Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Efektivní implementace registrace 3D povrchů

Plzeň 2022 Miroslav Levora

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Miroslav LEVORA**
Osobní číslo:	**A19B0122P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Efektivní implementace registrace 3D povrchů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s algorimem pro registraci 3D povrchů a jeho implementací vyvíjenou na KIV, identifikujte především jejich výkonnostní nedostatky.
2. Navrhněte způsob reimplementace algoritmu se zaměřením na efektivní vykonání.
3. Implementujte navržený postup a ověřte shodnost výsledků s původní implementací.
4. Vytvořený software důkladně otestujte a zdokumentujte.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Doc. Ing. Libor Váša, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **4. října 2021**
Termín odevzdání bakalářské práce: **5. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 14. října 2021

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 5. května 2022

Miroslav Levora

Poděkování

Chtěl bych velmi poděkovat vedoucímu práce panu doc. Ing. Liborovi Vášovi, Ph.D. za ochotu, vstřícnost a trpělivost, kterou mi v průběhu vypracování práce věnoval.

Abstract

The goal of this work is the effective implementation of the algorithm for surface registration developed at KIV.

Abstrakt

Cílem práce je efektivní implementace algoritmu pro registraci povrchů vyvíjeného na KIV.

Obsah

1 Úvod

Práce navazuje na práci L. Hrudy, J. Dvořáka a L. Váši, která se zabývala porovnáním různých metrik pro vyhodnocení vzdálenosti transformací [3]. V rámci této vědecké publikace vzniknul program registrující dvě trojúhelníkové sítě využívající zjištěných poznatků. Cílem této práce je efektivnější implementace tohoto programu.

Text je rozdělen do 3 velkých celků skládají se z několika kapitol. Prvním celkem je všeobecný, krátký, úvod do témat, jejichž pochopení je klíčové pro porozumění zbytku textu. Druhý celek je teoretický popis implementovaného algoritmu. Poslední, třetí, část popisuje využité optimalizační metody.

1.1 Úvod do počítačové grafiky

Počítačová grafika je disciplína informačních technologií zabývající se generováním digitálního vizuálního obsahu. Jelikož text práce a vzniklý program se převážně zabývají 3D grafikou, je třeba vysvětlit několik myšlenek a pojmů.

1.1.1 Reprezentace objektů

3D objekty mohou být reprezentovány matematicky - implicitním nebo parametrickým předpisem, ale z praktických důvodů bývají objekty často aproximované trojúhelníkovou sítí. Objekt s nekonečným množstvím bodů je tedy aproximován konečnou množinou trojúhelníků.

Obrázek 1.1: Koule(vlevo) reprezentovaná implicitním předpisem $x^2 + y^2 + z^2 = 1$ v kontrastu s koulí(vpravo) reprezentovanou trojúhelníkovou sítí

1.1.2 Registrace ploch

Registrace ploch je proces, kdy se pro dvě různé trojúhelníkové sítě, které představují části stejného objektu, hledá transformace taková, která po aplikování na jednu ze sítí má za důsledek to, že sítě tvoří logický celek.

Obrázek 1.2: Příklad registrace - vstupní objekty(1. a 2.) a výsledek registrace(3.)

1.1.3 Využití registrace

Potřeba registrace dvou povrchů je častá při převádění objektů reálného světa do digitální podoby. Reálné těleso je naskenováno hloubkovým skenerem. Výsledkem takového skenu je několik trojúhelníkových sítí. Pro kontrukci celého objektů z částí je tedy třeba registrace.

1.2 Křivost

Křivost je klíčovým termínem z oblasti diferenciální geometrie [6]. Pro pochopení problematiky nejprve uvažujme jednodušší verzi v dvoudimenzionálním prostoru $\mathbb{R}^2$.

1.2.1 Křivost v $\mathbb{R}^2$

Jsme-li v $\mathbb{R}^2$ uvažujme jednorozměrné geometrické těleso nazývající se křivka. Křivka je formálně definována jako funkce k s parametrem u, kde:

$$k(u) = \begin{bmatrix} x(u) \\ y(u) \end{bmatrix}$$

Funkce x je funkce x-ové souřadnice pro parametr u a funkce y je funkce y-ové souřadnice pro parametr u. Jako příklad můžeme úvést například jednotkovou kružnici se středem $[0; 0]$. Pro kružnici bude u z intervalu $< 0, 2\pi >$.

Funkcemi x, y pro parametr u budou:

$$x(u) = cos(u)$$

$$y(u) = sin(u)$$

Dále definujeme derivaci funkce k jako:

$$k'(u) = \begin{bmatrix} x'(u) \\ y'(u) \end{bmatrix},$$

která geometricky odpovídá tečně v bodě $[x(u), y(u)]^T$ pro libovolné u(viz. obrázek 1.4)

Jelikož ale pro jednu křivku existuje množství různých parametrizací u, je třeba pro matematickou přesnost přeparametrizovat křivku tak, aby nová parametrizace skutečně vypovídala o tvaru křivky.

Nový parametr nazveme parametrizací obloukem a označíme $s(u)$. Pro $s(u)$ platí:

$$s(u) = \int_a^u \|k'(t)\| \, dt$$

Geometrický význam nového parametru lze interpretovat jako vzdálenost uražená po křivce. Konečně, pomocí derivace a nové parametrizace můžeme formálně definovat křivost κ jako:

$$\kappa(s) = \|k''(s)\|$$

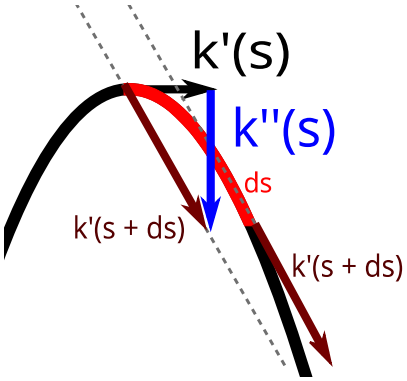
Křivost je tedy velikost změny tečného vektoru a může být interpretována jako míra vychýlení křivky od tečny. Alternativní definicí křivosti je:

$$\kappa(s) = k''(s) \cdot n(s)$$

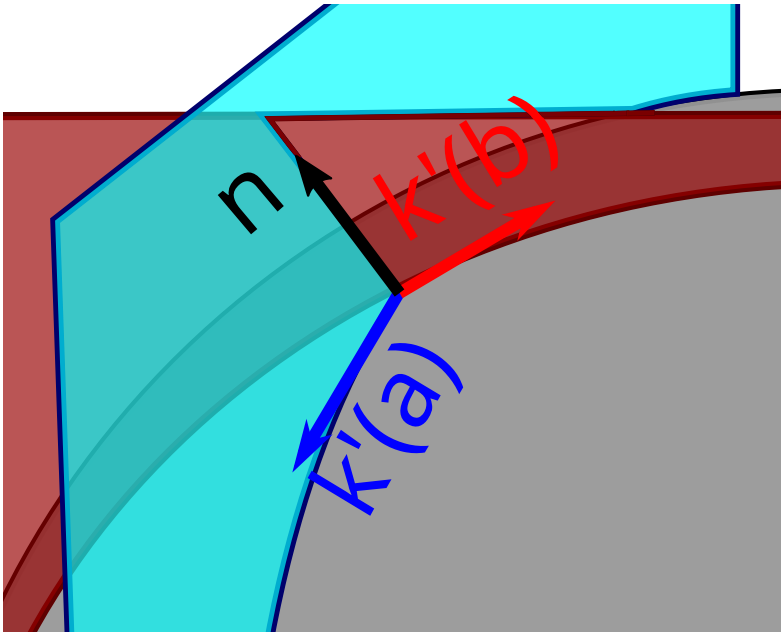
Tato definice má výhodu takovou, že křivost je se znaménkem a znaménko je determinované směrem normály.

1.2.2 Křivost v $\mathbb{R}^3$

Definice křivosti v $\mathbb{R}^3$ se oproti $\mathbb{R}^2$ liší v tom, že pro každý bod na povrchu existuje nekonečné množství možný křivek, které bodem procházejí. Křivost je tedy závislá na směru, který zvolíme. Pro nás budou důležité dva směry e_1 a e_2. Tyto dva vektory nazveme hlavní směry křivosti a bude platit, že v bodě p je největší křivost ve směru e_1 a nejmenší křivost ve směru e_2.



Obrázek 1.3: Geometrický význam druhé derivace křivky



Obrázek 1.4: Derivace dvou křivek procházející bodem ve směrech a a b

1.2.3 Výpočet křivosti v diskrétním prostoru

Předchozí kapitoly formálně definovaly křivost pro spojité matematické plochy. Jelikož program ale pracuje s trojúhelníkovými síťemi, je třeba uvést způsob, jakým se křivosti počítají na nespojité ploše. Pro výpočet křivostí byl použita modifikovaná Rusinkiewiczova metoda [8]. Spočívá v tom, že hledáme zpětně Shape operator pomocí přeurčené soustavy rovnic.

Mějme bod p a kolem něj N sousedů $v_1, v_2 \dots v_N$ s normálami $n_i, n_2 \dots n_N$, a ortonormální bázi, (u, v) ležící v rovině definované normálou(ortonormální bázi si lze představit jako vektory $k'(a)$, $k'(b)$ z obrázku 1.4). Poté pro kaž-

dého souseda v_i platí rovnice:

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} (v_i - p) \cdot u \\ (v_i - p) \cdot v \end{bmatrix} = \begin{bmatrix} n_i \cdot u \\ n_i \cdot v \end{bmatrix}$$

Vlivem diskretizace plochy nejsme schopni nalést přesné hodnoty a, b, c, proto řešíme soustavu N takových rovnic metodou nejmenších čtverců. Vlastní vektory Shape operatoru $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ jsou hlavní směry křivosti e_1, e_2 a vlastní čísla jsou hlavní křivosti.

2 Popis algoritmu

Vstupem algoritmu jsou dvě trojúhelníkové sítě **P** a **Q**. Výstupem algoritmu je transformace, která transformuje Q na P tak, aby se sítě vzájemně překrývaly a tvořily větší část výsledného objektu. Algoritmus probíhá ve 4 hlavních krocích.

Prvním krokem je rovnoměrné navzorkování sítě **P** a dekorelování bodů sítě **Q** rotací a translací. Rovnoměrné navzorkování sítě **P** je nutné z důvodů eliminování oblastí s velkou hustotou bodů. Tyto oblasti by poté mohly potenciálně ovlivňovat prostor vygenerovaných transformací. Dekorelovní bodů sítě **Q** je klíčové pro rychlé vyhodnocování vzdálenosti kandidátních transformací(viz. 2.2.3).

Druhým krokem je výpočet křivostí a směrů křivostí pro navzorkované body. Křivosti se počítají pro obě předzpracované trojúhelníkové sítě **P** a **Q**. Výpočet křivostí probíhá způsobem popsaným v sekci 1.2.3.

Třetím krokem je vytvoření kandidátních transformací. Po vytvoření transformace(popsáno v sekci 2.1) je třeba transformaci verifikovat. Verifikace probíhá tak, že po aplikování zkonstruované transformace na síť **Q** se určí procento bodů z **Q** ležící v blízkosti bodů z **P**. Je-li toto procento alespoň 3%, transformace je verifikována a prohlášena za kandidáta.

Čtvrtým krokem je hledání oblasti s největší koncentrací transformací v prostoru kandidátních transformací. Poté je nalezena transformace nejbližší k těžišti takové oblasti. Výslednou transformací, která registruje sítě, je poté kombinace transformace dekorelující síť **Q** a transformace nejbližší těžišti.

