VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Optimalizace simulátoru pro hru fotbal robotů

Optimization of the Simulator for Robot Soccer Game

2012 Jakub Simandl

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Simandl**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Optimalizace simulátoru pro hru fotbal robotů
Optimization of the Simulator for Robot Soccer Game

Zásady pro vypracování:

Student se v této práci zaměří na simulátor fotbalu robotů, jež je součástí vznikající globální knihovny pro hru fotbalu robotů. Použití simulátoru je takřka nezbytné z hlediska testování nově vyvíjených strategií či taktik. Fotbal robotů je z velké míry založen na co možná nejrychlejší reakci na aktuální situaci na hřišti, proto je velmi důležitá rychlost odezvy jak celého systému pro fotbal robotů, tak i simulátoru v němž samotná hra probíhá.

Cíle práce:
1. Popsat stávající architekturu systému pro hru fotbalu robotů a s ním spojeného simulátoru.
2. Optimalizace enginu aktuálně používaného simulátoru.
3. Spolupráce s dalšími kolegy pracujícími na částech tohoto systému, za účelem vytvoření komplexního systému pro hru fotbalu robotů.

Seznam doporučené odborné literatury:

G. Klančar, M. Lepetič, R. Karba, B. Zupančič: Robot Soccer Collision Modelling and Validation in Multi-Agent Simulator, Mathematical and Computer Moddeling of Dynamical Systems, 2003
S.Hamidreza Kasaei, S.Mohammadreza Kasaei and S.Alireza Kasaei: Development a Real Time Cooperative Behavior Approach for Autonomous Soccer Robots Applied in Robocup, IACSIT International Journal of Engineering and Technology, 2010
S. R. Mohd Shukri and M. K. Mohd Shaukhi: A Study on Multi-Agent Behavior in a Soccer Game Domain, 2008

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Václav Svatoň**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 4. května 2012 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Rád bych poděkoval vedoucímu své bakalářské práce Ing. Václavu Svatoňovi za ochotu ke konzultacím, pomoc, věcné připomínky a čas, který mi věnoval při řešení zadané problematiky.

Abstrakt

Simulátor fotbalu robotů je program vyvinutý na katedře informatiky, sloužící pro simulaci hry reálných robotů. Tato práce si klade za cíl navrhnout a naimplementovat optimalizace jednotlivých částí simulátoru tak, aby vedly ke zrychlení simulace a aby simulace více odpovídala reálné hře robotů. V práci je popsána současná architektura simulátoru a také jednotlivé dílčí úpravy provedené v simulátoru. Jedná se konkrétně o úpravu výběru optimální strategie týmu, optimalizaci logování průběhu hry, paralelizaci provádění strategií a taktik a implementaci pohybu otáčení robotů. V poslední části jsou popsány změny vytvořené spolu s dalšími členy týmu.

Klíčová slova: Optimalizace, Simulátor, Fotbal Robotů

Abstract

Robot Soccer Simulator is a program developed at the department of computer science and its purpose is to simulate real-world robot soccer fotball. The aim of this thesis is to propose and implement optimizations of parts of the simulator which will lead to increase in speed of the simulation and to further improve the simulation so that it better reflects real robot football. The thesis describes current architecture and then individual adjustments and implementaions made in simulator. Specificaly optimization of best strategy rule selection, optimization of logging of the game, paralelization of strategy and tactic execution and implementation of rotational motion of robots. Cooperation with other colleagues who also worked with the simulator is described in the last part of the thesis.

Keywords: Optimization, Simulator, Robot Soccer

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce je zaměřena na optimalizace 2D simulátoru fotbalu robotů vyvinutém na katedře informatky. V programu se vyskytují části, jejichž optimalizace je důležitá pro zrychlení běhu samotné simulace, případně zlepšení kvality celkové simulace tak, aby více odpovídala reálné hře robotů.

V první části je nejprve popsána stávající architektura simulátoru a detailněji popsány jednotlivé části simulátoru (hlavní aplikace simulátoru, strategie, taktiky a fyzikální enginu). Je zde také vysvětlena hlavní smyčka simulátoru, ve které dochází k výpočtům strategií, taktik, k pohybu objektů hrací scény a k samotnému vykreslení hrací scény do hlavního okna aplikace.

Ve druhé části jsou vysvětleny jednotlivé úpravy provedené v simulátoru. Úpravy byly prováděny jednak kvůli tomu, že některé části byly nevhodně algoritmicky zpracovány a vedly by tak k přílišnému zpomalení simulace, nebo také z důvodu snahy o co možná nejvíce realitě odpovídající simulaci. Konkrétní změny zahrnují úpravu procesu výběru optimálního pravidla pro tým, úpravu logování průběhu hry, paralelizaci provedení strategií a taktik pro jednotlivé týmy a implementaci pohybu otáčení robotů. Je zde také popsána spolupráce s kolegy, kteří řeší implementaci dalších vylepšení simulátoru. Jedná se konkrétně o predikci chování soupeřových robotů a o implementaci pohybu robotů kolem překážky pomocí metody lineární regrese. V práci zabývající se predikcí chování šlo konkrétně o eliminaci duplicit při generování nových pravidel strategií během hry. Popíši postupy, které jsme použili při implementaci a případné problémy, které se vyskytly.

V závěru jsou zhodnoceny provedené úpravy a nastíněny další možnosti úprav a optimalizací simulátoru.

2 Stávající architektura simulátoru

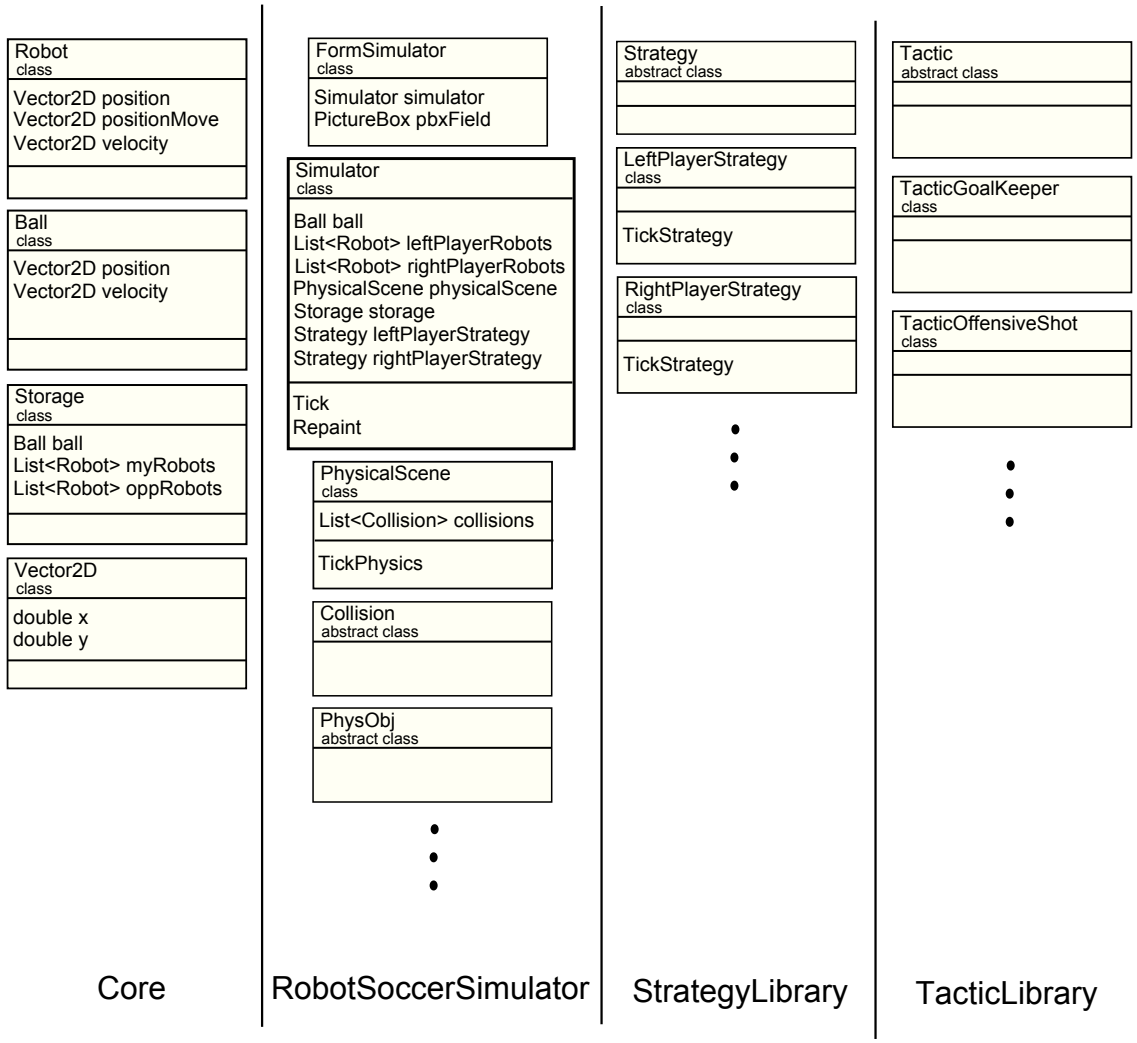
2.1 Architektura simulátoru obecně

Simulátor je napsán kompletně v programovacím jazyce C# [1] a je tedy postaven na .NET Frameworku [2]. Konkrétně se jedná o jednotlivé knihovny poskytující nám třídy, metody, enumerace, konstanty a funkce pro reprezentaci fotbalu robotů v reálném světě.

