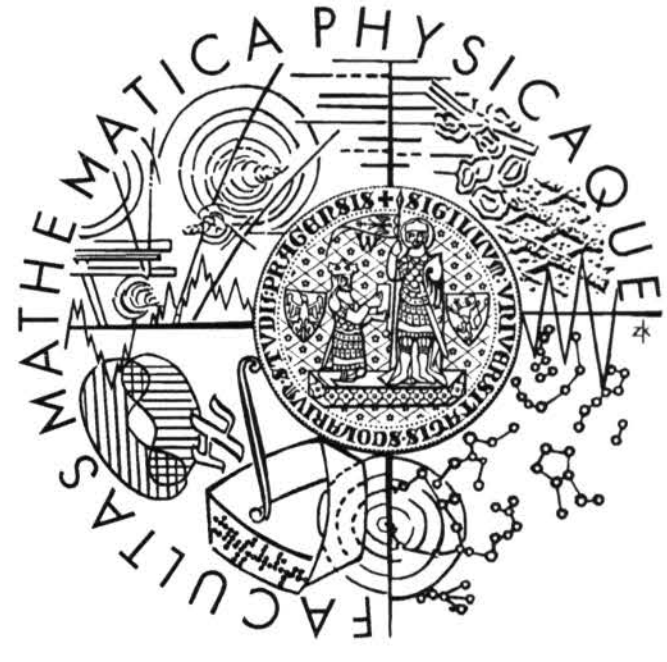
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jiří Kroužil

HTN plánování do akčních her

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Cyril Brom, Ph.D.
Studijní program: Informatika, Programování

2009

Děkuji panu Mgr. Cyrilu Bromovi, Ph.D. za vedení a množství cenných rad, které mi poskytoval při práci na tomto projektu. Dále děkuji Radimu Krupičkovi za pomoc při korektuře práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 7. srpna 2009

Jiří Kroužil

OBSAH

Název práce: HTN plánování do akčních her

Autor: Jiří Kroužil

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Cyril Brom, Ph.D.

e-mail vedoucího: brom@ksvi.mff.cuni.cz

Abstrakt: V této práci se zaměřujeme na návrh mechanismu postaveného na zkombinování POSH a HTN technik plánování, který ovládá skupinu agentů v herním světě. Mechanismus využívá POSH plánovač k výběru týmové akce závislé na aktuálním scénáři hry. HTN plánovač se používá k organizování taktických manévrů, které praktikují skutečné speciální jednotky v bojích v uzavřených prostorech. Tento typ manévrů je závislý na využívání pozičních vlastností prostředí. Proto je zde navržena struktura reprezentující svět, která disponuje detailním popisem pozičních vlastností. Tato struktura by byla ale zbytečná, pokud by neexistoval algoritmus schopný vyhledávat cestu zadanou pozičními vlastnostmi. Proto tato práce zahrnuje návrh algoritmu, který hledá cestu zadanou formou pozičních vlastností.

Klíčová slova: HTN plánování, taktická navigace, taktické HTN plánování, hybridní plánování

Title: HTN planning for opponents in computer games

Author: Jiří Kroužil

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Cyril Brom, Ph.D.

Supervisor's e-mail address: brom@ksvi.mff.cuni.cz

Abstract: In this work we concentrate on design of a mechanism built on combination of POSH and HTN planning techniques which controls a squad of agents in a game world. The mechanism uses POSH planner to team action selection dependent on actual game scenario. HTN planner is applied to organize tactical maneuvers which real special forces execute in close quarter battles. This kind of maneuvers depends highly on exploiting of world positional properties. And because of that the structure which represents game world and provides many positional properties is designed here. But the structure would be worthless unless an algorithm capable of path finding based on positional properties existed. That's why this work includes design of algorithm which searches for path defined in form of positional properties.

Keywords: HTN planning, tactical navigation, tactical HTN planning, hybrid planning

Úvod

Cílem této bakalářské práce je zkombinovat reaktivně plánovací mechanismus POSH a HTN plánování v jeden hybridní plánovač, s jehož pomocí bude centrálně řízena skupina agentů (herní oponenti) z akčních počítačových her. Doposud jsme se setkávali na poli akčních her spíše s agenty řízenými převážně jednoduchými konečnými stavovými automaty, zřídka byl použit plánovač postaven na hlubším teoretickém základu. Herní veřejnosti představil Jeff Orkin agenty ve hře F.E.A.R.©, pro jejichž chování byla využita plánovací architektura GOAP (Goal-Oriented Action Planning, více v [3]) zakomponovaná do jednoduchého STRIPS plánovače. V tomto případě se ale jednalo spíše o řízení jednotlivců. Čistě na akademické půdě zapsal Héktor Muñoz-Avila herní strategii pro mód Domination hry Unreal Tournament© pomocí HTN plánů, které už koordinovaly vetší skupinu agentů.

V této práci se zaměřím zejména na využití vlastností HTN plánování při taktické navigaci čtyřčlenné skupiny agentů ve virtuálním 2D světě. Jako podklady pro taktickou navigaci budou použity techniky vstupů a průchodů obytných částí (Close Quater Battle tactics - CQB) speciálních policejních jednotek jako je např. SWAT (Special Weapons and Tactics). HTN plánovač bude tedy zajišťovat přesun skupiny agentů mezi dvěma obytnými jednotkami (pokoj, chodba), zatímco úkolem POSH plánovače bude řešení aktuálních cílů pro celou skupinu v závislosti na konkrétním scénáři hry.

Simulátor pro tento projekt implementuji v programovacím jazyce Java. Řídící jednotka využívající výše zmíněného POSH – HTN plánovače bude navržena (jako prototyp) s ohledem na budoucí zakomponování do skutečné počítačové hry Unreal Tournament 2004© (UT2004) z dílny studia Epic Games, která nabízí rozhraní pro externí ovládání herních agentů (botů). S tím souvisí i výběr herního scénáře (modifikace, zkráceně módu). UT2004 nabízí mnoho různých herních scénářů. Pro tento projekt jsem zvolil boj o vlajku (Capture The Flag – CTF), protože svými vlastnostmi a dostupností již napsaných ovládacích prvků nejlépe odpovídá požadavkům pro simulaci taktiky pro boj v uzavřených prostorech.

První kapitola poslouží jako formální popis plánování a přehled současných plánovacích technik, které jsou použité nebo zmíněné v této práci. Kapitola č. 2 se věnuje detailnějšímu popisu cíle této práce. V kapitole č. 3 je celý návrh analyzován a veškeré techniky detailně popsány. Kapitola č. 4 se věnuje popisu implementace simulačního programu. V páté kapitole jsou prezentovány výsledky experimentů ze simulačního programu. Šestá kapitola obsahuje závěr shrnující celou práci.

1 Plánování a současné techniky

Obecně je plánování proces, který vyhledává posloupnost akcí vedoucí z výchozího stavu světa do cílového. V následujících podkapitolách je formálně popsáno klasické plánování včetně klasické reprezentace (použité techniky budou vycházet z této teorie), část je věnována HTN plánování a nakonec je popsán princip POSH mechanismu. K některým typům plánování jsou zároveň uvedeny příkladné implementace s krátkým popisem. Formální popis je čerpán z [2] a [4].

1.1 Klasické plánování

1.1.1 Klasická reprezentace

Klasická reprezentace pro plánování využívá predikátovou logiku pro zápis stavů a akcí. Mějme konečnou množinu predikátových symbolů a konstant bez funkčních symbolů a označme ji L (jazyk). Logický atom v této reprezentaci odpovídá predikátovému symbolu s argumenty, např. *atLocation(location1, thing1)*. Navíc je možné používat proměnné.

Definice stavu světa

Stav světa je množina instanciovaných atomů (bez proměnných). Pravdivostní hodnota některých atomů se může lišit v závislosti na stavu, pak takový atom nazýváme flexibilní (fluent). Jsou ale i atomy, které mají stejnou pravdivostní hodnotu v jakémkoliv stavu, tzv. neměnné atomy (rigid). Pokud nějaký atom není ve stavu explicitně uveden, tak neplatí (předpoklad uzavřeného světa).

Operátory a akce

Pravdivostní hodnotu atomů v konkrétních stavech mohou měnit operátory. Plánovací operátor je definován jako o = (name(o), precond(o), effects(o)), kde:

- name(o): jméno operátoru ve tvaru $n(x_1, \ldots, x_k)$, kde n je jednoznačný symbol operátoru a $x_1, \ldots, x_k$ jsou symboly proměnných, které se mohou objevit kdekoliv v daném operátoru,
- precond(o) alias předpoklady: literály, které musí být splnitelné, aby bylo možné operátor použít,
- effects(o) alias efekty: literály, které se stanou pravdivé aplikací operátoru.

Nechť je S množina literálů, pak S^+ značí pozitivní atomy z S a S^- značí atomy, jejichž

negace je v S. Akce jsou plně instanciované operátory. Pokud a je akce a s stav s vlastnostmi $precond^+(a) \square s$ a $precond^-(a) \cap s = \square$, potom je akce a aplikovatelná na stav s a výsledkem je stav $\gamma(a,s) = (s - effects^-(a)) \square effects^+(a)$.

Plánovací doména

Nechť L je jazyk a O množina operátorů. Plánovací doména Σ nad jazykem L a s operátory O je trojice (S,A,γ), kde:

- stavy $S \square \mathcal{P}(\{$všechny instanciované atomy nad L$\})$, kde $\mathcal{P}$ značí potenční množinu,
- akce A = {všechny instanciované operatory z O nad L},
- přechodová funkce γ: $\gamma(a,s) = (s - effects^-(a)) \square effects^+(a)$, je-li a použitelná na s,
- S je uzavřená vzhledem k γ, tzn. pro každou akci a, která je použitelná na stav s, platí $\gamma(a,s) \square S$.

Plánovací problém

Plánovací problém P je definován trojicí (Σ,s_0,g), kde:

- Σ je plánovací doména,
- s_0 je počáteční stav, $s_0 \square S$,
- g je množina instanciovaných literálů
  - stav s splňuje g právě tehdy, když $g^+ \square s$ a $g^- \square s = \square$,
  - $S_g = \{s \square S \mid s$ splňuje g$\}$ – množina cílových stavů.

Plánovací problém se zapisuje jako trojice (O,s_0,g).

Dosažitelnost stavů

Mějme stav s a definujme množinu jeho přímých následníků jako $\Gamma(s)=\{\gamma(s,a) \mid a \square A$ a a je aplikovatelná na s$\}$. Množina všech dosažitelných stavů ze stavu s je pak $\Gamma_\infty(s) = \Gamma(s) \square \Gamma(\Gamma(s)) \square ...$, která tvoří tranzitivní uzávěr.

Akce a je relevantní pro cíl g právě tehdy, když $g \cap effects(a) \neq \square$ a efekty akce nejsou v konfliktu s g, tzn. $g^- \cap effects^+(a) = \square$ a $g^+ \cap effects^-(a) = \square$.

Pokud je akce a relevantní pro cíl g, tak platí $\gamma^{-1}(g,a) = (g - effects(a)) \square precond(a)$, což definuje regresní (zpětnou) množinu cíle g pro akci a.

Množina všech regresivních množin přes všechny akce relevantní pro cíl g je definována

jako $\Gamma^{-1}(g) = \{\gamma^{-1}(g,a) \mid a \in A$ a a je relevantní pro g$\}$. Cílový stav g je dosažitelný v jednom kroku ze stavu s právě tehdy, když pro nějaký prvek $s' \in \Gamma^{-1}(g)$ platí $s' \in s$. Všechny stavy, které lze dosáhnout z cíle g, tvoří tranzitivní uzávěr $\Gamma^{-1}{}_{\infty}(g) = \Gamma^{-1}(g) \cup \Gamma^{-1}(\Gamma^{-1}(g)) \cup ....$

Plány a řešení

Plán π je posloupnost akcí $\langle a_1, \ldots, a_k \rangle$ a je řešením problému P právě když $\gamma(s_0,\pi)$ splňuje g. Existenci plánu lze ověřit dvěma způsoby, a to dopředně a zpětně. U dopředného ověření existence plánu musí platit $S_g \cap \Gamma_{\infty}(s_0) \neq \emptyset$. V případě zpětného ověření musí existovat stav $s' \in \Gamma^{-1}{}_{\infty}(g)$, pro který platí $s' \in s_0$, aby měl plánovací problém řešení.

1.1.2 Klasické plánování formálně

Obecně se plánování zabývá volbou a organizací akcí, které mění stav systému [4]. Systém Σ modelující stavy a přechody je zobecněním plánovací domény tím, že kromě množiny stavů S a množiny akcí A, přidává množinu vnějších událostí E a modifikuje přechodovou funkci γ: $S \times A \times E \rightarrow \mathcal{P}(S)$.

Klasické plánování je typ plánování, které dodržuje tyto podmínky:

1. systém je konečný,
2. systém je plně pozorovatelný,
3. systém je deterministický – $\forall s \in S \; \forall u \in (A \cup E)$: $|\gamma(s,u)| \leq 1$,
4. systém je statický – $E = \emptyset$,
5. cíle jsou omezené – cílem je dosažení některého stavu z množiny cílových stavů,
6. plány jsou sekvenční,
7. čas je implicitní – akce i události jsou instantní,
8. plánuje se offline – stav systému se nemění v průběhu plánování.

Principem většiny plánovacích technik je prohledávání. Zde se omezím na prohledávání stavového prostoru, kde uzly odpovídají stavům, hrany akcím a plán cestě v prohledávaném prostoru. Stavový prostor může být prohledáván buď od počátečního stavu k cílovému, dopředné prohledávání, nebo od cíle směrem k počátečnímu stavu, zpětné prohledávání.

Dopředné plánování

Dopředné plánování je založené na (dopředném) prohledávání prostoru, kdy začínáme v počátečním stavu a pokračujeme k některému z cílových stavů. Obecně je tento algoritmus úplný (pokud existuje řešení, tak ho algoritmus najde) a korektní (nalezené řešení je skutečně

řešením) v případě, kdy se vybírá další akce nedeterministicky, viz popis algoritmu v příloze A.1. Nedeterminismu se lze zbavit, pokud jsou akce voleny systematicky pomocí technik jako prohledávání do šířky, do hloubky, apod. Nevhodná volba prohledávacího algoritmu (hladové prohledávání) může mít ale vliv na úplnost a korektnost celého plánování.

Problémem dopředného plánování je vysoký větvící faktor (počet možností k výběru). Vhodná heuristika nebo ořezávání může omezit počet prohledávaných irelevantních větví. Naproti tomu je nutné si pamatovat všechny navštívené stavy, což vede v nejhorším případě k exponenciální paměťové náročnosti.

Zpětné plánování

Idea zpětného plánování je začít plánovací proces cílem a pomocí relevantní akce definovat nový podcíl jako regresní množinu. V následujících iteracích se postupuje stejně až do doby, než je aktuální podcíl podmnožinou počátečního stavu. Algoritmus je úplný a korektní. V obecné podobě nedeterministicky vybírá z množiny relevantních akcí další akci pro vytvoření nového podcíle, jak je vidět z popisu algoritmu v příloze A.2. Zpětné plánování může být implementováno deterministicky, k tomu ale potřebujeme detekovat cykly (nekonečné větve).

Větvení může být menší než u dopředného plánování, stále je ale zbytečně velké. Nápomocnou je technika zvaná liftování, neboli částečné instanciování akcí, která zmenšuje velikost větvení. Detekce cyklů se stane ale ještě obtížnější kvůli volným proměnným.

1.1.3 STRIPS

STRIPS (akronym pro Stanford Research Institute Problem Solver) je plánovač představený roku 1971 Richardem E. Fikesem a Nilsem J. Nilssonem. STRIPS reprezentuje model světa pomocí formulí predikátové logiky (klasická reprezentace) a pracuje na základě zpětného plánování. Prohledávací prostor se snaží redukovat tak, že mezi relevantními akcemi pro aktuální cíl volí tu, jejíž efekty splňují cíl. Následně se volí předpoklady této akce jako nový podcíl. Tato redukce má za následek neúplnost algoritmu, protože nemusí existovat akce, která bude splňovat všechny předpoklady následující akce. Dalším problémem STRIPS plánovače je, že generuje redundantní plány (Sussmanova anomálie).

1.2 HTN plánování

HTN alias Hierarchical Task Network plánování bylo vytvořeno jako rozšíření plánování pracujícího na principu STRIPS. Výhoda HTN oproti STRIPS spočívá v tom, že plánovač může využívat dodatečných informací o hierarchické dekompozici plánovací domény.

Dekompozice probíhá rozkládáním úlohy na další podúlohy. U HTN plánování existují dva typy úloh, a to primitivní úlohy (proveditelné, na úrovni STRIPS operátorů) a neprimitivní, které musí být rozloženy na další podúlohy. Návod, jak rozložit neprimitivní úlohy, reprezentují metody. Plánovač může generovat plně uspořádané (akce jsou prováděny přesně v pořadí, v jakém je obsahuje plán) nebo částečně uspořádané plány (akce nemusí být prováděny v přesném pořadí, v jakém je obsahuje plán). Teoretický model HTN dokáže řešit více plánovacích domén než STRIPS [6]. Hlavní nevýhodou HTN plánování je, že nevhodně popsaná plánovací doména (operátory, metody) může mít velký dopad na rychlost a řešitelnost plánovacích problémů.

1.2.1 HTN plánování formálně

Formální systém HTN je postavěn na predikátové logice 1. řádu stejně jako klasická reprezentace. Jako jazyk je zde označena množina L, která obsahuje další množiny:

- V – nekonečná množina symbolů pro proměnné,
- C – konečná množina symbolů pro konstanty,
- P – konečná množina symbolů pro predikáty,
- F – konečná množina symbolů pro tzv. primitivní úlohy (primitive tasks),
- T – konečná množina symbolů pro tzv. složené úlohy (compound tasks),
- N – nekonečná množina pomocných symbolů pro úlohy.

Typy úloh

Primitivní úloha je tvořená konstrukcí $f(x_1, x_2, \ldots, x_n)$, kde $f \square F$ a $x_1, x_2, \ldots, x_n$ jsou termy (proměnné nebo konstanty). Primitivní úloha se už dále nerozkládá, je přímo vykonatelná.

Složená úloha je reprezentována jako konstrukce $t(x_1, x_2, \ldots, x_n)$, kde $t \square T$ a $x_1, x_2, \ldots, x_n$ jsou termy. Tento typ úlohy nelze provést přímo, musí se provést několik dalších kroků (splnění podúloh).

Cílová úloha odpovídá achieve[l]. l je literál tvaru $p(y_1, y_2, \ldots, y_n)$, kde $p \square P$ a $y_1, y_2, \ldots, y_n$ jsou termy. Tato úloha určuje, co je potřeba splnit (v klasickém plánování odpovídá části cílového stavu), navíc ji nelze provést přímo. Cílové a složené úlohy se společně označují jako neprimitivní úlohy.

Úkolová síť je syntaktická konstrukce tvaru $[(n_1: \alpha_1), \ldots, (n_m: \alpha_m), \phi]$, kde:

- α_i je úloha pro i = 1, ..., m,

- n_i □ N je úkolový symbol pro α_i,
- ϕ je boolean formule popisující vztahy mezi $n_1, \ldots, n_m$, např. (n_i, n_j) znamená, že n_i musí být dokončeno před n_j.

Tato síť odpovídá cílovému stavu v klasickém plánování. Primitivní úkolová síť ja taková, která obsahuje pouze primitivní úlohy (jinak neprimitivní).

Stav světa je množina instanciovaných platných atomů.

Plánovací doména

Operátor je dán tvarem operator[f(x_1, x_2, ..., x_n), precond (l_1, l_2, …, l_m), effects(k_1, k_2, …, k_m)], kde f □ F, l_1, l_2, …, l_m jsou literály, které musí stav splňovat, aby bylo možné provést primitivní úkol f, k_1, k_2, …, k_m jsou literály určující efekty úkoly f. Termy x_1, x_2, ..., x_n se mohou vyskytovat v rámci l_1, l_2, …, l_m a k_1, k_2, …, k_m.

Jako návod na neprimitivní úkol slouží metoda, která má tvar (α:d), kde α odpovídá neprimitivnímu úkolu a d popisuje úkolovou síť. α lze tedy splnit tak, že se provedou všechny části úkolové sítě d.

Nyní už můžeme definovat plánovací doménu D jako dvojici množin operátorů a metod □O,M□.

Použití operátoru a metody

Operátor se aplikuje jako plně instanciovaný (akce v klasickém plánování) vždy na stav s. Výsledkem této aplikace je nový stav daný jako (s - effects$^-$(k_1, k_2, …, k_m)) □ effects$^+$(k_1, k_2, …, k_m).

