VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020

Martin Regec

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

OPTIMALIZACE SPOJOVÁNÍ DÍLČÍCH 3D MODELŮ POVRCHU Z HLEDISKA VÝPOČETNÍ DOBY
OPTIMIZATION OF 3D SURFACE MODEL MERGING FROM COMPUTATIONAL TIME PERSPECTIVE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE AUTHOR	**Martin Regec**
VEDOUCÍ PRÁCE SUPERVISOR	**Ing. Adam Chromý, Ph.D.**

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Martin Regec ***ID:*** 197779
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Optimalizace spojování dílčích 3D modelů povrchu z hlediska výpočetní doby

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je vylepšit a zrychlit stávající algoritmy pro sesazování několika dílčích skenů do jediného celku, a to na základě znalosti navzájem korespondujících bodů nebo na základě znalosti přesných absolutních souřadnic. Výsledky práce budou nasazeny v systému RoScan určeném pro medicínské skenování lidského těla.

Zadání:

1. Seznamte se s robotickým multispektrálním systémem RoScan, jeho stávající implementací a výsledky předchozí studentské práce na toto téma.
2. Doplňte předchozí studentskou práci tak, aby byla schopna spolehlivě realizovat spojování několika dílčích skenů.
3. Navrhněte optimalizaci současné implementaci tak, abyste dosáhli nižší výpočetní náročnosti. Navrhněte, jak by se k tomuto účelu dala využít grafická karta (GPGPU).
4. Implementujte navržené řešení v systému RoScan.
5. Realizujte experimenty, které prokáží vylepšené fungování modulu a jeho nižší výpočetní dobu.

DOPORUČENÁ LITERATURA:

Kim, H., Vuduc, R., Baghsorkhi, S., Choi, J., 2012. Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). Morgan & Claypool Publishers.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: Ing. Adam Chromý, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Práca sa zaoberá spájaním čiastkových 3D modelov povrchu, ktoré sa v určitej ploche prekrývajú, do výsledného celistvého modelu. Cieľom práce je vytvorenie dostatočne rýchleho a spoľahlivého algoritmu na spájanie 3D čiastkových modelov, ktorý by sa mohol stať súčasťou aplikácie RoScan Analyzer. Úvodná časť praktickej časti je venovaná krátkemu priblíženiu problémov práce v ktorej táto práca pokračuje, následne je navrhnuté riešenie a implementácia týchto problémov. Nakoniec sa overí, či je výsledok algoritmu správny. Implementácia je napísaná v štandarde Microsoft .NET.

KĽÚČOVÉ SLOVÁ

Spájanie čiastkových 3D modelov, prekrývajúce sa modely, triangulácia povrchu modelu, pretrojuholníkovanie povrchu, triangulácia medzery, GPGPU, paralelné programovanie.

ABSTRACT

The point of the work is connecting of partial 3D models of surface, which are overlapped in a certain area to a final whole model. The intention of the work is creation of algorithm that is fast enough and reliable enough to connecting partial 3D models and that can become a part of application RoScan Analyzer. In the introduction of the practical part of the work, we sum up problems of the work that we are keeping on. In the next part, the solution and implementation are suggested. In the last part, the outcome of the algorithm is veryfied. Implementation is written in Microsoft .NET Standard.

KEYWORDS

Connecting partial 3D models, overlapping models, model surface triangulation, surface retriangulation, gap triangulation, GPGPU, parallel programming.

REGEC, Martin. *Optimalizace spojování dílčích 3D modelů povrchu z hlediska výpočetní doby*. Brno, 2020, 65 s. Bakalárska práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedúci práce: Ing. Adam Chromý, Ph.D.

VYHLÁSENIE

Vyhlasujem, že som svoju bakalársku prácu na tému „Optimalizace spojování dílčích 3D modelů povrchu z hlediska výpočetní doby" vypracoval samostatne pod vedením vedúceho bakalárskej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej bakalárskej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno 7. 6. 2020 ....

podpis autora

POĎAKOVANIE

Chcel by som sa poďakovať vedúcemu bakalárskej práce Ing. Adamovi Chromému Ph.D. za cenné rady, ochotu a ústretový prístup. Veľké poďakovanie patrí aj rodičom za podpotu počas celého štúdia.

Brno 7. 6. 2020 ....

podpis autora

Obsah

Úvod **11**

1 Trojrozmerné modelovanie **13**

1.1 Proces vyhotovovania modelov 13

1.1.1 3D modely vytvorené Robotickým 3D skenerom 14

2 Spájanie modelov **16**

2.1 Nájdenie prekrytia 16

2.1.1 Mriežka boxov 16

2.1.2 K-d Strom 17

2.1.3 R-stromy 19

2.1.4 Priesečník lúča s trojuholníkom 21

2.2 Výsledný model 24

3 GPGPU **26**

3.1 OpenCL 26

3.1.1 Model Platformy 26

3.1.2 Výpočetný Model 27

3.1.3 Pamäťový Model 28

3.1.4 Programový Model 29

3.2 CUDA 29

4 Analýza východiskovej práce **31**

5 Návrh a implementácia nového riešenia **34**

5.1 Nájdenie prekrytia modelov 34

5.1.1 Nájdenie bodov a trojuholníkov v spoločnej oblasti modelov . 34

5.1.2 Rozdelenie spoločnej oblasti do mriežky boxov 36

5.1.3 Hľadanie priesešníku lúča a trojuholníkov 37

5.2 Kontrola správnosti nájdených bodov prekrytia 40

5.3 Spriemerovanie bodov v prekrytí 41

5.4 Vytvorenie nových trojuholníkov v prekrytí 41

5.5 Nájdenie bodov na vzniknutej medzere 44

5.6 Rozšírenie medzery 46

5.7 Vyplnenie medzery novými trojuholníkmi 47

5.8 Vymazanie nepoužitých bodov z modelu. 48

6 Prepracovanie algoritmu pre použie na GPU **49**
6.1 Návrh paralelného výpočtu 50
6.2 Použitie GPU 52

7 Experimentálne overenie **55**
7.1 Porovnanie s východiskovou prácou 55
7.2 Porovnanie rýchlostí počítania na CPU s GPU 58

8 Záver **60**

Literatúra **61**

Zoznam symbolov, veličín a skratiek **65**

Zoznam obrázkov

1.1 Základná myšlienka prevádzky Robotického 3D skeneru. 13
1.2 Prehľad spracovania Robotického 3D skeneru[13]. 14
2.1 Ukážka binárneho stromu[32]. 18
2.2 Päť rôznych binárnych stromov na troch uzloch[32]. 18
2.3 Vizualizácia R-stromu pre 3D body použitím ELKI softvéru[31, 6]. . . 20
2.4 Barycentrické súradnice sa dajú chápať ako suboblasť subtrojuholníkov CAP (pre u), ABP (pre v) a BCP (pre w) nad oblasťou trojuholníka ABC[4]. 22
2.5 Preklad a zmena počiatku lúča[23]. 23
2.6 Odstránenie prekryvajúcej plochy a nájdenie okrajových bodov[19]. . 24
2.7 Rekurzívne vyplnenie medzery novými trojuholníkmi[19]. 25
3.1 Platformový model[24]. 27
3.2 Výpočetný model[24]. 28
3.3 Pamäťový model[24]. 29
4.1 Výsledné spojené modely pospájané algoritmom z bakalárskej práce Mareka Lampáša[21]. 31
4.2 Nevhodné priemerovanie bodov pri spájaní čiastkových modelov[21]. . 32
4.3 Nesprávne pretrojuholníkovanie bodov v prekrytí. 33
5.1 Ukážka vytvorenia Bounding boxov pre modely, ktoré nie sú rovnobežné s osami. Bounding box je uznačený červenou farbou. 35
5.2 Ukážka princípu vytvorenia čiastkových bounding boxov. 36
5.3 Normálové vektory trojuholníkov a lúče[21]. 37
5.4 Lúč ktorý pretína 2D bounding box. 39
5.5 Lúče ktoré nepretínajú 2D bounding box. 40
5.6 Ukážka pretnutia lúča s trojuholníkom ktorý sa nenachádza v prekrytí . 41
5.7 Ukážka bodov prekrytia pred priemerovaním bodov[21]. 42
5.8 Ukážka bodov prekrytia po priemerovaní bodov referenčného modelu [21]. 42
5.9 Plochy pred pretrojuholníkovaním[21]. 43
5.10 Pretrojuholníkovaný povrch[21]. 43
5.11 Ukážka obrátenia normály so zmenou rotácie trojuholníku. 44
5.12 Ukážka princípu nájdenia medzery. 45
5.13 Detail na zúženú medzeru. 46
5.14 Detail rozšírenej a nerozšírenej medzery. 47
5.15 Výber nového trojuholníku v medzere. 47
5.16 Druhý krok výberu nového trojuholníku v medzere. 48
6.1 Sériové počítanie[7]. 49

6.2 Paralelné počítanie[7]. 50
6.3 Algoritmus pre výpočet najbližšieho priesečníku n lúčov s k trojuholníkmi. 51
7.1 Výsledne objekty. 55
7.2 Detail na medzeru. 56
7.3 Detail na úsek modelu, kde je vidieť zlepšenie algoritmu priemerovania bodov. 56
7.4 Detail na úsek modelu, kde je vidieť opravu plôch nad modelom. . . . 56

Zoznam tabuliek

7.1 Porovnanie času 57
7.2 Porovnanie času výpočtu na CPU a GPU 58

Úvod

RoScan je systém na skenovanie ľudského tela, ktorý behom niekoľkých desiatok sekúnd vie spraviť veľmi presný trojrozmerný počítačový model[33]. Každý z vytvorených modelov má niekoľko vrstiev a to farbu, teplotu, samotný povrch a hrubosť. Farba je vrstva, ktorá zobrazuje telo tak ako ho vidíme, je vhodná na meranie rozmerov (napr. opuchov, jaziev, popálenín, atď.). Teplota je vrstva, ktorá farebne znázorní teplotu ľudského tela, sú na nej vidieť nekrotické tkanivá, opuchy, či centrá infekcie. Samotný povrch je vrstva, ktorá slúži na detekciu dôležitých anomáli, ktoré sa môžu skrývať napr. pod farbou povrchu. Vďaka rozlíšeniu pod 0.2 milimentra sú na modeli vidieť aj tie najmenšie nerovnosti. Hrubosť je vrstva, ktorá farebne znázorní, či je povrch lesklý, matný, alebo hrubý[5].

Princíp tohto systému je založený na senzorickej hlave, ktorá sa pomocou robotického manipulátora pohybuje okolo skenovaného objektu po predom definovanej trajektórii[33]. Problém nastáva, keď treba naskenovať povrch, ktorý je väčší ako rozsah robotického manipulátora. Vtedy je potrebné naskenovať niekoľko dielčích modelov a programovo ich spojiť.

Tejto problematike sa venoval už aj môj predchodca Marek Lampáš v práci s názvom "Spojování dílčích 3D modelů povrchu do celkového modelu", no implementácia v tejto práci na hľadanie a pretrojuholníkovanie prekrytia nefungovala spoľahlivo a algoritmus na spojenie jednoduchých modelov trval desiatky minút. Cieľom tejto práce bude tieto nedostatky opraviť a vytvoriť tak dostatočne spolahlivý a rýchly program na spájanie 3D dielčích modelov, ktorý by sa mohol stať súčasťou aplikácie RoScan Analyzer.

V prvej kapitole je najskôr opísaný princíp a použitie 3D modelovania, ďalej sa priblíži funkčnosť a výstup RoScan 3D skeneru, s ktorým sú použité modely v tejto práci vytvorené.

V druhej kapitole je opísaný princíp spájania prekrývajúcich sa modelov, kde sme sa snažili nájsť čo najvhodnejšie riešenie detekcie prekrytia dvoch modelov s čo najväčšou minimalizáciou času vypočítania prekrytia. Podmienkou bolo, z výstupu nájdeného prekrytia vedieť spriemerovať plochy čiastkových modelov.

V tretej kapitole je priblížený pojem GPGPU a opísané dva najčastejšie používané štandardy.

Štvrtá kapitola je venovaná analýze stavu východiskovej práce, v ktorej sa hľadajú nedokonalosti, ktoré sme sa v tejto práci snažili čo najlepšie opraviť.

V piatej kapitole je popísaný návrh a implementácia riešenia nájdených problémov vo východiskovej práci, kde sme sa venovali algoritmom z východiskovej práce, ktoré fungujú dobre a ktoré sú naopak nefunkčné. K nefunkčným algoritmom je vytvyrený nový návrh a implementácia.

V šiestej kapitole si najskôr priblížime pojem paralélne počítanie, následne je ukázané akým spôsobom sme algoritmus na výpočet prekrytia dvoch modelov modifikovali tak, aby sa vypočítal paralélne s použitím grafickej karty.

Posledná kapitola sa venuje porovnaniu nášho výsledku s výsledkom východiskovej práce. Ďalej je ukázané o koľko bol výpočet na grafickej karte rýchlejší ako na procesore.

1 Trojrozmerné modelovanie

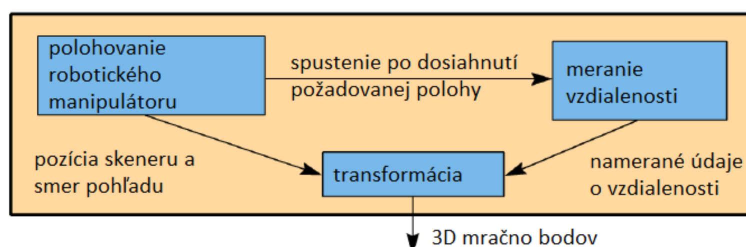
Trojrozmerné (3D) modelovanie objektu možno považovať za proces, ktorý začína získavaním údajov a končí 3D vizuálnym a interaktívnym modelom na počítači. Ako trojrozmerné modelovanie je často myslený iba proces prevodu nameraného mračna bodov do pretrojuholníkovanej siete (ang. mesh) alebo do textúrovaného povrchu, no za týmto pojmom sa skrýva aj komplexný proces rekonštrukcie objektu. Trojrozmerné modelovanie objektov a scény je dlhotrvajúcim výskumným problémom v oblasti grafiky, obrazu a fotogrametrii. Trojrozmerné modely sú potrebné v mnohých aplikáciach, ako napríklad navigácia, identifikácia objektu, vizualizácia a animácia[28].

V poslednej dobe sa 3D modely stali dôležitou súčasťou aj v medicíne. Trojrozmerné zobrazovacie senzory, navrhnuté špecialne pre ľudské telo sú čoraz dostupnejšie a poskytujú povrchové údaje s vysokým rozlíšením pre numericky a vnímateľne presné modely digitálnych tiel. Trojrozmerné modely sa používajú v plastickej chirurgii, ortopédii, protetickej ortodoncii, chirurgii a dermatológii[28].

Digitálne modely sú dnes všade prítomné, ich používanie a šírenie prostredníctvom internetu sa stáva veľmi populárne a môžu byť zobrazené už skoro na všetkých počítačoch. Hoci sa zdá ľahké vytvoriť jednoduchý 3D model, generácia presného, komplexného a realistického počítačového modelu si stále vyžaduje značné úsilie[30].

1.1 Proces vyhotovovania modelov

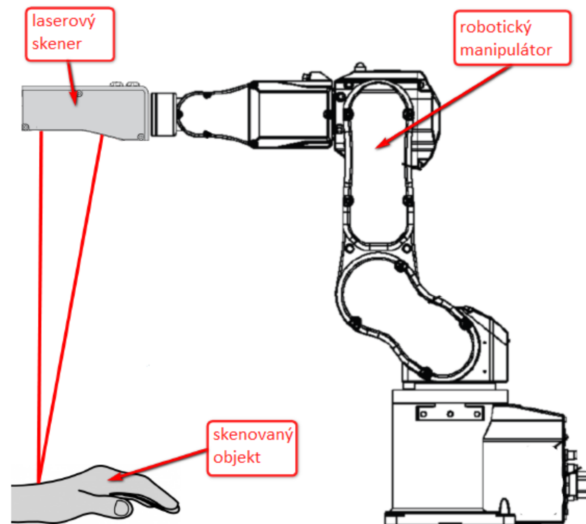
RoScan systém 3D modelovania je možné rozdeliť na dva subsystémy: prvým je Robotický 3D skener, čo je systém, ktorý prijíma požadovanú trajektóriu skenovania ako vstup, a na výstup generuje trojrozmerné mračno bodov (obr. 1.1) [13].



Obr. 1.1: Základná myšlienka prevádzky Robotického 3D skeneru.

Subsystém je zložený z komerčne dostupného manipulátora a laserového skeneru, ktorý je namontovaný priamo v koncovom bode manipulátora (obr. 1.2) vybaveného

softvérom v .NET framework. Druhý subsystém je softvér pre 3D modelovanie v ktorom sa špecifikuje trajektória skenovania a spracováva data z 3D skeneru do finálnej formy 3D modelu. Tento softvér je taktiež napísaný v Microsoft .NET štandarde[13].



Obr. 1.2: Prehľad spracovania Robotického 3D skeneru[13].

1.1.1 3D modely vytvorené Robotickým 3D skenerom

Vytvorené 3D modely sú uložené do súboru vo formáte, ktorý je založený na architekrúre XML, ktorá je podrobne popísaná v [13]:

- Metadata: Metadáta sú vo formáte XML a obsahujú informáciu o tvorcovi, čase vytvorenia a konfigurácii s ktorou bol model vytvorený. Každá informácia je uložená v samostanom prvku XML.
- Vertices: Binárne data, ktoré sú kódované v Base64 a obsahujú 3D súradnice bodov patriacich k modelu. Každá súradnica je uložená v štvor bajtovom formáte float.
- Indices: Binárne dáta, ktoré sú kódované v Base64, ktoré obsahujú indexy bodov s ktorými sa definujú trojuholníky (z každých troch indexov sa vytvorí jeden trojuholník). Každý index je uložený v štvor bajtovom formáte integer.
- Trajectory: Skenovania trajektória pre účely dodatočného spracovania.
- Reflected Intensity: Binárne dáta, ktoré sú kódované v Base64 a obsahujú intenzitu odrazu laserového lúča pre každý bod v modeli. Každá hodnota je uložená v dvoj bajtovom short integer formáte.

- Reflected Width: Binárne data, ktoré sú kódované v Base64 a obsahujú šírku odrazu laserového lúča pre každý bod v modeli. Každá hodnota je uložená v dvoj bajtovom integer formáte.

2 Spájanie modelov

Aplikácie ako sú virtuálne múzeá, reverzné inžinierstvo, alebo v medicíne potrebujú virtuálne modely, ktoré sú veľmi blízko k ich reálnemu modelu. Dnes už snímače poskytujú jednoduchý a rýchly spôsob snímania 3D údajov, ale väčšina z nich trpí rovnakým problémom, keďže údaje sa merajú len z jedného pohľadu. Naraz sa dá skenovať iba časť povrchu. Preto sa musia vykonať snímky z viacerých pohľadov a tieto jednotlivé snímky sa potom musia skombinovať, aby sa vytvoril konečný model[19].

Typicky môže byť proces modelovania rozdelený do troch fáz. Povrch objektu sa najskôr nasníma z niekoľkých pohľadov. V druhom kroku s názvom "registrácia"sú výsledné pohľady zarovnané, aby sa získala ich relatívna poloha vzhľadom na snímaný objekt. Nakoniec, sú pohľady zlúčené do jedinečnej siete, ktorá vytvorí jedinečný a celistvý objekt, čím sa zaoberá táto práca[19].

Zlúčovanie modelov odvodené z [19] prebieha v nasledujúcich krokoch:

1. Nájdenie prekrytia – hľadajú sa miesta snímaného objektu, ktoré sú totožné v oboch nasnímaných objektoch.
2. Vymazanie alebo spriemerovanie prekrytia.
3. Detekcia hrán na vzniknutej medzere.
4. Vyplnenie medzery novými trojuholníkmi.

2.1 Nájdenie prekrytia

Existuje mnoho metód pre nájdenie prekrytia dvoch modelov. Niekoľko z nich si opíšeme v tejto sekcii.

2.1.1 Mriežka boxov

Jednou z metód ktorou sa reprezentujú priestorové údaje je mriežka boxov.

V tejto sekcii je najskôr vysvetlený pojem bounding box, z ktorých sa daná mriežka skladá a potom sa opíše využitie tejto metódy na indikovanie zrážky dvoch objektov, ktorá by sa po správnej úprave dala využiť aj na indikáciu prekrytia.

Bounding box sa okolo objektu definuje extrémom súradníc plochy objektu[35].

Implementácia tejto metódy pre indikáciu zrážky dvoch objektov je podľa [16] vytvorenie N x N x N mriežky boxov rovnakých rozmerov nad pracovným priestorom, ktorý predstavuje tvar kocky. Experimenty v danej práci ukazujú, že výber parametru N by mal byť pomerne malý a to v rozmedzí 5 až 50 prvkov. Pre každý

box v mriežke sa vypočítajú a uložia do zoznamu trojuholníky, ktoré sa nachádzajú v prekážke a ktoré daný box pretínajú. Jeden trojuholník môže pretínať viacero boxov, teda sa uloží do viacero zoznamov z čoho vyplýva že veľkosť výslednej dátovej štruktúry môže byť väčšia s porovaním s veľkosťou vstupu. Čas tohto spracovania môže v najhoršom prípade trvať $O(n \cdot N^3)$, avšak spracovanie môže byť rýchlejšie, a to v čase úmernom ku súčtu veľkostiam zoznamov pre všetky boxy (kocky).

