SLOVENSKÁ TECHNICKÁ UNIVERZITA BRATISLAVA
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Tímový projekt

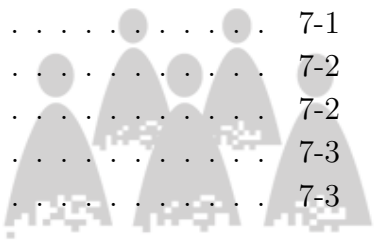
SIMULÁCIA DAVU

Dokumentácia k inžinierskemu dielu

Bc. Michal Fornádeľ
Bc. Adam Pomothy
Bc. Lukáš Pavlech
Bc. Marek Hlaváč
Bc. Daniel Petráš
Bc. Martin Košický

Vedúci tímového projektu: Ing. Peter Lacko, PhD.
Akademický rok: 2011/12

Obsah



Kapitola 1

Úvod

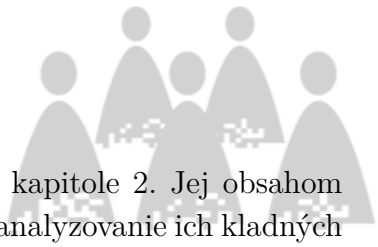
Dokument sa zaoberá problematikou simulovania davu a poskytuje komplexný objektívny pohľad na zadanú problematiku spolu s návrhom a realizáciou riešenia. Aplikácia sa zameriava na simuláciu davu v rámci interiéru a jej výsledkom je schopnosť simulácie stoviek až tisícov agentov umiestnených do prostredia predstavujúceho priestor budovy alebo iného objektu definovaného prostredníctvom mapy. Pri jej vývoja sa tím riadil agilným spôsobom vývoja softvéru SCRUM.

1.1 Účel a rozsah dokumentu

Účelom predkladaného dokumentu je dospieť k analýze, návrhu a implementácii riešenia, ktoré by svojím ponímaním čo najefektívnejšie a najlepšie uspokojilo potreby používateľa požadujúceho produkt z oblasti simulácie davu.

Dokument je výsledkom práce členov tímu v rámci predmetu Tímový projekt na Fakulte informatiky a informačných technológií Slovenskej Technickej Univerzity v Bratislave.

Dokument je určený pre každého používateľa alebo vývojára oboznámeného s problematikou simulovania davu.



1.2 Prehľad dokumentu

Simulácia davu v podaní iných spoločností je rozobratá v kapitole 2. Jej obsahom predstavenie štyroch zaujímavých konkurenčných riešení a zanalyzovanie ich kladných a záporných aspektov. Počas ich analýzy bol hlavný dôraz kladený na pochopenie ich koncepcie a spôsobu fungovania.

Kapitoly 3, 4, 5 a 6 obsahujú popis zložitejších a zaujímavých úloh riešených v jednotlivých šprintoch. Implementačné úlohy sú popísané z hľadiska analýzy problému, návrhu riešenia, implementácie a testovania.

Kapitola 2

Simulácia davu v podaní iných spoločností

Tématikou simulovania davu sa dodnes zaoberá množstvo spoločností. Táto oblasť je pomerne rozsiahla a nedá sa povedať, že by sa každé z predstavovaných riešení sústreďovalo na rovnakú problematiku. Rovnako ponímanie spracovávanej problematiky je rôznorodé a mnohokrát optimalizované pre odlišné úrovne výpočtovej sily počítača. Táto kapitola sa zaoberá analýzou troch konkrétnych prevedení a poukazuje rovnako na ich výhody ako aj nevýhody.

2.1 PathFinder

Pathfinder je jeden z moderných evakuačných simulátorov, ktorý pracuje na základe najnovších poznatkov z oblasti počítačových technológií zameraných na modelovanie pohybu jednotlivcov vo vnútorných prostrediach. Pathfinder poskytuje nástroje nevyhnutné pre tvorbu správnych rozhodnutí vyplývajúcich z vytvárania bezpečnostných a evakuačných systémových návrhu stavieb. Obsahuje podporu viacerých simulačných módov a možnosť nastavenia parametrov agentov podľa určitého scenáru. Keďže sa jedná o simulátor založený na agentoch, tak každý agent pracuje na základe sady parametrov a rozhoduje sa nezávisle počas celého trvania simulácie.

2.1.1 Spôsoby komunikácie

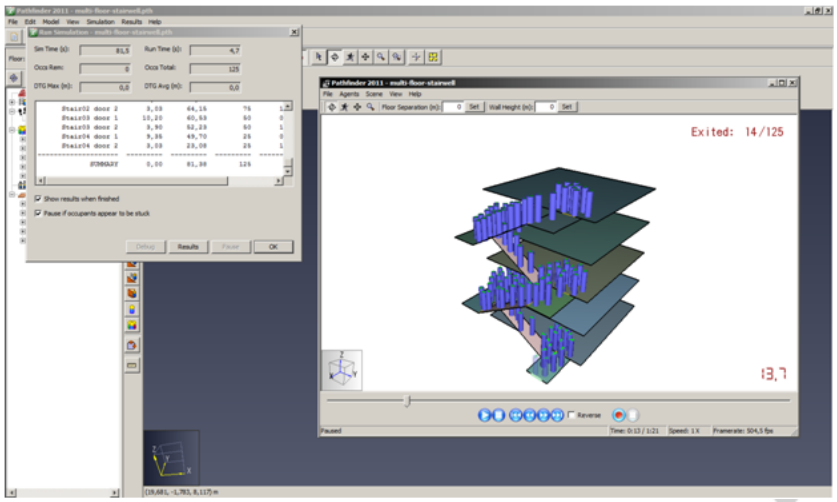
Komunikácia medzi simulačným prostredím a agentmi je možné rozdeliť na dve časti. Prvá sa týka toho ako prostredie vníma agentov a druhá ako agenty vnímajú prostredie. Simulačné prostredie sleduje pohyb agentov priebehom simulácie a poskytuje informácie ohľadom jednotlivých agentov (napr. pozícia). Naopak, agenty nepoznajú všetky východy z budovy a ich rozhodovanie je založené na kritériách vytvorenými

používateľom (napr. agenty môžu poznať hlavný východ, ale iba niektorí budú poznať sekundárne východy), informáciami z poschodia, skúsenosťou a v niektorých prípadoch aj informáciami ohľadom agentov, ktorí sú okolo neho. Pohyb agentov je v značnej miere ovplyvnený hustotou ostatných agentov v prostredí. Pri výpočte nasledujúcich akcií sa berú do úvahy tri fakty: vzdialenosť medzi agentmi, prekážky v prostredí a ohraničenia stavby.

2.1.2 Simulačné prostredie

Dôležitou časťou je editor s prehľadným používateľským rozhraním poskytujúci vytvorenie vlastného scenáru v 3D vizualizácii (obr. 2.1). Štruktúra prostredia je riešená takým spôsobom, že jednotlivé poschodia sú rozdelené do 2D plôch, ktoré umožňujú pohyb agentom z jedného bodu do iného v prostredí budovy. Na pohyb používajú agenty sadu určitých pravidiel, ktorá počíta s viacerými parametrami, medzi ktorými je možné započítať vzdialenosť od nepriechodných buniek, aktuálny stav agenta a iné. Po spustení simulácie sa prepočíta celá simulácia a výsledkom je interaktívna prezentácia, ktorá zobrazuje v 3D vizualizácií priebeh simulácie.

Jeden z problémov PathFinderu je jeho obmedzená časová použiteľnosť vo voľne dostupnej verzii a zdĺhavejší spôsob konfigurácie simulácie.



Obr. 2.1: Ukážka simulácie v programe PathFinder

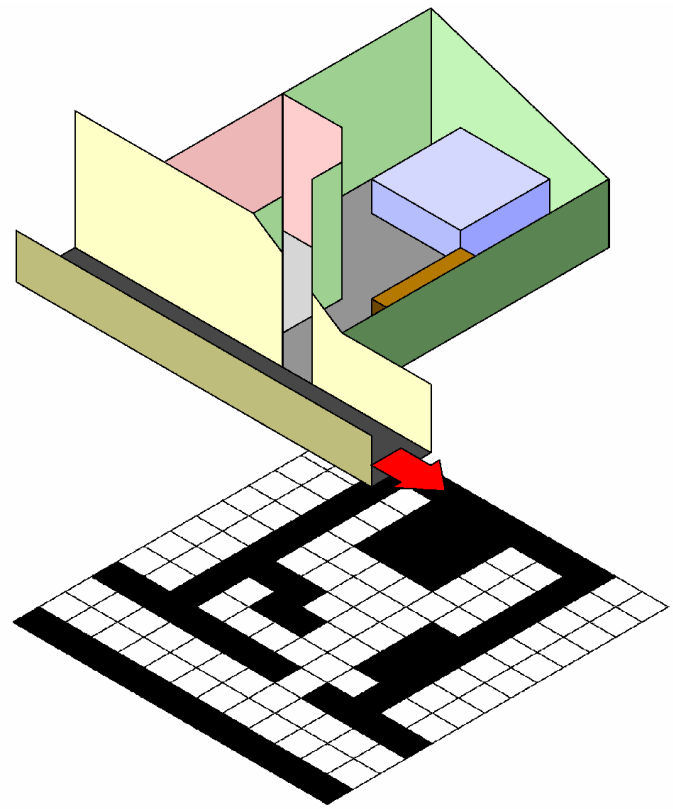
2.2 PedGo

PedGo predstavuje riešenie, ktoré svojimi malými nárokmi príliš nezaťažuje počítač. Využíva multi-agentový model a dokáže simulovať tisícky ľudí už na procesoroch

Pentium 3 o frekvencii 500 Mhz. To je umožnené aj minimalistickou vizuálnou stránkou - simulácia je zobrazená len v 2D priestore. Aplikácia je zameraná na simuláciu evakuácie pri rôznych katastrofických scenároch (napr. požiar).

2.2.1 Prostredie

Prostredie je vytvárané transformáciou plánu do dvojrozmernej mriežky buniek. Ukážka tvorby plánu/pôdorysu budovy je na obr. 2.2. Aplikácia rozoznáva 3 druhy buniek:



Obr. 2.2: Transformácia plánu budovy do 2D mriežky buniek

- voľná bunka – agent sa na ňu môže v nasledujúcej jednotke času presunúť
- stena – agent sa nemôže na túto bunku presunúť (predstavuje akúkoľvek prekážku pre pohyb)
- východ – táto bunka predstavuje napr. únikový východ, každý agent sa snaží dosta práve na túto bunku
- dvere – tieto bunky zmenšujú priepustnosť toku agentov, znižujú aj ich rýchlosť

- schody – taktiež znižujú rýchlosť agentov

Model človeka – tzv. agent sa pohybuje po týchto bunkách. Na pohyb môže využiť akúkoľvek bunku okrem tej, označenej ako stena. Na jednej bunke môže byť v jednom čase iba jeden agent.

2.2.2 Plánovanie cesty na základe buniek

Každá bunka je ohodnotená podľa vzdialenosti od východu. Čím je bližšie, tým má väčší potenciál. Východom sa nemyslí len definitívny východ znamenajúci úspešnú evakuáciu, ale aj všetky bunky označené ako dvere a schody. Potenciál sa takto šíri po bunkách, až kým nie sú všetky ohodnotené. Agent pri rozhodovaní pozerá na potenciál okolitých buniek a podľa toho si volí cestu.

Každý agent má na začiatku simulácie náhodne pridelené parametre:

- rýchlosť – udáva, koľko buniek môže prejsť agent za jednotku času
- trpezlivosť – ako dlho vydrží agent stáť na jednom mieste (napr. kvôli prekážkam) predtým, ako si zvolí inú cestu
- nevypočítateľnosť – aká je pravdepodobnosť, že sa nebude držat svojej cesty
- náhodnosť – aká je pravdepodobnnosť, že sa bude držať svojho smeru(daného potenciálom) a nezvolí si iný
- reakčný čas – ako dlho trvá agentovi reagovať na evakuačný signál
- spomalenie – pravdepodobnosť, že sa agent zastaví počas jednotky času a zostane stáť až do konca tejto jednotky
- prilnavosť – vyjadruje, ako veľmi sa agent drží v nejakej (danej) skupine agentov

Aplikácia ponúka dve predvolené nastavenia, ktore dané parametre prisposobujú podľa zvoleného času dňa – noc/deň.

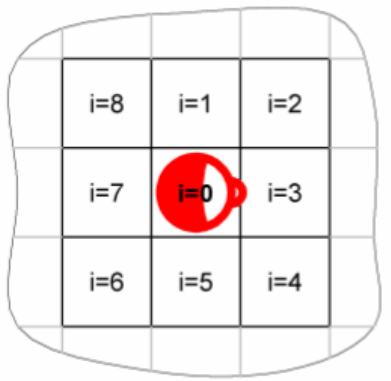
2.2.3 Rozhodovanie agenta

Na obr. 2.3 je znazornený agent (v strede) a okolité bunky. Agent si môže vybrať pohyb do 8 rôznych smerov v nasledujúcom časovom kroku. Pravdepodobnosť, že si vyberie bunku i sa počíta nasledujúcim vzorcom:

$$p_i = e^{-\frac{(P_i - P_0) + s}{S}},$$

pričom p_i vyjadruje pravdepodobnosť vybratia i-tej bunky, P_i potenciál bunky i, P_0 potenciál bunky 0 a S parameter označujúci nevypočítateľnosť.

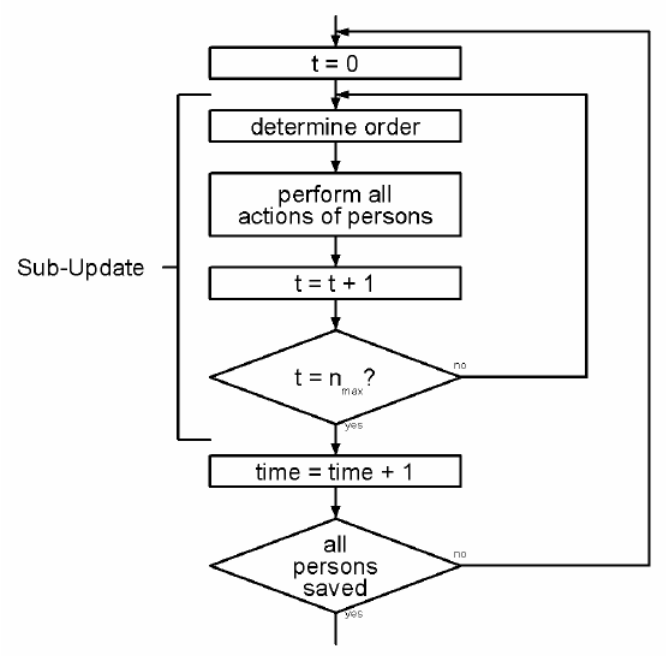
Po vypočítaní pravdepodobnosti výberu smeru je táto hodnota vynásobená hodnotou náhodnosti.



Obr. 2.3: Znázornenie agenta a možnosti jeho voľby

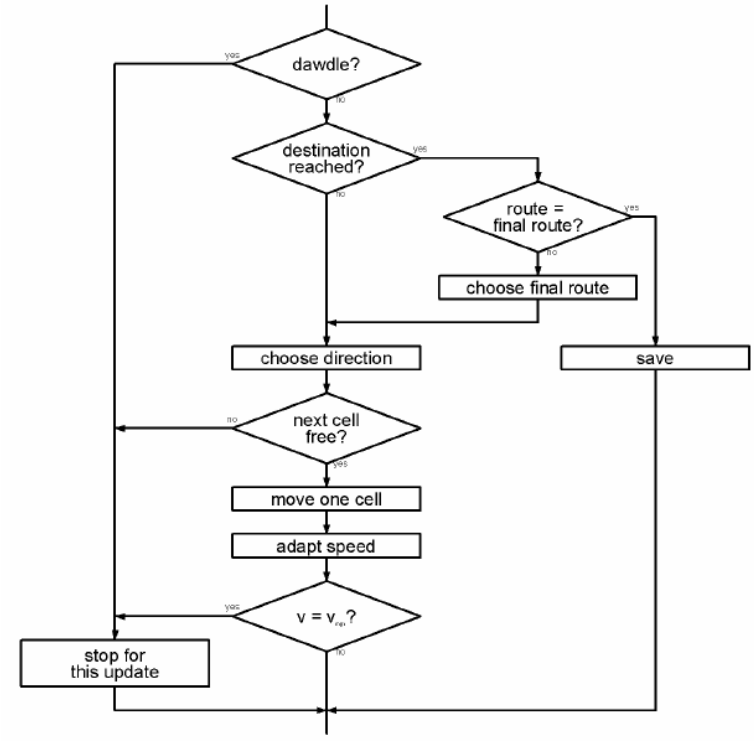
2.2.4 Pohyb a aktualizácia

Na obrázku 2.4 je znázornený diagram aktivít pre aktualizáciu celej populácie agentov, kde t je počet mini-aktualizácií, n rýchlosť celej populácie (koľko buniek sa môžu agenti posunúť za jednotku času) a *time* časový skok.



Obr. 2.4: Diagram aktivít pre aktualizáciu celej populácie agentov

Algoritmus riadenia pohybu jediného agenta je zobrazený na obrázku 2.5.



Obr. 2.5: Algoritmus pohybovania agenta

2.3 Fire Dynamics Simulator and Smokeview

Fire Dynamics Simulator je CFD (Computational fluid dynamics) model požiaru. Softvér numericky rieši tvar Navier-Stokesových rovníc, ktoré sú vhodné pre pomalý, tepelne riedený tok s dôrazom na transport dymu a tepla z ohňa.

Smokeview (SMV) je vizualizačný program, ktorý sa používa na zobrazenie výstupu z FDS a CFAST simulácií. FDS je voľný softvér vyvinutý NIST-om (National Institute of Standards and Technology of the United States Department of Commerce) v spolupráci s VTT výskumným technickým centrom vo Fínsku. Fire Dynamics Simulator spoločne s Smokeview boli oficiálne predstavený v Februári 2000. FDS číta vstupné dáta z textového súboru, vypočíta numerické riešenie riadiacich rovníc a výsledné dáta zapíše do súborov. Smokeview na základe týchto dát vytvára animáciu na obrazovke.

