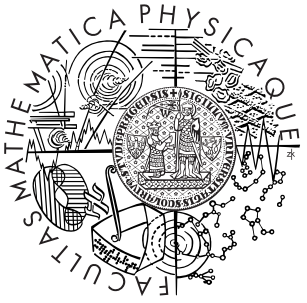
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Jan Stria

Simulace davu

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán
Studijní program: Informatika, obecná informatika

2008

Rád bych poděkoval RNDr. Josefu Pelikánovi za vedení ročníkového projektu a bakalářské práce a za poskytování cenných připomínek k nim.

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

V Letovicích dne 8. 8. 2008 Jan Stria

Obsah

Název práce: Simulace davu
Autor: Jan Stria
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
E-mail vedoucího: josef.pelikan@mff.cuni.cz

Abstrakt: V předložené práci studujeme možnosti simulace davu velkého množství chodců v reálném čase. Nejdříve představíme některé již používané modely simulace a vzájemně je porovnáme. Na jejich základě potom navrhneme model vlastní. Podrobně popíšeme hierarchickou reprezentaci světa, jež umožňuje simulovaným chodcům efektivně zjišťovat informace o něm. Představíme si algoritmy realizující plánování cest v tomto světě a algoritmy jednoduše simulující zrak virtuálních chodců. Dále studujeme možnosti paralelizace simulace a popíšeme jednoduchý vrstevnatý model chování chodců založený na řídících silách a zásobníku cílů. Nakonec se budeme zabývat některými implementačními detaily navrženého systému.
Klíčová slova: simulace davu, virtuální svět, paralelizace

Title: Crowd simulation
Author: Jan Stria
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: josef.pelikan@mff.cuni.cz

Abstract: In this work we study possibilities of simulating populous crowd of pedestrians in a real time. At first we present available models of simulation comparing them with each other. In terms of these models we design our own model. In detail we describe a hierarchical representation of a virtual world which makes an efficient answering of pedestrians' queries possible. We present algorithms realizing paths planning and simulating simple visual perception system of pedestrians. Next we study possibilities of parallelizing the simulation and then we describe a simple layered model of pedestrians' behavior based on steering forces and a stack of goals. In the end we consider some implementation details of the presented system.
Keywords: crowd simulation, virtual world, parallelization

Kapitola 1

Úvod

1.1 Stručná historie studia a simulace davu

Počátky výzkumu zabývajícího se davem sahají do druhé poloviny 19. století a jsou spjaty se jménem francouzského sociologa G. Le Bona. Ten ve své nejznámější a dnes již klasické práci [10] definuje dav jako „shromáždění jakýchkoliv jedinců bez ohledu na jejich národnost, povolání nebo pohlaví a bez ohledu na náhodu, která je svedla dohromady.“ Le Bon se zabývá především vznikem nových vlastností charakteristických pro dav, které jsou odlišné od vlastností jednotlivců jej tvořících a vycházejí z vrozeného chování a pudů. Davem z psychologického a sociologického hlediska se dále zabývali S. Freud, K. Young, R.W. Brown a další. Jejich myšlenky jsou sice zajímavé, pro simulaci davu však takřka nemají uplatnění, jsou totiž až příliš psychologicky zaměřené.

Mnoho významných prací zkoumajících dav vzniklo v 70. letech 20. stol. Velice přínosný byl zejména výzkum E. Goffmana [6], který se zabýval tím, jak si jednotliví chodci prohlíží své okolí, aby zabránili kolizím. Ukázal, že největší pozornost věnují objektům ve své blízkosti. Wollf v práci [30] zase popisuje, jakým způsobem se chodci mezi sebou vyhýbají v závislosti na vzájemné vzdálenosti a hustotě davu. Tímto problémem se dále zabývají také Sobel a Lillith [27]. Rozdíl oproti dřívějším pracím Le Bona a spol. je především v tom, že tyto výzkumy jsou již velmi platné pro simulaci pohybu chodců. Dávají totiž konkrétní „návody“, jak se lidé v davu chovají.

První pokusy simulovat dav pomocí počítačů potom datujeme na konec 70. let 20. století, kdy byl představen jednoduchý model pohybu chodců založený na magnetických silách [14]. V 80. letech se simulací hejna ptáků [17], kterou později rozšířil i na chodce, automobily a pohybující se objekty obecně [18], zabýval C. W. Reynolds. Zhruba ze stejné doby jako jeho rané práce pochází i buněčný model P. G. Gippse a B. Marksja [5].

V 90. letech potom začíná období velkého zájmu o simulaci davu, které trvá až dodnes. Vzniká celá řada stále složitějších a dokonalejších modelů, které se kromě prosté simulace pohybu jedinců v davu zabývají také napodobováním jejich vyššího chování, procesu rozhodování a výběru cílů, jejichž dosažením agenti splňují své potřeby. Začínají se čím dál tím více uplatňovat metody umělé inteligence. Simulace se dostává do popředí zájmu počítačových grafiků, mnoho prací se např. zabývá využitím dat získaných technikou motion-capture a jejich syntézou.

V poslední době navíc simulace davu překračuje hranice akademické půdy a vzniká celá řada komerčně úspěšných produktů. To souvisí mimo jiné také s neustálým nárůstem výkonu počítačů, Ty umožňují simulovat v reálném čase davy čítající tisíce jedinců, či naopak generovat velmi reálné výstupy pro potřeby filmových triků, byť za cenu dlouho trvajících výpočtů.

1.2 Proč simulovat dav?

Téměř všechny systémy, které mají za úkol simulovat dav lidí, lze na základě jejich účelu rozdělit do dvou kategorií. První z nich tvoří aplikace pro potřeby urbanistického modelování, druhou potom programy využívané v zábavním průmyslu.

1.2.1 Urbanistická simulace

Existence první kategorie simulátorů souvisí především s rozvojem městské zástavby. Na území měst, zejména těch velkých, totiž vznikají rozsáhlé a komplikované veřejně přístupné prostory, jako jsou obchodní domy, sportovní stadiony, vlaková nádraží, letiště apod. V těchto objektech se většinou pohybuje obrovské množství lidí, kteří navíc nejsou nijak centrálně řízeni. Jejich pohyb je možno považovat do určité míry za chaotický a nepředvídatelný.

Fakt, že se v objektu bude pohybovat velké množství lidí, je třeba zohlednit již v době jeho návrhu. Ukazuje se, že např. jeden špatně umístěný sloup v chodbě vedoucí do metra může mít negativní vliv na plynulost pohybu chodců. Doslova každý centimetr zde rozhoduje o tom, zda se budou či nebudou tvořit fronty. Také umístění nouzových východů v takových objektech a tvorbě evakuačních plánů je třeba věnovat zvýšenou pozornost. V minulosti došlo již k mnoha katastrofám, které si vyžádaly spoustu obětí a jejichž společným jmenovatelem byl početný dav [28], a to především na fotbalových stadionech a při velkých náboženských shromážděních.

Systémy zabývající se simulací davu z pohledu urbanistického modelování většinou nekladou velký důraz na grafický výstup. Mnohem podstatnější je, aby použitý model co nejlépe odpovídal realitě. Proto se výstupy simulace často testují oproti datům získaným např. pomocí kamery. Důležité bývá také přehledné zpracování výsledků ve formě statistik určujících, jaká byla průměrná rychlost chodců, jak se měnila hustota zalidnění jednotlivých částí daného prostoru apod.

Nejznámějšími produkty této kategorie jsou systémy Legion [36], DI-Guy [35] a SimWalk [38]. Jedná se většinou o komerční softwarové balíky, často velmi drahé. Modely použité pro simulaci bývají firemním tajemstvím a je tedy těžké o nich cokoliv podrobnějšího zjistit.

1.2.2 Simulace v zábavním průmyslu

Druhou skupinu tvoří systémy pro potřeby zábavního průmyslu, především filmů a počítačových her. Jejich společnou charakteristikou je kladení důrazu na výsledný grafický výstup. Pohyby agentů musí působit realisticky; především u filmových nástrojů

by neměl být poznat rozdíl mezi agentem a skutečným člověkem. Proto se také pro znázornění základních pohybů často využívá dat získaných technikou motion-capture.

Jedním z nejznámějších a komerčně nejúspěšnějších programů je Massive [37], který se používá zejména ve filmovém průmyslu. Je zaměřen na animátory, proto umožňuje snadno vytvářet logiku agentů prostou kombinací uzlů reprezentujích jejich jednotlivé reakce na vnější podměty. Simulace zde neprobíhá v reálném čase a výstup může být dále editován grafiky. Systém Massive byl v poslední době použit při tvorbě snímků, jako jsou Pán prstenů, Mumie či Letopisy Narnie.

Dalším známými nástroji jsou produkty společnosti SpirOps [39] určené především pro potřeby návrhu umělé inteligence a simulace davů v počítačových hrách. Byly použity při tvorbě hry Splinter Cell. Nakonec zmiňme nástroje ve formě doplňků pro 3D modelovací software. Jde o CrowdIT [34] určený pro 3D Studio MAX a o bezplatně šířený Blender People [33] pro editor Blender.

1.3 Jak simulovat dav?

Existují v zásadě dvě možnosti, jak simulovat dav lidí. První, jednodušší možností je částicová simulace, druhou možností, která umožňuje dosáhnout lepších výsledků, potom simulace založená na metodách umělé inteligence.

1.3.1 Částicová simulace

Při částicové simulaci ja každému simulovanému chodci přiřazen hmotný bod, jehož pohyb je následně ovlivňován působením nejrůznějších sil. Výhodou je, že tímto způsobem je možno simulovat velké množství lidí, výpočty sil většinou nebývají příliš složité. Na druhou stranu se tento přístup nehodí v případě, kdy je požadován realisticky vypadající výstup.

Příkladem částicového přístupu je model založený na magnetických silách [14] (magnetic force model) napodobující působení skutečných elektromagnetických sil. Každá částice reprezentující člověka, stejně tak i každá překážka dostanou přidělen kladný náboj, cíle, kterých se chodci snaží dosáhnout, potom mají náboj záporný. Souhlasné náboje se přitom odpuzují, opačné se navzájem přitahují. Výslednou sílu působící na každou částici a určující její zrychlení získáme součtem dvojice sil. První z nich je určena obdobou Coulombova zákona, známého z fyziky, a závisí na intenzitě nábojů a druhé mocnině vzálenosti vyhodnocovaných objektů (dvojice částice-částice, částice-překážka či částice-cíl). Druhá síla zajišťuje aby se chodec otáčel směrem od překážek či sousedních částic, s nimiž mu hrozí kolize, a závisí na jeho rychlosti a úhlu, který vektor rychlosti se sousedními objekty svírá.

Dalším modelem této kategorie je buněčný model (cellular model) popsaný v [5]. Každá z částic představující chodce je umístěna v jedné z buněk dvourozměrné mřížky, v každé buňce se přitom může nacházet nejvýše jedna částice. Každá buňka je navíc kladně ohodnocena na základě své vzdálenosti od cílové buňky a penalizována, jestliže se v jejím okolí nachází buňka obsazená jinou částicí či překážkou. V každém kroku simulace si částice ze svého 8-okolí vybírá tu buňku, která má nejlepší ohodnocení. Tím

je zajištěn pohyb částic směrem k cíli a zároveň se předchází kolizím.

Třetím významným částicovým modelem jsou sociální síly (social forces) představené v [8, 9]. Tento model umožňuje dosáhnout výrazně realističtějších výsledků než předchozí dva. Byl navržen speciálné pro potřeby simulace evakuace budov a jeho specialitou je, že si dokáže dobře poradit i s velmi hustým davem chodců, se kterým mívají ostatní modely někdy problém. Základem je, stejně jako u modelu magnetického, působení sil mezi chodci, překážkami a cíli. Síly jsou vyjádřeny pomocí exponenciálních funkcí, takže jejich hodnoty mohou velmi rychle růst do nekonečna. To takřka vždy zabraňuje kolizím. Pouhým použitím speciálních druhů sil potom model umožňuje simulovat i tak složité jevy, jako je ochota pomáhat při evakuaci slabším jedincům nacházejím se ve stejné sociální skupině (např. v rodině). Na druhou stranu sociální síly fungují uspokojivě pouze v případech, kdy se celý dav pohybuje jedním směrem (k evakuačnímu východu) a pro simulaci běžného davu nekoordinovaných chodců se příliš nehodí.

1.3.2 Simulace založená na umělé inteligenci

Druhou skupinu metod tvoří simulace založené na umělé inteligenci. Lidé v davu bývají často označováni jako agenti, této terminologie se také budeme držet. Pohyb agentů už není řízen pouhými silami, jak tomu bylo u částic, přestože pomocných sil bývá také často využíváno. Agenti však navíc mívají senzory, pomocí kterých vyhodnocují situaci ve svém okolí, sledují různé cíle, plánují si cesty, vybírají si pro danou chvíli nejvhodnější z více možných druhů chování. Složitější modely mohou dokoce simulovat potřeby a emoce agentů, sociální interakce s ostatními agenty apod.

Každý agent má v podstatě něco jako mozek, ať už jednodušší nebo komplexnější, a ten je zodpovědný za jeho činnost. Z toho lze také usoudit, že tyto modely bývají složitější na implementaci než modely částicové. Na druhou stranu je pomocí nich možno dosáhnout velice zajímavých výstupů. Simulace davu pro potřeby zábavního průmyslu využívají téměř výhradně modelů založených na umělé inteligenci.

V článku [17] byl představen jednoduchý model simulace hejna létajících ptáků, nazývaných zde boidi (boids). Svou povahou tento model stojí na pomezí s druhou kategorií, částicovými systémy, využívá totiž také převážně principu sil. Boid má však navíc svou vlastní geometrii, už se nejedná o pouhý hmotný bod. Přestože má každý boid k dispozici informaci o celém prostředí, při výběru svého chování, které už je realizováno právě prostřednictvím sil, bere v potaz pouze boidy ve svém sousedství kulovitého tvaru, lze tedy mluvit o primitivním vnímání. Boid si přitom vybírá ze třech různých druhů chování. Buďto si může udržovat odstup od ostatních boidů (separation), nebo se snažit zarovnat směr svého pohybu s hejnem (alignment), nebo se snažit do okolního hejna začlenit a pomáhat je tak tvořit (cohesion). Zajímavé je, že hejna vznikají zcela spontánně, pouze na základě pohybu jednotlivých boidů.

Model boidů Renolds dále rozpracoval v [18] do složitějšího modelu chování založeného na řízení (steering behaviors). Ten umožňuje simulovat i davy lidí, dopravu apod. Chování založené na řízení bylo použito i v této práci, proto je mu dále věnována samostatná kapitola.

Agenti v psychologickém modelu [21] (psychological model) mají k dispozici množinu

základních pohybů, jako je chůze vpřed, klus, úkrok stranou apod., ze kterých si v každém simulačním kroku jeden vyberou. Pro každého agenta jsou dále definovány dvě kruhové oblasti, ten se pak nachází v jejich společném středu. Jestliže se některý z ostatních agentů ocitne uvnitř první oblasti, s větším poloměrem, zvolí náš agent chůzi směrem od místa potenciální kolize nezměněnou rychlostí. Jestliže se však některý z okolních agentů objeví uvnitř kritické menší oblasti, sníží náš agent prudce svou rychlost, případně zvolí úkrok stranou. Tímto způsobem je simulován určitý psychický stres, kterým na sebe lidé v davu vzájemně působí a který roste s tím, jak se sobě blíží. Navigace agentů je pak realizována pomocí dvourozměrného pole vektorů. Pro každou pozici prostředí, ve kterém simulace probíhá, je v odpovídající buňce pole definován vektor, podle kterého si agenti vybírají směr dalšího pohybu. Všechny tyto vektory vedou směrem k cíli, přičemž obtékají překážky a zabraňují tak agentům v kolizích s nimi. Nevýhodou je, že všichni agenti musí mít stejný cíl. Případně lze vytvořit několik větších skupin jedinců, každou se svým vlastním polem navigačních vektorů. Potom lze snadno simulovat např. pohyb dvou proti sobě jdoucích skupin chodců, jak jej známe ze silničních přechodů.

Další významný model [4] je vystavěn na situačním kalkulu známém z umělé inteligence. Nad tímto kalkulem byl definován jazyk CML (cognitive modelling language). Tím jsou animátoři, pro něž je tento model převážně určen, odkloněni od nutnosti definovat chování agentů přímo pomocí predikátů logiky prvního řádu. Kognitivní modelování navíc stojí ještě o stupínek výše než předchozí modely založené na přímé definici chování agentů. Animátorovi stačí naskriptovat pouze jakýsi nástin toho, jak se má agent chovat, vyhovující detailnější akce potom zvolí model sám pomocí rozhodovacího modulu. Jazyka CML bylo využito k simulaci prehistorického světa obývaného dinosaury a k simulaci světa podmořského.

Jedny z nejlepších výstupů potom generuje model autonomních chodců (autonomous pedestrians) představený v článcích [22, 23, 24]. Vyniká především velmi dobrou hierarchickou reprezentací světa, jež byla s mírnými modifikacemi převzata pro potřeby této práce a kterou popíšeme dále v samostatné kapitole. Ta umožňuje agentům efektivně vnímat své sousedy a překážky v okolí a plánovat cesty na různé úrovni abstrakce. Také model chování agentů je hierarchický. Na nejnižším stupni stojí šest jednoduchých reaktivních rutin zamezujících kolizím. Nad nimi se nachází navigační a motivační vrstva zodpovědná za plánování a výběr cest, procházení úzkých chodeb apod. Nejvýše potom stojí vrstva kognitivní. Jejím úkolem je na základě potřeb a cílů agenta vybírat odpovídající chování z vrstev nižších. Modelu autonomních chodců bylo využito k simulaci davu na již zaniklé Pennsylvania Station v New Yorku. Dostupné výsledné grafické výstupy působí velmi přesvědčivě, k čemuž určitě hodně pomohlo i použití profesionálního nástroje DI-Guy [35] pro zobrazení agentů a jejich základních pohybů.

1.4 Obsah práce

Ve zbytku práce představíme možný návrh kompletního systému simulujícího početný dav chodců, označovaných zde jako agenti. Chodce jsme zvolili z toho důvodu, že jejich simulace je o dost jednodušší než např. simulace rozsáhlé bojové scény. I tak je ale

představený systém dost rozsáhlý a netriviální.

Naše práce patří do kategorie modelů využívajích metod umělé inteligence, přestože vykazuje i některé charakteristiky částicových systémů. Se zařazením do kategorie podle účelu je trošku problém. Blíže je systémům pro urbanistické plánování, jedná se však hlavně o práci experimentální. Cílem nebylo vytvořit model, který lze reálně nasadit v praxi.

Agenti — chodci se pohybují ve virtuálním světě, jehož hierarchická reprezentace je obsahem kapitoly 2. V této kapitole dále představíme mapy umožňující agentům plánovat cesty a mapy podporující vnímání překážek, které se nacházejí v jejich okolí. Zabývat se budeme samozřejmě i algoritmy provádějícími toto plánování a vnímání.

Obsahem kapitoly 3 je popis toho, jak je možné simulaci davu paralelizovat rozdělením agentů mezi různá simulační vlákna. Je zde také uveden způsob komunikace mezi jednotlivými vlákny. Jednotlivá simulační vlákna potom dovolují agentům vnímat chodce v jejich nejbližším okolí. Popis speciálních map, které to umožňují, je také obsahem této kapitoly.

V kapitole 4 jsou rozebrány modely použité pro simulaci pohybu, chování a rozhodování agentů založené na již známých modelech, které byly pro potřeby této práce netriviálně rozšířeny. Postupy zde uvedené využívají algoritmů a struktur představených v kapitolách 3 a 4.

Konečně nejrozsáhlejší kapitola 5 obsahuje popis návrhu a tvorby programu MASS založeného na představeném modelu. Jsou zde důkladně rozebrány implementační detaily některých algoritmů a uvedeny nejzajímavější použité programátorské techniky. Čtenář si zde může udělat velmi dobrou představu o tom, jak vlastně program funguje a jaké problémy je třeba řešit při implementaci tak rozsáhlého a složitého projektu, kterým simulace davu bezpochyby je.

V závěrečné kapitole potom zhodnotíme postupy použité v této práci. Uvedeme jejich výhody a nevýhody a podíváme se i na to, které části modelu by šlo do budoucna vylepšit.

Kapitola 2

Virtuální svět

2.1 Hierarchická reprezentace

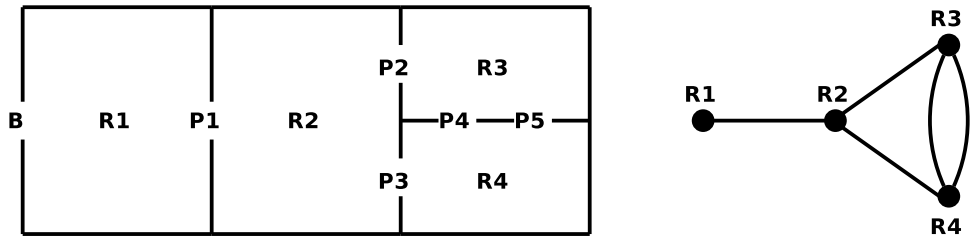
Chceme-li vytvořit systém na simulaci davu, musíme si nejdříve pečlivě zvolit reprezentaci virtuálního světa. Tomuto kroku je třeba věnovat dostatečnou pozornost, protože informace, které budou v příslušných datových strukturách obsaženy, do značné míry určují, co všechno půjde simulovat. Snahou obecně bývá, aby svět umožňoval odpovídat agentovi na dotazy typu: „Vyskytuje se v mém okolí nějaká překážka? A jestliže ano, tak kde přesně?“ Nebo: „Kudy se mám vydat, chci-li dosáhnout dané pozice?“ Případně: „Ve které oblasti světa se nachází ten který objekt?“

Samozřejmě by bylo možné reprezentovat svět čistě geometricky a veškerou jeho interpretaci nechávat pouze na agentech. Ti by pro získání informací potom používali některou z metod počítačového vidění. To by však bylo implementačně i výpočetně velice náročné, přestože by to nejvíce odpovídalo realitě. Právě kvůli efektivitě zodpovídání dotazů agentů je velice výhodné reprezentovat svět hierarchicky. Inspirací tady byla práce [23], kterou jsme rozšířili o myšlenku portálů a bran.