Hlavní částí simulátoru je Windows Forms aplikace, ve které dochází k samotnému průběhu simulace a jejímu vykreslování na hrací plochu.

Knihovny jsou rozděleny na "Core"knihovny, poskytující základní konstrukce, jako například matematický Vector2D, jenž je použit k reprezentaci ryhlosti a pozice míče či robotů. Dále poté na knihovnu Strategií, poskytující nám metody pro výpočet nejvhodnějšího pravidla strategie v každém kroku hry, podle nějž dochází k základnímu pozicování celého týmu (viz 2.3 - Strategie). Dalším modulem je Knihovna Taktik, jenž v každém kroku hry jemněji upravuje chování jednotlivých robotů, například to, že má útočící robot jít s míčem na bránu, nebo chování bránících robotů. Poslední důležitou knihovnou je knihovna fyzikálního enginu, která se stará o správné interakce mezi objekty herní scény tak, aby co možná nejvíce odpovídaly realitě (v současné verzi jsou třídy reprezentující fyzikální engine součástí hlavní Windows Forms aplikace, logicky jsou však tyto moduly plně oddělitelné).

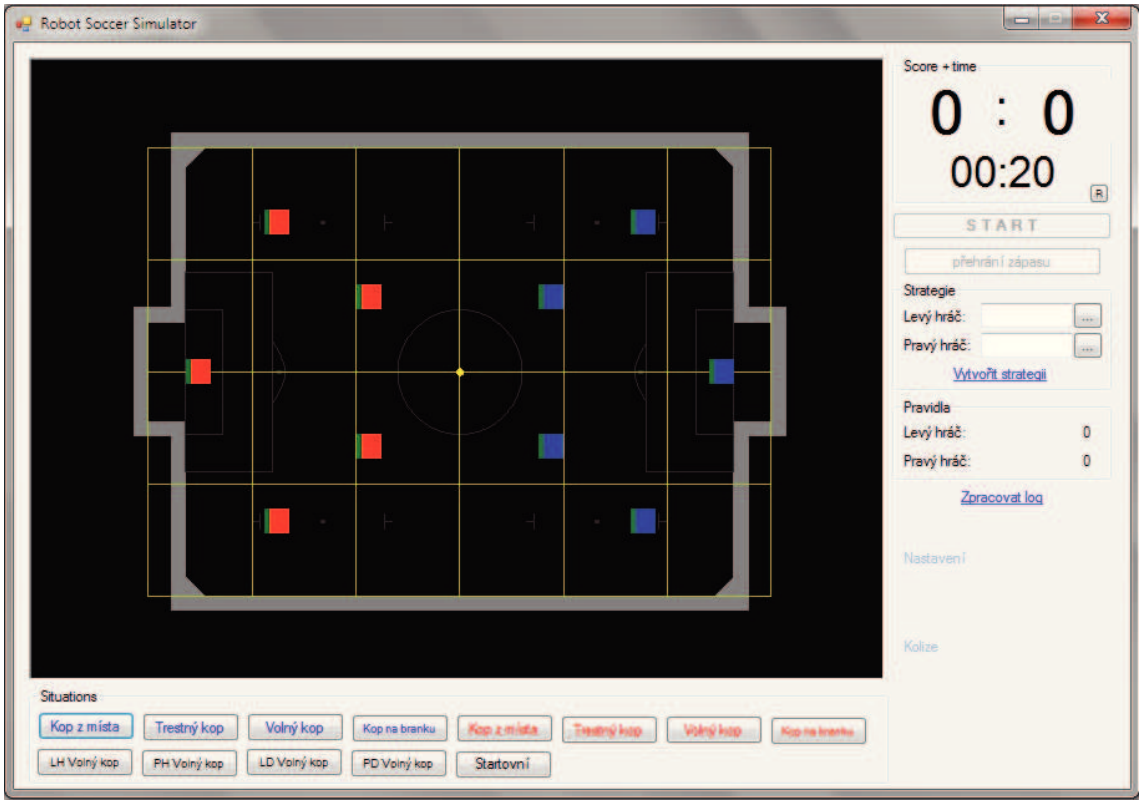
Funkce jednotlivých knihoven jsou poté všechny využívány právě v hlavní Windows Forms aplikaci. Jedná se tedy o poměrně modulární systém, který umožňuje rozšíření a úpravu jednotlivých částí (knihoven) bez výrazného efektu na části ostatní, pokud by byl zachován stejný interface. Pro lepší ilustraci systému jsou na obrázku 1. zobrazeny jednotlivé části simulátoru a do nich patřící třídy. Jelikož pro pochopení fungování simulátoru nejsou podstatné všechny třídy, datové položky a metody všech částí, uvádím jen ty důležité. Na obrázku 2. je poté zobrazeno hlavní okno aplikace, ve kterém je možno simulaci spouštět a ve kterém dochází k vykreslování herní situace v každém kroku hry.



Obrázek 1: Architektura simulátoru

2.2 Hlavní smyčka simulátoru

Hlavní smyčka simulátoru probíhá ve třídě FormSimulator (třída hlavního okna Windows Forms aplikace) v metodě timer1Tick, jež je vyvolána každých 80 milisekund komponentou timer1, což je časovač (instance třídy .NET frameworku Timer) zajišťující neustálý průběh hry. V tomto tiku se střídavě volá metoda Refresh datové položky pbxField, což je instance třídy PictureBox. Tato třída slouží ve WindowsForms aplikaci pro renderovací účely a dochází v ní k vykreslení celé herní situace.



Obrázek 2: Hlavní okno aplikace

Metoda Refresh tedy zajistí každý herní krok vykreslení hrací plochy s novými údaji a hodnotami, které jsou pro daný krok vypočteny. Poté dojde k zavolání metody Tick datové položky simul, což je instance hlavní třídy simulátoru a to třídy Simulator. Tik datové položky timer1 se spustí stisknutím tlačítka Start v hlavním okně aplikace. Zmíněná třída Simulator, zodpovědná za hlavní průběh simulace, obsahuje několik důležitých komponent:

- Položka storage - instance třídy Storage, což je jakýsi kontejner informací o všech objektech na hrací scéně. Tento kontejner se v každém volání metody Tick (viz níže) naplní informacemi o objektech hrací scény voláním příslušných metod. V každém tiku je poté nad daty z tohoto kontejneru provedeno několik výpočtů metodami z různých částí knihovny (strategie, taktiky, fyzikální engine). Provedením těchto

metod se změní hodnoty v kontejneru a v závislosti na těchto hodnotách se poté vykresluje celá hrací situace v již zmíněné metodě Refresh.

- Položky reprezentující strategie levého a pravého týmu (leftPlayerStrategy a rightPlayerStrategy), což jsou instance třídy Strategy. Tyto třídy mají důležitou metodu TickStrategy, která má jako vstupní parametr referenci na instanci třídy Storage. Tato metoda vypočte kam se mají roboti pohnout v dalším kroku, z informací které jí byly ve vstupním parametru dodány (v podobě kontejneru storage) a nastaví tyto nové informace opět v již zmíněném globálním kontejneru.

- Položky reprezentující jednotlivé roboty (rightPlayerRobots a leftPlayerRobots) - Listy instancí třídy Robot, což je třída nesoucí veškeré důležité informace o robotovi na hrací ploše - Jeho tým, jeho fyzikální reprezentaci určenou instancí třídy PhysObjSquare (to nám zaručuje, že se robot nechová jako např. jeden pixel, ale opravdu jako čtverec), jeho aktuální pozici v hrací ploše (reprezentovanou instancí třídy Vector2D, která zapouzdřuje x a y souřadnice 2D vektoru), jeho rychlost, což je vpodstatě další instance třídy Vector2D, dále metody AddRobotToScene (zajišťuje přidání robota do fyzikální scény), metoda GoTo případně Stop, které kontrolují nastavení jeho datových položek velocity podle toho jaký pohyb má robot vykonat v příštím kroku a metoda Paint, zajišťující vykreslení robota na hrací scéně.

- Položka ball, instance třídy Ball, což je reprezentace hracího míče v simulátoru. Obsahuje stejné datové položky jako Robot pro jeho použití v hrací scéně, fyzikální reprezentací je poté instance třídy PhysObjCircle.

- Položka physicalScene, instance třídy PhysicalScene - jedná se o třídu představující fyzikální reprezentaci hrací plochy. Tato třída obsahuje příslušné metody pro řešení pohybu objektů na scéně a jejich vzájemnou interakci (kolize).

Při inicializaci položky simul ve třídě FormSimulator dojde rovněž k inicializaci položky storage a k jejímu naplnění údaji z vytvořených instancí tříd Robot a Ball. U každého objektu na scéně je potřeba pro provedení výpočtů s nimi znát jejich vektor, určující sou-

časnou pozici (position) a směrový vektor, pomocí kterého dochází k pohybu robota (velocity).

Ve zmíněné metodě Tick třídy Simulator dochází k následujícím akcím:

1. Voláním příslušných metod na datové položce storage se nastaví v instanci třídy Storage pozice robotů a míče v okamžiku zavolání metody Tick.

2. Následně dojde k zavolání metod TickStrategy u instancí reprezentující strategie jednotlivých týmů (leftPlayerStratetgy a rightPlayerStrategy). Tyto metody provedou výpočet optimálního výběru pravidel pro roboty daných týmů. Náslědně dojde k provedení taktik nad jednotlivými týmy a nastavení příslušných datových položek opět v objektu storage (informace o tom kam se má robot pohnout - instance třídy Vector2D - položka positionMove u každého robota).

