VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

POROVNÁNÍ METOD PRO ROZKLAD KŘEHKÝCH TĚLES NA GPU POMOCÍ 3D VORONÉHO DIAGRAMU

COMPARISON OF BRITTLE BODY DECOMPOSITION

GPU BASED METHODS USING VORONOI DIAGRAM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MICHAEL ONČO**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ CHLUBNA**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Ončo Michael, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Porovnání metod pro rozklad křehkých těles na GPU pomocí 3D Voroného diagramu**
Comparison of Brittle Body Decomposition GPU Based Methods Using Voronoi Diagram
Kategorie: Počítačová grafika
Zadání:

1. Prostudujte knihovnu OpenGL a její nadstavby.
2. Seznamte se s metodami generování Voroného diagramu vhodnými pro GPU
3. Navrhněte framework pro vizualizaci výsledku a měření efektivity daných metod
4. Navrhněte efektivní způsob rozkladu konvexních objektů pomocí Voroného diagramu
5. Implementujte Delaunay a Tetrahedra-cutting metodu a rozklad objektů
6. Proveďte měření a zhodnocení časové a paměťové náročnosti
7. Vytvořte video pro prezentování projektu.

Literatura:

- Podle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4
- Funkční prototyp aplikace

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Chlubna Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 3. června 2020
Datum schválení: 1. listopadu 2019

Abstrakt

Tato diplomová práce se zabývá vytvářením Voroného diagramu ve 3D pomocí grafické karty. Zaměřuje se na jednotlivé algoritmy, které z množiny bodů tento geometrický útvar vytváří a na jejich porovnání. V rámci práce jsou implementovány dva algoritmy.

První je založený na vytvoření Delaunayho tetrahedralizace pomocí paralelního štěpení a transformací čtyřstěnů. Potom je tento útvar převeden na Voroného diagram. Druhý implementovaný algoritmus paralelně vytváří jednotlivé buňky postupným vyhledáváním nejbližších bodů v jejich okolí a osekáváním modelů pomocí vzniklých rovin.

Testování poukazuje na výhody a nevýhody jednotlivých algoritmů a v jakých případech se jejich rychlosti liší. Hlavně je zde upozorněno na citlivost druhé metody s ohledem na nevhodné rozložení bodů uvnitř obalového tělesa. Dále je poukázáno na pomalejší začátek algoritmu vytvářejícího Delaunayho triangulaci. Je ale zjevná i jeho výborná optimalizace pro práci s velkým počtem bodů.

Abstract

Following thesis regards itself with Voronoi diagram creation in 3D using a graphics card. It focuses on and compares certain algorithms that construct the diagram when given set of points in space. For this purpose there have been two algorithms implemented.

First one creates Delaunay tetrahedralization using parallel splitting and flipping of present tetrahedra. Then it transforms it into a Voronoi diagram. The second algorithms utilizes planes to cut a mesh until required shapes are created.

Testing shows the advantages and disadvantages of these algorithms and their relative performance. Main takeaway from this work for these algorithms is the relative sensitivity of the second method to the use of inappropriate shape in relation to given set of points. For the other algorithm its slower start and relative unsuitability for use with smaller sets of points is apparent, but it is greatly optimized for big sets.

Klíčová slova

3D, Voronoi, Delaunay, tetrahedralizace, triangulace, GPGPU

Keywords

3D, Voronoi, Delaunay, tetrahedralization, triangulation, GPGPU

Citace

ONČO, Michael. *Porovnání metod pro rozklad křehkých těles na GPU pomocí 3D Voroného diagramu*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Chlubna

Porovnání metod pro rozklad křehkých těles na GPU pomocí 3D Voroného diagramu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Chlubny. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michael Ončo
31. července 2020

Poděkování

Chtěl bych poděkovat Ing. Tomáši Chlubnovi, který mou práci vedl a všem, kteří mi během psaní pomohli jak radami, tak podporou.

Obsah

Kapitola 1

Úvod

Výpočetní geometrie se zabývá geometrickými problémy a algoritmy, které je řeší. Tyto problémy se ve všemožných formách objevují v mnoha odvětvích informatiky i v jiných oborech. Jedná se například o grafiku, umění nebo geografické systémy. Výpočetní geometrie umožňuje simulaci různých jevů, tvorbu modelů z naskenovaných dat, detekci kolizí, plánování pohybu či počítačové vidění. Proto je vhodné rozvíjet výzkum v této oblasti a hledat ideální postupy pro řešení těchto problémů. Mezi tyto problémy také patří vytváření Voroného diagramu. Ten může být použit třeba k rozkladu těles. To má využití například v analýze materiálů nebo jako efekt pro grafické ozvláštnění scény. Použití Voroného diagramu, tak jak je implementován v této práci je spíše vhodné pro grafické použití. Objekt je pomocí jeho dělení rozsekán na menší kousky, tedy roztříštěn.

V posledních letech začalo být výhodné používat běžně dostupné grafické akcelerátory (dále také označované GPU) pro náročné výpočty obecného charakteru. Tento koncept je nazýván GPGPU. Tyto výpočetní prostředky umožňují masivně paralelizovat algoritmy. Mezi populární technologie v tomto odvětví patří CUDA, OpenCL a „compute shader“ pro OpenGL (dále výpočetní shadery).

Tato práce se věnuje vytváření Voroného diagramu pomocí GPU. Zaměřuje se na dva přístupy, které tento problém řeší. V rámci této práce jsou tyto algoritmy popsány, implementovány a porovnány. V tomto textu jsou nejdříve v kapitole 2 popsány relevantní geometrické útvary, důležité vzorce analytické geometrie a práce s desetinnými čísly na počítači. V této kapitole jsou také popsány Voroného diagram a Delaunayho triangulace. Potom v kapitole 3 jsou nastíněny vybrané algoritmy a postupy. V poslední kapitole s převzatými teoretickými poznatky 4 jsou popsány důležité detaily práce na grafické kartě. Dále v kapitole 5 jsou informace z předešlých kapitol použity k návrhu řešení, která budou dále použita pro implementaci. Kapitola 6 pak popisuje důležité a zajímavé implementační detaily vytvořené testovací aplikace. V předposlední kapitole 7 jsou prezentována provedená měření implementovaných algoritmů. Nakonec je celá práce zhodnocena v poslední kapitole 8.

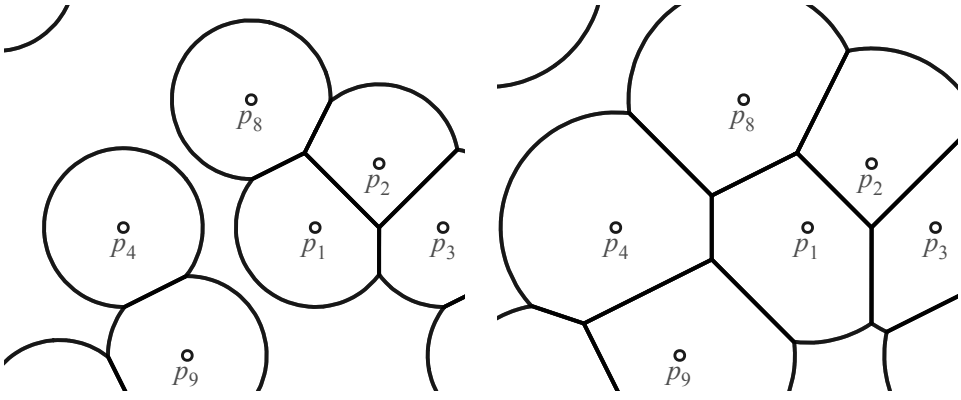
Kapitola 2

Použité principy výpočetní geometrie

Tato kapitola se zabývá problémy, které řeší výpočetní geometrie. Jsou zde popsány pro tuto práci důležité geometrické struktury a vzorce. Nejdříve je v sekci 2.1 popsána geometrická struktura Voroného diagram. Další podobný geometrický útvar, Delaunayho triangulace je potom popsána v následující sekci 2.2. Vztah mezi nimi je blíže popsán v sekci 2.3. Následuje sekce 2.4, zde jsou popsány pro tuto práci důležité vzorce analytické geometrie. Nakonec je popsána reprezentace desetinných čísel a problémy, které s sebou nese pro výpočetní geometrii v sekci 2.5.

2.1 Voroného diagram

Geometrická struktura nazývaná Voroného diagram (dále také zkracováno jako VD) rozděluje prostor do tzv. buněk. Tato struktura vyjadřuje vztah významných bodů mezi sebou a zároveň jejich vztah k okolnímu prostoru. Pro tyto body existuje více jmen, nejčastěji používaný je ale anglický výraz „sites“. V rámci této práce jsou označovány jako jádra buněk, nebo případně také středy buněk.



(a) (b)

Obrázek 2.1: Ukázka vztahu mezi buňkami VD a vzdáleností bodů od jader: Černé linie ohraničují body patřící do dané buňky. Maximální vzdálenost bodu od jádra buňky je omezena.

Jak již bylo zmíněno základem pro vytváření Voroného diagramu je schopnost hodnotit vzdálenost bodů v prostoru mezi sebou. V rámci této práce je použita metrika d pro výpočty vzdálenosti mezi body A a B z rovnice 2.1.

$$d(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2} \tag{2.1}$$

Jedná se o metriku trojrozměrného Euklidovského prostoru a pro účely Voroného diagramu v reálném světě se používá nejčastěji. Dvojrozměrná varianta této funkce vypouští člen $(A_z - B_z)^2$ a pro vyšší dimenze se naopak analogicky přidávají členy nové. Alternativně lze tuto funkci definovat i jiným způsobem pro jiné metrické prostory.

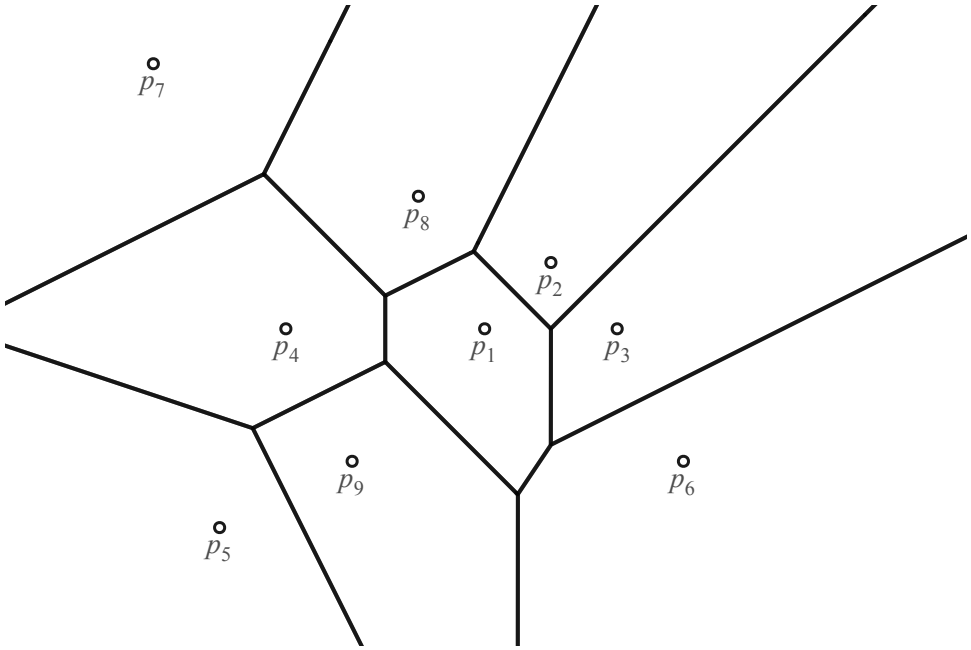
Při konstrukci diagramu pro konečnou množinu bodů P v prostoru $\mathbb{R}^n$ dochází k relativně komplexní interakci mezi body, kde jejich pozice vůči ostatním bodům určuje tvary a velikosti vznikajících buněk. Jedna buňka Voroného diagramu $V_C(P, p_i)$ vytvořená okolo bodu p_i patřícímu do P je pak množina všech bodů, které jsou blíže k tomuto bodu než k ostatním bodům množiny P. Buňka je pak formálně definována v definici 2.2.

$$V_C(P, p_i) = \{x \in \mathbb{R}^n : d\,(p_i, x) < d\,(p_l, x) \wedge p_i \neq p_l, \forall p_l \in P\} \tag{2.2}$$

Voroného diagram $V_D(P)$ konečné množiny bodů P je pak množina buněk, kde každá buňka je konstruována s jádrem z množiny P a její tvar je určen podle již zmíněných pravidel podle rozložení bodů v množině P. Formálně definován je VD v definici 2.3.

$$V_D(P) = \{V_C\,(P, p_i) : p_i \in P\} \tag{2.3}$$

Pro grafickou ilustraci Voroného diagramu viz obrázek 2.2. Dále je možné vidět naivní způsob vytváření buněk na obrázku 2.1. Zde se postupně rozšiřuje hranice buněk, dokud nenarazí na hranici sousední buňky. Hranice se zde rozšiřuje postupně tak, že všechny body buňky jsou vždy vzdáleny od jejího jádra maximálně určitou vzdálenost.



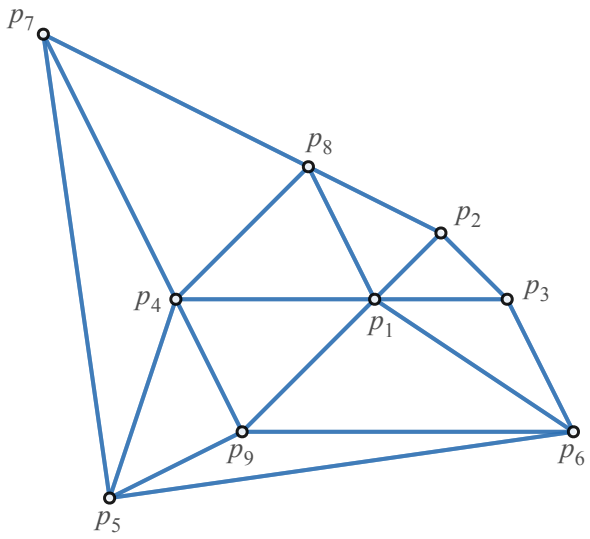
Obrázek 2.2: Příklad Voroného diagramu ve 2D

2.2 Delaunayho triangulace

Obecně je triangulace množiny bodů struktura, která tuto množinu rozděluje na simplexy. V rámci dvourozměrného prostoru se jedná o trojúhelníky a u trojrozměrného zase jde o čtyřstěny. Pro množinu bodů P v prostoru $\mathbb{R}^n$ je pak možné triangulaci chápat jako množinu k-tic $(p_1, p_2, \ldots, p_k)$, kde $k = n + 1$. Pro každou k-tici platí, že všechny body p_i patří do množiny P a v rámci jedné k-tice se jeden bod neobjevuje vícekrát. Pro $\mathbb{R}^2$ je pak triangulace $T(P)$ podmnožinou všech takovýchto trojic. Definice 2.4 pak obsahuje formální zápis definující takovou triangulaci.

$$T(P) \subseteq \{(p_1, p_2, p_3) : p_1, p_2, p_3 \in P \wedge p_1 \neq p_2 \wedge p_2 \neq p_3 \wedge p_1 \neq p_3\} \tag{2.4}$$

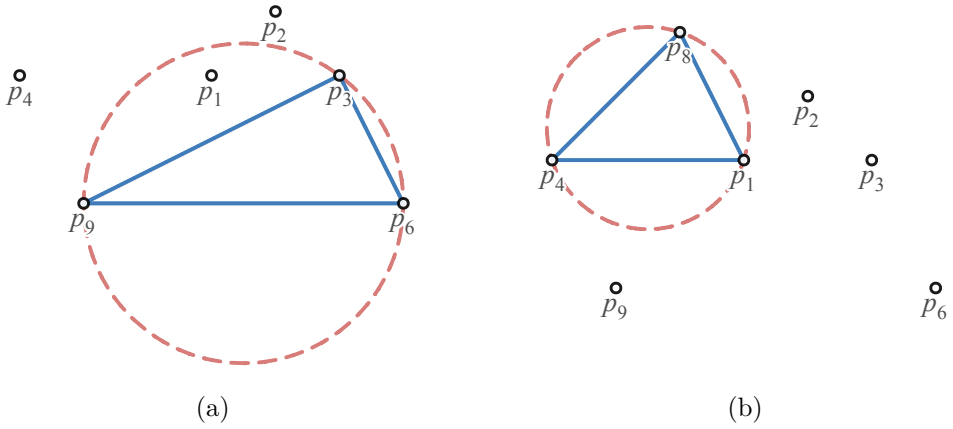
Triangulace v trojrozměrném prostoru bývá někdy označována jako tetrahedralizace. Jak již bylo zmíněno pro prostor $\mathbb{R}^3$ jsou trojice (p_1, p_2, p_3) v $T(P)$ nahrazeny čtveřicemi (p_1, p_2, p_3, p_4) reprezentujícími čtyřstěny. Podmínky pro p_4 jsou pak dále rozšířeny tak, aby bylo zajištěno, že se tento bod v jedné čtveřici nevyskytuje vícekrát. Pro triangulaci $T(P)$ musí dále platit, že se stěny simplexů navzájem neprotínají [2]. V triangulaci spolu mohou simplexy sdílet pouze body, hrany nebo stěny. V $\mathbb{R}^2$ spolu nikdy nemohou sdílet ani část své plochy a v $\mathbb{R}^3$ se nikdy nesmí protínat objemem.



Obrázek 2.3: Příklad Delaunayho triangulace

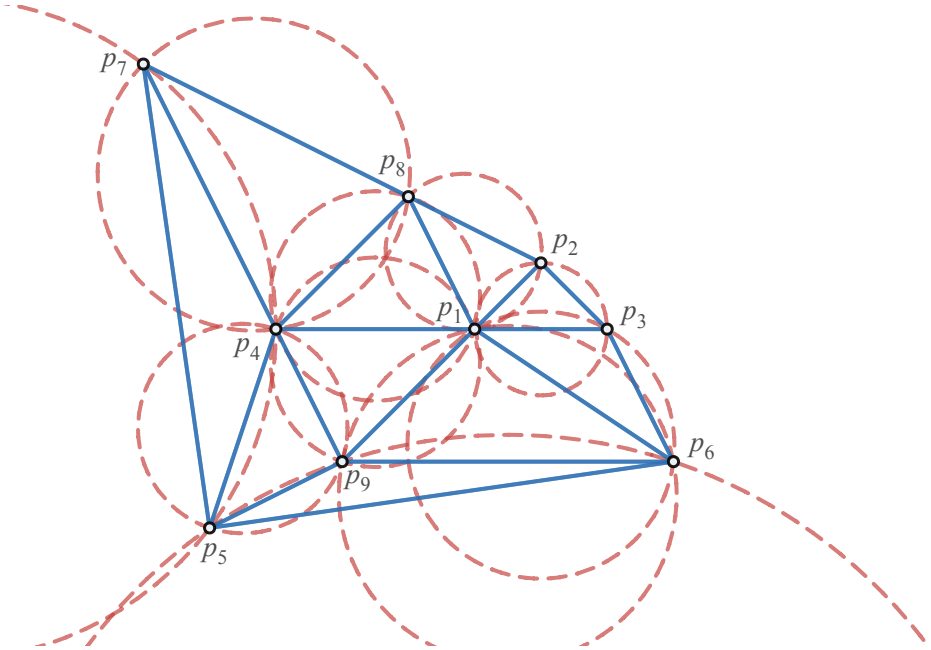
Delaunayho triangulace (dále také zkracované jako DT) je pak speciální typ triangulace, kde musí být zajištěno Delaunayho kritérium $K_D(t, P)$ pro každý simplex t z T. Musí platit, že opsaná hyperkoule simplexu (v prostoru $\mathbb{R}^2$ kruh) obsahuje pouze body tohoto simplexu a žádné jiné. Na jejím povrchu ale cizí body můžou být. Definice 2.5 formálně definuje popsanou Delaunayho triangulaci. Toto kritérium je graficky zobrazeno na obrázcích 2.4.

$$T_D(P) = \{(p_1, p_2, p_3) : K_D((p_1, p_2, p_3))\} \tag{2.5}$$



Obrázek 2.4: Ukázka porušení a dodržení Delaunayho kritéria:
První z těchto obrázků zobrazuje případ, kdy bylo Delaunayho kritérium porušeno. V kružnici je obsažen cizí bod. Druhý obrázek zase zobrazuje dodržení tohoto kritéria. Zobrazený trojúhelník je možné použít ve validní DT, jelikož neobsahuje žádné další body.

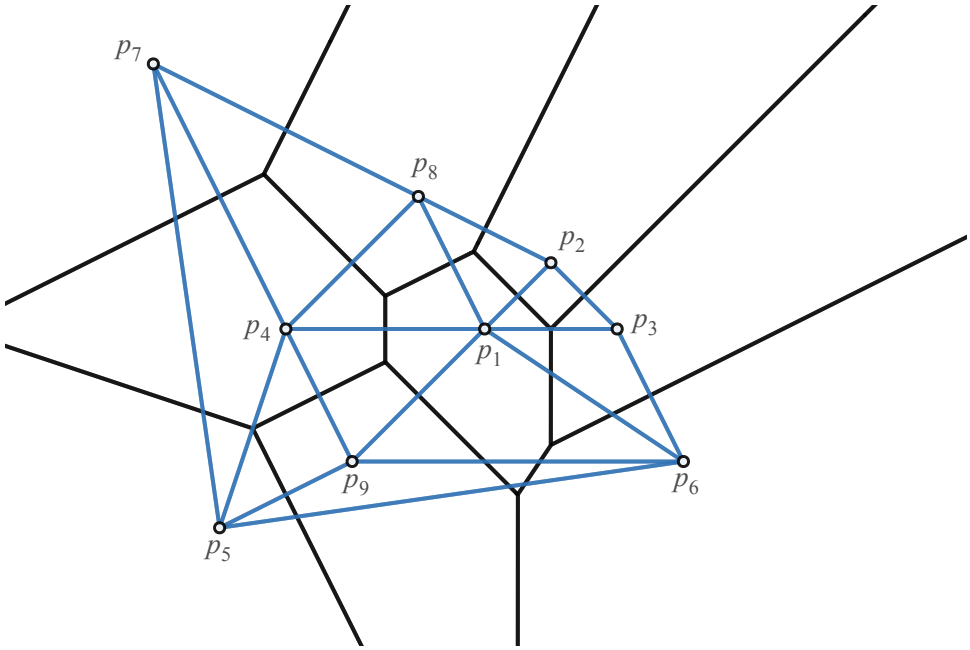
Jak již bylo řečeno toto kritérium musí platit pro všechny simplexy DT. Na obrázku 2.5 jsou pak vykresleny všechny opsané kružnice pro triangulaci z obrázku 2.3. Je zde možné vidět, že každý z trojúhelníků splňuje toto kritérium a že se skutečně jedná o platnou Delaunayho triangulaci.



Obrázek 2.5: Grafická ukázka platnosti Delaunayho kritéria v Delaunayho triangulaci

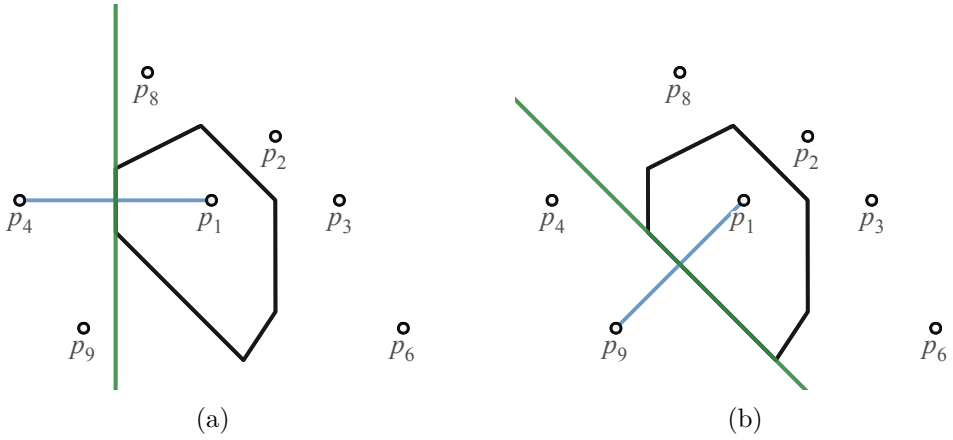
2.3 Princip duality

Dualita ve výpočetní geometrii je často využívána k transformaci špatně řešitelného problému na jiný jednodušší problém. O dualitě geometrických útvarů se hovoří, pokud z vlastností jedné struktury lze určit vlastnosti struktury druhé. Například je možné určit, zda body leží na jedné přímce v kartézské soustavě souřadnic podle toho, zda reprezentace těchto bodů v paralelních souřadnicích se protíná v jednom bodě [4].



Obrázek 2.6: Ilustrace duality Delaunayho triangulace a Voroného diagramu

Delaunayho triangulace je útvar duální k Voroného diagramu [2]. To znamená, že mezi těmito geometriemi je určitá závislost. Pro VD lze určit zda spolu buňky sousedí podle toho, jestli jsou jádra těchto buněk v rámci DT spojena hranami. Jak je možno vidět na obrázku 2.6, stěny Voroného diagramu jsou kolmé na hrany Delaunayho triangulace. Proto je možné provést mezi těmito strukturami převod pomocí využití ořezových hyperrovin, viz obrázek 2.7.



Obrázek 2.7: Ukázka ořezových hyperrovin získaných z Delaunayho triangulace: Tyto hyperroviny lze použít k převodu na Voroného diagram a jsou definovány jako množina všech bodů se stejnou vzdáleností k oběma sousedícím bodům. Buňka VD pak vznikne ořezáním prostoru pomocí všech hyperrovin definovaných pomocí jádra dané buňky a jader sousedících buněk.

2.4 Definice a rovnice analytické geometrie

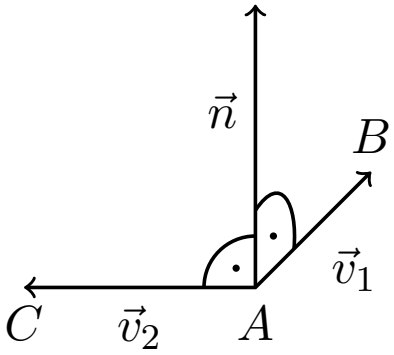
Pomocí matematických vzorců je možné popsat mnoho geometrických útvarů. Část geometrie, která se těmito výpočty a vzorci zabývá, se nazývá analytická. V této podkapitole jsou dále popsány analytické vzorce geometrických útvarů potřebných v rámci této práce. Jedná se o normálu, rovinu a přímku.

2.4.1 Výpočet normály trojúhelníku

Pro každý trojúhelník lze jednoduše vypočítat jeho normálu. Pro vrcholy trojúhelníku označené jako A, B, C a normálu $\vec{n}$ lze odvodit vektory $\vec{v}_1$ a $\vec{v}_2$. První z nich $\vec{v}_1$ se rovná rozdílu bodů C a A tedy $(C_x - A_x, C_y - A_y, C_z - A_z)$ a druhý $\vec{v}_2$ se zase rovná rozdílu bodů A a B tedy $(B_x - A_x, B_y - A_y, B_z - A_z)$. Pak normálu lze získat vektorovým součinem pomocí rovnice 2.6.

$$\vec{n} = \vec{v}_1 \times \vec{v}_2 \tag{2.6}$$

Směr normály vzniklé podle této rovnice je určen tím, jak jsou body trojúhelníku posazeny v prostoru. Normála vždy drží pravé úhly s vektory $\vec{v}_1$ a $\vec{v}_2$. Pokud se pozorovatel v prostoru orientuje tak, aby bylo možné body A, B a C propojit popořadě pohybem proti směru hodinových ručiček, pak je směr normály v rámci této orientace vzhůru k pozorovateli. Toto pravidlo je označováno jako pravidlo pravé ruky. Na obrázku 2.8 jsou pro lepší představu zobrazena relevantní konfigurace bodů a vzniklý směr normály.



