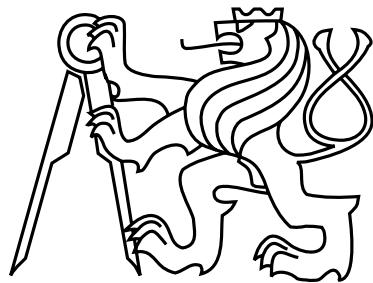


České vysoké učení technické v Praze  
Fakulta elektrotechnická  
Katedra počítačové grafiky a interakce



Bakalářská práce

## **Planární segmentace mračna bodů**

*Daniel Princ*

Vedoucí práce: Ing. David Sedláček

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimedia

24. května 2012



## **Poděkování**

Chtěl bych poděkovat vedoucímu této práce, panu Ing. Davidovi Sedláčkovi, za poskytnutí cenných rad a nápadů.



## Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 24. 5. 2015

.....



# Abstract

This bachelor's thesis concerns about point cloud processing, specifically about planar segmentation of point clouds. Planar segmentation finds its use in 3D digital reconstruction of objects from the real world. Currently, there has been significant interest to automate the process of data reconstruction, planar segmentation is one of the basic parts of this process.

Specifically, the goal of this thesis is to implement an algorithm, that will perform the segmentation as automatically as possible and in sufficient quality. The algorithm is implemented as a part of reconstruction software ArchiRec3D.

# Abstrakt

Tato bakalářská práce se zabývá problematikou zpracování mračna bodů, konkrétně segmentací na planární primitiva. Toto má uplatnění zejména při digitální 3D rekonstrukci objektů reálného světa. V současné době je snaha co nejvíce automatizovat proces zpracování naměřených dat, planární segmentace je jednou ze základních částí tohoto procesu.

Konkrétním cílem této bakalářské práce je implementovat algoritmus, který budou segmentaci provádět co nejvíce automaticky a zároveň v dostatečné kvalitě. Algoritmus je implementován jako součást 3D rekonstrukčního nástroje ArchiRec3D.



# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
1.1	Segmentace mračna bodů . . . . .	2
1.2	Struktura práce . . . . .	3
<b>2</b>	<b>Analýza a návrh řešení</b>	<b>5</b>
2.1	Segmentace na základě velikosti normálového vektoru . . . . .	5
2.1.1	Adaptivní válcová definice okolí . . . . .	5
2.1.2	Výpočet parametrů bodu . . . . .	7
2.1.3	Klastrování bodů . . . . .	9
2.2	Proložení roviny množinou bodů . . . . .	12
<b>3</b>	<b>Implementace</b>	<b>15</b>
3.1	Uložení bodů do akumulátorového pole . . . . .	15
3.2	Klastrování bodů v akumulátorovém poli . . . . .	23
3.3	Rekapitulace zadávaných parametrů a jejich použití . . . . .	28
3.4	Struktura a použití zdrojového kódu . . . . .	29
3.4.1	Struktura zdrojového kódu . . . . .	29
3.4.2	Ukázkové použití algoritmu . . . . .	31
<b>4</b>	<b>Testování</b>	<b>33</b>
4.1	Ověření funkčnosti na umělých datech . . . . .	33
4.2	Testování kvality algoritmů na reálných datech . . . . .	36
4.2.1	Vliv parametrů na výslednou segmentaci . . . . .	36
4.2.2	Testování časové a paměťové složitosti algoritmu . . . . .	41
4.2.3	Výsledky segmentace dodaných dat . . . . .	44
4.2.3.1	Dům 2 . . . . .	45
4.2.3.2	Stodůlky . . . . .	47
4.2.3.3	Červená Lhota . . . . .	49
4.2.3.4	Doudleby . . . . .	51
4.2.3.5	Faustův dům . . . . .	53
4.2.3.6	Langweil . . . . .	55
4.3	Shrnutí výsledků testování . . . . .	57
<b>5</b>	<b>Závěr</b>	<b>59</b>
<b>A</b>	<b>Seznam použitých zkratek</b>	<b>63</b>

**B Obsah přiloženého CD****65**

# Seznam obrázků

1.1	Ukázková scéna (nahoře) a její planární segmentace (dole).	2
2.1	Definice okolí bodu (boční pohled). (a) zobrazuje proložení původní roviny body uvnitř koule. Pro některé body je znázorněna jejich vzdálenost od roviny $d_i$ . (b) znázorňuje výslednou rovinu po iterativním procesu, který zahrnuje váhy jednotlivých bodů. Okolí tvoří modré vyznačené body uvnitř bufferu.	6
2.2	Diagram aktivit znázorňující proces definice okolí bodu.	7
2.3	Schématické znázornění situace, kdy jsou dvě roviny stejně vzdálené od bodu A, ale různě vzdálené od bodu B (a), zaznamenané hlasy v akumulátorovém poli (b). Vzdálenost roviny od bodu A je značena jako $d_A$ , od bodu B jako $d_B$ .	8
2.4	Ukázka akumulátorového pole (a), které reprezentuje mračno bodů se třemi kolmými rovinami (b). Svislá osa grafu reprezentuje počet bodů, vodorovné osy parametry bodů (tedy souřadnice v poli).	9
2.5	Prohledávání políček v akumulátorovém poli v průběhu jednotlivých iterací.	10
3.1	Ukázka vlivu velikosti okolí na vypočítávané parametry. Malé okolí může způsobit chybný výpočet (a), oproti tomu velké okolí je přesnější (b).	18
3.2	Diagram znázorňující získání potřebných nejbližších sousedů bodu.	19
3.3	Graf znázorňující relativní výkon (1 = nejvyšší výkon) různých knihoven při SVD dekompozici. Obrázek je převzat z [1].	21
3.4	Ukázka chyby, která může nastat při postupném vybírání bodů z akumulátorového pole, pokud dvě roviny vytvoří blízké vrcholy v akumulátorovém poli.	26
4.1	Ukázka testovací segmentace, která je v tabulce 4.1 označena jako 7-0. Vlevo je zobrazena výsledná segmentace, vpravo je detail na popsanou chybu při segmentaci.	35
4.2	Ukázka testovací segmentace, která je v tabulce 4.1 označena jako 6-0. Jedná se o stejné mračno z různých úhlů.	35
4.3	Ukázka mračna bodů dům 2 (a) a stodůlky (b).	36
4.4	Ukázka vybraných mračen bodů z testování. Obrázky odpovídají značení v tabulce: (a) d2-0.01, (b) d2-0.14, (c) d2-0.02, (d) d2-0.07.	38
4.5	Ukázka vybraných mračen bodů z testování z různých pohledů. Obrázky odpovídají značení v tabulce: (a) st-0.01, (b) st-0.02, (c) st-0.03, (d) st-0.06.	40
4.6	Graf zobrazující velikost využité paměti za běhu algoritmu.	41
4.7	Graf zobrazující velikost využité paměti při inicializaci algoritmu.	41

4.8	Graf zobrazující velikost využité paměti při spuštění algoritmu s různými parametry. . . . .	42
4.9	Graf zobrazující dobu běhu algoritmu v závislosti na zadaných parametrech. V grafu jsou znázorněna data z tabulky 4.4. . . . .	43
4.10	Segmentace mračna dům 2. (a) zobrazuje pouze segmentované body, (b) zobrazuje kromě segmentů i body, které nebyly segmentovány. . . . .	45
4.11	Segmentace mračna stodůlky. (a) zobrazuje první část segmentace, (b) zobrazuje celkovou segmentaci, (c) zobrazuje pohled na celkovou segmentaci shora, (d) zobrazuje celkovou segmentaci včetně bodů, které nebyly segmentovány. . . . .	47
4.12	Segmentace mračna stodůlky. (a) zobrazuje pouze segmentaci, (b) zobrazuje segmentaci včetně bodů, které nebyly segmentovány. . . . .	48
4.13	Ukázka mračna Červená Lhota. . . . .	49
4.14	Ukázka segmentace mračna Červená Lhota z různých úhlů. . . . .	50
4.15	Ukázka mračna Doudleby. . . . .	51
4.16	Ukázka segmentace mračna Doudleby. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány. . . . .	52
4.17	Ukázka mračna Faust. . . . .	53
4.18	Ukázka segmentace mračna Faustův dům. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány. . . . .	54
4.19	Ukázka mračna Langweil. . . . .	55
4.20	Ukázka segmentace mračna Langweil. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány. . . . .	56

# Seznam tabulek

3.1	Tabulka znázorňující rozdílné časy vybírání okolí z kd-stromu v závislosti na definovaném parametru. Čas je uváděn v sekundách, sloupec „Opak.“ určuje, pro jaké procento bodů bylo nutné z kd-stromu vybírat okolí více než jednu. Prázdná políčka jsem již nevyplňoval, protože předešlé výsledky byly dostatečně průkazné a další měření by bylo zbytečné. . . . .	20
3.2	Tabulka znázorňující časy výpočtu SVD různých knihoven. Časy jsou uvedeny v sekundách. Knihovny označené * jsou nastavené tak, aby při SVD počítaly pouze matici $V^T$ . . . . .	21
3.3	Tabulka zobrazuje rozdílné časy klastrování, pokud vyhledáváme nejbližší bod v kd-stromu nebo pokud počítáme průměrnou vzdálenost od všech bodů. . . . .	25
4.1	Tabulka s výsledky testování algoritmu na generovaných datech. P1 je parametr definující okolí, P2 parametr klastrování a P3 parametr určující kvalitu klastrování. . . . .	34
4.2	Tabulka zobrazuje výsledky testování algoritmu na mračnu dům 2 při různých parametrech. Parametry jsou popsány v příloze A. . . . .	37
4.3	Tabulka zobrazuje výsledky testování algoritmu na mračnu stodůly při různých parametrech klastrování. Parametry jsou popsány v příloze A. . . . .	39
4.4	Tabulka zobrazuje výsledky testování rychlosti algoritmu v závislosti na zadaných parametrech. Parametry jsou popsány v příloze A. . . . .	43
4.5	Tabulka zobrazuje informace o segmentaci mračna dům 2. Parametry jsou popsány v příloze A. . . . .	45
4.6	Tabulka zobrazuje informace o segmentaci mračna stodůlky. Parametry jsou popsány v příloze A. . . . .	47
4.7	Tabulka zobrazuje informace o segmentaci části mračna stodůlky. . . . .	48
4.8	Tabulka zobrazuje informace o segmentaci mračna bodů Červená Lhota. Parametry jsou popsány v příloze A. . . . .	49
4.9	Tabulka zobrazuje informace o segmentaci mračna bodů Doudleby. Parametry jsou popsány v příloze A. . . . .	51
4.10	Tabulka zobrazuje informace o segmentaci mračna bodů Faustův dům. Parametry jsou popsány v příloze A. . . . .	53
4.11	Tabulka zobrazuje informace o segmentaci mračna bodů Langewil. Parametry jsou popsány v příloze A. . . . .	55



# Kapitola 1

## Úvod

Rekonstrukce 3D modelů se dnes uplatňuje v mnoha oborech. Běžně se můžeme setkat např. s vizualizacemi, které se často využívají pro simulaci vzhledu města při výstavbě nových objektů. V takovém případě se obvykle naskenuje aktuální stav okolí výstavby, do kterého se přidá model plánované budovy. Výsledná vizualizace poté umožňuje posoudit vliv nové výstavby v kontextu okolí. Výhoda je také v tom, že takovéto simulace snadno pochopí i laický pozorovatel. Využití lze nalézt i v technické praxi, kde mohou 3D modely zachytit aktuální stav zařízení (např. potrubí v elektrárně), které nemusí odpovídat zastaralé či nekompletnej dokumentaci. Přesné 3D modely krajiny se mohou využívat např. při modelování záplav nebo šíření bezdrátového signálu. Podobně se modely mohou využívat k modelování šíření zvuku v uzavřených prostorech. Dále můžeme najít rozsáhlé využití v archeologii a ochraně kulturního dědictví, v dokumentaci důlních děl, monitorování krajiny pro detekci nebezpečných sesuvů, mapování pobřežních oblastí a mořského dna v okolí přístavů apod.

Existuje několik metod, které se ke sběru těchto dat využívají. Tradiční metodou je fotogrammetrie, kdy se informace o objektu získávají z několika fotografií a pro případné zpřesnění se může využít geodetického zaměření. Hlavní nevýhodou této metody je přesnost naměřených dat, která výrazně klesá se vzdáleností měření. Tato metoda nám obvykle poskytuje souřadnice charakteristických bodů, jako jsou hrany, vrcholy apod. To může být velká nevýhoda zejména u nepravidelných objektů.

Jednou z nejnovějších metod sběru dat je laserové skenování. Jeho hlavní výhodou je rychlý sběr velkého množství přesných dat v terénu. Nevýhodou oproti předchozí metodě je zejména špatná identifikace hran a vrcholů. Další nevýhodou může být fakt, že přístroje pro laserové skenování jsou velmi drahé. Slabinou této metody je také náročné zpracování, které se neobejde bez výkonné výpočetní techniky. Základním výstupem z laserového skenování je mračno bodů. Jedná se o tisíce až miliony bodů, které jsou definovány třemi kartézskými souřadnicemi ( $x$ ,  $y$ ,  $z$ ), dále mohou obsahovat informaci o barvě (r, g, b) a případně také normálu ( $n_x$ ,  $n_y$ ,  $n_z$ ) k ploše, na které se bod vyskytuje.

## 1.1 Segmentace mračna bodů

V této práci se zabývám zpracováním mračna bodů, konkrétně jeho planární segmentací. Cílem práce je tedy identifikovat ve vstupním mračnu bodů rovinné útvary, přiřadit jednotlivé body do těchto rovinných útvarů a případně zahodit body, které v žádné rovinně neleží (viz Obr. 1.1). Tato segmentace má smysl zejména v architektuře a archeologii, tedy při rekonstrukci budov a podobných objektů, které jsou složeny z relativně malého počtu velkých rovin (tedy stěny, střechy apod.).



Obrázek 1.1: Ukázková scéna (nahoře) a její planární segmentace (dole).

Segmentační metody můžeme hrubě rozdělit do dvou základních kategorií [8]. Jednak jsou to metody, které segmentují na základě vlastností jako je vzdálenost bodů v prostoru a případně podobnost lokálně odhadnutých normál. Sem spadají např. metody, jako je segmentace na základě skenovacích linií<sup>1</sup> nebo surface growing. První zmíněná metoda vychází z toho, že data jsou pořizována postupně podél skenovacích linií. Body v těchto liniích jsou nejprve rozděleny do rovných přímek a poté jsou ve 3D prostoru na základě podobných atributů slučovány do jednotlivých segmentů. Surface growing algoritmy fungují tak, že je vybrán rovinný či nerovinný prvek (seed region) a ten je poté postupně spojován s blízkými body, které mají podobné atributy. Tato metoda je ale silně závislá na volbě vhodného původního prvku.

---

<sup>1</sup>V anglické literatuře se označuje jako „scan line segmentation“ [8]

Do druhé kategorie spadají metody, které přímo odhadují parametry roviny na základě shlukování bodů a vyhledávání lokálních maxim v prostoru parametrů. Sem patří např. rozšíření Houghovi transformace pro 3D prostor. Jeden bod v mračnu, který se nachází na ploše v objektovém prostoru definuje rovinu v prostoru parametrů. Body na stejné ploše mají poté podobné parametry (vzdálenost od počátku a naklonění roviny) a na základě těchto parametrů jsou poté shlukovány. Jednotlivé shluky bodů pak podle dalších kritérií vytvoří jeden segment. Tato a další podobné metody mají zejména výpočetní problémy, protože mají velké paměťové nároky. Obvyklým problémem obou výše zmíněných kategorií je fakt, že jednotlivé metody se často zaměřují pouze na specifický typ vstupních dat (letecké skeny nebo pozemní skeny).

Článek [3] navrhuje řešení, které zohledňuje jak podobnost bodů v prostoru atributů, tak vzdálenost bodů v objektovém prostoru. Zároveň snižuje počet atributů pro vyšší efektivitu a menší paměťové nároky. Dále by také metoda neměla být závislá na typu vstupních dat.

## 1.2 Struktura práce

V této práci bych měl popsat a implementovat dva zadané algoritmy [3] a [6]. První algoritmus se ale v průběhu práce ukázal náročnější, než jsem předem očekával. Po dohodě s vedoucím práce je tedy v této práci popsán a implementován pouze tento první algoritmus.

Práce nejprve v kapitole 2 popisuje zadaný algoritmus [3] a věnuje se návrhu jeho implementace. Poté následuje kapitola 3 popisující samotnou implementaci algoritmu. Algoritmus je implementován v rámci dodaného nástroje ArchiRec3D [7], který je psaný v jazyku Java. V závěrečné části práce v kapitole 4 je otestována správná funkčnost algoritmu na uměle generovaných datech a na reálných datech je otestována kvalita algoritmu a jeho implementace. V závěru jsou shrnutы výsledky testování a analyzovány nedostatky algoritmu. Je zda také zhodnoceno, zda jsou nedostatky způsobeny návrhem algoritmu nebo jeho implementací.



## Kapitola 2

# Analýza a návrh řešení

### 2.1 Segmentace na základě velikosti normálového vektoru

Navržený proces segmentace v článku [3] zahrnuje tři hlavní kroky - definici okolí, výpočet atributů a klastrování bodů. Definice okolí bodu bere v úvahu 3D vzdálenost mezi body a tvar povrchu, na kterém se bod nachází. Parametry daného bodu jsou poté vypočteny na základě tohoto definovaného okolí. Pro nižší paměťové nároky jsou pro každý bod použity pouze dva parametry. Po vypočtení parametrů je provedeno klastrování bodů ležících ve stejné rovině, které zohledňuje jak podobnost v prostoru parametrů, tak vzdálenost bodů v prostoru.

#### 2.1.1 Adaptivní válcová definice okolí

Správná definice okolí bodu je zásadní podmínkou pro správnou funkčnost algoritmu, protože okolí bodu přímo ovlivňuje parametry, které jsou pro daný bod vypočteny. V tomto algoritmu je okolí definováno tak, že bere v úvahu vzdálenost bodů v prostoru, ale zároveň také tvar povrchu, na kterém se bod nachází.

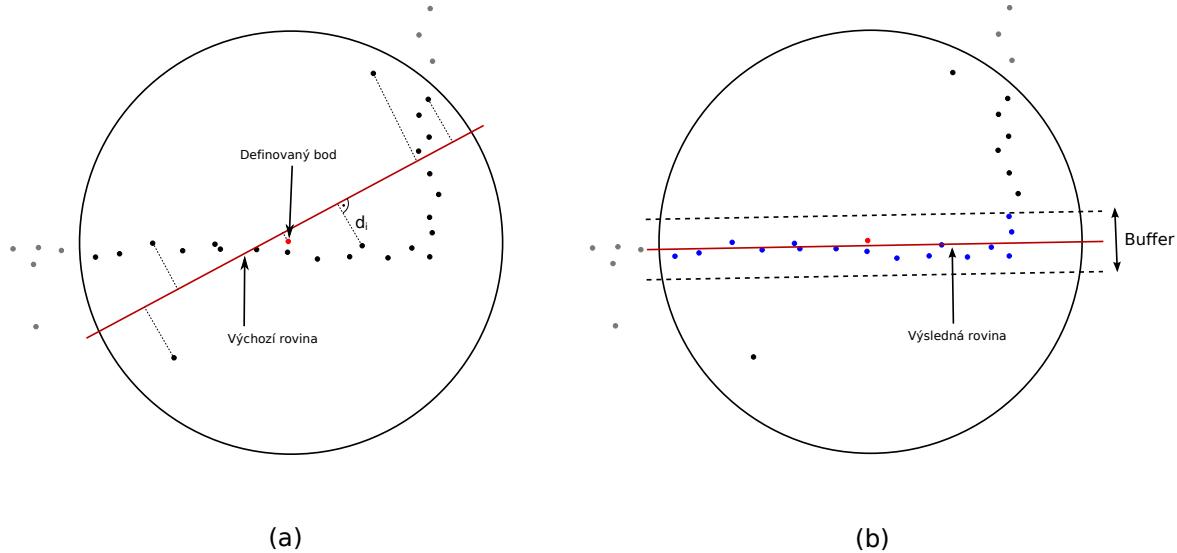
Proces definice okolí je znázorněn v diagramu na obrázku 2.2. Cílem tohoto procesu je získat lokální okolí zkoumaného bodu, konkrétně blízké sousední body ležící ve stejné rovině. Na základě toho okolí jsou poté pro každý bod vypočteny parametry, které slouží pro následné klastrování bodů. Prvním krokem při definici okolí je vytvořit kouli se středem v bodu, pro který okolí definujeme. Poloměr koule volíme takový, aby obsahovala dostatečný počet bodů pro spolehlivý výpočet parametrů. Tuto hodnotu nejde přesněji specifikovat, musí být odvozena z konkrétního mračna bodů, které zpracováváme<sup>1</sup>. Poté vezmeme všechny body, které leží uvnitř této koule, a proložíme je pomocí metody nejmenších čtverců výchozí rovinou (viz Obr. 2.1 (a)). Poté, co je rovina proložena, vypočítáme pro každý bod v kouli jeho vzdálenost od této roviny. Inverzi této vzdálenosti použijeme v další iteraci jako váhu bodu<sup>2</sup>, jak můžeme vidět v rovnici 2.1:

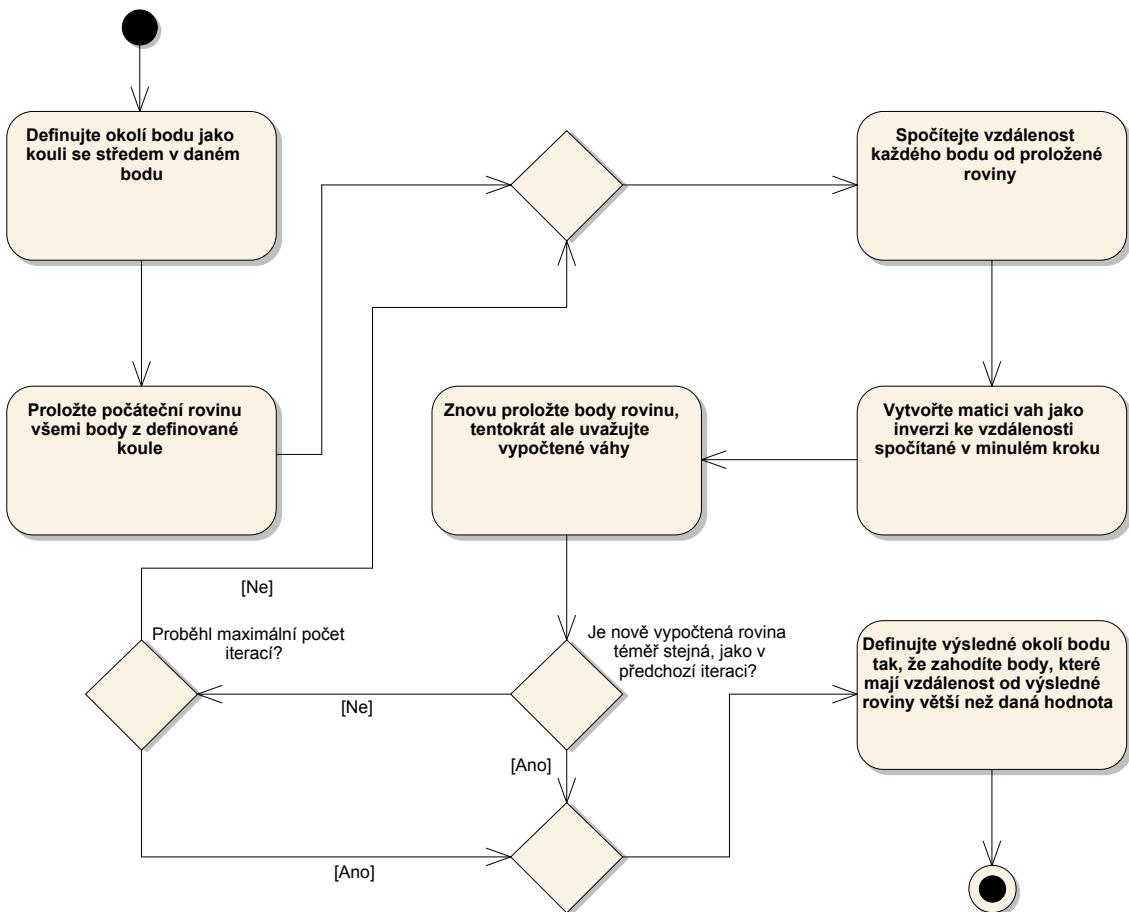
<sup>1</sup>Nicméně praxe ukazuje, že vhodná volba jsou řádově desítky až stovky bodů uvnitř koule.

