Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Konvexní obálka rozsáhlé množiny bodů v E^d

Plzeň, 2010 Petr Martínek

Čestné prohlášení

Prohlašuji, že jsem bakalářskou práci na téma: Konvexní obálka rozsáhlé množiny bodů v E^d vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 6.května 2010 Petr Martínek

Abstract

The purpose of my bachelor thesis was to write a program which calculates *Convex hull of large data set in E^d*, which is also the official name of this work. I had to get familiar with the concept of convex hull and with its construction. After studying available algorithms I chose *QuickHull*, because it was easier to implement for N dimensions than the others. During my work I also solved the problem how to work with very large data sets. Implementing QuickHull algorithm for multiprocessor parallel run was a way how to create the convex hull faster.

Obsah

Číslo kapitoly a název		strana
1.	Úvod	6
2.	Teoretická část	7
2.1.	Konvexní obálka	7
2.2.	Využití konvexní obálky	7
2.3.	Možnosti výpočtu	8
2.3.1.	Graham scan	8
2.3.2.	Inkrementální konstrukce	9
2.3.3.	Jarvis scan	10
2.3.4.	Divide and Conquer	11
2.3.5.	Quick hull	12
2.4.	Paralelní výpočet – Vlákna	14
2.4.1.	Úvod do vláken	14
2.4.2.	Jak vlákna fungují	16
2.4.3.	Synchronizace	17
2.4.4.	Stavy vláken	17
3.	Praktická část	19
3.1.	Metoda první: Sklápění	19
3.2	Metoda druhá: Quick hull	20
3.3.	Moje implementace Quick hull	21
3.4.	Moje paralelní zpracování	23
3.5.	Řešení nedostatku paměti	23
4.	Zhodnocení výsledků	23
4.1.	Vliv počtu vláken na dobu výpočtu	26
4.2.	Zvláštní případy	29
5.	Závěr	31

1. Úvod

S rozmachem počítačové vědy a hlavně oborů spojených s vizualizací, analýzou a grafikou. Vzrůstá i potřeba výpočtů různých objektů. Jedním z takových objektů je i konvexní obálka, která se využívá například k analýze tvarů objektů nebo analýze shluků, blíže uvedeno v kapitole dvě. Z tohoto důvodu jsem si také jako bakalářskou práci vybral naprogramovat program, který provede výpočet konvexní obálky. S postupem času vzrost počet metod, kterými lze konvexní obálku vypočítat, v dnešní době jich je známých něco okolo padesáti. V mojí práci se pokusím seznámit čtenáře s několika možnými způsoby výpočtu, které začínám rozebírat od kapitoly 2.3. Dále se pak pokusím blíže zasvětit do dvou algoritmů, oba rozebírám v praktické části práce - kapitola tři. Prvním je algoritmus sklápění, který jsem si sám vymyslel. Druhý je pak již známý algoritmus Quick hull.

Další věc, která se v poslední době hodně rozmáhá, je používání více jádrových procesorů. Téměř každý již má doma alespoň dual core. Tato skutečnost vyzývá k použití paralelních výpočtů. Proto se i můj zájem ubíral tímto směrem a kvůli urychlení výpočtu jsem do programu zařadil i paralelní zpracování. Seznámení s paralelizací je provedeno v kapitole 2.4 a samotná moje realizace je popsána v kapitole 3.4.

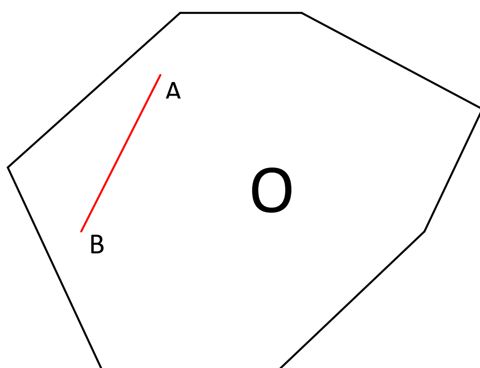
2. Teoretická část

2.1. Konvexní obálka

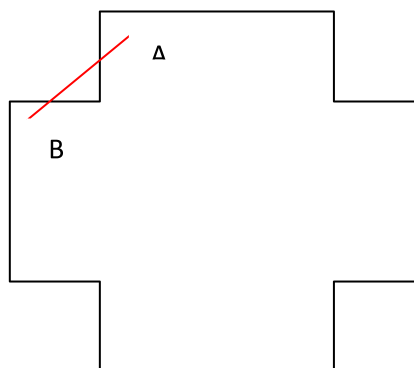
Níže uvedené definice konvexní obálky a možností konstrukce jsou nastudované podle [Bay2009].

Konvexní obálkou nazýváme množinu bodů, která splňuje následující vlastnosti.

- Minimální ohraničení určité množiny bodů.
- Žádný z bodů množiny nesmí být mimo obálku.
- Pokud spojíme libovolné dva body z množiny úsečkou, potom se musí celá úsečka nacházet uvnitř naší obálky. To znamená, že pro obálku O platí, že pro všechna a,b ∈ O je splněna podmínka: $\overline{ab} := \{\gamma a + (1-\gamma)b \mid 0 \leq \gamma \leq 1\} \subseteq O$.



Obr. 1 - Konvexní obálka O



Obr. 2 - Není konvexní obálka

2.2. Využití konvexní obálky:

Konvexní obálka je jedna z nejpoužívanějších geometrických struktur, pomocná struktura pro řadu algoritmů. Často se používá jako první odhad tvaru nějakého prostorového jevu.

Příklady použití:

- Detekce kolizí (např. při pohybu robotů).

- Využití v kartografii pro detekci natočení budov a jejich tvaru.
- Analýza tvarů objektu.
- Statistická analýza (analýza rozptylů, odhady atd.)
- Analýza shluků.

2.3. Možnosti výpočtu (konstrukce):

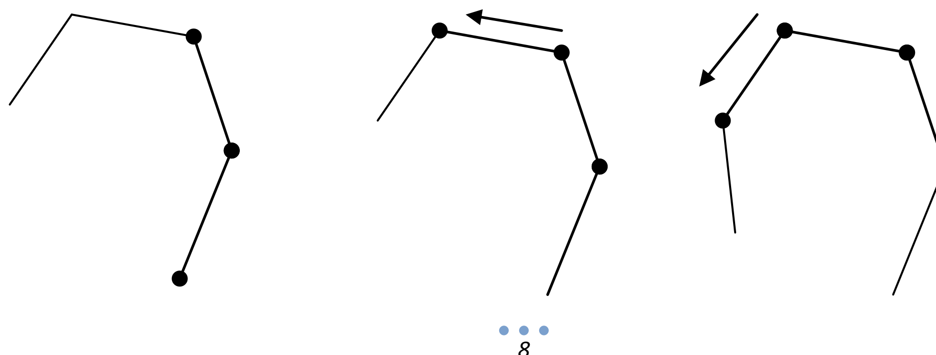
Možností jak konvexní obálky vytvořit je mnoho (např. Graham Scan, Divide and Conquer, Inkrementální konstrukce, Jarvis Scan, …..). Mnoho druhů výpočtů spočívá v jejich základních odlišnostech. Každý z algoritmů se díky odlišným paměťový a časovým nárokům značně liší a z toho vyplívá i odlišnost jejich použití. Krom toho se od sebe algoritmy dosti liší svojí robustností. Je například zbytečné pro výpočet obálky ve 2D z tisíce bodů pomocí algoritmu Quick hull, když nám tu samou službu vykoná i Jarvis scan, který naprogramovat je podstatně jednoduší. Já se zde budu věnovat dvěma metodám. Obě tyto metody jsem si sám zkusil naprogramovat a díky tomu jsem se s nimi i blíže seznámil. Nejprve se alespoň částečně pokusím nastínit, jak výše uvedené metody fungují.

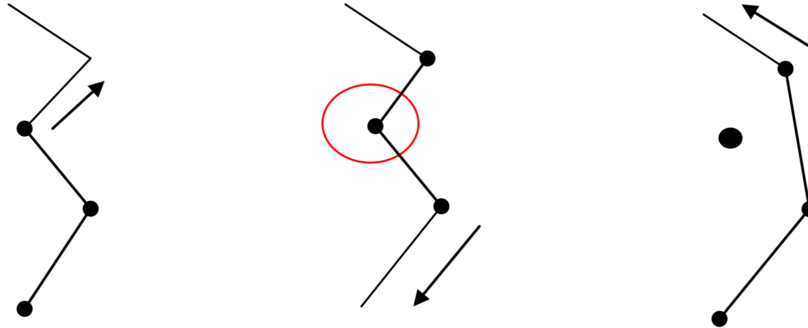
Pro popis algoritmů využiji množinu S v 2D, která obsahuje n bodů. R bude podmnožina S. Konvexní obálku označuji jako H.

2.3.1. Graham Scan

Pracuje takto: Každá uspořádaná trojice bodů p_{j-1},p_j,p_{j+1} Є H a musí splňovat kritérium levotočivosti. Pokud toto kritérium platí, body mohou ležet na obálce H (ale nemusí, je to pouze splněna první část podmínky), musíme ještě otestovat následující trojici bodů. Pokud kritérium nesplňují, prostřední vrchol odstraníme (nemůže ležet na H) a testujeme trojici p_{j-2},p_{j-1},p_{j+1}. Bod p je doplněn dvěma nejbližšími předchůdci.

Levotočivost znamená, že každý další bod, který nalezneme, se nachází vlevo od již nalezené stěny obálky, která je tvořena dvěma posledními body AB. Takový jednoduchý způsob jak levotočivou zjistit, je proložit si orientovanou úsečku AB přímkou (úsečka AB představuje spojnici dvou naposledy zařazených bodů do obálky), a otestovat, zda má nově nalezený bod stejné znaménko vzdálenosti od dané přímky jako ostatní body, které jsou již v obálce zařazeny.





Obr. 3 - Příklady levotočivosti

Veliké mínus tohoto algoritmu je jeho neschopnost rozšíření do vyšší dimenze, než je 2D. Z tohoto důvodu jsem si ho nezvolil pro svůj výpočet.

