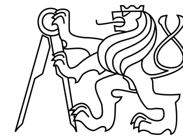
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE

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

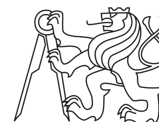
[1] Mitura, Peter, Efektivní algoritmy pro ešení problému nalezení konvexní obálky, BP FIT VUT, 2016
[2] Roger Hernando, Convex hull algoriths in 3D:
http://dccg.upc.edu/people/vera/wp-content/uploads/2014/11/GA2014-ConvexHulls3D-Roger-Hernando.pdf
[3] Petr Felkel, Convex hull algoriths in 3D: https://cw.fel.cvut.cz/wiki/_media/misc/projects/oppa_oi_english/courses/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.

8. ledna 2018

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 8. ledna 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 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, 2018.

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 naimplementované, optimalizované a paralelizované. Dále jsou tyto algoritmy porovnané mezi sebou a zároveň proti již existujícímu řešení, v podobě knihoven QHull a CGAL.

Klíčová slova konvexní obálka, výpočetní geometrie, QuickHull, Jarvis March, inkrementální algoritmus

Abstract

This thesis deals with efficient 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 parallelized. Also, these algorithms are compared against each other and against an existing solution, the QHull library and the CGAL library.

Keywords convex hull, computational geometry, QuickHull, Jarvis March, incremental algorithm

Obsah

Úvod **1**
Cíle práce . . . 1
Struktura práce . . . 2

1 Teorie **3**
1.1 Základní pojmy . . . 3
1.2 Konvexní obálka . . . 5
1.3 Dolní hranice složitosti . . . 10

2 Softwarové aspekty **11**
2.1 Použité technologie . . . 11
2.2 Existující řešení . . . 13
2.3 Datové struktury . . . 13

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 . . . 23
3.5 Brute force . . . 27
3.6 Další algoritmy . . . 27

4 Optimalizace **29**
4.1 Obecné optimalizace . . . 29
4.2 Jarvis March . . . 30
4.3 Inkrementální algoritmus . . . 31
4.4 Fat Planes . . . 33
4.5 Paralelizace . . . 34

5 Výsledky **37**

5.1 Testovací data 37
5.2 Celkové srovnání 38
5.3 Paralelizace 43

Závěr **51**

Literatura **53**

A Seznam použitých zkratek **57**

B Instalační příručka **59**
B.1 Požadavky 59
B.2 Instalace a použití 59
B.3 Formát vstupu a výstupu 60

C Obsah přiloženého CD **63**

Seznam obrázků

1.1 Konvexní a nekonvexní množina v $\mathbb{R}^2$ 5
1.2 Konvexní obálka v $\mathbb{R}^2$ a $\mathbb{R}^3$ 5
1.3 Nadrovina a její poloprostory v $\mathbb{R}^2$ 6
1.4 Opěrná nadrovina v $\mathbb{R}^2$ 7
1.5 Simplexy v prostorech $\mathbb{R}^1$, $\mathbb{R}^2$ a $\mathbb{R}^3$ 7
1.6 Topologie 3D-simplexu 8

2.1 Ukázka half-edge struktury 14

3.1 Sloučení dvou konvexních obálek 18
3.2 Sloučení dvou konvexních obálek 19
3.3 Horizont a viditelné stěny (bíle) 21
3.4 Sloučení stěny 22
3.5 Chybné stěny před sloučením 25
3.6 Příklad chybných stěn 25
3.7 Test konvexnosti stěn F_{Left} a F_{Right} 26

4.1 Funkce kosinus 30
4.2 Graf konfliktů 32
4.3 Přidání nové stěny 33
4.4 Fat planes 33
4.5 Nalezení jedné stěny vícekrát 35

5.1 Srovnání všech algoritmů pro testovací data č. 1. 39
5.2 Srovnání rychlejších algoritmů pro testovací data č. 1. 40
5.3 Srovnání všech algoritmů pro testovací data č. 2. 40
5.4 Srovnání rychlejších algoritmů pro testovací data č. 2. 41
5.5 Srovnání všech algoritmů pro testovací data č. 3. 41
5.6 Srovnání rychlejších algoritmů pro testovací data č. 3. 42
5.7 Srovnání všech algoritmů pro testovací data č. 4. 42
5.8 Srovnání rychlejších algoritmů pro testovací data č. 4. 43

5.9 Srovnání všech paralelních algoritmů pro 100 000 bodů ležících uvnitř koule. 44
5.10 Srovnání rychlejších paralelních algoritmů pro 1 000 000 bodů ležících uvnitř koule. 45
5.11 Srovnání zrychlení všech paralelních algoritmů pro 100 000 bodů ležících uvnitř koule. 45
5.12 Srovnání zrychlení rychlejších paralelních algoritmů pro 1 000 000 bodů ležících uvnitř koule. 46
5.13 Srovnání všech paralelních algoritmů pro 10 000 bodů ležících na povrchu koule. 47
5.14 Doba trvání paralelního algoritmu QuickHull pro 100 000 bodů ležících na povrchu koule. 47
5.15 Srovnání zrychlení všech paralelních algoritmů pro 10 000 bodů ležících na povrchu koule. 48
5.16 Zrychlení paralelního algoritmu QuickHull pro 100 000 bodů ležících na povrchu koule. 48

Úvod

Problém hledání konvexní obálky množiny bodů patří mezi jeden ze základních problémů výpočetní geometrie. Jeho podstatou je nalezení takového mnohostěnu, že všechny tyto 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 R^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 R^2 pracují v čase $\mathcal{O}(n \log h)$ a nemohou pracovat rychleji, jak bylo dokázáno v práci Kirkpatricka a Seidela, kteří též poskytli i konkrétní algoritmus [9].

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

Cíle práce

Cílem této práce je implementace některých známých algoritmů pro vytvoření konvexní obálky bodů v prostoru R^3. Dalším cílem je analýza možností paralelizace těchto algoritmů a implementace jejich paralelních verzí. Posledním cílem je porovnání jejich rychlosti, za jakou jsou tyto algoritmy schopné daný

problém vyřešit a zároveň porovnání, jak účinná je u těchto algoritmů jejich paralelizace.

Struktura práce

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

Ve druhé kapitole budou popsány 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 tři a popsány možnosti jejich paralelizace.

V páté kapitole bude ukázáno srovnání jednotlivých algoritmů podle jejich rychlosti na různých typech vstupních dat.

Kapitola 1

Teorie

V této kapitole budou zadefinovány důležité pojmy používané v této práci. Definice vychází z práce Petera 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 (vektorový) prostor* $\mathbb{R}^n$ je množina uspořádaných n-tic (*vektorů*) prvků z $\mathbb{R}$, vybavená dvěma operacemi

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

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

a pro kterou platí:

1. Pro každé dva prvky $u, v \in \mathbb{R}^n$ platí

$$u \oplus v = v \oplus u,$$

2. pro každé tři prvky $u, v, w \in \mathbb{R}^n$ platí

$$(u \oplus v) \oplus w = u \oplus (v \oplus w),$$

3. pro každé $\alpha, \beta \in \mathbb{R}$ a $u \in \mathbb{R}^n$ platí

$$\alpha \odot (\beta \odot u) = (\alpha \cdot \beta) \odot u,$$

4. pro každé $\alpha, \beta \in \mathbb{R}$ a $u \in \mathbb{R}^n$ platí

$$(\alpha + \beta) \odot u \ = (\alpha \odot u) \oplus (\beta \odot u),$$

5. pro každé $\alpha \in \mathbb{R}$ a každé dva prvky $u, v \in \mathbb{R}^n$ platí

$$\alpha \odot (u \ \oplus v) = (\alpha \odot u) \oplus (\alpha \odot v),$$

6. pro libovolný prvek $u \in \mathbb{R}^n$ platí

$$1 \odot u \ = u,$$

7. pro každé dva prvky $u, v \in \mathbb{R}^n$ platí

$$0 \odot u \ = 0 \odot v$$

Dále si zavedeme pojem *úsečka.* Nadále v této práci budeme prvky $\mathbb{R}^n$ nazývat body namísto vektory. Jednotlivé složky vektoru si můžeme představit jako souřadnice bodu v dané dimenzi n, 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

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

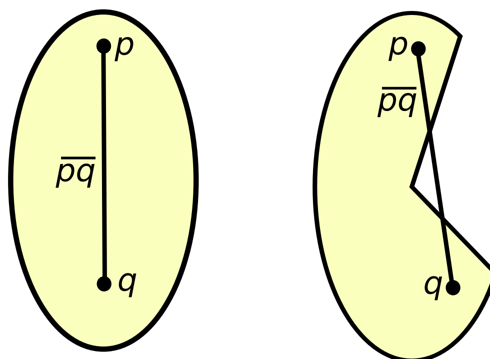
nazýváme *úsečkou s koncovými body x,y* a označujeme ji $\overline{xy}$.

Nyní můžeme zadefinovat konvexní množinu.

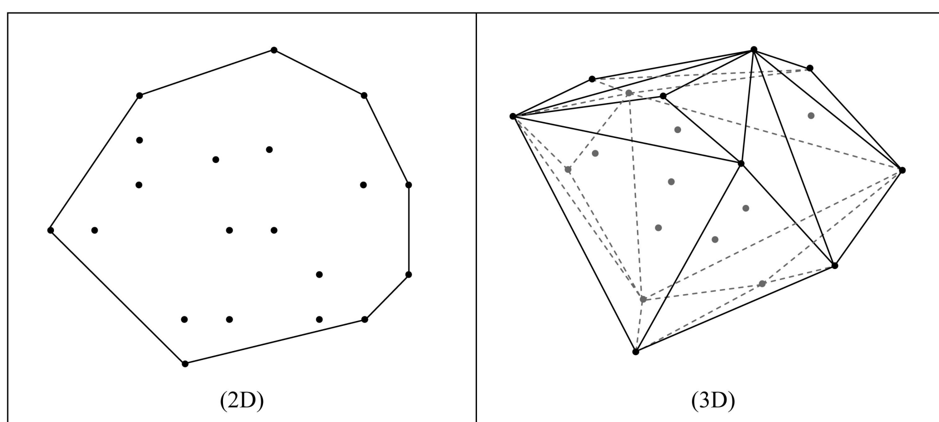
Definice 3. Množinu S v lineárním prostoru $\mathbb{R}^n$ nazýváme konvexní, pokud pro každou dvojici bodů x, y z S platí, že i úsečka $\overline{xy}$ se nachází v množině S.

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

Konvexnost množiny si můžeme ilustrovat na obrázku 1.1.



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



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

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.

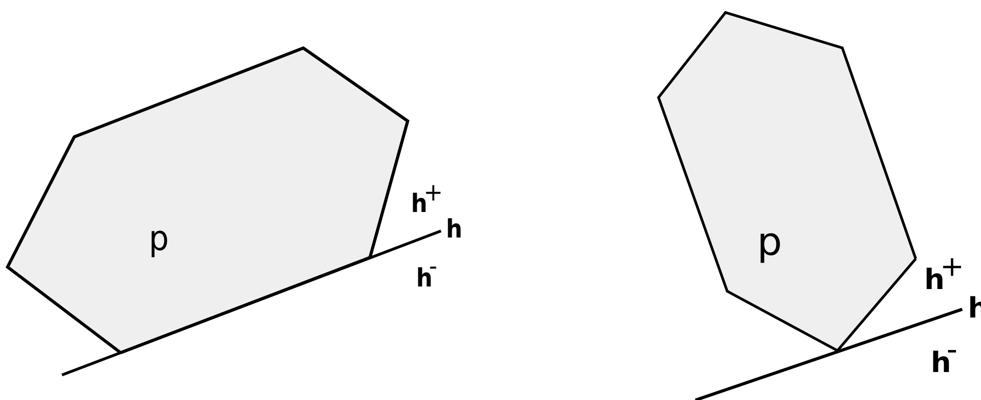
Definice 5. Nechť L s operacemi $\oplus, \odot$ je lineární prostor. Neprázdnou podmnožinu $L' \subseteq L$ nazveme *lineárním podprostorem* lineárního prostoru L, jestliže

1. pro všechny vektory $u, v \in L'$ platí, že $u \oplus v \in L'$,

2. pro všechna $\alpha \in \mathbb{R}$ a všechny vektory $u \in L'$ platí, že $\alpha \odot u \in L'$.

Definice 6. *Nadrovina* daného lineárního 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[13] dělí každá nadrovina h prostor na dva poloprostory h^+ a h^-, viz obrázek 1.3.



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

Definice 7. Nechť $v_0, v_1 \dots v_k$ jsou body v prostoru $\mathbb{R}^n$. Tyto body se nazývají afinně závislé, jestliže existují taková reálná čísla $\alpha_0, \alpha_1 \dots \alpha_k$, která nejsou všechna 0 tak, že $\sum_{i=0}^{k} \alpha_i v_i = 0$ a $\sum_{i=0}^{k} \alpha_i = 0$. V opačném případě se body nazývají afinně nezávislé.