Úkolová síť se rozkládá pomocí metod. Metoda, která má být aplikovaná na úkolovou síť, nemusí být v základním tvaru, tzn. pouze s konstantami, ale může být liftovaná (některé atomy můžou být instanciované, zbytek zůstane jako proměnné). Metoda (α:d) je použitelná k úkolové síti $[(n_1: \alpha_1), \ldots, (n_m: \alpha_m), \phi]$, pokud je α unifikovatelná s některým z $\alpha_1, \ldots, \alpha_m$. Mějme θ jako nejobecnější unifikaci (most general unifier) α a α_i, výsledkem aplikace metody (α:d) je úkolová síť $[(n_1: \alpha_1), \ldots, (n_{i-1}: \alpha_{i-1}), d\theta, (n_{i+1}: \alpha_{i+1}), \ldots, (n_m: \alpha_m), \phi']$, kde ϕ' je upravená ϕ podle $d\theta$.

Plánovací problém

Plánovací problém u HTN je zadán trojicí □d, I, D□, kde d odpovídá úkolové síti, pro kterou vytváříme plán, I je počáteční stav světa a D je plánovací doména.

Plán, který řeší problém □d, I, D□, je posloupnost σ instanciovaných primitivních úkolů

splňující následující body:

- pokud je d primitivní, pak je σ uspořádání primitivních úkolů v d tak, že σ je proveditelná z počátečního stavu I a jsou splněny všechny podmínky ϕ v síti d pro σ,
- v případě, kdy je d neprimitivní, existuje-li posloupnost rozkladů úkolové sítě d podle domény D taková, že výsledkem je primitivní síť d' a problém □d', I, D□ má řešení σ.

Metody s předpoklady

Metody mohou mít předpoklady stejně jako operátory. Původní konstrukce se tak rozšíří na tvar (α:precond (l_1, l_2, …, l_m):d), kde α je neprimitivní úkol, d je úkolová síť a l_1, l_2, …, l_m jsou literály, které musí aktuální stav splňovat, aby mohla být metoda aplikována. Rozšíření metod o předpoklady redukuje prohledávaný prostor.

1.2.2 Dostupné implementace HTN plánovačů

Dále jsou zmíněny nejznámější implementace HTN plánovačů včetně krátké charakteristiky.

Nonlin

Jeden z prvních reprezentantů HTN plánování. Nonlin je schopný produkovat částečně uspořádané plány. Vznikl ještě v době, kdy nebyl systém HTN plánování formálně popsán. Nonlin je implementován v jazyce Lisp.

UMCP (Universal Method Composition Planner)

UMCP je jako první HTN plánovací systém popsán HTN formalismem a zároveň je prokazatelně korektní a úplný. Plánovač je opět implementován v jazyku Lisp.

O-Plan

Doménově nezávislý systém odvozený od Nonlinu, který v sobě zahrnuje kromě HTN plánovače také uživatelské rozhraní a kontrolní mechanismus prováděných akcí. Navíc do plánování zahrnuje časové omezující podmínky a zdroje. Systém je implementován v Lispu.

SIPE-2

Velmi pokročilý plánovací systém využívaný při organizování vojenských operací, řízení letecké dopravy, minimalizaci škod při ropné havárii, apod. Tento systém umožňuje přeplánování během exekuce aktuálního plánu. Navíc zahrnuje omezující podmínky a zdoje do procesu generování plánů. Jeho implementace je dostupná jazyce Lisp.

(J)SHOP 1,2

Rodina HTN plánovačů vyvinutých na Marylandské univerzitě. SHOP (Simple Hierarchical Ordered Planner), JSHOP 1 a JSHOP 2 jsou doménově nezávislé plánovací mechanismy založené na uspořádané dekompozici úloh. SHOP 2 (viz příloha A.3) generuje i částečně uspořádané plány. JSHOP 2 navíc využívá techniky kompilace plánovací domény a definice problému [14]. SHOP a SHOP 2 jsou implementovány v jazyku Lisp, zatímco JSHOP 1 a JSHOP 2 jsou napsány v jazyce Java.

1.3 Reaktivní (dynamické) plánování

Obecně se jedná o metody výběru akce (action selection) pro autonomní agenty. Princip této skupiny plánování spočívá v tom, že plánovač pracuje s aktuálním stavem světa a vrací (zpravidla) pouze jednu akci v každém okamžiku rozhodování.

1.3.1 POSH plánování

POSH (Parallel-rooted, Ordered Slip-stack Hierarchical) plánování se používá v systémech postavených na principu Behavior-Oriented Design (BOD) [1]. BOD je metodika navržená na konstruování komplexních agentů. Jejím základem je dekompozice agentovy inteligence v řadu vyjádřitelných chování (např. jdi, najez se,...). V duchu Object-Oriented Design je každé chování zakódováno jako objekt a primitivní elementy (proveditelné akce) tohoto chování jsou jeho metody.

Plány tohoto mechanismu mají hierarchickou strukturu, jsou reaktivní a na každé úrovni hierarchie jsou podúlohy uspořádány podle svých priorit. Volba další akce je založena na aktuálním stavu okolního světa, který dokáže agent vnímat skrz své senzory.

Základ POSH plánu tvoří pět komponent, dva primitivní elementy (primitives) a tři agregáty (aggregates). Mezi primitivní elementy se řadí činy (acts) a vjemy (senses). Rozdíl mezi těmito dvěma elementy je v tom, že vjemy mohou vracet nějakou hodnotu (která může být následně porovnána s jinou), zatímco výsledkem činů je pouze úspěch nebo neúspěch. Tyto elementy jsou ale pouze listy hierarchického uspořádání plánu. Zkombinování primitivních elementů do reaktivních plánů zajišťují právě agregáty, ke kterým patří řídící kolekce (drive collection), kompetence (competence) a posloupnost akcí (action pattern). Řídící kolekce je kořenem hierarchické struktury plánu a na její úrovni se rozhoduje, jaký bude hlavní cíl agenta. Kompetence jsou reaktivní plány, které umožňují agentovi rychle dokončit zvolenou podúlohu v závislosti na aktuálním stavu světa. Posloupnost akcí nahrazuje kompetence v situacích, kdy dopředu víme, jaké akce a v jakém pořadí je nutné provést.

POSH plán obsahuje navíc mechanismus pro zlepšení reakčního času eliminováním zásobníku, který vzniká při průchodu celé struktury plánu. Tento mechanismus se nazývá Slip-stack hierarchie a jeho princip spočívá v tom, že si aktuálně zpracovávaná kompetence uchovává informaci o elementu řídící kolekce, ze kterého byla vyvolána. Při dokončení části této kompetence začne plánovač prohledávání od uloženého řídícího elementu [1].

Scheduled POSH

V tomto návrhu POSH plánovače jsou v paměti uloženy všechny aktivní podúlohy (od řídícího elementu až k poslední kompetenci nebo posloupnosti akcí) a navíc jiné rozpoznávací procesy mohou upravovat priority elementů [5]. Tyto vlastnosti snižovaly rychlost plánovacího procesu. Implementace je dostupná v jazyku Python a Lisp.

SPOSH

Strict Slip-stack POSH je „čistější" verzí plánovače. Od předchozí se liší změnami v gramatice plánu a dále dodržuje politiku slip-stack popsanou v obecné časti o POSH plánování. Implementace je dostupná v jazycích Python a Jython.

Simple POSH

Simple POSH je zjednodušená implementace přizpůsobená tomuto projektu. Na první pohled má nejblíže verzi SPOSH, je ale ochuzená o několik funkčností:

- nedodržení slip-stack politiky, při každém prohledávání plánu se začíná řídící kolekcí,
- plánovač neřídí agenta, ale agent používá plánovač pouze jako doplňující rychlý reaktivní systém,
- plánovač je napsán v jazyce Java a s agentem komunikuje prostřednictvím rozhraní, metody u agenta, které (přesně) odpovídají názvům primitivních elementů, se nemusí registrovat nebo definovat.

2 Popis problému

Tato práce prezentuje návrh a implementaci hybridního plánovače, s jehož pomocí je řízeno chování skupiny agentů ve virtuálním světě akčních her. Chování je podmíněno taktickou spoluprácí agentů inspirovanou skutečnými vojenskými nebo policejními jednotkami určenými pro boj v malých a uzavřených prostorech.

V této kapitole je podrobně popsána celá problematika návrhu plánovacího systému. Začneme prostředím, ve kterém agenti jednají. Část je věnována nárokům kladených na plánovací systém v závislosti na očekávaném chování, dále je zde popsána funkce hybridního plánovače, včetně rolí jeho jednotlivých vrstev. A v neposlední řadě popis jednotky, která bude mít na starosti „velení" skupině agentů.

2.1 Motivace

Jak jsem nastínil v úvodní kapitole, podstatou této práce je konstrukce hybridního dvouúrovňového plánovače, který bude schopný ovládat skupinu agentů v akčních hrách. Hry tohoto žánru se vyznačují velmi dynamickým prostředím, což znamená, že herní oponenti často mění své pozice, v okolí lze přemisťovat předměty, apod. Hráč musí v takovém prostředí často přehodnocovat své aktuální plány, aby se zvládl adaptovat rychle se měnícímu stavu hry. Pokud tedy nastane nějaká událost, která podstatně ovlivní stav hry, jak ji bude hráč (člověk) řešit? Zřejmě jako první věc zkusí zvolit novou herní strategii a potom už v rámci této strategie bude plánovat další kroky. Z této myšlenky vychází návrh hybridního plánovače, u kterého bude mít první vrstva na starosti volbu vhodné strategie pro celou skupinu agentů, zatímco druhá bude ovládat samotné agenty ve virtuálním světě.

2.2 Prostředí

Celý systém je navržen s ohledem na budoucí integraci do počítačové hry UT2004. Tato hra je příkladným reprezentantem her akčního žánru, konkrétně spadá do žánru FPS (First Person Shooter), kde hráč vnímá své okolí skrz oči řízeného agenta. Hra se navíc vyznačuje dynamickým prostředím, co se týká jednání ostatních agentů. Fyzikální model prostředí zůstal téměř opomenutý, to ale neubírá na potřebách pro tento projekt. UT2004 nabízí několik herních scénářů, ze kterých se jako nejvhodnější jeví CTF. Jedná se o týmový mód, jehož principem je získat vlajku z oponentovy základny a dostat ji do své, aniž by byla moje ukradena. Tým se tedy může soustředit na jeden cílový bod narozdíl od scénářů jako Domination, OnSlaught, apod.

Pro UT2004 byla napsána modifikace GameBots [9], která zprostředkovává připojení do UT2004 a ovládání herních agentů prostřednictvím síťového protokolu TCP/IP. Navíc je vyvíjeno mnoho ovládacích prostředků umožňujících snadné vytvoření herního agenta s rozhraním pro definování agentova jednání, více v [8].

Prvotní simulační prostředí bude implementováno v jazyce Java. Tento simulátor se bude od UT2004 lišit v těchto bodech:

- 2D reprezentace světa

  2D svět je očividně jednodušším prostředím (než 3D v UT2004), které postačí na testovací účely, což bude zřejmé až z následující kapitoly 2.3. Svět je reprezentován formou čtvercové sítě a pohled na něj je orientovaný kolmo shora.

- diskrétní čas

  Akce nejsou prováděny spojitě (v reálném čase), ale diskrétně, tzn. po krocích. Toto omezení umožní lepší sledování veškerých řídících a plánovacích procesů během simulace. Na druhou stranu je to jedna z hlavních odlišností oproti funkčnosti UT2004 a s tím související problémy bude problematické odhalit na simulaci.

- není postaven na architektuře client – server

  V UT2004 se agent (client) musí připojit na server, aby se mohl zapojit do hry. V simulátoru existuje jedna komponenta zastupující herní mód, a ta má na starosti správu provádění akcí (ve smyslu kdy je možné provést další krok) všech přítomných agentů, navíc všechny přítomné informuje o herních událostech.

- obdélníkové místnosti bez dynamických překážek

  Prostor, ve kterém agenti operují, je omezený na místnosti obdélníkových tvarů bez dynamických překážek. V UT2004 sice není omezení na tvar místností, nicméně obytné prostory mají zpravidla obdélníkový tvar. Zároveň v prostředí UT2004 nejsou přítomny dynamické překážky.

- jednopatrové místnosti

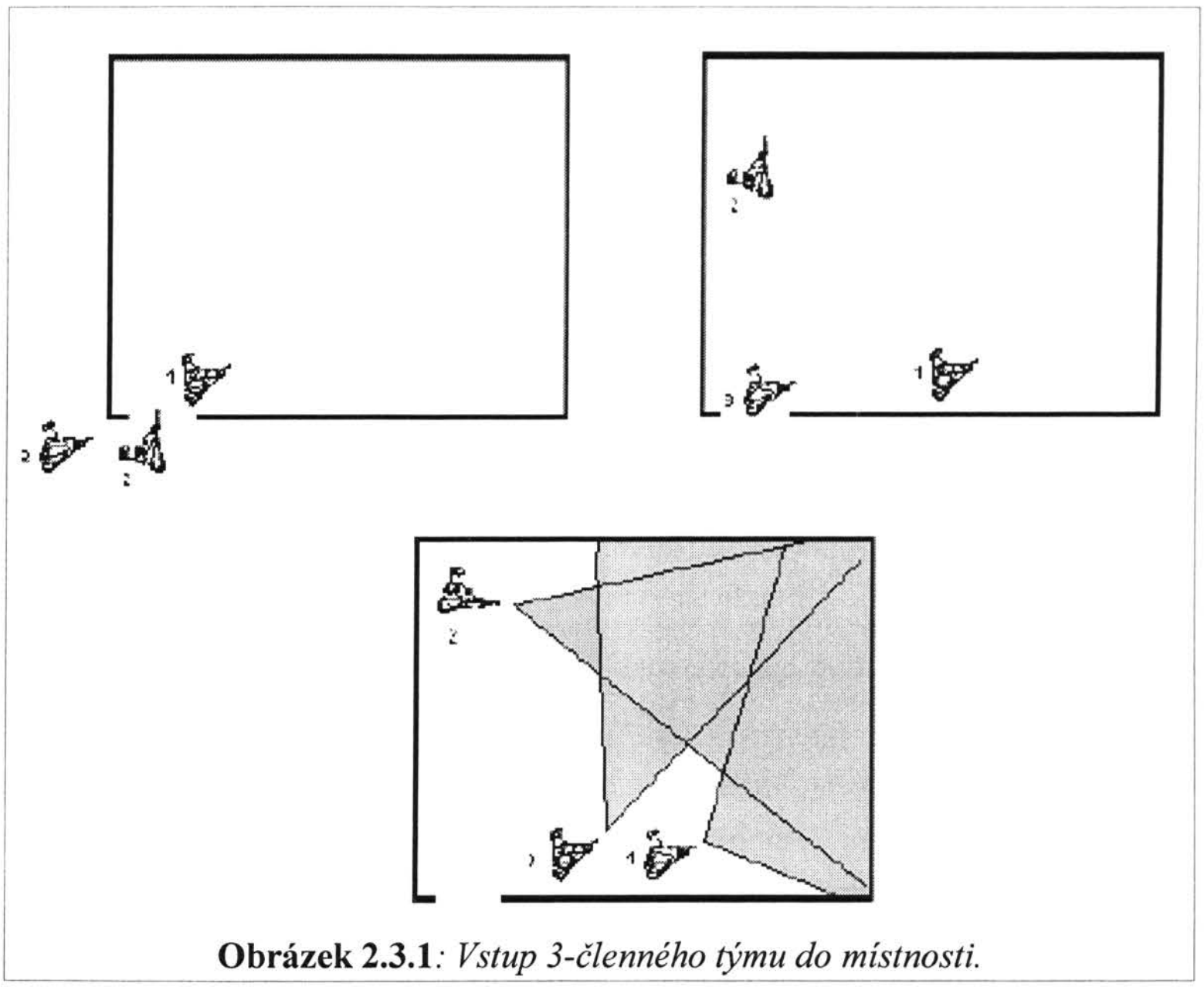
  Pro zjednodušení prostředí obsahuje dvou a více patrové prostory propojené skrz schody, výtahy, apod.

Simulace ale reflektuje problematické části, které by byly hůře odhalitelné a odladitelné (prohledávání prostoru v závislosti na pozičních vlastnostech, správnost a provádění HTN plánů, atd.), pokud by se plánovací systém implementoval přímo do UT2004.

2.3 Co se bude plánovat

Obecně jsou řídící plány přizpůsobeny strategii pro scénář CTF. Větší důraz je ale kladen na plánování postupu jednotky při plnění cílů z řídících plánů. Chování jednotky by mělo vykazovat známky týmové spolupráce na vyšší úrovni, inspirované operacemi speciálních jednotek v situacích souhrnně označovaných jako CQB [10].

Soupis taktik pro CQB nabízí široké spektrum metod, jak postupovat uzavřenými prostory s využitím strategických míst tak, aby se týmu povedlo co nejefektivněji a nejrychleji eliminovat hrozby, i když se většinou nachází ve znevýhodněné pozici. Tým agentů se snaží procházet místnostmi takovým způsobem, aby byla co největší část prostoru pokryta po většinu času. Obrázek 2.3.1 zobrazuje postup týmu do místnosti a vizuální kontrolu většiny prostoru.



Obrázek 2.3.1: *Vstup 3-členného týmu do místnosti.*

Celý tým bude řízený centrální jednotkou. Centrální řízení by bylo možné nahradit distribuovaným systémem, to by ale přinášelo spoustu dalších problémů (vyjednávání, připisování rolí agentům, …) spojených s multiagentními systémy.

Tým bude složen ze 4 agentů. Tento počet je dostatečný na simulaci většiny manévrů pro vstup a „vyčištění“ místnosti. Pro menší počet by taktické plánování nemělo až takový význam, na druhou stranu pro větší jednotku by byly plány komplikovanější, ale ve výsledku by se rozdíl projevil až u pohybu týmu skrz velké místnosti.

Jak bylo zmíněno výše, taktické plánování je spojené s využíváním strategických prvků prostředí. K těmto prvkům se řadí poziční vlastnosti jako roh místnosti, roh dveří, pozice u stěny, apod. Simulační prostředí by tedy mělo poskytovat dostatečně popsané poziční vlastnosti pro svůj svět.

2.4 Návrh hybridního plánovače

Řídící plánovací systém bude složen ze dvou plánovacích mechanismů uspořádaných do dvou úrovní. Horní vrstvu bude tvořit reaktivní plánovač POSH, základem spodní vrstvy bude HTN plánovač. Rozdělení plánovacího mechanismu do vrstev znamená, že výstupní akce plánovače ve vyšší vrstvě budou tvořit cíl pro plánování na nižší vrstvě.

Úlohou POSH vrstvy je výběr vhodného chování pro celou skupinu agentů podle aktuálního scénáře hry. Díky reaktivní povaze plánovače by měla být zajištěna včasná reakce na změnu stavu hry. Plánované akce nebudou proveditelné agenty jako jednotlivci (nebudou se odkazovat na konkrétního agenta), ale budou udávat směr, jakým se má celá skupina ubírat, aby byla schopná konkurovat oponujícímu týmu (skupinu si můžeme představit jako vyššího agenta). Uveďme si příklad ze scénáře CTF v situaci, kdy jsou vlajky v základnách příslušných týmů:

my_flag_in_my_base & enemy_flag_in_enemy_base => go-to-enemy-base

Výsledná akce go-to-enemy-base („jdi do nepřátelské základny“) bude dále rozplánována v nižší vrstvě, protože je společná pro všechny agenty ze skupiny.

Přeplánování nebude pod správou POSH plánovače, ale bude vyvoláváno v zavislosti na potřebě vyšší řídící jednotky, která plánovač využívá.

HTN plánovač bude plnit úlohu „taktického navigátora“ ve virtuálním světě. Narozdíl od POSH vrstvy budou plány přímo exekuovány skupinou agentů. V metodách budou zakódované manévry používané v CQB, jak je popsáno v kapitole 2.3. Jelikož simulační prostředí je 2D a navíc se neuvažují prostory složené z více pater, omezím se pouze na plánování vstupů a průchod místnostmi. Plány budou obsahovat převážně akce typu „jdi na pozici“ a „zaměř se na oblast“, což ve výsledku pro koordinovanou navigaci stačí.

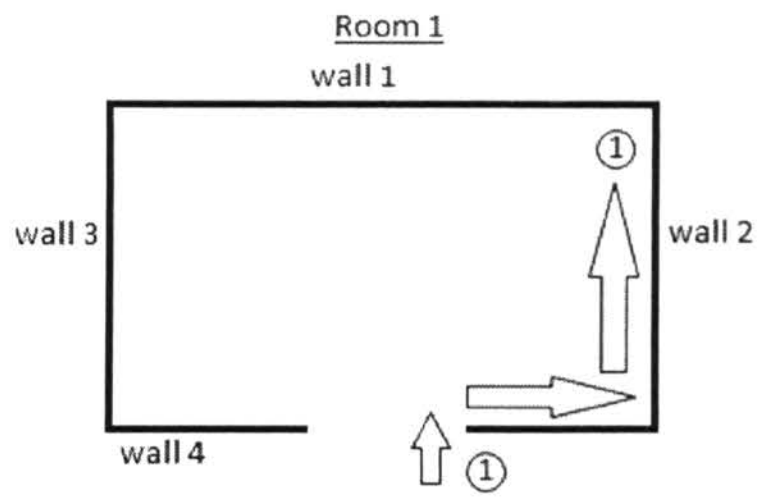
V akcích typu „jdi na pozici“ bude cílové místo určeno několika pozičními vlastnostmi, nikoliv konkrétními souřadnicemi. Konkrétní místo musí vyhovět daným požadavkům tím, že

bude obsahovat co možná nejvíce ze zadaných pozičních vlastností. Dále se nabízí otázka, zda záleží, jakým způsobem se agent do cíle dostane. A proto doplňkem těchto akcí budou ještě požadavky na cestu, prostřednictvím které by měl agent dosáhnout cílové pozice. Ty budou reprezentovány také formou pozičních vlastností. Tento doplněk je důležitý kvůli tomu, aby agent mohl využívat nejen strategických výhod cílové pozice, ale i cesty. V příkladu 2.4.1 je znázorněna cesta agenta podle zadaných vlastností.

