Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Plzeň, 2006

Přemysl Zítka

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Hierarchické povrchové modely

Plzeň, 2006 Přemysl Zítka

Anotace

Hierarchické a adaptivní přístupy k řadě problémů jsou v poslední době v počítačové grafice často diskutovanými tématy. Zejména se objevují v souvislosti s reprezentací prostoru nebo v souvislosti s pojmem Level of Detail (LOD) při zobrazování dat s velkým množstvím detailů. Tato práce zkoumá možnosti využití hierarchie a adaptivity v problematice hledání cest.

Zabývá se přípravou a testováním 3D adaptivní mřížky pro algoritmus vyhledávání cesty v potenciálně neznámém a dynamickém prostředí. Cílem je poskytnout navenek grafovou strukturu bodů, která umožní snadné plánování cesty. Vnitřní funkčnost grafové struktury však zajišťuje dynamická adaptivní mřížka, přinášející významnou úsporu výpočetních prostředků.

Abstract

Hierarchical and adaptive approaches to a variety of problems are at present often discussed topics in computer graphics. They appear mainly in relation to space representation and in detailed model visualization, known as Level of Detail (LOD). This work focuses on possibilities of use of hierarchy and adaptivity in the area of path finding.

The work concerns with creating 3-dimensional adaptive grid used in path-finding in possibly unknown and dynamic space. The goal is to provide output graph structure of points, easy to use for path-finding algorithms. Under the graph structure, however, a dynamic adaptive grid provides more cost-effective approach to computer's resources.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím uvedených citovaných pramenů.

V Plzni dne 21. srpna 2006 Přemysl Zítka

Poděkování

Rád bych poděkoval zejména Bc. Petru Brožovi, se kterým jsme na projektu spolupracovali. Jeho výborná bakalářská práce mi umožnila mnou zpracovaný systém nejen reálně použít, ale i otestovat a demonstrovat. Stejně bych velice rád poděkoval i vedoucí diplomové práce paní Doc. Dr. Ing. Ivaně Kolingerové za odbornou pomoc a užitečné rady a podnětné připomínky při zpracování mé diplomové práce.

Nemenší dík náleží také všem těm, kteří se mnou během mnoha měsíců, ve kterých jsem diplomovou práci vytvářel, měli téměř bezmeznou trpělivost, mé rodině i přátelům za jejich podporu a pochopení v průběhu celého mého studia.

Obsah

Kapitola 1

Úvod

Počítačová grafika a vizualizace dat je v dnešní době významným oborem informatiky a s rostoucím výkonem a dostupností počítačů v různých oborech roste i potřeba zobrazovat mnoho rozličných modelů a vizualizovat stále větší množství informací. Počítačová grafika se tak stává nepostradatelným prostředníkem mezi člověkem a jeho počítačovými daty. Vývoj moderních metod pro počítačovou grafiku proto nezaostává. Ke v poslední době často zmiňovaným a také aplikovaným postupům patří mimo jiné adaptivní algoritmy ve všech svých podobách.

Cílem této diplomové práce je prozkoumat některé běžné adaptivní postupy a algoritmy, jejich možnosti a jejich aplikace. Původní téma se mělo týkat hierarchie pro povrchové modely, ovšem na žádost vedoucí jsem se zabýval objemovou hierarchií pro běžící projekt plánování cest, který navazuje na práci Dr. Gavrilové z University of Calgary v Kanadě. Dalším úkolem této práce bylo ověření možnosti užití za předchozím účelem vytvořené struktury pro jiný projekt, konkrétně interaktivní modifikaci povrchu nástrojem VR, v návaznosti na práci Dr. Beneše z Purdue University ve Spojených Státech.

Kapitola 2 této práce se věnuje nejprve objasnění pojmů adaptivita a hierarchie a dále pak způsobům jejich využití v rámci oboru počítačové grafiky. Následující kapitola 3 pak zmiňuje některé běžně používané adaptivní

metody.

Po teoretickém úvodu následuje v kapitole 4 představení úlohy hledání cest v potenciálně neznámém a dynamickém 3D prostředí. Účelem práce je prokázat, zda je využití adaptivního dělení třírozměrného prostoru pro takovou úlohu vhodné a zda může přinést významné urychlení či úsporu prostředků. Následně je popsána úloha modifikace povrchu, která má ověřit použitelnost navrženého systému i v jiných souvislostech, než pro které byl původně vytvořen.

Kapitola 5 popisuje dílčí problémy, jejich možná řešení a výsledná řešení. V kapitole 6 následuje dekompozice a popis návrhu jednotlivých rozhraní a tříd a kapitola 7 uvádí detaily o implementaci, použitých prostředcích a požadavcích pro překlad a spuštění aplikace.

Kapitola 8 obsahuje informace o navržených a provedených testech, o podmínkách testování a o měření jednotlivých aspektů ovlivňujících výkon výsledné aplikace. Kapitola 9 se zabývá změnami a rozšířeními provedenými pro úlohu modifikace povrchu a testy na této úloze provedenými.

Závěrečné zhodnocení v kapitole 10 pak shrnuje klady i zápory zvoleného řešení a předkládá možné úpravy a dodatečná rozšíření pro budoucí vývoj.

Kapitola 2

Adaptivita a hierarchie

2.1 Adaptivita

Adaptivita je přístup přinášející rozumný kompromis mezi potřebou počítačové reprezentace zachytit požadované množství detailů (typický požadavek fotorealističnosti scény a modelů v ní zobrazených) na jedné straně a rychlostí a paměťovými nároky na straně druhé. S rostoucími nároky na detailnost totiž také narůstají nároky na paměťový prostor i výpočetní sílu. Nemusí se přitom nutně jednat jenom o výpočetní sílu potřebnou k zobrazování výsledné scény. Je třeba si uvědomit, že se všemi objekty se může podle typu úlohy provádět mnohem více operací, ať už grafické transformace nebo třeba výpočty různých fyzikálních veličin či aplikace vlivů působení modelovaného prostředí. Rostoucí požadavky na realističnost si však používání dat s velkým množstvím detailů vynucují.

V naprosté většině případů reprezentujeme taková data trojúhelníkovými sítěmi. Sítě získáváme převážně tesalací matematicky definovaných ploch modelu nebo triangulací bodů získaných skenováním fyzických objektů. V obou případech vznikají velmi detailní sítě a jejich ukládání na disk, vykreslování či přenos po komunikační lince jsou natolik náročné, že je celé problematice věnována významná pozornost.

[1] uvádí několik základních přístupů a směrů vývoje k řešení těchto problémů:

Mesh simplification (zjednodušení, decimace sítě): Sítě získané výše popsanými způsoby nejsou v převážné většině případů nijak optimalizovány pro efektivní vykreslování a mohou být aproximovány pro lidské oko neodlišitelnou sítí obsahující podstatně menší počet trojúhelníků.

Level-of-detail (LOD) approximation (úrovně detailů aproximující původní model): Běžným postupem bývá definování několik verzí modelu s různou úrovní detailů (s různým počtem trojúhelníků tvořících síť modelu). Detailní složitá síť je použita, pokud se objekt nachází v blízkosti pozorovatele, jednodušší sítě pak s rostoucí vzdáleností. Jelikož přepínání mezi jednotlivými úrovněmi detailů může vést k pozorovatelnému problikávání, rozumným přístupem je konstruovat plynulé přechody, *geomorfy*, mezi sítěmi různých rozlišení.

Progressive transmission (postupný přenos): Při přenosu trojúhelníkové sítě komunikační linkou očekáváme, že budeme mít velmi brzy po začátku přenosu k dispozici alespoň málo detailní model a společně s přicházejícím tokem dat bude následně docházet k postupnému zkvalitňování zobrazené sítě. Možný přístup je například posílat postupně jednotlivé úrovně detailů – od nejhorší až po nejkvalitnější.

Mesh compression (komprese sítě): Komprese je jiným přístupem k řešení problému minimalizace prostoru potřebného pro uložení trojúhelníkové sítě. Na rozdíl od decimace se nezaměřuje na zmenšení počtu trojúhelníků, ale na optimalizaci způsobu jejich ukládání.

Selective refinement (selektivní zjemňování): Každá z trojúhelníkových sítí jednotlivých LOD reprezentací představuje model v určité jednotné úrovni detailů. Často je výhodné adaptovat úroveň zjemnění odlišně v jednotlivých oblastech téhož modelu podle toho, kde jsou a nejsou detaily třeba.

Obrázek 2.1: Skalní útvar Uluru.

Adaptivitu lze obecně chápat jako přizpůsobování dat výkonu počítače. V oblasti počítačové grafiky má toto přizpůsobování svůj geometrický význam. V takovém pojetí pak využíváme toho, že často není třeba vzorkovat prostor či model na všech místech stejně, což lze výhodně využít zejména ve třech mírně odlišných situacích, které se pokusím vysvětlit jednoduchým příkladem:

> V Austrálii nalezneme poměrně známý a jedinečný skalní útvar jménem *Uluru*. Jedná se o horu, která znenadání vystupuje ze země uprostřed rozlehlé pouštní roviny (Obrázek 2.1). Při úloze zmapovat okolní poušť rychle dojdeme k závěru, že zdaleka není vhodné kvůli jedné skále vzorkovat výšková měření řekněme například s krokem jednoho metru po celé rozsáhlé ploše pouště, tvořící jednu velkou rovinu. To je třeba v případě jediné výjimky a tou je právě náš skalní útvar. Použijeme tedy adaptivní řešení – výšková měření vlastní pouště můžeme vzorkovat například po deseti kilometrech, bezprostřední okolí skalního útvaru pak například po deseti metrech a vlastní skálu po půl metru. Adaptivitu tedy aplikujeme tak, že na rozdíl od pravidelné mřížky mapy provádíme podrobné měření *pouze v oblastech, kde detaily fakticky existují*.
>
> Do druhé situace se snadno dostaneme jako autoři aplikace, která bude umožňovat zmapovaným terénem procházet. Již víme, jakým způsobem mohla být mapa navzorkována a uložena. Při vizualizaci skalního útvaru v naší aplikaci jistě uví-

táme možnost si jej podrobně prohlédnout. Ale má se v paměti při zobrazování v plných detailech udržovat i od pozorovatele odvrácená strana útvaru, nebo útvar celý v okamžiku, kdy je od něj pozorovatel odvrácen nebo výrazně vzdálen? Samozřejmě je výhodnější využít adaptivitu i zde a používat plné detaily ne ve všech místech, ale *kde nejen fakticky existují, ale tam, kde jsou i prakticky potřeba*

Třetí situace je pak způsob, jakým v naší aplikaci zobrazíme okolí skalního útvaru. Během vizualizace s výhodou adaptivitu původního měření využijeme a zobrazujeme poušť jen několika málo trojúhelníky, jako rovinu. Přesto může vzniknout potřeba v okolí pozorovatele uměle trojúhelníky rozdělit a umožnit například drobné zvlnění pouště nebo otisky stop pozorovatele pro lepší vizuální dojem z aplikace. Adaptivitu tedy nasazujeme v místech, kde *detaily fakticky neexistují a sami je uměle dotváříme*.

Adaptivita umožňuje věnovat dostatek prostředků na oblasti, kde jsou detaily požadovány, naopak prostředky ušetřit v oblastech, na které takový požadavek v daný okamžik nemáme a případně využít volné výpočetní síly k dotvoření detailů v místech, kde v datech ani reálně nejsou. Požadavek na detailnost může být dán například směrem, kterým se ve virtuálním světě dívá kamera, a vzdáleností, do které mají detaily smysl. Udržovat v paměti modely v takových detailech, že je zobrazovací zařízení za daných podmínek ani nedokáže vykreslit, totiž často smysl mít nemusí. Stejně tak nemusí mít smysl ani s takto zbytečně detailními objekty provádět všechny další případné operace. Daleko výhodnější je upravit detailnost podle aktuálních podmínek a požadavků a teprve s výsledkem pak dále pracovat. Kritéria pro rozhodování o potřebě detailů mohou být na základě zadání úlohy či typu problému poměrně rozličná. Velmi často se jedná o kritéria související se zobrazováním, zejména jde o směr pohledu a vzdálenost objektu od pozorovatele, ovšem pohled na celý problém může být i komplexnější. Do detailů se problematice adaptivity v závislosti na vizuálních kritériích věnuje [2].