Celé virtuální prostředí je rozděleno do několika obdélníkových oblastí nazývaných regiony. Předpokládá se přitom, že toto rozdělení je definováno již v popisu světa, proto se jím dále nezabýváme. Důležité je, že povrch každého regionu lze považovat za část horizontální roviny a potřebné informace o něm tak uchovávat pomocí 2D struktur.

Mezi jednotlivými regiony mohou agenti přecházet pouze na speciálních místech k tomu určených — ta nazýváme portály. Regiony označujeme jako sousední, existuje-li mezi nimi nějaký portál. Celý virtuální svět je potom možné reprezentovat pomocí neorientovaného multigrafu, jehož vrcholy korespondují s regiony a hrany mezi nimi představují portály. Multigraf je neorientovaný z toho důvodu, že portál je vždy průchozí z obou stran.

Další velice důležitou strukturou jsou tzv. brány. Ty označují místa, kterými agenti vchází do virtuálního světa, nebo jej tudy zase naopak opouštějí. Každá brána se váže k nějakému regionu, na jehož území se nachází, a lze ji vlastně považovat za portál spojující daný region s abstraktním regionem reprezentujícím vše, co leží mimo náš virtuální svět. Podstatný rozdíl proti portálu je potom v tom, že brány mohou být případně orientovány jenom jedním směrem a kromě vstupně-výstupních bodů tak lze simulovat i místa sloužící výhradně jako vchod či výhradně jako východ.



Obrázek 2.1: Vlevo: Hierarchická reprezentace budovy obsahující čtyři místnosti (regiony *R1-R4*), jeden hlavní vchod (brána *B*) a pět průchodů (portály *P1-P5*). Vpravo: Odpovídající multigraf.

Odpovídá-li virtuální svět např. nějaké budově, mohou regiony představovat místnosti, portály reprezentovat dveře či průchody (všimněme si, že mezi dvojicí místností se může nacházet více dveří) a brána potom hlavní vchod do budovy. Nikde však není řečeno, že jednotlivé regiony musí být ohraničené. Stejně tak může virtuální svět odpovídat třeba náměstí, obdélníkové regiony jsou nějaké jeho abstraktní části a portály se roztahují přes celé hrany obdélníků. V extrémním případě může být celý svět tvořen pouze jedním regionem; multigraf potom kolabuje do jediného vrcholu. Obecně lze však říct, že zde představený model světa nejlépe funguje právě pro vnitřní prostory s oddělenými místnostmi.

2.2 Plánování cest mezi regiony

Hierarchická reprezentace světa umožňuje plánovat cesty agentů na různých úrovních. Na nejvyšším stupni abstrakce stojí plánování pohybu mezi jednotlivými regiony. Tady s výhodou využijeme toho, že regiony společně s portály tvoří multigraf. Zadá-li agent dotaz na to, jakým způsobem se lze dostat z regionu, v němž se právě nachází, do určitého regionu cílového, zjistíme tuto informaci z tabulky [24], kterou si na začátku vytvoříme právě z multigrafu.

Tabulka obsahuje pro každou dvojici počátečního P a cílového C regionu množinu *M(P,C)*. Obsahem množiny jsou dvojice *[S,d]*, kde S je sousední region P na cestě do C. Číslo d potom vyjadřuje délku této cesty jako počet portálů, přes které je nutno projít (tj. počet hran v multigrafu). V množině *M(P,C)* jsou přitom pouze takové dvojice *[S,d]*, pro něž je d rovno *min(P,C)* nebo *min(P,C) + 1*, kde *min(P,C)* značí délku nekratší cesty z P do C.

Postup, jak takovou tabulku vytvořit, ukazuje následující algoritmus [24]:

```
1. Inicializace:
   pro každý počáteční region P
      pro každý cílový region C
         je-li P == C
            M(P,C) = {[P,0]}
         jinak
            M(P,C) = {}
```

```
2. Postupné určování cest délky 1..n-1 (n je počet regionů):
   pro všechny délky d = 1..n-1
      pro každý počáteční region P
         pro každý cílový region C
            pro každý region S sousedící s P a libovolný region X
               je-li [X,d-1] v množině M(S,C)
                  přidej [S,d] do M(P,C)

3. Odstranění příliš dlouhých cest:
   pro každý počáteční region P
      pro každý cílový region C
         nechť min(P,C) je minimální vzdálenost v M(P,C)
         pro každou dvojici [X,d] z M(P,C)
            je-li d > min(P,C) + 1
               odstraň dvojici [X,d] z M(P,C)
```

Odstraněním některých dvojic ve třetím kroku algoritmu mimo jiné dosáhneme toho, že tabulka nebude obsahovat cesty s cyklem. Tedy pro virtuální svět z obr. 2.1 nebude v množině *M(R2,R3)* dvojice *[R1,3]*, přestože cesta *R2-R1-R2-R3* délky 3 existuje.

2.3 Plánování cest v rámci regionu

2.3.1 Cesty

Dalším druhem cest, na které se agenti virtuálního prostředí ptají, jsou cesty v rámci jednoho regionu. Parametry takového dotazu jsou počáteční a cílová pozice zadané v kartézských souřadnicích a obsažené v oblasti náležící regionu. Odpovědí je potom cesta ve tvaru uspořádané n-tice bodů. Aby byla cesta platná, nesmí spojnice žádných dvou po sobě jdoucích jejích bodů protínat překážku. To však samo o sobě nestačí; je totiž žádoucí, aby se žádná překážka nevyskytovala ani v těsné blízkosti lomené čáry představující cestu. V reálném světě se lidé při chůzi kolem zdi o ni také bokem neotírají, třebaže by taková cesta snad byla nejkratší.

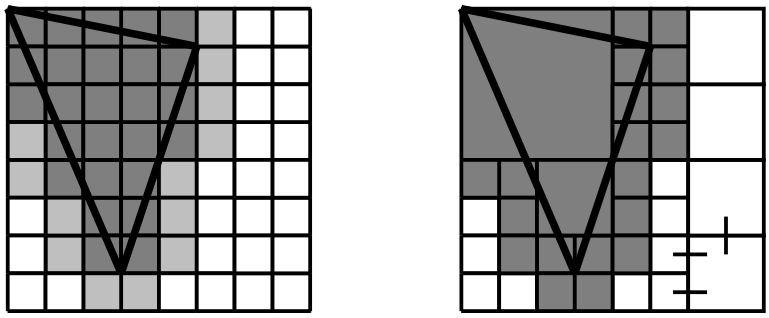
Dále si je třeba uvědomit, že cesty jsou voleny tak, aby se agent při jejich sledování vyhnul statickým překážkám, ne však kolizím s ostatními agenty. Při plánování cest na delší vzdálenosti (řádově desítky metrů) by to ani nebylo dost dobře možné, protože pohyb ostatních agentů nelze takto dopředu předpovídat. Body cesty dávají spíše jenom letmý návod, jak se regionem pohybovat, zbytek už je ponechán na inteligenci agentů. Tento přístup také poměrně dobře odpovídá realitě, kdy člověk sice sleduje při chůzi nějaké předem vytyčené cíle, zároveň však reaguje na jedince ve svém okolí a na okamžitou situaci, ve které se zrovna nachází.

2.3.2 Mapy

Aby mohly být cesty plánovány, je třeba pamatovat si umístění jednotlivých překážek v regionu pomocí map. Pro potřeby našeho modelu virtuálního prostředí byly zvoleny dva typy těchto map, stejně jako v [23]. Základem je mřížková mapa, pokročilejší strukturou potom mapa ve formě kvadrantového stromu.

Mřížkovou mapu si lze představit tak, že při pohledu na virtuální svět shora na něj přiložíme čtverečkovanou síť. Buňky sítě, do kterých zasahuje některá překážka označíme jako obsazené. Ostatní buňky budeme nazývat volné. Pro potřeby plánování cest volné buňky dále dělíme na hraniční (to jsou ty v těsné blízkosti obsazených) a schůdné (ty zbylé). Hraniční buňky nám zajišťují, že naplánované cesty nepovedou v těsné blízkosti překážek.

Důležité je správně zvolit délku hrany čtvercové buňky. Ta nesmí být velká, aby algoritmy využívající mřížkovou mapu byly dostatečně přesné. Na druhou stranu příliš malá velikost buněk implikuje jejich značné množství (pohybujeme se ve dvou dimenzích, takže počet buněk roste kvadraticky), což snižuje rychlost algoritmů. Zkoušením různých možností byla zvolena jako optimální velikost asi 25cm.



Obrázek 2.2: Vlevo: Mřížková mapa pro plánování cest s rozlišenými obsazenými, hraničními a schůdnými buňkami. Vpravo: Mapa ve formě kvadrantového stromu s naznačenými schůdnými a neschůdnými uzly. Pro uzel vpravo dole jsou navíc zakresleny ukazatele na jeho schůdné sousedy.

Mapa ve formě kvadrantového stromu obsahuje pouze dva druhy uzlů — schůdné a neschůdné (ty odpovídají sjednocení množiny obsazených a hraničních buněk mřížkové mapy). Nejspodnější patro kvadrantového stromu získáme přímo z mřížkové mapy, vyšší patra potom postupným přidáváním směrem zdola nahoru. Celý postup stavby stromu je podrobně popsán v dokumentaci.

Každý schůdný list výsledného kvadrantového stromu obsahuje seznam všech schůdných sousedních uzlů ze svého 4-okolí. Tím nám na schůdných listech vzniká graf, který je použit při plánování cest. Vrcholy tohoto grafu jsou tedy různě velké čtvercové oblasti regionu, hrany mezi nimi naznačují, že spolu čtverce sousedí některou svou stranou (viz obr. 2.2).

2.3.3 Plánování

Pro plánování cest v představených mapách byl zvolen často používaný algoritmus A* [7, 11], který slouží k prohledávání obecného stavového prostoru. Pro potřeby prohledávání si algoritmus definuje uzel představovaný trojicí *[s, r, f(s)]*, kde s je prohledávaný stav, r je rodič s (stav, ze kterého jsme se do s při prohledávání dostali) a *f(s)* ohodnocení stavu s pomocí heuristické funkce *f*. Ta je definována jako součet $f(s) = g(s)+h(s)$, kde

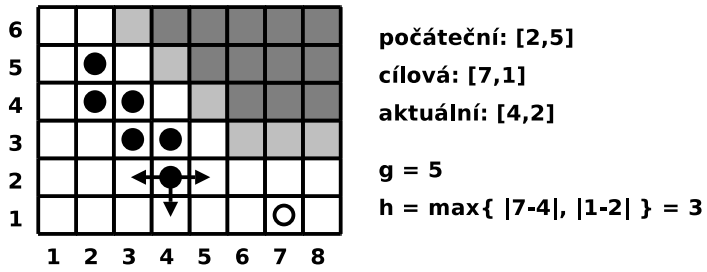
g(s) je délka nejkratší z dosud nalezených cest z počátečního stavu do s a *h(s)* odhad délky cesty z s do koncového stavu. Uzly si algoritmus A* drží ve dvou množinách. Ty uzly, jež představují již navštívené a expandované stavy, v množině *CLOSED*, uzly odpovídající zatím neexpandovaným stavům v *OPEN*.

Běh algoritmu lze symbolicky zapsat takto:

```
1. Inicializace pro počáteční stav p:
   OPEN = {[p,0,f(p)]}
   CLOSED = {}

2. Krok algoritmu:
   opakuj
      je-li OPEN == {}
         vrať neúspěch
      odstraň z OPEN uzel S = [s,x,f(s)] s minimálním f(s)
      přidej S do CLOSED
      je-li s cílový stav
         vrať úspěch
      expanduj uzel S a pro každý expandovaný uzel U = [u,s,f(u)]
         je-li [u,y,f'(u)] v OPEN pro nějaké y a f'(u) > f(u)
            odstraň [u,y,f'(u)] z OPEN
            vlož [u,s,f(u)] do OPEN
         je-li [u,y,f'(u)] v CLOSED pro nějaké y a f'(u) > f(u)
            odstraň [u,y,f'(u)] z CLOSED
            vlož [u,s,f(u)] do OPEN
         není-li [u,y,f'(u)] v OPEN ani CLOSED pro žádné y, f'(u)
            vlož [u,s,f(u)] do OPEN
```

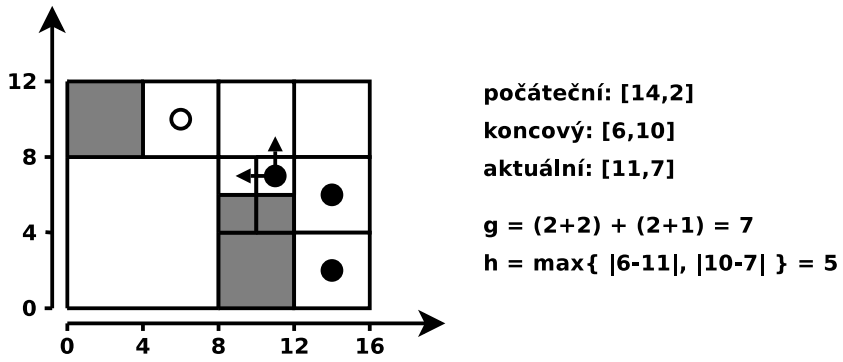
Lze dokázat [20], že aby algoritmus A* naleznul vždy optimální cestu, musí heuristická funkce h pro každý stav s dávat nezápornou hodnotu menší nebo rovnou délce nejkratší cesty z s do cílového stavu. Takovou heuristiku potom označujeme jako přípustnou. Z důvodu efektivnosti prohledávání je přitom žádoucí, aby funkce h délku této nejkratší cesty aproximovala co nejpřesněji.



Obrázek 2.3: Postup při vyhledávání cesty v mřížkové mapě se znázorněným ohodnocením aktuální buňky a s naznačením její expanze.

V mřížkové mapě jsou prohledávané stavy představovány jednotlivými schůdnými buňkami, počáteční a koncový stav odpovídají buňkám obsahujícím počáteční a cílovou pozici. Při expanzi buňky uvažujeme sousední schůdné buňky nacházející se v jejím 4-okolí. Tím máme také definovánu vzdálenost dvou buněk jako minimální počet posunů

mezi nimi ve vertikálním a horizontálním směru. Funkce g ohodnocující délku cesty z počáteční do současné buňky tak vyjadřuje počet navštívených buněk, pro vyhodnocení heuristické funkce h je použita maximová metrika (maximum z rozdílů indexů buněk v horizontálním a vertikálním směru). Je vidět, že tato heuristika je přípustná, protože nikdy nenadhodnocuje skutečnou vzdálenost a výsledná funkční hodnota je vždy kladná. Pro horizontální a vertikální směry ja potom zcela těsná. Příklad, jak hledání cesty v mřížkové mapě vypadá (i s naznačenou expanzí buňky), lze vidět na obr. 2.3.



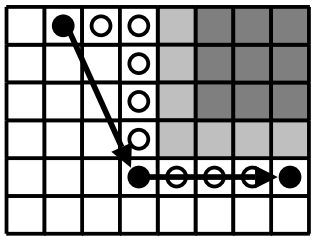
Obrázek 2.4: Postup při vyhledávání cesty v mapě v podobě kvadrantového stromu se znázorněným ohodnocením aktuálního listu a s naznačením jeho expanze.

Chceme-li najít cestu v mapě reprezentované kvadrantovým stromem, musíme nejdříve pro požadovanou počáteční a cílovou pozici rekurzivně najít odpovídající schůdné listy stromu. Protože listy se nacházejí na různých patrech stromu a představují tedy různě velké čtvercové oblasti, definujeme vzdálenost dvou sousedních listů jako součet polovin délek jejich hran (tedy vzdálenost středů ve vertikálním či horizontálním směru v závislosti na jejich vzájemné poloze). Funkce g tyto součty postupně akumuluje a ohodnocuje tak při prohledávání cestu z počátečního do současného listu. Jako heuristickou funkci h potom používáme maximovou metriku v kartézských souřadnicích mezi středy listů stromu. Příklad, jak hledání v kvadrantovém stromu vypadá (i s naznačenou expanzí listu), lze vidět na obr. 2.4.

Výhodou plánování cest v kvadrantovém stromu oproti plánování v mřížkové mapě je zpravidla daleko menší prohledávaný stavový prostor. Zejména v případě, kdy se v mapě nenachází příliš mnoho překážek, obsahuje strom schůdné listy představující velmi rozlehlou oblast, na jejíž reprezentaci mřížka potřebuje velký počet buněk.

2.3.4 Vyhlazování

Ať už použijeme pro plánování mřížkové mapy či mapy ve formě kvadrantového stromu, bývá získaná cesta často hodně zubatá. Abychom se zubů zbavili, vyhladíme cestu pomocí mřížkové mapy. Postupně zjišťujeme, která z buněk odpovídajících bodům cesty je ještě viditelná z počáteční buňky cesty. To děláme tak, že z počáteční buňky zkusíme rasterizovat (viz kapitola 2.4) do mřížky úsečku vedoucí do dané buňky. Jestliže rasterizovaná úsečka prochází pouze přes volné buňky, je daný bod viditelný. Všechny body

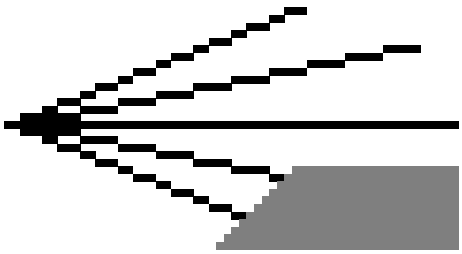


Obrázek 2.5: Při vyhlazování cesty vyhodíme všechny mezilehlé význačné body (zakresleny bílou barvou) až do posledního viditelného bodu, na který ukazuje šipka.

cesty mezi počáteční buňkou a poslední viditelnou buňkou odstraníme (tím se zbavíme v tomto úseku zubů) a pokračujeme obdobně dále, až do té doby, než dosáhneme cílové buňky. Příklad toho, jak vyhlazování cesty vypadá, lze vidět na obr. 2.5.

2.4 Vnímání okolních překážek

Dalším dotazem, jehož zodpovídání jsme po virtuálním světě v úvodu této kapitoly požadovali, byl dotaz na překážky v okolí agenta. Odpověď nalezneme pomocí dříve představené mřížkové mapy. Z buňky odpovídající pozici agenta vyšleme několik paprsků, které postupně rasterizujeme do volných buněk mapy. Ve chvíli, kdy narazíme na obsazenou buňku, víme, že daným směrem se překážka nachází.



Obrázek 2.6: Vnímání překážek pomocí rasterizace pohledových paprsků do mřížkové mapy.

Maximální délka, do které paprsek vysíláme, nám určuje maximální vzdálenost, na kterou agent překážky vidí a bere je v potaz. Úhel, který spolu svírají dva nejkrajnější vyslané paprsky, je zorným úhlem agenta a stejně jako u skutečného člověka bývá nastaven na 150° až 180°. Úhel mezi jednotlivými vysílanými paprsky potom určuje jakousi přesnost vnímání. Bylo vyzkoušeno, že nemá smysl volit tento úhel menší než cca 15°, taková přesnost je totiž více než dostačující.

Abychom nějak modelovali skutečnost, že člověk při svém vnímání věnuje více pozornosti objektům přímo před sebou a směrem do stran tato pozornost postupně klesá, má maximální délku pouze paprsek ve směru pohledu agenta a délka ostatních paprsků se postupně snižuje. Na základě [6] jsou délky paprsků ohraničeny parabolou $f(x) = -\frac{4}{d}x^2 + d$, kde d je délka paprsku ve směru pohledu.

Kapitola 3

Simulace

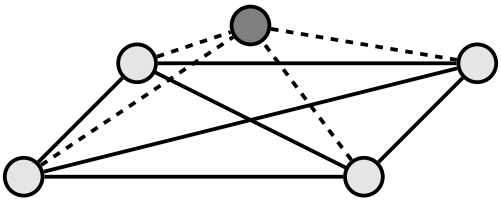
3.1 Paralelizace

Simulace agentů je velice náročná na strojový výkon, požadujeme-li, stejně jako v této práci, aby probíhala v reálném čase a aby jejím výsledkem byl nějaký gafický výstup. Sled po sobě jdoucích obrazů zachycujících pohyb se totiž jeví pro lidské oko jako plynulý, je-li rychlost promítání alespoň 20-30 obrazů za sekundu. Tolikrát bychom tedy měli také provést „posunutí“ agenta ve směru jeho pohybu. To samozřejmě neznamená, že při každém takovém posunutí musíme provádět kompletní simulaci rozhodování a chování agenta. Existují i modely, ve kterých se agent např. rozhodne udělat krok dopředu a několik dalších cyklů je věnováno pouze jeho provedení, není tedy třeba žádných složitých výpočtů. Obecně lze ale říct, že čím častěji probíhá celková simulace chování agenta, tím lépe, reálný člověk přece také myslí a koná spojitě.

