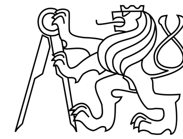
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název:	Efektivní algoritmy pro ešení problému nalezení konvexní obálky v 3D
Student:	Václav Motyka
Vedoucí:	doc. Ing. Ivan Šime ek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Nastudujte algoritmy pro ešení problému nalezení 3D konvexní obálky v 3D prostoru, zejména Gift Wraping, Divide and Conquer, Incremental algorithm (viz [1-5])
2) Po dohod s vedoucím vybrané algoritmy (alespo 3) naimplementujte.
3) Analyzujte možnosti jejich paralelizace a navrhn te jejich paralelní ešení pomocí technologie OpenMP.
4) Vygenerujte testovací data r zného charakteru (body uvnit r zných geometrických tvar , lokální shluky bod aj.)
5) Na fakultním serveru Star porovnejte výkonnost implementací algoritm a ov te jejich správnost s n jakým již existujícím ešením (nap . s knihovnou QHull [6]).

Seznam odborné literatury

[1] Mitura, Peter, Efektivní algoritmy pro ešení problému nalezení konvexní obálky, BP FIT VUT, 2016
[2] Roger Hernando, Convex hull algoriths in 3D:
http://dccg.upc.edu/people/vera/wp-content/uploads/2014/11/GA2014-ConvexHulls3D-Roger-Hernando.pdf
[3] Petr Felkel, Convex hull algoriths in 3D:
https://cw.fel.cvut.cz/wiki/_media/misc/projects/oppa_oi_english/courses/ae4m39vg/lectures/05-convexhull-3d.pdf
[4] M. de Berg, O. Cheong, M. van Kreveld, M.Overmars, Computational Geometry Algorithms and Applications, Third Edition (March 2008), Springer- Verlag
[5] T.M. Chan, Optimal output-sensitive convex hull algorithms in two and three dimensions Computational Geometry, April 1996, Volume 16, Issue 4,pp 361–368
[6] http://www.qhull.org/

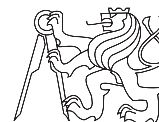
doc. Ing. Jan Janoušek, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 7. února 2017

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra Teoretické Informatiky



Bakalářská práce

Efektivní algoritmy pro řešení problému nalezení konvexní obálky v 3D

Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

11. května 2017

Poděkování

V první řadě děkuji doc. Ing. Ivanu Šimečkovi, Ph.D. za to, že svolil vést mi práci, přestože jsem se na něj obrátil na poslední chvíli. Dále mu děkuji za rady a zpětnou vazbu, kterou mi poskytl při psaní práce. Děkuji také své rodině a přítelkyni, že mne po dobu psaní práce podporovali a věřili mi i v časech, kdy jsem si já sám nevěřil.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 11. května 2017 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Václav Motyka. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Motyka, Václav. *Efektivní algoritmy pro řešení problému nalezení konvexní obálky v 3D*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se zabývá efektivními algoritmy pro řešení problému hledání konvexní obálky bodů ve 3D prostoru. V práci je tento problém zadefinován a jsou navrženy a popsány algoritmy pro jeho řešení. Tyto algoritmy jsou dále naimlementované, optimalizované a je analyzována možnost jejich paralelizovace. Dále jsou tyto algoritmy porovnané mezi sebou a zároveň proti již existujícímu řešení, v podobě knihovny QHull.

Klíčová slova konvexní obálka, výpočetní geometrie, QuickHull, gift wrapping, inkrementální algoritmus, C++

Abstract

This thesis deals with effective algorithms for finding convex hull in 3D space. In the thesis, the problem is defined and algorithms to solve it are described. These algorithms are implemented, optimized and analyzed for parallelization. Also, these algorithms are compared against each other and against an existing solution - QHull library.

Keywords convex hull, computational geometry, QuickHull, gift wrapping, incremental algorithm, C++

Obsah

Úvod **1**
Cíl práce 1
Struktura práce 2

1 Teorie **3**
1.1 Základní pojmy 3
1.2 Konvexní obálka 4
1.3 Dolní hranice složitosti 8

2 Softwarové a hardwarové aspekty **9**
2.1 Použité technologie 9
2.2 Existující řešení 11
2.3 Datové struktury 11

3 Algoritmy pro nalezení konvexní obálky ve 3D **15**
3.1 Jarvis March 15
3.2 Algoritmus Divide and Conquer 17
3.3 Inkrementální algoritmus 20
3.4 QuickHull 24
3.5 Další algoritmy 26

4 Optimalizace **27**
4.1 Obecné optimalizace 27
4.2 Jarvis March 28
4.3 Inkrementální algoritmus 29
4.4 QuickHull 31
4.5 Fat Planes 32
4.6 Možnosti paralelizace 34

5 Výsledky **37**

5.1 Testovací data 38
5.2 Jednotlivé algoritmy 38
5.3 Celkové porovnání 40

Závěr **43**

Literatura **45**

A Seznam použitých zkratek **49**

B Instalační příručka **51**
B.1 Požadavky 51
B.2 Instalace a použití 51
B.3 Formát vstupu a výstupu 51

C Obsah přiloženého CD **53**

Seznam obrázků

1.1 Konvexní a nekonvexní množina v $\mathbb{R}^2$ . . . 4
1.2 Konvexní obálka v $\mathbb{R}^2$ a $\mathbb{R}^3$ . . . 5
1.3 Nadrovina a její poloprostory v $\mathbb{R}^2$ . . . 5
1.4 Simplexy v prostorech $\mathbb{R}^1$, $\mathbb{R}^2$ a $\mathbb{R}^3$ . . . 6
1.5 Topologie 3-simplexu . . . 6

2.1 Ukázka half-edge struktury . . . 12

3.1 Sloučení dvou konvexních obálek . . . 18
3.2 Sloučení dvou konvexních obálek . . . 19
3.3 Čtyřstěn pro výpočet objemu . . . 21
3.4 (a) Konvexní obálka H_{i-1} před přidáním bodu v rohu (b) Konvexní obálka H_i . . . 22
3.5 Horizont a viditelné stěny (bíle) . . . 23

4.1 Funkce kosinus . . . 28
4.2 Graf konfliktů . . . 30
4.3 Přidání nové stěny . . . 31
4.4 Chybné stěny před sloučením . . . 32
4.5 Test konvexnosti stěn F_{Left} a F_{Right} . . . 33
4.6 Fat planes . . . 33

5.1 Kompilátorová optimalizace algoritmu Jarvis march . . . 39
5.2 Měřené časy inkrementálního algoritmu pro všechna testovací data 40
5.3 Porovnání algoritmů pro testovací data č. 1 . . . 41
5.4 Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 3 . . . 42
5.5 Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 4 . . . 42

Seznam tabulek

5.1 Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 2 40

Úvod

Problém hledání konvexní obálky patří mezi jeden ze základních problémů výpočetní geometrie. Jeho podstatou je nalezení takového mnohostěnu, že všechny body leží buď na stěnách tohoto mnohostěnu, nebo uvnitř něj. Mezi další významné problémy výpočetní geometrie, které jsou navíc s konvexní obálkou úzce spojené, patří například Delunayova triangulace [1] a Voronoiovy diagramy [2]. Důvod, proč tento problém patří mezí významné, je zejména jeho využití. Hledání konvexní obálky se využívá například pro detekci kolizí (hledání průniku dvou objektů) ve fyzikálních simulacích nebo počítačových hrách. Mimo to se dá nalézt využití i v robotice nebo jako součást složitějších algoritmů výpočetní geometrie. Díky významnosti tohoto problému se mu dostává pozornosti již dlouhou dobu [3, 4, 5, 6].

V Euklidovském prostoru E^2 existuje několik známých algoritmů pracujících v čase $\mathcal{O}(n \log n)$, kde n je počet vstupních bodů. Mezi ně patří například *Graham scan* [7]. Pro malá n je vhodné i použití algoritmů pracujících v čase $\mathcal{O}(nh)$ (kde n je počet vstupních bodů a h počet bodů, ležících na obálce), například *Jarvis march*, známý též pod názvem *Gift wrapping* [8]. Asymptoticky nejrychlejší algoritmy v E^2 pracují v čase $\mathcal{O}(n \log h)$, jak bylo dokázáno v práci Kirkpatricka a Seidela, jež poskytli i konkrétní algoritmus [9].

V prostoru E^3 jsou známé algoritmy pracující v čase $\mathcal{O}(n \log h)$ a hůře. Některé z těchto algoritmů budou popsány a implementovány v této práci, konkrétně v kapitole 3.

Cíl práce

Cílem této práce je implementace některých známých algoritmů pro vytvoření konvexní obálky bodů v prostoru E^3. Dalším cílem je porovnání jejich efektivity ve smyslu rychlosti, za jakou jsou tyto algoritmy schopné daný problém vyřešit.

Struktura práce

V první kapitole bude zadefinována konvexní obálka, problém jejího nalezení a další potřebné pojmy.

Ve druhé kapitole budou popsány hardwarowé a softwarové aspekty mé práce.

Ve třetí kapitole budou popsány nektěré známé algoritmy pro vytvoření konvexní obálky a vysvětlení jejich složitosti.

Ve čtvrté kapitole budou popsány možné optimalizace algoritmů z kapitoly dva a analyzována možnost jejich paralelizace.

V páté kapitole bude ukázáno srovnání jednotlivých algoritmů v závislosti na jejich výkonu.

Kapitola 1

Teorie

V této kapitole budou zadefinovány důležité pojmy používané v této práci. Definice vychází z práce kolegy Mitury [10] a knih Computational Geometry in C [5] a Computational Geometry: Algorithms and Applications [11].

1.1 Základní pojmy

Než definujeme pojem konvexní obálka, musíme zadefinovat pojem konvexní množina. Jelikož naše algoritmy budou pracovat na tělese reálných čísel, zadefinujeme si nejdříve lineární prostor $\mathbb{R}^n$:

Definice 1. Nechť $n \in \mathbb{N}, n > 0$. *Lineární prostor* $\mathbb{R}^n$ je množina uspořádaných n-tic (*vektorů*) prvků (*skalárů*) z $\mathbb{R}$, vybavená dvěma operacemi:

$$\oplus : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n : \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \oplus \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

$$\odot : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n : k \ \odot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} kx_1 \\ kx_2 \\ \vdots \\ kx_n \end{pmatrix}$$

Dále si zavedeme pojem *úsečka*. Nadále v této práci budeme pro prvky $\mathbb{R}^n$ nazývat body namísto vektory. Jednotlivé složky vektoru si můžeme představit jako souřadnice bodu v dané dimenzi, například vektor $\mathbb{R}^3$ se skládá ze tří složek (souřadnice bodu ve třech dimenzích).

Definice 2. Nechť *x,y* jsou body v lineárním prostoru $\mathbb{R}^n$. Potom množinu

$$\{\lambda x + (1 - \lambda y) \mid 0 <= y <= 1\}$$

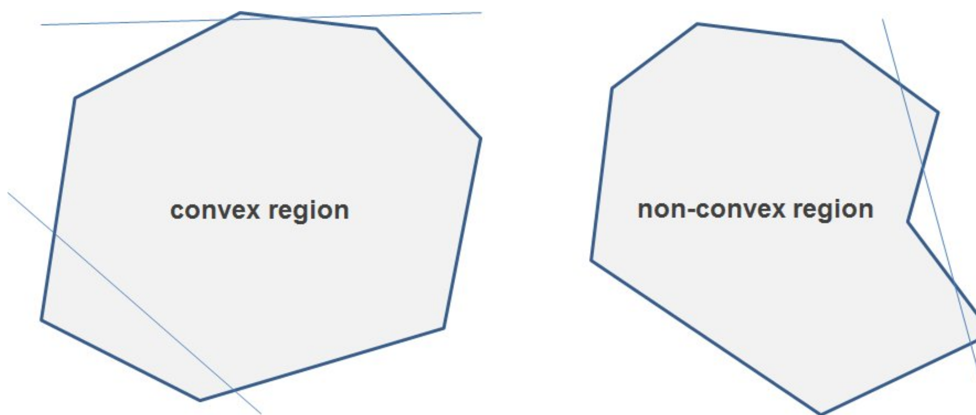
nazýváme *úsečkou s koncovými body x,y* a označujeme ji $\overline{xy}$.

Nyní můžeme zadefinovat konvexní množinu.

Definice 3. Množinu S v lineárním prostoru $\mathbb{R}^n$ nazýváme konvexní, pokud pro každou dvojici bodů x, y z S platí, že i úsečka $\overline{xy}$ se nachází v množině S.

$$\forall x, y \in S \ \Rightarrow \overline{xy} \in S$$

Konvexnost množiny si můžeme ilustrovat na obrázku 1.1.



Obrázek 1.1: Konvexní a nekonvexní množina v $\mathbb{R}^2$

zdroj: [12]

1.2 Konvexní obálka

Díky zavedení jednotlivých pojmů v předchozí sekci můžeme nyní definovat konvexní obálku.

Definice 4. *Konvexní obálka* konečné množiny bodů M v lineárním prostoru $\mathbb{R}^n$ je průnik všech konvexních množin obsahujících M.

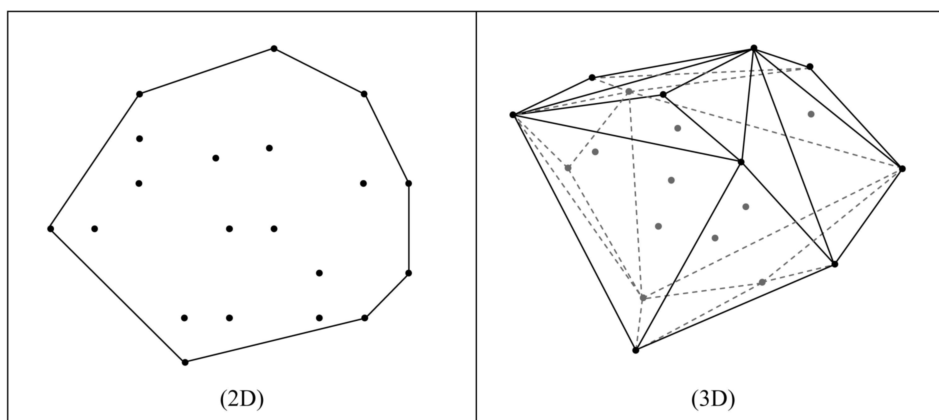
Ukázku konvexní obálky bodů v prostoru $\mathbb{R}^2$ a $\mathbb{R}^3$ můžeme vidět na obrázku 1.2.

