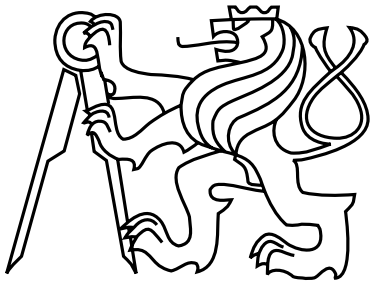
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Planární segmentace mračna bodů

Daniel Princ

Vedoucí práce: Ing. David Sedláček

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimedia

11. května 2012

Poděkování

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

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 25. 5. 2015 .....

Abstract

Translation of Czech abstract into English.

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 dva algoritmy, které budou segmentaci provádět co nejvíce automaticky a zároveň v dostatečné kvalitě. Algoritmy jsou implementovány jako součást 3D rekonstrukčního nástroje ArchiRec3D.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

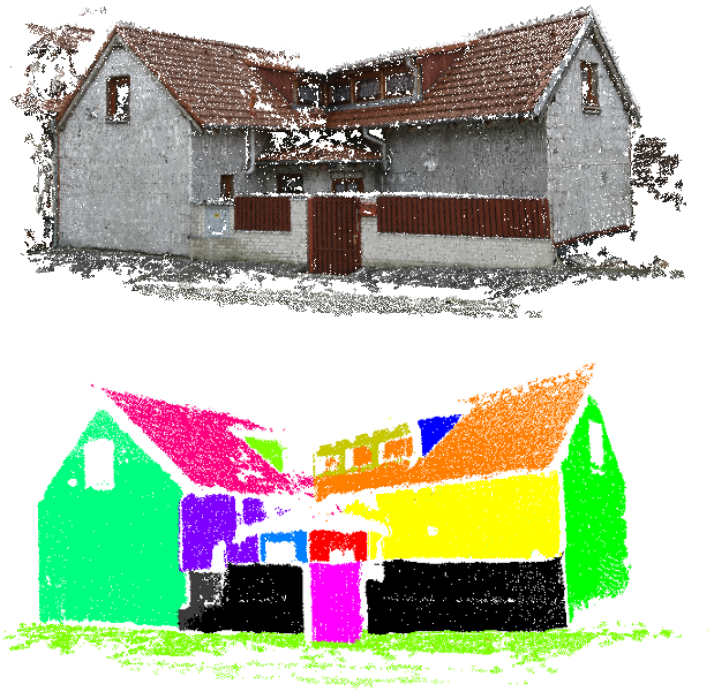
Rekonstrukce 3D modelů se dnes uplatňuje v mnoha oborech. Běžně se můžeme setkat např. s vizualizacemi, které se často využívají pro simulaci vzhledu města při výstavbě nových objektů. V takovém případě se obvykle naskenuje aktuální stav okolí výstavby, do kterého se přidá model plánované budovy. Výsledná 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 nekompletní 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[1] 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.

[1] 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

Tato práce nejprve v kapitole 2 popisuje zadané algoritmy [3] a [6] a věnuje se návrhu obou implementací. Poté následuje kapitola 3 popisující samotnou implementaci algoritmů. Algoritmy jsou implementovány v rámci dodaného nástroje ArchiRec3D [7], který je psaný v jazyku Java. V závěrečné části práce v kapitole 4 je otestována správná funkčnost obou algoritmů na uměle generovaných datech a na reálných datech je otestována kvalita algoritmů a jejich implementací. V závěru jsou shrnuty výsledky testování a analyzovány nedostatky algoritmů. 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 na stejné ploše, které zohledňuje jak podobnost v prostoru parametrů, tak vzdálenost bodů v prostoru.

2.1.1 Adaptivní válcová definice okolí

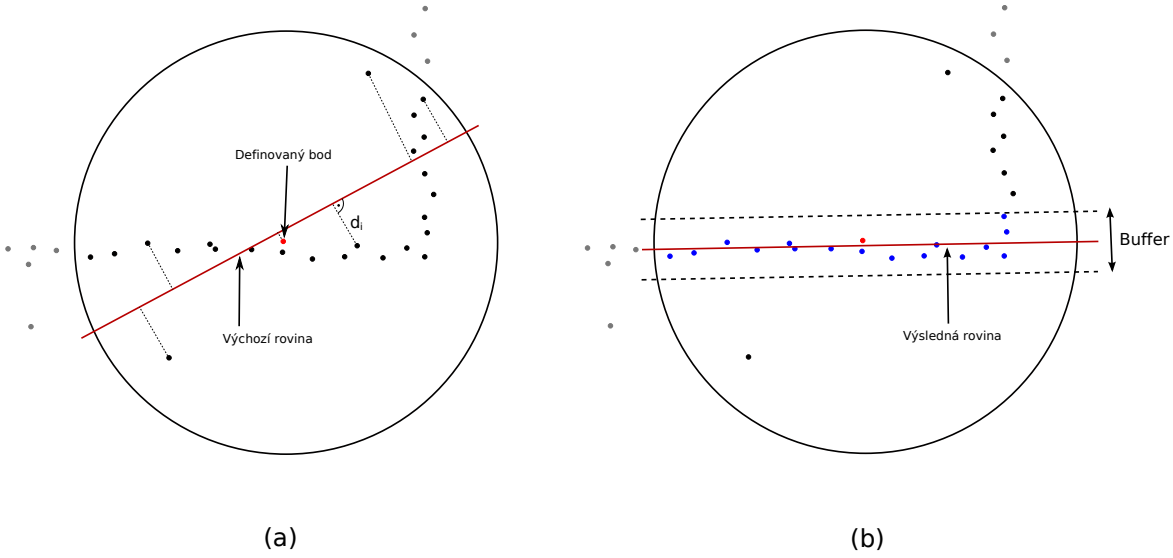
Správná definice okolí bodu je zásadní podmínkou pro správnou funkčnost algoritmu, protože okolí bodu přímo ovlivňuje parametry, které jsou pro daný bod vypočteny. V tomto algoritmu je okolí definováno tak, že bere v úvahu vzdálenost bodů v prostoru, ale zároveň také tvar povrchu, na kterém se bod nachází. To znamená, že do okolí jsou zahrnuty pouze takové body, které jsou prostorově blízko, ale zároveň leží na stejné ploše.