Dav agentů, který bychom chtěli simulovat, může čítat až tisíce jedinců. To nám spolu se zmíněnými 20-30 aktualizacemi každého agenta za vteřinu a požadavkem modelovat kompletní rozhodování v každém kroku dává řádově desítky až stovky tisíc cyklů za jednu sekundu. Jeden cyklus přitom obsahuje spoustu poměrně náročných výpočtů.

Řešením je využít naplno potenciál dnešních vícejádrových procesorů či dokonce víceprocesorových systémů a provádět simulaci paralelně. Nejdříve je nutné se rozhodnout, na jaké úrovni budeme paralelizovat — zda pouze nějaké dílčí úkoly nebo celou simulaci. Zde byla zvolena druhá varianta. Také možností, jak paralelizovat na úrovni celé simulace, máme hned několik. Jednou z nich by bylo střídavě přidělovat nově příchozí agenty mezi různá vlákna; vlákno by se potom o agenta staralo po celou dobu jeho existence ve virtuálním prostředí. Tímto způsobem bychom sice dosáhli toho, že všechna vlákna by se staralo o téměř stejný počet agentů a zátěž by byla mezi ně rovnoměrně rozložena, na druhou stranu by bylo značně obtížné vlákna mezi sebou synchronizovat, neboť mezi agenty jednoho vlákna by nebyla žádná prostorová souvislost.

Naším cílem je přitom omezit komunikaci mezi jednotlivými simulačními vlákny na nejnutnější minimum. Oblast celého světa je proto podle [16, 19] mezi ně na začátku rovnoměrně rozdělena a každé vlákno je zodpovědné pouze za simulaci agentů nacházejích se v odélníkovém prostoru jemu přiděleném. Jestliže některý z agentů z této vymezené oblasti vyjde, je předán dalšímu vláknu. Všechny obdélníky jsou přibližně stejně velké a tvoří mřížkovou strukturu, každé vlákno tedy komunikuje pouze s vlákny ve svém



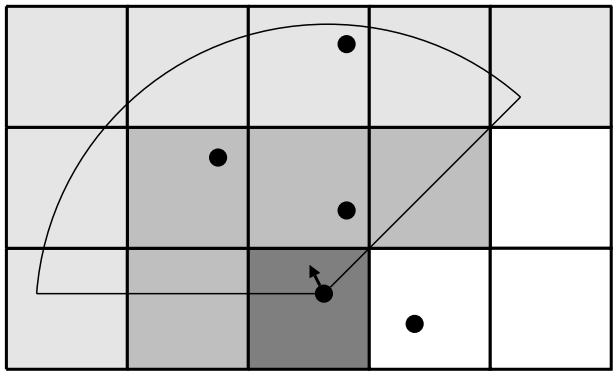
Obrázek 3.1: Komunikace mezi jedním hlavním a čtyřmi simulačními vlákny.

8-okolí (viz obr. 3.1). Kromě toho samozřejmě simulační vlákna musí komunikovat i s vláknem hlavním.

Také v tomto případě by bylo možné vyvažovat objem práce přidělené jednotlivým simulačním vláknům, stačilo by patřičným způsobem měnit velikosti jednotlivých obdélníkových oblastí v závislosti na hustotě zalidnění té které části světa. Tady se však tomuto tématu nevěnujeme.

3.2 Mapy agentů

Ať už použijeme pro simulaci chování a pohybu agentů jakýkoliv model, je výhodné držet si jejich pozice ve speciální mapě. Ta má, podobně jako jedna z map podporující plánování cest, podobu dvourozměrné mřížky. V mřížce však tentokrát nejsou uloženy žádné informace o překážkách. Namísto toho každá buňka mřížky obsahuje identifikátory těch agentů, kteří se právě nacházejí na jí odpovídajícím území. Velikost hrany buňky byla zvolena 5 metrů.

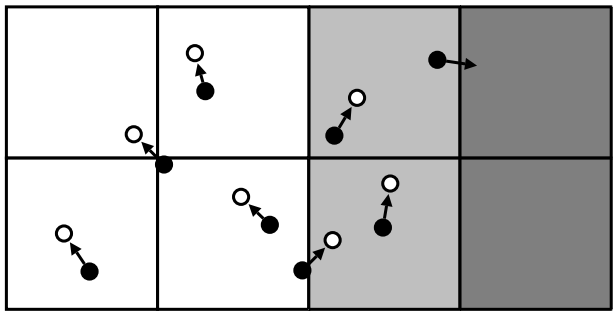


Obrázek 3.2: Vnímaní okolních agentů pomocí mřížkové mapy po jednotlivých úrovních na základě vzdálenosti.

Hlavním úkolem mapy je poskytovat agentům informace o okolních agentech nacházejících se v jejich zorném poli určeném zorným úhlem a maximální vzdáleností, na kterou agent svoje sousedy vnímá. To se děje tak, že jsou postupně prozkoumávány buňky mapy do tohoto zorného pole patřící. Agenti obsažení v těchto buňkách jsou načteni do paměti agenta, pro něhož je vnímání prováděno. Buňky blíže pozici agenta

jsou přitom prozkoumávány dříve, jak je vidět na obr. 3.2. To odpovídá tomu, že lidé věnují více pozornosti chodcům nacházejícím se v jejich nejbližším okolí, s nimi je totiž hrozba kolize největší.

Vnímání končí ve chvíli, jsou-li prozkoumány všechny buňky patřící do zorného pole, případně byl-li do paměti načten maximální možný počet okolních agentů. Ten je na základě psychologického experimentu omezen číslem sedm [13] a odpovídá maximálnímu počtu okolních chodců, jež je člověk při chůzi schopen brát v potaz.



Obrázek 3.3: Část mapy agentů pro jedno simulační vlákno s naznačeným pohybem agentů. Světle šedou barvou jsou vybarveny hraniční buňky, tmavě šedě klonované buňky.

Každé simulační vlákno obsahuje svou vlastní mapu agentů mu přidělených. Ta musí být aktualizována v každém kroku simulace, jak se agenti přesouvají mezi jednotlivými jejími buňkami. Ve chvíli, kdy se agent dostane mimo oblast daného vlákna (tedy i mimo oblast jeho mapy), je předán do patřičné buňky vlákna sousedního. Pohyb agentů v mapě lze vidět na obr. 3.3.

Jak již bylo řečeno, slouží mapa ke vnímání agentů v zorném poli daného agenta. Problém nastává pro agenty vyskytující se v blízkosti hranice se sousedním simulačním vláknem. Ti by totiž potřebovali vidět i agenty za touto hranicí, kteří se již v mapě našeho vlákna nevyskytují. Pro tento účel přidáme každé mapě na strany sousedící s jinými vlákny tzv. klonované buňky. Jejich obsahem jsou kopie agentů z hraničních buněk sousedního vlákna (viz obr. 3.4).

Na začátku každého cyklu provádějícího simulaci mu přidělených agentů odešle simulační vlákno kopie agentů z hraničních buněk do odchozích bufferů, z příchozích bufferů zase naopak přijme kopie agentů z hraničních buněk všech sousedních simulátorů a vloží je do svých klonovaných buněk. Jeden simulační cyklus potom vypadá takto [16]:

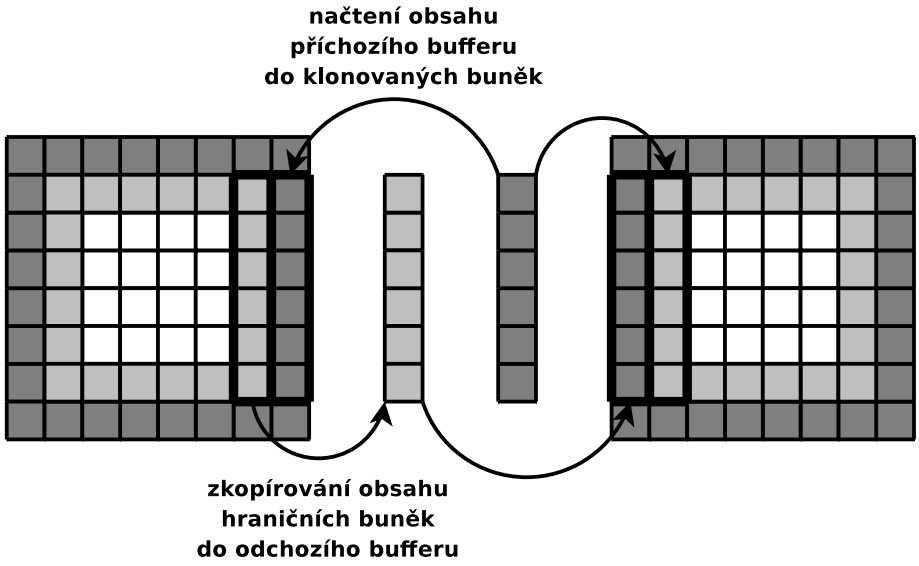
```
1. Výměna kopií agentů v oblasti hranic se sousedními vlákny:
   pro každý odchozí buffer
      zamkni buffer
      pro každou buňku odchozího bufferu
         vytvoř kopie agentů v odpovídající hraniční buňce mapy
         vlož kopie agentů
      odemkni buffer
   pro každý příchozí buffer
```

```
      zamkni buffer
      pro každou buňku bufferu
         přesuň kopie agentů do příslušné klonované buňky v mapě
      odemkni buffer

2. Simulační krok:
   pro každého agenta přiděleného simulátoru
      [i,j] = index buňky obsahující agenta
      simuluj pohyb agenta
      [k,l] = index buňky obsahující agenta
      je-li [i,j] != [k,l]
         odstraň agenta z buňky [i,j]
         je-li [k,l] platný index mapy
            vlož agenta do buňky [k,l]
         jinak
            vlož agenta do seznamu odchozích agentů

3. Výměna agentů se sousedními vlákny:
   zamkni seznam příchozích agentů
   vlož agenty do příslušných buněk mapy
   odemkni seznam příchozích agentů
   pro každé sousední vlákno
      zamkni jeho seznam příchozích agentů
      vlož do něj příslušné agenty ze seznamu odchozích agentů
      odemkni jeho seznam příchozích agentů
```

Přístup do sdílených bufferů a výměna agentů překročivších hranici jsou jediné operace, kdy spolu komunikují sousední vlákna a které je tedy třeba synchronizovat. Obě tyto operace jsou však velice rychlé, čímž máme zaručeno, že v kritických sekcích se sousední vlákna budou nacházet jenom po krátkou dobu. Podařilo se nám tedy splnit požadavek na omezení komunikace mezi sousedními simulačními vlákny na minimum.



Obrázek 3.4: Výměna agentů obsažených v buňkách mapy mezi dvěma sousedními simulačními vlákny. Bílou barvou jsou vybarveny vnitřní buňky mapy, světle šedou hraniční buňky, tmavě šedě klonované buňky.

Kapitola 4

Agenti

4.1 Reprezentace agentů

Hlavním požadavkem kladeným na agenty je, aby byli zcela autonomní. Mohou se sice ptát příslušných map na objekty ve svém okolí, jak bylo popsáno v předchozích kapitolách, avšak se získanými daty už si musí umět poradit sami. Existují sice i modely, kde je chování agentů řízeno nějak centrálně, náš autonomní přístup však lépe vystihuje realitu, skuteční lidé se přece také rozhodují a konají podle sebe.

Zde představený model se skládá ze dvou nad sebou ležících vrstev. Tou spodnější je vrsta reaktivního chování, která zajišťuje, aby se agent pohyboval po naplánovaných cestách, vyhýbal se okolním objektům apod. Lze zjednodušeně říci, že reaktivní vrstva je zodpovědná za základní pohyb agenta. Úkolem výše položené rozhodovací vrstvy je potom vymýšlet a spravovat cíle, jichž se agent snaží dosáhnout.

Agent je charakterizován několika základními parametry. Ty jsou každému nově příchozímu agentovi vygenerovány z normálního rozložení s určitou střední hodnotou a směrodatnou odchylkou. Normální rozložení bylo zvoleno z toho důvodu, že většina veličin měřitelných v populaci se řídí právě Gaussovou křivkou. Střední hodnotu generovaného parametru potom ještě většinou ovlivňuje pohlaví agenta (muži jsou totiž v průměru určitě vyšší a těžší než ženy).

K parametrům určujícím agenta patří kromě již zmíněného pohlaví, hmotnosti a výšky také maximální rychlost. Nejedná se přitom o maximální rychlost, kterou je schopen se pohybovat, ale o omezení rychlosti jeho chůze, jak bude vysvětleno později. Protože budeme těla agentů aproximovat pomocí kruhové hranice, je dalším parametrem poloměr tohoto kruhu. Ten vlastně určuje něco jako tloušťku agenta. Jestliže se hranice dvou agentů dotknou nebo dokonce protnou, dochází ke kolizi. Snahou našeho modelu je těmto kolizím co nejvíce předcházet, vyloučit je však úplně nelze. Poslední parametry souvisí s vnímáním agenta a byly uvedeny v kapitolách 2.4 a 3.2. Jde o zorný úhel, efektivní vzdálenost a parametr určující citlivost „zraku“.

4.2 Reaktivní vrstva

4.2.1 Princip řídících sil

Reaktivní vrstva agentů byla inspirována zejména prací [18] a v ní představeným modelem chování agentů založeném na řídících silách (steering forces model), který byl v některých ohledech rozšířen. Základní myšlenkou modelu řídících sil je, jak vyplývá i z názvu, působení sil na tělo agenta. Tyto síly ovlivňují pohyb agenta tím, že mění jeho vektor rychlosti, jako by se jednalo o skutečné fyzikální síly. Jejich původ však fyzikální není, jedná se spíše o jakési síly psychické vyjadřující vůli agenta něco udělat a síly sociální naznačující, jak na agenta působí okolní objekty.

K tomu, abychom mohli model psychologicko-sociálních řídících sil použít, musíme reprezentovat stav agenta pomocí několika fyzikálních veličin. Kromě již zmíněných neměnných parametrů hmotnost, maximální rychlost a poloměr kruhové aproximace těla jde o současnou pozici, aktuální rychlost, momentální orientaci a maximální možnou působící sílu. Zatímco veličiny hmotnost, maximální a aktuální rychlost, poloměr a maximální síla jsou skalární, pozici a orientaci agenta uchováváme ve formě dvourozměrných vektorů. Dva rozměry jsou postačující, neboť jiný pohyb agentů než v rovině není povolen. Vektor současné rychlosti potom snadno získáme vynásobením jednotkového vektoru určujícího orientaci a aktuální velikosti rychlosti.

Takto definovaná reprezentace agenta stojí někde na rozhraní mezi hmotným bodem a skutečným tělesem. Jeho tělo má sice podobu kruhu a je orientováno určitým směrem, na druhou stranu postrádá třeba moment setrvačnosti. Toto zobecnění nám však umožňuje síly snadno vyhodnocovat a aplikovat, což je prováděno v každém simulačním kroku agenta. Jeden krok simulace pro řídící sílu $\vec{F}$ tedy vypadá přibližně takto:

$$
\begin{aligned}
\vec{F} &= cut(\vec{F}, F_{max}) \\
\vec{a} &= \frac{\vec{F}}{m_a} \\
\vec{v}_a &= \vec{v}_a + t \cdot \vec{a} \\
\vec{v}_a &= cut(\vec{v}_a, v_{max}) \\
\vec{p}_a &= \vec{p}_a + t \cdot \vec{v}_a \\
\vec{o}_a &= \frac{\vec{v}_a}{\|\vec{v}_a\|}
\end{aligned}
$$

Tento výpočet vychází z [18], je však upraven o použití proměnné t. Ta určuje časovou periodu od provedení posledního simulačního cyklu a umožňuje tak modelovat pohyb agentů v reálném čase bez ohledu na počet cyklů prováděných za jednu sekundu.

Při podrobnějším pohledu na výpočet zjistíme, že se opravdu nejedná o nic jiného než o uplatnění fyzikálních vztahů známých z kinematiky a dynamiky. Proměnná $\vec{a}$ vyjadřuje zrychlení, m_a hmotnost agenta. Vektory $\vec{v}_a$, $\vec{p}_a$ a $\vec{o}_a$ potom popořadě jeho rychlost, pozici v kartézských souřadnicích a orientaci. Parametry F_{max} a v_{max} určují maximální velikost řídící síly a maximální velikost rychlosti pro daného agenta. Konečně funkce *cut* zajišťuje ořezání velikosti vektoru danou hodnotou.

4.2.2 Druhy řídících sil

Nyní se budeme zabývat prvním řádkem předchozího algoritmu, tedy vyhodnocením psychologicko-sociální řídící síly (steering force). Je jich několik různých druhů a my ke každé z nich uvedeme její účel a způsob výpočtu. Ve všech vzorcích přitom $\vec{p}_c$ značí pozici cíle v kartézských souřadnicích a $\vec{F}$ výslednou řídící sílu. Jinak je význam označení proměnných stejný, jako v předešlém vztahu.

První síla je motivována hledáním učité pozice. Její úlohou je natočit agentův vektor rychlosti tak, aby směřoval k této pozici. Výsledná síla je proto určena rozdílem mezi vektorem rychlosti mířící k cíli a vektorem současné rychlosti:

$$\begin{aligned} \vec{d} &= \frac{\vec{p}_c - \vec{p}_a}{\|\vec{p}_c - \vec{p}_a\|} \\ \vec{v} &= v_{max} \cdot \vec{d} \\ \vec{F} &= \vec{v} - \vec{v}_a \end{aligned}$$

Podobný účel má síla nutící agenta jít přímo k nějakému cíli. Tentokrát je ale určena přímo vektorem mířícím k cílové pozici. Jde vlastně o „agresivnější“ variantu předchozí metody. Místo abychom se snažili agenta natočit tak, aby k cíli směřoval, přímo jej k němu přitahujeme:

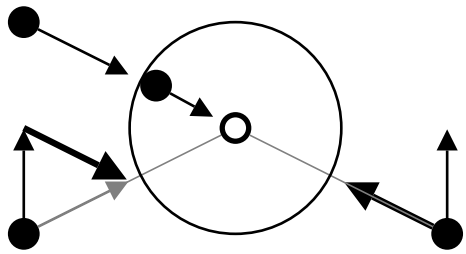
$$\vec{F} = \frac{\vec{p}_c - \vec{p}_a}{\|\vec{p}_c - \vec{p}_a\|}$$

Ve chvíli, kdy agent k cíli přichází, je často nutné snižovat postupně jeho rychlost. Parametrem výpočtu této síly je vzdálenost od cíle d_{max}, ve které má zpomalování začít. Mimo oblast omezenou touto vzdáleností je chování shodné s hledáním. Výpočet zapíšeme takto:

$$\begin{aligned} \vec{d} &= \vec{p}_c - \vec{p}_a \\ d &= \left\|\vec{d}\right\| \\ v_z &= v_{max} \cdot \frac{d}{d_{max}} \\ v &= min(v_z, v_{max}) \\ \vec{v} &= v \cdot \frac{\vec{d}}{d} \\ \vec{F} &= \vec{v} - \vec{v}_a \end{aligned}$$

V některých případech je nutné, aby se agent snažil dosáhnout nějaké dané velikosti rychlosti. To je např. tehdy, když chce zpomalit, aby se vyhnul hrozící kolizi, či naopak zrychlit, aby odněkud utekl. Síla, která nám toto zajistí, má směr vektoru určujícího orientaci agenta. Nechceme totiž, aby agent někam zatáčel. Sílu lze získat jako:

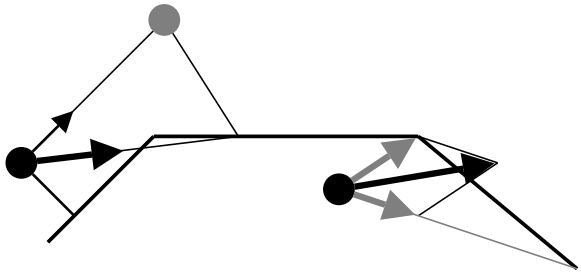
$$\begin{aligned} \Delta v &= \|\vec{v} - \vec{v}_a\| \\ \vec{F} &= \Delta v \cdot \vec{o}_a \end{aligned}$$



Obrázek 4.1: Vlevo nahoře: Příchod agenta k cíli s postupným snižováním rychosti po dosažení vzdálenosti znázorněné kruhem. Vlevo dole: Vyhledávání cíle, kdy agent jde nahoru, ale požadovaná rychlost je směrem k cíli. Vpravo dole: Tah přímo k cíli pro agenta mířícího nahoru.

Nyní bude následovat popis sil, jejichž výpočet je již o dost složitější, proto jeho podrobný výpis nebudeme dále uvádět, pouze vždy vysvětlíme princip. Složitost výpočtu dalších řídících sil souvisí s tím, že ty většinou realizují nějakou složitější reakci agenta.

Jestli-že má agent sledovat naplánovanou cestu reprezentovanou jako setříděný seznam význačných bodů (viz kapitola 2.3.1), je potřeba, aby byl k těmto bodům nějak přitahován. Ve chvíli, kdy se agent k význačnému bodu cesty přiblíží na určitou vzdálenost, začne jej pomalu přitahovat i další bod a dochází k interpolaci obou sil. Vliv první síly přitom slábne, vliv druhé se zvětšuje. Nakonec agent kolem bodu cesty projde a je již přitahován pouze k dalšímu bodu (viz obr. 4.2 vpravo).