2.3.1 Rozšírenia FDS

FDS + Evac – je evakuačný simulačný model pre FDS. Softvér je využívaný na simuláciu pohybu ľudí pri evakuačných situáciach. Hlavné črty sú:

- Simulácia ľudí založená na agentoch
- Pohybový algoritmus založený na pohybe Panic
- Post-processing pomocou softvéru SMV
- Účinky ohňa sú prepočítavané pomocou FED (Fractional Effective Dose)

2.3.2 HPC (Hight Performance Computing)

FDS je napísaný v programovacom jazyku Fortran 90/95, ale malá časť kódu je napísaná v jazyku C. Momentálne je jedným z problémov aj preskúmanie možností na prepis tejto malej časti kódu do Fortranu. V súčastnej dobe existujú dve verzie FDS: sériová a paralelná. Sériová verzia využíva len jeden proces na výpočet riešenia riadiacich rovníc. Paralelná verzia využíva MPI (Message Passing Interface) na spustenie viacerých procesov na rôznych strojoch. Od verzie 5.4 Christian Rogsch z univerzity vo Wuppental v Nemecku implementoval OpenMP direktívy do FDS. OpenMP umožňuje aby FDS bežalo na jednom PC, ale využívalo viacero procesorov a jadier procesora. Táto verzia sa oficiálne stále iba testuje, ale OpenMP by mal ostať štandardom pre FDS.

2.3.3 Nástroje vyvinuté tretími stranami

Pre FDS bolo vyvinutých niekoľko nástrojov pre zjednodušenie práce. Tieto nástroje môžeme rozdeliť do kategórií:

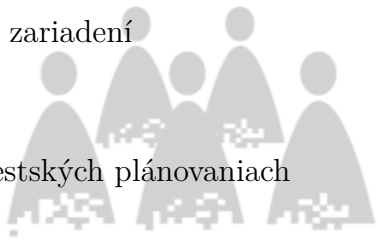
- GUI pre FDS – napr. Pyrosim, AspireSDS, BlenderFDS
- Nástroje pre konverziu CAD súborov – napr. 3dsolid2fds, acad2fds,
- Nástroje pre textový editor a tabuľkový procesor – napr. FDS v5 MESH Size Calculator, FDS 5 Syntax file
- Nástroje videa a obrazu – VideoEncoderGUI
- Post-processing a reportovacie nástroje - FDS Reporter
- Nástroje na modelovanie evakuácie - STEPS

2.4 SimWalk

Simwalk je pravdepodobne najviac komerčne používaným nástrojom v rámci simulácie davu za účelom zvýšenie efektivity v rámci logistiky. Logistiku môžeme charakterizovať ako proces plánovania, simulovania, implementovania a ovládania účinného, bezpečného a efektívneho toku chodcov. Riešenie okrem spomínanej simulácie davu ponúka aj simulovanie lietadiel, vozidiel mestskej hromadnej dopravy a vlakov idúcich z bodu A do bodu B pre účel účinných a bezpečných operácii. Simulovaním veľkého počtu ľudí a analýzami SimWalk pomáha vyriešiť problémy s neúčinnou pešou logistikou a prepravou. Nástroj podporuje všetky fázy projektovania logistiky a tiež umožňuje realistickú štúdiu a vylepšenie toku pasažierov. SimWalk PRO verzia rieši okrem iného aj evakuácie, štandardné letiskové operácie a problémy s prepravou handicapovaných ľudí.

Typické použitie SimWalk PRO

- Kapacita chodcov a účinnostná analýza komplexných zariadení
- Analýza kríženia sa chodcov v dopravných scenároch
- Analýza toku davu vo verejných priestranstvách a mestských plánovaniach
- Štúdia priechodnosti v architektúre
- Kapacita a prechodová analýza na letiskách
- Simulácia evakúacie



Výstupy aplikácie

- Kapacita
- Rýchlosť prechodov
- Počty a toky
- Úrovne služieb
- Strata rýchlosti

Toto riešenie sa javí ako veľmi užitočný softvér, ktorý má široké použitie. Jeho širokospektrálne zameranie predstavuje na druhú stranu aj problém v podobe ťažkej orientácie medzi ponúkanou funkcionalitou. Pre nového používateľa to môže znamenať v úvodnej fáze práce so softvérom početné komplikácie a problémy. Kedže je Simwalk komerčný nástroj, nebolo možné nájsť implementačné detaily o modularite aplikácie, preto nie je možné pridávať napríklad vlastných agentov v podobe knižnice alebo prostredníctvom interfejsu na to určeného. Azda najväčšou výhodou je umožnenie 2D aj 3D simulácie na jednom počítači bez využitia distribuovaného počítania.

Kapitola 3

Šprint#1

ID	Názov úlohy	Zodpovedný	Kapitola
1	Základná vizualizácia	Martin Košický	3.1
2	Komunikácia prostredia a agentov	Daniel Petráš	3.2
3	Základná funkcionalita agenta	Michal Fornádeľ	3.3
4	Vypracovanie metodiky Code Guidelines	Adam Pomothy	3.4
5	Zapracovanie analyzovaných riešení do šablóny dokumentu riadenia	Michal Fornádeľ	3.5
6	Integrovanie SVNka s vývojovým prostredím	Adam Pomothy	3.6
7	Zistenie formátu mapy v projekte VirtFIIT	Lukáš Pavlech	3.7
8	Nájdenie nástroja pre konverziu OSM -> CAD	Marek Hlaváč	3.8
9	Základná implementácia mapy	Marek Hlaváč	3.9
10	Analýza formátov máp	Marek Hlaváč	3.10
11	Základná funkcionalita prostredia	Lukáš Pavlech	3.11
12	Protokol agentových akcií a akcií prostredia	Daniel Petráš	3.12

3.1 Základná vizualizácia

Analýza problému
Existuje niekoľko spôsobov vykresľovania. Aplikácia je primárne tvorená pre Windows, takže možnosti sú vykresľovanie cez Direct3D, OpenGL , Windows GDI.

Direct3D je súčasťou DirectX SDK a má silnú podporu pre tvorbu grafických aplikácii.Direct 3D má funkcionalitu ekvivalentnú s OpenGL. Direct 3D ale nie je portabilné a funguje iba na Windows systémoch. Súčasťou DirectX SDK sú aj knižnice pre prácu so zvukom, práca so vstupom klávesnice, myši a iných zariadení.

OpenGL je knižnica, ktorá tiež ako Direct 3D umožňuje pracovať s grafickou kartou na nízkej úrovni a tak využívať 3D akceleráciu. OpenGL je portabilná knižnica, ktorá beží na viacerých operačných systémoch, dokonca aj na iných zariadeniach. Je to knižnica výhradne na grafiku a vstup zklávesnice a iných zariadení treba riešiť osobitne. Jednou z alternatív je knižnica SDL vďaka ktorej sa dá tiež pohodlne vytvoriť okno pre vykresľovanie 3D. Tiež je možnosť použiť Windows funkcie na zachytávanie aktuálne stlačených kláves a detekciu práce s myšou. Bez použitia SDL by musela byť tvorba okna manuálna.

Windows GDI umožňuje kresliť základné geometrické útvary ako čiary, body, obdĺžniky. GDI nemá žiadnu 3D akceleráciu a na vykresľovanie veľkého počtu agentov v reálnom čase nie je vhodné.

Návrh riešenia
Vizualizácia bude používať SDL a OpenGL ako základ na nízkej úrovni. Poskytujú vysoký výkon a pohodlne sa používajú. Vizualizácia bude fungovať v troch fázach: incializácia, cyklus a deinicializácia, ktorá súčasne ukončí celú aplikáciu.

Opis implementácie
Vizualizácia používa knižnice OpenGL pre vykresľovanie a SDL pre vytváranie okna.

Vizualizácia prebieha v troch fázach: inicializácia, cyklus pre každú snímku a deinicializácia.

Pri inicializácii sa vytvorí okno o rozmeroch 640x480 pixeloch s opengl vykresľovacím kontextom. Nastaví sa perspektíva, základný tieňovací model (ktorý sa momentálne nepoužíva), nastaví sa farba pozadia, porovnávacia funkcia na hĺbku a alokuje sa pamäť na grafickej karte, do ktorej sa načíta mapa.

Pre každú snímku sa vykonáva vyberanie všetkých udalostí cez SDL, medzi ktoré patrí aj stláčenie kláves. Pokiaľ je jedna z udalostí stlačenie klávesy ESC tak sa prejde k deinicializácii, inak sa vymaže hĺbkový buffer, opäť sa vykreslí mapa cez VBO a v cykle sa prechádzajú všetci agenti, ktorí sa vykresľujú na určenú pozíciu. Agenti vstupujú do vizualizácie cez komunikačný kanál z prostredia.

Deinicializácia spôsobí dealokáciu vykresľovacieho kontextu a vyhodí výnimku, ktorá signalizuje ukončenie aplikácie.

3.2 Komunikácia prostredia a agentov

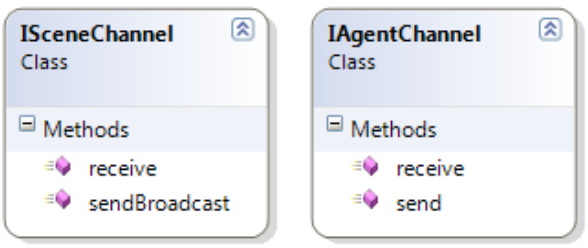
Analýza problému

Bolo treba navrhnúť spôsob, akým budú spolu komunikovať prostredie a agenty bez toho, aby boli naviazaní na konkrétnu implementáciu komunikácie. Spôsob komunikácie sa bude počas vývoja projektu meniť (napr. distribuované počítanie a pod.), takže bolo treba navrhnúť všeobecný model, s ktorého rozhraním budú agenty a prostredie narábať. Ďalšou úlohou bolo implementovať komunikačný kanál na základe navrhnutého modelu. Prvotná implementácia mala byť čo najjednoduchšia, kde agenty existovali v rovnakom procese ako samotné prostredie, takže sa jednalo iba o výmenu informácií medzi objektmi v rámci jedného programu.

Návrh riešenia

Riešenie spočíva v zadefinovaní dvoch rozhraní. Jedno rozhranie slúži na odosielanie správ všetkým agentom a získavanie správ od agentov (ISceneChannel). Druhé rozhranie na získavanie správ od prostredia a zasielanie správ prostrediu (IAgentChannel). To akým spôsobom sa správy dostanú z jednej strany na druhú už bude vecou implementácie. Metódy ktoré sa používajú na zasielanie správ sú asynchrónne, tj. neblokujúce, zatiaľ čo metódy na získanie zaslaných informácií, sú blokujúce až kým nepríde odpoveď. Obe rozhrania sú znázornené na obrázku 3.1.

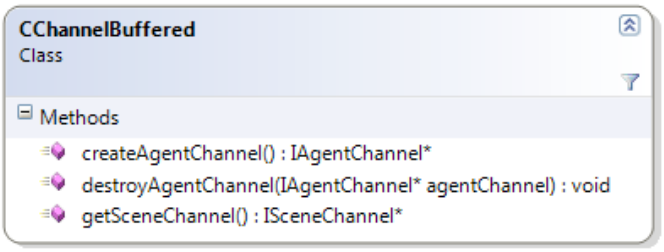
Opis implementácie



Obr. 3.1: Rozhranie komunikácie prostredia a agentov

Prostredie zapisuje správy ktoré chce poslať do vyrovnávacej pamäte. Ak sa pamäť minie začne prepisovať najstaršie správy. Vyrovnávacia pamäť je implementovaná pomocou poľa správ. Každý IAgentChannel si pamätá z ktorej pozície (poľa) naposledy čítal, a po precitaní novej správy sa o jedna posunie. Ak narazí na koniec, skočí na začiatok. Agenty zapisujú svoje správy na koniec fronty a prostredie ich načítava zo začiatku fronty. Z opisu je zrejmé, že obe implementácie rozhraní majú spoločné dátové štruktúry (pole a frontu). To je dosiahnuté pomocou továrne, ktorá na požiadanie vytvorí kanál pre agentov alebo prostredie. Keďže v simulácií vystupuje iba

jedno prostredie, aj kanál pre prostredie bude iba jeden. Tieto kanály zdieľajú spomínané dátové štruktúry práve od tejto továrne. Model továrne je zobrazený na obrázku 3.2.



Obr. 3.2: Model továrne

3.3 Základná funkcionalita agenta

Analýza problému
Za účelom overenia funkčnosti prostredia a komunikačného protokolu vznikla potreba dopracovania agenta. Agenty sú v počiatočnej fáze generované na ľubovoľnom mieste na mape a ich správanie nie je ovplyvnené žiadnymi faktormi.

Návrh riešenia
Prvotný návrh bolo vytvorenie primitívneho agenta, ktorého správanie bolo špecifikované generovaním náhodných krokov (v zmysle doľava, doprava, vpred, vzad) alebo uhla natočenia.

Implementácia
Implementácia agenta obsahovala jeho základný vnútorný stav - polohu, kde sa nachádza a natočenie. Každý vytvorený agent bol navrhnutý, aby neustále zásoboval prostredie požiadavky na pohyb. V priemere každá piata požiadavka bola v požiadavka pootočenia o uhol v rozmedzí -30 až 30 stupňov. Agenty sa pohybovali po mape nezávisle a nebrali do úvahy rozmery, ani umiestnenie na obrazovke.

3.4 Vypracovanie metodiky Code Guidelines

Pre dosiahnutie čo najväčšej kvality produktu z vývojárskeho hľadiska je veľmi dôležité stanoviť pravidlá, ktoré vývojári dodržiavajú pri implementácii. Výsledkom je zefektívnenie vývoja, väčšia produktivita a lepšia komunikácia medzi členmi vývojárskeho tímu. Pravidlá použité pri vývoji tejto aplikácie sa nachadzajú v dokumente

riadenia v prílohe A.

3.5 Zapracovanie analyzovaných riešení do šablóny dokumentu riadenia

Analyzované riešenia boli z dôvodu štruktúry jednotlivých dokumentov presunuté do dokumentu k inžinierskemu dielu. Tu sa doposiaľ vytvorené časti písané v prostredí Microsoft Word upravovali pre následnu integráciu do prostredia šablóny.

3.6 Integrovanie systému SVN s vývojovým prostredím

Na plné využitie výhod verziovacieho systému (v našom prípade Subversion) je potrebná aj jeho integrácia do vývojového prostredia. Už pri výbere systému sme brali do úvahy existenciu voľne dostupného doplnku do nami zvoleného vývojového prostredia (Microsfot Visual Studio 2010), ktorý by bol kompatibilný so systémom Subversion. Ako vhodný doplnok sme vybrali AnkhSVN, ktorý podporuje všetky pokročilé funkcie manažovania spoločného prístupu do zdrojových súborov. Doplnok umožňuje aktualizovať, uploadovať ale aj spájať (angl. merge) zdrojový kód priamo z vývojového prostredia. Vďaka tomu sa výrazne zefektívni implemetácia a predíde sa množstvu problémom pri zasahovaní do rovnakých súborov so zdrojovými kódmi.

3.7 Zistenie formátu mapy v projekte VirtualFIIT

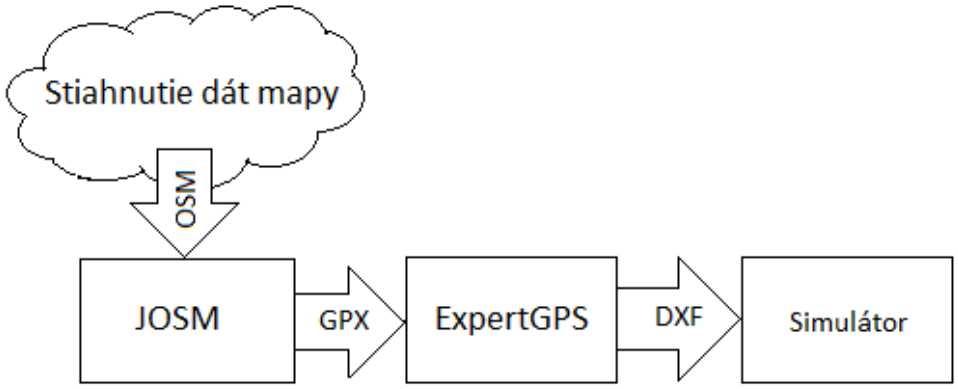
Analýza problému
Projekt Virtuálna FIIT bol riešený v ročníku 2010/2011 tímom Mgr. Aleny Kovárovej. V rámci tohto projektu ako výsledok bola vytvorená 3d web prezentácia školy. Model školy je vytvorený v nástroji 3ds Max. Tento formát môže byť výhodný pre náš tím, lebo sa dá konvertovať do rôznych formátov ako napr. AutoCad, ArchiCad atď.

3.8 Nájdenie nástroja pre konverziu OSM -> CAD

Analýza problému
Pre prácu s formátom OSM sme použili softvér JOSM (JavaOpenStreetMap Editor), ktorý umožňuje stiahnuť reálne údaje a následne s nimi pracovať. Ďalší krok spočíval v konverzii do DXF formátu na základe výberu formátu máp, ktorý je uvedený v Tab.1. v kapitole Analýza formátov máp. Keďže priamy export z JOSM do formátu

DXF nie je možný, tak sme boli nútený spraviť prieskum cielený na možnosti konverzie OSM formátu do formátu DXF.

Výsledok prieskumu je, že priama konverzia nie je podporovaná žiadnym voľne šíriteľným nástrojom. Ďalšou možnosťou je nájdenie tretieho formátu, ktorý by slúžil ako sprostredkovateľ konverzie medzi týmito formátmi, čo by znamenalo, že by bola podporovaná konverzia z OSM do tretieho formátu a následne ďalšia konverzia do formátu DXF. Ako sprostredkovateľ konverzie bol vybraný softvér ExpertGPS, ktorý umožňuje export do DXF formátu a import GPX súborov, ktoré je možné vyexportovať prostredníctvom JOSM (obr. 3.3).