Adaptivní algoritmy se využívají zejména ve dvou oblastech počítačové grafiky, a to v oblasti povrchových modelů a pro adaptivní dělení prostoru. Stejně tak se používají dva rozdílné přístupy. Jedním je v okamžiku potřeby pominout detaily v částech či oblastech, kde nejsou třeba. Adaptováním pak rozumíme jeho jakési „zhrubování " a pomíjení detailů. Druhý postup je v podstatě opačný. Oblast zájmu je dělena na menší buňky sítě. Adaptováním tedy naopak „zjemňujeme" a snažíme se detaily v oblasti zájmu zpřístupnit.

2.2 Hierarchie

Zatímco pojem adaptivita v počítačové grafice vyjadřuje výše popsanou schopnost algoritmu přizpůsobit se datům, pojem hierarchie popisuje způsob přístupu k této schopnosti adaptivity. Pokud si představíme například zobrazování modelu, pak při adaptivním přístupu je možno model zobrazit jak v úplném detailním provedení (za odpovídající spotřeby výpočetních prostředků), tak i v jeho nejhrubších rysech (naopak významná úspora paměti i výkonu). Lze ovšem zobrazit i mnoho různých mezikroků mezi těmito dvěma hraničními stavy. Každý z takové řady stavů pak nazveme jinou úrovní detailů - zaveden je anglický pojem *Level of Detail (LOD)*. Tyto úrovně lze snadno (například podle počtu zobrazených trojúhelníků) seřadit od nejdetailnějšího po nejméně detailní, čímž definujeme jejich hierarchii. Každá úroveň přináší oproti svojí předchůdkyni více detailů a objekt se během průchodu úrovněmi stává postupně komplexnějším. V rozhodovacím algoritmu se pak podle daného kritéria zvolí, kterou úroveň použít, případně se zvolí vhodná kombinace několika úrovní. Tedy například že požadavky na nejbližší modely ve scéně budou zpracovávány v plných detailech, objekty dále od kamery jen v polovičních a objekty daleko od kamery jen v hrubých rysech. Nebo lze kombinovat několik různých úrovní na jediném modelu. V nejmenší vzdálenosti od kamery budou použity plné detaily, v oblastech od kamery vzdálenějších detaily poloviční a v místech kameře skrytých (například odvrácená strana

objektu) pak jen v hrubých rysech.

Některé metody, například použitá a později v kapitole 3.5 popsaná metoda dělení prostoru pomocí octtree, kde jednotlivé úrovně detailů vlastně odpovídají jednotlivým úrovním stromové struktury, vedou na hierarchii přímo.

Obecně existují dva přístupy, jak s adaptivitou a hierarchií pracovat. Nazvěme je pro naše potřeby offline a online přístup.

U offline přístupu se jednotlivé úrovně detailů připraví v rámci předzpracování předem, aplikaci takového postupu uvádí například [3]. Algoritmy na decimaci, které se o snížení počtu trojúhelníků, a tedy i snížení úrovně detailů starají, mohou být komplikované a časově náročné, zejména pokud je kladen důraz na maximální vizuální kvalitu a věrnost i méně detailních modelů. V naprosté většině případů je proto výhodné provést decimaci předem a za chodu již jen používat připravené modely s různými úrovněmi detailů. V jednom „objektu“ pak nalezneme např. hned několik jeho modelů, se kterými pak následně algoritmy pracují.

[1] nabízí alternativní řešení, kdy je uložena nejhrubší síť a série inkrementálních operací postupně tuto síť zkvalitňujících. Zřejmé výhody jsou zejména při přenosu modelu pomalou linkou, kdy v úvodu se převede a je okamžitě k dispozici alespoň hrubý model a ten je následně přicházejícím proudem inkrementálních změn zkvalitňován. Zároveň je snadné definovat tzv. geomorfy (geomorph), což jsou plynulé přechody mezi jednotlivými definovanými úrovněmi detailů. Geomorfem se stane definovaná sekvence inkrementálních vylepšení sítě, která vede od jedné úrovně detailů ke druhé. Využitím geomorfů se podstatně zvýší vizuální kvalita LOD řešení, neboť se zabrání skokovým změnám a problikávání modelu, na které je lidské oko výrazně citlivé.

Jako online přístup popisuji variantu, kdy se adaptivita aplikuje až za vlastního běhu programu. Z mnoha důvodů může být nežádoucí nebo přímo nemožné si potřebné úrovně připravit v předzpracování. V takovém případě je pak třeba volit rozumný kompromis mezi kvalitou výsledku

a rychlostí celé adaptace zohledňující požadavky konkrétní úlohy.

Obecně samozřejmě platí, že adaptivita by za cenu komplikovanější implementace měla přinést buďto nárůst kvality detailů při zachování doby operací, nebo časovou úsporu při zachování kvality detailů. Případně rozumný kompromis mezi kvalitou a rychlostí. Případ, kdy použití adaptivních algoritmů povede ke zpomalení při zhoršení kvality detailů, nebo i jejího zachování, nás tak jasně informuje o problému. Buďto byl adaptivní algoritmus špatně zvolen či naimplementován, nebo se pro danou úlohu adaptivní přístup nemusí vůbec hodit. To je vždy třeba zvážit.

Kapitola 3

Používaná dělení prostoru

3.1 Mřížky

Dělení prostoru pravidelnou mřížkou je zřejmě nejjednodušší z možných způsobů. Kartézská mřížka (krok mřížky je ve všech směrech stejný, buňky tvoří krychle) s sebou přináší významné výhody – poloha vrcholu mřížky je dána přímo jeho indexy a stejně tak i sousedy ve všech směrech lze určit inkrementací či dekrementací patřičného indexu. Dále může být mřížka pravidelná (v každém směru je velikost kroku definována zvlášť, buňky tvoří obecně kvádry), pravoúhlá (proměnlivá velikost kroku v jednotlivých osách), či strukturovaná (buňky mají tvar zdeformovaných kvádrů). Všechny tyto varianty pracují s topologicky uspořádanými vrcholy.

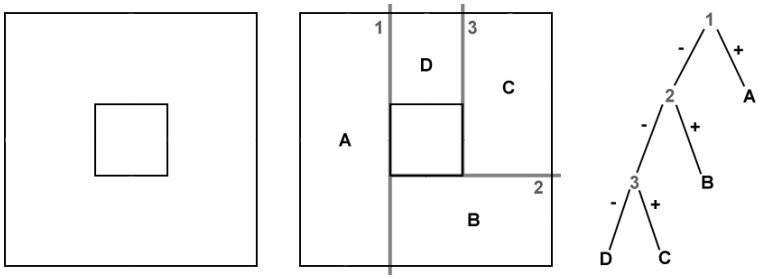
Další možností je mřížka nestrukturovaná, která pracuje s neuspořádanými vrcholy a topologii je tak nutno udržovat ve zvláštním poli.

Mřížky blokově strukturované nebo hybridní jsou pak kombinací ostatních v jednom objemu dat.

Detailnější popis a příklady jednotlivých mřížek lze nalézt například v [4].

3.2 BSP Tree

BSP (Binary Space Partitioning) strom reprezentuje rekurzivní hierarchické dělení n-rozměrného prostoru. Uzel stromu obsahuje informaci o dělící nadrovině, jeden syn odpovídá zápornému a druhý kladnému poloprostoru. Způsob dělení prostoru a výstavby odpovídajícího stromu naznačuje obrázek 3.1. Vlevo vidíme jednoduchou 2D scénu, uprostřed jedno z možných dělení a vpravo jemu odpovídající strom.



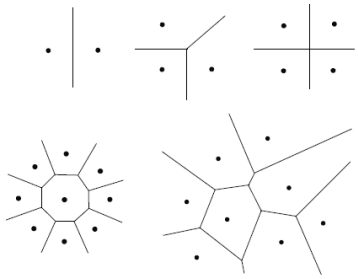
Obrázek 3.1: Příklad dělení scény pomocí BSP a jemu odpovídající stromová struktura.

Hierarchické dělení prostoru pomocí BSP stromů se využívá například při vykreslování trojrozměrné scény (určení viditelnosti, odstranění skrytých ploch), při detekcích kolizí nebo pro urychlování raytracingu. Detailní informace o BSP stromech lze nalézt například na [7].

3.3 Voronoiovy diagramy

Voronoiovy diagramy představují dělení prostoru podle v něm definované množiny vrcholů. Kolem každého vrcholu je definována buňka prostoru tak, že obsahuje všechny body, pro něž je daný vrchol nejbližším z celé množiny. Příklady dvourozměrných Voronoiových diagramů uvádí obrázek 3.2

Voronoiovy diagramy se používají například při triangulacích (přímkovým duálem Voronoiova diagramu je Delaunayova triangulace maxima-

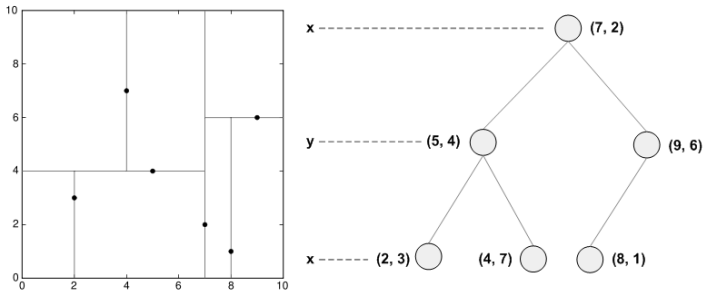


Obrázek 3.2: Příklady Voronoiových diagramů ve 2D.

lizující nejmenší úhel ve vznikajících polygonech), určování nejbližších bodů a sousedů či plánování cest. Více lze nalézt například na [8].

3.4 kD Tree

Jedná se o vícerozměrný binární strom pro organizaci k-rozměrného prostoru podle množiny vrcholů. Dělení provádíme nadrovinou kolmou k jedné z os, vedenou „prostředním“ vrcholem množiny. Vrchol, kterým dělící nadrovinou povedeme, vybereme například jako medián z bodů uspořádaných podle dané osy. Vzniklé podprostory opět dělíme nadrovinami kolmými k další z os a s postupujícím dělením osy pravidelně střídáme. Postup je naznačen na obrázku 3.3.



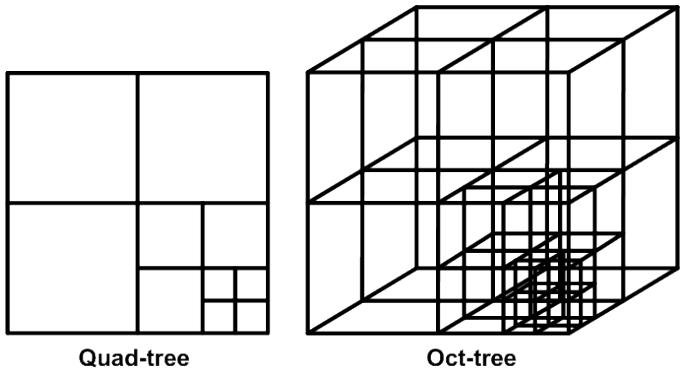
Obrázek 3.3: Postup vytváření struktury kD Tree..

Více o kD stromech lze nalézt na [9].

3.5 Quadtree a Octtree

Jelikož následná práce vychází právě ze struktury octtree, věnujme jejímu popisu větší pozornost.

Quadtree a octtree jsou hierarchické stromové struktury běžně používané pro adaptivní dělení prostoru. V obou případech se jedná o využití stejného principu, pouze quadtree je dělení 2D prostoru a octtree pak 3D prostoru. Princip dělení prostoru v obou případech je patrný na obrázku 3.4. Vlastní adaptivita je zajištěna pomocí dvou protichůdných operací – *split* (rozdělení) a *merge* (sloučení). Jejich vysvětlení uvedeme na příkladě quadtree.



