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

DIPLOMOVÁ PRÁCE

Plzeň, 2008 Roman Soukal

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

Diplomová práce

Aplikace algoritmu procházky v počítačové grafice

Vedoucí práce
Doc. Dr. Ing. Ivana Kolingerová

Plzeň, 2008 Roman Soukal

Zadání

1. Prostudujte problematiku lokace algoritmem procházky v kontextu triangulací a pseudotriangulací v E2 a v E3.
2. V návaznosti na své výsledky oborového projektu porovnejte a zhodnoťte jednotlivé verze algoritmů procházky a navrhněte vhodné varianty pro různé úlohy počítačové grafiky.
3. Po dohodě s vedoucí práce vyberte konkrétní problém řešený na pracovišti a vyzkoušejte řešení založené na algoritmu procházky v kontextu daného problému.
4. Zhodnoťte dosažené výsledky.

Poděkování

Rád bych poděkoval všem, kdo se jakýmkoliv způsobem (někdy i nevědomky) podíleli na vzniku této práce. Děkuji Prof. Ing. Václavu Skalovi za inspiraci ohledně algoritmu přímé procházky. Dále Martinu Konkolovi za pomoc s 3D vizualizací. Děkuji Ing. Michalu Zemkovi, Bc. Tomáši Vomáčkovi a Bc. Václavu Purchartovi za praktickou aplikaci mých poznatků, za jejich analýzu a za další cenné připomínky a nápady. Velké poděkování patří i Ing. Janu Trčkovi za svolení k použití jeho diplomové práce včetně zdrojových souborů implementace a souhlasu k jejich případné úpravě. Samozřejmě největší poděkování patří vedoucí této diplomové práce Doc. Dr. Ing. Ivaně Kolingerové za způsob vedení nejen diplomové práce, ale např. i oborového projektu. Děkuji za spoustu zajímavých nápadů, myšlenek a inspirací, děkuji za cenné konzultace, děkuji za poskytnutá testovací data, triangualizátory a spousty dalších poskytnutých materiálů, děkuji také za neustálou ochotu a především pak trpělivost, které bylo bezpochyby potřeba hodně v určitých fázích mého navazujícího studia. Velké díky patří i těm, bez kterých bych nemohl nikdy studovat – mým rodičům. Díky!

Prohlášení

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

V Plzni dne

…............................... …..

Roman Soukal

Abstract

For triangulated data in 2D and 3D including pseudotriangulations in 2D, a location problem usually means how to find a triangle, tetrahedron or pseudotriangle which contains the given point. Many efficient algorithms and data structures exist which are suitable for planar or tetrahedral meshes, such as DAGs (directed acyclic graphs), buckets, quadtrees or octrees, skip lists, data structures based on random sampling, etc. These methods mostly work in $O(\log n)$ expected or worst-case time per one location query. Unfortunately, memory requirements of location data structures, although usually $O(n)$, may be for large data prohibitive. Therefore, time-suboptimal approaches utilizing walk algorithms ($O(n^{1/3})$ up to $O(n^{1/2})$), have been developed and are used because no extra location data structures are needed and so no extra memory is consumed. This is the main advantage of walking algorithms.

Obsah

ZADÁNÍ 1

PODĚKOVÁNÍ 2

PROHLÁŠENÍ 3

ABSTRACT 4

OBSAH 5

1. ÚVOD 8

2. ALGORITMUS PROCHÁZKY V E^2 9
   - 2.1. Úvod 9
   - 2.2. Znaménkový test 9
     - 2.2.1. Dosazení do implicitního předpisu přímky 9
     - 2.2.2. Test pomocí determinantu 10
     - 2.2.3. Upravený determinantový test 10
     - 2.2.4. Shrnutí znaménkových testů 10
   - 2.3. Remembering walk 11
     - 2.3.1. Zkrácené vyhodnocování 12
     - 2.3.2. Předvedení algoritmu na příkladu 13
   - 2.4. Remembering Stochastic walk 13
   - 2.5. Vylepšení (Fast walk) 15
     - 2.5.1. Problém algoritmu Fast walk 15
   - 2.6. Algoritmus první hrany vpravo 17
     - 2.6.1. Problém Algoritmu první hrany vpravo 18
     - 2.6.2. Shrnutí algoritmu 18
   - 2.7. Distance Fast walk 19
     - 2.7.1. Volba počtu iterací 20
   - 2.8. Flag Fast Walk 22
     - 2.8.1. Shrnutí algoritmu 23
   - 2.9. Straight walk 23
     - 2.9.1. Určení hledaného trojúhelníku 24
   - 2.10. Distance Straight walk 25
     - 2.10.1. Shrnutí algoritmu 27
   - 2.11. Normal Straight walk 27
     - 2.11.1. Shrnutí Normal Straight walk 29
   - 2.12. Orthogonal walk 29
     - 2.12.1. Detailní popis algoritmu 29
     - 2.12.2. Shrnutí Orthogonal walk 33
   - 2.13. Výběr počátečního trojúhelníku 34
   - 2.14. Problém procházkových algoritmů 35

2.15. Adaptivní robustní znaménkový test ... 36

3. PRAKTICKÉ TESTY PROCHÁZEK V E2 ... 39
3.1. Použité programové prostředky ... 39
3.1.1. Testovací data ... 39
3.1.2. Testovací aplikace ... 39
3.1.3. Použité datové struktury ... 39
3.1.4. Způsob testování ... 40
3.1.5. Řešení nestandardních situací ... 40
3.2. Java Native Interface a ShewLib ... 41
3.2.1. Java Native Interface (JNI) ... 41
3.2.2. Použití Shewchukovy knihovny pro znaménkový test ... 41
3.2.3. Použití JNI pro C/C++ funkce/metody ... 42
3.3. Výběr počátečního trojúhelníku ... 44
3.3.1. Volba vhodného počtu výběrů počátečního trojúhelníku ... 45
3.3.2. Dosažené urychlení ... 48
3.3.3. Vliv vstupních dat na výběr počátečního trojúhelníku ... 49
3.4. Vlastnosti jednotlivých algoritmů ... 50
3.4.1. Algoritmy procházek dle viditelnosti ... 50
3.4.2. Distance Fast walk ... 54
3.4.3. Procházky přímé ... 56
3.4.4. Pravoúhlá procházka ... 58
3.4.5. Porovnání jednotlivých algoritmů ... 60
3.5. Praktická aplikace ... 61
3.5.1. Dynamická Delaunayova triangulace pro kinetická data ... 61
3.5.2. Deformace terénu pro virtuální realitu ... 63

4. ALGORITMUS PROCHÁZKY V E3 ... 64
4.1. Úvod ... 64
4.2. Znaménkový test ... 64
4.3. Problém procházkových algoritmů ... 64
4.4. Procházky dle viditelnosti v E3 ... 65
4.5. Výběr počátečního tetrahedronu ... 65

5. PRAKTICKÉ TESTY PROCHÁZEK V E3 ... 67
5.1. Použité programové prostředky ... 67
5.1.1. JOGL ... 67
5.2. Výběr počátečního tetrahedronu ... 68
5.2.1. Volba vhodného počtu výběrů počátečního tetrahedronu ... 69
5.2.2. Dosažené urychlení ... 70
5.2.3. Vliv vstupních dat na výběr počátečního tetrahedronu ... 71
5.3. Vlastnosti jednotlivých algoritmů ... 71
5.3.1. Odhadovaná očekávaná časová složitost ... 71
5.3.2. Porovnání jednotlivých algoritmů ... 72
5.4. Praktická aplikace ... 73
5.4.1. Využití regulárních triangulací pro hledání tunelů v molekulách proteinů ... 73

6. PROCHÁZKY V PSEUDOTRIANGULACI ... 75
6.1. Základní pojmy a definice ... 75
6.2. Testy orientace ... 76

6.3. Jednoduchý algoritmus procházky 78
6.3.1. Problém Jednoduchého algoritmu procházky 79
6.3.2. Problém randomizované verze jednoduchého algoritmu 82
6.4. Algoritmus procházky verze 1 83
6.4.1. Shrnutí algoritmu 85
6.5. Algoritmus procházky verze 2 85
6.5.1. Shrnutí algoritmu 87
7. PRAKTICKÉ TESTY PROCHÁZEK V PSEUDOTRIANGULACI 88
7.1. Vstupní data 88
7.2. Způsob testování 88
7.3. Počet hranových testů 89
7.4. Délka procházek 90
7.5. Opakované návštěvy stěn 91
7.6. Rychlost procházkových algoritmů 91
8. ZÁVĚR 94
POUŽITÉ ZDROJE 95
PŘÍLOHY 98
A Testy 2D procházkových algoritmů 98
B Testy Výběru počátečního tetrahedronu 99
C Vzor formátu vstupních dat 100
C.1. 2D 100
C.2. 3D 101
D SVK 2008 – Rozšířený abstrakt 102

1. Úvod

Lokace bodu je velmi často řešenou úlohou v oblasti počítačové grafiky. Efektivní lokační algoritmy používají různorodé datové struktury (DAG[1], skip list[2], quadtree[3], datové struktury s náhodným vzorkováním a další). Všechny tyto struktury jsou velmi efektivní, a proto nám zmíněné algoritmy pracují s očekávanou časovou složitostí $O(\log n)$ na jeden lokalizovaný bod, kde n je celkový počet vrcholů struktury (triangulace, pseudotriangulace, tetrahedronová síť apod.). Spolu s rychlostí si však nesou i jistou spotřebu paměti. Ačkoliv je v mnoha případech spotřeba paměti lineární (vzhledem k počtu bodů), někdy i to může být výrazně limitujícím faktorem vzhledem k tomu, že vstupní data bývají obrovská (milióny bodů). A tak někteří programátoři sahají k jinému, paměťově úspornějšímu řešení – například k lokaci pomocí některého z algoritmů procházky.

Obecně pod pojem algoritmů procházky řadíme všechny algoritmy, které v triangulacích, Voronoiových diagramech, pseudotriangulacích či tetrahedronových sítích apod. lokalizují bod pomocí průchodu přes jednotlivé sousedící prvky této sítě. Zmíněnou lokací rozumíme určení prvku, v kterém se hledaný bod nachází. Ačkoliv je tento přístup méně efektivní (s očekávanou časovou složitostí $O(n^{1/3})$ až $O(n^{1/2})$ na jeden lokalizovaný bod), nepotřebuje žádné další datové struktury, a tedy nespotřebovává ani žádnou paměť navíc.

[De02, Ko06, Tr07a]

Práce je rozdělena na tři hlavní části, podle datové struktury na kterou je algoritmus procházky aplikován (triangulace, tetrahedronová síť, pseudotriangulace). Každá část je rozdělena na dvě kapitoly, kapitolu teoretickou a kapitolu praktickou (způsob implementace, testy, …).

Byla implementována řada algoritmů procházky pro rovinné triangulace včetně procházek dle [De02] a [Ko06], z nichž některé byly vlastní a některé byly výrazně upraveny pro zvýšení efektivity vyhledávání. Dále byly implementovány spolehlivé algoritmy procházky pro tetrahedronové sítě (algoritmy procházky dle viditelnosti, protože ostatní algoritmy se těžko vyrovnávají se spoustou možných singulárních případů typických pro E^3). Byly zkoumány i algoritmy procházky v pseudotriangulacích dle [Tr07a]. Všechny algoritmy byly důkladně testovány a vzájemně porovnány

[1] DAG – z anglického „Directed Acyclic Graph“ – orientovaný acyklický graf [Fl91].
[2] Skip list – pravděpodobnostní varianta k tradičním vyváženým stromům [Ep05].
[3] Quadtree – stromová struktura, kdy každý uzel má čtyři syny [Ep05].

2. Algoritmus procházky v E^2

2.1. Úvod

Strategie procházky v triangulaci znamená, že algoritmus zkoumá jeden trojúhelník za druhým, přičemž do dalšího trojúhelníku cestuje přes hranu aktuálně prozkoumávaného trojúhelníku a jako další trojúhelník zvolí ten, který vyhovuje nejlépe kritériu pro přiblížení se k hledanému bodu. Počáteční trojúhelník bývá volen náhodně nebo se například zvolí z množiny náhodně vybraných trojúhelníků ten, který je k hledanému bodu nejblíže. Procházka může vést všemi trojúhelníky, které protínají úsečku mezi vrcholem počátečního trojúhelníku a hledaným bodem, v takovém případě hovoříme o tzv. *přímé procházce*[4]. Nebo cesta může být rozdělena podél os, kdy se algoritmus hledanému bodu blíží nejprve po jedné ose a teprve potom po ose druhé. Tato procházka bývá nazývána *pravoúhlá procházka*[5]. Naproti tomu *procházka dle viditelnosti*[6] používá test orientace hledaného bodu vůči hranám trojúhelníku. Test se záporným výsledkem pak určí, přes kterou hranu algoritmus půjde do dalšího trojúhelníku.

[De02, Ko06, Tr07a]

2.2. Znaménkový test

Ve všech procházkových algoritmech jsou využívány tzv. znaménkové testy. Pod pojmem znaménkový test v E^2 rozumíme test polohy bodu vůči přímce, kdy přímka je dána dvěma body a její směrový vektor je orientován od prvního bodu k druhému. Zkoumaný bod se potom může nacházet napravo, nalevo nebo na přímce.

2.2.1. Dosazení do implicitního předpisu přímky

Nejjednodušší způsob znaménkového testu je dosazení bodu $\mathbf{B}[Bx \quad By]$ do implicitního předpisu přímky definovaného jako $\boldsymbol{p}\mathbf{:}\ f(x,y) = a \cdot x + b \cdot y + c$, přičemž přímka je zadána dvěma body **M**, **N** a orientována ve směru od *M* k *N*.

Implicitní předpis přímky získáme nejlépe jako vektorový součin $\mathbf{P_h} = \mathbf{M_h} \times \mathbf{N_h}$ bodů $\mathbf{M_h}$, $\mathbf{N_h}$ zapsaných v homogenních souřadnicích jako $\mathbf{M_h}[Mx \quad My \quad 1]$ a $\mathbf{N_h}[Nx \quad Ny \quad 1]$. Přitom $\mathbf{P_h} = [P_hx \quad P_hy \quad P_hz] = [a \quad b \quad c]$. Výsledná poloha bodu vůči přímce je pak určená ze znaménka funkční hodnoty $f(x,y) = a \cdot Bx + b \cdot By + c$. Pokud je funkční hodnota záporná, nachází se bod

[4] Přímá procházka – v anglické terminologii „Straight walk“.
[5] Pravoúhlá procházka – v anglické terminologii „Orthogonal walk“.
[6] Procházka dle viditelnosti – v anglické terminologii „Visibility walk“.

B nalevo od přímky, pokud je hodnota kladná, nachází se bod **B** napravo od přímky a v případě, že je hodnota nulová, nachází se bod přímo na přímce.

2.2.2. Test pomocí determinantu

Stejně jako v podkapitole 2.2.1 je i zde testována poloha bodu **B** vůči přímce zadané dvěma body **M**, **N** a orientované ve směru od **M** k **N**. Výslednou orientaci potom určíme výpočtem funkční hodnoty funkce

$$f(\mathbf{M},\mathbf{N},\mathbf{B}) = \det\begin{vmatrix} \mathbf{M_h} \\ \mathbf{N_h} \\ \mathbf{B_h} \end{vmatrix} = \det\begin{vmatrix} Mx & My & 1 \\ Nx & Ny & 1 \\ Bx & By & 1 \end{vmatrix}$$ [7].

Výsledná poloha bodu vůči přímce je pak určena ze znaménka této funkční hodnoty. Pokud je funkční hodnota záporná, nachází se bod **B** nalevo od přímky, pokud je hodnota kladná, nachází se bod **B** napravo od přímky a v případě, že je hodnota nulová, nachází se bod přímo na přímce.

2.2.3. Upravený determinantový test

Standardním metodou výpočtu determinantů čtvercové matice řádu 3 je tzv. *Sarrusovo pravidlo* [Te05]. Na výpočet zmíněného determinantu pomocí *Sarusova pravidla* je pak potřeba celkově 5 operací sčítání/odčítání a 6 operací násobení. Test pomocí determinantu lze však snadno upravit na tvar z [De02]

$$f(\mathbf{M},\mathbf{N},\mathbf{B}) = \det\begin{vmatrix} Nx - Mx & Bx - Mx \\ Ny - My & By - My \end{vmatrix}.$$

Nyní je na první pohled patrné, že k výpočtu determinantu potřebujeme celkově 5 operací sčítání/odčítání a 2 operace násobení. Tedy oproti *Sarusovu pravidlu* ušetříme celkově 4 operace násobení. Navíc se tento test později ukázal jako numericky stabilnější.

2.2.4. Shrnutí znaménkových testů

V algoritmech popsaných dále jsou využívány dva způsoby výpočtu znaménkového testu. Vzhledem k tomu, že je znaménkový test provedený pomocí *determinantového testu* z podkapitoly 2.2.3 nejrychlejší, je tento znaménkový test upřednostňován. Test *dosazením do implicitního předpisu přímky* je používán v místech, kde se opakovaně testuje poloha bodů vůči stejné přímce (např. algoritmy *přímé procházky*), protože parametry přímky se spočtou jednorázově a pak už jsou jen dosazovány souřadnice jednotlivých testovaných bodů.

---

[7] Stejný test se používá pro test pravotočivosti, respektive levotočivosti. [Ko05]

2.3. Remembering walk

Algoritmus se řadí do *procházek dle viditelnosti*. Vychází jak ze znalosti vrcholů jednotlivých trojúhelníků, tak i ze znalosti sousedů jednotlivých trojúhelníků triangulace. Sousední trojúhelníky i body jsou vždy orientovány v jednom směru (v našem případě vždy proti směru hodinových ručiček – to je předpokládáno i v dalším textu) a vždy lze i říci, přes kterou hranu (danou dvěma body) se dostaneme do jakého trojúhelníku.

Algoritmus postupuje tak, že začne v libovolném trojúhelníku triangulace. Dále postupně prochází všechny hrany, dokud nenajde takovou hranu, od níž je hledaný bod napravo (zjistíme pomocí libovolného znaménkového testu z 2.2). V případě, že jsme takovouto hranu našli, přejdeme přes tuto hranu do sousedního trojúhelníku a postup opakujeme. Pokud už nenajdeme žádnou hranu, od které se nachází bod napravo, algoritmus končí a víme, že bod se nachází v právě kontrolovaném trojúhelníku.

Abychom si ušetřili test u hrany, přes kterou jsme do aktuálního trojúhelníku přešli, uchováváme si informaci o předchozím trojúhelníku (proto *remembering* – pamatující) a nejdříve zkoušíme zjistit, jestli právě zkoumaná hrana nevede do tohoto předchozího trojúhelníku. Takovéto porovnání je mnohem rychlejší a ve výsledku se nám algoritmus znatelně zrychlí (provádíme v průměru o 1/3 testů méně).

```
trojuhelnik rememberingWalk(bod cil)
{
 trojuhelnik aktTroj, predchozi, soused = null;
 hrana hrana = null;
 boolean found = false;

 aktTroj = libovolný trojúhelník z triangulace;8
 predchozi = aktTroj;

 while (not found)
 {
  for (int i = 1; i < 4; i++)
  {
   hrana = i-tá hrana z aktTroj;9
   soused = soused přes hranu hrana;
   if (soused!=predchozi and signum(znamenkovyTest(hrana, cil))==-1)
   {
    predchozi = aktTroj;
    aktTroj = soused;
    break;
   }
   if (i==3)
    found = true;
  }
 }
```

[8] Pro výběr prvního trojúhelníku lze využít vylepšení [2.13].
[9] *i*-tá hrana trojúhelníku zadaná *i*-tým a (*i*+1)-tým bodem trojúhelníku, kdy 4. bod trojúhelníku je totožný s 1. bodem, 5. bod s 2. bodem atp.

```
 //nyní se výsledný trojúhelník nachází v proměnné aktTroj
 return aktTroj;
}
```

Algoritmus 2.1 – *Remembering walk*

2.3.1. Zkrácené vyhodnocování

Ukončení procházky v případě nalezení správného trojúhelníku zařizuje podmínka na konci *for* cyklu, která nastaví logickou proměnou *found* na false v případě, kdy je hodnota řídící proměnné cyklu *i* rovna hodnotě 3 (již jsme otestovali i třetí (poslední) hranu v aktuálním trojúhelníku, ale trojúhelník jsme ani přesto neopustili). Nevýhoda tohoto způsobu ukončení spočívá v tom, že je podmínka mnohokrát testována zbytečně.

Logický výraz obsahující logické součty nebo součiny lze vyhodnocovat buď zcela nebo zkráceně (*short circuit*) nebo oběma způsoby. *Zkrácené vyhodnocování* znamená, že operandy výrazu jsou vyhodnocovány zleva doprava a jakmile je možno určit konečný výsledek, vyhodnocování okamžitě končí. To má význam především kvůli urychlení výpočtu, když v logickém součinu je první hodnota *false* nebo když v logickém součtu je první hodnota *true*. V obou zmíněných případech je ukončeno vyhodnocování už na první hodnotě. [He03a]

Následující upravená část vnitřního cyklu *for* původního algoritmu (*Alg. 2.1*) ukazuje, jak efektivně vyřešit problém ukončení algoritmu v jazycích umožňujících *zkrácené vyhodnocování*. Vhledem k tomu, že tento postup je závislý na konkrétním programovacím jazyce, v popisu následujících algoritmů použit není, i když by jeho použití bylo efektivnější.

```
for (int i = 1; i < 4 or not(found = true); i++)
{
 hrana = i-tá hrana z aktTroj;
 soused = soused přes hranu hrana;
 if(soused!=predchozi and signum(znamenkovyTest(hrana, cil))==-1)
 {
  predchozi = aktTroj;
  aktTroj = soused;
  break;
 }
}
```

Algoritmus 2.2 – *Zkrácené vyhodnocování*

Je potřeba zdůraznit, že relačním operátorem *or* v hlavičce cyklu *for* v pseudokódu výše (*Alg. 2.2*) se myslí relační operátor *or* se *zkráceným vyhodnocováním*. Je tedy zřejmé, že přiřazení[10] hodnoty *true* do proměnné *false* proběhne až v případě, kdy řídící proměnná cyklu *i* obsahuje hodnotu 4. Negace před tímto subvýrazem pak umožní ukončení cyklu.

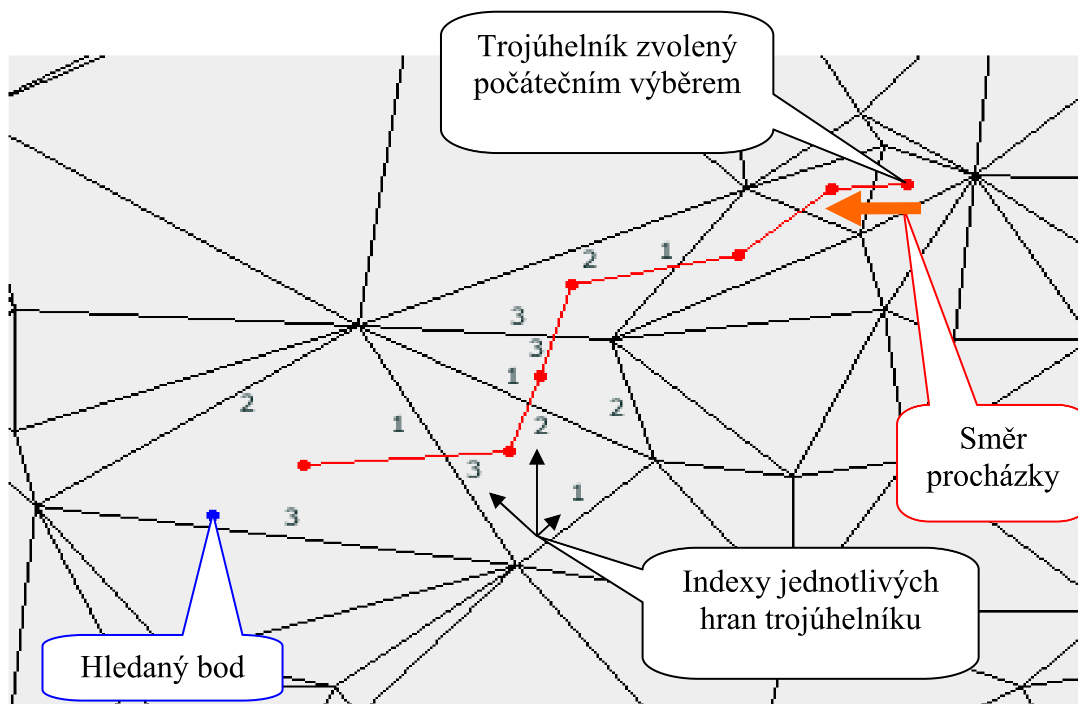
---

[10] Pozor, jde o jednoduché =, tudíž se nejedná o operátor porovnání, nýbrž jde o přiřazení.

2.3.2. Předvedení algoritmu na příkladu

Na následujícím obrázku (*Obr. 2.1*) je nastíněna funkčnost algoritmu. Nesmíme zapomínat, že hrany (respektive body) jsou v trojúhelníku uspořádány v určitém pořadí (po směru nebo proti směru hodinových ručiček), ale první hrana (podle indexace) může být jakákoliv hrana z trojúhelníku (závisí na způsobu tvorby triangulace). Samozřejmě vzhledem k jednotné orientaci jsou následující hrany již dané .

Procházka na ilustrovaném příkladu (*Obr. 2.1*) začíná vpravo nahoře a její směr je označen šipkou. Některé procházené trojúhelníky pak mají znázorněné i indexy svých hran.



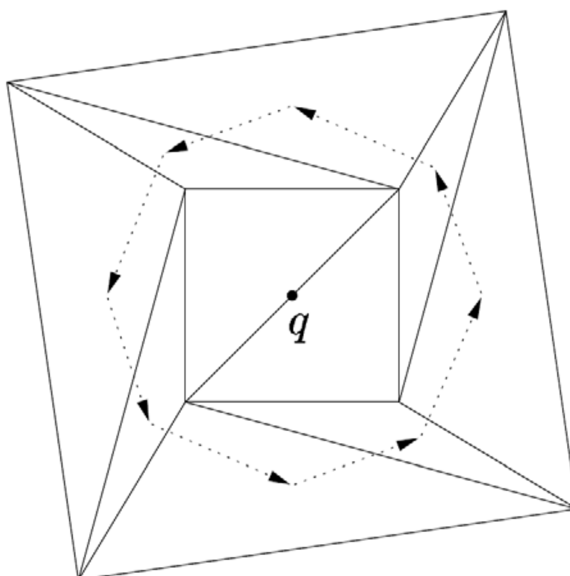
Obrázek 2.1 – Ukázka *Remembering walk*

2.4. Remembering Stochastic walk

Lokace bodu pomocí *Remembering walk* může zacyklit (s výjimkou Delaunayových triangulací – plyne z faktu, že na nich existuje částečné uspořádání stěn (viz [Fl91])). Na následujícím obrázku (*Obr 2.2*) je zobrazen případ zacyklení lokace na triangulaci jiné než Delaunayově. Pro obecné použití *Remembering walk* algoritmu je tudíž nutné implementovat jeho randomizovanou verzi. Randomizace je použita na určování pořadí výběru jednotlivých kontrolovaných hran, to zajistí, že se algoritmus již nezacyklí.

Je pochopitelné, že režie potřebná pro vygenerování náhodného výběru hran není zanedbatelná, proto byl kladen důraz na nalezení pokud možno

nejrychlejšího způsobu. Nejlepší se pak jevilo generování náhodných logických hodnot (true/false) a proto byla randomizace založená právě na tomto zjištění.



Obrázek 2.2 – Zacyklení *Remembering walk* [We98]

Vzhledem k tomu, že hrana, která náleží trojúhelníku, z nějž jsme do aktuálního trojúhelníku přišli, je ignorována, rozhodujeme pouze o pořadí zpracování dvou zbylých hran. Víme ale zároveň, že index ignorované hrany (hrany k předchozímu trojúhelníku) není předem znám, tzn. předem nevíme, mezi kterými dvěma hranami se máme rozhodovat. Pokud ale výběr hran v cyklu budeme provádět zpětným průchodem od poslední hrany, dostaneme pořadí dvou směrodatných hran pokaždé přesně opačné než pořadí při dopředném průchodu od první hrany.

Pokud tuto myšlenku zkombinujeme s faktem, že nejrychlejší je náhodné generování logických hodnot, postačí k požadované náhodnosti pouze volba způsobu průchodu hran v cyklu (dopředný průchod od první hrany nebo zpětný průchod od poslední hrany) na základě vygenerované logické hodnoty.

Následující pseudokód (*Alg 2.3*) naznačuje, jakým způsobem je nutné modifikovat část původního algoritmu *Remembering walk*, abychom prováděli procházku pomocí algoritmu *Remembering Stochastic walk*. Tečky značí kód původního *Remembering walk* algoritmu. Proměnná *poc* označuje první kontrolovanou hranu v trojúhelníku a proměnná *inc* je hodnota inkrementace. Obě proměnné jsou pak nastaveny pro každý kontrolovaný trojúhelník zvlášť v závislosti na vygenerované logické hodnotě. Na ukázce je pouze část *Alg 2.1*, kterou je potřeba upravit.

```
...

while (not found)
{
 if (nextBoolean())
```

```
{
 poc = 1;
 inc = 1;
}
else
{
 poc = 3;
 inc = -1;
}

for (int i = poc; i < 4 and i > 0; i++)
{
 hrana = i-tá hrana z aktTroj;

...
```

Algoritmus 2.3 – Úprava *Remembering walk* na *Remembering Stochastic walk*

2.5. Vylepšení (Fast walk)

V původním *Remembering walk* algoritmu se provádí vždy 1 až 2 hranové testy. V průměru tedy provádíme 1½ hranových testů. Následující algoritmus dle [Ko06] vychází z předpokladu, že si vystačíme vždy s jedním hranovým testem. Algoritmus pracuje tak, že provede první možný hranový test a pokud je hledaný bod napravo, přejde do trojúhelníku po této hraně, pokud ne, přejde do trojúhelníku po následující hraně (myšleno po následující hraně, po které by nepřešel do předchozího trojúhelníku). Lze tudíž říci, že vždy aplikujeme právě jeden hranový test, tudíž ušetříme v průměru ½ hranových testů.

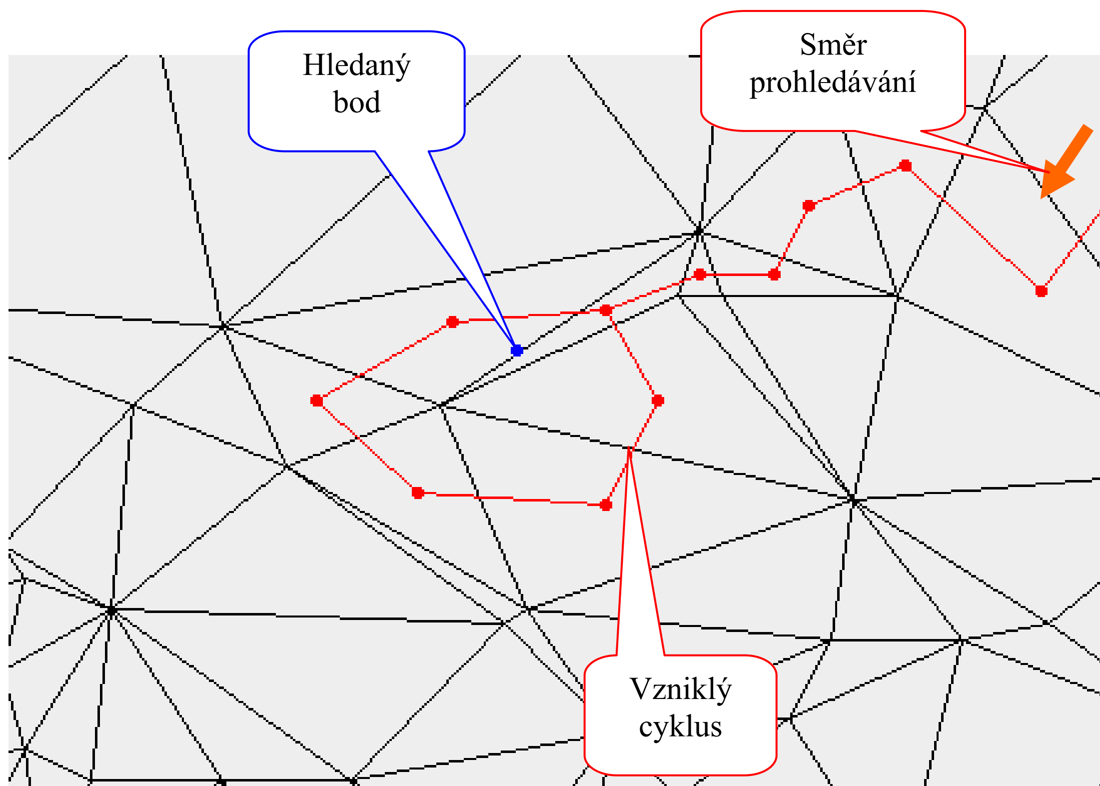
2.5.1. Problém algoritmu Fast walk

Hledání cesty tímto algoritmem funguje naprosto srovnatelně s *Remembering walk* algoritmem s tím rozdílem, že nejsme schopní určit, kdy už byl nalezen trojúhelník obsahující námi hledaný bod. Zde nastává problém, protože i když nalezneme trojúhelník obsahující námi hledaný bod, přejdeme ho a pokračujeme dál ve vyhledávání. Zmíněný problém je ilustrován na následujícím obrázku (*Obr. 2.3*). Procházka na ilustrovaném příkladu začíná vpravo nahoře a její směr je označen šipkou. Zároveň je z obrázku patrné i to, že nejsme schopni určit, kdy již algoritmus dosáhl hledaného trojúhelníku (viz vzniklý cyklus).

Proto stojíme před otázkou, jak zmíněný problém efektivně vyřešit. Dosavadní algoritmus (dle [Ko06]) pracoval tak, že určitý počet kroků procházky provedl algoritmem *Fast walk* a pak přešel na algoritmus *Remembering Stochastic walk*. Pokusy bylo zjištěno, že nejlepší se jeví použití *Fast walk* v právě $1{,}15 \cdot \sqrt[3]{n}$ [11] krocích (dle [Ko06]). U tohoto způsobu je udávána úspora v průměru okolo 8% hranových testů, což je ale stále hodně

[11] n – počet bodů triangulace

vzdálené potenciálnímu ušetření až 50% hranových testů. Algoritmus *Fast walk* dle [Ko06] je zachycen v následujícím pseudokódu (*Alg. 2.4*).



Obrázek 2.3 – Zachycení problému algoritmu *Fast walk*

```
trojuhelnik fastWalk(bod cil)
{
 trojuhelnik aktTroj, predchozi, soused = null;
 hrana hrana = null;
 int pocetKroku = 1,15.∛n ;¹²

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 predchozi = aktTroj;

 for (int i = 1; i < pocetKroku; i++)
 {
  hrana = náhodná hrana z aktTroj, která nevede do predchozi;
  soused = soused přes hranu hrana;
  if (signum(znamenkovyTest(hrana, cil))==-1)
  {
   predchozi = aktTroj;
   aktTroj = soused;
  }
  else
  {
   hrana = následující hrana z aktTroj, která nevede do predchozi;
   soused = soused přes hranu hrana;
   predchozi = aktTroj;
   aktTroj = soused;
  }
```

[12] n je celočíselná konstanta udávající počet vrcholů triangulace.

```
  }

 return remembering stochastic walk(cil, aktTroj);[13]
}
```

Algoritmus 2.4 – Algoritmus *Fast walk* dle [Ko06]

2.6. Algoritmus první hrany vpravo

Definujeme celočíselnou pomocnou proměnou (nazveme *p*), kterou na začátku nastavíme na nulu. Také definujeme celočíselnou konstantní proměnou maximálního počtu iterací (nazveme *K*), jejíž využití popisujeme dále. Tuto konstantu nastavíme na vhodnou hodnotu (vyplyne z dalšího textu).

První myšlenka na vylepšení *Fast walk* algoritmu spočívá v tom, že nejdříve nalezneme hranu, kterou jsme do trojúhelníku přišli. A hranový test provádíme hned na následující hraně v pořadí (tzn. při orientaci proti směru hodinových ručiček vždy hrana napravo od vstupní hrany). Pokud se hledaný bod nachází napravo od této hrany, přejdeme přes tuto hranu do příslušného trojúhelníku a vynulujeme pomocnou proměnou *p*. Pokud ale při hranovém testu zjistíme, že bod je nalevo od této hrany, potom stejně jako ve *Fast walk* přejdeme do trojúhelníku přiléhajícího následující hraně a navíc inkrementujeme pomocnou proměnou *p*. Při zpracování každého trojúhelníku pak kontrolujeme, zda není $p >= K$, pokud ano, končíme takto upravený *Fast walk* algoritmus a přejdeme na standardní *Remembering walk*.[14]

Pointa myšlenky tohoto algoritmu spočívá v tom, že pokud *Fast walk* mine ten správný trojúhelník, začne se točit v cyklu kolem jednoho z bodů cílového trojúhelníku a takto modifikovaný *Fast walk* bude využívat vždy až druhou potencionální hranu k přechodu do dalšího trojúhelníku. Při každém přechodu přes druhou hranu se tesy bude inkrementovat pomocná proměnná *p*. Až hodnota v *p* přeroste přes určitou mez *K*, najde se výsledný trojúhelník standardním algoritmem *Remembering walk*. Algoritmus první hrany vpravo je zachycen v následujícím pseudokódu (*Alg. 2.5*).

```
trojuhelnik prvniVpravo(bod cil, int K[15])
{
 trojuhelnik aktTroj, predchozi, soused = null;
 hrana hrana = null;
 int p = 0;

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 predchozi = aktTroj;
```

[13] Standardní algoritmus Remembering stochastic walk (dle *Alg 2.3*) s tím rozdílem, že jako počáteční trojúhelník procházky je volen trojúhelník předaný jako parametr (*aktTroj*).

[14] Samozřejmě se tímto algoritmem vzdáváme náhodnosti, tudíž je potenciální funkčnost omezená na Delaunayovy triangulace. Je totiž patrné, že na Obr. 2.2 by tento algoritmus nefungoval.

