Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Sodja

Segmentacija prostorskih medicinskih podatkov na GPE

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Matija Marolt

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00033/2013
Datum: 11.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **ANŽE SODJA**

Naslov: **SEGMENTACIJA PROSTORSKIH MEDICINSKIH PODATKOV NA GPE**
SEGMENTATION OF MEDICAL 3D VOLUMES ON A GPU

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V okviru diplomske naloge implementirajte algoritem za segmentacijo prostorskih medicinskih podatkov. Za hitrejše delovanje uporabite ogrodje OpenCL in algoritem implementirajte na grafični procesni enoti. Algoritem integrirajte v orodje za vizualizacijo vratnih žil Neck Veins.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

za

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Anže Sodja, z vpisno številko **63100270**, sem avtor diplomskega dela z naslovom:

Segmentacija prostorskih medicinskih podatkov na GPE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matija Marolta
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 19. september 2013 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Matiju Maroltu za nasvete in pomoč pri izdelavi diplomske naloge.

Prav tako se zahvaljujem as. mag. Cirilu Bohaku za pomoč pri implementaciji in za nasvete pri izdelavi diplomske naloge na splošno.

Posebna zahvala gre sestri Špeli za večkraten pregled dela. In jasno vsej družini za spodbudo in podporo skozi vsa leta študija.

Kazalo

Povzetek

Cilj diplomske naloge je učinkovita implementacija segmentacije medicinskih 3D volumnov, pridobljenih s 3D digitalno subtrakcijsko angiografijo. Za namene segmentacije so implementirani algoritmi za čiščenje podatkov in kreiranje 3D modela. Za hitrejše delovanje so algoritmi implementirani s knjižnico OpenCL za izvajanje na grafično procesni enoti. Segmentacija je vključena v že izdelan program za prikazovanje 3D modelov Neck Veins. V diplomskem delu so najprej opisani vsi implementirani algoritmi, nato sledi opis knjižnice za splošno procesiranje OpenCL in posebnosti implementacije algoritmov na grafično procesni enoti. Na koncu je predstavljena razširitev programa Neck Veins. Za prikaz učinkovitosti implementacije je dodana še primerjava hitrosti izvajanja algoritmov na centralno procesni enoti v programskem jeziku Java in na grafično procesni enoti s knjižnico OpenCL.

Ključne besede:
segmentacija, GPE, algoritmi, OpenCL, 3D volumni, Marching cubes

Abstract

The main goal of the thesis is an efficient implementation of segmentation of medical 3D volumes obtained by means of 3D digital subtraction angiography. For the purposes of segmentation different algorithms are implemented – algorithms for data purification and algorithms for creating 3D models. To achieve better performance algorithms are implemented on a graphic process unit with OpenCL API. Complete segmentation process is integrated into a program designed for visualization of 3D models called Neck Veins. The thesis first describes all implemented algorithms, followed by a description of the library for general processing OpenCL. Finally, it describes an extension of the program Neck Veins. A comparison of the performance of certain algorithms on a central processing unit in the Java programming language, and the implementation on the graphics processing unit on the OpenCL API is made to show the effectiveness of the implementation.

Keywords:
segmentation, GPU, algorithms, OpenCL, 3D volumes, Marching cubes

Poglavje 1

Uvod

Sodobna računalniška tehnologija je prisotna v vse več panogah: v gospodarstvu, zdravstvu, v finančnih institucijah in izobraževanju. Računalniški sistemi shranjujejo ogromno količino podatkov. Te podatke obdelajo, rezultate pa predstavijo uporabnikom. Za lažjo predstavitev podatkov se uporabljajo najrazličnejše vizualizacije - od prikaza povezav med ljudmi s pomočjo grafov, finančne uspešnosti s funkcijskimi grafi do predstavitve 3D volumnov s 3D grafiko.

Odkritje rentgenskih žarkov in rentgenskih slik leta 1895 [1], je v zdravstvu pomenilo velik napredek in pomoč pri odkrivanju obolenj pri človeku. Zdravnikom nudi vpogled v notranjost človeka in jim pomaga pri lažjem diagnosticiranju in zdravljenju bolezni. Najbolj razširjene so 2D rentgenske slike, ki prikažejo prerez določenega volumna. Danes se vedno bolj uveljavljajo 3D rentgentske slike [2]. S tem se odpira možnost 3D prikaza organov, žil, kosti ali drugih delov telesa, s tem pa natančnejše metode zdravljenja.

V diplomski nalogi sem se ukvarjal z obdelavo 3D volumnov žil, pridobljenih s 3D digitalno subtrakcijsko angiografijo (angl. digital subtraction angiography, v nadaljevanju DSA). Glavni cilj naloge je bila razširitev programa Neck Veins z učinkovito implementacijo algoritmov za segmentacijo

3D volumnov, ki bi omogočala skoraj takojšen prikaz zajetih 3D objektov z namenom zmanjšanja časovne premostitve med zajemom podatkov in njihovo predstavitvijo.