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[1]. 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[2], jak můžeme vidět v rovnici 2.1:

[1] Nicméně praxe ukazuje, že vhodná volba jsou řádově desítky až stovky bodů uvnitř koule.
[2] Jednoduše řečeno, čím je bod od plochy dál, tím je jeho váha nižší.

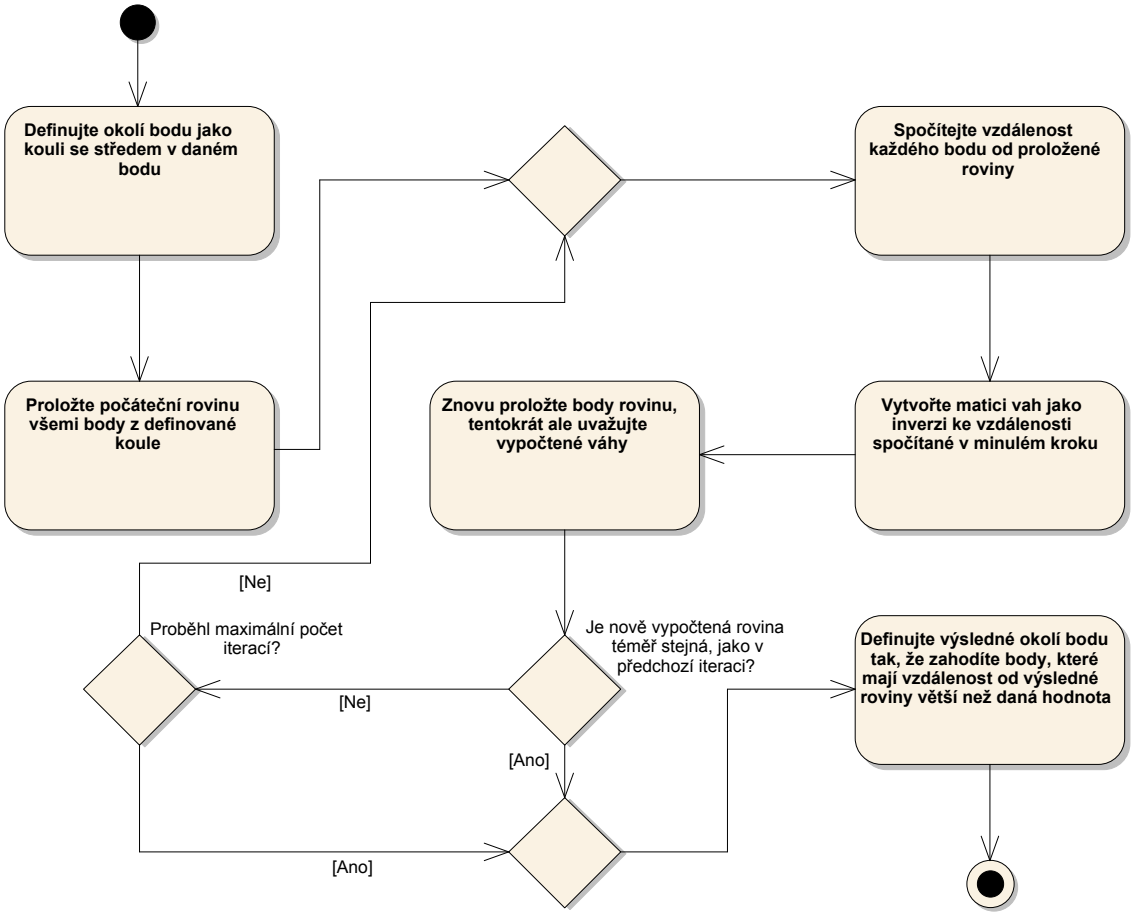
$$p_i = \frac{1}{d_i} \tag{2.1}$$

kde p_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 znovu 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 výsledné okolí (viz Obr. 2.1(b)). Ostatní body mimo buffer se již na okolí nijak nepodílejí.