<sup>2</sup>Jednoduše řečeno, čím je bod od roviny dál, tím je jeho váha nižší.

$$w_i = \frac{1}{d_i} \quad (2.1)$$

kde  $w_i$  značí váhu i-tého bodu a  $d_i$  je vzdálenost bodu od proložené roviny. Tím jsme pro každý bod v okolí získali váhu. Nyní okolí znova proložíme rovinou, ale vezmeme v úvahu vypočtené váhy. Tento proces „převažování roviny“ se iterativně opakuje, dokud se roviny v rámci iterací již nemění nebo dokud není dosaženo daného počtu iterací (např. 10). Poté je rovnoběžně nad a pod výslednou rovinou definován buffer. Velikost tohoto bufferu je závislá na očekávané velikosti šumu ve vstupních datech. Body, které jsou uvnitř tohoto bufferu, tvoří konečné okolí (viz Obr. 2.1(b)). Ostatní body mimo buffer se již na okolí nijak nepodílejí.





Obrázek 2.2: Diagram aktivit znázorňující proces definice okolí bodu.

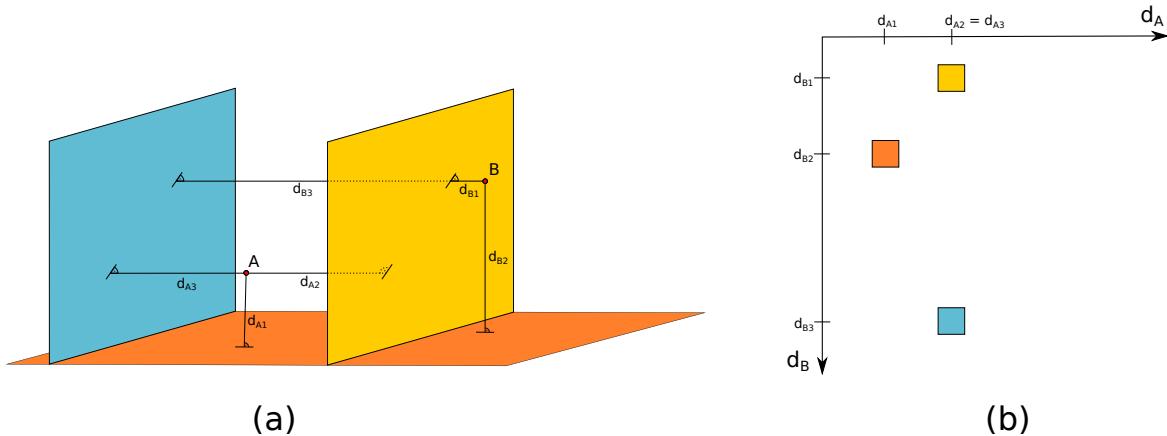
### 2.1.2 Výpočet parametrů bodu

Parametry bodu jsou, jak již bylo zmíněno, vypočítány na základě okolí bodu. Pokud zavedeme do mračna bodů nějaký referenční bod, můžeme definovat normálový vektor z tohoto bodu na rovinu, na které se daný bod nachází. Většina metod, které provádějí klastrování na základě vypočtených parametrů, využívá hlasování do akumulátorového pole vytvořeného v prostoru parametrů. Akumulátorové pole je v podstatě čítač, který si při hlasování ukládá počet hlasů na dané pozici. To nám umožňuje rychle vyhledat, která pozice má největší počet hlasů. Rozměry akumulátorového pole závisí na počtu použitých parametrů. Můžeme tedy použít 3 složky normálového vektora jako parametry a tím zajistíme, že všechny segmenty budou správně rozumnán<sup>3</sup>. Nicméně takovýto způsob vyžaduje vytvoření tříozměrného akumulátorového pole a hlasování do 3D pole je výpočetně náročná operace.

Pro snížení výpočetních nároků je tedy nutné snížit počet parametrů. Proto je jako

<sup>3</sup>Mělo by být jasné, že jeden normálový vektor jednoznačně definuje rovinu.

atribut využita velikost normálového vektoru<sup>4</sup>. Toto řešení přináší ale nebezpečí v tom, že může existovat více rovin ve stejné vzdálenosti od referenčního bodu a v takovém případě by několik rovin splynulo do jednoho segmentu. Jako řešení je zaveden druhý referenční bod a tedy i druhý parametr. Tím se značně snižuje riziko situace, kdy by více rovin mělo stejnou vzdálenost od dvou různých bodů. Obrázek 2.3 (a) schématicky znázorňuje situaci, kdy mají dvě roviny stejnou vzdálenost od jednoho bodu, ale různou vzdálenost od druhého bodu, každá rovina bude mít tedy jiný pár parametrů. Na obrázku 2.3 (b) jsou znázorněny hlasy jednotlivých rovin zaznamenané v akumulátorovém poli. Všechny body jsou v poli zaznamenané na základě těchto dvou vypočtených parametrů. Body, které patří do různých rovin, jsou tedy v akumulátorovém poli umístěny na různých místech. Hlavní výhodou tohoto řešení je tedy to, že jsme snížili počet parametrů definujících rovinu a tím také ubrali jeden rozměr akumulátorového pole.



Obrázek 2.3: Schématické znázornění situace, kdy jsou dvě roviny stejně vzdálené od bodu A, ale různě vzdálené od bodu B (a), zaznamenané hlasy v akumulátorovém poli (b). Vzdálenost roviny od bodu A je značena jako  $d_A$ , od bodu B jako  $d_B$ .

Důležité je tedy umístit oba referenční body tak, aby byly v datech rovnoměrně rozprostřeny. Pokud známe maximální a minimální hodnoty souřadnic v mračnu bodů, můžeme určit jejich polohu podle rovnice 2.2:

$$\begin{aligned} \text{referenční bod A} &= \begin{pmatrix} \min_x \\ \min_y \\ \min_z \end{pmatrix} + \frac{1}{3} \begin{pmatrix} \max_x - \min_x \\ \max_y - \min_y \\ \max_z - \min_z \end{pmatrix} \\ \text{referenční bod B} &= \begin{pmatrix} \min_x \\ \min_y \\ \min_z \end{pmatrix} + \frac{2}{3} \begin{pmatrix} \max_x - \min_x \\ \max_y - \min_y \\ \max_z - \min_z \end{pmatrix} \end{aligned} \quad (2.2)$$

<sup>4</sup>Tedy jde o vzdálenost roviny od referenčního bodu.

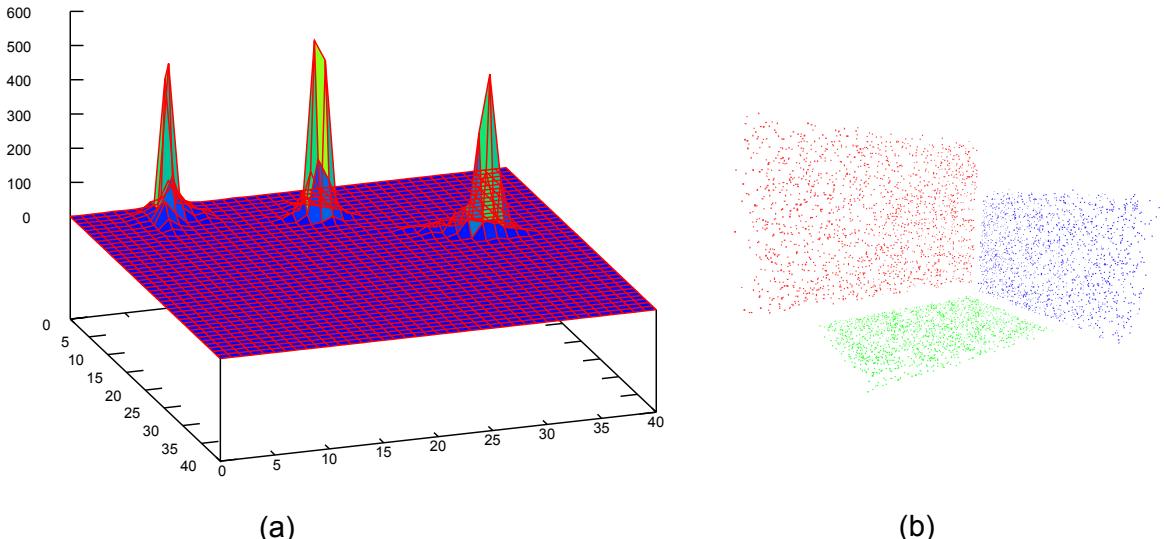
kde  $\min$  je bod v mračnu s nejmenšími souřadnicemi a  $\max$  s největšími souřadnicemi.

Umístěním referenčních bodů do těchto pozic značně snižujeme možnost, že se více rovin zobrazí na stejném místě v akumulátorovém poli. Nicméně pořád existuje možnost, že parametry dvou rozdílných rovin budou stejné nebo velmi podobné. Navíc v reálné situaci nebudou nikdy parametry všech bodů na jedné rovině stejné, ale budou v akumulátorovém poli rozptýlené ve více přilehlých políčkách. Tím se zvyšuje šance, že mohou v akumulátorovém poli dvě či více rovin zobrazit na blízké pozice. To by mohlo způsobit, že více rovin bude splynout do jediného segmentu. Proto nemůže být tato možnost ignorována a je potřeba data dále zpracovat. Tento proces detekce problému a jeho řešení je implementován v následující části, tedy v klastrování bodů.

Výpočet parametrů bodu probíhá konkrétně tak, že výsledným okolím bodu proložíme rovinu. Vypočítáme vzdálenost této roviny od obou referenčních bodů a tím získáme dvojici parametrů bodu. Na základě těchto parametrů je poté bod uložen do akumulátorového pole.

### 2.1.3 Klastrování bodů

Pokud máme vypočteny pozice obou referenčních bodů, můžeme spočítat parametry všech bodů v mračnu a uložit je do akumulátorového pole. Poté může být provedeno klastrování bodů. Body, které patří do různých rovin v objektovém prostoru, vytvoří různé vrcholy v prostoru parametrů. Na obrázku 2.4 je znázorněn příklad takové situace. Mračno bodů, které obsahuje tři kolmé roviny, vytvoří pomocí hlasování v akumulátorovém poli tři různé vrcholy.



Obrázek 2.4: Ukázka akumulátorového pole (a), které reprezentuje mračno bodů se třemi kolmými rovinami (b). Svislá osa grafu reprezentuje počet bodů, vodorovné osy parametry bodů (tedy souřadnice v poli).

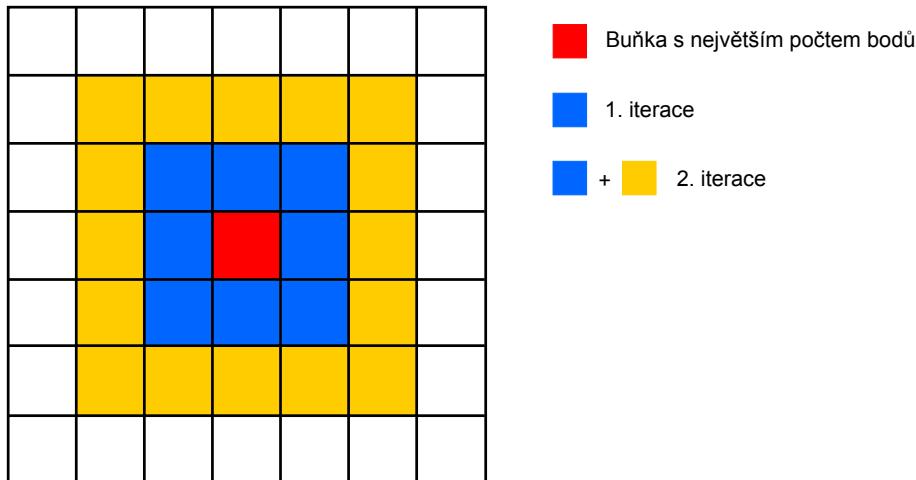
Pokud máme všechny body uloženy v akumulátorovém poli, vyhledáme buňku s největším počtem bodů (tedy nejvyšší vrchol). Velikost buňky v akumulátorovém poli je určena na

základě velikosti šumu ve vstupním mračnu. Tyto body z nejvyššího vrcholu proložíme rovinou pomocí metody nejmenších čtverců. Poté ohodnotíme kvalitu roviny výpočtem efektivní hodnoty (anglicky root mean square, dále jen RMS) vzdáleností všech bodů od proložené roviny. Pokud máme množinu  $n$  bodů  $\{x_1, x_2, \dots, x_n\}$  a jejich vzdálenosti od proložené roviny  $\{d_1, d_2, \dots, d_n\}$ , vypočítáme RMS podle vztahu 2.3:

$$RMS = \sqrt{\frac{1}{n} (d_1^2 + d_2^2 + \dots + d_n^2)} \quad (2.3)$$

Přijatelná hodnota je závislá na velikosti šumu v datech, obecně lze říct, že kvalita roviny je dostatečná, pokud je vypočtené RMS menší, než velikost šumu v datech. Pokud je RMS větší, pravděpodobně jsme detekovali výše popsaný problém, kdy více rovin splynulo v akumulátorovém poli do jednoho vrcholu. Řešení tohoto problému bude popsáno ke konci této sekce.

Pokud je tedy vypočtené RMS v přijatelných mezích, vytvoříme počáteční klastr, do kterého dáme všechny body dané buňky. Klastrování pokračuje body, které do této buňky nepatří, ale jsou v jejím okolí. Tento krok je nezbytný, protože data obvykle nebývají tak přesná, aby se všechny body z roviny trefily přesně na jedno místo v akumulátorovém poli, zejména jde o body v okolí hran a nerovností na povrchu. Tento postup probíhá iterativně a funguje tak, že v první iteraci je identifikováno 8 buněk v sousedství buňky s největším počtem bodů (viz Obr. 2.5).



Obrázek 2.5: Prohledávání políček v akumulátorovém poli v průběhu jednotlivých iterací.

Body z těchto osmi buněk přidáme do segmentu pouze, pokud splní dvě podmínky. První podmínkou je, že bod musí být prostorově blízko k bodům z původního klastru (např. 2x průměrná vzdálenost mezi body v původním klastru). Poté je spočítána vzdálenost bodů od roviny, která je definována body z původního klastru. Pouze body, které jsou k rovině blíže, než je předem daný práh, mohou být přidány do klastru. Tento práh je opět určen v závislosti na velikosti šumu v datech. Body, které nesplnily jednou z těchto dvou podmínek zůstanou na své pozici v akumulátorovém poli a jsou znova posouzeny v dalších iteracích. Po ukončení

každé iterace se přepočítají parametry klastru - tedy opět se body proloží rovina, ale jsou zahrnuty i nově přidané body. Poté proces pokračuje druhou iterací. V té jsou procházeny body z 24 sousedních buněk (viz Obr. 2.5). Proces je navržen tak, aby v této iteraci byly opět procházeny i ty body, které neprošly podmínkami v předchozích iteracích. Protože se na konci každé iterace přepočítávají parametry, mohou být tyto body v dalších iteracích do klastru zahrnuty. Takto iterativně proces pokračuje až do té doby, než do klastru nemohou být přidány žádné nové body. V tu chvíli je klasstru zaznamenán a všechny jeho body jsou odebrány z akumulátorového pole.

Nyní proces pokračuje k druhému nejvyššímu vrcholu v akumulátorovém poli, který je nalezen stejným způsobem, jako byl ten nejvyšší v předchozí iteraci. Poté se opakuje stejné kroky, jako v předchozím případě. Celý tento proces přesouvání se od nejvyššího vrcholu k druhému nejvyššímu pokračuje až do té doby, dokud je velikost nejvyššího vrcholu menší, než předdefinovaná hranice.

Poslední součástí algoritmu je řešení situace, kdy více rovin sdílí stejný vrchol v akumulátorovém poli. Tuto situaci jsme již detekovali pomocí výpočtu RMS. Pokud je hodnota vypočteného RMS příliš vysoká, s velkou pravděpodobností jsou ve vrcholu body z více různých rovin. V takovém případě jsou definovány dva nové referenční body. Pozice nových referenčních bodů je definovaná podle vztahu 2.4:

$$\begin{pmatrix} X_n \\ Y_n \\ Z_n \end{pmatrix} = \begin{pmatrix} X_o \\ Y_o \\ Z_o \end{pmatrix} + \begin{pmatrix} range \times rand \\ range \times rand \\ range \times rand \end{pmatrix} \quad (2.4)$$

kde  $(X_n, Y_n, Z_n)$  jsou souřadnice nového bodu,  $(X_o, Y_o, Z_o)$  jsou souřadnice původního bodu,  $range$  je hodnota určující jak nejdál může být nový bod posunut (např. 5m) a  $rand$  je náhodná hodnota z intervalu (0,1).

Poté je definováno nové akumulátorové pole pouze pro tyto body (způsobující problém) a na základě nově definovaných referenčních bodů jsou přepočteny parametry bodů. Poté jsou vyhledány nově vytvořené vrcholy a ty jsou klastrovány stejným způsobem, jako v běžném případě. Tento proces se opakuje do té doby, než jsou nalezeny dostatečně kvalitní roviny. V případě, že do určitého počtu iterací (např. 10) není nalezena žádná přijatelná rovina, vezme se druhá nejvyšší buňka ze zpracovávaného vrcholu v původním poli, proloží se rovinou a ohodnotí se její kvalita výpočtem RMS. Poté proces pokračuje standardně jednotlivými kroky, které již byly popsány. Tento proces by měl v praxi nastat jen výjimečně, obvykle situace splynutí dvou vrcholů vůbec nenastane a pokud ano, s velkou pravděpodobností bude vyřešena hned po změně pozic referenčních bodů.

Tímto jsme popsali celý algoritmus. Jak by mělo být patrné, algoritmus bere v úvahu vztahy bodů v prostoru parametrů a zároveň jejich vzdálenost a umístění v prostoru. Navíc je tento algoritmus poměrně robustní, protože při klastrování mají prioritu body, které mají podobnější parametry a zároveň jsou blízko v prostoru, před body, jejichž parametry jsou více odlišné.

## 2.2 Proložení roviny množinou bodů

Důležitou součástí algoritmu je proložení roviny množinou bodů pomocí metody nejmenších čtverců, viz [5]. Předpoklad tedy je, že chceme nalézt rovinu, která je co nejblíže od všech 3D bodů z dané množiny. Mějme množinu velikosti  $k$ , která obsahuje body  $(p_1, \dots, p_k)$ . Rovina je definována bodem  $c$ , který leží v rovině, a normálový vektorem  $n$ , který je na rovinu kolmý. Pro libovolný bod  $x$  ležící v rovině platí vztah  $n(x - c)^T = 0$ . Pokud vezmeme bod  $y$ , který v rovině neleží, potom platí  $n(y - c)^T \neq 0$ . Je tedy vhodné definovat odchylku podle vztahu 2.5:

$$(n(p_i - c)^T)^2 \quad (2.5)$$

Proložená rovina má být co nejblíže ke všem bodům. Jinými slovy hledáme takové  $n$ , které minimalizuje celkovou odchylku všech bodů. Rovinu tedy můžeme získat vyřešením následujícího vztahu:

$$\min_{c, \|n\|=1} \sum_{i=1}^k (n(p_i - c)^T)^2 \quad (2.6)$$

Řešení tohoto vztahu pro  $c$  je:

$$c = \frac{1}{k} \sum_{i=1}^k p_i \quad (2.7)$$

Bod  $c$  je tedy aritmetickým průměrem všech prokládaných bodů (pro tento bod se používá označení centroid). Pokud si definujeme matici  $M$ , která má v  $i$ -tého řádku hodnoty  $p_i - c$  (jde tedy o matici  $k \times 3$ ), můžeme vztah 2.6 přepsat do této formy:

$$\min_{\|n\|=1} \|Mn\|^2 \quad (2.8)$$

Řešením tohoto problému je SVD dekompozice matice  $M$ ,  $M = USV^T$ , kdy vektor  $n$  je určen pravým singulárním vektorem matice  $M$ , který odpovídá nejmenší singulární hodnotě. Konkrétně jde o poslední sloupec matice  $V^T$ .

Pokud tedy máme vypočtený bod  $c$  a získali jsme normálový vektor  $n$ , můžeme snadno vyjádřit rovinou např. obecnou rovnicí:

$$ax + by + cz + d = 0 \quad (2.9)$$

ve které parametry  $a, b, c$  jsou složky normálového vektoru  $n$  a  $d$  spočteme jednoduše dosazením bodu  $c$ :

$$d = -(n_1 c_1 + n_2 c_2 + n_3 c_3) \quad (2.10)$$

Toto řešení předpokládá, že všechny body mají stejnou váhu. Pokud máme pro body různé váhy, do výpočtu je zahrneme poměrně jednoduše. Pokud má bod  $p_i$  váhu  $w_i$ , musíme vztah 2.6 upravit tímto způsobem:

$$\min_{w_i, c, \|n\|=1} \sum_{i=1}^k w_i \times (n(p_i - c)^T)^2 \quad (2.11)$$

Stejně musíme upravit definici matice  $M$ , která bude v každém řádku obsahovat hodnoty  $w_i \times (p_i - c)$ . Ostatní postup zůstává stejný.



# Kapitola 3

## Implementace

Tato kapitola se zabývá popisem implementace algoritmu. Algoritmus je implementován v jazyce Java v rámci dodaného softwaru ArchiRec3D [7]. Kompletní zdrojový kód je k dispozici na přiloženém CD, viz kapitola B.

Implementaci algoritmu můžeme rozdělit na dvě hlavní části. První částí je výpočet parametrů pro všechny body a uložení bodů do akumulátorového pole. Druhou částí bude poté klastrování bodů v akumulátorovém poli. Tyto dvě části budou reprezentovány jako samostatné metody. První metoda bude mít na vstupu zpracovávané mračno bodů a jejím výstupem bude akumulátorové pole naplněné všemi body ze vstupního mračna. Druhá metoda bude mít toto akumulátorové pole na vstupu a výstupem budou body rozdělené do jednotlivých segmentů. Takovéto rozdělení má hlavní výhodu v tom, že obě části budou na sobě nezávislé. Bude tedy možné jednou uložit body do akumulátorového pole a poté provést klastrování několikrát s různými parametry bez nutnosti znova provádět první část.

### 3.1 Uložení bodů do akumulátorového pole

Tato část algoritmu bude procházet všechny body, na základě jejich okolí bude vypočítávat parametry, podle kterých budou body uloženy do akumulátorového pole. Vstupem bude tedy mračno bodů a výstupem akumulátorové pole. Prvním problémem bude tedy to, jak v paměti správě reprezentovat obě tyto struktury. To se samozřejmě musí odvíjet od operací, které budeme s daty provádět. Protože algoritmus implementují v rámci dodaného nástroje, mám k dispozici několik základních datových struktur jako např `vicitis.inner.Point3D` pro reprezentaci 3D bodu, `vicitis.inner.Plane3D` pro reprezentaci roviny apod., tyto datové struktury samozřejmě využiji. Navíc je zde také implementováno mnoho vhodných metod využívajících tyto třídy, takže je logické použít již funkční řešení a nevymýšlet vlastní, které by bylo navíc duplicitní.

Základním požadavkem na datovou strukturu reprezentující mračno bodů je možnost vyhledávat sousední body v 3D prostoru. Protože tuto operaci je nutné provádět pro každý bod, je důležité, aby byla co nejrychlejší. Naopak vytváření struktury se bude provádět pouze jednou a už do ní nebudeme za běhu přidávat ani odebírat body, takže na tyto operace nemusí být kladený velké nároky. Pro tyto účely se nejčastěji využívají stromy, které umožňují

rekurzivní dělení n-dimenzionálního prostoru na menší podprostory. Jedním z takovýchto algoritmů je octree, který rekurzivně dělí prostor na 8 podprostorů. Podobný algoritmus je kd-strom [4], kde uzly jsou k-dimenzionální body. Tento typ stromu dělí prostor na podprostory podél jednotlivých dimenzí. Pokud srovnáme tyto dva algoritmy, tak octree se rychleji staví a zároveň umí jednoduše přidávat nové body do již postaveného stromu. Oproti tomu do kd-stromu se body za běhu přidávají mnohem složitěji, ale algoritmus je efektivnější pro vyhledávání bodů a nejbližších sousedů. Protože jsem určil rychlé vyhledávání nejbližších sousedů jako nejdůležitější kritérium, vybral jsem nakonec pro reprezentaci mračna bodů kd-strom. Vyhledání nejbližšího souseda v kd-stromu (o  $n$  prvcích a  $k$  dimenzích) má v nejlepším případě asymptotickou složitost  $O(\log n)$  a v nejhorším případě  $O(k \times n^{1-\frac{1}{k}})$ , ve většině případů má však složitost blíže k nejlepšímu případu. Podrobný popis kd-stromu je k dispozici v [4]. Samozřejmě implementace kd-stromu je poměrně složitá, obzvláště, pokud má být co nejfektivnější. Proto jsem se rozhodl pro použití knihovny. Po vyzkoušení několika různých knihoven byla jednoznačně nejrychlejší implementace [9], kterou jsem použil.