Pri tem sem spoznal algoritme, ki se pogosto uporabljajo pri obdelavi podatkov. Algoritme sem implementiral na grafično procesni enoti (v nadaljevanju GPE) z uporabo knjižnice za splošno namensko procesiranje. Spoznavanje algoritmov in učenje novih tehnologij ter možnost implementacije postopka, ki bi v prihodnosti lahko poenostavil delo medicinskega osebja ob zajemu rentgenskih podatkov, je bila glavna motivacija pri izbiri teme za diplomsko delo.

Diplomsko delo je sestavljeno iz treh delov. V prvem delu so opisani algoritmi, ki so bili uporabljeni v implementaciji za segmentacijo zajetih 3D volumnov. V drugem delu je predstavljena knjižnica za splošno namensko procesiranje in posebnosti implementacij algoritmov na GPE. V zadnjem delu je prikazana implementacija segmentacije v programu za prikaz 3D objektov Neck Veins.

Poglavje 2

Algoritmi za segmentacijo

S segmentacijo razdelimo podatke na segmente z enakimi lastnostmi. V primeru segmentacije 3D volumnov žil poskušamo čim bolj natančno poiskati dele volumnov z vrednostmi, ki predstavljajo zapis žil. Glede na te vrednosti nato iz podatkov konstruiramo 3D objekt za prikaz.

3D volumni so pridobljeni s 3D DSA. 3D DSA [3] je medicinski postopek, kjer v določeno mehko tkivo vstavimo barvno kontrastno sredstvo in nato tkivo slikamo iz več zornih kotov z uporabo C-roke. Dobljene slike nato odštejemo od slik istega tkiva iz istih zornih kotov brez kontrastnega sredstva. Te slike nato združimo in dobimo 3D volumne, ki jih uporabimo za segmentacijo.

3D volumen je predstavljen kot zaporedje števil, kjer vrednost določenega števila predstavlja intenzivnost razlike kontrastnega sredstva in ozadja. V postopku segmentacije se glede na vrednosti volumna določi prag, ki predstavlja mejo med vrednostmi žile in ozadja. Ta prag lahko določimo sami, ali ga avtomatsko poiščemo z Otsujevo metodo. Glede na dobljeno vrednost nato z algoritmom Marching cubes konstruiramo približen objekt žil v podatkih. Objekt je zapisan kot množica trikotnikov. To množico trikotnikov na koncu pregledamo in poiščemo vse komponente, ki so sestavljene iz povezanih trikotnikov. Pred samo segmentacijo lahko podatke še počistimo z Gaussovim glajenjem. Implementacija segmentacije je opisana v 4.4.1.

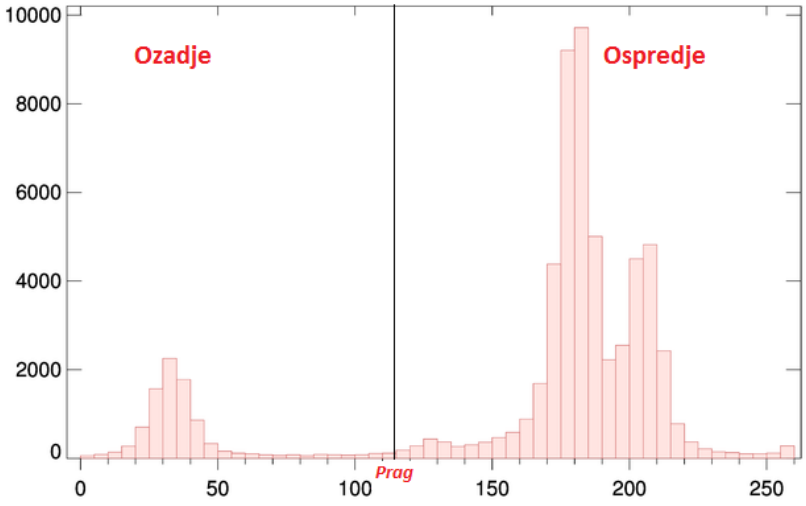
V naslednjem poglavju so opisani uporabljeni algoritmi za namene segmentacije.

2.1 Otsujeva metoda za iskanje praga

Pri izdelavi 3D objekta iz podatkov moramo poiskati in določiti prag, ki loči objekt, ki ga hočemo prikazati, od ostalih podatkov. Ta prag lahko ocenimo sami, dobra vrednost pa je pogosto kar polovica maksimuma vseh vrednosti. Vendar se lahko ocenjevanju izognemo in poiščemo optimalni prag avtomatsko s pomočjo algoritmov. Eden izmed algoritmov za avtomatsko iskanje praga je Otsujeva metoda.

Otsujeva metoda za iskanje praga avtorja Nobuyuki Otsuja je učinkovit postopek ločevanja objektov od ozadja na slikah. Namenjena je transformaciji sivinskih slik v slike s samo dvema vrednostima - črno in belo [4, 5].

V tej implementaciji segmentacije je bila uporabljena za iskanje optimalnega praga med objektom in ozadjem z namenom čim manjšega števila šumnih objektov v končnem modelu.



Slika 2.1: Določen prag na histogramu [6]

2.1.1 Algoritem

Glavna ideja algoritma je razdelitev slike na dva razreda: elemente, ki so v ozadju in elemente, ki so v ospredju. Tako za nek histogram algoritem določi razred za ozadje in razred za ospredje. Razreda sta ločena z izračunanim pragom (glej sliko 2.1) [5].