3. Poté dojde k zavolání metody GoTo pro všechny roboty na scéně. Tato metoda nastaví příslušnou datovou položku u instancí třídy Robot (položku velocity), právě na základě informace z položky positionMove. Pomocí vektoru velocity poté dojde k vykreslení pohybu robota. Informace o tom, jaké hodnoty se mají metodě GoTo předat, se získají v hlavní smyčce z globálního kontejneru storage.

4. Dojde k zavolání metody TickPhysics datové položky physicalScene, která zapouzdřuje fyzikální scénu hrací plochy. V této metodě dojde k detekci kolizí a jejich vyřešení, provedení pohybu míče a robotů (nikoliv však k jeho překreslení na hrací ploše, zde dochází pouze ke změně hodnoty položky udávající pozici objektů - vektor position) pomocí informací z jednotlivých instancí tříd Robot a Ball (detailnější popis v sekci 2.5 - Fyzikální Engine).

5. Překreslení celé hrací plochy, tentokrát již s novými údaji o všech objektech hrací scény metodou Repaint třídy FormSimulator.

2.3 Strategie

Hlavní částí knihovny určující pohyb a chování robotů je knihovna Strategie (Strategy). Strategie je v simulátoru chápána jako soubor pravidel pro každý tým, určující kam se mají roboti v současném kroku přesunout na základě pozic objektů na hrací ploše. Strategie je tedy určena pouze k pozicování týmů robotů a to v tzv. gridových souřadnicích. Gridové souřadnice rozdělují hrací plochu na zjednodušenou matici (v současné verzi hry má rozměry 6x4 prvků). Pro simulaci a pozicování robotů pomocí strategie je toto rozdělení výhodnější, jelikož je jednodušší provádět výpočty nad menší maticí. Rozdělení na gridové souřadnice je také pro realizaci strategií plně dostačující [9] Pro jemnější polohování robota, které se využívá u taktik je však vždy potřeba přepočítat gridové souřadnice zpět na reálné souřadnice (v současné verzi hry se může objekt pohybovat na ploše o rozměrech 180x110 pixelů).

Strategie je v každém herním kroku reprezentována jedním pravidlem (v simulátoru reprezentováno instancemi třídy GridRule), které popisuje pozice robotů obou týmů a míče v současném kroku a pozice kam by se měli roboti na základě této informace pohnout v kroku následujícím. Tato informace je uložena formou textového souboru, a po spuštění simulátoru a načtení dané strategie dojde k inicializaci instance třídy reprezentující množinu jednotlivých strategií pro každý tým (tato množina je reprezentována instancí třídy GridStrategy u strategií pro oba týmy) a její naplnění instancemi třídy GridRule (neboli jednotlivými pravidly). V textovém souboru je následujícím způsobem popsáno každé pravidlo:

.Rule 1 o Name
.Mine 3,3 2,4 2,2 2,1
.Oppnt 6,3 6,4 6,2 5,1
.Ball 5,5
.Move 4,4 5,4 3,2 3,1

Hodnoty v řádcích uvozených ".Mine"a ".Oppnt"udávájí pozice hráčů obou týmů jako souřadnice x a y v matici "gridového"hracího pole. Řádek ".Ball"udává pozici míče a řádek ".Move"udává ty pozice na které se v příštím kroku hry mají pohnout hráči z pozic ".Mine". Pro brankáre každého týmu není v pravidlech zapsáno jeho chování, jelikož brankář má specifickou úlohu (nepřemisťuje se po celém hrišti, ale pouze si "hlídá"prostor brankoviště. Pro brankáře je implementováno chování přímo v kódu metodou GoalKeeperStrategy ve třídách strategií obou týmů. Ta upravuje brankářův pohyb tak, aby se jeho reálná Y souřadnice pokud možno vždy rovnala reálné Y souřadnici míče a jeho reálná X souřadnice je konstantní v oblasti brankoviště (toto chování je v situaci kdy se míč nachází v blízkosti brankoviště přepsáno taktikou, která dále upraví pohyb brankáře - viz 2.4 - Taktiky).

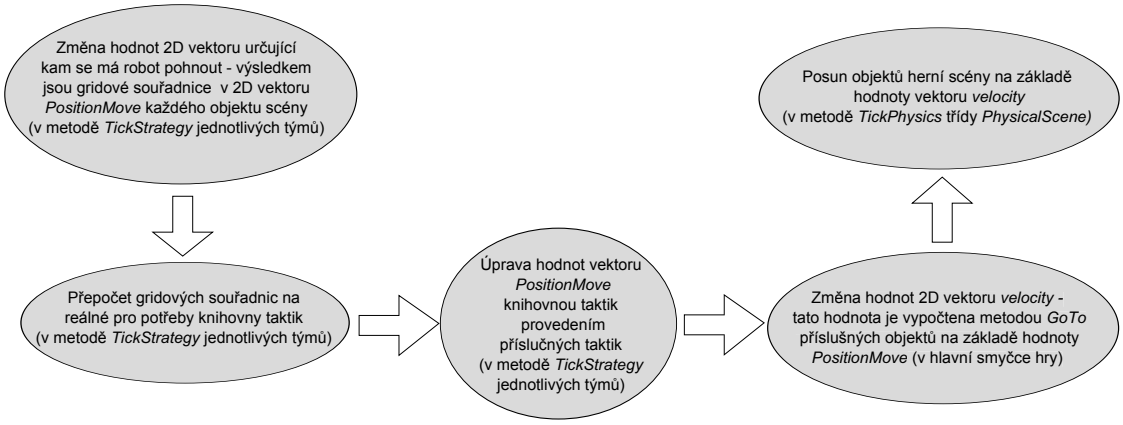
Pro přepočet mezi reálnými souřadnicemi pozice robota čí míče a tzv. gridovými souřadnicemi a naopak můžeme použít příslušné metody třídy GridStrategy a to RealToGrid respektive GridToReal. Po vypočtení optimálního pravidla pro daný tým je každému z robotů nastavena položka positionMove (instance třídy Vector2D udávající pozici robota) a na základě této informace je poté vypočten vektor rychlosti a směru pohybu robota (položka velocity) metodou GoTo třídy Robot.

Na základě vektoru velocity dochází ve fyzikálním enginu k samotnému pohybu robotů a míče herní scény v čase (metodou move příslušných objektů). Je zřejmé, že míč nemá datovou položku positionMove, jeho pozice a směrový vektor posunu jsou vždy vypočteny až ve fyzikálním enginu na základě jeho kolize s roboty a okraji herní plochy.

Celý princip "chování"objektů hrací scény v herním kroku je znázorněn na obrázku 3.

2.4 Taktiky

Knihovna taktik, která je zodpovědná za základní fotbalové konstrukce jako jsou například "útok", "obrana", či "přihrávka", případně za interakci jednotlivých robotů mezi sebou v rámci týmu, je v současné verzi simulátoru poměrně strohá a nedodělaná. Obsahuje vpodstatě jen základní metody pro výběr jednoho robota z každého týmu jako



Obrázek 3: Pohyb objektů hrací scény

útočníka a jeho následné vedení s míčem na bránu soupeře. Dále pak řízení pohybu brankáře pokud se v blízkosti jeho brankoviště ocitne míč tak, aby se snažil míč z brankoviště vytlačit. K výpočtům taktik dochází po provedení výběru optimální strategie a to opět předáním kontejneru storage a následným voláním metod, které přímo upravují položku positionMove jednotlivých robotů. Jelikož taktiky pracují s reálnými souřadnicemi (pro jemnost simulace taktik je nemožné použít gridové souřadnice), je potřeba převést po provedení strategií souřadnice objektů z gridových na reálné pomocí příslušných metod (viz 2.3 - Strategie). V budoucích verzích simulátoru je plánováno podstatné rozšíření právě této knihovny.

2.5 Fyzikální Engine

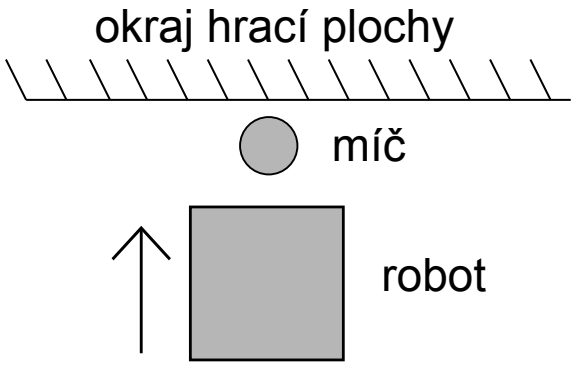
Jelikož má program simulovat reálnou hru robotů, je v něm implementován i fyzikální engine, rešící posun objektů po hrací ploše a řešení jejich vzájemných kolizí. Engine pracuje pouze s fyzikálními reprezentacemi objektů hrací scény, nad kterými provádí samotné výpočty. Podobně jako v práci, kterou řešil p. Klančar a jeho tým je proces nalezení kolizí rozdělen do dvou podfází [10] . Nejprve jsou mezi možné kolize zařazeny ty objekty, jejichž blízkost v hracím poli by znamenala potenciální kolizi v dalších krocích hry (u zmíněného týmu je toto realizováno na základě překryvu tzv. bounding boxů každého objektu) a až když dva objekty splňují tuto podmínku, dojde k jemnějšímu posouzení

samotné kolize. Toto je výhodnější zejména z hlediska časové náročnosti metody hledání kolizí.