2.1 Konstrukce kandidátních transformací

Konstrukce kandidátních transformací probíhá tak, že pro každý bod z množiny Q se hledá bod z množiny P takový, který má nejpodobnější křivosti. Původní bod z množiny Q označíme q. Nalezený bod z množiny P označíme p. Normálu q označíme n_q, hlavní směr křivosti κ_q a vektor kolmý na normálu a směr křivosti o_q. Analogicky pro p.
Poté hledáme matici, označíme R, která transformuje $n_q \rightarrow n_p$, $\kappa_q \rightarrow \kappa_p$ a $o_q \rightarrow o_p$. Transformace R se dá rozložit na součin dvou jednodušších trans-

formací: R_1 a R_2

$$R = R_1 R_2,$$

kde R_1 transformuje n_q, κ_q a o_q do kanonické báze. R_2 transformuje poté z kanonické báze do báze určené vektory n_p, κ_p a o_p.
Konstrukce R_2 je triviální, jelikož transformace z kanonické báze do libovolné báze spočívá v tom, že se vektory libovolné báze vloží do matice transformace jako sloupce.

$$R_2 = \begin{bmatrix} n_{px} & \kappa_{px} & o_{px} \\ n_{py} & \kappa_{py} & o_{py} \\ n_{pz} & \kappa_{pz} & o_{pz} \end{bmatrix}$$

Hledání R_1 není o moc složitější. Hledáme matici, která transformuje naše vektory na kanonické, čili řešíme rovnici:

$$R_1 \begin{bmatrix} n_{qx} & \kappa_{qx} & o_{qx} \\ n_{qy} & \kappa_{qy} & o_{qy} \\ n_{qz} & \kappa_{qz} & o_{qz} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Po separování R_1:

$$R_1 = \begin{bmatrix} n_{qx} & \kappa_{qx} & o_{qx} \\ n_{qy} & \kappa_{qy} & o_{qy} \\ n_{qz} & \kappa_{qz} & o_{qz} \end{bmatrix}^{-1}$$

Pro libovolnou matici rotace A platí:

$$AA^{-1} = AA^T = I$$

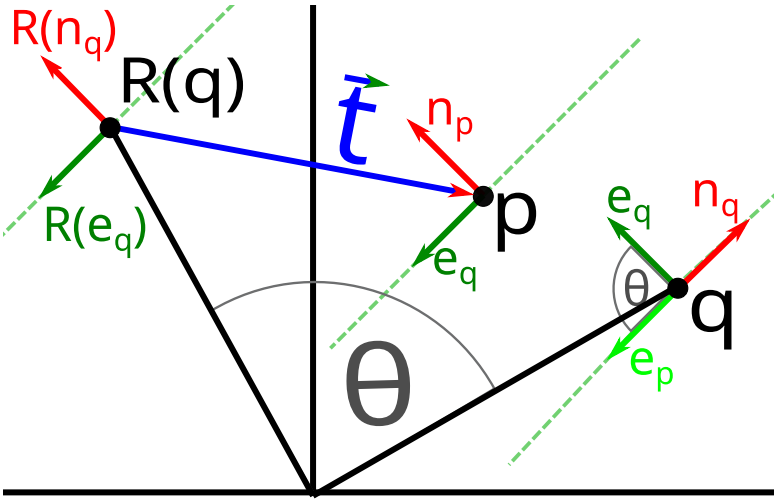
Po aplikování skutečnosti, že inverzní matice k matici rotace je matice rotace transponovaná, platí:

$$R_1 = \begin{bmatrix} n_{qx} & n_{qy} & n_{qz} \\ \kappa_{qx} & \kappa_{qy} & \kappa_{qz} \\ o_{qx} & o_{qy} & o_{qz} \end{bmatrix}$$

Po aplikování transformace na bod q se normála a směr křivosti sice s bodem p shodovat bude, ale souřadnice bodů ne(viz. obrázek 2.4). Rotace totiž pouze otáčí body kolem počátku.

Pro správné namapování bodů je třeba bod i posunout. Hledáme tedy $[t_x, t_y, t_y]^T$ takové, aby platilo:

$$R \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$



Obrázek 2.1: Potřeba translace

Po osamostatnění vektoru translace tedy:

$$\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} - R \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix}$$

Kompletní transformace, která transformuje normály, směry křivostí a souřadnice T je pro libovolný zdrojový bod q a cílový p definována jako spojení rotace a translace:

$$T(q,p) = Rq + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

2.2 Analýza prostoru transformací

Pro nalezení výsledné transformace je třeba najít oblast s nejvyšší hustotou transformací v prostoru kandidátních transformací. Toho je docíleno tak, že každé transformaci t_i z množiny kandidátních transformací M, která má $1, 2, 3 \ldots n$ transformací je přiřazeno skóre s:

$$s(t_i) = \sum_{j \neq i}^{n} K(d(t_i, t_j))$$

Kde d je metrika/vzdálenost dvou transformací a K je funkce mapující vzdálenosti na rozumné hodnoty. Na funkci K klademe požadavky:

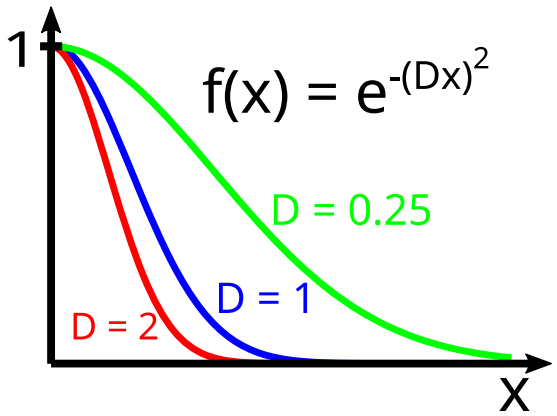
- K(0) je konstanta

- K je klesající
- rychlost klesání K je parametrizovatelné.
- Derivace v bodě 0 je nulová
- K > 0 a zároveň $\lim_{x \to \text{inf}} K(x) = 0$

Takové podmínky splňuje například funkce(viz. obrázek) :

$$K(x) = e^{(-Dx)^2}$$

Kde D je parametr určující rychlost klesání.



Obrázek 2.2: Funkce vzdálenosti

2.2.1 Funkce vzdálenosti

Hledání funkce vzdálenosti transformací d je složitější, jelikož prostor transformací je sice metrický, ale není euklidovský. Existuje velké množství různých metrik. Každopádně, závěrem vědeckého článku, na který je tato práce pokračování, je, že nelze objektivně(tj. bez dat, na které transformace aplikujeme) poměřit dvě transformace. Jako funkce vzdálenosti tedy byla zvolena funkce:

$$d(T_1, T_2) = \sum_{i}^{n} \|(T_1(p_i) - T_2(p_i)\|_2,$$

což není nic jiného než suma vzdáleností bodů po ztransformování. Výpočetním tvarem této metriky se zabývá sekce 2.2.3

2.2.2 Hledání transformace s nejvyšší hustotou transformací ve svém okolí

Hledání takové transformace pak probíhá velmi jednoduše. Skóre s je vypočteno pro všechny transformace a transformace s nejvyšším skórem je určena nejlepší možnou. Formálně tedy:

$$t^* = \max\{s(t_i) : i = 1, 2, \ldots n\}$$

2.2.3 Optimalizace výpočtu vzdáleností transformací

Vycházíme z myšlenky, že transformace nelze porovnávat bez dat. Můžeme tedy vzdálenost transformací definovat metrikou:

$$d(T_1, T_2) = \sum_i^n \|(T_1(p_i) - T_2(p_i)\|_2,$$

kde d je funkcí vzdáleností dvou transformací T_1, T_2, n je počet vrcholů trojúhelníkové sítě, p je bod reprezentující jeden z vrcholů sítě, a $\|.\|_2$ je euklidovská norma, čili:

$$\|.\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots x_n^2}$$

V případě porovnávání dvou bodů ze 3D prostoru tedy:

$$\|p - q\|_2 = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$$

Transformaci rozepíšeme jako translaci a rotaci:

$$d(T_1, T_2) = \sum_i^n \|(R_1 p_i + t_1 - (R_2 p_i + t_2))\|_2$$

Rozepíšeme normu tak, že se nejprve zbavíme odmocniny(výsledkem nebude d, ale d^2) a kvadráty převedeme na vektorové násobení

$$d(T_1, T_2)^2 = \sum_i^n (R_1 p_i + t_1 - (R_2 p_i + t_2))^T (R_1 p_i + t_1 - (R_2 p_i + t_2))$$

Přeskupením a vytknutím se dostaneme na výraz

$$d(T_1, T_2)^2 = 2\sum_i^n (p_i^T p_i) + 2(t_1 - t_2)^T R_1 \sum_i^n (p_i) + 2(t_2 - t_1)^T R_2 \sum_i^n (p_i)$$

$$+ n t_1^T t_1 - 2n t_1^T t_2 + n t_2^T t_2 - 2R_1^T R_2 : \sum_i^n (p_i p_i^T)$$

Vzdálenost transformací stále závisí na transformaci všech bodů. Můžeme ale eliminovat členy

$$2(t_1 - t_2)^T R_1 \sum_i^n (p_i) \quad \& \quad 2(t_2 - t_1)^T R_2 \sum_i^n (p_i)$$

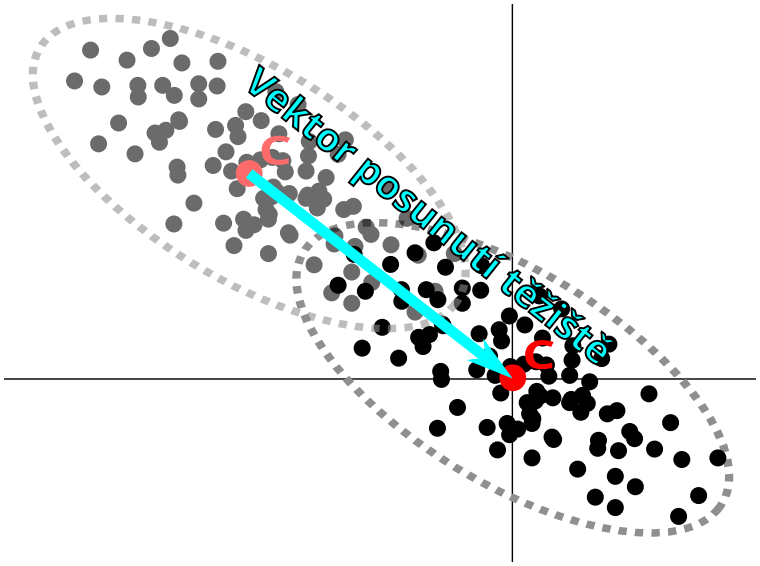
tím, když zaručíme, že suma všech bodů p_i bude 0. Toho je dosáhnuto přesunutím těžiště množiny bodů do středu soustavy souřadnic. Když tedy označíme těžiště množiny bodů c, tak:

$$c = \frac{1}{n} \sum_i^n (p_i)$$

Po přesunutí množiny bude platit:

$$0 = \frac{1}{n} \sum_i^n (p_i) \iff \sum_i^n (p_i) = 0$$

Jestli že je suma nulová, je celý člen nulový. Po předzpracování sítě translací



Obrázek 2.3: Posun těžiště množiny do středu soustavy souřadnic

máme tedy tvar vzdálenosti:

$$d(T_1, T_2)^2 = 2 \sum_i^n (p_i^T p_i) + n t_1^T t_1 - 2 n t_1^T t_2 + n t_2^T t_2 - 2 R_1^T R_2 : \sum_i^n (p_i p_i^T)$$