Algoritem z izčrpnim iskanjem minimizira razpršenost (varianco) *znotraj* razreda, ki ga definiramo kot uteženo vsoto razreda ozadja in ospredja. Torej minimizira kombinaciji razpršenosti obeh razredov. Kot utež vzamemo porazdelitveni funkciji razredov glede na prag. Formula za varianco pri pragu t je tako [4, 5]:

$$\sigma^2(t) = \omega_1(t) * \sigma_1^2(t) + \omega_2(t) * \sigma_2^2(t) \tag{2.1}$$

To se lahko prevede v razpršenost *med* razredoma, ki je definirana kot:

$$\sigma^2(t) = \omega_1(t) * \omega_2(t) * (\mu_1(t) - \mu_2(t))^2 \tag{2.2}$$

$\mu_i(t)$ je povprečna vrednost za oba razreda in se izračuna kot:

$$\mu_1(t) = \frac{\sum_{i=0}^{t} p(i) * i}{\omega_1(t)} \text{ oz. } \mu_2(t) = \frac{\sum_{i=t}^{255} p(i) * i}{\omega_2(t)} \tag{2.3}$$

Porazdelitvena funkcija $\omega_i(t)$ pa:

$$\omega_1(t) = \sum_{i=0}^{t} p(i) \text{ oz. } \omega_2(t) = \sum_{i=t}^{255} p(i) \tag{2.4}$$

p(i) je frekvenca določene vrednosti za histogram.

Algoritem se tako izvede v naslednjih korakih [5]:

1. Poiščemo histogram volumna
2. Za vsako vrednost histograma izračunamo frekvenco
3. Izračunamo porazdelitveni funkciji in povprečni vrednosti obeh razredov za vsako vrednost histograma

4. Poiščemo vrednost histograma, kjer je dosežen maksimum razpršenosti med razredoma ozadja in ospredja - ta vrednost je naš prag

5. Glede na prag se izvede binarizacija - slikovnim točkam (angl. pixel) določimo vrednosti 0 ali 1

V [5] je predlagano, da se poišče še drugi maksimum in se naredi povprečje dveh maksimumov. Prag dveh maksimumov je uporabljen tudi v implementaciji.

Ker se v nadaljevanju segmentacije uporablja algoritem Marching cubes, ki prav tako gleda vrednosti slikovnih točk glede na prag, se lahko binarizacija izvede kar tam. Ker pa s pravimi vrednostmi dobimo lepši 3D model segmentiranega volumna, ta del Otsujeve metode izpustimo.

2.2 Glajenje z Gaussovo funkcijo

Glajenje z Gaussovo funkcijo je zelo razširjen postopek pri obdelavi slik, uporablja pa se tudi za odpravljanja šuma v računalniškem vidu [7]. Uporaba Gaussovega glajenja ni omejena samo na dvodimenzionalne slike, temveč se lahko uporablja za glajenje v 3D prostoru. S tem namenom je Gaussovo glajenje uporabljeno tudi v tem diplomskem delu.

2.2.1 Gaussova funkcija

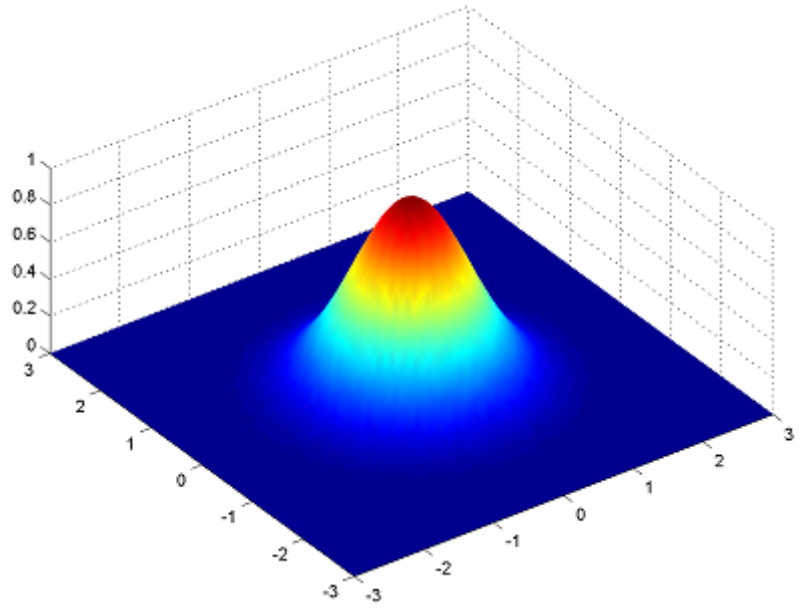
Gaussovo glajenje temelji na uporabi Gaussove funkcije z normalno porazdelitvijo. Uporaba Gaussove funkcije je zelo široka. Uporablja se v statistiki, obdelavi signalov, obdelavi slik ipd.