Příklad 2.4.1

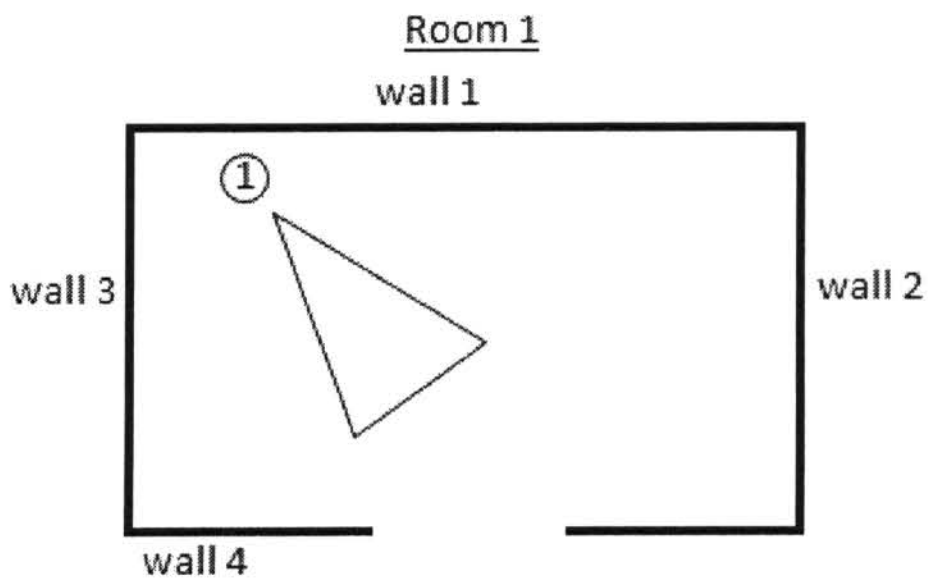
akce „jdi na pozici“:

- kdo: Agent 1
- cíl: roh místnosti Room 1 mezi stěnami wall 2 a wall 1
- cesta: podél stěny wall 4 a podél stěny wall 2



Obrázek 2.4.1: *Cesta agenta podle zadaných pozičních vlastností.*

Smyslem akcí „zaměř se na oblast“ je vizuální kontrola specifikované oblasti. Agent by měl sledovat každý objekt z této oblasti v pravidelných časových intervalech. Objekty, které bude možné zaměřit, budou zapsané stejným způsobem jako poziční vlastnosti. Na obrázku 2.4.2 je znázorněné, jak agent kontroluje vstupní dveře.



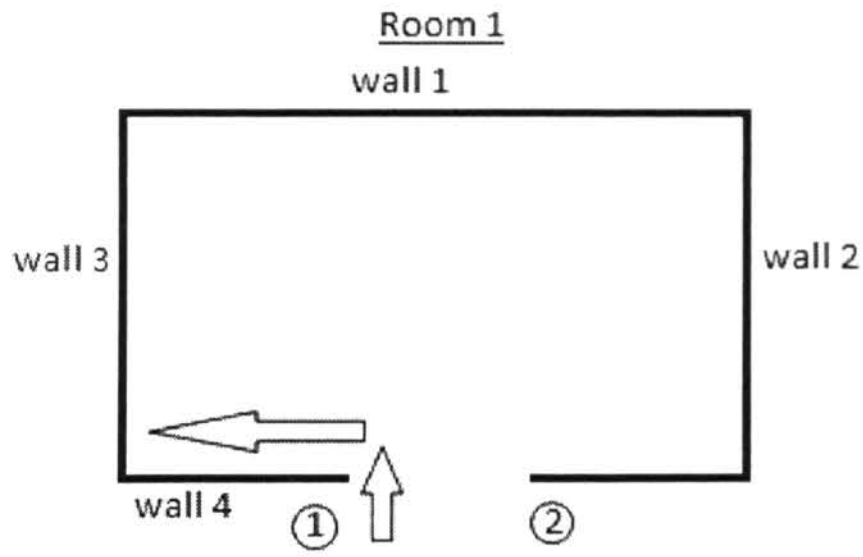
Obrázek 2.4.2: *Agent kontroluje vstupní dveře do místnosti.*

Taktika pro CQB je náročná na časovou koordinaci akcí pro členy týmu, protože přesun jednoho agenta na další pozici může záviset na dokončení akce druhého agenta. Tím pádem musí být v plánu obsažena informace, kdy může agent začít provádět danou akci. Příklad 2.4.2 představuje situaci, kdy může začít agent vykonávat svou akci, jakmile jiný dosáhne své cílové pozice:

Příklad 2.4.2:

akce „jdi na pozici“:

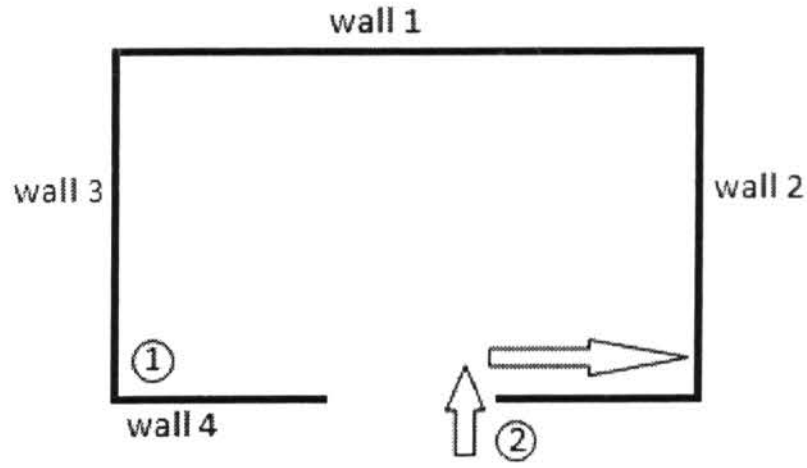
- kdo: Agent 1
- cíl: roh místnosti Room 1 mezi stěnami wall 4 a wall 3
- cesta: podél stěny wall 4



Obrázek 2.4.3_a: *Agent 1 začíná svou akci.*

akce „jdi na pozici“:

- kdo: Agent 2
- cíl: roh místnosti Room 1 mezi stěnami wall 4 a wall 2
- cesta:podél stěny wall 4
- kdy: jakmile Agent 1 dokončí svou poslední akci



Obrázek 2.4.3_b: *Agent 1 ukončil akci a agent 2 může začít svou.*

2.5 Hlavní řídící jednotka

V kapitole 2.3 bylo nastíněno, že čtyřčlennou skupinu agentů bude řídit centrálně nějaká vyšší jednotka, která bude mít svůj předobraz v jednání velitele jednotky v reálném světě. Tento post bude zastávat hlavní řídící jednotka a její cíle lze shrnout do těchto bodů:

- komunikace s herním prostředím
  - příjem a odesílání herních událostí v závislosti na scénáři hry
- plánování
  - správa hlavního plánovacího mechanismu
  - vytvoření mezivrstvy mezi POSH a HTN plánovači, která bude modifikovat a přeposílat akce z POSH vrstvy do HTN vrstvy
- koordinace agentů
  - extrakce akcí pro konkrétní agenty z plánu pro celý tým
- komunikace s agenty
  - zasílání informací o událostech ve hře
  - zasílání informací o stavu prováděných akcí ostatních agentů
  - příjem informací od agentů týkajících se jejich statusu

3 Analýza a řešení problému

Během procesu návrhu a implementace simulátoru se objevily problematické části. Jedná se zejména o návrh algoritmu, který je schopný hledat cestu v prostoru. V tomto případě je cesta zadaná formou pozičních vlastností prostředí. Další nejistou pasáž tvoří výběr vhodných implementací plánovacích systémů. Vzhledem k tomu, že úkolem agentů při taktických manévrech je vizuálně pokrýt určitý prostor, je nutné, aby byl v simulátoru zakomponovaný mechanismus schopný zjistit viditelnost dvou bodů.

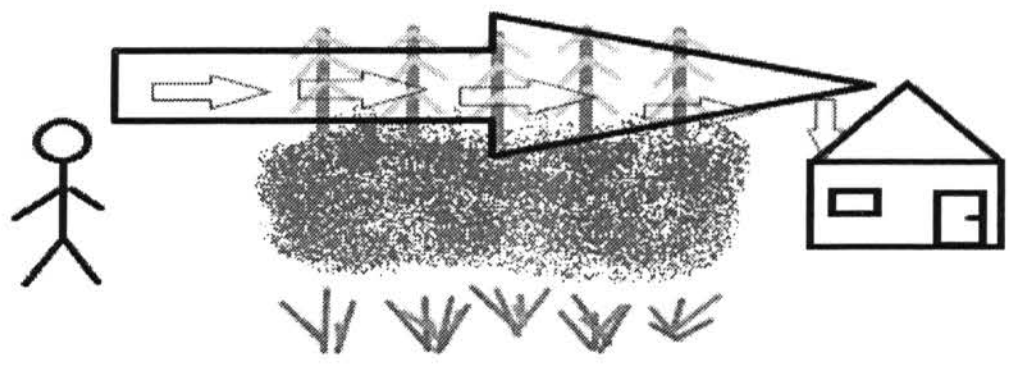
3.1 Vyhledávání cesty zadané pozičními vlastnostmi

3.1.1 Inspirace

Předlohou pro hledání cesty je lidský způsob myšlení při plánováni trasy v členitém prostoru. Člověk si zpravidla nejdříve vytyčí obecnější trasu, která uspokojuje nějaké potřeby, např. za jak dlouho se chce dostat do cíle, jaké cestovní prostředky jsou dostupné, kolem kterých míst by chtěl jít, apod. Jakmile si zvolí trasu tímto způsobem, už se nezajímá o jiné odbočky. Tak šetří i čas, protože nemusí na každé odbočce přemýšlet, jestli by nebylo lepší volit jinou cestu. Celý postup je naznačen v příkladu 3.1.

Příklad 3.1:

Mějme zloděje a dům. Zloděje dělí od domu velké prostranství tvořené zleva lesy, uprostřed loukou a vpravo křovím (obrázek 3.3.1.5). Zloděj se chce do domu dostat pokud možno nepozorovaně a tiše. Křoví mu nabízí poměrně slušný prostor, aby nebyl zahlédnut, nicméně při prodírání skrze malé větve asi pozornosti neunikne. Dále je nasnadě dojít k domu přes louku, zloděj by nebyl slyšet, ale zároveň by bylo težké ho nepřehlédnout. Další možností je les, který nabízí dostatečně dobrý prostor, aby se zloděj dokázal dostat k domu nepozorovaně a zároveň mu nehrozí prozrazení kvůli šramotu.



Obrázek 3.3.1.5: *Volba cesty pro zloděje.*

3.1.2 Popis algoritmu

Algoritmus pracuje v prostředí, ve kterém jsou místnosti rozděleny na menší podprostory, tzv. sektory. Rozloha celé místnosti je pokud možno rovnoměrně distribuována do těchto sektorů (většina sektorů má čtvercový tvar). Zároveň jsou do sektorů extrahovány nejsilnější poziční vlastnosti z polí spadajících do jejich výseče.

Celé vyhledávání se skládá ze dvou fází. První je zaměřena na nalezení cesty složené ze sektorů. Ve druhé fázi už je prohledáván prostor nejnižší úrovně, který je omezen pouze na pole obsažené ve výsečích sektorů z předchozí fáze. Sektory mají svůj význam jako jistý druh heuristiky, který nám pomáhá redukovat prohledávaný prostor.

Prohledávání prostoru v obou fázích funguje na principu algoritmu A^* [12] vždy s volbou nejdražšího prvku co se hodnocení týče.

Pseudokód algoritmu prohledávání na první úrovni:

NAJDI-CESTU-V-SEKTORECH(startSektor, cestaSCilem)

1 cílové sektory ← vyber sektory obsahující poziční vlastnosti, kterými je charakterizována cílová pozice

2 přidej startSektor do množiny sektorů určených k prohledání

3 **while** množina sektorů určených k prohledání není prázdná **do**

4 aktuální sektor ← vyber první sektor z množiny sektorů určených k prohledání

5 přidej aktuální sektor do množiny sektorů, které jsou prohledané

6 **if** počet navštívených cílových sektorů je roven počtu cílových sektorů **and** cílové sektory jsou v množine sektorů, které jsou prohledané **then**

7 cesta nalezena!

8 **for each** soused aktuálního sektoru **do**

9 **if** soused je v množině sektorů, které jsou prohledané **then**

10 zkonroluj dalšího souseda

11 **if** soused je mezi cílovými sektory **then**

12 spočítej cenu souseda jako cílového sektoru v závislosti na vlastnostech, kterými je charakterizována cílová pozice

13 přidej souseda do množiny navštívených cílových sektorů pokud v této množině ještě není nebo je ale s nižší cenou

```
14              else
15                  spočítej cenu souseda v závislosti na pozičních vlastnostech,
    kterými je charakterizována cesta k cíli
16                  přidej souseda do množiny sektorů, které jsou prohledané
```

Rozhodující roli hrají metody pro výpočet ceny přechodu na další sektor, protože právě v nich se berou v úvahu vlastnosti prostředí.

Algoritmy z první a druhé fáze jsou si velmi podobné. Jediný rozdíl spočívá v tom, že u druhé fáze už nemusíme prohledávat cíl jako víceprvkovou množinu, ale máme daný jediný cílový bod.

Na závěr popisu je dobré si uvědomit, že tento algoritmus není optimální co se týče nalezené cesty. Optimálnost se ztrácí v přechodu z první fáze do druhé, protože sektory sice obsahují nejsilnější vlastnosti polí ze své výseče, ale neobsahjí informaci o přechodu mezi sektory (mohou zde být překážky). Je to svým způsobem daň za redukci prohledávaného prostoru.

3.2 HTN plánování

3.2.1 Volba vhodného HTN plánovače

HTN plánovač hraje roli organizátora taktických manévrů v dynamickém prostředí, čili důležitým měřítkem je rychlost řešení zadaného problému. Z čehož vyplívá, že se můžeme omezit na plánovače JSHOP 1 a 2, protože ostatní jsou implementovány v jazyku LISP. Zakomponování LISPovského plánovače do systému by obnášelo vytvoření „mostu“ mezi dvěma aplikacemi (vzhledem k tomu, že simulátor je napsán v jazyku Java) a zpomalení výpočtu kvůli transformacím mezi reprezentacemi datových typů. JSHOP 1 se ukázal jako nejvhodnější kandidát i přesto, že JSHOP 2 dosáhl velmi dobrých výsledků v plánovacích soutěžích. JSHOP 2 nebyl vybrán, i když plánuje rychleji, ale samotnému procesu plánování předchází ještě proces kompilační, kdy se kompiluje plánovací doména i plánovací problém. A to s ohledem na budoucí integraci plánovacího systému do UT 2004 je časově velmi náročné (v rámci několika málo sekund).

JSHOP 1 je volně dostupný formou balíku tříd, což nám umožňuje vložit jeho instanci přímo do kódu simulátoru.

3.3.2 Reprezentace u JSHOP 1

Formální jazyk u plánovače JSHOP 1 (JSHOP) je postaven na predikátové logice 1. řádu, jak bylo zavedeno v kapitole 1.2.1, nicméně zápis je mírně odlišný [13].

Termy, logické atomy a literály

Term je proměnná, konstanta nebo výraz tvaru (f t_1 t_2 … t_n), kde f je funkční symbol a t_i je term □i.

Call-term je výraz tvaru (call f t_1 t_2 … t_n), kde f je funkční symbol a t_i je term nebo call-term □i. Pro JSHOP má call-term speciální význam. Indikuje, že k f přísluší nějaká procedura a kdykoliv JSHOP potřebuje vyhodnotit např. předpoklady, které obsahují call-term, nahradí tento call-term výsledkem aplikace funkce f na argumenty t_1 t_2 … t_n.

Logický atom je výraz tvaru (p t_1 t_2 … t_n) nebo (call p u_1 u_2 … u_n), kde p je predikátový symbol, u_i je term pro □i a t_i je term pro □i, který není call-term a ani žádný neobsahuje. V druhém případě je k p přiřazená funkce, jak bylo popsáno u call-termu.

Literál je buď logický atom a nebo výraz tvaru (not a), kde a je logický atom.

Konjunkce, axiomy a stav světa

Logická konjunkce je zde použitá ve dvou formách. První je běžná, daná konjunkcí všech literálů v seznamu (l_1 l_2 … l_n). Druhá forma se označuje jako tagovaná a má tvar (:first l_1 l_2 … l_n). Smysl této konjunkce je oznámit systému, který dokazuje tuto konjunkci, že stačí její první nalezený důkaz.

Axiom je výraz formulovaný jako (:- a C_1 C_2 … C_n), kde a je logický atom a C_i je konjunkce pro □i. Význam axiomu je, že a je pravdivé, pokud je C_1 pravdivá, pokud není, pak jestli je C_2 pravdivá, atd.

Stav světa je množina instanciovaných atomů.

Úkoly, operátory a metody

Úloha má tvar (s t_1 t_2 … t_n), kde s je symbol pro úlohu a t_i je term nebo call-term pro □i. S využítím pojmů zavedených v kapitole 1.2.1, s je primitivní úloha, pokud s □ F, a neprimitivní, pokud s □ T.

Operátor je zadaný tvarem (:operator h P D A), h je primitivní úloha označována také jako hlavička operátoru, ve které se nemohou vyskytovat call-termy, P odpovídá předpokladům operátoru a je složené logických atomů, D je list logických atomů (obsahuje proměnné pouze z h), které budou odebrány ze stavu světa po aplikaci operátoru, A je list logických atomů

(obsahuje proměnné pouze z h), které budou naopak do stavu světa přidány použitím operátoru.

Metoda je seznam tvaru (:method h C_1 T_1 C_2 T_2 … C_k T_k), kde h je úloha, která neobsahuje žádný call-term, C_i jsou konjunkce (běžné i tagované) pro □i představující předpoklady metody, T_i jsou úkolové sítě pro □i, kde úlohy mohou obsahovat i call-termy. Úkolová síť je zde seřazený seznam úloh, tzn. úlohy musí být plněny v pořadí, v jakém jsou zadány.

Plánovací doména, plánovací problém a plán

Plánovací doména je definovaná trojicí složené z množiny axiomů, operátorů a metod. Plánovací problém je trojice tvořená stavem světa, úkolovou sítí a reprezentací domény.

Plán je reprezentován tvarem (h_1 h_2 … h_n), h_i představuje instanciovanou hlavičku operátoru □i.

3.2.3 Využití JSHOPu 1

3.2.3.1 Uspořádanost plánů

Ačkoliv vrací JSHOP uspořádané plány (tzn. pořadí, v jakém by měly být akce prováděny), ne vždy jsou akce prováděny postupně. Začátek exekuce některých akcí může být závislý na dokončení jiných (při taktickém manévrování), a tedy může dojít k porušení pořadí, ve kterém byly akce uspořádány v plánu. JSHOP neumí pracovat s časem a nemůže plánovat během exekuce plánu. Problém je vyřešen zavedením časových známek do plánovacích operátorů formou běžného argumentu. Tyto známky podávají informaci o závislosti prováděných akcí. Časové známky se dělí na:

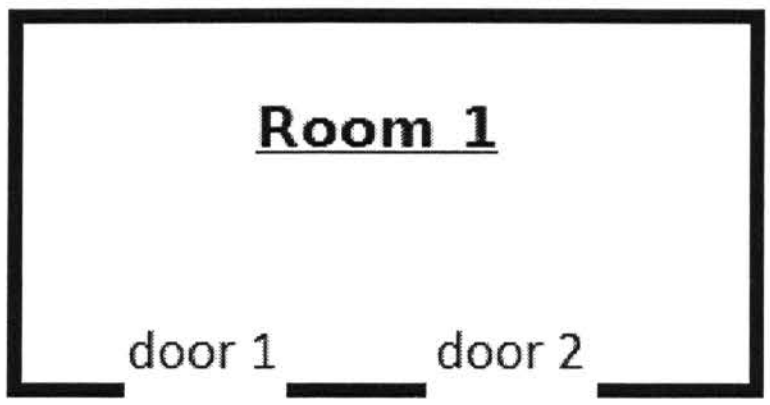
- absolutní – exaktní časové určení, kdy se má akce začít provádět (v časových jednotkách simulátoru),
- relativní – vyjádření závislosti na jiné akci, dané možnostmi:
  - ihned po dokončení specifikované akce,
  - ihned po dokončení akce, která je společná pro celou skupinu agentů (jakmile každý agent ze skupiny splní svou část), tzv. týmová akce.

