České vysoké učení technické
Fakulta elektrotechnická

Multiagentní systémy jako řídící algoritmy v počítačových hrách

Bakalářská práce

Ondřej Slánský

Praha, 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Ondřej Slánský**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Multiagentní systémy jako řídicí algoritmy v počítačových hrách**

Pokyny pro vypracování:

1. Seznamte se s problematikou multiagentních systémů (MAS)
2. Navrhněte a implementujte MAS pro řízení pohybujících se objektů v dvourozměrném prostoru (např. kosmických lodí)
3. Při návrhu použijte prioritní vrstevnaté struktury a vše implementujte v jazyce Objective-C

Seznam odborné literatury:

[1] A. Kubík, Inteligentní agenty, 1., Brno: Computer press, 2004.
[2] P. Burian, Webové a agentové technologie, 1., Praha: Grada, 2012.

Vedoucí: Ing. Matouš Pokorný

Platnost zadání: do konce zimního semestru 2013/2014

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 4. 2. 2013

Poděkování

Rád bych na tomto místě poděkoval svému vedoucímu Ing. Matouši Pokornému za cenné připomínky a rady při řešené bakalářské práce. Dále děkuji Tomáši Jukinovi za odborné konzultace. V neposlední řadě bych chtěl poděkovat své rodině za velkou podporu během dosavadního studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne 22.5.2013 Ondřej Hlávský

Vedoucí práce: Ing. Matouš Pokorný

Abstrakt

Cílem této práce je uvést náhled do problematiky multiagentních systémů (MAS) a demonstrovat jejich funkci pomocí implementace prioritní vrstevnaté struktury agentů řídících pohyb objektů (vesmírných lodí) ve 2D světě v demo aplikaci hry určené pro platformy se systémem iOS. Jsou zde uvedeny hlavní výhody MAS a důvody zvolení právě této metody. Implementace je provedena v jazyce Objective-C s použitím herního frameworku Cocos2D a fyzikálního frameworku Chipmunk vhodných pro řešení pohybu ve 2D světě.

Klíčová slova

Multiagentní systémy, Objective-C, počítačové hry, pohyb ve 2D prostoru, Chipmunk, Cocos2D

Abstract

The aim of this work is to bring insight into multi-agent systems (MAS) and to demonstrate their performance through implementation of priority layered agent structure controlling movement of objects (spaceships) in 2D world in demo version of computer game for iOS platform. The implementation is done in Objective-C using game engine Cocos2D and physical engine Chipmunk suitable for movement solutions in 2D world.

Keywords

Multiagent systems, Objective-C, computer games, movement in 2D space, Chipmunk, Cocos2D

Obsah

1 **Úvod** 1
1.1 Hry a teorie her 1
1.2 Multiagentní systémy 1
1.3 Osnova práce 1
2 **Problematika multiagentních systémů** 2
2.1 Multiagentní systémy 2
2.2 Agent 3
2.2.1 Reaktivní agent 5
2.2.2 Deliberativní (uvažující) agent 6
2.2.3 Sociální agent 7
2.2.4 Hybridní agent 7
2.3 Prostředí 7
2.4 Interakce agentů 8
3 **Rozbor úlohy** 9
3.1 Úvod 9
3.2 Vermírné lodě ve 2D vesmíru 10
3.3 Prostředí 11
3.4 Pohyb 12
3.4.1 Přímý pohyb 12
3.4.2 Rotační pohyb 14
3.4.3 Boční pohyb 15
3.4.4 Zastavení 16
3.4.5 Zabraňování kolizím 16
3.5 Střelba 20
3.6 Řízení autonomní lodi 21
3.6.1 Útok 22
3.6.2 Útěk 23
3.6.3 Vyčkávání 23
3.7 Schéma celé agentové struktury 25
4 **Implementace** 26

4.1 Objective-C 26
4.2 Agentový systém 26
4.2.1 Prostředí a jeho role 26
4.2.2 Vrstvy MAS 27
4.2.3 Nastavení agentů 27
4.2.4 Struktura agentů 27
4.2.5 Cocos2D & Chipmunk 28
5 **Závěr** 29
Literatura 31
A **Obsah přiloženého CD** 33

Seznam zkratek

MAS	Multiagentní systém
KQML	Knowledge Query and Manipulation (vyšší komunikační jazyk agentů)
Obj-C	Objective-C

Seznam obrázků

2.1 Základní schéma agenta *4*
2.2 Schéma reaktivního agenta *5*
2.3 Beruška na klíček (převzato z internetové adresy http://www.hrackarna.cz/slunicko-cervene-901191.html) *6*
2.4 Schéma deliberativního agenta (podle [1]) *7*

3.1 Snímek obrazovky demo aplikace SpaceShips *10*
3.2 Loď zahajující přesun na místo určení *11*
3.3 Turrety signalizují zaměření a zahájí natáčení směrem k zaměřené oblasti *11*
3.4 Znázornění funkce motorů pro přímý pohyb *12*
3.5 Znázornění funkce motorů pro rotační pohyb *14*
3.6 Znázornění funkce motorů pro boční pohyb *15*
3.7 Loď zahajující pohyb směrem na meteorit *17*
3.8 Radar lodi detekuje nebezpečí kolize *17*
3.9 Loď zahajuje úhybný manévr *17*
3.10 Loď míjí meteorit, kterému se úspěšně vyhnula *18*
3.11 Nebezpečí kolize pominulo, loď se stáčí směrem k původnímu místu určení *18*
3.12 Loď již směřuje k místu určení *18*
3.13 Loď dobržďuje *19*
3.14 Loď zastavila na místě určení *19*
3.15 Znázornění veličin důležitých pro řešení kolizí *20*
3.16 Ukázka střelby hráčské lodi *20*
3.17 Ukázka autonomní lodi napadené uživatelem v momentě změny strategie na útok a zahájení zaměřování turretů *22*
3.18 Ukázka autonomní lodi s nízkým počtem životů během strategie útěk pronásledované hráčskou lodí, která právě detekuje nebezpečí kolize *23*
3.19 Ukázka autonomní lodi během strategie vyčkávání *24*
3.20 Schéma agentové struktury *25*

1 Úvod

1.1 Hry a teorie her

Lidé odjakživa hráli hry a to nejen pro zábavu, ale také jako nástroj rozvíjení logického myšlení. Hry se postupem času rozvinuly natolik, že vznikl samotný vědní obor teorie her, který se zabývá řešením konfliktů, rozhodovacích situací, v nichž dochází ke střetnutí zájmů jejich účastníků. V roce 1944 shrnuli John von Neumann a Oskar Morgenstern v publikaci Theory of Games and Economic Behavior dosud známé poznatky z této oblasti a podstatně je rozvinuli a doplnili. Po vydání knihy se začala teorie her rychle rozvíjet a stala se jednou z nejrychleji se rozvíjejících vědních disciplín posledních desetiletí. Díky obrovskému přínosu této publikace se vnik teorie her často datuje právě do roku 1944 [3].

1.2 Multiagentní systémy

Multiagentní systém (MAS) je systém skládající z několika prostředím propojených agentů schopných společně řešit problém, který přesahuje znalosti a možnosti každého z nich. Jsou využívány mimo jiné k řešení problémů, které jsou pro jednoho agenta těžko řešitelné. V současnosti se jedná o hojně využívanou techniku v mnoha odvětvích viz 2.1. V této práci ukazuji funkci MAS propojením právě s výše zmíněnou oblastí her. Toto propojení je blíže popsáno např. v [4, 5].

1.3 Osnova práce

Práce obsahuje pět kapitol včetně úvodu a závěru. V kapitole 2 se zabývám problematikou MAS, jejich definicí (2.1), jednotlivými částmi a zavedením základních pojmů (2.2, 2.3). Kapitola 3 obsahuje rozbor úlohy a popisuje demo aplikaci a její fungování jako celek i funkci jednotlivých agentů. V kapitole 4 doplňuji popis implementace a strukturu implementovaného MAS. V závěru se nachází shrnutí práce a námět na její rozvinutí.

2 Problematika multiagentních systémů

2.1 Multiagentní systémy

Multiagentní systém je decentralizovaný systém skládající se z autonomních, inteligentních agentů a prostředí. Inteligence agentů může být velmi omezená (např. semafor, který přepíná mezi dvěma stavy), ale také poměrně vysoká (např. helikoptéra – létající agent prozkoumávající terén se schopností vnímat okolí a překážky a rozhodovat se na základě vyhodnocení situace a znalostí získaných komunikací s ostatními agenty, kterými mohou být např. další helikoptéry mapující terén).