Obrázek 2.8: Orientace bodů trojúhelníku A, B, C a vzniklé normály pro $\vec{v}_1 \times \vec{v}_2$

2.4.2 Rovnice roviny

V analytické geometrii je možné rovinu definovat jako jednu rovnici. Dosazením hodnot do ní a jejím výpočtem je možné zjistit pozici bodu vůči takto definované rovině. Takováto rovnice roviny je uvedena ve vzorci 2.7.

$$Ax + By + Cz + D = 0 \tag{2.7}$$

kde x, y a z jsou proměnné definující bod $[x, y, z]$ v třírozměrném prostoru. A, B, C a D jsou pak koeficienty určující pozici a orientaci dané roviny. Takovouto rovnici je možné vytvořit, pokud je známa normála $\vec{n}$ této roviny a bod reprezentovaný vektorem $\vec{p}$ na ní ležící. První tři koeficienty jsou získány dosazením $A = n_x$, $B = n_y$ a $C = n_z$. Poslední koeficient D je vypočítán dosazením $x = p_x$, $y = p_y$ a $z = p_z$ do neúplné rovnice. Tímto způsobem vznikne rovnice 2.8.

$$D = -(n_x p_x + n_y p_y + n_z p_z) \tag{2.8}$$

Rovnice $Ax + By + Cz + D = w$ nám pak umožní určit pozici bodu $[x, y, z]$ vůči rovině touto rovnicí definovanou následovně:

- $w = 0$: bod leží na rovině
- $w > 0$: bod leží mimo rovinu po směru normály
- $w < 0$: bod leží mimo rovinu proti směru normály

2.4.3 Rovnice přímky

Pro reprezentaci přímky ve trojrozměrném prostoru je možné použít parametrické rovnice ve tvaru zobrazeném v soustavě rovnic 2.9.

$$\begin{aligned} x &= d_x t + o_x \\ y &= d_y t + o_y \\ z &= d_z t + o_z \end{aligned} \tag{2.9}$$

Vektor $\vec{d}$ určuje směr přímky a $\vec{o}$ určuje počátek, tj. bod který je získán dosazením $t = 0$ do rovnic. Všechny body na takto definované přímce lze získat postupným dosazováním hodnot z intervalu $(-\infty, \infty)$ za parametr t.

Tento systém rovnic má pak řešení pro všechny body, které leží na dané přímce. Ostatní body řešení nemají. Pokud je třeba takto definovat úsečku, lze toho docílit správným určením vektorů $\vec{d}$ a $\vec{o}$. Pro úsečku s koncovými body A a B je možné tyto vektory určit pomocí rovnic 2.10 a 2.11.

$$\vec{o} = (A_x, A_y, A_z) \tag{2.10}$$

$$\vec{d} = (B_x - A_x, B_y - A_y, B_z - A_z) \tag{2.11}$$

Všechny body na takto definované úsečce lze získat postupným dosazováním hodnot z intervalu $\langle 0, 1 \rangle$ za parametr t.

2.4.4 Průsečík roviny a úsečky

Pro výpočet průsečíku roviny a úsečky je možné využít výše popsané rovnice. Dosazením parametrických rovnic přímky do rovnice roviny vznikne rovnice 2.12.

$$\begin{aligned} Ad_x t + Ao_x + Bd_y t + Bo_y + Cd_z t + Co_z + D &= 0 \\ Ad_x t + Bd_y t + Cd_z t &= -(Ao_x + Bo_y + Co_z + D) \\ t &= -\frac{Ao_x + Bo_y + Co_z + D}{Ad_x + Bd_y + Cd_z} \end{aligned} \tag{2.12}$$

Nakonec pokud dosadíme parametr t do rovnic přímky získáme bod, ve kterém tato přímka protíná danou rovinu, tedy průsečík. Dále pro hodnoty t v rámci úsečky platí, že:

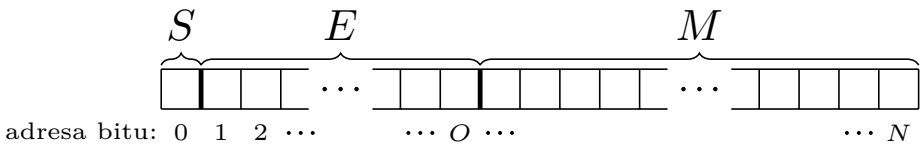
- $t \geq 0 \wedge t \leq 1$: průsečík leží na úsečce
- $t < 0 \vee t > 1$: průsečík leží mimo úsečku

2.5 Reprezentace desetinných čísel

Dnešní počítače jsou založeny na principech práce s informacemi reprezentovanými diskrétními hodnotami. To znamená, že všechny základní typy informací používané v procesorech jsou reprezentovány jako celá čísla o pevně daném počtu číslic. Pokud mají být uchovány jiné informace, jako například znaky abecedy nebo desetinná čísla, musí být použit převod na diskrétní celočíselnou informaci.

Desetinná čísla lze reprezentovat mnoha způsoby. Nejčastěji se ale používají čísla s plovoucí čárkou [7]. Další způsob je reprezentovat čísla s pevnou desetinnou čárkou. Čísla s fixní čárkou jsou jednodušší. Pracuje se s nimi velmi podobně jako s celými čísly. Fungují na principu implicitní desetinné čárky s předem určenou pozicí v rámci číslic.

Čísla s plovoucí čárkou jsou komplikovanější a procesory s nimi zpravidla pracují pomaleji. Za cenu této komplexity ale umožňují reprezentovat daleko větší rozsah čísel. Díky své flexibilitě je tento přístup k ukládání desetinných čísel tím populárnějším z těchto dvou možných reprezentací.



Obrázek 2.9: Organizace bitů reprezentujících číslo s plovoucí čárkou:
S označuje znaménkový bit, E označuje bity exponentu a M označuje bity mantisy. Dále jsou zde vyznačeny adresy jednotlivých bitů. Adresy začínají na nejvýznamnějším a pokračují k méně významným bitům.

Na obrázku 2.9 je zobrazena bitová reprezentace čísla s plovoucí čárkou o $N+1$ bitech, kde bit na adrese 0 je nejvýznamnější a na adrese N je nejméně významný. Jednotlivé sekce této reprezentace pak mají následující význam:

- S – znaménkový bit určuje přítomnost znaménka mínus, jeho hodnota 1 znamená záporné číslo a 0 znamená opak
- E – tyto bity reprezentují exponent, ten určuje rozmezí hodnot, které mantisa reprezentuje
- M – tyto bity reprezentují mantisu, ta vytváří mřížku v rozmezí určeném exponentem

Exponent

Hodnota exponentu se ukládá ve speciálním formátu na pozicích bitů označených na obrázku 2.9 jako E. Pro zpětné získání hodnoty v desítkové soustavě stačí bity exponentu přímo převést do desítkové soustavy a poté od získané hodnoty odečíst konstantu o. Tato konstanta je získána podle následujícího vzorce 2.13.

$$o = 2^{O-1} - 1 \tag{2.13}$$

Hodnota O odpovídá indexu posledního bitu E, viz obrázek 2.9. O pak odpovídá počtu bitů použitých k reprezentaci exponentu. Z bitové reprezentace exponentu například 0100 je pak tímto způsobem získána výslední hodnota exponentu -3. Konstanta O je pro uvedený exponent 4 a o je tedy 7. Dále je ještě třeba zmínit, že hodnoty exponentu reprezentované samými jedničkami nebo samými nulami jsou rezervované pro speciální případy. Ty pak umožňují reprezentovat například číslo 0 nebo nekonečno.

Další důležitou konstantou pro exponent je báze B. Zpravidla se používá dvojka nebo desítka jako její hodnota. Intuitivně lze exponent pak chápat tak, že v kombinaci s bází určuje hrubost a pozici mřížky na ose hodnot.

Mantisa

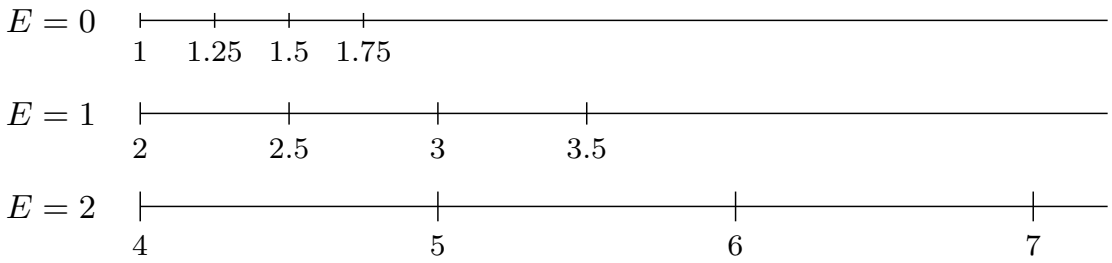
Hodnota mantisy se ukládá ve speciálním formátu na pozici bitů označených jako M na obrázku 2.9. Pro získání opravdové hodnoty je třeba před bity přidat ještě implicitní jedničku a desetinnou čárku. Mantisa 0100 pak odpovídá hodnotě 1,0100. Tu už stačí převést z dvojkové soustavy do desítkové na 1,25. Intuitivně lze mantisu pak chápat tak, že určuje pozici výsledné hodnoty na mřížce, viz obrázek 2.10.

Výsledné číslo

Jakmile je známa mantisa a exponent, lze výslednou hodnotu získat pomocí vzorce 2.14.

$$(-1)^S \cdot M \cdot B^E \tag{2.14}$$

Z výše popsaných vlastností a tohoto vzorce je možné určit, že pro bázi B rovnu dvěma vznikne mřížka podobná obrázku 2.10. Hodnoty, které nejsou na této mřížce přítomny nelze reprezentovat pomocí čísla s plovoucí čárkou s dvoubitovou mantisou. Vícebitová mantisa by hustěji zaplnila mřížku. Vždy se ale najdou čísla, která na mřížce chybí.



Obrázek 2.10: Ukázka mřížky vzniklé pro dvoubitovou mantisu a exponent E

Tato diskrétní mřížka je také hlavním zdrojem chyb v reprezentaci reálných čísel. Aby se čísla mohla zobrazit na mřížku musí být nějakým způsobem zaokrouhlena na hodnotu, která se na mřížce přímo nachází.

Z výše popsaných vlastností také vyplývá, že porovnávání dvou kladných čísel s plovoucí čárkou je možné provést dosti jednoduše. Porovnání může být provedeno stejně jako u celých čísel. To je možné díky tomu, že pozice jednotlivých bitů odpovídá jejich vlivu na velikost reprezentovaného čísla. Pro porovnávání i záporných čísel je ale třeba už porovnávání mírně pozměnit.

Dopady na výpočetní geometrii

Jelikož je geometrie přirozeně řešena ve spojitém prostoru, vzniká zde problém s reprezentací takového prostoru pomocí diskrétních hodnot. Existuje mnoho řešení s mnoha různými výhodami, nevýhodami a využitími. Jedním z triviálních řešení je při porovnávání dvou desetinných čísel a a b kontrolovat zda jsou tyto hodnoty v určitém rozmezí od sebe. Rozmezí se určuje proměnnou η. Porovnání na rovnost pak může vypadat jako rovnice 2.15.

$$a > b - \eta \wedge a < b + \eta \tag{2.15}$$

Toto řešení lze využít pouze pro určité problémy. Zdaleka neřeší všechny překážky, které musí výpočetní geometrie překonat s ohledem na reprezentaci desetinných čísel. Při návrhu algoritmů výpočetní geometrie je třeba znát nedostatky reprezentace čísel s plovoucí čárkou. Naštěstí některé problémy už jsou efektivně vyřešené, jako například implementace robustních geometrických predikátů [13].

Kapitola 3

Algoritmy a postupy pro paralelní vytváření Voroného diagramu

Tato kapitola se zaměřuje na popis algoritmů a postupů, které budou v rámci této práce použity. V sekci 3.1 jsou nejdříve obecněji popsány přístupy k vytváření Voroného diagramu. V další sekci 3.2 je popsán algoritmus vytváření Delaunayho triangulace [10] a na konci této sekce je také popsán algoritmus převodu VD na DT. Následující sekce 3.3 popisuje metodu Voronoi shape trimming. V rámci další sekce 3.4 je pak popsán řadící algoritmus radix sort a jeho možné varianty. Nakonec v sekci 3.5 jsou popsány techniky umožňující paralelizaci některých problémů.

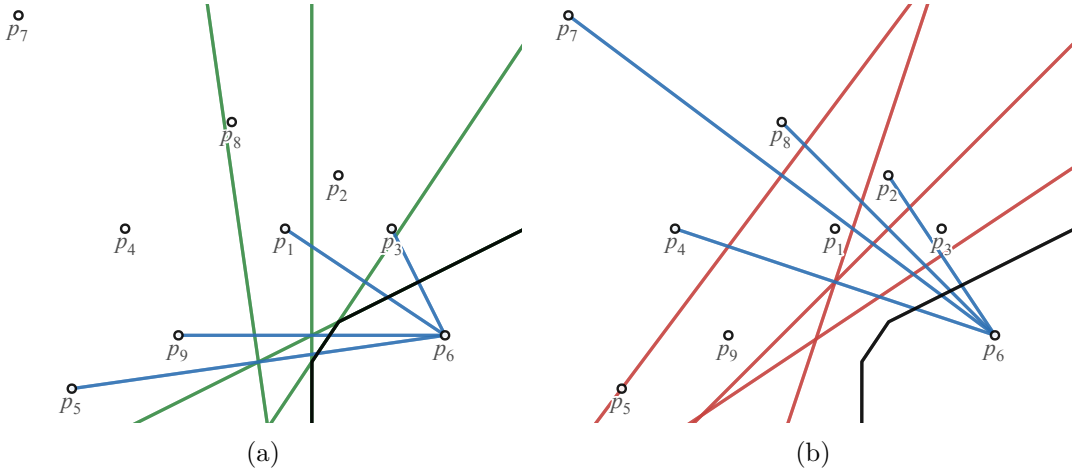
3.1 Přístupy ke konstrukci Voroného diagramu

Existuje mnoho algoritmů, které řeší problém vytváření Voroného diagramu. Některé prostor dělí do diskrétních částí, jiné se zase snaží aproximovat spojitý prostor. Některé jsou vhodné pro použití ve trojrozměrném prostoru, jiné jsou méně vhodné. Další důležité kritérium těchto algoritmů pro tuto práci je možnost efektivní paralelizace.

Pro vytváření Voroného diagramu ve 2D existuje mnoho algoritmů a obecně je tento problém považován za efektivně řešitelný. Jedním z nejvíce popularizovaných je Fortunův algoritmus [6]. Tento algoritmus pracuje na principu rozdělení 2D prostoru na dvě poloroviny, kde jedna polorovina je právě řešena a druhá ještě řešena není. Postupně se prochází prostor posouváním této přímky. Tuto metodu je obtížné paralelizovat, což ji dělá nevhodnou pro tuto práci.

Další metody využívají diskrétního rozdělení prostoru například na krychle. Pro tyto krychle se vypočítá nejbližší jádro buňky a podle toho je daná krychle přidělena k jedné z buněk. Přesnost těchto metod kvůli své diskrétní podstatě z velké části záleží na frekvenci, se kterou je prostor dělen.

Pro malé množství jader může také být možné použití primitivnějších přístupů, tzv. „brute force“. Tyto přístupy zpravidla dominují jednoduchostí implementace. Jejich hlavní nevýhodou ale bývá to, že pro větší množství dat bývají neefektivní, což ale paralelizace může řešit. Tyto metody mohou například vytvářet ořezové hyperroviny mezi všemi body a pak buňky vytvářet postupně každou zvlášť pomocí těchto hyperrovin, viz 3.1. Případně můžou opět využívat diskretizace prostoru a pro každý prvek prostoru můžou řešit jeho příslušnost k jedné z buněk.



Obrázek 3.1: Ukázka naivní tvorby Voroného buňky pomocí ořezových hyperrovin: Na prvním obrázku jsou zeleně vyznačeny okrajovací hyperroviny, které mají efekt na výsledný tvar vytvořené buňky. Naopak na druhém obrázku jsou vyznačeny červeně hyperroviny, které tvar výsledné buňky neovlivňují.

Poslední zmíněný přístup je převod z Delaunayho triangulace. Jak již bylo zmíněno v kapitole 2.3 díky tomu, že Delaunayho triangulace a Voroného Diagram jsou duální, je možné získat z Delaunayho triangulace hyperroviny definující buňky Voroného diagramu. Pro tento přístup je samozřejmě nejdříve třeba tuto triangulaci vytvořit. Jeden z populárních způsobů tvorby DT spočívá v postupném přidávání bodů do triangulace. Pomocí transformací simplexů je upravována tak, aby bylo vždy splněno Delaunayho kritérium [12].

3.2 Algoritmus sestavení DT a převodu na VD

První z vybraných algoritmů vhodných pro GPU vytváří nejdříve Delaunayho triangulaci [10]. Ta je až posléze transformována na Voroného diagram. Hlavní myšlenkou je použít vytvořenou DT k výběru sousedních bodů pro tvorbu buněk VD.

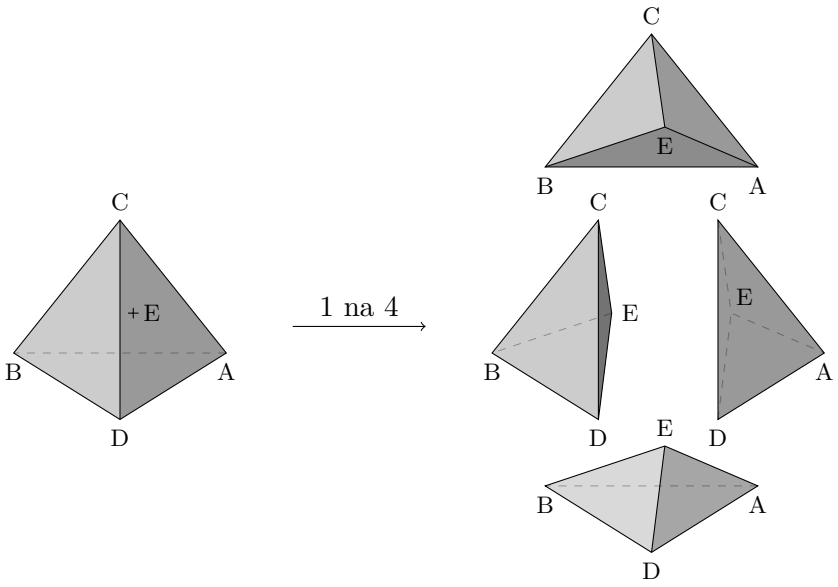
DT je vytvářena postupně vkládáním bodů a následovným transformováním triangulace tak, aby byla co nejkvalitnější, tedy aby co nejvíce splňovala Delaunayho kritérium. V každé iteraci algoritmu se nejdříve na začátku vloží nové body, jeden bod pro každý čtyřstěn, a pak se provedou zmiňované transformace. Po skončení této smyčky dojde k nápravě nevhodných čtyřstěnů, které mohou během algoritmu vzniknout.

```
1 gFlip3D(S)
2 |  while !S.empty()
3 |  |  pointInsert(S,T)
4 |  |  flipTri(S,T)
5 |  T = refineTriangulation(T)
6 |  return T
```

Výpis 3.1: Pseudokód hlavní části algoritmu pro sestavení DT [10]

Vkládání bodů

V každém kole vždy jeden bod může být vložen do již existujícího čtyřstěnu. Při přidání bodu dochází k zaniknutí původního čtyřstěnu a vzniknou čtyři nové čtyřstěny, viz 3.2. Toto štěpení je podobné transformacím, které jsou popsány v dalších podkapitolách. Tato transformace je také označována jako „flip 1-4“.



Obrázek 3.2: Štěpení čtyřstěnu (A, B, C, D) na čtyři nové vložením bodu E

Každý čtyřstěn má přiřazenou množinu zatím nevložených bodů, které jsou uvnitř něj. Z této množiny je každé kolo vybírán jeden bod. Vždy je to ten, který je nejblíže středu opsané koule daného čtyřstěnu. Nakonec je třeba aktualizovat všechny potřebné geometrické informace, jako jsou nové sousednosti čtyřstěnů, a je třeba přerozdělit nevložené body původního čtyřstěnu mezi ty nové.

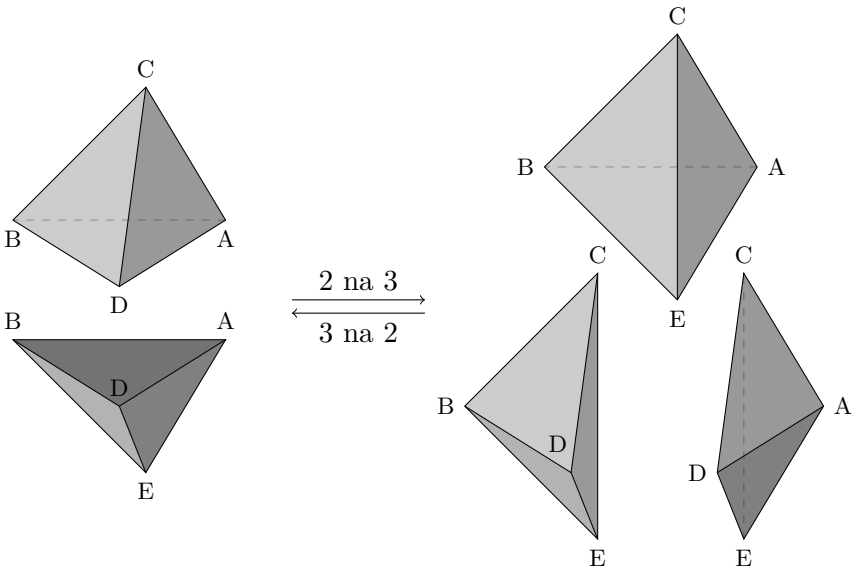
Vkládání bodů probíhá paralelně a lze ho provádět s minimální synchronizací. Paralelizace je pro tuto část algoritmu navržena tak, že při hledání vhodného bodu ke vložení může každý bod být zpracován paralelně. Jakmile je rozhodnuto o tomto bodu provede se jeho vložení. Každý čtyřstěn může v tu chvíli být štěpen vlastním vláknem. V dalším kroku jsou aktualizovány sousednosti. Každý nově vytvořený čtyřstěn paralelně informuje sousedy o své přítomnosti. Tento krok je navržen tak, že nevyžaduje žádnou synchronizaci. Nakonec je pro všechny body paralelně vypočítána jejich příslušnost k nově vytvořeným čtyřstěnům.

```
1 pointInsert(S,T)
2 |  for t in T.tetrahedra
3 |  |  if needsInserting(S,t)
4 |  |  |  p = pickPoint(S,t)
5 |  |  |  splitTetrahedron(t,p)
6 |  |  |  S.remove(p)
7 |  |  |  redistributePoints(S,T)
```

Výpis 3.2: Pseudokód pro vkládání bodů do triangulace v algoritmu sestavujícím DT [10]

Transformace

Existují dva druhy transformací již vytvořených čtyřstěnů: transformace tří čtyřstěnů na dva a transformace dvou čtyřstěnů na tři (také „flip 3-2“ a „flip 2-3“). Tyto transformace jsou graficky zobrazeny na obrázku 3.3. K transformacím čtyřstěnů pak dochází pouze tehdy, když pro tyto čtyřstěny neplatí Delaunayho kritérium popsané v kapitole 2.2.



Obrázek 3.3: Transformace dvou čtyřstěnů na tři a naopak

Během tohoto kroku je třeba nejdříve identifikovat čtyřstěny vhodné k transformacím. Provede se test na Delaunayho kritérium. Pokud jsou nalezeny dva nebo tři čtyřstěny ve správné konfiguraci, je tato konfigurace označena. Pokud existuje konflikt mezi takovými konfiguracemi, vybere se pouze jedna a ta se provede. Transformace se hledají a provádějí v cyklu, dokud už nelze nalézt žádnou další vhodnou konfiguraci. Na konci každé iterace tohoto cyklu jsou aktualizovány geometrické informace, jako například sousednost nových čtyřstěnů a příslušnost nevložených bodů k nim.

