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

Diplomová práce

Plánování cest pro virtuální realitu

Plzeň, 2008 Petr Brož

Zadání

Poděkování

Děkuji vedoucí této diplomové práce, Doc. Dr. Ing. Ivaně Kolingerové, za její vědecký dohled a podporu již od mého bakalářského studia, za užitečné rady a připomínky, které umožnily vznik této práce. Stejně důležité poděkování patří mým rodičům za jejich trpělivost a podporu během mého studia. V neposlední řadě pak děkuji Ing. Přemyslu Zítkovi a Ing. Michalu Zemkovi za jejich spolupráci a poskytnutí důležitých datových struktur, bez kterých by se systém vyvíjený v rámci mé diplomové práce neobešel.

Prohlášení

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

V Plzni dne ....

Path planning for virtual reality

Path planning in general is a well known problem that has been extensively studied in many scientific disciplines. It defines a task of finding a path between two given spots in an abstract environment representation so that the path satisfies certain criterion of optimality, e.g. the shortest path, the fastest path or the path with maximal clearance among all obstacles. Although there are many methods solving this task, they usually assume the examined space does not change and is completely known in advance. Today's applications, however, do not have to meet these requirements, especially in case of virtual reality or computer games. Therefore, this thesis proposes a general model for the real-time path planning in a dynamic environment. It uses a regular triangulation for dynamic space subdivision and a heuristic algorithm providing fast way to find a path among all obstacles registered in the triangulation.

Obsah

1 Úvod **1**
- 1.1 Cíl projektu . . . 1
- 1.2 Předchozí výzkum . . . 2
- 1.3 Zvolené řešení . . . 4
- 1.4 Organizace textu . . . 4

2 Teorie **5**
- 2.1 Definice problému . . . 5
- 2.2 Klasifikace algoritmů . . . 6
- 2.3 Reprezentace prostředí . . . 6
- 2.4 Statický path-planning . . . 7
  - 2.4.1 *Single source* algoritmy . . . 7
  - 2.4.2 *All-pairs shortest path* algoritmy . . . 10
  - 2.4.3 *Single pair shortest path* algoritmy . . . 11
- 2.5 Dynamický path-planning . . . 12

3 Původní řešení **14**
- 3.1 Návrh . . . 14
- 3.2 Implementace . . . 15
- 3.3 Analýza a výsledky . . . 16
  - 3.3.1 Standardní konfigurace . . . 16
  - 3.3.2 Vysoký počet překážek . . . 17
- 3.4 Komplexní překážky . . . 20

4 Nové řešení **21**
- 4.1 Obecný návrh . . . 21
- 4.2 Regulární triangulace . . . 22
  - 4.2.1 Metody konstrukce . . . 23
  - 4.2.2 Vkládání bodů . . . 24
  - 4.2.3 Odebírání bodů . . . 25
- 4.3 *Gaps filling* algoritmus . . . 25

5 Implementace **30**
- 5.1 Systém hledání cest . . . 30
- 5.2 Testovací aplikace **GalaxyWars** . . . 33
- 5.3 Testovací aplikace **Measuring** . . . 36

6 Experimenty a výsledky **37**
6.1 Doba předzpracování . . . 38
6.2 Doba registrace překážky . . . 39
6.3 Hledání cesty a její kvalita . . . 42
6.4 Různé velikosti překážek . . . 45
6.5 Rozmístění překážek . . . 45

7 Závěr **49**
7.1 Splnění cílů . . . 49
7.2 Výsledky . . . 49
7.3 Další postup . . . 50

Přehled zkratek a značení **51**

Autorský rejstřík **54**

Rejstřík zkratek **55**

A Obrázky **56**

B Uživatelská dokumentace **66**
B.1 Aplikace **GalaxyWars** . . . 66
B.2 Aplikace **Measuring** . . . 67

C Článek ASRC06 **69**

D Článek CESCG06 **78**

E Článek CESCG07 **86**

F Článek SCCG07 **94**

Seznam obrázků

1.1 Ukázka původního projektu [AG05] pro hledání cest ve 2D 3
1.2 Ukázka projektu bakalářské práce [Bro06a] 3

2.1 Graf viditelnosti 8
2.2 Hledání cest v rastru 8
2.3 Ukázka minimální kostry v orientovaném, hranově ohodnoceném grafu 9
2.4 Cesta pro "lidské" avatary (převzato z [BT98]) 13

3.1 Ukázka použití *image distance transform* 15
3.2 Původní řešení - počet clusterů u standardní konfigurace 17
3.3 Původní řešení - doba adaptace u standardní konfigurace 18
3.4 Původní řešení - nebezpečí cesty u standardní konfigurace 18
3.5 Původní řešení - počet clusterů při velkém počtu překážek 19
3.6 Původní řešení - doba adaptace při velkém počtu překážek 19

4.1 Power vzdálenost, ortogonalita dvojice vážených bodů 23
4.2 Chování *gaps filling* algoritmu 26
4.3 Algoritmus *gaps filling* - váhy cest při náhodných změnách grafu 28
4.4 Algoritmus *gaps filling* - počet zprac. uzlů při náhodných změnách grafu 28
4.5 Algoritmus *gaps filling* - váhy cest při lokalizovaných změnách grafu 29
4.6 Algoritmus *gaps filling* - počet zprac. uzlů při lokalizovaných změnách grafu . . 29

5.1 Základní struktura navrhovaného systému hledání cest **TrippSys** 31
5.2 Ukázky aplikace **GalaxyWars** (snímky 1 až 8) 34
5.3 Ukázky aplikace **GalaxyWars** (snímky 9 až 16) 35

6.1 **TrippSystem** - předzpracování pro různé počty překážek 38
6.2 **TrippSystem** - doba registrace pro různé počty překážek 39
6.3 **TrippSystem** - doba předzpracování 40
6.4 **TrippSystem** - doba registrace 40
6.5 **TrippSystem** - doba nalezení cesty 43
6.6 **TrippSystem** - váha nalezené cesty 43
6.7 **TrippSys** a **Dispatcher** - doba nalezení cesty 44
6.8 **TrippSys** a **Dispatcher** - průměrná váha cesty 44
6.9 **TrippSys** a **Dispatcher** - doba předzpracování (překážky různých velikostí) . 47
6.10 **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky různých velikostí) . 47
6.11 **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky rozdělené do shluků) 48
6.12 **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky v jednom shluku) . 48

A.1 Projekt **Dispatcher** - diagram tříd 57
A.2 Projekt **TrippSystem** - diagram tříd 58
A.3 Projekt **TrippSystem** - diagram grafových tříd 59
A.4 Aplikace **Measuring** - diagram tříd 60
A.5 Testovací aplikace **GalaxyWars** - hlavní menu 61
A.6 Testovací aplikace **GalaxyWars** - nastavení 62
A.7 Testovací aplikace **GalaxyWars** - ovládání 63
A.8 Testovací aplikace **GalaxyWars** - power diagram 1 64
A.9 Testovací aplikace **GalaxyWars** - power diagram 2 65

Seznam tabulek

4.1 Urychlení *gaps filling* algoritmu pro různé změny ohodnocení 27

6.1 **TrippSystem** a **Dispatcher** - předzpracování a registrace 41
6.2 **TrippSystem** a **Dispatcher** - předzpracování a registrace 41
6.3 Algoritmus *gaps filling* - urychlení a kvalita nalezené cesty 42
6.4 **TrippSys** a **Dispatcher** - charakteristiky cest pro překážky různých velikostí . 46

B.1 Ovládání aplikace **GalaxyWars** 67
B.2 Ovládání aplikace **Measuring** 68

Seznam algoritmů

2.1 Konstrukce minimální kostry 9
2.2 Prototyp algoritmu hledání cest 10
2.3 Floyd-Warshallův *all-pairs shortest path* algoritmus 11
2.4 Algoritmus prohledávání do šířky 12
4.1 *Gaps filling* algoritmus (úprava nalezené cesty) 26

Kapitola 1

Úvod

Hledání cest je jednou ze základních a také nejznámějších problematik v oblasti teorie grafů. Většina algoritmů řešících tento problém nachází uplatnění v širokém spektru vědeckých disciplín a oblastí, např. v počítačových sítích, teorii optimalizace nebo při návrhu datových struktur. S rostoucí náročností moderních aplikací se však objevují problémy, které často velmi omezují a v některých případech dokonce znemožňují použití konvenčních algoritmů. Tyto algoritmy obvykle předpokládají, že prostředí, ve kterém pracují, je statické a předem známé, ale v případě virtuální reality nebo moderních počítačových her jsou tyto vlastnosti spíše výjimkou. Pro takové typy aplikací je potom nutné modifikovat konvenční metody, případně definovat zcela nové přístupy.

1.1 Cíl projektu

Projekt řešený v rámci této diplomové práce navazuje na autorovu bakalářskou práci [Bro06a], která se zabývá obecným hledáním cest v dynamickém, případně předem neznámém prostředí. Nabídka moderních aplikací vyžadujících nalezení cesty v takových podmínkách je ale velmi rozmanitá a pro získání efektivního řešení je nutné tuto obecnou definici upřesnit. Tento projekt se zaměřuje na hledání cest v počítačových hrách, konkrétně na navigaci umělé inteligence (protivníci, pomocníci, autopiloti apod.), z čehož plynou následující požadavky a doporučení:

- Prostředí v počítačových hrách může být statické (terén, budovy) i dynamické (herní postavy, vozidla). Systém pro hledání cest by měl tyto skupiny oddělovat pro získání větší efektivity (statické prostředí může být předzpracováno), ale zároveň umožnit změnu statických překážek na dynamické a opačně.

- Kromě dělení překážek na statické a dynamické musí systém předpokládat také možnost objevení nové překážky nebo zrušení některé existující.

- Pohyb dynamických překážek může být deterministický (např. letící asteroidy), ale také nedeterministický (např. protivníci v síťové hře). Také zde by měl systém předpokládat a oddělovat obě varianty.

- V případě počítačových her je často zbytečné vyžadovat optimální řešení. V umělé inteligenci je naopak doporučováno použití tzv. *tolerate imperfection* [AI Depot], tedy "simulace nedokonalosti", např. u počítačových protivníků. Navrhovaný systém by tedy

měl (opět z hlediska efektivity) dát přednost rychlosti před optimalitou výsledku. Navíc si lze představit případy, kdy použití *non-optimal* přístupu nijak neovlivňuje dojem ze získaného řešení, např. ve hrách při hledání cest pro avatary, které hráč v danou chvíli nevidí.

Cílem projektu je tedy vývoj systému[1] pro hledání cest (tzv. *path planning engine*), který bude sloužit k pseudooptimální navigaci v prostoru s dynamickými i statickými překážkami, přičemž překážky v prostoru mohou přibývat nebo ubývat a u dynamických překážek se předpokládá deterministický i nedeterministický pohyb. Systém samotný bude zajišťovat vytvoření a správu určité vnitřní reprezentace prostředí pro hledání cest, samotné nalezení cesty a navigaci po této cestě s případnou zpětnou vazbou.

1.2 Předchozí výzkum

Původní systém vyvíjený v rámci bakalářské práce [Bro06a] navazoval na projekt pro plánování cest v dynamickém, předem neznámém 2D prostředí ([AG05], viz obrázek 1.1), který byl založen na dvou základních strukturách:

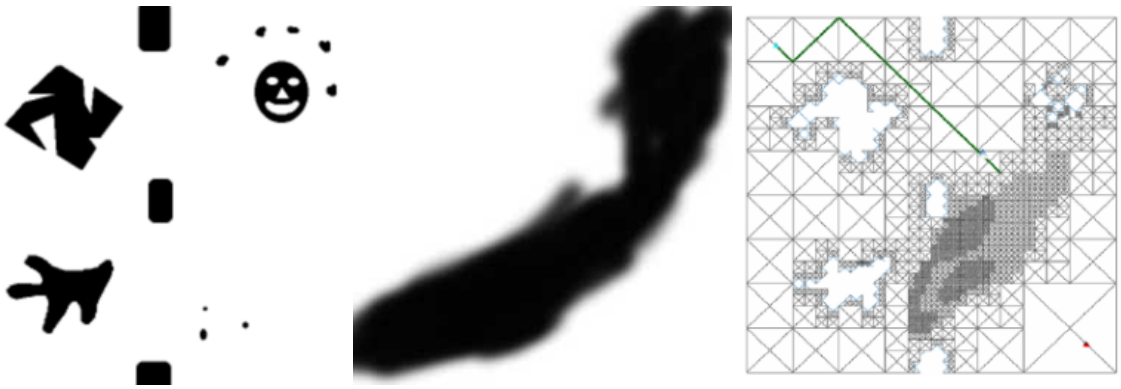
- 2D matice uchovávající hodnoty tzv. potenciálového pole překážek, tedy matice, ve které hodnota každé buňky odpovídá vzdálenosti této buňky od nejbližší překážky. Tyto hodnoty jsou získány pomocí techniky *image distance transform* (IDT) popsané v [AG05].
- Graf podobný struktuře *quadtree*, který se dynamicky adaptuje nad maticí potenciálového pole podle jejích hodnot. Konkrétně se zjemňuje v místech s vyšším potenciálem a naopak zjednodušuje v místech s nižším potenciálem. Hrany tohoto grafu pak slouží k nalezení samotné cesty.

Ve spolupráci s autory projektu (Dr. Marina Gavrilova[2] , Ing. Russel Ahmed Apu[3]) z University of Calgary v Kanadě bylo implementováno rozšíření do 3D s využitím adaptivní 3D struktury (viz obr. 1.2), kterou připravil Ing. Přemysl Zítka [Zit06] v rámci diplomové práce na ZČU. První výsledky této spolupráce pak byly prezentovány na Central European Conference on Computer Graphics [Bro06b] (viz příloha D). Jak se ale ukázalo při dalším výzkumu, mapování překážek do 3D matice potenciálového pole je příliš časově náročné na to, aby mohla být tato operace prováděna za běhu při každém pohybu překážky. Bližší informace o návrhu, implementaci a výsledcích tohoto přístupu jsou uvedeny v kapitole 3.

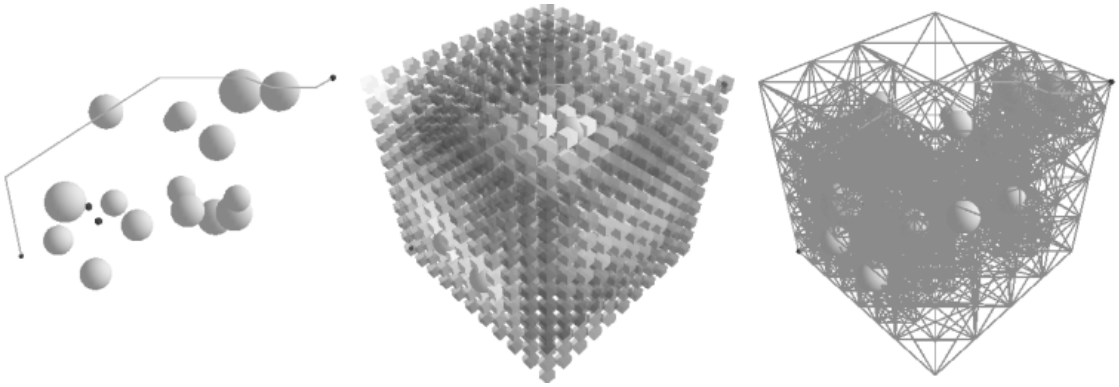
[1] V následujících kapitolách se pod pojmem *systém (hledání cest)* vždy rozumí skupina metod a datových struktur pro hledání cest.

[2] marina@cpsc.ucalgary.ca

[3] raapu@ucalgary.ca



Obrázek 1.1: Ukázka původního projektu [AG05] pro hledání cest ve 2D
(vlevo mapa překážek, uprostřed mapa hrozeb, vpravo výsledný adaptivní graf)



Obrázek 1.2: Ukázka projektu bakalářské práce [Bro06a]
(vlevo ukázková scéna, uprostřed potenciálové pole, vpravo výsledný adaptivní graf)

1.3 Zvolené řešení

Kvůli zmíněným nedostatkům původního řešení bylo nutné zvolit pro náš systém jiný způsob dělení prostoru pro měnící se data. Vzhledem k požadavkům uvedeným v podkapitole 1.1 byla jako nejvhodnější řešení zvolena regulární triangulace, kterou ve své diplomové práci [Zem07] implementoval a analyzoval Ing. Michal Zemek[4] . Triangulace v počítačové geometrii slouží, velmi zjednodušeně řečeno, k nalezení nejbližších dvojic ve vstupní množině bodů (v rovině, v prostoru). Regulární triangulace, zobecnění tzv. Delaunayovy triangulace, navíc uvažuje váhu vstupních bodů. Tato vlastnost je v projektu této diplomové práce využita pro reprezentaci velikosti, případně míry nebezpečí překážek, mezi kterými se hledá bezkolizní a "bezpečná" cesta. Adaptace na změny ve scéně je pak zajištěna přidáváním/ubíráním bodů triangulace a vlastním algoritmem pro přeplánování již nalezené cesty při změně ve vstupním grafu [Bro07]. Detailní popis návrhu, implementace a výsledků analýzy tohoto řešení lze najít v kapitolách 4, 5 a 6.

1.4 Organizace textu

Text je rozdělen do 7 kapitol včetně této úvodní. Kapitola 2 je věnována teorii hledání cest, definuje základní pojmy, poskytuje přehled konvenčních algoritmů pro řešení tohoto problému a představuje některé moderní přístupy k hledání cest v dynamickém prostředí. V kapitole 3 je stručně popsán původní systém pro hledání cest spolu s důvody, pro které byl tento přístup shledán jako neefektivní. Kapitola 4 potom představuje nové řešení a definuje použité datové struktury a algoritmy, jejichž implementace je následně popsána v kapitole 5. Kapitola 6 je věnována analýze zvoleného řešení a jeho implementace, představuje metody testování a získané výsledky. Kapitola 7 nakonec shrnuje tyto výsledky, splnění vytčených cílů a naznačuje další možné směry vývoje v této problematice. Závěr dokumentu pak nabízí přehled použitých zkratek, použitou literaturu a rejstříky. V příloze jsou nakonec k dispozici ukázky testovací aplikace, uživatelská a programátorská dokumentace a autorovy články z konferencí Central European Conference on Computer Graphics, Spring Conference of Computer Graphics a ze soutěže ACM Students Research Competition[5].

[4] mzemek@kiv.zcu.cz

[5] Práce získala v roce 2006 3. místo v česko-slovenském finále soutěže.

Kapitola 2

Teorie

2.1 Definice problému

Plánování cest (*path planning*) v našem kontextu definuje problém nalezení optimální cesty mezi dvěma konkrétními místy v určité abstraktní reprezentaci prostředí. Pojem hledání cest (*path finding*) pak bývá v literatuře s touto problematikou někdy interpretován jako provedení pohybu po naplánované cestě se zpětnou vazbou pro plánovací modul, tzv. *path planner*. V našem případě označuje plánování i hledání cest obecně celou problematiku a význam těchto pojmů se tedy nerozlišuje. Pod pojmem *avatar* pak rozumíme konkrétní entitu, která provádí naplánovaný pohyb (např. protivník v počítačové hře).

Každý z algoritmů pro hledání cest je založen na určité abstraktní reprezentaci prostředí, ve kterém pracuje. Typů reprezentace je několik a my si je popíšeme v sekci 2.3 této kapitoly. Na každou takovou reprezentaci lze ale obecně nahlížet jako na určitý graf $G(V, E)$, kde V je konečná množina uzlů odpovídajících určitým stavům a E je množina hran spojujících uzly grafu. Hodnoty $n = |V|$, resp. $m = |E|$ pak představují počet uzlů, resp. hran. Hrany grafu navíc mohou být orientované nebo neorientované a potom platí

$$E \subset \binom{V}{2} \text{ pro neorientované grafy}$$

$$E \subset V^2 \text{ pro orientované grafy}$$

Kromě takto definované struktury se dále často popisují vlastnosti jednotlivých elementů grafu, nejčastěji reálnou funkcí w tak, že

$$w : V \to \mathbb{R} \text{ pro uzlově ohodnocené grafy}$$

$$w : E \to \mathbb{R} \text{ pro hranově ohodnocené grafy}$$

V problematice plánování cest se obvykle uvažují pouze hranově ohodnocené grafy. Ty lze navíc převést na uzlově ohodnocené grafy a naopak. Mezi další důležité vlastnosti grafů (podle [AI Depot]) pak patří:

Hustota propojení reprezentuje množství hran v grafu. Jde o velice důležitou vlastnost především z hlediska složitosti algoritmů, která může být $O(m)$ nebo např. $O(m^2)$. Grafy s vysokým počtem hran bývají označovány jako husté (*dense graphs*), zatímco grafy s nízkým počtem hran jako řídké (*sparse graphs*).

Homogenita ohodnocení určuje, zda mají hrany přibližně stejnou váhu. Rovnoměrné rozdělení vah je v cizojazyčné literatuře označováno jako *homogeneous edge costs*, naopak nerovnoměrné rozdělení vah je označováno jako *irregular edge costs*.

Logická/náhodná propojení označují, zda jsou uzly grafu propojeny v určité logické struktuře nebo čistě náhodně. Také tato vlastnost je důležitým faktorem výsledného chování algoritmu a je dobré vzít ji v úvahu při testování.

2.2 Klasifikace algoritmů

Problém hledání cest je zkoumán od samotného počátku teorie grafů a pro jeho řešení dnes existuje nepřeberné množství algoritmů a technik, které je potřeba určitým způsobem klasifikovat. Následující seznam představuje nejvýznamnější charakteristiky a požadavky, podle kterých lze algoritmy pro plánování cest rozlišovat.

- Nejdůležitějším faktorem pro klasifikaci je **rozsah hledaných řešení**, jinými slovy to, zda má konkrétní algoritmus sloužit k nalezení jediné cesty mezi dvěma danými uzly (*single-pair* algoritmy), k nalezení všech cest z konkrétního uzlu (*single-source* algoritmy) nebo k nalezení cest mezi všemi uzly grafu (*all-pairs shortest path*).
- Druhým nejdůležitějším prvkem pro klasifikaci algoritmů hledání cest jsou **vlastnosti prostředí**, ve kterém tyto algoritmy pracují. Mezi tyto vlastnosti patří reprezentace prostředí, jeho předběžná znalost nebo zda jde o statické, resp. dynamické prostředí. Jelikož je prostředí nejčastěji reprezentováno grafem, rozlišují se dále **vlastnosti grafů**, např. možnost záporného ohodnocení, hustota apod.
- Z hlediska avatara, který bude navigován konkrétním systémem, je nutné zvážit, zda jde o **avatara zanedbatelné velikosti** nebo zda je nutné uvažovat jeho rozměry.
- Především v případě aplikací, jejichž prostředí se může průběžně měnit, je důležité rozhodnout, zda je úkolem algoritmu **nalezení kompletní cesty** nebo jen její části.
- Je dobré zvážit, zda bude připravovaný systém sloužit k nalezení cesty pro jediného avatara nebo k **současné navigaci více avatarů**, např. u strategických her. Takové typy systémů jsou obvykle označovány pojmem *multi-agent path planning*.
- Z hlediska aplikací, které dávají přednost rychlosti před kvalitou výsledného řešení, mohou být použité algoritmy rozděleny podle toho, zda slouží k nalezení **optimální nebo suboptimální cesty** (*t-optimal path planning*).

Jak již bylo zmíněno v úvodní kapitole, základním problémem při použití konvenčních přístupů pro hledání cest v moderních aplikacích je fakt, že jsou tyto metody navrženy pro statické, neměnné prostředí. Algoritmy popisované ve zbytku této kapitoly jsou proto rozděleny na algoritmy pracující ve statickém (sekce 2.4) a dynamickém prostředí (sekce 2.5).

2.3 Reprezentace prostředí

Abstraktní reprezentace prostředí, nad kterou pracují algoritmy hledání cest, je nejčastěji klasifikována do následujících tříd:

Grafová reprezentace popisuje dané prostředí konečnou množinou stavů, mezi kterými se lze pohybovat po definovaných hranách (viz obr. 2.1). V tomto případě odpovídají stavy diskrétním bodům v n-rozměrném prostoru (v našem případě jsou to nejčastěji polohové souřadnice ve 3D).

Rastrová reprezentace definuje prostředí konečnou množinou hodnot v matici, která svou dimenzí odpovídá dimenzi reprezentovaného prostoru. V takovéto diskrétní reprezentaci pak algoritmy hledání cest postupují po jednotlivých buňkách matice (viz obr. 2.2).

Virtuální realita ani počítačové hry obvykle neposkytují reprezentaci prostředí, která by okamžitě umožnila použití některého z algoritmů hledání cest. Obecně lze předpokládat, že k dispozici je pouze seznam překážek, případně definice jejich tvaru. Strukturu pro hledání cest je pak nutné určitým způsobem získat právě z těchto informací.

V případě grafové reprezentace jsou často používány tzv. grafy viditelnosti (*visibility graphs*). Množina uzlů grafu viditelnosti odpovídá vrcholům jednotlivých překážek a hranami jsou pak spojeny ty uzly, které buď leží na jedné hraně překážky nebo se vzájemně "vidí", tedy pokud přímka spojující tyto uzly neprochází žádnou z překážek. Příklad grafu viditelnosti lze najít na obr. 2.1. Pro upřesnění dodejme, že mezi vrcholy překážek jsou obvykle zařazeny také body, mezi kterými má být nalezena cesta (na obr. 2.1 označeny jako s a t). Kromě grafů viditelnosti se dále používají různé struktury pro dělení prostoru [Sam90], například tzv. *quadtrees*, *octrees*, *kD-trees*, *BSP trees* nebo různé triangulace[1].

U rastrových reprezentací existuje několik způsobů, podle kterých lze interpretovat hodnoty v dané matici. Nejjednoduší variantou je použití logických hodnot určujících, zda se v oblasti odpovídající dané buňce v matici vyskytuje některá překážka. V takové matici pak lze jednoduše najít bezkolizní cestu mezi danými body procházením buněk s hodnotou $false$. Mnohem častěji se však využívá tzv. IDT[2] technika, která v buňkách matice uchovává vzdálenost středu této buňky od nejbližší překážky. Vytváří tak určité potenciálové pole, ve kterém lze procházením buněk s největšími hodnotami hledat "nejbezpečnější" cestu mezi překážkami.

2.4 Statický path-planning

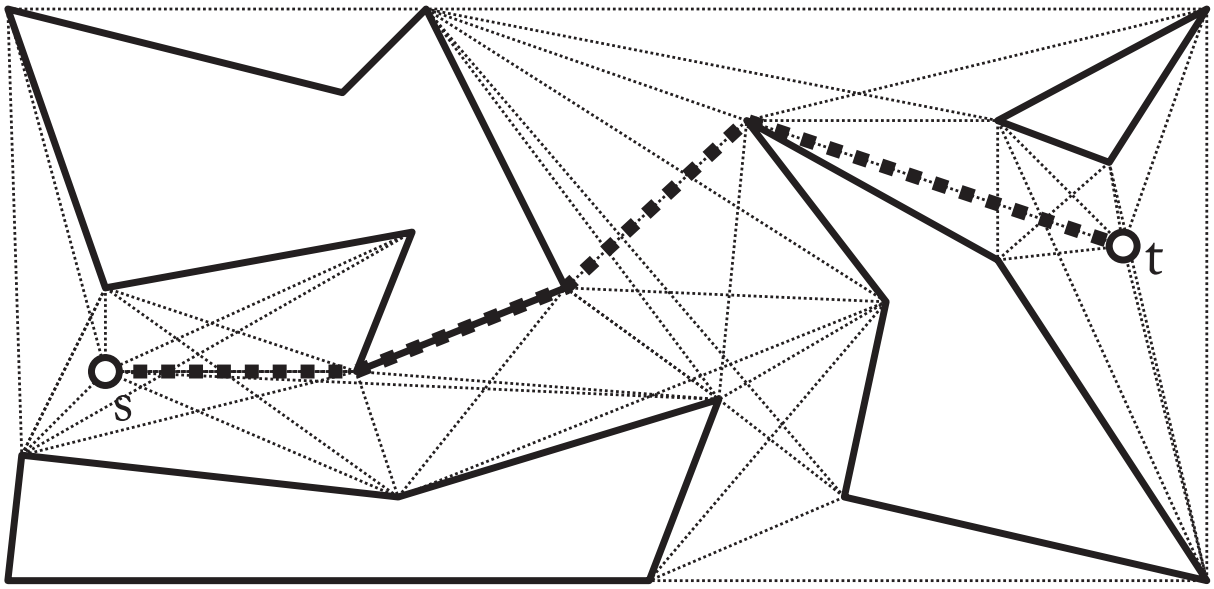
2.4.1 *Single source* algoritmy

Single source algoritmy slouží k nalezení optimálních cest z daného výchozího uzlu do všech ostatních uzlů grafu. Takové řešení je nejčastěji reprezentováno tzv. minimální kostrou (*minimum spanning tree* nebo MST) grafu, která obsahuje veškeré uzly grafu a podmnožinu hran potřebných pro všechny cesty (viz obrázek 2.3). Optimální cestu do konkrétního uzlu grafu lze určit průchodem této stromové struktury směrem ke kořeni. Základním přístupem[3] pro nalezení MST je tzv. relaxace [AI Depot], která prochází hrany grafu a testuje vzdálenost jejich koncových uzlů od určitého výchozího uzlu. Pokud do koncového uzlu vede kratší cesta právě přes tuto hranu, je hrana označena jako součást kostry. Algoritmus končí ve chvíli, kdy nelze najít kratší cestu do žádného z vrcholů grafu. Postup konstrukce je představen v pseudokódu 2.1.

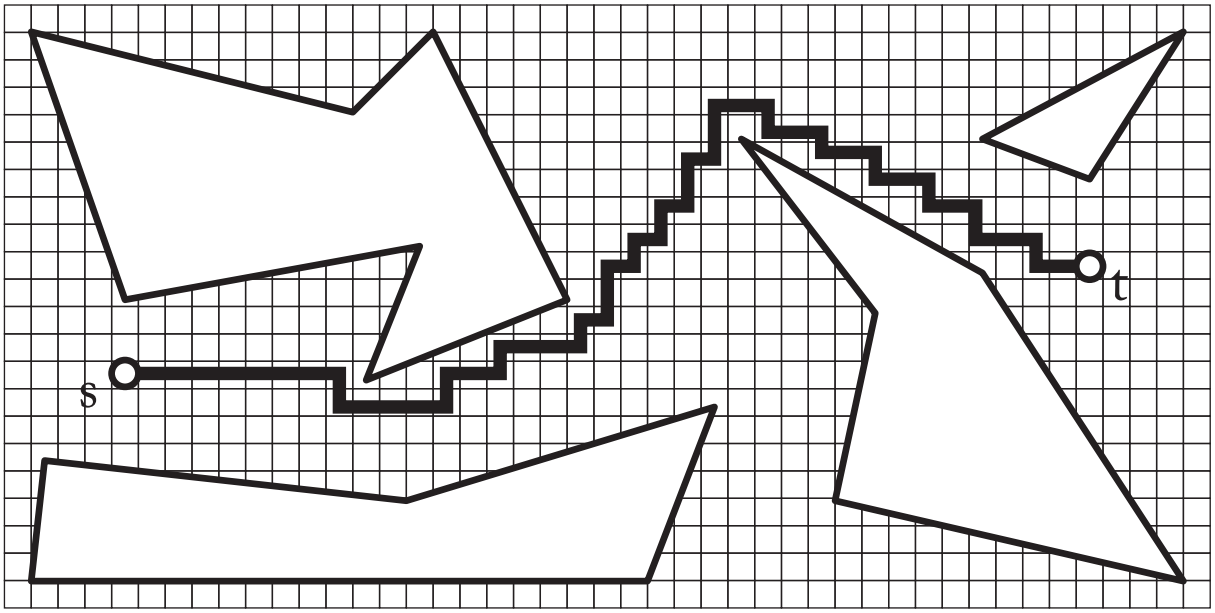
[1] O triangulacích budeme blíže hovořit v sekci 4.2.

[2] Bližší informace o této technice jsou uvedeny v sekci 3.1.

[3] Z dalších metod konstrukce MST uveďme pro zajímavost tzv. Primův algoritmus [Ski97].



Obrázek 2.1: Graf viditelnosti

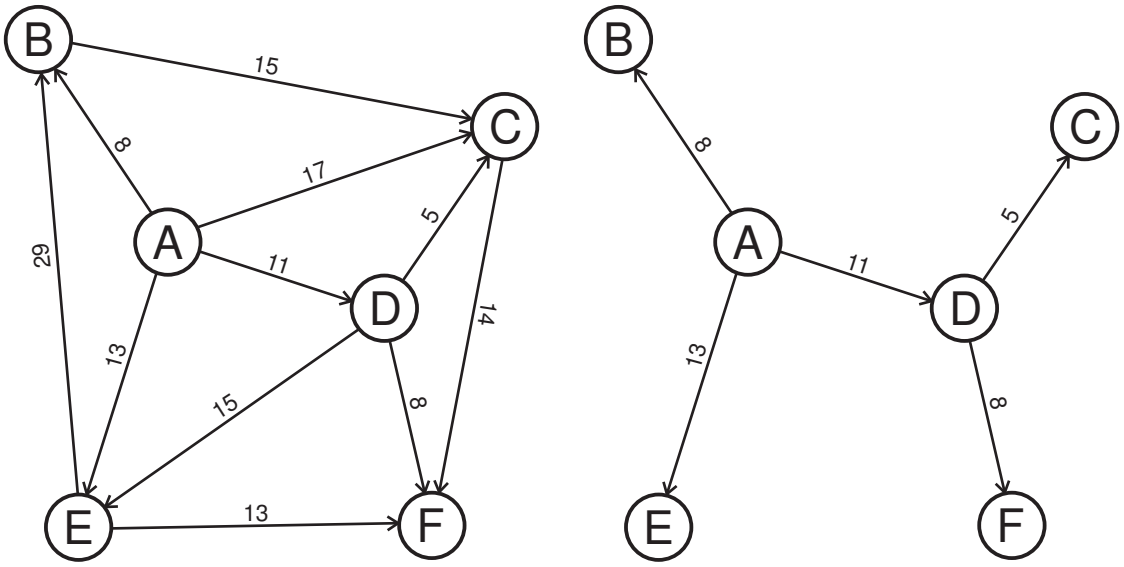


Obrázek 2.2: Hledání cest v rastru

Algoritmus 2.1 Konstrukce minimální kostry

Vstup: *graph* obsahující konečnou množinu uzlů a hran
Vstup: *root* je jedním z uzlů grafu *graph*

```
root.parent ← NULL
root.distance ← 0
for all node in graph.nodes do
  node.parent ← NULL
  node.distance ← inf
end for
while graph není kostra do
  for all edge in graph.edges do
    dist ← edge.start.distance + edge.weight
    if edge.end.distance > dist then
      edge.end.parent ← edge.start
      edge.end.distance ← dist
    end if
  end for
end while
```



Obrázek 2.3: Ukázka minimální kostry v orientovaném, hranově ohodnoceném grafu (vlevo původní graf, vpravo minimální kostra pro kořenový uzel A)

Ačkoliv je časová složitost této metody $O(2^m)$, algoritmy s nižší složitostí jsou založeny právě na konstrukci MST. Profesor Edsger W. Dijkstra v roce 1956 formalizoval problém nalezení optimální cesty v grafu a představil jeho řešení (pro grafy s kladným hranovým ohodnocením), které bylo v roce 1986 zobecněno v tzv. prototypu algoritmu hledání cest. Ten využívá tzv. seznam otevřených uzlů, tedy uzlů určených k dalšímu zpracování. Na začátku výpočtu je do seznamu vložen počáteční uzel hledané cesty. Dokud jsou v seznamu uzly, algoritmus vybírá jeden z nich (odebírá jej ze seznamu) a pro všechny jeho sousední uzly testuje, zda do nich nevede kratší cesta právě přes něj. Pseudokód výpočtu je uveden v algoritmu 2.2, realizace metod `Insert` a `Select` je prostorem pro úpravy v konkrétních algoritmech.