Celý problém bude ale pravděpodobně zřetelnější po přečtení kapitoly věnující se implementaci.

3.2.3.2 Optimalizace

Jelikož JSHOP plánuje taktické akce závislé ve velké míře na popisu prostředí, musí mu být struktura místností dostatečně popsána prostřednictvím metod a stavu světa.

Stav světa obsahuje pouze popis místností, vstupních prostor a rolí agentů. Plánuje se přechod skupiny agentů z jedné místnosti do druhé, proto je do stavu světa přidána informace o těchto dvou místnostech rozšířená navíc o popis vstupních prostor, které náleží do místností. Jedinými pozičními vlastnostmi popsanými ve stavu světa jsou polohy dveří v místnostech, typ stěny (postranní/centrální stěna), ve které jsou dveře situovány, a poloha vstupních prostor vzhledem k ostatním, které se nachází ve stejné stěně (viz obrázek 3.2.3.2).



Obrázek 3.2.3.2: *Informace o poloze dveří ve stavu světa.*

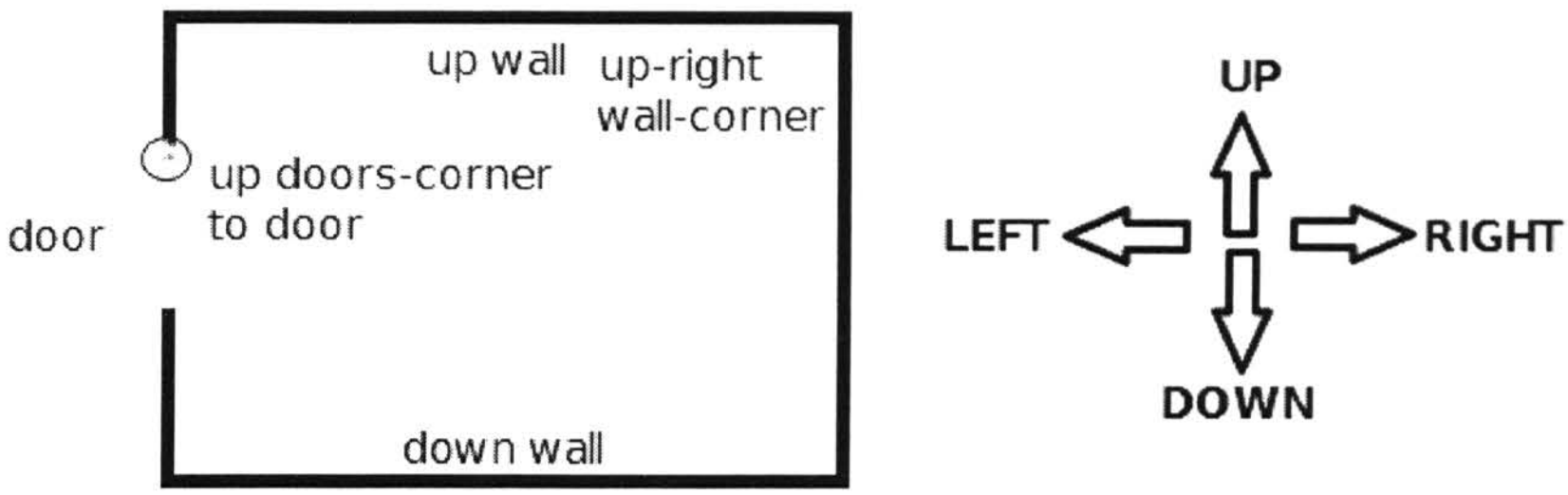
Zbytek pozičních vlastností je zapsán do metod, které pracují na obecném modelu místnosti. Díky zobecnění náhledu na místnosti se urychlí prohledávání plánovací domény, protože ušetříme čas u předpokladů metod (které by jinak u vlastností testovaly kategorii, pozici, ...) a stav světa není zahlcen detailním popisem místností.

Obecný model místnosti umožňuje označit jakoukoliv poziční vlastnost, jen je potřeba doplnit systém určování pozic. Zde je tento systém navržen tak, aby nebyl závislý na systému, který je používán v simulačním prostředí. Oddělení pozičních značení má za následky:

- plánovač přistupuje ke každé místnosti ze stejného výchozího směru a v závislosti na něm se pak odkazuje na jakoukoliv pozici (určenou vlastnostmi) v místnosti,
- akce výsledného plánu se odkazují na poziční vlastnosti pomocí pozičního značení plánovače,
- při sestavování stavu světa vkládá hlavní řídící jednotka informaci o konverzi ze svého pozičního systému do systému používaného v plánovací doméně,
- hlavní řídící jednotka přijímá informaci ve formě speciální akce o tom, jak má

doménový poziční systém převést na svůj (výchozí směr domény a jeho převod na směr zadaný řídící jednotkou ve stavu světa).

Na obrázku 3.2.3.3 jsou znázorněny některé vlastnosti prostředí, jejichž pozice je určena pomocí pozičního sys tému plánovače.



Obrázek 3.2.3.3: *Příklady pozičních vlasností z pohledu plánovače.*

3.3 POSH plánování

3.3.1 Volba vhodného POSH plánovače

U implementací POSH a SPOSH je agent řízen plánovačem. Navíc tento plánovač sám stanoví, kolik času má agent na dokončení akce nebo kdy dostane přednost úloha s vyšší prioritou. Hlavní řídící jednotka, která zastřešuje veškeré plánování pro skupinu agentů, ale není řízena žádným plánovačem, protože sama určuje, kdy a na jaké úrovni plánovat a kdy je pro ni vhodné přeplánovat. Proto byl zvolen Simple POSH, což je zjednodušená verze od SPOSH. Neobsahuje Slip-Stack hierarchii, a to v případě simulátoru až tak nevadí. Smyslem Slip-Stack politiky je zlepšit reakční čas, nicméně plány nejsou velmi rozsáhlé, takže systém je schopný plánovat rychle. Neměla by nastat ani situace, kdy bude skupina agentů nehnutě stát, protože jim bude plánovač měnit stále úkol, aniž by měli agenti šanci jej dokončit. Řídící jednotka tomu předchází, protože po naplánování chování pomocí Simple POSH dochází ještě k plánování na nižší úrovni, a tím pádem k dalšímu plánování na POSH vrstvě dojde až s časovým odstupem.

3.3.2 Reprezentace plánu pro Simple POSH

Gramatický popis struktury plánů je převzat z [5] a vypadá následovně:

```
          name :: [a token]

     competence :: (C <name> <other-elements>)

drive-collection :: (RDS <name> <drive-elements>)
                      RDS root decision sequence
  other-elements :: (ELEMENTS <comp-element>+)

  drive-elements :: (DRIVES <drive-element>+)

    comp-element :: (<name> (TRIGGER <ap>) <name>)

   drive-element :: (<name> (TRIGGER <ap>) <name>)

              ap :: <act|sense>*

  action-pattern :: (AP <name> (<ap>))
```

Element `name` zde může zastupovat jméno jakékoliv POSH komponenty, tedy vjemu (`sense`), činu (`act`), řídící kolekce (`drive-collection`, `drive-element`), kompetence (`competence`) a posloupnosti akcí (`action-pattern`). Priority jednotlivých elementů u kompetence (`competence`) a řídící kolekce (`drive-elements`) jsou definovány posloupností jejich zápisu.

3.4 Viditelnost

Viditelnost je nedílnou součástí akcí založených na CQB. Implementace algoritmu, který umí určit viditelnost v prostředí, není ale až tak jednoduchá záležitost. Jelikož simulační program nemusí odpovídat přesně realitě, byl zde sestrojen algoritmus založený na pixelové rasterizaci přímek v počítači [15]. Tento algoritmus funguje ve dvou krocích. Nejprve se pomocí rasterizace zjistí jednotlivé souřadnice přímky mezi dvěma body (odpovídá cestě mezi těmito body) a potom se každá souřadnice kontroluje na mapě prostředí, jestli neobsahuje nějaký druh překážky.

4 Implementace

Ve druhé kapitole bylo popsáno několik důležitých oblastí, které budou tvořit orientační body pro zdárnou implementaci celého projektu. Každá z těchto oblastí v sobě přináší jisté problematické časti, které jsou zřejmé už při návrhu „na papír“ nebo se projeví až při samotné

implementaci. Tato kapitola se tedy zaměří hlavně na implementaci stěžejních tříd a algoritmů, díky kterým agenti vykazují známky týmové spolupráce na úrovni taktických manévrů při průchodu obytnými prostory.

4.1 Koncept simulačního programu

Simulační program je navržen tak, aby se do jisté míry podobal hře UT2004. Nejpodstatnější rozdíl spočívá v chápání času. Simulátor pracuje v diskrétním čase, což umožňuje sledovat veškeré probíhající akce daleko detailněji než jak je tomu v UT2004.

Základní strukturu návrhu celého programu tvoří tři třídy:

- `Simulator` - třída, která zprostředkovává grafické rozhraní pro znázornění a ovládání simulačního procesu,
- `Game` - abstraktní třída, slouží jako předchůdce pro herní scénáře,
- `WorldMap` - třída reprezentující herní prostředí.

Další třídy už vznikají v závislosti na zvoleném scénáři hry. Konkrétně na obrázku 4.1 jsou znázorněné třídy, které byly vytvořeny pro scénář CTF. Logika scénáře je zakomponovaná do třídy `CTFGame`, která navíc komunikuje o stavu hry s přítomnými hlavními řídícími jednotkami. Tyto řídící jednotky mají společného předchůdce `CTFCommander` sdružujícího specifickou funkčnost pro CTFGame. Potomci podědění z `CTFCommander` se mohou věnovat jen logice svého chování. V tomto případě jsou zde dva zástupci, a to `HTNCommander`, využívající převážně HTN plánování, a `POSHCommander`, pracující pouze na mechanismu POSH. Každá z těchto řídících jednotek obsahuje skupinu objektů reprezentujících herní agenty. Třídy těchto agentů se liší podle mechanismu, na jehož principu pracuje jejich řídící jednotka.

4.2 Platforma

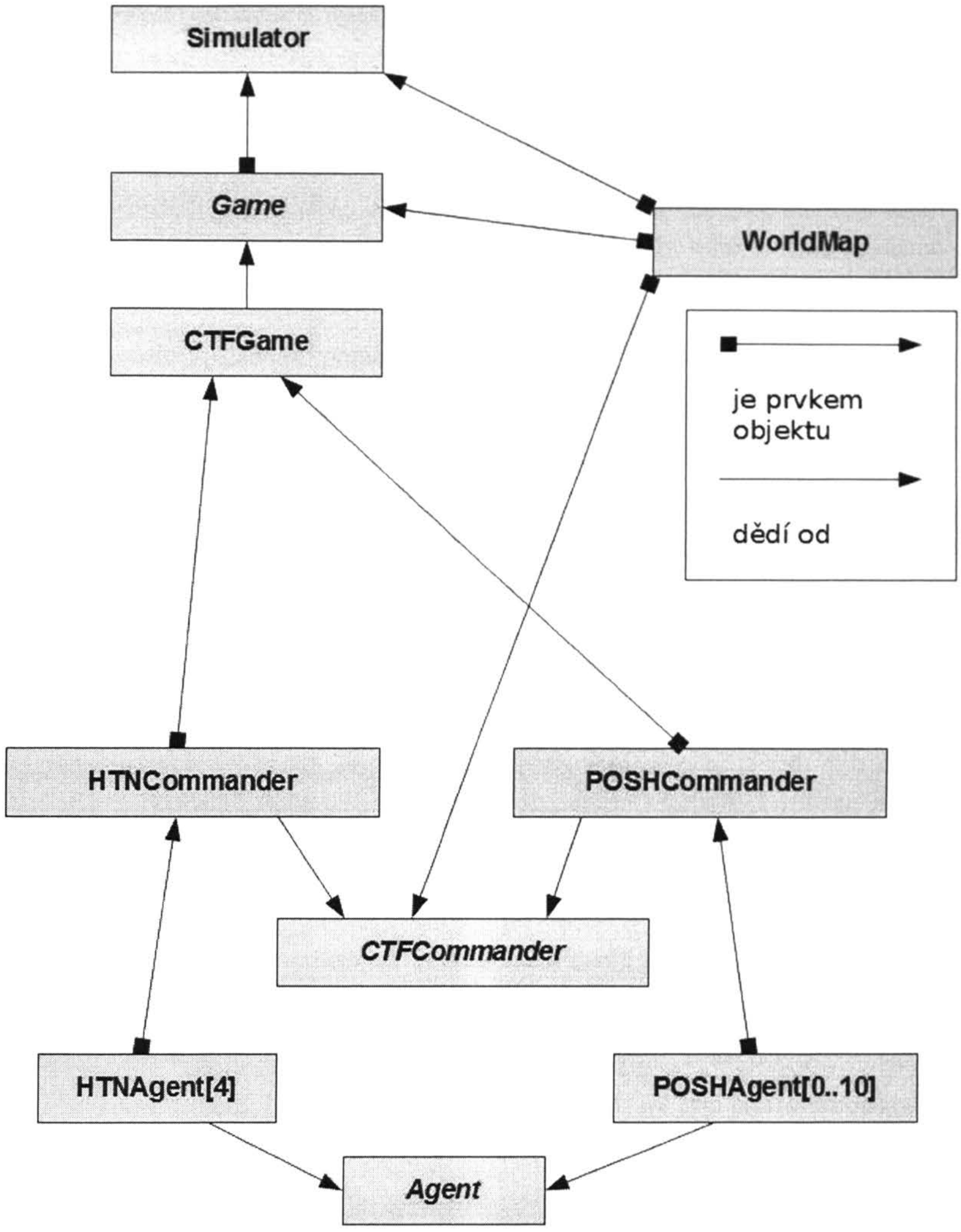
Celý projekt je napsán v programovacím jazyce Java pomocí JDK[1] verze 1.6.0_13 a vývojového prostředí Eclipse SDK[2] ve verzi 3.4.2. Kromě nativních balíků tříd z Java Core API, jsou použité nástroje pro logování (log4j[3]), mapování dat uložených v XML do Java objektů (JiBX[4]) a rozpoznávání gramatických konstrukcí (ANTLR[5]). Všechny tyto prostředky jsou volně dostupné pod veřejnými licencemi.

---

1 Java Development Kit, dostupnost na http://java.sun.com/javase/downloads/index.jsp
2 Software Development Kit, více na http://www.eclipse.org/downloads/
3 http://logging.apache.org/log4j/1.2/index.html
4 Binding XML to Java Code, http://jibx.sourceforge.net/
5 Another Tool for Language Recognition, více na http://www.antlr.org/

4.3 Od prostředí k taktické navigaci

Základními kameny taktického navigačního systému tvoří detailně popsané prostředí, zde zastoupeno třídou `WorldMap`, potom HTN plánování, zprostředkované zapojením externího plánovače JSHOP 1, a nakonec vyhledávání cesty určené pomocí pozičních vlastností v prostředí. Tyto prostředky můžeme brát jako pomůcky, které navigaci usnadňují. Důležitou součástí je i prezentace výsledků těchto komponent, což umožňují mechanismy v řídící jednotce `HTNCommander` (velitel jednotky) a třídě `HTNAgent` (herní agent). Doplněním je ještě SimplePOSH plánovač, ve kterém je zapsána strategie pro CTF. `HTNCommander` funguje jako prostředník mezi POSH (SimplePOSH) a HTN plánováním (JSHOP 1), který překládá akce vrácené POSHem do úkolů pro HTN.



Obrázek 4.1: *Diagram návrhu závislosti tříd pro scénář CTF.*

4.3.1 Prostředí

4.3.1.1 Entity prostředí

Celé herní prostředí je postaveno na entitách odpovídajících několika úrovním abstrakce při pohledu na svět. Tyto entity můžeme rozdělit do kategorií podle toho, k čemu se vážou:

- místnosti – struktury popisující samotné místnosti až k nejmenším stavebním prvkům,
- vstupní prostory – „dveře" propojující dvě sousední místnosti,
- poziční vlastnosti – příznaky o speciálních objektech v blízkosti dané pozice,
- distanční vlastnosti – popis oblasti vzdálené od konkrétní pozice v určitém směru.

Poziční a distanční vlastnosti budou popsány až po zadefinování tříd, které formují svět. Jakmile budeme mít strukturu místností a dveří, bude jasnější, které vlastnosti budeme schopni vyčíst.

Každá místnost je tvořena třemi úrovněmi, jak je vidět na obrázku 4.2. Nejnižší vstvu vytváří nejmenší části ve smyslu základní jednotky pohybu, které reprezentuje třída `MapField`. Třída `MapField` je charakterizována těmito atributy:

- `worldMapPosition` – reprezentace pozice ve světě, jelikož je tento svět dvourozměrný, tak jsou souřadnice určeny dvojicí [x,y],
- `type` – indikátor typu pole, zda-li se jedná o průchozí (`MapField.WALKABLE`) nebo překážku (`MapField.OBSTACLE`),
- `neighbours` – seznam sousedních políček (`MapField`), na které se lze bezprostředně dostat (krok délky 1),
- poziční vlastnosti:
  - `adjacentObjects` – množina pozičních vlastností, které jsou buď'to situovány přímo na objektu nebo jsou v bezprostřední blízkosti,
  - `spreadObjects` – množina pozičních vlastností, které nejsou v bezprostřední blízkosti,
  - `segments` – seznam objektů typu `MapUnitSegment`, který určuje příslušnost do segmentového rozdělení místnosti,
  - `epObjects` – objekty typu `EntryPoint`, které v tomto případě značí přilehlé vstupní prostory.

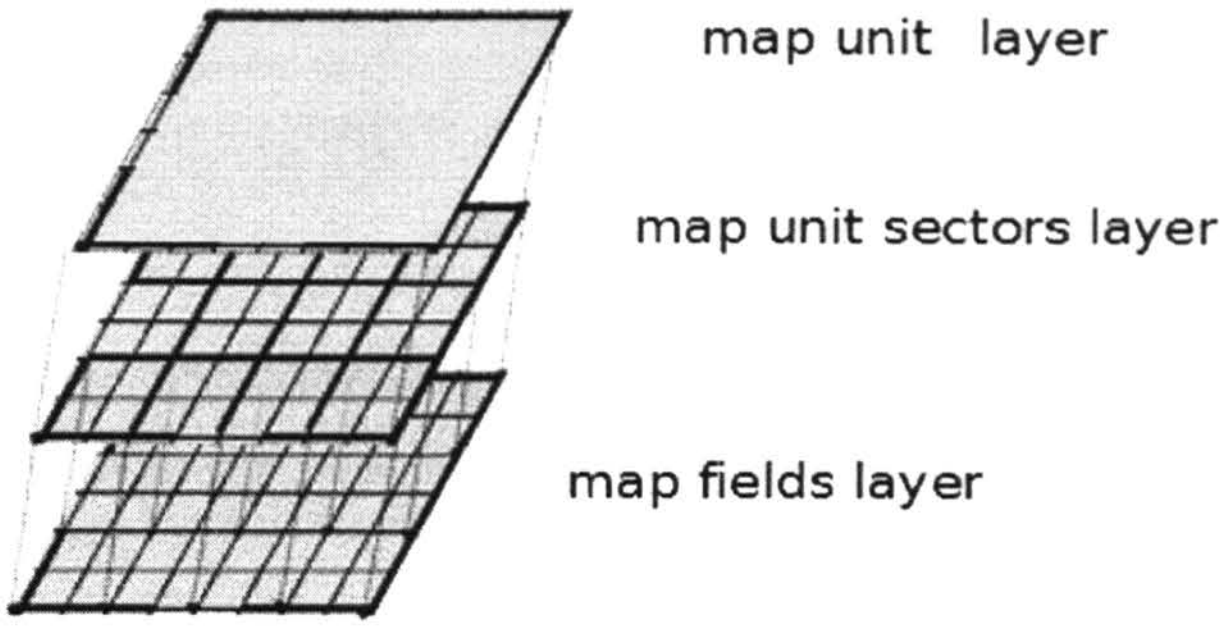
Střední vrstvu představují sektory zastoupeny třídou `MapUnitSector`. Jejich smysl bude jasnější až po zavedení algoritmu pro vyhledávání cesty. Zatím si definujeme nejdůležitější atributy třídy `MapUntiSector`:

- `parentUnit` – místnost typu `MapUnit`, pod kterou daný sektor spadá,
- `positionInParentUnit` – pozice v místnosti určená souřadnicemi `x` a `y`, protože místnost je rozložena do dvourozměrného pole sektorů,
- `neighbours` – seznam sousedních sektorů, které nemusí nutně ležet ve stejné místnosti,
- `members` – dvourozměrná výseč objektů typu `MapField` z mapy celého světa, které spadají do tohoto sektoru,
- poziční vlastnosti:
  - `properties` – poziční vlastnosti vyextrahované z objektů typu `MapField` přítomných v atributu `members`, které spadají do kategorie bezprostředních pozičních vlastností,
  - `spreadProperties` – poziční vlastnosti vyextrahované z objektů typu `MapField` přítomných v atributu `members`, které nespadají do kategorie bezprostředních pozičních vlastností,
  - `segments` - seznam objektů typu `MapUnitSegment` určujících příslušnost do segmentového rozdělení místnosti v závislosti na příslušnosti všech prvků v atributu `members`.