Definice 8. *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 čtyřstěn, jak je vidno na obrázku 1.5.

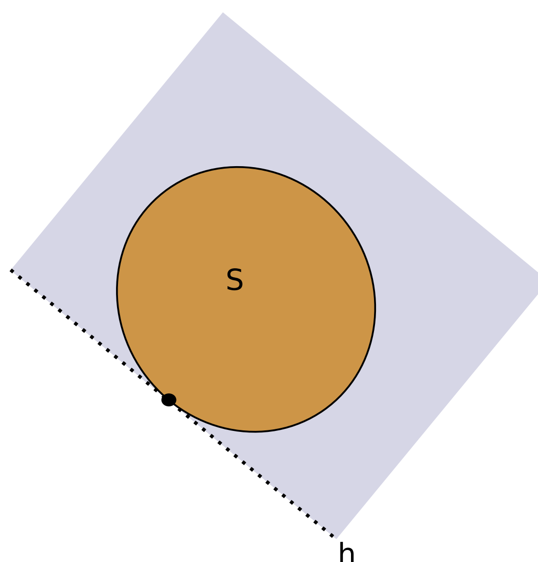
Definice 9. *Opěrná nadrovina h* množiny S v Eukleidovském prostoru $\mathbb{R}^n$ je nadrovina, pro kterou platí:

1. S je kompletně obsažena v jednom ze dvou uzavřených poloprostorů ohraničených nadrovinou h,

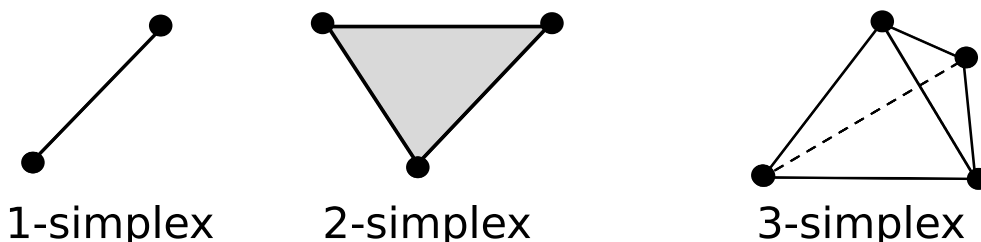
2. alespoň jeden bod množiny S leží na h

Ukázku opěrné nadroviny v prostoru $\mathbb{R}^2$ ilustruje obrázek 1.4.

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



Obrázek 1.4: Opěrná nadrovina v $\mathbb{R}^2$



Obrázek 1.5: Simplexy v prostorech $\mathbb{R}^1$, $\mathbb{R}^2$ a $\mathbb{R}^3$

Definice 10. Stěna *konvexní obálky (polytopu)* v $\mathbb{R}^n$ je průnik této obálky s její opěrnou nadrovinou.

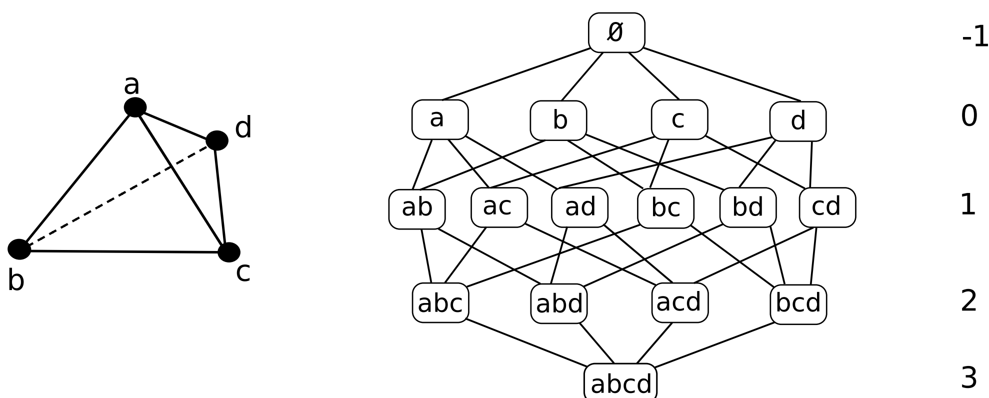
Předchozí definice je trochu krkolomná a nejlépe se ukáže na příkladu: v $\mathbb{R}^2$ jsou stěnami úsečky a v $\mathbb{R}^3$ to jsou mnohoúhelníky. Z toho a z předchozích definic si můžeme odvodit, že zatímco v $\mathbb{R}^2$ je konvexní obálkou 2D-polytop (mnohoúhelník), v $\mathbb{R}^3$ je jí 3D-polytop (mnohostěn).

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 $\mathbb{R}^n$ se skládá z vlastních stěn a nevlastních stěn. Vlastní stěny jsou stěny dimenze $0 \dots (d-1)$, nevlastní jsou pak stěny dimenze -1 (prázdná množina) a dimenze d (celý polytop). Díky tomuto můžeme ukázat na příkladu 3D-simplexu topologii polytopu 3D-simplex (obrázek 1.6).

O polytopech dále platí několik tvrzení:

- Ohraničení polytopu tvoří sjednocení jeho pravých stěn.



Obrázek 1.6: Topologie 3D-simplexu

- Každý polytop je tvořen konečným množstvím stěn.
- 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 $\mathbb{R}^3$, 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:*

$$E \leq 3V - 6$$

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:

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

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

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

Jednoduchými úpravami se dostáváme k původní rovnici

$$E \leq 3V - 6$$

□

Důsledek 1. Nechť M je konvexní mnohostěn s V vrcholy v $\mathbb{R}^3$. 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 11. Nechť body a, b, c jsou body ležící v prostoru $\mathbb{R}^3$, potom tyto body nazveme *kolineární*, pokud jsou afinně závislé.

Definice 12. Nechť body a, b, c, d jsou body ležící v prostoru $\mathbb{R}^3$, potom tyto body nazveme *koplanární*, pokud leží v jedné rovině. Formálněji, tyto body jsou koplanární tehdy a pouze tehdy, když platí

$$[(b-a) \times (d-a)] \cdot (c-a) = 0$$

kde $\times$ je vektorový součin a $\cdot$ je skalární součin.

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

U všech implementovaných algoritmů se využívá pro jejich správnou funkci uspořádání bodů po, resp. proti směru hodinových ručiček. V prostoru $\mathbb{R}^3$ mohou tři body definovat rovinu, která jimi prochází. Tato rovina dále dělí prostor $\mathbb{R}^3$ na dva poloprostory. Proto pro uspořádanou množinu tří bodů nelze určit, zda je uspořádaná po, resp. proti směru hodinových ručiček, dokud neznáme bod, ze kterého se na tyto body díváme. Při pohledu z jednoho poloprostoru budou uspořádané po směru, při pohledu z druhého naopak proti směru hodinových ručiček.

Definice 13. Mějme body a,b,c v prostoru $\mathbb{R}^3$. Vektor v_{ab} z bodu a do bodu b, vektor v_{ac} z bodu a do bodu c. Dále mějme bod d, neležící ve stejné rovině jako body a, b, c a vektor v_{ad} z bodu a do bodu d. Normálový vektor v_n roviny procházející body a, b, c spočteme jako vektorový součin $v_{ab} \times v_{ac}$. Potom pokud je úhel mezi v_n a v_{ad} menší než 90°, jsou tyto body a,b,c uspořádané proti směru hodinových ručiček při pohledu z bodu d. Pokud je tento úhel větší než 90°, body jsou uspořádány po směru hodinových ručiček při pohledu z bodu d.

Definice 14. Mějme množinu bodů $f = \{p_1, p_2, p_3 \ldots p_n\}$. Tato množina bodů je uspořádaná podle směru, resp. proti směru hodinových ručiček při pohledu z bodu d, pokud pro každou trojici bodů

$$\{p_i, p_{i+1}, p_{i+2} : 1 \leq i \leq n-2\}$$

platí, že je uspořádaná po směru, resp. proti směru hodinových ručiček.

1.3 Dolní hranice složitosti

Při porovnání rychlosti algoritmů se jako základní ukazatel využívá horní hranice jejich složitosti. U algoritmů pro nalezení konvexní obálky v $\mathbb{R}^2$ platí, že horní 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 $\mathbb{R}^3$, nejrychlejší známé algoritmy pracují v čase $\mathcal{O}(n \log n)$.

Kapitola 2

Softwarové aspekty

V této kapitole budou popsány softwarové 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. V mojí práci jsem nejčastěji využíval kontejnery z STL knihovny C++ [14]. 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. Dále byla využita knihovna *algorithm* [15], resp. funkce *sort* pro řazení polí, funkce *find* pro nalezení prvku v poli nebo množině a funkce *random_shuffle* pro promíchání prvků pole nebo *vectoru*. Dále jsem využíval pro matematické výpočty knihovnu *cmath* [16]. Konkrétně byly využity funkce pro výpočet absolutní hodnoty, funkce cosinus a odmocnina. 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 doplněny 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 15. *Zrychlení S* definujeme jako poměr doby běhu sekvenčního (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** [17] 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** [18] 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.

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++ [14].

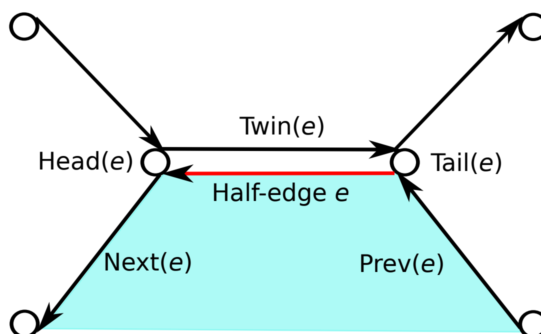
2.3.1 Reprezentace mnohostěnu

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 imlementaci algoritmu *QuickHull*. V ostatních implementacích byla pro jednodušší práci využita struktura obyčejnější, a to reprezentace stěny jako množiny bodů a reprezentace mnohostěnu jako množiny stěn.

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[19]. Hlavní rozdíl oproti naivní struktuře je rozdělení každé hrany na dvě s opačnou orientací, každou náležející jedné z incidujících stěn. Ukázka této struktury je na obrázku 2.1.



Obrázek 2.1: Ukázka half-edge struktury

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 konkrétní implementaci, zda je využije nebo nikoliv. V mé implementaci je dostačující uchovávat si pouze stěny, které daný vrchol obsahují.

2.3.1.2 Další možné struktury

Mezi další často používané datové struktury pro reprezentaci mnohostěnů patří *Winged-edge* struktura[5, str. 146], která si pro každou hranu uchovává její incidující stěny a jejího 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 [20].

Kapitola 3

Algoritmy pro nalezení konvexní obálky ve 3D

V této kapitole budou popsány některé známé algoritmy pro nalezení konvexní obálky v $\mathbb{R}^3$, zároveň bude 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 $\mathbb{R}^2$ 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 $\mathbb{R}^n$, ovšem se složitostí $\mathcal{O}(n^2)$ [21]. 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 $\mathbb{R}^2$. Toho docílíme tak, že najdeme normálový vektor roviny, která těmito body prochází a následně u těchto bodů eliminujeme jednu souřadnici (x, y nebo z), která je u normálového vektoru nenulová. Pokud bychom eliminovali souřadnici, která je u normálového vektoru nulová, body by po promítnutí ležely na přímce. Dále najdeme konvexní obálku těchto bodů 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 při pohledu zvenčí obálky (pokud tak bude nastaven algoritmus pro hledání obálky v $\mathbb{R}^2$). Tento mnohoúhelník značíme 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 dále.