Algoritmus 2.2 Prototyp algoritmu hledání cest

Vstup: *graph* obsahuje kon. množinu uzlů a hran
Vstup: *root* je jedním z uzlů grafu *graph*
Vstup: *list* je prázdný seznam uzlů

```
list.Insert(root)
while list není prázdný do
  node ← list.Select() {Vybraný uzel je ze seznamu odstraněn}
  for all edge in graph.edges do
    if edge.end.distance > dist then
      edge.end.parent ← edge.start
      edge.end.distance ← dist
      list.Insert(edge.end)
    end if
  end for
end while
```

Dijkstrův algoritmus metodu `Insert` vůbec nevyžaduje a k výběru následujícího uzlu ke zpracování využívá tzv. *greedy* přístup - vždy je vybrán uzel, který je nejblíže počátečnímu uzlu. Pokud je metoda pro výběr nejbližšího uzlu realizována lineárním prohledáváním, je její složitost $O(n)$ a složitost celého algoritmu potom $O(n^2)$, proto byly dále představeny datové struktury pro snížení složitosti metody `Select`, např. *d-Heaps* [Epp94]. V případě grafů, kde může být ohodnocení hran také záporné, lze použít Bellman-Fordův algoritmus [Ski97] s časovou složitostí $O(m.n)$, která se však u grafů s vysokou hustotou propojení uzlů blíží $O(n^3)$.

2.4.2 *All-pairs shortest path* algoritmy

Jak je patrné z názvu, *all-pairs shortest path* algoritmy hledají optimální cesty mezi všemi uzly grafu. Narozdíl od předchozí skupiny algoritmů by však byla tvorba n minimálních koster velmi paměťově náročná, proto se v tomto případě často hovoří o tzv. *all-pairs shortest distance* algoritmech, které poskytují pouze vzdálenosti optimálních cest mezi jednotlivými uzly. Tyto hodnoty jsou uchovávány v matici řádu n, u neorientovaných grafů je pak možné využít symetrie a uchovávat pouze polovinu hodnot.

Naivní implementaci pro tento typ algoritmů je možné získat aplikací Dijkstrova algoritmu, resp. Bellman-Fordova algoritmu na každý uzel grafu s výslednou časovou složitostí $O(n^3)$, resp. $O(n^4)$. Nejznámějším algoritmem v této třídě je pak Floyd-Warshallův algoritmus [Ski97], který postupně upravuje hodnoty tzv. distanční matice (řádky/sloupce představují

jednotlivé uzly a hodnoty v matici pak vzdálenost mezi nimi; vzdálenost mezi uzly u a v označme $D(u, v)$) tak, aby platilo

$$D(u, v) = min(D(u, v), D(u, w) + D(w, v)), \forall w \in V$$

V pseudokódu 2.3 lze vidět, že Floyd-Warshallův algoritmus pracuje se složitostí $O(n^3)$. Tento postup lze navíc použít také u grafů se záporným ohodnocením hran (ovšem bez záporných cyklů).

Algoritmus 2.3 Floyd-Warshallův *all-pairs shortest path* algoritmus

```
Vstup: d distanční matice grafu
Vstup: graph obsahující konečnou množinu uzlů a hran
  for i = 0 to graph.nodes.count do
    d[i][i] ← 0
  end for
  for all edge in graph.edges do
    d[edge.start][edge.end] ← edge.weight
  end for
  for k = 0 to graph.nodes.count do
    for i = 0 to graph.nodes.count do
      for j = 0 to graph.nodes.count do
        d[i][j] ← min(d[i][j], d[i][k] + d[k][j])
      end for
    end for
  end for
```

Prolomení kubické složitosti u algoritmů pro nalezení cest mezi všemi uzly grafu se ukázalo být velkým problémem. V roce 1977 byl představen algoritmus, který kombinuje Dijkstrův a Bellman-Fordův algoritmus a za předpokladu řídkého orientovaného grafu bez cyklů pracuje se složitostí $O(n^2 \log n + m.n)$. Bez těchto předpokladů ale může být kubická složitost prolomena pouze aproximací nejkratší cesty. Aproximace nejkratší cesty je označována pojmem t-optimální cesta, který byl definován v sekci 2.2. Taková cesta může být nanejvýš t-krát horší než cesta optimální, např. t-krát delší. V roce 1997 např. představili Dor, Halperin a Zwick efektivní algoritmus pro nalezení 3-optimální cesty [DHZ00].

2.4.3 *Single pair shortest path* algoritmy

Základní variantou path planningu jsou *single pair* algoritmy, jejichž úkolem je nalezení optimální cesty mezi dvěma danými uzly. Výsledné řešení pak může být reprezentováno buď posloupností hran nebo uzlů. Nejznámějšími algoritmy v této třídě [Ski97] jsou prohledávání do hloubky (*depth first search* nebo jen *DFS*) a prohledávání do šířky (*breadth first search*, *BFS*), které má velmi blízko k *single source* algoritmům. Z teoretického hlediska jsou *single source* algoritmy a *single pair* algoritmy stejně časově náročné, avšak v praxi jsou hledané cesty často přímé nebo alespoň částečně přímé a *single pair* algoritmy jsou proto efektivnější. Pseudokód 2.4 naznačuje algoritmus prohledávání do šířky. Při použití zásobníku namísto fronty by pak stejný algoritmus realizoval prohledávání do hloubky.

Algoritmus A* [AI Depot] je založen právě na předpokladu, že je hledaná cesta alespoň částečně přímá, a využívá heuristiku pro "podhodnocování" uzlů vybíraných k dalšími zpracování. Heuristickou funkcí je v tomto případě euklidovská vzdálenost od cílového uzlu. Uzel, který je blíže cíli, má proto větší šanci být vybrán k dalšímu zpracování. Zobecněním algoritmu A* je potom tzv. *best-first search* algoritmus, avšak pro obecnou aplikaci je těžké najít lepší heuristickou funkci.

Algoritmus 2.4 Algoritmus prohledávání do šířky

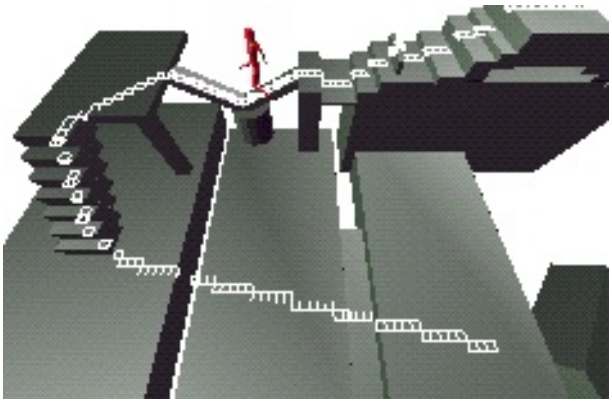
```
Vstup: source je počáteční uzel
Vstup: target je koncový uzel
Vstup: queue je prázdná fronta uzlů
  queue.Enqueue(target)
  while queue není prázdná do
    node ← queue.Dequeue()
    for all neighbour in node.neighbours do
      if neighbour is source then
        return Cesta nalezena
      else if neighbour.visited == false then
        queue.Enqueue(neighbour)
        neighbour.visited ← true
      end if
    end for
  end while
  return Cesta neexistuje
```

2.5 Dynamický path-planning

Algoritmus D* [Ste94], modifikace algoritmu A*, je navržen pro grafy, u kterých může dojít ke změně ohodnocení hran grafu v době jeho procházení. Je tedy určen pro neznámá, částečně známá nebo měnící se prostředí. Podobně jako u popisovaných single-source algoritmů si D* také uchovává seznam právě zpracovávaných uzlů a v tomto seznamu dále distribuuje informace o změnách ohodnocení jednotlivých hran. Každý ze zpracovávaných uzlů uchovává informaci o nejmenší odhadované vzdálenosti do cílového uzlu, tzv. *key function*, podle které jsou uzly řazeny. Při změně ohodnocení některé z hran je pak tato změna propagována do všech příslušných "klíčových funkcí", následkem je přeuspořádání uzlů, které mají být dále zpracovány. Algoritmus poskytuje optimální řešení, funkčně je shodný s *brute-force* přístupem, který při každé změně prostředí hledá celou cestu znovu, je však efektivnější.

Systém [ACF01] je dobrým příkladem přístupu pro strategické hry, kde je obvykle nutné plánování cest pro velké počty avatarů. Odděluje statické překážky, stojící avatary a pohybující se avatary. V rámci preprocessingu je pro statické překážky připraven graf viditelnosti (viz obr. 2.1), který nabízí pevný základ pro hledání cest. Pozastavené entity jsou sloučeny do shluků a cesta nalezená v grafu viditelnosti je pak upravena tak, aby s těmito entitami nebo shluky nekolidovala. V každém kroku provádění cesty jsou pak vzájemně sledovány pohybující se entity a v případě, kdy by mezi nimi mělo dojít ke kolizi, je náhodně zvolená entita dočasně pozastavena, dokud jí opět není uvolněna cesta.

Technika popsaná v [BT98] používá metodu rasterizace, která byla popsána v sekci 2.3. Scéna je diskretizována do matice buněk jednotné velikosti (tzv. *uniform cells*) a v nich je následně hledána optimální cesta s použitím již známého A* algoritmu. Jelikož jde o cestu pro "lidské" avatary, základním kritériem výsledné cesty je, aby vedla po povrchu překážek dosažitelných člověkem a ne "vzduchem". Ukázka takové cesty je znázorněna na obrázku 2.4.



Obrázek 2.4: Cesta pro "lidské" avatary (převzato z [BT98])

Kapitola 3

Původní řešení

Jak již bylo uvedeno v podkapitole 2.1, existuje mnoho způsobů, jak klasifikovat algoritmy pro hledání cest. Jednou z nejvýznamnějších vlastností je druh reprezentace prostředí, nad kterým tyto algoritmy pracují. Z tohoto hlediska nejčastěji rozlišujeme algoritmy pracující s grafovou a rastrovou reprezentací. *Grafové* metody jsou velice rychlé při hledání cesty, avšak samotné generování grafu může být v případě moderních aplikací velmi komplikované, v některých případech dokonce nemožné. Jako příklad uvažme aplikaci, kde je úkolem nalezení nejkratší cesty mezi dvěma místy v prostředí, které není předem známé. Naopak *rastrové* algoritmy se s dynamickým nebo předem neznámým prostředím vypořádají mnohem lépe, ale hledání cesty je kvůli obvykle velkému počtu prvků rastru (2D nebo 3D matice) velmi časově náročné.

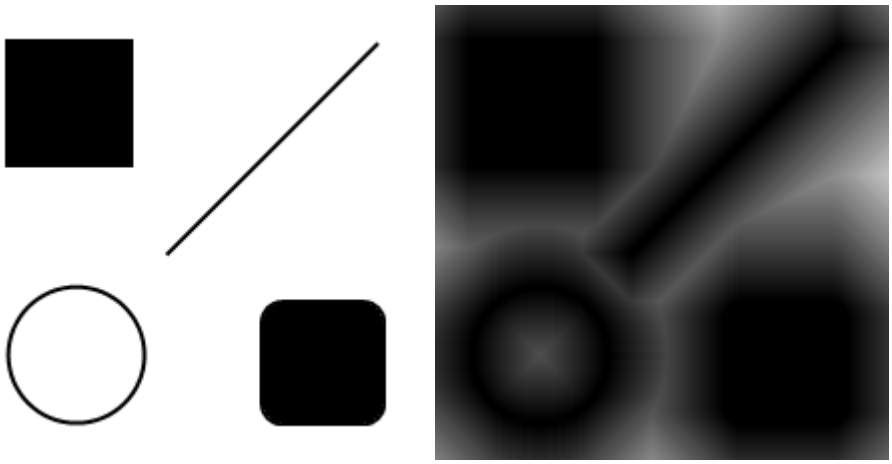
Původní řešení předcházející této práci navazuje na práci R. A. Apu a M. Gavrilové [AG05] a navrhuje suboptimální řešení kombinaci grafové a rastrové reprezentace prostředí. Přesněji řečeno, použitím 3D matice pro reprezentaci překážek a hrozeb spolu s grafem podobným struktuře pro dělení prostoru známé jako *octree*, který "obaluje" 3D matici a přizpůsobuje se v ní uloženým hodnotám - zjemňuje se v místech s vyšším nebezpečím a zjednodušuje v místech s menším nebezpečím. Výsledná kombinace pak může být díky 3D matici použita v dynamickém, předem neznámém prostředí a použitý graf navíc zajišťuje rychlou odezvu při samotném hledání cesty.

3.1 Návrh

Navrhovaný systém pro hledání cest sestává z následujících prvků:

Map trojrozměrná matice reálných hodnot reprezentujících prostředí. Reprezentace je zajištěna technikou zvanou *image distance transform* (dále jen IDT), kdy je do každé buňky matice uložena vzdálenost středu této buňky od nejbližší překážky. Matice pak obsahuje hodnoty "potenciálového pole" tvořeného překážkami (viz obrázek 3.1).

Mesh graf podobný struktuře *octree* složený z disjunktních kvádrových oblastí, tzv. *clusterů*. Každý cluster může být rekurzivně rozdělen na 8 clusterů pokrývajících ten samý prostor. V článku [AG05] je tato stromová struktura označena jako *adaptive spatial memory* nebo ASM.



Obrázek 3.1: Ukázka použití *image distance transform*
(Vlevo vstupní obraz s překážkami, vpravo jejich potenciálové pole)

Navrhovaný přístup používá dvě mapy stejné velikosti. První z nich, tzv. *mapa překážek*, slouží k reprezentaci statických překážek a druhá, tzv. *mapa hrozeb*, představuje pohyblivé překážky, které jsou v tomto kontextu označovány jako hrozby. Nad stejným prostorem je potom "rozprostřena" uvedená grafová struktura, která se v každé iteraci rekurzivně adaptuje podle hodnot uložených v obou mapách, přesněji řečeno dělí se na clustery v místech s větším nebezpečím (viz tmavé oblasti potenciálového pole na obrázku 3.1). Na vrcholy a hrany těchto clusterů lze nakonec nahlížet jako na uzly a hrany grafu, ve kterém lze hledat cestu do konkrétního místa v tomto mapovaném prostoru. Ohodnocení vrcholů a hran grafu je definováno hodnotami map v příslušných buňkách.

3.2 Implementace

Projekt byl pod označením **Dispatcher** (Discrete space path searcher) implementován jako dynamická knihovna v jazyce C# 2.0, kterou využívá testovací aplikace založená na grafické knihovně DirectX. Adaptivní grafovou strukturu připravil v rámci své diplomové práce [Zit06] Ing. Přemysl Zítka. Třídy a rozhraní popisovaného systému jsou znázorněny na obrázku A.1. Rozhraní `IMap` představuje obecnou definici mapy a slouží k tomu, aby uživatel mohl v navrhovaném systému používat vlastní třídy pro mapy implementující toto rozhraní. `IMap` vyžaduje pouze implementaci metod `float Weight(Point p)` (hodnota mapy v konkrétním místě) a `float Weight(Block b)` (hodnota mapy v konkrétní oblasti). Toto rozhraní implementují třídy `ObstaclesMap` a `ThreatsMap` odpovídající již zmíněným mapám pro překážky a hrozby. Ve druhé skupině jsou zařazeny třídy související s adaptivním grafem. Rozhraní `IGraph` opět definuje základní vlastnosti používaného grafu pro případné použití jiné než připravené třídy `Mesh`. `IGraph` vyžaduje jedinou metodu `IEnumerable Vertices()`, která vrací posloupnost všech vrcholů implementovaného grafu. Tyto vrcholy musí implementovat rozhraní `IEngram`, jinými slovy musí implementovat metodu `IEnumerable Descendants()` pro získání všech sousedů daného vrcholu a metodu (lépe řečeno *property*) `Point Position`, která vrací pozici vrcholu. V připravené aplikaci toto rozhraní implementuje třída `Engram`.

Všechny základní třídy jsou uživateli zpřístupněny prostřednictvím třídy `Dispatcher`, která nabízí veškeré potřebné operace - vložení/odebrání překážek nebo hrozeb, adaptaci na provedené změny, seznam všech hran grafu a nalezení cesty mezi danými místy v mapovaném prostoru. Adaptace na provedené změny je provedena za běhu testovací aplikace při každé změně některé z překážek. Během této operace je v případě potřeby přepočítána mapa hrozeb/překážek a podle nových hodnot v mapách dochází k adaptaci grafové struktury - strom clusterů je nejprve prohledán od listů ke kořeni (od nejmenších clusterů) a veškeré clustery, v jejichž prostoru není "příliš velké nebezpečí", jsou sloučeny do původního superclusteru. Následně je strom clusterů prohledán od kořene k listům přes neprouzkoumané uzly a každý cluster, v jehož prostoru je "dostatečně velké nebezpečí", je pak rozdělen na 8 clusterů.

3.3 Analýza a výsledky

Pro účely analýzy a testování navrženého systému hledání cest byla připravena jednoduchá aplikace v jazyce C# 2.0 s použitím knihoven DirectX. V této aplikaci je k dispozici scéna složená z určitého počtu překážek tvořených koulemi různých průměrů a náhodně se pohybujících bodů představujících hrozby. Za běhu aplikace jsou hrozby navigovány na různá místa scény a v každé iteraci programu je pak vyžádána adaptace systému, tedy přepočítání hodnot v mapě hrozeb a přizpůsobení grafu. Mapa překážek je připravena v rámci preprocessingu, složitost statických překážek tedy přímo neovlivňuje efektivitu systému za běhu programu.

V rámci analýzy systému byly sledovány následující vlastnosti:

Počet clusterů představuje celkový počet všech clusterů, jinými slovy počet všech uzlů stromu popisované adaptivní struktury.

Čas adaptace reprezentuje celkový čas potřebný k přepočítání mapy hrozeb a adaptaci grafu v jedné iteraci.

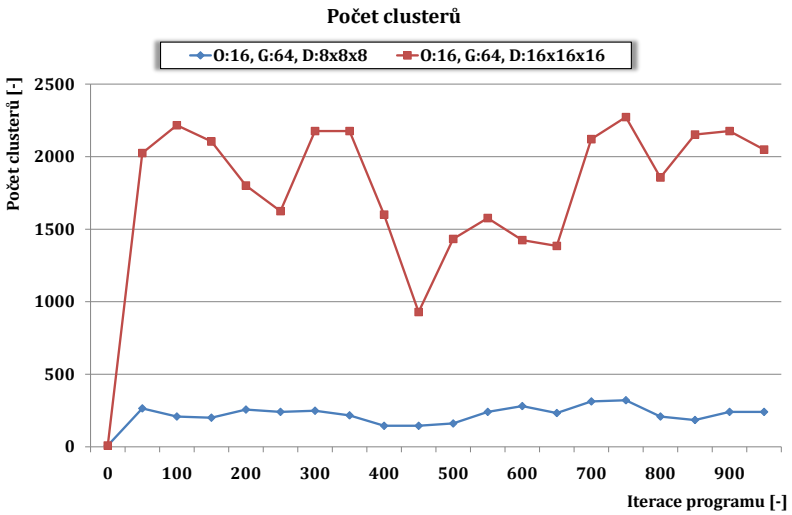
Nebezpečí cesty představuje nejmenší vzdálenost k některé z překážek vzhledem k jednotlivým uzlům naplánované cesty. Jinak řečeno, mezi všemi uzly cesty je vybrána jejich nejmenší vzdálenost od některé z překážek.

3.3.1 Standardní konfigurace

Pro představení některých vlastností systému bylo nejprve provedeno testování na jednoduché konfiguraci s 16 překážkami, kvalitou map 64 (tedy matice pro každou mapu obsahuje 64x64x64 hodnot) a adaptivním grafem, který může být rozložen až na 8x8x8 clusterů, resp. na 16x16x16 clusterů. Ve všech následujících grafech této kapitoly jsou hodnoty popisovány zkratkami O (obstacles) pro počet překážek, G (grid) pro řád matice a D (division) pro maximální zjemnění adaptivního grafu. Označení *O:16, G:64, D:8x8x8* pak definuje testovací scénu uvedenou na začátku tohoto odstavce.

Na obrázku 3.2 jsou znázorněny počty clusterů během prvních 1000 iterací programu. V prvních iteracích lze pozorovat prudký nárůst počtu clusterů způsobený počáteční adaptací grafu na statické objekty definované v mapě překážek. Následný kolísavý průběh u obou testovacích případů lze vysvětlit následujícím způsobem: pokud se některá z dynamických překážek (hrozeb) dostane do prostoru bez překážek, zvýší tím nebezpečí v původně bezbečné oblasti a evokuje tak zjemnění adaptivního grafu v této oblasti. Tyto případy lze v grafu pozorovat kolem iterací č. 100, 350 nebo 750. Obrázek 3.3 reprezentuje časy adaptace, opět

během prvních 1000 iterací testovacího programu. Je zřejmé, že časová náročnost adaptace je závislá na maximální úrovni zjemnění grafu a také kolísá v závislosti na pozici náhodně se pohybujících překážek. Poslední charakteristikou měřenou ve standartní konfiguraci je již zmíněné nebezpečí cesty znázorněné na obr. 3.4.

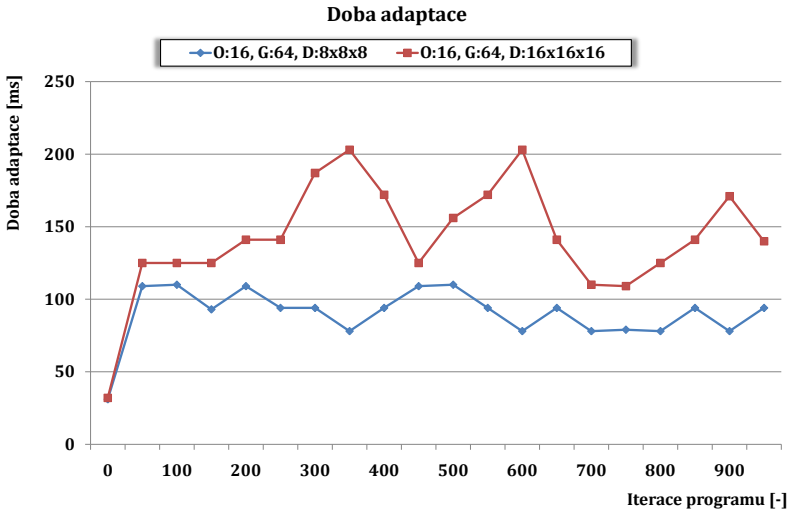


Obrázek 3.2: Původní řešení - počet clusterů u standardní konfigurace

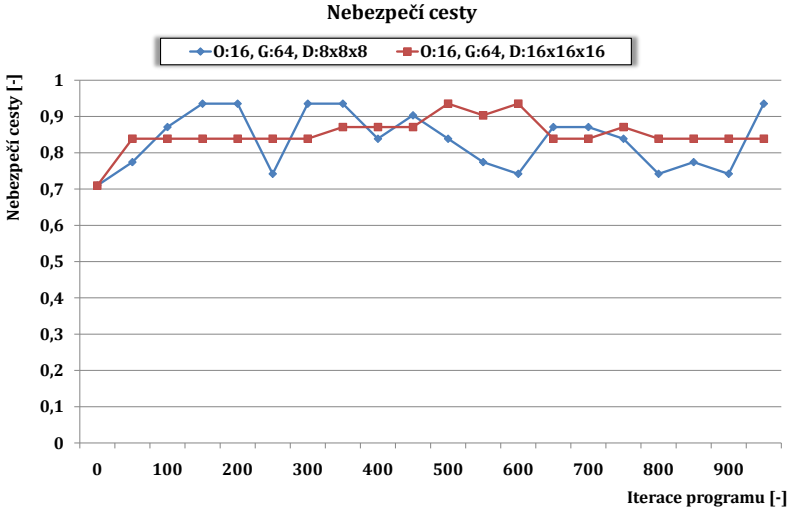
3.3.2 Vysoký počet překážek

Další naměřené hodnoty a grafy ukazují, že efektivita navrženého systému není přímo závislá na počtu statických překážek. Pro měření byly použity dvě testovací konfigurace s 256, resp. 2048 náhodně rozmístěnými překážkami. Obě konfigurace pak používají mapy řádu 64 a graf s maximálně 16x16x16 clustery.

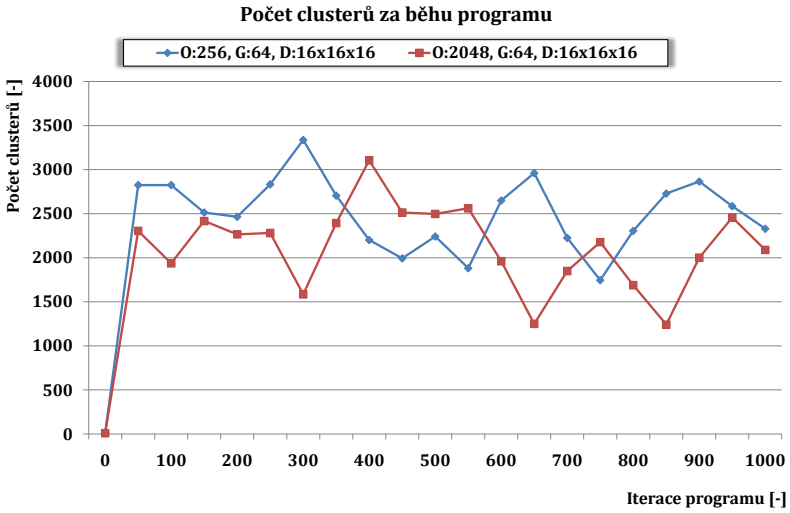
Grafy 3.5 a 3.6 naznačují vývoj počtu clusterů a časy adaptace v průběhu prvních 1000 iterací programu. Lze vidět, že ani jedna z těchto vlastností přímo nezávisí na počtu statických překážek, jelikož ty jsou v rámci předzpracování zaznamenány do mapy překážek, která má pevný počet hodnot, v našem případě 64x64x64.



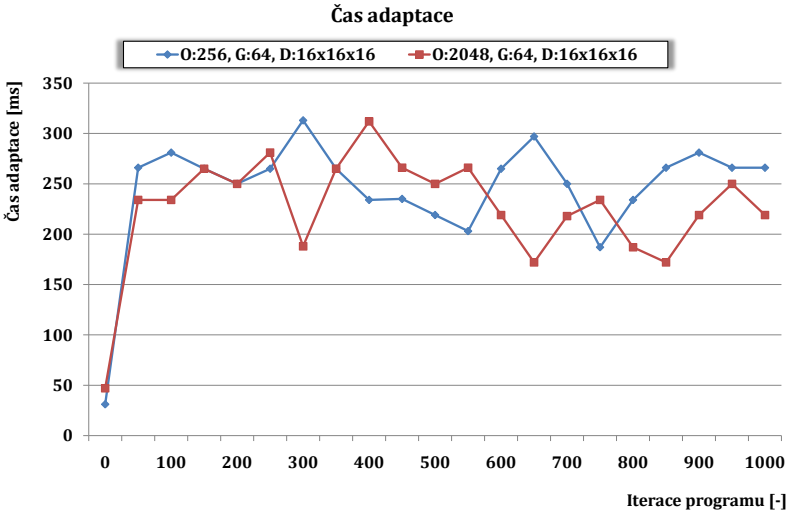
Obrázek 3.3: Původní řešení - doba adaptace u standardní konfigurace



Obrázek 3.4: Původní řešení - nebezpečí cesty u standardní konfigurace



Obrázek 3.5: Původní řešení - počet clusterů při velkém počtu překážek



Obrázek 3.6: Původní řešení - doba adaptace při velkém počtu překážek

3.4 Komplexní překážky

V počáteční fázi vývoje systému byly pro jednoduchost uvažovány pouze statické překážky kulového tvaru o různých průměrech (tzv. *bounding spheres*). Výpočet hodnot potenciálového pole těchto překážek tak bylo velmi jednoduché. Mapování překážek komplexních tvarů je však mnohem komplikovanější, jelikož pro výpočet hodnoty potenciálového pole v konkrétním místě je nutné pracovat s překážkami na úrovni trojúhelníků. Naivní algoritmus pro mapování, který do každé buňky matice ukládá nejmenší ze vzdáleností od jednotlivých trojúhelníků, zřejmě pracuje se složitostí $O(g^3t)$, kde g je řád matice a t je počet trojúhelníků všech překážek. Přestože existují efektivní postupy pro výpočet vzdálenosti bodu od trojúhelníka v prostoru [Ebe99], je zřejmé, že takové mapování překážky po každém jejím pohybu by bylo příliš časově náročné.

Bylo tedy nutné najít způsob, který by odstranil potřebu opakovaného mapování. Jako možné řešení bylo představeno tzv. *dynamické rastrové pole*, jehož hlavní myšlenkou je sledování dynamických překážek (změny pozice, orientace a velikosti) a aplikace příslušných geometrických transformací na jim odpovídající hodnoty v matici potenciálového pole. Takový přístup by samozřejmě vyžadoval, aby matice potenciálového pole obsahovala kromě hodnot také referenci na překážku, které tato hodnota patří a která je tedy danému místu nejblíže. Způsob aplikace geometrické transformace v rastrovém poli vysvětluje následující příklad: pokud se některá z překážek posune o známý vektor, jsou v tomto směru posunuty také hodnoty potenciálového pole v matici, jinými slovy jsou tyto hodnoty přesunuty do jiných buněk. Nové buňky jsou přepsány jen v případě, kdy se v nich nevyskytuje vyšší hodnota potenciálového pole některé jiné překážky. I nadále ale zůstávají problémy s "dírami", které vznikají při většině geometrických transformací. Tyto buňky matice je samozřejmě nutné naplnit novými hodnotami. Ani implementace takového přístupu se tedy neprojevila jako dostatečně efektivní a bylo nutné opustit variantu použití 3D matice hodnot potenciálového pole.

Kapitola 4

Nové řešení

Protože se použití 3D matice pro reprezentaci potenciálového pole překážek ukázalo jako neefektivní (viz sekce 3.4), bylo nutné zvolit jinou strukturu pro dělení prostoru, která by umožnila vkládání, odebírání a úpravu bodů "za běhu". Vzhledem k požadavkům kladeným na navrhovaný systém (sekce 1.1) byla pro další výzkum zvolena tzv. regulární triangulace v kombinaci s pseudooptimálním algoritmem pro hledání cest v dynamickém grafu, tedy grafu, jehož uzlové/hranové ohodnocení a stejně tak topologie se může v čase měnit. V následujících podkapitolách je představen obecný návrh takového systému a dále je podán teoretický základ stavebních kamenů, které tento systém využívá a kombinuje.

4.1 Obecný návrh

Podobně jako v případě předchozího řešení označovaného jako **Dispatcher** dostal také nově navrhovaný systém pracovní označení **TrippSys** nebo **TrippSystem** (Triangulation based path planning system). Jak je patrné z názvu, tento systém využívá k hledání cest triangulaci nad množinou překážek, které se mohou pohybovat, přibývat nebo ubývat. Triangulace budou blíže popsány a vysvětleny v následující sekci 4.2, prozatím je zjednodušeně definujme jako struktury spojující "nejbližší sousedy" v dané množině vstupních bodů. **TrippSys** používá tzv. regulární triangulaci - ta oproti jiným typům triangulací uvažuje také váhu vstupních bodů a této vlastnosti systém využívá pro definici různé míry nebezpečí jednotlivých překážek. Pro každou překážku je vypočítána tzv. obalová koule (*bounding sphere*), tedy nejmenší koule, která obsahuje všechny vrcholy překážky. Souřadnice a váhy vstupních bodů regulární triangulace jsou pak dány středy a poloměry obalových koulí příslušných překážek. Výstupem triangulace v $\mathbb{E}^3$ je množina čtyřstěnů (označovaných také jako *tetrahedrony* nebo *tetraedry*) a z těch lze jednoduše získat duální strukturu regulární triangulace, která bude taktéž popsána v sekci 4.2, tzv. *power diagram*. **TrippSys** na tento diagram nahlíží jako na neorientovaný, uzlově ohodnocený graf a používá jej k hledání cest mezi místy definovanými uživatelem. Při odebrání, vložení nebo změně některé z překážek je příslušná změna promítnuta do množiny vstupních bodů triangulace. Úpravou regulární triangulace samozřejmě dochází ke změně jejího power diagramu a jelikož je tento diagram využíván jako graf pro hledání cest, je důležité si uvědomit, že vrcholy takového grafu mohou průběžně přibývat nebo mizet. Pokud taková událost způsobí přerušení cesty nalezené v některé předchozí iteraci, **TrippSys** používá pseudooptimální algoritmus, který opravuje naposledy nalezenou cestu v místech, kde došlo k jejímu přerušení nebo ke zhoršení ohodnocení uzlů. Algoritmus je detailně popsán v sekci 4.3.

4.2 Regulární triangulace[1]

Triangulace (v obecném slova smyslu) představuje výstup tzv. *triangularizace*, typické úlohy z oblasti výpočetní geometrie, jejímž úkolem je pro danou množinu vstupních bodů rozdělit prostor (přesněji řečeno konvexní obálku těchto bodů) na oblasti, jejichž vrcholy korespondují s některými ze vstupních bodů, tak, aby tyto oblasti splňovaly určitá kritéria.

Před popisem regulární triangulace uveďme několik základních pojmů a vysvětlení:

Triangulace Mějme konečnou množinu vstupních bodů $P \subset \mathbb{R}^3$, $|P| = n$. Triangulací množiny P (značíme $T(P)$) rozumíme rozdělení konvexní obálky množiny P (dále jen $CH(P)$) na maximální množinu tetrahedronů (čtyřstěnů), jejichž vrcholy korespondují s body množiny P a které se vzájemně nepřekrývají.

Vrchol, hrana, trojúhelník triangulace Čtyřstěny triangulace se mohou dotýkat v jednom bodě (vrchol triangulace), svými hranami (hrana triangulace) nebo celou stěnou (trojúhelník triangulace).

Delaunayova triangulace Jak již bylo zmíněno v úvodu této kapitoly, při tvorbě triangulací jsou obvykle definována kritéria, která musí triangulace splňovat. Delaunayova triangulace množiny P (dále jen $DT(P)$) je taková triangulace $T(P)$, kde koule opsaná každému jejímu čtyřstěnu neobstahuje žádný další bod z množiny P (tzv. *minimum containment sphere* optimalizace). Časová složitost Delaunayovy triangulace v $\mathbb{E}^3$ je v nejhorším případě $O(n^2)$.