Naopak výhodou tohoto algoritmu je, že má nízkou časovou složitost O(n * log(n)), tudíž ho lze použít i na rozsáhlé množiny bodů.

2.3.2. Inkrementální konstrukce

Princip konstrukce: Body z S jsou přidávány po jednom do vytvářené konvexní obálky H, jejíž tvar je modifikován. Nechť R představuje podmnožinu bodů S obsahující m bodů a p přidávaný bod, pak

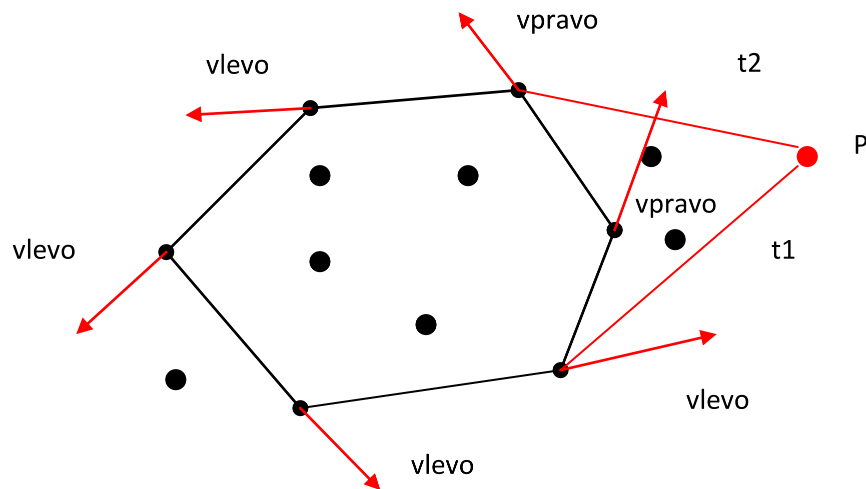
$$H_{m+1} = H_m \cap p.$$

Při přidání bodu p do H_m, mohou nastat dvě situace.

1. $p \in H_m$
   Bod p může být zanedbán, neovlivní tvar H_{m+1}. Test polohy p vzhledem k H_m je realizován prostřednictvím Half Edge testu.

2. $p \notin H_m$
   Bod p ovlivní tvar H_{m+1}.Je nutno najít horní a dolní tečny t1 a t2 procházející bodem p a kolmé k H_m.

Half edge test je druh testu, kterým určíme, jestli je nalezený bod vlevo nebo vpravo od orientované úsečky. Něco na způsob levotočivosti.



Obr. 4 - Ukázka nalezení tečen

(t2 – horní tečna, t1 – dolní tečna)

Nechť součastné body obálky jsou označeny k_i.

Pro t1 platí: bod P je vlevo od k_i a vpravo do k_{i+1}.

Pro t2 platí: bod P je vpravo od k_i a vlevo od k_{i+1}.

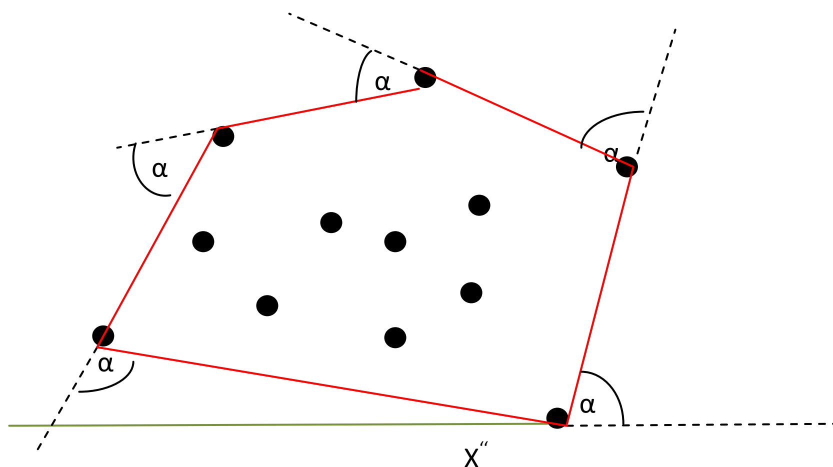
Po přiřazení bodu P do obálky, odstraníme současné stěny mezi dotykovými body tečen.

Výhodou tohoto algoritmu je rychlá konstrukce a složitost O(n*log(n)). Algoritmus lze převést i do vyšších dimenzí.

2.3.3. Jarvis Scan

Princip tohoto algoritmu je často popisován k balení dárku. Předpokladem pro algoritmus Jarvis scan je, že nesmí ležet 3 body na jedné přímce.

Princip algoritmu: Nechť p_{j-1},p_j představují dva poslední body množiny H a bod p_{j+1} je aktuálně nalezený bod přidávaný do H. Bod p_{j+1} přitom musí splňovat podmínku minimálního úhlu α mezi stranou p_{j+1} p_j a stranou p_{j-1} p_j . Bod p_j hledáme mezi všemi body, které již netvoří obálku H.



Obr. 4 - Ukázka Jarvis scanu

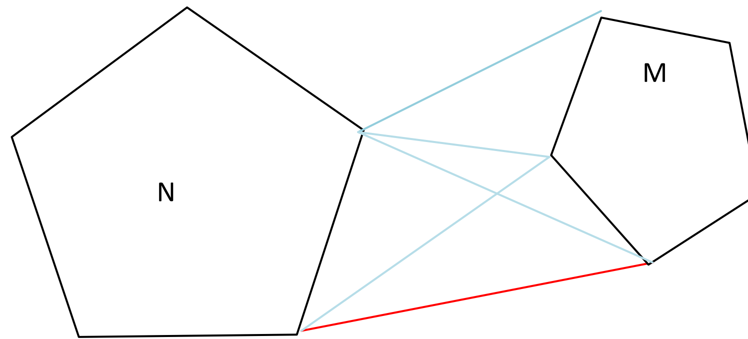
Tento algoritmus je lehce implementovatelný a jednoduše jde rozšířit do 3D. Do vyšších dimenzí už je rozšiřování neefektivní a časově náročné jak na implementaci, tak i na následný výpočet. Nevýhodou tohoto algoritmu je také složitost $O(n^2)$, které lze dosáhnout, pokud body z množiny S leží na kružnici Běžný čas výpočtu bývá O(n*h), kde n je standardně počet bodů v souboru a h je počet bodů, které tvoří obálku H.

2.3.4. Divide and Conquer

Princip algoritmu: Problém nad množinou S rozdělíme na dvě podmnožiny S_1 , S_2 o stejné velikosti, tyto podmnožiny jsou zpracovávány samostatně. Obě řešení poté spojíme horní a dolní tečnou. Body, které zůstanou uvnitř, jsou samozřejmě z obálky vyřazeny. Tím nám vznikne celkové řešení.

Toto dělení se stále opakuje, dokud nenarazíme na triviální řešení o třech bodech. Po nalezení triviálních obálek (triviální obálka je trojúhelník ve 2D), je třeba tyto obálky spojit. Spojování se provádí pomocí nalezení dolních a horních tečen. Pro jednoduchost se pokusím popsat nalezení tečen algoritmem, kterému se říká „procházka“.

Naše dvě malé obálky nazveme M a N, obě obálky mají extrémní body n_i a m_i. Nyní potřebujeme k bodu n najít takový bod m, aby jejich spojnice byla dolní tečnou M. Podobně hledáme z bodu m bod n, jejichž spojnice tvoří dolní tečnu N. Tento postup opakujeme tak dlouho, dokud nenajdeme dolní tečnu, která je dolní tečnou N i M.



Obr. 5 - Ukázka nalezení dolní tečny

(vyznačena červeně)

Z algoritmu Divide and Conquer částečně vychází i algoritmus Quick Hull, který jsem si pro svou práci vybral.

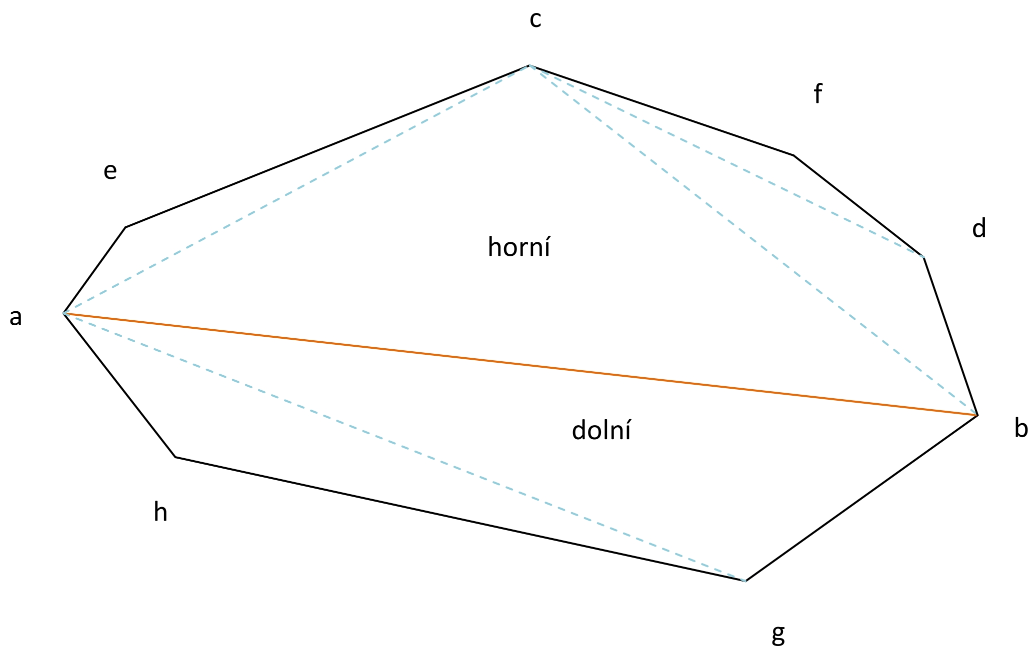
Tento algoritmus má také složitost O(n*log(n)) a lze velmi jednoduše rozšířit do vyšších dimenzí, však implementace je o něco složitější.