Obr. 3.3: Konverzia formátu OSM do DXF

3.9 Základná implementácia mapy

Analýza riešenia
Vzhľadom na základnú funkcionalitu simulácie je nutné do mapy implementovať prvky, ktoré budú poskytovať potrebné informácie. Každý typ prvkov je v mape umiestnený na vlastnej vrstve. Dôvodom oddelenia je ľahšie spracovanie údajov pri importe mapy do vytváranej aplikácie. Základné mapy obsahujú 5 vrstiev:

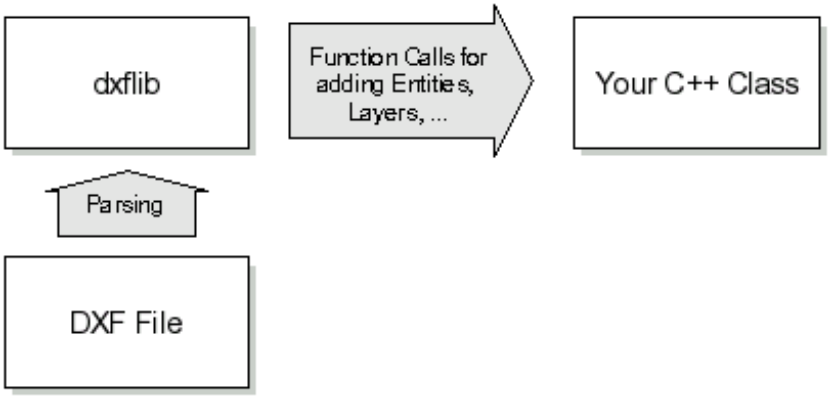
- Steny – reprezentujú ohraničenie budovy, v ktorej sa má simulácia odohrávať.
- Prekážky – reprezentujú objekty v budove, cez ktoré nie je možné prejsť.
- Štarty (angl. spawns) – sú plochy, v ktorých sa generujú agenty na začiatku simulácie.
- Dvere – reprezentujú oblasti medzi miestnosťami, ktoré uľahčujú navigáciu agentov.

- Východy – sú špeciálny druh dverí, ktoré po dosiahnutí agentom ukončujú jeho simuláciu.

Všetky prvky sú vo zvolenom DXF formáte reprezentované ako krivky (angl. polylines), ktoré sú reprezentované postupnosťou bodov, pričom prvý a posledný bod je rovnaký. Takýmto spôsobom sme schopní navrhnúť vlastné jednoduché mapy, po prípade použiť koverziu medzi rôznymi formátmi.

Návrh riešenia

Knižnica dxflib je open source C++ knižnica zameraná na čítanie a následné spracovanie informácií z textových súborov (angl. parsing) formátu DXF. Knižnica pri spracovávaní informácií volá funkcie, ktoré sú zadefinované v používateľskej triede. Tieto funkcie sa volajú na základe entít, ktoré sú aktuálne spracovávané, to znamená, že pre každý typ entity môžeme vytvoriť unikátne spracovanie. Ďalšou dôležitou vlastnosťou je jednoduché spracovávanie údajov prostredníctvom vrstiev. Knižnica dxflib taktiež umožňuje zápis údajov, ale pri používaní sa predpokladá hlbšia znalosť formátu DXF. Štruktúra knižnice dxflib je zachytená na obrázku 3.4.



Obr. 3.4: Štruktúra knižnice dxflib

Implementácia

Pre prvé jednoduché simulácie sme vytvorili nasledujúce mapy, ktorých cieľom je overenie základnej funkcionality simulácie vzhľadom na jednotlivé oblasti:

- spracovávanie informácií aktuálneho stavu simulácie prostredím
- plánovanie agentov
- navigácia agentov
- vizualizácia agentov v prostredí

- výpočet vnútornej mapy agentov
- generovanie agentov
- konfliktné situácie agenta s agentom alebo agenta s prekážkami, či stenami
- odchod agentov zo simulácie

3.10 Analýza formátov máp

OpenStreetMaps
OpenStreetMaps (OSM) je celosvetový projekt, ktorého cieľom je vytváranie voľne dostupných zemepisných informácií a ich následná transformácia do máp so zámerom využitia pre topografickú vizualizáciu. OSM dáta sa vytvárajú na základe záznamov z GPS (Global Positioning System) prijímačov alebo iných digitálnych zariadení, ktoré sú licenčne kompatibilné. OSM využíva vlastný formát dát, ktorý je postavený na značkovacom jazyku XML. Primitívnymi konštrukciami formátu OSM sú:

- Uzly – body, ktoré musia obsahovať unikátny identifikátor a pozíciu reprezentovanú prostredníctvom zemepisnej šírky (angl. latitude) a dĺžky (angl.longitude).
  <node id=”15680” lat=”61.808395385742” lon=”10.849707603454”
  visible”true” timestamp=”2005-07-30T14:27:12+01:00”/>
- Cesty – postupnosti uzlov, ktoré znázorňuje krivku alebo uzavretú krivku – polygon. Musia obsahovať zoznam referencií na konkrétne uzly a označenie prostredníctvom špeciálneho elementu <tag/>.
  <way id=”35” visible=”true” timestamp=”2006-03-14T10:07:23+00:00” user=”johnz”>
  <nd ref=”156804”/>
  <nd ref=”156805”/>
  <nd ref=”156806”/>
  <tag k=”highway” v=”secondary”/>
  </way>
- Relácie – skupiny uzlov, ciest a ďalších relácií, ktorým zámerom je opísať tieto skupiny.
  <relation id="77"visible="true"timestamp="2006-03-14T10:07:23+00:00üser="fred»
  <member type="way"ref="343"role="from"/>
  <member type="node"ref="911"role=ät"/>
  <member type="way"ref="227"role="to"/>
  <tag k="type"v="turn_restriction"/>
  </relation>

Na základe týchto konštrukcií je možné vytvoriť komplexné topografické mapy. Tieto mapy je možné použiť pri najrôznejších typoch vizualizácií a simulácií, pričom je možné mapy jednoduchým spôsobom rozšíriť o špecifické vlastnosti. Problémom pri OSM je, že satelitné informácie, ktoré sú vstupom pre konverziu do OSM je možné zachytiť len vonkajší pohľad, a teda sme značne obmedzení pri vytváraní simulácie vo vnútornom prostredí štruktúr a taktiež použitím 3D máp.

Drawing Exchange Format
Drawing Exchange Format (DXF) je CAD (Computer Aided Design) formát vyvinutý spoločnosťou Autodesk, ktorý umožňuje výmenu dát medzi softvérom AutoCAD a inými druhmi softvérov.

DXF formát je poskladaný z dvojíc, kde kódu prislúcha konkrétna hodnota. Skupiny kódov (angl. group codes) indikujú typ hodnoty, ktorá za nimi nasleduje. DXF súbor je organizovaný na základe skupín kódov s príslušnými hodnotami do sekcií, ktoré sú poskladané zo záznamov, pričom jednotlivé záznamy sa skladajú zo skupín kódov a dátových položiek. Každá skupina kódu a hodnota je na jednom riadku v DXF súbore. DXF súbor obsahuje viacero textových sekcií:

- Hlavička – obsahuje nastavenia premenných hodnôt, ktoré súvisia s výkresom.
- Triedy – uchovávajú informácie pre aplikačné definície tried, ktorých inštancie sa vyskytujú v sekciách bloky, entity a objekty.
- Tabuľky – obsahujú definície mien základných prvkov zobrazenia.
- Bloky – obsahujú definície prvkov obsiahnutých v blokoch výkresu.
- Entity – obsahujú všetky prvky súboru vrátane umiestenia blokov.
- Objekty – obsahujú informácie aplikované na negrafické objekty.
- Náhľad výkresu
- Koniec súboru

Prostredníctvom DXF formátu sa dá znázorniť akákoľvek mapa, či už 2D alebo 3D. Toto zobrazenie je realizované využitím základných geometrických primitív (body, úsečky, krivky, kružnice, atď.), ktoré je možné skladať do väčších celkov v jednotlivých vrstvách výkresu. Takýmto spôsobom je možné vytvoriť komplexné a ľahko rozšíriteľné mapy, či už vonkajších alebo vnútorných prostredí. DXF je výmenný formát, takže je možné vykonávať konverziu mnohých formátov z alebo do DXF formátu.

Výber formátu
Pri výbere formátu sme sa riadili 4 aspektmi pohľadu:

- Použiteľnosť – reprezentuje hodnotu použitia formátu vo vonkajších a vnútorných prostrediach, podporu knižníc a ďalších nástrojov pri získavaní informácií.
- Konverzia – reprezentuje podporu konverzie medzi formátmi, čím je možné v budúcnosti používať aj mapy v iných formátoch.
- 3D – podpora reprezentácie mapy v 3-dimenziálnom zobrazení.
- Čitateľnosť – charakterizuje úroveň pochopenia čítaného textového formátu dát človekom.
- Vrstvy – podpora vrstevnej reprezentácie mapy.

Formát	Použiteľnosť	Konverzia	3D	Čitateľnosť	Vrstvy
OSM	++	+	- - -	+++	+++
DXF	+++	+++	+++	-	+++

3.11 Základná funkcionalita prostredia

Návrh riešenia
V prvotnom prototype prostredie vykonáva nasledujúce činnosti:

- pri spustení programu musí umiestniť agentov
- príjma od jednotlivých agentov správy, ktoré obsahujú informácie o zmene natočenie, polohy
- prepočítava nové pozície, natočenia pre agenta
- posiela agentom informácie o ich súčastnom umiestnení

Opis implementácie
Pri štarte aplikácie sa vytvárajú agenti. V rámci vytvárania agenta sa agentovi generuje náhodná pozícia vo vnútri budovy (prostredie už predtým získa údaje o mape) a natočenie agenta. Potom sa spustí metóda void CSceneImpl::simulate() v ktorej sa nachádza nekonečný cyklus while. V tomto cykle sa pri každom prechode pošlú inforácie všetkým agentom. Každý agent vykoná svoju úlohu a pošle naspäť prostrediu výslednú akciu. Prostredie dekóduju túto správu podľa Protocol.h a vykoná náležité zmeny.

3.12 Protokol agentových akcií a akcií prostredia

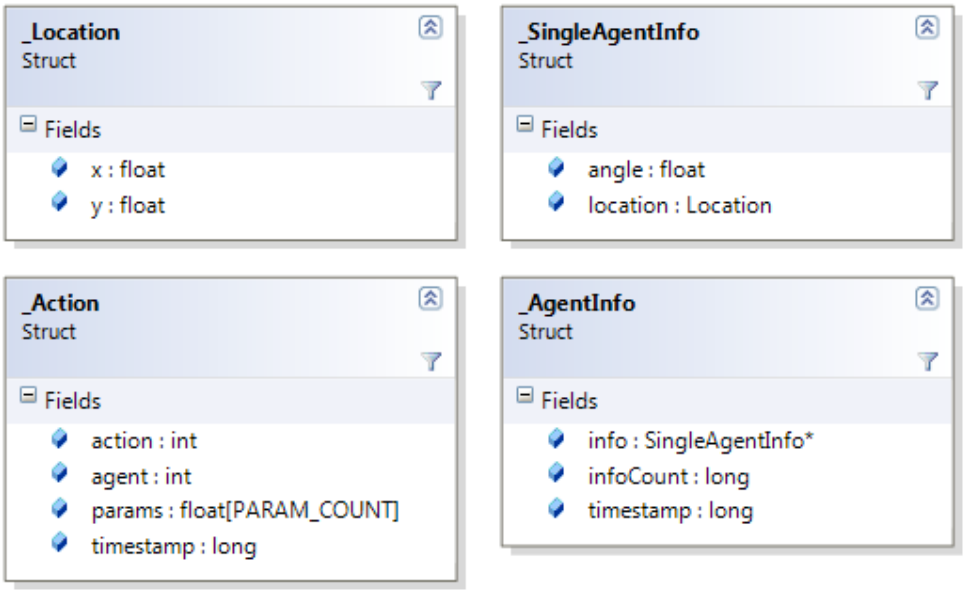
Analýza problému
Základným problémom bolo navrhnúť protokol, pomocou ktorého si budú môcť prostredie a agenty vymieňať informácie. Agent potrebuje vedieť, kde v prostredí sa nachádza a kde sa nachádzajú ostatné agenty. Prostredie zase potrebuje vedieť, akú akciu sa snaží agent vykonať. Základné informácie o agentovi sú jeho poloha a natočenie (uhol). Potrebné sú aj doplnkové informácie, ako napríklad kedy bola daná akcia vygenerovaná a kým.

Návrh riešenia
Informácia o pozícii všetkých agentov sa dá jednoducho zaznamenať pomocou poľa. Prvkami poľa budú dátové štruktúry obsahujúce práve informácie o polohe a natočení agenta. K týmto informáciám je pridaný aktuálny čas kedy bola informácia zaslaná agentom.

Akcia, ktorú posiela agent prostrediu, bude definovaná svojim identifikátorom. Podľa toho, o akú akciu pôjde, môže ďalej obsahovať niekoľko parametrov. Takisto bude obsahovať čas kedy bola akcia zaslaná prostrediu ako aj identifikátor agenta, ktorý akciu vygeneroval.

Opis implementácie
Protokol je implementovaný pomocou niekoľkých dátových štruktúr, ktoré sú zobrazené na obrázku 3.5.
Maximálny počet parametrov v štruktúre *Action* je 3. V prípade potreby je možné tento počet kedykoľvek zmeniť. Pomocou tohto prístupu je kedykoľvek možné pridať ďalšiu akciu bez toho, aby sa musel meniť komunikačný protokol. Zadefinovane akcie sú:

- ACTION_TURN (id = 1)
  - Akcia otočenia sa v smere hodinových ručičiek (v prípade kladného uhla).
  - Počet parametrov: 1
  - Prvý parameter: uhol (v stupňoch) zmeny aktuálneho natočenia z intervalu (-360.0; 360.0)
- ACTION_STEP (id = 2)
  - Akcia kroku vpred, samotný smer závisí od aktuálneho natočenia.
  - Počet parametrov: 0



Obr. 3.5: Štruktúra prenášaného protokolu

Kapitola 4

Šprint#2

ID	Názov úlohy	Zodpovedný	Kapitola
1	Simulovanie pohľadu agenta	Martin Košický, Michal Fornádeľ	4.1
2	Hľadanie dverí zo strany agenta	Adam Pomothy	4.2
3	Plánovanie pohybu agenta	Marek Hlaváč	4.3
4	Integrácia vyhotovenej dokumentácie do šablóny dokumentu riadenia	Michal Fornádeľ	4.4
5	Detekcia pretínajúcich sa stien	Daniel Petráš	4.6
6	Riešenie kolízií agentov	Lukáš Pavlech	4.7
7	Generovanie agentov do mapy	Lukáš Pavlech	4.8
8	Vytvorenie koncepcie oddelených modulov (DLL knižnice)	Martin Košický	

4.1 Simulovanie pohľadu agenta

Úloha v čase ukončenia šprintu nebola splnená.

4.2 Hľadanie dverí zo strany agenta

Analýza problému

Každý agent má k dispozícii svoju lokálnu mapu. Agent sa vďaka nej dokáže rozhodnúť, aký bude jeho nasledujúci krok. Ak v danom momente nepozná bezpečnostnú zónu, je preňho prvoradé nájsť všetky dostupné dvere. O dverách potrebuje vedieť, či sa nachádzajú v rovnakej miestnosti ako on a potrebuje poznať aj ich aktuálnu vzdialenosť.

Návrh riešenia
Proces hľadania dverí pozostáva z nasledujúcich krokov:

1. Získanie zoznamu všetkých dverí z aktuálnej mapy agenta.
2. Pre každé dvere zistiť, či sa medzi nimi a agentom nenachádza prekážka (stena alebo iná prekážka)
3. Vypočítanie vzdialenosti ku každým dverám.
4. Vytvorenie zoznamu dverí s ich vzdialenosťami od agenta a informáciou o prípadnej prekážke medzi nimi a agentom.

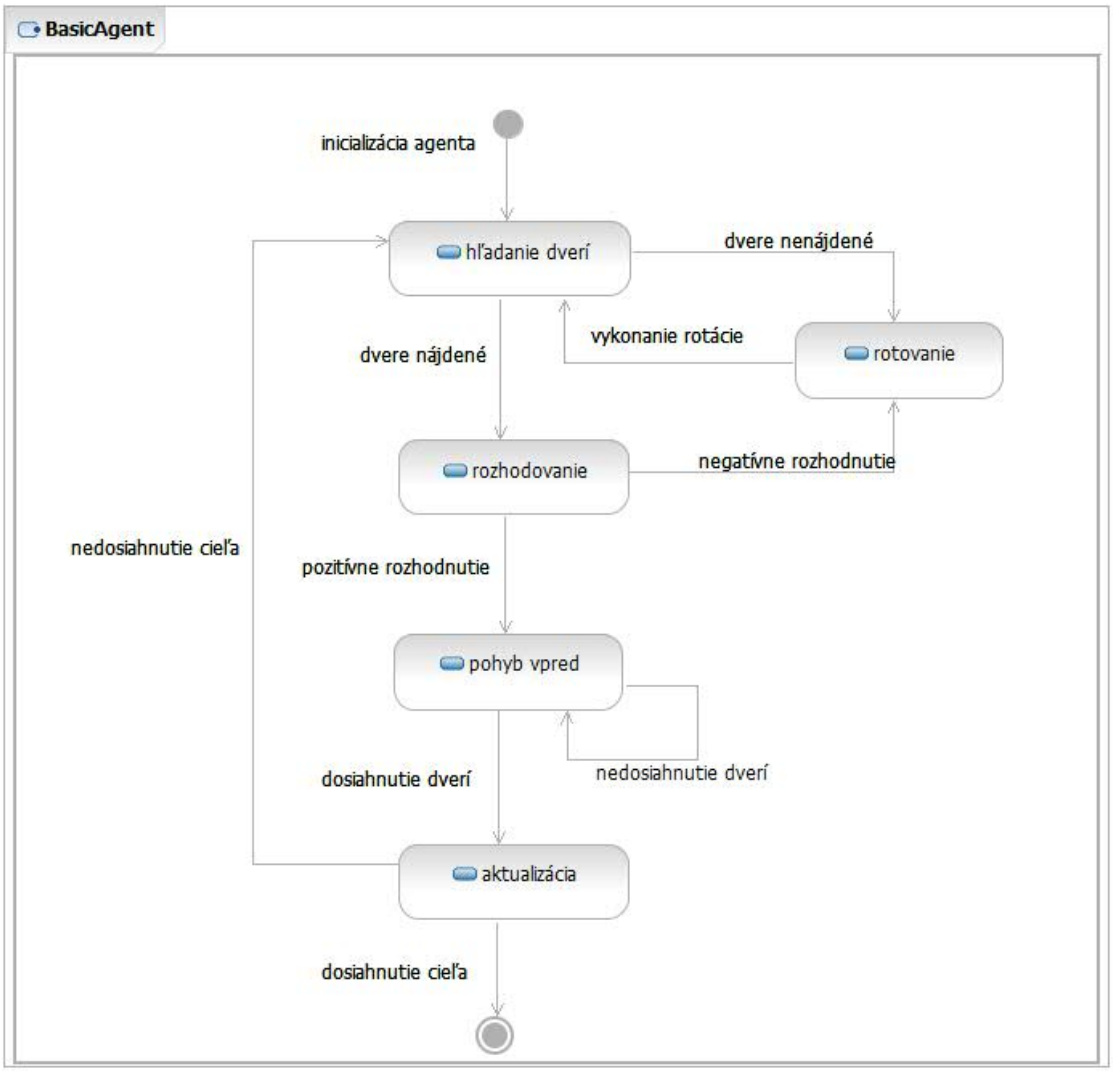
Opis implementácie
Pri vykonávaní procesu je potrebné najprv vypočítať súradnice stredu dverí, nakoľko sú reprezentované ako obdĺžnik. Potom sa vzdialenosť agenta od dverí počíta ako dĺžka spojnice aktuálnej polohy agenta a vypočítaného stredu obdĺžníka reprezentujúceho dvere. Pri počítaní vzdialeností je ďalej potrebné zistiť, či sa spojnica agenta a dverí nepretína so stenou alebo inou prekážkou - čo znamená, že sa dvere nachádzajú v inej miestnosti. Na to je potrebné prejsť všetky prekážky a steny (reprezentované ako vektory) a uistiť sa, že nemajú žiadny spoločný bod so spojnicou agent-stred dverí. Výstup tohoto procesu je zoznam dverí obsahujúci súradnice dverí, ich aktuálnu vzdialenosť k agentovi a informáciu o prípadných prekážkach medzi nimi a agentom.

