iba mapy. Roboti a strely sa nemôžu generovať, pretože robota zastupuje štartovné políčko a strely sa nepotulujú virtuálnym svetom, pokiaľ robot nezaútočí. A tak zostali štartovné políčka, obyčajné a posuvné steny a prepadliská. Obyčajné steny sú základným prvkom, s ktorými robot počíta, takže tie sa generujú.

Prepadliská sú obtiažne generovateľné z toho dôvodu, že náhodne generované prepadliská sa môžu stať úzkym hrdlom nejakého koridoru. To znamená, že vzniknú dve oddelené časti, kam robot môže len so značnou stratou na životoch. Je možné tieto situácie nechať ošetriť užívateľom, ale to by musel prejsť celú mapu a hľadať takéto miesta. To ale nie je užívateľsky príjemné. Je možné použiť tú heuristiku, ktorá tieto miesta nájde a odstráni. Taká heuristika nie je náročná na implementovanie - stačila iba kontrola úzkych hrdiel, možných prepojení s ostatnými miestami,

Posuvné steny neboli zahrnuté do generovania. Ich generovania totiž obsahuje aj to, či sú priesvitné a posuvné a či náhodou nevytvoria neriešiteľný virtuálny svet. Preto generovanie posuvných stien je prenechane užívateľovi.

Samotný algoritmus generovania vychádza z nápadu - nie zaplniť miesto objektami, ale miesto plne objektov postupne vyprázdňovať. Týmto spôsobom sa nagerujú steny. Ďalší generovateľný objekt - prepadliská sa potom dogenerujú dodatočne prejednením vygenerovanej kostry virtuálneho sveta a testovaním, či existencia prepadliska neodporuje vyššie uvedeným kritériám.

Popis pazraveho algoritmu. Týmto spôsobom vygenerovaná mapa obsahuje dostatočne veľký priestor pre manevrovanie robota a súčasne sa dostatočne veľakrát podarí vygenerovať miesta, ktoré robot použije ako kryt. Pre naše účely testovania algoritmov to postačuje.

2.2 Programovateľnosť postavy

Chovanie robota na mape bude mať užívateľ možnosť naprogramovať v nejakom jednoduchom jazyku. Ináč: Užívateľ bude mať možnosť naprogramovať chovanie robota na mape v nejakom jednoduchom jazyku. Z toho vyplývajú ďalšie nároky na svet ako aj nároky na programovací jazyk robota.

2.2.1 Rozšírenie sveta vzhľadom na programovateľnosť

Získavanie informácií o svete

Užívateľ má napísať algoritmus, ktorým by sa robot riadil. Preto je nutné vhodným spôsobom získavať informácie o okolí robota, na ktorý má robot reagovať. virtuálny svet máme dvojrozmerný a tak si užívateľ môže získať prehľad o tom, čo je vo svete pozrením do mapy (vizuálne). Hneď bude vedieť, s čím môže počítať na danom mieste, ak sa tam robot ocitne. Ak by robot získaval informácie inak, ako to vidí užívateľ, nastane nekompatibilita v chápaní toho, čo práve robot vidí. Výsledkom môže byť nezrozumiteľnosť vykonávania algoritmu. Preto je pre písanie algoritmu jednoduchšie, ak podobným spôsobom ako hráč reaguje na okolie sveta aj robot. Robot bude teda objekty vidieť.

Videnie robota zahŕňa ešte ďalšie otázky, na ktoré sa treba zamerať, a to ako ďaleko

a akým spôsobom bude robot vidieť. Otázka, ako ďaleko je pre algoritmus dôležitá, pretože čím ďalej robot vidí, tým ucelenejšiu predstavu o svete bude mať a tým je pravdepodobnejšie, že nájde nepriateľského robota a bude mu mocť škodiť *referencianacie*. 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 začneme vymedzením priestoru, v akom by mal robot vidieť. Sme v dvojrozmernom priestore a rozhodli sme sa pre plynulý pohyb, je zmysluplne 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 objekty budú robotom viditeľné a dajú sa popísať nejakým dvojrozmerným útvarom.

Najvhodnejšia sa javí kruhová výseč, podobná tej ako vrhá baterka v tme. Tento spôsob je pre ľudské oko prirodzený *referencianagrafiku*? a preto bol aplikovaný. Viditeľnosť robota tak bude znamenať, brať do úvahy všetky objekty, ktoré sa nachádzajú v kruhovej výseči, definovanej polomerom a uhlom. Ak by všetky tieto objekty boli viditeľné, nemal by zmysel objekt sveta typu stena, ktorá ma 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 už určitej vzdialenosti stojí robot a jednak nie je logicky dôvod, aby sa viditeľnosť správala rozdielne pre rôzne typy objektov. Preto je u každého objektu okamžite jasné, či je priehľadný alebo nie.

Priehľadnosť objektov bola vybraná nasledovne:

- Strela by nemala poskytovať úkryt a teda blokovať viditeľnosť. Je to len nástroj k prevedeniu útoku na diaľku a ako taká by nemala 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 viditeľnosť. Z rovnakého dôvodu to platí pre posuvnú stenu.
- Prepadisko neblokujú viditeľnosť. Z definície vieme, že prepadisko neblokujú pohyb, iba ho sťažuje. Preto je rozumné, aby robot vedel, že cez neho môže prejsť. To znamená, že musí cez prepadisko vidieť.
- Samotný robot blokuje viditeľnosť. Toto rozhodnutie vyplýva z toho, že všetky objekty, ktoré sa pohybujú smerom k robotovi, sa v prípade kolízie zastavia. Preto sa da čakať, že robotom neprejde ani lúč, určujúci viditeľnosť.

Navrhované algoritmy pre výber objektov sú tieto:

Vygenerovanie bitových masiek:

Algoritmus ráta s tým, že výseč sa pokryje najmenším obdĺžnikom a s ním sa potom pracuje. Výhody tejto metódy:

- lineárna časová zložitosť vzhľadom na veľkosť pokrytej plochy

Nevýhody:

- Nutnosť rozdeliť výseč na rovnaké políčka a tým buď nerešpektovať veľkosť objektov alebo rátať s objektami vyskytujúcimi sa na viacerých políčkach.
- Existuje situácia, pre ktorú tento spôsob nenájde skutočnú viditeľnosť tak, ako by to hráč očakával. *OBR*. Tento prípad je však jediný a v simulácii nenastáva často.

Vzhľadom na uvedené nevýhody bola navrhnutá a použitá nasledujúca výsečová metóda:

Ta pozostáva z vytvorenia výseče pre každý objekt, ktorý môže robot vidieť. Následne sa zistí pokrytie, viz. $< OBR >$. Najskôr pre každý objekt zistíme, či sa nachádza v pásme viditeľnosti. Pre každý objekt, ktorý blokuje viditeľnosť, vytvoríme ďalšiu výseč s počiatkom v mieste robota. Potom je viditeľný každý objekt, ktorého ľubovoľná časť je mimo zjednotenia týchto výsečov. V praxi to znamená, že u každého objektu si zapamätáme, akú oblasť by roboti?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 a tým úspešne oddeľuje tie objekty, ktoré nie je vidieť. Avšak existuje situácia, keď tento algoritmus zlyhá, ako je vidieť na obrázku $< OBR >$. Táto metóda je veľmi presná. Počíta s rozdielnymi veľkosťami objektov, je však X-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 užívateľa podnetné napísať algoritmus. K tomu sa viažu podmienky úspešnosti algoritmu. Vzhľadom na súťaž algoritmov sa ponúkajú nasledovné ciele robotov:

Zostať na bojisku posledný

Úspešným algoritmom je ten, ktorý zostane na bojisku posledný. Spôsob, akým možno dosiahnuť tento cieľ, je iba boj so súperiacimi robotmi, útočenie na ich reprezentáciu vo svete. Tento cieľ spĺňa požiadavky na súboj algoritmov - je potrebné napísať čo najlepší bojový algoritmus, t.j. *algorithmous*, ktorý na danej mape v danom prostredí vyhrá. Tento spôsob bol použitý, lebo je intuitívny.