Obrázek 2.1: Definice okolí bodu (boční pohled).
(a) zobrazuje proložení původní roviny body uvnitř koule. Pro některé body je znázorněna jejich váha d_i.
(b) znázorňuje výslednou rovinu po iterativním procesu, který zahrnuje váhy jednotlivých bodů. Výsledné okolí tvoří modře vyznačené body uvnitř bufferu.

Výsledné okolí se tedy nachází v útvaru, který téměř odpovídá nízkému válci, jehož osa je normála k ploše, na které se původní bod nachází. Tato osa navíc může být v průběhu algoritmu oproti výchozí hodnotě značně upravena, proto je tato metoda označena jako adaptivní válcová definice okolí.



Obrázek 2.2: Diagram aktivit znázorňující proces definice okolí bodu.

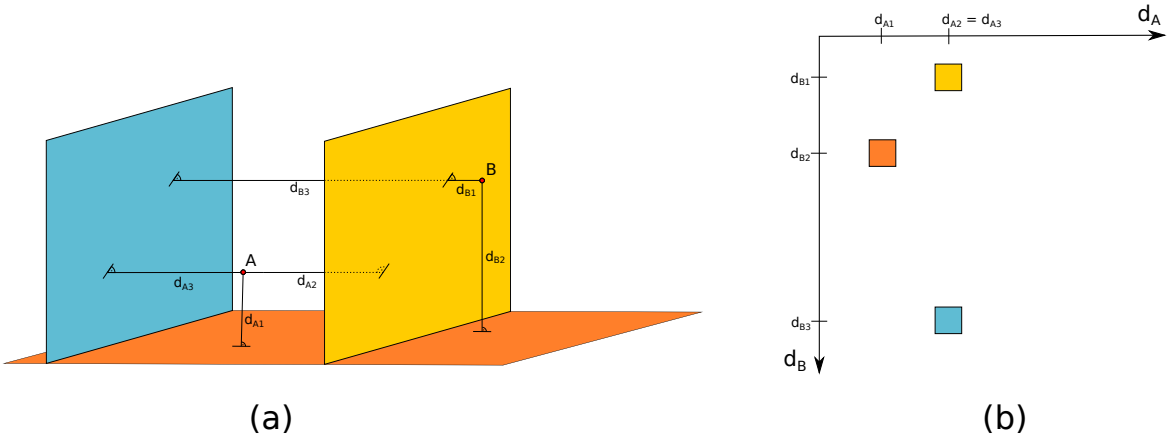
2.1.2 Výpočet parametrů bodu

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. Rozměry akumulátorového pole závisí na počtu použitých parametrů. Můžeme tedy použít 3 složky normálového vektoru jako atributy a tím zajistíme, že všechny segmenty budou správně rozeznány[3]. Nicméně takovýto způsob vyžaduje vytvoření třírozměrného akumulátorového pole a hlasování do 3D pole je výpočetně náročná operace.

Pro snížení výpočetních nároků je tedy nutné snížit počet parametrů. Proto je jako atribut využita velikost normálového vektoru[4]. Toto řešení přináší ale nebezpečí v tom, že

[3]Mělo by být zjevné, že jeden normálový vektor jednoznačně definuje rovinu.
[4]Tedy jde o vzdálenost roviny od referenčního bodu.

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 zaznamenány na základě těchto dvou vypočtených parametrů. Body, které patří do různých rovin, jsou tedy v akumulátorovém poli umístěné 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:

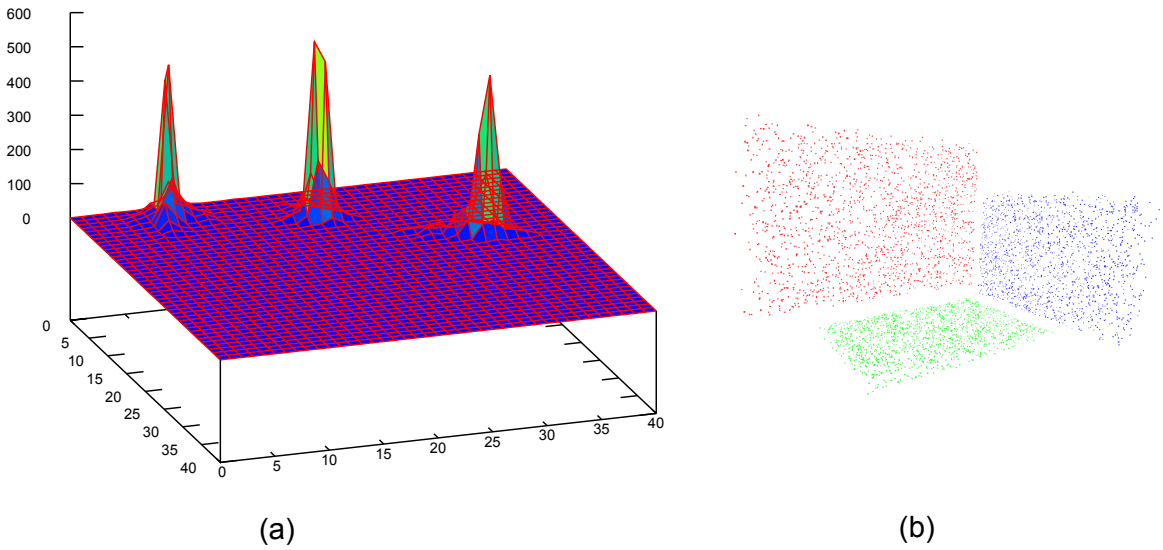
$$\begin{aligned} \text{referenční bod A} &= \begin{pmatrix} minX \\ minY \\ minZ \end{pmatrix} + \frac{1}{3} \begin{pmatrix} maxX - minX \\ maxY - minY \\ maxZ - minZ \end{pmatrix} \\ \text{referenční bod B} &= \begin{pmatrix} minX \\ minY \\ minZ \end{pmatrix} + \frac{2}{3} \begin{pmatrix} maxX - minX \\ maxY - minY \\ maxZ - minZ \end{pmatrix} \end{aligned} \tag{2.2}$$

