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

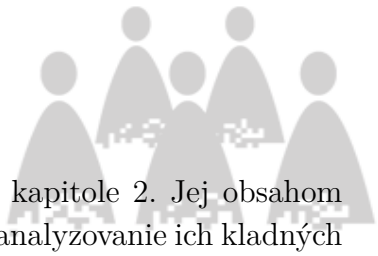
Dokument sa zaoberá problematikou simulovania davu a poskytuje komplexný objektívny pohľad na zadanú problematiku spolu s návrhom a realizáciou riešenia. Aplikácia sa zameriava na simuláciu davu v rámci interiéru a jej výsledkom je schopnosť simulácie stoviek až tisícov agentov umiestnených do prostredia 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, 6, 7, 8, 9, 10 a 11 obsahujú popis úloh riešených v jednotlivých šprintoch. Dokumentované sú predovšetkým úlohy, ktoré svojím charakterom a zložitosťou vyžadujú dodatočnú dokumentáciu. Implementačné úlohy sú popísané z hľadiska analýzy problému, návrhu riešenia, implementácie a testovania.

Kapitola 12 sa zameriava na hlavné komponenty produktu z hľadiska architektúry.

Zhrnutie projektu je predmetom kapitoly 13. Projekt je sumarizovaný z mnohých uhľov pohľadu a načrtnuté sú tu aj možné alternatívy pre budúce smerovanie. Opisom používania programu z pohľadu používateľa aj vývojára sa zaoberajú prílohy A a B.

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

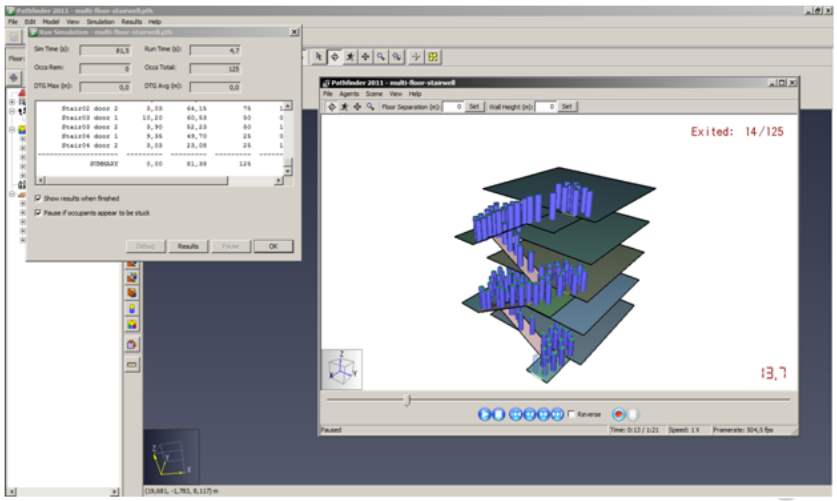
Komunikácia medzi simulačným prostredím a agentmi je možné rozdeliť na dve časti. Prvá sa týka toho ako prostredie vníma agentov a druhá ako agenty vnímajú prostredie. Simulačné prostredie sleduje pohyb agentov priebehom simulácie a poskytuje informácie ohľadom jednotlivých agentov (napr. pozícia). Naopak, 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. 8.4). 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

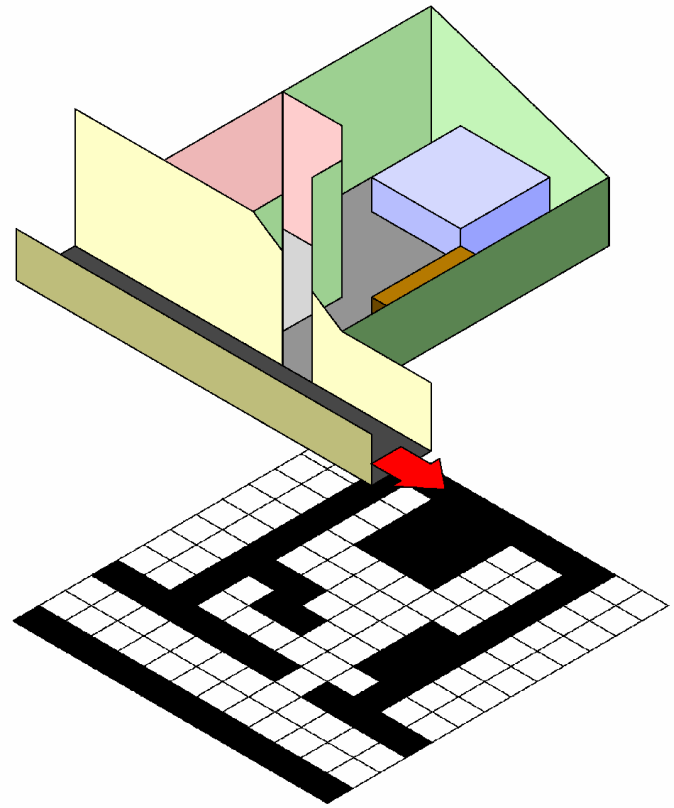
2.2 PedGo

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.

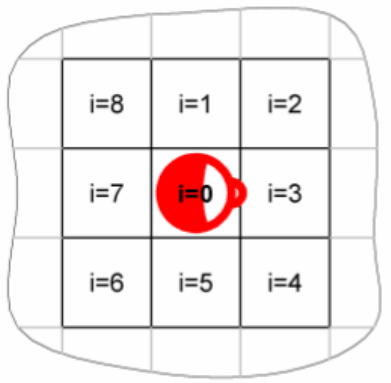
2.2.3 Rozhodovanie agenta

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.

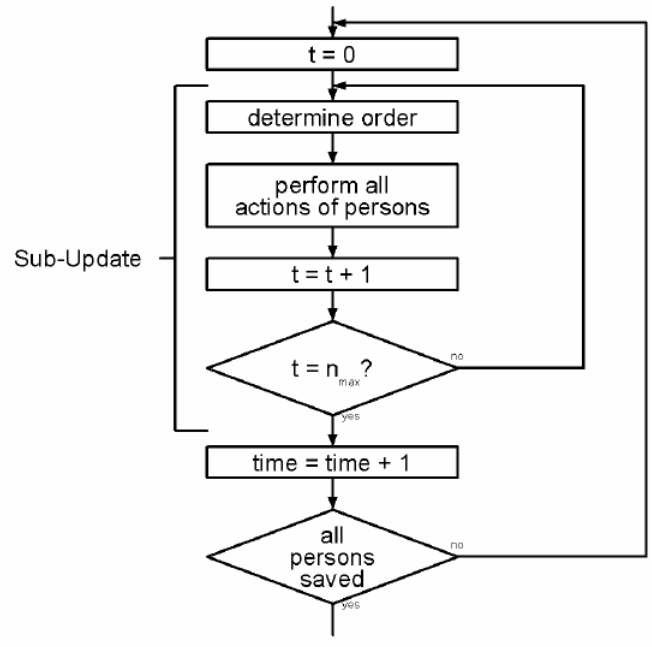
Po vypočítaní pravdepodobnosti výberu smeru je táto hodnota vynásobená hodnotou náhodnosti.



Obr. 2.3: Znázornenie agenta a možnosti jeho voľby

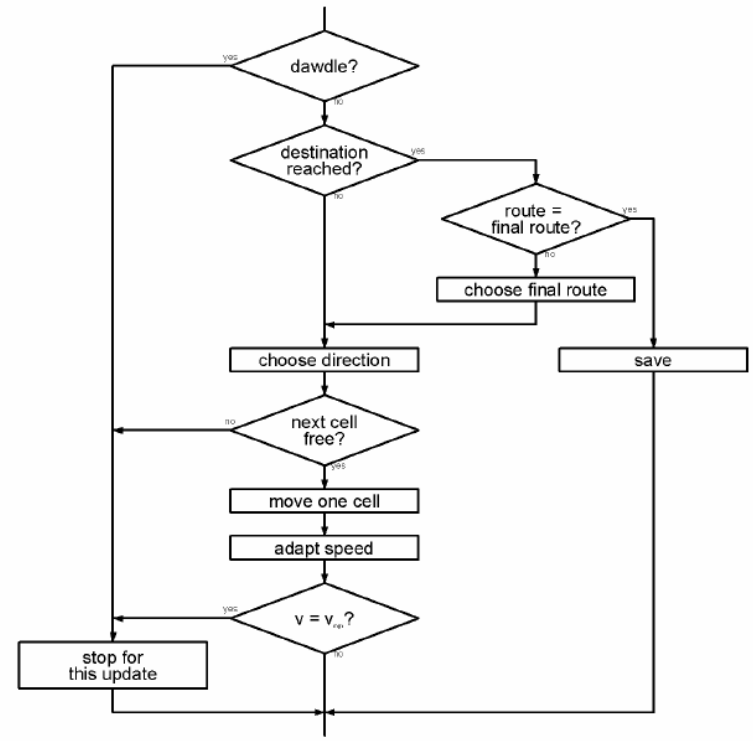
2.2.4 Pohyb a aktualizácia

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

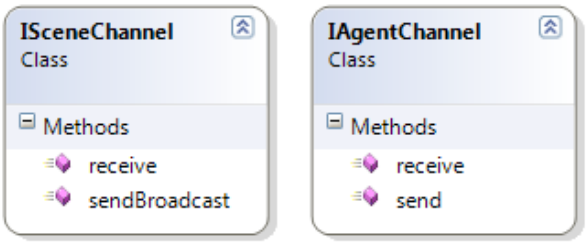
Analýza problému

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.

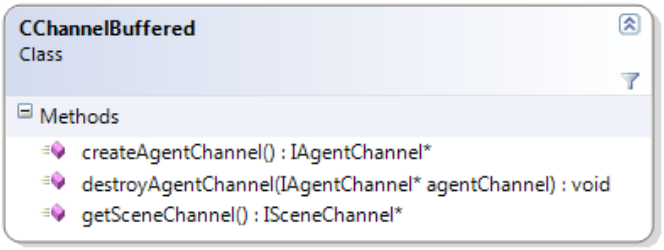
Opis implementácie



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

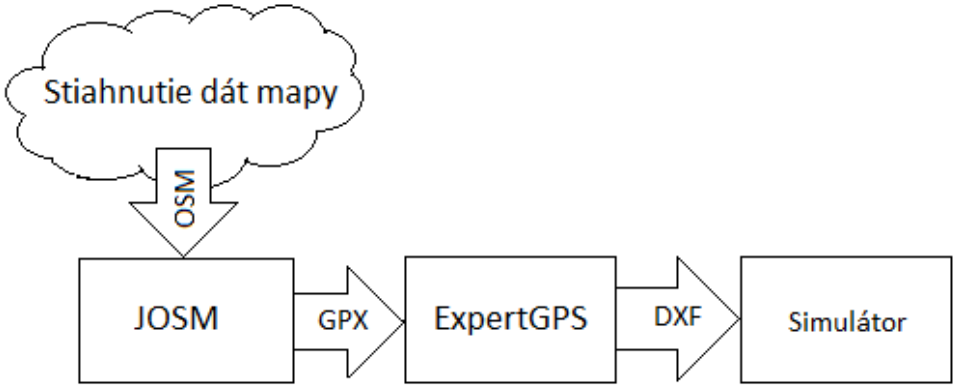
Analýza problému
Projekt Virtuálna FIIT bol riešený v ročníku 2010/2011 tímom Mgr. Aleny Kovárovej. V rámci tohto projektu ako výsledok bola vytvorená 3d web prezentácia školy. Model školy je vytvorený v nástroji 3ds Max. Tento formát môže byť výhodný pre náš tím, lebo sa dá konvertovať do rôznych formátov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

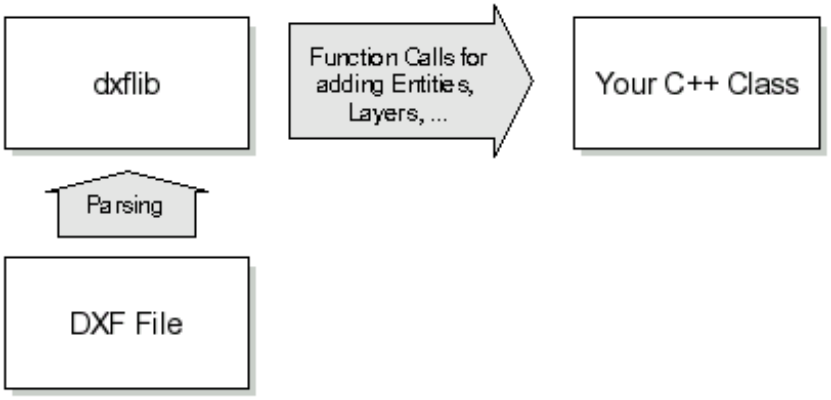
- Steny – reprezentujú ohraničenie budovy, v ktorej sa má simulácia odohrávať.
- Prekážky – reprezentujú objekty v budove, cez ktoré nie je možné prejsť.
- Štarty (angl. spawns) – sú plochy, v ktorých sa generujú agenty na začiatku simulácie.
- Dvere – reprezentujú oblasti medzi miestnosťami, ktoré uľahčujú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

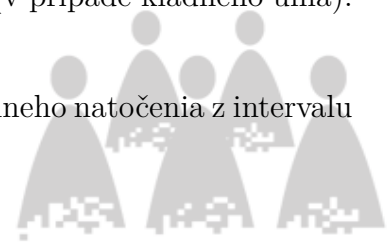
Analýza problému
Základným problémom bolo navrhnúť protokol, pomocou ktorého si budú môcť prostredie a agenty vymieňať informácie. Agent potrebuje vedieť, kde v prostredí sa nachádza a kde sa nachádzajú ostatné agenty. Prostredie zase potrebuje vedieť, akú akciu sa snaží agent vykonať. Základné informácie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

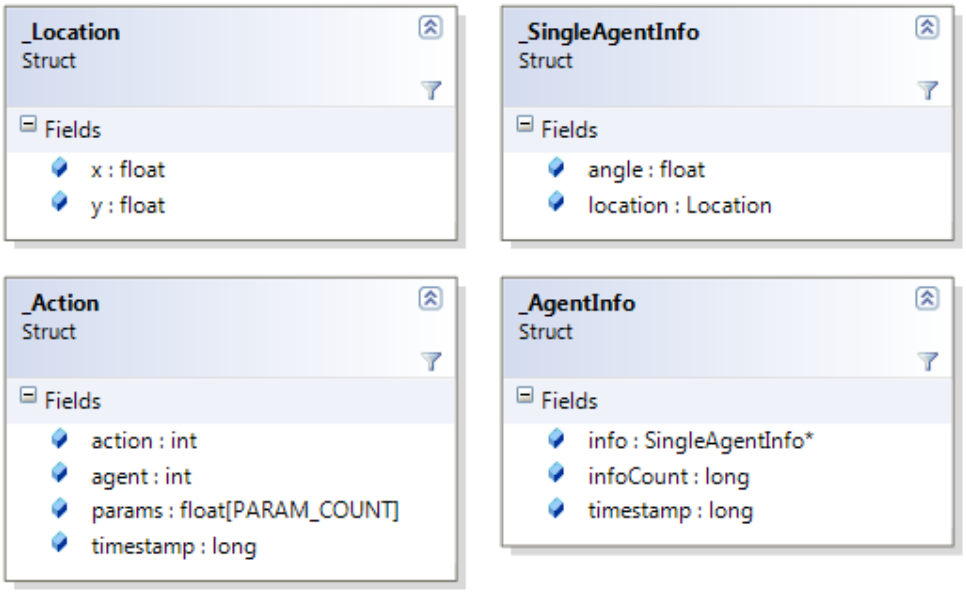
Návrh riešenia
Informácia o pozícii všetkých agentov sa dá jednoducho zaznamenať pomocou poľa. Prvkami poľa budú dátové štruktúry obsahujúce práve informácie o polohe a natočení agenta. K týmto informáciám je pridaný aktuálny čas kedy bola informácia zaslaná agentom.

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.

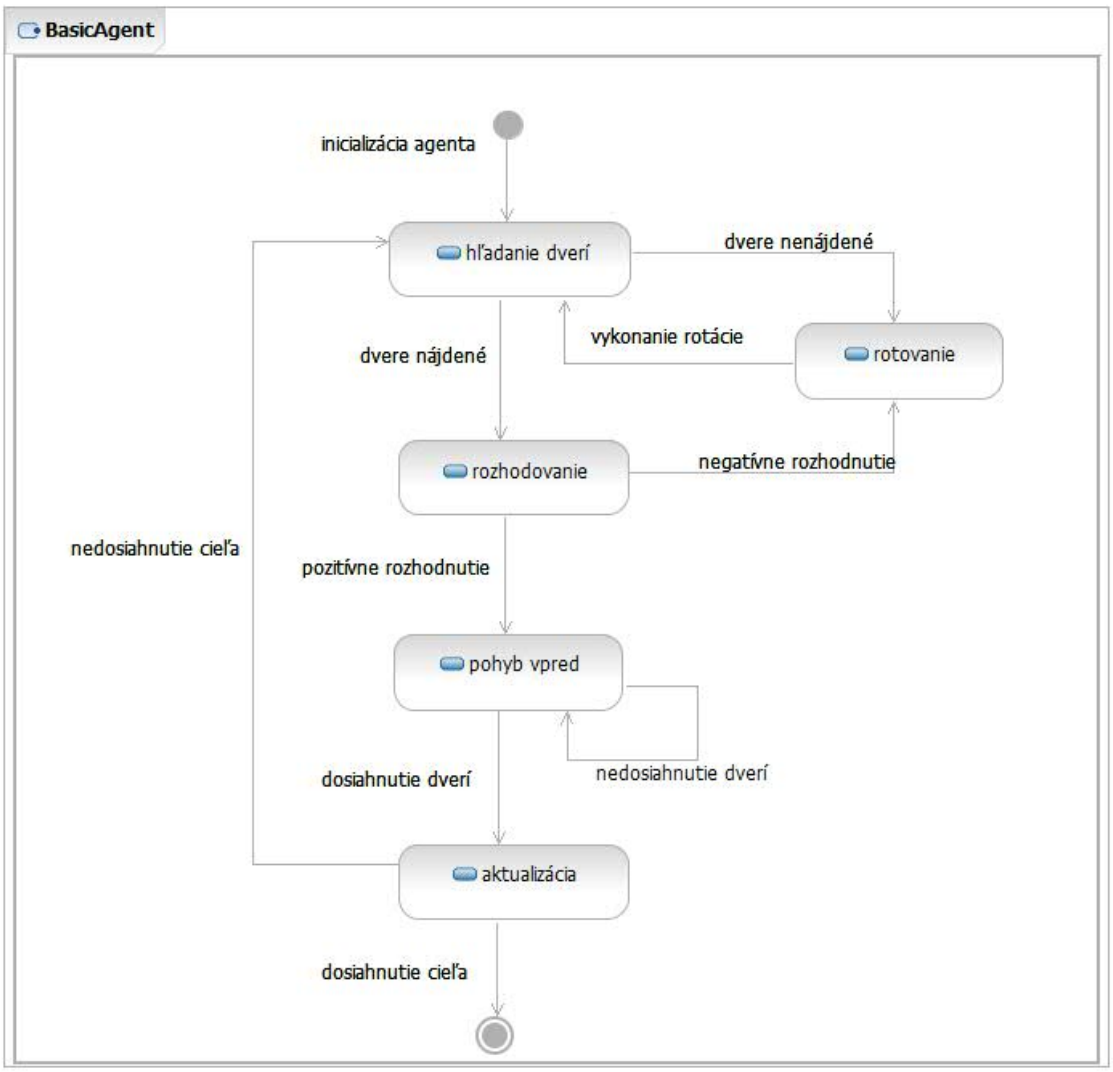
Opis implementácie
Pri vykonávaní procesu je potrebné najprv vypočítať súradnice stredu dverí, nakoľko sú reprezentované ako obdĺžnik. Potom sa vzdialenosť agenta od dverí počíta ako dĺžka spojnice aktuálnej polohy agenta a vypočítaného stredu obdĺžníka reprezentujúceho dvere. Pri počítaní vzdialeností 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

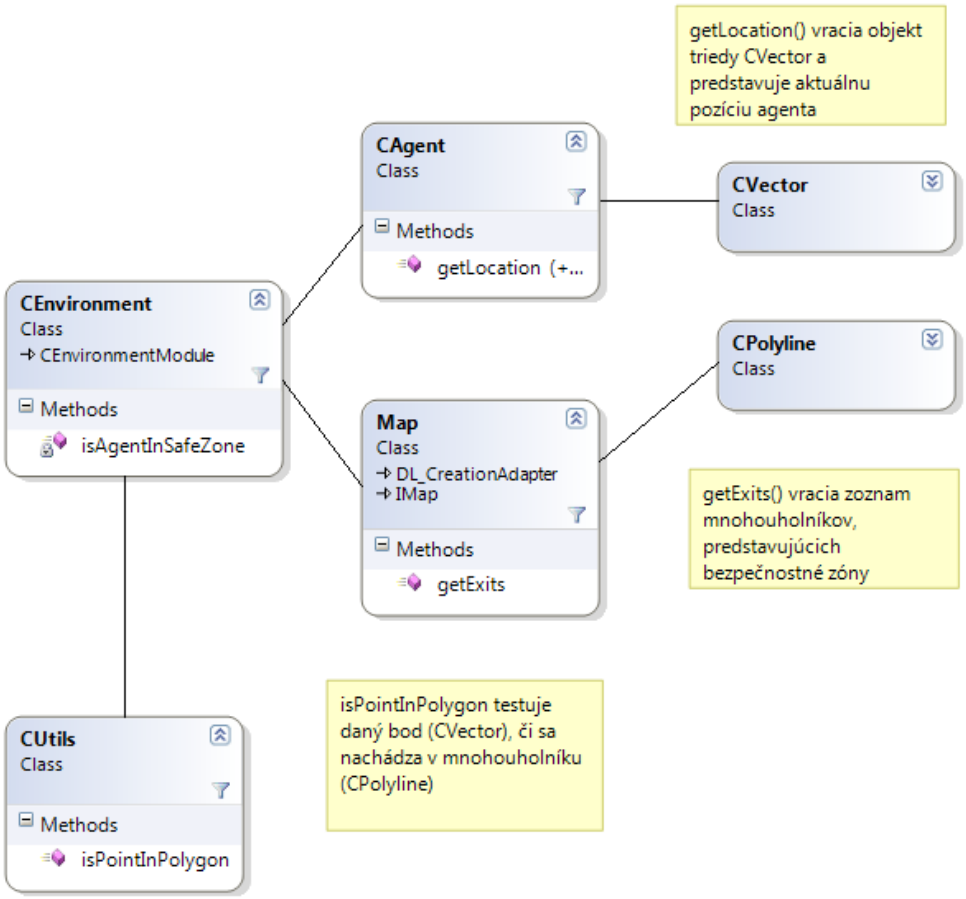
spracovávaných agentov. Aby mohla aplikácia zistiť, či môže daného agenta prestať manažovať, je potrebné zistiť, či sa nachádza v jednej z bezpečnostných zón.