2.3.5. Quick Hull

Tento algoritmus mi byl doporučen panem ing. Josefem Kohoutem, a také proto jsem si ho vybral ke své práci. Detailnější popis tohoto algoritmu provedu později. Teď se zaměřím pouze na obecné fungování algoritmu Quick hull.

Princip algoritmu: Konvexní obálka, při výpočtu tímto algoritmem, je konstruována pomocí dvou obálek, kterým se říká horní a dolní obálka. Horní obálka je nad spojnicí dvou extrémních bodů (např. minimum a maximum podle x). Dolní obálka je pod touto spojnicí. Nad každou nalezenou stranou obálky a,b hledáme nejvzdálenější bod c, který se stane novým bodem obálky. Po přiřazení bodu do obálky se stávající strana rozpadne a vzniknou nové dvě. Výpočet na obou stranách obálky probíhá odděleně. Na konec se pak dolní a horní obálka spojí a vznikne nám celkový výsledek.

Rychlost tohoto algoritmu spočívá v tom, že vždy hledáme nejvzdálenější bod od již nalezeného facetu. Pracujeme jen s body, které mají předpoklad k tomu, že tvoří obálku. Takže odhad složitosti $O(n^2)$ je velmi pesimistický a málo kdy je dosažen. V praxi bývá často dosahováno složitosti O(n * log (n)).



Obr. 6 - Ukázka algoritmu Quick Hull

Přesnější popis tohoto algoritmu popíši níže, při popisu mojí práce.

Tento algoritmus jsem si vybral zejména kvůli časové náročnosti na výpočet, ale také kvůli implementaci, která mi nepřišla nijak složitá. Ovšem toto byl pouze první odhad a při pokusu naimplementovat tento algoritmus, jsem narazil na řadu komplikací.

Podle mého názoru je algoritmus Quick hull asi nejlepším algoritmem pro výpočet konvexní obálky. Díky rekurzi není samotný algoritmus ani nijak rozsáhlý, ale jak už to bývá, rekurze dokáže vše udělat složitější.

Tímto jsem popsal základní metody tvorby konvexních obálek a přesuneme se k další části práce, která spočívala v provedení výpočtu obálky pomocí vláken, tudíž paralelnímu výpočtu.

2.4. Paralelní výpočet – vlákna

Část o vláknech je nastudována a následně citována podle [Alba2008].

2.4.1. Úvod do vláken

Některé výpočty trvají velmi dlouho, takovým klasickým příkladem je výpočet π. Proto je vhodné použít paralelního výpočtu, pokud to ovšem algoritmus dovolí. Další podmínkou, pro použití paralelního výpočtu, je psaní programu pro počítač, který má alespoň dvou jádrový procesor.

Paralelní výpočet je realizován pomocí vláken. Program tohoto typu je tvořen hlavním vláknem, které spouští několik dalších. Zpravidla hlavní vlákno nepočítá žádný výpočet, to obstarají vlákna vedlejší. Hlavní vlákno se stará pouze o ovládání programu, a výpočet zůstane na vláknech vedlejších. Celý proces se tím zpravidla urychlí, ale je náchylnější na chyby, které vznikají při I/O operacích.

Pokud používáme program, který neprovádí výpočet pomocí vláken, tak se po zahájení výpočtu (hlavně při delších a časově náročných výpočtech) okno s programem zasekne a nehneme s ním. Nefungují žádná tlačítka a s programem nelze nic dělat. Při použití vláken je možné program „pozastavit“ a například dodat nějaké hodnoty.

Jelikož jsem si vybral programovací jazyk C#, tak zde uvedené příklady budou v tomto jazyce. Pro použití vláken je třeba importovat namespace System a System.Threading.

C# má ve výchozím stavu jen jedno vlákno, které pro nás vytváří běhové prostředí CRL, toto vlákno označujeme jako primární vlákno a na něm běží celá aplikace. Při použití vláken, CRL[1] přiděluje každému vláknu jeho vlastní zásobník paměti. Díky tomu mohou mít vlákna své vlastní proměnné. Po zániku vlákna je paměť uvolněna Garbage Collectorem[2] a znovu rozdělena mezi ostatní vlákna.

[1] Common Language Runtime. Je to implementace CLI (Common Language Infrastructure) od Microsoftu. CRL umožňuje ignorovat programátorovi mnoho kritérií ohledně CPU.

[2] Běhová část programovacího jazyka, které má za úkol zjistit, která část paměti programu je již nevyužívána a připravit jí k znovupoužití.

```
Class PrvniVlakno{
        static void Main(){
                Thread t = new Thread(PisB);
                t.start();                              //spustí PisB v novem vlákně
                while (true) Console.Write(„a“);        //primární vlákno bude psát x
        }
        static void PisB(){
                while (true) Console.Write(„b“);
        }
`
```

Jednoduchá ukázka programu s použitím vlákna. Program bude vypisovat a, b náhodně. Primární vlákno vytvoří vlákno se jménem t, které spustí metodu PisB. Zároveň s výpisem b bude prováděn na primárním vlákně výpis písmene a.

Kdybychom v tomto případě nepoužili vlákna a napsali program pouze tímto způsobem:

```
Class PrvniVlakno{
        static void Main(){
                while (true) Console.Write(„a“);
                while (true) Console.Write(„b“);
        }
}
```

Vypisovalo by se pouze písmeno a. Na výpis písmene b by nikdy nedošlo, protože první cyklus while by nikdy neskončil.

Výše zmiňovaný CRL nám umožní za použití lokálních proměnných více vlákny. Každé vlákno bude mít svoji lokální proměnnou a nebudou se navzájem ovlivňovat.

```
Class PrvniVlakno{

        static void Main(){

                new Thread(vypisuj).start();

                vypisuj();

        }

        static  void vypisuj(){

                for(int i = 0; i < 3;i++) Console.Write(„pes“);

        }

}
```

Výstupem tohoto programu bude šestkrát za sebou napsáno slovo pes. Šestkrát proto, že třikrát ho vypíše hlavní vlákno a třikrát nově vytvořené. Toto je přesvědčivý důkaz o práci CRL, které pro obě vlákna vytvořilo zásobníky (pro každé vlákno jeden) a v každém zásobníku proměnnou i. V každém vlákně je podmínka i<3 splněna třikrát. Proto šest slov pes.

Samozřejmě lze v různých vláknech přistupovat ke stejné proměnné. Toho lze dosáhnout použitím globální proměnné, nebo předáním pomocí odkazu na objekt.

2.4.2. <u>Jak vlákna fungují</u>

Vlákna jsou řízena pomocí „thread schedulerem“ (plánovač vláken). Zajišťuje každému vláknu čas na běh a také zajišťuje, aby čekající / uspaná vlákna nespotřebovávala procesorový čas.

V praxi většinou bývá použito tolik vláken, kolik máme k dispozici jader procesoru. V takovém to případě počítá každé jádro jedno vlákno. Však i pro jedno-jádrové počítače se dá vlákna využít. Zde běží několik vláken na jednou a „time-slicing“ mezi nimi velmi rychle přepíná. Tím pádem každé vlákno běží chviličku a budí dojem paralelního výpočtu. Čas pro jednotlivá vlákna není přidělován konstantně. V našem prvním příkladě se může stát, že se nám jednou vypíše 6 a, a pak 5 b, poté 10 a a 12 b a tak dále. Je to způsobeno tím, že ani počítač nedokáže přepnout mezi vlákny úplně přesně po uplynutí stejného časového intervalu.

2.4.3 Synchronizace

Synchronizace je velmi důležitý prvek při práci s vlákny. Zajistí nám práci vlákna pouze s daty a metodami, s kterými má opravdu pracovat. Nestane se například při zápisu do souboru, že budeme mít přeházené věty, body nebo jakákoliv jiná data.

Pro synchronizaci se používá celá řada konstrukcí, jsou to tyto:

Sleep – uspí vlákno na nějaký čas

Join – počká na další vlákno, než neukončí svou práci

Lock – zajistí, že pouze jedno vlákno smí přistoupit např. do metody

Mutex – složitější lock (jazyková konstrukce)

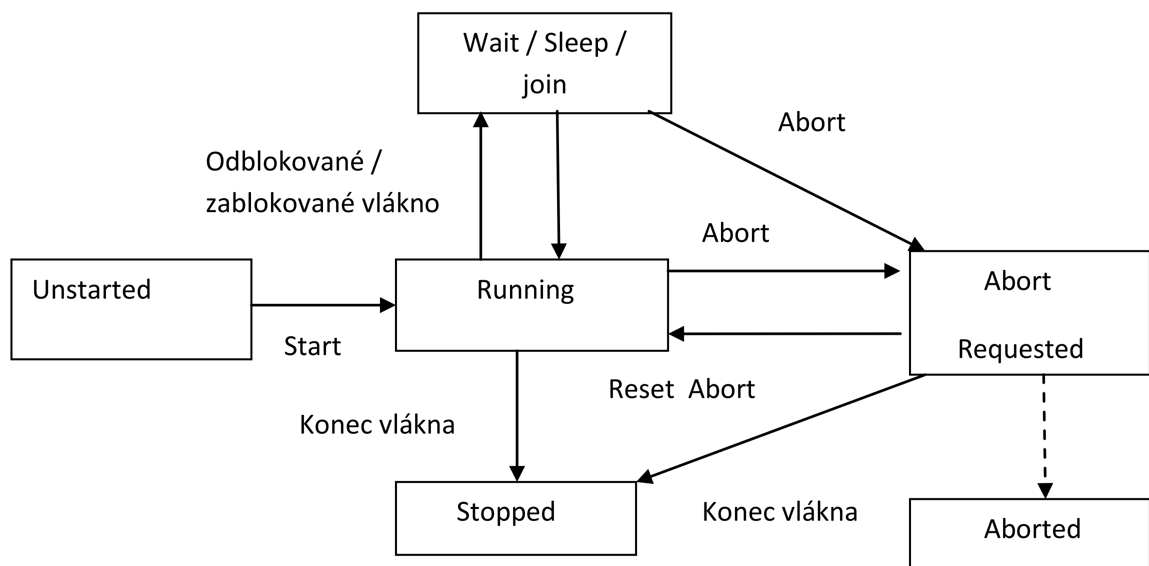
Semafor – určí, kolik vláken může přistoupit do metody / části kódu

Monitor – konstrukce programovacího jazyka, která bývá vytvořena pomocí jiného synchronizačního primitiva

Velmi užitečnou věc, kterou C# umožňuje je použití přímo synchronizovaného readeru / writeru. Stačí jednoduše použít Stream.Synchronized(„název FileStreamu“) a ten nám zajistí synchronizovaný zápis do souboru a také čtení z něj. Díky této konstrukci není zapotřebí používat pro I/O metodu Semafor, nebo nějaký podobný synchronizační prvek.