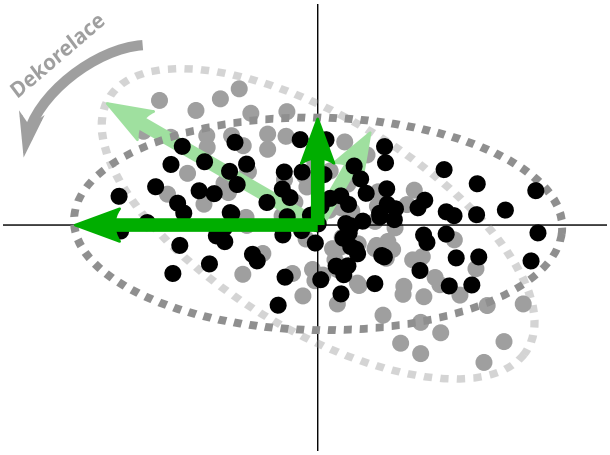
Když obohatíme myšlenku předzpracování trojúhelníkové sítě ještě o rotaci, můžeme značně zjednodušit poslední člen

$$-2 R_1^T R_2 : \sum_i^n (p_i p_i^T)$$

Kvůli výskytu zjednodušené kovariační matice $\sum_i^n (p_i p_i^T)$ můžeme zaručit, že matice bude diagonální tehdy, když trojúhelníkovou síť zrotujeme tak, aby její body byly vzájemně dekorelované. Bude-li tedy zjednodušená kovariační matice diagonální, kvůli charakteru Frobeniova maticového násobení stačí, abychom ze součinu matic rotací uvažovali také jen diagonálu. Bude tedy platit:

$$diag(-2R_1^T R_2) : diag(\sum_i^n (p_i p_i^T))$$

Po předzpracování trojúhelníkové sítě můžeme vzdálenost tedy definovat



Obrázek 2.4: Dekorelování množiny bodů v 2D

jako:

$$d(T_1, T_2)^2 = 2\sum_i^n (p_i^T p_i) + nt_1^T t_1 - 2nt_1^T t_2 + nt_2^T t_2 - diag(-2R_1^T R_2) : diag(\sum_i^n (p_i p_i^T))$$

Kde pro jedny vstupní data (jednu trojúhelníkovou síť) jsou tyto dva mnohočleny předpočitatelné konstanty

$$d(T_1, T_2)^2 = 2\sum_i^n (p_i^T p_i) \quad \& \quad diag(\sum_i^n (p_i p_i^T))$$

Ačkoliv jsou tedy vzdálenosti závislné na vstupních datech, jejich vzdálenost se dají **určit s $\mathcal{O}(1)$ složitostí**. Tato metrika má nevýhodu takovou, že hodnota vzdáleností závisí na hustotě bodů. Problém s hustotou vzorkování lze vyřešit tím, že místo vzdáleností bodů uvažujeme vzdálenosti trojúhelníků. Myšlenka finální metriky tedy zůstává stejná, jen místo sumy přes body děláme integrály přes plochy trojúhelníků.

3 Uživatelská příručka

Před prvním spuštěním programu je program nejdřív nutné přeložit. Program využívá hlavičkového souboru `pthreads` a funkcí specifických pro systém Linux, program tedy není na Windows přenositelný a nutnou prerekvizitou pro překlad a spuštění programu je operační systém Linux. Překlad se provádí pomocí nástroje `make` a překladače `gcc`. V kořenovém adresáři projektu je tedy nutno z příkazové řádky zadat příkaz:

make

Po přeložení se ve složce out vygeneruje soubor **registrator**. Program předpokládá dva argumenty a spouští se příkazem:

out/registrator <object1.obj> <object2.obj>

Kde *object1* a *object2* jsou názvy souborů ve formátu .obj představující dvě trojúhelníkové sítě, které budou registrovány. Spolu s aplikací jsou v adresáři objects přiložena i testovací data. Příklad spuštění s testovacími daty:

out/registrator objects/arm1.obj objects/arm2m.obj

Výstupem programu je transformace, která po aplikování na *object2* sítě registruje. Příklad užití:

```
» out/registrator objects/arm1.obj objects/arm2m.obj
[0.999521434307 -0.011904919520 0.028550948948 | -0.001115618972]
[0.012726054527 0.999505519867 -0.028754234314 | 0.003650948405]
[-0.028194606304 0.029103934765 0.999178588390 | -0.004376575351]
```

4 Implementace

Pro implementaci algoritmu byl zvolen programovací jazyk C, jelikož je to jazyk, který je ve skupině nejvíce výkonných jazyků[7] a je zároveň osobní preferencí autora. Nebyly použité žádné externí knihovny z výkonnostních důvodů. Většina matematických operací je ručně vektorizována(viz. 6).

4.1 Struktura projektu

V kořenovém adresáři se nachází 4 hlavní podadresáře:

- Adresář **src** - Obsahuje zdrojové, .c, soubory aplikace
- Adresář **headers** - Obsahuje hlavičkové, .h, soubory aplikace
- Adresář **out** - Obsahuje výstup překladu
- Adresář **objects** - Obsahuje testovací soubory .obj

4.2 Popis důležitých částí

V kapitole 2 byl uveden teoretický popis algoritmu. Tato sekce vysvětlí implementaci některých netriviálních kroků.

4.2.1 Hledání dekorelační transformace

Hledání dekorelační transformace probíhá na úplném začátku ve funkci `cluster_registration_execute` a probíhá ve dvou krocích. Prvním krokem je nalezení těžiště, těžiště spolu s váženým těžištěm je spočítáno funkcí `trianglemesh_init_radiuses`. Síť **Q** v programu označená `mesh2` je poté vycentrována.

Hledání rotace dekorelující body je provedeno pomocí analýzy hlavních komponent(funkce `find_rotation`). Funkce nejprve vypočte zjednodušenou kovarianční matici(ve funkci značeno `m`) a nad ní vypočte vlastní vektory(výpočet vlastních vektorů implementován ručně v souboru `math_utils`). Vlastní vektory představují bázi dekorelující transformace. Body a normály jsou poté pomocí této báze ztransformovány.

4.2.2 Verifikace kandidátů

O verifikaci kandidátů se stará soubor `proximity_verifier`. Pro efektivní verifikaci je místo zjišťování vzdálenosti přes všechny body **P** zkonstruovaná mřížka(značeno `grid`). Při verifikaci libovolného bodu je tedy bod převeden na souřadnice mřížky, poté je bod považován za validní(leží v okolí nějakého bodu **P**) jestli že je v mřížce hodnota 1(true).

Grid je vytvořen tak, že pro všechny body z sítě **P** je na odpovídající souřadnici v mřížce vložena 3D koule jedniček. Tvorba gridu je paralelizována pomocí `pthread`.

Verifikování samotné transformace poté probíhá tak, že se postupně na body z množiny **Q** transformace aplikuje a je určen počet bodů validních. Transformace všech bodů z **Q** je zbytečná a pomalá. Transfromuje se tedy jenom navzorkovaný počet(udáno proměnnou `config_sample_size_mesh2`). Je-li poté podíl počtu validních bodů a navzorkovaných bodů alespoň 3% (`config_acceptance_threshold`) je transformace prohlášena za validní a kandidátní.

4.2.3 Konstrukce kandidátů

Konstrukce kandidátů je implementována v souboru `curvature_oracle` a probíhá ve dvou krocích. Nejprve jsou napočítány křivosti pro navzorkovanou síť **P**. Počítání křivosti probíhá metodou popsanou v teoretické části s tím, že sousedi vrcholu se hledají za pomocí datové struktury CornerTable(sekce 7.1). Vrcholy, pro které jsou křivosti počítány jsou určené vzorkováním a výpočet křivostí je paralelizován(navzorkované pole je rozděleno na n částí, kde n je určeno počtem vláken procesoru).

Poté je zkonstruován KD strom, kde vstupní data jsou křivosti bodů ze sítě **P**. Potom začíná samotná konstrukce kandidátů. Ta opět probíhá paralelizovaně a probíhá ve 4 krocích:

- Vypočtení křivosti pro bod z **Q**
- Nalezení, pomocí KD strom, bodu s nejpodobnějšími křivostmi z **P**
- Konstrukce transformace mapující bod z **Q** na bod z **P** (viz. 2.1)
- Verifikace vygenerované transformace.

4.2.4 Hledání nejlepší transformace v prostoru transformací

Nejprve je třeba inicializovat data potřebná pro vyhodnocení vzdálenosti dvou transformací. To se děje v souboru `cluster_registration.c` pomocí funkce `metric_data_init`.

Hledání transformace je implementováno v souboru `density_clusterer.c`. Výpočet hustoty se řídí podle teoretické části s tím rozdílem, že pro transformaci není prohledáván celý prostor, ale jsou uvažovány jenom transformace blízké(kvůli vlastnostem funkce K vzdálené transformace k hustotě nepřispívají). Blízké transformace jsou hledány pomocí datové struktury Vantage Point Tree, viz. sekce 7.3.

5 Překladačová optimalizace

Automatická optimalizace překladačem je velmi důležitá, jelikož poskytuje možnost psát čitelný kód bez úkoru na rychlosti. Automatické optimalizace je dosaženo dvěma hlavními způsoby:

- Volbou správných možností/příznaků při překládání.
- Používání klíčových slov během psaní kódu.

5.1 Možnosti překladu

Zvolený překladač **gcc** poskytuje 4 hlavní optimalizační módy: O0, O1, O2, O3, kde O0 žádným způsobem kód neoptimalizuje a O3 optimalizuje kód všemi prostředky. Pro dosažení nejlepších výsledků je tedy třeba před překladem specifikovat možnost `-O3`. Další užitečnou možností je `-march=native`, která překladač informuje o tom, že může pro předklad použít instrukce specifické pro konkrétní procesor. Například využití vektorových instrukcí(kapitola 6.

5.2 Klíčová slova

Předkladač nerozumí sémantice překládaných funkcí a kvůli tomu často volí zbytečně dlouhou reprezentaci strojovými instrukcemi. Pomocí klíčových slov informujeme překladač o skutečnostech, které vyplývají z kontextu, ve kterém funkce používáme. Překladač pak dokáže na základě těchto skutečností často generovat rychlejší kód.

5.2.1 Klíčové slovo `inline`

Jedná se však o velmi důležitý koncept. Inlining znamená, že místo volání funkce je tělo funkce vloženo na místo volání. Uvažujme následující příklad:

```
int foo() {
   return 2;
}
void main() {
    int i = foo();
    printf("%d\n", i);
```

}

Přeložíme-li program bez jakékoliv optimalizace, inlining nebude proveden a ve vygenerovaném strojovém kódu se bude vyskytovat instrukce skoku.

```
...
<main+17>    callq  0x555555555149 <foo>
...
```

Po zkompilování s optimalizační příznakem `-O1` uvidíme změnu ve vygenerovaném strojovém kódu.

```
<foo+4>    mov     $0x2,%eax
<foo+9>    retq

<main+4>     sub     $0x8,%rsp
<main+8>     mov     $0x2,%edx
<main+13>    lea     0xe9d(%rip),%rsi
<main+20>    mov     $0x1,%edi
<main+25>    mov     $0x0,%eax
```

`foo` bylo přeloženo(první dvě řádky), ale v těle funkce `main` se nikde nevyskytuje. Celé volání je nahrazeno řádkou s prefixem `<main+8>`.

Důležitá poznámka

Prezence tohoto klíčového slova v kódu nemusí být pro efekt inliningu nutná. Důvod je ten, že jakýkoliv optimalizační příznak O1+ povoluje možnost `-finline-functions`. Překladač se tedy snaží inlinovat všechny funkce bez ohledu na toto klíčové slovo. Při výchozím nastavení překladače gcc se inlinování aplikuje jenom uvnitř jedné překládací jednotky. To znamená, že chceme-li propagovat inline funkce do jednotek, kde nejsou funkce deklarovány je praktické funkce(i s tělem), určené k inlinování, napsat do hlavičkových souborů.