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

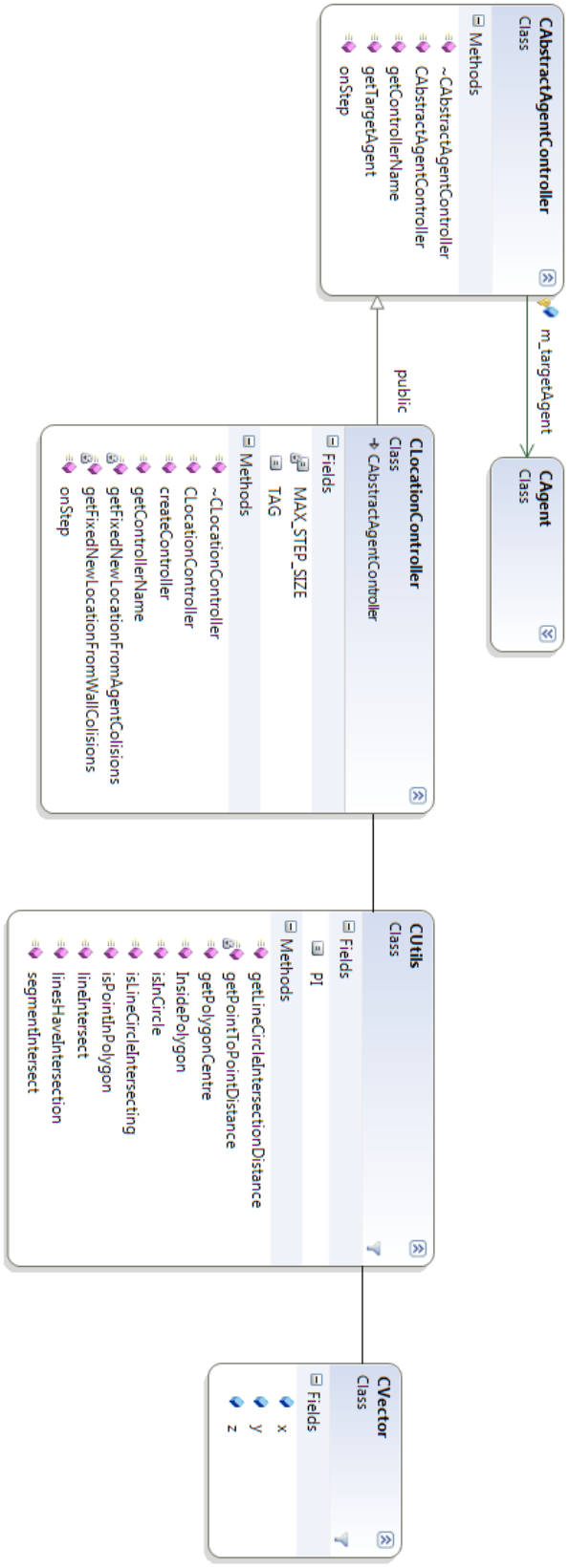
Návrh riešenia

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

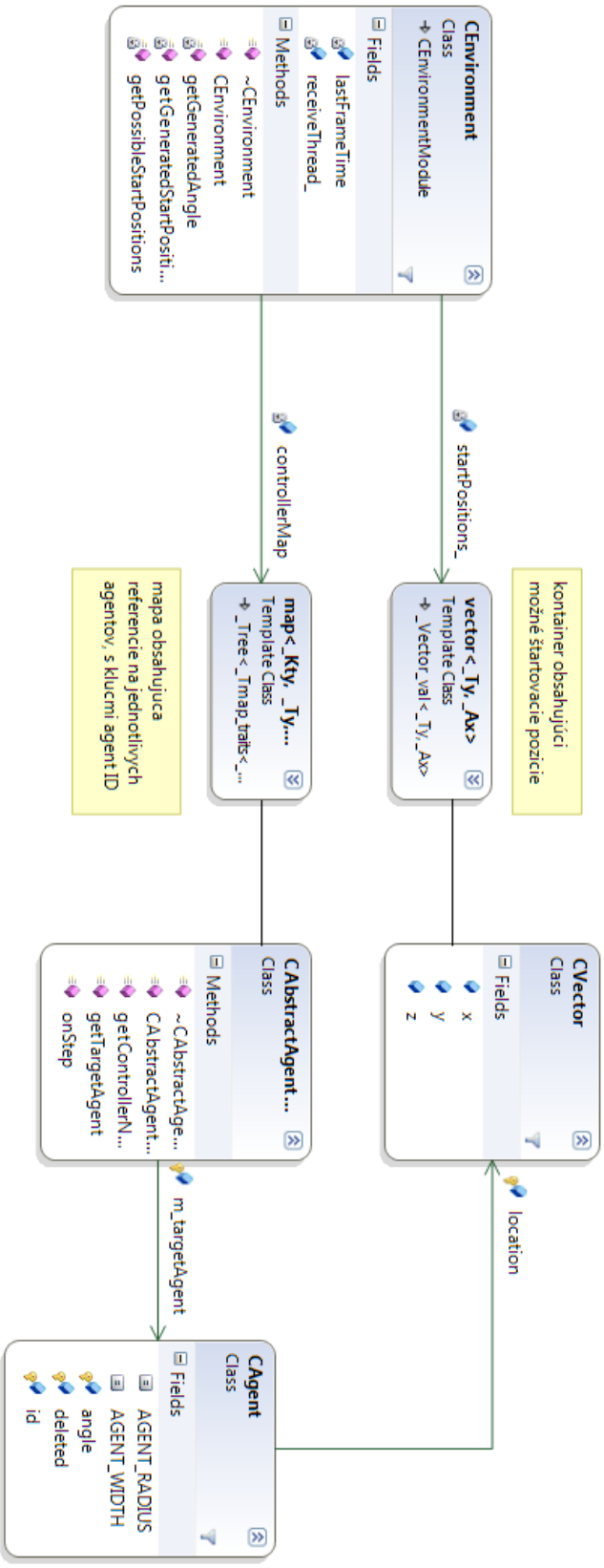
Generovanie agentov do spanov je zatiaľ len v projekte *CrowdSim*. Túto funkcionalitu je potrebné prepísať aj do separátneho projektu Enviroment.

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

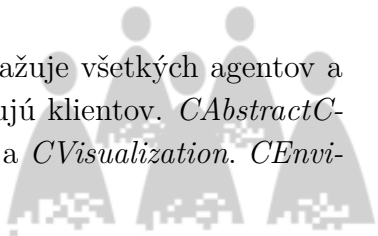
Modul, ktorého úloha je simulovať správanie agentov.

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

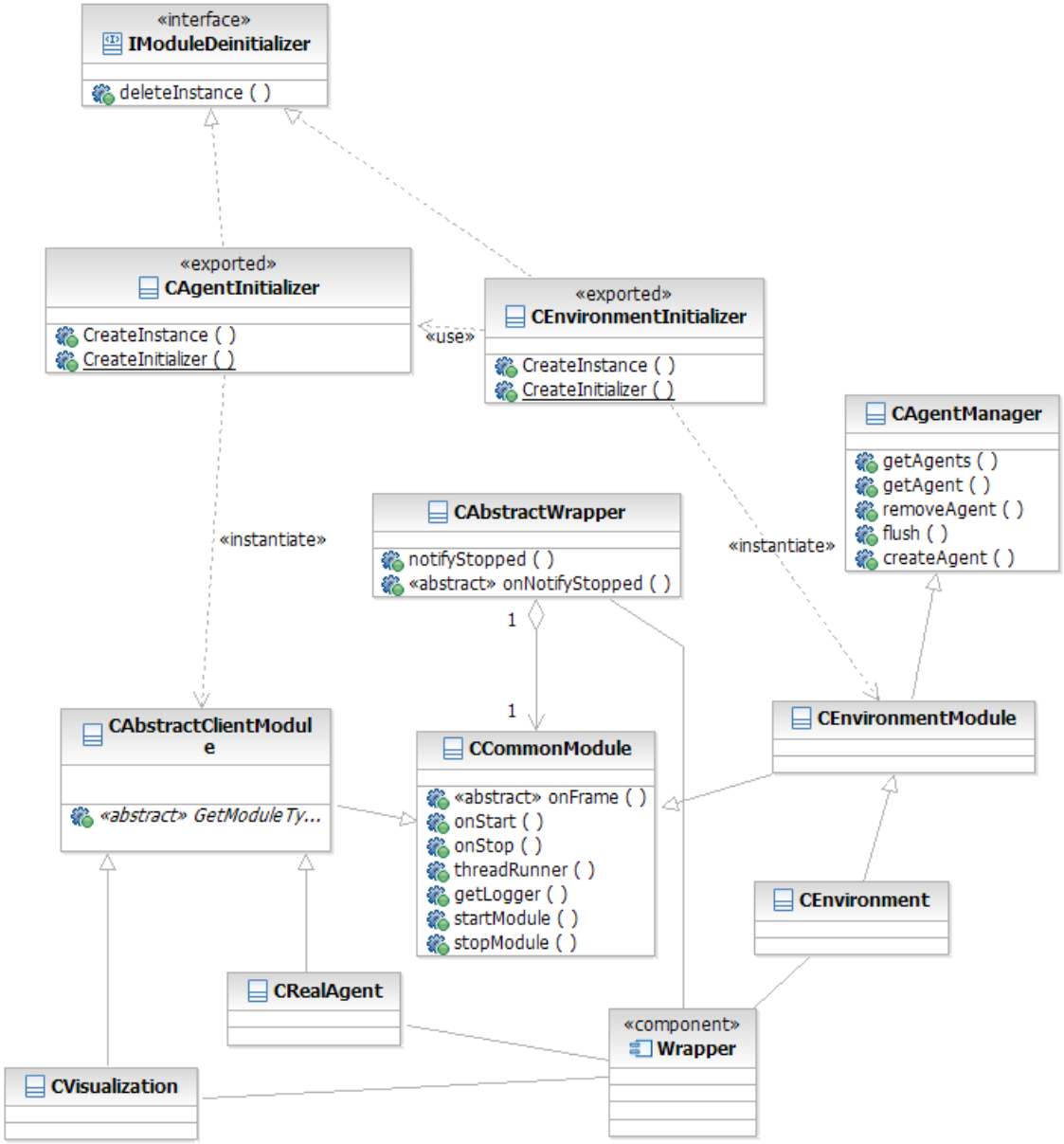
CAgentInitializer a CEnvironmentInitializer

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.

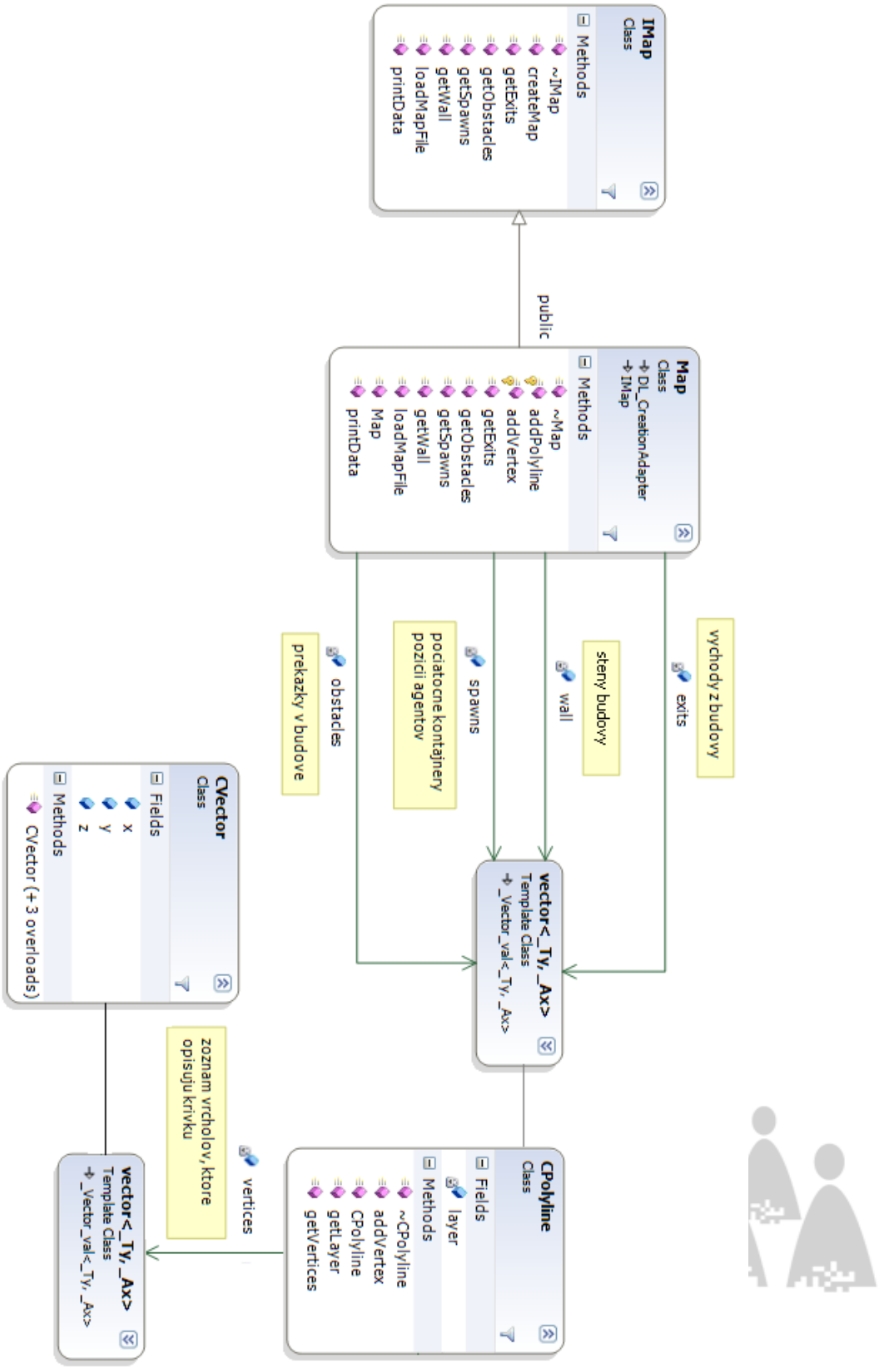
Keďže pre beh aplikácie stačí načítanie len jednej mapy, tak nám stačí spravovať len jednu inštanciu mapy. Toto zabezpečíme použitím návrhového vzoru Abstract Factory, ktorý pri prvom volaní vytvorí pri inicializácii mapu a následne je mapa pri ďalších volaniach vrátená. Týmto spôsobom zabezpečíme optimalizovaný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

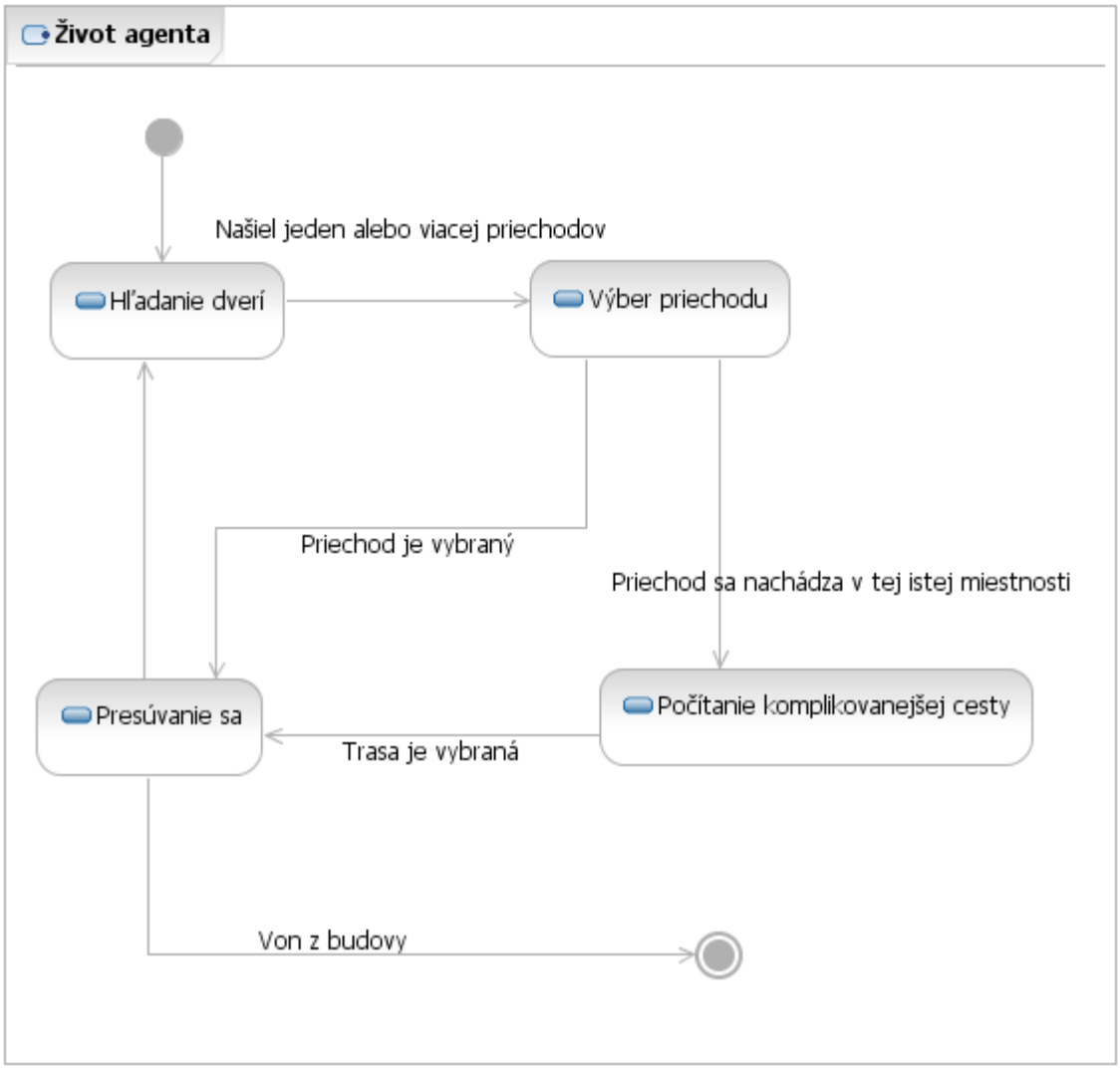
5.8 Návrh plánovania rozhraní pohybu agentov

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

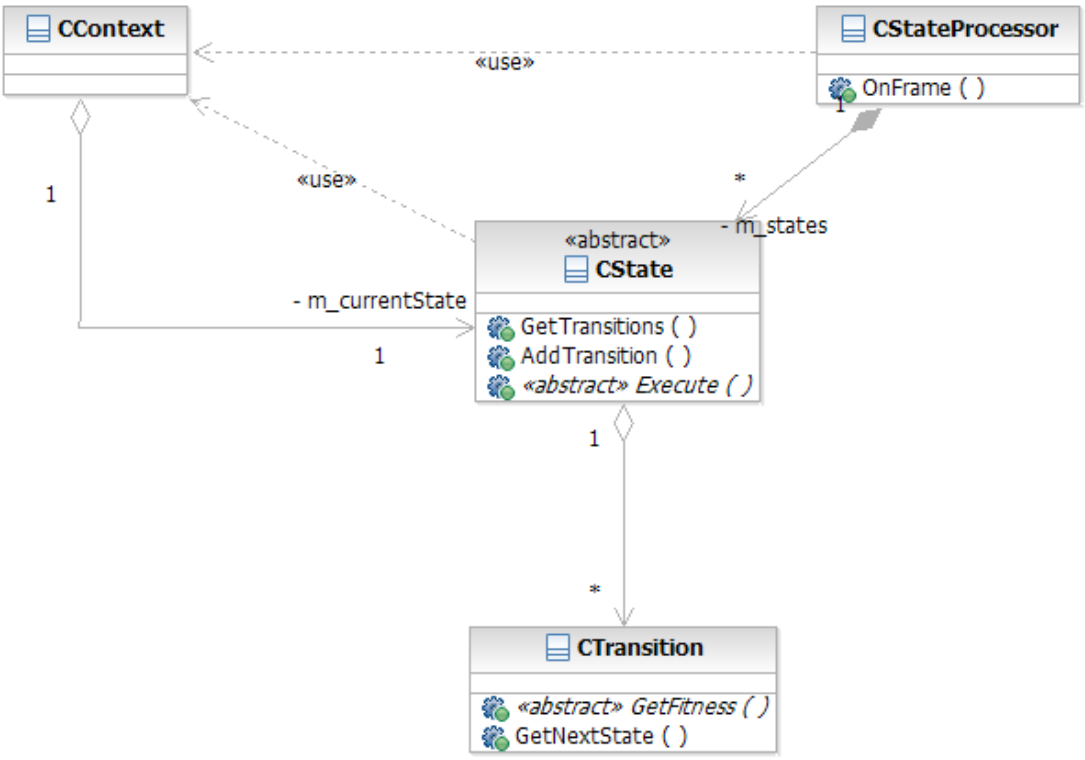
Navrhnutý stavový automat je na obr. 5.6.