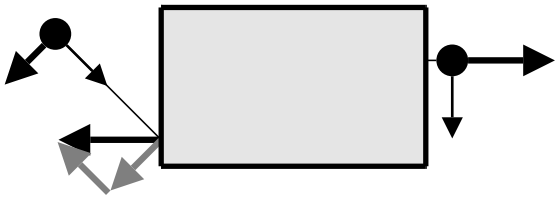


Obrázek 4.2: Vlevo: Predikovaná pozice agenta je příliš daleko od cesty, proto jej přitahujeme zpět. Vpravo: Interpolace sil při průchodu agenta kolem zlomu cesty.

Druhou možností je považovat cestu za lomenou čáru s určitým okolím k ní patřícím. Nejprve provedeme predikci pozice agenta za nějaký časový interval. Nachází-li se predikovaná pozice v okolí cesty, je všechno v pořádku, agent se totiž očividně po cestě pohybuje. V opačném případě nalezneme takový bod na cestě, ve kterém by se agent v budoucnosti nacházel, kdyby se po ní pohyboval, a určíme sílu směrem k tomuto bodu (follow path force). Tím zajistíme, že vždy když agent bude mít tendeci odcházet z cesty, bude k ní opět přitažen (viz obr. 4.2 vlevo).

Dalším problémem, se kterým se pomocí sil musíme vypořádat, je vyhýbání se překážkám. Jestliže agentovi hrozí kolize s některými z nich, jsou nejdříve tyto možné kolize uspořádány vzestupně dle vzdálenosti od agenta a dále je brána v úvahu pouze

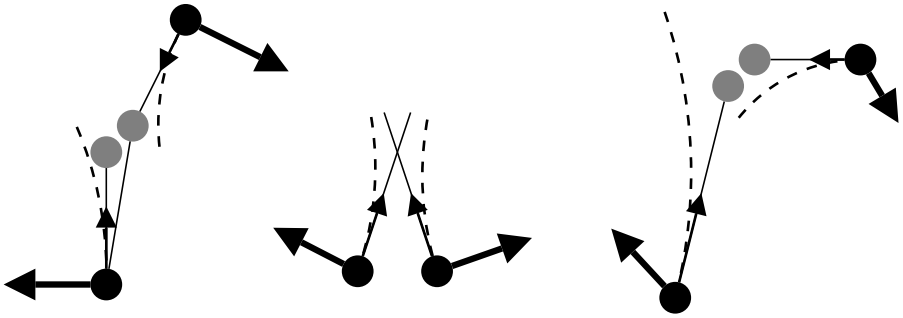
ta nejbližší. V místě na hranici překážky, ve kterém by k nebližší kolizi došlo, je poté vygenerována normála na tuto hranici. Ta určuje odpudivou sílu směrem od překážky. Její složka kolmá na orientaci agenta je potom výsledná řídící síla, která vlastně nutí agenta otáčet se směrem od místa kolize (viz obr. 4.3 vlevo).



Obrázek 4.3: Vlevo: Síla zamezující kolizi s překážkami. Vpravo: Síla udržující odstup od překážek.

Nevýhodu předchozího postupu je, že nefunguje příliš dobře pro boční kolize s překážkami, kdy o ně agent jakoby drhne svým ramenem. Proto byla vymyšlena další síla, udržující boční odstup agentů od překážek. Nachází-li se napravo či nalevo od agenta překážka, je nejdříve nalezen bod její hranice nejbližší pozici agenta. V tomto bodě je opět vygenerována normála na hranici, jejíž složka kolmá na orientaci agenta jej od překážky odpuzuje (viz obr. 4.3 vpravo). Důležité je, že tato boční síla roste s tím, jak se agent překážce přibližuje.

Poslední a na vyhodnocení zároveň i nejsložitější řídící silou je síla zabraňující kolizi se sousedními agenty. Stejně jako u překážek je z potenciálních kolizí zvolena ta, která by nastala za nejkratší dobu. Ostatní hrozby jsou zanedbány. Výpočet výsledné řídící síly závisí na vzájemné poloze obou ohrožených agentů a jejich orientaci.



Obrázek 4.4: Vlevo: Vyhýbání agentů při hrozbě čelní kolize. Uprostřed: Vyhýbání agentů orientovaných stejným směrem. Vpravo: Vyhýbání agentů při hrozbě boční kolize.

Jestliže agentům hrozí čelní kolize, směřuje výsledná síla do boku od pozice, na které by se v čase kolize nacházel sousední agent (viz obr. 4.4 vlevo — pozice horního agenta v době kolize je vpravo od dolního, proto dolní agent zatáčí postupně doleva). Tím pádem agent vždycky uhýbá do volnějšího prostoru.

Jestliže je orientace agentů takřka stejná, snaží se v případě hrozby kolize oba ustoupit na druhou stranu, než se nachází jejich soused. Tedy např. na obr. 4.4 uprostřed se levý agent snaží uhnout doleva, neboť druhý agent se v dané chvíli nachází po jeho pravici.

Třetím typem kolize je boční kolize (viz obr. 4.4 vpravo). Té se agenti snaží předejít tak, že rychlejší z agentů (tj. levý dolní na obr. 4.4 vpravo) zrychlí svůj krok a zatáčí jakoby před místo kolize. Pomalejší naopak ještě více zpomalí a vydá se směrem ke kolizi. Tím pádem je zaručeno, že rychlejší agent má čas pomalejšího obejít.

4.2.3 Použití řídících sil

Dále je nutné navrhnout správný způsob, jakým budou různé druhy sil vyhodnocovány a aplikovány. Existuje přitom hned několik možností [18].

První z nich je vyhodnotit v každém kroku všechny uvedené síly a jako výslednou řídící sílu uvažovat jejich součet. Nevýhodou tohoto přístupu je však to, že síly působí různě proti sobě. Výslednice je tak často náchylná na časté změny, proto pohyb může působit chaoticky. Navíc dochází velmi často ke kolizím, neboť nad silami jí zabraňujícím převládne např. síla přitahující agenta k cílovému bodu.

Částečným řešením tohoto problému je přiřadit jednotlivým řídícím silám během sčítání rozdílné váhy a definovat tak jejich důležitost. Nyní už dochází ke kolizím méně často. Stále však zůstává další problém, a to že jsou zbytečně vyhodnocovány i síly, kterých není v danou chvíli vůbec potřeba.

Proto byl po mnohém testování zvolen postup ještě zcela odlišný. Jeho hlavní myšlenkou je nejdříve rozhodnout, které ze sil bude v dané chvíli potřeba. Nenachází-li se v blízkém okolí agenta žádná překážka, není třeba počítat sílu udržující od překážek odstup. Nehrozí-li kolize s žádným agentem, nemusíme se jí snažit zabraňovat. Ve fázi rozhodování přitom agent využívá svých znalostí o okolním světě uložených v jeho paměti překážek a agentů.

Při rozhodování postupujeme od nejdůležitějších sil k těm méně důležitým. Zjednodušeně lze říct, že nejdříve se snažíme vyhýbat překážkám, potom okolním agentům a teprve nakonec sledovat cesty, zpomalovat při příchodu k cíli, dosahovat určité rychlosti apod. Zamezení kolizi je tedy priorotou i za cenu dočasného přerušení sledování určitého cíle.

V momentě, kdy je nalezena síla, kterou by bylo v dané situaci třeba aplikovat, ke zkoumání ostatních sil už vůbec nepřistupujeme. Tím zabráníme tomu, aby se síly mezi sebou různě přetahovaly. Aby nebyla použita vždy pouze nejdůležitější síla, je dále možno přiřadit ke každé určitou pravděpodobnost, že bude přeskočena, přestože by měla být správně v dané situaci aplikována.

4.3 Rozhodovací vrstva

Nad reaktivním chováním založeným na principu řídících sil stojí rozhodovací vrstva agenta. Jejím úkolem je zejména spravovat a určovat cíle jichž se agent snaží dosáhnout.

Dále musí poskytovat spodní vrstvě cesty, po kterých se má agent pohybovat, zajišťovat korektní přecházení mezi jednotlivými regiony apod.

Základem rozhodovací vrstvy je tzv. zásobník cílů. Ten obsahuje všechny cíle agenta, na vrcholu se přitom nachází cíl aktuálně plněný. Ve chvíli, kdy je tohoto cíle dosaženo, je z vrcholu zásobníku odstraněn a aktuálním se stává cíl nacházející se pod ním. Použití zásobníku byl inspirován systémem STRIPS [3], jenž umí řešit obecnou úlohu plánování pro potřeby umělé inteligence.

Některé z cílů jsou přitom jednoduché, jiné vyžadují pro své splnění nejdříve dosáhnout několika podcílů. Příkladem jednoduchého cíle je situace, kdy má agent dosáhnout pozice nacházející se ve stejném regionu, jako je on sám. V tom případě stačí pomocí algoritmů uvedených v kapitole 2.3.1 naplánovat k dané pozici cestu, po které se agent vydá. Cíle je potom dosaženo, jestli-že se agent k této pozici dostatečně přiblíží.

Naopak dosažení pozice nacházející se v jiném regionu (viz kapitola 2.2) je příkladem cíle složeného. Ten je proto rozdělen na jednotlivé podcíle, které jsou ve správném pořadí přidány do zásobníku nad náš cíl. Splněním těchto dílčích podcílů potom umožníme splnit i cíl hlavní. V tomto konkrétním případě je třeba nejdříve dosáhnout portálu do prvního regionu na cestě k regionu cílovému, projít všemi mezilehlými regiony a nakonec v regionu cílovém dosáhnout požadované pozice.

Dalším příkladem složeného cíle jsou konjunkce a disjunkce podcílů. V případě disjunkce je náhodně zvolen jeden z podcílů, který přidáme na vrchol zásobníku a snažíme se jej splnit. Ve chvíli, kdy se nám to podaří, splnili jsme i disjunkci cílů a můžeme ji taktéž odstranit ze zásobníku. S konjunkcí je postup obdobný s tím rozdílem, že musíme splnit postupně všechny podcíle.

Pomocí cílů lze jednoduše simulovat i takové chování, jako je náhodné procházení se agenta. Pouze vytvoříme speciální cíl, který nemůže být nikdy splněn a jež jako svůj podcíl vždy vygeneruje dosažení nějaké náhodné pozice, a to ať už v současném regionu agenta, nebo v jakémkoliv jiném.

Kapitola 5

Implementace

5.1 Základy

5.1.1 Požadovaná funkčnost

Cílem projektu MASS (Moving Agents Simulation System) bylo vytvořit aplikaci simulující v reálném čase početný dav chodců, čítající řádově stovky až tisíce jedinců, na běžném osobním počítači. Protože je taková simulace velice náročná na výpočetní výkon, jedním z hlavních požadavků bylo, aby program běžel v několika paralelně prováděných vláknech a využíval tak plně možností moderních vícejádrových procesorů, které se v dnešní době staly už de facto standardem. Mezi další požadavky patřilo jednoduché grafické zobrazení průběhu simulace a možnost toto zobrazení interaktivně ovlivňovat pomocí základního uživatelského rozhraní. Aby byl program reálně využitelný, bylo dále třeba definovat formát map prostředí, které se při inicializaci aplikace načítají ze vstupního souboru a ve kterých poté simulace probíhá. Původním záměrem projektu bylo i vytvoření nějakého editoru těchto map, z toho však nakonec sešlo z důvodu přílišné rozsáhlosti projektu. V neposlední řadě měla existovat možnost ovlivňovat i parametry simulace samotné, jako je například počet vláken, pomocí konfiguračního souboru.

Z hlediska zdrojového kódu byla cílem hlavně přehlednost a snadná rozšiřitelnost, čehož bylo dosaženo rozdělením programu do několika víceméně nezávislých modulů a využitím všech výhod objektově-orientovaného návrhu, mezi které patří zapouzdření, dědičnost a polymorfismus. Dalším z požadavků byla optimalizace použitých algoritmů a datových struktur, která je vzhledem k dříve zmiňované náročnosti simulace na strojový výkon alespoň v základní míře nezbytná. Ke snadné rozšiřitelnosti funkčnosti aplikace přispívá také velké množství podrobných komentářů. A to jak komentářů popisujících rozhraní tříd, které lze automaticky exportovat pomocí nástroje Doxygen do různých formátů (PDF, HTML apod.), tak i komentářů uvnitř definic funkcí vysvětlujících použité algoritmy. Snahou bylo psát kód nezávislý na operačním systému, čemuž byl uzpůsoben i dále popsaný výběr použitých technologií.

Lze říci, že všechny výše uvedené cíle byly splněny, pomineme-li to, že se nepodařilo vytvořit editor map. Přesto v projektu zbývá spousta prostoru na vylepšení a rozšíření funkčnosti některých modulů a nebyl by problém na vývoji ještě dlouho pokračovat.

5.1.2 Použité technologie

Jako programovací jazyk bylo především z důvodu rychlosti zkompilovaných programů vybráno C++. Dalším faktorem bylo to, že C++ snad jako jediný jazyk umožňuje psát jak nízkoúrovňový kód, tak zároveň vytvářet obecné a znovupoužitelné vysokoúrovňové návrhy. Protože je C++ mezi vývojáři hodně rozšířené a jeho výuka patří k základním přednáškám na MFF UK, existuje velká pravděpodobnost, že kdyby v projektu pokračoval někdo jiný, odpadnul by problém s učením se nového jazyka. S rozšířeností C++ souvisí také existence celé řady knihoven, které byly při programování ve velké míře využívány. Jedná se zejména o STL (Standard Template Library) obsahující implementaci nejzákladnějších algoritmů a datových struktur. V neposlední řadě svou roli sehrála i velká míra zkušeností s C++ v porovnání s ostatními programovacími jazyky.

Aby byl výsledný projekt přenositelný mezi platformami, bylo nutné pro přístup k systémovým zdrojům použít nějakou knihovnu. Volba padla na SDL (Simple Direct-Media Layer) ve verzi 1.2.13 šířenou pod licencí GNU LGPL. Této knihovny využívají nejčastěji hry, emulátory a software pro přehrávání multimedií. My jsem si ji vybrali zejména pro její snadnou použitelnost, pro dobrou komunikaci s OpenGL a také proto, že poskytuje jednoduché rozhraní pro práci s vlákny a pro jejich synchronizaci. SDL sice neumožňuje snadno vytvářet složité uživatelské rozhraní, v našem případě to ale vůbec nevadilo, neboť se v programu žádná menu a ovládací prvky nevyskytují. Jedinou naší výhradou vůči SDL je to, že pomocí ní nelze vytvářet fonty, což například podobně zaměřená knihovna GLUT umožňuje. Nakonec ale byla tahle nepříjemnost vyřešena pomocí bitmapového fontu uloženého přímo v kódu programu.

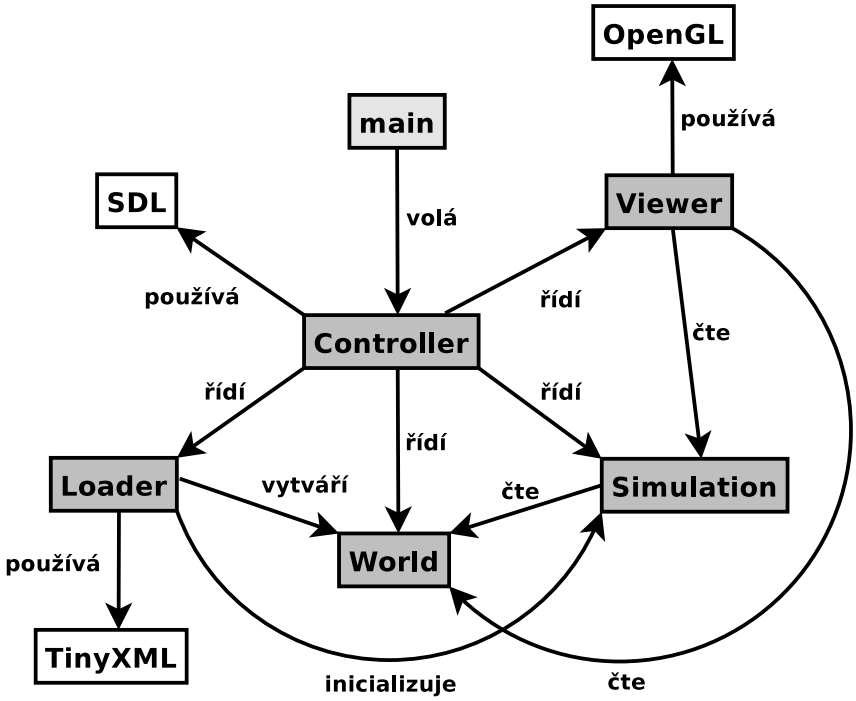
Pro grafický výstup na povrch okna poskytovaného SDL jsme použili průmyslového standardu OpenGL (Open Graphics Library), který zřejmě není třeba nijak blíže představovat. Volba na OpenGL proti konkurenčnímu rozhraní DirectX padla především kvůli platformové nezávislosti. Nevýhodou je naopak to, že v současné verzi OpenGL neposkytuje objektové rozhraní.

Protože program MASS ke svému běhu potřebuje virtuální svět definovaný pomocí map, jakožto i nastavení některých parametrů simulace, bylo třeba zvolit formát, ve kterém budou vstupní soubory zadány. K tomu byl využit obecný značkovací jazyk XML (eXtensible Markup Language). Abychom nemuseli psát vlastní syntaktický analyzátor, použili jsme hotovou knihovnu TinyXML poskytující obsah souboru v reprezentaci DOM (Document Object Model). Výhodou knihovny TinyXML je její jednoduchost a malá velikost, nevýhodou potom to, že neumí načítané soubory validovat. Proto součástí projektu není přesná definice formátu vstupních souborů pomocí DTD (Document Type Definition).

Projekt byl vyvíjen a testován pod operačním systémem MS Windows XP. Přestože byl kladen důraz na multiplatformnost, jak jsme již několikrát předeslali, pod žádným jiným operačním systémem nebyl nikdy testován a nelze proto zaručit jeho správnou funkčnost. Při vývoji bylo používáno integrované prostředí MS Visual Studio Professional ve verzích 2003 a 2005, které je studentům MFF UK poskytováno bezplatně prostřednictvím programu MSDN Academic Alliance. O správu jednotlivých verzí se potom staral systém SVN (Subversion).

5.1.3 Modulární architektura

Celý program lze rozdělit do pěti základních modulů (viz obr. 5.1) a několika dalších modulů pomocných. Některé z modulů odpovídají pouze jedné třídě, jiné jsou zase tvořeny celým balíkem tříd. Tento modulární systém byl zvolen po pečlivém rozvážení a jeho hlavní předností je přehlednost zdrojových kódů a snadná rozšiřitelnost či změna implementace (např. by nebyl problém vyměnit současný `Viewer` používají standard OpenGL za jiný, který by vykresloval pomocí DirectX).



Obrázek 5.1: Pět hlavních modulů programu, komunikace mezi nimi a způsob využití externích knihoven.

Hlavním modulem aplikace je `Controller` představovaný stejnojmennou třídou a obsahující pouze statické proměnné a statické metody (takovou třídu budeme dále v textu označovat jako statickou). `Controller` je volán přímo z funkce `main()` a zodpovídá za řízení všech ostatních modulů, inicializaci SDL a zpracování zpráv SDL.

Funkce modulu `Loader` je zřejmá, jeho úkolem je zpracovávat vstupní soubory a na jejich základě inicializovat jak datové struktury reprezentující virtuální svět, tak i nastavovat parametry simulace. Z programátorského hlediska je tento modul nezajímavý, jedná se pouze o zpracování XML souborů v DOM reprezentaci za použití externí knihovny TinyXML, proto mu nebude dále věnována pozornost. Zmínku si snad zaslouží jedině to, že zdrojové kódy knihovny TinyXML byly přidány přímo do repository SVN. Rozhodli jsem se tak především kvůli její malé velikosti a také proto, že tato knihovna není tak známá a rozšířená jako například SDL, která součástí repository není.

Modul `World` reprezentuje virtuální svět, ve kterém simulace probíhá. Mluvíme-li o objektech obsažených ve virtuálním světě, máme však na mysli pouze objekty statické

(překážky), nikoliv pohybující se agenty, které zahrnujeme spíše do části `Simulation`. V modulu `World` se také nachází informace o rozdělení světa do menších logických jedotek (regionů), o propojení těchto regionů pomocí portálů, o bránách umožňujících vstup do světa a výstup z něj. Dále je tento modul zodpovědný za uchovávání struktur umožňujících plánování cest.

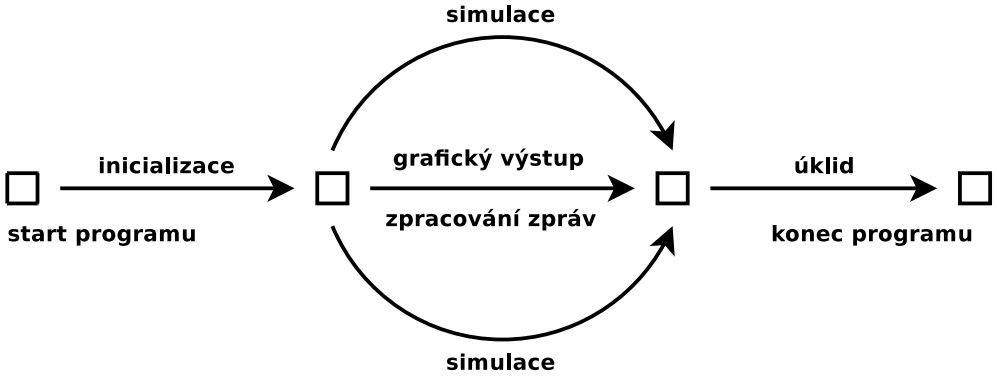
Modul `Simulation` je potom tím nejdůležitějším v celém programu. Jeho hlavním úkolem je simulovat činnost všech agentů pohybujících se ve virtuálním světě, udržovat datové struktury umožňující efektivní lokalizaci agentů, zodpovídat dotazy agentů na dění v jejich okolí, plánovat cesty apod. Ke zjišťování některých informací o světě `Simulation` pokládá dotazy modulu `World`. Dalším důležitým posláním části `Simulation` je správa vláken, ve kterých simulace probíhá, a synchronizace mizi nimi.