5.2.2 Klíčové slovo `restrict`

Klíčové slovo **restrict** řeší problém takzvaného pointer aliasingu. Jazyk C(narozdíl třeba od FORTRANU) vždy předpokládá, že se oblasti dvou pointerů překrývají[1][2]. Neboli že jeden pointer je aliasem druhého. Tento předpoklad platí i pro funkce, jejichž sémantika udává, že tomu tak není. Uvažujme následující příklad:

```
typedef struct {
```

```
    int x, y;
} Point;
void main() {
    Point p0 = {1, 1};
    Point p1 = {1, 2};
    Point p2 = {1, 3};
    Point p3 = {2, 3};
    Point points[] = {p1, p2, p3};

    add_point_to_arr(points, &p0);
}
```

S funkcí `add_point_to_arr(Point* points, Point* p)`, která ke všem bodů v poli `points` přičte bod `p`.

```
void add_point_to_arr(Point* points, Point* p) {
    for(int i = 0; i < 3; ++i) {
        points[i].x += p->x;
        points[i].y += p->y;
    }
}
```

Podíváme-li se na vygenerovaný strojový kód, zjistíme, že překladač program přeložil velmi neoptimálně.

```
<add_point_to_arr+4>     mov    (%rsi),%eax
<add_point_to_arr+6>     add    %eax,(%rdi)
<add_point_to_arr+8>     mov    0x4(%rsi),%eax
<add_point_to_arr+11>    add    %eax,0x4(%rdi)
<add_point_to_arr+14>    mov    (%rsi),%eax
<add_point_to_arr+16>    add    %eax,0x8(%rdi)
<add_point_to_arr+19>    mov    0x4(%rsi),%eax
<add_point_to_arr+22>    add    %eax,0xc(%rdi)
<add_point_to_arr+25>    mov    (%rsi),%eax
<add_point_to_arr+27>    add    %eax,0x10(%rdi)
<add_point_to_arr+30>    mov    0x4(%rsi),%eax
<add_point_to_arr+33>    add    %eax,0x14(%rdi)
<add_point_to_arr+36>    retq
```

Vidíme neustále opakující se řádky

```
<add_point_to_arr+4>     mov    (%rsi),%eax
<add_point_to_arr+8>     mov    0x4(%rsi),%eax
```

Tyto řádky odpovídají načítání složek x a y z bodu **p** do paměti. Co je zvláštní ale je, že se toto děje opakovaně. Přirozenější by bylo načíst tyto hodnoty do dvou registrů před cyklem a poté je jenom přičítat. Jak bylo ale řečeno, C předpokládá, že by se pointery mohly překrývat. To prakticky znamená něco jako takové volání:

```
add_point_to_arr(points, &(points[1]));
```

Pro takový příklad by ale kód vygenerovaný překladačem oproti přirozenějšímu nahrání registrů před začátkem cyklu vrátil výsledek správný. Často se stane, že funkce ačkoliv přebírá pointery, sémanticky se pointery nikdy nebudou překrývat, v takovém případě tuto vlastnost funkce sdělí programátor překladači pomocí klíčového slova `restrict`. Přepíšeme tedy funkci na:

```
void add_point_to_arr(Point* restrict points,\
                      Point* restrict p) {
    for(int i = 0; i < 3; ++i) {
        points[i].x += p->x;
        points[i].y += p->y;
    }
}
```

A opět prozkoumáme vygenerovaný strojový kód

```
<add_point_to_arr+4>     mov    (%rsi),%edx
<add_point_to_arr+6>     mov    0x4(%rsi),%eax
<add_point_to_arr+9>     movdqu (%rdi),%xmm2
<add_point_to_arr+13>    add    %edx,0x10(%rdi)
<add_point_to_arr+16>    movd   %edx,%xmm0
<add_point_to_arr+20>    movd   %eax,%xmm1
<add_point_to_arr+24>    add    %eax,0x14(%rdi)
<add_point_to_arr+27>    punpckldq %xmm1,%xmm0
<add_point_to_arr+31>    punpcklqdq %xmm0,%xmm0
<add_point_to_arr+35>    paddd  %xmm2,%xmm0
<add_point_to_arr+39>    movups %xmm0,(%rdi)
<add_point_to_arr+42>    retq
```

Je vidět, že pomocí klíčového slova `restrict` překladač nejenom bod `p` přednahrál do registrů, ale je i schopen výpočet vektorizovat.

5.2.3 Klíčové slovo `const`

`const` se **zdá** jako další velmi **užitečné** klíčové slovo, pomocí kterého může kompilátor ušetřit několik instrukcí. Bohužel **tomu tak ale není**. Uvažujme příklad:

```
int foo(const Point* p) {
    return (p->x) * 2;
}
 void test(const Point* p) {
     int sum = p->x + p->y + p->z;
     printf("sum: %d\n", sum);

     foo(p);

     sum = p->x + p->y + p->z;
     printf("sum: %d\n", sum);
 }
void main() {
    Point p = {1, 2, 3};
    const Point* pp = &p;
    test(pp);
}
```

> Poznámka: V C je třeba rozlišovat konstantní pointer, pointer na konstantní hodnotu a konstantní pointer na konstantní hodnotu, v tomto pořadí: `const int* i`, `int* const i` a `const int* const i`. V příkladu je tedy uvažován pointer na konstantní hodnotu.

Kde funkce `main`, `test` a `foo` jsou funkce v jiných překladacích jednotkách(aby při maximální optimalizaci nedošlo k inlinování). Napadne nás, že použití `const` v takovém příkladu překladači dá najevo, že funkce `foo` nijak nemění hodnotu `p`, překladač tedy usoudí, že hodnotu `sum` ve funkci `test` stačí vypočítat jen jednou. Po prozkoumání vygenerovaného strojového kódu, ale zjistíme, že opak je pravdou

```
...
# Nascitavani sumy souradnic do registru EDX
<test+12>    mov    %rdi,%rbx
<test+15>    mov    0x4(%rdi),%edx
<test+20>    add    (%rdi),%edx
```

```
<test+22>   add    0x8(%rdi),%edx
...
<test+30>   callq  0x555555555070 <__printf_chk@plt>
...
<test+38>   callq  0x555555555220 <foo>
# Po zavolani foo opet nascitavani znovu
<test+43>   mov    0x4(%rbx),%edx
<test+46>   mov    $0x1,%edi
<test+51>   add    (%rbx),%edx
<test+60>   add    0x8(%rbx),%edx
...
```

Předělání `const Point*` na `const Point* const` nebo dokonce na variantu bez `const` výsledný strojový kód nijak neovlivňuje. Důvod, proč se překladač nesnaží optimalizovat konstantní proměnné je nejspíše fakt, že v C lze velmi jednoduše klíčové slovo odstranit jednoduchým přetypováním pointeru.

```
Point p = {1, 2, 3};
const Point* const_pointer = &p;
// error: assignment of member "x" in read-only object
const_pointer->x = 2;

Point* normal_pointer = (Point*) const_pointer;
normal_pointer->x = 2; // Validni
```

Je tedy možné, že by funkce `foo`, z původního příkladu, `p` modifikovala, ačkoliv je `p` předána jako konstantní.

6 Vektorizace

Vektorizace je proces přepsání programu takový, aby bylo možné několik jednotlivých procesorových instrukcí nahradit jednou SIMD(*Single instruction multiple data*) instrukcí. SIMD instrukce využívají takzvaných vektorových registrů. Kapitola čerpá ze zdrojů [4][1]

6.1 Vektorové registry

Vektorové registry se liší od obvyklých registrů svojí velikostí. Jsou trojího typu: 128 bitové, 256 bitové a 512 bitové.
V programu se využívají jen 128 bitové registry. Text od teď bude mluvit jen o nich.
Výhoda a účel takových registrů je, že uvnitř registru není uložena jedna proměnná, ale rovnou proměnných několik. Počet promměných, které můžeme uložit závisí na jejich datovém typu. Platí:

$$Počet\ buněk = \frac{Velikost\ vektorového\ registru(V\ našem\ případě\ 128\ bitů)}{Velikost\ datového\ typu}$$

Konkrétně tedy, lze uložit do jednoho 128 bitového vektorového registru:

- 16x `char`
- 8x `short`
- 4x `float`, `int`
- 2x `double`

6.2 SIMD

SIMD je skupina instrukcí, kde jednotlivé instrukce nahlíží na vektorové registry jako na vektory. Konceptuálně je tedy jeden vektor rozdělen na **N** disjunktních složek. Příklad:
Uvažujme dva vektorové registry `xmm0`, `xmm1`. V registrech chceme sčítat

floaty, jsou tedy k dispozici 4 složky.

$$\texttt{xmm0} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \texttt{xmm1} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Jsou-li tedy v registrech data, můžeme jednou instrukcí(na procesorech s podporou SEE instrukční sady konkrétně `addps`) sečíst vektory tak, že:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_0 + b_0 \\ a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$$

Pro kontrast při ne-vektorovém sčítání by bylo třeba provést instrukci sčítání 4x.

6.3 Příklad užití a dopad na výkon

Jak už bylo řečeno, SIMD je sada instrukcí, využití ve vysokoúrovňovém jazyce by tedy bylo možné pomocí *inline assembly*(Psaní assemblerovského kódu přímo do C kódu). Takový přístup ale nese řadu nevýhod. Elegantnější řešení je použití funkcí, které jednotlivé instrukce obalují, z hlavičkového souboru `immintrin.h`.Užití obalovacích funkcí a demonstrování zvýšení rychlosti je vidět na následujícím příkladu:
Příklad: Iterativní výpočet dominantního vlastního vektoru.
Tělo testovacích programů bude v obou případech stejné:

```
#define TEST_COUNT 1000000
void main() {
    Point3D result = {1, 1, 1};
    Transform3D t;
    t.rotation[0] = 0.5;
    t.rotation[1] = 0.3;
    t.rotation[2] = 0.2;
    t.rotation[3] = 0.4;
    t.rotation[4] = 0.2;
    t.rotation[5] = 0.3;
    t.rotation[6] = 0.3;
    t.rotation[7] = 0.3;
    t.rotation[8] = 0.4;
    for(int i = 0; i < TEST_COUNT; ++i) {
```

```
        rotate_point(t, &result);
    }
    /* Checking correct result + gcc would optimize out the
     * whole program if result wasn't processed in any way. */
    printf("%e %e %e\n", result.x, result.y, result.z);
}
```

`Transform3D` je struktura reprezentující matici, pro kterou se dominantní vlastní vektor hledá. `Point3D` je struktura reprezentující bod v $\mathbb{R}^3$.
Vektorově optimalizovaný program a neoptimalizovaný program se budou lišit právě v těchto strukturách a v implementaci funkce `rotate_point`.
Pro kontrast nejprve uvedeme neoptimalizovanou, lépe pochopitelnou verzi:

```
typedef struct {
    float x, y, z;
} Point3D;

typedef struct {
    float rotation[9];
} Transform3D;

inline void rotate_point(Transform3D t, Point3D* p) {
    float x = t.rotation[0] * p->x + \
              t.rotation[1] * p->y + \
              t.rotation[2] * p->z;
    float y = t.rotation[3] * p->x + \
              t.rotation[4] * p->y + \
              t.rotation[5] * p->z;
    float z = t.rotation[6] * p->x + \
              t.rotation[7] * p->y + \
              t.rotation[8] * p->z;
    p->x = x;
    p->y = y;
    p->z = z;
}
```

Jak z kódu vyplývá, implementací funkce `rotate_point` je klasické násobení matice a vektoru. Pro konstrukci verze využívající vektory je třeba si uvědomit následující skutečnost:

$$\begin{bmatrix} t_0 & t_1 & t_2 \\ t_3 & t_4 & t_5 \\ t_6 & t_7 & t_8 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} t_0 \cdot p_x + t_1 \cdot p_y + t_2 \cdot p_z \\ t_3 \cdot p_x + t_4 \cdot p_y + t_5 \cdot p_z \\ t_6 \cdot p_x + t_7 \cdot p_y + t_8 \cdot p_z \end{bmatrix} = \begin{bmatrix} t_0 \cdot p_x \\ t_3 \cdot p_x \\ t_6 \cdot p_x \end{bmatrix} + \begin{bmatrix} t_1 \cdot p_y \\ t_4 \cdot p_y \\ t_7 \cdot p_y \end{bmatrix} + \begin{bmatrix} t_2 \cdot p_z \\ t_5 \cdot p_z \\ t_8 \cdot p_z \end{bmatrix}$$

Problém tedy lze vektorově přepsat tak, že místo 9x násobení a 6x sčítání se

pouze 3x násobí a 3x sčítá. Přepsání pomocí funkcí z hlavičkového souboru `immintrin.h` tedy vypadá takto:
Přepis `Point3D`:

```
typedef union {
    struct {
        float x, y, z;
    };
    __m128 vector;
} Point3D;
```

`__m128` je datový typ, u kterého je garance, že bude při zpracování uložen do 128 bitového vektorového registru. Struktura byla změněna na `union` z důvodu pohodlného přístupu k jednotlivým složkám.
Přepis funkce `rotate_point`:

```
inline void rotate_point(Transform3D t, Point3D* p) {

    __m128 col0 = _mm_set_ps(0, t.rotation[6], \
                             t.rotation[3], \
                             t.rotation[0]);
    __m128 colp0 = _mm_set_ps1(p->x);
    __m128 res0 = _mm_mul_ps(col0, colp0);

    __m128 col1 = _mm_set_ps(0, t.rotation[7], \
                             t.rotation[4], \
                             t.rotation[1]);
    __m128 colp1 = _mm_set_ps1(p->y);
    __m128 res1 = _mm_mul_ps(col1, colp1);

    __m128 col2 = _mm_set_ps(0, t.rotation[8], \
                             t.rotation[5], \
                             t.rotation[2]);
    __m128 colp2 = _mm_set_ps1(p->z);
    __m128 res2 = _mm_mul_ps(col2, colp2);

    __m128 add01 = _mm_add_ps(res0, res1);
    p->vector = _mm_add_ps(res2, add01);
}
```

Jak je vidět, jsou využity funkce z hlavičkového souboru `immintrin.h`. Krátký jejich popis:

- `__m128 _mm_set_ps(float e3, float e2, float e1, float e0)`: Argumenty funkce postupně uloží do jednoho vektorového registru. (`e0`

jako *least significant byte*).