Pro akumulátorové pole potřebujeme nějaké dvourozměrné pole, které nám bude udržovat body s danými parametry na jednotlivých pozicích v poli. Bylo by možné použít obyčejné `int [][]` pole, které by pouze ukládalo počet bodů na jednotlivých políčkách. Takové řešení by na první pohled mohlo pro vyhledávání vrcholů při klastrování dostatečné, nicméně by bylo výpočetně náročné zpětně zjistit, které body na dané pozici jsou. Proto bude jednodušší, když budou body uložené přímo v akumulátorovém poli. Proto jsem zvolil 2D pole seznamů `ArrayList<Point3D>[][]`, které bude udržovat body na jednotlivých políčkách pole. Taktéž budu při vyhledávání vrcholů v poli jednoduše vědět, kolik bodů se na daném políčku nachází, ale zároveň půjde tyto body ihned zpracovávat bez dalšího vyhledávání.

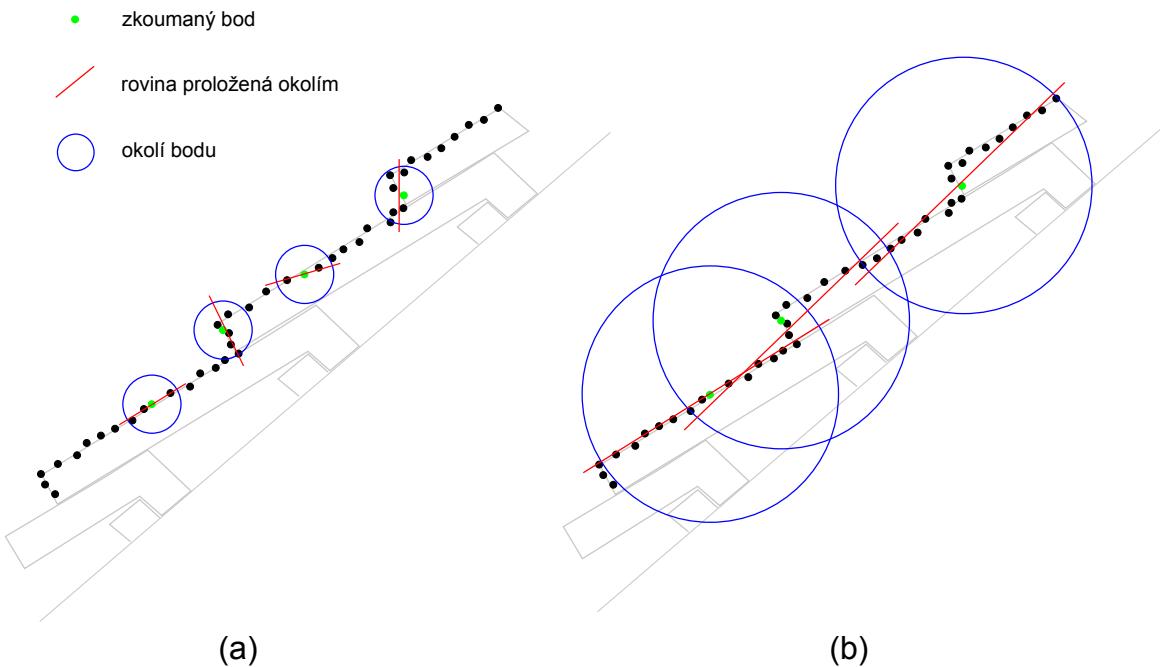
Velmi důležité jsou také rozměry akumulátorového pole. Podle teorie je velikost políčka v akumulátorovém poli určena na základě velikosti šumu ve vstupním mračnu. Bohužel testovací data, která mám k dispozici, nemají šum definovaný. Proto je nutné vymyslet vlastní strategii pro určení velikosti akumulátorového pole. Pokud je akumulátorové pole příliš malé, jednotlivé vrcholy mohou splynout, nepůjde je v poli detekovat a algoritmus prakticky nebude funkční. Pokud naopak bude akumulátorové pole příliš velké, vrcholy v poli nebudou výrazné a výsledek bude stejný jako v předchozím případě - algoritmus nebude fungovat tak, jak by se od něj čekalo. Původně jsem vycházel z úvahy, že velikost políčka v akumulátorovém poli bude např. jedna tisícina z velikosti mračna bodů. V takovém případě je ale velikost pole pro všechny vstupní data stejná (např.  $1000 \times 1000$ ). To ale není vhodné řešení, protože vstupní data mohou obsahovat řádově tisíce až miliony bodů a nezdá se logické, aby pro oba případy bylo akumulátorové pole stejně velké (a v praxi se to také ukázalo jako nefunkční řešení). Z toho jsem tedy usoudil, že velikost akumulátorového pole bude výrazně závislá na počtu vstupních bodů. Proto se zdá rozumné odvodit velikost pole právě od této hodnoty. Nejprve jsem zkoušel určit velikost pole jako tisícinu z počtu bodů. To se také neukázalo jako vhodné řešení, protože pro malá mračna bylo pole příliš malé a pro velká mračna naopak příliš velké (pro 1,5 milionu bodů by bylo potřeba alokovat pole o rozloze  $15000 \times 15000$ , což by bylo paměťově extrémně náročné). Lineární závislost mezi počtem bodů a velikostí akumulátorového pole tedy není správné řešení. Došel jsem tedy k úvaze, že by mohlo být vhodné zvolit stejný počet políček v akumulátorovém poli, jako je počet bodů. To nám zajistí, že pole nikdy nebude příliš malé, protože se předpokládá, že body ležící ve stejně rovině budou jen v několika sousedních políčkách. Proto by mělo

v poli zbýt dostatek volných políček, aby jednotlivé vrcholy nesplynuly dohromady. Po vyzkoušení na reálných datech jsem zjistil, že velikost pole není ani příliš velká a vrcholy jsou dostatečně výrazné. Jako rozměr akumulátorového pole jsem tedy určil odmocninu z počtu bodů (to odpovídá stavu, že počet políček v poli se rovná počtu bodů). V takovém případě je pro mračno s 10 tisíci body alokováno pole velikosti  $100 \times 100$ , pro 1,5 milionu bodů je to pole o rozměrech  $1225 \times 1225$ . Samozřejmě ani toto není optimální řešení. Velikost pole by měla být nejlépe odvozena od konkrétního zpracovávaného mračna a jeho vlastností jako očekávaný počet rovin, hustota bodů, počet bodů mimo rovinu apod. Nicméně takový algoritmus by nebyl vůbec jednoduchý a pravděpodobně by vyžadoval nějaké vstupní parametry zadáne uživatelem. To je ale nevhodné, za prvé se očekává, že algoritmus bude co nejvíce automatický, za druhé by běžný uživatel bez hlubších znalostí nebyl schopen takové parametry zadat. Proto jsem nakonec zvolil uvedené řešení.

Nyní máme připravené potřené datové struktury, můžeme tedy začít se samotným algoritmem. Nejprve si tedy vytvoříme novou instanci kd-stromu, konkrétně instanci třídy `SqrEuclid`, což je kd-strom, který využívá klasickou euklidovskou vzdálenost mezi body. Poté kd-strom v cyklu naplníme body a zároveň získáme jednoduchým porovnáním největší a nejmenší hodnoty souřadnic v celém mračnu bodů. Tyto souřadnice využijeme k výpočtu pozic referenčních bodů podle vztahu [2.2](#).

Ještě před samotným výpočtem parametrů bodů musíme stanovit velikost koule, která definuje okolí (viz sekce [2.1.1](#)). V popisovaném článku je tato velikost udávána euklidovskou vzdáleností (např. 3m). Já ale v testovacích datech žádné měřítko nemám, navíc mně dostupná data jsou poměrně odlišná od dat, které používali pro testování v článku. Proto je tato definice nevhodná a je potřeba ji nějak obejít. Protože mohou mít různá data výrazně jinou hustotu bodů, přijde mi rozumné velikost koule vypočítat z daného konstantního počtu bodů. Bohužel určit tento počet naprostě obecně pro libovolný typ dat je téměř nemožné. Proto jsem se rozhodl nechat tuto hodnotu jako volný parametr, který bude zadán uživatelem. To má i další důvod, tato hodnota velmi zásadně ovlivňuje dobu běhu algoritmu. Vyhledávání okolí v kd-stromu je drahá operace a pokud se pro každý bod vyhledává okolí v rádu stovek bodů, algoritmus může běžet desítky minut, což jistě není vždy optimální. Zároveň ale do určité míry platí, že čím větší okolí bodu se bere v úvahu, tím přesnější jsou vypočtené parametry bodu. Tím, že necháváme tento parametr na uživateli, mu dáváme možnost vybrat si mezi rychlostí nebo kvalitou. Pokud tedy uživatel zadá malé hodnoty tohoto parametru, např. 10 - 40, tak je algoritmus poměrně rychlý, ale může vracet horší výsledky. Pokud zadá větší hodnoty, např. 80 - 200, tak může být doba běhu algoritmu výrazně pomalejší, zato s kvalitnějšími výsledky. Obecně bych doporučil volit hodnoty z intervalu 20 - 200 s tím, že vhodný kompromis mezi kvalitou a rychlosťí je obvykle 40 - 80. Konkrétní dopad těchto hodnot na dobu běhu a kvalitu segmentace bude popsán v kapitole [4](#). Je důležité zmínit, že tato hodnota se musí odvíjet zejména od vstupních dat, nikoliv pouze od toho, zda chceme kvalitu nebo výkon. Pokud máme data hodně přesná (tedy body v rovině mají od této roviny jenom minimální odchylku), můžeme volit menší hodnoty pro velikost okolí a výsledek bude stále kvalitní. Pokud máme ale např. data reprezentující budovu s nerovným povrchem (např. s taškovou střechou), který chceme rozpozнат jako segment, musíme volit okolí větší. Na obrázku [3.1](#) je ukázka takové situace, kdy volba příliš malého okolí značně ovlivní výsledný výpočet.

Hodnota, kterou uživatel zadá, se ještě přímo nevyužije k definici okolí. Tato hodnota



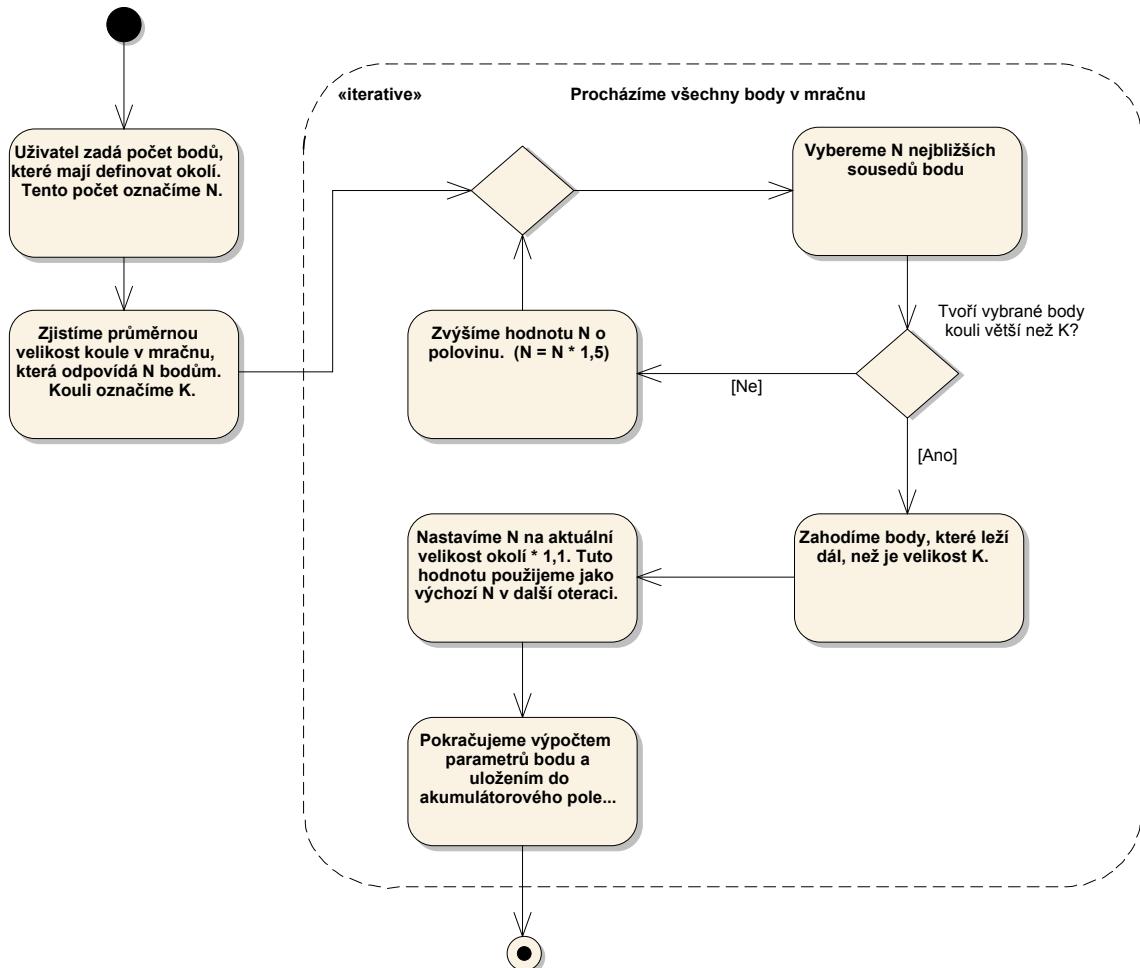
Obrázek 3.1: Ukázka vlivu velikosti okolí na vypočítávané parametry. Malé okolí může způsobit chybný výpočet (a), oproti tomu velké okolí je přesnější (b).

se využije pouze k vypočtení velikosti výsledné koule. Výpočet probíhá tak, že se náhodně vybere z mračna určitý počet bodů (např. setina ze všech bodů). Pro každý náhodně vybraný bod se vybere takový počet nejbližších sousedů, jaká je hodnota zadaného parametru. Poté se spočítá poloměr koule, které vybrané body tvoří (tedy vzdálenost střed - nejvzdálenější bod). Nakonec se spočítá aritmetický průměr všech vypočtených poloměrů. Tento aritmetický průměr je konečná velikost koule, která bude v algoritmu definovat okolí bodu. Pro všechny body bude mít tedy okolí stejnou velikost, ale bude pokaždé obsahovat různý počet bodů.

V tuto chvíli již máme všechny potřebné informace pro výpočet parametrů všech bodů. Máme načtené body v kd-stromu, máme definované oba referenční body, dále máme vypočtenou velikost koule pro okolí bodu a máme připravené akumulátorové pole, do kterého budeme body ukládat. Vytvoříme tedy cyklus, pomocí kterého budeme procházet všechny body v mračnu.

Nyní potřebujeme pro každý bod získat jeho okolí, tedy všechny body, které leží v kouli se středem v daném bodu a poloměrem, který jsme vypočítali. Víme, kolik průměrně bodů by mělo v této kouli být. Pro první iterovaný bod tedy vezmeme tento počet nejbližších sousedů. Vyhledávání  $n$  nejbližších sousedů je v kd-stromu definovaná operace, takže je to velmi jednoduché. Dále si spočítáme vzdálenost zkoumaného bodu a nejvzdálenějšího bodu z vybraných sousedů (kd-strom vrací body seřazené podle vzdálenosti od původního bodu, takže nejvzdálenější bod je ten poslední v seznamu). Pokud je tato vzdálenost větší, než je velikost koule definující okolí, tak pomocí binárního prohledávání projdeme seznam sousedů a nalezneme nejvzdálenější bod, který ještě leží uvnitř koule. Všechny body, které jsou dál, zahodíme. Pokud nastane situace, kdy ještě nemáme dostatek bodů, vynásobíme

aktuální počet vybraných sousedů  $1,5 \times$  a tento nový počet opět vybereme z kd-stromu a testujeme. Takto pokračujeme do té doby, než máme dostatek bodů. Je potřeba zmínit, že tento iterativní proces by měl nastat co nejméně, protože vyhledávání nejbližších sousedů v kd-stromu je pomalá operace. Celý výše popsaný proces získávání bodů pro okolí je přehledně znázorněn na diagramu 3.2.



Obrázek 3.2: Diagram znázorňující získání potřebných nejbližších sousedů bodu.

Protože blízké body v mračnu budou mít pravděpodobně velmi podobné okolí, využijeme výslednou velikost okolí jako výchozí hodnotu pro bod v další iteraci<sup>1</sup>. Tady nastává otázka, jestli použít přesně tu samou hodnotu nebo jí o trochu zvýšit. Vyhledávání bodů v kd-stromu je pomalá operace a nechceme, aby bylo v další iteraci zbytečně provedeno dvakrát, protože chybělo několik málo bodů. Zároveň je ale nutné vzít v úvahu, že čím více bodů v kd-stromu vyhledáváme, tím pomalejší toto vyhledávání je. Je tedy otázka, jaký je

<sup>1</sup>Očekáváme tedy, že body na vstupu jsou nějakým způsobem seřazeny, a že v naprosté většině případů jsou dva po sobě jdoucí body velmi blízko v prostoru.

optimální poměr navýšení počtu bodů do další iterace. Proto jsem se rozhodl určit tento parametr experimentálně. Porovnal jsem 3 různá vstupní mračna o rozdílných velikostech a volil jsem velikost okolí 50 a 200 bodů. V tabulce 3.1 je zaznamenán celkový čas, který vybíráni sousedních bodů zabralo a také je zde uvedeno procento bodů, pro které bylo nutné vyhledávat v kd-stromu vícekrát<sup>2</sup>. V naprosté většině případů byly výsledky nejlepší při hodnotě parametru 1,10. Pro další bod při iterování bude tedy použit  $1,1 \times$  násobek aktuální velikosti okolí.

Okolí	Data 1: 291232 bodů				Data 2: 81294 bodů				Data 3: 1299900 bodů			
	50 bodů		200 bodů		50 bodů		200 bodů		50 bodů		200 bodů	
	Opak.	Čas	Opak.	Čas	Opak.	Čas	Opak.	Čas	Opak.	Čas	Opak.	Čas
1,00	50,83%	21,67	51,45%	79,67	49,59%	6,15	49,83%	22,54	50,51%	135,52	50,52%	659,17
1,05	27,43%	17,04	16,84%	56,95	33,06%	5,35	24,92%	18,01	27,69%	114,88	15,84%	452,84
<b>1,10</b>	<b>17,89%</b>	<b>15,19</b>	<b>10,11%</b>	<b>53,28</b>	<b>24,67%</b>	<b>4,51</b>	<b>16,70%</b>	<b>17,38</b>	<b>17,36%</b>	<b>110,57</b>	<b>9,31%</b>	<b>457,39</b>
1,15	12,88%	16,18	7,31%	57,25	18,64%	5,14	12,36%	19,69	12,60%	111,30	6,72%	488,15
1,25	8,04%	16,49	4,93%	60,14	13,89%	4,55	8,70%	20,54	7,98%	120,90	4,38%	522,85
1,50	4,26%	17,99	2,91%	70,07	7,72%	5,07	5,34%	21,17	3,79%	133,85	2,48%	604,66
2,00	2,17%	23,90	1,62%	99,68								
2,50	1,51%	26,34	1,15%	121,76								

Tabulka 3.1: Tabulka znázorňující rozdílné časy vybíráni okolí z kd-stromu v závislosti na definovaném parametru. Čas je uváděn v sekundách, sloupec „Opak.“ určuje, pro jaké procento bodů bylo nutné z kd-stromu vybírat okolí více než jednou. Prázdná polička jsem již nevyplňoval, protože předešlé výsledky byly dostatečně průkazné a další měření by bylo zbytečné.

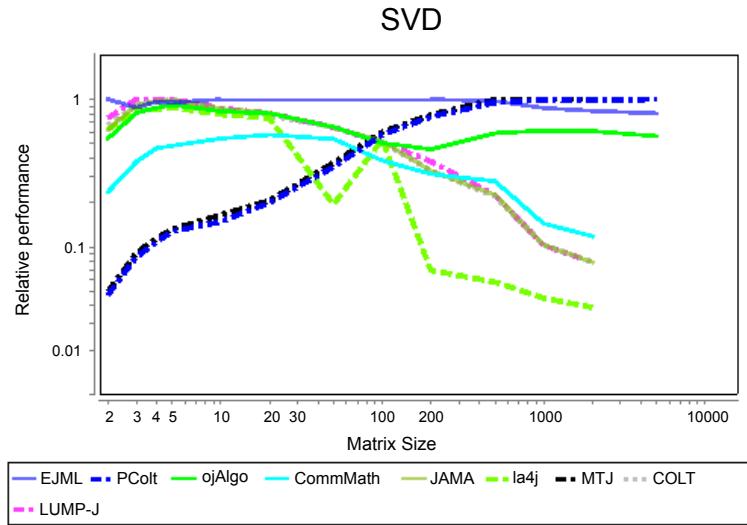
V tuto chvíli máme pro bod vybrány všechny sousední body, které leží uvnitř dané koule. Abychom získali konečné okolí bodu, musíme proložit body rovinou a poté provést převažování této roviny pomocí procesu popsaného v sekci 2.1.1. Dalším krokem v algoritmu je tedy pomocí metody nejmenších čtverců proložit rovinu body uvnitř koule.

Teorie k proložení roviny množinou bodů pomocí metody nejmenších čtverců je popsána v sekci 2.2. Implementace této metody přímo vychází z teorie. Prvním krokem je výpočet centroidu, což je bod ležící na výsledné rovině. Tento bod se spočte jednoduše jako aritmetický průměr souřadnic všech vstupních bodů. Poté se vytvoří matice, které má 3 sloupce a stejný počet řádků, jako je vstupních bodů. Na  $i$ -té řádku je po souřadnicích zapsán rozdíl hodnot  $bod_i - centroid$ . Nad touto maticí je nutné provést SVD dekompozici. Poté stačí pouze vybrat z matice  $V^T$  poslední sloupec a dopočítat koeficient  $d$ , viz vztah 2.10. Tím získáme všechny 4 potřebné parametry a můžeme vrátit novou instanci roviny.

Ještě je nutné uvést, jak se provede SVD dekompozice. To je poměrně složitý a výpočetně náročný algoritmus, takže má smysl využít nějakou kvalitní matematickou knihovnu. Těch je ale velký počet a mohou se výrazně lišit ve výkonu. Na webových stránkách [1] je dostupné srovnání výkonu několika nejběžnějších knihoven pro matematické výpočty. Na obrázku 3.3 je graf, který znázorňuje výkon jednotlivých knihoven při výpočtu SVD.

Nejlépe z testu vychází knihovna EJML [2]. Nicméně je nutné poznamenat, že tato knihovna je od stejného autora, jako je daný test. Proto jsem se rozhodl provést vlastní test na reálných datech. Nedělal jsem žádný speciální test, pouze jsem měnil knihovny pro

<sup>2</sup>Tato informace je pouze pro zajímavost, relevantní pro porovnání je pouze celkový čas.



Obrázek 3.3: Graf znázorňující relativní výkon (1 = nejvyšší výkon) různých knihoven při SVD dekompozici. Obrázek je převzat z [1].

výpočet SVD v rámci již hotového algoritmu. Použil jsem mračno 389002 bodů, nastavil velikost okolí 100 bodů<sup>3</sup> a měřil jsem čas, který výpočet SVD zabere. Měření je uvedeno v tabulce 3.2.

Měření	EJML*	EJML	Jama*	UJMP	CommMath	Colt	jBlas
1	88,958	136,998	152,786	157,150	185,178	206,535	>300,000
2	88,860	135,831	151,418	156,057	182,576	201,198	-
Ø	88,909	136,415	152,102	156,604	183,877	203,867	>300,000

Tabulka 3.2: Tabulka znázorňující časy výpočtu SVD různých knihoven. Časy jsou uvedeny v sekundách. Knihovny označené \* jsou nastavené tak, aby při SVD počítaly pouze matici  $V^T$ .

Jak je z tabulky patrné, nejrychlejší je knihovna EJML, stejně jako ve výše uvedeném testování. Rozdíl byl natolik výrazný, že jsem provedl pouze dvě měření, přišlo mi zbytečné měření opakovat na jiných datech. Navíc EJML umožňuje při SVD vypočítávat pouze matici  $V^T$ , která je pro proložení roviny potřebná, což výrazně zvyšuje celkový výkon. Z těchto důvodů jsem se rozhodl pro výpočty SVD využít EJML.