Nejvyšší vrstvu představuje třída `MapUnit`, která odpovídá místnosti jako celku a sdružuje do sebe nižší vrstvy. Objekt je popsán:

- `id` – jednoznačný identifikátor místnosti,
- `type` – výčtový typ `MapUnitType`, který řadí místnost do určité skupiny podle rozlohy,
  - `MapUnitType = {SMALL_ROOM, MEDIUM_ROOM, LARGE_ROOM, SHORT_HALL, MEDIUM_HALL, LONG_HALL}`
- `fromX, toX, fromY, toY` – oblast na mapě světa, přes kterou se místnost rozprostírá,
- `entryPoints` – seznam všech vstupních prostor typu `EntryPoint` vedoucích do této místnosti,
- `sectors` – dvourozměrné pole s prvky typu `MapUnitSector`, které reprezentuje rozložení prostoru místnosti (místnost jako matice sektorů),
- `neighbours` – seznam sousedních místností,

- properties – všechny poziční vlastnosti vytažené ze sektorů z atributu sectors.



Obrázek 4.2: *3-vrstvá hierarchická struktura místnosti.*

Možnost propojení dvou místností, tedy dvou objektů typu MapUnit, dává třída EntryPoint:

- id – jednoznačný identifikátor objektu,
- eptype – typ vstupních dveří podle jejich šířky daný výčtovým typem EntryPointType
  - EntryPointType = {SMALL_DOOR, MEDIUM_DOOR, LARGE_DOOR}
- mapUnits - dvě místnosti, které jsou těmito dveřmi spojené,
- epWalls – identifikáry pozičních objektů typu stěna, kde jsou dveře umístěny, vztahující se k místnostem, které dveře propojují,
- position – indikátor vertikální/horizontální polohy dveří,
- fromX, toX, fromY, toY, x, y – určení polohy dveří ve světě pomocí trojice souřadnic, a to fromX, toX, y pokud jsou dveře horizontální, nebo fromY, toY, x v případě vertikálních dveří.

Kromě exaktního určování pozic, pomocí souřadnic, je možné zadávat obecnější polohy. K tomu slouží výčtový typ DirectionFlag, který nabízí množinu směrů reprezentovaných pomocí světových stran (jak je známe ze skutečného světa), tedy:

DirectionFlag = {NORTH, NORTHWEST, WEST, SOUTHWEST, SOUTH, SOUTHEAST, EAST, NORTHEAST, CENTER, NONE}.

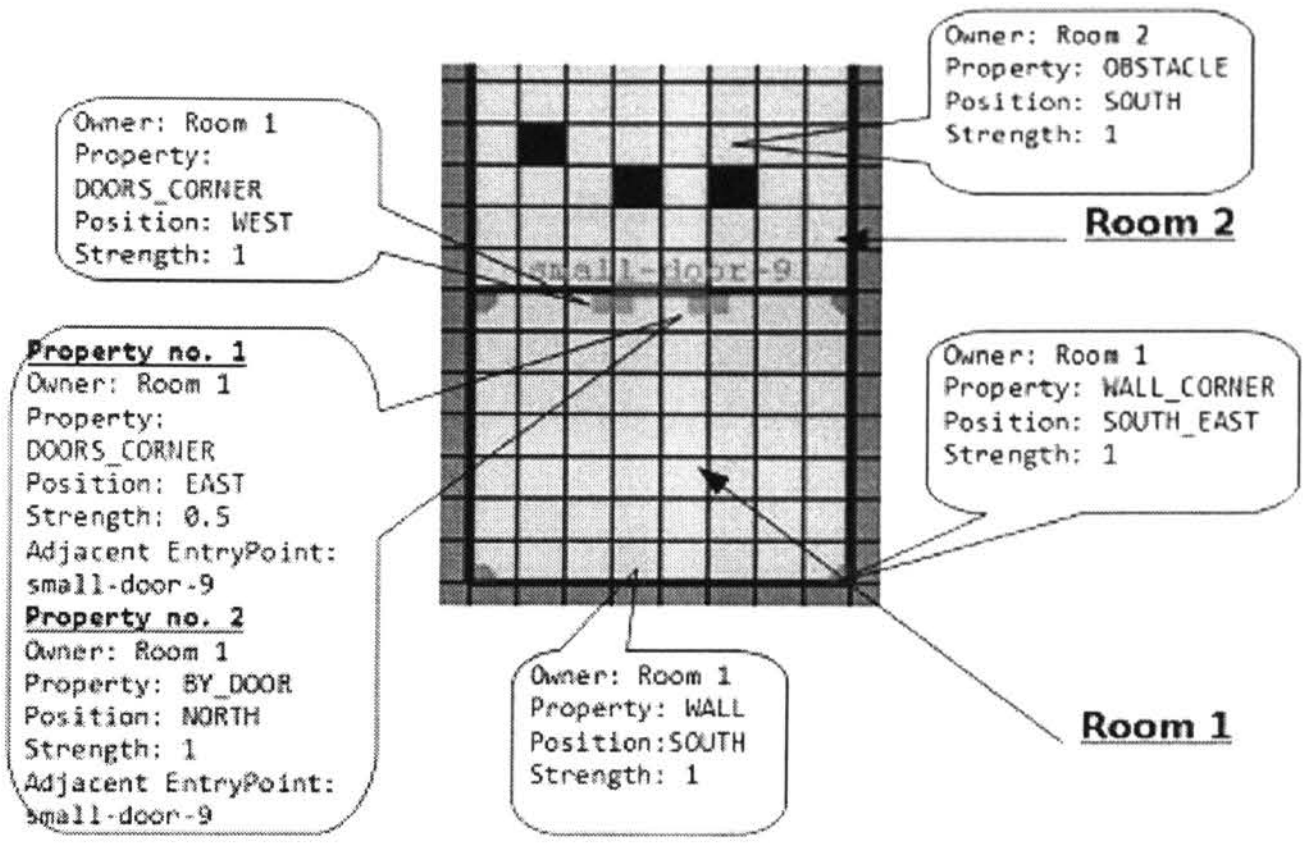
Nyní už zavedeme poziční vlastnosti, které vystihují specifické vztahy v prostředí. Svět máme složený z místností a vstupních prostor, jaké vlastnosti můžem tedy popsat? V rámci místností nás může zajímat, zda se zrovna nacházíme na pozici u stěny nebo v rohu dvou stěn, dále v kontextu se vstupními prostory je dobré vědět o pozici u rohu dveří, apod. Kompletní výčet přítomných pozičních vlastností závislých na objektech přítomných ve vytvořeném světě je definován výčtovým typem `PositionPropertyType`, který obsahuje následující:

- `WALL` – vlastnost indikující přítomnost stěny místnosti,
- `WALL_CORNER` – políčko s touto vlastností se nachází na průseku dvou stěn místnosti,
- `OBSTACLE` – vlastnost signalizuje, že v bezprostřední blízkosti políčka je neprůchodná překážka,
- `BY_DOOR` – políčko s touto vlastností se nachází v bezprostředním prostoru dveří,
- `DOORS_CORNER` – označuje pozici u rohu vstupního prostoru,
- `FLAG_POINT` – označení pro políčko, které slouží jako výchozí pozice pro umístění vlajky,
- `BY_AGENT` – dynamická poziční vlastnost, kterou obsahuje políčko v bezprostřední blízkosti nějakého agenta.

Samotné vlastnosti ale moc nevypoví, protože z nich nevyplívá, kde se přesně nachází. Proto se používá abstraktní třída `PositionPropertyObject`, která doplňuje poziční vlastnost o lokalizaci v prostoru. Jejími nejdůležitějšími atributy jsou:

- `owner` – specifikace objektu typu `MapUnit`, ve kterém se vlastnost nachází,
- `property` – konkrétní poziční vlastnost daná typem z `PositionPropertyType`,
- `position` – určení polohy z konkrétního políčka k dané vlastnosti pomocí třídy `DirectionFlag`,
- `strength` – vyjadřuje sílu (dosažitelnost, působnost) poziční vlastnosti.

Pro každou vlastnost z `PositionPropertyType` je vytvořena specifická třída, která dědí od třídy `PositionPropertyObject`. Každá z těcht tříd obsahuje další informace specifické pro svůj typ, které se využijí např. při vyhledávání cesty. Na obrázku 4.3 jsou znázorněné některé poziční vlastnosti.



Obrázek 4.3: *Příklady pozičních vlastností.*

Místnosti disponují, kromě výše zmíněných pozičních vlastností, ještě obecnějším rozdělením pomocí segmentů. Místnost je tak rozdělena na pět vertikálních a horizontálních částí, které usnadní navigaci skrz prostor celé místnosti, aniž bychom byli vázáni pouze na objekty tvořené stěnami a vstupními prostory. Jednotlivé typy segmentů jsou definované výčtovým typem `MapUnitSegmentation`:

- horizontální segmenty:
  - `{NORTH_WALL_SEGMENT, NORTH_WING_SEGMENT, ENTER_HORIZONTAL_SEGMENT, SOUTH_WING_SEGMENT, SOUTH_WALL_SEGMENT}`
- vertikální segmenty:
  - `{WEST_WALL_SEGMENT, WEST_WING_SEGMENT, CENTER_VERTICAL_SEGMENT, EAST_WING_SEGMENT, EAST_WALL_SEGMENT}`

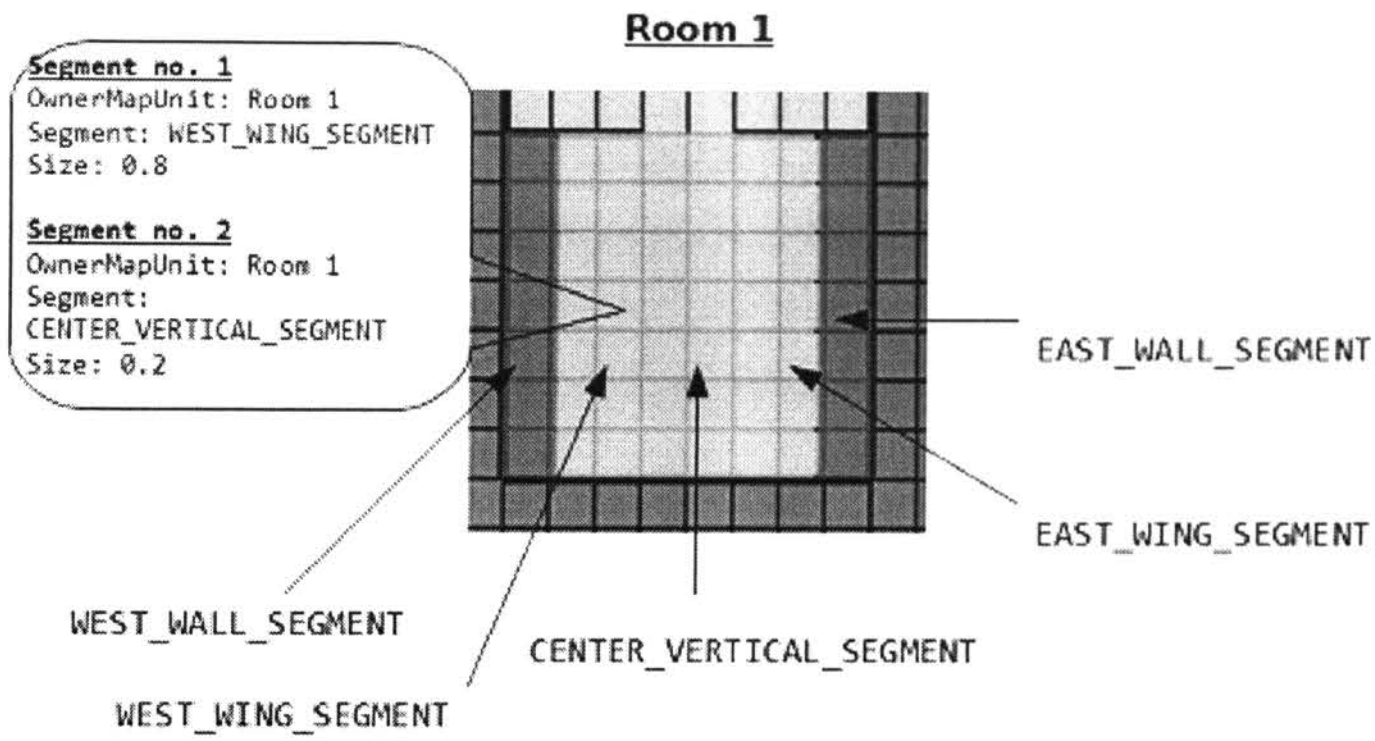
Navíc tato třída doplňuje typ o rozsah v místnosti určený percentuálním intervalem, např. `WEST_WALL_SEGMENT` se rozprostírá ve vertikálním rozmezí 0% - 15% z prostoru místnosti.

Podobně jak tomu bylo u pozičních vlastností, samotný typ je doplněn obalující třídou se specifickými informacemi o konkrétní poloze, což plní třída `MapUnitSegment`:

- `segment` – konkrétní typ segmentu reprezentovaný třídou `MapUnitSegmentation`,
- `size` – rozsah segmentu na konkrétním místě v místnosti (políčko, sektor), odpovídá síle u pozičních vlastností,

- `ownerMapUnit` – místnost (`MapUnit`), pod kterou segment spadá.

Na obrázku 4.4 je zobrazeno rozdělení místnosti na vertikální segmenty.



Obrázek 4.4: *Vertikální rozdělení místnosti.*

Distanční vlastnosti jsou definované v třídě `DistanceToObject` pomocí atributů:

- `target` – poziční vlastnost jako obecná pozice, ke které se distanční vlastnost vztahuje,
- `minDistanceFromTarget` – minimální vzdálenost od poziční vlastnosti `target`,
- `directionFromTarget` – směr daný typem `DirectionFlag`, který určuje, v jakém směru by se oblast od `target` měla nacházet,
- `targetPosition` – konkrétní pozice v mapě daná souřadnicemi, která se definuje podle výběru nejvhodnějšího políčka s poziční vlastností `target`.

4.3.1.2 Třída WorldMap

Mapa prostředí (světa) je reprezentována třídou `WorldMap`. Tato třída může přistupovat jak k nejnižší vstvě mapy, tak k vyšším celkům jako jsou místnosti a dveře. Navíc sdružuje všechny poziční vlastnosti, které jsou přítomné v mapě. Třída je popsána těmito atributy:

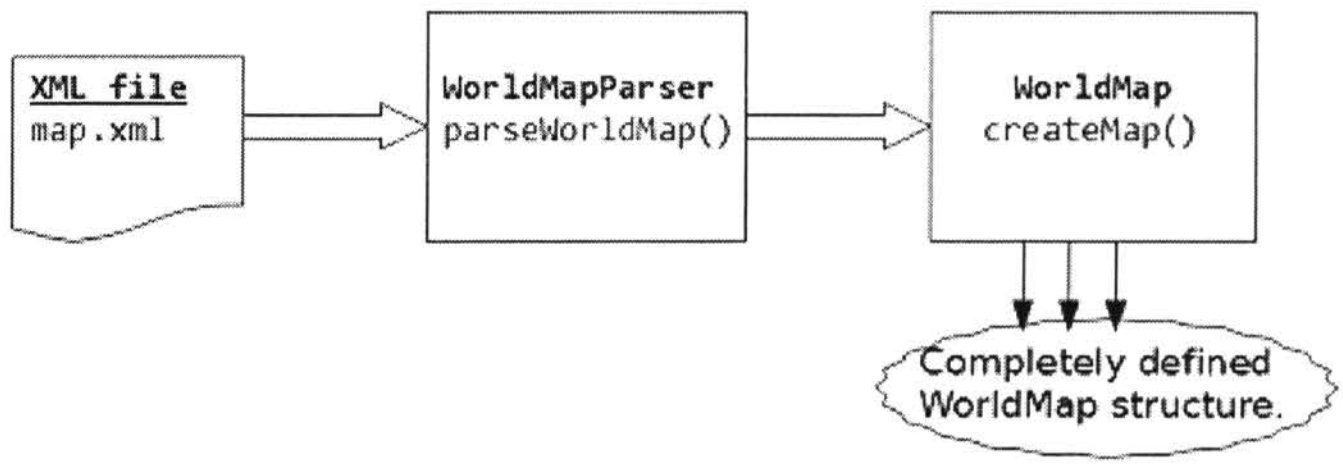
- `basicMap` – dvoudimenzionální matice složená z objektů třídy `MapField` reprezentující svět na nejnižší úrovni, jednotlivé políčka matice jsou inicializovány v závislosti na poloze místností,
- `properties` – slouží jako seznam všech pozičních vlastností v celém prostředí,

- `map` – seznam objektů třídy `MapUnit` reprezentující místnosti, které jsou přítomny v daném světě,
- `entryPoints` – objekty třídy `EntryPoint` představující všechny vstupní prostory v mapě.

Třída `WorldMap` pouze nezastřešuje přístup k mapě prostředí, její další úloha spočívá v automatickém vygenerování kompletní struktury světa. Základní popis mapy je uložen ve formátu XML, příklad je uveden v příloze C.1. Z tohoto souboru je prostřednictvím třídy `WorldMapParser` vytvořená prvotní struktura prostředí, která zahrnuje:

- `objekty třídy MapUnit popsané pouze pomocí souřadnic polohy,`
- objekty třídy `EntryPoint` propojující výše vytvořené místnosti,
- poziční vlastnosti typu `OBSTACLE`, `WALL`, `WALL_CORNER`, a `FLAG_POINT`, které jsou přímo zapsané v XML souboru nebo je lze vyčíst pomocí zadaných souřadnic místností.

V dalším kroku je tento koncept předložen třídě `WorldMap`, která dokončí mapu vygenerováním dalších informací charakterizujících svět do požadovaných detailů. Celý cyklus je znázorněn na obrázku 4.5.



Obrázek 4.5: *Schéma vytvoření kompletní instance třídy* `WolrdMap`.

Generování finální struktury je provedeno v metodě `createMap()`, jejíž kroky jsou:

1. vytvoření matice `basicMap`
   - do matice jsou inicialzovány objekty třídy `MapField` v závislosti na poloze a rozloze jednotlivých místností
   - instanci třídy `MapField` je zároveň přiřazena příslušnost do segmentace místnosti
2. vložení základní sady pozičních vlastností k příslušným políčkům struktury `basicMap`
   - při parsovacím procesu v metodě `parseWorldMap(mapSource)` na třídě

`WorldMapParser` byly inicializovány poziční vlastnosti typů `WALL`, `WALL_CORNER` a `FLAG_POINT`

3. vložení vstupních prostor (dveří) a s nimi souvisejících pozičních vlastnosti do `basicMap`
   - nastavení dveří do místností, které spojuje
   - podle umístění dveří se označí políčka v jejich beprostřední blízkosti vlastnostmi `BY_DOOR` a `DOORS_CORNER`, pokud se políčko nachází na rozmezí stěny a prostoru dveří (jelikož je políčko reprezentováno jako čtverec, musí být jeden z jeho vrcholů spojnicí mezi stěnou a začátkem dveří)
4. označení pozic okolo překážek, neboli doplnění poziční vlastnosti `OBSTACLE`
   - pro každý objekt z `basicMap`, jehož atribut `basicMap[x][y].type == MapField.OBSTACLE`, se označí všechny sousední pole vlastností `OBSTACLE`
5. doplnění informací pro některé poziční vlastnosti a vstupní prostory
   - pro vlastnost `WALL_CORNER` se stanoví stěny, ke kterým roh místnosti náleží (jako průsečík)
   - u objektů typu `EntryPoint` se doplní identifikátory stěn z místností, které jsou spojené skrz konkrétní dveře
6. extrakce pozičních vlastností z `basicMap` do místností uložených v `map`
   - ze všech polí spadajících do polohy místnosti se vybírají pouze poziční vlastnosti v bezprostřední blízkosti, tedy z `MapField.adjacentObjects`, navíc ty s největší sílou (`PropertyPositionObject.strength`)
7. generování rozšířených (spread) pozičních vlastností v každé místnosti
   - zde se konkrétně rozšiřuje pouze vlastnost `WALL`
   - šíření probíhá ve směru opačném k původní políčku a směru k dané poziční vlastnosti, vztahuje se jenom na konkrétní místnost, která je jejím vlastníkem
   - rozšířená poziční vlastnost se ukládá v objektu třídy `MapField` do atributu `spreadObjects` a liší se od původní vlastnosti v síle, tzn. atribut `PositionPropertyObject.strength`, která se zmenšuje tím víc, čím je políčko vzdálenější od původního
8. vygenerování sektorů v místnostech
   - prostor místnosti je rozdělen na síť sektorů, kde je každý sektor (pokud možno)

stejně velký

- každý sektor si extrahuje vlastnosti (stejným způsobem jako v kroku č. 6) z políček příslušných do prostoru jeho výseče

9. propojení všech místností
   - místnost si nastavuje své sousedy podle propojení skrz dveře
10. propojení všech sektorů
    - propojení sektorů v závislosti na sousedech vnitřních políček, tedy pokud má políčko z daného sektoru za souseda pole z jiného sektoru, pak jsou tyto i sektory sousedé

Automatické generování veškerých vztahů v prostředí zajišťuje, že nebude docházet k chybám způsobených např. designérem mapy.