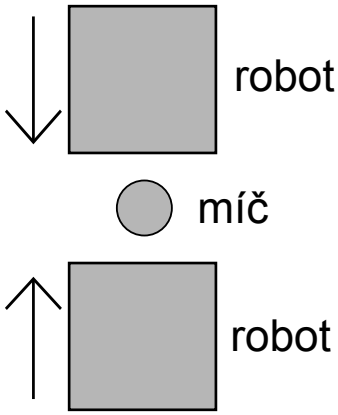
Fyzikální engine je v simulátoru reprezentován hlavní třídou PhysicalScene. Třída obsahuje Listy instancí tříd představující fyzikální zastoupení objektů v hrací ploše (míč má fyzikální reprezentaci instancí třídy PhysObjCircle, robot je reprezentován třídou PhysObjSquare, okraje hrací plochy poté třídou PhysObjPlane). Při inicializaci simulátoru dojde k naplnění těchto listů instancemi příslušných objektů všech robotů, míče a okrajů plochy na scéně pomocí metod AddRobotToScene, AddBallToScene a AddAreaToScene. Při zavolání metody TickPhysics v hlavní smyčce simulátoru dojde k zavolání tří podstatných metod třídy PhysicalScene:

- FindNextCollisions - Metoda, která na základě údajů z naplněných listů zjišťuje, zda ve scéně nedochází mezi některými fyzikálními objekty ke kolizi. Pokud ke kolizi dochází (kolize jsou v enginu řešeny předikčně - tedy ke skutečnému "překrytí"dvou objektů scény prakticky nikdy nedojde), je tato informace přidána do listu uchovávajícím všechny kolize v současném kroku hry (list collisions třídy PhysicalScene). Typy kolizí, se kterými fyzikální engine pracuje, mohou být následující:
  - Kolize míče a okraje herní plochy (třída CollisionCirclePlane)
  - Kolize míče a robota (třída CollisionCircleSquare)
  - Kolize míče, robota a okraje herní plochy zároveň (CollisionSquareCirclePlane) viz obrázek 4.
  - Kolize míče a dvou robotů (CollisionSquareCircleSquare) viz obrázek 5.
  - Kolize robota a okraje herní plochy (CollisionSquarePlane)

- DeleteDupeCollisions - Metoda starající se o to, aby se v kolekci kolizí nevyskytovaly nadbytečné či duplicitní kolize. Pokud se ve scéně vyskytne například kolize dvou robotů a míče, metoda FindNextCollisions tuto situaci vyhodnotí jako 3 různé kolize a to kolizi prvního robota s míčem, kolizi druhého robota s míčem a poté kolizi obou



Obrázek 4: Kolize míče, robota a okraje herní plochy

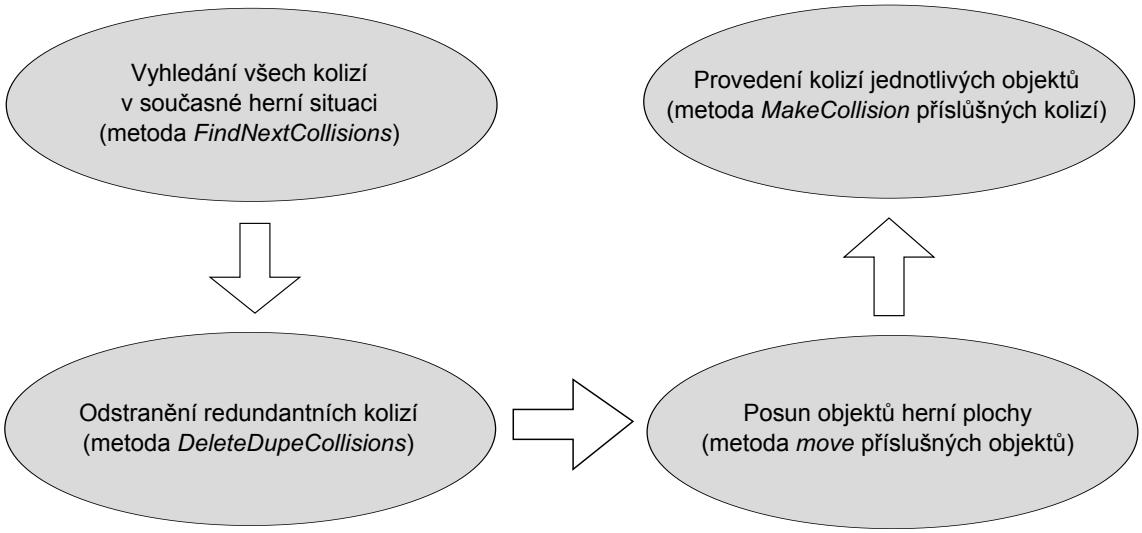


Obrázek 5: Kolize robota, míče a robota

robotů a míče. Je tedy potřeba odstranit první a druhou přebytečnou kolizi, jelikož je již obsažena ve třetí.

- MakeCollision - Metoda provádějící všechny kolize z kolekce kolizí
- Move - Metoda příslušných fyzikálních objektů scény, provádějící posun robotů a míče v čase ve fyzikální scéně na základě informací z jejich vektoru velocity.

Průběh metody TickPhysics je znázorněn na obrázku 6.



Obrázek 6: Průběh metody TickPhysics

Detailněji je samotný fyzikální engine i řešení jednotlivých kolizí popsáno v diplomové práci, která se zabývala implementací enginu do simulátoru [3].

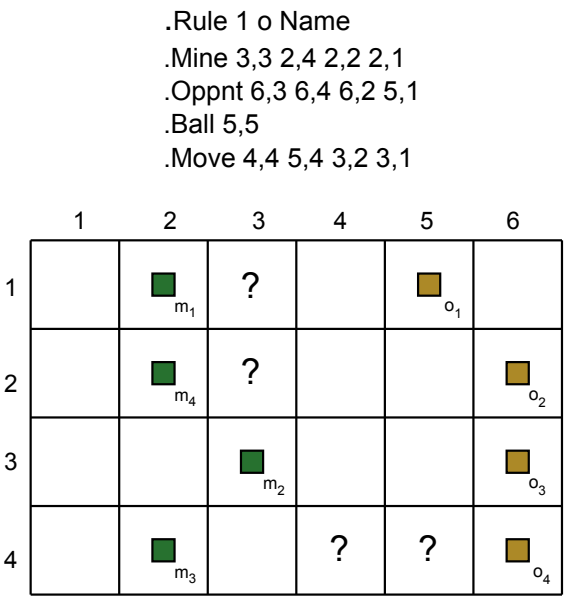
3 Optimalizace částí simulátoru

3.1 Alrgoritmus výběru optimálního pravidla

3.1.1 Současný algoritmus

V současné verzi hry je algoritmus výběru optimálního pravidla ze strategie týmu implementován takto: ve třídě pro strategii týmu (leftPlayerStrategy, rightPlayerStrategy) je v každém kroku hry zavolána metoda pro provedení strategie každého týmu (TickStrategy) jejímž parametrem je storage - kontejner obsahující informace o pozicích robotů a míče v daném kroku hry. V této metodě je nejprve nastavena již zmíněná strategie pro gólmana a poté je metodou pro výběr optimálního pravidla (FindBestRule) vybráno optimální pravidlo z množiny pravidel pro daný tým (množina pravidel je instance třídy GridStrategy, obsahující jednotlivá pravidla - instance třídy GridRule). Na konci metody TickStrategy je ještě zavolána metoda pro vykonání taktik pro daný tým - metoda ChooseTactic třídy TacticChooser.

Princip současné verze metody pro výběr optimálního pravidla je následující: Pro každý objekt na scéně, jehož pozice je uvedena v jednotlivých pravidlech (tedy pro 8 robotů a míč - v textovém souboru pravidel uvozeny jako .Mine, .Ball a .Oppnt) je spočítána vzdálenost mezi jeho současnou pozicí a pozicí v každém pravidle, abychom mohli určit, které pravidlo nejlépe odpovídá současnému stavu hry. Tyto vzdálenosti jsou poté sečteny a pravidlo s nejmenší celkovou vzdáleností (nazvěme tuto vzdálenost koeficientem vzdálenosti) je vybráno jako optimální pravidlo pro tento krok hry a na základě tohoto pravidla jsou poté roboti posláni na pozice v kroku příštím. Jelikož však není známo, který záznam o pozici robota v každém pravidle reprezentuje jakého robota v současné hře, je tento výpočet nutné provést permutačně na všechny možné kombinace pořadí robotů (do kontejneru jsou roboti přiřazeni vždy ve stejném pořadí, nehledě na jejich současnou pozici ve hře). Algoritmus tedy obsahuje pole 24 polí obsahující permutaci 4 pozic robotů pro každý tým (tento model je značně nevýhodný, jednak z pohledu rychlosti a také z pohledu rozšiřitelnosti, při zvednutí počtu hráčů na 5 by toto pole bylo již 120 prvkové). Na obrázku 7. je zobrazena situace při výběru optimálního pravidla pro levý tým:



Obrázek 7: Situace při výběru pravidla

Znakem „?“ jsou zobrazeny pozice, na které se mají roboti levého týmu podle zadaného pravidla v přístím kroku hry pohnout. Nevíme však, který robot se má pohnout na kterou souřadnici. Je tedy potřeba zjistit, u kterého pořadí robotů v poli robotů odpovídá rozestavení nejvíce zadané situaci v pravidle (proto tedy musíme porovnávat situaci v pravidle s permutací pořadí robotů v listu robotů).

V následujícím kódu je uvedena současná metoda pro nalezení optimálního pravidla pro tým. Pole *indexes* je zmíněné 24 prvkové pole obsahující všechny možné pořadí robotů v listu robotů.