Pro lepší pochopení zavádíme ještě další definice.

Definice 5. *Nadrovina* daného prostoru dimenze n je jakýkoliv jeho podprostor dimenze $n - 1$.

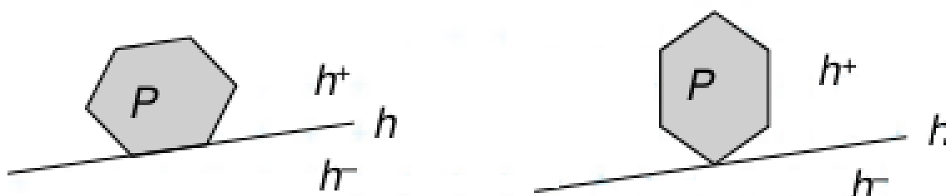
V $\mathbb{R}^2$ je tedy nadrovinou přímka a v $\mathbb{R}^3$ je nadrovinou rovina. Platí, že v eukleidovském prostoru dělí každá nadrovina h prostor na dva poloprostory h^+ a h^-, viz obrázek 1.3.

Konvexní obálku bodů v $\mathbb{E}^n$ můžeme též nazvat *polytop*. Tento termín použijeme v další definici.



Obrázek 1.2: Konvexní obálka v $\mathbb{R}^2$ a $\mathbb{R}^3$

zdroj: [13]



Obrázek 1.3: Nadrovina a její poloprostory v $\mathbb{R}^2$

Převzato z [14]

Definice 6. Nechť $v_0, v_1 \dots v_n$ jsou body v prostoru $\mathbb{R}^n$. Tyto body se nazývají afinně nezávislé, jestliže neexistují taková reálná čísla $\alpha_0\alpha_1 \dots \alpha_n$, která nejsou všechna 0 tak, že $\sum_{i=0}^{k} \alpha_i v_i = 0$ a $\sum_{i=0}^{k} \alpha_i = 0$

Definice 7. *K-simplex* je konvexní obálka $k+1$ afinně nezávislých bodů.

K-simplex je tedy speciálním typem konvexní obálky množiny bodů, která obsahuje všechny její body. Simplexem v rovině je tedy trojúhelník a simplexem v prostoru je jehlan, jak je vidno na obrázku 1.4.

Definice 8. Stěna *konvexní obálky (polytopu)* v $\mathbb{R}^n$ je průnik této obálky s nějakou její nadrovinou.

Předchozí definice je trochu krkolomná a nejlépe se ukáže na příkladu: v $\mathbb{R}^2$ jsou stěnami úsečky a v $\mathbb{R}^3$ to jsou mnohoúhelníky. Z toho a z předchozích

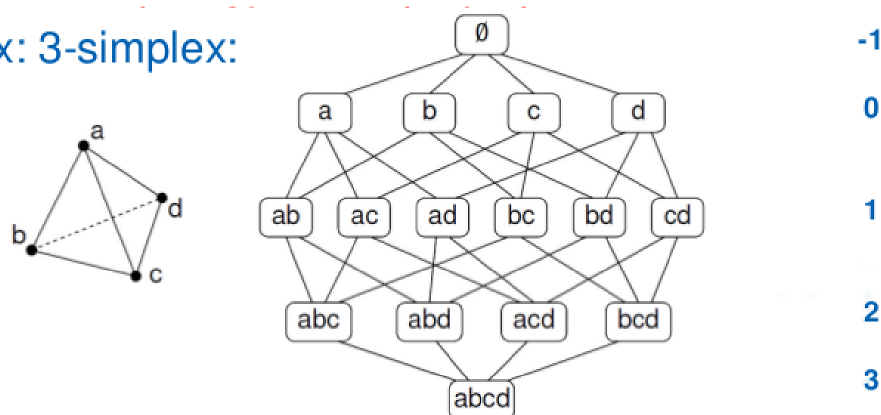
Obrázek 1.4: Simplexy v prostorech $\mathbb{R}^1$, $\mathbb{R}^2$ a $\mathbb{R}^3$

Převzato z [14]

definic si můžeme odvodit, že zatímco v $\mathbb{R}^2$ je konvexní obálkou mnohoúhelník, v $\mathbb{R}^3$ je jí mnohostěn (pro formální definici se odkážeme na [15]).

Ještě je hodno dodat, že existují speciální názvy pro stěny v dimenzi 0 a 1. V dimenzi 0 budeme stěny nazývat *vrcholy* a v dimenzi 1 *hrany*.

Každý polytop v $\mathbb{E}^d$ se skládá z pravých stěn a nepravých stěn. [1] Pravé stěny jsou stěny dimenze $0 \ldots (d-1)$, nepravé jsou pak stěny dimenze -1 (prázdná množina) a dimenze d (celý polytop). Díky tomuto můžeme ukázat na příkladu 3-simplexu topologii polytopu 3-simplex (obrázek 1.5).



Obrázek 1.5: Topologie 3-simplexu

Převzato z [14]

O polytopech dále platí několik tvrzení [14]:

- Ohraničení polytopu tvoří sjednocení jeho pravých stěn
- Každý polytop je tvořen konečným množstvím stěn
- Každá stěna polytopu je polytop

[1] Název pravé a nepravé stěny je pouze můj překlad výrazů *proper face* a *improper face*, definovaných v [14]

- Polytop je konvexní obálkou svých vrcholů

Na konci této kapitoly je třeba ještě uvést jednu důležitou větu a její důsledky. Tato věta je důležitá pro pozdější důkazy složitosti jednotlivých algoritmů.

Věta 1 (Eulerova rovnice)**.** *Nechť M je konvexní mnohostěn, V je počet jeho vrcholů, E počet jeho hran a F počet jeho stěn. Potom platí:*

$$V - E + F = 2 \tag{1.1}$$

Pro důkaz této věty by bylo třeba zadefinovat další pojmy, které pro tuto práci nejsou potřebné, proto se pro důkaz odkážeme na [5, str. 106–108].

Nyní za využití předchozí věty můžeme uvést ještě jednu, která z ní vyplývá:

Věta 2. *Nechť M je konvexní mnohostěn, V je počet jeho vrcholů a E počet jeho hran. Potom pro M platí:*

$$h \leq 3v - 6$$

Důkaz. Označíme F jako počet stěn mnohostěnu M. Každá hrana inciduje právě se dvěmi různými stěnami a každá stěna je tvořena minimálně třemi hranami. Potom platí:

$$F \leq \frac{2h}{3} \tag{1.2}$$

Nyní můžeme z 1.1 vyjádřit s a dosadíme do něj 1.2, dostaneme

$$\frac{2h}{3} \geq 2 - V + E \tag{1.3}$$

Jednoduchými úpravami se dostáváme k původní rovnici

$$h \leq 3v - 6$$

□

Důsledek 1. Nechť M je konvexní mnohostěn. Potom M má $\mathcal{O}(V)$ stěn a $\mathcal{O}(V)$ hran.

Nakonec je ještě důležité zadefinovat dva pojmy, které budou využívány u popisů jednotlivých algoritmů.

Definice 9. Nechť body a, b, c jsou body ležící v prostoru $\mathbb{R}^3$, potom tyto body nazveme *kolineární*, pokud leží na jedné přímce.

Definice 10. Nechť body a, b, c, d jsou body ležící v prostoru $\mathbb{R}^3$, potom tyto body nazveme *koplanární*, pokud leží v jedné rovině. Zároveň dvě stěny mnohostěnu jsou *koplanární*, pokud leží ve stejné rovině.

V dalším textu budeme užívat značení n pro počet vstupních bodů algoritmu a h pro počet bodů tvořících výsledkou konvexní obálku.

1.3 Dolní hranice složitosti

Při porovnání efektivity a výkonu algoritmu se jako základní ukazatel využívá dolní hranice jejich složitosti. U algoritmů pro nalezení konvexní obálky v $\mathbb{E}^2$ platí, že dolní hranice jejich složitosti je v nejhorším případě minimálně taková, jakou mají algoritmy na řešení řazení reálných čísel [10]. Pro řazení reálných čísel platí, že zatím neznáme algoritmy pracující rychleji než v čase $O(n \log n)$, a proto můžeme říci, že algoritmy pro hledání konvexní obálky pracují v čase $\Omega(n \log n)$. Toto ovšem platí pouze u algoritmů, kde všechny vstupní body leží na obálce. Pokud všechny body na obálce neleží, byly nalezené algoritmy, jejichž horní hranice složitosti je $\mathcal{O}(n \log h)$. Algoritmy, jejichž složitost závisí na jejich výstupu, nazýváme *output-sensitive*, neboli *citlivé na výstup*, opakem jsou pak algoritmy *necitlivé na výstup*. U algoritmů pro nalezení konvexní obálky se setkáváme s oběma typy.

Jak již bylo řečeno v úvodu, pokud chceme nalézt konvexní obálku v $\mathbb{E}^3$, nejrychlejší algoritmy pracují v čase $\mathcal{O}(n \log h)$. Podobný důkaz pro dolní hranici složitosti jako u algoritmů v $\mathbb{E}^2$ jsem ale bohužel nenašel.

Kapitola 2

Softwarové a hardwarové aspekty

V této kapitole budou popsány softwarové a hardwarové aspekty mé práce. Budou zde popsány použité knihovny, datové struktury a existující řešení.

2.1 Použité technologie

V mé práci jsem se rozhodl algoritmy implementovat samostatně, ne v podobě knihovny, jako v případě kolegy Mitury [10]. Zároveň jsem přemýšlel o rozšíření jeho práce, nakonec jsem z ní ale pouze vycházel. Má implementace byla vyvíjena pod operačním systémem GNU/Linux a na něj je i cílená. Instalační příručka, potřebné technologie a popis použití je uveden na konci práce. Přestože je implementace realizována pod GNU/Linux, není vyloučené ani použití pod jiným operačním systémem.

2.1.1 Programovací jazyk

Pro implementaci jednotlivých algoritmů byl zvolen programovací jazyk C++. Jedná se o nízkoúrovňový programovací jazyk, z tohoto důvodu by měl poskytovat dostatečně dobrý výkon, lepší než jazyky vysokoúrovňové, a to z důvodu nižší režie, možnosti optimalizovat program na nižší úrovni a možnosti využít automatické kompilátorové optimalizace. Jazyk C++ je často využíván v oblastech výpočetní geometrie a v odvětvích využívajících rychlé geometrické výpočty, jako je herní průmysl nebo grafické aplikace.

2.1.2 Použité knihovny

Samozřejmostí moderního programování je využití knihoven pro zjednodušení práce. Jejich hlavní výhodou je odladěnost a optimální výkon. V mojí práci

jsem nejčastěji využíval kontejnery z STL knihovny C++ [16]. Nejčastěji využívaným kontejnerem byl bezesporu *vector*, ulehčující práci s poli. Využit byl ve všech algoritmech, převážně pro postupné ukládání stěn, hran a vrcholů, nalezených v průběhu daného algoritmu. Dále byl využíván kontejner *set* pro práci s množinami. Kromě STL knihovny byla využita knihovna *algorithm* [17], resp. funkce *sort* pro řazení polí z ní dostupná. Dále jsem využíval pro matematické výpočty knihovnu *cmath* [18]. Tato knihovna ovšem neobsahuje funkce pro výpočty geometrie (například funkce pro skalární součin vektorů nebo vektorový součin), tyto funkce byly převzaty z práce kolegy Mitury [10] a doplněné o další potřebné.

2.1.3 OpenMP

Knihovna *OpenMP* je knihovna pro paralelní programování v jazycích Fortran, C a C++. Důvod použití *OpenMP* je výrazně jednodušší použití oproti základní knihovně pro paralalelní programování v C++ `pthread.h`, implementující standard POSIX. Hlavním důvodem paralelizace je rozložení práce mezi několik jader procesoru a tím zrychlení běhu daného algoritmu.

Definice 11. *Zrychlení S* definujeme jako poměr doby běhu neparalaleního algoritmu T_{old} oproti době běhu jeho paralelní verze T_{new}.

$$S = \frac{T_{old}}{T_{new}}$$

Ne vždy ovšem paralelizace vede ke zrychlení. Pokud je $S < 1$, mluvíme o *zpomalení*. V průběhu paralelního algoritmu může vznikat *časově závislá chyba*, způsobená současným čtením ze stejného paměťového úseku a zápisem do něj z různých vláken programu. To s sebou přináší možné chyby jako čtení z již neplatné adresy, zápis na neplatnou adresu nebo čtení nesprávné hodnoty způsobené současným zápisem. Pro eliminaci těchto chyb je třeba v programu definovat takzvané *kritické sekce*, což jsou bloky kódu, do kterých v jednu chvíli může přistupovat pouze jedno vlákno.

Možnosti OpenMP

Knihovna OpenMP nabízí širokou škálu nástrojů nejen pro paralelizaci. Využívána byla například funkce pro měření času běhu algoritmu. Pro paralelizaci nabízí OpenMP několik možností, nejdůležitější z nich si nyní uvedeme. Jejich společným znakem je direktiva `#pragma omp`, která překladači říká, že nyní bude využíváno OpenMP.

- **Direktiva `for`** Slouží pro paralelizaci cyklů přiřazením jednotlivých iterací vláknům. Způsob, jakým jsou iterace přidělovány, je definován klauzulí `schedule`. Mezi hlavní možnosti patří statické plánování, které

iterace naplánuje vláknům ještě před samotným cyklem, případně plánování dynamické, které iterace přiděluje za běhu. To je náročnější na režii, ale často umožňuje rovnoměrnější využití vláken.

- **Direktiva `task`** Vytvoří novou úlohu, která se následně uloží do takzvaného *taskpoolu*, odkud si ji mohou vlákna vybrat a následně vykonat. Direktivou `taskwait` se dá definovat místo v programu, kde se čeká na dokončení všech úloh. Nejčastěji je využívána pro rekurzivní algoritmy.
- **Direktiva `section`** Funguje podobně jako `task`, danému bloku kódu přiřadí nové vlákno. Má výrazně nižší režii než `task`, z důvodu chybějící nutnosti režie *taskpoolu*.

2.2 Existující řešení

V současnosti existuje několik implementací algoritmů pro nalezení konvexní obálky ve 3D prostoru. Přesto ale uvedu pouze dvě, z důvodu, že jsou nejznámější a hlavně řádně otestované.