Jejich využití je celá řada, jsou používány všude tam, kde klasické centralizované systémy s předem definovaným chováním nevyhovují. Buď protože předem definovat chování celého systému není vůbec možné (řízení letového provozu, automatizovaný provoz výroby automobilů), nebo se jedná o aplikace v emergenčních a krizových situacích. V takových případech mohou nastat stavy, na které klasický centralizovaný systém neumí reagovat a došlo by ke kolapsu celého systému. Při použití autonomních agentů může zkolabovat jeden či více z nich, ostatní však movou situaci řešit autonomně na základě svých interních znalostí a kolabs celého systému nenastane. Příkladem může být predikce nebezpečných přírodních událostí jako sesuvů půdy, či tsunami v Indonésii a Japonsku. Mezi další příklady využití MAS patří např. logistika nebo modelování finančních trhů v ekonomii. Jejich využití však není tak rozšířené v počítačových hrách, a také proto mi přišlo zajímavé funkci multiagentních systémů demonstrovat právě na příkladu jednoduchého základu hry, na kterém je zároveň chování jednotlivých agentů zároveň velmi dobře pozorovatelné.

Mezi základní vlastnosti multiagentních systémů patří již zmíněné:

- *autonomie prvků*
- *decentralizace* – neexistuje globální řízení systému; data jsou decentralizovaná

a dále také:

- *robustnost* – při chybě nebo kolapsu části programu nezkolabuje celý systém
- *adaptibilita na změny prostředí*

Agenty mohou být jak hardwarové (viz příklad myši na klíček v sekci 2.2.1), tak softwarové. V této kapitole se dále budu věnovat především MAS ve vztahu k softwarovému inženýrství.

2.2 Agent

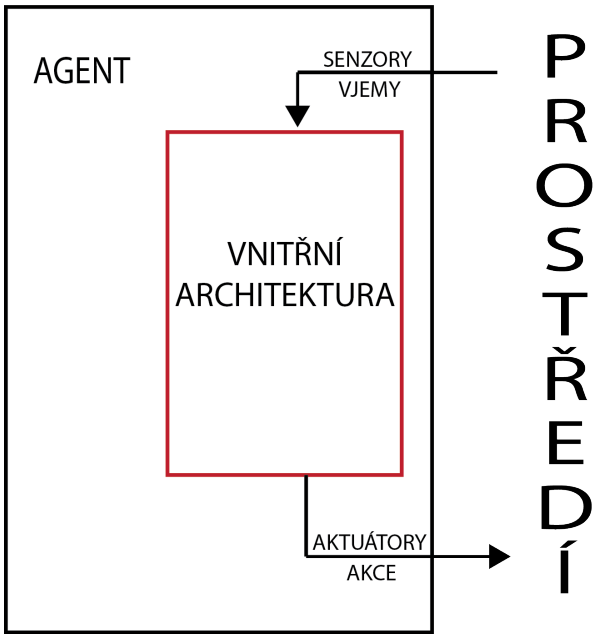
Definice agenta podle [1]:

> *„Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktuátorů. Agent přitom ovlivňuje podmínky v prostředí tak, aby se přibližoval k plnění cílů."*

Z této definice jsou tedy patrné nejzákladnější části agenta (viz obrázek 2.1):

- *senzory* – vnímají vnější prostředí agentů, jsou jeho vstupním rozhraním, příkladem senzoru může být například čidlo nebo radar, ať už hardwarový, nebo softwarový (např. třída v programu)
- *aktuátory* – konají akce ve vnějším prostředí agentů, jsou jeho výstupním rozhraním, pomocí aktuátorů agent může měnit stav okolního světa, např. kola pohonu (hardwarový), nebo zápis na disk (softwarový)
- *vnitřní architektura* – na základě získaných vjemů ze senzorů rozhoduje o vhodných akcích vedoucích k plnění cílů, které budou provedeny pomocí aktuátorů

Ve srovnání agenta s objektem, nejpropracovanější abstrakcí reality, komponentou modelovaného systému v objektově-orientovaném inženýrství, má agent několik důležitých vlastností navíc:



Obrázek 2.1: Základní schéma agenta

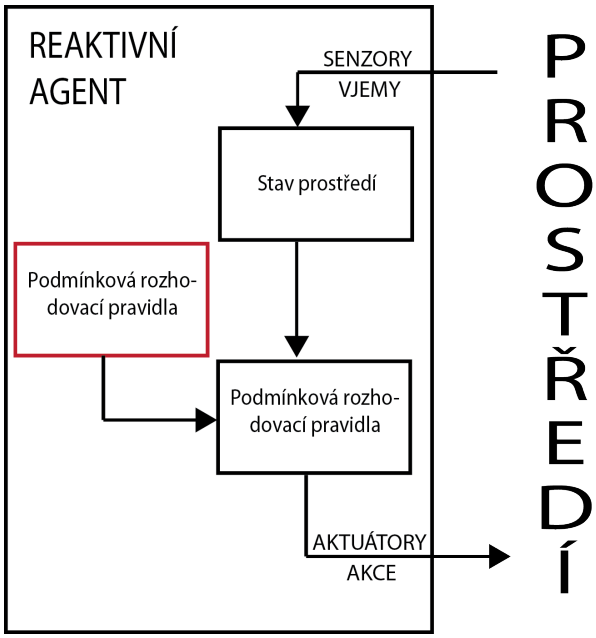
- nezávislost na ostatních agentech - autonomie
- žádný agent nemá plný náhled celého systému
- existence v prostředí
- propojení s prostředím pomocí senzorů a aktuátorů

Agenty se z hlediska složitosti organizace vnitřních komponent (vnitřní architektury) a jejich racionality dají rozdělit do čtyř skupin:

- reaktivní
- deliberativní
- sociální
- hybridní

2.2.1 Reaktivní agent

Typ agenta s nejjednodušší vnitřní architekturou, který je tvořen několika paralelními procesy – druhy chování (v extrémním případě jen jedno chování). Ty jsou spouštěny na základě podnětu z prostředí, nebo agentního vnitřního stavu, případně jejich kombinací. Kromě vyrovnávací paměti, obsahující záznam stavu, nemá agent vnitřní reprezentaci prostředí a akce jsou reakcí na stav prostředí bez možnosti plánování.



Obrázek 2.2: Schéma reaktivního agenta

Příkladem jednoho z nejprimitivnějších agentů podle [1] je beruška na klíč. Jedná se o speciální případ reaktivního agenta, tzv. čistě reaktivní agent. Je to reaktivní agent, který nemá svou interní paměť, ale využívá komunikaci s prostředím a ostatními agenty pro uchování výsledků vykonání akci. Beruška, kterou vidíme na obrázku 2.3, se dokáže pohybovat po desce stolu, aniž by z něj spadla. Pohybu po stole je schopna díky dvěma aktuátorům – kola pohonu, záchranné kolečko a jednomu senzoru – detekční pacička.

Kola pohonu zajišťují přímý pohyb berušky po stole a jsou poháněna

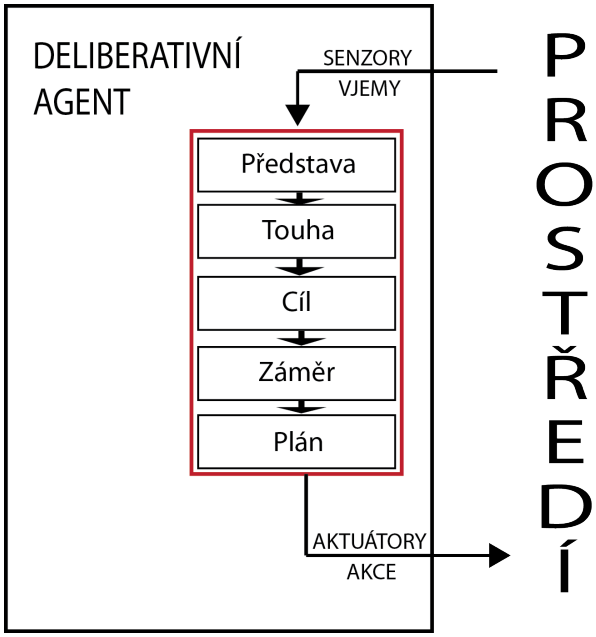
Obrázek 2.3: Beruška na klíček (převzato z internetové adresy http://www.hrackarna.cz/slunicko-cervene-901191.html)

pomocí pružiny natažené klíčkem. Detekční pacička je druhý opěrný bod, po kterém beruška jede během přímého pohybu, dokud se nepřiblíží k hraně stolu. V tu chvíli senzor detekční pacička spadne za hranu a pomocí aktuátoru – záchranné kolečko, které tím pádem dosedne na desku stolu a unese přední část berušky ve směru svého otáčení – kolmo ke směru otáčení kol pohonu. Poté, co se beruška působením tohoto aktuátoru otočí tak, že je detekční pacička opět na stole, pokračuje pohybem podél hrany, dokud se znovu nedostane na její konec, kde se celá situace opakuje. Pomocí dvou aktuátorů a jednoho senzoru je tedy možné vytvořit jednoduchého, v tomto případě hardwarového agenta, schopného samostatného pohybu po hraně stolu.

2.2.2 Deliberativní (uvažující) agent

Od reaktivního agenta se liší tím, že ke svému cíli putuje po naplánované (vědomé) cestě. Na rozdíl od reaktivního agenta, který postupuje při dosahování cíle jen jedním způsobem, deliberativní agent může volit mezi několika možnými způsoby jeho dosahování. Tento agent tedy vytváří plány pro dosažení svých cílů na základě symbolické reprezentace prostředí a vnitř-

ních stavů, které si uchovává. Plány mohou být hierarchicky uspořádané a cíle ohodnoceny přiřazenou prioritou pro výběr adekvátního plánu.