Postup je taký, že pre daný objekt je spočítaný bounding box (BB (F)), kde je určená množina súradnicovej siete boxov ktoré pretínajú daný BB (F) jednoducho zistením x , y a z rozsahov BB(F). Potom autori považujú pre každý trojuholník prekážky (T) prepojenie s každým boxom v mriežke. Keď $BB(T) \cap BB(F) = \emptyset$, potom sa vie, že trojuholník T nepretína F. Keď $BB(T) \cap BB(F) \neq \emptyset$ tak sa trojuholník v modeli F nachádza a ďalej sa kontroluje v ktorom zmenšenom boxe (kocke) sa nachádza presne. V najhoršom prípade sa skontroluje každý trojuholník T proti všetkým zmenšeným boxom modelu F [16].

2.1.2 K-d Strom

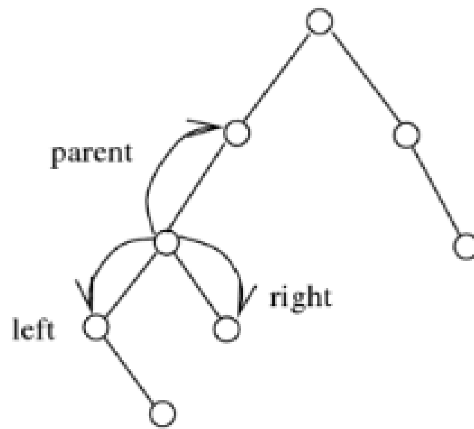
Ďalšou metódu ktorou sa reprezentujú priestorové údaje je k-d strom. K-d strom navrhol JL Bentley v práci [9] ktorá bolá poblikovaná v roku 1975.

V tejto sekcii sú vysvetlené pojmy binárny strom a hyperplocha, na ktorých je k-d strom založený. Potom je opísaný samotný k-d strom. A v poslednom rade je popísaná metóda ktorá používa k-d strom na indikovanie zrážky dvoch objektov, ktorá by sa pri správnej implementácii dala pretvoriť aj na detekciu prekrytia.

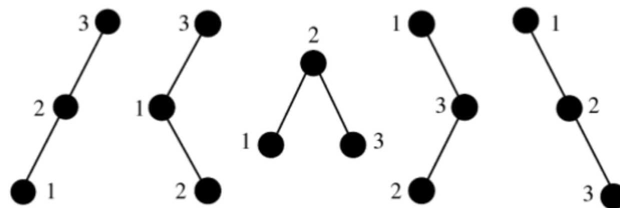
Binárny strom sa v informatike definuje ako stromová dátová štruktúra, v ktorej má každý uzol najviac dvoch potomkov, ktorí sa označujú ako ľavý a pravý potomok (obr. 2.1) [32].

Koreňový binárny strom je rekurzívne definovaný ako prázdny, alebo tvorený z uzla, ktorý sa nazýva koreň, spoločne z dvoma koreňovými binárnymi stromami nazývanými ľavý a pravý podstrom. V koreňových binárnych stromoch je dôležité aj poradie medzi “bratskými” uzlami, teda ľavý sa líši od pravého. Na obrázku 2.2 môžete vidieť tvary piatich rôznych binárnych stromov, ktoré sa môžu vytvoriť na uzloch stromu [32].

Hyperplocha je v geometrii definovaná ako subpriestor rozmeru n-1 alebo ekvivaletné kodimenzie 1 n-dimenzionálneho priestoru [10]. Jednoducho povedané, hyperplocha je podpriestor, ktorého rozmer je o jeden menší ako rozmer okolitého prostredia. Čiže keď je priestor 3-dimenzionálny, tak hyperplochy sú 2-dimenzionálne a keď je priestor 2-dimenzionálny, tak hyperplocha je 1-dimenzionálna čiara. Tento pojem



Obr. 2.1: Ukážka binárneho stromu[32].



Obr. 2.2: Päť rôznych binárnych stromov na troch uzloch[32].

sa dá použiť v akomkoľvek priestore, v ktorom je definovaná koncepcia rozmeru podpriestoru [29].

K-d strom je binárny strom, v ktorom každý uzol ktorý nemá potomkov (koncový uzol) je k-dimenzionálny bod. Body ktoré majú potomkov sa dajú považovať za body v ktorých je vygenerovaná hyperplocha ktorá delí priestor do dvoch častí. Body ktoré sa nachádzajú v ľavo od tejto hyperplochy sú reprezentované ľavým podstromom tohto uzla a body ktoré sú napravo od hyperplochy sú reprezentované pravým podstromom. Smer hyperplochy sa volí tak, že každý uzol v strome sa spojí s jednou dimeziou k, kde sa potom hyperplocha vytvorí kolmo na os tejto dimenzie. Napríklad keď je pre konkrétne rozdelenie vybraná os "x", tak všetky body podstromu s menšiou hodnotou ako je hodnota "x" v rodičovskom uzle, sa objavia v ľavom podstrome a všetky body s väčšou hodnotou "x" budú v pravom podstrome. V takom prípade by sa hyperplocha nastavila podľa hodnoty "x", teda

jej normála by bola os “x” [9].

Implementácia pre nájdenie zrážky podľa [16] začína vytvorením koreňového uzla, ktorý je spojený s pracovnou plochou. Každý uzol stromu je spojený s hyperplochou a implicitne so sadou trojuholníkov, ktoré sa nachádzajú v prekážke a pretínajú danú hyperplochu. Každý nekoncový uzol je tiež spojený s hyperplochou. Na určenie potomka uzla musíme splniť dve podmienky a to:

1. Hyperplocha musí byť kolmá na os x, y alebo z.
2. Na akej hodnote vybranej súradnicovéj osy bude rozdelenie umiestnené.

Ak počet trojuholníkov (ktoré sa nachádzajú v prekážke) pretínajúcich hyperplochu klesne pod prahovú hodnotu K, potom sa hyperplocha nerozdelí a zodpovedajúci uzol sa stane koncovým uzlom stromu. Následne sa pre každý koncový bod uloží zoznam prekrývajúcich trojuholníkov, ktoré pretínajú danú hyperplochu. Spracovanie dotazu q (čo reprezentuje oblasť ktorú testujeme na priesečník s prekážkou), sa začína tým, že q sa porovná s rovinou rezu koreňa. Keď q leží na jednej strane roviny rezu, tak sa algoritmus rekurzívne posunie na daného zodpovedajúceho potomka, keď leží v strede, tak treba navštíviť oboch potomkov. Keď sa algoritmus dostane na koncový bod, ktorý má počet trojuholníkov menší ako K, tak pre každý trojuholník skontroluje priesečník s oblasťou q .

2.1.3 R-stromy

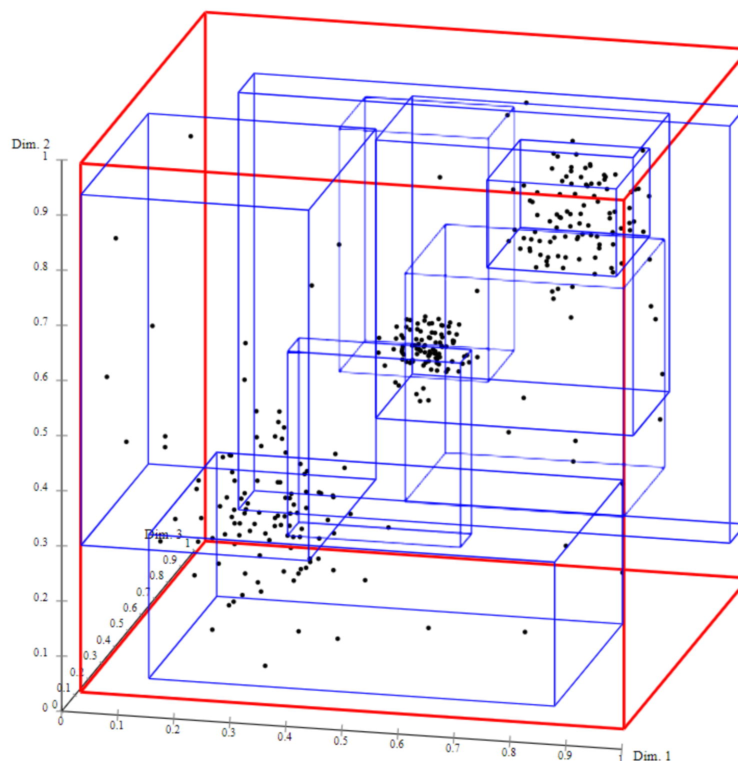
Alternatívnou stromovou reprezentáciou pre indexovanie priestorových informácii sú “R-Trees (R-Stromy)” a jej varianty [8]. R-strom navrhol Antonin Guttman v práci [15], ktorá bola publikovaná v roku 1984 a našli významné využitie v rôznych teoretických aj praktických prácach [22].

V tejto sekcií sa najskôr opíše pojem stromová datová štruktúra, na ktorej je R-strom založený. Potom sa opíše samotný princíp R-stromu a nakoniec sa opíše použitie R-stromu na indikovanie zrážky dvoch objektov, ktoré sa používa v práci [11].

Stromová datová štruktúra je určená k rekurzívnemu rozkladu priestoru a poskytuje kompaktný hierarchický popis objektu[27]. V informatike je strom veľmi populárnym abstraktným datovým typom (ADT). Je to datová štruktúra implementujúca ADT, simulujúca hierarchickú stromovú štruktúru s koreňovou hodnotou a podstromami potomkov s nadradeným uzlom a je reprezentovaná ako skupina prepojených uzlov. Stromová datová štruktúra sa dá definovať ako súbor uzlov (začínajúcich koreňovým uzlom), kde každý uzol je datová štruktúra pozostávajúca z

hodnoty a zoznamu referencií na uzly (potomkov) s obmedzením, že žiadna referencia na potomka nemôže byť duplikovaná a žiadny uzol nemôže mať referenciu na koreň [25].

R-stromy sú stromové datové štruktúry používané pre metódy priestorového prístupu, napríklad na indexovanie viacrozmerných informácii, ako sú geografické súradnice, obdĺžniky alebo polygóny [34]. Kľúčovou myšlienkou tejto datovej štruktúry je zoskupenie blízkych objektov a ich reprezentácia s minimálnym ohraničujúcim boxom v ďalšej vyššej úrovni stromu. A pretože všetky objekty ležia v tomto ohraničujúcom boxe, dotaz, ktorý nepretína ohraničujúci box nemôže tiež pretínať žiadny z obsiahnutých objektov. Na koncovej úrovni opisuje každý box ako jeden objekt, na vyšších úrovniach ako agregácia rastúceho počtu objektov. Môžeme to považovať za čoraz hrubšiu aproximáciu súboru údajov. R-strom má typickú vlastnosť, že každý koncový, vnútorný alebo koreňový uzol má počet potomkov preddefinovaných v rozsahu [m , M] [11]. Vizualizácia r-stromu trojrozmerných bodov je ukázaná na obrázku 2.3.



Obr. 2.3: Vizualizácia R-stromu pre 3D body použitím ELKI softvéru[31, 6].

Implementácia pre detekciu zrážky dvoch objektov opísaná v práci [16] je postavená na verzii binárnych stromov. A to tak, že každý uzol stromu zodpovedá oblžníkovému boxu a podmnožine trojuholníkov z prekážkového modelu. Koreň je spojený s celým pracovným priestorom a so všetkými prekážkami. V práci je ku každnému uzlu priradených T (určitý počet) trojuholníkov. Keď $|T| \leq K$, kde K je prah (ktorého hodnota je parametrom), tak sa uzol stáva koncovým uzlom, a trojuholníky T uložíme do zoznamu spojeného s týmto uzlom. Keď $|T| > K$ tak rozdelíme T (zhruba) na polovicu pomocou vypočítania strednej hodnoty pre súradnice "x", "y" alebo "z" z ťažísk trojuholníkov zo zoznamu T a priradením trojuholníkov k potomkom podľa toho, aké sú ťažiská jednotlivých trojuholníkov s porovnaním ku strednej hodnote. Vyberá sa spomedzi troch možností rozdelenia na základe minimalizovania (a) $max\{V_1, V_2\}$, alebo (b) $V_1 + V_2$, kde V_1 a V_2 sú objemy bounding boxov dvoch podmnožín, ktoré sú výsledkom výberu rozdelenia.

2.1.4 Priesečník lúča s trojuholníkom

V predchádzajúcich metódach sa riešila minimalizácia plochy ktorú treba skontrolovať pri prekrytí avšak neriešilo sa, ako sa k daným bodom jedného modelu priradia trojuholníky druhého modelu, čo je v našej aplikácii dôležité, pretože pri skenovaní ľudkých častí tela sa často stáva že dva skeny modelov nie sú úplne totožné, pretože napr. ruka sa môže pri skenovaní pohnúť, alebo sa na nej môže napnúť sval a teda treba priestor v prekrytí spriemerovať.

V tejto sekcii si najskôr opíšeme čo je to lúč, trojuholník a barycentrické súradnice ktoré sa používajú v metóde ktorú navrhol Tomas Möller a Ben Trumbore v práci [23], ktorá bola publikovaná v roku 1997.

Barycentrické súradnice

Barycentrické súradnice sa používajú na vyjadrenie polohy ktoréhokoľvek bodu na trojuholníku pomocou troch skalárov. Poloha tohto bodu zahŕňa akúkoľvek polohu vo vnútri trojuholníka, akúkoľvek polohu na ktoromkoľvek z troch okrajov trojuholníka alebo akúkoľvek polohu z vrcholov samotného trojuholníka. Na výpočet polohy daného bodu pomocou barycentrických súradnic sa používa následujúca rovnica:

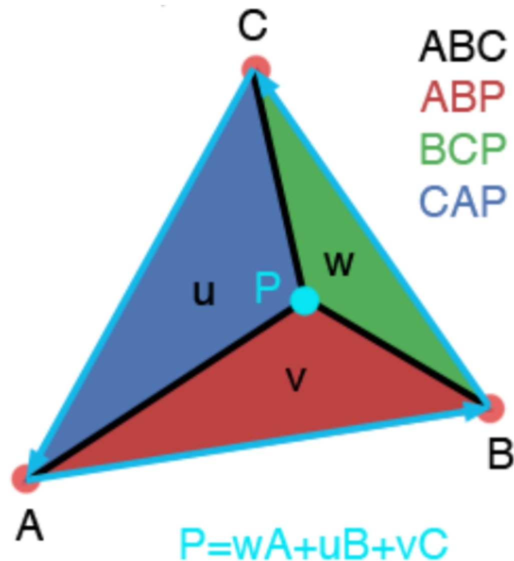
$$P = uA + vB + wC \tag{2.1}$$

kde A B a C sú vrcholy trojuholníka a u, v ,w (barycentrické súradnice) sú tri reálne čísla (skaláry), že musí platiť

$$u + v + w = 1 \tag{2.2}$$

čo znamená, že barycentrické súradnice sú normalizované. Rovnica definuje polohu bodu P v rovine trojuholníka tvoreného vrcholmi A, B a C. Bod je vo vnútri trojuholníka keď platí $0 \leq u, v, w \leq 1$. Ak je niektorá zo súradníc menšia ako nula, alebo väčšia ako jedna, bod je mimo trojuholníka. Ak je niektorá z nich nula, P sa bude nachádzať na jednej z čiar spájajúcich vrcholy trojuholníka [4].

Barycentrické súradnice známe tiež ako súradnice plochy, kde tento výraz naznačuje, že súradnice sú úmerné ploche troch podtrojuholníkov definovaných bodom P, ktorý sa nachádza v trojuholníku s vrcholmi trojuholníka (A, B, C). Tieto tri podtrojuholníky sú označené ako ABP, BCP, CAP (obr.2.4) [4].



Obr. 2.4: Barycentrické súradnice sa dajú chápať ako suboblasť subtrojuholníkov CAP (pre u), ABP (pre v) a BCP (pre w) nad oblasťou trojuholníka ABC[4].

Möller-Trumborov algoritmus

Lúč R(t) s počiatkom O a normalizovanou smernicou D je definovaný ako

$$R(t) = O + tD \tag{2.3}$$

a trojuholník je definovaný troma vrcholmi V_0, V_1 a V_2.

Pre bod T(u, v) po vyjadrení w v rovnici 2.2 a dosadení do rovnice2.1 dostaneme

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{2.4}$$

kde (u, v) sú barycentrické súradnice, ktoré musia spĺňať $u \geq 0$, $v \geq 0$ a $u+v \leq 1$. Vypočítanie priesečníka medzi lúčom R(t) a trojuholníkom T(u, v) je ekvivalentné

k $R(t) = T(u, v)$, po dosadení:

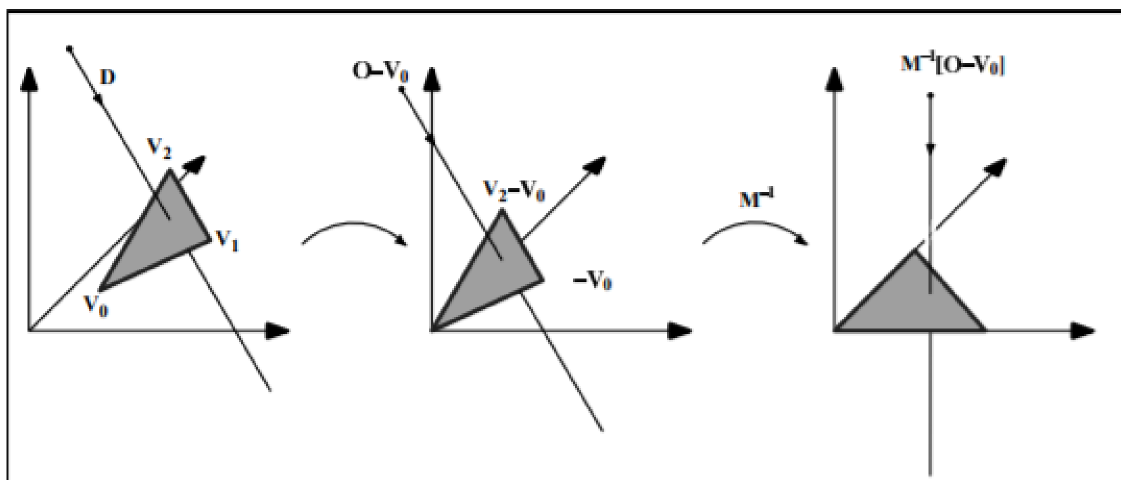
$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \tag{2.5}$$

Preusporiadaním výrazov dostaneme:

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{2.6}$$

To znamená, že barycentrické súradnice (u, v) a vzdialenosť t od počiatku lúča k bodu priesečníku sa dá nájsť vyriešením vyšie uvedeného lineárneho systému rovníc.

Vyšie uvedené môže byť geometricky myslené ako preklad trojuholníku do počiatku novej súradnicovej siete a transformovanie ho na jednotkový trojuholník v y a z so smerom lúča zarovnaným s x, ako je možné vidieť na obrázku 2.5 (kde $M = [-D, V_1 - V_0, V_2 - V]$ je maticou v rovnici 2.6]).



Obr. 2.5: Preklad a zmena počiatku lúča[23].

Už Bogart a Arenberg v [12] sa zaoberali spôsobom nájdenia priesečníka lúča a trojuholníka pomocou vypočítania barycetrických súradníc, kde tiež skonštruovali 3×3 maticu, ale miesto smernice lúča použili normálu trojuholníka. Táto metóda si vyžaduje ukladanie normály pre každý trojuholník, alebo jej vypočítanie za behu.

Označením $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ a $T = O - V_0$, a požitím Cramerového pravidla na rovnicu 2.6 získame riešenie v tvare:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D, E_1, E_2|} \begin{bmatrix} |T, & E_1, & E_2| \\ |-D, & T, & E_2| \\ |-D, & E_1, & T| \end{bmatrix} \tag{2.7}$$

Pomocou použitia lineárnej algebry, kde $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$ môže byť rovnica 2.7 prepísaná ako:

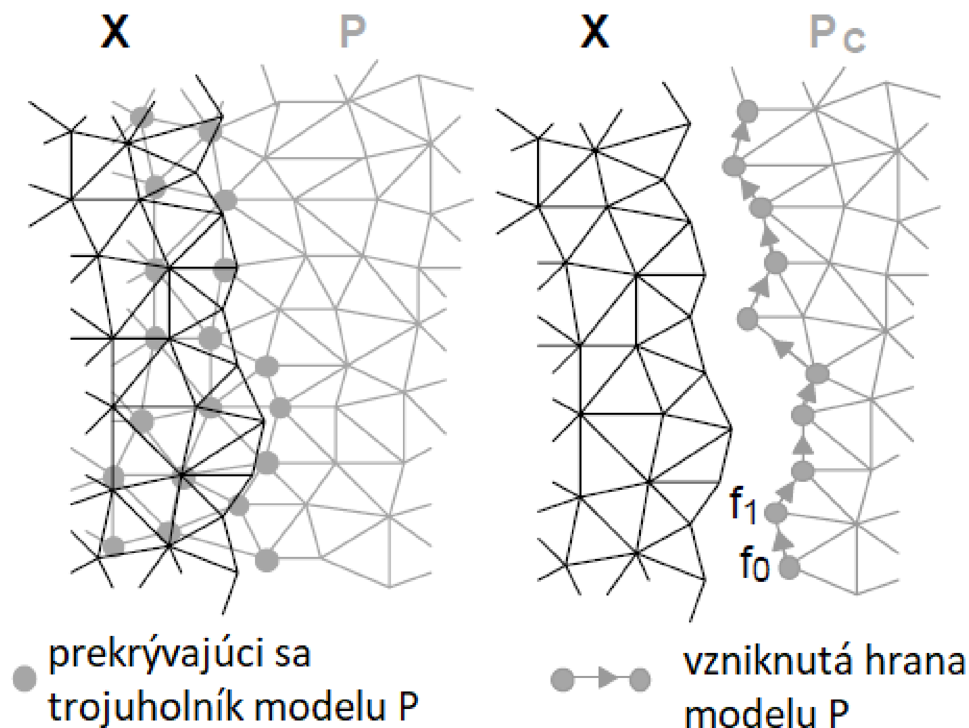
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} (Q \cdot E_2 \\ (P \cdot T \\ (Q \cdot D \end{bmatrix} \tag{2.8}$$

kde $P = (D \times E_2)$ a $Q = T \times E_1$.