2.4.4. Stavy vláken

Vlákna se mohou za svého „života“ vyskytovat v různých stavech. Tyto stavy se pokusím znázornit v následujícím diagramu.



Obr. 7 - stavový diagram vláken

Tento stavový diagram (Obr. 7) představuje stavy vláken, tyto stavy jsou rozděleny do tří vrstev. První vrstva znázorňuje blokované vlákno. Druhá nám říká, jestli vlákno běží na pozadí, nebo popředí a třetí vrstva nám řekne o stavu vlákna vůči metodě Suspend. Metoda suspend buď pozastaví vlákno nebo pokud je již pozastaveno, neučiní nic.

Po zjištění těchto skutečností už nechybělo nic, co bych potřeboval pro to, abych mohl zpracovat program, s co možná nejlepší metodou výpočtu a algoritmem.

3. Praktická část

3.1. Metoda první: Sklápění

První metoda, kterou jsem si vyzkoušel, je metoda „**Sklápění**“. Tuto metodu jsem napsal ještě dříve, než jsem se dověděl o algoritmu Jarvis scan. Celý algoritmus jsem si vymyslel, proto je algoritmus poněkud jednoduchý a neefektivní. Implementace této metody byl pouze takový první pokus, při kterém jsem se blíže seznámil s programovacím jazykem C#. Takže pro mě tato implementace měla spíše hodnotu po stránce seznámení se s prostředím Visual studia a již zmiňovaného jazyka C#.

Algoritmus pracuje takto:

1. Zvolím si minimální prvek (například podle svislé osy).
2. Tímto bodem si proložím vodorovnou přímku, kterou postupně sklápím.
3. Po sklopení o nejmenší dílek, který si zvolím, projdu pole bodů a porovnám, jestli nějaký nevyhovuje rovnici přímky.
4. Pokud ne, hledám dál, až projdu všechny body. Pokud, nevyhovuje žádný, sklopím přímku o další dílek a znovu porovnávám.
5. Pokud ano, zapamatuju si sklopení přímky, zapíšu si bod, který tvoří obálku a sklápím dál, tentokráte už z nalezeného bodu.
6. Tímto způsobem projdu celou množinu bodů.
7. Až se dostanu do počátečního bodu, tak mám hotovo.

Takto lze provést výpočet pouze ve 2D. Ve vyšší dimensi bychom museli nejprve najít tímto způsobem počáteční rovinu a poté jí sklápět (ve 4D a vyšší dimensi by se z roviny našel nějaký facet, který by se sklápěl).

Tento způsob výpočtu však není příliš vhodný pro praktické použití. Je sice jednoduchý na implementaci, ale při sklápění ztrácíme přesnost, protože i když sklápíme o sebemenší dílek, tak nám někde v nekonečnu může ležet bod, který přeskočíme. Kvůli tomu nemusíme vždy najít konvexní obálku. Krom toho se zmenšováním dílku, o který sklápíme, nám narůstá čas výpočtu. To znamená, že čím se pokusíme o větší přesnost, tím prodloužíme čas výpočtu, který je už tak velmi vysoký. Kvůli předem stanovené velikosti dílku se algoritmus stává neefektivní, protože je v danou chvíli schopný vypočítat pouze konkrétní množinu bodu (např. buď celá čísla, nebo desetinná).

Při programování tohoto algoritmu jsem použil i řadu vylepšení a urychlení. Např. jsem si množinu bodů rozdělil podle kvadrantů (kartézské souřadnice), díky tomu jsem mohl omezit v daném kvadrantu maximální a minimální hodnotu dílku, podle kterého sklápím.

Před optimalizací jsem tímto algoritmem počítal množinu o velikosti jednoho milionu bodů zhruba 15 minut. Po použití všech možných optimalizací jsem tuto množinu zvládl vypočítat za necelých 10 minut.

Po těchto výsledcích jsem od metody „sklápění“ upustil a věnoval jsem se jiné metodě.

3.2. Metoda druhá: Quick Hull

Druhá metoda, kterou jsem použil, se nazývá **„QuickHull“**.

Funguje zhruba takto:

1. Z množiny bodů si najdu extrémní body, např. na ose x (nazveme je A a B). Tyto body uložíme do pole O, ve kterém budou body obálky.
2. Tyto body mi vytvoří přímku, která mi rozdělí množinu na dvě, horní a dolní.
3. Začneme počítáním v horní polovině. Od naší přímky nalezneme nejvzdálenější bod (nazveme ho C), který uložíme do pole O.
4. V dalším kroku spojíme pomyslnou přímkou body AC a nalezneme od nich nejvzdálenější bod, který leží na opačné straně od bodu B. Nalezený bod, uložíme do pole O. Naše přímka se rozpadne na dvě nové. To samé provedeme s body BC.
5. Tímto způsobem hledáme nejvzdálenější body od přímky, dokud nevyčerpáme všechny v horní polovině.
6. Po nalezení všech bodů, tvořících obálku v horní polovině, vytvoříme analogicky i dolní polovinu.
7. V poli O pak máme výslednou konvexní obálku.

Tento algoritmus je lehce rozšiřitelný do jakékoliv dimense, toto rozšíření se dá jednoduše realizovat konstruováním facetů místo úseček. K tomu je pouze potřeba výpočet determinantu v jakékoliv dimenzi, pro zjištění obecné rovnice facetu, od kterého se pak hledá nejvzdálenější bod.

. Je založený na Divide and Conquer z čehož vyplívá složitost $O(n^2)$, jak jsem již uvedl výše v praxi bývá složitost spíše O(n * log (n)). Což se zde pokusím ukázat:

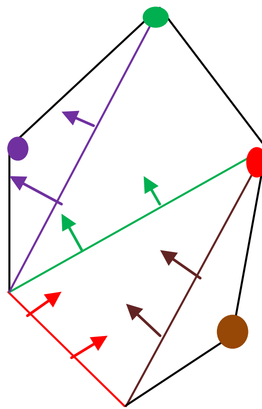
- Nalezení extrému vyžaduje čas O(n)
- Časová náročnost závisí na rekurzivním výpočtu dolní a horní obálky
- Nejlepší případ: pokud jsou obě části zhruba stejně rozsáhlé
  $T(n) = 2T(n/2) + O(n)$ z toho vyplívá složitost $O(n * \log(n))$
- Nejhorší případ nastává, pokud je jedna množina několikrát větší
  $T(n) = T(n-1) + O(n)$ z čehož vyplívá $O(n^2)$

V praxi jsou počáteční body voleny tak, aby obě poloviny množiny S byly přibližně stejné.

Toto bylo publikováno v [1]. Já však musím, toto tvrzení upřesni. Nejhorší případ nastává, pokud jsou všechny body přivedené na vstup zároveň výstupem programu. Z toho pak vyplívá, že Quick hull je velmi náchylný na výstupní data. Čím víc bodů bude tvořit obálku, tím déle Program poběží. Toto tvrzení dokazuji i v grafu uvedeném v kapitole 4 (Graf 4). Potom také musíme upřesnit složitost algoritmu, která bude O(n * m), kde n jsou vstupní body a m výstupní.

Algoritmus QuickHull je velice efektivním a přesným algoritmem, protože pro nalezení nejvzdálenějšího bodu používá pouze body z množiny, nad kterou hledáme konvexní obálku. To znamená, že při správné implementaci se nemůže stát, aby nalezená obálka nebyla konvexní. Co se týče implementace, tak v tomto ohledu je poněkud složitější. Pro správné naprogramování je potřeba využít výpočtu determinantu a vzdálenosti bodu od facetu.

Algoritmus Quick hull má sám o sobě jednu chybu. Je velmi náchylný na volbu počátečních bodů. Pokud se zvolí špatně, může se stát, že některé body zůstanou neobjevené. Proto je velmi důležité najít vždy extrémní body. Viz Obr. 8.



Obr. 8

-Červeně je vyznačena základní úsečka, z které odstartuju hledání.

-Dolní obálka zde není žádná, kvůli tomu hledáme jen směrem nahoru.

-Šipky vyznačují směr hledání. Vždy jsem označil úsečku a bod, který z ní najdeme stejnou barvou.

Na obrázku je krásně vidět, že hnědý bod nebude nikdy objeven.

Je zřejmé, že ve 2D se toto málokdy stane, ale vyšší dimenze už jsou na toto náchylnější. Moje řešení tohoto problému proberu níže.

3.3. Moje implementace Quick hull:

Abych si ulehčil výpočty, tak vždy ze začátku volím počet počátečních bodů podle velikosti dimense souřadnic (ve 3D vezmu minX, maxX a minY atd.). Snížím tím počet hledaných bodů, tím snížím počet složitých hledání extrémních bodů a hlavně nemusím ztrácet čas hledáním, u vyšších dimensí (než 2D), počátečních facetů (nepoužívat tento systém výpočtu, musel bych nejprve najít přímku, pak rovinu ….).