Nakonec jsmi si nechali popis části `Viewer`. Jejím úkolem je zobrazovat průběh simulace, přičemž využívá informací z modulů `World` a `Simulation`. Kromě toho se ještě stará o výpis obsahu virtuální konzole na obrazovku.

5.1.4 Vlákna

Jak již bylo zmíněno výše, jednou z hlavních předností MASS oproti konkurečním systémům je jeho schopnost efektivně využívat vícejádrové procesory, a to použitím vícevláknového přístupu. Všechny inicializace a tvorba pomocných struktur po spuštění programu běží v hlavním vlákně. Zde by bylo zřejmě zbytečné cokoliv paralelizovat a urychlovat, neboť se nejedná o časově kritickou část. Po spuštění samotné simulace je úkolem hlavního vlákna reagovat na uživatelské vstupy ve formě zpráv SDL, vykreslovat grafický výstup pomocí modulu `Viewer` a starat se o vedlejší vlákna. Tyto činnosti již v podstatě není možné rozdělovat mezi více vláken, neboť smyčku SDL i vykreslování pomocí OpenGL se obecně nedoporučuje provádět v jiném než hlavním vlákně.

Všechna ostatní vlákna, jejichž počet je jedním z konfiguračních parametrů aplikace, provádí simulaci agentů. O jejich běh a vzájemnou synchronizaci se stará modul `Simulation`, jak bude podrobněji popsáno dále. Jelikož se hlavní vlákno simulací vůbec nezabývá, musí vždy existovat alespoň jedno vedlejší, simulační vlákno. Program tak pokaždé běží alespoň ve dvou vláknech.



Obrázek 5.2: Znázornění běhu programu v jednom hlavním a dvou simulačních vláknech.

Pro správu vláken používáme na platformě nezávislých funkcí knihovny SDL. Ta zároveň poskytuje i základní synchronizační primitiva, z nichž byl zvolen pouze mutex, který je nejjednodušší a do kritické sekce umožňuje vstoupit pouze jednomu vláknu. To bylo pro naše potřeby zcela dostačující, složitější struktury semafor a condition variable proto v programu nevyužíváme. Nad rozhraní poskytované SDL byla přidána ještě jednu vrstva definovaná v souboru `threads.h`. Ta umožňuje vytvořit a spustit nové vlákno, počkat na skončení vlákna a přerušit jej. Pro potřeby synchronizace potom lze vytvořit mutex, uzamknout jej a znovu uvolnit. K přidání mezivrstvy vedlo především to, že vlákna jsou používána na mnoha různých místech programu a nechtěli jsme se připravit o možnost snadno nahradit SDL za jinou knihovnu. Jelikož všechny funkce mezivrstvy pouze převolávají funkce SDL a jsou deklarovány jako `inline`, nemělo by docházet k žádnému zpomalení programu.

5.2 Modul `Controller`

5.2.1 Architektura

Obsahem této kapitoly je podrobnější popis implementace kontrolního modulu zodpovědného za řízení všech ostatních modulů, jenž je reprezentován statickou třídou `Controller`. Ta funkci `main()` zpřístupňuje 4 veřejné statické funkce; jsou to `parseCommandLine()`, `init()`, `run()` a `finish()` a volají se v uvedeném pořadí.

Funkce `parseCommandLine()` dostává jako parametr obsah příkazové řádky, který by měl být tvořen cestou ke konfiguračnímu souboru a souboru s mapou. Pokud tomu tak není, pokusí se program použít implicitní soubory. Jesti-že ani ty nejsou k dispozici, program v tuto chvíli končí, neboť není co simulovat.

Dále se volá funkce `init()`, která inicializuje všechny ostatní moduly. Jako první vytvoří pomocný modul `Logger`, aby bylo možno vypisovat informace o průběhu programu. Poté je volán modul `Loader`, který načte mapy a nastavení simulace. V tomto kroku je tedy mimo jiné vytvářen virtuální svět. Následuje inicializace subsystémů SDL, kontextu OpenGL a vytvoření okna aplikace. Nyní je už možno inicializaci dokončit vytvořením modulu `Viewer`.

Funkce `run()` požádá simulační modul, aby rozběhnul samotnou simulaci v nových vláknech. Sama potom v nekonečné smyčce zpracovává zprávy SDL a umožňuje tak reagovat na podměty uživatele, překreslovat obsah okna a řídit shora celou simulaci. Provádění nekonečné smyčky končí ve chvíli, kdy k tomu dá uživatel příkaz.

Konečně funkce `finish()` zodpovídá za zrušení všech ostatních modulů a za korektní ukončení programu a jako taková není příliš zajímavá.

5.2.2 Smyčka zpráv

Náš návrh na zpracování událostí SDL původně vycházel z tradiční smyčky zpráv, která je v takřka nezměněné podobě obsažena snad v každé aplikaci tuto knihovnu používající:

```
run = true;
while (run) {
```

```
   while ( SDL_PollEvent(&event) ) {
      switch (event.type) {
         ... reakce na zprávy ...
         case SDL_QUIT:
            run = false;
            break;
      }
   }
   draw();   // grafický výstup
}
```

Její nevýhoda je však zřejmá — neustálé provádění dvou smyček zbytečně zatěžuje procesor. Navíc může snadno dojít k tomu, že funkce `draw()` je volána příliš často, což vede k další nadbytečné zátěži stroje.

Proto jsme raději navrhli a implementovali architekturu jinou, která místo neustálých pokusů o vyzvednutí zprávy z fronty na zprávy raději čeká:

```
while ( SDL_WaitEvent(&event) ) {
   switch (event.type) {
      ... reakce na zprávy ...
      case SDL_USEREVENT:
         draw();
         break;
      case SDL_QUIT:
         return;
   }
}
```

Grafický výstup je potom zajištěn přidáním časovače, který vždy jednou za určitý interval přidá do fronty speciální zprávu obsahující požadavek na aktualizaci modulu `Viewer` a překreslení okna. Aby nedošlo k tomu, že fronta zpráv bude zahlcena příliš mnoha požadavky na vykreslení, přidá se nová zpráva pouze tehdy, byla-li předchozí již vyzvednuta. Jestliže nebyla, další pokus o přidání proběhne za poloviční časový interval, aby se zamezilo případnému trhání výstupu. V případě, že simulace neběží, časovač nedává požadavky na vykreslení vůbec a vykresluje se pouze v případě, kdy je to nezbytně nutné (např. při změně velikosti okna, zásahu do oblasti okna oknem jiným apod.). Výhodou našeho přístupu je také to, že při běžící simulaci se scéna překresluje pouze tak často, jak o to žádá časovač. Tím pádem je možno nastavit pomocí konfiguračního souboru maximální počet snímků, které se zobrazí za jednu sekundu (FPS).

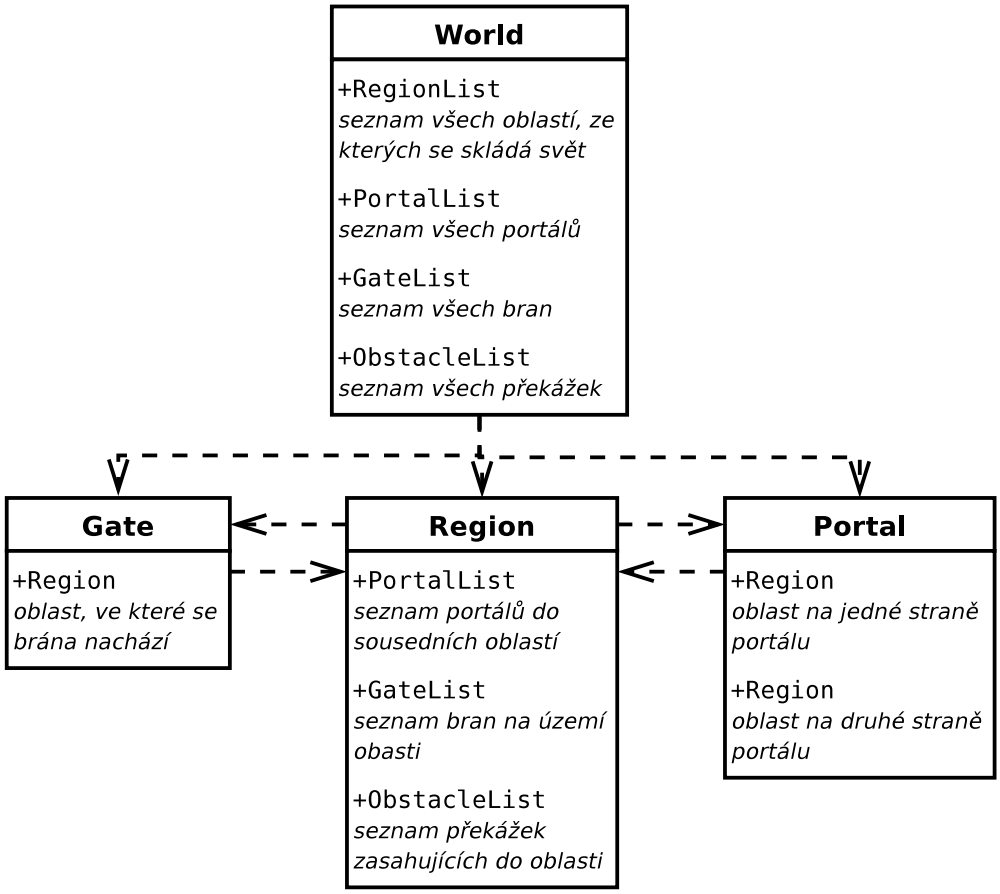
Další zprávou zasílanou pomocí časovače je požadavek na aktualizaci manažeru simulace, aby bylo možno simulačním vláknům přidat nové agenty, případně již neplatné agenty odebrat. Přitom je nutné, aby hlavní vlákno komunikovalo s vlákny vedlejšími; nachází se zde tedy jedna z kritických sekcí a je třeba zajistit exkluzivní přístup k seznamům agentů, což je podrobně popsáno v kapitole o modulu `Simulation`.

Dalšími zprávami, které smyčka zpracovává, jsou události vzniklé na popud uživatelského vstupu, a to vstupu z klávesnice a myši. Tyto zprávy jsou potom na základě tabulky definující ovládání programu přeloženy na zprávy ovlivňující pozici kamery a možnosti zobrazení a zaslány modulu `Viewer`. Ten na ně v závislosti na implementaci

může, ale také nemůsí reagovat (je zřejmé, že pro nějaké jednoduché 2D zobrazení by třeba rotace kamery okolo scény nebyla možná).

5.3 Modul `World`

5.3.1 Reprezentace světa



Obrázek 5.3: Třídy sloužící k reprezentaci světa. Šipka naznačuje, že první třída obsahuje druhou.

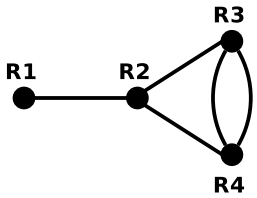
Virtuální svět jako celek je představován třídou `World`. Ta uvnitř obsahuje seznam ukazatelů na instance třídy `Region`, která reprezentuje jednu oblast světa, třídy `Portal`, jež představuje propojení dvou sousedních oblastí, a třídy `Gate` označující bránu, která slouží jako východ ze světa nebo vchod do něj. Seznamem je v tomto případě myšlena datová struktura `std::vector`. Důvodem, proč jsou v seznamech namísto samotných objektů uloženy pouze ukazatele na ně, je zamezení duplicitě dat. Ukazatel na každou z bran je totiž navíc uložen v instanci třídy `Region`, na jejímž území se brána nachází, kromě toho si `Region` drží i seznam všech portálů do všech dostupných sousedních

regionů. Konečně portál si uvnitř pamatuje oba dva regiony, které spojuje. Celé schéma je přehledně zobrazeno na obr. 5.3.

Posledním důležitým seznamem je potom seznam ukazatelů na překážky. Každý region má přitom svůj vlastní seznam těch překážek, které zasahují do plochy regionem vymezené. Kromě toho si ještě uvitř třídy `World` držíme seznam všech překážek obsažených ve světě. To je nutné z toho důvodu, že jedna překážka může zasahovat i do více regionů a nebylo by tedy jasné, který z nich je zodpovědný za její správu a korektní zrušení.

5.3.2 Třída `World`

Hlavním úkolem této třídy je poskytovat agentům informaci o tom, do kterého ze sousedních regionů regionu současného mají dále pokračovat, chtějí-li dosáhnout určitého regionu cílového. K tomu slouží pomocná mapa, jejíž tvorba byla popsána dříve. Základní položkou uloženou v mapě je struktura obsahující ukazatel na cílový region a celé číslo určující vzdálenost tohoto regionu. Tyto položky jsou potom ve formě seznamů uloženy ve dvourozměrné matici. Jak taková matice vypadá např. pro čtyři regiony *R1–R4* tvořící topologii danou grafem z obr. 5.4 lze vidět v tabulce 5.5.



Obrázek 5.4: Graf reprezentující možný příklad virtuálního světa. Vrcholy odpovídají regionům, hrana mezi regiony značí, že mezi nimi existuje portál.

	R1	R2	R3	R4
R1	{[R1,0]}	{[R2,1]}	{[R2,2]}	{[R2,2]}
R2	{[R1,1]}	{[R2,0]}	{[R3,1], [R4,2]}	{[R4,1], [R3,2]}
R3	{[R2,2], [R4,3]}	{[R2,1], [R4,2]}	{[R3,0]}	{[R4,1], [R2,2]}
R4	{[R2,2], [R3,3]}	{[R2,1], [R3,2]}	{[R3,1], [R2,2]}	{[R4,0]}

Tabulka 5.5: Výsledná mapa pro plánování cest mezi regiony uložená jako matice.

Jestliže se agent chce dostat např. z regionu *R3* do regionu *R2*, čemuž odpovídá seznam v řádku *R3* a sloupci *R2* tabulky 5.5, má dvě možnosti, do kterého sousedního regionu se vydat. Může jít buďto přímo do *R3*, přičemž cesta bude mít délku 1, nebo do *R4*, přičemž cesta bude alespoň délky 2. Kdyby se rozhodnul pro druhou z možností, mohlo by poté dojít k tomu, že v regionu *R4* si vybere pro pokračování do *R2* opět cestu délky 2 a dostane se tak zpátky do *R3* (viz řádek *R4* a sloupec R2 tabulky 5.5).

Takové opakované návraty by zřejmě nevypadaly příliš realisticky, proto se v současné implementaci volí pouze mezi sousedy ležícími na nejkratší cestě. Další možností, jak toto vyřešit, by bylo dát na počátku agentovi pouze několik málo jednotek, o které si cestu může prodloužit, poté už by mohl pokračovat pouze po té nejkratší. Uspokojivě by mohl fungovat také pravděpodobnostní přístup, kdy by se při vybírání souseda dávala větší váha těm regionům, jež leží na nejkratší cestě.

5.3.3 Třídy `Portal` a `Gate`

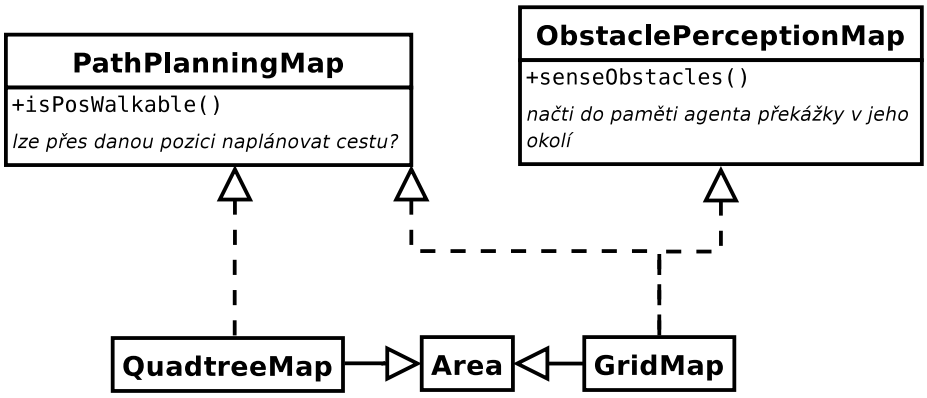
Implementace těchto dvou tříd je velice podobná, proto je popíšeme zároveň. Základní funkcí, kterou třídy poskytují, je vygenerování bodu, který patří do oblasti jimi určené. Tyto body se poté stávají cílem cest agentů. U brány se jedná o bod, kterým agent vstupuje do světa, nebo o bod, kterým z něj naopak vychází. U portálu je situace poněkud složitější. Portál jednak umí vygerovat bod, který leží uvnitř stejného regionu, jako je ten, ve kterém se agent v dané chvíli nachází. Poté, co se dostane do blízkosti tohoto bodu, je vygenerován druhý bod nacházející se v sousedním regionu a podsunut agentovi jako cíl jeho cesty. Tím je realizován přechod mezi dvěma sousedními regiony, který by jinak nebyl možný, neboť plánování cest, po kterých se agenti pohybují, lze provádět pouze v rámci jednotlivých regionů.

Dalším problémem, který je v souvislosti s portály a branami třeba řešit, je to, že leží velmi často v blízkosti nějakých zdí (často se totiž jedná o průchod). A protože v těsném okolí překážek mají agenti zakázano plánovat si cesty, je nejdříve třeba oblasti vhodné pro generování bodů odpovídajícím způsobem ořezat. Tím je zajištěno, že cílem cesty bude korektní pozice. Ořezání se provádí poté, co jsou všechny překážky rasterizovány do mřížkové mapy, prostým zmenšením oblasti portálu (resp. brány) o zakázaná políčka.

5.3.4 Třída `Region` a reprezentace map

Třída `Region` slouží k reprezentaci jedné oblasti světa. Tato oblast má tvar obdélníku, jehož strany jsou rovnoběžné s osami souřadnic. Hlavním úkolem regionu je poskytovat agentům datové struktury umožňující efektivní plánování cest a podávat jim infomaci o překážkách v jejich okolí. Pro tyto účely byly vyvinuty dva druhy map. Jedná se o potomky abstraktních tříd `PathPlanningMap` a `ObstaclePerceptionMap`. Hierarchii, do které jsou tyto mapy uspořádány, lze vidět na obr. 5.6. Rozhraní obou typů map implementuje třída `GridMap`, která pro ukládání dat používá dvourozměrné pole. Pouze pro plánování cest potom slouží `QuadtreeMap`, která data ukládá ve formě kvadrantového stromu.

Fakt, že mřížková mapa (tj. `GridMap`) implementuje jak rozhraní třídy `PathPlanningMap`, tak i třídy `ObstaclePerceptionMap`, je velice důležitý z hlediska šetření paměti. Aby totiž bylo vnímání i plánování cest dostatečně přesné, délka strany jedné buňky se pohybuje v řádu několik desítek centimetrů (to dává např. pro svět o rozměrech 100m $\times$ 100m a pro buňku velikosti 25cm $\times$ 25cm celkem $1.6 \cdot 10^5$ buněk). Taktéž z důvodu šetření paměti je buňka představována pouhým jedním ukazatelem na překážku. Je-li buňka obsazena, míří tento ukazatel přímo na objekt, který překážku reprezentuje. Jestliže je buňka mapy schůdná, odpovídající ukazatel je nulový. A v pří-



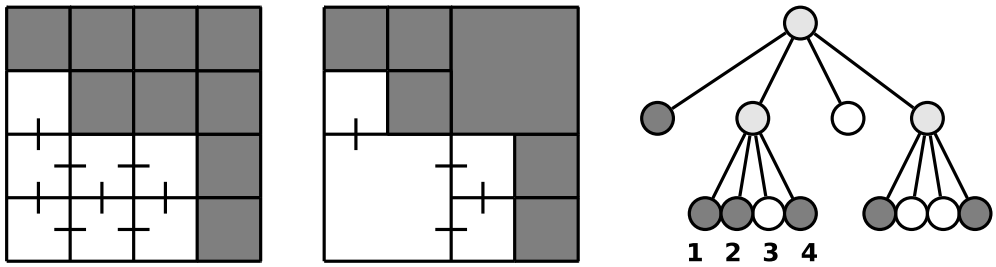
Obrázek 5.6: Mapy pro plánování cest v rámci regionů a pro podporu vnímání okolních překážek.

padě, že se jedná o hraniční buňku (tj. o buňku v blízkosti překážky, přes kterou je ovšem zakázáno plánovat cesty), obsahuje příslušný ukazatel speciální adresu, jež by neměla být dynamicky alokovanému objektu nikdy přidělena.

