Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Dokumentácia k inžinierskemu dielu

Tím BAREKO

Bc. Michal Baňas

Bc. Šimon Harvan

Bc. Ernest Loureiro

Bc. Daniel Lukáč

Bc. Marko Moravčík

Bc. Lukáš Rešutík

Bc. Dávid Roba

Vedúci projektu: Ing. Ivan Kapustík

Predmet: Tímový projekt I

Ročník: 2017/2018

Obsah

1. Úvod

Bc. Daniel Lukáč

RoboCup je medzinárodná súťaž v robotickom futbale, na ktorom sa zúčastňujú univerzity z rôznych krajín sveta. Existuje viacero líg pre RoboCup a jednou z nich je aj liga pre simulovaný robotický futbal. Fakulta informatiky a informačných technológií Slovenskej technickej univerzity sa venuje vývoju simulovaného robotického futbalu už od roku 2000.

Tento projekt je vyvíjaný každým rokom v rámci predmetov Diplomový projekt, Bakalársky projekt a Tímový projekt. Naša fakulta organizuje taktiež turnaje v simulovanom robotickom futbale s názvom RoboCup at FIIT. Počiatky tohto projektu boli vyvíjané v 2D prostredí a od roku 2006 sa naša fakulta zamerala na vývoj v 3D.

Hráčov v simulovanom robotickom futbale tak nahradili modely robotov, ktoré boli vytvorené na základe reálnej podoby robotov z nesimulovaného robotického futbalu. Každý hráč je zložený z viacerých častí, ktoré sú prepojené ohybnými kĺbmi a disponuje rôznymi funkciami pre pohyb, kopnutie do lopty, zorientovanie sa v priestore, postavenie a ďalšie, ktorým sa budeme v našom projekte venovať.

Hlavným cieľom projektu je pokračovanie na vývoji hráča simulovaného robotického futbalu, za účelom zdokonalenia už existujúcich riešení a vytvorenia ďalších. Súčasťou projektu je aj testovací nástroj pre prácu s agentom, ktorý budeme v našom projekte využívať. Testovací nástroj bol vyvinutý pre podporu budúceho vývoja projektu a obsahuje radu funkcií pre prácu s agentom.

Táto dokumentácia bude obsahovať technickú časť k práci, ktorá bude na projekte vykonaná a bude poskytovať na projekt ucelený pohľad, pre pochopenie jeho častí. V dokumentácii zahrnieme naše vylepšenia a vykonanú prácu a budeme sa venovať zisteniam skutočností, ktoré zanalyzujeme.

2. Celkový pohľad na Robocup projekt

Bc. Lukáš Rešutík

Výsledkom niekoľko ročného projektového snaženia v rámci Robocup-u na FIIT STU je vývoj 3D hráča robotického futbalu pod názvom JIM. V súčasnosti je postavený na programovacom jazyku Java s využitím XML a projekt je rozdelený do troch častí:

- Jim
- RoboCupLibrary
- TestFramweork

Program aktuálne beží na serveri rcsserver3d-0.6.7. Server má informácie o aktuálnom stave hry, ponúka grafické zobrazenie pomocou rcsmonitor3d.

Dôležitou súčasťou projektu je Wiki stránka projektu. Na jej doplnení, aktualizovaní jej štruktúry sa zamerali staršie tímy, avšak náš tím považuje za nutné Wiki neustále aktualizovať a dopĺňať, ktoré budú nápomocné pre nasledujúce tímové, bakalárske a diplomové projekty. Táto kapitola opisuje celkový pohľad na aktuálny stav systému a všetkých jeho súčastí.

2.1 Jim

Bc. Lukáš Rešutík

Predstavuje Java aplikáciu samotného agenta, ktorý sa pripojí k serveru a hrá futbal. Agent by mal dodržiavať nakonfigurovanú a naprogramovanú taktiku. Pred spustením agenta je nutné spustiť RCSS server. Aplikácia agenta obsahuje jednoduché ovládanie prostredníctvom GUI umožňujúce opätovné načítanie pohybov a preplánovanie. Inicializačným bodom agenta je trieda *sk.fiit.jim.init.Main*.

Agent dokáže kopať do lopty, otočiť sa o 60 stupňov, postaviť sa za maximálne za 3s, stabilizovať sa pri vstávaní, behať, chodiť. Taktiež má agent naimplementované základné vnímacie taktiky (či tím útočí alebo bráni).

Spustenie agenta je pomerne zdĺhavý proces. Na počítači so slabším výkonom čas spustenia sa pohybuje cca. 10 sekúnd. Na začiatku spúšťania Jima sa vždy vykonajú nasledovné operácie:

- Načítavanie pohybov z XML súborov v adresári moves
- Načítanie anotácií
- Spracovanie parametrov príkazového riadku
- Spustenie TFTP servera použitého pre komunikáciu s TestFrameworkom
- Prípadné vytvorenie GUI
- Vstup do hlavného cyklu

2.1.1 Pohyby

Správanie agenta je riadené kódom, a možno ho rozdeliť do niekoľkých vrstiev. Prvou vrchnou vrstvou je plánovač riadený triedou *HighSkillPlanner* nachádzajúci sa v balíku *sk.fiit.jim.agent.highskill.runner*. Plánovač vytvára inštancie tzv. high skillov (vyšších pohybov). High skilly počas svojho behu vyberajú nižšie pohyby (nižšia vrstva) tzv. low skill, ktorý sa ma vykonať. Low skill predstavuje len skupinu metadát, zapísaných pomocou XML, pričom určujú jeho názov a ďalšie informácie.

SkillsFromXmlLoader he trieda, ktorá slúži na načítanie pohybov z XML súborov. Triedy *LowSkills* a *Phases* spravujú objekty fáz a pohybov a sú umiestnené v balíku *sk.fiit.jim.agent.moves*. Keďže načítanie pohybov je značne pomalé, ukladá sa ich parsovaná podoba do súboru *./movecache*, ktorý sa pri budúcom načítaní použije, pokiaľ od jeho vytvorenia nebol žiadny pohyb zmenený. V takom prípade sa súbor vytvorí nanovo.

2.2 RoboCupLibrary

Bc. Daniel Lukáč

Knižnica RoboCupLibrary obsahuje triedy, ktoré sú potrebné pre rôzne matematické výpočty, anotácie alebo reprezentáciu geometrických objektov. Táto knižnica sa vužíva len minimálne a v súčasnosti neexistuje dokumentácia alebo bližší popis pre získanie informácii o jej využití a to ani na Wiki. Avšak aj napriek jej minimálnemu využitiu je pre projekt potrebná.

Náš tím v súčastnosti obohatil túto knižnicu efektívnejšími výpočtami sin a cos funkcií, ktoré sú implementované v celom projekte. Triedy z knižnice sa využívajú ako v Test frameworku tak aj v agentovi. Cieľom je čo najviac odstrániť tieto závislosti medzi Jim-om a TestFrameworkom, čo však v súčasnosti nie je dodržané. V akademickom roku 2014/2015 prebehol kompletný refactoring tejto knižnice, tímom Infinity. Refactoring zahŕňal dodržanie konvencie kódu, zmazanie nepotrebných tried a funkcií a tak isto oddelenie testov od logiky knižnice.

2.3 TestFramework

Bc. Dávid Roba

Využíva sa na získanie spätnej väzby od hráča. Hlavným cieľom je vytvoriť robotického futbalového trénera, ktorý by schopný učiť hráčov novým taktikám a pohybom automaticky.

Na interakciu medzi agentom a serverom sa využíva posielanie správ, ktoré obsahujú tzv. S-

výrazy, pričom hráč len odosiela správy a TestFramework ich len príjma. Server (TestFramework) je implementovaný pomocou triedy AgentMonitor v balíku sk.fiit.testframework.monitor, ktorá slúži na spracovanie nových spojení od agentov. Každé z týchto spojení je reprezentované triedou AgentMonitorThread, ktorá sa nachádza tiež v balíku sk.fiit.testframework.monitor a slúži na spracovanie prichádzajúcich správ od agenta prostredníctvom triedy AgentMonitorMessage, ktorá je umiestnená v rovnakom balíku ako trieda AgentMonitorThread. V agentovi sa na komunikáciu s TestFrameworkom využíva trieda TestFrameworkCommunication, ktorá sa nachádza v balíku sk.fiit.jim.agent.communication.testframework.

2.4 Identifikované nedostatky

Bc. Dávid Roba

Na základe testovania projektu a tiež podľa dokumentácií z predchádzajúcich rokov boli identifikované nasledujúce nedostatky:

- Použité trigonometrické funkcie (*sin* a *cos*) patria medzi najpomalšie matematické operácie
- Pri vnímaní čiar sa do žiadnej miery nezohľadňuje náklon hráča.
- Komunikácia Jima s TestFrameworkom cez pomalý TFTP protokol
- Chôdza pomocou ZMP (Zero Moment Point) nie je implementovaná v žiadnej z taktík.
- Meranie disciplín turnaja v TestFramework-u neobsahuje všetky disciplíny turnaja.

3. Ciele

Bc. Daniel Lukáč

Na úvodnom stretnutí sme sa oboznámili s problematikou RoboCupu a stretli sme sa s tímom z predošlého akademického roka. Zástupcovia tímu nám odprezentovali časť och práce a oboznámili nás s niektorými nedostatkami. Na základe týchto skutočností sme stanovili ciele našej práce na projekte.

3.1 Ciele pre zimný semester

3.1.1 Optimalizácia pomalých funkcií

Projekt obsahuje mnoho funkcií, ktoré je možné optimalizovať. V prvotnej fáze sme sa zamerali na funkcie počítajúce sin a cos, ktoré sme úspešne implementovali. K optimalizovaným funkciám je potrebné vytvoriť správne testovanie a sledovať ich správanie.

3.1.2 Zlepšenie orientácie Jima

Cieľom je pokračovať na rozpracovaní vnímania čiar, ktoré je rozpracované minuloročným tímom a pokúsiť sa implementovať funkciu pre vnímanie len samotných čiar. V prvotnej fáze je pre túto implementáciu potrebné venovať čas analýze už doposiaľ vytvorením funkciám pre vnímanie čiar.

3.1.3 Analýza a prípadná optimalizácia stabilizácie

Je potrebné analyzovať zisťovanie náklonu z pevných bodov ihriska, získaných zo servera a zisťovanie náklonu z gyroskopu Jima. Na základe analýz je potrebné pokúsiť sa zlepšiť stabilizáciu Jima. Cieľ súvisí s cieľom 2. Zlepšenie orientácie Jima.

3.1.4 Aktualizácia Wiki

Súčasťou projektu je Wiki webová stránka, z ktorej čerpajú informácie študenti pri riešení bakalárskych a diplomových projektov, ako aj pre tímy pokračujúce v rámci tímového projektu. Preto je potrebné túto stránku udržiavať aktualizovanú.

3.2 Ciele pre letný semester

Ako výstup z brainstormingu po zimnom semestri vzišla formulácia nasledových cieľov. Jednotlivé ciele budú aktualizované aj na tímovej stránke.

- Implementácia Kalmanovho filtra na pevne body.
- Implementácia Kalmanovho filtra na loptu.
- Implementácia zohľadnenia naklonu na os Z.
- Implementácia zohľadnenia náklonu pre os Y.
- Zozbieranie a zoskupenie podkladov pre pochopenie fyziky.
- Modifikácia logovania v TestFrameworku pre lepšiu prácu s výstupnými dátami.

4. Používané technológie

Bc. Lukáš Rešutík

4.1 Server

V rámci projektu prebieha komunikácia medzi agentom a serverom oboma smermi. Agent posiela informácie serveru o zmene nastavenia jednotlivých kĺbov a server posiela agentovi informácie o aktuálnom stave hry v S-správach.

Predchádzajúce tímy experimentovali s rôznymi verziami rcsserver3d na rôznych operačných systémoch (napr. Windows, Unix). Táto podkapitola sa venuje rôznym verziám, ich rozdielom a možnosti použitia na projekte.

Niekoľko hodín práce sme sa pokúšali spojazdniť novšie verzie servera (0.6.10, 0.6.9) avšak stále sa ich nepodarilo úspešne spustiť. Preto používame pôvodnú verziu 0.6.7. Nasledujúcim tímom preto doporučujeme používať starší server.

4.1.1 rcsserver3d-0.6.8

- Podporuje rozoznávanie tímov (identifikácia správ, od ktorého tímu prišla)
- Pridanie tabule so skóre
- Maximálne 11 druhov robotov

4.1.2 rcsserver3d-0.6.9

- Pridanie nového pravidla – lopta sa musí dotýkať protihráča alebo spoluhráča ak sa lopta nenachádza v stredovom kruhu
- 11 metrové kopy (penalty) sú priame

4.1.3 rcsserver3d-0.6.10

- Nové modeli pre vizuálne odlíšenie iných typov robotov

4.2 Pluginy

Bc. Lukáš Rešutík

V rámci vývojového procesu je veľmi výhodné použiť vybraté pluginy do nástroja Eclipse, ktoré zlepšujú kvalitu kódu a zároveň uľahčujú prácu v tíme BAREKO počas práce na projekte. Nižšie uvádzame len použiteľné pluginy. Tieto pluginy taktiež využívali, niektoré staršie tímy a doporučujú ich používať.

- FindBugs
- CheckStyle
- Eclipse PMD

4.2.1 FindBugs

Findbugs je plugin, ktorý detekuje chyby v jazyku Java, pričom využíva statickú analýzu založenú približne na 200 vzoroch. Jeho výstupom je poukázanie na zlý kód (nekonečné rekurzívy, nesprávne použtie Java knižníc, uviaznutia v kóde). Doporučuje sa použiť na rozsiahle projekty, kde je predpoklad zhruba 1 poruchy na 1000 – 2000 riadkov kódu.

4.2.2 Checkstyle

CheckStyle je plugin, ktorého úlohou je dohliadať na dodržiavanie štandardov v kóde. Výsledkom tohto pluginu je zlepšená čitateľnosť kódu. V tímovom prostredí, kde každý člen má svoj štýl písania kódu (napr. na základe programovacieho jazyka), je nutné mať jeden štandard písania kódu, ktorý je zhrnutý v metodike code conventions.

4.2.3 Eclipse PMD

Eclipse PMD je plugin, ktorý integruje analyzátor zdrojového kódu do prostredia Eclipse. Účelom PMD je vyhľadanie zlých vzorov kóde. Tento plugin, po každom uložení zdrojového kódu, preskenuje kód a nájde potencionálne problémy (bugy, neoptimálne, duplicitné, kognitívne náročné časti kódu). Tento plugin ponuká možnosť automatickej opravy, ak je to možné.

5. UT Austin

Bc. Ernest Loureiro

Tím UT Austin Villa je tímom Texaskej Univerzity v Austine (US), ktorý pôsobí v tejto oblasti od roku 2004. Patria medzi svetovú špičku v simulovanom robotickom futbale a od roku 2011 sú majstrami sveta, s výnimkou roku 2013, kedy skončili na druhom mieste. Svoj úspech pripisujú tzv. „vše-smerovej“ chôdzi, pri ktorej robot iba prekračuje na mieste a smer chôdze je udávaný pomocou náklonu, tj. preneseniu váhy. Vďaka takejto implementácii sa môže robot hýbať všetkými smermi, čo z neho robí veľmi rýchleho protivníka. Túto implementáciu uverejnili na portály GitHub v januári 2016. Na rozdiel od robota na našej fakulte, robot tímu Austin Villa je naprogramovaný pomocou C++, kvôli čomu je komplikované niektoré funkcie prevziať a aplikovať ich na nášho robota. Zároveň je tento kód vyvíjaný pod OS Ubuntu, ktorým náš tím nedisponuje.

6. Wiki

Bc. Ernest Loureiro

Hlavnou prácou tímu A55 Kickers (akad.rok 2012/2013) bola wikipédia k projektu RoboCup na našej fakulte. Táto wikipédia zoskupuje všetky poznatky o tomto projekte naprieč ľuďmi a tímami, ktoré na tomto projekte pracovali. Je rozčlenená do siedmich častí nasledovne:

1- Analýza tímov – obsahuje analýzy všetkých predchádzajúcich tímov, ktoré na projekte pracovali, ako aj zahraničných tímov súťažiacich v RoboCupe

2- Jim – obsahuje informácie o agentovi, ako s ním pracovať, ako implementovať nové techniky a skilly

3- TestFramework – popisuje fungovanie a implementáciu aplikácie TestFramewok

4- Wiki – Obsahuje informácie pre prácu so samotnou wikipédiou

5- Záverečné práce – záverečné práce diplomantov a bakalárov

6- Ruby old – staré dokumenty z čias keď niektoré funkcie boli implementované pomocou jazyka ruby

7- Nezaradené – obsahuje všetky ostatné dokumenty

Pomocou tejto wikipédie je možné prehlbovať svoje znalosti o projekte na základe zdedených dokumentov a následne na týchto znalostiach stavať pokrok v tomto projekte.

Náš tím prispel k fungovaniu tejto wikipédie aktualizáciou návodov, konkrétne návodu na „rozbehanie“ prostredia pod OS X, a následným formátovaním celej stránky s návodmi a inštaláciami, nakoľko bola táto stránka spísaná neprehľadne a častokrát nezrozumiteľne. Do budúcna by sme chceli opraviť nefunkčné odkazy a tejto stránke, ktoré nám pri inštalácii spôsobovali problémy, a aktualizovať návody na jednotlivé operačné systémy.

7. Ovládanie Simsparku

key	function
q	quit monitor
left mouse button	mouse look
right mouse button	move camera up
keypad plus	move camera up
pageup	move camera up
pagedown	move camera down
keypad minus	move camera down
a	move camera left
left arrow	move camera left
d	move camera right
right arrow	move camera right
w	move camera forward
up arrow	move camera forward
s	move camera backward
down arrow	move camera backward
1	camera to left goal
2	camera to left corner
3	camera to middle left
4	camera to middle right
5	camera to middle
6	camera to right corner
7	camera to right goal
l	free kick left
r	free kick right
k	kick off
b	drop ball
m	move agent
n	shoot ball
x	move ball
p	pause the playback of a log file
f	move forward in the log file
b	move backwards in the log file
l	playback the log file

Obr. 1: Základné ovládanie aplikácie SimSpark

Pozn.: číslovanie obrázkov a tabuliek je prispôsobené konkrétnym dokumentom.

8. Implementácia vlastnej funkcie sin/cos

Bc. Marko Moravčík, Bc. Michal Baňas

Po analýze časti kódu, kde dochádza k výpočtom sin a cos, prostredníctvom knižnice Math (Math.sin(RADIAN_ANGLE) , Math.cos(RADIAN_ANGLE)), sme sa zhodli, že dokážeme zvýšiť výkonnosť systému implementovaním vlastných funkcií sin a cos, ktoré fungujú nasledovne:

Funkcia *calculateCosAndSin* slúži na výpočet sin a cos , a následne uloženie výsledných hodnôt do tabuľky.

```
public void calculateCosAndSin() {

    double actAngle = 0;

    while(actAngle < 90) {
        sinArray[(int)(actAngle*100+0.05d)] = Math.sin(Math.toRadians(actAngle));
        cosArray[(int)(actAngle*100+0.05d)] = Math.cos(Math.toRadians(actAngle));

        actAngle += 0.01;
        actAngle = (double) Math.round(actAngle * 100) / 100;
    }
}
```

Obr.2: Funkcia *calculateCosAndSin*

Funkcia *getCos*, vyberie z vyrátanej tabuľky hodnotu cos podľa zadaného uhla.

```
public double getCos(double angle) {

    angle = Math.toDegrees(angle);
    if (angle >= 0 && angle < 90) {
        return cosArray[(int)(angle*100+0.05d)];
    }
    else if (angle >= 90 && angle < 180) {
        return -cosArray[(int)((180-angle)*100+0.05d)];
    }
    else if (angle >= 180 && angle < 270) {
        return -cosArray[(int)((angle-180)*100+0.05d)];
    }
    else if (angle >= 270 && angle < 360) {
        return cosArray[(int)((360-angle)*100+0.05d)];
    }
    else return 0;
}
```

Obr.3: Funkcia *getCos*

Funkcia *getSin*, vyberie z vyrátanej tabuľky hodnotu sin podľa zadaného uhla.

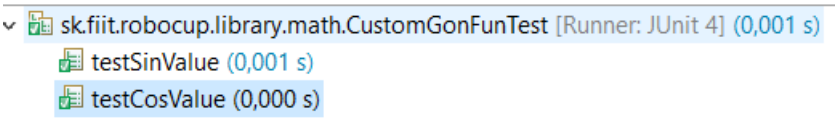
```
public double getSin(double angle) {

    angle = Math.toDegrees(angle);
    if (angle >= 0 && angle < 90) {
        return sinArray[(int)(angle*100+0.05d)];
    }
    else if (angle >= 90 && angle < 180) {
        return sinArray[(int)((180-angle)*100+0.05d)];
    }
    else if (angle >= 180 && angle < 270) {
        return -sinArray[(int)((angle-180)*100+0.05d)];
    }
    else if (angle >= 270 && angle < 360) {
        return -sinArray[(int)((360-angle)*100+0.05d)];
    }
    else return 0;
}
```

Obr.4: Funkcia *getSin*

Funckie sú implementované v triede *CustomGonFun* v balíku *sk.fiit.robocup.library.math* v projekte *RoboCupLibrary*.

Naša implementácia bola následne otestovaná s využitím JUnit testov. Ako prvá sa otestovala správnosť výsledkov, ktoré vypočítali naše funkcie.

Výsledok testu:



Obr.5: Ukážka testov

Testom sme dokázali, že výsledky sú správne. Testovala sa aj rýchlosť vykonania funkcii v porovnaní s pôvodnými funkciami z knižnice Math. Pri 10 000 000 opakovaniach. Výsledky s využitím našej implementovanej funkcie boli niekoľkokrát rýchlejšie ako pri pôvodnej (približne o 2 sekundy).

9. Návod k spusteniu test frameworku

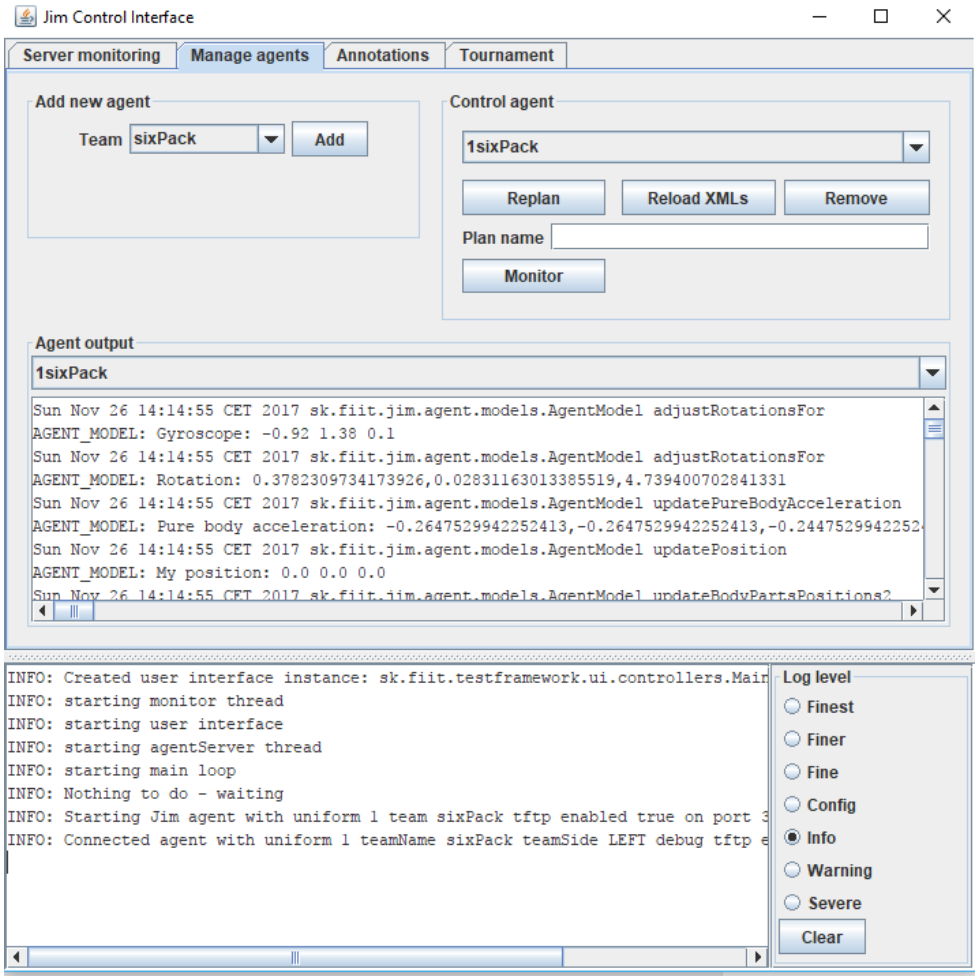
Bc. Lukáš Rešutík

9.1 Bežné spustenie

V rámci práce na tímovom projekte tímom BAREKO, tento dokument opisuje ako pridať Jima do TestFrameworku. Štandardné spustenie spustenie prebieha v nasledujúcich krokoch:

1. Spustenie servera rcssserver3d
2. Spustenie monitru rcssmonitor3d
3. Spustenie mainu v TestFrameworku (sk.fiit.testframework.init)
4. V zobrazenom okne TestFrameworku prepnúť na kartu Manage Agents
5. Kliknúť na tlačidlo Add

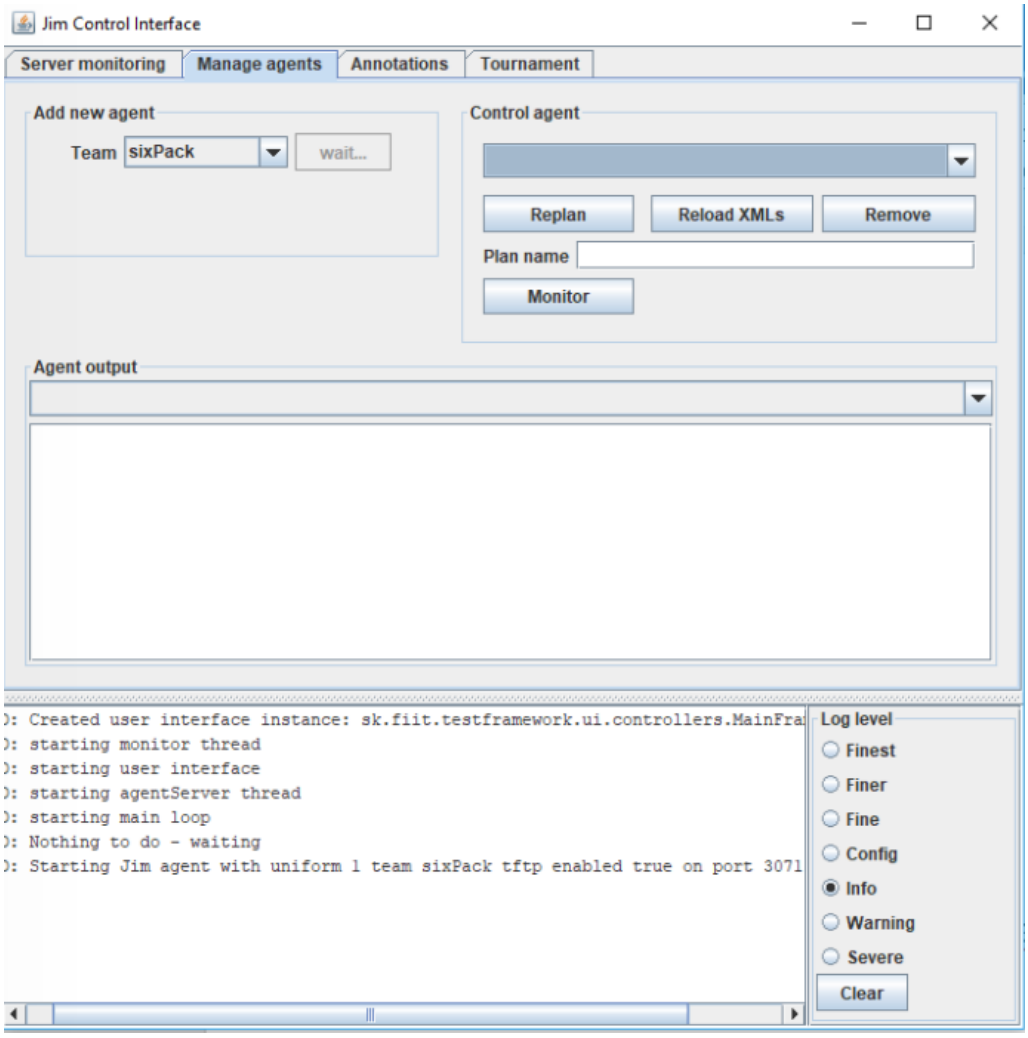
Ak všetko prebehlo správne okno by malo končiť s hláškou a vyzerať ako na obrázku 1: *„INFO: Connected agent with uniform 1 teamName sixPack teamSide LEFT debug tftp enabled: true with port 3071“.*



Obr. 6: Ukážka správneho spustenia

9.2 Spustenie pri zamrznutí tlačidla Add

V prípade že pri pridávaní Jima tlačidlom Add, toto tlačidlo zamrzne a ostane v stave zobrazenom na obrázku 2.



Obr. 7: Tlačidlo Add zamrzlo

Treba použiť nasledujúci postup:

1. Spustenie servera rcssserver3d
2. Spustenie monitoru rcssmonitor3d
3. Spustenie mainu v TestFrameworku (sk.fiit.testframework.init)
4. Spustenie mainu v Jimovi (sk.fiit.jim.init)
5. Vypnúť inštanciu Jima
6. V zobrazenom okne TestFrameworku prepnúť na kartu Manage Agents
7. Kliknúť na tlačidlo Add

10. Starý Kalmanov Filter

Bc. Dávid Roba, Bc. Marko Moravčík

10.1 Analýza starších tímov FIIT ohľadom kalmana

Ako prvý, tím v roku 2009 implementoval kalmanov filter do projektu 3D futbal. Od vtedy ho používali všetky ostatné tímy pričom nedošlo k žiadnym zmenám v jeho implementácií.

Okrem využitia Apache commons Kalman filter knižnice, ktorá vykonáva výpočty Kalmana, tak došlo k implementácií nasledovných tried:

- KalmanForVector – Kalmanov filter pre vektor
- KalmanForVariable – popisuje triedu určujúcu výpočet linearno-kvadratického Gaussovho regulátora pre premennú
- KalmanTest – trieda, slúžiaca na testovanie Kalmanovho filtra
- KalmanAdjuster – trieda slúžiaca na výpočet pozície objektov na ihrisku(lopta, zástavky) použitím kalmanovho filtra, čím sa redukuje šum a tak sa znižuje chybovosť vnášaná serverom -> žiadna zmena od roku 2009
- KalmanAdjusterTest - tri testy pre Kalmanove filtre. Ich úlohou je preveriť funkčnosť metód zodpovedných za redukciu šumu pri výpočte pozície lopty a fixného bodu (ľavá rohová zastávka domáceho tímu)

10.2 Analýza implementácie kalmana

10.2.1 Definícia

Algoritmus, ktorý používa sériu meraní pozorovaných v priebehu času, obsahujúcich štatistický šum a iné nepresnosti na odhad premenných v širokom spektre procesov. Využíva k tomu nielen naposledy nameraných dát a model systému, ale aj vektor údajov o predchádzajúcom stave systému. Je široko používaný pre spracovanie signálov, navigáciu, robotiku a ďalšie úlohy.

10.2.2 Výhody

- Dobré výsledky v praxi z dôvodu optimality a štruktúry
- Pohodlná forma pre online spracovanie v reálnom čase
- Jednoduché formulovanie a implementácia s ohľadom na základné pochopenie
- Meracie rovnice nemusia byť obrátené

10.2.3 Kroky

Lineáre systémy

V uvedených rovniciach *A*, *B* a *H* sú matice, *k* je časový index, *x* je tzv. stav systému, *u* je známy vstup do systému a *w* a *z* sú šumy. Premenná *w* reprezentuje tzv. procesný šum a *z* tzv. merací šum. Každá z týchto veličín je (obvykle) vektor a preto obsahuje viac ako jednu zložku. Vektor *x* obsahuje všetky dostupné informácie o súčasnom stave systému, ale nie je možné ho merať priamo. Namiesto toho meriame *y* , ktoré je funkciou x (teda *y=f*(*x*)), ale je "zašumená" šumom *z* . Je možné použiť *y* ako pomoc na získanie odhadu x, ale nemôžeme nutne vziať informáciu z *y* ako reprezentujúcu hodnotu, pretože obsahuje aj šum.

- stavová rovnica prechodu $x_{k+1} = Ax_k + Bu_k + w_k$
- rovnica výstupu $y_k = Hx_k + z_k$

Keďže je pohyb hráča priamočiary, môžeme povedať, že stav pozostáva z jeho polohy p a jeho rýchlosti v. Vstupná veličina u je nariadené zrýchlenie a výstupná veličina y je meraná poloha. Povedzme, že sme schopní meniť akceleráciu a merať polohu každých Ts sekúnd. V takomto prípade bude rýchlosť v definovaná rovnicou:

$$v_{k+1} = v_k + T_s\, u_k$$

To znamená, že rýchlosť vzdialená jednu vzorku od teraz (teda *Ts* sekúnd) bude rovná súčasnej rýchlosti prenásobenej nariadeným zrýchlením *u* a intervalom *Ts*. Predošlá rovnica však nedáva presnú hodnotu pre T_{k+1}. Namiesto toho bude rýchlosť ovplyvnená šumom. Šum rýchlosti je náhodnou premennou, ktorá sa mení s časom. Preto realistickejšie zapísaná rovnica vyzerá nasledovne:

$$v_{k+1} = v_k + T_s\, u_k + \sim v_k$$

kde $\sim v_k$ je šum rýchlosti. Podobnú rovnicu je možne odvodiť aj pre polohu *p*:

$$p_{k+1} = p_k + T_s\, v_k + \frac{1}{2} T_s^2 u_k + \sim p_k$$

kde $\sim p_k$ je šum rýchlosti . Na základe čoho môžeme definovať stavový vektor, ktorý pozostáva z polohy a rýchlosti:

$$x_k = \begin{pmatrix} pk \\ vk \end{pmatrix}$$

Keďže je meraná výstupná veličina úmerná polohe, je možné do rovníc za premenné A,B a H dosadiť nasledovné matice:

$$X_{k+1} = \begin{pmatrix} 1 & Ts \\ 0 & 1 \end{pmatrix} x_k + \begin{pmatrix} \frac{Ts^2}{2} \\ Ts \end{pmatrix} u_k + w_k$$

$$Y_k = (1\ \ 0)\, x_k + z_k$$

Kde matica *A* sa nazýva predikčná matica a matica B sa nazýva riadiaca matica. Keďže predpoveď nemusí byť sto percentne presná, vstupuje v tomto bode do procesu Kalmanov filter. Riešenie Kalmanovým filtrom nie je možné aplikovať kým nie sú splnené určité predpoklady o šume. Je potrebné si uvedomiť, že v rovniciach (modelu systému) v úvode vystupujú dve premenné - *w* je šum v procese a *z* je šum v meraní. Taktiež je ďalej možné skonštatovať, že neexistuje korelácia medzi *w* a *z* - teda v akomkoľvek čase *k*, w_k a z_k sú nezávislé náhodné premenné. Potom je možné zadefinovať kovariančné matice šumu *Q* a *R* ako:

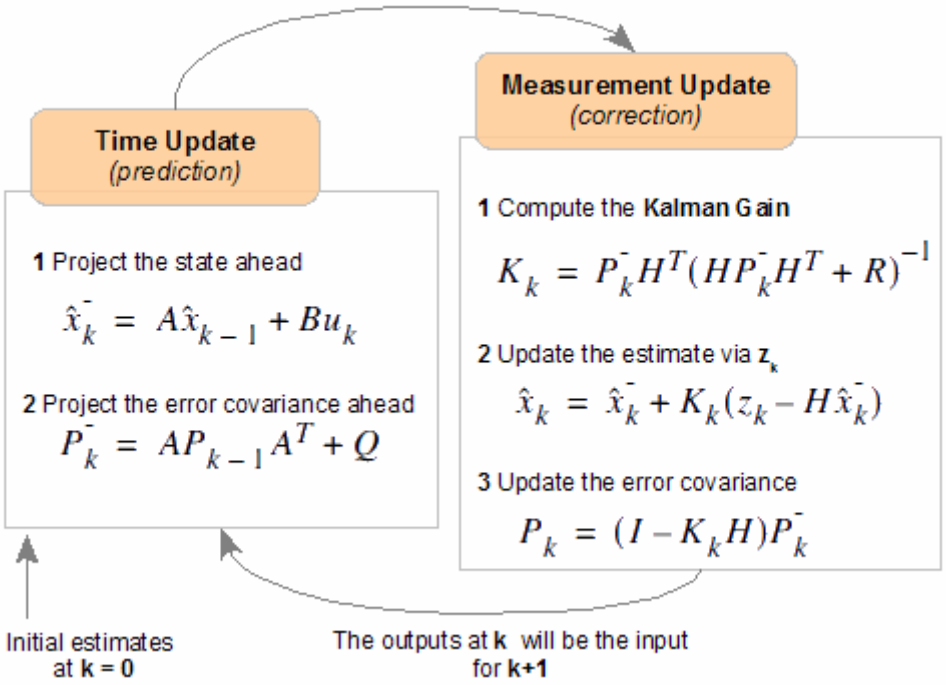
- Kovariancia šumu v procese: $Q = (w_k w_k^T)$
- Kovariancia šumu v meraní: $R = (z_k z_k^T)$

kde w^T a z^T označujú transpozíciu náhodných vektorov šumov. Teraz je možné použiť Kalamnové rovnice.

10.2.4 Kalmanove rovnice

Kalmanov filter sa delí na dve fázy. Predikčnú (časovú), v ktorej sa určuje nový najlepší odhad $\hat{x}_k^-$ ako predikcia z predchádzajúceho najlepšieho odhadu plus korekcia známych vonkajších vplyvov. A chyba kovariancie P_k^- je predpovedaná na základe predchádzajúcej chyby kovariancie spolu s určitou dodatočnou chybou z prostredia.

Tieto hodnoty sú následne použité v druhej fáze Kalmanovho filtra, ktorá sa nazýva Korekčná (filtračná), ktorá pozostáva z rovníc na nájdenie $\hat{x}_k$, ktorý predstavuje odhad *x* v čase *k* (teda, to čo chceme nájsť). Druhý výraz v rovnici je tzv. Korekcia P_k a reprezentuje veľkosť korekcie, akou treba korigovať odhad stavu na základe meraní. Rovnica K_k sa nazýva rovnica Kalmanovho zisku (zosilnenia). Rozborom rovnice pre K zistíme, že ak je šum z merania veľký, *R* bude veľké, takže *K* bude malé a nebudeme dávať veľkú váhu meraniu *y* , keď budeme počítať ďalšie. Na druhej strane však ak je šum v meraní malý, *R* bude malé a *K* veľké, čo znamená, že meraniu priradíme veľkú váhu pri výpočte ďalšieho.



Obr.8: Priebeh Kalmanovho filtra

10.3 Použitie Kalmanovho filtra na pozíciu hráča

Bc. Dávid Roba

Pozícia hráča sa určuje v triede *AgentModel.java* volaním funkcie *updatePosition(data)* z triedy *AgentPositionCalculator.java*, ktorej parametrom sú prijaté dáta zo servera. Pozícia sa určuje 3 rôznymi spôsobmi na základe toho, koľko fixných bodov a čiar vidí hráč v danej chvíli.

10.3.1 oneLineSeen(ParsedData)

V prípade, že hráč nevidí žiadny fixný bod a vidí iba jednu čiaru, určí sa jeho pozícia podľa nasledujúcich krokov.

1. Z posledných 10-tich pozícií hráča sa vyhodnotí, v ktorej časti ihriska sa hráč nachádza. Pri vyhodnotení sa ignoruje stredový kruh (t.j. uvažujú sa iba časti 1-4). Vyhodnotenie časti prebieha hlasovaním – každý záznam má vplyv na určenie podľa typu, akým bola daná poloha vyhodnotená. Celkové skóre pre danú časť ihriska je súčet typov vyhodnotení pozície hráča pre danú časť ihriska.
2. Identifikujú sa koncové body čiary – bod, ktorý je naľavo od hráča má väčšie *Phi* a bod, ktorý je napravo má naopak menšie *Phi*.

3. Oba koncové body čiary sa otočia v závislosti od aktuálneho natočenia hráča.
    a) Ak je rozdiel medzi y-ovými súradnicami bodov po natočení väčší ako rozdiel medzi x-ovými – hráč vidí horizontálnu čiaru (pohľad zhora na ihrisko, pričom bránky sú na ľavej a pravej strane)
    b) V opačnom prípade vidí hráč vertikálnu čiaru
4. Na základe predchádzajúcich pravidiel, súradníc bodov a časti ihriska, kde sa pravdepodobne nachádza, sa vytvorí čiara, kde by sa mohol hráč nachádzať.
5. Od poslednej známej pozície (posledný záznam v histórii) sa vytvorí kružnica s polomerom rovným *accDistance* z poslednej položky v histórii.
6. Výpočet priesečníka kružnice s úsečkou.
    a) Ak je jeden priesečník – nastaví sa daný bod ako miesto, kde sa hráč nachádza
    b) Inak sa vytvorí nová čiara z posledných dvoch záznamov v histórii. Vypočíta sa priesečník tejto čiary a čiary, kde by sa hráč mohol nachádzať.
        i. Ak neexistoval priesečník s kružnicou – priesečník dvoch čiar sa nastaví ako poloha, kde by sa mal hráč nachádzať
        ii. Inak sa nastaví priesečník kružnice, ktorý je bližšie k priesečníku dvoch čiar

10.3.2 oneFixedObjectSeen(ParsedData, int)

V prípade, že hráč vidí jeden kontrolný bod a čiary, funkcia vráti 2 namapované body z čiar, podľa nasledujúcich krokov.

1. Ak hráč vidí vlajku, pravdepodobne vidí aj ďalšie (minimálne) dve čiary. Najskôr sa zistí, či čiara, ktorú hráč vidí, má spoločný bod s vlajkou (vzor čiar „čiara a bod“). Informácia o tom, že daná čiara má spoločný bod, nepostačuje na určenie, o ktorú čiaru sa jedná (môžu byť dve). Ak agent vidí *F1R* – „horizontálnu čiaru“, vidí druhý bod čiary pod väčším uhlom theta (*theta1*) ako „vertikálnu čiaru“ (*theta2*). Uhly *theta1* a *theta2* sú theta uhly ku koncovým bodom čiary.
    a) Ak hráč vidí *F1R* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „horizontálnej čiary“ a *theta2* je bodom z vertikálnej čiary.
    b) Ak hráč vidí *F2R* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „vertikálnej čiary“ a *theta2* je bodom z „horizontálnej čiary“.
    c) Ak hráč vidí *F1L* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „vertikálnej čiary“ a *theta2* je bodom z „horizontálnej čiary“.
    d) Ak hráč vidí *F2L* a *theta1* > *theta2*, potom bod, ktorý hráč vidí pod uhlom *theta1*, je bodom z „horizontálnej čiary“ a *theta2* je bodom z „vertikálnej čiary“.