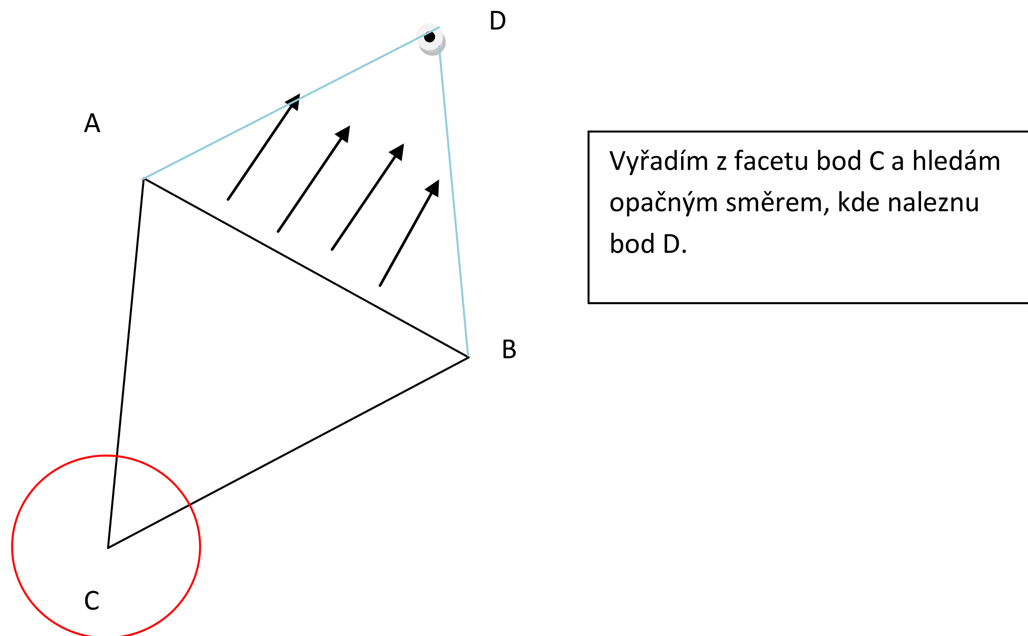
Poté už standardně počítám algoritmem Quick hullu. Vezmu počáteční facet k němu si vypočítám normálu pomocí determinantu, který vypočítávám L-U rozkladem. Nejvzdálenější bod vypočítám ze vztahu $|M\alpha| = |am_1 + bm_2 + cm_3 + \dots + d$, kde M je bod se souřadnicemi $\{m_1, m_2, m_3 \dots\}$ a α je rovnice roviny (facetu) ax + by + cz+ ….. + d = 0, potom písmena a,b,c jsou souřadnice normály. Nejvzdálenější bod je bodem konvexní obálky a uložím ho tedy do výsledného pole.

Algoritmus Quick hullu jsem trošku pozměnil, konkrétně část, která rozhoduje o směru hledání dalšího bodu. Standardní algoritmus hledá v horní části obálky nahoru a v dolní části dolů. Což je i logické, ale pokud se toto hledání neošetří proti existenci slepých úhlů. V obálce nám vznikne „díra" jednoduše nějaký bod zůstane neobjeven. Toto ošetření jsem provedl následujícím způsobem. Při nalezení nejvzdálenějšího

bodu D od facetu A B C uložím do zásobníku facety ABD – C, ADC – B a DBC – A za pomlčkou je bod (Obr. 9), který byl z facetu vyřazen a pomocí něj zjistíme na kterou stranu hledat další bod. Po prohledání prostoru nad/pod facetem, daný facet vyřadím ze zásobníku. Polorovinu, ve které budu hledat, určím jednoduše vypočtením vzdálenosti vyměněného (bodu za pomlčkou) bodu, v této vzdálenosti se podívám na znaménko a podle něj vyberu směr výpočtu.

Jelikož každý bod smí být započítán pouze jednou, vytvořím si ještě jedno pole o stejné velikosti, jako je pole bodů a naplním ho jedničkami. Po označení bodu jako nejvzdálenějšího a zařazení ho do obálky, změním hodnotu bodu, který odpovídá indexem nejvzdálenějšímu bodu od facetu, hodnotu na 2. Tím zajistím, že se každý bod vyskytne v množině s obálkou jen jednou.

Tento způsob zařazování bodů do výsledné obálky mi přinesl i jednu menší komplikaci. Po nalezení bodu jsem zařazoval bod do zásobníku pro body obálky, ale také sem z něj rovnou tvořil další facety (ze kterých probíhal další výpočet). Když už byl jednou bod objeven, nehledal sem z něj ani facet, tím pádem jsem objevil slepý úhel, ve kterém mi zůstaly neobjevené body. Po několika testech jsem zjistil, že zařadit bod do obálky jednou, ale do zásobníku s facety dimenze*dimenze krát je ideální a neztratím tím žádný bod, zejména pro rozsáhlé množiny je důležité pomocí jednoho bodu vytvořit facet víckrát.



Obr. 9 - Ukázka určení směru hledání dalšího bodu

Veškeré výpočty jsou prováděny obecně(determinant, obecné rovnice a výpočty zdáleností), tudíž není třeba provádět žádné úpravy pro zpracování bodů ve vyšších dimenzích. Algoritmus jsem testoval i na 5D a 6D a výpočet proběhl bez problému.

3.4. Moje paralelní zpracování:

Kvůli urychlení výpočtu jsem se rozhodl do knihovny umístit také vlákny, která mi umožnila paralelní výpočet konvexní obálky.

Nejprve uživatel zadá, kolik vláken chce pro výpočet použít. Poté rozpočítám, kolik bodů připadne na vlákno a ty mu při načítání přidělím, když několik málo bodů zbude, je zde vlákno, které se jich ujme a zpracuje je. Ve skutečnosti se tedy může stát, že si uživatel zadá čtyři vlákna, ale poběží jich pět.

Body načítám do globálního pole, které je dostupné pro všechna vlákna. Jednotlivým vláknům předám jako parametr „začátek“, to je hodnota kde v tomto globálním poli mají začít počítat, a „počet bodů“, to je hodnota kolik bodů z tohoto pole mají zpracovávat.

Výsledky z jednotlivých vláken ukládám do dočasného souboru, abych mohl v každou chvíli výpočtu načíst do paměti co největší množství bodů. Tento způsob ukládání bodů do souboru místo ponechání jich v paměti jsem zvolil po několika testech, které ukázali, že zpracování a výpočet obálky zabere více času než zápis a čtení ze souboru. Krom toho čím víc bodů zpracovávám naráz, tím víc bodů vyřadím a už s nimi nemusím dál počítat a tím se urychlí výpočet.

Jako synchronizační prvek jsem použil pouze Sync stream, který zaručuje synchronizovaný zápis a čtení ze souboru. Tento Stream jsem použil pouze při ukládání výsledků z vláken do dočasného souboru. Zde to bylo nutné, aby nedošlo k různým chybám typu míchání souřadnic bodů, nebo míchání malých obálek. Obě tyto chyby by mohly být způsobeny tím, že chvíli bude zapisovat první vlákno a chvíli druhé. První z chyb by pak vedla k nesprávnému vypočtení obálky, protože by vznikaly body, které vůbec nemají existovat. V případě druhé by se nic nestalo, protože na pořadí bodů nezáleží.

Pro načítání bodů ze souboru již nebylo třeba synchronizaci využívat, protože vlákna jsou spouštěna až po načtení bodů do paměti.

3.1. Řešení nedostatku paměti

Jednou z podmínek zpracovávání bodů bylo také to, že program má zvládat zpracovat i rozsáhlou množinu bodů. Rozsáhlá množina bodů je například množina o sto milionech bodů ve 3D a samozřejmě je třeba zpracovat i mnohem větší množiny. Ta v textové podobě zabere 2,1GB paměti na disku. Tak rozsáhlou množinu do paměti již nevtěsnám, proto bylo třeba vymyslet způsob, kterým dokážu zpracovat všechny body, aniž bych je musel v jednu chvíli mít načtené v paměti.
Tento problém jsem řešil postupným načítáním bodů ze souboru do paměti. Stanovil jsem si konstantu, která značí maximální počet bodů, které na jedno načtení můžu do paměti umístit. Před samotným načítáním bodů do paměti, si soubor s body přečtu celý a tím zjistím, kolik bodů obsahuje. Toto číslo pak vydělím maximálním počtem bodů, které do paměti uložím. Tím zjistím, kolikrát načtení maximálního počtu bodů proběhne. O tyto výpočty se stará tolik vláken, kolik si uživatel určí. Každé vlákno mi pak z přidělených bodů spočte konvexní obálku. Pokud nějaké body

v souboru zůstanou a nedostane se na ně v těchto několika zpracování, je zavoláno posední vlákno, které ze zbylých bodů konvexní obálku.

Výsledky jednotlivých cyklů načítání ukládám do dočasného souboru. Tuto variantu jsem zvolil z důvodu urychlení, protože na každý cyklus načtení chci zpracovat co největší počet bodů, nechci ztrácet operační paměť uchováváním dočasných výsledků.

Po uložení všech výsledků do dočasného soubodu tyto body znovu načtu do paměti, nyní jich již bude mnohokrát méně (až na extrémní případy např. body koule), tyto body již standardně zpracuji a vypočtu z nich výslednou konvexní obálku, kterou vrátím uživateli a soubor s dočasnými body smažu.

Příklad zpracovávání bodů:

- Mám celkově 82 milionů bodů (C = 82 milionů)
- Do paměti načtu maximálně 20 milionů bodů (M = 20 milionů)
- Vypočtu si počet cyklů C/M a vyjde mi 4,1 cykly
- Zpustím čtyřikrát cyklus načítání a vypočtu z těchto bodů konvexní obálky
- Na zbylé body pak použiji poslední vlákno

4. Zhodnocení výsledků

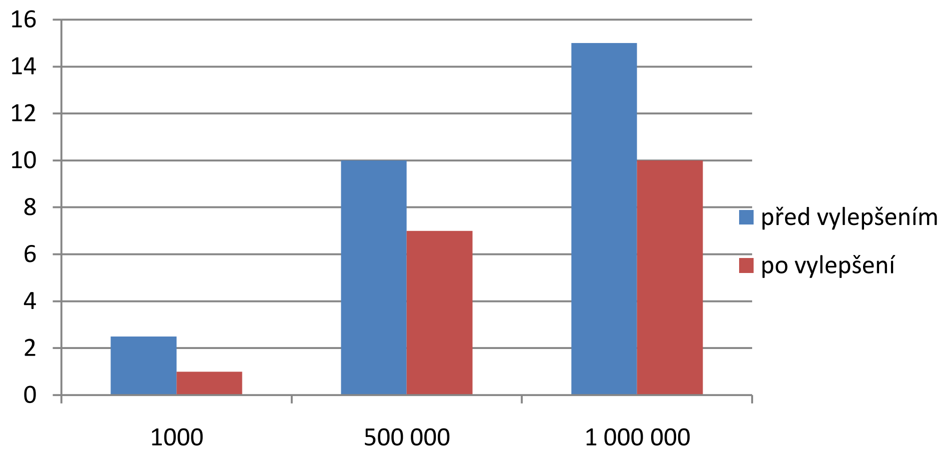
Díky implementaci dávkového zpracování nikdy nezahltím paměť tak, aby aplikace spadla. Téměř po celou dobu běhu programu je spotřebováváno stejné množství paměti. Mimo to při výpočtu pomocí x vláken na x jádrovém procesoru je využití procesoru sto procentní, pokud pak je vláken méně, jsou vidět rezervy a procesor je využit obvykle na hodnotu okolo šedesáti procent. Tuto hodnotu ovlivní další procesy, které v systému běží na pozadí, proto to není přesně padesát procent.