Algoritmus 1 Jarvis March

```
1: t ← findInitialFacet()
2: fresh ← {(t0, t1), (t1, t2), (t2, t0)} {množina nezpracovaných hran}
3: H ← t {obálka}
4: while fresh ≠ ∅ do
5:   e ← fresh.pop()
6:   q ← borderPoints(e) {všechny ostatní body leží na jedné straně roviny
     procházející body z q}
7:   f ← findHull(q)
8:   H ← H ∪ {f}
9:   for all e ∈ edges(f) do
10:    if e ∉ closed then fresh ← fresh ∪ e
11:    end if
12:    if e ∈ fresh then closed ← closed ∪ e
13:    end if
14:  end for
15: 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 $\mathcal{O}(nh)$ [5, str. 109].*

3.2 Algoritmus Divide and Conquer

Jak bylo řečeno v úvodu, spodní hranice složitosti pro konstrukci konvexní obálky v $\mathbb{R}^3$ je $\Omega(n \log n)$. Přirozeně nás tedy zajímá algoritmus, který této složitosti dosahuje. Přestože bylo řečeno, že některé algoritmy se dají rozšířit z $\mathbb{R}^2$ do $\mathbb{R}^3$, jediný známý algoritmus, který dosahuje optimální časové složitosti $\mathcal{O}(n \log n)$, je algoritmus představený v roce 1977 Francem P. Preparatou a S. J. Hongem [22] [5].

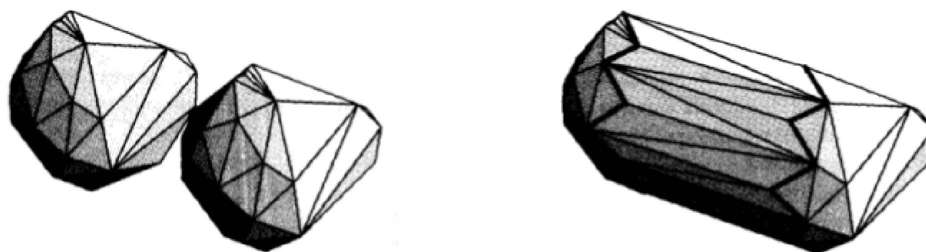
3.2.1 Popis algoritmu

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 algo-

ritmů, popřípadě metodou *brute force*, která bude popsaná dále v sekci 3.5 [1]), a následně provedeme jejich sloučení. Abychom splnili podmínku pro časovou náročnost $\mathcal{O}(n \log n)$, musí sloučení proběhnout v amortizovaném 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 amortizovaném čase $\mathcal{O}(n)$ za předpokladu, že přidání nové stěny trvá konstantní čas.

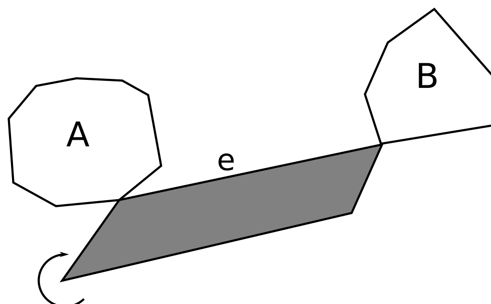
Sloučení A a B začíná nalezením nějaké hrany, která je propojuje, nazvěme ji e, a která má koncové body $a \in A$ a $b \in B$. To se dá dosáhnout promítnutím A a B do roviny (například eliminací souřadnice z) a využitím algoritmu pro hledání spodního tangentu[2] dvou mnohoúhelníků. Tento algoritmus se využívá i ve 2D verzi *Divide and Conquer* a je popsán například v [5, str. 94–96]. V další části využijeme algoritmu *Jarvis March* (3.1) a obalujeme A a B rovinou procházející e, dokud nenarazíme na bod, nazvěme ho c, ležícího na A nebo B. Postup nalezení tohoto bodu je stejný jako u algoritmu *Jarvis March* a je popsán v sekci (3.1).

První stěnu vytvoříme z koncových bodů úsečky e a bodu c. Leží-li bod c na mnohostěnu A, další zpracovávanou hranou bude úsečka $\overline{cb}$, leží-li na B, další hranou bude úsečka $\overline{ac}$.

[1]V mé implementaci využívám algoritmu QuickHull, jakmile je počet bodů menší než 500, aby se zamezilo zbytečnému množství rekurzivních zanoření.

[2]spodní tangent dvou mnohoúhelníků A a B je úsečka $\overline{ab}$, taková, že $a \in A$, $b \in B$ a všechny ostatní body z A a B leží nad ní nebo na ní.

Nyní v obalování pokračujeme stejným způsobem, dokud nově nalezená hrana nebude opět původní 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

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.

Posledním krokem algoritmu je sloučení koplanárních stěn. Jelikož při sloučení obálek A a B vznikají pouze trojúhelníkové stěny, je potřeba na konci sloučit ty stěny, které jsou koplanární mezi sebou, příp. koplanární s nějakou stěnou ležící na jedné ze dvou slučovaných obálek. Abychom nemuseli kontrolovat každou dvojici stěn, budeme kontrolovat jen ty, které mohou nějakou koplanární stěnu mít. V každém kroku sloučení přidáváme jednu stěnu, pokud je nějaký z vrcholů A nebo B s touto stěnou koplanární (a netvoří tuto stěnu), uložíme si ji do množiny možných koplanárních stěn. Po dokončení sloučení bude stačit zkontrolovat tyto uložené stěny. Pro každou z těchto stěn bude potřeba najít všechny stěny sloučené obálky, které jsou s ní koplanární a sloučit je do jedné pomocí nějakého algoritmu pro nalezení konvexní obálky ve 2D. Tento postup je podobný jako u algoritmu *Jarvis March* 3.1.

Pro detaily algoritmu odkazuji na původní práci Preparaty a Honga [22], kde jsou všechny kroky popsány detailně, zároveň s důkazem složitosti tohoto algoritmu.

3.2.2 Detekce viditelných stěn

V tomto, ale i v dalších popisovaných algoritmech je důležité detekovat, které stěny jsou viditelné z určitého bodu. Za předpokladu, že jsou všechny stěny obálky uspořádány proti směru hodinových ručiček při pohledu zvenčí obálky, budeme tvrdit následující:

Definice 16. Stěna f konvexní obálky je viditelná z nějakého bodu d právě tehdy, když je při pohledu z tohoto bodu uspořádána proti směru hodinových ručiček.

V dalším textu budeme dále mluvit o pozici bodu pod, resp. nad rovinou.

Definice 17. Nechť f je stěna konvexní obálky a r rovina, která jí prochází. Potom řekneme, že bod d leží nad rovinou r právě tehdy, když je stěna f uspořádána proti směru hodinových ručiček při pohledu z něj a že leží pod rovinou r právě tehdy, když je stěna f uspořádána po směru hodinových ručiček při pohledu z něj.

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 [23] a jeho základní varianta dosahuje složitosti $\mathcal{O}(n^2)$.

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 kroku přidáváme, potom

$$H_i \leftarrow H_{i-1} \cup [p_i] - \overline{H_{i-1}}$$

, kde $[p_i]$ jsou nově přidané stěny z bodu p_i a $\overline{H_{i-1}}$ jsou stěny ležící uvnitř obálky H_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.

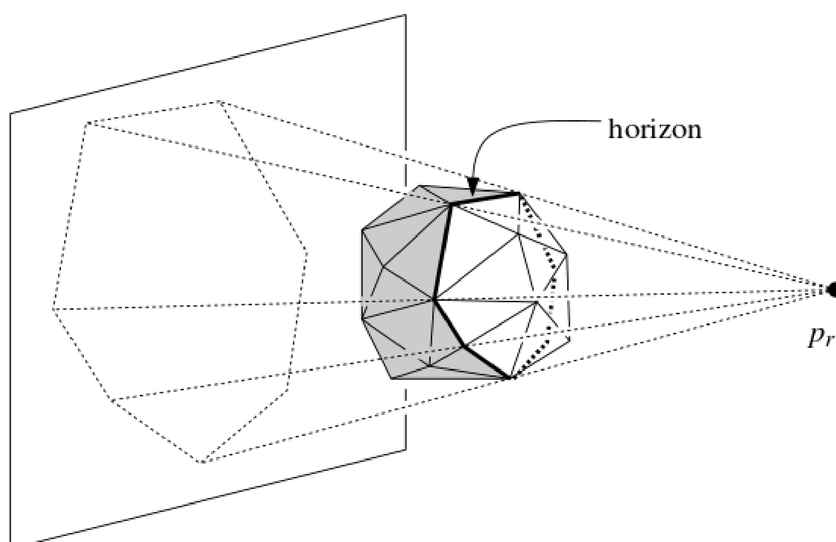
Bod p_i bude ležet uvnitř mnohostěnu H_{i-1} právě tehdy, když žádná stěna mnohostěnu H_{i-1} nebude z bodu p_i viditelná.

Jak bylo řečeno, pokud bod p_i 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_i.

Pro každou stěnu si tedy určíme, zda je viditelná z bodu p_i. Pokud není viditelná ani jedna stěna, bod p_i leží uvnitř obálky H_{i-1} a přejdeme na další iteraci. Pokud tomu tak není, je potřeba vytvořit nové stěny a smazat stěny, které budou ležet uvnitř nové obálky H_i. Jak tyto stěny poznat již víme, zbývá zjistit, kam napojit stěny nové. Jedním z vrcholů každé nové stěny bude bod p_i. Zbylé dva body každé nové stěny budou tvořit koncové body hran incidujících

s jednou stěnou viditelnou z bodu p_i a jednou stěnou, která z bodu p_i vidět není. Tyto hrany budou tvořit takzvaný *horizont* (obrázek 3.3).

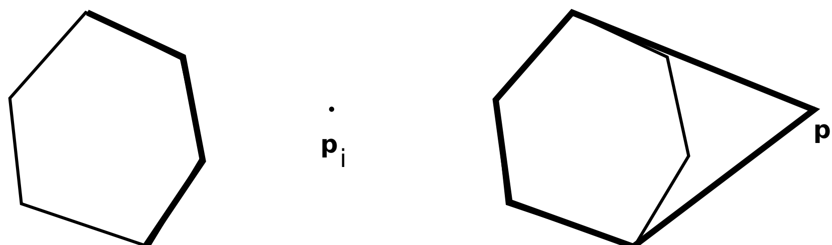
Poslední důležitou poznámkou, kterou je třeba zmínit je, že pokud máme vstup degenerovaný (najdeme v něm více než tři body ležící v rovině), musíme při přidávání bodu p_i sloučit všechny koplanární stěny. To se bude dít v případě, že zjistíme, že bod p_i je koplanární s nějakou stěnou H_{i-1}. Pokud nějakou takovou stěnu najdeme, uložíme si ji. Na konci iterace pak z každé takové stěny vytvoříme stěnu novou, obsahující i bod p_i. Toho docílíme tak, že zjistíme, které hrany *horizontu* leží na této stěně. Ty poté ze stěny odstraníme, čímž na stěně dostaneme dva body, které budou incidovat pouze s jednou hranou. Z těchto bodů následně vytvoříme hrany nové, vedoucí do bodu p_i. Toto sloučení ilustruje obrázek 3.4.



Obrázek 3.3: Horizont a viditelné stěny (bíle)

Převzato z [11]

Pro pseudokód inkrementálního algoritmu viz Algoritmus 2.



Obrázek 3.4: Sloučení stěny

Vlevo tučně stěny ležící na horizontu, vpravo tučně sloučená stěna.

Algoritmus 2 Inkrementální algoritmus

```
points ← allPoints
T ← findInitialTetrahedron() {body p1, p2, p3, p4}
H ← faces(T) {vytvoří stěny mnohostěnu T}
for all point p ∈ points do
  for all facet f ∈ H do
    if p ∈ T then
      continue
    end if
    if facetIsVisibleFromPoint(f, p) then
      markVisible(f)
    end if
    if facetIsCoplanar(f, p) then
      markCoplanar(f)
    end if
  end for
  if any faces visible then
    h ← findHorizon(H, p)
    if any facet coplanar then
      for all coplanar facet cf do
        mergeFacet(cf, p)
        deleteUsedEdgedFromHorizon()
      end for
    end if
```