Gaussovo funkcijo za eno spremenljivko lahko zapišemo kot [8]:

$$G(x) = \frac{1}{\sqrt{2 * \pi * \sigma^2}} * e^{-\frac{(x-\mu)^2}{2*\sigma^2}} \tag{2.5}$$

oz. v več dimenzijah kot:

$$G_n(\vec{x}) = \frac{1}{(\sqrt{2 * \pi} * \sigma)^n} * e^{-\frac{|\vec{x}-\vec{\mu}|^2}{2*\sigma^2}} \tag{2.6}$$



Slika 2.2: Gaussova funkcija dveh spremenljivk [10]

Pri tem je σ standardna deviacija in grafično pomeni sploščenost Gaussove funkcije. μ je aritmetična sredina funkcije. V primeru naše uporabe σ pomeni velikost vpliva sosednjih slikovnih točk [9], μ pa je pozicija slikovne točke na sliki, ki jo povprečimo (v implementaciji to poenostavimo in Gaussove vrednosti izračunamo samo v $\mu = 0$) . Na sliki 2.2 je prikazana Gaussova funkcija dveh spremenljivk.

2.2.2 Implementacija glajenja

Gaussova funkcija se uporablja kot utež za povprečenje določene slikovne točke na podlagi sosednjih. Pred začetkom izberemo vrednost parametra sigma in na podlagi tega izračunamo radij vpliva slikovnih točk oz. okno vpliva [9]. Večji kot je parameter sigma, večji bo vpliv sosednjih slikovnih točk. Za velikost okna se izračunajo vrednosti Gaussove funkcije, ki se normalizirajo, da imajo skupno vsoto 1. Vrednosti se nato zapišejo v okno na ustrezno mesto. V 3D je to okno kocka, katere vrednosti padajo z oddalje-

vanjem od središča.

Uteženo povprečenje vsake slikovne točke se izvede z navideznim premikanjem okna po slikovnih točkah volumna. Za trenutno slikovno točko v središču okna množimo vrednosti slikovnih točk, ki jih okno pokrije z vpadajočimi vrednostmi okna. Zaradi distribucije vrednosti v oknu je vpliv vrednosti volumna na robu okna manjše, izven okna vrednosti nimajo vpliva, vpliv vrednosti blizu centra pa je največji. Vse vrednosti se nato sešteje in nadomesti vrednost trenutne slikovne točke z vsoto [9].

Ena izmed lepih lastnosti Gaussove funkcije je separabilnost po dimenzijah. [8]. Ta lastnost nam omogoča uporabo Gaussovega glajenja za vsako dimenzijo posebej. S tem se znebimo kompleksnosti in pohitrimo izvajanje algoritma.

Tako se celoten algoritem za 3D prostor izvede v naslednjih korakih:

1. Izračunamo okno za eno dimenzijo z vrednostjo σ
2. Izvedemo povprečenje za vse slikovne točke po dimenziji x
3. Izvedemo povprečenje za vse slikovne točke po dimenziji y
4. Izvedemo povprečenje za vse slikovne točke po dimenziji z

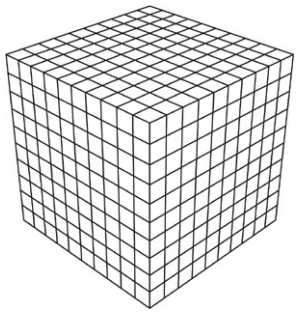
Za še hitrejše glajenje se uporablja hitra Fourierjeva transformacija (angl. Fast Fourier Transform ali FFT). Vendar v diplomskem delu ni bila implementirana.

2.3 Algoritem Marching cubes

Osrednji algoritem diplomske naloge je algoritem Marching cubes. Algoritem na podlagi predoločenih tabel postavitev trikotnikov in vrednosti slikovnih točk 3D volumna konstruira 3D model [11]. Algoritem je mogoče enostavno paralelizirati, kar omogoča izrabo GPE.

2.3.1 Algoritem

Za izvedbo algoritma Marching cubes moramo 3D prostor razdeliti na manjše kocke oz. voksle (glej 2.3). Kocke so sestavljene iz osmih vrednosti, ki predstavljajo oglišča. Algoritem se nato sprehodi skozi vse kocke, za vsako kocko določi postavitev trikotnikov in njihovih normal.



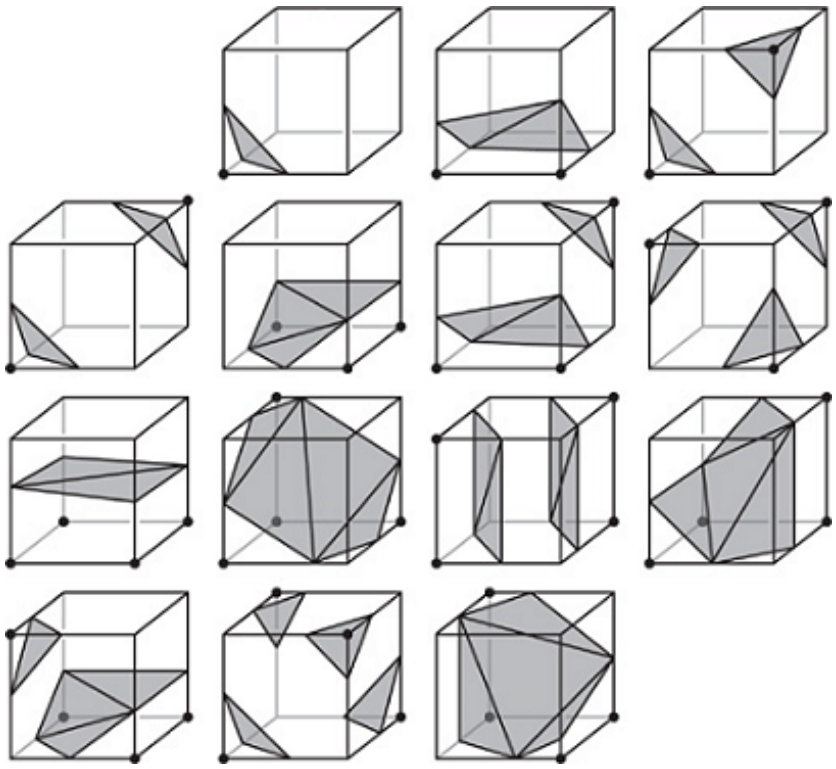
Slika 2.3: 3D prostor razdeljen na kocke oz. voksle [12]