Pokud nebude uvedeno jinak, tak všechny níže uvedené hodnoty jsou naměřeny na počítači s 2 GB RAM a procesorem Intel Core 2 Duo CPU T7300 s 2x 2.00 GHz.

Nejprve pro ukázku uvedu výsledky, které jsem dosáhl s algoritmem „sklápění“. Jeho hodnoty jsou uvedeny v Grafu 1, který můžete vidět níže.

Na ose y je uveden čas výpočtu v minutách a na ose x počet zpracovávaných bodů ve 2D (Graf 1). Před vylepšením pak znamená, před implementací části algoritmu, která určí, od jaké minimální hodnoty mám sklápět a jaké maximální hodnoty mohu dosáhnout v daném kvadrantu. Po vylepšení je pak s touto implementací.

Sklápění



Graf 1 - Sklápění pro body ve 2D

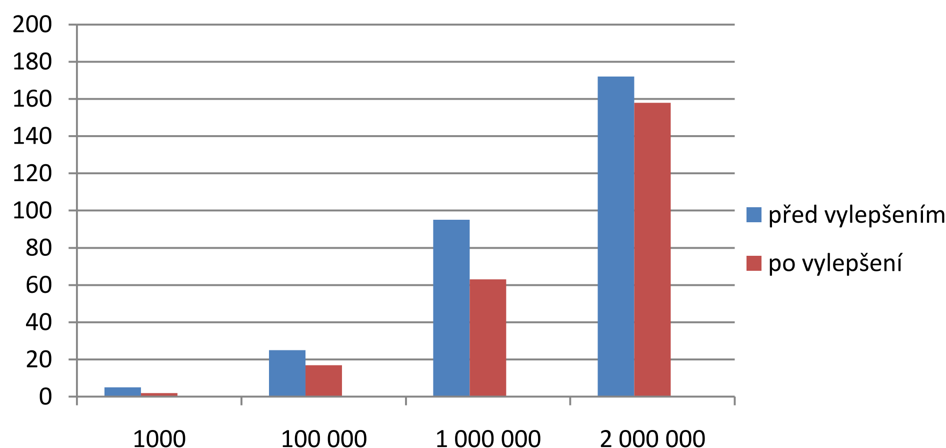
Krom velmi vysokého času výpočtu, uchovávání bodů v paměti jí zabíralo obrovské množství. Na jedno načtení jsem do paměti dokázal dát maximálně pět milionů bodů, abych zabral 1GB RAM. Tento fakt byl způsoben uchováváním bodů v polích a kopírováním bodů do nových a nových polí. Krom toho jsem měl zvlášť pole pro souřadnice na ose x,y a z. To znamená, že jsem některé body zbytečně uchovával několikrát a zabíral tím paměť.

Po dosažení těchto výsledků jsem usoudil, že budu muset zvolit jiný algoritmus. Vybral jsem si již výše uvedený Quick Hull.

Krom algoritmu výpočtu jsem zvolil také jiný způsob uchovávání bodů v paměti. Vytvořil jsem si pole objektů (Itemů), každý objekt obsahuje pole double o velikosti dimenze. Krom paměťové úspory mi tento způsob uchovávání bodů přinesl také mnohem jednoduší manipulaci s body, již není třeba si hlídat indexy dvou polí. Dále také už nekopíruji hodnoty bodů mezi poli, ale pouze přesouvám ukazatele na hodnoty. Tímto způsobem jsem schopný do paměti uložit dvacet pět milionů bodů ve 3D a zaberou mi také 1GB RAM (pokud si odmyslíme nějaké reference, které také nějakou paměť spotřebují)

Pole objektů:	0	1	2	3	4
Souřadnice x:	12	7	5	4	8
Souřadnice y:	10	6	7	2	1

Obr.10 - přiklad uložení bodů



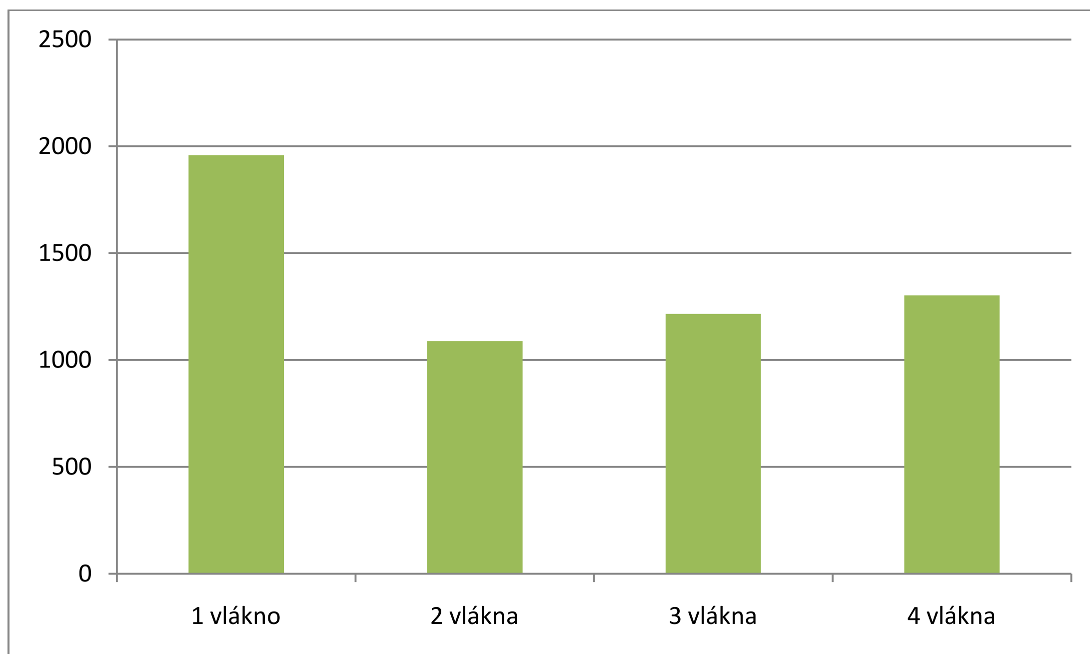
Graf 2 – Quick hull pro množinu bodů ve 3D

Tento graf (Graf 2) představuje výpočet konvexní obálky pomocí algoritmu Quick hull. Na ose x je znázorněn počet bodů, které jsem při testu zpracovával, na ose y je čas výpočtu znázorněn ve vteřinách. Před vylepšením je čistý algoritmus, po vylepšení je pak optimalizovaný algoritmus. Jsou zkráceny některé cykly, které probíhaly zbytečně, a také jsem implementoval metodu pro hledání maximálních bodů, z kterých začínám výpočet.

S ním jsem dosáhl o poznání lepších výsledků, ale pořád to nebylo ono. Počítat množinu o sto milionech bodů by bylo opravdu na hodně dlouho. Proto jsem se rozhodl, že se pokusím o paralelní zpracování, pomocí něho by už mohl výpočet proběhnout v rozumném čase.

4.1. Vliv počtu vláken na dobu výpočtu

Jak jsem již uvedl výše, paralelní výpočet se vyplácí pouze při použití více procesorů. Pokud má každé vlákno svoje jádro procesoru, tak teprve poté je vidět nějaké časové zlepšení výpočtu. Pokud ovšem musí CRL pořád přepínat mezi vlákny, časové vylepšení se ztrácí. Jako důkaz uvedu (Graf 3) naměřené hodnoty pro jedno, dvě a čtyři vlákna na dvou-jádrovém procesoru.



Graf 3 – Závislost počtu vláken na čase výpočtu bodů ve 3D

Na ose x je počet vláken a osa y vyjadřuje dobu výpočtu v sekundách (testováno na 100 milionech bodů). Na grafu je dobře vidět, že víc vláken neznamená kratší dobu výpočtu. I kdyby výpočet proběhl o něco málo rychleji, tak se celková doba zpracování protáhne. Toto prodloužení je způsobeno dvěma aspekty. Vlákna do souboru zapisují postupně, to znamená, že píše jedno a další čekají, až na ně přijde řada. Toto je nezbytná věc, je to otázka synchronizace. Druhý důvod pomalejšího výpočtu, je ten, že čím více malých obálek, tím více bodů mi zůstane na výpočet celkové konvexní obálky.