Jako první je vytvořena mřížková mapa. Na počátku jsou všechny její buňky označeny jako prázdné. Ve chvíli, kdy je přidána nová překážka zasahující do daného regionu, je třeba promítnout ji odpovídajícím způsobem do mapy. Jelikož každá překážka musí mít definovaný obal ve tvaru polygonu či kruhu, je pro jednoduchost dále uvažován jenom ten. V závislosti na typu obalu je vytořena instance jednoho z potomků abstraktní třídy `Renderer`. Tito potomci umožňují do dvourozměrné mřížky rasterizovat konkrétní obal, pro který byla jejich instance zkonstruována. Takto jsou do mapy regionu postupně přidány obaly jejich překážek. Poté jsou buňky mřížky v jistém malém okolí obsazených políček označeny jako hraniční, čímž je tvorba mřížkové mapy dokončena.

Výsledná mřížková mapa je poté použita pro stavbu kvadrantového stromu. Stavba probíhá směrem zdola nahoru. V prvním kroku je vytvořeno nejnižší patro stromu, přičemž jeden uzel stromu odpovídá čtvercové skupině buněk mřížkové mapy. Jestliže jsou všechny buňky schůdné (tedy v řeči v mřížkové mapy na nich není překážka a nejedná se ani o hraniční buňky nějaké překážky), je označen daný uzel také jako volný. Jsou-li všechny buňky obsazené, je uzel označen jako obsazený. Konečně vyskytují-li se v dané skupině buňky obou typů, je uzel označen jako smíšený, při plánování cest se s ním však počítá jako s obsazeným. Tato situace ale v současné době nemůže nastat, neboť jeden uzel stromu při stavbě nejnižšího patra odpovídá jedné buňce mřížkové mapy. Dále je pro každý volný uzel spodního patra stromu vytvořen seznam volných sousedních uzlů z jeho 4-okolí. Všechny uzly spodního patra jsou nakonec označeny jako listy.

Při stavbě každého dalšího patra stromu se potom využívá patra předchozího. Postup slučování uzlů spodnějšího patra v uzly patra horního je dobře patrný z obr. 5.7. Čtveřice schůdných listů je nahrazena větším schůdným listem, stejně tak čtveřice obsazených listů větším obsazeným listem. Ve všech ostatních případech čtveřice uzlů nahrazována není, pouze je k ní shora připojen vnitřní uzel smíšeného typu. Pro nový



Obrázek 5.7: Vlevo: Stavba dvou po sobě jdoucích pater kvadrantového stromu se znázorněním hran do volných sousedních uzlů. Vpravo: Výsledný kvadrantový strom. Očíslování uzlů odpovídá jednotlivým kvadrantům.

volný list je ještě potřeba vytvořit seznam jeho schůdných sousedů. Ten vznikne sloučením seznamů všech čtyř poduzlů, ze kterých byl vytvořen. Navíc je nutné aktualizovat i seznamy patřící sousedním uzlům a nahradit v nich odkazy na „malé“ listy odkazem na nově vytvořený „velký“ list. Také tohle je pěkně vidět na obr. 5.7. Ve chvíli, kdy postupným přidáváním pater dosáhneme jediného uzlu obsahujícího plochu celého regionu, stavba stromu končí a náš uzel je prohlášen za jeho kořen.

Jak již bylo řečeno, kromě plánování cest umožňuje `GridMap` agentům získávat přehled o překážkách v jejich okolí. To probíhá tak, že z buňky odpovídající pozici agenta jsou do mřížky postupně rasterizovány paprsky do požadovaných směrů. Protože takových parpsků je v závislosti na požadované přesnosti vnímání vysláno v každém kroku simulace a pro každého agenta třeba až 16 (zorný úhel 150°, paprsky po 10°), bylo třeba, aby rasterizace probíhala pokud možno co nejrychleji. Z tohoto důvodu byl zvolen celočíselný Bressenhamův algoritmus implementovaný třídou `LineBressenham`. Tato třída je podrobněji popsána dále.

5.4 Modul `Simulation`

Modul `Simulation` je nejrozsáhlejší a programátorsky zřejmě nejzajímavější částí celého systému. Zahrnujeme do něj třídy pro správu a synchronizaci jednotlivých simulačních vláken, třídy reprezentující agenty i třídy umožňující jejich správu a náhodné generování.

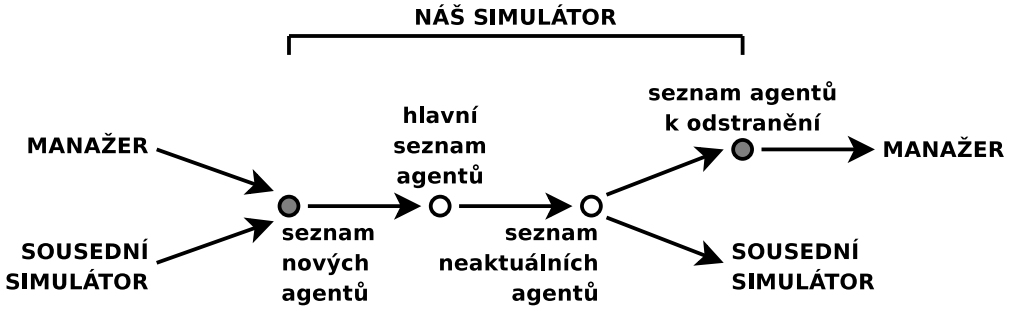
5.4.1 Třídy `SimManager` a `Simulator`

Hlavním úkolem třídy `SimManager` je poskytovat třídě `Controller` jednoduché rozhraní pro potřeby ovládání simulace. Nejdříve je nutné vytvořit struktury potřebné pro běh simulace a výměnu informací mezi jednotlivými simulačními vlákny, za což je zodpovědná funkce `init()`. Provedením této funkce také dojde k rozdělení virtuálního světa mezi jednotlivá simulační vlákna, představovaná instancemi třídy `Simulator`. Zavoláním funkce `start()` manažeru říkáme, aby všechny simulátory spustil. Pozastavit provádění simulace lze zavoláním metody `pause()`, znovu rozběhnout pomocí `run()`, o

úplné ukončení se potom stará funkce `stop()`.

Kromě toho je uvnitř třídy manažeru uložen seznam ukazatelů na všechny agenty vyskytující se ve virtuálním světě. Je to proto, že hlavní vlákno aplikace se prostřednictvím třídy `SimManager` stará o základní správu agentů. Je-li nový agent přidáván do simulačního procesu, uloží se nejdříve do tohoto globálního seznamu a poté jej manažer na základě jeho pozice deleguje příslušnému simulátoru. Stejně tak při odstranění agenta ze simulace se jej příslušný simulátor pouze vzdá a podá o tom správu manažeru simulátorů, který se už postará o jeho korektní zrušení. Z toho důvodu `SimManager` definuje veřejnou funkci `update()`, která tuto správu agentů provádí a které je volána v hlavním vláknu jednou za určitý časový interval.

Jedno vlákno zodpovědné za simulaci mu přidělené části světa je symbolizováno třídou `Simulator`. Ta definuje, podobně jako manažer, funkce `start()`, `pause()`, `run()` a `stop()`. Každá z těchto funkcí je volána z jí odpovídající funkce manažeru.



Obrázek 5.8: Seznamy uvnitř simulátoru a pohyb ukazatelů na agenty mezi nimi; šedě označené seznamy je třeba zamykat.

Každá instance třídy `Simulator` si drží čtyři různé seznamy agentů. Prvním z nich je hlavní seznam agentů, jejichž simulaci má v danou chvíli na starost. Dále se jedná o seznam nových agentů, a to jak těch, kteří přišli z oblasti jiného simulátoru, tak i těch, kteří jsou úplně noví ve virtuálním světě. Přístup k tomutu seznamu musí být hlídán mutexem. Agenty do něj totiž přidává hlavní simulační vlákno i sousední simulátory, slouží jim k tomu metoda `addAgent()`. Kromě toho si ze seznamu nových agentů náš simulátor na začátku každého simulačního cyklu agenty vyzvedává a zařazuje je do svého hlavního seznamu. Do třetího seznamu jsou zařazování agenti, kteří v průběhu cyklu přestanou být pro simulátor zajímaví (buďto vyšli z části světa mu přidělené, nebo opustili svět úplně). Současně se zařazením do tohoto seznamu je třeba agenta vyjmout ze seznamu hlavního. Na konci cyklu jsou potom neaktuální agenti buďto předáni jinému simulátoru (voláním jeho metody `addAgent()`), nebo přidáni do čtvrtého seznamu agentů určených k odstranění ze světa. Také přístup k tomuto seznamu musí být hlídán mutexem, neboť kromě našeho simulačního vlákna k němu přistupuje vlákno hlavní prostřednictvím manažeru `SimManager`. Ten si odtud agenty vyzvedne a provede jejich korektní uvolnění. Vše je pěkně vidět na obr. 5.8.

5.4.2 Mixiny

V této sekci představíme jeden nepříliš známý programátorský idiom — mixin [26]. Jedná se o třídu, jejímž účelem není existovat osamoceně, ale stát se součástí jiné třídy a předat jí tak svou funkcionalitu. Takových mixinů potom můžeme mít několik, každý s jiným účelem, a z nich složit různé výsledné třídy, aniž by docházelo k duplicitám v kódu. Termín mixin se poprvé objevil v souvislosti s programovacím jazykem Lisp, postupně se však objevovaly implementace i v ostatních jazycích. Jednou z možností, jak používat tento idiom v C++ je vícenásobná dědičnost, kdy je výsledná třída potomkem jednotlivých mixinů.

Druhým způsobem je potom napsat mixin jako šablonovou třídu parametrizovanou typem svého předka:

```
template <class Super>
class Mixin : public Super
{
   ... tělo třídy Mixin ...
};
```

Jasnou výhodou tohoto přístupu je, že se vícenásobné dědičnosti zbavíme, výsledná třída je totiž vytvořena skládáním jednotlivých mixinů „za sebe“. Naopak nevýhodou může být to, že z konstruktoru třídy `Mixin` zpravidla nelze volat jiný než implicitní konstruktor třídy `Base`, jelikož její typ není znám. Tím pádem ani nelze říct, jaké parametry bude konstruktor předka `Super` vyžadovat. Tuto nepříjemnost lze vyřešit tak, že se vždy volá bezparametrický konstruktor mixinu a jeho faktická inicializace proběhne až voláním funkce k tomu určené. Výsledná složená třída tak inicializuje všechny předky v hirarchii přímo, např. v těle svého konstruktoru.

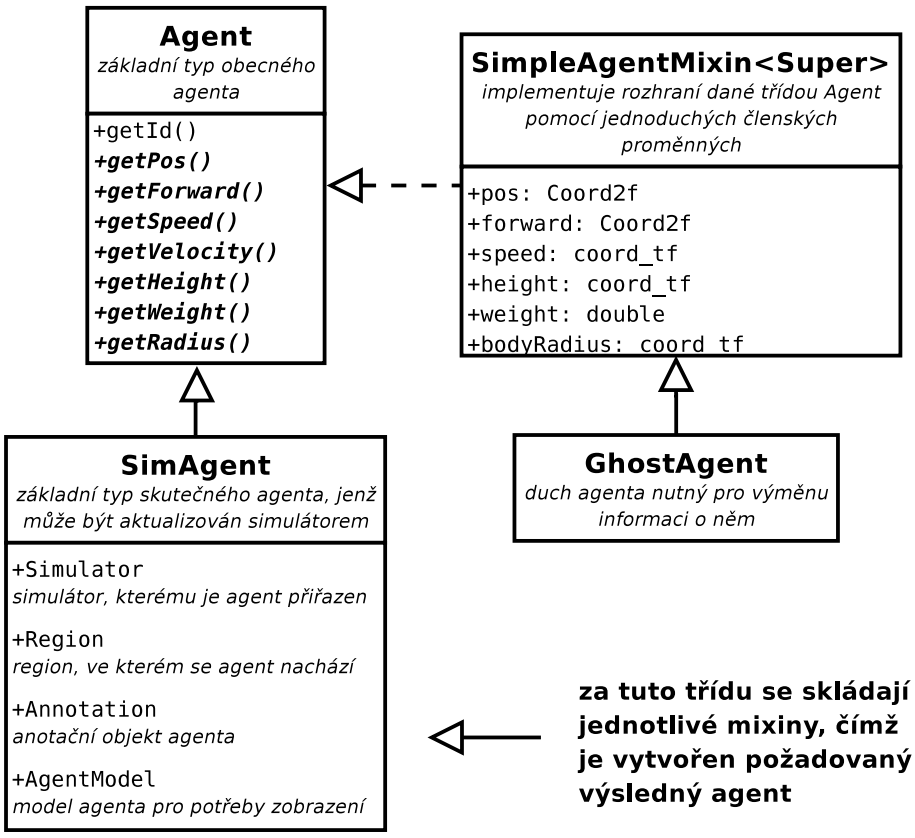
Pro daný mixin je velmi jednoduché psát šablonové funkce, které jej berou jako jeden z parametrů:

```
template <class Super>
void function(const Mixin<Super>& arg) { ... }
```

V závislosti na použité hierarchii, do které jsou jednotlivé mixiny složeny, je potom vygenerována konkrétní funkce, případně více funkcí, existuje-li více hierarchií. V každém případě je opět zamezeno duplicitě kódu.

5.4.3 Základ hierarchie agentů

Objektový model agentů bylo třeba navrhnout velmi pečlivě, s důrazem na snadnou rozšiřitelnost a znovupoužitelnost. Jako základní třída, která musí být předkem každého agenta v aplikaci, slouží abstraktní `Agent`. Ta také určuje rozhraní v podobě čistě virtuálních funkcí, které musí každý agent poskytovat, aby s ním mohli ostatní komunikovat (viz obr. 5.9). Nad těmito virtuálními funkcemi dále třída `Agent` implementuje metody na predikci času do nejbližší kolize s jiným agentem, predikci času do okamžiku, kdy bude vzdálenost dvou agentů nejmenší, určení vzájemné polohy dvou agentů (napravo, nalevo, vepředu, vzadu) a jiné. Jedná se především o různé matematické výpočty, které



Obrázek 5.9: Základ hierarchie tříd pro tvorbu jednotlivých agentů.

jsou podrobně okomentovány přímo v kódu. Kromě toho každý základní agent obsahuje členskou proměnnou, která jej pro potřeby simulace identifikuje.

Jednu z možných implementací rozhraní daného třídou `Agent` dává šablonová třída `SimpleAgentMixin<Super>`, naprogramovaná jako mixin. Ta si drží stav agenta, jako je např. jeho rychlost, pomocí několika číselných členských proměnných (viz obr. 5.9). Lze si ale představit i implementaci jinou, která by namísto přesné rychlosti obsahovala pouze nějakou diskrétní veličinu (agent stojí, jde pomalu, jde rychle, běží) a číselnou hodnotu určovala např. pomocí fuzzy pravidel.

Mixinu `SimpleAgentMixin<Super>` využívá třída `GhostAgent`, která slouží pro potřeby kopírování agentů nacházejících se v hraničních zónách simulátorů. Duch nemůže být aktualizován, slouží pouze k tomu, aby na základě jeho stavu mohli být aktualizováni agenti v jeho okolí. Proto také nemá každý duch jednoznačný identifikátor, namísto toho obsahují všichni stejnou hodnotu identifikátoru, která naznačuje, že se nejedná o reálného agenta.

Přímým potomkem základní třídy `Agent` je `SimAgent`. Ten musí být předkem každého typu agenta, jenž má být simulován. `SimAgent` si pamatuje ukazatel na třídu `Region` (tedy je ve světě umístěn) a na `Simulator` (tedy je přiřazen určitému simulačnímu vláknu). Kromě toho může taktéž prostřednictvím ukazatele přistupovat ke své anotaci (anotace jsou vysvětleny dále).

Za třídu `SimAgent` se potom skládají jednotlivé mixiny. Prvním většinou bývá `SimpleAgentMixin<Super>`, nebo jiný mixin implementující rozhraní třídy `Agent`, dále pokračujeme dle požadovaných vlastností agenta. Přestože se v simulaci zatím vyskytují pouze chodci, v budoucnosti by mohli přibýt třeba prodavači ve stánku s občerstvením. Pro ně by byl mixin podporující plánování cest zbytečný, namísto toho by spíše potřebovali mixin implementující prodej párků v rohlíku. Nebo by v některých částech světa mohly viset nástěnné mapy, podle nichž by si agenti plánovali cesty. Tyto mapy by mohly mixin na plánování cest obsahovat, naopak by určitě nepotřebovaly mixin implementující vnímání okolních agentů. Systém je tak snadno rozšiřitelný o nové typy agentů (a obecně objektů) prostým naskládáním již vytvořených mixinů „za sebe“.

5.4.4 Plánování cest

Aby mohl každý agent používat jiný algoritmus na plánování cest, byla definována šablona `PathPlanningAlgorithm<Map>` parametrizovaná typem mapy, nad kterou algoritmus běží. Ta deklaruje čistě virtuální funkci `planPath()`.

Potomkem obecného algoritmu na plánování cest je obecná implementace algoritmu A* představovaná šablonovou třídou `AStarAlgorithm<Map, Node, NodeAllocator>`. Jak lze vidět, je parametrizovaná typem mapy, nad kterou se má cesta plánovat, typem uzlů, které algoritmus A* při hledání používá, a externím alokátorem těchto uzlů. Takto obecná definice algoritmu A* nám umožní vytvořit jeho specializaci pro danou mapu prostou implementací několika čistě virtuálních metod, které šablona deklaruje. Jsou to metody na vytvoření počátečního uzlu, koncového uzlu, expanzi uzlu a výpočet pozice ve světě odpovídající danému uzlu. Vedle obecné implementace algoritmu A* je dále definována třída `AStarNode`, která slouží jako předek konkrétního typu `Node`. Dalšími pomocnými strukturami, které algoritmus A* pro svůj běh potřebuje, jsou množina uzlů určených k expanzi a množina již navštívených a expandovaných uzlů. Uvnitř šablony `AStarAlgorithm<Map, Node, NodeAllocator>` proto najdeme jejich základní implementaci `OpenSet` a `ClosedSet`. `ClosedSet` využívá datové struktury `std::list`, `OpenSet` potom `std::vector`, nad kterou pomocí algoritmů STL udržuje haldu.

Nyní je už jednoduché implementovat algoritmus A* nad mřížkovou mapou i nad kvadrantovým stromem. Nejdříve vytvoříme třídy `GridAStarNode` a `QuadtreeAStarNode`, potomky obecného uzlu `AStarNode`. První z nich si pamatuje index políčka mřížkové mapy, které představuje, druhá si drží ukazatel na odpovídající list kvadrantového stromu. Uzly těchto typů jsou potom využity třídami `GridAStarAlgorithm` a `QuadtreeAStarAlgorithm`, které odpovídajícím způsobem implementují výše zmíněné čtyři virtuální metody obecného algoritmu A* nad mapami `GridMap` a `QuadtreeMap`. Jako alokátor uzlů používají obě dvě šablonu `PoolAllocator<T>`, jež je popsána v kapitole 5.5.5. Tento alokátor je ve skutečnosti majetkem instance třídy `Simulator` a algoritmus toho kterého agenta si jej pouze „půjčuje“. Tím je zajištěno, že k jednomu alokátoru bude přistupovat pouze jedno vlákno, takže není třeba při plánování cest provádět žádnou synchronizaci.

Všechno, co agent potřebuje k plánování cest, je zabaleno do mixinu `PathMixin<Super>`. Ten obsahuje instanci třídy `QuadtreeAStarAlgorithm`, třídy `GridAStarAlgorithm` (kdyby hledání v kvadrantovém stromě selhalo) a cestu ve formě

instance třídy `Path`. Ta je tvořena seznamem bodů, instancí třídy `Waypoint`, a kromě přístupu k nim pomocí iterátorů definuje operace, jako je výpočet délky cesty, získání bodu na cestě nejbližšího dané pozici apod. Tyto metody jsou potom využívány agentem, který cestu sleduje.

5.4.5 Paměť a vnímání

Paměť agenta je realizována šablonovou třídou `PerceptionMemory<T, int>`. První parametr určuje typ objektů ukládaných do paměti, druhý maximální možnou kapacitu paměti. Jednotlivé vjemy jsou uchovávány v poli a lze k nim přistupovat pomocí iterátorů, které šablona definuje. Navíc poskytuje základní operace, jako je přidání nového vjemu, vyčištění obsahu paměti apod.

Představená šablona je využita hned dvakrát. Jednou ve formě specializace `AgentPerceptionMemory`, uchovávající ukazatele na viděné okolní agenty, podruhé jako specializace `ObstaclePerceptionMemory`, do které se ukládají instance speciální třídy `ObstaclePerception` obsahující informaci o „vjemu“ překážky (pozici, na které byla spatřena, úhel, pod kterým ji agent viděl apod.). Rozdíl proti paměti obsahující okolní agenty je ten, že jedna překážka může být uložena víckrát, a to ve formě několika různých vjemů.

Aby potom mohl agent oba dva druhy paměti i funkce poskytované třídami `Region` a `Simulator` pro potřeby vnímání snadno používat, je všechno potřebné zabaleno do dvou mixinů. Jsou to `AgentPerceptionMixin<Super>` a `ObstaclePerceptionMixin<Super>`.

5.4.6 Chování agentů

Pro potřeby reaktivního chování byly vytvořeny dva různé mixiny. První z nich, založený na přístupu řídících sil, je `SteeringBehaviorMixin<Super>`. Druhé chování, založené na metodě sociálních sil, je realizováno šablonou `SocialForcesMixin<Super>`. Obě třídy kromě jednotlivých metod vyhodnocujících síly působící na agenta samozřejmě obsahují i členské proměnné ovlivňující parametry chování, lze je tedy opět pouze přidat do hierarchie součástí agenta a snadno používat.