2.2 Výsledný model

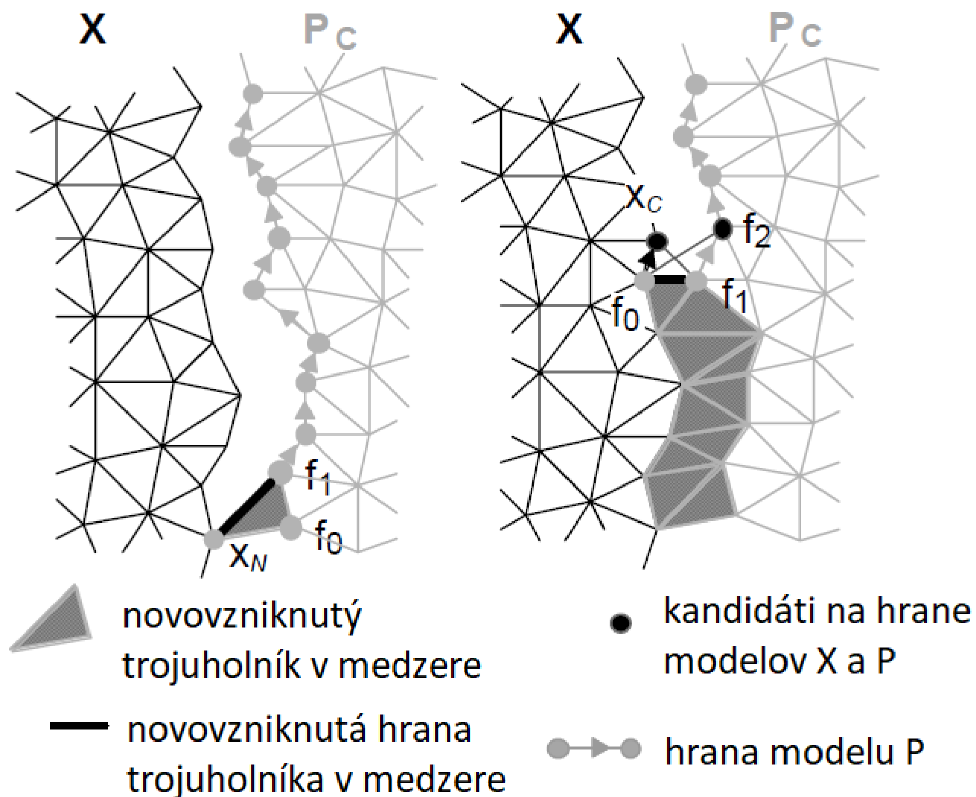
Po nájdení prekytia oboch modelov sa vo väčšine prípadov postupuje tak, že z jedného modelu sa povrch v prekrytí odstráni a následne sa pretrojuholníkuje vzniknutá medzera medzi týmito modelmi.

Podobný princíp opísali už Hügli a Jost v [19]. Kde sa ešte pred odstránením plochy vypočíta pre každý bod modelu do koľkých trojuholníkov patrí. Potom sa plocha z toho modelu odstráni a body, v ktorých sa zmenil počet trojuholníkov sú okrajové body vzniknutej medzery. Odstránenie plochy a nájdené okrajových bodov je znázornené na obrázku č. 2.6.



Obr. 2.6: Odstránenie prekryvajúcej plochy a nájdenie okrajových bodov[19].

Následne trieba pretrojuholníkovať vzniknutú medzeru a to sa spraví tak, že sa najskôr spoja dva z nájdených bodov, ktoré sú najviac pri okraji medzery, potom sa ku ním nájde najbližší bod z druhého okraju a tým vznikne nový trojuholník. Potom sa rekurzívne tvoria nové trojuholníky tak, že vieme že ďalší trojuholník bude mať spoločnú hranu s predošlým trojuholníkom a tretí bod sa hľadá tak, že sa nájdu susedia bodov na spoločnej hrane a tretím bodom trojuholníka sa stane ten z ktorého vyjde trojuholník s menším pomerom hrán. Tento pricíp je popísaný na obrázku č. 2.6 [19, 26].



Obr. 2.7: Rekurzívne vyplnenie medzery novými trojuholníkmi[19].

Pri tvorení nových trojuholníkov v medzere si treba dávať pozor na dve veci, a to či nie je ďalší kandidát veľmi ďaleko od koreňových bodov, a či novovytvorený trojuholník nemá zápornú smernicu ako ten predošlý, čo by znamenalo že nový trojuholník ide proti smeru trojuholníkovania medzery [19].

3 GPGPU

Graphics processing units (GPU) boli pôvodne navrhnuté na podporu počítačovej grafiky. Ich architekrúra podporuje rýchlu manipuláciu s pamäťou a vysoký výkon spracovania pomocou masívneho paralelizmu, vďaka čomu sú vhodné na efektívne riešenie bežných grafických úloh. Táto architekrúra je však vhodná aj pre ďalšie programovacie úlohy, čo viedlo k vzniku nového pojmu *General-purpose computing on graphics processing units* (GPGPU). Spočiatku sa GPGPU venoval hlavne štandard *Compute Unified Device Architecture* (CUDA), no od roku 2006 sa pre programovanie GPGPU stáva štandard *Open Computing Language* (OpenCL) kvôli svojej nezávislosti na platforme čoraz populárnejším [20].

Táto kapitola sa bude najskôr venovať štandardu OpenCL a potom sa zľahka priblíži aj CUDA.

3.1 OpenCL

OpenCL je voľne dostupný štandard umožnujúci paralelné programovanie heterogénnych architektúr. OpenCL implementuje viac ako 10 predajcov a to vrátane AMD, Intel, IBN a Nvidia. Ďalšou výhodou je že OpenCL program je môžné spustiť na rôznych zariadeniach bez nutnosti rekompilácie [18]. OpenCL je určený pre paralelné programovanie a obsahuje jazyk, *Application programming interface* (API), knižnice a behové prostredie (ang. runtime system) na podporu vývoja softvéru. Napríklad pomocou OpenCL môže programátor písať programy ktoré sa spúšťajú na GPU bez potreby mapovania týchto algoritmov do 3D grafického API ako je napríklad OpenGL alebo DirectX [24].

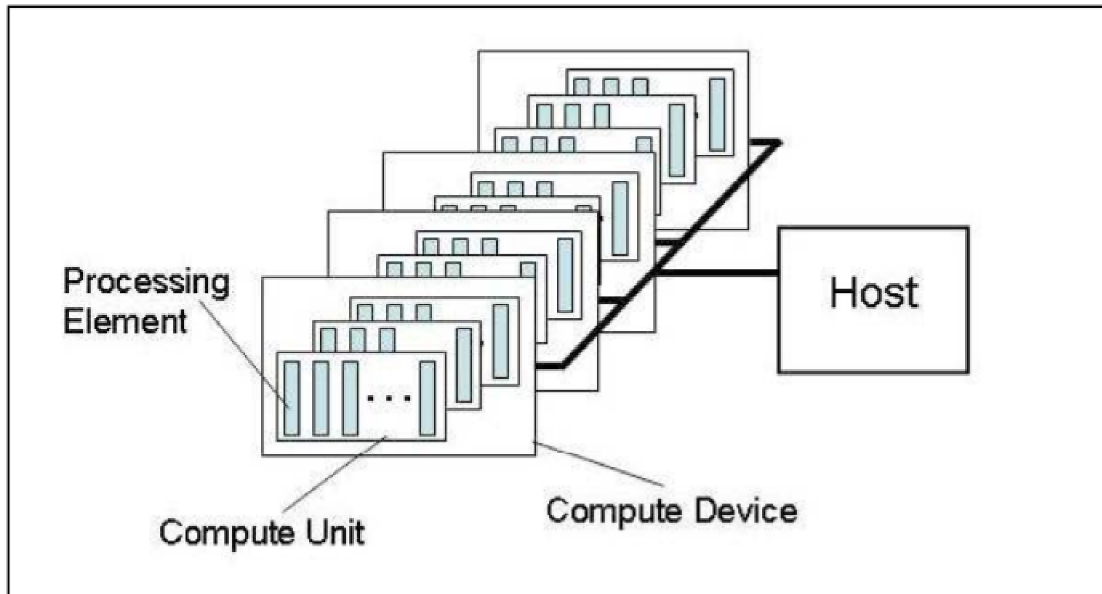
Na popísanie základnej myšlienky OpenCL sa používa hierarchia modelov, a to:

- Model Platformy,
- Výpočetný Model,
- Pamäťový Model,
- Programový Model [24].

3.1.1 Model Platformy

Model Platformy pre OpenCL je definovaný na obrázku č. 3.1. Model sa skladá z hostiteľa, ktorý je pripojený k jednému, alebo k viacerým zariadeniam OpenCL. Zariadenie OpenCL je rozdelené do jedného, alebo viacerých výpočetných jednotiek (*Compute Units* (CUs)), ktoré sa ďalej delia na jeden alebo viac prvkov pre spracovanie dát (*Processing elements* (PEs)). Výpočty na zariadení sa uskutočnujú vo vnútri PEs [24].

Aplikácia OpenCL beží na hostiteľovi podľa modelov, ktoré sú natívné na hostiteľskú platformu. Aplikácia OpenCL potom odošle príkazy od hostiteľa na vykonanie výpočtov do PEs. Tie ďalej spracujú data buď ako *Single Instruction, Multiple Data* (SIMD), kde sa všetky inštrukcie vykonajú v jednom prúde, alebo *Single program, Multiple data* (SPMD), kde má každé PE svoj vlastný čítač inštrukcií (*Program Counter* (PC)) [24].



Obr. 3.1: Platformový model[24].

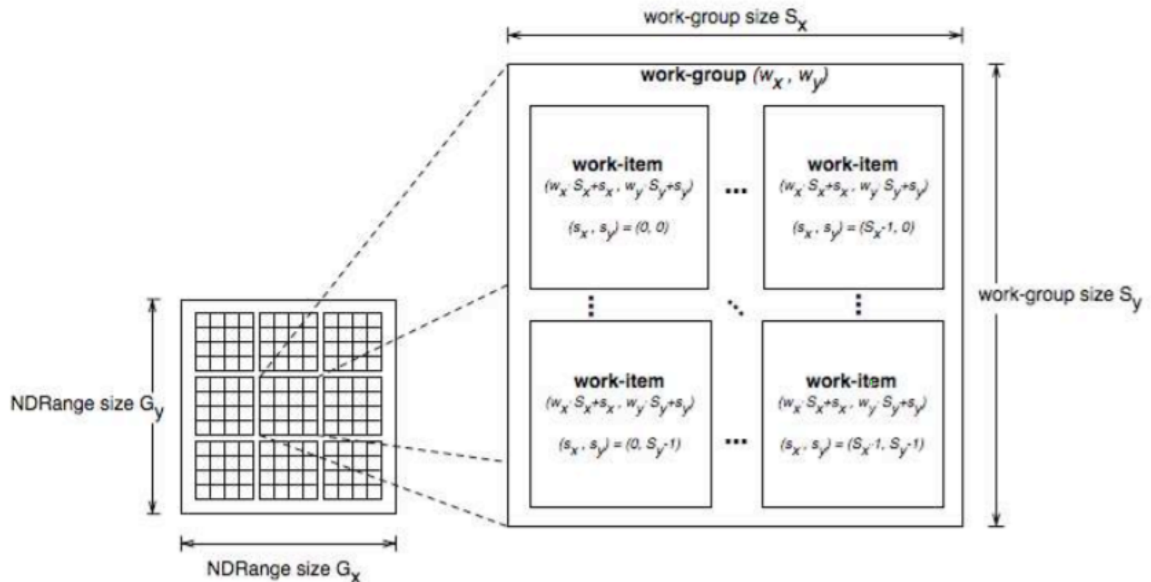
3.1.2 Výpočetný Model

Priebeh programu OpenCL može byť rozdelený do dvoch častí, a to do jadier (kernels), kde jadrá sú OpenCL funkcie, ktoré môžu byť vykonávané na jednom, alebo viacerých zariadeniach OpenCL, a na hostiteľský program, ktorý sa vykonáva na hostiteľovi. Hostiteľský program definuje kontext jadier a riadi ich [24].

Základ výpočetného modelu tkvie v spôsobe spracovania jadra. Keď je jadro hostitelom odoslané na vykonanie, tak sa definuje indexový priestor do ktorému sa priradí novovytvorená inštancia jadra v pamäti, ktorá sa nazýva pracovná jednotka (ang. work-item) a je identifikovaná svojim bodom v indexovom priestore, ktorý poskytuje globálne ID pre danú pracovnú jednotku [24].

Pracovné jednotky sú ďalej usporiadané do pracovných skupín (ang. work-groups), ktoré poskytujú hrubozrnejší rozklad indexového priestoru. Pracovným skupinám je tiež pridelené jedinečné ID s rovnakou rozmernosťou ako indexový priestor použitý pre pracovné jednotky [24].

Indexový priestor sa v OpenCL 1.0 nazíva NDRange, kde písmeno N určuje rozmer priestoru. Pre príklad dvojdimenzionálny indexový priestor je ukázaný na obrázku č. 3.2 [24].



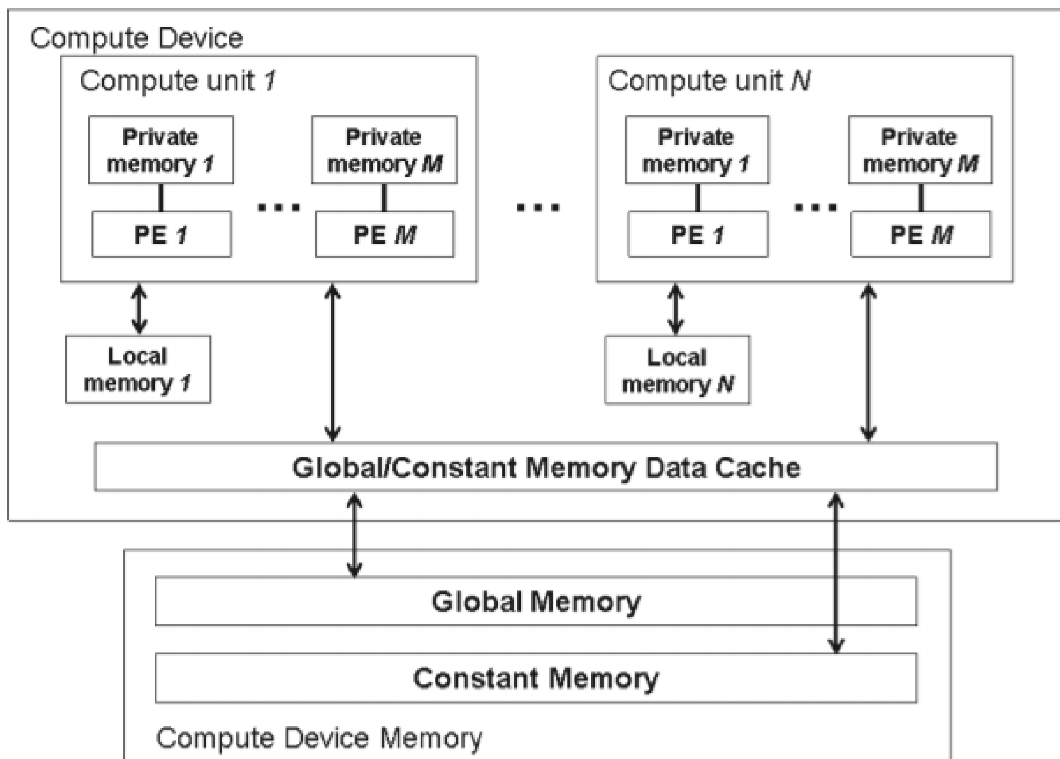
Obr. 3.2: Výpočetný model[24].

3.1.3 Pamäťový Model

Pracovné jednotky môžu pristupovať ku štyrom odlišným oblastiam pamäte:

- Globálna pamäť. Do tejto oblasti pamäte majú prístup na čitanie alebo zápis všetky pracovné jednotky a aj hosťujúca aplikácia.
- Konštatná pamäť. Časť globálnej pamäti, ktorá zostáva konštantná počas celého vykonávania jadra. Túto pamäť inicializuje hostitel a počas vykonávania jadra majú do nej pracovné jednotky prístup iba na čítanie.
- Lokálna pamäť. Táto pamäť je určená pre zdielanie premenných v priestore jednej pracovnej skupiny. Hostiteľ do tejto pamäti nemá prístup.
- Privátna pamäť. Oblasť pamäti určená len jedinej pracovnej jednotke. Premenné ktoré sú definované v danej jednotke nie sú viditelné pre iné jednotky a ani pre hostiteľkú aplikáciu [24].

  Pamäťové oblasti a ich vzťah k modelu platformy sú opísané na obrázku č. 3.3.



Obr. 3.3: Pamäťový model[24].

3.1.4 Programový Model

OpenCL podporuje dátovo paralelné a úlohovo paralelné programové modely. Primárny model, ktorý poháňa dizajn OpenCL je dátovo paralelný programový model [24].

OpenCL poskytuje hierarchický dátovo paralélny programovací model. Existujú dva spôsoby, ako určiť hierarchické rozdelenie. V explicitnom modeli programátor definuje celkový počet výpočetných jednotiek, ktoré sa majú vykonať paralelne a tiež ako sa tieto položky delia do pracovných skupín. V implicitnom modeli programátor špecifikuje iba celkový počet pracovných jednotiek, ktoré sa majú vykonať paralelne a rozdelenie na pracovné skupiny je už riadené implementáciou OpenCL [24].

Úlohovo paralélny model definuje výpočet ako jednu funkciu, ktorá je vykonaná nezávisle od indexového priestoru. Je to ekvivalentné spusteniu jadra s výpočetnou oblasťou, ktorá obsahuje iba jednu pracovnú jednotku [24].

3.2 CUDA

CUDA je paralelný platformový programovací model vytvorený spločnosťou Nvidia a poskytuje podporu len pre použitie grafických kariet od spoločnosti Nvidia. Im-

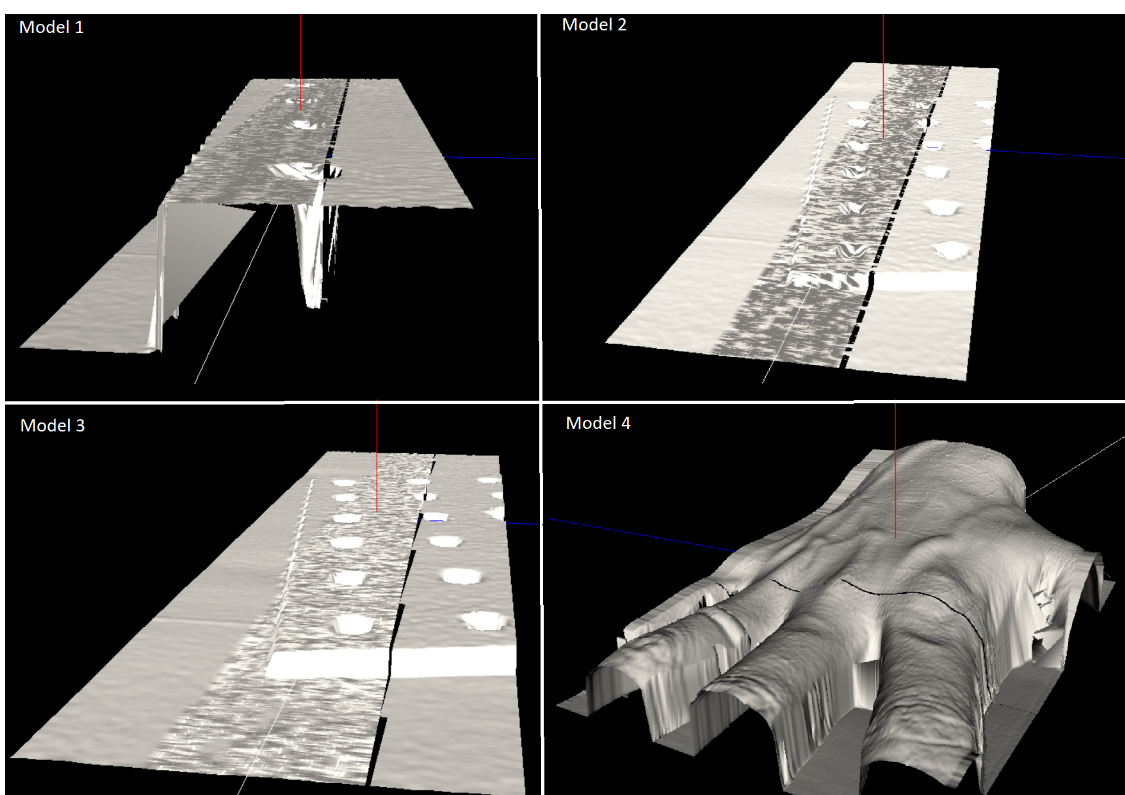
plementácia CUDA je svojím dizajnom takmer identická s implementáciu OpenCL [18].

Kvôli problému s tým že CUDA pre svoje pracovanie potrebuje programy CUDA Toolkit a Visual Studio a môže byť používaná iba na grafických kartách od spoločností Nvidia sa v našej práci bude používať OpenCL štandard.

4 Analýza východiskovej práce

Problematike spájania dvoch 3D modelov sa venoval už Marek Lampáš v bakalárskej práci [21].