Iskanje trikotnikov

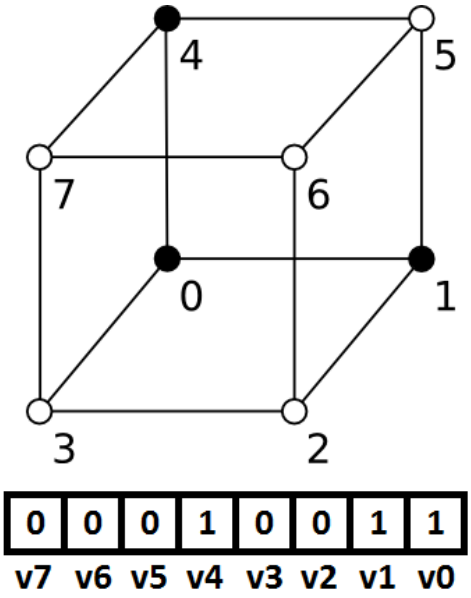
Glede na prag, ki smo ga predhodno določili, je vsako oglišče lahko znotraj ali zunaj objekta. Različnih možnosti je tako 2^8 oz. 256. Teh 256 različnih možnosti se lahko poenostavi na 14 in eno prazno. Možnosti so prikazane na sliki 2.4. Tu upoštevamo, da so možne različne rotacije in simetrije iste predstavitve. Poleg tega se predstavitev ne spremeni v komplementarnih primerih, npr. ko so znotraj objekta oglišča 1, 2, 3, 4, 5 ali če so znotraj oglišča 6, 7, 8 [13].

Za lažjo implementacijo pripravimo tabelo velikosti 256, ki vsebuje vse možne primere. Za vsako kocko pogledamo, koliko vrednosti je znotraj objekta in ustvarimo indeks, 8 bitno število, katere posamezen bit pomeni ali je oglišče znotraj ali zunaj objekta. Preko indeksa poizvedemo po predoločeni tabeli, ki nam pove, katere robove naš objekt seka, glede na vrednosti v ogliščih [13]. Način kreiranja indeksa je prikaz na sliki 2.5.

Ko iz tabele dobimo sekane robove, izračunamo točke na robovih. V



Slika 2.4: 14 možnih kombinacij [14]



Slika 2.5: Kreiranje indeksa

najenostavnejši implementaciji sečne točke postavimo kar na sredino roba. Vendar dobimo boljši model, če postavitev točke linearno interpoliramo glede na vrednosti v obeh ogliščih roba. Dobljene točke na robovih predstavljajo oglišča trikotnikov [13].

Računanje normal

Za pravilno osvetljevanje objekta je dobro, da izračunamo tudi normale. Najpreprosteje je, da uporabimo normale pravokotne na trikotnik. Take normale se izračunajo kot vektorski produkt dveh stranic trikotnika.

Boljše rezultate dobimo, če izračunamo normale kot gradiente v vsaki točki trikotnikov posebej. Gradiente za vsako oglišče kocke lahko aproksimiramo z izračunom razlik vrednosti v sosednjih točkah po isti dimenziji.

Gradient za točko (i, j, k) tako lahko aproksimiramo kot [13]:

$$G_x = \frac{D(i-1,j,k) - D(i+1,j,k)}{\Delta x} \tag{2.7}$$

$$G_y = \frac{D(i,j-1,k) - D(i,j+1,k)}{\Delta y} \tag{2.8}$$

$$G_z = \frac{D(i,j,k-1) - D(i,j,k+1)}{\Delta z} \tag{2.9}$$

Nato gradiente oglišč kocke še interpoliramo enako kot v primeru robov. Tako dobimo normale v točkah trikotnikov.

Celoten postopek algoritma za vsako kocko posebej lahko zapišemo kot:

1. Izdelamo indeks oglišč
2. Iz tabele preberemo presečišča
3. Glede na presečišča izračunamo koordinate oglišč trikotnikov
4. Izračunamo normale za trikotnike

2.4 Iskanje povezanih komponent 3D modela

Pri kreiranju 3D modelov je bilo ugotovljeno, da kreirani modeli vsebujejo veliko ločenih manjših objektov, ki niso del prikazanih žil. Teh motečih objektov se lahko znebimo z uporabo Gaussovega glajenja ali s spreminjanjem praga. Vendar pa uporaba premočnega Gaussovega glajenja ali prevelikega praga vpliva na prikaz modela žil. Zaradi Gaussovega glajenja lahko postane model žil preveč popačen, pri prevelikem pragu pa lahko odrežemo prevelik del ožilja.