Nejdříve se hledají vhodné konfigurace pro každý čtyřstěn paralelně. Konflikty mezi nimi se řeší bez explicitní synchronizace. Přednost má konfigurace s nižším indexem vlákna, které tuto transformaci našlo. Jakmile jsou nalezeny všechny vhodné konfigurace, dojde k paralelnímu provedení transformací. Následné nastavování sousedností se pro každý nový čtyřstěn provádí paralelně. Znovu je synchronizace minimalizována. Pomocí pomocných struktur se dohledá, jak spolu nové čtyřstěny sousedí. Nakonec jsou přerozděleny body. Každý z nich může být samostatně zpracován vlastním vláknem.

```
flipTri(S,T)
  changed = true
  while changed
    changed = false
    for f in T.faces
      t0, t1 = getSharedTetrahedra(f,T)
      p1 = t1 - f
      if !isDelaunay(t0,p1)
        if isAnyFlipPossible()
          if !isFlipConflict()
            performAvailableFlip()
    changed = redistributePoints(S,T)
```

Výpis 3.3: Pseudokód pro transformaci simplexů v algoritmu sestavujícím DT [10]

Převod z Delaunayho triangulace na Voroného diagram

Nakonec se výsledná triangulace převede na jednotlivé buňky pomocí ořezových rovin. Roviny jsou získávány pro každé jádro buňky pomocí sousedních bodů. Jako sousední jsou chápány všechny body, které jsou s daným jádrem přímo spojeny hranou v rámci DT. Ořezová rovina je pak definována jako množina všech bodů, které mají stejnou vzdálenost k oběma těmto sousedícím jádrům. Ta je potom použita k odřezání přebytečné části buňky. Po ořezání buňky je zaplněna vzniklá díra novou stěnou. Paralelizace této části je navržena, tak že každá buňka je obsluhována jedním vláknem s minimální synchronizací mezi sebou. Pokud je VD sestrojován pro okamžité vykreslení vytvoří se s buňkou nakonec i záznam s informacemi o buňce potřebné pro vykreslení na grafické kartě.

```
createVoronoiDiagram(S)
  DT = gFlip3D(S)
  VD = newVoronoiDiagram()
  for s~in
S~c = newCell()
    N = emptyList()
    while n = findNextNeighbour(DT,s,N)
      N.append(n)
      cutCell(c,s,n)
    addNewCell(VD,c)
  return VD
```

Výpis 3.4: Pseudokód algoritmu pro převod DT na VD

3.3 Metoda Voronoi shape trimming

Tento přístup k vytváření VD je zde nazván metodou, protože na rozdíl od algoritmu se nejedná o úplně přesný návod jak postupovat. Metoda Voronoi shape trimming (dále také VST nebo Voroného okrajování těles) pouze určuje, že se pro okrajování původního tělesa postupně vybírají body od těch nejbližších k jádru buňky a okrajování je ukončeno po

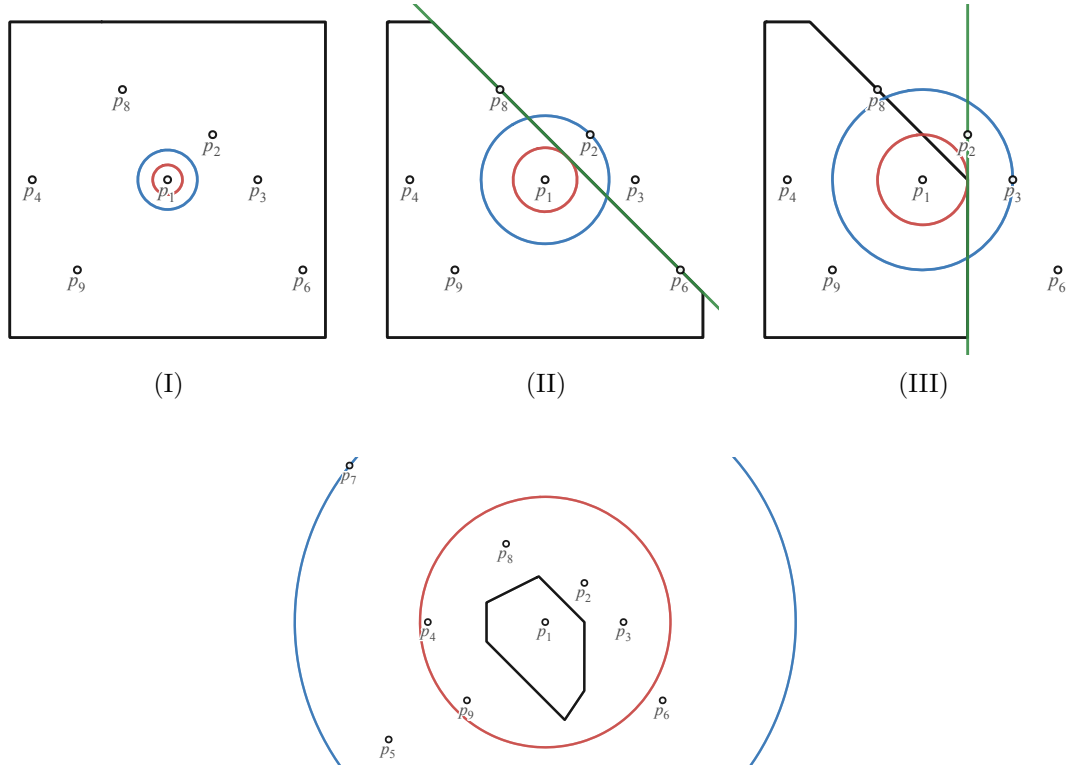
splnění kritéria. Není přímo určeno, jakým způsobem mají být reprezentovány buňky nebo jak má probíhat okrajování.

Tato metoda funguje ze dvou hlavních důvodů. Voroného buňku lze vytvořit naivním ořezáním všemi možnými ořezovými hyperrovinami. Na prvním obrázku z 3.1 lze vidět, že stačí použít pouze ořezové hyperroviny sousedních buněk k vytvoření validní buňky. Na druhém obrázku z 3.1 zase naopak je vidět, že použití dalších ořezových hyperrovin definovaných jádry buněk, které nesousedí s vytvářenou buňkou, nijak výsledný tvar buňky neovlivní. Proto nevadí, že k okrajování tělesa jsou použity i nesprávné body. Pro korektnost vzniklých buněk záleží pouze na tom, aby byla použita všechna jádra sousedních buněk VD. Druhá důležitá myšlenka této metody se týká kritéria pro ukončení okrajování buňky. Toto kritérium je přesněji definováno níže, viz výpis 3.5 a doplňující text pod tímto výpisem. Funkčnost tohoto kritéria je zajištěna tím, jak je vytvářena ořezová hyperrovina. Ta vždy vzniká uprostřed mezi jádrem buňky a bodem, který je zrovna zvolen pro okrajování. Pokud je nejvzdálenější okraj buňky vzdálen od jádra méně než okrajovací rovina, logicky nedojde k ořezání buňky. Všechny další body, které jsou dále vybírány už nemůžou být blíže než právě vybraný bod, a tím pádem všechny další ořezové roviny jsou také vzdálenější. Tyto vlastnosti jasně zajišťují, že žádné další ořezávání už není potřebné.

```
1  VST(s,S)
2  |  cell = newCell();
3  |  while !S.empty()
4  |  |  closest, point = findClosestPoint(s,S)
5  |  |  S~= S~- point
6  |  |  furthest = findFurthestVertex(cell)
7  |  |  if closest < 2 * furthest
8  |  |  |  cell = cutEdges(cell,s,point)
9  |  |  |  cell = createNewFace(cell)
10 |  |  else break
11 |  return cell
```

Výpis 3.5: Pseudokód okrajovacího algoritmu, převzato z [3]

Na výpisu 3.5 je možno vidět pseudokód vytváření jedné buňky. Každé kolo algoritmu si nejdřív hledá nejbližší bod k jádru buňky, který ještě nebyl použit k okrajování. Pokud je splněna podmínka, že tento nejbližší bod je vzdálen od jádra buňky méně než dvojnásobek vzdálenosti nejvzdálenější části buňky od tohoto jádra, má smysl dále buňku okrajovat. Nejdříve se buňka ořízne hyperrovinou definovanou jádrem buňky a vybraným bodem. Potom je vznikla díra zalepena. Pokud jsou všechny body použity k ořezání nebo byla výše zmíněná podmínka porušena je ořezávání buňky ukončeno. Paralelizace je zde řešena tak, že každá buňka může být zpracovávána jedním vláknem. Dále je možné více paralelizovat vyhledávání nejbližšího bodu, ale toto už záleží na jednotlivých implementacích metody. Na obrázcích 3.4 je pak možné vidět demonstraci několika kroků této metody.



Obrázek 3.4: Ukázka několika kroků metody VST

Na prvních třech obrázcích jsou nultý, první a druhý krok metody VST. Na těchto obrázcích lze dobře vidět proces postupného ořezávání původního tělesa. Zde se jedná o čtverec, což lze vidět na prvním obrázku. Dále je zde zobrazena vzdálenost nejbližšího bodu reprezentována modrou kružnicí. Červená kružnice reprezentuje polovinu této vzdálenosti. Ta se využívá v kritériu pro skončení ořezávání. Na posledním obrázku je možné sledovat konec algoritmu splněním tohoto kritéria. Jeho splnění potvrzuje červená kružnice, která je za hranicí buňky. To znamená, že vybraný bod pro ořezávání je více než dvojnásobně daleko než nejvzdálenější bod buňky.

3.4 Radix sort

Na rozdíl od většiny řadících algoritmů se v rámci radixového řazení nevyužívá porovnávání hodnot jednotlivých prvků pole mezi sebou. Místo toho dochází k postupnému rozdělování čísel do tříd. Prvky se pak přesouvají v poli podle příslušnosti do těchto tříd a znovu se rozdělují do tříd [9].

Slovo radix v matematice znamená: báze číselné soustavy. Tedy popisuje počet unikátních číslic, které lze při reprezentaci čísel v dané soustavě použít. Podle jednotlivých číslic daných prvků se pak hodnoty dělí do těchto tříd. U každé třídy známe jejich vztah k ostatním třídám a podle těchto vztahů je možné hodnoty v rámci pole přesouvat a tím ve výsledku i seřadit. Pro desítkovou soustavu jsou třídy následující:

$$0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$$

Číslice pro třídění jsou vybírány postupně buď od nejvíce významné (nejdříve 1092 a v dalším kroku 1092), nebo naopak od té nejméně významné (nejdříve 1092 a v dalším kroku 1092). Při práci s bity se tato významnost také označuje jako MSB (most significant bit, tedy nejvýznamnější bit) a LSB (least significant bit, tedy nejméně významný bit).

Počet tříd tedy záleží na číselné soustavě, ve které jsou čísla reprezentována. V rámci této práce se řeší verze algoritmu pracující v binární soustavě a jako číslice tedy jsou přímo použity samotné bity bezznaménkových (anglicky „unsigned“) reprezentací čísel. Níže popsány jsou pak dvě varianty tohoto algoritmu, které se liší přístupem ke směru výběru číslic. Pro obě verze algoritmu jsou využívány bitové masky k výběru kontrolovaného bitu.

Od nejvýznamnější číslice

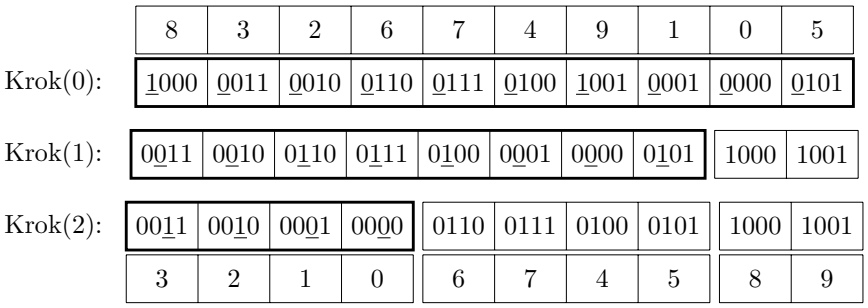
Hlavní výhodou verze algoritmu od nejvýznamnější číslice je, že čísla jsou ihned dělena do tříd, kde každá ze tříd je vůči ostatním prvkům seřazena. Toto umožňuje seřadit například jen část pole nebo používat hybridní přístup, kdy řazení jednotlivých tříd je dokončeno jiným algoritmem. Bohužel ale rekurzivní podstata celého algoritmu komplikuje implementaci na GPU, kde klasická rekurze není povolena. Toto je hlavní nevýhodou tohoto přístupu, což může značně komplikovat implementaci.

```
1  radixSortMSB(L)
2  |  bitCount = bitSizeOfOneElement(L)
3  |  currentBit = 1 << (bitCount - 1)
4  |  radixSort(L,currentBit)
5
6  radixSortMSB(L,currentBit)
7  |  if L.size() == 1 || currentBit == 0
8  |  |  return L
9  |  L1 = emptyList()
10 |  L2 = emptyList()
11 |  for e in L
12 |  |  if currentBit & e == 0
13 |  |  |  L1.append(e)
14 |  |  else
15 |  |  |  L2.append(e)
16 |  currentBit = currentBit >> 1
17 |  L1 = radixSort(L1,currentBit)
18 |  L2 = radixSort(L2,currentBit)
19 |  L = L1.extend(L2)
20 |  return L
```

Výpis 3.6: Pseudokód algoritmu radix sort od nejvýznamnějšího bitu [9]

Tato verze algoritmu pro každou iteraci vytváří dva nové seznamy reprezentující dvě třídy. Do těchto seznamů přiřazuje postupně příslušné prvky původního seznamu podle právě kontrolovaného bitu určeného proměnnou `currentBit`. Nakonec se rekurzivně zavolá pro oba tyto nové seznamy algoritmus znovu s bity v proměnné `currentBit` posunutými doprava a výsledky těchto dvou volání jsou spojeny do finálního seznamu který je nakonec vrácen. Rekurze je ukončena, pokud není již co třídit nebo byly použity pro třídění všechny bity.

Běh algoritmu z výpisu 3.6 je možné sledovat na následujícím obrázku 3.5, kde je možné vidět postupné rekurzivní dělení do tříd. Pro každou z těchto tříd platí, že prvky nalevo od této třídy jsou menší a napravo jsou větší než kterýkoli z prvků dané třídy.



Obrázek 3.5: Ukázka algoritmu radix sort od nejvýznamnější číslice:
Pole je postupně rozdělováno do tříd. Rekurzivně zpracovávaná část pole je zde zvýrazněna hrubým okrajem. Na konci lze vidět, že před dokončením algoritmu jsou třídy mezi sebou již seřazeny.

Od nejméně významné číslice

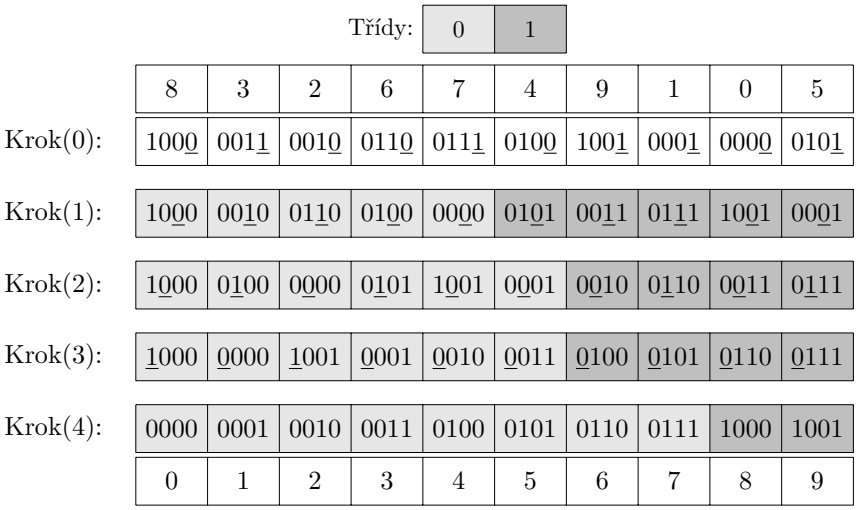
Hlavní výhodou přístupu od nejméně významné číslice je jednoduchost implementace bez využití rekurze. Toto je velmi výhodné pro implementaci na GPU, kde rekurze není možná. Nevýhodou zase je, že celý seznam čísel je seřazen až na konci běhu algoritmu. V průběhu třídění nedochází k jasnému řazení spojitých částí pole.

```
1  radixSortLSB(L)
2  |  currentBit = 1
3  |  bitStop = 1 << bitSizeOfOneElement(L)
4  |  while currentBit != bitStop
5  |  |  L1 = emptyList()
6  |  |  L2 = emptyList()
7  |  |  for e in L
8  |  |  |  if currentBit & e == 0
9  |  |  |  |  L1.append(e)
10 |  |  |  else
11 |  |  |  |  L2.append(e)
12 |  |  L = L1.extend(L2)
13 |  |  currentBit << 1
14 |  return L
```

Výpis 3.7: Pseudokód algoritmu radix sort od nejméně významného bitu [9]

Tato verze algoritmu v hlavním cyklu posouvá bity proměnné `currentBit` zprava doleva. Tato proměnná se používá k výběru kontrolovaného bitu. Pří každé iteraci třídění se vytváří dva seznamy reprezentující třídy. Tyto seznamy jsou postupně plněny příslušnými hodnotami. Nakonec jsou tyto dva seznamy spojeny dohromady a výsledek je vložen do původního seznamu a cyklus se opakuje. Konec cyklu zajišťuje proměnná `bitStop`, která obsahuje konfiguraci bitů po posledním relevantním testu.

Algoritmus z výpisu 3.7 je možné sledovat na následujícím obrázku 3.6. Je možné vidět, jak se při rozdělování do tříd prvky promíchávají. Vždy si ale pro určitý krok v rámci jedné třídy zachovávají svoji pozici vůči ostatním prvkům této třídy. Tímto způsobem dojde až v posledním kroku k úplnému seřazení pole.

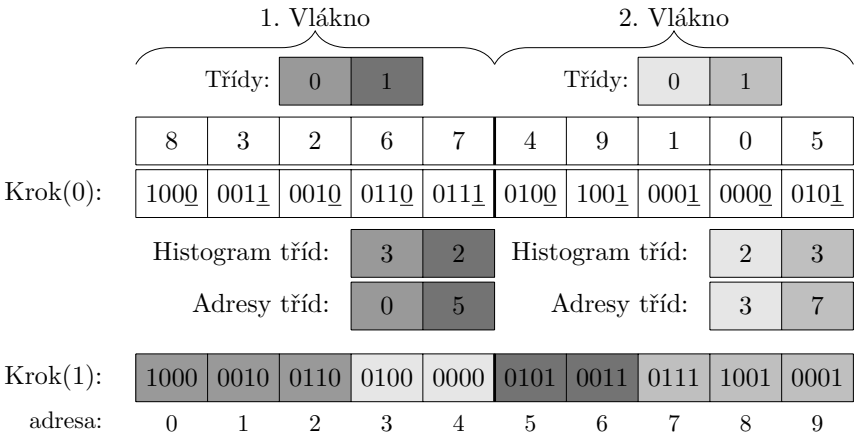


Obrázek 3.6: Ukázka algoritmu radix sort od nejméně významné číslice
Při každé iteraci algoritmu se celé pole přerozděluje do dvou tříd podle jednoho z bitů. Třídy jsou barevně rozlišeny.

Práce s pamětí

Aby byla zaručena vhodnější práce s pamětí a aby byly použity datové struktury vhodnější pro GPU, je možné seznamy v algoritmu nahradit poli. Pro efektivní přesouvání hodnot rovnou na odpovídající pozice v poli, je třeba nejdříve při prvním průchodu polem spočítat počet jednotlivých prvků v rámci každé třídy. Pro paralelní implementaci s více procesy si musí každý z procesů vypočítat počty prvků tříd v části pole, která jim byla přidělena. Až poté se vypočítají adresy, na které bude třeba prvky přenést, viz obrázek 3.7.

Dalším možným vylepšením je kontrola více bitů na jednou. V takovém případě se zvyšuje množství tříd. Vždy se ale jedná o mocniny dvojky. Například při kontrole dvou bitů najednou by byly třídy čtyři a pro tři bity by jich bylo osm. Tímto způsobem by se dalo snížit množství přístupů do paměti vzhledem k tomu, že by bylo třeba méně iterací třídění za cenu větší náročnosti jednotlivých kroků třídění.



Obrázek 3.7: Ukázka určení adresy tříd pro paralelizovaný algoritmus radix sort [15]: Výpočet adres pro přesný přesun hodnot na svá místa v rámci třídy probíhá tak, že je nejdříve vypočítán histogram prvků tříd. Ten si každé vlákno počítá zvlášť v rámci jemu přidělené části pole. Dále je možné z těchto histogramů vypočítat adresy, kam má každé vlákno přemisťovat jednotlivé prvky.

3.5 Optimalizace algoritmů pro paralelní výpočty

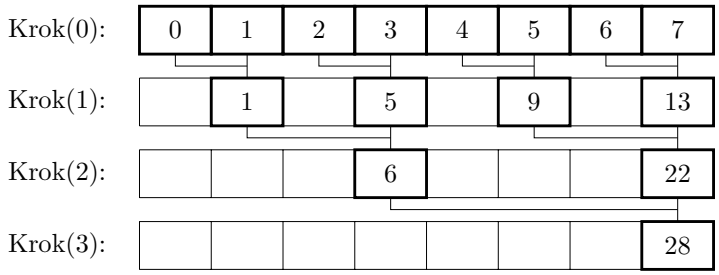
Paralelizace algoritmů umožňuje až několikanásobně zvyšovat výkon aplikace, nese s sebou ale různá úskalí. Nejčastějšími problémy jsou závislosti operací na sobě a konflikty při přístupu k paměti. Pokud není použit vhodný algoritmus nebo zajištěna správná synchronizace mezi procesy, může dojít ke korupci dat. Navíc velká míra synchronizace procesů zhoršuje výkonnost. Proto se vyvinuly různé přístupy k tomu, jak se některým těmto neduhům vyhnout a relativně efektivně paralelizovat určité částí algoritmů. Zde například patří cykly, ve kterých se prochází pole a operace nad jejich prvky jsou závislé na výsledcích předchozích iterací.

3.5.1 Paralelní redukce

Tento postup využívá asociativity operací. Tedy pokud nezáleží na pořadí, v jakém jsou operace prováděny a závisí jich více na sobě, je možné použít paralelní redukci [8].

Ideální způsob, jak tento postup použít je při použití $\frac{n}{2}$ procesů pro n prvkové pole. Potom pro první iteraci se nejdřív provede $\frac{n}{2}$ operací, tak aby byly využity všechny prvky. V následujících krocích se provádí vždy polovina operací oproti kroku předešlému. Tyto operace se provádí nad výsledky předešlých iterací až nakonec je získán jeden poslední výsledek.

Pro cyklus, kde na n prvků se aplikuje asociativní binární operace a každá iterace závisí na výsledku té předešlé, je možné tento cyklus o n iteracích zkrátit na $log_2(n)$ kroků. Tento postup je graficky ilustrován na obrázku 3.8.

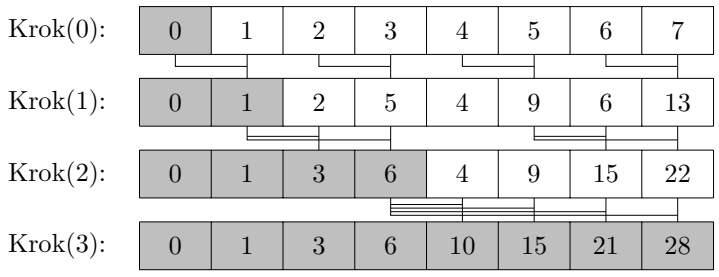


Obrázek 3.8: Ilustrace paralelní redukce pro součet pole:
Mezivýsledky pro každou iteraci jsou zvýrazněny hrubým okrajem. Každou iteraci pracuje polovina vláken než v té předešlé. Operace nad prvky pole je součet.

3.5.2 Paralelní prefixový součet

Tento postup je podobný paralelní redukci, místo jednoho výsledku je ale potřeba získat n výsledných hodnot pro n prvků pole. Zpravidla je cílem pro každý prvek pole získat jako výsledek součet všech předcházejících hodnot a daného prvku. Existují i varianty, které nepřičítají k celému součtu daný prvek. Zobecněná verze tohoto postupu se anglicky nazývá „parallel scan“ [8].

Tento problém lze ideálně paralelizovat pro n prvkové pole tak, aby místo cyklu o n iteracích výpočet proběhl v $log_2(n)$ krocích. Tato verze algoritmu je ale méně efektivní než jiné implementace, které ale zase zaberou více kroků. Při použití masivní paralelizace na GPU je vhodné se soustředit na efektivitu implementace a správně přerozdělovat práci vláknům tak, aby byly vždy zaměstnané. Jedna z možných verzí paralelního prefixového součtu je ilustrována na obrázku 3.9.



Obrázek 3.9: Ilustrace paralelního prefixového součtu:
Šedou barvou jsou zde vyznačeny prvky pole s již vypočítaným konečným výsledkem. Při každé iteraci dochází ke čtyřem součtům.

Kapitola 4

Principy grafických karet

Grafické karty (dále také označované GPU) pracují odlišně oproti dnešním běžně dostupným procesorům (dále také CPU). Proto se i při vytváření programů pro ně musí brát ohledy vůči těmto odlišnostem. Tato kapitola se zabývá specifickými vlastnostmi grafických karet.

Nejdříve je v sekci 4.1 popsán vývoj grafických karet a jejich využití. Dále je v této sekci nastíněn moderní přístup k vykreslování na GPU. V další sekci 4.2 je nastíněno použití grafických karet pro obecné výpočty. V rámci této sekce jsou popsány různé typy paměti, které se na GPU vyskytují. Dále je popsán způsob, jakým je vykonáván kód. Nakonec jsou zde zmíněny způsoby, jak docílit co největšího výkonu.

4.1 Hardwarová akcelerace vykreslování

Vytváření grafického výstupu aplikací může být výpočetně velmi náročný úkol, zvláště pokud je zpracováván pouze sériovým způsobem. Jelikož je obrazový rastr zpravidla dvourozměrný, jakékoli zvětšení rozlišení může mnohonásobně vytváření obrazu zpomalit. Vyvinuli se proto grafické akcelerátory, které dokáží zpracovávat větší množství obrazových bodů (pixelů) najednou.

Historie

Nejdříve byly grafické akcelerátory koncipovány pouze jako hardwarové urychlovače některých funkcí používaných při vykreslování obrazu [14]. Tyto karty nebylo možné programovat, místo toho se pouze volaly fixní funkce, kterým byly předávány argumenty. Další vývoj vedl k zavedení programovatelných částí. Program určený pro vykonání na grafické kartě se anglicky nazývá „shader“. Jelikož neexistuje žádné pěkné české slovo pro pojmenování těchto programů, bude dále používána počeštěná verze anglického termínu. Data se začala předávat ve větším množství a nároky na paměť rostly. S postupně se zvyšujícím výkonem začala také růst relativní cena režie procesoru. Dnes je snaha předat co nejvíce dat grafické kartě a co nejméně voláními pak tyto data vykreslit. Během tohoto vývoje se také začaly objevovat tendence využívat grafickou kartu k obecným výpočtům. Tento koncept je nazýván GPGPU (general purpose GPU).

Moderní vykreslovací techniky

Jak již bylo popsáno výše, moderní vykreslovací techniky se snaží minimalizovat komunikaci mezi GPU a procesorem. Jeden z těchto přístupů se nazývá „instanced rendering“. Tato

technika umožňuje vykreslit jeden model vícekrát v různých variacích pomocí jednoho volání z CPU. O něco pokročilejší je pak technika „indirect rendering“ (nepřímé vykreslování), ta přidává možnost vytvořit pole argumentů, které jsou dále použity k několika vykreslovacím voláním [1]. Procesor pouze grafické kartě oznámí typ těchto volání a kolik je v tomto poli připravených argumentů. GPU se pak už postará o zbytek. Toto umožňuje vykreslit více různých modelů, a ještě k tomu každý z nich vícekrát. Prakticky je tímto způsobem možné vykreslit celou scénu, nebo aspoň všechny objekty v ní, které používají stejný shader.

4.2 Obecné výpočty na GPU

Jakmile se GPU začali otevírat možnostem programovatelnosti, začali se také otevírat možnostem využití jejich potenciálně obrovského výkonu pro obecnější výpočty. Během adaptace ze specificky zaměřeného masivně paralelního grafického akcelerátoru na obecněji zaměřenou výpočetní jednotku se vyvinuly jiné přístupy k řešení problémů, než se používají u CPU. Kód se vykonává jiným způsobem i s pamětí se pracuje jinak. Pro odemknutí maximálního potenciálu GPU je třeba tyto rozdíly znát a umět je správně využít. Tato kapitola nejdůležitější specifika práce s GPU přiblíží.

4.2.1 Rozdělení paměti

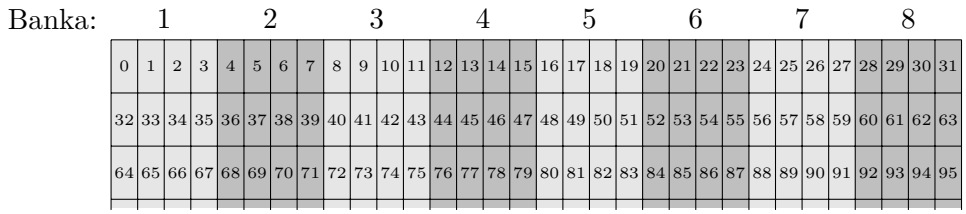
Paměť se na GPU dělí na různé typy, kde každý z nich má určité výhody a nevýhody. Ty určují, jakým způsobem je třeba paměť využívat, k jakému účelu se hodí.Mezi tyto typy patří i globální, sdílená a konstantní paměť. Posledním typem paměti jsou registry. Hlavními vlastnostmi, kterými se jednotlivé typy od sebe liší, jsou jejich rychlost, možnosti přístupu nebo způsob práce s nimi.