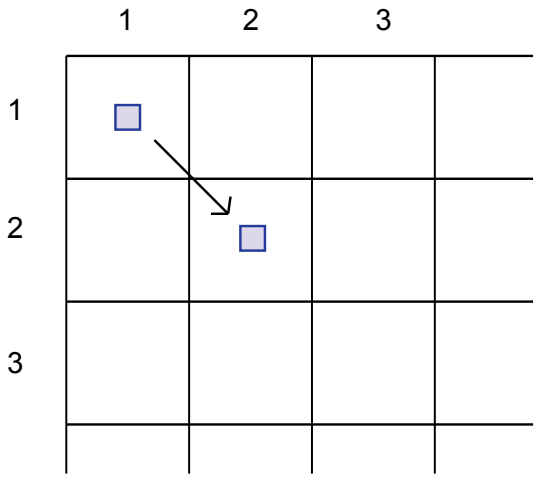
```
GridRule FindBestRule(ref int[] perm, Storage storage) {
    GridRule bestRule = null;
    int [][] indexes = { new int[] { 1, 2, 3, 0 }, ...};
    //Celkový koeficient pravidla
    double min = double.MaxValue;

    //Pro každé pravidlo z množiny pravidel daného týmu
    foreach (GridRule rule in gStrategy.Rules) {
        //Vypočteme vzdálenost současné pozice míče a pozice míče v pravidle
        double ballRuleSize = rule.Ball.DistanceFrom(storage.Ball)

        //Koeficient vzdálenosti mezi současnými pozicemi
        //robotů soupeře a pozicemi robotů soupeře v pravidle
        double oppRuleSize = double.MaxValue;

        //Pro všechny možné kombinace pořadí soupeřových robotů
        //spočteme jejich vzdálenosti od pozic v pravidle
        foreach (int [] i in indexes) {
            double size = rule.Oppnt[0].DistanceFrom(storage.OppRobots[i[0]] + rule.Oppnt[1].
                DistanceFrom(storage.OppRobots[i[1]]) +
            rule.Oppnt[2].DistanceFrom(storage.OppRobots[i[2]]) + rule.Oppnt[3].DistanceFrom(
                storage.OppRobots[i[3]]);

            if (size < oppRuleSize)
            oppRuleSize = size;
        }
        //Pro všechny možné kombinace pořadí našich robotů
        //spočteme jejich vzdálenosti od pozic v pravidle
        foreach (int [] i in indexes) {
            double myRuleSize = rule.Mine[0].DistanceFrom(storage.MyRobots[i[0]]) + rule.Mine[1].
                DistanceFrom(storage.MyRobots[i[1]]) +
            rule.Mine[2].DistanceFrom(storage.MyRobots[i[2]]) + rule.Mine[3].DistanceFrom(storage.
                MyRobots[i[3]]);

            //Pokud je koeficient vzdálenosti pravidla menší než dosavadní
            if (ballRuleSize + oppRuleSize + myRuleSize < min) {
            min = ballRuleSize + oppRuleSize + myRuleSize;
            bestRule = rule;
            perm = i;
            }
        }
    }
    //Metoda vrací nejoptimálnější pravidlo
    return bestRule;
}
```

Výpis 1: Současný algoritmus výběru optimálního pravidla

Nejprve je tedy spočítána vzdálenost mezi současnou pozicí míče a pozicí v každém pravidle a poté tyto vzdálenosti i pro každého robota. Metoda DistanceFrom spočítá vzdálenost dvou objektů pomocí 2D vektorů udávající jejich pozice, v případě naší hry je však tato metoda nepřesná, jelikož používá vzdálenost dvou bodů v prostoru a neuvažuje gridové rozdělení hrací plochy. Tato vzdálenost je vypočtena následovně: Pro daný vektor $p(x, y)$ je vzdálenost od vektoru $q(x, y)$ rovna:

$$distance(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

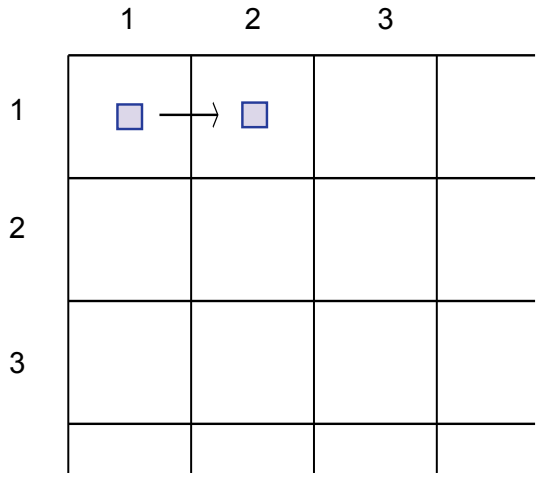
Nepřesnost této metody je uvedena na následujícím příkladu: Na obrázku 8. je robot nejprve v pozici $x = 1$, $y = 1$ a poté v pozici $x = 2$, $y = 2$. Vzdálenost je rovna přibližne 1,41:



Obrázek 8: Pozice robota

$$distance(p, q) = \sqrt{(1 - 2)^2 + (1 - 2)^2} \cong 1,41$$

Na obrázku 9. je robot nejprve v pozici $x = 1$, $y = 1$ a poté v pozici $x = 1$, $y = 2$. Vzdálenost je rovna 1:



Obrázek 9: Pozice robota

$$distance(p,q) = \sqrt{(1-1)^2 + (1-2)^2} = 1$$

Vzdálenost robota mezi jednotlivými situacemi na obrázku 8. je tedy současnou metodou vyhodnocena jako 1,41. V gridových souřadnicích hry ovšem potřebujeme, aby vzdálenost byla rovna 1, jelikož pozice spolu přímo sousedí (to znamená, že robot se posunul jen o jedno pole - jak pozice na obr. 8. tak na obr. 9. by měly být vyhodnoceny jako stejné situace). V části Implementace nového algoritmu je popsáno řešení tohoto problému novým algoritmem.

3.1.2 Implementace nového algoritmu

Nejprve byl implementován nový algoritmus pro výpočet vzdálenosti dvou bodů v gridových souřadnicích. Metoda pro tento výpočet má jako parametry dva 2D vektory a vrací vzdálenost jako celé číslo. Princip metody je popsán následujícím kódem:

```
int PointDistance(Vector2D a, Vector2D b)
{
   int tempx;
```

```
    int tempy;

    if (a.X < b.X) tempx = (int)(b.X − a.X);
    else if (a.X == b.X) tempx = 0;
    else tempx = (int)(a.X − b.X);

    if (a.Y < b.Y) tempy = (int)(b.Y − a.Y);
    else if (a.Y == b.Y) tempy = 0;
    else tempy = (int)(a.Y − b.Y);

    if (tempx == tempy) return tempx;
    else if (tempx == 0) return tempy;
    else if (tempy == 0) return tempx;
    else if (tempx > tempy) return tempy + (tempx − tempy);
    else return tempx + (tempy − tempx);
}
```

Výpis 2: Metoda pro výpočet vzdálenosti dvou bodů v gridových souřadnicích

Pro vstupy vektorů $p(1,1)$ a $q(2,2)$ vrátí metoda výsledek 1. Metoda tedy správně počítá vzdálenost dvou bodů v hracím poli jako počet kroků, kterými se z jednoho bodu dostaneme do druhého, pakliže je jeden krok roven pohybu z jedné gridové souřadnice na souřadnici sousedící.

S použitím této metody je nyní možno vypočítat celkový koeficient vzdálenosti mezi současnou herní situací a situací popsanou v každém z pravidel. Označme koeficient vzdálenosti dvou situací jako K_c , pak:

$$K_c = K_n + K_s + K_m$$

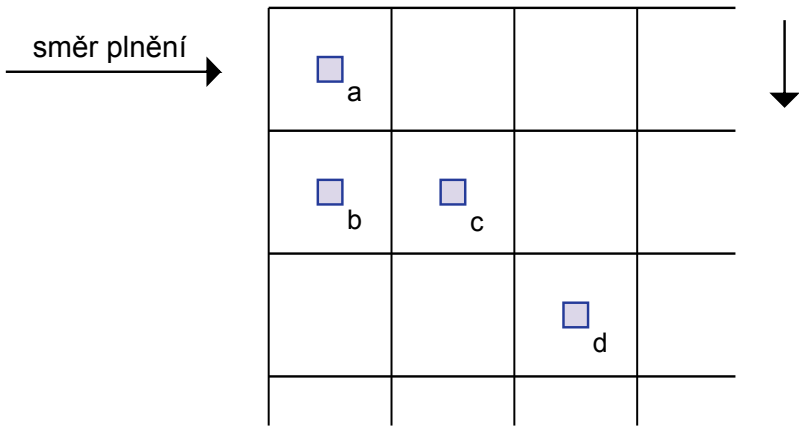
kde K_n je rovno součtu vzdáleností pozic našich robotů mezi současnou situací a situací v pravidle, K_s je roven součtu vzdáleností pozic soupeřových robotů a K_m je roven vzdálenosti pozic míče. Jelikož pozice robotů v poli můžeme zapsat jako matici, kdy jednotlivé řádky tvoří dvourozměrné vektory určující pozici robota, můžeme koeficienty K_n a K_n určít pomocí metody na výpočet vzdálenosti mezi dvěma těmito maticemi. Matici budeme pro algoritmický výpočet reprezentovat jako list dvourozměrných vektorů. Metoda tedy bude mít jako parametry dva tyto listy a bude vracet vzdálenost matic jako celé číslo představující sumu vzdálenosti jednotlivých vektorů v maticích. Je však třeba vyřešit to, který vektor z první matice budeme porovnávat s kterým vektorem z matice druhé. U současného algoritmu je to vyřešeno pomocí permutace, kdy se porovnává každý vektor

