VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# NEJKRATŠÍ CESTA MEZI DVĚMA BODY PO POVRCHU OBJEKTU

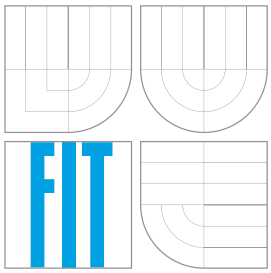
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ MEZERA

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# NEJKRATŠÍ CESTA MEZI DVĚMA BODY PO POVRCHU OBJEKTU

SINGLE PAIR SHORTEST PATH ON SURFACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ MEZERA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ONDŘEJ ŠILER
SUPERVISOR

BRNO 2008

Zadání bakalářské práce/6642/2007/xmezer02

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií Akademický rok 2007/2008

# Zadání bakalářské práce

Řešitel: **Mezera Lukáš**
Obor: Informační technologie
Téma: **Nejkratší cesta mezi dvěma body po povrchu objektu**
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s nástroji a knihovnami pro práci s 3D modely.
2. Seznamte se s problémem nejkratší cesty a jeho optimálním řešením (Dijkstrův algoritmus).
3. Prostudujte aproximační řešení.
4. Implementujte Dijkstrův algoritmus.
5. Implementujte některý aproximační algoritmus.
6. Porovnejte metody.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- První tři body zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Šiler Ondřej, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

L.S.

doc. Dr. Ing. Pavel Zemčík
*vedoucí ústavu*

Licenční smlouva je uvedena v archivním výtisku uloženém v knihovně FIT VUT v Brně.

# Abstrakt

Hledání nejkratší cesty patří mezi základní problémy řešené v počítačové geometrii. Optimálním řešením je výpočet pomocí Dijkstrova algoritmu. Existuje ale i celá řada aproximačních algoritmů, které je také možné pro výpočet nejkratší cesty použít.



# Klíčová slova

Nejkratší cesta, trojúhelníkové sítě, Dijkstrův algoritmus, aproximační algoritmy, algoritmus Kanai Suzuki, MDSTk, VectorEntity, OpenSceneGraph



# Abstract

Finding the shortest path is a fundamental problem in computational geometry. Optimal solution is computation by force of Dijkstra algorithm. There are many approximation algorithms which we can use for calculate the shortest path.



# Keywords

Shortest path, triangular meshes, Dijkstra algorithm, approximation algorithms, Kanai Suzuki algorithm, MDSTk, VectorEntity, OpenSceneGraph



# Citace

Lukáš Mezera: Nejkratší cesta mezi dvěma body po povrchu objektu, bakalářská práce, Brno, FIT VUT v Brně, 2008

# Nejkratší cesta mezi dvěma body po povrchu objektu

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Šilera

.....
Lukáš Mezera
13. května 2008

## Poděkování

Děkuji vedoucímu mé bakalářské práce Ing. Ondřeji Šilerovi za odborné vedení a cenné rady a připomínky při zpracování bakalářské práce. Dále bych chtěl poděkovat Mgr. Anně Porazilové a Doc. RNDr. Františku Ježkovi, CSc. za poskytnutí odborné literatury.

# Obsah

# Kapitola 1

# Úvod

Hledání nejkratší cesty je jedním ze všeobecných problémů dnešní doby. Vezměme si jako příklad vědní obor logistika, jehož nedílnou součástí je optimalizace najetých kilometrů při dodávkách různých druhů zboží. Právě vyhledávání nejkratší cesty mezi jednotlivými body cesty je asi každému z nás nejbližší problém tohoto vědního oboru. Kdo by také chtěl jet z Brna do Prahy například přes Ostravu?

V moderní době ale sezení u map a vyhledávání nejkratší cesty už není tak obvyklé jako dříve. Souvisí to zejména s moderní technologií, která za nás tento problém je schopna vyřešit. Asi každý z nás při plánování cesty na dovolenou otevřel stránku na internetu, která nejen obsahovala mapu, ale i vyhledání nejkratší nebo nejrychlejší cesty mezi dvěma body. Ten, kdo tento problém neřešil u internetové mapy, je nejspíše vlastníkem GPS navigace, která vyhledá nejkratší cestu také a navíc Vás podle ní na určené místo navede.

Tímto se dostáváme do oné úzké spojitosti mezi hledáním nejkratší cesty v reálném životě a výpočetní technologií, která je tento problém schopna vyřešit. Přejděme tedy od reálného života do života techniky.

Hledání nejkratší cesty po povrchu objektu patří mezi základní problémy řešené v počítačové geometrii a dalších aplikacích z oblasti robotiky, navigace, bioinformatiky, geografických informačních systémů (GIS) aj. Zatímco efektivní rešení úloh hledání cest mezi dvěma body ve 2D je již delší dobu k dispozici, metody známé ve 3D mají stále značnou časovou a pameťovou složitost.

Problém hledání nejkratší cesty lze rozdělit do dvou kategorií. První z nich je nalezení nejkratší cesty mezi zdrojovým bodem $s$ a všemi ostatními body povrchu objektu. Jako druhou kategorii lze označit nalezení nejkratší cesty mezi zdrojovým bodem $s$ a bodem cílovým $t$. Tento text bude dále zaměřen na hledání nejkratší cesty mezi zdrojovým $s$ a cílovým $t$ bodem.

V tomto textu se na problém hledání nejkratší cesty podíváme dvěma rozdílnými úhly pohledu. Pokud bych opět tento problém přirovnal k problému dopravy, dalo by se s nadsázkou říci, že pokud nemáme dálniční známku, musíme nejkratší cestu hledat po okresních komunikacích (Dijkstrův algoritmus). Pokud dálniční známku vlastníme, jsme oprávněni jet po dálnici (Kanai Suzuki algoritmus). Je zřejmé, že každý úhel pohledu má své klady a zápory. Pokud bude zvolena cesta po okresních silnicích, není nutné platit poplatek za dálniční známku, avšak cesta bude pravděpodobně delší. Jakmile ale bude zakoupena dálniční známka, je možné po dálnici jezdit jak často se nám zlíbí.

Podobnost těchto dvou dopravních situací s řešením algoritmů pro výpočet nejkratší cesty má také svou spojitost. Dijkstrův algoritmus vypočítá nejkratší cestu po známých hranách trojúhelníkové sítě, ale nedosáhneme jím ve většině výpočtů takové nejkratší

cesty, jako algoritmem aproximačním (Kanai Suzuki). Aproximační algoritmus Kanai Suzuki vypočítá nejkratší cestu s takovou přesností, která je od něj vyžadována. Zápornou vlastností tohoto algoritmu je ale jeho časová náročnost.

Nechme však už analogií dopravy a výpočtu nejkratší cesty a podívejme se blíže na to, čím se budeme v následujícím textu dále zabývat a to problémem hledání nejkratší cesty po povrchu modelu. K výpočtům nejkratší cesty budou sloužit dva algoritmy. Dijkstrův algoritmus 2.2 a algoritmus Kanai Suzuki 2.4.

V kapitole *Rozbor problematiky* bude z počátku zmíněn pojem *Trojúhelníkové sítě* 2.1. Další část této kapitoly pojednává o Dijkstrově algoritmu a jeho vlastnostech 2.2. Následuje krátký přehled aproximačních algoritmů 2.3 a na závěr je popsán algoritmus Kanai Suzuki 2.4 a jeho vlastnosti. U obou algoritmů je zmíněn i pseudokód pro jednodušší představu jejich implementace.

Druhá kapitola *Implementace* obsahuje návrh implementace praktické části práce. Je zde krátce popsán toolkit MDSTk 3.1 a podrobněji rozebrána knihovna VectorEntity, která je v praktické části využívána. Dále je zde krátce popsán OpenSceneGraph 3.2, díky kterému má praktická část grafické okno pro zobrazení modelu a vypočtené nejkratší cesty mezi zadanými dvěma body tohoto modelu. Poslední částí kapitoly je sekce *Vlastní implementace* 3.3, ve které jsou blíže popsány jednotlivé moduly a funkce implementované v praktické části práce.

Předposlední kapitolou je kapitola *Výsledky* 4. Zde je předložen návod pro zjištění optimálních hodnot parametrů $\gamma$ a $m$ algoritmu Kanai Suzuki. Dále bude zpracováno srovnání Dijkstrova a Kanai Suzuki algoritmu z pohledu času a přesnosti výpočtu nejkratší cesty.

Poslední kapitolou je *závěr* 5 ve ktrém budou ve stručnosti shrnuty poznatky o algoritmech pro výpočet nejkratší cesty.