4.3.1.3 Vyhledávání cesty podle pozičních vlastností

V tomto okamžiku máme k dispozici detailně popsaný svět pomocí pozičních vlastností. Dále se zaměříme na algoritmus pro vyhledávání cesty zadané pozičními vlastnostmi, který je zapsán ve třídě `PathFinder`. Ještě než budou popsány prohledávací algoritmy, definujeme struktury, které algoritmus využívá.

Třída, která v sobě drží informaci o vlastnostech cesty a cíle, se nazývá `Checkpoint` a obsahuje tyto atributy:

- `pathProperties` – vlastnosti, které by měla cesta splňovat,
- `targetPosition` – vlastnosti, které musí splňovat cílová pozice.

Oba atributy ve třídě `Checkpoint` jsou stejného typu, a to objekty třídy `PositionObjects`, která zastřešuje možné typy vlastností:

- `staticProperties` – statické vlastnosti reprezentované pozičními vlastnostmi,
- `distanceProperties` – distanční vlastnosti.

`PathFinder` k třídě `Checkpoint` přistupuje prostřednictvím rozhraní `IPathFinderPosition`, které pouze definuje metody na získání atributů `pathProperties` a `targetPosition`.

Jelikož je nutné ohodnotit sektory a pole v průběhu prohledávání, používá `PathFinder` třídy, které obalují prvky obou úrovní prohledávání. Třída, která obaluje `MapUnitSector`, se nazývá `PathFinderSector`, a třída obalující `MapField` je `PathFinderField`.

Třída `PathFinderSector` má strukturu:

- `sector` – reference na objekt typu `MapUnitSector`, který se odkazuje do prostředí,
- `previousSector` – předchozí sektor, ze kterého jsme se dostali na aktuální (rozuměj tato instance) sektor při vyhledávání cesty,
- `cost` – cesta sektoru stanovená při vyhledávání cesty,
- `isInTargetArea` – indikátor, jestli sektor přísluší do cílové oblasti.

Třída `PathFinderField` je velmi podobná předchozí třídě, atibuty mají stejnou funkci, ale jsou jiného typu:

- `field` – reference na objekt typu `MapField`, který se odkazuje do prostředí,
- `previousField` – předchozí pole, ze kterého jsme se dostali na aktuální (rozuměj tato instance) pole při vyhledávání cesty,
- `cost` – cesta pole stanovená při vyhledávání cesty.

`PathFinder` ještě používá podtřídu `PathWithTarget`, která slouží pouze pro předání cesty složené ze sektorů a cílového pole do další fáze prohledávání:

- `sectorPath` – uspořádaný seznam sektorů, které tvoří cestu k cílovému poli,
- `target` – cílové pole.

Nyní bude popsán dvouúrovňový algoritmus vyhledávání cesty zadané pomocí pozičních vlastností. První fáze vyhledávání na úrovní sektorů:

```
FindPathInSectors(MapUnitSector startSector, Checkpoint pathToTarget): PathWithTarget
0       openList, closedList, visitedTargetArea ← empty list
        pathFound ← false
1       targetArea ← chooseTargetArea(pathToTarget)
2       actualSector ← new PathFinderSector(startSector)
3       openList □ actualSector
4       if actualSector □ targetArea then
5               visitedTargetArea □ actualSector
6       while openList ≠ □ & !pathFound do
7               actualSector ← remove first member of openList
8               closedList □ actualSector
9               if |visitedTargetArea| = |targetArea| & visitedTargetArea □ closedList then
10                      pathFound ← true
11              else
12                      for each neighbour: neighbours of actualSector do
13                              newSector ← new PathFinderSector(neighbour)
14                              if newSector □ closedList then continue
15                              if newSector □ targetArea then
16                                      newSector.cost ← actualSector.cost +
countTargetSectorCost(newSector.sector, actualSector.sector, pathToTarget)
```

```
17                                          newSector.isInTargetArea ← true
18                                          if newSector □ visitedTargetArea then
19                                                  visitedTargetArea □ newSector
20                                          else if □ x, x □ visitedTargetArea: x.sector = newSector.sector
                                                    & x.cost < newSector.cost then
21                                                  replace x by newSector in visitedTargetArea
22                                  else
23                                          newSector.cost ← actualSector.cost + countSectorCost(newSector.sector,
pathToTarget) + sectorHeuristic(newSector, actualSector, id of MapUnit where targetArea is, center coordinates of targetArea)
24                                  openList □ newSector in descendant order
25
26      if !pathFound then return null
27      else
28              path ← extractPathFromSectors(visitedTargetArea, pathToTarget)
29              return path
```

Zjednodušeně, algoritmus prohledává prostor na stejném principu jako A* [12], jen je modifikován pro práci s vlastnostmi prostředí a vyhledává cestu s nejvyšší cenou. Rozhodující roli hrají metody pro výpočet ceny cesty na sektor, protože právě v nich se berou v úvahu vlastnosti prostředí. Za zmínku zde stojí, že algoritmus ukončí hledání až v případě, že jsou zkontrolovány všechny sektory z cílové oblasti targetArea. Tím docílíme nalezení nejoptimálnější cesty k cílové pozici.

Následující metoda vybírá množinu sektorů, které splňují požadavky na cílovou oblast:

```
ChooseTargetArea(Checkpoint pathToTarget): set of PathFinderSector
0       staticProperties ← pathToTarget.targetPosition.staticProperties
        distanceProperties ← pathToTarget.targetPosition.distanceProperties
1       sectors ← all sectors with static property: remove first member of staticProperties
2       for each property: staticProperties do
3               sectors ← sectors ∩ all sectors with static property: property
4       for each property: distanceProperties do
5               sectors ← sectors □ all sectors with distant property: property
6       targetSectors ← convert sectors to set of PathFinderSector
7       return targetSectors
```

Hodnotu sektoru, který spadá do cílové oblasti, počítá metoda CountTargetSectorCost(...). Kromě vlastností pro cíl, bere v potaz i vlastnosti cesty, i když už ne s takovou váhou.

```
CountTargetSectorCost(MapUnitSector sector,MapUnitSector previousSector, Checkpoint pathToTarget): double
0       sectorTargetProperties ← sector.properties ∩ pathToTarget.targetPosition.staticProperties
        sectorPathProperties ← sector.properties ∩ pathToTarget.pathProperties.staticProperties
        cost ← basic cost for target sector
1       for each property: sectorPathProperties do
2               cost ← cost + property.strength
3       for each property: sectorTargetProperties do
4               cost ← cost + property.strength
5       cost ← cost * |sectorTargetProperties|
6       return cost
```

Metoda na výpočet ceny obyčejného sektoru Count-Sector-Cost(MapUnitSector nextSector, Checkpoint pathToTarget): double je téměř totožná s předchozí metodou až na výjímku, kdy se do

ceny nezapočítávají vlastnosti cíle (sectorTargetProperties).

Do výpočtu hodnoty obyčejného sektoru přispívá heuristika, která modifikuje cenu, pokud je aktuální sektor vzdálenější od cílové pozice než sektor, ze kterého jsme se dostali na aktuální, a v případě, kdy je aktuální sektor v jiné místnosti než cílová oblast.

```
SectorHeuristic(PathFinderSector actualSector, PathFinderSector previousSector, MapUnit targetAreaUnit, Point2D
targetAreaCenterPosition): double
0       actualToTarget ← distance(position of actualSector in map, targetAreaCenterPosition)
        previousToTarget ← distance(position of previousSector in map, targetAreaCenterPosition)
1       heuristic ← 0
2       if actualToTarget > previousToTarget then heuristic ← 2*(previousToTarget-actualToTarget)
3       if actualSector.parentUnit ≠ targetAreaUnit then heuristic ← heuristic+punishment, punishment < 0
4       return heuristic
```

Za zmínku stojí ještě metoda extractPathFromSectors(PathFinderSector[] visitedTargetArea, Checkpoint pathToTarget): PathWithTarget. V této metodě se nejprve zvolí konkrétní cílová pozice ze sektorů uložených ve visitedTargetArea tak, že jsou ohodnoceny jednotlivé pole z každého sektoru metodou countFieldCostAsTarget(MapField field, Checkpoint pathToTarget): double a následně je vybráno pole s nejvyšší cenou.

```
CountFieldCostAsTarget(MapField field, Checkpoint pathToTarget): double
0       fieldProperties ← field.properties ∩ pathTotarget.targetPosition.staticProperties
        cost ← 0
1       for each property: fieldProperties do
2               cost ← cost + property.strength
3       for each distanceProperty: pathToTarget.pathProperties.distanceProperties do
4               distanceDifference ← manhattanDistance(field, distanceProperty.target) –
        distanceProperty.minDistanceFromTarget
5               angleDifference ← 1
6               if distanceProperty.target ≠ field.position then
7                       cosOfAngle ← cos of angle between vectors (field.position.x – distanceProperty.target.x,
        field.position.y – distanceProperty.target.y) and (distanceProperty.directionFromTarget.dx,
        distanceProperty.directionFromTarget.dy)
8                       angleDifference ← arccos(cosOfAngle)
9               if distanceDifference < 0 then
10                      cost ← cost – punishment*(angleDifference + 1), punishment >> 0
11              else
12                      cost ← cost – m*(distanceDifference + angleDifference), m > 0
13      return cost
```

Po první fázi jsme získali instanci struktury PathWithTarget, která se předává do druhé fáze:

```
findPathInFields(MapField start, PathWithTarget pathWithTarget, Checkpoint pathToTarget): MapField[]
0       openList, closedList ← empty list
        sectorPath ← pathToTarget.sectorPath
        target ← pathToTarget.target
        pathFound ← false
1       distanceStartToTarget ← distance(start, target)
2       openList □ new PathFinderField(start)
3       if actualSector □ targetArea then
4               visitedTargetArea □ actualSector
5       while openList ≠ □ & !pathFound do
6               actualField ← remove first member of openList
```

```
7               closedList □ actualField
8               if actualField = target then
9                       lastField ← actualField
10                      pathFound ← true
11                      continue
12              for each neighbour: neighbours of actualField do
13                      newField ← new PathFinderField(neighbour)
14                      newField.previousField ← actualField
15                      if newField □ closedList || newField.field.type = MapField.OBSTACLE then continue
16                      if newField.field = target then
17                              lastField ← actualField
18                              pathFound ← true
19                              break
20                      if sector of newField □ sectorPath then
21                              closedList □ newField
22                      if abs(sectorPath.indexOf(sector of newField) – sectorPath.indexOf(sector of actualField)) > 1
        then
23                              continue
24                      newField.cost ← actualField.cost + countFieldCost(newField.field, target, actualField.field,
        pathToTarget)
25                      openList □ newField in descendant order
26
27      if !pathFound then return null
28      else
29              path ← empty list
30              while lastField ≠ null do
31                      path □ lastField at first place
32                      lastField ← lastField.previousField
30              return path
```

Tento algoritmus prohledává prostor téměř stejným způsobem jako jeho předchůdce v první fázi. Ohodnocování polí je analogické, hlavní rozdíl spočívá v určené cílové pozici. Už ji netvoří oblast, ale konkrétní pole a prohledávání končí v okamžiku, kdy je toto pole dosaženo. Jak bylo zmíněno v průběhu této kapitoly, v úvahu se berou pouze pole, které spadají do cesty složené ze sektorů nalezených v první fázi.

4.3.2 HTN plánování

Generování plánů pomocí HTN plánovače zajišťuje třída `HTNPlanner`. Tato třída neimplementuje samotný HTN plánovač, ale používá implementaci JSHOP 1 [13] (dále pouze JSHOP). JSHOP je volně dostupný formou balíku tříd, což nám umožňuje vložit jeho instanci do atributů třídy `HTNPlanner` a volat plánovací metody přímo v kódu, aniž bychom museli vytvářet most v případě externího plánovače napsaného v jiném jazyku. Hlavní třídou JSHOPu, která zprostředkovává metody potřebné k plánování, je `JSJshop`. Třída `HTNPlanner` je popsána těmito atributy:

- `planners` – soubor několika instancí třídy `JSJshop`, kde každá instance je vázána na nějaký druh akce.

Důvodem vytvoření více instancí třídy plánovače JSHOP je pouze udržení přehlednosti

plánů. Plány pro všechny akce by mohly být obsažené pouze v jedné doméně, což by tuto doménu udělalo velmi chaotickou a špatně udržovatelnou. Jako malou výhodu můžeme brát i menší prohledávaný prostor.

4.3.2.1 Inicializace plánovací domény

Konstruktoru třídy `PathFinder` je předán argument s cestou k adresáři, který obsahuje reprezentace plánovacích domén rozdělených podle různých typů akcí. Výčet těchto akcí je dán typem `HTNActionType`:

```
HTNActionType = {STRIKE_INTO_SMALL_ROOM, STRIKE_INTO_MEDIUM_ROOM,
STRIKE_INTO_LARGE_ROOM, STRIKE_INTO_SHORT_HALL, STRIKE_INTO_MEDIUM_HALL,
STRIKE_INTO_LONG_HALL}
```

Spojením každého prvku tohoto typu a adresáře dostaneme cestu k plánovací doméně. Inicializace plánovací domény JSHOPu probíhá při konstrukci nové instance třídy `JSJshop`. Konstruktoru se tedy předloží cesta k souboru s doménou:

```
new JSJshop(domainsDirectory + actionType + ".shp")
```

4.3.2.2 Plánování

Plánování probíhá voláním metody `plan(problem, action)` na instanci třídy `HTNPlanner` ve čtyřech krocích:

1. výběr vhodné instance třídy `JSJshop` podle typu akce
   - další kroky se už týkají pouze vybrané instance třídy `JSJshop`
2. smazání množiny problémů z přechozího plánování
   - metoda `clearProblemSet()`
3. inicializace plánovacího problému
   - metoda `initProblem(problem)`, problém bude popsán v kapitole 4.3.4
4. řešení plánovacího problému a získání plánu
   - metoda `solveProblem()`

4.3.3 POSH plánování

Úlohu reaktivního plánovače plní třída `SimplePOSHPlanner` postavená na mechanismu SimplePOSH. Tato třída obsahuje atributy:

- `plan` – reprezentace plánu pro selekci akcí,

- `world` – reference na jednotku, pro kterou mechanismus plánuje.

Plán `plan` je instancí třídy `SimplePOSHPlan`, která svou strukturou připomíná strom s uzly, které odpovídají elementům popsaných v kapitole o POSH plánování. Atribut `world` je reprezentován rozhraním `ISimplePOSHWorld`. Toto rozhraní poskytuje pouze jednu metodu, a to `boolean isConditionSatisfied(Condition condition)`, která je vyvolána v okamžiku, kdy se snaží plánovací mechanismus ověřit platnost nějaké podmínky v průběhu prohledávání struktury `plan`.

4.3.3.1 Inicializace plánovací domény

Inicializace plánovací domény probíhá v konstruktoru třídy `SimplePOSHPlanner`, kterému je předán argument s cestou k souboru s doménou (dalším argumentem je třída implementující `ISimplePOSHWorld`). Tato doména je transformována pomocí parsovacího nástroje ANTLR do atributu `plan`.

4.3.3.2 Plánování

Jednotka, pod kterou instance plánovače spadá, vyvolá plánování metodou `planNextAction()`. Návratovou hodnotou je uspořádané pole proveditelných akcí zastoupených instancemi třídy `SimpleAction`.

4.3.4 Agent

Reprezentantem herního agenta je třída `HTNAgent`. Tato třída představuje koncovou jednotku, která je vykonavatelem veškerých akcí simulujících CQB. Disponuje proto strukturami schopnými udržovat informace o aktuálně vykonávaném plánu, o rozvržení posloupnosti plánů, apod. K hlavním atributům třídy `HTNAgent` patří:

- `id` – jednoznačný identifikátor agenta,
- `commander` – reference na instanci třídy `HTNCommander` jako na hlavní řídící jednotku, pod kterou agent spadá,
- `position` – reprezentace pozice v prostředí (`WorldMap`),
- `role` – role agenta, pod kterou vystupuje v plánování,
- `aimAngle` – úhel, pod kterým agent aktuálně pozoruje prostředí,
- `aimAbility` – parametricky definované mířící schopnosti,
- `reachedEvents` – události týkající se stavu akcí (provedené/neprovedené),
- `taskCalendar` – objekt reprezentovaný třídou `TaskCalendar`, uchovává v sobě veškeré

akce včetně exekučních časů, které musí agent vykonat,

- `actualTimePlan` – aktuálně zpracovávaný plán jako instance třídy `TimePlan`,
- `actualCommand` – aktuálně zpracovávaná hlavní akce z plánu `actualTimePlan`,
- `expandedCommand` – akce z `actualTimePlan` transformovaná do primitivních hlavních akcí, tzn. akcí, které modifikují pozici agenta,
- `actualAction` – primitivní hlavní akce připravená k provedení agentem,
- `parallelTask` – paralelní primitivní úloha, kterou je agent schopný vykonat, aniž by měnil pozici (vizuální kontrola nějaké oblasti, střelba), tedy může být provedena současně s primitivní hlavní akcí z `expandedCommand`.
- poznámka - všechny primitvní akce agenta jsou vymezené výčtovým typem:

```
AgentSimpleCommand = {AIM_FOR, MOVE_TO, WAIT}
```

Velmi důležitým atributem je `taskCalendar` zastoupený třídou `TaskCalendar`, která transformuje posloupnost agentových akcí do jisté formy kalendáře. Celá posloupnost akcí je rozdělena do menších podposloupností. Každá taková podposloupnost spadá v tomto kalendáři pod časovou značku, která signalizuje, jestli může agent danou podposloupnost začít vykonávat. Struktura třídy `TaskCalendar` je:

- `calendar` – seznam všech podposloupností agentových akcí, kde každá podposloupnost představuje instanci třídy `TimePlan`.

Třída `TimePlan` je tvořena atributy:

- `startTime` – časová značka daná třídou `TimeStamp`, která reprezentuje:
  - absolutní i relativní čas,
  - relativní čas je určen identifikátorem události, po splnění této události se stává i tato časová značka platná.
- `plan` – posloupnost akcí, která začíná být proveditelná v okamžiku dosažení platnosti časové značky `startTime`.

Pokud chce agent získat novou podposloupnost akci z kalendáře `taskCalendar`, předloží mu aktualní čas v simulaci a atribut `reachedEvents`. Mechanismus v `taskCalendar` zkontroluje časovou značku u následující instance třídy `TimePlan` a označí ji jako aktuální (`actualTimePlan`) v případě, že:

- časová značka je absolutní a menší nebo rovna aktuálnímu času,
- časová značka je relativní a `reachedEvents` obsahují identifikátor události, která plní

tuto časovou značku.

Třída `HTNAgent` je potomkem třídy `Agent`. Ta deklaruje metody `executeNextTask(gameEvents, agentsEvents)`, která vyvolá u agenta zpracování primitivní hlavní akce, a `executeParallelTasks(agentEvents)`, která se věnuje provedení paralelní akce.

4.3.5 Hlavní řídící jednotka

Hlavní řídící jednotka zastřešuje veškeré plánování, jak pro přizpůsobení chování skupiny agentů pro scénář CTF, tak pro akce v rámci CQB. Jednotku představuje třída `HTNCommander` s atributy:

- `game` – reference na herní scénář,
- `teamId` – identifikátor týmu, který jednotka zastupuje,
- `time` – aktuální čas simulace,
- `worldMap` – odkaz na prostředí, které je zprostředkováno instancí třídy `WorldMap`,
- `pathFinder` – objekt třídy `PathFinder` umožňující vyhledávání cesty zadané pozičními vlastnostmi,
- `htnPlanner` – instance třídy `HTNPlanner`, pomocí které jednotka plánuje CQB akce,
- `htnPlanParser` – parser reprezentovaný třídou `PlanParser`, převádí plán z `htnPlanner` na plán složený pomocí objektů, které je jednotka a agenti dokáží dále zpracovat,
- `poshPlanner` – instance třídy `SimplePOSHPlanner`, která má na starosti plánování v závislosti na herním scénáři,
- `gamePlan` – aktuální plán vygenerovaný pomocí `poshPlanneru`,
- `executingAction` – příznak indikující, jestli řídící jednotka zpracovává nějakou akci z `gamePlan`,
- `teamActions` – seznam týmových akcí a agentů, kteří už dokončili svůj podíl na aktuální týmové akci (jako aktuální se považuje akce, která je v seznamu na první pozici), týmové akce jsou specifikované výčtovým typem:

  ```
  HTNActionType = {ENTRY_FORMATION, PENETRATION}
  ```

- `actionPath` – cesta do nějaké místnosti v prostředí, která byla vyhledána v závislosti na aktuální akci z `gamePlan`,
- `agents` – skupina agentů řízenou touto jednotkou, tvoří ji instance třídy `HTNAgent`,

- `agentsRoles` – role, které agentům přiřazuje hlavní řídící jednotka pro plánování CQB akcí,
- `reachedGameEvents` – události na herní úrovni,
- události agentů:
  - `reachedTaskEvents` – události vyvolané splněním jistých akcí agenty,
  - `newlyReachedTaskEvents` – nově vzniklé události vyvolané splněním jistých úkolů agenty,
  - `startedTaskEvents` – události vyvolané začátkem exekuce akcí agenty.