Obrázek 2.4: Schéma deliberativního agenta (podle [1])

2.2.3 Sociální agent

Za sociálního agenta je považován takový agent, který komunikuje s jinými agenty ve vyšším komunikačním jazyce (např. KQML [6]). Komunikací se však mohou vyznačovat i reaktivní agenty.

2.2.4 Hybridní agent

Příklad agentu, který kombinuje v jeden celek některé nebo všechny z uvedených architektur.

2.3 Prostředí

Prostředí hraje v multiagentních systémech velmi důležitou roli. Můžeme se setkat jednak s prostředím určeným pro správu agentů s úkolem umož-

nit jejich běh, nebo např. zajistit doručení zpráv v případě komunikace. Narozdíl od toho pak agent prostředí (Environmental Agent) slouží k udržování reprezentace modelovaného světa, na jehož podobu se mohou jednotlivé agenty dotazovat a podle určitých pravidel ho měnit.

2.4 Interakce agentů

Významnou roli v multiagentních systémech hrají interakce mezi agenty. Můžeme je rozdělit na:

- koordinaci
- kooperaci
- komunikaci

Koordinace zajišťuje propojení jednotlivých kompoment v systému, což umožní řešení problém decentrlizací vykonávaných úkolů. Kooperací je myšlena řízená forma koordinace s uspořádáním agentů ve skupině s cílem dosáhnout společného řešení problému. Každý agent má přesně určenou roli a vztahy s ostatními agenty vedoucí k tomu, aby bylo dosaženo globálního cíle. Komunikace mezi agenty probíhá formou vyšších komunikačních jazyků, jehož příkladem je například KQML (Knowledge Query and Manipulation Language) [6].

Téma multiagentních systémů je velmi rozsáhlé. Mým záměrem není pokrýt tuto problematiku celou, ale přiblížit základy MAS pro pochopení praktické úlohy, kterou budu popisovat v následujících kapitolách. Proto tímto teoretická část končí a následuje popis úlohy.

3 Rozbor úlohy

3.1 Úvod

Zadáním je navrhnout a implementovat MAS pro řízení pohybujících se objektů v dvourozměrném prostoru (např. kosmických lodí), použít prioritní struktury a vše implementovat v jazyce Objective-C. V této kapitole popisuji funkční princip aplikace jako celek i funkci jednotlivých agentů každého zvlášť. Kapitola je rozdělená na několik částí – prostředí, pohyb, střelba a řízení, jejichž vyjmenované pořadí odpovídá postupu při implementaci.

Multiagentní systémy jsem se rozhodl použít k řešení tohoto problému vzhledem k jejich výhodám oproti jiným metodám. Místo složitého řídícího algoritmu je pomocí MAS možné dekomponovat problém na jednotlivé entity v podobě agentů a celý program vyvinout složením několika jednoduchých částí. Díky nezávislosti jednotlivých agentů není nutná synchronizace vláken, a zároveň je možné celý systém velmi jednoduše rozšiřovat, udržovat a modifikovat, ať už úpravou stávajících agentů, nebo přidáním dalších. Vzhledem k autonomii jednotlivých agentů, jejichž interakce senzorů a aktuátorů s prostředím probíhá kontinuálně, případné zkolabováním jednoho, či více agentů, nevede k selhání funkce ostatních, ani kolapsu celé aplikace. To můžeme například pozorovat při vyřazení libovolného agenta ze systému a sledování činnosti ostatních.

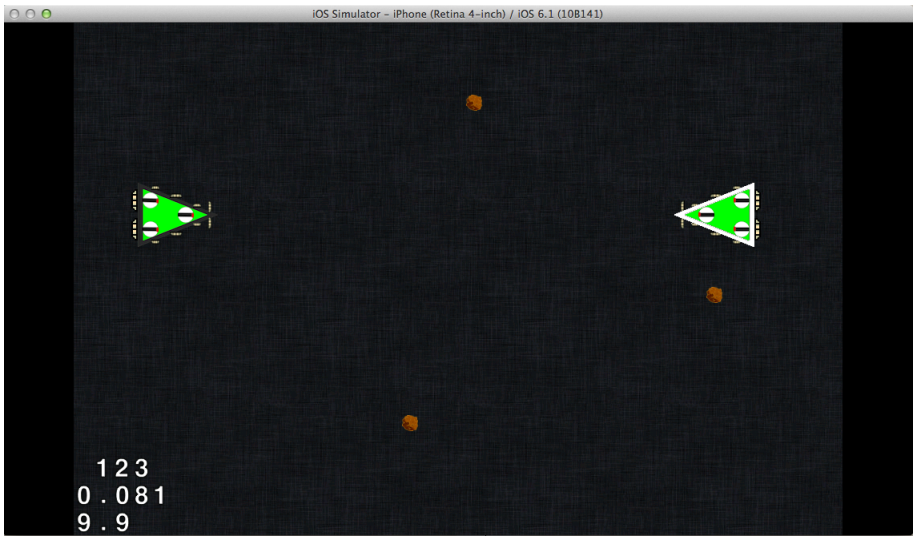
Při vytváření celého systému od začátku může být nevýhodou tohoto přístupu nutnost vytvoření celé agentové struktury z jednotlivých agentů, jejich senzorů, aktuátorů a prostředí. Na druhou stranu se však čas a práce s tím spojené v konečném důsledku vrátí díky výše uvedeným výhodám a to ať už při údržbě systému, rozšiřování, propojování několika MAS dohromady apod. V oblasti her, do které spadá i tato úloha, je například velmi výhodné, když mohu jednoduše přidat dalšího agenta zajišťujícího novou funkci, bez narušení nebo nutnosti úpravy agentů již implementovaných, nebo jednoduše přidat další loď.

Vzhledem k tomu, že je implementace provedena v anglickém jazyce,

objevují se v kapitole 3 a 4 při popisu funkce systému a jeho implementace anglické názvy odpovídající výrazům v kódu.

3.2 Vermírné lodě ve 2D vesmíru

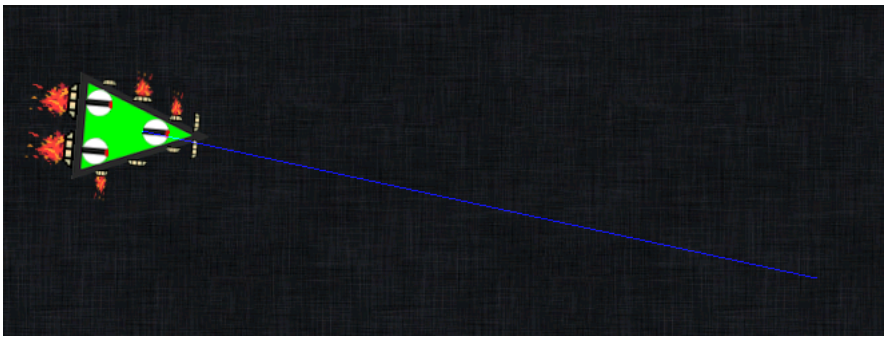
V demo aplikaci se nachází dvě vesmírné lodi vybavené několika motory pro pohyb a turrety (střílnami) viz obrázek 3.1. Kromě lodí se zde nacházejí také meteority komplikující pohyb v prostoru.



Obrázek 3.1: Snímek obrazovky demo aplikace SpaceShips

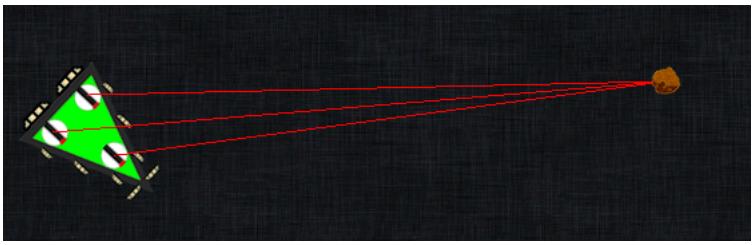
První loď (vlevo) je ovládána uživatelem, který může určovat její pohyb pomocí jednoduchého kliknutí na místo, na které se loď má přesunout. Přesun je signalizován modrou přímkou viz obrázek 3.2. Ta pak přesun sama řídí, dokud se na určené místo nedostane. Během přesunu dokáže detekovat nebezpečí kolize a upravit dráhu letu tak, aby se objektu vyhnula.

Pomocí dvojitého kliknutí nastaví hráč cíl palby, čímž aktivuje turrety a loď pak sama pálí do zaměřené oblasti, dokud dvojklikem na stejné místo není střelba zastavena. Zaměření je znázorněno červenými přímkami pro každý turret zvlášť viz obrázek 3.3. Pokud je dvojitým kliknutím označen nějaký objekt, loď si hlídá, aby bylo neustále zaměřeno na jeho aktuální



Obrázek 3.2: Loď zahajující přesun na místo určení

pozici a ostřeluje ho.