Obranný robot

Nastáva otázka, či robot môže zvíťaziť aj inak než útočením na súperov. Viac sofistikovane vyzerá robot, ktorý zmätie svojho protivníka natoľko, že začne bojovať s inými robotmi. Nazvime tohoto robota robot-manipulátor a robota, ktorý má za úlohu ostatných zničiť, robot-agresor.

Otázkou je, či svet, ktorý sme vytvorili, je vhodný aj pre robotov-manipulátorov. Je, pretože všetko, čo využíva agresor, používa aj manipulátor. Strelbou 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 snaží zostať nažive s minimom zabitých nepriateľov na konte.

Problémom však je, že ak robot nebude mať za úlohu zničiť nejakého robota, 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. Preto tento koncept sám o sebe je zaujímavý, ale je dosť ťažko zhodnotiteľný. V CodeWars koncept čisto obranného robota preto nepodporujeme.

Zničenie konkrétného robota

Aby sa roboti jednotlivých hráčov dali rozlíšiť, sú pomenovaní unikátnymi menami. Ďalšou možnosťou je tak zničenie konkrétného robota. Potom robot, ktorý má za úlohu zničiť takéhoto robota, vyhrá v okamžiku, keď tohoto cieľového robota zničí 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 nie je zaujímavý.

Navštívenie miesta Problémom algoritmu obranného robota je to, že nie je primerane akčný. Rozhýbanie (potulovanie svetom) ho môže prinútiť dodatočná podmienka, napríklad nájdenie nejakého objektu. Potom sa pre víťazstvo musí robot skutočne hýbať. Objektom ale môžu byť aktuálne len roboti, (čo v podstate splýva s predchádzajúcim tvrdením) alebo steny, prepadliská, ktoré sa nedajú nepovažovať za cieľ. (prepadlisko môže robota zničiť a namiesto oslavy bude pohreb, na miesto steny sa robot nikdy nedostane, stretnutie strelou tiež končí fatálne). Preto je nutné, definovať špeciálne miesta, ktoré nebudú objekty a nebudú mať význam pre nikoho iného, len pre robota, ktorý si o toto miesto požiadal. Toto miesto by malo byť súčasťou mapy, aby užívateľ videl, čo presne po robotovi chce. Aby to robot nemal také ľahké a aby užívateľ mohol testovať algoritmus, možno ako špeciálne miesto označiť plánované štartovné pozície robotov.

Obmedzenie na počet nepriateľov Nájdenie miesta, ktoré si robot vybral, môže vyzerat aj tak, že robot bude mať bojový algoritmus, s ktorým prejde celú mapu a raz na svoje pole dorazí, použije stratégiu "dôjdem tam a kto sa mi proti mne postaví, je mŕtvy robot". Tým sa robot nebude nijak líšiť od obvyčajného agresora. Preto sa hodí kombinácia hľadaniamiesta a obmedzenie maximálnym počtom robotov, ktoré môže robot zabiť pred dosiahnutím cieľa. Pre praktické účely bude predefinovaný maximálny počet zabitých nepriateľov rovný $pocetrobotovvmape - 2$ (robot a aspoň jeden protivník). Podobne je možné, použiť takéto obmedzenie zhora na zadanie maximálneho počtu súbojov pred dosiahnutím miesta, kde má robot doraziť. Robot tu už musí mať za sebou nejaké súbojové skúsenosti.

Kombinácie Kombinácie vyššie uvedených cieľov zvyšujú zložitosť vytváraných algoritmov, čím spĺňajú základný cieľ, ktorým je výzva naprogramovať algo-

ritmus. Preto sú kombinácie uvedených cieľov algoritmov nielen možné, ale dokonca žiadúce.

Z rovnakých dôvodov je vhodné kombinovať lokalizáciu s obmedzením počtu zabíjajúcich nepriateľov.

Každému robotovi teda môžeme prideliť špeciálny cieľ. Tento cieľ sa však vždy nepodarí splniť. Napríklad robot, ktorý bol cieľom, zahynie pri pokuse o prechod prepadliskom. Preto je nutné zaviesť buď takzvaný supercieľ (cieľ, ktorý je možné splniť kedykoľvek) na zakončenie simulácie. Tiež je možné odstrániť robota, ktorý nesplnil cieľ. To ale môže vyvolať retazovú reakciu u ďalšieho robota, ktorý ale stále mal možnosť zvíťaziť. Nebolo by spravodlivé, aby bol odstránený len preto, že zlyhala misia iného robota. Preto bolo spravene rozhodnutie o existencii supercieľa. Nech supercieľom bude zničiť ostatných robotov na bojisku v zmysle zachovania princípu "keď už som zlyhal, nech to aspoň niekto neprežije!". Tento cieľ je vždy dosiahnuteľný a obtiažnosť napísania algoritmu zodpovedá požadovanej obtiažnosti.

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 trafi. Podobne rozsah toho čo vidí, by mal byť väčší, aby si mohol lepšie napláňovať trasu pohybu. Preto bol zvolený spôsob modifikovania vlastností. Vlastnosti, ktoré robot má a ktoré sme už uviedli vyššie, sú: útok na blízko, na diaľku, počet životov a definovanie, ako ďaleko robot vidí, počet striel. Na to, aby si robot zisťoval informácie zo sveta a následne podľa nich vyhodnocoval stratégiu, si potrebuje robot niekde uchovávať informácie. Nazvime túto pamäť **úložisko**. Každý robot bude mať vlastné úložisko, aby mohol súťažiť na úrovni sveta a nie prepisovať algoritmus nepriateľov. V tomto úložisku bude pre jednoduchosť akákoľvek informácia zaberať práve jedno miesto. Veľkosť úložiska potom ovplyvňuje znalosti o svete a tým aj algoritmus, ktorý znalosti využíva. Preto by malo byť jednou z ďalších voliteľných vlastností veľkosť úložiska (pamäte)

Ďalšou vlastnosťou, o ktorej je možné uvažovať pre podporu algoritmu, je rýchlosť robotov. Čím vyššia rýchlosť pohybu robotov, tým viac je pravdepodobné, že robot nájde iného robota, popripade ho dobehne a zaútočí. Rýchlosť je ale potrebné obmedziť. Je povolený pohyb všetkými smermi a tak je pri veľkej rýchlosti robota nutné kontrolovať celú možnú trasu. Tak môžeme zistiť, 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. Potom je pomerne náročne vypočítavať všetky odrazy striel, ktorých môže byť príliš mnoho. Celkove zobrazovanie simulácie sa tam môže dosť spomaliť. Rýchlosť bude teda pevne vymedzená, ale v rámci intervalu s ňou môže každý robot manipulovať.

Ďalej je nutné zadať hornú hranicu pre každú vlastnosť, aby niektorá z vlastností nemohla byť aj blízka nekonečnu. To sa môže stať pretože robota, ktorý má maximálny počet životov, maximálnu rýchlosť a pod. je najlepší, akého môžeme vytvoriť. Nastavenie vlastností 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.

Viditeľnosť bude obmedzená intervalom (0-180), čo sú stupne, pod akými ešte robot vidi. Stupne udávajú pod akým uhlom môže robot vidieť doľava, rovnaký stupeň je potom použitý doprava. Tento koncept sa javí najprirodzenejší, nič však nebráni asymetrickému rozhľadu. Robot tak môže pokryť celých 360 stupnov. 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.)