- **QHull** [19] Jedná se o asi nejznámější open-source knihovnu, implementuje algoritmus *QuickHull* a pracující v libovolné dimenzi. Kromě něj obsahuje spoustu dalších nástrojů, jako například modul `rbox`, umožňující generování bodů v dané dimenzi.
- **CGAL** [20] Tato knihovna umožňuje stejně jako *QHull* nalezení konvexní obálky v libovolné dimenzi. Pro body v prostoru $\mathbb{R}^3$ nabízí možnost využít inkrementální algoritmus nebo *QuickHull*. Knihovna je vyvíjena jako open-source pod licencemi GPL/LGPL.

Kromě těchto dvou knihoven existují ještě další implementace, mezi hlavními zmíním implementaci *QuickHullu* v Javě [21] a implementaci inkrementálního algoritmu z knihy [5].

2.3 Datové struktury

Jelikož jsem u každého implementovaného algoritmu vycházel z jiného zdroje, tak se lehce odlišují i využité datové struktury. Často byly využívány kontejnery z STL knihovny C++ [16]. Jejich hlavní výhodou je odladěnost, měly by pracovat bezchybně a s optimálním výkonem.

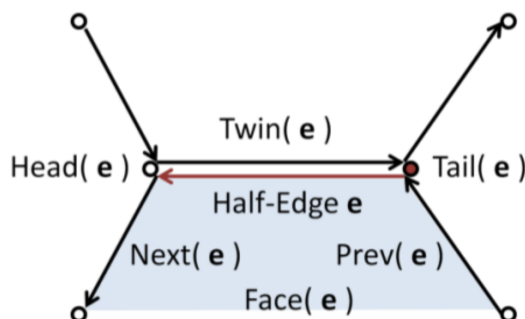
2.3.1 Reprezentace mnohostěnu

Jak správně reprezentovat mnohostěn pro co nejsnadnější práci, je v oblastech počítačové grafiky, geometrického modelování a výpočetní geometrie, důležitá otázka. Z tohoto důvodu vznikly různé návrhy datových struktur, které práci

s mnohostěny usnadňují. Jeden z nich, konkrétně *half-edge* struktura, byla využita i v mé imlementaci.

2.3.1.1 Half-edge struktura

Jak již bylo řečeno, mnohostěn je tvořen *vrcholy, hranami* a *stěnami.* Každá hrana inciduje právě se dvěmi vrcholy a dvěma stěnami. Vrchol může náležet několika hranám a stěnám, vždy ale alespoň třem. V průběhu algoritmů je často potřeba iterovat přes všechny stěny, hrany nebo vrcholy. Také je často potřeba znát, s jakými stěnami hrana inciduje nebo jaké hrany či vrcholy tvoří stěnu. Vhodnou reprezentací, která nám umožňuje tyto informace uchovávat je *half-edge* struktura. Hlavní rozdíl oproti naivní struktuře je rozdělení každé hrany na dvě s opačnou orientací, každou náležející jedné z incidujících stěn. To nám zároveň umožňuje stěny reprezentovat v uspořádání po, případně proti směru hodinových ručiček. Ukázka této struktury je na obrázku 2.1.



Obrázek 2.1: Ukázka half-edge struktury

Převzato z [22]

Pro každou stěnu si uchováváme jednu jeji hranu (na obrázku jako *e*). Jelikož hrany budou u každé stěny tvořit uzavřený kruhový spojový seznam, díky znalosti jediné hrany můžeme iterovat přes všechny hrany stěny.

Pro každou hranu si uchováváme informace o stěně, ke které patří *Face(e)*, následující *Next(e)* i předchozí hraně *Prev(e)* této stěny a hraně protilehlé *Twin(e)*. Dále je třeba znát koncový bod této hrany (vrchol, *Head(e)*). Není třeba znát její počáteční bod *Tail(e)*, ten můžeme najít jako koncový bod hrany přechozí.

Co se týče vrcholů, záleží, co za informace o nich budeme využívat. Můžeme si uchovávat seznam stěn, ke kterým vrchol patří, případně seznam hran, které z něj nebo do něj vedou. Tyto informace nejsou ale vždy potřebné, a proto záleží na konrétní implementaci, zda je využije nebo nikoliv.

2.3.1.2 Další možné struktury

Mezi další často používané datové struktury pro reprezentaci mnohostěnů patří *Winged-edge* struktura, která si pro každou hranu uchovává své incidující stěny a svého předchůdce a následníka na obou těchto stěnách. Tyto následníci a předchůdci tvoří jakási *křidýlka* (anglicky *wings*) této hrany, od toho také název této struktury. Složitější strukturou, kterou jsem se do detailu nezabýval, je ještě *quad-edge* struktura [23].

KAPITOLA 3

Algoritmy pro nalezení konvexní obálky ve 3D

V této kapitole budou popsány některé známé algoritmy pro nalezení konvexní obálky v $\mathbb{E}^3$, zároveň bude i řečena a dokázána jejich složitost, případně bude odkázáno na literaturu, kde se důkaz složitosti nachází.

Některé z těchto algoritmů, jako je například *Jarvis march* (3.1) nebo *QuickHull* (3.4) jsou rozšířením své 2D verze. Algoritmy pracující v prostoru $\mathbb{E}^2$ v této práci však popisovat nebudu, budu ovšem vycházet z popisu uvedeného v práci kolegy Mitury [10].

3.1 Jarvis March

Jedná se o jeden ze základních algoritmů pro hledání konvexní obálky, též často referovaný jako *Gift Wrapping* (balení dárků), který byl ve své 2D verzi představen v roce 1973 R. A. Jarvisem [8] a dosahoval rychlosti $\mathcal{O}(nh)$. Již v roce 1970 byl Donaldem Chandem a Shamem Kapurem ukázán algoritmus pracující v libovolném prostoru, ovšem se složitostí $\mathcal{O}(n^2)$ [24]. Jarvisův algoritmus je založen právě na algoritmu Chanda a Kapura.

3.1.1 Popis algoritmu

Jak již bylo řečeno, tento algoritmus je rozšířením své 2D verze. Narozdíl od ní ovšem nehledáme strany mnohoúhelníku, nýbrž stěny mnohostěnu.

Na začátku tohoto algoritmu je třeba najít úsečku, ležící na nějaké jeho stěně (hranu nějaké jeho stěny). Máme jistotu, že extrémní body (s maximální hodnotou souřadnice x, y nebo z) budou ležet na konvexní obálce a budou jedněmi z jejích vrcholů. Označme si M vstupní množinu všech bodů. Lineárním průchodem všech těchto bodů najdeme bod s nejmenší hodnotou souřadnice y. Označme ho a. Nyní zbývá nalézt druhý bod úsečky. Označme si R rovinu, která prochází bodem a a má normálový vektor $(0, 1, 0)$. Pro všechny body

z $M \setminus \{a\}$ spočítáme úhel svírající s rovinou R v bodě a. Bod, který svírá úhel nejmenší, bude druhý bod hledané úsečky. Pokud takových bodů nalezneme více, vybereme libovolný, výsledný bod označíme b.

Pro nalezení první stěny hledaného mnohostěnu je třeba najít ještě jeden bod, označme ho c. Označme si R' rovinu určenou body a, b, c. Bod c bude takový bod, pro který platí, že všechny ostatní body z $M \setminus \{a, b, c\}$ budou ležet ne jedné pevně zvolené straně roviny R' a nebo přímo na ní (tyto body jsou *koplanární – ležící na stejné rovině* s body a,b,c). Tento krok provedeme opět lineárním průchodem všech bodů kromě a,b, přičemž na začátku si zvolíme za c první bod, na který průchodem narazíme (tím vytvoříme rovinu R') a pokud při průchodu zbylých bodů narazíme na nějaký, který neleží na dané straně R', zvolíme tento bod jako nové c.

Pokud při průchodu narazíme na nějaký bod, ležící na stejné rovině jako body a, b, c, uložíme si ho do množiny X. Tato množina bude představovat body, jež mohou tvořit aktuálně hledanou stěnu. Obsah této množiny smažeme, jakmile najdeme nový bod c. Jakmile dokončíme hledání bodu c, je potřeba najít hledanou stěnu. Víme, že ji budou tvořit některé nebo všechny body množiny $X \cup \{a, b, c\}$. Pro nalezení hledané stěny tyto body promítneme do prostoru $\mathbb{R}^2$ a najdeme jejich konvexní obálku některým algoritmem pro hledání obálky v $\mathbb{R}^2$. Tyto nalezené body budou tedy tvořit mnohoúhelník – jednu stěnu hledané obálky a budou seřazené proti směru hodinových ručiček (pokud tak bude nastaven algoritmus pro hledání obálky v $\mathbb{R}^2$). Označíme ho jako Z.

Jelikož každá hrana mnohostěnu inciduje právě se dvěma jeho stěnami, pro každou stranu nalezeného mnohoúhelníka zbývá najít ještě jednu stěnu. V průběhu algoritmu si budeme udržovat dvě množiny hran – $fresh$ a *closed*. Jakmile dokončíme hledání jedné stěny, všechny její hrany, které neleží v množině *closed*, uložíme do množiny $fresh$. Pokud se v této množině přidávaná hrana již nachází, z množiny ji vyřadíme a přidáme do množiny *closed*. Nyní pro každou hranu z množiny $fresh$ opakujeme stejný postup (hledání bodu c), dokud množina $fresh$ nebude prázdná. Jakmile bude prázdná, algoritmus ukončíme.

Pseudokód algoritmu je uveden níže.

Algoritmus 1: Jarvis March

```
1: t ← findInitialTriangle()
2: Q ← {(t_0, t_1), (t_1, t_2), (t_2, t_0)}
3: H ← t
4: while Q ≠ ∅ do
5:    e ← Q.pop_back()
6:    if notProcessed(e) then
7:       q ← pointsWithMaxAngle(e)
8:       f ← grahamScan(q)
9:       H ← H ∪ {f}
10:      Q ← edges(f)
11:      markProcessed(e)
12:   end if
13: end while
```

3.1.2 Složitost algoritmu

Věta 3. *Nechť vstupní množina bodů M obsahuje n bodů a mnohostěn tvořící její konvexní obálku má h vrcholů. Potom složitost algoritmu Jarvis march pro nalezení této obálky je $\mathcal{O}(nh)$.*

Důkaz. Důkaz tohoto tvrzení je založen na důsledku 1. Nalezení prvních dvou bodů jsme schopni provést lineárním průchodem, tedy v čase $\mathcal{O}(n)$. Při tvorbě stěny je potřeba pro $n-2$ bodů zkontrolovat, na jaké straně roviny leží, toto je třeba udělat pro $\mathcal{O}(h)$ stěn, ležících na výsledném mohostěnu. Dohromady tedy vychází složitost $\mathcal{O}(nh)$.

Ještě je třeba započítat složitost práce s množinami $fresh$ a *closed*. Přidání a mazání prvku v množině je možné uskutečnit v čase $\mathcal{O}(\log n)$ [25], kde n je počet prvků množiny. V našem případě může být množina velká maximálně tolik, kolik je hran, což je podle důsledku 1 $\mathcal{O}(v)$. Složitost práce s množinou je tedy maximálně $\mathcal{O}(\log h)$. S množinou pracujeme v každém průchodu, celkově tedy práce s množinami zabere $\mathcal{O}(n \log h)$ času.

Společně dostáváme složitost $\mathcal{O}(n) + \mathcal{O}(nh) + \mathcal{O}(n \log h) = \mathcal{O}(nh)$. Algoritmus *Jarvis march* je tedy *citlivý na vstup.* Pokud by všechny vstupní body ležely na konvexní obálce, jeho složitost by byla $\mathcal{O}(n^2)$. □

3.2 Algoritmus Divide and Conquer

Jak bylo řečeno v úvodu, spodní hranice složitosti pro konstrukci konvexní obálky v $\mathbb{E}^3$ je $\Omega(n \log n)$, stejně tak jako v $\mathbb{E}^2$. Přirozeně nás tedy zajímá algoritmus, který této složitosti dosahuje. Přestože bylo řečeno, že některé algoritmy se dají rozšířit z $\mathbb{E}^2$ do $\mathbb{E}^3$, jediný algoritmus, který dosahuje optimální

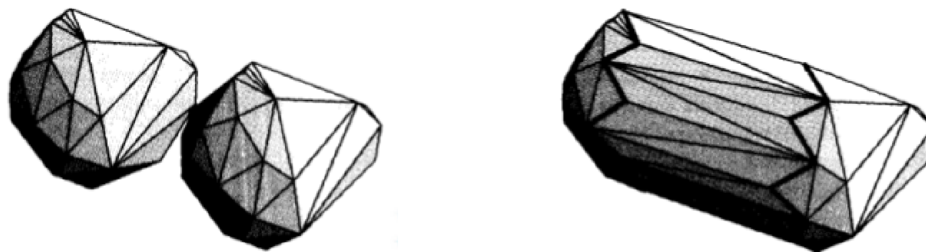
časové složitosti $\mathcal{O}(n \log n)$, je algoritmus představený v roce 1977 Francem P. Preparatou a S. J. Hongem [26] [5]. Tento algoritmus nakonec pro svou náročnost nebyl implementován, místo něj jsem pro implementaci vybral algoritmy popsané dále (3.3 a 3.4). Přesto je tento algoritmus zajímavý, proto poskytneme alespoň jeho popis.

3.2.1 Popis algoritmu

Algoritmus *Divide and Conquer* je založen na stejném principu jako jeho 2D verze.

Na začátku algoritmu je potřeba body seřadit na základě jedné ze souřadnic. Vybereme například souřadnici x. Dále body rekurzivně rozdělíme do dvou skupin, sestrojíme konvexní obálku těchto skupin (jakmile máme malé množství bodů, je možné sestrojit obálku za použití nějakého z ostatních algoritmů, popřípadě metodou *brute force*), a následně provedeme jejich sloučení. Abychom splnili podmínku pro časosovou náročnost $\mathcal{O}(n \log n)$, musí sloučení proběhnout v čase $\mathcal{O}(n)$. Jelikož se většina práce algoritmu odehrává ve fázi sloučení, budeme se nyní soustředit na ni.

Nechť A a B jsou konvexní obálky, které chceme sloučit. Konvexní obálku $A \cup B$ budou tvořit stěny, tvořící objekt připomínající válec bez podstav, jak je vidět například na obrázku 3.1.