Následne sa vypočíta dĺžka videnej časti čiary a odpočíta/pripočíta sa dĺžka čiary k súradnici fixného bodu v závislosti od čiary a vlajky (fixný bod).

2. Ak hráč vidí bránku, najskôr sa zistí, či čiara, ktorú hráč vidí, je koncová „vertikálna čiara“ ihriska. Následne sa vypočíta vzdialenosť od priemetu bodu ku koncom čiary, ktoré vidí agent. Ďalej nasleduje výpočet absolútnych súradníc bodov – pripočítanie/odpočítanie vzdialenosti od y-ovej súradnice bránkového bodu v závislosti od uhla, pod akým hráč vidí daný bod.

10.3.3 justLinesSeen(ParsedData, int)

V prípade, že hráč vidí niekoľko čiar, vráti funkcia n–namapovaných bodov z čiar. Táto metóda na určenie pozície hráča ešte doposiaľ nebola implementovaná.

10.4 Zhrnutie

Bc. Marko Moravčík

Výsledkom tejto analýzy je zistenie, že Kalmanov filter sa v projekte nepoužíva pri určovaní pozície hráča. V projekte je implementovaný ako observer *KalmanAdjuster*, ktorý je naviazaný na prijatú správu zo servera. Implementované je to spôsobom, že sa automaticky optimalizuje/odhaduje poloha lopty a pevné rohy ihriska – vlajky.

11. Analýza vnímania prostredia

Táto kapitola sa zaoberá analýzou vnímania prostredia hráčom. Spôsobom ako reprezentuje priestor, akým spôsobom vníma čiary. Aký vplyv na vnímanie prostredia majú náklony hlavy, chôdza.

11.1 Analýza vnímania čiar pri náklone hlavy

Bc. Daniel Lukáč

11.1.1 Reprezentácia čiar

Čiary sú v projekte reprezentované triedou *public class Line.* Každá čiara je v triede definovaná ako absolútna hodnota prvého bodu, typu Vector3D a druhého bodu typu Vector3D. K týmto premenným sa pristupuje pomocou metód get() a set(), ktoré sú súčasťou triedy *public class Line.*

```
/** absolute position of first point of line **/
private Vector3D position1 = Vector3D.ZERO_VECTOR;

/** absolute position of second point of line **/
private Vector3D position2 = Vector3D.ZERO_VECTOR;
```

Obr.9: Implementácia čiar

11.1.2 Hlava

Otáčanie hlavy funguje na princípe xml súborov, ktoré sa nachádzajú v priečinku move v Jimovi. Xml súbory obsahujú radu parametrov, pre otáčanie jednotlivých kĺbov Jima. Pohyby hlavy sú reprezentované súbormi:

- head_left_120.xml
- head_right_120.xml
- head_left_down.xml
- head_right_down.xml

Súbory head_left_120.xml a head_right_120.xml zabezpečujú otáčanie hlavy do strán pod maximálnym uhlom 120° a súbory head_left_down.xml a head_right_down.xml zabezpečujú náklony hlavy nadol a nahor.

Analýzou sme zistili, že náklony hlavy nahor a nadol boli z implementácie vylúčené tímom Infinity v akademickom roku 2014/2015, v dôsledku nulovej pridanej efektivity pre vnímanie prostredia a lopty. Tým sme nemohli analyzovať vnímanie čiar pre úklony. Jim tak vníma čiary len vo vodorovnej polohe, čím nezohľadňuje žiaden náklon hlavy, resp. vidí len vodorovne. Ďalej sme analýzou prišli na to, že Jimova hlava pri chôdzi z časti mení svoju rotáciu vzhľadom na celé telo, no táto rotácia v priestore sa pri vnímaní čiar nijako nezohľadňuje.

```
} /*else if (!leftDownLook) {
    leftDownLook = true;
    return LowSkills.get("head_left_down");
}
else if (!rightDownLook) {
    rightDownLook = true;
    return LowSkills.get("head_right_down");
} */
else if (!leftLook) {
    leftLook = true;
    return LowSkills.get("head_left_120");
}
else if (!rightLook) {
    rightLook = true;
    return LowSkills.get("head_right_120");
}
```

Obr. 10: Skilly pre pohyb hlavy

```
    } /*else {
        if (moveForward(relativizedBall)) {
            logger.log(LogType.HIGH_SKILL, "Move forward");
            return LowSkills.get(walkSkill);
            /*if(playerDistance(relativizedBall)){
                planner.addHighskillAsFirst(new HeadDownSkill());
            } else{
                return LowSkills.get(walkSkill);
            }*/

       /* } else if (moveBack(relativizedBall)) {
            logger.log(LogType.HIGH_SKILL, "Move back");
            return LowSkills.get("walk_back");
        } else {
            return kick();
        }
    }*/
}else if (!isInVerticalRange(relativizedBall)) {
    logger.log(LogType.HIGH_SKILL, "Move vertical");

    if (moveForward(relativizedBall)) {
        /*this can be used for executing some action in the moment when player come to ball
         * if(playerDistance(relativizedBall)){
            System.out.println("we are before ball");
            logger.log(LogType.HIGH_SKILL, "Move forward");
             planner.addHighskillAsFirst(new HeadDownSkill());
        }*/
```

Obr. 11: Zakomentované pohyby hlavy

11.2 Analýza náklonu hrude

Bc. Šimon Harvan

V tejto kapitole sme analyzovali vnímanie čiar pri náklone hráča. Analyzovali sme prezrením kódu a spustením testovacieho frameworku. Čiary sú jeden z pomocných prvkov pri určovaní polohy hráča a poloha čiar je chybná.

11.2.1 Analýza

V tejto kapitole popíšeme ako sa vykonáva rozpoznávanie čiar.

11.2.1.1 Prijatie dát zo servera.

11.2.1.2 Spracovanie a uloženie dát.

Pri spracovaní čiar nie sú využívané žiadne dáta z gyroscopu. Ak však je vidno rohové zástavky je vykreslenie dostatočne presné na určenie polohy:

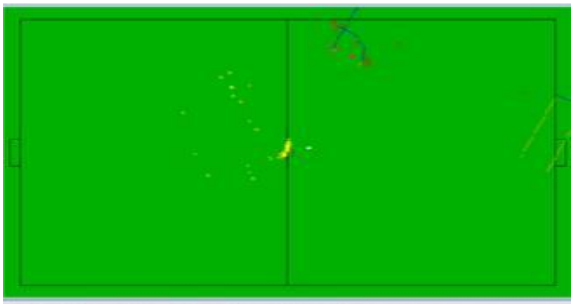
```
Line2D l1 = new
Line2D(line1.getPosition1().getX(),line1.getPosition1().g
etY(),
line1.getPosition2().getX(),line1.getPosition2().getY());
             Line2D l2 = new
Line2D(line2.getPosition1().getX(),line2.getPosition1().g
etY(),
line2.getPosition2().getX(),line2.getPosition2().getY());
```

11.2.1.3 Využitie čiar na určenie polohy. Ak hráč nevidí ani jeden pevný bod, tak využije čiary.

```
If ((data.fixedObjects == null || data.fixedObjects.size() < 1) &&
data.lines != null && data.lines.size() == 1) {
      oneLineSeen(data);
      return;
}

if ((data.fixedObjects != null && data.fixedObjects.size() == 1) &&
data.lines != null && data.lines.size() > 0) {
      mPoint.addAll(oneFixedObjectSeen(data));
```

```
} else if ((data.fixedObjects == null || data.fixedObjects.size() ==
0) && data.lines != null && data.lines.size() > 1) {
      mPoint.addAll(justLinesSeen(data));
}
```

Ako môžeme vidieť na obrázku, je pri páde hráča, kde stratí z obzoru všetky objekty vykreslenie čiar chybové.



Obr. 12: Ukážka hracieho poľa

11.2.2 Záver

V analýze sme zistili, že pri vypočítavaní čiar nie je zohľadňovaný náklon hráča. Keďže sa hráč pri pohybe nakláňa dopredu, dozadu ale aj doboka je jeho pozícia vypočítavaná s chybnými údajmi z čoho vyplýva hráčova dezorientácia.

11.3 Pozícia lopty, videnie lopty hráčom

Bc. Marko Moravčík

Dôležité triedy pri určovaní pozície lopty:

DynamicObject.java - trieda slúžiaca na opis polohy a rýchlosti objektov pohybujúcich sa v priestore – lopta

WorldModel.java -opisuje stav sveta, teda ostatných hráčov, lopty,(objektov, ktorých agent vidí), odhaduje ich pozície na základe odhadnutej rýchlosti

ParsedData - Trieda obsahuje dáta, ktoré boli získané z prijatej správy od servera. Sú to dáta z receptorov (zrak, sluch, sila pôsobiaca na nohy), a tiež dáta opisujúce aktuálnu situáciu na ihrisku.

Metóda pre vypočítanie vzdialenosti medzi loptou a hráčom (pozícia lopty, ktorú berie v úvahu je nastavená vo WorldModel.

```java
public double getDistanceFromBall() {
    if(this.worldModel.getBall() == null) {
        return 0;
    }
    else
        return DistanceHelper.computeDistanceBetweenObjects(worldModel.getBall().getPosition(), this.getPosition());
}
```

Obr. 13: Metóda pre vypočítanie vzdialenosti medzi loptou a hráčom

11.3.1 Graf dát

Graf dát predstavuje informácie, ktoré dokáže agent o svete získať. Informácie sú na rôznych stupňoch abstrakcie. Najnižšiu vrstvu predstavujú tie, ktoré prichádzajú zo servera a predstavujú dáta, ktoré agent dostáva zo snímačov. V grafe sú to uzly, do ktorých nevchádza žiadna hrana. Model grafu dát je zobrazený na obrázku 1 .

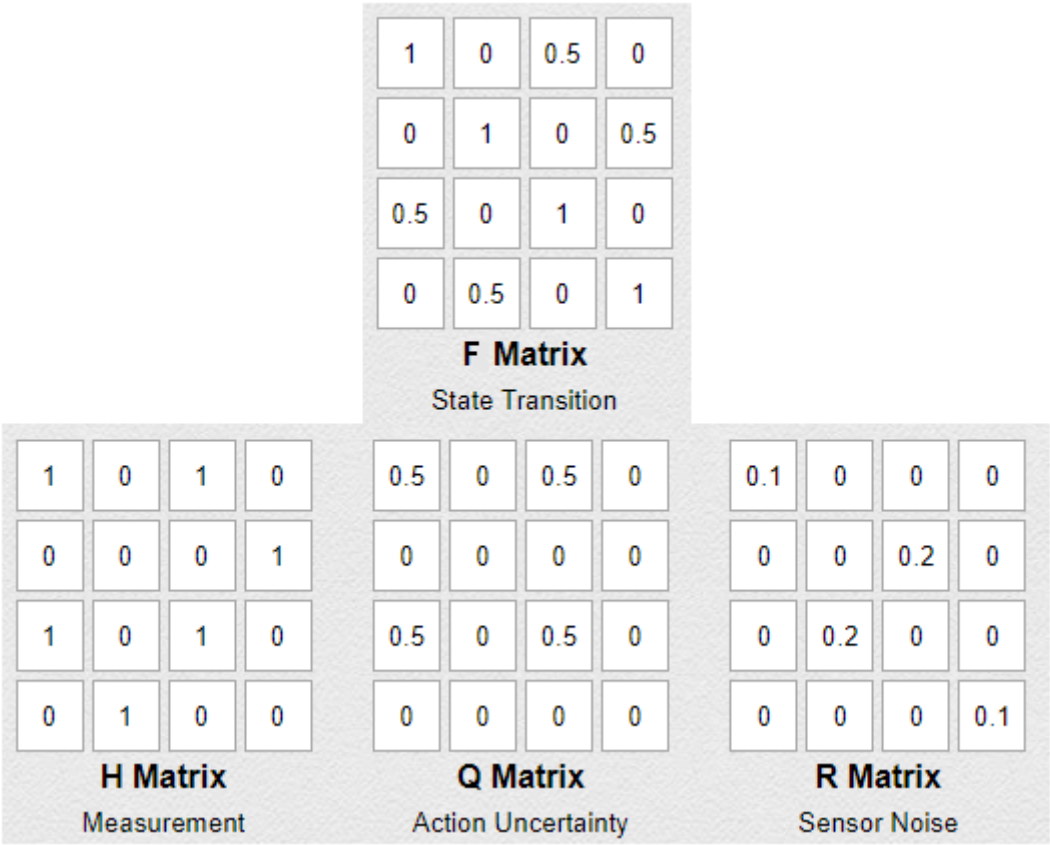
V grafe môžeme vidieť ako prebieha videnie lopty hráčom. Od počiatočného videnia, kde získa Agent pozíciu lopty, následne určí polohu lopty na ihrisku. A podľa tejto polohy, ďalej vyhodnotí rôzne iné situácie ako, kto má loptu? (druhý tím, náš tím), ako ďaleko je daný Agent od lopty, či je pri nej najbližšie a podobne. Následne podľa týchto informácií určí hernú situáciu a podľa nej taktiku, ktorá sa vykoná.

Okrem toho berie do úvahy aj históriu zmien polohy lopty v závislosti od času, ktorá mu slúži pri predikcii polohy lopty.

12. Zistenie hodnôt parametrov Kalmana

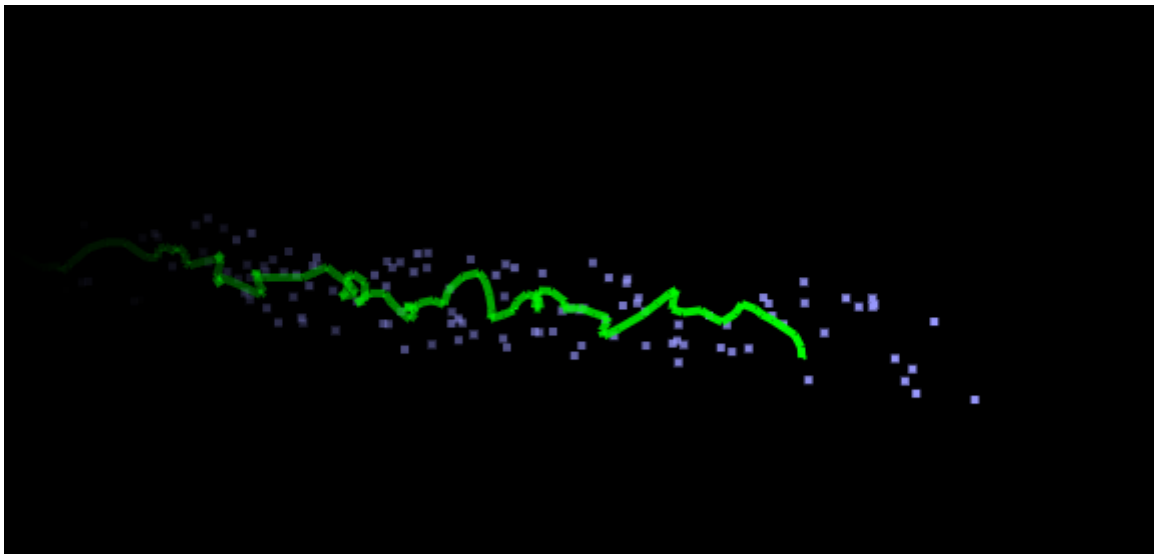
12.1 Parametre matíc

Na určenie hodnôt matice, ktorá sa bude používať pri výpočte sme použili hotovú simuláciu Kalmanovho filtra, ktorá nám umožnila ľahko upravovať hodnoty matíc a následne sledovať ako dané hodnoty ovplyvnili celkové zníženie šumu a prípadnú predpoveď umiestnenia lopty na základe predchádzajúcich hodnôt. Prvotné testovanie sme vykonali s hodnotami na Obr. 1.a.



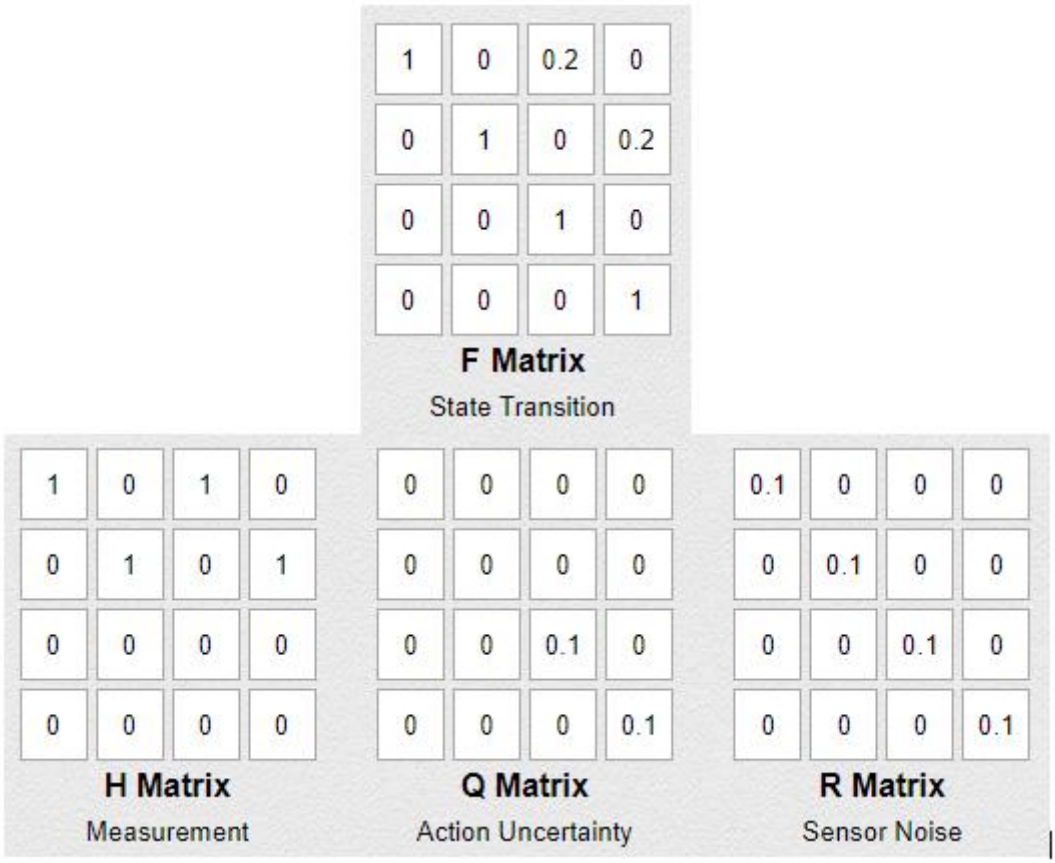
Obr. 1.a: Prvotné hodnoty matíc

Výsledok je možné vidieť vidieť na Obr. 1.b, kde je jasné vidieť že zníženie šumu nebolo presné preto sme pokračovali v testovaní s inými hodnotami.



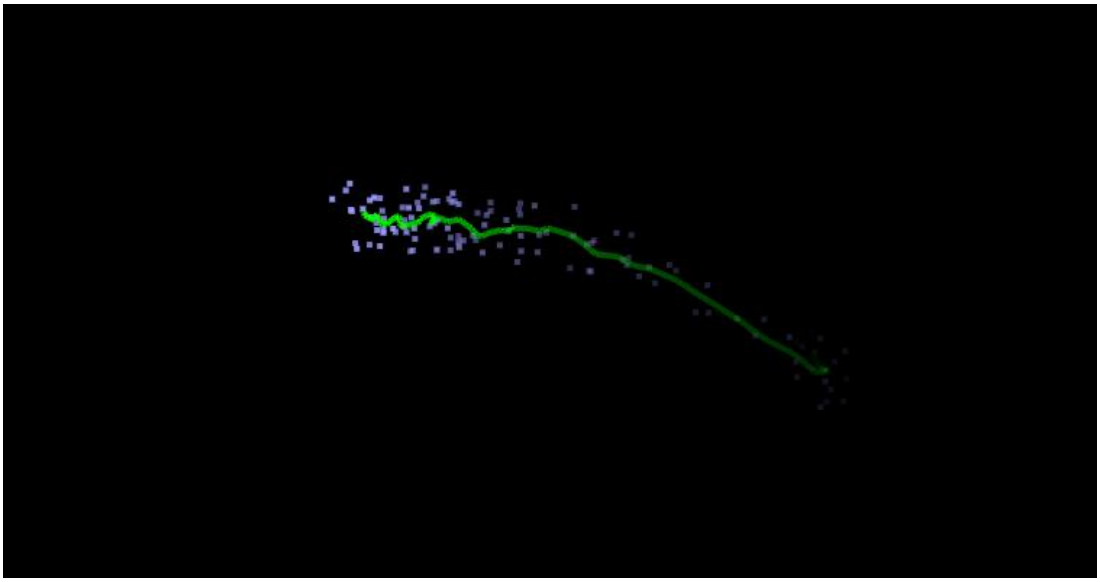
Obr. 1.b: Výsledné zníženie šumu

Po viacerých testoch sa nám podarilo zostaviť matice z hodnôt zobrazených na Obr. 1.c.



Obr. 1.c: Definitívne hodnoty matíc

Na Obr. 1.d je vidieť najpresnejšie zníženie šumu, preto sme sa rozhodli implementovať matice s týmito hodnotami a na základe skutočných testov po implementovaní daných matíc v našom projekte sa overí správnosť zvolených parametrov.



Obr.1.d: Najpresnejšie zníženie šumu

13. Pôvodná implementácia Kalman filtra pomocou Vektorov

Pôvodná implementácia je riešená pomocou vektorov, ktorú sme testovali porovnaním hodnôt, ktoré vstupujú do Kalmanovho filtra a hodnôt, ktoré sú výstupom po aplikácií Kalmanovho filtra. Z daných meraní sa zistilo, že aktuálna implementácia nepomáha znížiť šum na pozíciu lopty žiadnym spôsobom. Výpis bol doplnený do funkcie *adjustBallPosition* v triede *KalmanAdjuster* (Obr. 1).

```
// function for adjusting ball position
private void adjustBallPosition(ParsedData data) {
    if (data.ballRelativePosition == null) {
        return;
    }

    if (isObsolete(lastTimeBallSeen) || ballKalman == null) {
        ballKalman = freshKalman();
    }

    LOG.log(LogType.VELOCITY, "POLOHA LOPTY PRED KALMANOM: X: " + data.ballRelativePosition.getX() +
            " a Y: " + data.ballRelativePosition.getY());
    data.ballRelativePosition = ballKalman.update(data.ballRelativePosition);
    LOG.log(LogType.VELOCITY, "POLOHA LOPTY PO KALMANOVI: X: " + data.ballRelativePosition.getX() +
            " a Y: " + data.ballRelativePosition.getY());
    lastTimeBallSeen = now;
}
```

Obr. 1: Logy na otestovania Kalmana

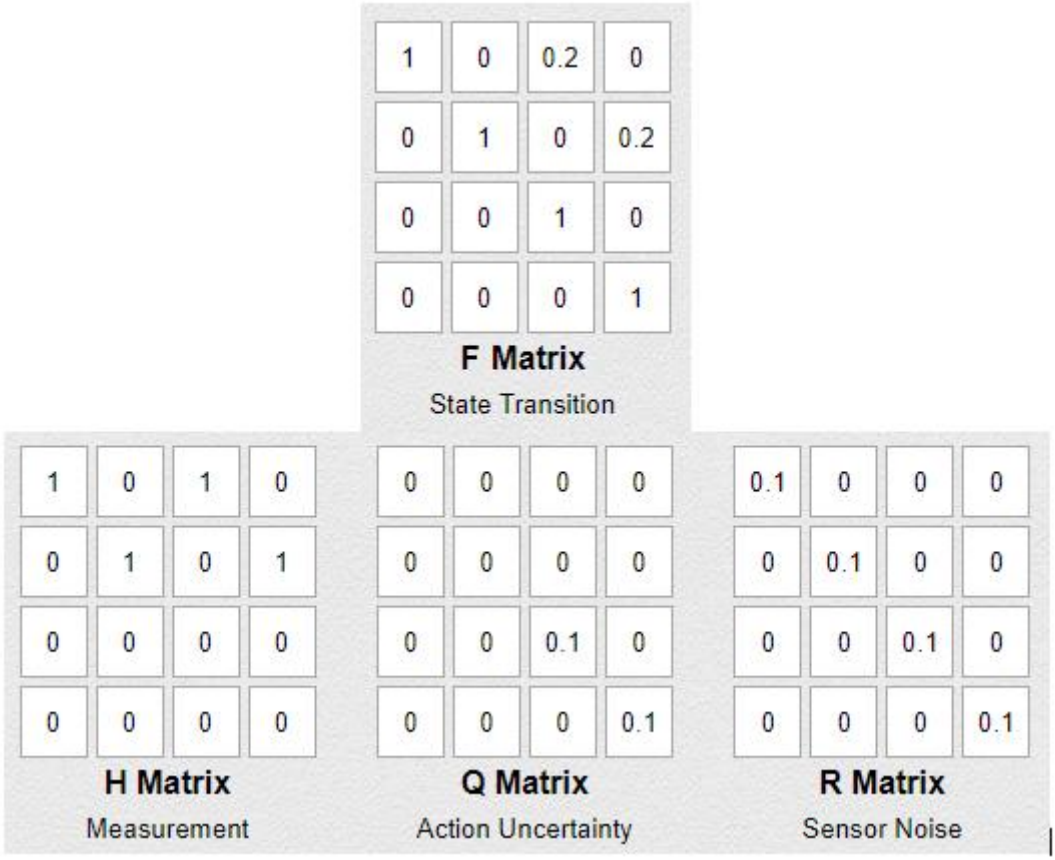
Hodnoty, ktoré vstupovali do filtra a hodnoty, ktoré boli vrátené po vykonaní funkcie update sú znázornené na Obr. 2, na ktorom je jasné vidieť, že Kalman v rámci osi X zlepšil polohu lopty o pol desatiny ale v rámci osi Y došlo dokonca k zhoršeniu určenia polohy.

```
VELOCITY: POLOHA LOPTY PRED KALMANOM: X: -0.184 a Y: 0.75
Sun Mar 04 22:29:13 CET 2018 sk.fiit.jim.agent.models.KalmanAdjuster adjustBallPosition
VELOCITY: POLOHA LOPTY PO KALMANOVI: X: -0.17901638361677727 a Y: 0.7525447950092301
Sun Mar 04 22:29:13 CET 2018 sk.fiit.jim.agent.models.WorldModel returnBallRelativePosition
```

Obr. 2: Výsledky testovania

13.1 Prvá implementácia Kalman filtra

Na základe zistených skutočností sme sa rozhodli implementovať Kalmana pomocou matíc F, H, Q a R, ktoré sa na začiatku nainicializujú na hodnoty zobrazené na Obr. 3. F je stavová prechodová matica. Táto matica ovplyvňuje vektor merania. H je meracia matica. Táto matica ovplyvňuje zisk Kalmana. Q je matica neistoty akcie. Táto matica implikuje procesnú šumovú kovarianciu. R je matrica šumu snímača. Táto matica implikuje kovarianciu chyby merania na základe množstva šumu snímača. z predstavuje merací vektor, obsahuje údaje o polohe a rýchlosti hráča. Pri simulácii sa pridá šum zo senzora náhodným posunom aktuálnych hodnôt polohy hráča.



Obr. 3: Definitívne hodnoty matíc

Vzájomné násobenie matíc v jednotlivých funkciách, ktoré slúžia na predikovanie a updatovanie je znázornené na Obr. 4.

```java
@Override
public void configure(DMatrixRMaj F, DMatrixRMaj Q, DMatrixRMaj H) {
    this.F = new SimpleMatrix(F);
    this.Q = new SimpleMatrix(Q);
    this.H = new SimpleMatrix(H);
}

@Override
public void setState(DMatrixRMaj x, DMatrixRMaj P) {
    this.x = new SimpleMatrix(x);
    this.P = new SimpleMatrix(P);
}

@Override
public void predict() {
    // x = F x
    x = F.mult(x);

    // P = F P F' + Q
    P = F.mult(P).mult(F.transpose()).plus(Q);
}
```

```java
@Override
public void update(DMatrixRMaj _z, DMatrixRMaj _R) {
    // a fast way to make the matrices usable by SimpleMatrix
    SimpleMatrix z = SimpleMatrix.wrap(_z);
    SimpleMatrix R = SimpleMatrix.wrap(_R);

    // y = z - H x
    SimpleMatrix y = z.minus(H.mult(x));

    // S = H P H' + R
    SimpleMatrix S = H.mult(P).mult(H.transpose()).plus(R);

    // K = PH'S^(-1)
    SimpleMatrix K = P.mult(H.transpose().mult(S.invert()));

    // x = x + Ky
    x = x.plus(K.mult(y));

    // P = (I-kH)P = P - KHP
    P = P.minus(K.mult(H).mult(P));
}
```

Obr. 4: Implementácia matíc

13.2 Zhrnutie

Po implementovaní daných matíc sme narazili na problém, keďže v celom projekte sa používajú vektory. Problém bude potrebné najpravdepodobnejšie odstrániť upravením matíc *x* a *z* na vektory. Následne by malo byť možné otestovať novú implementáciu a zistiť, či sa zlepšilo odstránenie šumu pri určovaní polohy lopty.

14. Hodnoty matíc pre Kalman filter

Hodnoty matíc, ktoré sa aktuálne využívajú sú zobrazené Obr. 1. F je stavová prechodová matica. Táto matica ovplyvňuje vektor merania. H je meracia matica. Táto matica ovplyvňuje Kalmanov zisk. Q je matica neistoty akcie. Táto matica implikuje procesnú šumovú kovarianciu. R je matrica šumu snímača. Táto matica implikuje kovarianciu chyby merania na základe množstva šumu snímača. x predstavuje odhad stavu v čase k, ktorý obsahuje údaje o pozícií lopty na osy x a y a taktiež o rýchlosti lopty v rámci osy x a y. P predstavuje kovariančnú maticu chybovosti.

```
F = new double[][] {
            { 1, 0, 0, 0 },
            { 0, 1, 0, 0 },
            { 0, 0, 1, 0 },
            { 0, 0, 0, 1 }
        };

//only observe first 2 values - the position coordinates
H = new double[][] {
            { 1, 0, 1, 0 },
            { 0, 1, 0, 1 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 }
        };
Q =     new double[][] {
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0.1,0 },
            { 0, 0, 0, 0.1 }
        };


P = new double[][] {
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 }
    };

R = new Array2DRowRealMatrix(new double[][] {
        { 0.1, 0, 0, 0 },
        { 0, 0.1, 0, 0 },
        { 0, 0, 0.1, 0 },
        { 0, 0, 0, 0.1 }
});

x = new ArrayRealVector(new double[] { 0, 0, 0, 0 });
```

Obr. 1: Definitívne hodnoty matíc

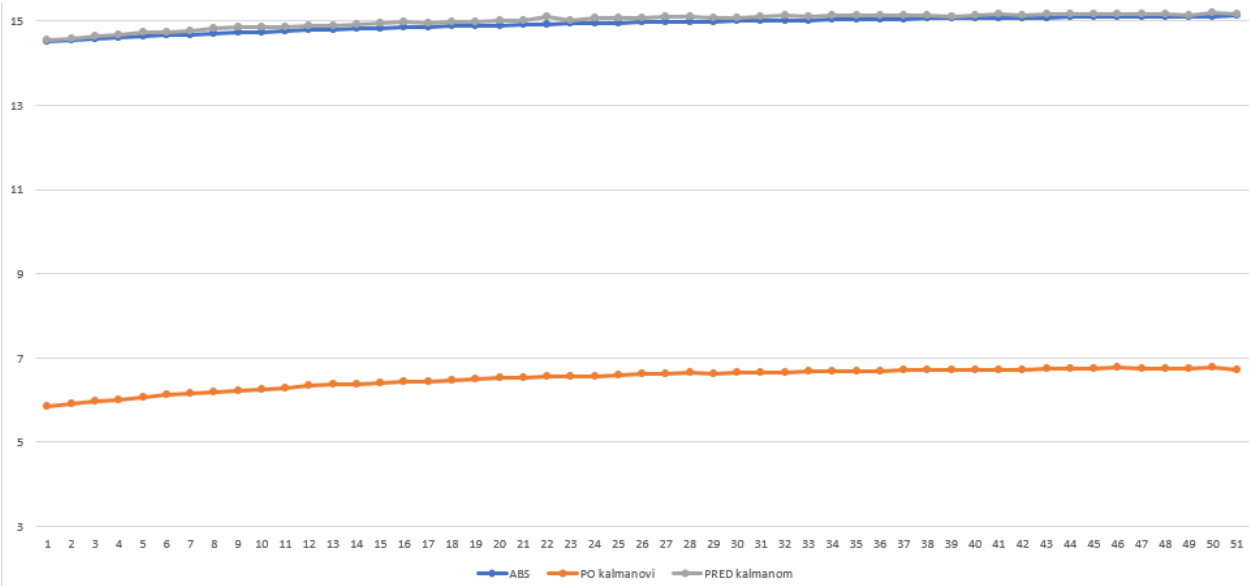
14.1 Zhrnutie

Keďže zatiaľ nebolo možné naplno otestovať našu implementáciu Kalmanovho filtra, neobjavili sa žiadne známky toho, že aktuálne hodnoty matíc sú zvolené nesprávne. Nie je však vylúčené, že po dôkladnom otestovaní budú niektoré hodnoty mierne upravené.

15. Overenia Kalman filtra pomocou grafov

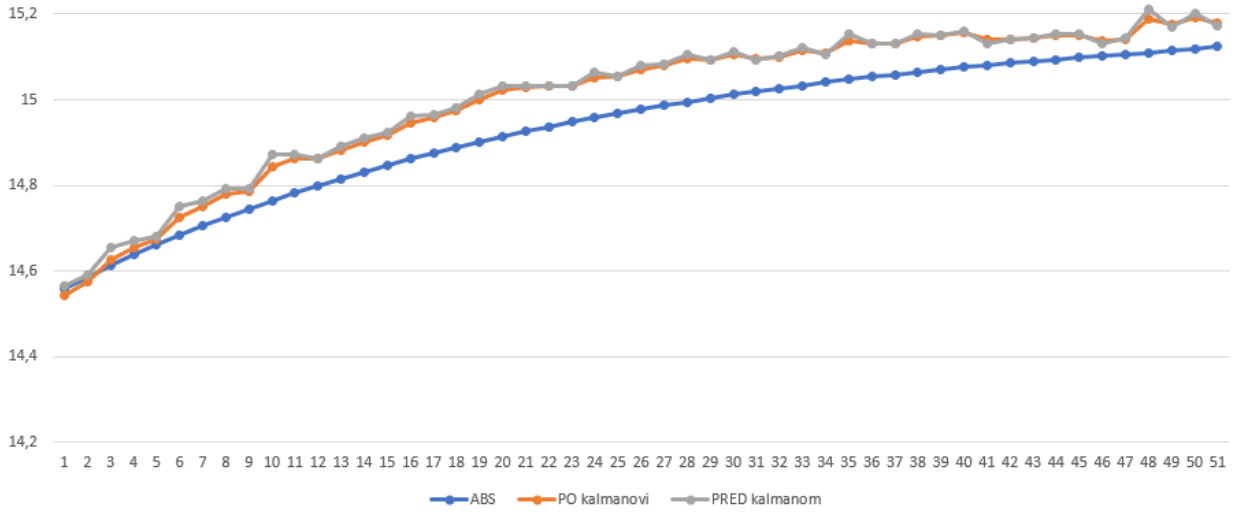
Autor: Michal Baňas

Po implementovaní Kalman filtra prostredníctvom matíc bolo potrebné overiť dané riešenie a následne ho porovnať s pôvodnou implementáciou. Test prebiehal na rovnakom princípe a to tak, že hráč bol umiestnený na jeden bod, na ktorom stál a lopta bola následne zo stredu ihriska vystrelená smerom od hráča na súperovu bránu. Výsledok novej implementácie je možné vidieť na Obr. 1.a.



Obrázok 1.a: Aktuálna implementácia Kalman filtra

Ná Obr. 1.b je pre porovnanie zobrazená pôvodná implementácia Kalman filtra pomocou vektorov.



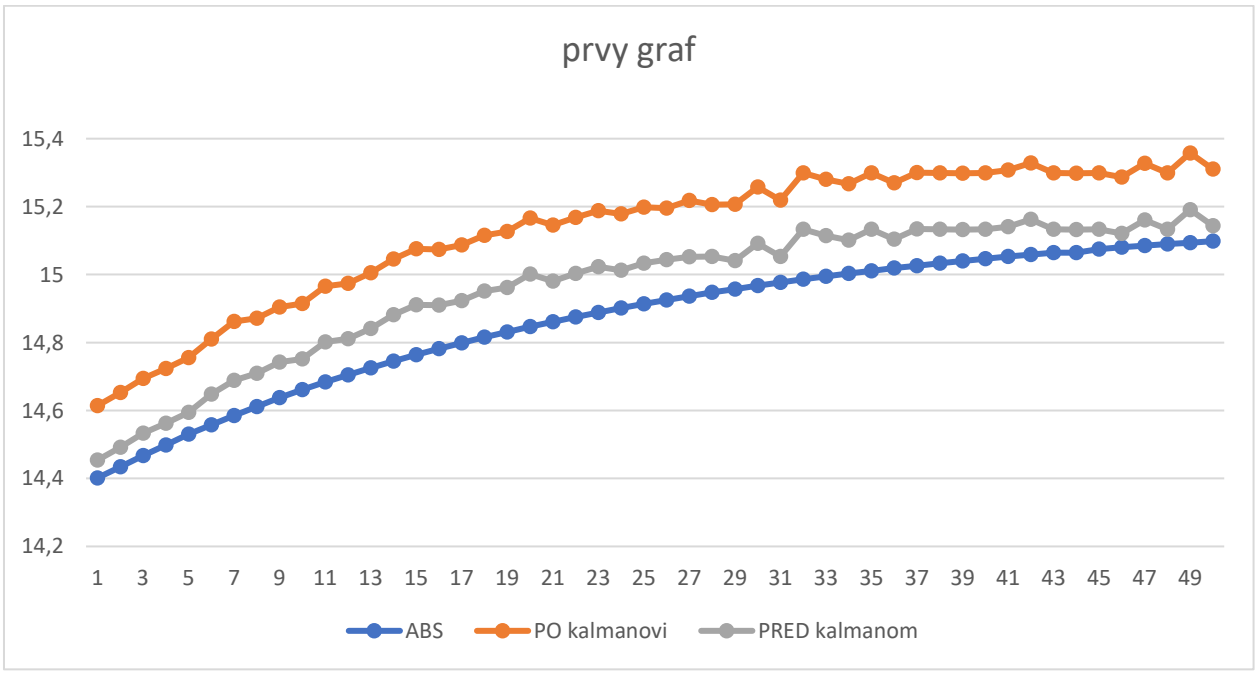
Obrázok 1.b: Pôvodná implementácia Kalman filtra

Na základe týchto porovnania týchto grafov je jasne vidieť, že nová implementácia je aktuálne oveľa nepresnejšia oproti pôvodnej implementácii. V ďalšom priebehu je preto potrebné zamerať sa na hodnoty matíc, keďže problém bude najpravdepodobnejšie v hodnotách jednotlivých matíc, ktoré bude nutné poriadne otestovať a tak zistiť, na základe akých hodnôt bude nová implementácia Kalman filtra dávať presnejšie výsledky oproti pôvodnej.

16. Testovanie matíc

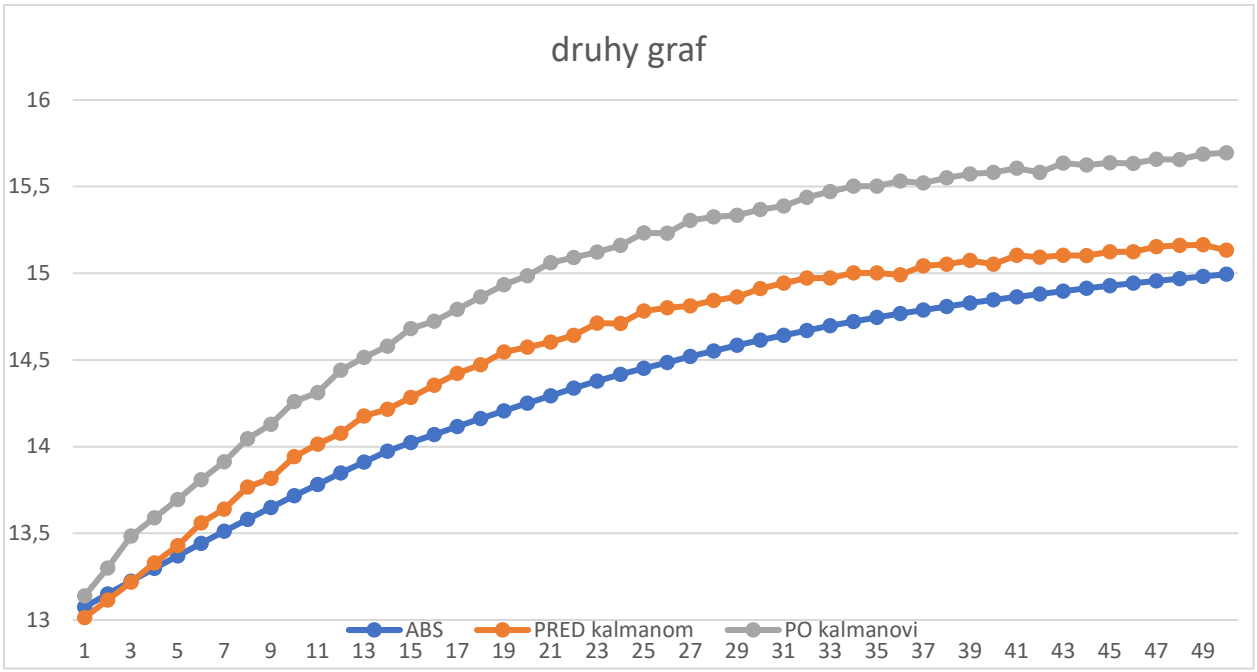
Na prevod súradníc bolo potrebné získať relatívne hodnoty súradníc pred a po Kalmanovom filtri, ku ktorým však bolo potrebné prirátať vzdialenosť od hráča. Pre jednoduchosť sme hráčovi definovali pevnú pozíciu na súradniciach [-3:0:0.4]. Lopte bola stanovená klasická štartovacia pozícia na súradniciach [0:0:0.4]. Na základe toho sa teda od x-ovej súradniceodčítala hodnota 3. Absolútnu pozíciu sme získali z Testframeworku. Loptu sa následne po nastavení pozície hráča vystrelila smerom od hráča na súperovu bránu. Získane hodnoty sme následne exportovali do csv súborov, z ktorých sa nakoniec vyskladal výsledný graf, na ktorom je vidieť odchýlky medzi relatívnymi súradnicami, ktoré dostáva hráč o pozícií lopty a medzi absolútnymi súradnicami, na ktorých sa lopta nachádza.

Výsledok prvého testu, kde boli hodnoty matice H na nastavené na hodnotu 0.9 a 0.2 je zobrazený na Obr. 1.