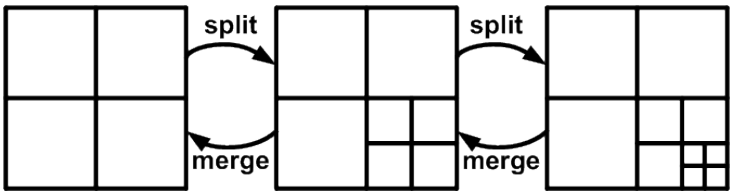
Obrázek 3.4: Struktury quadtree a octtree.

Vymezenou část roviny ohraničíme do čtverce a prohlásíme za základní buňku quadtree úrovně 0. Rozdělení této buňky na kvadranty (operace *split*) provedeme tak, že rozdělíme čtverec vodorovným a svislým řezem na stejně velké čtvrtiny. Získáme tak 4 (odtud *quad*) dceřiné buňky úrovně 1, čtverce o poloviční délce strany než měla buňka rodičovská. Libovolnou ze vzniklých buněk můžeme dále stejným postupem rozdělit na další 4 buňky úrovně 2 a pokračovat lze rekurzivně tak dlouho, dokud je třeba.

Naznačeným postupem pak můžeme získat adaptivní rozdělení prostoru podle toho, ve které oblasti a do jaké hloubky jsme při dělení buněk postupovali, což zvolíme podle potřeb konkrétní úlohy.

Pokud není získané dělení nadále třeba, je možno postupovat obdobným způsobem, pouze pozpátku. Vždy 4 buňky nižší úrovně se spojí v jednu buňku úrovně vyšší (operace *merge*) a tímto spojením zaniknou přebytečné vrcholy i buňky nižší úrovně samotné. Jejich místo zaujímá buňka sloučená.

Pro jasnou představu jsou obě operace *split* a merge zachyceny na obrázku 3.5.



Obrázek 3.5: Operace *split* a *merge* nad strukturou quadtree.

V případě octtree probíhá vše naprosto stejně, jenom buňka není čtverec, ale krychle, nové vrcholy při dělení vznikají ve středech hran, stěn a vlastním středu krychle a rozdělením rodičovské buňky získáme rovnou 8 buněk dceřiných. Odtud tedy ono *oct* v názvu.

Jak uvádí [4], s výhodou se popsané adaptivní struktury využívají nejen jako primární reprezentace objektů, ale často i jak pomocná datová struktura doplňující jiný popis objektů, zejména pro potřeby urychlení mnoha algoritmů. Takový bude i náš případ plánování cest.

Kapitola 4

Stanovení úkolů práce

4.1 Adaptivní datová struktura pro plánování cest v dynamickém prostředí

4.1.1 Původní úloha plánování cest

Původní kanadská práce [6] se zabývá, problémem autonomních agentů hledajících cestu v potenciálně neznámém dynamickém 2D prostředí. V prostoru se pohybují hrozby jakožto zdroje nebezpečí a agenti plánují svoje trasy tak, aby obešli překážky a zároveň se všem hrozbám vyhnuli, nebo alespoň souhrn nebezpečí na svojí trase minimalizovali. Vlastní chování agentů a algoritmy hledání cesty nejsou tématem této diplomové práce, tyto problémy řeší bakalářská práce [5]. Důležitá je ovšem reprezentace překážek a samotného prostoru pro celou úlohu hledání cest vymezeného. Zatímco překážky jsou určeny pravidelnou mřížkou (konkrétně jsou definovány bitmapou), samotný prostor je reprezentován mřížkou adaptivní, vybudovanou nad strukturou quadtree. Autor pro ni používá pojem *ASM (Adaptive Spatial Memory)*.

ASM slouží jako centrála, do které agenti zasílají získané informace o původně neznámém prostoru a zároveň pak následně na základě sloučení

těchto jednotlivých údajů získávají navigační body pro plánování svojí trasy. Mřížka je v základním tvaru sice poměrně hrubá, ovšem adaptuje se podle mapy překážek, kterou agenti postupně skládají dohromady. V průběhu vlastního hledání cest se pak adaptivní mřížka přizpůsobuje všem změnám dynamického prostředí, zejména pohybu, vznikání a zanikání případných překážek nebo hrozeb.

Vlastní hledání cest je na adaptivní mřížce závislé, neboť agenti plánují cesty mezi jednotlivými uzly mřížky. Díky adaptaci již mají v okolí překážek a hrozeb připraveno detailnější rozdělení procházeného prostoru (dostatek vrcholů grafu v dané zajímavé oblasti) a mohou naplánovat cestu odpovídajícím způsobem, aby se nebezpečným nebo neprůchodným oblastem vyhnuli.

Implementace a demonstrace těchto principů byla připravena v jazyce C++ a o zobrazování se stará knihovna OpenGL.

4.1.2 Požadovaná rozšíření

Naším úkolem bylo ukázat, zda je možné úlohu prostým způsobem rozšířit do 3D, a pokud ne, pokusit se identifikovat problematická místa a navrhnout vlastní úpravy tak, aby výsledné řešení funkční a použitelné být mohlo.

Jako zásadní se z tohoto hlediska ukázaly být právě změny v datových strukturách. Zatímco v originále používané 2D struktury plně úloze dostačují a zároveň nepředstavují zásadní výpočetní nebo paměťové problémy, jejich 3D varianty jsou na tom o poznání hůře. S nárůstem o jednu dimenzi se veškerá paměťová složitost navyšuje na třetí mocninu a je třeba počítat s obdobným nárůstem počtu operací a tedy i trvání celého výpočtu. Možným řešením je významně omezit rozměry či rozlišení map, nebo mapy reprezentovat a zpracovávat úplně jiným způsobem.

Obdobně u adaptivní mřížky narůstá při přechodu od quadtree k octtree stejným způsobem složitost paměťová a významně i složitost výpočetní.

Mřížka totiž musí kvůli nad ní pracujícím vyhledávacím algoritmům poskytovat informace o sousednosti vrcholů. S rozšířením o jeden rozměr počet sousedů vrcholů výrazně narůstá a přibývá i jednotlivých možností jejich pozic a případů jejich přítomnosti. Udržování těchto informací v konzistentním stavu je pak podstatně komplikovanější

4.1.3 Výsledné rozdělení a řešení úlohy

Vzhledem k velkému rozsahu jsme se s vedoucí práce rozhodli úlohu rozdělit mezi dva řešitele, jak již bylo výše naznačeno.

Bakalářská práce [5] se zabývala problémy map překážek a hrozeb, vlastním hledáním cesty a přípravou demonstrační aplikace. Mapy zůstaly v podstatě 3D bitmapami, které jsou zejména z paměťových a výpočetních důvodů používány pouze v malých rozměrech. Hledání cest probíhá, jak již bylo řečeno, nad vrcholy adaptivní mřížky.

Diplomová práce, kterou držíte v rukou, pak, vzhledem ke svému tématu hierarchických modelů, zahrnuje analýzu, návrh a práci na implementaci hierarchického modelu prostoru úlohy. Ten je reprezentován již zmiňovanou hybridní adaptivní mřížkou založenou částečně na stromové struktuře octtree (viz kapitola 3.5). Hybridní proto, že jde z jedné strany o adaptivní mřížku a ze strany druhé o grafovou strukturu poskytující v každém okamžiku přímé sousedy jednotlivých vrcholů.

Společnou prací pak byl úvodní návrh aplikace, komunikačního rozhraní mezi oběma částmi a také po implementaci následující ladění, testování, měření výsledků a dodatečné úpravy implementace.

Struktura celé aplikace a konkrétní funkčnost vyhledávání cest není předmětem této diplomové práce a proto v přílohách B a C uvádím na několik konferencí nabídnuté články, které celou aplikaci popisují. V dalším textu této diplomové práce se nadále budeme věnovat zejména používané adaptivní datové struktuře a rozhraní, přes které je přístupná.

Obecně panoval předpoklad, že naše práce ve výsledku povede ke zjištění zásadních, ať již paměťových nebo výpočetních, komplikací takové povahy, že se ukáže nemožné či nepraktické prosté rozšíření provést. Zejména jsme očekávali nezvladatelný nárůst paměťových i výpočetních nároků rozšířených datových struktur. Stejně tak i zvolené programové prostředí .NET s sebou přinášelo riziko zpomalení. Přesto bylo naším úkolem konkrétní problémy a místa jasně identifikovat a navrhnout možnosti řešení či zmírnění komplikací, nebo jejich obejití použitím jiného přístupu k řešení daného problému.

4.2 Úloha modifikace povrchu

Mým samostatným úkolem bylo pokusit se adaptivní strukturu připravenou pro výše uvedenou úlohu využít pro jiný účel a ukázat tak její univerzálnost. Zvoleným problémem byl pokus otiskovat do materiálu reprezentovaného adaptivní mřížkou jiný objekt. Výsledkem by měla být na správných místech adaptovaná struktura s vyznačeným povrchem otiskovaného objektu. Kromě ověření možnosti jiné aplikace jsem se zaměřil zejména na další zkoumání rychlosti, jakou je struktura schopná se adaptovat.

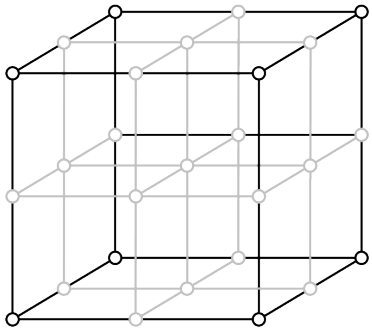
Kapitola 5

Adaptivní mřížka

5.1 Role adaptivní mřížky při hledání cesty

Adaptivní mřížka hraje v úloze roli prostředníka mezi bitovými mapami reprezentujícími překážky a hrozby a vlastním algoritmem pro hledání cest v prostředí definovaném těmito mapami. Zajišťuje převod bitmapových informací na grafovou datovou strukturu, se kterou umějí vyhledávací algoritmy dále pracovat. Vrcholy mřížky jsou dostupné jako uzly grafu a informace o sousednosti je použita ke spojování těchto uzlů cestami.

Reprezentovaný prostor je dynamický, a tak se s postupem času mění bitové mapy. Mřížka na tyto změny neustále reaguje a adaptuje se podle aktuální situace. Jelikož by změna měla probíhat co nejblíže reálnému času, kladl jsem při návrhu a implementaci adaptivní mřížky důraz zejména na její rychlost, paměťová složitost pak byla druhým, přesto velmi významným argumentem. Výsledné řešení je, myslím si, nakonec rozumným kompromisem mezi těmito dvěma hlavními neaplikačními požadavky.



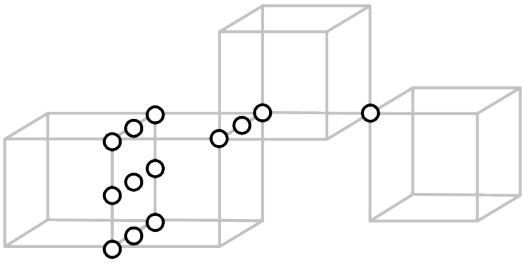
Obrázek 5.1: Vlastní a nevlastní vrcholy buňky.

5.2 Vrcholy a jejich sousednost

Zatímco u pravidelně strukturovaných mřížek je znalost sousednosti vrcholů samozřejmá (sousedy lze snadno získat kombinací inkrementací a dekrementací jednotlivých indexů určujících dotazovaný vrchol mřížky), použitím jiné než pravidelné struktury tuto výhodu ztrácíme. Znalost sousednosti vrcholů je i přesto základním aplikačním požadavkem kladeným na adaptivní mřížku. Bylo tedy nutné používanou strukturu octtree rozšířit a upravit tak, aby tento požadavek splňovala.