Viditeľnosť sme obmedzili na konkrétne číslo, čo ju odlišuje od ostatných vlastností. Je možné k viditeľnosti pristupovať i tak, že si užívateľ presne definuje, v akom polomere bude robot vidieť.

Z uvedeného vyplýva, že sa nám rysujú dva typy vlastností. Tie, ktoré nie sú nijak obmedzené a tie, ktoré je nutné obmedziť. Preto budú vlastnosti rozdelené do dvoch sekcií, a to: Prvá sekcia, ktorá nie je nijak obmedzená, obsahuje:

- veľkosť úložiska
- životnosť
- útok strely
- životnosť strely
- útok zblízka
- počet striel

Vlastnosti, ktoré je nutné z rôznych príčin obmedziť sú:

- uhol - uhol 0-180
- vzdialenosť viditeľnosti - táto vlastnosť by mohla byť neobmedzená, ale je v tejto sekcii kvôli tomu, aby vzdialenosť závisela na uhle
- rýchlosť

Bodovací systém znamená, že dostaneme dve čísla, ktoré pre nás predstavujú body. Tieto body prerozdělíme medzi jednotlivé sekcie.

Prvá sekcia nie je neobmedzená priamo, iba technickými parametrami (veľkosť RAM pamäte), ktorými nie je vhodné obťažovať čitateľa ani hráča, preto bude prvá sekcia obmedzená vhodne veľkým počtom bodov- Bolo vybrané 1000 bodov.

Vzhľadom na to, že viditeľnosť je teda obmedzená na 180 (maximálna čiastka, čo tam môžeme dať), môžeme z toho vydedukovať maximálny počet bodov, aký bude možné zadať. Rýchlosť môžeme zadať číslom, ktoré hovorí, koľkokrát je rýchlosť vyššia ako normálna rýchlosť. Tá je 1. V prípade, že prekročí počet únosný implementáciou, je to už problém vykresľovania, ktorý v čase zadania nie je možné určiť, preto naň užívateľ nebude upozorňovaný. Minimálna rýchlosť je 1. Teda maximálny počet bodov pre sekciu 2 je $180 + \text{maxRychlost} + \text{maxViditelnost}$. maxRychlost a maxViditelnost obmedzíme dostatočne veľkým číslom, čo je pri plánovanom

najpomalšom pohybe 50px/s a najrýchlejšom 400px/s je 8 pre rýchlosť a 10 pre viditeľnosť. Maximálny počet bodov je 198.

Užívateľ môže ale nechtiac prekročiť sumárne číslo sekcie pri zadávaní vlastností. Zadavanie vlastností je záväzné pre všetkých robotov, aby ani jeden nebol zvýhodnený, potom je potreba zadané hodnoty upraviť. Rozumným spôsobom sa zda skalovanie hodnôt zo súčasného súčtu na deklarovaný súčet sekcie, pretože zachováva istým spôsobom základnú myšlienku algoritmu (útok na blízko veľiky, útok na ďaleko malé, pretože nebude zbytočne strieľať na diaľku).

Rovnako užívateľ môže podceňiť rozdelenie bodov. Vtedy nastavujú dve možnosti, buď to bolo zamerne (zamerne slabý robot so stratégiou "aj tam na nezničiť"), alebo robot, ktorý bol vytvorený pri znalosti iného rozdelenia bodov. V tom prípade sa dá uplatniť tiež skalovanie. Ktorý z týchto rozhodnutí sa použije, by malo byť na tvorcu algoritmu.

Algoritmus je potrebné napísať pomocou príkazov robotstiny. V zájmu zachovania spravodlivosti by sa roboti mali chovať tak, aby ani jeden z nich nebol zvýhodnený, nevykonával 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 subežne a nemuseli by sme vykonaných častí kódu nijak kontrolovať. Pre každý algoritmus by sa dalo využiť paralelné vlákna, takže paralelizácia by prebehla na najnižšej úrovni. Takéto rozhodnutie obsahuje niekoľko problémov. Pri jednojadrových procesoroch sa vlákna striedajú na základe prideleného časového kvanta a tak by sa algoritmy tak či tak realizovali sekvencne. Tento spôsob má navyše tu nevýhodu, že sa čas, ktorý jednotlivé vlákno reálne dostane, závisí na výťažnosti procesora, softwarových a hardwarových prerušení a pod. takže nemožeme zaručiť spravodlivosť.

Použitie multiprocesora princíp spravodlivosti mierne vylepší. Vznika 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 vykonával približne rovnakú časť kódu algoritmu.

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 algoritmu, nazveme časové kvantum. « j vystiznejšie "časová dotácia"» Je pre všetkých rovnaké, kvôli zaisteniu spravodlivosti. Proces, keď roboti toto kvantum využívajú, nazveme kolo. Kola sa periodicky musia opakovať, aby mohol prebehnúť celý algoritmus. V opačnom prípade 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 slovo « j (aktivitu?). Nazveme 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 častí algoritmu na-

raz. Robot teda striktne vykona prave jednu, rovnako velku cast svojho kodu a preda slovo dalsim robotom. Tento sposob prinasa

Obmedzenie casom

Planovac tohoto typu priradi kazdemu robotovi cas, za ktorý moze vykonavat svoj program. To znamena, ze na rozdiel od predchadzajuceho typu je mozne postupit v programe dalej v jednom kole.

Robot by mal vykonavat svoj algoritmus dovtedy, pokiaľ ho niekto externe nezmeni. To znamena, ze algoritmus sa musi opakovane vykonavat potencialne do nekoneca. No nie je vhodné, aby na to daval pozor sam hrac, Jednak musel opakovanie deklarovať u kazdeho svojho naprogramovaného robota a jednak tato deklaracia je iba manualny zapis, ktorý nema vplyv na pozitu strategiu. Preto sa v práci dba na to, aby v okamihu, keď by robot mal skončiť sa jeho algoritmus spustil odznova v nekonecnom cykle.

Doteraz sme mlčky predpokladali, ze kazda cast kodu ma rovnaku vahu. Tym, ze niektore casti algoritmu vyhlasiame za tazsie spravitelne, potom vytvorime novu instanciu sveta. Tam sa nemeni obsah, ale sposob narabania s algoritmom (FUJ). Hrac potom musi vymyliet taky algoritmus, ktorý pocita s danym nastavenim. Potrebujeme ale vediet, ze to skutocne prispeje k atraktivnosti hladania vhodnej strategie. Rozoberme si teda, ako to bude vplyvat na algoritmus, ak by jednotlivé casti boli iba ocenené, t.j. trvali iny počet kol. Potom by bol uzivatel nuteny pouzít taky algoritmus, ktorý dlho trvajuce casti pouziva minimalne. Napríklad ak by robotovi trvalo styri kola na to, aby sa otocil, potom hrac sa pravdepodobne bude snazit vyuzít minimalny počet otocení. Tym sposobom sa moze vymyslanie strategii posunúť na hlbsiu uroveň, co vyhovuje naroku na prácu. Vysledne planovace potom dostanu ine vlastnosti,

2.2.2 Mozne pristupy k programovaniu sveta

Doležitou castou práce je predstaviť spôsob, akym bude uzivatel zapisovat vymysleny algoritmus. Uvazujeme s nasledujucimi jazykmi:

Graficky jazyk

Pod grafickým jazykom rozumieme jednoduche graficke zobrazenie zapisu algoritmov. Tento sposob sa najskor vyuzival pri výuke programovacích jazykov ako ľahký a jednoduchý spôsob písania algoritmu. Jednoduchým sledovaním sipliek (OBR) sa da zistiť, v akom stave sa program nachadza pri pociatocných podmienkach. Implementovanie vlastného grafického jazyka, kde by boli prikazy len pre potrebu programatora, by bola dost narocna. Jediný nástroj, ktorý splnoval zakladne predstavy, bola ale Microsoft Visual Language, ktorý je pre nekomercne vyuzitie zdarma. Bohuzial je závisly na operacnom systeme, takže sa ukazal ako nevhodny

scriptovací jazyk LUA LUA je scriptovací jazyk, ktorý sa vyuziva prave na programovanie problemov umelej inteligencie, co je aj nas pripad. Jeho výhodou je, ze tento jazyk je jednoduchý, volne siritelny a prenositelny. [REF]

LUA je však konštruovaná na kompletne prevedenie daného algoritmu, takže by sme nemali priamo kontrolu nad jednotlivými časťami kódu. Napríklad kontrola obsadeného uložiska, ktoré môže robot používať, by sa skomplikovala. Rovnako rozhodnutie, kedy algoritmus vykoná dosť príkazov a je nutné ho pozastaviť. Preto nebol použitý jazyk LUA.

vlastný jazyk Ďalšou možnosťou je vytvoriť vlastný Domain Specific Language, ktorý bude použiteľný len na úlohy typu Codewars. Vytvorenie zahŕňa definíciu vlastného jazyka v spôsob, akým sa budú jednotlivé príkazy interpretovať, komunikovať s robotmi. Rozhodli sme sa preto pre tento koncept.

2.2.3 Robotstina

Algoritmus bude teda zapísaný vo vlastnom jazyku - robotstine. To nám kladie netriviálne nároky.

minimalne schopnosti

Najskôr sa zameriame na to, 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
- cakanie (ako opak pohybu - ničnerobenie istý počet kol)« na zmenu sveta?
- útok
- otocenie sa
- získavanie informácií o objektoch z okolia

Tieto príkazy podmieňujú vznik operácií s objektami.

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

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

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

- *null* pre oznamenie robotovi, ze nema ulozeny ziadny objekt
- *this* pre referenciu svojho vlastneho objektu

Operacie s cislami sa objavuju v suvislosti s premennymi. Budu podporovane iba niektore zakladne operacie s realnymi cislami, a to :

- scitanie
- odcitanie
- nasobenie
- delenie, pricom vysledok delenia celych cisel bude realne cislo
- modulo pre cele cisla = %

Ostatne aritmeticke operacie nebudu podporovane, nakoľko nie su nevyhnutne potrebne. Aby sme docieli delenie vysledok delenia v celom cisle, bude podporovana automaticka konverzia do celcel cisla z realneho. Uzivatel toto nemusí specialne osetrovat.

Operacie s objektami davaju moznost reagovat na svet t.j. umoznuju zistit:

- akym smerom je objekt otoceny (vhodne pre robota, aby zisital, ci nie je na muske) *getDirection(o)*
- akym smerom sa objekt pohybuje *isMoving(o)*
- ci je to strela, stena alebo robot -*isMissile(m)*, *isWall(o)*, *isPlayer(p)*, *isEnemy(o)*
- ci bol nepriateľsky robot zasiahnutý - *isHit(robot)*

Relacne operacie $>$, $<$, $=$, \neq a ich kombinacia pre vsetky typy premennych. Využívajú sa napríklad pri zistovaní najbližšieho objektu atď. Pre objekty je nezmysel porovnávať, či je menší alebo väčší, preto u objektov je povolené iba relacný operátor $=$ a \neq .

Podmienky výrazne prispievajú k eliminovaniu pre robot neprijemných udalostí, ktoré môžu vo svete nastávať (ak je tam mina, tak tam neslapni). Viazu sa k nim ďalšie kódové slová TRUE (1) a FALSE (0).

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

Procedury a funkcie umoznuju sprehladnovat a clenit kod. Prispievaju k lepsiemu kodovaniu algoritmov, vyuzivaniu pamate, ktora sa po skončení procedury uvoľní. Parametre k funkciám sa dajú predávať odkazom alebo referenciou. Definovanie predávania parametra referenciou je značené kódovým slovom *ref* pred premennou, ktorá je takto predávaná. Poradie parametrov predávaných odkazom a hodnotou nehra rolu. Parameter s preddefinovanou hodnotou nie je podporovaný

Definovanie cielov a vlastnosti reprezentuje chovanie robotuv.

Premenne sa ukladaju do pamati. Ak uz nie je volne miesto v pamati, bolo by nemile, aby preto robot umrel. Rovnako je to nefer aj voci robotom, ktory ho lovia. Namiesto toho bude premenna ukazovat na miesto v pamati, ktora je uz obsadena. Algoritmus tak moze efektivne prepisat hodnotu, na ktory sa spolieha (napríklad hodnotu TRUE) ale to zodpoveda tomu, ze sa robot z nedostatku pamate zblaznil. Obzvlast to je viditelne, ak si prepise premennu, na ktoru sa spolieha, napríklad hodnotu TRUE bude zrazu FALSE.

Aby sa predišlo tymto neprijemnostiam, je vhodne vediet, kedy premenne vznikaju (v zmysle obsadzujú pamat) a kedy zanikaju (uvolnujú pamat).

Pri deklarovaní premennej sa premenna automaticky vytvori, čo je normálne chovanie zname takmer vo všetkých vyšších programovacích jazykoch. Pri volaní funkcie s parametrami sú tieto parametre kopirované, pokiaľ neboli definované kodovým slovom `ref`. To znamená, že sa vytvoria znova všetky premenne a prida sa im hodnota, s akým je funkcia volaná. Navratová hodnota sa vytvori v ukamziku volania `return`. V prípade, že ide o proceduru, navratová hodnota sa nevytvori. Premenna sa môže dočasne vytvoriť aj v bloku označenom `.` Po ukončení bloku sa premenna v uložisku uvoľní. Preto sa v nasledujúcom kóde (mimo bloku) nemala dáť použiť. Premennu s rovnakým menom nie možné vzhľadom na implementáciu vytvoriť ďalej s iným typom než bola prvý krát deklarovaná. Toto ale nie je závažný problém, keďže sa jedná iba o vytvorenie názvu pre premennu a rôzne mená s rovnakým typom iba majú neskorsieho citateľa.

Premenne obsadzujú miesto v uložisku. Vzhľadom na obmedzenie uložiska je namieste určiť, aké množstvo jednotiek uložiska jednotlivé premenne zaberajú. premene typu `Object`, `Real` a `Integer` budú zaberat 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 pamati bolo jednoduché a pochopiteľné. Ďalej tu je položka `Location`. Tá má z definície dve zložky. Preto v pamati bude zaberat 3 jednotky, pre 3 premenne typu `integer`, ktoré sú jej zložkami a jednu pre seba. Podobne obsadzujú pamat zložené prvky (polia). To znamená jedno pre samotnú premennu a potom súčet obsadenia všetkých jeho premenných.

Syntax jazyka

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

vlastnosť je jedno z:

- `hipoints`
- `attack`

- mAttack
- mHealth
- angle

nasledovane cislom, ktore definuje vlastnost. V pripade, ak uzivatel pocet bodov v jednotlivej sekcii prektoci urcene cislo, vlastnosti sa budu skalovat.