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

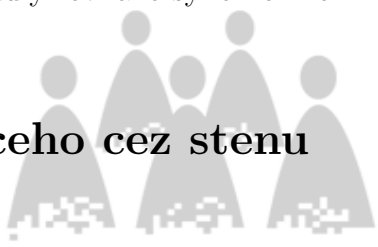
Prostredie musí medzi jednotlivými krokmi čakať. Doba čakania by mala byt primeraná, tak aby sa za 1 sekundu stihlo prostredie urobiť 20 simulačných krokov (20 FPS).

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

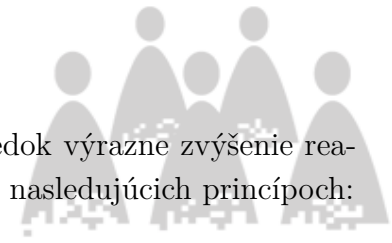
Kapitola 6

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

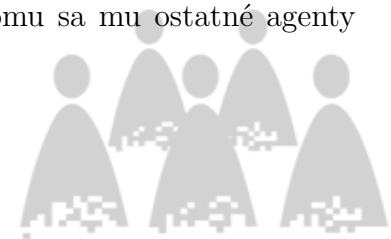
Rýchlosť je ovplyvnená nie len povrchom prostredia, ale najmä hustotou iných agentov. Rýchlosť sa zmenšuje, ak ide agent proti prúdu ostatných agentov a nemení sa, ak ide po prúde. Inými slovami, rýchlosť je density-dependent. Závisí na hustote. Rýchlosť je vyjadrená vektorom – a teda vyjadruje zároveň 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 8.1

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

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 8.2)

$$\int_P C ds$$

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 8.2.

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.

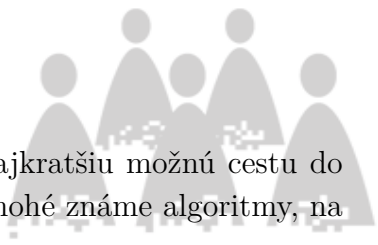
Následne je vektor rýchlosti každého agenta upravený podľa tohto sumárneho vektora, čim sa simuluje jednoliatosť kvapaliny.

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

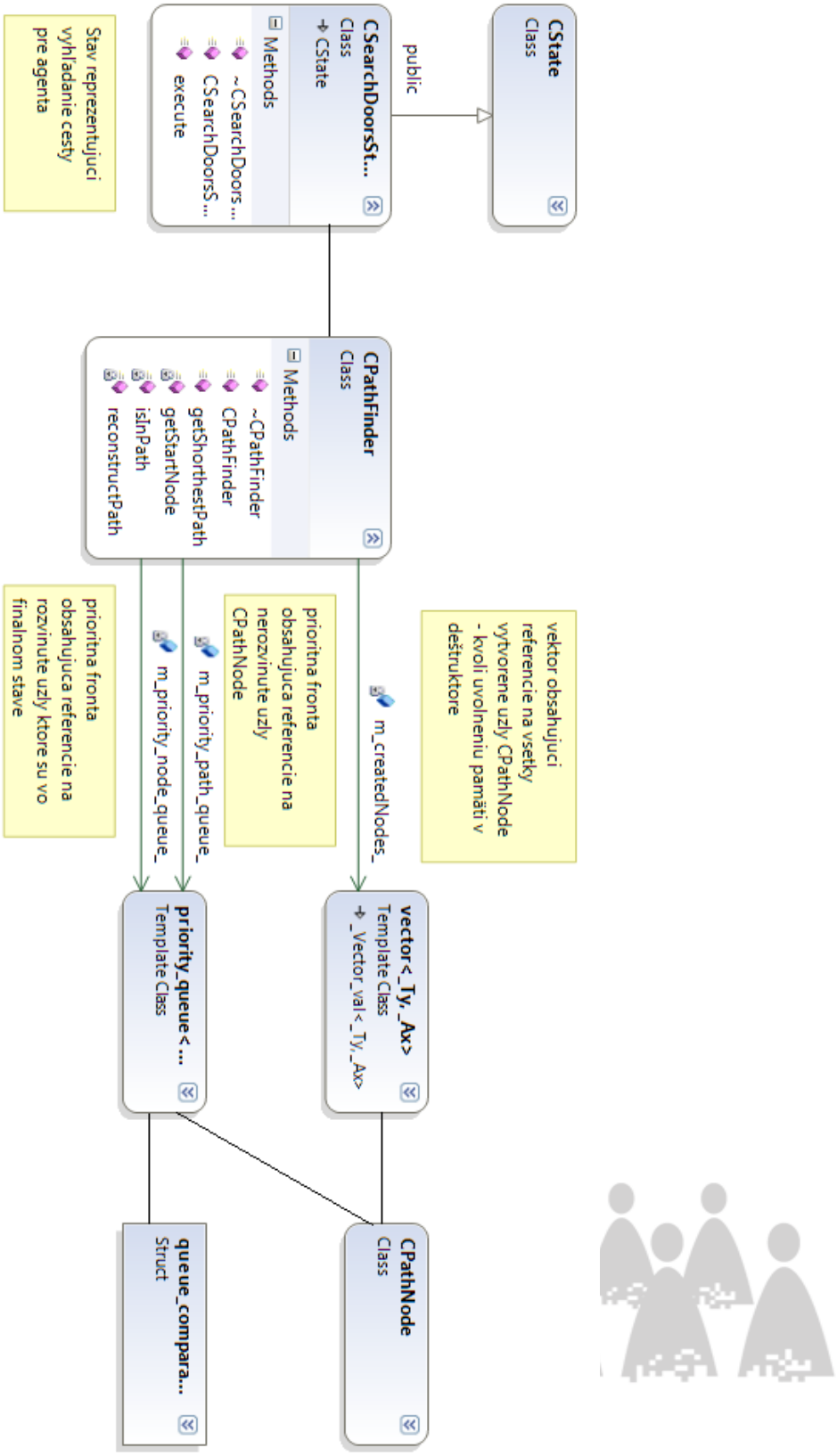
Na základe načítanej NavigationMesh je potrebné nájsť najkratšiu možnú cestu do cieľa (východ z budovy). Prehľadávaniu grafu sa venujú mnohé známe algoritmy, na účel problému bol zvolený algoritmus A*.

Návrh riešenia

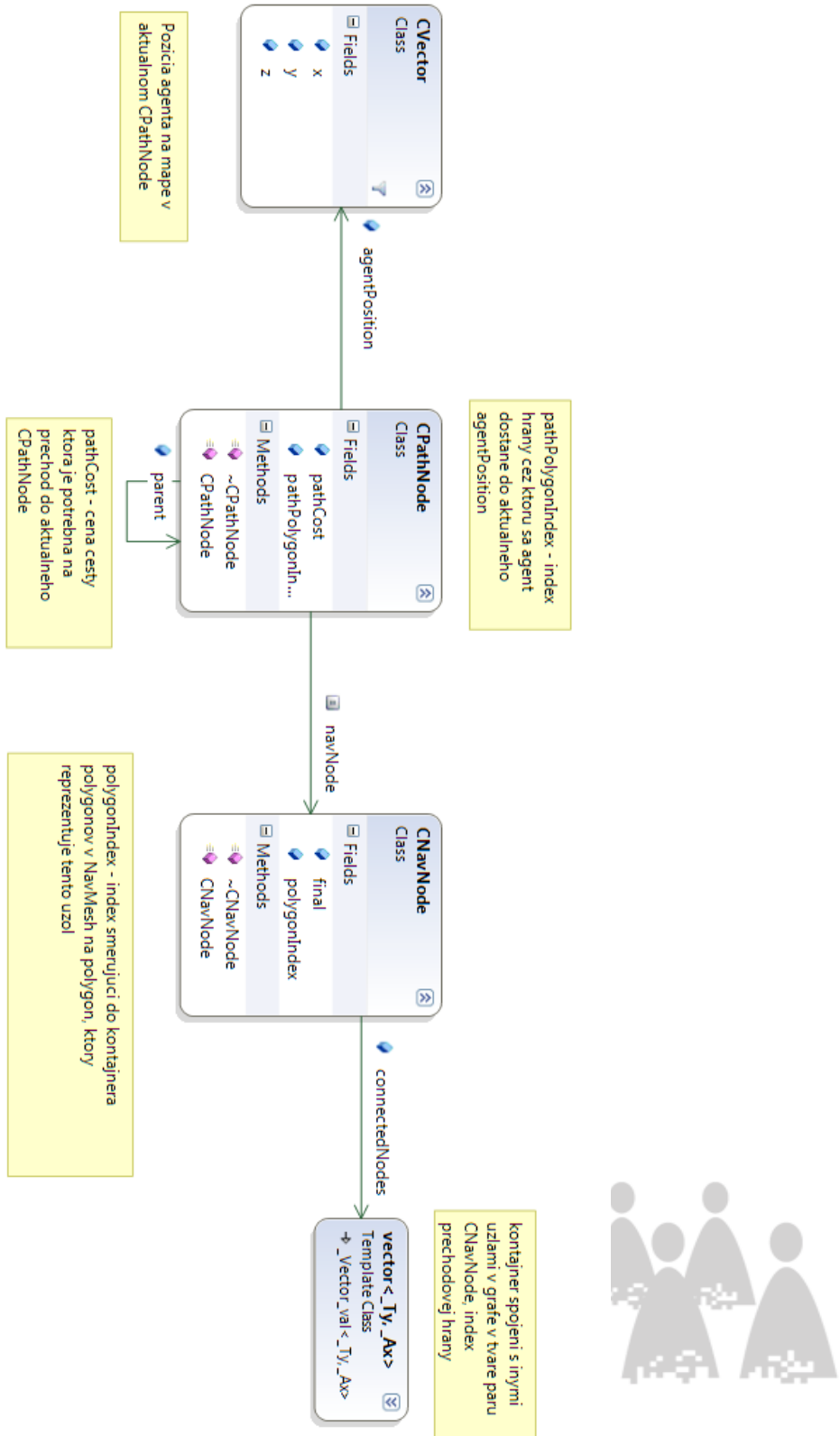
Agent pozná celú mapu budovy. Preto je možné ľahko vyhľadať najkratšiu možnú cestu z budovy. V stave *searchDoor* agent vyhľadá pomocou A* algoritmu najkratšiu možnú cestu a vráti je do kontext-u v tvare kontainera prechodových uzlov s zvolenými prechodovými hranami.

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

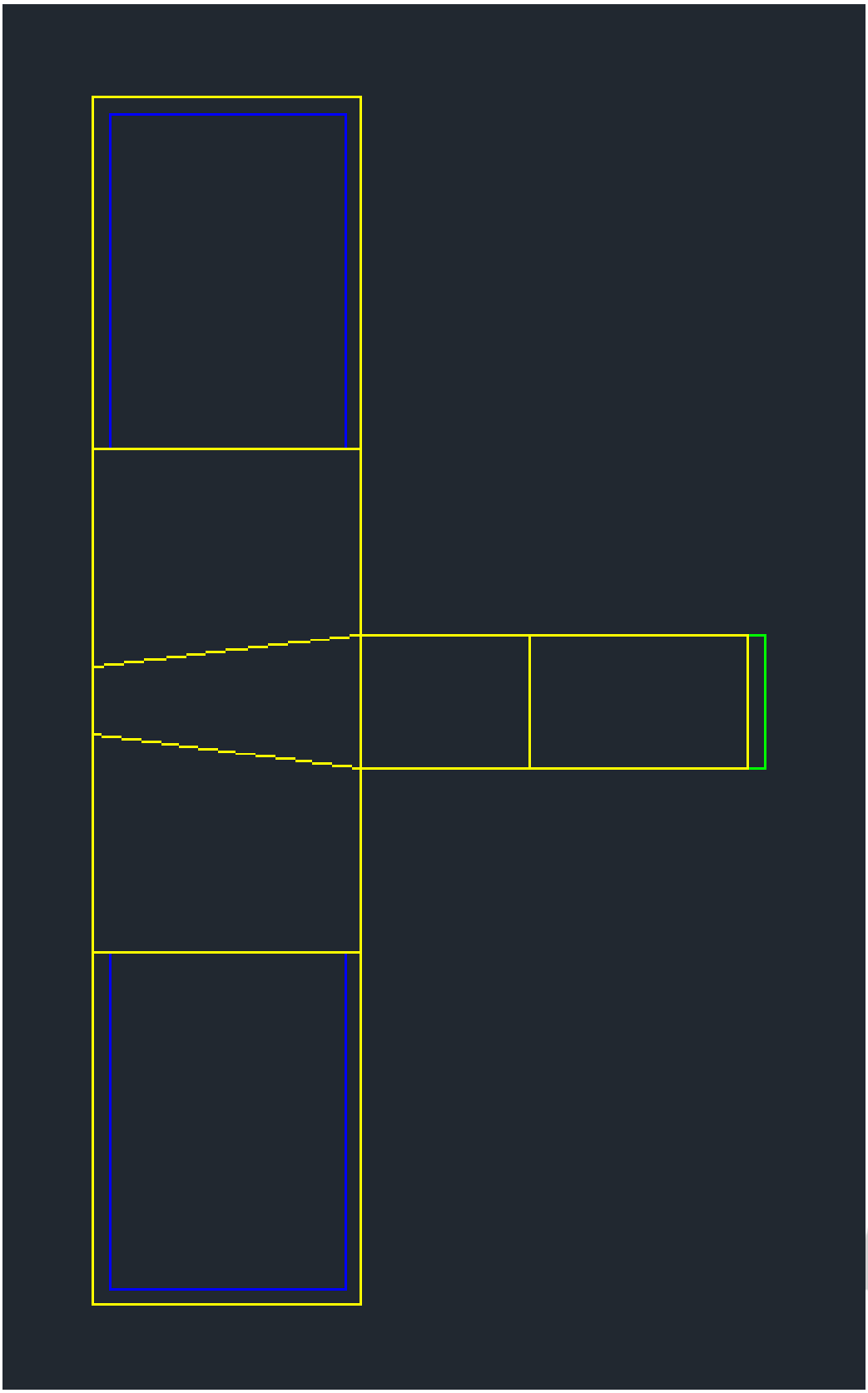
Ideálnym riešením je použitie NavigationMesh, ktorá predstavuje navigačnú sieť pre konkrétnu mapu. Pod navigačnou sieťou sa rozumie vrstva obsahujúca sadu prepojených objektov, ktorých cieľom je reprezentácia ciest. Takýmto spôsobom môžeme ohraničiť oblasti, v ktorých je možný pohyb agentov.

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.

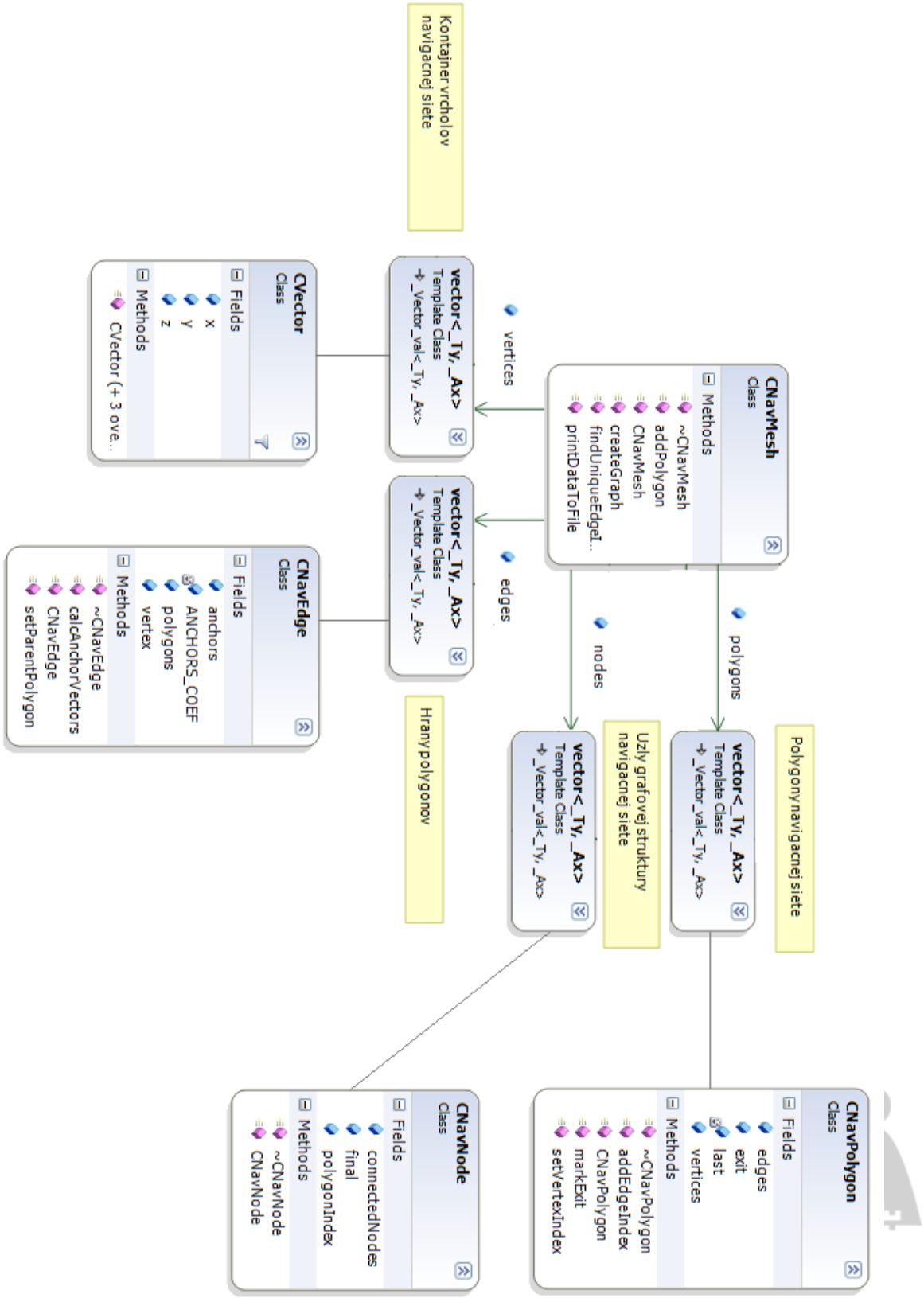
Opis implementácie

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.

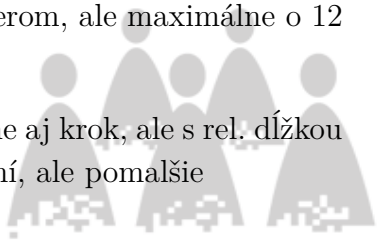
Separácia agentov jeden od druhého a vyhýbanie sa prekážkam, bude realizovaná úpravou vektora smerujúceho do ich cieľa. Samotná veľkosť úpravy bude váhovaná, ale vo všeobecnosti platí, že ak sa agent dostane do hraničnej blízkosti iného agenta alebo steny, bude sa snažiť od neho vzdialiť. Čím bližšie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

Šprint#5

ID	Názov úlohy	Zodpovedný
1	Analýza správania agentov s použitím existujúcich technológií	Lukáš Pavlech

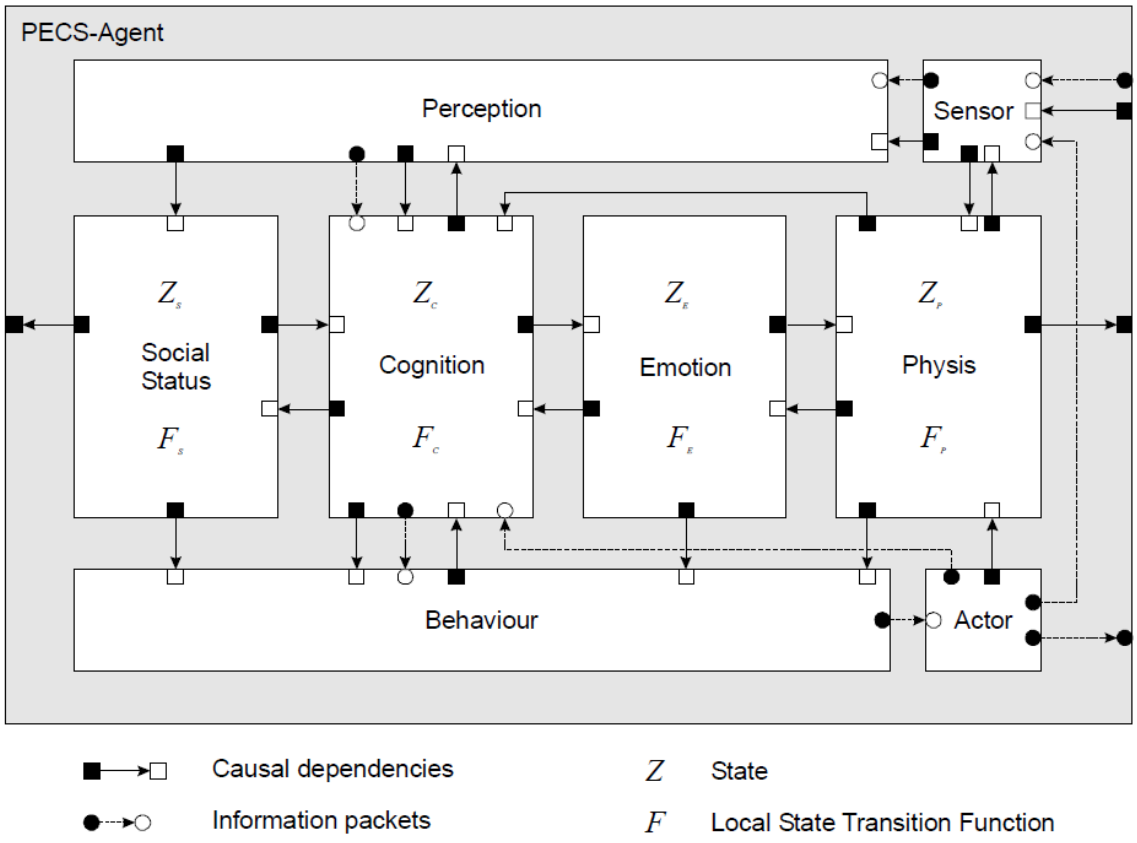
7.1 Analýza správania agentov s použitím existujúcich technológií

Analýza problému

Pri analýze problému boli preštudované modely ako napr. PECS a BDI. Tieto modely sa zaoberajú príliš všeobecnou reprezentáciou ľudského správania, pre použitie v tímovom projekte nie sú vhodné. Poskytujú len návod ako reprezentovať ľudské správanie, ktorého reprezentácia je zobrazená na obrázku číslo 7.1.