# Kapitola 2

# Rozbor problematiky

## 2.1 Trojůhelníkové sítě

Mnohostěnné povrchy, obzvláště sítě skládající se z rovinných trojůhelníkových ploch, jsou základní geometrickou reprezentací v počítačové grafice a souvisejících oblastech. Proces generování trojúhelníkových sítí označujeme jako triangulaci. V současné době existuje celá řada triangulačních metod. Triangulační metody můžeme rozdělit na metody nepřímé, metody příme, metody využívající kvadrantových stromů atd. Popis těchto metod lze naleznout v referenční literatuře [11]

## 2.2 Dijkstrův algoritmus

Jako zdroj informací pro tuto kapitolu byla použita tato literatura: [9] [8] [6]

Dijkstrův algoritmus byl poprvé popsán nizozemským informatikem Edsgerem Dijkstrou. Algoritmus slouží k nalezení nejkratší cesty v kladně ohodnoceném grafu. Pro grafy se záporným ohodnocením hran lze využít algoritmus Bellman-Fordův, který je však pomalejší.

Algoritmus je určen pro výpočet nejkratší cesty mezi zdrojovým bodem $s$ a všemi ostatními body modelu. Pro výpočet nejkratší cesty mezi dvěma body modelu (zdrojovým $s$ a cílovým $t$) lze však tento algoritmus využít také. V tomto případě se algoritmus ukončí, jakmile je cílový bod $t$ prohlášen za trvalý. Nejjednodušší implementace Dijkstrova algoritmu používá pro uložení prioritní fronty pole a má asymptotickou časovou složitost $O(|V|^2 + |E|)$, kde $|V|$ je počet vrcholů a $|E|$ počet hran. Implementace využívající pro prioritní frontu binární haldy je schopná pracovat s časovou složitostí $O((|E|+|V|)\log|V|)$. V rozhodnutí, kterou z implementací je vhodné použít, nám napomůžou následující vztahy. Pokud je počet hran $E = \Omega(V^2)$, je vhodnější implementace v poli. Pokud počet hran klesne pod $V^2/\log V$, výhodnějším řešením je binární halda.

Dijkstrův algoritmus je konečný, protože v každé iteraci prohlásí jeden vrchol za trvalý.

### 2.2.1 Popis algoritmu

Dijkstrův algoritmus rozděluje vrcholy grafu do dvou skupin, na vrcholy s trvalým ohodnocením a dočasným ohodnocením. Ohodnocením vrcholu rozumíme délku cesty od zdrojového vrcholu $s$ k danému vrcholu. Na začátku jsou všechny vrcholy dočasné a nejkratší cesta k těmto vrcholům je nastavena na nekonečno s vyjímkou startovního vrcholu $s$, který má nejkratší cestu nastavenu na nula. Trvalé ohodnocení je takové, které už nebudeme

měnit, protože o něm víme, že lepšího ohodnocení již nemůžeme dosáhnout. Celý algoritmus pracuje ve třech krocích:

1. nalezení vrcholu V s minimálním dočasným ohodnocením
2. prohlášení vrcholu V za trvalý
3. změna ohodnocení sousedů tak, že $|N| = min(|N|, |V| + |VN|)$, kde $N$ je soused $V$, $|N|$ je ohodnocení sousedního vrcholu a $|V|$ ohodnocení trvalého vrcholu V. $|VN|$ označuje délku hrany mezi vrcholy $V$ a $N$

Nejdříve je v prvním cyklu nalezen vrchol s minimálním dočasným ohodnocením (v prvním průchodu cyklu nalezneme vrchol s hodnotou nula). Tento vrchol bude prohlášen za trvalý. Nyní přejdeme na krok druhý, ve kterém nalezneme všechny sousedy (vrcholy, do kterých vede z trvale ohodnoceného vrcholu hrana). Tyto vrcholy jsou dočasné a jejich ohodnocení je v prvním cyklu iterace nastaveno na nekonečno. Podle třetího kroku ohodnocení těchto bodů získáme jako součet ohodnocení trvalého bodu a délky hrany vycházející z tohoto bodu k bodu dočasnému. Nyní již můžem vstoupit do druhé iterace cyklu. V této iteraci nalezneme další dočasný bod s minimálním ohodnocením, přidáme jej do trvalých, zjistíme všechny jeho sousedy a v případě nutnosti je podle třetího kroku cyklu přehodnotíme.

Tyto tři kroky se opakují v cyklu, dokud cílový bod $t$ není prohlášen za trvalý. Prohlášení cílového bodu za trvalý znamená, že z počátečního vrcholu do vrcholu cílového je již známa nejkratší cesta a nelze najít jinou, kratší cestu.

Vlastní nalezení nejkratší cesty probíhá při zpětném průchodu od cílového vrcholu tak, že máme pomocné pole indexované vrcholy, a pro každý vrchol zde bude uloženo číslo vrcholu, ze kterého jsme do tohoto vrcholu našli nejkratší cestu.

### 2.2.2 Pseudokód

Nechť vrchol `Vs` odpovídá startovnímu vrcholu výpočtu, `Vt` vrcholu cílovému, `vertices` seznamu všech vrcholů mnohostěnu, `VerticesVisited` seznamu vrcholů, prohlášených za trvalé a `VerticesNotVisited` seznamu dočasně ohodnocených vrcholů. Dále pole `distance[]` indexované vrcholy obsahuje pro každý vrchol nejkratší cestu, pole `path[]` indexované vrcholy obsahuje předchozí vrchol, ze kterého se po nejkratší cestě do vrcholu indexovaného dostaneme. Pak lze algoritmus pseudokódem popsat následovně:

```
function Dijkstra(Vs, Vt, vertices, edges):
   //inicializace
   for each vertex in vertices:
      add vertex to VerticesNotVisited
      distance[vertex] := infinity
      path[vertex] := undefined
   distance[Vs] := 0
   //vlastní cyklus algoritmu
   while(Vact ≠ Vt):
      //aktuální zpracovánaný vrchol je vrchol
      //s nejmenším dočasným ohodnocením
      Vact := VerticesNotVisited & min(distance[])
      add vertex to VerticesVisited & delete vertex from VerticesNotVisited
      //pokud je třeba, změna přehodnocení sousedů
```

```
for each neighbor Vact from VerticesNotVisited
   if distance[neighbor] > distance[Vact] + length edge(neighbor,Vact):
      distance[neighbor] = distance[Vact] + length edge(neighbor,Vact)
      path[neighbor] = Vact
```

## 2.3 Aproximační algoritmy

Existuje velké množství algoritmů pro nalezení nejkratší cesty jak pro 2D, tak i pro 3D povrchy. Nicméně algoritmy pro nalezení přesné nejkratší cesty na mnohostěnných površích mají většinou velké nároky jak na výpočetní čas, tak i na výpočetní prostor. Proto tyto algoritmy není vhodné používat pro husté mnohostěnné povrchy. Tuto nevýhodu se snaží odstranit algoritmy aproximační.

Nejkratší cesta vypočtená pomocí těchto algoritmů typicky prochází skrz plochy trojúhelníků. Ve většině případů nelze proto využít při výpočtu Dijkstrova algoritmu. Oproti tomu se zde využívá vlastností geodetik, geodetické křivosti, tečné roviny atd. Mezi zástupce těchto algoritmů patří Martínez, Velho a Carvalho [1], algoritmus Mgr. Porazilové [4] a další.

Jako dalšího zástupce z řady aproximačních algoritmů můžeme uvést například algoritmus Lanthier [2]. Tento algoritmus redukuje původní problém na hledání nejkratší cesty v aproximačním diskrétním váženém grafu. Algoritmus nejprve přidá body doprostřed hran. Dále je vytvořen diskrétní graf z těchto bodů a bodů originálních. V následujícím kroku je aplikován Dijkstrův algoritmus. Podobným způsobem pracuje i Kanai Suzuki algoritmus.

## 2.4 Algoritmus Kanai a Suzuki

Algoritmus Kanai Suzuki [10] se zaměřuje na vyhledání aproximační nejkratší cesty. Tento algoritmus je postaven na využití Dijkstrova algoritmu a využívá výběrového upřesňování diskrétního grafu mnohostěnu. Dijkstrův algoritmus je zde využíván v každé iteraci hlavního výpočetního cyklu k omezení části grafu, ve kterém existuje nejkratší cesta.

Z tohoto základního popisu je zřejmé, že celkový potřebný čas pro výpočet algoritmu velmi závisí na čase výpočtu nejkratší cesty pomocí Dijkstrova algoritmu. Všeobecně by se dalo říci, že časová náročnost algoritmu se může vyčíslit jako $O(n \log n)$. Záleží však i na počtu hran v inicializačním grafu $G^0$ a tudíž i na parametru $\gamma$, který určuje kompromis mezi přesností aproximace a časovými nároky algoritmu.