[15] *P*ředem definovaná hodnota krokové konstanty *K*

```
 while(p<K);
 {
  hrana = 1. hrana aktTroj napravo od hrany do predchozi;
  soused = soused přes hranu hrana;
  if (signum(znamenkovyTest(hrana, cil))==-1)
  {
   p = 0;
   predchozi = aktTroj;
   aktTroj = soused;
  }
  else
  {
   p++;
   hrana = následující hrana z aktTroj, která nevede do predchozi;
   soused = soused přes hranu hrana;
   predchozi = aktTroj;
   aktTroj = soused;
  }
 }

 return remembering walk(cil, aktTroj); [16]
}
```

Algoritmus 2.5 – *Algoritmus první hrany vpravo*

2.6.1. Problém Algoritmu první hrany vpravo

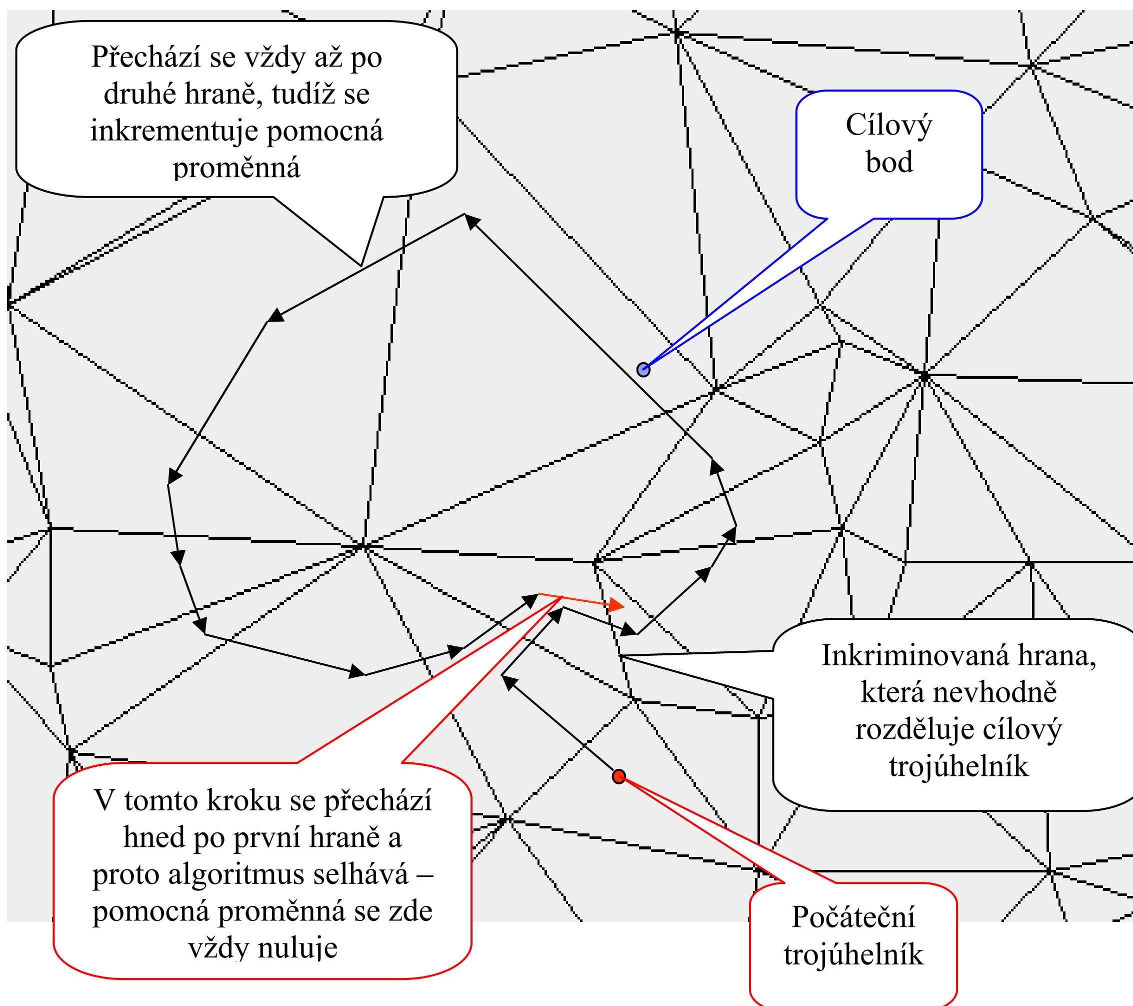
Na první pohled jednoduché a elegantní řešení, jenže bohužel existují případy, pro které toto řešení selhává. Jde o takové případy, kdy přímka procházející hranou sousedního trojúhelníku (tj. trojúhelníku, který má s cílovým trojúhelníkem společný bod a kterým algoritmus v cyklu prochází) dělí cílový trojúhelník na dvě části a cílový bod je v té části napravo od přímky (uvažujeme-li orientaci přímky totožnou s hranou). Nejlépe je takováto konfigurace patrná z obrázkového příkladu (*Obr. 2.4*). Na ilustrovaném příkladu začíná procházka dole uprostřed a její směr je patrný ze šipek.

2.6.2. Shrnutí algoritmu

Je patrné, že algoritmus nepracuje v některých obecných konfiguracích Delaunayovy triangulace korektně (vhledem k nechtěnému nulování pomocné proměnné se zacyklí). Ale celkově vzato je škoda úplně algoritmus zatracovat, protože je zde možnost, že algoritmus může fungovat v některých jiných typech triangulacích (jiných než je obecná Delaunayova triangulace).

Pod jinými typy triangulací si můžeme představit například Delaunayovy triangulace, které vyhovují určitým omezujícím podmínkám, ale i jiné typy triangulací, které nesplňují podmínky Delaunayovy triangulace. Například lze s jistotou prohlásit, že algoritmus bude fungovat pro všechny triangulace, které budou obsahovat pouze trojúhelníky s ostrými úhly (možný je i jeden pravý úhel) ve všech vrcholech.

[16] Standardní algoritmus Remembering walk (dle *Alg 2.1*) s tím rozdílem, že jako počáteční trojúhelník procházky je volen trojúhelník předaný jako parametr (*aktTroj*).



Obrázek 2.4 – Problém *Algoritmu první hrany vpravo*

2.7. Distance Fast walk

Algoritmus je založen na algoritmu *Fast walk*. Rozdíl oproti předchozím algoritmům je ten, že se snažíme odhadnout nalezení cílového bodu pomocí vzdálenostních testů. Průběžně je kontrolována vzdálenost aktuálního trojúhelníku od cílového bodu. Dokud se tato vzdálenost zmenšuje, stále pokračujeme ve vyhledávání. Pokud ale vzdálenost naroste (již jsme pravděpodobně bod přešli), přecházíme na standardní *Remembering (Stochastic) walk*, kterým již přesně dohledáme cílový trojúhelník.

Definujeme celočíselnou konstantní proměnou *POCET_ITERACI* a dále definujeme reálnou proměnou *staraVzdalenost*, které na začátku nastavíme maximální reálnou hodnotu. Potom vylepšení stávajícího algoritmu *Fast walk* funguje tak, že tělo algoritmu *Fast walk* proběhne právě v *POCET_ITERACI* iteracích. Následně se spočte kvadrát[17] vzdálenosti aktuálního trojúhelníku od cílového bodu[18]. Tato vzdálenost se uloží do reálné proměnné *novaVzdalenost*.

[17] Nezajímá nás přesná hodnota, ale pouze vzdálenosti pro porovnání mezi sebou, proto postačí kvadrát.
[18] Pro zjednodušení se počítá vzdálenost libovolného vrcholu aktuálního trojúhelníku od cílového bodu.

Pokud je nyní hodnota *novaVzdalenost* menší než *staraVzdalenost* (což v podstatě znamená, že jsme se přiblížili), přiřadíme do *staraVzdalenost* hodnotu z *novaVzdalenost* a postup opakujeme. Pokud tomu tak není (už se k cíli nepřibližujeme), končíme zmíněný upravený *Fast Walk* a dále pokračujeme algoritmem *Remembering (stochastic)*[19] *walk.* Následujícím pseudokódem (*Alg. 2.6*) je popsána obecně použitelná randomizovaná verze algoritmu *Distance Fast walk*.

```
trojuhelnik distanceFastWalk(bod cil, int POCET_ITERACI20)
{
 double novaVzdalenost, staraVzdalenost;
 trojuhelnik aktTroj, predchozi, soused = null;
 hrana hrana = null;

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 predchozi = aktTroj;
 novaVzdalenost = Double.MAX_VALUE;

 do
 {
  staraVzdalenost = novaVzdalenost;
  for (int i=0; i<POCET_ITERACI; i++)
  {
   hrana = náhodná hrana z aktTroj, která nevede do predchozi;
   soused = soused přes hranu hrana;
   if (signum(znamenkovyTest(hrana, cil))==-1)
   {
    predchozi = aktTroj;
    aktTroj = soused;
   }
   else
   {
    hrana = následující hrana z aktTroj, která nevede do predchozi;
    soused = soused přes hranu hrana;
    predchozi = aktTroj;
    aktTroj = soused;
   }
  }
  novaVzdalenost = kvadrát vzdálenosti cil od lib. vrcholu aktTroj;
 } while(novaVzdalenost<staraVzdalenost);

 return remembering stochastic walk(cil, aktTroj);
}
```

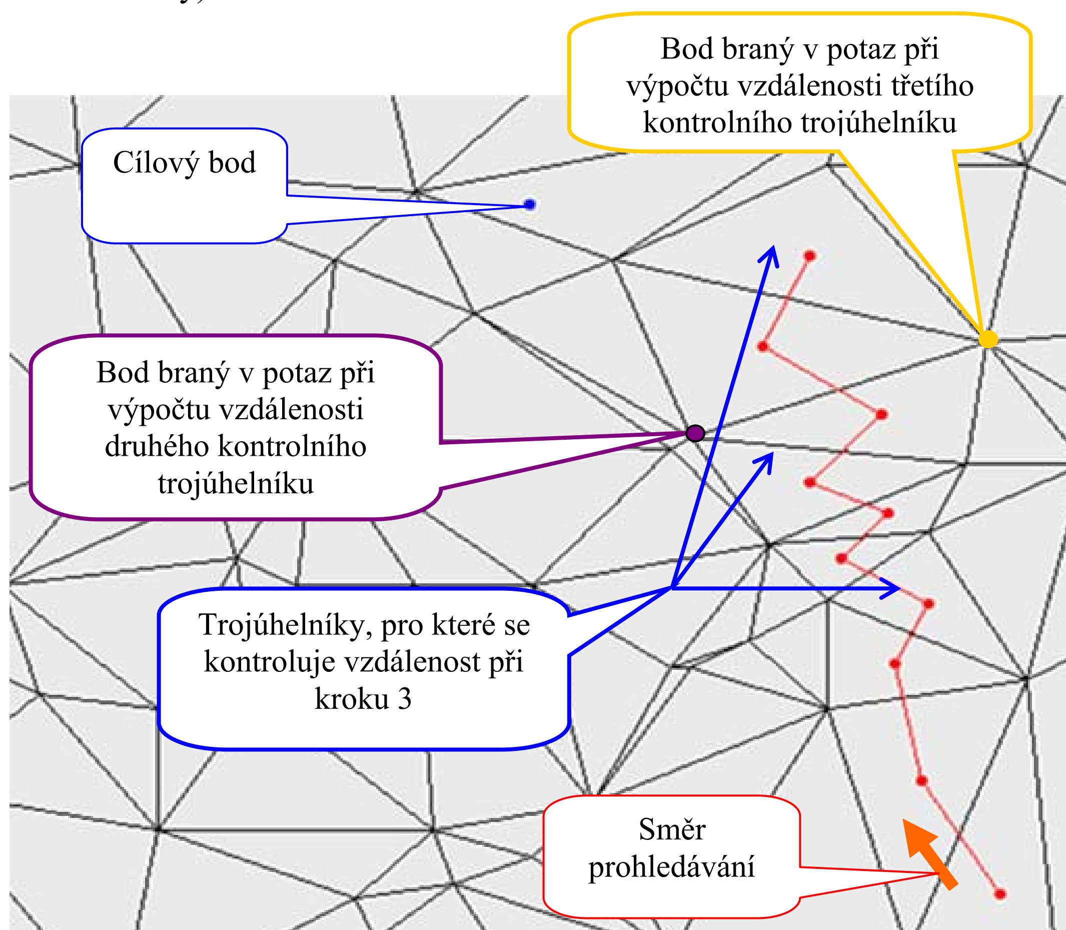
Algoritmus 2.6 – Algoritmus *Distance Fast walk*

2.7.1. Volba počtu iterací

Velmi důležitým aspektem tohoto algoritmu je volba počtu iterací (*POCET_ITERACI*), po kterém je kontrolována vzdálenost. Pokud je *POCET_ITERACI* číslo příliš malé, může se stát (a stane se v praxi poměrně často), že algoritmus skončí hodně brzy, protože přece jen je vzdálenost

[19] Samozřejmě v případě, kdy víme, že budeme vyhledávat v Delaunayově triangulaci, je rychlejší použít nerandomizovanou verzi [Fl91].
[20] Hodnota zvolená vhodně vzhledem k objemu dat a typu triangulace.

počítána od jakéhokoliv bodu aktuálního trojúhelníku, a tudíž je možné, že i u bližšího trojúhelníku (z hlediska algoritmu procházky) bude vypočtena vyšší vzdálenost. Samozřejmě čím vyšší je hodnota *POCET_ITERACI*, tím je riziko takovéto situace nižší. Zmíněný problém je patrný z následujícího obrázku (*Obr. 2.5*). Procházka na ilustrovaném příkladu začíná vlevo dole a její směr je označen šipkou. Dále jsou na obrázku vyznačeny trojúhelníky, pro které je kontrolována vzdálenost – kontrolní trojúhelníky (u některých troj. vyznačeny i kontrolní body).



Obrázek 2.5 – Předčasné ukončení algoritmu kvůli malému kroku

Na druhou stranu, pokud je *POCET_ITERACI* číslo příliš vysoké, může algoritmus provést zbytečně mnoho kroků navíc. Proto je důležité zvolit jakýsi kompromis, a to i s ohledem na celkový objem vstupních dat. Pokud víme, že procházka bude pravděpodobně dlouhá (velký objem dat), nevadí nám „několik kroků navíc“, ale předčasné ukončení této dlouhé procházky (především na počátku procházky) vinou malého kroku (*POCET_ITERACI*) je pro nás nežádoucí.

Naopak při malém objemu vstupních dat, kdy se dá procházka očekávat relativně krátká, je pro nás méně žádoucí, když je algoritmus *Fast walk* použit ve zbytečně mnoha krocích navíc, než když algoritmus degraduje na *Remembering Stochastic walk*.

2.8. Flag Fast Walk

Stejně jako předchozí algoritmus využívá i tento základní myšlenku *Fast walk*. Při procházení pomocí *Fast walk* je každému trojúhelníku nastaven příznak, který říká, že trojúhelník byl již navštíven. Ještě před nastavením samotného příznaku je každý trojúhelník nejdříve zkontrolován, jestli už příslušný příznak nastavený není. V případně, že je nastaven[21], procházka pomocí *Fast walk* končí a lokace pokračuje dále pomocí *Remembering (Stochastic) walk*, kterým je cílový trojúhelník bezpečně nalezen.

Důležitý je datový typ i hodnota samotného příznaku. Vzhledem k tomu, že je potřeba ve stejné triangulaci lokalizovat body opakovaně, je pro nastavování příznaku nutné zvolit datový typ s dostatečným rozsahem. Pro každou lokaci bodu je poté nutné zvolit takovou hodnotu příznaku, která ještě nebyla dříve použita, abychom měli jistotu, že nenarazíme na trojúhelník, který již byl označen v jiném průchodu. Pochopitelně pokud zvolíme pro příznak datový typ s malým rozsahem, po vyčerpání všech hodnot v rozsahu tohoto datového typu jsme nuceni nastavit všem trojúhelníkům příznak opět na inicializační hodnotu. Následující pseudokód (*Alg. 2.7*) obecně použitelnou randomizovanou verzi algoritmu *Flag Fast walk*.

```
trojuhelnik flagFastWalk(bod cil)
{
 int priznak;
 trojuhelnik aktTroj, predchozi, soused = null;
 hrana hrana = null;

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 predchozi = aktTroj;
 priznak = poslední hodnota příznaku + 1;

 while(true)
 {
  if (aktTroj.priznak==priznak)
   break;
  else
   aktTroj.priznak = priznak;

  hrana = náhodná hrana z aktTroj, která nevede do predchozi;
  soused = soused přes hranu hrana;
  if (signum(znamenkovyTest(hrana, cil))==-1)
  {
   predchozi = aktTroj;
   aktTroj = soused;
  }
  else
  {
   hrana = následující hrana z aktTroj, která nevede do predchozi;
   soused = soused přes hranu hrana;
   predchozi = aktTroj;
   aktTroj = soused;
  }
```

[21] To znamená, že byl již tento trojúhelník navštíven a procházka nyní běží v cyklu (Obr. 2.3).

```
  }
 return remembering stochastic walk(cil, aktTroj);
}
```

Algoritmus 2.7 – Algoritmus *Flag Fast walk*

2.8.1. Shrnutí algoritmu

Jednoduché a účinné řešení, které s sebou ale nese řadu nevýhod. První nevýhodou je rozšíření datové struktury trojúhelníku o příznak. Tím se zvýší paměťové nároky, což může mít při velkém objemu vstupních dat i rozhodující vliv na použitelnost algoritmu. Dost často je pak také potřeba implementovat procházkový algoritmus do již fungujícího programu, kde již není možnost stávající datové struktury modifikovat .

S problémem se setkáváme také v případě paralelního využití lokace bodu. Pokud bychom strukturu neměnili, mohli bychom paralelně lokalizovat jednotlivé body nad společnou datovou strukturou, což ale v případě použití *Flag Fast walk* nepřipadá v úvahu, protože by mohla nastat situace, kdy by dva procesy nezávisle na sobě měnily stejné části společné datové struktury a to by mohlo vyhledávání výrazně zpomalit nebo dokonce zacyklit.

Nevýhodou také zůstává omezený rozsah příznaku. Ať už zvolíme pro příznak jakýkoliv datový typ, může se stát, že vyčerpáme jeho rozsah a budeme nuceni nastavit všem trojúhelníkům příznak opět na inicializační hodnotu.

S ohledem na všechny zmíněné nevýhody algoritmu *Flag Fast walk* je třeba případné jeho použití důkladně zvážit. Jestliže není pro plánované využití algoritmu žádná ze zmíněných nevýhod podstatná, je tento algoritmus jednoduchá a spolehlivá volba.

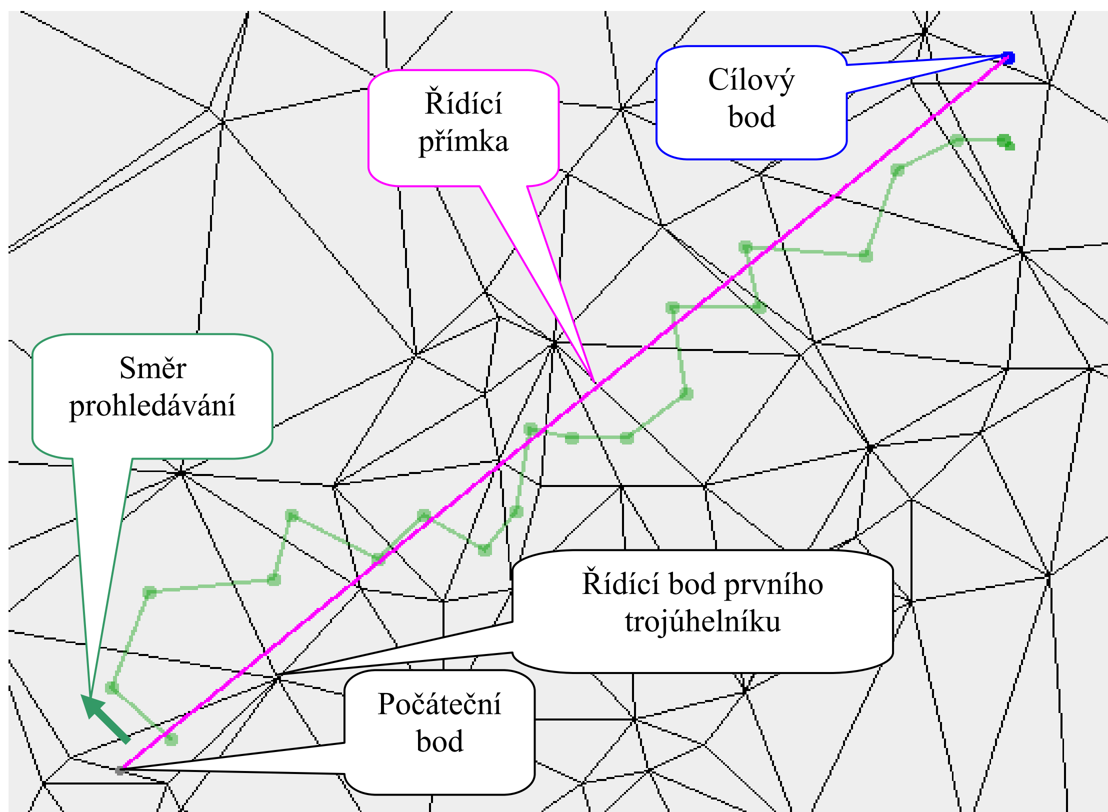
2.9. Straight walk

Algoritmus *Straight walk* neboli přímá procházka existuje v mnoha variantách [De02, Bh01, Zh03]. V následujícím textu budeme pod pojmem *Straight walk* mluvit o jednom konkrétním algoritmu přímé procházky popsaném dále.

Nejdříve zvolíme počáteční trojúhelník a počáteční bod procházky[22]. Vytvoříme řídící přímku (přímka procházející počátečním a cílovým bodem). Zvolíme počáteční hranu jako jednu z hran počátečního trojúhelníku[23]. V samotném algoritmu dále postupujeme cyklicky, kdy po přechodu do trojúhelníku dosadíme souřadnice řídícího bodu[24] trojúhelníku do implicitního předpisu přímky (2.2.1). Pokud je výsledek kladný, přecházíme do následujícího trojúhelníku po hraně nalevo od řídícího bodu. Pokud je výsledek

[22] Počáteční bod může být bod kdekoliv uvnitř počátečního trojúhelníku.
[23] Volíme hranu, která je naproti tomu vrcholu trojúhelníku, který je nejblíže cílovému bodu..
[24] Řídící bod trojúhelníku je vrchol naproti hraně, po které jsme do trojúhelníku přešli, v případě počátečního trojúhelníku je řídící bod naproti počáteční hraně.

záporný, přecházíme po hraně napravo od řídícího bodu. Takto postupně projdeme až k hledanému trojúhelníku obsahujícímu cílový bod. Algoritmus je nastíněn na následujícím obrázku (*Obr. 2.6*). Na ilustrovaném příkladu začíná procházka vlevo dole.



Obrázek 2.6 – Lokace bodu pomocí algoritmu *Straight walk*

2.9.1. Určení hledaného trojúhelníku

Při implementaci tohoto algoritmu narazíme na otázku, kterou je potřeba vyřešit. Jak zjistit, že se již nacházíme v cílovém trojúhelníku? Metody, které indikují dosažení cíle přímo v algoritmu *Straight walk,* jsou poměrně komplikované a často podstatně zpomalí celý chod algoritmu, protože testy na nalezení cílového trojúhelníku probíhají průběžně a nezávisí na aktuální vzdálenosti od cílového trojúhelníku. Díky tomu všechny následující zmíněné algoritmy založené na algoritmu *Straight walk* tento způsob využívat nebudou.

Na určení cílového trojúhelníku se proto nabízí použití jiného procházkového algoritmu, ale to až v případě, že existuje podezření, že se cílový trojúhelník nachází dostatečně blízko. Vzhledem k tomu, že přechod na jiný procházkový algoritmus proběhne až v těsné blízkosti cílového trojúhelníku, je nejjednodušší a nejspolehlivější volbou použití algoritmu *Remembering stochastic walk* (podkap. 2.4) (eventuálně *Remembering walk*

podkap. 2.3 pro Delaunayovy triangulace). Způsob určení nejvhodnější doby přechodu na tento algoritmus se pak odvíjí od konkrétního algoritmu.

2.10. Distance Straight walk

Tento algoritmus využívá podobného principu jako algoritmus *Distance Fast walk*. Po určeném počtu kroků provedených algoritmem *Straight walk* je kontrolována aktuální vzdálenost od cílového bodu[25] vůči poslední kontrolované vzdálenosti[26]. Tento postup opakujeme tak dlouho, dokud je nová vzdálenost menší (to znamená, že se k cílovému bodu stále přibližujeme). Pokud zjistíme, že jsme se v posledním bloku *Straight walk* již nepřiblížili, přecházíme na algoritmus *Remembering Stochastic walk* (eventuálně na rychlejší *Remembering walk* pro Delaunayovy triangulace), kterým lokalizujeme cílový bod. Následující pseudokód (*Alg. 2.8*) popisuje lokaci bodu pomocí algoritmu *Distance Straight walk*. Na finální dohledání bodu je použit algoritmus *Remembering Stochastic walk*.

```
trojuhelnik distanceStraightWalk(bod cil, int POCET_ITERACI27)
{
 double novaVzdalenost, staraVzdalenost;
 trojuhelnik aktTroj, predchozi;
 hrana hrana;
 bod pocatecniBod, ridiciBod;
 primka ridiciPrimka;  // implicitní předpis přímky

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 novaVzdalenost = Double.MAX_VALUE;
 hrana = hrana z aktTroj naproti vrcholu aktTroj nejblíže cil28;
 pocatecniBod = střed hrany hrana;
 ridiciPrimka = přímka procházející cílem a bodem pocatecniBod;
 predchozi = trojúhelník přes hranu hrana;

 do
 {
  staraVzdalenost = novaVzdalenost;
  for (int i=0; i<POCET_ITERACI; i++)
  {
   hrana = hrana mezi predchozi a aktTroj;
   ridiciBod = bod z aktTroj naproti hraně hrana;
   predchozi = aktTroj;
   if (signum(znamenkovyTest(ridiciPrimka, ridiciBod))==-1)29
    aktTroj = soused přes hranu napravo od ridiciBod;
   else
    aktTroj = soused přes hranu nalevo od ridiciBod;
  }
```

[25] Stejným způsobem jako v podkap. 2.7

[26] V případě první kontroly, kdy poslední kontrolovaná vzdálenost ještě nebyla spočítána je nastavena poslední kontrolovaná vzdálenost na maximální reálnou hodnotu.

[27] Hodnota zvolená vhodně vzhledem k objemu dat a typu triangulace.

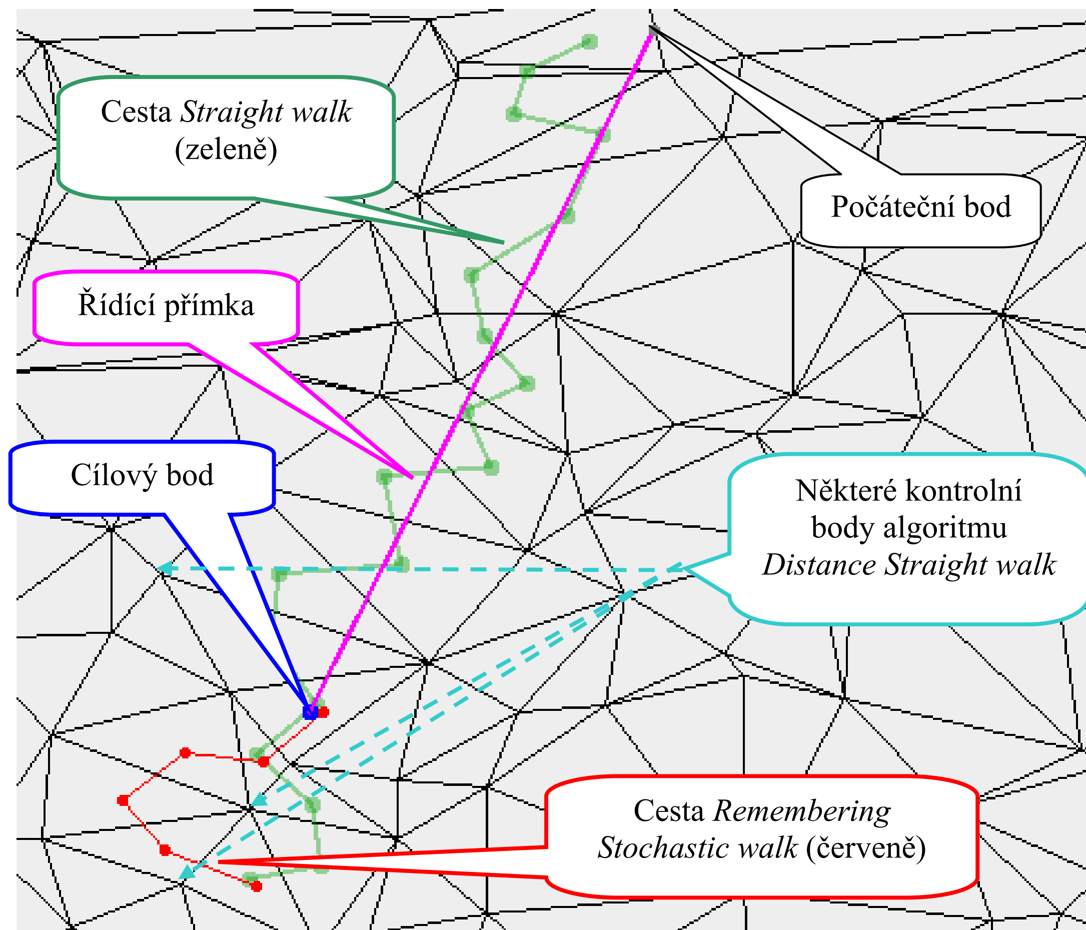
[28] Určíme vrchol, který je nejblíže cíli (výpočtem vzdáleností pro všechny body), potom volíme hranu naproti tomuto bodu.

[29] Dosazení do implicitního předpisu přímky dle 2.2.1

```
  novaVzdalenost = kvadrát vzdálenosti cil od ridiciBod;
 } while (novaVzdalenost<staraVzdalenost);

 return remembering stochastic walk(cil, aktTroj);
}
```

Algoritmus 2.8 – Algoritmus *Distance Straight walk*

Průběh algoritmu je znázorněn na obrázku (*Obr. 2.7*). *Straight walk* procházka začíná v horní části obrázku a je vyznačena zeleně. Pro kontrolu vzdálenosti na obrázku je zvolena hodnota kroku 3 (vzdálenost se kontroluje pro každý 3. trojúhelník). Na obrázku jsou pak modrozelenou šipkou označeny některé kontrolní body (jde o vrcholy kontrolovaného trojúhelníku, pro které se počítá vzdálenost a které se nacházejí blízko cílovému bodu). Pro zmíněný výpočet vzdálenosti kontrolovaných trojúhelníků od cílového bodu byly zvoleny řídící body příslušné vybraným trojúhelníkům. V případě, že kontrolovaný bod je od cílového bodu dále než předchozí kontrolovaný bod, je ukončena přímá procházka a algoritmus pokračuje dále pomocí procházky dle viditelnosti (na obrázku vyznačeno červeně).



Obrázek 2.7 – *Distance Straight walk* s krokem *3*

2.10.1. Shrnutí algoritmu

Stejně jako u *Distance Fast walk* (podkap. 2.7) je i u tohoto algoritmu velmi důležitým aspektem volba počtu iterací (*POCET_ITERACI*), nebo-li kroku, po kterém je kontrolována vzdálenost. Na rozdíl od *Distance Fast walk* však už není zdaleka tak pravděpodobné předčasné ukončení *Straight walk*[30] při malé velikosti kroku. Přesto při malém kroku je často vypočítávána vzdálenost, což představuje určitou režii a zpomalení algoritmu. Stejně tak je ale potřeba myslet na to, že pokud je *POCET_ITERACI* číslo příliš vysoké, může se stát, že algoritmus přejde cílový trojúhelník o další úsek, což bude v konečném efektu opět znamenat zpomalení algoritmu.

Proto je důležité stejně jako u *Distance Fast walk* zvolit jakýsi kompromis, a to i s ohledem na celkový objem vstupních dat. Pokud víme, že procházka bude pravděpodobně dlouhá (velký objem dat), nevadí nám „několik kroků navíc", a tudíž volíme *POCET_ITERACI* vyšší než u menšího objemu vstupních dat.

2.11. Normal Straight walk

Nabízí se nám otázka, jestli na zjišťování dosažení cílového bodu nepoužívat stejný princip, na kterém je založen celý algoritmus *Straight walk*, to znamená určit si kontrolní přímku kolmou na řídící přímku a zároveň procházející cílovým bodem, a pokud se dostaneme za tuto přímku (znaménkový test po dosazení do implicitního předpisu této přímky vrátí hodnotu s opačným znaménkem (v našem případě hodnotu -1) než vracel na začátku procházky), lze říci, že máme přejít na algoritmus *Remembering (Stochastic) walk*, protože jsme již přešli linii cílového bodu. Algoritmus *Normal Straight walk* je popsán následujícím pseudokódem (*Alg 2.9*)

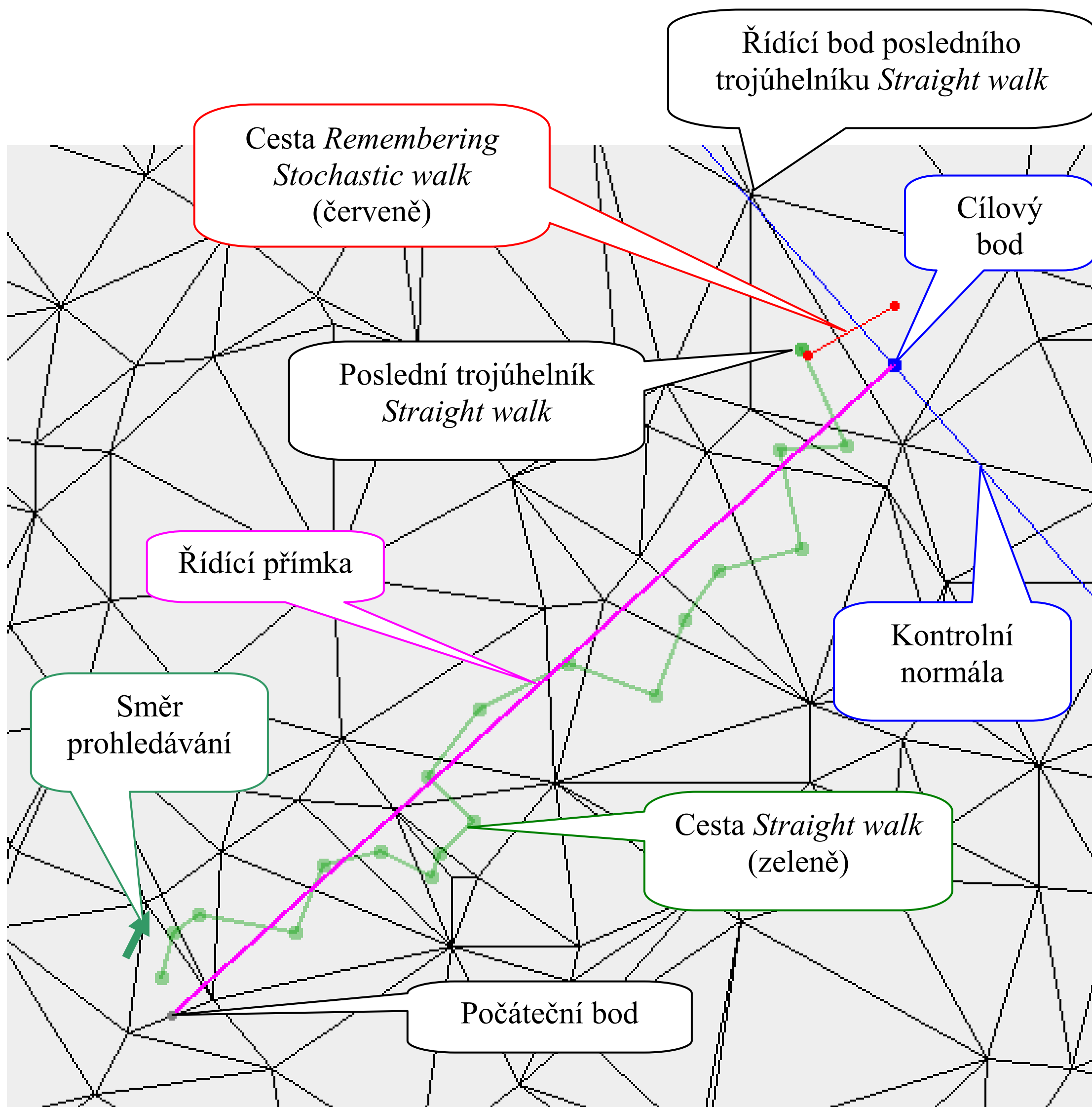
```
trojuhelnik normalStraightWalk(bod cil)
{
 trojuhelnik aktTroj, predchozi;
 hrana hrana;
 bod pocatecniBod, ridiciBod;
 primka ridiciPrimka, kontrolniNormala;    // implicitní předpis

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 hrana = hrana z aktTroj naproti vrcholu aktTroj nejblíže cil[31];
 pocatecniBod = střed hrany hrana;
 ridiciPrimka = přímka procházející cílem a bodem pocatecniBod;
 kontrolniNormala = přímka kolmá na ridiciPrimka procházející cil;
 predchozi = trojúhelník přes hranu hrana;

 do
 {
```

[30] Díky tomu, že není vzdálenost počítána od libovolného vrcholu aktuálního trojúhelníku, ale od řídícího bodu.
[31] Určíme vrchol, který je nejblíže cíli (výpočtem vzdáleností pro všechny body), potom volíme hranu naproti tomuto bodu.

```
      hrana = hrana mezi predchozi a aktTroj;
      ridiciBod = bod z aktTroj naproti hraně hrana;
      predchozi = aktTroj;

      if (signum(znamenkovyTest(ridiciPrimka, ridiciBod))==-1)
       aktTroj = soused přes hranu napravo od ridiciBod;
      else
       aktTroj = soused přes hranu nalevo od ridiciBod;
    } while (signum(znamenkovyTest(kontrolniNormala, ridiciBod))==-1);

   return remembering stochastic walk(cil, aktTroj);
}
```

Algoritmus 2.9 – Algoritmus *Normal Straight walk*

Průběh algoritmu je patrný z následujícího obrázku (*Obr. 2.8*). Procházka začíná vlevo dole a je vyznačena zeleně, růžově je pak vyznačena řídící přímka a modře kontrolní normála (přímka procházející cílovým bodem a kolmá na



Obrázek 2.8 – *Normal Straight walk*

řídící přímku[32]). Dále je v ilustrovaném příkladu vyznačen řídící bod posledního trojúhelníku přímé procházky, díky němuž přímou procházku ukončujeme, protože se již nachází za kontrolní normálou. Červeně je vyznačena cesta provedená algoritmem procházky dle viditelnosti.