Umístěním referenčních bodů do těchto pozic značně snižujeme možnost, že se více rovin zobrazí na stejné místo 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 splynout v jeden vrchol, který by poté byl reprezentován jako jeden segment. 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 representuje mračno bodů se třemi kolmými rovinami (b). Svislá osa grafu reprezentuje počet bodů, vodorovné osy parametry bodů (tedy souřadnice v poli).

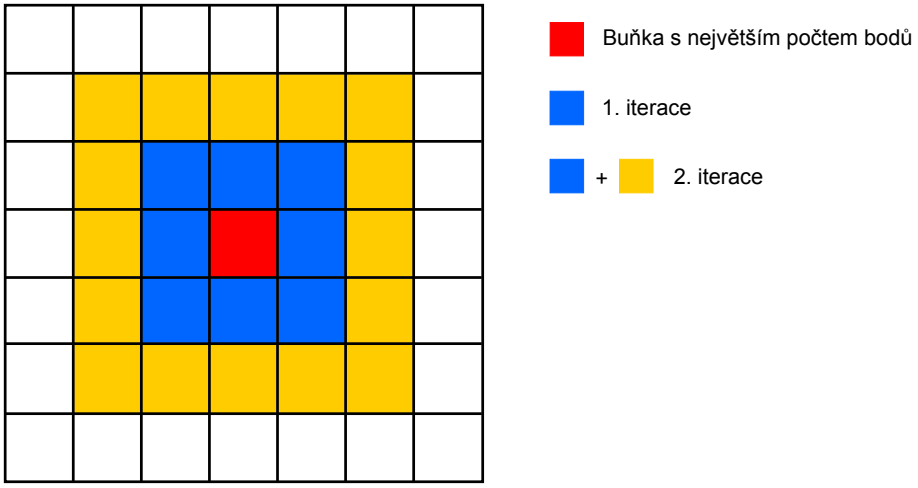
Pokud máme všechny body uloženy v akumulátorovém poli, vyhledáme buňku s největším počtem bodů (tedy nejvyšší vrchol). Velikost buňky v akumulátorovém poli je určena na

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 $\{x_1, x_2, \ldots, x_n\}$ a jejich vzdáleností od proložené roviny $\{d_1, d_2, \ldots, d_n\}$, vypočítáme RMS podle vztahu 2.3:

$$rms = \sqrt{\frac{1}{n}\left({d_1}^2 + {d_2}^2 + \ldots + {d_n}^2\right)} \tag{2.3}$$

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 segmentu. 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 clusteru. Pouze body, které jsou k rovině blíže, než je předem daný práh, mohou být přidány do klastru. Tento práh je opět určit v závislosti na velikosti šumu v datech. Body, které nesplnily jednou z těchto dvou podmínek zůstanou na své pozici v akumulátorovém poli a jsou znovu 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 první iteraci. 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 klastr 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 opakují 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 větší, 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:

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

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 základně 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 popsané.

Tímto jsme popsali celý algoritmus. Jak by mělo být patrné, algoritmus bere v úvahu vztahy bodů v prostoru parametrů a zároveň jejich vzdálenost 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 Segmentace pomocí vyhledávání dominantních os

[6]

2.3 Proložení roviny množinou bodů

Důležitou součástí prvního 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 $(p_1, \ldots, p_k)$. Rovina je definována bodem c, který leží v rovině, a normálový vektorem n. 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:

$$(n(p_i - c)^T)^2 \tag{2.5}$$

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

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

Řešení tohoto vztahu pro c je:

$$c = \frac{1}{k} \sum_{i=1}^{k} p_i \tag{2.7}$$

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-tém řádku hodnoty $p_i - c$ (jde tedy o matici $k \times 3$), můžeme vztah 2.6 přepsat do této formy:

$$\min_{\|n\|=1} \|Mn\|^2 \tag{2.8}$$

Řešením tohoto problému je SVD dekompozice matice $M = USV^T$, 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 normálový vektor n, můžeme snadno vyjádřit rovinou např. obecnou rovnicí:

$$ax + by + cz + d = 0 \tag{2.9}$$

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

$$d = -(n_1c_1 + n_2c_2 + n_3c_3) \tag{2.10}$$

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:

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

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

Kapitola 3

Implementace

Tato kapitola se zabývá popisem implementace obou algoritmů. Algoritmy jsou implementovány v jazyce Java v rámci dodaného softwaru ArchiRec3D [7]. Kompletní zdrojový kód je k dispozici na přiloženém CD, viz kapitola C.