- `__m128 _mm_set_ps1(float e)`: Argument `e` je uložen do všech složek vektoru.
- `__m128 _mm_mul_ps(__m128 a, __m128 b)`: Násobení dvou vektorových registrů.
- `__m128 _mm_add_ps(__m128 a, __m128 b)`: Sčítání dvou vektorových registrů.

6.3.1 Analýza vygenerovaných strojových kódů

Pro plné pochopení síly vektorizace uvedeme strojový kód, který kompilátor vygeneruje:
Naivní implementace:

```
<main+4>    movss  0xfa3(%rip),%xmm9
<main+13>   pxor   %xmm6,%xmm6
<main+17>   movss  0xf9b(%rip),%xmm5
<main+25>   movss  0xf96(%rip),%xmm11
<main+34>   movss  0xf91(%rip),%xmm8
<main+43>   movss  0xf8d(%rip),%xmm7
<main+51>   movaps %xmm9,%xmm1
<main+55>   movaps %xmm9,%xmm2
<main+59>   movaps %xmm9,%xmm0
<main+63>   movss  0xf7c(%rip),%xmm10
<main+72>   nopl   0x0(%rax,%rax,1)
<main+80>   movaps %xmm1,%xmm12
<main+84>   movaps %xmm0,%xmm3
<main+87>   movaps %xmm2,%xmm4
<main+90>   mulss  %xmm5,%xmm12
<main+95>   addss  %xmm9,%xmm6
<main+100>  mulss  %xmm11,%xmm0
<main+105>  mulss  %xmm8,%xmm4
<main+110>  mulss  %xmm8,%xmm1
<main+115>  comiss %xmm6,%xmm10
<main+119>  addss  %xmm12,%xmm0
<main+124>  addss  %xmm4,%xmm0
<main+128>  movaps %xmm3,%xmm4
<main+131>  mulss  %xmm7,%xmm4
<main+135>  mulss  %xmm5,%xmm3
<main+139>  addss  %xmm1,%xmm4
```

```
<main+143>  movaps %xmm2,%xmm1
<main+146>  mulss  %xmm5,%xmm1
<main+150>  addss  %xmm12,%xmm3
<main+155>  mulss  %xmm7,%xmm2
<main+159>  addss  %xmm4,%xmm1
<main+163>  addss  %xmm3,%xmm2
<main+167>  ja     0x5555555550b0 <main+80>
```

Na první pohled je možné si všimnout, že se ve výstupu už objevují zmíněné vektorové registry `xmm`, čili se nabízí otázka, jestli kompilátor už sám naivní implementaci vektorově nezoptimalizoval. Skutečnost, že tomu tak není, můžeme ověřit například nástrojem **gdb**, pomocí kterého můžeme vypsat obsahy registrů. Konkrétně tedy ověříme například řádku s prefixem `<main+90>`, kde je první výskyt násobení. Po výpisu registrů zjistíme, že obsahy registrů jsou:

- `xmm5: 0x3e99999a` odpovídající hodnotě 0.3
- `xmm12: 0x3f800000` odpovídající hodnotě 1

Je tedy vidět, že v obou registrech, ačkoliv vektorových, je jen jeden `float`.

Chceme-li analyzovat hlavně kód samotného výpočtu a pomineme inicializaci proměnných, tak pár prvních instrukcí vynecháme a soustředíme se na řádky začínající řádkou s prefixem `<main+80>`. Napočítáme instrukce násobení(`mulss`) a instrukce sčítání(`addss`) a zjistíme, že výsledky přibližně **odpovídají předpokládanému teoretickému** odhadu 9 operací násobení a 6 operací sčítání.

Vektorová implementace:

```
<main+4>    movaps 0xfa5(%rip),%xmm0
<main+11>   movaps 0xfae(%rip),%xmm5
<main+18>   mov    $0xf423f,%eax
<main+23>   movaps 0xfb2(%rip),%xmm4
<main+30>   movaps 0xfbb(%rip),%xmm3
<main+37>   nopl   (%rax)
<main+40>   movaps %xmm0,%xmm1
<main+43>   movaps %xmm0,%xmm2
<main+46>   shufps $0x0,%xmm0,%xmm1
<main+50>   shufps $0x55,%xmm0,%xmm2
<main+54>   mulps  %xmm5,%xmm1
```

```
<main+57>    shufps $0xaa,%xmm0,%xmm0
<main+61>    mulps  %xmm4,%xmm2
<main+64>    mulps  %xmm3,%xmm0
<main+67>    addps  %xmm2,%xmm1
<main+70>    addps  %xmm1,%xmm0
<main+73>    sub    $0x1,%eax
<main+76>    jne    0x1088 <main+40>
```

Na první pohled je vidět, že celý vygenerovaný kód obsahuje podstatně méně instrukcí než předchozí naivní implementace. Opět budeme brát v potaz pouze efektivní výpočet, tedy instrukce začínající řádkou s prefixem `<main+40>`. Napočítáme-li instrukce násobení a součtu, zjistíme, že zase odpovídají teoretickému odhadu. Pro plné pochopení podrobná analýza:

```
<main+4>     movaps 0xfa5(%rip),%xmm0
<main+11>    movaps 0xfae(%rip),%xmm5
<main+18>    mov    $0xf423f,%eax
<main+23>    movaps 0xfb2(%rip),%xmm4
<main+30>    movaps 0xfbb(%rip),%xmm3
```

Tato část je inicializace proměnných, to je tedy incializace bodu a matice. Stav vektorových registrů po tomto kroku vypadá takto:

- `xmm0: 0x3f8000003f6666673f800000` $\rightarrow \{1; 0.9; 1\}$
- `xmm5: 0x3f0000003eccccd3e99999a` $\rightarrow \{0.5; 0.4; 0.3\}$
- `xmm4: 0x3e99999a3e4ccccd3e99999a` $\rightarrow \{0.3; 0.2; 0.3\}$
- `xmm3: 0x3e4ccccd3e99999a3eccccd` $\rightarrow \{0.2; 0.3; 0.4\}$

Kompilátor tedy uložil první sloupec matice do registru `xmm5`, druhý sloupec do registru `xmm4` a třetí sloupec do `xmm3`. `xmm0` obsahuje hodnotu závislou na počáteční hodnotě vektoru `p` a matice `t`. Je to totiž první iterace výpočtu(proč se tahle optimalizace děje je neznámé, ale gcc s optimalizační vlaječkou `-O3` vždycky předpočítá první iteraci).

```
<main+40>    movaps %xmm0,%xmm1
<main+43>    movaps %xmm0,%xmm2
<main+46>    shufps $0x0,%xmm0,%xmm1
<main+50>    shufps $0x55,%xmm0,%xmm2
```

Výkonný kód začíná těmito 4 instrukcemi, které počáteční data načítají do registrů nad kterými poté bude provedeno násobení. Instrukce `shufps` je překladačová optimalizace nahrazující přesun jednoho `floatu` do jednotlivých složek. Nejlépe vysvětleno na příkladu s prvním výskytem:

```
<main+46>    shufps $0x0,%xmm0,%xmm1
```

Vstupem instrukce jsou dva registry, ze kterých z obou budou vybrány 2 floaty. Tyto dvě skupiny budou poté uloženy do posledního operandu(v tom případě `xmm1`) tak, že skupina z `xmm1` bude uložena do spodní poloviny a skupina z `xmm0` do horní poloviny. To, jaké skupiny budou vybrány, určuje první operand (v příkladu `0x0`). Představíme-li si `xmm` registr jako floatové pole s indexy 0-3. První operand je potom maska, kde:

- 1. - 2. bit určují index floatu z posledního operandu(`xmm1`), který bude uložen na index 0 do posledního operandu(`xmm1`).
- 3. - 4. bit určují index floatu z posledního operandu(`xmm1`), který bude uložen na index 1 do posledního operandu(`xmm1`).
- 5. - 6. bit určují index floatu z 2. operandu(`xmm0`), který bude uložen na index 2 do posledního operandu(`xmm1`).
- 7. - 8. bit určují index floatu z 2. operandu(`xmm0`), který bude uložen na index 3 do posledního operandu(`xmm1`).

Po provedení instrukce z příkladu bude tedy výsledek:

$$\texttt{xmm1} = \{\texttt{xmm1[0]}, \texttt{xmm1[0]}, \texttt{xmm0[0]}, \texttt{xmm0[0]}\},$$

což kvůli tomu, že `xmm0` a `xmm1` jsou stejné(řádka s prefixem `<main+40>`), vyústí v to, že všechny složky v registru `xmm1` jsou tedy 1. složka z registru `xmm0`, čili 1. složka vstupního vektoru `p`. Další `shufps` probíhá analogicky.

```
<main+50>    shufps $0x55,%xmm0,%xmm2
```

jenom s jiným výsledným registrem a jinou maskou.

$$0\text{x}55 = 01010101 = 01\ 01\ 01\ 01$$

Ve všech složkách registru `xmm2` je tedy 2. složka vektoru `p`. Ob jednu instrukci následuje další `shufps`, po jejím provedení tedy vypadají registry takto:

- `xmm0: 0x3f8000003f8000003f8000003f800000` $\rightarrow \{1; 1; 1\}$
- `xmm1: 0x3f8000003f8000003f8000003f800000` $\rightarrow \{1; 1; 1\}$
- `xmm2: 0x3f6666673f6666673f6666673f666667` $\rightarrow \{0.9; 0.9; 0.9\}$

Nahrání všech proměnných tedy proběhlo podle očekávání a následuje výpočet.

```
<main+54>    mulps   %xmm5,%xmm1
<main+61>    mulps   %xmm4,%xmm2
<main+64>    mulps   %xmm3,%xmm0
```

Násobení také probíhá podle očekávání. První sloupec matice je vynásoben 1. složkou vektoru `p`. A to samé analogicky platí pro ostatní sloupce.

```
<main+67>    addps   %xmm2,%xmm1
<main+70>    addps   %xmm1,%xmm0
```

Posledním krokem je sečtení vektorových registrů. Výsledek celého maticového násobení je uložen v registru `xmm0`.
Na závěr výkonnostní porovnání pomocí nástroje `time`.

- Naivní implementace: 276ms
- Vektorová implementace: 111ms

Vektorovou implementací je tedy možno dosáhnout **více než dvojnásobného zrychlení**.