Globální paměť

Tento typ paměti na GPU bývá zpravidla nejhojnější. V moderních grafických kartách často nabízí gigabyty místa. Její hlavní nevýhodou je pomalá rychlost v porovnání s ostatními typy. Načtení dat zpravidla trvá déle než jeden cyklus multiprocesoru. Tento problém je řešen, pomocí přepínání vláken, tento koncept je popsán níže v podkapitole 4.2.2.

Sdílená paměť

Každý multiprocesor má svou vlastní paměť na čipu. Její velikost je v desítkách kilobytů pro každý multiprocesor a přístup do ní zajišťují banky. Ty dokážou během jednoho cyklu přečíst 4 byty. Takže pokud během jednoho taktu je požadován přístup k více než 4 bytům této banky, dojde ke konfliktu a čtení ze sdílené paměti pak trvá déle. Je tedy důležité zajistit vhodný způsob rozdělení dat různých vláken v lokální paměti. Ilustrace rozdělení paměti do bank je zobrazena na obrázku 4.1



Obrázek 4.1: Ukázka rozložení paměti při použití 8 bank:
Banky jsou rozloženy vždy tak, aby umožnili přístup vláknům k spojitému bloku paměti během jednoho taktu.

Ostatní typy paměti

Dalším typem paměti je konstantní. Je rychlejší než globální, může z ní číst jakékoli vlákno, ale žádné do ní namůže zapisovat. Speciální typ globální paměti je lokální paměť. Z hlediska výkonu se jedná o obyčejnou globální paměť. Hlavní rozdíl je v tom, že tato paměť je přístupná pouze lokálně, ne pro více vláken. Nakonec jsou v GPU registry. Ty jsou ze všech typů paměti nejrychlejší, ale je jich méně než jiné paměti. Na každý multiprocesor spadá množství registrů v rámci tisíců.

4.2.2 Vykonávání kódu

Obecně jsou dnešní GPU optimalizovány pro masivní paralelizaci výpočtů. Hlavním uplatňovaným principem je vykonávání jedné operace zároveň nad větším množstvím dat, tzv. SIMD (single instruction multiple data). Kód vykonávaný na GPU je překládán a nahráván až za běhu programu. Kernel (také shader) se pak nazývá jeden program pro GPU.

Vlákno v rámci GPU je možné chápat jako jednu invokaci kernelu. Každé vlákno má číslo, které ho identifikuje a v rámci každého musí být znám stav registrů a jaká instrukce bude následovat. Vlákna jsou vykonávána po skupinách, pro které je v rámci této práce použito jméno warp. Jedná se o názvosloví firmy NVIDIA. Firma AMD zase používá název wavefront. Předpokládá se, že v ideálním případě všechna vlákna ve warpu procházejí stejnou část kódu a snaží se v jednu chvíli provést stejnou instrukci. Ta je pro všechny vlákna vykonána najednou. Toto umožňuje multiprocesor. Tedy se u GPU vykonává jedna instrukce pro více vláken a místo výrazu SIMD se přesněji používá označení SIMT (neboli single instruction multiple threads). V rámci warpu může také dojít k rozdělení cesty v programu, například kvůli podmíněnému skoku. Tomuto se říká divergence vláken. V takovém případě jednotlivé větve programu nemohou být vykonány paralelně a dochází ke zpomalení.

Warpy se dále sdružují do pracovních skupin. V rámci pracovní skupiny spolu vlákna mohou komunikovat a sdílet paměť což umožňuje efektivnější spolupráci. Je možné zajistit, aby na sebe v rámci pracovní skupiny všechna vlákna počkala. Je třeba také dodat, že všechna vlákna patřící do jedné pracovní skupiny jsou vykonávána na jednom multiprocesoru. V rámci GPU je přítomno několik takovýchto multiprocesorů což umožňuje další zvýšení paralelizace. Což také znamená, že pro plné využití výpočetního výkonu GPU je zpravidla potřeba, aby pracovalo více skupin.

Jedním z hlavních principů, které umožňují maximální využití výpočetních prostředků GPU je vykrývání paměťových latencí pomocí výměny warpu, čekajícího na data za jiný, který má data již načtené a může být spuštěn ihned. Při dostatečném zatížení jednotlivých

výpočetních jednotek na GPU (tzv. okupanci) je možné tyto paměťové latence úplně vyblokovat a dostat tak maximální výkon z dostupných výpočetních prostředků. Nakonec je třeba poukázat na fakt, že kernely zpravidla neumožňují psaní komplikovaných programů a některé koncepty používané v programování na CPU nejsou přítomné. Jedná se například o rekurzi nebo ukazatele. Některé tyto omezení je však možné obcházet za použití jiných přístupů které umožňují podobné výsledky.

4.2.3 Výkon

Pro zajištění maximálního využití GPU je nejdůležitější dosáhnout vytíženosti jednotlivých multiprocesorů. Maximálně bude vytížená grafická karta tehdy, pokud na každém jejím multiprocesoru budou každý cyklus všechna vlákna warpu vykonávat užitečné instrukce. Tohoto lze docílit dvěma způsoby. Buď je naplánováno na každém multiprocesoru tolik vláken, že dojde k vykrytí veškeré latence globální paměti, nebo je paměť využívána tak efektivně, že je možné každý cyklus poskytnout data pro vykonávaná vlákna bez přepínání. Pak nedochází tak často k neefektivnosti způsobené latencemi. Prvního přístupu lze zpravidla dosáhnout minimalizací zdrojů, které vlákna využívají. Tím pádem je možné mít co největší počet vláken na jednom multiprocesoru. Aby tento přístup dával smysl, musí se zpracovávat velké množství dat. Tím pádem se využívá globální paměť a její hlavní nevýhoda, pomalý přístup k datům, je negován možností vykrývat latence.

Druhý přístup je vhodný pro případy, kdy se nezpracovává velké množství dat. V tomto případě je třeba využívat sdílenou paměť, registry a konstantní paměť na maximum. Tyto typy paměti umožňují načítat data dostatečně rychle, aby nebylo nutné čekat nebo přepínat warpy. V tomto případě vlákna využívají lokální zdroje multiprocesoru na maximum. Je ale nutné správně navrhnout algoritmus tak, aby byly tyto zdroje maximálně efektivně využity. Dalším důležitým faktorem pro maximalizaci výkonu je vyhýbání se obecně chyby při návrhu algoritmu. Tím je například špatný návrh větvení programu, který by mohl způsobit divergenci vláken. Další možný zabiják výkonu je tzv. „register spilling“. Toto označuje případ, kdy je v rámci multiprocesoru využíváno tak velké množství registrů, že dojdou a musí se využívat jiná, pomalejší paměť.

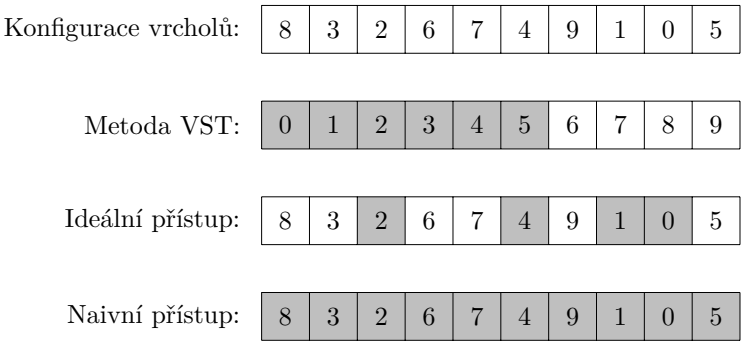
Kapitola 5

Návrh a teoretická analýza řešení

V této kapitole je popsán způsob využití teoretických znalostí popsaných v předešlých kapitolách. V sekci 5.1 je nastíněn hlavní rozdíl jednotlivých přístupů k vytváření Voroného diagramu. Další sekce 5.2 popisuje možné problémy spojené s metodou VST a způsoby, jak tuto metodu vylepšit. V následující sekci 5.3 jsou vysvětleny základní principy převodu Delaunayho triangulace na Voroného diagram. Poslední sekce 5.4 pak řeší způsoby, jak reprezentovat buňky a jak s těmito reprezentacemi pracovat. Je zde popsaná reprezentace pomocí trojúhelníků a také pomocí polygonů popsaných jejich hranami a vrcholy.

5.1 Teoretické porovnání přístupů k ořezávání objektů

Algoritmy vybrané v rámci této práce všechny využívají ořezávání pomocí okrajovacích hyperrovin. Z určitého pohledu na věc je možné tvrdit, že hlavním rozdílem mezi těmito metodami je způsob, jakým se vybírají body pro okrajování. Znalost Delaunayho triangulace umožňuje ideálně vybírat tyto body, viz obrázek 5.1.



Obrázek 5.1: Ukázka různých přístupů k výběru vrcholů k ořezávání:
Tento obrázek demonstruje různé přístupy k výběru bodů k okrajování buněk. Jednotlivé prvky polí reprezentují body množiny, pro kterou je VD vytvářen. Šedě zvýrazněné prvky reprezentují body, které jsou v rámci daného přístupu vybrány k ořezání buňky. V rámci metody VST může být vhodné vrcholy seřadit podle vzdálenosti, toto je vyznačeno seřazenou posloupností těchto bodů na obrázku. Dále je možné pozorovat, že v rámci této metody je potřeba použít méně bodů než při naivním přístupu. Ideální přístup zase využije jen ty body k ořezání buňky, které mají vliv na její konečný tvar.

Dále je na obrázku 5.1 zobrazen přístup metody VST, který v ideálním případě může být skoro stejně efektivní jako převod přímo z DT. V nejhorším případě ale odpovídá naivnímu přístupu. Toto lze také vidět v příloze B. Předpokládaná efektivita metody VST by měla záviset hlavně na typu distribuce náhodných bodů a na tom, jak dobře tyto body vyplňují rozřezávaný objekt. Toto chování lze sledovat v již zmiňované příloze. Na demonstraci z přílohy B.1 je možné vidět nejlepší možné chování metody. V tomto případě je způsobeno dvěma faktory. Za prvé jsou zde okolní body blízko a relativně na husto rozmístěné v okolí a za druhé je blízko i kraj ořezávaného objektu. Naopak u demonstrace z přílohy B.2 lze sledovat nejhorší možné chování metody. Body jsou zde okolo buňky rozmístěny ve více směrech velmi řídce a pouze v jednom směru jsou více koncentrovány. Navíc je okraj objektu daleko. Tato kombinace vlastností může vést k velkému rozměru buňky. Tímto se následně zvyšuje potřebná vzdálenost vybraného bodu pro splnění kritéria ukončení ořezávání.

5.2 Úskalí a vylepšení metody VST

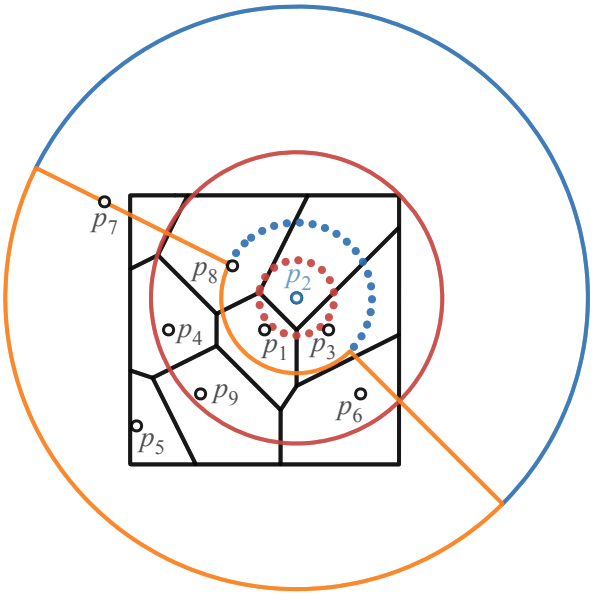
Jelikož celý postup není v rámci metody VST do detailu specifikován, je možné určité části algoritmu navrhnout více způsoby. V této sekci se řeší tyto části metody. Například výběr bodů od nejbližšího po nejvzdálenější lze řešit mnoha způsoby. Jsou zde diskutována možná vylepšení a případy, kdy by se mohla vyplatit. Dále zde jsou zmíněny potenciální úskalí této metody a jejich možná řešení.

Hlavním problémem této metody je citlivost na nevhodné rozložení bodů v objektu. Tento problém je částečně způsoben způsobem výběru bodů a ukončovacím kritériem. Tato a další témata budou dále popsána v této podkapitole.

Příliš velký obalový objekt a nevhodné rozložení bodů

Jak již bylo zmíněno výkon této metody závisí z velké části na tom, jakým způsobem jsou rozmístěny body v objektu. Předpoklad je takový, že metoda VST nejlépe funguje pro body uniformně rozdělené skrze celý objekt a se vzdálenostmi mezi sebou úměrné velikosti objektu. Příklad takového rozmístění bodů je možné vidět v příloze B.3. Hlavním problémem při nevhodném rozložení bodů v objektu je velikost vzniklé buňky. Vzdálenost bodů, které jsou použity k ořezávání, je dvojnásobek vzdálenosti nejvzdálenějšího bodu buňky od jejího jádra. Proto má velikost buňky a rozložení bodů velký vliv na množství bodů, které jsou vybrány k ořezání.

Na obrázku 5.2 je zobrazena taková v jednom směru příliš velká buňka. Tento směr má vliv na horní poloměr žluté zóny. Dále je možné vidět vliv velikosti buňky v prakticky opačném směru na spodní poloměr tohoto prostoru. Jelikož je tento bod na okraji množiny bodů vytvářející tento VD, určují i ostatní okrajové body hranici této žluté zóny.



Obrázek 5.2: Ukázka efektu relativně velké buňky na ukončovací kritérium: Tečkovaně je zobrazena vzdálenost bodů, která by ideálně měla zajistit ukončení ořezávání. Spojitou kružnicí je zobrazena opravdová vzdálenost pro ukončení. V žluté zóně se nacházejí body, které nemají vliv na konečný tvar buňky, ale budou použity k ořezávání.

V praxi nelze vždy zajistit ideální podmínky při vytváření VD, proto je třeba hledat vhodná řešení, jak tento nedostatek metody VST řešit. Jedno možné řešení je zavedení maximální vzdálenosti, do které se hledají v okolí sousední body. Tento přístup ale může způsobit nekorektní vytváření buněk. Pro velmi nepravidelné rozmístění jader objektem, kde v jedné části objektu je velká hustota těchto bodů a v jiné části jsou zase body rozmístěny řídce, není možné určit ideálně tuto maximální velikost. Pokud by ale byly body rozmístěny relativně pravidelně v jedné části objektu a na jiných místech by se vůbec nevyskytovali, mohl by tento přístup řešit daný problém relativně úspěšně. Zvláště pokud je zajištěno, že okrajové body nesousedí s jinými relativně dalekými body.

Další přístup k řešení tohoto problému, který by mohl přinést robustnější výsledky, se zdá být vylepšení samotného ukončovacího kritéria. Problém při pozdním ukončení ořezávání je často způsoben velikostí buňky v jednom směru a přítomností mnoha bodů ve směru jiném. Proto je vhodné v rámci kritéria posuzovat i informace o směrech, ve kterých se nachází okolní body a tvar buňky. Dále je toto řešení projednáváno v sekci níže.

Vylepšení kritéria

Kritérium, jak je definováno v metodě VST, může pro některé účely fungovat výborně. Pro jiné případy ale bohužel ideální není. V této části textu se řeší možná vylepšení tohoto kritéria. Jako jedna možnost se zdá být zavedení lokálního kritéria, které by řídilo, zda dojde k použití bodu k ořezání. Takové lokální kritérium by vyžadovalo použití funkce, která by aproximovala velikost buňky v každém směru. Samozřejmě by neumožňovalo ukončit celé ořezávání. Na to by byla vyžadována i informace o pozicích všech bodů v těchto směrech.

To že buňka nevyžaduje žádné další úpravy v jednom směru neznamená, že je nevyžaduje v nějakém jiném.

Jako možní kandidáti pro tento účel se jeví „support mapping“ funkce z algoritmu GJK [5], nebo použití aproximace pomocí obalového objektu, například osově zarovnaný kvádr (také AABB). Koule je pro tento účel nevhodná, jelikož hlavní kritérium ji už reprezentuje. Pro tuto aproximační funkci je důležité, aby byla obecně méně náročná než samotné okrajování.

Samozřejmě efektivita takového vylepšení záleží na složitosti implementované aproximační funkce, velikosti a tvaru buňky. Obecně se ale dá předpokládat, že pozitivní vliv by toto vylepšení mohlo mít hlavně v případech, kdy jsou body nevhodně rozmístěny v rozkrajovaném objektu. V takových případech může totiž docházet k většímu množství ořezávání, než je potřeba.

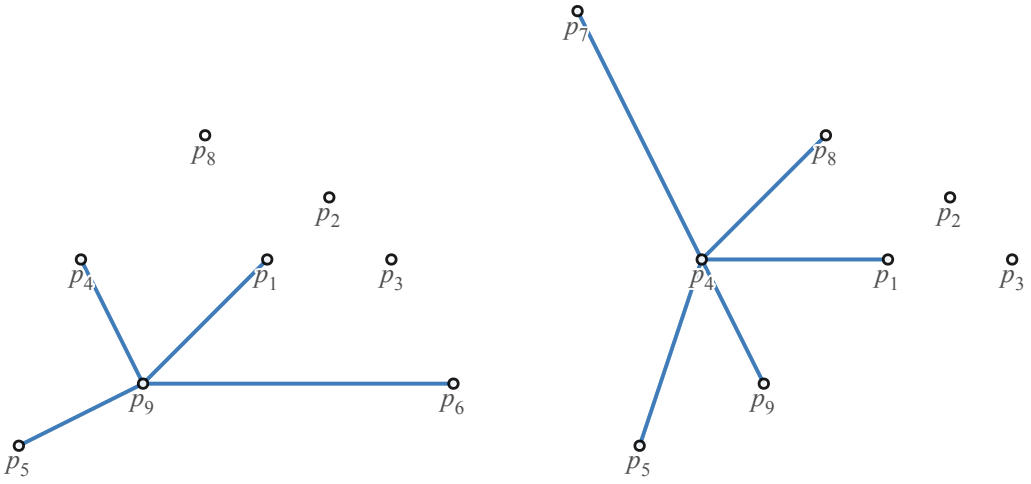
Seřazení bodů podle vzdálenosti

Další možné vylepšení, které by mohlo zefektivnit vytváření buněk je před začátkem samotného okrajování seřadit body podle vzdálenosti ke zpracovávanému jádru. Pak už není třeba při každé iteraci hledat další bod k ořezávání. Místo toho díky seřazení bodů je předem známa pozice dalšího bodu. Pro velké množství bodů se ale seřazování všech bodů nemusí vyplatit. Zvláště pokud jsou vhodně rozloženy v objektu, pak je třeba použít jen několik nejbližších bodů. Jak již bylo popsáno v sekci 3.4, některé verze algoritmu radix sort umožňuje částečné řazení pole hodnot. Díky této vlastnosti a relativně dobré možnosti paralelizace je algoritmus radix sort od nejvýznamnější číslice ideální jako počáteční bod pro výslednou implementaci řadícího algoritmu. Na GPU je zde hlavně třeba vyřešit nemožnost použití rekurze.

5.3 Převod z DT na VD

Pokud je známa Delaunayho triangulace, je možné ji využít k výběru sousedních bodů pro ořezávání buněk. K vytváření Voroného diagramu dochází obdobným způsobem jako u VST, výběr bodů je ale řešen jinak. DT obsahuje informace o všech sousedech buňky, nedochází tedy při převodu k žádným zbytečný ořezáváním. Formát zápisu Delaunayho triangulace, který je použitý v této práci a ze kterého bude docházet k převodu je podobný trojúhelníkové reprezentaci zobrazené na obrázku 5.4. Místo trojic jsou zde ale použity čtveřice reprezentující jednotlivé čtyřstěny. Tento zápis umožňuje hranu mezi dvěma sousedními body mít reprezentovanou ve více záznamech. Proto je vhodné pamatovat si body, které již byly použity k ořezávání, aby nedocházelo k jejich dalšímu zbytečnému použití. Pro vytvoření jedné buňky pak stačí najít všechny sousedy jejího jádra v triangulaci a použít je k ořezání.

Možným vylepšením tohoto postupu je předzpracování DT a převod na jiný způsob zápisu. Tuto reprezentaci vhodnou pro převod do VD je možné chápat jako seznam, kde každý záznam reprezentuje jeden z bodů triangulace. Každý takový záznam pak obsahuje seznam jeho sousedních bodů. Převod z tohoto formátu je velmi jednoduchý, jelikož každý záznam o bodu DT obsahuje přesný návod pro ořezávání příslušné buňky VD. Na obrázcích 5.3 je pak možné vidět „hvězdy“ vzniklé spojením jader VD s jádry sousedních buněk. Jak již bylo řečeno tento obrazec odpovídá DT. Přesněji odpovídá výše popsanému alternativnímu zápisu DT.



Obrázek 5.3: Grafická ukázka možné struktury reprezentující Delaunayho triangulaci: Na těchto obrázcích je graficky ilustrována reprezentace DT pomocí seznamu sousedů pro pro body p_9 a p_4.

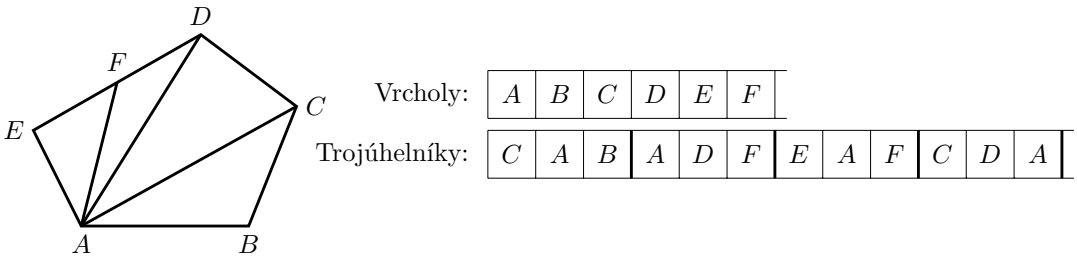
5.4 Reprezentace objektů a práce s nimi

Na počítači je možné modely reprezentovat mnoha způsoby. Často používaný způsob reprezentace objektů je pomocí trojúhelníků. Dále je ale možné modely reprezentovat i jinými způsoby, například voxely, pomocí mračen bodů nebo pomocí seznamů vrcholů, hran a stěn. Ne všechny jsou ale vhodné pro účely této práce. Dále je řešena trojúhelníková a polygonální reprezentace definovaná hranami a vrcholy. Bude popsána práce s těmito reprezentacemi a budou diskutovány jejich výhody a nevýhody.

5.4.1 Trojúhelníková reprezentace

Hlavní výhodou reprezentace buněk Voroného diagramu pomocí trojúhelníků je možnost využití přímo této reprezentace při vykreslování na GPU. Nevýhodou zase je, že pro reprezentaci jednoho polygonu je třeba zhruba jeden trojúhelník na jednu hranu a někdy i více. Tím pádem také dochází při ořezávání k více kontrolám.

Trojúhelníky jsou ukládány do paměti jako trojice. V rámci trojice záleží, v jakém pořadí jsou body zapsány. Toto může být důležité pro vytváření normál. Další informace o tomto jsou popsány níže. Na obrázku 5.4 lze vidět ilustraci stěny tvořené trojúhelníky a jejich datovou reprezentaci v paměti.



Obrázek 5.4: Ukázka trojúhelníkové reprezentace v prostoru a v paměti

Generování normál

Trojúhelníková reprezentace umožňuje jednoduše zajistit správnou orientaci všech trojúhelníků. Toto platí, pokud jsou všechny trojúhelníky původního rozkrajovaného objektu stejně orientovány. Jako orientaci trojúhelníku je možné chápat v jakém pořadí jsou body seřazeny v záznamu o daném trojúhelníku. Očekávají se vždy dvě možné orientace, ve směru hodinových ručiček nebo proti jejich směru. Tuto orientaci je možné určit kolmým pohledem na trojúhelník z vnějšku objektu. Je pak důležité zajistit, aby vždy pro všechny trojúhelníky byla zachována tato orientace a aby byla vždy určena stejným způsobem.

Například na obrázku 5.4 je možné pozorovat záznamy trojúhelníků v orientaci proti směru hodinových ručiček. Pro tuto konfiguraci trojúhelníku je možné použít přímo postup pro výpočet normály popsaný v podkapitole 2.4.1. Pro opačnou orientaci stačí postup modifikovat prohozením vektorů $\vec{v}_1$ a $\vec{v}_2$ v rovnici 2.6. Případně existuje ještě druhá možnost řešení korektního výpočtu normál. Tento postup je popsán v podkapitole 5.4.2 a využívá jádra buněk.

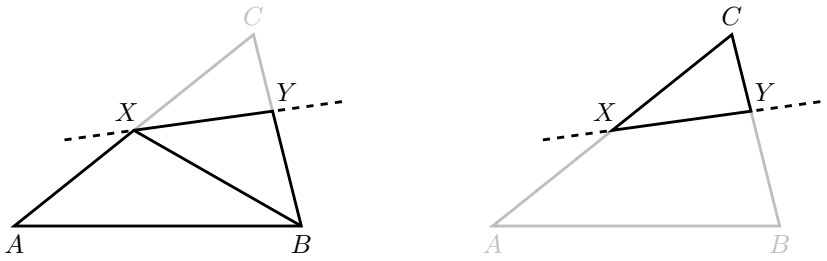
Ořezávání trojúhelníků

Při rozřezávání trojúhelníku rovinou dochází minimálně ke kontrole pozice tří bodů vůči ořezové rovině. Potom může dojít ke čtyřem různým možnostem:

- Všechny body trojúhelníku jsou odřezány.
- Dva body trojúhelníku jsou odřezány.
- Jeden bod trojúhelníku je odřezán.
- Žádný z bodů trojúhelníku není odřezaný.

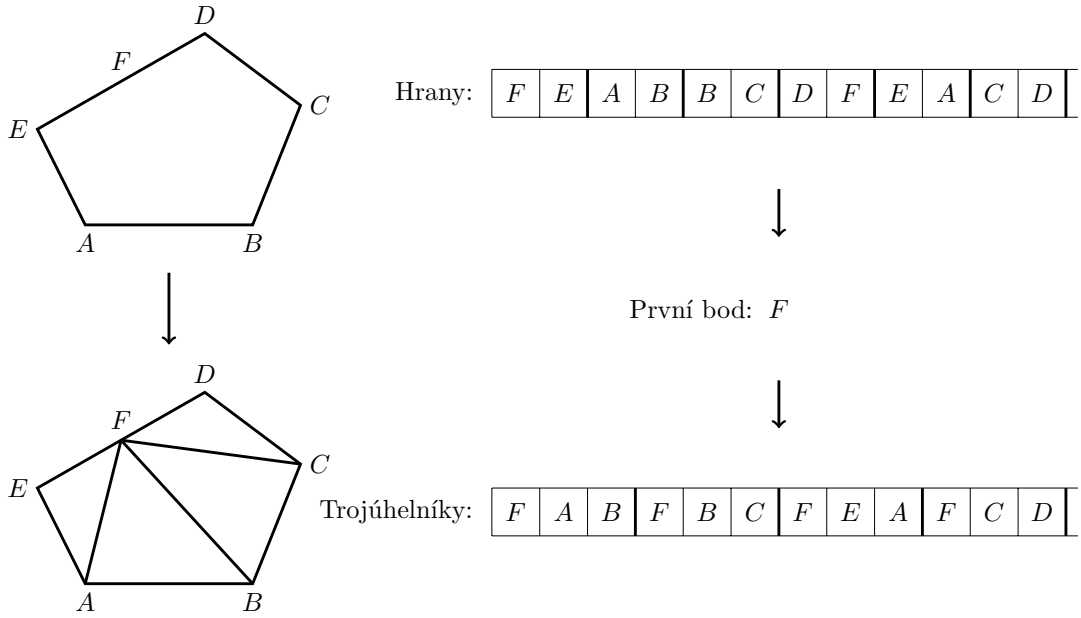
V případech, kdy se mají všechny body buď odřezat nebo se naopak nemá odřezat žádný z nich, není třeba provádět další testy. V těchto případech je celý trojúhelník rovnou odstraněn, nebo je naopak ponechán beze změny.