Třída `HTNCommander` dědí od třídy `CTFCommander`, která nutí každého potomka implementovat metody zajišťující exekuci akcí v rámci simulačního kroku. Jedná se o metody `executeNextStep(time, gameEvents, agentsEvents)` a `executeParallelTasks(agentsEvents)`. První metoda provádí hlavní akce, tzn. akce, které mění pozici agentů, a druhá spouští akce, které můžou agenti provádět „při chůzi", tzn. jedná se hlavně vizuální kontrolu prostředí.

4.3.6 CTF scénář

Herní scénář, zde konkrétně CTF, je implementován do třídy `CTFGame`. Tato třída koriguje běh celé hry, tzn. informuje účastníky hry, kdy mohou provést další herní krok, sbírá od nich informace o dosažených událostech a zároveň dále tyto informace přeposílá. Struktura této třídy je tvořena:

- `worldMap` – odkaz na prostředí zprostředkované instancí třídy `WorldMap`,
- `time` – aktuální čas simulace,
- `commanders` – hlavní řídící jednotky jako účastníci hry zastoupené instancemi třídy `CTFCommander`,
- `flags` – reprezentace vlajek, o které účastnící vzájemně soupeří,
- `actualGameEvents` – seznam aktuálních událostí na úrovní scénáře hry, každá událost je popsána identifikátorem týmu a typem zastoupeným prvkem z: `GameEventType = {AGENT_KILLED, ENEMY_KILLED, MY_FLAG_STOLEN, MY_FLAG_DROPPED, ENEMY_FLAG_STOLEN, ENEMY_FLAG_DROPPED, ENEMY_GOT_MY_FLAG_TO_FLAGPOINT, TEAM_GOT_ENEMY_FLAG_TO_FLAGPOINT}`
- `actualAgentsEvents` – seznam aktuálních událostí vyvolaných agenty, každá událost tohoto druhu je specifikována pomocí stručných informací o agentovi (pozice, směr

pohledu, zaměřený nepřítel, ...), který tuto událost přivodil, a typem z:
`AgentEventType = {AGENT_IN_UNIT_WITH_FLAG, AGENT_IN_UNIT_WITH_ENEMY_FLAG, AGENT_CAN_SEE_ENEMY_CARRIER, AGENT_CAN_SEE_ENEMY, AGENT_AT_POSITION_WITH_ENEMY_FLAG, AGENT_AT_POSITION_WITH_FLAG, AGENT_AT_FLAG_POINT, AGENT_SHOOTS_ENEMY}`

- `gameStatistics` – záznam herních statistik, konkrétně počet ukradených vlajek a eliminovaných nepřátel pro každého účastníka.

4.3.7 Tok hry

Celý běh simulace začíná u uživatele, který dává příkaz k provedení dalšího simulačního kroku kliknutím na příslušné tlačítko (více v příloze D). O zbylou režii se stará instance třídy `CTFGame`, která umožňuje účastníkům hry (hlavním řídícím jednotkám) provést další krok v realizaci svých plánů.

4.3.7.1 Simulační krok v rámci třídy CTFGame

Třída `CTFGame` disponuje metodou `executeNextStep()`, která je vyvolána při požadavku na provedení simulačního kroku. Metoda obsahuje tyto kroky:

```
executeNextStep()
1       agentsEvents ← checkAgentsPositions()
2       gameEvents ← checkGameState()
3       time ← time + 1
4       for each commander: commanders do
5               commander.executeNextStep(time, gameEvents[commander.teamId],
        gameEvents[commander.teamId])
6       agentsVisibility ← checkVisibility()
7       for each commander: commanders do
8               commander.executeParallelTasks(agentsVisibility[commander.teamId])
9       postCheckGameState()
```

Metoda `checkAgentsPositions()` kontroluje pozice agentů všech účastníků. Pokud se agent nachází v místnosti s vlajkou (týmovou nebo oponentovou), je tato informace zapsána formou události vyvolané agentem. Metoda `checkGameState()` prochází události z atributu `actualGameEvents` a třídí je v závislosti na jejich významu a adresátovi. `actualGameEvents` a `actualAgentEvents` jsou plněny skrz metody přijímající události odpovídajících typů, když účastnící hry provádí akce v rámci simulačního kroku. Exekuci paralelních úloh každého z účastníků předchází kontrola viditelnosti agentů metodou `checkVisibility()`. Závěrečné vyhodnocení událostí `actualAgentsEvents` vyvolaných paralelními akcemi je provedeno v metodě `postCheckGameState()`.

4.3.7.2 Simulační krok v rámci třídy HTNCommander

V jednom simulačním kroku jsou na instanci třídy `HTNCommander` volány dvě metody, které mají na starosti exekuci hlavních a paralelních akcí. Jedná se o metody `executeNextStep(time, gameEvents, agentsEvents)` a `executeParallelTasks(agentsEvents)`. Pořadí jejich volání je určeno třídou reprezentující herní scénář, jak je popsáno v předchozí kapitole. Věnujme se teď procesům, které se spustí zavoláním těchto metod.

Metoda `executeNextStep(time, gameEvents, agentsEvents)` obsahuje:

```
executeNextStep(time, gameEvents, agentsEvents)
1       checkGameState(gameEvents, agentsEvents)
2       plan()
3       setTime(time)
4       broadcastTimeAndEvents(time)
```

Stručně popsáno, nejprve řídící jednotka zkontroluje stav hry. Při této kontrole může být generován plán na úrovni POSH vrstvy. Ve druhém kroku dochází k podrobnému rozplánování akcí na úrovni HTN plánování. Plán je pak distribuován mezi agenty. Třetí krok se stará pouze o aktualizaci času simulace. Nakonec řídící jednotka rozešle informaci o čase a dosažených událostech mezi agenty a vyvolá na nich metodu pro exekuci simulačního kroku v rámci hlavních akcí (pohyb agentů). Dále se zaměříme na detailnější popis kroku 1, 2 a 4.

```
checkGameState(gameEvents, agentsEvents)
1       if gamePlan is empty & □ agent □ agents finished all tasks then
2               gamePlan ← poshPlanner.planNextAction()
3               executingAction ← false
4       else if replanningRequired(gameEvents, agentsEvents) then
5               interruptAllPlansToAllAgents()
6               gamePlan ← poshPlanner.planNextAction()
7               executingAction ← false
```

Z metody `checkGameState(gameEvents, agentsEvents)` plyne, že k plánování na POSH vrstvě dochází v případě, kdy všichni agenti ze skupiny dokončili své plány a navíc `gamePlan` neobsahuje žádnou akci, nebo pokud události `gameEvents` a `agentsEvents` jsou natolik důležité, že je vynuceno přeplánování, což obnáší i okamžité přerušení veškerých plánů agentů.

Metoda `plan()` se věnuje dalšímu rozplánování akcí obsažených v `gamePlan`. Některé akce plánované na POSH úrovni mohou být primitivní, tedy proveditelné přímo hlavní řídící jednotkou. Primitivní akce jsou charakterizovány tím, že pro jejich vykonání není potřeba celé skupiny agentů. Akce zaměstnávající celou skupinu agentů, tzn. neprimitivní akce, má na starosti z jedné části řídící jednotka a z druhé HTN plánovač. Neprimitivní akce se týkají

přechodu mezi místnostmi, takže úlohou řídící jednotky je vyhledat cestu složenou z místností a postupně nechat HTN plánovač organizovat přesun agentů z jedné místnosti do druhé.

```
plan()
1       if not actionExecuted then
2               action ← get first element from gamePlan
3               actionExecuted ← true
4               if action is primitive then
5                       commander plans action
6               else
7                       actionPath ← commander searches path according to action
8       if not (actionPath is empty) & □ agent □ agents finished all tasks then
9               planStrikeIntoRoom(get first element from actionPath)
```

Pokud je vyvolána metoda `planStrikeIntoRoom(mapUnit)`, bude plánován taktický manévr vstupu do místnosti určené argumentem `mapUnit`:

```
planStrikeIntoRoom(mapUnit)
1       teamPosition ← get map unit where squad of agents is situated
2       action ← choose action type from HTNActionType according to the type of mapUnit
3       entryPoint ← choose entry point joining teamPosition and mapUnit for action
4       additionalInfo ← getAdditionalInformationForAction(action, mapUnit, entryPoint)
5       task ← generate task for HTN planner
6       problem ← generateLocalProblem(action, additionalInfo, task, agents, teamPosition, mapUnit)
7       plan ← htnPlanner.plan(problem, action)
8       parsedPlan ← htnPlanParser.parsePlan(plan)
9       readPlan(parsedPlan)
```

V prvním kroku je určena pozice jako místnost, kde se nachází skupina agentů. Potom je zvolena akce v závislosti na typu cílové místnosti, např. pokud je cílová místnost typu `SMALL_ROOM`, vybere se akce `STRIKE_INTO_SMALL_ROOM`. Řídící jednotka tímto jednoduchým krokem pomáhá plánovači v omezení se na specifickou plánovací doménu. Ve třetím kroku řídící jednotka volí nejvhodnější vstupní prostor do cílové místnosti. Potom jsou vygenerovány informace obsahující konverzi mezi pozičním systémem prostředí simulace a plánovací domény, popis rolí agentů a další specifické informace v závislosti na vybrané akci. V pátém kroku je popsána úloha pro HTN plánovač a následně je vygenerovaná kompletní reprezentace problému sdružující dodatečné informace k akci, popis cílové a aktuální (pozice skupiny agentů) místnosti a nakonec informace o úloze. Takto popsaný problém je společně s typem akce předán HTN plánovači. Výsledný plán je pomocí instance třídy `PlanParser` transformován do objektů, se kterými dokáže řídící jednotka pracovat. Velmi důležitou roli sehrává metoda `readPlan(plan)`, která obsahuje tyto kroky:

```
readPlan(plan)
1       for each command: plan do
2               if command is executed by commander then
3                       commander processes command
```

```
4               else if command is executed by an agent then
5                       agent ← get executor from command
6                       localize(command)
7                       generateEventForCommand(command)
8                       if command is executed in relative time then
9                               relatedAgent ← get agent to which execution of command is related
10                              lastEvent ← get event for last command of relatedAgent
11                              add command to agent.taskCalendar with relative time stamp given by
    lastEvent
12                      else
13                              add command to agent.taskCalendar
```

Metoda `readPlan(plan)` tedy prochází celý plán uspořádaně a rozděluje akce (`commands`) exekutorům, kterými jsou buď agenti, nebo řídící jednotka. V případě, že exekutorem je nějaký agent, prochází akce fází lokalizace, ve které se transformují vlastnosti zadané pomocí pozičního systému plánovače do systému použitého v simulaci. Dále je pro akci vygenerována událost, která slouží jako identifikátor akce a nastane v okamžiku jejího dokončení. Nakonec je akce zařazena do kalendáře agenta. Pokud se vykonání této akce vztahuje na akci jiného agenta, získá se skrz `taskCalendar` tohoto agenta událost závislé akce a pod touto událostí je vložena původní akce do kalendáře svého agenta.

Tímto jsme uzavřeli metodu `plan()`. Poslední krok v `executeNextStep(time, gameEvents, agentsEvents)` tvoří metoda `broadcastTimeAndEvents(time)`, která aktualizuje informaci o čase a dosažených událostech v každém agentovi a následně na každém vyvolá metodu `executeNextTask()`.

Metoda `executeParallelTasks(agentsEvents)` pouze volá na každém agentovi stejnojmennou metodu, která řídí provádění paralelních akcí.

4.3.7.3 Simulační krok v rámci třídy HTNAgent

V třídě `HTNAgent` je zpracování jednoho simulačního kroku rozloženo do provedení hlavních a paralelních úloh. Nadřazená jednotka, kterou je v tomto případě `HTNCommander`, určuje, v jakém okamžiku se tak stane. `executeNextTask(gameEvents, agentsEvents)` je metoda, která zajišťuje provedení hlavních akcí. `executeParallelTasks(agentEvents)` je metoda obstarávající provedení paralelních úloh.

Metoda `executeNextTask(gameEvents, agentsEvents)` funguje následujícím způsobem:

```
executeNextTask(gameEvents, agentsEvents)
1       setNextTask()
2       executeNextTask()
3       checkIfCommandAccomplished()
```

Úkolem metody `setNextTask()` je vybrat primitivní akci, která bude následující metodou `executeNextTask()` provedena. Poslední metoda `checkIfCommandAccomplished()` kontroluje, jestli naposledy vykonaná primitivní akce ukončila nadřazenou akci, tzn. akci na úrovni elementu plánu generovaného HTN plánovačem.

Důležitou roli hraje metoda `setNextTask()`:

```
setNextTask()
1       if agent does not perform any action then
2               if expandedCommand is not empty then
3                       actualAction ← get first element of expandedCommand
4               else if actualTimePlan is not empty then
5                       actualCommand ← get next action from actualTimePlan
6                       expandCommand(actualCommand)
7                       setNextTask()
8               else if next subplan from taskCalendarCanBeExecuted then
9                       actualTimePlan ← get next subplan from taskCalendar
10                      setNextTask()
11              else
12                      actualAction ← some default action
```

V průběhu této metody je vybrána primitivní akce `actualAction`. Než se ale algoritmus k této akci dopracuje, prochází několik struktur, kde každá obsahuje akce na různých úrovních proveditelnosti. Dejme tomu, že agent nezačal exekuovat žádný z plánů v `taskCalendar`. Začněme průchod od řádku osm. V podmínce se agent dotazuje struktury `taskCalendar`, jestli může spustit provádění první podposlounosti akcí, kterou `taskCalendar` obsahuje. Pokud ano, je tato podposlounost uložena do `actualTimePlan` a metoda rekurzivně zavolá sama sebe. Nyní se dostáváme na čtvrtý řádek. `ActualTimePlan` je definovaný (berme případ, že je neprázdný) a tak se z něj odebere první akce a uloží do `actualCommand`. Jelikož se jedná o akci z HTN plánu, je nutné ji přeložit do primitivních akcí agenta pomocí metody `expandCommand(actualCommand)`. Metoda `expandCommand(actualCommand)` zároveň rozřazuje akce do hlavních a paralelních. Pokud spadá akce mezi paralelní, je převedena na primitivní paralelní akci a uložena do struktury `parallelTask`. Pokud se akce řadí mezi hlavní, je v závislosti na jejím typu převedena na primitivní hlavní akce a uložena do `expandedCommand`. V tomto okamžiku jsme v metodě `setNextTask()` na sedmém řádku, kdy opět volá rekurzivně sama sebe. Pokud byla akce v `actualCommand` paralelní, dostaneme se opět na řádek č. 4 nebo 8. V opačném případě máme naplněnou strukturu `expandedCommand`, můžeme z ní odebrat první prvek a uložit jej do `actualAction`, čím tento algoritmus končí.

Metoda `checkIfCommandAccomplished()` kontroluje, zda se provedly všechny primitivní hlavní akce spadající pod `actualCommand`. V případě, že agent tuto akci dokončil (všechny související primitivní hlavní akce), zavolá se příslušná metoda na atributu `commander`, která informuje hlavní řídící jednotku o splnění akce identifikované skrz událost (byla ji přiřazena v

metodě `readPlan(plan)` na instanci třídy `HTNCommander`).

4.4 Výsledek implementace

Celý projekt je implementován v jazyce Java, jak bylo napsáno v kapitole 4.2, a přístupný na DVD mediu v příloze E, která obsahuje i informace o programátorské dokumentaci a dalších užitečných součástech projektu. Simulační program je zobrazen skrz své grafické rozhraní (viz obrázek 4.6), které nabízí několik ovládacích a monitorovacích prostředků. Jejich popisu je věnována uživatelská dokumentace obsažená v příloze D.

Obrázek 4.6: *Vzhled grafického rozhraní simulačního programu.*

5 Experimenty

Experimenty jsou přizpůsobeny aktuální funkčnosti simulačního programu. Ještě není možné testovat kompletní scénář CTF, protože schopnosti jednání jednotky agentů jsou pouze na úrovni taktického procházení malých místností. Nicméně agenti nechodí bezduše po mapě, ale směřují do místnosti, kde se nachází nepřátelská vlajka. Podle takto omezeného chování jsou navrženy dva testovací scénáře.

V prvním scénáři se oponující agenti chovají podle těchto bodů:

- jsou rozmístěni náhodně na mapě,
- jsou statičtí, tzn. nemění své pozice,
- vizuálně pokrývají vstupy do místnosti, ve které se nachází.

Skupina agentů začíná ve stejné místnosti a v tomto scénáři uspěje, pokud se dostane do místnosti s nepřátelskou vlajkou, aniž by byl v průběhu cesty zabit jakýkoliv týmový agent. V opačném případě vyhrává oponující tým.

Druhý scénář připomíná dobývání budov, protože se oponující agenti řídí podle následujících pravidel:

- jsou rozmístěni náhodně na mapě, nejlépe ale blízko místnosti se svou vlajkou,
- pokud nejsou dostatečně blízko místnosti se svou vlajkou, přesunou se do nějaké,
- pokud jsou dostatečně blízko místnosti se svou vlajkou, jsou statičtí,
- vizuálně pokrývají vchody místnosti, a to způsobem:
  - pokud se nachází v místnosti s vlajkou, tak všechny vchody vedoucí do této místnosti,
  - pokud se nachází v místnosti sousedící s místností, kde se nachází vlajka, pak se zaměřují na všechny vchody mimo ty vedoucí do místnosti s vlajkou.

Úspěch a výchozí pozice skupiny agentů má stejné předpoklady jako u prvního scénáře.

Navíc má skupina agentů lepší mířící schopnosti oproti oponujícímu týmu, protože představuje jednotku pro boj v uzavřených prostorách. A členy jednotky tohoto typu jsou v reálném světě trénovaní profesionálové.

Výsledky jsou znázorněny v tabulkách:

počet nepřátelských agentů	4	5	6
průměrný počet zabitých nepřátel	4	5	4.2
počet vyhraných kol	10/10	10/10	7/10

Tabulka 1: Výsledky prvního scénáře.

počet nepřátelských agentů	4	5	6
průměrný počet zabitých nepřátel	4	4.5	4.2
počet vyhraných kol	10/10	9/10	7/10

Tabulka 2: Výsledky druhého scénáře.

Ze získaných výsledků během testování scénářů je jasně vidět, že skupina agentů navigovaných pomocí CQB taktiky jasně předčila oponující tým.

6 Závěr

Celý projekt má své světlé i stinné stránky. Jelikož celá problematika okolo taktického plánování je poměrně rozsáhlá, v implementaci nejsou pokryté zdaleka všechny pasáže. Na druhé straně se ale podařilo odhalit oblasti, které tvoří stěžejní roli při taktické navigaci.

V průběhu této práce byla navržena struktura reprezentující herní prostředí, která disponuje popisem polohových vlastností. Tyto vlastnosti jsou vystihnuty na úrovni detailů umožňujících strategicky využívat prostor. Závisle na takto popsaném prostředí byl zkonstruován algoritmus schopný vyhledávat cestu, která není popsaná pouze konkrétním cílovým místem ale obecněji, tzn. pomocí polohových vlastností. Dále byl popsán mechanismus plnící roli hlavní řídící jednotky pro skupinu agentů. Tato řídící jednotka využívá, jak výše zmíněného algoritmu pro vyhledávání cesty určené pozičními vlastnostmi, tak plánovačů postavených na POSH a HTN systémech. POSH plánovač v tomto případě používá řídící jednotka k výběru týmové akce, která se odvíjí od scénáře hry. S pomocí HTN plánovače a algoritmu na vyhledávání cesty vede řídící jednotka podřízené agenty při průchodu mezi místnostmi v herním prostředí. Agenti vykazují při těchto akcích známky taktického chování, které má svůj předobraz v jednání speciálních jednotek při boji v uzavřených prostorech.

Součástí této práce je simulační program (příloha E), pomocí kterého jsou znázorněny některé taktické manévry. Jedná se zatím pouze o prototypovou verzi, protože obsahuje plány operující nad jedním typem místností. I když není program kompletní, simulace ukazuje, že jednotka řízená navrženým mechanismem je úspěšnější oproti méně organizovanému oponentovi.