Struktura octtree zajišťuje hierarchické dělení prostoru na menší buňky. Jednotlivými vrcholy takto vznikajících buněk se ale nijak nezabývá. V prvním kroku tedy bylo nutné přidat k octtree vrcholy a operace s nimi. Oproti 2D řešení quadtree má buňka octtree místo 4 vlastních vrcholů 8. Kromě vlastních vrcholů definuji buňce ještě vrcholy, které nazývám nevlastními. V základním tvaru se jedná o neobsazené pozice, připravené pro nově vzniklé vrcholy přímých potomků buňky. Na obrázku 5.1 jsou vlastní vrcholy vyznačený černě a vrcholy nevlastní šedě. Jak je patrné, spolu s nevlastními vrcholy jich každá buňka obsahuje celkem 27. Operace split a merge jsou proto doplněny o vytváření a rušení vrcholů a jejich správné obsazování do odpovídajících pozic.

Sousedící buňky nebo buňky do sebe zanořené (rodič a jeho potomci) se navzájem dotýkají a proto mají několik totožných vrcholů. Sousedící buňky mohou sdílet společnou stěnu (4 vlastní a 5 nevlastních vrcholů), hranu (2

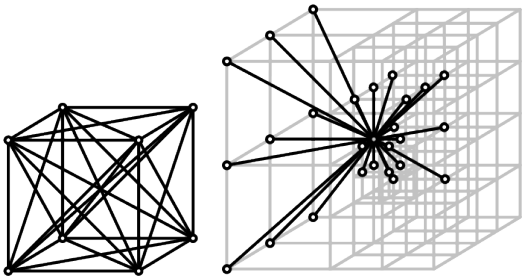


Obrázek 5.2: Možnosti sdílení vrcholů mezi sousedícími buňkami.

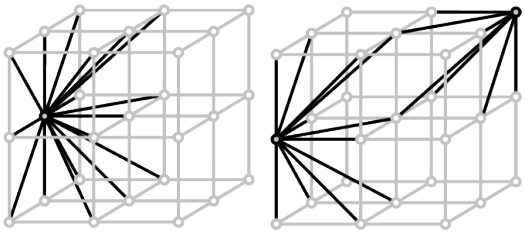
vlastní a 1 nevlastní vrchol), nebo 1 rohový vrchol, jak ukazuje obrázek 5.2. Vzhledem k předpokládanému velkému množství buněk v síti a nárokům na paměť není žádoucí, aby se v celkové struktuře takové vrcholy udržovaly duplicitně. Před jejich vytvářením proto probíhá kontrola ověřující, zda již na dané pozici struktury existuje použitelný vrchol a naopak při rušení se kontroluje, zda daný vrchol nepoužívají i jiné než rušené buňky. Taková kontrola, prováděná sice efektivně, ale prováděná při vytváření a rušení každé buňky, má za následek zpomalení adaptace sítě.

Druhým krokem rozšíření struktury octtree je přidání možnosti snadného a hlavně rychlého určování sousedů. Po vzoru 2D řešení sousedí v rámci jedné buňky vlastní vrcholy každý s každým, což znamená 28 vzájemných vazeb. Nevlastní vrcholy jsou připravené pozice pro dceřiné buňky na nižší úrovni a do sousednosti v rámci buňky tak nezasahují. Kromě okrajových případů je každý vrchol mřížky obklopen osmi buňkami a v každé z nich má svoje sousedy, kterých může být dohromady až 26. Buňky obklopující vrchol ovšem nemusejí být na stejné úrovni stromu. Sousednost v rámci buňky a příklad všech sousedů v okolí bodu ukazuje obrázek 5.3. Zmíněné okrajové případy, jedná se o vrcholy na stěně či hraně mřížky nebo o rohový bod, jsou znázorněný na obrázku 5.4.

Z uvedených ukázek je patrné je patrné, že sousednost vrcholů uvnitř adaptované mřížky je poměrně komplikovaná záležitost. Vyhledávání sousedů bez dodatečné informace uložené ve struktuře by tak bylo příliš složité a tedy i časově náročné. Vzhledem k tomu, že zjišťování sousedů velkého množství vrcholů provádí aplikace při určování cesty v grafu po každé adaptaci sítě, je požadavek na rychlost zásadní. Ukládat do kaž-



Obrázek 5.3: Vzájemné vazby uvnitř buňky a příklad sousedů v různých úrovních okolo jednoho bodu.



Obrázek 5.4: Okrajové případy počtu a pozic sousedů vrcholu.;

dého vrcholu 26 odkazů na sousedy by sice umožnilo jejich okamžité určení, ovšem paměťové nároky struktury by tím povážlivě narostly. Stejně tak i modifikace odkazů na sousedy u všech vrcholů během operací split a merge by adaptaci sítě významně zpomalovala.

Zvolené řešení představuje kompromis mezi oběma protichůdnými požadavky. Vrchol neobsahuje odkazy na svých 26 sousedů, ale na 8 buněk, které jej obklopují. Jedná se o buňky nejvyšší úrovně, která vrchol používá. Bod na této úrovni vznikl a dokud existuje, existují i obklopující buňky. Díky tomu se odkazy během adaptace sítě nemusí upravovat a zůstávají platné po celou dobu existence vrcholu. Pro každou z obklopujících buněk je zkoumaný bod vrcholem vlastním a leží tedy v jednom z rohů buňky. Bezprostřední sousedé se pak naleznou již poměrně snadno. Stačí určit nejhlouběji zanořenou podbuňku v daném rohu a její vlastní vrcholy jsou hledanými sousedy.

Popsaným způsobem je dosaženo více než dvoutřetinové úspory paměti potřebné k uchování sousednosti a zároveň je vlastní adaptace sítě zbavena

jakéhokoliv zpomalení z hlediska přepočítávání odkazů. Ani zaznamenání obklopujících buněk během vytváření vrcholu není časově náročné a je přijatelnou cenou za zisk v podobě paměťové úspory a rychlého hledání sousedů.

Kapitola 6

Objektově orientovaná analýza

Již ze zadání jsme měli představu o rozdělení úkolů, tedy že návrh a implementace adaptivních struktur a s nimi pracujících algoritmů budou součástí této diplomové práce a příprava vyhledávání cest součástí práce bakalářské. Celou analýzu jsme tomu podřídili a navrhli tak dva samostatné celky, komunikující co nejjednodušším rozhraním.

6.1 Návrh rozhraní

Rozhraní mezi našimi součástmi tvoří dva jednoduché interfacy. Prakticky jsou zestručněny na základní požadavek, že vyhledávací část požaduje ke svojí práci graf navigačních uzlů a datová struktura, třebaže z principu negrafová, jej musí poskytnout. Interfacy jsou navrženy tak, že součásti na obou jejich stranách lze snadno nahradit jinými, z hlediska daného interfacu ekvivalentními implementacemi, které mohou vnitřně pracovat na zcela jiném principu. Adaptivní strukturu je tak možno do budoucna v případě potřeby snadno nahradit strukturou jinou a stejně tak vyhledávání cest lze teoreticky nahradit jiným algoritmem, který adaptivní strukturu využije jiným způsobem pro svoje účely.

6.1.1 Interface IEngram

Interface IEngram popisuje požadavky kladené na jednotlivé uzly grafu, se kterým vyhledávací algoritmus pracuje. Proto jej musí implementovat jakýkoliv objekt, který by měl v libovolné implementaci struktury takový uzel reprezentovat. Jsou definovány dvě základní metody:

- Každý uzel musí poskytovat vyhledávacímu algoritmu svoje souřadnice v prostoru úlohy, podle kterých se následně určuje, v jaké oblasti se bod nachází, jak velké v okolí hrozí nebezpečí a podobně.
- Získání seznamu sousedních uzlů. Každý uzel musí být schopen poskytnout aktuálně platný seznam vrcholů, se kterými bezprostředně sousedí. Díky tomu může algoritmus hledání cesty snadno plánovat úseky, které dohromady výslednou trasu složí.

6.1.2 Interface IGraph

Interface IGraph popisuje, jakým způsobem bude algoritmus hledání cest komunikovat se samotnou datovou strukturou a jaké údaje po ní bude požadovat. Tento interface tedy musí implementovat každá struktura, nad kterou v našem systému bude vyhledávání cest probíhat. Jsou definovány dvě základní metody:

- Získání seznamu všech uzlů. Jedná se o základní grafový požadavek. Algoritmus hledání cesty potřebuje znát seznam navigačních bodů. Mezi kterými bude trasu vytyčovat. Struktura splňující interface IGraph mu tento seznam musí poskytovat s tím, že prvky seznamu musejí navíc implementovat již popsaný interface IEngram. Touto kombinací je pro vyhledávací algoritmus zajištěna nutná znalost grafových informací.
- Získání uzlu nejbližšího zvolenému místu v prostoru. Jelikož vyhledávací algoritmus nemá žádnou znalost o adaptivní struktuře, nebyl

by schopen v ní sám dohledávat body jinak než procházením jejich seznamu a například porovnáváním souřadnic s hledanými. Takový postup by samozřejmě nebyl efektivní. Naproti tomu jakákoliv struktura za interfacem ukrytá může snadno využít znalosti o svojí stavbě a bod nejbližší požadovanému místu snadno a efektivně dodat. Vracený vrchol opět musí implementovat interface IEngram.

Po definici rozhraní a rozdělení si úkolů jsme následně mohli každý pokračovat v samostatné práci na svojí části úlohy. Kromě ostatních výhod nám navržené rozhraní také významně usnadnilo počáteční fáze vývoje. Umožnilo nám během této práce používat místo ještě zdaleka ne hotové kolegovy součásti dočasnou jednoduchou náhradu, která, třebaže zdaleka neplnila požadované úkoly požadovaným způsobem, pomohla při testování a ladění v úvodu implementace vznikajícího kódu.

6.2 Návrh tříd

V analýze datových struktur jsem navrhl tři základní třídy. Třídu specifikující vlastní adaptivní mřížku, třídu reprezentující buňku této mřížky a třídu zastupující vrchol buňky. Návrh zahrnuje všechny potřebné vlastnosti a data popsaná v kapitole 5 a metody s nimi pracující. Mřížka je navržena dostatečně obecně, aby ji bylo možno použít i pro jiné aplikace bez nutnosti výrazných změn.

6.2.1 Třída Mesh

Třída Mesh popisuje samotnou adaptivní mřížku, implementuje tedy v kapitole 6.1.2 popsaný interface IGraph. Obaluje modifikovanou strukturu octtree, resp. jejich větší množství. V obecném návrhu jsme se totiž neomezili na pouze krychlový prostor, ale mřížka může mít tvar kvádru složeného z více krychlí. Každá z takových krychlí se může sama o sobě

adaptovat jako octtree s tím, že jsou všechny navzájem propojeny a přilé-hající stěny krychlí navzájem sdílejí vrcholy ve všech úrovních adaptace.

Třída obsahuje metody pro procházení vnitřních struktur pro získávání seznamů existujících elementů (buněk či vrcholů) a také pro jejich udržování a aktualizaci. Nově vznikající elementy se tak do těchto seznamů registrují a zanikající elementy se naopak ze seznamů odstraňují.

6.2.2 Třída Cluster

Třída Cluster reprezentuje buňku adaptivní mřížky. Pro udržování stromové struktury má referenci na svého rodiče a na všechny svoje potomky. Zároveň si udržuje informaci o hloubce úrovně, na které je buňka zkonstruována.

Třída obsahuje odkazy na svoje vlastní vrcholy i pozice připravené pro vrcholy nevlastní. Zároveň definuje metody pro výpočet souřadnic těchto nevlastních bodů. Ty se budou nacházet vždy v polovině hran a úhlopříček, výpočet proto není složitý.

Cluster definuje dvě základní metody split a merge, které implementují stejnojmenné operace popsané v kapitole 3.5. V souvislosti se zavedením vrcholů buněk a jejich sousednosti řeší metody split a merge i dodatečné problémy popsané v kapitole 5.2.

V rámci dělení buňky jsou volány metody pro správné nastavení vlastností nově vzniklých vrcholů a buněk. Zejména se jedná o udržování správné hierarchie stromu, nastavování rodičů a potomků a nastavení členství vrcholů v jednotlivých buňkách. Pro zajištění funkčnosti třídy Mesh (výpis všech vrcholů sítě) jí dává Cluster k dispozici metodu pro výpis všech vrcholů ve svém prostoru, tedy všech vrcholů svých i vrcholů potomků.