Obr. 1 - Porovnanie relatívnych a absolútnych súradníc

Výsledok druhého testu, kde boli hodnoty matice H na nastavené na hodnotu 0.9 a 0.5 je zobrazený na Obr. 2.



Obr.2 - Porovnanie relatívnych a absolútnych súradníc

Na základe grafov je možné vidieť, že hodnoty matíc stále nie sú úplne presné, keďže nenastalo želané zlepšenie priodchýlke relatívnych súradníc od absolútnych.

17. Práca s maticami kalman filtra

Bc. Dávid Roba

Pri aplikovaní Kalman filtra sa využíva 5 matíc, ktorých hodnoty je možné v prípade potreby jednoducho zmeniť. Matica H predstavuje maticu merania, ktorá ovplyvňuje hodnotu Kalmanovho zisku. V tejto matici sú vyplnené len prvé dva riadky, keďže v našom prípade využívame len os *x* a *y*. Hodnoty v prvom riadku znamenajú posun na osi y, respektíve rýchlosť pohybu po osi y. Hodnoty v druhom riadku sa týkajú osi x, kde prvá hodnota znamená posun na osi x a druhá zase rýchlosť pohybu po tejto osi. Príklad je zobrazený na Obr. 1

```
H = new double[][] {
        { 0.9, 0, 0.1, 0 },
        { 0, 0.9, 0, 0.1 },
        { 0, 0, 0, 0 },
        { 0, 0, 0, 0 }
    };
```

Obr. 1: Matica merania H

Matica F predstavuje stavovú prechodovú maticu, ktorá ovplyvňuje vektor merania. Pri tejto matici sú okrem prvých dvoch riadkov, ktoré sú vyplnené na základe rovnakého princípu ako pri predchádzajúcej matici, diagonálne doplnený aj tretí a štvrtý riadok, z toho dôvodu, že pri výpočtoch sa používa aj transponovaná verzia tejto matice. Príklad je zobrazený na Obr. 2.

```
F = new double[][] {
        { 1, 0, 0.1, 0 },
        { 0, 1, 0, 0.1 },
        { 0, 0, 1, 0 },
        { 0, 0, 0, 1 }
    };
```

Obr. 2: Stavová prechodová matica F

Matica R predstavuje maticu šumu snímača, čo v našom prípade predstavuje šum, ktorý agent dostáva zo servera. Táto matica implikuje kovarianciu chyby merania, na základe množstva šumu. Pri tejto matici sa v našom prípade ukázala ako najlepšia voľba diagonálne vyplnená matica. Príklad je zobrazený na Obr. 3.

```
R = new double[][] {
        { 0.1, 0, 0, 0 },
        { 0, 0.1, 0, 0 },
        { 0, 0, 0.1, 0 },
        { 0, 0, 0, 0.1 }
};
```

Obr. 3: Matica šumu snímača R

Matica Q predstavuje maticu nepresnosti, ktorá implikuje procesnú šumovú kovarianciu. Pri tejto matici je možné ľubovoľne si zvoliť úroveň nepresnosti, na základe ktorej je možné pozorovať u agenta nepresnosti pri vnímaní okolitého sveta. V našom prípade sú vyplnené podľa diagonály tretí a štvrtý riadok. Príklad je zobrazený na Obr. 4.

```
Q = new double[][] {
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0.1,0 },
            { 0, 0, 0, 0.1 }
        };
```

Obr. 4: Matica nepresnosti Q

Matica P predstavuje počiatočnú maticu, ktorá vstupuje do kroku predikcie a mala by začínať ako nulová, avšak v našom prípade už od začiatku nadobúda hodnoty znova na prvých dvoch riadkoch, ktoré ovplyvňujú posun a rýchlosť pohybu lopty po osi x a y. Príklad je zobrazený na Obr. 5.

```
P = new double[][] {
    { 1, 0, 0.8, 0 },
    { 0, 1, 0, 0.8 },
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 }
};
```

Obr. 5: Počiatočná matica P

18. Finalizácia matíc

Bc. Dávid Roba

Po poslednom testovaní jednotlivých matíc Kalman filtra sme v momentálnom stave dosiahli najlepšie výsledky s maticami, ktorých hodnoty sú znázornené Obr.1 až Obr.5. Hodnoty daných matíc sa môžu líšiť v závislosti od zvolenej implementácie, preto nie je vylúčené, že sa v budúcnosti, pri implementovaní Kalman filtra na samotného hráča alebo fixné body, budú hodnoty jednotlivých matíc meniť, keďže pri týchto implementáciách sa bude s najväčšou pravdepodobnosťou brať do úvahy aj os z.

```
H = new double[][] {
          { 0.9, 0, 0.1, 0 },
          { 0, 0.9, 0, 0.1 },
          { 0, 0, 0, 0 },
          { 0, 0, 0, 0 }
      };
```

Obr. 1: Matica merania H

```
F = new double[][] {
          { 1, 0, 0.1, 0 },
          { 0, 1, 0, 0.1 },
          { 0, 0, 1, 0 },
          { 0, 0, 0, 1 }
       };
```

Obr. 2: Stavová prechodová matica F

```
R = new double[][] {
        { 0.1, 0, 0, 0 },
        { 0, 0.1, 0, 0 },
        { 0, 0, 0.1, 0 },
        { 0, 0, 0, 0.1 }
};
```

Obr. 3: Matica šumu snímača R

```
Q = new double[][] {
          { 0, 0, 0, 0 },
          { 0, 0, 0, 0 },
          { 0, 0, 0.1,0 },
          { 0, 0, 0, 0.1 }
       };
```

Obr. 4: Matica nepresnosti Q

```
P = new double[][] {
    { 1, 0, 0.8, 0 },
    { 0, 1, 0, 0.8 },
    { 0, 0, 0, 0 },
    { 0, 0, 0, 0 }
};
```

Obr. 5: Počiatočná matica P

19. Využitie konkrétnych hodnôt šumu podľa simsparku

Bc. Marko Moravčík

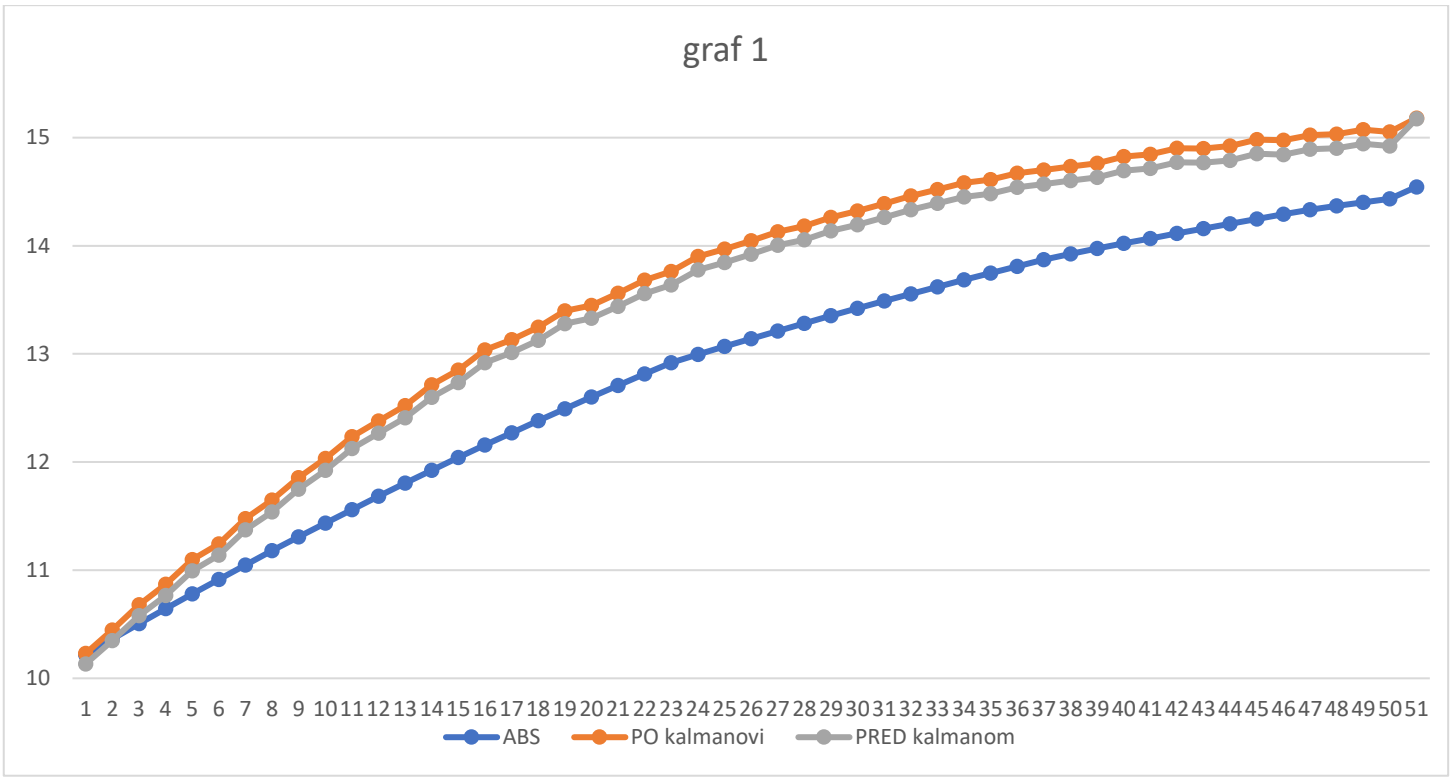
Podľa konrétnych hodnôt šumu, ktoré sa našli v simsparku došlo k úprave matice R (matica pozorovaného šumu), čo prinieslo zlepšenie pre náš implementovaný Kalman filter.

```
R = new Array2DRowRealMatrix(new double[][] {
        { 0.0965, 0, 0, 0 },
        { 0, 0.0965, 0, 0 },
        { 0, 0, 0.1480, 0 },
        { 0, 0, 0, 0.1480 }
        });
```

Obr. 1: Matica R s konkrétnymi hodnotami šumu

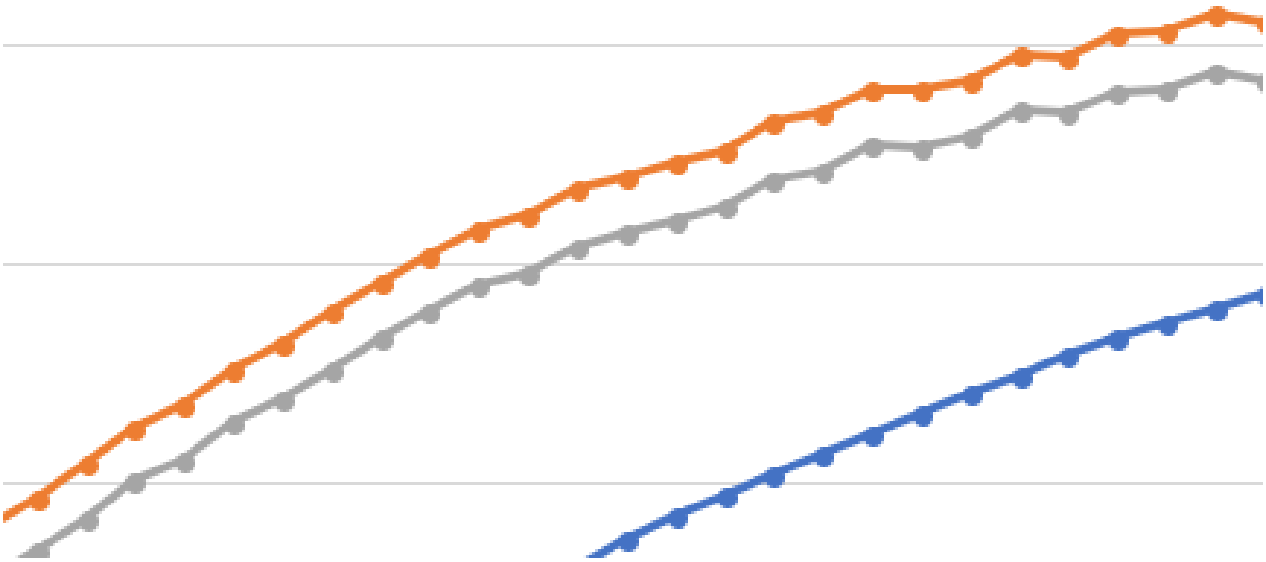
20. Overenie Kalmana pomocou grafov

Na základe logovaných dát sme vyhotovili 2 grafy, na ktorých je viditeľný posun presnosti nových matíc pre Kalmana , ktorý ale treba ešte doladiť :

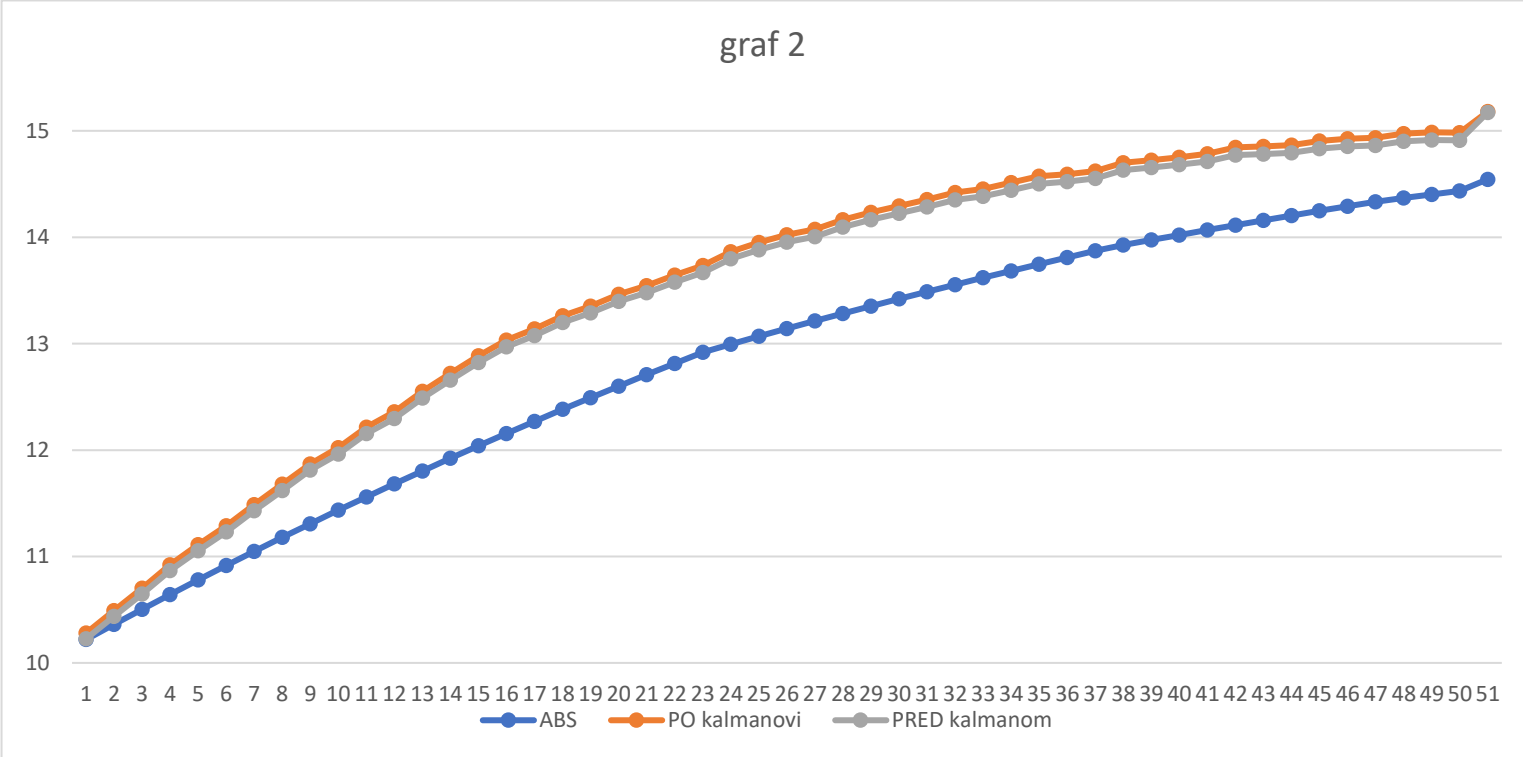


Obr. 1: Pohyb lopty po osi X pred a po spustení Kalmana

Detail:

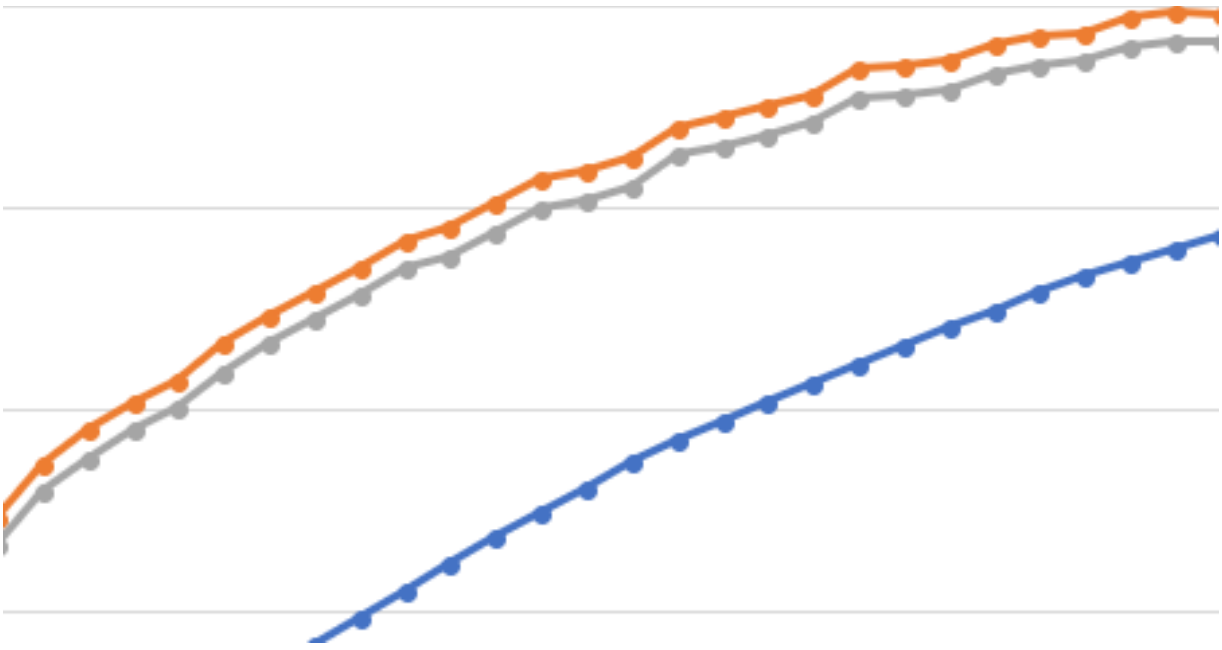


Obr. 2:Detail pohybu lopty po osi X pred a po spustení Kalmana



Obr. 3: Pohyb lopty po osi X pred a po spustení Kalmana

Detail:



Obr. 4: Detail pohybu lopty po osi X pred a po spustení Kalmana

21. Celkové zhrnutie Kalman filtra

Bc. Marko Moravčík

V tomto dokumente je uvedená finálna implementácia Kalmana (zdrojové kódy s vysvetlivkami), ktorá bola po sérií doposiaľ vykonaných testov, označená ako najviac vyhovujúca.

Implementácia Kalman filtra je tvorená jednou hlavnou triedou ***KalmanReal***, ktorá obsahuje nenaplnené atribúty pre matice a vektory (Obr. 1).

- *RealVector _x* - predikovaný stav
- *RealMatrix _P* - predikovaná kovariancia stavu
- *RealMatrix_K* – optimálny Kalman zisk
- *RealMatrix _F* - predvolený procesný model (stavová matica prechodu)
- *RealMatrix _Q* - predvolený šum procesu
- *RealMatrix _B* - predvolený model riadenia (mapuje kontrolný vektor na stavový priestor) -> nepoužíva sa
- *RealMatrix _H* - predvolený model pozorovania (mapuje pozorovania do stavového priestoru).
- *RealMatrix _R* - predvolený šum pozorovania

Ďalej obsahuje, okrem getterov a setterov aj hlavné metódy (Obr. 2):

- *predict* - Používa predchádzajúci odhad stavu a poskytnutý model pohybu na vytvorenie odhadu aktuálneho stavu.
- *update* – Aktuálne informácie o meraní (t.j. vektor *_z*, ktorý obsahuje polohu daného objektu (napr. lopty) a rýchlosť, ktorú dosahuje na osi x a y) sa používajú na úpravu odhadov stavu pomocou poskytnutého modelu pozorovania (matica *_H*).
- *getStateV* – Getter pre získanie aktuálneho stavu vektora *_x* vo forme triedy Vector3D.

```
 * Predicted state.
 */
protected RealVector _x;

/**
 * Predicted state covariance.
 */
protected RealMatrix _P;

/**
 * Optimal Kalman gain.
 */
protected RealMatrix _K;

/**
 * Default process model (state transition matrix).
 */
protected RealMatrix _F;

/**
 * Default process noise.
 */
protected RealMatrix _Q;

/**
 * Default control model (maps control vector to state space).
 */
protected RealMatrix _B;

/**
 * Default observation model (maps observations to state space).
 */
protected RealMatrix _H;

/**
 * Default observation noise.
 */
protected RealMatrix _R;
```

Obr. 2: Nenaplnené atribúty pre matice a vektory

```
/**
 * Uses the previous state estimate and the provided motion model to produce
 * an estimate of the current state.
 * @param F the process model.
 * @param Q the process noise.
 * @param B the control model.
 * @param u the current control input.
 */
public void predict() {
    _x = _F.operate(_x);
    _P = _F.multiply(_P).multiply(_F.transpose()).add(_Q);
}


/**
 * Current measurement information is used to refine the state estimate
 * using the provided observation model.
 * @param H the observation model.
 * @param R the observation noise.
 * @param z the current measurement.
 */
public void update(RealVector z) {


    // Apply the rest of the Kalman update
    RealVector y = z.subtract(_H.operate(_x));
    RealMatrix S = _H.multiply(_P).multiply(_H.transpose()).add(_R);
    RealMatrix invS = new LUDecomposition(S).getSolver().getInverse();
    _K = _P.multiply(_H.transpose()).multiply(invS);

    // Create a non-square identity matrix
    RealMatrix I = MatrixUtils.createRealMatrix(_K.getRowDimension(), _H.getColumnDimension());
    int dim = Math.min(_K.getRowDimension(), _H.getColumnDimension());

    I = I.add(MatrixUtils.createRealIdentityMatrix(dim));
    _x = _x.add(_K.operate(y));
    _P = I.subtract(_K.multiply(_H)).multiply(_P);

}

public Vector3D getStateV(double z) {
    return Vector3D.cartesian(_x.copy().getEntry(0), _x.copy().getEntry(1), z);
}
```

Obr. 3: Metódy predict, update a getStaveV

Pre naplnenie vyššie uvedených matíc a vektorov slúži trieda ***KalmanAdjuster,*** v ktorej dochádza aj k volaniu vyššie uvedených metód *predict*, *update* a *getStaveV* v metóde *adjustBallPosition* po každej správe od servera (Obr. 3).

K naplneniu matíc a vektorov, nachádzajúcih sa v triede KalmanReal slúži metóda *initMatrixes (Obr. 4)* , ktorá sa volá taktiež po každej správe od servera. Matice a vektory sú zadefinované prostredníctvom vytvorenia nového konštruktora KalmanReal. Hodnoty uvedené v maticiach boli testované a momentálne uvedené sú zatiaľ najlepšie. Určite sa s tými hodnotami dá ešte

vyhrať a pomocou ďalších testov dosiahnuť ešte lepšie výsledky.

```
public void processNewServerMessage(ParsedData data) {
    initMatrixes();
    this.now = data.SIMULATION_TIME;
    adjustBallPosition(data);
    adjustFixedPointsPosition(data.fixedObjects);

}

private void adjustBallPosition(ParsedData data) {
    if (data.ballRelativePosition == null) {
        return;
    }

    /*if (isObsolete(lastTimeBallSeen) || ballKalman == null) {
        //ballKalman = freshKalman();
        kr = new KalmanReal(x, P, F, Q, H, R);
    }*/

    Vector3D relSpeed = WorldModel.getInstance().getBall().getRelativeSpeed();
    RealVector z = new ArrayRealVector(new double[] { data.ballRelativePosition.getX(),
    data.ballRelativePosition.getY(), relSpeed.getX(), relSpeed.getY() });
    kr.predict();
    kr.update(z);
    data.ballRelativePosition = kr.getStateV(data.ballRelativePosition.getZ());
    lastTimeBallSeen = now;
}
```

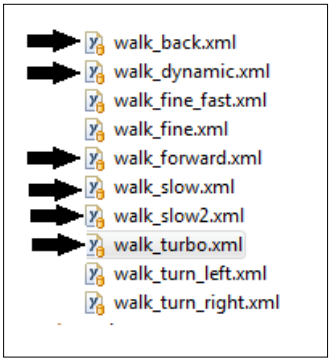
Obr. 4: Metóda adjustBallPosition

```
private void initMatrixes() {

    F = new Array2DRowRealMatrix(new double[][] {
                { 1, 0, 0.1, 0 },
                { 0, 1, 0, 0.1 },
                { 0, 0, 1, 0 },
                { 0, 0, 0, 1 }
                });

    //only observe first 2 values - the position coordinates
    H = new Array2DRowRealMatrix(new double[][] {
                { 0.9, 0, 0.1, 0 },
                { 0, 0.9, 0, 0.1 },
                { 0, 0, 0, 0 },
                { 0, 0, 0, 0 }
                });
    Q = new Array2DRowRealMatrix(new double[][] {
                { 0, 0, 0, 0 },
                { 0, 0, 0, 0 },
                { 0, 0, 0.1,0 },
                { 0, 0, 0, 0.1 }
                });


    R = new Array2DRowRealMatrix(new double[][] {
             { 0.0965, 0, 0, 0 },
             { 0, 0.0965, 0, 0 },
             { 0, 0, 0.1480, 0 },
             { 0, 0, 0, 0.1480 }
             });

    P = new Array2DRowRealMatrix(new double[][] {
             { 1, 0, 0.8, 0 },
             { 0, 1, 0, 0.8 },
             { 0, 0, 0, 0 },
             { 0, 0, 0, 0 }
             });

    x = new ArrayRealVector(new double[] { 0, 0, 0, 0 });


    kr = new KalmanReal(x, P, F, Q, H, R);
}
```

Obr. 5: Metóda initMatrixes

22. Používané chôdze

Bc. Daniel Lukáč



Obr.22 : ukážka xml chôdzí (Jim/moves)

22.1 Kráčanie – rýchle

Začiatočná poloha hráča: Vector3D.cartesian(-5, 1, 0.4);

Pozícia lopty: Vector3D.cartesian0, 0, 0.4);

Test č.	WALK_SLOW	WALK_MEDIUM	WALK_FAST
	pády		
1	0	0	2
2	0	2	0
3	1	1	1
4	1	0	3
5	0	1	2
6	0	0	1
7	1	0	1
8	0	0	0
9	1	0	2
10	0	1	0
11	0	2	0
12	0	1	3
13	0	0	2
14	2	1	1
15	1	2	0
16	0	1	1
17	0	1	2
18	0	0	2
19	1	1	0
20	0	0	1

Tab.1: kráčanie rýchle

22.2 Kráčanie – stabilné

Začiatočná poloha hráča: Vector3D.cartesian(-5, 1, 0.4);

Pozícia lopty: Vector3D.cartesian0, 0, 0.4);

Test č.	WALK_SLOW	WALK_MEDIUM	WALK_FAST
	pády		
1	1	1	4
2	0	2	1
3	1	0	1
4	0	1	1
5	0	1	2
6	1	2	2
7	0	2	0
8	0	0	1
9	1	2	0
10	1	1	3
11	0	2	2
12	1	0	0
13	0	0	2
14	0	1	3
15	2	0	3
16	0	2	0
17	2	1	1
18	0	1	2
19	1	1	1
20	0	0	2

Tab.2: kráčanie stabilné

22.3 Porovnanie

	Rýchla	Stabilná
WALK_SLOW	7	11
WALK_MEDIUM	14	20
WALK_FAST	24	31
Spolu pádov	45	62
Priemer na 1 test	2,25	3,1

Tab.3: porovnanie výsledkov

22.4 Úprava highskillov pre chôdzu

V konečnom dôsledku sa mi nepodarilo nájsť hodnoty, ktoré by zlepšili efektivitu chôdze a tým znížili počet pádov počas nej. Mnou volené hodnoty chôdzu viac zhoršovali a nepodarilo sa mi nájsť spôsob akým znížiť počet pádov (pomocou úpravy highskillov).

Hodnotím, že efektivitu chôdze a zníženie počtu pádov je potrebné riešiť na inej úrovni v projekte(napríklad pracovať na šume zo 61erver alebo zohľadnenie náklonu pri videní čiar).

22.5 Prechod z WALK_MEDIUM na WALK_SLOW

Balík: sk.fiit.jim.agent.highskill.move

Trieda: MovementSkills.java

Funkcia: double getWalkSuitability(MoveSkillPostionObject,MovementSpeedEnum)

Pri tomto tasku som sa venoval aj zmenšeniu vzdialenosti hráča voči lopte, pri prechode z WALK_MEDIUM rýchlosti na WALK_SLOW rýchlosť. Vzdialenosť sa mi podarilo zmenšiť úpravou premennej rangeKoef na hodnotu 0.35(pôvodná 1.0). Avšak túto úpravu je ešte potrebné prekonzultovať, keďže nebola súčasťou tasku.

```
if (speedEnum == MovementSpeedEnum.WALK_SLOW){
    speedKoef = 0.5;
    rangeKoef = 0.35;
    successKoef = 1.5;
}
else if (speedEnum == MovementSpeedEnum.WALK_MEDIUM) {
    speedKoef = 1.0;
    rangeKoef = 0.25;
    successKoef = 1.2;
}
else if (speedEnum == MovementSpeedEnum.WALK_FAST){
    speedKoef = 1.5;
    rangeKoef = 0.15;
    successKoef = 0.01;
}
```

Obr.13: úprava vzdialeností

Hodnotím, že zmena vzdialenosti nemala vplyv na pády Jima počas chôdze, len mierne znížila čas potrebný na presun k lopte v sekundách. Touto úpravou tak hráč najpomalšou chôdzou prejde menšiu vzdialenosť(príklad – po novom sa pomalá rýchlosť aktivuje 1 meter od lopty, po starom 2 metre). V prílohe prikladám video s názornou ukážkou.

22.6 Zmena vzdialenosti

Na určenie vzdialenosti hráča od lopty, pri ktorej hráč spomalí svoj pohyb k lopte sa používa konštanta *EDGE_DISTANCE_FROM_BALL* v balíku *sk.fiit.jim.decision.situation*, konkrétne v triede *Situation*. Hodnota tejto premennej bola nastavená na hodnotu 2. Aby sme dosiahli zrýchlenie hry,

zmenili sme jej hodnotu na 1.5, na základe ktorej hráč spomalí v kratšej vzdialenosti od lopty a skôr sa k nej dostane, vďaka čomu sa zlepší plynulosť samotnej hry (Obr. 1.a).

```
public abstract class Situation {

    public static final double EDGE_DISTANCE_FROM_BALL = 1.5;
```

Obr. 14: nová hodnota konštanty

23. Zisťovanie náklonu z gyroskopu

Bc. Marko Moravčík

Gyroskop ukazuje presné natočenie hráča v rámci súradnicovej sústavy a teda poskytuje dáta aj v prípade, že sa hráč práve nepohybuje.

Dáta z gyroskopu (po ich vykonanej úprave) znázorňujú rotáciu polohy hráča okolo jednotlivých osí (X – dopredu a dozadu, Y – doprava a doľava, Z – smer chôdze vzhľadom na pomyselnú zvislú priamku).

Gyroscope = gyroscope.rotateOverX(-rotationX).rotateOverY(-rotationY).rotateOverZ(-rotationZ);

Dáta z gyroskopu, ktoré zo servera prichádzajú v tvare uhlovej rýchlosti trupu, je potrebné upraviť. Najskôr je nutné získať celkovú polohu hráča a potom túto prekonvertovať z radiánov na stupne.

Úprava dát z gyroskopu:

```
rotationX += Math.toRadians(gyroscope.getX() * TIME_STEP);

rotationX = Angles.normalize(rotationX);

rotationY += Math.toRadians(gyroscope.getY() * TIME_STEP);

rotationY = Angles.normalize(rotationY);

rotationZ += Math.toRadians(gyroscope.getZ() * TIME_STEP);

rotationZ = Angles.normalize(rotationZ);
```

```
AGENT_MODEL: Gyroscope: 7.16 -7.62 -3.85
AGENT_MODEL: Rotation: 6.050491614591326,0.7892194703897707,3.9274006720670704
AGENT_MODEL: My position: 2.0011807794944496 4.764789076323906 -3.3932924599581824

AGENT_MODEL: Gyroscope: 1.48 -2.75 1.66
AGENT_MODEL: Rotation: 6.051161492814601,0.790059590241202,3.928008606766997
AGENT_MODEL: My position: 2.0011807794944496 4.764789076323906 -3.3932924599581824

AGENT_MODEL: Gyroscope: 0.77 -1.34 0.81
AGENT_MODEL: Rotation: 6.051480207605542,0.7904766924931654,3.928317511075732
AGENT_MODEL: My rotation: [-0.20746692339572434,0.7853981633974483,
2.356194490192345
AGENT_MODEL: My position: 3.0224991015413414, 5.535243315818208, -4.749144712675851
AGENT_MODEL: My position: 3.0224991015413414 5.535243315818208 -4.749144712675851

AGENT_MODEL: Gyroscope: 0.55 -0.86 0.53
```

AGENT_MODEL: Rotation: 6.075916657232189,0.7856692204853579,3.927210860695021
AGENT_MODEL: My position: 3.0224991015413414 5.535243315818208 -4.749144712675851

AGENT_MODEL: Gyroscope: 1.6 -2.23 1.24
AGENT_MODEL: Rotation: 6.076371002528098,0.7864181948604685,3.927792084111603
AGENT_MODEL: My position: 3.0224991015413414 5.535243315818208 -4.749144712675851

AGENT_MODEL: Gyroscope: 1.34 -1.84 1.08
AGENT_MODEL: Rotation: 6.076755736802494,0.7870320888014067,3.928290631582682
AGENT_MODEL: My rotation: [-0.14951118181198883,0.7853981633974483,-2.356194490192345
AGENT_MODEL: My position: 3.5413106301508037, 5.840372101323397, -5.346272627470111
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

AGENT_MODEL: Gyroscope: 8.31 1.44 -5.06
AGENT_MODEL: Rotation: 6.130850407240128,0.7871468783183769,3.9278598804850975
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

AGENT_MODEL: Gyroscope: 10.49 3.21 -6.57
AGENT_MODEL: Rotation: 6.126944932926719,0.7889989183006875,3.9289735709055824
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

AGENT_MODEL: Gyroscope: 11.12 3.24 -7.2
AGENT_MODEL: Rotation: 6.12279115508478,0.791031933737831,3.930102968589547
AGENT_MODEL: My rotation: [-0.2986915761170743,0.7853981633974483,-2.356194490192345
AGENT_MODEL: My position: 3.5413106301508037, 5.840372101323397, -5.346272627470111
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

AGENT_MODEL: Gyroscope: 11.35 3.11 -7.35
AGENT_MODEL: Rotation: 5.980178994090464,0.787177899603523,3.928284342713908
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

AGENT_MODEL: Gyroscope: 11.32 2.96 -7.32
AGENT_MODEL: Rotation: 5.975905568856876,0.7889862934926422,3.9295796181995555
AGENT_MODEL: My position: 3.5413106301508037 5.840372101323397 -5.346272627470111

24. Pridanie a setup premenných pre náklon

Bc. Michal Baňas

```
// Tilt of axis X.
double tiltX = 0.0;
// Tilt of axis Y.
double tiltY = 0.0;
// Tilt of axis Z.
double tiltZ = 3.0 * Math.PI / 2.0;

Vector3D tiltVec = Vector3D.ZERO_VECTOR;
```

Obr. 15: Deklarácia premenných pre naklonenie a vektora, ktorý ich obsahuje.

```
// Setting tilt of agent from gyroscope
private void setTilt(Vector3D gyroscope) {
    if (gyroscope == null) {
        return;
    }

    // Delete data saved in variables for tilt
    deleteTilt(tiltVec);
    deleteTilt(tiltX, tiltY, tiltZ);

    tiltX = Math.toRadians(gyroscope.getX() * TIME_STEP);
    tiltX = Angles.normalize(tiltX);
    tiltY = Math.toRadians(gyroscope.getY() * TIME_STEP);
    tiltY = Angles.normalize(tiltY);
    tiltZ = Math.toRadians(gyroscope.getZ() * TIME_STEP);
    tiltZ = Angles.normalize(tiltZ);

    tiltVec = Vector3D.cartesian(tiltX, tiltY, tiltZ);

    LOG.log(LogType.AGENT_MODEL, "Tilt: " + tiltX + "," + tiltY + "," + tiltZ);
    LOG.log(LogType.AGENT_MODEL, "Tilt via Vector: " + tiltVec.getX() + "," + tiltVec.getY() + "," + tiltVec.getZ());
}
```

Obr. 16: Vynulovanie starých premenných, nastavenie premenných a vektoru pre náklon a logovanie ich obsahu

```
// Function for deleting of tilt variables via Vector3D
private void deleteTilt(Vector3D tilt) {
    tilt.setX(0);
    tilt.setY(0);
    tilt.setZ(0);
}

// Function for deleting of tilt variables by resetting them
private void deleteTilt(double x, double y, double z) {
    tiltX = 0;
    tiltY = 0;
    tiltZ = 0;
}
```

Obr. 17: Vymazanie obsahu premenných a vektora pre naklonenie

```
private double getTiltX() {
    //return tiltX;
    return tiltVec.getX();
}

private double getTiltY() {
    //return tiltY;
    return tiltVec.getY();

}

private double getTiltZ() {
    //return tiltZ;
    return tiltVec.getZ();

}
```

Obr. 18: Gettery pre hodnoty náklonu, v komente ťahanie z premenných, pod tým z vektora

```
307         updateJointPositions(data);
308         adjustRotationsFor(data.gyroscope);
309         setTilt(data.gyroscope);
310         updateRotations(data);
311         updatePureBodyAcceleration(data);
```

Obr. 19: Volanie funkcie na nastavenie náklonu

25. Rozšírenie world modelu o náklon

Bc. Michal Baňas

Pre uchovanie informácie o náklone robota boli vytvorené premenné *rotationX, rotationY* a *rotationZ.* Do týchto premenných sa následne ukladá hodnota náklonu od daných osí.

```
// Rotation by axis X value.
double rotationX = 0.0;
// Rotation by axis Y value.
double rotationY = 0.0;
// Rotation by axis Z value.
double rotationZ = 3.0 * Math.PI / 2.0;
```

Obr. 20: Deklarácia a inicializácia premenných

Pre správne nastavenie rotácií je potrebné ich najprv prekonvertovať do intervalu 0 – 2π (0-180°) a následne ich prekonvertovať na stupne.

```
// Logging data about rotation of agent
private void adjustRotationsFor(Vector3D gyroscope) {
    if (gyroscope == null) {
        return;
    }
    LOG.log(LogType.AGENT_MODEL,
            "Gyroscope: " + gyroscope.getX() + " " + gyroscope.getY() + " " + gyroscope.getZ());
    gyroscope = gyroscope.rotateOverX(-rotationX).rotateOverY(-rotationY).rotateOverZ(-rotationZ);
    rotationX += Math.toRadians(gyroscope.getX() * TIME_STEP);
    rotationX = Angles.normalize(rotationX);
    rotationY += Math.toRadians(gyroscope.getY() * TIME_STEP);
    rotationY = Angles.normalize(rotationY);
    rotationZ += Math.toRadians(gyroscope.getZ() * TIME_STEP);
    rotationZ = Angles.normalize(rotationZ);

    LOG.log(LogType.AGENT_MODEL, "Rotation: " + rotationX + "," + rotationY + "," + rotationZ);
}
```

Obr. 21: správne nastavenie rotácie v stupňoch

Na nastavenie hodnôt rotácií pre každý typ rotácie sa používajú na to určené *settery*.

```java
/**
 * Set rotation around X axis
 *
 * @param rotationX
 */
public void setRotationX(double rotationX) {
    this.rotationX = rotationX;
}

/**
 * Set rotation around Y axis
 *
 * @param rotationY
 */
public void setRotationY(double rotationY) {
    this.rotationY = rotationY;
}

/**
 * Set rotation around Z axis
 *
 * @param rotationZ
 */
public void setRotationZ(double rotationZ) {
    this.rotationZ = rotationZ;
}
```

Obr. 22: Settery pre rotácie okolo daných osí

Na zisťovanie hodnôt sa používajú pre každý typ rotácie na to určené *gettery.*

```java
/**
 * Returns agents rotation by axis X.
 *
 * @return double value of rotationX variable
 */
public double getRotationX() {
    return rotationX;
}

/**
 * Returns agents rotation by axis Y.
 *
 * @return double value of rotationY variable
 */
public double getRotationY() {
    return rotationY;
}

/**
 * Returns agents rotation by axis Z.
 *
 * @return double value of rotationZ variable
 */
public double getRotationZ() {
    return rotationZ;
}

/**
```

Obr. 23: Gettery pre rotácie daných osí

26. Analýza nameraných hodnôt z rôznych perceptorov

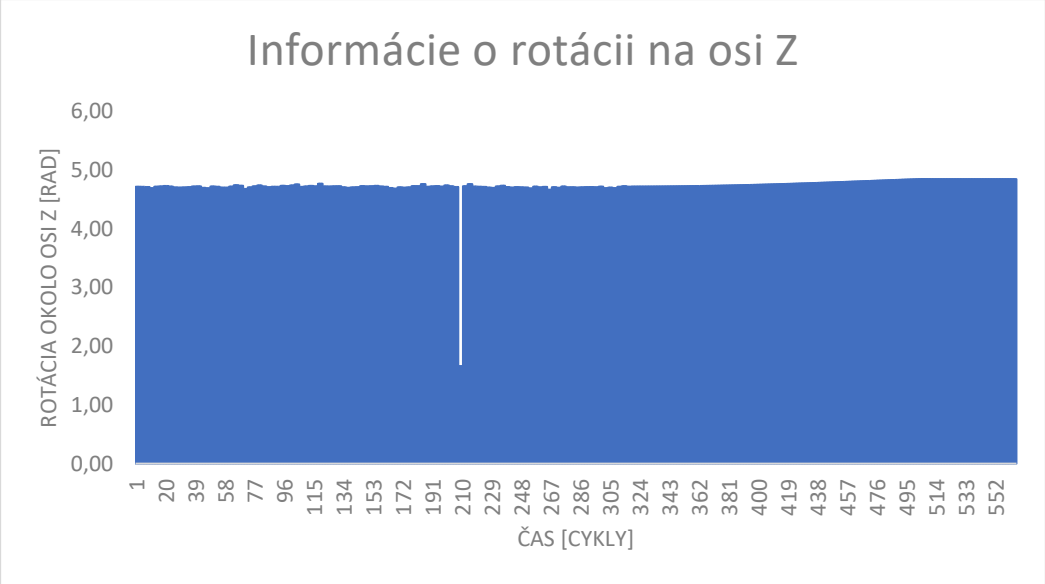
Bc. Ernest Loureiro

26.1 Úvod

Na začiatku tohto dokumentu je reprezentovaná zmena rotácie voči osi Z, tak ako ju vníma agent. Následne dokument popisuje a vizualizuje namerané hodnoty z rôznych perceptorov. Pre každý perceptor je uvedený graf dát reprezentujúcich absolútny náklon z počiatočnej pozície, do pozície predklonu (90°) v priebehu 10 sekúnd. Prvotným cieľom bolo na základe dát z perceptorov a informácii o rotácii analyzovať koreláciu medzi týmito hodnotami, čo sa však ukázalo za nevhodný prístup, z dôvodu malej odchýlky v rotácii medzi 0° a 90° náklonom.