Obrázek 3.3: Turrety signalizují zaměření a zahájí natáčení směrem k zaměřené oblasti

Druha vesmírná loď je plně autonomní, od uživatelské lodi se tedy liší přidaným autonomním řízením, na základě kterého sama rozhoduje, kde se chce nacházet a zda bude nějaký objekt ostřelovat.

V dalších částech této kapitoly podrobněji popisuji vlastnosti lodi, jejich výše nastíněné chování a vysvětluji, jak je vše realizováno pomocí agentové struktury.

3.3 Prostředí

Velmi důležitou roli v celém MAS hraje agent prostředí – EnvironmentalAgent, který udržuje reprezentaci modelovaného vesmíru a umožňuje běh agentů. Modelovaný vesmír má nastavitelné různé vlastnosti jako např. tření nebo gravitaci. Vzhledem k vlastnostem reálného vesmíru jsem na-

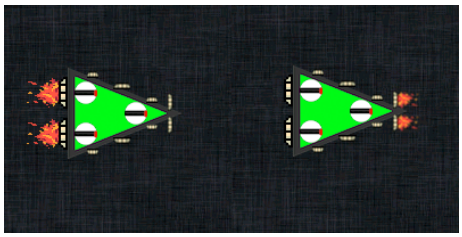
stavil tření i gravitaci jako velmi malé. Do EnvironmentalAgent jsou přidány všechny objekty – lodě s motory, turrety a také meteority. Jednotlivé agenty se mohou pomocí senzorů dotazovat na aktuální stav vesmíru a pomocí aktuátorů prostředí měnit. EnvironmentalAgent zároveň v každém svém kroku zaktualizuje o jeden krok všechny své součásti – prvky modelovaného vesmíru.

3.4 Pohyb

Řízení pohybu je rozděleno podle jednotlivých složek pohybu (dopředná, úhlová a boční), aby manévrování kosmické lodi vypadalo reálně a odpovídalo skutečnosti, zároveň ho tímto způsobem mohu dekomponovat mezi jednotlivé pohybové agenty – DirectMovementAgent (agent přímého pohybu), RotationMovementAgent (agent rotačního pohybu), SideMovementAgent (agent bočního pohybu) a dále také StoppingAgent (agent pro zastavení) a CollisionAvoidanceAgent (kolizní agent). Každá loď má nastavitelné zrychlení všech složek pohybu, díky kterým je možné měnit vlastnosti jejího pohybu. Senzorem i aktuátorem všech pohybových agentů je SpaceShip (vesmírná loď).

3.4.1 Přímý pohyb

Řízením složky přímého pohybu je v této úloze myšlena změna rychlosti ve směru podélné osy lodi. Kladnou změnu rychlosti v tomto směru, směr od zádi k přídi, zajišťují dva zadní motory, brždění pak dva motory na přídi otočené opačným směrem. Činnost motorů pro přímý pohyb je znázorněna na obrázku 3.4.



Obrázek 3.4: Znázornění funkce motorů pro přímý pohyb

Přímý pohyb řídí DirectMovementAgent (agent přímého pohybu). V případě, že je zahájen přesun na jiné místo, DirectMovementAgent si v každém kroku ze senzoru načte aktuální velocityVector (vektor rychlosti lodi), rotationVector (vektor natočení lodi), spaceshipPosition (pozice lodi), transferPoint (místo přesunu) a mainAcceleration (maximální zrychlení lodi ve směru hlavní osy) a na základě těchto načtených dat vypočítá a následně nastaví pomocí aktuátoru mainVelocityChange (aktuální zrychlení ve směru hlavní osy lodi). Nastavované zrychlení mainVelocityChange je buď rovno mainAcceleration s kladným znaménkem, pokud je třeba zrychlovat, nebo se záporným znaménkem v případě brždění. Pokud je loď na místě určení, tak je nulové.

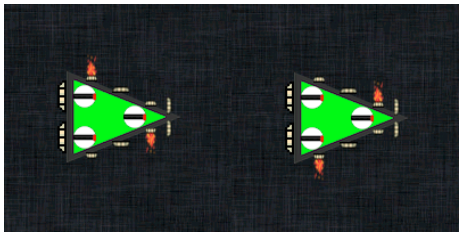
Ve výpočtu DirectMovementAgent pracuje s každou složkou vektoru rychlosti zvlášť a počítá, jakou vzdálenost na ose x a na ose y by loď urazila, kdyby v daný moment začala maximálně brzdit se zachováním stejného natočení. Získá tak xBrakingTrajectory (brzdná dráha ve směru osy x) a yBrakingTrajectory (brzdná dráha ve směru osy y), které porovná se vzdáleností do místa určení ve směru jednotlivých os, tedy xDistToTarget a yDistToTarget. Pokud jsou obě brzdné dráhy xBrakingTrajectory a yBrakingTrajectory kratší, než dané vzdálenosti xDistToTarget a DistToTarget vždy ve stejném směru, tak DirectMovementAgent dále zrychluje, tedy nastaví mainVelocityChange = mainAcceleration. V opačném případě brzdí, tedy pomocí aktuátoru nastaví lodi mainVelocityChange = –mainAcceleration. Toto platí pro pohyb v prvním kvadrantu, pro ostatní kvadranty jsou v rovnicích upravená znaménka. V algoritmu DirectMovementAgent používá v každém kroku ve výpočtech fyzikálních vztahů 3.1 a 3.2, kde s je dráha, a je zrychlení a t je čas.

$$s = \frac{1}{2} a \cdot t^2 \tag{3.1}$$

$$a = \frac{v}{t} \tag{3.2}$$

3.4.2 Rotační pohyb

Vzhledem k tomu, že zadní, ani přední motory nejsou natočitelné, nedokáží otáčet lodí. Tohoto pohybu dosahuji čtyřmi menšími bočními motory vepředu i vzadu na obou stranách lodi, které zažehají vždy do kříže podle směru požadované změny rotace viz obrázek 3.5.



Obrázek 3.5: Znázornění funkce motorů pro rotační pohyb

Rotační pohyb je sice znázorněn zmíněnými čtyřmi motory, v algoritmu rotačního agenta RotationMovementAgent je však jejich funkce pro zjednodušení reprezentována jedním rotačním zrychlením. Cílem rotačního agenta je udržovat loď natočenou přídí k cílovému destinaci. Veličiny, které agent načítá ze senzoru a používá k výpočtům jsou spaceshipAngle (úhel natočení lodi), spaceshipAngleVelocity (úhlová rychlost lodi), již zmíněné spaceshipPosition a transferPoint a zrychlení nás při řízení rotačního pohybu zajímá úhlové – spaceshipAngleAcceleration. Podobným způsobem jako DirectMovementAgent porovnával brzdnou dráhu se vzdáleností k cíli, porovnává RotationMovementAgent dva vypočtené časy linearStoppingTime (čas potřebný pro zastavení rotace za konstantního záporného zrychlení na úhlové dráze rovné rozdílu aktuálního a požadovaného úhlu) a possibleStoppingTime (čas potřebný k zastavení rotace s daným zrychlením spaceshipAngleAcceleration). Tyto dva časy RotationMovementAgent porovná a pokud je possibleStoppingTime menší, než linearStoppingTime, tak RotationMovementAgent nastaví lodi pomocí aktuátoru kladné úhlové zrychlení angleVelocityChange ve směru aktuální rotace o velikosti spaceshipAngleAcceleration, v opačném případě brzdí rotační pohyb lodi nastavením úhlového zrychlení angleVelocityChange o velikosti spaceshipAngleAcceleration proti směru aktuální rotace. Rotation-

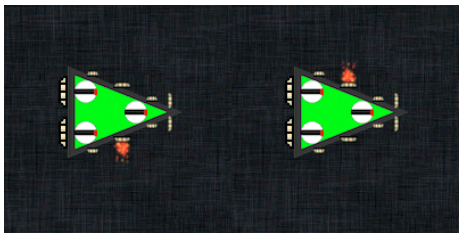
MovementAgent ve své architektuře využívá k výpočtu v každém tiku fyzikální vztahy 3.3 a 3.4, kde ω je úhlová rychlost, ε je úhlové zrychlení, α je úhlová dráha a t je čas.

$$\omega = \varepsilon \cdot t \tag{3.3}$$

$$\alpha = \omega \cdot t \tag{3.4}$$

3.4.3 Boční pohyb

Poslední složkou pohybu je boční korekce směru, kterou obstarávají boční motory působící kolmo k podélné ose lodi. Činnost motorů pro boční korekci směru vidíme na obrázku 3.6.