Na obrázku 5.5 jsou zobrazeny zbývající dva případy, které mohou nastat při ořezávání trojúhelníků. Část trojúhelníku, která je odřezána je vyznačena šedě. Při obou z nich vzniknou dva nové body. Na obrázku jsou tyto body označeny jako X a Y. Pokud jsou odkrojeny dva body, stačí pouze u konfigurace daného trojúhelníku tyto dva body vyměnit za nově vzniklé X a Y. V konfiguraci na obrázku X nahradí A a Y nahradí B.



Obrázek 5.5: Ukázka dvou případů ořezávání trojúhelníků

Složitější případ nastává, pokud je odkrojen pouze jeden bod. V tomto případě vzniknou dva nové trojúhelníky. Jeden z nich nahradí starou konfiguraci původního trojúhelníku. V konfiguraci na obrázku stačí nahradit bod C za nový bod X v rámci zpracovávaného trojúhelníku. Pro druhý vytvořený trojúhelník je pak třeba vymezit místo a vložit správně konfiguraci bodů. Aby byla zachována orientace bodů jako je na obrázku 5.4 je vhodné body seřadit následovně: X, B, Y.



Obrázek 5.6: Ukázka zaplnění polygonu trojúhelníky:

Seznam hran definuje okraj polygonu, který má být vyplněn trojúhelníky. Orientace bodů v rámci těchto hran ovlivní orientaci výsledných trojúhelníků. Je tedy potřeba zajistit jejich správnou orientaci. Seznam trojúhelníku pak vznikne spojením prvního bodu ze seznamu hran a jiných hran vhodných pro tento účel. Vhodné hrany neobsahují tento první bod a tím pádem mohou s tímto bodem ve výsledku vytvořit trojúhelník.

Zalepování okrajovaných děr

Po úspěšném dokončení ořezávání konvexního objektu rovinou zpravidla zůstane v síti trojúhelníků díra. Pro úspěšné vytváření Voroného diagramu je ji třeba uzavřít novou stěnou buňky. Tohoto lze docílit relativně jednoduše pro konvexní objekty. Při rozkrojení trojú-

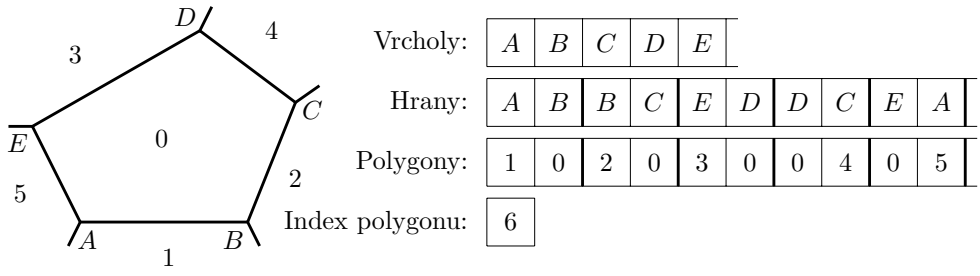
helníku totiž vždy dojde ke vzniku dvou nových bodů. Pro vytvoření nového trojúhelníku obsaženého ve stěně stačí k těmto dvěma bodům přidat už jen jeden bod této stěny. Tedy stačí si zapamatovat první vytvořený bod vzniklý při okrajování danou rovinou. Takže po každém dalším rozkrojení trojúhelníku může vzniknout nová část stěny ve formě nového trojúhelníku. Je si ale třeba dát pozor, aby jeden z nově vzniklých bodů se nerovnal prvnímu vytvořenému bodu. Tento proces je zobrazen na obrázku 5.6.

5.4.2 Polygonální reprezentace

Na rozdíl od trojúhelníkové reprezentace ta polygonální umožňuje ukládat jednotlivé stěny buněk pomocí menšího množství hran. Toto může zajistit menší množství kontrol hran při okrajování složitějších stěn buněk. Pro vykreslení výsledné buňky je ale nakonec třeba převést tuto reprezentaci na trojúhelníky.

Je více způsobů, jak reprezentovat tyto polygony v paměti. První možnost je reprezentace pomocí seznamu polygonů, kde každá položka reprezentuje jeden mnohoúhelník seznamem vrcholů. Ten je seřazen tak, že sousedící vrcholy vždy definují jednu hranu polygonu. První a poslední bod seznamu vytváří hranu také.

Další možnost záznamu je mít pouze jeden seznam, do kterého by se zapisovali přímo pouze hrany. Pro zjednodušení převodu na trojúhelníky je ale třeba znát příslušnost těchto hran k jednotlivým stěnám. Hrany vždy dělí dva sousední polygony. Pro každou zapsanou hranu tedy stačí zároveň zapsat i tyto stěny. Tento zápis je zobrazen na obrázku 5.7 a bude dále použit.



Obrázek 5.7: Ukázka polygonální reprezentace v prostoru a v paměti:
V rámci této reprezentace čísla reprezentují jednotlivé polygon a písmena zase vrcholy. Index polygonu odpovídá počtu vytvořených polygonů a tím pádem se zároveň jedná o index polygonu, který bude vytvořen při dalším úspěšném okrajování.

Ořezávání polygonů

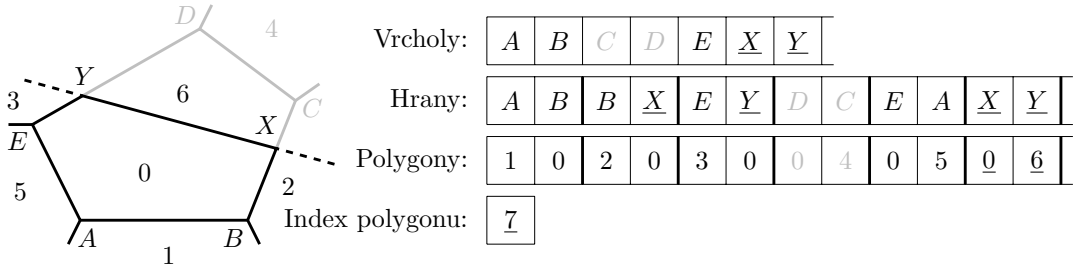
Polygony se rozřezávají podobně jako trojúhelníky. Kontrolují se zde ale jednotlivé hrany. Pro každou hranu jsou nejdříve zkontrolovány pozice obou jejích vrcholů a dále můžou nastat tři různé situace:

- Oba vrcholy jsou odřezány.
- Jeden vrchol je odřezán.
- Ani jeden vrchol není odřezán.

V případech, kdy je celá hrana buď odřezána, nebo naopak není odřezán ani jeden z jejích bodů, už není potřeba provádět další testování. Stačí takovou hranu celou odstranit ze seznamu, nebo ji nijak dále neměnit. Pokud ale je odstraněn pouze jeden z vrcholů, je třeba zjistit přesně v jakém místě je hrana přerušena a vytvořit zde nový bod. Tento bod v záznamu o hraně jednoduše nahradí bod odstraněný. Dále pro každý okrajovaný mnohoúhelník můžou nastat tři možné případy:

- Všechny hrany mnohoúhelníku jsou odstraněny.
- Mnohoúhelník je rozkrojen.
- Ani jedna hrana mnohoúhelníku není změněna.

Opět, obdobně jako v předešlých případech, pokud nebyl polygon změněn nebo byl odstraněn, není třeba dohledávat další informace. Pokud byl ale mnohoúhelník rozříznut, pak platí, že ho okrajovací rovina protíná v právě dvou místech a tím pádem vzniknou dva nové body pro tento polygon. Na obrázku 5.8 je možné vidět rozkrojený polygon s již zalepenou dírou.



Obrázek 5.8: Ukázka ořezávání mnohoúhelníku:
Podtržené položky reprezentují data, které byly nově vytvořeny, nebo upraveny. Šedou barvou jsou zde označena odstraňovaná data. Index polygonu byl inkdementován oproti obrázku 5.7, jelikož nově vznikl polygon s indexem 6.

Zalepování okrajovaných děr

Tato část postupu je u polygonální reprezentace na rozdíl od té trojúhelníkové velmi jednoduše řešitelná. Jak již bylo řečeno pro každý rozkrojený mnohoúhelník vzniknou dva nové body. Tyto body vytvoří novou hranu. Ta je pak jednoduše přidána do seznamu hran. Tato hrana odděluje dva polygony, nově vznikající mnohoúhelník a rozřezávaný mnohoúhelník. Toto lze pozorovat na obrázku 5.8. Tyto hrany jsou reprezentovány čísly 0 a 6.

Převod na trojúhelníkovou reprezentaci

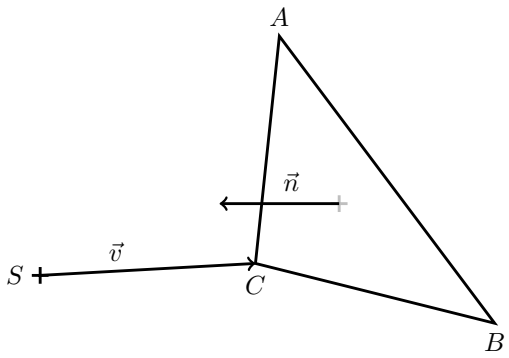
Jak již bylo zmíněno, převod na trojúhelníky je vyžadován pro vykreslení na GPU. Postup při převádění je velmi podobný zalepování děr u trojúhelníkové reprezentace. Každá hrana mnohostěnu obsahuje dva body a pro vytvoření trojúhelníku stačí přidat už jen jeden bod z této stěny. Proto je možné převod řešit tak, že při nalezení první hrany určitého polygonu se jeden z vrcholů zapamatuje. Další nalezené hrany tohoto mnohostěnu už budou po přidání tohoto bodu k nim vytvářet trojúhelníky.

Generování normál

Pokud není možné zajistit správnou orientaci trojúhelníků. Což u převodu z polygonální reprezentace nemusí být úplně možné. Je třeba umět detekovat a napravit špatně vygenerované normály. Správně vygenerovaná normála by měla směřovat ven z objektu. Naopak špatně vygenerovaná normála bude směřovat dovnitř objektu. Tento problém lze jednoduše řešit pro konvexní objekty. V rámci každé buňky je totiž známa pozice jejího jádra. Na obrázku 5.9 je možné vidět příklad špatně vygenerované normály $\vec{n}$. Pro tuto buňku je možné vypočíst vektor $\vec{v}$. Ten směřuje od jádra buňky k bodu B. Tento bod se podílel na původním výpočtu normály $\vec{n}$ podle postupu v kapitole 2.4.1. Nyní je možné zkontrolovat, zda je tato normála ve správné orientaci pomocí vzorce 5.1

$$p = \vec{n} \cdot \vec{v} \tag{5.1}$$

Pokud p vyjde jako kladné číslo, znamená to, že normála má správnou orientaci. Špatná orientace normály je pak značena zápornou hodnotou p. Oprava orientace je možná jednoduchým obrácením znamének v celém vektoru na $-\vec{n}$.



Obrázek 5.9: Ukázka nevhodné orientace normály v buňce:
Bod S je jádro dané buňky. Je tedy vidět, že normála směřuje dovnitř buňky a tím pádem je její orientace nevalidní.

Kapitola 6

Implementace

V této kapitole jsou popsány jednotlivé implementační detaily použitých řešení. Nejdříve v sekci 6.1 jsou popsány jednotlivé nástroje a knihovny, které byly v rámci tohoto projektu použity. V následující sekci 6.2 je uvedena struktura výsledného testovacího programu a jsou zde popsány jeho jednotlivé části. V další sekci 6.3 jsou detailněji popsané jednotlivé implementace algoritmů vytvářející Voroného diagram a způsob jakým je u nich řešena paralelizace. Nakonec jsou v poslední sekci 6.4 popsány některé detaily přístupu k vykreslování výsledné scény. V příloze A je pak možné vidět výsledný grafický výstup aplikace.

6.1 Použité nástroje a knihovny

V rámci této práce byly použity různé nástroje a knihovny. Jejich role v rámci projektu je dále popsána. Práce je napsána v jazyce C++ 11. Vytváření souborů potřebných pro překlad na různých platformách zajišťuje nástroj CMake. V rámci projektu je použita implementace algoritmu vytváření Delaunayho triangulace na GPU gDel3D. Ta využívá technologii CUDA, což je API pro výpočty na grafických kartách firmy NVidia. Dále je pro ostatní implementace algoritmů na GPU použito grafické rozhraní pro vykreslování a výpočty na GPU OpenGL. Toto API je již možné použít i na grafických kartách jiných výrobců. Pro zpřístupnění tohoto rozhraní jsou použity knihovny GLEW a SDL2. Dále je použita knihovna GLM ke zpřístupnění datových typů jazyka GLSL a funkcí pro práci s nimi. Tento jazyk je používán pro definici shaderů v OpenGL. Knihovna SDL2 je také použita pro práci s okny v rámci operačního systému a pro zachytávání vstupů z klávesnice a myši. Všechny výše zmíněné knihovny umožňují zprovoznění projektu na více platformách. Jmenovitě se jedná hlavně o systémy Linux a Windows.

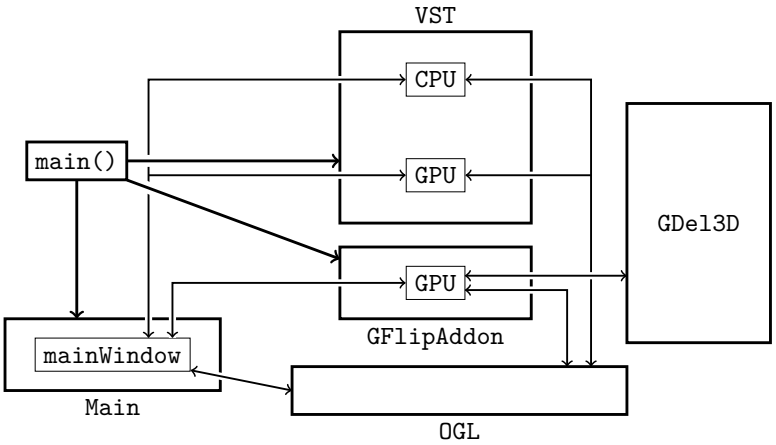
Nejdůležitější části OpenGL byly v rámci práce obaleny miniaturním frameworkem. Ten umožňuje objektový přístup jazyka C++ k objektům v rámci OpenGL a byl během práce přizpůsobován podle potřeb. Tento framework také zjednodušuje práci s okny knihovny SDL2 a inicializace prostředí OpenGL v rámci tohoto okna.

6.2 Struktura programu

Program je rozdělen do několika částí, modulů. Každá z částí je odlišena od ostatních pomocí vlastního jmenného prostoru (anglicky „namespace“). Jejich názvy jsou `Main`, `OGL`, `VST` a `GFlipAddon`.

- `Main` - Hlavní část programu, která inicializuje ostatní části a spouští jednotlivé algoritmy
- `OGL` - Modul zajišťující zjednodušení práce s okny a OpenGL
- `VST` - Modul s implementacemi metody VST
- `GFlipAddon` - Modul propojující implementaci `gDel3D` se zbytkem programu a řešící převod DT na VD

Jelikož je výsledný program určen pouze pro otestování algoritmů popsaných v této práci, je jeho strukturu velmi jednoduchá. Interakce mezi jednotlivými částmi programu jsou ilustrovány na obrázku 6.1.



Obrázek 6.1: Ilustrace struktury programu:
Jednotlivé moduly (jmenné prostory) jsou reprezentovány obdélníky s jejich jmény buď pod nebo nad sebou. Uvnitř jmenných prostorů jsou dále vyznačené jejich důležité části. Zde `CPU` a `GPU` značí jednotlivé implementace daných algoritmů.

`OGL`

OpenGL ve své základní formě nevyužívá objekty. Způsob, jak se s ním pracuje, je ale tomu objektovému velmi podobný. Hlavní úlohou tohoto modulu je vytvořit vrstvu nad OpenGL, která umožňuje objektový přístup k některým jeho částem. Dále také zaobaluje vytváření oken pomocí knihovny SDL2. A obecně se snaží zjednodušit používání často potřebné funkcionality OpenGL.

```
1 GLuint buffer0;
2 glGenBuffers(1, &buffer0);
3 glBindBuffer(GL_ARRAY_BUFFER, buffer0);
4 glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
```

Výpis 6.1: Ukázka kódu pro OpenGL

Na výpisu 6.1 a 6.2 je možné vidět rozdíly mezi zápisy ekvivalentních úkonů pro OpenGL a nadstavbu OGL. U OGL je dále ukázána možnost dalšího zkrácení zápisu pro často

používané konstrukce. Z tohoto porovnání je jasné, že převod OpenGL na objektový model může být velmi žádoucí.

```
1 OGL::Buffer buffer1(GL_STATIC_DRAW);
2 buffer1.bind(GL_ARRAY_BUFFER);
3 buffer1.createImutable(sizeof(data), data);
4
5 //dalsi moznost zapisu podobneho ukonu
6 OGL::Buffer buffer2(GL_STATIC_DRAW, GL_ARRAY_BUFFER, sizeof(data), data);
```

Výpis 6.2: Ukázka kódu s OGL

`VST`

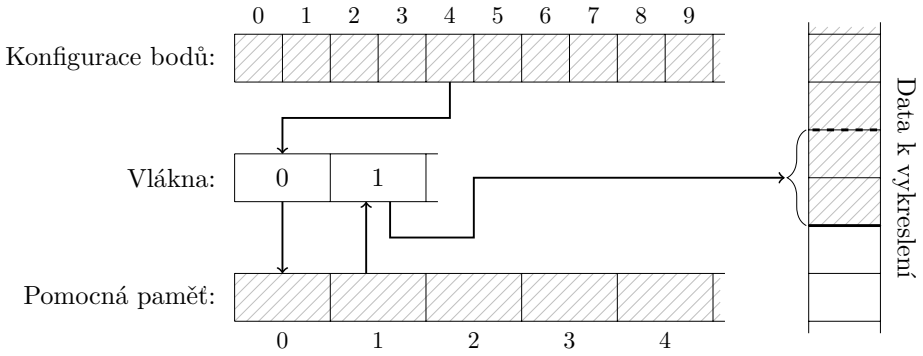
Modul VST obsahuje 2 implementace metody stejného jména. Jedna z nich je pouze kontrolní implementace na procesoru. druhá je pro grafické karty. Tato implementace využívá pouze jednoho volání grafické karty. Zde dochází k hledání nejbližšího bodu vždy před každým krájením.

`GFlipAddon`

Modul GFlipAddon je do výsledného programu zakomponován pouze pokud je přítomen kód implementace gDel3D a v nástroji CMake je zapnut přepínač gFlip3D. Potom je i tento modul přeložen spolu cizím kódem zmiňovaného algoritmu. Potom tento modul volá externí kód implementace vytváření Delaunayho triangulace. Dále je v rámci tohoto modulu přidán převod výsledné DT na VD pomocí GPU.

6.3 Detaily paralelizovaných algoritmů

Tato část textu se zaměřuje na jednotlivé implementační detaily algoritmů vytváření VD. Hlavně se zde popisují detaily týkající se použití GPU. Mezi nejdůležitější detaily patří způsob použití paměti v jednotlivých implementacích, způsob rozvržení práce mezi jednotlivá vlákna a synchronizace mezi nimi.



Obrázek 6.2: Ilustrace práce vláken s pamětí:
Na obrázku je zobrazeno zobecněné schéma vláken vytvářejících buňky na základě dat podle konfigurace bodů. Toto schéma je relevantní jak pro metodu VST, tak pro převod z DT na VD. Šrafování vyznačuje data uložená do paměti. Na obrázku je vyznačeno vlákno 0 při výběru bodu a ořezávání buňky. Vlákno 1 zase zapisuje výslednou buňky na finální místo v paměti. Odtud můžou být data použita k vykreslení.

6.3.1 Algoritmus využívající Delaunayho triangulace

Jak již bylo zmíněno v rámci tohoto algoritmu je využita implementace vytváření DT na GPU, gDel3D. Jelikož tato implementace využívá API CUDA od firmy NVidia, je vyžadována pro zprovoznění této části programu grafická karta od zmiňované společnosti. Implementace tohoto přístupu je tedy podmíněna z velké části vlastnostmi externí implementace algoritmu gDel3D. Ta na svém výstupu generuje seznam čtyřstěnů. Ten je v paměti uložen jako pole čtveřic. Každá položka čtveřice je index do pole všech bodů přítomných v triangulaci.

Takže v rámci tohoto algoritmu se nejdříve zavolá implementace gDel3D, která vygeneruje pole čtyřstěnů. Toto pole se přemístí do paměti na GPU v rámci OpenGL. Dále se spustí převod DT na VD. Ten je koncipován tak, že se spustí určitý počet vláken rozdělených do několika pracovních skupin tak, aby se skupiny mohli rozdělit na všechny multiprocesory GPU. Každé vlákno pak má přidělenou vlastní pomocnou paměť. Aby nebyly problémy s nedostatkem místa, je tento prostor vymezen v globální paměti. Tato pomocná paměť je v rámci této implementace použita pouze pro ukládání geometrie nedokončené buňky a k poznamenání již použitých okolních bodů. Každé vlákno dostane přiděleno jedno jádro buňky. Každé vlákno pracuje samostatně na vytváření jedné buňky, projde triangulaci a vyhledá všechny sousedy, které pak použije k ořezávání. Práci vláken lze vidět na obrázku 6.2.

6.3.2 Metoda VST

Pro implementaci v rámci této práce byl vybrán jeden přístup. Ten využívá pouze jeden shader, který každou iteraci algoritmu nejdřív vyhledá nejbližší bod a poté ho použije k ořezání buňky. Nedochází zde k řazení buněk, místo toho se před prvním hledáním nejbližšího bodu vytvoří pro každé vlákno pole se vzdálenostmi jednotlivých bodů od příslušného jádra. Při každé iteraci se pak hledá nejbližší bod vzdálenější než ten předešlí. Jedno vlákno zde zpracovává jednu buňku. Dále by ale šlo tuto implementaci vylepšit použitím většího množství vláken pro části shaderu, kde je hledán nejbližší bod. Obrázek 6.2 lze zase použít

k reprezentaci práce vláken spouštěných pro tento přístup. Tentokráte ale musí při každém výběru bodu pro okrajování každé vlákno projít celé své pole se vzdálenostmi bodů. Okrajování pak probíhá stejně jako u převodu z DT.

6.4 Vykreslování

Jak již bylo v kapitole 4 popsáno, v moderním vykreslování na GPU je tendence minimalizovat režii procesoru. Toho je v této práci docíleno pomocí použití techniky nepřímého vykreslování. Další zajímavě řešeným problémem v rámci tohoto projektu je výpočet normál. Ty jsou vypočítávány až během vykreslování. Osvětlení je řešeno velmi jednoduše. Je použit Phongův osvětlovací model [11]. Ten není pro práci zásadní a nebude blíže vysvětlen.

Generování dat a jejich vykreslení

V rámci výpočetních shaderů se vytváří nejen geometrie Voroného diagramu, ale také všechna další potřebná data pro vykreslování. Během algoritmů se postupně do pomocné paměti vytváří vrcholy a indexy specifikující trojúhelníky reprezentující vzniklou buňku Voroného diagramu. Následně si vlákno zpracovávající daný střed buňky zabere potřebné místo v paměti pro zapsání všech dat spojených s nově vytvořenou buňkou. To je zajištěno atomickým přičtením počtu vzniklých indexů a vrcholů k odkazu na konec dat v paměti určené pro vykreslovaná data. V rámci tohoto zápisu jsou vygenerovány argumenty pro nepřímé vykreslování.

Pak již stačí při každé aktualizaci obrazu předat GPU informaci o pozici kamery, pozici úlomků a spustit nepřímé vykreslování jedním voláním. Výsledkem je vykreslení všech úlomků najednou na správných místech na obrazovce.

Výpočet normál

Pro výpočet jednoduchého osvětlovacího modelu scény je potřeba znát normály jednotlivých trojúhelníků v této scéně. Jelikož scéna není příliš složitá, je možné normály vypočítat až během vykreslování. Tento přístup zmenšuje paměťovou náročnost programu na GPU za cenu zvýšení náročnosti výpočetní. Jak již bylo ale řečeno, vykreslení této scény je triviální, takže tento výpočet nebude mít příliš negativní dopad na výkon při vykreslování. Samotný výpočet zajišťuje geometrický shader, jelikož pouze během této fáze vykreslovacího řetězce je možné přistupovat ke všem bodům trojúhelníku společně. Postup výpočtu závisí na orientaci bodů v trojúhelníku a je popsán v kapitolách 5.4.1 a 5.4.2.

Kapitola 7

Měření

Tato kapitola se zaměřuje na testování výsledného programu. V sekci 7.1 je popsán způsob měření výkonu výsledného testovacího programu. Je zde také popsán hardware, který je použit pro testování. V následující sekci 7.2 jsou pak uvedeny jednotlivé testy, jejich výsledky a komentáře k těmto dosaženým hodnotám.

7.1 Popis metody měření

Testovací program využívá multiplatformní nástroje a knihovny. Je tedy možné zprovoznit program na různých systémech. V rámci práce byl kód testován na systémech Linux i Windows. Jelikož implementace gDel3D využívá API CUDA, je omezena pouze na grafické karty firmy NVIDIA. Dále tato implementace algoritmu nepodařila zprovoznit na systému Windows. Všechny testy, kde figuruje gDel3D budou prováděny pouze na systému Linux. Dále jelikož implementace metody VST jsou vytvořeny pomocí OpenGL, je možné tyto implementace testovat i na GPU od firmy AMD.

Většina testů se zaměřuje na rychlost vytvoření Voroného diagramu. Je tedy vhodné uvést přesný způsob měření času. Existuje více způsobů jak na počítači čas měřit. Hlavní dva přístupy k měření jsou následující:

- Měření procesorového času
- Měření reálného času

Procesorový čas je doba, po kterou program zaměstnává CPU případně GPU. Jelikož se testuje výkon programu vytvářeného z velké části pro GPU, měření času využitého na CPU by nemělo smysl. Další potencionální problém měření procesorového času je měření, respektive neměření latencí. Může se stát, že program nebude využívat celý výpočetní potenciál, protože bude čekat například na načítání dat z paměti. Potom může být výpočetní jednotka využita k jiným účelům a tento čas nebude započítán. Proto bude použit k měření výkonu reálný čas. Pro zajištění co nejmenšího zkreslení ostatními procesy systému, jsou nepotřebné aplikace vypnuty. Čas je vždy měřen od spuštění výpočtu na GPU až po jeho konec. Do času není zahrnuto nahrávání množiny bodů do paměti GPU.