6.4 Využití načítacích instrukcí

Příklad z předchozí sekce ukázal urychlení oproti naivní implementaci. Nereprezentuje ale realistickou rychlost zpracování. Jak bylo vidno z analýzy strojového kódu, výpočet probíhal vně registrů a nikdy mimo registry hodnota nebyla ukládána. V aplikaci, ačkoliv matice zůstává stejná, se body mění a je třeba je ukládat. Realičtější kód můžeme dostat z aplikace:

```
<proximity_verifier_verify+144>  movss  (%rsi),%xmm0
<proximity_verifier_verify+148>  movss  0x4(%rsi),%xmm9
<proximity_verifier_verify+154>  shufps $0x0,%xmm0,%xmm0
<proximity_verifier_verify+158>  shufps $0x0,%xmm9,%xmm9
...
<proximity_verifier_verify+174>  movss  0x8(%rsi),%xmm9
<proximity_verifier_verify+180>  shufps $0x0,%xmm9,%xmm9
```

Poznámka - byla uvedena jenom sekce, pro kterou se strojový kód liší s uvedeným vektorovým příkladem.
Jak je vidět, vyskytuje se zde třikrát navíc instrukce `movss`. Ačkoliv je struktura v programu stejná jako vektor `p` z minulého příkladu, kompilátor přeloží program tak, aby postupně načetl složky, místo toho, aby načetl celý vektor najednou. Tohoto nežádoucího chování se můžeme zbavit použitím funkce `_mm_load_ps` jejímž parametrem je adresa začátku vektoru. Funkce s sebou ale nese poměrně velký požadavek. Citace z Intelovského průvodce:

> Load 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) from memory into dst. mem_addr must be **aligned on a 16-byte boundary** or a general-protection exception may be generated.

Je třeba tedy aby začátek vektoru byl umístěn na adrese dělitelné 16 bajty beze zbytku(poslední číslo v adrese je 0).

6.4.1 Lepší implementace matice

Kvůli této skutečnosti, se naskytuje možnost reprezentovat matici transformace tak, aby mohla být rychleji načtena. Místo 9 skalárních načítání by se načetla po sloupcích pomocí 3 vektorových operací. Implementace by tedy mohla vypadala nějak takto:

```
typedef union {
    float rotation[12];
    struct {
        float col0[4];
        float col1[4];
        float col2[4];
    };
} Transform3D;
```

Příklad prvních pár řádek z funkce `rotate_point`:

```
inline void rotate_point(Transform3D* t, Point3D* p) {
    __m128 col0 = _mm_load_ps(&(t->col0));
    __m128 colp0 = _mm_set_ps1(p->x);
    __m128 res0 = _mm_mul_ps(col0, colp0);
    /* Nacitani 2. a 3. sloupce probiha analogicky */
    ...
}
```

Tento přístup přináší několik nevýhod. Pole `rotation` není reprezentováno řádkově, ale sloupcově. Reprezentace matice je větší než být musí. Největší nevýhodou ale je třeba matici zarovnávat na 16 bajtů. Výhod tento přístup skoro nepřináší. Takováto reprezentace by se hodila v případě, kde máme jeden bod a velké množství matic. V programu se ale spíše vyskytuje případ opačný, kde pro jednu matici transformujeme velké množství bodů. Tento přístup tedy implementován nebyl.

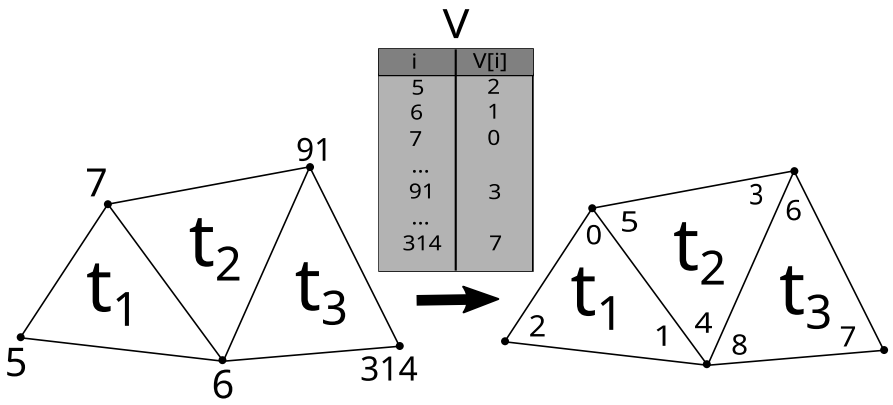
7 Použité datové struktury

Použití datových struktur je pro zvýšení výkonu absolutně klíčové, jelikož oproti naivní implementaci určitých operací poskytují lepší asymptotickou složitost.

7.1 CornerTable

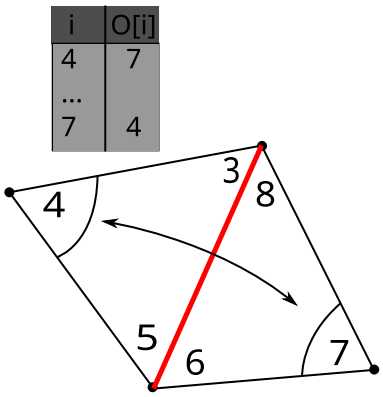
CornerTable, česky tabulka rohů(dále jen CornerTable), je datová struktura, jež umožňuje v konstatním čase hledat sousedy pro konkrétní vrchol [5]. Všechny operace, které tabulka poskytuje, operují pomocí dvou polí, v literatuře označované jako **O** a **V**.

- Pole **V** je mapování indexů vrcholů z trojúhelníkové sítě takové, že pro vrchol vrchol s indexem v_i je V[v_i] index jednoho z rohů vrcholu příslušících. Nejlépe vidět na obrázku 7.1.
- Pole **O**(**O**pposite) je indexováno pomocí indexů rohů. Pro roh s indexem i je v O[i] index jiného rohu, který se vyskytuje oproti rohu i přes společnou hranu, viz. obrázek 7.2.



Obrázek 7.1: Příklad a demonstrace pole **V**

Jak je z příkladů vidět, tabulka tedy místo na vrcholech operuje na rozích jednotlivých trojúhelníků (od této vlastnosti vzniklo její jméno).

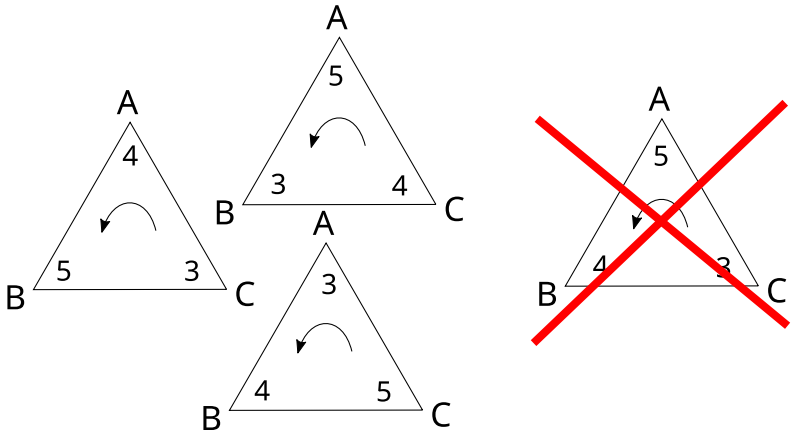


Obrázek 7.2: Příklad a demonstrace pole **O** a červené společné hrany

7.1.1 Konstrukce

Jak bylo řečeno v úvodu, tabulka je jasně definována dvěma poli. Konstrukce celé tabulky je tedy úloha složená z dvou podúloh - tvoření pole **V** a pole **O**.

Konstrukce pole **V** probíhá velmi jednoduše, iteruje se nad všemi trojúhelníky zpracovávané trojúhelníkové sítě $t_1 \dots t_n$. Indexy rohů trojúhelníku t_i jsou závislé na indexu i tak, že trojúhelník t_i má rohy $\{3i; 3i+1; 3i+2\}$. Pro jeden libovolný roh lze zvolit jakýkoliv index ze zmíněné množiny, ostatní rohy jsou na této volbě závislé a musí se podmínit orientací trojúhelníku. viz. obrázek 7.3

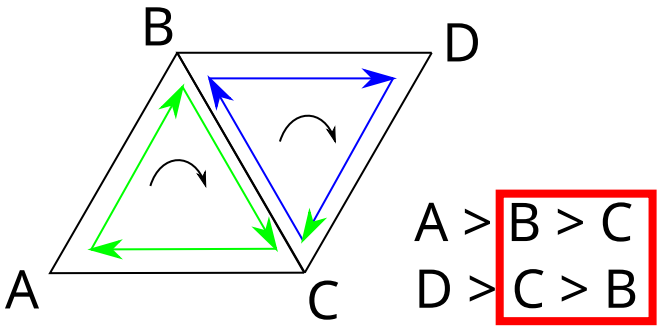


Obrázek 7.3: Příklad možných indexování rohů

Pole **O** je zkonstruováno pomocí dodatečné datové struktury - slovníku - v textu označeno dále **D**(**D**ictionary) sloužícího jako asociativní pole mezi hranou a rohem. Jak je vidět z obrázku v manifoldní trojúhelníkové síti(viz.

obrázek 7.2) pro jednu hranu existují právě dva rohy jí příslušící. Opět tedy iterujeme přes všechny trojúhelníky. Pro každý roh s indexem j, v trojúhelníku je determinována jemu příslušná hrana e_j. Další krok je prohledání **D** pro hranu e_j. Můžou nastav dva stavy:

- Hrana se již ve slovníku nachází: Známe tedy index k rohu ležící na opačné straně hrany e. Do pole **O** tedy uvedeme záznam o obou rozích, tedy $O[k] = j$ a $O[j] = k$.
- Hrana se ve slovníku nenachází: Do slovníku přidáme pro **opačně orientovanou** hranu ${e_j}^{-1}$ index rohu, tedy $\mathbf{D}({e_j}^{-1}) = j$. Nutnost přidání opačně orientované strany je způsobena orientací trojúhelníků, viz. obrázek 7.4



Obrázek 7.4: Orientace sousedních trojúhelníků

7.1.2 Hledání sousedů

Jak bylo řečeno v úvodu, hlavní sílou CornerTable je fakt, že pro vrchol v_i dokáže najít sousedy v konstatním čase. Pro vysvětlení principu je nutno definovat dva pojmy týkající se indexů rohů v trojúhelníku. Mějme tedy 3 rohy v jednom trojúhelníku s indexy i, $i+1$ a $i+2$. Definujeme tři operace **n**, **p** a **v**

- **n**(**n**ext): následující roh dle orientace. Platí:

  $\mathbf{n}(i) = i + 1$

  $\mathbf{n}(i + 1) = i + 2$

  $\mathbf{n}(i + 2) = i$

- **p**(**p**revious): předchozí roh dle orientace. Platí:

  $\mathbf{p}(i) = i + 2$

$$\mathbf{p}(i+1) = i$$
$$\mathbf{p}(i+2) = i+1$$

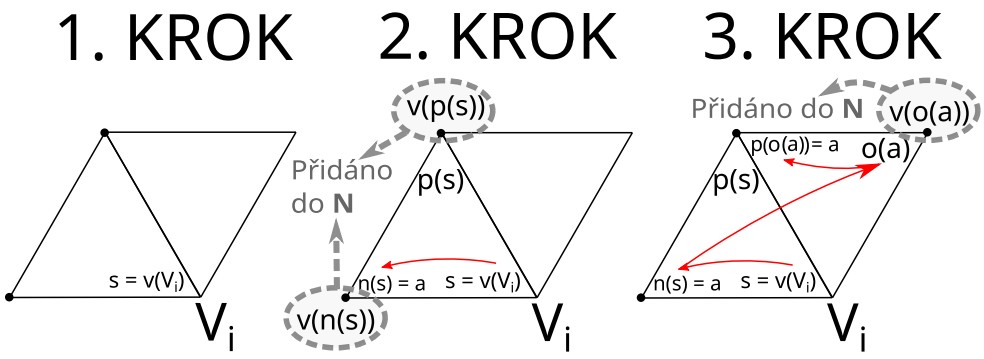
- $\mathbf{v}$(**v**ertex): Pro index rohu vrací index vrcholu trojúhelníkové sítě. Víme, že trojúhelník s indexem j má rohy $\{3j; 3j+1; 3j+2\}$. Zjistíme tedy jednoduše, kterému trojúhelníku roh náleží vydělením $j = floor(\frac{i}{3})$. Zbytek po dělení 3 poté dohromady se zvoleným indexování(stanoveno v konstrukci v sekci 7.1.1) udává index vrcholu uvnitř trojúhelníku j.