3.1 Segmentace na základě velikosti normálového vektoru

Implementaci tohoto 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 sementů. 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 znovu provádět první část.

3.1.1 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 implementuji v rámci dodaného nástroje, mám k dispozici několik základních datových struktur jako např `vicitis.inner.Point3D` pro reprezentaci 3D bodu, `vicitis.inner.Plane3D` pro reprezenatci roviny apod., tak samozřejmě tyto datové struktury 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 kladeny 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 jednotlivé uzly jsou k-dimenzionální body. Tento typ stromu dělí prostor na podprostory podél těchto 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 nejefektivnější. 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. Takto 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.

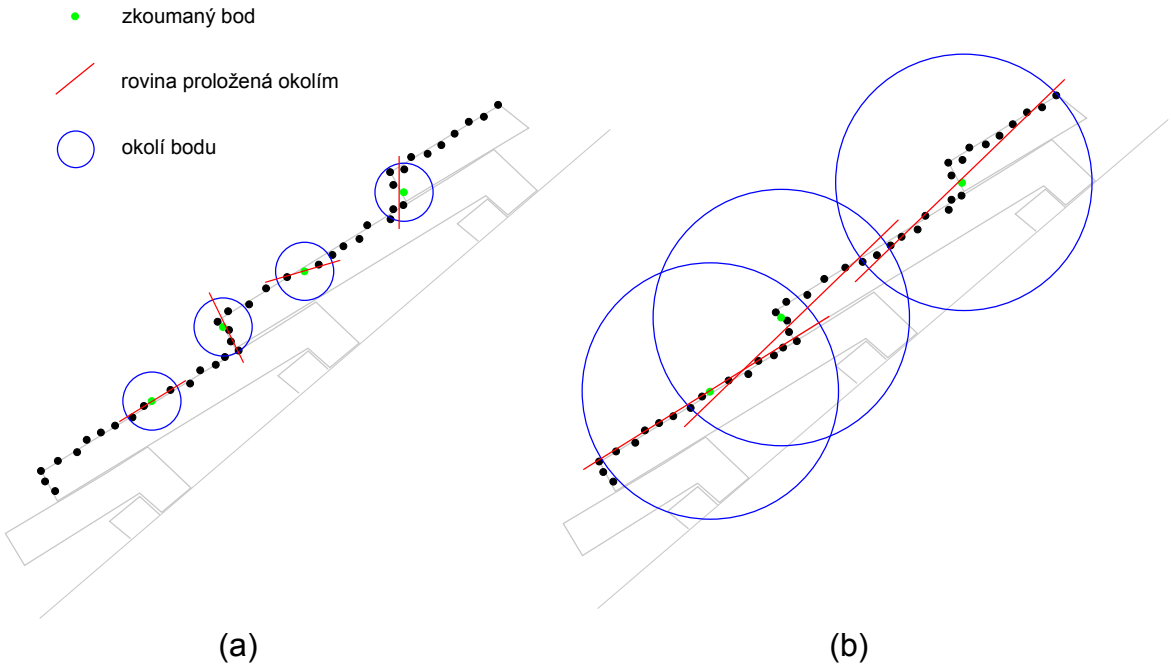
Velmi důležité jsou také rozměry akumulátorového pole. Podle teorie je políčko v akumulátorovém poli určeno na základě velikosti šumu ve vstupním mračnu. Bohužel testovací data, která mám k dispozici, nemají šum definovaný. Proto je nutné vymyslet vlastní strategii pro určení velikosti akumulátorového 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 zkusil 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 rozměrech 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 mohla být vhodná volba 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 stejné 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 rozeznatelné. 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 100×100, pro 1,5 mil bodů je to pole o rozměrech 1225×1225. 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 roviny apod. Nicméně takový algoritmus by nebyl vůbec jednoduchý a pravděpodobně by vyžadoval nějaké vstupní parametry zadané 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 algoritmu je tato velikost popisována 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 naprosto 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 řádu 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 - 50, 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 rychlostí 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 rozpoznat jako segment, musíme volit okolí větší. Na obrázku 3.1 je ukázka takové situace, kdy volba příliš malého okolí značně ovlivní výsledný výpočet.