Algoritmus Kanai Suzuki má i některé významné výhody, kterými jsou např. rychlost, vysoká přesnost výpočtu a menší paměťové nároky.

### 2.4.1 Popis algoritmu

Nechť $M$ je povrch mnohostěnu, nejkratší cesta $L$ ze zdrojového vrcholu $V_s$ do vrcholu cílového $V_d$ povrchu $M$ je definována jako:

$$L = \{v_1, v_2, ..., v_n, e_1, e_2, ..., e_{n-1}\},$$

$$V_s = v_1, V_D = v_n, e_i = \{v_i, v_{i+1}\}, i = 1...n-1,$$

kde $v_i$ a $e_i$ značí vrchol a hranu plochy $M$.

Nechť $l = |L|$ je délka nejkratší cesty $L$. Pak platí:

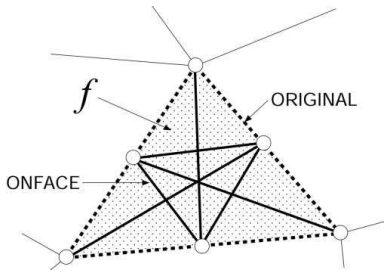
$$l = |L| = |e_1| + |e_2| + ... + |e_{n-1}|,$$

kde $|e_i|$ je Euklidovská vzdálenost mezi vrcholy $v_1$ a $v_{i+1}$ hrany $e_i$.

Algoritmus definuje rozdělení hran povrchu $M$ do dvou skupin:

- seznam seznam hran ORIGINAL se odkazuje na hrany originální nebo hrany vzniklé rozdělením těchto hran,
- seznam hran ONFACE obsahuje hrany ostatní.

Rozdělení je lépe pochopitelné z obrázku 2.1. Seznamy slouží k posouzení, zda hrana bude v průběhu algoritmu dělena SPs body či nikoliv.




Obrázek 2.1: Rozdělení hran

Steinerovy body (dále jen SPs) jsou body, které přidáváme při běhu algoritmu do grafu pro výpočet. Počet Steinerových bodů přidaných pro každou hranu závisí na délce této hrany $|e|$ a na parametru $\gamma$. Označme délku originální hrany $|e|$, pak počet Steinerových bodů přidaných hraně odpovídá vztahu $(\gamma/|e|) - 1$.

$\gamma$ je uživatelem definovaná proměnná určující kompromis přesnosti aproximace proti paměťovým a časovým nárokům algoritmu.

### 2.4.2 Inicializace algoritmu

Inicializace algoritmu zahrnuje vytvoření počátečního diskrétního grafu $G^0(v, e)$ z hran a vrcholů povrchu $M$. Graf $G^0(v, e)$ obsahuje všechny vrcholy a hrany povrchu $M$ plus přidané hrany, které vznikly přidáním vrcholů (SPs).

Přidané hrany grafu $G^0$ jsou vytvářeny pomocí originálních a přidaných vrcholů. Hrany přidáváme pokud vrcholy přidávané hrany jsou vedle sebe na originální hraně (rozdělíme originální hranu na více částí), nebo pokud vrcholy přidávané hrany jsou na různých stranách trojůhelníku.

### 2.4.3 Vlastní algoritmus

Algoritmus pracuje v cyklu, který je ukončen, pokud $|l^i - l^{i-1}| < \epsilon$. Při každém průchodu cyklem je inkrementováno počítadlo průchodů $i$ s inicializační hodnotou $i = 0$.

Proměnná $m$ používaná u třetího kroku iterace určuje kompromis mezi snížením počtu iterací a zvýšením potřebného výpočetního času a paměti.

Jednotlivé kroky iterace:

1. Výpočet nejkratší cesty $L^i$ mezi zdrojovým $V_S$ a cílovým $V_D$ vrcholem grafu $G^i$ pomocí Dijkstrova algoritmu.

2. vytvoření nového grafu $G^{i+1}$ pomocí zjištěné nejkratší cesty $L^i$. Tato nejkratší cesta je vrchol po vrcholu procházena a všechny vrcholy této cesty jsou přidány do $G^{i+1}$. Dále jsou přidány i vrcholy, které jsou spojeny hranou s vrcholem cesty příslušející některé ORIGINAL hraně.

3. Přidání SPs a hran do grafu $G^{i+1}$. SPs vrcholy přidáváme pouze hranám, které jsou označeny jako ORIGINAL. Počet přidaných SPs vrcholů odpovídá uživatelské proměnné $m$. Hrany jsou mezi nové vrcholy přidány, pokud jsou na stávající hraně vedle sebe (hrana se rozdělí na části) a nová hrana je přidána do seznamu ORIGINAL, nebo pokud vrcholy přidávané hrany leží na dvou rozdílných stranách trojúhelníku. Tyto hrany jsou přidány do seznamu ONFACE.

4. Aktualizace grafu. Graf $G^i$ je aktualizován na graf $G^{i+1}$.

### 2.4.4 Pseudokód

Nechť vrchol `Vs` odpovídá startovnímu vrcholu výpočtu, `Vt` vrcholu cílovému, `vertices` seznamu všech vrcholů mnohostěnu, `edges` seznamu všech hran mnohostěnu, `tris` seznamu všech trojúhelníků mnohostěnu, `epsilon, gamma, m` uživatelským proměnným, se kterými program pracuje. Dále `original` odpovídá seznamu originálních hran. Pak lze algoritmus pseudokódem popsat následovně:

```
function KanaiSuzuki(Vs, Vt, vertices, edges, tris, epsilon, gamma, m):
   //přidání všech hran do seznamu original
   for each edge:
      add edge to original
   //přidání SPs vrcholů a hran, které vznikly rozdělením
   //originál hran a odstranění rozdělených original hran
   AddSPsEdges(vertices, edges, original, gamma)
   //přidání hran, jejichž vrcholy leží na odlišných
   //stranách trojůhelníku
   AddOppositeEdges(tris, edges, vertices)
   //triangulace nově vzniklé sítě vrcholů a hran
   GenerateNewTris(tris, edges, vertices)

   //vlastní cyklus algoritmu
   i = 0
   while(není dosaženo požadované přesnosti):
      //vypocet nejkratsi cesty Dijkstrovým algoritmem
      result = Dijkstra(Vs, Vt, vertices, edges)
      //pridani vrcholu obsazenych v nejkratsi ceste
      // do vrcholu pro dalsi iteraci
      MakeNewGraph(VNext, vertices, result->path)
      //pridání vrcholů nacházejících se v seznamu original
      //spojených hranou s některým z vrcholů cesty
      AddOriginalVertices(VNext, vertices, original, result->path)
      //přidání hran, kde v předchozí iteraci byly
      //plus přidání těchto hran do seznamu original
      MakeNewEdges(VNext, vertices, ENext, edges, ONext, original)
```

```
//triangulace nově vzniklé sítě vrcholů a hran
GenerateNewTris(TNext, ENext, VNext)
//přidání SPs vrcholů a hran, které vznikly rozdělením
//originálních hran a odstranění rozdělených original hran
AddSPsEdges(VNext, ENext, ONext, m)
//přidání hran, jejichž vrcholy leží na odlišných
//stranách trojůhelníku
AddOppositeEdges(TNext, ENext, VNext)
//triangulace nově vzniklé sítě vrcholů a
//hran pro další iteraci
GenerateNewTris(TNext, ENext, VNext)
//aktualizace grafu
tris = TNext, edges = ENext
vertices = VNext, original = ONext
i++
```

*Pozn.: Bližší popis funkcí zahrnutých v pseudokódu naleznete v sekci implementace 3.3.2*

# Kapitola 3

# Implementace

V této kapitole se budeme zabývat vlastní implementací programu pro výpočet nejkratší cesty po povrchu objektu. Jako implementační jazyk byl zvolen jazyk C/C++, proto bude tato kapitola úzce spjata s tímto jazykem. Dále bude v této kapitole představen toolkit MDSTk vyvýjený na Fakultě informačních technologií v Brně, zejména jeho část VectorEntity. Představíme si i OpenSceneGraph, což je nadstavba OpenGL pro vytváření složitých 3D scén.