6.2.3 Třída Engram

Třída Engram reprezentuje vrchol buňky mřížky. Nese si informaci o svojí přesné pozici v prostoru a kompletní výčet všech buněk, pro které byl vrchol vytvořen (až 8 obklopujících buněk). Díky tomu je jasně definovaná struktura, jakou vrcholy v prostoru tvoří. Každý engram si zároveň s tím nese i informaci o úrovni zanoření, na které byl vytvořen. Pak může poskytovat rychlou metodu vracející aktuálně platné sousedy, postupem popsaným v kapitole 5.2.

Kapitola 7

Implementace

Zdrojové kódy implementace všech popsaných tříd jsou dostupné na přiloženém datovém nosiči, doplněny dokumentačními komentáři jak je u jazyka C# zvykem. Kód a dokumentace se tak navzájem doplňují a poskytují velmi zevrubný a zároveň přehledný popis samotné implementace. Konkrétní umístění všech souborů naleznete popsané v souboru *readme.txt* v kořenovém adresáři nosiče.

Implementace se drží analýzy uvedené v kapitole 6. Odpovídající class-diagramy jednotlivých tříd, jak je poskytuje visual studio, jsou k dispozici v příloze A a poskytují rychlou představu o přesné podobě tříd a interfaců.

Jelikož adaptivní struktura je sama o sobě pouhým nástrojem pro další použití, je na přiloženém datovém nosiči dodána i ukázková aplikace. Jedná se právě o systém vyhledávající nad adaptivní strukturou cestu, jak byl popsán v kapitole 4.1.

Aplikace pracuje následujícím způsobem. Je definována mapa překážek zaznamenávající nepohyblivé objekty (jako překážky byly pro jednoduchost zvoleny koule různých poloměrů) a mapa hrozeb zachycující vždy aktuální pozici a velikost pohyblivých nebezpečných objektů. Vlivy překážek a hrozeb jsou v mapách zaznamenány číselnými váhami. Je vytvořena adaptivní mřížka v základním tvaru (zatím nijak adaptovaná jedna velká

buňka), která pokrývá prostor reprezentovaný mapami.

Adaptace mřížky probíhá tak, že pro každou buňku mřížky (v úvodu tedy pro jedinou) je váhovou funkcí kombinující hodnoty z obou map určena celková váha bloku prostoru reprezentovaného buňkou. Pokud je hodnota váhy moc vysoká, znamená to, že v daném bloku jsou překážky nebo hrozby a buňka je proto rozdělena operací split. Tato operace se nad celým prostorem opakuje až dokud nejsou váhy všech bloků nižší než mezní hodnota, nebo dokud není dosaženo maximální povolené úrovně dělení. Obdobným způsobem se vyhodnocuje a provádí i operace merge nad buňkami, jejichž celková váha je nižší než ke sloučení stanovená mezní hodnota.

Algoritmus vyhledávající cestu pak k adaptované mřížce přistupuje jako ke grafu, nad jehož uzly a cestami mezi nimi určuje optimální trasu.

Spolu s pohybem hrozeb se síť iterativně adaptuje a v každém kroku se znovu hledá výsledná cesta odpovídající pozměněným podmínkám.

Aplikace nebyla vytvořena v rámci této diplomové práce (pouze ji využívá ke svojí činnosti) a její detailnější popis a ukázky jsou proto dostupné v článcích uvedených jako přílohy B a C tohoto textu.

7.1 Použité programovací prostředky

Pro implementaci jsme zvolili objektově orientované prostředí Microsoft .NET Framework a jazyk C#, vizuální stránku obstarává DirectX. Z důvodu změny platformy jsme se rozhodli pro kompletní vlastní návrh a implementaci aplikace, bez využití či inspirace v původním C++ kódu. Naším výchozím bodem byl tedy teoretický popis principů aplikace s možností konzultace u autorů metody.

Na prostředí .NET padla naše volba zejména z následujících důvodů. Na rozdíl od původně použitého C++ je C# čistě objektově orientovaný jazyk a usnadnil nám tak poměrně čistý a jasný návrh a také zvýšil výslednou

čitelnost a srozumitelnost vzniklého kódu. Objektově orientovaný přístup nám také umožnil snadnou definici rozhraní mezi jednotlivými částmi úlohy a tím i snadnou spolupráci.

7.2 Požadavky na překlad a spouštění dodané aplikace

Pro překlad aplikace je třeba mít nainstalován operační systém Microsoft Windows, Microsoft .NET Framework 2.0 SDK a DirectX 9.0 SDK October 2005. Vzhledem k jejich vývoji nelze zaručit stoprocentně správnou funkčnost s novějšími verzemi. Doporučeno je i Microsoft Visual Studio 2005, ve kterém je celý projekt zpracován, překlad by ovšem měl být teoreticky možný i bez jeho instalace.

Pro vlastní spuštění by pak měl stačit opět operační systém Microsoft Windows, Microsoft .NET Framework 2.0 Runtime a DirectX 9.0.

Kromě operačního systému a Visual Studia se jedná o produkty volně dostupné z webu společnosti Microsoft (www.microsoft.com) a zároveň jsou připraveny na datovém nosiči přiloženém k této práci. Jejich konkrétní umístění naleznete popsané v souboru *readme.txt* v kořenovém adresáři nosiče.

Kapitola 8

Testování

Vzhledem k cílům úlohy a použité aplikaci se většina testování věnovala výsledné implementace hledání cest. Přesto se některá měření zaměřila na specifické vlastnosti mřížky samotné a i výsledky testů ostatních jsou na její práci významně závislé.

8.1 Podmínky průběhu testů

Chování testovací aplikace bylo měřeno na PC s procesorem Athlon XP 2000+ a 512 MB DDR RAM, operačním systémem Windows XP SP2 a bez jiných zbytečně spuštěných programů či služeb.

Významnou část měření jsem neprováděl sám, ale probíhala jako spolupráce při přípravě článku uvedeného v příloze B tohoto textu. Článek je psaný v anglickém jazyce, proto vždy uvedu český popis podmínek testování i testu samotného. Použiji však některé z původních grafů s anglickými popisky.

V aplikaci jsme vygenerovali zvolené množství pevně definovaných překážek různé velikosti, vizualizovaných kulovými objekty v prostoru, a pohyblivé hrozby reprezentované malou červenou krychličkou. Za běhu pro-

gramu se hrozby náhodně pohybují z místa na místo a adaptivní mřížka na jejich pohyb reaguje úpravou svojí struktury.

Mapa překážek je předpočítána a vyplněna odpovídajícími hodnotami ještě před započetím hlavní smyčky programu, její příprava tedy nijak neovlivňuje samotný průběh měření běhu programu. Optimální cesta se přepočítává po každé adaptaci sítě. Měření je vždy zaměřeno na vlastní operace a nezahrnuje čas potřebný k vizualizaci jejich výsledků.

8.2 Testovací scény

Pro prozkoumání chování algoritmů nad odlišnými druhy a rozsahy dat jsme definovali několik různorodých typů scén. Pokud není u konkrétních měření uvedeno jinak, prováděly se všechny testy nad následujícími čtyřmi scénami.

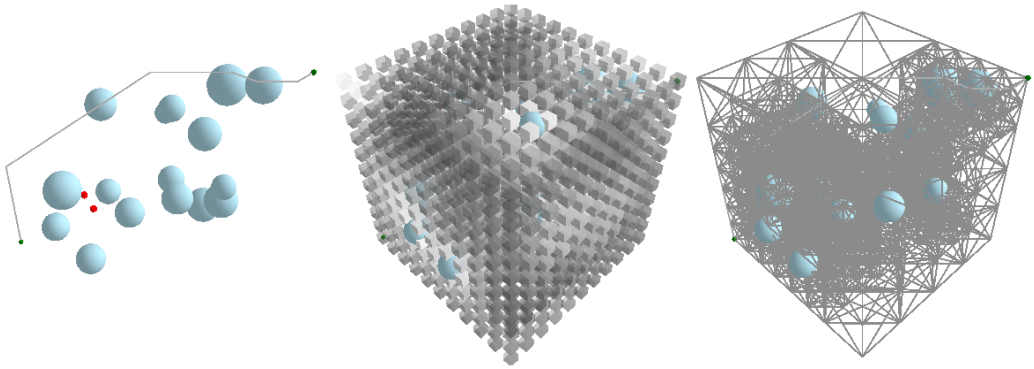
Scéna 1 je složena z 8 náhodně rozmístěných kulových překážek náhodně zvolené velikosti, které zaujímají přibližně 3% objemu prostoru úlohy. Mapy překážek a hrozeb mají rozlišení 32 x 32 x 32 hodnot a maximální povolená úroveň dělení adaptivní mřížky je 4 (nejmenší buňka má tedy hranu o délce $1/2^4$ celkového rozměru).

Scéna 2 je definována stejně jako scéna 1, ovšem povoluje o stupeň hlubší úroveň dělení, tedy do úrovně 5.

Scéna 3 se skládá z 16 překážek, které zaujímají přibližně 6% prostoru úlohy. Rozlišení map překážek a hrozeb bylo 64 x 64 x 64 hodnot a maximální úroveň dělení adaptivní mřížky je 4.

Scéna 4 je definována stejně jako scéna 3, ovšem povoluje o stupeň hlubší úroveň dělení, tedy do opět úrovně 5.

Příklad scény s 16 překážkami a dvěma pohyblivými hrozbami je uveden na obrázku 8.1. Vlevo vidíme samotnou scénu společně s cestou nalezenou



Obrázek 8.1: Příklad scény zpracovávané aplikací.

mezi dvěma protějšími rohy reprezentovaného prostoru. Uprostřed jsou na scéně zobrazeny hodnoty vah mapy překážek. Čím světlejší barva, tím větší část daného bloku zaujímají překážky a naopak. Vpravo pak je pak v téže scéně zobrazena samotná adaptivní mřížka.

8.3 Provedené testy a jejich výsledky

8.3.1 Rychlost operací split a merge v závislosti na povolené hloubce úrovně

Měření bylo zaměřeno na vlastní adaptivní mřížku a zpracování překážek, hrozeb i samotný algoritmus hledání cesty byly odpojeny. Měření tedy neprobíhalo nad výše definovanými testovacími scénami. Měřeny byly pouze čisté rychlosti operací split a merge.

Pro měření byl použit jeden základní cluster (úrovně 0), který následně umělým zásahem kompletně rozdělen na potomky až do požadované úrovně a následně byly tyto nově vytvořené clustery opětovně sloučeny do původního jediného.

Při každém splitu do nižší úrovně je buňka sítě rozdělena na 2^{d*n} buněk dceřiných, kde n je úroveň zanoření a d je dimenze, tedy v našem případě

level	pocet clusteru	pocet engramu
0	1	8
1	9	27
2	73	179
3	585	1395
4	4681	11123
5	37449	88947
6	299593	711539

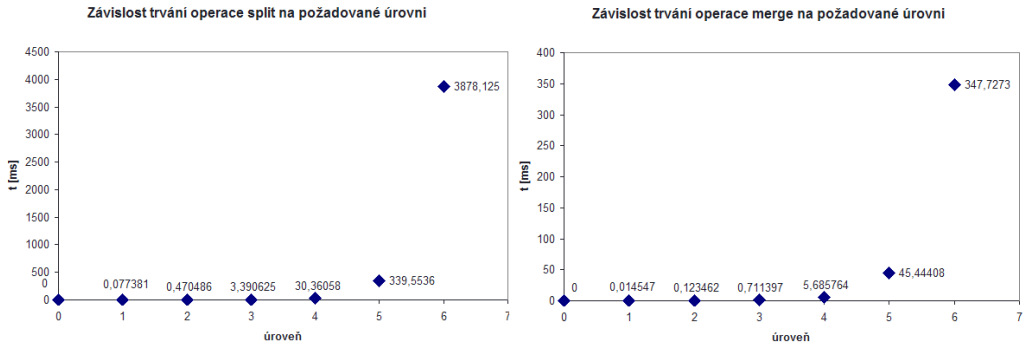
Tabulka 8.1: Exponenciální nárůst počtu elementů sítě s rostoucí úrovní dělení.