Ciele algoritmu sa sklada z:

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

Prikaz na robota je prikaz z mnoziny *step(X), wait(X), see(), eye(X), turn(X)* $X \in N, seeEnemy()$

Prikaz na informaciu je prikaz z mnoziny (*Locate(o), isWall(o), isPlayer(o), o* \in *Objekt, seen[n]* $n \in N$)

priklady pouzitia

Nasledujuce priklady vysvetluju, ako sa pouzivaju jednotlivé prikazy, nemaju vsak za ulohu demonstrovat skutocny kod

```
robot R1 {
    attack 20
    mAttack 100
    hitpoint 60
    memory 10
    kill R1
    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 uvidíme mechanizmus ovládania robota pomocou príkazov ??

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 rozlíšiť, 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 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. Je možné napríklad sa striedať, keď interpret prejde určitý počet slov. To ale nie je spravodlivé. Napríklad 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.

Ďalsou možnosťou je použiť kompilér nejakého skutočného programovacieho jazyka. To znamená upraviť jazyk, ako sme ho predstavili do podoby vhodnej pre robotštinu. Možno tu definovať 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 vytvoriť aj vlastnú *virtual machine*, ktorý bude jednotlivé preložené časti v

medzikóde interpretovať. Inšpiráciou pre takúto metódu bola Java virtual machine ???. 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 znamená opäť obtiažnu správu pamäte robota.

2.3.2 Práca s jazykom

Bolo teda rozhodnuté, 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 ???.

Najskôr bolo nutné si premyslieť formát medzikódových inštrukcií. Jedným z najčastejších foriem je trojadresný, štvoradresný alebo formát zásobníkového čítača. Trojadresný a štvoradresný znamenajú, že medzikódové inštrukcie operujú nad dvoma alebo tromi operandami a majú ešte ďalší ukazateľ na to, kde ukládajú 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, viz *OBR*

Nastáva otázka, čo sa do tohto zásobníka ukladá. Je možné sa na tento zásobník pozeráť 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 zásobník 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), ktorý zodpovedá požiadavkám nasledujúcej inštrukcií. To je ale dosť neefektívne. Tento spôsob má problém aj pri premenných zloženého typu. Tento neprijemný efekt môžeme odstrániť implementovaním polymorfného zásobníka. To znamená, že sa zásobník 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 zásobník 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 načítaní premennej by robot dostal inú hodnotu ako očakávanú, čo je tiež spôsob prepisu. Problémom tejto metódy je, že počas života algoritmu bude v úložišti často rôzny počet rôznych premenných (pri odchode z funkcie sa niektoré premenné typu Integer zničia a pri zavolaní následnej funkcie sa na uložišti objaví premenná typu Integer). Takto sa bude musieť úložisko dynamicky meniť na úrovni programu. To znamená častú alokáciu a dealkáciu premenných za behu algoritmu. To je dosť neefektívne obsadzovanie pamäte, keďže vieme presne, aké veľké úložisko bude robot mať.

Preto bol navrhnutý nasledovný spôsob : Každá premenná bude obsahovať všetky základné typy a v prípade, že ide o pole, aj 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 pamäť. Výsledný efekt je ale uspokojivý.

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

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 ??.

Je potrebné rozhodnúť, aké inštrukcie bude medzikód používať. Tie vyplývajú zo syntaxe robotštiny a ďalej sa s nimi budeme zaoberať až v sekcii o vykonávaní medzikódu.

Preklad jazyka

Sekcia popisuje, aké štruktúry sú nutné pre generovanie a uloženie jazyka. Vynecháme spôsob definovania vlastností 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. Instruction.

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 nám 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 funkcii, sa budú ukladať pod menom $F\#$ premenná. $\#$. Tento prístup bol zvolený, pretože nie je povolený znak $\#$ v jazyku. Oddeľuje názov funkcie/procedúry, kde bola definovaná premenná a tak je možné ľahko spätne rekonštruovať jej názov.

Problém nastáva s premennou, ktorá bola definovaná vnútri nejakého bloku. Potom ju poznáme pod týmto menom a s už daným typom.

Pre? zisťovanie, ako a kde sa v algoritme používajú premenné, je nutné vyhľadávať podľa reťazca. Minimálne pri parsovaní by bolo vhodné nájsť takú štruktúru, ktorá nie je z najpomalších (lepšia ako lineárne prechádzanie zoznamu). Ponuka sa hash-mapa, ktorú štandardne obsahuje STL (Standard Template Library). Pretože sa predpokladalo (v záchvate paniky), že bude nutné premenné vyhľadávať aj počas vykonávania programu, bol zvolený Burst trie??. Toto rozhodnutie sa žiaľ neskôr ukázalo ako predčasné z dôvodov, ktoré budú jasné pri objasnení práce interpreta medzikódu. Z dôvodu fungovania tohoto kusu kódu bolo ale ponechané.

Podľa 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, keď 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 vykonávanú inštrukciu.

Súčasťou medzikódu su 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 vykonávaných inštrukcií. Na rozdiel od príkazov je nutné si zapamätať, odkiaľ bol spravnený skok, aby bolo možné sa vrátiť a pokračovať. Teda namiesto jedno PC budeme potrebovať opäť zásobník PCciek.

2.3.3 Interpret medzikódu

Otázkou, ktorou sa v tejto časti budeme zaoberať, je, na aké časti má byť algoritmus rozkúskovaný, a aby plánovač, ako bol popísaný, korektne vykonal príkazy a vrátil očakávaný výsledok. Prioritou je teda rozdeliť kód na čo najmenšie časti. Tu su nasledujúce robotické inštrukcie :

InstructionCreate pridá premennej priestor v úložisku

InstructionLoadVariable na zásobník s hodnotami pridá hodnotu premennej

InstructionLoadElement zo zásobníku vyberie $n \in N$ a premennú typu pole. Na zásobník pridá n -ty element tohoto poľa

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

InstructionConversionToReal zmenu premennú typu Integer na Real

InstructionDuplicate zduplicujú hodnotu na zásobníku

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

InstructionStoreInteger, InstructionStoreReal, InstructionStoreObject uložia načítanú príslušnú hodnotu do premennej na vrchole zásobníka.

InstructionCall spôsobí uloženie PC a založenie nového

InstructionPop zruší 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 skočiť.nie skončiť?

InstructionContinue podobne ako break, ale case prekladu je jasne, kam skočí

InstructionReturn vytvorí návratovú hodnotu

InstructionRestore obnoví stav, aký bol pred volaním funkcie

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

Operácie vezmú dva prvky a na zásobník vložia výsledok operácie

aritmetické operácie pre Integer a Real

binárne a logické OR a AND

InstructionNot vezme celé číslo zo zásobníka a ak je to 0, vloží tam 1, inak 0
inštrukcie rovnosť a nerovnosť? pre objekt vezme dva prvky zo zásobníka a vloží tam výsledok operácie. Objekty nemajú relačné operátory na nerovnosť.