```
    for all edge e ∈ h do
      f ← createFace(e, p)
      H ← H ∪ {f}
    end for
    for all f ∈ visible(p) do
      H ← H − {f}
    end for
  end if
end for
```

3.3.2 Analýza složitosti

Věta 4. *Nechť vstupní množina bodů M obsahuje n bodů. Potom složitost výše uvedené implementace inkrementálního algoritmu pro nalezení konvexní obálky těchto bodů je $\mathcal{O}(n^2)$.*

Důkaz. Pro důkaz se odkazuji na [24]. □

V sekci 4.3 ukážeme, že 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 [25].

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, najít *horizont* a vytvořit nové stěny spojující *horizont* a bod e. Pro nalezení viditelných stěn a horizontu můžeme postupovat takto. Uložíme si na zásobník (jiný, než kde mám uložené stěny) všechny hrany aktuální stěny F a označíme si ji jako navštívenou. Vyjmeme ze zásobníku první hranu, nazvěme ji h a najdeme k ní opačnou stěnu (stěnu s ní incudující, ale ne tu, ze které jsme přišli). Tu označíme jako F_{opp}. Pokud platí, že stěna F_{opp} není viditelná, potom hrana h bude ležet na horizontu. Pokud jsou obě stěny z bodu e viditelné, stěnu F_{opp} označíme jako navštívenou všechny její hrany uložíme na zásobník. Toto opakujeme, dokud není zásobník prázdný. Na konci budeme znát všechny hrany tvořící horizont. Nyní můžeme vytvořit nové stěny z body e a hran horizontu stejně, jako u Inkrementálního algoritmu.

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.

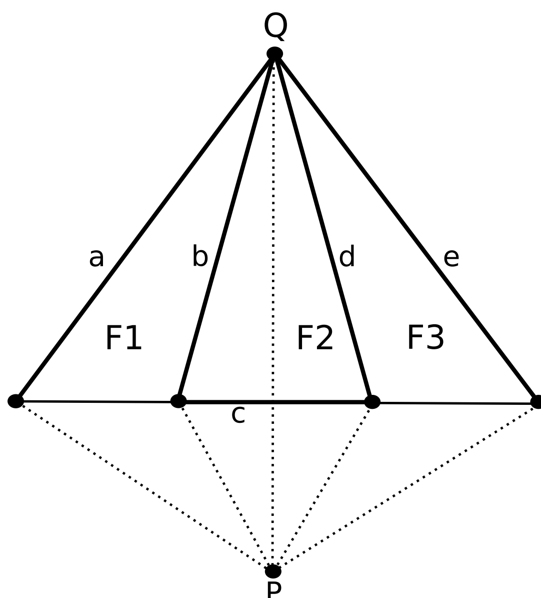
Poslední problém, který v průběhu algoritmu musíme řešit, je slučování stěn. Problém spočívá v numerické nepřesnosti výpočtů při počítání s desetinnými čísly ve formátu plovoucí desetinné čárky [26]. Tyto výpočty s sebou přinášejí malé nepřesnosti, které v algoritmu mohou zapříčinit jeho nesprávnou funkčnost. Jeden z problémů, který může nastat, byl popsán již v původní práci Barbera a Dobkina [25] a zároveň s ním i jeho řešení, spočívající právě ve slučování stěn.

Problém si ukážeme na obrázku 3.5. 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.

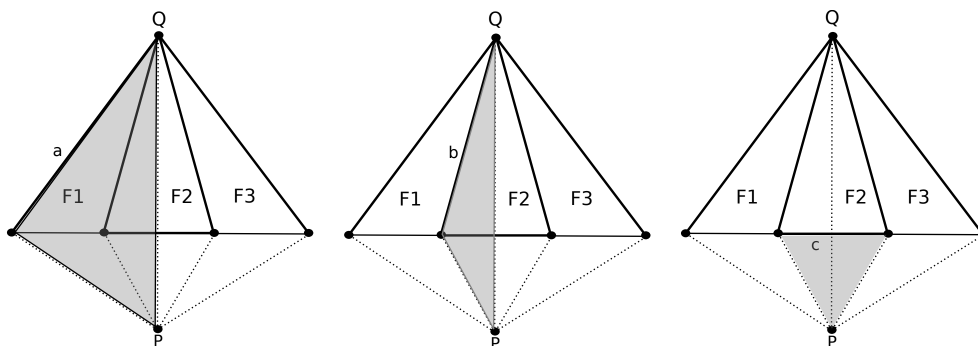
Řešení tohoto problému spočívá v kontrole obálky na konci každé iterace – pokud nalezneme stěny, které nejsou konvexní, sloučíme je. Pojmenujme si nové stěny podle hran, které by tvořili horizont bodu P, tedy $a - e$ (ukázku třech z pěti špatně vytvořených stěn ilustruje obrázek 3.6).

Stěny a, b, d, e budou spolu sdílet jednu hranu (PQ). Jelikož víme, že každá hrana musí 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 slučování skončit, v opačném případě tyto poslední dvě stěny sloučíme do jedné.

Zda jsou dvě stěny konvexní lze zkontrolovat jednoduchým testem, uká-



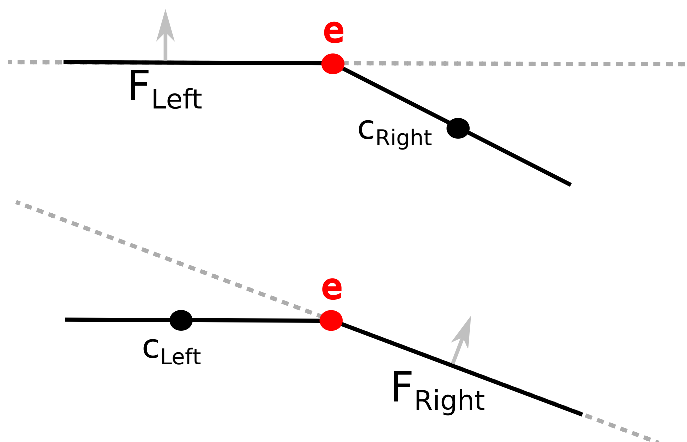
Obrázek 3.5: Chybné stěny před sloučením



Obrázek 3.6: Příklad chybných stěn

zaným na obrázku 3.7. 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}.

Dále nalezneme pseudokód *QuickHullu*:



Obrázek 3.7: Test konvexnosti stěn F_{Left} a F_{Right}

Algoritmus 3 QuickHull

$T \leftarrow findInitialTetrahedron()$
$H \leftarrow faces(T)$ {obálka}
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**
  $f \leftarrow Q.pop()$
  $p \leftarrow mostDistantPointOutside(f)$
  $h \leftarrow findHorizon(p)$
  $newFaces \leftarrow \emptyset$
  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 faces $g \in newFaces$ **do**
    $assignPointsToFace(g)$
  end for
  for all face $g \in newFaces$ **do**
    $Q.push(g)$
  end for
end while

3.4.2 Analýza složitosti

Složitost algoritmu v prostoru $\mathbb{R}^3$ závisí, podobně jako například u algoritmu *QuickSort* [27], na pořadí přidávaných bodů. Stejně jako u něj může dosahovat složitosti $\mathcal{O}(n^2)$ v nejhorším případě, ale v průměrném případě dosahuje složitosti $\mathcal{O}(n \log n)$. Důkaz složitosti můžeme nalézt v původní práci Barbera a Dobkina [25].

3.5 Brute force

Metoda brute force (řešení hrubou silou) spočívá v kontrole všech možných trojic vstupních bodů jako stěn výsledné obálky. Algoritmus pro každou trojici bodů zkontroluje, zda všechny ostatní body leží na jedné straně roviny, které těmito body prochází (tzn. že pro všechny body je tato stěna viditelná nebo neviditelná). Pokud tomu tak je, znamená to, že tato stěna bude součástí výsledné obálky. Tento algoritmus pracuje s časovou složitostí $\mathcal{O}(n^4)$ a proto není vhodným řešením, pokud vstupní množina je větší než několik desítek bodů. Zároveň je třeba dodat, že takto implementovaná verze by tvořila pouze trojúhelníkové stěny a pro degenerovaný vstup by bylo potřeba přidat post-processing v podobě sloučení koplanárních stěn. Z důvodu velké časové složitosti algoritmu nebyl tento algoritmus implementován.

3.6 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 [28].

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, případně příslušných podílů. Tato optimalizace bude popsána dále u algoritmu *Jarvis March*.

4.1.1 Kompilátorové optimalizace

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

- `-O3`: Zapne veškeré možné optimalizace na úrovni překladače, mezi které patří například *loop unrolling* nebo vektorizace cyklů. Seznam všech obsažených optimalizací je možné nalézt v [29].

Ukázalo se, že přepínač `-O3` byl pro některé algoritmy velkou optimalizací mé implementace, když například algoritmus *Jarvis march* zrychlil téměř desetkrát u velkých vstupů.

4.2 Jarvis March