Obrázek 3.6: Znázornění funkce motorů pro boční pohyb

Pomocí agentů DirectMovementAgent a RotationMovementAgent je již loď schopná přesunu z místa na místo, nicméně bez přidání dalšího agenta je pohyb lodi značně neohrabaný. Například pokud již loď letí s nenulovým vektorem rychlosti a my změníme místo určení, nedokáže se dobře vypořádat se změnou směru, resp. s vynulováním původního vektoru rychlosti. Výsledkem je spirálovitý pohyb okolo místa určení. V případě prostředí s větším třením by původní složka pohybu sama odezněla, nicméně ve vesmíru setrvává dokud není vyrušena jinou silou. Tento problém by bylo možné vyřešit přidáním řídícího pohybového agenta, který by nejdříve natočil loď proti původnímu směru letu, v tomto směru loď zastavil a až poté zahájil akceleraci ve směru k místu určení, ke kterému by již loď dorazila přímo. Případně by upravoval natočení lodi po celou dobu tak, aby

změna směru proběhla plynule. Mnohem jednodušší a podstatně efektivnější řešení je přidání agenta SideMovementAgent zajišťujícího boční korekci směru pomocí motorů na obrázku 3.6. SideMovementAgent si ze senzoru načítá spaceshipPosition, transferPoint, spaceshipVelocityVector a sideAcceleration. Ve svém algoritmu agent porovnává úhel vektoru rychlosti spaceshipVelocityVector s úhlem, kterým by loď měla letět, aby se pohybovala směrem k určenému místu transferPoint. Následně pak koriguje směr letu, tedy úhel vektoru spaceshipVelocityVector tak, aby byl shodný s požadovaným úhlem letu. Tato korekce je realizována pomocí aktuátoru nastavujícího v každém kroku sideVelocityChange na hodnotu bočního zrychlení sideAcceleration buď s kladným, nebo záporným znaménkem podle požadovaného směru korekce. Pokud loď letí správným směrem, je sideVelocityChange = 0 a agent vyčkává, dokud se loď od správného směru letu opět neodchýlí. Výpočet požadovaného úhlu letu loď provádí z načtených spaceshipPosition a transferPoint pomocí goniometrických funkcí.

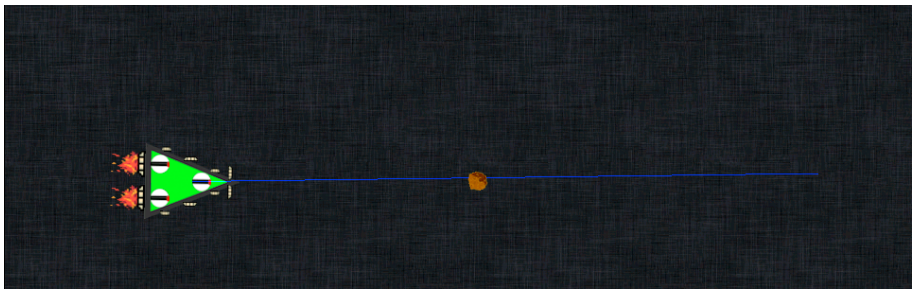
3.4.4 Zastavení

Součástí pohybového mechanismu je také StoppingAgent, který zakročí ve chvíli, kdy loď dorazí na místo určení. Pomocí aktuátoru nastaví lodi stav moving (moving = YES – loď se pohybuje, moving = NO – loď se nepohybuje) na hodnotu NO, čímž vyřadí z funkce motory a loď zůstane stát, dokud pomocí kliknutí nebo autonomního řídícího agenta nedostane pokyn k dalšímu pohybu.

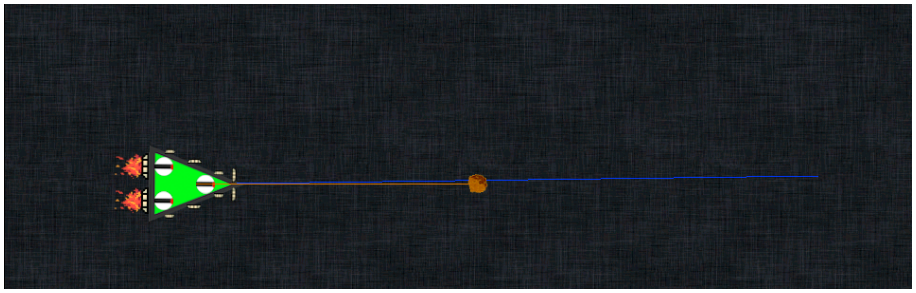
3.4.5 Zabraňování kolizím

V rámci pohybu je vesmírná loď schopná nejen detekovat nebezpečí kolize, ale také se meteoritům vyhnout. Detekci kolizí zajišťuje senzor SpaceShip pomocí jeho součásti Radar, kterým jsou obě lodi vybavené. Radar má omezený dosah, a proto loď při zahájení pohybu může nejprve vyrazit směrem k meteoritu a nebezpečí kolize je zaznamenáno až v určité blízkosti. V momentě rozpoznání kolizního nebezpečí je aktivován CollisionAvoidanceAgent (kolizní agent), který dočasně převezme řízení směru

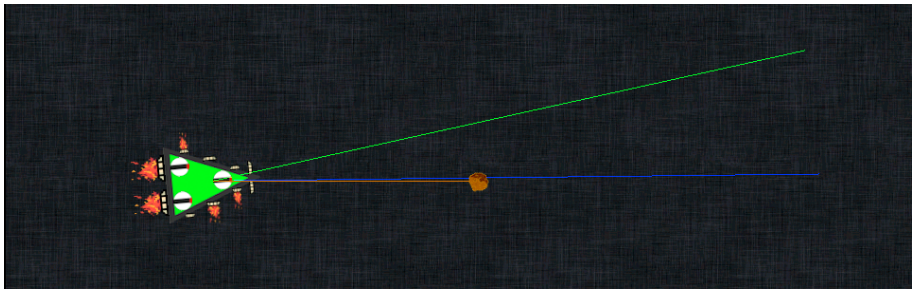
pohybu lodi. Využívá při tom všech výše zmíněných pohybových agentů, upraví však požadovaný směr letu tak, aby se loď meteoritu bezpečně vyhnula, a po zažehnání nebezpečí kolize přestane CollisionAvoidanceAgent do řízení pohybu zasahovat a loď se vrátí k přesunu na původní místo určení. Tento průběh řešení kolizních situací a zároveň funkci všech již zmíněných agentů ilustrují dobře obrázky 3.7 až 3.14.



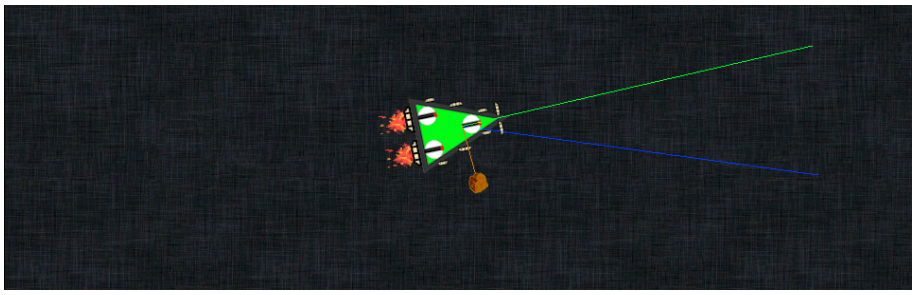
Obrázek 3.7: Loď zahajující pohyb směrem na meteorit



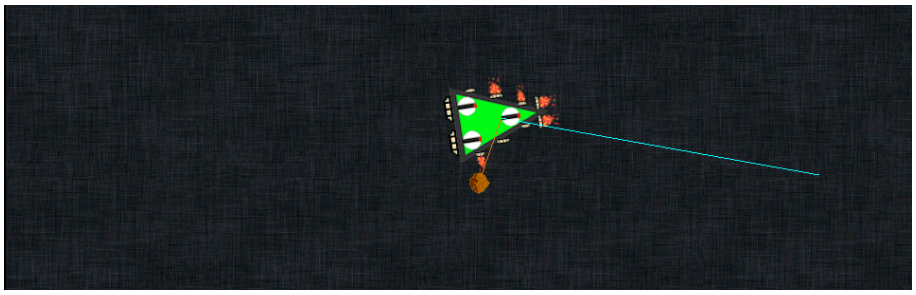
Obrázek 3.8: Radar lodi detekuje nebezpečí kolize



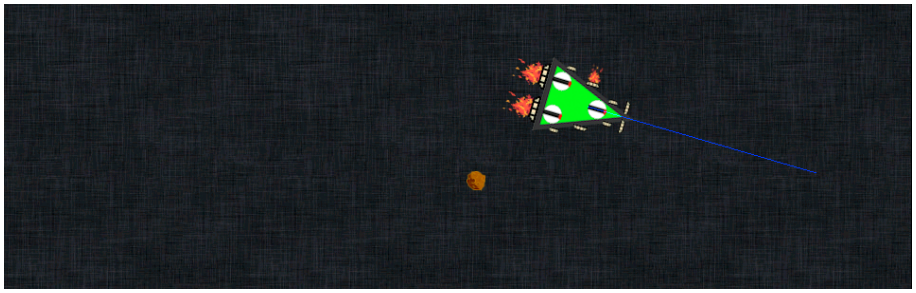
Obrázek 3.9: Loď zahajuje úhybný manévr



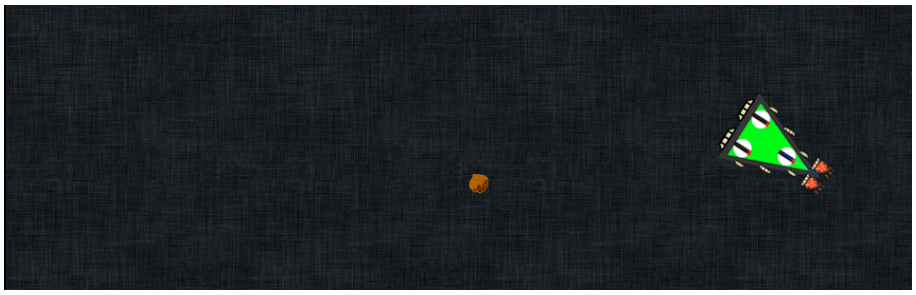
Obrázek 3.10: Loď míjí meteorit, kterému se úspěšně vyhnula