Zato bi radi, da se filtrira objekte, ki imajo majhno število povezanih trikotnikov. V ta namen bi lahko preverili povezane komponente pred ali po izvedbi algoritma Marching cubes. Če bi preverjali pred izvedbo bi lahko

uporabili algoritme za označevanje povezanih komponent (angl. Connected-component labeling) [15]. Vendar pa sem se odločil za preverjanje povezanih komponent po izvedbi algoritma Marching cubes.

Z algoritmom Marching cubes dobimo trikotnike, ki predstavljajo naš 3D model. Če hočemo poiskati povezane trikotnike, se moramo sprehoditi po vseh trikotnikih in pregledati sosede, kjer sta soseda dva trikotnika, ki imata vsaj eno skupno oglišče. Ker pa ne vemo, katere točke si trikotniki delijo in je vsak trikotnik zapisan kot svoja trojica oglišč, moramo najprej poiskati skupna oglišča. Algoritem se tako izvede v dveh korakih.

1. Sprehodimo se po trikotnikih in za vsako oglišče dodamo referenco na trikotnik. Tako ustvarimo zbirko unikatnih oglišč, kjer vsako oglišče kaže na listo trikotnikov, ki vsebujejo to oglišče. S tem dobimo vse sosede trikotnikov. Vsak trikotnik pa hrani referenco na unikatno oglišče, ki mu pripada.

2. Za vsako unikatno oglišče hranimo vrednost ali so bili trikotniki, ki vsebujejo to oglišče označeni ali ne oz. ali je bilo oglišče obiskano. Za vsak trikotnik hranimo oznako. Sprehodimo se po trikotnikih. Za vsak trikotnik pogledamo, ali že ima oznako. Če je nima, mu jo dodamo ter pogledamo oglišča. Če katero izmed oglišč še ni označeno, pripadajoče trikotnike dodamo v vrsto. Nato postopek ponovimo za vse trikotnike v vrsti. Ko je vrsta prazna, povečamo oznako in gremo na naslednji neoznačen trikotnik. S tem preiščemo trikotnike v širino (angl. Breadth-first search). (Za drugi korak algoritma glej Algorithm 1).

Alternativno bi sosednost iskali na podlagi robov. Torej bi bila dva trikotnika sosednja, če bi imela skupen rob.

```
Data: oglišča z referencami pripadajočih trikotnikov, vsi trikotniki
Result: označeni trikotniki
inicializiramo vrsto;
oznaka = 1;
for za vse trikotnike do
    if trenutni trikotnik nima oznake then
        označimo trikotnik;
        for globalno neobiskana oglišča trikotnika do
            označimo oglišče kot obiskano;
            dodamo oglišču pripradajoče trikotnike v vrsto;
        end
        while vrsta ni prazna do
            trikotnik = vzamemo prvi trikotnik iz vrste;
            if trikotnik nima oznake then
                označimo trikotnik;
                for globalno neobiskana oglišča trikotnika do
                    označimo oglišče kot obiskano;
                    dodamo oglišču pripradajoče trikotnike v vrsto;
                end
            end
        end
        oznaka = oznaka + 1;
    end
end
```

Algorithm 1: 2. korak iskanja povezanih komponent

Poglavje 3

Implementacija algoritmov na GPE

S pojavom računalniških zaslonov so se okrog leta 1980 začele pojavljati prve grafične kartice. Grafične kartice so dodatni kos strojne opreme, ki pospeši delo z 2D in 3D objekti in tako omogoči boljše rezultate pri prikazu na zaslonu [16].

Grafične kartice morajo obdelati veliko objektov naenkrat. Ker ti objekti niso odvisni eden od drugega, to lahko izvajajo paralelno. Sodobne grafične kartice imajo v ta namen več procesnih enot, ki so v grobem zelo podobne centralnim procesnim enotam, s to razliko, da so prilagojene procesiranju grafičnih objektov [16].

Z napredkom so grafične kartice postale programabilne, kar pomeni, da se lahko na sodobnih grafičnih karticah izvajajo programi, ki niso zapečeni na čip. S tem se lahko manipulira z grafičnimi objekti kar na grafični kartici. Kmalu za tem so se začeli razvijati tudi programski vmesniki (angl. Application programming interface, v nadaljevaju API) za namene obdelovanja podatkov, ki niso vezani na grafiko (angl. general-purpose computing) [16].

3.1 OpenCL

OpenCL je vmesnik za programiranje (angl. Application programming interface, v nadaljevanju API) namenjen za izvajanje paralelnega računanja. Olajša delo z grafično kartico na nizkem nivoju. Programerju tako ni treba skrbeti, kako se bodo ukazi izvajali na ravni strojne opreme.