Stejně jako u 2D verze je vždy, když hledáme další stěnu obálky, je potřeba najít takovou, aby všechny ostatní vstupní body ležely na jedné straně roviny touto stěnou tvořené. V každém kroku zpracováváme jednu hranu (jednu stěnu, nazvěme ji f_1, která tuto hranu obsahuje, jsme již našli). Hledáme tedy bod takový, že stěna tvořená tímto bodem a krajními body zpracovávané hrany bude se stěnou f_1 svírat 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:

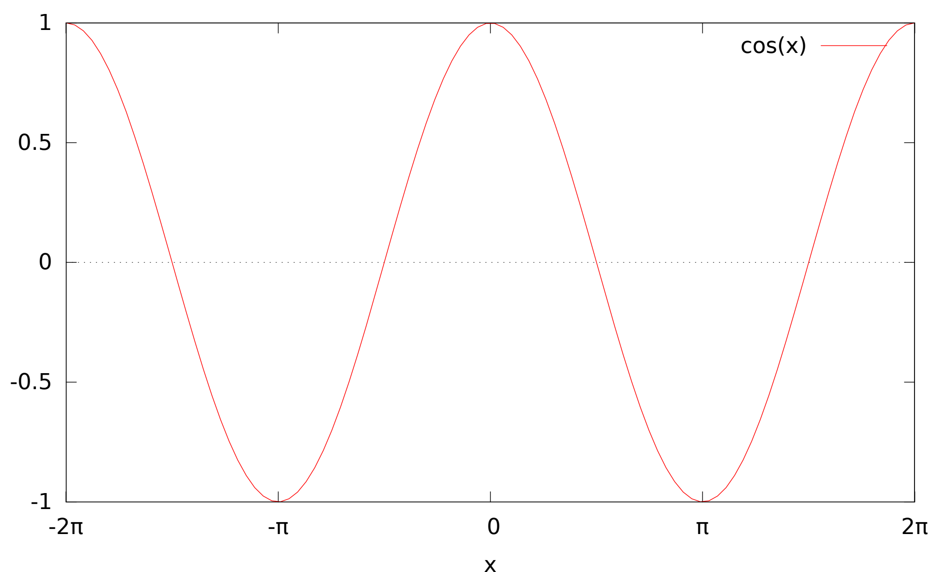
$$\cos \alpha = \frac{|n \cdot m|}{|n| \cdot |m|}$$

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 několik desítek až stovek 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.

4.3 Inkrementální algoritmus

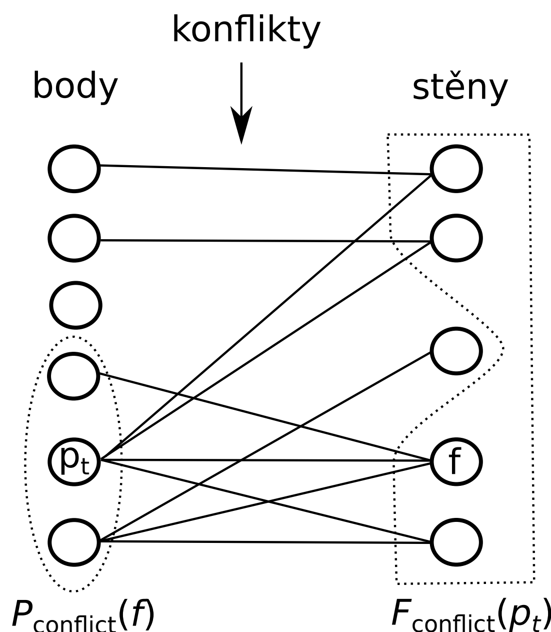
U inkrementálního algoritmu jsme si v kapitole 3 ukázali, že jeho složitost je $\mathcal{O}(n^2)$. Za použití vhodných datových struktur se ale tento algoritmus dá výrazně urychlit. Tento postup se nazývá *Randomizovaný inkrementální algoritmus* a byl představen v roce 1993 Raimundem Seidelem [30]. Kromě 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 $\mathcal{O}(n)$ stěn v každém kroku. To ovšem můžeme urychlit uchááváním 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 do obálky H_{i-1} vytvořené v minulé iteraci. 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

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

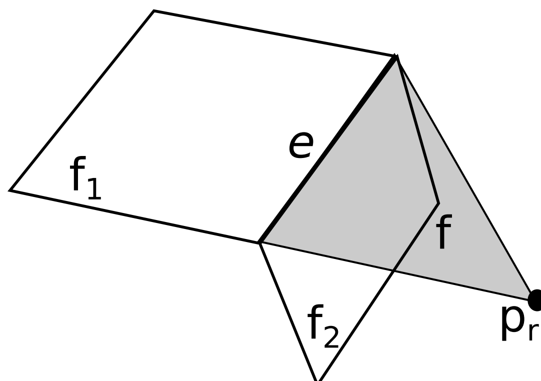
Věta 5. *Při přidávání nového bodu p_i je pro aktualizaci grafu konfliktů potřeba prohledat pouze body, které vidí stěny incidující s hranami tvořícími horizont bodu p_i.*

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. $\square$

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.

4.3.1 Složitost algoritmu

Složitost takto upraveného algoritmu bude očekávaná (v průměrném případě) $\mathcal{O}(n \log n)$. Složitost závisí na náhodné permutaci vstupu. Stejně jako například *QuickSort* má složitost $\mathcal{O}(n^2)$, jeho složitost v průměrném případě se uvádí také $\mathcal{O}(n \log n)$ a je závislá na permutaci prvků ve vstupní množině.

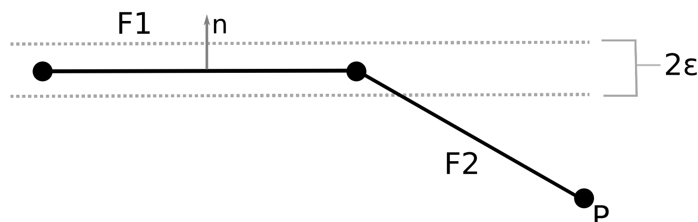


Obrázek 4.3: Přidání nové stěny

Důkaz složitosti randomizovaného inkrementálního algoritmu je možné nalézt například v [11, str. 250–252].

4.4 Fat Planes

Jak již bylo řečeno u popisu algoritmu *QuickHull* (3.4), 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 *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 4.4.



Obrázek 4.4: Fat planes

Nechť $F1$ je rovina s normálovým vektorem n, bod P ležící v prostoru a s

vzdálenost bodu P od roviny $F1$. Potom:

$$\begin{cases} s \; > \varepsilon \text{ a } F1 \text{ je viditelná z } P & P \text{ leží nad rovinou } F1 \\ s \; > \varepsilon \text{ a } F1 \text{ není viditelná z } P & P \text{ leží pod rovinou } F1 \\ s \; \leq \varepsilon & P \text{ je koplanární s } F1 \end{cases}$$

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 [25] o *QuickHullu* je uveden vzorec, který tento nedostatek odstraňuje a pracuje s maximálními absolutními hodnotami souřadnic vstupních bodů a strojovou odchylkou datového typu *double*:

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

4.5 Paralelizace

V této sekci ukážeme, jak je možné jednotlivé algoritmy paralelizovat. Všechny uvedené možnosti paralelizace byly implementovány v mé práci. Algoritmus *Brute force* jsem neimplementoval, proto o něm v této sekci nepíšu, nicméně jedna z možností paralelizace je využít stejný způsob, jaký je uveden u Inkrementálního algoritmu a algoritmu *QuickHull*.

4.5.1 Jarvis march

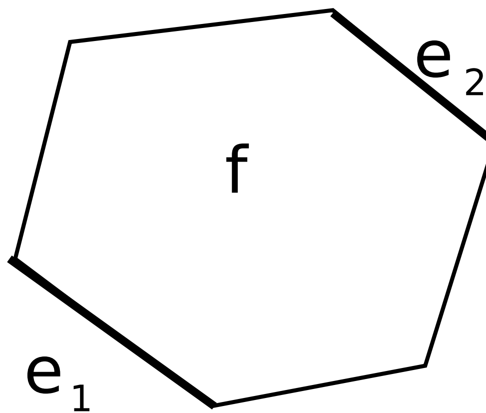
Na začátku algoritmu *Jarvis march* hledáme nějaký extrémní bod, u kterého budeme mít jistotu, že bude ležet na obálce. Těchto bodů je maximálně šest, jedná se o body s maximální a minimální souřadnicí v každé dimenzi. Potom pro každý extrémní bod je možné vytvořit vlastní vlákno, které bude obálku tvořit. Teoreticky je možné dosáhnout až lineárního zrychlení, pokud každé vlákno vytvoří stejně velkou část výsledné obálky.

Kromě množin *fresh* a *closed* budeme nyní pracovat i s množinou *opened*. Množiny *opened* a *closed* budou společné pro všechna vlákna, množinu *fresh* bude mít každé vlákno vlastní.

Na začátku algoritmu najdeme až šest extrémních bodů a tedy až šest stěn, které budou součástí výsledné obálky. Každé vlákno bude zpracovávat jednu z těchto stěn. Množinu *fresh* budou u každého vlákna na začátku tvořit hrany stěny, kterou vlákno zpracovává. Nyní popíši práci jednoho vlákna. Na začátku každé iterace vyjmeme z množiny *fresh* jednu hranu. Pokud tato hrana patří do sdílené množiny *closed*, znamená to, že ji už nějaké vlákno zpracovalo a známe již obě stěny, které tuto hranu obsahují. Nemusíme ji tedy nadále zpracovávat v tomto vlákně. Pokud do množiny *closed* nepatří, využijeme sekvenční algoritmus *Jarvis March* tak, jak je popsán v sekci 3.1. Tím najdeme další stěnu, která bude tvořit obálku. Pro každou hranu, nazvěme ji e, nově nalezené stěny zkontrolujeme, zda patří do množiny *opened*. Pokud

ano, znamená to, že už byla nalezena nějakým jiným vláknem, neboli že už na obálce existuje stěna, která tuto hranu obsahuje a nyní jsme našli stěnu k ní sousední. Proto tuto hranu uložíme do množiny *closed* a pokud je obsažena v lokální množině *fresh*, smažeme ji z ní. Pokud hrana e v množině *opened* neleží, uložíme ji tam a zároveň ji uložíme i do lokální množiny *fresh*. Takto pokračujeme v každém vlákně, dokud není ve všech vláknech množina *fresh* prázdná.