4.3 Plánovanie pohybu agenta

Návrh riešenia
Po inicializácii sa agent nachádza v stave hľadania dverí. Dôvod je ten, že mapa je pre neho neznáma a vzhľadom na tento fakt je nutné, aby sa porozhliadol po miestnosti so zámerom nájdenia dverí. Dvere slúžia agentovi ako navigácia po mape. V prípade, že agent nevidí dvere, tak rotuje svoju pozíciu, až kým nejaké nenájde. Ako náhle sa mu podarí nájsť dvere, tak sa dostáva do stavu rozhodovania, v ktorom sa snaží rozhodnúť, či pokračovať v hľadaní ďalších dverí alebo nie. Cieľom rozhodovania je zamedzenie pohybu agentovi cez rovnaké dvere a umožniť mu prechádzať cez ešte nenavštívené dvere efektívnym spôsobom. Heuristika rozhodovania môže byť rozšírená o znalosť mapy, ktorá umožňuje simulovať pokročilejšie správanie agentov.

V prípade, že agent je rozhodnutý pokračovať k vybraným dverám, tak sa aktivuje pohyb smerom vpred vzhľadom na natočenie agenta. V každej iterácií pohybu sa kontroluje dosiahnutie dverí. Po úspešnom dosiahnutí sa aktualizuje stav agenta doplnením zoznamu novou položkou dverí, ktoré boli dosiahnuté. Následne sa kontroluje

dosiahnutie cieľa, resp. východu. Pri úspešnej kontrole sa simulácia agenta ukončuje, v opačnom prípade začína celý cyklus odznovu.
Stavový diagram rozhodovania agenta je znázornený na obrázku 4.1.



Obr. 4.1: Stavový diagram rozhodovania agenta

4.4 Integrácia vyhotovenej dokumentácie do šablóny dokumentu riadenia

Úloha spočívala v korekcii a prepise vyhotovených častí dokumentácií do prostredia šablóny dokumentácie riadenia. Niektoré časti boli z koncepčného hľadiska presunuté do dokumentácie ku inžinierskemu dielu.

4.5 Vytvorenie koncepcie oddelených modulov (DLL knižnice)

Analýza problému
Existuje niekoľko prístupov k riešeniu projektov na ktorom pracuje viacej ľudí. Môže to byť jedna veľká aplikácia, čo môže viesť k problémom keď veľa ľudí súčasne zasahuje do toho istého projektu. Iná možnosť je rozdeliť aplikáciu na samostatné aplikácie. V takom prípade je problém so zdielaním prostriedkov a pamäte, keďže všetky bežiace inštancie aplikácie majú vlastný pamäťový priestor. Tretí prístup je tvorba modulov, ktoré majú prístup do jednej pamäte. Tento prístup rieši problém so zdielaním informácií. Moduly sa navyše dajú jednoducho testovať.

Návrh riešenia
Navrhujeme vytvoriť 4 projekty, 3 moduly, jeden pre simuláciu agentov, jeden pre prostredie, jeden pre vizualizáciu a jeden tzv. "obalovač" (angl. wrapper) ktorý bude spúšťať moduly. Moduly budú komunikovať cez spoločný komunikačný kanál ktorý bude dostatočne abstraktný. V prípade snahy rozdeliť aplikáciu na samostatné aplikácie bude treba len zmeniť konfiguráciu projektu a zmeniť implementáciu kanála.

Opis implementácie
Projekt vizualizácie davu bol rozdelený na 4 samostatné projekty. 3 dynamické knižnice a jeden"obalovač" ktorý tieto knižnice spúšta. Komunikácia medzi knižnicami sa robí cez vytvorený komunikačný kanál. Jeden je na strane prostredia. Pre každý ďalší inicializovaný modul sa vytvorí kanál pre agenta a vizualizáciu osobitne. Keďže je predpoklad, že moduly ktoré budú simulovať agentov bude spúšťať prostredie tak prostredie dostáva pri konštrukcii na vstupe alokátor, ktorý spustí modul pre simuláciu agenta. Tiež dostane na vstupe alokátor, ktorý priradí kanál, cez ktorý bude prostredie komunikovať s modulom.

4.6 Detekcia pretínajúcich sa stien

Analýza problému
Pomocou uhla natočenia agenta a veľkosti kroku vieme určiť, aká bude nasledujúca poloha agenta. Spojením pôvodnej a novej pozície agenta dostaneme úsečku, po ktorej sa bude pohybovať. Táto úsečka sa nesmie pretínať so žiadnou stenou v mape prostredia. Vylepšenie tohto algoritmu spočíva namiesto použitia jednej pohybovej úsečky dvoch, kde každá z nich začína a končí na bokoch agenta. Tým sa vylúči, že by agent prešiel cez medzeru, do ktorej sa v podstate nezmestí.

Návrh riešenia
Prvoradé je vytvorenie nástroja, ktorý nám jednoducho povie, či sa dve úsečky pretínajú alebo nie. Pomocou tohto nástroja jednoducho implementujeme potrebnú funkcionalitu. K samotnému nastaveniu pozície dôjde pomocou LocationController-a. To je trieda zodpovedná za zmenu pozície agenta. Každý agent ma niekoľko definovaných niekoľko kontrolerov, pričom v opise implementácie bude pozornosť upriamená iba na jeden samostatný kontroler.

Opis implementácie
Detekcia pretínajúcich sa úsečiek bola realizovaná triedou CUtils, pomocou ktorej sa dá jednoducho zistiť, či sú úsečky rovnobežné, totožné, alebo či sa pretínajú. Ak sa pretínajú, je možné zistiť aj bod, v ktorom k tomu dochádza. Pre každého agenta sú následne určené jeho bočné body predstavujúce v abstraktnom ponímaní pozície ramien človeka. Tie závisia od pozície agenta v prostredí a jeho aktuálneho natočenia. Z týchto bodov boli skonštruované úsečky v smere natočenia o veľkosti jedného kroku. Každá z týchto úsečiek je následne porovnávaná so stenami v mape. Ak je zistený prienik, pohyb agenta skončí na tej stene, ktorá je k nemu najbližšie. Nakoniec treba vykonať úpravu pozície tak, aby sa agent dotýkal steny, ale zároveň dodržal naplánovaný smer. Tento odstup od konkrétneho priesečníka závisí od uhla agenta vzhľadom na stenu, na ktorú narazil. Pri kolmom náraze je táto úprava najväčšia, pri malom uhle je minimálna.

4.7 Riešenie kolízií agentov

Úloha nebola v čase ukončenia šprintu splnená.

4.8 Generovanie agentov do mapy

Návrh riešenia
V mape, ktorá sa inicializuje pri štarte aplikácie sa nachádzajú plochy určené na generovanie agentov. Tieto plochy (obdĺžniky) sú rozdelené na mriežku, kde jedna bunka má veľkosť 0,5 metra. Táto veľkosť reprezentuje maximálnu šírku agenta. Do kontajnera startPositions sú pridané všetky možné štartovacie pozície agentov. Následne sa v časti generovania štartovacej pozície generuje číslo v intervale od 0 po veľkosť kontajnera startPositions.

Opis implementácie
Pri štarte aplikácie sa cez parameter konštruktora inicializuje mapa pre prostredie. Pomocou tejto mapy sa vypočítajú všetky možné umiestnenia agentov (tak aby ne-

nastavali kolízie) a tieto pozície sa uložia do kontajnera startPositions. Pri vytváraní jednotlivých agentov sa skontroluje, či sa v kontajneri startPositions nachádzajú ešte voľné pozície - ak áno, tak sa vygeneruje náhodná pozícia v danom kontajneri. Z tejto pozície sa vyberú informácie o pozícií na mape a priradia sa novému agentovi. Daná pozícia sa následne odstráni z kontajnera startPositions.

Kapitola 5

Šprint#3

ID	Názov úlohy	Zodpovedný	Kapitola
1	Kontrola dosiahnutia výstupného stavu agenta	Adam Pomothy	5.1
2	Riešenie kolízií agentov	Lukáš Pavlech	5.2
3	Prenos generovania agentov do nového projektu	Lukáš Pavlech	5.3
4	Prenos základnej funkcionality agentov	Lukáš Pavlech	5.4
5	Zdokumentovanie DLL	Martin Košický	5.5
6	Prenos máp do nového projektu	Marek Hlaváč	5.6
7	Distribúcia mapy modulom	Martin Košický	5.7
8	Návrh plánovania rozhraní pohybu agentov	Martin Košický	5.8
9	Implementácia časovania prostredia	Daniel Petráš	5.9
10	Detekcia agenta prechádzajúceho cez stenu	Daniel Petráš	5.10
11	Monitorovanie a integrácia implementačných riešení	Michal Fornádeľ	
12	Analýza a návrh Flowfield	Adam Pomothy, Michal Fornádeľ	
11	Vykreslenie načítanej mapy	Martin Košický	

5.1 Kontrola dosiahnutia výstupného stavu agenta

Analýza problému

Nakoľko je našou úlohou simulovať evakuáciu ľudí z budovy, máme na každej mape zadefinovanú časť mapy - tzv. *safe-zone*, ktorá predstavuje bezpečnú zónu, napríklad východ z budovy. Keď agent dosiahne takúto zónu na mape, je v bezpečí a jeho simulovanie končí. V takom prípade ho treba odstrániť zo zoznamu zobrazovaných a

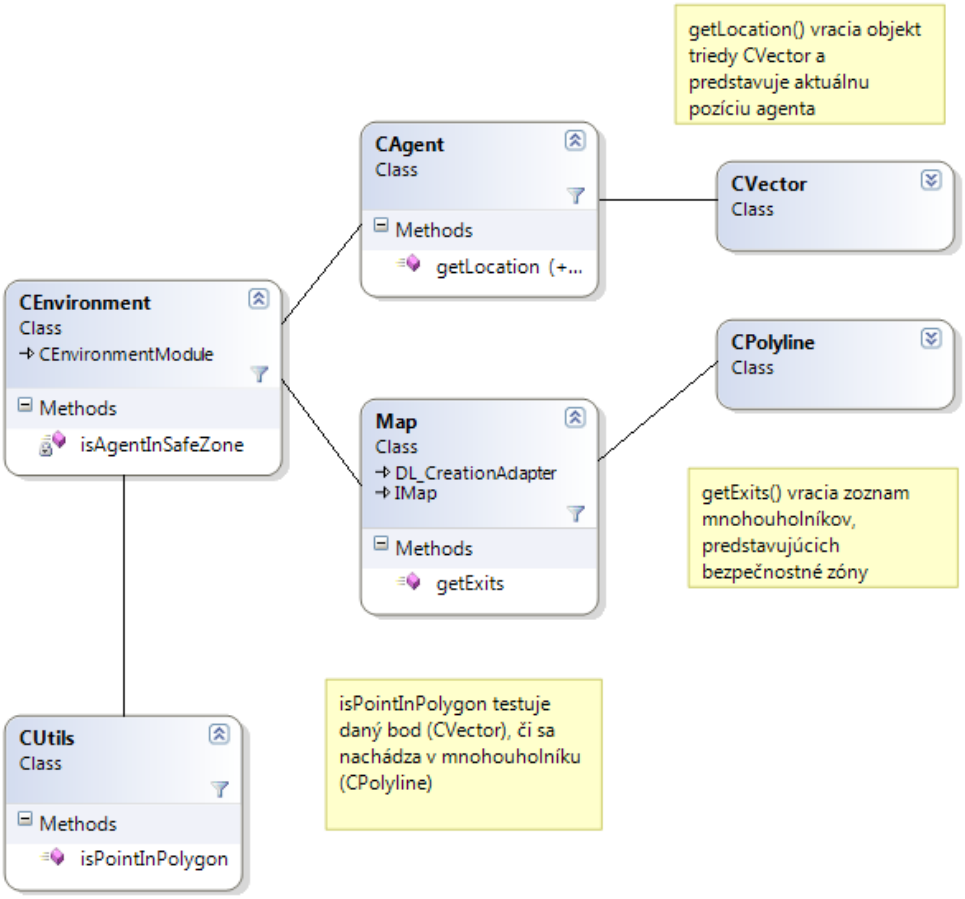
spracovávaných agentov. Aby mohla aplikácia zistiť, či môže daného agenta prestať manažovať, je potrebné zistiť, či sa nachádza v jednej z bezpečnostných zón.

Návrh riešenia

V každom kroku simulácie je potrebné iterovať cez všetkých agentov. Každý agent musí byť následne otestovaný, či sa nenachádza v jednej z bezpečnostných zón. Celá funkcionalita je teda založená na vytvorení funkcie, ktorá kontroluje, či sa bod (aktuálna pozícia agenta vyjadrená súradnicami) nachádza v obdĺžniku (bezpečnostná zóna je reprezentovaná ako obdĺžnik).

Opis implementácie

Do triedy *CEnvironment* bola doplnená funkcia *isAgentInSafeZone*, ktorá berie ako paramater pointer na agenta. Funkcia z agenta zistí jeho aktuálnu polohu a preiteruje všetky bezpečnostné zóny na mape. Pre každú zónu zavolá funkciu, ktorá kontroluje, či sa bod nachádza v obdĺžniku. Za týmto účelom bola vytvorená funkcia *isPointInPolygon* v triede *CUtils*. Tá berie parametre - bod (trieda *CVector*), ktorý sa testuje a objekt triedy *CPolyline*, ktorá predstavuje mnohouholník. Implementácia je vyjadrená diagramom tried na obr. 5.1.



Obr. 5.1: Diagram tried pre kontrolu dosiahnutia výstupného stavu agenta

5.2 Riešenie kolízii agentov

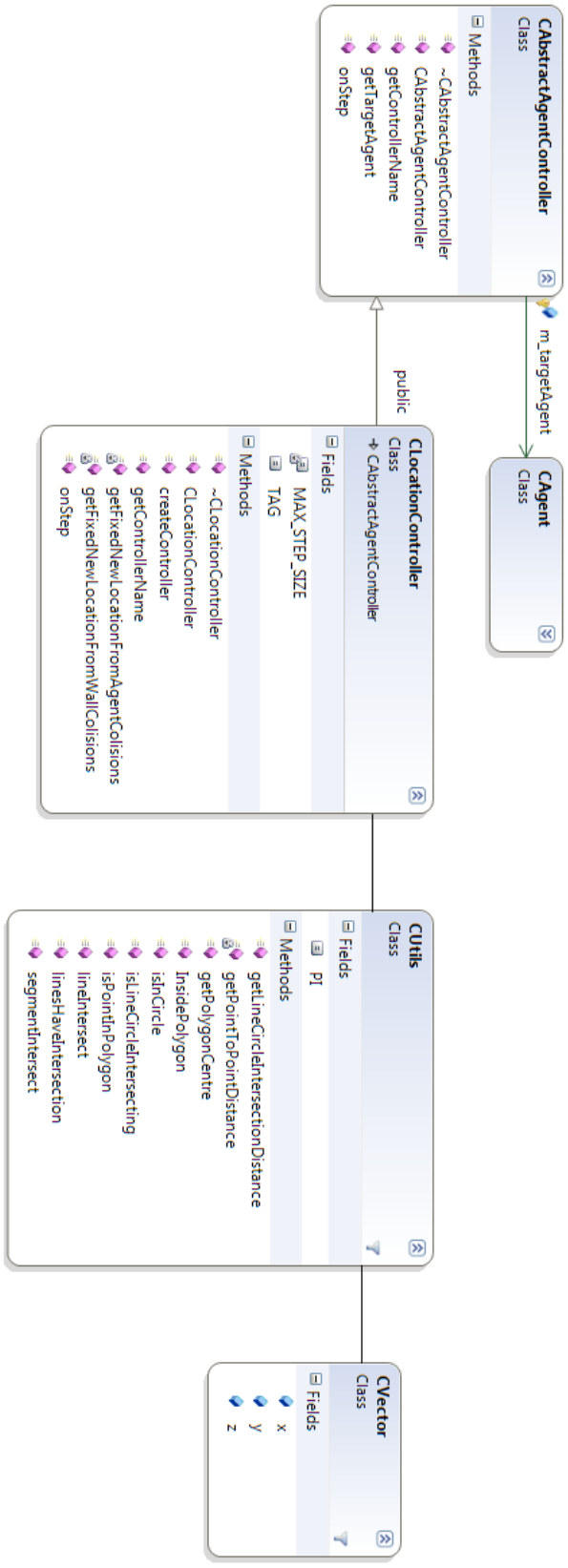
Návrh riešenia

Počas simulácie môže nastať situácia, kedy sa agent snaží dostať na pozíciu, kde sa nachádza už druhý agent. V takomto prípade je nutné, aby prostredie agenta dostalo len na pozíciu kam je možné sa dostať – zabráni sa tým kolíziám.