26.2 Informácie o rotácii na osi Z

Zmena súradníc polohy hráča pri izolovanom predklone hráča po osi Z:

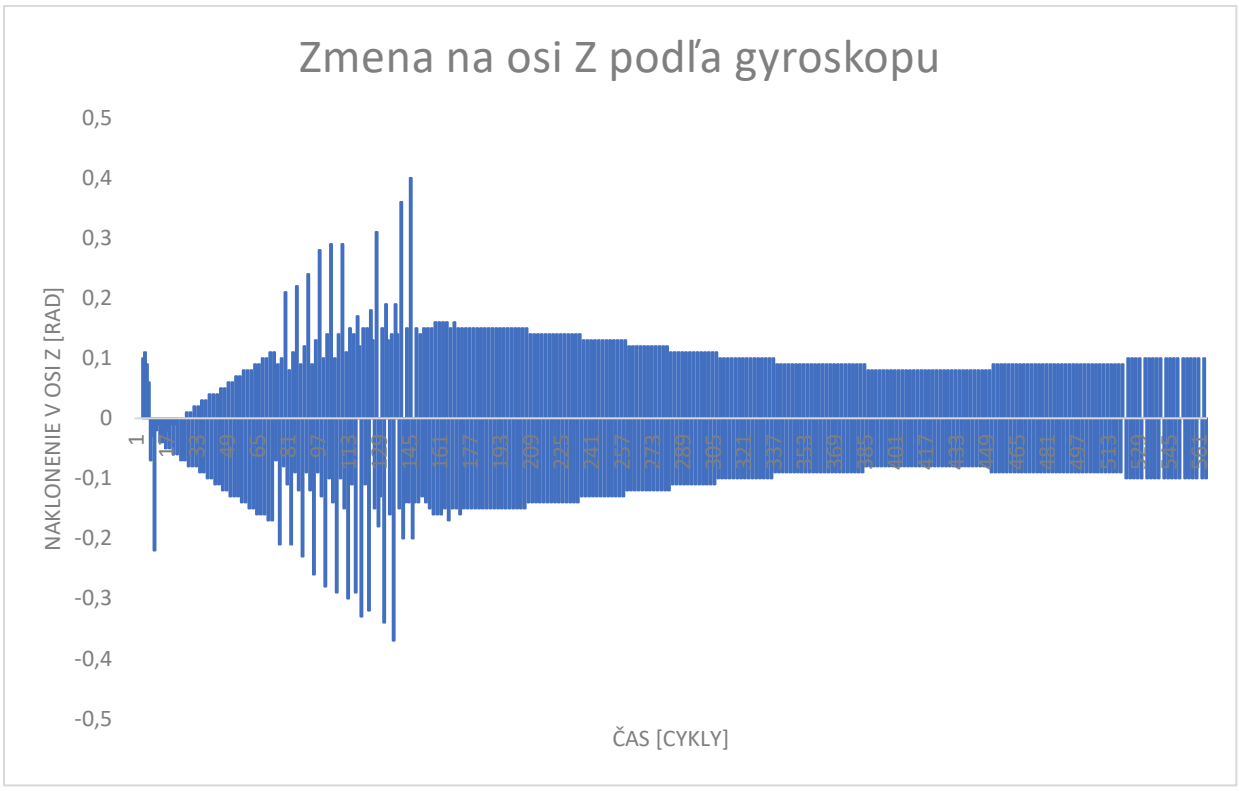


Graf 1: Vedomie agenta o svojej rotácií okolo osi Z

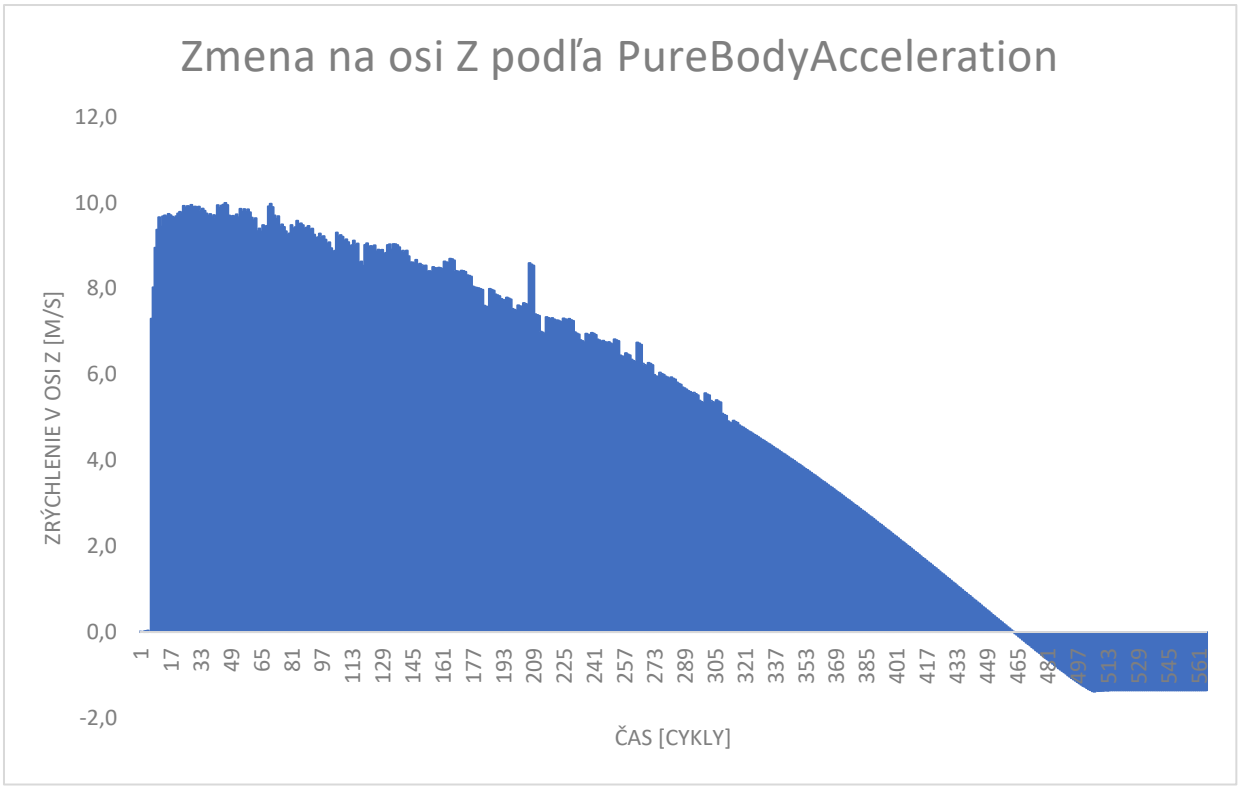
Ako možno vidieť na obrázku rotácia na osi Z je aj pri 90° predklone veľmi nepatrná. Pohybuje sa medzi hodnotami 4,71 pri 0° náklone po 4,84 pri 90° náklone. Tieto hodnoty neodporúčam využívať pre prepočet bodov. Z dôvodu nízkej odchylky pri vysokej zmene uhlu náklonu.

26.3 Dáta z perceptorov

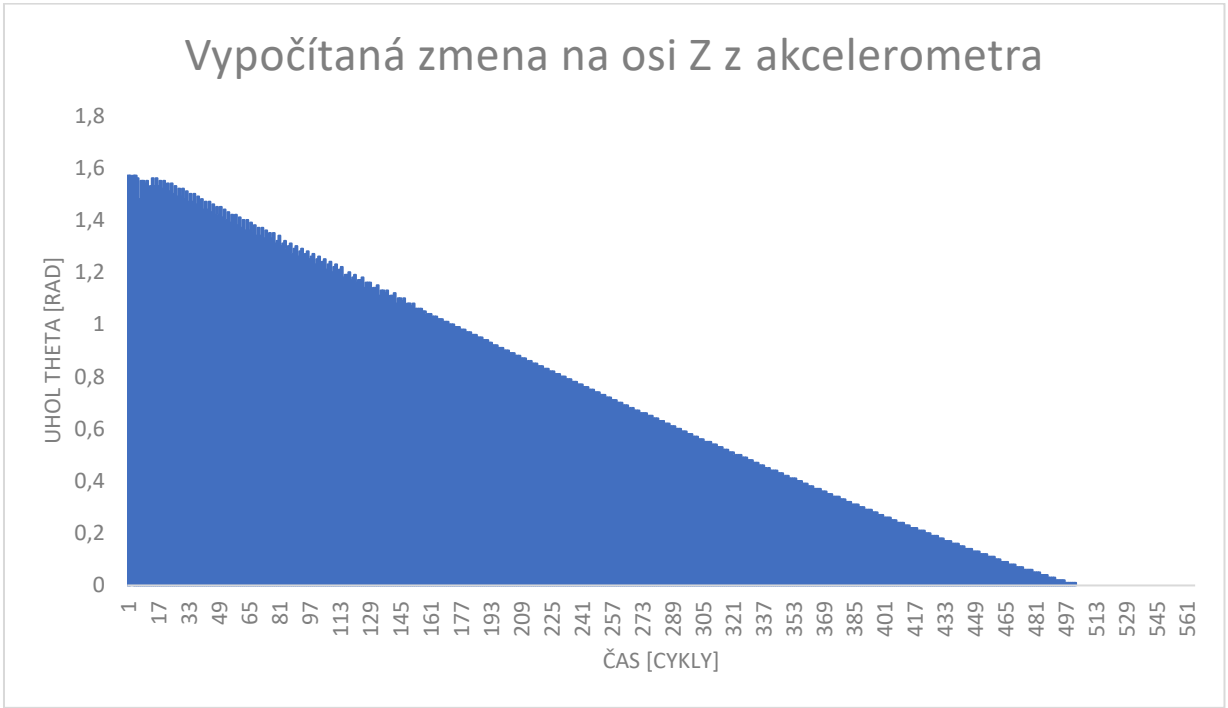
Na základe zozbieraných dát z akcelerometra a gyroskopu uvádzam grafy reprezentujúce zmeny v jednotlivých meraniach. Na základe týchto vizualizácii som vybral vhodných kandidátov na zakomponovanie prepočtu



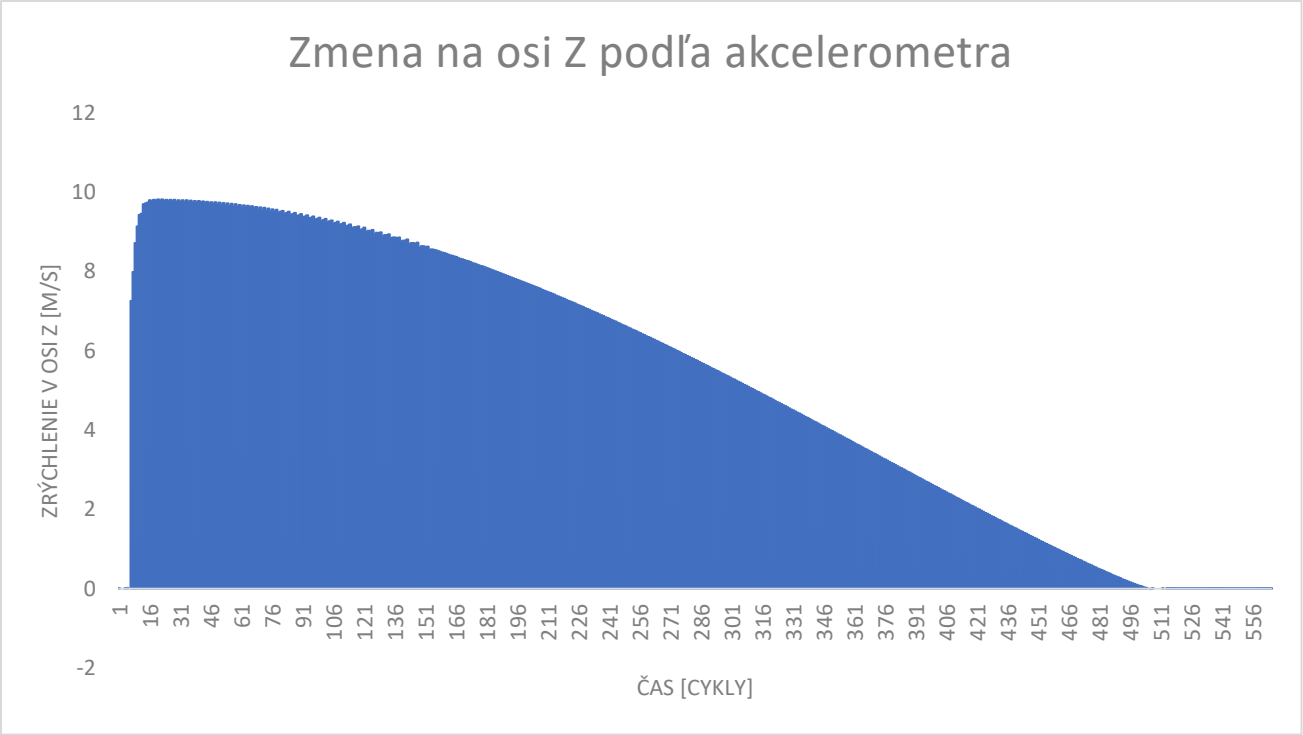
Graf 2: Dáta na osi Z namerané z gyroskopu pri ohnutí agenta do 90°



Graf 3: Dáta na osi Z vypočítané podľa metódy PureBodyAcceleration pri ohnutí agenta do 90°



Graf 4: Uhol theta na osi Z vypočítaný z dát akcelerometra pri ohnutí agenta do 90°



Graf 5: Dáta na osi Z namerané z akcelerometra pri ohnutí agenta do 90°

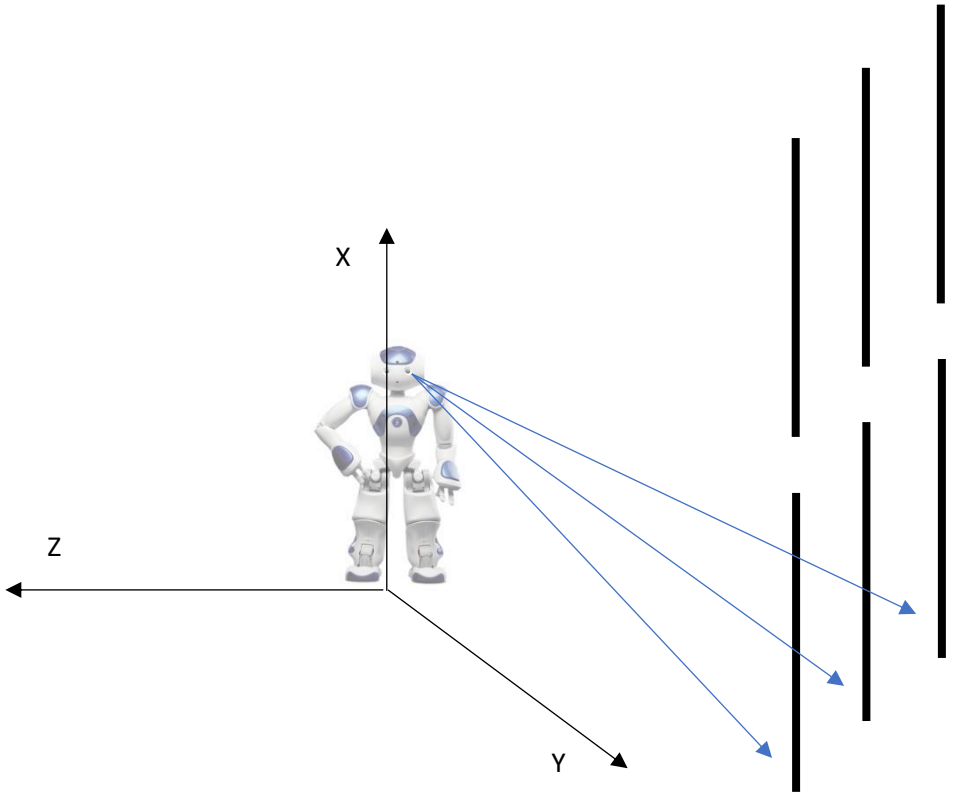
26.4 Záver

Ako možno vidieť na grafoch, vhodnými kandidátmi na prepočet vnímania sveta sú dáta z perceptoru akcelerometru, konkrétne informácia o uhle theta a zmene na osi Z. Tieto dáta zodpovedajú zmene v náklone. V prípade 90° náklonu bola hodnota uhlu theta na akcelerometri 0 a zároveň hodnota osi Z na tomto perceptore bola tiež 0. Z tohto dôvodu predpokladám, že tieto hodnoty majú potenciál pre prepočet vnímania okolitého sveta agentom.

27. Analýza dát na základe nameraných hodnôt z gyroskopu

Bc. Ernest Loureiro

Rotácia agenta je určená troma atribútmi X,Y a Z. Jeho X a Y hodnoty dokážeme vypočítať na základe pohybu pevných bodov.



Obr. 1: Ukážka vnímania agenta pevných bodov (zástaviek)

Z obrázku môžeme vidieť ako vníma agent pevné body. Keďže vieme ich absolútnu pozíciu a vieme ich pozíciu, ktorú vnímame. Vieme povedať ako sa poloha odlišuje od pôvodnej. To však len v prípade, že poznáme pozíciu troch alebo viacerých pevných bodov.

Toto sa v projekte počíta v AgentRotationCalculator.java. Tá sa má dáta o pevných bodoch, ktoré robot videl a má dáta o tom, kde majú absolútnu pozíciu.

To môžeme podľa vzorca:

$$Z_{needed} = Z_{known} - \frac{(Y_2 - Y_1)\, x\, (Z_{known} - Y_1) x\, (Y_2 - Y_1)}{(Y_2 - Y_1)\,.\,(Y_2 - Y_1)}$$

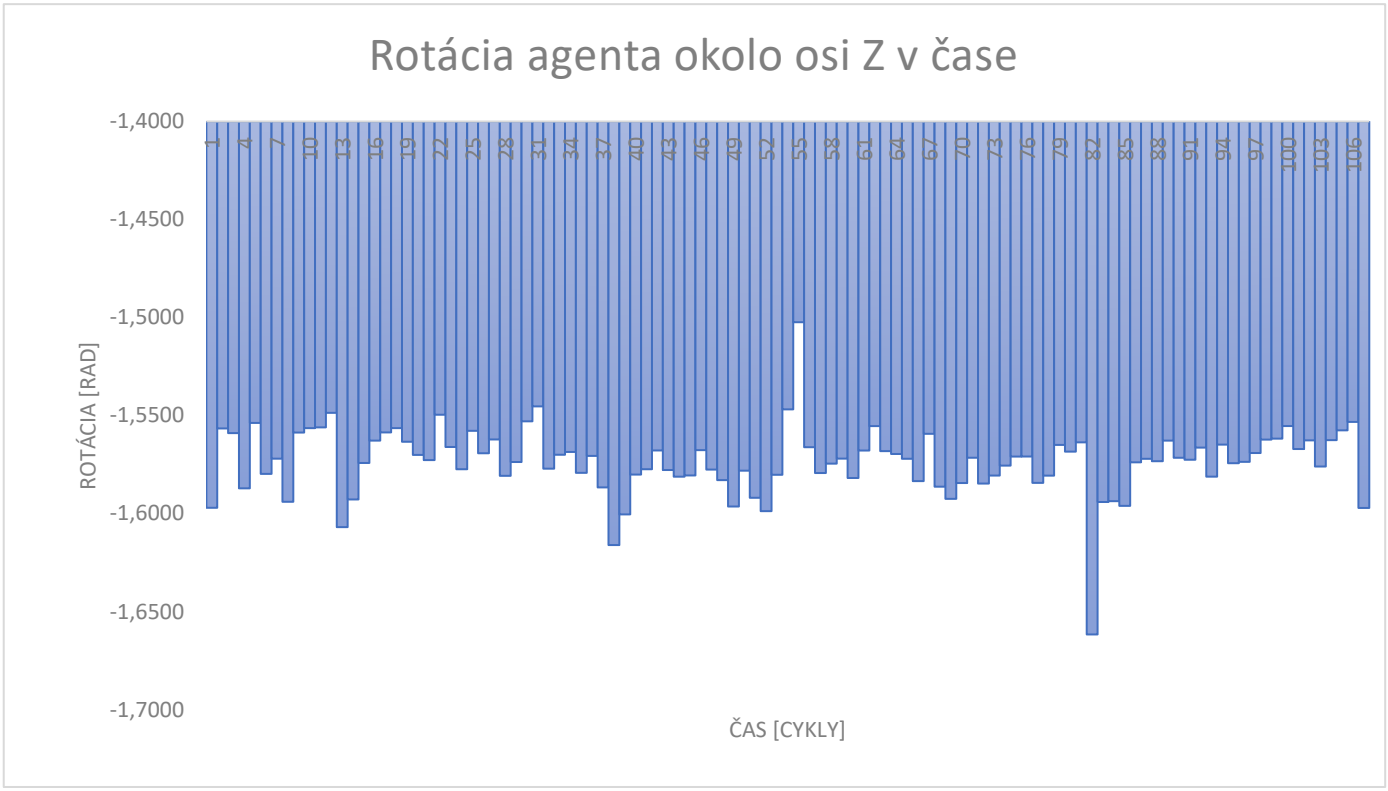
Potom dokážeme rotáciu vypočítať pomocou goniometrických funkcií:

```
double rotationX = asin(-xAxis.getZ());
double rotationZ = atan2( -xAxis.rotateOverX(rotationX).getY() /
cos(rotationX), xAxis.rotateOverX(rotationX).getX() /
cos(rotationX));
double rotationY = atan2( -yAxis.rotateOverX(rotationX).getZ() /
```

```
cos(rotationX), zAxis.rotateOverX(rotationX).getZ() /
cos(rotationX));
```

27.1 Pokus

Urobili sme pokus ako agent vníma svoju rotáciu v čase. Pri testovacom scenári, kde sa zo vzpriamenej polohy dostal na uhol 90°. V grafe 1 sú vyobrazené namerané hodnoty



Graf 6: Zobrazenie hodnôt rotácie agenta okolo osi Z

Z grafu 1 vidíme, že rotácia okolo osi Z nie je meraná v tejto implementácie správne.

27.2 Záver

Algoritmus sme testovali na testovacom scenári, kde sa hráč predkloní do úplného predklonu (teda 90 stupňov) v priebehu 10 sekúnd. Z pokusu sme zistili, že hodnoty vypočítané týmto algoritmom, neboli správne.

Toto je možné nesprávnou implementáciou prepočtu z videných bodov. Svoju úlohu zohráva aj zašumenie dát.

28. Spísané poznatky nového člena tímu

28.1 Inštalácia

1. V prvom rade som sa pokúšal rozbehať projekt na svojom notebooku, na ktorom beží operačný systém windows 10, inštaláciu som robil podľa návodu, no projekt sa mi nedal rozbehať. Po konzultácii s kolegami sme dospeli k záveru, že by bola potrebná reinštalácia windowsu.
2. V druhom rade som skúšal inštaláciu na macOSx prostredníctvom virtualboxu a windowsu 10. V tomto prípade som taktiež postupoval podľa návodu na stránke no nebolo možné získať licenciu na windows 10 prostredníctvom konta na študenta cez dreamspark tak som si stiahol iba trial verziu na 90 dni, ktorá by mala byť postačujúca na dokončenie projektu. No vo virtualboxe sa mi projekt na windows 10 taktiež nepodarilo rozbehať.
3. V poslednom rade som skúšal inštaláciu prostredníctvom bootcampu na macu, kde som si nahral windows 10 a následne som postupoval podľa návodu na stránke, kedy sa mi projekt konečne podarilo rozbehať.

28.2 Analýza projektu / wiki

Z analýzy RobocupTP wiki. Som zistil základné informácie o robotoch.

Aktuálne pohyby robota sú, chôdza,kopnutie,úkroky,otáčania,vstávanie a sadanie,pády.

Hráč/robot sa skladá z 22 kĺbov 6 v kazdej nohe,4 v kazdej ruke a 2 v krku.

High Skill

Low Skill

Funckionalita low skillov, high skillov, taktík a stratégií bola vysvetlená na tímovom stretnutí členom tímu Ernest Loureiro.

28.3 Perceptory

Na vnímanie sveta v RoboCup 3D slúžia perceptory. Server zasiela agentovi informácie o samotnom agentovi, videných objektoch a podobne.

GyroRate slúži na opis orientácie tela robota vzhľadom k jednotlivým osiam. Tento perceptor je umiestnený v hornej časti robota Nao.

HingeJoint určuje o koľko stupňov je ktorý kĺb natočený.

***ForceResistance* perceptor** slúži na oznámenie o pôsobiacej sile na časť tela a jej vektore. Správa obsahuje dvoje súradnice, jedny určujú bod, na ktorý sila pôsobí a druhé súradnice určujú vektor tejto sily

Accelerometer perceptor umiestnený v hornej časti tela počíta zrýchlenie.

***Vision* perceptor** dostáva informácie o objektoch videných agentom (ostatní agenti, lopta, statické body ihriska). Zorné pole robota Nao je 120°.

GameState vysiela niekoľko základných informácií o aktuálnom stave hry.

HearPercepto*r* – priama komunikácia medzi robotmi nie je povolená, ale môžu si vymieňať správy cez simulačný server. Na túto komunikáciu slúži tento perceptor.

28.4 Efektory

Create – v momente ako je agent pripojený na server nie je viditeľný a nemôže zasiahnuť do simulácie. Pomocou Create hráč odovzdá serveru cestu ku konfiguračnému súboru, ktorý definuje fyzickú reprezentáciu agenta a množinu jeho perceptorov a efektorov.

HingeJoint efektor dávajúci príkaz na ohnutie kĺbu.

***Beam* efektor** určuje umiestnenie agenta na hracej ploche pred začatím polčasu

Say efektor slúži na odoslanie správy pre komunikáciu medzi hráčmi.

Init priradí hráča k tímu a pridelí mu číslo. Ak agent posiela ako číslo 0, server priradí hráčovi najbižšie voľné číslo automaticky

28.5 Analýza modelu sveta

Model sveta je obsiahnutý v komponente Models. Kde sa nachádzajú 3 časti, a to AgentModel, EnvironmentModel a WorldModel. V triede AgentModel sa počítajú pozície a rotácie kĺbov, v triede WorldModel sa počítajú pozície ostatných hráčov a lopty a v triede EnvironmentModel sa ukladá mód hry, čas a verzia servera.

28.6 Analýza matiky a logiky

V tejto časti sú popísané výpočty jednotlivých tried.

28.7 Analýza výstupov z gyroskopu a akcelerometra

V tejto časti je vykonaná analýza dát z jednotlivých receptorov.

28.8 Jednoduché pohyby

Táto podkapitola sa zameriava na analýzu jednoduchých pohybov. Od vstávania z brucha až po kop do lopty.

28.9 Komunikácia agenta so serverom

Celá hra prebieha ako interakcia medzi agentom a serverom. Server má informácie o aktuálnom stave hry, tj. Rozloženie hráčov, čas, poloha lopty, … . Tento stav sa dá graficky zobraziť pomocou aplikácie rcssmonitor3d.Server poskytuje minimálne dva typy robotov – „Nao“ a

„Socerbot“. Nao predstavuje virtuálnu reprezentáciu rovnomenného robota a je používaný aj v projekte Robocup na FIIT.

28.10 High Skill

XML súbory s low skillmi a fázami musia pre použitie v high skilloch okrem vyššie uvedených požiadaviek spĺňať nasledovné: každý musí mať aspoň jednu fázu s isFinal=true

Je tu taktiež opísaný priebeh vykonávania high skillu a sú tu aj opísané existujúce high skilly

28.11 Low Skill

Jednotlivé nižšie pohyby agenta sú opísané prostredníctvom XML súborov (nižšie pohyby sú napríklad kopnutie do lopty, postavenie sa, otočenie hlavou a podobne). Každý pohyb sa skladá z viacerých fáz, pričom každá fáza predstavuje určitú zmenu natočenia určitých kĺbov za určitý čas. V tejto kapitola je taktiež popísaná štruktúra xml súborov.

28.12 Vyhodnocovanie polohy agenta

Výpočet polohy agenta prebieha v triede AgentPositionCalculator. Výpočet polohy pozostáva z nasledujúcich krokov:

1. Prijatie správy zo servera
2. Sparsovanie informácii
3. Vyhodnotenie polohy na základe jednej čiary
4. Mapovani čiar ak agent vidí iba čiary
5. Mapovani čiar ak agent vidí aspoň jeden kontrolny bod
6. Výpočet polohy agenta
7. Aktualizácia polohy
8. Aktualizácia histórie agenta

Všetky tieto kroky sú v kapitole podrobne vysvetlené.

28.13 Správy zo servera

Správu zo servera agent dostáva približne každých 0.02s, pričom každá tretia takáto správa obsahuje informácie o tom, čo agent vidí (podľa simspark wiki by mala každá druhá správa obsahovať informácie o tom, čo agent počuje).

28.14 Vytvorenie pohybu v editore

V tejto kapitole je opísaný celý proces vytvárania pohybu v editore.

28.15 Vzory Čiar na Ihrisku

V tejto sekcii je opísaná trieda LinePatternRecognition a aj všetky jej funkcie.

28.16 Analýza metodík

V rámci tohto šprintu som si prešiel aj jednotlivé metodiky. A to konkrétne:

- **Metodika konvencie kódu**. – metodika sa zaoberá tým ako by mal byť správne písaný kód, ako písať komentáre a ako formátovať kód a ako robiť deklarácie.
- **Metodika kontrolovania kódu. –** metodika sa zaoberá tým ako kontrolovať kód a opisuje celý postup od pull requestov až po opravenie kódu.
- **Metodika vykazovania prác v nástroji scrumdesk. –** metodika opisujúca celý proces vykazovania práce, definuje typy úloh
- **Metodika písania dokumentov.**
- **Metodika komunikácie. –** metodika opisuje spôsobom komunikácie medzi členmy tímu, definuje uložíská a aj nástroje v ktorých sa komunikuje.
- **Metodika rizík**.
- **Metodika testovania.**
- **Metodika commitovania a verziovania.**

29. Analýza fixácie osi Z pri vnímaní čiar.

Bc. Daniel Lukáč

Bc. Ernest Loureiro

V stave v akom sme projekt preberali, agent nemal zohľadnené vnímanie čiar pre nakláňanie na osi Z. Našou úlohou bolo analyzovať, kde dochádza k vnímaniu čiar z kamerového senzoru.

Vnímanie čiar (trieda WorldModel.java):

```
private void calculateLines(ParsedData data) {
    List<ParsedLineWithFlags> linesFromServer = data.lines;
    lines.clear();
    int i = 1;

    for (ParsedLineWithFlags line : linesFromServer) {
        Line ln = new Line();
        // absolute positions
        ln.setPosition1(agentModel.globalize(line.getPosition1()));
        ln.setPosition2(agentModel.globalize(line.getPosition2()));

        // relative positions
        ln.setLine(line);

        LOG.log(LogType.SINGLE_LOG,  msg: "Line pos1-" + i + ":" + line.getPosition1());
        LOG.log(LogType.SINGLE_LOG,  msg: "Line pos2-" + i + ":" + line.getPosition2());
        lines.add(ln);
    }
}
```

Obr 1: Aktuálna implementácia vnímania

Ako možno vidieť na predchádzajúcom úryvku kódu, postupne sa prechádza zoznam čiar, ktoré agent vidí a prepočítava sa ich poloha. Problém nastáva pri predklone hráča. Celý jeho svet sa začne nakláňať a tým pádom jednotlivé čiary sa zdvíhajú do vzduchu. Úvaha nad týmto fenoménom nás priviedla k záveru, že body od ktorých sa jednotlivé čiary vypočítavajú, bez ohľadu na náklon budú vždy pre os Z označujúcu výšku vždy v tom istom bode.

Na základe tejto úvahy sme nastavili na pevno pre tieto body súradnice na osi Z fixne na - 0.56. Po simulácii vnímania čiar pomocou TestFrameworku sme zistili že dochádza ku kuželovitému skresleniu, keďže jedna zo súradníc v trojrozmernom priestore bola zafixovaná. Tento fakt však nemá kardinálny dopad za bežných okolností a vnímanie čiar hráča pri hre je efektívnejšie aj napriek tomuto skresleniu.

Súradnice na osi Z boli zafixované nasledovným spôsobom:

```
for (ParsedLineWithFlags line : linesFromServer) {
    Line ln = new Line();
    // absolute positions
    ln.setPosition1(agentModel.globalize(line.getPosition1().setZ(-0.56d)));
    ln.setPosition2(agentModel.globalize(line.getPosition2().setZ(-0.56d)));
```

Obr. 3: Nastavenie súradníc osi Z

30. Analýza fixácie osi Z pri vnímaní čiar pre loptu.

- Bc. Daniel Lukáč
- Bc. Ernest Loureiro
- Bc. Richard Pintér

V stave v akom sme projekt preberali, agent nemal zohľadnené vnímanie lopty pre naklánanie na osi Z. Našou úlohou bolo analyzovať, kde dochádza k vnímaniu lopty z kamerového senzoru.

Vnímanie lopty (trieda WorldModel.java):

```
// Calculates ball's position from specified parsed data.
private void calculateBallPosition(ParsedData data) {
    returnBallRelativePosition(data.ballRelativePosition, data.SIMULATION_TIME);
    LOG.log(LogType.WORLD_MODEL,  msg: "Player IDs are not implemented yet!!!");
}
```

Obr. 1: Aktuálna implementácia vnímania

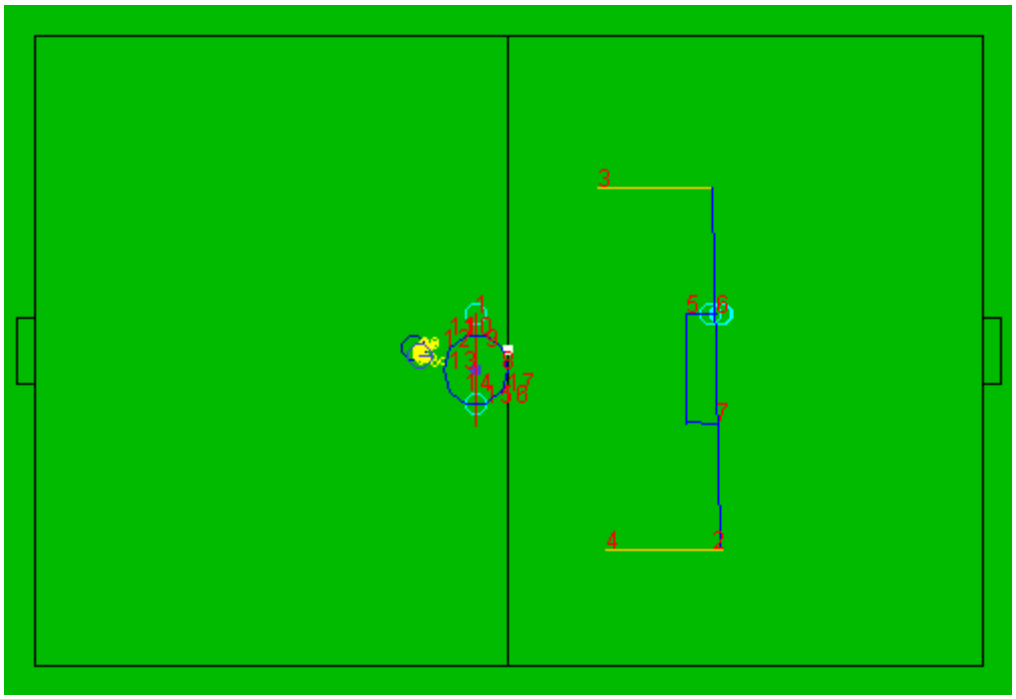
Obr. 2 Vnímanie sveta pri predklone

Ako možno vidieť na *Obr. 1: Aktuálna implementácia vnímania*, pre zistenie polohy lopty prebieha obyčajný výpočet na základe relatívnych súradníc. Keďže sme mali z predchádzajúceho šprintu

skúsenosti s analýzou osi Z pre čiary, skúsili sme os Z zafixovať.

Obr. 2: Vnímanie sveta pri predklone reprezentuje vnímanie sveta pri náklone. Pri vnímaní dochádza ku kuželovitému skresleniu, ktoré však v praxi je efektívnejšie ako celkový náklon sveta. V prípade nezafixovania osi Z pre výpočet polohy lopty je evientne vidieť, že lopta nepodlieha kuželovitému skresleniu (vnímaná pozícia lopty – modrá bodka, je mimo stredového kruhu).

Toto pozorovanie a predchádzajúce skúsenosti nás priviedli k záveru, že je možné vyskúšať fixáciu osi Z aj na loptu. Problém by sa v budúcnosti mohol vyskytnúť v prípade kopu do výšky, avšak pre aktuálne schopnosti hráča je táto funkcionalita postačujúca.



Obr. 3: Vnímanie sveta po fixácii osi Z

Ako reprezentuje *Obr. 3: Vnímanie sveta po fixácii osi Z* , lopta ostáva v strede kruhu a teda je na ňu aplikované kuželovité skreslenie tak, ako na čiary, čo v praxi vedie k lepšiemu výsledku hráča, nakoľko jeho svet sa skresluje rovnomerne.

Súradnice na osi Z boli zafixované nasledovným spôsobom:

```
for (ParsedLineWithFlags line : linesFromServer) {
    Line ln = new Line();
    // absolute positions
    ln.setPosition1(agentModel.globalize(line.getPosition1().setZ(-0.56d)));
    ln.setPosition2(agentModel.globalize(line.getPosition2().setZ(-0.56d)));
```

Obr 4: Nastavenie súradníc osi Z

31. Návod na používanie nového logovania

- Bc. Lukáš Rešutík

31.1 Logovanie do súboru

Trieda *JimLocalFileCreator* umožňuje vytvorenie logovania do vlastného súboru. Pokiaľ je použitý konštruktor bez argumentov, vytvorí sa defaultne súbor **JavaLocalLogs.txt** a vytvorí singleton inštanciu**.** Metóda *writeLog(String text)* zapíše text do jedného riadku logovacieho súboru so spomenutým examplom. Použitie tohto spôsobe pri každom reštarte Jima zmaže **JavaLocalLogs.txt** súbor.

```
@Test
public void testSingletonInstance() {
    JimLocalFileCreator.getInstance().writeLog("Log 0");;
    for(int i = 0; i < 100; i++) {
        JimLocalFileCreator.getInstance().writeLog("Log " + (i+1));
    }
}
```

Obr.1 - logovanie do súboru

31.2 Logovanie do súboru s vlastným menom

Pokiaľ chceme logovať do súboru s vlastným názvom trieda *JimLocalFileCreator*ponúka konštruktor s parametrom názvu súboru. Tento konštruktor vytvorí súbor so zadaným menom. Následne logovanie umožňuje metóda writeLogFile(String text). Nie je nutné zadávať meno súboru.

```
public void testLogovanie() {
    JimLocalFileCreator file = new JimLocalFileCreator("Testovanie.txt");
    file.writeLogFile("Prvy riadok");
    file.writeLogFile("Druhy riadok");
}
```

Obr.2 - logovanie do súboru s vlastným menom

31.3 Logovanie do súboru s vlastným menom aj s časom a dátumom vytvorenia záznamu

Trieda *JimLocalFileCreator*umožňuje zapisovanie do logovacie súboru s aktuálnym časom (hhmmss).