Dříve v této kapitole jsem odůvodňoval, proč je průměrná velikost okolí zadávána uživatelem. Jedním z argumentů bylo, že uživatel si částečně může zvolit, jestli chce segmentaci provést rychle nebo spíše pomaleji a kvalitněji. Je potřeba zmínit, že rychlosť algoritmu není daná pouze vybíráním okolí z kd-stromu. Další velkou složkou z celkového času je právě výpočet SVD při prokládání roviny. Ve skutečnosti prokládání bodů rovinou zabere větší část z celkového času běhu algoritmu, než vybírání bodů z kd-stromu. Nicméně obě tyto hodnoty jsou řádově stejné a dohromady tvoří velkou část z celkové doby běhu. Stejně jako

<sup>3</sup>To odpovídá průměrně 100 bodům v okolí, tedy i matici 3x100 při výpočtu SVD.

u kd-stromu je i prokládání bodů rovinou je výrazně závislé na počtu vstupních bodů. Mělo by tedy být jasné, že velikost okolí naprosto zásadně ovlivňuje dobu běhu celého algoritmu, uživatel může tedy vhodnou volbou okolí výrazně přizpůsobit celkový čas algoritmu svým potřebám.

Zpět k implementaci algoritmu: jsme ve fázi, kdy máme vybrané okolí bodu a tímto okolím jsme proložili rovinu. Je nutné zmínit, že při prokládání roviny může nastat výjimka, kdy bod nemá žádný bod ve svém okolí nebo má jenom jeden. V takovém případě není proložení roviny z principu možné. Pro takovýto bod tedy nemůžeme vypočítat parametry a je nutné ho zahodit. Nicméně takových bodů by mělo být naprosté minimum a vzhledem k tomu, že bod nemá okolí, tak velmi pravděpodobně neleží na žádné rovině a jde spíš o nějakou chybu ve vstupních datech, takže jeho zahození nepředstavuje problém.

Pokud se rovinu podaří proložit, pokračujeme úpravou této roviny pomocí převažování. Nejprve pro každý bod spočítáme váhu, což je inverze ke vzdálenosti bodu od roviny. Po vypočítání všech vah proložíme znova rovinu všemi body, nicméně tentokrát využijeme váhy a váženou metodu nejmenších čtverců. Implementace vážené varianty metody nejmenších čtverců je téměř stejná jako běžné varianty, jenom se uvažují váhy při výpočtu centroidu a při konstrukci matice, nad kterou se poté provádí SVD. Po proložení roviny se rozhoduje, zda se opět vypočítají váhy a rovina se znova proloží nebo zda se cyklus ukončí. Ukončení cyklu nastává ve dvou případech, pokud jsme již v desáté iteraci nebo pokud se parametry nové roviny výrazně neliší od poslední iterace. To zjistíme tak, že si udržujeme normálový vektor roviny z předchozí iterace a zjistíme jeho odchylku od normálového vektoru roviny ze současné iterace. Pokud je odchylka menší než cca.  $2^\circ$ , tak cyklus ukončíme.

Ačkoliv tento proces převažování vypadá na první pohled jednoduše, je zde malý problém s výpočtem vah, které se v rámci iterací mění pomalu a provádí se zbytečně velký počet iterací. Jako řešení se nabízí použít inverzi k druhé mocnině vzdálenosti bodu od roviny. Toto řešení funguje poměrně dobře a výrazně snižuje potřebný počet iterací a tím i celkový čas běhu algoritmu. Nicméně i toto řešení má jeden problém, váhy bodů blízko rovině mohou narůstat do výrazně velkých čísel, což může ovlivnit výsledek. Problém nastává také v okamžiku, kdy bod leží přímo na dané rovině. V tu chvíli jeho váha odpovídá hodnotě  $1/0$ , což Java převede na hodnotu `Double.INFINITY`. Pokud se takováto váha dostane do výpočtu SVD, tak sice nevznikne žádná výjimka a výpočet proběhne, nicméně výsledek výpočtu je vždy chybný. Tento problém jsme se snažil vyřešit tím, že jsem hodnotu nekonečna nahrazoval nějakou velkou konstantou (zkoušel jsem různé varianty od  $10e5$  do  $10e50$ ), nicméně v tomto případě se výpočet SVD choval velmi nepředvídatelně a vracel špatné výsledky. Takže jediné řešení, které se mi podařilo najít a funguje, je body s „nekonečnou váhou“ do výpočtu roviny vůbec nezahrnovat. Toto řešení také není příliš dobré, protože body, které by měly rovinu určovat nejlépe jsou z výpočtu vynechány. Naštěstí takovýchto bodů je při výpočtu zanedbatelné množství (v průměru je ovlivněna váha jednoho bodu u 1 - 2% výpočtů rovin), takže na výslednou segmentaci nemá tento problém žádné rozpoznatelné důsledky.

Po ukončení procesu převažování již můžeme definovat výsledné okolí, ze kterého přímo vypočítáme parametry bodu. Toto okolí definujeme tak, že nad a pod výslednou proloženou rovinou definujeme buffer (viz Obr. 2.1) a body, které leží mimo tento buffer, zahodíme. Definice bufferu popsaná v sekci 2.1.1 používá pro určení velikosti bufferu šum v datech. Jak jsem již zmiňoval na začátku této kapitoly, v mých testovacích datech žádný šum není,

proto je potřeba tuto definici nějak obejít. Jako řešení se nabízí vypočítat RMS a porovnat jednotlivé vzdálenosti bodů od roviny s touto průměrnou hodnotou. Body, jejichž vzdálenost je menší jak  $2 \times \text{RMS}$ , zahrneme do výsledného okolí. Body, které leží dále, z okolí zahodíme.

Tímto procesem jsme získali výsledné okolí, ze kterého vypočítáme parametry bodu. To provedeme tak, že výsledným okolím jednoduše proložíme rovinou. Poté spočítáme vzdálenost této roviny od obou referenčních bodů. Tuto vzdálenost musíme ještě převést na souřadnice akumulátorového pole. To provedeme podle následujícího vztahu:

$$s = \text{param} \times \frac{\text{velikost akumulátorového pole}}{\text{velikost celého mračna bodu}} \quad (3.1)$$

kde  $s$  je souřadnice v akumulátorovém poli,  $\text{param}$  je vypočtená vzdálenost roviny od referenčního bodu a velikost celého mračna bodů odpovídá vzdálenosti bodů s největšími a nejmenšími souřadnicemi v mračnu bodů<sup>4</sup>. Takto vypočteme obě souřadnice a poté už pouze na vypočtenou pozici vložíme daný bod. Jak bylo zmíněno na začátku kapitoly, akumulátorové pole je pole seznamů, takže bod vkládáme přímo do seznamu na dané pozici. Před vložením musíme ještě ověřit, jestli je dané pozici již vytvořená instance seznamu, případně ji musíme vytvořit.

Tímto máme první část algoritmu kompletně hotovou. Ve chvíli, kdy vypočteme parametry pro všechny body a uložíme je do akumulátorového pole, tak toto pole vrátíme a segmentace pokračuje druhou částí - klastrováním bodů.

## 3.2 Klastrování bodů v akumulátorovém poli

Tato část algoritmu se zabývá tím, jak v akumulátorovém poli správně identifikovat jednotlivé segmenty a jak je poté z pole vybrat. Teorie je taková, že každý segment vytvoří v akumulátorovém poli vrchol (viz Obr. 2.4), který detekujeme a poté všechny body z takového vrcholu zahrneme do jednoho segmentu. Ačkoliv se to může zdát poměrně jednoduché, v praxi lze narazit na několik problémů. Jedním z takových problémů může být např. to, že každý vrchol je jinak vysoký a obsahuje jiný počet bodů, takže je nutné stanovit nějakou hranici, kdy vrchol ještě označíme za segment a kdy už bude jenom nějakým náhodným výsledkem šumu. Dalším problémem je situace, kdy jsou dva vrcholy velmi blízko u sebe a nejde rozlišit, kde je mezi nimi hranice. V takovém případě musíme využít nějakou další znalost o bodech z objektového prostoru. Může nastat další podobný problém, kdy jsou dva vrcholy blízko u sebe a na první pohled nedokážeme určit, že se jedná o dva vrcholy a nejen jeden. Problematická se poté ukáže i samotná definice vrcholu. V praxi může být jeden takový vrchol tvořen několika menšími vrcholy v těsné blízkosti. Musíme být tedy schopni rozlišit, kdy takové vrcholy tvoří jeden segment a kdy už ne. Toto jsou ve stručnosti hlavní problémy, kterými se budu v této kapitole zabývat.

Prvním důležitým krokem je tedy správně definovat vrchol a vytvořit algoritmus, který takové vrcholy v poli vyhledá. Vrchol jsem definoval tak, že je to souvislá oblast v poli, ve které všechny políčka obsahují více bodů, než je nějaký předem stanovený práh (hodnota).

---

<sup>4</sup>Toto jsme již vypočítali hned na začátku algoritmu při definování pozic referenčních bodů.

S takovouto definicí beru v úvahu, že se jeden vrchol může skládat z více spojených vrcholů v blízkém okolí. Zároveň pomocí této definice můžu vrcholy vyhledávat iterativně, nejprve nastavím práh vyšší a vyberu vrcholy, které budou s velkou pravděpodobností reprezentovat segment a tento práh můžu postupně snižovat a vyhledávat nové vrcholy, dokud bude jejich kvalita přijatelná.

Nyní je tedy potřeba navrhnout algoritmus, který takové vrcholy v poli vyhledá. Na první pohled by se mohlo zdát, že by bylo vhodné využít nějaký algoritmus na bázi gradientního vzestupu či simulovaného žíhání, nicméně u takových algoritmů není zaručeno, že lokálně najdou vždy ten nejvyšší vrchol. Navíc takovéto algoritmy ani nezaručují nalezení všech vrcholů. Proto je vhodné o tomto problému uvažovat trochu jinak, než o pouhém hledání lokálních maxim. Protože jsem vrchol definoval jako spojitou oblast splňující danou podmínu (oblast vyšší než daný práh), můžeme tento problém převést na vyhledávání spojitých oblastí. K tomu se nabízí využít algoritmus flood fill, který se používá např. pro implementaci plechovky v grafických editorech (tedy „vylévá“ spojité oblasti stejnou barvou). Algoritmus implementujeme tak, že budeme procházet akumulátorové pole po jednotlivých políčkách. Pokud narazíme na políčko, kde je počet bodů vyšší, než zadaný práh, pustíme flood fill na toto políčko a necháme ho vyhledat celou spojitu oblast, která je vyšší než práh. Z této oblasti poté vybereme jednoduše největší políčko a jeho souřadnice si uložíme do pomocného pole. Zároveň si vytvoříme bitové pole o stejné velikosti, jako je akumulátorové pole. Do tohoto pole si budeme poznamenávat, na kterých pozicích jsme identifikovali vrchol. Tyto pozice poté již neprohledáváme. Tím zajistíme, že nebudeme pouštět flood fill na jeden vrchol několikrát. Toto pole by nemělo přidat žádnou velkou paměťovou zátěž, protože pro označení políčka použijeme pouze jeden bit, takže pro toto pole potřebujeme alokovat pouze osminu paměti oproti původnímu akumulátorovému poli. Pomocí tohoto algoritmu projdeme celé pole a vyhledáme všechny vrcholy.

Samotný flood fill implementujeme pomocí fronty. Do té se nejprve přidá výchozí políčko. Poté začneme z fronty políčka vybírat. Vždy se zkontroluje všech 8 sousedních směrů a pokud nalezneme políčko, které je vyšší než práh a ještě jsme ho neprohledali, přidáme ho do fronty. Takto pokračujeme, dokud jsou ve frontě nějaká políčka. Zároveň si při tomto procesu ukládáme nejvyšší nalezené políčko, na kterém jsme byli. To využijeme k identifikaci daného vrcholu.

Celý tento proces jsme vytvořili na základě toho, že vrchol je spojitá oblast vyšší, než daný práh. Zatím jsme však neurčili, jak velikost prahu získat, což je samozřejmě pro algoritmus naprostě zásadní. Tato hodnota bude velmi závislá na počtu bodů a počtu očekávaných rovin v mračnu. Počet bodů v mračnu nicméně není pro definici tolik zásadní, protože ten ovlivňuje velikost akumulátorového pole. Tím pádem také nedojde jednoduše určit vztah mezi velikostí vrcholů a počtem vstupních bodů. Důležitějším faktorem bude počet rovin ve vstupním mračnu. Pokud bude mít mračno např. 4 roviny, dá se očekávat, že vzniknou 4 vysoké vrcholy. Pokud bude ale v mračnu rovin 15, bude pravděpodobně většina vrcholů mnohem nižších. Počet rovin však algoritmus nemůže žádným způsobem ovlivnit, bude tedy nutné, aby tento parametr nějakým způsobem ovlivnil uživatel. Byl by samozřejmě nesmysl, aby uživatel zadával samotný práh nebo případně nějakou očekávanou velikost vrcholů, uživatel samozřejmě nemusí vůbec vědět, jak algoritmus funguje a takové hodnoty by nebyly schopny odhadnout. Uživatel je však schopen odhadnout počet rovin v mračnu. Práh jsem se tedy rozhodl určit tak, že jeho výchozí hodnota bude nastavená na velikost akumulátorového pole

(tím bude částečně reflektovat počet bodů v mračnu). Uživatel poté zadá hodnotu, která bude rádově odpovídat očekávanému počtu rovin a touto hodnotou vydělíme výchozí nastavení prahu. Je nutné zdůraznit, že počet rovin bude tomuto parametru odpovídat pouze velmi přibližně. Uživatel spíše zadá ze začátku nižší hodnotu a pokud bude nalezeno příliš málo rovin, tak hodnotu zvýší. Algoritmus bude uzpůsobený tomu, aby měl uživatel možnost tuto hodnotu za běhu upravovat. Více o použití a zadávání této hodnoty bude zmíněno v sekci [3.3](#) a v kapitole [4](#).

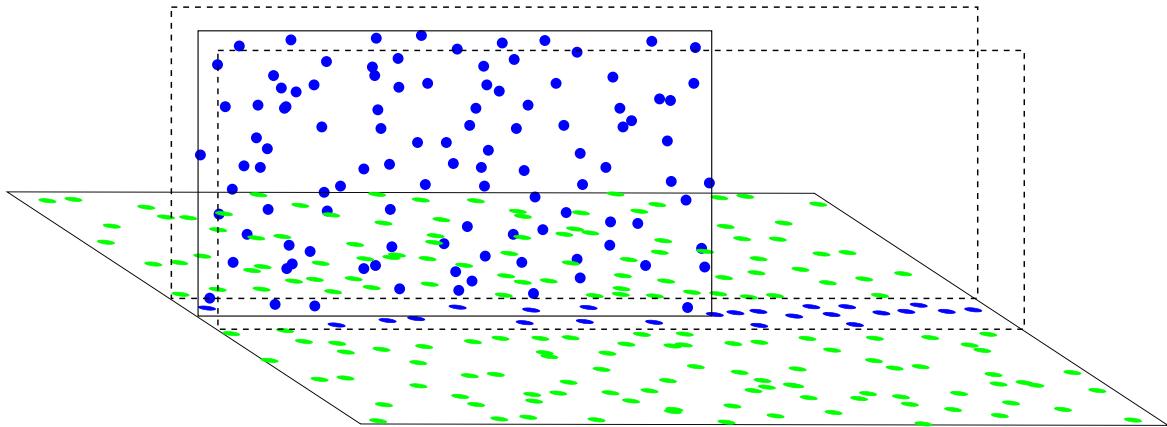
Máme tedy hotovou část, kdy jsme identifikovali vrcholy v poli a máme uložené souřadnice nejvyšších políček z těchto vrcholů. Nyní potřebujeme z každého vrcholu vybrat body, které patří do jednoho segmentu. Postup je takový, že vždy vybereme body z nejvyšší buňky, ty proložíme rovinou a vypočítáme RMS. Poté procházíme body z okolních buněk (viz Obr. [2.5](#)), pokud mají body vzdálenost od proložené roviny menší než  $2 \times \text{RMS}$  (leží ve stejně rovině) a zároveň jsou blízko k ostatním bodům v prostoru, přidáme je do segmentu. Tímto procesem bychom měli postupně projít všechny vrcholy. Proces má v zásadě dvě problematické části. Prvním problémem je, jak určit, že je bod blízko k ostatním bodům v prostoru. Podle teorie by se měla spočítat průměrná vzdálenost mezi všemi body v nejvyšším políčku vrcholu. Pro každý bod by se poté zjištěvalo, jestli je jeho průměrná vzdálenost ke všem bodům srovnatelná. Toto řešení je ale velmi neefektivní, protože pro každý bod musíme počítat vzdálenost ke všem ostatním bodům v segmentu. Nicméně podobnou podmíinku je třeba při přidávání bodů do segmentu zachovat. Proto zkonztruujeme nad body z nejvyšší buňky kd-strom. Při přidání nového bodu vždy vezmeme nejbližšího souseda a zjistíme jeho vzdálenost. Pokud je tato vzdálenost srovnatelná s průměrnou vzdáleností mezi body (tu v tomto případě také musíme spočítat), tak můžeme bod do segmentu přidat. Tabulka [3.3](#) ukazuje rozdíly v časech při klastrování s použitím průměrné vzdálenosti od všech bodů a při použití kd-stromu. Z tabulky je patrné, že použití kd-stromu je výrazně efektivnější, než druhá varianta. Parametry klastrování uvedené v tabulce budou vysvětleny později v sekci [3.3](#), pro pochopení tabulky nejsou podstatné.

<b>Data</b>	<b>Parametry</b>	<b>Čas [s]</b>	
		<b>Kd-strom</b>	<b>Prům. Vzd.</b>
291232 bodů	5 a 1	3	388
	10 a 5	14	559
723695 bodů	2 a 1	26	>3600

Tabulka 3.3: Tabulka zobrazuje rozdílné časy klastrování, pokud vyhledáváme nejbližší bod v kd-stromu nebo pokud počítáme průměrnou vzdálenost od všech bodů.

Druhou problematickou částí procesu je postupné vybírání bodů z akumulátorového pole. Jak jsem již zmínil, pokud začneme vybírat body z vrcholu, testujeme, zda jsou ve stejné rovině a blízko u sebe. Může ale nastat situace, kdy jsou dva vrcholy v akumulátorovém těsně vedle sebe a při vybírání z prvního vrcholu splní tuto podmíinku i některé body z druhého vrcholu (tedy i z druhé roviny). Ukázka takové situace je znázorněna na obrázku [3.4](#).

Řešení vůbec není jednoduché, nepodařilo se mi najít žádný způsob, kterým by šel tento problém detektovat a výrazně to nezhoršilo výkon algoritmu. Proto je nutné tento problém obejít a případně aspoň co nejvíce minimalizovat. Jedním ze způsobů, jak omezit výskyt



Obrázek 3.4: Ukázka chyby, která může nastat při postupném vybírání bodů z akumulátorového pole, pokud dvě roviny vytvoří blízké vrcholy v akumulátorovém poli.

takové situace, je nevybírat vrcholy z akumulátorového pole po jednom, ale vybírat všechny vrcholy najednou. Protože vybírání bodů z vrcholu je iterativní proces, implementace postupného vybírání všech vrcholů najednou je poměrně jednoduchá. Vrcholy v poli budeme procházet iterativně a budeme je vybírat vždy po malých částech ze všech vrcholů najednou. V první iteraci vybereme z akumulátorového pole nejvyšší políčka ze všech vrcholů. V druhé iteraci rozšíříme vyhledávání v každém vrcholu o okolní buňky (viz Obr. 2.5) a opět takto vybereme body postupně z každého vrcholu. V další iteraci opět zvětšíme okolí. Proces je tedy téměř stejný, jako byl původně navržený, jen nevybíráme jeden vrchol po druhém, ale iterativně je vybíráme všechny najednou. Tím jsme omezili výše popsaný problém, protože i když budou dva vrcholy blízko sebe, tak pokud se vybírání bodů dostane do oblasti druhého vrcholu, body z tohoto vrcholu už budou také vybrané a nehrozí, že budou zahrnuty do špatného segmentu.

Nyní máme navržený celý proces klastrování, tak podrobněji popíšeme implementaci. Vrcholy v poli jsme již našli a máme uložené souřadnice všech nejvyšších políček z vrcholů. Protože budeme přidávat body do všech segmentů najedou, budeme potřebovat nějakou datovou strukturu, která nám bude udržovat body ze segmentu, společně s dalšími informacemi. Vytvoříme tedy třídu `SegmentWrapper`, která bude obsahovat souřadnice vrcholu, všechny body, které do segmentu patří, dále bude ukládat aktuální RMS, průměrnou vzdálenost mezi body a booleovskou hodnotu, která bude určovat, zda je segment již uzavřený nebo jestli do něj ještě přidáváme body. Poté vytvoříme seznam instancí těchto objektů na základě počtu nalezených vrcholů v poli. Velikost seznamu bude odpovídat počtu nalezených vrcholů v akumulátorovém poli, každá instance bude inicializována body z nejvyšší buňky. Tyto body budou přidány do segmentu a odstraněny z akumulátorového pole.

Výsledný segment bude velmi ovlivněn parametry, které se vypočítají z těchto iniciálních bodů (zejména RMS). Proto je vhodné tyto body nejprve zpracovat a zahodit body, které nejsou s ostatními v rovině (tedy nějaký šum, který se náhodně trefil do vrcholu v akumulátorovém poli). Tento proces implementujeme velmi podobně, jako při definici okolí bodu. Nejprve vezmeme všechny body a proložíme je rovinou. Tuto rovinu převážíme stejným

způsobem, jako při definici okolí. Poté spočteme RMS a ze segmentu odebereme všechny body, které jsou od výsledné roviny dále, než  $5 \times \text{RMS}$ . Tyto body nezahodíme úplně, ale vrátíme je zpět do akumulátorového pole na jejich původní pozici. V tuto chvíli potřebujeme ošetřit situaci, kdy může více rovin splynout v jeden vrchol (situace je podrobně popsána na konci sekce 2.1.3). Tento problém detekujeme tak, že opět přepočítáme RMS bodů v segmentu. Pokud je tato hodnota výrazně velká, body s největší pravděpodobností neleží v rovině. Velikost RMS porovnáme s velikostí koule definující okolí, kterou jsme vypočítali úplně na začátku algoritmu. Tato velikost bude vždy větší, než očekávané RMS u běžné roviny. Pokud tedy detekujeme problém, měli bychom ho řešit vytvořením nového akumulátorového pole a změnou pozice referenčních bodů podle rovnice 2.4. Tento postup je ale značně neefektivní, protože vyžaduje alokovat druhé akumulátorové pole stejně velikosti, jako to původní. Jednodušší postup je rekurzivně vytvořit novou instanci celého algoritmu a předat jí na vstupu pouze body způsobující problém. Nová instance je vrátí segmentované a tím jsme problém vyřešili. Akorát původní segment s více rovinami odstraníme ze seznamu a přidáme nově identifikované segmenty. Těmto segmentům nastavíme stejné souřadnice vrcholu a budeme do nich přidávat body stejným způsobem, jako do ostatních segmentů. Jediný problém, který při tomto procesu může teoreticky nastat je zacyklení v rekurzi. To vyřešíme tím, že nedovolíme algoritmu v rekurzi jít do větší hloubky. Taková situace je opravdu spíše jen teoretická, proto nemá smysl vymýšlet složitější oštření takové situace.

Ať už popsaný problém nastane nebo ne, výsledkem tohoto procesu je seznam segmentů, které mají určené souřadnice svých vrcholů, obsahují body z nejvyšší buňky vrcholů, mají těmito body proloženou rovinu a vypočtené RMS. Všechny tyto segmenty by už měly reprezentovat roviny a budou zahrnuty ve výstupu z algoritmu. Zbývá vybrat body z jednotlivých vrcholů a přiřadit je do těchto segmentů způsobem, který jsme navrhli výše. Budeme tedy iterativně procházet všechny vrcholy a v každé iteraci zvětšíme okolí vrcholu, které procházíme. Při procházení vrcholu testujeme každý bod, zda leží ve stejně rovině s ostatními body (pomocí RMS) a zda je blízko k ostatním bodům (pomocí kd-stromu). Bod, který přidáme do segmentu odebereme z akumulátorového pole. Pokud bod podmínky nesplní, v akumulátorovém poli ho ponecháme. Otázka je, kdy přesně ukončit přidávání bodů do segmentu. V analýze bylo navrženo, aby byl tento proces ukončen ve chvíli, kdy už do vrcholu nejsou přidány žádné body. To je ale v praxi samozřejmě nereálné, protože v pozdějších iteracích se prohledává poměrně velké množství políček a s velkou pravděpodobností by se vždy několik málo bodů přidal a tento proces by se nemusel zastavit vůbec. Tuto podmínu tedy upravíme tak, že pokud se za jednu iteraci přidá nějaký zlomek z celkového počtu bodů v segmentu, bude proces ukončen. Tento zlomek jsem se rozhodl nenastavovat napevno, ale ponechat tuto volbu na uživatele. Tím tedy zavádíme do algoritmu třetí (a také poslední) parametr, pomocí kterého uživatel může ovlivňovat chování segmentace. Důvod je ten, že při vybírání bodů z vrcholu se v prvních iteracích rychle vybere většina bodů. Pokud tedy chceme proces co nejrychlejší, nastavíme zlomek např. na jednu setinu. Ve chvíli, kdy se v jedné iteraci přidá méně jak setina bodů segmentu, segment je uzavřen a už se do něj body nepřidávají. Takovýto proces je velmi rychlý, pro běžná mračna bodů trvá klastrování s tímto parametrem maximálně několik desítek vteřin. Pokud nastavíme zlomek na jednu tisícinu, proces bude výrazně pomalejší, ale segmenty budou obsahovat více bodů (budou kvalitnější). Tedy stejně jako předchozí parametry, i tento umožňuje uživateli výběr mezi rychlostí a kvalitou. Pro zjednodušení se bude na vstupu od uživatele očekávat číslo řádově

od jedné do deseti. Toto číslo se poté vynásobí stem a použije se jako jmenovatel zlomku. Pokud uživatel zadá jedna, proces bude tedy rychlý, ale méně kvalitní, oproti tomu při zadání hodnoty deset bude proces pomalejší a kvalitnější. Uživatel bude mít samozřejmě možnost zadat libovolnou hodnotu z tohoto intervalu, případně i větší hodnotu než 10, pokud to bude mít smysl. Konkrétní dopady těchto hodnot na výslednou segmentaci jsou ukázány v kapitole 4 věnované testování.