Relačné operácie pre Integer a real vezmu dva prvky zo zásobníka avložia výsledok porovnania (0 = npravdivé tvrdenie, 1 inak)

InstructionBegin nastaví príznak, že začal nový blok, kvôli premenám a ich neskorším dealokáciám

InstructionEndBlock vyčistí pamäť od premenných definovaných v tomto bloku

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

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

InstructionFetchState uloží na? zásobník výsledok poslednej akcie, ktorú robot robil (pohyb, strelba)

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

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

InstructionShootAngle vezme zo zásobníka celé číslo a vystrelí v danom smere

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

InstructionTurnR robot sa otočí doprava o 90°

InstructionTurnL robot sa otočí doľava o 90°

InstructionHit zoberie zo zásobníka objekt a uloží jeho životnosť

InstructionLocate vyberie zo zásobníkov objekt a uloží jeho? pozíciu (ak je robotom viditeľný? a , inak uloží $[-1, 1]$)

InstructionIsXXX vyberie zo zásobníka objekt a uloží 0/1 v závislosti na vlastnosti objektu v premennej zobratej zo zásobníka. XXX môže byť missile, wall, player...

InstructionTarget dá na zásobník cieľové miesto

InstructionSaveVariable uloží premennú mimo zásobník. Táto inštrukcia vznikla kvôli priradzovaniu 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 priradzujeme $A = B$, kde A,B sú typu `integer[6][2]`. A sa rozloží na 12 integerov v poradí `integer[0][0].integer[0][1]` atď., postupnosť takýchto 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 zásobník

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

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

V prípade niektorých inštrukcií, ako napríklad **InstructionBreak**, nie je v čase analýzy zrejmé, kde má 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 čítaní 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 obsahovať 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 zostávajú rovnaké, stačí nám ako štruktúra opäť zásobník. Pri uvoľňovaní 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 vašom prípade nám ale nezáleží na tom, kde sú premenné uložené, nijak to neovplyvňuje výkonnosť. Teda v pamäti si stačí udržiavať informácie o poslednom obsadenom prvku. Nasledujúce voľné miesto sa nájde tak, že lineárne sa budú prechádzať 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

Implementacia

Tato kapitola pojednava o konkrétnych implementacných technikách, ktoré vyplývajú zo záverov analýzy.

Aplikácia sa skladá z troch hlavných častí, ktoré sa dajú používať samostatne. Sú to:


- System okien
- Mapa sveta
- Preklad napísaného algoritmu do medzikódu
- Modul pre generovanie diskretného bludiska

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

3.1 Sprava okien

Ako grafická knižnica bola zvolená SDL *REF*, ktorá bola vyvinutá pre rýchle zobrazovanie a je jednou z najčastejšie používaných knižníc pri tvorbe 2D hier. *< REF >*. Aplikácia bude používať minimálny užívateľský vstup z klávesnice. Jediný vstup rozsiahlejšieho charakteru je napísanie samotného algoritmu. Pre užívateľa v Unixe, zvyknutého na svoj obľúbený editor, makrá a zkratky, bola ponechaná možnosť vytvoriť si v ňom aj algoritmus.

Vzhľadom na to, že zvládanie okien, ktoré má aplikácia zvládnuť, má tiež minimálne požiadavky, bolo v práci použité vlastné rozhranie spravy okien a nebolo nutné prikompiľovať žiadnu dodatočnú knižnicu, ktorá by sa zaoberala užívateľskými vstupmi.

V zdrojovom kóde to bolo realizované pomocou triedy **Window** a potomkov abstraktnej triedy **Menu**. System práce triedy **Window** je naznačená na obrázku . Ide o to, že ak sa má nastaviť v obrazovke nové menu, potom treba oknu, ktoré instanciu triedy **Window** reprezentuje, povedať, aby toto menu pridal do **ZASOBNIKA**. Keďže okno je v SDL povolené maximálne jedno, instanciu triedy **Window** bude práve jedna. Preto ďalej v texte budeme pod označením `bfWindow` myslieť práve túto jednu

instanciu a nie triedu.

Pomocou ZASOBNIKA na pridavania menu je zrejmé, do ktoreho menu sa program vráti po zrušení obrazovky so základným Menu. Zrušenie obrazovky sa prevedie zavolaním metódy `pop()`. Táto metóda odoberie menu zo zásobníka a pomocou metódy `Menu::clean` uvoľní zložky potrebné pre vykreslenie. Efektívne sa tak setí pamäť. Pre potomka triedy Menu musia byť implementované nasledovné metódy:

Init pre inicializáciu dynamických položiek nutných k správnej funkcii príslušného menu (nacítanie obrázkov, vygenerovanie písma a pod.)

Draw pre vykreslenie celého okna

Process , ktorý je volaný v nekonečnom cykle pre kvôli spracovaniu udalostí z klavesnice

Resume uvedie menu do pôvodnej podoby

Resize pre prípadné zmeny rozlíšenia okna na prepocítanie premenných určujúcich pozíciu

Clean pre deinicializáciu dynamických premenných pre čo najmenšie množstvo alokovanej pamäte. V tomto ohľade sa táto implementácia, čo sa obsadenej pamäte týka, dokonca predčí implementáciu pomocou grafickej knižnice QT. QT je zameraná na užívateľskú prívetivosť. Jednotlivé formuláre sa vytvárajú pomocou `WIDGETOV`(prvkov), ktoré sú vo formulári neustále prítomné.

Trieda Window potom pri žiadosti o pridanie menu do ZASOBNIKA inicializuje menu, ktoré sa má pridať a v prípade úspechu inicializácie ho pridať a vykresliť. Vďaka cyklickému volaniu metódy `process` objektu na vrchole zásobníka je zaistené, že reakcia na udalosť (klavesnica, myš) sa bude týkať práve nasadeného menu.

3.2 Svet

Svet pozostáva z objektov. Každý konkrétny objekt je potomkom abstraktnej triedy `Object`. Navzájom riešia výsledky kolízií, výsledky príkazov na zistenie pozície a pod.

Súčasťou objektu je aj spôsob, akým sa objekt vykresľuje na plochu, čím sa zaoberá trieda `ImageSkinWorker`.

Obrázky reprezentujúce jednotlivé objekty sú najskôr nacítané triedou `Skin`. Počet obrázkov je presne N , kde N je počet stavov, aké môže objekt dosiahnuť. Stavy sú nasledujúce:

defaultne

Stav, ktorý má objekt, keď sa nič nedeje.

- cakajuci objekt
 - spiaci objekt. Tento stav sa nastavi v okamziku, ked je objekt prilis dlho v defaultnom stave
- trvale • pohybujuci sa
- cakajuci
- docasne • zasiahnuty
- utociaci

Pre niektore objekty, ako je napríklad stena, je definovanie tychto stavov velmi vela, preto rozne objekty maju svoju vlastnu 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 obrázku mení kvôli animácii, smerom dole sa vykresľovaný obrázok mení vzhľadom na to, ako má byť objekt otoceny. Potrebujeme pritom vedieť veľkosť obrázku, ktorý sa má vykresľovať. Veľkosť obrázku nemusí súhlasiť s veľkosťou obrázka, ktorá spôsobí kolíziu. Preto je v každom adresári, odkiaľ sa nahrávajú obrázky, este subor config. Subor hovorí, ako je vykresľovaný obrázok veľký, odkiaľ začína kolízny obdĺžnik a aké má rozmery. ¡OBR¿

Trieda ImageSkinWorker potom narába s nahranými obrázkami. V jej vnútornej interpretácii sa nachádza este aj minizasobník, ktorý hovorí, ktorý zo stavov "trvalý", "dočasný", "default" je aktívny. Dočasný stav má vždy vyššiu prioritu ako trvalý a trvalý vyššiu ako default. Stav sa nastavuje pomocou metódy *SetState(State, whichState)* s parametrami nadobúdajúcimi hodnoty State [] a whichStat [].