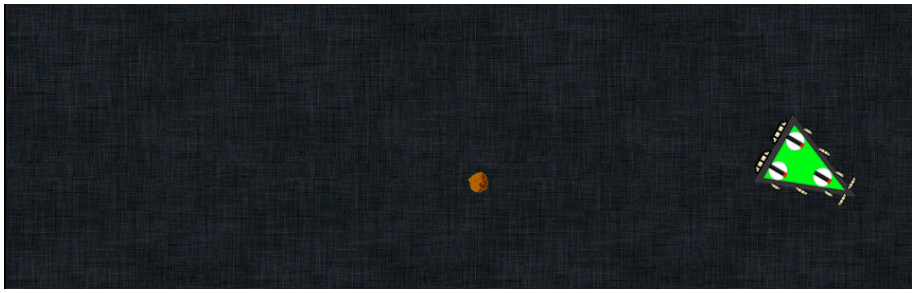
Obrázek 3.11: Nebezpečí kolize pominulo, loď se stáčí směrem k původnímu místu určení



Obrázek 3.12: Loď již směřuje k místu určení



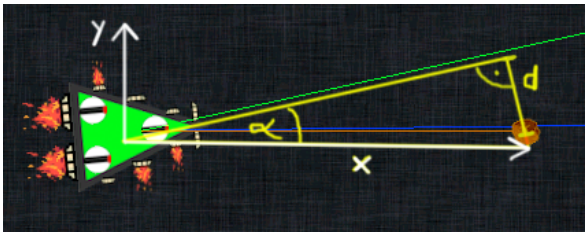
Obrázek 3.13: Loď dobržďuje



Obrázek 3.14: Loď zastavila na místě určení

CollisionAvoidanceAgent načítá ze senzoru následující data – spaceshipPosition, spaceshipVelocityVector, desiredTransferPoint (původní požadované místo určení, které je během řešení kolize uchováno, aby k němu mohla loď po pominutí kolizního nebezpečí doletět), collidingObjectPosition (pozice meteoritu, který se nachází v dráze letu a hrozní s ním kolize) a desiredPassingDistance (vzdálenost v jaké vzdálenosti od kolidujícího objektu má CollisionAvoidanceAgent naplánovat průlet; nastaveno jako vlastnost lodi). CollisionAvoidanceAgent na základě dat ze senzoru počítá, jaký úhel vektoru rychlosti by potřebovala loď mít, aby kolem meteoritu proletěla v požadované vzdálenosti. Znázornění problému vidíme na obrázku 3.15, d = desiredPassingDistance a α značí požadovaný úhel vektoru rychlosti. CollisionAvoidanceAgent pak počítá požadovaný úhel podle vzorce 3.5, α_0 = aktuální úhel vektoru rychlosti a d_0 = vzdálenost průletu od meteoritu pro α_0. Tento vzorec vychází z linearizace goniometrických funkcí, kterou si můžeme dovolit, protože CollisionAvoidanceAgent řeší situace pro

malá $\Delta\alpha$ a zároveň jsou jeho výpočty upravovány v každém tiku vesmíru. To potvrzuje i bezchybná práce CollisionAvoidanceAgenta, který na základě vypočteného α nastaví pomocí aktuátoru lodi nový transferPoint tak, aby výsledný úhel vektoru rychlosti odpovídal α. Ve chvíli, kdy Radar lodi přestane hlásit nebezpečí kolize viz obrázek 3.11, CollisionAvoidanceAgent přestane zasahovat do řízení a loď se vrátí k přesunu na desiredTransferPoint.

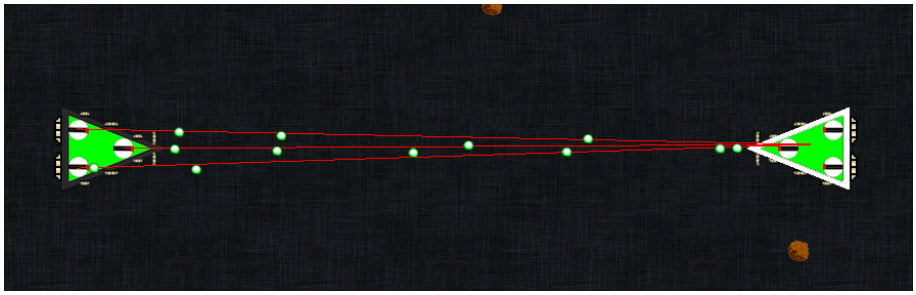


Obrázek 3.15: Znázornění veličin důležitých pro řešení kolizí

$$\alpha = \frac{d}{d_0} \cdot \alpha_0 \tag{3.5}$$

3.5 Střelba

Loď střílí pomocí tří turretů umístěných na její palubě. Turrety mají možnost natáčení o 360° a každý z nich má svého agenta na zaměřování i na střelbu. Na obrázku 3.16 vidíme jednotlivé turrety a názornou ukázku střelby.



Obrázek 3.16: Ukázka střelby hráčské lodi

TurretAimingAgent (zaměřovací agent), který se stará o zaměřování turretu,

funguje na podobném principu jako RotationMovementAgent, tedy jeho funkci detailně popisovat již nebudu. Liší se od rotačního agenta samozřejmě daty, které si ze senzorů načítá a namísto rotačního zrychlení lodi nastavuje rotační zrychlení turretu. Kromě senzoru SpaceShip, stejně jako u pohybových agentů, je senzorem pro TurretAimingAgent také Turret, který je zároveň jeho aktuátorem. V případě, že je zahájena střelba na příslušné lodi, každý TurretAimingAgent načte spaceshipAngle (úhel natočení lodi) ze senzoru SpaceShip a turretAngle (úhel natočení turretu vůči lodi), turretAngleVelocity (úhlovou rychlost turretu), turretShootingTarget (zaměřené místo pro každý turret), turretAngleAcceleration (maximální rotační zrychlení turretu) a turretPosition (pozice turretu ve vesmíru) ze senzoru Turret. Dále nastavuje pomocí aktuátoru každému turretu novou hodnotu angleVelocityChange (rotační zrychlení), aby udržoval turret nasměrovaný na zaměřené místo.

TurretShootingAgent (palební agent) zajišťuje střelbu z turretu, pakliže je turret natočený v nastaveném rozptylu v okolí svého turretShootingTarget. Právě pro kontrolu správného natočení turretu, které je podmínkou střelby palebního agenta, načítá TurretShootingAgent ze senzorů SpaceShip a Turret hodnoty spaceshipAngle, turretAngle, turretShootingTarget a turretPosition.

Lodím při zásahu střelami ubývají životy a jejich zbarvení se podle toho mění zelené až po červenou. V případě vyčerpání všech životů se loď bezvládně pohybuje vesmírem.

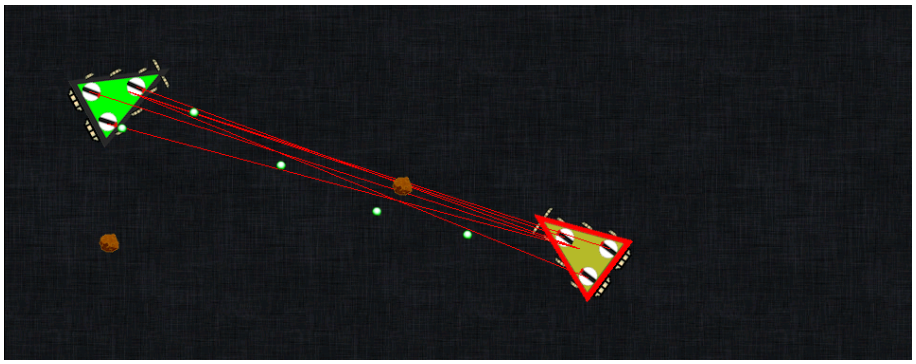
3.6 Řízení autonomní lodi

Jak již bylo zmíněno v úvodu této kapitoly, do vesmíru je přidána i druhá loď, která kromě již zmíněných agentů navíc obsahuje CaptainAgent (agenta kapitána) se schopností primitivního rozhodování mezi předem určenými modely chování. Díky tomuto agentu je loď autonomní. Senzorem i aktuátorem agenta kapitána je SpaceShip. CaptainAgent načítá ze senzoru autonomyShipLivesCount (počet životů autonomní lodi), playerShipLivesCount (počet životů hráčské lodi), autonomyShipPosition (pozice autonomní lodi), playerShipPosition (pozice hráčské lodi) a underAttack (informace o tom, zda je autonomní loď ostřelována hráčem) a livesMin (minimální počet životů pro

útok), dangerousDistance.