Pokud jsou všechny segmenty ukončeny, proces ještě není hotový. Již na začátku kapitoly byl nastíněn problém, kdy mohou být dva vrcholy v akumulátorovém poli blízko sebe a algoritmus je nerozezná. V takovém případě bude vybrán jen jeden vrchol a druhý v akumulátorovém poli zůstane. Proto je celý proces klastrování navržen iterativně. V tuto chvíli projdeme aktuální stav akumulátorového pole a stejným způsobem jako na začátku vyhledáme vrcholy. Pokud jsou nějaké vrcholy nalezeny, algoritmus pokračuje standardním způsobem. Pokud žádné vrcholy nejsou nalezeny, prohledávání akumulátorového pole končí a metoda navrátí seznam všech nalezených segmentů. Tím je kompletně ukončena segmentace.

Uživatel má však v tuto chvíli stále k dispozici ukazatel na zbylé akumulátorové pole. Může se tedy rozhodnout změnit hodnotu parametrů a segmentaci nad tímto zbylým polem provést znova. Tento způsob dává uživateli poměrně mocný nástroj, jak vybírat roviny z akumulátorového pole postupně. Roviny, které vybere nejdříve, jsou teoreticky ty nejpřesnější a nejkvalitnější. Pokud bude uživatel postupně měnit parametry a vybírat další roviny, bude jejich kvalita pravděpodobně klesat. Uživatel může tedy měnit parametry a vybírat roviny do té doby, dokud mu kvalita vybíraných rovin přijde dostatečná. Tento proces ale také vůbec nemusí použít a může navolit parametry tak, aby bylo hned napoprvé vybráno co nejvíce rovin. V tomto ohledu nabízí implementace algoritmu poměrně velkou volnost.

### 3.3 Rekapitulace zadávaných parametrů a jejich použití

Pro přehlednost bych rád zrekapituloval, jakým způsobem může uživatel ovlivnit výsledky segmentace. Definovali jsme celkem tři parametry, velikost koule definující okolí při výpočtu parametrů bodu a poté dva parametry při klastrování bodů. První parametr ovlivňuje okolí bodu, čímž zásadně ovlivňuje i vypočítané parametry bodu a tím celou segmentaci. Parametr se řádově volí z intervalu 20 - 200 bodů, kdy běžné hodnoty by měly být v rozsahu 40 - 80 bodů. Čím více bodů, tím by měl být výpočet parametrů bodu přesnější, ale také pomalejší. Zároveň ale musíme přizpůsobit hodnotu parametru vstupním datům, kdy pro horší data musíme volit spíše vyšší hodnotu parametru.

Druhý parametr také velmi zásadně ovlivňuje výslednou segmentaci. Tento parametr ovlivňuje práh, který určuje jednotlivé vrcholy v akumulátorovém poli. Má tedy zejména vliv na kvalitu výsledné segmentace a nemá téměř žádný vliv na její rychlosť. Hodnota parametru však není předem úplně jasná, je velmi závislá na vstupním mračnu bodů. Hodnota velmi přibližně odpovídá očekávanému počtu rovin, ale nejde to vzít jako přesné kritérium. Parametr je nastavený tak, že čím větší hodnota je zadána, tím je práh v akumulátorovém poli nižší - očekává se tedy více menších vrcholů. Pokud má vstupní mračno např. 3 nebo 4 roviny, hodnota parametru se pravděpodobně bude pohybovat v rozsahu 1 - 3. Pokud má vstupní mračno rovin 15, parametr se pravděpodobně bude pohybovat v rozmezí 8 - 15.

Uživatel má 2 hlavní možnosti, jak tento parametr odhadovat. První možností je vybrat nějakou přibližnou hodnotu a ostatní parametry nastavit tak, aby byla segmentace rychlá a udělat „testovací“ segmentaci. Pokud je uživatel s výsledky spokojen, hodnotu parametru ponechá a ostatní parametry nastaví na větší kvalitu. Pokud není spokojen, změní parametr a provede testovací segmentaci znova. Druhou možností je nastavit parametr ze začátku na nízkou hodnotu. Segmentace proběhne a vybere z mračna jen ty největší a nejkvalitnější roviny. Uživatel má ale stále k dispozici zbylé akumulátorové pole, takže může parametr zvýšit a segmentaci spustit znova. Segmentace by opět měla nalézt několik nových rovin. Tímto způsobem lze parametr postupně zvyšovat (a tím pádem snižovat práh), dokud je kvalita výstupních segmentů dostatečná.

Třetí parametr ovlivňuje počet bodů zahrnutých do segmentu. Zde má uživatel opět velkou možnost volby mezi rychlosťí a kvalitou. Pokud zadá nízkou hodnotu, např. 1, klastrování bude velmi rychlé, ale segmenty budou obsahovat méně bodů, zvláště budou chybět body v okolí hran a nerovností. Oproti tomu pokud zadá vysokou hodnotu, např. 10, algoritmus bude mnohem pomalejší, ale v segmentech bude zahrnuto více bodů.

Konkrétní vliv těchto parametrů na výslednou segmentaci je popsán v následující kapitole 4, která se věnuje testování.

## 3.4 Struktura a použití zdrojového kódu

### 3.4.1 Struktura zdrojového kódu

Protože se jedná o implementaci jednoúčelového algoritmu, struktura použitých tříd je poměrně jednoduchá. Pro algoritmus je vytvořena samostatná třída umístěná v balíčku `vicitis.tools`. Dále jsem vytvořil dvě třídy, které implementují prokládání bodů rovinou pomocí metody nejmenších čtverců. Jde o třídu `LeastSquare`, která implementuje základní verzi, a `WeightedLeastSquare`, která implementuje váženou verzi metody nejmenších čtverců. Tyto třídy jsou oddělené od obou algoritmů, aby bylo možné je použít i jinde v systému. Nacházejí se v balíčku `vicitis.inner.methods`. Obě třídy jsou na použití velmi jednoduché, obsahují jednu statickou metodu `fitPlane`, která má na vstupu body a případně jejich váhy a vráci instanci třídy `Plane3D`, tedy reprezentaci roviny.

Samotný algoritmus se nachází v třídě `PlanarSegmentation`. Rozhraní třídy pro běžné používání je poměrně jednoduché. Obsahuje dvě metody, pomocí kterých se třídě nastavuje vstupní mračno bodů, resp. se získává výsledné segmentované mračno. Tyto metody jsou na ukázce 3.1.

Zdrojový kód 3.1: Metody pro nastavení vstupu a získání výstup algoritmu.

```

1 /**
2  * Nastavuje mračno bodu pro segmentaci
3  * @param pc vstupní mračno bodu (pointcloud)
4  */
5 public void setPointCloud(ArrayList<Point3D> pc) {
6     this.pointCloud = pc;
7 }
8

```

```

9  /**
10   * Vraci segmentovane mracno bodu
11  * @return segmentovane mracno bodu
12 */
13 public HashMap<Integer, ArrayList<Point3D>> getSegmentedPointCloud() {
14     return segmentedPointCloud;
15 }

```

Pro použití samotného algoritmu třída obsahuje další dvě metody, viz ukázka 3.2. Jedna metoda slouží k uložení bodů do akumulátorového pole a druhá ke klastrování těchto bodů v akumulátorovém poli. Parametry, které tyto metody vyžadují, jsou popsány v předchozí sekci 3.3.

Zdrojový kód 3.2: Metody určené pro používání algoritmu.

```

1 /**
2  * Pouze provede segmentaci do akumulatoroveho pole
3 *
4  * @param pointPerSphere pocet bodu, ktere prumerne definuji velikost okoli
5  *                      bodu, zasadne ovlivnuje vysledky i rychlosť algoritmu (cim mene bodu, tim
6  *                      rychlejsi) doporučeno 20-200, obvykle 40-80
7  * @return akumulatorove pole, ktere obsahuje vsechny body ze vstupniho
8  *                      mracna
9 */
10 public ArrayList<Point3D>[][] doSegmentation(int pointPerSphere) {
11     if(pointCloud == null)
12         throw new NullPointerException("Neni nastaveno zadne vstupni mracno
13                                bodu.");
14     return this.doSegmentation(pointCloud, pointPerSphere);
15 }
16
17 /**
18  * Provadi klastrovani bodu v akumulatorovem poli
19  *
20  * @param acc akumulatorove pole naplnene body
21  * @param minPeakCountRatio parametr, ktery urcuje "velikost" rovin. Cim
22  *                          vetsi, tim vice mensich mensich rovin by melo vstupni mracno mit. Hodnota
23  *                          priblizne odpovida poctu rovin
24  * @param quality kvalita klastrovani, cca 1 - 10, cim vic, tim vice
25  *                          segmentovanych bodu, ale delsi cas
26 */
27 public void doClustering(ArrayList<Point3D>[][] acc, int minPeakCountRatio,
28     int quality) {
29     if(acc == null)
30         throw new NullPointerException("Akumulatorove pole je null.");
31     this.doClustering(acc, pointCloud.size(), minPeakCountRatio, quality);
32 }

```

Tyto dvě public metody jsou delegovány na přetížené private metody, které již přímo implementují algoritmus. Součástí této třídy jsou ještě dvě vnitřní třídy, které jsou použity jako interní datové struktury využitelné jen v rámci tohoto algoritmu. První třídou je SegmentWrapper, která slouží pro reprezentaci segmentu při klastrování bodů. Druhou vnitřní třídou je BitArray, která slouží k reprezentaci bitového pole. Do toho pole se

během vyhledávání vrcholů poznamenává, která políčka byla již projita.

### 3.4.2 Ukázkové použití algoritmu

Jak bylo naznačeno v sekci 3.3, jsou dvě základní možnosti, jak algoritmus používat. V ukázce použití uvedu tu složitější variantu, kdy se nejprve provede uložení bodů do akumulátorového pole a poté je možné provést klastrování bodů vícekrát. Zdrojový kód této varianty je zobrazen v ukázce 3.3. Z ukázky je pro přehledost odebráno ošetření výjimek.

Zdrojový kód 3.3: Ukázkové použití implementace algoritmu.

```

1 //nacteme vstupni mracno
2 ArrayList<Point3D> pc = new ArrayList<Point3D>();
3 File[] files = new File("input/").listFiles();
4 for(File f : files) {
5     pc.addAll(new PLYpcLoader(f, true).getPointCloud());
6 }
7 //instance segmentace
8 PlanarSegmentation segmentation = new PlanarSegmentation();
9 segmentation.setPointCloud(pc);
10 //parametr urcujuici velikost okoli
11 int pps = getInput("Zadejte velikost okoli bodu (cca. 20-200): ");
12 //ulozime body do akumulaturovoeho pole
13 ArrayList<Point3D>[][] acc = segmentation.doSegmentation(pps);
14
15 //klastrovani muzeme provadet vicekrat
16 while(true) {
17     //klon akumulatoroveho pole
18     ArrayList<Point3D>[][] accClone = new ArrayList[acc.length][acc.length];
19     /* tady je potreba projit cele akumulatorove pole a
20      * naklonovat jednotliva policka, pro ukazku nepodstatne */
21
22     //nastavime parametry pro klastrovani
23     int param = getInput("Zadejte hodnotu parametru pro klastrovani: ");
24     int qlt = getInput("Zadejte 'kvalitu' segmentu (cca 1-10): ");
25     //muzeme vybirat segmenty postupne
26     do {
27         segmentation.doClustering(accClone, param, qlt);
28         HashMap<Integer, ArrayList<Point3D>> spc = segmentation.
29             getSegmentedPointCloud();
30         VRMLexporter.exportSegmentedPointClouds(spc, new File("out-iterace.wrl"));
31
32         param = getInput("Pokud chcete pokracovat v tomto klastrovani, "
33                         + "zadejte vyssi hodnotu parametru pro klastrovani, "
34                         + "(0 = konec tohoto klastrovani): ");
35         if(param == 0) break;
36     } while(true);
37     //podminka, zda pokracovat novym klastrovanim...
38 }
```

---

Pro načtení uživatelského vstupu je zde použita metoda `getInput`, která uživateli vypíše zadaný text a vrací jím zadaný parametr. Tato metoda není nikde implementována, to není účelem této ukázky, je zde jen pro ilustraci. Důležité je, že po zadání velikosti okolí se provede

uložení mračna bodů do akumulátorového pole, viz řádek 13. Toto se provede za celý běh algoritmu pouze jednou. Poté následuje cyklus, který nám umožňuje získané akumulátorové pole segmentovat vícekrát. To je zajištěno tak, že na začátku každého cyklu se vytvoří klon původního akumulátorového pole, které se vůbec nezpracovává, operace se provádí pouze nad klonovaným polem. Poté jsou od uživatele vyžádány zbylé dva parametry segmentace a následně se provede klastrování bodů, viz řádek 28. Uloží se výsledek klastrování a uživatel dostane na výběr, jestli chce v tomto jednom klastrování pokračovat. Pokud ano, zadá vyšší parametr klastrování a to proběhne opět na zbylém akumulátorovém poli. V tomto kroku se vyberou další, menší roviny. Tento krok je možné opakovat, dokud jsou segmentované roviny dostatečně kvalitní. Po ukončení tohoto cyklu se proces vrací na začátek, kdy se naklonuje nové pole, a je možné provést klastrování znovu od začátku s jinými parametry.

# Kapitola 4

## Testování

Testování algoritmu je provedeno dvěma způsoby. Nejprve je nutné na nějakých „jednoduchých“ datech ověřit, zda algoritmus funguje správně a chová se tak, jak očekáváme. Teprve poté může být algoritmus testován na reálných datech, kde se testuje jednak správnost výsledků a dále také výkon algoritmu. Testovat správnost segmentace však není úplně jednoduché, protože je složité nějak ohodnotit kvalitu segmentace. Jedním kritériem může být kompletnost - kolik bodů z celkového počtu všech bodů bylo zahrnuto do výsledných segmentů. Jenže v dodaných reálných datech je vždy poměrně velký počet bodů, které v žádných rovinách neleží a očekává se, že budou během segmentace zahrozeny. Takže tato hodnota má jen omezenou vypovídající hodnotu. Jako další kritérium pro testování se nabízí počet výsledných segmentů. Tato hodnota se dá porovnat s očekávaným počtem segmentů v daném mračnu. Zároveň se však musí porovnat očekávané segmenty jednotlivě, tedy jestli nalezený segment odpovídá danému očekávanému segmentu. Testování bude postaveno na těchto dvou kritériích. Samozřejmě bude také testován výkon, zajímat nás bude zejména doba segmentace a spotřeba paměti za běhu algoritmu.

Veškeré testování probíhalo v této konfiguraci:

- procesor: Intel Core 2 Duo T6670 2,20 GHz, 2MB L2 Cache, FSB 800 MHz
- paměť: 2 x 2GB DDR3 1066 MHz
- operační systém: OpenSUSE 12.1 32bit, jádro verze 3.3.3-21
- Java: verze 1.6.0\_24, Java HotSpot VM 20.4-b02

### 4.1 Ověření funkčnosti na umělých datech

Pro ověření funkčnosti jsem vytvořil jednoduchý generátor mračna bodů. Generátoru se zadá počet rovin, které chceme generovat, poté velikost těchto rovin a hustota bodů. Generátor vyvoří zadaný počet rovin v prostoru relativně blízko sebe, aby se některé roviny protínaly. Body v rámci roviny mají vždy nějakou malou dochylku, neleží přesně v rovině. Zdrojové kódy generátoru jsou na přiloženém CD, konkrétně se jedná o třídu

`PlaneGenerator` z balíčku `vicitis.inner.segmentation.pointcloudSegmentation`. Stejně tak vstupní a výstupní mračna ze všech testů jsou k dispozici na přiloženém CD, viz seznam souborů v kapitole [B](#).

Segmentování bylo provedeno za pomoci popsaného generátoru mračna bodů. Testoval se počet bodů, které byly z celkového počtu všech bodů segmentovány. V generovaných datech všechny body spadají do nějaké roviny, takže teoreticky by mohl algoritmus segmentovat vždy 100% bodů. Dalším kritériem testování byl počet bodů, které byly chybně přiřazeny do segmentu, do kterého nepatřily (nejčastěji body podél hran). Tato hodnota byla pouze řádově odhadnuta podle výsledků segmentace, není možné jednoduše určit přesnou hodnotu. Při této testovací segmentaci byla provedena celá segmentace najedou, parametr klastrování nebyl během segmentace upravován. Výsledky testování jsou uvedeny v tabulce [4.1](#).

test	parametry			vstup		výsledná segmentace			
	P1	P2	P3	roviny	bodů	segmentů	seg. bodů	chyb. bodů	
<b>1-0</b>	20	2	1	1	1800	1	1800	100,00%	0 0,00%
<b>1-1</b>	20	2	1	1	1800	1	1800	100,00%	0 0,00%
<b>2-0</b>	20	2	1	2	3600	2	3600	100,00%	0 0,00%
<b>2-1</b>	20	2	10	2	3600	2	3589	99,69%	5 0,14%
<b>3-0</b>	20	2	10	3	5400	3	5385	99,72%	15 0,28%
<b>3-1</b>	20	2	10	3	5400	3	5383	99,69%	20 0,37%
<b>4-0</b>	20	2	10	4	7200	4	7195	99,93%	20 0,28%
<b>4-1</b>	20	4	10	4	7200	4	7153	99,35%	10 0,14%
<b>5-0</b>	40	4	10	5	9000	5	8959	99,54%	10 0,11%
<b>5-1</b>	40	4	10	5	9000	5	8992	99,91%	250 2,78%
<b>6-0</b>	40	4	10	6	10800	6	10765	99,68%	30 0,28%
<b>6-1</b>	40	4	10	6	10800	6	10724	99,30%	80 0,74%
<b>7-0</b>	40	4	10	7	12600	7	12550	99,60%	250 1,98%
<b>7-1</b>	40	4	10	7	12600	7	12563	99,71%	20 0,16%

Tabulka 4.1: Tabulka s výsledky testování algoritmu na generovaných datech. P1 je parametr definující okolí, P2 parametr klastrování a P3 parametr určující kvalitu klastrování.

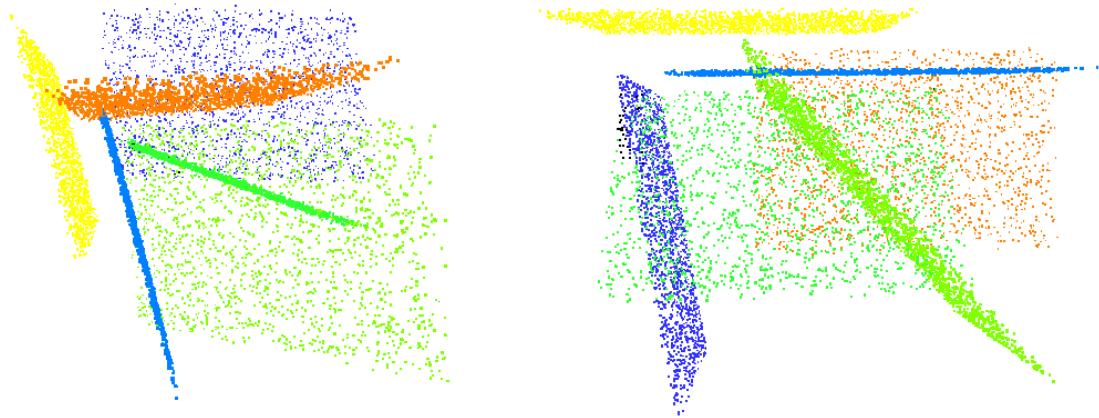
Jak je z tabulky patrné, ve všech testech byly všechny roviny rozděleny do segmentů. Ve všech případech bylo segmentováno více jak 99% vstupních bodů. Horší výsledky jsou však u chybných bodů. Ve dvou případech (měření 5-1 a 7-0) jsou tyto hodnoty řádově vyšší, než v ostatních případech. U těchto dvou testů nastal problém, kdy byly dva vrcholy v akumulátorovém poli blízko sebe a v první iteraci byl nalezen pouze jeden z nich. Při vybírání tohoto vrcholu byly vybrány i některé body z druhého vrcholu. Tento druhý vrchol byl poté nalezen až v druhé iteraci prohledávání pole a byl správně segmentován, nicméně již bez dříve odebraných bodů. Tento problém byl popsán v sekci [3.2](#) a znázorněn na obrázku [3.4](#). Řešení tohoto problému se mi bohužel nalézt nepodařilo, ve zmiňované kapitole jsou pouze popsány kroky, které minimalizují vznik problému. Je nutné poznamenat, že v tomto typu generovaných dat vzniká problém výrazně častěji, než na reálných datech. Generovaná data totiž obsahují velké množství rovin na malém prostoru a mračna obsahují jen malý počet bodů. Akumulátorové pole je tedy poměrně malé a je větší šance, že vrcholy budou v

poli blízko u sebe a zároveň budou tyto vrcholy reprezentovat roviny, které jsou velmi blízko v objektovém prostoru (to jsou podmínky, kdy tento problém nastává). Ukázka segmentace 7-0 je na obrázku 4.1. Ve všech ukázkách segmentací v této kapitole jsou jednotlivé segmenty označeny barevně (každý segment má jinou barvu). Pokud jsou v ukázce černé body, tyto body nebyly přiřazeny do žádného segmentu.



Obrázek 4.1: Ukázka testovací segmentace, která je v tabulce 4.1 označena jako 7-0. Vlevo je zobrazena výsledná segmentace, vpravo je detail na popsanou chybu při segmentaci.

Pro ilustraci přidávám ještě jednu ukázku testovací segmentace, na obrázku 4.2 je mračno označené v tabulce jako 6-0.



Obrázek 4.2: Ukázka testovací segmentace, která je v tabulce 4.1 označena jako 6-0. Jedná se o stejné mračno z různých úhlů.

Vzhledem k tomu, že ve všech testovacích případech byly nalezeny všechny roviny, algoritmus rozhodně funguje správně. Zároveň ve všech případech bylo segmentováno více jak 99,3% bodů. Ve většině případů byl počet špatně přiřazených bodů velmi malý, kromě dvou již zmínovaných případů.

## 4.2 Testování kvality algoritmů na reálných datech

V této části jsou otestovány oba algoritmy na reálných mračnech bodů, které jsem měl k dispozici. Testován je jednak výkon algoritmů a dále také kvalita samotné segmentace. Všechna testovaná mračna jsou k dispozici na přiloženém CD, stejně tak CD obsahuje výsledky všech segmentací v této kapitole. Výsledky segmentace budou vždy shrnuté v tabulce, kde bude mít každý test unikátní označení. Soubory na CD budou pro přehlednost označeny stejně, takže vždy půjde jednoduše přiřadit soubor na CD k datům popisovaným v textu.

Během testování algoritmu bude nejprve na dvou různých mračnech bodů ukázáno, jaký vliv mají parametry segmentace na dobu a výstup segmentace. Poté bude ukázáno, jaký je vliv parametrů segmentace na dobu běhu algoritmu a jaká je spotřeba paměti algoritmu. Dále budou v kapitole ukázány výsledky segmentace z dalších mračen bodů, zde už budou ukázány pouze výsledky s nejlepšími parametry.

### 4.2.1 Vliv parametrů na výslednou segmentaci

Pro ukázkou vlivu parametrů na výslednou segmentaci jsem vybral dvě různá mračna (viz Obr. 4.3). První mračno je označeno jako „dům 2“, toto mračno je složeno z 723695 bodů a obsahuje 3 velké roviny. Oproti tomu druhé mračno „stodůlky“ obsahuje více jak 10 rovin a skládá se ze 604411 bodů. Obě mračna reprezentují část budovy, ale zároveň obsahují poměrně velké množství bodů, které do žádné roviny nepatří.