Kromě tohoto základního reaktivního chování, je v projektu pomocí šablonové třídy `GoalDrivenBrainMixin<Super>` implementován jednoduchý „mozek“ agenta. Ten si udržuje ve struktuře `std::stack` ukazatele na objekty reprezentující jednotlivé cíle, o jejichž splnění se agent snaží. Základem hierarchie cílů je abstraktní třída `Goal`, konkrétní cíle jsou potom jejími potomky. Třída `Goal` deklaruje několik čistě virtuálních funkcí, nutných pro práci s cíly. Jedná se zejména o funkci zjišťující, zda bylo daného cíle agentem již dosaženo, a o funkci, která zajistí rozložení cíle na podcíle nutné k jeho splnění.

5.4.7 Anotace

Výsledky simulace je třeba zobrazovat pomocí modulu `Viewer`. Při vykreslování překážek či jiných informací o statických částech světa nenastává žádný problém. Avšak

při vykreslování těl agentů či znázorňování jejich vnitřních stavů je třeba uvážit, že k těmto datům přistupuje jak hlavní vlákno, které se o vykreslování stará, tak příslušné simulační vlákno, kterému je agent v danou chvíli přiřazen. Dle stupně požadované synchronizace těchto vláken je program možné zkompilovat ve dvou verzích na základě direktivy preprocesoru: jedné rychlejší a druhé pomalejší, avšak s většími možnostmi vykreslování.

První možností je žádnou synchronizaci mezi vlákny neprovádět. Hlavní vlákno totiž pro potřeby vykreslování data jednotlivých agentů pouze čte, nemůže tedy dojít k zápisu ze dvou vláken ve stejný moment. Je ale možné znázorňovat pouze ta data, která jsou reprezentována primitivními typy, jako je pozice agenta v kartézských souřadnicích. Zápis do takové proměnné sice také nemusí být atomický (např. pro typ `double`), na druhou stranu nemůže dojít k žádné nebezpečné chybě. Jestliže hlavní vlákno načte hodnotu proměnné v průběhu zápisu do ní, ta bude obsahovat stále platné číslo, byť třeba nesmyslné. Agent by tak mohl být na jednom snímku zobrazen na špatné pozici, což ale příliš nevadí. Hlavní je, že simulace bude probíhat stále korektně. Horší by to ale bylo třeba s výpisem paměti překážek v okolí agenta reprezentované pomocí pole. Mohlo by snadno dojít k tomu, že hlavní vlákno bude do pole přistupovat zrovna ve chvíli, kdy bude měněn počet položek v něm, a hlavní vlákno se dostane mimo rozsah pole. Proto zrovna obsah paměti bez synchronizace zobrazovat nejde.

Právě kvůli pokročilým možnostem zobrazení stavů agentů existuje druhá možnost, která využívá techniku double-bufferingu. Každý agent obsahuje ukazatel na objekt anotace, potomka třídy `Annotation`, složený z anotací jednotlivých částí agenta. Anotace každé části si pak pamatuje zjednodušený stav té které části, např. anotace paměti překážek v okolí agenta obsahuje jejich počet a pozice, na kterých byly spatřeny. Seznam anotací pro agenty v jeho kompetenci si drží každý simulátor. Poté co jsou během simulačního cyklu aktualizováni všichni agenti, je seznam anotací uzamčen mutexem a dle stavu agentů jsou aktualizovány i jejich anotace. Prohlížeč potom pro zobrazení agentů používá pouze jejich anotace, k nimž přistupuje právě pomocí uzamykatelných seznamů v jednotlivých simulátorech. Tím je zajištěno, že všechny zobrazované informace budou vždy korektní a hlavně že nedojde k žádnému čtení neplatné paměti u složitějších struktur.

5.5 Ostatní moduly

5.5.1 Modul `Viewer`

Modul `Viewer` slouží ke zobrazování výsledků simulace. Základem tohoto modulu je stejnojmenná abstraktní třída deklarující dvě čistě virtuální funkce. První z nich je funkce `draw()`, která prohlížeči říká, aby celou scénu „vykreslil“. Uvozovky jsou zde použity záměrně, lze si totiž snadno představit, že jedna z možných implementací prohlížeče by třeba pouze vypsala pozice jednotlivých překážek a agentů do textového souboru. Druhou funkcí definující rozhraní obecného prohlížeče je potom `processEvent()`. Ta jako parametr dostává událost (instanci třídy `ViewerEvent` nebo jejího potomka), na kterou má zareagovat. Událostí může být např. změna pozice či rotace kamery, změna stylu

zobrazení agentů, zapnutí/vypnutí vykreslování překážek apod. Každá implementace prohlížeče si potom může vybrat množinu událostí, na které bude reagovat.

Třída `OpenGLViewer` implementuje rozhraní obecného prohlížeče za použití knihovny OpenGL. K vykreslování nejzákladnějších primitiv (úsečky, disky, koule apod.) používá statické funkce deklarované třídou `OpenGLSupport`. Postupným voláním těchto jednoduchých funkcí potom `OpenGLViewer` vykresluje jednotlivé objekty virtuálního světa.

K vykreslování koulí a válců byla původně používána knihovna GLU (OpenGL Utility Library), která umožňuje pro tyto účely vytvářet a vykreslovat tzv. kvadriky. Jelikož jsou ale kvadriky příliš pomalé, implementovali jsme raději vlastní funkce, které rozloží povrch koule či kvádru na jednotlivé trojúhelníky. Vykreslením těchto trojúhelníků je potom dosaženo zobrazení požadovaného tělesa. Další zrychlení přináší uložení modelů těl agentů do display-listů.

Do modulu `Viewer` zahrnujeme také třídu `OpenGLTextViewer`, která umí vypisovat text do okna přiděleného OpenGL. Lze ji zkompilovat ve dvou verzích. První z nich je přenositelná a využívá bitmapového fontu definovaného přímo v kódu programu. Ta je také implicitně nastavena. Lze však zkompilovat i verzi jinou, která při inicializaci vezme font operačního systému Windows a pro každý tisknutelný znak vytvoří jednu bitmapu. Bitmapy nakonec pro potřeby rychlejšího vykreslování uloží do display-listů.

5.5.2 Modul `Log`

Účelem tohoto modulu je poskytovat rozhraní pro textové výpisy. Základní třídou modulu je abstraktní `BaseLog`. Ta definuje výčtové typy určující „upovídanost“ daného logu a důležitost vypisované zprávy a deklaruje čistě virtuální funkci `print()`. V současnosti existují tři potomci definující tuto metodu. Prvním z nich je `LogFile`, který vypisuje zprávy do zadaného souboru, druhým `LogConsole`, jenž pro výstup používá konzoli. Třetí implementací je potom `MemoryLog`. Tato třída si přijaté zprávy ukládá ve formě řetězců `std::string`.

Právě třídu `MemoryLog` využívá tzv. virtuální konzole (statická třída `VirtualConsole`). Ta si uložené zprávy z paměti vyzvedává, aby je `OpenGLViewer` mohl vypsat do okna s grafickým výstupem. Kromě toho `VirtualConsole` uchovává informace o časovém průběhu simulace, o zatíženosti jednotlivých simulátorů a o počtu snímků vykreslených prohlížečem za jednotku času.

Jednotlivé logy lze organizovat do skupin (instancí třídy `LogGroup`). Je-li potom nějaká zpráva takové skupině předána, vypíše se do všech logů v ní obsažených.

Nejvýše stojí statická třída `Logger` spravující jednotlivé logy a jejich skupiny. Hlavním jejím úkolem je zajišťovat k nim exkluzivní přístup, neboť jeden log může být použit i ve více vláknech současně. Proto by k logům a jejich skupinám nemělo být nikdy přistupováno přímo, ale pouze prostřednictvím jejich identifikátorů a statických metod třídy `Logger`.

5.5.3 Modul `Graphics2D`

Jak již bylo řečeno, každá překážka vyskytující se ve virtuálním světě musí mít definovaný svůj obal. Obalem je přitom myšlen určitý obrazec, potomek abstraktní třídy `Shape`. Tento obal je nutný z toho důvodu, že hranice překážky by mohla být ve skutečnosti velmi členitá a jako s takovou by nebylo snadné s ní pracovat. Abstraktní třída `Shape` definuje rozhraní, které takový obrazec musí splňovat. Jde zejména o geometrické funkce provádějící posunutí obrazce, expanzi jeho hranice, výpočet nejbližšího bodu hranice k danému vnějšímu bodu, výpočet normály hranice v tomto nejbližším bodu, určení průsečíku obrazce s polopřímkou apod.

Co je velmi důležité pro reprezentaci překážky v mřížkové mapě, obrazec musí být schopen na základě svého typu vytvořit odpovídajícího potomka třídy `Renderer`, který dokáže vnitřní body obrazce či jeho hranici do této mapy rasterizovat. Pro tyto účely `Renderer` deklaruje čistě virtuální funkci `getNextPoint()`, která vrátí souřadnice dalšího bodu obrazce v rastru. Tím pádem je možno použít daného rendereru nad obecnou mřížkou.

V současné době jsou implementovány dva různé typy obrazců. Prvním z nich je polygon reprezentovaný stejnojmennou třídou `Polygon`) a mu odpovídající rasterizéry vnitřku `PolygonRenderer` a hraničních bodů `PolygonBorderRenderer`. Druhým obrazce je potom kruh reprezentovaný třídou `Circle` a rasterizér `CircleRenderer`.

Kromě obrazců a jejich rasterizérů modul obsahuje třídu `LineBressenham` implementující celočíselný algoritmus na kreslení úseček a polopřímek v rovině.

Třída `PolygonRenderer` implementující rozhraní dané abstraktní třídou `Renderer` používá k rasterizaci polygonu algoritmus řádkového vyplňování se seznamem aktivních hran. Principem algoritmu je vykreslovat vnitřní body polygonu postupně po řádcích od minimálního y k maximálnímu. K tomuto účelu si algoritmus pro každou hranu polygonu pamatuje záznam obsahující následující proměnné:

- dy: počet řádků, do nichž hrana zasahuje
- x: souřadnice x průsečíku hrany s aktuálně vykreslovaným řádkem
- dx: přírůstek souřadnice x průsečíku při přechodu na další řádek

Záznamy o jednotlivých hranách algoritmus udržuje v tabulce hran (TH) uspořádané dle souřadnice y. Další datovou strukturou je seznam aktivních hran (SAH) obsahují pouze ty hrany, jež mají průsečík s aktuálně zpracovávaným řádkem. Princip fungování algoritmu lze zapsat takto (podrobněji potom v [2, 32]):

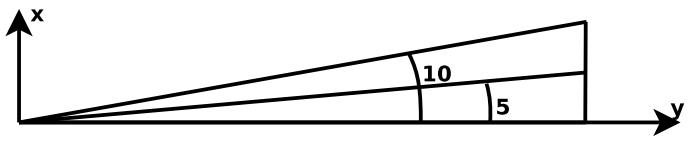
```
1. Inicializace:
   pro každou hranu polygonu
      je-li vodorovná
         vynechej ji a pokračuj
      uprav orientaci od ymin k ymax a zkraťji o 1 na straně ymax
   vytvoř tabulku hran TH uspořádanou dle y
   vytvoř prázdný seznam aktivních hran SAH uspořádaný dle x
   nastav y na první souřadnici y v TH

2. Běh algoritmu:
```

```
dokud nejsou TH i SAH prázdné
   přesuň do SAH hrany z TH[y]
   aktualizuj uspořádání SAH dle x
   vykresli úseky mezi lichými a sudými hranami v SAH
   zruš z SAH hrany, pro něž je dy == 0
   pro každou hranu z SAH
      dy = dy - 1
      x = x + dx
   y = y + 1
```

Rasterizaci kruhu reprezentovaného třídou `Circle` zajišťuje třída `CircleRenderer`. Ta na základě poloměru nejdříve určí čtvercovou oblast buněk mřížky, které by mohly ležet uvnitř kruhu (průměr kruhu odpovídá délce strany čtverce). Potom jednu po druhé tyto buňky testuje na vzdálenost od středu kruhu. Takto padne přibližně $\pi/4 \approx 80\%$ testovaných buněk do kruhu. Navíc tento algoritmus běží stejně pouze během inicializace, takže to, že je poněkud těžkopádný, ničemu nevadí.

Poslední třídou patřící do modulu `Graphics2D` je `LineBressenham` implementující celočíselný Bressenhamův algoritmus na kreslení čáry do mřížky. Touto čárou může být v závislosti na použitém konstruktoru buďto úsečka mezi dvěma body s celočíselnými souřadnicemi, nebo polopřímka vycházející z vrcholu s celočíselnými souřadnicemi pod daným úhlem zadaným ve stupních.



Obrázek 5.10: Příklad dvou trojúhelníků umožňujících parametrizovat modifikovaný Bressenhamův algoritmus na kreslení polopřímky úhlem, který polopřímka svíra s osou x.

Protože na polopřímky není originální Bressenhamův algoritmus „stavěný“, bylo třeba jej poněkud modifikovat. Během inicializace programu jsou pro každý celočíselný úhel z množiny $\{0°..359°\}$ do tabulky předpočítány takové parametry algoritmu, jako bychom chtěli kreslit přeponu pravoúhlého trojúhelníku svírající s osou x daný úhel (viz obr. 5.10 pro úhly 5° a 10°). Při požadavku na vykreslení polopřímky je potom úhel, který svírá s osou x, zaokrouhlen na celé stupně a použijí se parametry z odpovídajícího řádku tabulky. To zároveň umožňuje velmi rychlou inicializaci algoritmu, neboť parametry není třeba na začátku vyhodnocovat, stačí je načíst. Koncový bod polopřímky je nastaven v nekonečnu. Určitým omezením je, že lze kreslit polopřímky s přesností pouze 1°, což je ale pro potřeby simulace více než dostačující.

Další zajímavostí třídy `LineBressenham` je to, že neprovádí přímo rasterizaci úsečky či polopřímky do mřížky, ale namísto toho definuje funkci `getNextPoint()`, která vrátí souřadnice dalšího bodu. Tím pádem je možno algoritmus použít nad obecnou strukturou, ne pouze pro vyplňování mřížky třídy `GridMap`.

Symbolicky je Bressenhamův algoritmus pro řídící osu x (tedy pro úsečky a polopřímky s absolutní hodnotou směrnice menší než 1) možné zapsat takto (podrobnější

popis algoritmu včetně jeho odvození lze najít v [2, 32]):

```
1. Inicializace pro počáteční bod [x1,y1] a koncový bod [x2,y2]:
   dx = |x2 - x1|
   dy = |y2 - y1|
   k1 = 2 * dy          // přírůstek pro p <= 0
   k2 = 2 * (dy - dx)   // přírůstek pro p > 0
   p = 2 * dy - dx      // rozhodovací člen
   [x,y] = [x1,y1]      // počáteční bod

2. Generování každého dalšího bodu [x,y]:
   x = x + 1
   je-li p > 0
      p = p + k2
      y = y + 1
   je-li p <= 0
      p = p + k1
```

5.5.4 Modul `Math`

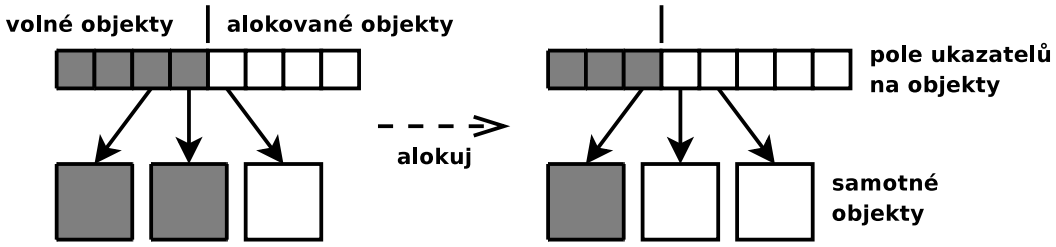
Jak z názvu modulu vyplývá, zahrnuje pomocné matematické třídy a funkce. Jedná se zejména o dvě šablonové třídy `Vector2D<T>` a `Vector3D<T>` a operace nad nimi. Pro potřeby simulace musely být dále implementovány funkce generující náhodná čísla s normálním rozložením, funkce provádějící různé druhy interpolací, geometrické funkce určující nejbližší bod úsečky, polygonu a kružnice apod. Důležitou třídou tohoto modulu je také statická `TableFunctions`, která pro vybrané domény uchovává hodnoty nejpoužívanějších funkcí v tabulkách. Tím za cenu obsazení určitého kusu paměti dosáhneme výrazného urychlení simulace.

5.5.5 Alokace malých objektů

Implicitní alokátor dynamicky přidělované paměti není primárně stavěn na alokaci mnoha malých kousků paměti. Místo toho je napsán dostatečně obecně, aby pracoval uspokojivě v každé situaci. Musí totiž fungovat jak pro programy běžící na serverech měsíce bez přerušení, tak i pro ty krátké trvající necelou sekundu. Musí umět alokovat jak velké objekty či pole, tak i malé proměnné o velikosti několika málo bytů [1, 12]. Z toho vyplývá, že pro velmi častou alokaci/dealokaci malých objektů zdaleka nefunguje ideálně. Je totiž jednak celkem pomalý, navíc poměrně dost plýtvá s pamětí. V [1] lze najít podrobně popsanou implementaci vlastního alokátoru.

Já jsem však zvolil poněkud odlišnou cestu. Místo aby si alokátor držel volnou surovou paměť, kterou by potom na požádání přiděloval, pamatuje si raději v poli ukazatele na předem vytvořené objekty konkrétního typu. Tím pádem je nutné, aby třída, jejíž instanceme chceme spravovat tímto alokátorem, měla definovaný bezparametrický konstruktor. Zavolaním funkce `allocate()` se část pole obsahující ukazatele na volné objekty zmenší o jedna a příslušný ukazatel je přidělen žadateli (viz obr. 5.11). Voláním funkce `deallocate()` je příslušný ukazatel opět přidán do pole volných ukazatelů. Tímto způsobem jsme dostali velice rychlý alokátor objektů daného typu, operace totiž

vyžadují pouze test jednoduché podmínky, přiřazení a inkrementaci/dekrementaci čítače.



Obrázek 5.11: Schéma alokátoru malých objektů a příklad jedné alokace.

Ve chvíli, kdy volné objekty dojdou, stačí vytvořit nové a odpovídajícím způsobem zvětšit pole ukazatelů. Jelikož je celá třída napsána jako šablona `PoolAllocator<T>`, je použitelná pro objekty jakéhokoliv typu s bezparametrickým konstruktorem.

Kapitola 6

Závěr

Cílem práce bylo navrhnout systém schopný paralelně simulovat velké množství chodců v rozsáhlém virtuálním prostředí a navržený model následně implementovat tak, aby simulace probíhala v reálném čase na běžném osobním počítači.

Představená hierarchická reprezentace virtuálního světa, který dělíme na obdélníkově regiony, se ukázala být velice výhodnou pro plánování cest. Přestože je algoritmus A* poměrně výkonný, zejména zvolíme-li dobře heuristiku aproximující vzdálenost do cílové pozice, ve velkých mapách může jeho provádění trvat dlouho. Dochází totiž k expanzi příliš mnoha uzlů. Omezíme-li však prohledávanou oblast na jeden region, je možno plánovat velké množství cest v reálném čase. Snad ještě důležitější pro rychlost algoritmu A* je použití map ve formě kvadrantových stromů. Jejich výhoda je tím větší, čím méně překážek region obsahuje. Rozsáhlé plochy bez překážek, na jejichž reprezentaci by mřížková mapa potřebovala třeba i stovky buněk, jsou totiž potom často obsaženy v jednom schůdném listu.

Způsob, kterým agenti vnímají své okolí, byl také zvolen velmi dobře. Jednak poměrně dost věrně simuluje proces vidění skutečného člověka, když bere v potaz pouze blízké objekty nacházející se v zorném poli agenta. Navíc je velmi efektivní a pro pevně dané parametry vnímání a pro jednoho agenta pracuje v konstantním čase. Nezávisí tedy na počtu simulovaných agentů či na velikosti virtuálního prostředí.

Z psychologického hlediska je zajímavé použití omezené paměti objektů, které agent bere v potaz při svém pohybu. To nám zároveň zajišťuje pro danou velikost paměti konstantní čas nutný pro simulaci jednoho cyklu pohybu agenta. Celkově tedy simulace probíhá lineárně vzhledem k počtu agentů. V mnoha modelech, jež byly představeny v začátku práce, bývá přitom asymptotická složitost kvadratická, neboť se uvažuje každá dvojice agentů.

Dalším významným vylepšením proti většině ostatních modelů je paralelizace simulace, která vůbec nebývá běžná. V dnešní době se navíc zdá, že vývoj procesorů nejde cestou neustálého zvyšování frekvence, jak tomu bylo dříve, ale zvětšováním počtu jader. Proto má v sobě náš model poměrně velký potenciál do budoucna.

Reaktivní chování agentů založené na řídících silách je poměrně jednoduché a jedná se zřejmě o nejslabší článek celé simulace. Problémem je zde mimo jiné správné nastavení parametrů určujících, kdy se má která síla použít, a nastavení konstant ovlivňujících výpočty jednotlivých sil. Kromě modelu řídících sil byly vyzkoušeny ještě sociální síly

popsané zběžně v kapitole 1.3.1. Jelikož ale sociální síly byly primárně navrženy pro potřeby simulace evakuace budov pomocí nouzových východů, nedokázaly generovat věrohodný pohyb agentů. Proto také nebyly v práci nijak podrobněji rozebírány.