3. Tento exponenciální nárůst množství elementů sítě (buněk a vrcholů) při plném dělení jedné buňky zachycuje tabulka 8.3.1.

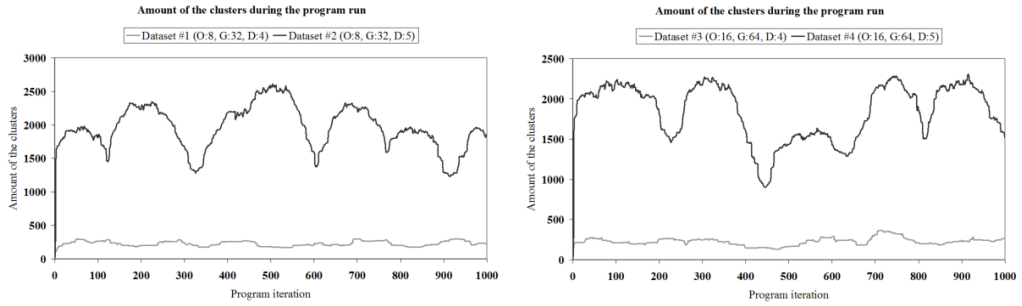
Vzhledem k tomu, že zejména pro nižší úrovně zanoření (relativně nízký počet vznikajících a zanikajících elementů) jsou operace velmi rychlé (teoreticky zlomky milisekund), bylo měření prováděno opakovaně a výsledný čas pak vydělen počtem opakování. S rostoucí dobou potřebnou k provedení daných operací jsem pak počet opakování snižoval: pro úroveň 1 10 000x, pro úroveň 2 5 000x, pro úroveň 3 1 000x, pro úroveň 4 500x, pro úroveň 5 100x a pro úroveň 6 50x.

Výsledky získané při měření rychlostí operací split a merge na jednotlivých úrovních maximálního povoleného dělení ukazuje obrázek 8.2.

Z těchto výsledků je dobře patrné, že operace merge je řádově rychlejší než split. Při porovnání s počty zpracovávaných elementů je také zřejmé, že prodloužení doby trvání operací zhruba odpovídá nárůstu objektů, které se musejí zpracovávat. Zatímco nárůst rychlosti operace merge se tempa růstu elementů drží poměrně přesně (např. vzrůst z předposlední na poslední hodnotu přibližně 8x stejně jako u počtu elementů), u splitu je nárůst o něco větší (ve zmiňovaném případě přibližně 11x). Na vině je pravděpodobně nutná kontrola, kdy při vytváření nového clusteru je třeba nejprve odhalit v okolí již existující vrcholy, které lze pro cluster použít.



Obrázek 8.2: Rychlost operací split a merge v závislosti na povolené úrovni dělení.



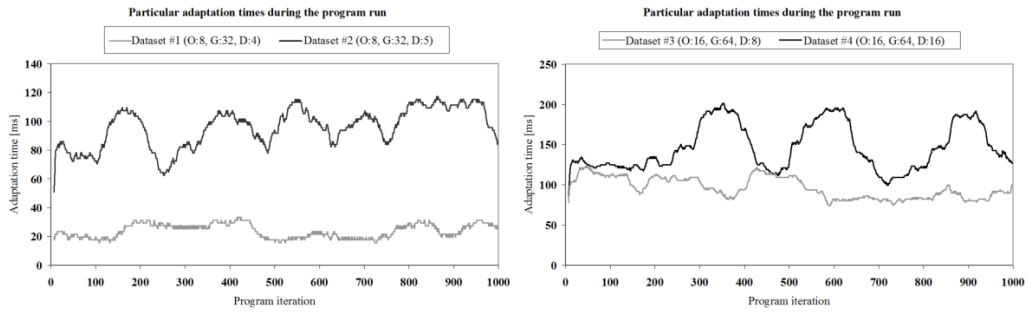
Obrázek 8.3: Vývoj počtu buněk v čase.

Svoji roli může hrát i to, že zatímco při operaci split vždy vznikají nové objekty (buňky a vrcholy), v průběhu operace merge ne nutně tyto objekty zanikají. Jsou pouze připraveny ke zrušení, které ale řídí garbage collector prostředí .NET.

8.3.2 Časový vývoj množství buněk sítě

Sledovali jsme vývoj počtu buněk sítě v závislosti na časovém průběhu adaptace jednotlivých testovacích scén. Tento vývoj zachycuje obrázek 8.3, kde vlevo vidíme situaci na scénách 1 a 2 a vpravo na scénách 3 a 4.

V počátku průběhu pozorujeme rychlý nárůst počtu buněk z počáteční



Obrázek 8.4: Vývoj rychlosti adaptace sítě v čase.

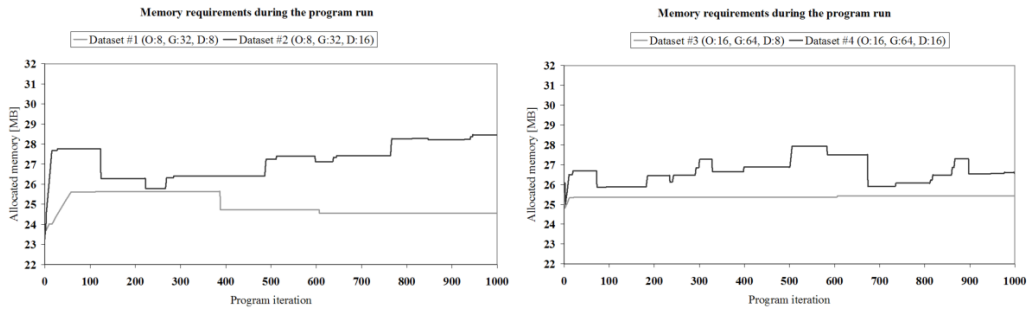
nulové hodnoty, jak se síť adaptuje okolo překážek v prostoru. Následný průběh pak vykazuje nepravidelné kolísání. To je způsobeno náhodným pohybem hrozeb. Dokud se hrozba pohybuje v blízkém okolí některé z překážek, jsou do nejnižší úrovně rozděleny buňky v jejich společném okolí. Jakmile se ovšem hrozba vzdálí do původně prázdného prostoru, zvedne hodnotu vah v dané oblasti a tím způsobí další dělení a nárůst počtu buněk. Oblast, odkud se vzdálila, ovšem zůstává kvůli přítomné překážce i nadále rozdělená.Toto chování pozorujeme na všech testovacích scénách.

Ve scénách 2 a 4 je toto kolísání více patrné, počty buněk se během kolísání liší daleko výrazněji než v případě scén 1 a 3. To je ale způsobeno tím, že nárůst počtu buněk je s rostoucí úrovní exponenciální, jak jsme již viděli v kapitole 8.3.1. Každá další úroveň tak znamená mnohem vyšší nárůst počtu elementů sítě.

8.3.3 Rychlost adaptace sítě

Zajímavým aspektem je rychlost adaptace sítě v jednotlivých iteracích běhu programu. Abychom se zbavili náhodných výkyvů, nepoužili jsme přímo naměřené hodnoty, ale jejich klouzavý průměr se zvolenou periodou 8. Takto zpracované hodnoty jsou zachyceny v grafech na obrázku 8.4.

Porovnáním průběhů je dobře patrné, že rychlost adaptace sítě výrazně závisí zejména na maximální povolené úrovni dělení sítě. Na grafech mů-



Obrázek 8.5: Vývoj spotřeby paměti v čase.

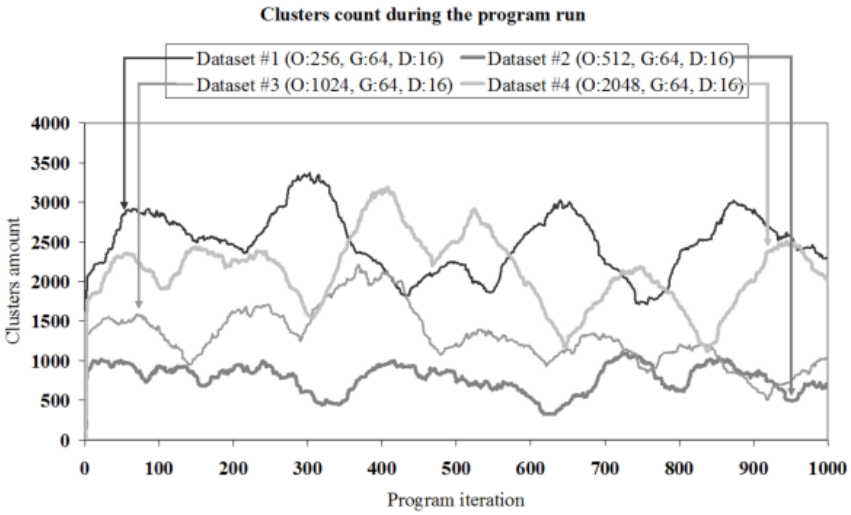
žeme opět pozorovat kolísání měřené doby, které stejně jako v minulém případě souvisí s náhodným pohybem hrozeb. Pokud si hrozba svým pohybem vynutí dělení více buněk, je adaptace pomalejší.

8.3.4 Spotřeba paměti

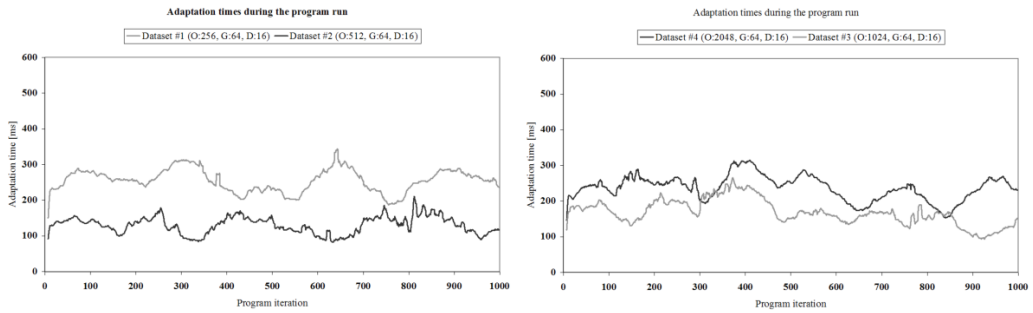
Na obrázku 8.5 je znázorněno množství paměti v MB, kterou aplikace v průběhu iterací alokuje. Naměřené hodnoty nemusí být úplně vypovídající, neboť jsou ovlivněny garbage collectorem prostředí .NET. Přesto lze říci, že podstatná část paměti je alokována pro mapy překážek a hrozeb, které například pro rozměr 32 udržují každá 32 * 32 * 32 hodnot typu float. Toto množství zabrané paměti je ale po celou dobu běhu aplikace konstantní a schodovité změny v grafech jsou tedy způsobeny samotnou adaptací sítě, byť poklesy nemusejí kvůli činnosti garbage collectoru prostředí .NET nastávat bezprostředně při operacích merge.

8.3.5 Závislost na počtu překážek

Dalším cílem našeho testování bylo ověřit, zda je chod aplikace závislý na počtu překážek definovaných v prostoru úlohy. K tomuto testu jsme připravili čtyři scény, které se tentokrát lišily pouze počtem překážek (256,



Obrázek 8.6: Vývoj počtu buněk sítě pro různé počty překážek.