Táto funkcionalita je prístupná metódou writeLogFile(Stringtext, booleandate). Treba poslať ako argument datetrue aby logovalo s dátumom.

```
public void testLogovanie() {
    JimLocalFileCreator file = new JimLocalFileCreator("Testovanie2.txt");
    file.writeLogFile("Prvy riadok", true);
    file.writeLogFile("Druhy riadok", true);
}
```

Obr.3 - logovanie do súboru s vlastným menom aj dátumom vytvorenia záznamu

31.4 Logovanie do csv súboru

Trieda *JimLocalCsvFileCreator* poskytuje možnosť vytvorenia vlastného csv súboru. Konštruktor triedy s parametrom mena súboru vytvorí súbor do ktorého môžeme zapisovať metódou writeCsvLo(String...stlpceCsv). Konštruktor triedy taktiež umožňuje nastaviť separátor a ohraničnia dát v csv súbore.

```
public void createCsvFile() {
    JimLocalCsvFileCreator csv = new JimLocalCsvFileCreator("TestovacieCsv");
    csv.writeCsvLog(new String[] {"0","prve","3","17"});
    csv.writeCsvLog(new String[] {"10", "druhe", "5", "32"});
}
```

Obr.4 - logovanie do csv súboru

31.5 Logovanie do vlastného csv súboru aj s dátumom

Pokiaľ chceme pridať do csv s dátumom zápisu trieda *JimLocalCsvFileCreator* vlastní metódu writeCsvLog(boolean Date, String...data). Použitím true ako argumentu boolean date sa csv súbor vytvorí spolu s dátumom zapísania.

```
public void createCsvFile() {
    JimLocalCsvFileCreator csv = new JimLocalCsvFileCreator("TestovacieCsv4");
    csv.writeCsvLog(true, new String[] {"0","prve","3","17"});
    csv.writeCsvLog(true, new String[] {"10", "druhe", "5", "32"});
}
```

Obr.5 - logovanie do csv súboru s dátumom zapísania

32. Lokalizácia implementácie presných súradníc

Dávid Roba

Implementácia presných súradníc lopty, na ktorých sa aktuálne nachádza je implementovaná v triede *Scene.java*, ktorá sa nachádza balíku *sk/fiit/testframework/worldrepresentation/models*. Na získanie aktuálnej pozície slúži funkcia *getBallLocation*, ktorá je znázornená na Obr. 1.a.

```
/**
 * Returns ball location
 * @return ballLocation - instance of Vector3D
 */
public Vector3D getBallLocation() {
    return ballLocation;
}
```

Obr. 1.a: Funkcia na získanie polohy lopty

Na nastavenie aktuálnej polohy lopty slúži funkcia *setBallLocation*, ktorá nastaví novú pozíciu lopty a následne prepíše predchádzajúcu pozíciu tou aktuálnou. Táto funkcia je znázornená na Obr. 1.b.

```
/**
 * Sets new location of ball and write previous ball location
 * @param ballLocation - new location of ball
 */
public void setBallLocation(Vector3D ballLocation) {
    this.previousBallLocation=this.ballLocation;
    this.ballLocation = ballLocation;
    //logger.info("Predchadajudza poloha: " + previousBallLocation);
    //logger.info("Aktualna poloha: " + ballLocation);
}
```

Obr.1.b: Funkcia na nastavenie aktuálnej pozície lopty

V rámci tejto triedy je tiež implementovaná funkcia *isBallMoving*, ktorá slúži na výpočet vzdialenosti medzi predchádzajúcou a aktuálnou pozíciou na danej osi. Funkcia je znázornená na Obr. 1.c.

```
public boolean isBallMoving(){
    return this.previousBallLocation.getXYDistanceFrom(this.ballLocation)>0;
}
```

Obr.1.c: Funkcia na výpočet vzdialenosti medzi predchádzajúcou a aktuálnou pozíciou lopty

Výpočet vzdialenosti sa vykoná volaním funkcie *getXYDistanceFrom*, kde sa vykoná výpočet zobrazený na Obr. 1.d.

```
/**
 * Returns distance from another vector.
 *
 * @param b
 *            vector
 * @return distance from another vector
 */
public double getXYDistanceFrom(Vector3D b) {

    if(this.equals(b)) {
        return 0.0;
    }

    return (Math.sqrt(Math.pow(x - b.x, 2) + Math.pow(y - b.y, 2)));
}
```

Obrázok 1.d: Funkcia na výpočet vzdialenosti

Všetky tieto hodnoty sa zobrazujú po spustení testframeworku v gui okne, ktoré je možné vidieť na Obr. 1.e.

```
Ball
======
At start: x, y, z: [0,00, 0,00, 0,04] r, phi, theta: [0,04, 4,71, 1,57]
Now: x, y, z: [-15,32, 0,00, 0,07] r, phi, theta: [15,32, 1,57, 0,00]
Dist: x, y, z: [-15,32, 0,00, 0,02] r, phi, theta: [15,32, 1,57, 0,00]
```

Obr.1.e: Výpis hodnôt z predchádzajúcich funkcií

33. Logovanie súradníc do súborov

Autor: Dávid Roba

Na logovanie relatívnych súradníc do súboru sa využil novo naimplementované logovanie, vďaka ktorému bolo jednoduchšie získať tie údaje, ktoré boli pre nás podstatné. Pre potreby overenia implementácie Kalman filtra bolo potrebné do rozdielnych súborov logovať relatívne súradnice lopty, ktoré dostáva hráč pred aplikovaním Kalman filtra a následne aj relatívne súradnice lopty po aplikovaní Kalman filtra. Logovanie týchto súradníc sa nachádza v Jimovi v triede *KalmanAdjuster*. Kód je znázornený na Obr. 1.a.

```
file1.writeCsvLog("Lopta_PREDK",String.valueOf(data.ballRelativePosition.getX()),
        String.valueOf(data.ballRelativePosition.getY()),
         String.valueOf(data.ballRelativePosition.getZ()));

file2.writeCsvLog("Lopta_PREDK_CONST",String.valueOf(data.ballRelativePosition.getX() - 3.0),
         String.valueOf(data.ballRelativePosition.getY() + 0.0),
         String.valueOf(data.ballRelativePosition.getZ() + 0.0));

ke.predict();
ke.update(data.ballRelativePosition.getX(),data.ballRelativePosition.getY(), R);
data.ballRelativePosition = ke.getState();

file3.writeCsvLog("Lopta_POK",String.valueOf(data.ballRelativePosition.getX()),
        String.valueOf(data.ballRelativePosition.getY()),
         String.valueOf(data.ballRelativePosition.getZ()));

file4.writeCsvLog("Lopta_POK_CONST", String.valueOf(data.ballRelativePosition.getX() - 3.0),
         String.valueOf(data.ballRelativePosition.getY() + 0.0),
         String.valueOf(data.ballRelativePosition.getZ() + 0.0));
```

Obrázok 1.a: Logovanie relatívnych súradníc

Na logovanie absolútnych súradníc lopty sme opäť využili nový spôsob logovania priamo do súboru. Kód logovania sa nachádza v TestFrameworku v triede *Scene* a je znázornený na Obr. 1.b.

```
public void setBallLocation(Vector3D ballLocation) {
    this.previousBallLocation=this.ballLocation;
    this.ballLocation = ballLocation;
    file5.writeCsvLog("Lopta_ABS",String.valueOf(ballLocation.getX()),
            String.valueOf(ballLocation.getY()),
            String.valueOf(ballLocation.getZ()));
}
```

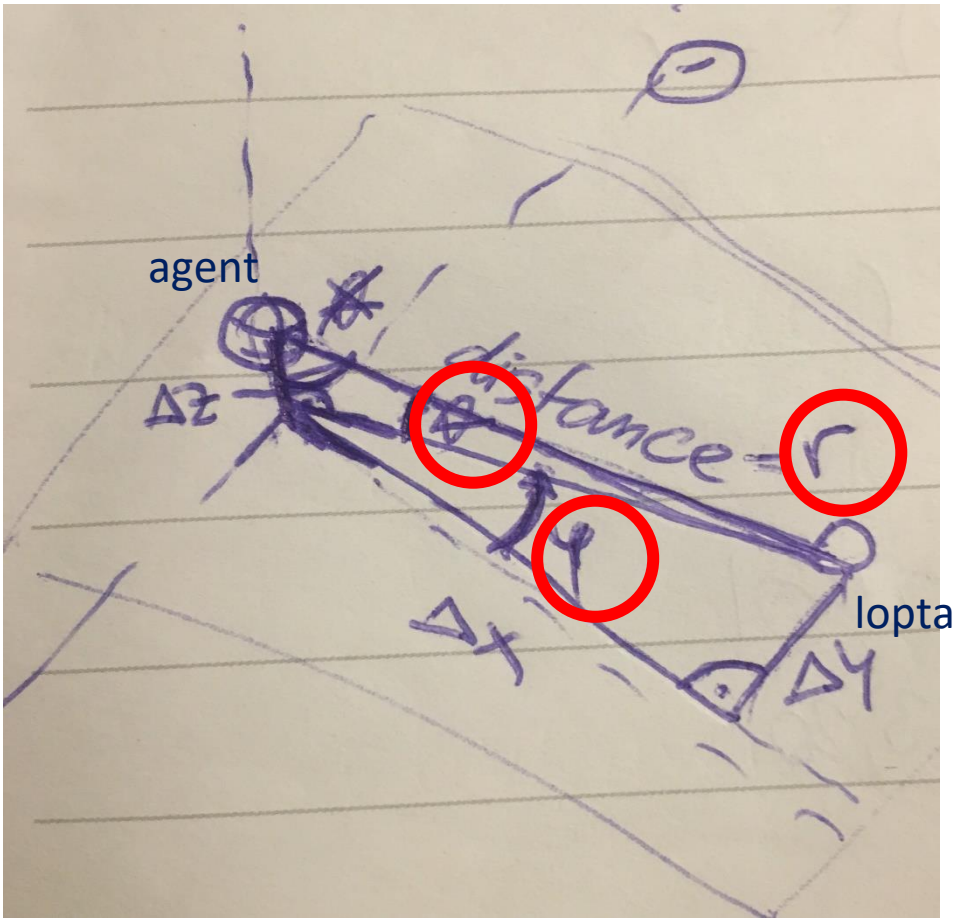
Obrázok 1.b: Logovanie absolútnych súradníc

Týmto spôsobom je teda možné jednoducho testovať rôzne hodnoty matíc Kalman filtra a následne ich porovnať s absolútnymi hodnotami pozície lopty.

34. Sférické súradnice a vektory v projekte RoboCup

V prípade simulovaného robotického futbalu sa používajú tzv. **Sférické súradnice**. Je to system súradníc, ktorý sa skladá z troch komponentov:

- r – vzdialenosť
- θ – vertikálny uhol (medzi Z a Y)
- ϕ – horizontálny uhol (medzi X a Y)



Obr. 4: Ilustrácia sveta pomocou sférických súradníc

Ako možno vidieť na Obr. 1: Ilustrácia sveta pomocou sférických súradníc, **horizontálny uho**l označuje o koľko stupňov sa objekt predomnou nachádza vľavo/vpravo (0 = priamo predo mnou) a **vertikálny uhol** určuje pod akým uhlom je objekt k mojmu perceptoru naklonený (POZOR ! je to uhol pri objekte nie pri perceptore – hlave) Dĺžka úsečky od perceptoru k objektu je označená ako **r** – **vzdialenosť**. Jednotlivé delty potom označujú posun objektu voči agentovi v rámci jednotlivých osí. Správy zo servera o objektoch (hráči, lopta a fixné body) prichádzajú do agenta v sférických

súradniciach. Pre prevody sférických súradníc na kartéziánské a naopak slúži trieda *Vector3D*. Na základe týchto prevodov je možné vypočítať relatívne súradnice objektov pre systém so začiatkom na mieste agenta.

```
/*
 * Computes cartesian coordinates from spherical coordinates.
 */
private void calculateCartesian() {
    y = cos(theta) * cos(phi) * r;
    y = (int) (y * 1000.0) / 1000.0;
    x = cos(theta) * sin(phi) * r;
    x = (int) (x * 1000.0) / 1000.0;
    z = sin(theta) * r;
    z = (int) (z * 1000.0) / 1000.0;
}
```

Obr. 5: Prevod do karteziánskych súradníc

```
/*
 * Computes spherical coordinates from cartesian coordinates.
 */
private void calculateSpherical() {
    r = sqrt(x * x + y * y + z * z);
    theta = asin(z / r);
    if (r == 0.0)
        theta = 0.0;
    phi = atan2(y, x);
    phi -= PI / 2.0d;

    phi = Angles.normalize(phi);
    theta = Angles.normalize(theta);
}
```

Obr. 6: Prevod na sférické súradnice

Trieda *Vector3D* zároveň implementuje operácie spojené s vektormi, ako aj možnosť vytvorenia nového vektora na základe karteziánskych alebo sférických súradníc. Pre zvolenú možnosť je následne vykonaný prepočet do druhej sústavy a je vrátený vektor obsahujúci komponenty pre obe sústavy.

35. Refactor zaseknutia Jima v testframeworku

Bc. Daniel Lukáč

- príčina: zahltenie pamäte pri výpisoch do konzoly v testframeworku tým pádom spomalenie celkového fungovania Jima
- riešenie: vypol som logovanie do konzoly, skúšal som viaceré a krajšie varianty ale všetko bolo horšie ako toto

- dôsledky:

- Jim prestal toľko padať
- dokáže sa prekopať ku gólu aj na 2 kopy (110 sekúnd)
- myslím že vypnutie logovania do konzoly zrýchlilo celý kód a tým pádom prebiehajú aj lepšie výpočty a spracovávanie dát zo servera

- analýza:

- hodnotím, že fyzika Jima je v dobrom stave, Jim toľko nepadá a dokáže sa udržať, pri najlepšom prípade dal gól len s jedným spadnutím
- kód je príliš pomalý a komplikovaný, čo spomaľuje všetky procesy Jima pre jeho správne fungovanie
- ak som pridal výpis ďalších logov do konzoly vývojového prostredia, Jim sa ešte zhoršil, takže netreba mať príliš veľa logov na výpis, odporúčam len tie potrebné
- lepšie fungovanie Jima nastáva ak je debug taktika nastavená v properties na true (obrázok 2)
- ak som nastavil debug taktiku v properties na false (obrázok 3), fungovanie Jima sa mierne zhoršilo, no aj tak bol schopný dopracovať sa ku gólu v lepšom čase ako doteraz
    - z posledných 2 odrážok hodnotím, že taktiky, ktoré Jim využíva sú v zlom stave a práve tie zhoršujú jeho fungovanie, keďže mám pocit, že jeho stabilita a orientácia je v dobrom stave

```java
private void readForever(String agentName, BufferedReader stream) {
    String line = null;

    try {

        while((line = stream.readLine()) != null) {
            if(line.length() > 100)
                break;
            AgentJim agent = runningAgents.get(agentName);

            if (agent != null) {
                agent.getStdout().append(line + "\n");

                for (IAgentManagerListener listener : managerListeners) {
                    listener.agentOutputLine(agent,  line: line + "\n");
                }
            }
        }
        stream.close();
    }
    catch (IOException e) {
        logger.warning( msg: "Error while reading agent's output: " + e.getMessage());
    }
}
```

Obr.1 - vypnutie logovania AgentManager.java

```
# Debug tactic
DEBUG_TACTIC_ENABLE = true
DEBUG_TACTIC_NAME = KickToVector
```

Obr.2 - debug taktika true

```
# Debug tactic
DEBUG_TACTIC_ENABLE = false
DEBUG_TACTIC_NAME = KickToVector
```

Obr.3 - debug taktika false

36. Zistenie implementácie prevodu relatívnych súradníc na absolútne

Bc. Ernest Loureiro

Výpočet rotácie agenta prebieha v triede *AgentRotationCalculator*. V tejto triede sa na základe videných bodov (minimálne 3 na jednej strane ihriska) vypočíta os Y, pre rotáciu. Následne sa dopočíta kolmá os Z na ňu. Pri oprave vymenených súradníc X a Y došlo k problému v tejto implementácii. Keďže táto funkcionalita bola vyvinutá nad chybe definovanými osami, bolo potrebné ju po výmene osí X a Y opraviť. K oprave došlo pomocou zmeny podmienky pre zistenie osí na základe poradia videných bodov.

Nová podmienka s opačnými znamienkami rovnosti:

```
private int[] orderToFormAxes(List<Vector3D> absolute) {
    Vector3D first = absolute.get(0);
    Vector3D second = absolute.get(1);
    Vector3D third = absolute.get(2);

    if(first.getZ() == second.getZ()) {
        if (first.getY() <= second.getY()) {
            return new int[]{0, 1, 2};
        }
        return new int[]{1, 0, 2};
    }

    if(first.getZ() == third.getZ()) {
        if (first.getY() <= third.getY()) {
            return new int[]{0, 2, 1};
        }
        return new int[]{2, 0, 1};
    }

    if (second.getY() <= third.getY()) {
        return new int[]{1, 2, 0};
    }
    return new int[]{2, 1, 0};
}
```

Obr. 1 - Oprava výpočtu osi Y

Po následnom štúdiu som zhodnotil, že v tejto triede dochádza len k výpočtu rotácie. K výpočtu absolútnych súradníc by malo dochádzať v triede *AgentPositionCalculator* ktorú sme zatiaľ nepreskúmali. Po dôkladnom preskúmaní môžeme predpokladať opravu chyby posunu obrazu v prípade, že Jim vidí len rohové body.

37. Kontrola zámeny os X a Y

Bc. Ernest Loureiro

Pri testovaní hodnôt z Kalmanovho filtra sme odhalili, že dochádza k zámene osí X a Y. Podľa dokumentácie je os X od brány k bráne a os Y je umiestsnená podľa šírky ihriska. Pri strieľaní lopty podľa osi X na základe dokumentácie dochádzalo k zmene na osi Y a nie na osi X, tak ako by malo. Na základe toho sme usúdili že výpočet karteziánskych súradníc v triede *Vector3D.java*je nesprávny. Tento predpoklad sme potvrdili na základe dostupných zdrojov na internete pre prevod sférických súradníc na karteziánske.

Došlo k nasledujúcim zmenám:

- výmena X a Y súradnice pre výpočet karteziánskych súradníc
- oprava spätného prepočtu z karteziánskych súradníc na sférické
- pridanie atribútov pre uhly theta a phi v stupňoch
- výpis vektora s uhlami v stupňoch, namiesto v radiánoch

Pôvodný prevod sférických súradníc na karteziánske *(Vector3D.java):*

```
private void calculateCartesian() {
        x = cos(theta) * sin(phi) * -r;
        x = (int) (x * 1000.0) / 1000.0;
        y = cos(theta) * cos(phi) * r;
        y = (int) (y * 1000.0) / 1000.0;
        z = sin(theta) * r;
        z = (int) (z * 1000.0) / 1000.0;
}
```

Obr. 1 - Pôvodná implementácia calculateCartesian()

Nový prevod sférických súradníc na karteziánske*(Vector3D.java*):

```
private void calculateCartesian() {
    y = cos(theta) * sin(phi) * -r;
    y = (int) (y * 1000.0) / 1000.0;
    x = cos(theta) * cos(phi) * r;
    x = (int) (x * 1000.0) / 1000.0;
    z = sin(theta) * r;
    z = (int) (z * 1000.0) / 1000.0;

}
```

Obr. 2 - Nová implementácia calculateCartesian()

Pôvodný prevod karteziánskych súradníc na sférické *(Vector3D.java)*:

```
private void calculateSpherical() {
        r = sqrt(x * x + y * y + z * z);
        theta = asin(z / r);
        if (r == 0.0)
                theta = 0.0;
        phi = atan2(y, x);
        phi -= PI / 2.0d;

        phi = Angles.normalize(phi);
        theta = Angles.normalize(theta);
}
```

Obr. 37 - Pôvodná implementácia calculateSpherical()

Nový prevod karteziánskych súradníc na sférické*(Vector3D.java)*:

```
private void calculateSpherical() {
    r = sqrt(x * x + y * y + z * z);
    theta = asin(z / r);
    if (r == 0.0)
        theta = 0.0;
    phi = atan2(x, y);
    phi -= PI / 2.0d;

    phi_deg = toDegrees(phi);
    theta_deg = toDegrees(theta);

    phi = Angles.normalize(phi);
    theta = Angles.normalize(theta);
}
```

Obr. 4- Nová implementácia calculateSpherical()

Vytvorenie vektora pomocou sférických súradníc *(Vector3D.java)*:

```
public static Vector3D spherical(double r, double phi, double theta) {
    Vector3D vector = new Vector3D();
    vector.r = r;
    vector.phi = phi;
    vector.theta = theta;
    vector.theta_deg = toDegrees(theta);
    vector.phi_deg = toDegrees(phi);
    vector.calculateCartesian();
    return vector;
}
```

Obr. 5- Pridanie atribútov pre uhly v stupňoch

Výpis objektu Vector3D so súradnicami v stupňoch *(Vector3D.java):*:

```
@Override
public String toString() {
    return String
            .format("x, y, z: [%.2f, %.2f, %.2f] r, phi, theta: [%.2f, %.2f, %.2f]",
                    x, y, z, r, phi_deg , theta_deg);
}
```

Obr. 6- Formátovanie výpisu sférických súradníc

38. Refactor funkcie calculateSpherical()

Bc. Ernest Loureiro
Bc. Daniel Lukáč

- zmena if else

- aktualny stav: vypocet **theta** teraz neprebieha pri kazdom volani funkcie ale len v prípade potreby

- dôsledok: zrýchlenie kódu

- testovanie: hodnotím OK

```
/*
 * Computes spherical coordinates from cartesian coordinates.
 */
private void calculateSpherical() {
    r = sqrt(x * x + y * y + z * z);
    if (r == 0.0) {
        theta = 0.0;
    }
    else {
        try {
            theta = asin(z / r);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    phi = atan2(y, x);
    phi -= PI / 2.0d;

    phi = Angles.normalize(phi);
    theta = Angles.normalize(theta);
}
```

Obr.5 - calculateSpherical refactor Vector3D.java

39. Analýza zámeny pozície rohov a bránkoviska

Bc Ernest Loureiro, Bc. Daniel Lukáč

Pri úlohe analýza zámeny pozície rohov a bránkoviska, som zistil, že tento problém má na svedomí zrkadlová rotácia osi Y. Aj keď sa projekt po oprave zámeny súradníc X a Y zdal byť v poriadku, pri dôkladnom preskúmaní tejto problematiky som odhalil, že os Y bola zrkadlovo otočená. Priestor bránkoviska sa síce nachádzal na správnej strane, ale robot ho vnímal dole hlavou. Tento problém bol odstránený a vyriešili sa všetky predošlé problémy.

Problémom však ostáva výpočet absolútnej pozície objektov na základe relatívnych súradníc, a rotácii v závislostí od osí.

Vykonané zmeny:

RoboCupLibrary/src/sk/fiit/robocup/library/geometry/Vector3D.java

Hunk 1 : Lines 119-125

```
119 119         * Computes cartesian coordinates from spherical coordinates.
120 120         */
121 121        private void calculateCartesian() {
122         -         y = cos(theta) * sin(phi) * -r;
    122     +         y = cos(theta) * sin(phi) * r;
123 123           y = (int) (y * 1000.0) / 1000.0;
124 124           x = cos(theta) * cos(phi) * r;
125 125           x = (int) (x * 1000.0) / 1000.0;
```

Obr. 8: Zmena pri výpočte karteziánskych súradníc

Jim/src/sk/fiit/jim/agent/models/AgentRotationCalculator.java

Hunk 1 : Lines 133-152

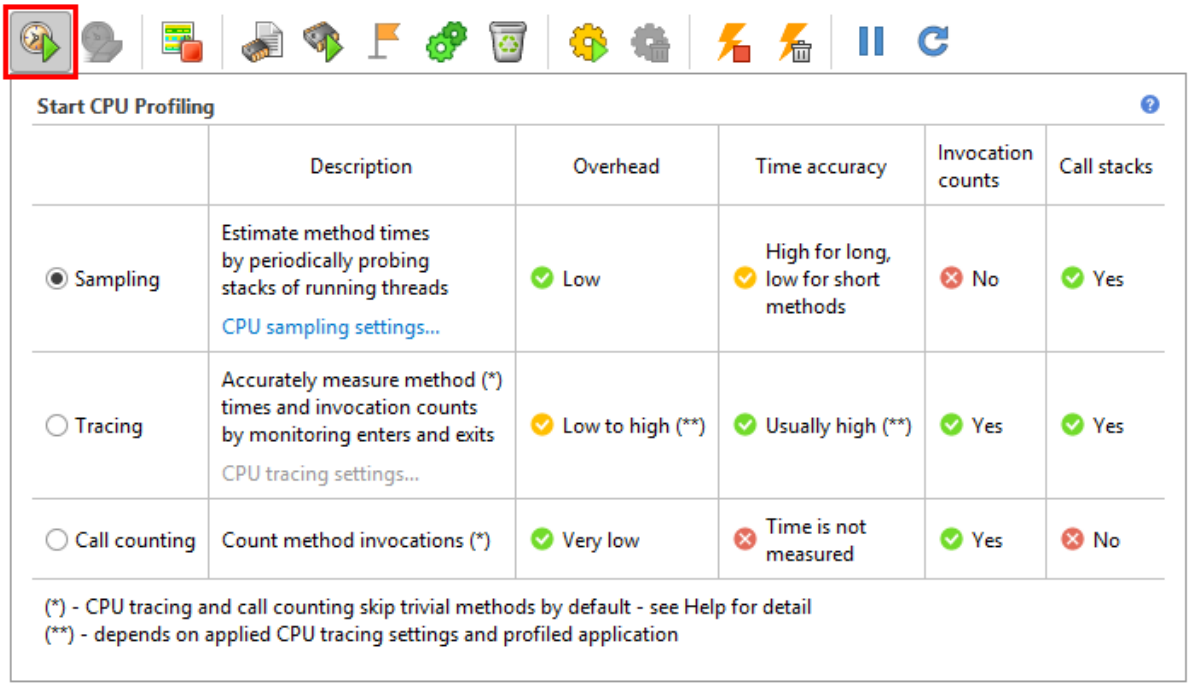
```
133 133           Vector3D third = absolute.get(2);
134 134
135 135           if(first.getZ() == second.getZ()) {
136         -             if (first.getY() <= second.getY()) {
    136     +             if (first.getY() > second.getY()) {
137 137                   return new int[]{0, 1, 2};
138 138               }
139 139               return new int[]{1, 0, 2};
140 140           }
141 141
142 142           if(first.getZ() == third.getZ()) {
143         -             if (first.getY() <= third.getY()) {
    143     +             if (first.getY() > third.getY()) {
144 144                   return new int[]{0, 2, 1};
145 145               }
146 146               return new int[]{2, 0, 1};
147 147           }
148 148
149         -         if (second.getY() <= third.getY()) {
    149     +         if (second.getY() > third.getY()) {
150 150               return new int[]{1, 2, 0};
151 151           }
152 152           return new int[]{2, 1, 0};
```

Obr. 9: Zmena naspäť pre vytvorenie osí

40. Yourkit pre Eclipse

Cpu Profiling:

Poznáme 3 typy CPU profilingu: Sampling, Tracing, Call Counting, ktoré si môžeme zvoliť pri spúšťaní profileru v eclipse.



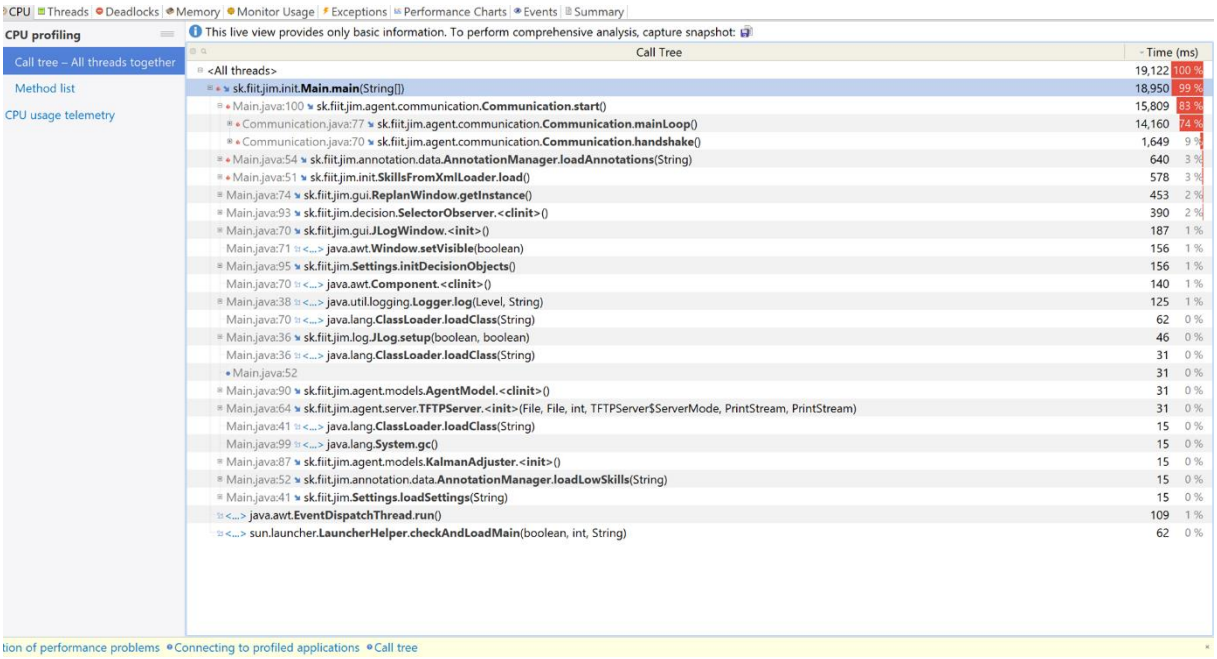
Start CPU Profiling

	Description	Overhead	Time accuracy	Invocation counts	Call stacks
(•) Sampling	Estimate method times by periodically probing stacks of running threads CPU sampling settings...	Low	High for long, low for short methods	No	Yes
( ) Tracing	Accurately measure method (*) times and invocation counts by monitoring enters and exits CPU tracing settings...	Low to high (**)	Usually high (**)	Yes	Yes
( ) Call counting	Count method invocations (*)	Very low	Time is not measured	Yes	No

(*) - CPU tracing and call counting skip trivial methods by default - see Help for detail
(**) - depends on applied CPU tracing settings and profiled application

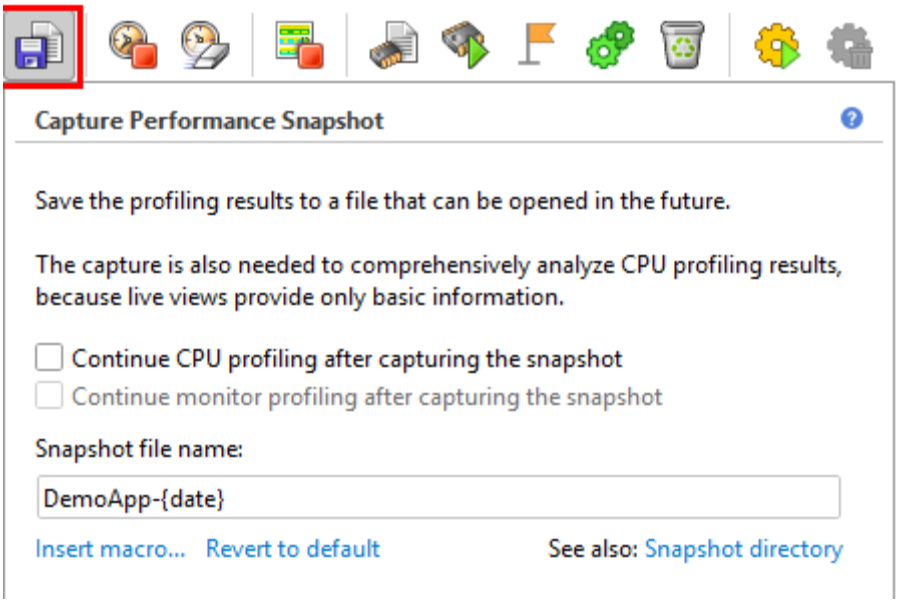
Obr. 1 Sampling, Tracing, Call Counting

V momente ako je zavolaný CPU profiling môžeme výsledky vidieť v tzv. Call Tree, ktorý môžeme vidieť na nasledujúcom obrázku.



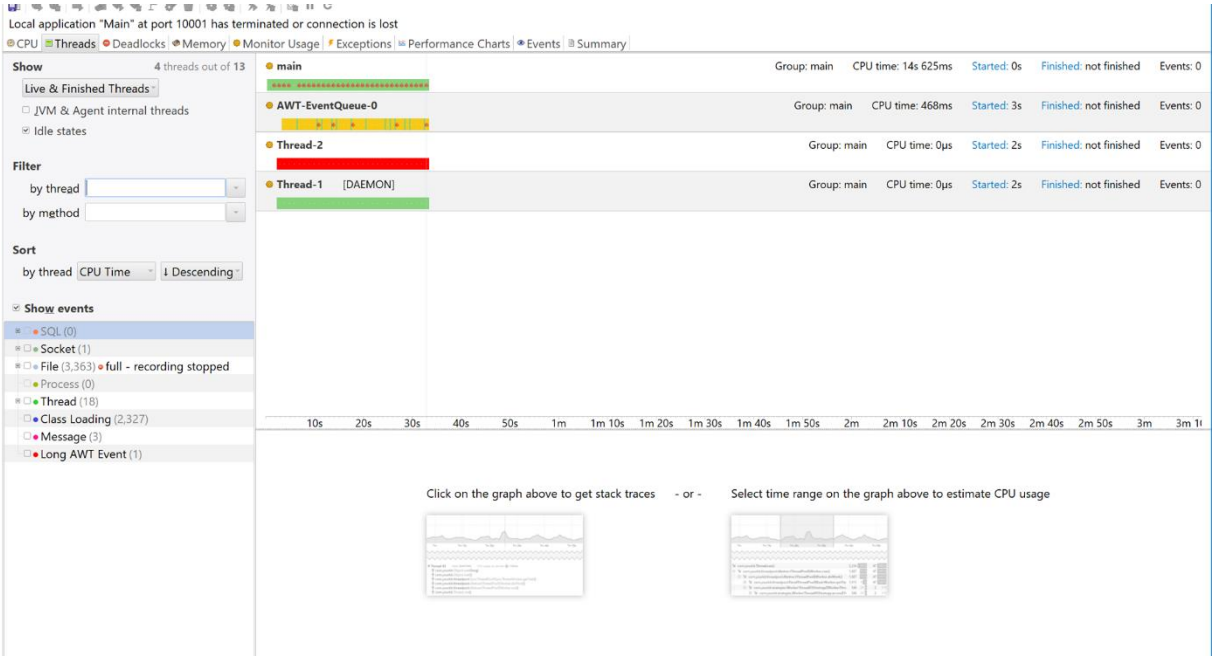
Obr. 2 CPU profiling

Ak úloha, ktorú sme naplánovali, skončila (alebo bola vykonaná dostatočne dlho),môžeme vytvoriť snapshots CPU so všetkými zaznamenanými informáciami spôsobom, ktorý je zaznamenaný na nasledujúcom obrázku.



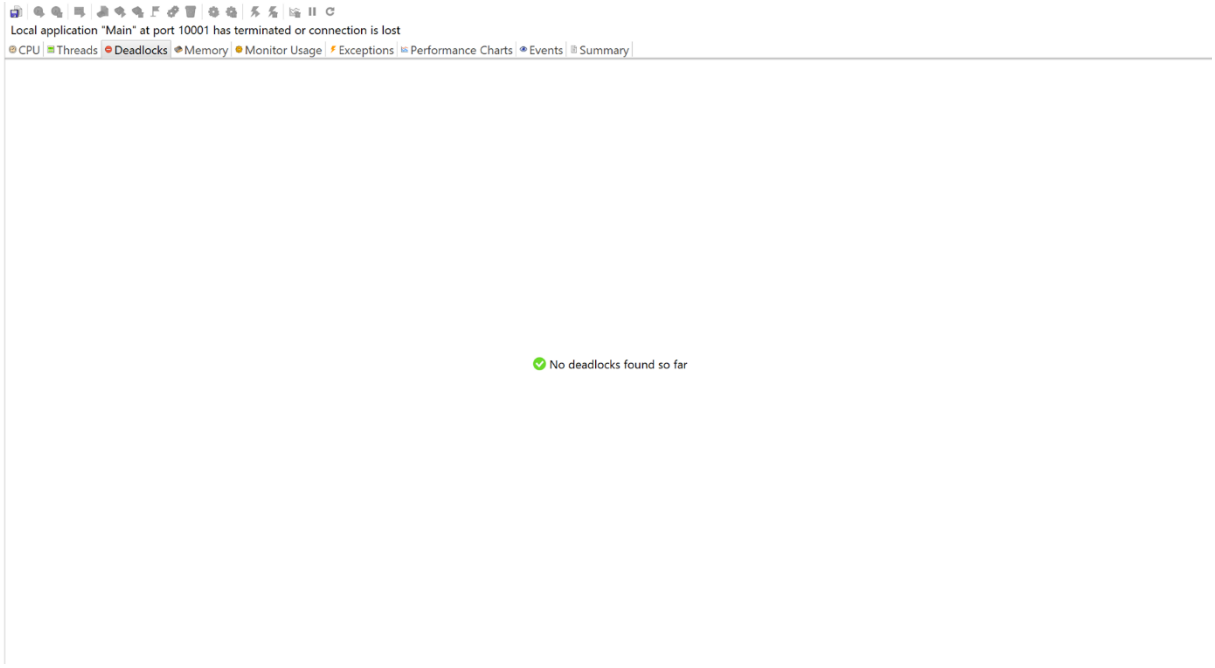
Obr. 3Snapshot

V aplikáci si taktiež môžeme pozrieť rozdelenie threadov.



Obr. 4 Threads

Taktiež máme možnosť zistiť, či vrámci programu vzniká nejaký deadlock.

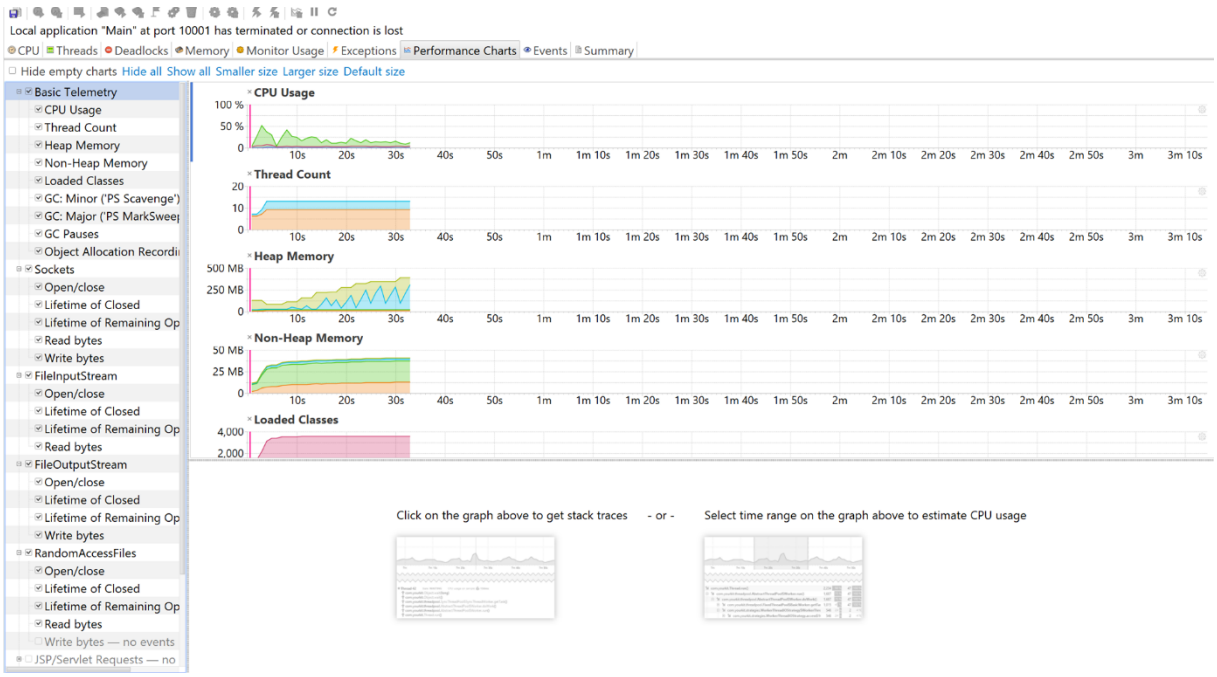


Obr. 5 Deadlock

Jedno z ďalších vymožeností je taktiež, funkcionalita, kde si môžeme pozrieť ako prebieha alokácia pamäte, kedy a ako sa pamäť využíva.

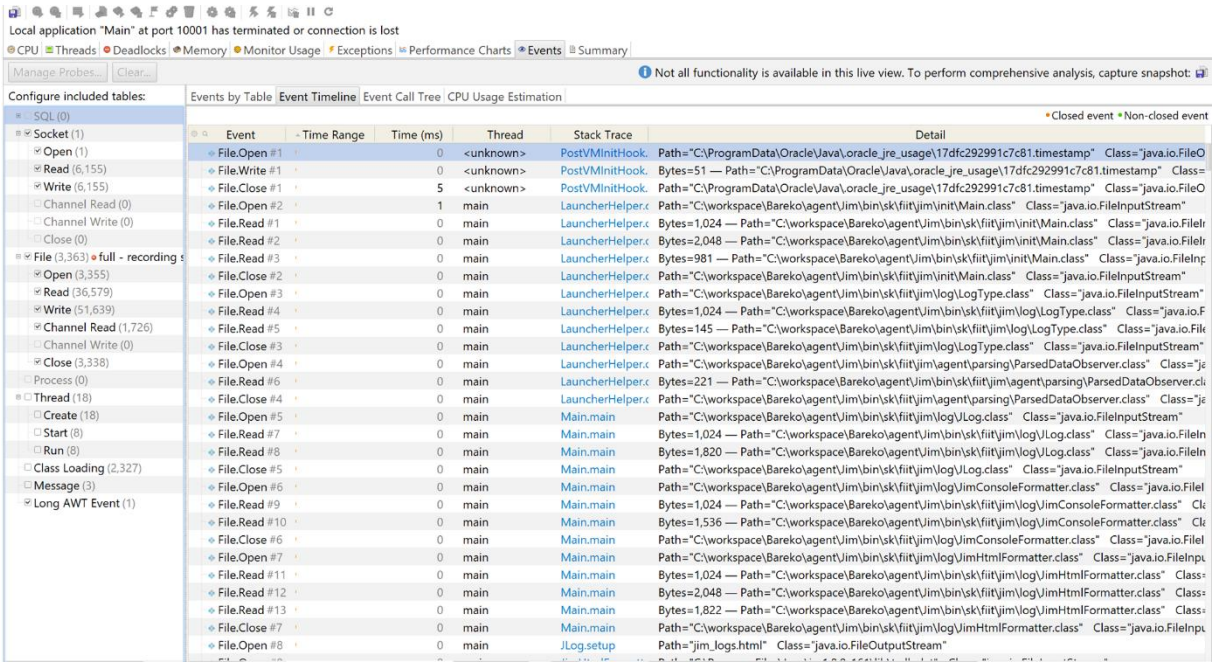
Obrázok 6 Memory

Taktiež si vrámci aplikácie môžeme zobraziť tzv. Performance Charts na základe, ktorých máme celkový prehľad o využití CPU, Threadov, Heap memory, Non-heap memory a taktiež o počte lodanutých tried.



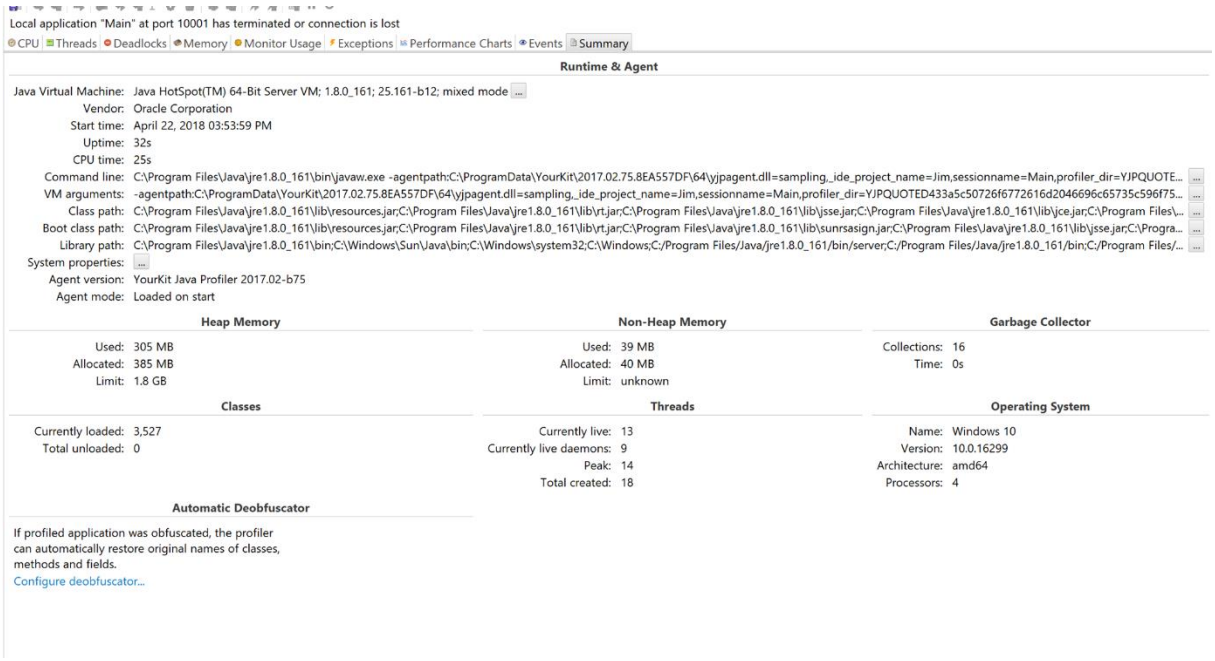
Obrázok 7 Performance Charts

Sekcia events nám zobrazuje, postup eventov v akom poradí sa vykonávajú na akom threade sa vykonávajú a rôzne iné detaily.