Opis implementácie

Agent sa počas jedného framu može posunúť najviac o 5cm. Štartovací bod agenta spolu s koncovým bodom (želaným posunom) vytvárajú vektor pohybu. V koncovom bode podľa dvnojnásobnej šírky agenta (súčet šírky posunutého agenta s šírkou agenta s ktorým nastáva kolízia) sa zistí, či nastáva kolízia s inými agentami. Ak áno, získa sa pozícia najbližšieho agenta k štartovaciemu bodu. Následne sa vytvorí kružnica s polomerom šírky agenta a jej priesečník s vektorom pohybu je maximálny posun

agenta. Implementácia bola ostestovaná pomocou debugovania vo Visual Studio 2010. Implementácia je vyjadrená diagramom tried na obr. 5.2.



Obr. 5.2: Diagram tried pre riešenie kolízii agentov

5.3 Prenos generovania agentov do nového projektu

Analýza problému

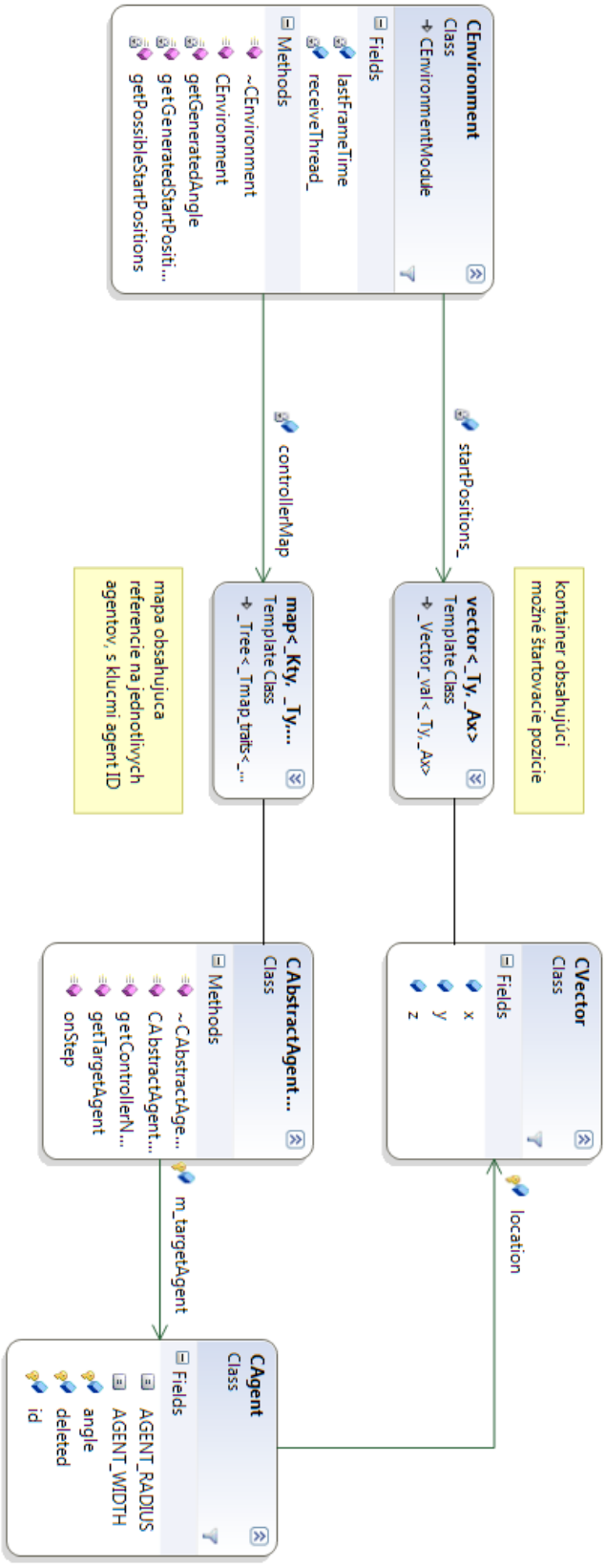
Generovanie agentov do spanov je zatiaľ len v projekte *CrowdSim*. Túto funkcionalitu je potrebné prepísať aj do separátneho projektu Enviroment.

Návrh riešenia

V mape ktorá sa inicializuje pri štarte aplikácie sa nachádzajú plochy určené na generovanie agentov. Tieto plochy (obdĺžniky) rozdelím na mriežku, kde jedna bunka má veľkosť 0,5 metra. Táto veľkosť reprezentuje maximálnu šírku agenta. Do kontajnera *startPositions* pridám všetky možné štartovacie pozície agentov. Následne v časti generovania štartovacej pozície generujem číslo v intervale od 0 po veľkosť kontajnera *startPositions*.

Opis implementácie

Pri štarte aplikácie sa cez parameter konštruktora inicializuje mapa pre prostredie. Pomocou tejto mapy sa vypočítajú všetky možné umiestnenia agentov (tak aby nenastali kolízie) a tieto pozície sa uložia do kontajnera *startPositions*. Pri vytváraní jednotlivých agentov sa skontroluje, či sa v kontajneri *startPositions* nachádzajú ešte voľné pozície - ak áno, tak sa vygeneruje náhodná pozícia v danom kontajnery. Z tejto pozície sa vyberú informácie o pozícií na mape a priradia sa novému agentovi. Daná pozícia sa následne odstráni z kontajnera *startPositions*. Implementácia bola ostestovaná pomocou debugovania vo Visual Studio 2010. Implementácia je vyjadrená diagramom tried na obr. 5.3.



Obr. 5.3: Diagram tried pre generovania agentov

5.4 Prenos základnej funkcionality agentov do nového projektu

Analýza problému

Základná funkcionalita agentov (generovanie aktivity – natočenie, pohyb) sa nachádza len v pôvodnom projekte *CrowdSim*. Funkcionalitu je potrebné premiestniť aj do separátneho projektu *Agent*.

Návrh riešenia

Prvotný návrh bolo vytvorenie primitívneho agenta, ktorého správanie bolo špecifikované generovaním náhodných akcií pohyb alebo uhol natočenia.

Opis implementácie

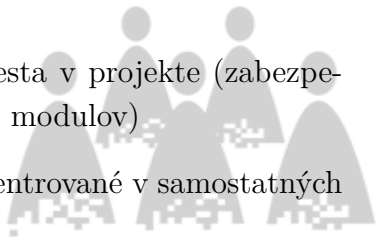
Implementácia agenta obsahovala jeho základný vnútorný stav - polohu, kde sa nachádza a natočenie. Každý vytvorený agent bol navrhnutý, aby neustále zásoboval prostredie požiadavky na pohyb. V priemere každá piata požiadavka bola akcia na pootočenie o uhol v rozmedzí -30 až 30 stupňov. Agenty sa pohybovali po mape nezávisle a nebrali do úvahy rozmery, ani umiestnenie na obrazovke. Implementácia bola ostestovaná pomocou debugovania vo Visual Studio 2010.

5.5 Zdokumentovanie DLL

Analýza problému

Implementovanie samostatných modulov bolo vykonané z viacerých dôvodov:

- Jednotlivé celky tvoria samostatné komponenty (moduly)
- Centralizované kontrolovanie vytvárania aj ukončovania jednotlivých DLL projektov v rámci jedného modulu
- Centralizované inicializovanie modulov z jedného miesta v projekte (zabezpečuje trieda, ktorá nepotrebuje vedieť detaily ohľadom modulov)
- Implementačné detaily jednotlivých modulov sú koncentrované v samostatných projektoch (nižšie vrstvy)
- Komunikácia medzi modulmi musí mať spoločné rozhranie, aby bolo možné moduly distribuovať, avšak moduly nesmú poznať konkrétnu implementáciu komunikačného rozhrania



- Musí existovať aplikácia, ktorá spúšťa moduly a z predchádzajúceho bodu vyplýva, že musí poskytovať komunikačný kanál pre moduly, ktoré aplikácia spustila

Opis implementácie

Implementácia a spolupráca modulov je vyjadrená diagramom tried na obr. 5.4. Opis konkrétnej implementácie všetkých modulov:

Wrapper

Komponent, ktorý spúšťa moduly *CVisualization, CRealAgent, CEnvironmentModule* a poskytuje im komunikačný kanál. Jeho úlohou je zabaliť načítané moduly do konkrétnej implementácie *CAbstractWrapper*, ktorá musí byť zadefinovaná v samotnom *Wrapper* komponente. Účel je taký, že *Wrapper* je notifikovaný, keďsa jeden z modulov ukončí. Vtedy sa vypne celá aplikácia.

CVisualization

Modul, ktorého úloha je vykresľovať simuláciu.

CRealAgent

Modul, ktorého úloha je simulovať správanie agentov.

CEnvironment

Konkrétna implementácia prostredia.

CAbstractClientModule a CEnvironmentModule

Sú založené na *Client-Server* architektúre. Prostredie manažuje všetkých agentov a teda vystupuje ako server a ostatné komponenty predstavujú klientov. *CAbstractClientModule* je hlavná trieda hlavná trieda pre *CRealAgent* a *CVisualization*. *CEnvironmentModule* je hlavná trieda pre *CEnvironment*.

CCommonModule

Je spoločná hlavná trieda pre *CEnvironmentModule* a *CAbstractClientModule*. Tu sa vykonáva riadenie inicializácie a ukončovania modulov - rozhoduje sa, kedy zavolať funkcie *onStart, onStop a onFrame*. Metóda *onStart* je callback metóda, ktorá sa zavolá hneď po spustení modulu, *onStop* sa zavolá predtým ako sa modul ukončí

a *onFrame* je zavolaná pre každý frame. Funkcia *onFrame* sa vykonáva 20-krát za sekundu.

CAgentManager

Je to manažér všetkých agentov. Tento objekt vytvára a rušia noví agenti. Poskytuje metódy:

- getAgents – vráti zoznam všetkých agentov
- getAgent – vráti referenciu na jedného agenta podľa jeho identifikátora
- removeAgent – označí agenta na vymazanie
- flush – vymaže všetkých agentov, ktorí boli označení na vymazanie

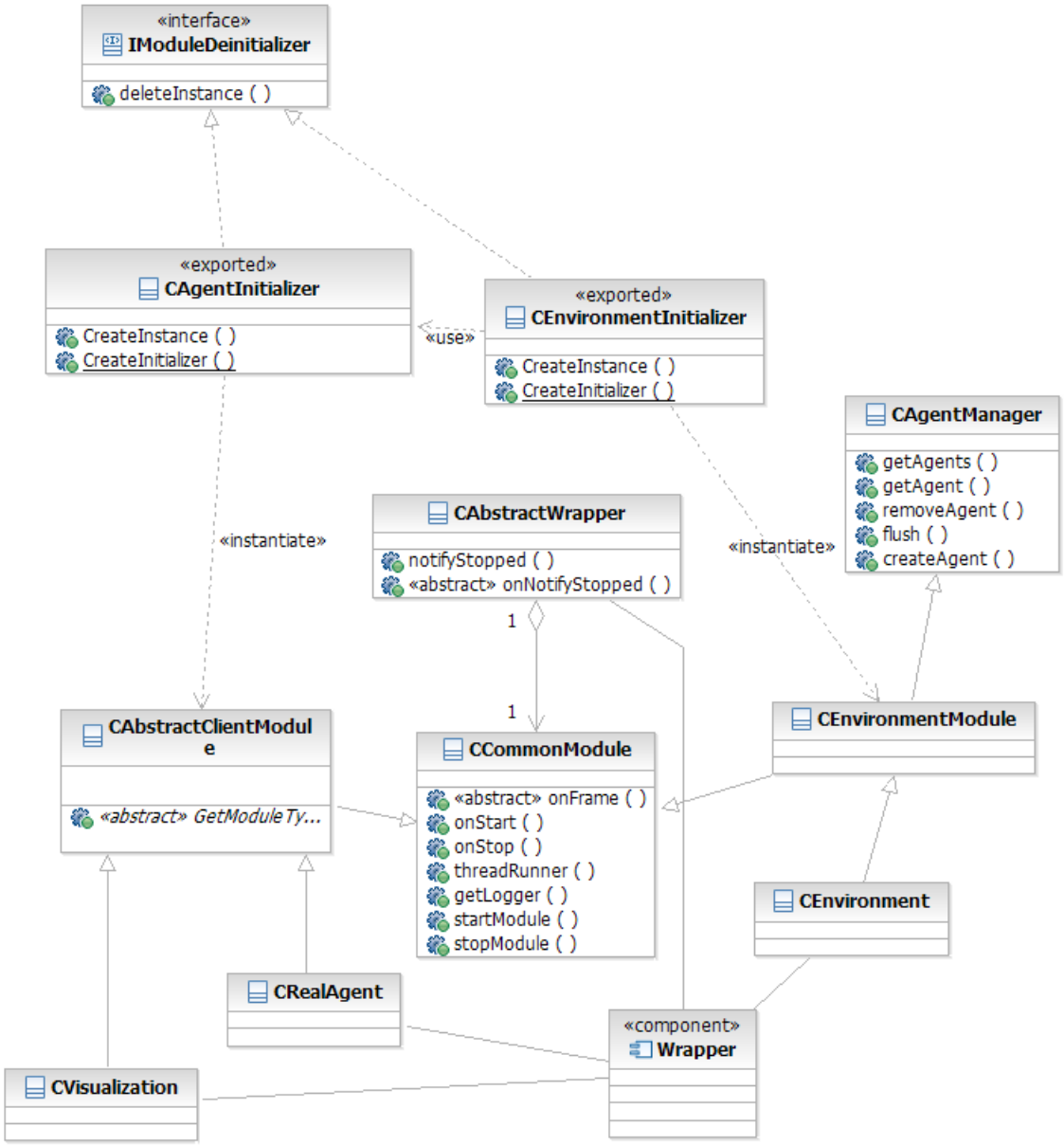
CAgentInitializer a CEnvironmentInitializer

Tieto triedy majú za úlohu vytvoriť samotné moduly. Metódy:

- CreateInitializer – statická metóda, ktorá vytvorí *CAgentInitializer*, alebo *CEnvironmentInitializer* (cesta k .dll knižnici sa zadáva ako parameter)
- CreateInstance – vytvorí novú inštanciu *CAbstractClientModule* alebo *CEnvironmentModule*

IModuleDeinitializer

Tento interface má na starosti deštrukciu vytvoreného modulu.



Obr. 5.4: Diagram tried pre návrh modulov

5.6 Prenos máp do nového projektu

Analýza problému

Základná implementácia máp je plne funkčná, ale na základe rozhodnutia prerobiť štruktúru starého projektu a presunúť funkčnosť do nového projektu je nutné prispôsobiť starú implementáciu máp prostrediu nového projektu.

Návrh riešenia

Mapy sú používané vizualizačným modulom, modulom prostredia a modulom pre správu agentov. Z toho vyplýva, že mapy nemôžeme do projektu vložiť ako samostatný modul, ale je nutné ich nastaviť v novom projekte ako zdieľané, aby sme mali priamy prístup k ich funkcionalite.

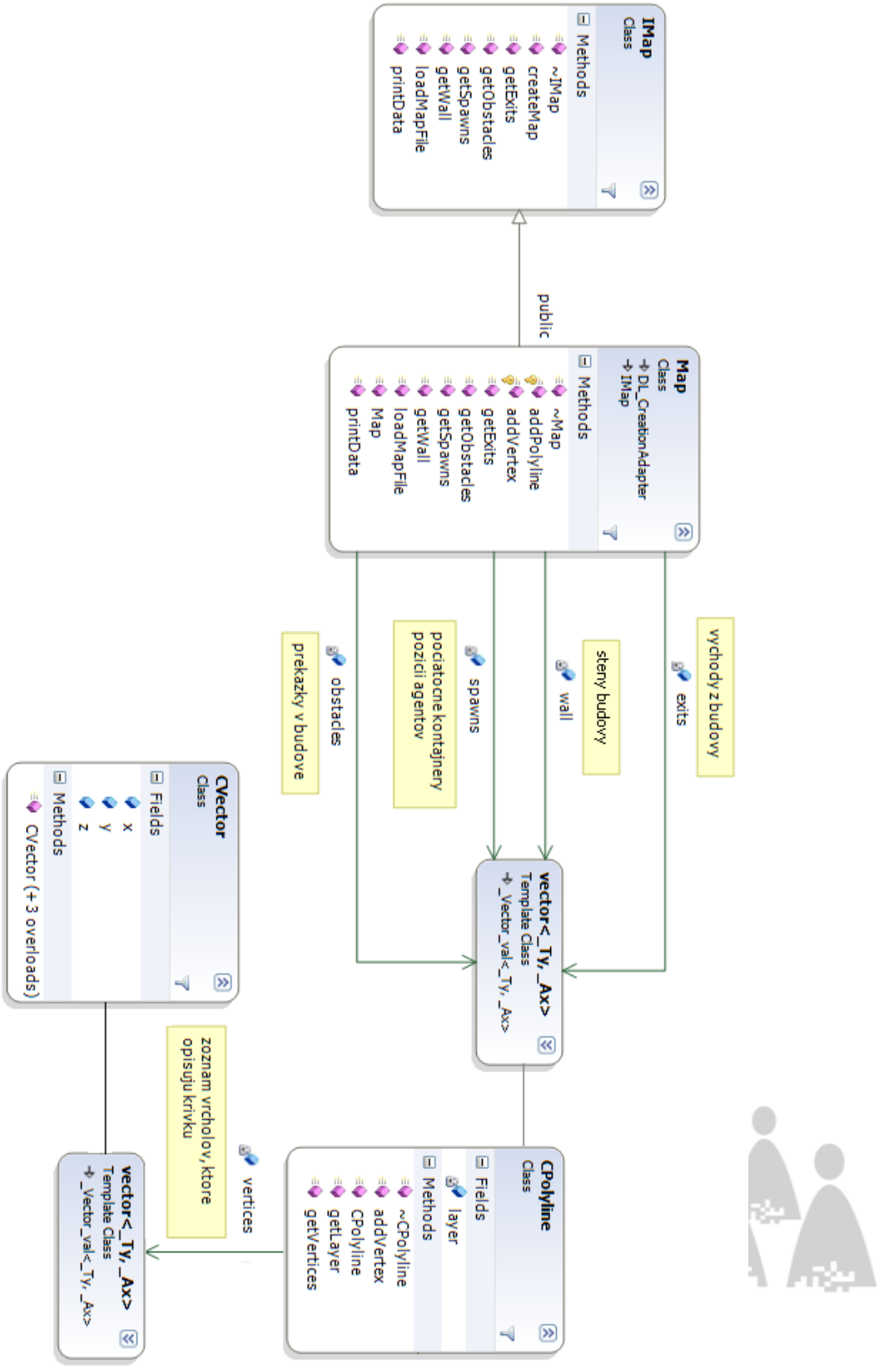
Keďže pre beh aplikácie stačí načítanie len jednej mapy, tak nám stačí spravovať len jednu inštanciu mapy. Toto zabezpečíme použitím návrhového vzoru Abstract Factory, ktorý pri prvom volaní vytvorí pri inicializácii mapu a následne je mapa pri ďalších volaniach vrátená. Týmto spôsobom zabezpečíme optimalizovaný prístup k mape.