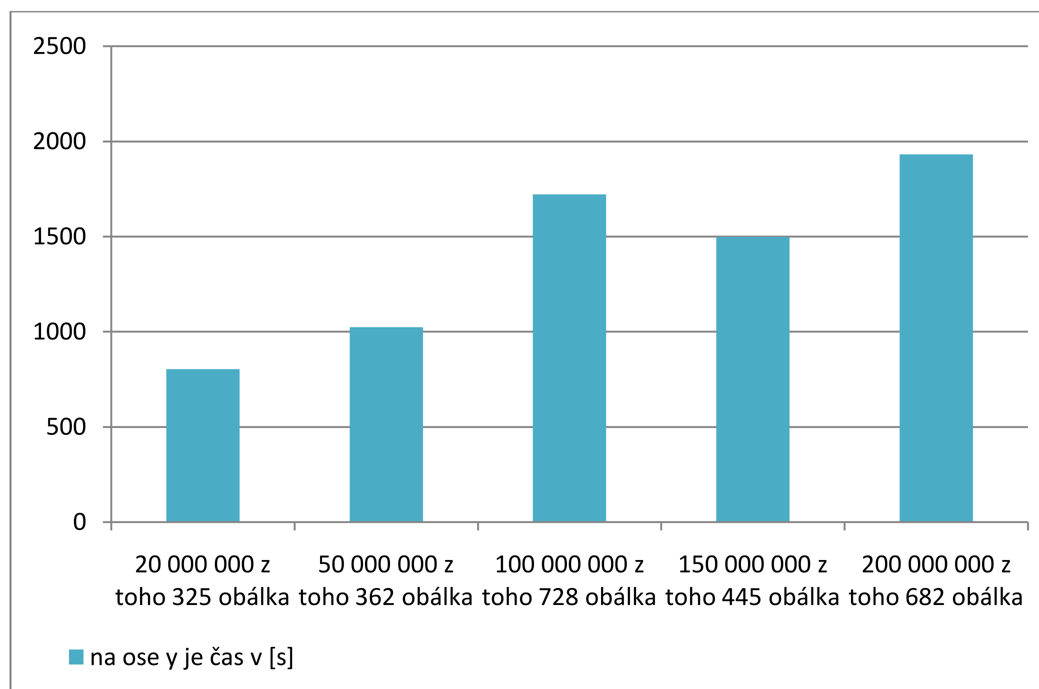
Z tohoto důvodu je vždy důležité rozmyslet si, kolik vláken je ideální použít. Proto jsem do programu zařadil možnost, zadat jako počet vláken parametr 0, když tak uživatel učiní, program si sám zjistí, kolik jader procesoru má k dispozici a výpočet pak proběhne co možná nejefektivněji.

Samozřejmě počet vláken, není jediná věc, která ovlivní délku výpočtu. Další velmi podstatnou věcí, která čas výpočtu ovlivní, jsou body, které počítáme, pochopitelně mám na mysli jejich rozložení v prostoru (ne jejich kvantitu, u té je samozřejmé, že ovlivní délku výpočtu). Rozdílný čas výpočtu bude u milionu bodů, které byly vygenerovány náhodně, a jiný bude u kružnice, nebo v prostoru u koule. Výpočet u těchto objektů bude znatelně vyšší, toto je dáno již zmiňovanou závislostí na počtu výstupních bodů.

Dalším grafem se pokusím dokázat, že nezáleží tolik na počtu kontrolovaných bodů, ale spíše na jejich rozložení v prostoru. Je ozkoušeno, že když jsou body blízko u sebe a je málo extrémních bodů, výpočet proběhne velice rychle. Toto je způsobeno tím, že při několika prvních hledání jsou vnitřní body odfiltrovány a počítáme už jen s těmi, co mohou tvořit extrémy. V obálkách, kde máme hodně extrémů, trvá výpočet podstatně déle. To je způsobeno tím, že se zvyšujícím se počtem extrémních bodů se

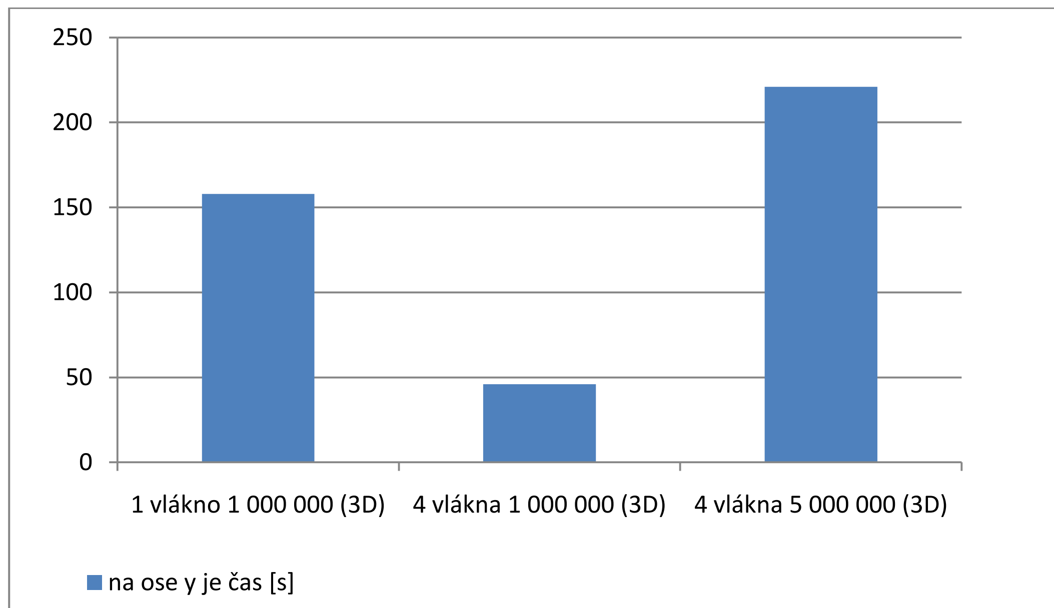
zvyšuje i počet facetů. Z každého facetu je pak třeba počítat v dalších dvou směrech, kde hledáme další extrémní body.

Níže uvedený graf (Graf 4) znázorňuje závislost délky výpočtu u náhodně vygenerovaných bodů. Na ose y je znázorněn čas ve vteřinách a osa x představuje celkový počet bodů spolu s počtem nalezených extrémních bodů. Jak je na grafu vidět, čas výpočtu se nebude zásadně měnit (u náhodně vygenerovaných bodů) s počtem bodů, které přivedeme na vstup.



Graf 4 – Závislost počtu extrémních bodů na délce výpočtu (body jsou ve 3D)

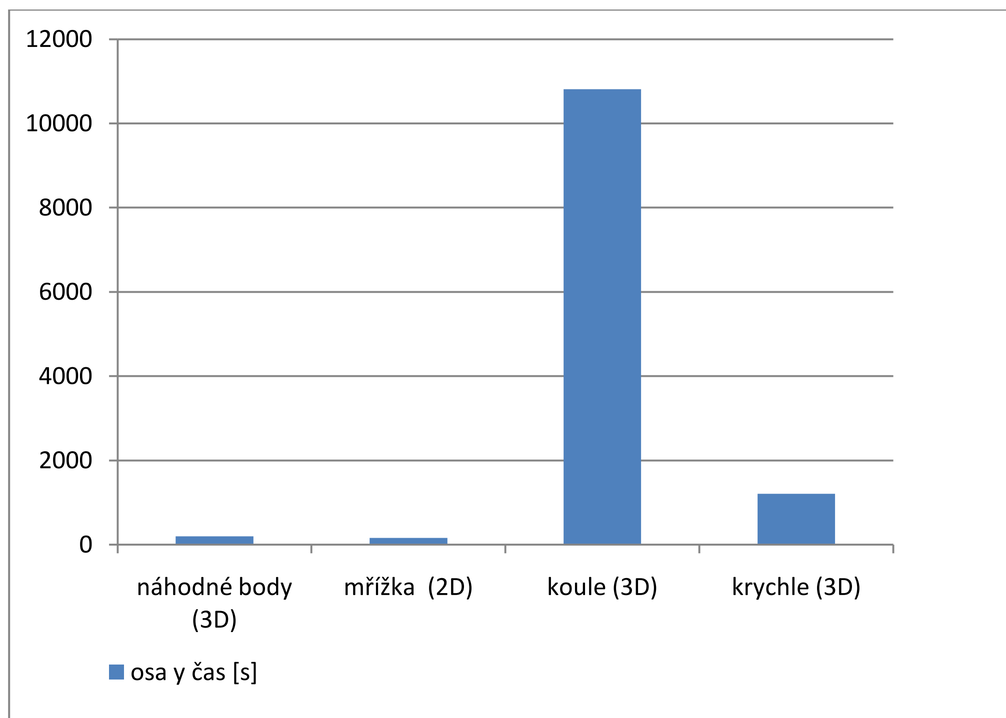
Další test, který jsem provedl, byl výpočet obálky na čtyř-jádrovém procesoru Nejprve pro jedno vlákno a pak pro čtyři. Časové výsledky nejsou čtvrtinové oproti jedno-jádrovému procesoru. Důvodem je čekání vláken na zapisování do souboru. Zde je dosti patrné čekání vláken. Jednak na již zmíněné zapsání do souboru, ale také na obdržení další práce. Výsledky tohoto testu jsem zanesl do grafu (Graf 5), který je uveden níže. První sloupec představuje použití jednoho vlákna na čtyř-jádrovém procesoru, další dva jsou již pro čtyři vlákna.



Graf 5 – Test délky výpočtu na čtyř-jádrovém procesoru (body ve 3D)

4.2. Zvláštní případy dat

Jako takovou perličku chci ukázat graf (Graf 6), ve kterém je patrný rozdíl ve výpočtu náhodně vygenerovaných bodů, mřížky, koule a krychle (samozřejmě ve 3D). Každá množina shodně obsahuje jeden milion bodů. Výpočet byl proveden pouze jedním vláknem.