Testovací systémy

Během měření byly dostupné dva testovací systémy. Jeden z nich je notebook od firmy Acer zakoupený roku 2014, jeho specifikace jsou uvedeny v tabulce 7.1 a dále bude označován

jako systém 1. Druhý systém je stolní počítač sestavený v roce 2016, jeho specifikace jsou uvedeny v tabulce 7.2 a dále bude označován jako systém 2.

1: Notebook	
CPU	Intel Core i3-4000M 2.40GHz
GPU	NVIDIA GeForce GT 840M 1GB
RAM	8GB
Windows	8.1 64bit
GPU Driver	441.22
Linux	Mint 18 64bit
GPU Driver	384.130
CUDA Toolkit	7.5

Tabulka 7.1: Specifikace testovacího systému 1: Notebook

2: Stolní počítač	
CPU	Intel Core i5-6500 3.20GHz
GPU	NVIDIA GeForce GTX 1060 6GB
RAM	16GB
Windows	8.1 Pro 64bit
GPU Driver	451.67
Linux	Debian 9 64bit
GPU Driver	418.74
CUDA Toolkit	9.2

Tabulka 7.2: Specifikace testovacího systému 2: Stolní počítač

7.2 Výsledky

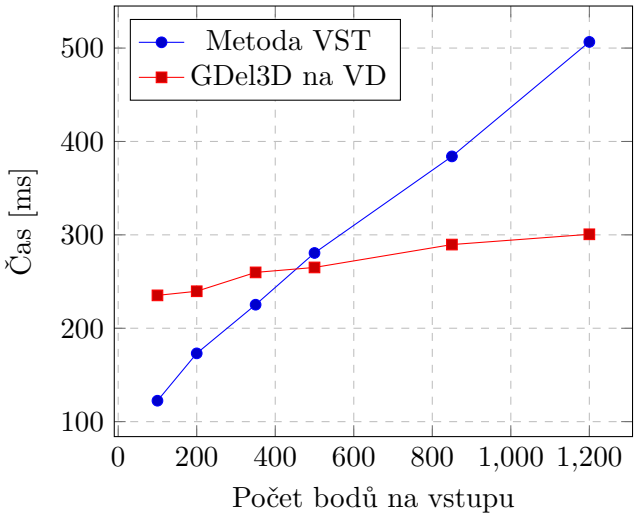
Program byl testován na obou systémech. Na obou se ho podařilo přeložit i spustit, jak na operačním systému Windows, tak na Linuxu. Výjimkou byl ale algoritmus GDel3D, který se podařilo zprovoznit pouze na Linuxu. Pokud není dále uvedeno jinak, jsou výsledky měření prezentované dále v této sekci získány na systému 2 za použití operačního systému Linux.

Implementace metody VST s využitím dvou shaderů nebyla v čase testování dovedena do stavu vhodného k měření. Proto bude k měření použita pouze implementace s jedním shaderem.

Výkon obou implementací pro různé počty bodů

Zde je měřen výkon implementace algoritmu využívající převod z DT a metody VST. Přístup využívající GDel3D používá při těchto měřeních 16 pracovních skupin po 32 vláknech. V rámci měření byla použita implementace metody VST používající jeden shader, zde se využívá 8 pracovních skupin o 32 vláknech. V posledním měření pro 100 bodů je počet

pracovních skupin pak změněn na 4. Konfigurace bodů a čtyřstěnu je pro obě metody také trochu pozměněna. Jelikož GDel3D netrpí problémem nevhodného obalového tělesa, neměl by ho tento fakt nijak znevýhodnit. V rámci VST je použit čtyřstěn s vrcholy $[0, 0, 0]$, $[1, 0, 0]$, $[0, 1, 0]$ a $[0, 0, 1]$. Body jsou pak rozprostřeny po celém jeho objemu. Druhý algoritmus místo toho používá třikrát větší čtyřstěn a body v něm jsou umístěny v osově zarovnané krychli definované body $[0, 0, 0]$ a $[1, 1, 1]$. Výsledky měření je možné vidět na obrázku 7.1.

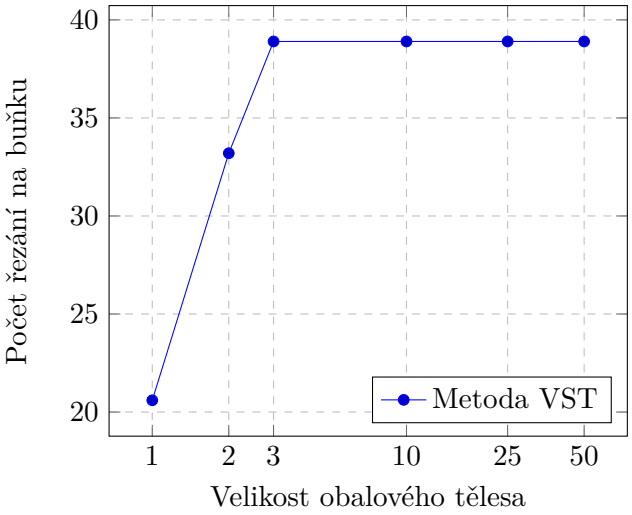


Obrázek 7.1: Graf výsledků měření rychlosti metody VST a algoritmu využívajícího DT k vytvoření VD: Na grafu lze vidět rozdílnou citlivost jednotlivých implementací vůči množství vstupních bodů. Je zde možné vidět pomalejší start algoritmu GDel3D a vyšší úměrnost zpomalení metody VST vůči počtu bodů.

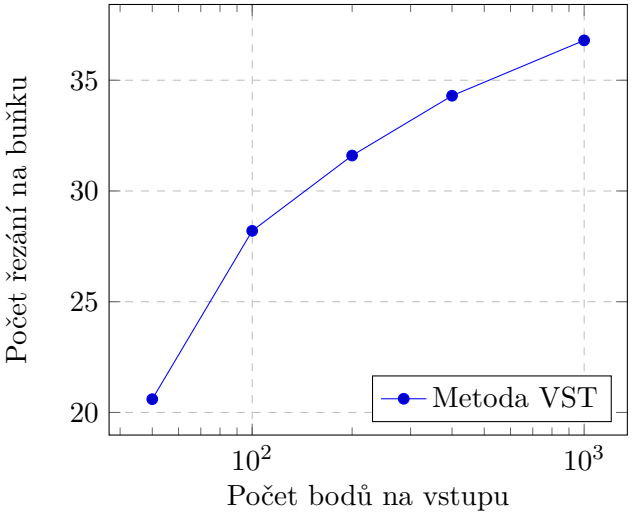
Během těchto měření se při některých vytváření VD objevily zvláštní odchylky u obou algoritmů. Častěji se objevovali u algoritmu převodu z DT. Tyto odchylky nebyly započítány do tohoto měření. Během případů, kdy se vyskytly tyto odchylky, nebyl detekován žádný zvláštní nárůst počtu ořezávání ani vzniklých vrcholů nebo hran.

Dopad nevhodného obalového tělesa na počet řezání metodou VST

Zde je provedeno měření počtu řezání provedených v rámci metody VST pro různé konfigurace bodů a obalového čtyřstěnu. Nejdříve je ukázán efekt nevhodně zvoleného obalového tělesa na obrázku 7.2, zde je použita konfigurace bodů do osově zarovnané krychle definované body $[0, 0, 0]$ a $[1, 1, 1]$. Pokud by ale měl některý z takto generovaných bodů skončit mimo těleso je pro něj hledána jiná pozice. Na druhém obrázku 7.3 je zase ukázáno, jaký efekt má vkládání většího množství bodů do vhodně zvoleného tělesa. Body jsou zde rozmístěny po celém obalovém tělese, tedy čtyřstěnu. Nejlevější uzly vyobrazené na obou těchto grafech reprezentují jednu a tu samou situaci se stejnou konfigurací bodů i velikostí obalového tělesa.



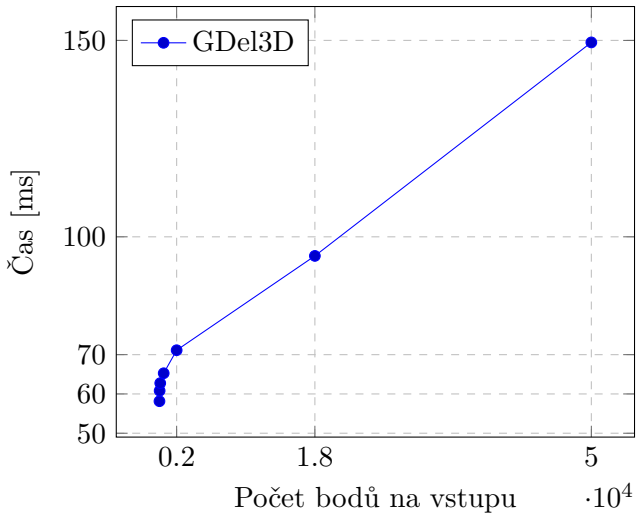
Obrázek 7.2: Graf výsledků měření počtu řezání potřebných pro vytvoření jedné buňky pro 50 jader v závislosti na velikosti tělesa: Velikost tělesa značí délku tří na sebe kolmých hran čtyřstěnu. Na grafu je možné vidět, že existuje hranice, za kterou už příliš velká velikost tělesa nemá vliv na výkon metody VST. Další zajímavou informací je fakt, že pro případy, kdy vliv nevhodného obalového tělesa dosáhl maxima, došlo k použití všech okolních bodů pro ořezání buňky v 25 případech. Tzn. polovina buněk byla prakticky jakoby vytvářena naivním ořezáním.



Obrázek 7.3: Graf výsledků měření počtu řezání potřebných pro vytvoření jedné buňky pro ideálně velké těleso s různým počtem buněk: Zde je možné vidět pozitivní účinky vhodného obalového tělesa, které je hustě zaplněno body. Pro 50 bodů dojde průměrně k 20 krájením. Pro 1000 bodů se sice počet krájení blíží 40, ale poměrově k počtu vstupních bodů je tato metoda s přibývajícími body více efektivní.

Rychlost vytvoření DT pomocí gDel3D

V této sekci je měřen výkon externí implementace GDel3D. Pro různé počty bodů je měřen čas, který trvá vytvoření Delaunayho triangulace. Je měřen reálný čas od volání funkce, která vytváří DT až po vystoupení z této funkce. Výsledky těchto měření byly vykresleny na obrázku 7.4.

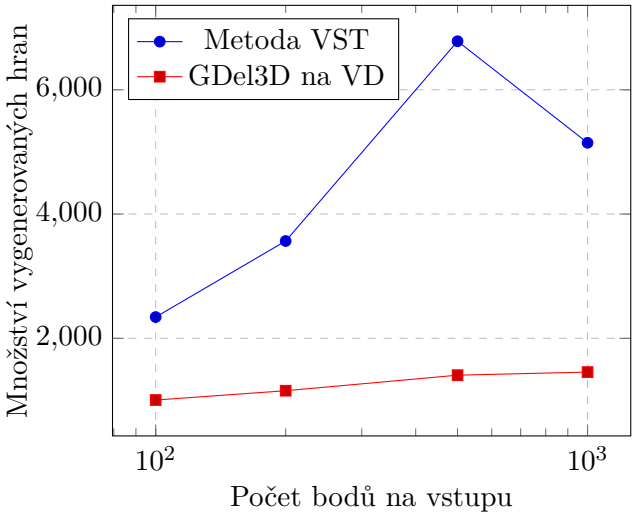


Obrázek 7.4: Graf výsledků měření rychlosti algoritmu GDel3D při vytváření DT:

Zde je vidět relativně lineární růst doby výpočtu. Časový nárůst pro prvních zhruba 1000 bodů je strmější. Což potvrzuje výbornou optimalizovanost tohoto algoritmu pro větší počty bodů. Dále to také utvrzuje ideu, že postupné vkládání bodů algoritmu GDel3D relativně zpomaluje nástup jeho výkonu a že metoda VST může fungovat lépe pro menší množství bodů.

Paměťová náročnost

V rámci testovaných implementací byla testována i paměťová náročnost. Jelikož oba algoritmy nakonec dojdou stejnému výsledku nemá cenu porovnávat spotřebu paměti přímo pro Voroného diagram. V rámci implementace je generování nových hran řešeno ukládáním do pomocné paměti, která má vždy předem určenou velikost. Při vývoji se paměť vyhrazená hranám prokázala jako nejnáročnější. Pro správnou funkci programu musí být předem vyhrazeno dostatečné množství paměti pro každé vlákno na ukládání hran. Je zde použita stejná konfigurace bodů a obalových těles jako při prvním testu, tedy relativně ideální rozložení bodů pro metodu VST a mírně větší obalové těleso pro převod z DT na VD. Počet pracovních skupin by na tento test neměl mít žádný vliv. I když pro celkovou velikost potřebné pomocné paměti je vliv množství vláken velký. Výsledek tohoto měření je potom možné vidět na obrázku 7.5.



Obrázek 7.5: Graf reprezentující potenciální paměťovou náročnost generování hran při ořezávání pro jednotlivé algoritmy: U metody VST je možné vidět náchylnost k relativně velkým výkyvům v závislosti na rozložení jader buněk. Větší množství ořezávání generuje větší množství hran. Proto je zde metoda VST v nevýhodě oproti převodu z DT, kdy je možné vybírat ořezové roviny naprosto ideálně a tím pádem minimalizovat množství vznikajících hran.

Kapitola 8

Závěr

V rámci této práce byly popsány a prozkoumány dva přístupy vytváření Voroného diagramu. Byly vytvořeny dvě hlavní implementace pro testování, jedna pro metodu VST a jedna pro algoritmus využívající externí implementaci GDel3D pro vytvoření Delaunayho triangulace a následné převedení na Voroného diagram. V rámci práce také proběhly pokusy o implementaci dalších variant algoritmů. Z časových důvodů, ale zůstaly tyto pokusy u návrhu. Podařilo se objevit několik specifických vlastností metody VST, které tuto metodu znevýhodňují pro určitá použití. Mezi tyto vlastnosti patří například náchylnost na nevhodně zvolený okrajovaný objekt, nebo naopak nevhodné rozložení bodů v něm. Jinými slovy, hlavním přínosem této práce bylo hlubší prozkoumání vlastností metody VST a obzvláště nastínění potenciálního problému s rozložením jader v tělese. Ten je také v rámci testování demonstrován na obrázcích 7.2 a 7.3.

Zadání práce bylo z velké části splněno. Bohužel implementace algoritmů nejsou dostatečně optimalizovány pro získání absolutních závěrů o výkonnosti algoritmů, nicméně pro relativní porovnání chování těchto dvou metod jsou dostačující a nastiňují jejich výhody a nevýhody. Hlavní výhodou algoritmu GDel3D pro vytváření Delaunayho triangulace je jeho vysoce optimalizované zpracování velkého množství bodů. Na druhou stranu má ale nevýhodu v pomalejším startu algoritmu a tím pádem pro malé počty bodů může být pomalejší než jiné přístupy, což je možné vidět na obrázku 7.4. Dále je algoritmus GDel3D náchylnější na nevhodné rozložení bodů v prostoru, kdy pro některé příliš degenerované konfigurace tato implementace algoritmu odmítá DT vytvořit.

Oproti tomu hlavní výhodou metody VST je její jednoduchost a otevřenost úpravám. Prakticky není možné, aby tato metoda kompletně selhala. Degenerace rozložení bodů ani nepřesnosti ve výpočtech nezpůsobí totální kolaps této metody. Navíc pro tuto metodu při vhodných podmínkách může pro menší množství bodů poskytnout podobný nebo lepší výkon než algoritmus využívající GDel3D.

Jako pokračování této práce by bylo dobré vyzkoušet a blíže otestovat různá vylepšení nastíněná v této práci. Jelikož je metoda VST více otevřená úpravám programátora, je pro tyto experimenty vhodnější a zajímavější než GDel3D. Kdyby bylo možné změnit něco na tom jak se vyvíjel tento projekt, určitě by bylo vhodné zaměřit se na metodu VST a více experimentovat. Například by bylo zajímavé vyzkoušet různá rozdělení práce mezi více vlákny, případně celými shadery, různé způsoby řazení a výběru bodů, nebo případné úpravy kritéria ukončujícího řezání. Asi nejzajímavěji vypadá možnost využití částečného řazení pomocí radix sortu v rámci jednoho shaderu přímo s okrajováním buňky.

Literatura

[1] *Vertex Rendering: Indirect Rendering* [online]. Khronos Group, září 2017 [cit. 2020-07-30]. Dostupné z: https://www.khronos.org/opengl/wiki/Vertex_Rendering#Indirect_rendering.

[2] Berg, M. d., Cheong, O., Kreveld, M. v. a Overmars, M. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN 3540779736.

[3] Chlubna, T. *Simulace křehkých těles*. Vysoké učení technické v Brně. Fakulta informačních technologií, 2018.

[4] Dubska, M., Herout, A. a Havel, J. PClines - Line detection using parallel coordinates. In: *CVPR 2011*. IEEE, 2011, s. 1489–1494. ISBN 9781457703942.

[5] Ericson, C. *The Gilbert-Johnson-Keerthi algorithm* [online]. 2005 [cit. 2020-07-30]. Dostupné z: http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_GJK_notes.pdf.

[6] Fortune, S. A Sweepline Algorithm for Voronoi Diagrams. In: *Proceedings of the Second Annual Symposium on Computational Geometry*. New York, NY, USA: Association for Computing Machinery, 1986, s. 313–322. SCG '86. ISBN 0897911946.

[7] Goldberg, D. *What Every Computer Scientist Should Know About Floating-Point Arithmetic* [online]. 1991. Aktualizováno 5. 4. 2000 [cit. 2020-07-30]. Dostupné z: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[8] Harris, M., Sengupta, S. a Owens, J. D. *Chapter 39. Parallel Prefix Sum (Scan) with CUDA* [online]. NVIDIA Corporation [cit. 2020-07-30]. Dostupné z: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda.

[9] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN 0201896850.

[10] Nanjappa, A. *Delaunay triangulation in R3 on the GPU*. 2012. 177 s. Disertační práce. National University of Singapore.

[11] Phong, B. T. Illumination for Computer Generated Pictures. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. červen 1975, sv. 18, č. 6, s. 311–317. ISSN 0001-0782.

[12] Rebay, S. Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. *J. Comput. Phys.* USA: Academic Press Professional, Inc. Květen 1993, sv. 106, č. 1, s. 125–138. ISSN 0021-9991.

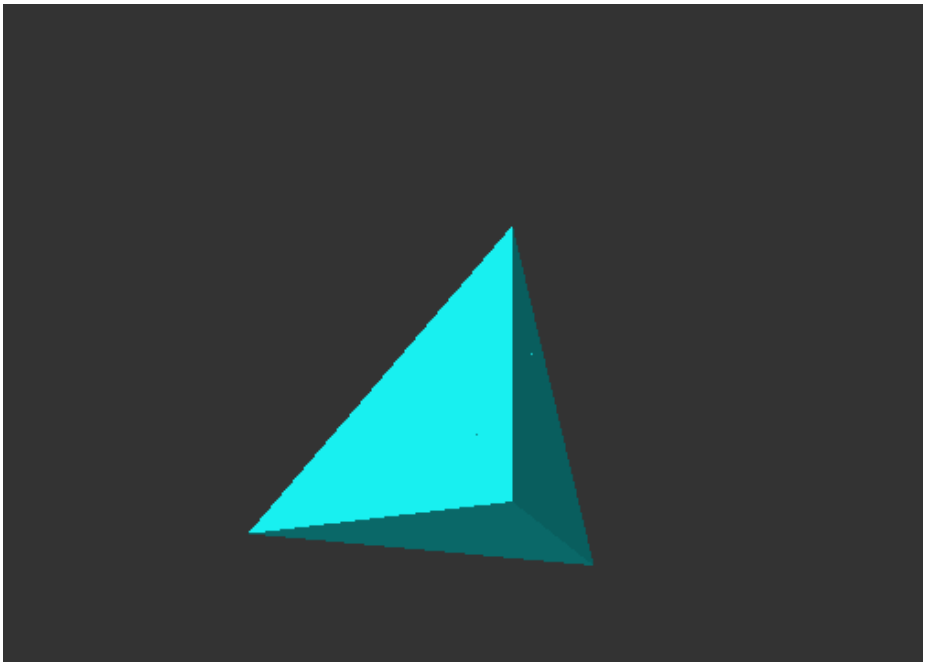
[13] Shewchuk, J. R. Robust Adaptive Floating-Point Geometric Predicates. In: *Proceedings of the Twelfth Annual Symposium on Computational Geometry*. New York, NY, USA: Association for Computing Machinery, 1996, s. 141–150. SCG '96. ISBN 0897918045.

[14] Tatourian, A. *NVIDIA GPU architecture and CUDA programming environment* [online]. Září 2013 [cit. 2020-07-30]. Dostupné z: https://tatourian.blog/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/.

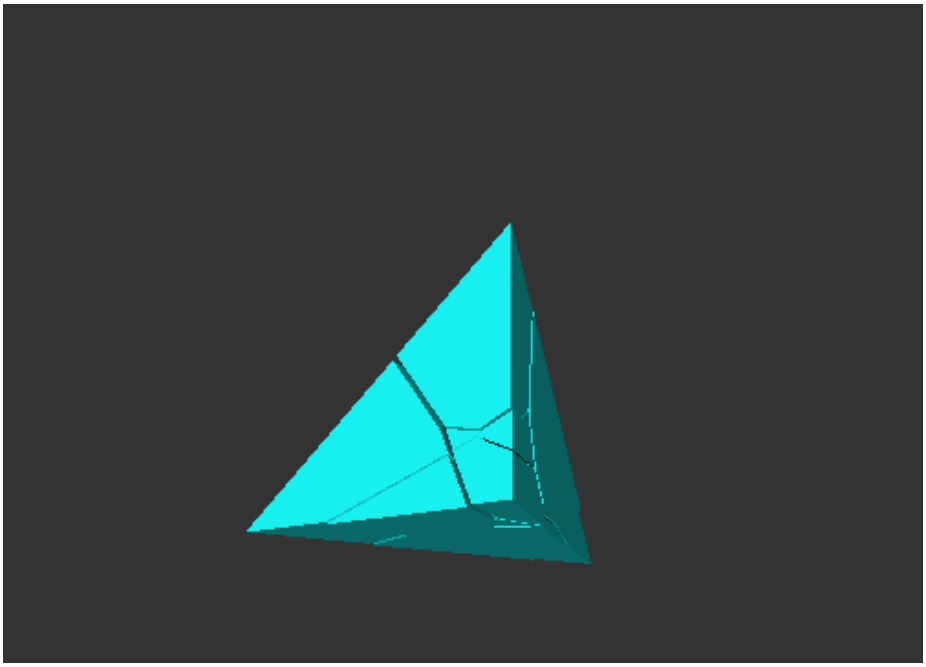
[15] Zagha, M. a Blelloch, G. E. Radix Sort for Vector Multiprocessors. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 1991, s. 712–721. Supercomputing '91. ISBN 0897914597.

Příloha A

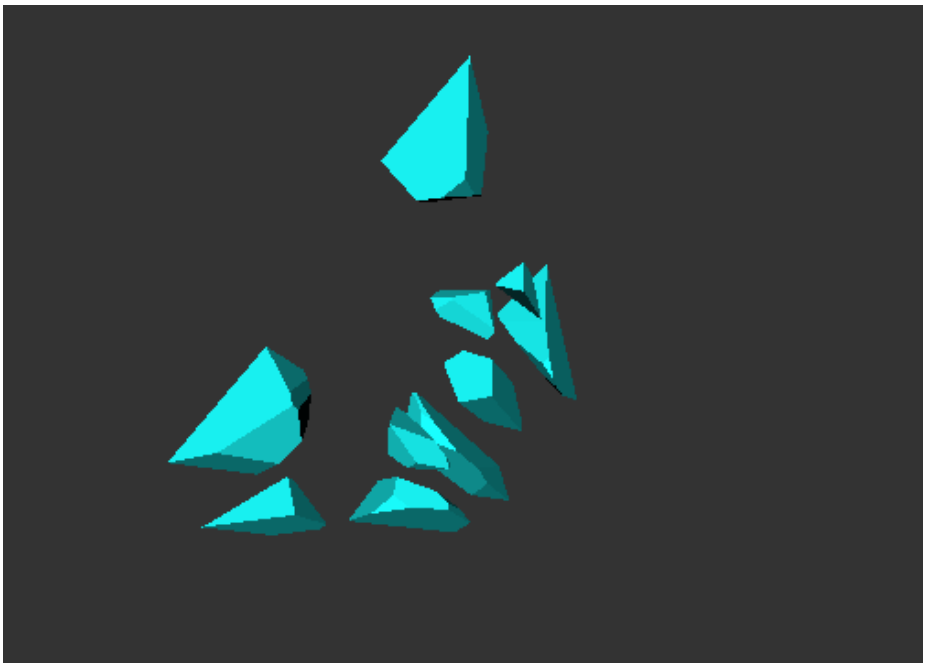
Aplikace



Obrázek A.1: Grafický výstup aplikace: původní objekt



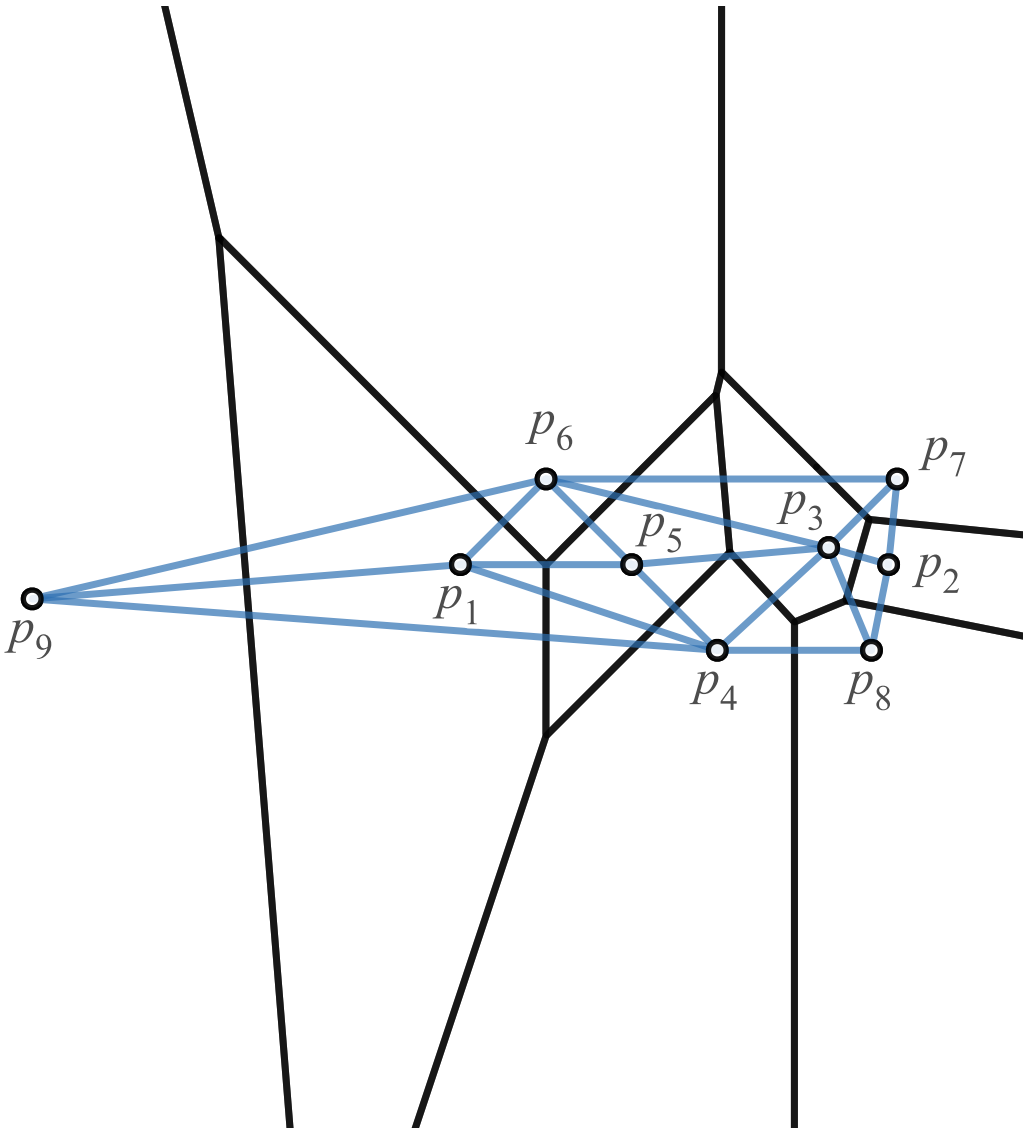
Obrázek A.2: Grafický výstup aplikace: malé oddělení úlomků, vznik prasklin