Návrh riešenia

Pre potreby tímového projektu bude ľudské správanie ovplyvňované pomocou hodnoty strachu susediacich agentov alebo hodnoty strachu miestnosti, v ktorej sa nachádza (navigation mesh). Na začiatku bude agent v stave pokoja, kedy bude kľudne chodiť po miestosti, poprípade medzi miestnosťami. Pri interakcií agenta s vystrašeným agentom, alebo s miestnosťou, v ktorej sa nachádza strach, agentovi stúpne hodnota strachu a prejde do stavu evakuácie. V stave evakuácie rýchlosť agenta bude závisieť od hodnoty strachu - čím vyššia hodnota strachu, tým väčšia rýchlosť.



Obr. 7.1: PECS model)

Kapitola 8

Šprint#6

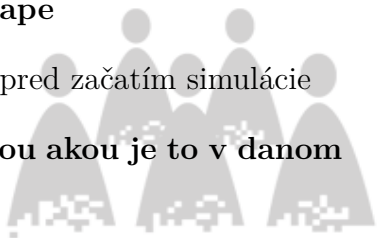
ID	Názov úlohy	Zodpovedný
1	Aplikovanie hustoty na model	Adam pomothy, Michal Fornádeľ
2	Hľadanie Q-learning	Lukáš Pavlech
3	Doplnenie stavu kľudu pre agentov	Lukáš Pavlech
4	Vytvorenie veľkej mapy	Marek Hlaváč
5	Vytvorenie MPI paralelizácie	Martin Košický
6	Optimalizácia steering behaviors	Daniel Petráš
7	Vytvorenie konfiguračného súboru	Daniel Petráš
8	Vytvorenie dotazníka do TP cupu	Lukáš Pavlech

8.1 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 ostaný agenti 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 8.1

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

Obr. 8.1: 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 pomcoou integrálu (výraz 8.2)

$$\int_P C ds$$

Obr. 8.2: 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 klesá hodnota funkcie 8.2.

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.

8.2 Hľadanie Q-learning

Analýza problému

Doterajšia implementácia A* algoritmu na nájdenie najkratšej cesty z danej pozície do cieľa (východ) sa ukázala byť neefektívna pri veľkej mape a to hlavne kvôli faktu, že každý agent musel pri hľadaní cesty vyskúšať mnoho ciest. Preto bolo potrebné vytvoriť nový systém vyhľadania cesty.

Návrh riešenia

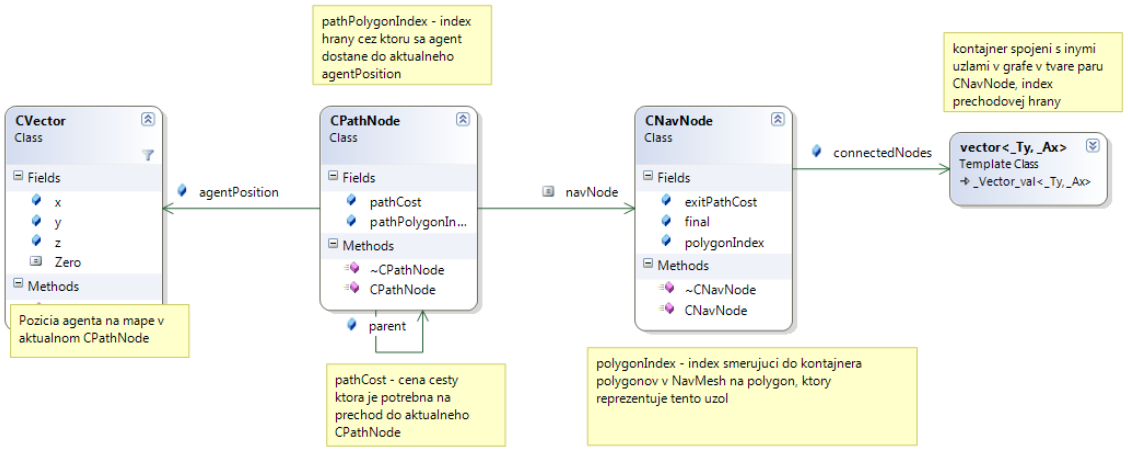
Pre dosiahnutie čo najefektívnejších výsledkov vyhľadania cesty sme sa rozhodli použiť modifikáciu hľadania pomocou Q-learningu. Prepočítavanie cien cesty začína v cieľových úlohách, odkiaľ sa šíry vlna. Každému zo jeho susedov sa priradí hodnota ceny cesty do vrchola - ak vrchol už obsahuje cenu cesty, vyberie sa tá menšia.

Opis implementácie

Pre každý uzol navigačnej mriežky sa pridá nový parameter - cena cesty do cieľa (východ). Pri spustení agent modulu sa prepočíta celá navigačná mriežka a každému vrcholu sa priradí hodnota najkratšej vzdialenosti do cieľa. Vďaka tejto informácií agent v stave searchDoorState veľmi rýchlo zistí najkratšiu cestu do cieľa.

Obrázok 8.3 zobrazue grafovú štrukturu použitú pri hľadaní cesty pre jedného agenta

(hľadanie sa najprv skúša pomocou q-learningu, po implementácii strachu a požiaru sa pre vyhľadanie cesty pri neúspešnom hľadaní q-learningom použije A* star algoritmus.



Obr. 8.3: Grafová štruktúra pri hľadaní cesty

Obrázok 8.4 zobrazuje class diagram triedy PathFinder, ktorú pri inicializácii agent modulu volá trieda RealAgent. Pri tejto inicializácii sa prepočíta mapa vzdialenosti jednotlivych navigačných mriežok. Agent pri vyhľadávaní cesty zo stavu CSearchDoorState, ktorá dedi od triedy CState vyhľadá cestu pomocou triedy PathFinder cez metódu getShorthestPath.

8.3 Doplnenie stavu kľudu pre agentov

Analýza problému

Vykonaná v úlohe Analýza správania agentov s použitím existujúcich technológií.

Návrh riešenia

Pre doplnenie stavu kľudu pre agentov je potrebné vytvorenie nového stavu do stavového automatu. Tento stav bude reprezentovať agenta v kľudnom stave - agent sa bude prechádzať k náhodne zvoleným bodom.

Opis implementácie

Nový stav bude rovnako ako už vytvorené stavy stavového automatu, dediť od triedy CState. Pri aktivácií stavu sa pre agenta vygeneruje bod cesty - tento bod sa nachádza

buď v aktulnej navigačnej mriežke alebo v niektorej z jeho susedov. Následne sa agent snaží k danému bodu dostať - ak sa tak stane, pre agenta sa vygeneruje znovu nový bod.

8.4 Vytvorenie veľkej mapy

Analýza problému

Po vytvorení základného prototypu simulácie je potrebné otestovať správanie simulácie pri väčšom zaťažení s väčším počtom simulovaných agentov. Na základe tohto problému je nutné vytvorenie veľkej mapy, ktorá by poskytovala priestor pre pohyb agentov. Mapa by mala byť realistická, čo znamená, že prvky, ktoré obsahuje, korešpondujú s realitou. Taktiež je nutné, aby v nej bolo možné odsimulovať špeciálne situácie, ktoré by pomohli odhaliť potenciálne chyby pri výpočte v jednotlivých krokoch simulácie.

Návrh riešenia

Riešenie je navrhnuté na základe pôdorysu existujúcej budovy, ktorá bola náhodne vyhľadaná. Avšak, aby bolo zabezpečené komplexné simulovanie, tak je nutné budovu mierne upraviť so zámerom nahradenia nevhodných oblastí pôdorysu za interiéry, ktoré budú testovať špecifické situácie. Vzhľadom na rozsiahlu veľkosť mapy bol model vytvorený s kolmými stenami, aby sa predišlo nežiaducim efektom pri výpočte a testovalo sa len základné správanie davu a izolovaných skupiniek.

Opis implementácie

Vytvorená veľká mapa obsahuje 286 navigačných mriežok, ktoré reprezentujú možné oblasti pohybu agenta v mape. Umožňujú tak plánovať a zároveň testovať dlhodobé plánovanie agenta, pričom je možné simulovať jeho komplexnejšie správanie. Mapa je namodelovaná spôsobom, aby zodpovedal reálnej situácií. Obsahuje množstvo malých a veľkých miestností, chodieb, priestranstiev, dva východy a špecifické interiéry ako napríklad úzke priechody, prekážky a viacej možností únikových ciest (viď Obr. 8.5).

8.5 Vytvorenie MPI paralelizácie

Analýza problému

Architektúra aplikácie bola navrhnutá za účelom vytvorenia samostatných modulov. Vďaka modulárnosti je možné každý komponent spustiť ako samostatný proces využiť

tak možnosti distribuovaného počítania, pričom je potrebné navrhnúť aj spôsob, ako tieto komponenty medzi sebou komunikujú. Vzájomnú výmenu správ rieši použitie technológie MPI (angl. *Message Passing Interface*).

Návrh riešenia

Pre vytvorenie MPI Wrapper-a je potrebné napísať implementácie pre inicializátory, ktoré budú dodané do prostredia a budú vracať agent modul, ktorý pracuje ako proxy spojenie medzi prostredím a agentom. Pre tento účel budú vytvorené nasledovné triedy (diagram tried je znázornený na obr. 8.6):

CEmptyInitializer

- metóda createInstance vyrobí novú inštanciu triedy
- metóda deleteIeInstance vymaže vyrobenú inštanciu CDummyAgentModule

CDummyAgentModule

Táto trieda preposiela volania na proces, kde beží agent modul.

CEnvironmentWrapper

Štandardný spúšťač modulu, kde beží prostredie.

CVisualWrapper

Štandardný spúšťač vizualizácie.

CommunicationLayer

Táto trieda ma za úlohu zabezpečovať spojenie medzi procesmi.

Opis implementácie

Na komunikáciu komponentov je použitá technológia *MPICH2*, ktorá predstavuje implementáciu protokolu MPI a boost::mpi C++ wrapper-a, ktorý umožnuje objektovo orientovaný prístup k MPI volaniam. *MPICH2* nie je bezpečný pre prístup z viacerých vlákien. Z toho dôvodu, sa všetky posielania vykonávajú v jednom vlákne, čo vedie k spomaľovaniu pre sústavné súperenie o zámok.

8.6 Optimalizácia steering behaviors

Analýza problému

Pri výpočte separácie agentov od ostatných agentov sa porovnávajú vzdialenosti agenta od všetkých ostatných agentov. Časová náročnosť tohto algoritmu pre všetkých agentov je teda $O(n^2)$. Aby sme túto náročnosť optimalizovali, nesmieme porovnávať pozície všetkých agentov, ale iba tých v bezprostrednom okolí agenta.

Návrh riešenia

Je potrebné implementovať dátovú štruktúru grid (mriežku), ktorá nám pre agenta vráti jeho pozíciu v grid-e. Túto pozíciu budeme následne porovnávať s pozíciou ostatných agentov v tomto grid-e. Toto porovnanie je menej náročné, než výpočet vzdialenosti dvoch agentov. Až keď sa ukáže, že agenty sú blízko seba, bude nasledovať pôvodný výpočet separácie agentov od seba.

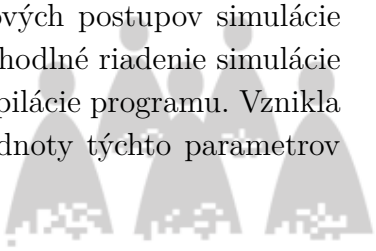
Opis implementácie

Do agenta pridáme parametre *gridX* a *gridY*, ktoré budú reprezentovať pozíciu v gride. V podstate pôjde o pozíciu agenta vydelenú veľkosťou gridu s orezanou desatinou časťou. Vytvoríme pomocnú triedu *CGrid*, ktorá bude zastrešovať aktualizáciu grid súradníc na základe pozície agenta. Pri vyhodnocovaní akcie agenta sa v prípade, že sa agent pohne, aktualizujú aj jeho grid súradnice. Trieda *CGrid* obsahuje metódu na kontrolu, či sa agenty nachádzajú blízko seba. Táto metóda spočíva v jednoduchom teste, či sa jeden agent nachádza v mriežke 3x3 v okolí druhého agenta.

8.7 Vytvorenie konfiguračného súboru

Analýza problému

Postupným vylepšovaním aplikácie a implementovaním nových postupov simulácie davu, sa do aplikácie dostalo množstvo parametrov. Pre pohodlné riadenie simulácie je potrebné umožniť menenie parametrov bez nutnosti kompilácie programu. Vznikla tak potreba konfiguračného súboru, v ktorom sa budú hodnoty týchto parametrov nastavovať.



Návrh riešenia

Konfiguračný súbor bude jednoduchý *INI* súbor. Bude obsahovať viacero skupín ako ”agent“, ”environment“, ’visualization“ atď. Bude treba skontrolovať zdrojový kód

programu a identifikovať premenné, potrebné pre riadenie simulácie. Následne implementovať triedu, ktorá bude schopná načítať parametre z *INI* súboru, alebo príkazového riadku. Nakoniec bude treba rozdistribuovať túto triedu pre všetky spustené moduly.

Opis implementácie

Na načítanie *INI* súboru a príkazového riadku sme využili *boost* knižnicu, konkrétne triedu *boost::program_options::options_description*. Samotné načítanie prebieha v module *Wrapper*, odkiaľ sú konfiguračné parametre predané ďalej ostatným modulom. Zoznam možných parametrov programu aj s ich popisom môžeme získať pomocou parametra "—help" pri spustení programu. Zoznam nastavení sa uloží do logovacieho súboru programu.

8.8 Vytvorenie dotazníka do TP cupu

V rámci súťaže TP cup bolo potrebné vytvoriť dotazník obsahujúci požadované informácie ohľadom charakteristiky projektu, našej vízie ďalšieho uberania sa, ako aj predstavenia celého tímu. Dotazník je priložený v nasledujúcej sekcii.

TÍM č. 08

Členovia tímu (študenti): Bc. Adam Pomothy, Bc. Lukáš Pavlech, Bc. Michal Fornádeľ, Bc. Marek Hlaváč, Bc. Martin Košický, Bc. Daniel Petráš

Ved. tímu (pedagóg): Ing. Peter Lacko, PhD.

Motto tímu: *Zachráň sa kto môžeš!!!*

Názov projektu: Simulácia Davu

O ČOM JE NÁŠ PROJEKT? (cca 300 slov)

Náš projekt je o vytvorení nového nástroja pre účely simulácie davu v uzavretých priestoroch. Na rozdiel od mnohých iných nástrojov, nami vyvinuté simulačné prostredie dokáže vykonávať a vizualizovať simuláciu v reálnom čase. Tento fakt je umožnený použitím distribuovaného počítania. Výsledná vizualizácia je následne konvertovaná do 3D zobrazenia pre čo najrealistickejší pohľad na evakuačné situácie.
Simulácia davu sa vo všeobecnosti delí na dva základné prístupy:

- makroskopický prístup
- mikroskopický pristup

Makroskopický prístup simulácie vníma dav ako jednu entitu. Vďaka tomu že sa ignorujú rôzne aspekty ľudského správania, nie je simulácia moc reálna. Na druhú stranu sú simulácie tohto typu veľmi efektívne, lebo zanedbávajú detaily individuálneho správania.

Mikroskopicky prístup sa na rozdiel od makroskopického zameriava na čo najvierohodnejšie modelovanie správania jedincov. Človek je definovaný ako autonómny inteligentný jedinec - agent. Simulácia je síce realistickejšia, výpočtové nároky sú ale výrazne vyššie. Tento typ simulácie je vhodný najmä na modelovanie situácii v interiéroch.

Pre dosiahnutie čo najlepšieho výsledku náš projekt využíva myšlienky z oboch základných prístupov. Vďaka ich kombinácií je dosiahnuté čo najvierohodnejšie simulovanie ľudského správania. Každý agent je samostatnou entitou, ktorá rozhoduje o tom kde sa chce pohnúť. Následne je pohyb každého agenta ovplyvňovaný pohybom okoloidúcich agentov - čiastočne je strhnutý s davom.

ČO NÁM DÁVA PRÁCA NA TOMTO PROJEKTE? (cca 300 slov)

Vďaka tomuto projektu sme sa naučili pracovať a komunikovať v rámci tímu, čo určite využijeme v budúcnosti využijeme aj v profesionálnom živote. Prešli sme si viacerými fázami rozvoja tímu a teraz sme schopní bez problémov spolupracovať aj napriek rôznym povahám a znaklostiam jednotlivých členov tímu.

Spoznali sme praktické stránky vývoja a manažovania softvérového projektu. Zistili sme čo všetko takýto vývoj obnáša po technickej podpore, ktorá je nevyhnutná najmä pri spolupráci na diaľku. Oboznámili sme sa s mnohými prekážkami pri pridelovaní, distribuovaní a koordinácii práce.

Vďaka zvolenej téme sme sa oboznámili s mnohými novými technológiami, ktoré sa týkajú problematiky simulácie davu. Mnohí z nás sa prvýkrát stretli s programovaním fyzikálnych a sociálnych interakcií.

V neposlednom rade sme spoznali nových ľudí a získali nové kamarátstva.

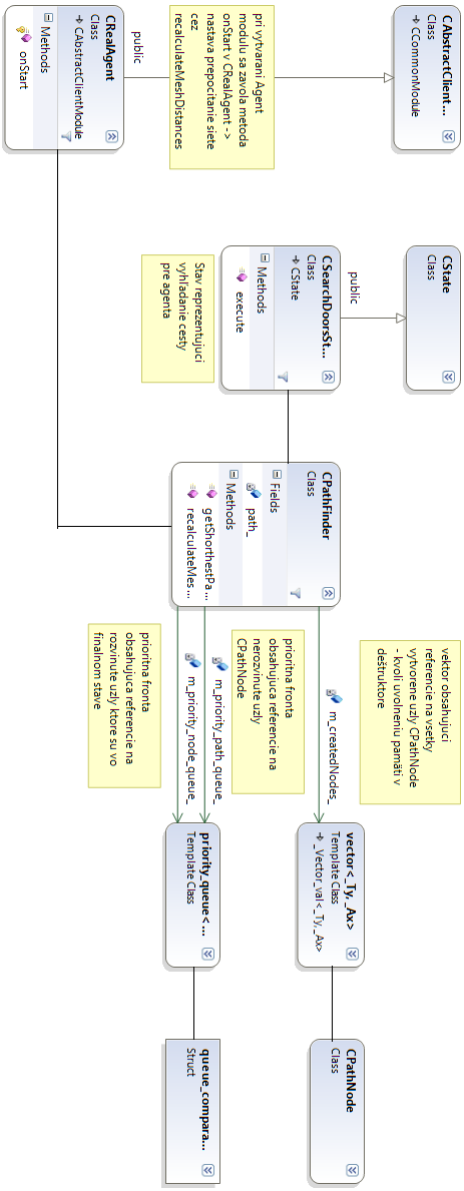
PREČO JE NÁŠ PROJEKT ZAUJÍMAVÝ? (cca 80-100 slov)
Existuje mnoho prístupov k simulácii davu. Jednotlivé existujúce riešenia si vyberajú len niektoré z nich, nakoľko vznikli pre konkrétny účel a snažia sa naimplementovať len potrebné minimum, aby boli čo najefektívnejšie. Navyše, skutočne dobré riešenia sú platené.

Náš nástroj na simuláciu neobsahuje žiadne kompromisy. Spájame viaceré technológie aby sme dosiahli čo najrealistickejší pohľad na situáciu. Zvýšené výpočtové nároky riešime distribuovaným počítaním, pri ktorom využívame samostatné, navzájom komunikujúce moduly.