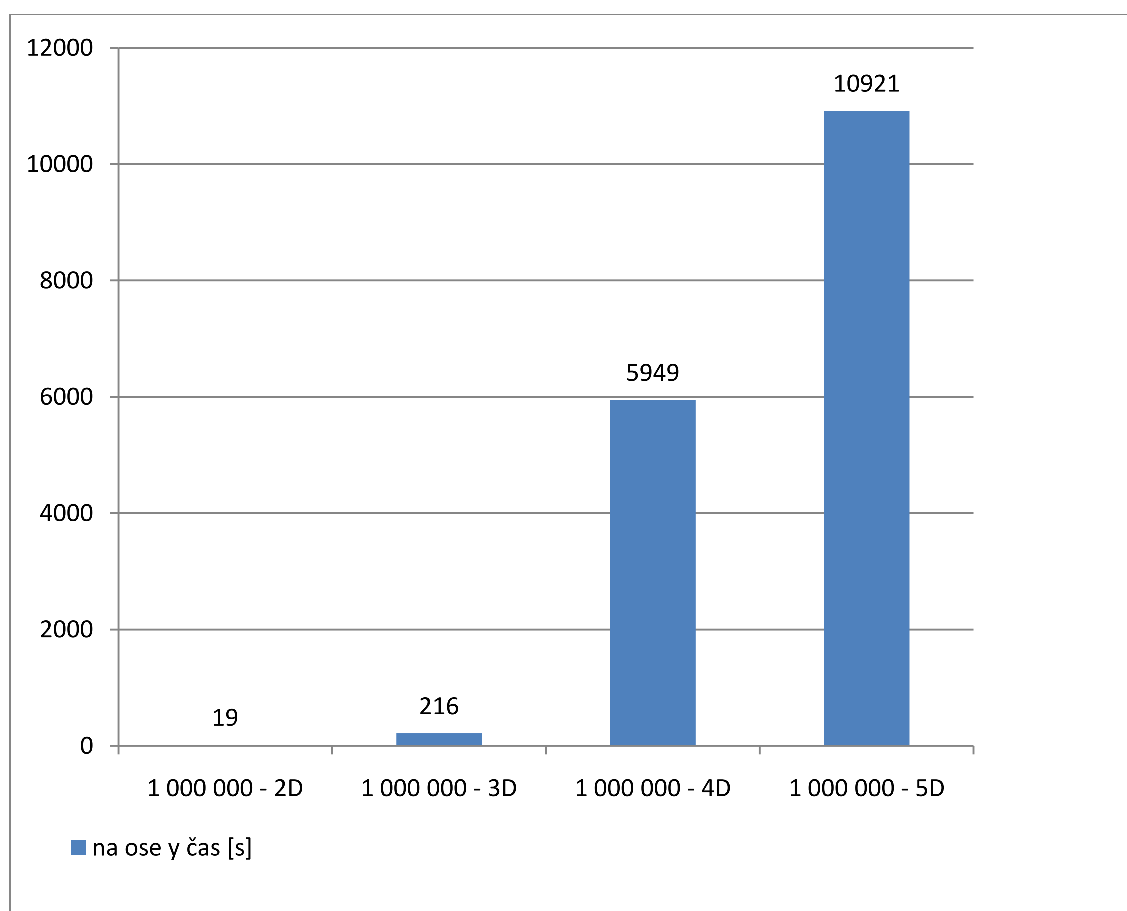


Graf 6 – Test zvláštních případů

Na tomto grafu je krásně vidět, jak počet extrémních bodů zvýší čas výpočtu. V případě koule to bylo až neúnosných 180 minut (výpočet jsem nenechal dokončit, proběhlo zhruba 30%), kdežto náhodné body byly vypočítány za 3 minuty.

Pro zajímavost uvedu graf (Graf 7), který uvádí výsledky čtyř testů. Všechny jsou pro milion bodů na stejném, dvou-jádrovém procesoru. První sloupec uvádí časový údaj o výpočtu ve 2D, druhý ve 3D, třetí ve 4D a čtvrtý v 5D. Na grafu je vidět, že při zvyšování dimenze se zvyšuje také obtížnost výpočtu a počet vzniklých facetů. Jak již uvádím výše, s nárůstem počtu facetů vzrůstá také počet rekurzivních volání výpočtu. Ve 2D přidáním jednoho bodu vzniknou dva nové facety, ve 3D vznikají 3 nové facety atd. Z tohoto důvodu je vidět časový nárůst výpočtu. Samozřejmě jsou tyto výsledky pouze orientační, protože jak již bylo řečeno, záleží na počtu bodů, které tvoří obálku. Z důvodu velmi malých hodnot ve 2D a 3D jsem u grafu (Graf 7) uvedl popisky dat.

Pro výpočet v 5D bylo potřeba pouze pozměnit zápis do souboru v aplikaci. Knihovna zvládne výpočet bez problému.



Graf 7 – Závislost dimenze na času výpočtu

5. <u>Závěr</u>

Po předchozím nepovedeném pokusu o implementaci algoritmu „sklápění“, který by bylo velice obtížné a ne příliš efektivní rozšiřovat do vyšších dimensí. Po domluvě s panem supervizorem ing. Josefem Kohoutem jsem se rozhodl, že zkusím jiný algoritmus a to algoritmus Quick hull. Tento algoritmus je jednoduše rozšiřitelný do jakékoliv dimense, mnohem přesnější a nesrovnatelně rychlejší. Algoritmem „sklápění“ jsem milion bodů ve 3D počítal zhruba 15 minut, tuto množinu zvládne Quick hull vypočítat za 160 vteřin.

Další požadavek pana supervizora byl, abych z hlavních funkcí, které vypočítávají obálku, vytvořil knihovnu. Výpočetní funkce jsem tedy předělal, aby se daly použít v knihovně a vytvořil jsem knihovnu s názvem *KnihovnaQhull.dll,* kterou lze snadno použít v jakékoliv aplikaci. Stačí ji pouze poslat cestu k souboru a počet vláken. Ona nám vrátí konvexní obálku v poli také s double prvky. Pro použití je třeba (v jazyce C#) použít *using QhullKnihovna* a poté už jen použít knihovní funkci *pouzitiKnihovny*. K této knihovně jsem naimplementoval i aplikaci, která mojí knihovně posílá body.

Do paměti načítám tolik bodů, aby zabraly zhruba 1GB operační paměti. Díky tomu nedojde k jejímu zahlcení. Na 1GB připadne zhruba 20 milionů bodů ve 3D. Tolik bodů zvládnu spočítat na jeden zátah. Body, které se nevešly do prvního kola výpočtu, jsou zpracovány v dalších. Body, které vrátí jednotlivá vlákna, zapíšu do dočasného souboru. Po doběhnutí všech vláken body ze souboru znovu načtu do paměti a vypočítám z nich výslednou konvexní obálku.

Pro zhodnocení výsledků bylo zapotřebí určit, jestli je vypočtená obálka opravdu správně. Z tohoto důvodu jsem do aplikace, která knihovnu volá, zařadil algoritmus, který ověří správnost výpočtu.

Pro malé množiny (přibližně půl milionu bodů) bodů ve 3D, 4D a pro jakkoliv velké množiny ve 2D, dostanu vždy správný výsledek. Bohužel při testech rozsáhlých obálek jsem zjistil, že mi vzniká malý prostor, kterým neprovedu výpočet. Tudíž do obálky nezařadím několik bodů. Toto je způsobeno tím, že při velikém množství facetů naleznu některé body vícekrát. To je způsobeno špatným směrem hledání dalších bodů. Místo hledání směrem ven z tělesa, občas hledám směrem dovnitř. Příčinou je měnící se orientace facetu. Kvůli tomu se musí dynamicky měnit i směr výpočtu, což já udělám pouze při přechodu mezi horní a dolní obálkou. Tato chyba by se dala odstranit dvěma způsoby. První je měnit směr hledání s ohledem na orientaci facetu. Druhý způsob je pak zjemnění kroku, tzn. zmenšení počtu zpracovávaných bodů na minimum. První způsob řeší problém elegantně a nezhorší se výsledky časů, zatím co druhý sice problém vyřeší, ale za cenu obrovského časového nárůstu. Bohužel jsem tuto chybu objevil až při konečných testech a již nebyl čas ji odstranit.

Použité zdroje:

Přímo jsem použil:

[Bay2009] Bayer T. 2009, Prezenace UK v Praze „Konvexní obálka množiny bodů“ < http://web.natur.cuni.cz/~bayertom/Adk/adk4.pdf > [citováno 15.dubna 2010]

[Alba2008] Albahari J. 2008, „Threading iv C#“, <http://www.albahari.com/threading/threading_czech.pdf > citováno podle překladu publikace „Threading in C#“. Překlad této publikace obstaral Jakub Kottnauer. [citováno 15.dubna 2010]

[1]<http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/ConvexHull/quickHull.htm > [citováno 9.května 2010]

Dále jsem pak nahlížel do těchto zdrojů:

- www.wikipedie.cz – význam některých výrazů
- www. en.wikipedia.org
- http://www.ahristov.com/tutorial/geometry-games/convex-hull.html
- http://www.microsoft.com/cze/msdn/ - informace i příkazech v jazyce C#
- Katedra matematiky (výpočet obecné rovnice v jakékoliv dimenzi). www.kma.zcu.cz

Příloha č.1

Uživatelská dokumentace

Uživatelská dokumentace k programu

Pro použití knihovny v jiném, než je mnou připraveném programu je nejprve třeba ji importovat. Toto provedeme následovně: Ve Visual studiu klikneme pravým tlačítkem na *References* a zvolíme možnost *add references* a poté provedeme samotný import *using QhullKnihovna*. Dalším krokem je pak zavolání knihovny např. tímto příkazem:

```
double[]  vysledneBody = Qhull.pouzitiKnihovny(cesta, pocetVlaken);
```

vysledneBody – pole do kterého nám knihovna vrátí výsledek

cesta – cesta k souboru, zadaná jako string (řetězec)

pocetVlaken – představuje počet vláken, které provedou výpočet

Při zadání počtu vláken 0 si program sám zjistí, kolika jádrový procesor má k dispozici a podle počtu jader spustí daný počet vláken.

Pro snadné testování jsem naprogramoval také vzorovou aplikaci, která knihovnu využívá a je v ní možno použít i kontrolní algoritmus, který je její součástí. Ovládání aplikace je triviální a není třeba ho tu popisovat.

Formát vstupu: textový soubor s body ohraničenými hranatými závorkami a na každém řádku uveden jeden bod. Souřadnice jsou pak odděleny středníkem. Desetinný oddělovač je brán v podobě čárky.

Př. [12;20;11]

[1,1;50;99,2]

[4;4;4]

Formát vstupu v podobě tohoto textového souboru jsem si vybral z důvodu, že se obtížněji parsuje, než přímo binární, ale pro převedení na binární se jen odebere parsování a použitý reader zvládne i toto.

Výstup: Knihovna vrací pole hodnot double.

Moje aplikace provede výpis do textového souboru ve stejném formátu jako je vstup.

Příloha č.2

CD s programem a texty