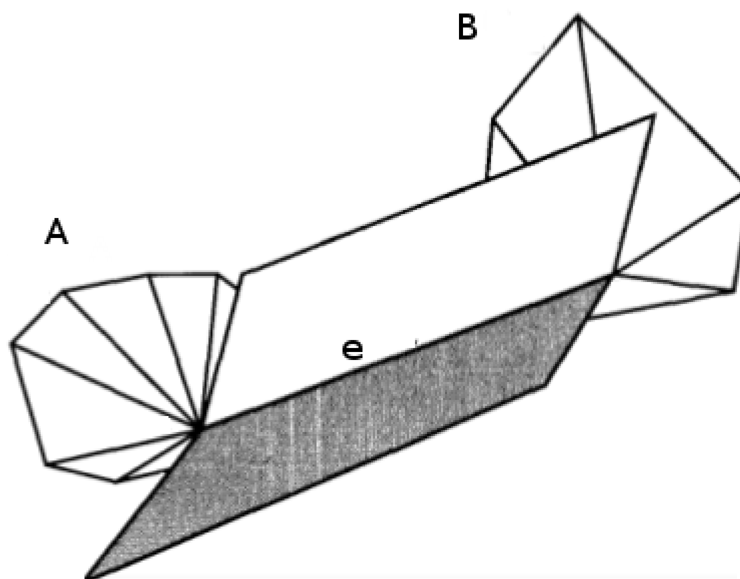
Obrázek 3.1: Sloučení dvou konvexních obálek

Převzato z [5]

Počet těchto nově přidaných stěn bude závislý lineárně na velikosti A a B. Každá nově přidaná stěna využívá alespoň jednu hranu z A nebo B, takže počet přidaných stěn bude maximálně tolik, kolik je hran. Tudíž spojení $A \cup B$ se dá dosáhnout v čase $\mathcal{O}(n)$ za předpokladu, že přidání nové stěny trvá konstantní čas.

Sloučení A a B začíná nalezením nějaké hrany, která je propojuje, nazvěme ji e. To se dá dosáhnout promítnutím A a B do roviny a využitím algoritmu pro hledání spodního tangentu dvou mnohoúhelníků. Tento algoritmus se využívá i ve 2D verzi *Divide and Conquer* a je popsán například v [5, str. 94–96].

V další části využijeme algoritmu *Gift Wrapping* (3.1) a obalujeme A a B rovinou procházející e, dokud nenarazíme na bod, ležící na A nebo B. To nám vytvoří první stěnu. Nyní v obalování pokračujeme, dokud nenarazíme opět na e. V tu chvíli máme vytvořené sloučení A a B, ještě je však třeba zbavit se stěn ležících uvnitř tohoto sloučení – některých z původních stěn A a B. Průběh obalování demonstruje obrázek 3.2.



Obrázek 3.2: Sloučení dvou konvexních obálek

Převzato z [5]

Stěny, které je třeba odstranit, jsou stěny A, jež jsou viditelné z nějakého vrcholu B a naopak. Bohužel algoritmus nedokáže tyto stěny za chodu detekovat, dokáže ale detekovat hrany, se kterými tyto stěny incidují – to jsou hrany, které nalezneme obalováním ve fázi sloučení, ležící buď na A nebo B, v obrázku 3.1 označeny tučně. Stačí tedy zkontrolovat stěny incidující s těmito hranami a viditelné stěny odstranit. [2]

Z důvodu naročnosti nebyl tento algoritmus implementován a kvůli tomu ani jeho popis nebyl tak detailní, přesto a možná právě proto lépe pochopitelný. Pro detaily odkazuji na původní práci Preparaty a Honga [26], kde jsou všechny kroky popsány detailně, zároveň s důkazem složitosti tohoto algoritmu.

[2]Detekce viditelných stěn bude využita i v algoritmu *QuickHull*, kde byla implementována a bude tedy popsána v sekci 3.3.

3.3 Inkrementální algoritmus

Další z algoritmů, který je rozšířením své 2D verze [5, str. 88–91]. Je občas též nazývaný metodou *beneath–beyond*, pokud se jedná o jeho rozšíření do více dimenzí. Byl prvně představen v roce 1984 Michaelem Kalleyem [27] a jeho základní varianta dosahuje složitosti $\mathcal{O}(nh)$.

3.3.1 Popis algoritmu

Tento algoritmus spočívá v postupném přidávání bodů do částečně hotové obálky. V každém kroku přidáváme jeden bod. Nechť H_i je obálka tvořená v i–tém kroku a p_i bod, který v tomto kroce přidáváme, potom $H_i \leftarrow H_{i-1} \cup p_i$. V každém kroku mohou nastat dvě situace – buď je přidávaný bod p_i uvnitř mnohostěnu H_{i-1} nebo mimo něj. V prvním případě bod p_i nemusíme dále zpracovávat, v druhém případě je třeba vytvořit novou obálku, ve které bude p_i jedním z vrcholů.

Zda bod leží uvnitř mnohostěnu nebo mimo něj, je možné zjistit stejným způsobem jako u 2D verze algoritmu.

Věta 4. *Bod p leží uvnitř mnohostěnu M, pokud se nachází na kladné straně každé roviny určenými stěnami mnohostěnu M.*

Pro to, abychom zjistili, na jaké straně roviny bod leží, využijeme vzorce pro objem čtyřstěnu, vyjádřeného pomocí determinantu.

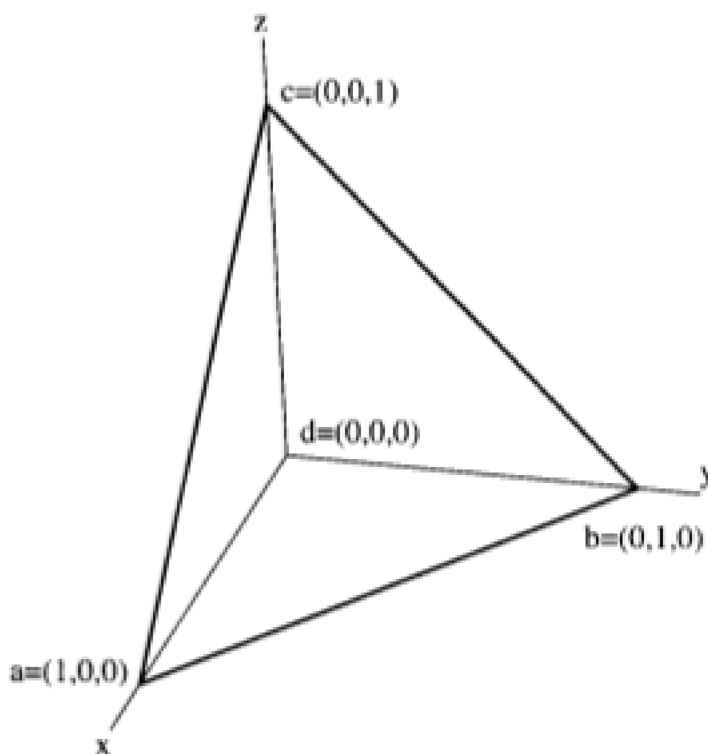
Lemma 1. Nechť a,b,c,d jsou body v prostoru $\mathbb{E}^3$. Potom pro objem $\mathcal{V}$ čtyřstěnu tvořeného těmito body platí

$$6\mathcal{V} = \begin{vmatrix} a_0 & a_1 & a_2 & 1 \\ b_0 & b_1 & b_2 & 1 \\ c_0 & c_1 & c_2 & 1 \\ d_0 & d_1 & d_2 & 1 \end{vmatrix}$$

Za předpokladu, že všechny stěny hledané konvexní obálky budeme reprezentovat body uspařádanými proti směru hodinových ručiček, můžeme tvrdit následující.

Lemma 2. Nechť R je rovina v $\mathbb{E}^3$ určená body a,b,c uspořádanými proti směru hodinových ručiček a d bod ležící mimo tuto rovinu. Potom objem $\mathcal{V}$ čtyřstěnu tvořeného body a,b,c,d je kladný, pokud body a,b,c jsou uspořádány podle směru hodinových ručiček při pohledu z bodu d a záporný naopak.

Důsledek 2. Nechť n je normálový vektor roviny R, který má směr podle pravidla pravé ruky. Rovina R dělí prostor $\mathbb{R}^3$ na dva poloprostory P^+ a P^-. Řekněme, že n směřuje do poloprostoru P^+. Potom je objem $\mathcal{V}$ kladný, pokud bod d leží v P^- a záporný, pokud leží v P^+.



Obrázek 3.3: Čtyřstěn pro výpočet objemu

Převzato z [5]

Tyto lemmata a důsledek můžeme též najít v [5, str. 22-23]

Důsledek předchozí věty si můžeme ukázat na příkladu na obrázku 3.3.

Vidíme, že body a,b,c jsou při pohledu z bodu d uspořádané po směru hodinových ručiček. Objem čtyřstěnu a,b,c,d vychází

$$6\mathcal{V} = \begin{vmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{vmatrix} = -0\cdot1\cdot0+0\cdot0\cdot0+0\cdot0\cdot0-1\cdot0\cdot0-0\cdot0\cdot1+1\cdot1\cdot1 = 1$$

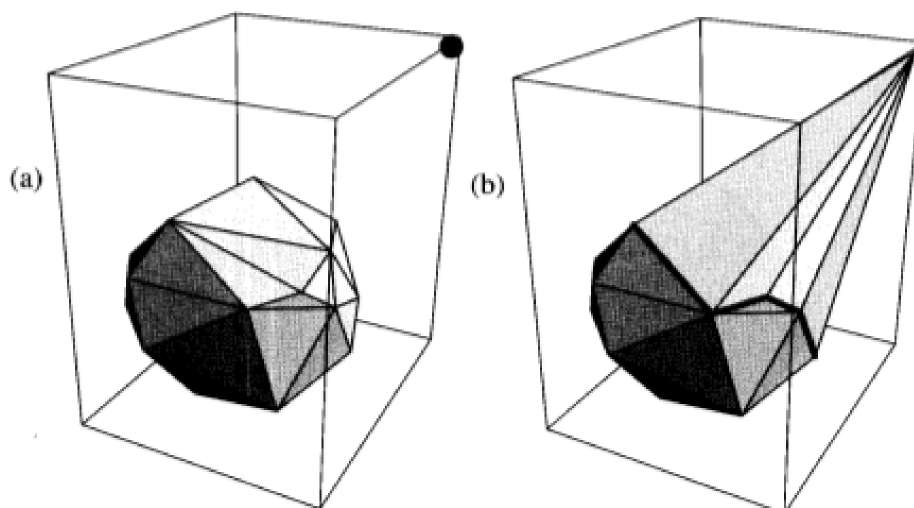
Normálový vektor roviny určené body a,b,c směřuje směrem k nám, bod d leží tedy v poloprostoru P^-.

Zda bod leží uvnitř mnohostěnu můžeme tedy zjistit tak, že vytvoříme čtyřstěn z každé stěny a přidávaného bodu p a spočítáme jeho objem. Pokud budou všechny tyto objemy kladné, můžeme říci, že bod leží uvnitř mnohostěnu.

Jak bylo řečeno, pokud bod p leží uvnitř obálky, můžeme ho přeskočit a pokračovat na další iteraci. Pokud bod leží mimo, je třeba obálku upravit,

a to přidáním nových stěn a odstraněním těch stěn ze staré obálky, které budou ležet uvnitř nové. To, které stěny budou ležet uvnitř, můžeme zjistit jednoduše – budou to stěny, které jsou viditelné z bodu p. Opět platí, že stěna f je viditelná z bodu p tehdy a pouze tehdy, když p leží v kladném poloprostoru, určeným rovinou f. f je tedy viditelná, pokud čtyřstěn tvořený stěnou f a bodem p má záporný objem.

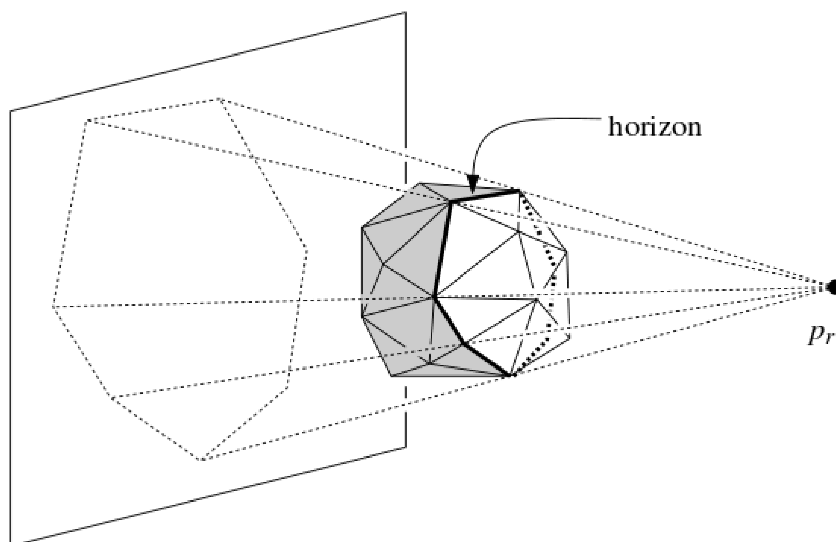
Pro každou stěnu si tedy určíme, zda je viditelná z bodu p. Pokud není viditelná ani jedna stěna, bod p leží uvnitř obálky a přejdeme na další iteraci. Pokud ne, je potřeba vytvořit nové stěny a smazat stěny, které by byly uvnitř nové obálky. Jak tyto stěny poznat již víme, zbývá zjistit, kam napojit stěny nové. Jedním z vrcholů každé nové stěny bude bod p. Zbylé dva body každé nové stěny budou tvořit koncové body hran incidujících s jednou stěnou viditelnou z bodu p a jednou stěnou, která z bodu p vidět není. Tyto hrany budou tvořit takzvaný *horizont* (obrázek 3.5).