Obrázek 4.3: Ukázka mračna bodů dům 2 (a) a stodůlky (b).

Cílem tohoto testu je ukázat, jak jednotlivé parametry ovlivňují výslednou segmentaci. Cílem tedy není docílit co nejlepší segmentace, toho se bude týkat až další část testování.

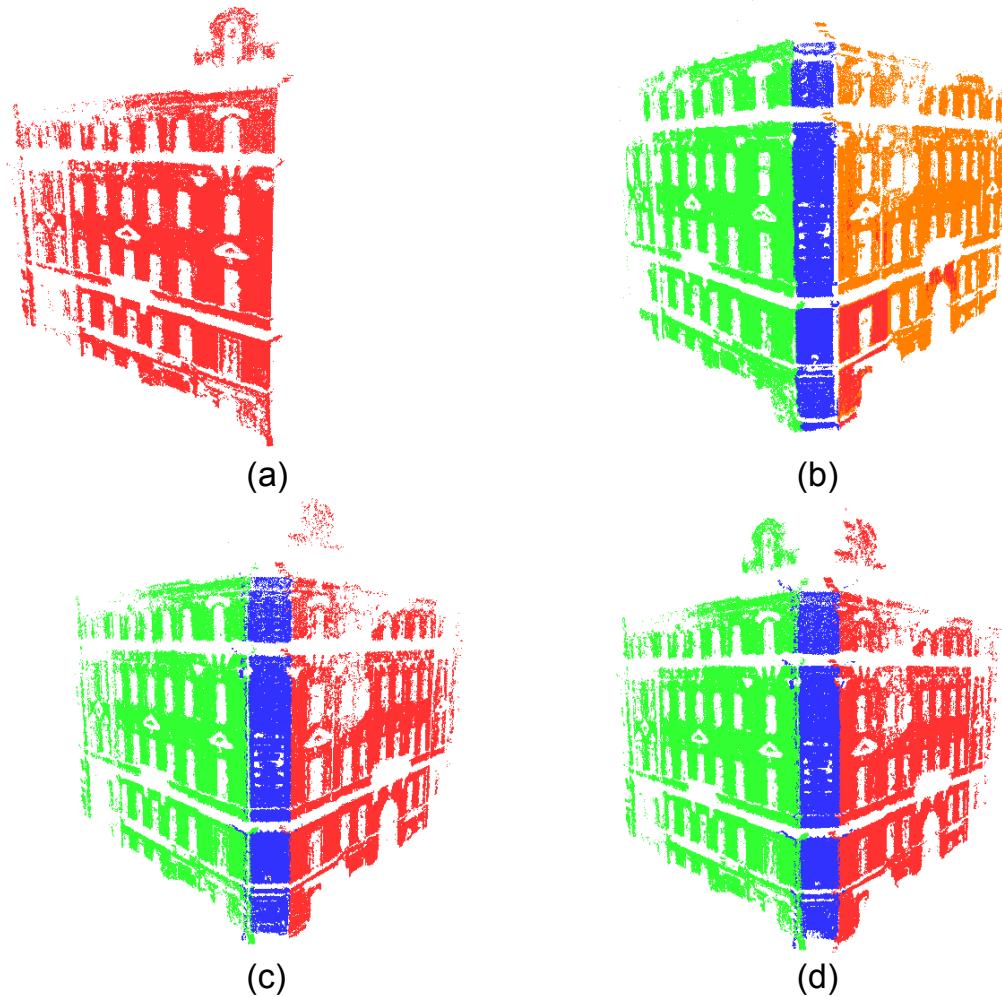
Nejprve bude testováno mračno dům 2. Tabulka 4.2 ukazuje výsledky testování mračna. Na obrázku 4.4 jsou zobrazeny některé z výsledků tohoto testování.

test	P1	P2	P3	seg. bodů	segmentů
d2-0.00	10	1	1	25,68%	1
d2-0.01	10	1	10	31,37%	1
d2-0.02	10	2	1	54,53%	3
d2-0.03	10	2	2	56,65%	3
d2-0.04	10	2	3	60,09%	3
d2-0.05	10	2	5	63,00%	3
d2-0.06	10	2	10	63,36%	3
d2-0.07	10	2	20	63,79%	3
d2-0.08	10	3	1	53,75%	3
d2-0.09	10	4	1	53,71%	3
d2-0.10	10	5	1	53,75%	3
d2-0.11	10	10	1	57,00%	4
d2-0.12	20	2	1	54,79%	3
d2-0.13	40	2	1	56,91%	3
d2-0.14	80	2	1	61,12%	4
d2-0.15	120	2	1	54,38%	4

Tabulka 4.2: Tabulka zobrazuje výsledky testování algoritmu na mračnu dům 2 při různých parametrech. Parametry jsou popsány v příloze A.

Protože mračno obsahuje poměrně přesné roviny bez větších chyb a nerovností, pro testování stačilo i velmi malé okolí, segmentace fungovala správně již při okolí 10 bodů. Z tabulky by mělo být patrné, že pokud byl zadán parametr pro klastrování 1, byla tato hodnota příliš nízká a byl rozpoznán pouze jeden segment. Nicméně při zvýšení této hodnoty na 2 byly nalezeny již všechny tři očekávané segmenty. Při této hodnotě jsem dále měnil parametr určující kvalitu klastrování. Podle očekávání při zvýšení tohoto parametru narůstal počet bodů obsažených v segmentech. Při zvýšení parametru z 1 na 20 bylo segmentováno o 9% bodů více. Jak je ale z tabulky patrné, největší nárůst přidaných bodů byl při postupném zvyšování parametru od 1 do 5, následné zvyšování parametru již nemělo na segmentaci výrazný vliv. Pro běžné podmínky tedy nemá smysl tento parametr nastavovat na vyšší hodnotu než 10. V dalších testech byl zvyšován parametr klastrování (jde o data označená d2-0.8 - d2-0.11). Z tabulky je patrné, že se při počátečním zvyšování parametru výsledná segmentace výrazně neměnila. To je logické, protože při zvyšování parametru se snížovala hranice pro vyhledávání vrcholů v akumulátorovém poli, ale protože body ze tří velkých rovin vytvořily tři vysoké vrcholy, byly nalezeny stejně při všech těchto hodnotách parametru. Až při hodnotě 10 byl práh při prohledávání pole tak nízko, že našel nějaká zbytková data tvořící rovinu. Tato situace bude popsána v následujícím odstavci.

V posledních 4 řádcích tabulky byla postupně zvyšována velikost okolí. Zvýšení na 20 a 40 bodů nemělo na segmentaci žádný výrazný vliv, segmentace byla jen o málo přesnější, což se projevilo ve zvýšeném počtu segmentovaných bodů o několik procent. Při dalším zvyšování ve-



Obrázek 4.4: Ukázka vybraných mračen bodů z testování. Obrázky odpovídají značení v tabulce: (a) d2-0.01, (b) d2-0.14, (c) d2-0.02, (d) d2-0.07.

likosti okolí však bylo nalezeno více segmentů, než se očekávalo. Toto je paradoxně způsobeno tím, že parametry bodů byly při větším okolí vypočteny přesněji. Konkrétní problém je v tom, že body tvořící nejvyšší buňku vrcholu jsou velmi přesně v rovině. Na základě těchto bodů je vypočteno RMS, které v následném klastrování určuje „tloušťku“ roviny, tedy přesněji vzdálenost, v jaké může bod od roviny být, aby byl ještě do segmentu zahrnut. Pokud je ale toto RMS malé (protože původní body byly velmi přesně v rovině), jsou vybrány pouze ty nejpřesnější body. To má za následek, že v akumulátorovém poli zůstanou body, které tvoří rovinu, akorát méně přesnou. Bodů je však stále ještě dostatek, aby byl vrchol v akumulátorovém poli výrazný. Tím pádem je vrchol v další iteraci prohledávání akumulátorového pole detekován a z bodů je vytvořen nový segment. Výsledkem je tedy to, že z jedné roviny jsou vytvořeny dva segmenty. První segment obsahuje nejpřesnější body z této roviny, druhý obsahuje zbylé, méně přesné body. Tento problém se při klastrování velmi těžko detekuje. Je založen na faktu, že vypočtené RMS je malé. Nicméně v rámci algoritmu

neexistuje žádná referenční hodnota, podle které by šlo rozhodnout, že RMS je malé. Proto se mi tento problém nepodařilo detekovat během klastrování. Řešením by mohla být dodatečná detekce po klastrování, kdy by se zjišťovalo, zda jednotlivé segmenty nereprezentují stejnou rovinu. Řešení ani detekce tohoto problému nejsou v této práci implementovány. Jako možné řešení se nabízí sloučení segmentů ležících v jedné rovině. Toto by šlo provést až na základě výsledků segmentace bez nutnosti zashovat do hotového algoritmu.

Celkově lze tedy říci, že algoritmus je poměrně robustní pro mračno s malým počtem velkých rovin. Není výrazně závislý na zadané velikosti okolí, až při větších okolí nastává výše popsaný problém. Zároveň není algoritmus příliš závislý na parametru klastrování, fungoval správně od hodnoty 2 do hodnoty 10. Při této hodnotě však již byl nalezen jeden segment navíc. I v tomto případě byly 2 segmenty součástí jedné roviny jako ve výše popsané situaci. Příčina zde ale byla trochu jiná. Protože kvalita segmentů byla nastavena na 1, po vybrání 3 největších segmentů stále v akumulátorovém poli zbyly zbytky těchto rovin, které nebyly segmentovány. A protože byl parametr klastrování nastaven na 10, práh pro prohledávání pole byl tak nízký, že tyto zbytky byly detekovány jako segment. Řešením této situace je snížit parametr klastrování nebo zvýšit kvalitu klastrování.

Za zmínku ještě stojí poměrně nízký počet segmentovaných bodů, který byl v nejlepším případě necelých 64% bodů. To je dáno tím, že mračno obsahuje poměrně velký počet bodů, které v daných rovinách neleží (viz Obr. 4.3 (a)).

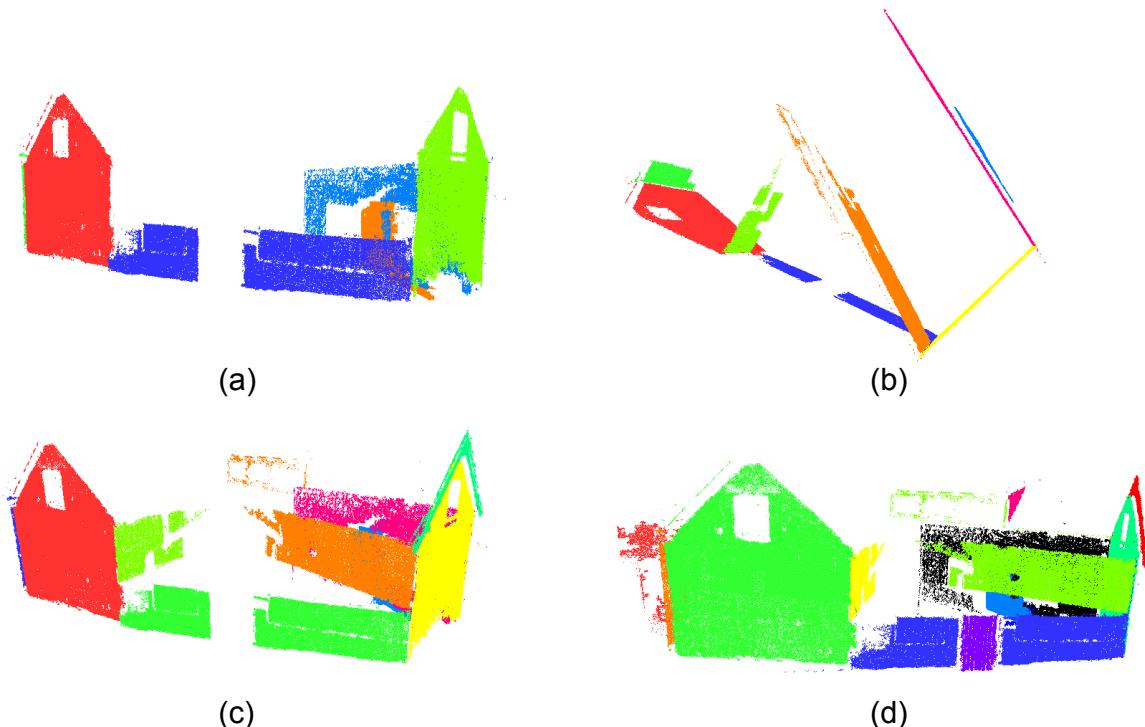
Pro druhou ukázku vlivu parametrů na segmentaci použiji mračno stodůlky. Na tomto mračnu bude ukázáno, jaký vliv má parametr klastrování na výslednou segmentaci. Výsledky testování jsou uvedeny v tabulce 4.3.

test	P1	P2	P3	seg. bodů	segmentů
st-0.00	30	1	5	37,42%	3
st-0.01	30	3	5	55,24%	6
st-0.02	30	4	5	61,33%	8
st-0.03	30	5	5	62,51%	9
st-0.04	30	6/7	5	65,42%	11
st-0.05	30	8	5	66,10%	12
st-0.06	30	9/10	5	66,40%	13
st-0.07	30	11	5	66,50%	14

Tabulka 4.3: Tabulka zobrazuje výsledky testování algoritmu na mračnu stodůly při různých parametrech klastrování. Parametry jsou popsány v příloze A.

Z tabulky je jasné vidět, že při zvyšování parametru klastrování postupně přibývá počet nalezených segmentů. Nejprve jsou nalezeny segmenty s největším počtem bodů, případně segmenty s body, které jsou velmi přesně v rovině. Při zvyšování jsou postupně nacházeny menší nebo méně kvalitní roviny. V tabulce můžeme vidět, že při zvyšování parametru až na hodnotu 8 jsou nacházeny segmenty, které řádově přidávají aspoň procenta bodů z celkového mračna. Při dalším zvyšování už jsou nalezeny pouze velmi malé segmenty, dá se tedy očekávat, že při vyšší hodnotě parametru jak 8 už nebudou nalezeny žádné významné segmenty ale pouze již nějaké velmi malé segmenty, případně budou chybět segmentovány

zbytky již nalezených segmentů, jak bylo popsáno dříve v této kapitole. Na obrázku 4.5 jsou zobrazeny ukázky některých testovacích segmentací.

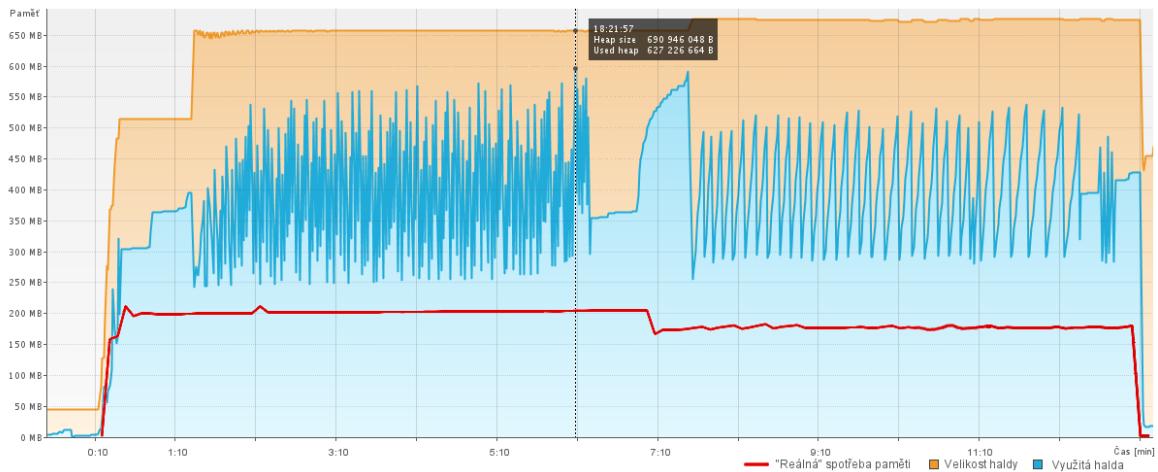


Obrázek 4.5: Ukázka vybraných mračen bodů z testování z různých pohledů. Obrázky odpovídají značení v tabulce: (a) st-0.01, (b) st-0.02, (c) st-0.03, (d) st-0.06.

Z obrázků je patrné, že algoritmus nedetekoval dvě střechy, které jsou rovinné a obsahují poměrně velké množství bodů (viz Obr. 4.3). Tento problém nastal proto, že na velkém měřítku střechy vypadají rovně, ale jejich povrch je ve skutečnosti hodně hrbolatý, na malém měřítku tedy není dostatečně rovný. Parametry jednotlivých bodů na střeše jsou tím pádem poměrně rozdílné a body ze střechy nevytvorí v akumulátorovém poli dostatečně výrazný vrchol. Aby byl algoritmus schopen střechy detektovat, je potřeba použít výrazně větší okolí, než 30 bodů, které bylo použito v tomto testu. Nicméně jak jsem zmiňoval na začátku této sekce, cílem těchto dvou testů bylo ukázat, jak parametry ovlivňují segmentaci, nikoliv získat při segmentaci co nejlepší výsledky. Na to dojde v pozdější části této kapitoly.

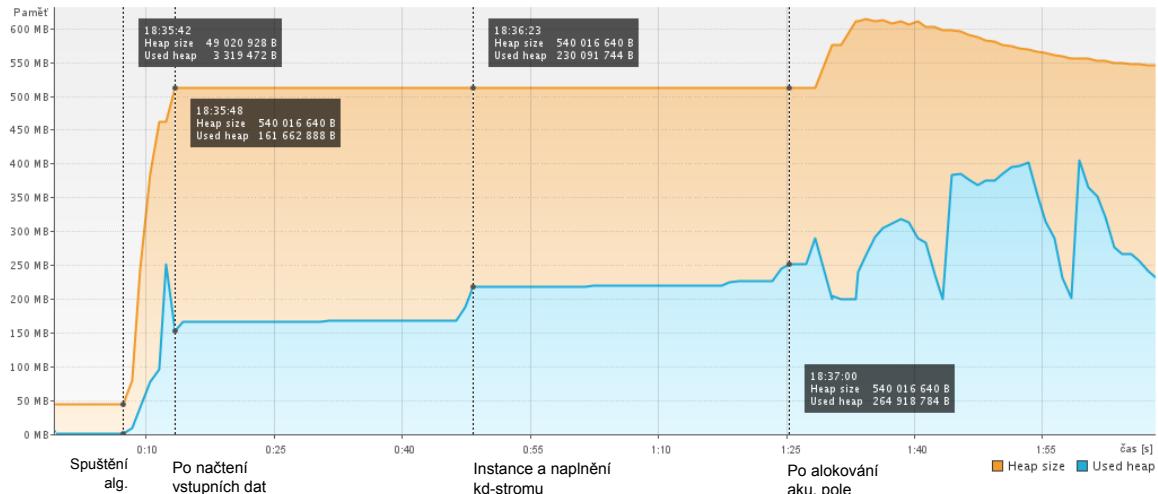
### 4.2.2 Testování časové a paměťové složitosti algoritmu

Protože je algoritmus poměrně komplikovaný a používá složitější datové struktury, nejde příliš jednoduše odhadnou asymptotickou složitost algoritmu. Proto jsem pouze měřil paměť za běhu programu a dobu běhu programu. Pro sledování alokované paměti jsem použil program VisualVM verze 1.3.4. Nejprve jsem testoval segmentaci mračna bodů dům 2 (723695 bodů) s velikostí okolí 80 bodů, parametrem klastrování 1 a kvalitou klastrování 10. Výsledky měření jsou zachyceny v grafu na obrázku 4.6.



Obrázek 4.6: Graf zobrazující velikost využité paměti za běhu algoritmu.

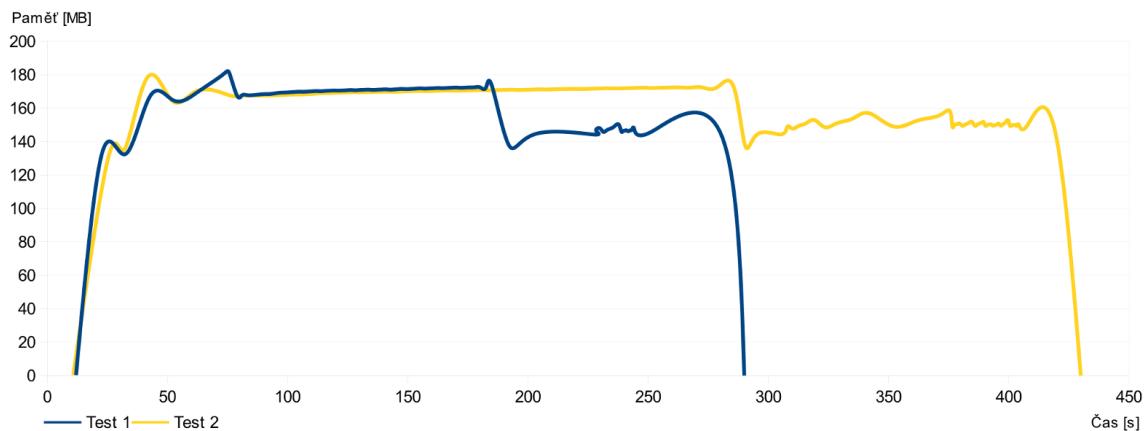
Graf zobrazuje velikost alokované haldy a její reálné využití za běhu aplikace. Detail začátku grafu s popisem je zobrazen na obrázku 4.7. Graf je ovšem z jiného měření při stejných datech, takže se jeho hodnoty mohou nepatrně lišit.



Obrázek 4.7: Graf zobrazující velikost využité paměti při inicializaci algoritmu.

Důležité hodnoty v prvním grafu jsou v čase 1:30, kdy začal výpočet parametrů a dále čas 6:50, kdy začalo vybírání bodů z akumulátorového pole. To skončilo v čase 12:30, poté byl vyexportován výstup a v čase 13:00 byl algoritmus ukončen. Protože se během výpočtu parametrů bodů a při vybírání bodů z akumulátorového pole používá velké množství krátkodobých instancí, je výsledný graf velmi ovlivněn chováním garbage collectoru. Proto jsem provedl druhé měření, kdy jsem před každým odečítáním paměti explicitně volal garbage collector - to tedy odpovídá hodnotě paměti, kterou zabírají „živé“ instance. Toto měření je v grafu zakreslené červenou křivkou. Protože šlo o dvě různá měření, grafy se mohou na ose x vzájemně lišit, nicméně maximálně o několik vteřin.

Z grafů by mělo být patrné, že velikost potřebné paměti je nejvíce závislá na množství vstupních dat, algoritmus poté potřebuje další paměť na vytvoření kd-stromu a alokaci akumulátorového pole, ale dále za běhu již nealokuje žádné výrazné množství paměti. Na paměťových nárocích by se tedy neměly výrazně projevit ani použité parametry. Na obrázku 4.8 je graf, který zobrazuje využitou paměť při segmentaci mračna stodůlky s různými parametry. Před každým odebraným vzorkem paměti byl volán garbage collector. První test zobrazuje segmentaci s velikostí okolí 20, parametrem klastrování 8 a kvalitou klastrování 1, druhý test je proveden s parametry 80, 8 a 10. Z grafu je patrné, že alokovaná paměť je v obou případech stejná. Pokles paměti v prostřední části grafu odpovídá ukončení ukládání bodů do akumulátorového pole a začátku vybírání segmentů.

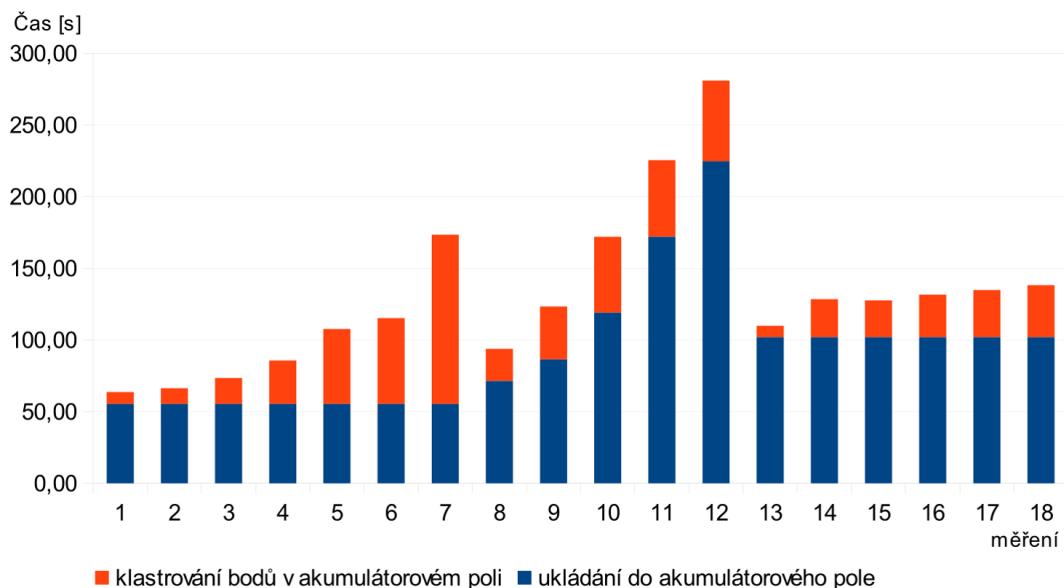


Obrázek 4.8: Graf zobrazující velikost využité paměti při spuštění algoritmu s různými parametry.