Objekt je základny prvok sveta, všetky ďalšie typy vzniknú len jeho dedením.

Pohyb sa v objekte rieši nastavením počtu pixelov, ktoré má prejsť. V okamihu, keď je nastavený nenulový počet pixelov, objekt sa pri volaní metódy *move* pohne svojou rýchlosťou zadaným smerom a odcíta príslušný počet pixelov. Po skončení pohybu sa zavola metóda *endMove()*, ktorá odstráni permanentný pohybujúci sa stav. U robota je situácia trochu zložitejšia, pretože potrebujeme naviac vedieť, čo už náhodou nestúpil na políčko, 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 prípade, keď do nej niečo narazí, sa musí chovať ako útočník. Ak by ale útočníkom bola opäť strela, výsledkom by bolo zacyklenie. K takej situácii ale nikdy nesmie dôjsť, čo ošetruje štruktúra zastrešujúca objekty.

Objekt, ktorý posobením útoku zomrie, registruje, kto smrť priamo spôsobil. Je to zabezpečené premennou "Owner", vlastníka objektu. V prípade, že je obeťou robot, má potom robot možnosť poslať správu útočníkovi (vlastníkovi strely alebo tela robota). Tak sa robot dozvie, že

niekoho zabil a zisti, ci sa tym nespnila podmienka pre ukoncenie simulacie. Objekt sa nasledne prida do struktury, ktora zhrna mrtve objekty pre neskorsiu dealokaciju.

Robot na rozdiel od obycajneho objekty ma aj dalsie metody. Menovite je to *shoot()*, *see()*

Strielanie robota je jednoduché, pokiaľ je známy smer, v ktorom má robot strieľať. Znamená to iba pridanie objektu na miesto, kde nebude kolidovať s robotom, ktorý ju vypustil.

Robotové videnie je podobne realizované pomocou triedy *Seer*. »¿ TAK JE TO skoro vždy Tej sa predhodia objekty, ktoré má spracovať.«¿ navrhujem vyhodit K objektom, ktoré boli ohodnotené ako viditeľné možno pristupovať pomocou metódy *getObject(int index)*

Vzájomné vplyvy objektov sa rieša len medzi objektami samotnými. Jedine, čo treba zistiť, je ktorý objekt má vplyvať na ktorý. Takouto zastrešujúca štruktúra je mapa.«¿ Digraf bez slučiek, kde vrcholmi sú objekty a orientované hrany väzby medzi nimi

Mapa je riešená ako dvojrozmerné pole oblastí (obdĺžnikov), kvôli kolíziám, ako sme spomínali v ?? Oblasť je ohraničená pociatocnými susedníkmi, 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 kolízie to prináša nepríjemný jav, pretože je nutné kontrolovať všetku oblasť, kde by sa daný objekt mohol vyskytovať. Mapa ďalej obsahuje startovacie políčka – užívateľom definované významné políčka, ktoré sa dajú neskôr definovať ako cieľ.

¡OBR?¿

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 neskorsim pocítaním 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 v 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 pridané. Potom pri dealokovaní mapy mapa samozrejme nemá možnosť zistiť, či tieto objekty boli skutočne dealokované, alebo či si ich instancie vytvorila samostatne. Preto ticho predpokladá druhú možnosť a pri skocení na všetky prvky, ktoré zostali v mape, sa použije *delete*. Ak bol prvkom mapy objekt, s ktorým programátor ešte počíta, musí ho najskôr z mapy odobrať.

Program podporuje uloženie a opätovné nahranie mapy. Keďže mapa tieto objekty iba zobrazuje, postaci pre každý objekt zistiť, čo to vlastne je. To kladie nárok na jednotlivé objekty, ktoré si musia pamätať uni-

katne cislo oznacujuce tento typ. Mapa potom pri nacistani sveta podla typu vytvori objekt. Dalsimi polozkami, ktore je nutne uchovat a ktore nie su objekty, su startovne policka a miesta cialov. Tieto sa ukladaju osobitne Mapa musi dbat na to, aby objekty, ktore obsahuje, nepresli mimo vykreslovanej plochy. Aby sme nemuseli kontrolovat objekt pri kazdom pohybe a okrem kontroly kolizii kontrolovat aj pripustnost pozicie, su ku kazdej nahravanej mape pridane steny okolo celej mapy. Po vytvorení prostredia (mapy, sveta) sa v mape nenachadzaju roboti, pretoze nie podporovany ziadny dalsi mechanizmus na ovladanie tela robota (nic by sa nedialo).

Mapu je mozne okamzite vykreslit a objekty neskôr az neskôr. Mapa dalej nijak nevyhodnocuje vysledok kolizie, sluzi iba ako nastroj na ich detekciu. Detekuje koliziu iba pre objekty, ktore sa pohybujú, t.j. pre kazdy pohybujúci sa objekt prejde lineárne všetky objekty v okolí, ci nastala kolizia. V okamziku, keď objekty skoliduju, mapa vyberie najblizsi objekt, s ktorým sa objekt skolidoval. Pod kolidaciou rozumieme prekrytie obrazov dvoch objektov. Kedze mame kombináciu dvoch objektov, ktore spolu nekoliduju, je zaistit riesenie aj pre takuto kombináciu. Na implementacnej úrovni to znamena, ze sa bud:

- zavola aj tak riesenie kolizii u objektu. Az objekt, ktorý bol zasiahnutý, spozna, ze ku kolizii vlastne vobec nedoslo. To ma nevychodu, ze objekt, ktorý sme kontrolovali na koliziu, zostane na mieste, ktorý mu pohyb urcil. Kedze ale kontrolujeme iba najblizsi kolizny prvok, moze sa stat, ze aj po vyrieseni kolizie bude prvok stale kolidovat
- nedojde k volaniu riesenia kolizii medzi objektami. Objekt nebude uvazovany pre koliziu. Toto riesenie bolo zvolene, kedze ma najlepsie vysledky. Pre kontrolu toho, ci objekt ma alebo nema by uvazovanu pre koliziu, bolo dany priznak Solid. Objekt, ktoreho logicky sucet (or) zasiahnuteľnosti s uvazovanom objektom ma priznak solid (aspon jeden z objektov ma solid = true), je uvazovany pre koliziu

Mapa ako vykreslovacia struktura musi poznat, kedy by bolo vhodne objekt vykreslit a kedy naopak nie. Vyrazne to setri procesorovy cas. Preto bola u kazdeho objektu implementovana metoda *changed()*

3.2.1 Interpret a jazyk

Ako bolo zmienene, jazyk sa generuje pomocou parovacich nastrojov Bison a Flex. Tieto nastroje umoznuju velmi lahko menit syntax jazyka a su lahko udrzovatelne.

Robotov moze byt niekoľko a vďaka cieľom, ako je KILL, je mozne sa na nich odkazovat pomocou mena. Preto bolo nutne vytvori zastresujucu strukturu. Tou je trieda Robots. Ta potom musi obsahovat:

- pocet bodov, ako bol defaultne definovany. Toto defaultne nastavenie sa tam potom da jednoducho priradzovat vytvorenym robotom.

- pole generovaných robotov
- informácie o prave vytváranom robotovi
- zoznam nevyriešených požiadaviek na zabitie robota
- zoznam nevyriešených požiadaviek na miesto na mape

Novovytvorený robot sa skladá z týchto častí:

- názov robota pre definovanie cieľu KILL
- použitý plánovač
- prázdnu štruktúru na definované typy
- prázdnu štruktúru pre medzikód
- prázdne informácie o najdených chybach
- trie pre premenné
- pomocnú štruktúru pre spracovanie medzikódu
- chybové kódy
- štruktúra pre prácu s medzikódom (Core)

Algoritmus bude zapísaný v textovom súbore. Z neho sa pomocou parsovacích nástrojov vygeneruje kód, ako je popísaný v ???. V prípade, že nastala chyba, bude potom táto zapísaná pomocou stringu. Výsledný medzikód je potom zapísaný v XML-súbore, ktorého názov je kombinácia názvu robota a vstupného súboru. Napríklad ...Technologia XML bola vybraná ako jedna dobre zrozumiteľných dostupných zobrazení.