Algoritmus bol testovaný na štyroch rôznych modeloch (obr. 4.1). Model 1 až 3 majú rovnaký počet bodov, čiže by mali mať približne rovnakú časovú náročnosť na výpočet. Prvý model bol naskenovaný kvôli otestovaniu správania algoritmu pri rôznych zaobleniach a priepastiach. Druhý model je tvarovo najjednoduchší pre spracovanie. Model 3 je naskenovaný tak, aby otestoval spájanie modelov pri situácii keď línie skenov nejdú paralélne a modely majú medzi sebou menší uhol. Posledný model je naskenovaná ruka, ktorá otestuje fungovanie algoritmu v praxi.



Obr. 4.1: Výsledné spojené modely pospájané algoritmom z bakalárskej práce Mareka Lampáša[21].

Prvým najviac viditeľným nedostatom je to, že po spojení vznikol na každom modeli úzky pás v ktorom sa nenachádza žiadné prepojenie. Ten vzniká z dôvodu zlyhania algoritmu, ktorý pri poslednom modeli nájde iba 118 okrajových bodov v medzere, čo je veľmi málo, pretože model má šírku 1280 bodov [21].

Ďalším problémom je, že z výsledných modelov vystupujú atypické plochy. Tieto vystupujúce plochy môžete vidieť na obrázku 4.2. Vo východiskovej práci [21] sa

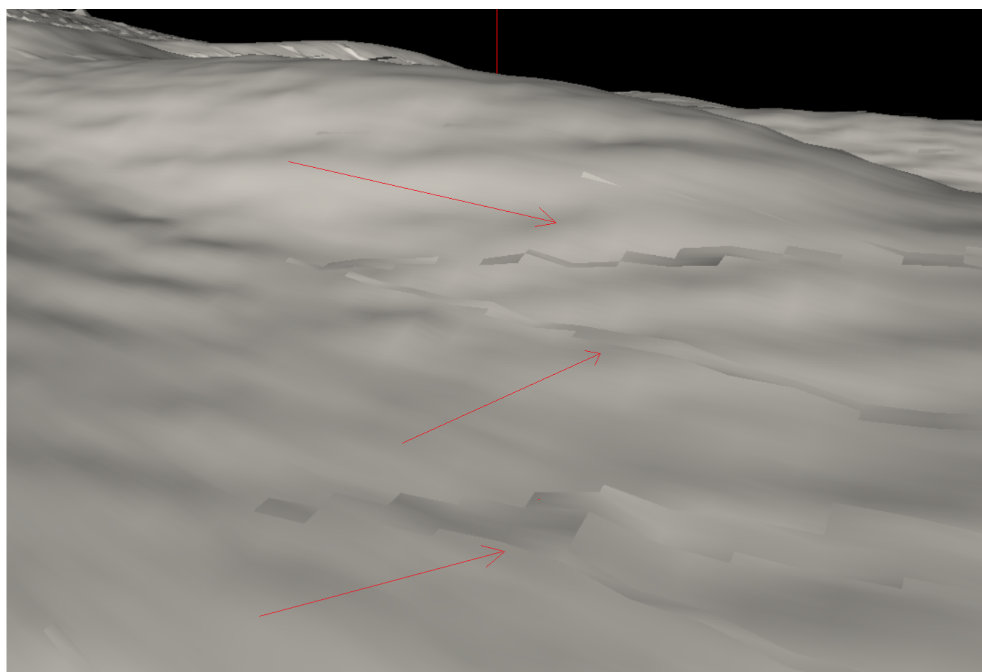
tvrdí, že tieto plochy vznikajú kvôli nesprávnemu priemerovaniu bodov.

Obr. 4.2: Nevhodné priemerovanie bodov pri spájaní čiastkových modelov[21].

Ďalším nedostatkom sú vznášajúce sa malé plochy pod alebo nad výsledným modelom, ktoré vznikajú v oblasti prekrytia. Tieto malé plochy sú ukázané na obrázku 4.3.

Posledným problémom, alebo skôr nevýhodou práce je veľká časová náročnosť. Spojenie pri prvom, druhom a treťom modeli ktoré obsahujú približne 100 000 bodov trvá približne 15 minút. Spojenie pri poslednom modeli ruky, ktorý obsahuje približne 500 000 bodov trvá až 5 hodín.

Úlohou tejto práce bude všetky tieto nedostatky opraviť a vytvoriť tak plne funkčný a spoľahlivý algoritmus.



Obr. 4.3: Nesprávne pretrojuholníkovanie bodov v prekrytí.

5 Návrh a implementácia nového riešenia

Postup riešenia našej práce je podobný postupom opísaným v prácach [21, 19, 26] a dá sa zhrnúť do bodov, ktoré si bližšie špecifikujeme v následujúcich sekciách:

1. Nájdenie prekrytia modelov.
2. Kontrola správnosti nájdených bodov prekrytia.
3. Spriemerovanie bodov v prekrytí.
4. Vytvorenie nových trojuholníkov v prekrytí, tak aby sa použili body z oboch modelov.
5. Nájdenie vzniknutej medzery medzi modelmi.
6. Rozšírenie nájdenej medzery.
7. Vyplnenie medzery novými trojuholníkmi.
8. Odstránenie nepoužitých bodov.

5.1 Nájdenie prekrytia modelov

Tomuto bodu sme sa v práci venovali najviac, pretože bol výpočetne najnáročnejší a potrebovali sme dosiahnúť čo najmenší čas pri rovnakej presnosti.

Postup nájdenia prekrytia je podobný postupu opísaný v [21], kde sa hľadal priesečník trojuholníka s lúčom, kde ako počiatok lúča sa berie bod v modeli a smernica lúča je vytvorená súčtom normál všetkých trojuholníkov do ktorých daný počiatok patrí.

Problém tohto návrhu spočíval v tom, že sa kontroloval každý lúč jedného modelu s trojuholníkmi druhého modelu a naopak. Tento spôsob hľadania prekrytia je časovo náročný a preto sme ho museli vylepšiť.

V našom postupe sa hľadanie prekrytia dá zhrnúť do následujúcich bodov.

1. Nájdenie bodov a trojuholníkov v spoločnej oblasti modelov.
2. Rozdelenie spoločnej oblasti do mriežky boxov. Kde každému boxu sa priradia trojuholníky, ktoré sa v ních nachádzajú.
3. Hľadanie priesešníku lúča a trojuholníkov. Kde sa najskôr nájde s ktorým boxom má lúč priesečník a následne sa berú trojuholníky iba z daného boxu.

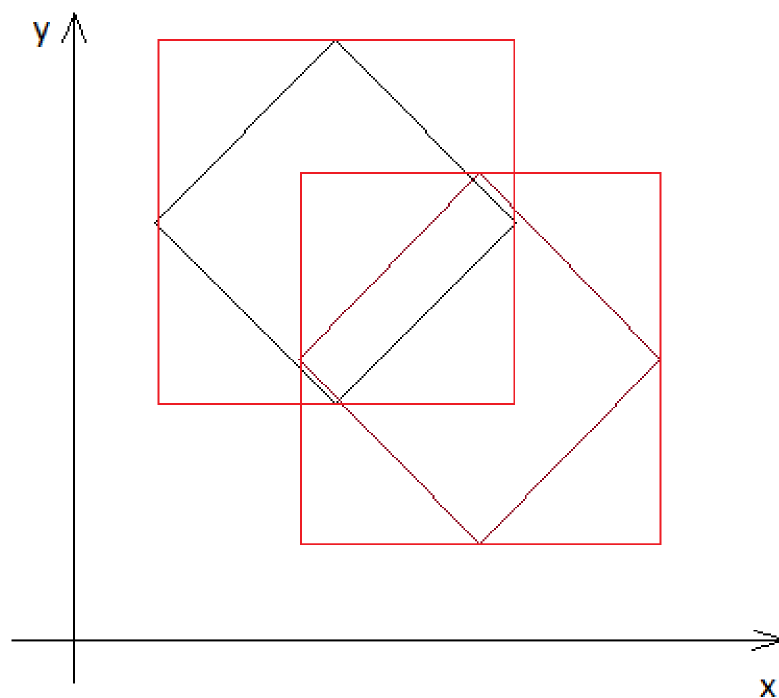
5.1.1 Nájdenie bodov a trojuholníkov v spoločnej oblasti modelov

Kvôli minimalizovaniu plochy na ktorej bude treba prepočítavať priesečníky lúčov a trojuholníkov sme si najskôr vytvorili bounding box pre oba modely. Bounding box je tvorený dvoma bodmi a to minimálnym a maximálnym bodom. Kde jeho strany sú paralelné s osou x, y a z [30]. Tie sa pre oba modely, ktoré sú tvorené

trojdimenzionálnymi bodmi nájdu tak, že sa prejdú všetky body daného modelu a hľadá sa najväčšia hodnota súradnic x, y a z. Po vypočítaní oboch boxov sa ďalej hľadajú body a trojuholníky, ktoré sa nachádzajú v spoločnej oblasti.

Body v spoločnej oblasti sa nájdu tak, že sa pre každý bod modelov zistí, či sa nachádzajú v oboch boxoch zároveň. Podmienkou či sa bod nachádza v boxe je, že veľkosti súradníc x, y a z sa musia nachádzať medzi minimálnou a maximálnou hodnotou boxu.

Problémom tejto minimalizácii plochy je, že pre najväčšiu minimalizáciu musia byť modely skenované čo najviac rovnobežne s osami. Čím väčší bude mať sken uhol k jednej z osi, tým menej sa zminimalizuje spoločná plocha. Tento problém je ukázaný v 2D na obrázku 5.1, kde sú bounding boxy označené červenou farbou. Na obrázku je vidieť, že prekrytie modelov je oveľa menšie ako prekrytie vytvorených boxov. Tento problém nemá vplyv na tvar výsledného modelu, akurát pri horšej minimalizácii sa bude musieť počítať viac priesečníkov.



Obr. 5.1: Ukážka vytvorenia Bounding boxov pre modely, ktoré nie sú rovnobežné s osami. Bounding box je uznačený červenou farbou.

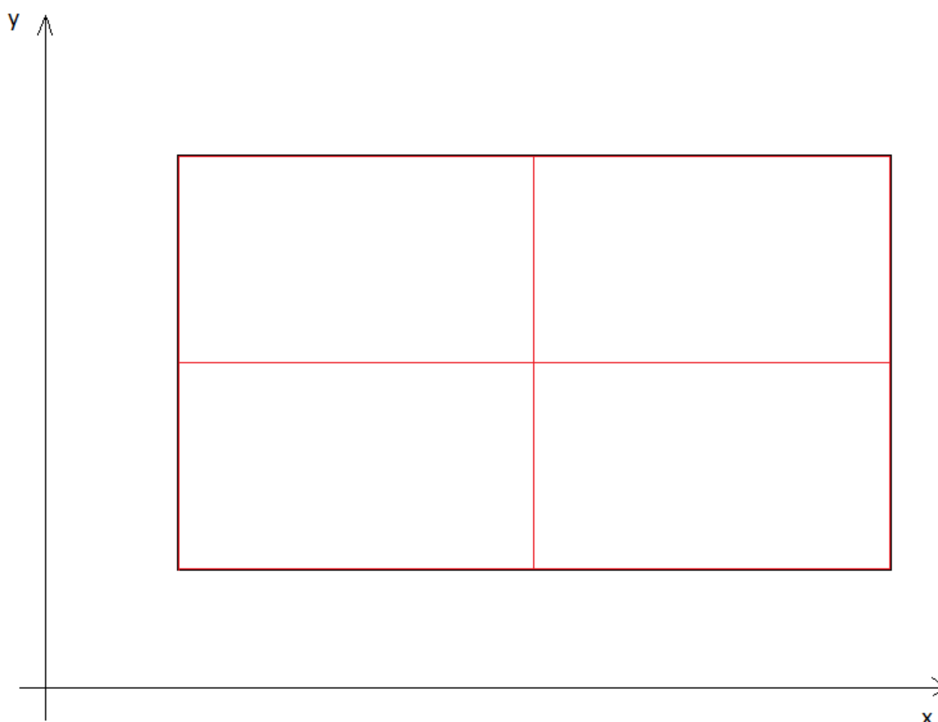
5.1.2 Rozdelenie spoločnej oblasti do mriežky boxov

Ďalšiu minimalizáciu plochy pre počítanie priesečníkov riešime pomocou rozdelenia jedného bounding boxu do viacerých čiastkových bounding boxov podobne ako v práci [17], ktorej postup je opísaný vyššie.

V našej implementácii sa najskôr vypočíta bounding box z minimalizovanej plochy ktorá bola nájdená v predošlom kroku a tá sa potom rozdelí do $X \times Y \times Z$ čiastkových bounding boxov.

Postup pre vytvorenie bounding boxu pre 3D plochu je už popísaný vyššie, kde sa prejdú všetky body a nájdu sa najväčšie hodnoty pre každú súradnicu.

Rozdelenie jedného bounding boxu do $X \times Y \times Z$ čiastkových bounding boxov sme navrhli tak, že najskôr sa pre každú dimenziu spočíta dĺžka boxu a potom sa táto dĺžka podelí podľa čísel X, Y a Z. Tento pricíp je ukázaný na obrázku 5.2, kde sa pre základný box označený čiernou farbou vypočítajú 2×2 čiastkové bounding boxy, tento obrázok je síce dvojdimezionálny, no na ukázanie princípu to stačí.



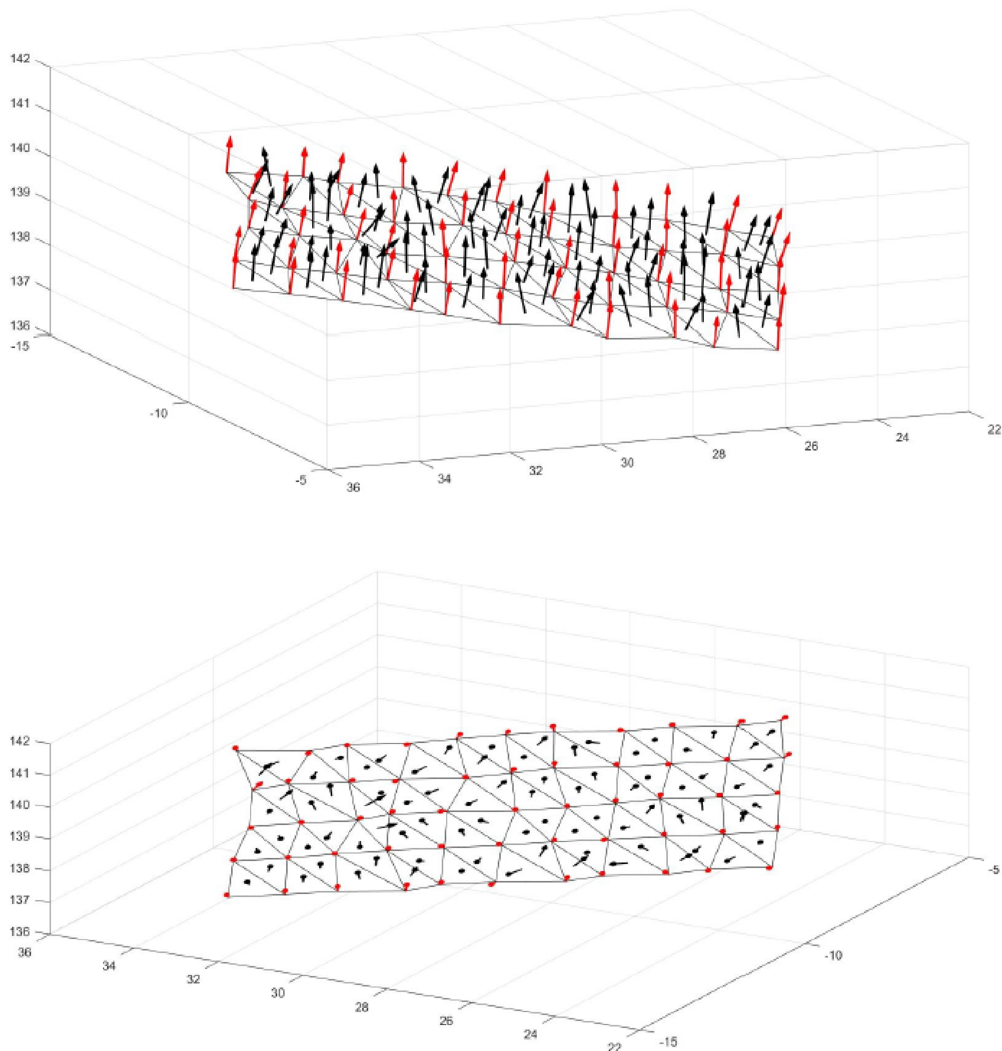
Obr. 5.2: Ukážka princípu vytvorenia čiastkových bounding boxov.

Následne sa ku každému čiastkovému boxu priradia trojuholníky oboch modelov, ktoré sa v danom boxe nachádzajú.

5.1.3 Hľadanie priesešníku lúča a trojuholníkov

Tento krok sa najskôr zaoberá výpočtom lúčov, potom nájdenia s ktorými čiastkovými boxmi (vypočítanými v predošlom kroku) sa lúč pretína a následne sa hľadá priesečník medzi každým lúčom a trojuholníkmi patriacích do daného pretínajúceho boxu.

Výpočet lúča prebieha pre každý bod, kde poloha bodu je počiatkom lúča, v minimalizovanej ploche tak, že sa nájde každý trojuholník do ktorého daný bod patrí. Pre tieto trojuholníky sa vypočítajú normály a súčet týchto normál sa stáva smernicou pre daný lúč. Pre lepšie pochopenie je možné na obrázku obr. 5.3 vidieť povrch, na ktorom sú čiernou farbou vyznačené normály plôch trojuholníkov a červenou farbou vypočítané lúče.



Obr. 5.3: Normálové vektory trojuholníkov a lúče[21].

Keď máme vypočítané lúče, tak treba zistiť akými čiastkovými boxami lúč prechádza. Priesečník lúča s boxom sa nájde tak, že sa minimálna a maximálna hodnota boxu porovná s rovnicou lúča 2.3. Teda rovnica lúča sa porovná s bodmi boxu pre každú súradnicu, z čoho pre 3D priestor vznikne šesť lineárnych rovníc

$$\begin{aligned} t0_x &= (B0_x - O_x)/D_x \\ t1_x &= (B1_x - O_x)/D_x \\ t0_y &= (B0_y - O_y)/D_y \\ t1_y &= (B1_y - O_y)/D_y \\ t0_z &= (B0_z - O_z)/D_z \\ t1_z &= (B1_z - O_z)/D_z \end{aligned} \tag{5.1}$$

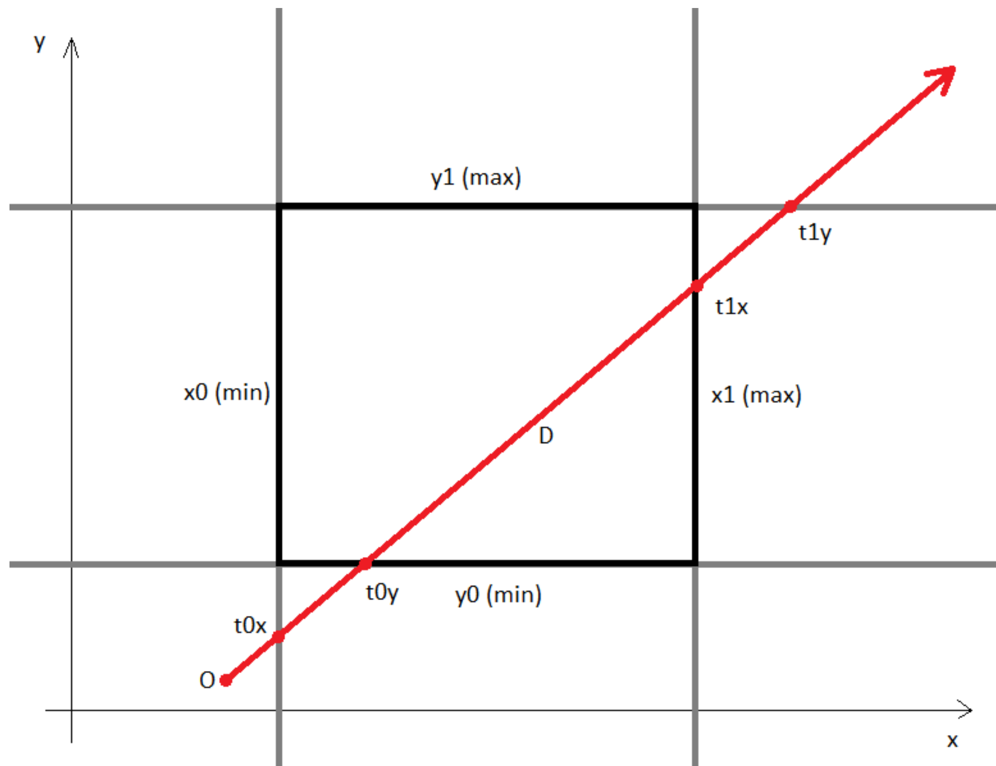
kde t je vzdialenosť počiatku lúča od priesečníku a *B0, B1* sú minimálny a maximálny bod bounding boxu. Treba brať do úvahy aj možnosť, keď je lúč rovnobežný s osou, tak sa nebude pretínať s ohraničujúcou rovinou (v tomto prípade je rovnica priamky pre lúč zredukovaná na konštantu a rovnica nemá riešenie). Keď vieme kde lúč pretína plochy, tak stačí nájsť či je tento priesečník vo vnútri, alebo von z bounding boxu a to zistíme na základe jednoduchého testu, ktorý si najskôr vysvetlíme na obrázku 5.4, ktorý je zakreselný v 2D rovine, kde lúč najskôr pretína plochy definované minimálnym bodom boxu v dvoch miestach: $t0_x$ a $t0_y$. Pri rôznych smeroch lúča môžeme matematicky zistiť ktorý z bodov leží v boxe tak, že zistíme ktorá hodnota vzdialenosti t je väčšia. Proces nájdenia druhého bodu v ktorom lúč pretína box je podobný tomu predchádzajúcemu, avšak hľadáme najmenšiu hodnotu z $t1_x$ a $t1_y$ [3].