Z grafu je také jasné vidět, že hodnota parametrů ovlivňuje zejména dobu běhu algoritmu. To však není žádné překvapení, důvody jsou popsány v kapitole 3. Následující testování se tedy bude věnovat měření doby běhu algoritmu při různých vstupních parametrech. Zajímavé bude také srovnání, jaká část z celkové doby běhu algoritmu odpovídá vyhledávání okolí v kd-stromu a počítání SVD dekompozice při prokládání roviny množinou bodů. Testování proběhlo pouze na mračnu stodůlky. Výsledky testování rychlosti algoritmu jsou shrnutы v tabulce 4.4.

test	Parametry			Čas [sec]							
	P1	P2	P3	celkem	ukládání do aku.	vybírání z aku.	kd-strom	SVD			
1	10	8	1	64,34	55,48	86,24%	8,21	12,76%	12,82	19,93%	17,76 27,60%
2	10	8	2	66,97	55,48	82,85%	10,84	16,18%	12,82	19,15%	17,82 26,61%
3	10	8	3	73,48	55,48	75,51%	18,00	24,49%	12,82	17,45%	17,90 24,36%
4	10	8	5	85,70	55,48	64,74%	30,22	35,26%	12,82	14,96%	17,91 20,90%
5	10	8	8	107,66	55,48	51,54%	52,17	48,46%	12,82	11,91%	17,92 16,64%
6	10	8	10	115,26	55,48	48,14%	59,78	51,86%	12,82	11,12%	17,93 15,56%
7	10	8	20	173,47	55,48	31,99%	117,98	68,01%	12,82	7,39%	17,94 10,34%
8	20	8	5	93,84	71,50	76,19%	22,34	23,81%	18,07	19,26%	23,58 25,13%
9	30	8	5	123,38	86,55	70,15%	36,83	29,85%	23,78	19,27%	26,42 21,41%
10	50	8	5	172,05	119,14	69,25%	52,91	30,75%	34,93	20,30%	37,08 21,55%
11	80	8	5	225,54	172,19	76,34%	53,35	23,66%	54,56	24,19%	49,84 22,10%
12	100	8	5	281,07	224,82	79,99%	56,24	20,01%	71,51	25,44%	72,01 25,62%
13	40	1	5	109,90	102,05	92,86%	7,85	7,69%	28,53	27,96%	31,17 30,54%
14	40	2	5	128,48	102,05	79,43%	26,43	20,57%	28,53	22,21%	31,19 24,27%
15	40	3	5	127,64	102,05	79,95%	25,59	20,05%	28,53	22,35%	31,21 24,45%
16	40	4	5	131,66	102,05	77,51%	29,62	22,49%	28,53	21,67%	32,36 24,58%
17	40	6	5	134,90	102,05	75,65%	32,85	24,35%	28,53	21,15%	33,35 24,72%
18	40	8	5	138,28	102,05	73,80%	36,23	26,20%	28,53	20,63%	33,37 24,13%

Tabulka 4.4: Tabulka zobrazuje výsledky testování rychlosti algoritmu v závislosti na zadaných parametrech. Parametry jsou popsány v příloze A.



Obrázek 4.9: Graf zobrazující dobu běhu algoritmu v závislosti na zadaných parametrech. V grafu jsou znázorněna data z tabulky 4.4.

Některá data z tabulky jsou zachycena v grafu na obrázku 4.9, konkrétně je v grafu znázorněn celkový čas segmentace a zároveň část, kterou zabralo ukládání bodů do akumulátorového pole a poté klastrování bodů v poli. Prvních 7 řádků v tabulce (tedy prvních 7

sloupců v grafu) zobrazuje změnu doby běhu algoritmu při zvyšování kvality klastrování. Jak se dá očekávat, zvýšení parametru zvyšuje dobu trvání klastrování bodů v poli. Ukládání do pole není ovlivněno vůbec (bylo provedenou pouze jednou, proto jsou u všech měření hodnoty totožné). V následujících 5ti měřeních je zvyšována velikost okolí bodu při ukládání bodů do akumulátorového pole. Opět se podle očekávání zvyšuje doba běhu se zvětšováním okolí. Doba klastrování se liší jen podle toho, jaká byla kvalita nalezených segmentů. V posledních 6ti měřeních je měněn parametr klastrování. Z naměřených dat je zřetelné, že tento parametr nemá na dobu běhu zásadní vliv. Mění se pouze doba klastrování bodů v akumulátorovém poli, která je závislá na počtu nalezených segmentů<sup>1</sup>.

#### 4.2.3 Výsledky segmentace dodaných dat

V této části budou ukázány nejlepší výsledky segmentace všech dodaných testovacích mračen bodů. U každého testu segmentace je uvedena tabulka s výsledky. Tabulka zobrazuje, kolik bodů z celkového počtu bylo segmentováno. Dále je v tabulce uvedeno, kolik segmentů bylo nalezeno, kolik segmentů bylo očekáváno<sup>2</sup> a také kolik nalezených segmentů je správných - tím je myšleno, jestli segment odpovídá jedné rovině v mračnu bodů. Dále je zde také uveden celkový čas segmentace, který je ještě rozdělen na dvě části - uložení bodů do akumulátorového pole a klastrování bodů v poli. Také je zde pro zajímavost uveden čas, který zabralo vyhledávání sousedů v kd-stromu a SVD dekompozice. Tyto dvě hodnoty jsou již zahrnuty v celkovém času.

Protože kvalitu segmentace lze jen těžko ohodnotit podle jednoduše měřitelných kritérií, bude u každého testovaného mračna zobrazen obrázek segmentovaných dat z několika úhlů. Každý segment je označen barevně (body stejně barvy patří do jednoho segmentu). Samozřejmě výsledek každé segmentace je na přiloženém CD, pro každý test jsou přiloženy dva soubory, jeden obsahuje pouze obarvená segmentovaná data, druhý navíc obsahuje i body, které nebyly segmentovány.

---

<sup>1</sup>Tato hodnota zde není uvedena, ale je totožná s daty uvedenými v tabulce 4.3

<sup>2</sup>Tato hodnota odpovídá počtu rovin v mračnu. Je to také trochu odhad, záleží na tom, co ještě za považujeme rovinu a jaké roviny tedy chceme od algoritmu detektovat.

### 4.2.3.1 Dům 2

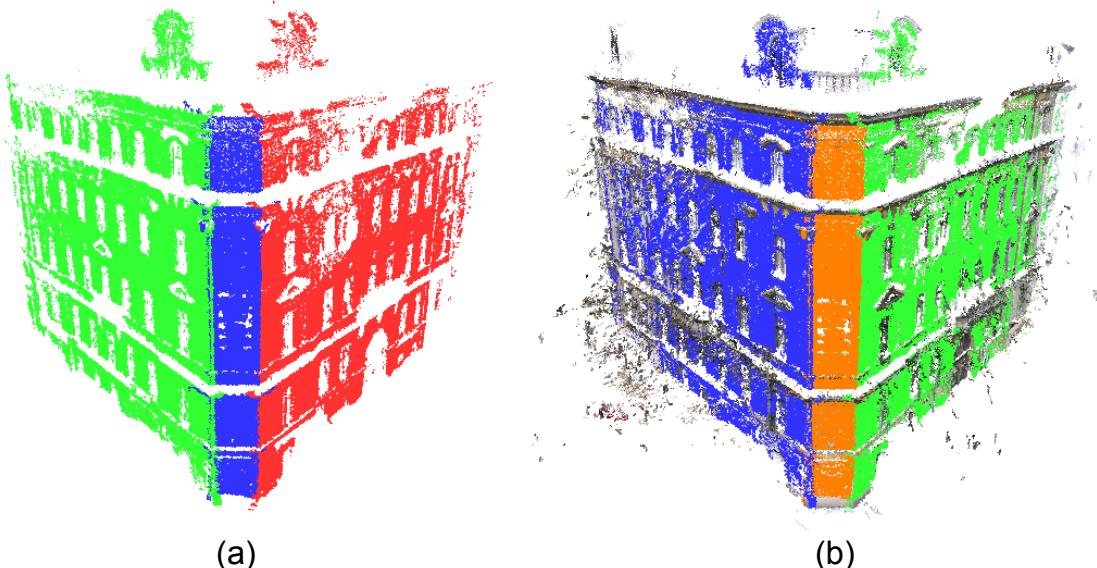
Prvním testovaným mračnem je dům 2. Vstupní mračno je zobrazeno na obrázku 4.3. Parametry segmentace a měřitelné výsledky segmentace jsou uvedeny v tabulce 4.5. Ukázka segmentovaných dat je zobrazena na obrázku 4.10.

test	Parametry			Body		Segmenty			...
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných	
d2-1.0	10	2	10	723695	488026   67,44%	3	3	3	...

...	Čas [s]				
...	Celkem	Segmentace	Klastrování	kd-strom	SVD
...	164,92	67,35	97,58	14,25	22,14

Tabulka 4.5: Tabulka zobrazuje informace o segmentaci mračna dům 2. Parametry jsou popsány v příloze A.



Obrázek 4.10: Segmentace mračna dům 2. (a) zobrazuje pouze segmentované body, (b) zobrazuje kromě segmentů i body, které nebyly segmentovány.

Výsledek segmentace považuji za správný a bez větších problémů. Bylo segmentováno 67% ze všech bodů, zde by byl možná ještě prostor pro zlepšení, pravděpodobně by bylo možné přidat ještě cca 5% bodů, které do segmentů patří. Zbylé body jsou však jen šum a do žádných rovin nepatří, jak je vidět na obrázku 4.10 (b).



### 4.2.3.2 Stodůlky

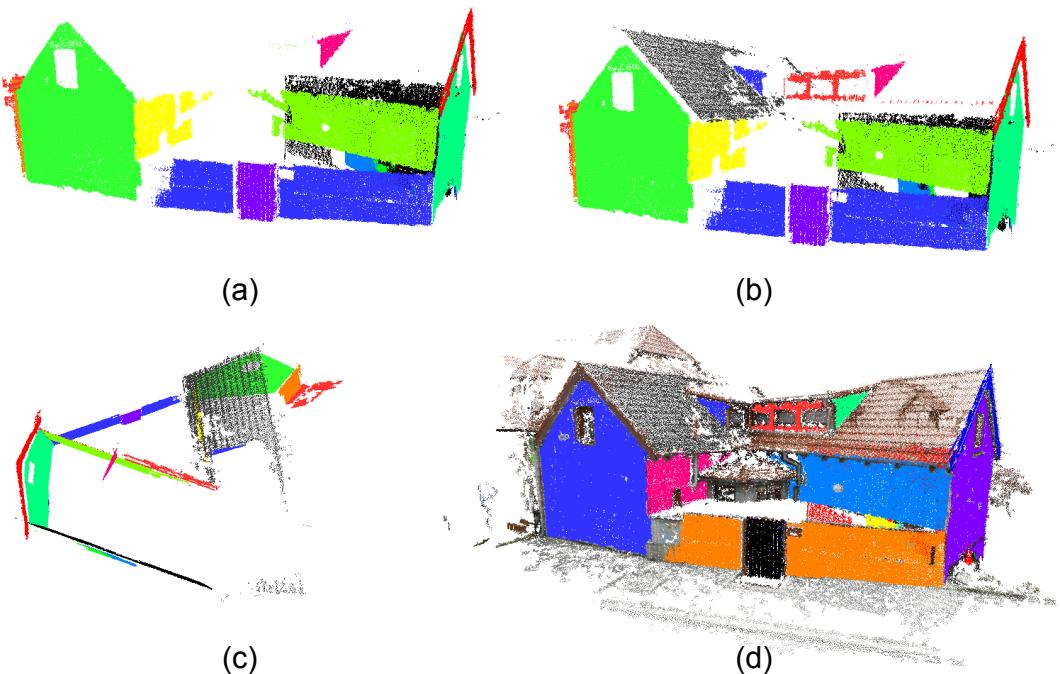
Vstupní mračno je zobrazeno na obrázku 4.3. Segmentace byla v tomto případě provedena ve dvou krocích, nejprve bylo mračno segmentováno s parametrem klastrování 9, poté bylo provedeno klastrování ve zbylém akumulátorovém poli s parametrem 12. Výsledky segmentace jsou uvedeny v tabulce 4.6, ukázka segmentovaných dat je na obrázku 4.11.

test	Parametry			Body		Segmenty			...	
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných		
st-1.0	48	9	10	604411	406188	67,20%	14	21	12	...
st-1.1	48	12	10	604411	16408	2,71%	3	7	3	...
celkem:				604411	422596	69,92%	17	21	15	...

...	Čas [s]				
	Celkem	Segmentace	Klastrování	Kd-strom	SVD
...	190,39	118,19	72,21	34,62	36,47
...	15,07	0	15,07	0	0,02
...	205,46	118,19	87,27	34,62	36,49

Tabulka 4.6: Tabulka zobrazuje informace o segmentaci mračna stodůlky. Parametry jsou popsány v příloze A.



Obrázek 4.11: Segmentace mračna stodůlky. (a) zobrazuje první část segmentace, (b) zobrazuje celkovou segmentaci, (c) zobrazuje pohled na celkovou segmentaci shora, (d) zobrazuje celkovou segmentaci včetně bodů, které nebyly segmentovány.

Tato segmentace již není tak bezproblémová, jako ta předchozí. Na první pohled je patrné, že střecha na pravé části domu nebyla rozeznána. To je způsobeno tím, že povrch střechy je hrbolatý a proto není výpočet parametrů příliš přesný. Navíc střecha obsahuje menší počet bodů, než by se na první pohled zdálo (body jsou zde řidší, než na stěnách). Z těchto dvou důvodů body ze střechy nevytvoří v akumulátorovém poli dostatečný vrchol, aby byl segment rozeznán. V tomto případě nepomůže ani výrazně zvětšení parametru definujícího okolí. Dále jsem uvedl 21 očekávaných segmentů a nalezených pouze 15. Nebylo tedy nalezeno ještě 5 dalších segmentů - nicméně tyto plochy jsou již velmi malé a algoritmus je v zašuměném akumulátorovém poli nenašel. Dále byly nalezeny 2 segmenty, které jsem neoznačil jako správné. Tyto segmenty byly vytvořeny ze „zbytků“ jiného segmentu při iterativním prohledávání pole - vznikla tedy situace, kdy jsou nalezeny 2 segmenty v jedné rovině - tento problém byl již popsán dříve, v sekci 4.2.1. Jeden z těchto segmentů je na zadní straně mračna (zelená a modrá stěna), druhý chybný segment je příliš malý, aby byl na obrázku zřetelný.

V tomto případě je za nenalezené segmenty zodpovědné větší množství šumu a duplicita v datech. Šum nedovoluje prohledávat akumulátorové pole příliš podrobně a mračno navíc obsahuje velké množství duplicitních bodů (obě čelní stěny jsou v mračnu dvakrát). To zbytečně zvyšuje velikost akumulátorového pole, tím pádem nejsou vrcholy v poli tak výrazné. Mračno stodůlky je rozděleno do tří souborů. Jeden z těchto souborů obsahuje polovinu bodů, ale naprostou většinu rovin. Navíc neobsahuje duplicitní body (jednotlivé soubory se překrývají, tím vznikají duplicita). Pokud provedu segmentaci pouze na tomto jednom souboru, výsledky jsou výrazně lepší, viz tabulka 4.7 a obrázek 4.12.

test	Parametry			Body		Segmenty			...	
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných		
st-2.0	70	12	2	291232	215678	74,06%	18	19	18	...

...	Čas [s]				
	Celkem	Segmentace	Klastrování	kd-strom	SVD
	80,45	69,42	11,03	19,22	22,49

Tabulka 4.7: Tabulka zobrazuje informace o segmentaci části mračna stodůlky.



Obrázek 4.12: Segmentace mračna stodůlky. (a) zobrazuje pouze segmentaci, (b) zobrazuje segmentaci včetně bodů, které nebyly segmentovány.

V tomto případě byly nalezeny správně všechny rovinné segmenty kromě malé střechy vlevo nahoře, která obsahuje jen velmi málo bodů a není příliš rovná. Z testu vyplývá, že algoritmus je citlivý na různou hustotu bodů v rámci jednotlivých rovin (způsobenou duplicitami v původním mračnu), a že šum v datech může zhorsit výsledky segmentace.

### 4.2.3.3 Červená Lhota

Toto mračno reprezentuje zámek Červená Lhota. Mračno obsahuje poměrně velké množství bodů, které jsou velmi daleko od snímaného objektu (šum) a výrazná část bodů je obsažena mimo stěny budovy, viz obrázek 4.13. Mračno je tedy poměrně odlišné od dvou předchozích a bude zajímavé, jak si s ním algoritmus poradí.



Obrázek 4.13: Ukázka mračna Červená Lhota.

Výsledky segmentace jsou zaznamenány v tabulce 4.8 a ukázka výsledné segmentace z různých úhlů je vidět na obrázku 4.14. Obrázky této segmentace bohužel nejsou tolik přehledné jako u předešlých testování, lepší je prohlédnou si celé mračno na přiloženém CD.

test	Parametry			Body		Segmenty			...		
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných			
cl-0.0	60	10	5	611095	331336	54,22%		16	18	16	...

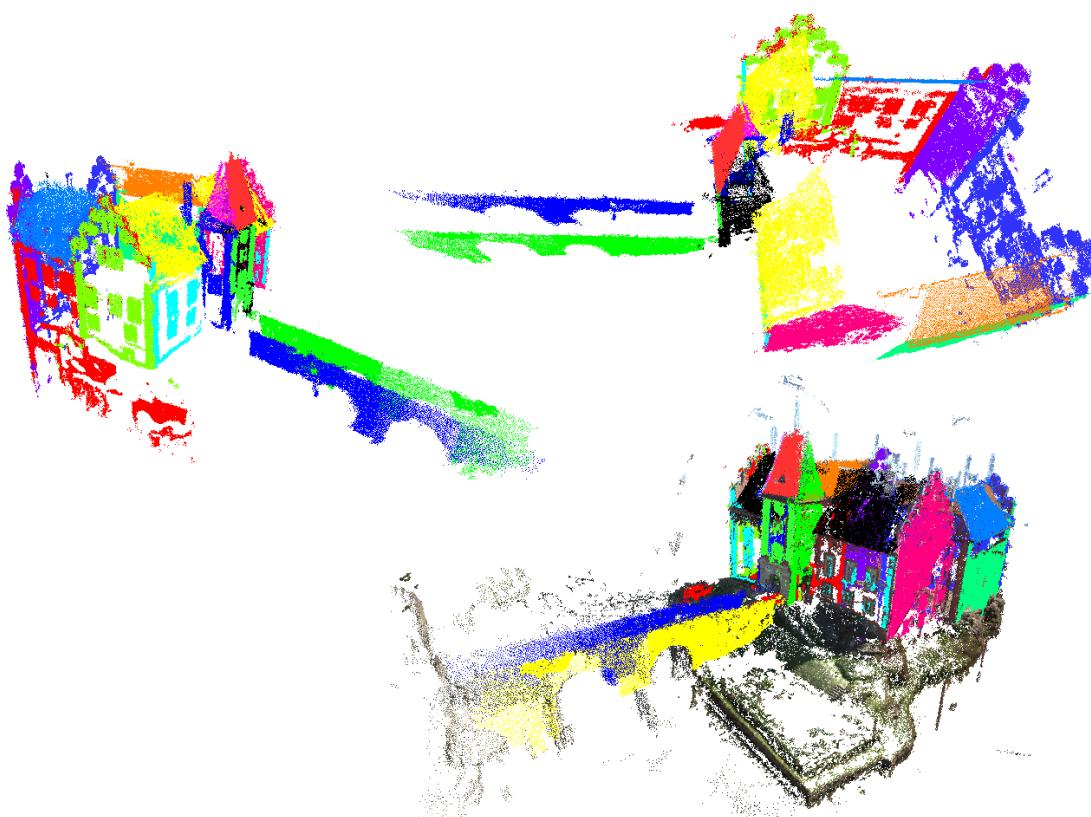
  

...	Čas [s]				
	...	Celkem	Segmentace	Klastrování	kd-strom
		328,81	163,63	165,18	48,19
	...	50,83			

Tabulka 4.8: Tabulka zobrazuje informace o segmentaci mračna bodů Červená Lhota. Parametry jsou popsány v příloze A.

Segmentace identifikovala téměř všechny roviny v rámci hlavního objektu. V tabulce je uvedeno, že nebyly nalezeny 2 roviny. Algoritmus nenašel střechu na zadní straně. Střechu našel při zvýšení parametru klastrování, ale to se již také začaly objevovat chybné segmenty, proto jsem vybral na ukázku tuto segmentaci, i když tuto jednu rovinu nenalezla. Poté ještě nebyla malá rovina spojující boční stěny na pravé straně zámku, ta obsahuje poměrně málo bodů, pravděpodobně proto již nebyla při segmentaci rozpoznána.

Dále by se mohlo zdát, že z čelní a levé strany chybí v segmentech body. To ale není chyba algoritmu, v těchto stěnách je takto málo bodů už ve vstupním mračnu, algoritmus



Obrázek 4.14: Ukázka segmentace mračna Červená Lhota z různých úhlů.

jich naopak segmentoval naprostou většinu. Výraznější chyba v segmentaci je červená rovina vlevo, která zahrnuje i body, které již nepatří do stěny, ale jde o část skály. Rovina měla pravděpodobně již od začátku větší RMS, proto se takto „rozrostla“ i o body, které do ní přímo nepatří, včetně několika bodů ze sousedních stěn. Takovéto chybě se nedá jednoduše vyhnout, pokud je již ze začátku kvalita roviny (velikost RMS) horší, rovina často obsáhne i okolní body, které do ní již přímo nespadají.

Celkově bych však segmentaci ohodnotil jako úspěšnou, kromě dvou menších rovin byly na objektu rozpoznány všechny stěny. Segmenty navíc obsahují naprostou většinu bodů z těchto rovin. Segmentace také nebyla viditelně ovlivněna větším množstvím chybných bodů mimo roviny.

#### 4.2.3.4 Doudleby

Toto mračno reprezentuje zámek Doudleby nad Orlicí. Mračno je poměrně velké a obsahuje několik výrazných rovin, viz obrázek 4.15.



Obrázek 4.15: Ukázka mračna Doudleby.

Výsledky segmentace jsou zaznamenány v tabulce 4.9 a ukázka výsledné segmentace z různých úhlů je vidět na obrázku 4.16.

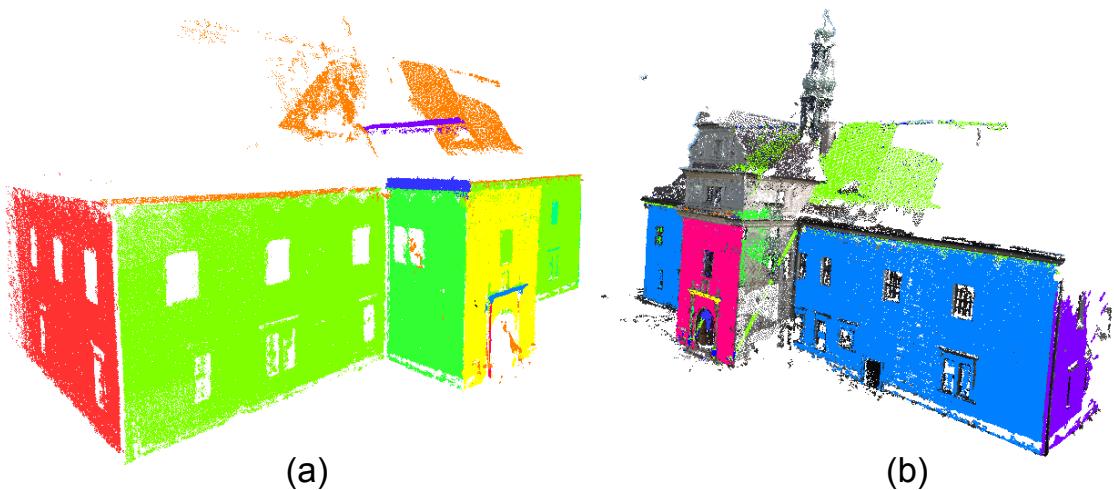
test	Parametry			Body		Segmenty			...		
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných			
do-0.0	80	20	10	1299900	1032386	79,42%		11	16	11	...