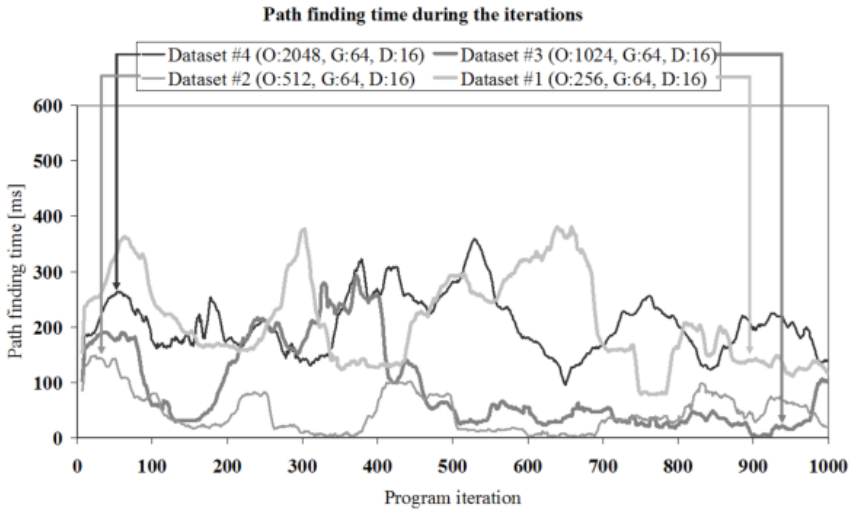


Obrázek 8.7: Doby trvání adaptace sítě pro různé počty překážek.

512, 1024 a 2048). Rozlišení map je ve všech případech 64 x 64 x 64 a maximální povolená úroveň dělení 4.

Na obrázku 8.6 vidíme časový vývoj počtů buněk pro všechny definované scény a je patrné, že tyto počty zřejmě na množství překážek přímo závislé nejsou. Rozdíly mezi jednotlivými průběhy jsou způsobeny zejména náhodným rozmístěním překážek.

Stejná situace nastává i na obrázku 8.7, kde je zachycen vývoj dob trvání adaptace sítě a obrázek 8.8,ukazující vývoj dob hledání cesty. Z důvodu odstranění náhodných výkyvů, jsme v obou případech opět místo namě-



Obrázek 8.8: Doby trvání hledání cesty pro různé počty překážek.

řených hodnot použili jejich klouzavý průměr.

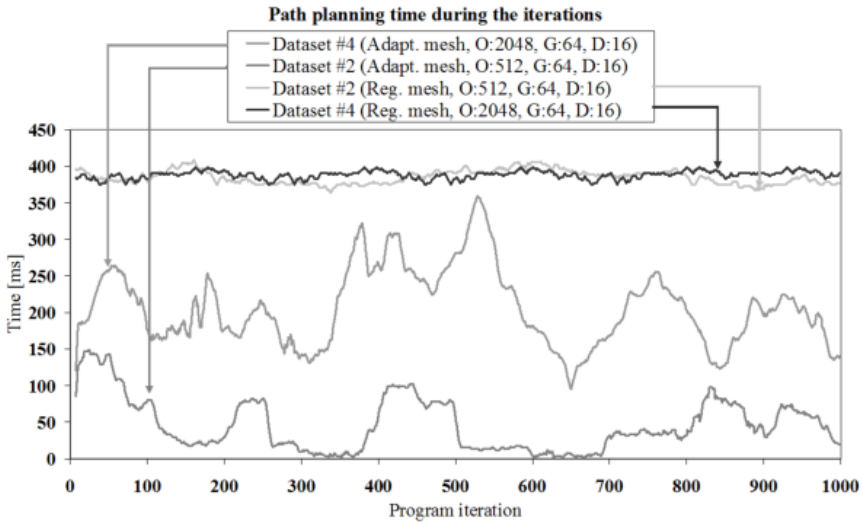
Ukazuje se tedy, že výše uvedené operace nejsou na množství překážek přímo závislé. Jedinou operací závislou na počtu překážek tedy zůstává preprocessing, který po spuštění aplikace všechny překážky prochází a vyplňuje jejich mapu odpovídajícími hodnotami vah. Po tomto úvodu již ale množství překážek zásadní roli nehraje.

8.3.6 Porovnání adaptivní a pravidelné mřížky

Vzhledem k tomu, jak je aplikace implementována, bylo možné v ní pro testování nahradit adaptivní mřížku mřížkou pravidelnou. Díky tomu lze snadno tyto dva přístupy porovnat.

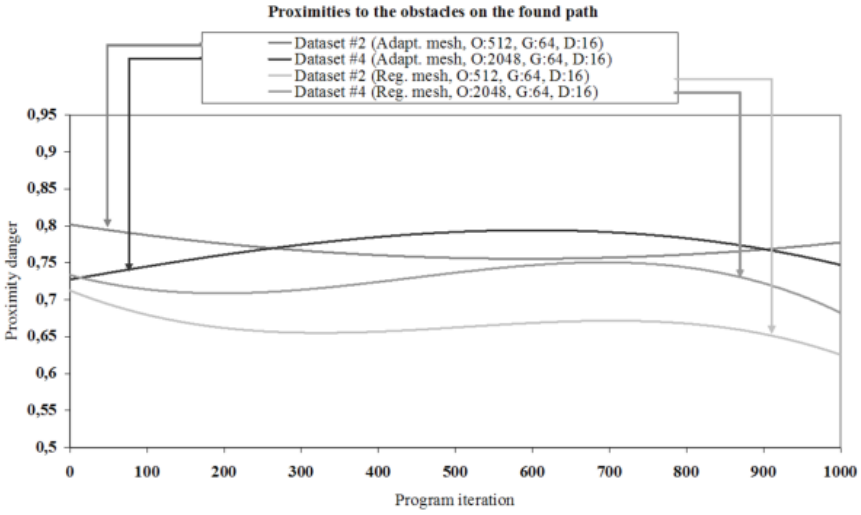
Porovnání bylo provedeno nad dvěma scénami z předchozí kapitoly s rozlišením map 64 x 64 x 64 a stejně definovanými scénami nad pravidelnou mřížkou o odpovídajících rozměrech 64 x 64 x 64 buněk.

Na obrázku 8.9 vidíme, že zatímco u pravidelné mřížky se časy potřebné k nalezení cesty pohybují po celou dobu běhu programu v oblasti kolem



Obrázek 8.9: Doby hledání cesty nad pravidelnou a adaptivní mřážkou.

400ms, časy potřebné u adaptivní mřížky kolísají, jak se dalo předpokládat. Podstatné ovšem je, že tyto časy zdaleka nedosahují stejně vysoké hodnoty. U scény 4 je průměrná hodnota přibližně 200ms a u scény 2 dokonce jen asi 45ms. Výhoda použití adaptivní datové struktury je tedy jasně patrná. Podstatné je i to, že obě implementace dávají kvalitativně si odpovídající výsledky. Optimalita nalezené cesty byla měřena pomocí váhy nebezpečí na nalezené trase. Hodnota této váhy byla měřena ve všech uzlech, z nichž se cesta skládala, a z těchto hodnot bylo vybráno maximum. Maxima byla zaznamenána během chodu aplikace a následně proložena polynomiální regresí třetího řádu. Čím nižší maxima získáváme, tím lepší je kvalita dosaženého výsledku. Obrázek 8.10 porovnává průběhy maxim pro jednotlivé scény a je patrné, že si dosažené výsledky přibližně odpovídají. Rozdíly jsou stejně jako dříve způsobeny náhodným rozmístěním překážek.



Obrázek 8.10: Vzdálenost cesty od překážek.

8.3.7 Počet buněk sítě v závislosti na rozložení překážek a hrozeb

Během předchozích testů se jasně projevila závislost stavu adaptivní mřížky na rozmístění překážek a hrozeb v reprezentovaném prostoru. Pokud je jejich rozmístění takové, že zůstávají v prostoru dostatečně velké volné oblasti, je mřížka v těchto oblastech nerozdělená a obsahuje tak menší počet buněk. Oblasti obsazené jsou naopak rozděleny do nejvyšší možné úrovně. Hranice mezi obsazenými a neobsazenými oblastmi jsou pak adaptovány částečně, podle klesajícího vlivu překážek a hrozeb.

Pokud je ovšem překážek dostatečné množství nebo jsou rozmístěny víceméně rovnoměrně po celém objemu prostoru, je mřížka adaptována do nejvyšší povolené úrovně prakticky celá. Protože má každá překážka i hrozba definovaný určitý rozsah vlivu na váhy v mapách i mimo svůj vlastní objem, často jsou rozděleny i buňky v prostorách mezi těmito objekty, kde se i přes jejich nepřítomnou vlivy blízkých objektů nasčítají přes stanovenou mez.

Modifikace výpočtu váhové funkce a samotného zpracování překážek

a hrozeb může tak chování celé aplikace velmi výrazně ovlivnit. Jejich vhodnou změnou by bylo zřejmě možné rozdělování volného prostoru mezi objekty a v jejich bezprostřední blízkosti výrazněji omezit.

Kapitola 9

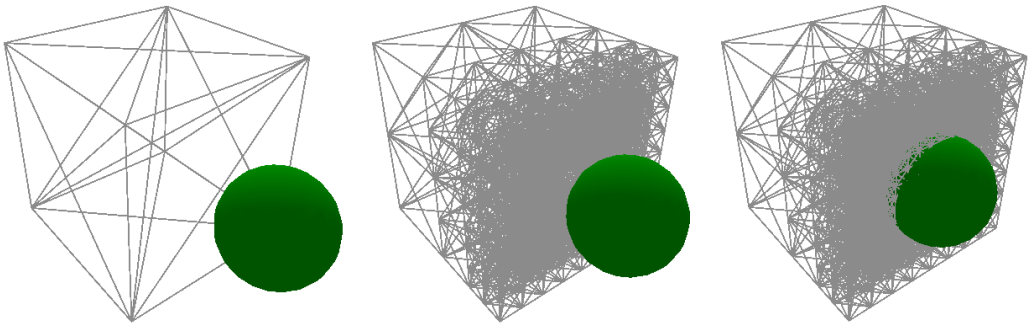
Modifikace povrchu

9.1 Potřebná rozšíření a úpravy

Pro úlohu modifikace povrchu jsem použil zcela nepozměněnou adaptivní strukturu ve formě, v jaké byla vyvinuta pro původní zadání a popsána v kapitole 6.2. Změny si však pochopitelně vyžádala ukázková aplikace. V té využívám původní metody volající adaptaci sítě, systém práce s hrozbami a vizualizační část. Kód starající se o vyhledávání cesty, pohyb agentů a zpracování překážek nebyl použit.

Systém pohybu hrozeb jsem rozšířil o specifickou hrozbu představující „razítko“ modifikující povrch a upravil jeho vizualizaci. Razítko je pro jednoduchost reprezentováno koulí, která se při svém pohybu otiskuje do povrchu kvádru tvořeného jednou základní buňkou adaptivní mřížky. Koule do kvádru opakovaně vniká a zase jej opouští, čímž vyvolává adaptaci mřížky oběma směry.

Požadavkem úlohy bylo vytvořit pokud možno věrný otisk předmětu v reálném čase. Jelikož je adaptivní mřížka tvořená krychlovými buňkami, znamená věrný otisk malé rozměru těchto krychliček v otiskové oblasti a tedy vysokou úroveň dělení sítě. Již předchozí testy ovšem odhalily, že nejvýznamnějším parametrem ovlivňujícím rychlost adaptace, je právě



Obrázek 9.1: Ukázka otiskování razítka do materiálu.

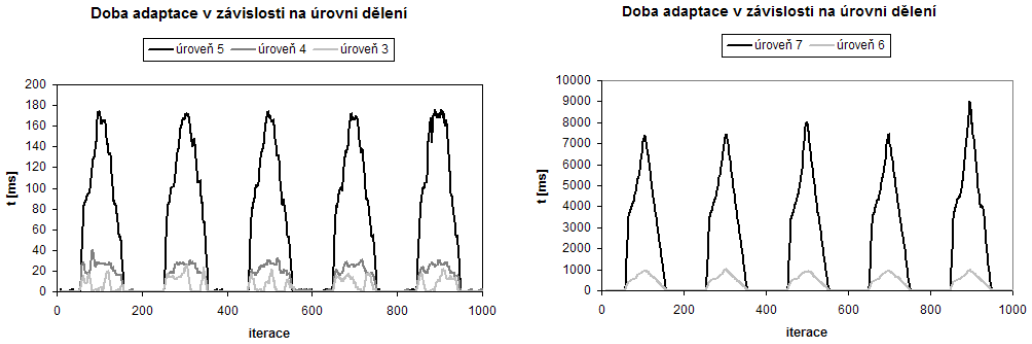
maximální povolená úroveň dělení z důvodu exponenciálního nárůstu počtu elementů sítě, jak je uvedeno v kapitole 8.3.1.