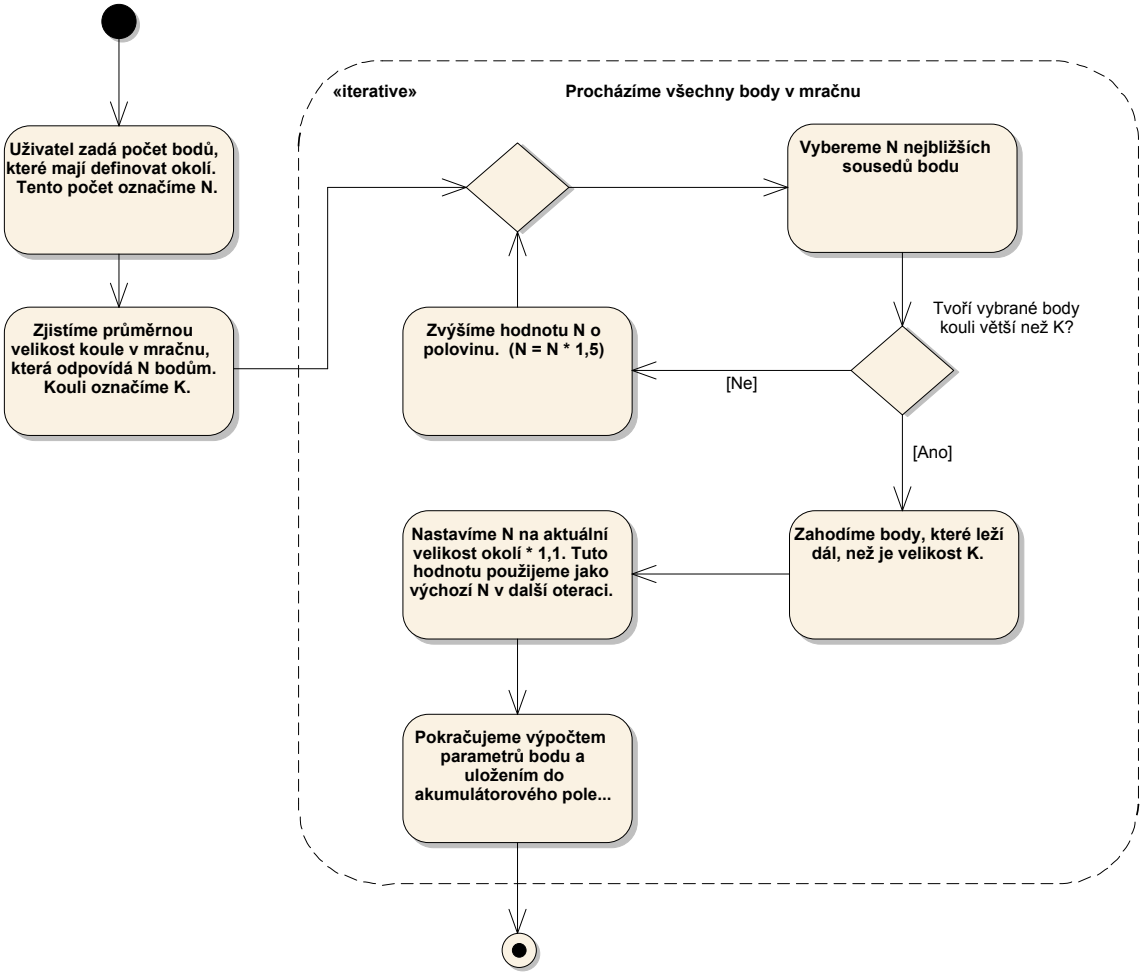
Obrázek 3.1: Ukázka vlivu velikosti okolí na vypočítávané parametry. Malé okolí může způsobit chybný výpočet (a), oproti tomu velké okolí je přesnější (b).

Hodnota, kterou uživatel zadá, se ještě přímo nevyužije k definici okolí. Tato hodnota 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 by mělo být v této kouli průměrně bodů. 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× 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 bodů budou mít pravděpodobně velmi podobné okolí, využijeme výslednou velikost okolí jako výchozí hodnotu pro bod v další iteraci. 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 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 vybírání sousedních bodů zabralo a také je zde uvedeno procento bodů, pro které bylo nutné vyhledávat v kd-stromu vícekrát[1]. V naprosté většině případů byly výsledky nejlepší při hodnotě parametru 1,10. Pro další bod při iterování bude tedy použit 1,1× násobek aktuální velikosti okolí.

	Data 1: 291232 bodů				*Data2: 81294 bodů*				*Data3: 1299900 bodů*			
Okolí	*50 bodů*		*200 bodů*		*50 bodů*		*200 bodů*		*50 bodů*		*200 bodů*	
Param.	*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
1,10	**17,89%**	**15,19**	**10,11%**	**53,28**	**24,67%**	**4,51**	**16,70%**	**17,38**	**17,36%**	**110,57**	**9,31%**	**457,39**
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 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 jednou. Prázdná políčka jsem již nevyplňoval, protože předešlé výsledky byly dostatečně průkazné a další měření by bylo zbytečné.

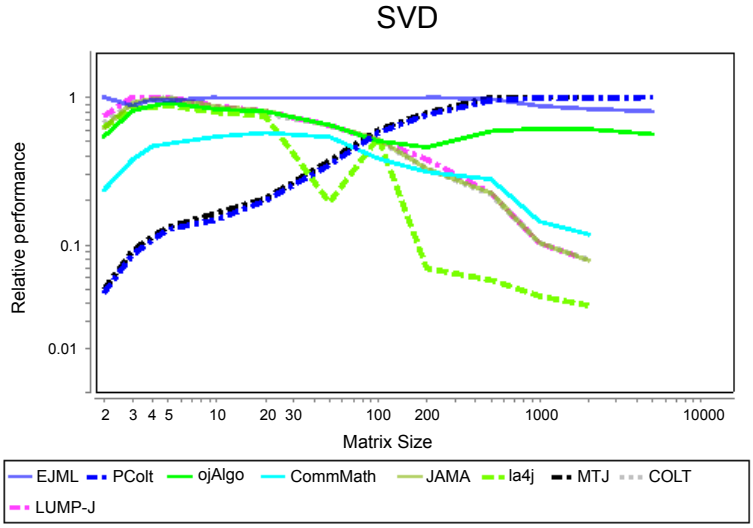
V tuto chvíli máme pro bod vybrány všechny sousední body, které leží uvnitř dané koule. Abychom získali konečné okolí bodu, musíme proložit body rovinou a poté provést převažování této roviny pomocí procesu popsaného v sekci 2.1.1. Dalším krokem v algoritmu je tedy pomocí metody nejmenších čtverců proložit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