3.6.1 Útok

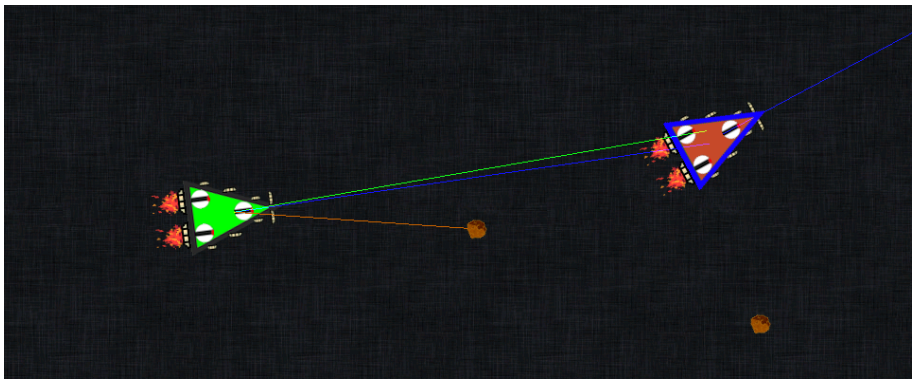
- CaptainAgent nastaví strategii útoku (Attacking) pokud:
  - autonomní loď atakována hráčem (underAttack = YES)
  - a zároveň autonomní loď má dostatek životů (autonomyShipLivesCount > livesMin) nebo hráčské lodi také docházejí životy (playerShipLivesCount < livesMin)
- zvolením strategie útoku CaptainAgent:
  - udržuje polohu autonomní lodi tak, aby měla hráčskou loď na dostřel
  - zahájí palbu a jako zaměřené místo autonomní lodi nastaví hráčskou loď
  - tuto strategii znázorňuje bílé lemování autonomní lodi viz obrázek 3.17



Obrázek 3.17: Ukázka autonomní lodi napadené uživatelem v momentě změny strategie na útok a zahájení zaměřování turretů

3.6.2 Útěk

- CaptainAgent nastaví strategii útěku (RunningAway) pokud nastanou následující podmínky:
  - autonomní lodi dochází životy (autonomyShipLives < livesMin)
  - a zároveň autonomní loď má dostatek životů (playerShipLives > livesMin)
- zvolením strategie útěku CaptainAgent:
  - nastaví autonomní lodi pomocí aktuátoru transferPoint tak, aby se pohybovala opačným směrem, než se nachází hráčská loď
  - pokud je hráčská loď na dostřel a stále útočí, pokračuje autonomní loď v palbě a jako zaměřené místo udržuje hráčskou loď
  - tuto strategii znázorňuje modré lemování autonomní lodi viz obrázek 3.18



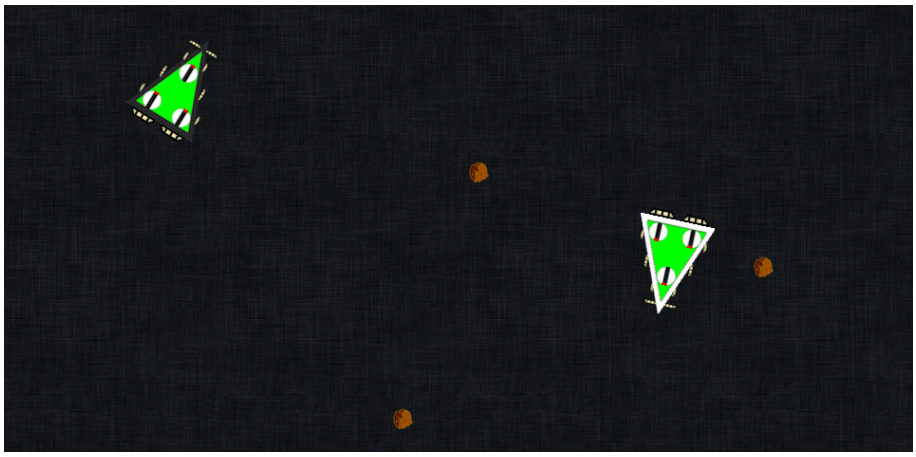
Obrázek 3.18: Ukázka autonomní lodi s nízkým počtem životů během strategie útěk pronásledované hráčskou lodí, která právě detekuje nebezpečí kolize

3.6.3 Vyčkávání

- CaptainAgent nastaví strategii vyčkávání (WaitingForAction) pokud:

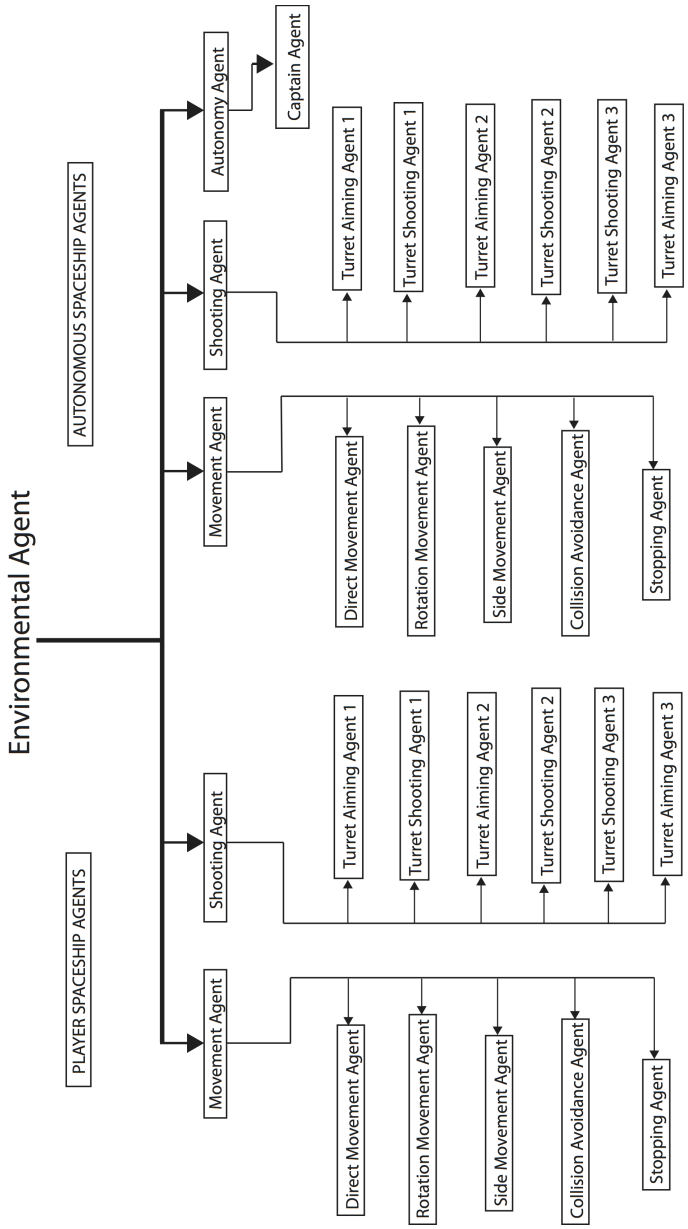
  - hráčská loď přeruší útok (underAttack = NO)
  - a zároveň hráčská loď se nenachází v nebezpečné zóně (vzdálenost lodí není menší, než dangerousDistance)

- zvolením strategie vyčkávání CaptainAgent:
  - přeruší palbu autonomní lodi
  - udržuje autonomní loď z dostřelu hráčské lodi
  - tuto strategii znázorňuje bílé lemování autonomní lodi viz obrázek 3.19



Obrázek 3.19: Ukázka autonomní lodi během strategie vyčkávání

3.7 Schéma celé agentové struktury



Obrázek 3.20: Schéma agentové struktury

4 Implementace

V této předposlední kapitole doplňuji detailnější informace týkající se implementace. Pro implementaci multiagentních systémů existuje řada vývojových prostředí jako např. JADE [8], MASON [9] nebo NETLOGO [10], které jsou podrobněji rozebrány v [7].Rozhodl jsem se však implementovat celý systém bez použití podobných nástrojů a vytvořit vlastní fungující systém od začátku. Zde se proto dále zmiňuji o tom, na jakém principu systém funguje, jakým způsobem je zajišťována funkce agentů a aktualizace vesmíru, jak jsou agenti uvnitř realizovaní a jakým způsobem jsou přidáváni do systému.