Pomocí polí **O**, **V** a operací **n**, **p** a **v** hledáme sousedy **N** vrcholu V_i. Uvedeme nejdřív základní myšlenku algoritmu, který funguje pro ideální případ:

1. Pro vrchol V_i pomocí pole **V** získáme jeden z rohů. Tento roh označíme s a prohlásíme za počáteční.

2. Do **N** přidáme předchozímu a dalšímu rohu odpovídající vrcholy - $\mathbf{v}(\mathbf{p}(s))$ a $\mathbf{v}(\mathbf{n}(s))$. Roh $\mathbf{n}(s)$ označíme a a pomocí něj budeme iterovat okolo počátku s.

3. Přejdeme na roh $\mathbf{o}(a)$ je-li $\mathbf{n}(a)$ rovno s znamená to, že se opět nacházíme ve startovacím trojúhelníku a iterování můžeme ukončit. Není-li tomu tak, přidáme do **N** tento vrchol $\mathbf{v}(a)$, položíme $a = \mathbf{p}(a)$ a opakujeme krok 3.



Obrázek 7.5: Demonstrace fungování ideálního případu CornerTable

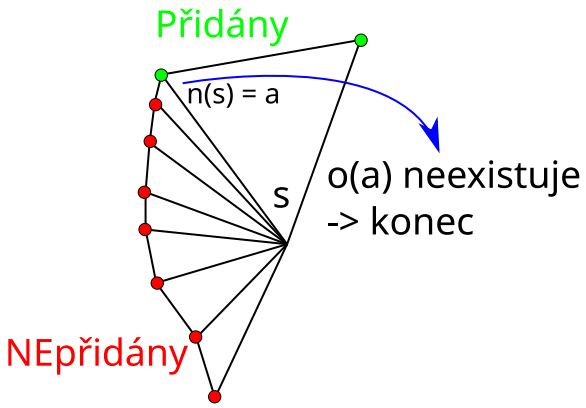
Povšimněte si znovu 2 skutečností.

1. Vlivem orientace trojúhelníku se prvně ze středu vychází pomocí dalšího rohu $\mathbf{n}(s)$, poté ale ve všech dalších iteracích se prochází pomocí předchozích rohů $\mathbf{p}(\mathbf{o}(a))$

2. Do **N** se přidávají vrcholy náležící oběma sousedním rohům jen v první iteraci/druhém kroku. V každé další iteraci se přidává vrchol jen nového rohu $\mathbf{o}(a)$, jelikož, jak je možné si povšimnout z obrázku 7.5 vrchol příslušící bodu $\mathbf{p}(\mathbf{o}(a))$ je vždy už zahrnut z předchozí iterace.

Případné problémy

Problémy takové implementace vznikají v moment, kdy vrchol V_i leží na okraji trojúhelníkové sítě. Tím, že prohledáváme jenom jedním směrem(buďto ve směru nebo proti směru hodinových ručiček) se může stát, že v poli **O** nebude záznam o protějším rohu. Hledání ukončíme, ale výsledkem bude jenom podmnožina všech sousedů, viz. obrázek 7.6. Řešení tohoto problému



Obrázek 7.6: Černý scénář prohledávání jen jedním směrem

odhaluje i finální implementaci hledání sousedů. Algoritmus zůstává, zjistí-li se ale, že iterování skončilo neúspěšným pokusem o nalezení protějšího rohu, začne se opět od začátku s, tentokrát ale v opačném směru. Zkráceně a pomocí značek tedy popis finálního algoritmu:

$$s \to \mathbf{n}(s) = a \to \mathbf{o}(a) \to \mathbf{p}(a) = a \to \mathbf{o}(a) \to \mathbf{p}(a) = a \to \dots$$

Po skončení cyklus když $\mathbf{o}(a)$ neexistuje (je rovno -1) tak:

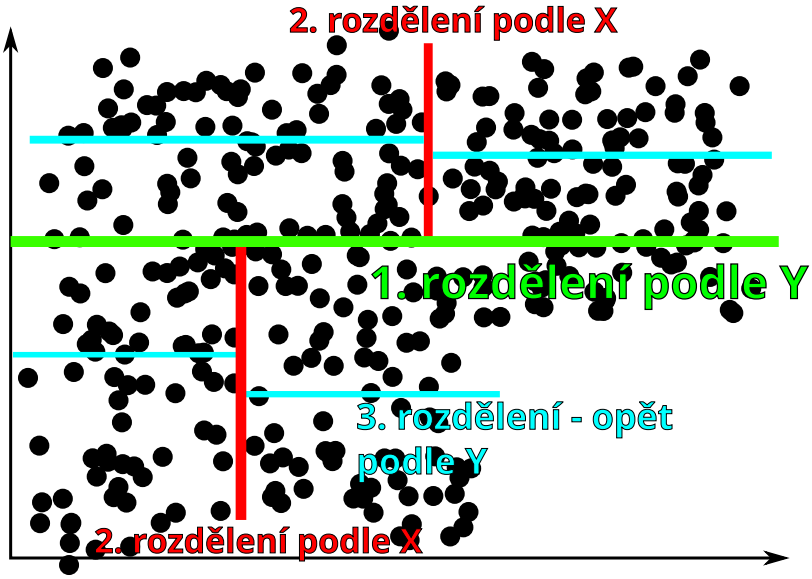
$$s \to \mathbf{p}(s) = a \to \mathbf{o}(a) \to \mathbf{n}(a) = a \to \mathbf{o}(a) \to \mathbf{n}(a) = a \to \dots$$

7.2 KD tree

KD tree je speciálním případem binárního stromu. Značně usnadňuje hledání blízkých bodů. Myšlenka spočívá v tom, že známe-li souřadnice bodu, můžeme eliminovat část prostoru, kde vstupnímu bodu bod nejbližší nebude

Obrázek 7.7: Myšlenka za KD tree pro body v $\mathbb{R}^2$

ležet. Otázka spočívá v tom, podle jaké souřadnice prostor dělit, odpověď je elegantní a to taková, že KD tree dělí oblasti podle složek postupně. Je-li tedy prvně rozdělen podle 1.složky(například pro body v $\mathbb{R}^2$ složka y-ová), dvě vzniklé oblasti potom budou děleny dál podle složky další(x-ové). Poté, co dojdou složky se opět pokračuje od začátku.



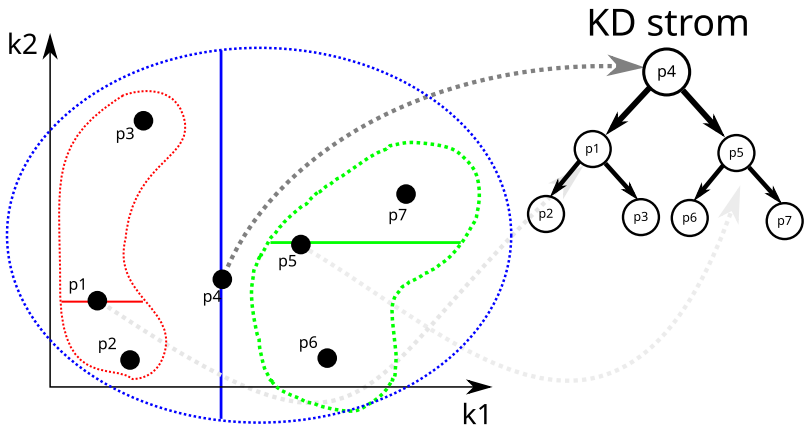
Obrázek 7.8: Příklad dělení $\mathbb{R}^2$ prostoru

V programu se KD strom používá na hledání bodů s podobnými křivostmi. Zbytek textu o KD stromu se tedy bude zabývat 2D variantou, se dvěma složkami k_1 a k_2.

7.2.1 Konstrukce

Vstupem je pole bodů, $p_1, p_2 \dots p_n$, kde každý bod má dvě složky, $p_i = [p_{ik1}, p_{ik2}]^T$. Konstrukce probíhá rekurzivně s tím, že se podle hloubky stromu mění složka bodu, podle které se body porovnávají. V prvním zavolání rekurzivní funkce se tedy body porovnávají podle složky k_1, v další k_2 a ve 3. zase k_1. Pro přehlednost pojmenujeme porovnávanou složku odpovídající hloubce uzlu k_m. Rekurzivní funkce provádí:

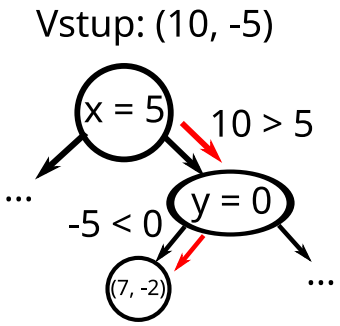
- Cílem je vždy rozdělit prostor na poloviny. Je tedy nalezen bod, jež je podle složky k_m mediánem.
- Po nalezení mediánu je medián z pole vyjmut a jsou zkonstruovány 2 pole. Pole prvků, jejichž k_m jsou menší než k_m mediánu a pole prvků, jejichž k_m jsou větší. Tyto dvě pole jsou poté vstupem do dalších iterací rekurze.



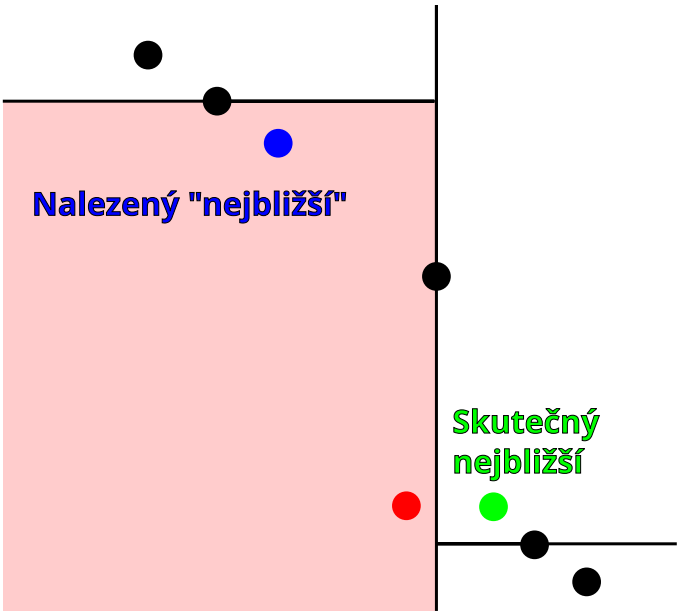
Obrázek 7.9: Příklad konstrukce KD tree

7.2.2 Prohledávání KD stromu

Prohledávání opět funguje rekurzivně. Mějme bod q pro který hledáme nejbližší bod p_i z pole bodů. V každém uzlu u s bodem u_p porovnáváme bod q s u_p podle složky k_m. Další prohledávaná větev je určena skupinou, do které bod q oproti bodu u_p patří. Pro nalezení nejbližšího souseda ale nestačí procházet pouze větev, do které by bod býval patřil. Může totiž nastat

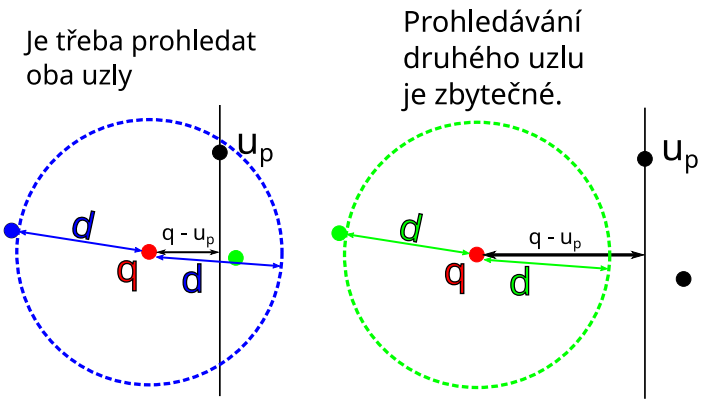


Obrázek 7.10: Myšlenka za průchodem KD tree



Obrázek 7.11: Černý scénář pro naivní průchod

situace ilustrována na obrázku 7.12. Je tedy nutné zajistit, aby v případě, že je to nutné byla prohledána i druhá větev. To je zajištěno tím, že je vždy pro každý nalezený bod spočtena euklidovská vzdálenost od vstupního bodu. Po vrácení se z jednoho z uzlů je prohledávání větve druhé podmíněno podmínkou $(q - u_p) < d$. Rozdíl $q - u_p$ je rozdíl složek k_m těchto dvou bodů a d je euklidovská vzdálenost nejbližšího dosavadně nalezeného bodu a vstupního bodu q. Druhý uzel je prozkoumáván, je-li možné, že v prostoru příslušící druhému uzlu leží bod, který je blíž bodu q.