Další typy triangulací Mezi další známé triangulace patří např. datově závislá triangulace (*data dependent triangulation*) $DDT(P)$, tzv. "hltavá" triangulace (*greedy triangulation*) $GT(P)$ nebo triangulace s omezením $CDT(P)$, $CGT(P)$, kterých se ale ve 3D užívá jen zřídka.

Regulární triangulace (dále jen $RT(P)$), zobecnění Delaunayovy triangulace, na vstupu očekává tzv. *vážené body*, tedy body definované nejen souřadnicemi, ale také určitou reálnou vahou $w_p \in \mathbb{R}$, která pak ovlivňuje výsledný tvar triangulace. Paměťová i časová složitost je stejně jako v případě Delaunayovy triangulace v nejhorším případě $O(n^2)$. Vážené body obvykle bývají reprezentovány jako koule s poloměrem $\sqrt{w_p}$. Triangulace $T(P)$ je regulární, jestliže každý její čtyřstěn splňuje kritérium *globální regularity*. Tento pojem a další s ním související jsou definovány následovně (dle [Zem07]):

Power vzdálenost Power vzdálenost mezi váženým bodem p s vahou w_p a obyčejným bodem $x \in \mathbb{R}^3$ je definována jako $\pi_p(x) = |xp|^2 - w_p$ a může být interpretována jako druhá mocnina délky tečny z bodu x na kouli se středem v bodě p a poloměrem $\sqrt{w_p}$ (viz obr. 4.1 vlevo). V případě, že se bod x nachází uvnitř uvedené koule, je power vzdálenost záporná.

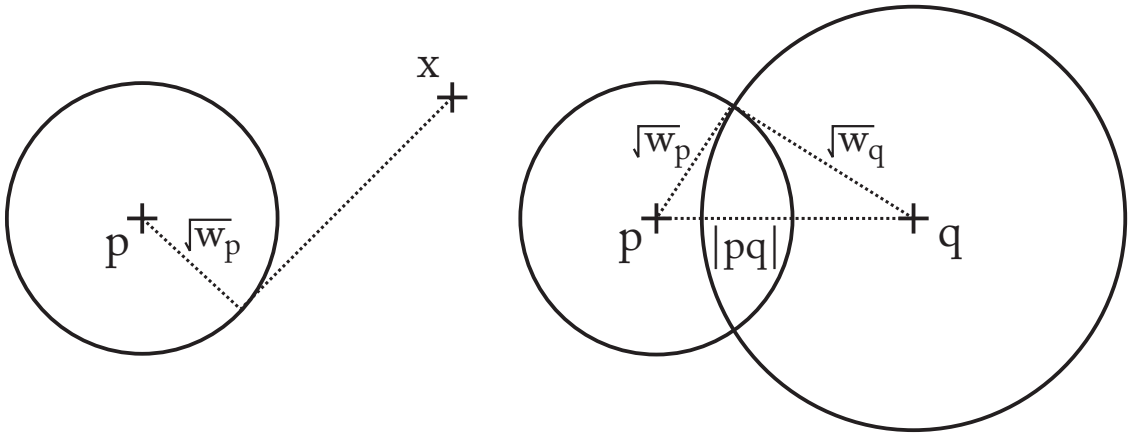
Ortogonalita Vážené body p, q jsou ortogonální, jestliže platí $|pq|^2 = w_p + w_q$ (viz obr. 4.1 vpravo).

[1] Pro detailní informace o této oblasti výpočetní geometrie odkazujeme čtenáře na diplomovou práci Ing. Michala Zemka [Zem07], který pro tento projekt poskytl teoretický základ regulárních a Delaunayových triangulací spolu s implementací jejich inkrementální konstrukce. Stručný teoretický přehled v této kapitole čerpá právě z jeho práce.

Ortogonální střed Vážený bod z je ortogonálním středem tetrahedronu $abcd$, jestliže z je ortogonální na všechny vrcholy tohoto čtyřstěnu.

Globální regularita Necht' z je ortogonálním středem tetrahedronu $abcd$. Tetrahedron $abcd$ je globálně regulární, jestliže pro všechny body $p \in P - \{a, b, c, d\}$ platí $\pi_z(p) > w_p$.

Lokální regularita Mějme dva tetrahedrony $abcd$ a $abce$ sdílející jednu stěnu a bod z, který je ortogonálním středem tetrahedronu $abcd$. Sdílenou stěnu abc pak nazveme lokálně regulární právě tehdy, když platí $\pi_z(e) > w_e$.



Obrázek 4.1: Power vzdálenost, ortogonalita dvojice vážených bodů
(převzato z [Zem07])

4.2.1 Metody konstrukce

Existuje několik přístupů k získání regulární triangulace pro vstupní množinu vážených bodů. Následuje stručný popis nejznámějších metod:

Inkrementální vkládání Metoda tvorby triangulace postupným přidáváním vstupních bodů. Po připojení každého bodu do triangulace je provedena série lokálních úprav, které zajistí, že je výsledná triangulace opět regulární.

Inkrementální konstrukce Algoritmus nejprve nalezne globálně regulární tetrahedron a k jeho stěnám pak připojuje další vstupní body, čímž přidává nové tetrahedrony. Základní časová složitost této metody je $O(n^3)$. Podle [BS*05] je však s použitím vhodných úprav možné získat skoro lineární složitost.

Převod do vyšší dimenze Výpočet triangulace je možné řešit také převodem na úlohu nalezení konvexní obálky ve vyšší dimenzi. Pokud pro vstupní množinu vážených bodů P definujeme množinu $P^+ = \{(x_p, y_p, z_p, x_p^2 + y_p^2 + z_p^2 - w_p) | p \in P\}$, pak lze zpětnou projekcí P^+ do $\mathbb{R}^3$ získat právě $RT(P)$.

Pro popisovaný systém hledání cest je použita první z těchto metod, tedy metoda inkrementálního vkládání. Samotný postup vkládání a odebírání bodů je naznačen v následující

podkapitole. Algoritmus potřebuje určitý výchozí čtyřstěn, do kterého jsou pak vkládány body vstupní množiny. Pro tyto účely jsou obvykle uvažovány 4 zvláštní body s takovým umístěním, aby jimi definovaný čtyřstěn obsahoval všechny vstupní body z P. Pro přesnost dodejme, že existuje také další varianta tohoto algoritmu, která obalový čtyřstěn nevyžaduje a pracuje s konvexní obálkou množiny vstupních bodů.

4.2.2 Vkládání bodů

Prvním krokem při vkládání bodu do triangulace je nalezení čtyřstěnu, ve kterém se tento bod nachází. To lze zajistit některou metodou ze třídy tzv. "procházek v triangulaci". Jako příklad uveďme stochastickou procházku, která začíná v libovolném čtyřstěnu *abcd* a z něj pak přechází do sousedního čtyřstěnu *abce* právě tehdy, když rovina daná body a, b, c odděluje bod d a bod, který hledáme. Tímto způsobem algoritmus postupuje až do doby, kdy toto pravidlo nesplňuje ani jedna stěna aktuálního čtyřstěnu a hledaný bod se tedy nachází v něm. Pro test orientace je použita níže definovaná funkce $orient(a, b, c, p)$. Očekávaná složitost algoritmů procházek v triangulaci je $O(\sqrt[4]{n})$ pro jeden bod, přičemž nejsou zapotřebí žádné další datové struktury.

$$orient(a,b,c,p) = \begin{vmatrix} x_a & y_a & z_a & 1 \\ x_b & y_b & z_b & 1 \\ x_c & y_c & z_c & 1 \\ x_p & y_p & z_p & 1 \end{vmatrix} = \begin{cases} < 0, p \text{ leží nad rovinou } abc \\ > 0, p \text{ leží pod rovinou } abc \\ = 0, p \text{ leží v rovině } abc \end{cases}$$

Jakmile jsme úspěšně lokalizovali čtyřstěn incidující s vkládaným bodem, můžeme triangulaci upravit podle následujících případů, které mohou nastat[2]:

- Jestliže vkládaný bod p leží uvnitř tetrahedronu *abcd*, jde o nejjednodušší případ, kdy je tento čtyřstěn rozdělen na 4 nové (*abcp*, *bcdp*, *cdap* a *dabp*).
- V případě, že bod p leží na stěně (např. *abc*) tetrahedronu *abcd* a existuje sousední tetrahedron sdílející tuto stěnu, jsou oba tetrahedrony nahrazeny 6 novými.
- Nejsložitějším případem je situace, kdy vstupní bod p leží na hraně (např. *ab*) tetrahedronu *abcd*. Všechny sousední tetrahedrony sdílející tuto hranu je pak nutné rozdělit na dva nové a původních N čtyřstěnů je tedy nahrazeno $2N$ novými.
- Jestliže bod p splývá s některým z vrcholů tetrahedronu, nemusí být do triangulace vůbec zařazen.

Po změně některých čtyřstěnů je nutné aplikovat na triangulaci sérii lokálních úprav tak, aby byla výsledná triangulace opět regulární. Jestliže stěna některého čtyřstěnu nesplňuje kritérium lokální regularity (viz sekce 4.2), je na tento čtyřstěn (a případně na jeho sousedy) aplikována lokální úprava podobná té při vkládání bodu a u vnějších stěn nových čtyřstěnů je opět testováno kritérium lokální regularity. Tento test je řešen podobně jako test orientace funkcí $regular(a, b, c, d, e)$. Stěna *abc* čtyřstěnu *abcd* je lokálně regulární právě tehdy, když $regular(a, b, c, d, e) < 0$.

[2]Neuvažujeme obecnou polohu bodů, tedy žádné 4 body neleží v rovině a žádných 5 bodů neleží na kouli. Postup v takových případech popisuje např. [Joe91].

$$regular(a,b,c,d,e) = orient(a,b,c,d) \begin{vmatrix} x_a & y_a & z_a & (x_a^2+y_a^2+z_a^2-w_a) & 1 \\ x_b & y_b & z_b & (x_b^2+y_b^2+z_b^2-w_b) & 1 \\ x_c & y_c & z_c & (x_c^2+y_c^2+z_c^2-w_c) & 1 \\ x_d & y_d & z_d & (x_d^2+y_d^2+z_d^2-w_d) & 1 \\ x_e & y_e & z_e & (x_e^2+y_e^2+z_e^2-w_e) & 1 \end{vmatrix}$$

Časová složitost inkrementálního vkládání je v nejhorším případě $O(n^2)$. Urychlení je pak možné získat především ve fázi hledání čtyřstěnu obsahujícího vkládaný bod. Při použití algoritmu procházky je očekávaná složitost $O(n\sqrt[4]{n})$.

4.2.3 Odebírání bodů

Odebrání bodu v trojrozměrné regulární triangulaci lze realizovat obrácením postupu vkládání nového bodu do triangulace v metodě inkrementálního vkládání [VPC02]. Tento postup používá prioritní frontu, do které jsou zpočátku vloženy všechny čtyřstěny incidující s odebíraným bodem. Stěna s největší prioritou je pak z fronty odebrána a aplikují se na ni již popsané lokální úpravy, přičemž nové stěny obsahující odebíraný bod jsou opět zařazeny do fronty. Tento postup se opakuje, dokud není ve frontě právě 6 stěn. Ty pak představují 4 čtyřstěny, které spolu sousedí a sdílejí odebíraný bod. Nakonec stačí tyto tetrahedrony nahradit jediným. Další možností realizace odebrání bodu z 3D triangulace je rozšíření Devillersova algoritmu původně navrženého pro 2D Delaunayovy triangulace, které detailně popisuje [Zem07].

4.3 *Gaps filling* algoritmus[3]

Algoritmus pracovně označovaný jako *gaps filling* představuje variantu pro rychlé nalezení suboptimální cesty v grafech, jejichž ohodnocení nebo topologie se v čase mění. Tento přístup nehledá celou cestu znovu po každé změně v grafu, ale upravuje naposledy nalezenou cestu v uzlech, které byly odstraněny nebo u kterých došlo ke zhoršení ohodnocení. Příklad takové úpravy je znázorněn na obr. 4.2 - v levém grafu je zvýrazněna nalezená cesta mezi uzly s a t, v pravém grafu je pak naznačeno přeplánování této cesty po zhoršení ohodnocení uzlů a, b. Původní cesta je zpracována po jednotlivých uzlech následujícím způsobem: pokud je daný uzel n cesty stále platným uzlem grafu a pokud nedošlo ke zhoršení jeho ohodnocení, je zařazen do nové cesty, v opačném případě je vynechán. Zařazení uzlu n do nové cesty má potom dva možné postupy. Pokud uzel n přímo sousedí s posledním uzlem u nové cesty, je k ní jednoduše přidán, jinak je tato cesta doplněna o uzly cesty mezi u a n nalezené Dijkstrovým algoritmem. Postup algoritmu *gaps filling* je naznačen v pseudokódu 4.1.

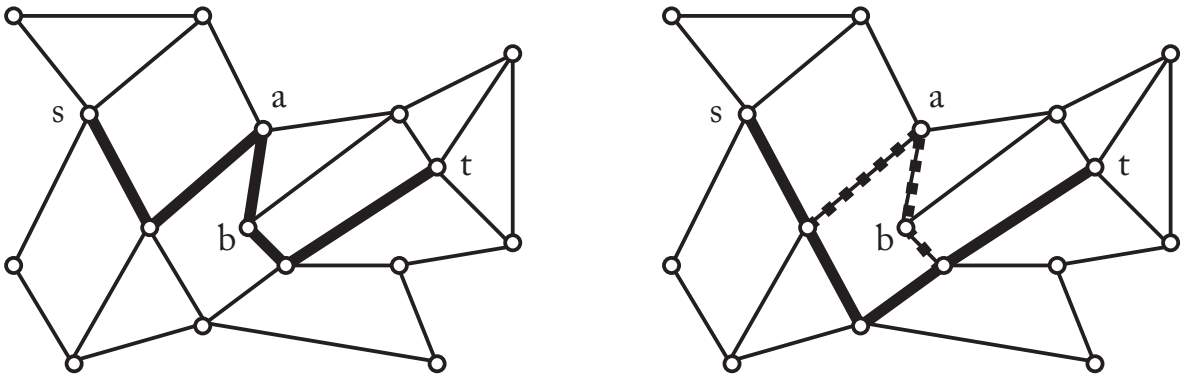
Za cenu pseudooptimálního řešení poskytuje tento algoritmus velmi dobré urychlení co se týče počtu uzlů, které je nutné zpracovat k nalezení cesty. Grafy 4.3-4.6 naznačují urychlení pro planární graf o 32x32 uzlech, které jsou spojeny s náhodně vybranými sousedními uzly.

[3] Algoritmus byl autorem tohoto dokumentu detailně popsán a analyzován v článku [Bro07], který je součástí této práce jako příloha E.

Algoritmus 4.1 *Gaps filling* algoritmus (úprava nalezené cesty)

Vstup: graf $G(V, E)$, cesta $P \subset V$, poč. a konc. uzly $s, t \in P$
Výstup: nová cesta P'

```
P' ⇐ {s}
u ⇐ NULL {pomocný uzel}
for all n ∈ P − {s} do
  if n byl odstraněn ∨ n má horší ohodnocení then
    if u == NULL then
      u ⇐ předchůdce uzlu n na cestě P
    end if
  else
    if u == NULL then
      P' ⇐ P' ∪ {n}
    else
      P' ⇐ P' ∪ path(u, n) {Nalezení cesty z uzlu (u) do n Dijkstrovým alg.}
      u ⇐ NULL
    end if
  end if
end for
```



Obrázek 4.2: Chování *gaps filling* algoritmu
(převzato z [Bro07])

Graf 4.3 představuje váhu cest[4] nalezených oběma algoritmy (Dijkstrův algoritmus a *gaps filling*) pro graf, kde se ohodnocení uzlů mění náhodně v čase. Snadno z něj lze vypozorovat již zmíněnou skutečnost, že algoritmus *gaps filling* nabízí suboptimální řešení, v průměru přibližně 1,1-optimální výsledky (pro informace o t-optimalitě viz sekce 2.4.2). Výjimkou jsou oblasti kolem iterací číslo 120 a 220, kdy tento algoritmus nachází cestu, jejíž váha je shodná s váhou optimální cesty nalezené Dijkstrovým algoritmem. Tato skutečnost zřejmě může mít dvě různé příčiny: v těchto iteracích buď došlo ke zhoršení ohodnocení všech uzlů cesty a algoritmus *gaps filling* tak přeplánoval celou cestu s použitím Dijkstrova algoritmu anebo se lokálně upravená cesta stala shodnou s optimální cestou.

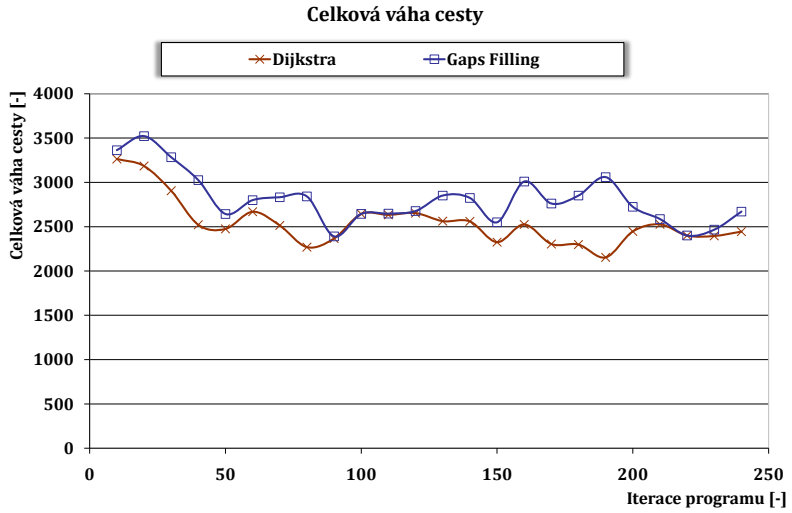
V grafu 4.4 je naznačen počet uzlů, které algoritmy zpracovaly pro nalezení cesty mezi stejnými dvěma uzly. Přestože pseudooptimální přístup poskytuje již zmíněné 1,1-optimální výsledky , k nalezení takového řešení potřebuje v průměru pouze 36% z uzlů zpracovaných Dijkstrovým algoritmem. V tabulce 4.1 lze najít urychlení oproti Dijkstrovu algoritmu pro různé počty uzlů, u kterých došlo od posledního hledání cesty ke změně ohodnocení. Graf 4.4 navíc dokazuje důležitou věc popisovanou v předchozím odstavci: přestože algoritmus *gaps filling* poskytl v iteracích číslo 120 a 220 optimální řešení, k jeho nalezení zpracoval jen část uzlů zpracovávaných Dijkstrovým algoritmem.

Grafy 4.5 a 4.6 pak představují celkovou váhu cesty a počet zpracovaných uzlů pro stejnou množinu algoritmů, ale v grafu, jehož změny jsou lokalizované v jedné rozšiřující se oblasti. Za cenu nalezení 1,3-optimálního řešení zpracuje *gaps filling* metoda oproti Dijkstrovu algoritmu 26% uzlů.

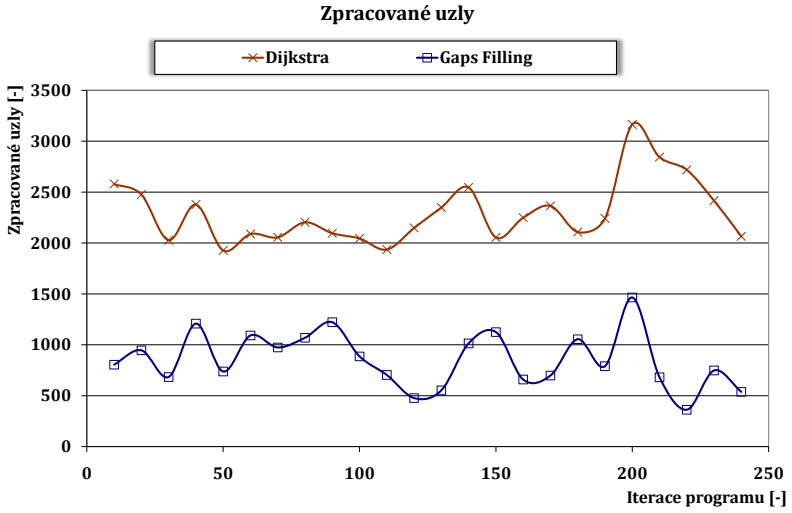
Počet změněných uzlů [%]	Počet zprac. uzlů [%]	Váha cesty [%]
25	13,70	126,26
50	37,75	115,76
75	85,67	108,67

Tabulka 4.1: Urychlení *gaps filling* algoritmu pro různé změny ohodnocení

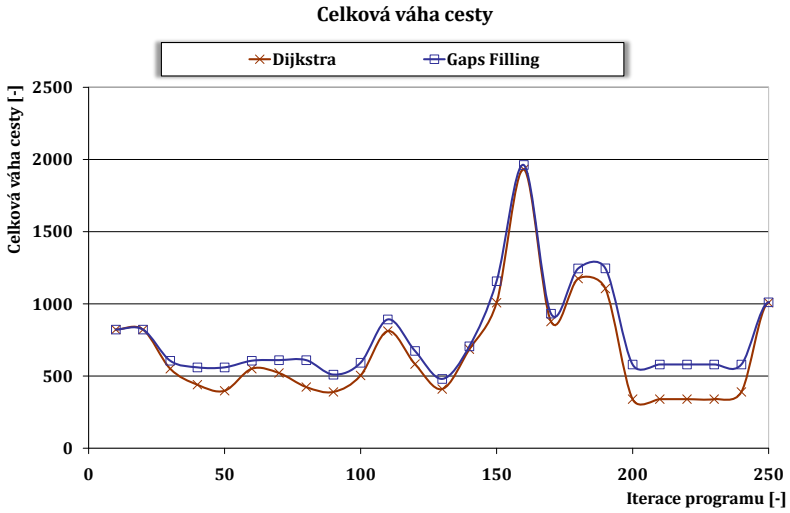
[4]Váhou cesty se v našem kontextu rozumí součet vah jednotlivých uzlů cesty. Nalezená cesta je tím výhodnější, čím je menší její váha.



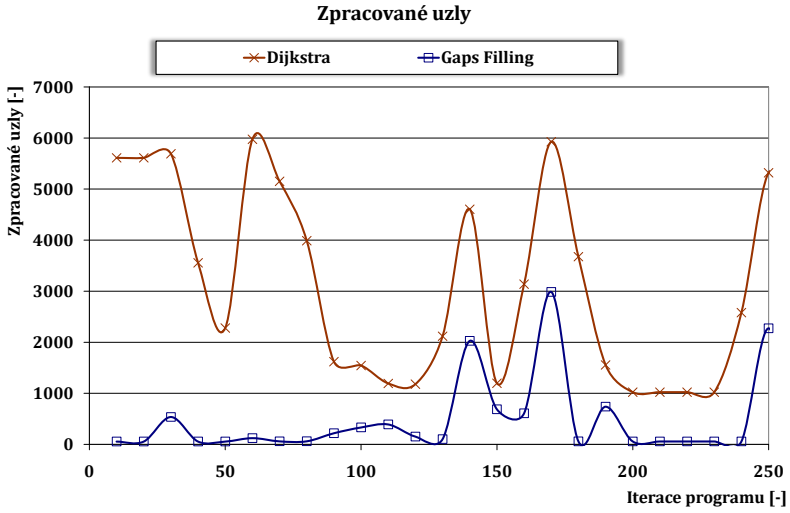
Obrázek 4.3: Algoritmus *gaps filling* - váhy cest při náhodných změnách grafu



Obrázek 4.4: Algoritmus *gaps filling* - počet zprac. uzlů při náhodných změnách grafu



Obrázek 4.5: Algoritmus *gaps filling* - váhy cest při lokalizovaných změnách grafu



Obrázek 4.6: Algoritmus *gaps filling* - počet zprac. uzlů při lokalizovaných změnách grafu

Kapitola 5

Implementace

Pro implementaci systému hledání cest a testovacích aplikací byl hned z několika důvodů zvolen jazyk C# 2.0. Jde o moderní programovací jazyk nabízející mnoho prvků, díky kterým je kód intuitivní a dobře pochopitelný, např. bezpečné prototypy funkcí (tzv. *delegáti*), události, generické datové struktury apod. Navíc použitá knihovna[1] Ing. Michala Zemka zajišťující práci s triangulacemi je stejně jako původní autorův systém (viz kapitola 3) realizována právě v tomto jazyce. Autor tohoto dokumentu používá uvedenou knihovnu regulárních triangulací ve vlastním systému hledání cest a dále implementoval ukázkovou herní aplikaci pro jeho prezentaci a konzolovou aplikaci pro analýzu.

5.1 Systém hledání cest

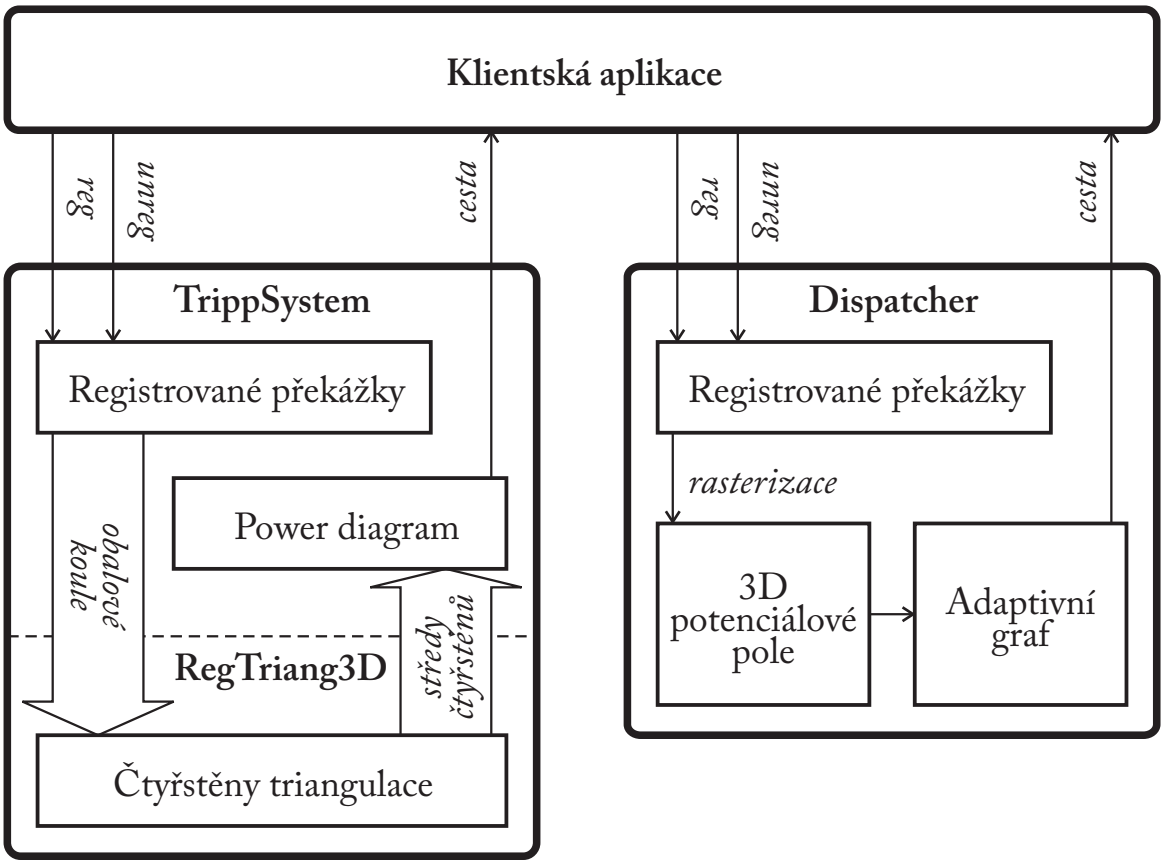
Na obrázku 5.1 je naznačen návrh systému **TrippSys** a způsob jeho komunikace s klientskou aplikací (pro účely porovnání je do návrhu zahrnut také původní systém **Dispatcher**). Před samotným popisem návrhu uveďme základní množinu operací, které figurují v komunikaci mezi klientskou aplikací a systémem hledání cest:

reg Registrace překážky do systému. Překážka klientské aplikace může měnit svůj tvar, pozici nebo orientaci a systém musí být schopen reagovat na změnu každé registrované překážky.

unreg Odhlášení překážky ze systému. Taková překážka nadále nebude ovlivňovat výstup systému.

cesta Nalezení cesty podle daného kritéria mezi definovanými místy. Při hledání se samozřejmě přihlíží k překážkám registrovaným v systému.

[1] Knihovna zajišťuje tvorbu regulární triangulace metodou inkrementálního vkládání. Implementace tohoto algoritmu poskytovala možnost vkládání dalších bodů, avšak odebírání bodů z triangulace bylo v době vzniku tohoto dokumentu ve fázi testování a v dokumentaci tak není uvažováno.



Obrázek 5.1: Základní struktura navrhovaného systému hledání cest **TrippSys**

Obrázek A.2 představuje nejdůležitější třídy tvořící rozhraní mezi systémem hledání cest a klientskou aplikací. Hlavní třídou celého projektu je `TrippSystem`, která zprostředkovává veškeré potřebné operace pro klientskou aplikaci, tedy přihlášení/odhlášení překážek, nalezení cesty mezi definovanými místy a přeplánování původní cesty. Systém je schopný zaregistrovat instanci libovolné třídy, která dědí od abstraktní třídy `Obstacle`. Tato abstraktní třída vyžaduje implementaci několika metod:

- `IEnumerable<Vector> Vertices()` pro výčet veškerých vrcholů překážky (relativní souřadnice vzhledem ke středu překážky).
- `void GetPosition(out float x, out float y, out float z)` pro zjištění pozice překážky.
- `void GetRotation(out float x, out float y, out float z)` pro zjištění orientace překážky.
- `void GetScale(out float x, out float y, out float z)` pro zjištění měřítka překážky.

Systém hledání cest je pak díky těmto metodám schopný zjistit veškeré potřebné informace o překážce a v rodičovské třídě `Obstacle` si navíc pro každou z nich uchovává vlastní důležité

informace, konkrétně index do interního pole obalových koulí registrovaných překážek (viz *Registrované překážky* v obrázku 5.1). Při registraci překážky do systému je prostřednictvím uvedených metod určena pozice a poloměr obalové koule, která je následně uložena v lineárním poli pevné velikosti. Toto pole následně slouží jako vstup algoritmu pro výpočet regulární triangulace. Abstraktní třída `Obstacle` proto nabízí události, které musí od ní odvozené třídy vyvolat vždy, když dojde ke změně překážky:

- `event ObstacleChangeHandler VerticesChange` při změně modelu překážky.
- `event ObstacleChangeHandler PositionChange` při změně pozice překážky.
- `event ObstacleChangeHandler RotationChange` při změně orientace překážky.
- `event ObstacleChangeHandler ScaleChange` při změně měřítka překážky.

Vyvoláním některé z těchto událostí na straně klientské aplikace dojde automaticky k odpovídající změně v poli obalových koulí a následně také v triangulaci na straně systému hledání cest. Na obrázku A.3 je k dispozici class diagram další důležité skupiny tříd projektu **TrippSystem**, tříd souvisejících s hledáním cest mezi registrovanými překážkami. Třída `Vertex` představuje vrchol grafu tvořeného duální strukturou k regulární triangulaci, již zmiňovaným *power diagramem*, a udržuje veškerá data potřebná pro samotné hledání cest:

- `Vector Position` pozice vrcholu, konkrétně ortogonální střed čtyřstěnu, ke kterému tento vrchol patří.
- `List<Vertex> Neighbours` seznam vrcholů, se kterými tento vrchol sousedí.
- `float Distance` vážená vzdálenost od výchozího vrcholu (při použití Dijkstrova algoritmu).
- `int LastVisited` číslo iterace, kdy byl tento vrchol naposledy navštíven (obdoba obarvování navštívených uzlů v BFS).
- `int TetraId` index tetrahedronu, ke kterému vrchol patří. Umožňuje zjištění nekonzistence, kdy tetrahedron, ke kterému náleží daný vrchol, již v triangulaci neexistuje.

Třída `Paths` poskytuje statické metody pro nalezení cesty mezi konkrétními vrcholy pomocí několika algoritmů. Připraven byl klasický *breadth-first search* algoritmus, Dijkstrův algoritmus a algoritmus A*, který je možné použít také s vlastní funkcí definující váhu vrcholu grafu. Metoda `List<Vertex> Path_AStar(Vertex src,Vertex dest,VertexWeight w)` hledá cestu v grafu mezi uzly `src` a `dest`, přičemž jako heuristiku pro ohodnocení jednotlivých uzlů používá metodu určenou parametrem `w`, tedy libovolnou metodu splňující definici delegáta `delegate float VertexWeight(Vertex v, Vertex src, Vertex dest)`.

Hlavní třída systému, `TrippSystem`, nakonec nabízí metody `List<Vertex> Path(Vector src, Vector dest)` (resp. `List<Vertex> Path(Vector src, Vector dest, List<Vertex> oldPath)`) pro nalezení nové (resp. přepracování původní) cesty mezi danými body v prostoru. Obě tyto metody nejprve lokalizují čtyřstěny obsahující počáteční a cílový bod a poté využívají právě statické metody třídy `Paths`.