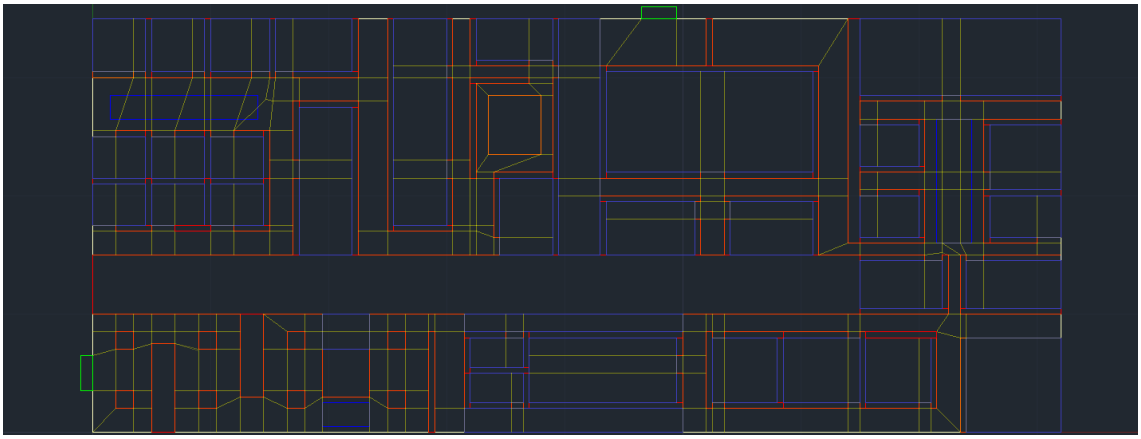
Navyše sme sa rozhodli zamerať aj na vizuálnu stránku simulácie a dosiahnutie čo najlepšieho zážitku zo sledovania evakuačných situácií. Preto je simulácia v reálnom čase pretransformovaná do 3D podoby.

POUŽITÉ TECHNOLÓGIE: C++, MPI, OPENMP, OGRE3D, GLUT

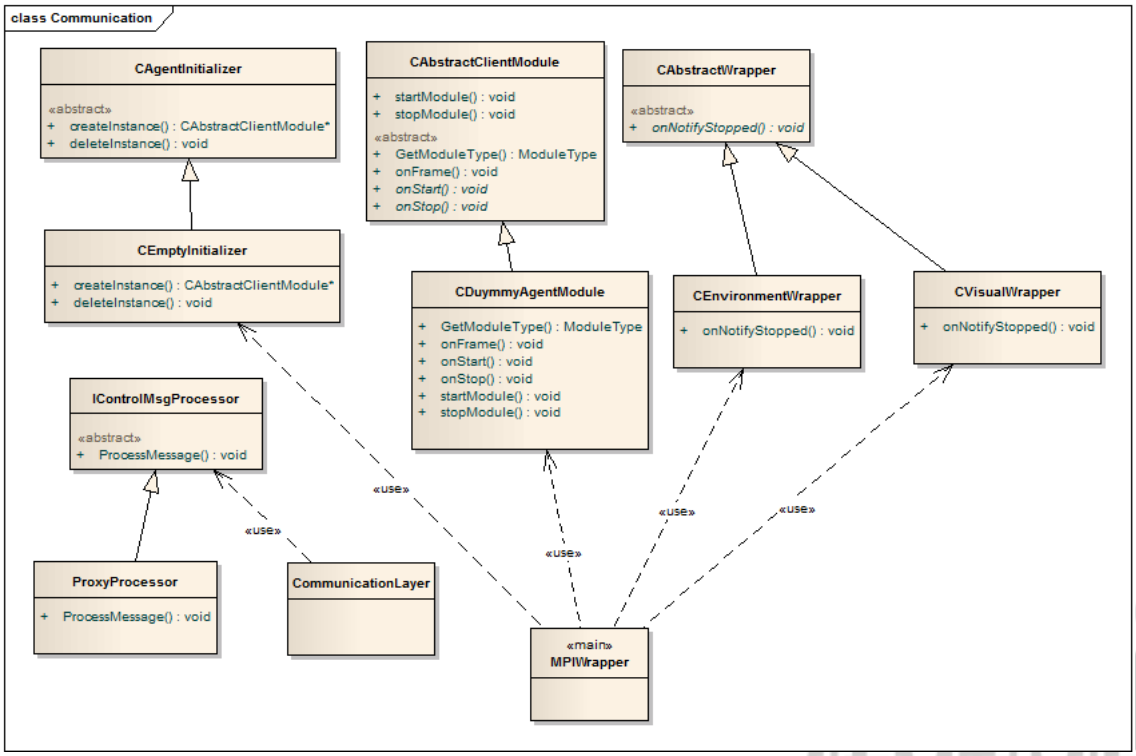
O ČOM TO VLASTNE JE? (Hlavná myšlienka - skrátená verzia odpovede na 1. otázku – max. 50 slov)
Projekt simulácie davu je o vytvorení nového simulačného prostredia, ktoré bude schopné vykonávať simuláciu evakuácie v uzavretých priestoroch v reálnom čase. Pre dosiahnutie požadovanej funkcionality, je potrebné efektívne využiť výpočtové schopnosti - preto projekt využíva možnosti distribuovaného počítania. Na dosiahnutie čo najrealistickejšieho pohľadu je evakuácia konvertovaná do 3D zobrazenia.



Obr. 8.4: Diagram tried pri inicializácii agent modulu triedou RealAgent



Obr. 8.5: Náhľad veľkej mapy



Obr. 8.6: Diagram tried - MPI paralelizácia

Kapitola 9

Šprint#7

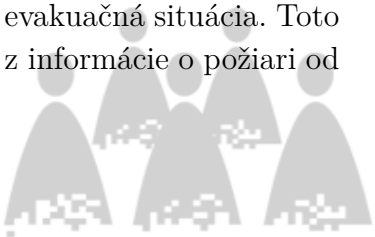
ID	Názov úlohy	Zodpovedný
1	Strach a šírenie strachu	Adam pomothy, Michal Fornádeľ
2	Optimalizácia strachu (pomocou A*)	Lukáš Pavlech
3	Validácia mapy	Marek Hlaváč
4	Vizualizácia východov	Martin Košický
5	3D Rozmer	Martin Košický
6	Vizualizácia strachu	Martin Košický
7	Optimalizácia Path following	Daniel Petráš

9.1 Strach a šírenie strachu

Analýza problému

Aby sme zvýšili realistickosť správania agentov v stresových situáciach, bolo potrebné identifikovať všeobecnú črtu v správaní ľudí v reálnych evakuačných situáciach. Touto črtou je strach. Ak sa osoba dozvie o požiari, okamžite to ovplyvní jej správanie. Reakcia človeka na bezprostredné ohrozenie života je útek, ktorý je podmienený strachom.

Hladina strachu vzrastie ihneď po zistení, že sa vyskytla evakuačná situácia. Toto zistenie môže vyplývať z priameho svedectva napr. požiaru, z informácie o požiari od iného agenta alebo zo spustenia alarmu.



Návrh riešenia a opis implementácie

Pri návrhu riešenia sme museli v prvom rade myslieť na efektívnosť. Z toho dôvodu sme chceli v čo možno najväčšej miere využívať už existujúce štruktúry, týkajúce sa digitálnej reprezentácie máp.

Hlavnou štruktúrou pre implementáciu strachu je navigačná mesh mapa, ktorá sa využíva pre plánovaní cesty a navigácii agenta. Oheň nie je bod, ale viaže na sa na konkrétny mesh v rámci mapy, ktorý sa nastavuje prostredníctvom parametra pri spúštaní simulácie.

Pre potreby šírenia strachu a ovplyvňovanie navigácie agenta na základe pozície ohňa sme museli doplniť nové vlastnosti agentom aj jednotlivým meshom a zároveň bolo potrebné doplniť spúšťanie alarmu po používateľom zadanom intervale.

Atribút strachu pre agenta

Agent dostal atribút vyjadrujúci úroveň strachu. Po tom, ako dostane informáciu o ohni, sa jeho hodnota nastaví na maximum. Hodnota strachu priamo ovplyvňuje rýchlosť pohybu agenta. Čím väčší strach agent má, tým sa pohybuje rýchlejšie. Postupom času hladina strachu klesá, spolu s ňou teda klesá aj rýchlosť agenta. Ak už ale agent raz nadobudol maximálnu hranicu strachu, už nikdy neklesne pod určenú hranicu - rovnako, ako sa v skutočnosti osoba nezbaví strachu úplne až pokiaľ sa nedostane z budovy.

Atribút strachu pre navigačný mesh

Rovnako aj každý mesh dostal nový atribút typu *boolean* - strach. Na začiatku vypuknutia požiaru má tento atribút hodnotu *true* iba v meshi, kde vypukol požiar. Po vypuknutí alarmu sa táto hodnota nastaví na *true* na každom meshi. Môže sa však zmeniť aj skôr - ak agent vstúpi do tohto, alebo bezprostredne susediacich meshov, tak sa jeho strach zvýši na maximum a začne utekať k najbližšiemu východu. Ak sa cestou dostane do meshu, ktorý ešte nemá hodnotu strachu nastavenú na *true*, tak ju zmení na *true*. Následne sa všetkým agentom v danom meshi zvýši strach na maximum. Tým sa simuluje predávanie informácie o požiari medzi agentmi.

Spustenie alarmu

Alarm je implementovaný ako samostatné vlákno, ktoré sa spustí hneď po začiatku simulácie. Po uplynutí nastaveného času sa všetkým meshom nastaví hodnota strachu na *true*.

Ovplyvňovanie plánovania evakuačnej cesty agenta

Vypuknutie požiaru musí ovplyvňovať aj agentovo plánovanie únikovej cesty, potom ako sa dozvie o potrebe evakuovať budovu. Plánovanie cesty má na starosti algoritmus, ktorý je opísaný v samostatnej časti dokumentu. Tento algoritmus plánuje na základe ohodnotenia uzlov, pričom si vyberá primárne uzly s menším ohodnotením.

Našou úlohou bolo teda zvýšiť ohodnotenie uzla s požiarom a toho zväčšené ohodnotenie rovnomerne rozšíriť do okolitých meshov. Ide o jednoduchý algoritmus, ktorý zvýši hodnotenie uzla s požiarom o hodnotu X, získa kolekciu obsahujúcu susedov daného uzla a všetkým zväčší ohodnotenie o hodnotu $X * Y$, kde $Y < 1$. A toto sa opakuje pokiaľ sa neminú uzly.

9.2 Optimalizácia strachu (pomocou A*)

Analýza problému

Doteraz implementovaná verzia požiaru, resp. strachu nepočíta s faktom, že určitá sila požiaru (zadymenia) je pre evakujúceho sa človeka bezpečnou. Zároveň pri vyhľadávaní cesty cez algoritmus A* končí hľadanie až po prejdení všetkých možných ciest, čo je za určitých podmienok zbytočné a veľmi neefektívne.

Návrh riešenia

Pre simulačné prostredie sa zadefinujú nové parametre v konfiguračnom súbore ako napr: *fire_path_cost* (sila ohňa v uzle, kde oheň začal), *fire_distribution_power* (konštanta rozšírenia ohňa do susediacich nav. mesh), *save_fire_cost* (hodnota ohňa ktorá je ešte bezpečná pre chodca). Navigačný uzol získa nový parameter - silu ohňa v tomto uzle. Vyhľadávanie cesty pomocou A* získa novú heuristiku: cesty nachádzajúce sa v prioritnej fronte budú vrátené na základe funkcie f, ktorá bude počítaná ako podiel ceny cesty ku počtu prejdených vrcholov v ceste. V prípade, ak hľadaná cesta sa dostane k vrcholu, ktorého sila ohňa je vyššia ako maximálna bezpečná hodnota, cesta nie je bezpečná, teda nepoužije sa a hľadá sa iná. Ak nie je možné nájsť cestu do cieľa, aktivuje sa metóda na získanie čo najväčšej vzdialenosti od ohňa.

Opis implementácie

Pre každý uzol navigačnej mesh sa pridá nový parameter - cena ohňa v danom vrchole. Pri spustení agent modulu sa najprv prepočítajú vzdialenosti v navigačnej mesh od cieľa (východu z budovy). Následne v prípade ak hodnota *fire_node_index* sa nachádza v zozname vrcholov sa spustí prepočet upravený prepočet cien cesty použitý už v úlohe Implementácia strachu. Ten preráta navigačnú mriežku s prihliadnutím na parametre sily a rozšírenia ohňa. Agent pri vyhľadaní cesty najprv skúša vyhľadať cestu pomocou Q-learning algoritmu. Ak táto možnosť je neúspešná (narazil na oheň), aktivuje sa hľadanie pomocou A* algoritmu, ktorý sa snaží o vyhľadanie akejkoľvek bezpečnej cesty do cieľa. Ak nie je možné vyhľadať cestu do cieľa ani pomocou A*

algoritmu (agent je v slepej uličke), aktivuje sa metóda na získanie bezpečnej cesty k čo najvzdialenejšieho bodu od požiaru.

9.3 Validácia mapy

Analýza problému

Na základe problémov, ktoré vznikali pri používaní chybných dát získaných z mapy, je potrebné vykonávať ich validáciu, aby sa predišlo nežiadúcim udalostiam, ako napríklad:

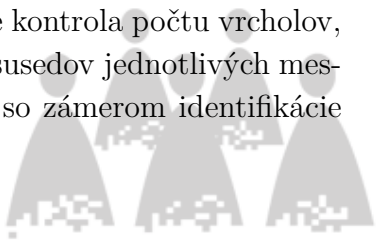
- uviaznutie agentov v mape,
- nesprávne hodnoty reprezentujúce meshe a prekážky,
- chýbajúce prepojenie meshov v navigačnej sieti s východmi,
- nespojené susedné meshe v navigačnej sieti,
- duplicitné informácie.

Návrh riešenia

Riešenie spočíva vo vytvorení špeciálneho kroku, ktorý je realizovaný medzi spracovaním dát z mapy a ich následným používaním vo výpočte simulácie. Po načítaní dát z mapy je nutná špeciálna kontrola každého jedného objektu, ktorý reprezentuje citlivé informácie. Validáciu je následne možné rozdeliť do niekoľkých menších krokov, z ktorých každý testuje dáta vzhľadom na konkrétny problém.

Opis implementácie

Implementácia validácie sa skladá z funkcií, ktoré sú volané v triedach CNavMesh a Map. Celkovo sa validácia skladá z troch krokov a niekoľkých testov, ktoré sú uskutočňované pri vytváraní grafu ciest z navigačnej siete. Prvým je kontrola počtu vrcholov, resp. hrán jednotlivých objektov. Ďalším je kontrola počtu susedov jednotlivých meshov. Posledným krokom je postupné prechádzanie grafom so zámerom identifikácie stratených, resp. chybných meshov.



9.4 Vizualizácia východov

Analýza problému

Pre zlepšenie vizuálnej stránky aplikácie je potrebné vyznačiť miesta, predstavujúce východy pre agentov.

Návrh riešenia a opis implementácie

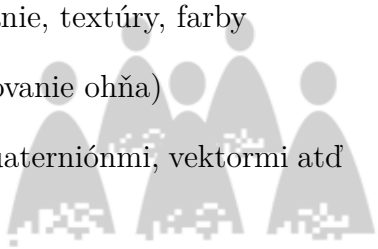
V metóde *CVisualization::InitGL* sa bude implementovať inicializácia geometrie, ktorá bude predstavovať východy. V metóde *CVisualization::DrawGLScene* sa bude vykonávať vykresľovanie. V metóde *CVisualization::DrawGLScene* sa vykresľuje pomocou sekvencií metód *glEnableClientState, glVertexPointer, glDrawArrays, glDisableClientState*.

9.5 3D Rozmer

Analýza problému

Pre vytvorenie 3D vizualizácie je niekoľko alternatív. Je možné vytvoriť vykresľovanie aj so spravovaním scény ručne pomocou *OpenGL*, alebo *Direct3D*. Nevýhoda tohto prístupu je časová náročnosť. Preto sme zvolili použitie existujúcej knižnice *Ogre3D*, vytvorenej pre tieto účely. *Ogre3D* má nasledovné vlastnosti:

- poskytuje možnosť výberu vykresľovacieho podsystému(*OpenGL*, *Direct3D*)
- umožňuje manažovanie všetkých načítaných zdrojov aj detekciu tzv. *memory leak* situácií
- umožňuje manuálne načítavanie geometrie, alebo použitie geometrie vstavanej v systéme, ktorá je *.mesh* formáte
- umožňuje prácu s materiálmi – viacpásové vykresľovanie, textúry, farby
- umožňuje prácu s časticovým systémom (pre vykresľovanie ohňa)
- poskytuje podporné knižnice pre prácu s maticami, quaterniónmi, vektormi atď
- umožňuje animáciu 3D modelov



Návrh riešenia a opis implementácie

Diagram tried pre 3D vizualizáciu sa nachádza na obr. 9.1.

3D vizualizáciu má na starosti trieda *COgreVisualizationModule*. Obsahuje množinu metód, z ktorých sú niektoré volané pri vytváraní triedy, niektoré v každom snímku a niektoré na konci životného cyklu triedy.

Ďalšie komponenty, ktoré sa starajú o 3D vizualizáciu:

Ogre::FrameListener

Je rozhranie v *Ogre*. Jeho metódy sa volajú vždy každú snímku. Jednou z metód je *frameStarted*, ktorá sa volá vždy keď sa spúšťa jeden snímok.

Ogrem::WindowEventListener

Je rozhranie, ktorého metódy sa volajú, keď sa zmení stav okna.

OIS::KeyListener

Je rozhranie v knižnici OIS, ktorú OGRE používa a slúži pre spracovanie zachytenia zmeny stavu klávesnice.

OIS::MouseListener

Je rozhranie, ktoré zachytáva zmenu stavu myši.

OgreBits::SdkTrayListener

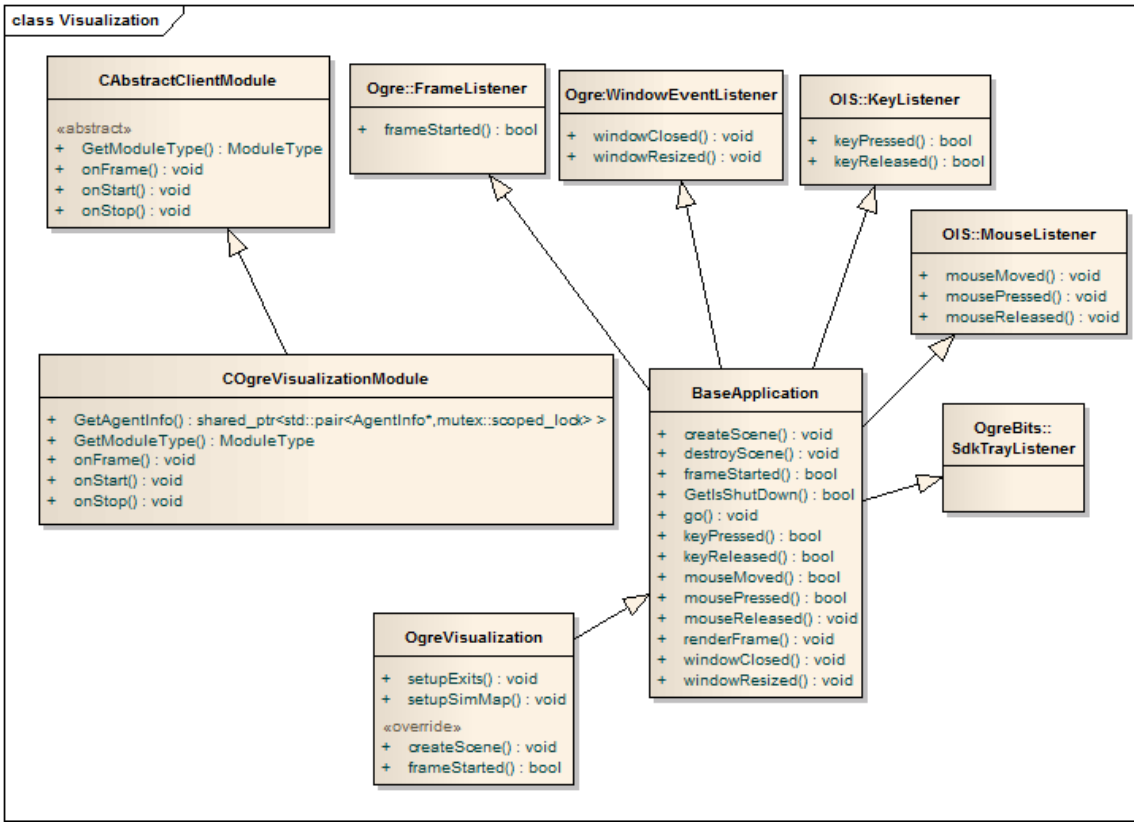
Je trieda, ktorá vnútorne zabezpečuje niektoré základné funkcionality 3D vizualizácie. My budeme používať kameru v prvej osobe.

BaseApplication

Je trieda, ktorá vytvára a spúšťa celý systém vykresľovania. Táto trieda nie je priamo určená pre konkrétnu aplikáciu, preto sú *OgreVisualization* a *BaseApplication* sú rozdelené.

OgreVisualization

Je trieda, ktorá vykonáva načítavania zdrojov pre konkrétne využitie.V našom prípade to je načítanie mapy, načítanie modelov, zmenu animácie a polohu modelov vo svete.



Obr. 9.1: Diagram tried - 3D Rozmer

9.6 Vizualizácia strachu

Analýza problému

Aby bolo možné vizuálne identifikovať oblasti, v ktorých sa šíri medzi agentami strach, je potrebné nastaviť farby agentov podľa úrovne strachu. Čím má agent väčší strach tým je jeho farba bližšie k červenej.

Návrh riešenia a opis implementácie

Pre každého agenta bol vytvorený vlastný materiál, pretože štandardne majú všetci agenti spojenie s tou istou inštanciou materiálu. Tento materiál sa vytvára pri prvom vykresľovaní, pretože vizualizácia dovtedy nevie koľko bude agentov a ich počet sa bude pri postupnej evakuácii zmenšovať. Pri každom vykreslení sa zmení difúzna zložka materiálu podľa strachu - tým sa sa teda upraví ich farba.