...	Čas [s]				
...	Celkem	Segmentace	Klastrování	Kd-strom	SVD
...	660,43	542,61	117,82	149,93	174,69

Tabulka 4.9: Tabulka zobrazuje informace o segmentaci mračna bodů Doudleby. Parametry jsou popsány v příloze A.

Segmentace mračna sice segmentovala téměř 80% bodů, nicméně nenalezla 5 rovin. Nebyla nalezena levá část střechy, dále chybí celá pravá část vstupní části budovy. Dále nebyly nalezeny 3 menší segmenty zepředu a na boku vstupní části budovy. Všechny tyto segmenty mají společnou jednu vlastnost, a to je velmi malá hustota bodů oproti ostatním stěnám. Domnívám se, že kvůli této malé hustotě bodů není výpočet parametrů tak přesný, navíc



Obrázek 4.16: Ukázka segmentace mračna Doudleby. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány.

celkově obsahují tyto nenalezené roviny malý počet bodů. K tomu se ještě přidává fakt, že celkově toto mračno obsahuje téměř 1,3 milionu bodů, takže vytvořené akumulátorové pole je poměrně velké. Myslím si, že spojení těchto tří vlivů dohromady způsobilo nenalezení těchto menších segmentů. Myslím si ale, že je to spíše problém celé této metody segmentace, než konkrétní implementace. Segmentace je založena zejména na tom, že roviny vytvoří vrchol v akumulátorovém poli. Pokud však několik různých vlivů způsobí, že body v rovině nevytvorí výrazný vrchol, není téměř možné toto detektovat a tím pádem ani řešit.

Aby byla nalezena aspoň ta největší střecha, bylo nutné použít větší parametr pro velikost okolí (80), což v kombinaci s téměř 1,3 miliony body způsobilo, že segmentace trvala 11 minut. To je výrazně více, než u všech předešlých testů. Celkově však segmentace není úplně špatná. U rovin, které byly nalezeny, byly segmentovány téměř všechny možné body a celkově bylo segmentováno téměř 80% ze všech vstupních bodů.

#### 4.2.3.5 Faustův dům

Vstupní mračno zachycuje část Faustova domu na Karlově náměstí. Mračno je zobrazeno na obrázku 4.17. Toto mračno je největší, které jsem měl při testování k dispozici, obsahuje necelých 1,5 milionu bodů.



Obrázek 4.17: Ukázka mračna Faust.

Výsledky testování jsou uvedeny v tabulce 4.10 a ukázka segmentace je zobrazena na obrázku 4.18.

test	Parametry			Body		Segmenty			...		
	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných			
fa-0.0	35	10	10	1492940	965492	64,67%		6	8	5	...

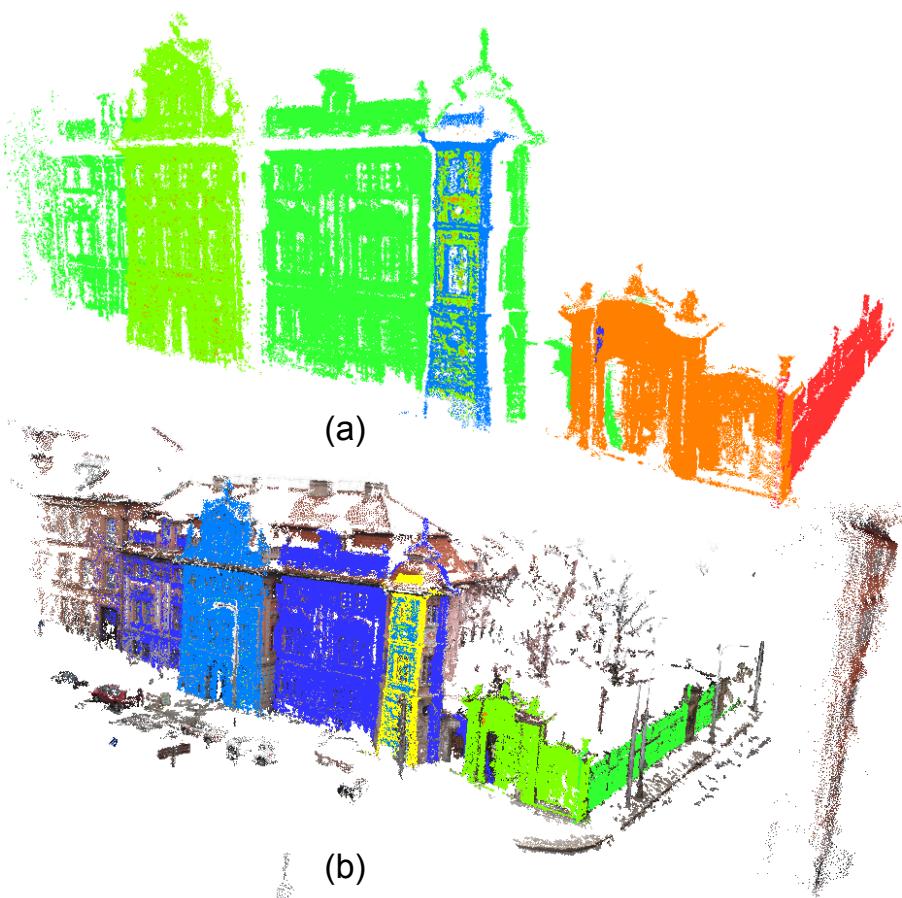
  

...	Čas [s]				
	Celkem	Segmentace	Klastrování	Kd-strom	SVD
	1033,98	336,76	697,22	93,58	105,49

Tabulka 4.10: Tabulka zobrazuje informace o segmentaci mračna bodů Faustův dům. Parametry jsou popsány v příloze A.

Na této segmentaci se projevila chyba, kdy z jedné stěny nebyly vybrány všechny body a při další iteraci byl v té samé stěně nalezen nový segment. Tento problém byl již dříve v této kapitole popsán. Tomuto problému jsem se snažil vyhnout tím, že jsem nastavil kvalitu klastrování na 10. Doufal jsem, že při první iteraci bude vybrána většina bodů z roviny a v další iteraci již nezbydou body, které by byly rozumným segmentem. Bohužel ani toto se neukázalo jako řešení, vedlejším důsledkem byl navíc velký čas klastrování bodů - více jak 11 minut, celkový čas segmentace přesáhl 17 minut a byl nejdélší ze všech testů.

Dále nebyly nalezeny menší roviny spojující největší stěny domu. Tyto roviny jsou ale velmi malé a zároveň rohy domu jsou při bližším pohledu výrazně zakulacené, takže pouze



Obrázek 4.18: Ukázka segmentace mračna Faustův dům. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány.

malá část z této roviny je doopravdy rovná. Bohužel je tato část příliš malá, aby jí algoritmus v tak velkém počtu bodů našel. Dále by se dalo segmentaci vytknout, že v levé části budovy mohlo být do segmentu přiřazeno více bodů.

Celkově bych tuto segmentaci označil za nejhorší ze všech testovaných. Všechny největší segmenty segmenty byly nalezeny, ale určitě je zde prostor pro zkvalitnění segmentace.

#### 4.2.3.6 Langweil

Toto mračno je sken části Langweilova modelu Prahy, což je papírový model pražského starého města z devatenáctého století. Mračno je velmi malé, obsahuje pouze 125 tisíc bodů, mračno je zobrazeno na obrázku 4.19



Obrázek 4.19: Ukázka mračna Langweil.

Výsledky testování jsou uvedeny v tabulce 4.11 a ukázka segmentace je zobrazena na obrázku 4.20.

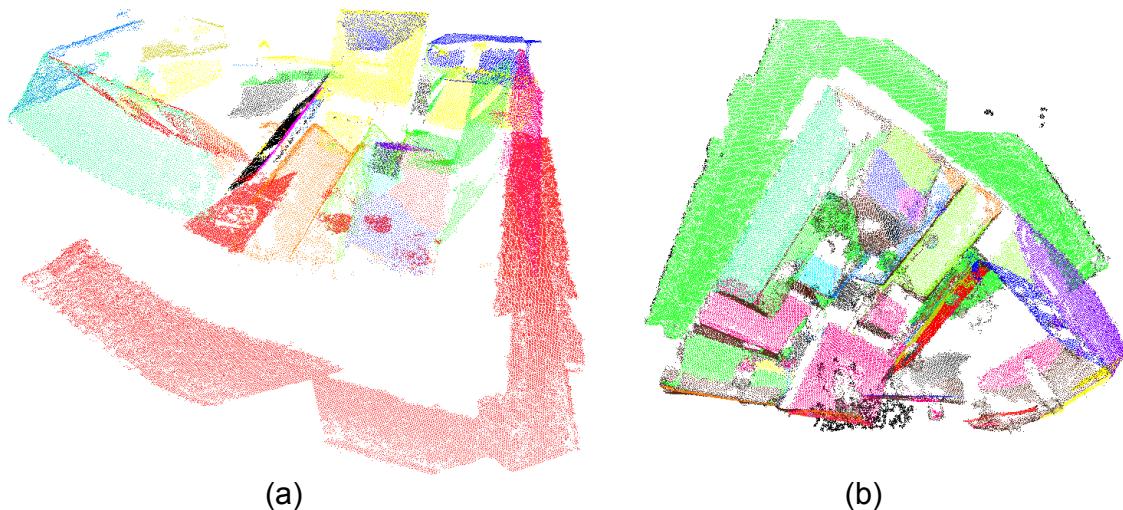
	Parametry			Body		Segmenty			...	
test	P1	P2	P3	Celkem	Segmentováno	Nalezeno	Očekáváno	Správných	...	
lw-0.0	10	15	3	125909	93816	74,51%	25	cca. 28	?	...

...	Čas [s]				
...	Celkem	Segmentace	Klastrování	Kd-strom	SVD
...	26,83	10,69	16,14	2,01	3,69

Tabulka 4.11: Tabulka zobrazuje informace o segmentaci mračna bodů Langewil. Parametry jsou popsány v příloze A.

Segmentace mračna není určitě tak kvalitní, jak by se dalo očekávat. Problém v tomto případě však není vyloženě v algoritmu segmentace, ale ve vstupních datech. Protože se jedná o sken starého papírového modelu, většina střech není rovná, naopak střechy jsou prohnuté či různě lomené. Segmentace tedy některé střechy nenalezla, protože nejsou rovné, u jiných



Obrázek 4.20: Ukázka segmentace mračna Langweil. (a) zobrazuje pouze nalezené segmenty, (b) zobrazuje segmenty včetně bodů, které nebyly segmentovány.

našla např. dva segmenty na jedné střeše, protože střecha je různě zprohýbaná. Výsledky segmentace tedy nejdou příliš dobře zhodnotit, vstupní mračno není vhodné pro testování planární segmentace.

Jako pokus jsem zvolil malé okolí bodu, hodnotu jsem nastavil na 10. Tím jsem docílil toho, že výpočet parametrů nebyl příliš přesný, identifikované segmenty měly tedy větší RMS a obvykle obsahly více bodů ze svého okolí. Tím jsem chtěl dosáhnout toho, že když střechy nejsou příliš rovné, horší přesnost vypočtených parametrů by mohla částečně tento problém „zakrýt“. Tato myšlenka se ve skutečnosti ukázala správná a opravdu bylo nalezeno méně segmentů, které více odpovídaly střechám. Pokud jsem při testování používal větší velikost okolí, algoritmus nacházel více segmentů, který byly sice relativně přesnější, ale méně odpovídaly „reálným“ rovinám.

Celkově tedy segmentace příliš neodpovídá tomu, co bychom na první pohled očekávali, nicméně minimálně spodní velká rovina byla nalezena, stejně tak několik rovnějších střech bylo identifikováno správně. Algoritmus by tedy neměl mít problém segmentovat menší mračna s menší hustotou bodů.

## 4.3 Shrnutí výsledků testování

V první části testování jsme ověřovali funkčnost algoritmu na uměle generovaných datech. Ve všech těchto testech byly nalezeny všechny roviny a bylo segmentováno více jak 99% bodů, algoritmus tedy na těchto ideálních datech funguje správně.

Testování segmentace na reálných datech už nebylo tak bezproblémové, jako u generovaných dat. Celkově však algoritmus vždy našel většinu rovin a do segmentu přiřadil většinu bodů z dané roviny. Nicméně ne všechny roviny byly vždy nalezeny. Pokud byla nějaká rovina při segmentaci vynechána, obvykle byla tato rovina výrazně menší, než ostatní nalezené roviny, resp. obsahovala výrazně méně bodů. Problematická se tedy ukázala i situace, kdy měly roviny výrazněji odlišnou hustotu bodů. Roviny s malou hustotou bodů (tedy i malým počtem bodů) často v akumulátorovém poli nevytvářily dostatečně výrazný vrchol a nebyly nalezeny. Dále také nemusejí být nalezené roviny, které na malém měřítku nejsou příliš rovné, naopak jsou různě zvlněné nebo hrbolaté. Do určité míry se dá toto kompenzovat správným nastavením parametrů, někdy ale ani to nestačí.

Dále jsme při testování také narazili na dva problémy, které se netýkali přímo nalezení segmentů, ale jejich kvality či správnosti. Prvním problémem byla situace, kdy v akumulátorovém poli dvě roviny vytvořily vrcholy těsně vedle sebe a při vyhledávání vrcholů byl nalezen jen jeden. V této situaci mohly být do segmentu přiřazeny body i z druhého vrcholu, tedy z jiné roviny. Taková situace při reálném testování je zobrazena na obrázku 4.16 (b), kde zelený segment střechy obsahuje i body z boční stěny. Druhým problémem byla situace, kdy v nejvyšší buňce z vrcholu v akumulátorovém poli byly velmi přesné body. Segment vytvořený z tohoto vrcholu obsahoval pouze body, které ležely velmi přesně v rovině. V akumulátorovém poli poté zbyly body, které stále ještě ležely v rovině, která akorát nebyla tak přesná. Pokud bylo takových bodů dostatečné množství, při další iteraci prohledávání akumulátorového pole mohly být identifikovány jako segment. Výsledkem tedy byly dva segmenty v jedné rovině. Příklad této situace je zobrazen na obrázku 4.18.

V celkovém součtu všech testů<sup>3</sup> mračna obsahovala 85 rovin a algoritmus správně detekoval 68 rovin, což je přesně 80%. Algoritmus tedy není bezchybný, ale je schopen správně detektovat většinu rovin. Navíc nenalezené roviny jsou vždy menší či méně přesné, nestalo se, aby algoritmus nenašel nějakou výraznou rovinu.

Co se týče paměťových a časových nároků, ty jsou logicky nejvíce závislé zejména na velikosti vstupních dat. Doba běhu algoritmu je navíc závislá na zadaných parametrech, které mohou segmentaci zkvalitnit, vždy ale na úkor delšího běhu algoritmu.

---

<sup>3</sup>Není zahrnut poslední test mračna Langweil, které nebylo pro testování příliš vhodné.



# Kapitola 5

## Závěr

Cílem této práce bylo nastudovat, implementovat a otestovat dva zadané algoritmy. V průběhu práce na prvním algoritmu jsem bohužel zjistil, že pochopení a následná implementace algoritmu bude náročnější, než jsem předem očekával. To bylo způsobené částečně tím, že tato práce byla moje první seznámení s 3D rekonstrukcí mračen bodů, obecně byla i mojí první větší zkušenost se zpracováním takto velkých vstupních dat. Z těchto důvodů jsem strávil příliš mnoho času při studování základních principů a technik. V neposlední řadě jsem se dlouhou dobu učil debugovat takovýto druh algoritmu, což není jednoduché, protože veškerá zpracovaná data mají pouze matematické vyjádření (body, roviny), které se nedá jednoduše představit a vzhledem k množství vstupních dat se často nedá ani jednoduše vykreslit. Z těchto důvodů jsem se s vedoucím práce dohodl, že zpracuji pouze první (a také složitější) algoritmus, oba algoritmy bych bohužel zpracovat nestihl. Nicméně nemyslím si, že by celkový rozsah práce byl nedostatečný. Tím, že jsem implementoval pouze jeden algoritmus, jsem se na něj mohl více zaměřit a také jsem mohl více času věnovat testování, aby bzla segmentace reálných dat co nejkvalitnější.

Co se týče zpracovaného algoritmu, i když byl teoreticky popsán velmi podrobně, jeho implementace vyžadovala mnoho vlastních úprav a myšlenek. Předně bylo důležité zvolit vhodné datové struktury pro reprezentaci mračna bodů a pro správnou reprezentaci akumulátorového pole. Dále bylo nutné vyhledat knihovny pro reprezentaci kd-stromu a SVD dekompozici s důrazem na co nejvyšší výkon. Než jsem nalezl knihovny, které jsem ve výsledku použil, testoval jsem desítky různých více či méně kvalitních řešení. Při testování těchto knihoven mně osobně překvapilo, že i když mají dané metody ve všech knihovnách stejnou asymptotickou složitost, výsledný reálný výkon těchto knihoven byl velmi závislý na zvolené implementaci a mezi jednotlivými knihovnami se výrazně (někdy až řádově) lišil.

Dále jsem se při implementaci algoritmu musel vypořádat se skutečností, že autoři původního článku používali pro testování odlišný typ vstupních dat s definovaným šumem (odchylkou bodů v daných směrech), na jehož základě odvozovali mnohé rozdíly a hodnoty. Já jsem takovýto šum v datech neměl definovaný, proto jsem musel vymyslet vlastní definice pro výpočet velikosti okolí, velikosti bufferu při definici okolí, velikosti akumulátorového pole apod. Dále jsem upravil metodu pro vybírání bodů z akumulátorového pole, kdy jsem místo vybírání vrcholů po jednom navrhl vybírání všech vrcholů postupně, čímž jsem předešel nežádoucím chybám v situaci, kdy byly dva vrcholy v akumulátorovém poli v těsné blízkosti.

V neposlední řadě jsem implementaci rozdělil na dvě nezávislé části, které je možné provádět odděleně, tím je dána uživateli možnost provádět jednu část algoritmu opakovaně bez nutnosti provádět celý algoritmus, čímž lze výrazně ušetřit výsledný čas strávený segmentací. Nakonec jsem také definoval tři parametry, které uživateli umožní přizpůsobit kvalitu a rychlosť segmentace na konkrétní zpracovávaná data. Toto je sice na úkor automatizace celého algoritmu, nicméně nedovedu si představit, že by takovýto algoritmus fungoval plně automaticky na naprosto různých vstupních datech.

Algoritmus ale samozřejmě není dokonalý, testování ukázalo několik problematických situací, kdy segmenty nebyly nalezeny nebo naopak byly nalezeny dva segmenty v rámci jedné roviny. Zde je rozhodně prostor pro případné rozšíření práce nebo navázání na práci. Je zde prostor pro definování lepsí metody určení velikosti akumulátorového pole, určitě by také bylo možné vylepšit metodu vyhledávání vrcholů v akumulátorovém poli a jejich následné vybírání. Vylepšení této části algoritmu by se mohlo výrazně projevit na zlepšené kvalitě segmentace, která by byla schopna detektovat větší množství segmentů. Je však nutné vzít v úvahu, že zde jsou omezení dané samotným principem hlasování do akumulátorového pole. Nedá se očekávat, že by byl algoritmus schopný nalézt roviny o desítkách až stovkách bodů v mračnech, které obsahují statisíce až miliony bodů s nezanedbatelným množstvím šumu a bodů, které v žádných rovinách neleží. Dále se také nabízí možnost možnost zjednodušit či vylepšit systém zadávaných parametrů, případně nějaký parametr efektivně odebrat. Každý takovýto povinný uživatelský parametr výrazně snižuje použitelnost algoritmu, uživatel je nyní nucen pochopit alespoň základní princip algoritmu, aby byl schopen parametry vhodně zvolit.

Pokud bych měl celkově zhodnotit kvalitu algoritmu, tak si myslím, že algoritmus je poměrně efektivní a je schopen ve velmi různorodých mračnech bodů nalézt většinu požadovaných rovin v přijatelné kvalitě, ale rozhodně zde je prostor pro vylepšení.

# Literatura

- [1] ABELES, P. Java Matrix Benchmark, .  
<http://code.google.com/p/java-matrix-benchmark/>, stav z 1.5.2012.
- [2] ABELES, P. Efficient Java Matrix Library (EJML), .  
<http://code.google.com/p/efficient-java-matrix-library/>, stav z 1.5.2012.
- [3] KIM, C. et al. Segmentation of Laser Scanning Data using Approach based on Magnitude of Normal Vector. *ISPRS Journal of Photogrammetry and Remote Sensing*. 2012.
- [4] Přispěvatelé Wikipedie. *k-d tree* [online]. 2012. [cit. 28. 4. 2012].  
[http://en.wikipedia.org/wiki/K-d\\_tree](http://en.wikipedia.org/wiki/K-d_tree).
- [5] Přispěvatelé Wikipedie. *Least squares* [online]. 2012. [cit. 1. 5. 2012].  
[http://en.wikipedia.org/wiki/Least\\_squares](http://en.wikipedia.org/wiki/Least_squares).
- [6] REISNER-KOLLMANN, I. – LUKSCH, C. – SCHWÄRZLER, M. Reconstructing Buildings as Textured Low Poly Meshes from Point Clouds and Images. *EUROGRAPHICS 2011 - Short Papers*. 2011.
- [7] SEDLÁČEK, D. ArchiRec3D.  
<http://dcgi.felk.cvut.cz/cs/members/sedlad1/>, stav z.
- [8] VOSSELMAN, G. et al. Recognising structure in laser scanner point clouds. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*. 2004.
- [9] web:kdtree. Knihovna s efektivním kD-stromem.  
<http://robowiki.net/wiki/User:Rednaxela/kD-Tree>, stav z 1. 6. 2010, 08:37.



## Příloha A

### Seznam použitých zkratek

**RMS** Root Mean Square, česky efektivní hodnota. Určuje velikost měnící se veličiny.

**P1** Parametr, který určuje velikost okolí bodu.

**P2** Parametr klastrování, který určuje práh vyhledávání vrcholů v akumulátorovém poli.

**P3** Parametr, který určuje kvalitu klastrování.



## Příloha B

# Obsah přiloženého CD

```
├── README - soubor se základními informacemi
├── Javadoc - složka s vygenerovaným javadocem
│   ├── index.html - tímto souborem se otevírá javadoc
│   ├── ...
│   └── ... - jsou zde další soubory, jejich výpis je nepodstatný
│       └── vicitis - zdrojové soubory pro javadoc
├── Tabulky - složka obsahuje všechny tabulky uvedené v textu ve formátu ods (Open Office)
├── Testování - všechna data, co se týče testování, včetně vstupních mračen
│   ├── dodaná mračna bodů - zdrojové soubory dodané vedoucím práce
│   │   ├── červená_lhota - všechny složky obsahují několik souborů, jejich výpis je zbytečný
│   │   ├── doudleby
│   │   ├── dum2
│   │   ├── faust
│   │   ├── hurka_hriste
│   │   └── langweil
│   ├── readme - informace k formátu testovacích souborů
│   ├── testování funkčnosti, kap. 4.1.1 - data použitá pro testování funkčnosti algoritmu
│   │   ├── readme - informace k formátu dat
│   │   ├── 1-0-gen.wrl - jednotlivé soubory s testy, kompletní výpis vynechán
│   │   ├── ...
│   ├── testování kvality, kap 4.2 - všechna reálná testovací data
│   │   ├── readme - informace k formátu a názvům souborů
│   │   ├── 4.2.1.1 - data z dané kapitoly, ukazují vliv parametrů na segmentaci
│   │   │   ├── d2-0.00-seg-all.wrl
│   │   │   ├── ...
│   │   ├── 4.2.3.2 - data z dané kapitoly, nejlepší výsledky testování této mračen
│   │   │   ├── cervena-lhota - každá složka obsahuje několik souborů, výpis by byl zbytečný
│   │   │   ├── doudleby
│   │   │   ├── dum2
│   │   │   ├── faust
│   │   │   ├── langweil
│   │   │   └── stodulky
│   ├── Text - text práce
│   │   ├── BP-princdn-2012.pdf - text práce
│   │   ├── latex - zdrojové soubory k textu práce
│   │   │   ├── bakalarka.tex - zdrojový TeX soubor
│   │   │   ├── ...
│   │   │   └── figures - všechny obrázky použité v práci
└── Zdrojové kódy - všechny zdrojové kódy
    ├── ArchiRec3D - tato složka obsahuje kompletní zdrojové kódy celého systému
    └── Princdn - tato složka obsahuje mnoho vytvořené zdrojové kódy
        └── vicitis
            ├── inner
            │   └── methods
            │       ├── LeastSquare.java - implementace metody nejmenších čtverců
            │       └── WeightedLeastSquare.java - implementace vážené metody nejmenších čtverců
            └── segmentation
                └── pointcloudSegmentation
                    └── PlaneGenerator.java - náhodný generátor mračen bodů obsahujících roviny
            └── tools
                └── PlanarSegmentation.java - třída s implementací planární segmentace
```