Teorie k proložení roviny množinou bodů pomocí metody nejmenších čtverců je popsána v sekci 2.3. 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-tém řádku je po souřadnicích zapsán rozdíl hodnot $bod_i - centroid$. Nad touto maticí je poté 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

[1] 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].

test na reálných datech. Nedělal jsem žádný speciální test, pouze jsem měnil knihovny pro výpočet SVD v rámci již hotového algoritmu. Použil jsem mračno 389002 bodů, nastavil velikost okolí 100 bodů[2] 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 rychlost 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

[2]To odpovídá průměrně 100 bodům v okolí, tedy i matici 3x100 při výpočtu SVD.

jsou řádově stejné a dohromady tvoří naprostou většinu z celkové doby běhu. Stejně jako 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.

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 znovu 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 znovu 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 např. 2°, 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×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:

$$s = param \times \frac{velikost\ akumulatroveho\ pole}{velikost\ celeho\ mracna\ bodu} \tag{3.1}$$

kde s je souřadnice v akumulátorovém poli, *param* je vypočtená vzdálenost roviny od referenčního bodu a velikost celého mračna bodů odpovídá vzdálenosti bodů s největšími a nejmenšími souřadnicemi v mračnu bodů[3]. 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.

3.1.2 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 situaci, 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 ne jen 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).

[3] 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 podmínku (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 spojitou 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 zároveň ukládáme nejvyšší 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 naprosto 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 nejde 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 odhadnout, 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 nebyl schopný 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 řádově 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 3.1.3 a v kapitole 4.

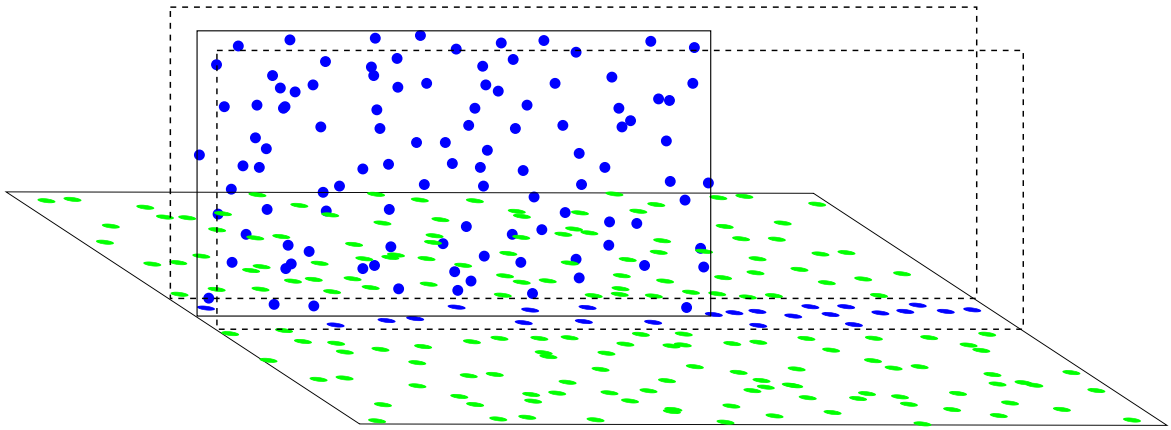
Máme tedy hotovou část, kdy jsme identifikovali vrcholy v poli a máme uložené souřadnice nejvyšších políček z těchto vrcholů. Nyní potřebujeme z každého vrcholu vybrat body, které patří do jednoho segmentu. Postup je takový, že vždy vybereme body z nejvyšší buňky, ty proložíme plochou a vypočítáme RMS. Poté procházíme body z okolních buněk (viz Obr. 2.5), pokud mají body vzdálenost od proložené roviny menší než 2×RMS (leží ve stejné rovině) a zároveň jsou blízko k ostatní 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 zjišťovalo, 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 podmínku je třeba při přidávání bodů do segmentu zachovat. Proto zkonstruujeme 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 3.1.3.

Data	**Parametry**	**Čas** [s]	
		Kd-strom	**Prům. Vzd.**
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 podmínku 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 detekovat a výrazně to nezhoršilo výkon algoritmu. Proto je nutné tento problém obejít a případně aspoň co nejvíce minimalizovat. Jedním ze způsobů, jak omezit výskyt takové situace, je nevybírat vrcholy z akumulátorového pole po jednom, ale vybírat všechny



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

vrcholy najednou. Protože vybírání bodů z vrcholu je iterativní proces, implementace postupného vybírání všech vrcholů najednou je poměrně jednoduchá. Implementace bude taková, že budeme iterativně procházet jednotlivé vrcholy. 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). Proces je tedy téměř stejný, jako byl původně navržený, jen nevybíráme jeden vrchol po druhém, ale iterativně je vybíráme všechny najednou.