s každým a poté se určí u kterého pořadí byl výsledek nejnižší. V novém algoritmu je použito tzv. mapování, jehož princip je vysvětlen na následujícím příkladu: Mějme pole vektorů M_1 a M_2 obsahující čtyři dvourozměrné vektory a_1 až a_4. Tato pole reprezentují matici rozestavení robotů v herní situaci:

$$M_1 = (a_1, a_2, a_3, a_4)$$

$$M_2 = (a_1, a_2, a_3, a_4)$$

Nyní budeme v cyklu postupně měřit vzdálenosti mezi jednotlivými vektory, a pokud nalezneme vektory, jejichž vzdálenost je rovna určité hodnotě kterou s každým krokem navyšujeme, tak je označíme za "namapované", jejich vzdálenost přičteme k celkové vzdálenosti matic a v příštím kroku budeme tyto vektory pro výpočty ignorovat. Pro účely mapování si tedy u každého vektoru zavedeme pomocnou vlastnost isMapped, určující zda už byl s nějakým vektorem namapován. V simulátoru může být maximální vzdálenost mezi dvěma objekty v gridových souřadnicích rovna 5 a minimální vzdálenost rovna 0 (robot je v obou herních situacích ve stejné souřadnici), proto bude cyklus začínat na 0 a končit na 5, s krokem 1. Výpočet vzdálenosti mezi dvěma body probíhá pomocí výše popsané metody PointDistance. Vektory představující pozice robotů je však potřeba v poli vždy seřadit, a to nejprve podle y souřadnice a poté podle x souřadnice. To nám zajistí, že pořadí vektorů v poli bude vždy odpovídat rozmístění hráčů na herní ploše v určitém sledu. Viz obázek 10.:



výsledné pole vektorů: M = {a, b, c, d}

Obrázek 10: Směr naplňování pole vektorů

```
int MatrixDistance(List<Vector2D> a, List<Vector2D> b) {
    //Celková "vzdálenost" dvou herních situací
    int sumDistance = 0;

    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 4; j++) {
            for (int k = 0; k < 4; k++) {
                //Vzdálenost 2 vektorů se vypočte jen pokud ještě oba nejsou namapovány
                if (a[j].isMapped == false && b[k].isMapped == false) {
                    //Výpočet vzdálenosti mezi dvěma vektory
                    int tempDistance = PointDistance(a[j], b[k]);
                    if (tempDistance == i) {
                        //Když úspěšně "namapujeme" 2 vektory, tak jejich vzdálenost se přičte k
                            celkové vzdálenosti matic
                        sumDistance += tempDistance;
                        //A u obou vektorů nastavime isMapped na true, aby se v příštích krocích
                            ignorovaly
                        a[j].isMapped = true;
                        b[k].isMapped = true;
                    }
                }
            }
        }
        //Pokud jsou již všechny vektory namapovány, nemá cenu pokračovat dále, celková
            vzdálenost je vypočtena
        if (a[0].isMapped && a[1].isMapped && a[2].isMapped && a[3].isMapped) {
```

```
                goto navesti;
            }
        }
        navesti:
            //Metoda vrací sumu vzdáleností vektorů z obou matic
            return sumDistance;
    }
```

Výpis 3: Metoda pro výpočet vzdálenosti dvou matic reprezentující pravidla

Samotný algoritmus výpočtu nejvhodnějšího pravidla je poté podobný tomu v současné verzi hry. Projdeme všechna pravidla v množině pravidel daného týmu a pro každé pravidlo vypočteme koeficient vzdálenosti matic (reprezentující současnou situaci a situaci v pravidle) K_c. Následně vybereme pravidlo s nejmenším koeficientem jako nejvýhodnější. Velkou výhodou algoritmu pro výpočet vzdálenosti dvou matic novou metodou je jeho časová náročnost při stoupajícím počtu hráčů. Oproti původní metodě bude počet průchodů smyčkou pořád stejný (maximální možná vzdálenost bude konstantní, pakliže tedy nedojde s rozšířením hry i o rozšíření hřiště a tedy změně gridových souřadnic), bude pouze docházet k většímu množství porovnání vzdálenosti mezi roboty. Při stavu, kdy jsou roboti vůči nějakému pravidlu ve velmi podobných pozicích dojde jen k potřebnému průchodu smyčkou (v situaci kdy jsou roboti na totožných pozicích dojde pouze k jedinému průchodu). Maximální počet průchodů je poté 5, což je největší možná vzdálenost v gridovém rozdělení hrací plochy. Srovnání časové náročnosti obou algoritmů je znázorněno v následující části.

3.1.3 Porovnání obou způsobů výběru optimálního pravidla

Měření časové náročnosti různých algoritmů bylo prováděno pomocí třídy .NET frameworku - Stopwatch [6, 7]. Jedná se o třídů z jmenného prostoru System.Diagnostics, jejíž použití je ukázáno na následujícím úryvku kódu:

```
int main() {
    Stopwatch s = new Stopwatch();
    s.Start();
     ...
    (blok měřeného kódu)
     ...
    s.Stop();
    double elapsedMS = s.ElapsedMilliseconds;
```

```
        double elapsedTick = s.ElapsedTicks;

        ...
    }
```

Výpis 4: Ukázka použití třídy Stopwatch

Před blokem kódu, u něhož chceme zjistit jeho přibližný čas vykonávání, nejprve vytvoříme novou instanci třídy Stopwatch a zavoláme metodu Start, spouštějící měření času. Poté můžeme po vykonání bloku měřeného kódu zjišťovat hodnoty buďto uplynulých milisekund nebo časových tiků od spuštění měření položkami ElapsedMilliseconds respektive ElapsedTicks objektu třídy Stopwatch.

Měřením se potvrdila menší časová náročnost nové metody, průměrná hodnota trvání výpočtu obou strategií původní metodou byla pro vzorek přibližně 1000 herních kroků stanovena na přibližne 3ms, u nové verze čas nedosáhl ani 1ms (hodnoty ElapsedTicks: původní verze - průměrně 4700 ticks, nová verze - průměrně 950 ticks). Tento rozdíl by se stoupajícím počtem robotů výrazně rostl.

To, zda nový algoritmus vybírá vhodnější pravidlo pro tým nelze snadno objektivně posoudit z porovnání dvou her - změny v celkovém výběru pravidel (a tedy v celkové hře) se totiž odvíjí již od prvního rozdílně vybraného pravidla.

3.2 Logování v průběhu hry

3.2.1 Implementace nového způsobu

Logování hry je pro simulátor důležité. Hodí se zejména pro zpětnou analýzu hry robotů a také pro některé další aplikace (z logu se dá například hra zpětně přehrát, nebo by mohl sloužit jako základ pro generování strategií, které by vedly ke zlepšení výsledků týmu). V současné verzi hry je logování prováděno v hlavní smyčce simulátoru, tento model je však nevyhovující, jelikož při každém kroku hry jsou zapisovány do souboru potřebné informace, což by při větším množství informacé mohlo vést ke zpomalování provádění hlavní smyčky simulace. Lepším řešením by bylo během hry pouze uchovávat tyto informace v paměti, a vždy po skončení hry je teprve zapsat na disk. Byla proto vytvořena třída LogHolder představující kontejner informací o průběhu hry, do kterého

jsou v každém kroku přidány informace o daném kroku (jedná se o pozice všech robotů v poli a pozici míče - informace jsou ukládány jako listy 2D vektorů, představující tyto pozice).

Po ukončení hry (po řádném doběhnutí, či přerušení časomíry) jsou zavolány metody sloužící k zapsání informací do textového, případně i XML souboru. Dalším možným zlepšením logování v průběhu hry by bylo zapisovat informace ze třídy LogHolder do výstupních souborů v průběhu hry ve vlákně pracujícím na pozadí. Mohl by být použit nějaký druh fronty, do které by byla během hry data zapisována a ze které by postupně vlákno odebíralo informace a zapisovalo je do výstupních souborů. Implementací nového způsobu se podařilo urychlit hlavní smyčku simulace přibližně o 1ms. Je však nutno brát v úvahu, že v budoucích verzích simulátoru by mohlo dojít k potřebě logovat mnohem více informací, případně by mohl být simulátor rozšířen na větší počet robotů, což by vedlo k dalšímu zpomalení logování v průběhu hry.

3.3 Paralelizace výběru strategií pro levý a pravý tým

3.3.1 Důvod paralelizace

V současné verzi hry probíhá v hlavní smyčce hry sekvenčně výběr optimální strategie a provedení taktiky pro první tým a poté pro druhý. Pokud tedy jeden z týmů má rychlejší algoritmus například na vyhledání optimální strategie, či provedení taktik, je mu to v současné verzi k ničemu, neboť i přes rychlejší provedení jeho algoritmu bude muset do příštího provedení čekat celý jeden krok hry (tedy i na provedení algoritmu soupeře). Neexistuje zde tedy ani ta možnost, že by tým s lepší sadou pravidel ve strategii mohl reagovat na situaci na hřišti rychleji než tým druhý. Proto byla zavedena paralelizace výpočtu hledání optimální strategie a provedení taktiky každého týmu. Konkrétně se jedná o paralelizaci již zmíněných metod TickStrategy, třídy reprezentující strategie jednotlivých týmů (leftPlayerStrategy, rightPlayerStrategy).