2.11.1. Shrnutí Normal Straight walk

Za evidentní výhodu tohoto algoritmu lze pokládat fakt, že na rozdíl od *Distance Straight walk* nemůže nastat situace, kdy přímá procházka cílový bod přejde. Navíc lze předpokládat, že po ukončení přímé procházky bude cílový bod velice blízko[33] a délka procházky pomocí *Remembering (Stochastic) walk* pak bude minimální.

2.12. Orthogonal walk

Orthogonal walk neboli *pravoúhlá procházka* je procházka, kdy se nejdříve k cílovému bodu přibližujeme po jedné souřadné ose a pak po druhé ose. Následující algoritmus bude využívat principu přímé procházky (podkap. 2.9, 2.10, 2.11) s tím rozdílem, že algoritmus bude rozdělen na dvě části, kdy každá část procházky bude řízena jinou řídící přímkou. První část procházky bude mít řídící přímku rovnoběžnou se souřadnou osou X a zároveň bude řídící přímka procházet počátečním bodem. Druhá část bude mít řídící přímku rovnoběžnou se souřadnou osou Y a zároveň bude řídící přímka procházet cílovým bodem.

Je pochopitelné, že takováto procházka bude z hlediska prošlých trojúhelníků v průměru delší než procházky předchozích algoritmů. Kouzlo tohoto algoritmu spočívá však ve využití faktu, že jsou jednotlivé řídící přímky rovnoběžné se souřadnými osami. Proto není při procházce potřeba jakýchkoliv výpočtů, postačí pouze porovnání příslušné souřadnice řídícího bodu se souřadnicí řídící přímky[34]. Pouhá porovnání příslušné souřadnice nám budou stačit i při kontrole, zda se má příslušná část procházky ukončit. Díky tomu se i delší pravoúhlá procházka stává rychlejší než kratší, ale výpočetně náročnější procházky.

2.12.1. Detailní popis algoritmu

V první části algoritmu se přibližujeme k cílovému bodu ve směru osy X. Vycházíme z předpokladu, že již máme vybraný počáteční trojúhelník. Jako počáteční hranu zvolíme takovou hranu, která je naproti tomu bodu z počátečního trojúhelníku, který je nejblíže (pouze ale v X souřadnici)

---

[32] To, že na obrázku nesvírá kontrolní normála s řídící přímkou pravý úhel, je dáno skutečností, že měřítko osy X je trochu odlišné od měřítka osy Y.
[33] Z hlediska počtu trojúhelníků mezi aktuálním trojúhelníkem a trojúhelníkem obsahujícím cílový bod.
[34] Jaká souřadnice (jestli X nebo Y) záleží na směru, ve kterém zrovna procházíme.

cílovému bodu. Jako počáteční bod pak zvolíme střed počáteční hrany. V procházce definujeme pojem řídící bod trojúhelníku. Tento bod je v každém trojúhelníku procházky jednoznačně určen a je to bod přímo naproti hraně, po které jsme do trojúhelníku přišli. V případě počátečního trojúhelníku je řídící bod přímo naproti počáteční hraně. Nyní zjistíme, jestli má počáteční bod menší nebo větší X souřadnici než cílový bod a od toho odvíjíme směr procházky.

Nyní začneme samotnou procházku. Následující popis činnosti se týká případu, kdy měl počáteční bod X souřadnici menší než cílový bod. Opakovaně kontrolujeme řídící bod aktuálního trojúhelníku[35]. V případě, že má řídící bod aktuálního trojúhelníku Y souřadnici nižší než počáteční bod, přecházíme do následujícího trojúhelníku po hraně napravo od řídícího bodu. Ve zbylých případech přecházíme do následujícího trojúhelníku po hraně nalevo. Tento způsob procházky končí v okamžiku, kdy má řídící bod aktuálního trojúhelníku X souřadnici vyšší než cílový bod.

V případě, že má počáteční bod X souřadnici vyšší než cílový bod, je způsob procházky obdobný. Jediné rozdíly jsou v tom, že místo hrany napravo od řídícího bodu volíme hranu nalevo a naopak. Samozřejmě také procházení nekončí při vyšší souřadnici X jako v předchozím případě, ale končí při nižší X souřadnici řídícího bodu než je X souřadnice bodu cílového.

Po skončení procházky ve směru osy X přichází na řadu procházka ve směru osy Y. Princip této procházky je naprosto identický jako v případě procházky ve směru osy X a stejně jako u procházky ve směru osy X můžou nastat 2 možné případy, a to, že má cílový bod Y souřadnici vyšší než počáteční bod nebo má cílový bod Y souřadnici nižší než počáteční bod, kdy počáteční bod je střed hrany počátečního trojúhelníku[36] naproti bodu z počátečního trojúhelníku, který je nejblíže (ve směru osy Y) cílovému bodu.

Potom v případě, kdy je X souřadnice počátečního bodu menší než u cílového bodu, procházíme triangulaci způsobem, že kontrolujeme vždy řídící bod aktuálního trojúhelníku a když je jeho X souřadnice nižší než X souřadnice počátečního bodu, přecházíme do trojúhelníku po hraně napravo od řídícího bodu. V opačném případě volíme následující trojúhelník přes hranu nalevo od řídícího bodu. Toto opakujeme, dokud Y souřadnice řídícího bodu není vyšší než Y souřadnice cílového bodu. Pokud je podmínka splněna, pravoúhlá procházka končí.

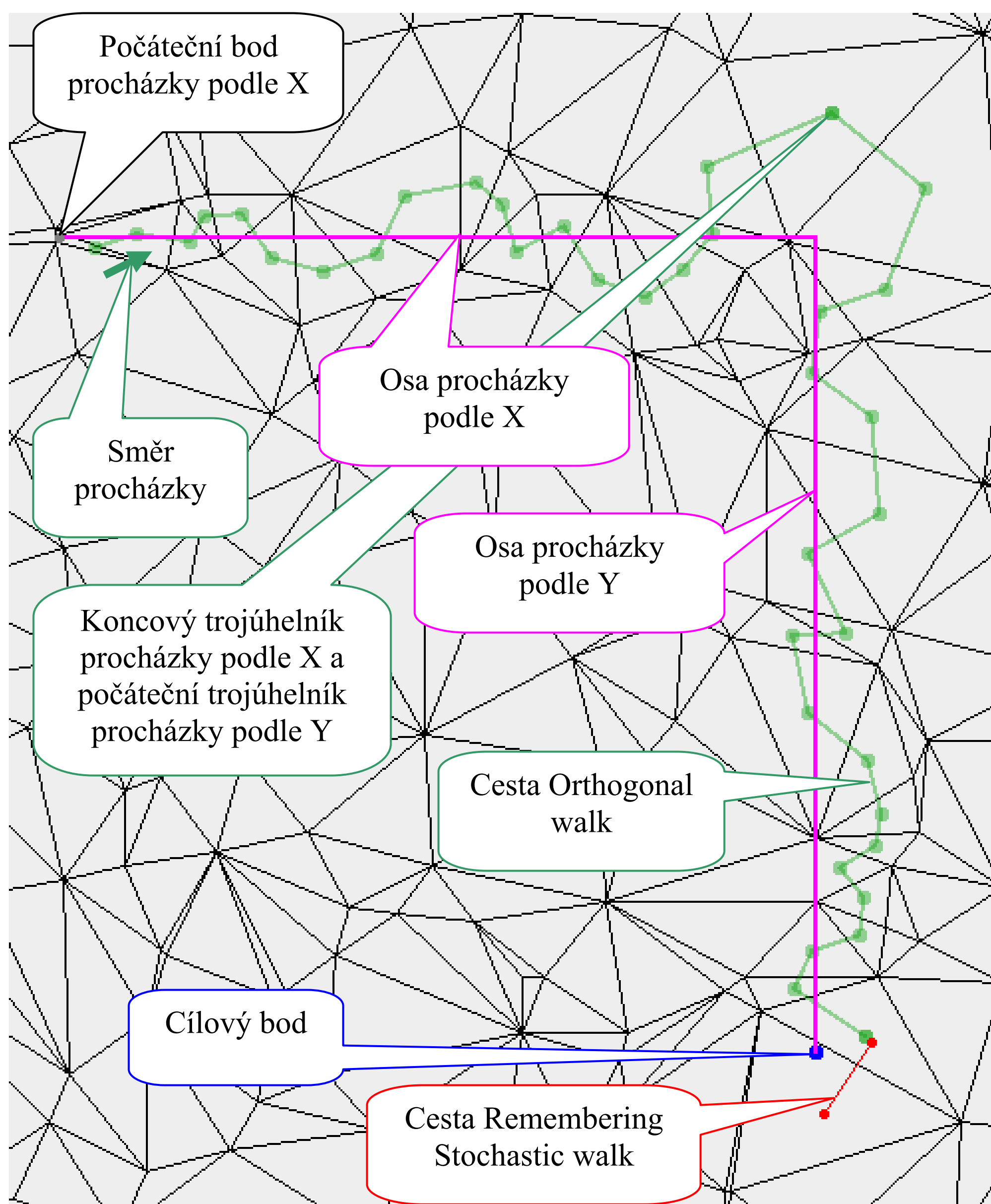
Nyní už jsme velice blízko trojúhelníku obsahujícímu cílový bod, ale není zaručeno, že se v cílovém trojúhelníku nalézáme, proto k dohledání cílového trojúhelníku využijeme jakýkoliv jiný procházkový algoritmus, nejlépe *Remembering (Stochastic) walk* .

---

[35] Řídící bod aktuálního trojúhelníku je řídící bod trojúhelníku, ve kterém se procházkový algoritmus v daném okamžiku nachází.

[36] Počáteční trojúhelník je v tomto případě trojúhelník, ve kterém skončila procházka ve směru osy X.

Na následujícím obrázku (*Obr. 2.9*) je nastíněn celý princip *Orthogonal walk*. Procházka začíná vlevo nahoře a fialově jsou vyznačeny osy podle nichž algoritmus prochází. Vpravo nahoře je následně označen trojúhelník, ve kterém končí procházka podle X a začíná procházka podle Y. Algoritmus *Orthogonal walk* je pak popsán v následujícím pseudokódu (*Alg 2.10*).



Obrázek 2.9 – *Orthogonal walk*

```
trojuhelnik orthogonalWalk(bod cil)
{
 trojuhelnik aktTroj, predchozi;
 hrana hrana;
 bod pocatecniBod, ridiciBod;

 aktTroj = libovolný trojúhelník z triangulace; // viz podkap. 2.13
 ridiciBod = vrchol z aktTroj, který je nejblíže cil v X ose;
```

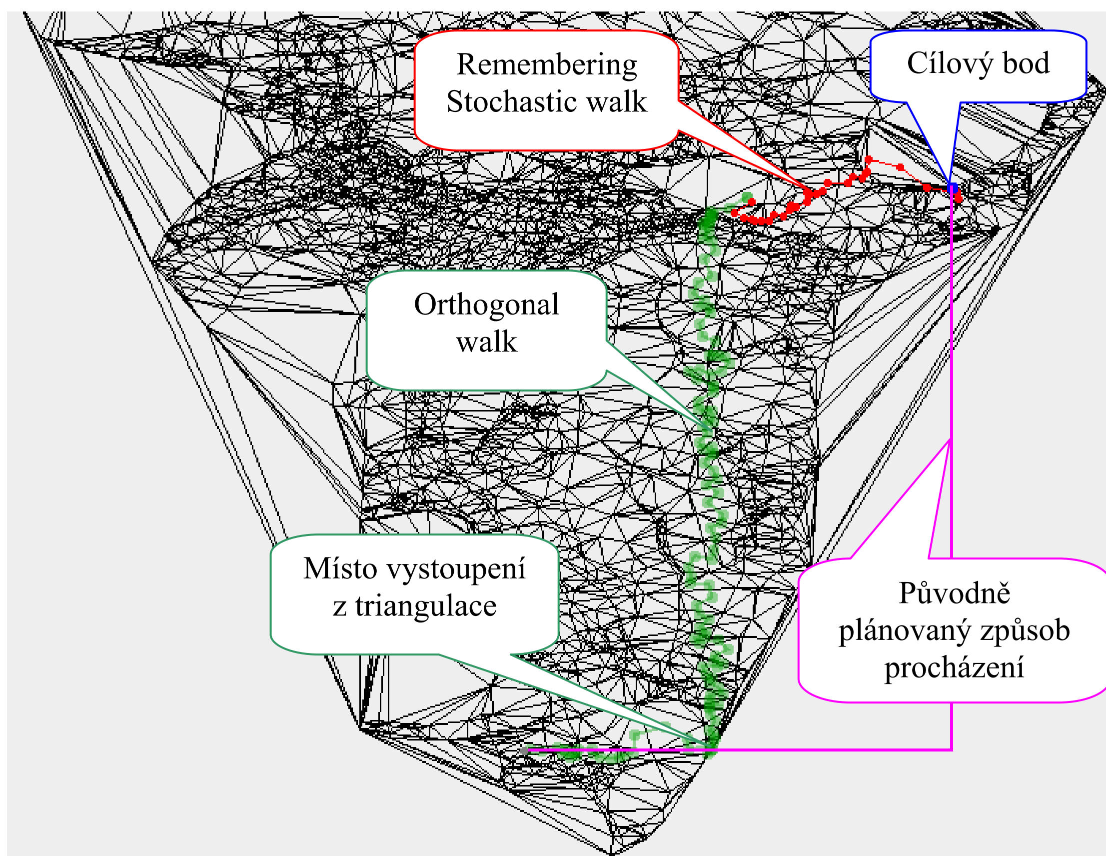
```
hrana = hrana z aktTroj naproti ridiciBod;
pocatecniBod = střed hrany hrana;
predchozi = trojúhelník přes hranu hrana;

if (pocatecniBod.X < cil.X)
{
 while (ridiciBod.X < cil.X)
 {
  hrana = hrana mezi predchozi a aktTroj;
  ridiciBod = bod z aktTroj naproti hraně hrana;
  predchozi = aktTroj;
  if (ridiciBod.Y > pocatecniBod.Y )
   aktTroj = soused přes hranu napravo od ridiciBod;
  else
   aktTroj = soused přes hranu nalevo od ridiciBod;
 }
}
else
{
 while (ridiciBod.X > cil.X)
 {
  hrana = hrana mezi predchozi a aktTroj;
  ridiciBod = bod z aktTroj naproti hraně hrana;
  predchozi = aktTroj;
  if (ridiciBod.Y > pocatecniBod.Y)
   aktTroj = soused přes hranu nalevo od ridiciBod;
  else
   aktTroj = soused přes hranu napravo od ridiciBod;
 }
}

ridiciBod = bod z aktTroj, který je nejblíže cil v Y ose;
hrana = hrana z aktTroj naproti bodu ridiciBod;
pocatecniBod = střed hrany hrana;
predchozi = trojúhelník přes hranu hrana;

if (pocatecniBod.Y < cil.Y)
{
 while (ridiciBod.Y < cil.Y)
 {
  hrana = hrana mezi predchozi a aktTroj;
  ridiciBod = bod z aktTroj naproti hraně hrana;
  predchozi = aktTroj;
  if (ridiciBod.X > pocatecniBod.X)
   aktTroj = soused přes hranu napravo od ridiciBod;
  else
   aktTroj = soused přes hranu nalevo od ridiciBod;
 }
}
else
{
 while (ridiciBod.Y > cil.Y)
 {
  hrana = hrana mezi predchozi a aktTroj;
  ridiciBod = bod z aktTroj naproti hraně hrana;
  predchozi = aktTroj;
  if (ridiciBod.X > pocatecniBod.X)
   aktTroj = soused přes hranu nalevo od ridiciBod;
  else
   aktTroj = soused přes hranu napravo od ridiciBod;
 }
}
```

```
 return remembering stochastic walk(cil, aktTroj);
}
```

Algoritmus 2.10 – Algoritmus *Orthogonal walk*

2.12.2. Shrnutí Orthogonal walk

Velice elegantní algoritmus, obzvlášť díky téměř absolutní nepotřebnosti jakýchkoliv výpočtů, což může být v určitých aplikacích dosti podstatné. Za částečnou nevýhodu lze ale pokládat fakt, že vzhledem ke způsobu procházení můžeme při přibližování po ose X vystoupit z triangulace (*Obr. 2.10*). Tuto situaci ovšem není veliký problém ošetřit.



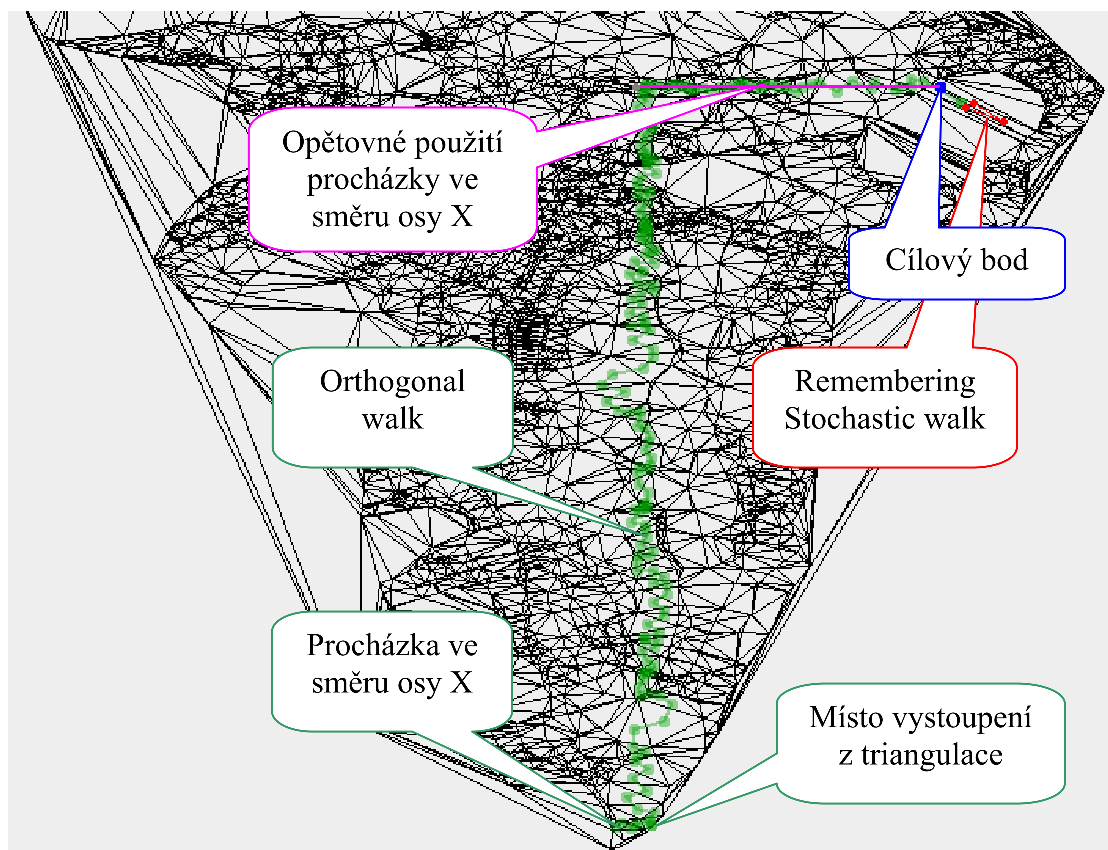
Obrázek 2.10 – *Orthogonal walk* (Vystoupení z triangulace)

V případě, že přes vybranou hranu nelze přejít do dalšího trojúhelníku, končíme procházku ve směru X a automaticky přecházíme na procházku ve směru Y. Po skončení procházky ve směru Y pak dojdeme do cílového trojúhelníku pomocí jiného procházkového algoritmu (v našem případě *Remembering (Stochastic) walk*).

Může se také stát, že v určitých případech budeme po skončení procházky ve směru osy Y stále dost daleko od cílového bodu. Proto je potřeba zvážit, jak pravděpodobná je tato situace a jestli by nebylo výhodnější použít rychlejší

algoritmus k dohledání cíle než je *Remembering (Stochastic) walk*, například algoritmus *Normal Straight walk.*

Jako lepší řešení se ale jeví podmíněné použití takovéhoto algoritmu, tedy použití pouze v případě, že procházka z triangulace vystoupila. Při hlubší analýze navíc zjišťujeme, že se i v takovémto případě zpravidla stačí přiblížit pouze ve směru osy X, proto je pravděpodobně nejlepší řešení tohoto problému opakované použití procházky ve směru osy X v případě, že již skončila procházka ve směru osy Y a procházka ve směru osy X byla ukončena předčasně vystoupením z triangulace (*Obr. 2.11*).



Obrázek 2.11 – *Orthogonal walk* (Opětovné použití procházky ve směru X)

2.13. Výběr počátečního trojúhelníku

Důležitou součástí algoritmů procházky je výběr trojúhelníku, z kterého procházku začínáme. Do *výběru počátečního trojúhelníku* se vyplatí investovat i nějaký čas určený pro samotnou procházku, protože pak můžeme zmiňovanou procházku dosti výrazně zkrátit.

Úplně nejjednodušším, ale také velmi účinným způsobem *výběru počátečního trojúhelníku* je volba trojúhelníku pokud možno co nejbližšího hledanému bodu [Mu99]. Výpočet vzdálenosti trojúhelníku od hledaného bodu

zjednodušíme na výpočet kvadratické[37] vzdálenosti libovolného bodu trojúhelníku od cílového bodu..

Samotný algoritmus *výběru počátečního trojúhelníku* pak z náhodného vzorku několika trojúhelníků vybere trojúhelník nejbližší cílovému bodu. Důležitou otázkou ale zůstává správný výběr velikosti takovéhoto „náhodného vzorku“. Pokud zvolíme vzorek příliš veliký nebo naopak příliš malý, můžeme se hodně vzdálit očekávanému časovému optimu. Proto jsem se zabýval i vhodnou volbou velikosti takovéhoto vzorku odvíjeného od očekávané velikosti vstupu n[38] (podkap. 3.3). Algoritmus *výběru počátečního trojúhelníku* popisuje následující pseudokód (*Alg. 2.11*).

```
global trojuhelnik[] trojuhelniky;[39]

trojuhelnik vyberPrvniho(bod cil, int vyberu[40])
{
 double novaVzdalenost, staraVzdalenost;
 bod vybranyBod;
 trojuhelnik vybranyTroj, t;

 vybranyTroj = trojuhelniky[0];
 vybranyBod = libovolný vrchol z vybranyTroj;
 staraVzdalenost = kvadratická vzdálenost vybranyBod od cil;

 for (int i=0; i<vyberu; i++)
 {
  t = náhodně zvolený trojúhelník z trojuhelniky;
  vybranyBod = libovolný vrchol t;
  novaVzdalenost = kvadratická vzdálenost vybranyBod od cil;
  if (novaVzdalenost < staraVzdalenost)
  {
   staraVzdalenost = novaVzdalenost;
   vybranyTroj = t;
  }
 }

 return vybranyTroj;
}
```

2.14. Problém procházkových algoritmů

Všechny procházkově algoritmy jsou založené na různých znaménkových testech (podkap. 2.2) nebo využívají procházkové algoritmy, které dále znaménkové testy potřebují (např. *Orthogonal walk* apod.). Samozřejmě veškeré znaménkové testy pracují s reálnými čísly. Tato čísla jsou pak v počítači reprezentována jako čísla s pohyblivou řádovou čárkou (viz standard IEEE 754 [Ka96]), což znamená, že můžou být někdy i částečně nepřesná.

[37] Jelikož nám vzdálenost slouží pouze pro porovnání s ostatními vzdálenostmi, je použití operace odmocniny v tomto případě zbytečné.
[38] n = počet bodů triangulace
[39] Globální proměnná procházené triangulace.
[40] Proměnná udávající počet výběrů počátečního trojúhelníku.

Vlivem těchto drobných nepřesností se pak může stát, že znaménkový test vrátí jinou hodnotu, než jakou bychom očekávali (např. [He03a]).

Procházkové algoritmy jsou velice stabilní v souvislosti se zacyklením. Ovšem ve výjimečných případech se může stát, že se zacyklí i procházkový algoritmus. Jde však pouze o případy, kdy cílový bod je totožný (nebo velice blízký) s některým jiným vrcholem triangulace. Ovšem i v případě totožnosti cílového bodu s některým vrcholem triangulace je takovéto zacyklení velice nepravděpodobné.

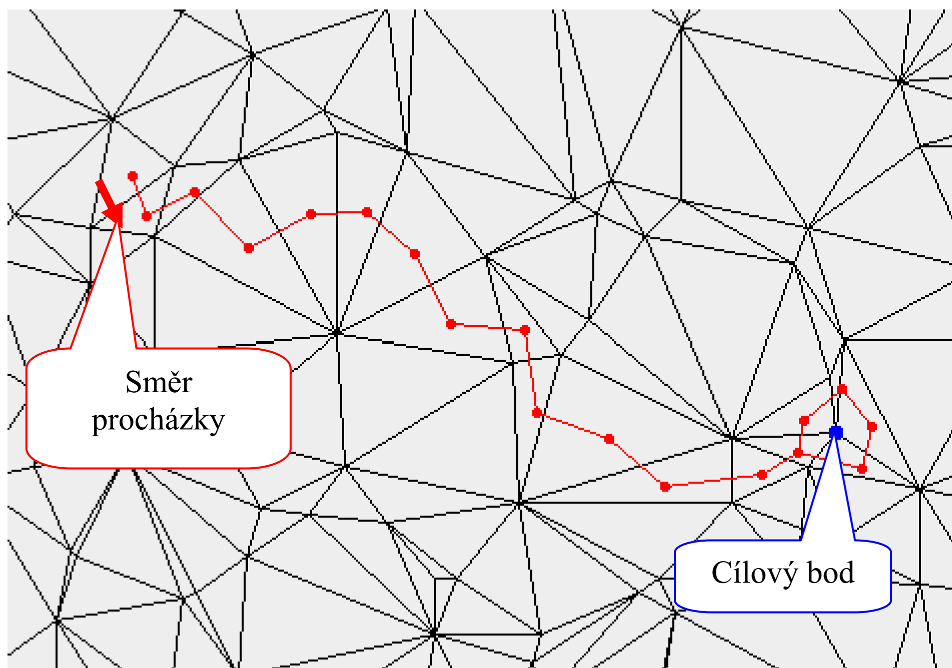
Obecně vzato k zacyklení dojde v případě, že znaménkové testy budou v každém trojúhelníku vracet alespoň jednu zápornou hodnotu, která bude signalizovat přechod do následujícího trojúhelníku (přes hranu, která způsobila vrácení záporné hodnoty). To bude znamenat, že v každém trojúhelníku procházky bude existovat alespoň jedna hrana, od které se nachází bod nalevo (tedy mimo trojúhelník). V případě, že by byly znaménkové testy stoprocentně přesné, nemůže pochopitelně tato situace nikdy nastat, protože i v případě bodu například na hraně bude navrácena hodnota nulová. Ovšem s vědomím, že znaménkové testy můžou mírně zkreslovat, zjišťujeme, že například místo nulové hodnoty může znaménkový test vrátit zápornou hodnotu blízkou nule.

Zamyslíme-li se nad tím, v jakém případě může zacyklení nastat, nabízí se okamžitě bod nacházející se kdekoliv na hraně mezi dvěma trojúhelníky. Při kontrole prvního trojúhelníku pak u jedné z hran (na které se nachází cílový bod) dostaneme v některých případech vlivem nepřesnosti znaménkových testů zápornou hodnotu blízkou nule místo očekávané nuly. Tedy přejdeme po této hraně do sousedního trojúhelníku. V tomto trojúhelníku pak zápornou hodnotu ze znaménkového testu můžeme potenciálně dostat pouze u hrany, po které jsme do trojúhelníku přišli. Nesmíme ovšem zapomínat, že používáme tzv. *Remembering procházkové algoritmy*, které si pamatují předchozí trojúhelník a hranu vedoucí k tomuto trojúhelníku již nekontrolují. Je tedy pochopitelné, že zacyklení v tomto případě nastat nemůže.

Zjišťujeme tedy, že k zacyklení může dojít pouze v případě, když se cílový bod nachází na několika hranách zároveň (nebo velmi blízko těmto hranám). To je právě případ, kdy je cílový bod totožný (nebo velice blízký) s jiným bodem triangulace. Z následujícího obrázku (*Obr. 2.12*) je patrné zacyklení v *Remembering Stochastic walk*, které nastalo při lokaci cílového bodu totožného s jedním z bodů triangulace. Procházka začíná vlevo nahoře. V dolní části vpravo je potom vidět cílový bod, který je zároveň vrcholem triangulace (celkově vrcholem pěti trojúhelníků).

2.15. Adaptivní robustní znaménkový test

Problém procházkových algoritmů vzniklý využíváním čísel s pohyblivou řádovou čárkou (zmíněno v podkap. 2.14) lze odstranit, použijeme-li k výpočtu speciální adaptivní robustní algoritmy. Tyto algoritmy (např. [Fo93, Fo96,



Obrázek 2.12 – Ukázka zacyklení (Remembering Stochastic walk)

Li07]) počítají do určité míry srovnatelně s klasickými výpočetními postupy a taktéž využívají čísla s pohyblivou řádovou čárkou. Jejich speciální vlastností je ale adaptivita. V případě podezření na vznik jistých numerických nepřesností algoritmy test opakují s rozšířenou přesností reálných čísel (na rozšíření přesnosti lze využít například celočíselné datové typy [Fo96]). Takto se přesnost opakovaně zvyšuje, dokud je stále podezření na vznik numerických nepřesností. V případě, že již podezření není, je vrácen výsledek upravený na nejbližší reálné číslo reprezentované klasicky s pohyblivou řádovou čárkou.

Algoritmy zmíněné výše jsou sice velice spolehlivé a robustní, ale jejich velikou nevýhodou je, že jsou poměrně pomalé. To je způsobeno především množstvím různých kontrolních testů, případně opakovaném dopočítávání výsledků se zvyšovanou numerickou přesností. Tento problém se snaží řešit J. R. Shewchuk [Sh96a, Sh96b, Sh97]. Přistupuje k různým geometrickým úlohám zvlášť a snaží se provádět výpočty s ohledem na jejich pozdější využití. V případě testu orientace (ekvivalentní test znaménkovému testu), kdy je podstatné určit pouze, o kterou orientaci se jedná (levotočivá, pravotočivá, v přímce), není důležitá výsledná hodnota, ale znaménko výsledné hodnoty (případně hodnota rovna 0), proto stačí zpřesňovat pouze takové výpočty, které potenciálně můžou ovlivnit samotné znaménko. Díky i dalším vylepšením oproti jiným zpřesňujícím algoritmům se v normálním případě (v případě, kdy není potřeba provádět testy opakovaně s rozšiřováním přesnosti) stává *Shewchukův znaménkový test* velice rychlým a je jen o málo pomalejší než běžný výpočet s čísly s pohyblivou řádovou čárkou.

Díky těmto vlastnostem je možno *Shewchukův znaménkový test* využít ve všech algoritmech procházky. Je však potřeba myslet na to, že v určitých případech můžou být prováděny zpřesňující výpočty častěji, a tudíž se i procházkový algoritmus výrazně zpomalí. Vzhledem k tomu, že málokdy

máme předem informace o numerických vlastnostech dané triangulace, je otázka nasazení *Shewchukova znaménkového testu* velice nejednoznačná. O případném plném nasazení je proto potřeba rozhodnout s přihlédnutím k požadavkům na danou aplikaci (především jde o rychlostní požadavky).

Pokud bychom ale použili *Shewchukův znaménkový test* pouze částečně, to znamená pouze na malou (z hlediska celkové délky procházky zanedbatelnou) část procházky a to konkrétně až v případě, kdy budeme cílovému bodu blízko, nastává jiná situace. Na celkovém čase se totiž případné zpřesňující testy projeví pouze minimálně. Naskýtá se otázka jakým způsobem zjistit, že jsme již blízko cílovému bodu. Nejlepší je využít blízkosti k cílovému bodu vyplývající přímo z charakteru samotného algoritmu a neprovádět žádné výpočty navíc. Velice vhodné je proto použití *Shewchukova znaménkového testu* v algoritmech *Flag Fast walk* (podkap. 2.8), *Normal Straight walk* (podkap. 2.11) a *Orthogonal walk* (podkap. 2.12), kde by byl test použit pouze na finální fázi algoritmu – konečnou lokaci bodu pomocí *Remembering (Stochastic) walk*. Už méně vhodné je použití *Shewchukova znaménkového testu* na finální fázi algoritmů *Fast walk* (podkap. 2.5), *Distance Fast walk* (podkap.2.7) a *Distance Straight walk* (podkap. 2.10), protože z charakteru algoritmů nelze zaručit, že v momentě přechodu na finální fázi budeme cílovému bodu dostatečně blízko.

3. Praktické testy procházek v E2

3.1. Použité programové prostředky

3.1.1. Testovací data

Jako testovací data jsou používány triangulace uložené ve vstupních souborech v předepsaném formátu. Vstupní soubor obsahuje 3 části, kdy první část obsahuje seznam bodů triangulace, jejichž počet je určen číslem na prvním řádku. Každý bod je určen jedním řádkem, který obsahuje 2 reálná čísla oddělená mezerou (souřadnice X a Y).

Další část obsahuje jednotlivé trojúhelníky. Jejich počet je určen celočíselnou hodnotou a každý trojúhelník je definován na jedné řádce, na které jsou mezerou oddělené celočíselné hodnoty, které udávají indexy jednotlivých bodů z první části. Body jsou orientovány proti směru hodinových ručiček.

Třetí část obsahuje informace o sousedech jednotlivých trojúhelníků. Jejich počet je určen opět celočíselnou hodnotou a pro každý trojúhelník jsou všechny sousední trojúhelníky určeny jednou řádkou, na které jsou mezerou oddělené celočíselné hodnoty, které udávají indexy jednotlivých trojúhelníků z druhé části. Sousedé jsou uspořádáni v takovém pořadí, aby každý soused s určitým indexem odpovídal bodu trojúhelníku se stejným indexem, přičemž bod odpovídá sousedovi v případě, že je naproti hraně, po které lze do příslušného souseda přejít.

3.1.2. Testovací aplikace

Testovací aplikace pro 2D procházkové algoritmy je kompletně implementovaná v programovacím jazyce *Java* a je distribuovaná v podobě spustitelného *.jar* souboru. Kromě standardního menu je součástí aplikace i množství ovládacích prvků, s jejichž pomocí lze zvolit konkrétní procházkový algoritmus i nastavit množství volitelných parametrů konkrétní vybrané procházky. Pro vstupní data obsahující 10000 a méně bodů je zobrazována i vizualizace procházky. V menu je možné navolit množství různých srovnávacích testů.

3.1.3. Použité datové struktury

Vzhledem k tomu, že na implementaci testovací aplikace pro 2D procházkové algoritmy byl použit programovací jazyk *Java*, veškeré datové struktury byly navrženy s důrazem na objektovou reprezentaci. Krom toho byl při návrhu datových struktur kladen důraz na paměťové nároky, a proto datové struktury obsahují pouze součásti nezbytně nutné pro procházkové algoritmy.

Rozlišujeme dva základní typy objektů. Objekt bod (*Point*), který obsahuje X a Y souřadnici příslušného bodu, případně i index příslušného bodu. Objekt trojúhelníku (*Triangle*) pak obsahuje pole bodů (pole objektů typu *Point*), které trojúhelník generují, a dále obsahuje pole sousedních trojúhelníků (pole objektů typu *Triangle*) [41]. Případně objekt trojúhelníku obsahuje i index příslušného trojúhelníku v triangulaci. Důležitou vlastností objektu *Triangle* je orientace generujících bodů proti směru hodinových ručiček. Další důležitá vlastnost objektu *Triangle* je také, že index sousedního trojúhelníku v poli sousedních trojúhelníků je vždy stejný jako index bodu v poli bodů, který je naproti hraně, po které se do zmíněného sousedního trojúhelníku přejde. Této vlastnosti lze velice dobře využívat například v přímých nebo pravoúhlých procházkách.

Pro potřeby procházky stačí uchovávat už jen pole všech trojúhelníků v triangulaci. Díky tomu dosáhneme přiměřených paměťových nároků.

3.1.4. Způsob testování

Aplikace nejdříve vyžaduje načtení triangulace v předepsaném formátu (3.1.1), z které vytvoří odpovídající datovou strukturu (3.1.3). Dále závisí na uživateli, jestli načte lokalizované (cílové) body ze souboru nebo si je nechá vygenerovat přímo aplikací. Poté uživatel zvolí příslušný procházkový algoritmus, který vyžaduje povinný celočíselný parametr udávající počet iterací *výběru počátečního trojúhelníku* (podkapitola 2.13). Některé algoritmy můžou vyžadovat i další parametry. Po spuštění algoritmus lokalizuje všechny cílové body a vrátí čas potřebný k jejich lokaci. V průběhu algoritmu není žádným způsobem modifikovaná výchozí datová struktura[42] (aby šlo identické lokace provést znovu jiným procházkovým algoritmem – z důvodu lepšího srovnání jednotlivých algoritmů). Pokud bude v dalším textu zmiňováno číslo n v souvislosti s velikostí triangulace, představuje vždy počet vrcholů triangulace.

Algoritmy byly testovány pod operačním systémem *MS Windows XP Professional* na Javě verze 5. Hardwarová konfigurace počítače: procesor Intel Pentium D 3,0GHz 2x2MB L2 cache přetaktovaný na 3,3GHz s 3GB RAM.

3.1.5. Řešení nestandardních situací

Nejdříve je nutno říci, jak se algoritmy chovají v případě, že se bod nachází přímo na přímce (výsledek znaménkového testu je 0). Už ze samotných pseudokódů je patrné, že se algoritmy chovají úplně stejně, jakoby byl výsledek kladný (tzn. bod by se nacházel vpravo od přímky).

---

[41] Pro potřeby algoritmu *Flag Fast walk* [2.8] obsahuje objekt *Triangle* i informaci o předchozím navštívení trojúhelníku.

[42] S výjimkou algoritmu *Flag Fast walk* [2.8], v jehož průchodu jsou měněny příznaky procházených trojúhelníků.