Obrázok 8 Events

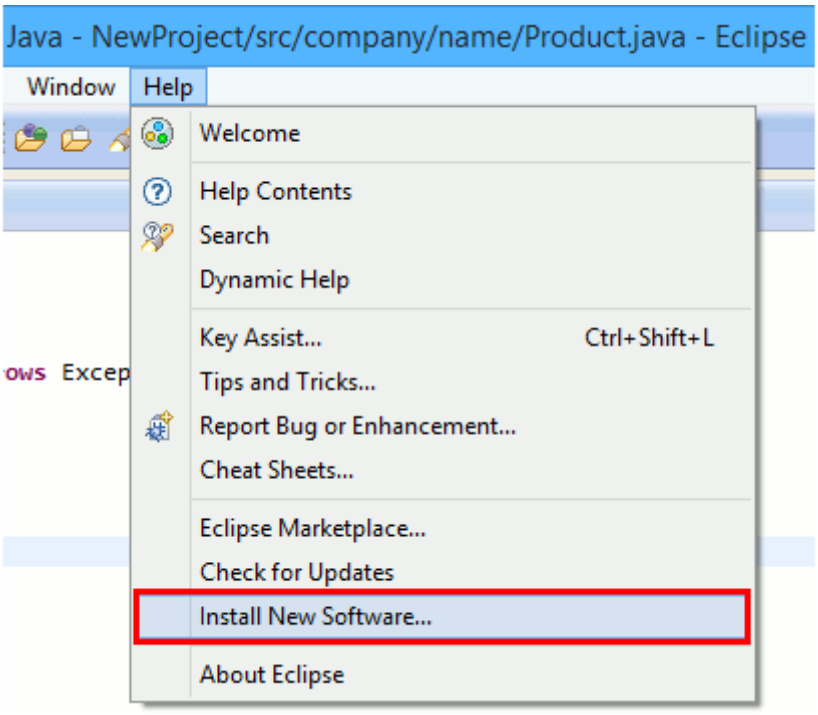
Sekcia summary, nám udáva základný všeobecný prehľad.



Obrázok 9 Summary

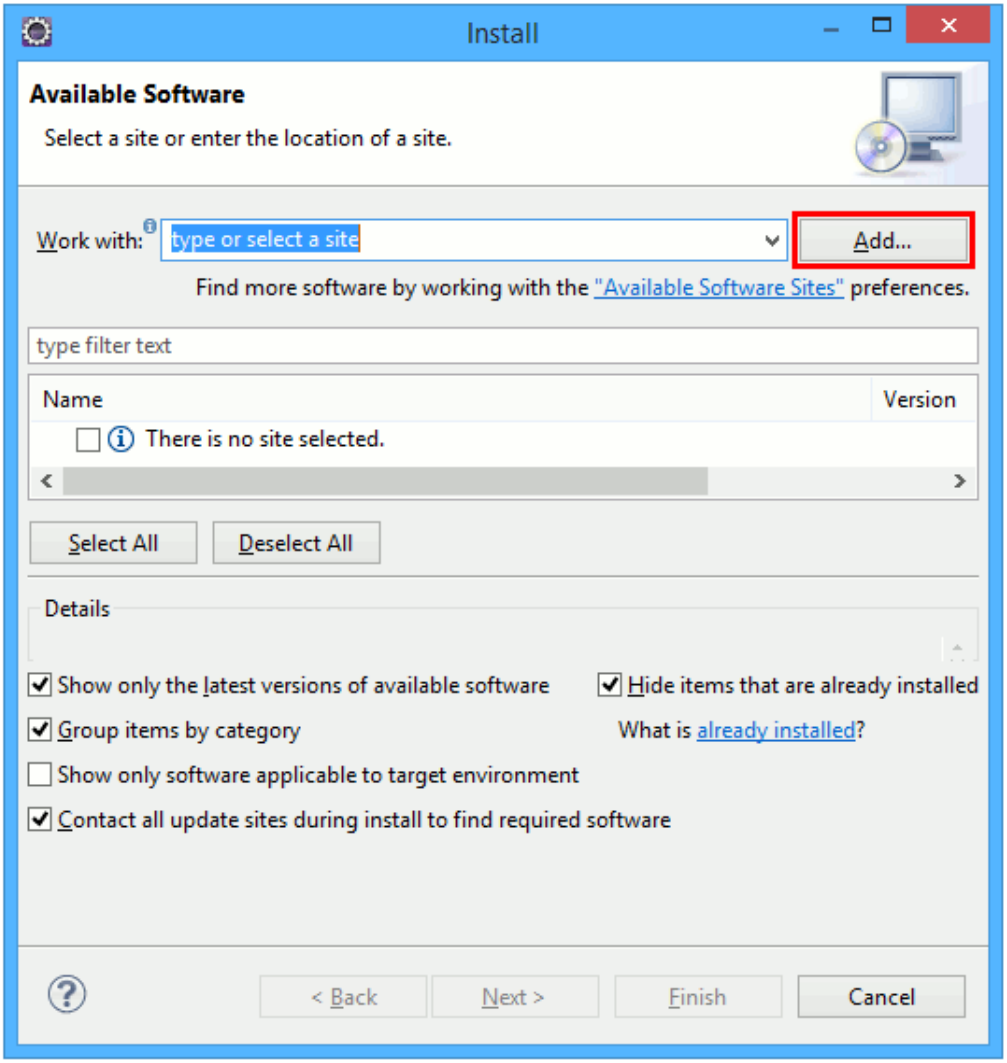
40.1 Inštalácia YourKit do Eclipse

V prvom rade je potrebné otvoriť si Eclipse, kde následne vyberiete v toolbare Help a zakliknete možnosť Install New Software.



Obrázok 11 Insta New Software

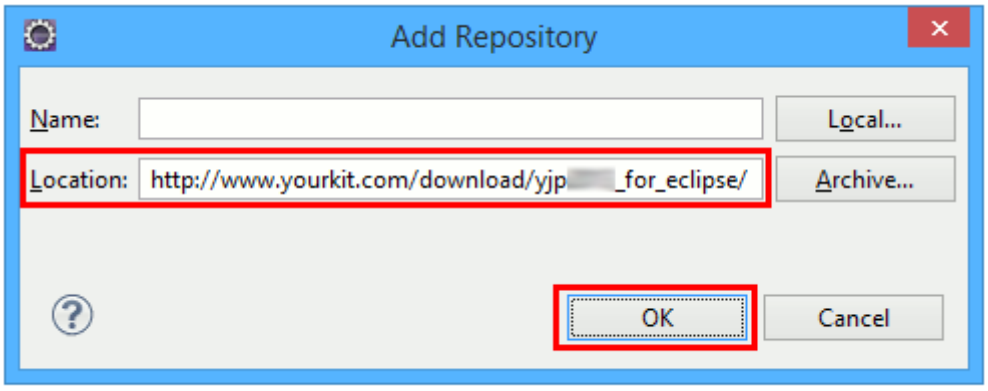
Následne vyberiete možnosť Add ako na nasledujúcom obrázku.



Obrázok 12 ADD

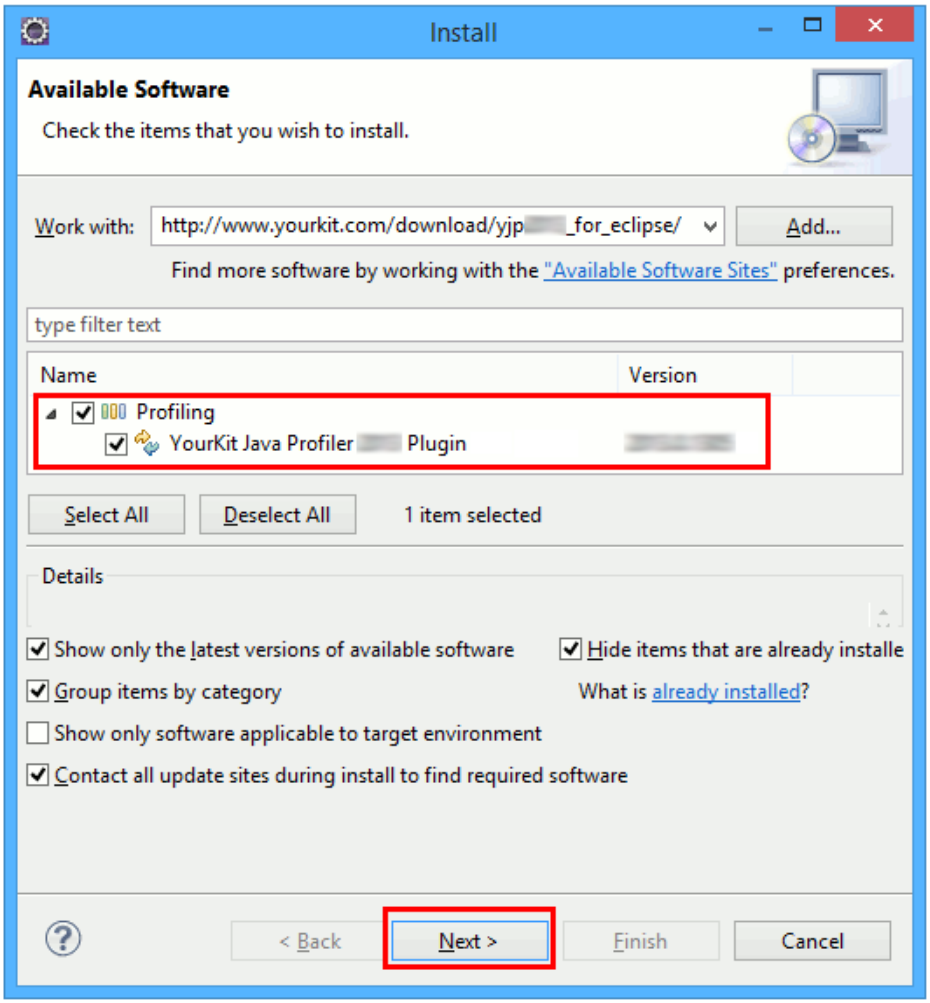
Do sekcie Location vložite vhodnú URL.

https://www.yourkit.com/download/yjp2017_02_for_eclipse/ A zvolíte možnosť OK.



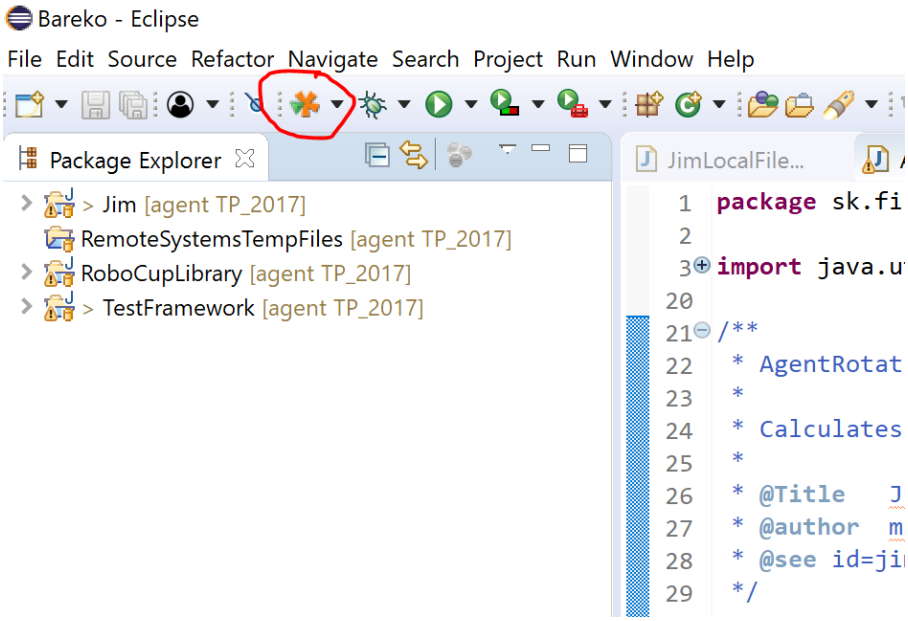
Obrázok 13 Link

Následne zakliknete, že chcete Profiling a YourKit Java Profiler.



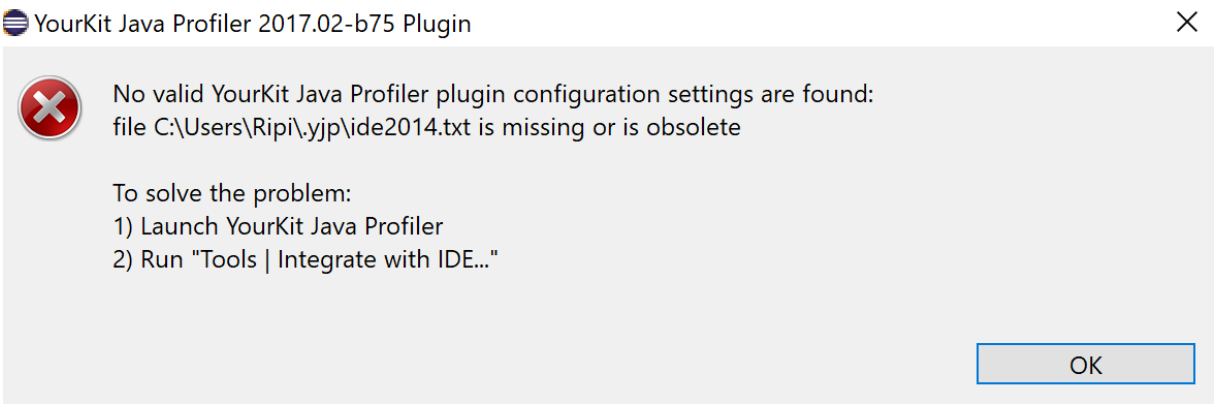
Obrázok 14 Yourkit

Po zakliknutí možnosti Next sa tento tool v eclipse začne inštalovať a následne vás eclipse požiada o reštartovanie. Po reštartovaní sa vám v eclipse toolbare zobrazí YourKit.



Obrázok 15 Yourkit v Eclipse

Potom ako budete chcieť spustiť tento tool sa vám zobrazí táto chyba.



Obrázok 16 Chyba

Následne budete musieť stiahnuť tool YourKit z ich oficálnej stránky z jedného z príslušných linkov.

URL: https://www.yourkit.com/

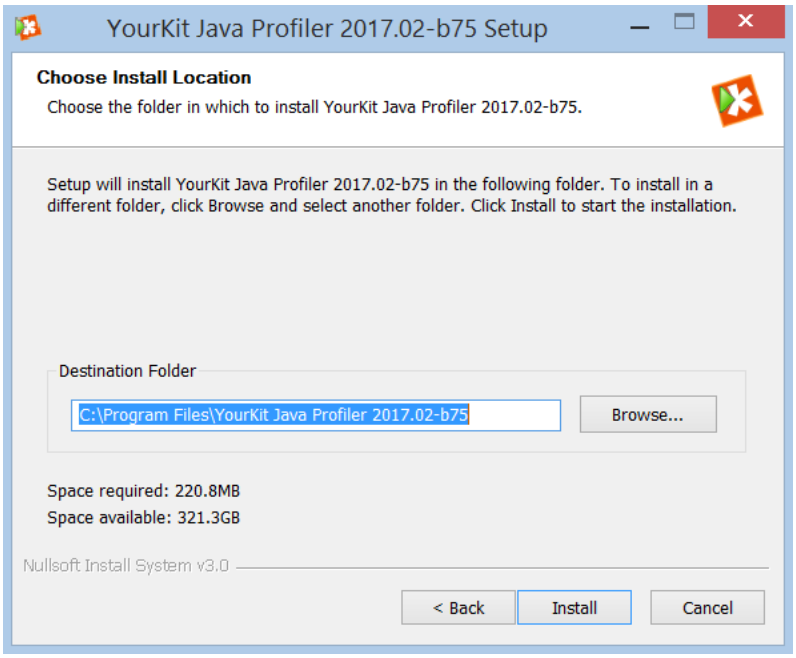
Na stiahnutie

Windows x86, x64: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75.exe

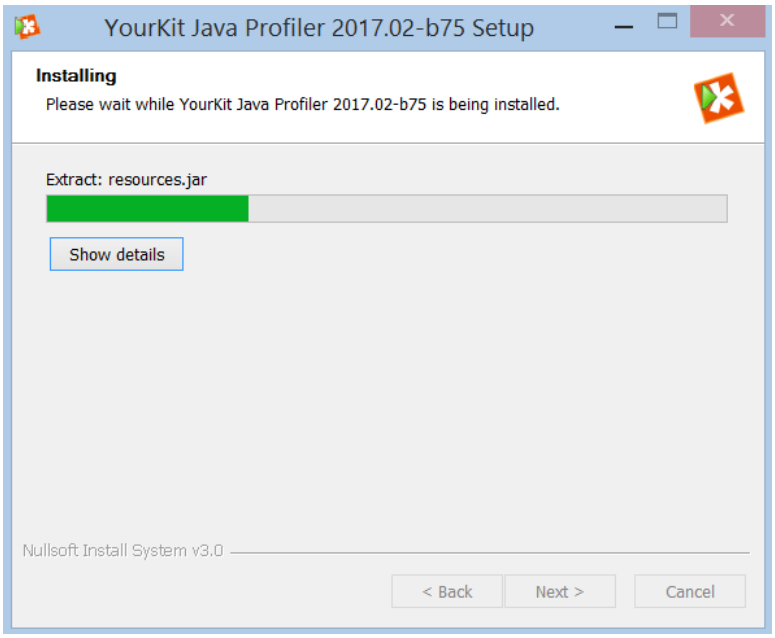
macOS Intel: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75-mac.zip

Linux: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75.zip

Obrázok 17 - inštalácia 1



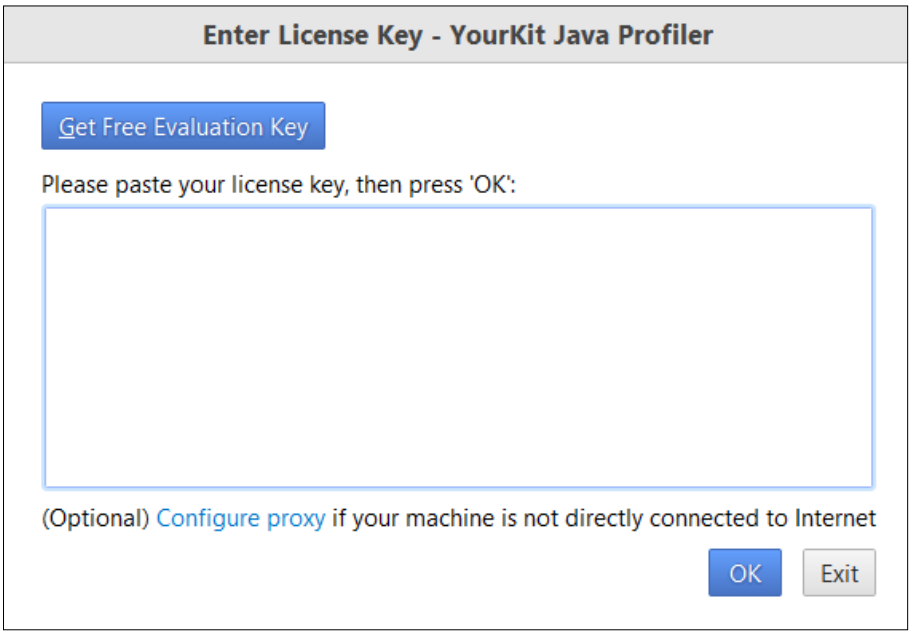
Obrázok 18 - inštalácia 2



Obrázok 19 - inštalácia 3

40.1.1 Integrácia YourKit-u do Eclipse

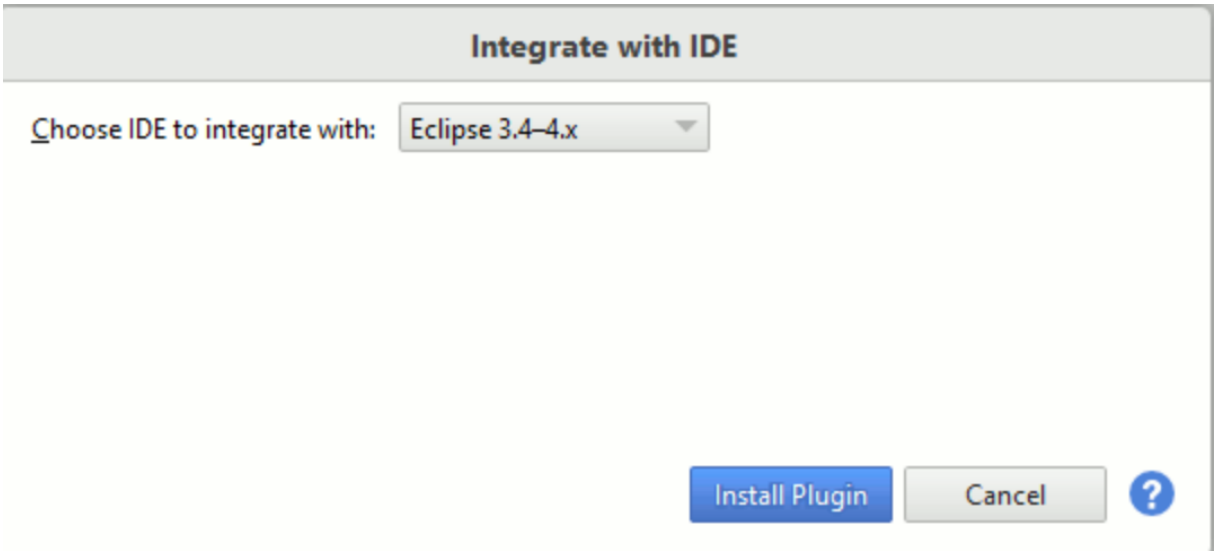
Po úspešnom nainštalovaní sa spustí aplikácia. V úvode vyskočí dialógové okno s možnosťou zadania licenčného kľúča. Licenčný klúč môžete získať bud ako trial verziu alebo si môžete zohnať licenciu v prípade, že napíšete priamo tímu yourkitu, že by ste potrebovali využiť ich nástroj na nekomerčné účely.



Obrázok 20 - licenčný kľúč

Po zadaní licenčného kľúča je potrebné pripojiť Yourkit k vášmu IDE. Dialógové okno vyskočí automaticky po zadaní kľúča. Zvolíte jednu z možností, v našom prípade sme mali pri inštaláciách

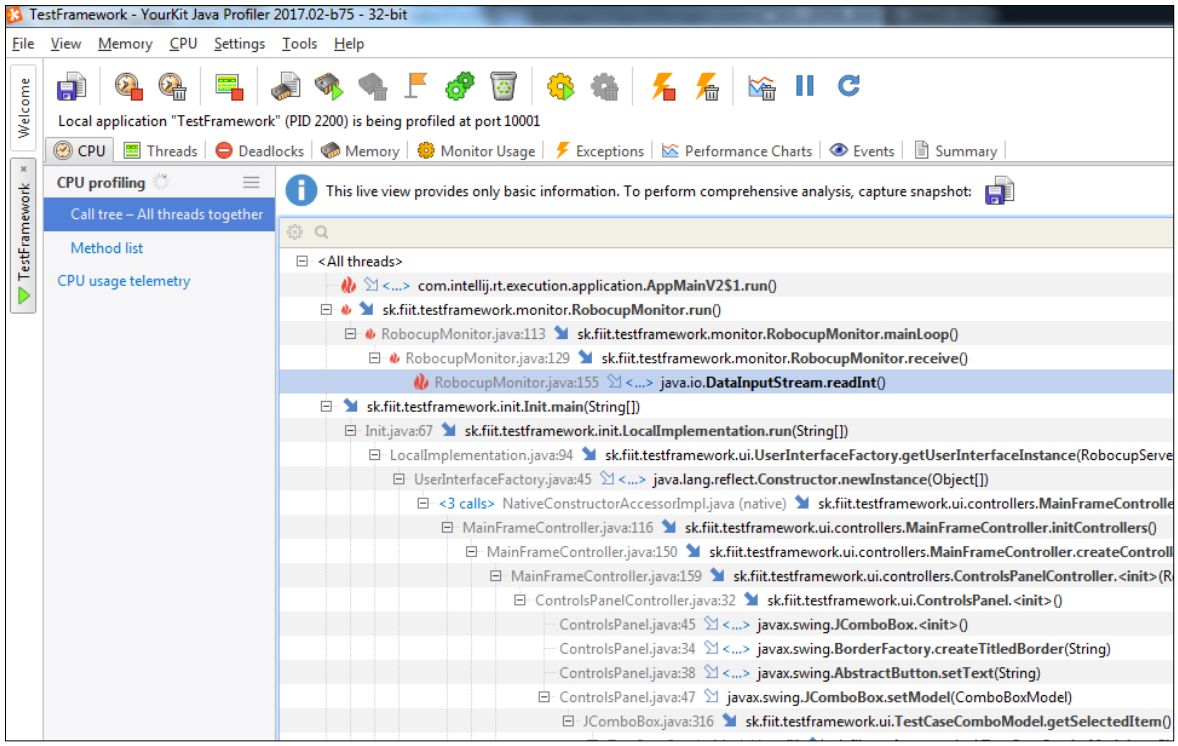
nastavované defaultné priečinky aj pre IntelliJ IDE a nepoužívali sme žiadne dodatočné pluginy.



Obrázok 21 - integrácia s ECLIPSE

Po úspešnej integrácii Yourkitu do Eclipse vyriešíme chybu a následne môžeme kliknúť na logo zobrazené na obrázku 5.

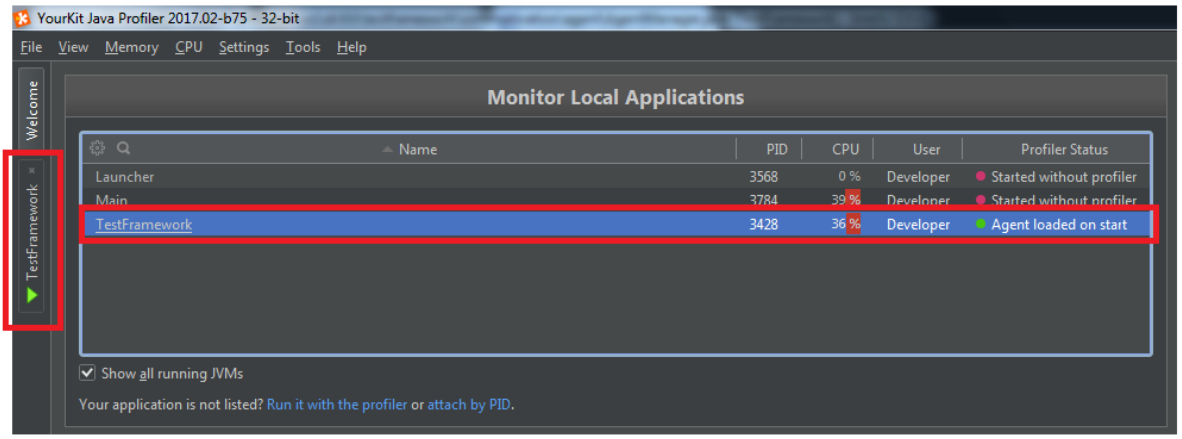
Po úspešnom spustení TestFrameworku sa otvorí aplikácia YourKit, ktorá je namapovaná na náš projekt. V ponuke je viacero možností sledovania performance, opísané sú v príručke používania.



Obrázok 22 - sledovanie performance TestFrameworku

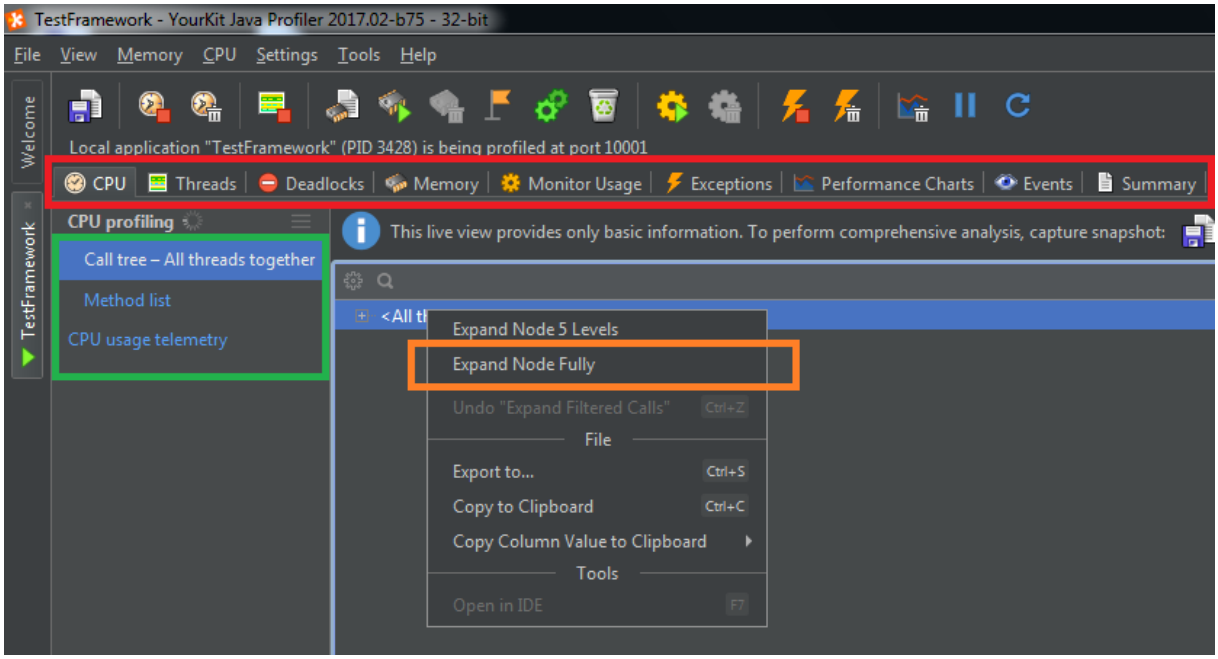
41. YourKit pre IntelliJ IDEA

1. Pri spustení projektu je vidieť viacero aplikácií, ktoré je možné pomocou Yourkitu analyzovať. Dvojklikom na náš TestFramework sa vytvorí další TAB s monitorovaním (v ľavo).



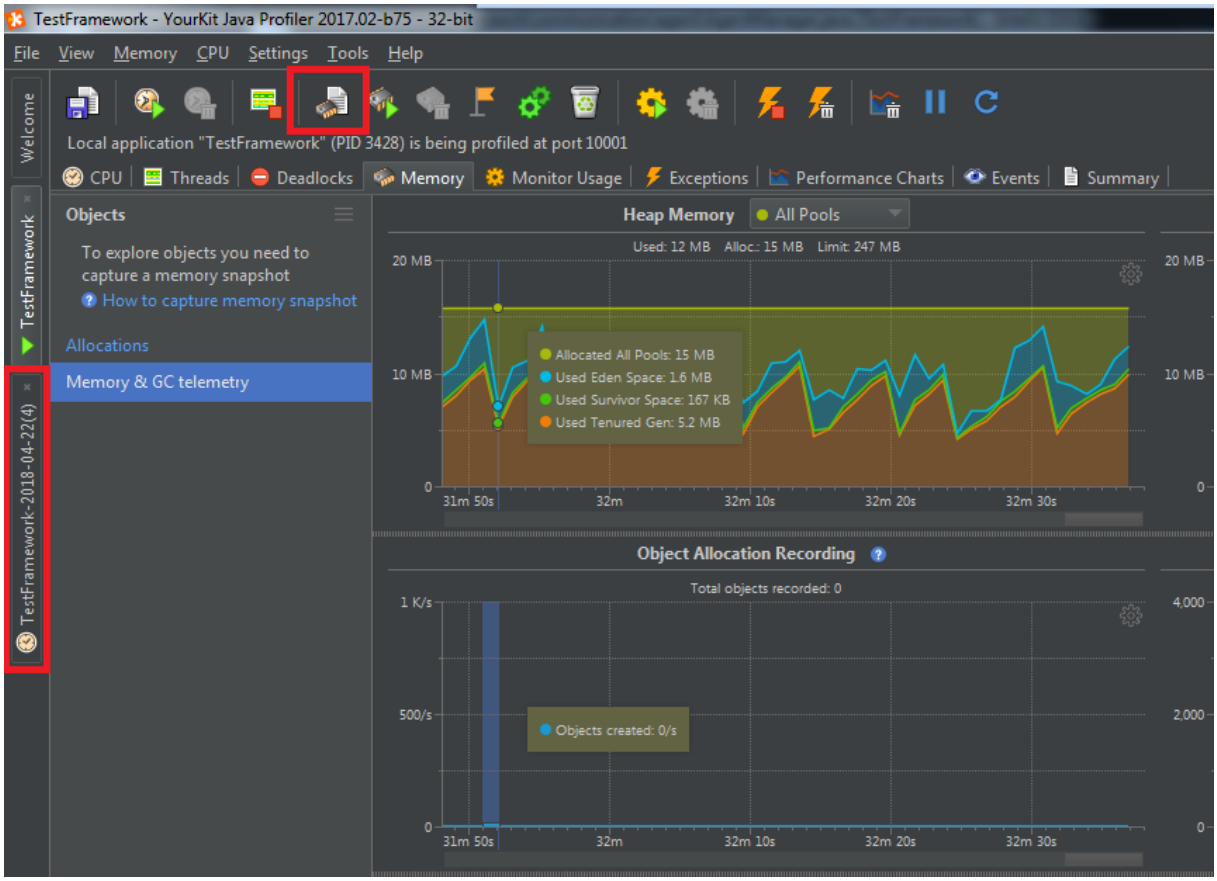
Obr.23 - všetky aplikácie

2. Zobrazí sa monitorovacie menu pre projekt. V červenom obdĺžniku je zobrazené rozdelenie monitorovania na jednotlivé typy. V zelenom obdĺžniku je zobrazené menu pre výber zobrazenia všetkých threadov a ich jednotlivých metód alebo samostatne len metód. Pomocou funkcie expand node fully sa rozbalí automaticky celý strom.



Obr.24 - monitorovanie základ

3. Existuje možnosť vytvorenia snapshotu aktuálneho stavu merania. Vytvorí sa nový súbor, ktorý sa zobrazí opäť v bočných taboch a je možné ho analyzovať v offline režime aplikácie.

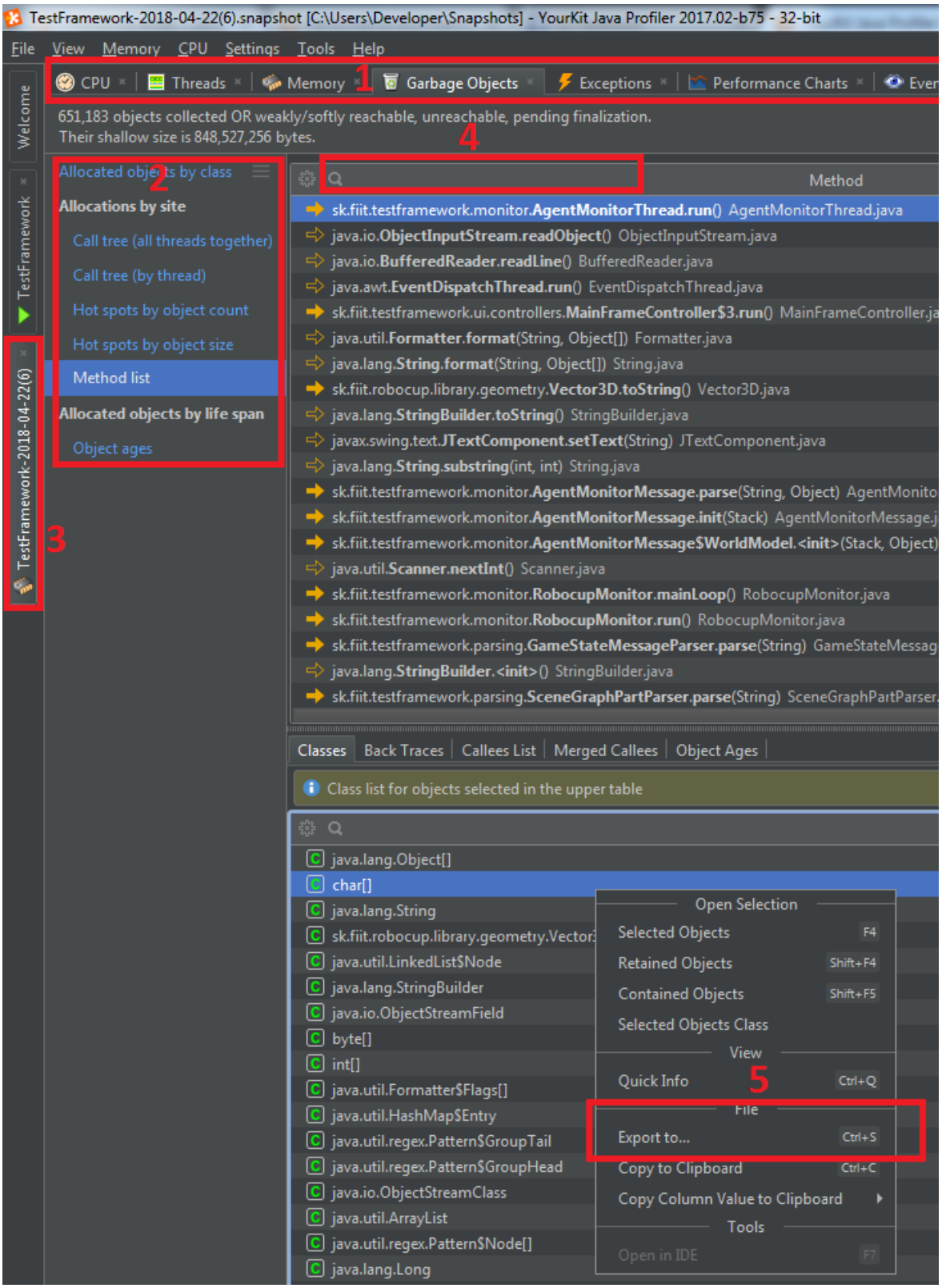


Obr.25 - snapshot

4. Pre zaznamenanie Garbage collectora pre aplikáciu je potrebné pred snapshotom spustiť Object allocation recording.

Obr.26 - allocation recording

5. Pri analýze snapshotu môžeme pracovať s následovným:
   - 1 - panel pre prepínanie rôznych monitorovacích módov
   - 2 - v rámci niektorého monitorovacieho módu je možné zobrazovať rôzne štruktúry (thready, metódy, triedy...) a sledovať ich využitie
   - 3 - tab snapshotu
   - 4 - pole pre vyhľadávanie
   - 5 - je možnosť exportu monitorovacieho módu napríklad do .csv súboru



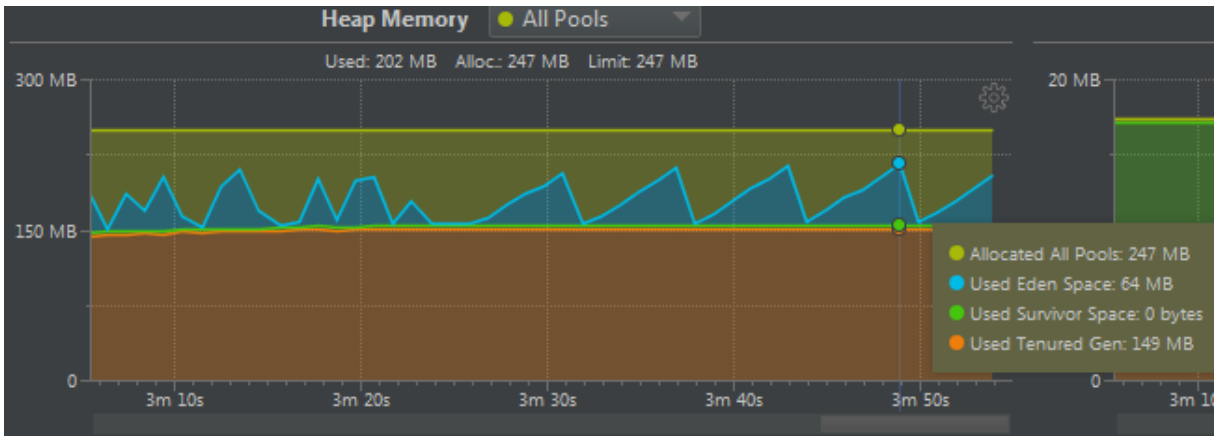
Obr.27 - práca so snapshotom

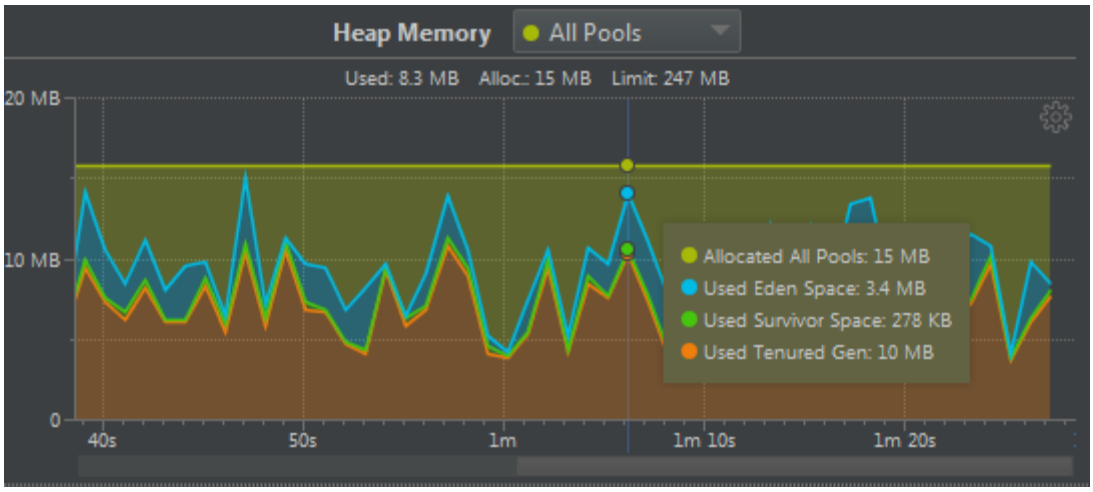
6. Prehľad performance a jednotlivých tried je možné zobraziť v grafoch.