## 3.1 Medical Data Segmentation Toolkit

Medical Data Segmentation Toolkit (dále jen MDSTk) [5] je soubor multiplatformních OpenSource knihoven (BSD-like license) a nástrojů pro zpracování 2D/3D obrazu. Obsahuje podporu pro volumetrická obrazová data, knihovny třetích stran (ATLAS, CLAPACK, UMFPACK), formáty obrázků JPG a PNG. Implementace byla provedena v jazyce C++ a celý systém je navržen jako množina konzolových aplikací komunikujících mezi sebou pomocí jednotného rozhranní. MDSTk vyvíjí Fakulta informačních technologií VUT v Brně, resp. Ústav počítačové grafiky a multimédií.

Důležitou součástí MDSTk je low-level knihovna pro efektivní reprezentaci mnohostěnných modelů a trojúhelníkových sítí. Této knihovně se bude věnovat další část této kapitoly.

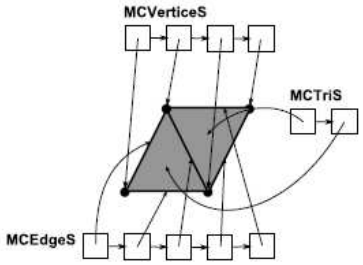
### 3.1.1 VectorEntity

Jak již bylo výše zmíněno, VectorEntity je součást toolkitu MDSTk, pro poskytnutí efektivní reprezentace mnohostěnů a trojůhelníkových sítí. Tato knihovna se skládá z 18 tříd. Tyto třídy slouží pro uložení a zpracovávání dat 3D bodů, vrcholů, hran, trojúhelníků atd.

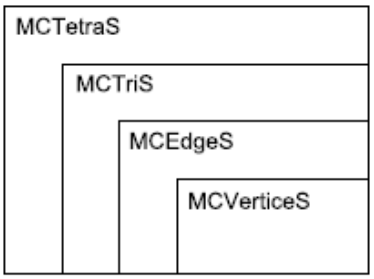
Knihovna obsahuje čtyri typy geometrických entit. Jsou to vrcholy, hrany, trojúhelníky a čtyřstěny. Pro všechny entity je k dispozici obousměrný seznam, ve kterém jsou entity uloženy 3.1. Všechny vyšší typy entit (hrany, trojúhelníky i čtyřstěny) jsou složené z vrcholů.

Dále jsou v knihovně zastoupena grafická primitiva. Jedná se především o objektově zapouzdřené obsluhy vrcholu, hrany, trojúhelníku a čtyřstěnu. Pro všechna grafická primitiva existují i kontejnery pro jejich uložení. Pro ilustraci je uveden obrázek rozložení kontejnerů 3.2.

Za zmínku ještě jistě stojí, že knihovna umožňuje import a export modelů v binárním formátu STL a export do formátu VRML.




Obrázek 3.1: Geometrické entity




Obrázek 3.2: Rozložení kontejnerů

Pro zjištění dalších informací o knihovně VectorEntity můžete nahlédnout do referenční literatury, nebo dokumentace vygenerované k této knihovně programem doxygen.

## 3.2 OpenSceneGraph

OpenSceneGraph (dále jen OSG) [7] je multiplatformní soubor nástrojů pro vývoj výkoných grafických aplikací jako jsou letecké simulátory, hry, virtuální realita a vědecké vizualizace. OSG poskytuje objektově orientovanou konstrukci založenou na OpenGL, která osvobozuje vývojáře od programování grafických aplikací na nízké úrovni OpenGL a poskytuje mnoho dodatečných nástrojů pro rychlý vývoj grafických aplikací. OSG je kompletně napsán v jazyce C++, což umožňuje tvorbu dalších knihoven. Mezi hlavní výhody OSG patří výkon, přenositelnost a rozšiřitelnost.

Datové struktury OSG jsou popsány v literatuře [3]. OSG používá pro zobrazení graf scény. Grafová reprezentace scény je výhodná zejména kvůli efektivitě renderování, jelikož zjednodušuje a urychluje procesy jako je určování viditelnosti, řazení apod. Graf scény má obvykle tvar stromu. V listech stromu se nacházejí informace o geometrii a materiálech, které objekty používají, zatímco vnitřní uzly stromu definují prostorové a logické skupiny těchto objektů.

Jako základní datový typ pro všechny typy uzlů v rámci grafu scény slouží třída **Node**, která obsahuje seznam ukazatelů na své rodičovské uzly a přístupové metody pro různá uživatelská data.

Datový typ pro vnitřní uzly grafu se nazývá **Group** a instance tohoto typu obsahují seznam ukazatelů na své potomky. Díky své přizpůsobivosti lze objektům a třídám, odvozených od **Group** nastavit nejrůznější vlastnosti. Kromě seskupování uzlů dle určitých

pravidel lze uzly tohoto typu použít například pro sestrojení různých typů stromů, definici různých úrovní detailů, umísťování podstromů pomocí matic atd.

Listové uzly jsou tvořeny speciálními uzly, pro něž je používán typ **Geode**. Tyto uzly obsahují seznam objektů typu **Drawable**, reprezentující zobrazitelná data.

## 3.3 Vlastní implementace

V této sekci bude popsána vlastní praktická část bakalářské práce. Nejprve uvedu základní popis aplikace, přehled modulů, ze kterých se praktická část práce skládá a poté budou blíže popsány jednotlivé moduly a funkce v nich obsažené.

### 3.3.1 Základní popis implementace

Cílem praktické části bakalářské pracé bylo implementovat dva algortimy pro výpočet nejkratší cesty po povrchu modelu. Jak vyplývá z předchozího textu, implementovány byly algoritmy Dijkstrův a aproximační algoritmus Kanai Suzuki. Pro implementaci Dijkstrova algoritmu byl zvolen přístup s uložením prioritní fronty do pole, jelikož modely většinou obsahují velké množství hran.

Výsledkem implementace je konzolová aplikace s grafickým výběrem počátečního a koncového bodu nejkratší cesty. Tento výběr probíhá pomocí pravého tlačítka myši. Dále je možné modelem pomocí levého tlačítka myši libovolně otáčet a pomocí pravého tlačítka, po zobrazení nejkratší cesty, model přiblížit či oddálit. Zmenšení grafického výstupu do okna (a naopak) lze provést stisknutím klávesy F. Celá grafická část programu byla napsána pomocí OSG.

Jak jsem již zmínil, jedná se o konzolovou aplikaci. Proto ostatní funkčnost programu závisí na parametrech příkazové řádky, které jsou aplikaci předány. Syntax programu je následující:

- Pro zobrazení nápovědy programu
  `shortest_path -h`
- Pro výpočet nejkratší cesty pomocí Dijkstrova algoritmu
  `shortest_path -model` model.stl `-metoda` dijkstra
- Pro výpočet pomocí algoritmu Kanai Suzuki
  `shortest_path -model` model.stl `-metoda` ks `-přesnost` des_číslo `-gamma` des_číslo `-m` celé_číslo

*Pozn.: Parametry je nutné zadat přesně v tomto formátu.*
Jednotlivé parametry programu a jejich význam:

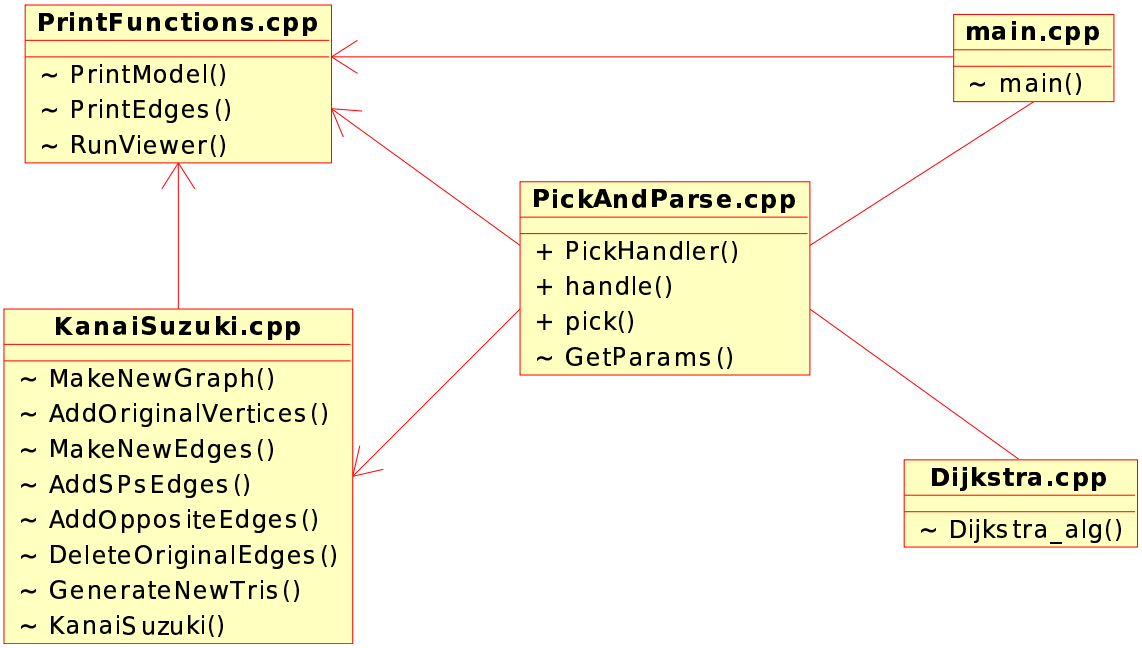
- `-model` model.stl
  Parametr model.stl určuje, na jakém modelu bude výpočet probíhat. Zadaný model musí být v binárním formátu STL.
- `-metoda` metoda
  Jak je zřejmé z syntax programu, metoda slouží pro určení metody, kterou bude výpočet prováděn. Pro výpočet pomocí Dijkstrova algoritmu použijeme `-metoda dijkstra`, pro výpočet pomocí algoritmu Kanai Suzuki `-metoda ks`