Systém hledání cest **TrippSystem** je realizován jako dynamická knihovna v jazyce C#2.0, která využívá další knihovnu s implementací metod pro tvorbu a práci s regulární triangulací [Zem07], `Regular3D.dll`. Při tvorbě triangulace jsou navíc využívány další knihovny, **TrippSystem** tak musí být distribuován společně s těmito součástmi:

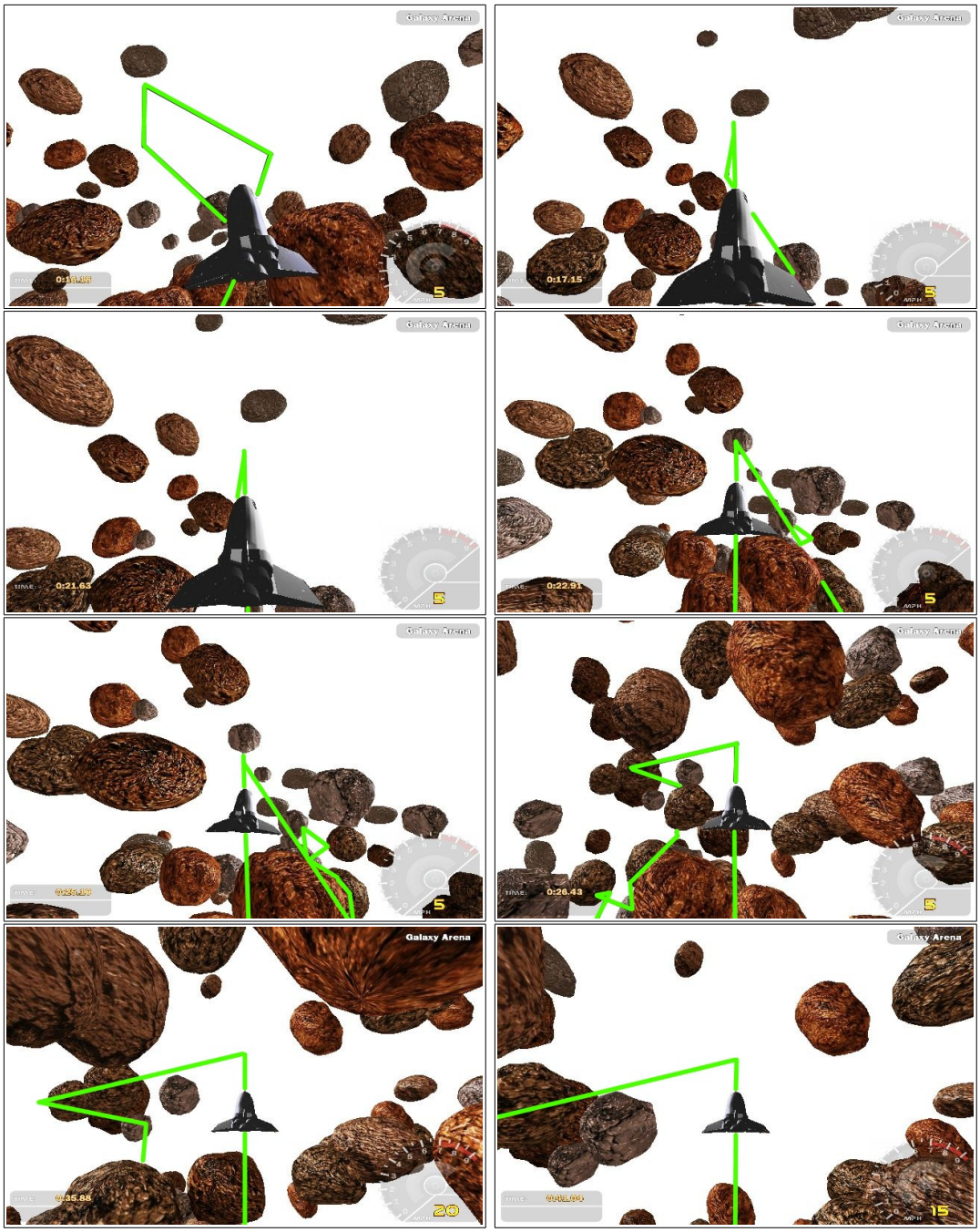
- `TrippSystem.dll` - knihovna s implementací metod hledání cest a definicí všech datových struktur používaných v klientské aplikaci.
- `Regular3D.dll` - tvorba a úprava regulární triangulace, kterou používá projekt **TrippSystem** pro dynamické dělení prostoru.
- `MathNet.Iridium.dll` - open-source knihovna pro platformu .NET umožňující symbolické a numerické výpočty[2].

Projekt samotný je přiložen na CD v adresáři `Source/GalaxyWars/TrippSystem` a podrobná dokumentace[3] je připravena v adresáři `Documentation/TrippSystem` v několika různých formátech.

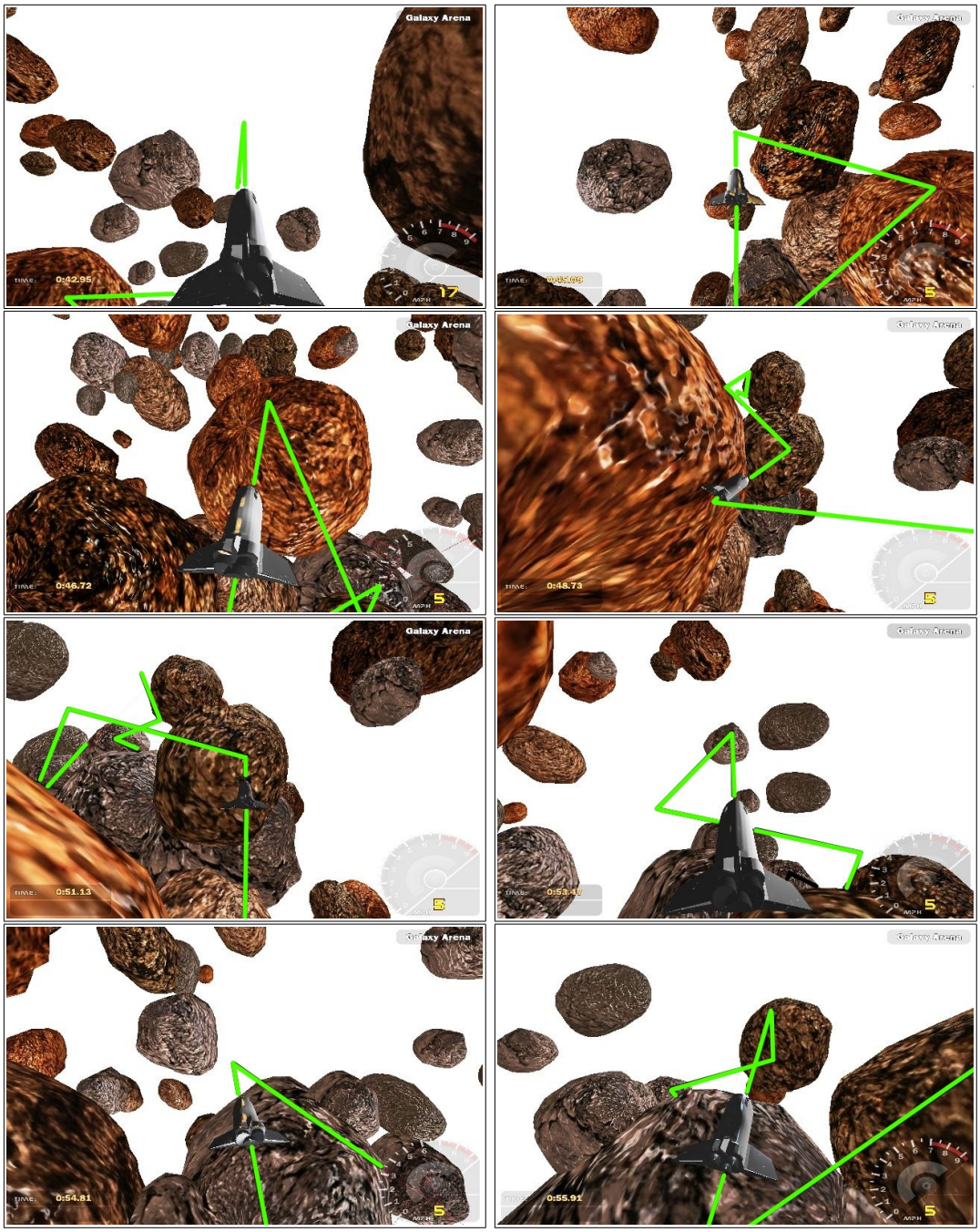
5.2 Testovací aplikace GalaxyWars

Pro účely vizualizace výsledků systému **TrippSys** byla připravena jednoduchá aplikace[4] umožňující průlet trojrozměrným polem asteroidů. Aplikace využívá knihovny **XNA Game Studio 2.0**[5], které zastřešují knihovny **Microsoft DirectX**[6] a přidávají další prvky užitečné při vývoji her. Na začátku aplikace je připraveno pole pevné velikosti s asteroidy, kterým je náhodně přiřazena konkrétní velikost a pozice tak, aby pokrývaly určitý pevně stanovený prostor. Všechny tyto asteroidy pak implementují abstraktní třídu `Obstacle` systému **TrippSys** a po spuštění aplikace jsou do něj zaregistrovány. Ještě před samotným vstupem do hry je nakonec **TrippSys** použit k nalezení cesty. Pro jednoduchost jde vždy o cestu mezi dvěma protilehlými rohy zmíněné oblasti, kterou pokrývají náhodně generované asteroidy. Na následujících obrázcích je představeno několik snímků z průletu po nalezené cestě a v přílohách A.5-A.9 jsou pak k dispozici další ukázky ze stejné aplikace.

[2] http://mathnet.opensourcedotnet.info
[3] Dokumentace byla vygenerována programem **Doxygen** (http://www.doxygen.org).
[4] Příloha B.1 poskytuje její uživatelskou dokumentaci.
[5] http://creators.xna.com
[6] http://msdn.microsoft.com/directX



Obrázek 5.2: Ukázky aplikace **GalaxyWars** (snímky 1 až 8)



Obrázek 5.3: Ukázky aplikace **GalaxyWars** (snímky 9 až 16)

5.3 Testovací aplikace Measuring

Pro snazší analýzu systémů popisovaných v kapitolách 3 a 4 byla dále připravena jednoduchá konzolová aplikace, která používá oba tyto systémy ve stejném prostředí. V tomto prostředí je podle parametrů, se kterými je aplikace spuštěna, připravena konkrétní konfigurace překážek a vlastnosti obou systémů jsou následně zaznamenávány do souborů pro další zpracování.

Na obrázku A.4 je k dispozici diagram tříd této aplikace: `Program`, hlavní třída aplikace, zpracovává ve vstupní metodě `void Main(string[] args)` parametry předané aplikaci a podle nich poté spouští testy systémů pro jednotlivé konfigurace překážek ve scéně. Využívá k tomu několik dodatečných tříd:

- `MyMesh` - definice tvaru překážek používaných v aplikaci. Data pro tuto třídu jsou načítána z externích souborů, kde je na prvním řádku uložen počet vrcholů a na dalších řádcích pak souřadnice jednotlivých vrcholů s komponentami oddělenými středníkem.
- `MyObstacle` reprezentuje konkrétní překážku její pozicí, velikostí, orientací a geometrickým tvarem (instance třídy `MyMesh`). Zároveň implementuje abstraktní třídu `Obstacle` systému **TrippSys** a rozhraní `IThreat` systému **Dispatcher**, aby do nich následně mohla být registrována.
- `MyOutput` - vlastní výstup do souboru umožňující zápis v několika formátech (viz výčet `OutputType` v diagramu tříd): hodnoty oddělené středníkem (*comma separated values* nebo CSV), hodnoty oddělené tabulátorem (*tabulator separated values* nebo TSV) a formát pro tvorbu ohraničené/neohraničené tabulky v jazyce LaTeX.
- `MySettings` definuje jednotlivá nastavení aplikace. Instance této třídy je vytvořena a inicializována na základě parametrů předaných aplikaci.

Kapitola 6

Experimenty a výsledky

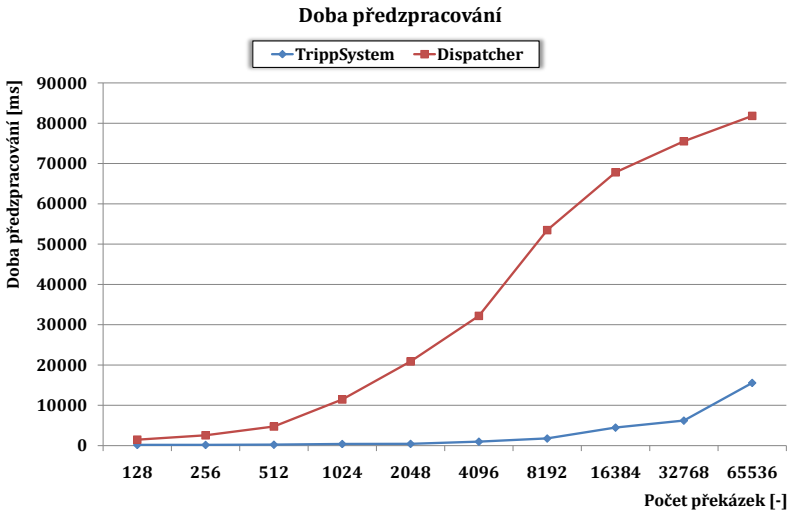
Před analýzou připraveného systému hledání cest je důležité definovat vlastnosti, které je u něj nutné nebo vhodné zkoumat. Podle oblasti aplikace a požadavků uvedených v sekci 1.1 má smysl sledovat následující charakteristiky:

- Doba předzpracování (tzv. *preprocessing*), tedy doba inicializace systému ještě před hlavní smyčkou programu, ve které je následně systém využíván. Přestože doba předzpracování nemusí hrát v případě virtuální reality zásadní roli, stále jde o důležitou vlastnost.
- Doba odezvy na změnu ve scéně, tedy čas potřebný k tomu, aby se systém přizpůsobil určité změně v prostředí. Doba takové odezvy je pravděpodobně jedním z nejdůležitějších faktorů, neboť v kontextu herních aplikací a virtuální reality mohou tyto změny nastávat takřka v každé iteraci.
- Doba nalezení cesty je zřejmě nejdůležitější charakteristikou systému, protože tuto operaci může klientská aplikace využívat v každé iteraci programu několikrát, např. pro nalezení více cest pro několik počítačem ovládaných avatarů.
- Kvalita cesty, lépe řečeno odchylka nalezené cesty od cesty optimální. Opět jde o vlastnost, která není pro hledání cest v herních aplikacích zásadní. Je však dobré poukázat na souvislost mezi pseudooptimalitou nalezené cesty a dobou jejího nalezení.

Výše uvedené vlastnosti byly testovány v závislosti na počtu překážek ve scéně, na jejich velikosti a rozložení. V konzolové aplikaci **Measuring** popsané v sekci 5.3 byla vždy náhodně vygenerována množina konkrétního počtu překážek a aplikace následně měřila inicializaci systému a čas potřebný k registraci nové překážky. Stejným způsobem byl současně testován také původní přístup popisovaný v kapitole 3. Po inicializaci byla vždy nalezena optimální cesta s použitím Dijkstrova algoritmu a pseudooptimální cesta pomocí algoritmu *gaps filling*, přičemž výsledkem tohoto měření byl čas nalezení cesty a její váha, resp. součet vah jejích uzlů, který je v optimálním případě minimální. Jednotlivá pozorování a výsledky měření jsou podrobně popsány v následujících podkapitolách. Pro úplnost ještě dodejme, že veškerá data byla naměřena na počítači s procesorem AMD Turion^{64} X2 (1,6GHz, 1MB L2 cache) a pamětí 1GB DDR2.

6.1 Doba předzpracování

Graf na obrázku 6.1 představuje dobu předzpracování nově implementovaného řešení v porovnání s původním systémem **Dispatcher** využívajícím 3D matici a adaptivní graf. Doby inicializace jsou naměřeny pro 2^7 až 2^{16} překážek, přičemž v systému **Dispatcher** jsou všechny tyto překážky interpretovány jako statické[1]. Původní řešení vyžaduje v případě statických překážek vyšší čas pro inicializaci kvůli nutnosti mapovat každou z nich do 3D matice potenciálového pole (viz sekce 3.1), v případě 2^{16} statických překážek je to téměř 82 sekund. Závislost času inicializace nového systému využívajícího regulární triangulaci je lineární a ve stejném případě 2^{16} překážek vyžaduje přibližně 16 sekund. Pokud jde o dynamické překážky, nově navrhované řešení je pomalejší, protože projekt **Dispatcher** dynamické překážky nijak nepředzpracovává, zatímco **TrippSys** musí vždy odpovídajícím způsobem upravit triangulaci.

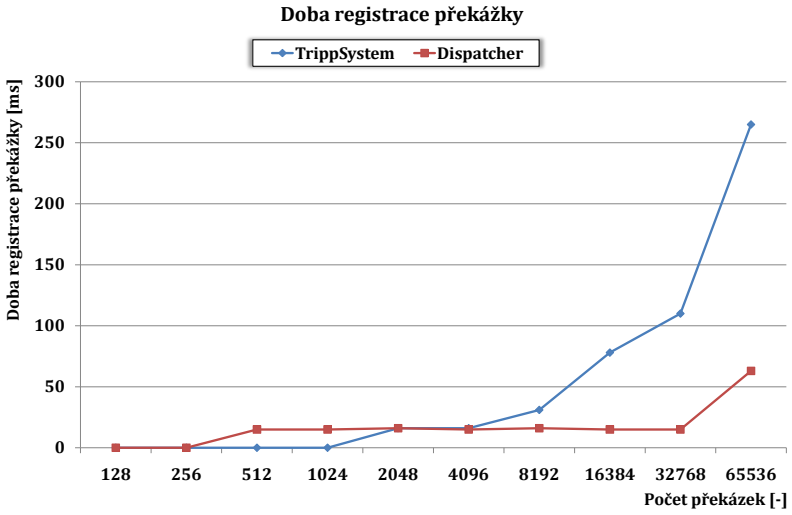


Obrázek 6.1: **TrippSystem** - předzpracování pro různé počty překážek
(v systému **Dispatcher** jsou všechny překážky interpretovány jako statické)

[1] Jak bylo popsáno v kapitole 3, **Dispatcher** striktně odděluje statické a dynamické překážky a s každou třídou zachází jiným způsobem.

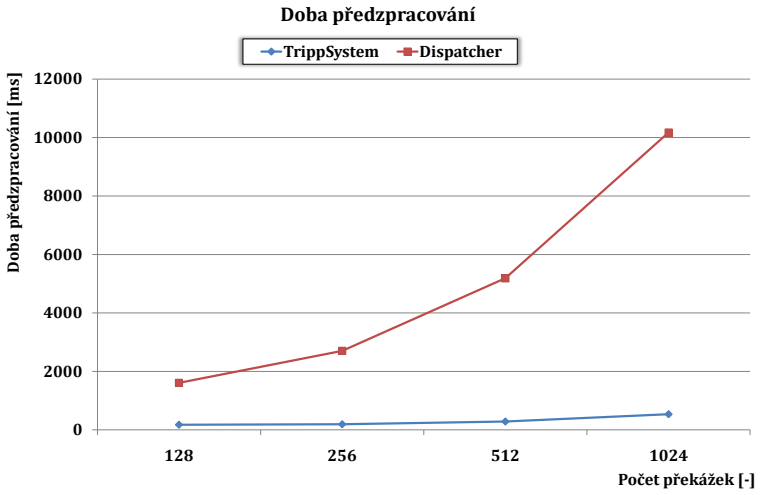
6.2 Doba registrace překážky

V grafu 6.2 a tabulce 6.1 jsou (opět pro oba systémy) znázorněny doby potřebné k registraci[2] nové překážky v závislosti na počtu již zaznamenaných objektů. Přestože regulární triangulace potřebuje k adaptaci na nově přidanou překážku časy v milisekundách i pro vysoké počty již obsažených překážek, lze vidět, že registrace dynamické překážky u původního řešení (**Dispatcher**) je mnohem rychlejší. Jak již bylo zmíněno v předchozím odstavci, původní řešení odděluje statické překážky od dynamických, statické překážky mapuje do 3D matice v rámci předzpracování a dále už umožňuje manipulaci pouze s dynamickými překážkami. V tomto případě je vkládaná překážka jako jediná interpretována jako dynamická. Pro lepší pochopení této vlastnosti jsou v grafech 6.3, 6.4 a v tabulce 6.2 znázorněny doby inicializace a registrace pro 2^7 - 2^{10} překážek v případě, kdy je polovina z nich interpretována jako statická a polovina jako dynamická. Po vložení nové překážky upravuje projekt **Dispatcher** adaptivní graf podle hodnot nebezpečí v jednotlivých místech scény. Nebezpečí je však definováno nejen hodnotami potenciálového pole statických překážek, ale také vzdáleností okolních dynamických překážek. Čím více jich tedy systém obsahuje, tím delší je čas adaptace datových struktur tohoto systému. Z grafu 6.4 je pak zřejmé, že registrace nové překážky v systému **Dispatcher** je časově mnohem náročnější.

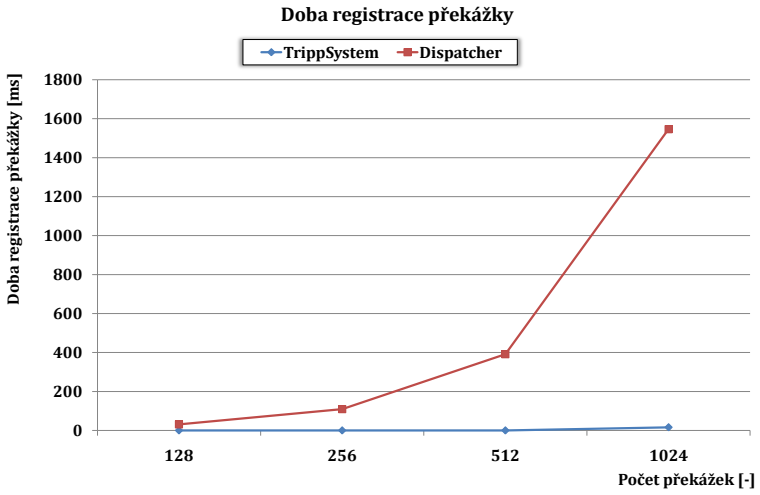


Obrázek 6.2: **TrippSystem** - doba registrace pro různé počty překážek
(v systému **Dispatcher** jsou všechny překážky interpretovány jako statické)

[2]Pod pojmem registrace v tomto případě rozumíme operaci přihlášení nové překážky do systému a následné přizpůsobení jeho vnitřních datových struktur.



Obrázek 6.3: **TrippSystem** - doba předzpracování (polovina překážek statická a polovina dynamická)



Obrázek 6.4: **TrippSystem** - doba registrace (polovina překážek statická a polovina dynamická)

	TrippSys		**Dispatcher**	
# překážek	Předzprac. [ms]	Registrace [ms]	Předzprac. [ms]	Registrace [ms]
2^7	172	0	1484	0
2^8	172	0	2562	15
2^9	234	0	4750	15
2^{10}	407	0	11453	16
2^{11}	437	16	20906	0
2^{12}	985	16	32203	15
2^{13}	1765	31	53469	16
2^{14}	4469	78	67828	15
2^{15}	6203	110	75532	15
2^{16}	15547	265	81813	63

Tabulka 6.1: **TrippSystem** a **Dispatcher** - předzpracování a registrace (v systému **Dispatcher** jsou všechny překážky interpretovány jako statické)

	TrippSys		**Dispatcher**	
# překážek	Předzprac. [ms]	Registrace [ms]	Předzprac. [ms]	Registrace [ms]
2^7	172	0	1609	31
2^8	188	0	2703	109
2^9	266	0	5187	391
2^{10}	437	16	10172	1546

Tabulka 6.2: **TrippSystem** a **Dispatcher** - předzpracování a registrace (polovina překážek statická a polovina dynamická)

6.3 Hledání cesty a její kvalita

Pro různé počty překážek byla v inicializovaném systému nalezena cesta mezi dvěma pevně danými body pomocí optimálního Dijkstrova algoritmu a po registraci nové překážky pak byla tato cesta opravena pomocí algoritmu *gaps filling* (viz sekce 4.3). Tabulka 6.3 a grafy 6.5, 6.6 ukazují srovnání těchto algoritmů. Podobně jako u výsledků představených v sekci 4.3, i zde přináší použití pseudooptimálního řešení značné urychlení. Z grafu 6.5 je zřejmé, že čas potřebný k nalezení cesty roste spolu s počtem překážek. To je samozřejmě způsobeno rostoucím počtem čtyřstěnů triangulace, tedy rostoucím počtem vrcholů power diagramu, ve kterém se cesta hledá. Jak je vidět, algoritmus *gaps filling* nachází v průměru 1,1-optimální řešení[3] (v nejhorším případě pak 1,16-optimální řešení) v mnohem menším čase.

	Dijkstrův algoritmus		*Gaps filling*	
# překážek	Nalezení [ms]	Váha [-]	Nalezení [ms]	Váha [-]
2^{12}	0	0.616861	0	0.6614693
2^{13}	0	0.6066013	0	0.664586
2^{14}	31	0.6860374	0	0.7374998
2^{15}	63	0.7271375	0	0.7808142
2^{16}	78	0.8220786	15	0.879591

Tabulka 6.3: Algoritmus *gaps filling* - urychlení a kvalita nalezené cesty

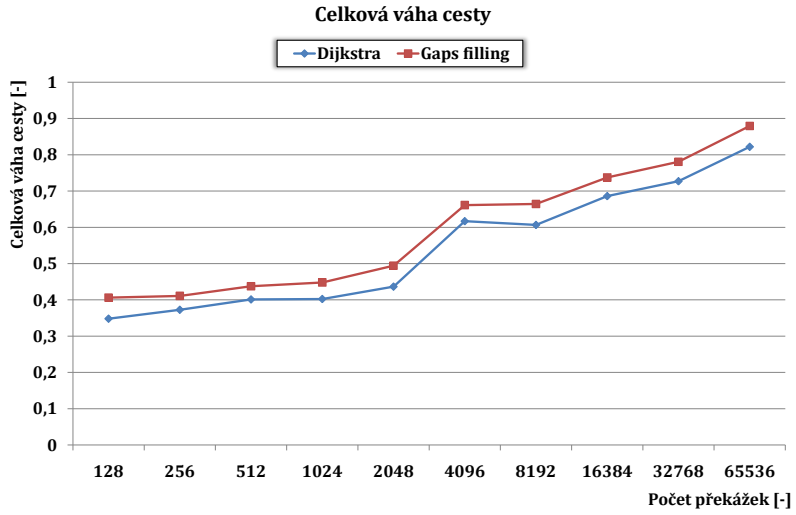
Důležitou částí této kapitoly jsou pak grafy 6.7 a 6.8, které představují srovnání systémů **TrippSys** a **Dispatcher** z hlediska rychlosti nalezení cesty a její kvality reprezentované průměrnou vahou nebezpečí[4] všech uzlů cesty. Z grafu 6.7 je viditelná zásadní informace popsaná již v kapitole 3. V případě systému **Dispatcher** je doba nalezení cesty závislá na maximální povolené úrovni zjemnění popisované adaptivní struktury. V grafu jsou proto pro názornost zobrazeny dvě varianty tohoto systému: s maximálním povoleným rozdělením adaptivní struktury na 16x16x16 clusterů a na 32x32x32 clusterů. U projektu **TrippSys** je doba nalezení cesty zřejmě závislá na počtu překážek - s rostoucím počtem překážek ve scéně roste také počet uzlů duální struktury, mezi kterými se cesta hledá. Jak je navíc zřejmé z grafu 6.8, **TrippSys** dokáže najít v kratším čase cestu, jejíž průměrná váha nebezpečí je mnohem menší[5]. Tato vlastnost je způsobena tím, že zatímco **Dispatcher** hledá cestu "pouze" po hranách a diagonálách clusterů, při použití regulární triangulace definují uzly její duální reprezentace nejbezpečnější místa vzhledem ke vzdálenosti od okolních překážek. Z grafu lze vyčíst ještě další důležitou vlastnost původního řešení: pokud je systému **Dispatcher** povolena vyšší úroveň zjemnění adaptivní struktury, hledání cesty ve větším počtu clusterů pak trvá delší dobu, ale výsledná cesta je kvalitnější právě díky většímu počtu hran.

[3] Celková váha cesty je opět definována součtem vah jednotlivých uzlů, kterými tato cesta prochází.

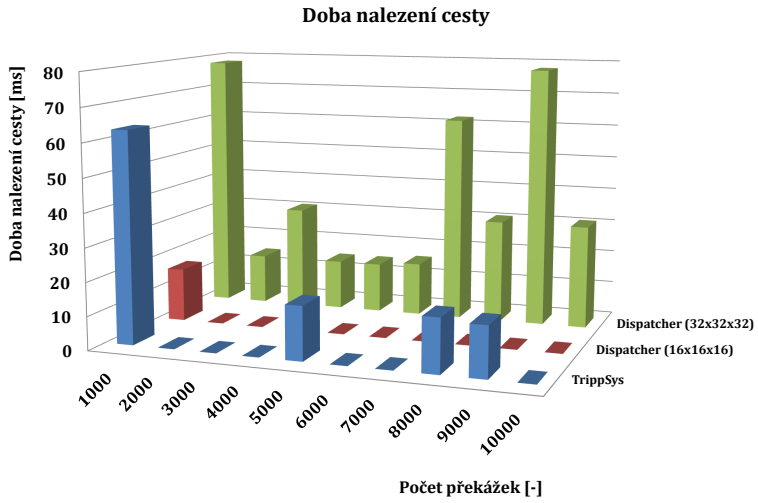
[4] V tomto případě je váha nebezpečí definována vzdáleností od nejbližší překážky. Její hodnota je rovna jedné, pokud uzel leží na obalové kouli překážky. Menší hodnoty představují větší vzdálenost, vyšší hodnoty pak signalizují, že uzel leží uvnitř obalové koule.

[5] V grafu 6.8 je pro všechny varianty použit Dijkstrův algoritmus. Váhy cest jsou tedy vždy optimální vzhledem ke grafu, který je danému systému k dispozici.

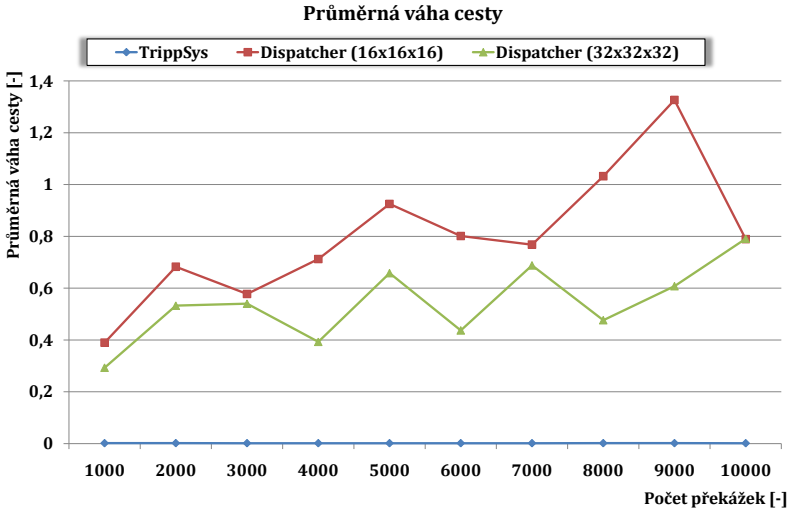
Obrázek 6.5: **TrippSystem** - doba nalezení cesty



Obrázek 6.6: **TrippSystem** - váha nalezené cesty



Obrázek 6.7: **TrippSys** a **Dispatcher** - doba nalezení cesty



Obrázek 6.8: **TrippSys** a **Dispatcher** - průměrná váha cesty
(váha cest nalezených systémem **TrippSys** není nulová, je o 1-2 řády nižší)

6.4 Různé velikosti překážek

Další testovanou vlastností popisovaných systémů je jejich závislost na velikosti překážek, mezi kterými má být hledána cesta. Grafy 6.9 a 6.10 vizualizují naměřené hodnoty systémů **TrippSys** a **Dispatcher** ve scénách s pevným počtem překážek (konkrétně 5000 náhodně rozmístěných překážek), jejichž velikost se postupně zvyšuje. Z grafu 6.9 lze vypozorovat, že ani u jednoho z popisovaných systémů nezávisí doba předzpracování na velikosti překážek. Systém **TrippSys** totiž v rámci předzpracování generuje regulární triangulaci, kde počet čtyřstěnů přímo nezávisí na velikosti překážek, zatímco **Dispatcher** mapuje překážky do trojrozměrné matice pevné velikosti, takže ani zde nemá velikost mapované překážky zásadní vliv. Ani v tabulce 6.4 a grafu 6.10 nelze pozorovat žádnou závislost na velikosti překážek. Průměrná váha cest nalezených systémem **TrippSys** byla oproti systému **Dispatcher** stejně jako v předchozí sekci o 1-2 řády nižší.

6.5 Rozmístění překážek

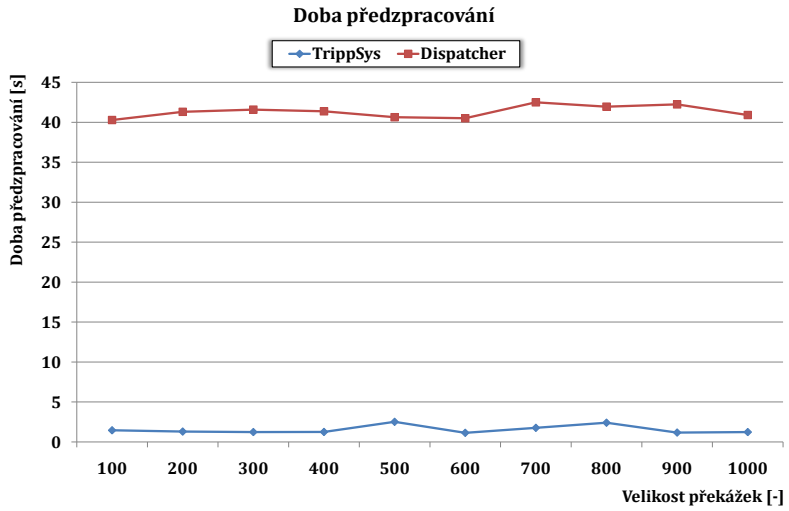
V poslední fázi analýzy byly oba představené systémy testovány ve speciálních případech rozložení překážek v prostoru. Do scény (přesněji řečeno do její mapované oblasti o rozměrech 1024x1024x1024 umístěné v počátku) byly nejprve vkládány shluky překážek tak, že tyto shluky o průměru 256 vždy sestávaly z konstantního počtu 1024 náhodně rozmístěných překážek. První shluk překážek byl vložen do počátku souřadného systému a další shluky pak byly vkládány do jednotlivých rohů mapované oblasti rohů se souřadnicemi (-512; -512; -512) a (512; 512; 512), mezi kterými se hledala cesta. V grafu 6.11 jsou znázorněny průměrné váhy cest nalezených v systémech **TrippSys** a **Dispatcher** (s použitím standardního Dijkstrova algoritmu). Jak je vidět, původní systém našel ve všech případech vždy stejnou cestu (cestu stejné váhy). Ihned po vložení prvního shluku na souřadnice (0; 0; 0) totiž došlo v systému **Dispatcher** ke zjemnění adaptivní sítě právě kolem počátku souřadného systému a byla nalezena cesta, kterou už další vkládání shluků a zjemňování adaptivní struktury v rozích nijak neovlivnilo. U systému **TrippSys** je naopak zřejmý velký skok mezi momentem, kdy byl uprostřed scény jediný shluk překážek, a následujícím krokem, kdy byl do scény vložen další shluk do jednoho z rohů mapované oblasti. V případě jednoho shluku jsou v duální struktuře regulární triangulace k dispozici pouze hrany mezi překážkami tohoto shluku, ale přidáním dalšího shluku překážek do scény dochází v duální struktuře ke vzniku nových hran, díky kterým je pak možné nalezení výhodnější cesty.

Dalším způsobem rozložení překážek v prostoru, pro který byly sledovány vlastnosti popisovaných systémů hledání cest, byl jediný shluk překážek, jehož průměr se postupně měnil. Do počátku již zmíněné mapované oblasti byl vždy umístěn shluk o určitém průměru s pevným počtem překážek. Graf 6.12 představuje průměrnou váhu cesty mezi body (-512; -512; -512) a (512; 512; 512), kterou našly systémy ve scéně s jediným shlukem 1024 překážek. Systém **Dispatcher** v tomto případě vykazuje zhoršování kvality cesty s rostoucím průměrem shluku. To je způsobeno tím, že pro malé průměry shluku překážek nachází původní řešení cestu, která tento shluk obchází. Jak je ale zvyšován průměr shluku, taková cesta postupně ztrácí na své kvalitě. U systému **TrippSys** lze naopak pozorovat rostoucí kvalitu cesty, respektive klesající průměrnou hodnotu nebezpečí v jejích uzlech. S rostoucím průměrem shluku náhodně generovaných překážek dochází ke zvyšování rozestupu mezi nimi a u cest vedoucích skrze shluk tak klesá nebezpečí definované právě nejmenší vzdáleností od nejbližší překážky.