Po dogenerovaní všetkých instrukcií sa na záver pridá ešte jedna instrukcia, ktorá spôsobí opakovanie celého programu. (instructionMust-Jump). Po správnom vygenerovaní je robot schopný vykonávať program pomocou metódy `execute()` alebo `action()`. Tu sa prejaví vplyv plánovaca. Robot má vlastnú instanciu plánovaca, každá instrukcia si vnútorne drží číslo skupiny, do ktorej patrí. Jednolivé názvy skupín sa volajú intuitívne podľa instrukcií. Metóda `action()` volá plánovaca. V okamžiku, keď plánovač povolí vykonanie instrukcie, do plánovaca sa zanesie, ako moc je instrukcia hodnotená a následne je táto vykonaná, Vykonávanie instrukcií pozostáva z volania metódy `execute(Core *)` , kde trieda `Core` poskytuje prostriedky pre interpretovanie medzikódu. Trieda `Core` tiež obsahuje telo robota, ktorým sa bude hybať po mape uložisko. 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 prípade, keď zostal sám a cieľom nebolo len zničenie ostatných vráti metóda `action` `false`, ak hráč umrel, inak vráca `true`. Je vhodné ale súčasne kontrolovať splnenie podmienok. Metóda `action` nemože vrátiť viac premenných naraz, preto má navyše parameter odovzdávaný referenciou.

3.2.2 Generovanie bludiska

Generovanie bludiska zavisí na objektoch, ktoré bude výsledná mapa obsahovať. Je to kvôli kolíziám, ktoré sú závislé na veľkosti obrázku. Raz vygenerované bludisko pevne určí pozíciu objektov, preto je závislé na konkrétnych veľkosti objektov sveta. Navrhnutý algoritmus vygeneruje len diskretne bludisko rozdelené na políčka pevnej veľkosti. Vygenerujú sa z dôvodov vysvetlených v analýze len základné steny. Teraz potrebujeme vhodne tieto pevne steny z diskretneho bludiska preniesť do nášho sveta. Aby sme zachovali pomer rozostavenia, ako je vo vygenerovanom bludisku, steny sa rozmiestnia rovnomerne.

Kapitola 4

Porovnanie

Existuje mnoho nástrojů, které se věnují šutazne algorimom zameraným na programovanie robotov. Z typických zastupcov mozeme uviesť už spomínaných PO-GAMUT, ARES, RoboCode, ako aj neuvedené C++ robots, MindRover, Grobots a množstvo ďalších ???. Vacsinou sa zameriavajú na prezitie v arene. Podľa corewar terminologie sa tento cieľ nazýva "King Of The Hill" (ďalej KotH)

Stručne porovname výsledný program bakalárskej práce s uvedenými aplikáciami, zhrnieme výhody a nevýhody ich prístupov a čo nové prinášame. 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 je veľmi podobná hra o prezitie. Roboti sú písaní v C++ a majú k dispozícii jedinu zbraň kanon. Svet je ale veľmi jednoduchý, je to priestor 100x100 m ohraničený stenami. Celá hra je na rozdiel od codewars koncipovaná ako turnaj, robot si postupne vyberá súperov. Codewars navyše ponúka možnosť pustiť všetky algoritmy naraz s tým, že hráč môže využívať aj ostatných robotov, aby za neho spravili špinavú prácu.

ARES je typická obranno-útočná hra, ktorá je ale hlavne určená programátorom. Svet sa mení veľmi dynamicky a roboti nevedia vôbec nič o svete. Len si sa domnievajú výsledky útoku a podľa toho reagujú. Codewars je z tohto pohľadu úplne opačná aplikácia, Roboti vedia veľmi dobre mapovať svet a spoznávajú nepriateľa. Spoločný prvok tak môžu mať len v spôsobe interpretácie algoritmu. V Codewars bol tento koncept dovedený ad absurdum, keď sa konfiguráciou môžu meniť chovanie napísaných algoritmov.

RoboCode je tiež zložitá hra, ktorá sa dlho vyvíjala. Je určená pre užívateľov, ktorí začínajú s programovacím jazykom Java. Svet sa riadi svojimi vlastnými pravidlami pre strelbu a víťazenie. Víťazný algoritmus je taký, ktorý získá najviac bodov. Codewars bolo silne inšpirované týmto programom. Líšia sa ale významne v spôsobe, akým vykonávajú algoritmus (RoboCode sa snaží o paralelizáciu) s definovaním cieľa. Codewars oproti Robocode prináša možnosť prispôsobiť si robota pomocou vlastností jeho algoritmu a s možnosťou napísať aj čisto mierumilovného no chyt-

Tabuľka 4.1: Charakteristiky porovnavaných hier						
	Svet	zbrane	ciele na algoritmus	obmedzenia	pohyb	ob
POGAMUT	3D	strely	rozne	ziadne	komplexny	mn
ARES	1D	zdielana pamat	KotH	velkost pamate	vykonanie instr.	č
Codewars	2D	strely, roboti	volitelne	volitelne	jednoduchy	varia
C++Robots	2D	strely	roboti	KotH	ziadne	jedn

reho robota.

Koncepty prijate v jednotlivých hrách sú popísané v tabuľke ?? . Tabuľka obsahuje iba tie hry, ktoré sa od seba výrazne líšia.

Kapitola 5

Zaver

5.0.3 Zhodnotenie splnitelnosti cielov

V nasom programe sa podarilo implementovat prostredie pre robotov, kde sa da sledovat postup algoritmu. Svet je navrhnuty tak, aby jeho zmena sposobila rozdielne naroky na algoritmy a tym zvyšovala obtiaznost hry v zmysle naroku na uspesnost algoritmu.

Program je pripraveny na testovanie algoritmov. Z casovych dovodov sa vsak nepodarilo otestovat algoritmy, ktore su povazovane v niektorých hrach za dobre, v danom rozsahu.

5.0.4 Mozne rozsirenje CODEWARS

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

J

edným zo spomenutých rozsireni je navrhnutie viditeľnosti tak, že aby robot mal asymetrické 'oci', t.j priamka definovaná smerom robota v ktorom je prave otoceny nebude rozdeľovať výsek na dva rovnake časti. Tento skulavy robot bude do nejakej strany vidieť viac ako do druhej. Dokonca je možné pripustiť extrém, kedy by robot vidieť iba za seba a tým mätol superov. Je povolená chôdza do akejkoľvek strany a tak jediný výsledok by bol, že by nesedelo zobrazovanie. Prinasa to síce možnosť, že sa robot nemože spoľahnúť ani na to, ako je otoceny. Algoritmus by musel sofistikove testovať, kam robot vidí, alebo si často a správne tipnúť. Takto hendikepovaný robot je skutočnou výzvou najmä pri hre skúseného hráča so začiatčikom. Preto toto rozsirenje je hodne pozornosti

Dalsou možnosťou je nechať užívateľa definovať, aký plánovač bude konkrétny robot používať. Možno očakávať, že výsledky by mohli byť zaujímavé nakoľko to vedie na programovaní robota, ktorý má podľa všetkeho rýchlejší algoritmus.

rozsirenje vzhľadom na jazyk

Jazyk robota aktualne nepodporuje deklarovanie premenných, ktoré boli vo funkcii už niekedy deklarované. Toto chovanie je síce pochopiteľné, avšak programátorský nepríjemné. Preto by sa dalo uvažovať o primeranej naprave. Jazyk dostatočne pokrýva základné požiadavky na popisania chovania robota. Avšak je príliš nízkoúrovňový, užívateľ si musí mnohé detaily ošetriť sám. To na jednej strane môže posobiť blahodárne na vymyslenie stratégie, na druhej strane môže užívateľa znechutiť napríklad, že si jeho robot zabúda vsimáť, že je ostreľovaný. Preto by bolo možné rozšíriť jazyk o funkcie, ktoré sa automaticky spustia pri vyvolaní udalosti. Toto by však vyžadovalo hlbší zásah do kódu, pretože robot ako objekt a jazyk ako hybatel robota sú implementované ako dve nezávislé entity. Bolo by nutné implementovať príslušné komunikačné rozhranie.

Dalsou možnosťou je pri implementovaní reakcie na udalosti nechať užívateľa vopred definovať tieto udalosti, na ktoré bude robot reagovať (napríklad "on seeEnemy() & 0"). Pri dobrom komunikačnom protokole by stačilo súčasne s vykonávaním kódu kontrolovať zoznam podmienok. Obmedzenie počtu udalostí, na ktoré robot môže reagovať, by tiež mohlo prispieť k zaujímavým úvahám nad stratégiou.