Obrázek 3.4: (a) Konvexní obálka H_{i-1} před přidáním bodu v rohu (b) Konvexní obálka H_i

Převzato z [5]

Pseudokód inkrementálního algoritmu nalezenem níže:



Obrázek 3.5: Horizont a viditelné stěny (bíle)

Převzato z [11]

Algoritmus 2: Inkrementální algoritmus

$T \leftarrow findInitialTetrahedron()$ {points p_1, p_2, p_3, p_4}
$H \leftarrow H \cup faces(T)$
for $i = 4, \ldots, n$ **do**
  for all face $f \in H$ **do**
    $V \leftarrow volume(f, p_i)$
    if $V < 0$ **then**
      $markVisible(f)$
    end if
  end for
  if any faces visible **then**
    $h \leftarrow findHorizon(H, p_i)$
    for all edge $e \in h$ **do**
      $f \leftarrow createFace(e, p_i)$
      $H \leftarrow H \cup \{f\}$
    end for
    for all $f \in visible(p_i)$ **do**
      $H \leftarrow H - \{f\}$
    end for
  end if
end for

3.3.2 Analýza složitosti

Věta 5. *Nechť vstupní množina bodů M obsahuje n bodů. Potom složitost inkrementálního algoritmu pro nalezení konvexní obálky těchto bodů je $\mathcal{O}(nh)$.*

Důkaz. Za použití důsledku 1 víme, že iterace přes všechny stěny a hrany jsme schopni uskutečnit v čase $\mathcal{O}(h)$. V každém kroku algoritmu přidáváme jeden bod, tedy dohromady n–krát musíme iterovat přes všechny stěny a hrany. Celková složitost je tedy $\mathcal{O}(nh)$. □

V sekci 4.3 ukážeme, že za použití vhodných struktur jsme schopni dosáhnout složitosti nižší.

3.4 QuickHull

QuickHull je opět algoritmem, který je rozšířením své 2D verze. Podobně jako *inkrementální algoritmus* přidává v každé iteraci jeden bod do výsledné obálky. Tento algoritmus byl poprvé představen v roce 1995 [28].

3.4.1 Popis algoritmu

Na začátku algoritmu je potřeba nalézt co největší čtyřstěn, jehož hraniční body budou ležet na výsledné obálce. Abychom toho docílili, musíme nalézt 6 extrémních bodů, a to bodů s maximální a minimální hodnotou na souřadnicích *x,y,z*. Z těchto šesti bodů vybereme dva, které jsou od sebe nejdále, tyto dva body bodou tvořit první hranu hledaného čtyřstěnu, nazvěme si je *a,b*. Dalším bodem čtyřstěnu bude bod, který leží nejdéle od úsečky $\overline{ab}$. Tento bod nazvěme *c*. Posledním hledaným bodem bude bod nejvzdálenější od roviny určené body *a,b,c*. Nyní z těchto čtyř bodů můžeme sestrojit čtyřstěn, který nám poslouží pro další práci algoritmu, nazvěme ho M.

Nyní budou všechny body přiřazeny nějaké stěně z M, vytvoříme *seznam konfliktů* – pro každou stěnu si vytvoříme seznam bodů, ze kterých je tato stěna viditelná. Stejně tak pro všechny body si vytvoříme seznam stěn, které jsou z něj viditelné. Body, ze kterých není viditelná žádná stěna (body uvnitř čtyřstěnu) můžeme již zanedbat a nepracovat s nimi dále. Všechny čtyři stěny čtyřstěnu si uložíme na zásobník a pokračujeme do další fáze algoritmu.

Ve druhé fázi algoritmu vždy vyjmeme jednu stěnu ze zásobníku (nazvěme ji F) a najdeme k ní nejvzdálenější bod z jejího *seznamu konfliktů*, nazvěme ho e. Další průběh bude podobný jako v předchozím algoritmu. Opět musíme najít všechny stěny, které jsou viditelné z bodu e. Tentokrát ale stačí prohledávat pouze stěny sousedící se stěnou F. Dále opět musíme najít *horizont* a vytvořit nové stěny spojující *horizont* a bod e. Tento krok je stejný jako v předchozím algoritmu a nebudu ho popisovat do detailu podruhé .

Nakonec je potřeba aktualizovat *seznam konfliktů*, protože se nyní na naší obálce vyskytují nové stěny. Zároveň odstraníme původní stěny, jež byly viditelné z bodu e, a to jak z obálky, tak i ze zásobníku. Nakonec na zásobník přidáme nové stěny a pokračujeme, dokud zásobník nebude prázdný.

Níže nalezneme pseudokód *QuickHullu*:

Algoritmus 3: QuickHull

$T \leftarrow findInitialTetrahedron()$
$H \leftarrow H \cup faces(T)$
for all face $f \in H$ **do**
  $assignPointsToFace(f)$
end for
$Q \leftarrow \{f_1, f_2, f_3, f_4\}$
while $Q \neq \emptyset$ **do**
  $f \leftarrow Q.pop_back()$
  $p \leftarrow mostDistantPoint(f)$
  $h \leftarrow findHorizon(p)$
  $newFaces \leftarrow \emptyset$
  for all edge $e \in h$ **do**
    $f_n \leftarrow createFace(e, p)$
    $h \leftarrow H \cup f_n$
    $newFaces \leftarrow newFaces \cup f_n$
  end for
  for all $f \in visible(p)$ **do**
    $H \leftarrow H - \{f\}$
  end for
  for all face $g \in newFaces$ **do**
    $assignPointsToFace(g)$
  end for
  for all face $g \in newFaces$ **do**
    $Q.push_back(g)$
  end for
end while

3.4.2 Analýza složitosti

Složitost algoritmu v prostoru $\mathbb{R}^3$ závisí, stejně jako například u algoritmu *QuickSort* [29], na pořadí přidávaných bodů. Stejně jako u něj může dosahovat složitosti $\mathcal{O}(n^2)$ v nejhorším případě, ale v průměrném případě dosahuje složitosti $\mathcal{O}(n \log n)$. Toto závisí na tom, zda je průběh algoritmu *vybalancovaný*. Definici a důkaz složitosti můžeme nalézt v původní práci Barbera a Dobkina [28].

3.5 Další algoritmy

Kromě výše popsaných algoritmů jsem našel ještě jeden, založený na algoritmu *Divide and Conquer* a jedná se o jeho minimalistickou verzi vytvořenou Timothy M. Chanem jako studijní materiál pro *University of Waterloo*. Přestože je tento algoritmus implementovaný na pouze přibližně 100 řádek kódu, není triviální a jeho popis je možné nalézt v [30].

Kapitola **4**

Optimalizace

V této kapitole budou popsány možné optimalizace algoritmů z kapitoly 3 a analyzována možnost jejich paralelizace.

4.1 Obecné optimalizace

Mezi obecné optimalizace, které můžeme použít u všech algoritmů, patří eliminace zbytečných matematických operací. Pro příklad je možné u geometrických vzorců často eliminovat výpočet odmocniny, například při výpočtu vzdálenosti dvou bodů a a b ($|ab| = \sqrt{a_x - {b_x}^2 + a_y - {b_y}^2 + a_z - {b_z}^2}$) nebo při výpočtu velikosti vektoru v ($|v| = \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2}$).

Pokud nepotřebujeme znát přesnou hodnotu, ale pouze porovnáváme hodnoty mezi sebou, je možné odmocniny vynechat a porovnávat hodnoty umocněné na druhou. Důvod pro tuto optimalizaci je náročnost operace odmocnění. Podobnou optimalizací je vynechání výpočtu velikosti úhlu přes hodnotu funkce kosinus, ale porovnávání samotných kosinů. Tato optimalizace bude popsána dále u algoritmu *Jarvis March*.

Kromě optimalizace matematických operací byly algoritmy optimalizované na úrovni překladače zapnutím optimalizací při překladu. Využito bylo dvou:

- `-O3`: Zapne veškeré možné optimalizace na úrovni překladače, mezi které patří například *loop unrolling* nebo vektorizace cyklů.
- `-ffast-math`: Povoluje optimalizace výpočtů při počítání s plovoucí desetinnou čárkou na úkor možného způsobení malých nepřesností výpočtu.

Ukázalo se, že přepínač `-O3` byl asi největší optimalizací mojí implementace, když například algoritmus *Jarvis march* zrychlil téměř desetkrát u velkých vstupů. To bude ukázáno v kapitole 5 spolu s porovnáním jednotlivých algoritmů.

4.2 Jarvis March

Stejně jako u 2D verze je vždy, když hledáme nový bod obálky, potřeba najít bod, aby nová stěna se stěnou zpracovávanou svírala co největší úhel. Platí, že úhel, který svírají dvě roviny, je stejný jako úhel, který svírají jejich normálové vektory. Dále platí, že úhel α mezi dvěma vektory n a m můžeme spočítat takto:

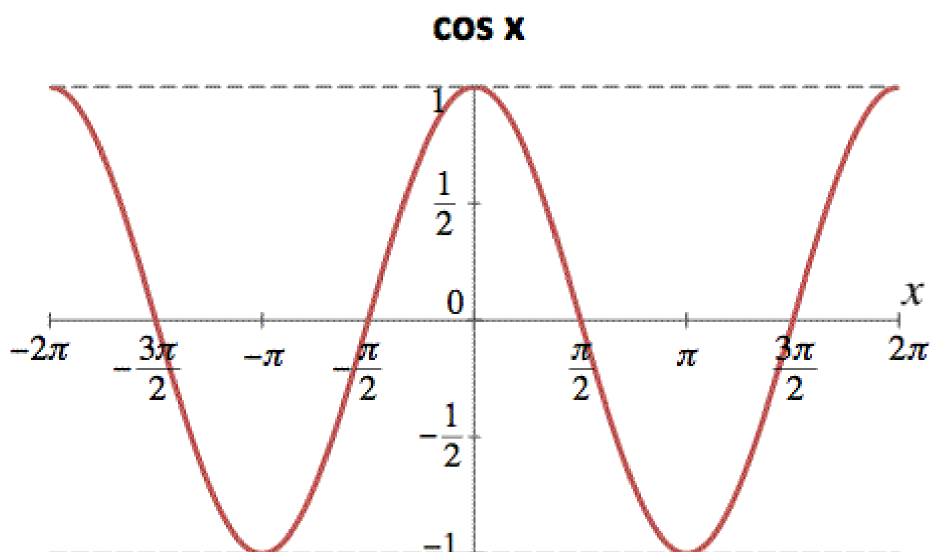
$$\cos \alpha = \frac{|n \cdot m|}{|n| \cdot |m|}$$

a tedy

$$\alpha = \arccos \frac{|n \cdot m|}{|n| \cdot |m|}$$

Jelikož je nutné projít vždy všechny zbývající body, bylo by nutné v každém kroku počítat $\mathcal{O}(n)$-krát funkci arccos. Tato funkce může zabírat až několik desítek cyklů procesoru, proto tento přístup není efektivní.

Při použití postupu uvedeného výše ovšem není potřeba počítat přesný úhel α, postačí nám znát hodnotu $\cos \alpha$. Na obrázku 4.1 si připomeňme graf funkce kosinus.



Obrázek 4.1: Funkce kosinus

Můžeme vidět, že na intervalu $[0, \pi]$ je funkce klesající. Máme jistotu, že hledaný bod bude ležet právě v tomto intervalu, protože pokud by ležel mimo něj, obálka by nebyla konvexní. Proto stačí porovnávat pouze vypočítané hodnoty funkce kosinus – čím bude menší, tím větší bude úhel mezi rovinami.

Přesto nakonec ani tento postup nebyl v mé implementaci použit a přestože se jeví jako nejjednodušší, od začátku jsem algoritmus implementoval tak, jak je popsáno v kapitole 3, tedy že pro nalezený bod musí všechny ostatní

ležet na dané straně roviny určené tímto bodem a hranou, kterou aktuálně zpracováváme. Hlavním důvodem bylo, že již od začátku jsem vycházel ze zdroje [10], kde je takto algoritmus popsán.

4.3 Inkrementální algoritmus

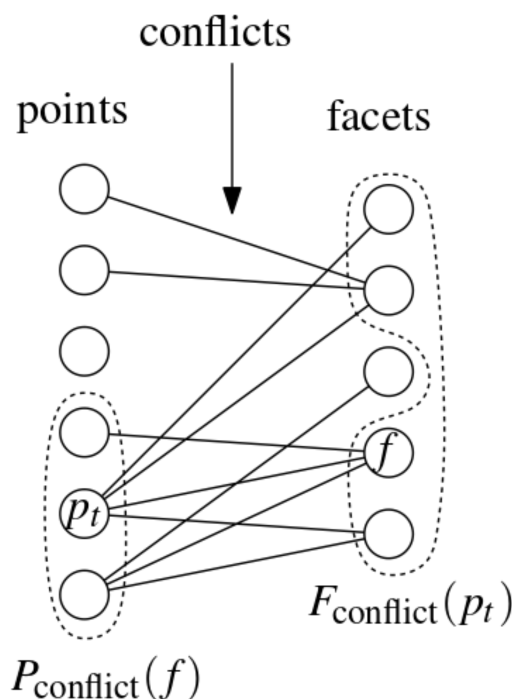
U inkrementálního algoritmu jsme si v kapitole 3 ukázali, že jeho složitost je $\mathcal{O}(nh)$. Za použití vhodných datových struktur se ale tento algoritmus dá výrazně urychlit. Tento postup se nazývá *Randomizovaný inkrementální algoritmus* a byl představen v roce 1993 Raimundem Seidelem [31]. Kromě využití vhodných struktur se zakládá na vytvoření náhodné permutace zbylých bodů po vytvoření prvního čtyřstěnu na začátku algoritmu.

V každém kroku algoritmu je potřeba zkontrolovat, které stěny jsou viditelné z nově přidávaného bodu. To znamená zkontrolování $\mathcal{O}(h)$ stěn v každém kroku. Ten ovšem můžeme urychlit uchávááváním dodatečných informací.

Budeme si uchovávat informace o tom, která stěna je viditelná z kterého bodu ve formě bipartitního grafu, kdy na jedné straně budou body, které jsme ještě nezpracovali a na straně druhé budou stěny aktuální obálky. Nechť H_i je obálka vytvořená v kroce i přidáním bodu p_i. Potom pro každou jeji stěnu f uchováváme množinu $P_{conflict}(f) \subseteq \{p_{i+1}, p_{i+2}, \ldots, p_n\}$ obsahující body, ze kterých je f viditelná. Dále pro všechny body p_t, kde $t > i$ uchováváme množinu $F_{conflict}(p_t)$ obsahující stěny H_i, které jsou z bodu p_t vidět. Tento graf nazvěme *grafem konfliktů*. Dále bod $p \in P_{conflict}(f)$ budeme nazývat, že je *v konfliktu* se stěnou f. Tento název je odvozen od faktu, že v jedné konvexní obálce nemůže být zároveň bod p vrcholem a f stěnou. Množiny $P_{conflict}(f)$ a $F_{conflict}(p_t)$ budeme nazývat *seznamy konfliktů*.