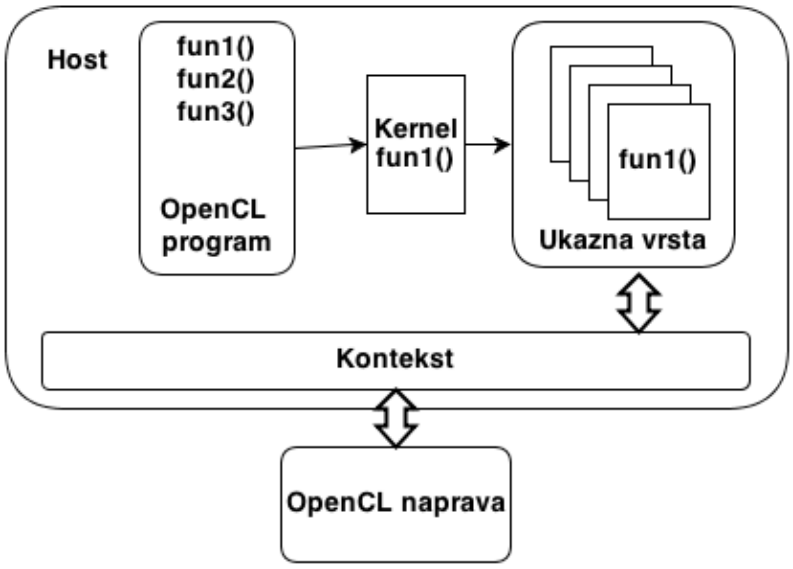
Programe za OpenCL je mogoče poganjati na širokem naboru strojne opreme in ni omejeno samo na grafične kartice [17]. Prenosljivost je glavni razlog, da je za namene pohitritve algoritmov s paralelizacijo, OpenCL API uporabljen v tem diplomskem delu. Druga razširjena alternativa je CUDA podjeta Nvidia, ki pa je omejena na strojno opremo podjetja Nvidia.

3.1.1 Logični model OpenCL

Logični model OpenCL je razdeljen v več enot [18], prikazan je na sliki 3.1. Prva enota se imenuje gostiteljski program (angl. host program). Gostiteljski program se izvaja na centralni procesni enoti. Napisan je lahko v C-ju, C++, Javi ali drugem programskem jeziku. Preko gostiteljskega programa se nadzira samo izvajanje kode OpenCL - prevajanje programov, alociranje pomnilnika, zaporedje izvajanja funkcij ipd.

Druga logična enota so programi OpenCL. Programi OpenCL vsebujejo kodo napisano v programskem jeziku OpenCL C, namenjeno za izvajanje OpenCLa. Programi OpenCL se prevedejo v funkcije OpenCL - kernele. En kernel OpenCL predstavlja določeno akcijo, ki se izvede nad podatki, npr. iskanje maksimuma, izvedba algoritma Marching cubes ipd. Kerneli predstavljajo tretjo logično enoto.

Četrta logična enota je ukazna vrsta (angl. Command-queue). Preko ukazne vrste gostiteljski program pošlje ukaz za nadzor nad izvajanjem OpenCL akcij na strojni opremi, kjer se izvaja koda OpenCL. Povezavo med gostiteljskim programom in OpenCL napravo predstavlja kontekst (angl. Context), ki je definiran znotraj gostiteljskega programa.



Slika 3.1: Shema logičnega modela

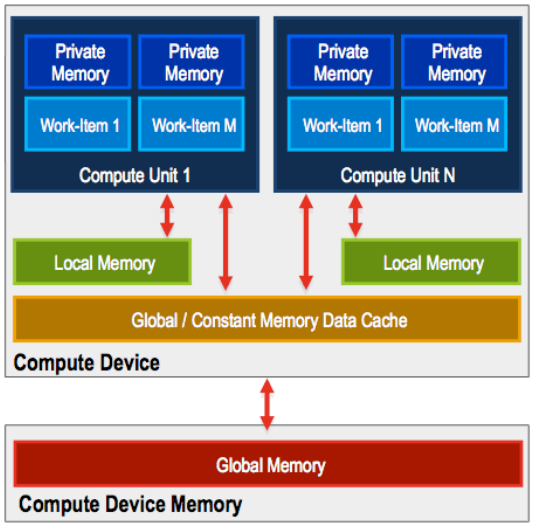
3.1.2 Predstavitev pomnilnika

Za učinkovito implementacijo algoritmov je dobro poznati delo s pomnilnikom in vrste pomnilnikov. OpenCL nudi 5 različnih vrst pomnilnikov [18], predstavljenih s sliko 3.2. Globalni, lokalni, konstantni in privatni pomnilnik so del strojne opreme, na kateri se izvaja kernel OpenCL, dodatno pa je definiran še gostiteljski pomnilnik (angl. host memory), do katerega lahko dostopa gostiteljski program.

Globalni pomnilnik v OpenCLu predstavlja pomnilnik, v katerega lahko pišejo in berejo vse delovne enote, ne glede na postavitev. Ta pomnilnik je tudi počasnejši od ostalih. V OpenCL C ga definiramo kot *global*.

Konstantni pomnilnik je podobno kot globalni dosegljiv vsem delovnim enotam, s to razliko, da je njegova vsebina konstanta in vanj ni mogoče pisati. V OpenCL C ga definiramo kot *constant*.