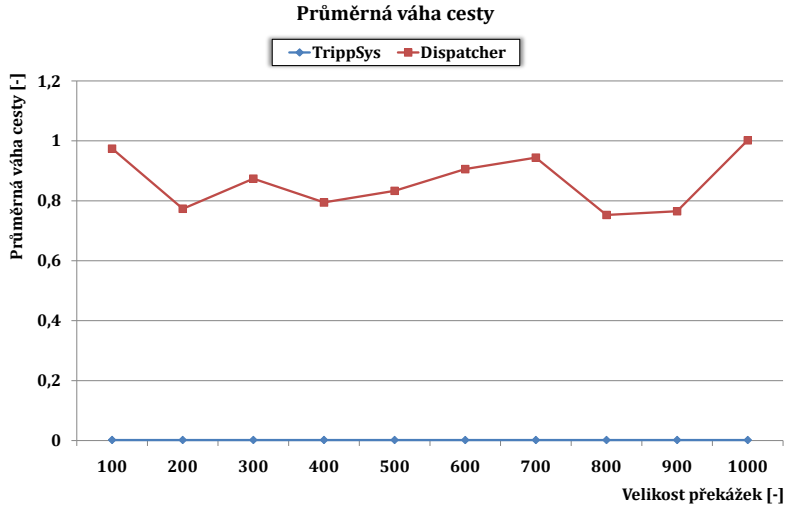
Vlastnosti obou systémů pozorované v této sekci lze shrnout do několika pozorování. Obecně lze říci, že u nového řešení **TrippSys** je výsledná cesta narozdíl od systému **Dispatcher** do určité míry závislá na topologii překážek. Především proto, že v systému **TrippSys** jsou to právě překážky, které definují množinu všech hran, nad kterými může být následně hledána cesta. Pokud překážky nejsou rozloženy rovnoměrně v rámci mapovaného prostoru, systém nemusí mít k dispozici hrany, které by vedly bezpečnějšími oblastmi. Naopak **Dispatcher** má k dispozici hrany v celém mapovaném prostoru, ale jen v oblastech s menším nebezpečím je jich méně. Přesto však nové řešení poskytuje kvalitnější cesty než původní systém **Dispatcher**. Jak již bylo uvedeno v sekci 6.3, u původního řešení se cesta hledá v rámci adaptivní struktury po hranách a diagonálách clusterů. Vzhledem k tomuto "nedostatku volnosti" pak výsledná cesta není tak kvalitní (bezpečná) jako v systému **TrippSys**, kde se hrany triangulace nachází v maximální vzdálenosti od nejbližších překážek.

	TrippSys (Dijkstrův alg.)		**Dispatcher** (Dijkstrův alg.)	
Vel. překážek [-]	Doba [ms]	Prům. váha [-]	Doba [ms]	Prům. váha [-]
100	0	0.001805208	31	0.974307883
200	0	0.001669333	47	0.773603992
300	16	0.001678923	47	0.87408289
400	0	0.001738532	0	0.794711668
500	0	0.001613457	15	0.833338829
600	0	0.001499755	0	0.905964757
700	0	0.001577041	16	0.943980884
800	31	0.001576423	47	0.752994169
900	0	0.00168283	0	0.765281741
1000	0	0.001571933	16	1.001918449

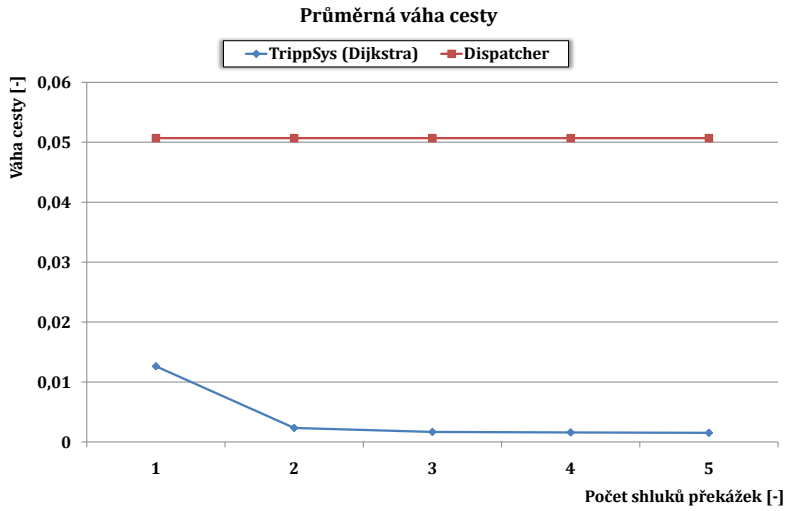
Tabulka 6.4: **TrippSys** a **Dispatcher** - charakteristiky cest pro překážky různých velikostí



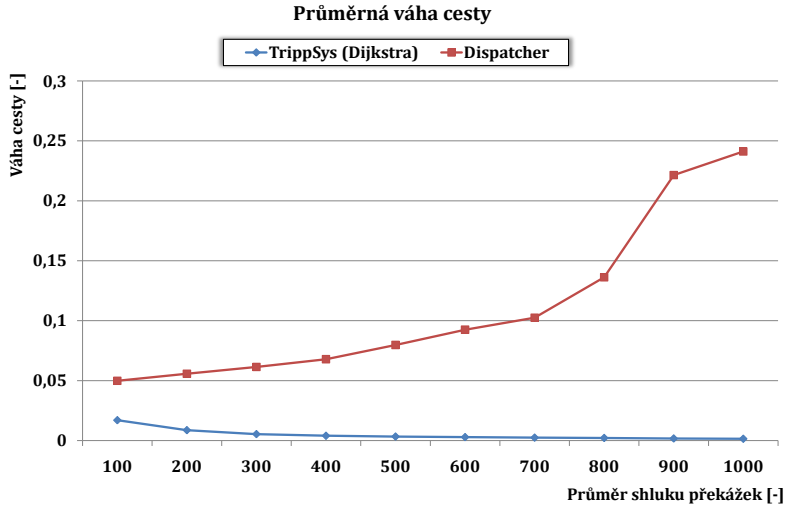
Obrázek 6.9: **TrippSys** a **Dispatcher** - doba předzpracování (překážky různých velikostí)



Obrázek 6.10: **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky různých velikostí)



Obrázek 6.11: **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky rozdělené do shluků)



Obrázek 6.12: **TrippSys** a **Dispatcher** - průměrná váha cesty (překážky v jednom shluku)

Kapitola 7

Závěr

7.1 Splnění cílů

Podle požadavků uvedených v sekci 1.1 byl navržen, implementován a otestován systém s pracovním označením **TrippSys** pro hledání cest v herních aplikacích a prostředí virtuální reality. Ten používá regulární triangulaci [Zem07] pro dělení prostoru mezi překážkami, přesněji řečeno k jejímu generování používá pozice a průměry obalových koulí překážek. Jelikož je ke konstrukci triangulace použit algoritmus inkrementálního vkládání, umožňuje systém velmi rychlou registraci nových překážek. Duální struktura triangulace je následně použita pro hledání cest v prostoru překážek, přičemž k dispozici je Dijkstrův algoritmus, vlastní suboptimální algoritmus (*gaps filling*) a A* algoritmus s možností definice vlastní heuristické funkce pro ohodnocení vrcholů grafu.

Pro představení a otestování navrženého přístupu byla připravena herní aplikace simulující průlet polem asteroidů a konzolová testovací aplikace. Oba programy využívají dynamickou knihovnu projektu **TrippSys** a pro účely testování také knihovnu původního řešení **Dispatcher**.

7.2 Výsledky

U implementace představovaného řešení byly testovány charakteristiky, které jsou důležité nebo zajímavé vzhledem ke kontextu použití tohoto systému, a v relevantních případech byly jeho vlastnosti porovnány s původním řešením, které popisuje kapitola 3. Konstrukce regulární triangulace pro počáteční množinu 2^{10} překážek vyžaduje čas přibližně 437ms a přidání nové překážky do této množiny pak trvá méně než 16ms. Pro srovnání uveďme, že původní systém **Dispatcher** vyžaduje pro stejný počet překážek, kde je polovina z nich interpretována jako statická a druhá polovina jako dynamická, přibližně 10s pro svou inicializaci a 1,5s pro adaptaci na nově přidanou překážku.

Pro různé počty překážek byl sledován také čas nalezení cesty v již zmíněné duální grafové struktuře. Navrhovaný pseudooptimální algoritmus *gaps filling* dokázal v případě 2^{16} překážek nalézt 1,1-optimální řešení v až 5-krát kratším čase oproti optimálnímu Dijkstrovu algoritmu, kterému v tomto případě trvalo nalezení cesty přibližně 78ms. Tento čas by zřejmě byl dostatečně krátký pro použití v jednotlivých iteracích programu, ale přesto je velmi důležité hledat možnosti urychlení, protože v jedné iteraci programu může být požadováno nalezení více cest, např. pro různé počítačem ovládané avatary.

7.3 Další postup

Schopnost regulární triangulace odebírat body byla v době vzniku tohoto dokumentu ve fázi testování. Hlavním úkolem v nadcházejícím výzkumu je tedy odladění této operace a její plné využití v projektu **TrippSystem**, který prozatím nabízí pouze možnost registrace nových překážek. Do další analýzy a experimentů bude začleněno také srovnání s Delaunayovo triangulací. Momentální řešení neuvažuje překážky jako takové, ale pouze jejich obalové koule. Další z možných cest při budoucím vývoji tohoto systému by tak mohlo být rozšíření, které by (pseudo)optimální cestu mezi obalovými koulemi překážek dále zpracovávalo a upravovalo podle jejich konkrétního tvaru. **TrippSystem** navíc v době vzniku tohoto dokumentu pracuje s bodovým avatarem a jeho rozměry prozatím zanedbává. Budoucí vývoj bude místo hledání samotných cest orientován spíše na hledání určitých "tunelů", ve kterých se následně může pohybovat avatar o konkrétních rozměrech.

Přehled zkratek a značení

Následující tabulka popisuje veškeré zkratky a značení použitá v této práci.

A*	Heuristický algoritmus hledání cest
ASM	Adaptive spatial memory (adaptivní struktura použitá v [AG05])
BFS	Breadth first search (algoritmus prohledávání do šířky)
CDT(P)	Delaunay triangulace s omezením nad množinou bodů P
CGT(P)	"Žravá" triangulace s omezením nad množinou bodů P
CH(P)	Konvexní obálka množiny bodů P
D*	Úprava algoritmu A* pro dynamické prostředí
DDT(P)	Datově závislá triangulace nad množinou bodů P
DFS	Depth first search (algoritmus prohledávání do hloubky)
DT(P)	Delaunayova triangulace nad množinou bodů P
G(V,E)	Graf s množinou uzlů V a množinou hran E
GT(P)	"Žravá" triangulace nad množinou bodů P
IDT	Image distance transform (výpočet potenciálového pole překážek)
MST	Minimum spanning tree (minimální kostra grafu)
RT(P)	Regulární triangulace nad množinou bodů P
T(P)	Obecná triangulace nad množinou bodů P

Literatura

[ACF01] *O. Arikan, S. Chenney, D. A. Forsyth:* **Efficient Multi-Agent Path Planning**
Computer Animation and Simulation, 2001

[AD98] *N. M. Amato, L. K. Dale:* **Probabilistic Roadmap Methods are Embarrassingly Parallel**
Technical Report 98-024, Texas A&M University, 1998

[AG05] *R. A. Apu, M. Gavrilova:* **Adaptive spatial memory representation for real-time motion planning**
Proceedings of the 8th International Conference on Computer Graphics and Artificial Intelligence, France, p.21-32 (ISBN: 2-914256-07-8), 2005

[AI Depot] *A. J. Champandard:* **Path-Planning from Start to Finish**
Internetový zdroj: www.ai-depot.com/BotNavigation

[AI*90] *G. Ausiello, G. F. Italiano, A. M. Spaccamela, U. Nanni:* **Incremental Algorithms for Minimal Length Paths**
Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, San Francisco, California, USA, p.12-21 (ISBN:0-89871-251-3), 1990

[Bro06a] *P. Brož:* **Hledání cesty pro robota/avatara ve VR**
Bakalářská práce, Západočeská univerzita v Plzni, 2006

[Bro06b] *P. Brož:* **Path Planning in Combined 3D Grid and Graph Environment**
Central European Conference on Computer Graphics, Častá-Papiernička, Slovensko, 2006

[Bro07] *P. Brož:* **Exact and Heuristic Path Planning Methods for a Virtual Environment**
Central European Conference on Computer Graphics, Budměrice, Slovensko, 2007

[BS*05] *T. Beyer, G. Schaller, A. Deutsch, M. Meyer-Hermann:* **Parallel Dynamic and Kinetic Regular Triangulation in Three Dimensions**
Computer Physics Communications 172, p.86-108, 2005

[BT98] *S. Bandi, D. Thalmann:* **Space Discretization for Efficient Human Navigation**
Computer Graphics Forum (Volume 17 Issue 3), p.195-206, 1998

[DFI01] *C. Demetrescu, I. Finocchi, G. F. Italiano:* **Dynamic Graphs**
CRC Press, 2001

[DHZ00] *D. Dor, S. Halperin, U. Zwick:* **All-Pairs Almost Shortest Paths**
SIAM Journal on Computing, p.1740-1759, 2000

[DI02] *C. Demetrescu, G. F. Italiano:* **Fully Dynamic All Pairs Shortest Paths with Real Edge Weights**
Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science, p.260, 2002

[Ebe99] *D. Eberly:* **Distance Between Point and Triangle in 3D**
Geometric Tools LLC (www.geometrictools.com)

[Epp94] *D. Eppstein:* **Finding the k-Shortest Paths**
35th Annual Symposium on Foundations of Computer Science, p.154-165, 1994

[Joe91] *B. Joe:* **Construction of Three-dimensional Delaunay Triangulations Using Local Transformations**
Computer Aided Geometric Design (Volume 8, Issue 2), p.12-142, 1991

[KS97] *P. N. Klein, S. Subramanian:* **A Fully Dynamic Approximation Scheme for Shortest Paths in Planar Graphs**
Algorithmica (ISSN:0178-4617), Springer New York, p.235-249, 1997

[Sam90] *H. Samet:* **The Design and Analysis of Spatial Data Structures**
Addison-Wesley Series In Computer Science (ISBN:0-201-50255-0), 1990

[Ski97] *S. S. Skiena:* **The Algorithm Design Manual**
Springer-Verlag, New York, 1997

[Ste94] *A. Stentz:* **Optimal and Efficient Path Planning for Partially-Known Environments**
Proceedings IEEE International Conference on Robotics and Automation, San Diego, California, USA, p. 3310-3317 (ISBN:0-8186-5330-2), 1994

[VPC02] *M. Vigo, N.Pla, J. Cotrina:* **Regular Triangulations of Dynamics Sets of Points**
Computer Aided Geometric Design 19, p.127-149, 2002

[WWZ03] *D. Wagner, T. Willhalm, Ch. Zaroliagis:* **Dynamic Shortest Paths Containers**
Electronic Notes in Theoretical Computer Science 92 No. 1, 2003

[Zem07] *M. Zemek:* **Dělení prostoru pro rozsáhlá a měnící se data**
Diplomová práce, Západočeská univerzita v Plzni, 2007

[Zit06] *P. Zítka:* **Hierarchické povrchové modely**
Diplomová práce, Západočeská univerzita v Plzni, 2006

Autorský rejstřík

Apu, Russel Ahmed, 2, 14

Bellman, Richard, 11

Dijkstra, Edsger Wybe, 9, 11, 26, 27, 38
Dor, Dorit, 11

Floyd, Robert, 11

Gavrilova, Marina, 2, 14

Halperin, Shay, 11

Warshall, Stephen, 11

Zemek, Michal, 4, 22, 30
Zwick, Uri, 11
Zítka, Přemysl, 2, 15

Rejstřík zkratek

A*, 12, 13
ASM, 14

BFS, 12, 32

CDT(P), 22
CGT(P), 22
CH(P), 22

D*, 13
DDT(P), 22
DFS, 12
DT(P), 22

G(V,E), 5
GT(P), 22

IDT, 2, 7, 14

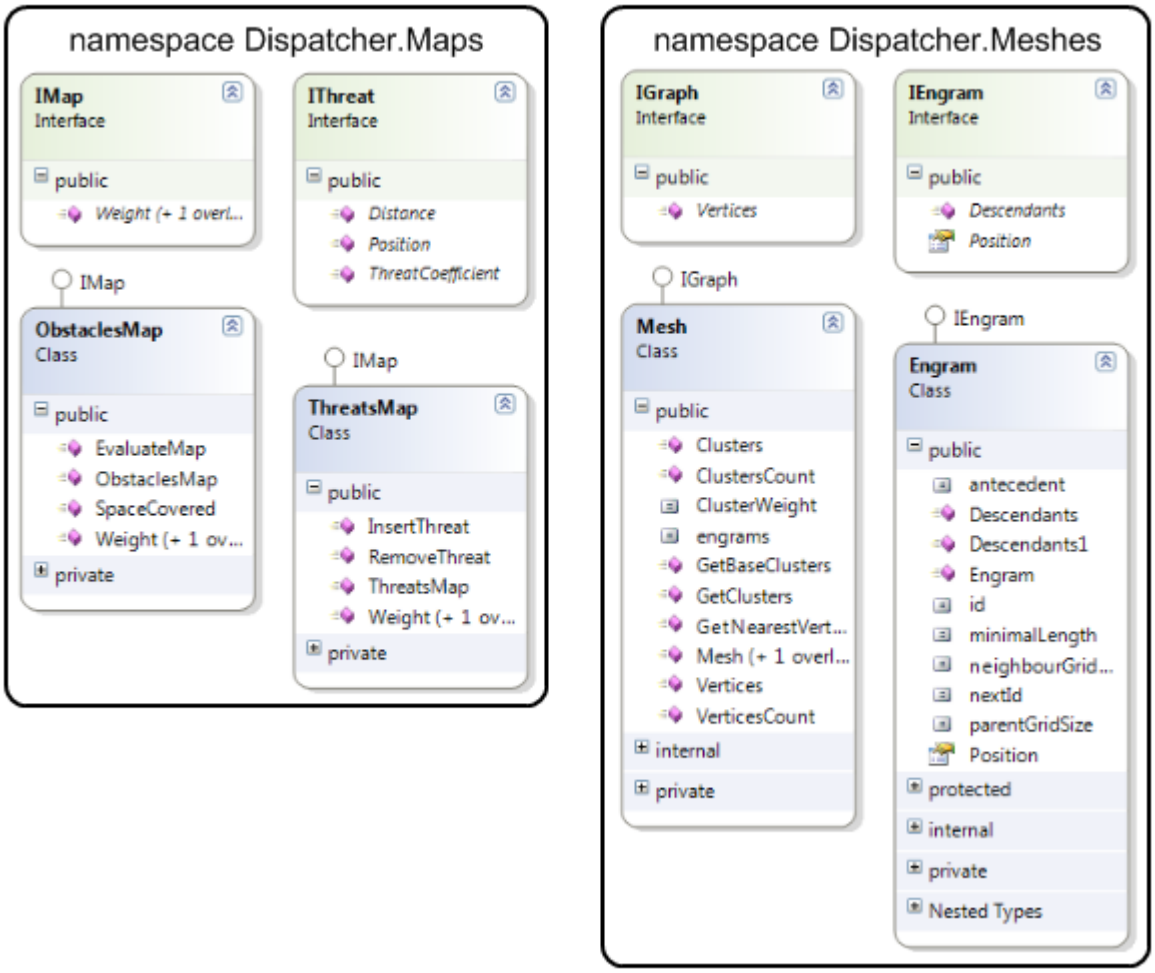
MST, 9

RT(P), 22

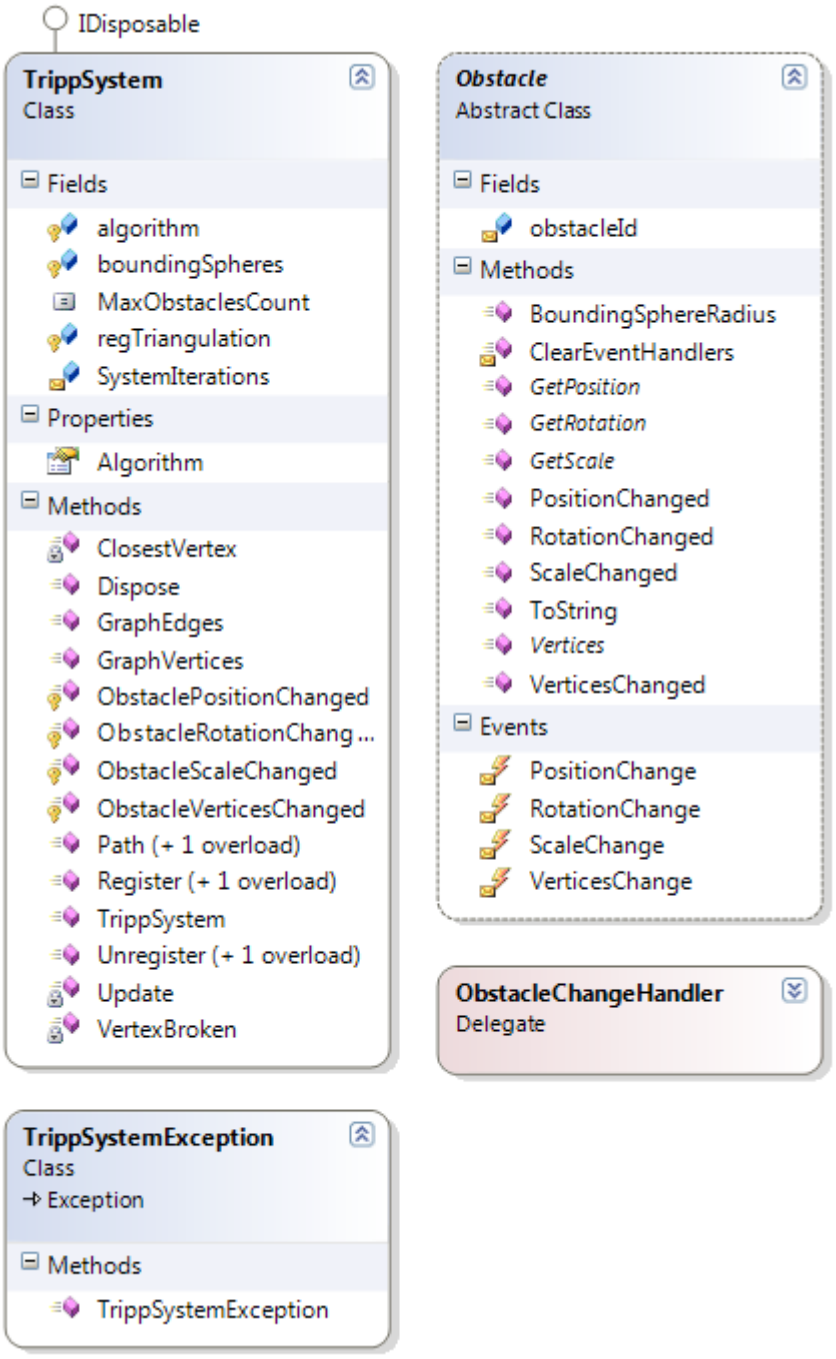
T(P), 22

Dodatek A

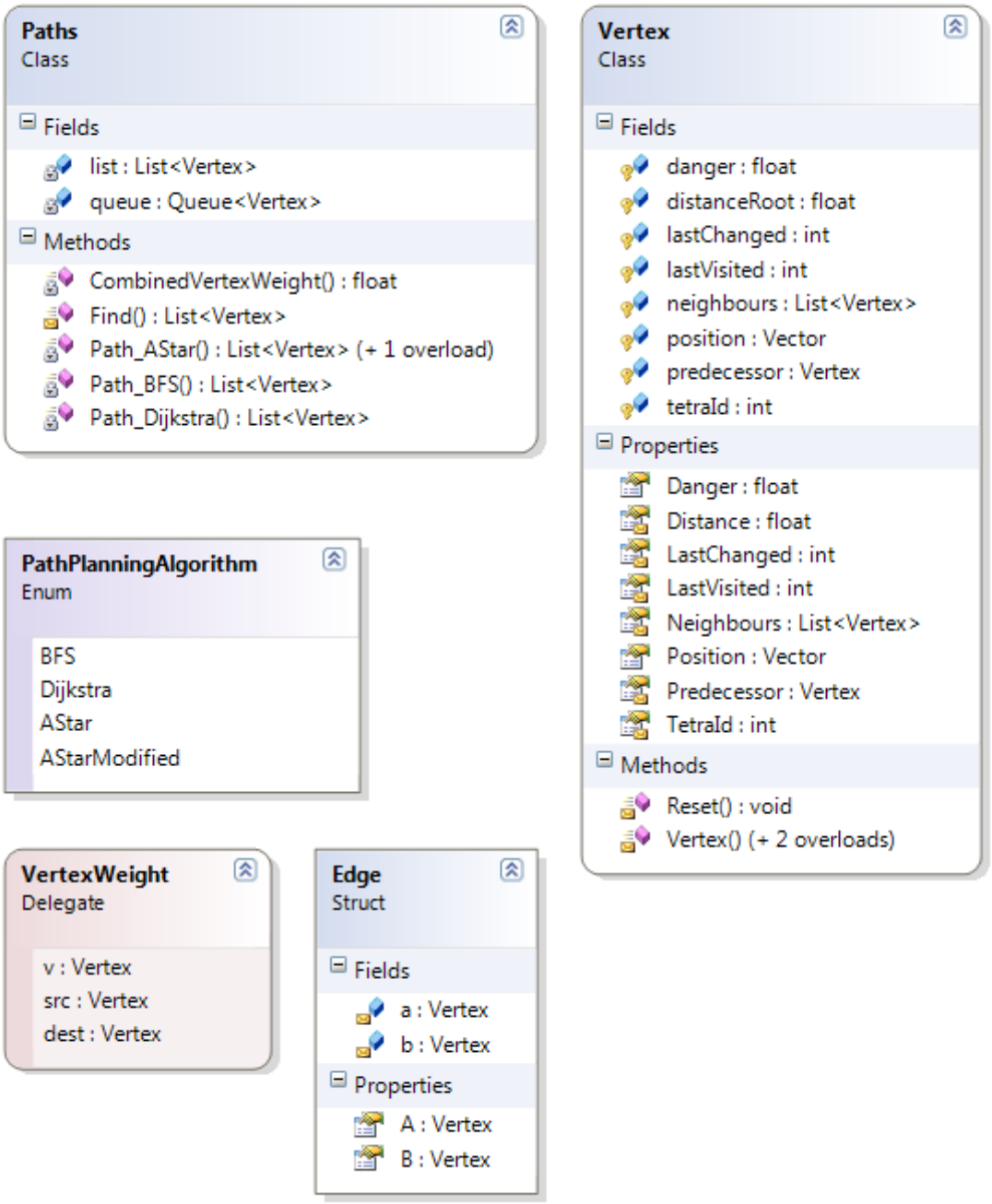
Obrázky



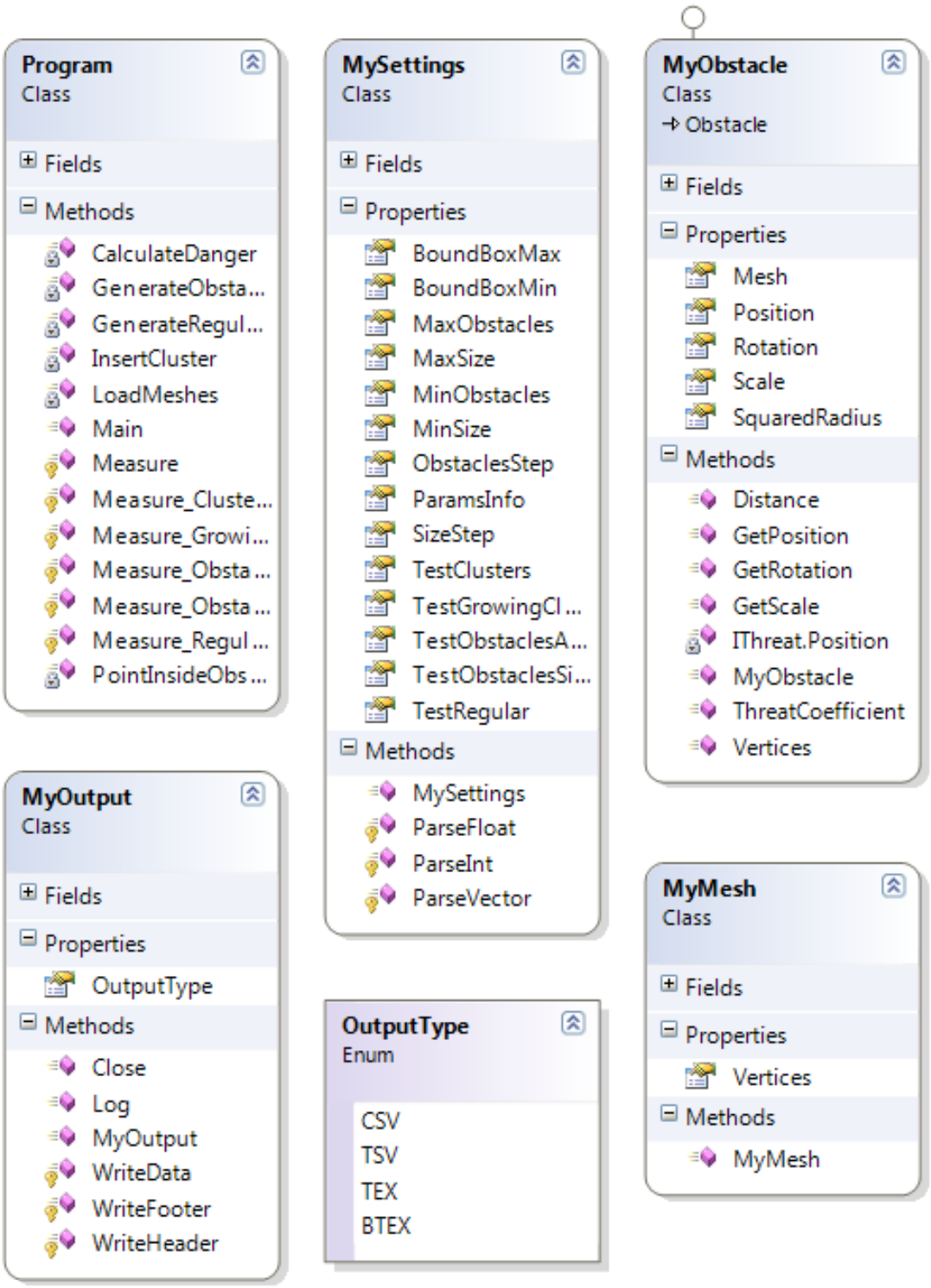
Obrázek A.1: Projekt **Dispatcher** - diagram tříd



Obrázek A.2: Projekt **TrippSystem** - diagram tříd



Obrázek A.3: Projekt **TrippSystem** - diagram grafových tříd



Obrázek A.4: Aplikace **Measuring** - diagram tříd

Obrázek A.5: Testovací aplikace **GalaxyWars** - hlavní menu

Obrázek A.6: Testovací aplikace **GalaxyWars** - nastavení

Obrázek A.7: Testovací aplikace **GalaxyWars** - ovládání

Obrázek A.8: Testovací aplikace **GalaxyWars** - power diagram 1

Obrázek A.9: Testovací aplikace **GalaxyWars** - power diagram 2

Dodatek B

Uživatelská dokumentace

Následující sekce popisují instalaci a ovládání ukázkových, resp. testovacích aplikací, které jsou přiloženy k tomuto dokumentu na CD.

B.1 Aplikace GalaxyWars

Ukázková herní aplikace **GalaxyWars** byla připravena v jazyce C# 2.0 s použitím knihoven **XNA Game Studio 2.0**[1], které zastřešují knihovny **Microsoft DirectX**[2] a přidávají další prvky užitečné při vývoji her. K jejímu spuštění je tedy vyžadována instalace základní skupiny knihoven pro **.NET Framework 2.0**[3], **DirectX**[4] a **XNA Game Studio 2.0**[5]. Zdrojové kódy aplikace je možné nalézt v adresáři `Source/GalaxyWars` a zkompilovaná release verze pro architektury x86 je připravena v adresáři `Binaries/GalaxyWars`. XML soubor `GalaxyWars.settings` uložený společně se zkompilovanou aplikací definuje její nastavení, uveďme ty nejdůležitější (veškerá nastavení pak lze měnit přímo v aplikaci, viz obr. A.6):

ResolutionWidth, ResolutionHeight určuje rozlišení obrazovky pro běh aplikace. V případě, že jsou tyto hodnoty nastaveny na nulu, je při spuštění aplikace použita automatická detekce rozlišení.

Fullscreen nabývá logické hodnoty `true` nebo `false`. Nastavením na `true` je aplikace spuštěna v celoobrazovkovém režimu a naopak.

PostScreenEffects opět nabývá logických hodnot `true` nebo `false` a lze pomocí ní zapnout nebo vypnout grafické efekty, které mají velký vliv na rychlost a plynulost aplikace.

Ovládání samotné aplikace je možné několika způsoby. Základní ovládání lze vyčíst z obr. A.7 a kompletní přehled pak představuje tabulka B.1.

[1] http://creators.xna.com
[2] http://msdn.microsoft.com/directX
[3] K dispozici na CD: `Source/dotnetfx.exe`
[4] K dispozici na CD: `Source/directx_9c_redist.exe`
[5] K dispozici na CD: `Source/xnafx20_redist.msi`

Akce	Ovládání
Orientace lodi	Pohyb myši, kurzorové klávesy
Zrychlení	Levé tl. myši, klávesa A
Zpomalení	Pravé tl. myši, klávesa Z
Zap./vyp. zobrazení pozadí	klávesa S
Zap./vyp. zobrazení cesty	klávesa P
Zap./vyp. zobrazení grafu	klávesa D
Zap./vyp. zobrazení vrcholů	klávesa V
Zap./vyp. automatické navigace	klávesa F
Přesunutí na začátek cesty	klávesa R

Tabulka B.1: Ovládání aplikace **GalaxyWars**

B.2 Aplikace Measuring

Measuring je čistě konzolová aplikace, která slouží k analýze a testování představeného systému bez grafického výstupu (výsledky analýzy jsou vždy zaznamenány do souborů). Ovládat ji lze předáním parametrů při spouštění. Výčet akceptovaných parametrů je uveden v tabulce B.2, přičemž vektor může být zadán jako <x;y;z>, (x;y;z) nebo [x;y;z]. Následující příklad provede test systémů **Dispatcher** a **TrippSys** pro různé počty překážek (1000 - 5000 překážek s krokem 100) a dále pro různé velikosti překážek (ponechané původní hodnoty) v mapované oblasti se středem v počátku a s rozměry 256x256x256:

```
Measuring.exe amount size minObstacles=1000 maxObstacles=5000
obstaclesStep=100 bBoxMin=(-128;-128;-128) bBoxMax=(128;128;128)
```

Zdrojové kódy aplikace je možné nalézt v adresáři `Source/Measuring` a zkompilovanou verzi pro architektury x86 lze najít v adresáři `Binaries/Measuring`.