Na obrázku 5.4 sú zakreslené lúče ktoré 2D box nepretínajú (pár možných variantov). Tieto prípady tiež ľahko identifikujeme porovnávaním vzdialeností *t*. Ako je vidieť na obrázku 5.4, lúč box nepretína, keď *t0x* je väčšie ako *t1y* a keď *t0y* je väčšie ako *t1x* [3].

Nakoniec môžeme túto techniku rozšíriť na 3D rovinu výpočtom hodnôt vzdialeností pre z súradnicu a porovnať ich ku vzdialenostiam vypočítaných pre súradnice x a y[3].

Keď máme pre každý lúč priradené všetky boxy ktoré pretína, tak posledným krokom je zistiť či lúč pretína nejaké trojuholníky z daného boxu. Test na pretnutie sa vypočíta pomocou Möller-Trumborového algoritmu, ktorý je opísaný v sekcii 2.1.4. Tento krok sme sa rozdodli vypočítať na grafickej krate, pretože aj napriek zminimalizovaniu plochy sa každému lúču pridelí veľký počet trojuholníkov s ktorými treba spraviť test.

Ako štandard pre komunikáciu s grafickou kartou sme si vybrali OpenCL, ktorý je opísaný v sekcii 3.1. Pre použitie štandardu OpenCL v .NET rozhraní používame

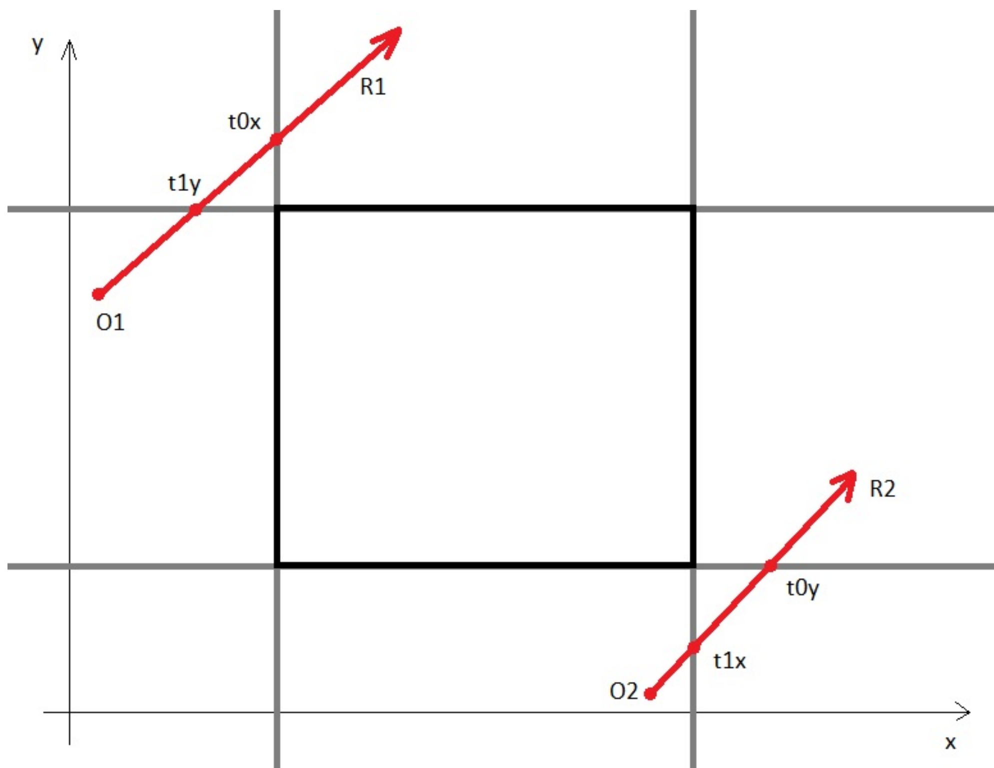


Obr. 5.4: Lúč ktorý pretína 2D bounding box.

knižnicu CUDAfy, ktorá .NET kód najskôr prekompiluje na OpenCL kód a potom daný kód spustí na grafickej karte [14].

Program bolo treba najskôr prepísať do OpenCL štandardu kde sa nemôžu používať metódy v štruktúrach a vo funkciách sa nemôžu alokovať nové štruktúry.

Výpočet pre všetky lúče prebiehal následovne a vychádza z poznatku, že jedna aplikácia na grafickej karte môže mať iba obmedzený počet inštrukcií. Najskôr sa na hosťovi alokujú polia v ktorých je určené ktoré čiastkové bounding boxy lúče pretínajú a ktoré trojuholníky sa v danom boxe nachádzajú. Taktiež sa musí alokovať priestor aj pre výsledky. Potom sa na grafickej karte alokuje miesto pre všetky lúče a trojuholníky ktoré sa nachádzajú v prekrytí, do ktorého sa následne tieto hodnoty nakopírujú a taktiež sa musí alokovať aj priestor pre výsledky. Potom sa postupne a asynchrónne (neblokojúco) púšťajú jadrá (OpenCL funkcie) do ktorých sú tiež asynchrónne (neblokojúco) kopírované indexy lúčov a trojuholníkov po čiastkových blokoch podľa ukazateľa do hosťovskej pamäti. Čiže každé jadro počíta s lúčmi, ktoré pretínaju ten istý bounding box s trojuholníkmi ktoré sa v danom boxe nachádzajú. Po ukončení jadra sa výsledky asynchrónne prekopíruju do predom alokovanej hosťovskej pamäti, podľa indexu ktorý sa vypočíta vynásobením indexu jadra a počtu lúčov v danom boxe. Následne prebieha synhronizácia jadier (čo je blokujúca fun-



Obr. 5.5: Lúče ktoré nepretínajú 2D bounding box.

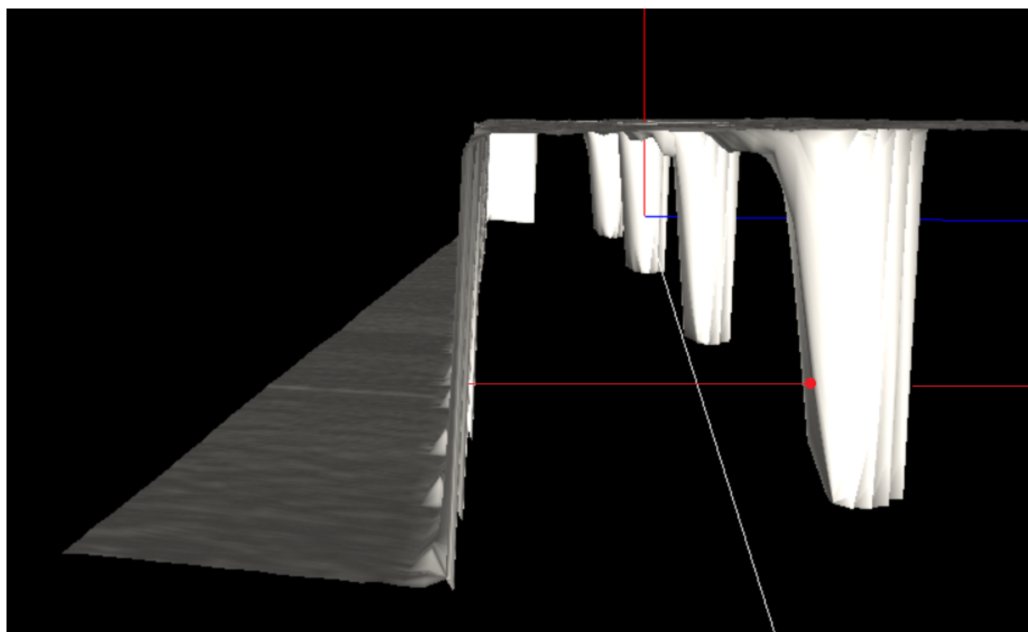
kcia ktorá čaká na ukončenie všetkých jadier). Po synchronizácii sa všetky výsledky skopírujú späť do .NET poľa, v ktorom sa nachádzajú indexy na lúč a trojuholník, ktorých vzdialenosť pretnutia je najmenšia pre danú vzdialenosť počiatku lúča od pretnutia. Bližší opis návrhu algoritmu pre grafickú kartu je rozpísaný v kapitole 6.

Keď už máme pre každý lúč nájdené, ktorý trojuholník pretína treba ešte skontrolovať, či neexistuje viacero lúčov ktoré pretínajú ten istý trojuholník. Táto kontrola prebieha tak, že sa prejdu všetky lúče a keď sa pri jednom trojuholníku nachádza viacero lúčov, tak sa vyberie ten najbližší.

5.2 Kontrola správnosti nájdených bodov prekrytia

Postup z predošlého kroku hľadá pretnutia trojuholníkov referenčného modelu so smernicami bodov nereferenčného modelu, avšak treba myslieť aj na pretnutia trojuholníkov, ktoré nemusia byť len z prekrytia (obr. 2.6). Tento problém sa dá vyriešiť jednoducho a to tak, že sa najskôr nájdu všetký pretnutia trojuholníkov, potom sa urobí priemer vzdialeností všetkých bodov od týchto pretnutí a nakoniec sa vymažú vzdialenosti, ktoré su väčšie ako n-tý násobok priemeru. Čím menšie je číslo n, tým väčšia je šanca, že sa v nasledujúcom kroku nepoposúvajú nesprávne body (obr.

2.2).



Obr. 5.6: Ukážka pretnutia lúča s trojuholníkom ktorý sa nenachádza v prekrytí .

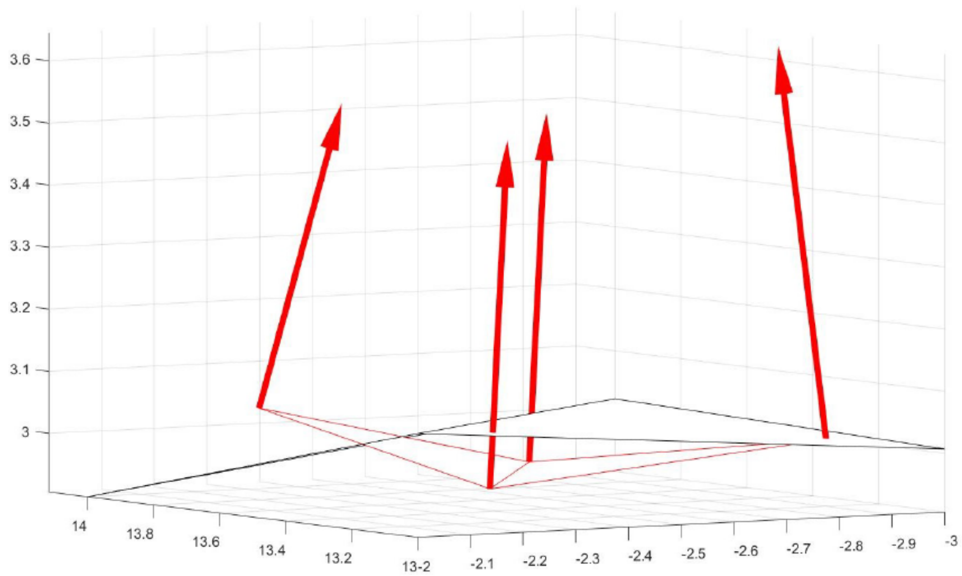
5.3 Spriemerovanie bodov v prekrytí

Keďže chceme spájať 3D modely ľudských časti tela, ktore sa aj pri dvoch rôznych skenoch v krátkom časovom rozmedzí môžu zmeniť, napr. napnutie svalu alebo mierne pohnutie, nebudú prekrytia úplne totožné (obr. 5.7)[21]. Keď chceme mať čo najrealistickejšiu repliku skenovanej časti tela, tak musíme nájdené body prekrytia spriemerovať. Priemerovanie týchto bodov sa uskutočnuje tak, že keď sa pre daný bod referenčného modelu nájde priesečník lúča smerujúceho z toho bodu s trojuholníkom nereferenčného modelu a bod sa posunie po smernici do polovice vzdialenosti, ktorá je medzi bodom a priesečníkom (obr. 5.8)[21]. Kvôli čo najvyhladenejšiemu povrchu musíme spriemerovať body pre oba modely [21].

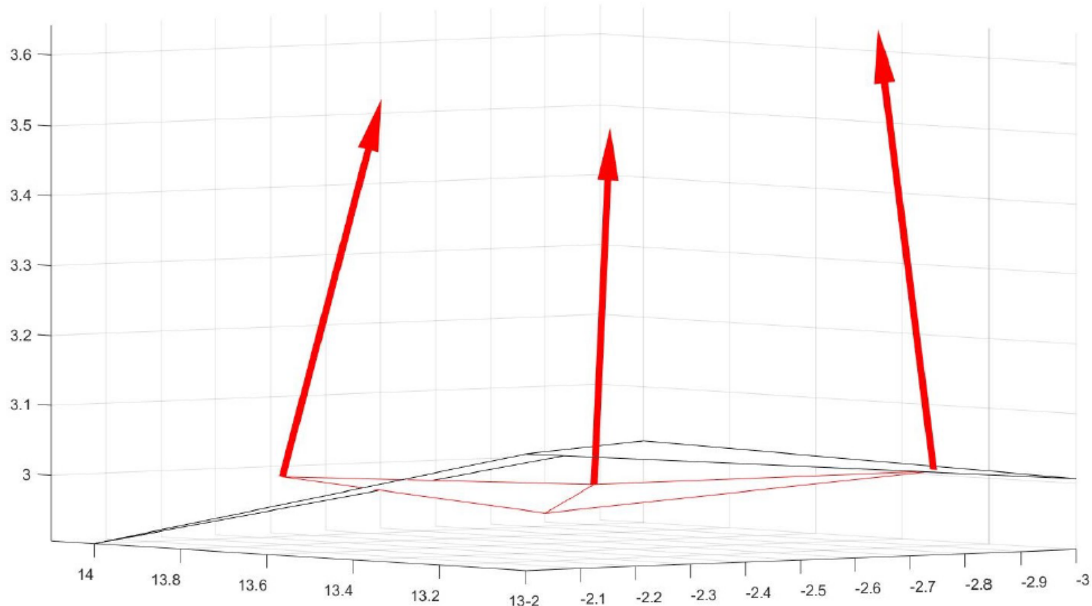
Vo východiskovej práci tento algoritmus fungoval správne, takže ho nebol dôvod ho meniť.

5.4 Vytvorenie nových trojuholníkov v prekrytí

Po priemerovaní bodov nám vznikol povrch, kde sa jednotlivé trojuholníky jednotlivých modelov navzájom prekrývajú. To vzniklo z toho dôvodu, že body sú síce



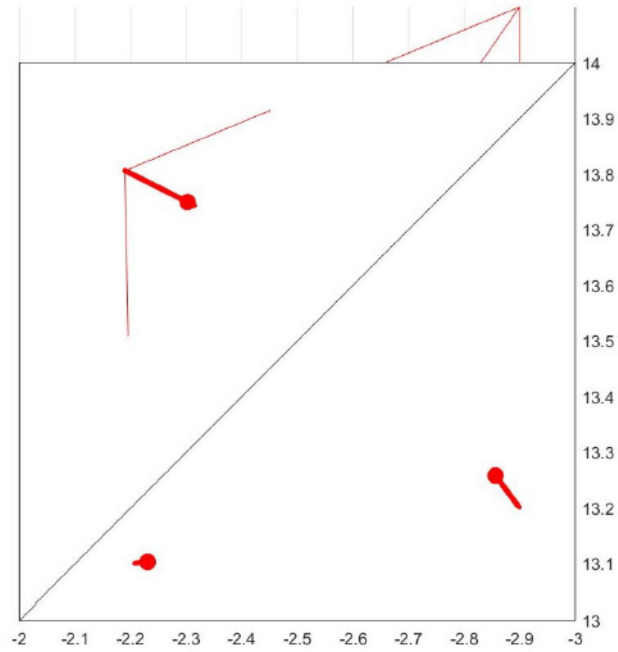
Obr. 5.7: Ukážka bodov prekrytia pred priemerovaním bodov[21].



Obr. 5.8: Ukážka bodov prekrytia po priemerovaní bodov referenčného modelu [21].

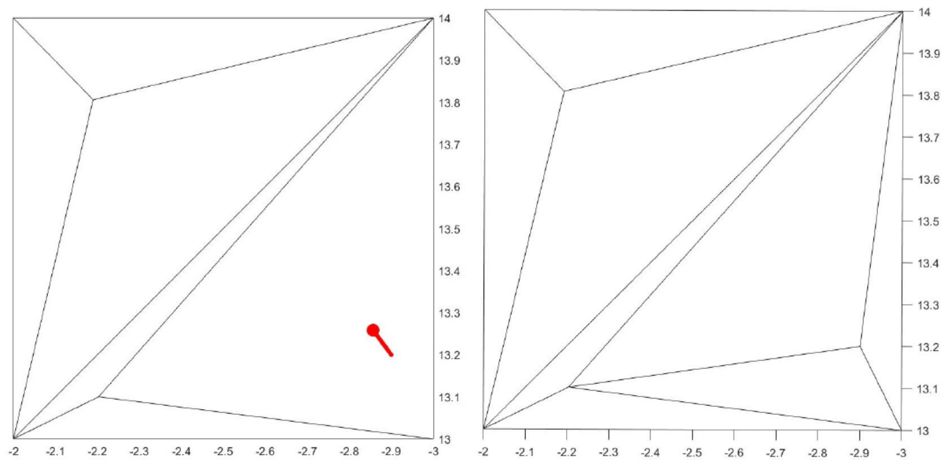
poposúvané, ale v oblasti prekrytia sa nachádzajú dve plochy z oboch modelov. Túto situáciu môžeme vidieť na obrázku 5.9.

Pre vytvorenie súvislého a jedninečného povrchu je treba zvoliť jeden z modelov ako referenčný. Ako prvé sa odstránia všetky trojuholníky v nereferenčnom modeli ktoré obsahujú body nájdené v prvom kroku. V prvom kroku sa tak isto ku týmto



Obr. 5.9: Plochy pred pretrojuholníkovaním[21].

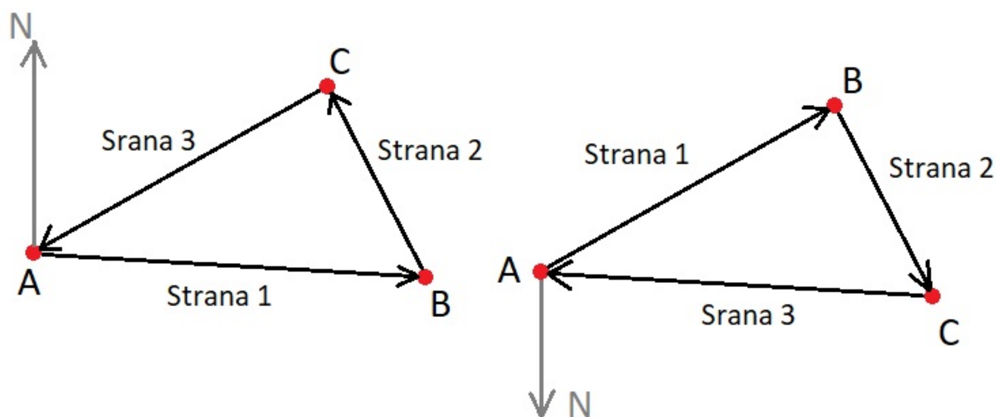
bodom v nereferenčnom modeli priradí aj trojuholník referenčného modelu, ktorý pretla smernica bodu. Potom sa každý trojuholník referenčného modelu, ku ktorému bol priradený bod nereferenčného modelu rozdelí na tri nové trojuholníky ktoré budú mať jeden spoločný vrchol a to bod nereferenčného modelu(obr. 5.10) [21].



Obr. 5.10: Pretrojuholníkovaný povrch[21].

Teoretické riešenie je vo východiskovej práci vymyslené dobre. Avšak pri tvo-

rení nových trojuholníkov sa nedbalo na poradie indexov bodov v novovytvorenom trojuholníku. Na poradí indexov bodov v trojuholníku záleži kvôli geometrickému pricípu počítania normál, kde normála (čo je vektor kolmý na plochu) trojuholníka sa vypočíta vektorovým súčinom prvej a poslednej hrany trojuholníka. Na obrázku 5.11 môžeme vidieť, že keď zameníme strany v trojuholníku (zmena smeru rotácia trojuholníku) tak sa zamení aj prvá a posledná strana a teda vektorový súčin vyjde v opačnej orientácii. Geometrické riešenie tohto problému je že najskôr sa vypočíta normála základného trojuholníka (miesto ktorého sa vytvoria tri nové trojuholníky) a potom sa pred pridaním nového trojuholníka výpočíta skalárny súčin (ang. dot product) základného a novovytvoreného trojuholníka. Keď je výsledok skalárneho súčinu kladný, tak normály trojuholníkov smerujú rovnakým smerom, a keď je skalárny súčin záporný, tak sa len vymenia body B a C novovytvoreného trojuholníku.



Obr. 5.11: Ukážka obrátenia normály so zmenou rotácie trojuholníku.