Obrázek A.3: Grafický výstup aplikace: úplně oddělené úlomky

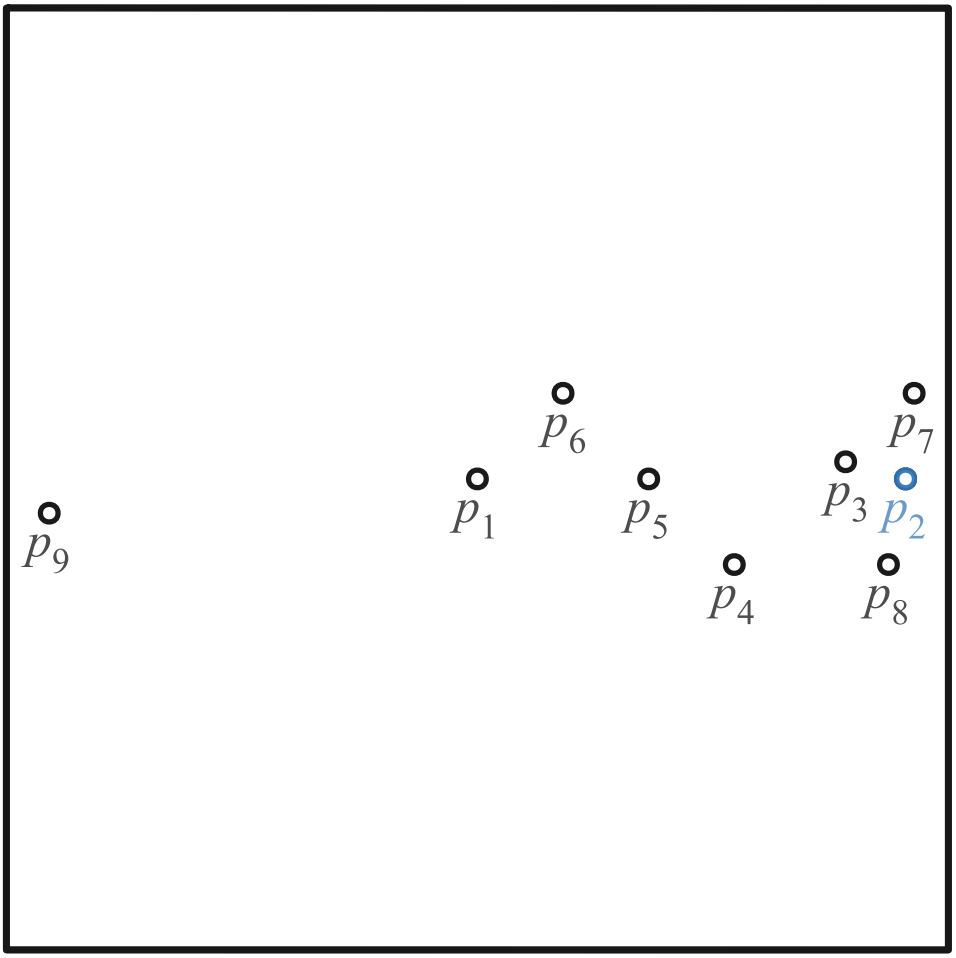
Příloha B

Ilustrace metody VST

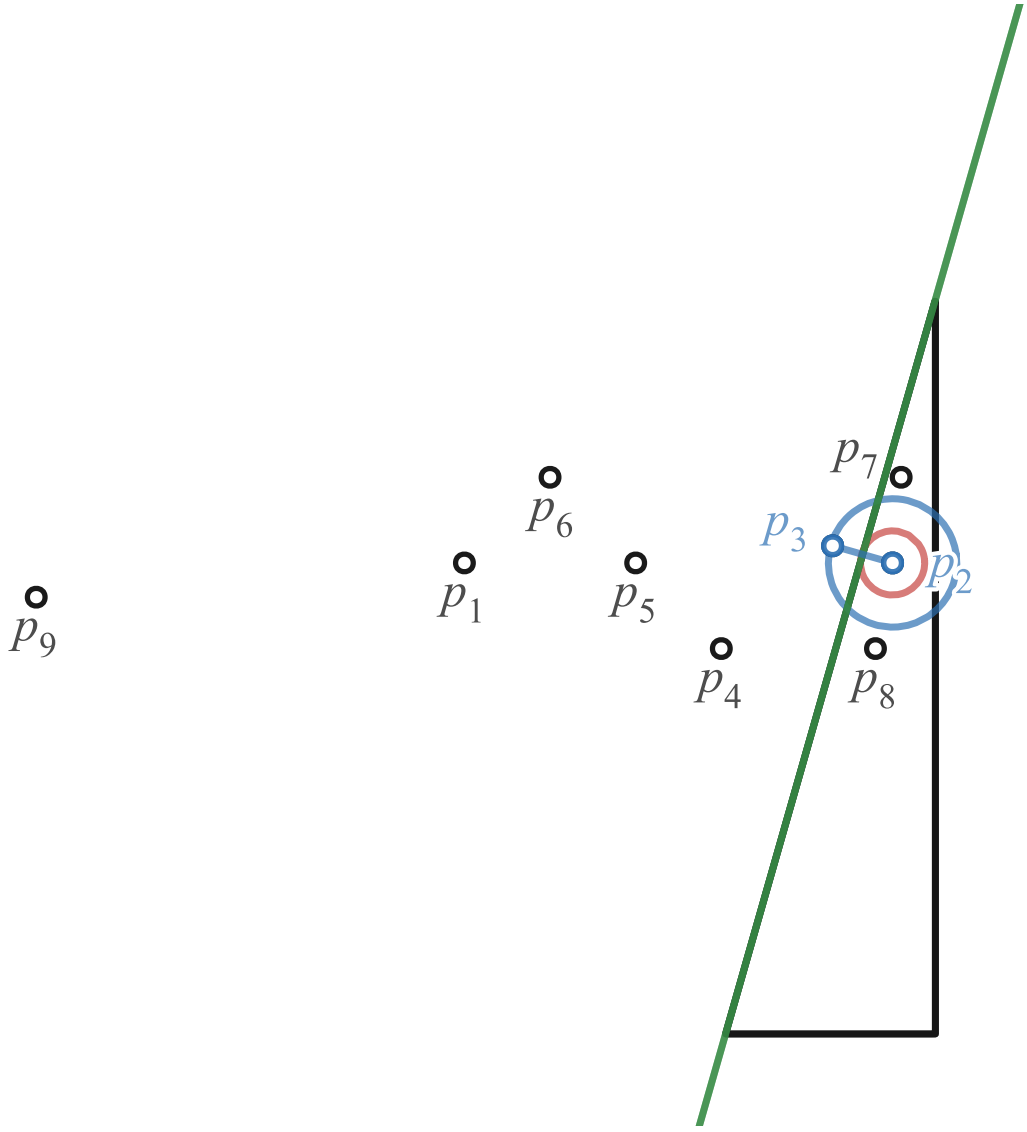


Obrázek B.1: Konfigurace množiny bodů, jejich Voroného diagram a Delauneho triangulace

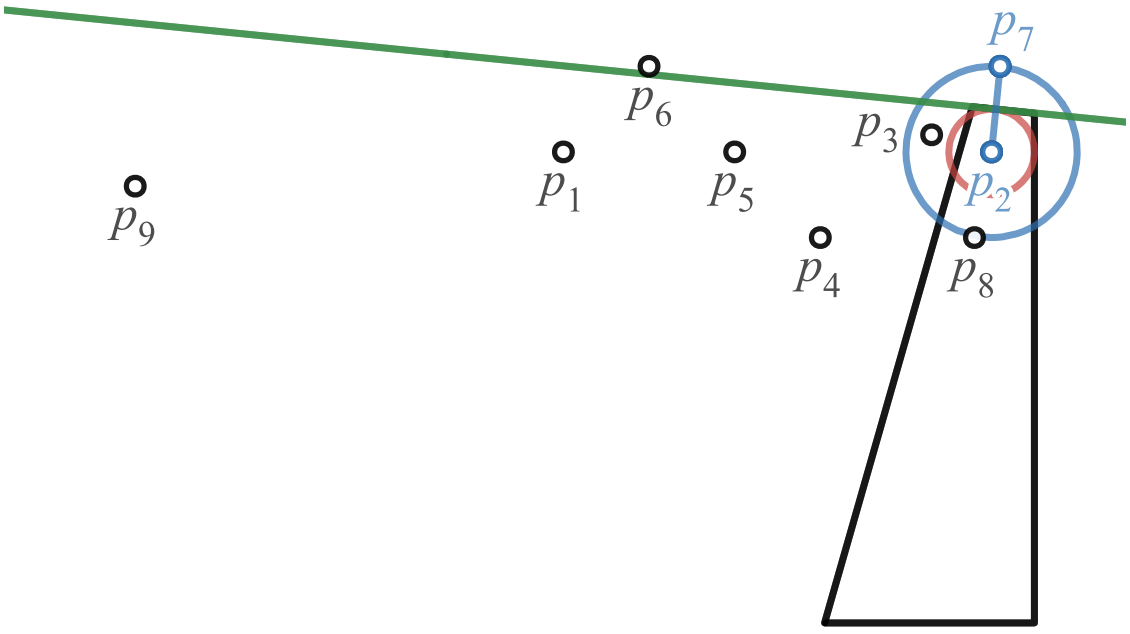
B.1 Nejlepší případ



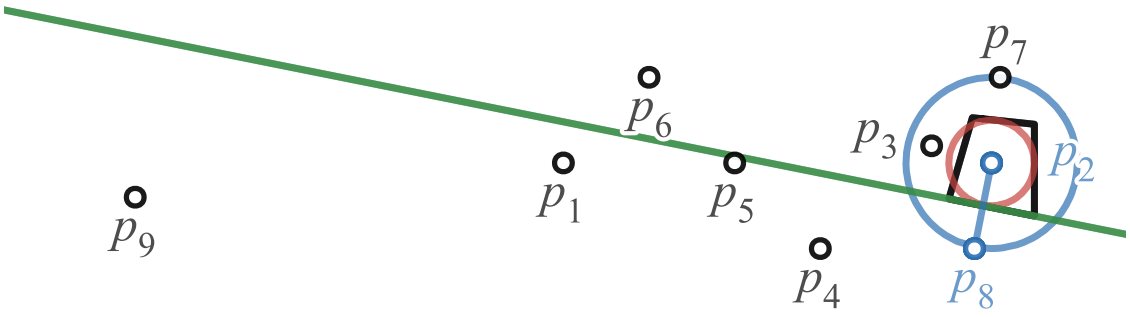
Obrázek B.2: Nejlepší případ aplikace metody VST – krok 0



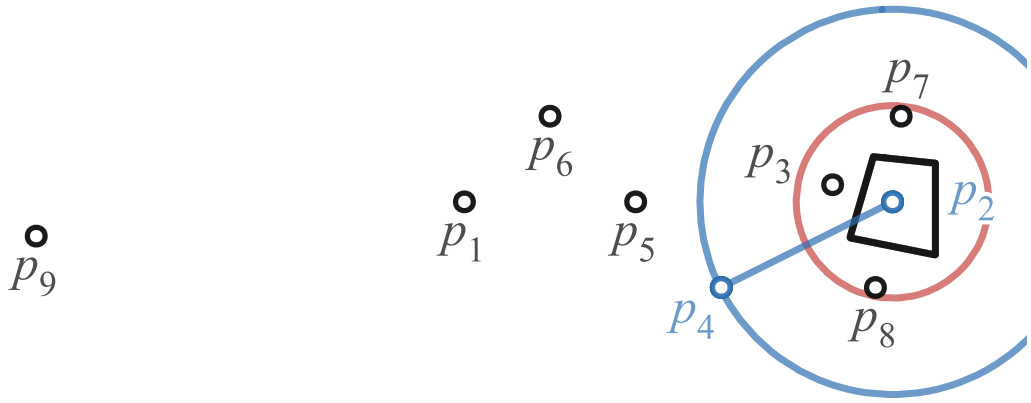
Obrázek B.3: Nejlepší případ aplikace metody VST – krok 1



Obrázek B.4: Nejlepší případ aplikace metody VST – krok 2

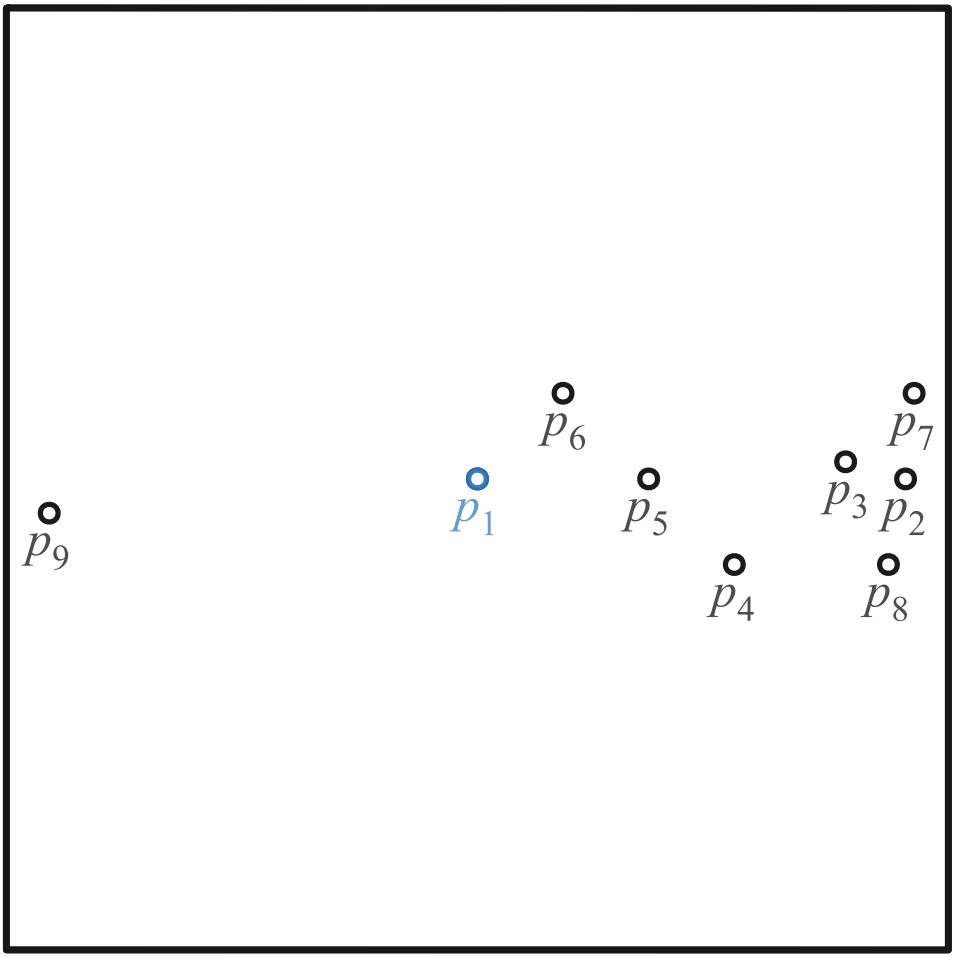


Obrázek B.5: Nejlepší případ aplikace metody VST – krok 3

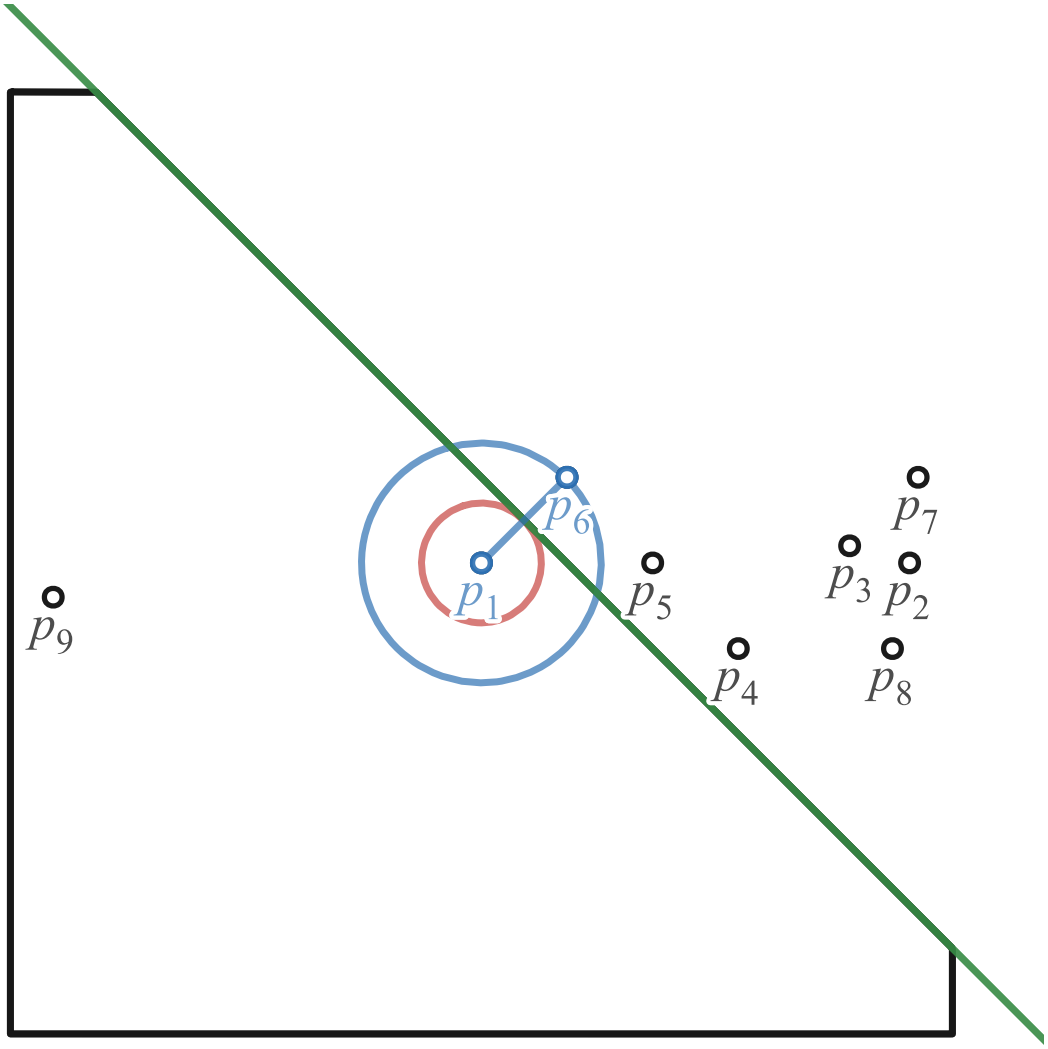


Obrázek B.6: Nejlepší případ aplikace metody VST – krok 4 (kritérium pro ukončení splněno – konec algoritmu)

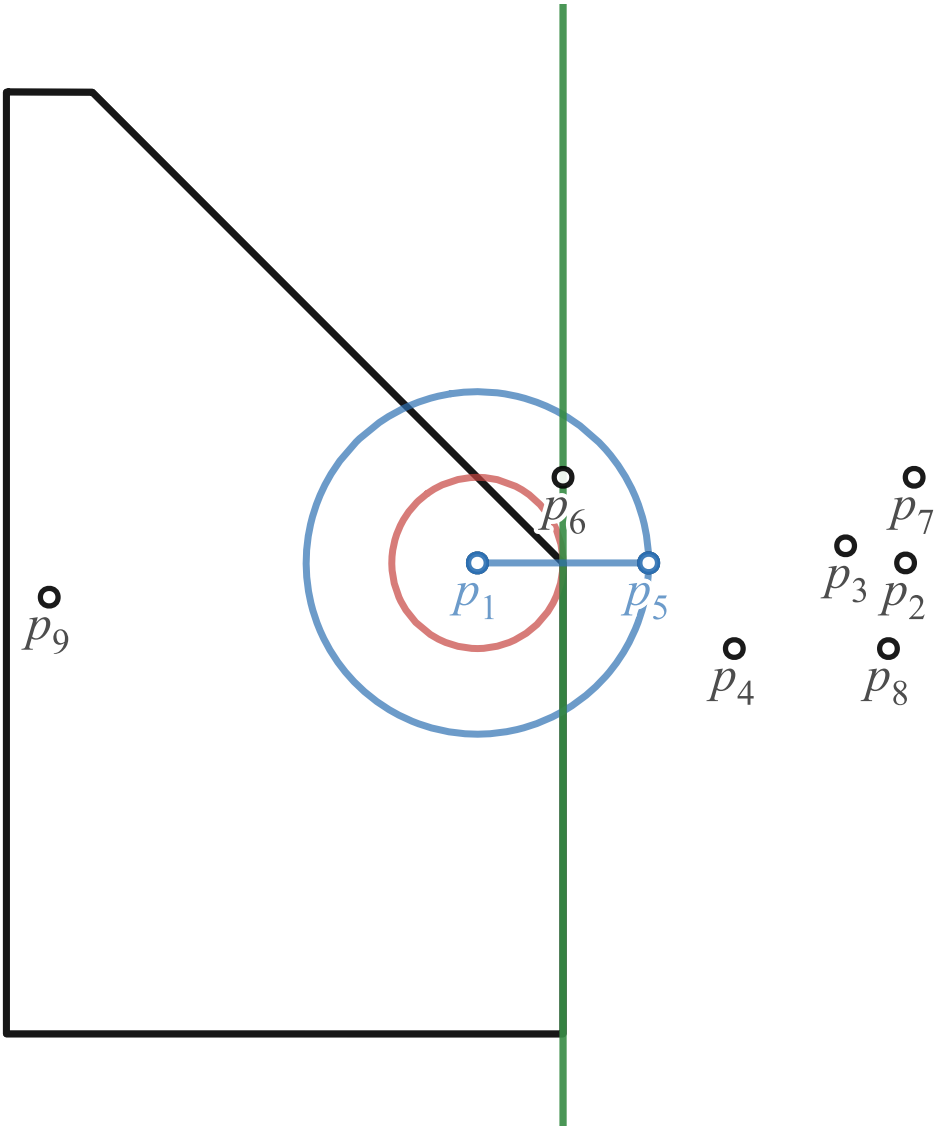
B.2 Nejhorší případ



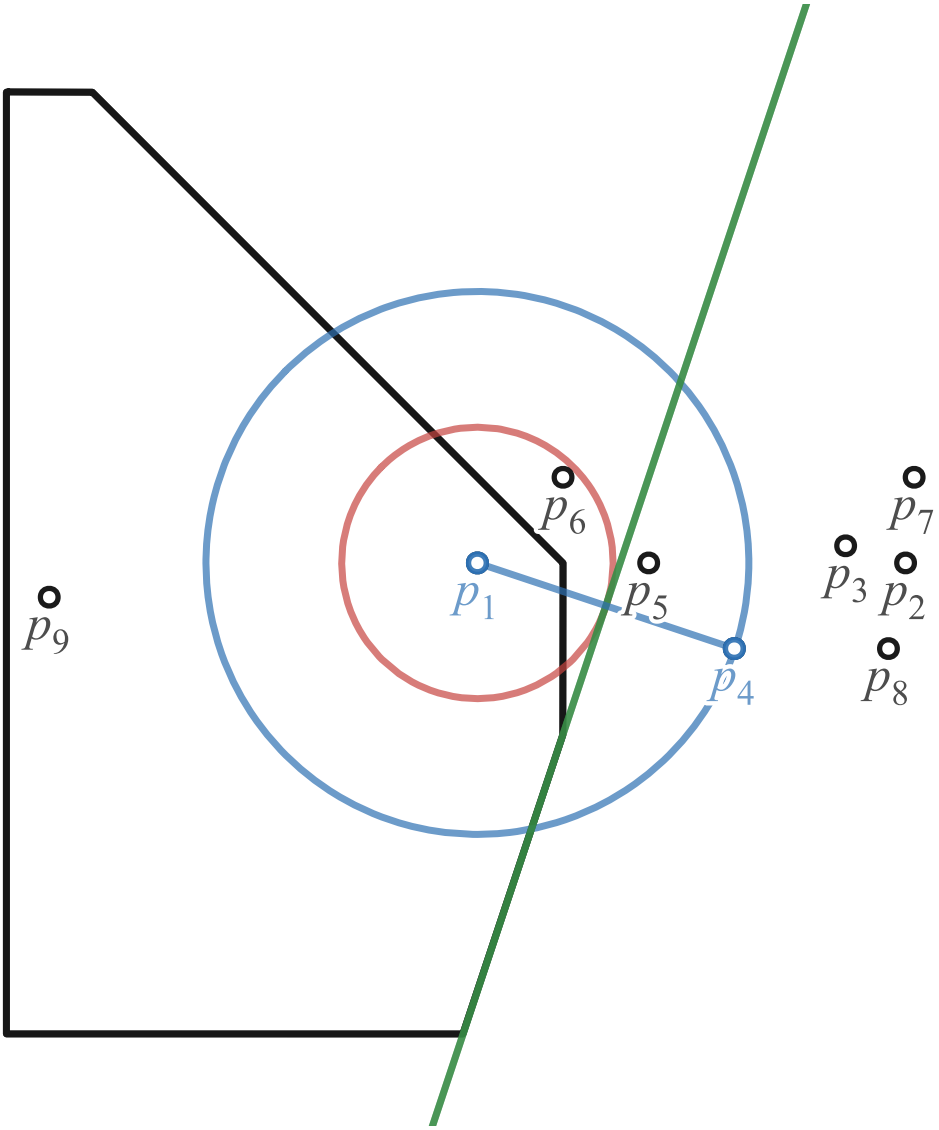
Obrázek B.7: Nejhorší případ aplikace metody VST – krok 0