Parametr	Význam
amount	Test závislosti vlastností systémů na počtu překážek (výsledky jsou uloženy do souboru `obstacles-count.csv`)
size	Test závislosti vlastností systémů na velikosti překážek (výsledky jsou uloženy do souboru `obstacles-size.csv`)
clusters	Test závislosti vlastností systémů na shlukové topologii překážek (výsledky jsou uloženy do souboru `cluster-obstacles.csv`)
growingcluster	Test závislosti vlastností systémů na topologii překážek s jedním rostoucím shlukem překážek (výsledky jsou uloženy do souboru `growing-cluster.csv`)
bBoxMin=*vektor*	Minimální hodnoty pro náhodné rozmístění překážek (bez jeho zadání je použit vektor (-512, -512, -512))
bBoxMax=*vektor*	Maximální hodnoty pro náhodné rozmístění překážek (bez jeho zadání je použit vektor (+512, +512, +512))
minSize=*celé_číslo*	Minimální hodnota pro náhodné rozměry překážek (bez jejího zadání je použita hodnota 100)
maxSize=*celé_číslo*	Maximální hodnota pro náhodné rozměry překážek (bez jejího zadání je použita hodnota 1000)
sizeStep=*celé_číslo*	Kroková změna rozměrů překážek (bez jejího zadání je použita hodnota 100)
minObstacles=*celé_číslo*	Minimální počet generovaných překážek (bez jeho zadání je použita hodnota 1000)
maxObstacles=*celé_číslo*	Maximální počet generovaných překážek (bez jeho zadání je použita hodnota 10000)
obstaclesStep=*celé_číslo*	Kroková změna počtu překážek (bez jeho zadání je použita hodnota 1000)

Tabulka B.2: Ovládání aplikace **Measuring**

Dodatek C

Combined environment representation in a new path planning approach

ACM Student Research Competition 2006
Praha, Česká republika

Combined environment representation in a new path planning approach

Petr BROŽ

FAV, Západočeská univerzita v Plzni,
Univerzitní 22, 306 14 Plzeň
pebro@students.zcu.cz
bakalářský projekt

Vedoucí práce: Doc. Dr. Ing. Ivana Kolingerová

Abstract. Solutions for the common problem of path planning in an abstract environment have been extensively developed in many scientific disciplines. However, almost all explored techniques assume the environment is static and completely known. In our proposal, we introduce a model for the real-time path planning in a known, unknown and dynamic environment. Our hybrid technique combines a graph and grid representation of the examined space while preserving the advantages from both ways of the representation. As the result, the method offers faster path retrieval then the classical raster approaches in a dynamic environment where it is impossible to apply the conventional graph based techniques.

Keywords: path planning, adaptivity, virtual reality, computer graphics.

1 Introduction

The first methods for finding an optimal path in an abstract environment were developed even before the computer science appeared. Therefore, there are many fast and efficient techniques for solving this general task. They are often subdivided according to the representation of the environment they are able to work with. Some algorithms demand a graph-like definition (e.g., geometrical definition of all obstacles) and other algorithms assume the discrete representation is available.

The conventional algorithms have considerable disadvantages when they are to be used in the real applications. The graph representation of the real environment is rarely available and its construction is very difficult. The discrete representation of the examined space is much easier accessible but the algorithm itself is in most cases very time-consuming. In addition, almost all methods for the path finding and planning need well-known or static environment which is not often available, either.

In this paper, we combine an adaptive mesh to define all available transitions for searched paths and a simple 3D matrix to store all raster based values. As the result, we provide a technique that is faster than the raster based approaches and suitable for the dynamic environment (preliminary version of our method, without the adaptivity, has been published in [1]).

2 State of the art

Path planning in general denotes a basic problem of finding an optimal path between two specified spots in an abstract environment representation. In this context, the optimal path means the path satisfying one or more given objectives (the shortest, the cheapest or the fastest path, etc.). The abstract environment can be represented in a variety of ways but the algorithms are focusing mainly on evaluated graphs and grids. There are many ways these environments can be distinguished (dynamic/static, known/unknown environments, etc.) which implies a similar distinction of the path planning techniques.

2.1 Graph based methods

The Visibility graph [3] technique extends the provided definition of the environment with the edges connecting the points that can "see" each other. The new edges (and the edges defining the sides of each obstacle) then represent the possible transitions for the optimal path retrieval. An example of such a preprocessing can be seen in Figure 1 on the left.

The Minkowski sum [4] technique is a similar approach that considers the shape of the passing object and „inflates" the borders of obstacles so that the collision-free path can be solved. An example of this approach is presented in Figure 1 on the right.

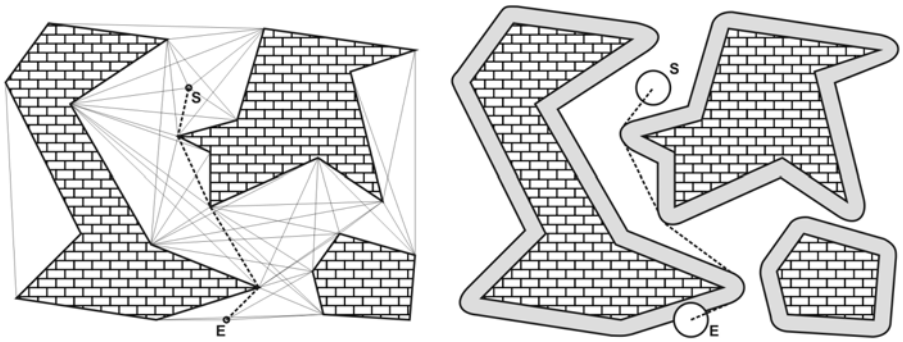


Figure 1: Examples of a scene processed with the Visibility graph technique (on the left) and Minkowski sum technique (on the right)

2.2 Raster based methods

A grid representation can be precomputed or modified at the beginning of the algorithm. In reference to the modification of the explored grid, the **potential field** model [5] can be used for filling the grid with the discrete values of a specific potential field generated by all obstacles. Passing through the grid elements with the lowest potential values then ensures finding the path with the maximal clearance among all obstacles. The most known techniques for searching itself are for example the **A*** and the **D*** (for the changing environment) algorithms.

3 The proposed model

In the proposed solution, we come out of a general idea of a fictive terrain exploration with the help of autonomous robots. These robots (also called scouts or agents) survey certain locations of the examined terrain according to the orders from certain headquarters (HQ). Headquarters then keep specific "paper maps" to sketch in the discovered obstacles and other threats which are then periodically complemented and updated with the actual values measured by the scouts. After certain time, the static obstacles are fully mapped throughout the explored space, the safest paths (in terms of the maximal clearance among all obstacles) are known and the scouts are then guided only to the locations with a suspicion of the possible threats.

Following the mentioned idea of the sensor based terrain, we advance in the development of a general model for the real-time and adaptive path planning that was pioneered by R. A. Apu in [2]. The proposed model can be used for both 2D and 3D applications and works in a complex and dynamic environment which is assumed to be provided in the raster representation and can be well known, partially known or even unknown. The described path planning system is based on the following main headstones:

- A graph-like spatial structure (hereafter referred to as a **mesh**) that adapts itself to the examined environment and defines all reachable positions and transitions with its vertices and edges.
- A grid structure for the discrete representation of certain environment hazards (hereafter referred to as a **map**), e.g., the proximity to an obstacle or the dynamic threats.

The main approach uses two separate maps of the same size for the environment description. The first one, called **obstacles map**, represents the danger weights as the proximities to the nearest obstacle in the mapped space and the second one, called **threats map**, represents the potential field generated by all located and observed threats in the examined terrain. In the following, the proposed model keeps a mesh that is „widespread" over the examined space covered also by the mentioned mapping structures. This mesh defines all available waypoints and transitions for the path retrieval and continuously copes with the changes in the mapping structures. Such an adaptation is achieved by refinement of the mesh in the places with higher error values (calculated from the obstacles map and threats map) and by merging of the mesh in the unimportant areas.

The algorithm itself is based on the real-time development of the adaptive mesh during the particular iterations. According to the presently recorded values in the maps, the mesh structure is to be refined in the areas with the higher importance and merged in the areas with the lower importance. In the proposed solution, the adaptive mesh is used only to define the available waypoints and transitions for the path retrieval, not for the visualization. Therefore, T-vertices in the mesh do not bring any problems typical for them in the visualisation of the meshes (they may cause creases in the model). Foldovers in the mesh are not possible in our case as the vertices are not moved, just refined.

In our current solution and demonstrating application, we assume the obstacles in the environment are already completely explored – the obstacles map is filled with the IDT (Image Distance Transform) technique based on the Voronoi diagrams [6]. Concretely, the elements of the obstacles map are evaluated according to their proximity to the nearest obstacle with the real value from 0 (minimal proximity) to 1 (maximal proximity to the nearest obstacle or the obstacle itself). The elements of the threats map are then evaluated in a similar manner during the mesh adaptation.

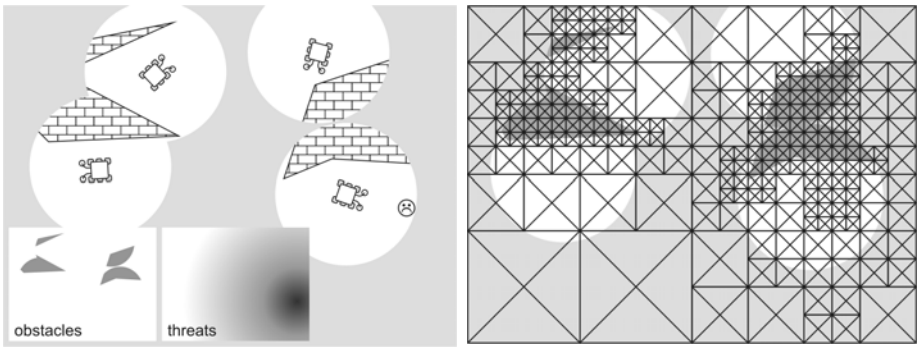


Figure 2: An example of the 2D sensor-based terrain exploration and corresponding mesh adaptation

4 Experiments & results

We have implemented a simple application (in the C# language with the Direct3D libraries) to provide the results and comparisons of our algorithm. In this demonstrating application, we generate a certain number of obstacles formed by the solid spheres with different radii and we also add some abstract threats represented by the small red cubes. During the program run, the threats are directed to the random locations of the examined space and the adaptive mesh is refined in each iteration. The obstacles map is precomputed and filled with the corresponding values before the main loop of the program. To demonstrate the adaptivity of the algorithm, the optimal path is recomputed after each refinement of the mesh.

4.1 Survey of the tests

For our survey, we have selected and measured the following variables as the most characteristic and important parameters of the proposed method:

- **Clusters amount** defines the count of the atomic elements in the adaptive structure (in Figure 2, these are the smallest squares with all corners connected to each other).
- **Adaptation time** determines the time needed for a single iteration of the adaptive structure progression.

- **Path finding time** denotes the time needed for finding the optimal path between two constant spots in the opposite corners of the explored space.

The parameters have been measured with a number of testing datasets to present the qualities and possible weak points of our implementation. These datasets are desrcibed in the legend of each plot. The letter 'O' marks the obstacles count, letter 'G' indicates rank of the grid and the letter 'D' stands for the maximum level of the mesh division.

First of all, we present the functional dependence of the clusters amount on the particular iterations of the program in Figure 3. At the very beginning of the main loop, we can see a rapid growth of the clusters amount as the adaptive mesh refines itself around the obstacles. The remaining behaviour highly varies due to the movement of the threats in the scene. When the threat shifts off from the near obstacles, it raises the weight value of the previously unimportant locations and so evokes a new refinement of the mesh around these locations.

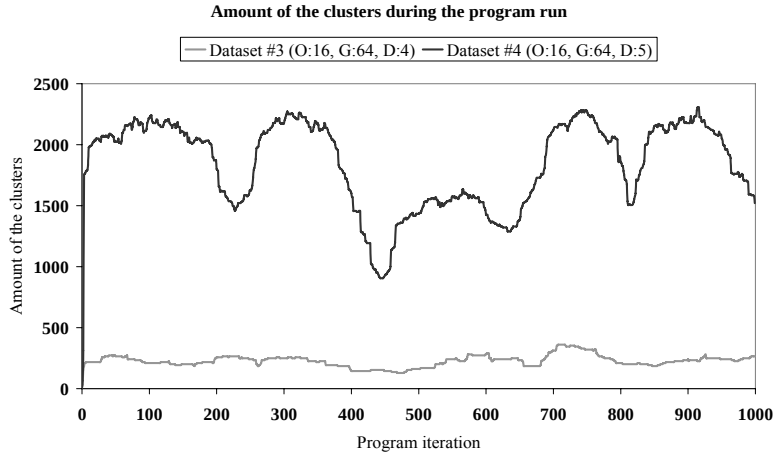


Figure 3: A clusters count dependent on the program iterations

The environment of the testing application changes itself in an absolutely random way as the threats are directed to the random locations in it. It is still possible to discover some events in the application from the presented dependencies. We can take a look for example at the graph for the dataset 4 in Figure 3: the growth in the graph (during the iterations 0-50, 250-300, 450-500, 650-750 and 800-900) are due to the threats that move away from the nearest obstacles and so invoke the mesh refinement in the previously unimportant locations. In Figure 3, we can also see the dataset 4 is much more varying than the dataset 3 but there is a clear explanation for it. With the higher level of maximum mesh division, there are more clusters reacting on the threats movement.

4.2 Obstacles count independence

We have measured the properties of our solution for 256, 512, 1024 and 2048 obstacles randomly dislocated throughout the space. Figure 4 shows the clusters amount and Figure 5 shows the path finding times for these new configurations. The graphs indicate that the qualities of our method are not directly dependent upon the obstacles amount. The differences in these graphs are caused by the obstacles topology in the scene that is randomly generated for each dataset.

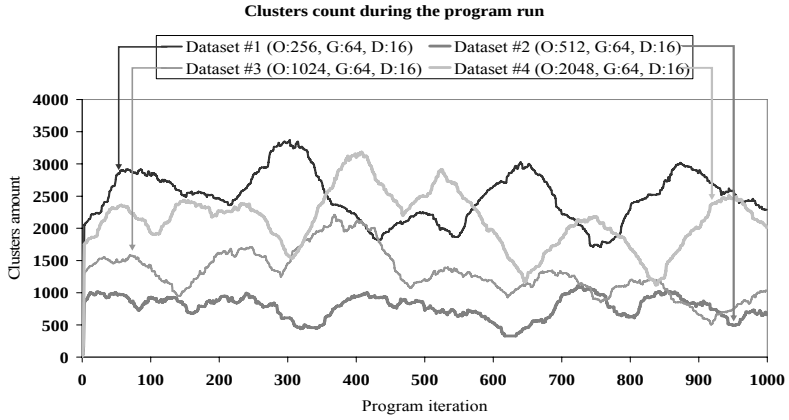


Figure 4: Clusters counts in the particular iterations

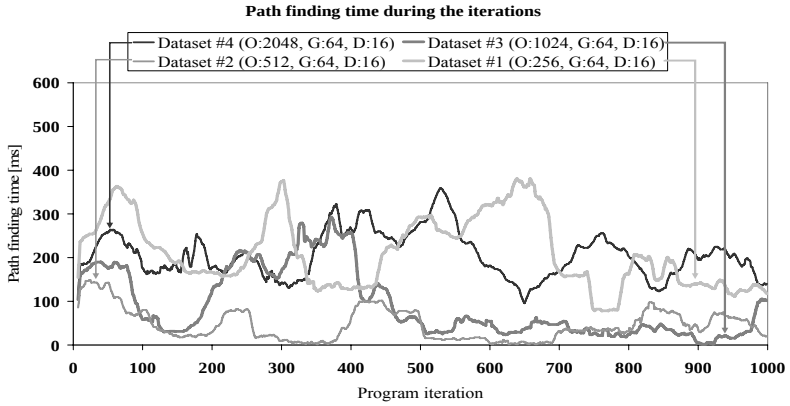


Figure 5: Path finding times dependent on the particular iterations of the program

4.3 Non-adaptive mesh comparison

Figure 6 compares the results to the two equivalent adjustments of our old algorithm with the regular mesh (in this case, 'G:64' means the regular mesh consisting of 64x64x64 clusters). The times for the regular mesh stay around the value 400ms whereas the times for finding the path in the adaptive mesh strongly vary but they never achieve the time needed by the old algorithm. Figure 7 then presents the values measured simultaneously with the path planning times for Figure 6 (values are interpolated by the polynomial regression of the third degree) and demonstrates the identical results in the path optimality for these techniques. In this context, the path optimality is evaluated according to the highest danger weight in the waypoints on the found path where this optimality grows with the descending maximal weight (Figure 7 shows this maximal weight during the program run).

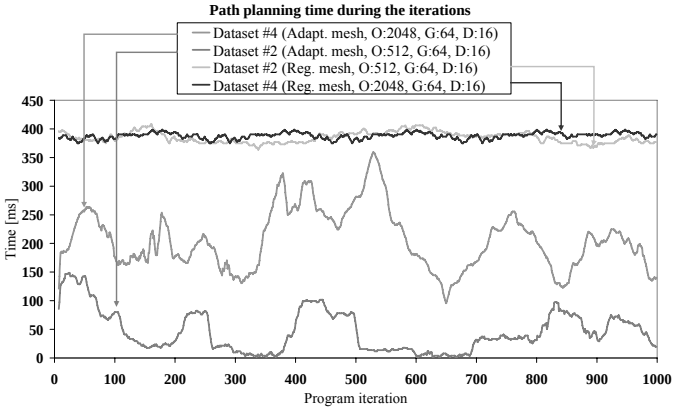


Figure 6: Path planning times dependent on the particular iterations of the program

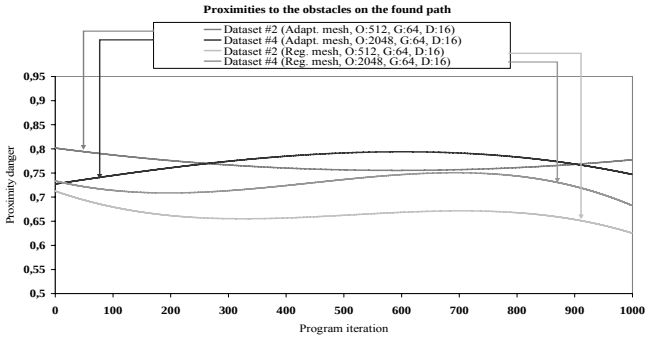


Figure 7: Proximities to the nearest obstacles on found paths during the program iteration

The measurements show the proposed path planning method is a suitable alternative for the path planning in dynamic environments: it is faster than the raster based approach and it is usable in the applications where the graph based techniques fail. On the other hand, there are still high memory requirements due to the 3D matrix for the grid used in our solution.

5 Conclusion

We have outlined the possible model for the path planning system that eliminates the described disadvantages of the conventional approaches applied in the virtual reality. We have measured the most important properties of our implementation and provided the gained dependencies, some of them compared to the results of the conventional approach. The provided method can be used in 2D and 3D applications and works in the known, partially known or unknown and dynamic environment. In comparison with the regular mesh, the method with the adaptive mesh needs only about 10-50% of the original time for finding the optimal path (while the optimality results are equal). The more detailed description of this method can be found in a scientific paper "Path planning in dynamic environment using an adaptive mesh" submitted to the VRST conference in Limassol, Cyprus, 2006. The proposed solution is still under development and there are many possible ways to improve this path planning model.

References

Referred papers:

1. Brož, P.: Path planning in combined 3D Grid and Graph Environment. *Proceedings of the 10th Central European Seminar on Computer Graphics* (2006).
2. Apu, R.A., Gavrilova, M.: Adaptive Spatial Memory Representation for Real-Time Motion Planning. *Proceedings of the 8th International Conference on Computer Graphics and Artificial Intelligence* (2005).
3. Hershberger, J.: Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size. *Annual Symposium on Computational Geometry, Proceedings of the third annual symposium on Computational geometry* (1987).
4. Ramkumar, G.D.: An Algorithm to Compute the Minkowski Sum Outer-face of Two Simple Polygons. *Annual Symposium on Computational Geometry, Proceedings of the 12th annual Symposium on Computational Geometry* (1996).
5. Warren, C.W.: Multiple Path Coordination using Artificial Potential Fields. *Proceedings of the IEEE International Conference on Robotics and Automation* (1990).

Referred books:

6. O'Rourke, J.: *Computational geometry in C* (2nd edition). Cambridge University Press, 1998.

Dodatek D

Path planning in combined 3D grid and graph environment

Central European Conference on Computer Graphics 2006
Častá - Papiernička, Slovensko

Path planning in combined 3D grid and graph environment

Petr Brož[1]

Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic

Abstract

In research literature and many scientific disciplines, solution to the common problem in path planning for an autonomous robot has been extensively developed. Almost all explored techniques assume the robot has complete and detailed overview of the environment he is moving in. In addition to, many methods work over the graph representation of this environment which can be very difficult to construct or obtain in the real applications. This paper introduces a hybrid technique combining graph and grid representations of an examined space and capable of planning paths in known, partially known, unknown and dynamic environment at the price of the pseudo optimality of results.

1 Introduction

General problem of finding and planning of an optimal path is a highly explored topic in several scientific areas. There are many approaches and techniques for solving this task. They are in most cases based either on graph or grid representation of the examined environment. In other words, some algorithms for path planning demand graph-like geometric definition of the processed scene (e.g., definition of all obstacles and forbidden areas) and other algorithms assume the discrete representation of the surrounding environment is provided. Most of these techniques generally do not distinguish the dimension of examined scene - they can be used either in 2D or 3D applications without any difficult modifications. Graph based approaches usually derive special structures from the provided environment description and work with them whereas the raster based approaches usually do not need such pre-processing and search the path directly in the provided grid.

Both ways of environment representation have crucial and radical disadvantages. Graph representation of a real environment is rarely available and its construction is - if possible at all - very complicated and difficult.

On the other hand, discrete representation of the examined space is much easier accessible and measurable but the algorithm itself is in most cases (due to the amount of raster elements to inspect) very time-consuming. In addition, almost all methods for path finding and planning need either well-known or static environment which is not always available, either.

A great improvement for this type of applications can be achieved with the combination of discrete and graph environment approaches. Such a technique could use adaptive spatial structure as a graph with vertices and edges evaluated according to the values from the provided grid. Then it would be able to discover pseudo optimal path (optimal among all available transitions in the graph) and, for example, continuously adapt this spatial structure to the actual state of environment and other dynamic influences.

In this paper, we propose a possibility for path planning over the combined environment representation which eliminates (or at least reduces) the disadvantages of mentioned conventional approaches at the price of the pseudo optimality of results. The content of the paper is as follows. Section 2 explains state of the art together with the best known techniques. Section 3 describes the proposed path planning model and in the section 4, our actual solution and implementation is outlined. Section 5 shows the results gained by our solution and in section 6, the future work of the proposed path planning approach is presented.

2 State of the art

Path planning denotes a basic problem of finding an optimal path between two specified spots in an abstract environment representation. In this context, optimal path means a path satisfying one or more given objectives (the shortest, the cheapest or the fastest path, etc.). Environment can be represented in a variety of ways but the path planning algorithms are focusing mainly on evaluated graphs and grids. There are many ways these environments can be differentiated (dynamic/static or known/unknown environments, etc.) which implies a similar distinction of path planning techniques according to the types of environment they are able to work with.

[1]pebro@students.zcu.cz

First, let us introduce the approaches based on the graph representation of the surrounding environment. **Visibility graph** technique [Her87] extends the basic provided graph with edges connecting vertices that can "see" each other whereas the source and destination position is treated as an obstacle, too. New edges (together with edges defining sides of each obstacle) then represent possible transitions and through them, the optimal path can be found. Example of such pre-processing in 2D application can be seen in Figure 1: edges of all obstacles (bricks pattern filling), starting and ending position (points labelled S and E) are connected according to their mutual visibility and over possible transitions (thin lines and obstacles sides), the optimal path (dashed lines) is selected.

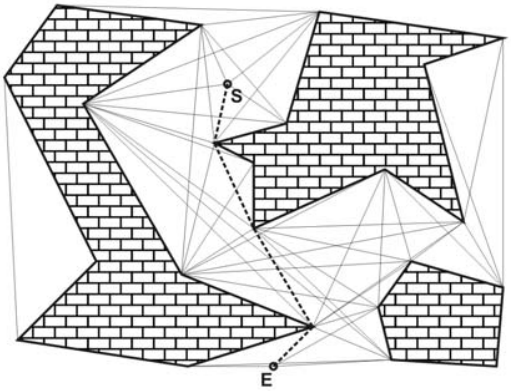


Figure 1: An example of scene processing with the "Visibility graph" technique

Minkowski sum [Ram96] is a similar approach that (unlike the previous method) considers the shape of passing object and "inflates" borders of obstacles so that the collision-free path can be solved. Example of such pre-processing is presented in Figure 2: the same obstacles as in Figure 1 are inflated with the radius of obstacle (gray areas) and the collision-free path (dashed lines) between starting and ending position (spheres labelled S and E) is selected. With the special structure prepared, both approaches can use **Dijkstra's algorithm** [DPV04] or similar to find the appropriate path.

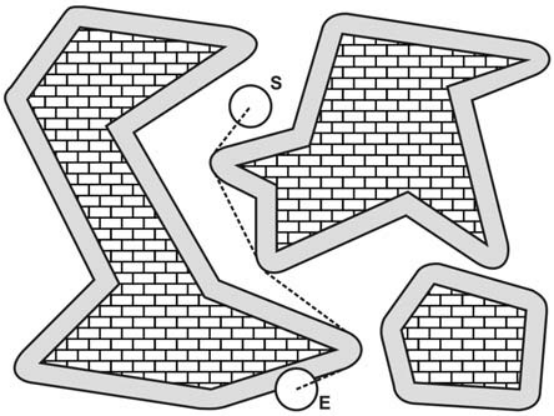


Figure 2: An example of path planning with the "Minkowski sum" method

Second, we are providing insight into the techniques based on the grid representation. Such a grid can be precomputed (if not provided) or modified at the beginning of the algorithm. In reference to the modification of the explored grid, a **potential field** model [War90] can be used for filling the grid with discrete values of a specific potential field created by all obstacles – passing through the grid elements with the lowest potential values then ensures finding the path with the maximal clearance among all obstacles. Most known techniques for searching itself are for example **A*** (for well-known environment; [Bat04]) and **D*** (for unknown, partially known or changing environment; [Ste94]) algorithms. Figure 3 documents the manner of such path finding in the grid: obstacles from Figures 1 and 2 are now splitted into the grid and in this grid, optimal path between starting and ending position (cells labelled S and E) over the grid cells is illustrated.

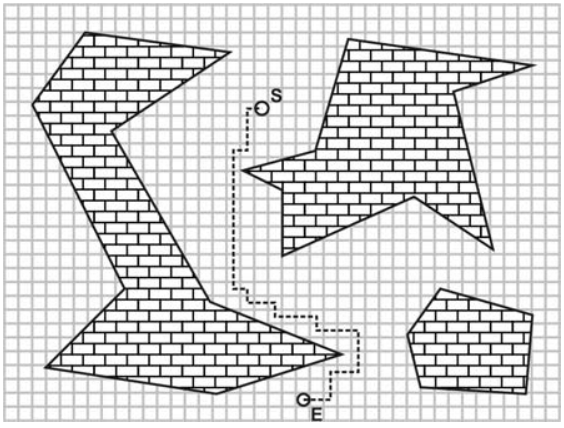


Figure 3: An example of the raster based path planning

3 The proposed solution

To provide a suitable method for the applications where the mentioned techniques fail, we are focussing on a general path planning technique that works in the known, partially known or unknown discrete environment and is designed for the virtual reality with the support of the exploring avatars.

In the proposed solution, we come out of a general idea of a fictive terrain exploration with the help of autonomous robots that are controlled from a specific kind of headquarters (HQ). These robots (also called scouts or agents) are equipped with specific sensors (dependent on the type of the application) and explore certain locations of the examined terrain according to the orders from HQ. Such headquarters keep specific „paper maps“ to sketch in the discovered obstacles and other threats which are then periodically complemented and updated with actual values measured by the scouts. Agents are then guided to the unexplored locations or to the important locations according to the actual state of these maps. After certain time, the static obstacles are fully mapped throughout the explored space, the safest paths (in term of the maximal clearance among all obstacles) are known and the scouts are then guided only to locations with a suspicion of possible threats. Figure 4 represents an example of such environment exploration in 2D application: 4 agents in the terrain collect and send the information about the obstacles (bricks pattern filling) and specific kinds of threats (angry face) to the headquarters and there, the measured values are logged into the obstacles map (impassable areas) and into the threats map (a potential field of discovered threat).

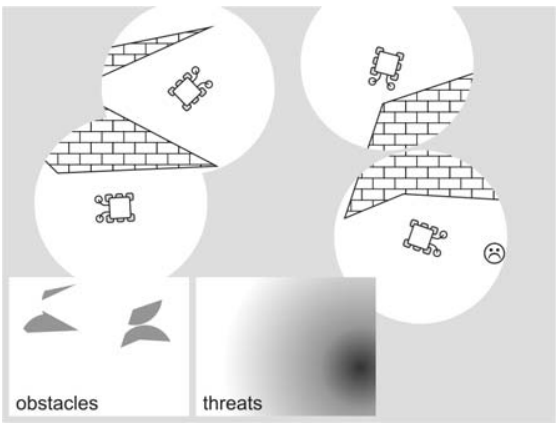


Figure 4: Preview of 2D terrain sensor-based exploration with the autonomous robots

Following the mentioned idea of the sensor based terrain exploration with the autonomous agents, we advance in the development of a general model for the real-time and adaptive path planning that was pioneered by R. A. Apu in [AG05]. The proposed model can be used for both 2D and 3D applications (the only difference lies in the undermentioned adaptive graph-like structures) and works in a complex and dynamic environment which is assumed to be provided in the raster representation and can be well known, partially known or even unknown. The described path planning system is based on three main headstones:

- A graph-like spatial structure (hereafter referred to as a **mesh**) that adapts itself to the examined environment and defines all available positions and crossings with its vertices and edges.

- A grid structure for discrete representation of certain environment hazards (hereafter referred to as a **map**), e.g., proximity to an obstacle or dynamic threats.

- An autonomous AI entity (hereafter referred to as an **agent**) for the real-time space exploration and influencing the mesh adaptation with its behaviour.

The main approach uses two separate maps of the same size for the environment description. The first one, called **obstacles map**, represents danger weights as proximities to the nearest obstacle in the mapped space and the second one, called **threats map**, represents potential fields of all located and observed threats in the space. In the following, the algorithm keeps a mesh that is „widespread over each map“ and defines all available paths the agents can travel during their exploration. This mesh continuously copes with the changes in both maps and with behaviour of all agents. Such an adaptation is achieved by refinement of the mesh in the places with higher error values (calculated from the obstacles map and threats map) and by merging of the mesh in the least visited and unimportant places.

The whole algorithm is based on real-time development of the adaptive mesh in particular iterations. According to the recorded values in the maps, mesh structure is refined in the locations with a higher importance (the darker locations in the obstacles map and the threats map in the Figure 4) and it is merged in the places with a lower importance (in the least visited graph vertices). In the proposed path planning system, the adaptive mesh is used only to define the available waypoints and transitions for the movement and navigation of the agents, not for visualization. Therefore, T-vertices in the mesh do not bring any problems typical for them in the visualisation of meshes (they may cause creases in the model). Foldovers in the mesh are not

possible in our case as vertices are not moved, just refined.

In the mentioned fictive application, continuous prospecting of the environment was a task of the robots but in our approach and demonstrating application, we assume the obstacles in the environment are completely explored - the obstacles map is filled with weights at the beginning of the algorithm with an IDT (Image Distance Transform) technique based on the Voronoi diagrams [Rou98]. Concretely, the elements of the obstacles map are evaluated according to their proximity to the nearest obstacle with the real value from 0 (maximal proximity) to 1 (minimal proximity or the obstacle itself). The elements of the threats map are then evaluated in a similar manner during the mesh adaptation.

One iteration of the mesh adaptation in the fictive application consists of the following general steps (similar as in [AG05]):

1. *Maps completion and updating*
   The current sensor readings are evaluated in the close neighbourhood of each agent and the corresponding map elements are updated or eventually complemented with the measured values.

2. *Influence depletion and replenishment*
   An importance of the recorded values (so called **influence**) of each vertex in the adaptive mesh is partially depleted and then again partially replenished according to the count and distance of the agents near this vertex. The more agents are in the proximity of the vertex, the bigger is the amount of the influence replenishment.

3. *Error function evaluation and refinement*
   The specific error function with the values from the obstacles map, threats map and influences is evaluated for each block (in [AG05], the blocks are called **engrams** in a specific spatial structure ASM – Adaptive spatial mesh) of the adaptive mesh and according to the result, the blocks are merged, splitted or left. Figure 5 shows a single stage of the adaptive mesh for the 2D scene presented in the Figure 4: the mesh is refined in important regions (above the obstacles and nearby the threat) and coarsened in less important or unexplored regions.

4. *Orders execution*
   Each agent executes its orders – he finds an optimal path to the goal position with the provided cost function or follows already computed waypoints (if the path cannot be travelled due to the refinements of the mesh, the path is recomputed to the last waypoint).

5. *Exploration*
   If all goals are reached, agents ensure an exploration of unvisited locations in the examined space – they automatically plan the path to the vertices with no values recorded.

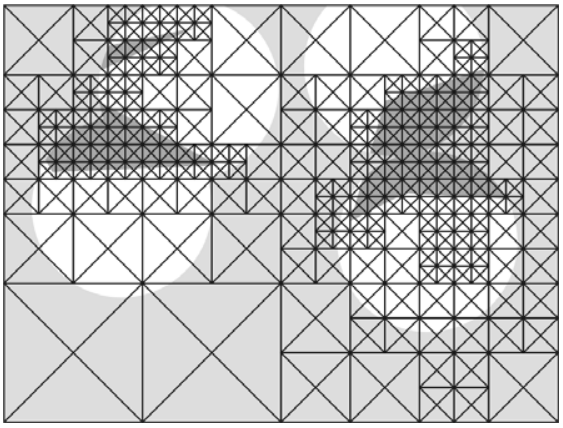


Figure 5: Example of the stage of the adaptive mesh for the same type of the explored terrain

With this approach, after a certain time, the mesh is fully adapted to the static obstacles and copes only with the dynamic influences – threats. A pseudo optimal path for the user can then be computed using Dijkstra's algorithm with the cost function similar to the function used by robots during their exploration in [AG05].

4 Our solution & implementation

This section provides an overview of our implementation of the path planning model and closely describes the implementation details. Therefore, the readers interested in the algorithm only can skip it. At this moment, our solution does not fulfil the first mentioned requirement – use of an adaptive spatial structure for the graph part – and so it is degraded to the basic type of raster-based path planning methods. Implementation fundamentals for this structure have been prepared and we will mske this generalization in the near future.

The main implementation of path planner is realized in C# language and the whole proposal is designed for providing high-level modularity – for each structure required by the proposed algorithm an interface is prepared. Each interface defines basic operations the concrete implemented structure must provide. Figure 6 shows basic elements of scene mapping part: **IRasterMap** interface must be implemented by every mapping structure used in the path planning algorithm. In compliance with the interface definition, such mapping structure must be able to provide weight of

mapped space in a certain area or position. Classes **ThreatsMap** and **ObstaclesMap** implement this interface while work in different way. **ThreatsMap** keeps only a list of threats (instances of class that implements **IThreat** interface) and **ObstaclesMap** class keeps 3D array for whole mapped space.