- `-přesnost` des_číslo
  Parametr des_číslo určuje přesnost výpočtu nejkratší cesty u algoritmu Kanai Suzuki.

- `-gamma` des_číslo
  Parametr des_číslo je uživatelem definovaná hodnota určující kompromis přesnosti aproximace proti paměťovým a časovým nárokům algoritmu Kanai Suzuki. Ve skutečnosti se jedná o hodnotu, kterou když délka hrany $n$krát překročí, bude rozdělena *n-1* SPs body při vytváření grafu $G^0$.

- `-m` celé_číslo
  Parametr celé_číslo je uživatelem definovaná hodnota udávající kompromis mezi snížením počtu iterací a zvýšením potřebného výpočetního času a paměti. Ve skutečnosti se jedná o počet SPs vrcholů, které budou přidány do každé z originálních hran grafu $G^i$, kde $i > 0$.

### 3.3.2 Moduly implementace




Obrázek 3.3: Moduly implementované aplikace

Na obrázku 3.3 jsou znázorněny moduly implementované v aplikaci. Šipkami jsou spojeny moduly, mezi kterými probíhá jednosměrná komunikace. Obousměrná komunikace mezi moduly je označena jako úsečka mezi moduly.

#### Popis vlastního programu

Při spuštění program přejde do modulu `main.cpp`, konkrétně do funkce `main()`. Jako první se ve funkci `main()` zavolá funkce modulu `PickAndParse.cpp GetParams()`, která zpracuje parametry příkazové řádky a předá je zpět funkci `main()`. Poté funkce `main()` načte model z binárního souboru formátu stl. Následuje vytvoření trojůhelníkové sítě, příslušných

hran a vrcholů. Nyní je hlavní funkce programu schopna uložit model do pole pro vykreslení prostřednctvím OSG. Toto se děje prostřednictvím funkce `PrintModel()` uložené v modulu `PrintFunctions.cpp`. Posledním úkolem hlavní funkce programu je zobrazení modelu pomocí OSG. K tomu využívá funkci `RunViewer()` uloženou v modulu `PrintFunctions.cpp`.

Jakmile hlavní funkce programu spustí vykreslení, je předána kontrola funkci `handle()` třídy `PickHandler`, která se nachází v souboru `PickAndParse.cpp`. Tato funkce běží ve smyčce, dokud nedojde k vypnutí vykreslovacího okna a stará se o všechny události, které je okno schopné zpracovávat. Pomocí funkce `pick()` stejné třídy, kterou funkce `handle()` používá pro výpočet startovního a cílového bodu, zjistíme tyto dva body. Jakmile je startovní a cílový bod znám, spustí funkce výpočet nejkratší cesty pomocí Dijkstrova nebo Kanai Suzuki algoritmu. K tomu využívá funkce `Dijkstra_alg()` uložené v modulu `Dijkstra.cpp`, resp. `KanaiSuzuki()` modulu `KanaiSuzuki.cpp`. U Dijkstrova algoritmu navíc funkce `handle()` zajišťuje zobrazení nejkratší cesty na modelu. Toto provádí prostřednictvím funkce `PrintEdges()` modulu `PrintFunctions.cpp`.

Jakmile je uzavřeno vykreslovací okno, funkce `handle()` předá řízení zpět hlavní funkci programu(`main()`). Nyní již nezbývá nic jiného, než program ukončit.

Více informací o jednotlivých funkcích získáte v následující části textu. 3.3.2

### Stručný popis jednotlivých funkcí programu

Funkce popisované v této části textu budou uvedeny bez parametrů, které jim jsou předávány. Toto omezení zavádím kvůli čitelnosti a přehlednosti textu. Je však možné, že bude zmíněn v komentáři pod funkcí nějaký parametr. To pouze z důvodu lepší pochopitelnosti jeho významu. Detailní informace je možné zjistit ve zdrojových kódech programu.

Funkce modulu `KanaiSuzuki.cpp`:

- `void MakeNewGraph()`
  Funkce pro přidání vrcholů, obsažených v nejkratší cestě zjištěné Dijkstrovým algoritmem, do grafu následující iterace $G^{i+1}$.

- `void AddOriginalVertices()`
  Funkce pro přidání vrcholů do grafu následující iterace $G^{i+1}$, které jsou spojeny hranou s některým vrcholem cesty a zároveň leží na originální hraně grafu $G^i$.

- `void MakeNewEdges()`
  Funkce pro přidání hran mezi vrcholy grafu $G^{i+1}$ tam, kde byly v grafu $G^i$. Dále funkce zjistí, zda hrana grafu $G^i$ byla originální. Pokud byla, přidáme ji do seznamu originálních hran i pro graf $G^{i+1}$.

- `void AddSPsEdges()`
  Funkce pro přidání SPs vrcholů a hran, které vrcholy vytvořili rozdělením originální hrany grafu $G^i$, do grafu $G^{i+1}$. Funkce uloží přidané hrany do hran originálních grafu $G^{i+1}$ a naplní vektor `hrana` pro uchování informací o rozdělené hraně. Parametry `gamma` a `m` ovlivňují počet přidaných SPs bodů.

- `AddOppositeEdges()`
  Funkce pro přidání hran ležících na různých hranách trojúhelníku do grafu $G^{i+1}$. Informace o hranách a vrcholech čerpá z vektoru `hrana`.

- `void DeleteOriginalEdges()`
  Funkce pro odstranění originálních hran grafu $G^{i+1}$, které byly rozděleny SPs body

a které jsou tím pádem v grafu $G^{i+1}$ zbytečné. Zároveň probíhá i odstranění hran ze seznamu originálních hran grafu $G^{i+1}$

- `void GenerateNewTris()`
  Funkce pro vytvoření trojúhelníkové sítě pomocí vrcholů a hran.

- `void KanaiSuzuki()`
  Funkce pro vlastní výpočet nejkratší cesty pomocí algoritmu Kanai Suzuki. Funkce po vypočítání nejkratší cesty dle zadané přesnosti vykreslí nejkratší nalezenou cestu.

Funkce modulu `Dijkstra.cpp`:

- `dijkstra_result Dijkstra_alg()`
  Funkce pro výpočet nejkratší cesty po povrchu objektu pomocí Dijkstrova algoritmu. Návratový typ `dijkstra_result` obsahuje tři části. Délku nejkratší cesty, pole vrcholů nejkratší cesty a pole vrcholů pro vykreslení nejkratší cesty pomocí OSG.

Funkce modulu `PrintFunctions.cpp`:

- `void PrintModel()`
  Funkce naplní pole vrcholů pro vykreslení pomocí OSG. Toto pole je interpretováno jako 3D pole trojúhelníků. Dále jsou nastaveny barvy trojúhelníkového modelu a vykresleny hrany pro zvýraznění obrysu modelu.

- `void PrintEdges()`
  Funkce zobrazující hrany modelu předaných této funkci polem pro vykreslení OSG. Parametrem `path` lze nastavit, zda se jedná o zobrazení nejkratší cesty či o zobrazení hrany.

- `void RunViewer()`
  Funkce pro zobrazení grafického okna. Mimo jiné zajišťuje rakce na změny velikosti okna a nastavuje osvětlení modelu.

Funkce modulu `PickAndParse.cpp`:

- `bool PickHandler::handle()`
  Funkce pro zpracovávání událostí myši grafického okna.