Během implementace byl kladen důraz na rychlost plánování, což vedlo k mnoha optimalizacím v plánovacích a s nimi spojených procesech, i když simulační program pracuje v diskrétním čase. Smyslem urychlení celé režie okolo řízení jednotky je příprava na budoucí zakomponování řídící jednotky do hry UT2004. Nicméně i po dostatečné optimalizaci veškerých procesů spojených s plánováním a řízením nebude hned možné řídící jednotku zasadit do prostředí této hry, dokud se nevyřeší problém spočívající v reprezentaci herního světa, který není popsán pomocí polohových vlastností. Tento problém by mohl být částečně vyřešen, pokud by byla pro herní mapu z UT2004 sestrojena rozlohou odpovídající dvourozměrná mapa a navíc by existoval transformační mechanismus, který by dokázal převádět polohu z 3D prostoru do 2D. Další řešení by bylo vázané už na proces návrhu mapy pro UT2004, kde by každý objekt prostředí obsahoval informaci o vlastnosti, kterou ve vztahu k okolí nabízí, a jejím dosahu.

7 Literatura

[1] Bryson, J. J.: *Inteligence by design: Principles of Modularity and Coordination for Engeneering Complex Adaptive Agents*, dizertační práce, Massachusetts Institute of Technology, 2001

[2] Nau, D. a Ghallab, M.: *Automated Planning: theory and practise*, Morgan Kaufman, 2004, ISBN 1-55860-856-7

[3] Orkin, J.: *Three States and a Plan: The A.I. of F.E.A.R.*, San Jose: Game Developers Conference, 2006

[4] Barták, R.: *Plánování a rozvrhování*, [Online] 2009 http://kti.mff.cuni.cz/~bartak/planovani/index.html

[5] Bryson, J. J.: *POSH Action Selection*, [Online] 2009 http://www.cs.bath.ac.uk/~jjb/web/posh.html

[6] Lekavý, M. a Návrat, P.: *Expressivity of STRIPS-Like and HTN-Like Planning*, Agent and Multi-Agent Systems: Technologies and Applications, Vol. 4496/2007, strany 131-140, 2007

[7] Fikes, R. E. a Nilsson, N. J.: *STRIPS: A New Approach to the Application of Theorem Proving to Problem solving*, London: 2nd IJCAI, 1971

[8] Pogamut [Online], 2009 https://artemis.ms.mff.cuni.cz/pogamut

[9] GameBots [Online], 2009 http://www.cs.rit.edu/~jdb/gamebots/

[10] Close Quarter Battle [Online], 2009 http://www.cs.rit.edu/~jdb/gamebots/

[11] Surynek, P.: *Plánování a rozvrhování: HTN*, [Online] 2009 http://ktiml.mff.cuni.cz/~surynek/index.html.php?select=teaching&term=2005_2006&subject=irregular

[12] Patel, A.: *The A star algorithm*, [Online] 2009 http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#S3

[13] Yaman, F.: *Documentation for JSHOP 1.01*, University of Maryland, 2002

[14] Ilghami, O. a Nau D.: *A General Approach to Synthesize Problem-Specific Planner*, Technical Report CS-TR-4597 a UMIACS-TR-2004-40, 27. říjen 2003

[15] Bresenham's Line-Drawing algorithm, [Online] 2009
http://www.falloutsoftware.com/tutorials/dd/dd4.htm

Příloha A

Plánovací algoritmy

Tato příloha obsahuje několik základních a pokročilých plánovacích algoritmů zapsaných v pseudokódu.

A.1 Dopředné plánování

```
Forward-search(O, s_0, g)
    s ← s_0
    π ← the empty plan
    loop
        if s satisfies g then return π
        applicable ← {a | a is a ground instance of an operator in O,
                          and precond(a) is true in s}
        if applicable = ∅ then return failure
        nondeterministically choose an action a ∈ applicable
        s ← γ(s, a)
        π ← π . a
```

Obrázek A.1: *Pseudokód nedeterministického dopředného plánování.*

A.2 Zpětné plánování

```
Backward-search(O, s_0, g)
    π ← the empty plan
    loop
        if s_0 satisfies g then return π
        relevant ← {a | a is a ground instance of an operator in O
                        that is relevant for g}
        if relevant = ∅ then return failure
        nondeterministically choose an action a ∈ applicable
        π ← a . π
        g ← γ^-1(g, a)
```

Obrázek A.2: *Pseudokód nedeterministického zpětného plánování.*

A.3 SHOP 2

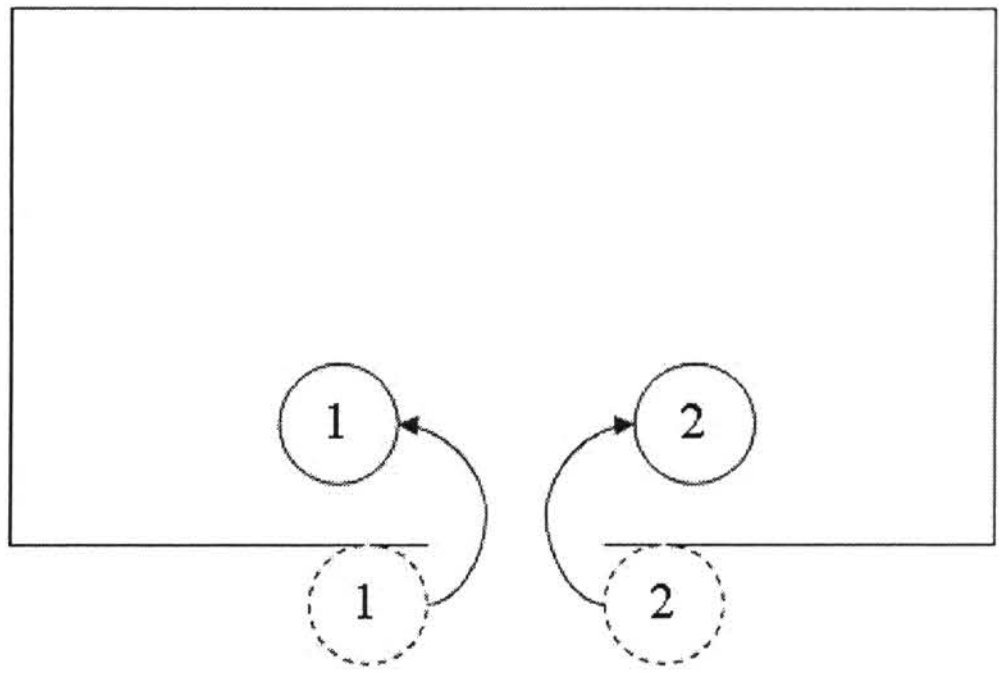
```
procedure SHOP2
1. vstupní plánovací problém P = <d,I,<O,M>>
2. σ ← prázdný plán
3. s ← I
4. if d = ∅ return σ
5. nedeterministicky zvol (n:α) jeden z úkolů v d, který
   nemá předchůdce
6. if α je primitivní then
     active ← množina základních instancí operátorů z O
              aplikovatelných na stav s
     if active = ∅ then return FAILURE
     else
       nedeterministicky zvol jeden operátor o z množiny active
       s ← aplikace operátoru o na stav s
       d ← d po odstranění primitivního úkolu (n:α)
       σ ← σ∪{primitivní úkol příslušný operátoru o}
7. else {α je neprimitivní}
     active ← množina základních instancí metod z M
              aplikovatelných na stav s
     if active = ∅ then return FAILURE
     else
       nedeterministicky zvol jednu metodu m z množiny active
       d ← d po aplikaci metody m na neprimitivní úkol (n:α)
8. goto krok 4.
```

Příloha B

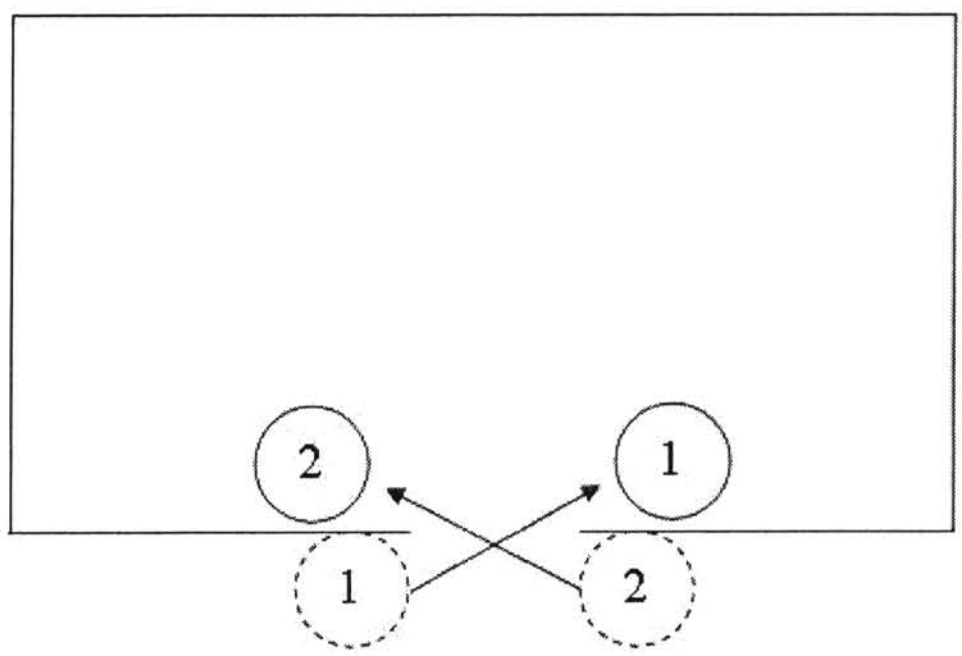
Taktické manévry

V této příloze jdou vizuálně znázorněny některé techniky používané při CQB akcích. První příloha se věnuje technikám vstupů do místností. Příloha B.2 je zaměřená na postup týmu v místnosti tak, aby eliminovala všechny přítomné hrozby („vyčištění místností“).

B.1 Techniky vstupu do místností

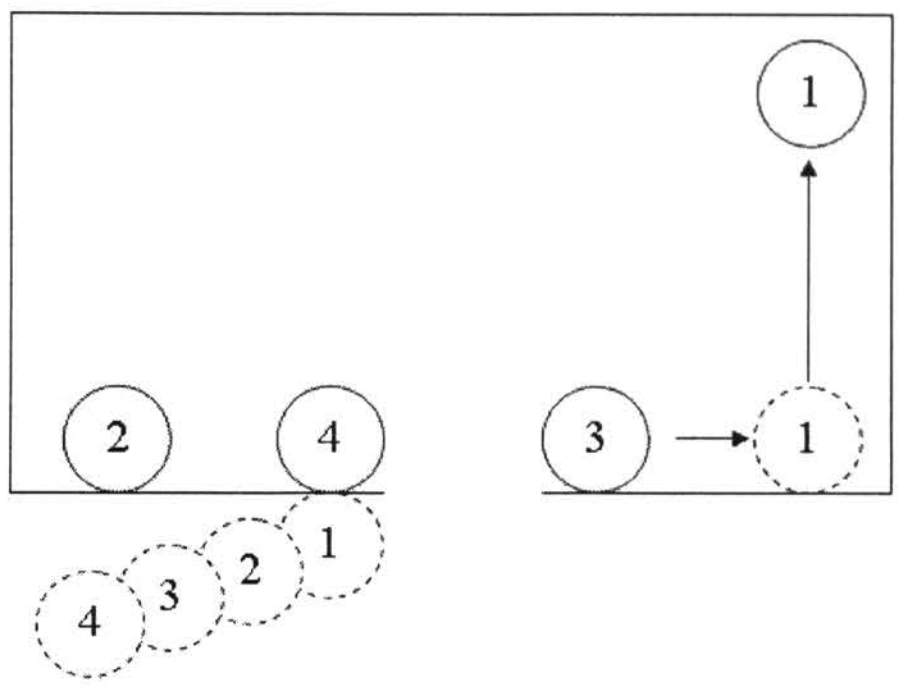


Obrázek B.1: *Vstupní manévr Button Hook.*

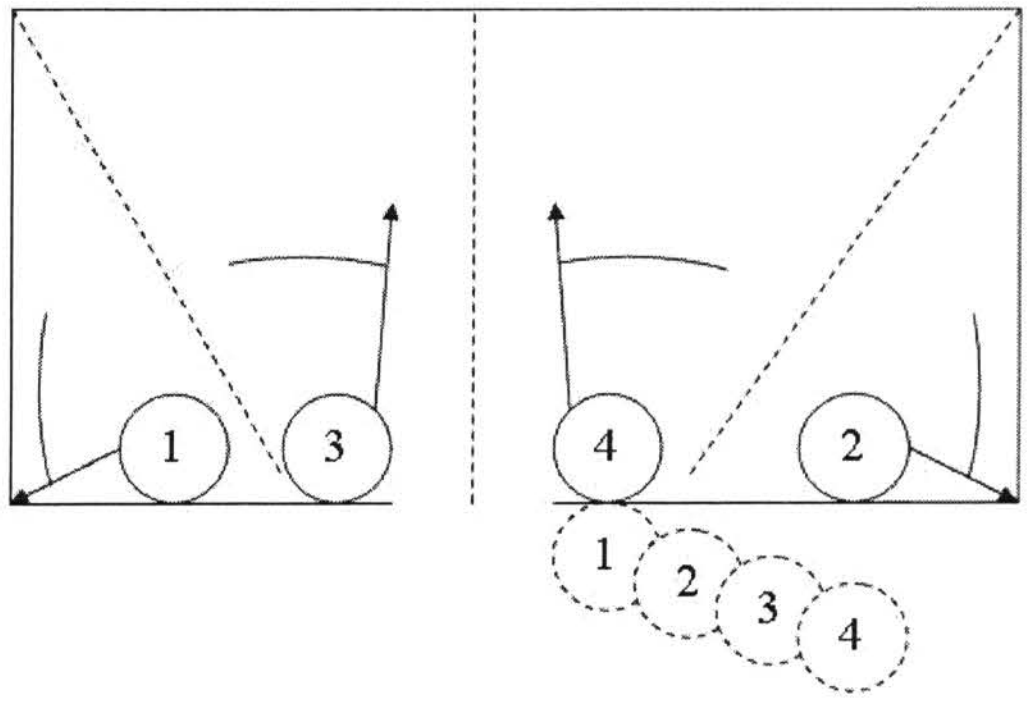


Obrázek B.2: *Vstupní manévr Cross Over.*

B.2 Techniky vyčistění místností



Obrázek B.3: *Delta Wedge manévr.*



Obrázek B.4: *4-man wall flood manévr.*

Příloha C

C.1 Vstupní definice místnosti v XML

```
<?xml version="1.0" encoding="windows-1250" standalone="yes"?>
<map
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsi:noNamespaceSchemaLocation="file://d:/projects/eclipse/team_bc/map/schemas.xsd"
width="100" height="100" gameType="CTF">
    <mapUnit>
        <name>small-room-1</name>
        <type>SMALL_ROOM</type>
        <fromX>11</fromX>
        <toX>18</toX>
        <fromY>7</fromY>
        <toY>12</toY>
    </mapUnit>
    <mapUnit>
        <name>small-room-3</name>
        <type>SMALL_ROOM</type>
        <fromX>21</fromX>
        <toX>26</toX>
        <fromY>13</fromY>
        <toY>18</toY>
    </mapUnit>
    <mapUnit>
        <name>small-room-2</name>
        <type>SMALL_ROOM</type>
        <fromX>19</fromX>
        <toX>26</toX>
        <fromY>7</fromY>
        <toY>12</toY>
        <obstacle x="4" y="1"/>
        <obstacle x="2" y="3"/>
    </mapUnit>
    <horizontalEntryPoint>
        <name>small-door-1</name>
        <type>SMALL_DOOR</type>
        <mapUnit_1>small-room-3</mapUnit_1>
        <mapUnit_2>small-room-2</mapUnit_2>
        <horizontal>
            <fromX>23</fromX>
            <toX>24</toX>
            <y>12.5</y>
        </horizontal>
    </horizontalEntryPoint>
    <verticalEntryPoint>
        <name>small-door-2</name>
        <type>SMALL_DOOR</type>
        <mapUnit_1>small-room-2</mapUnit_1>
        <mapUnit_2>small-room-1</mapUnit_2>
        <vertical>
            <fromY>9</fromY>
            <toY>10</toY>
            <x>18.5</x>
        </vertical>
    </verticalEntryPoint>
    <flagPoint id="0" mapUnit="small-room-3" x="2" y="3"/>
    <flagPoint id="1" mapUnit="small-room-1" x="2" y="2"/>
</map>
```

Obrázek C.1: *Popis mapy ve formátu XML.*

Příloha D

Uživatelská dokumentace simulačního programu

Grafické rozhraní simulačního programu nabízí několik možností, pomocí kterých lze sledovat a ovládat průběh simulace hry. Jeho vzhled s odkazy na užitečné oblasti je znázorněn na obrázku D.1, kde jednotlivá čísla označují:

1 – Map,

2 – Map units,

3 – Position properties,

4 – Entry points,

5 – Agents,

6 – AI Agents,

7 – Simulation execution,

8 – Statistics,

9 – Output console,

10 – Simulation menu.

V této příloze jsou tyto oblasti z funkčního hlediska popsány.

D.1 Map

Grafické zobrazení mapy prostředí, na kterém se odehrává simulace hry. V základním režimu jsou na mapě zobrazené místnosti, překážky, vstupní prostory, místa rezervovaná pro vlajky, samotné vlajky a agenti se svým zorným polem. Mapa prostředí může být rozlehlá do takové míry, že oblast v okně nebude dostatečně velká, aby mohla být celá mapa zobrazená. K posouvání mapy slouží kombinace levý SHIFT + levé tlačítko myši (držet a táhnout do strany).

D.2 Map units

Oblast Map units obsahuje seznam všech místností z prostředí popsaných svými jmény. Vedle jména každé místnosti je zaškrtávací pole. Pokud označíme některou (jednu a více) místnost, zobrazí se její poziční vlastnosti označené v oblasti Position properties.

D.3 Position properties

Seznam všech pozičních vlastností používaným při simulaci. Pokud označíme některou (jednu a více) z vlastností, zobrazí se tyto vlastnosti na mapě v těch místnostech, které jsou zaškrtnuté v Map units. Navíc jsou v seznamu dvě možnosti, které nespadají mezi poziční vlastnosti. Jsou to field connections a sector connections zobrazující propojenost polí a sektorů.

D.4 Entry points

Seznam všech vstupních prostor, které se nachází v aktuálním prostředí. V seznamu jsou zapsány pod svými jmény včetně stručného popisu místností, které propojují. Pokud některé zaškrtneme, zobrazí se na mapě.

D.5 Agents

Agenti, kteří jednají na základě hybridního spojení POSH a HTN plánování. Jsou zobrazení pod svými jmény. Pokud je nějaký z agentů označený, můžeme inicializovat nebo měnit jeho pozici na mapě, a to tak, že klikneme levým tlačítkem do prostoru jedné z místností. Měnit a inicializovat pozice je možné pouze v případě, že simulace je zastavená.

D.6 AI Agents

Oblast s několika ovládacími prvky pro oponující agenty. Pokud chceme přidat nového agenta do hry, stačí označit pole „Add new agent“ a kliknout myší do prostoru nějaké místnosti (stejně jako v kapitole D.5). Agent se pak zobrazí na mapě a jeho jméno je přidáno do seznamu, který se nachází přímo pod polem „Add new agent“. V případě, že chceme změnit pozici některého z přidaných agentů, stačí označit pole „Change position“, označit agenta ze seznamu přidaných a kliknout do mapy. Pod polem „Change position“ se nachází seznam herních strategií. Agenti se řídí podle aktuálně označené.

D.7 Simulation execution

V této nabídce jsou ovládací prvky, které řídí běh simulace. Funkce tlačítka „Next step“ je provedení simulačního kroku, tlačítko „Start simulation“ spouští simulaci a tlačítko „Stop simulation“ simulaci zastavuje. Navíc je v této oblasti informace o aktuálním čase simulace pod názvem „Actual time stamp“. Zaškrtávací pole „animated“ zatím není funkční.

D.8 Statistics

Herní statistiky zobrazené formou tabulky.

D.9 Output console

Textová oblast, kam jsou logovány veškeré výstupní informace ze simulačního programu. Slouží jako nástroj pro pozorování průběhu výpočtů, stavu akcí agentů, apod. Logované informace lze smazat, pokud klikneme prvým tlačitkem myši kamkoliv do textové oblasti.

D.10 Simulation menu

Menu simulace, které nabízí načtení nové mapy a přepnutí do režimu editoru map.

Obrázek D.1: Grafické rozhraní simulačního programu.

Příloha E

E.1 DVD

DVD obsahuje simulační program, potřebné balíky externích programů, programátorskou dokumentaci (E:\team_bc\doc), vývojové prostředí Eclipse, návod na importování projektu do Eclipse a spuštění simulace.

Pokud by si chtěl čtenář prohlédnout plány, které jsou v simulačním programu použité, nalezne je ve složce E:\team_bc\domains\htn a E:\team_bc\domains\simplePOSH.