Basic elements of adaptive mesh part are shown in Figure 7: Generic class **Mesh** implements **IGraph** interface (presented in Figure 8) and so provides basic operations for passing the graph such as passing all vertices or passing the descendants of the specified vertex. **Engram** class is for internal use of the **Mesh** class and the delegate labelled **BlockWeight** defines the only way for adaptation of the mesh - **BlockWeight** is a specific kind of a safe pointer to the function and requires method that is able to compute weight of certain area of mapped space. Genericity of the **Mesh** class ensures that the class works with any type of vertices. Only condition for each class of vertex is implementation of the **IVertex** interface presented in Figure 8.

Figure 8 documents top-level elements of proposed path planning algorithm. Main class **Dispatcher** requires above all instance of adaptive mesh class implementing the generic interface **IGraph** and list of instances for mapping the examined space (instances of a class implementing the **IRasterMap** interface).

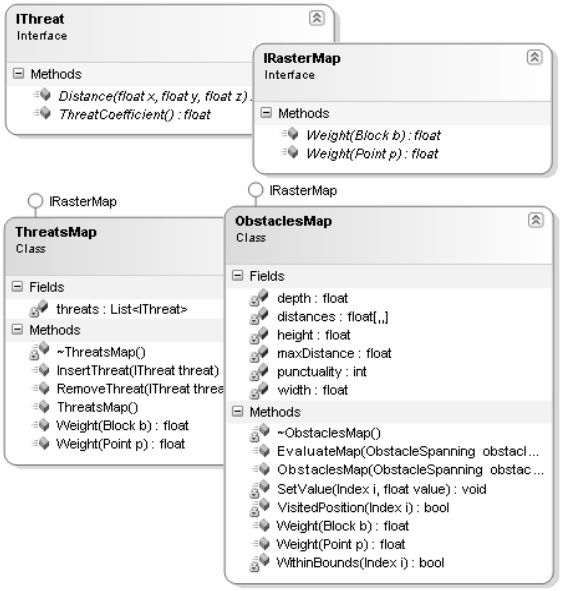


Figure 6: Class diagram of interfaces and classes in the mapping part of path planner

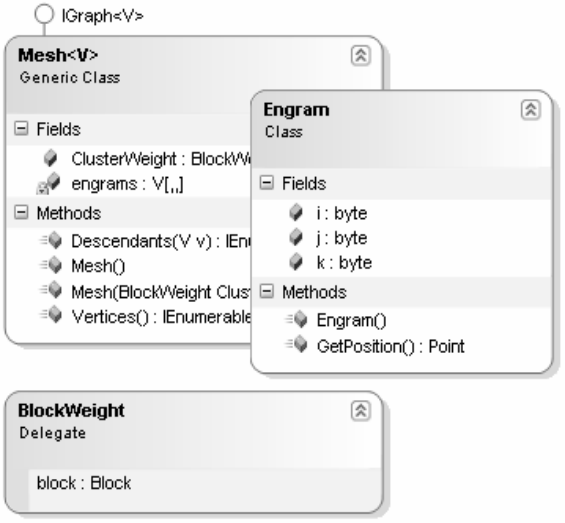


Figure 7: Class diagram of interfaces and classes in the part for adaptive mesh

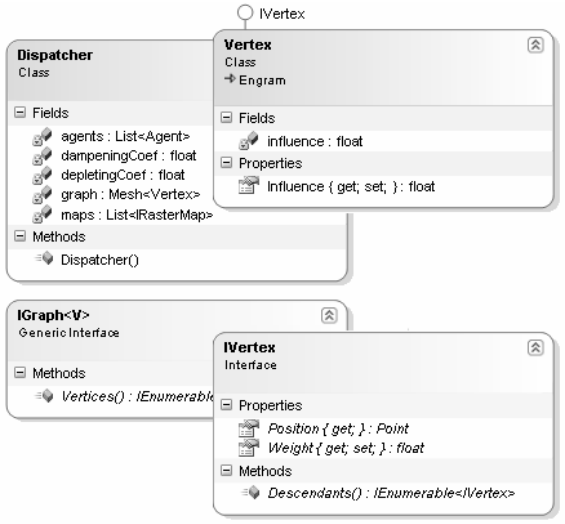


Figure 8: Class diagram of interfaces and classes in the main path planner

The proposed model provides solution for applications in known, partially known or unknown discretized environment at the price of the pseudo optimality of the final path. It comes to this, that this consequent path is optimal “only” with respect to the adaptive mesh.

5 Experiments & results

To survey our current solution, we have prepared a test application that creates a space with obstacles and uses the proposed technique for finding the path between two constant positions. Figure 9 shows the discovered pseudo optimal path in the space with the obstacles in the form of spheres with different radii. In the zoom, the same scene snapshot with weight markers is presented (lighter markers represent a low level of danger and darker markers represent a high level of danger).

The obstacles map is precomputed at the beginning of the algorithm and so the rank of this map (a count of the grid elements in each dimension) does not affect the time needed for the path planning itself. On the other hand, rank of (for the present) non-adaptive mesh has a great impact on this time. Functional dependence of the elapsed time on the mesh rank is evaluated and presented in the figure 10. The considered example finds a path in the same scene as in Figure 9 with obstacles map rank equal to 64. The computer configuration is disposed with AMD Athlon XP 1.67 GHz and 512 MB DDR RAM.

Figure 11 demonstrates using of the proposed path planning system in 2D applications and indicates the dependence of the path optimality on the rank of the mesh. Optimality of the path increases with the punctuality of the mesh (Figure 11 presents examples for the ranks 24, 36 and 48). Increase of the rank implicates the growth of the memory usage.

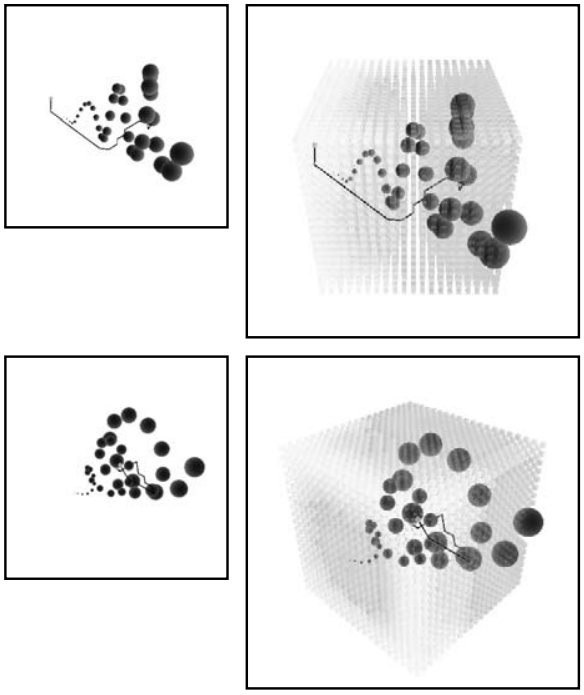


Figure 9: Mapped space with the obstacles and found path (in the zoomed snapshot with weight markers)

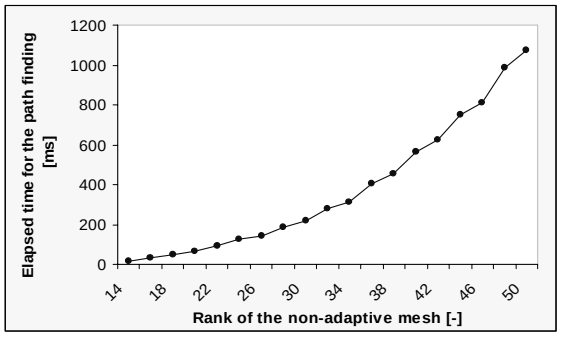


Figure 10: Elapsed time for the path finding dependent on the rank of the used non-adaptive mesh

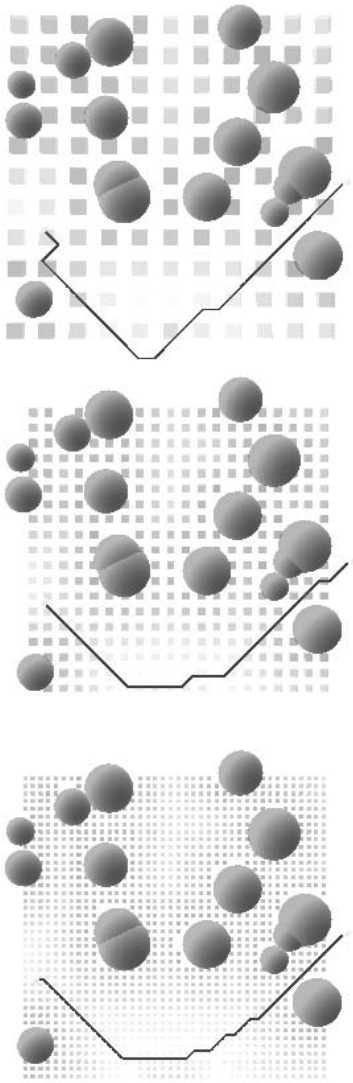


Figure 11: The discovered path for the different ranks of the adaptive mesh

6 Conclusion & future work

We have proposed hybrid and real-time path planning technique for real applications with the ability to read information about environment through the specific devices equipped with sensors. The provided method can be used in both 2D and 3D applications and works in known, partially known or unknown environment. We cooperate on research with authors of this technique in the team from the University of Calgary.

In the future, we are going to improve several parts of the algorithm solution:

- Enhancement of a method for filling the obstacles map according to the scene description: In our solution, we use unoptimized code for filling the obstacles map. If we want our technique to be usable in computer defined and abstract environment, we must assume the scene will be provided in one of the description formats. So it would be better to improve the way the map grid is created and filled from this scene representation.

- Enhancement of the adaptive structure: The way the adaptive structure copes with changes in mapped space is another great factor of algorithm performance. Topology and adaptation behaviour of this structure are main ways we want to focus on.

- Interleaving the waypoints of the found path with a specific curve is another step to make the path planning results look more human-like. While creating this curve, we must keep the collision-free property of found path and that will be another way of development we are going to take.

Acknowledgements

This work was done in cooperation with the University of Calgary in Canada. I would like to thank to Dr. M. Gavrilova and Mr. R. A. Apu for providing me the starting knowledge, 2D implementation of their solution and advices to continue in this project. I would also like to thank to Dr. I. Kolingerova from the University of West Bohemia, Pilsen, Czech Republic, for supervision and help with the paper preparation.

References

[AG05] R.A. Apu, M. Gavrilova. *Adaptive Spatial Memory Representation for Real-Time Motion Planning.* Proceedings of the 8th International Conference on Computer Graphics and Artificial Intelligence, 2005.

[Her87] John Hershberger. *Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size*. Annual Symposium on Computational Geometry, Proceedings of the third annual symposium on Computational geometry, 1987.

[Ram96] G.D. Ramkumar. *An Algorithm to Compute the Minkowski Sum Outer-face of Two Simple Polygons*. Annual Symposium on Computational Geometry, Proceedings of the twelfth annual symposium on Computational geometry, 1996.

[War90] C.W. Warren. *Multiple Path Coordination using Artificial Potential Fields*. Proceedings of the IEEE International Conference on Robotics and Automation, 1990.

[Ste94] Anthony Stentz. *Optimal and Efficient Path Planning for Partially-Known Environments.* Proceedings of the IEEE International Conference on Robotics and Automation, 1994.

[Rou98] J. O'Rourke. *Computational geometry in C (2. edition)* (http://maven.smith.edu/~orourke/books/compgeom.html). Cambridge University Press, 1998.

[DPV04] S. Dasgupta, C.H. Papadimitriou, U.V. Vazirani. *Paths in graphs* (http://inst.cs.berkeley.edu/~cs170/sp04/notes/dijkstra.pdf).

[Bat04] Ch. Batten. *Algorithms for Optimal Assembly* (http://www.mit.edu/~cbatten/work/ssbc04/optassembly-ssbc04.pdf).

Dodatek E

Exact and heuristic path planning methods for a virtual environment

Central European Conference on Computer Graphics 2007
Budměrice, Slovensko

Exact and heuristic path planning methods for a virtual environment

Petr Brož[1,2]

Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic

Abstract

Path planning belongs to the best-known and well explored problems in computer science. However, in today's real-time and dynamic applications, such as virtual reality, existing algorithms for static environment are considerably insufficient and, surprisingly, almost no attention is given to techniques for dynamic graphs. This paper introduces two methods to plan a path in an undirected graph with evaluated nodes whose value can vary in time: a simple modification of Dijkstra's algorithm to find an optimal path while processing considerably less nodes than the standard Dijkstra's algorithm and a heuristic method to find a suboptimal path while processing even smaller amount of nodes. The heuristic was developed for a virtual reality path planning application but its use is more general.

Keywords: Path planning, Dijkstra, Virtual reality, Computer graphics

1 Introduction

Path planning belongs to the basic problems not only in the computer science. For that reason, there were developed many methods for determining a path that satisfies one or more optimality criteria according to a utilizing application. However, most of these path planning methods assume that the input structure, in most cases an evaluated graph, does not change its topology or rating. Nowadays, in time of the virtual and augmented reality, many dynamic applications arise and surprisingly, almost no attention was given to algorithms for the dynamic path planning, namely for fast, suboptimal solutions.

To follow requirements and context of our currently developed path planning system for the virtual reality, we focus on a slightly different definition of a dynamic graph. The overall system uses a 3D raster and an adaptive spatial structure (Figure 1) with rated nodes to enable finding a suboptimal path with the maximal clearance among all obstacles and moving threats. Therefore, we understand a dynamic graph as a graph with varying evaluation of the nodes. As for our virtual reality application the speed is more critical than optimality, we concentrate on suboptimal approach.

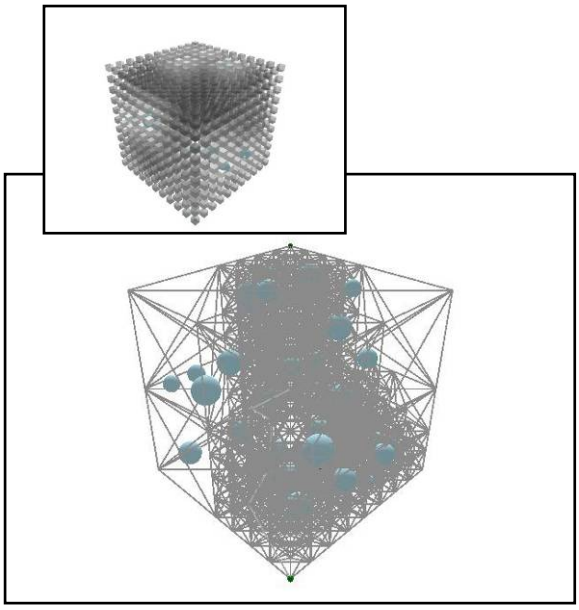


Figure 1: A visualization of data structures in our related VR project

In this paper, we present a modified Dijkstra's algorithm to plan an optimal path and a heuristic method to plan a suboptimal path in a dynamic graph without using any particular data structure. Our simple heuristic uses path information from the last graph traversal and provides suboptimal results in a notably shorter time than repeated optimal path computation. We compare this heuristic with the optimum on a dynamic graph with different ways of behavior.

[1] pebro@students.zcu.cz
[2] This project is supported by the Ministry of Education of the Czech Republic – project No. LC 06008

Section 2 describes the best known techniques for the static path planning and for the graphs with the possibility to insert or remove an edge. Section 3 presents the proposed algorithms and section 4 outlines experimentally gained characteristics and results. Section 5 then compares the presented algorithms with other techniques.

2 State of the art

Path planning represents a general task of finding an optimal path between two given spots in an abstract environment representation, in most cases in an undirected graph with weighted nodes or edges. Depending on a utilizing application, an objective of this task can be, e.g., the shortest path, the fastest path or the cheapest path.

First, we shortly summarize the standard algorithms for planning an optimal path in a static graph. The best known algorithm, the so called breadth-first search [3], finds shortest paths in any graph with unit weight of all edges with the overall running time O(|V|+|E|) where |V| means number of the vertices and |E| represents number of the edges in the graph. Dijkstra's algorithm [4] finds optimal paths in a more general graph whose edge lengths are positive integers with the running time O((|V|+|E|).log|V|). A simplified version of this technique is shown in an Algorithm 1 – ***distance*** property defines an integer distance between a current node and a starting node; ***previous*** property refers to a preceding node on the presently found path.

Finally, the shortest paths in a graph whose edges can be evaluated even with negative number can be found with the Bellman-Ford algorithm [3] with the time complexity O(|V|.|E|).

Next, we survey the algorithms for a graph with the possibility to remove or insert an edge. There are only few methods for planning an optimal path in such a graph [1, 4, 7, 9]. Most of these methods solve the so-called all pairs shortest path problem and are based on a special data structure. For a graph with unit edge costs, Ausiello and Italiano presented a data structure [1] which is able to find the shortest path between every pair in linear time O(|E|). However, maintaining the data structure requires the total time $O(|V|^3.\log|V|)$ in the worst case of insertion of at most $O(|V|^2)$ edges. Similar approaches [4, 5, 6, 11, 12] were presented to solve the all pairs shortest path problem, again by using a special data structure.

Input: the graph ***G(V, E)***,
the starting node ***S***,
the target node ***T***
Output: evaluation of nodes in ***G***
Auxiliary: ***P*** - set of complet. nodes
U,W – auxiliary nodes

- ***P*** ← empty set
- For each node in ***V***
  - Set ***distance*** to infinity
  - Set ***previous*** to null
- Set ***S.distance*** to 0
- Insert ***S*** to ***P***
- While not (***P*** contains all nodes)
  - From ***V***, find an edge between $U \in P$ and $W \notin P$ such that ***U.distance+W.weight*** is minimal
  - Set value of the ***W.distance*** to ***U.distance+W.weight***
  - Set ***W.previous*** to ***U***
  - Insert ***W*** to ***P***
  - If ***W*** is the target node, break the cycle

Algorithm 1: A simplified graph processing using the Dijkstra's algorithm

3 The proposed methods

The presented methods come out from the standard path planning algorithms for static graphs. The heuristic is adapted to find a suboptimal path without using any special data structure in dynamic graphs with varying evaluation of their nodes. The input graph can have nodes of different degree with a positive integer evaluation which may vary in time. The evaluation changes may be randomly scattered over the whole graph but in the context of virtual reality applications, we expect the changes to be concentrated according to the movement of „threatening" avatars.

Our first technique was inspired by [10] and assumes that after the evaluation change, the existing path is affected mainly by the „near changes" of the graph evaluation. In addition, it assumes that with each iteration, the starting position – actually the momentary position of a moving avatar – gets nearer to the fixed destination point. Therefore, we use the standard Dijkstra's algorithm to find the path in the reverse direction – from the destination node to the current starting node. Let us denote this approach Backward Dijkstra. In comparison with the forward direction, fewer nodes are visited. Figure 2 shows the

algorithm planning a path from the starting node **S** to the destination node **T** at the beginning and after two iterations – the nodes visited during the graph traversal are highlighted with gray color.

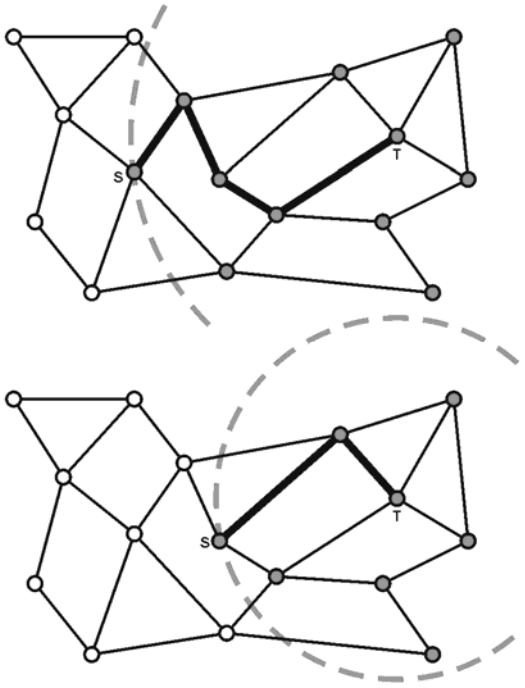


Figure 2: An example of path planning using the Backward Dijkstra's algorithm

Our second method uses the strategy that preserves as much of the original path as possible instead of finding the new optimal path after each change in the graph. It updates the last found path only in the nodes with changed evaluation. This resulting path obviously does not satisfy the optimality criteria. However, the mentioned virtual reality project and similar applications insist on the overall speed rather than on the path optimality. We call this approach Gaps filling method because of its corresponding behavior. In Algorithm 2, we show how the Gaps filling method refreshes the last found path. It can be seen that the technique stores a list of nodes on the presently found path together with their last known evaluations. In next iteration, it preserves the nodes with equal or better evaluation and spans the worse nodes with a new subpath. Figure 3 shows an original path between nodes **S** and **T** together with a new subpath through nodes **A**, **B** after the original path was disconnected in the middle node. In case that all nodes of the previous path have worse evaluation, a complete path from the starting node to the target node is found.

Input:	the graph ***G(V, E)***,
	the last found path ***W***,
Output:	the new suboptimal path ***W***
Auxiliary:	***N*** – set of new nodes

- Move the starting node from ***W*** to ***N***
- For each node ***X*** in ***N***
  - If ***X*** has equal or better evaluation or if it is the target point, find a new path from ***last(N)*** to ***X*** and insert the new nodes to ***N***

Algorithm 2: A graph processing using the gaps filling heuristic

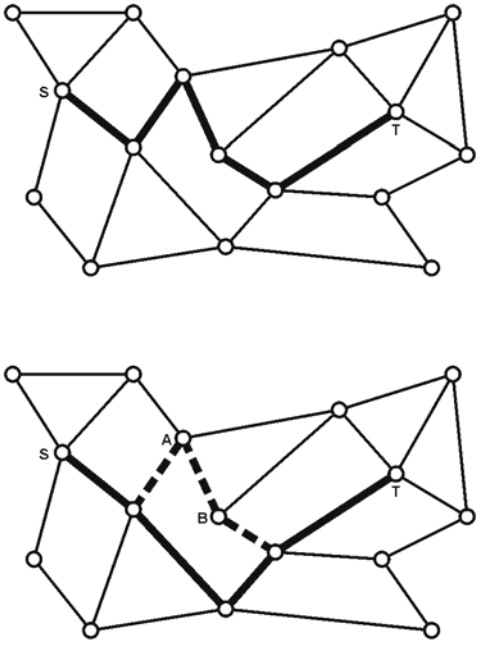


Figure 3: An example of path planning using the Gaps filling method

4 Experiments & results

For the purposes of comparison of the presented algorithms, a simple C# application was prepared to examine the behavior of the standard techniques and the proposed methods applied on an identical graph. The application was designed to enable easy extraction of the results and properties of each method. Figure 4 shows a screenshot of the application with paths generated by the tested algorithms and 3 additional windows showing the trends of the particular methods.

For now, the proposed techniques have not yet been used in the related VR project. However, we test these methods on similar datasets – we use graphs defined by an adaptive spatial structure similar to an octree with a weight value in each vertex of the smallest undivided areas.

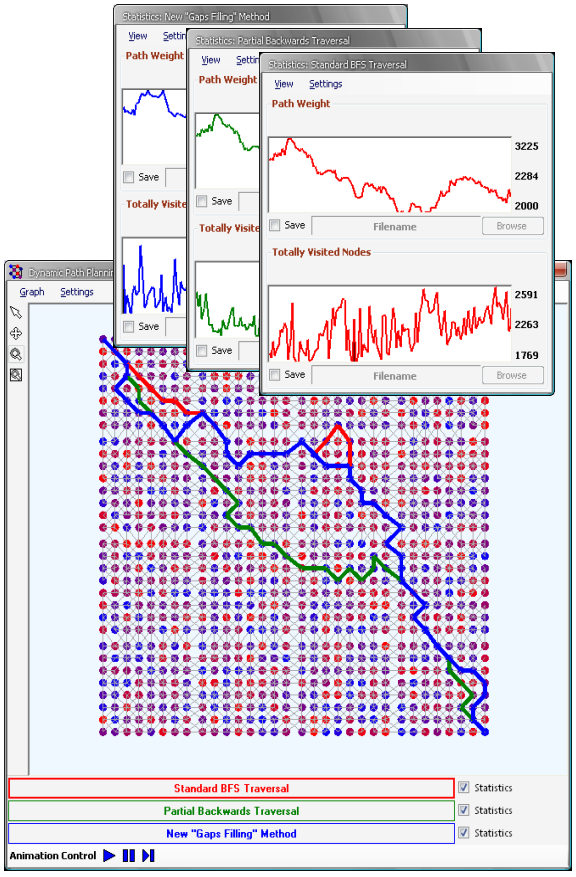


Figure 4: A screenshot of the testing application with 3 additional windows showing the trends of the particular methods

Again for the purposes of the virtual reality project, the testing data were a dense and nearly regular graph with an adjustable probability of the connection of adjacent nodes. Here, the 100% probability means that each node is connected with all neighbouring nodes, e.g., with 8 nodes within the scope of our testing planar graph. The dynamics of the graph is simulated by enabling to adjust the evaluation change probability for its nodes or to handle the evaluation changes directly through the user input. The compared path planning approaches were measured for the following cases:

- A graph with 32x32 nodes, connection probability 75% and random changes of the nodes evaluation with different probabilities (25%, 50% and 75%).
- A graph with 32x32 nodes, connection probability 75% and mouse driven changes of the nodes evaluation. Weight of each node was calculated according to its proximity to the mouse cursor.

The first case should provide a general idea about behavior of the methods, the second one simulates moving threats in a VR application.

As representative qualities of the examined techniques, the following properties were measured:

- An overall path weight – a sum of the weights of all nodes on the found path. An optimal path represents a path with the lowest overall weight.
- An amount of totally processed nodes – an amount of graph nodes visited by the particular graph traversal.

In order to concisely describe the suboptimal results of the proposed path planning approaches, an α-optimality term is used. The value $\alpha \geq 1$ stands for a ratio between suboptimal and optimal results. Following this definition, a 1,25-optimal path is 1,25 times longer or slower (according to the definition of the optimality) than the optimal path.

Random changes

For a constant probability of the evaluation change 50%, the proposed Gaps filling method finds a path that is 1,1-optimal on average (1,4-optimal in the worst case) and processes one third of nodes processed by the standard Dijkstra's algorithm. Figure 5 displays the overall weights of the paths found by the examined path planning methods. It can be seen that the results for the standard Dijkstra's algorithm and for the Backward Dijkstra are identical. Figure 6 then shows the amount of visited nodes.

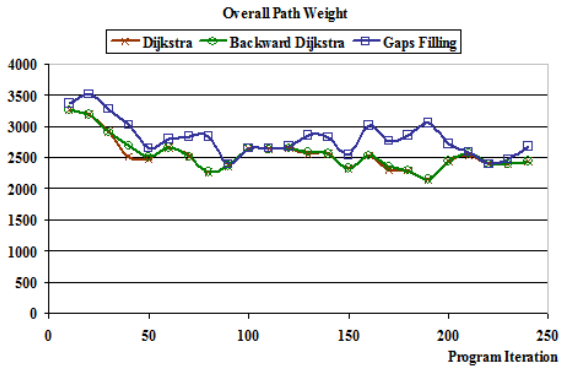


Figure 5: Overall path weight during the program run for each presented method

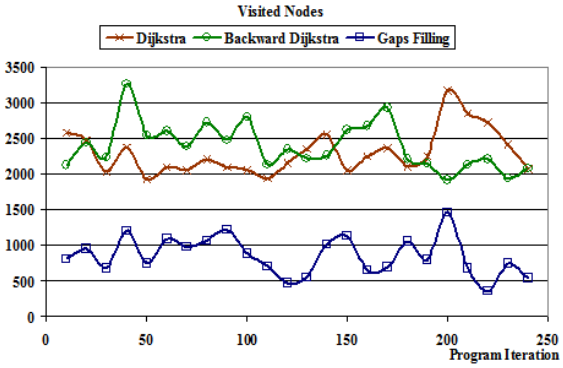


Figure 6: Total number of visited nodes during the program run for each presented method

It can be seen that Backward Dijkstra often visits more nodes than the standard Dijkstra's algorithm which was a bit surprising result for us at first but the explanation is simple – total number of visited nodes obviously depends on the initial node for the graph traversal. The probability of the evaluation change highly affects the results of the presented heuristic algorithm. For higher probabilities, more nodes can deteriorate their evaluation and the last found path has to be recomputed in more segments but the suboptimality is for that reason closer to the optimum.

Change probability [%]	Processed nodes [%]	Overall path weight [%]
25	13,70	126,26
50	37,75	115,76
75	85,67	108,67

Table 1: Different properties of the Gaps filling for the particular change probabilities

Another important factor for the quality of the Gaps filling heuristic is the amount of nodes in the graph. The higher is the amount of nodes the higher savings can be achieved by this method. On the other hand, there is a bigger chance of rising of a new optimal path that will not be registered by the presented Gaps filling method.

Driven changes

During the mouse driven changes in the graph evaluation, the gaps filling method finds a 1,3-optimal path on average and traverses one sixth of nodes processed by the Dijkstra's algorithm. Figure 7 shows an example of the testing application based on a user interaction to change the evaluation of graph nodes.

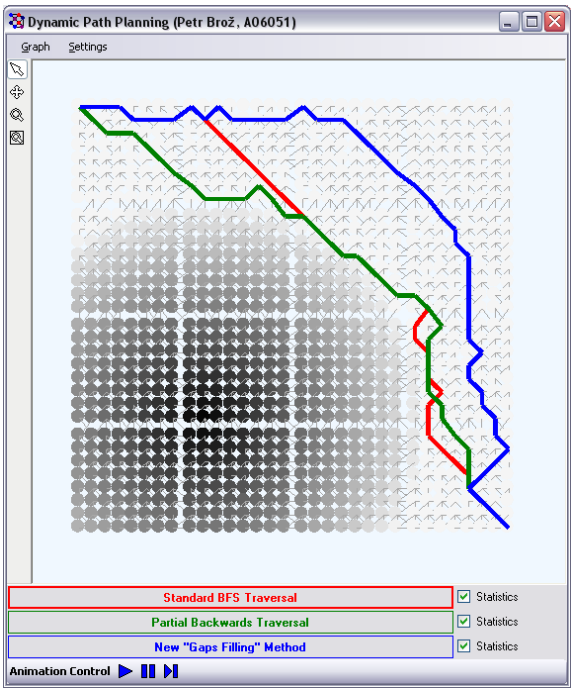


Figure 7: An example of the graph with mouse driven evaluation of its nodes

Figure 8 shows the overall path weights during the program run. Again, the results for the standard Dijkstra's algorithm and the Backward Dijkstra are identical. Figure 9 displays the totally visited nodes during the program run for each presented method. The results presented in figures 8 and 9 are measured for specific evaluation changes – all nodes in the graph are continuously evaluated according to their proximity to a potential point which is moving from the bottom-left corner to the upper-right corner of the area covered by the graph.

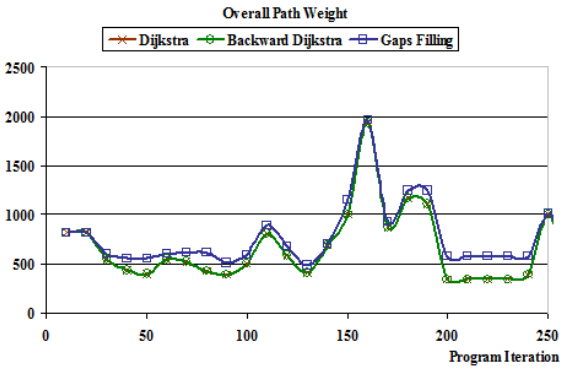


Figure 8: Overall path weight during the program run for each presented method

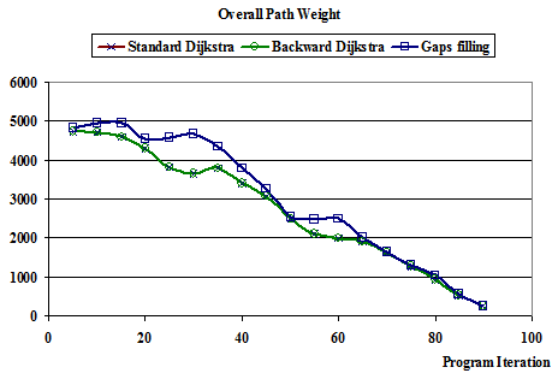


Figure 10: Overall path weight for the case of moving starting point

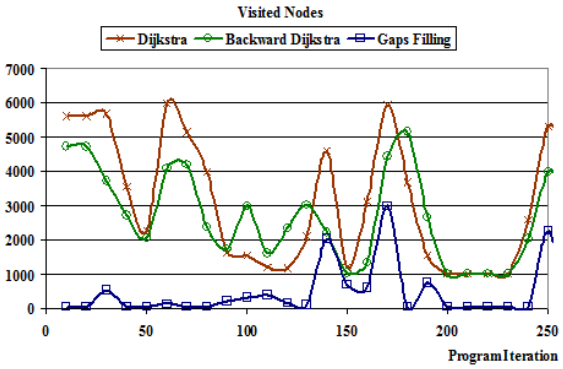


Figure 9: Totally visited nodes during the program run for each presented method

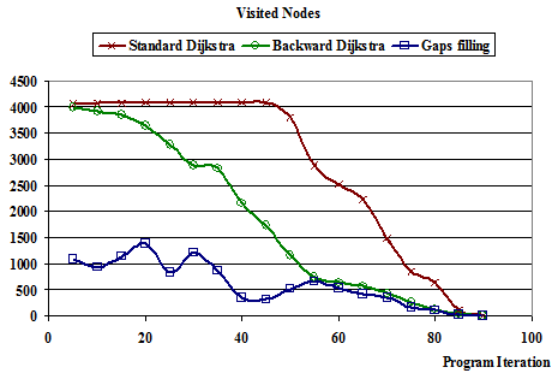


Figure 11: Visited nodes for the case of a moving starting point

To show the difference of the Backward Dijkstra's algorithm, we offer results of the measured techniques in case of a moving starting point. In each iteration, the starting node of the path moves to its successor – it continuously approaches the target node. Figure 10 again shows the overall path weight in about 90 iterations. It can be seen that the results of the standard Dijkstra's algorithm and the backward Dijkstra's algorithm are equal while the weight of a path found by the gaps filling approach stays a bit higher. Figure 11 then shows a number of processed nodes during the same measurement. There is a noticeable difference between the amount of nodes processed by the standard Dijkstra's algorithm and the backward Dijkstra's algorithm. In this example case, the backward approach finds an optimal path by visiting 55% of nodes processed by the standard technique and the gaps filling approach finds a 1,1-optimal path by processing 25% of nodes.

5 Conclusion & future work

The Backward Dijkstra's algorithm provides better results when used for planning an optimal path from a moving initial position. In these cases, it can be a suitable alternative to the standard Dijkstra's algorithm.

The proposed Gaps filling approach provides 1,1-optimal results on the average for the graphs with randomly changing evaluation and 1,3-optimal results on the average for the user driven changes in the graph. When applied on an adaptive mesh from the mentioned virtual reality project, the described approach found a 1,3-optimal path on the average by visiting 26% of the nodes processed by the standard Dijkstra's algorithm.