Chování algoritmů v případě, že hledaný bod se nachází mimo triangulaci, je velice jednoduché. Využíváme situace, kdy je v případě, že se po hraně opouští triangulace, nastaven již při vytváření struktury soused příslušný takovéto hraně trojúhelníku na hodnotu *null*. Procházkový algoritmus není poté třeba již nijak modifikovat, pouze jsou hlídány úseky kódu, v kterých může dojít ke změně aktuálního trojúhelníku. Aktuální trojúhelník je pak po každé změně kontrolován a pokud je jeho hodnota nastavená na hodnotu *null*, končíme procházku s výsledkem, že hledaný bod se nachází mimo triangulaci.

3.2. Java Native Interface a ShewLib

3.2.1. Java Native Interface (JNI)

Java Native Interface neboli *JNI* je framework dovolující javovským programům volat nativní metody (metody vytvořené v jiném programovacím jazyce než je *Java* – nejčastěji *C*, *C++* nebo *Assembler*) nebo naopak. V dalším textu se budeme zabývat pouze možností volání nativních metod z javovského kódu.

Pro volání nativních metod může existovat řada důvodů. Jedním z těchto důvodů je nedostupnost zdrojového kódu v *Javě* a nemožnost předělání dostupného zdrojového kódu do *Javy* (například z časových nebo technologických důvodů). Dalším důvodem pro použití *JNI* v *Javě* mohou být různé mimofunkční požadavky na inkriminované metody, nejčastěji pak požadavek na rychlost. Vzhledem k tomu, že spuštění nativní metody je nezávislé na programovacím jazyce *Java* a závisí pouze na konkrétním jazyce, v jakém je metoda napsána, je pochopitelné, že metody nativních jazyků jako je *C* nebo *C++* mohou být výrazně rychlejší.

Použití JNI sebou ale nese i řadu nevýhod. Za hlavní nevýhodu lze považovat ztrátu přenositelnosti. Nativní metody jsou volány z nativních knihoven, které jsou načteny před samotným voláním metod. Je pochopitelné, že je potřeba pro každý typ operačního systému mít zvlášť přeložené nativní knihovny, čímž ztrácíme přenositelnost. Navíc musíme zajistit i dostupnost nativních knihoven na klientském počítači, pravděpodobně distribucí těchto knihoven přímo s programem. Za další nevýhodu lze považovat i velmi obtížné ladění nativních metod. Z těchto důvodů je třeba případné použití nativních metod důkladně zvážit.

[Sp02]

3.2.2. Použití Shewchukovy knihovny pro znaménkový test

V podkapitole 2.15 je vysvětlen význam *Shewchukova znaménkového testu* pro využití v procházkových algoritmech. Při použití *Shewchukova znaménkového testu* v programovacím jazyce *Java* narazíme na řadu překážek. Jednou z těchto překážek je volná dostupnost kódu znaménkového testu

v programovacím jazyce *C/C++*, protože předělání zdrojového kódu do programovací jazyka *Java* by zabralo relativně hodně času. Navíc je otázkou, jestli je toto přepsání rozumná volba, protože interpretovaný javovský kód bude jistě pomalejší než původní verze v *C/C++*. Proto i přes ztrátu přenositelnosti byla pro testovací aplikaci zvolena možnost jednoduší a rychlejší – využití metod nativní knihovny.

Pro potřeby 2D procházek využívá testovací aplikace dvě metody z *Shewchukovy knihovny* [Sh96b]. Jde o inicializační metodu *exactinit()*, která je volána pouze jednou před samotným využitím znaménkových testů, pro které je využívána metoda *orient2d(pa, pb, pc)*, kde jednotlivé parametry *pa*, *pb*, *pc* jsou dvourozměrné vektory jednotlivých bodů[43].

3.2.3. Použití JNI pro C/C++ funkce/metody

V následujícím textu se pokusím vysvětlit postup začlenění nativních metod do javovské aplikace. Postup bude popisovat integraci *Shewchukovy knihovny* včetně použití příslušných nativních metod potřebných pro 2D procházkové algoritmy.

Nejdříve je potřeba doplnit javovskou třídu o příslušné nativní metody. V našem případě se jedná o třídu *ShewLib* v balíčku *shewlib* (dále je zobrazen relevantní úsek kódu této třídy).

```
...
private native void exactinit();
public native double orient2d(double ax, double ay, double bx, double
by, double cx, double cy);
...
```

Po přeložení této třídy je nutné vygenerovat hlavičkový soubor jazyka *C*, který použijeme k implementaci nativních metod (v našem případě k implementaci volání nativních metod). K vygenerování využijeme nástroj *javah*, který je standardní součástí *JDK*[44]. V našem případě bude příkaz pro vygenerování hlavičkového souboru třídy *ShewLib* vypadat následovně.

```
javah shewlib.ShewLib
```

Vygenerovaný hlavičkový soubor *shewlib_ShewLib.h* pak vypadá takto:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

/* Header for class shewlib_ShewLib */
#ifndef _Included_ shewlib_ShewLib
#define _Included_shewlib_ShewLib
#ifdef __cplusplus extern "C"
```

[43] Každý bod je reprezentován pomocí dvourozměrného pole reálných hodnot.
[44] JDK – Z anglického *Java Development Kit*. *JDK* je soubor základních nástrojů pro vývoj aplikací pro platformu *Java*. Někdy bývá označován jako *Java SDK*

```
{
  #endif

  JNIEXPORT void JNICALL Java_shewlib_ShewLib_exactinit
    (JNIEnv *, jobject);

  JNIEXPORT jdouble JNICALL Java_shewlib_ShewLib_orient2d (JNIEnv *,
  jobject, jdouble, jdouble, jdouble, jdouble, jdouble, jdouble);

  #ifdef __cplusplus
}
#endif
#endif
```

Nyní je potřeba nativní metody implementovat. V našem případě budeme implementovat pouze volání již existujících metod. Nejdříve založíme prázdný *C*/*C*++ projekt. Do prázdného projektu přidáme vygenerovaný hlavičkový soubor *shewlib_ShewLib.h*. Nyní je potřeba přidat do projektu cesty na adresáře potřebné pro *JNI* framework, kde se nachází soubor *jni.h* a další důležité soubory. Jedná se o podadresáře *include* a *include/win32* v domácím adresáři *JDK*. V případě, že chceme využít již existující implementaci (to je i náš případ), přidáme do projektu příslušné zdrojové soubory nebo eventuálně nastavíme reference na používané knihovny. V našem případě jde pouze o přidání souboru *predicates.c* [Sh96b].

Nyní přistoupíme k samotné implementaci nativních metod. Přidáme tudíž do projektu další zdrojový soubor, který v našem případě pojmenujeme *ShewLib.c*. Tento soubor bude pouze volat příslušné metody z *predicates.c* (viz následující zdrojový kód).

```
#include "shewlib_ShewLib.h"
#include "predicates.c"
#include <stdio.h>

JNIEXPORT void JNICALL Java_shewlib_ShewLib_exactinit (JNIEnv *env,
jobject obj)
{
 exactinit();
}

JNIEXPORT jdouble JNICALL Java_shewlib_ShewLib_orient2d (JNIEnv *env,
jobject obj, jdouble ax, jdouble ay, jdouble bx, jdouble by, jdouble
cx, jdouble cy)
{
 double d[6];
 d[0] = ax;
 d[1] = ay;
 d[2] = bx;
 d[3] = by;
 d[4] = cx;
 d[5] = cy;
 return orient2d(d, (double*)d+2, (double*)d+4);
}
```

Je potřeba poznamenat, že nativní metoda v javovském kódu mohla mít pouze tři parametry reálných polí znázorňujících body (stejně jako v *predicates.c*), ale vzhledem k tomu, že pole jsou v *Javě* alokovány na haldě, by nutné vytváření těchto polí již v *Javě* značně zpomalilo celý algoritmus. Na rozdíl od *Javy* se v *C/C++* alokují všechny lokální proměnné na zásobníku, tudíž vytvoření předávaných polí až v nativním kódu značně urychlí celý mechanismus volání znaménkového testu.

Dalším krokem je vytvoření a načtení nativní knihovny. Vytvoření je nyní již jen formalitou. Buďto celý projekt přeložíme jako knihovnu přímo v použitém vývojovém prostředí nebo vytvoříme standardní *makefile*.

Máme-li knihovnu přeloženou, je potřeba ji v javovském kódu načíst ještě před samotným použitím nativních metod. To provedeme pomocí statické metody *loadLibrary()* ze třídy *System*. Argumentem této metody je cesta (ať už absolutní či relativní) k načítané knihovně. Cesta nezahrnuje příponu knihovny, protože tu si virtuální stroj *Javy* doplní sám v závislosti na operačním systému (např. pro systémy *MS Windows* automaticky doplňuje *.dll*). Díky tomu je možné částečně vykompenzovat ztrátu přenositelnosti tím, že máme ve stejném adresáři knihovnu přeloženou v různých verzích pro různé platformy. Na základě operačního systému virtuální stroj potom zvolí správnou knihovnu.

Pro samotné zavedení knihovny do paměti se v praxi nejčastěji používá statický blok, protože máme jistotu, že načtení knihovny proběhne právě jednou. Tento způsob načtení však není nutnou podmínkou. V následující ukázce je ale načtení provedené právě pomocí statického bloku.