Jelikož může nastat situace, že bude nějaká stěna nalezena vícekrát (hlavně v případě, že máme na obálce stěny s více než třemi hranami), musíme na konci algoritmu zkontrolovat, zda tomu tak není a případné duplikáty z obálky odstranit. Důvodem vzniku těchto duplikátů je to, že stěna může být nalezena ve více různých vláknech, pokud v těchto vláknech zpracováváme zároveň dvě různé hrany, které na této stěně leží. Ilustrovat si tuto skutečnost můžeme na obrázku 4.5. Stěna f bude nalezena dvakrát, pokud ve dvou vláknech budou zároveň zpracovávány hrany e_1 a e_2.



Obrázek 4.5: Nalezení jedné stěny vícekrát

4.5.2 Algoritmus Divide and Conquer

Algoritmus *Divide and Conquer* v každém kroku dělí body do dvou množin a poté sestrojí obálku těchto množin bodů tak, že pokud je počet bodů v dané množině malý, využije nějaký již popsaný algoritmus, příp. metodu *brute force*, jinak tyto skupiny (již hotové obálky menších množin bodů) spojí tak, jak je popsáno v sekci 3.2. Nabízí se tedy možnost řešit vždy každou ze dvou množin bodů odděleně najednou. Můžeme využít mechanismu `task` z knihovny `OpenMP`. Vždy pro každou ze dvou množin bodů vytvoříme nový `task`, který nalezne obálku obou těchto množin způsobem popsaným výše. Ve chvíli, kdy máme hotové konvexní obálky obou těchto množin, můžeme je spojit. Takto rekurzivně vytvoříme obálku původní množiny všech bodů.

4.5.3 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, a proto jsem volil jiný způsob paralelizace, který vychází z algoritmu *Divide & Conquer* 3.2. Stejně jako v něm se na začátku body seřadí podle souřadnice x. Podle počtu vláken se poté seřazené body rovnoměrně rozdělí do tolika částí, kolik je vláken. Každé vlákno poté udělá pomocí Inkrementálního algoritmu obálku přidělené části bodů. Nakonec se všechny tyto části spojí stejným způsobem, jaký je uveden u algoritmu *Divide & Conquer* 3.2.

Jelikož může být vláken (a tedy i obálek ke spojení) několik, je pro urychlení možné paralelizovat i tato spojení, a to rekurzivním způsobem podobným tomu, který je uveden u algoritmu *Divide & Conquer*. Například pro 8 vláken spojuje jedno vlákno první dvě obálky, další druhé dvě atd. Až se všechny tyto obálky spojí, budou nám zbývat ke spojení již jen 4 větší obálky a ty spojíme stejným způsobem.

Další způsob, jak obálky spojit, je postupně je připojovat na například nejlevější obálku a tu tím zvětšovat. Tento postup ovšem není paralelní a tím pádem může být pomalejší.

Stejný způsob paralelizace byl použit i pro verzi algoritmu popsanou v 4.3.

4.5.4 QuickHull

Algoritmus *QuickHull* byl paralelizován stejným způsobem jako Inkrementální algoritmus.

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* (verze 2015.2) [17] a knihovnou CGAL (verze 4.11)[18]. Knihovna QHull implementuje algoritmus *QuickHull*, knihovna CGAL algoritmus *QuickHull* a Inkrementální algoritmus, obě knihovny implementují jejich sekvenční verze.

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 [31]. Pro dosažení co nejlepších časů byl kód kompilován s optimalizacemi na úrovni kompilátoru, uvedenými v sekci 4.1.

Před samotným testováním rychlosti bylo ještě potřeba otestovat, zda naše algoritmy pracují správně. K tomuto účelu byla implementována třída *Tester*, která testuje výstup jednotlivých algoritmů oproti výstupu z knihovny *QHull*. Tato třída implenentuje různé způsoby testování, například testování správnosti výstupů algoritmů, testování rychlosti algoritmů na různých typech dat a testování algoritmů na různém počtu vláken. Správnost všech algoritmů byla otestována na různých typech vstupních dat, konkrétně na vstupech obsahujících body různě vzdálené od povrchu krychle a koule pro různé počty bodů a různé velikosti koule a krychle. Zároveň byly na těchto datech otestovány i paralelní verze algoritmů.

5.1 Testovací data

V rámci testování jsem používal čtyři druhy testovacích dat. Ty byly generovány pomocí nástroje `rbox` z knihovny *QHull* [17]. 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 náhodně ve 100y% 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 na povrchu krychle.
- **Testovací data č. 4**: Body ležící na povrchu koule.

5.2 Celkové srovnání

5.2.1 Testované algoritmy

V mé práci jsem kromě algoritmu *Brute force* implementoval všechny algoritmy popsané v kapitole 3. Implementovány byly tak, jak jsou popsány v této kapitole. U Inkrementálního algoritmu byla implementována i jeho optimalizovaná verze popsaná v sekci 4.3. Všechny algoritmy byly dále paralelizovány způsobem popsaným v sekci 4.5. Jelikož algoritmy *QuickHull* a *Jarvis March* dosahují ve většině případů výrazně větší rychlosti než ostatní, budu u každého typu dat ukazovat srovnání všech algoritmů na menším počtu vstupních bodů a následně srovnání těchto rychlejších algoritmů na větším množství vstupních bodů.

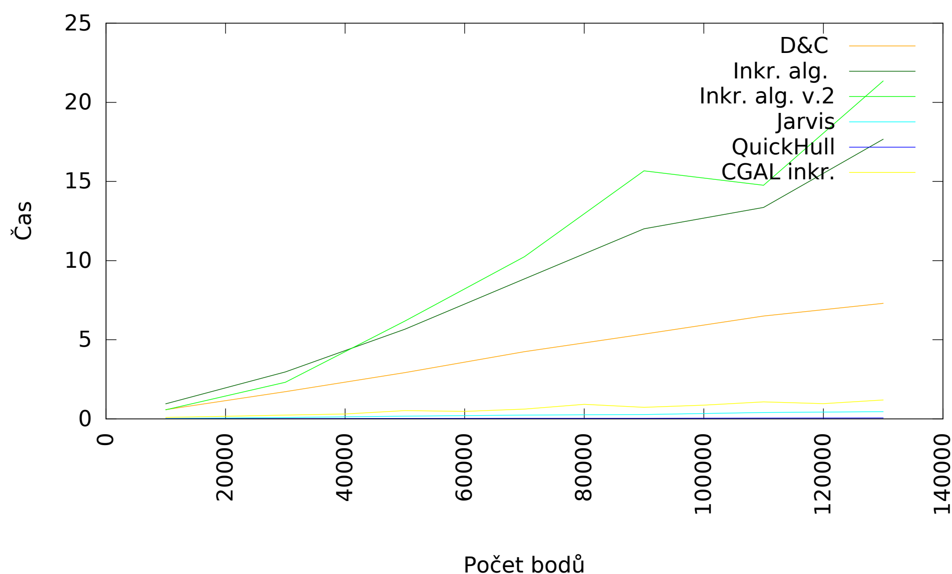
- **Jarvis March** (v grafech jako *Jarvis*) implementován tak, jak je popsán v 3.1 za využití optimalizací popsaných v 4.2 a paralelizován tak, jak je popsáno v 4.5.1.
- **Divide & Conquer** (v grafech jako *D&C*) implementován tak, jak je popsán v 3.2 a paralelizován tak, jak je popsáno v 4.5.2.
- **Inkrementální algoritmus** (v grafech jako *Inkr. alg.*) implementován tak, jak je popsán v 3.3 a paralelizován tak, jak je popsáno v 4.5.3.
- **Optimalizovaný Inkrementální algoritmus** (v grafech jako *Inkr. alg. v.2*) implementován tak, jak je popsán v 4.3.

- **QuickHull** implementován tak, jak je popsán v 3.4 a paralelizován tak, jak je popsáno v 4.5.4.

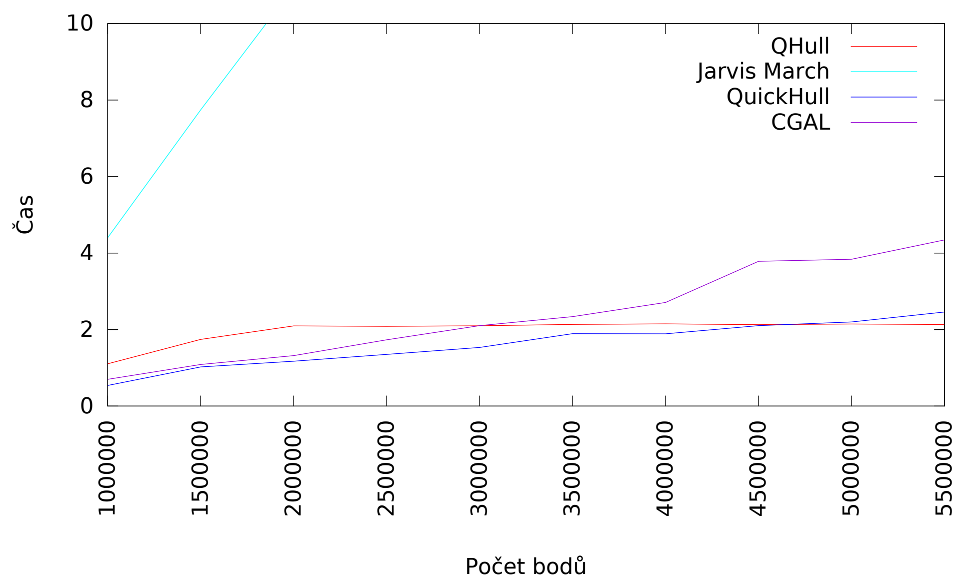
Nyní ukážeme rychlost sekvenčních verzí algoritmů. V grafech bude *CGAL* značit algoritmus *QuickHull* implementovaný v knihovně CGAL a *CGAL inkr.* Inkrementální algoritmus implementovaný v knihovně CGAL.