- `void PickHandler::pick()`
  Funkce pro zjišťění startovního a cílového bodu pro výpočet nejkratší cesty

- `params GetParams()`
  Funkce pro zpracování parametrů příkazové řádky. Návratový typ `params` obsahuje všechny důležité parametry pro spuštění a běh programu jako model, metoda atd.

# Kapitola 4

# Výsledky

V této kapitole bude nejprve popsán způsob, jak zjistit parametry $\gamma$ a $m$, které jsou pro daný model nejvhodnější z hlediska výpočetního času algoritmu Kanai Suzuki. Jakmile budeme schopni zadat optimální hodnoty parametrů $\gamma$ a $m$, porovnáme obě metody pro výpočet nejkratší cesty po povrchu modelu z hlediska časového a z hlediska přesnosti výpočtu. Pro zpracování výsledků a testování praktické části práce byly použity modely uvedené v tabulce 4.1.

| **Model** | **Počet** | | | **Průměrná délka hrany** |
|---|---|---|---|---|
| | **vrcholů** | **hran** | **trojúhelníků** | |
| hranol | 8 | 18 | 12 | 98,8069 |
| válec | 66 | 192 | 128 | 49,3316 |
| koule | 482 | 1 440 | 960 | 10,7603 |
| toroid | 512 | 1 536 | 1 024 | 8,0853 |
| bunny | 1 048 | 3 138 | 2 092 | 7,1114 |
| sierpinsky | 8 200 | 30 732 | 20 488 | 0,0544 |
| goldberg | 21 280 | 63 900 | 34 920 | 0,1293 |

Tabulka 4.1: Tabulka modelů.

## 4.1 Optimální volba parametrů $\gamma$ a $m$ algoritmu Kanai Suzuki

Pro zjištění optimálních hodnot parametrů $\gamma$ a $m$ algoritmu Kanai Suzuki vyjdeme ze dvou modelů. Modelu hranolu a válce. Trojúhelníkové sítě těchto modelů můžeme považovat za krajní hodnoty.

Model hranolu obsahuje hrany, které můžeme z hlediska jejich délky považovat téměř za totožné. Rozdíl délek hran modelu hranolu se liší od průměrné hodnoty délky hrany modelu maximálně o třicet procent průměrné hodnoty délky hrany.

Naproti tomu model válce obsahuje hrany, které se liší svojí délkou od průměrné délky hrany velice výrazně (délka průměrné hrany je 49,3316 a délka hran se pohybuje mezi 7,8414 až 100,307). Tento model můžeme tedy brát za druhý extrém co se délek hran týče.

Pro uživatelem definovaný parametr $\gamma$ jsou délky hran určujícím faktorem. Jak již bylo popsáno v kapitole 2.4.1, hrany jsou v inicializaci algoritmu rozděleny SPs body právě podle

hodnoty parametru $\gamma$.

Na obou modelech byl provedel experiment, který nám může být návodem pro zjištění optimálních hodnot parametrů $\gamma$ a $m$. Jako vstup programu byly zadávány různé kombinace hodnot parametrů $\gamma$ a $m$. Výstupem experimentu byl čas v milisekundách, který byl potřeba k výpočtu nejkratší cesty pomocí algoritmu Kanai Suzuki. Parametr *presnost* byl vždy konsktantí pro všechny výpočty. Tabulky 4.2 a 4.3 uvádějí čas v milisekundách potřebný k výpočtu nejkratší cesty prostřednictví Kanai Suzuki algoritmu v závislosti na zadaných parametrech $\gamma$(zadána v procentech průměrné délky hrany) a $m$.

V případě modelu hranol, měl parametr přesnost hodnotu *presnost=0,01*. Pro model válce byla zadána *presnost=0,1*.

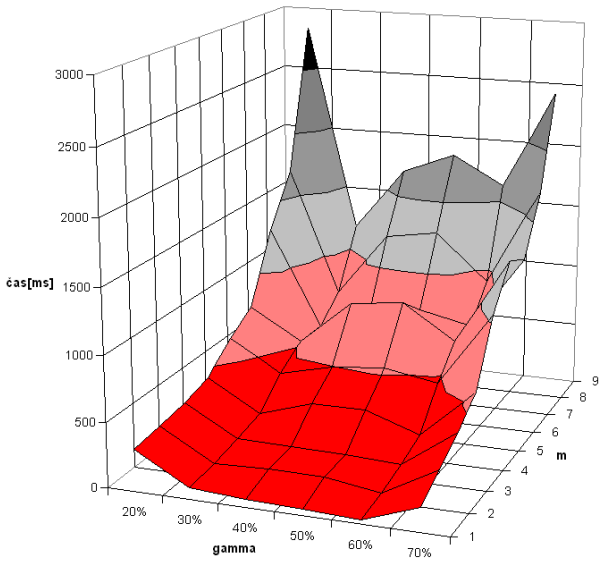
Jako ilustrace časové závislosti na parametrech $\gamma$ a $m$ jsou uvedeny grafy 4.1 a 4.2. Tyto grafy pro přehlednost obsahují pouze část hodnot tabulek 4.2 a 4.3, a to tu část, kde je dosaženo nejlepších časů výpočtu. Po zahrnutí i okrajových (špatných kombinací $\gamma$, $m$) by grafy neilustrovali nejlepší časové závislosti algoritmu a byly by nepřehledné.

| | **m** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **gamma** | **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10% | 3 685 | 3 779 | 3 801 | 3 927 | 4 133 | 4 328 | 4 325 | 4 557 | 5 210 |
| 20% | 323 | 346 | 387 | 445 | 631 | 765 | 1 296 | 1 701 | 2 843 |
| 30% | 88 | 113 | 152 | 209 | 391 | 529 | 527 | 701 | 1 169 |
| 40% | 59 | 132 | 145 | 340 | 497 | 887 | 704 | 1 215 | 1 673 |
| 50% | 49 | 125 | 143 | 342 | 533 | 936 | 752 | 1 286 | 1 849 |
| **60%** | 29 | 62 | 118 | 200 | 462 | 669 | 664 | 919 | 1 619 |
| 70% | 186 | 297 | 425 | 585 | 1 051 | 1 359 | 1 251 | 1 697 | 2 415 |
| 80% | 1 216 | 4 192 | 3 985 | 6 948 | 19 922 | 30 028 | 36 914 | 48 156 | 62 075 |
| 90% | 793 | 3 599 | 5 351 | 16 342 | 39 514 | 90 972 | 61 294 | 117 315 | 213 249 |

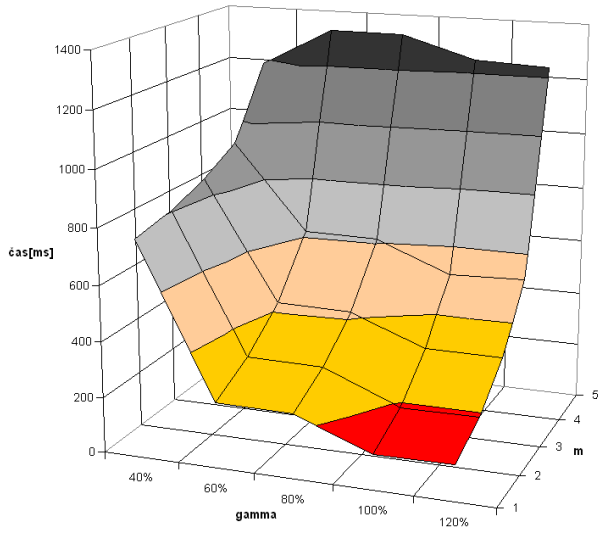
Tabulka 4.2: Závislost času výpočtu na parametrech $\gamma$ a $m$ pro model hranolu. Parametr *přesnost=0,01*.

| | **m** | | | | |
|---|---|---|---|---|---|
| **gamma** | **1** | 2 | 3 | 4 | 5 |
| 20% | 5 089 | 5 793 | 5 853 | 5 947 | 6 195 |
| 40% | 774 | 804 | 856 | 931 | 1 185 |
| 60% | 232 | 305 | 420 | 616 | 1 327 |
| 80% | 231 | 301 | 415 | 612 | 1 326 |
| **100%** | 125 | 190 | 312 | 505 | 1 245 |
| 120% | 127 | 191 | 311 | 514 | 1 235 |
| 140% | 831 | 2 931 | 5 707 | 12 069 | 32 044 |
| 160% | 836 | 2 937 | 5 689 | 12 064 | 32 069 |
| 180% | 830 | 2 938 | 5 686 | 12 060 | 31 917 |
| 200% | 831 | 2 938 | 5 688 | 12 059 | 31 976 |

Tabulka 4.3: Závislost času výpočtu na parametrech $\gamma$ a $m$ pro model válce. Parametr **přesnost=0,1**.