```
static
{
 System.loadLibrary("lib/ShewLib");
}
```

Nyní již jsou splněné všechny úkony potřebné pro načtení nativní knihovny. Dále už je možné libovolně využívat nativní metody. V našem případě při používání *Shewchukovy knihovny* tedy před samotným voláním znaménkových testů zavoláme nejprve inicializační metodu *exactinit()* (3.2.2).

3.3. Výběr počátečního trojúhelníku

Velmi účinným a přitom jednoduchým způsobem urychlení procházkových algoritmů je opakované hledání počátečního trojúhelníku (dle podkap. 2.13), do kterého se vyplatí investovat i podstatnou část očekávané doby běhu algoritmu. V následující tabulce (*Tab. 3.1*) jsou uvedeny časy algoritmu *Remembering walk* (podkap. 2.3) (v *ms*) potřebné pro nalezení 50000 náhodně vygenerovaných bodů v závislosti na počtu *výběrů počátečního trojúhelníku*. Konkrétní časy nejsou podstatné, podstatné je určení časově optimálního počtu *výběrů počátečního trojúhelníku* pro různé velikosti

triangulace *n*. Tabulka je značně redukovaná (jsou vybrané jen některé hodnoty počtu *výběrů počátečního trojúhelníku*). Navíc každé měření bylo provedeno v mnoha iteracích a v tabulce je uložen průměr ze všech naměřených hodnot. Hodnoty označené jako N/A nebyly změřeny, protože neměly pro následující analýzy význam.

počet bodů triang. (*n*)	**Počet *výběrů počátečního trojúhelníku***															
	0	1	2	3	4	5	6	8	10	15	20	30	40	50	60	70
100	198	177	167	164	165	168	175	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
200	268	227	210	202	199	199	203	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
500	415	339	302	285	271	270	265	267	273	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1000	579	463	406	370	350	339	332	326	329	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2000	789	621	538	489	457	437	427	410	408	425	N/A	N/A	N/A	N/A	N/A	N/A
5000	1252	980	823	745	684	650	621	588	573	567	593	N/A	N/A	N/A	N/A	N/A
10000	1820	1388	1195	1057	975	906	870	815	780	754	761	N/A	N/A	N/A	N/A	N/A
20000	2624	2148	1860	1648	1520	1425	1353	1263	1210	1150	1159	1231	N/A	N/A	N/A	N/A
50000	4760	3976	3366	2990	2733	2541	2380	2208	2068	1924	1868	1923	2062	2242	N/A	N/A
100000	7530	6152	5164	4574	4131	3832	3591	3259	3046	2733	2605	2578	2688	2828	3025	N/A
200000	11065	8865	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3479	3265	3274	3392	3547	N/A
500000	17500	14306	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	4767	4597	4598	4685	7094
1000000	27593	20725	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	6432	6168	6056	5989	6079
2000000	39738	29133	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	8785	8165	7862	7717	7773

Tabulka 3.1 – Časová závislost (v *ms*) *Remembering walk* (podkap. 2.3) na velikosti triangulace *n* a na počtu výběrů počátečního trojúhelníku (viz podkap. 2.13)

Samozřejmě všechny časy uvedené v tabulce (*Tab. 3.1*) v sobě zahrnují i čas potřebný pro *výběr počátečního trojúhelníku*. Sloupec s 0 *výběry počátečního trojúhelníku* obsahuje hodnoty naměřené pro náhodně vybrané počáteční trojúhelníky, zatímco u hodnot v sloupci s 1 *výběrem počátečního trojúhelníku* už probíhá právě jedno porovnání vhodnosti počátečního trojúhelníku (2.13) s dalším náhodně vybraným trojúhelníkem.

Z tabulky lze vyčíst, že vhodným *výběrem počátečního trojúhelníku* lze celý procházkový algoritmus výrazně urychlit. I když tabulka není kompletní, evidentně dokazuje, že na počtu *výběrů počátečního trojúhelníku* velmi záleží.

3.3.1. Volba vhodného počtu výběrů počátečního trojúhelníku

Důležitým krokem je odvození závislosti optimálního počtu počátečních výběrů na velikosti vstupní triangulace *n*. Lze očekávat, že se tato závislost bude odvíjet od způsobu implementace jednotlivých částí. Pokud bude rychlejší (poměrově vhledem k počtu *výběrů počátečního trojúhelníku*), bude optimální počet *výběrů počátečního trojúhelníku* menší. To bude platit i naopak. Stejně tak se dají očekávat odlišné výsledky i u různých typů procházek.

Nyní vyjádříme z tabulky, která obsahuje kompletní výčty počtu *výběrů počátečního trojúhelníku* pro algoritmus *Remembering walk* (obdobná tabulka jako *Tab. 3.1*, ale pro značnou rozsáhlost není zobrazena celá – některé hodnoty tudíž v *Tab. 3.1* nenajdeme), tabulku optimálních počtů *výběrů počátečního trojúhelníku* v závislosti na velikosti vstupu. Následující tabulka (*Tab 3.2*) obsahuje zmíněnou závislost u algoritmu *Remembering walk*.

n	**100**	**200**	**500**	**1K**	**2K**	**5K**	**10K**	**20K**	**50K**	**100K**	**200K**	**500K**	**1M**	**2M**
Optimum	3	5	7	8	10	12	14	16	21	27	32	43	55	66

Tabulka 3.2 – Optimální počet výběrů počátečního trojúhelníku [2.13] pro algoritmus *Remembering walk* v závislosti na velikosti triangulace *n*

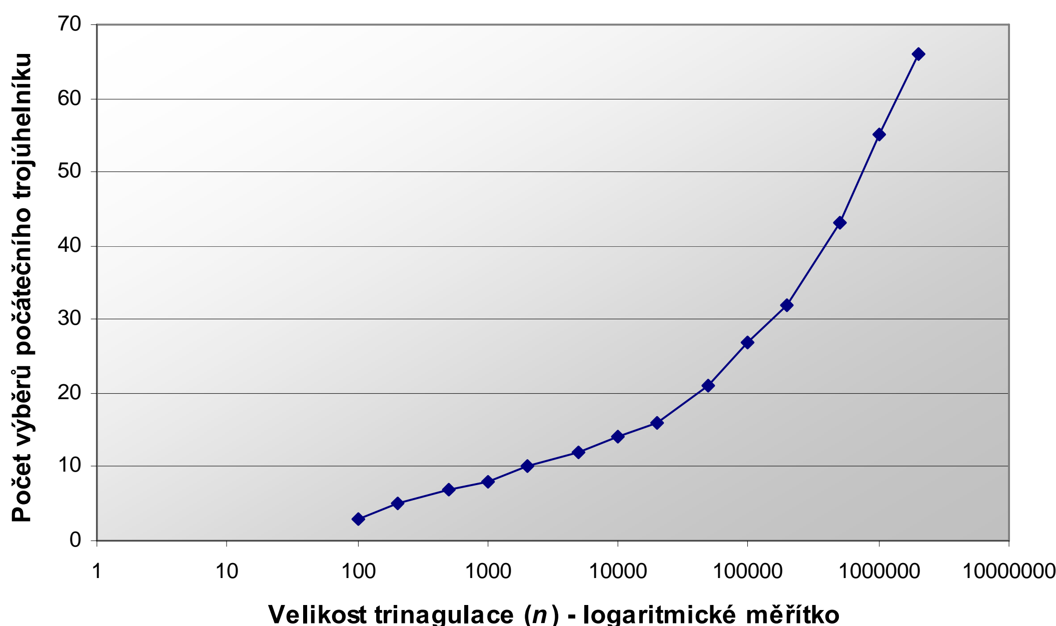
Z tabulky lze říci, že optimální hodnota počtu výběrů závisí přímo úměrně na velikosti triangulace. Navíc je i patrné, že optimální hodnota počtu *výběrů počátečního trojúhelníku* roste mnohem pomaleji než roste velikost triangulace, tudíž je zřejmé, že závislost bude nižší než lineární (sublineární). Nabízí se nám tím samozřejmě logaritmická závislost. Aby byla závislost logaritmická, musí vždy při řádově stejném nárůstu vstupu ekvivalentně narůst i optimální hodnota. Například pokud při řádově stejném nárůstu (například dvojnásobném) u velikosti triangulace neroste optimální hodnota stejným způsobem (o stejnou absolutní hodnotu), není závislost logaritmická.

Jednoduché ověření logaritmické závislosti provedeme u konkrétního algoritmu pomocí grafu, který vytvoříme z tabulky optimálního počtu *výběrů počátečního trojúhelníku* pro daný algoritmus. Ose *X,* udávající velikost triangulace, nastavíme logaritmické měřítko. Pokud graf utvoří přímku, jde prokazatelně o logaritmickou závislost. Následující graf (*Graf 3.1*) sestavený z *Tab. 3.2* charakterizuje závislost optimálního počtu *výběrů počátečního trojúhelníku* na velikosti triangulace *n* u algoritmu *Remembering walk*.

Z grafu je patrné, že závislost je vyšší než logaritmická. Prozkoumáme nyní tempo růstu různých odmocninových výrazů ($\sqrt[2]{n}, \sqrt[3]{n}, \sqrt[4]{n}$). Zjišťujeme, že tempo růstu výrazu $\sqrt[2]{n}$ je příliš rychlé (oproti skutečnému růstu), tempo růstu výrazu $\sqrt[3]{n}$ je stále poměrně rychlé, ale tempo růstu výrazu $\sqrt[4]{n}$ je naproti tomu již pomalejší než ve skutečnosti. Proto se nyní budeme snažit najít koeficient *k* závislosti $\sqrt[k]{n}$, která vystihuje nejlépe zmíněnou závislost. K řešení použijeme doplněk *Microsoft Excel – Řešitel*[45] [Ro97]. Je pochopitelné, že výsledná hodnota *k* bude v intervalu $(3 \quad 4)$. Po drobném zaokrouhlení dostáváme pomocí *MS Excel řešitele* hodnotu *3,5*.

Nyní máme zjištěnou míru závislosti optimálního počtu *výběrů počátečního trojúhelníku* na velikosti vstupní triangulace *n* (zjednodušeně by se to dalo nazvat jako *charakteristické tempo růstu*), nikoliv však přesný optimální

[45] Z anglického “solver”.



Graf 3.1 – Graf závislosti optimálního počtu výběrů počátečního trojúhelníku na velikosti triangulace *n* pro algoritmus *Remembering walk*

počet výběrů pro danou velikost triangulace *n*. Zavedeme proto výraz ve tvaru $a \cdot \sqrt[3{,}5]{n} + b$. Opět pomocí *MS Excel řešitele* vypočítáme hodnoty koeficientů *a* a *b*. Dostáváme výraz $1{,}04 \cdot \sqrt[3{,}5]{n} - 0{,}28$ pomocí něhož u algoritmu *Remembering walk* (po zaokrouhlení na nejbližší celé číslo) určíme optimální počet výběrů pro danou velikost triangulace *n* velice uspokojivě.

Nevyřešenou otázkou stále zůstávají závislosti optimálního počtu počátečních výběrů na velikosti vstupní triangulace u ostatních algoritmů. Dále budeme proto postupovat identicky. Stejným způsobem jako pro *Remembering walk* doplníme tabulku *Tab 3.2* o údaje optimálních počtů výběrů pro ostatní algoritmy. Zmíněné údaje obsahuje následující tabulka (*Tab. 3.3*).

n	100	200	500	1K	2K	5K	10K	20K	50K	100K	200K	500K	1M	2M
Rem. walk	3	5	7	8	10	12	14	16	21	27	32	43	55	66
Flag Fast walk	3	4	7	8	9	11	12	14	18	23	26	36	43	54
Rem. Stoch. Walk	4	6	9	11	15	18	25	31	40	50	62	81	102	122
Flag Fast walk (Stoch.)	3	5	8	9	10	12	16	20	25	31	39	50	57	73
Normal Straight walk	1	2	3	6	8	10	13	15	19	24	32	43	51	64
Orthogonal walk	0	1	2	3	3	5	8	9	13	17	23	31	38	46

Tabulka 3.3 – Optimální počet výběrů počátečního trojúhelníku pro jednotlivé algoritmy v závislosti na velikosti triangulace *n*

Ověříme-li míru závislosti optimálního počtu *výběrů počátečního trojúhelníku* na velikosti vstupní triangulace *n*, kterou jsme spočítali již pro *Remembering walk*, zjišťujeme, že optimální počet výběrů se i pro ostatní algoritmy dá úspěšně určit vzorcem $a \cdot \sqrt[3{,}5]{n} + b$. Dále už jen pomocí

MS Excel řešitele opět dopočítáme koeficienty *a* a *b* pro jednotlivé algoritmy. Následující tabulka (*Tab 3.4*) shrnuje spočtené hodnoty koeficientů.

Algoritmus	Koeficienty	
	a	*b*
Remembering walk	1,04	-0,28
Flag Fast walk	0,82	0,87
Remembering Stochastic Walk	2	-3,27
Flag Fast walk (Stochastic)	1,15	-0,06
Normal Straight walk	1,04	-2,45
Orthogonal walk	0,79	-3,28

Tabulka 3.4 – Optimální hodnoty koeficientů pro jednotlivé algoritmy

3.3.2. Dosažené urychlení

Již z tabulky *Tab. 3.1* je patrné, že opakovaným *výběrem počátečního trojúhelníku* lze dosáhnout velmi výrazného urychlení, především potom při manipulaci s větším objemem dat. Samozřejmě maximální urychlení se odvíjí i od použitého lokačního algoritmu. V následující tabulce (*Tab. 3.5*) je čas bez výběru počátečního trojúhelníku označen jako *B*, čas s optimálním počtem výběrů označen jako *O* a procentuální urychlení je značeno jako *U*.

Algoritmus	*n*=2000			*n*=20000			*n*=200000			*n*=2000000		
	B[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]
Rem. walk	164,9	84,1	49	566,4	241,2	57,4	2582,7	672	74	8450	1625	80,8
Flag Fast walk	156,1	94,2	39,7	498,4	243,6	51,1	2075,3	651,5	68,6	7742	1531	80,2
Rem. Stoch. Walk	218,6	104,8	52,1	778,2	299,8	61,5	3187,4	825,7	74,1	9652	1969,3	79,6
Flag Fast walk (Stoch.)	231,1	126,6	45,2	764,6	313,7	59	2906,2	781,4	73,1	8864,6	1802,6	79,7
Normal Straight walk	127,1	79,3	37,6	435	224,3	48,4	2018,5	608	69,9	6921,8	1462,4	78,9
Orthogonal walk	73,3	57	22,2	263,6	174,1	34	1175,1	527,1	55,1	4578,2	1291,2	71,8

Tabulka 3.5 – Porovnání časů s použitím a bez použití optimálního počtu výběrů počátečního trojúhelníku pro jednotlivé algoritmy (časy lokace 10000 bodů)

Z tabulky (*Tab. 3.5*) je patrné, že není samozřejmě míra urychlení *U[%]*[46] závislá pouze na konkrétním algoritmu, ale závisí i na velikosti vstupní triangulace. Čím vyšší je objem vstupních dat, tím se i vhodným *výběrem počátečního trojúhelníku* urychlí celý lokační algoritmus. Dá se tudíž předpokládat, že se při dalším růstu objemu vstupních dat zvýší i nynější urychlení, které je nyní pro 2000000 bodů u některých algoritmů až pětinásobné ($U = 80\%$).

[46] Míra urychlení *U[%]* udává kolik procent času ušetříme při lokaci s použitím optimálního výběru počátečního trojúhelníku oproti lokaci bez zmíněného vylepšení.

3.3.3. Vliv vstupních dat na výběr počátečního trojúhelníku

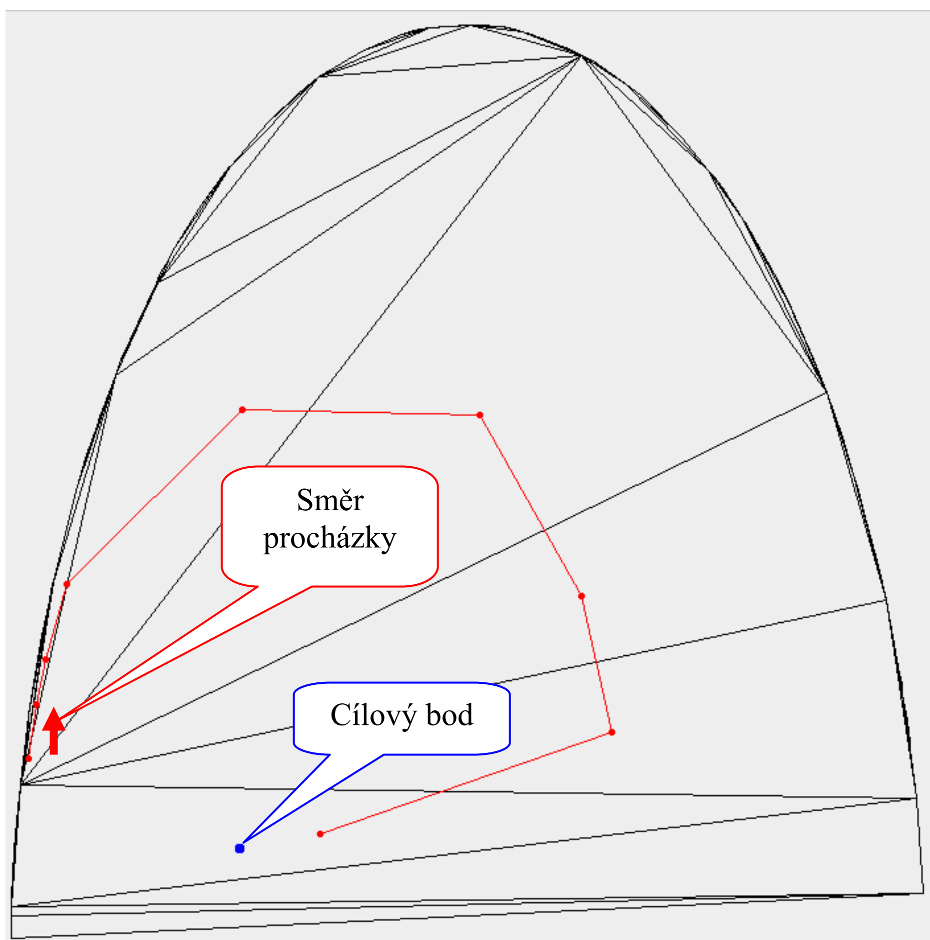
Předchozí výsledky vycházely z pokusů na *Delaunayově triangulaci* s rovnoměrným rozložením náhodně vygenerovaných bodů. Nyní se budeme věnovat analýze výsledků naměřených na triangulacích se specifickými vlastnostmi (půjde opět o *Delaunayovy triangulace* vytvořené nad body, které jsou rozmístěny z určitého úhlu pohledu specificky).

Byly testovány všechny algoritmy z *Tab 3.5* na testovacích datech (body uspořádány do shluků (*clusters*), body rozmístěné ve čtvercové síti (*grid*), body náhodně generované na základě Gaussova rozložení pravděpodobnosti, body rozmístěné na základě reálných dat nebo body umístěné na oblouku). Zpravidla u všech testovaných případů pro všechny algoritmy odpovídaly výsledky rovnoměrnému rozložením bodů. Jedinou výjimku představovaly body rozmístěné na oblouku.

V případě bodů rozmístěných na oblouku totiž případným *výběrem počátečního trojúhelníku* celý algoritmus pouze zpomalujeme. To je způsobeno tím, že u triangulací sestavených z bodů ležících na oblouku platí pravidlo, že většina hledaných bodů leží v několika málo velkých trojúhelnících uvnitř oblouku. Naproti tomu většina trojúhelníků leží na obvodu oblouku (jsou to ale trojúhelníky z hlediska plochy velice malé). Při hledání nejbližšího trojúhelníku s nejvyšší pravděpodobností potom vybereme jako nejbližší trojúhelník právě jeden z malých trojúhelníků na obvodu oblouku (plyne z faktu, že takovýchto trojúhelníků je nejvíce). Algoritmus pak spotřebuje mnoho kroků jen na pouhé opuštění okraje oblouku, který obsahuje množství malých podlouhlých trojúhelníků (*Obr. 3.1*).

Problémem je i to, že když v případě hledání počátečního trojúhelníku testujeme jeden z velkých trojúhelníků (například přímo trojúhelník, který hledaný bod obsahuje), je poměrně pravděpodobné, že bude pro test vzdálenosti vybrán bod (souvisí s charakterem algoritmu pro *výběr počátečního trojúhelníku* [2.13]), který je od cílového bodu poměrně daleko (riziko je patrné z *Obr. 3.1*), a proto nebude ani tento trojúhelník zvolen jako nejbližší. Na následující obrázkové ukázce (*Obr. 3.1*) začíná procházka prováděná algoritmem *Remembering walk* vlevo dole, kde se nachází také cílový bod.

Vzhledem k faktu, že hledání vhodného nejbližšího počátečního trojúhelníku v případě bodů rozmístěných na oblouku algoritmus lokace nikterak neurychlí a vyžaduje určitý čas navíc, je nejlepším řešením tento způsob volby počátečního trojúhelníku vůbec nepoužívat (například vybrat počáteční trojúhelník náhodně nebo v lepším případě zvolit jako počáteční trojúhelník jeden z velkých trojúhelníků uprostřed triangulace).



Obrázek 3.1 – Lokace bodu pomocí algoritmu *Remembering walk* s použitím *výběru počátečního trojúhelníku* [2.13] na triangulaci vzniklé z bodů v oblouku

3.4. Vlastnosti jednotlivých algoritmů

3.4.1. Algoritmy procházek dle viditelnosti

U všech procházkových algoritmů jsou velice důležité vlastnosti, jako je průměrný počet znaménkových testů na jednu lokaci, případně maximální počet znaménkových testů v lokaci nebo průměrná délka procházky (průměrný počet navštívených trojúhelníků), případně maximální délka.

Následující tabulka (*Tab. 3.6*) shrnuje tyto zmíněné vlastnosti u vybraných[47] algoritmů *procházek dle viditelnosti*. Bylo lokalizováno vždy 100000 náhodně vygenerovaných bodů v *Delaunayově triangulaci* vzniklé z rovnoměrně rozložených náhodně vygenerovaných bodů.

[47] Algoritmus *Distance Fast walk* [0] do testů zahrnut nebyl, protože jeho výsledky závisí na volbě parametru kroku. Tomuto algoritmu se bude věnovat jiná část.

Algoritmus	počet bodů triang. (*n*)	Bez *výběru počátečního trojúhelníku*				Optimální počet *výběrů počátečního trojúhelníku*				Testů hran na Δ	
		Testy hran		Délka cesty		Testy hran		Délka cesty		Bez poč. výběru	S opt. poč. výběr.
		Ø počet	max	Ø počet	max	Ø počet	max	Ø počet	max		
Remembering walk	200	31,89	79	17,00	43	17,25	59	8,68	33	1,876	1,988
	2000	91,31	244	51,68	133	33,91	152	18,12	83	1,767	1,871
	20000	291,98	741	166,14	420	73,72	300	40,85	173	1,757	1,805
	200000	902,76	2316	519,01	1349	159,81	852	90,38	491	1,739	1,768
	2000000	2752,30	7020	1587,43	4036	362,23	1356	207,16	801	1,734	1,749
Remembering Stochastic walk	200	30,76	91	16,47	44	15,03	52	7,51	27	1,867	2,000
	2000	89,27	225	50,38	128	27,93	106	14,85	61	1,772	1,881
	20000	280,01	692	161,57	408	54,88	230	30,36	133	1,733	1,808
	200000	859,95	2276	500,40	1348	115,72	476	65,69	272	1,719	1,762
	2000000	2858,53	6870	1668,67	3999	255,02	887	147,01	511	1,713	1,735
Flag Fast walk	200	27,13	61	23,76	52	18,94	47	15,51	38	1,142	1,221
	2000	63,82	193	59,99	135	30,00	93	26,50	86	1,064	1,132
	20000	174,67	587	169,95	419	54,03	246	50,40	197	1,028	1,072
	200000	540,49	1570	535,41	1330	110,89	406	107,13	397	1,009	1,035
	2000000	1663,86	4457	1656,29	4262	237,86	1033	233,85	1030	1,005	1,017
Flag Fast walk (Stochastic)	200	27,74	61	24,34	55	19,17	56	15,83	42	1,139	1,211
	2000	64,01	178	60,41	145	28,71	100	25,27	73	1,060	1,136
	20000	178,21	554	173,62	405	49,99	192	46,29	190	1,026	1,080
	200000	544,20	2095	534,56	1294	96,11	551	91,97	314	1,018	1,045
	2000000	1641,25	5779	1622,75	3979	205,27	1416	199,53	826	1,011	1,029

Tabulka 3.6 – Komplexní testy vlastností *procházek dle viditelnosti*

Testy byly provedeny pro každý algoritmus a datovou sadu dvojí. Nejdříve byl testován příslušný algoritmus bez v*ýběru počátečního trojúhelníku* a poté s optimální hodnotou v*ýběru počátečního trojúhelníku* pro daný algoritmus dle (3.3). Kompletní tabulka testů provedených na *procházkách dle viditelnosti* je k nalezení v příloze (příloha A – *Tab. A.1*).

Velmi zajímavým kritériem pro hodnocení procházkových algoritmů je vztah mezi počtem znaménkových testů a délkou procházky (počet znaménkových testů na jeden trojúhelník). Toto kritérium je zkoumáno v tabulce (*Tab. 3.6*) v posledních dvou sloupcích, kde je uvedeno jak pro měření bez v*ýběru počátečního trojúhelníku*, tak i pro optimální hodnotu *výběru počátečního trojúhelníku* dle (3.3). Z těchto testů pochopitelně vycházejí nejlépe oba algoritmy *Flag Fast walk.*

U všech algoritmů navíc můžeme vidět tendenci poklesu kritéria počtu znaménkových testů na jeden trojúhelník se vzrůstající velikostí triangulace. Původ tohoto jevu je v podstatě velice jednoduchý. Čím blíže jsme totiž cílovému bodu, tím je méně pravděpodobné, že do dalšího trojúhelníku přejdeme již po první testované hraně a že tedy musíme provést další znaménkový test (v případě *Remembering walk* algoritmů). Jelikož v případě použití *výběru počátečního trojúhelníku* je celková délka procházky kratší, pochopitelně vychází v průměru i více hranových testů na trojúhelník. V případě *Flag Fast walk* algoritmů se samozřejmě v první fázi provádí vždy

jeden znaménkový test na trojúhelník, ale v druhé fázi je na dohledání použit algoritmus *Remembering (Stochastic) walk*. Toto finální dohledání je ale relativně krátké a proto se v kritériu projeví se vzrůstajícím rozměrem dat čím dál méně. Počet znaménkových testů na jeden trojúhelník v případě *Flag Fast walk* algoritmů potom konverguje postupně k jednomu znaménkovému testu na trojúhelník.

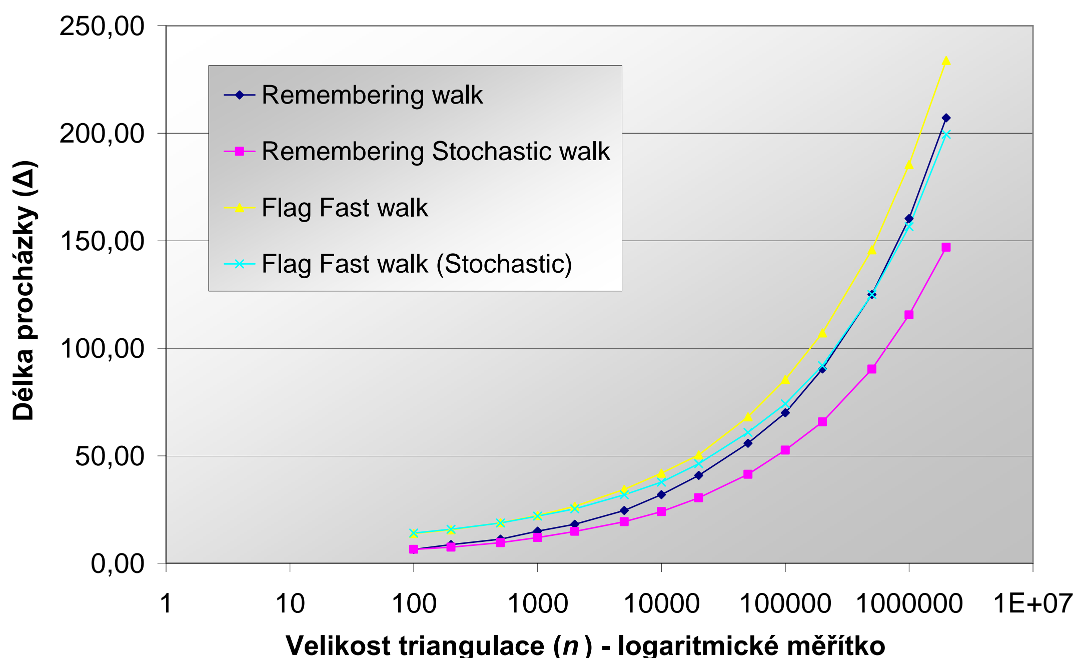
Vrátíme-li se k hodnotám kritéria u *Remembering walk* algoritmů, zjišťujeme, že neplatí původní předpoklad, že se provede v průměru 1,5 znaménkového testu na trojúhelník. Vzhledem k tomu, že tento předpoklad neplatí, je nutné říci, že potenciální zlepšení všech *Fast walk* algoritmů je větší než v [Ko06] a v podkapitole 2.5 odhadovaných 50%. Lze totiž mluvit o potenciálním zlepšení od 70% (pro objemné triangulace v řádu miliónů bodů) až téměř po 100% (pro velmi malé triangulace).

Nyní se zaměříme na odhad časové složitosti algoritmů. V dalších odhadech nebudeme uvažovat singulární případy a budeme počítat pouze očekávanou složitost v běžných případech. Nejdříve se zaměříme na odhad očekávané složitosti algoritmů bez použití *výběru počátečního trojúhelníku*. Na první pohled je z tabulky (*Tab. 3.6*, respektive *Tab A.1*) patrné, že v tomto případě roste délka procházky poměrně rychle a nelze tudíž uvažovat o potenciální logaritmické složitosti, proto budeme předpokládat aproximující sublineární funkci ve tvaru $a \cdot \sqrt[b]{n}$, kde *b* bude z intervalu $(1 \quad \infty)$. Na vyřešení tohoto problému použijeme doplněk *Microsoft Excel – Řešitel*, kde se budeme snažit minimalizovat absolutní rozdíl vypočítané a reálné délky procházky (u všech dostupných velikostí triangulací (*Tab. A.1*)) pomocí změn koeficientů *a* a *b*. Výpočet aplikujeme na všechny zmíněné 4 algoritmy v tabulce, i když vzhledem k podobným hodnotám je očekáván i podobný výsledek. Navíc je třeba počítat i s tím, že *Flag Fast walk* algoritmy udělají vždy několik kroků navíc způsobených tím, že přejdou cíl. Počet těchto kroků nevíc není nijak závislý na velikosti triangulace (pomineme-li fakt, že na větších triangulacích můžou nastat extrémní případy, které ale zanedbáme). Testováním bylo zjištěno, že se v průměru navštíví *Flag Fast walk* algoritmy 6,5 trojúhelníku navíc. Z tohoto důvodu byl upraven vzorec aproximující funkce pro očekávanou složitost těchto algoritmů na $a \cdot \sqrt[b]{n} + 6{,}5$. Koeficient *a* již dále nebudeme uvažovat, protože pro očekávanou složitost není směrodatný. V následující tabulce (*Tab. 3.7*) jsou uvedeny vypočtené koeficienty *b*.

Algoritmus	**Remembering walk**	**Remembering Stochastic walk**	**Flag Fast walk**	**Flag Fast walk (Stochastic)**
b	2,024520865	1,976263058	2,009264607	2,015021238

Tabulka 3.7 – Hodnoty koeficientu *b* (bez *Výběru poč. troj.*)

Vzhledem k tomu, že koeficienty se příliš neliší, lze říci, že odhadovaná časová složitost všech zkoumaných *procházek dle viditelnosti* je přibližně $O\left(\sqrt[2]{n}\right)$. Nyní se budeme zabývat odhadem časové složitosti u všech algoritmů s použitím optimální hodnoty *výběru počátečního trojúhelníku*. Předem tedy již máme omezenou výslednou časovou složitost zdola složitostí $O\left(\sqrt[3,5]{n}\right)$, která je potřebná pro *výběr počátečního trojúhelníku*. Nyní budeme zjišťovat časovou složitost samotné lokace. Již tabulka (*Tab. 3.6*, respektive *Tab A.1*) nám napoví, že pravděpodobně nepůjde o logaritmickou složitost. To dokážeme i grafem (*Graf 3.2*), kde bylo pro osu X použito logaritmické měřítko.



Graf 3.2 – Graf závislosti počtu testů hran na jeden trojúhelník na velikosti triangulace *n* pro algoritmy *procházky dle viditelnosti*

Je jasné, že pro logaritmickou závislost by musel graf konkrétního algoritmu utvořit přímku, což v našem případě neutvoří zobrazení výsledků ani jednoho algoritmu, z čehož plyne, že závislost logaritmická není. Budeme proto předpokládat sublineární složitost ve tvaru $a \cdot \sqrt[b]{n}$, kde b bude z intervalu $(1 \quad \infty)$. Na vyřešení tohoto problému použijeme opět doplněk *Microsoft Excel – Řešitel*. Stejně jako v předchozím případě se budeme snažit minimalizovat absolutní rozdíl vypočítané a reálné délky procházky (u všech dostupných velikostí triangulací (*Tab. A.1*)) pomocí změn koeficientů a a b. Samozřejmě také stejně jako v předchozím případě platí, že *Flag Fast walk* algoritmy udělají vždy několik kroků navíc způsobených tím, že přejdou cíl. Proto pro výpočet jejich časové složitosti bude opět použit mírně upravený vzorec $a \cdot \sqrt[b]{n} + 6{,}5$. Navíc stejně tak jako v předchozím případě koeficient a již dále nebudeme uvažovat, protože pro očekávanou složitost není směrodatný. V následující

tabulce (*Tab. 3.8*) jsou uvedeny vypočtené koeficienty b pro jednotlivé algoritmy.

Algoritmus	Remembering walk	Remembering Stochastic walk	Flag Fast walk	Flag Fast walk (Stochastic)
b	2,840343269	2,928940205	2,833718625	2,952410619

Tabulka 3.8 – Hodnoty koeficientu b (s optimálním *Výběrem poč. troj.*)

Při pohledu na výsledky zjišťujeme, že na rozdíl od varianty bez použití *výběru počátečního trojúhelníku* vycházejí o něco lépe stochastické verze algoritmů. Při pohledu na výsledky navíc zjišťujeme, že se výsledná očekávaná složitost pohybuje kdesi v intervalu $\left(O\left(\sqrt[3]{n}\right)\quad O\left(\sqrt[2,8]{n}\right)\right)$. Navíc obecně pro různé procházky dle viditelnosti (nezávisle na *výběru počátečního trojúhelníku*) by se dalo říct, že očekávaná složitost je v intervalu $\left(O\left(\sqrt[3]{n}\right)\quad O\left(\sqrt[2]{n}\right)\right)$, což také uvádí mimo jiné prameny [De02] a [Ko06].

Je potřeba dodat, že všechny 4 testované algoritmy byly otestovány i na dalších vstupních datech (Delaunayovy triangulace vzniklé z bodů uspořádaných do shluků (*clusters*), bodů rozmístěných ve čtvercové síti (*grid*), bodů náhodně generovaných na základě Gaussova rozložení pravděpodobnosti, bodů rozmístěných na základě reálných dat nebo bodů umístěných na oblouku). Většina výsledků byla velmi podobná výsledkům na náhodně rozmístěných datech, proto není dále nikterak zmiňována. Výjimku představovaly pouze data rozmístěná do oblouku. Chování procházkových algoritmů v tomto případě je zcela specifické a jeho analýza by byla v konečném důsledku velice rozsáhlá. Vzhledem k tomu, že takováto specifická data nemají příliš časté a široké uplatnění v souvislosti s lokací bodů a triangulacemi vůbec, nebyla zmíněná analýza problému v této práci provedena.

3.4.2. Distance Fast walk

Analýza tohoto algoritmu je poněkud komplikovanější problém. Vše je způsobeno nutností vstupního parametru (parametr kroku, po kterém se kontroluje vzdálenost od cílového bodu). Problém je v tom, že v podstatě nelze určit optimální kombinaci *výběru počátečního trojúhelníku* a výběru vstupního parametru, protože obojí se vzájemně silně ovlivňuje. Pro změření časů zobrazených v následující tabulce (*Tab. 3.9*) byla zvolena jako nástřel hodnota právě $\sqrt[3,5]{n}$ *výběrů počátečního trojúhelníku*. Hodnoty označené jako *N/A* nebyly změřeny, protože pro určení optima neměly význam.

Z tabulky je patrné, že se optimální krok algoritmu (krok, po kterém je kontrolována vzdálenost) začíná zvyšovat až pro větší objemy dat. Testováním bylo zjištěno, že v případě bez použití *výběru počátečního trojúhelníku* se optimální velikost kroku odvíjí úplně jiným směrem (roste mnohem rychleji),

ale vzhledem k tomu, že nás zajímá především nejrychlejší konfigurace algoritmu, nebudeme vůbec tuto variantu zkoumat. Nyní se pokusíme odvodit funkce závislosti parametru kroku na velikosti vstupních dat, která nám určí optimální velikost kroku pro jednotlivé *Distance Fast walk* algoritmy. Postupovat budeme podobně jako v [3.3] a funkci se pokusíme najít pomocí doplňku *Microsoft Excel – Řešitele*. Budeme hledat funkce ve tvaru $a \cdot \sqrt[b]{n} + c$. Bylo zjištěno, že optimální hodnoty kroku jsou celočíselné hodnoty vzniklé zaokrouhlením funkce $0{,}34 \cdot \sqrt[4{,}58]{n} + 0{,}56$ pro nestochastickou verzi a $0{,}23 \cdot \sqrt[4{,}90]{n} + 0{,}68$ pro stochastickou verzi algoritmu.

***n* (Distance Fast walk)**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
100	187	188	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
200	203	218	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
500	266	282	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1000	344	327	391	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2000	422	390	421	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5000	562	562	578	577	N/A	N/A	N/A	N/A	N/A	N/A	N/A
10000	749	720	719	782	N/A	N/A	N/A	N/A	N/A	N/A	N/A
20000	1125	1124	1123	1141	1172	N/A	N/A	N/A	N/A	N/A	N/A
50000	1906	1875	1875	1876	1890	1906	N/A	N/A	N/A	N/A	N/A
100000	2547	2515	2531	2500	2500	2515	N/A	N/A	N/A	N/A	N/A
200000	3250	3266	3236	3229	3224	3235	3249	N/A	N/A	N/A	N/A
500000	4625	4610	4577	4532	4517	4515	4517	4531	N/A	N/A	N/A
1000000	6047	6000	5939	5859	5812	5844	5796	5813	5890	N/A	N/A
2000000	7921	7874	7812	7766	7671	7657	7640	7609	7688	7719	7718
***n* (Distance Fast Stochastic walk)**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
100	214	219	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
200	235	282	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
500	313	343	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1000	406	453	437	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2000	501	515	563	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5000	703	735	781	812	N/A	N/A	N/A	N/A	N/A	N/A	N/A
10000	1250	1233	1297	1359	N/A	N/A	N/A	N/A	N/A	N/A	N/A
20000	1564	1546	1608	1688	1750	N/A	N/A	N/A	N/A	N/A	N/A
50000	2032	2031	2046	2078	2125	2141	N/A	N/A	N/A	N/A	N/A
100000	2766	2734	2749	2782	2859	2859	N/A	N/A	N/A	N/A	N/A
200000	3611	3640	3609	3642	3656	3735	3718	N/A	N/A	N/A	N/A
500000	5077	5094	5077	5046	5110	5110	5141	5156	N/A	N/A	N/A
1000000	6609	6624	6563	6557	6559	6577	6596	6656	6672	N/A	N/A
2000000	8579	8625	8609	8563	8485	8531	8594	8561	8610	8657	8689

Tabulka 3.9 – Časy lokace 50.000 bodů [ms] algoritmem *Distance Fast walk* pro různé hodnoty kroku algoritmu a volbu $\sqrt[3{,}5]{n}$ *Výběrů počátečního trojúhelníku*

Nevýhodou těchto algoritmů je, že nejsou příliš dobře aplikovatelné na některé typy vstupních dat (na rozdíl od předchozích algortimů *procházky dle viditelnosti*). Jde především o data, která mají dlouhé a úzké trojúhelníky (shluky, reálná data, oblouk), protože test vzdálenosti vůči jednomu z bodů trojúhelníku může podávat velice zkreslující údaje. Z tohoto důvodu již dále nebudou algoritmy *Distance Fast walk*, respektive optimální poměr *výběru počátečního trojúhelníku* a kroku algoritmu, zkoumány.

3.4.3. Procházky přímé

Stejně jako v podkapitole 3.4.2 je i u algoritmu *Distance Straight walk* (podkap. 2.10) určení optimálního poměru *výběru počátečního trojúhelníku* a kroku algoritmu velmi koplikovaným úkolem. Proto pro zjednodušení předem určíme hodnotu optimálního počtu *výběrů počátečního trojúhelníku* na $\sqrt[3{,}5]{n}$, u které budeme předpokládat blízkost optimální hodnotě, a dále se pokusíme z naměřených výsledků odhadnout optimální hodnotu kroku kontroly vzdálenosti pro tento způsob *výběru počátečního trojúhelníku*. Následující tabulka (*Tab. 3.10*) obsahuje časy lokace 50000 bodů algoritmem *Distance Straight walk* s $\sqrt[3{,}5]{n}$ *výběry počátečního trojúhelníku*. Pro konečné dohledání bodu byla použita stochastická varianta algoritmu *Remembering walk*.

n **(Distance Straight walk)**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
100	188	203	233	265	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
200	250	250	297	312	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
500	312	313	312	344	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1000	391	359	359	423	438	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
2000	515	454	453	468	469	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5000	656	609	562	563	688	703	N/A	N/A	N/A	N/A	N/A	N/A	N/A
10000	1031	921	765	734	797	812	N/A	N/A	N/A	N/A	N/A	N/A	N/A
20000	1343	1266	1155	1108	1157	1250	1266	N/A	N/A	N/A	N/A	N/A	N/A
50000	2109	2047	1890	1845	1875	1906	1905	1999	N/A	N/A	N/A	N/A	N/A
100000	2952	2843	2640	2531	2547	2593	2641	2703	N/A	N/A	N/A	N/A	N/A
200000	3750	3593	3390	3249	3297	3219	3250	3297	3328	N/A	N/A	N/A	N/A
500000	5360	5218	4906	4485	4453	4547	4469	4454	4531	4609	N/A	N/A	N/A
1000000	6842	6782	6454	5906	5672	5766	5719	5750	5813	5796	5891	N/A	N/A
2000000	9124	8907	8578	7860	7515	7390	7390	7437	7500	7547	7593	7641	7703

Tabulka 3.10 – Časy lokace 50000 bodů [ms] algoritmem *Distance Straight walk* pro různé hodnoty kroku algoritmu a volbu $\sqrt[3{,}5]{n}$ *výběrů počátečního trojúhelníku*

Nyní se pokusíme odvodit funkce závislosti parametru kroku na velikosti vstupních dat, která nám určí optimální velikost kroku pro algoritmus *Distance Straight walk*. Postupovat budeme podobně jako v podkapitolách 3.3 a 3.4.2. Výslednou funkci se pokusíme najít pomocí doplňku *Microsoft Excel – Řešitele*. Budeme hledat funkci ve tvaru $a \cdot \sqrt[b]{n} + c$. Zjištěná funkce má pak tvar

$0{,}92 \cdot \sqrt[7{,}7]{n} + 0{,}42$. Předpokládáme, že tato funkce charakterizuje optimální volbu parametru kroku, i když ve skutečnosti vystihuje pouze optimální volbu kroku pro $\sqrt[3{,}5]{n}$ *výběrů počátečního trojúhelníku*. Vzhledem k tomu, že určení optimálního poměru *výběru počátečního trojúhelníku* a kroku algoritmu je velmi obtížné, budeme dále pokládat zvolenou konfiguraci za optimální, protože máme velmi dobrý důvod se domnívat, že bude svými vlastnostmi optimálním hodnotám velice blízko.

Následující tabulka (*Tab. 3.11*) zobrazuje přímé porovnání obou algoritmů *přímé procházky*. Tabulka zobrazuje jak průměrnou a maximální délku cesty přímé procházky, tak i průměrnou a maximální délku cesty Remembering walk algoritmu, kterým provede finální dohledání. Pro oba algoritmy bylo zvoleno optimální nastavení (*Počet výběrů počátečního trojúhelníku* a případně parametr kroku u *Distance Fast walk*).

Algoritmus	Normal Straight walk				Distance Straight walk			
počet bodů triang. (*n*)	Straight walk		Remembering Stochastic walk		Straight walk		Remembering Stochastic walk	
	Ø počet	max	Ø počet	max	Ø počet	max	Ø počet	max
100	8,12	30	1,682	8	6,14	24	3,195	21
200	9,61	38	1,743	15	6,87	28	3,526	29
500	11,85	51	1,756	8	11,50	45	4,284	37
1000	15,16	69	1,751	8	13,88	60	4,706	58
2000	18,30	88	1,761	17	16,13	69	5,365	66
5000	25,56	114	1,763	9	20,56	114	7,467	97
10000	31,40	147	1,761	9	30,74	148	6,239	99
20000	39,67	208	1,763	7	37,65	184	7,303	167
50000	54,31	254	1,765	8	54,89	295	6,989	241
100000	68,50	343	1,762	8	68,37	335	7,630	244
200000	85,75	423	1,764	9	85,45	385	8,906	330
500000	117,78	589	1,765	8	122,29	546	8,163	380
1000000	150,97	670	1,764	8	152,95	720	8,872	635
2000000	192,14	937	1,767	189	198,25	1029	8,735	696

Tabulka 3.11 – Komplexní testy vlastností *přímých procházek*

Údaje v tabulce byly získány lokací 100000 bodů a zprůměrňovány (samozřejmě neplatí v případě maximální délky procházky). Z tabulky lze vyčíst, že mnohem přesnější je ukončení u algoritmu *Normal Straight walk*, kdy k dohledání stačí pouze zkontrolovat libovolným[48] *Remembering walk* algoritmem v průměru 1,76 trojúhelníků, kde navíc první trojúhelník se musí zkontrolovat vždy, protože jde o trojúhelník, ve kterém skončila lokace *Normal Straight walk* algoritmem. Porovnáme-li celkové doby procházek, vycházejí

[48] Vzhledem k tomu, že tato procházka je už velice krátká, vyplatí se použít *Remembering Stochastic walk* algoritmus s použitím knihovny *Shewchukovy knihovny*. A tedy máme jistotu, že nedojde k zacyklení, zaplacenou minimální cenou z celkové doby běhu algoritmu.

velice podobně, i když *Distance Straight walk* potřebuje více kroků na konečné dohledání. To je dáno tím, že lokace *Distance Straight walk* často končí dříve nebo naopak hledaný bod přejdou, ale v průměru pak vyjdou v první fázi podobně dlouhé.

Nyní se zaměříme na odhad časové složitosti algoritmů. V dalších odhadech nebudeme uvažovat singulární případy a budeme počítat pouze očekávanou složitost v běžných případech. Na první pohled je z tabulky (*Tab. 3.11*) patrné, že v tomto případě roste délka procházky poměrně rychle a nelze tudíž uvažovat o potenciální logaritmické složitosti, proto budeme předpokládat aproximující sublineární funkci ve tvaru $a \cdot \sqrt[b]{n}$, kde b bude z intervalu $(1 \quad \infty)$. Na vyřešení tohoto problému použijeme doplněk *Microsoft Excel – Řešitel*, kde se budeme snažit minimalizovat absolutní rozdíl vypočítané a reálné délky procházky pomocí změn koeficientů a a b.

K výpočtu složitosti byla pochopitelně využita pouze data *Straight walk* lokace, protože druhá část lokace je zanedbatelná (vzhledem k tomu, že roste pomaleji). Popsaným způsobem byla odhadnuta přibližná časová složitost u *Normal Straight walk* algoritmu na $O\left(\sqrt[2,93]{n}\right)$ a u *Distance Straight walk* algoritmu na $O\left(\sqrt[2,82]{n}\right)$. Takže oba zkoumané procházkové algoritmy spadají délkou procházky do intervalu $\left(O\left(\sqrt[3]{n}\right) \quad O\left(\sqrt[2,8]{n}\right)\right)$ - do stejného intervalu jako *procházky dle viditelnosti*.

Algoritmy byly opět testovány na všech dalších dostupných typech dat (Delaunayovy triangulace vzniklé z bodů uspořádaných do shluků (*clusters*), bodů rozmístěných ve čtvercové síti (*grid*), bodů náhodně generovaných na základě Gaussova rozložení pravděpodobnosti, bodů rozmístěných na základě reálných dat nebo bodů umístěných na oblouku). U *Normal Straight walk* algoritmu byly výsledky velmi podobné na všech typech dat (kromě oblouku, kde algoritmus fungoval spolehlivě, ale zhodnocení výsledků dosažených na oblouku by bylo na velmi rozsáhlý rozbor, protože výsledky jsou velice specifické). U *Distance Straight walk* algoritmu nastal stejný problém jako je popsaný v [3.4.2]. V případě dlouhých a úzkých trojúhelníků (clustery, reálná data, oblouk) byl často rozpoznán konec pozdě nebo příliš brzy.

3.4.4. Pravoúhlá procházka

Samozřejmě je evidentní, že *pravoúhlá procházka* bude delší než ostatní procházky. V následující tabulce (*Tab. 3.12*) je nastíněná průměrná a maximální délka *pravoúhlé procházky* s optimální hodnotou *výběru počátečního trojúhelníku* a samozřejmě i průměrná a maximální délka

následného dohledání trojúhelníku pomocí zvoleného[49] *Remembering walk* algoritmu. Hodnoty byly naměřeny lokací 100000 bodů.

počet bodů triang. (*n*)	**Orthogonal walk**		**Remembering Stochastic walk**		**počet bodů triang. (*n*)**	**Orthogonal walk**		**Remembering Stochastic walk**	
	Ø počet	**max**	**Ø počet**	**max**		**Ø počet**	**max**	**Ø počet**	**max**
100	13,13	38	2,269	30	20000	60,97	319	2,189	102
200	18,82	56	2,361	42	50000	81,96	478	2,181	143
500	23,14	85	2,239	55	100000	102,38	572	2,183	203
1000	27,04	114	2,153	71	200000	128,47	735	2,178	120
2000	29,49	132	2,149	58	500000	177,47	882	2,180	160
5000	39,12	180	2,164	108	1000000	222,48	1142	2,185	188
10000	48,22	241	2,173	79	2000000	282,95	1349	2,193	263

Tabulka 3.12 – Komplexní testy vlastností *pravoúhlé procházky*

Na výsledcích je patrné, že následné dohledání je v průměru vždy podobně krátké v intervalu $\left(2{,}14 \quad 2{,}37\right)$ téměř nezávislé na velikosti vstupních dat. Opět se zaměříme na odhad časové složitosti algoritmů. V dalších odhadech nebudeme uvažovat singulární případy a budeme počítat pouze očekávanou složitost v běžných případech. Na první pohled je z tabulky (*Tab. 3.12*) patrné, že v tomto případě roste délka procházky poměrně rychle a nelze tudíž uvažovat o potenciální logaritmické složitosti, proto budeme předpokládat aproximující sublineární funkci ve tvaru $a \cdot \sqrt[b]{n}$, kde *b* bude z intervalu $\left(1 \quad \infty\right)$. Na vyřešení tohoto problému použijeme doplněk *Microsoft Excel – Řešitel*, kde se budeme snažit minimalizovat absolutní rozdíl vypočítané a reálné délky procházky pomocí změn koeficientů *a* a *b*.

Popsaným způsobem byla odhadnuta přibližná časová složitost na $O\left(\sqrt[3{,}01]{n}\right)$, což jen mírně vybočuje z původně určeného intervalu složitosti procházek $\left(O\left(\sqrt[3]{n}\right) \quad O\left(\sqrt[2{,}8]{n}\right)\right)$.

I tento algoritmus byl testován na všech dalších dostupných typech dat (Delaunayovy triangulace vzniklé z bodů uspořádaných do shluků (*clusters*), bodů rozmístěných ve čtvercové síti (*grid*), bodů náhodně generovaných na základě Gaussova rozložení pravděpodobnosti, bodů rozmístěných na základě reálných dat nebo bodů umístěných na oblouku). Výsledky algoritmu byly velmi podobné na všech typech dat (kromě oblouku, kde algoritmus fungoval spolehlivě, ale zhodnocení výsledků dosažených na oblouku by bylo na velmi rozsáhlý rozbor, protože výsledky jsou velice specifické). V případě triangulací, kde se předpokládají neobvyklé tvary (triangulace, kde je v průběhu možné z triangulace vystoupit (viz *Obr. 2.10* a *Obr. 2.11*)), je však potřeba algoritmus

[49] Vzhledem k tomu, že tato procházka je už velice krátká, vyplatí se použít *Remembering Stochastic walk* algoritmus s použitím knihovny *Shewchukovy knihovny*. Tím pádem máme jistotu, že nedojde k zacyklení, zaplacenou minimální cenou z celkové doby běhu algoritmu.

modifikovat tak, aby se pravoúhlá procházka použila opakovaně (platí například často pro reálná data, ale i pro oblouk).

Algoritmus se ukázal velmi spolehlivý. Obrovskou výhodou je zde i možnost využití externí *Shewchukovy knihovny* na krátké finální dohledání. Díky tomu, že je finální dohledání tak krátké, se zpomalení prakticky neprojeví, ale odstraníme tím riziko možného zacyklení.

3.4.5. Porovnání jednotlivých algoritmů

Následující tabulka (*Tab. 3.12*) obsahuje vzájemné porovnání časů lokace pomocí všech probíraných algoritmů. Pro všechny algoritmy bylo voleno optimální nastavení *výběru počátečního trojúhelníku*, případně dalšího vstupního parametru. V případě, že optimální nastavení nebylo známo, bylo voleno takové nastavení, které lze považovat za téměř optimální. Pro kvalitnější výsledky byly změřeny časy lokace 500000 bodů. Testování probíhalo na počítači s parametry uvedenými v podkapitole 3.1.4.

počet bodů triangulace (*n*)	**100**	**500**	**1000**	**5000**	**10000**	**50000**	**100000**	**500000**	**1000000**	**2000000**
Rem. walk	1923	2685	3342	5843	7797	19344	26188	48673	63593	85515
Rem. Stoch. walk	2300	3423	4267	8543	13797	24794	32532	60013	77468	99704
Rem. Stoch. walk[50]	2812	4499	5577	11497	16253	28657	36252	66107	85064	110251
Flag Fast walk	2626	3765	4249	8499	12608	20918	25952	47235	60452	80376
Flag Fast walk (Stoch.)	3248	4720	5405	10422	14484	23719	30216	53172	68638	89234
Distance Fast walk	2143	3155	3750	8313	8749	20205	25047	47703	62653	80047
Dist. Fast walk (Stoch.)	2202	3331	4221	9251	12543	23609	29936	54125	70158	91451
Distance Straight walk	2111	3328	3905	6763	9797	19470	24235	45547	58641	76671
Normal Straight walk	2076	3001	3658	6217	9170	18406	21812	42564	55386	73716
Orthogonal walk	1576	2280	2465	4485	5013	14577	19812	37034	48455	63596

Tabulka 3.12 – Časová závislost (v *ms*) jednotlivých algoritmů na velikosti triangulace *n*

Hodnoty byly naměřené na *Delaunayově triangulaci* s rovnoměrným rozložením náhodně vygenerovaných bodů. Samozřejmě triangulace jiná než *Delaunayova* vylučuje použití nestochastických verzí algoritmů (v našem případě *Remembering walk*, *Flag Fast walk* a *Distance Fast walk*). Algoritmy s kontrolou vzdálenosti pak mají horší výsledky na datech, v kterých se vyskytují dlouhé a úzké trojúhelníky (clustery, reálná data, oblouk). Algoritmy *Remembering Stochastic walk* s *adaptivním robustním znaménkovým testem*, *Normal Straight walk* a *Orthogonal walk* jsou pak plně odolné vůči zacyklení, protože používají k finálnímu dohledání *Shewchukovu knihovnu* na *adaptivní*