Due to the local updates of the last found path, the presented method does not perceive a possibly much better path, which may arise far from the last found path. A possible case may occur if there is an evaluation improvement around the last found path – the gaps filling method responds only to an aggravation of the nodes evaluation and it can miss the better path as well. However in the virtual reality, the behavior of the avatars tends to stay on

the previously found path rather than walk around in a completely new way.

In the future, we will incorporate the proposed approaches into the mentioned VR system [2] and we would also like to apply these methods in the project related to searching and visualisation of paths in chemicals (Masaryk University, Brno, Czech Republic). Figure 12 – a screenshot from this project [8] – shows a tunnel in protein molecules with a path found by the standard Dijkstra's algorithm.

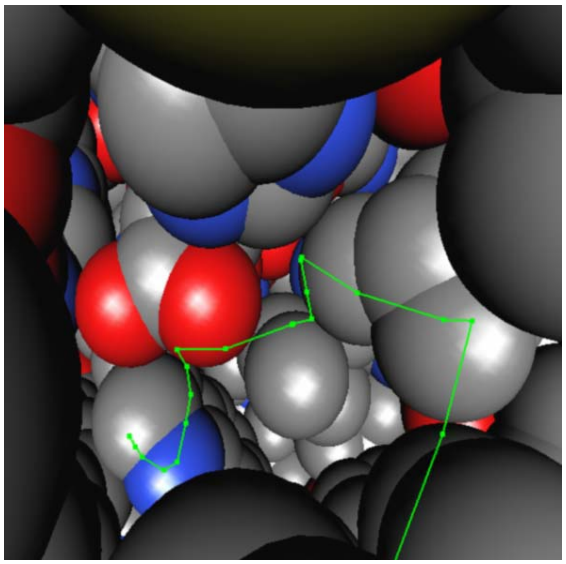


Figure 12: An example of a tunnel in protein molecules [8]

Acknowledgements

I would like to thank to Dr. I. Kolingerová from the University of West Bohemia, Pilsen, Czech Republic, for supervision and help with the paper preparation. I would also like to thank to Prof. Z. Ryjáček and Dr. F. Vávra from the same university for their consultations and valuable suggestions.

References

[1] G. Ausiello, G. F. Italiano. *Incremental algorithms for minimal length paths*. J. Algorithms, 1990.

[2] P. Brož. *Path planning in combined 3D grid and graph environment*. Proc. Central European Seminar on Computer Graphics, 2006.

[3] S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani. *Paths in graphs* (http://www.cse.ucsd.edu/users/dasgupta/mcgrawhill/chap4.pdf).

[4] S. Even, H. Gazit. *Updating distances in dynamic graphs*. Methods of Operations Research, 1985.

[5] G. F. Italiano. *Amortized efficiency of a path retrieval data structure*. Theor. Comput. Science, 1986.

[6] G. F. Italiano. *Finding paths and deleting edges in DAG*. Inform. Proc. Letter, 1988.

[7] P. N. Klein, S. Sairam. *Fully dynamic approximation schemes for shortest path problems in planar graphs*. Proc. 3rd Worksh. Algorithms and Data Structures, 1996.

[8] P. Medek, P. Beneš, J. Sochor. *Computation of tunnels in protein molecules using Delaunay triangulation*. Proc. Winter School of Computer Graphics, 2007.

[9] H. Rohnert. *A dynamization of the all pairs least cost path problem*. Proc. 2nd Annual Symp. on Theoretical Aspects of Comp. Science, 1985.

[10] Z. Ryjáček. Personal Communication, 2006.

[11] P. M. Spira, A. Pan. *On finding and updating spanning trees and shortest paths*. SIAM, J. Comput., 1975.

[12] R. E. Tarjan. *Depth-first search and linear graph algorithms*. SIAM, J. Comput., 1972.

Dodatek F

Path planning in dynamic environment using an adaptive mesh

Spring Conference on Computer Graphics 2007
Budměrice, Slovensko

Path planning in dynamic environment using an adaptive mesh

Petr Brož[1*], Ivana Kolingerová[2*], Přemysl Zítka[3]
Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic

Russel Ahmed Apu[4], Marina Gavrilova[5]
Department of Computer Science
University of Calgary
Calgary, Canada

Abstract

Solutions for the common problem of path planning in an abstract environment have been extensively studied in many scientific disciplines. However, many explored techniques assume the environment does not change and that there is a complete and detailed overview of this examined space. In addition, many path planning methods need to derive a specific graph structure from the environment representation and it can be often very difficult to obtain this structure in some real applications.

In our paper, we introduce a general model for the real-time path planning in a dynamic environment and provide a hybrid technique that combines a graph and grid representation of the examined space. The proposed path planning method uses an adaptive mesh for its graph part to provide the capability of the assimilation to the changing environment.

The presented method offers faster times for the path retrieval than the classical raster based approaches and works in a dynamic environment where the conventional graph based techniques fail. On the other hand, there are still some higher memory requirements of the proposed solution due to the necessary raster representation of the examined environment.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.6 [Computer Graphics]: Methodology and Techniques

Keywords: path planning, motion planning, computer graphics, adaptivity, level of detail, virtual reality

1 Introduction

The first methods for finding an optimal path in an abstract environment were developed even before the informatics appeared. Therefore, there are many fast and efficient techniques for solving this general task. In most cases, these conventional methods are subdivided according to the representation of the examined environment they are able to work with. Some path planning algorithms demand a graph-like definition of the processed scene, e.g., geometrical definition of all obstacles and other forbidden areas, and other algorithms assume the discrete representation of the surrounding environment is available. The graph based approaches often need to derive special graph-like structures from the provided environment description and work as lately as with this structure prepared whereas the raster based approaches usually do not need such preprocessing and search the optimal path directly in the provided grid representation of the environment.

[1]pebro@students.zcu.cz
[2]kolinger@kiv.zcu.cz
[3]premysl@students.zcu.cz
[4]apu@cpsc.ucalgary.ca
[5]marina@cpsc.ucalgary.ca
[*]Supported by the Ministry of Education of the Czech Republic (project No. LC 06008)

The path planning algorithms based on both types of the environment representation have crucial and radical disadvantages when they are to be used in the real applications. The graph representation of the real environment is rarely available and its construction is – if possible at all – very complicated and difficult. On the other hand, the discrete representation of the examined space is much easier accessible and primarily measurable but the algorithm itself is in most cases very time-consuming due to the amount of the raster elements to inspect. In addition, almost all methods for the path finding and planning need well-known or static environment which is not often available, either.

A good improvement for the mentioned applications of these path planning systems can be achieved with the combination of the discrete and graph environment approaches. Such a technique could use an adaptive spatial structure as a graph with its vertices and edges evaluated according to the specific values from a collaborative grid structure. It could discover an optimal path among all available transitions in the graph (but not among the grid values) and continuously adapt this spatial structure to the dynamic influences.

In this paper, we propose a possibility for the path planning over the combined environment representation which reduces the main disadvantages of the mentioned conventional approaches. We use an adaptive mesh to define all available waypoints and transitions for searched paths and a simple 3D matrix to store all raster based values. As the result, we provide a path planning technique that is faster than the raster based approaches and suitable for the applications in the dynamic environment. Preliminary version of our method, without the adaptivity, has been published in [Brož 2006].

Section 2 shows some of the best known techniques for a dynamic path planning, section 3 then describes our proposed method and section 4 presents results of this technique compared to conventional methods. Finally, section 5 concludes characteristics of the presented algorithm and shows a possible future work.

2 State of the art

Path planning in general denotes a basic problem of finding an optimal path between two specified spots in an abstract environment representation. In this context, the optimal path means the path satisfying one or more given objectives (the shortest, the cheapest or the fastest path, the path with the maximal clearance among all surrounding obstacles, etc.). The mentioned abstract environment can be represented in a variety of ways but the path planning algorithms are focusing mainly on evaluated graphs and 2D or 3D grids. There are many ways these environments can be differentiated (dynamic/static, known/unknown environments, etc.) which implies a similar distinction of the path planning techniques according to the types of the environment they are able to work with.

2.1 Graph based methods

First, let us introduce the best known approaches based on the graph representation of the examined space. The Visibility graph [Hershberger 1987] technique extends the basic provided geometrical definition of the environment with the edges connecting the points that can "see" each other whereas the source and destination position is treated as an obstacle, too. The new edges together with the edges defining the sides of each obstacle then represent the possible transitions and through them, the optimal path can be found. An example of such preprocessing in a 2D application can be seen in Figure 1: the edges of all obstacles (filled with the bricks pattern), the starting and ending position (the points labelled with the letters S and E) are connected according to their mutual visibility and over the possible transitions (the thin lines together with the obstacles sides), the optimal path (the dashed lines) is found.

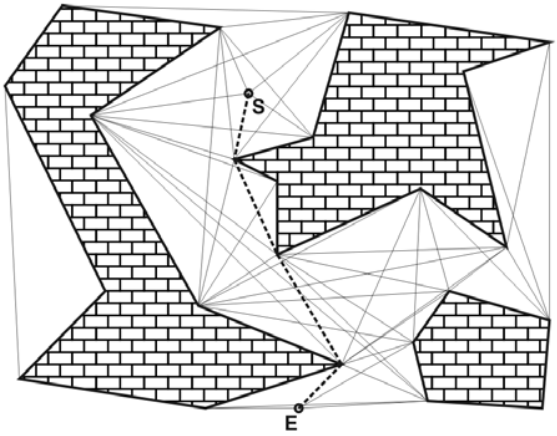


Figure 1. An example of the scene processing with the Visibility graph technique.

The Minkowski sum [Ramkumar 1996] technique is a similar approach that considers the shape of the passing object and „inflates" the borders of each obstacle so that the collision-free path can be solved. An example of such preprocessing is presented in Figure 2: the same obstacles as in Figure 1 are inflated with the radius of the passing object (the gray areas) and the collision-free path (the dashed lines) between the starting and ending position (the spheres labelled with the letters S and E) is found. With the specific structure prepared, both approaches can use the Dijkstra's algorithm [Dasgupta et al. 2004] or similar one to find the appropriate path.

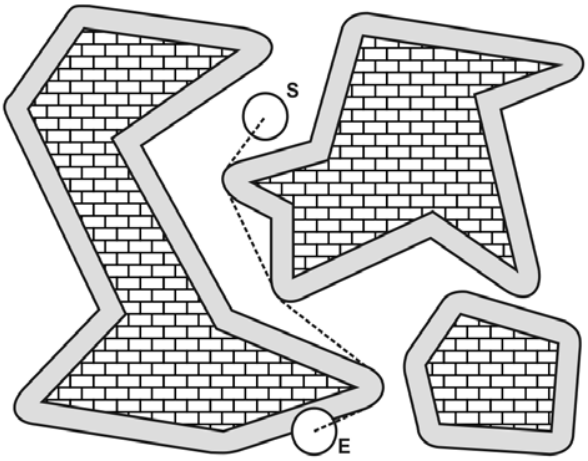


Figure 2. An example of the path planning with the Minkowski sum method.

2.2 Raster based methods

After the short introduction into the graph-based methods for the path planning, we are providing insight into the techniques based on the grid representation. Such a grid can be precomputed or modified at the beginning of the algorithm. In reference to the modification of the explored grid, the potential field method [Warren 1990] can be used for filling the grid with the discrete values of a specific potential field generated by all obstacles. Passing through the grid elements with the lowest potential values then provides the path with maximal clearance among all obstacles. The known techniques for searching itself are for example the A* algorithm for the well-known environment [Batten 2004] and the D* algorithm for the unknown, partially known or changing environment [Stentz 1994]. Figure 3 shows the manner of such path finding in the grid: the obstacles from Figure 1 and 2 are now splitted into the raster and in this raster, the optimal path between the starting and ending position (the cells labelled with the letters S and E) over the grid cells is outlined.

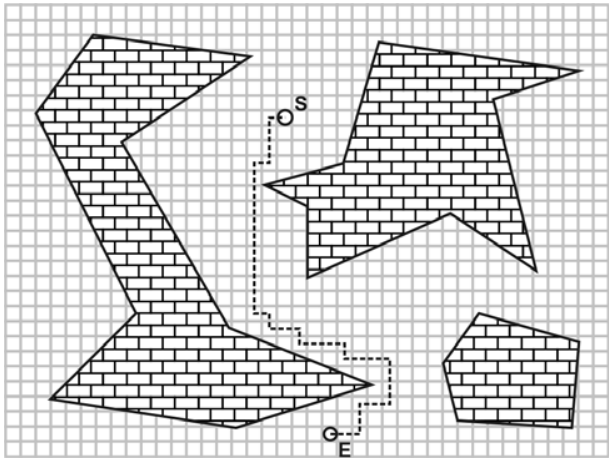


Figure 3. An example of the raster based path planning.

3 The proposed model

To provide a suitable path planning model for the mentioned applications where the conventional approaches fail, we are focusing on a general path planning technique that is able to work in the known, partially known or unknown discrete environment.

We advance in the development of a general model for the real-time and adaptive path planning that was pioneered by R. A. Apu [2005]. The proposed model can be used for both 2D and 3D applications and works in a complex and dynamic environment which is assumed to be provided in the raster representation and can be well known, partially known or even unknown. The described path planning system is based on these main headstones:

- A graph-like spatial structure (hereafter referred to as a **mesh**) that adapts itself to the examined environment and defines all reachable positions and transitions with its vertices and edges.
- A grid structure for the discrete representation of certain environment hazards (hereafter referred to as a **map**), e.g., the proximity to an obstacle or the dynamic threats.

The main approach uses two separate maps of the same size for the environment description. The first one, called obstacles map, represents the danger weights as the proximities to the nearest obstacle in the mapped space and the second one, called threats map, represents the potential field generated by all threats in the examined terrain. In the following, the proposed path planning model keeps a mesh coming out of an oct-tree that is „widespread“ over the examined space. This mesh then defines all available waypoints and transitions for the path retrieval and continuously copes with the changes in both mapping structures. Such an adaptation is achieved by refinement of the mesh in the places with higher error values calculated from the obstacles map and threats map and by merging of the mesh in the unimportant areas.

The algorithm is based on the real-time development of the mesh during the particular iterations. According to the presently recorded values in the maps, the mesh is refined in the areas with the higher importance and merged in the areas with the lower importance. Decision for the refinement or merging of the particular cluster is derived from the specific influences of the obstacles and threats in each volume unit of this cluster. A constant $\delta \geq 0$ defines a user defined refinement threshold so that the cluster is merged with its neighbours for the criterion result less than $1-\delta$, refined for the criterion result greater than $1+\delta$ or left in its original state. Exact formulation of this criterion is beyond the scope of the paper.

In the proposed solution, the adaptive mesh is used only to define the available waypoints and transitions for the movement, not for the visualization. Therefore, T-vertices in the mesh do not bring any problems typical for them in the visualisation of the meshes. Foldovers in the mesh are not possible in our case as the vertices are not moved, just refined.

In our demonstrating application, we assume the obstacles in the environment are already completely explored and the obstacles map is filled with the Image Distance Transform technique based on the Voronoi diagrams [O'Rourke 1998]. Concretely, the elements of the obstacles map are evaluated according to their proximity to the nearest obstacle with the real value from 0 (minimal proximity) to 1 (maximal proximity to the nearest obstacle or the obstacle itself). The elements of the threats map are then evaluated in a similar manner during the mesh adaptation. After a certain time, the mesh is fully adapted to the static obstacles and copes only with the dynamic influences – with the threats. A pseudo optimal path for the user can then be computed using Dijkstra's algorithm with the specific cost function. There are many ways to specify the cost functions and in addition to, they can differ according to the type of the application. For that reason, we are not focusing on the concrete cost function definition.

4 Experiments & results

We have implemented a simple application (in the C# language with the Direct3D libraries) to provide the results and comparisons of our algorithm. Unfortunately, the proposed technique needs to map the dynamic obstacles to the raster in each step and we are looking for a suitable solution to solve this complex process. Therefore, we provisionally use only a simple scene composed from a certain number of obstacles formed by the solid spheres with different radii and we also add some abstract threats represented by the small red cubes. During the program run, the threats are directed to the random locations of the examined space and the adaptive mesh is refined in each iteration. The obstacles map is precomputed and filled with the corresponding values before the main loop of the program. Therefore, the complexity of this structure and its creation does not affect the qualities of the real-time method itself. To demonstrate the adaptivity of the algorithm, the optimal path is recomputed after each refinement of the mesh.

An example of such a testing scene with 16 obstacles and 2 threats is shown in the following images: Figure 4 shows the basic scene only with the found path between two opposite corners of the examined space, Figure 5 then shows the same scene together with the actual state of the adaptive mesh (with the maximum level of the mesh division equal to 4) and Figure 6 shows again the same scene but this time with the weight values from the obstacles map (the darker cubes define safer locations and the lighter cubes define more dangerous locations).

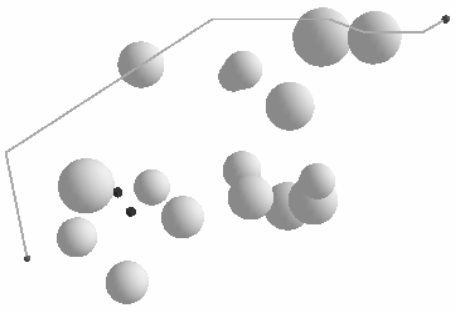


Figure 4. An example of the random scene in the demonstrating application.

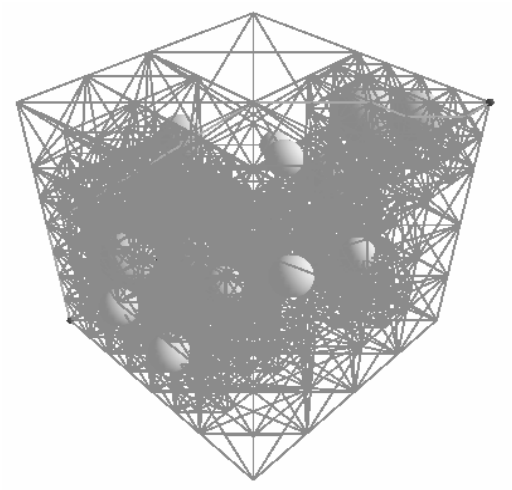


Figure 5. A snapshot of the scene from Figure 4 with the adaptive mesh.

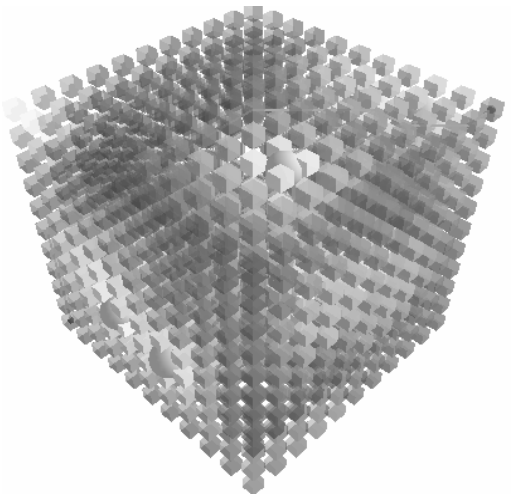


Figure 6. A snapshot of the scene from Figure 4 with the values from the obstacles map presented.

4.1 Survey of the tests

For our survey, we have selected and measured the following variables as the most characteristic and important parameters of the proposed method:

- **Clusters amount** defines the count of the atomic elements in the adaptive structure. In Figure 5, these are the smallest cubes with all corners connected to each other.
- **Adaptation time** determines the time needed for a single iteration of the adaptive structure progression.
- **Allocated memory** defines the memory requirements of our C# implementation (debug version).
- **Path finding time** denotes the time needed for finding the optimal path between two constant spots in the opposite corners of the explored space.

The mentioned parameters have been measured with the following testing datasets to present the qualities and possible weak points of our implementation:

- **Dataset #1** describes the algorithm in the environment consisting of 8 solid spheres. These obstacles fill about 3% in the volume of the examined space. The rank of the grid used in this preset is equal to 32 - the obstacles map contains 32 768 elements and the maximum level of the mesh division is set to 4. That means that the smallest cluster in the mesh has one sixteenth of the original width.
- **Dataset #2** defines the same adjustment of the algorithm as the dataset #1 except that the maximum division level is equal to 5. 8 solid spheres with different radii are randomly dislocated in the examined environment and fill again about 3% of the space.
- **Dataset #3** specifies the configuration with 16 obstacles filling about 6% of the explored environment, with the rank of the obstacles map equal to 64 and the maximum level of the mesh division set to 4.
- **Dataset #4** again defines the same adjustment of the previous dataset except that the maximum division level is set to 5. 16 obstacles fill about 6% of the examined environment, 64x64x64 elements in the grid used for the obstacles map.

First of all, we present the functional dependence of the clusters amount on the particular iterations of the program in Figure 7. At the very beginning of the main loop, we can see a rapid growth of the clusters amount as the adaptive mesh refines itself around the obstacles. The remaining behaviour of this dependence highly varies due to the movement of the threats in the scene. When the threat shifts off from the near obstacles, it raises the weight value of the previously unimportant locations and so evokes a new refinement of the mesh around these locations. Such situations then evoke the peeks of the clusters amount that are visible in all mentioned dependencies in Figure 7.

The environment of the testing application changes itself in an absolutely random way as the threats are directed to the random locations in it. It is still possible to discover some events in the application from the presented dependencies. We can take a look for example at the graph for the dataset 2 in Figure 7: the growth in the graph (during the iterations 150-200, 350-500, 600-700, 780-800 and 920-980) are due to the threats that move away from the nearest obstacles and so invoke the mesh refinement in the previously unimportant locations. In Figure 7, we can also see the datasets 2 and 4 are much more varying than the datasets 1 and 3 but there is a clear explanation for it. With the higher level of maximum mesh division, there are more clusters reacting on the threats movement. Figure 8 shows the time requirements of each single mesh adaptation. We present the moving average (8 periods) values instead of the original values. Apparently, the time requirements of the adaptation depend mainly on the maximum level of the mesh division. Also in these graphs, the peeks can be explained with the behaviour of the threats. When the threats shift away from the near obstacles, they fill more elements of the grid with higher values and so there are more areas for the refinement.

Finally, Figure 9 provides common insight into the memory requirements of our implementation. The measured values are approximate due to the garbage collector used in the C# programs. High and constant amount of the memory is required for the grid structure (32x32x32 float matrix for the rank 32) and so the step changes in the dependencies are caused only by the mesh adaptation itself.

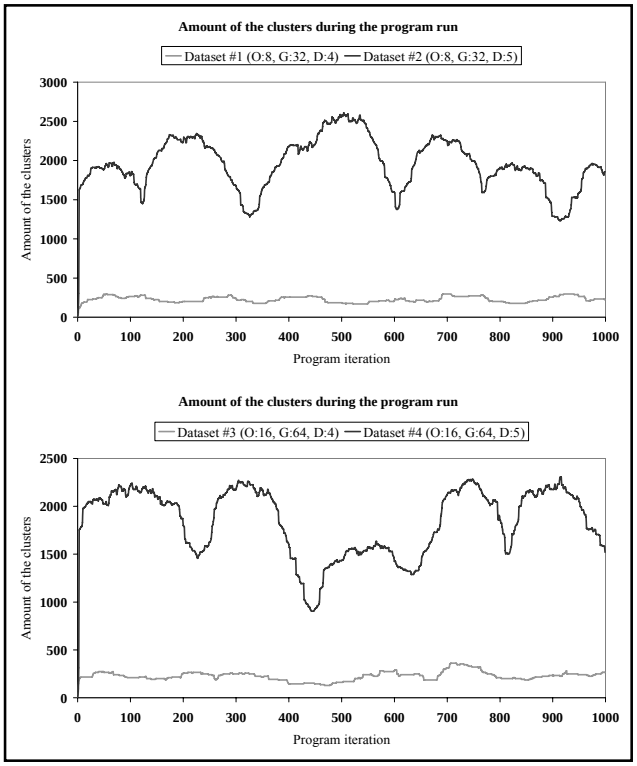


Figure 7. A clusters count dependent on the program iterations.

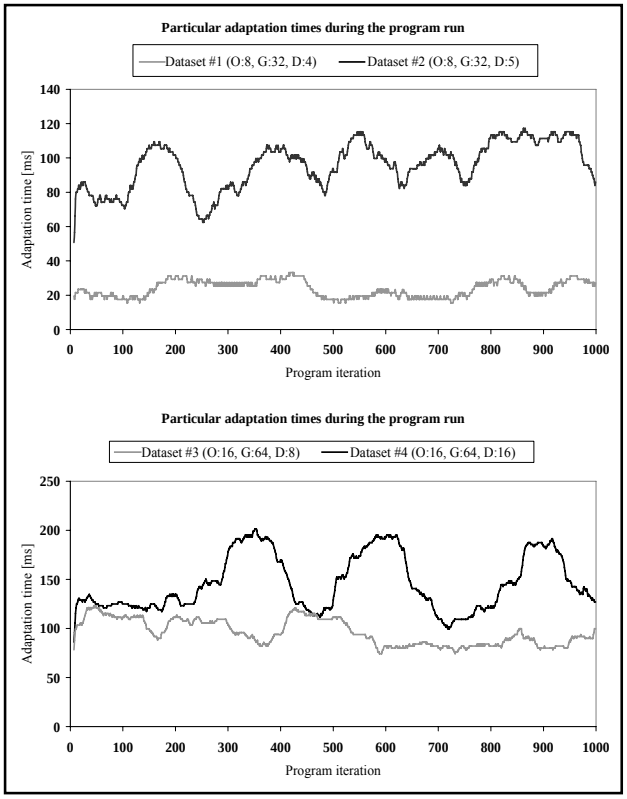


Figure 8. Time requirements of the mesh adaptation dependent on the program iterations.

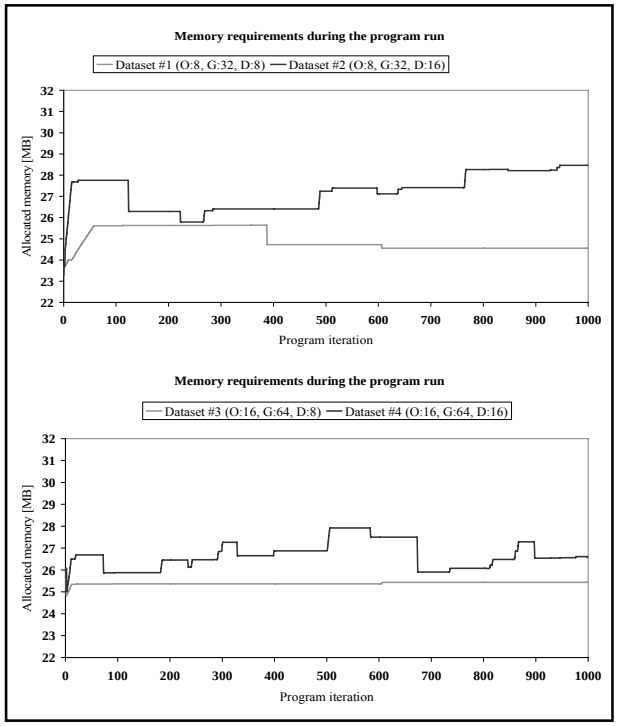


Figure 9. Memory requirements dependent on the particular iterations.

4.2 Obstacles count independence

The demonstrating application was also used to survey the algorithm independence of the obstacles amount. We have defined other adjustments of the algorithm similar to the presets from the subsection 4.2 and we have measured the properties of our solution for 256, 512, 1024 and 2048 obstacles randomly dislocated throughout the space. The concrete values are again defined in the parentheses behind each preset in the legends – 'O' stands for the obstacles count, 'G' stands for the rank of the grid and 'D' stands for the maximum level of the mesh division.

Figure 10 shows the functional dependence of the clusters amount and the particular iterations of the program for these new configurations. The presented graphs indicate that the clusters count is not directly dependent upon the obstacles amount. The differences in these graphs are caused by the topology of all obstacles in the scene that is randomly generated for each dataset. The same situation of this separateness occurs in Figure 11 with the functional dependence of the mesh adaptation times and in Figure 12 with the dependence of the time for finding the path. Figures 11 and 12 are again presented in the form of the moving averages due to the oscillating measured values. The considered obstacles are preprocessed in the grid and that is the only part of our solution that is affected by the count of the obstacles.

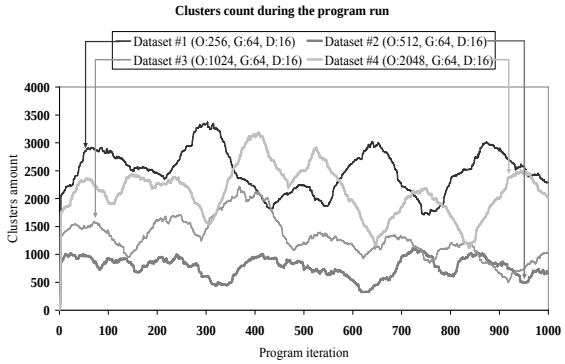


Figure 10. Clusters counts in the particular iterations of the program.

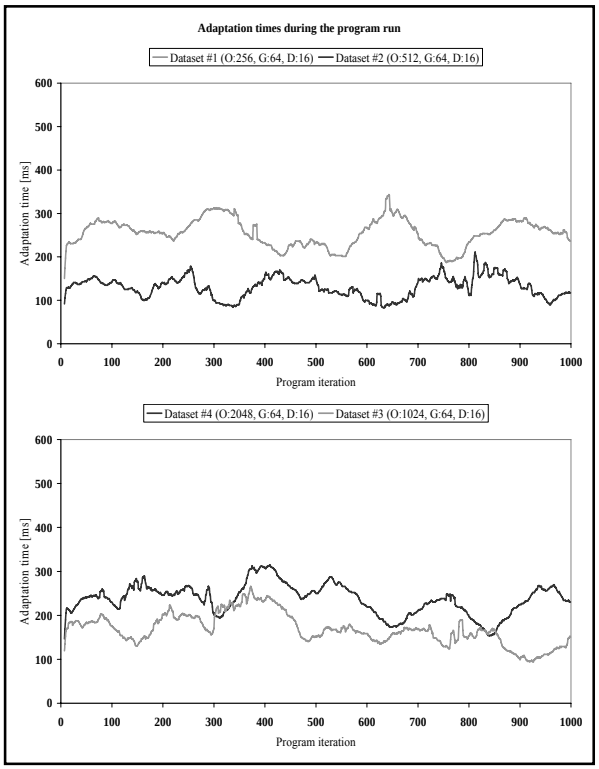


Figure 11. Mesh adaptation times dependent on the particular iterations of the program.

4.3 Non-adaptive mesh comparison

We also have an implementation of the proposed path planning technique with the regular mesh - it provided us the opportunity to compare this old solution to our current implementation.

Figure 13 shows two selected configurations from the subsection 4.2 together with the results from the two equivalent adjustments of the old algorithm with the regular mesh - in this case, 'G:64' means the regular mesh consisting of 64x64x64 clusters. The times for the regular mesh stay around the value 400ms whereas the times for finding the path in the adaptive mesh strongly vary but they never achieve the time needed by the old algorithm. Average time for the preset #2 is about 45ms and for the preset #4 about 200ms.

The measurements show the proposed path planning method is a suitable alternative for the path planning in dynamic environments: it is faster than the raster based approach and it is usable in the applications where the graph based techniques fail. On the other hand, there are still high memory requirements due to the 3D matrix for the grid used in our solution.

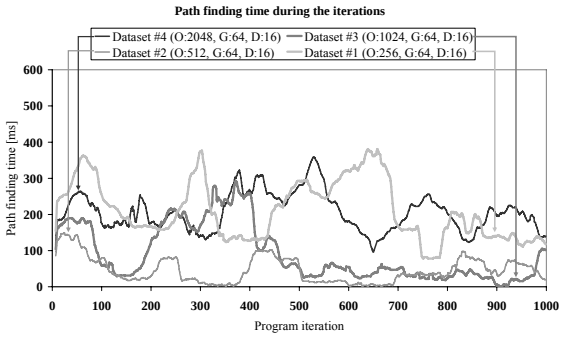


Figure 12. Path finding times dependent on the particular iterations of the program.

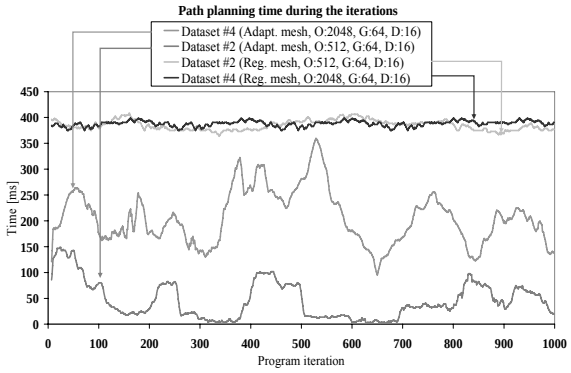


Figure 13. Path planning times dependent on the particular iterations of the program.

5 Conclusion & future work

We have outlined the possible model for the path planning system that eliminates the described disadvantages of the conventional approaches applied in the virtual reality. The provided method can be used in 2D and 3D applications, works in the dynamic environment and needs only about 10-50% of the original time for finding the optimal path with the Dijkstra's algorithm. However, the proposed solution is still under development and there are many possible ways to improve this model for the real-time path planning.

In the future, we would like to incorporate the described approach in a project related to searching and visualization of paths in chemicals [Medek et al. 2007] (Masaryk University, Brno, Czech Republic).

References

APU, R. A., GAVRILOVA, M. 2005. Adaptive Spatial Memory Representation for Real-Time Motion Planning. *Proceedings of the 8th International Conference on Computer Graphics and Artificial Indelligence.*

BATTEN, CH. 2004. Algorithms for Optimal Assembly. *(http://www.mit.edu/~cbatten/work/ssbc04/optassembly-ssbc04.pdf).*

BROŽ, P. 2006. Path Planning in Combined 3D Grid and Graph Environment. *Proceedings of the 10th Central European Seminar on Computer Graphics.*

DASGUPTA, S., PAPADIMITRIOU, C. H., VAZIRANI, U. V. 2004. Paths in Graphs. *(*http://inst.cs.berkeley.edu/~cs170/ *sp04 /notes/dijkstra.pdf).*

HERSHBERGER, J. 1987. Finding the Visibility Graph of a Simple Polygon in Time Proportional to its Size. *Annual Symposium on Computational Geometry, Proceedings of the 3rd annual symposium on Computational Geometry.*

MEDEK, P., BENEŠ, P., SOCHOR, J. 2007. Computation of tunnels in protein molecules using Delaunay triangulation. *Proc. Winter School of Computer Graphics.*

RAMKUMAR, G. D. 1996. An Algorithm to Compute the Minkowski Sum Outer-Face of Two Simple Polygons. *Annual Symposium on Computational Geometry, Proceedings of the 12th annual Symposium on Computational Geometry.*

O'ROURKE, J. 1998. Computational Geometry in C (2nd edition). *(http://mavem.smith.edu/~orourke/books/ compgeom.html), Cambridge University Press.*

STENTZ, A. 1994. Optimal and Efficient Path Planning for Partially-Known Environments. *Proceedings of the IEEE International Conference on Robotics and Automation.*

WARREN, C. W. 1990. Multiple Path Coordination using Artificial Potential Fields. *Proceedings of the IEEE International Conference on Robotics and Automation.*