Pri presune je dôležité, aby knižnica DXFLib, ktorá zabezpečuje preklad máp z DXF formátu do dátových štruktúr bola vhodne importovaná do projektu. Dôvodom je, že knižnice sa nachádzajú mimo hlavné adresáre nového projektu.

Opis implementácie

Pri inicializácii sa postupne čítajú údaje zo zvoleného súboru. Čítanie každej komponenty mapy je zabezpečené prostredníctvom volaní metód knižnice DXFLib. Pre správne načítanie mapy je potrebné spracovať údaje kriviek a vrcholov. Každá spracovávaná krivka zavolá metódu *addPolyline*, ktorá zistí identifikátor vrstvy, na ktorom sa krivka nachádza a na jej základe pridelí krivku do konkrétneho kontajnera podľa charakteru objektu.

Pre každú novú krivku je následne nutné spracovať údaje vrcholov metódou *addVertex*, ktoré opisujú pozíciu krivky v mape. Každá krivka obsahuje 5 vrcholov, pričom ak sú prvý a posledný vrchol v zhode, tak sa jedná o uzavreté objekty. Krivky sa môžu nachádzať v štyroch vrstvách podľa toho aký druh objektu reprezentujú. Východy z budovy sú v kontajneri *exits*, ktoré sa snažia agenty dosiahnuť, *walls* sú steny budovy, ktoré ohraničujú priestor pohybu agentov, *obstacles* sú vnútorné prekážky v budove a *spawns* predstavujú oblasti počiatočných pozícií agentov. Implementácia je vyjadrená diagramom tried na obr. 5.5.



Obr. 5.5: Diagram tried pre prenos máp

5.7 Distribúcia mapy modulom

Analýza problému, Návrh riešenia, Opis implementácie

Najjednoduchšie riešenie je posielať každému modulu cestu k súboru s mapou. Po implementácii modulov opísaných v časti *Zdokumentovanie DLL* stačí pridať do konštruktora triedy *CCommonModule* parameter, ktorý predstavuje názov súboru s mapou.

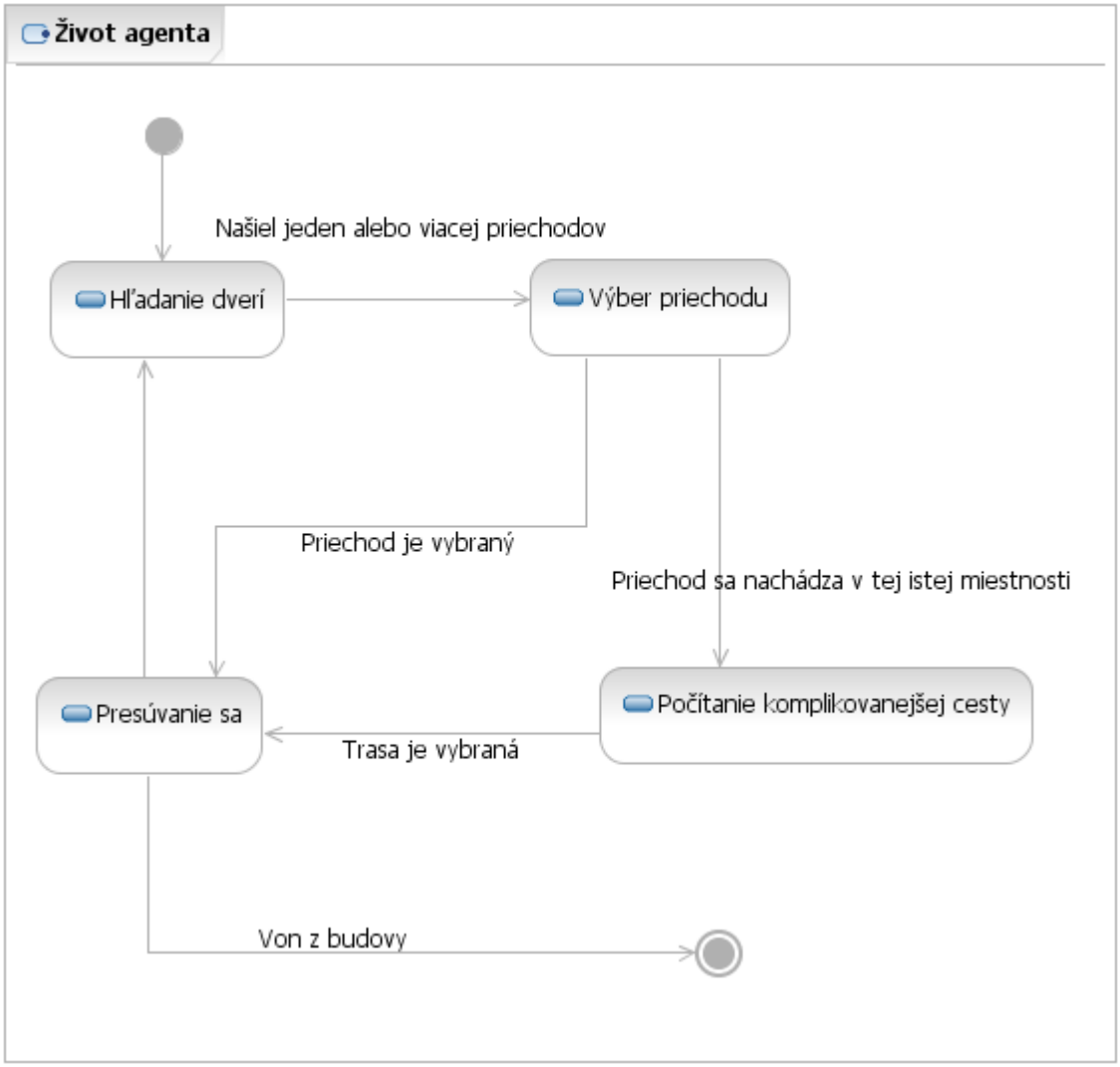
5.8 Návrh plánovania rozhraní pohybu agentov

Analýza problému

Agent počas svojho životného cyklu potrebuje vykonať určité činnosti, ako hľadanie dverí, pohyb po vybranej ceste, premýšlanie nad alternatívnou cestou atď. Z tohto dôvodu je vhodné použiť stavový automat, ktorý rozhoduje, aká činnosť bude vykonaná v nasledujúcom kroku.

Návrh riešenia

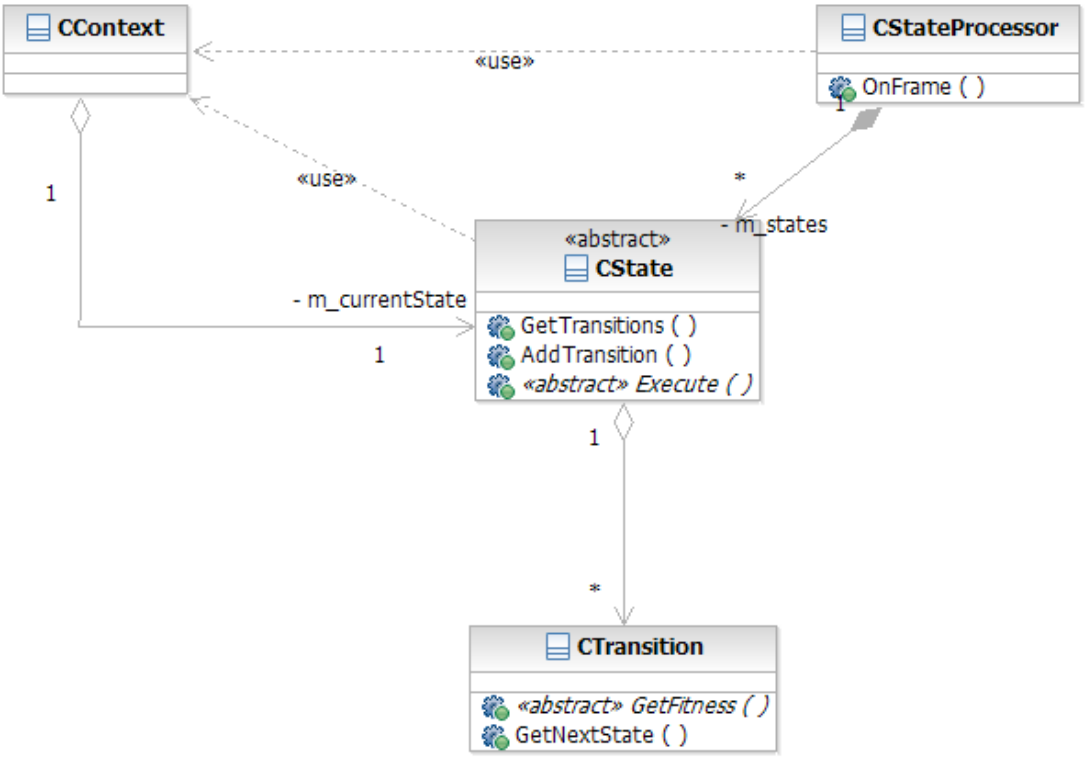
Navrhnutý stavový automat je na obr. 5.6.



Obr. 5.6: Stavový diagram agenta

Opis implementácie

Pre simulovanie stavov a prechodov medzi stavmi je použitý jeden procesor stavového automatu, jedna abstraktná trieda stavu a abstraktná trieda prechodov, ktorá vracia určitú fitnes. Ak je hodnota fitnes niektorého stavu väčšia ako nula tak sa prejde do tohto stavu. Ak je viac takých stavov, tak sa pôjde do stavu, ktorý má najvyššiu fitnes hodnotu. Implementácia je vyjadrená diagramom tried na obr. 5.7.



Obr. 5.7: Diagram tried pre stavový automat

CStateProcessor

Je trieda, ktorá ovláda celý automat. Pre každého agenta sa vyhodnocovanie automatom spúšťa cez funkciu *OnFrame*. Počas behu aplikácie je vytvorená len jedna inštancia *CStateProcessoru* ale vstup do metódy *OnFrame* je inštancia *CContext*, čo je kontajner, ktorý opisuje všetky možné vlastnosti, ktoré by agent pri prechode medzi stavmi potreboval.

CTransition

Je abstraktná trieda, ktorá reprezentuje jeden prechod medzi stavmi. Funkcia *GetFitness* je abstraktná a pre konkrétny prechod určí, či sú splnené podmienky, aby sa prešlo do ďalšieho stavu. Funkcia *GetNextState* vráti stav do ktorého sa má agent presunúť.

CState

Abstraktná trieda pre jeden stav. Abstraktná funkcia *Execute* je funkcia, ktorá vykoná, čo sa má stať počas jedného framu. *AddTransition* je funkcia ktorá pridá prechod pre stav.

5.9 Implementácia časovania prostredia

Analýza problému

V aktuálnom stave prostredie vykonáva kroky simulácie čo najväčšou rýchlosťou, čo má za následok maximálne vyťaženie procesora. Zároveň, ak nie sú agenti schopní v takom tempe zasielané informácie spracovávať, v komunikačnom kanáli sa nahromadí priveľké množstvo informácií. V extrémnom prípade môže zaplniť aj všetku operačnú pamäť systému.

Návrh riešenia

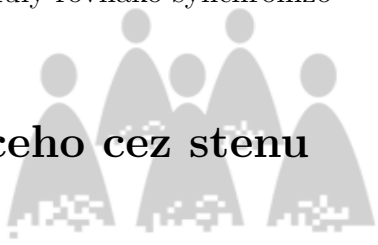
Prostredie musí medzi jednotlivými krokmi čakať. Doba čakania by mala byt primeraná, tak aby sa za 1 sekundu stihlo prostredie urobiť 20 simulačných krokov (20 FPS).

Opis implementácie

V triede *CCommonModule* sme upravili metódu predstavujúcu vlákno modulu. V životnom cykle modulu, v ktorom sa opakovane volá metóda *onFrame* (simulačný krok) sme pridali časovanie. Časovanie spočíva v nasledujúcich činnostiach:

1. Určenie aktuálneho času
2. Posunutie tohto času k najbližšej časovej reprezentácii simulačného kroku v budúcnosti
3. Vykonanie simulačného kroku
4. Počkanie, až kým sa nedostaneme do času určeného v bode č. 2
5. Pokračujeme bodom č. 1

Tým pádom, ak krok trval dlhšie ako sme na jeden krok vyhradili, čakanie vôbec nenastane. Výhoda tohto riešenia je aj tá, že sú všetky moduly rovnako synchronizované, aj keď nie všetky to teraz reálne využívajú.



5.10 Detekcia agenta prechádzajúceho cez stenu

Analýza problému

Treba zabrániť tomu, aby agenty mohli prechádzať cez steny. Aj keď by sa mali narážaniu do stien vyhýbať, je treba implementovať ochranu v rámci prostredia ktorá to zabezpečí - pre prípad, že agent urobí chybu.

Návrh riešenia

Pri každom kroku agenta budeme kontrolovať, či jeho konečná pozícia po kroku nebude kolidovať s niektorou zo stien alebo prekážok. Ak by malo dôjsť ku kolízii, agent naplánovaný krok nevykoná. To si môžeme dovoliť vďaka tomu, že kroky ktoré agent vykonáva sú veľmi malé, takže agent sa aj naďalej bude môcť priblížiť veľmi blízko ku stene.

Opis implementácie

Funkcionalita je implementovaná v rámci triedy *LocationController* v module *Environment*. Algoritmus je popísaný nasledujúcimi krokmi:

- V každom kroku
  - Pre každú stenu a prekážku
    - Zistíme, či existuje priesečník steny a kružnice so stredom v koncovom bode pohybu agenta a polomerom agenta
    - Ak existuje, krok sa nevykoná

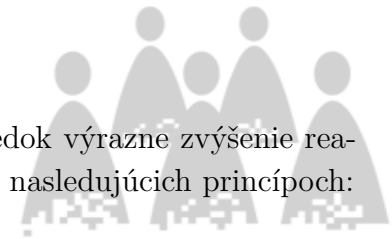
Kapitola 6

Šprint#4

ID	Názov úlohy	Zodpovedný	Kapitola
1	Návrh koncepcie ovplyvňovania pohybu agentov na základe hustoty výskytu ostatných agentov	Adam pomothy, Michal Fornádeľ	6.1
2	Vytvorenie kompletnej cesty agenta do cieľa na základe NavigationMesh	Lukáš Pavlech	6.2
3	Vytvorenie vrstvy pre NavigationMesh do mapy	Marek Hlaváč	6.3
4	Programová reprezentácia NavigationMesh	Marek Hlaváč	6.4
5	Implementácia návrhu stavového automatu pre agenta	Martin Košický	
6	Generovanie pohybov agenta na základe naplánovanej cesty	Daniel Petráš	6.5
7	Integrácia existujúcej dokumentácie	Michal Fornádeľ	
8	Zosúladenie zápisníc zo stretnutí a doplnenie ID z Redminu	Michal Fornádeľ	

6.1 Aplikovanie hustoty na model

Analýza problému



Použitie dynamiky tekutín pri simulácii davu má za následok výrazne zvýšenie realistickosti správania agentov. Táto technika je založená na nasledujúcich princípoch:

1. **Každá osoba má svoj cieľ – určitú oblasť na mape**
    - Každá osoba musí mať svoj cieľ daný explicitne pred začatím simulácie

2. **Každá osoba sa pohybuje maximálnou rýchlosťou akou je to v danom okamžiku možné**

    - Na rýchlosť vplýva prostredie (napr. po chodníku sa dá ísť rýchlejšie ako po tráve)
    - Rýchlosť ovplyvňujú aj iní ľudia

3. **Mapa obsahuje zóny nepohodlia**

    - Osoba si pri voľbe cesty vyberie tú cestu, ktorá ma menšiu hodnotu nepohodlia

Rýchlosť

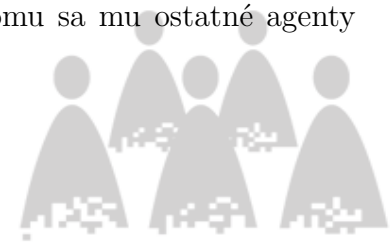
Rýchlosť je ovplyvnená nie len povrchom prostredia, ale najmä hustotou iných agentov. Rýchlosť sa zmenšuje, ak ide agent proti prúdu ostatných agentov a nemení sa, ak ide po prúde. Inými slovami, rýchlosť je density-dependent. Závisí na hustote. Rýchlosť je vyjadrená vektorom – a teda vyjadruje zároveň aj smer agenta.

Hustota

Každý agent má svoju hustotu. Najvyššia je v jeho strede a postupne klesá smerom od jeho stredu. Polomer hustoty agenta sa nastaví explicitne. Celková hustota davu sa vypočíta ako suma všetkých hustôt agentov. V oblastiach s malou hustotou je rýchlosť agenta rovná rýchlosti danej prostredím. Tu sa rieši najmä sklon povrchu. Ak je agent v oblasti s veľkou hustotou, pohybuje sa rýchlosťou davu (ktorý spôsobil tú hustotu). Čo je veľká a čo je malá hustota sa explicitne určí. V miestach so strednou hodnotou sa robí kombinácia rýchlosti danej prostredím a rýchlosti toku davu.