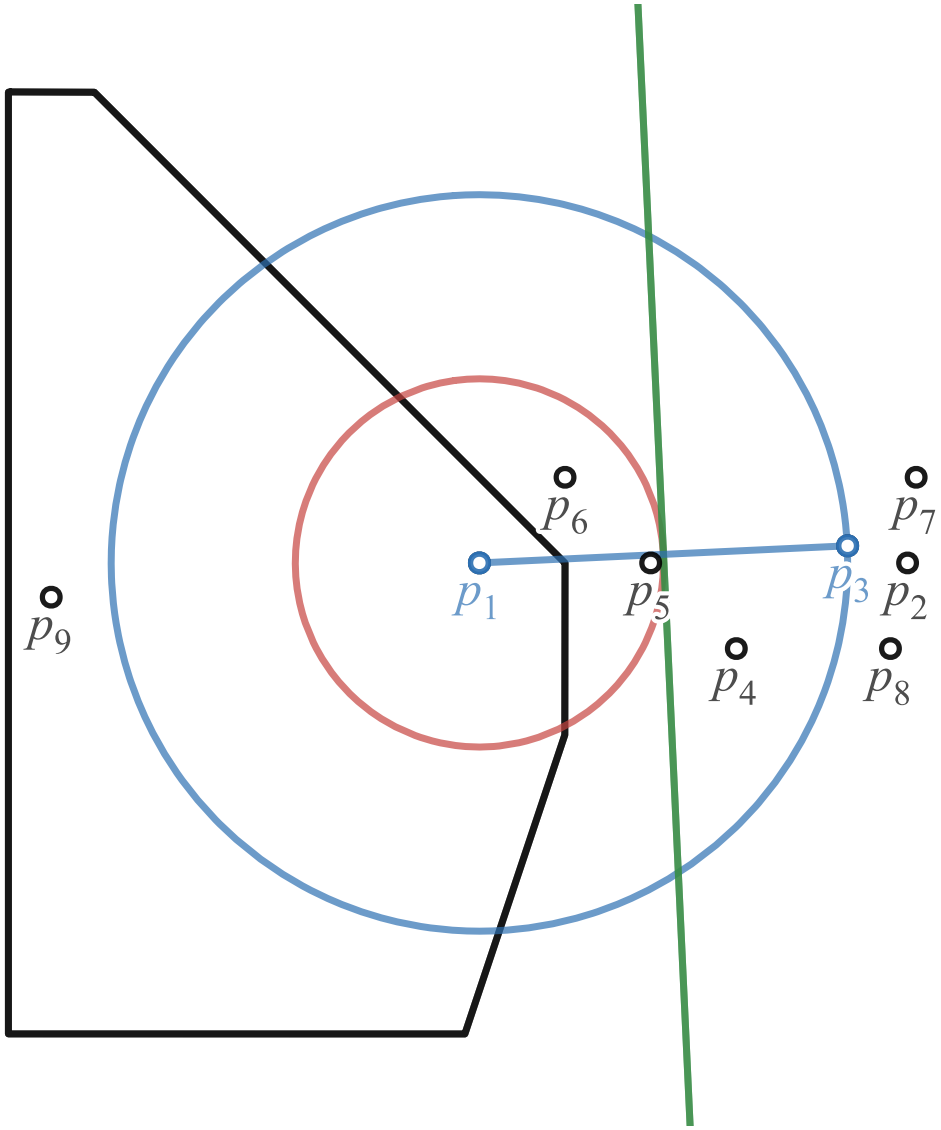
Obrázek B.8: Nejhorší případ aplikace metody VST – krok 1



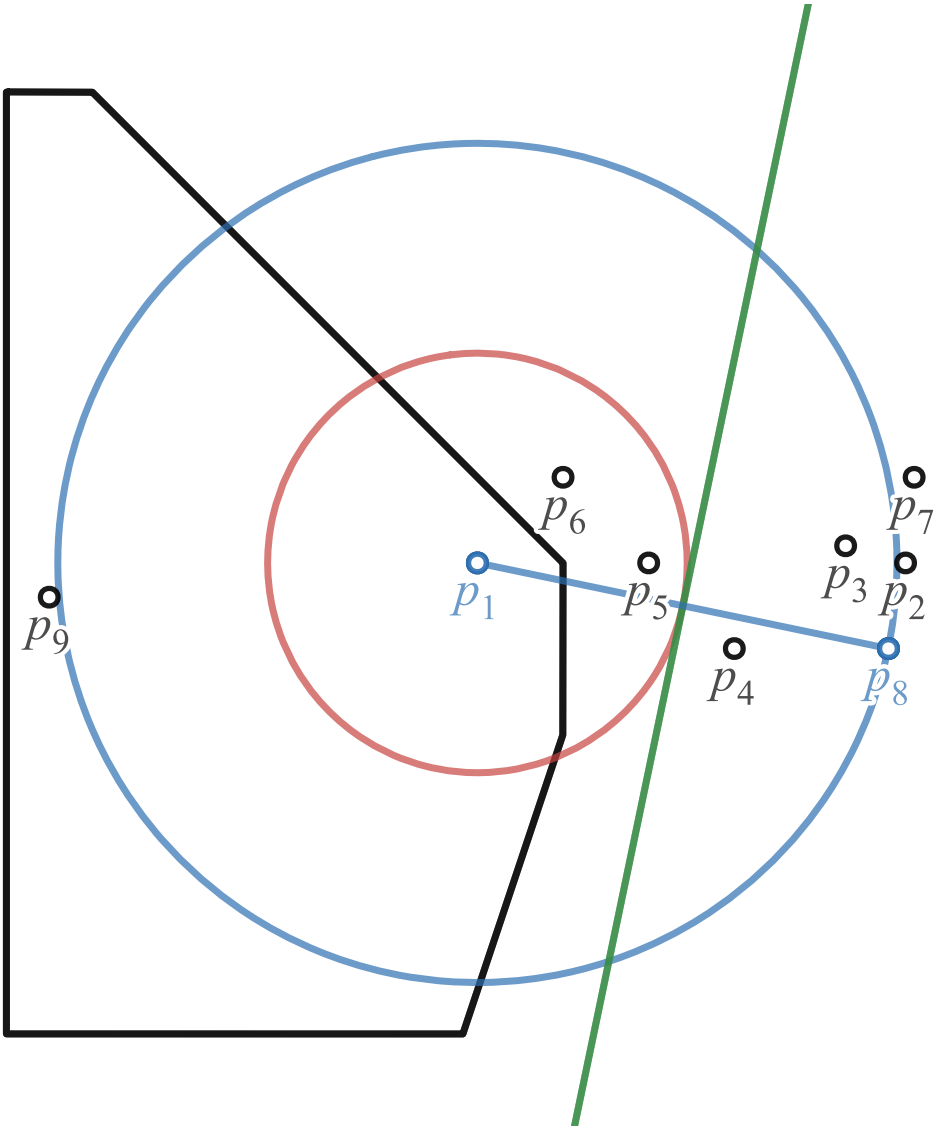
Obrázek B.9: Nejhorší případ aplikace metody VST – krok 2



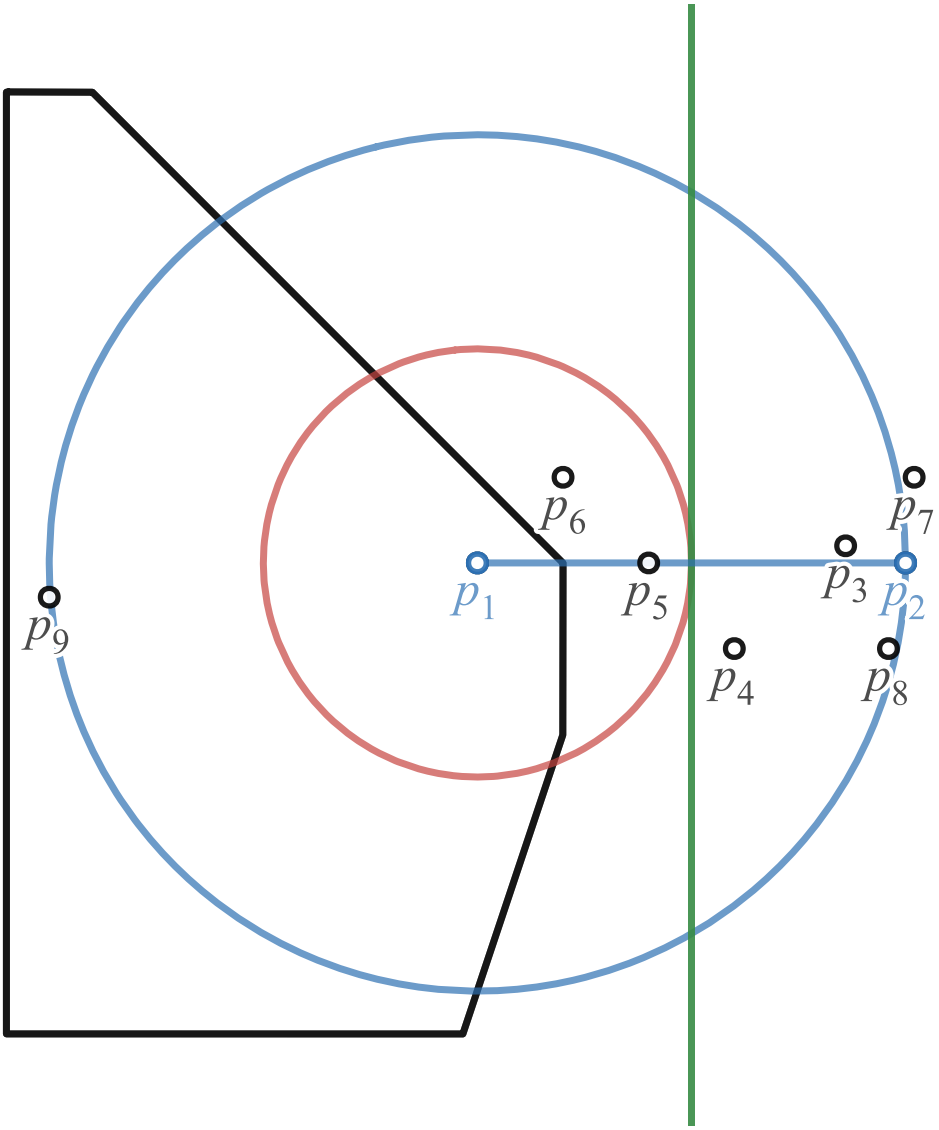
Obrázek B.10: Nejhorší případ aplikace metody VST – krok 3



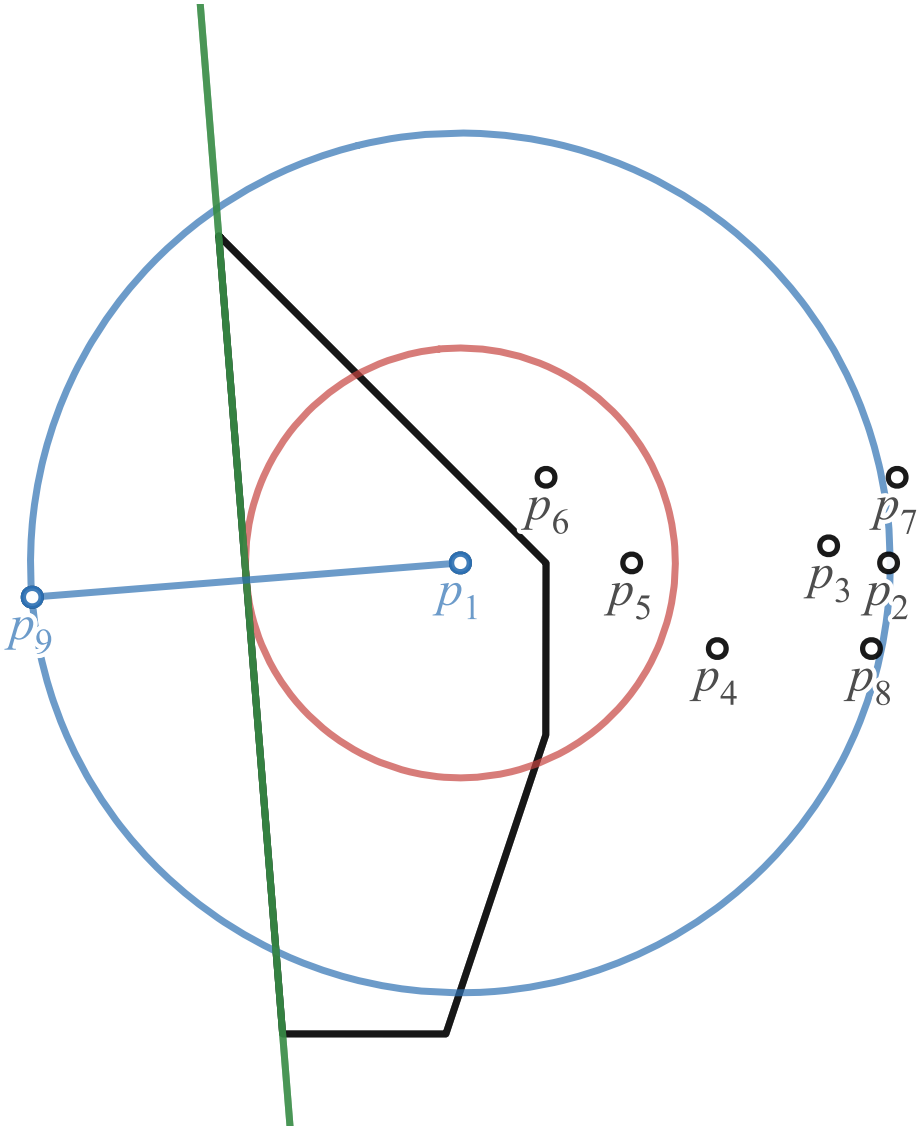
Obrázek B.11: Nejhorší případ aplikace metody VST – krok 4



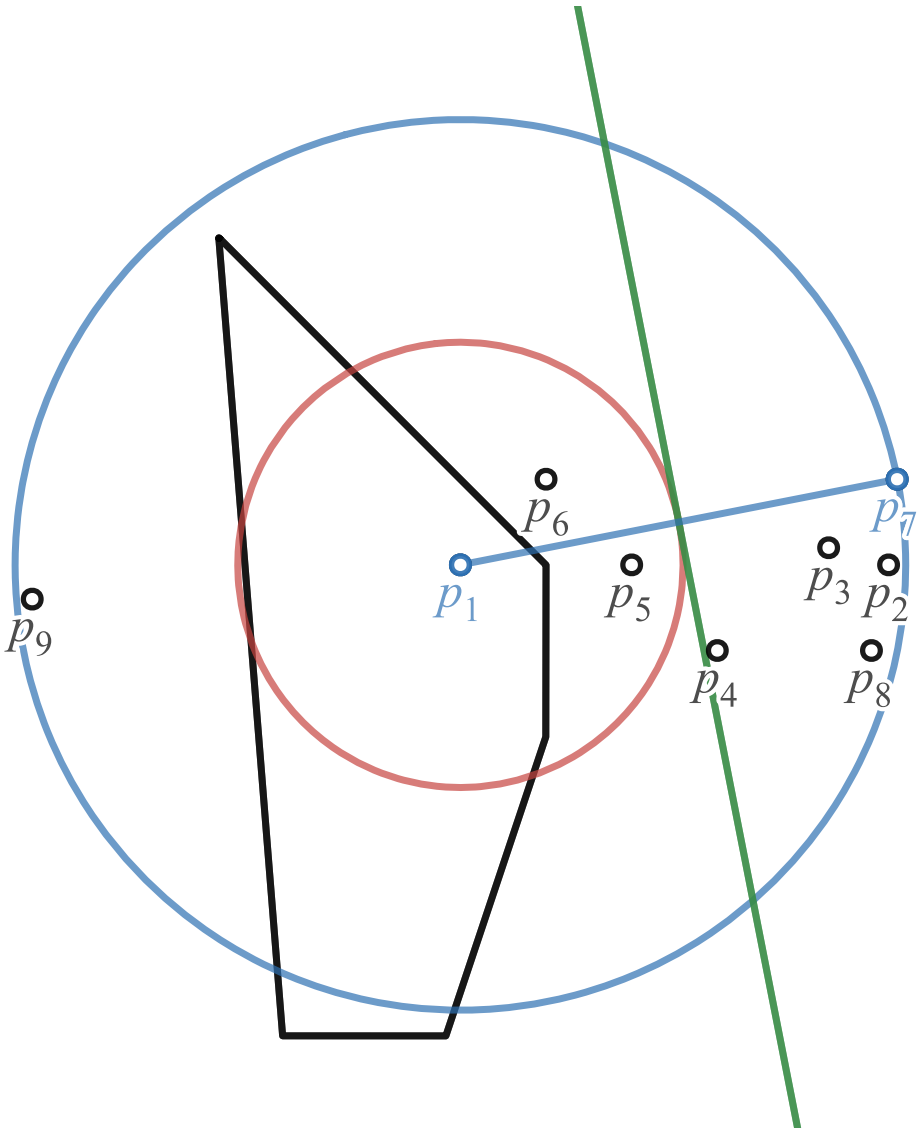
Obrázek B.12: Nejhorší případ aplikace metody VST – krok 5



Obrázek B.13: Nejhorší případ aplikace metody VST – krok 6

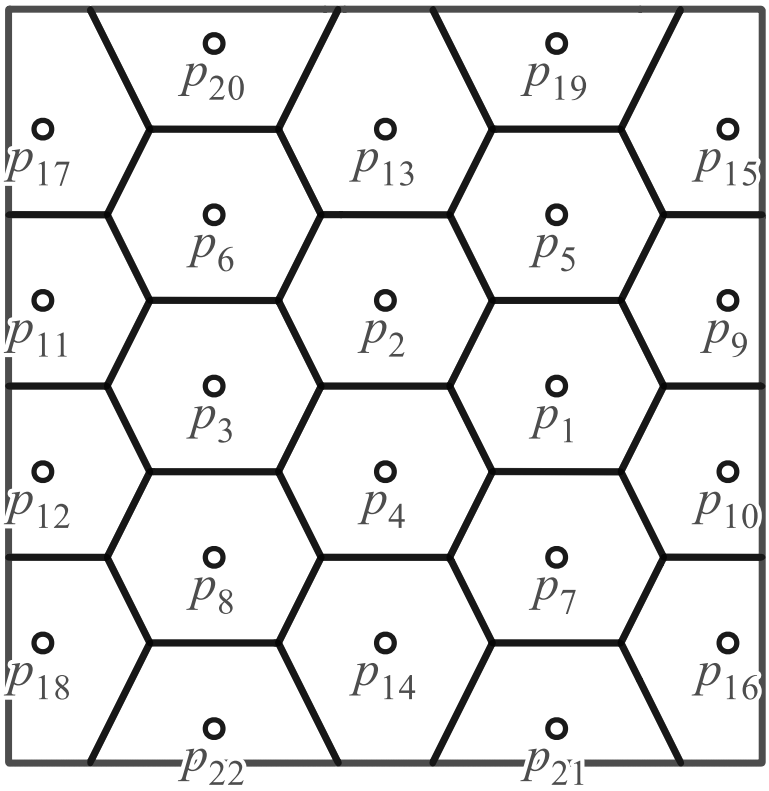


Obrázek B.14: Nejhorší případ aplikace metody VST – krok 7

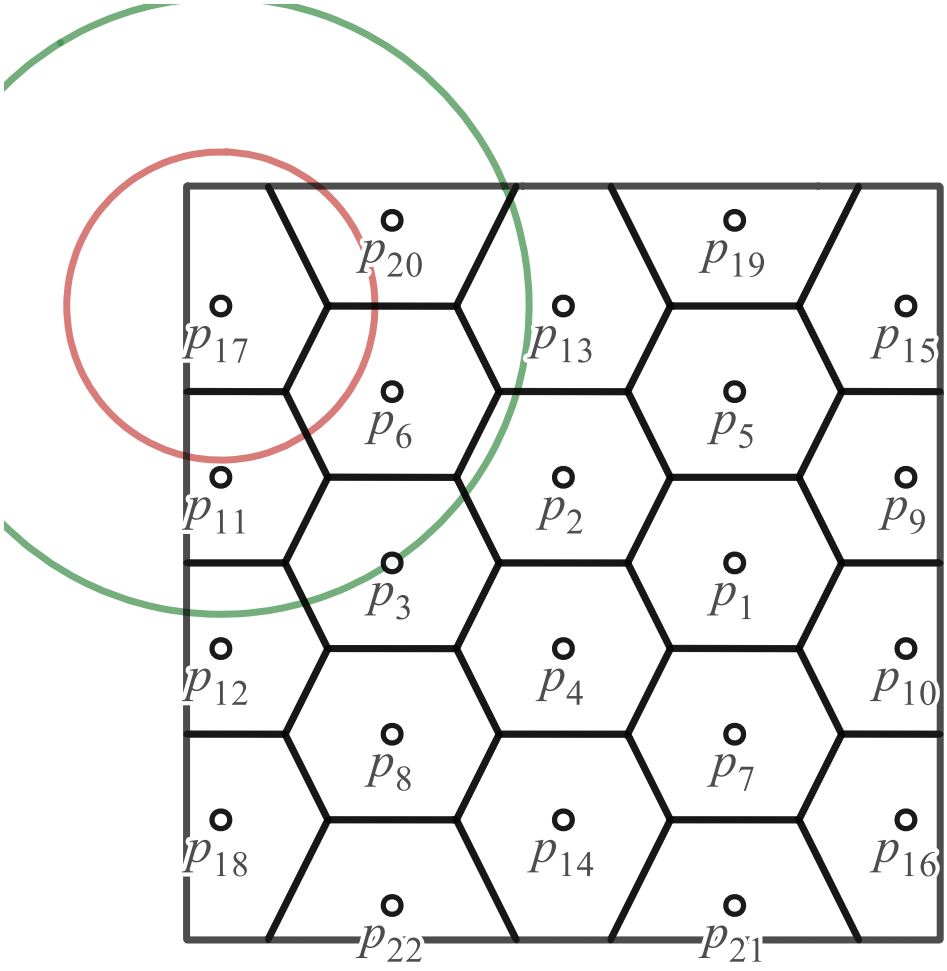


Obrázek B.15: Nejhorší případ aplikace metody VST – krok 8 (všechny body byly otestovány – ukončení algoritmu)

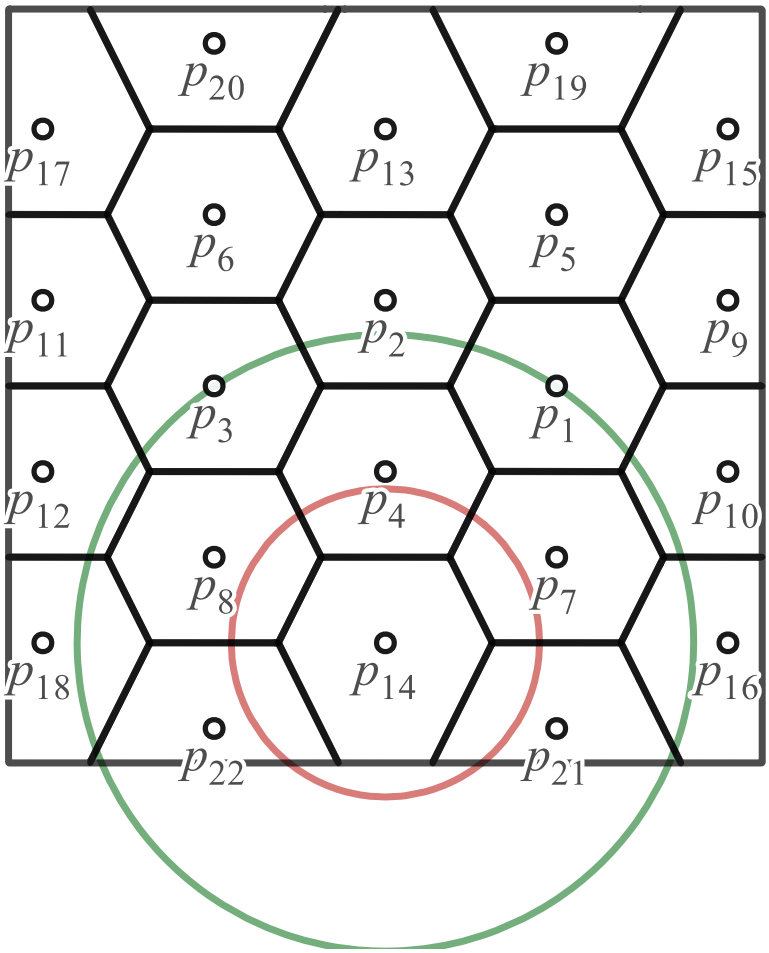
B.3 Ideální rozmístění jader



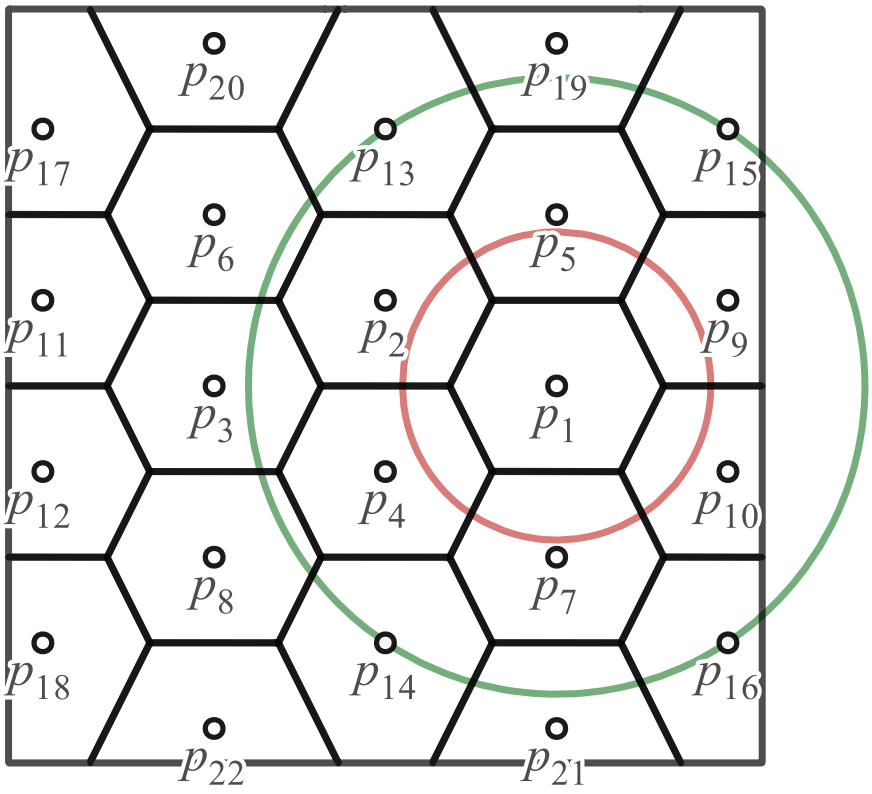
Obrázek B.16: Ukázka možného ideálního rozmístění jader v rozkrajovaném objektu



Obrázek B.17: Ukončovací kritérium pro p_{17} – ořezávání odpovídá ideálnímu přístupu



Obrázek B.18: Ukončovací kritérium pro p_{14} – ořezávání odpovídá ideálnímu přístupu



Obrázek B.19: Ukončovací kritérium pro p_1 – ořezávání odpovídá ideálnímu přístupu