9.7 Optimalizácia Path following

Analýza problému

Úlohou je zlepšenie navigácie agentov pomocou úpravy vhodných nastavení v konfiguračnom súbore. Cieľom je dosiahnuť plynulejší a realistickejší pohyb agentov, zamedziť bezdôvodnému zasekávaniu a nepredvídateľným zmenám smeru pohybu.

Návrh riešenia

Začne sa simulovaním na jednoduchších mapách s jedným agentom. Experimentovaním s hodnotami parametrov sa pokúsime dosiahnuť správny pohyb agenta. Následne sa postupne budú pridávať ďalšie agenty a doladí sa ich spoločný pohyb. Nakoniec sa otestuje pohyb agentov na veľkej a komplexnej mape.

Opis implementácie

Po experimentovaní boli parametre boli nastavene nasledovne:

- agent.width$\bar{0}$.4
- agent.stop_walk_angle$\bar{1}$.0
- agent.slow_walk_angle$\bar{0}$.4
- agent.max_angle_per_frame$\bar{0}$.2
- agent.max_step_size$\bar{1}$
- agent.max_slow_step_size$\bar{0}$.5
- agent.separation_dist$\bar{0}$.25
- agent.avoidance_dist$\bar{0}$.2
- agent.separation_min_dist$\bar{0}$.4
- agent.avoidance_min_dist$\bar{0}$.4
- agent.path_following_weight$\bar{0}$.6
- agent.separation_weight$\bar{0}$.3
- agent.avoidance_weight$\bar{0}$.3
- agent.flow_field_weight$\bar{0}$.3

Kapitola 10

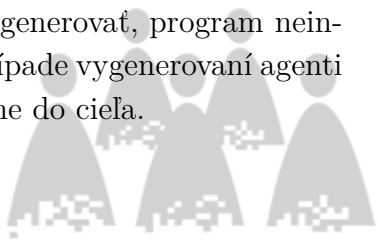
Šprint#8

ID	Názov úlohy	Zodpovedný
1	Logovanie chyby preplnených spawnov	Lukáš Pavlech
2	Úprava NavMesh pre FIIT	Marek Hlaváč
3	Vizualizácia východov 3D	Martin Košický
4	Pridanie pohľadu kamery "top"	Martin Košický
5	Rozdistribuovanie agent modulu	Martin Košický
6	Resize mapy	Martin Košický
7	Ovládanie 2D vizualizácie	Martin Košický
8	Vizualizácia ohňa	Martin Košický
9	Optimalizácia pochodu agentov (točenie agenta)	Daniel Petráš
10	Presunutie strachu do agenta	Daniel Petráš
11	Odstránenie warningov	Daniel Petráš
12	Odstránenie problému s točiacim sa agentom	Daniel Petráš

10.1 Logovanie chyby preplnených spawnov

Analýza problému

Pri zadaní väčšieho počtu agentov ako je fyzicky možné vygenerovať, program neinformuje používateľa žiadnym spôsobom. Zároveň v tomto prípade vygenerovaní agenti sa skoro vôbec nehýbu - trvá dlhý čas, kým sa agent dostane do cieľa.



Návrh riešenia

V module prostredia sa pred fyzickým vytvorením agenta skontroluje, či zadaný počet agentov nie je väčší ako počet možných pozícií. Ak tomu tak je, počet vygenerovaných agentov sa rovná počtu možných pozícii. Problém nehýbania sa agentov spôsobuje

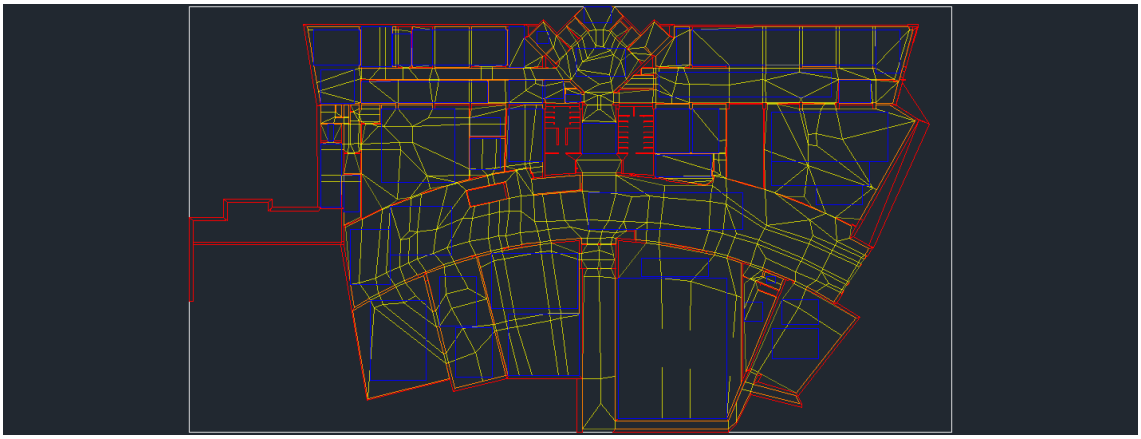
generovanie agentov príliš blízko pri sebe - ich osobný priestor sa rovná ich šírke. Riešenie tohto problému je zdvojnásobenie veľkosti osobného problému, vďaka čomu detekcia koliízií a steering vectors nerobia problémy.

10.2 Úprava NavMesh pre FIIT

Analýza problému

Úprava navigačnej siete pre FIIT mapu spočíva najmä v odstránení problémov s nežiaducim správaním agentov v malých priestoroch. Súvisiacou záležitosťou je rozšírenie siete na celú mapu a jej prispôsobenie novým oblastiam mapy.

Finálna mapa pozostáva z 550 meshov a umožňuje vygenerovať veľké množstvo agentov, ktorých počiatočné pozície sú rozmiestnené v zhlukoch po celej mape. Mapa obsahuje 4 východy, hlavnú chodbu, dve auly a veľké množstvo miestností a priestranstiev (viď. Obr. 10.1). Celá je postavená na základe skutočného 3D modelu reálnej budovy vo výstavbe.



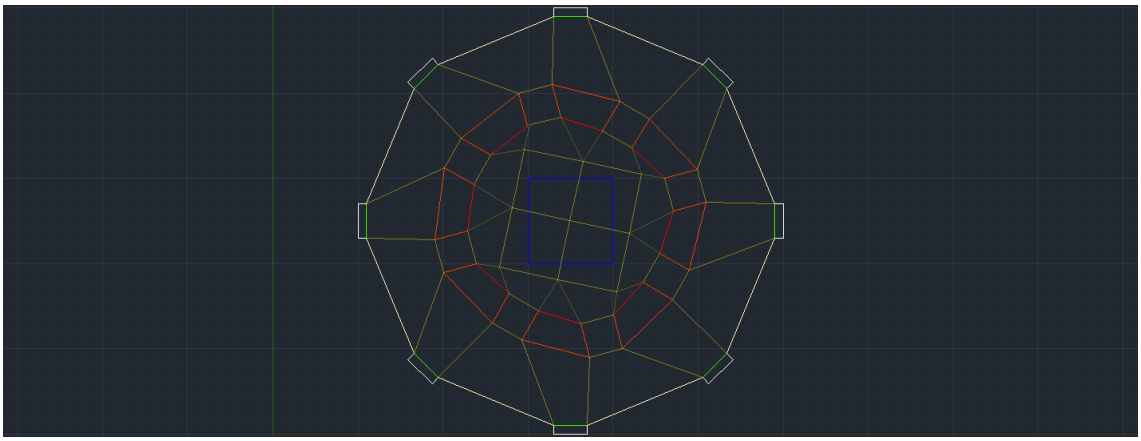
Obr. 10.1: Náhľad FIIT mapy

Návrh riešenia a opis implementácie

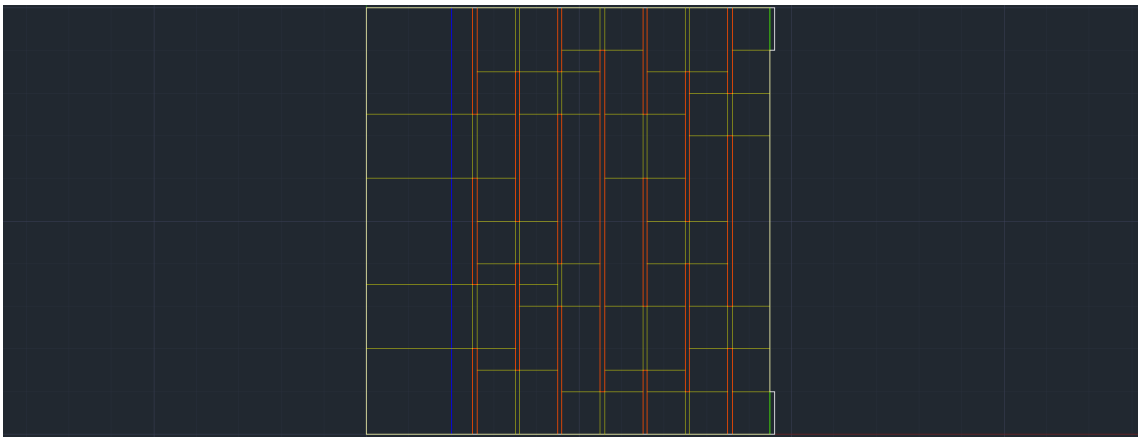
Navrhnutých je 5 máp, ktorých cieľom je simulácia nasledovných situácií:

- zlievanie väčších (hlavných) prúdov agentov s menšími (vedľajšími), ktoré je možné sledovať pri reálnych evakuáciách budov,
- zúženie priechodov agentov, čo má za dôsledok zaseknutie, resp. čakanie agentov v dave, ktorý je natlačený na hrdle priechodu,

- hľadanie vhodnej únikovej cesty v jednoduchom prostredí s viacerými možnosťami úniku (viď. Obr.MapaROZCHOD.),
- hľadanie vhodnej únikovej cesty v komplikovanom prostredí s mnohými prekážkami (viď. Obr.MapaPOLE.),
- správanie a ovplyvňovanie protichodných skupiniek agentov.



Obr. 10.2: Náhľad mapy "Rozchod".



Obr. 10.3: Náhľad mapy "Pole".

10.3 Vizualizácia východov 3D

Analýza problému

Úloha spočíva naimplementovaní zobrazenia východov v 3D vizualizácii.

Návrh riešenia

V metóde *setupExits* sa vytvorí manuálne vytvorenie geometrií, ktoré budú predstavovať východy z budov.

Opis implementácie

Pre manuálne vytvorenie geometrie bolo treba vytvoriť manuálny mesh. Ten má zdieľané body medzi všetkými submeshmi. Každý submesh obsahuje indexy ku geometrii. Geometria a indexy sú pridané ako indexy k zásobníkom, ktoré sú nahraté do pamäte grafickej karty.

10.4 Pridanie pohľadu kamery "top"

Analýza problému

Pri spustení 3D vizualizácie má byť pohľad nastavený tak, aby bolo vidieť mapu, bez toho aby bolo treba posúvať kameru.

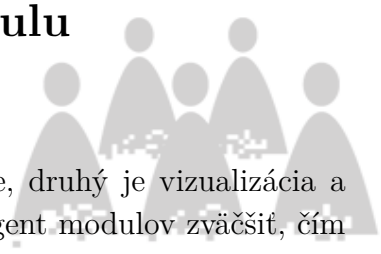
Návrh riešenia

Pozícia kamery je vypočítaná, ako stred celej mapy, a tento pohľad sa posunie do výšky o 20 bodov a tiež po osi x aj po osi y o pár bodov. Vektor kamery je vypočítaný ako stred súradnicovej sústavy mínus pozícia kamery. Tento vektor musí byť následne normalizovaný.

Opis implementácie

Inicializácia pohľadu kamery sa vykonáva vo funkcii *setupSimMap* v triede *OgreVisualization*.

10.5 Rozdistribuovanie agent modulu



Analýza problému

Aplikácia spúšťa maximálne 3 procesy. Prvý je prostredie, druhý je vizualizácia a tretí je agent modul. Je potrebné aby bolo možné počet agent modulov zväčšiť, čím sa zvýši výkon aplikácie.

Návrh riešenia a opis implementácie

Počet agent modulov bude počet všetkých procesov, ktoré sú spustené v rámci MPI paralelizovaného prostredia mínus dva. Agent modul bude vytvárať kontexty len pre podmnožinu agentov a tieto podmnožiny budú disjunktné. V každej podmnožine bude počet agentov vydelený počtom agent modulov.

10.6 Resize mapy

Analýza problému

2D vizualizácia má roztiahnuť mapu tak, aby mala pri štarte rozmer obrazovky.

Návrh riešenia

Pre roztiahnutie mapy je potrebné prepočítať aké sú rozmery okna v súradniciach oka, ak sú známe rozmery okna v pixloch. Treba preto zistiť bod v hĺbke nula v rohu obrazovky v súradniciach oka z pozície 1,1 v pixlových súradniciach. Tento istý bod treba vypočítať v hĺbke 1. Potom sa zoberie polpriamka, ktorá ide z prvého bodu, cez druhý. Tam kde sa pretína s rovinou obsahujúcou mapu, vypočítame súradnice, ktoré predstavujú roh mapy.

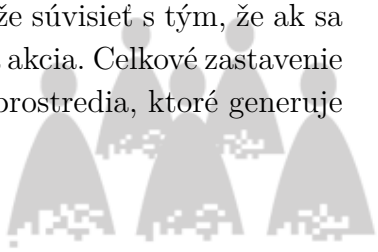
Opis implementácie

Pre zjednodušenie výpočtov sa použila funkcia *gluUnProject*.

10.7 Optimalizácia pochodu agentov (točenie agenta)

Analýza problému

Občas sa stáva, že agent zastane a začne sa točiť na mieste. Prípadne zastane a nerobí nič, aj keď by mal. Ukázalo sa, že točenie agenta môže súvisieť s tým, že ak sa nevygeneruje nová akcia, chybne sa použije posledná použitá akcia. Celkové zastavenie sa agentov súvisí zo zahltením modulu agent správami od prostredia, ktoré generuje správy rýchlejšie, ako ich dôkaze modul agent spracovávať.



Návrh riešenia

Točeniu agentov je možné zabrániť explicitným vynulovaním starej akcie pred ďalším spracovaním, takisto opravou chyby v kóde, ktorý vracal príznak, že sa akcia vygenerovala, aj keď tomu tak nemuselo byť.

K zahlteniu dochádza kvôli nestíhaniu agent modulu, takže možné riešenie, je optimalizácia jeho vykonávania. Konkrétne:

1. vylepšenie grid algoritmu
2. spojenie výpočtov *separation* a *flow field* (nakoľko obe potrebujú iterovať cez všetkých agentov)

Pre definitívne riešenie treba zabezpečiť, aby agent modul spracovával vždy aktuálnu správu od prostredia, staršie správy budeme zahadzovať.

Opis implementácie

Vylepšený *grid* algoritmus spočíva v predspracovaní zoznamu agentov zaslaného z prostredia. Agenty sa zadelia do hash mapy, kde kľúčom bude pozícia agenta v gride a hodnota bude vektor agentov na tejto pozícii. Týmto spôsobom budeme môcť rýchlo určiť relatívne malé množstvo agentov, ktoré bude treba spracovať pri výpočte separácie a *flow field*, namiesto spracovávania všetkých agentov. Predspracovanie bude vykonané iba raz pre každé prijatie správy z prostredia.

Spojenie výpočtov separácie a *flow field* je jednoduchá úprava, kde namiesto dvoch cyklov použijeme jeden, v ktorom budeme rátať oba vektory.

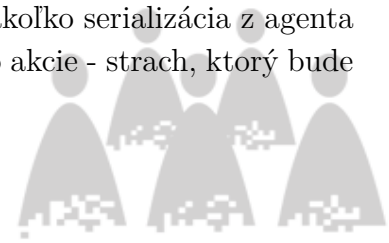
10.8 Presunutie strachu do agenta

Analýza problému

Aby sme boli schopný vizualizovať strach agenta, je potrebné, aby bol parameter vyjadrujúci strach súčasťou agenta. Doteraz bol súčasťou *Context*-u, nakoľko sa využíval iba pri výpočtoch v module *Agent*.

Návrh riešenia

Bude treba premiestniť pole *fear* z kontextu do agenta. Nakoľko serializácia z agenta do prostredia, prebieha pomocou akcií, vytvoríme nový typ akcie - strach, ktorý bude slúžiť na zmenu strachu agenta.



Opis implementácie

Po premiestnení poľa *fear* do agenta, bolo treba v projekte upraviť jeho použitie nasledovne:

1. Tam, kde sa čítal strach z kontextu, bolo treba čítať strach z agenta

2. Tam kde sa strach do kontextu zapisoval, bolo treba vygenerovať akciu strach

10.9 Ovládanie 2D vizualizácie

Analýza problému

Je potrebné vytvoriť ovládanie vizualizácie, čo zahŕňa pohyb obrazovky do strán a ovládanie približovania.

Návrh riešenia

Uhol pohľadu sa bude meniť podľa stlačených kláves. Posun sa bude vykonávať cez transláciu.

Opis implementácie

Aby bolo možné zachytiť stlačené klávesy, bolo vytvorené pole typu *bool* o veľkosti 256, pričom hodnota true znamená, že klávesa je stlačená a false, že nie. Pri zachytení správy o zmene stavu klávesnice, sa zmení prvok v poli na stlačené podľa toho aká klávesa bola stlačená, respektíve pustená. V každej snímke sa zmení súradnica pohybu podľa toho v akom je stave pole stlačených kláves.

10.10 Vizualizácia ohňa

Analýza problému

Pre lepšiu vizualizáciu je treba vytvoriť vykreslenie zdroja požiaru.

Návrh riešenia

Miesto požiaru je zadané používateľom ako identifikátor navigačného mesh-u. V danom mesh-i sa vytvorí zdroj časticového systému, na čo ma *Ogre* priamu podporu.

Opis implementácie

Pre časticový systém bolo potrebné vytvoriť skript pre *Ogre*, ktorý opisuje materiál a správanie časticového systému. Tieto veličiny opisujú farbu, rozptyl, rýchlosť, počet, životnosť.

10.11 Odstránenie warningov

Analýza problému

Pri kompilácii programu vzniká veľké množstvo varovných hlásení. Väčšina z nich nespôsobuje chyby, ale ľahko sa medzi nimi stratí naozaj dôležité varovanie, ktoré môže znamenať logickú chybu v programe.

Návrh riešenia

Varovania, s ktorými sa nedá nič robiť, ako napríklad knižnice tretích strán, treba ignorovať (nastavenie v projekte) a ostatné treba odstrániť bežným spôsobom, čiže zásahom do kódu.

Opis implementácie

V prípade kompilácie *dxf_lib* (slúži na načítanie dát mapy) sme použili preprocesor definíciu *_CRT_SECURE_NO_WARNINGS*, ktorá zabezpečí ignorovanie varovaní o použití nie zabezpečených funkcií ako strcpy namiesto *s_strcpy* a pod.

Vo všetkých projektoch ignorujeme varovanie *C4290*, ktoré hovorí o tom, že kompilátor ignoruje "*throws* (typ)" deklarácie metód. Tieto deklarácie nám v kóde slúžia čisto na informačné účely, takže ignorovanie tohto varovanie je úplne v poriadku.

V prípade potreby vnoriť hlavičkový súbor tretej strany, ktorý obsahuje kód na ktorý nemáme dosah, môžeme použiť nasledujúci kus kódu. Budeme ignorovať varovanie iba pre súbory medzi riadkami #pragma warning(push) a #pragma warning(pop). Varovanie *C4996* je varovanie ohľadom použitia *deprecated* funkcionality.

10.12 Odstránenie problému s točiacim sa agentom

Analýza problému

Problém nastáva, keď agent nevie nájsť cestu von z budovy a ostane uväznený v budove. V takom prípade sa vygeneruje cesta – slučka po ktorej má agent prejsť. Kvôli spôsobu navigácie po hranách, nie je tento pohyb možný. Problém nastane, aj keď sa agent blíži k cieľovej pozícii, ale nikdy ju celkom netrafí a preto točí na mieste.

Návrh riešenia

Je potrebné agenty navigovať do stredu nesledujúceho uzla. Po prejdení do tohto uzla ich opäť navigovať do nasledujúceho uzla, atď. Ak sa agent nachádza v poslednom uzle, navigovať do jeho stredu. Ak sa agent dostatočne priblíži k svojmu cieľu, už sa ďalej nebude pohybovať.

Opis implementácie

Reprezentáciu vyhľadanej cesty sme zmenili z páru (uzol, hrana) na uzol. Túto zmenu bolo treba vykonať v *CPathFinder* aj v *CPathFollowState*. Nakoľko sa týmto čiastočne zmenil pohyb agentov po mape, bolo treba spraviť vo veľkej mape pár úprav, aby sa agenty nezasekávali na ostrých zatáčkach. Pri tejto úprave sme odstránili aj niektoré iné problémy s mapou (nesprávne pospájané uzly).

Keďže stav agenta *CCalmState* vznikol ako kópia *CPathFollowState* s následnou úpravou, obsahoval ešte starú verziu navigácie, ktorá už ďalej nefungovala. Rozhodli sme sa navigačnú funkcionalitu vyňať do triedy *CNavigator*. Ako *CCalmState* aj *CPathFollowingState* upravujú a využívajú túto triedu na navigáciu agentov po mape. Ak sa v budúcnosti bude upravovať alebo optimalizovať kód na navigáciu agenta, bude stačiť spraviť úpravu na jednom mieste.

Ako neplánované vylepšenie v procese implementácie triedy *CNavigator*, sme pridali cachovanie cieľového bodu agenta, takže pokým sa nezmení node v ktorom sa agent nachádza (čo je väčšina času), netreba cieľový bod znovu rátať.