Vyhýbanie agentov

Každý agent má pred sebou zónu nepohodlia – vďaka tomu sa mu ostatné agenty vyhnú.
Agent vyberá cestu s najlepším pomerom hodnôt:



- Dĺžka cesty
- Čas, koľko zaberie cesta
- Miera nepohodlia za jednotku času na ceste

Tieto hodnoty sú vyjadrené výrazom 6.1

$$C \equiv \frac{\alpha f + \beta + \gamma g}{f}$$

Obr. 6.1: Unit cost field

Kde

- C - tzv. unit cost field – cena za trasu
- α - dĺžka cesty
- β - čas, koľko zaberie cesta
- γ - miera nepohodlia za jednotku času na ceste

Následne sa vypočíta táto hodnota pre celú potenciálnu cestu pomocou integrálu (výraz 6.2)

$$\int_P C ds$$

Obr. 6.2: Unit cost field pre celú trasu

Kde

- P - označuje trasu/cestu
- C - unit cost field
- ds - predstavuje celkovú dĺžku trasy agenta

Mapa sa rozdelí do štvorcov a vytvorí sa vektorové pole pre mapu pomocou aplikovania gradientu tejto funkcie na jednotlivé štvorce. Agenty sa potom pohybujú v opačnom smere vektorov gradientov, čo znamená, že si vyberajú smer, kde sa klesá hodnota funkcie 6.2.

Návrh riešenia

Naša simulácia davu neobsahuje viaceré parametre, ktoré sú potrebné pre základný model Continuum Crowds, ktorý je popísaný vyššie. Naším cieľom nie je aplikovať presný model, ale využiť ho ako inšpiráciu pri integrovaní hustoty davu do nášho projektu. Aby sme sa priblížili pôvodnej myšlienke podobnosti davu ľudí s pohybom kvapaliny, rozhodli sme sa taktiež vytvoriť na mape vektorové pole.

V našom prípade berieme do úvahy iba vektory rýchlosti. Celá mapa je rozdelená na štvorce. Pre každý štvorec sa urobí súčet vektorov rýchlosti pre všetkých agentov, ktorí sa nachadzajú v danom štvorci. Tým získame výsledný vektor pre daný štvorec.

Následne je vektor rýchlosti každého agenta upravený podľa tohto sumárneho vektora, čim sa simuluje jednoliatosť kvapaliny.

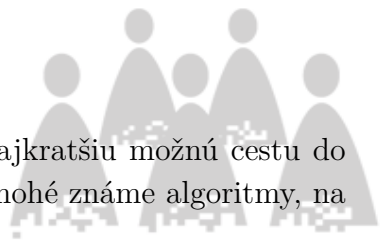
Opis implementácie

Prvým krokom je rozdelenie mapy na štvorce. Strana štvorca je zadefinovaná ako konštanta v triede DensityController, aby sa s ňou dalo jednoducho experimentovať. Pre každý identifikovaný štvorec v mape je potrebné testovať, či sa v ňom nachádza agent. To znamená iterovanie cez všetky štvorce a pre každý štvorec iteráciu cez všetkých agentov. Navyše, pri mapách zložitejších tvarov vznikajú štvorce, ktoré sú pre agentov úplne nedostupné, lebo sa nachádzajú za stenou. Preto sme zvolili iný prístup. Štvorce generujeme iba na miestach, kde sa naozaj nachádzajú agenty. Výsledný zoznam Štvorcov a k nim prislúchajúcich agentov je uložený v hash-mape, ktorá ako kľuč berie index štvorca a ako hodnotu berie zoznam agentov pre daný štvorec. Týmto sa výrazne zníži počet potrebných iterácií.

Následne je potrebné pre všetky záznamy v hash-mape prejsť všetky agenty a postupne robiť vektorový súčet ich vektorov rýchlosti. Po vypočítaní výsledného vektoru sa upravia smery vektorov rýchlosti každého agenta.

6.2 Vytvorenie kompletnej cesty agenta do cieľa na základe NavigationMesh

Analýza problému



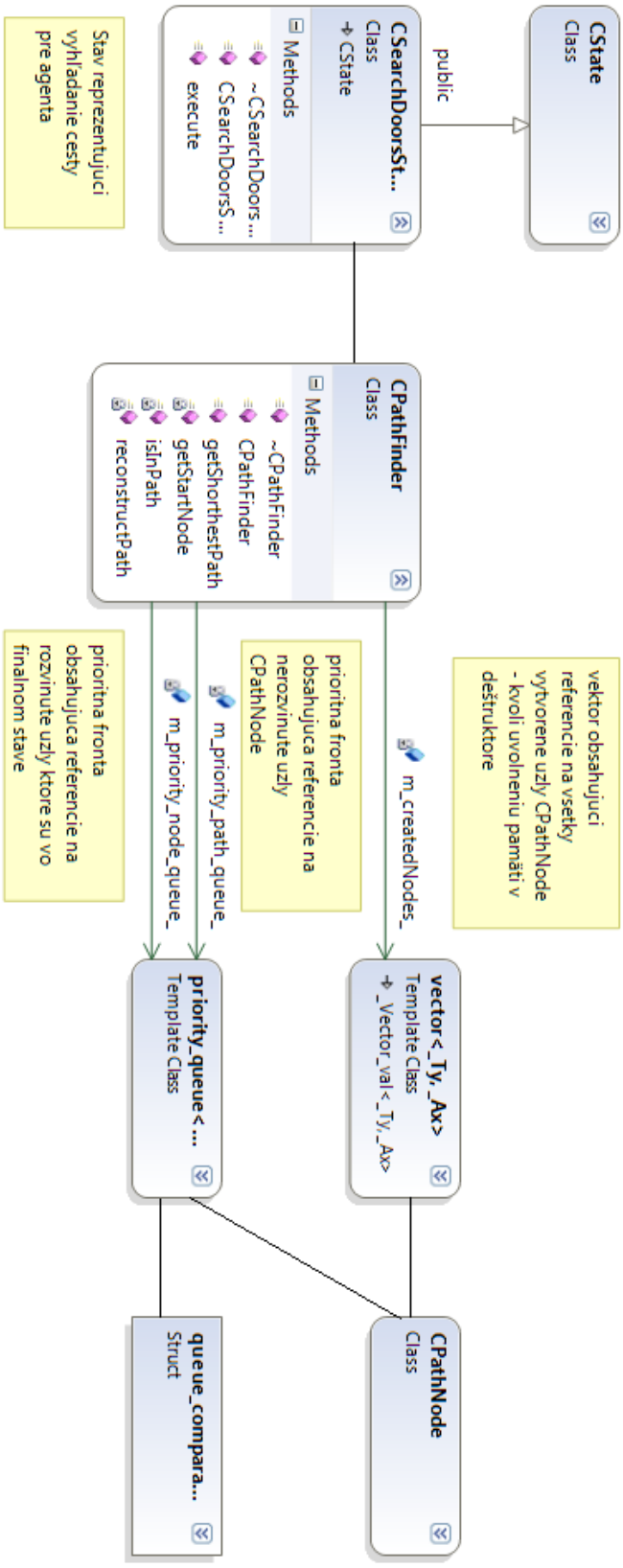
Na základe načítanej NavigationMesh je potrebné nájsť najkratšiu možnú cestu do cieľa (východ z budovy). Prehľadávaniu grafu sa venujú mnohé známe algoritmy, na účel problému bol zvolený algoritmus A*.

Návrh riešenia

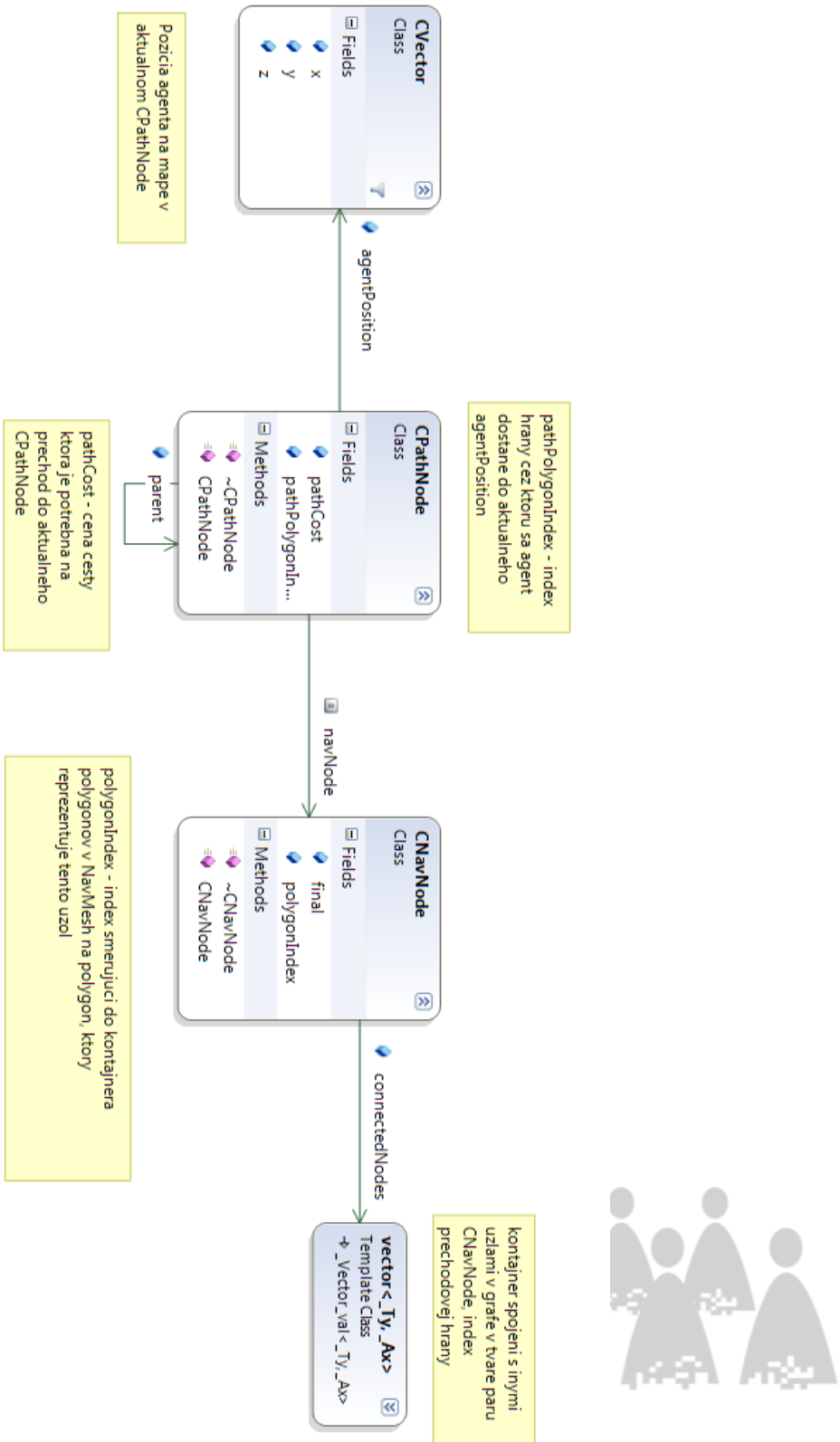
Agent pozná celú mapu budovy. Preto je možné ľahko vyhľadať najkratšiu možnú cestu z budovy. V stave *searchDoor* agent vyhľadá pomocou A* algoritmu najkratšiu možnú cestu a vráti je do kontext-u v tvare kontainera prechodových uzlov s zvolenými prechodovými hranami.

Opis implementácie

Agent v stave *searchDoor* získa z kontext-u navigationMesh a svoje umiestnenie. V triede *PathFinder* sa najprv zistí, v akom uzle sa agent nachádza. Následne sa začne prehľadávať graf uzlov, kde cena cesty sa rovná súčtu vzdialeností medzi doteraz navštívenými uzlami. Výber nasledujúceho prehľadávaného uzla sa určí podľa najnižšej hodnoty cesty v prioritnej fronte ciest. Ak sa počas prehľadávania dostaneme do cieľového uzlu, daná cesta sa uloží v prioritnej fronte ciest, ktoré sa rovnako ako uzly radia podľa najnižšej ceny cesty. Nakoniec sa vyberie z prioritnej fronty ciest cesta s najnižšou hodnotou ceny cesty a tá sa zrekonštruluje a vráti do kontext-u, čím agent prejde do stavu *followPath*. Implementácia bola ostestovaná pomocou debugovania vo Visual Studio 2010. Na obr. 6.3 je znázornená implementácia. Uzol grafu a jeho závislosti sú znázornené v diagrame tried na obr. obr. 6.4.



Obr. 6.3: Diagram tried pre vytvorenie kompletnej cesty agenta do cieľa na základe NavigationMesh



Obr. 6.4: Diagram tried - uzol v grafe, použitý pri vytváraní cesty agenta do cieľa

6.3 Vytvorenie vrstvy pre NavigationMesh do mapy

Analýza problému

Keďže navigácia agentov v priestore je problematická vzhľadom na objem výpočtov v jednom výpočtovom kroku aplikácie, tak je nutné nájsť vhodnú optimalizáciu, ktorá by uľahčovala výpočet pohybu agentov v priestore a plánovanie ďalších akcií.

Návrh riešenia

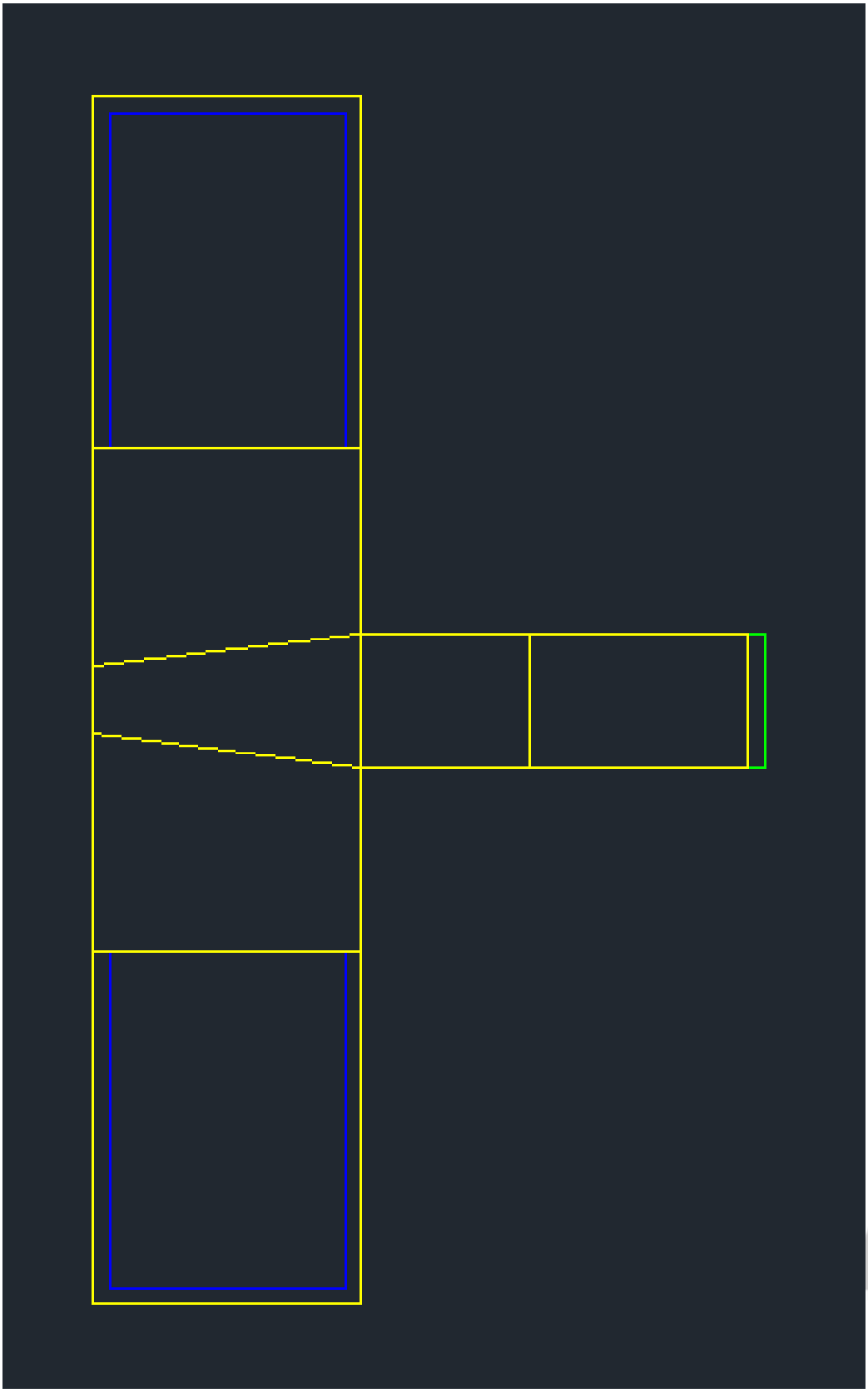
Ideálnym riešením je použitie NavigationMesh, ktorá predstavuje navigačnú sieť pre konkrétnu mapu. Pod navigačnou sieťou sa rozumie vrstva obsahujúca sadu prepojených objektov, ktorých cieľom je reprezentácia ciest. Takýmto spôsobom môžeme ohraničiť oblasti, v ktorých je možný pohyb agentov.

Navigačná sieť je zložená z polygónov, ktoré sa skladajú zo 4 hrán. To znamená, že každý polygón môže susediť so 4 inými polygónmi. Takýmto spôsobom môžeme poskladať sieť, ktorá je ľahko prispôsobiteľná štruktúre budovy.

Keďže navigačná sieť je natiahnutia na celú mapu, tak je zabezpečené zjednodušenie plánovania cesty agenta. Agent si môže ľahko získať polygón, v ktorom sa nachádza a následne zistiť možné postupy vzhľadom na susedné polygóny. Takýmto spôsobom môžeme prehľadávať odkryté a neodkryté časti mapy a prispôsobiť následné kroky agenta.

Opis implementácie

Do existujúcich máp je vložená nová vrstva, ktorá reprezentuje navigačnú sieť. Ostatné vrstvy sú bez zmeny. Polygóny sú vytvorené, takým spôsobom aby bola zabezpečená konzistencia navigačnej sieti vzhľadom na štruktúru budovy, teda žiadna hrana siete neprechádza cez steny štruktúry. Dôležitým faktorom, ktorý musel byť splnený je, aby susediace polygóny mali zdieľanú hranu. Ak by to nebolo splnené, tak by sme nevedeli zistiť grafovú štruktúru navigačnej siete. Ukážka navigačnej siete je na obr. 6.5.



Obr. 6.5: Ukážka vytvorenej navigačnej siete.

6.4 Programová reprezentácia NavigationMesh

Analýza problému

Navigačnú sieť je nutné vhodným spôsobom zapracovať do projektu. Toto zapracovanie nesie so sebou rozhodnutie vzhľadom na programovú reprezentáciu štruktúry siete. Dátové štruktúry musia byť vhodne zvolené, aby nebol narušený chod výpočtu v rámci jedného kroku.