Nyní máme navržený celý proces klastrování, tak podrobněji popíšeme implementaci. Vrcholy v poli jsme již nalezli 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 $5\times$RMS. Tyto body ale 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 seg-

mentu. 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 stejné 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 řešení.

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 stejné 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 přidalo a tento proces by se nemusel zastavit vůbec. Tuto podmínku 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 nenastavit napevno, ale ponechat tuto volbu na uživateli. Tím tedy zavádím 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 hodnota zlomku. Pokud uživatel zadá jedna, proces bude tedy rychlý, ale méně kvalitní, oproti tomu deset bude 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 znovu. 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.

3.1.3 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 rychlost. Hodnota parametru však není předem úplně jasná, je to velmi závislé 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 zadaná, 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 znovu. 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 znovu. 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 rychlostí 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.

3.2 Struktura a použití zdrojového kódu

3.2.1 Struktura zdrojového kódu

Protože se jedná o implementaci jednoúčelových algoritmů, struktura použitých tříd je poměrně jednoduchá. Pro každý algoritmus je vytvořená samostatná třída umístěná v balíčku `vicitis.tools` Dále jsem vytvořil dvě třídy, které implementují prokládání bodů rovinou pomocí metody nejmenších čtverců. Jde o třídy `LeastSquare`, která implementuje základní verzi, a `WeightedLeastSquare`, která implementuje váženou verzi metody nejmenších čtverců. Obě metodv balíčku `vicitis.inner.methods`. Tyto třídy jsou oddělené od obou algoritmů pro lepší znovupoužitelnost v rámci celého systému. Obě třídy jsou na použití velmi jednoduché, obsahují jednu statickou metodu `fitPlane`, která má na vstupu body a případně jejich váhy a vrací instanci třídy `Plane3D`, tedy reprezentaci roviny.

První algoritmus, segmentace na základě velikosti normálového vektoru, se nachází v třídě `PlannarSegmentation`. Rozhraní třídy pro běžné používání je poměrně jednoduché. Obsahuje dvě jednoduché metody, pomocí kterých se třídě nastavuje vstupní mračno bodů, resp. 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 pointcloud pro segmentaci
3   * @param pc vstupni pointcloud (mracno bodu)
4   */
5  public void setPointcloud(ArrayList<Point3D> pc) {
6    this.pointCloud = pc;
7  }
8
9  /**
10  * Vraci segmentovany pointcloud
11  * @return segmentovany pointcloud
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 jsou do těchto metod zadávané, jsou popsané v předchozí sekci 3.1.3.

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

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 klastrování bodů poznamenává, která políčka byla již projita.

3.2.2 Ukázkové použití algoritmu

Jak bylo naznačeno v sekci 3.1.3, jsou dvě základní možnosti, jak algoritmus používat. V ukázce použití uvedu tu složitější variantu, kdy se nejprve provede uložení bodů do aku-

mulátorového pole a poté je možné provést klastrování bodů vícekrát. Zdrojový kód této varianty je zobrazen v ukázce 3.3.

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  PlannarSegmentation segmentation = new PlannarSegmentation();
9  segmentation.setPointcloud(pc);
10 //parametr urcujici velikost okoli
11 int pps = getInput("Zadejte velikost okoli bodu (cca. 20-200): ");
12 //ulozime body do akumulatrovoeho 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.
           getSegmentedPointCloud();
29     VRMLexporter.exportSegmentedPointClouds(spc,new File("out-iterace.wrl"));
30
31     param = getInput("Pokud chcete pokracovat v tomto klastrovani, "
32       + "zadejte vyssi hodnotu parametru pro klastrovani, "
33       + "(0 = konec tohoto klastrovani): ");
34     if(param == 0) break;
35   } while(true);
36   //podminka, zda pokracovat novym klastrovanim...
37 }
```

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 probíhalo algoritmu dvěma způsoby. Nejprve je nutné na nějakých „jednoduchých“ datech ověřit, zda algortimus 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 zahozeny. 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

4.2 Testování kvality algoritmů na reálných datech

- Způsob, průběh a výsledky testování.
- Srovnání s existujícími řešeními, pokud jsou známy.

Kapitola 5

Závěr

- Zhodnocení splnění cílů DP/BP a vlastního přínosu práce (při formulaci je třeba vzít v potaz zadání práce).
- Diskuse dalšího možného pokračování práce.

Literatura

[1] ABELES, P. Java Matrix Benchmark, . http://code.google.com/p/java-matrix-benchmark/, stav z 1.5.2012.

[2] ABELES, P. Efficent 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]. Dostupné z: ¡http://en.wikipedia.org/wiki/K-d_tree¿.

[5] Přispěvatelé Wikipedie. *Least squares* [online]. 2012. [cit. 1. 5. 2012]. Dostupné z: ¡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.

[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

2D Two-Dimensional

ABN Abstract Boolean Networks

ASIC Application-Specific Integrated Circuit

⋮

Příloha B

Instalační a uživatelská příručka

Tady asi nemám moc co psát...

Příloha C

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce.

Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.