[50] S použitím externí *Shewchukovy knihovny* na *adaptivní robustní znaménkový test.*

robustní znaménkový test (*Remembering Stochastic walk* jí používá na celou procházku).

Globálně vzato ze všech testů vychází nejlépe *procházka pravoúhlá* (*Orthogonal walk*). Je podstatně rychlejší a navíc plně spolehlivá. Na pomyslném druhém místě by se rychlostí i spolehlivostí umístil algoritmus *Normal Straight walk*, který je o něco jednoduší na implementaci než *procházka pravoúhlá* (především pak, pokud se v *pravoúhlé procházce* musí řešit opětovná aplikace na datech, kdy se při procházení *pravoúhlou procházkou* opětovně opouští datový set (viz *Obr. 2.10* a *Obr. 2.11*)). Nesmíme však zapomenout na implementačně nejjednodušší, ale nejdůležitější, algoritmus, který je základním kamenem všech zmíněných algoritmů – algoritmus *Remembering (Stochastic) walk*, bez kterého se v žádném uváděném procházkovém algoritmu neobejdeme.

Byla kontrolována i odhadovaná očekávaná složitost v souvislosti se změřenými časy a zjistilo se, že odhad je poměrně přesný. Dalším poznatkem bylo, že složitost v našich případech není zase tolik směrodatná, protože v exponentu jsou čísla relativně blízká a ve výsledném čase se daleko více projeví samotná násobící konstanta (ve většině předchozích příkladů značená jako *a*) a rychlost zvládnutí jednotlivých kroků. To vede k myšlence, že by do určitého objemu dat (možná i dosti vysokého) mohly z hlediska času procházkové algoritmy (s očekávanou časovou složitostí v intervalu $\left(O\left(\sqrt[3,05]{n}\right) \quad O\left(\sqrt[2,8]{n}\right)\right)$) úspěšně konkurovat i algoritmům s logaritmickou časovou složitostí (algoritmy, které využívají sofistikované datové struktury), které mají sice logaritmickou složitost, ale cena jednotlivých operací je poměrně vysoká (operace jsou poměrně pomalé).

3.5. Praktická aplikace

3.5.1. Dynamická Delaunayova triangulace pro kinetická data

Aplikace Deluanayovy triangulace na kinetická data umožňuje významné zjednodušení některých operací prováděných nad těmito daty. Zejména je potom důležitá skutečnost, že využitím Delaunayovy triangulace lze snížit algoritmickou složitost problému detekce kolizí v rámci množiny pohybujících se bodů.

Vzhledem k tomu, že pohyblivost bodů si neodvratně vynutí topologické změny v triangulaci (chceme-li zachovat kritérium Delaunayovy triangulace pro každý časový okamžik, abychom ji mohli využít pro detekci kolizí), je třeba detekovat časy, kdy k těmto změnám dochází a triangulaci podle toho upravovat. Aby bylo možné tohoto dosáhnout, je třeba vědět, jakým způsobem se body pohybují a jakým způsobem bude na jejich pohyb triangulace reagovat.

Přidávání bodů do triangulace je vyřešeno elegantně, pomocí konstrukce triangulace algoritmem inkrementálního vkládání, který tento problém sám o

sobě řeší. Na druhé straně, odstraňování bodů z triangulace je potřeba řešit sofistikovanějšími postupy. V souběžně zpracovávané diplomové práci je autorem Tomášem Vomáčkou zvolen postup, který odstraní bod z triangulace a vzniklý mnohoúhelník nahradí množinou Delaunayových trojúhelníků, které tvoří uši tohoto mnohoúhelníku a mnohoúhelníků vzniklých postupným odebíráním jeho vrcholů v důsledku retriangulizace. Je potřeba zmínit, že právě použití procházkového algoritmu odebírání bodů velice zjednodušuje, protože údržba sofistikované datové struktury pro odebírání bodů bývá zpravidla velmi komplikovaná.

[Vo07a, Vo07b, Vo08a, Vo08b]

Lokace bodů pomocí procházek byly v této práci používány na konstrukci Delaunayovy triangulace inkrementálním vkládáním a na určení trojúhelníku, v kterém se nachází odstraňovaný bod. Autorem pak byly poskytnuty výsledky (viz *Tab. 3.13*) časů tvorby Delaunayovy triangulace s lokací provedenou původním algoritmem (*Remembering Stochastic walk*) a poté doporučeným algoritmem (*Orthogonal walk*).

vrcholů	Remembering Stochastic walk	Orthogonal walk
1000	125	94
5000	1375	1219
10000	4563	4016
50000	122844	113031
100000	680859	626734

Tabulka 3.13 – Doba konstrukce Delaunayovy triangulace algoritmem inkrementálního vkládání (v *ms*)

Na první pohled přinesl algoritmus *Orthogonal walk* v této praktické aplikaci znatelné zrychlení. Podivné jsou příliš vysoké časy, což může být způsobené jednak slabou hardwarovou konfigurací nebo i ne příliš efektivní implementací, například bylo použito $\sqrt[3,5]{n}$ *výběrů počátečního trojúhelníku*, kdy bylo $\sqrt[3,5]{n}$ vypočítávána vždy pro aktuální n. Vzhledem k tomu, že umocňování (odmocňování) je velice drahá operace, mohlo i to vést k částečnému zpomalení. Je také těžké určit kolik času z celkové doby konstrukce triangulace skutečně spotřebuje procházkový algoritmus a kolik zbylé činnosti. Při testování nebyl objeven žádný problém ve funkčnosti algoritmu.

3.5.2. Deformace terénu pro virtuální realitu

Cílem projektu Václava Purcharta je ověřit, zda jsou nepravidelné trojúhelníkové sítě použitelné v real-time aplikacích pro deformaci terénu. Úkolem bylo vytvořit „virtuální pískoviště“, tj. model terénu s písečným povrchem, do kterého bude možno v reálném čase provádět změny pomocí virtuálního nástroje. Budoucí využití projekt nalezne např. ve virtuální realitě na haptických zařízeních.

Lokace bodů pomocí procházek byly v této práci stejně jako v práci předchozí používány na konstrukci Delaunayovy triangulace inkrementálním vkládáním, potom pro práci s virtuálními nástroji a pro fyzikální vrstvu – když se virtuální nástroj snaží měnit výšku terénu tam, kde není žádný vrchol. Měření bylo ale stejně jako v předchozím případě použito pouze na konstrukci Delaunayovy triangulace inkrementálním vkládáním a výsledné časy zobrazuje následující tabulka (*Tab. 3.14*). Každý z algoritmů má pak v tabulce dva různé sloupečky, přičemž první sloupeček udává celkovou dobu konstrukce triangulace (v sekundách) a druhý udává čas z toho spotřebovaný při procházce. [Pu07]

vrcholů	Remembering Stochastic walk		Orthogonal walk	
	triang.	lokace	triang.	lokace
1000	0,011	0,002	0,008	0,001
3000	0,059	0,012	0,053	0,011
6000	0,144	0,051	0,145	0,055
10000	0,320	0,113	0,294	0,101
30000	1,386	0,763	1,333	0,678
60000	3,290	2,083	3,294	2,053
100000	6,229	4,404	5,895	4,041
300000	27,029	21,834	24,461	19,451
600000	72,580	60,883	69,718	57,607
1000000	159,774	134,610	139,782	115,335

Tabulka 3.14 – Doba konstrukce Delaunayovy triangulace algoritmem inkrementálního vkládání a čas spotřebovaný na lokace bodů (obojí v *s*)

Je evidentní, že ke znatelnému zrychlení došlo, avšak toto zrychlení je menší, než by se očekávalo dle podkapitoly 3.4.5. To může být způsobeno například i použitými datovými strukturami nebo případně hodnotou volenou pro optimální počet *výběrů počátečního trojúhelníku*.

4. Algoritmus procházky v E3

4.1. Úvod

Strategie procházky v tetrahedronové síti je velice podobná strategii procházky v triangulaci. Podstatou algoritmu je, že algoritmus zkoumá jeden tetrahedron (čtyřstěn) za druhým, přičemž do dalšího tetrahedronu cestuje přes stěnu aktuálně prozkoumávaného tetrahedronu a jako další tetrahedron zvolí ten, který vyhovuje nejlépe kritériu pro přiblížení se k hledanému bodu. Počáteční tetrahedron bývá volen náhodně nebo se například zvolí z množiny náhodně vybraných tetrahedronů ten, který je k hledanému bodu nejblíže. Procházky v E3 lze rozdělit stejně jako v E2 podle charakteru procházení (*přímé procházky*, *pravoúhlé procházky* a *procházky dle viditelnosti*). Zjišťujeme ale, že v E3 může nastat mnohem více singulárních situací než v E2, a proto jsou *přímé* a *pravoúhlé procházky* velice špatně použitelné. Z tohoto důvodu se budeme v E3 zabývat pouze *procházkami dle viditelnosti*.

4.2. Znaménkový test

Podobně jako v E2 testujeme polohu bodu vůči hraně trojúhelníku (vůči přímce), v E3 testujeme polohu bodu vůči stěně tetrahedronu (vůči rovině). Stěna je zadána třemi vrcholy, které jsou orientovány proti směru hodinových ručiček (body ***M***, ***N***, ***O***). Díky tomu je poloha bodu ***B*** určená jednoznačně a výsledné znaménko testu nám určí, na které straně roviny se testovaný bod nachází. K výpočtu polohy bodu vůči rovině je nejvýhodnější použít následující vzorec převzatý z [De02]:

$$f(\mathbf{M},\mathbf{N},\mathbf{O},\mathbf{B}) = \det\begin{vmatrix} Nx-Mx & Ox-Mx & Bx-Mx \\ Ny-My & Oy-My & By-My \\ Nz-Mz & Oz-Mz & Bz-Mz \end{vmatrix}.$$

4.3. Problém procházkových algoritmů

Stejně jako v E2 existuje i v E3 riziko zacyklení procházkového algoritmu. Situace je obdobná jako u E2. Hledaný bod je buďto totožný s jedním z bodů tetrahedronové sítě nebo se nachází tomuto bodu velice blízko. Při procházení procházkovým algoritmem potom opakovaně dostáváme v jednom ze znaménkových testů (vlivem zaokrouhlovací chyby v reprezentaci reálného čísla v pohyblivé řádové čárce) výsledek vně tetrahedronu, i když tomu tak být nemusí. Z tohoto důvodu se pak procházka může zacyklit. Možné řešení tohoto problému je stejné jako v E2 – použití adaptivního robustního

znaménkového testu, konkrétně *Shewchukova znaménkového testu pro E3* dle [Sh96a, Sh96b, Sh97].

4.4. Procházky dle viditelnosti v E3

Algoritmy p*rocházky dle viditelnosti v E3* pracují v podstatě stejně jako v E2. Rozdíl nalezneme pouze v tom, že místo trojúhelníků jsou ve strukturách uložené tetrahedrony a s nimi se také pracuje. Stejně jako u E2 pak platí, že v případě použití E3 modifikace algoritmu *Remembering walk* může dojít k zacyklení (s výjimkou Delaunayových tetrahedronových sítí – plyne z faktu, že na nich existuje částečné uspořádání stěn [Fl91]). Pro lokace bodu v tetrahedronových sítích, které nejsou Delaunayovské, je proto vhodnější použít E3 verzi algoritmu *Remembering Stochastic walk*. Náhodnosti dosáhneme stejným způsobem jako v E2.

Na následujícím obrázku (*Obr. 4.1*) je zobrazena vizualizace lokace náhodného bodu v tetrahedronové síti o velikosti 2000 bodů algoritmem *Remembering Stochastic Walk pro E3*. Procházka je znázorněná vybarvením procházených tetrahedronů. Barva vybarvení těchto tetrahedronů se postupně mění v závislosti na přiblížení k tetrahedronu, který obsahuje hledaný bod (od zelené k světle oranžové).

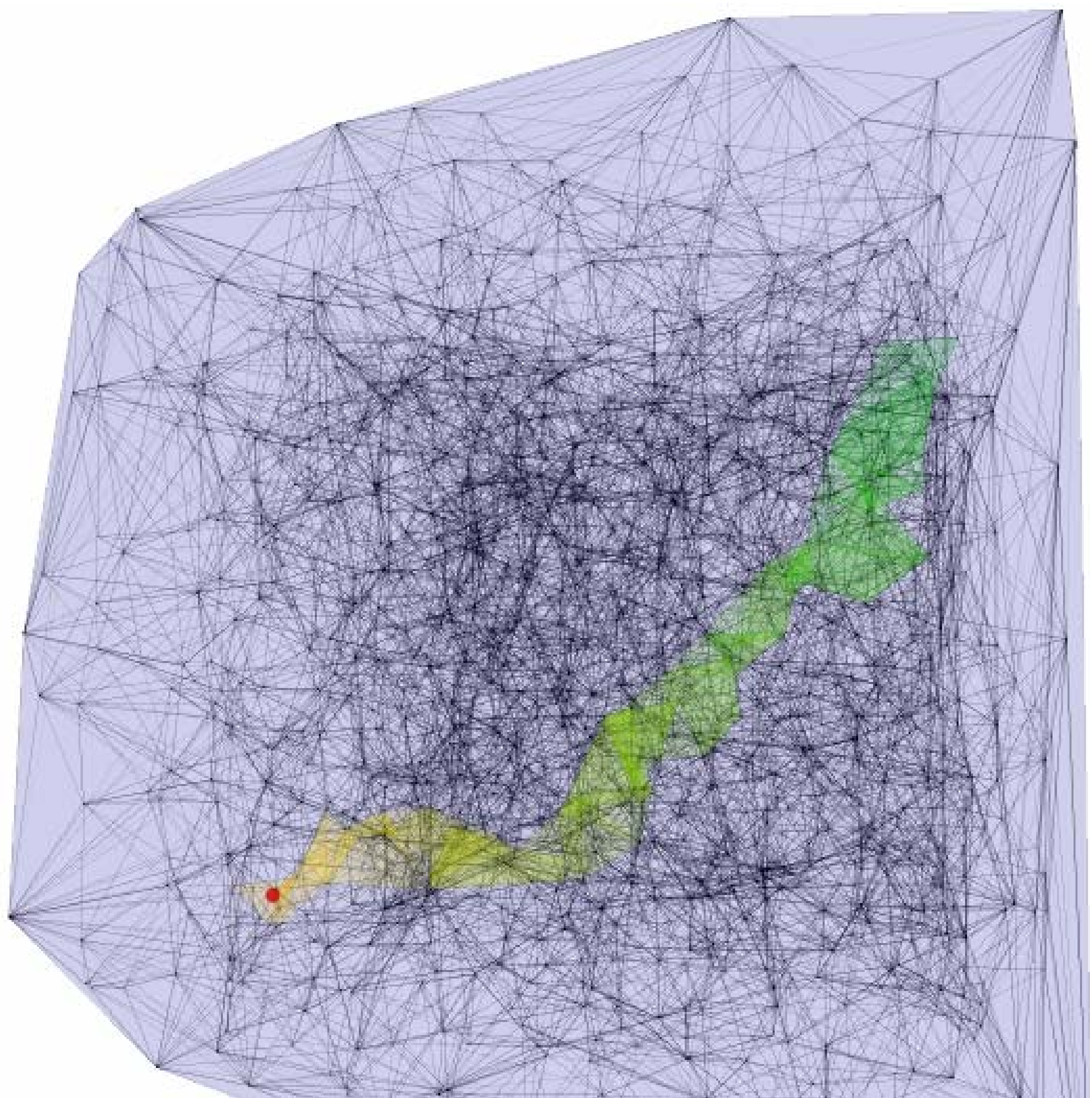
4.5. Výběr počátečního tetrahedronu

Důležitou součástí algoritmů procházky je výběr trojúhelníku (nebo v E3 v našem případě tetrahedronu), z kterého procházku začínáme. Do *výběru počátečního tetrahedronu* se vyplatí investovat i nějaký čas určený pro samotnou procházku, který může zmiňovanou procházku dosti výrazně zkrátit.

Úplně nejjednodušším, ale také velmi účinným způsobem *výběru počátečního tetrahedronu* je volba tetrahedronu pokud možno co nejbližšího hledanému bodu. Výpočet vzdálenosti tetrahedronu od hledaného bodu zjednodušíme na výpočet kvadratické vzdálenosti libovolného bodu tetrahedronu od cílového bodu.

Samotný algoritmus *výběru počátečního tetrahedronu* pak obdobně jako v E2 z náhodného vzorku několika tetrahedronů vybere ten nejbližší cílovému bodu. Důležitou otázkou ale zůstává správný výběr velikosti takovéhoto „náhodného vzorku“. Pokud zvolíme vzorek příliš veliký nebo naopak příliš malý, můžeme se hodně vzdálit očekávanému časovému optimu. Proto jsem se zabýval i vhodnou volbou velikosti takovéhoto vzorku odvíjeného od očekávané velikosti vstupu n[51]. Je pochopitelné, že změřené výsledky budou odlišné oproti E2.

[51] n = počet vrcholů tetrahedronové sítě



Obrázek 4.1 – Lokace bodu pomocí algoritmu *Remembering Stochastic walk pro E3*

5. Praktické testy procházek v E3

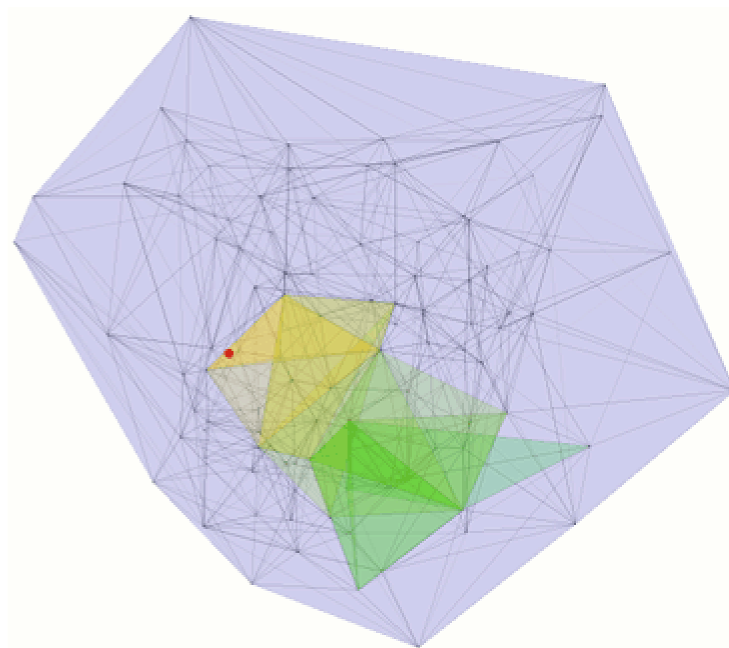
5.1. Použité programové prostředky

Implementace v E3 vznikla předěláním aplikace pro E2, proto zde také zůstaly zachovány všechny základní postupy. Struktury byly pouze modifikovány a upraveny pro potřeby E3 procházkových algoritmů. Většinu detailů ohledně programových prostředků lze proto najít v podkapitole 3.1.

5.1.1. JOGL

JOGL (*Java OpenGL*) byl použit na vizualizaci procházek v prostoru. Technologie má tu výhodu, že *OpenGL* vizualizace je zobrazena na klasickou Swing komponentu, takže původní 2D verzi nebylo třeba výrazně předělávat.

V aplikaci byla nejdříve vykreslena tetrahedronová síť v poloprůhledném drátěném modelu a poté i jednotlivé vybarvené (modré) stěny tetrahedronů, ovšem také s velkou mírou průhlednosti jednotlivých tetrahedronů (kvůli celkové přehlednosti). Při vizualizaci samotné procházky se zobrazí hledaný bod jako malá červená kulička. Cesta procházky pak zabarví všechny tetrahedrony, kterými prochází. V průběhu procházky je navíc postupně měněná barva ze zelené přes žlutou až na světle oranžovou. Se zobrazenými daty nelze žádným způsobem manipulovat, ale pro zdokonalení a zlepšení prostorového dojmu se zobrazená data na komponentě neustále otáčí. Na následujícím obrázku (*Obr. 5.1*) se nachází malá ukázka 3D vizualizace procházky pomocí *JOGLu*.



Obrázek 5.1 – 3D vizualizace procházky pomocí *JOGLu*

5.2. Výběr počátečního tetrahedronu

Velmi účinným a přitom jednoduchým způsobem urychlení procházkových algoritmů je opakované hledání počátečního tetrahedronu, do kterého se vyplatí investovat i podstatnou část očekávané doby běhu algoritmu. V následující tabulce (*Tab. 5.1*) jsou uvedeny časy algoritmu *Remembering walk pro E3* [4.4] (v *ms*) potřebné pro nalezení 25000 náhodně vygenerovaných bodů v závislosti na počtu *výběrů počátečního tetrahedronu*. Tabulka je značně redukovaná (jsou vybrané jen některé hodnoty počtu *výběrů počátečního tetrahedronu*). Navíc každé měření bylo provedeno v mnoha iteracích a v tabulce je uložen průměr ze všech naměřených hodnot. Hodnoty označené jako N/A nebyly změřeny, protože neměly pro následující analýzy (pro stanovení optimálního počtu *výběrů počátečního tetrahedronu*) význam. Pro testy byla použita konfigurace počítače zmíněná v podkapitole 3.1.4.

počet bodů sítě (*n*)	**Počet *výběrů počátečního tetrahedronu***												
	0	**1**	**2**	**3**	**4**	**5**	**6**	**8**	**10**	**15**	**20**	**30**	**40**
100	121	114	108	110	111	114	114	121	129	N/A	N/A	N/A	N/A
200	150	134	129	126	128	136	144	143	151	N/A	N/A	N/A	N/A
500	212	181	171	169	163	165	164	168	177	N/A	N/A	N/A	N/A
1000	264	229	211	204	198	195	196	197	205	N/A	N/A	N/A	N/A
2000	331	286	265	251	245	241	239	239	240	263	N/A	N/A	N/A
5000	478	418	385	367	356	354	347	347	347	369	N/A	N/A	N/A
10000	631	565	523	496	487	476	468	465	472	494	530	N/A	N/A
20000	857	776	708	670	644	631	613	612	609	630	673	N/A	N/A
50000	1267	1138	1032	971	920	891	870	843	830	841	872	N/A	N/A
100000	1688	1478	1337	1245	1178	1134	1100	1054	1024	1021	1043	1132	N/A
200000	2190	1903	1706	1586	1487	1433	1381	1314	1276	1237	1248	1327	N/A
500000	3107	2623	2346	2158	2023	1938	1859	1757	1689	1623	1609	1670	1780

Tabulka 5.1 – Časová závislost (v *ms*) *Remembering walk pro E3* na velikosti tetrahedronové sítě *n* (počet vrcholů sítě) a na počtu *Výběrů počátečního tetrahedronu*

Samozřejmě všechny časy uvedené v tabulkách (*Tab. 5.1* a *Tab. B.1*) v sobě zahrnují i čas potřebný pro *výběr počátečního tetrahedronu*. Sloupec s 0 *výběry počátečního tetrahedronu* obsahuje hodnoty naměřené pro náhodně vybrané počáteční tetrahedrony, zatímco u hodnot v sloupci s 1 *výběrem počátečního tetrahedronu* už probíhá právě jedno porovnání vhodnosti počátečního tetrahedronu s dalším náhodně vybraným tetrahedronem.

5.2.1. Volba vhodného počtu výběrů počátečního tetrahedronu

Důležitým krokem je odvození závislosti optimálního počtu počátečních výběrů na velikosti tetrahedronové sítě *n*. Lze očekávat, že se tato závislost bude odvíjet od způsobu implementace jednotlivých částí. Pokud bude rychlejší (poměrově vhledem k počtu *výběrů počátečního tetrahedronu*), bude optimální počet *výběrů počátečního tetrahedronu* menší. To bude platit i naopak. Stejně tak se dají očekávat odlišné výsledky i u různých typů procházek.

Nyní vyjádříme z podobných tabulek jako *Tab. 5.1* a *Tab. B.1*, které obsahují kompletní výčty počtu *výběrů počátečního tetrahedronu* pro jednotlivé algoritmy (obdobné tabulky jako *Tab. 5.1* a *Tab. B.1*, ale pro značnou rozsáhlost nejsou zobrazeny celé), tabulku optimálních počtů *výběrů počátečního tetrahedronu* v závislosti na velikosti vstupu. Následující tabulka (*Tab 5.2*) obsahuje zmíněnou závislost v jednotlivých algoritmech.

n	100	200	500	1K	2K	5K	10K	20K	50K	100K	200K	500K
Rem. walk	2	3	4	5	6	7	8	9	12	14	16	21
Rem. walk[52]	6	7	9	11	12	13	14	15	18	23	28	30
Rem. Stoch. walk	4	5	6	7	8	9	10	11	12	13	17	19
Rem. Stoch. Walk[49]	8	9	10	12	13	14	15	17	19	21	23	27

Tabulka 5.2 – Optimální počet výběrů počátečního tetrahedronu

Na první pohled je patrné, že optimální hodnota počtu výběrů závisí přímo úměrně na velikosti tetrahedronové sítě. Navíc je i patrné, že optimální hodnota počtu *výběrů počátečního tetrahedronu* roste mnohem pomaleji než roste velikost sítě, tudíž je zřejmé, že závislost bude nižší než lineární (sublineární). Také je zřejmé, že pro vyšší velikost vstupních dat roste rychleji, tudíž nepůjde o logaritmickou závislost.

Prozkoumáme nyní tempo růstu různých odmocninových výrazů ($\sqrt[3]{n}, \sqrt[4]{n}, \sqrt[5]{n}$). Zjišťujeme, že tempo růstu výrazu $\sqrt[3]{n}$ je příliš rychlé (oproti skutečnému růstu), tempo růstu výrazu $\sqrt[4]{n}$ je stále poměrně rychlé, ale temto růstu výrazu $\sqrt[5]{n}$ je naproti tomu již pomalejší než ve skutečnosti. Proto se nyní budeme snažit najít koeficient k závislosti $\sqrt[k]{n}$, která vystihuje nejlépe zmíněnou závislost. K řešení použijeme doplněk *Microsoft Excel – Řešitel*. Je pochopitelné, že výsledná hodnota k bude v intervalu $(4 \quad 5)$. Po drobném zaokrouhlení dostáváme pomocí *MS Excel řešitele* hodnotu 4,*5*.

Nyní máme zjištěnou míru závislosti optimálního počtu *výběrů počátečního tetrahedronu* na velikosti vstupní triangulace *n* (zjednodušeně by se to dalo nazvat jako *charakteristické tempo růstu*), nikoliv však přesný optimální počet výběrů pro danou velikost tetrahedronové sítě *n*. Zavedeme

[52] S použitím externí *Shewchukovy knihovny* na *adaptivní robustní znaménkový test pro E3*.

proto výraz ve tvaru $a \cdot \sqrt[4{,}5]{n} + b$. Opět pomocí *MS Excel řešitele* vypočítáme optimální hodnoty koeficientů *a* a *b* pro jednotlivé algoritmy. Následující tabulka (*Tab 5.4*) shrnuje spočtené hodnoty koeficientů.

Algoritmus	Koeficienty	
	a	*b*
Remembering walk	1,13	-0,57
Remembering walk[53]	1,49	2,54
Remembering Stochastic walk	0,87	3,00
Remembering Stochastic walk[50]	1,10	6,71

Tabulka 5.3 – Optimální hodnoty koeficientů pro jednotlivé algoritmy

5.2.2. Dosažené urychlení

Již z tabulky *Tab. 5.1*, respektive *Tab. B.1*, je patrné, že opakovaným *výběrem počátečního tetrahedronu* lze dosáhnout velmi výrazného urychlení, především pak při manipulaci s větším objemem dat. Samozřejmě maximální urychlení se odvíjí i od použitého lokačního algoritmu. V následující tabulce (*Tab. 5.4*) je čas bez *výběru počátečního tetrahedronu* označen jako *B*, čas s optimálním počtem výběrů označen jako *O* a procentuální urychlení je značeno jako *U*.

Algoritmus	n=500			n=5000			n=50000			n=500000		
	B[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]	*B*[*ms*]	*O*[*ms*]	*U*[%]
Remembering walk	76,3	62,9	17,6	181,6	137,3	24,4	491,6	324,0	34,1	1206,2	632,5	47,6
Remembering walk[50]	180,4	130,9	27,4	397,5	247,8	37,7	934,7	507,8	45,7	2153,4	969,3	55,0
Rem.Stoch. walk	94,8	73,5	22,5	218,4	150,7	31,0	573,4	354,4	38,2	1425,3	690,0	51,6
Rem. Stoch. walk[50]	200,0	132,5	33,8	436,0	285,5	34,5	1158,5	591,6	48,9	2600,3	1102,5	57,6

Tabulka 5.4 – Porovnání časů s použitím a bez použití optimálního počtu výběrů počátečního tetrahedronu pro jednotlivé algoritmy (časy lokace 10000 bodů)

Z tabulky (*Tab. 5.4*) je patrné, že není samozřejmě míra urychlení *U[%]*[54] závislá pouze na konkrétním algoritmu, ale závisí i na velikosti tetrahedronové sítě. Čím vyšší je objem vstupních dat, tím se i vhodným *výběrem počátečního tetrahedronu* urychlí celý lokační algoritmus. Dá se tudíž předpokládat, že se při dalším růstu objemu vstupních dat zvýší i nynější urychlení, které je nyní pro 500000 bodů u některých algoritmů více jak dvojnásobné (U >50%).

[53] S použitím externí *Shewchukovy knihovny* na *adaptivní robustní znaménkový test pro E3*.
[54] Míra urychlení *U[%]* udává kolik procent času ušetříme při lokaci s použitím optimálního výběru počátečního trojúhelníku oproti lokaci bez zmíněného vylepšení.

5.2.3. Vliv vstupních dat na výběr počátečního tetrahedronu

Předchozí výsledky vycházely z pokusů na *Delaunayově tetrahedronizaci* s rovnoměrným rozložením náhodně vygenerovaných bodů. Algoritmy byly testovány i na tetrahedronových sítích vzniklých z reálných dat a výsledky byly v podstatě stejné jako v předchozích případech. Kromě reálných dat jsem již neměl k dispozici jiná 3D data.

5.3. Vlastnosti jednotlivých algoritmů

Vzhledem k tomu, že nás zajímají pokud možno nejlepší výsledky, budou v této podkapitole zkoumány pouze hodnoty vzniklé nastavením optimální hodnoty *Výběru počátečního tetrahedronu* pro příslušné algoritmy.

5.3.1. Odhadovaná očekávaná časová složitost

U všech procházkových algoritmů jsou velice důležité vlastnosti, jako je průměrný počet znaménkových testů na jednu lokaci, případně maximální počet znaménkových testů v lokaci nebo průměrná délka procházky (počet navštívených tetrahedronů), případně maximální délka.

Následující tabulka (*Tab. 5.5*) shrnuje tyto zmíněné vlastnosti u algoritmů *procházek dle viditelnosti pro E3*. Bylo lokalizováno vždy 500000 náhodně vygenerovaných bodů v *Delaunayově tetrahedronizaci* vzniklou z rovnoměrně rozložených náhodně vygenerovaných bodů.

počet bodů sítě. (*n*)	Remembering walk				Remembering Stochastic walk				Testů stěn na tetrahedron	
	Testy stěn		Délka cesty		Testy stěn		Délka cesty		Rem. walk	Rem. Stoch. walk
	Ø počet	max	Ø počet	max	Ø počet	max	Ø počet	max		
100	18,6	77	7,6	35	15,9	67	6,9	34	2,428	2,300
200	24,7	94	10,6	40	20,1	78	8,9	39	2,333	2,253
500	31,7	116	13,9	50	26,8	95	12,4	45	2,284	2,167
1000	37,1	135	16,4	64	31,3	108	14,8	55	2,257	2,122
2000	43,2	146	19,6	72	36,9	140	17,8	70	2,206	2,071
5000	54,5	206	25,1	99	46,2	164	22,9	88	2,175	2,019
10000	64,5	231	30,1	112	54,8	183	27,6	95	2,146	1,985
20000	74,6	256	35,1	118	65,5	220	33,5	123	2,127	1,954
50000	93,4	327	44,5	153	82,0	292	42,7	160	2,097	1,922
100000	110,2	384	52,9	184	98,8	372	52,0	200	2,081	1,899
200000	131,3	486	63,6	248	117,2	434	62,3	234	2,066	1,881
500000	163,1	548	79,5	276	147,5	550	79,3	300	2,051	1,860

Tabulka 5.5 – Testy *procházek dle viditelnosti pro E3*

Velmi zajímavým kritériem pro hodnocení procházkových algoritmů je vztah mezi počtem znaménkových testů a délkou procházky (počet

znaménkových testů na jeden tetrahedron). Toto kritérium je zkoumáno v tabulce (*Tab. 5.5*) v posledních dvou sloupcích, kde je uvedeno jak pro stochastickou, tak nestochastickou verzi algoritmů. Zajímavé je, že u stochastické verze vychází toto kritérium lépe. Díky tomu lze očekávat částečnou kompenzaci času, kterou musí stochastická verze vynaložit navíc na randomizaci.

U obou algoritmů navíc můžeme vidět tendenci poklesu kritéria počtu znaménkových testů na jeden tetrahedron se vzrůstající velikostí triangulace. Původ tohoto jevu je v podstatě velice jednoduchý. Čím blíže jsme totiž cílovému bodu, tím je méně pravděpodobné, že do dalšího tetrahedronu přejdeme již po první (respektive druhé) testované stěně, a tedy musíme provést další znaménkový test respektive testy.

Nyní se zaměříme na odhad časové složitosti algoritmů. Předem tedy již máme omezenou výslednou časovou složitost zdola složitostí $O\left(\sqrt[4,5]{n}\right)$, která je potřebná pro *výběr počátečního tetrahedronu*. Nyní budeme zjišťovat časovou složitost samotné lokace. V dalších odhadech nebudeme uvažovat singulární případy a budeme počítat pouze očekávanou složitost v běžných případech. Na první pohled je z tabulky (*Tab. 5.5*) patrné, že v tomto případě roste délka procházky poměrně rychle a nelze tudíž uvažovat o potenciální logaritmické složitosti, proto budeme předpokládat aproximující sublineární funkci ve tvaru $a \cdot \sqrt[b]{n}$, kde b bude z intervalu $(1 \quad \infty)$. Na vyřešení tohoto problému použijeme doplněk *Microsoft Excel – Řešitel*, kde se budeme snažit minimalizovat absolutní rozdíl vypočítané a reálné délky procházky (u všech dostupných velikostí triangulací (*Tab. 5.4*)) pomocí změn koeficientů a a b. Výpočet aplikujeme pouze na jeden z algoritmů v tabulce, protože vzhledem k velmi podobným hodnotám lze očekávat i velice podobný výsledek. Koeficient a již dále nebudeme uvažovat, protože pro očekávanou složitost není směrodatný. Výpočtem byl určen koeficient b jako 3,95. Lze tedy říci, že odhadovaná časová složitost *procházek dle viditelnosti pro E3* je přibližně $O\left(\sqrt[3,95]{n}\right)$ v případě, že použijeme optimální hodnotu *výběru počátečního tetrahedronu*. Je potřeba dodat, že v případě reálných dat dostáváme výsledky velmi podobné.

5.3.2. Porovnání jednotlivých algoritmů

Následující tabulka (*Tab. 5.6*) obsahuje vzájemné porovnání časů lokace pomocí všech probíraných algoritmů pro E3. Pro všechny algoritmy bylo voleno optimální nastavení *výběru počátečního trojúhelníku*, případně dalšího vstupního parametru. V případě, že optimální nastavení nebylo známo, bylo voleno takové nastavení, které lze považovat za téměř optimální. Pro kvalitnější výsledky byly změřeny časy lokace 500000 bodů. Testování probíhalo na počítači popsaném v podkapitole 3.1.4.

počet bodů tetrahedronové sítě (*n*)	100	500	1000	5000	10000	20000	50000	100000	200000	500000
Rem. walk	2154	3106	3735	6811	9200	12140	16640	20438	24564	32317
Rem. walk[55]	4498	6518	7781	12828	16343	19593	25698	31424	37591	48125
Rem. Stoch. walk	2675	3706	5356	8860	10658	13435	18782	21769	26577	35139
Rem. Stoch. walk[52]	4953	6843	8424	15221	18518	23252	29406	32217	38909	49830

Tabulka 5.6 – Časy porovnání algoritmů pro E3 (v m*s*)

Dle očekávání jsou časy algoritmu *Remembering Stochastic walk pro E3* jen nepatrně horší, protože algoritmus provádí méně znaménkových testů, což částečně vykompenzuje režii na randomizaci.

5.4. Praktická aplikace

5.4.1. Využití regulárních triangulací pro hledání tunelů v molekulách proteinů

Proteiny (bílkoviny) jsou obsaženy v každé organické buňce, kde plní stavební, transportní, obranné, řídící a další funkce. Skládají se z několika stovek až tisíců aminokyselin. Proteinové inženýrství zkoumá vlastnosti proteinů a cílenými mutacemi se snaží vybrané vlastnosti změnit. K tomu využívá (mimo jiné) analýzu tunelů v proteinech. Existence a šířka tunelu vedoucího z určitého místa proteinu na jeho povrch ovlivňuje chemické vlastnosti proteinu – má-li dojít k chemické reakci mezi nějakou látkou a určitou částí proteinu, musí být tato část zvnějšku přístupná. Znalost geometrické aproximace tunelu může pomoci odborníkům zaměřit se při modifikacích proteinu na jeho patřičnou část. V této práci je navržena metoda hledání tunelů v proteinech pomocí power diagramu a jeho duální struktury – regulární triangulace. Výsledné tunely jsou detailně porovnány s tunely nalezenými známou metodou v Delaunayově triangulaci.

[Ze07]

Procházky byly testované v práci M. Zemka na profileru pro reálná data (proteiny). Celkově byly testy provedeny na 3 různých proteinech. Porovnání s původním *Remembering Stochastic walk* algoritmem je zobrazeno v následující tabulce (*Tab. 5.7*). Časy v tabulce jsou uvedené v sekundách.

Nutno poznamenat, že pro porovnávání nebyl použit *výběr počátečního tetrahedronu*. Z tabulky je patrné, že už pro protein s 13600 body je zlepšení oproti původnímu algoritmu více než 10% a dá se očekávat, že zlepšení by se projevilo ještě více pro větší objemy dat.

[55] S použitím externí *Shewchukovy knihovny* na *adaptivní robustní znaménkový test pro E3*.

počet vrcholů	čas lokace bodů včetně volání vnořených funkcí		čas lokace	
	implementace dle podkapitoly 4.4	původní implementace	implementace dle podkapitoly 4.4	původní implementace
1200	0,2902	0,2899	0,136	0,1359
4700	1,3199	1,3467	0,6023	0,6394
13600	6,4162	6,7269	2,8137	3,2582

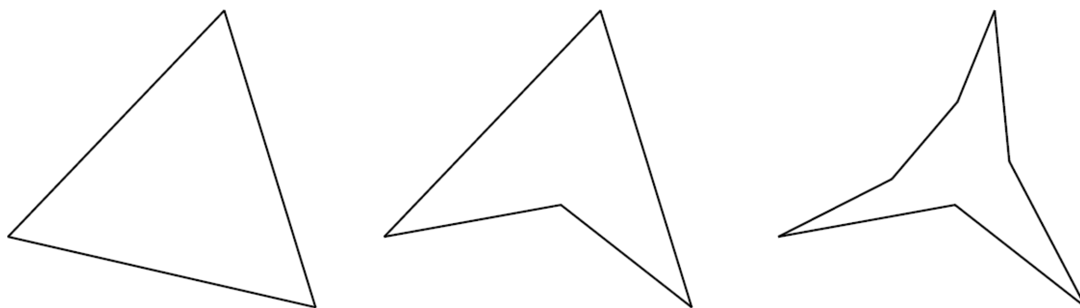
Tabulka 5.7 – Porovnání algoritmů v praktické aplikaci (obojí v *s*)

6. Procházky v pseudotriangulaci

Obsah této kapitoly (včetně obrázků) je z velké části převzat z [Tr07a] nebo z [Tr07b].

6.1. Základní pojmy a definice

Pseudotrojúhelník je rovinný polygon, který má právě tři konvexní vrcholy (dále nazývané rohy) s vnitřním úhlem menším než 180°. Následující obrázek (*Obr. 6.1*) ukazuje, že jednotlivé rohy spojují až tři konkávní řetězce hran (dále nazývané strany). Vrcholy, v němž má pseudotrojúhelník vnitřní úhel větší než úhel přímý, se nazývají tupoúhlé vrcholy. Trojúhelník je jednoduchým příkladem pseudotrojúhelníku.



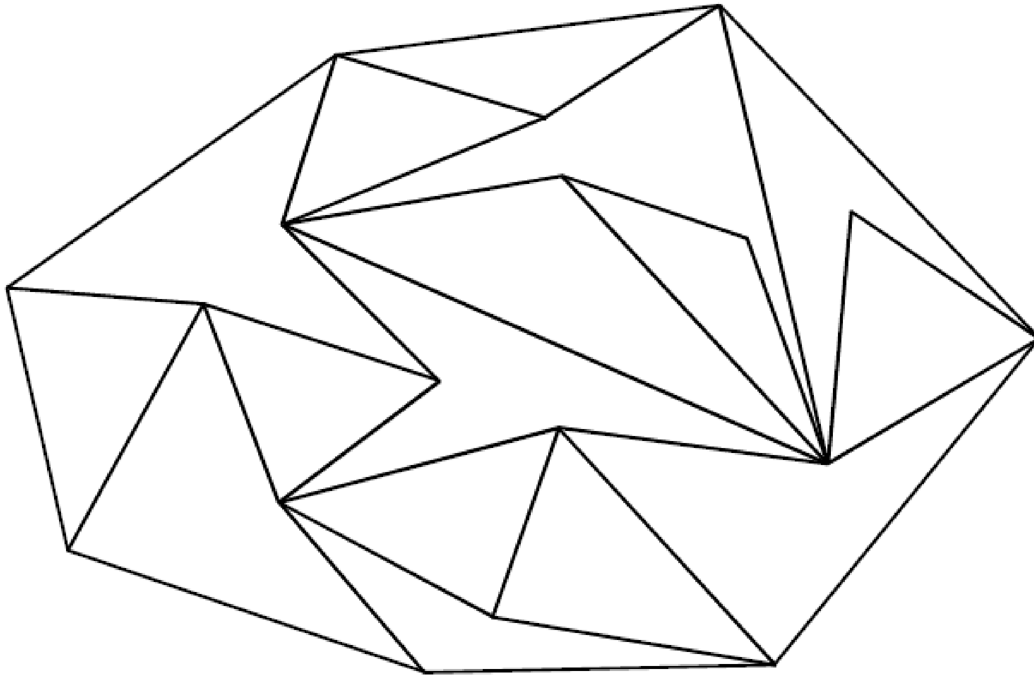
Obrázek 6.1 - Pseudotrojúhelníky

Pseudotriangulace konečné množiny $\boldsymbol{S}$ n bodů v rovině je pak rovinné dělení konvexního obalu množiny $\boldsymbol{S}$ na pseudotrojúhelníky, jejichž množina vrcholů je právě množina $\boldsymbol{S}$. Ukázková pseudotriangulace množiny bodů je zobrazena na následujícím obrázku (*Obr. 6.2*).

Samozřejmě potom je triangulace speciálním případem pseudotriangulace. Pojmy roh a tupoúhlý vrchol se přenášejí i do pseudotriangulace. Může se tedy stát, že jeden vrchol je současně pro jeden pseudotrojúhelník rohem, zatímco pro jiný pseudotrojúhelník je tupoúhlým vrcholem.

Základní rozdíl mezi trojúhelníkem a pseudotrojúhelníkem je v tom, že zatímco trojúhelník má vrcholy spojené jednou hranou, pseudotrojúhelník má rohy spojené stranou, která však může obsahovat víc než jen jednu hranu. Tedy pro algoritmus procházky na pseudotriangulaci je třeba minimálně přeformulovat test (orientace bodu vůči hraně trojúhelníku – viz [0]) určující volbu následujícího pseudotrojúhelníku.

Pro jednodušší popis algoritmů je třeba nadefinovat několik pojmů. Nechť je hledán bod $\boldsymbol{V}$ a aktuálně testovaný pseudotrojúhelník *PT* má rohy $\boldsymbol{C_1}$, $\boldsymbol{C_2}$ a $\boldsymbol{C_3}$. Hrana *PT* je označována *e*. Virtuální hranou *PT* je nazývána úsečka $\boldsymbol{C_1C_2}$,



Obrázek 6.2 – Ukázka *pseudotriangulace*

$\boldsymbol{C_2C_3}$ nebo $\boldsymbol{C_1C_3}$ a je značena *ve*. Konkávní řetězec hran, odpovídajících virtuální hraně *ve* je značen *vE*. Virtuálním trojúhelníkem *PT* je nazýván konvexní obal *PT*. Neboli virtuální hrana *PT* je hranou virtuálního trojúhelníku *PT*. Nechť je dán *PT* a jedna jeho hrana *e*. Hrana *e* náleží řetězci hran, který odpovídá nějaké virtuální hraně *ve*. Nechť *ve* obsahuje dva rohy $\boldsymbol{C_1}$, $\boldsymbol{C_2}$ daného *PT*. Třetí roh $\boldsymbol{C_3}$ je pak nazýván protější vrchol vůči hraně *e*. Vrchol $\boldsymbol{C_3}$ je možno nazvat protější také vůči virtuální hraně *ve*.

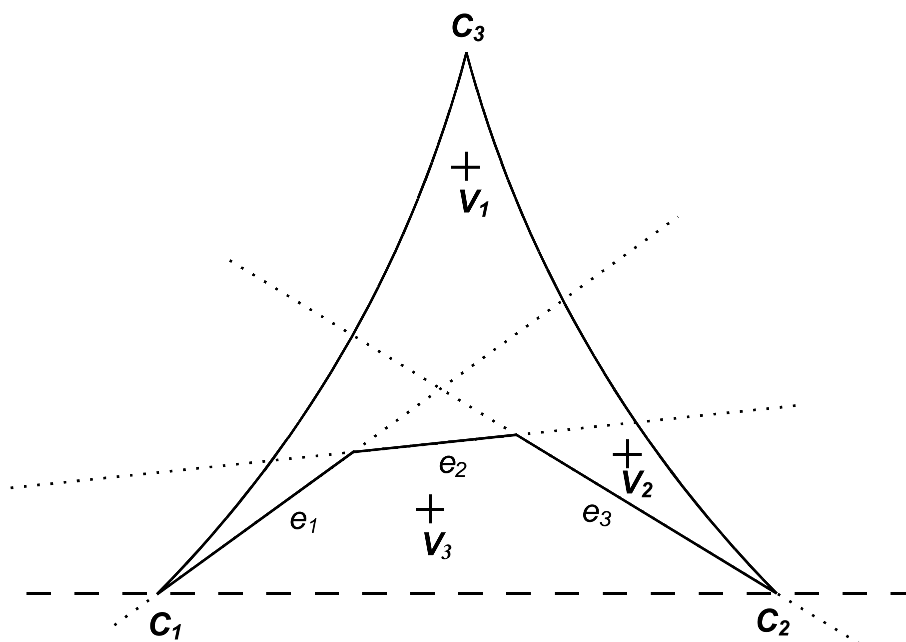
6.2. Testy orientace

Nechť je dán *PT*, jedna jeho hrana *e* a bod ***V***, vůči kterému se testuje, potom testem orientace bodu vůči hraně rozumíme znaménkový test ve smyslu podkapitoly 2.2.

Nechť je dán *PT*, jedna jeho virtuální hrana *ve* a bod ***V***, vůči kterému se orientace testuje. Virtuální hraně *ve* odpovídá konkávní řetězec hran *vE*. Je-li bod ***V*** vůči všem hranám z *vE* vně, pak je možno říct, že test orientace bodu ***V*** vůči řetězci hran *vE* skončil s výsledkem vně. Pokud pro alespoň jednu hranu platí, že bod ***V*** je uvnitř, pak je možno říct, že test pozice bodu ***V*** vůči řetězci hran *vE* skončil s výsledkem uvnitř. Zkráceně pak, že bod ***V*** je uvnitř *PT* vůči *vE*, respektive bod ***V*** je vně *PT* vůči *vE*.

Předchozí definici je ilustrována na obrázku (*Obr. 6.3*), na kterém je testována orientace bodů $\boldsymbol{V_1}$, $\boldsymbol{V_2}$ a $\boldsymbol{V_3}$ vůči konkávnímu řetězci hran odpovídající horizontální virtuální hraně (a jejíž prodloužení je znázorněno čárkovanou čárou). Tečkovanou čárou jsou pak znázorněny prodloužení jednotlivých hran

z řetězce. Z definice plyne, že test pro body V_1 a V_2 skončí s výsledkem uvnitř, zatímco pro bod V_3 skončí test s výsledkem vně.



Obrázek 6.3 – Orientace bodů vůči konkávnímu řetězci hran

Nebudou-li vrcholy spolu s vyhledávaným bodem v obecné poloze, je třeba ještě ukázat, jak se zachovat v situaci, kdy vyhledávaný bod leží na hraně či ve vrcholu pseudotriangulace nebo když bude uprostřed běhu algoritmu vyhledávaný bod ležet na přímce procházející testovanou hranou.

Test orientace bodu vůči hraně počítá, zdali bod leží v polorovině nebo polorovině opačné. Bylo tedy určeno, že přímka dělící rovinu na dvě poloroviny náleží oběma polorovinám. Což ale taky znamená, že pro bod ležící na hranici dvou stěn může algoritmus vrátit oba pseudotrojúhelníky dvě stěny jako výsledek (neboli že bod na hraně náleží zároveň více stěnám). A pro bod ležící ve vrcholu pseudotriangulace algoritmus může vrátit libovolný pseudotrojúhelník, který má daný vrchol na své hranici (ne nutně jako roh).

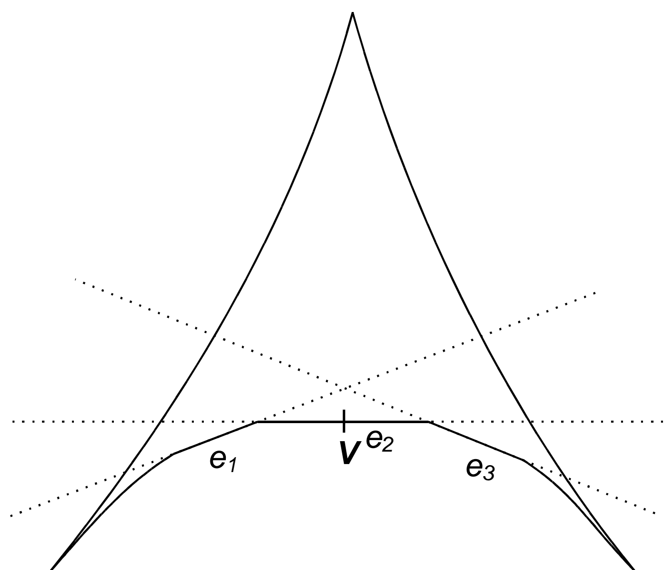
Test byl realizován jako třístavový: pokud testovaný bod leží uvnitř dané poloroviny a zároveň ne na přímce, test vrací kladné číslo, pokud leží na přímce (procházející testovanou hranou, tzv. „na hraně“), pak test vrací nulu a pokud leží na opačné polorovině a zároveň ne na přímce, pak vrací záporné číslo. Jak se tedy algoritmus zachová, když test vyjde s výsledkem na hraně?

V případě testů virtuálních hran pokračuje, jako kdyby výsledek vyšel s výsledkem uvnitř, přičemž si ale pamatuje, že pro danou virtuální hranu vyšel test s výsledkem na hraně. Pokud potom dojde na test konkávních řetězců hran (neboli algoritmus zůstane v aktuální stěně pro krok 3), je testován pouze konkávní řetězec odpovídající virtuální hraně s výsledkem na hraně. Ostatní řetězce již není třeba testovat, protože pro ně musí testy vyjít uvnitř.

V případě testů řetězců hran výsledek na hraně již v podstatě znamená konečný výsledek – bod leží na nějaké hraně pseudotriangulace. Test orientace vůči řetězci hran se však skládá z několika testů vůči jednotlivým hranám, pro které může test také vyjít s výsledkem na hraně. Pokud takto pro nějakou hranu z řetězce test vyjde, algoritmus pokračuje, jako kdyby šlo o výsledek uvnitř, ale zapamatuje si, že test vyšel na hraně. Této informace pak využije k ošetření singulárních případů (neboli k případům, kdy vyhledávaný bod leží na hraně nebo ve vrcholu pseudotriangulace) následovně:

Pokud vyhledávaný bod leží ve „vnitřním" vrcholu řetězce, pak pro dvě sousedící hrany (které daný vrchol sdílejí) vyjde test vůči řetězci s výsledkem na hraně.

Pokud vyhledávaný bod leží na hraně z řetězce hran, pak pro tři po sobě jdoucí hrany musí platit: pro první hranu test vyjde s výsledkem vně, pro druhou hranu vyjde test s výsledkem na hraně a pro třetí hranu vyjde test s výsledkem vně, jak ukazuje *Obr. 6.4*. Pokud bod leží na poslední hraně z řetězce, pak stačí, aby pro předposlední hranu vyšel test s výsledkem vně a pro poslední hranu s výsledkem na hraně.



Obrázek 6.4 – Test řetězce hran s výsledkem na hraně

Snížení celkového počtu testů hran lze získat triviální úpravou. Je-li třeba testovat konkávní řetězec hran, který obsahuje právě jednu hranu, pak to již není třeba dělat, protože výsledek je již znám z testu pro odpovídající virtuální hranu.

6.3. Jednoduchý algoritmus procházky

Každá vnitřní hrana pseudotriangulace (kromě hran na konvexním obalu) náleží právě dvěma pseudotrojúhelníkům. Nechť hrana *e* náleží právě PT_1 a

PT_2. Přechodem do dalšího pseudotrojúhelníku je pak míněna změna aktuálního pseudotrojúhelníku z PT_1 na PT_2. Řekneme, že jsme do aktuálního *PT* přešli přes virtuální hranu *ve*, pokud jsme do aktuálního *PT* přešli přes hranu *e*, která patří do řetězce hran *vE*.

Hlavní myšlenkou *Jednoduchého algoritmu procházky* je nahrazení testu orientace hledaného bodu vůči hraně (jak je tomu v triangulaci) za test vůči virtuální hraně. Pokud test skončí s výsledkem vně, pak algoritmus přejde do následujícího pseudotrojúhelníku přes nějakou (danou) hranu z odpovídajícího řetězce hran. Algoritmus si také pamatuje, přes kterou virtuální hranu do aktuálního pseudotrojúhelníku přišel a tuto virtuální hranu již netestuje. Algoritmus rozepsaný do kroků vypadá následovně (*Alg. 6.1*).

1) Zvolí se počáteční (aktuální) *PT*

2) Otestuje se pozice hledaného bodu vůči maximálně dvěma virtuálním hranám aktuálního *PT* (kterými se do aktuálního *PT* nepřišlo). Testy, jednotlivých virtuálních hran se provádí v daném pořadí.

   a) Pokud je bod vně (vůči nějaké virtuální hraně *ve*), pak se přejde do dalšího pseudotrojúhelníku přes danou hranu z řetězce hran *vE* a pokračuje se znovu krokem **2**.
   b) Pokud však takový pseudotrojúhelník neexistuje, skončí se s výsledkem, že bod leží mimo strukturu pseudotriangulace.
   c) Pokud je bod uvnitř vůči oběma dvěma virtuálním hranám, pokračuje se krokem **3**.

3) Otestuje se pozice hledaného bodu vůči řetězcům hran *PT* neobsahujícím hranu *e*, kterou se do aktuálního *PT* přišlo. Jednotlivé řetězce jsou testovány v daném pořadí.

   a) Pokud je bod uvnitř vůči řetězcům hran (kterými se do *PT* nepřišlo), končí se s výsledkem, že byl bod nalezen v aktuálním *PT*.
   b) Pokud je bod vně (vůči alespoň jednomu řetězci *vE*), přejde se do dalšího pseudotrojúhelníku přes danou hranu z řetězce hran *vE* a pokračuje se krokem **2**.
   c) Pokud však takový pseudotrojúhelník neexistuje, skončí se s výsledkem, že bod leží mimo pseudotriangulaci.

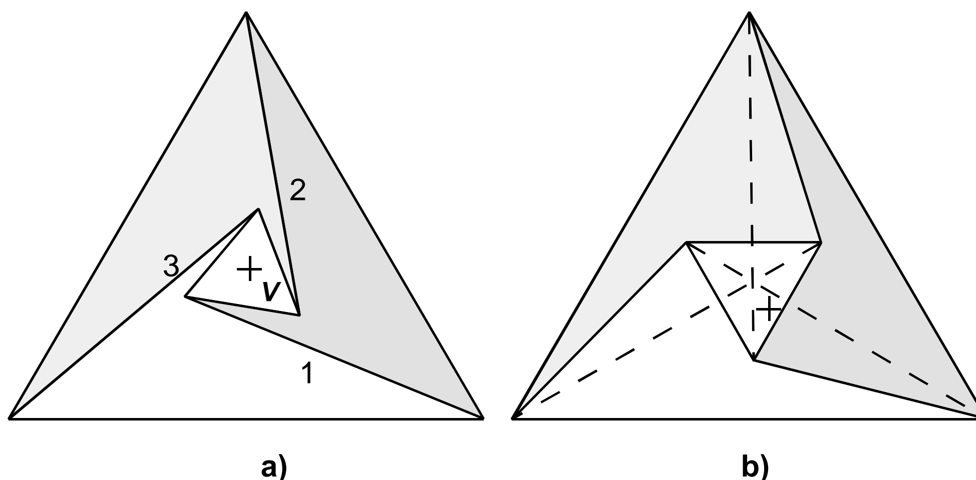
Algoritmus 6.1 – *Jednoduchý algoritmus procházky* v pseudotriangulaci

6.3.1. Problém Jednoduchého algoritmu procházky

Algoritmus je jednoduchý a přímočarý, nicméně se ukazuje, že v původním algoritmu nestačí takto jednoduše upravit test, aby se dal použít i pro pseudotriangulaci.

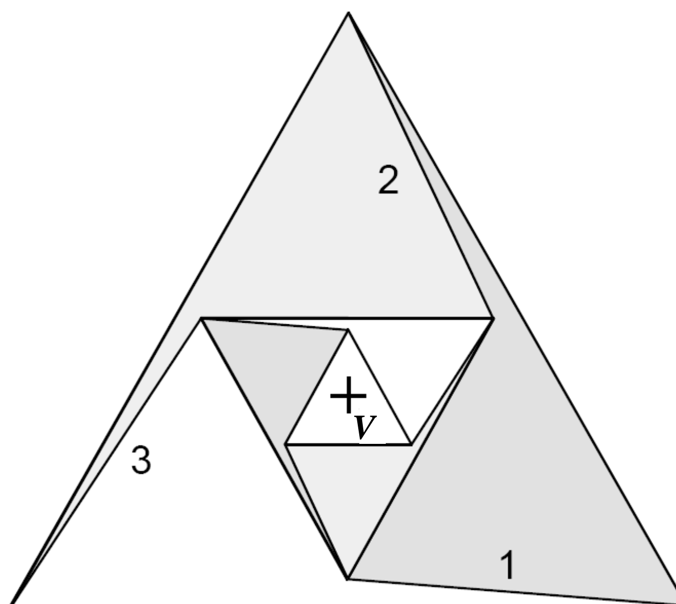
Jak už bylo řečeno, na nespeciálních (ne-Delaunayových) triangulacích se může nerandomizovaný algoritmus procházky zacyklit (viz *Obr. 2.2*). Pro obecné pseudotriangulace pak platí, že se *Jednoduchý algoritmus procházky* může zacyklit (viz *Obr. 6.5a*). Vzhledem k tomu, že algoritmus provádí testy orientace bodu ***V*** vůči virtuálním hranám v daném pořadí, může se stát, že sekvence hran zvolená algoritmem způsobí zacyklení (na *Obr. 6.5a* nastane

zacyklení, bude-li se v každém pseudotrojúhelníku provádět první test orientace bodu ***V*** vůči hraně s číslem – postupně 1, 2, 3).



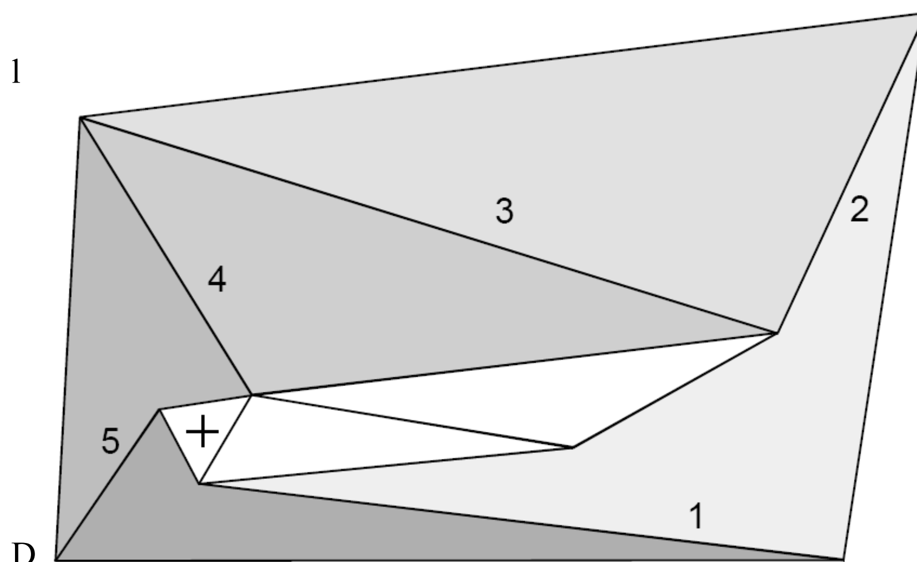
Obrázek 6.5 – Zacyklení *Jednoduchého procházkového algoritmu*

Na další variantě „zacyklených“ pseudotrojúhelníků spolu s jejich virtuálními hranami (*Obr. 6.5b*) lze vidět, že vyhledávaný bod nemusí ležet uvnitř všech virtuálních trojúhelníků, které se podílejí na cyklu. Pohledem na *obrázek 6.4a* by se zdálo, že pokud je bod uvnitř virtuálního trojúhelníku *PT* a zároveň neleží uvnitř pseudotrojúhelníku *PT*, pak musí nutně ležet v pseudotrojúhelníku, se kterým má *PT* společnou hranu. Není těžké dokázat, že tvrzení neplatí (*Obr. 6.6*). Algoritmus se bude cyklit opět přes hrany 1, 2 a 3, přičemž ale hledaný bod leží ve vnitřním trojúhelníku, který nesousedí s žádným pseudotrojúhelníkem podílejícím se na cyklu.



Obrázek 6.6 - Zacyklení *Jednoduchého procházkového algoritmu*

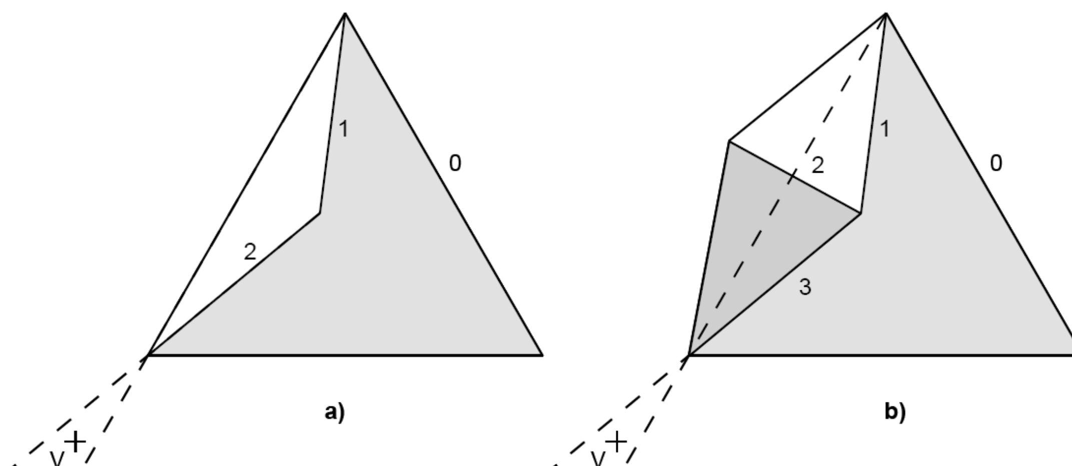
Protože se algoritmus chová lokálně (neboli v daném okamžiku nezná nic jiného než aktuálně zkoumaný pseudotrojúhelník) a protože algoritmus může mít od pseudotrojúhelníku podílejícího se na cyklu až ke hledanému bodu „vzdálenost“ několik dalších pseudotrojúhelníků, nepůjde zřejmě cyklus vyřešit přídavnými testy (např. zkontrolováním sousedních stěn) při současném zachování lokality. Lokalita je však základním kamenem algoritmů procházky a nelze ji porušit. Navíc se ukazuje, že cyklus může obsahovat libovolný počet pseudotrojúhelníků (viz *Obr. 6.7*, kde cyklus přechází přes hrany 1, 2, 3, 4, 5).



Obrázek 6.7 – Cyklus procházky obsahující více stěn

Dále se ukazuje, že se algoritmus nemusí zacyklit pouze na konci procházky v cyklu stěn okolo vyhledávaného bodu, ale dokonce může navštívit pseudotrojúhelník dvakrát už v průběhu procházky (*Obr. 6.8a* – vstup hranou 0 a průchod přes hrany 1, 2). Pokud by si algoritmus pamatoval předchozí pseudotrojúhelník (aby se do něj nevracel), tak protipříklad ukazuje (viz *Obr. 6.8b* – vstup hranou 0 a průchod přes hrany 1, 2, 3), že ani to nezabrání opakované návštěvě jednoho pseudotrojúhelníku, protože cyklus může procházet přes víc jak jednu stěnu. Pro názornost je přerušovanou čárou vyznačeno prodloužení dvou virtuálních hran dvou stěn.

Z předchozích případů plyne, že je třeba použít randomizovanou verzi algoritmu, neboli algoritmus náhodné procházky. Randomizace pak může probíhat na dvou úrovních: při výběru virtuální hrany pro test orientace vyhledávaného bodu a při výběru hrany, přes kterou algoritmus přejde do následujícího pseudotrojúhelníku. Je nutné použít obě úrovně randomizace. Následující úpravou (*Alg. 6.2*) původního popisu (*Alg. 6.1*) dostáváme randomizovanou variantu algoritmu.



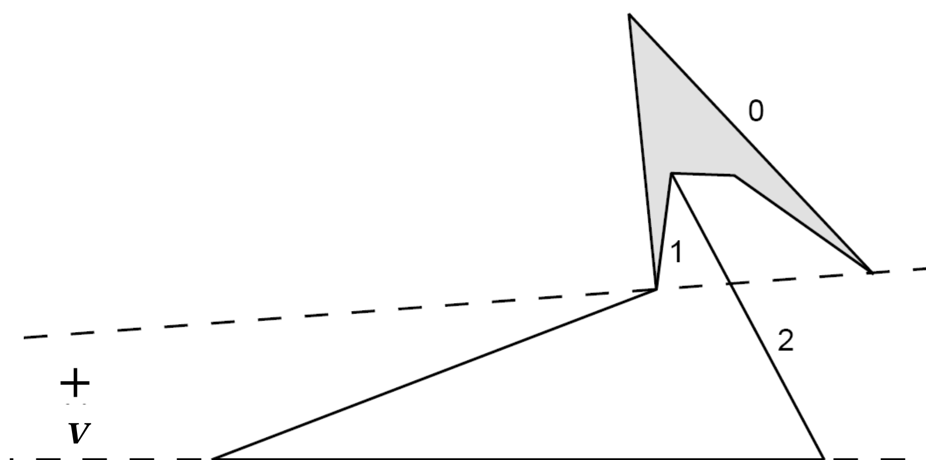
Obrázek 6.8 – Vícenásobné navštívení jednoho pseudotrojúhelníku