Návrh riešenia

Pre voľbu vhodných dátových štruktúr pre reprezentáciu navigačnej siete môže existovať viacej riešení. Navrhované riešenie spočíva v rozdelení siete na jednotlivé častí, ktoré sú komponentmi siete a zároveň zabezpečujú funkcionalitu na vhodne zvolenej úrovni.

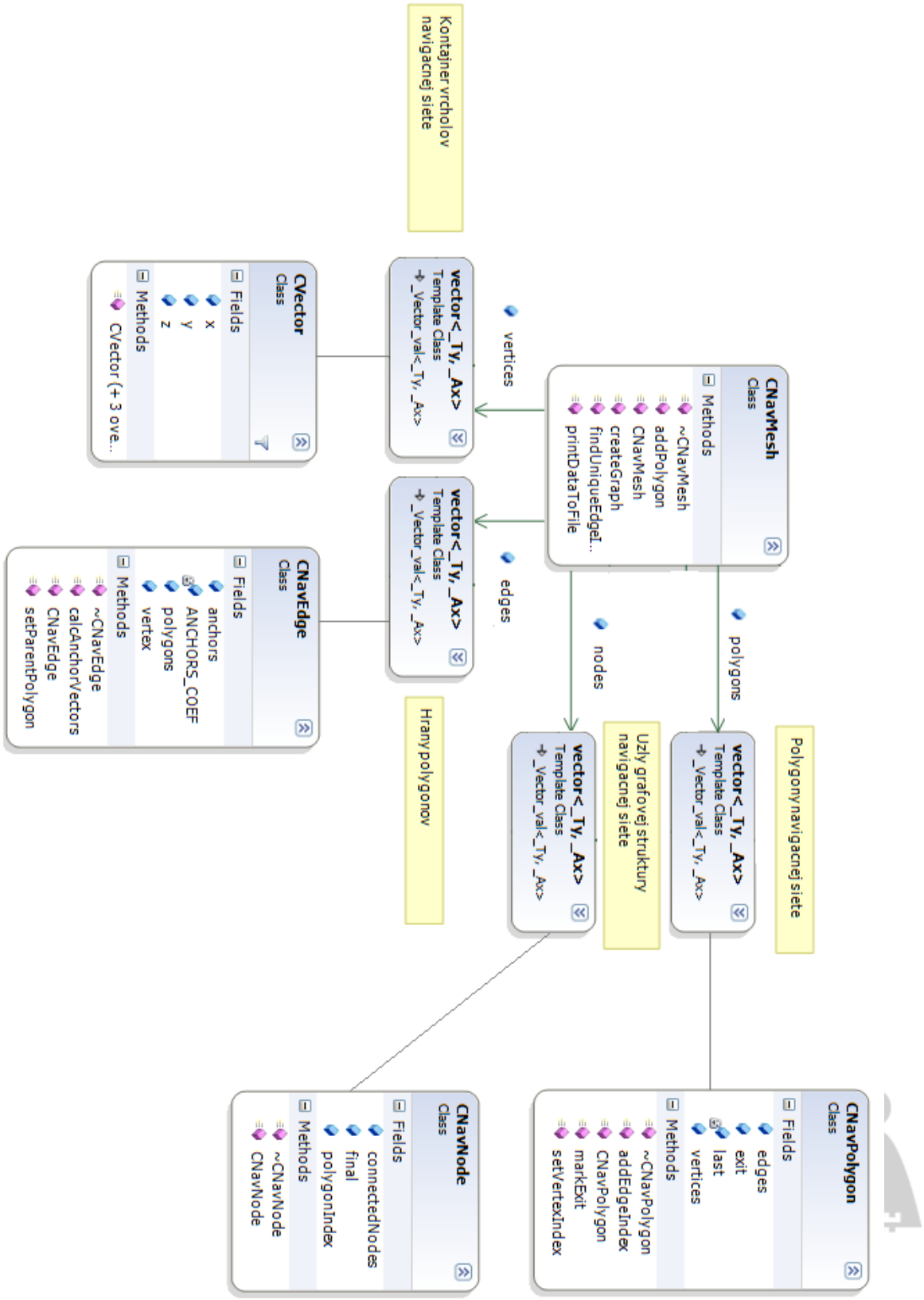
Takýmto spôsobom môžeme navigačnú sieť rozdeliť na komponenty, ktoré reprezentujú navigačnú sieť ako celok, ktorá obsahuje polygóny. Jednotlivé polygóny sa skladajú z hrán, pričom hrany sú reprezentované dvoma vrcholmi. Celá štruktúra grafu, ktorá je potrebná pre prácu v iných moduloch sa skladá z uzlov, ktoré sú alternatívou k polygónom. Avšak uzly v grafe reprezentujú navigačnú sieť z iného pohľadu. Ich cieľom je popis prepojení medzi jednotlivými uzlami v grafe prostredníctvom zdieľaných hrán.

Opis implementácie

V programe je sieť reprezentovaná prostredníctvom triedy *CNavMesh*, ktorá obsahuje všetky údaje navigačnej siete. Celá informácia siete v mape pozostáva z komponent, ktoré boli prečítané z konkrétnej vrstvy. Navigačná sieť je inicializovaná po vytvorení mapy. Navigačná sieť je najprv postupne napĺňaná polygónmi siete (angl. *meshes*). V programe sú reprezentované triedou *CNavPolygon*, ktorý sa skladá z hrán *CNavEdge* a tie z vrcholov *CVector*.

Grafová reprezentácia siete, ktorá je potrebná pre iné moduly sa vytvorí po inicializácii objektov. Pre každý polygón sa vytvorí uzol a následne sa určia prepojenia medzi dvojicami uzlov, ktoré majú rovnakú hranu. Pre východy je nutné taktiež vytvoriť polygóny, aby sme mohli identifikovať cieľový uzol, resp. stav v ktorom agenty opúšťajú simuláciu.

Všetky objekty sú držané v hlavnej inštancii navigačnej siete a samotné objekty si držia len referenčné indexy do príslušných kontajnerov v *CNavMesh* (*polygons, nodes, edges, vertices*). Ak chce niektorý modul pristupovať ku konkrétnym údajom, tak si prostredníctvom indexov dokáže vyhľadať potrebné údaje. Implementácia je vyjadrená diagramom tried na obr. 6.6.



Obr. 6.6: Ukážka vytvorenej navigačnej siete.

6.5 Generovanie pohybov agenta na základe naplánovanej cesty

Analýza problému

Po naplánovaní cesty by agent mal byť schopný navigácie po zvolenej ceste do cieľa. Keďže veľa agentov bude mať rovnakú naplánovanú cestu, treba zabezpečiť, aby do seba agenti príliš nenarážali.

Návrh riešenia

Agent sa bude do cieľa presúvať po stredoch hrán, ktoré spájajú susedné polygóny po naplánovanej ceste. Na to bude treba určiť, v ktorom polygóne sa agent aktuálne nachádza. Ako sa agent bude blížiť k najbližšej hrane, v istom momente, ešte pred tým ako vstúpi do ďalšieho polygónu, bude jeho cieľ zmenený na nasledujúcu hranu. Táto optimalizácia je zvolená kvôli tomu, aby sa agenty nesnažili za každú cenu dostať na presné miesto najbližšieho stredu hrany, aj keď logické by pre nich bolo smerovať svoje kroky už ďalej po ceste.

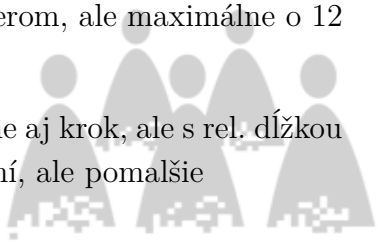
Separácia agentov jeden od druhého a vyhýbanie sa prekážkam, bude realizovaná úpravou vektora smerujúceho do ich cieľa. Samotná veľkosť úpravy bude váhovaná, ale vo všeobecnosti platí, že ak sa agent dostane do hraničnej blízkosti iného agenta alebo steny, bude sa snažiť od neho vzdialiť. Čím bližšie sa bude nachádzať, tým väčší bude vektor pomocou ktorého sa bude chcieť vzdialiť.

Opis implementácie

Opísaný je jeden simulačný krok na strane agenta v stave *CFollowPathState* v module *Agent*:

1. Ak nie je naplánovaná cesta, nič sa nestane
2. Získame uzol, v ktorom sa agent nachádza - hľadá sa medzi uzlami v naplánovanej ceste
3. Ak sa taký uzol nenašiel, znamená to, že sa agent dostal mimo plánovanej cesty, nasledovať bude prechod do stavu plánovania cesty (žiadnu inú akciu nevykonáme)
4. Inak pokračujeme získaním najbližšej hrany (ktorá je ale vzdialená viac ako je polomer agenta) po ceste
5. Ak sme popri tom narazili na koncový uzol, cieľom je jeho stred

6. Inak je cieľ stred nájdenej hrany
7. Vektor smerujúci do cieľa normalizujeme a vynásobíme váhou sledovania cesty
8. Získame separačný vektor
   (a) Postupne získame vzdialenosť medzi aktuálnym a všetkými ostatnými agentmi
   (b) Ak je táto vzdialenosť väčšia ako polomer agenta, ideme na ďalšieho v zozname
   (c) Inak pridáme do separačného vektora normalizovaný vektor od druhého agenta a aktuálného agenta a vydelíme ho ich vzdialenosťou (minimálne však 0,1)
   (d) Výsledok sú takto spočítané vektory
9. Získame vektor vyhýbania sa prekážkam
   (a) V aktuálnom polygóne nájdeme všetky hrany, ktoré nesusedia so žiadnym iným polygónom (čiže sa jedná o steny, alebo prekážky)
   (b) Postup je rovnaký ako pri určovaní separačného vektora, len tu sa berú do úvahy vzdialenosti agenta od stien a prekážok
   (c) Týmto spôsobom nie je treba prechádzať všetky steny v mape ale iba 4 hrany v jednom polygóne, v ktorom sa agent práve nachádza
10. Na cieľový vektor aplikujeme (pričítame) separačný vektor a vektor vyhýbania sa vynásobené ich váhami
11. Ak je cieľový vektor nulový, žiadnu akciu nevykonáme
12. Získame uhol o koľko sa musí agent natočiť, aby smeroval do cieľa
13. Ak je uhol blízky nule, spravíme krok s rel. dĺžkou kroku rovnej dĺžke cieľového vektora, ale maximálne rovnej 1
14. Ak uhol nie je blízky nule, natočíme sa žiadaným smerom, ale maximálne o 12 stupňov
15. Ak je potrebné otočenie menej ako 90 stupňov, urobíme aj krok, ale s rel. dĺžkou iba 0,5 – takže sa agent bude pohybovať aj pri otáčaní, ale pomalšie



Kapitola 7

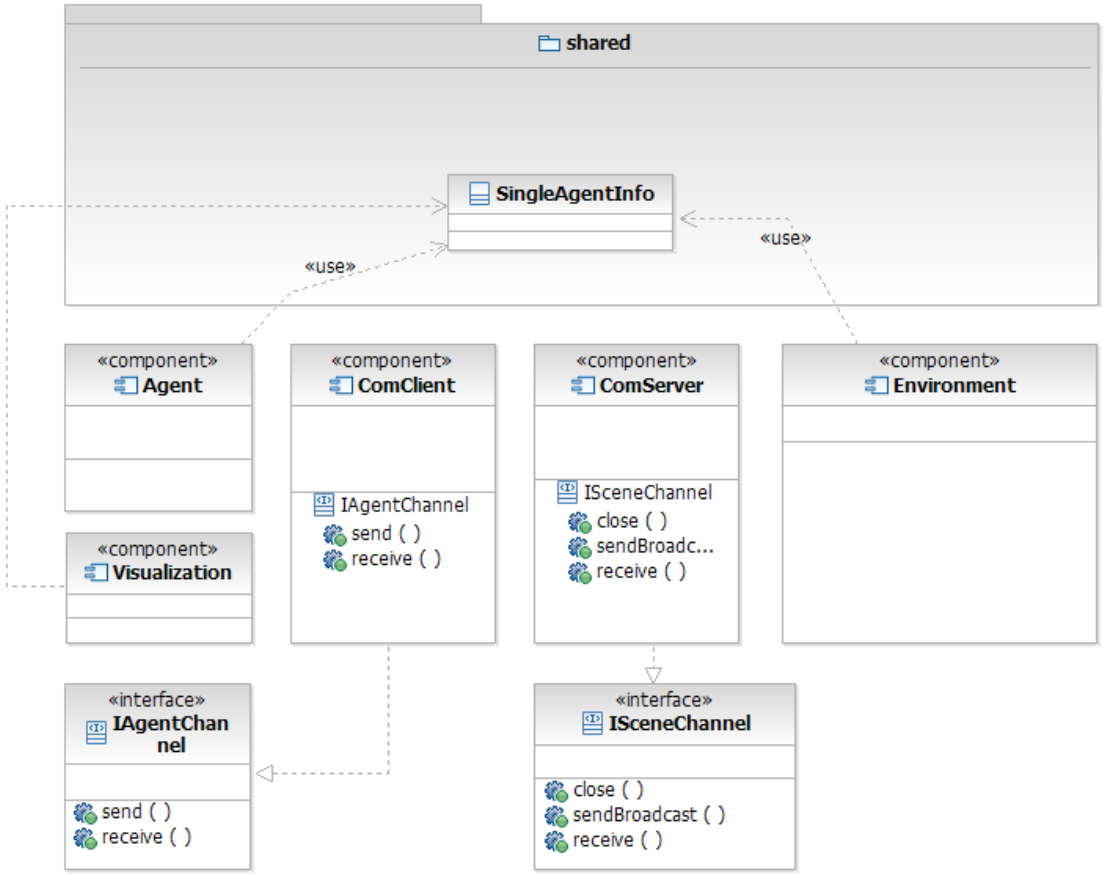
Opis prototypu

7.1 Architektúra prototypu

Celá architektúra je založená na princípe modulárnosti - aplikácia je rozdelená do niekoľkých komponentov. Komponent *Agent* a *Visualization* vystupujú v role klientov, ktorí komunikujú so server kompomentom *Environment* prostredníctvom komunikačných kanálov. Komunikačné kanály su reprezentované samostatnými triedami. Na strane servera je to *ComServer*, ktorý implementuje rozhranie *ISceneChannel* a na strane klientov je to *ComClient* , ktorý implementuje *IAgentChannel*.

Takýmto rozdelením je možné vytvoriť buď tri nezávislé aplikácie (.exe), alebo vytvoriť tri nezávislé projekty (.dll), ktoré bude spúšťať jeden Wrapper. Naša aktuálna implementácia vychádza z druhej varianty. Wrapper poskytuje systému komunikačné kanály, takže v prípade snahy o distribúciu je treba zmeniť implementáciu komunikačných kanálov.

Implementačné a funkcionálne detaily sú popísané v kapitolách, ktoré sa venujú jednotlivým šprintom. Základná architektúra systému je na obrázku 7.1.



Obr. 7.1: Základné komponenty prototypu

7.1.1 Komunikácia

Komunikácia medzi jednotlivými modulmi prebieha cez komunikačný kanál. V tejto komunikácii vystupuje modul prostredia ako server. Od agentov získava všetky potrebné informácie, na základe ktorých dokáže vizualizovať aktuálnu situáciu na mape. Aktuálnu mapu posiela agentom a modulu vizualizácie. Vizualizácia aktuálnu mapu zobrazí (zobrazovanie sa vykonáva každých 20 ms) a agent vykonáva na základe diania na aktuálnej mape rozhodnutia do ďalšieho kroku. Keď agent vykoná rozhodnutie o akcii v nasledujúcom kroku, pošle túto informáciu naspäť prostrediu. Viac detailov o vizualizácii a komunikácii medzi komponentami sa nachádza v kapitole 3.1 a 3.2.

7.1.2 Prostredie

Prostredie má na starosti aj správu agentov. Má na starosti ich pridávanie a vymazávanie z mapy. Taktiež rieši kolízie agentov s inými agentami alebo so stenami. Kolízie rieši tak, že agent v danom kolíznom kroku simulácie počká (nepohne sa) a v nasle-

dujúcom kroku sa už prekážke vyhne. Funkcionalite prostredia sa podrobne venuje kapitola 3.10.

7.1.3 Agent

Jednotlivé agenty majú za úlohu simulovať správanie človeka - rozhodujú, čo vykonajú v nasledujúcom kroku. Pri rozhodovaní berú do úvahy pozície ostatných agentov aj steny. Celé rezhodovanie sa riadi stavovým automatom. Podľa aktuálneho stavu sa agent rozhodne, či sa pohne a ktorým smerom sa pohne. Jeho hlavným cieľom je dosiahnutie bezpečnostnej zóny. Na začiatku simulácie si každý agent nájde najbližší východ a snaží sa tam dostať. Funkcionalite agenta sa podrobne venuje kapitola 4.3.

7.1.4 Mapa

Dôležitou súčasťou simulácie je vhodná voľba reprezentácie mapy, resp. statických objektov, ktoré vstupujú do procesu simulácie. Objekty je možné do mapy pridávať vzhľadom na cieľový stav simulácie, ktorý chceme prispôsobiť reálnym podmienkam. Formát pre reprezentáciu mapy sme zvolili AutoCAD DXF. Dôvodom výberu je jednoduchá tvorba základných máp vzhľadom na testovanie dôležitých funkcionalít prototypu, rozsiahle možnosti konverzie z iných použiteľných formátov máp a jednoduchý prenos existujúcej funkcionality reprezentácie pre 3D formáty.

Mapa obsahuje 5 vrstiev, ktoré reprezentujú konkrétne typy objektov vystupujúcich v mape. Prvou sú steny budovy, v ktorých prebieha simulácia a teda ohraničujú možnosti pohybu agentov. Druhou sú prekážky v vnútri budovy, ktoré zabraňujú agentom v pohybe. Treťou vrstvou sú kontajnery pre počiatočné pozície agentov, v ktorých je počiatok realizácie v simulácii. Štvrtá vrstva obsahuje východy z budovy, ktoré sa snažia agenty dosiahnúť a teda byť úspešne evakuovaný. Poslednou vrstvou je navigačná sieť, ktorá slúži ako prostriedok pre plánovanie agentov v priestore. Na základe týchto informácií môžeme opísať simulačné prostredie a simulovať základné správanie agentov.