Na obrázku 4.2 ilustrujme ukázku grafu konfliktů. Můžeme vidět, že hrany v tomto grafu spojují uzly pro jednotlivé body a stěny, které jsou v konfliktu. Jinými slovy, je zde hrana mezi uzlem pro bod $p_t \in P$ a pro stěnu $f \in H_i$, jestliže $i < t$ a f je viditelná z p_t. Při využití tohoto grafu konfliktů můžeme pro daný bod p_t množinu $F_{conflict}(p_t)$ nalézt v lineárním čase, stejně tak pro stěnu f můžeme v lineárním čase nalézt množinu $P_{conflict}(f)$. Hlavní výhodou tohoto přístupu je, že ve chvíli, kdy budeme přidávat bod p_i ke konvexní obálce H_{i-1}, nemusíme kontrolovat všechny stěny, zda jsou z tohoto bodu viditelné – stačí najít $F_{conflict}(p_t)$ v grafu konfliktů. To u velkých obálek může přinést výrazné urychlení, za cenu režie spojené s aktualizací grafu konfliktů.

Na začátku algoritmu po vytvoření prvního čtyřstěnu inicializujeme graf konfliktů, to lze udělat v lineárním čase průchodem všech zbylých bodů a nalezením stěn, které jsou z nich viditelné. V každém dalším kroku je potřeba graf pozměnit. Při přidávání bodu p_i do obálky odstraníme z grafu všechny uzly představující stěny viditelné z p_i a hrany s nimi incidující. Dále odstraníme i uzel pro bod p_i a přidáme uzly pro nové stěny propojující p_i a jeho horizont.



Obrázek 4.2: Graf konfliktů

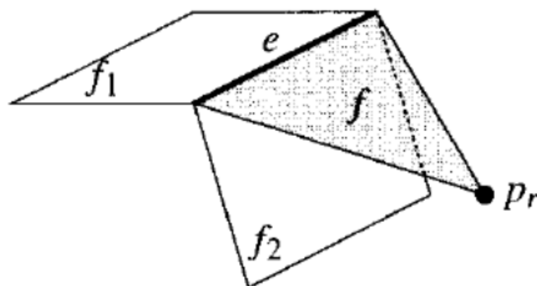
Převzato z [11]

Klíčový krok je nalezení seznamu konfliktů pro tyto nové stěny. Využijeme přitom následující věty.

Věta 6. *Při přidávání nového bodu p_i je pro aktualizaci grafu konfliktů potřeba prohledat pouze body, které vidí stěny incidující s hranami tvořícími horizont bodu p_i.*

Důkaz. Předpokládejme, že nově vytvořená stěna f je viditelná z bodu $p_t, t > i$. Potom je z tohoto bodu viditelná i hrana e tvořící stěnu f a ležící naproti bodu p_i. Tato hrana je součástí horizontu bodu p_i a byla součástí konvexní obálky H_{i-1}. Jelikož $H_{i-1} \subset H_i$, musela být hrana e z bodu p_t viditelná už v kroku $i-1$. To je možné pouze v případě, pokud jedna ze dvou stěn incidujících s e v H_{i-1} byla viditelná z bodu p_t. □

Toto tvrzení můžeme ilustrovat na obrázku 4.3. Pokud si stěny incidující s e označíme f_1 a f_2, stačí tedy zkontrolovat body náležící do $P_{conflict}(f_1)$ a $P_{conflict}(f_2)$. Ještě zbývá dodat, že pokud je stěna f koplanární se stěnou f_1, má stejný seznam konfliktů jako f_1, a to samé platí i pro f_2.



Obrázek 4.3: Přidání nové stěny

Převzato z [11]

4.3.1 Složitost algoritmu

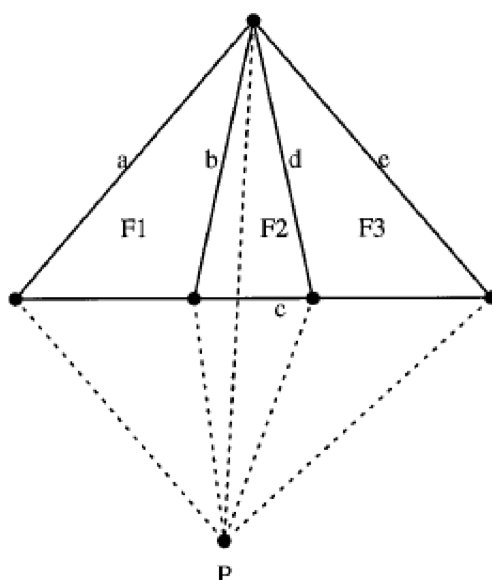
Složitost takto upraveného algoritmu bude předpokládaná $\mathcal{O}(n \log n)$. Složitost je pouze předpokládaná proto, že závisí na náhodné permutaci bodů, vytvořené na začátku algoritmu. Stejně jako například *QuickSort* má složitost $\mathcal{O}(n^2)$, jeho předpokládaná složitost v průměrném případě se uvádí také $\mathcal{O}(n \log n)$ a je závislá na permutaci prvků ve vstupní množině. Důkaz složitosti randomizovaného inkrementálního algoritmu je netriviální, a proto ho nebudu uvádět. Je možné ho najít například v [11, str. 250–252].

4.4 QuickHull

Hlavní optimalizací, jež bylo třeba udělat a která je zároveň potřebná, aby algoritmus fungoval správně pro obecný vstup, je slučování stěn. Pokud by tato optimalizace nebyla implementována, algoritmus by mohl pro vstup, kde jsou 4 a více koplanárních bodů, produkovat špatný výstup. Problém spočívá v numerické nepřesnosti výpočtů při počítání s desetinnými čísly ve formátu plovoucí desetinné čárky [32]. Tyto výpočty s sebou přinášejí malé nepřesnosti, které v algoritmu mohou zapříčinit jeho nesprávnou funkčnost. Jeden z problémů, který může nastat, byl popsán již v původní práci Barbera a Dobkina [28] a zároveň s ním i jeho řešení, spočívající právě ve slučování stěn.

Problém si ukážeme na obrázku 4.4. Z důvodu nepřesnosti se může stát, že stěny $F1$ a $F3$ jsou z bodu P viditelné, zatímco $F2$ nikoliv (bod P leží nad $F1$ a $F3$, ale pod $F2$). Algoritmus nahradí stěny $F1$ a $F3$ novými, zatímco $F2$ zůstane na konvexní obálce, přestože tam být nemá.

Řešení tohoto problému spočívá v kontrole obálky na konci každé iterace – pokud nalezneme stěny, které nejsou konvexní, sloučíme je. Pojmenujme si nové stěny podle hran, které by tvořili horizont bodu P, tedy $a - e$. Stěny a, b, d, e budou spolu sdílet jednu hranu. Jelikož víme, že každá hrana musí



Obrázek 4.4: Chybné stěny před sloučením

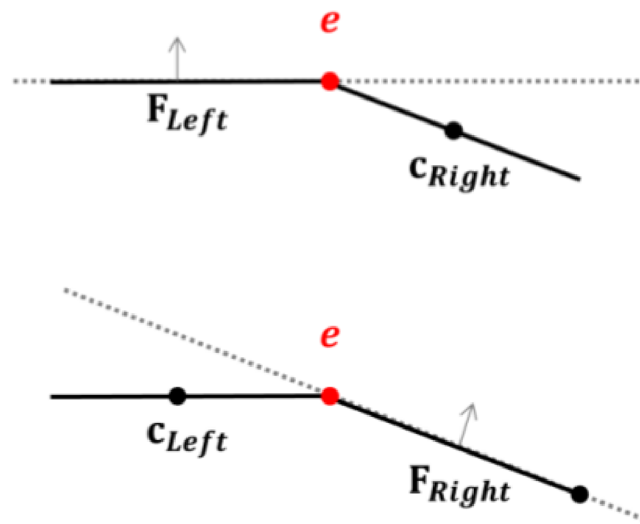
Převzato z [28]

incidovat právě se dvěmi stěnami, je třeba toto opravit. *QuickHull* tuto chybu opraví nejdříve sloučením dvou nejbližších stěn – řekněme b a d. Vrcholy stěn $F2$ a c jsou obsaženy ve sloučené stěně bd, a tak je do ní sloučíme také. Vznikne stěna $bcd2$, ta bude sousedit se dvěmi stěnami (je důležité si uvědomit, že stěny $F1$ a $F3$ jsme již odstranili), a proto ji opět sloučíme s bližší z nich, řekněme e. Pokud následná stěna $bcde2$ bude konvexní se stěnou a, můžeme skončit.

Zda jsou dvě stěny konvexní lze zkontrolovat jednoduchým testem, ukázaným na obrázku 4.5. Řekněme, že máme stěny F_{Left} a F_{Right}, které mají společnou hranu e. Pro obě tyto stěny nalezneme jejich *centroidy* jako průměr všech jejich vrcholů, nazvěme je c_{Left} a c_{Right}. Potom tyto stěny budou konvexní, pokud bude centroid c_{Left} ležet pod rovinou stěny F_{Right} a centroid c_{Right} pod rovinou stěny F_{Left}.

4.5 Fat Planes

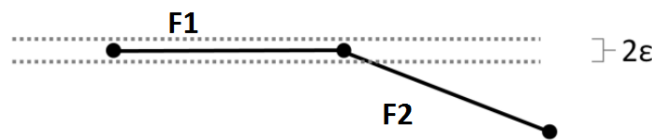
Jak již bylo řečeno u popisu optimalizace algoritmu *QuickHull* (4.4), nepřesnost výpočtů v plovoucí desetinné čárce s sebou přináší problémy, které mohou narušit správnost nalezené konvexní obálky. Jeden z problémů, se kterým se setkáváme, je nesprávné určení bodů, které jsou koplanární s nějakou rovinou, případne dvou koplanárních rovin. Bod p koplanární s rovinou f může být chybně určen, jako že leží nad a nebo pod ní pouze kvůli nepřesnosti výpočtu. Z tohoto důvodu se při tvorbě konvexní obálky používají místo stěn takzvané



Obrázek 4.5: Test konvexnosti stěn F_{Left} a F_{Right}

Převzato z [22]

fat planes, neboli *tlusté stěny*, aby se tomuto problému do co největší míry zamezilo. Řešení spočívá v použití malé odchylky ε tak, že pokud bod bude ležet nad a nebo pod rovinou do vzdálenosti ε, budeme ho brát jako s touto stěnou koplanární. Ilustrujme si to na obrázku 4.6.



Obrázek 4.6: Fat planes

Převzato z [22]

Nechť $F1$ je rovina, jejíž normálový vektor směřuje směrem nahoru, bod P ležící v prostoru a s vzdálenost bodu P od roviny $F1$. Potom:

$$\begin{cases} s \ > \varepsilon & P \text{ leží nad rovinou } F1 \\ s \ < -\varepsilon & P \text{ leží pod rovinou } F1 \\ -\varepsilon \leq s \ \leq \varepsilon & P \text{ je koplanární s } F1 \end{cases}$$

Otázkou přirozeně zůstává, jakou zvolit hodnotu ε. Kolega Mitura ve své práci [10] experimentálně našel hodnotu $1 \cdot 10^{-6}$, tato hodnota ovšem nebere v úvahu vstupní data. V práci [28] o *QuickHullu* je uveden vzorec, který tento nedostatek odstraňuje a pracuje s maximálními absolutními hodnotami souřadnic vstupních bodů a strojovou odchylkou datového typu *double*:

$$\varepsilon = 3\left(max\left(|x|\right) + max\left(|y|\right) + max\left(|z|\right)\right) \cdot double_prec$$

4.6 Možnosti paralelizace

V této sekci zanalyzujeme možnosti, jak jednotlivé algoritmy paralelizovat, případně uvedeme důvod, proč paralelizace není možná.

4.6.1 Jarvis march

Na začátku algoritmu *Jarvis march* hledáme nějaký extrémní bod, u kterého budeme mít jistotu, že bude ležet na obálce. Těchto bodů je maximálně šest, jedná se o body s maximální a minimální souřadnicí v každé dimenzi. Potom pro každý extrémní bod je možné vytvořit vlastní vlákno, které bude obálku tvořit. Teoreticky je možné dosáhnout až lineárního zrychlení, pokud každé vlákno vytvoří stejně velkou část výsledné obálky. Důležité by bylo analyzovat kritické sekce, mezi které by patřil například přístup k množinám *fresh* a *closed* z důvodu, že by jedno vlákno mohlo přidat hranu do množiny *closed* zatímco jiné by ji tam hledalo a nemuselo by ji najít. Tímto by mohli vznikat některé stěny vícekrát.

4.6.2 Inkrementální algoritmus

V každém kroku tohoto algoritmu je přidáván do obálky jeden bod, logicky by se tedy nabízela možnost v každém kroku přidávat paralelně více bodů najednou. Problém tohoto přístupu je v tom, že každý přidávaný bod může obálku modifikovat (a nebude pouze v případě, že bude ležet uvnitř ní). Z tohoto důvodu by bylo nutné modifikaci obálky (odstranění nepotřebných stěn a přidání nových) označit za kritickou sekci. To by zřejmě výrazné urychlení nepřineslo. Přesto by šlo paralalezivat některé části algoritmu, jako například výpočet viditelných stěn. Pro aktuálně zpracovávaný bod p by každé vlákno dostalo část stěn aktuální obálky a vypočítalo by, zda jsou z bodu p viditelné či nikoliv. Další možností, jak algoritmus paralelizovat je uchovávat si jedno speciální vlákno pro kontrolu bodů uvnitř obálky. Vlákno by běželo nezávisle na vláknu hlavním a jak by se postupně konstruovala obálka, stále by kontrolovalo, zda některý z ještě nevyužitých bodů neleží uvnitř ní. Pokud by nějaký takový našlo, označilo by ho pro hlavní vlákno jako nepotřebný a to by tento bod nemuselo již zpracovávat. S tímto přístupem by nebylo potřeba tvořit velké kritické sekce, protože toto vlákno by nemohlo tvořit falešně pozitivní

výsledky (bod označený jako ležící uvnitř a přitom ležící mimo), a to z důvodu, že obálka se každým krokem zvětšuje, nemůže tedy nastat, že v kroku i bude bod ležet uvnitř obálky a v kroku $t > i$ mimo ni. Mohlo by se stát, že pokud by se zrovna obálka modifikovala, tak by bod ležící uvnitř mohl identifikovat jako ležící mimo, to by ale nevadilo, protože by byl odhalen v dalším průchodu, případně hlavním vláknem.