Obrázek 4.1: Graf závislosti času výpočtu na parametrech $\gamma$ a $m$ pro model hranolu. Parametr *přesnost=0,01*




Obrázek 4.2: Graf závislosti času výpočtu na parametrech $\gamma$ a $m$ pro model válce. Parametr *přesnost=0,01*

Zvýrazněné sloupce a řádky v tabulkách 4.2 a 4.3 odkazují na hodnoty argumentů $\gamma$ a $m$. Pro modely s délkami hran ne příliš rozdílnými je tedy nejlepší zvolit parametr *m=1* a parametr $\gamma$ nastavit na šedesát procent průměrné délky hrany. U modelů, kde se délky hran výrazně liší, je dobré zvolit také hodnotu *m=1*, ale hodnotu parametru $\gamma$ nastavíme na průměrnou délku hrany modelu.

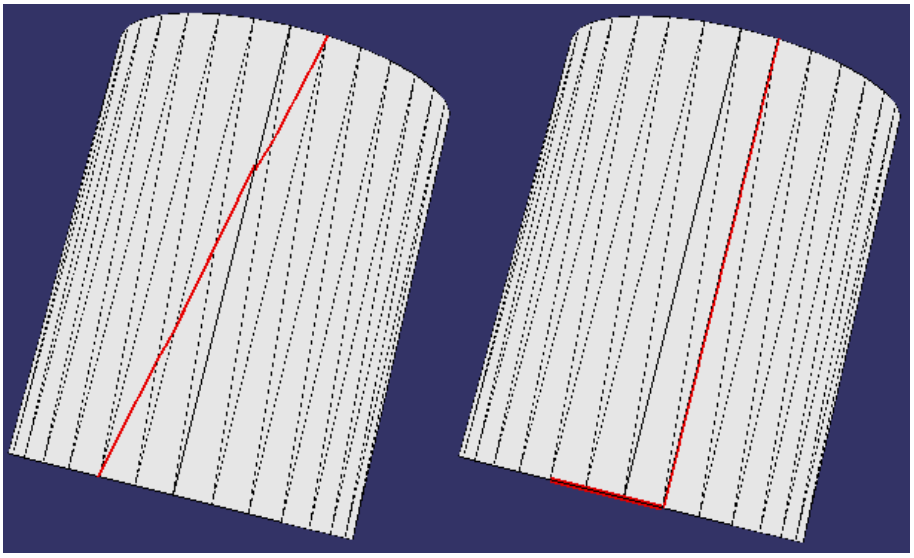
Mohlo by se zdát, že nastavením parametru $\gamma$ na osmdesát procent průměrné délky hrany dosáhneme kompromisu mezi objema možnostmi. Opak je ale pravdou. Pokud se podíváme pozorněji na tabulku 4.2, zjistíme, že pro tuto hodnotu má algoritmus v případě

modelů s podobnou délkou hran jednu z nejhorších časových závislostí.

## 4.2 Porovnání Dijkstrova a Kanai Suzuki algoritmu

V předchozí kapitole bylo popsáno, jak zadat nejlépe parametry $\gamma$ a $m$. Díky tomu víme i jak dosáhnout nejlepšího času výpočtu u algoritmu Kanai Suzuki. Díky této skutečnosti můžeme přejít k vlastnímu srovnání Dijkstrova a Kanai Suzuki algoritmu.

Jak už sám název aproximační napovídá, algoritmus Kanai Suzuki hledá nejkratší cestu aproximační metodou. Oproti tomu algoritmus Dijkstrův prochází pouze hrany modelu a hledá nejkratší cestu pomocí těchto hran. Nejkratší cesty vypočítané pomocí obou metod jsou znázorněny na obrázku 4.3.



Obrázek 4.3: Nejkratší cesta vypočítaná pomocí Kanai Suzuki algoritmu (vlevo) a pomocí Dijkstrova algoritmu (vpravo)

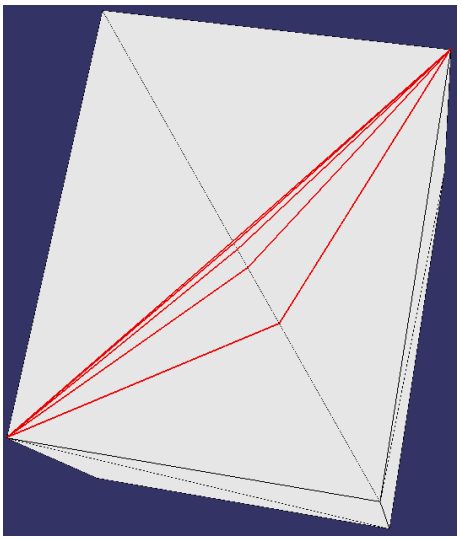
Na obrázku 4.4 jsou znázorněny jednotlivé aproximační kroky, kterými algoritmus Kanai Suzuki zjišťuje nejkratší cestu.

### 4.2.1 První experiment

Pro srovnání obou metod použijeme model bunny, který obsahuje 2 092 trojúhelníků. Je zřejmé, že algoritmus Dijkstrův dojde k výsledku rychleji, ale za cenu nižší přesnosti výpočtu.

Výsledky výpočtu pomocí Dijkstrova algoritmu:

- Nejkratší cesta: 118,888
- Hran nejkratší cesty: 17
- Čas výpočtu: 3030 ms
- Ostatní hodnoty odpovídají hodnotám bunny v tabulce 4.1



Obrázek 4.4: Zobrazení jednotlivých iterací algoritmu Kanai Suzuki

Pro výpočet pomocí algoritmu Kanai Suzuki použijeme hodnoty parametrů $\gamma = 7,114$, což je průměrná hodnota délky hrany. Vycházíme z doporučení v předchozí kapitole (hrany jsou v délkách od 1,5153 po 22,2682, což se od délky průměrné hrany značně liší). Parametr *m=1* a parametr *presnost=1*. Jednotlivé statistické údaje pro výpočet nejkratší cesty algoritmem Kanai Suzuki shrneme do tabulky 4.4.

| | **Počet** | | | | | | |
|---|---|---|---|---|---|---|---|
| **Iterace** | **vrcholů** | **hran** | **trojúhelníků** | **Dijkstra** | **KS** | **Hran** | **Cesta** |
| 0 | 1 326 | 3 841 | 2 848 | 4 614 | 10 488 | 17 | 118,888 |
| 1 | 933 | 8 323 | 12 356 | 2 791 | 262 308 | 28 | 112,946 |
| 2 | 2 701 | 61 034 | 167 115 | 23 335 | 23 338 | 44 | 112,625 |

Tabulka 4.4: Statistické informace o iteracích algoritmu Kanai Suzuki.

Jak je patrné z tabulky 4.4, v každé iteraci včetně nulté jsou přidávány do trojúhelníkové sítě modelu vrcholy. Po přidání vrcholů jsou mezi přidanými vrcholy vytvořeny hrany podle pravidel popsaných v kapitole 2.4.1. Následuje triangulace povrchu, čímž vzniká nová trojúhelníková síť.

V každém kroku iterace je nejdříve proveden výpočet nejkratší cesty pomocí Dijkstrova algoritmu. Poté následuje vlastní přidání vrcholů, hran a trojúhelníků. Již známe přesnou funkcionalitu algoritmu Kanai Suzuki a díky tomu můžem provést další porovnání.

Ve sloupcích Dijkstra a KS tabulky 4.4 jsou uloženy časy v milisekundách, které odpovídají časové náročnosti jednotlivých částí Kanai Suzuki algoritmu. V nulté iteraci je nejkratší cesta pomocí Dijkstrova algoritmu vypočítána za 4 614 ms. Na zbylé úkony algoritmu Kanai Suzuki tedy zbývá $10\,488 - 4\,614 = 5\,874$ ms. Během této doby je zúženo výpočetní pásmo pro nejkratší cestu na 933 vrcholů. Dijkstrův algoritmus bude ale v nadcházející iteraci procházet již 8 323 hran. Jelikož tento algoritmus po nalezení nejkratší cesty ze zdrojového bodu výpočtu nejkratší cesty do cílového ukončí svou činnost, je proveden za 2 791 ms. Další funkce algoritmu Kanai Suzuki pak zaberou bezmála 260 s. Pro další

Obrázek 4.5: Nejkratší cesta vypočtená po povrchu modelu bunny pomocí Dijkstrova algoritmu (vlevo) a Kanai Suzuki algoritmu (vpravo)

zpřesnění nejkratší cesty algoritmem Kanai Suzuki bude Dijkstrovým algoritmem vypočítávána nejkratší cesta již na 61 034 hranách. Tento výpočet trvá 23 335 ms. Po výpočtu zjistíme, že nejkratší cesta již odpovídá požadované přesnosti a ukončíme i Kanai Suzuki algoritmus. Nalezená nejkratší cesta ale obsahuje oproti nejkratší cestě nalezené Dijkstrovým algoritmem 44 hran, což je více než 2,5 násobek hran cesty vypočtené Dijkstrovým algoritmem. Obrázek 4.5 zobrazuje výsledky výpočtu nejkratší cesty pomocí obou algoritmů na modelu bunny.