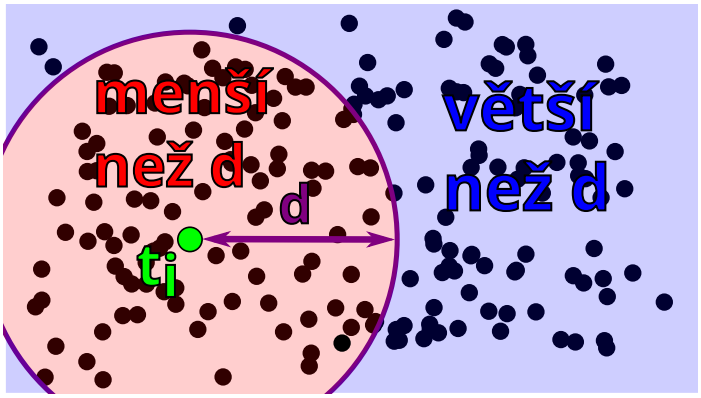


Obrázek 7.12: Příklad nutnosti prohledávání obou uzlů

7.3 Vantage point tree

Vantage point tree je velmi podobná struktura KD tree. Stejně jako KD strom je to binární strom a poskytuje efektivní způsob nalezení blízkých prvků. V programu se používá pro hledání blízkých transformací.

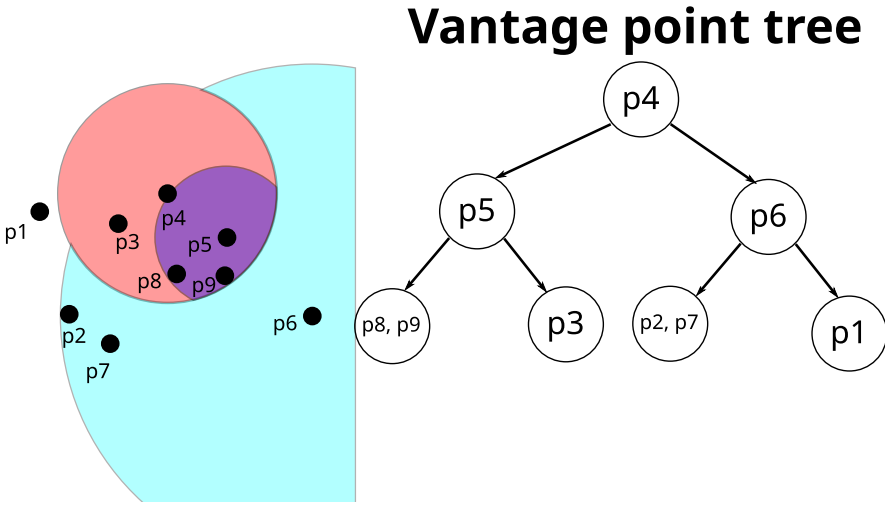
Motivace pro využití struktury, která poskytuje stejnou fukcionalitu jako KD strom je taková, že prostor transformací je těžce dělitelný podle složek. Můžeme ale prostor dělit na základě vzdáleností, jelikož dokážeme spočítat vzdálenost dvou transformací. Pro libovolnou transformaci v prostoru transformací t_i jsme tedy schopni spočítat vzdálenosti všech ostatních transformací. Můžeme tedy prostor rozdělit na transformace bližší než d a transformace vzdálenější než d, kde d je medián vzdáleností od t_i.



Obrázek 7.13: Rozdělení prostoru podle vzdáleností

7.4 Konstrukce

Konstrukce probíhá velmi podobně jako konstrukce KD tree. Vstupem je tedy pole transformací $t_1, t_2 \dots t_n$. Konstrukce probíhá rekurzivně tak, že je z pole náhodně zvolena jedna transformace t_i (vantage point). Pro tu jsou spočítány vzdálenosti všech ostatních transformací. Z těchto vzdáleností je určen medián m. Ostatní transformace jsou poté roztříděny do dvou skupin(dvou polí). Transformace se vzdáleností od t_i větší než d a transformace se vzdáleností od t_i menší než m. Nad těmito skupinami je poté funkce rekurzivně zavolána znovu.



Obrázek 7.14: Příklad konstrukce Vantage point tree

7.5 Prohledávání

Na rozdíl od prohledávání KD stromu nehledáme jednoho nejbližšího souseda, ale všechny sousedy ve stanoveném rádiusu od transformace. Označíme-li transformaci, pro kterou hledáme sousedy t_i, transformaci t_u, jíž odpovídá momentálně prohledávaný uzel u, medián vzdáleností m od transformace t_u a rozsah, ve kterém hledáme r.

Pro uzel prohledáváme uzel s prvky bližšími než m v případě, že platí $d(t_i, t_u) \leq m + r$. Uzel s prvky vzdálenějšími než m v případě, že platí $d(t_i, t_u) \geq m - r$. Navíc v každém uzlu u, zda-li platí podmínka $d(t_i, t_u) \leq r$, přidáme transformaci t_u do pole hledaných sousedů.

8 Testování a dosažené výsledky

Testování probíhalo na sadě dvojic trojúhelníkových sítí. Pro každou dvojici platí, že sítě již jsou v pozicích výsledné registrace. Očekávanou výslednou rotací je v tomto případě jednotková matice a nulová translace. Volba takových dat nijak nezlehčuje průběh algoritmu, který zejména kvůli počáteční dekorelaci bodů musí projít stejným množstvím kroků jako pro libovolnou startovní pozici.

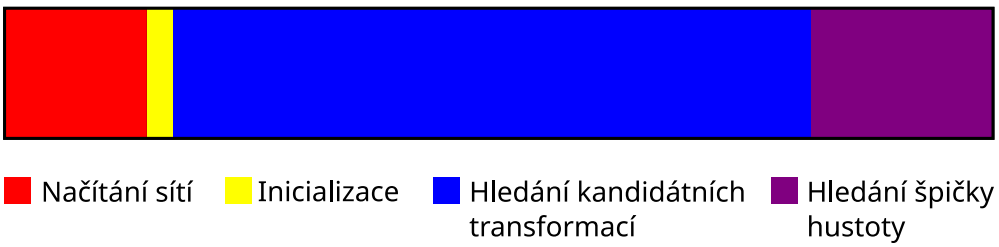
Testování výkonnostního zlepšení bylo oproti referenčnímu programu. Pomocí nástroje `time` byla 10x změřena doba průběhu programu pro jednotlivé dvojice, z těchto 10 vzorků byl vypočten průměr a zjištěny následující data:

	Nový registrátor	původní registrátor	Poměr
arm1.obj, arm2m.obj	505ms	1595ms	3.16
bub1.obj, bub2m.obj	170ms	876ms	4.97
bud1.obj, bud2m.obj	742ms	1902ms	2.56
dra1.obj, dra2m.obj	544ms	1522ms	2.79
hip1.obj, hip2m.obj	334ms	1030ms	3.08
kac1.obj, kac2m.obj	373ms	1011ms	2.71

diverzita výsledků je způsobena jinou strategií získávání kandidátních transformací a rychlostí předzpracování sítě. Tento přístup se odvědčil zejména pro malé sítě(viz. dvojice bub1.obj, bub2m.obj).

8.0.1 Potenciální vylepšení

Časové rozložení jednotlivých částí je vidět na diagramu 8.1. V momen-



Obrázek 8.1: Procentuální časové rozložení částí

tální verzi programu je velký *bottleneck* hledání sousedů kolem vrcholu za pomocí CornerTable. Vylepšení této části by značně vylepšilo výkon celého programu. Myšlenka možného vylepšení procházení je taková, že místo užívání CornerTable pro získání bezprostředních sousedů a využití zásobníku, by se využily její vlastnosti sofistikovaněji pro získání hned celé množiny sousedů(řádově myšlenku ukazuje obrázek na straně 4 v publikaci [5]).

Dalším možným zrychlením by byla změna načítání sítí z .obj souborů. Momentální přístup, ačkoliv programátorsky pohodlný a přehledný, se jeví jako pomalý oproti alternativně zpracovávání souboru proudově za pomocí konečného stavového automatu.

8.0.2 Potenciální problémy

Při testování programu byl zjištěny problémy, které nebyly adresovány, jelikož pro momentální sadu testů problémy nepůsobil.

1. Problémem je občasná nepřesnost při verifikování transformací. Jelikož je jedním z kroků verifikace násobení převrácenou hodnotou kroku, tak vlivem nepřesnosti operací s floaty se některé body mapují o buňku špatně. Řešením by mohlo být kontrolování okolí ϵ kolem bodu.

2. Problémem je potenciální memory leak způsoben nevyužitím mutexu ve funkci `neighbours_cache_get`. Program byl otestován nástrojem `valgrind` a nebyl zjištěn žádný únik paměti. Je teoreticky ale možné, že 2 vlákna přistoupí ve stejný čas do funkce se stejným parametrem `index`. V ten moment bude jeden z pointerů ztracen a vznikne únik paměti.

9 Závěr

Nový program běží průměrně **přibližně třikrát rychleji** než program původní. Žádný předpoklad o teoreticky možném zrychlení před začátkem vývoje nebyl, čili je trojnásobné zrychlení považováno za dostatečně dobrý výsledek pro prohlášení práce za úspěšnou.

Testování proběhlo oproti původnímu programu s kterým program sdílí řádově stejné výsledky(výsledky se liší minimálně kvůli náhodnému vzorkování dat).

Program je nepochybně možné ještě zrychlit, některé myšlenky o potenciální vylepšeních jsou uvedeny v sekci 8.0.1.

Literatura

[1] Drepper, U. What Every Programmer Should Know About Memory. November 2007. Dostupné z: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[2] Fog, A. *Optimizing software in C++* [online]. [cit. 4/5/22]. Dostupné z: https://agner.org/optimize/optimizing_cpp.pdf.

[3] Hruda, L. – Dvořák, J. – Váša, L. On evaluating consensus in RANSAC surface registration. August 2019. doi: 10.1111/cgf.13798. Dostupné z: https://doi.org/10.1111/cgf.13798.

[4] *Intel® Intrinsics Guide* [online]. Intel. [cit. 4/5/22]. Dostupné z: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[5] Jarek, R. – Alla, S. – Andrzej, S. Edgebreaker on a Corner Table: A Simple Technique for Representing and Compressing Triangulated Surfaces. doi: 10.1007/978-3-642-55787-3_3. Dostupné z: http://www.cs.cmu.edu/~alla/Rossignac.pdf.

[6] Mario, B. et al. *Polygon Mesh Processing*. 2010.

[7] Pereira, R. et al. Energy Efficiency across Programming Languages. doi: 10.1145/3136014.3136031. Dostupné z: https://dl.acm.org/doi/10.1145/3136014.3136031.

[8] Rusinkiewicz, S. Estimating Curvatures and Their Derivatives on Triangle Meshes. In *Symposium on 3D Data Processing, Visualization, and Transmission*, September 2004.