4.6.3 QuickHull

Možnosti paralelizace jsou podobné jako u algoritmu inkrementálního. V průběhu *QuickHullu* je vždy přidáván jeden bod, protože vždy vyjímáme ze zásobníku jednu stěnu a k ní hledáme nejvzdálenější bod. Bylo by tedy možné zpracovávat v oddělených vláknech vždy jednu stěnu ze zásobníku, ale stejně jako v předchozím případě přidáním bodu modifikujeme obálku, takže by byla nutná velká spolupráce mezi vlákny, proto tento postup nevidím jako užitečný. Opět bych viděl hlavní možnost paralelizace v hledání viditelných stěn pro aktuálně přidávaný bod, kdy by každé vlákno zpracovávalo část stěn a pro ně určilo, zda jsou viditelné či nikoliv.

Kapitola 5

Výsledky

V této kapitole budou shrnuty výsledky mé práce, budou zde ukázány časy běhu jednotlivých algoritmů na různých typech dat. Následně budou porovnány jednotlivé algoritmy mezi sebou a budou srovnány s knihovnou *QHull* [19].

Testování probíhalo na fakultním serveru `STAR` s následující specifikací:

- **CPU**: 2x Intel® Xeon® Procesor E5-2620 v2 (15 MB Cache, 2.10 GHz, 6 fyzických jader, 12 logických jader s technologií Hyper-Threading)
- **RAM**: 32 GB

Kompilace byla provedena překladačem GCC [33]. Pro dosažení co nejlepších časů byl kód kompilován s optimalizacemi na úrovni kompilátoru, uvedenými v sekci 4.1.

Před samotným testováním bylo ještě potřeba otestovat, zda naše algoritmy pracují správně. U implementovaných algoritmů byly tedy vytvořeny metody pro testování jejich správnosti. Tyto testy pracují s odchylkou ε definovanou v sekci 4.5. Po vytvoření obálky se provádí kontrola její konvexity. Všechny stěny musí být konvexní, neboli nesmí obsahovat žádnou nekonvexní hranu. Konvexita stěn a hran byla definována u algoritmu *QuickHull* 4.4, se započítáním právě odchylky ε (pokud centroid jedné stěny leží nad rovinou druhé do vzálenosti menší než ε, jsou tyto stěny stále považovány za konvexní).

Samotný tento test by nebyl dostačující, protože by mohl vytovřit obálku menší, než jaká má ve skutečnosti být. Je třeba ještě otestovat, zda nějaký bod neleží mimo obálku. Proto je pro každou stěnu třeba zkontrolovat všechny body. Bod je považován mimo obálku, pokud jeho vzdálenost k nějaké stěně je větší než 10ε. Tato tolerance je převzatá z práce [28] o *QuickHullu*, kde autoři i podobnou metodiku testování uvádějí.

Posledním, ale možná nepotřebným testem je ještě kontrola Eulerovy rovnice 1.1 a jejího důsledku 2.

5.1 Testovací data

V rámci testování jsem používal čtyři druhy testovacích dat. Ty byly generovány pomocí nástroje `rbox` z knihovny *QHull* [19]. Tato testovací data byla generována příkazem `rbox N n [s] [Bx] [Wy]`, kde

- `N` je počet generovaných bodů
- `s` generuje body v kouli, jinak v krychli
- `Bx` je bounding-box; budou generovány body se souřadnicemi v intervalu $[-x, x]$
- `Wy` body budou distribuovány ve 100`y`% od povrchu tělesa. Například pro `B0` budou body ležet na povrchu tělesa, pro `B1` budou kdekoliv uvnitř nebo na povrchu tělesa

Generoval jsem tyto testovací sady dat:

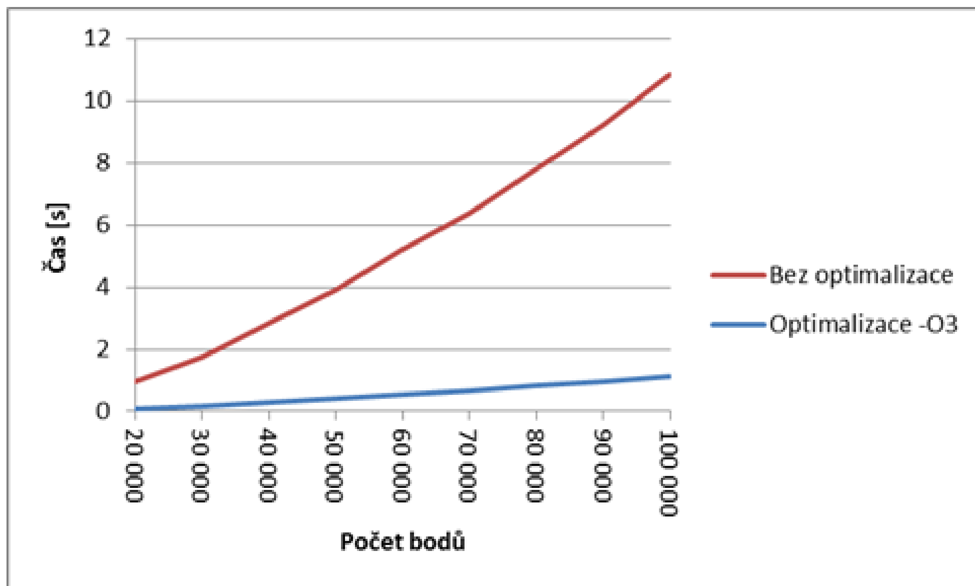
- **Testovací data č. 1**: Body ležící kdekoliv uvnitř krychle
- **Testovací data č. 2**: Body ležící kdekoliv uvnitř koule
- **Testovací data č. 3**: Body ležící do 10% od povrchu krychle
- **Testovací data č. 4**: Body ležící na povrchu koule

5.2 Jednotlivé algoritmy

Nyní budou ukázány časy běhu jednotlivých algoritmů. Ze čtyř typů generovaných dat jsem si pro tento účel vybral data generované uvnitř koule, protože podle mého názoru mají z těchto čtyř typů nejobecnější charakter.

Jelikož byla u každého algoritmu implementována pouze jedna jeho verze, nemůžeme mezi sebou jednotlivé verze srovnat. Můžeme ale ukázat vliv kompilátorových optimalizací na jednotlivé algoritmy. Při kompilaci byl využit přepínač `-O3`, který zapne všechny dostupné kompilátorové optimalizace. U algoritmu *Jarvis march* je zrychlení největší, a to téměř desetinásobné, jak ilustruje obrázek 5.1. U zbylých algoritmů toto zrychlení již není tak výrazné. Kromě přepínače `-O3` byl použit ještě přepínač `-ffast-math`, jeho použití – respektive nepoužití ale žádné zrychlení – respektive zpomalení, nepřineslo.

V dalším textu budou uváděny měřené časy za použití obou těchto přepínačů.



Obrázek 5.1: Kompilátorová optimalizace algoritmu Jarvis march

5.2.1 Jarvis March

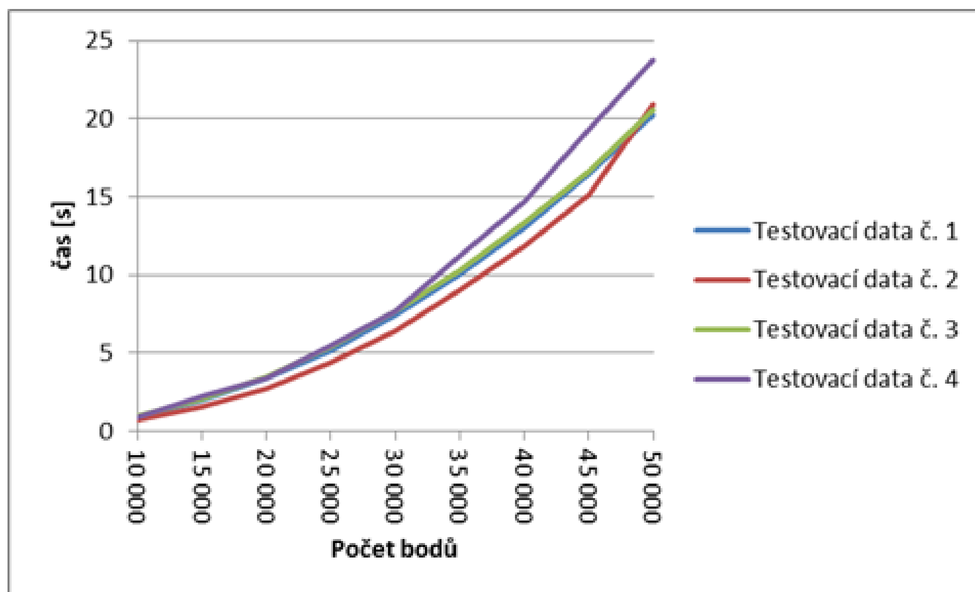
Jak bylo řečeno, v mé implementaci jsem použil verzi algoritmu popsanou v sekci 4.2. Čas běhu algoritmu v závislosti na velikosti vstupní množiny bodů ilustruje tabulka 5.1. Bohužel pro každou sadu generovaných dat se liší počet bodů tvořících výslednou obálku, jelikož `rbox` toto nastavení nemá. Přesto můžeme vidět, že algoritmus odpovídá své předpokládané asymptotické složitosti.

5.2.2 Inkrementální algoritmus

U inkrementálního algoritmu byla implementována jeho základní verze popsaná v sekci 3.3. Z toho vyplývá, že tento algoritmus bude kvůli své složitosti nejpomalejší ze všech tří algoritmů, které jsem testoval. To dokazuje i obrázek 5.2. Vidíme, že algoritmus dosahuje velmi vysokých časů už při velmi nízkém počtu bodů. Z časových důvodů jsem nestihl implementovat optimalizovanou verzi, popsanou v sekci 4.3, která by měla dosahovat výrazně lepšího výkonu.

5.2.3 QuichHull

Jak bylo popsáno, u algoritmu *QuickHull* bylo pro správnou funkčnost potřeba implementovat slučování hran, proto jsem algoritmus implementoval tak, jak je popsán v sekci 4.4. Jak ukazuje tabulka 5.1, *QuickHull* se ukázal jako výrazně nejrychlejší ze všech tří implementovaných algoritmů.



Obrázek 5.2: Měřené časy inkrementálního algoritmu pro všechna testovací data

5.3 Celkové porovnání

V této sekci ukážeme dobu běhů implementovaných algoritmů proti sobě na různých typech dat, popsaných v úvodu kapitoly. Zároveň budou algoritmy porovnány s knihovnou *QHull* [19].

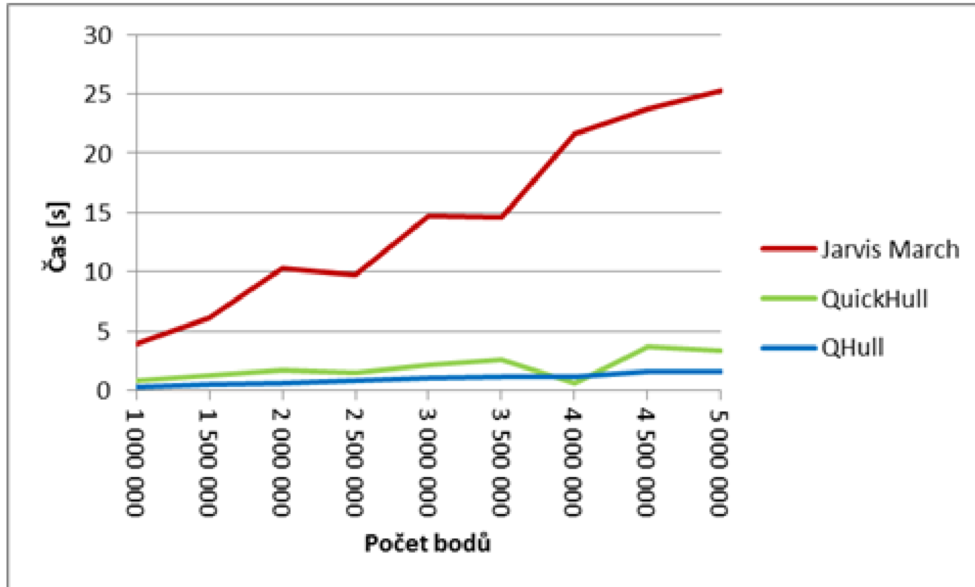
Dobu běhu při na náhodném rozmístění vstupních bodů uvnitř koule ilustruje tabulka 5.1.

	Čas [s]		
počet bodů	Jarvis March	QuickHull	QHull
100000	1.103	0.062	0.040
200000	3.165	0.125	0.080
300000	5.742	0.194	0.110
400000	8.997	0.263	0.140
500000	12.324	0.345	0.170
600000	16.328	0.407	0.200

Tabulka 5.1: Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 2

Jako další typ dat jsem zvolil body ležící uvnitř krychle. Výkon ilustruje obrázek 5.3. Můžeme vidět například výrazně zlepšení u algoritmu *Jarvis march*, který si vysvětluji mnohem menším počtem bodů na obálce, než v případě koule. Jelikož složitost inkrementálního algoritmu nezávisí na počtu bodů na

obálce, tak u tohoto typu dat pracuje téměř stejně rychle, jak ukazuje obrázek 5.2.