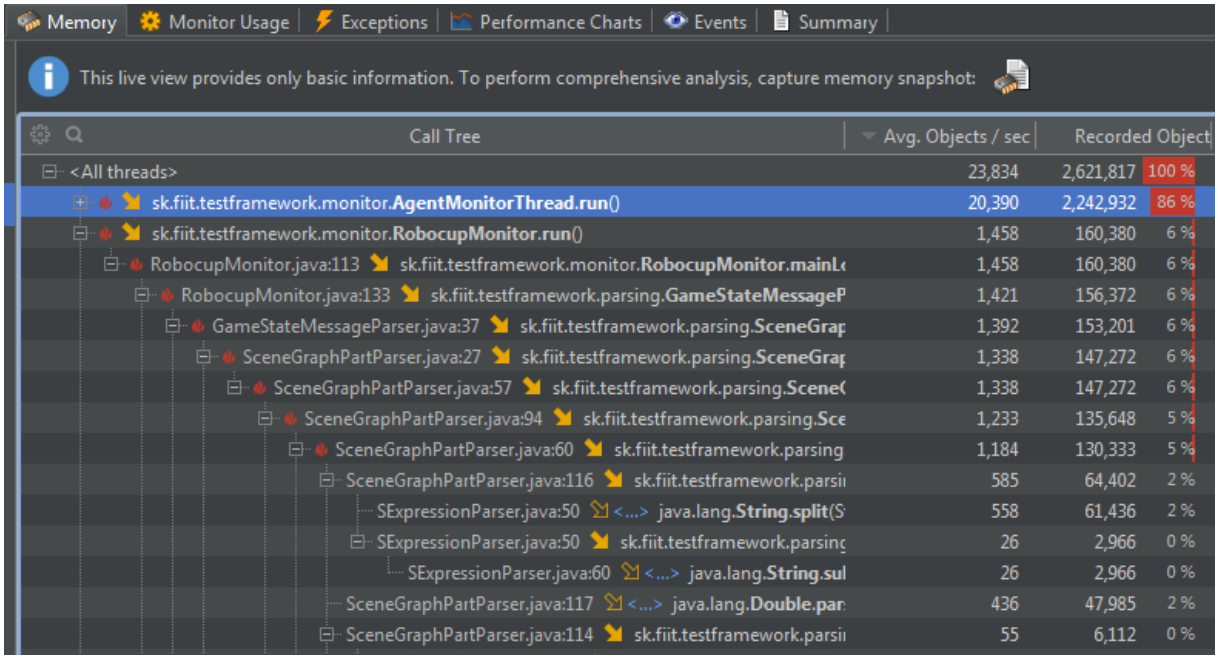
Obr.28 - performance

Zaujímavosti:

- sledovanie exceptions
- detekcia deadlockov
- grafy performance
- základný sumár
- sledovanie memor
- sledovanie CPU
- rozdelenia úrovní
    - na základe threadov
    - na základe metód
    - na základe tried
    - na základe packagov
    - celková stromová štruktúra
- export do .csv
- grabage collection hodnoty




Heap Memory
All Pools
Used: 8.3 MB Alloc.: 15 MB Limit: 247 MB
20 MB
10 MB
0
40s
50s
1m
1m 10s
1m 20s
Allocated All Pools: 15 MB
Used Eden Space: 3.4 MB
Used Survivor Space: 278 KB
Used Tenured Gen: 10 MB


Memory
Monitor Usage
Exceptions
Performance Charts
Events
Summary
This live view provides only basic information. To perform comprehensive analysis, capture memory snapshot:
Call Tree
Avg. Objects / sec
Recorded Object
<All threads> 23,834 2,621,817 100 %
sk.fiit.testframework.monitor.AgentMonitorThread.run() 20,390 2,242,932 86 %
sk.fiit.testframework.monitor.RobocupMonitor.run() 1,458 160,380 6 %
RobocupMonitor.java:113 sk.fiit.testframework.monitor.RobocupMonitor.mainL 1,458 160,380 6 %
RobocupMonitor.java:133 sk.fiit.testframework.parsing.GameStateMessageP 1,421 156,372 6 %
GameStateMessageParser.java:37 sk.fiit.testframework.parsing.SceneGrap 1,392 153,201 6 %
SceneGraphPartParser.java:27 sk.fiit.testframework.parsing.SceneGrap 1,338 147,272 6 %
SceneGraphPartParser.java:57 sk.fiit.testframework.parsing.Scene(1,338 147,272 6 %
SceneGraphPartParser.java:94 sk.fiit.testframework.parsing.Sce 1,233 135,648 5 %
SceneGraphPartParser.java:60 sk.fiit.testframework.parsing 1,184 130,333 5 %
SceneGraphPartParser.java:116 sk.fiit.testframework.parsi 585 64,402 2 %
SExpressionParser.java:50 <...> java.lang.String.split(S 558 61,436 2 %
SExpressionParser.java:50 sk.fiit.testframework.parsing 26 2,966 0 %
SExpressionParser.java:60 <...> java.lang.String.sul 26 2,966 0 %
SceneGraphPartParser.java:117 <...> java.lang.Double.par 436 47,985 2 %
SceneGraphPartParser.java:114 sk.fiit.testframework.parsi 55 6,112 0 %

41.1 Inštalácia YourKit do IntelliJ

URL: https://www.yourkit.com/

Na stiahnutie

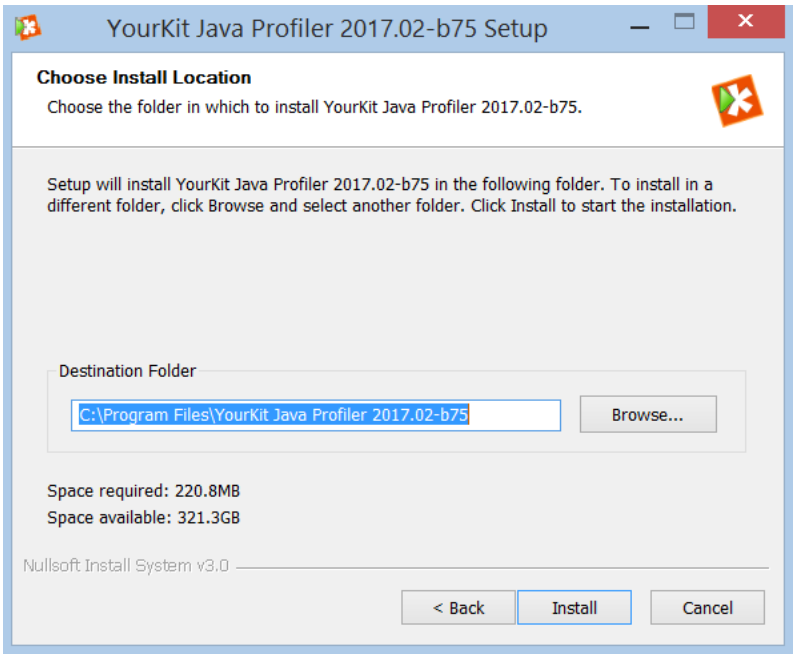
Windows x86, x64: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75.exe

macOS Intel: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75-mac.zip

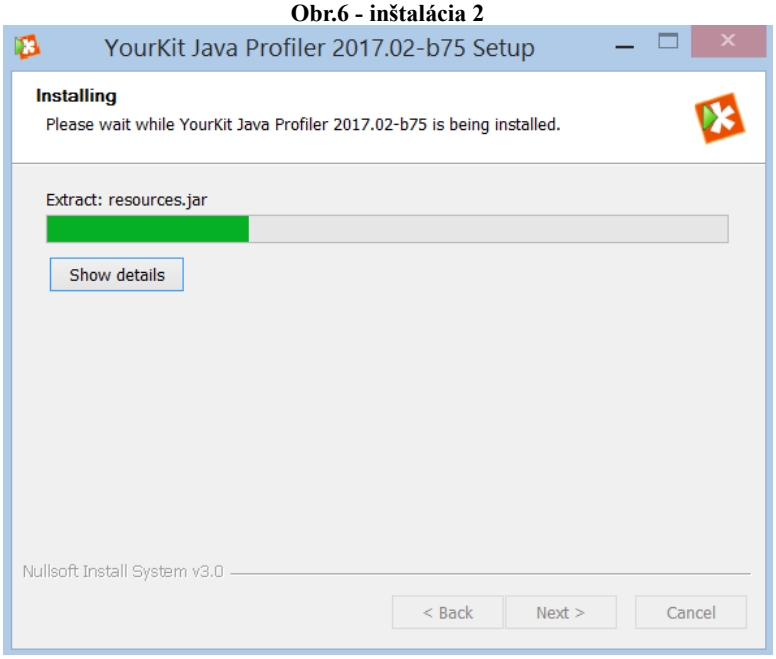
Linux: https://www.yourkit.com/download/YourKit-JavaProfiler-2017.02-b75.zip

Poznámka: Odporúčam nainštalovať Yourkit do default priečinka v Program files. Na virtuálnom stroji Windows 7, spúšťaný cez VMware, nastali pri spúšťaní problémy s cestou k projektu. Inštalácia prebieha ako každá iná.

Obr.29 - inštalácia 1



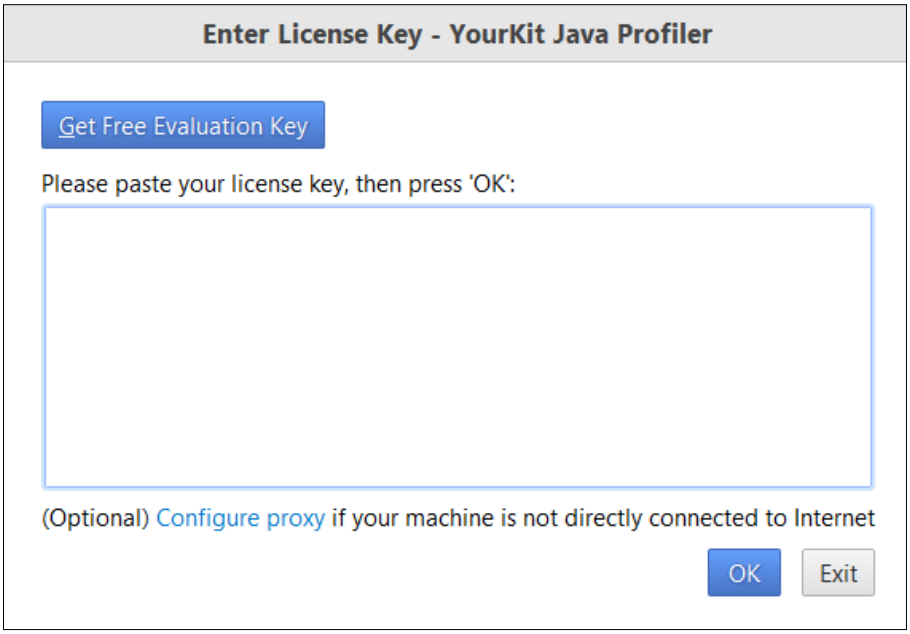
Obr.6 - inštalácia 2



Obr.30 - inštalácia 3

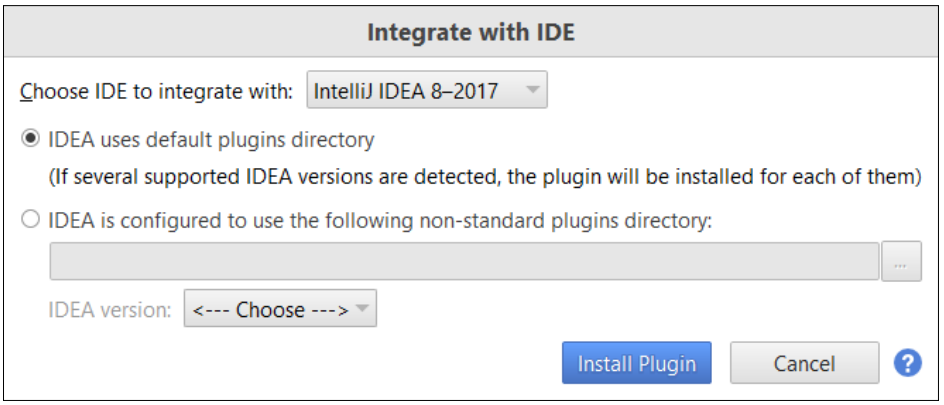
41.1.1 Integrácia YourKit-u do IntelliJ IDEA

Po úspešnom nainštalovaní sa spustí aplikácia. V úvode vyskočí dialógové okno s možnosťou zadania licenčného kľúča. Bolo vybavených 10 licencií pre projekt RoboCup FIIT STU (zodpovedný Bc. Loureiro Ernest). Licencie sú platné 1 rok od apríla 2018.



Obr.313 - licenčný kľúč

Po zadaní licenčného kľúča je potrebné pripojiť Yourkit k vášmu IDE. Dialógové okno vyskočí automaticky po zadaní kľúča. Zvolíte jednu z možností, v našom prípade sme mali pri inštaláciách nastavované defaultné priečinky aj pre IntelliJ IDE a nepoužívali sme žiadne dodatočné pluginy.

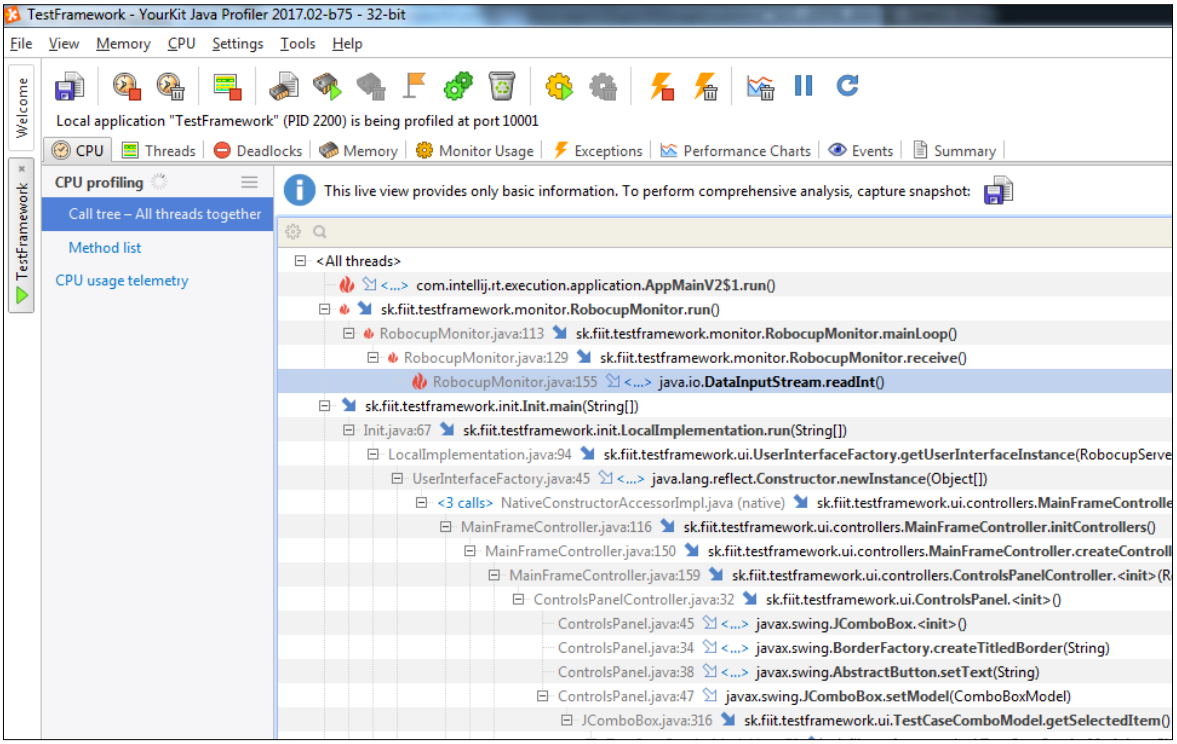


Obr.32 - integrácia s IntelliJ IDE

Po úspešnej integrácii Yourkitu do IDE otvoríme IntelliJ IDE a v menu bare uvidíme novú ikonu pre spustenie, ktorá obsahuje logo Yourkitu. V prípade potreby jeho použitia spúšťame TestFramework pomocou tohto.

Obr.33 - spustenie TestFrameworku cez Yourkit

Po úspešnom spustení TestFrameworku sa otvorí aplikácia YourKit, ktorá je namapovaná na náš projekt. V ponuke je viacero možností sledovania performance, opísané sú v príručke používania.



Obr. 34 - sledovanie performance TestFrameworku

42. Testovania scenára true pre debug taktiku bez logov

Počiatočný stav bez pridaného a zapnutého Jima:

Heap memory:

- 15 MB alokovanej
- 4 MB využitej

Non - Heap memory

- 15 MB alokovanej
- 14,6 MB využitej

Garbage collection

- 1/s

Garbage collection pauses

- striedanie 0ms s 14ms

Classes

- 2693 načítaných

Object allocation recording

- 60 K/s

CPU usage telemetry

- CPU TIME user + kernel 10-15% cca
- CPU TIME kernel 4-7% cca
- Time spent on GC 0-1%

Threads

- 16 threadov z toho 11 daemon
- hodnota state sa nepredvídateľne menila (blocked, waiting, waiting in native, runnable)

Stav počas spustenej taktiky:

Heap memory:

- 15 MB alokovanej
- 11 MB využitej

Non - Heap memory

- 17 MB alokovanej
- 16,1 MB využitej

Garbage collection

- 15/s

Garbage collection pauses

- priemer 4-7 ms s maximálnymi hodnotami až po 41 ms

Classes

- 2909 načítaných

Object allocation recording

- 200-250 K/s

CPU usage telemetry

- CPU TIME user + kernel 40-65% cca
- CPU TIME kernel 5-11% cca
- Time spent on GC 4-9%

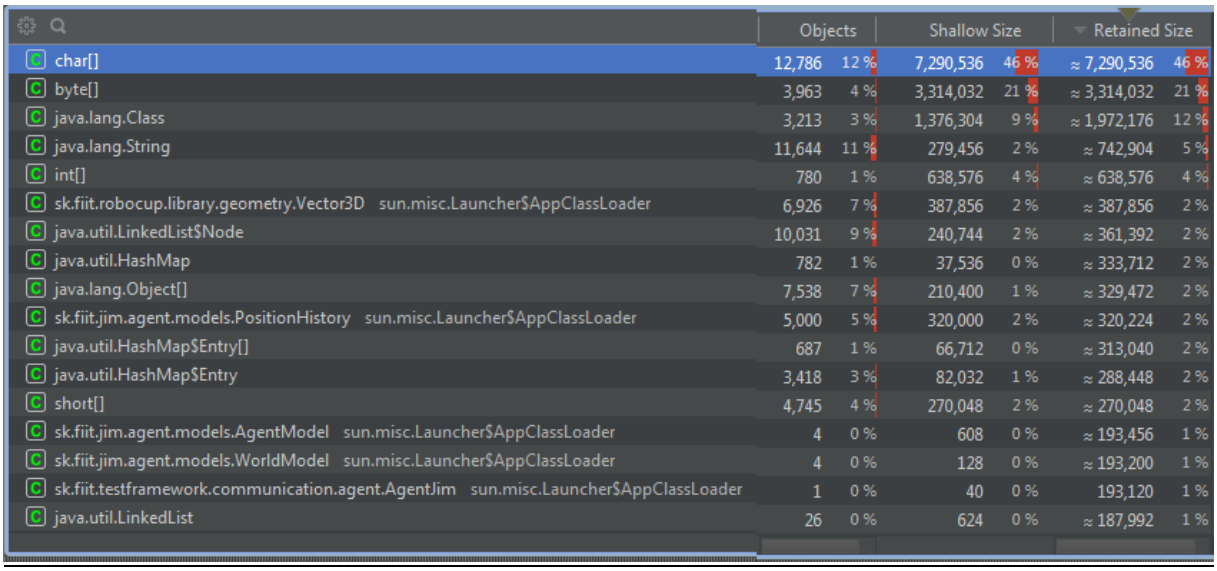
Threads

- 18 threadov z toho 12 daemon
- hodnota state sa nepredvídateľne menila (blocked, waiting, waiting in native, runnable)

Zistiť čo je

- used eden space
- used survivor space
- used tenured gen
- used perm gen
- use code cahe

Hodnoty memory pre class:



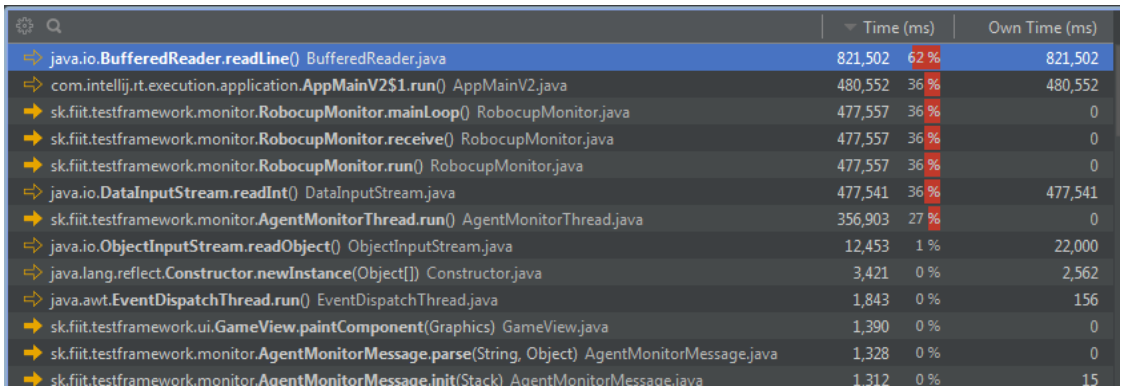
	Objects		Shallow Size		Retained Size	
char[]	12,786	12 %	7,290,536	46 %	≈ 7,290,536	46 %
byte[]	3,963	4 %	3,314,032	21 %	≈ 3,314,032	21 %
java.lang.Class	3,213	3 %	1,376,304	9 %	≈ 1,972,176	12 %
java.lang.String	11,644	11 %	279,456	2 %	≈ 742,904	5 %
int[]	780	1 %	638,576	4 %	≈ 638,576	4 %
sk.fiit.robocup.library.geometry.Vector3D sun.misc.Launcher$AppClassLoader	6,926	7 %	387,856	2 %	≈ 387,856	2 %
java.util.LinkedList$Node	10,031	9 %	240,744	2 %	≈ 361,392	2 %
java.util.HashMap	782	1 %	37,536	0 %	≈ 333,712	2 %
java.lang.Object[]	7,538	7 %	210,400	1 %	≈ 329,472	2 %
sk.fiit.jim.agent.models.PositionHistory sun.misc.Launcher$AppClassLoader	5,000	5 %	320,000	2 %	≈ 320,224	2 %
java.util.HashMap$Entry[]	687	1 %	66,712	0 %	≈ 313,040	2 %
java.util.HashMap$Entry	3,418	3 %	82,032	1 %	≈ 288,448	2 %
short[]	4,745	4 %	270,048	2 %	≈ 270,048	2 %
sk.fiit.jim.agent.models.AgentModel sun.misc.Launcher$AppClassLoader	4	0 %	608	0 %	≈ 193,456	1 %
sk.fiit.jim.agent.models.WorldModel sun.misc.Launcher$AppClassLoader	4	0 %	128	0 %	≈ 193,200	1 %
sk.fiit.testframework.communication.agent.AgentJim sun.misc.Launcher$AppClassLoader	1	0 %	40	0 %	193,120	1 %
java.util.LinkedList	26	0 %	624	0 %	≈ 187,992	1 %

Obr.1: Hodnoty memory pre jednotlivé classy

Class	Shallow size / one
byte[]	836
int[]	818
char[]	570
Class	428
PositionHistory	64
Vector3D	56
short[]	56

Tab.1: Hodnoty memory pre jednu jednotku classy

CPU hodnoty pre metódy:



	Time (ms)		Own Time (ms)
java.io.**BufferedReader.readLine**() BufferedReader.java	821,502	62 %	821,502
com.intellij.rt.execution.application.**AppMainV2$1.run**() AppMainV2.java	480,552	36 %	480,552
sk.fiit.testframework.monitor.**RobocupMonitor.mainLoop**() RobocupMonitor.java	477,557	36 %	0
sk.fiit.testframework.monitor.**RobocupMonitor.receive**() RobocupMonitor.java	477,557	36 %	0
sk.fiit.testframework.monitor.**RobocupMonitor.run**() RobocupMonitor.java	477,557	36 %	0
java.io.**DataInputStream.readInt**() DataInputStream.java	477,541	36 %	477,541
sk.fiit.testframework.monitor.**AgentMonitorThread.run**() AgentMonitorThread.java	356,903	27 %	0
java.io.**ObjectInputStream.readObject**() ObjectInputStream.java	12,453	1 %	22,000
java.lang.reflect.**Constructor.newInstance**(Object[]) Constructor.java	3,421	0 %	2,562
java.awt.**EventDispatchThread.run**() EventDispatchThread.java	1,843	0 %	156
sk.fiit.testframework.ui.**GameView.paintComponent**(Graphics) GameView.java	1,390	0 %	0
sk.fiit.testframework.monitor.**AgentMonitorMessage.parse**(String, Object) AgentMonitorMessage.java	1,328	0 %	0
sk.fiit.testframework.monitor.**AgentMonitorMessage.init**(Stack) AgentMonitorMessage.java	1,312	0 %	15

Obr.2: CPU vyťaženie jednotlivých metód

43. Testovanie scenára false pre debug taktiku s logmi

OS: windows 10 version 10.0.16299 amd64 proccesors 4

Vývojové prostredie: Eclipse

Nástroj: Yourkit

43.1 Stav pred spustením taktiky

Heap memory:

- 72 mb allocated
- 40 mb used
- Limit 1.8 GB

Non – Heap memory:

- 38mb allocated
- 37mb used
- Limit unknown

Garbage collection:

- 1/s
- Collections: 282
- Time: 0s

Classes:

- Currently loaded: 3486

CPU usage telemetry

- CPU time (user + kernel): 14% - 47%
- CPU time (kernel): 0%- 3%
- Time spent in GC: 0%

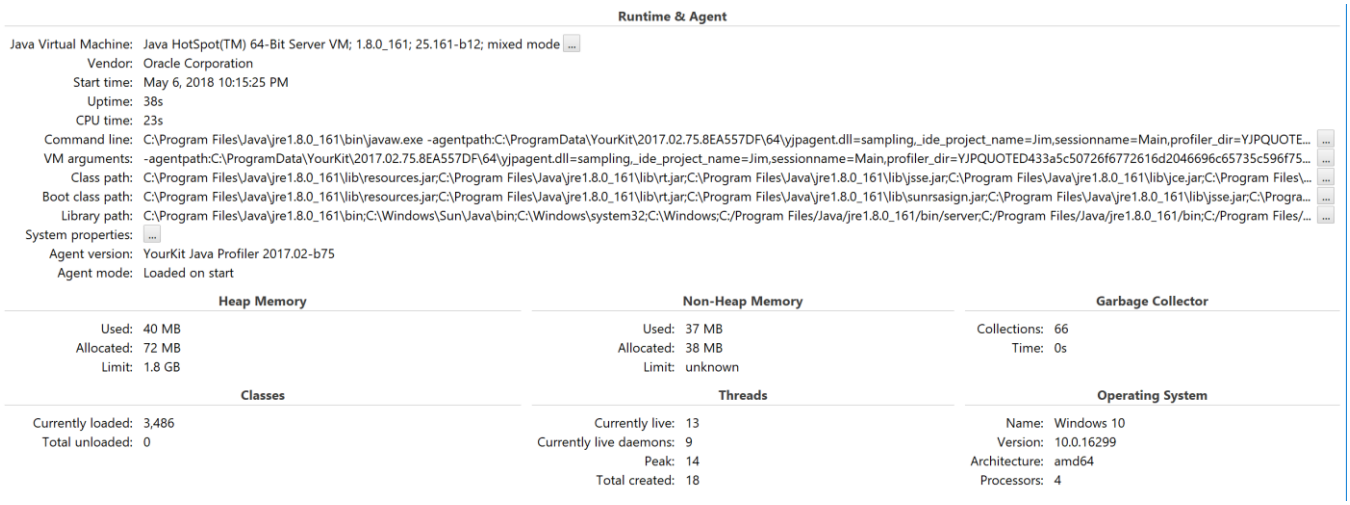
Threads

- 13 threads
- 9 daemon threads
- 14 peak
- Total created 18

43.2 Stav počas spustenej taktiky

1. **Heap memory:**
    a. 61-73mb allocated
    b. 36 mb used
    c. Limit 1.8 GB
2. **Non – Heap memory:**
    a. 41mb allocated
    b. 40mb used
    c. Limit unknown
3. **Garbage collection:**
    a. 1/s
    b. Collections: 642
    c. Time: 1s
4. **Classes:**
    a. Currently loaded: 3540
5. **CPU usage telemetry**
    a. CPU time (user + kernel): 14% - 47%
    b. CPU time (kernel): 0%- 3%
    c. Time spent in GC: 0%
6. **Threads**
    a. 13 threads
    b. 9 daemon threads
    c. 14 peak
    d. Total created 18

43.3 Summary pred a po spustení



Runtime & Agent

Java Virtual Machine:	Java HotSpot(TM) 64-Bit Server VM; 1.8.0_161; 25.161-b12; mixed mode
Vendor:	Oracle Corporation
Start time:	May 6, 2018 10:15:25 PM
Uptime:	38s
CPU time:	23s
Command line:	C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe -agentpath:C:\ProgramData\YourKit\2017.02.75.8EA557DF\64\yjpagent.dll=sampling,_ide_project_name=Jim,sessionname=Main,profiler_dir=YJPQUOTE...
VM arguments:	-agentpath:C:\ProgramData\YourKit\2017.02.75.8EA557DF\64\yjpagent.dll=sampling,_ide_project_name=Jim,sessionname=Main,profiler_dir=YJPQUOTED433a5c50726f6772616d2046696c65735c596f75...
Class path:	C:\Program Files\Java\jre1.8.0_161\lib\resources.jar;C:\Program Files\Java\jre1.8.0_161\lib\rt.jar;C:\Program Files\Java\jre1.8.0_161\lib\jsse.jar;C:\Program Files\Java\jre1.8.0_161\lib\jce.jar;C:\Program Files\...
Boot class path:	C:\Program Files\Java\jre1.8.0_161\lib\resources.jar;C:\Program Files\Java\jre1.8.0_161\lib\rt.jar;C:\Program Files\Java\jre1.8.0_161\lib\sunrsasign.jar;C:\Program Files\Java\jre1.8.0_161\lib\jsse.jar;C:\Progra...
Library path:	C:\Program Files\Java\jre1.8.0_161\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:/Program Files/Java/jre1.8.0_161/bin/server;C:/Program Files/Java/jre1.8.0_161/bin;C:/Program Files/...
System properties:	
Agent version:	YourKit Java Profiler 2017.02-b75
Agent mode:	Loaded on start

Heap Memory

Used:	40 MB
Allocated:	72 MB
Limit:	1.8 GB

Non-Heap Memory

Used:	37 MB
Allocated:	38 MB
Limit:	unknown

Garbage Collector

Collections:	66
Time:	0s

Classes

Currently loaded:	3,486
Total unloaded:	0

Threads

Currently live:	13
Currently live daemons:	9
Peak:	14
Total created:	18

Operating System

Name:	Windows 10
Version:	10.0.16299
Architecture:	amd64
Processors:	4

Obrázok 7 Summary Pred

Runtime & Agent

Java Virtual Machine:	Java HotSpot(TM) 64-Bit Server VM; 1.8.0_161; 25.161-b12; mixed mode
Vendor:	Oracle Corporation
Start time:	May 6, 2018 10:15:25 PM
Uptime:	3m 34s
CPU time:	1m 17s
Command line:	C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe -agentpath:C:\ProgramData\YourKit\2017.02.75.8EA557DF\64\yjpagent.dll=sampling,_ide_project_name=Jim,sessionname=Main,profiler_dir=YJPQUOTE...
VM arguments:	-agentpath:C:\ProgramData\YourKit\2017.02.75.8EA557DF\64\yjpagent.dll=sampling,_ide_project_name=Jim,sessionname=Main,profiler_dir=YJPQUOTED433a5c50726f6772616d2046696c65735c596f75...
Class path:	C:\Program Files\Java\jre1.8.0_161\lib\resources.jar;C:\Program Files\Java\jre1.8.0_161\lib\rt.jar;C:\Program Files\Java\jre1.8.0_161\lib\jsse.jar;C:\Program Files\Java\jre1.8.0_161\lib\jce.jar;C:\Program Files\...
Boot class path:	C:\Program Files\Java\jre1.8.0_161\lib\resources.jar;C:\Program Files\Java\jre1.8.0_161\lib\rt.jar;C:\Program Files\Java\jre1.8.0_161\lib\sunrsasign.jar;C:\Program Files\Java\jre1.8.0_161\lib\jsse.jar;C:\Progra...
Library path:	C:\Program Files\Java\jre1.8.0_161\bin;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;C:/Program Files/Java/jre1.8.0_161/bin/server;C:/Program Files/Java/jre1.8.0_161/bin;C:/Program Files/...
System properties:	
Agent version:	YourKit Java Profiler 2017.02-b75
Agent mode:	Loaded on start

Heap Memory

Used:	41 MB
Allocated:	72 MB
Limit:	1.8 GB

Non-Heap Memory

Used:	40 MB
Allocated:	41 MB
Limit:	unknown

Garbage Collector

Collections:	619
Time:	1s

Classes

Currently loaded:	3,540
Total unloaded:	0

Threads

Currently live:	13
Currently live daemons:	9
Peak:	14
Total created:	18

Operating System

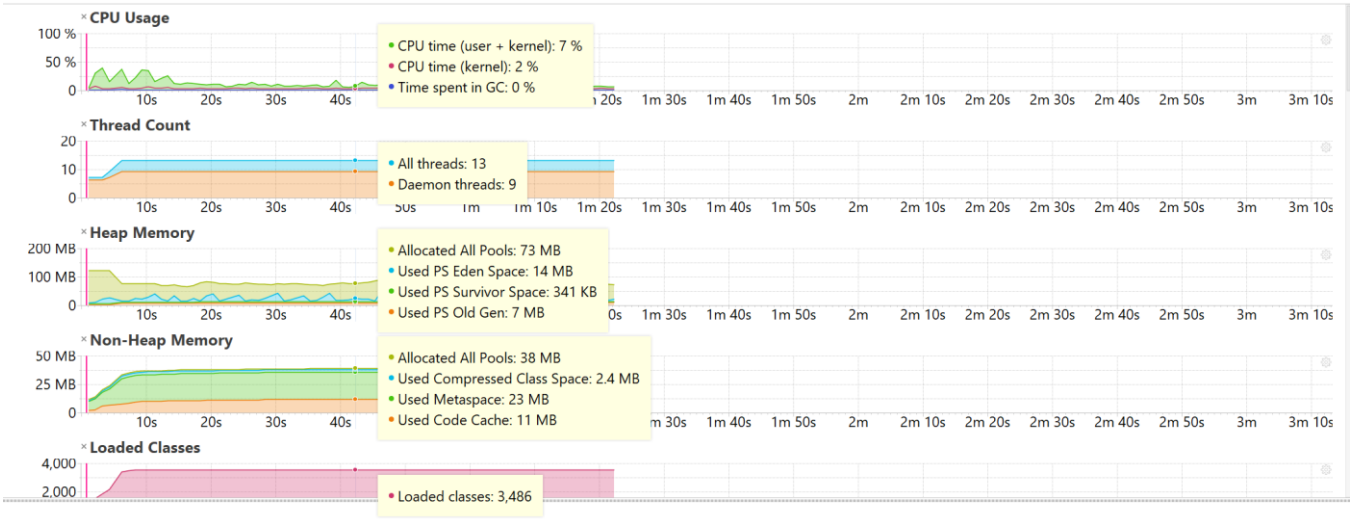
Name:	Windows 10
Version:	10.0.16299
Architecture:	amd64
Processors:	4

Automatic Deobfuscator

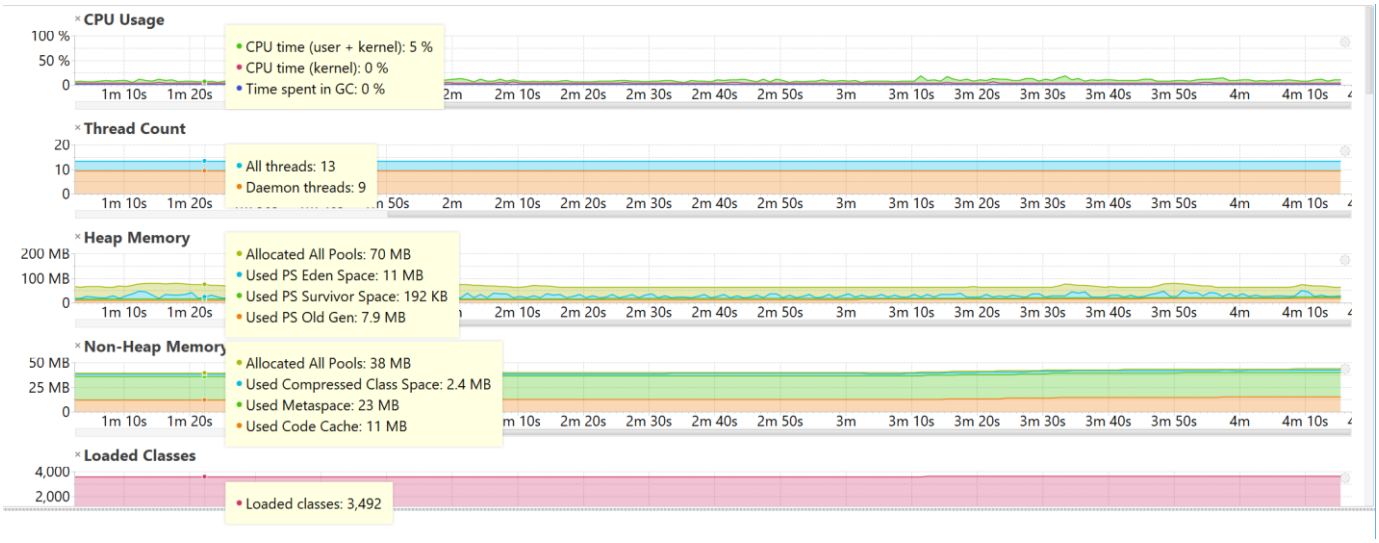
Obrázok 8 Summary Po

43.4 Performance Chart pred spustením vs:

Performance Chart po spustení:

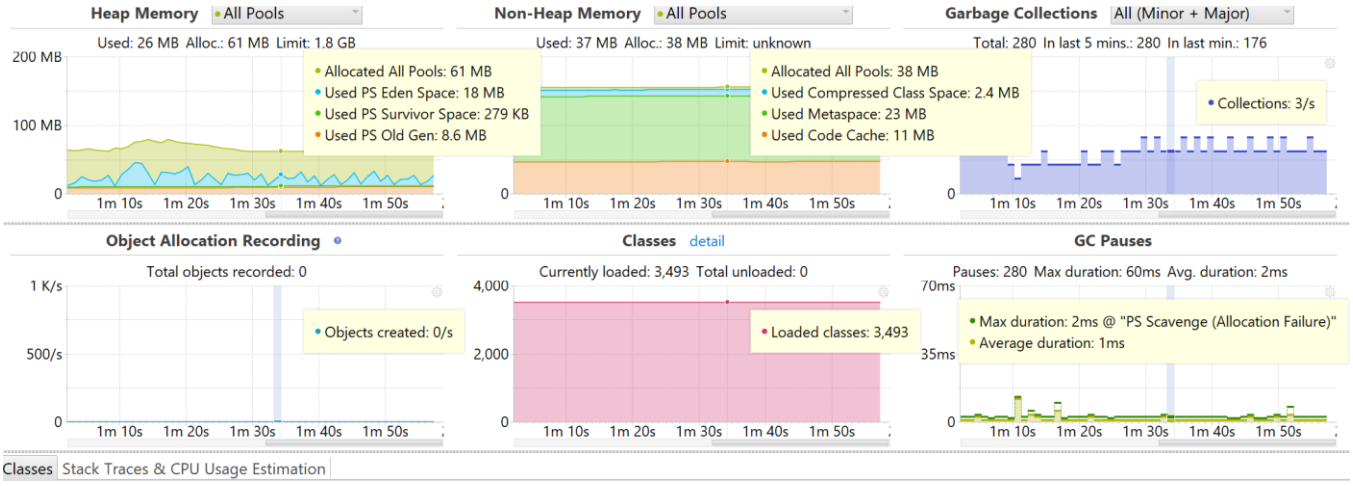


Obrázok 9 Performance Chart Pred

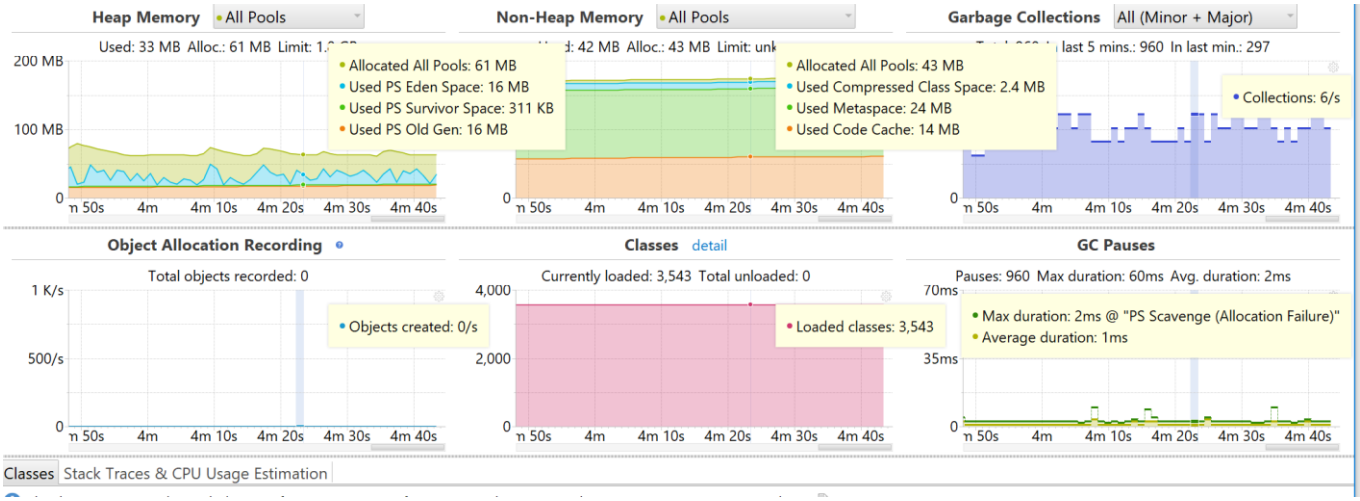


Obrázok 10 Performance Chart Po

43.5 Memory Chart pred a po spustení



Obrázok 11 Memory Pred



Obrázok 12 Memory Po

43.6 Metody pred a po spustení

Method	Time (ms)	Own Time (ms)
sk.fiit.jim.init.**Main.main**(String[]) Main.java	59,817 99 %	0
sk.fiit.jim.agent.communication.**Communication.start**() Communication.java	56,536 94 %	< 0.1
sk.fiit.jim.agent.communication.**Communication.mainLoop**() Communication.java	55,547 93 %	< 0.1
java.io.**DataInputStream.readInt**()	28,021 47 %	28,021
sk.fiit.jim.agent.communication.**Communication.receive**() Communication.java	27,989 47 %	0
sk.fiit.jim.agent.parsing.**Parser.parse**(String) Parser.java	27,609 46 %	0
sk.fiit.jim.agent.parsing.**Parser.notifyObservers**() Parser.java	27,390 46 %	0
sk.fiit.jim.agent.models.**AgentModel.processNewServerMessage**(ParsedData) AgentModel.java	24,312 41 %	0
sk.fiit.jim.agent.models.**AgentModel.updateRotations**(ParsedData) AgentModel.java	19,875 33 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.updateRotations**(ParsedData) AgentRotationCalculator.java	19,843 33 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.getFlagsOfSideWithMoreFlagsSeen**(Map) AgentRotationCalculator.java	19,796 33 %	0
java.io.**BufferedWriter.close**()	17,078 29 %	17,078
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFile**(String) JimLocalFileCreator.java	15,203 25 %	0
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFileGameTime**(String) JimLocalFileCreator.java	7,171 12 %	0
java.util.logging.**Logger.log**(Level, String)	3,796 6 %	1,890
sk.fiit.jim.agent.models.**AgentModel.updateBodyPartsPositions2**() AgentModel.java	3,375 6 %	0
sk.fiit.jim.agent.models.**WorldModel.processNewServerMessage**(ParsedData) WorldModel.java	2,906 5 %	15
sk.fiit.jim.agent.communication.testframework.**Message$WorldModel.changed**(WorldModel) Message.java	2,828 5 %	93
sk.fiit.jim.log.**JimLocalCsvFileCreator.writeCsvLog**(String[]) JimLocalCsvFileCreator.java	2,828 5 %	0
sk.fiit.jim.log.**JimHtmlFormatter.format**(LogRecord) JimHtmlFormatter.java	1,859 3 %	15
java.util.logging.**LogRecord.getSourceClassName**()	1,546 3 %	1,546
java.io.**FileWriter.<init>**(String, boolean)	1,515 3 %	1,515
org.apache.commons.net.util.**Base64.encodeBase64String**(byte[]) Base64.java	1,359 2 %	1,359
sk.fiit.jim.log.**JimLocalCsvFileCreator.<init>**(String) JimLocalCsvFileCreator.java	1,062 2 %	0
sk.fiit.jim.agent.communication.**Communication.handshake**() Communication.java	989 2 %	0
sk.fiit.jim.agent.communication.**Communication.transmit**(String) Communication.java	906 2 %	0
java.io.**DataOutputStream.writeInt**(int)	843 1 %	843
sk.fiit.robocup.library.geometry.**Vector3D.cartesian**(double, double, double) Vector3D.java	796 1 %	15
sk.fiit.jim.log.**JimLocalFileCreator.<init>**() JimLocalFileCreator.java	781 1 %	0
sk.fiit.robocup.library.geometry.**Vector3D.calculateSpherical**() Vector3D.java	781 1 %	0
sk.fiit.jim.annotation.data.**AnnotationManager.loadAnnotations**(String) AnnotationManager.java	703 1 %	0
sk.fiit.jim.annotation.data.**XMLParser.parse**(File) XMLParser.java	703 1 %	0