```
1) V kroku 2 se nebudou provádět testy virtuálních hran v daném, ale
   v náhodném pořadí.

2) V kroku 3 se nebudou testovat jednotlivé řetězce v daném, ale v
   náhodném pořadí.

3) V krocích 2 a 3 se nebude přecházet do dalšího pseudotrojúhelníku
   přes danou hranu z příslušného řetězce, ale přes hranu z řetězce
   náhodně zvolenou.
```

Algoritmus 6.2 – Randomizace *jednoduchého algoritmu procházky* (*Alg. 6.1*)

6.3.2. Problém randomizované verze jednoduchého algoritmu

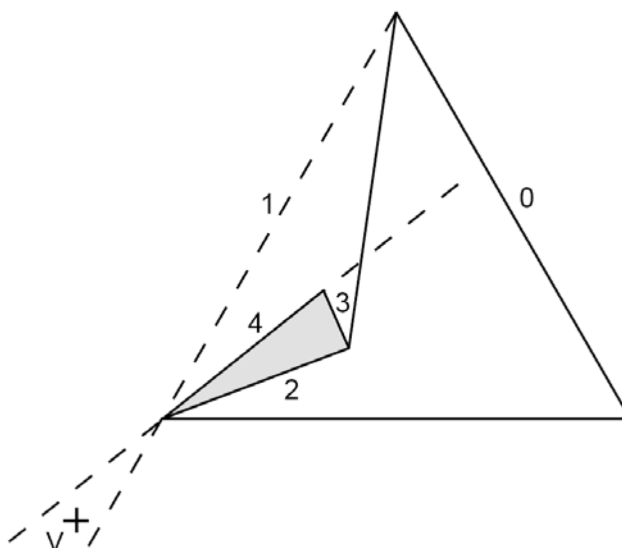
Díky randomizaci se zajistí, že se algoritmus nezacyklí. Nicméně ihned vyvstane další problém. Ukazuje se, že nestačí test orientace bodu vůči hraně nahradit testem orientace bodu vůči virtuální hraně a následně bez dalšího testu přejít přes libovolnou hranu do další stěny (*Obr. 6.9*).



Obrázek 6.9 – Problém netestování třetího řetězce hran

Obrázek ukazuje situaci uprostřed běhu algoritmu, který se právě dostal do šedého pseudotrojúhelníku přes hranu 0. Náhodně pro test zvolil spodní virtuální hranu a protože test pro tuto virtuální hranu vyšel s výsledkem vně, přešel do bílého pseudotrojúhelníku přes náhodně vybranou hranu 1. Jelikož pro virtuální hrany 2 a 3 bílého pseudotrojúhelníku skončí test s výsledkem uvnitř, celý algoritmus skončí s výsledkem, že bod leží uvnitř bílého pseudotrojúhelníku. Bod však v bílém pseudotrojúhelníku neleží a algoritmus by měl pokračovat přes třetí netestovanou virtuální hranu.

Situace může být ještě jednodušší (viz *Obr. 6.10*). Na obrázku se opět nacházíme uprostřed běhu algoritmu, který se právě se dostal do bílého pseudotrojúhelníku přes hranu 0. Pro test náhodně algoritmus zvolil virtuální hranu 1 a protože vyšel s výsledkem vně, přešel do šedého (pseudo)trojúhelníku přes hranu 2. Jelikož pro virtuální hrany 3 a 4 skončí test pro (pseudo)trojúhelník s výsledkem uvnitř, tak algoritmus skončí s výsledkem bod nalezen uvnitř šedého (pseudo)trojúhelníku. Což opět není pravda a algoritmus by měl pokračovat zpátky do bílé stěny.



Obrázek 6.10 – Problém návratu do předchozího pseudotrojúhelníku

6.4. Algoritmus procházky verze 1

Kvůli případům ukázaným v předchozí podkapitole (0 - Algoritmus procházky – *jednoduchá verze*) je tedy algoritmus upraven tak, aby testoval i třetí virtuální hranu, respektive hrany z odpovídajícího třetího řetězce hran, a mohl se tak i vrátit do předchozí stěny, bude-li to situace vyžadovat. Výsledek znázorňuje následující algoritmus (*Alg. 6.3*).

1) Zvolí se počáteční (aktuální) pseudotrojúhelník PT

2) Otestuje se pozice hledaného bodu vůči virtuálním hranám (minimálně jedné, maximálně všem třem) aktuálního *PT*. Pořadí testovaných virtuálních hran ve_1, ve_2, ve_3 se volí následovně:

   a) Nejdříve se testují virtuální hrany, kterými se do aktuálního *PT* nepřišlo (pořadí těchto dvou virtuálních hran je náhodné) a naposledy se pak testuje virtuální hrana, kterou se do aktuálního *PT* přišlo.
   b) Pokud je bod vně (vůči nějaké virtuální hraně *ve*), pak se přejde do dalšího pseudotrojúhelníku přes hranu z řetězce hran *vE* a pokračuje se znovu krokem **2**. Hrana *e* z *vE*, přes kterou se přejde do následujícího *PT*, se volí náhodně, ale snahou je se nevracet.

      i) Neboli, pokud by se mělo jít z aktuálního *PT* do dalšího přes virtuální hranu *ve*, kterou se do aktuálního *PT* nepřišlo, tak se zvolí hrana z *vE* libovolně.
      ii) Pokud by se mělo jít z aktuálního *PT* do dalšího přes stejnou virtuální hranu, přes kterou se do aktuálního *PT* přišlo, pak se zvolí z odpovídajícího řetězce hran libovolná jiná hrana než je hrana e_{old}, kterou se do *PT* skutečně přišlo.
      iii) Pokud však jiná taková hrana neexistuje (řetězec hran *vE* obsahuje jen hranu $e=e_{old}$), je třeba se vrátit do předchozího pseudotrojúhelníku právě přes hranou *e*.
      iv) Pokud však takový pseudotrojúhelník neexistuje (do kterého se dá přejít), skončí se s výsledkem, že bod leží mimo strukturu pseudotriangulace.
      v) Pokud je bod uvnitř vůči všem třem virtuálním hranám, pokračuje se krokem **3**.

3) Otestuje se pozice hledaného bodu vůči všem třem řetězcům hran *PT*. Řetězce jsou testovány ve stejném pořadí jako v kroku **2**.

   a) Pokud je bod uvnitř všech řetězců hran, končí se s výsledkem, že byl bod nalezen v aktuálním *PT*.
   b) Pokud je bod vně (vůči alespoň jednomu řetězci *vE*), přejde se do dalšího pseudotrojúhelníku přes hranu z řetězce hran *vE* a pokračuje se krokem **2**. Hrana pro přechod se vybírá stejně jako v kroku **2**, tedy náhodně, ale snahou je se nevracet.

      i) Neboli, pokud by se mělo jít z aktuálního *PT* do dalšího přes virtuální hranu *ve*, kterou se do aktuálního *PT* nepřišlo, tak se zvolí hrana z *vE* libovolně.
      ii) Pokud by se mělo jít z aktuálního *PT* do dalšího přes stejnou virtuální hranu, přes kterou se do aktuálního *PT* přišlo, pak se zvolí z odpovídajícího řetězce hran libovolná jiná hrana než je hrana e_{old}, kterou se do *PT* skutečně přišlo.
      iii) Pokud však jiná taková hrana neexistuje (řetězec hran *vE* obsahuje jen hranu $e=e_{old}$), je třeba se vrátit do předchozího pseudotrojúhelníku právě přes hranou *e*.
      iv) Pokud však neexistuje pseudotrojúhelník, do kterého by se dalo přejít, skončí se s výsledkem, že bod leží mimo pseudotriangulaci.

Algoritmus 6.3 – *Algoritmus procházky verze 1*

6.4.1. Shrnutí algoritmu

V algoritmu je hrana *e* z řetězce *vE* (přes který se přešlo do aktuální stěny) pro test vybírána docela speciálně. Pro randomizaci algoritmu však není nutno volit hranu zrovna takto, například by stačilo, kdyby se pro test vybírala hrana *e* z celého řetězce *vE* úplně náhodně. Nicméně testy na našich datech ukazují, že problémové situace nastávají zřídka, a tedy se vyplatí hranu e_{old}, testovat jako poslední. Je totiž velká pravděpodobnost, že tento test vyjde s výsledkem uvnitř, a tak by se test prováděl zbytečně.

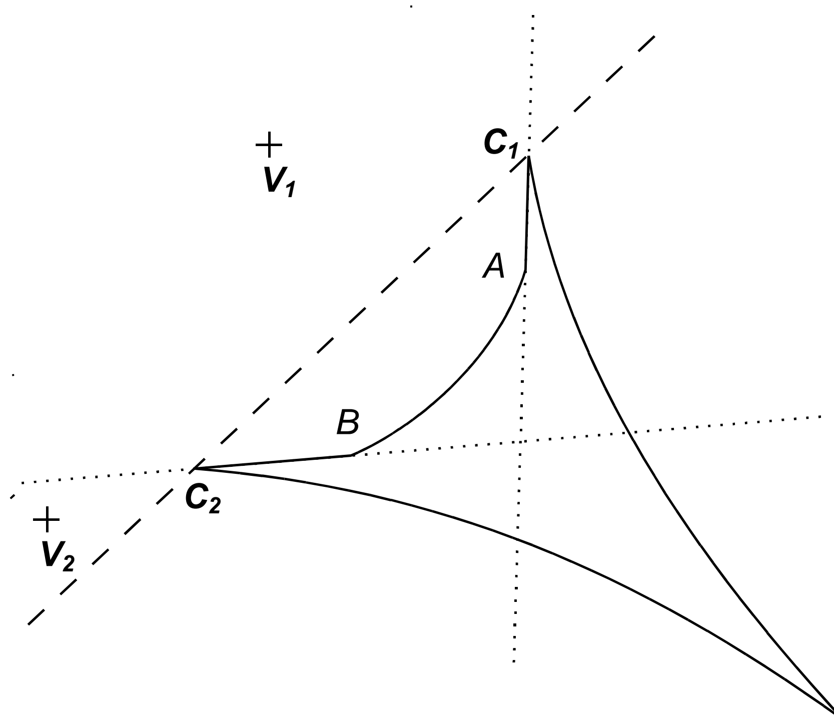
Ačkoliv pro *Algoritmus procházky verze 1* není znám důkaz správnosti, ukázal se v testech dostatečně stabilní, tj. nezacyklil se a pro bod uvnitř pseudotriangulace vracel pseudotrojúhelník. Pro bod mimo pseudotriangulaci pak vracel odpověď, že se bod nalézá mimo strukturu. Navíc je třeba poznamenat, že pokud algoritmus nalezne bod uvnitř nějakého pseudotrojúhelníku, pak se v něm bod musí nutně nalézat, protože pro tento pseudotrojúhelník algoritmus musel provést test orientace vůči všem hranám na jeho okraji s výsledkem uvnitř.

6.5. Algoritmus procházky verze 2

Nechť *PT* je pseudotrojúhelník a *ve* jedna jeho virtuální hrana, pro kterou byl proveden test pro bod ***V*** s výsledkem vně. Potom je jasné, že v odpovídajícím konkávním řetězci hran *vE* existuje alespoň jedna hrana *e*, pro kterou test pro bod ***V*** vyjde také s výsledkem vně. To zřejmě platí, pokud *vE* obsahuje jen jednu hranu *e*.

Nechť tedy *vE* obsahuje alespoň 2 hrany. Označme polorovinu danou virtuální hranou *ve* a bodem ***V*** jako vnější polorovinu. Nechť *ve* spojuje rohy $\boldsymbol{C_1}$ a $\boldsymbol{C_2}$, které jsou orientovány proti směru hodinových ručiček. První hranou *vE* nazveme takovou hranu, která obsahuje vrchol $\boldsymbol{C_1}$. Poslední hranou *vE* nazveme takovou hranu, která obsahuje vrchol $\boldsymbol{C_2}$. Potom je zřejmě vnější polorovina obsažena ve sjednocení polorovin, z nichž první je dána první hranou *vE* a rohem $\boldsymbol{C_2}$ a druhá je dána poslední hranou *vE* a rohem $\boldsymbol{C_1}$. Z toho plyne, že buď pro první nebo poslední hranu *vE* vyjde test pro bod ***V*** s výsledkem vně. Situace je ilustrována na obrázku (*Obr. 6.11*), kde vyhledávaný bod $\boldsymbol{V_1}$ je vně první i poslední hrany *vE* a vyhledávaný bod $\boldsymbol{V_2}$ je pouze vně poslední hrany *vE*. Tečkovanou čárou jsou zvýrazněny prodloužení první a poslední hrany *vE*, čárkovanou pak prodloužení virtuální hrany.

Porovnáme-li problémové situace (*Obr. 6.9* a *Obr. 6.10*), zjistíme, že problémovou částí je přechod do další stěny přes hranu z konkávního řetězce hran poté, co jsme udělali test pouze pro odpovídající virtuální hranu. Právě nesourodost „řetězec hran – virtuální hrana" způsobuje situace, které je třeba speciálně ošetřovat. Zatímco pro virtuální hranu test vychází s výsledkem vně, pro hranu, přes kterou následně přecházíme do následujícího



Obrázek 6.11 – Důkaz testu s výsledkem vně vůči hraně

pseudotrojúhelníku, už může test vycházet s výsledkem uvnitř. Následující algoritmus se problémům elegantně vyhýbá tím, že přechází pouze přes hranu, pro kterou test vychází s výsledkem vně. Z úvodu této podkapitoly plyne, že taková hrana vždy existuje, tedy takový přechod je možno vždy provést. Zároveň není těžké ověřit, že pokud přejdu přes hranu *e* s výsledkem testu vně do nového pseudotrojúhelníku, pak už musí nutně v novém pseudotrojúhelníku vyjít test s výsledkem uvnitř pro všechny hrany ze stejného řetězce jako je hrana *e*, po které jsme do pseudotrojúhelníku přešli. Algoritmus rozepsaný do kroků vypadá následovně (*Alg. 6.4*).

1) Zvolí se počáteční (aktuální) *PT*

2) Otestuje se pozice hledaného bodu vůči maximálně dvěma virtuálním hranám aktuálního *PT* (kterými se do aktuálního *PT* nepřišlo). Testy jednotlivých virtuálních hran se provádí v náhodném pořadí.

   a) Pokud je bod vně (vůči nějaké virtuální hraně *ve*), pak se přejde do dalšího pseudotrojúhelníku přes náhodně zvolenou hranu z řetězce hran *vE*, pro kterou test vyjde s výsledkem vně, a pokračuje se znovu krokem **2.**
   b) Pokud však takový pseudotrojúhelník neexistuje, skončí se s výsledkem, že bod leží mimo strukturu pseudotriangulace.
   c) Pokud je bod uvnitř vůči oběma dvěma virtuálním hranám, pokračuje se krokem **3.**

3) Otestuje se pozice hledaného bodu vůči řetězcům hran *PT* neobsahujícím hranu *e*, kterou se do aktuálního *PT* přišlo. Jednotlivé řetězce jsou testovány v náhodném pořadí.

   a) Pokud je bod uvnitř vůči řetězcům hran (kterými se do *PT* nepřišlo), končí se s výsledkem, že byl bod nalezen v aktuálním *PT*.
   b) Pokud je bod vně (vůči alespoň jednomu řetězci *vE*), přejde se do dalšího pseudotrojúhelníku přes náhodně zvolenou hranu z