3.3.2 Třída Mutex

Jelikož vlákna zajišťující výpočty optimální strategie a taktiky každého z týmů budou probíhat paralelně s hlavní smyčkou hry, ve které bude docházet k zapisování a čtení informací do/z kontejneru, je třeba zabránit chybám spojeným s touto paralelizací. Mohlo by dojít k tomu, že by metody v hlavní smyčce hry četly informace z kontejneru zrovna ve chvíly kdy by byly tyto informace přepisovány novými a poté by například část robotů byla překreslena na základě informace staré a část na základě informace dodané novým výpočtem z vlákna pro daný tým. Obecně se pro tyto účely používají nějaké formy zámků, které zabrání přístupu k určitému objektu zároveň více vlákny, a tedy nedojde k tzv. souběhu (anglicky race condition). K této chybě pochopitelně nedochází při sekvenčním provedení metody kroku hry. Pro účely odstranění souběhu je v .NET frameworku k dispozici třída Mutex [8] z jmenného prostoru System.Threading, jejíž použití je následující:

```
static Mutex mut = new Mutex();

static void main() {
    Thread myThread = new Thread(Method1);
    Thread myThread2 = new Thread(Method2);
    myThread.Start();
    myThread2.Start();
}
static void Method1() {
    UseResource();
}
static void Method2() {
    UseResource();
}

static void UseResource()
{
    mut.WaitOne();
    Console.WriteLine(...);
    mut.ReleaseMutex();
}
```

Výpis 5: Použití třídy Mutex

Pokud "obalíme"blok kódu v metodě UseResource metodami WaitOne a ReleaseMutex objektu *mut* třídy Mutex, je k tomuto bloku možno přistoupit vždy pouze z jednoho

vlákna, zatímco druhé je v tu chvíli pozastaveno, a to až do doby, než první vlákno blok opustí a uvolní výlučný zámek na tomto bloku [5].

Pro potřeby simulátoru tedy byla vytvořena statická třída SharedMutex zajišťující přístup ke statickému objektu třídy Mutex kdekoliv z aplikace. Zvolené části simulátoru, kde může dojít k souběhu, používají právě tento Mutex k zajištění korektnosti dat.

3.3.3 Implementace paralelizace

Při spuštění simulace jsou vytvořena dvě nová vlákna, která v průběhu celé hry ve smyčce provádějí výpočet optimální strategie obou týmů. Metody TickStrategy tříd leftPlayer-Strategy a rightPlayerStrategy (viz 2.3 - Strategie) byly tedy přesunuty z hlavní smyčky simulátoru do samostatných metod prováděných jednotlivými vlákny. Vlákna běžící na pozadí dodávají výsledky výpočtů do kontejneru storage, ze kterého jsou v hlavní smyčce tato data přečtena a situace je překreslena v hlavním okně.

Při spuštění simulace je vyvolána metoda StartThreads třídy Simulator, spouštějící jednotlivá vlákna. Na následujícím úryvku kódu je ukázána implementace pro levý tým.

```
public void StartThreads() {
    Thread t1 = new Thread(new ThreadStart(LeftTeamTick));
    Thread t2 = ...

    t1.Start();
    t2.Start();
}

public void LeftTeamTick() {
    while(true) {
        // získání výlučného zámku nad storage
         SharedMutex.getMutex().WaitOne();
        // metoda pro výběr optimálního pravidla a provedení taktiky levého týmu
        leftPlayerStrategy.TickStrategy(storage)
        // uvolnění výlučného zámku
        SharedMutex.getMutex().ReleaseMutex();
    }
}
```

Výpis 6: Paralelizace výběru pravidel

Třída Mutex je také použita v hlavní smyčce simulátoru jelikož dochází k zápisu současných pozic objektů na hřišti a následnému čtení nově vypočtených pozic. Paralelizací bylo docíleno toho, že nyní může být pro tým přínosem lépe promyšlená strategie, protože

bude moci rychleji reagovat na situaci na hřišti pokud by měl například méně záznamů v množině pravidel. Paralelizace je však velmi důležitá, pokud by v nějaké následující verzi simulátoru mohl být pro různé týmy použit i různý algoritmus pro hledání optimálního pravidla a pro provedení taktiky, kdy by týmy musely zvažovat i jeho časovou náročnost. Dalo by se poté objektivně posoudit, který algoritmus je pro tým výhodnější. To však zatím není účelem simulátoru. Podstatnou nevýhodou je například problém při debugování aplikace, jelikož běží 1 hlavní vlákno a 2 další obstarávající výpočty.

3.4 Implementace pohybu otáčení robotů

3.4.1 Současný stav otáčení

Doposud nebyl v simulátoru implementován pohyb otáčení robotu. Pokud se tedy robot v jednom kroku hry pohyboval jedním směrem a v dalším kroku mu byl vypočten směrový vektor směřující například na úplně opačnou stranu, došlo k okamžitému otočení robota a jeho pohybu ve směru novém. Takovéto chování pochopitelně vůbec neodpovídá reálné hře. Proto bylo potřeba implementovat pohyb otáčení robotů.

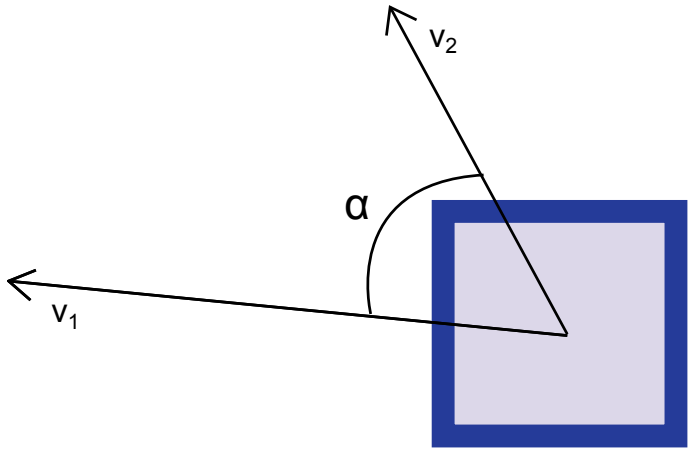
3.4.2 Implementace nového způsobu otáčení robotů

Jelikož je pohyb (a tedy i směr pohybu) robotů řešen jako pohyb pomocí směrového vektoru, který je objektům hrací plochy přepočten metodou GoTo na základě informace o pozici na kterou se mají pohnout (vektor positionMove), bude implementace pohybu otáčení probíhat právě změnou směru a velikosti vektoru velocity každého robota. Je potřeba zvážit jak budeme chtít, aby se robot v otáčení choval. Současný model je nevyhovující kvůli jeho nereálnosti, robot se může otočit při libovolné rychlosti (velikosti vektoru *velocity*) o libovolně velký úhel, otáčí se tedy ve skocích.

Nejprve bylo nutné omezit úhel, o který se robot v daném kroku může otočit - tímto omezením vpodstatě rozkouskujeme otočení v jednom kroku o velký úhel, na malé úhly v průběhu více herních kroků. Takovým modelem bychom sice docílili jakéhosi pseudo-otáčeni, avšak v reálné hře mají roboti tu vlastnost, že se mohou otočit i velmi rychle,

vpodstatě okamžitě, a to pokud stojí na místě (toto jim umožňuje jejich typ pohonu, kdy může jedno kolečko být zastaveno a druhým kolečkem dojde k otočení na místě). Bylo tedy potřeba měnit tento výsledný omezený úhel také v závislosti na současné rychlosti robota.

Nejprve tedy definujme úhel α, jako úhel mezi jednotlivými směrovými vektory robota vždy dvou po sobě jdoucích herních kroků (viz obrázek 11.).



Obrázek 11: Úhel mezi směrovými vektory

Pokud je tento úhel větší než určitá stanovená hodnota (v našem případě velikost 30°), je potřeba úhel omezit na nižší hodnotu. Tím docílíme toho, že se v jednom kroku nemůže úhel změnit o více než právě tuto hodnotu oproti kroku minulém (nedojde tedy nikdy k otočení směru robota o více než stanovenou hodnotu během jednoho kroku). Velikost úhlu, o kterou se robot v jednom kroku hry může otočit je poté závislá na aktuální velikosti směrového vektoru, jelikož tato velikost reprezentuje rychlost pohybu robota (podle směrového vektoru robota dochází ve fyzikálním enginu k posunu robota po hrací ploše, velikost tohoto vektoru poté určuje jakou rychlostí bude robot posunut). Vztah mezi rychlostí robota a jeho možném úhlu natočení v jednom kroku hry bude následující:

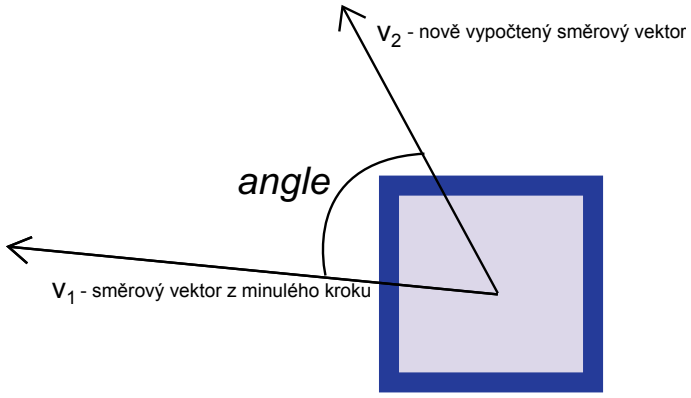
$$angle = (1 - speed/1) * maxRotate + minRotate$$

$speed$ udává velikost směrového vektoru pohybu robota. Tuto velikost můžeme vypočítat následujícím vzorcem: pro vektor $v(x, y)$ platí $speed(v) = \sqrt{v_x * v_x + v_y * v_y}$ [4]. $maxRotate$ je konstanta, udávající o kolik se může robot maximálně v daném kroku natočit. Po několika testech byla tato konstanta zvolena o velikosti 30. Robot se tedy může natočit až o 30°) pokud je jeho velikost jeho směrového vektoru ($speed$) blízká nule. $minRotate$ je rovněž konstanta, udávající minimální možné natočení v jednom kroku. Tato konstanta byla zvolena o velikosti 5, robot se tedy v daném kroku může natočit minimálně o 5°, pakliže se velikost jeho směrového vektoru blíží hodnotě 1.