Obrázok 13 Metody Pred

Method	Time (ms)	Own Time (ms)
sk.fiit.jim.init.**Main.main**(String[]) Main.java	159,280 99 %	0
sk.fiit.jim.agent.communication.**Communication.start**() Communication.java	155,999 98 %	< 0.1
sk.fiit.jim.agent.communication.**Communication.mainLoop**() Communication.java	155,010 97 %	46
java.io.**DataInputStream.readInt**()	102,485 64 %	102,485
sk.fiit.jim.agent.communication.**Communication.receive**() Communication.java	102,454 64 %	0
sk.fiit.jim.agent.parsing.**Parser.parse**(String) Parser.java	51,046 32 %	0
sk.fiit.jim.agent.parsing.**Parser.notifyObservers**() Parser.java	50,703 32 %	0
sk.fiit.jim.agent.models.**AgentModel.processNewServerMessage**(ParsedData) AgentModel.java	37,625 24 %	0
sk.fiit.jim.agent.models.**AgentModel.updateRotations**(ParsedData) AgentModel.java	28,890 18 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.updateRotations**(ParsedData) AgentRotationCalculator.java	28,859 18 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.getFlagsOfSideWithMoreFlagsSeen**(Map) AgentRotationCalculator.java	28,812 18 %	0
java.io.**BufferedWriter.close**()	25,046 16 %	25,046
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFile**(String) JimLocalFileCreator.java	22,593 14 %	0
sk.fiit.jim.agent.models.**WorldModel.processNewServerMessage**(ParsedData) WorldModel.java	12,531 8 %	109
sk.fiit.jim.agent.communication.testframework.**Message$WorldModel.changed**(WorldModel) Message.java	12,328 8 %	687
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFileGameTime**(String) JimLocalFileCreator.java	10,171 6 %	0
java.util.logging.**Logger.log**(Level, String)	7,328 5 %	3,343
sk.fiit.jim.agent.models.**AgentModel.updateBodyPartsPositions2**() AgentModel.java	6,687 4 %	0
org.apache.commons.net.util.**Base64.encodeBase64String**(byte[]) Base64.java	5,281 3 %	5,281
sk.fiit.jim.log.**JimHtmlFormatter.format**(LogRecord) JimHtmlFormatter.java	3,937 2 %	46
sk.fiit.jim.log.**JimLocalCsvFileCreator.writeCsvLog**(String[]) JimLocalCsvFileCreator.java	3,890 2 %	0
java.util.logging.**LogRecord.getSourceClassName**()	3,437 2 %	3,437
java.security.**MessageDigest.digest**(byte[])	3,203 2 %	3,203
java.io.**ObjectOutputStream.writeObject**(Object)	2,718 2 %	5,281
sk.fiit.jim.agent.communication.**Communication.transmit**(String) Communication.java	2,138 1 %	0
java.io.**FileWriter.<init>**(String, boolean)	2,109 1 %	2,109
java.io.**DataOutputStream.writeInt**(int)	1,930 1 %	1,930
sk.fiit.robocup.library.geometry.**Vector3D.cartesian**(double, double, double) Vector3D.java	1,484 1 %	15
sk.fiit.robocup.library.geometry.**Vector3D.calculateSpherical**() Vector3D.java	1,468 1 %	0
sk.fiit.jim.log.**JimLocalCsvFileCreator.<init>**(String) JimLocalCsvFileCreator.java	1,421 1 %	0
java.lang.**Math.asin**(double)	1,125 1 %	1,125
sk.fiit.jim.log.**JimLocalFileCreator.<init>**() JimLocalFileCreator.java	1,093 1 %	0

Obrázok 14 Metody Po

44. Testovanie scenára true pre debug taktiku s logmi

OS: windows 10 version 10.0.16299 amd64 proccesors 4

Vývojové prostredie: Eclipse

Nástroj: Yourkit

44.1 Stav pred spustením taktiky

1. **Heap memory:**
    - 124mb allocated
    - 15-99 mb used
    - Limit 1.8 GB
2. **Non – Heap memory:**
    - 39mb allocated
    - 38mb used
    - Limit unknown
3. **Garbage collection:**
    - 1/s
    - Collections: 156
    - Time: 0s
4. **Classes:**
    - Currently loaded: 3503
5. **CPU usage telemetry**
    - CPU time (user + kernel): 14% - 47%
    - CPU time (kernel): 0%- 3%
    - Time spent in GC: 0%
6. **Threads**
    - 13 threads
    - 9 daemon threads
    - 14 peak
    - Total created 18

44.2 Stav počas spustenej taktiky

Heap memory:

e. 61-73mb allocated
f. 36 mb used
g. Limit 1.8 GB

Non – Heap memory:

h. 41mb allocated
i. 40mb used
j. Limit unknown

Garbage collection:

k. 1/s
l. Collections: 642
m. Time: 1s

Classes:

n. Currently loaded: 3554

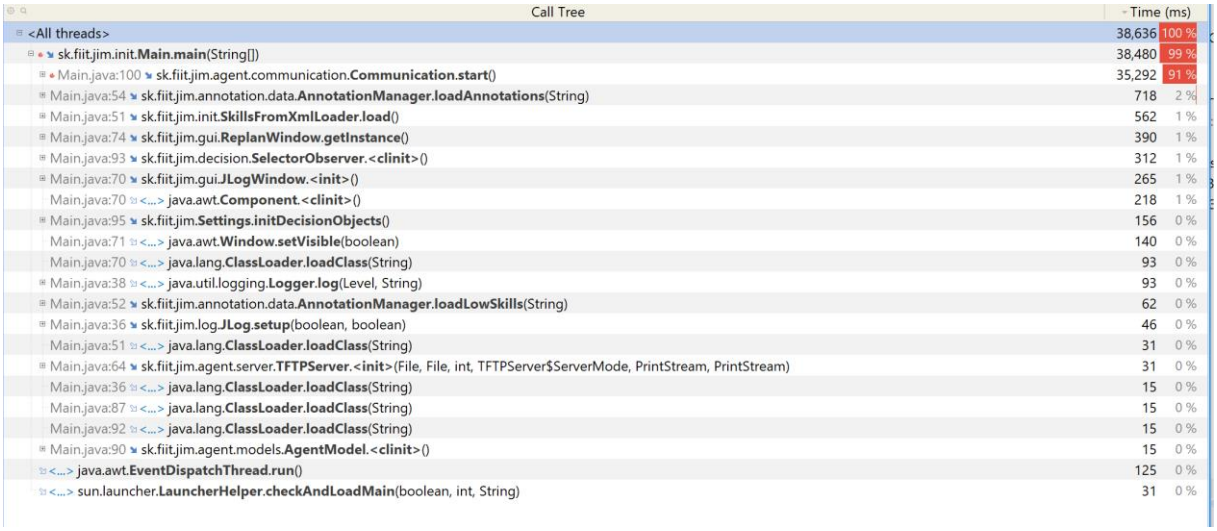
CPU usage telemetry

o. CPU time (user + kernel): 14% - 47%
p. CPU time (kernel): 0%- 3%
q. Time spent in GC: 0%

Threads

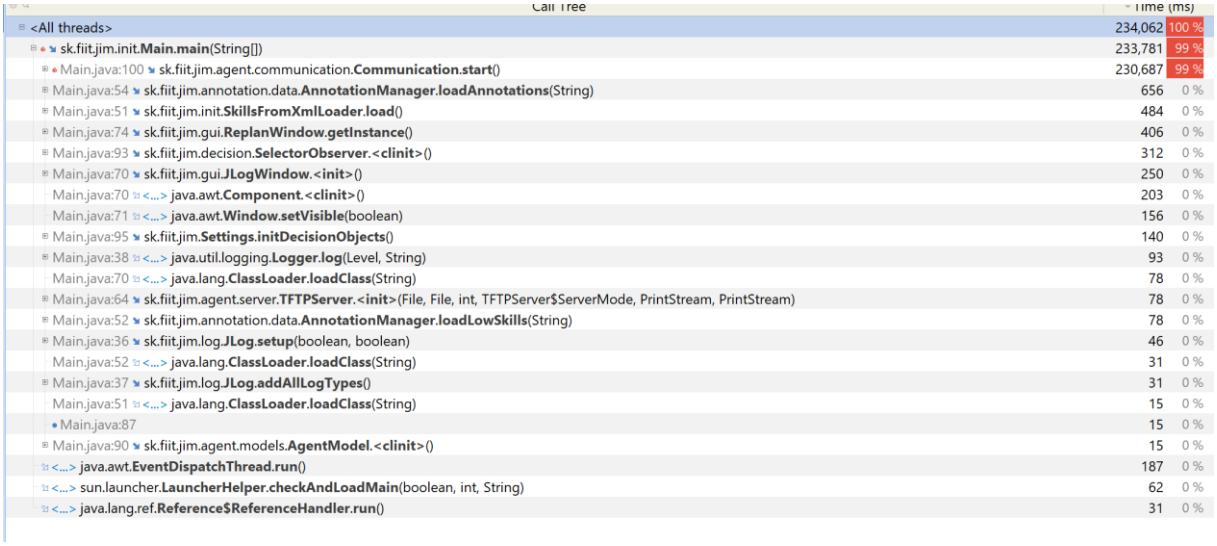
r. 13 threads
s. 9 daemon threads
t. 14 peak
u. Total created 18

44.3 Call Tree pred spustením vs. Call Tree po spustení



Call Tree	Time (ms)	
<All threads>	38,636	100 %
sk.fiit.jim.init.**Main.main**(String[])	38,480	99 %
Main.java:100 sk.fiit.jim.agent.communication.**Communication.start**()	35,292	91 %
Main.java:54 sk.fiit.jim.annotation.data.**AnnotationManager.loadAnnotations**(String)	718	2 %
Main.java:51 sk.fiit.jim.init.**SkillsFromXmlLoader.load**()	562	1 %
Main.java:74 sk.fiit.jim.gui.**ReplanWindow.getInstance**()	390	1 %
Main.java:93 sk.fiit.jim.decision.**SelectorObserver.<clinit>**()	312	1 %
Main.java:70 sk.fiit.jim.gui.**JLogWindow.<init>**()	265	1 %
Main.java:70 <...> java.awt.**Component.<clinit>**()	218	1 %
Main.java:95 sk.fiit.jim.**Settings.initDecisionObjects**()	156	0 %
Main.java:71 <...> java.awt.**Window.setVisible**(boolean)	140	0 %
Main.java:70 <...> java.lang.**ClassLoader.loadClass**(String)	93	0 %
Main.java:38 <...> java.util.logging.**Logger.log**(Level, String)	93	0 %
Main.java:52 sk.fiit.jim.annotation.data.**AnnotationManager.loadLowSkills**(String)	62	0 %
Main.java:36 sk.fiit.jim.log.**JLog.setup**(boolean, boolean)	46	0 %
Main.java:51 <...> java.lang.**ClassLoader.loadClass**(String)	31	0 %
Main.java:64 sk.fiit.jim.agent.server.**TFTPServer.<init>**(File, File, int, TFTPServer$ServerMode, PrintStream, PrintStream)	31	0 %
Main.java:36 <...> java.lang.**ClassLoader.loadClass**(String)	15	0 %
Main.java:87 <...> java.lang.**ClassLoader.loadClass**(String)	15	0 %
Main.java:92 <...> java.lang.**ClassLoader.loadClass**(String)	15	0 %
Main.java:90 sk.fiit.jim.agent.models.**AgentModel.<clinit>**()	15	0 %
<...> java.awt.**EventDispatchThread.run**()	125	0 %
<...> sun.launcher.**LauncherHelper.checkAndLoadMain**(boolean, int, String)	31	0 %

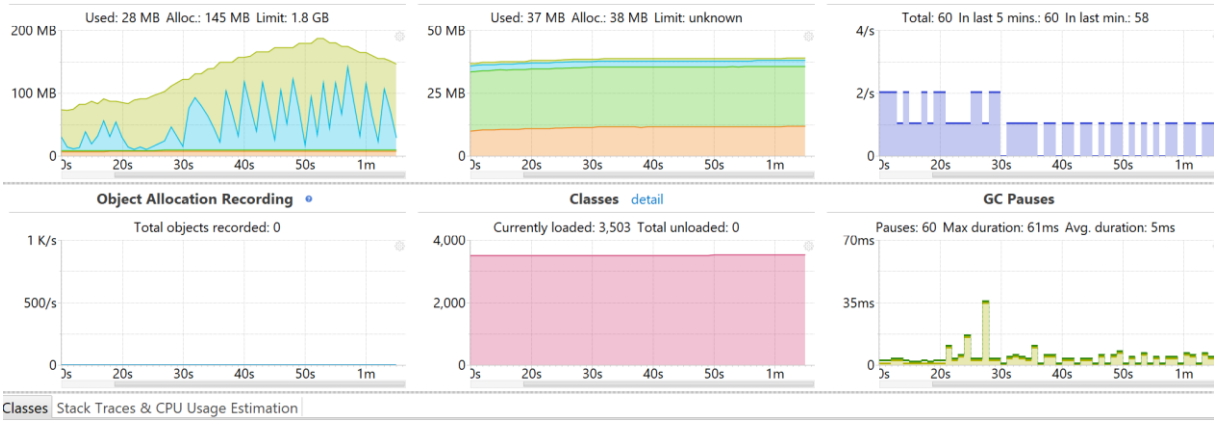
Obrázok 15 Call tree pred spustením



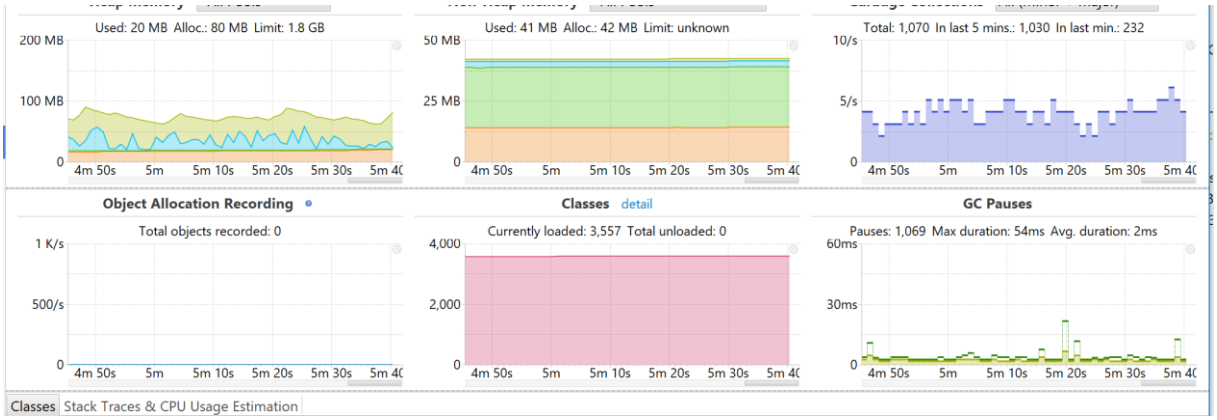
Call Tree	Time (ms)	
<All threads>	234,062	100 %
sk.fiit.jim.init.**Main.main**(String[])	233,781	99 %
Main.java:100 sk.fiit.jim.agent.communication.**Communication.start**()	230,687	99 %
Main.java:54 sk.fiit.jim.annotation.data.**AnnotationManager.loadAnnotations**(String)	656	0 %
Main.java:51 sk.fiit.jim.init.**SkillsFromXmlLoader.load**()	484	0 %
Main.java:74 sk.fiit.jim.gui.**ReplanWindow.getInstance**()	406	0 %
Main.java:93 sk.fiit.jim.decision.**SelectorObserver.<clinit>**()	312	0 %
Main.java:70 sk.fiit.jim.gui.**JLogWindow.<init>**()	250	0 %
Main.java:70 <...> java.awt.**Component.<clinit>**()	203	0 %
Main.java:71 <...> java.awt.**Window.setVisible**(boolean)	156	0 %
Main.java:95 sk.fiit.jim.**Settings.initDecisionObjects**()	140	0 %
Main.java:38 <...> java.util.logging.**Logger.log**(Level, String)	93	0 %
Main.java:70 <...> java.lang.**ClassLoader.loadClass**(String)	78	0 %
Main.java:64 sk.fiit.jim.agent.server.**TFTPServer.<init>**(File, File, int, TFTPServer$ServerMode, PrintStream, PrintStream)	78	0 %
Main.java:52 sk.fiit.jim.annotation.data.**AnnotationManager.loadLowSkills**(String)	78	0 %
Main.java:36 sk.fiit.jim.log.**JLog.setup**(boolean, boolean)	46	0 %
Main.java:52 <...> java.lang.**ClassLoader.loadClass**(String)	31	0 %
Main.java:37 sk.fiit.jim.log.**JLog.addAllLogTypes**()	31	0 %
Main.java:51 <...> java.lang.**ClassLoader.loadClass**(String)	15	0 %
Main.java:87	15	0 %
Main.java:90 sk.fiit.jim.agent.models.**AgentModel.<clinit>**()	15	0 %
<...> java.awt.**EventDispatchThread.run**()	187	0 %
<...> sun.launcher.**LauncherHelper.checkAndLoadMain**(boolean, int, String)	62	0 %
<...> java.lang.ref.**Reference$ReferenceHandler.run**()	31	0 %

Obrázok 16 Call tree po spustení

44.4 Memory pred vs Memory Po spustení:



Obrázok 17 Memory pred spustením



Obrázok 18 Memory po spustení

Method	Time (ms)	Own Time (ms)
sk.fiit.jim.init.**Main.main**(String[]) Main.java	48,632 99 %	0
sk.fiit.jim.agent.communication.**Communication.start**() Communication.java	45,444 93 %	< 0.1
sk.fiit.jim.agent.communication.**Communication.mainLoop**() Communication.java	45,243 93 %	0
sk.fiit.jim.agent.parsing.**Parser.parse**(String) Parser.java	34,750 71 %	0
sk.fiit.jim.agent.parsing.**Parser.notifyObservers**() Parser.java	34,578 71 %	0
sk.fiit.jim.agent.models.**AgentModel.processNewServerMessage**(ParsedData) AgentModel.java	30,312 62 %	0
sk.fiit.jim.agent.models.**AgentModel.updateRotations**(ParsedData) AgentModel.java	24,687 51 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.getFlagsOfSideWithMoreFlagsSeen**(Map) AgentRotationCalculator.java	24,687 51 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.updateRotations**(ParsedData) AgentRotationCalculator.java	24,687 51 %	0
java.io.**BufferedWriter.close**()	20,984 43 %	20,984
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFile**(String) JimLocalFileCreator.java	18,968 39 %	0
sk.fiit.jim.agent.communication.**Communication.receive**() Communication.java	9,915 20 %	0
java.io.**DataInputStream.readInt**()	9,899 20 %	9,899
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFileGameTime**(String) JimLocalFileCreator.java	8,562 18 %	0
java.util.logging.**Logger.log**(Level, String)	4,828 10 %	2,468
sk.fiit.jim.agent.models.**AgentModel.updateBodyPartsPositions2**() AgentModel.java	4,593 9 %	15
sk.fiit.jim.agent.models.**WorldModel.processNewServerMessage**(ParsedData) WorldModel.java	3,984 8 %	31
sk.fiit.jim.agent.communication.testframework.**Message$WorldModel.changed**(WorldModel) Message.java	3,890 8 %	171
sk.fiit.jim.log.**JimLocalCsvFileCreator.writeCsvLog**(String[]) JimLocalCsvFileCreator.java	3,546 7 %	0
java.io.**FileWriter.<init>**(String, boolean)	2,375 5 %	2,375
sk.fiit.jim.log.**JimHtmlFormatter.format**(LogRecord) JimHtmlFormatter.java	2,343 5 %	46
java.util.logging.**LogRecord.getSourceClassName**()	2,000 4 %	2,000
sk.fiit.jim.log.**JimLocalCsvFileCreator.<init>**(String) JimLocalCsvFileCreator.java	1,640 3 %	0
org.apache.commons.net.util.**Base64.encodeBase64String**(byte[]) Base64.java	1,500 3 %	1,500
java.io.**ObjectOutputStream.writeObject**(Object)	1,234 3 %	2,343
sk.fiit.jim.log.**JimLocalFileCreator.<init>**() JimLocalFileCreator.java	1,187 2 %	0
sk.fiit.robocup.library.geometry.**Vector3D.calculateSpherical**() Vector3D.java	953 2 %	0
sk.fiit.robocup.library.geometry.**Vector3D.cartesian**(double, double, double) Vector3D.java	953 2 %	0
java.lang.**Math.asin**(double)	765 2 %	765
sk.fiit.jim.agent.communication.**Communication.transmit**(String) Communication.java	732 2 %	0
java.security.**MessageDigest.digest**(byte[])	718 1 %	718
sk.fiit.jim.annotation.data.**AnnotationManager.loadAnnotations**(String) AnnotationManager.java	718 1 %	0

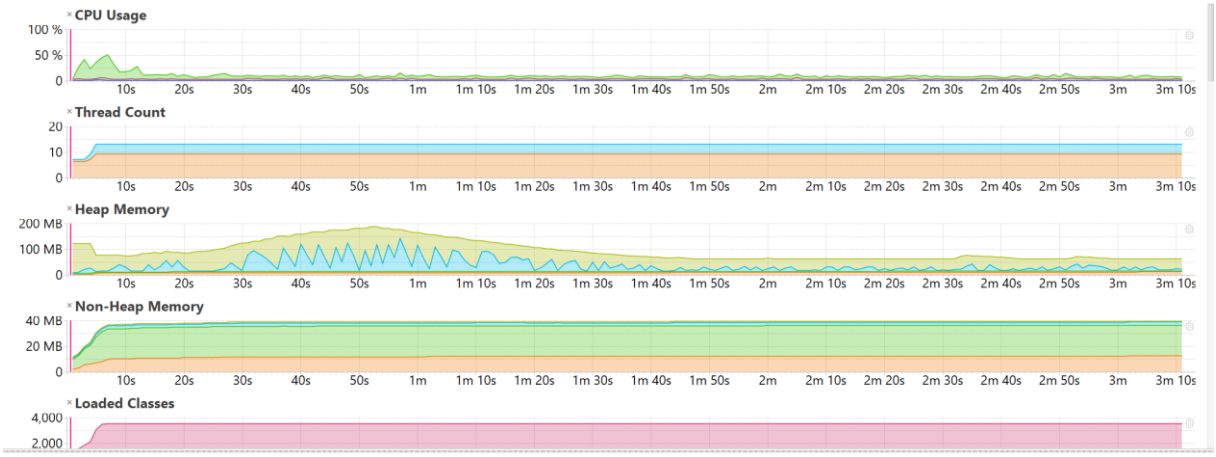
Obrázok 19 MethodListCPUPredSpustením

Method	Time (ms)	Own Time (ms)
sk.fiit.jim.init.**Main.main**(String[]) Main.java	248,524 99 %	15
sk.fiit.jim.agent.communication.**Communication.start**() Communication.java	245,431 99 %	0
sk.fiit.jim.agent.communication.**Communication.mainLoop**() Communication.java	244,196 98 %	46
sk.fiit.jim.agent.communication.**Communication.receive**() Communication.java	120,732 49 %	0
java.io.**DataInputStream.readInt**()	120,644 48 %	120,644
sk.fiit.jim.agent.parsing.**Parser.parse**(String) Parser.java	120,375 48 %	0
sk.fiit.jim.agent.parsing.**Parser.notifyObservers**() Parser.java	119,609 48 %	62
sk.fiit.jim.agent.models.**AgentModel.processNewServerMessage**(ParsedData) AgentModel.java	76,062 31 %	0
sk.fiit.jim.agent.models.**AgentModel.updateRotations**(ParsedData) AgentModel.java	48,781 20 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.updateRotations**(ParsedData) AgentRotationCalculator.java	48,781 20 %	0
sk.fiit.jim.agent.models.**AgentRotationCalculator.getFlagsOfSideWithMoreFlagsSeen**(Map) AgentRotationCalculator.java	48,734 20 %	0
sk.fiit.jim.agent.models.**WorldModel.processNewServerMessage**(ParsedData) WorldModel.java	41,718 17 %	312
sk.fiit.jim.agent.communication.testframework.**Message$WorldModel.changed**(WorldModel) Message.java	41,171 17 %	1,859
java.io.**BufferedWriter.close**()	40,984 16 %	40,984
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFile**(String) JimLocalFileCreator.java	35,984 14 %	0
sk.fiit.jim.agent.models.**AgentModel.updateBodyPartsPositions2**() AgentModel.java	22,640 9 %	15
java.util.logging.**Logger.log**(Level, String)	21,625 9 %	9,765
org.apache.commons.net.util.**Base64.encodeBase64String**(byte[]) Base64.java	17,656 7 %	17,656
sk.fiit.jim.log.**JimLocalFileCreator.writeLogFileGameTime**(String) JimLocalFileCreator.java	15,718 6 %	0
sk.fiit.jim.log.**JimHtmlFormatter.format**(LogRecord) JimHtmlFormatter.java	11,781 5 %	203
java.security.**MessageDigest.digest**(byte[])	10,562 4 %	10,562
java.util.logging.**LogRecord.getSourceClassName**()	10,250 4 %	10,250
java.io.**ObjectOutputStream.writeObject**(Object)	10,062 4 %	19,781
sk.fiit.jim.log.**JimLocalCsvFileCreator.writeCsvLog**(String[]) JimLocalCsvFileCreator.java	7,468 3 %	0
sk.fiit.robocup.library.geometry.**Vector3D.calculateSpherical**() Vector3D.java	5,703 2 %	0
sk.fiit.robocup.library.geometry.**Vector3D.cartesian**(double, double, double) Vector3D.java	5,703 2 %	0
java.io.**FileWriter.<init>**(String, boolean)	4,140 2 %	4,140
java.lang.**Math.asin**(double)	4,093 2 %	4,093
sk.fiit.jim.log.**JimLocalCsvFileCreator.<init>**(String) JimLocalCsvFileCreator.java	3,609 1 %	0
sk.fiit.jim.agent.communication.**Communication.transmit**(String) Communication.java	3,558 1 %	< 0.1
java.io.**DataOutputStream.writeInt**(int)	3,272 1 %	3,272
sk.fiit.jim.agent.models.**BodyPart.computeRelativePositionsToTorso**(Map, Map) BodyPart.java	2,875 1 %	0

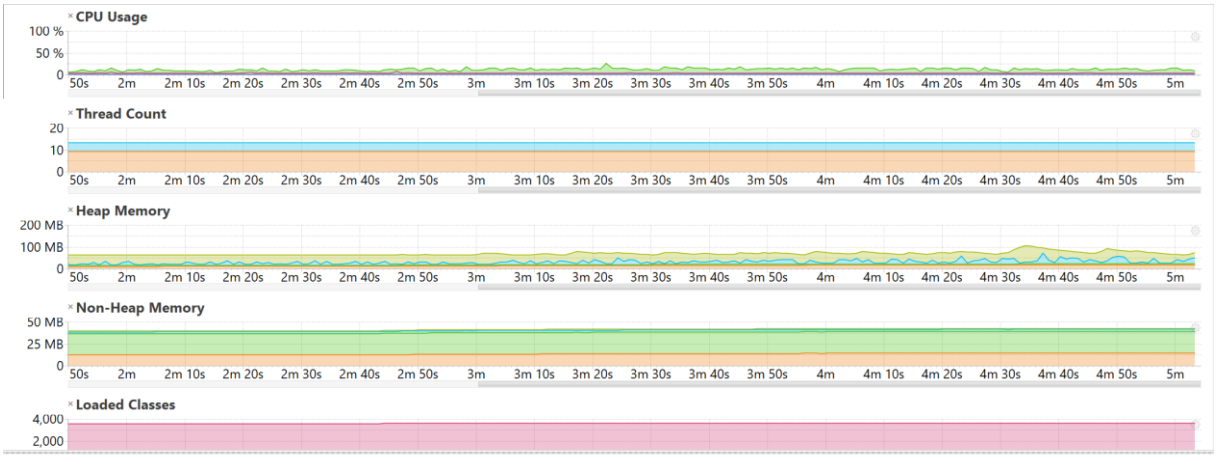
Obrázok 20 MethodListCpuPoSpustení

44.5 Performance Chart pred spustením vs:

Performance Chart po spustení:

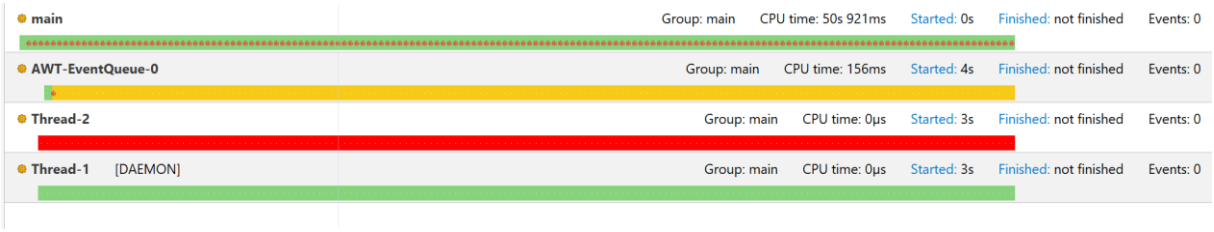


Obrázok 21 Performance Chart Pred spustením

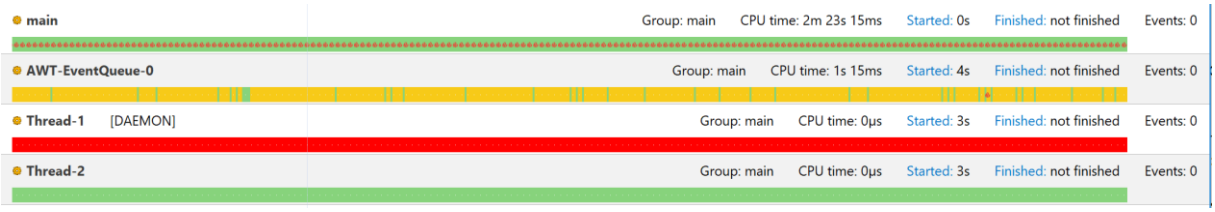


Obrázok 22 Performance Chart Po spustení

44.6 Thready pred spustením vs Thready po spustení:



Obrázok 23 Thread yPred spustením



Obrázok 24 Thready Po spustení

45. Záver

45.1 Prvý semester

V prvej etape sme sa oboznámili s projektom a do istej miery analyzovali aktuálny stav projektu.

Pre túto etapu sme mali zvolené štyri hlavné ciele:

1. Optimalizácia pomalých funkcií
2. Zlepšenie orientácie Jima
3. Analýza stabilizácie
4. Wiki

Optimalizácia pomalých funkcií

Projekt obsahoval mnoho funkcií, ktoré bolo možné optimalizovať. V prvotnej fáze sme sa zamerali na funkcie počítajúce sin a cos, ktoré sme úspešne implementovali. K optimalizovaným funkciám sme vytvorili testy.

Zlepšenie orientácie Jima

Orientácia robotického hráča Jima bola minuloročným tímom zlepšená o vnímanie čiar. Tieto čiary sú vnímané pomocou kamery, ktorú ma Jim zabudovanú v robotickej hlave. Tieto vstupy sú zašumené a preto sme sa snažili vylepšiť Kalmanov filter. Cieľom bolo správne analyzovať a nájsť implementáciu Kalmanovho filtra v doterajšom projekte. To sa nám do istej miery podarilo a s výsledkami z analýzy sme pracovali v ďalšej etape.

Analýza stabilizácie

Stabilita Jima je základná pre rekvizita hrania futbalu. Z analýz počas semestra sme zistili, že Jim si je síce vedomí svojho náklonu, ale nevyhodnocuje ho k zlepšeniu stability a preto sa stáva, že počas behu padá. V tejto etape sme našli túto chybu a vypracovali sme dokument na základe ktorého sme pokračovali v ďalšom semestri.

Aktualizácia Wiki

Súčasťou projektu je Wiki, ktorá sa s implementáciou stáva neaktuálna a s novými poznatkami neúplná. V tomto semestri sme pokračovali v dokumentácií dokuwiki, ktorú nastavili minulý rok.

45.2 Druhý semester

Na základe analýz z prvej etapy sme nastavili tieto ciele:

1. Implementácia Kalmanovho filtra na loptu
2. Implementácia Kalmanovho filtra na pevne body
3. Implementácia zohľadnenia náklonu na os Z
4. Implementácia zohľadnenia náklonu pre os Y
5. Modifikácia logovania v TestFrameworku pre lepšiu prácu s výstupnými dátami
6. Wiki

Implementácia Kalmanovho filtra na loptu

Kalmanov filter na loptu sa nám podarilo implementovať. Po dôkladnej analýze na začiatku etapy sme vytvorili testovací scenár na ktorom agent kopol do lopty a na základe súradníc lopty a vnímaných súradníc sme zisťovali ako je filter úspešný. Výsledok implementácie bol popísaný v sekcii (Celkové zhrnutie Kalman filtra **Chyba! Nenašiel sa žiaden zdroj odkazov.**). Na základe výsledkov z experimentov sme zistili, že novej implementácií sa podarilo zlepšiť odstránenie šumu z údajov zo servera.

Implementácia Kalmanovho filtra na pevné body

Kvôli zložitosti a časovej náročnosti riešenia implementácie Kalmanovho filtra na loptu sa Kalmanov filter na pevné body nepodarilo implementovať.

Implementácia zohľadnenia náklonu na os Z

Os Z sme spravili tak, že sme opravili výšku v ktorej robot vníma dane body na 0.56, čo je výška robota. Tým pádom sa mu svet nedvíha ale dochádza ku kužeľovitému skresleniu ktoré má za následok menšiu osciláciu oproti pôvodnému riešeniu s náklonom sveta

Implementácia zohľadnenia náklonu pre os Y

Os Y sme nestihli opraviť, ale keby sa implementovali rotačné matice tak ako som bolo popísané v prislúchajúcom dokumente predpokladáme, že by to malo za následok opravenie toho náklonu

Modifikácia logovania v TestFrameworku pre lepšiu prácu s výstupnými dátami

V TestFrameworku sme vytvorili logovací nástroj, ktorý dokáže vyexportovať logy do textového alebo csv súboru. Vytvorený nástroj sme otestovali a vytvorili návod, ktorý sme zverejnili aj na Wiki. Tento nástroj sme počas tejto etapy aj aktívne využívali.

Aktualizácia Wiki

Wiki sme aj v tejto etape aktualizovali a doplnili sme novú organizáciu.

46. Odporúčania pre pokračovanie v projekte

V rámci práce tímového projektu v akademickom roku 2017/2018 sme vytvorili pár odporúčaní pre pokračovanie v projekte. Tieto nápady nie sú uvádzané v žiadnej postupnosti, ani nevytvárajú nový profil pre ďalšie pokračovanie. Poskytujú len obraz toho, na čom všetkom by sme pracovali my, ak by sa nám zvýšilo viac času.

Jednotlivé odporúčania:

- bolo by vhodné urobiť refactor dostupných stratégií, konkrétne zanalyzovať, ktoré ako fungujú a následne sa zaoberať problematikou ich úpravy, prípadne vytvorenia nových, efektívnejších stratégií
- z našich záverov vyplýva, že kód môže byť miestami neefektívny, čo znamená, že sa treba zamerať na optimalizáciu niektorých častí projektu, prípadne vyradenie alebo nahradenie istých častí kódu
- náš tím implementoval kalman filter pomocou matíc, s ktorými je ale potrebné ďalej pracovať, dolaďovať ich hodnoty a optimalizovať, prípadne rozšíriť matice
- ponúka sa možnosť, už implementovaného kalmana použiť aj na pevné body alebo ho začať používať na hráča
- v rámci refactoringu je vhodné naučiť sa pracovať s programom Yourkit, ktorý je vhodný pre vytvorenie analýz a štatistík efektivity kódu (jednotlivých package-ov, tried, metód a podobne), pre ktorý náš tím vybavil 10 ročných licencií, ktoré má u seba Bc. Loureiro Ernest
- je vhodné analyzovať súčasnú problematiku robocupu a niektoré ďalšie metódy, technológie, ktoré sa používajú na zefektívnenie hry
- Vytvoriť mechanizmus na rýchle implementovanie dokončených BP a DP. Bolo by dobré, aby každá BP/DP mala vytvorenú vetvu na BitBucket-e, pričom by sa po obhajobe vetvy zlúčili do hlavnej vetvy (master).

Príloha A

A Testovanie druhou stranou

Členovia tímu Breyslet (tím 6) nám predviedli výsledok ich tímového projektu. Pracujú na inteligentných hodinkách, ktoré by mali sledovať fyziologické hodnoty ľudského tela ako teplota, tep a okysličenie krvi. Hodinky by mali používať gyroskop na zistenie toho, či nositeľ spadol. Taktiež sledujú GPS pozíciu nositeľa. Vytvorili prototyp hardvéru, ktorý komunikuje so serverom a posiela informácie na Android, iOS a webovú aplikáciu. Ich prototyp však zatiaľ dokáže zmerať len tep a odoslať ho prostredníctvom technológie Sigfox na server. Na hodinkách by sa tiež malo nachádzať tlačidlo záchrany, po jeho stlačení sa pošle správa na server a server kontaktuje zodpovednú osobu prostredníctvom mobilnej aplikácie.

Testovanie vyžadovalo prítomnosť členov tímu nakoľko sme museli otestovať ich hardvérový prototyp. Pri testovaní sme skúšali celkovú funkčnosť projektu a prepojenie jednotlivých jeho častí - prepojenie hardvérovej časti s aplikáciami.

Hlavný testovací scenár pozostával s namerania tepu a jeho následné zobrazenie v mobilnej Android aplikácií. Celý proces vyžadoval okrem namerania tepu senzorom aj spracovanie tejto hodnoty, ktorá je následne odoslaná na aplikačný server, kde sa prijatá správa zo zariadenia spracuje a uloží do databázy. Testovaný proces prebehol bez problémov a po nameraní sme si hodnoty mohli pozrieť v mobilnej aplikácií. Ďalší testovací scenár bolo otestovanie tlačidla záchrany, ktorého fungovanie je nevyhnutné vzhľadom na povahu projektu. Pri stlačený tlačidla sa muselo ihneď odoslať upozornenie na príslušné mobilné zariadenia. Tento scenár prebehol v poriadku a upozornenie sa ihneď zobrazilo v mobile. Ďalej sme testovali registráciu používateľa, prihlásenie používateľa a prvotné priradenie breysletu k používateľovi. Scenáre prebehli dobre a pri registrácií ani prihlásení neboli žiadne problémy. Pri priradení breysletu je nutné vedieť a zadať jeho jedinečne ID.

Mobilné aplikácie sa správali v každom testovacom scenári korektne. Na iOS aplikácii nebolo možné otestovať notifikovanie po stlačení tlačidla záchrany, pretože tímu nebol poskytnutý vývojársky Apple účet ktorý povoľuje aplikácii odoslať notifikáciu na pozadí. (Takýto účet je spoplatnený čo je dôvod prečo ho k dispozícii tím 6 nemal.) Na Android zariadení sa tento scenár odohral bez problémov kedy

zobudil Android zariadenie z tzv. "hlbokého spánku" a zazvonila notifikácia. V čase testovania sa nám podarilo objaviť chybu kedy sme si zo serveru boli schopní (prostredníctvom Android aplikácie) vypýtať dáta v príliš veľkom rozsahu čo malo za následok ukončenie komunikácie so serverom. Konkrétne sa jednalo o stiahnutie dát histórie vybratím kombinácie rozsahu "Lifetime" a intervalu "15 min". Vytvorili aj webovú aplikáciu, ktorá však neposkytovala fukcionalitu porovnateľnú s aplikáciami. To bolo zapríčinene tým, že tímu odišli členovia, a webovej aplikácií sa nemal kto dostatočne venovať.

Vzhľadom na okolnosti sme si vedomí, že projekt má svoje nedostatky, pretože tím opustili dvaja členovia a pracujú na veľmi rozsiahlom projekte len piati. Navyše žiaden z nich nikdy nepracoval na hardvérovom projekte a nemá skúsenosti s programovaním firmvéru. Členovia tímu tiež spomínali, že pôvodne pracovali s vývojovou doskou procesora ATmega128, ale miesiace mali problémy s implementáciou spojenia s určitým druhom senzorov. Preto sa rozhodli použiť Arduino, s ktorým už také problémy nemali.