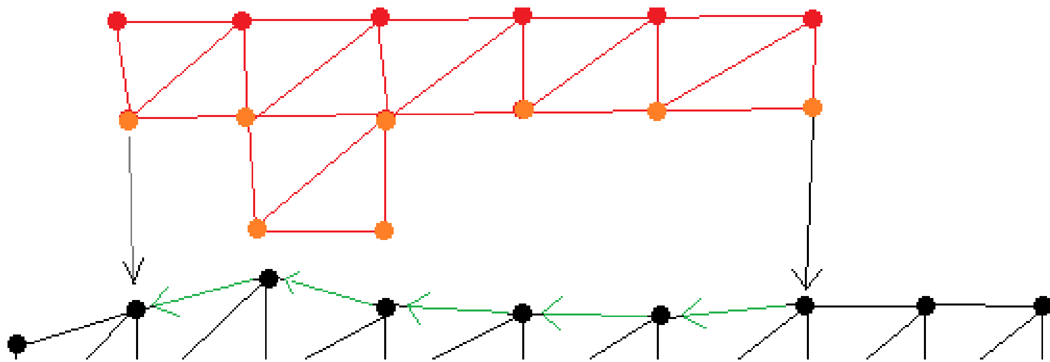
5.5 Nájdenie bodov na vzniknutej medzere

Medzera na konci referenčného modelu vzniká v nereferenčnom modeli a to z dôvodu odstráňovania plochy v prekrytí z nereferenčného modelu. V kroku odstraňovania nastane situácia, kde sa odstránia aj trojuholníky ktoré sa nachádzahú v prekrytí len čiatočne. Túto situáciu môžete vidieť na obázku 2.2.

Princíp nájdenia medzery je podobný ako v práci [19]. Medzera sa nájde tak, že ešte pred odstránením trojuholníkov z nereferenčného modelu sa pre každý bod vypočíta počet trujholníkov v ktorých sa daný bod nachádza. Potom sa vymažú trojuholníky v prekrytí a následne sa urobí rovnaký výpočet s novym modelom. Dané zoznamy sa porovnajú a hľadajú sa tie body pri ktorých sa zmenil počet

trojuholníkov. Zmena by sa mala týkať spravidla dvoch typov bodov, a to body prekrytia v ktorých by mal byť výsledný počet trojuholníkov, v ktorých sa dané body nachádzajú, nulový a hľadané body nereferenčného modelu na jednej hrane medzery.

Nájdením bodov v ktorých bol zmenený počet trojuholníkov a vynechaním bodov, ktoré nepatria do žiadneho trojuholníka by sa mohlo zdať, že výsledné body sú len body na hrane nereferenčného modelu (obr. 5.12, oranžové body), no môže sa stať, že pri vymazávaní prekrytia sa nenájdu všetky body prekrytia a tým pádom sa nemusia vymazať ani všetky trojuholníky nereferenčného modelu, čo spôsobí že pri porovnávaní počtu trojuholníkov do ktorých body patria sa nájdu aj body prekrytia, ktoré sa v predošlom kroku nevymazali. Tento problém sa dá vyriešiť tak, že ku každému nájdenému bodu sa nájde počet nájdených bodov, ktoré s daným bodom susedia. To či dva body susedia určuje to, či sa nachádzajú v spoločnom trojuholníku. Po spočítaní susedov sa nájdu body ktoré majú iba jeden susedný bod a tie body sú okrajové body hrany medzery na nereferenčnom modeli.



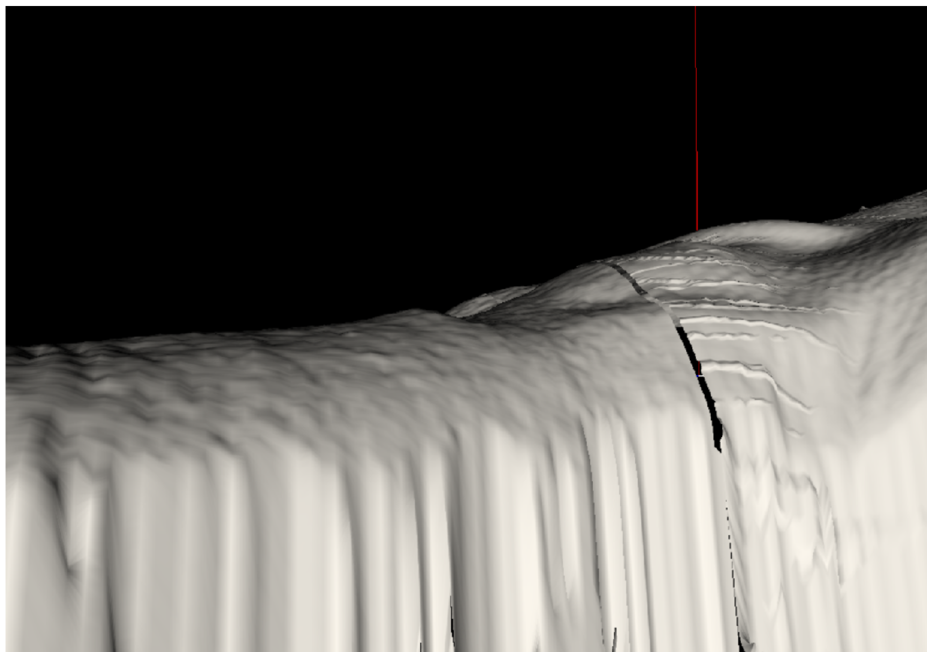
Obr. 5.12: Ukážka princípu nájdenia medzery.

Po nájdení okrajových bodov medzery nereferenčného modelu sa okrajové body medzery referenčného modelu (okrajové body druhej hrany medzery) nájdu jednoducho tak, že sa nájdu body z referenčného modelu, ktoré sú najbližšie k nájdeným bodom (obr. 5.12, čierne šípky).

Zvyšné body (body medzi okrajovými bodmi) sa nájdu tak, že sa vyberie jeden okrajový bod ako referenčný, ku nemu sa nájdu všetky susediace body a bod, ktorý sa bude nachádzať v menej ako piatich trojuholníkoch (kontrola hrany) a zároveň smer pohybu po hrane bude smerovať ku koncovému bodu na opačnej strane, je nasledujúcim bodom na hrane medzery. Tento nájdený bod sa stane referenčným a algoritmus sa bude znova opakovať až kým sa referenčným bodom nestane okrajový bod na opačnej strane. Takto sa nájdu body medzery na oboch hranách. Princíp je grafický znázornený na obrázku 5.12 zelenými šípkami.

5.6 Rozšírenie medzery

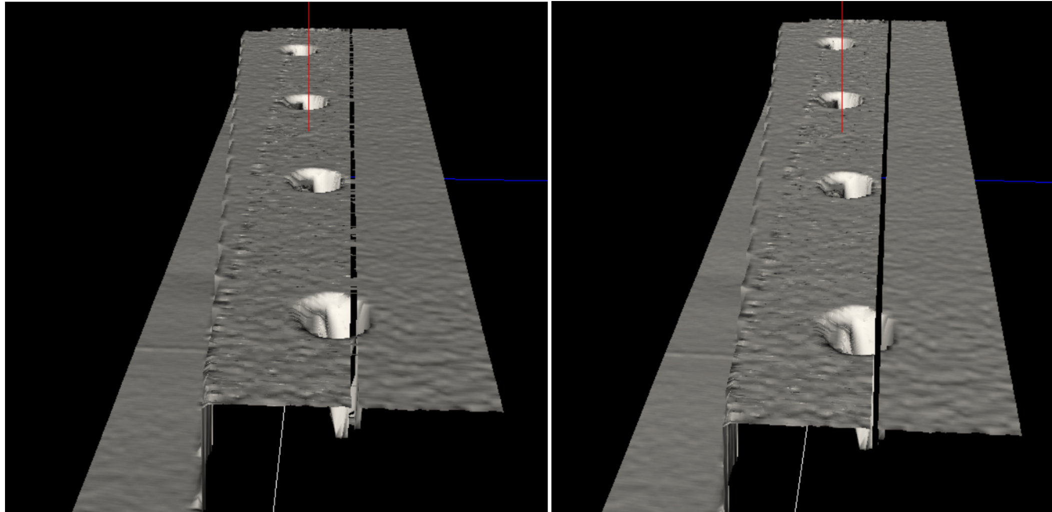
Problem so šírkou medzery nastáva vtedy, keď lúč z bodu referenčného modelu nemieri presne kolmo na nereferenčný model. Vtedy sa stane to, že sa medzera zúži alebo niekedy až jeden model vniká pod druhý ako na obrázku 5.13. Vtedy by nebolo vhodné medzeru hneď pretrojuholníkovať, pretože by na mieste medzery vznikol neprirodzený skok. Vhodnejším riešením bude medzeru rozšíriť a až po rožšírení použiť algoritmus pretrojuholníkovania.



Obr. 5.13: Detail na zúženú medzeru.

Rožsirovanie medzery funguje tak, že najskôr musíme mať nájdené body na hranách medzery, ktoré sa nájdu v predošlom kroku, potom sa hrana na ktorej sa nachádzajú výseky určíme ako referenčnú. Ku každému bodu na referenčnej hrane sa nájde najbližší bod z nereferenčnej hrany a vypočíta sa vzdialenosť medzi týmito bodmi. Tieto vzdialenosti sa spriemerujú a nájdu sa všetky páry bodov, ktoré sú pri sebe veľmi blízko v porovnaní s priemerom. Z týchto párov bodov sa vytiahnú všetky body referenčnej hrany, ku ním sa nájdu všetky trojuholníky pri ktorých sú vrcholom. Tieto body aj trojuholníky sa z modelu vymažú. Ďalej sa znova budú musieť nájsť všetky body na referenčnej hrane medzery a tie sa nájdu rovnako ako už bolo popísané v predošlom kroku.

Porovnanie medzi rožšírenou a nerozšírenou medzerou môžete vidieť na obrázku 5.14.

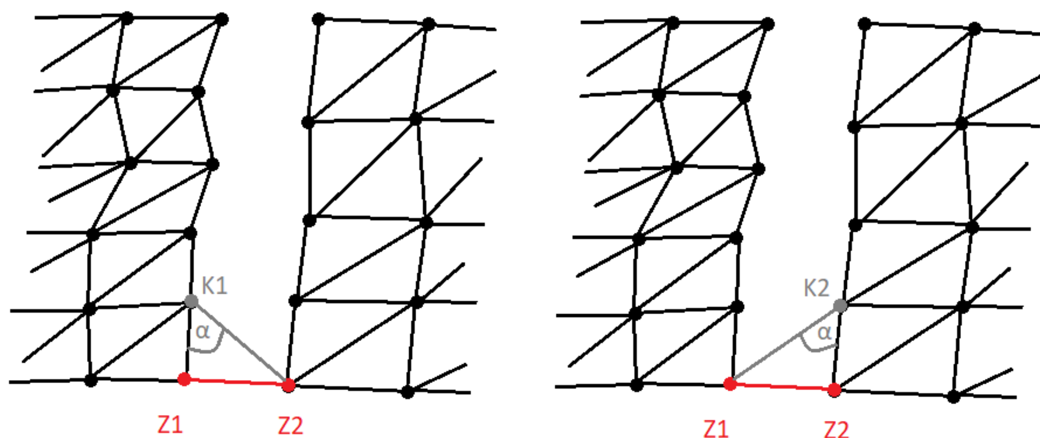


Obr. 5.14: Detail rozšírenej a nerozšírenej medzery.

5.7 Vyplnenie medzery novými trojuholníkmi

Keď už máme nájdené všetky okrajové body medzery, tak vyplnenie medzery trojuholníkmi je jednoduché. Algoritmus je navrhnutý podobne ako v prácach [19, 26].

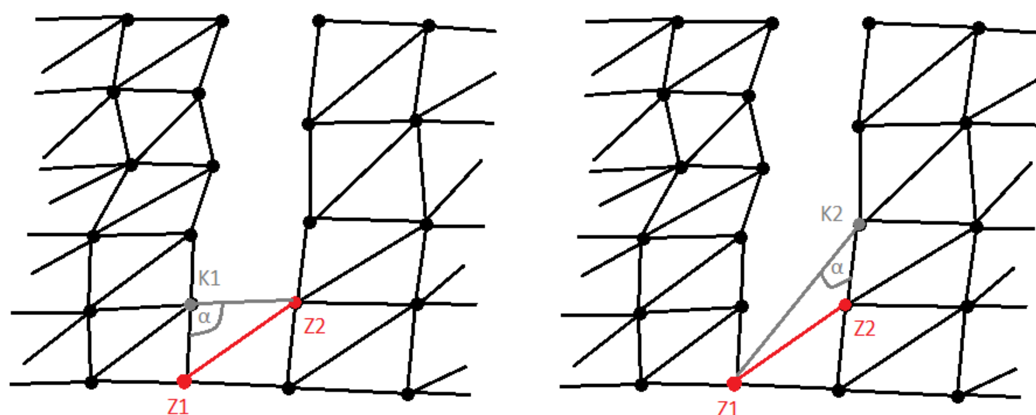
Tento algoritmus je založený na nájdenom zozname bodov v presnom poradí, ako za sebou na hranách medzery postupujú. Algoritmus funguje rekurzívne, a to tak, že v každom kroku máme určenú jednu hranu trojuholníku, ktorá je určená dvomi zdrojovými bodmi. V prvom kroku sú tieto zdrojové body počiatočnými bodmi zoznamu medzery a potom sa menia podľa toho aký trojuholník sa do medzery pridá (na obrázku 5.15 označené písmenom Z).



Obr. 5.15: Výber nového trojuholníku v medzere.

Ďalej sa ku zdrojovým bodom hľadá tretí bod trojuholníku tak, že najskôr sa zvolia dvaja kandidáti. Kandidáti sa volia tak, že sa zoberú následujúce body v zoznamoch bodov na hrane medzery. Potom sa vypočíta uhol α v kandidátskom trojuholníku, na obrázku 5.15 označený sivou farbou. Keď máme oba uhly, tak novým trojuholníkom v medzere sa stáva ten, pri ktorom mal uhol väčšiu hodnotu. Potom sa celý proces opakuje. Novým zdrojovým bodom sa stane kandidát z minulého procesu (obrázok 5.16).

Celý postup sa opakuje, až kým sa zdrojovou hranou nestane hrana medzi poslednými bodmi na hranách medzery.



Obr. 5.16: Druhý krok výberu nového trojuholníku v medzere.

5.8 Vymazanie nepoužitých bodov z modelu.

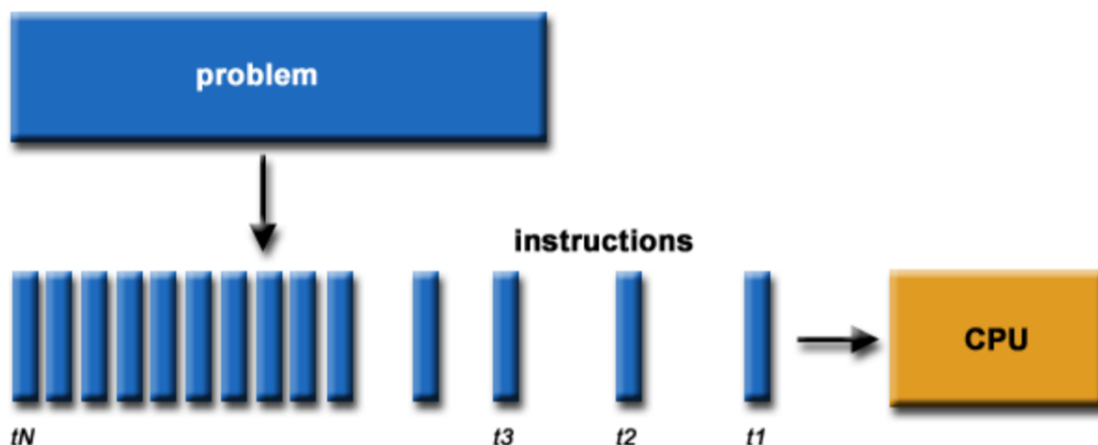
Ako už bolo spomenuté v minulom kroku, tak pri hľadaní prekrytia sa nemusia nájsť všetky body prekrytia nereferenčného modelu. Tým pádom vedľa modelu ostanú malé plochy, ktoré tam nepatria.

Tieto malé plochy sa vymažú jednoducho tak, že sa zoberie ľubovolný bod z referenčného modelu, ku nemu sa nájdu všetky susedné body, teda body ktoré sa nachádzajú s daným bodom v trojuholníku a tak sa pokračuje, až kým sa nájdené body nebudú rovnať v dvoch po sebe idúcich cykloch. Potom sa len nájdu body, ktoré do nájdenej celistvej plochy nepatria, ku ním sa nájdu všetky trojuholníky do ktorých dané body patria a body aj trojuholníky sa vymažú.

Pri mazaní bodov si treba dávať pozor na zmenu indexov v trojuholníkoch, čiže po každom zmazanom bode bude treba prejsť všetky trojuholníky modelu a tam kde je hodnota indexu bodu vyššia ako index zmazaného bodu bude treba zmenšiť o jeden.

6 Prepracovanie algoritmu pre použie na GPU

Tradične sa softvér spracováva sekvenčne (sériové spracovanie), kde je problém rozdelený na sériu inštrukcií ktoré sa vykonávajú postupne jedna po druhej v jednom procesore. Čo znamená, že v jednom momente môže byť vykonaná iba jedna inštrukcia (obrázok 6.1)[7].



Obr. 6.1: Sériové počítanie[7].

Pre ušetrenie času sa pre riešenie viac komplexných alebo väčších algoritmov používa paralelné počítanie. Paralelné počítanie sa v [7] predstavuje ako súčasné využitie viacerých zdrojov na vyriešenie problému:

- Problém je rozdelený na menšie nezávislé bloky, ktoré sa môžu vykonávať súčastne.
- Každý blok je ďalej rozdelený na sériu inštrukcií.
- Inštrukcie z každého bloku sa vykonávajú šúčastne na viacerých procesoroch.

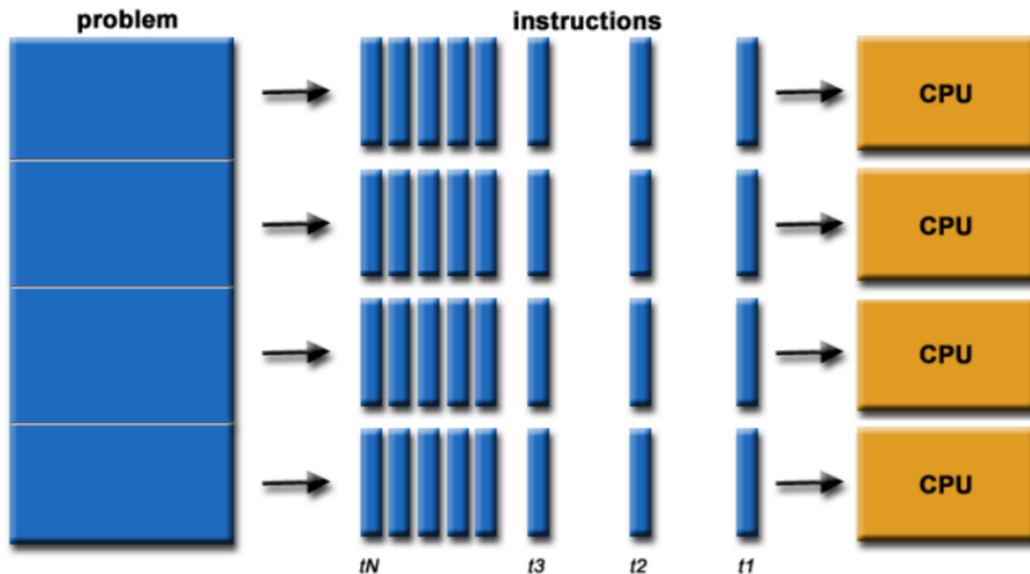
Tento princíp je ukázaný na obrázku 6.2.

Pre použitie pararelizmu v našej práci sme museli najskôr nájsť časovo najnáročnejší algoritmus, vymyslieť ako by sa dal tento algoritmus počítať paralelne a nakoniec modifikovať tento algoritmus pre výpočet na grafickej karte.

Grafickú kartu sme si na paralelné počítanie vybrali z dôvodu veľkého rozdielu v počte jadier oproti CPU. Pre príklad použijeme špecifikácie notebooku, ktorý bol použitý na daný výpočet. Používaná grafická karta sa nazýva NVIDIA GeForce GTX 950M a má 640 jadier [1], čo je radovo v stovkách viac ako má Intel Core i5-7300HQ ktorý má len 4 jadrá [2].