Je také třeba vyřešit výslednou velikost směrového vektoru robota při otočení (neboli jeho rychlost). Tato rychlost je závislá na úhlu, který je vypočten mezi původním směrovým vektorem a nově vypočteným směrovým vektorem v tomto kroku, závislým na původní rychlosti robota. Vztah mezi těmito veličinami je následující:

$$speed = (1 - angle/180) * 0.9 + 0.1;$$

Hodnota $angle$ je rovna již zmíněnému úhlu mezi směrovým vektorem z předchozího kroku a nově vypočteným vektorem pro současný krok (viz obrázek 12.).



Obrázek 12: Úhel mezi směrovými vektory

Hodnotou $speed$ je poté přenásoben směrový vektor robota, čímž docílíme zpomalení, případně zrychlení výsledného směrového vektoru.

Implementací tohoto algoritmu je tedy dosaženo přiblížení chování robota realitě. Pokud se robot pohybuje velmi rychle, je logické, že pakliže by se chtěl prudce otočit o veliký úhel, musí nejprve patřičně zpomalit na požadovanou rychlost a naopak při malé rychlosti (či stání) může provádět otáčení o veliký úhel.

3.5 Spolupráce s ostatními kolegy

V rámci spolupráce s ostatními kolegy, pracujícími taktéž na simulátoru, byly implementovány některé dodatečné metody a bylo rozhodnuto, do které části simulátoru budou jejich práce implementovány. Jelikož obě práce (implementace hledání cesty pro robota pomocí metody regrese a predikce chování soupeřových robotů) vyžadují reakci na současnou situaci v každém kroku hry, byly funkce těchto prací využity přímo v hlavní smyčce simulátoru.

3.5.1 Implementace metody odstranění duplicit u predikce chování

Jedním z výsledků práce zabývající se predikcí chování je také dynamické tvoření nových pravidel strategií v průběhu hry a jejich následné použití pro tým. Tyto pravidla jsou tedy ukládány do množiny již existujících pravidel strategií použitelných pro daný tým. Bylo tedy potřeba vyřešit to, aby nedocházelo k ukládání duplicitních, nebo velmi podobných pravidel v porovnání s pravidly, které se už v množině nachází. Pro tento účel byla použita upravená verze metody pro výběr nejvhodnějšího pravidla, kdy je před jeho vložením nové pravidlo nejprve porovnáno s ostatními z množiny metodou MatrixDistance (viz 3.1.2) a k vložení dojde jen pokud je rozdíl (nebo také vzdálenost) mezi těmito dvěma pravidly (výsledek metody MatrixDistance) větší než zvolená konstanta. Po provedených testech byla konstanta zvolena na takovou hodnotu, aby nedocházelo k duplicitám u strategií, tudíž aby byly dostatečně rozeznány unikátní strategie, které by mohl tým využít (hodnota bylo zvolena taková, aby k přídání pravidla došlo například u pravidla které má oproti ostatním pravidlům vzdálenost matic minimálně o 3, což odpovídá například změně pozic 3 robotů o 1 pozici v gridových souřadnicích).

3.5.2 Implementace pohybu robotů pomocí metody regrese

V práci zabývající se pohybem robotů pomocí metody regrese bylo potřeba upravovat pohyb robotů, tedy přímo výsledný vektor velocity vypočtený metodou GoTo každého robota z pozice na kterou se má daný robot pohnout. Metody pro změnu těchto vektorů bylo tedy nutné použít přímo v hlavní smyčce simulátoru a reagovat tak v každém kroku na nové informace o tom kam se mají roboti pohnout.

4 Závěr

Cílem práce bylo implementovat různé optimalizace simulátoru pro hru fotbal robotů. Byly vybrány určité oblasti, které nebyly zcela optimálně navrhnuty, a proto došlo k jejich zlepšení. Jednalo se o optimalizaci výběru optimální strategie týmu, změnu procesu logování hry, paralelizaci provedení stategií pro jednotlivé týmy a také pohyb otáčení robotů. V případě optimalizace výběru optimálního pravidla strategie jsem naimplementoval nový způsob výběru strategie a také nový způsob jakým jsou měřeny vzdálenosti herních objektů v gridových souřadnicích, kdy objekty v sousedících buňkách mají vždy stejnou vzdálenost, nehledě na to, zda sousedí diagonálně nebo v rovině. Objektivně posoudit, zda je tento nový způsob pro tým výhodnější, je dle mého názoru nemožné, posuzoval jsem pouze výsledky výběru u určité množiny pravidel a herních situací a ukázalo se, že nový způsob vybírá pravidla přesněji, než původní algoritmus. Je také zřejmé, že nový způsob je časově méně náročný, což by se ještě více projevilo u simulace s větším počtem robotů. Toto bylo potvrzeno měřením.

Změnou procesu logování sice nedošlo k významné úspoře času (maximální naměřená úspora byla přibližně 1ms), v nové verzi je však logování přesunuto mimo hlavní smyčku hry.

Paralelizací hry jsem docílil toho, že tým s lépe navolenou sadou pravidel ve strategii nyní může častěji reagovat na hru, jelikož proces výběru jeho strategie a provedení taktik může být rychlejší než u soupeře a není již na provedení výpočtů soupeře závislý (tato změna by se však projevila nejvíce, pakliže by simulátor byl navržen tak, aby mohl každý z týmu implementovat i vlastní algoritmy pro provedení strategií a taktik).

V poslední fázi jsem implementoval pohyb otáčení robotů. Pohyb byl implementován tak, aby co možná nejvíce odpovídal reálnému pohybu robotů. Jelikož v předešlé verzi hry nebylo otáčení implementováno vůbec, působí teď pohyb robotů mnohem plynuleji a také mnohem více odpovídá chování reálných robotů.

V rámci práce jsem také spolupracoval s ostatními kolegy, kteří do stejné verze simulátoru implementovali různá vylepšení. Rozhodovali jsme do kterých částí simulátoru

budou jejich metody implementovány a v jednom případě byla použita upravená verze metody pro výběr optimálního pravidla strategie.

Společně jsme tedy vytvořili novou, v mnoha ohledech lepší verzi simulátoru.

4.1 Další vývoj

V budoucích fázích vývoje je potřeba zaměřit se na dokončení implementace taktik. Tato složka simulátoru je vpodstatě jádrem hry každého týmu a je tedy velmi důležité jak bude implementována. Při implementaci taktik by se dalo inspirovat některými použitými modely ostatních týmů. Další složkou, která by mohla být podstatně zlepšena, je řešení kolizí ve fyzikálním enginu. Metoda TickPhysics třídy PhysicalScene je v současné verzi asi největším bottleneckem simulátoru - při velkém počtu kolizí trvá její provedení řádově i desítky milisekund. Mohly by být použity některé úspěšné metody jakými řešili ostatní týmy kolizi objektů ve scéně (například detekce kolizí pomocí překrytí bounding boxů [10]), případně nějaký externí fyzikální engine.

5 Reference

[1] Albahari, Joseph a Ben Albahari. *C# 4.0 in a nutshell.* 4th ed. Sebastopol: O'Reilly, 2010.

[2] Liberty, Jesse. *Programming .NET 3.5.* Sebastopol: O´Reilly, 2008.

[3] Šimeček, Roman. *Fotbal robotů.* Diplomová práce na Fakultě elektrotechniky a informatiky Vysoké školy báňské. Ostrava, 2007.

[4] Kočandrle, Milan a Leo Boček. *Matematika pro gymnázia.* 3. vyd. Praha: Prometheus, 2010.

[5] Teilhet, Stephen a Jay Hilyard. *C# cookbook.* Sebastopol, CA: O'Reilly, 2004.

[6] *Stopwatch Class.* [online]. URL: ¡http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.aspx¿ Poslední revize 19.4.2012 [cit. 20.4.2012]

[7] Bolton, David. *How do I do High Resolution Timing in C# on Windows?* [online]. URL: ¡http://cplus.about.com/od/howtodothingsinc/a/timing.htm¿. Poslední revize 12.4.2012 [cit. 20.4.2012].

[8] *Mutex Class.* [online]. URL: ¡http://msdn.microsoft.com/en-us/library/system.threading.mutex.aspx¿ Poslední revize 17.4.2012 [cit. 20.4.2012]

[9] Snášel, V., Ochodková, E., Žo ltá, L., Wu, J. a A. Abraham. *Robot Soccer - Strategy Description and Game Analysis.* 2010.

[10] Klančar, G., Lepetič, M., Karba, R. a B. Zupanič. *Mathematical and Computer Modelling of Dynamical Systems.* 2003.

6 Přílohy

I. Příloha na CD/DVD

Obsah CD/DVD

Adresář	Popis
Robot soccer	Aplikace simulátoru
Text	Text bakalářské práce