5.2.2 Testovací data č.1

Srovnání rychlosti běhu algoritmů na testovacích datech č.1 (body ležící uvnitř krychle) ilustrují obrázky 5.1 a 5.2.



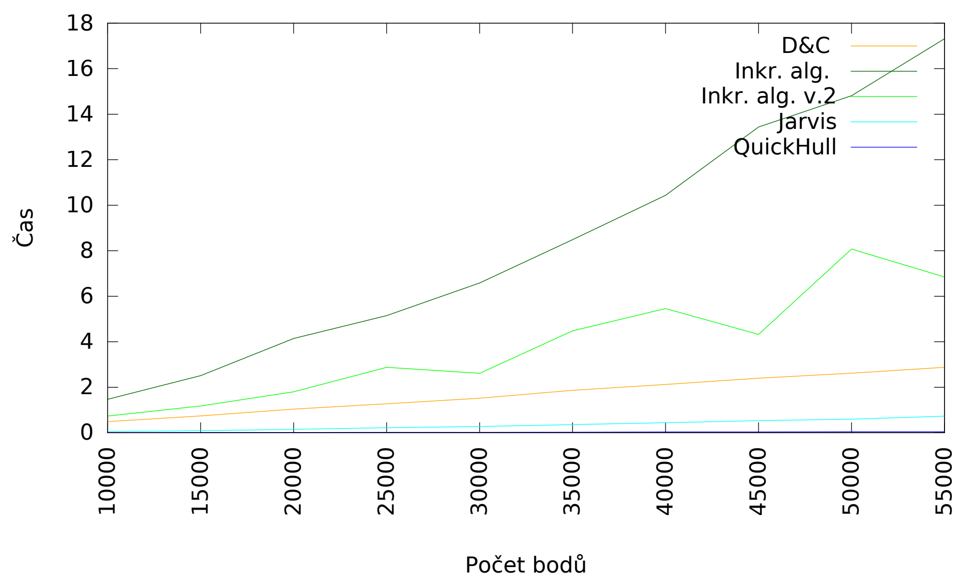
Obrázek 5.1: Srovnání všech algoritmů pro testovací data č. 1.



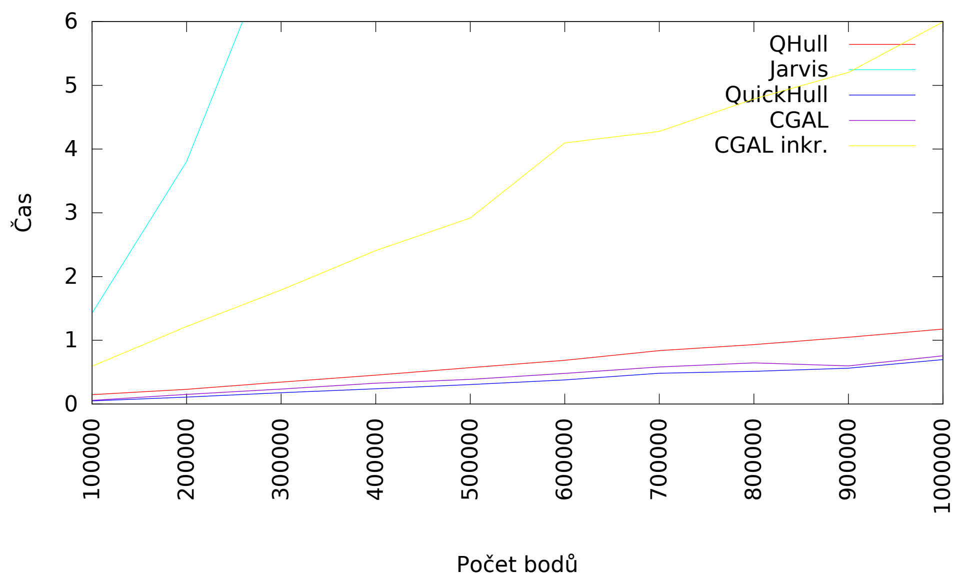
Obrázek 5.2: Srovnání rychlejších algoritmů pro testovací data č. 1.

5.2.3 Testovací data č.2

Srovnání rychlosti běhu algoritmů na testovacích datech č.2 (body ležící uvnitř koule) ilustrují obrázky 5.3 a 5.4.



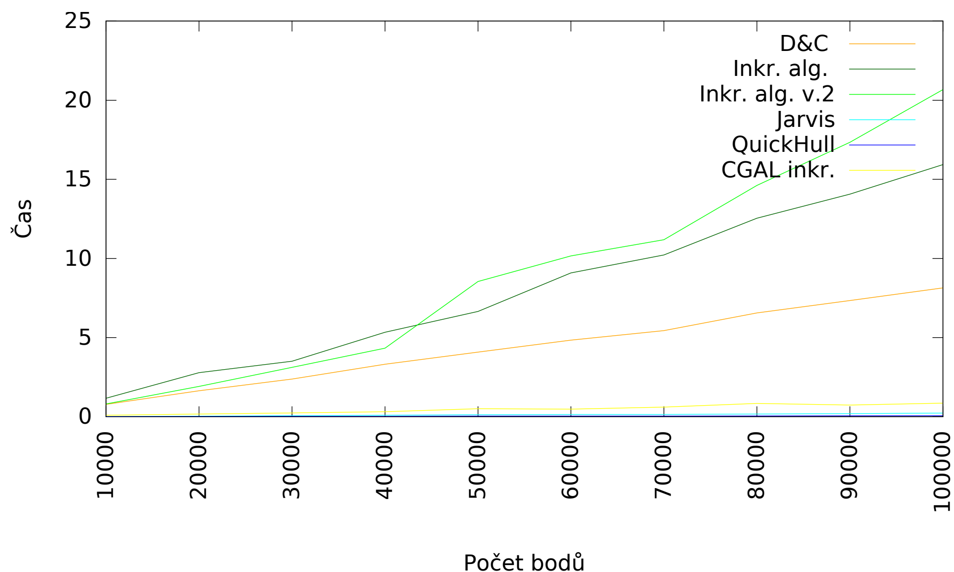
Obrázek 5.3: Srovnání všech algoritmů pro testovací data č. 2.



Obrázek 5.4: Srovnání rychlejších algoritmů pro testovací data č. 2.

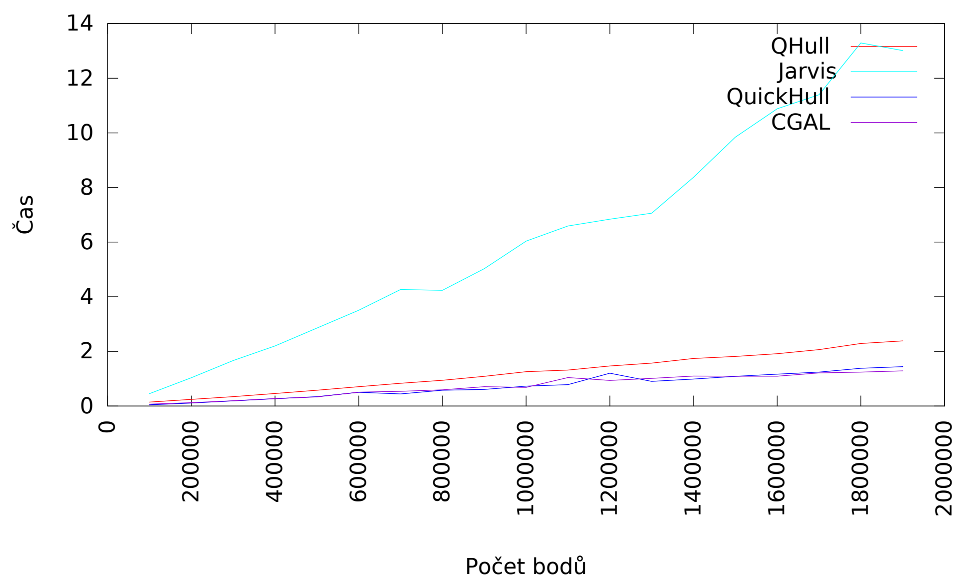
5.2.4 Testovací data č.3

Srovnání rychlosti běhu algoritmů na testovacích datech č.3 (body ležící na povrchu krychle) ilustrují obrázky 5.5 a 5.6.



Obrázek 5.5: Srovnání všech algoritmů pro testovací data č. 3.

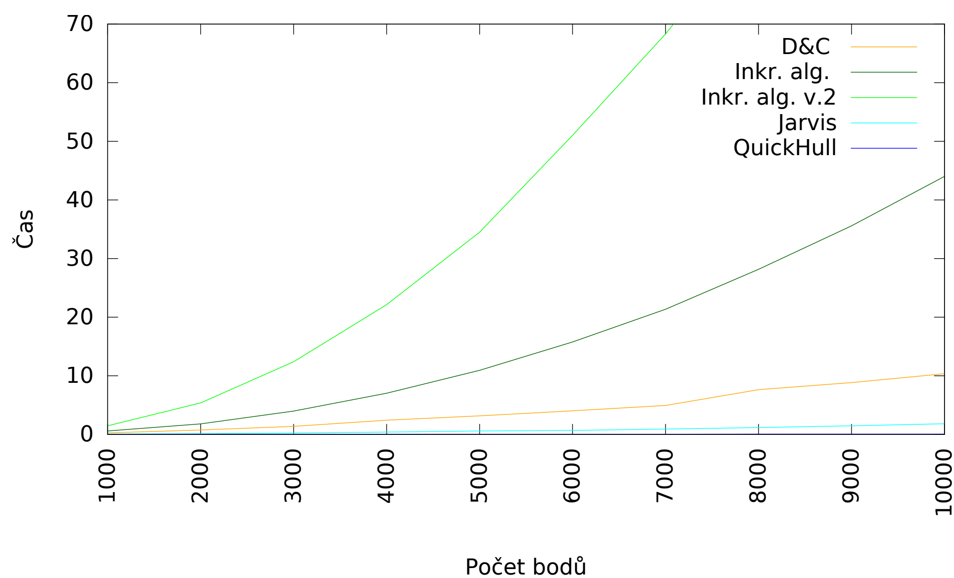
Pro 100 000 bodů Jarvis March: 0.227928s, QuickHull: 0.066343s



Obrázek 5.6: Srovnání rychlejších algoritmů pro testovací data č. 3.

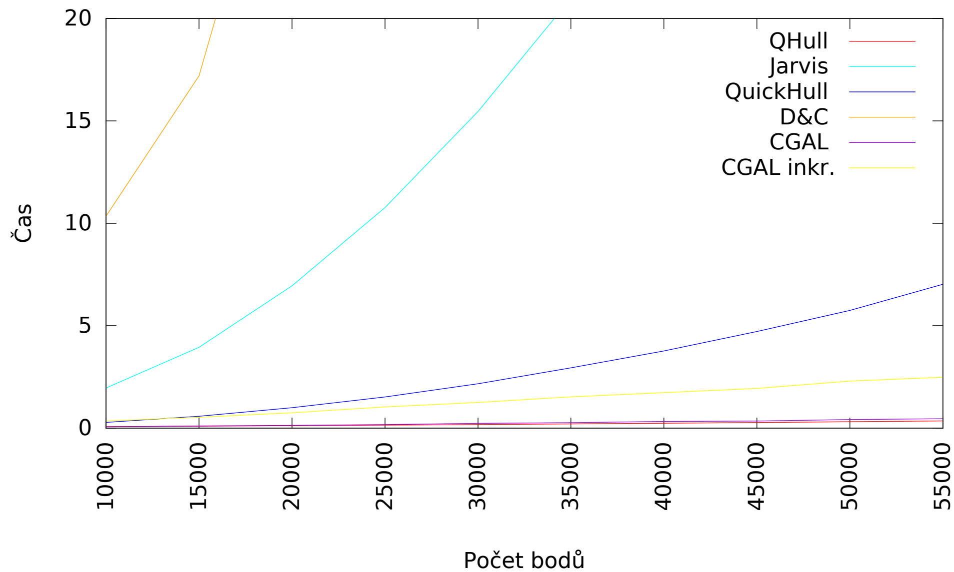
5.2.5 Testovací data č.4

Srovnání rychlosti běhu algoritmů na testovacích datech č.4 (body ležící na povrchu koule) ilustrují obrázky 5.7 a 5.8.



Obrázek 5.7: Srovnání všech algoritmů pro testovací data č. 4.

Pro 10 000 bodů QuickHull: 0.0428294s



Obrázek 5.8: Srovnání rychlejších algoritmů pro testovací data č. 4.

Pro 55 000 bodů QHull: 0.35s, CGAL: 0.45s

5.2.6 Vyhodnocení

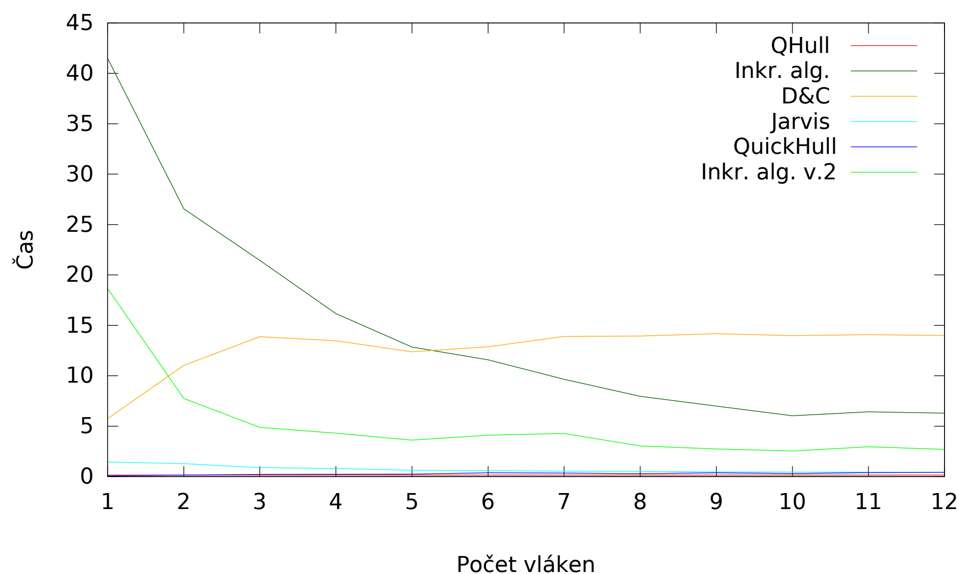
U všech typů dat se ukazuje jako nejrychlejší algoritmus *QuickHull*. To potvrzuje i rychlost tohoto algoritmu implementovaného v knihovnách *QHull* a CGAL. Algoritmus *Jarvis March* se v mé implementaci ukazuje jako druhý nejrychlejší, následovaný algoritmem *Divide & Conquer*. Jako nejpomalejší se naopak ukazují obě verze Inkrementálního algoritmu, kdy jeho optimalizovaná verze je občas pomalejší, než verze obyčejná, což si vysvětluji vysokou režií správy grafu konfliktů a nevyužití *halfedge* struktury v mé implementaci tohoto algoritmu. Ze stejného důvodu vidím výrazně výrazně vyšší rychlost Inkrementálního algoritmu implementovaného v knihovně CGAL oproti mé implementaci.

5.3 Paralelizace

Nyní zbývá ukázat, jak úspěšná byla paralelizace jednotlivých implementovaných algoritmů. Pro srovnání jsem výsledky měřil na dvou typech dat, a to na bodech ležících uvnitř koule a bodech ležících na jejím povrchu. Důvod výběru těchto dvou datových sad je různorodost počtu stěn na výsledné obálce. Zatímco u bodů ležících uvnitř koule bude počet stěn relativně malý (např. pro 10000 vstupních bodů vznikne okolo 500 stěn, pro 100000 vstupních bodů okolo 1600 stěn), u bodů ležících na povrchu koule je počet výsledných stěn

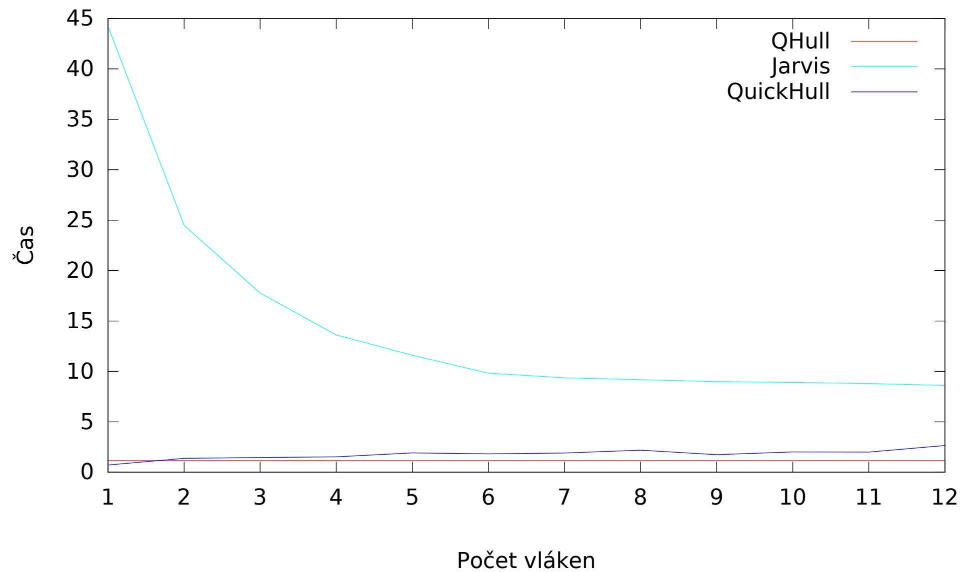
výrazně vyšší (pro 10000 vstupních bodů přibližně 20000 stěn, pro 100000 vstupních bodů okolo 200000 stěn).

Pro první typ dat, tedy body ležící uvnitř koule, byla paralelizace velmi efektivní u Inkrementálního algoritmu a algoritmu *Jarvis March*, jak ilustrují obrázky 5.9 a 5.10. U algoritmu *QuickHull* dochází se zvyšujícím počtem jader k mírnému poklesu rychlosti, které je způsobené dobou trvání sloučení dvou obálek. Stejně tak je tomu i u algoritmu *Divide & Conquer*, kde je zhoršení způsobeno nejspíše i režií `tasků`.



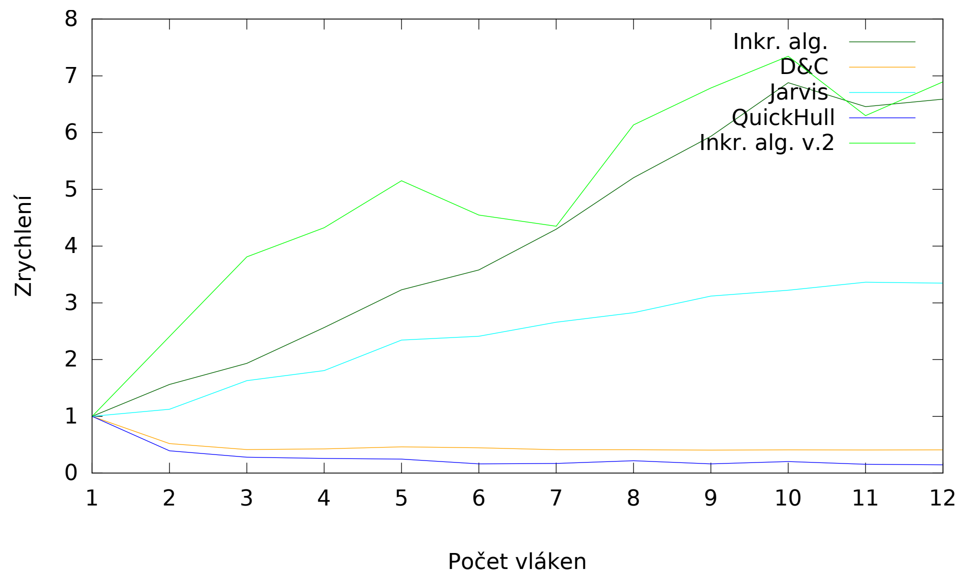
Obrázek 5.9: Srovnání všech paralelních algoritmů pro 100 000 bodů ležících uvnitř koule.

Pro 12 jader QuickHull: 0.412s, Jarvis March 0.432s, sekvenční QHull 0.147s

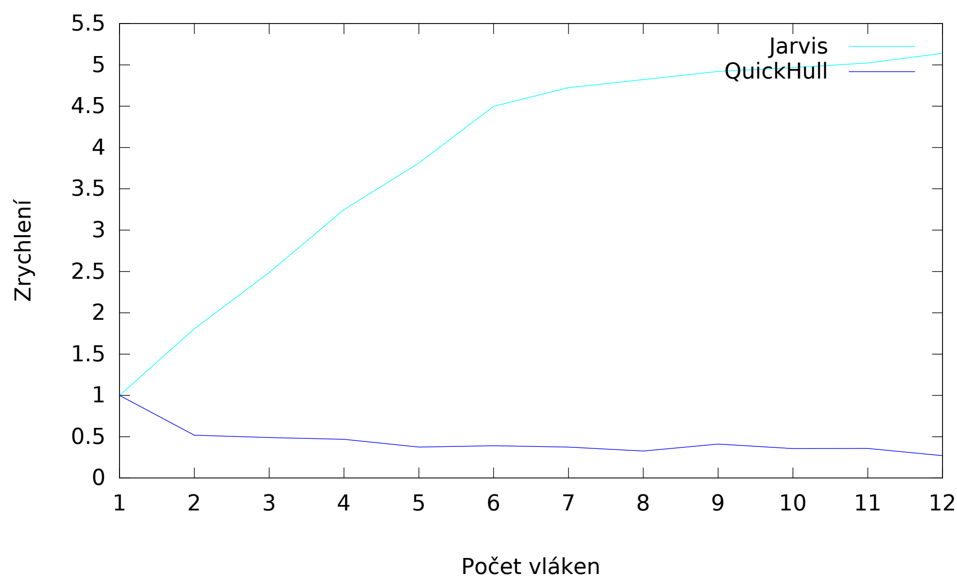


Obrázek 5.10: Srovnání rychlejších paralelních algoritmů pro 1 000 000 bodů ležících uvnitř koule.

Pro lepší ilustraci zrychlení dále uvádím ještě grafy zrychlení jednotlivých paralelní algoritmů v závislosti na počtu jader, konkrétně se jedná o obrázky 5.11 a 5.12.

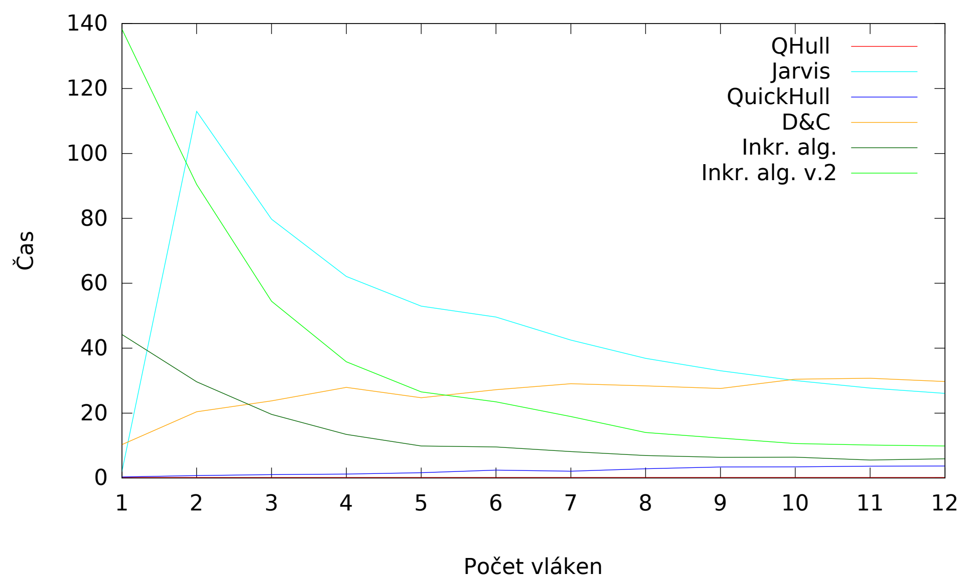


Obrázek 5.11: Srovnání zrychlení všech paralelních algoritmů pro 100 000 bodů ležících uvnitř koule.



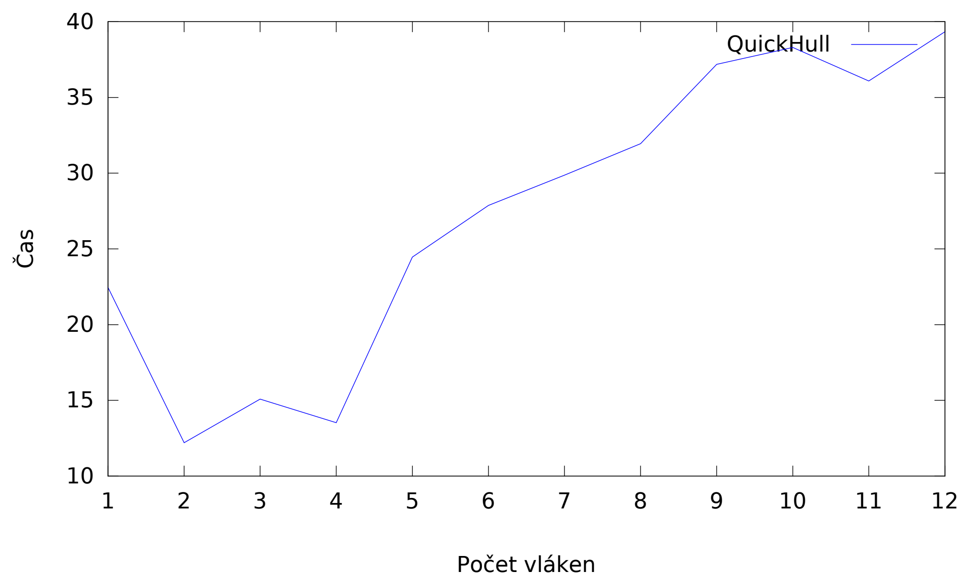
Obrázek 5.12: Srovnání zrychlení rychlejších paralelních algoritmů pro 1 000 000 bodů ležících uvnitř koule.

Výsledky pro druhý typ dat (body ležící na povrchu koule) ilustrují obrázky 5.13 a 5.14. Stejně jako u přechozího typu dat dochází ke zrychlení i Inkrementálního algoritmu a pro malé množství jader i u algoritmu *QuickHull*. Pro 5 a více jader zabere spojení obálek více času, než kolik paralelizace ušetří, proto celkový čas roste. Narozdíl od předchozího typu dat je ovšem velmi výrazné zhoršení u algoritmu *Jarvis March*, které je způsobeno kontrolou duplikátních stěn na konci algoritmu. Pro více jader potom doba výpočtu klesá, což je způsobeno paralelizací právě kontroly duplikátů, nicméně i na velkém množství jader je algoritmus stále o mnoho pomalejší než jeho sekvenční verze.



Obrázek 5.13: Srovnání všech paralelních algoritmů pro 10 000 bodů ležících na povrchu koule.

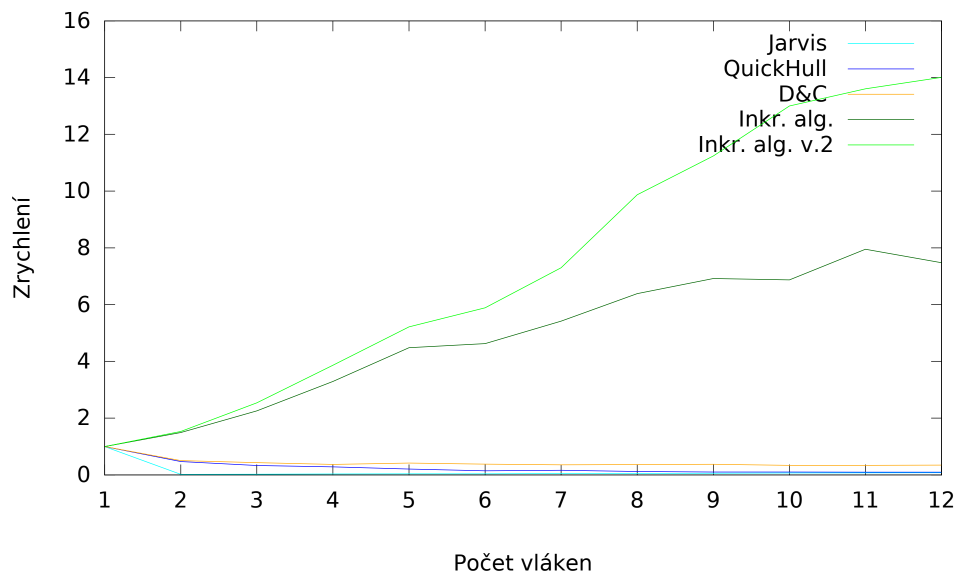
Sekvenční QHull: 0.17s



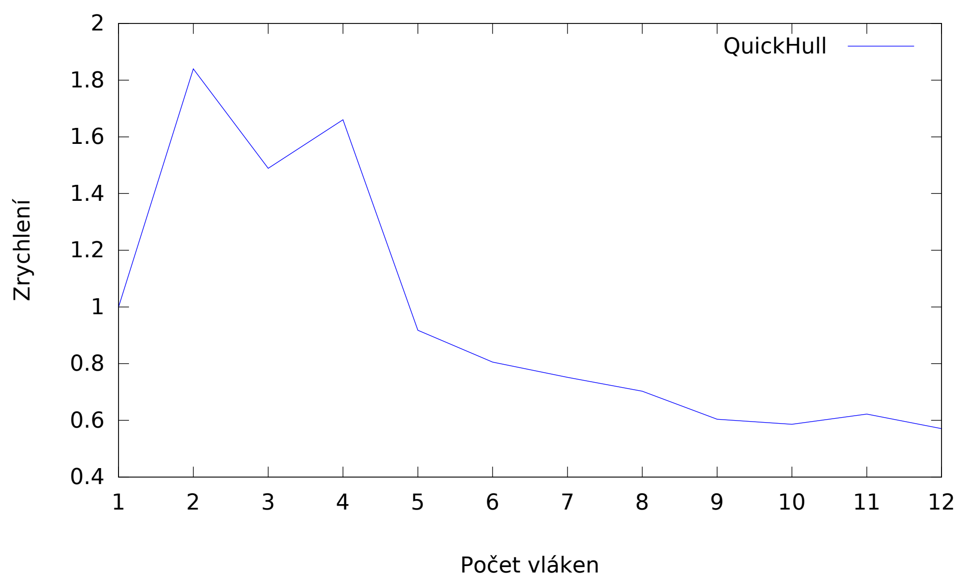
Obrázek 5.14: Doba trvání paralelního algoritmu QuickHull pro 100 000 bodů ležících na povrchu koule.

Stejně jako u předchozího typu dat pro ilustraci uvádím dále grafy zrych-

lení jednotlivých algoritmů (obrázky 5.15 a 5.16).



Obrázek 5.15: Srovnání zrychlení všech paralelních algoritmů pro 10 000 bodů ležících na povrchu koule.



Obrázek 5.16: Zrychlení paralelního algoritmu QuickHull pro 100 000 bodů ležících na povrchu koule.