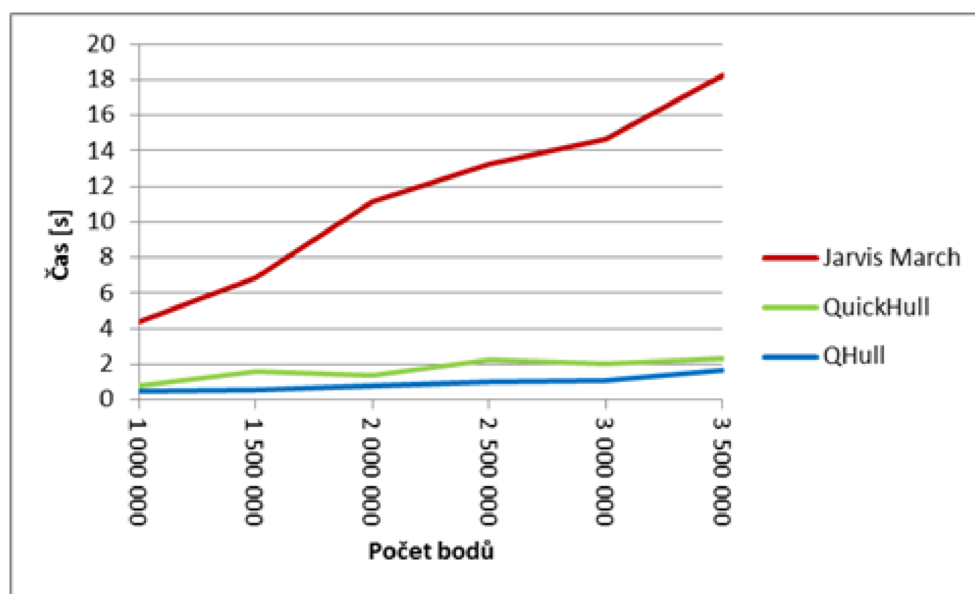


Obrázek 5.3: Porovnání algoritmů pro testovací data č. 1

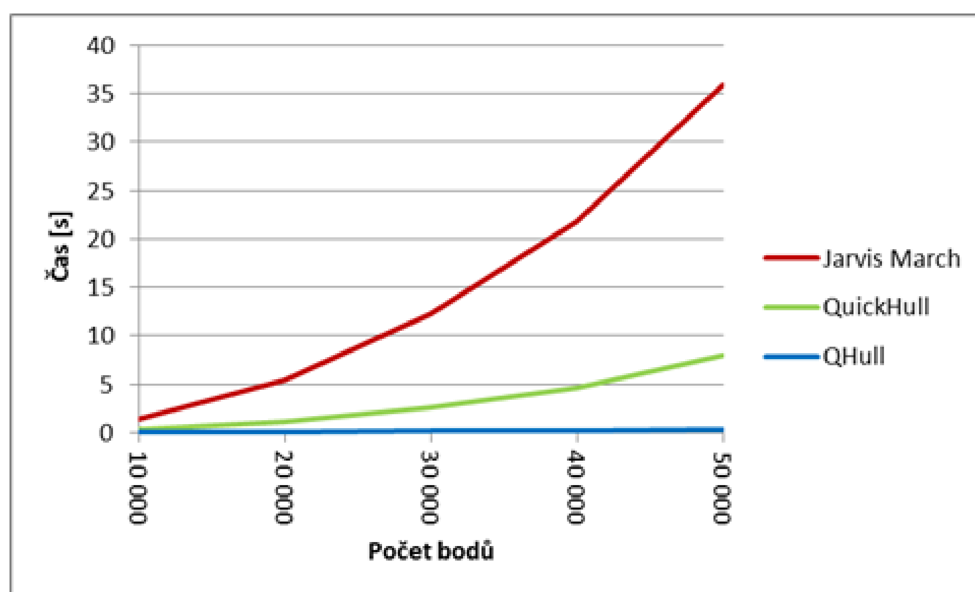
Třetím typem vstupních dat jsou body umístěné u povrchu krychle. Nezvolil jsem body ležící přímo na stěnách, protože u některých vstupů se algoritmus *Jarvis march* ani po dlouhé době (více než 5 minut) neukončil, přestože u vstupu, který byl větší, doběhl za několik sekund. Důvod tohoto chování jsem neodhalil. Jako vstupní body jsem tedy generoval body, které leží maximálně do 10% od stěn krychle. Výsledky ilustruje obrázek 5.4. Výsledky inkrementálního algoritmu opět můžeme vidět na obrázku 5.2

Posledním typem dat jsou body, ležící na povrchu koule. Na stejný problém jako u krychle jsem nenarazil, nicméně je třeba dodat, že u tohoto typu dat pracují algoritmy *Jarvis march* a *QuickHull* pomaleji, a to z důvodu, že velké množství vstupních bodů na povrchu koule bude ležet i na výsledné obálce. Výsledky ukazuje obrázek 5.5. Pro rychlost inkrementálního algoritmu se opět odkazuji na obrázek 5.2

Jak je vidět, nejrychlejším z mých implementovaných algoritmů je jednoznačně *QuickHull*. Oproti knihovně *QHull* je moje implementace u všech testovacích dat, kromě testovacích dat č. 3, pomalejší přibližně dvakrát. U testovacích dat č. 3 je toto zpomalení větší z důvodu velkého počtu bodů na obálce, se kterým se *QHull* umí vypořádat lépe.



Obrázek 5.4: Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 3



Obrázek 5.5: Měřené časy algoritmů Jarvis march, QuickHull a knihovny QHull pro testovací data č. 4

Závěr

V této práci jsme v souladu s prvním cílem implementovali různé algoritmy pro řešení problému konvexní obálky ve 3D prostoru. Zároveň byl splněn i druhý cíl – porovnání jednotlivých algoritmů mezi sebou i s existujícím řešením.

Hlavním výsledkem mé práce je konstatování, že ze tří mnou implementovaných algoritmů, pracuje algoritmus *QuickHull* výrazně nejrychleji. Částečně se tak podařilo navázat na práci kolegy Mitury [10], který dospěl ke stejnému závěru i u rovinné verze algoritmu. Výkonnost tohoto algoritmu dokazuje i jeho použití ve dvou nejrozšířenějších knihovnách pro řešení problému nalezení konvexní obálky – QHull [19] a CGAL [20].

Má implementace navíc výrazně nezaostává za implementací v knihovně *QHull*, což považuji za úspěch. Za neúspěch naopak považuji neimplementování paralelních verzí algoritmů a vylepšené verze inkrementálního algoritmu, způsobený pro mě velkou náročností implementace základních verzí daných algoritmů a nedostatku času pro paralelizaci tím způsobenou.

Literatura

[1] Wikipedia: Delaunay triangulation — Wikipedia, The Free Encyclopedia. Dostupné z `http://en.wikipedia.org/w/index.php?title=Delaunay%20triangulation&oldid=767721079`, 2017, [Online; citováno 5.4.2017].

[2] Wikipedia: Voronoi diagram — Wikipedia, The Free Encyclopedia. Dostupné z `http://en.wikipedia.org/w/index.php?title=Voronoi%20diagram&oldid=773513589`, 2017, [Online; citováno 05.4.2017].

[3] Edelsbrunner, H.: *Algorithms in Combinatorial Geometry*. Berlín: Springer-Verlag, 1987, ISBN 978-3-642-61568-9.

[4] Mulmuley, K.: *Computational Geometry: An Introduction Through Randomized Algorithms*. Londýn: Springer-Verlag, 1994, ISBN 978-0133363630.

[5] Rourke, J. O.: *Computational Geometry in C*. Cambridge: Cambridge University Press, 1994, ISBN 978-0521649766.

[6] Preparata, F. P.; Shamos, M.: *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985, ISBN 978-1-4612-1098-6.

[7] Graham, R. L.: An efficient algorith for determining the convex hull of a finite planar set. *Information processing letters*, ročník 1, č. 4, 1972: s. 132–133.

[8] Jarvis, R. A.: On the identification of the convex hull of a finite set of points in the plane. *Information processing letters*, ročník 2, 1973: s. 18–21.

[9] Kirkpatrick, D. G.; Seidel, R.: The ultimate planar convex hull algorithm. *SIAM Journal on Computing*, ročník 15, č. 1, 1986: str. 287–299.

[10] Mitura, P.: *Efektivní algoritmy pro řešení problému nalezení konvexní obálky.* Bakalářská práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

[11] de Berg, M.; Cheong, O.; van Kreveld, M.; aj.: *Computational Geometry: Algorithms and Applications.* Santa Clara: Springer-Verlag, třetí vydání, 2008, ISBN 978-3540779735.

[12] Jacobs, J.: Squared Analytics: Analyzing Crime, Sports and People. Dostupné z `https://squared2020.files.wordpress.com/2015/11/convex-non-convex.jpg`, [Online; citováno 7.5.2017].

[13] Haskell: Haskell: An advanced, purely functional programming language. Dostupné z `https://www.haskell.org/communities/11-2008/html/Figure.jpg`, [Online; citováno 7.5.2017].

[14] Felkel, P.: Convex hull in 3 dimensions. Dostupné z `https://cw.fel.cvut.cz/wiki/_media/misc/projects/oppa_oi_english/courses/ae4m39vg/lectures/05-convexhull-3d.pdf`, 2014, [Online; citováno 05.4.2017].

[15] Ramaswami, S.: Convex Hulls: Complexity and Applications (a Survey). Dostupné z `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1272&context=cis_reports`, 1993, [Online; citováno 05.4.2017].

[16] Containers — CPP reference. Dostupné z `http://www.cplusplus.com/reference/stl/`, [Online; citováno 17.4.2017].

[17] Standard Template Library: Algorithms. Dostupné z `http://www.cplusplus.com/reference/algorithm/`, [Online; citováno 28.4.2017].

[18] C numerics library. Dostupné z `http://www.cplusplus.com/reference/cmath/`, [Online; citováno 28.4.2017].

[19] Barber, B. C.; Dobkin, D. P.: Qhull 2015.2 [software]. Dostupné z `http://qhull.org`, [Citováno 24.4.2017].

[20] The CGAL Project: *CGAL User and Reference Manual.* CGAL Editorial Board, 4.9.1 vydání, 2017. Dostupné z: `http://doc.cgal.org/4.9.1/Manual/packages.html`

[21] Lloyd, J.: QuickHull3D: A Robust 3D Convex Hull Algorithm in Java. Dostupné z `https://www.cs.ubc.ca/~lloyd/java/quickhull3d.html`, [Online; citováno 28.4.2017].

[22] Gregorius, D.: Implementing QuickHull. Dostupné z `http://box2d.org/files/GDC2014/DirkGregorius_ImplementingQuickHull.pdf`, [Online; citováno 20.4.2017].

[23] Heckbert, P.: Quad-Edge Data Structure and Library. Dostupné z `https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/src/a2/quadedge.html`, 2001, [Online; citováno 21.4.2017].

[24] Chand, D. R.; Kapur, S. S.: An Algorithm for Convex Polytopes. *Journal of the ACM*, ročník 17, 1970: s. 78–86.

[25] std::set — CPP reference. Dostupné z `http://www.cplusplus.com/reference/set/set/`, [Online; citováno 6.4.2017].

[26] Preparata, F. P.; Hong, S. J.: Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, ročník 20, 1977: s. 87–93.

[27] Kallay, M.: The complexity of incremental convex hull algorithms in $\mathbb{R}^d$. *Information processing letters*, ročník 19, č. 4, 1984: str. 197.

[28] Barber, B. C.; Dobkin, D. P.: The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, ročník 22, č. 4, 1996: s. 469–483.

[29] Hoare, C. A. R.: Algorithm 64: Quicksort. *Communications of the ACM*, ročník 4, č. 7, 1961: str. 321.

[30] Chan, T. M.: A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm. Dostupné z `https://www.cs.jhu.edu/~misha/Spring16/Chan03.pdf`, 2003, [Online; citováno 28.4.2017].

[31] Seidel, R.: Backwards analysis of randomized geometric algorithms. In *New trends in discrete and computational geometry*, Springer, 1993, s. 37–67.

[32] Wikipedia: Floating point arithmetics — Wikipedia, The Free Encyclopedia. Dostupné z `https://en.wikipedia.org/wiki/Floating-point_arithmetic`, 2017, [Online; citováno 20.4.2017].

[33] Free Software Foundation, I.: GCC (GNU Compiler Collection) 4.9.3 [software]. Dostupné z `https://gcc.gnu.org/`, [Citováno 7.5.2017].

Příloha A

Seznam použitých zkratek

STL Standard Template Library

CPU Central Processing Unit

RAM Random Access Memory

Příloha **B**

Instalační příručka

V této příloze stručně popíši instalaci aplikace, formát vstupu a výstupu a příklad použití.

B.1 Požadavky

Má implementace byla vyvíjena pod operačním systémem GNU/Linux s kompilátorem GCC (resp. G++) ve verzi podporující standard C++11. Aplikace se dá zkompilovat pomocí nástroje GNU Make. Úspěch kompilace pod jiným operačním systémem není zaručen.

B.2 Instalace a použití

Program je možné zkompilovat spuštěním příkazu `make` v kořenovém adresáři projektu – `ConvexHull`. Vygenerují se spustitelné soubory v adresáři `bin` pod názvy `convex` a `incremental`. Program `convex` implementuje algoritmy *Jarvis march* a *QuickHull*, program `incremental` implementuje inkrementální algoritmus. Programy je možné spustit jednoduše z kořenového adresáře příkazem `./bin/convex` resp. `./bin/incremental`.

U programu `convex` je možné vynutit použití pouze jednoho ze dvou implementovaných algoritmů spuštěním příkazu `./bin/convex j` pro algoritmus *Jarvis march*, resp. `./bin/convex q` pro algoritmus *QuickHull*.

B.3 Formát vstupu a výstupu

Program přijímá na standartním vstupu nejdříve číslo d označující dimenzi (program umí ovšem pracovat jen s dimenzí 3, z důvodu možného rozšíření jsem však ponechal možnost volby), dále číslo n označující počet vstupních bodů a dále n trojic reálných neboli celých čísel, označujících souřadnice jednotlivých bodů. Jednotlivá čísla musí být odděleny bílými znaky. Příklad

vstupu:

```
3
6
0 0 0
10 5 6
2.2 3.5 9
-10 -10 -10
2.5 2.5 2
8 8 8
```

Výstup programu obsahuje nejdříve název použitého algoritmu, počet stěn nalezené konvexní obálky bodů ze vstupu a následně pro každou stěnu počet jejich vrcholů a jejich souřadnice. Příklad výstupu:

```
QUICKHULL:
6 faces
3 vertices
10 5 6
2.2 3.5 9
-10 -10 -10
3 vertices
8 8 8
2.2 3.5 9
10 5 6
3 vertices
2.5 2.5 2
-10 -10 -10
2.2 3.5 9
3 vertices
2.5 2.5 2
2.2 3.5 9
8 8 8
3 vertices
2.5 2.5 2
8 8 8
10 5 6
3 vertices
2.5 2.5 2
10 5 6
-10 -10 -10
```

Příloha C

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
ConvexHull ........................... kořenový adresář implementace
  src ................................ zdrojové kódy implementace
  bin ................................ cílový adresář pro spustitelné soubory
thesis ............................... text práce
  src ................................ zdrojové kódy práce
  BP_Motyka_Vaclav_2017.pdf .......... text práce ve formátu PDF
```