Postup pararelizmu je v práci rozdelený do dvoch krokov:

1. Návrh paralelného výpočtu najnáročnejšej časti



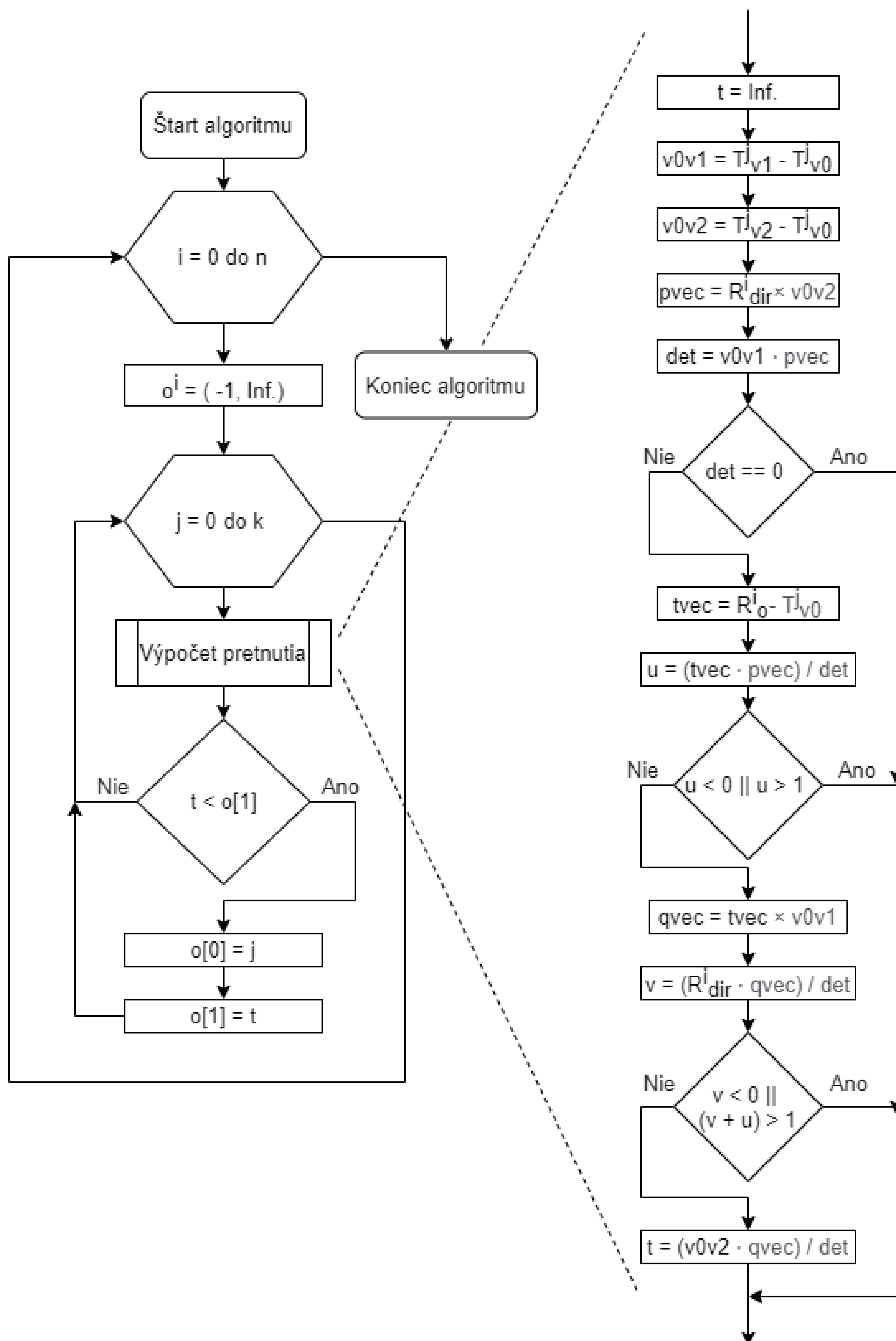
Obr. 6.2: Paralelné počítanie[7].

2. Využitie GPU pre daný paralélny výpočet

6.1 Návrh paralelného výpočtu

Najviac výpočetne náročnou časťou tejto práce je hľadanie priesečníku všetkych lúčov so všetkými trojuholníkmi a preto sme sa rozhodli túto časť vypočítať na grafickej karte, kde sme využili jednu z vlastností tohto algoritmu, kde jednotlivý výpočet lúča nezávisí na ostatných výpočtoch. Daný algoritmus výpočtu prekrytia vychádza z [23], ktorého princíp je opísaný v 2.1.4, je ukázaný vo vývojovom diagrame na obrázku 6.3, kde medzi vstupné hodnoty algoritmu patrí pole lúčov (vo vývojovom diagrame označené R) o veľkosti n, kde každý má dva vektory a to pozíciu (R_o) a smer (R_{dir}). Ďalšou vstupnou hodnotou je pole trojuholníkov v prekrytí (označené T) o veľkosti k, kde každý trojuholník obsahuje tri vektory, ktoré označujú pozíciu bodov v trojuholníku (R_{v0}, R_{v1}, R_{v2}). Poslednou vstupnou hodnotou je alokované pole n-tíc (označené ako o) v rovnakej veľkosti ako počet lúčov (teda n), do ktorého sa podľa indexu lúča ukladá najbližší trojuholník (nultý index n-tice) a vzdialenosť priesečníku daného trojuholníku od počiatku lúča.

Z vývojového diagramu na obrázku 6.3 je vidieť, že pre každý lúč sa musí vypočítať priesečník s k trojuholníkmi. To znamená, že keď chceme tento algoritmus vypočítať paralelne, tak musíme lúče rozdeliť do blokov a dané bloky počítať nezávisle od seba na samostaných jadrách.



Obr. 6.3: Algoritmus pre výpočet najbližšieho priesečníku n lúčov s k trojuholníkmi.

6.2 Použitie GPU

Pre prácu s GPU sme použili sadu knižníc CUDAfy, ktoré sú podrobne popísané v [14]. CUDAfy umožnuje z programového rozhrania *Microsoft .NET* programovanie grafických kariet pomocou NVIDIA CUDA, alebo pomocou OpenCL. Nástroj CUDAfy je rozdelený do štyroch častí:

1. Cudafy Translator, ktorý konvertuje .NET kód do CUDA, alebo OpenCL C kódu.
2. Cudafy Library, ktorá obsahuje CUDA, alebo OpenCL podporu pre .NET.
3. Cudafy Host. Používa sa ako obal (ang. wrapper) hostiteľského zariadenia.
4. Cudafy Math. Matematické operácie, ktoré sa môžu používať vo funkciách spustených na grafickej karte.

Programovanie grafických kariet pomocou CUDAfy je jednoduché. V prvom rade je potrebné napísať daný algoritmus v OpenCL štandarde, kde sa vo funkciách nemôže alokovať nová pamäť a v štruktúrach sa nemôžu používať žiadné metódy. Následne sa pri spustení programu zavolá Cudafy Translator, ktoré dané funkcie napísané v .NET kóde konvertuje na OpenCL kód. Pred volaním funkcie na GPU je potrebné alokovať priestor na GPU do ktorého musíme nakopírovať premenné s ktorými chceme počítať. Potom sa pri volaní danej funkcie, ktorú chceme vypočítať na GPU (v OpenCL nazývaná *Kernel*), musí určiť rozmer indexového priestoru a rozmer pracovných skupín ktoré sú opísané v podsekcii 3.1.2. Po zavolaní funkcii na GPU sa do parametrov funkcie pridá objekt *GThread* pomocou ktorého zisťujeme index aktuálne používanej pracovnej skupiny a index aktuálne používanej pracovnej jednotky v danej skupine, vdaka čomu vieme algoritmus *Kernelu* riadiť paralelne. Pre lepšie pochopenie si tento princíp ukážeme na nasledujúcom príklade, kde si vytvoríme dvojdimenzionálne pole v ktorom pomocou CUDAfy inkrementujeme každý prvok poľa o jeden (na GPU):

```
1  using System;
2  using System.Collections.Generic;
3  using Cudafy;
4  using Cudafy.Host;
5  using Cudafy.Translator;
6
7  namespace CudafyPriklad
8  {
9    class Program
10   {
11     private const int X = 1024;   // rozmer prvej dimenzie
12     private const int Y = 1024;   // rozmer druhej dimenzie
13
14     [Cudafy] // potrebné označenie pre Cudafy Translator
```

```
15      public static void test_kernel
16      (GThread thread, int[,] array, int X, int Y)
17      {
18        // získanie aktuálneho indexu prvej dimenzie
19        int id_x = thread.threadIdx.x +
20                thread.blockIdx.x * thread.blockDim.x;
21        // získanie aktuálneho indexu druhej dimenzie
22        int id_y = thread.threadIdx.y +
23                thread.blockIdx.y * thread.blockDim.y;
24        // kontrola prekročenia velkosti pola
25        if (id_x < X && id_y < Y)
26        {
27          // inkrementácia aktuálneho prvku o jeden
28          array[id_x][id_y] += 1;
29        }
30      }
31
32      static void Main(string[] args)
33      {
34        // nastavenie OpenCL štandardu pre GPU
35        CudafyModes.Target = eGPUType.OpenCL;
36        // nastavenie preklad .NET kódu na OpenCL kód
37        CudafyTranslator.Language = eLanguage.OpenCL;
38        // konvertovanie funkcie test_kernel do OpenCL kódu
39        CudafyModule km = CudafyTranslator.Cudafy();
40        // vytvorenia rozhrania s OpenCL zariadením
41        GPGPU gpu = CudafyHost.GetDevice(CudafyModes.Target,
42              CudafyModes.DeviceId);
43        // načítanie modulu do zariadenia
44        gpu.LoadModule(km);
45
46        // vytvorenie prázdneho pola
47        int[,] array = new int[X, Y];
48        // alokovanie pamati pre dané pole na používanom zariadení
49        int[,] dev_array = gpu.Allocate<int>(X, Y);
50        // nakopírovanie pola z pamati programu do pamati
51        // používaného zariadenia
52        gpu.CopyToDevice(array, dev_array);
53        // spustenie funkcii na používanom zariadení
54        gpu.Launch(new dim3(1, 1), new dim3(X, Y)).
55            test_kernel(dev_array, X, Y);
56        // alebo metoda pomocou prerozdelenia pracovných jednotiek
57        // do rovnako velkých blokov pracovných skupín, čo sa robí z
58        // dovodu obmedzenej velkosti jednej pracovných skupiny
59        // (napr. rozdelenie do štyroch pracovných skupin):
60        // gpu.Launch(new dim3(4, 4), new dim3(X/4, Y/4))
61        //     .test_kernel(dev_array, X, Y);
```

```
62
63        // prekopírovanie inkrementovaného pola z používaného
64        // zariadenia naspať do pamati programu
65        gpu.CopyFromDevice(dev_array, array);
66      }
67    }
68 }
```

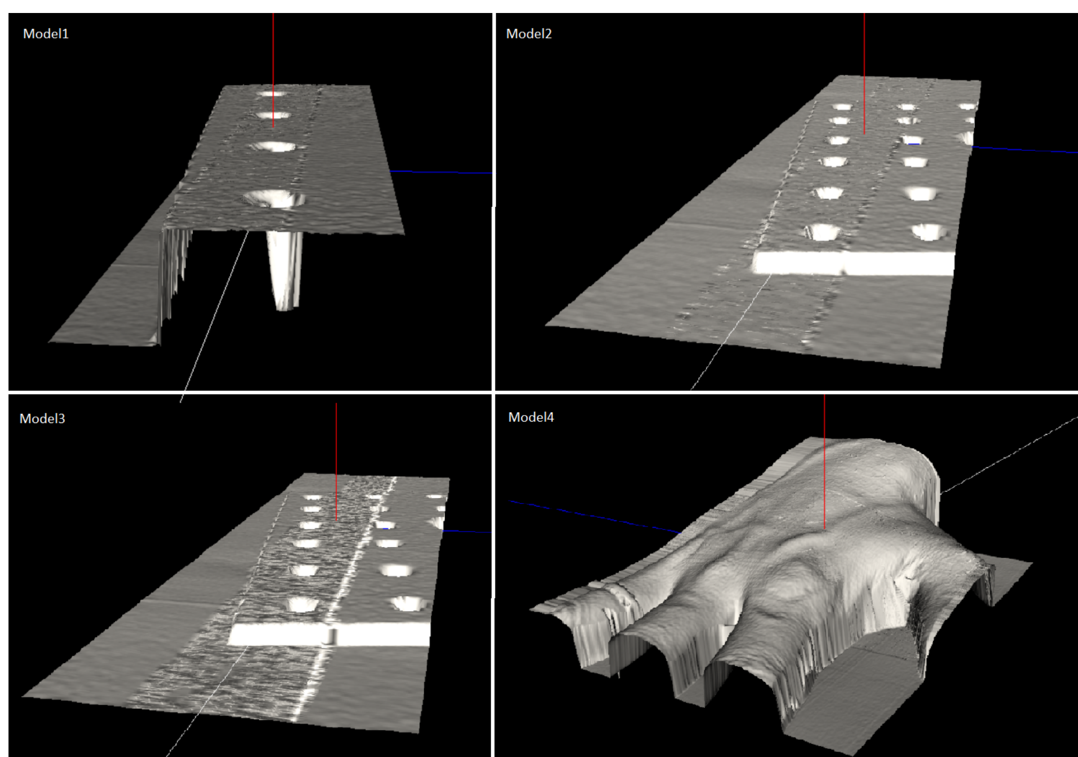
Podobný princíp využívame aj pre výpočet všetkých pretnutí lúčov s trojuholníkmi. Pri návrhu budeme vychádzať z algoritmu opísaného na obrázku 6.3, kde n sme si určili ako velkosť prvej dimenzie a k ako veľkosť druhej dimenzie. Následne si vo funkcií miesto cyklov potrebujeme vypočítať premenné i a j, tak že sa získa aktuálny index dimenzií rovnako ako v kóde vyššie a pracovné jednotky sa do pracovných skupín rozdelia tak, aby indexy prvej dimenzie prešli všetky lúče a indexy druhej dimenzie zase všetky trojuholníky.

Toto riešenie je správne, avšak si treba dávať pozor na veľkosť dat privedenú do jedného *kernelu*, pretože *kernel* má len obmedzený počet inštrukcií ktoré môže vykonať, preto je vhodné data rozdeliť do častí a *kernel* volať postupne po čiastiach až kým sa neprejdú všetky data. V našej práci je rozdelenie navrhnuté tak, že vychádzame z čiastkových blokov, ktoré sú opísané vyšie v sekcií 5.1.3, tak, že do jedného *kernelu* posielame vypočítať priesečníky lúčov s trojuholníkmi iba jedného daného bounding boxu pričom keď je v danom boxe príliš veľa trojuholníkov tak sa rozdelia aj oni. Postup tohoto algoritmu je zhrnutý v nasledujúcich krokoch:

- Nakopírovanie dvoj-dimenzionálneho poľa do hosťovského OpenCL zariadenia, kde prva dimenzia určuje index boxu a v druhej sa nachádzajú všetky indexy lúčov v danom boxe.
- Nakopírovanie dvoj-dimenzionálneho poľa do hosťovského zariadenie, kde prvá dimenzia určuje index boxu a v druhej sa nachádzajú všetky indexy trojuholníkov pre daný box.
- Nakopírovanie všetkych lúčov, trojuholníkov do GPU.
- Alokovanie miesta na GPU pre ukladanie lúčov a trojuholníkov o takej veľkosti, aby sa neprekročil maximálny počet inštrukcií v jednom *kerneli*.
- Alokovanie dvoch polí na GPU o veľkosti počtu všetkých lúčov v prekrytí do ktorých sa bude dávať index najbližšieho trojuholníka a vzialenosť počiatku lúča od priesečníka indexovaných podľa indexu lúča.
- Cyklické asynchrónne (neblokujúce) kopírovanie indexov lúčov a trojuholníkov z hosťovského OpenCL zariadenia do GPU a následné asynchrónne spustenie *kernelu* s danými indexmi.
- Po prejdení všetkých bonding boxov sa GPU zosynchrozuje, čo znamená, že sa počká na dokončenie všetkých *kernelov* a polia s obsahom najblizších trojuholníkov a vzdialeností sa prekopírujú naspät do .NET aplikácie.

7 Experimentálne overenie

Cieľom našej práce bolo pokračovať v práci Mareka Lampáša, čo znamená odstrániť všetky nedostatky. Výsledky spojených modelov môžeme vidieť na obr. 7.1. Modely vyhotovené algoritmom z bakalárskej práce od Mareka Lampáša sa nachádzajú na obrázku 5.1. Hlavným problémom výsledných modeloch je zlé tienovanie plôch. Toto mohlo nastať z dôvodu nesprávneho vypočítania normál trojuholníkov.



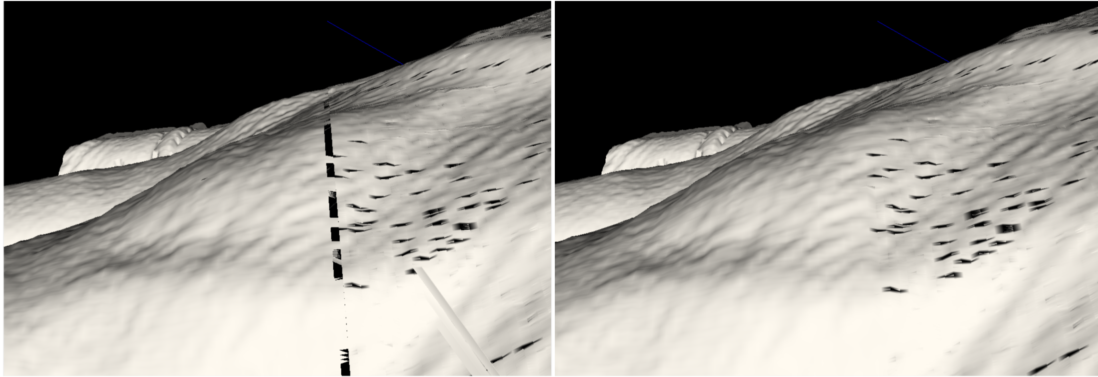
Obr. 7.1: Výsledne objekty.

7.1 Porovnanie s východiskovou prácou

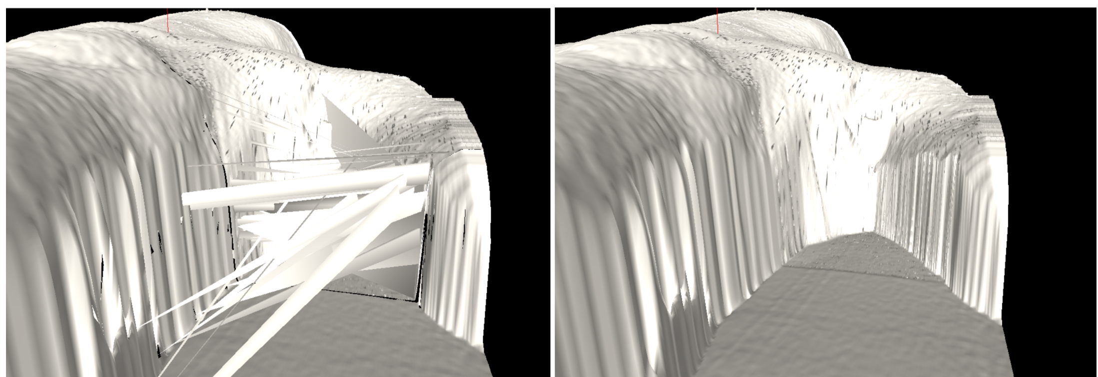
Prvým spomenutým problémom práce Mareka Lampáša bola medzera v strede výsledneho modelu, zaplnenie medzery je vidieť už na referenčných modeloch na obrázku 7.1. Detail na pretrojuholníkovanie medzery môžete vidieť na obr. 7.2.

Taktiež sa podarilo odstrániť aj problém s nesprávnym priemerovaním bodov (obr. 6.3). A aj odstránenie plôšok v poslednom kroku (obr. 6.4).

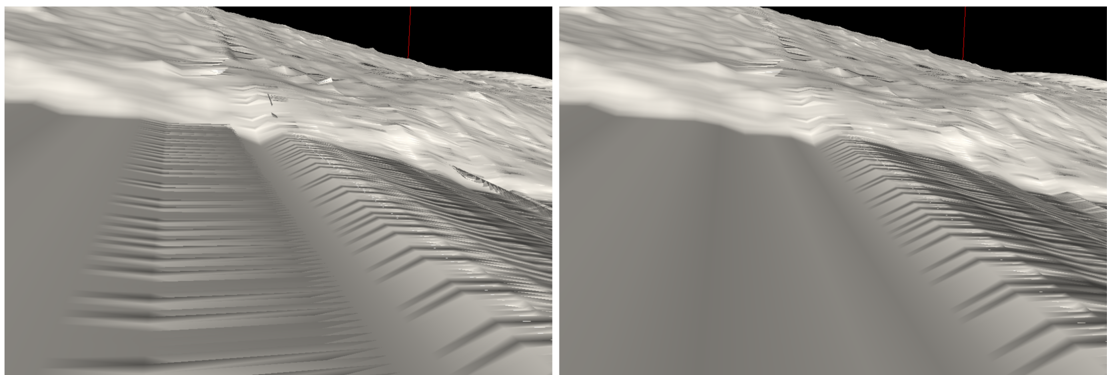
V poslednom rade sa pozrieme na dobu trvania spájania modelov, pretože aj toto bolo jedným z dôležitých cieľov práce. Kde v tabuľke 7.1 vidieť rozdiel medzi časmi východzej a tejto práce. V prvom a druhom modeli je rozsiahlosť výpočtov takmer



Obr. 7.2: Detail na medzeru.



Obr. 7.3: Detail na úsek modelu, kde je vidieť zlepšenie algoritmu priemerovania bodov.