Z předchozích grafů lze tedy o mnou implementovaných algoritmech říci

následující:

U algoritmu *Jarvis March* a vstup takový, že výsledná obálka má málo stěn, je nejrychlejší jeho paralelní verze běžící na větším počtu vláknech (ideálně 6 a více, zde už pak není rozdíl velký). Pokud ovšem obálka bude stěn obsahovat hodně, pak výrazně nejrychlejší je naopak jeho sekvenční verze.

U algoritmu *Divide & Conquer* je nejrychlejší jeho sekvenční verze.

U Inkrementálního algoritmu je rychlejší jeho paralelní verze a (alespoň pro 12 a méně vláken) platí, že čím více vláken, tím je algoritmus rychlejší.

U algoritmu *QuichHull* a vstup takový, že výsledná obálka má málo stěn, je sekvenční verze rychlejší než paralelní. Naopak pro vstup takový, že výsledná obálka má stěn hodně, je rychlejší paralelní verze algoritmu, ideálně pro malý počet vláken (2–4).

Závěr

V této práci jsem v souladu s prvním cílem implementoval 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.

Konkrétně bylo implementováno pět algoritmů, a to: algoritmus *Jarvis March*, algoritmus *Divide & Conquer*, dvě verze Inkrementálního algoritmu a algoritmus *QuickHull*.

Hlavním výsledkem mé práce je konstatování, že ze pěti mnou implementovaných algoritmů, pracuje algoritmus *QuickHull* (sekvenční i paralelní verze) 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 [17] a CGAL [18].

Rychlost všech algoritmů se ukázala závislá na počtu vstupních bodů, které leží na výsledné obálce. Nejvíce tímto ovlivněný je Inkrementální algoritmus (který zřejmě zpomaluje hlavně režie přidávání nových a odebírání již neplatných stěn), nejméně ovlivněný je naopak algoritmus *QuickHull*.

V souladu s dalším cílem byly všechny mnou implementované algoritmy paralelizovány. Pro některé typy vstupních dat se paralelizace ukázala jako účinný nástroj pro zrychlení těchto algortimů, naopak pro některé se ukázala jako výrazně zpomalující faktor.

Literatura

[1] Maur, P.: *Delunay Triangulation in 3D.* Disertační práce, Plzeň: Západočeská univerzita v Plzni, Katedra informatiky a výpočetní techniky, 2002.

[2] van Kreveld, M.: Voronoi diagrams. Dostupné z `http://www.cs.uu.nl/docs/vakken/ga/slides7.pdf`, [Online; citováno 7.1.2018].

[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.; Overmars, M.: *Computational Geometry: Algorithms and Applications*. Santa Clara: Springer-Verlag, třetí vydání, 2008, ISBN 978-3540779735.

[12] 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].

[13] Barto, L.: Afinní a euklidovský prostor. Dostupné z `http://www.karlin.mff.cuni.cz/~barto/student/Afineuklid.pdf`, [Online; citováno 7.1.2018].

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

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

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

[17] Qhull 2015.2 [software]. Dostupné z `http://qhull.org`, [Citováno 24.4.2017].

[18] The CGAL Project: *CGAL User and Reference Manual*. CGAL Editorial Board, 4.11 vydání, 2017. Dostupné z: `https://doc.cgal.org/latest/Manual/index.html`

[19] McGuire, M.: The Half-Edge Data Structure. Dostupné z `http://ldc.usb.ve/~vtheok/cursos/ci6322/articulos/half_edge.pdf`, 2000, [Online; citováno 2.1.2018].

[20] 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].

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

[22] 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.

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

[24] Agarwal, P. K.: Convex Hulls in Higher Dimensions. Dostupné z https://www.cs.duke.edu/courses/fall05/cps234/notes/lecture04.pdf, [Online; citováno 7.12.2017].

[25] 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.

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

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

[28] 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].

[29] Options That Control Optimization. Dostupné z https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [Citováno 7.1.2018].

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

[31] Free Software Foundation, I.: GCC (GNU Compiler Collection) 4.8.5 [software]. Dostupné z https://gcc.gnu.org/, [Citováno 21.12.2017].

Příloha A

Seznam použitých zkratek

STL Standard Template Library

CGAL The Computational Geometry Algorithms Library

OpenMP Open Multi-Processing

D&C Divide & Conquer

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. Pro správné fungování testování algoritmů (parametr `-t`, viz dále) je potřebné mít nainstalovanou knihovnu `qhull` a z ní dostupný program `rbox`.

B.2 Instalace a použití

Program je možné zkompilovat spuštěním příkazu `make` v kořenovém adresáři projektu – `ConvexHull`. V adresáři `bin` se vygeneruje spustitelný soubor `convex`. Ten je možné spustit jednoduše z kořenového adresáře příkazem `./bin/convex`.

Při spuštění programu je možné použít následující přepínače:

- `-h` Nápověda k použití programu.
- `-t` Testování algoritmů.
- `-a [alg]` Vynutí použití algoritmu `alg`. `alg` může nabývat hodnot
  - `a`: Všechny algoritmy
  - `j`: Jarvis March
  - `d`: Divide & Conquer
  - `i`: Inkrementální algoritmus

  - `io`: Optimalizovaný Inkrementální algoritmus
  - `q`: QuickHull

- `-p [thr]` Využití paralelních verzí algoritmů s `thr` vlákny.

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 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
    Makefile .................... zdrojový soubor pro nástroj make
thesis .......................... text práce
    src ......................... zdrojové kódy práce
    BP_Motyka_Vaclav_2018.pdf ... text práce ve formátu PDF
```