9.2 Testování

Pro měření jsem připravil scénu čítající prázdnou buňku adaptivní mřížky a kulové razítko, které se do buňky do poloviny zanoří a tím způsobí její adaptaci. Ukázka scény je uvedena na obrázku 9.1.

Měřené varianty se lišili jen maximální povolenou hloubkou dělení, otestoval jsem rozsah od 3 do 7. Naměřené hodnoty jsem opět pro vyhlazení náhodných výkyvů proložil jejich klouzavým průměrem a výsledek je vidět na obrázku 9.2. V časovém průběhu vidíme pravidelné střídání nárůstů a poklesů měřených časů, které souvisí se zanořováním koule do materiálu a jeho opětovným opouštěním.Pokud je koule mimo materiál, není mřížka rozdělena a časy potřebné k zpracování jednoho kroku

Je patrné, že doba potřebná k adaptaci spolu s úrovní dělení výrazně narůstá. Jak se dalo očekávat již z dřívějších měření, nejsou výsledky příliš uspokojivé. Již na úrovni dělení 5 trvá každá adaptace přibližně 180ms. Systém je tedy schopen provést asi 5 – 6 výpočtů během jedné vteřiny, což už jistě nemůže působit plynulým dojmem. Doba tedy přestává být únosná pro jakékoliv otiskování v reálném čase. Naměřené časy navíc postihují



Obrázek 9.2: Rychlost adaptace v závislosti na úrovni detailů

pouze vlastní výpočet a nijak nezohledňují dobu potřebnou k vizualizaci. Při reálném použití by tedy potřebné časy ještě vzrostly.

Je třeba si uvědomit, že omezená hloubka dělení významně ovlivňuje kvalitu výsledného otisku. Obecně rozdělení do n-té úrovně přinese schopnost rozlišovat detaily otisku jen do rozměru $1/2^n$ délky hrany původní buňky. Při použití ještě únosné hloubky dělení 4 tedy dostaneme poměrně hrubý otisk složený z krychliček o hraně $1/16$ délky hrany základní buňky, což zdaleka nemusí být vizuálně uspokojivý výsledek

Kapitola 10

Závěr

V rámci diplomové práce jsem prozkoumal několik různých variant možných adaptivních dělení prostoru a navrhl a implementoval 3D adaptivní mřížku založenou na silně modifikované struktuře octtree. V průběhu návrhu se postupně podařilo vyřešit všechny zásadní problémy způsobené přechodem z 2D do 3D prostoru. Výsledná implementace pak překonala původně skeptická očekávání a v aplikaci pro hledání cest je výsledná mřížka paměťově i rychlostně uspokojivá. I porovnání s obyčejnou pravidelnou mřížkou jasně demonstruje výhody použitého řešení. S využitím navržené datové struktury pak Bc. Petr Brož úspěšně vytvořil kompletní systém hledání cest v neznámém dynamickém 3D prostředí a články zabývající se tímto navrženým systémem byly nabídnuty na několik mezinárodních konferencí.

Druhá část práce, aplikace navržené adaptivní mřížky v rámci odlišné úlohy, již tak úspěšná nebyla. Jasně se ukázal zásadní rozdíl mezi použitím adaptivní mřížky jako pomocné datové struktury (případ hledání cest) a jako modelu pro vizualizaci, jak tomu bylo v případě druhé úlohy. Zatímco v prvním případě jsou na mřížku kladeny nároky pouze aplikací a uspokojivých výsledků je možno dosáhnout i při nižších úrovních dělení, v případě druhém je prakticky vždy požadována vysoká úroveň detailů minimalizující „hranatost“ vizualizovaných modelů. Měření jasně uká-

zala, že pro dosažení takových detailů implementovaná mřížka vhodná není. Otázka do budoucna může znít, jak by testy dopadli při použití jednodušší struktury, která nebude poskytovat grafové rozhraní, jež nebylo pro úlohu třeba. Přesto ale pokusy prokázaly, že adaptivní mřížka je připravena natolik obecně, že ji lze aplikovat bez jakýchkoliv úprav i v naprosto jiném vhodném typu úlohy, než pro jaký byla navržena.

V budoucí práci by mohlo být zajímavé rozšířit mřížku o prioritní fronty, jednu pro buňky určené k rozdělení a jednu pro buňky určené ke slučování. Bylo by zajímavé prozkoumat, zda nelze vhodným řazením takových front dosáhnout urychlení procesu adaptace a úspor systémových prostředků. Podnětným směrem by mohlo být také zkoumání různých kritérií pro řazení v těchto frontách.

V aplikaci pro hledání cest by bylo také vhodné prozkoumat detailněji váhovou funkci rozhodující o rozdělení a slučování buněk, která má zásadní vliv na chování celé aplikace. Prozkoumat přesný vliv jednotlivých započítávaných parametrů a určit několik možných váhových funkcí pro různá chování sítě, například pro přesnější odlišení obsazeného a volného prostoru se snahou omezit dělení oblastí sousedících s překážkami. Taková změna by ve výsledku mohla znamenat menší počet buněk v síti a proto další urychlení její adaptace. Zajímavým pokusem by mohlo být i adaptivní určování maximální povolené hloubky dělení v závislosti na vypočítaných vahách v různých oblastech reprezentovaného prostoru. Jako pozitivní změnu bych viděl i jiný způsob reprezentace překážek. V momentální implementaci je prostor překážky rozdělen na maximální povolenou úroveň v celém objemu překážky. Přitom její vnitřek není zajímavý a vzniklé buňky jen zpomalují rychlost celého systému. Volání adaptace sítě pouze okolo povrchu překážek by mohlo mít rovněž pozitivní dopad na rychlost.

Diplomová práce uspokojivě splnila stanovené cíle a zároveň klade možnost dalších úprav a vývoje celého systému do budoucna.

Použité zkratky a pojmy

ASM – Adaptive Spatial Memory (adaptivní prostorová paměť)

BSP Tree – Binary Space Partitioning (stromové binární dělení prostoru)

CLUSTER – buňka adaptivní sítě

ENGRAM – vrchol adaptivní sítě

GEOMORPH – plynulý přechod mezi dvěma úrovněmi detailů modelu (v práci je používána počeštěná varianta GEOMORF)

kD Tree – k-rozměrný strom

LOD – Level of Detail (úroveň detailů)

MERGE – operace sloučení potomků do rodičovské buňky

OCTTREE – stromová struktura adaptivního dělení 3D prostoru

QUADTREE – stromová struktura adaptivního dělení 2D prostoru

SPLIT – operace dělení buňky na potomky

Literatura

[1] Hoppe, Hugues. *Progressive Meshes*. Computer Graphics, SIGGRAPH'96 Proceedings, Microsoft Research, 1996.

[2] Hoppe, Hugues. *View-Dependent Refinement of Progressive Meshes*. Computer Graphics, SIGGRAPH'97 Proceedings, Microsoft Research, 1997.

[3] Hoppe, Hugues; DeRose, Tony; Duchamp, Tom; McDonald, John; Stuetzle, Werner. *Mesh Optimalization*. Computer Graphics, SIGGRAPH'93 Proceedings, 1993.

[4] Žára, Jiří; Beneš, Bedřich; Felkel, Petr. *Moderní počítačová grafika*. Computer Press, 1998.

[5] Brož, Petr. *Path Planning in Combined 3D Grid and Graph Environment*. Proceedings of the 10th Central European Seminar on Computer Graphics, 2006.

[6] Apu, Russel Ahmed; Gavrilova, Marina. *Adaptive Spatial Memory Representation for Real-Time Motion Planning*. Proceedings of the 8th International Conference on Computer Graphics and Artificial Intelligence, 2005.

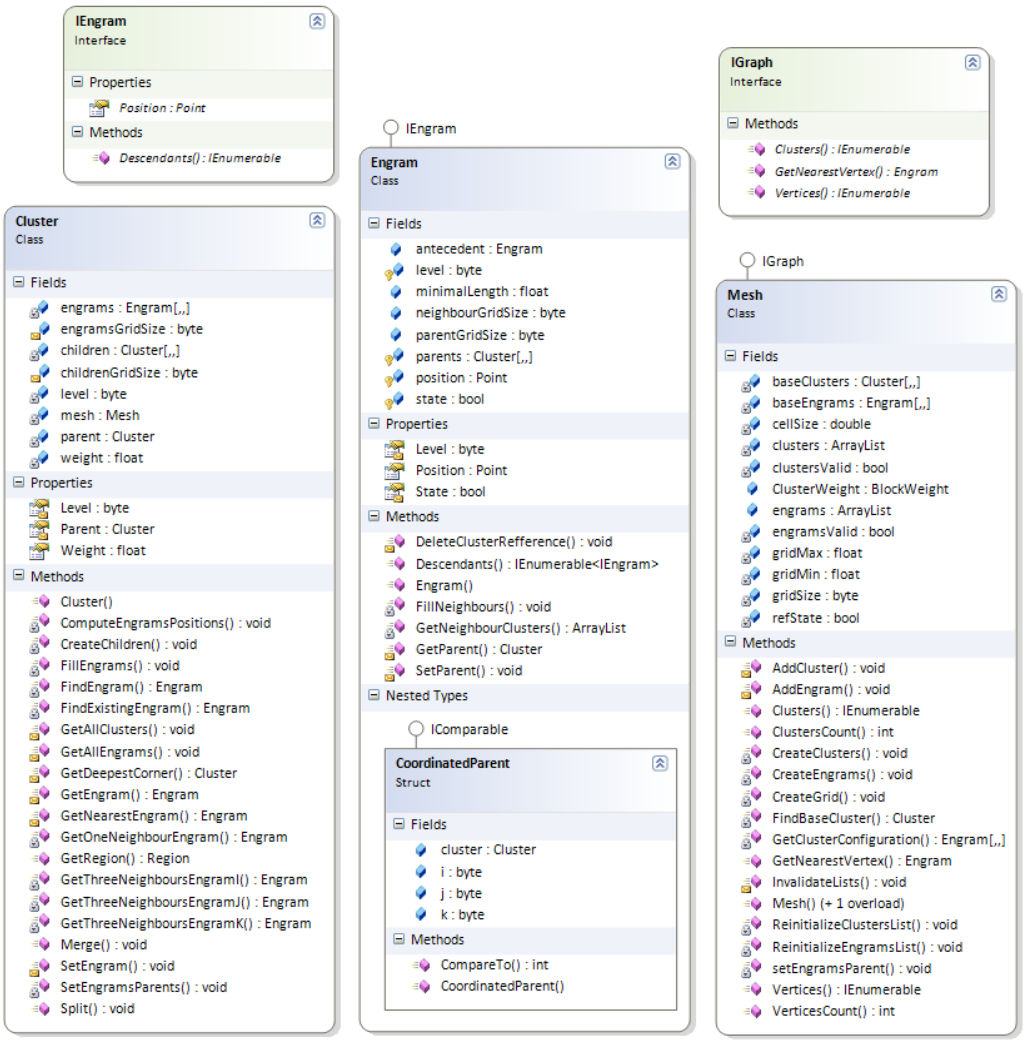
[7] *http://en.wikipedia.org/wiki/BSP_tree*.

[8] *http://en.wikipedia.org/wiki/Voronoi_diagram*.

[9] *http://en.wikipedia.org/wiki/Kd_tree*.

Příloha A

Class diagramy



Obrázek 10.1: Class diagramy tříd a interfaců adaptivní datové stuktury.

Příloha B

Path planning in combined 3D grid and graph environment

Příloha C

Path planning in dynamic environment using an adaptive mesh