```
   řetězce hran vE, pro kterou test vyjde s výsledkem vně. A
   pokračuje se krokem 2.
c) Pokud však takový pseudotrojúhelník neexistuje, skončí se s
   výsledkem, že bod leží mimo pseudotriangulaci.
```

Algoritmus 6.4 – *Algoritmus procházky verze 2*

6.5.1. Shrnutí algoritmu

Algoritmus verze 2 oproti *Algoritmu verze 1* nemusí testovat virtuální hranu, přes kterou přešel do aktuálního pseudotrojúhelníku, což je myšlenka převzatá z algoritmu procházky pro triangulaci.

Algoritmus verze 2 oproti A*lgoritmu verze* 1 dělá navíc testy jednotlivých hran při přechodu do dalšího pseudotrojúhelníku, zatímco *Algoritmus verze 1* oproti *Algoritmu verze 2* dělá navíc testy na třetí straně aktuálního pseudotrojúhelníku.

7. Praktické testy procházek v pseudotriangulaci

Obsah této kapitoly vychází z [Tr07a] a z [Tr07b]. Kompletní testování je prováděno celé znovu a jako testovací aplikace je použita aplikace, která byla součástí diplomové práce J. Trčky. Vzhledem k tomu, že se v [Tr07a] provádějí i testy lokace na původní triangulaci (triangulace z které vznikla pseudotriangulace), pro srovnání byla do testování zahrnuta i implementace lokace bodu podle podkapitoly 3.1 pomocí algoritmu *Remembering Stochastic walk* implementovaná dle pseudokódu v podkapitole 2.4.

7.1. Vstupní data

Algoritmy byly testovány na různých vstupních datech (rovnoměrné rozložení náhodně vygenerovaných bodů, body uspořádány do shluků (*clusters*), body rozmístěné ve čtvercové síti (*grid*), body náhodně generované na základě Gaussova rozložení pravděpodobnosti a body rozmístěné na základě reálných dat), přesněji řečeno na různých sadách bodů, z nichž jsme (externím programem) vytvořili Delaunayovu triangulaci. Tyto triangulace posloužily jako vstup pro algoritmus minimální pseudotriangulace. Vzhledem k tomu, že se výsledky testů na datech typu *clusters* a datech bodů náhodně generovaných na základě Gaussova rozložení pravděpodobnosti velice podobaly výsledkům testů na datech vzniklých z rovnoměrného rozložení náhodně vygenerovaných bodů, nebudou tyto výsledky dále zmiňovány.

7.2. Způsob testování

Body byly nejdříve lokalizovány algoritmem *Remembering Stochastic walk* (podkap. 2.4) (pro triangulaci), přičemž procházky začínaly v náhodném trojúhelníku. Pak byla triangulace převedena na minimální pseudotriangulace metodou *ET* [Tr07a] (odebráním hran a trojúhelníků) a stejné body (jako v případě jednotlivých triangulací) byly lokalizovány oběma variantami algoritmu náhodné procházky pro pseudotriangulaci. Lokalizace označená *Pseudo 1* je implementace algoritmu z 6.4, lokalizace označená *Pseudo 2* je implementace algoritmu z (6.5). Obě verze procházek na pseudotriangulaci začínaly z náhodné počáteční stěny. Počáteční pseudotrojúhelníky nemusely být v případech *Pseudo 1* a *Pseudo 2* stejné.

V testech jsme se zaměřili především na měření počtu testů hran (ať už virtuálních nebo skutečných), délku procházky a cyklení jednotlivých algoritmů.

7.3. Počet hranových testů

Popis jednotlivých sloupců tabulky (*Tab. 7.1*) je následující (další tabulky mají sloupce odpovídajícího významu). *Počet* udává průměrný počet testů hran na jednu procházku. *Zlepšení* udává o kolik procent se zmenšil počet testů na pseudotriangulaci oproti triangulaci (počítáno jako $(1 - Pseudo\,1/Triang)\%$ respektive $(1 - Pseudo\,2/Triang)\%$). *Max testů hran* udává maximální počet testů hran během jedné procházky.

První tabulka (*Tab 7.1*) ukazuje, kolik bylo třeba provést testů hran, ať už hran virtuálních nebo skutečných. Z ní můžeme vyčíst, že počet testů hran se v případě algoritmu *Pseudo 1* snížil typicky o více jak 10%. Zmenšení počtu testovaných hran v *Pseudo 1* se může zdát výrazné, nicméně to plyne i z faktu, že samotných hran v pseudotriangulacích je obvykle méně než v triangulacích, tudíž se dá očekávat, že i hran v procházce bude testováno méně. Rozptyl od lehce záporného zlepšení až po 18% v *Pseudo 1* považujeme za vliv vstupních dat. V případě 100 vrcholů je vzorek nedostatečně velký, ostatní odchylky jsou pak pouze v řádu jednotek procent. V případě *Pseudo 2* pak hodnota přibližně odpovídá počtu testů na triangulaci.

Zajímavým ukazatelem jsou řádky zobrazující výsledky pro data umístěné v pravidelné mřížce, protože na nich je pseudotriangulace totožná s triangulací, a řádky nám tak ukazují, jak se algoritmy *Pseudo 1* a *Pseudo 2* chovají na triangulaci. Můžeme vidět, že na triangulaci se algoritmy *Triang*, *Pseudo 1* a *Pseudo 2* přibližně rovnají, odchylku v řádu jednotlivých procent přisuzujeme vlivu vstupních dat.

Poslední sloupce tabulky pak ukazují, že i výsledky měření délek jednotlivých procházek odpovídají sloupcům předchozím, tedy délka nejdelší procházky na pseudotriangulaci algoritmem *Pseudo 1* vykazuje zlepšení oproti triangulaci, naproti tomu algoritmus *Pseudo 2* zlepšení téměř nevykazuje. Za výjimku můžeme považovat výsledky pro některá data se 100 vrcholy, kde je nejdelší procházka na pseudotriangulaci delší než na triangulaci. Toto zhoršení přiřazujeme (krom vlivu vstupních dat) také vlivu „cyklů“ na pseudotriangulacích (zatímco na vstupní Delaunayově triangulaci se algoritmus procházky zacyklit nemůže).

Poznamenejme, že struktury s 1000 vrcholů mají počet stěn větší než 2500. Pokud bylo v testu lokalizováno pouze 2500 bodů, pak tyto struktury nutně musí obsahovat stěny, jejichž žádný bod nebyl lokalizován. Proto mohou být výsledky těchto struktur zkreslené.

Větší počet testů hran algoritmu *Pseudo 2* oproti *Pseudo 1* přiřazujeme vlivu nutného jednoho testu navíc při přechodu do dalšího pseudotrojúhelníku (kdy musíme dodržet, že pro hranu, přes kterou přecházíme, musí test vyjít s výsledkem vně). To by mohlo být vyváženo případným testováním třetí virtuální hrany (a řetězce hran) během *Pseudo 1*, ale zdá se, že takové situace nastávají zřídka.

Data	Vrcholů	Testy hran							
		Počet			Zlepšení		Max testů hran		
		Triang	Pseudo 1	Pseudo 2	Pseudo 1	Pseudo 2	Triang	Pseudo 1	Pseudo 2
Random rovnoměrný	100	17,26	17,42	17,91	-0,89%	-3,71%	45	55	87
	500	37,24	34,34	37,95	7,79%	-1,92%	100	90	98
	1000	52,09	46,22	51,84	11,27%	0,49%	142	116	137
	5000	113,26	97,91	110,84	13,55%	2,14%	269	256	296
	10000	158,20	134,66	154,33	14,88%	2,45%	394	345	410
	50000	348,36	295,42	340,70	15,20%	2,20%	846	751	869
	100000	492,92	414,66	479,50	15,88%	2,72%	1267	1071	1208
Grid	1000	49,99	50,91	50,03	-1,84%	-0,08%	121	125	121
	10000	156,75	157,59	156,53	-0,53%	0,15%	399	384	381
	100000	487,21	488,83	488,18	-0,33%	-0,20%	1230	1215	1211
Reálná data	4897	122,31	103,48	119,09	15,39%	2,63%	409	355	385
	15820	211,21	176,41	204,91	16,48%	2,98%	647	519	589
	70433	416,57	348,27	406,16	16,40%	2,50%	1364	1127	1357

Tabulka 7.1 – Testy hran náhodné procházky (průměr pro 10000 bodů)

7.4. Délka procházek

V délkách procházek (*Tab. 7.2*) lze vidět jednoznačný trend zmenšení počtu navštívených stěn oproti náhodné procházce na triangulaci, ať už použijeme *Pseudo 1* nebo *Pseudo 2*. Výsledek je jednoduše zdůvodnitelný menším počtem stěn v pseudotriangulacích (oproti počtu stěn v triangulacích). Tomu by odpovídalo i to, že pokud se algoritmy *Pseudo 1* a *Pseudo 2* chovají

Data	Vrcholů	Délka procházky							
		Počet			Zlepšení		Max délka procházky		
		Triang	Pseudo 1	Pseudo 2	Pseudo 1	Pseudo 2	Triang	Pseudo 1	Pseudo 2
Random rovnoměrný	100	11,92	10,42	10,50	12,54%	11,87%	32	36	42
	500	27,27	23,39	23,58	14,26%	13,54%	70	63	60
	1000	38,73	32,67	32,98	15,65%	14,84%	105	86	89
	5000	86,19	72,86	73,69	15,47%	14,51%	207	186	192
	10000	121,06	101,48	102,66	16,18%	15,20%	301	268	271
	50000	269,13	227,00	230,11	15,65%	14,50%	663	572	589
	100000	381,81	320,22	324,77	16,13%	14,94%	980	836	830
Grid	1000	36,12	36,09	36,10	0,08%	0,06%	93	93	92
	10000	115,67	115,59	115,56	0,06%	0,09%	299	288	291
	100000	361,43	361,93	362,02	-0,14%	-0,16%	943	930	941
Reálná data	4897	92,31	76,75	77,76	16,85%	15,77%	306	267	246
	15820	160,04	133,04	134,95	16,88%	15,68%	497	386	383
	70433	320,36	266,97	272,04	16,67%	15,08%	1046	886	930

Tabulka 7.2 – Délka náhodné procházky (průměr pro 10000 bodů)

náhodně, měly by jejich délky procházek vycházet podobně (protože oba algoritmy se pro přechod do dalšího pseudotrojúhelníku rozhodují stejně, pouze pro přechod jinak volí hranu). Tento předpoklad se zřejmě potvrzuje.

7.5. Opakované návštěvy stěn

Následující tabulka (*Tab. 7.3*) si všímá „cyklů“, neboli opakovaných vstupů na jednotlivé stěny v průběhu procházky. Sloupeček *Vstupů na navštívenou stěnu* ukazuje počet opakovaných vstupů na (alespoň jednou již navštívenou) stěnu zprůměrovaných na jednu procházku. Např. hodnota 2 by znamenala, že v každé procházce algoritmus dvakrát vstoupil na již navštívenou stěnu. Tabulka ukazuje, že se algoritmy vracejí pouze minimálně. Pouze u velkých dat (100000 vrcholů) je počet opakovaně navštívených stěn větší než jedna, což je považováno za zanedbatelnou hodnotu vzhledem k průměrné délce procházky, která činí okolo 300 stěn. Na druhou stranu čísla ukazují, že se s opakovanými návštěvami musí počítat.

Data	Vrcholů	Vstupů na navštívenou stěnu		
		Triang	Pseudo 1	Pseudo 2
Random rovnoměrný	100	0	0,01	0,02
	500	0	0,04	0,07
	1000	0	0,06	0,12
	5000	0	0,14	0,28
	10000	0	0,21	0,42
	50000	0	0,49	0,98
	100000	0	0,70	1,38
Grid	1000	0	0,00	0,00
	10000	0	0,00	0,00
	100000	0	0,00	0,00
Reálná data	4897	0	0,19	0,38
	15820	0	0,40	0,72
	70433	0	0,70	1,36

Tabulka 7.3 – Opakované návštěvy stěn (průměr pro 10000 bodů)

7.6. Rychlost procházkových algoritmů

Další část testů se týká rychlosti jednotlivých algoritmů. Testování probíhalo na počítači s následujícími parametry: procesor Intel Pentium D 3,0GHz 2x2MB L2 cache přetaktovaný na 3,3GHz a 3GB RAM, operační systém Windows XP. Algoritmy lokalizovaly vždy 100000 bodů, přičemž šlo jak o pozitivní, tak negativní testy. Negativní testy se objevují především na datech s menším počtem vrcholů, tedy výsledky mohou být trochu větší, než by

bylo jen v případě pozitivních testů. Vzhledem k tomu, že účelem těchto testů je spíše poukázat na trend doby běhu algoritmů, nepovažujeme vliv negativních testů za podstatný. Algoritmy náhodných procházek začínaly v náhodně vybraných stěnách, přičemž stěny byly vybrány dopředu a tedy čas potřebný pro výběr počáteční stěny není ve statistikách započítán. Testy opět proběhly na triangulacích a minimálních pseudotriangulacích z nich vzniklých pomocí metody *ET* [Tr07a] (stejná vstupní data jako v předchozích případech).

Všechny výše zmiňované testy byly provedeny na aplikaci příslušné k diplomové práci J. Trčky [Tr07a] a to i včetně testování procházky pomocí algoritmu *Remembering Stochastic walk* nad původní triangulací. Lze předpokládat, že nevýkonnostní statistické výsledky jsou shodné s implementací *Remembering Stochastic walk* dle podkapitoly 3.1. To ovšem nelze tvrdit o výkonnostních testech, protože způsob implementace je do jisté míry rozdílný. To je způsobeno především faktem, že celé pojetí aplikace příslušné k [Tr07a] (především použité datové struktury) je voleno tak, aby bylo optimalizováno pro využití k lokaci v pseudotriangulacích. V následující tabulce (*Tab. 7.4*) je proto pro porovnání zobrazeno i měření času algoritmu *Remembering Stochastic walk* v podkap. 2.4 implementovaného dle 3.1.

Data	Vrcholů	Délka běhu procházky [ms]			
		Triang (2.4) dle (3.1)	Triang	Pseudo 1	Pseudo 2
Random rovnoměrný	100	498,5	601,9	572,0	531,4
	500	1120,6	1365,2	1184,3	1142,3
	1000	1481,2	1834,2	1624,9	1546,2
	5000	3412,4	5447,1	4700,7	4516,9
	10000	4771,9	9040,1	8085,9	7702,5
	50000	12884,1	28108,9	24380,9	24169,3
	100000	19512,4	40505,1	35867,7	36753,5
Grid	1000	1454,6	1911,3	1799,0	1455,2
	10000	4894,1	8839,2	8524,1	7817,9
	100000	19359,4	39177,9	38330,9	36463,6
Reálná data	4897	3885,8	5655,8	4895,3	4687,2
	15820	7121,5	23274,6	19671,3	20823,1
	70433	17243,8	33583,2	28637,9	28907,4

Tabulka 7.4 – Délka běhu procházky (průměr pro 100000 bodů)

Z tabulky (*Tab. 7.4*) je patrné, že implementace lokace bodu pomocí procházky v původní triangulaci (sloupec *Triang*) je pomalejší než lokace bodů v pseudotriangulacích vzniklých z této triangulace (viz podkapitoly 6.4 a 6.5). Výsledek to na první pohled není příliš podivný, protože procházka

v pseudotriangulaci je kratší (navštíví se méně pseudotrojúhelníků v pseudotriangulaci než trojúhelníků v triangulaci a také se v případě pseudotriangulace provede méně testů hran). Ovšem podezřelý se zdá být tento výsledek u triangulací (respektive pseudotriangulací) vzniklých nad body rozmístěnými ve čtvercové síti (*grid*). Vzhledem k tomu, že pseudotriangulace nad takovýmito daty je téměř totožná s původní triangulací, očekával bych u lokačních algoritmů pro pseudotriangulace horší výsledky než u lokačního algoritmu pro triangulaci, protože algoritmy dle podkapitol 6.4 a 6.5 provádějí nezanedbatelné množství testů navíc.

Na výsledcích algoritmu *Remembering Stochastic walk* podle 2.4 implementovaného dle 3.1 se tato skutečnost také potvrzuje. Dokonce je procházka v triangulaci podle očekávání výrazně rychlejší. To je způsobeno především použitím mnohem jednodušších datových struktur, ale i dalších výhodných vlastností triangulace, které pseudotriangulace nemá. To že je dle [Tr07a] rychlejší lokace bodu v pseudotriangulaci je způsobeno pravděpodobně tím, že autor nejdříve implementoval lokaci bodu v pseudotriangulaci a následnou lokaci v triangulaci implementoval na již vytvořené (nejspíše jen mírně upravené) struktury, které lépe vyhovují požadavkům lokace bodů v pseudotriangulacích.

Pro lokace bodů v triangulaci lze použít i rychlejší algoritmy než *Remembering Stochastic walk* (např. *Orthogonal walk*), proto budou lokace bodů v triangulacích vždy výrazně rychlejší než lokace bodů v odpovídajících pseudotriangulacích.

Z tabulky (*Tab. 7.4*) dále můžeme vyčíst skutečnost, že náhodná procházka *Pseudo 1*, ačkoli provádí více testů, je nepatrně rychlejší než *Pseudo 2*. Tuto skutečnost můžeme přičíst složitější implementaci *Pseudo 2*, kde menší počet testů je vyvážen složitější rozhodovací logikou. Všechny zmiňované testy byly provedeny bez použití algoritmu *výběru počátečního (pseudo)trojúhelníku* (viz podkapitola 2.13), aby byly lépe vzájemně porovnatelné.

8. Závěr

Lokace bodů pomocí procházkových algoritmů je velice zajímavou problematikou poskytující celou řadu různých možností. Obrovskou výhodou procházkových algoritmů je, že můžou být přímo začleněny do všech možných aplikací téměř bez změny ve strukturách. Navíc jejich implementace je přímočará a v mnohých variantách i poměrně snadná. Nesmíme opomenout ani fakt, že v případě, že jsme limitováni paměťovými požadavky, jednoznačně vyhrávají procházkové algoritmy, které nekladou žádné dodatečné paměťové nároky.

Jako nejlepší algoritmus pro lokace v triangulacích se jevila vlastní verze algoritmu *pravoúhlé procházky* (viz *Orthogonal walk* podkapitola 2.12) kombinovaná se stochastickou verzí algoritmu *Remembering walk* (podkap. 2.4), kde je použit *Shewchukův znaménkový test* (podkap. 2.15). Pro tetrahedronové sítě se jevila lepší stochastická verze algoritmu *Remembering walk* (podkap. 4.4), protože i když je nepatrně pomalejší, lze ji využít na všechny druhy tetrahedronových sítí. Algoritmy pro pseudotriangulace se ukázaly spolehlivé, ovšem znatelně pomalejší než algoritmy procházky v triangulacích.

S ohledem na velmi dobré výsledky některých procházkových algoritmů bych je v budoucnu velmi rád porovnal s lokačními algoritmy s logaritmickou složitostí (algoritmy využívající speciální datové struktury) a diskutoval vhodnost použití jednotlivých algoritmů na různé praktické aplikace. Vzhledem k dobrým výsledkům je v současné době mým názorem, že do určitého objemu dat budou procházky z časového hlediska dokonce výhodnější než některé lokační algoritmy s logaritmickou složitostí. V budoucnu bych chtěl i určit, do jaké míry je moje mínění pravdivé a případně zjistit i velikost vstupních dat, od které je přibližně výhodnější použití lokačních algoritmů s logaritmickou složitostí.

Použité zdroje

[Ar03] L. Arge, A. Danner, S. Teh: I/O-efficient point location using persistent B-trees, *Journal of Experimental Algorithmics (JEA)*, 2003

[Bh01] B. Bhattacharya, A. Mukhopadhyay, G. Narasimhan: Optimal Algorithms for Two-Guard Walkability of Simple Polygons, *Algorithms and Data Structures: 7th International Workshop*, 2001

[Ch03] B. Chazelle, D. Liu, A. Magen: Sublinear Geometric Algorithms, *ACM Symphosium on Theory of Computing, San Diego*, 2003

[Co04] K. Conroy: JOGL – A Beginner's Guide and Tutorial, 2004

[Cz06] A. Czumaj, Ch. Sohler: Sublinear-time Algorithms, *Bulletin of the EATCS*, 2006

[Da06] I. A. Darwin: Java – Kuchařka programátora, *Computer Press*, 2006

[De98] L. Devroye, E. P. Mucke, Binhai Zhu: A Note on Point Location in Delaunay Triangulations of Random Points, *Algorithmica 22*, 1998

[De02] O. Devillers, S. Pion, M. Teillaud: Walking in a Triangulation, *International Journal of Foundations of Computer Science*, 2002

[Ep05] D. Eppstein, M. T. Goodrich, J. Z. Sun: The Skip Quadtree - A Simple Dynamic Data Structure for Multidimensional Data, *21st Annual Symposium on Computational Geometry*, 2005

[Fl91] L. D. Floriani, B. Falcidieno, G. Nagy, C. Pienovi: On sorting triangles in Delaunay tessellation, *Algorithmica 6*, 1991

[Fo93] S. Fortune, C. J. Van Wyk: Efficient Exact Arithmetic for Computational Geometry, *Proceedings of the 9th Annual Symposium on Computational Geometry*, 1993

[Fo96] S. Fortune, C. J. Van Wyk: Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry. *To appear in Transactions on Mathematical Software*, 1996

[Go97] M. T. Goodrich, M. Orletsky, K. Ramaiyer: Methods for Achieving Fast Query Times in Point Location Data Structures, *Symposium on Discrete Algorithms: Proceedings of the 8th annual ACM-SIAM*, 1997

[He01] P. Herout: Grafické uživatelské prostředí a čeština, *Kopp České Budějovice*, 2001

[He03a] P. Herout: Učebnice jazyka Java, *Kopp České Budějovice*, 2003

[He03b] P. Herout: Bohatství knihoven, *Kopp České Budějovice*, 2003

[Ia01] J. Iacono: Optimal Planar Point Location, *Symposium on Discrete Algorithms: Proceedings of the 12th annual ACM-SIAM*, 2001

[Ja01] The Java Tutorial, *http://java.sun.com/docs/books/tutorial*

[Ja02] JDK 5.0 Documentation

[Ja03] Java 2 Platform Standard Edition 5.0 – API Specification

[Jo01] Java OpenGL (JOGL) 1.1.1 – API Specification

[Ka96] W. Kahan: Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, *IEEE – The world's leading professional association for the advancement of technology*, 1996

[Ka05] M. Kallmann: Path Planning in Triangulations, *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005

[Ko05] I. Kolingerová: Mariály k předmětům KIV/PRO a KIV/VAM, 2005

[Ko06] I. Kolingerová: A Small Improvement in the Walking Algorithm for Point Location in a Triangulation, *22nd European Workshop on Computational Geometry*, 2006

[Li99] Sheng Liang: Java Native Interface – Programmer's Guide and Specification, *Sun Developer Network*, 1999

[Li07] Yong Li and Jun-Hai Yong: Efficient Exact Arithmetic over Constructive Reals, *The 4th Annual Conference on Theory and Applications of Models of Computation*, 2007

[Mu99] E. P. Mücke, I. Saias, B. Zhu: Fast Randomized Point Location Without Preprocessing in Two and Three dimensional Delaunay Triangulations, *Computational Geom.: Theory & Applications*, 1999

[Pu07] V. Purchart: Deformace terénu pro virtuální realitu – datová, logická a vizualizační část modelu, *Bakalářská práce*, 2007

[Ro97] E. M. Rosen: Excel Solutions to the Chemical Engineering Problem Set, *ASEE Chemical Engineering Summer School held*, 1997

[Ro02] G. Rote, C. A. Wang, L. Wang, Y. Xu: On Constrained Minimum Pseudotriangulations, *Manuscript FU-Berlin*, 2002.

[Ru06] R. Rubinfeld: Sublinear time Algorithms, 2006

[Sh96a] J. R.Shewchuk: Robust Adaptive Floating-Point Geometric Predicates, *Proceedings of the 12th Annual Symposium on Computational Geometry*, 1996

[Sh96b] J. R.Shewchuk: Routines for Arbitrary Precision Floating-point Arithmetic and Fast Robust Geometric Predicates (zdrojový kód), *Public domain of J. R. Shewchuk*, 1996

[Sh97] J. R.Shewchuk: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, *Discrete & Computational Geometry 18*, 1997

[Sp02] B. Spell: Java – Programujeme profesionálně, *Computer Press*, 2002

[Te05] L. Tesková: Lineární algebra, *Západočeská univerzita Plzeň*, 2005

[Tr07a] J. Trčka: Pseudo-triangulace a jejich využití v aplikované výpočetní geometrii, *Diplomová práce*, 2007

[Tr07b] Trčka, I. Kolingerová: The Stochastic Walk Algorithm for Pseudotriangulations, 2007

[Vo07a] T. Vomáčka: Detekce a zpracování topologických událostí v Delaunayově triangulaci nad kinetickými daty, *semestrální práce k předmětu KMA/GPM*, 2007

[Vo07b] T. Vomáčka: Dynamická Delaunayova triangulace ve 2D pro kinetická data, *semestrální práce k předmětu KIV/VAM*, 2007

[Vo08a] T. Vomáčka: Delaunay Triangulation of Moving Points in Plane, *diplomová práce*, 2008

[Vo08b] T. Vomáčka: Delaunay Triangulation of Moving Points, *Central European Seminar on Computer Graphics for students*, 2008

[We98] F. Weller, On the Total Correctness of Lawson's Oriented Walk Algorithm, *10th Canadian Conf. Computational Geometry*, 1998
[Wi01] Wikipedia, *www.wikipedia.org*
[Ze07] M. Zemek, I. Kolingerová: Použití Power diagramů a regulárních triangulací pro hledání tunelů v molekulách proteinů, *Studentská Vědecká Konference – Plzeň*, 2007
[Zh03] H. Zhang, X. He: On Even Triangulations of 2-Connected Embedded Graphs, *Computing and Combinatorics: 9th Annual International Conference*, 2003

Přílohy

A Testy 2D procházkových algoritmů

Algoritmus	počet vrcholů triang. (*n*)	Bez *výběru počátečního trojúhelníku*				Optimální počet *výběrů počátečního trojúhelníku*			
		Testy hran		Délka cesty		Testy hran		Délka cesty	
		Ø počet	max	Ø počet	max	Ø počet	max	Ø počet	max
Remembering walk	100	22,98	56	11,67	31	13,68	46	6,48	24
	200	31,89	79	17,00	43	17,25	59	8,68	33
	500	49,35	118	26,83	69	21,93	96	11,22	48
	1000	68,32	191	37,53	100	28,38	105	14,92	61
	2000	91,31	244	51,68	133	33,91	152	18,12	83
	5000	149,19	372	84,51	209	45,11	200	24,48	109
	10000	208,19	517	118,01	286	58,09	279	31,90	158
	20000	291,98	741	166,14	420	73,72	300	40,85	173
	50000	440,13	1217	252,05	691	99,81	429	55,82	248
	100000	630,15	1616	361,40	929	124,50	495	69,99	290
	200000	902,76	2316	519,01	1349	159,81	852	90,38	491
	500000	1450,67	3661	835,57	2094	219,98	1010	125,03	557
	1000000	1988,65	4926	1147,08	2838	281,05	1066	160,26	620
	2000000	2752,30	7020	1587,43	4036	362,23	1356	207,16	801
Remembering Stochastic walk	100	22,14	61	11,58	32	13,16	45	6,48	24
	200	30,76	91	16,47	44	15,03	52	7,51	27
	500	47,27	127	26,03	69	18,56	69	9,50	37
	1000	66,87	167	37,42	92	22,87	85	11,96	52
	2000	89,27	225	50,38	128	27,93	106	14,85	61
	5000	142,57	383	81,45	221	35,73	151	19,31	83
	10000	199,52	508	114,54	292	43,98	190	24,04	101
	20000	280,01	692	161,57	408	54,88	230	30,36	133
	50000	440,26	1074	255,11	620	73,89	296	41,41	167
	100000	660,15	1646	383,72	940	93,10	332	52,59	192
	200000	859,95	2276	500,40	1348	115,72	476	65,69	272
	500000	1412,87	3558	823,50	2072	157,95	633	90,32	367
	1000000	1985,81	4790	1158,34	2822	201,23	905	115,50	525
	2000000	2858,53	6870	1668,67	3999	255,02	887	147,01	511
Flag Fast walk	100	22,13	48	18,68	39	17,25	40	13,75	34
	200	27,13	61	23,76	52	18,94	47	15,51	38
	500	37,46	113	33,85	77	22,47	60	18,94	54
	1000	48,57	133	45,06	108	25,76	71	22,23	66
	2000	63,82	193	59,99	135	30,00	93	26,50	86
	5000	92,35	255	88,58	223	37,95	143	34,39	131
	10000	125,92	458	121,89	303	45,50	162	41,88	139
	20000	174,67	587	169,95	419	54,03	246	50,40	197
	50000	272,64	927	268,58	639	71,84	283	68,09	281
	100000	368,41	970	363,99	967	89,29	426	85,45	422
	200000	540,49	1570	535,41	1330	110,89	406	107,13	397
	500000	843,54	2882	835,48	2089	149,82	617	145,82	561

	1000000	1195,01	3137	1189,14	2977	189,43	1058	185,36	774
	2000000	1663,86	4457	1656,29	4262	237,86	1033	233,85	1030
Flag Fast walk (Stochastic)	100	22,62	51	19,22	45	17,39	43	14,02	37
	200	27,74	61	24,34	55	19,17	56	15,83	42
	500	38,17	113	34,59	78	22,13	70	18,69	53
	1000	47,23	151	43,64	102	25,35	82	21,93	60
	2000	64,01	178	60,41	145	28,71	100	25,27	73
	5000	93,20	327	89,07	211	35,36	120	31,80	118
	10000	127,92	393	123,79	302	41,32	181	37,74	143
	20000	178,21	554	173,62	405	49,99	192	46,29	190
	50000	268,69	958	263,07	641	64,81	296	60,83	225
	100000	366,60	1384	358,25	952	78,14	302	74,15	269
	200000	544,20	2095	534,56	1294	96,11	551	91,97	314
	500000	825,24	3429	810,18	2057	129,48	507	124,94	502
	1000000	1163,36	4417	1147,96	2983	161,40	633	156,49	630
	2000000	1641,25	5779	1622,75	3979	205,27	1416	199,53	826

Tabulka A.1 – Kompletní testy procházek dle viditelnosti

B Testy Výběru počátečního tetrahedronu

počet vrcholů sítě (*n*)	**Remembering walk s použitím Shewchukovy knihovny**												
	0	**1**	**2**	**3**	**4**	**5**	**6**	**8**	**10**	**15**	**20**	**30**	**40**
100	269	242	229	224	222	221	219	225	227	N/A	N/A	N/A	N/A
200	336	293	274	266	260	256	256	255	256	N/A	N/A	N/A	N/A
500	453	389	357	335	327	319	315	307	309	N/A	N/A	N/A	N/A
1000	570	482	436	415	395	384	373	369	364	N/A	N/A	N/A	N/A
2000	715	605	545	516	490	475	459	445	435	435	N/A	N/A	N/A
5000	1005	837	760	710	680	657	640	619	609	602	N/A	N/A	N/A
10000	1267	1080	977	917	876	843	819	790	775	769	787	N/A	N/A
20000	1616	1381	1244	1160	1105	1073	1030	988	966	946	956	N/A	N/A
50000	2314	1969	1762	1634	1544	1475	1427	1353	1306	1258	1260	N/A	N/A
100000	3010	2537	2263	2091	1969	1882	1807	1698	1639	1549	1522	1558	N/A
200000	3851	3217	2852	2634	2476	2341	2258	2107	2015	1880	1840	1846	N/A
500000	5352	4425	3899	3566	3345	3163	3021	2842	2691	2489	2398	2351	2399

počet vrcholů sítě (*n*)	**Remembering Stochastic walk**												
	0	**1**	**2**	**3**	**4**	**5**	**6**	**8**	**10**	**15**	**20**	**30**	**40**
100	143	129	128	126	127	127	130	138	145	N/A	N/A	N/A	N/A
200	180	160	151	147	147	146	149	154	161	N/A	N/A	N/A	N/A
500	241	209	195	187	182	180	182	181	188	N/A	N/A	N/A	N/A
1000	300	257	235	225	218	218	217	215	218	N/A	N/A	N/A	N/A
2000	379	323	293	281	274	266	263	260	264	275	N/A	N/A	N/A
5000	536	467	432	411	397	389	384	381	381	400	N/A	N/A	N/A
10000	721	635	587	564	534	520	514	509	501	526	561	N/A	N/A
20000	972	843	774	727	700	674	666	655	644	667	705	N/A	N/A
50000	1410	1229	1108	1023	975	945	914	887	867	869	900	N/A	N/A
100000	1897	1619	1447	1337	1274	1218	1176	1123	1096	1069	1081	1168	N/A
200000	2485	2126	1865	1727	1624	1548	1496	1402	1372	1316	1322	1395	N/A
500000	3481	2899	2587	2368	2222	2106	2031	1908	1837	1736	1715	1747	1852

počet vrcholů sítě (n)	Remembering Stochastic walk s použitím Shewchukovy knihovny												
	0	1	2	3	4	5	6	8	10	15	20	30	40
100	297	268	251	241	238	236	232	235	238	N/A	N/A	N/A	N/A
200	371	325	302	286	278	272	268	269	270	N/A	N/A	N/A	N/A
500	501	422	387	364	349	342	337	328	326	N/A	N/A	N/A	N/A
1000	621	517	468	446	421	411	401	387	385	N/A	N/A	N/A	N/A
2000	799	659	593	554	526	507	497	474	464	459	N/A	N/A	N/A
5000	1100	907	816	763	726	700	681	649	638	632	N/A	N/A	N/A
10000	1408	1179	1065	1004	944	905	879	845	823	807	815	N/A	N/A
20000	1806	1535	1374	1276	1203	1158	1118	1067	1042	1018	1018	N/A	N/A
50000	2545	2131	1902	1749	1653	1573	1511	1435	1385	1325	1317	N/A	N/A
100000	3275	2732	2424	2229	2085	1986	1910	1797	1717	1626	1601	1621	N/A
200000	4164	3467	3066	2819	2632	2502	2392	2245	2140	1997	1938	1939	N/A
500000	5835	4752	4172	3825	3568	3373	3221	3078	2859	2658	2535	2479	2509

Tabulka B.1 – Testy *Výběru počátečního tetrahedronu* pro zbylé algoritmy

C Vzor formátu vstupních dat

C.1. 2D

```
10
0.80000 0.20000
0.50000 0.30000
0.10000 0.70000
0.00000 0.40000
0.20000 0.10000
0.50000 0.90000
0.00000 0.70000
0.00000 0.90000
0.30000 0.50000
0.70000 0.80000
11
  4   8   3
  8   2   3
  2   6   3
  6   2   7
  7   2   5
  2   8   5
  5   8   9
  8   1   9
  4   1   8
  9   1   0
  1   4   0
11
  1  -1   8
  2   0   5
 -1   1   3
  4  -1   2
  5  -1   3
  6   4   1
  7  -1   5
  9   6   8
  7   0  10
 10  -1   7
 -1   9   8
```

C.2. 3D

```
10
0.10000 0.20000 0.60000
0.30000 0.10000 0.40000
0.70000 0.30000 0.90000
0.70000 0.60000 0.40000
0.40000 0.30000 0.20000
0.10000 0.80000 0.30000
0.30000 0.00000 0.20000
0.10000 0.90000 0.70000
0.50000 0.00000 0.70000
0.70000 0.80000 0.50000
15
  0   1   4   6
  1   0   2   8
  1   6   0   8
  4   6   1   8
  1   4   8   3
  1   8   2   3
  3   6   4   8
  6   4   0   5
  3   0   5   7
  3   2   0   7
  3   0   4   5
  5   3   7   9
  7   3   2   9
  1   0   4   3
  1   2   0   3
15
  3   7   2  13
 -1   5   2  14
 -1   1   3   0
  2   4   6   0
  6   5  13   3
 -1  14   4   1
  3   4  -1  -1
 10  -1  -1   0
 -1  11   9  10
 -1   8  12  14
  7  -1   8  13
 12  -1  -1   8
 -1  -1  11   9
 10   4  14   0
  9  13   5   1
```

PROCHÁZKOVÉ LOKAČNÍ ALGORITMY

Roman Soukal[1], **Ivana Kolingerová**[2]

1 ÚVOD

Lokace bodu je velmi často řešenou úlohou v oblasti počítačové grafiky. I když je v následujícím textu nejčastěji popisována lokace bodů v triangulacích, stejně tak to může být i lokace bodů v tetrahedronových sítích nebo např. v pseudotriangulacích. Efektivní lokační algoritmy používají různorodé datové struktury (DAG, skip list, quadtree, datové struktury s náhodným vzorkováním a další). Algoritmy využívající tyto datové struktury dosahují očekávané složitosti $O(\log n)$ na jeden lokalizovaný bod, kde n je celkový počet vrcholů v triangulaci. Velikou nevýhodou těchto algoritmů je to, že mají poměrně velké paměťové nároky (zpravidla $O(n)$), a to je limitujícím faktorem u obrovských vstupních dat (milióny bodů). Někteří programátoři tak sahají k jinému paměťově úspornějšímu řešení – například k lokaci pomocí procházkových algoritmů.

Obecně pod pojem algoritmů procházky řadíme všechny algoritmy, které v triangulacích, Voronoiových diagramech, pseudotriangulacích či tetrahedronových sítích lokalizují bod pomocí průchodu přes jednotlivé sousedící prvky této sítě. Zmíněnou lokací rozumíme určení prvku, v kterém se hledaný bod nachází. Ačkoliv je tento přístup méně efektivní (s očekávanou složitostí $O(n^{1/3})$ až $O(n^{1/2})$ na jeden lokalizovaný bod), nepotřebuje žádné další datové struktury, a tedy nespotřebovává ani žádnou paměť navíc.

Strategie procházky znamená, že algoritmus zkoumá jeden prvek (trojúhelník, pseudotrojúhelník, tetrahedron atd.) za druhým, přičemž do dalšího prvku cestuje přes hranu aktuálního prvku a jako další prvek zvolí ten, který vyhovuje nejlépe kritériu pro přiblížení se k hledanému bodu. Počáteční prvek bývá volen náhodně nebo jak ukázal Mücke et. al. (1999) se zvolí z množiny náhodně vybraných prvků ten, který je k hledanému bodu nejblíže.

2 DĚLENÍ PROCHÁZKOVÝCH ALGORITMŮ

Procházka může vést přes všechny prvky, které protínají úsečku mezi vrcholem počátečního prvku a hledaným bodem, v takovém případě hovoříme o tzv. přímé procházce (Straight walk). Nebo cesta může být rozdělena podél os, kdy se algoritmus hledanému bodu blíží po jednotlivých souřadných osách. Tato procházka bývá nazývána pravoúhlá procházka (Orthogonal walk). Naproti tomu procházka dle viditelnosti (Visibility walk) používá test orientace hledaného bodu vůči stěnám prvku a podle výsledku testu (případně výsledků testů) se rozhoduje, do jakého sousedního prvku přejde.

3 PROVEDENÝ VÝZKUM

Byla implementována řada algoritmů pro procházky v rovinných triangulacích (algoritmy procházky přímé, pravoúhlé i dle viditelnosti) včetně procházek dle Devillers et. al. (2001) a

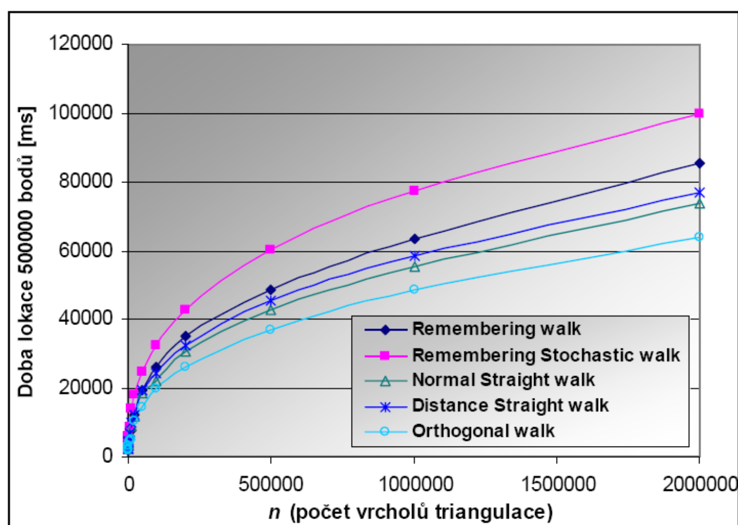
---

[1] Roman Soukal, student magisterského studijního programu Inženýrská informatika, obor Informatika a výpočetní technika, specializace Počítačová grafika a výpočetní systémy, email: romantic@students.zcu.cz.

[2] Doc. Dr. Ing. Ivana Kolingerová, ZČU v Plzni, FAV, Katedra informatiky, Univerzitní 22, 306 14 Plzeň, email: kolinger@kiv.zcu.cz, (vedoucí diplomové práce).

dle Kolingerová (2006), z nichž některé byly vlastní a některé byly výrazně upraveny pro zvýšení efektivity vyhledávání. Dále byly implementovány spolehlivé algoritmy procházky pro tetrahedronové sítě (algoritmy procházky dle viditelnosti, protože ostatní algoritmy se těžko vyrovnávají se spoustou možných singulárních případů typických pro E^3) ve verzi stochastické (algoritmus netestuje stěny v pořadí daném strukturou, ale pořadí stěn volí náhodně) i nestochastické (algoritmus je rychlejší, ale v určitých případech se může zacyklit). Byly zkoumány i algoritmy procházky v pseudotriangulacích dle Trčka et. al. (2006). Algoritmy byly testovány na spolehlivost a byly také výkonnostně porovnány s algoritmy procházky v triangulacích.

Všechny algoritmy byly důkladně otestovány: byla hledána optimální hodnota počtu výběrů počátečního prvku dle Mücke et. al. (1999), byla zjišťována očekávaná složitost těchto algoritmů, byly úspěšně testovány vlastnosti (spolehlivost, časy (výsledky pro E^2 viz Graf 1) apod.) implementovaných algoritmů nad strukturami vytvořených nad různými datovými sety (rovnoměrné rozložení bodů (E^2 i E^3), Gaussovo rozložení bodů (E^2), shluky(E^2), rastr (E^2), body rozmístěné na základě reálných dat (E^2 i E^3), body umístěné do oblouku (E^2).



Graf 1: Porovnání časů vybraných algoritmů pro E^2

4 ZÁVĚR

Jako nejlepší algoritmus pro triangulace se jevil vlastní algoritmus pravoúhlé procházky kombinovaný se stochastickým algoritmem procházky dle viditelnosti. Pro tetrahedronové sítě se jevila lepší stochastická verze algoritmu, protože i když je o trochu pomalejší, lze ji využít na všechny druhy tetrahedronových sítí. Algoritmy pro pseudotriangulace se ukázaly spolehlivé, ovšem pomalejší než algoritmy procházky v triangulacích.

LITERATURA

Devillers, O., Pion, S., Teillaud, M. , 2001. Walking in a Triangulation. *International Journal of Foundations of Computer Science.*

Mücke, E. P., Saias, I., Zhu, B., 1999. Fast Randomized Point Location without Preprocessing in Two- and Three-dimensional Delaunay Triangulations. *Computational Geometry: Theory and Applications*.

Kolingerová, I., 2006. A Small Improvement in the Walking Algorithm for Point Location in a Triangulation. *22nd European Workshop on Computational Geometry*.

Trčka, J., Kolingerová, I., 2008. The Stochastic Walk Algorithm for Pseudotriangulations. *Rukopis*.