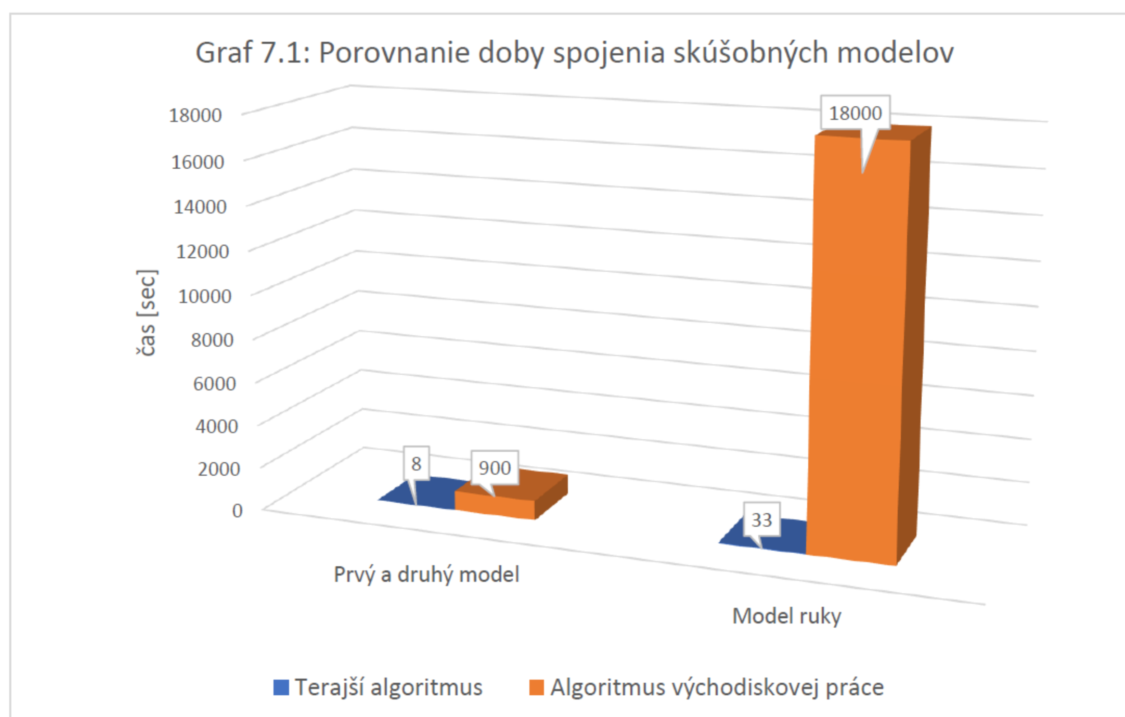


Obr. 7.4: Detail na úsek modelu, kde je vidieť opravu plôch nad modelom.

identická, čomu odpovedá aj rovnaký čas potrebný na spojenie ktorý vyšiel približne 8 sekúnd. V treťom modeli je rovnako bodov a trojuholníkov ako v predošlích dvoch, akurát objekty majú medzi sebou uhol, čo znamená že sa musí počítať o trochu

Tab. 7.1: Porovnanie času

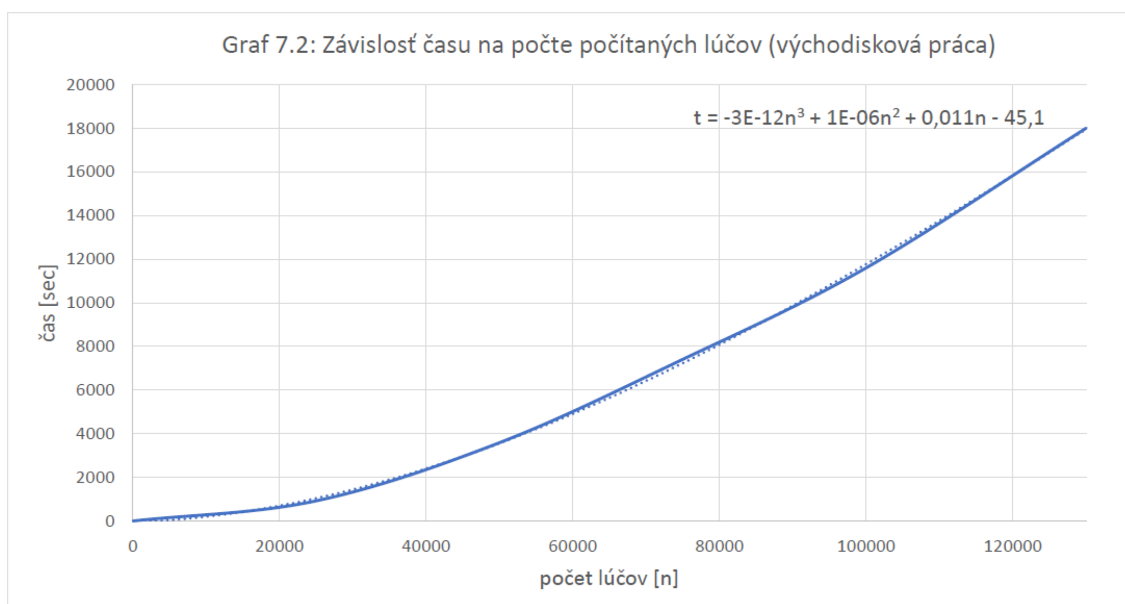
Č.	Počet bodov	Počet trojuholníkov	Čas predtým [sec]	Čas teraz [sec]
1	104 960	203 549	900	8
2	104 960	203 433	900	8
3	104 960	203 893	900	8.5
4	517 120	769 321	18000	33



väčšia plocha prekrytia čiže doba tohto výpočtu trvala 8.5 sekúnd. V poslednom modeli ruky je počet bodov a aj plocha prekrytia najväčšia z čoho vyplýva, že aj čas spojenia bude najväčší a to 33 sekúnd. V porovnaní s východiskovou prácou je to obrovská zmena, ale v tomto smere sa algoritmus bude ešte musieť zdokonaliť, pretože 33 sekúnd na spojenie dvoch čiastkových modelov ruky je stále veľa. Pre lepšie predstavenie sú dané hodnoty vynesené do stĺpcového grafu 7.1.

Časovú náročnosť východiskovej práce podľa počtu počítaných lúčov a trojuholníkov, kde trojuholníkov je dvojnásobok ako lúčov môžete je vynesená do grafu 7.2, ktorý je preložený polynomickou trendovou spojnicou tretieho radu, kde v trendovej rovnici je písmenom t označený výsledný čas v sekundách a písmenom n počet lúčov.

Časovú náročnosť tejto práce podľa počtu počítaných lúčov a trojuholníkov, kde trojuholníkov je dvojnásobok ako lúčov je vynesená do grafu 7.3, ktorý je preložený polynomickou trendovou spojnicou druhého radu, kde v trendovej rovnici je



Tab. 7.2: Porovnanie času výpočtu na CPU a GPU

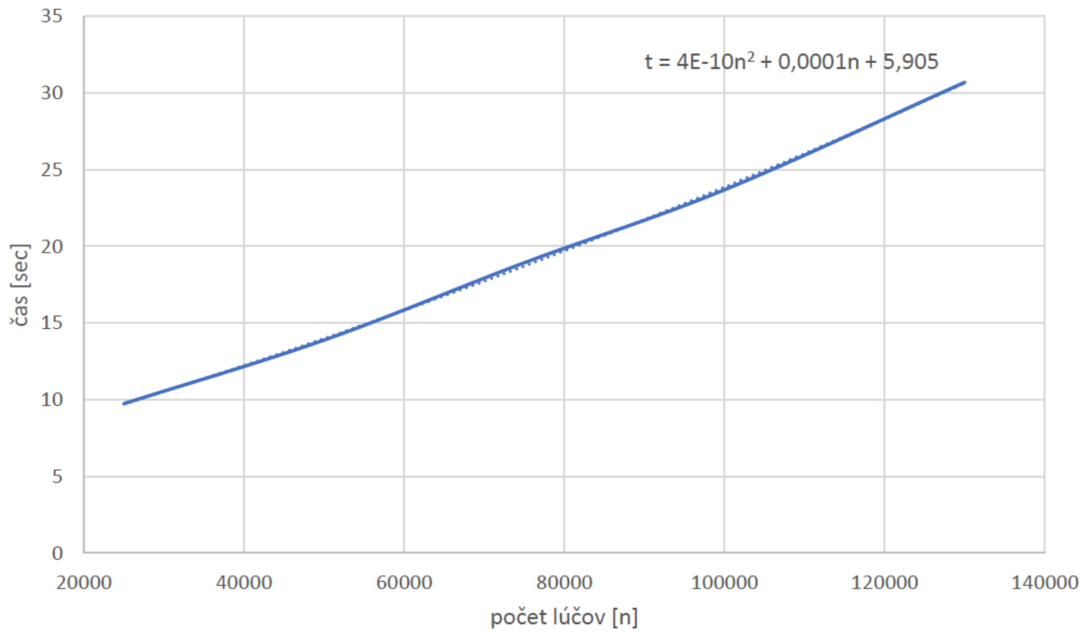
Počet lúčov	Počet trojuholníkov	Čas CPU [sec]	Čas GPU [sec]
25 000	50 000	124	9.8
50 000	100 000	320	13.9
75 000	150 000	537	18.9
100 000	200 000	745	23.7
130 000	260 000	950	30.7

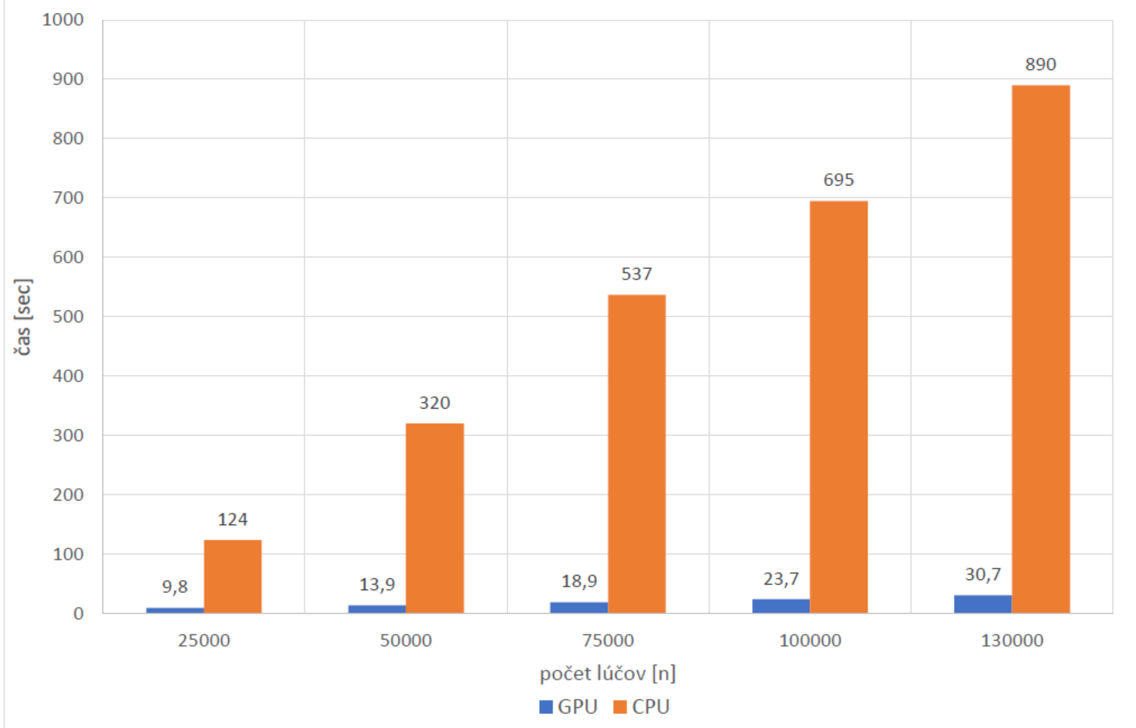
písmenom t označený výsledný čas v sekundách a písmenom n počet lúčov.

Pri porovnaní oboch závislostí je vidieť, že časová náročnosť algoritmu našej práce nestúpa tak razantne s počtom bodov ako východisková. Avšak je tu vidieť ďalší problém nášho algoritmu, kde pri malom počte (blízkemu nule) neklesá doba trvania na nulu, ale ostáva približne na dvoch až troch sekundách, čo je z dôvodu použitia knižníc CUDAfy, ktoré pri inicializácii zaberú dosť času.

7.2 Porovnanie rýchlostí počítania na CPU s GPU

V tejto sekcií si porovnáme časy, keď algormitmus spojenia modelov opísaný v tejto práci spustíme s počítaním priesečníkov lúča s trojuholníkom na CPU a potom na GPU. Výsledky môžete vidieť v tabuľke 7.2 a vynesené do stĺpcového grafu 7.4, kde je vždy počet trojuholníkov dvojnásobný ako počet lúčov.


Graf 7.3: Závislosť času na počte počítaných lúčov (táto práca)
t = 4E-10n2 + 0,0001n + 5,905
čas [sec]
0
5
10
15
20
25
30
35
20000
40000
60000
80000
100000
120000
140000
počet lúčov [n]


Graf 7.4: Porovnanie času výpočtu priesečníkov na CPU a GPU
čas [sec]
0
100
200
300
400
500
600
700
800
900
1000
9,8
124
13,9
320
18,9
537
23,7
695
30,7
890
25000
50000
75000
100000
130000
počet lúčov [n]
GPU
CPU

8 Záver

Práca sa zaoberá návrhom algoritmu, ktorý spája trojrozmerné čiastkové modely, do výsledného celistvého modelu.

Pri testoch na reálnych modeloch sa dokázalo, že algoritmus si dokázal poradiť so všetkými chybami, ktoré sa objavili vo východiskovej práci ktorej autorom je Mareka Lampáš. Taktiež sa nám podarilo skrátiť priebeh výpočtu pri modeli ruky na približne 33 sekúnd, čo je veľký pokrok oproti východiskovej práci, kde výpočet trval 300 minút.

Výsledný povrch po spojení 3D čiastkových modelov má však nedokonalosť a to, že plocha obsahuje trojuholníky, ktoré sú tmavé. Tento problém mohol nastať z viacerých dôvodov, a to buď z toho že po priemerovaní plôch vznikli malé slepé tunely, alebo z toho dôvodu, že pri tvorení nových trojuholníkov v prekrytí sa bod, z ktorého sa tvorili nové trojuholníky, nachádzal presne na hrane starého trojuholníka, čo spôsobilo problém pri počítaní nových normál.

Pokračovanie tejto práce by sa malo najskôr zaoberať opravením tmavých, nesprávne nasvietených plôch a ďalej pracovať na zrýchlovaní algoritmu.

Literatúra

[1] GeForce GTX 950M Dedicated Graphics for Laptops. Library Catalog: www.geforce.com.
URL https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-950m/specifications

[2] Intel® Core™ i5-7300HQ Processor (6M Cache, up to 3.50 GHz) Product Specifications. Library Catalog: ark.intel.com.
URL https://ark.intel.com/content/www/us/en/ark/products/97456/intel-core-i5-7300hq-processor-6m-cache-up-to-3-50-ghz.html

[3] A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.) (Ray-Box Intersection).
URL https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection

[4] Ray Tracing: Rendering a Triangle (Barycentric Coordinates).
URL https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

[5] RoScan – Robotic System Project.
URL http://roscan.ceitec.cz/

[6] R-tree. Máj 2020, page Version ID: 955552664.
URL https://en.wikipedia.org/w/index.php?title=R-tree&oldid=955552664

[7] Barney, B.; aj.: Introduction to parallel computing. *Lawrence Livermore National Laboratory*, ročník 6, č. 13, 2010: str. 10.

[8] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; aj.: The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, Atlantic City, New Jersey, USA: Association for Computing Machinery, Máj 1990, ISBN 978-0-89791-365-2, s. 322–331, doi:10.1145/93597.98741.
URL https://doi.org/10.1145/93597.98741

[9] Bentley, J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM*, ročník 18, č. 9, September 1975: s. 509–517, ISSN 0001-0782, doi:10.1145/361002.361007.
URL https://doi.org/10.1145/361002.361007

[10] Beutelspacher, A.; Albrecht, B.; Rosenbaum, U.: *Projective Geometry: From Foundations to Applications*. Cambridge University Press, Január 1998, ISBN 978-0-521-48364-3, google-Books-ID: I4OqBcaKAJ0C.

[11] Bissyande, T. F.; Sie, O.: *e-Infrastructure and e-Services for Developing Countries: 8th International Conference, AFRICOMM 2016, Ouagadougou, Burkina Faso, December 6-7, 2016, Proceedings*. Springer, Október 2017, ISBN 978-3-319-66742-3, google-Books-ID: YjE5DwAAQBAJ.

[12] Bogart, R.; Arenberg, J.: Ray/Triangle Intersection with Barycentric Coordinates. *Ray Tracing News*, ročník 1, č. 11, 1988: s. 17–19.

[13] Chromy, A.; Zalud, L.: Novel 3D Modelling System Capturing Objects with Sub-Millimetre Resolution. *Advances in Electrical and Electronic Engineering*, ročník 12, č. 5, December 2014: s. 476 – 487–487, ISSN 1804-3119, doi:10.15598/aeee.v12i5.1123.
URL `http://advances.utc.sk/index.php/AEEE/article/view/1123`

[14] DSP, H.: User guide-cudafy. net. *Online]. File: CUDAfy User Manual*, ročník 1: str. 22.

[15] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, Boston, Massachusetts: Association for Computing Machinery, Jún 1984, ISBN 978-0-89791-128-3, s. 47–57, doi:10.1145/602259.602266.
URL `https://doi.org/10.1145/602259.602266`

[16] Held, M.; Klosowski, J. T.; Mitchell, J. S.: Evaluation of collision detection methods for virtual reality fly-throughs. In *Canadian Conference on Computational Geometry*, Citeseer, 1995, s. 205–210.

[17] Held, M.; Klosowski, J. T.; Mitchell, J. S. B.: *Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs*. 1995.

[18] Herdman, J. A.; Gaudin, W. P.; McIntosh-Smith, S.; aj.: Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, November 2012, s. 465–471, doi:10.1109/SC.Companion.2012.66.

[19] Hügli, H.; Jost, T.: A match and merge method for 3D modeling from range images. *Signal Processing, 2002 6th International Conference on*, ročník 2, Január 2002: s. 1783–1786 vol.2, doi:10.1109/ICOSP.2002.1180148.

[20] Karimi, K.; Dickson, N. G.; Hamze, F.: A Performance Comparison of CUDA and OpenCL. *arXiv:1005.2581 [physics]*, Máj 2011, arXiv: 1005.2581.
URL http://arxiv.org/abs/1005.2581

[21] Lampáš, M.: Spojování dílčích 3D modelů povrchu do celkového modelu. Jún 2019.
URL https://www.vutbr.cz/studenti/zav-prace/detail/119318?zp_id=119318

[22] Liu, S.; Lin, Y.: *Grey information: theory and practical applications*. Springer Science & Business Media, 2006.

[23] Möller, T.; Trumbore, B.: Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools*, ročník 2, č. 1, Január 1997: s. 21–28, ISSN 1086-7651, doi:10.1080/10867651.1997.10487468, publisher: Taylor & Francis _eprint: https://doi.org/10.1080/10867651.1997.10487468.
URL https://doi.org/10.1080/10867651.1997.10487468

[24] Munshi, A.: The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, August 2009, s. 1–314, doi:10.1109/HOTCHIPS.2009.7478342.

[25] Nayebi, D. F.: *Swift Functional Programming*. Packt Publishing Ltd, Apríl 2017, ISBN 978-1-78728-345-9, google-Books-ID: 70EwDwAAQBAJ.

[26] Pito, R.: Mesh integration based on co-measurements. In *Proceedings of 3rd IEEE International Conference on Image Processing*, ročník 2, September 1996, s. 397–400 vol.2, doi:10.1109/ICIP.1996.560846, iSSN: null.

[27] Puech, C.; Yahia, H.: Quadtrees, octrees, hyperoctrees: a unified analytical approach to tree data structures used in graphics, geometric modeling and image processing. In *Proceedings of the first annual symposium on Computational geometry*, SCG '85, Baltimore, Maryland, USA: Association for Computing Machinery, Jún 1985, ISBN 978-0-89791-163-4, s. 272–280, doi:10.1145/323233.323268.
URL https://doi.org/10.1145/323233.323268

[28] Remondino, F.; El-Hakim, S.: Image-based 3D Modelling: A Review. *The Photogrammetric Record*, ročník 21, č. 115, 2006: s. 269–291, ISSN 1477-9730, doi:10.1111/j.1477-9730.2006.00383.x.
URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1477-9730.2006.00383.x

[29] Rockafellar, R. T.: *Convex Analysis*. Princeton University Press, 1970, ISBN 978-0-691-01586-6, google-Books-ID: GV6YDwAAQBAJ.

[30] Sansoni, G.; Trebeschi, M.; Docchio, F.: State-of-The-Art and Applications of 3D Imaging Sensors in Industry, Cultural Heritage, Medicine, and Criminal Investigation. *Sensors*, ročník 9, č. 1, Január 2009: s. 568–601, doi:10.3390/s90100568.
URL `https://www.mdpi.com/1424-8220/9/1/568`

[31] Schubert, E.; Zimek, A.: ELKI: A large open-source library for data analysis - ELKI Release 0.7.5 "Heidelberg". *arXiv:1902.03616 [cs, stat]*, Február 2019, arXiv: 1902.03616.
URL `http://arxiv.org/abs/1902.03616`

[32] Skiena, S. S.: *The Algorithm Design Manual*. Springer Science & Business Media, Apríl 2009, ISBN 978-1-84800-070-4, google-Books-ID: 7XUSn0IKQEgC.

[33] Zalud, L.; Kalvodova, P.; Burian, F.: RoScan 2.0 - Multispectral Hi-Resolution Scanner. In *Modelling and Simulation for Autonomous Systems*, editácia J. Mazal, Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, ISBN 978-3-030-14984-0, s. 202–214, doi:10.1007/978-3-030-14984-0_16.

[34] Zbakh, M.; Essaaidi, M.; Manneback, P.; aj.: *Cloud Computing and Big Data: Technologies, Applications and Security*. Springer, Júl 2018, ISBN 978-3-319-97719-5, google-Books-ID: BLhmDwAAQBAJ.

[35] Zhao, Y.; Su, X.: Chapter 8 - The Immersed Membrane Method and Fluid-Structure Interaction. In *Computational Fluid-Structure Interaction*, editácia Y. Zhao; X. Su, Academic Press, Január 2019, ISBN 978-0-12-814770-2, s. 109–126, doi:10.1016/B978-0-12-814770-2.00008-8.
URL `http://www.sciencedirect.com/science/article/pii/B9780128147702000088`

Zoznam symbolov, veličín a skratiek

GPGPU	General-purpose computing on graphics processing units
GPU	Graphics processing units
CPU	Central processing units
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
API	Application programming interface
CUs	Compute Units
PEs	Processing elements
SIMD	Single Instruction, Multiple Data
SPMD	Single program, Multiple data
PC	Program Counter
3D	Troj-Dimenzionálna