Kapitola 11

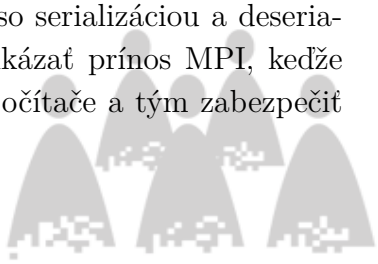
Šprint#9

ID	Názov úlohy	Zodpovedný
1	Otestovanie MPI paralelizácie pomocou viacerých počítačov	Lukáš Pavlech
2	Vytvorenie exemplárnych videí pre IIT.SRC	Marek Hlaváč
3	Oprava problému pri vyhľadaní agentovej pozície v navigačnej mesh	Marek Hlaváč
4	Otestovanie MPI paralelizácie pomocou viacerých počítačov	Marek Hlaváč
5	Zdvojenie obrazu	Martin Košický
6	Oprava zasekávania sa agentov	Daniel Petráš

11.1 Otestovanie MPI paralelizácie pomocou viacerých počítačov

Analýza problému

Použitie MPI rozhrania len pre jeden počítač nemá veľký zmysel, keďže program spomaľuje okrem štandardných výpočtov aj réžia spojená so serializáciou a deserializáciou dát. Pri použití viacerých počítačov, by sa mal ukázať prínos MPI, keďže jednotlivé moduly je možné rozmiestniť fyzicky na rôzne počítače a tým zabezpečiť zvýšenie zdrojov na dané výpočty.



Opis implementácie

Pre úspešné otestovanie paralelizácie bolo potrebné fyzicky poprepájať počítače cez switch a následne ich správane nakonfigurovať. Konfigurácia prebiehala z niekoľkých krokov. Najprv bolo potrebné zabezpečiť aby všetky počítače boli v rovnakej “workg-

roup". Následne muselo mať vytvorené jedno rovnaké spoločné konto s rovnakým prístupovým heslom. V tomto konte, alebo pre všetky konta na danom počítači, bolo potrebné nainštalovať MPICH2 do rovnakého priečinka. Zároveň program s potrebnými konfiguračnými súbormi bolo taktiež potrebné dať do rovnakého priečinka. Pre úspešné odsimulovanie bolo nutné vykonať ešte nasledujúce kroky:

1. pridanie okolitých počítačov do host súboru nachádzajúceho sa v C:/Windows/system32/drivers/etc/hosts :
   192.168.11.1 host1
   192.168.11.2 host2
2. pridanie MPICH2 do premennej PATH (nepovinne)
3. zaregistrovanie mpiexec (nepovinne)
4. nastavenie smpd
   (a) inštalácia smpd: *smpd –install*
   (b) pridanie host-ov do smpd: *smpd –sethosts host1 host2*
   (c) kontrola host-ov: *smpd -hosts*
   (d) povolenie mpiexec vo windows firewall
   (e) spustenie simulácie
5. Prvou alternatívou je spustenie identického rozdelenia počtu procesov (pri 2 počítačoch sa nasledujúci príklad rozdelí na 3 procesy pre prvý počítač a 3 procesy pre druhý počítač):
   mpiexec -n 6 "path/MPIWrapper.exe-config ../config.ini
   Druhá alternatíva je manuálna špecifikácia počtu procesov na jednotlivých počítačoch:
   mpiexec -n 3 -host master-pc "path/MPIWrapper.exe-config ../config.ini : -n 2 -host slave-pc "path/MPIWrapper.exe-config ../config.ini

Testovanie

Simuláciu sa podarilo úspešne spustiť pomocou viacerých počítačov. Už pri 500 agentoch boli badateľné výkonnostné zmeny v porovnaní s MPI simuláciou na jednom počítači. Najvýraznejší rozdiel bol badateľný pri požití 1500 agentov, mapy virtuálnej FIIT a spustení požiaru. Požiar bol umiestnený na takom mieste, aby väčšina agentov musela hľadať cestu pomocou A* algoritmu. Lokálne spustenie MPI paralelizácie vyústovalo do viditeľného zasekávania a FPS pod 1, na rozdiel od MPI s distribuovaným počítaním, kde FPS nekleslo pod 5 ani v čase vyhľadania cesty pre všetkých agentov. Bohužiaľ, veľké počty agentov nebolo možné odsimulovať kvôli neznámej MPI chybe, ktorá je pravdepodobne zapríčinená pretečením buffera komunikácie.

11.2 Vytvorenie exemplárnych videí pre IIT.SRC

Analýza problému

S projektom sme sa zúčastnili študentskej vedeckej konferencie IIT.SRC, v ktorej sme mali jeden z príspevkov v rámci tímových projektov pre súťaž TP CUP. Jednou z úloh, ktorú bolo nutné vykonať pre IIT.SRC, bola príprava video ukážok simulácií, ktoré boli súčasťou prezentácie pred komisiou a účastníkmi konferencie.

Návrh riešenia

Cieľom každej vytvorenej ukážky simulácie bolo vytvorenie prezentovanie určitého aspektu simulácie. Vytvorených bolo 5 simulácií, ktoré sa zamerali na 2D a 3D vizualizáciu:

- kolektívneho správania veľkého množstva agentov ,
- evakuácie z reálnej budovy ,
- priechodu veľkého počtu agentov úzkymi chodbami,
- stretnutie dvoch davov s rovnakým cieľom,
- plánovania agentov.

11.3 Oprava problému pri vyhľadaní agentovej pozície v navigačnej mesh

Analýza problému

Jedným z problémov, ktoré sa vyskytujú v navigačnej sieti počas vykonávania simulácie a nie je ich možné odstrániť automatickou validáciou mapy je problém, ktorý vzniká pri nesprávnom manuálnom nastavení pozícií susedných meshov. Dôsledkom problému je nesprávne spracovávanie informácií pri vyhľadávaní evakuačnej cesty. Dôvod problému je v tom, že jednotlivé meshe v navigačnej sieti sú chybne nastavené a prepojenie so susedmi nie je možné dohľadať. To spôsobuje, že nie je možné pokračovať v hľadaní pokračovania cesty a agenty sú tak uviaznutý v slepej uličke.

Návrh riešenia

Riešenie je len jedno, a to manuálne opraviť konkrétne chyby v mape, ktoré sú dôvodom vzniku problému s vyhľadaním evakuačnej cesty. Manuálna oprava je uľahčená

zobrazením informácie ohľadom chybných meshov, ktoré sú vypísané v logoch simulácie.

11.4 Vytvorenie prezentácie pre finálne odprezentovanie dosiahnutých výsledkov projektu

Analýza problému

Cieľom úlohy je vypracovanie jednotlivých tém, ktoré budú prezentované na finálnom stretnutí tímového projektu za účelom obhajoby nášho riešenia, ktoré spočíva v opise problémovej oblasti, architektúry aplikácie, použitých technológií a ukážok finálneho produktu.

Návrh riešenia

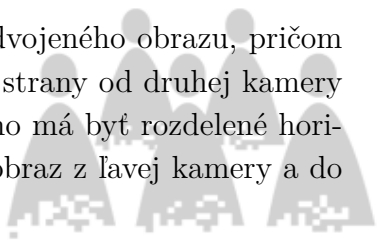
Digitálna prezentácia pozostáva z niekoľkých snímok, ktoré sa postupne zaoberajú:

- zadaním problému a opisom problémovej oblasti,
- architektúrou aplikácie,
- princípmi práce agentov a simulačného prostredia,
- výpočtovými metódami použitými v simulácii,
- reprezentáciou simulácie a vizualizáciou,
- testovaním.

11.5 Zdvojenie obrazu

Analýza problému

Pre 3D spracovanie obrazu sa ako možnosť javí použitie zdvojeného obrazu, pričom kamera jedného obrazu je paralelne umiestnená mierne do strany od druhej kamery (simulovanie rozloženia obrazu pre obe oči separátne). Okno má byť rozdelené horizontálne na dve polovice, pričom do ľavej polovice má ísť obraz z ľavej kamery a do pravého okna obraz z pravej kamery.



Ogre3D ponúka ako jedinú možnosť vytvorenie rozdeleného obrazu. To znamená, že je potrebné vytvoriť ďalší tzv. *viewport*, do ktorého sa umiestni kamera, ktorá predstavuje samotný 3D obraz vhodne pootočený pre konkrétne oko.

Návrh riešenia a opis implementácie

Ako riešenie boli vytvorené nové konfiguračné parametre:

- **visualization.split_screen** - parameter bude typu *bool* a bude mať hodnotu *true* ak sa bude vykresľovať vo formáte rozdelenej obrazovky pre zdvojený obraz.
- **visualization.split_screen_offset** - parameter je typu *float* a udáva vzdialenosť kamier.

Podľa toho hodnoty parametra *visualization.split_screen* sa zavolá metóda **OgreVisualization::createCameraSplitScreen** (ak ide o zdvojený obraz) alebo metóda **OgreVisualization::createStandardCamera**.

11.6 Oprava zasekávania sa agentov

Analýza problému

Niekedy sa stane, že sa agent zasekne aj bez zjavnej príčiny, aj keď sa v jeho okolí žiaden iný agent alebo stena nenachádza. V tomto prípade sa s najväčšou pravdepodobnosťou jedná o chybu v mape, kvôli ktorej sa agent dostane do miesta, ktoré nie je pokryté navigačnou sieťou (navigation mesh) a tým pádom nemá spôsob, ako by určil smer ďalšieho pohybu.

Iná možnosť ako môže dôjsť k zaseknutiu je prípad, keď sa veľmi veľa agentov nachádza v stiesnenom priestore. Agent ktorý by mohol teoreticky oblasť opustiť, je ale obklopený ostatnými agentmi tak, že výsledný smer jeho pohybu smeruje do steny. Tieto agenty sa tiež nemôžu pohnúť, vzhľadom na ďalšie agenty ktoré sú na nich natlačené atd., atd. Vzniká akýsi dead lock kde sa skupina agentov zasekne napr. v úzkom priechode.

Návrh riešenia

Pre prvý spomínaný prípad s chybnou mapou bude treba podrobne prekontrolovať mapu a nájsť a opraviť miesta, ktoré nie sú pokryté navigačnou sieťou.

V druhom prípade bude treba nastaviť váhy *steering vectors* tak, aby sa predišlo *dead locku* a aby prioritný smer bol smer sledovania cesty aj v prípade, keď sa agent nachádza pri stene alebo druhom agentovi. Tento zásah by však spôsobil v podstate nefunkčnosť *steering behaviours* vo všetkých ostatných situáciách, kde má prevládať pravé vyhýbanie sa agentom a prekážkam nad sledovaním cesty. Navrhujeme teda čiastočné upravenie váh *steering vectors* v prospech sledovania cesty, ale iba do tej

miery, aby to nepoškodilo správanie sa agenta v ostatných situáciách, ale zároveň znížilo počet výskytov skupinového sa zaseknutia agentov.

Opis implementácie

Na mape 08_bigmap_mesh sme naozaj našli jedno miesto, ktoré nebolo pokryté navigačnou sieťou. Toto miesto sme následne opravili. Po tejto úprave sa na danom mieste už agenty nezasekávali. Pri probléme skupinového sa zaseknutia agentov sme váhy upravili tak, ako sme popísali v návrhu riešenia.

Kapitola 12

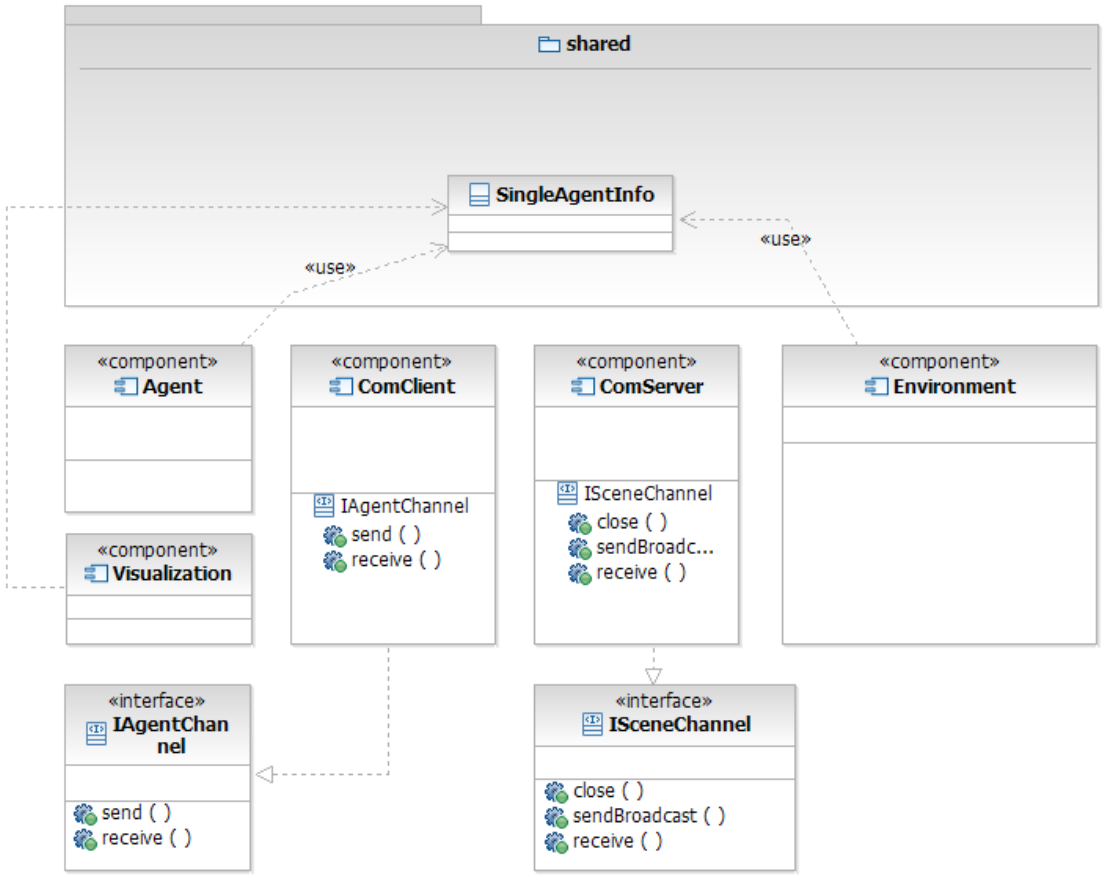
Opis prototypu

12.1 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 12.1.



Obr. 12.1: Základné komponenty prototypu

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.1.4 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.

Kapitola 13

Zhrnutie

Cieľom práce na tomto projekte bolo vytvorenie nástroja na simuláciu davu, ktorý by spĺňal niekoľko podmienok, ktoré boli stanovené samotným zadaním, vedúcim projektu ale aj členmi nášho tímu za účelom vytvorenia kvalitného produktu. Nástroj by mal byť schopný simulovať veľké množstvo ľudí v krajných situáciách ako sú evakuácie z uzavretých priestorov. Simulácia by mala byť čo najvernejším obrazom reálneho správania ľudí vo veľkých davoch. Problematika správania ľudí má niekoľko osvedčených postupov, ktoré sú aktuálne využívané v profesionálnych nástrojoch. Našim cieľom bolo naštudovať čo najviac existujúcich riešení, zobrať ich najzaujímavejšie črty a vytvoriť ich vhodnú kombináciu. Od začiatku sme chceli zaujať nie len komplexným modelom správania, ale aj vizuálnou stránkou situácie - preto bola ďalším cieľom aj konverzia simulácie do 3D zobrazenia. Navyše sme sa rozhodli všetky výpočty aj renderovanie grafickej časti simulácie vykonávať v reálnom čase, na rozdiel od veľkého počtu ostatných nástrojov, ktoré si celú simuláciu vypočítajú ešte pred vykresľovaním. Takýto cieľ kladie ohromné nároky na výpočtovú silu hosťovského počítača. Z toho dôvodu mala byť architektúra aplikácie od samého počiatku navrhovaná tak, aby bolo v neskorších fázach vývoja možné využiť distribuované počítanie.

Existujúce nástroje na simulácie, z ktorých je drvivá väčšina veľmi drahá, sú výsledkom dlhoročného vývoja. Preto boli naše plány veľmi ambiciózne. Aj napriek tomu sa nám podarilo splniť väčšinu vytýčených cieľov na veľmi uspokojivej úrovni.

13.1 Výsledný model správania

Simulácii davu sa venovalo už mnoho vedeckých tímov, alebo komerčných subjektov, z čoho vznikol veľký počet rôznych modelov. Tieto modely sa vo všeobecnosti delia na dve základné skupiny a to:

- makroskopický model

- mikroskopický model

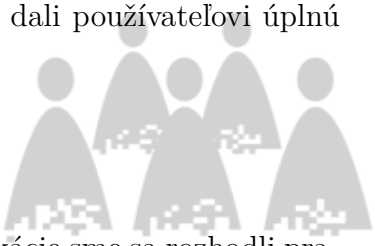
Starší, makroskopický model sa pozerá na dav ako na celok. Toto ponímanie davu bolo zvolené kvôli nedostatku výpočtových prostriedkov v dobe, kedy vznikali modely. Simulácia je vhodná pre veľké počty ľudí, kde nie je potrebný detailný model ľudského správania. Typicky je dav modelovaný ako prietok kvapaliny v prostredí.

Mikroskopické modely na rozdieľ od makroskopických, berie každého človeka ako samostatného agenta, schopného autonómneho správania a rozhodovania. Prístup je vhodný na detailné modelovanie ľudského správania, avšak medzi jeho hlavnú nevýhodu patrí vysoká výpočtová náročnosť.

Pre dosiahnutie čo najlepších výsledkov z pohľadu reálnosti, ale aj výkonu, sme sa podobne ako najkvalitnejšie simulačné nástroje rozhodli skĺbiť dobré vlastnosti obidvoch prístupov. Na mikroskopickej úrovni je individuálne správanie sa agenta riadené pomocou stavového automatu. Vďaka jeho použitiu agent dokáže jednoducho reagovať na aktuálne dianie okolo seba a podľa zvolených podmienok prechádzať medzi zvolenými stavmi. Agent, ktorý sa pôvodne nachádzal v stave kľudu, si v prípade nebezpečenstva vyhľadá cestu k najbližšiemu východu pomocou globálneho vyhľadania cesty. Po úspešenom vyhľadaní cesty začne utekať z budovy, kde jeho rýchlosť je ovplyvnená aktuálnou hodnotou strachu. V prípade že pri utekaní stretne iných agentov, nastraší ich správa o nebezpečenstve, vďaka čomu začnú aj oni utekať.

Aby sme docielili realistickú interakciu s veľkým množstvom ľudí v bezprostrednom okolí indivídua, rozhodli sme sa jeho pohyb dynamicky ovplyvňovať na základe pohybu okolitých ľudí. Tento vzájomne ovplyvnený pohyb podobný pohybu tekutín (makroskopický model) dopĺňa náš model o ďalší aspekt správania - tzv. davová psychózu, pri ktorej sú správanie a rozhodnutia jednotlivca ovplyvnené okolitou masou ľudí.

Všetky parametre správania a pohybu ľudí v rámci simulácie sa dajú meniť pred každou simuláciou pomocou konfiguračného súboru. Tým sme dali používateľovi úplnú kontrolu nad simuláciou.



13.2 Mapy

Pre dosiahnutie čo najväčšej možnej použiteľnosti našej aplikácie sme sa rozhodli pracovať s mapami vo formáte dxf, ktoré predstavujú štandardný formát na modelovanie interiérov aj exteriérov v prostredí CAD.
Aby mohli byť mapy použité na simuláciu, potrebujú aj dodatočnú úpravu. Okrem

pôdorysov budovy je potrebné simulácii poskytnúť niekoľko nových vrstiev, ktoré obsahujú informácie o núdzových východoch a navigačných prvkoch, ktoré využívajú simulovaní ľudia pri pokuse o úniku do bezpečia pri evakuácii.

Po načítaní mapy do programu sa vytvorí grafová štruktúra priestoru nazývaná navigačná mesh. Keďže vyhľadanie najkratšej únikovej cesty je veľmi drahou operáciou, rozhodli sme sa mapu pre agentov predpočítať pomocou upraveného algoritmu Q-Learning. Vďaka jeho použitiu je vyhľadanie únikovej cesty lacnou operáciou. Bohužiaľ vyhľadanie cesty sa nedá kompletne zrealizovať cez tento algoritmus, keďže sa v mape môže vyskytnúť nepriechodná prekážka - napr. oheň. V týchto prípadoch sa pri neúspešnom vyhľadaní použije algoritmus A*, ktorý v prípade že existuje bezpečná cesta tak ju nájde. Ak takáto cesta neexistuje, agent uteká čo najďalej od nebezpečenstva.

13.3 Vizualizácia

Výsledný produkt ponúka pre užívateľa niekoľko možností grafického výstupu medzi ktoré patrí 2D vizualizácia, 3D vizualizácia a reálna 3D vizualizácia. 2D vizualizácia nie je veľmi efektná, ale na druhú stranu je na niektoré účely vhodnejšia, prehľadnejšia a v neposlednom rade aj omnoho menej náročná na hosťovský počítač - čo umožňuje spúšťanie simulácie aj na starších počítačoch.

3D vizualizácia predstavuje veľmi efektný spôsob vyobrazenia evakuačných situácii, pri ktorom môže používateľ ľubovoľne pohybovať kamerou po celej mape. Tento druh zobrazenia je možné pustiť aj do reálneho 3D v ktorom je vizuálizácia zdvojená a pri použití špecializovaného zobrazovacieho zariadenia (napr. 3D TV) je možné sledovať real time 3D simuláciu davu.