Na modelu bunny proběhl tedy výpočet nejkratší cesty pomocí Dijkstrova algoritmu v době 3 030 ms. Pomocí algortimu Kanai Suzuki byla vypočtena nejkratší cesta za 297 381 ms, což je téměř stokrát více. Rozdíl ve vypočtených nejkratších cestách je $118,888 - 112,625 = 6,263$, což je přibližně 5 procent. Proto je nutné vždy uvážit, kdy který algoritmus použít.

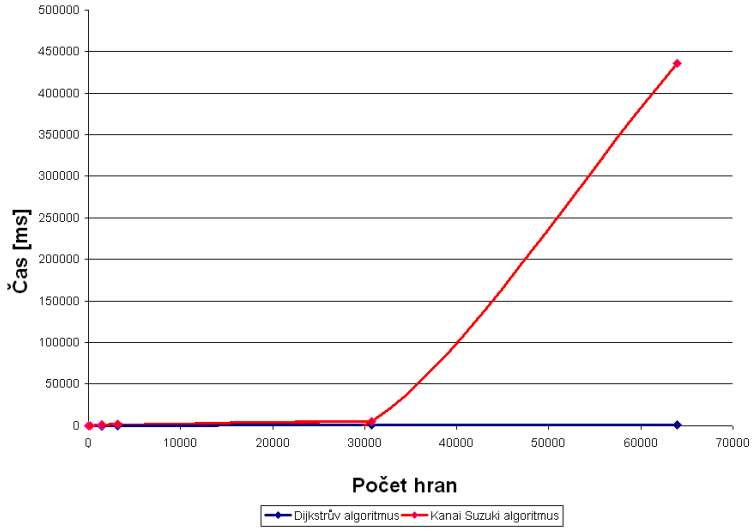
### 4.2.2 Druhý experiment

Pro druhý experiment bylo využito všech modelů uvedených v tabulce 4.1. Pro každý z těchto modelů byla zjištěna optimální hodnota parametrů $\gamma$ a $m$. Přesnosti byly nastaveny pro každý model na jednu desetinu průměrné délky hrany.

Samotný experiment spočíval ve výpočtu nejkratší cesty pomocí Dijkstrova i Kanai Suzuki algortimu. Nejkratší cesta byla vždy vyhledávána na segmentu podobném čtverci na obrázku 4.4, což zaručilo srovnatelné výsledky. Statistické údaje zjišťené tímto experimentem jsou shrnuty v tabulce 4.5. Sloupce *T Dijkstra* a *T KS* označují čas potřebný pro výpočet nejkratší cesty pomocí Dijkstrova, resp. Kanai Suzuki algoritmu v milisekundách. *NC* je označení pro délku nejkratší cesty.

| Model | Hran | T Dijkstry | NC Dijkstra | T KS | NC KS | Iterací KS |
|---|---|---|---|---|---|---|
| hranol | 18 | 0 | 180 | 30 | 128,062 | 2 |
| válec | 192 | 11 | 107,841 | 117 | 100,307 | 2 |
| koule | 1 440 | 29 | 22,5032 | 875 | 16,0294 | 3 |
| toroid | 1 536 | 21 | 15,9051 | 785 | 11,7034 | 3 |
| bunny | 3 138 | 114 | 21,022 | 1 864 | 16,1505 | 3 |
| sierpinski | 30 732 | 454 | 0,1051 | 4 332 | 0,0949 | 3 |
| goldberg | 63 900 | 896 | 0,1849 | 435 533 | 0,1686 | 3 |

Tabulka 4.5: Statistické informace vypočtené nejkratší cesty pro všechny modely.




Obrázek 4.6: Porovnání časové složitosti Dijkstrova a Kanai Suzuki algoritmu

Dále byl sestrojen graf závislosti výpočetního času na počtu hran 4.6. Z tohoto grafu je zřejmé, že časové nároky Kanai Suzuki algoritmu rostou velice rychle s přibývajícím počtem hran trojúhelníkového modelu.

### 4.2.3 Shrnutí

Z těchto experimentů vyplývá, že výpočet pomocí algoritmu Kanai Suzuki je velice časově náročný. Se stoupajícím počtem iterací nebo trojúhelníků v trojúhelníkové síti roste velice rychle i časová náročnost. Proto je důležité pečlivě zvážit, který z algoritmů pro výpočet nejkratší cesty použít.

# Kapitola 5

# Závěr

V této práci byly prezentovány dvě metody pro výpočet nejkratší cesty po povrchu trojúhelníkového modelu. Dijkstrův algoritmus jako optimální řešení tohoto problému a aproximační Kanai Suzuki algoritmus.

Porovnáním obou algoritmů jsem došel k závěru, že aproximační algoritmus Kanai Suzuki vypočítá sice nejkratší cestu mezi dvěma vrcholy trojúhelníkového modelu se zadanou přesností, ale jeho nevýhodou je větší potřebné množství výpočetního času. Narozdíl od algoritmu Kanai Suzuki algoritmus Dijkstrův proběhne v kratším čase, ale výsledek zpravidla nedosahuje tak vysoké přesnosti jako u algoritmu aproximačního.

Pro vylepšení časových vlastností aproximačního algoritmu Kanai Suzuki jsem se rozhodl neudržoval seznam hran ONFACE. Udržovat tento seznam nemá pro běh algoritmu opodstatnění. Veškeré operace důležité pro běh programu totiž vychází ze seznamu hran ORIGINAL.

Dalším způsobem pro vylepšení časových vlastností tohoto algoritmu je jistě navrhnutí datových struktur přímo pro tento algoritmus. Jelikož jsem i v tomto algoritmu využíval knihovnu VectorEntity, není tento algoritmus jistě zpracován nejlepším možným způsobem. Vhodným navržením datových struktur by se jistě ušetřilo mnoho vnořených cyklů nebo složitých porovnání, z čehož vyplývá i časová úspora při běhu algoritmu.

Díky této bakalářské práci jsem nejen získal nové teoretické i praktické znalosti v oboru počítačové grafiky, ale také jsem si prohloubil znalosti jazyka C/C++. Navíc jsem se seznámil s knihovnami VectorEntity a OSG, které jsou beze sporu velmi dobře propracované a rozhodně se další práci s těmito knihovnami nebudu vyhýbat.

# Literatura


[1] P. C. Carvalho D. Martínez, L. Velho. *Geodesic Paths on Triangular Meshes.* IMPA, Rio de Janeiro, Brazil.

[2] J.-R. Sack M.Lanthier, A.Maheshwari. *Approximating Weighted Shortest Paths on Polyhedral Surface.* Sixth ACM symposium on Computational Geometry, 1998.

[3] Z. Novotný. *Sdílený virtuální svět, Diplomová práce.* Západočeská univerzita, Plzeň, 2006.

[4] A. Porazilová. *The Shortest Path.* 25. konference o geometrii a počítačové grafice.

[5] WWW stránky. Medical data segmentation toolkit. `http://www.fit.vutbr.cz/španel/mdstk/`.

[6] WWW stránky. Nejkratší cesta v grafu. `http://kam.mff.cuni.cz/k̃uba/ka/min_cesta.pdf`.

[7] WWW stránky. Openscenegraph website. `http://www.openscenegraph.org/projects/osg`.

[8] WWW stránky. Reboot - nejkratší cesta v ohodnoceném grafu. `http://reboot.cz/howto/programovani/nejkratsi-cesta-v-ohodnocenem-grafu/articles.html?id=223`.

[9] WWW stránky. Wikipedia - dijkstrův algoritmus. `http://cs.wikipedia.org/wiki/Dijkstr%C5%AFv_algoritmus`.

[10] H. Suzuki T. Kanai. *Approximate shortest path on polyhedral surface and its applications.* Computer-Aided Design, 2001.

[11] J. Zábranský. *Triangulace povrchů a úlohy na nich, Diplomová práce.* Západočeská univerzita, Plzeň, 2004.