4.1 Objective-C

Implementace je provedena v jazyce Objective-C ve vývojovém prostředí Xcode s použitím herního frameworku Cocos2D a fyzikálního frameworku Chipmunk. Objective-C, často nazývaný ObjC, je objektově orientovaný programovací jazyk implementovaný jako rozšíření jazyka C. V současné době se používá převážně v operačních systémech Mac OS X, iOS.

4.2 Agentový systém

4.2.1 Prostředí a jeho role

Velmi důležitou roli celého systému hraje prostředí. V tomto případě se jedná o specializovaného agenta prostředí (EnvironmentalAgent) přidaného do třídy SpaceLayer. SpaceLayer slouží k reprezentaci modelovaného vesmíru a spolu s agentem prostředí zajišťují chod celého systému. Reprezentace je diskrétní a jeden krok vesmíru odpovídá jednomu zavolání metody update ve SpaceLayer. Zavoláním update metody ve třídě SpaceLayer se zavolá update v agentu prostředí a ten zaktualizuje všechny agenty, které obsahuje.

4.2.2 Vrstvy MAS

Celý MAS je rozdělený do několika vrstev. Agenty jsou sdružené do skupin podle funkce v tzv. kontejnerových agentech, které dědí od třídy LayeredAgent a jsou obsažené v agentu prostředí. Pohybové agenty jsou sdružené v MovementAgent, střílecí agenti v ShootingAgent a CaptainAgent je zatím samotný řídící agent v AutonomyAgent, který implementuje pouze autonomní loď. ShootingAgent, MovementAgent a AutonomyAgent sdružující ostatní agenty se opět starají o jejich update a to v pořadí podle priority. Jednotlivé agenty mají uložené ve speciálním seznamu, ze kterého vyvolávají update jednoho po druhém v pořadí, v jakém jsou do seznamu přidaní. Zároveň je možné agentům nastavovat prioritu nezávisle na pořadí v seznamu a kontejnerové agenty pak mohou zohledňovat nastavenou prioritu před pořadím v seznamu. Pořadí agentů v seznamu jsem volil podle důležitosti, tedy například CollisionAvoidanceAgent má větší prioritu, než ostatní pohybové agenty.

4.2.3 Nastavení agentů

Celý agentový systém je uložený v EnvironmentalAgent, agenty patřící k jednotlivým lodím jsou však nastavovány pro každou loď v její metodě setupAgents. Ta přidá do systému agenty, které potřebuje ke svému řízení. Tento přístup je zvolen především pro jednoduché přidávání dalších lodí do vesmíru tak, že každá loď si automaticky zaregistruje své vlastní agenty.

4.2.4 Struktura agentů

Každý agent má svojí update metodu volanou z update metody jeho kontejnerového agenta. V update metodách jednotlivých akčních agentů se určuje, zda agent chce, či nechce provézt akci. U agentů DirectMovementAgent, RotationMovementAgent a SideMovementAgent je například při volbě agenta, zda chce akci provézt, rozhodující, jestli je loď vůbec ve stavu pohybu. Například CollisionAvoidanceAgent však kromě toho především vyhodnocuje v update metodě radarová data o riziku kolize a o akci se přihlásí

v případě kolizního nebezpečí. Pokud agent v update metodě vyhodnotí, že chce provézt akci, zavolá se metoda performAction, kterou také implementují všechny akční agenty, a z ní se již dále volají algoritmické metody agentů a na základě z nich získaných hodnot se z metody performAction konají akce pomocí aktuátorů.

4.2.5 Cocos2D & Chipmunk

Stejně tak jako agenty, i vesmírné objekty a lodi jsou ve struktuře kódu reprezentovány jednotlivými třídami. Třídy objektů dědí od třídy PhysicsSprite z knihovny Cocos2D, která umožňuje snadnější práci s objekty v rámci simulace. Pro každý z objektů je tak možné nastavovat například hmotnost, tvar, velikost atd., což následně ovlivňuje jejich chování při simulaci a pohyb působí, také díky fyzikálnímu frameworku Chipmunk, reálně.

5 Závěr

Úspěšně byla naimplementována multiagentní demo aplikace ve vývojovém prostředí Xcode v jazyce Objective-C s využitím herního frameworku Cocos2D a fyzikálního frameworku Chipmunk. Vyvinutý multiagentní systém pomocí reaktivních pohybových agentů řídí pohyb vesmírných lodí ve 2D vesmíru. Lodě se dokáží přesouvat na místo určení a během letu detekovat nebezpečí kolize, kterým se následně vyhnou úpravou letové dráhy. Pomocí střílecích agentů atakují protihráče střelbou z turretů a ubírají mu tak životy. Uživatel může jednu z lodí řídit, příčemž druhá loď s přidaným autonomního řízení zastává roli oponenta. Dekompozicí problematiky na jednotlivé části se podařilo vytvořit propracovaný systém, který je schopný autonomně fungovat a reagovat na vstupy od hráče. Díky rozdělení na jednotlivé agenty je také celý systém velmi názorný a především snadno modifikovatelný (např. doplněním dalších agentů, lodí apod.)

Příkladem možného rozšíření je např. přidání komplexnějších deliberativních agentů, které místo jednoduché podmínkové architektury přiřazující vstupům ze senzorů jednotlivé akce, budou implementovat složitější architekturu s možností uchovávání reprezentace prostředí a vnitřních stavů a se schopností učení se. Pomocí těchto agentů by bylo možné přidat umělou inteligenci oponentské lodi. Ta by pak např. na základě sledování svých i protihráčových tahů a výsledkům, ke kterým vedly, byla schopná měnit své strategie a ve výsledku napodobit chování hráče. Dále bych rád vybudoval systém zasílání zpráv, na jehož základě by mezi agenty probíhala komunikace. Ta by sloužila k informování komplexních ingeligentních deliberativních agentů, kteří by mohli takové informace vyhodnocovat a upravovat podle nich své následující kroky. Příkladem takového počínání je informování deliberativního řídícího agenta o docházející munici v některých turretech. On pak může, pokud není zaneprázněný něčím jiným, takové zprávy zpracovat a upravit podle nich volbu svých kroků vedoucích k cíli, který plní. Stejně tak by mohli komunikovat i reaktivní agenti mezi sebou, kde by například taková komunikace mohla sloužit ke kooperaci

„shooting agents" a domluvě, který turret bude mířit na jakou část atakovaného objektu.

Možných rozšíření implementovaného MAS se dá najít celá řada, a proto bych se touto problematikou rád zabýval dále a rozvinul implementovaný multiagentní systém v rámci diplomové práce, která může být zaměřená například na interakci mezi agenty a jejich schopnost učení.

Literatura

[1] KUBÍK, Aleš: *Inteligentní agenty: Tvorba aplikačního softwaru na bázi multiagentových systémů, 1. vydání,* Brno: Computer Press, 2004, 280 s., ISBN 80-251-0323-4

[2] *Wikipedia – The Free Encyclopedia: Multi-agent system,* [online] 3.4.2013, dostupné z WWW: `http://en.wikipedia.org/wiki/Multi-agent_system`

[3] DOUBRAVOVÁ, Hana: *Využití teorie her při řešení konfliktiních situací*, bakalářská práce, České Budějovice, Jihočeská univerzita v Českých Budějovicích, Ekonomická fakulta, 2007

[4] SHOHAM, Yoav; LEYTON-BROWN, Kevin: *Muliagent Systems: Algorithnic, Game-Theoretic, and Logical Foundations*, New York: Cambridge University Press, 2009, ISBN 978-0-521-89943-7

[5] *OLLSON, Fredrik: A Multi-Agent System for playing the board game Risk*, diplomová práce, Švédsko, 2005, Blekinge Institute of Technology, vedoucí práce Stefan Johansson

[6] *KQML as an Agent Communication Language,* [online] 17.4.2013, Dostupné z WWW: `http://www.csee.umbc.edu/csee/research/kqml/papers/kqmlacl.pdf`

[7] *ŠMAJCL, Jan: Vývojová prostředí pro modelování multi-agentních systémů, bakalářská práce, České Budějovice, 2008, Jihočeská univerzita v Českých Budějovcích*, [online]. 14.5.2013, dostupné z WWW: `http://theses.cz/id/jh2lj4/downloadPraceContent_adipIdno_9379`

[8] *TILAB: Java Agent DEvelopment Framework,* [online] 14.5.2013, dostupné z WWW: `http://jade.tilab.com/`

[9] *MASON,* [online], 14.5.2013, Dostupné z WWW: `http://cs.gmu.edu/~eclab/projects/mason/`

[10] *NetLogo* [online], 14.5.2013, Dostupné z WWW: http://ccl.northwestern.edu/netlogo/

[11] *Wikipedia – The Free Encyclopedia: Cocos2d,* [online] 14.5.2013, dostupné z WWW: http://en.wikipedia.org/wiki/Cocos2d

[12] *CHIPMUNK: Game Dynamics,* [online] 14.5.2013, dostupné z WWW: http://chipmunk-physics.net/

A Obsah přiloženého CD

- text bakalářské práce ve formátu pdf
- zdrojový kód spustitelný ve vývojovému prostředí Xcode pomocí iOS simulátoru