13.4 Paralelizácia

Na zabezpečenie čo najväčšieho možného počtu agentov v simulácií, sme sa rozhodli využiť možnosti distribuovaného počítania. Túto možnosť sme riešili pomocou existujúceho rozhrania MPI. Simuláciu je v tomto režime jednoducho spustiť (konkrétne prepnutie modulu wrapera v konfigurácií a spustenie programu pomocou programu mpiexec). Simulácia v distribuovanom móde potrebuje minimálne 4 procesy pre úspešné spustenie, z hora je počet procesov teoreticky neobmedzený. Minimálny počet je potrebný z dôvodu samostatného spustenia modulov v procesoch - prvý proces wrapper, druhý pre modul prostredia, tretí pre modul vizualizácie a posledný pre modul agentov. V prípade zvolenia väčšieho počtu modulov sa vytvorí viacero agent

modulov (keďže agent modul má najväčšie výpočtové nároky), do ktorých sú následne rozdelení agenti.

13.5 Možnosti rozšírenia

Pri podobných nástrojoch je veľa možností na zlepšovanie a rozširovanie a to najmä v modeli správania ľudí. Žiadny existujúci model nie je natoľko komplexný, aby dokázal úplne presne simulovať niečo tak zložité ako je ľudské správania v krízových situáciách. Postupným dopĺňaním nových aspektov správania a experimentovaním s hodnotami parametrov, ktoré ovplyvňujú jednotlivé črty správania je ale možné dosiahnuť veľmi vysokú úroveň realistickosti. Podobné experimenty ale vyžadujú veľa času, ktorý sme nemohli venovať len správnej konfigurácii parametrov. Každopádne predstavujú jednu z možností ako do budúcnosti zlepšiť simuláciu.

Okrem samotného správania by bolo vhodné rozšíriť podporu máp na exteriérové mapy a viacúrovňové mapy (napr. viacposchodové budovy). Ďalšia možnosť ako zvýšiť hodnotu nášho nástroja je vytvorenie modulu, ktorý by počas simulácie zaznamenával rôzne štatistiky, týkajúce sa simulácie, ako je čas evakuácie, počet evakuovaných ľudí, čas šírenia poplašnej správy medzi ľuďmi a veľa iných.

Okrem funkcionality by bolo vhodné vylepšiť aj kvalitatívne aspekty produktu - najmä výkonnosť. Pre všetky nástroje na simuláciu davov je veľmi dôležité, aby zvládali simulovať čo najväčší počet ľudí. V našom prípade by bolo možné upraviť, pozmeniť a optimalizovať jednotlivé protokoly, ktoré sa používajú na vzájomnú komunikáciu medzi jednotlivými modulmi aplikácie, čo by určite viedlo k zvýšeniu výkonnosti.

13.6 Testovanie

Už na začiatku projektu bola za účelom zefektívnenia a sprehľadnenia testovania a teda aj kvality produktu vytvorená metodika pre tzv. unit testy. Vzhľadom ale ku skutočnosti, že tímové úlohy boli rozdelené väčšinou separátne v zmysle že jednotliví členovia boli zodpovední za tematicky separátne funkcionality systému, unit testy sa postupne vytratili. Retrospektívne môžeme tento fakt zhodnotiť pozitívne, pretože absencia vytvárania unit testov sa neodzrkadlila na kvalite a teda počte vzniknutých chýb.

Samotné testovanie bolo vykonávané priebežne každým členom samostatne. Ku kontrolovanie korektného správania dopomáhala aj prítomnosť kontrolných výpisov (lo-

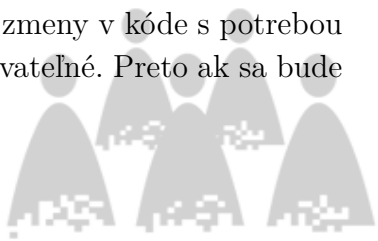
gov), ktoré umožňovali identifikovať aj chyby, ktoré priamo nespôsobovali pád programu simulácie davu. Kontrolné výpisy boli implementované taktiež každým členom v jeho prislúchajúcej oblasti.

Počas vývoja softvéru vzniklo dohromady 11 chýb, ktoré boli natoľko komplexné, že boli predmetom úlohy v šprinte. Aj táto skutočnosť vypovedá o tom, že voľba testovania každým členom samostatne bola dobrá. S určitou samozrejmosťou ešte existujú spôsoby, ktoré by v budúcom pôsobení tímu mohli dopomôcť ku zlepšeniu procesu testovania - ako napríklad spätná kontrola kódu aspoň 1 ďalším členom alebo nasadenie podporných nástrojov monitorujúcich stav operačnej pamäte za účelom predídenia zbytočného zahltenia pamäte.

13.7 Otestovanie MPI paralelizácie pomocou viacerých počítačov

Pre dosiahnutie vyššieho výkonu aplikácie, sme sa rozhodli použiť distribuované počítanie pomocou MPI rozhrania. Už pri 500 agentoch boli badateľné výkonnostné zmeny v porovnaní s MPI simuláciou na jednom počítači. Najvýraznejší rozdiel bol badateľný pri požití 1500 agentov, mapy virtuálnej FIIT a spustení požiaru. Požiar bol umiestnený na takom mieste, aby väčšina agentov musela hľadať cestu pomocou A* algoritmu. Lokálne spustenie MPI paralelizácie vyústovalo do viditeľného zasekávania a FPS pod 1, na rozdiel od MPI s distribuovaným počítaním, kde FPS nekleslo pod 5 ani v čase vyhľadania cesty pre všetkých agentov.

Bohužiaľ veľké počty agentov nebolo možné odsimulovať kvôli neznámej MPI chybe, ktorá je pravdepodobne zapríčinená pretečením buffera komunikácie. Odstránenie tohto problému by bolo možné zmenou jadra programu a rozdelením priestoru na menšie časti, kde by sa každý proces staral len o vlastnú časť a na okrajoch priestoru by sa synchronizoval s susedom o ktorého sa stará iný proces. Druhou možnosťou by bola zmena komunikácie pomocou vlastného protokolu, ktorý by prenášal menší počet dát. Tak či onak tieto zmeny si vyžadujú veľmi veľké zmeny v kóde s potrebou veľkého časového intervalu, takže sú momentálne nezrealizovateľné. Preto ak sa bude na projekte pracovať, tieto otázky by bolo vhodné vyriešiť.



Dodatok A

Používateľská príručka

Inštalácia

Na inštaláciu simulačného nástroja nie je potrebné spúšťať žiadny inštalátor. Stačí rozbaliť súbor *CrowdSimulation.zip* do používateľom zvoleného adresára. Po rozbalení bude daný adresár obsahovať všetky nástroje, dáta a konfiguračné nástroje potrebné pre beh simulácie.

Spustenie simulácie

Po inštalácii bude vytvorená nasledovná adresárová štruktúra:

- *bin*\
  - *Release*\- adresar obsahujúci spúšťací súbor
  - *config.ini* - konfiguračný súbor
- *data*\
  - *maps*\- obsahuje súbory s mapami
  - *materials*\- obsahuje textúry použité pri 3D zobrazení

Pre spustenie simulácie je potrebné spustiť súbor **koreňový adresár inštalácie\bin\release \Wrapper.exe** s parametrom ***–config=../config.ini***. Tento parameter udáva cestu k súboru, ktorý obsahuje aplikačné parametre ovplyvňujúce prebeh simulácie. Používateľ môže tieto parametre ľubovoľne meniť. Podrobný opis jednotlivých parametrov sa nachádza v kapitole **Konfiguračný súbor**.

Pre zobrazenie pomoci je potrebné spustiť súbor **koreňový adresár inštalácie\bin\release \Wrapper.exe** s parametrom ***–help***.

Konfiguračný súbor

Tabuľka A.1 obsahuje zoznam všetkých parametrov, ktoré sa nachádzajú v konfiguračnom súbore - **koreňový adresár inštalácie\bin\config.ini** v poradí, v akom sú v súbore. Sekcie parametrov sú v súbore označené hranatými zátvorkami a majú za úlohu logické delenie parametrov podľa funkcionality. Ak je kolízia v názvoch parametrov, tak je v prvom stĺpci tabuľky navyše označenie sekcie daného parametru. Súbor obsahuje tieto sekcie:

- [**map**] - nastavenia mriežky na mape
- [**grid**] - nastavenia mriežky na mape
- [**agent**] - nastavenia týkajúce sa správania agenta
- [**environmnet**] - nastavenia týkajúce sa prostredia
- [**vizualization**] - nastavenia vizualizácie

Všetky tieto parametre môže používateľ zmeniť. Aby sa zmeny prejavili, je potrebné spustiť novú inštanciu aplikácie.

Tabuľka A.1: Zoznam parametrov a ich význam

Názov	**Opis**	**Východzia hodnota**
file	cesta k súboru s mapou	-
[*grid*]size	udáva veľkosť mriežky na mape	2
[*agent*]dll_path	cesta ku knižniciam agenta	Agent.dll
use_avoidance	prepínač používania	true
use_separation	prepínač používania	true
use_flowfield	prepínač používania flow field	true
width	veľkosť polomeru agenta	0.4
stop_walk_angle	uhol natočenia, pri ktorom agent zastane	1.0
slow_walk_angle	uhol natočenia, pri ktorom agent spomalí	0.4
max_angle_per_frame	maximálny uhol o ktorý sa agent v jednom kroku môže otočiť	0.2
max_step_size	veľkosť maximálneho kroku agenta v stave evakuácie	1
max_slow_step_size	veľkosť minimálneho krok agenta v stave evakuácie	0.5
separation_dist	vzdialenosť, pri ktorej sa agent snaží vzdialiť od ostatných agentov	0.25

avoidance_dist	vzdialenosť, pri ktorej sa agent snaží vzdialiť od stien	0.2
separation_min_dist	minimálna povolená veľkosť vektoru, ktorý sa používa na vychýlenie agenta pri vyhýbaní iným agentom	0.4
avoidance_min_dist	minimálna povolená veľkosť vektoru, ktorý sa používa na vychýlenie agenta pri vyhýbaní stenám	0.4
path_following_weight	koeficient, ktorý určuje, nakoľko striktne agent sleduje naplánovanú cestu	0.6
separation_weight	váha určujúca, ako veľmi sa snaží vyhýbať od ostatných agentov	0.3
avoidance_weight	váha určujúca, ako veľmi sa snaží vyhýbať od stien	0.3
flow_field_weight	koeficient, ktorý určuje, ako veľmi sa prejavuje kohézia agentov	0.3
flowfield_probability	koeficient, ktorý určuje pravdepodobnosť, že agent pôjde s davom	0.7
calm_max_step_size	veľkosť maximálneho krok v pokojnom stave	0.5
calm_max_slow_step_size	veľkosť minimálneho kroku v pokojnom stave	0.2
stay_in_mesh_probability	určuje, s akou pravdepodobnosťou sa agent rozhodne prejsť do inej miestnosti, keď je v pokojnom stave	0.5
[*environment*]dll_path	cesta ku knižnici prostredia	Environment.dll
action_timeout	čas, za ktorý sa ignorujú ešte nespracované správy medzi modulmi	1000
agent_step_size_ratio	hodnota usmerňujúca rýchlosť agentov v kontexte s veľkosťou prostredia	0.1
use_agent_agent_collision	prepínač, určujúci, či sa bude detekovať kolízia agentov	true
agents_count	počet agentov na mape	1000
fire_node_index	určuje, v ktorej miestnosti vypukne požiar	8
alarm_timeout	určuje koľko sekúnd po začiatku simulácie vypukne alarm	10
fire_path_cost	určuje zvýšenie ceny uzla, kde vypukol požiar (pri plánovaní únikovej cesty)	1000

fire_distribution_power	určuje ako veľmi sa zvýšenie ceny uzla s požiarom rozšíri do ďalších uzlov na mape	0.5
save_fire_cost	určuje aká cena uzla je príliš veľká, aby ním agent nechcel prejsť	1
[*vizualization*]dll_path	cesta ku knižnici vizualizácie	Visualization.dll
resolution_x	horizontálny rozmer okna simulácie v pixeloch	900
resolution_y	vertikálny rozmer okna simulácie v pixeloch	600

Dodatok B

Príručka vývojára

Cieľom tejto kapitoly je spolu s prezentovaním uvedených technológií, ktoré boli použité pri vývoji aj ponúknuť ucelenú sumarizáciu, ako postupovať v prípade, že by jednotlivec, prípadne iný timový projekt chcel pokračovať vo vývoji tohto projektu.

B.1 Visual Studio

Vzhľadom na to, že sme sa rozhodli riešenie realizovať a optimalizovať iba na platformu Microsoft Windows, ideálnou voľbou ako vývojového prostredia bolo Microsoft Visual Studio, konkrétne vo verzii 2010. V rámci neho bolo relatívne jednoduché doinštalovať prídavné moduly či už pre komunikáciu s verziovacím systémom (AnkhSVN) alebo pre zjednodušenie práce s programovým kódom (Visual Assist).

B.1.1 Otvorenie a spustenie projektu

Projekt je z dôvodu modulárnosti a paralelizácie organizovaný do samostatných častí pre jednotlivé komponenty (agent, prostredie, vizualizácia, wrapper, mpiwrapper). Všetky spomínané častí možno otvoriť v prostredí pomocou *CrowdSimulation.sln* nachádzajúceho sa v podadresári *src/prototypes/* hlavného adresára projektu.

Aby bolo možné spúšťať aplikáciu priamo z prostredia Visual Studia, je potrebné okrem nadefinovania prístupových ciest ku konkrétnym knižniciam (opísané ďalej) otvoriť v rámci pohľadu (angl. *View*) *Solution explorer Properties* a buď pre *Debug* alebo *Release* v nastaveniach *Configuration Properties* nadefinovať v sekcii *Debugging*:

- pre pole *Command Arguments* hodnotu *"–config ../config.ini"*
- pre pole *Working Directory* hodnotu "$(TargetDir)"

B.2 Graphics Rendering Engine - OGRE

Simuláciu davu je možné pozorovať v dvoch režimoch - dvojrozmernom a trojrozmernom. Uvedený grafický engine OGRE je nutné inštalovať a konfigurovať iba v prípade 3D variantu. Ak vývojárovi postačuje dvojrozmerný pohľad na mapu s pohybujúcimi sa agentami, môže túto sekciu vynechať a pokračovať tou nasledujúcou.

Prvým krokom je stiahnutie príslušnej verzie OGRE pre Microsoft Visual Studio 2010 (my sme použili inštalačný súbor zo stránky `http://sourceforge.net/projects/ogre/files/ogre/1.7/OgreSDK\_vc10\_v1-7-4.exe/download`). Po jeho rozbalení je potrebné skopírovať všetky dynamické knižnice (*.dll) a binárne súbory do adresára `bin\Debug`, resp. `bin\Release` projektu simulácie davu (prítomnosť spomínaných knižníc je kľúčová pre celkový chod aplikácie). Jedná sa všetky súbory v adresári `OGRE_ROOT/bin/Debug`, resp. `OGRE_ROOT/bin/Release`. Okrem toho je vhodné nastaviť systémovú premennú OGRE_ROOT v operačnom systéme (v našom prípade bola jej hodnota `C:\ProgramFiles(x86)\OgreSDK_vc10_v1-7-4\`).

Následne je nutné vykonať nastavenie požadovaných ciest boost knižníc a zdrojových súborov v prostredí MS Visual Studio. V rámci pohľadu (angl. *View*) *Property Manager* je potrebné buď pre *Debug* alebo *Release* v nastaveniach *Microsoft.cpp.win32.user* nadefinovať nasledovné :

- VC++ Directories -> Include Directories -> pridať `OGRE_ROOT/include/OGRE`
- VC++ Directories -> Include Directories -> pridať `OGRE_ROOT/include/OIS`
- VC++ Directories -> Library Directories -> pridať `OGRE_ROOT/lib/debug`
- VC++ Directories -> Library Directories -> pridať `OGRE_ROOT/lib/release`
- VC++ Directories -> Executable Directories -> pridať `OGRE_ROOT/lib`

B.2.1 Knižnica d3dx9_42.dll

Ak by sa vyskytol problém pri spúšťaní aplikácie a problémom by bolo, že nie je možné nájsť knižnicu *d3dx9_42.dll*, je nutné ju stiahnuť (napr. z odkazu `http://www.dll-files.com/d3dx9_42.zip?0WFfPIXEhT`) a nakopírovať ju do adresára `bin\Debug`, resp. `bin\Release` projektu simulácie davu.

B.3 MPI

V rámci paralelizácie funkcionality bolo riešenie simulovania davu rozdistribuované medzi viaceré moduly (procesy), ktoré medzi sebou musia komunikovať prostredníctvom posielania správ (na to slúži MPI - Message Passing Interface). Túto komunikáciu sme zabezpečili konkrétnym riešením MPICH2. Inštalačný súbor je možné stiahnuť z adresy `http://www.mcs.anl.gov/research/projects/mpich2/downloads/tarballs/1.4.1p1/mpich2-1.4.1p1-win-ia32.msi`.

Z hľadiska zvýšenia prehľadnosti je vhodné nastaviť systémovú premennú MPICH_ROOT v operačnom systéme (v našom prípade bola jej hodnota `C:\ProgramFiles(x86)\MPICH2\`).

Následne je nutné vykonať nastavenie požadovaných ciest boost knižníc a zdrojových súborov v prostredí MS Visual Studio. V rámci pohľadu (angl. *View*) *Property Manager* je potrebné buď pre *Debug* alebo *Release* v nastaveniach *Microsoft.cpp.win32.user* nadefinovať nasledovné :

- VC++ Directories -> Include Directories -> pridať `MPICH_ROOT/include`
- VC++ Directories -> Library Directories -> pridať `MPICH_ROOT/lib`
- VC++ Directories -> Executable Directories -> pridať `MPICH_ROOT/lib`

Ďalšia konfigurácia je uvedená v nasledujúcej sekcii Boost, pretože vyžaduje prítomnosť iných konfiguračných súborov.

B.4 Boost

Boost predstavuje kolekciu knižníc obsahujúcich funkcionalitu pre efektívnejšiu a z hľadiska vývojára zjednodušenú prácu z oblastí programových štruktúr (napr. v Jave veľmi dobre známych kolekcií), práce s vláknami (angl. multithreading), rôznymi pokročilými matematickými operáciami alebo jednotkovými testami (angl. unit tests). Jeho funkcionalitu sme využili aj pri implementácii distribúcie funncionality do viacerých modulov schopných fungovať oddelene.
Počas vývoja sme jeho funkcionalitu využívali v rámci všetkých vyvíjaných modulov (agent, prostredie, wrapper aj vizualizácia), vďaka čomu sme sa odľahčili od riešenia nízkoúrovňovej funkcionality a naše zameranie sme mohli koncentrovať iným smerom.

Boost možno stiahnuť napríklad z odkazu `http://sourceforge.net/projects/boost/files/boost/1.48.0/boost_1_48_0.zip/download`. Po rozbalení archívu je dostupná nasledovná štruktúra:

- *index.htm* - kópia www.boost.org HTML stránky pre úvodné pokyny
- *boost* - všetky boost hlavičkové súbory
- *lib* - predkompilované binárne súbory (predvolene prázdny adresár)
- *libs* - testy, .cpp súbory, dokumenty, atď.
- *index.html* - dostupná dokumentácia
- *status* - boost testovacia sada
- *tool* - užitočné súčasti ako napr. Boost.Build, quickbook
- *more* - dokumenty ohľadom bezpečnosti
- *doc* - dodatočný adresár pre všetku boost dokumentáciu
- *bootstrap.bat* - skript pre vytvorenie chýbajúcich predkompilovaných binárnych súborov

B.4.1 Dodatočná konfigurácia MPI

Aby sme mohli využívať MPI, je potrebné v rámci rozbaleného adresára *boost* v podadresári */tools/build/v2* pripojiť na koniec súboru *user-config.jam* nasledujúci text:

```
using mpi : :
<find-static-library>mpi
<library-path>“C:/Program Files (x86)/MPICH2/lib”
<include>“C:/Program Files (x86)/MPICH2/include”
:
“\“C:\\Program Files (x86)\\MPICH2\\bin\\mpiexec\\””;
```

B.4.2 Dokončenie konfigurácie Boost

Z hľadiska zvýšenia prehľadnosti je vhodné nastaviť systémovú premennú BOOST_ROOT v operačnom systéme (v našom prípade bola jej hodnota `C:\ProgramFiles(x86)\boost\boost_1_48_0`).

Aby boli súbory používané pri vývoji kompletné, je nutné dodatočne spustiť skript pre vytvorenie chýbajúcich prekompilovaných binárnych súborov (knižníc). Nachádza sa v hlavnej adresárovej štruktúre rozbaleného archívu a jeho názov je *bootstrap.bat*. Pre kompletizáciu inštalácie je nutné v príkazovom riadku vykonať nasledovný príkaz:

b2.exe –libdir=BOOST_ROOT/lib –includedir=BOOST_ROOT/include –build-type=complete link=shared runtime-link=shared threading=multi –with_mpi

ako aj príkaz

bjam.exe

Následne je nutné vykonať nastavenie požadovaných ciest boost knižníc a zdrojových súborov v prostredí MS Visual Studio. V rámci pohľadu (angl. *View*) *Property Manager* je potrebné buď pre *Debug* alebo *Release* v nastaveniach *Microsoft.cpp.win32.user* nadefinovať nasledovné :

- VC++ Directories -> Include Directories -> pridať `BOOST_ROOT/include`
- VC++ Directories -> Library Directories -> pridať `BOOST_ROOT/lib`
- VC++ Directories -> Executable Directories -> pridať `BOOST_ROOT/lib`

Boost je po vykonaní spomínaných krokov pripravený na použitie.