Naproti tomu model rozhodování agentů založený na zásobníku cílů se velmi osvědčil a zajisté by dobře fungoval i pro simulaci nějakých složitějších sociálních interakcí mezi agenty. Cílem totiž může být prakticky jakákoliv činnost agenta. Kdybychom chtěli zajít ještě dále, mohl by každý agent mít nějaké potřeby a na jejich základě zásobník cílů průběžně modifikovat. To by zřejmě vedlo k přidání ještě jedné, vyšší vrstvy.

Literatura

[1] Alexandrescu A. (2001): Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, Reading.

[2] Foley J., van Dam A., Feiner S., Hughes J. (1990): Computer Graphics — Principles and Practice. Addison-Wesley, Reading.

[3] Fikes R., Nilsson N. (1971): STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2, str. 189–208.

[4] Funge J., Tu X., Terzopoulos D. (1999): Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. Proceedings of SIGGRAPH 99, str. 29-–38.

[5] Gipps P. G., Marksjo B. (1985): A Micro-Simulation Model for Pedestrian Flows. Mathematics and Simulation 27, str. 95–105.

[6] Goffman E. (1971): Relations in Public: Microstudies of the Public Order. Basic Books, New York.

[7] Hart P. E., Nilsson N. J., Raphael B. (1968): A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics SSC4 (2), str. 100—107.

[8] Helbing D., Molnár P. (1995): Social force model for pedestrian dynamics. Physical Review E 51, str. 4282–4286.

[9] Helbing D., Farkas I., Vicsek T. (2000): Simulating Dynamical Features of Escape Panic. Nature 407, str. 487–490.

[10] Le Bon G. (1997): Psychologie davu. KRA, Praha.

[11] Mařík V., Štěpánková O., Lažanský J. a kol. (2004): Umělá inteligence (1). Academia, Praha.

[12] Meyers S. (2005): Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley, Reading.

[13] Miller, G. A. (1956): The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review 63, str. 81-–97.

[14] Okazaki S.(1979): A Study of Simulation Model for Pedestrian Movement in Architectural Space. Transactions of Architectural Institute of Japan, str. 283–285.

[15] Prata S. (2004): Mistrovství v C++. Computer Press, Brno.

[16] Quinn M. J., Metoyer R. A., Hunter-Zaworski K. (2003): Parallel Implementation of the Social Forces Model. Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, str. 63–74.

[17] Reynolds C. W. (1987): Flocks, Herds, and Schools: A Distributed Behavioral Model. SIGGRAPH '87 Conference Proceedings, str. 25–34.

[18] Reynolds C. W. (1999): Steering Behaviors For Autonomous Characters. Game Developers Conference 1999 Proceedings, str. 763–782.

[19] Reynolds C. W. (2006): Big Fast Crowds on PS3. Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, str. 113–121.

[20] Russell S. J., Norvig P. (1995): Artificial Intelligence: A Modern Approach. Prentice Hall, New Jersey.

[21] Sakuma T., Mukai T., Kuriyama S. (2005): Psychological model for animating crowded pedestrians. Computer Animation and Virtual Worlds 16, str. 343–351.

[22] Shao W., Terzopoulos D. (2005): Autonomous Pedestrians. Eurographics/ACM SIGGRAPH Symposium on Computer Animation.

[23] Shao W., Terzopoulos D. (2005): Environmental Modeling for Autonomous Virtual Pedestrians. SAE Symposium on Digital Human Modeling for Design and Engineering.

[24] Shao W. (2006): Animating Autonomous Pedestrians. PhD Thesis, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University.

[25] Schreiner D., Woo M., Neider J., Davis T. (2007): OpenGL Programming Guide: The Official Guide to Learning OpenGL. Addison-Wesley, Reading.

[26] Smaragdakis Y., Batory D. (2000): Mixin-Based Programming in C++. University of Texas at Austin CS Technical Report, str. 98–27.

[27] Sobel R. S., Lillith N. (1975): Determinant of Nonstationary Personal Space Invasion. Journal of Social Psychology 97, str. 39–45.

[28] Still G. K. (2000): Crowd Dynamics. PhD Thesis, University of Warwick.

[29] Sutter H., Alexandrescu A. (2004): C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley, Reading.

[30] Wolff M. (1973): Notes On The Behaviour Of Pedestrians. Peoples In Places: The Sociology Of The Familiar, str. 35–48, New York.

[31] Wright R. S., Lipchak B., Haemel N. (2007): OpenGL SuperBible: Comprehensive Tutorial and Reference. Addison-Wesley, Reading.

[32] Žára J., Beneš B., Sochor J., Felkel P. (2004): Moderní počítačová grafika. Computer Press, Brno.

[33] Roland Hess: Blender People.
http://www.harkyman.com/bp.html

[34] Mohsen Mousavi: CrowdIT.
http://www.crowdit.worldofpolygons.com

[35] Boston Dynamics: DI-Guy.
http://www.diguy.com

[36] Legion: Legion Studio a Legion 3D.
http://www.legion.com

[37] Massive Software: Massive Prime a Massive Jet.
http://www.massivesoftware.com

[38] Savannah Simulations AG: SimWalk a CityASim.
http://www.savannah-simulations.com

[39] SpirOps: SpirOps A.I., SpirOps Path a SpirOps Crowd.
http://www.spirops.com

Dodatek A

Uživatelská dokumentace

A.1 Požadavky na systém

K tomu, aby program MASS běžel správně, je zapotřebí následující konfigurace počítače:

- grafická karta podporující standard OpenGL ve verzi 1.4 nebo vyšší, nainstalovaný a funkční ovladač
- monitor s rozlišením alespoň 800x600
- dostatečné množství volné operační paměti[1]
- operační systém Microsoft Windows XP[2]

Dále je velmi vhodné, aby počítač obsahoval výkonný vícejádrový procesor. Jen tak je možno naplno využít potenciálu paralelního přístupu k simulaci. Ačkoliv výsledný grafický výstup je spíše jednodušší, pro zobrazení většího množství agentů je doporučena alespoň středně výkonná grafická karta, rozhodně ne pouze integrovaná.

A.2 Instalace a spuštění programu

Program není třeba nijak instalovat. Stačí zkopírovat obsah složky `/mass` na pevný disk, aby do adresáře `/mass/output` mohly být vypisovány textové záznamy o průběhu jeho běhu.

Spustitelné soubory se nachází ve složce `/mass/bin` a jsou dva. Pomocí `mass.exe` spustíme standardní verzi programu poskytující pokročilé zobrazení průběhu simulace. To uživateli umožní lépe pochopit, jaké postupy jsou při simulaci používány. Naproti

[1] Jako se základem je třeba počítat s více než 10MB. Dále množství použité paměti závisí na mnoha faktorech, jako je reprezentace virtuálního světa pomocí map (cca $1.5\text{MB}/1000m^2$), reprezentace jednotlivých agentů (méně než 1kB/jedinec), počet překážek, počet simulačních vláken apod.

[2] Program by měl bez problémů běžet i pod novějšími verzemi operačního systému Windows, avšak nelze to zaručit, neboť pod těmito novějšími verzemi nebyl testován.

tomu soubor `mass-fast.exe` spustí rychlejší verzi programu, ve které ovšem toto zobrazení budeme muset oželet. V této druhé verzi sice simulace chodců také probíhá zcela korektně, při vykreslování ale může občas docházet k chybám. Ty by ale neměly ohrozit stabilitu programu.

Obě verze se spouští z příkazové řádky se dvěma povinnými parametry. Prvním z nich je cesta k souboru obsahujícímu nastavení programu, druhým cesta k souboru definujícímu virtuální svět. Jejich formát včetně ukázek je popsán dále, příklady těchto souborů lze nalézt v adresáři `/mass/data`. Příkaz ke spuštění programu potom může vypadat např. takto: `mass.exe ../data/config1.xml ../data/map1.xml`. Pokud některý z parametrů chybí, případně nebyl-li nalezen některý ze souborů, pokusí se aplikace použít implicitně nastavené soubory.

A.3 Popis programu a jeho ovládání

Během inicializace program vypisuje do konzole podrobné informace o jejím průběhu. Jestliže při vytváření některého z modulů dojde k nějaké kritické chybě, je zpráva o tom taktéž vypsána do konzole a program se následně ukončí. Pokud ale proběhne vše v pořádku, je na konci inicializace vytvořeno okno pro potřeby grafického výstupu. Následuje vypsání obsahu nápovědy do konzole a spuštění samotné simulace.

Nyní by měl uživatel v okně s grafickým výstupem vidět celý virtuální svět v pohledu shora. V levém horním rohu obrazovky se nacházejí informace o průběhu simulace. V prvním řádku lze vidět dobu trvání simulačního kroku pro jednotlivé simulátory. Jedná se o průměrnou hodnotu za několik posledních period. Ve druhém řádku se zobrazuje celková doba, po kterou simulace běží. Konečně třetí řádek obsahuje informaci o průměrném počtu snímků zobrazovaných za sekundu za několik posledních period. Jestliže simulaci pozastavíme a po chvíli znovu spustíme stisknutím klávesy *Pause*, může být tento údaj nějakou dobu lehce zkreslený. Do levého dolního rohu se potom vypisují zprávy programu určené uživateli v reakci na jeho vstupy. Celkově může být zobrazeno až 8 zpráv, každá z nich po 3 sekundách automaticky mizí.

Ovládání programu je velice jednoduché a děje se prostřednictvím myši a klávesnice. Myší slouží k ovlání kamery. Je-li stisknuto levé tlačítko, lze pohybem myši rotovat kamerou okolo průsečíku pohledového paprsku s rovinou virtuálního světa. Citlivost je nastavena tak, že pohyb myši od jednoho okraje okna k druhému odpovídá rotaci 180°, a to v horizontálním i vertikálním směru. Je-li stisknuto pravé tlačítko, lze kamerou pohybovat do strany a nahoru/dolů (oba dva pohyby jsou myšleny ve směru lokálních souřadnicových os kamery). Otáčením kolečka myši lze potom scénu přibližovat a oddalovat (jedná se vlastně o pohyb kamery ve směru pohledového paprsku).

Klávesnicí lze ovládat jak kameru, tak i nastavení zobrazení. Popis funkce jednotlivých kláves je přehledně shrnut v tabulce A.1.

Stiskem klávesy m měníme způsob zobrazení mapy. Buďto se nezobrazuje žádná mapa (tento režim je nastaven implicitně), nebo si můžeme prohlédnout mapu ve formě kvadrantového stromu umožňující plánování cest, nebo mřížkovou mapu určenou pro podporu plánování cest a vnímání okolních překážek. V mřížkové mapě lze vidět buňky obsazené překážkami znázorněné světle zelenou barvou a hraniční buňky překážek tmavě

Klávesa	Funkce
Esc	ukončení programu
Pause	pozastavení/spuštění simulace
h	vypsání nápovědy do konzole
$\uparrow, \downarrow, \leftarrow, \rightarrow$	pohyb kamery
+/-	přiblížení/oddálení kamery
f	zapnout/vypnout zobrazení přes celou obrazovku
m	změnit způsob zobrazení mapy
b	zapnout/vypnout zobrazení regionů, portálů a bran
r	zobrazit/skrýt měřítko
o	změnit způsob zobrazení překážek
p	změnit způsob zobrazení agentů
v	zobrazit/skrýt vektor rychlosti agent;
1	zobrazit/skrýt anotaci cest
2	zobrazit/skrýt anotaci sousedních agentů
3	zobrazit/skrýt anotaci okolních překážek
4	zobrazit/skrýt anotaci anotaci steering behavior

Obrázek A.1: Klávesy pro ovládání programu

zelenou barvou.

Překážky lze zobrazovat třemi způsoby: v plné barvě, poloprůhledné (implicitní nastavení) nebo zcela neviditelné. Třetí možnost je tu zejména proto, aby bylo dobře vidět mapy.

Také u agentů existují tři možnosti jejich zobrazení. První možností, která je nastavena implicitně, je zobrazení jednoduchých modelů těl agentů. Tyto modely připomínají kuželky. Druhou možností je zobrazení agentů jako trojbokých hranolů, na kterých je pěkně vidět natočení agentů. Třetím způsobem je potom jednoduché dvourozměrné zobrazení. Dále si lze všimnout, že agenti se na obrazovce vyskytují ve dvou různých barvách. Ta je jim přidělována na základě jejich pohlaví — muži jsou vykreslováni modře, ženy červeně.

Klávesy *1-5* slouží ke zobrazení či skrytí tzv. anotací. Anotace jsou doprovodné informace o agentech, které umožňují uživateli lépe pochopit, jak simulace probíhá. Tyto anotace jsou dostupné pouze tehdy, byl-li program zkompilován s jejich podporou. Nevýhodou anotací totiž je, že mohou znatelně brzdit provádění simulace.

Je-li zapnuta anotace cest, lze vidět cesty, po kterých se jednotliví agenti pohybují. Cesty jsou zobrazeny jako množina význačných bodů spojených šipkami. Anotace sousedních agentů pomocí šipek naznačuje, o kterých svých sousedech daný agent ví a uvažuje je při plánování pohybu. Stejně funguje i anotace okolních překážek. Poslední anotace souvisí s reaktivním chováním agenta realizovaným řídícími silami a naznačuje, jaká síla na něj v daný okamžik působí.

A.4 Příklad konfiguračního souboru

Konfigurační soubor sloužící k nastavení parametů simulace a programu obecně může vypadat např. takto:

```
<?xml version="1.0" encoding="utf-8"?>
<mass>
   <simCount>2</simCount>
   <maxFPS>30</maxFPS>
   <entries>
      <entry gate="g1" mean="2.0" deviation="1.0"/>
   </entries>
   <goals>
      <randomWalk probability="0.2">
      <reachExit gate="g1" probability="0.3"/>
      <reachExit gate="g2" probability="0.5"/>
   </goals>
</mass>
```

Parametr `simCount` určuje počet vláken, která mají být použita k provádění simulace. Vždy je třeba alespoň jednoho simulačního vlákna, shora je jejich počet omezen číslem 16. Kromě simulačních vláken je nutné počítat ještě s jedním hlavním vláknem. Obecně se doporučuje, aby celkový počet vláken (či pouze počet simulačních vláken) odpovídal počtu jader procesoru. V tom případě by měl být výkon aplikace pro daný stroj optimální.

Při testování programu na nějakých počítačích docházelo k tomu, že zobrazování se pro určité počty simulačních vláken chvílemi zasekávalo. Důvodem tohoto sekání je, že plánovač v některých momentech odebere hlavnímu vláknu procesor a zhruba po dobu $0.1s$ nemůže tak být výstup zobrazován. Je to způsobeno zřejmě přílišným soutěžením jednotlivých vláken o přidělení procesoru, určitě se nejedná o synchronizační problém. Jestliže toto nastane, stačí pouze v konfiguračním souboru navolit jiný počet simulačních vláken a program by měl opět běžet plynule (je-li ovšem výkon počítače dostatečný).

Parametr `maxFPS` určuje maximální počet snímků výstupu vykreslovaných za jednu sekundu. Je zbytečné, aby hodnota parametru přesahovala 30, neboť veškerý pohyb se i tak zdá zcela plynulý. Důležité je, že takto nastavená hodnota je maximální, skutečný počet vykreslovaných snímků může být v závislosti na vytížení počítače podstatně nižší.

Uvnitř seznamu `entries` se nachází definice jednotlivých vstupů, kterými agenti vchází do virtuálního světa. Jeden záznam `entry` obsahuje v atributu `gate` identifikátor brány, která má být použita jako vstup. Je přitom nutné, aby identifikátor byl platnou branou virtuálního světa a ta aby byla definována jako vstupní. Čas příchodu mezi dvěma agenty je generován náhodně z normálního rozložení se střední hodnotou danou atributem `mean` a směrodatnou odchylkou `deviation`.

Seznam `goals` definuje globální cíle jednotlivých agentů. Každý z cílů musí mít pomocí atributu `probability` definovanou pravděpodobnost, že právě on bude agentovi přidělen. Součet pravděpodobností přitom nemusí být roven 1, všechny hodnoty jsou totiž normalizovány dle jejich celkového součtu. Agentům lze zvolit dva druhy cílů: náhodnou procházku `randomWalk` a cestu k nějakému z východů `reachExit`. Pro cíl

`reachExit` musí být dále definován platný identifikátor některé z bran sloužících jako východ.

A.5 Příklad definice virtuálního světa

Celý popis virtuálního světa musí být uzavřen uvnitř párové značky `world`. Všechny souřadnice a rozměry jsou udávány v metrech v rovině xy:

```
<?xml version="1.0" encoding="utf-8"?>
<world>
   <origin x="0" y="0"/>
   <size x="40" y="20"/>
```

Nejdříve je třeba definovat počáteční bod světových souřadnic (`origin`) a rozměry světa (`size`). Je tedy vidět, že celý svět musí mít tvar obdélníku, jehož strany jsou rovnoběžné s osami x a y.

Následuje popis rozdělení světa do jednotlivých regionů:

```
   <regionList>
      <region id="r1">
         <origin x="0" y="0"/>
         <size x="20" y="20"/>
      </region>
      <region id="r2">
         <origin x="20" y="0"/>
         <size x="20" y="20"/>
      </region>
   </regionList>
```

Také regiony mají tvar obdélníků, jejichž strany jsou zarovnány s osami x a y, a každý z nich musí být identifikovatelný pomocí unikátního názvu `id`. Důležité je, aby regiony pokrývaly celou plochu světa a aby se mezi sebou neprotínaly, pouze na sebe těsně doléhaly. Toto omezení je striktní, jsou povoleny pouze chyby v řádech milimetrů, které by mohly vzniknout v důsledku zaokrouhlování.

Dalším v pořadí je výčet všech portálů, jež označují místa, přes která lze přecházet mezi jednotlivými regiony:

```
   <portalList>
      <portal id="p1" firstRegion="r1" secondRegion="r2">
         <begin x="20" y="5"/>
         <end x="20" y="15"/>
      </portal>
   </portalList>
```

Každý portál musí být jednoznačně identifikovatelný podle svého názvu `id`. Aby bylo poznat, které regiony spojuje, obsahuje v atributech `firstRegion` a `secondRegion` jejich identifikátory. Pozice portálu se zadává pomocí jeho počátečního a koncového bodu. Aby bylo umístění platné, musí se portál nacházet na hranici mezi regiony a musí být zarovnán s některou z os x nebo y (to je dáno tím, že i regiony musí být zarovnané). V oblasti portálu se nesmí vyskytovat žádné překážky.

Následuje seznam bran (tj. míst sloužích agentům ke vstupu do virtuálního světa nebo výstupu z něj):

```
   <gateList>
      <gate id="g1" region="r1" type="in/out">
         <begin x="0" y="5"/>
         <end x="0" y="15"/>
      </gate>
      <gate id="g2" region="r2" type="out">
         <begin x="40" y="5"/>
         <end x="40" y="15"/>
      </gate>
   </gateList>
```

Pro brány platí podobná pravidla jako pro portály. Rozdíl je pouze v tom, že portál spojuje dva regiony a brána vede pouze do jednoho regionu, jehož identifikátor také obsahuje. Brána navíc ještě musí obsahovat informaci o svém typu (atribut `type`), tj. zda se jedná pouze o vchod (hodnota `in`), pouze o východ (`out`), nebo o vstupně-výstupní bránu (`in/out`). Je důležité, aby byla definována alespoň jedna vstupní či vstupně-výstupní brána, jinak by nebylo kudy do světa vejít.

Nakonec je třeba uvést v seznamu všechny požadované překážky:

```
   <obstacleList>
      <obstacle>
         <bound>
            <polygon>
               <vertex2d x="5" y="5"/>
               <vertex2d x="15" y="5"/>
               <vertex2d x="15" y="15"/>
               <vertex2d x="5" y="15"/>
            </polygon>
         </bound>
      </obstacle>
      <obstacle>
         <bound>
            <circle x="30" y="10" radius="5"/>
         </bound>
      </obstacle>
   </obstacleList>
</world>
```

Jediným parametrem, který v nynější verzi programu každá překážka (značka `obstacle`) obsahuje, je její obal, jehož definice leží uvnitř párové značky `bound`. V současnosti jsou implementovány dva různé obalové tvary. Prvním z nich je kružnice (`circle`) určená svým středem a poloměrem. Druhým potom mnohoúhelník (`polygon`), jež se zadává jako seznam vrcholů. Může být konvexní i nekonvexní. Je povoleno, aby obal překážky byl tvořen pouze jedním tvarem.

Dodatek B

Obsah CD

`/doc`
text bakalářské práce, programátorské a uživatelské dokumentace ve formátech PostScript, PDF a DVI

`/doc/doxygen`
automaticky generovaná dokumentace ve formátech HTML a RTF

`/doc/tex`
zdrojové soubory bakalářské práce a dokumentace ve formátu TEX včetně obrázků

`/mass/bat`
předpřipravené skripty ve formě dávkových souborů BAT pro snadné spouštění programu

`/mass/bin`
spustitelné soubory a dynamicky linkovaná knihovna SDL

`/mass/data`
ukázkové konfigurační soubory a mapy

`/mass/output`
sem se ukládají textové výstupy simulace

`/src`
zdrojové kódy programu MASS včetně souboru s projektem pro MS Visual Studio 2005, externí knihovny nutné pro správný překlad, ukázková data