Lokalni pomnilnik lahko pišejo in berejo le delovne enote določene delovne skupine. Za vsako delovno skupino obstaja ločen pomnilnik. Ta pomnilnik je ponavadi manjši od globalnega in zaradi tega veliko hitrejši, kar lahko



Slika 3.2: Pomnilnik predstavljen v OpenCL APIju [19]

izkoristimo za optimizacijo algoritmov. V OpenCL C ga definiramo kot *local*.

Privatni pomnilnik je namenjen samo določeni delovni enoti in ni dostopen ostalim delovnim enotam. V OpenCL C ga definiramo kot *private*.

Gostiteljski pomnilnik je namenjen uporabi gostiteljskemu programu in ni dostopen preko OpenCL APIja. Preko OpenCLa samo določimo, kako se bo gostiteljski pomnilnik preslikal v pomnilnik na OpenCL napravi.

Delo s pomnilnikom

Za delo s pomnilnikom na strojni opremi na kateri se izvajajo kerneli OpenCL, OpenCL ne nudi možnosti alociranja in sproščanja pomnilnika med izvajanjem. Za alociranje pomnilnika in sproščanje je potrebno poskrbeti v gostiteljskem programu. Za alokacijo pomnilnika ustvarimo objekt Buffer ali bolj napreden objekt Image. Preko objektov tega tipa pišemo podatke na napravo in beremo podatke iz naprave, kjer se izvaja OpenCL [18].

Če je velikost podatkov večja kot globalni pomnilnik naprave, kjer izvajamo kernel, potem se poslužujemo razčlenitve podatkov na manjše dele. Vsak del podatkov pošljemo v kernel, njihove rezultate posebej preberemo

in združimo v gostiteljskem programu.

Pri izvajanju programa lahko zaradi napake v programu dostopamo do naslovov pomnilnika naprave OpenCL, za katere naš program nima pravice [17]. Pri izvajanju na grafični kartici nas naprava o tem ne obvesti. Zato se lahko zgodi, da se ustavi celoten program ali celo operacijski sistem in je potrebno ponovno zagnati računalnik. V najboljšem primeru se ne zgodi nič in se program normalno izvede.

Druga težava pri delu s pomnilnikom nastane, ko alociramo prevelik del globalnega pomnilnika. Na strojni opremi, ki sem jo uporabljal za implementacijo, se je v tem primeru program izvedel in ni vračal napake, kot trdi dokumentacija. Kljub temu, da se je program normalno izvedel, pa podatki niso bili obdelani.

Sinhronizacija

OpenCL ne zagotavlja sinhronizacije podatkov med delovnimi enotami oz. delovnimi skupinami (za delovne enote in skupine glej 3.2). Tako delovne enote, ki vršijo operacije na istem delu pomnilnika, proizvedejo napake. Napaka se lahko izvrši tudi, če se isti del pomnilnika spreminja na različnih mestih kode [17].

Do napake pride že pri preprostih operacijah, kot je na primer povečevanje spremenljivke, ki se izvaja v dveh različnih delovnih enotah. Zato OpenCL omogoča sinhronizacijo znotraj delovnih skupin s pomočjo `barrier()` funkcije [20, 21]. Ko posamezna delovna enota pride do ukaza barrier() se mora ustaviti in počakati dokler vse ostale delovne enote v isti delovni skupini ne pridejo do te funkcije. Težave nastanejo, če pride do razvejitev v programu in delovne enote ne pridejo do iste funkcije. Takrat se lahko izvajanje v določenih delovnih enotah ustavi in program čaka dokler ga ne zatre operacijski sistem.

Vendar pa s funkcijo barrier() ne moremo sinhronizirati delovnih enot globalno. V ta namen lahko uporabimo funkcije `Atomic`. Funkcije Atomic

omogočajo izvedbo atomarnih operacij, torej operacij, ki jih lahko nad enim delom pomnilnika izvaja samo ena delovna enota. Funkcije Atomic lahko vključimo z cl_khr_local_int32_base_atomics : enable. [20]

Glavna pomankljivost atomarnih operacij je, da niso podprte v vseh napravah na katerih lahko uporabljamo OpenCL. Poleg tega so časovno drage, zlasti pri uporabi počasnejšega globalnega pomnilnika. Zaradi tega lahko boljše rezultate dobimo s prilagoditvijo algoritma [21] - tak primer je iskanje maksimuma (glej 3.3.1).

3.1.3 Celoten postopek klica OpenCL programa

Zgoraj so opisane logične enote, ki jih definira OpenCL in delo s pomnilnikom. Poznavanje vseh teh elementov je potrebno za izvedbo klica programa OpenCL. Celoten postopek klica programa OpenCL strnemo v naslednje korake:

1. Inicializacija OpenCL konteksta in ukazne vrste
2. Prevajanje programa OpenCL
3. Kreiranje kernela OpenCL
4. Alokacija pomnilnika in prenos podatkov
5. Nastavitev parametrov kernela in zagon kernela
6. Branje rezultatov
7. Sproščanje spomina
8. Brisanje programa, ukazne vrste in konteksta

Za zagon različnih kernelov ni treba ponovno čez vse korake. Tako poljubnokrat iteriramo med koraki od 3. do 7. Možna je tudi uporaba večjega števila programov, ukaznih vrst in kontekstov.

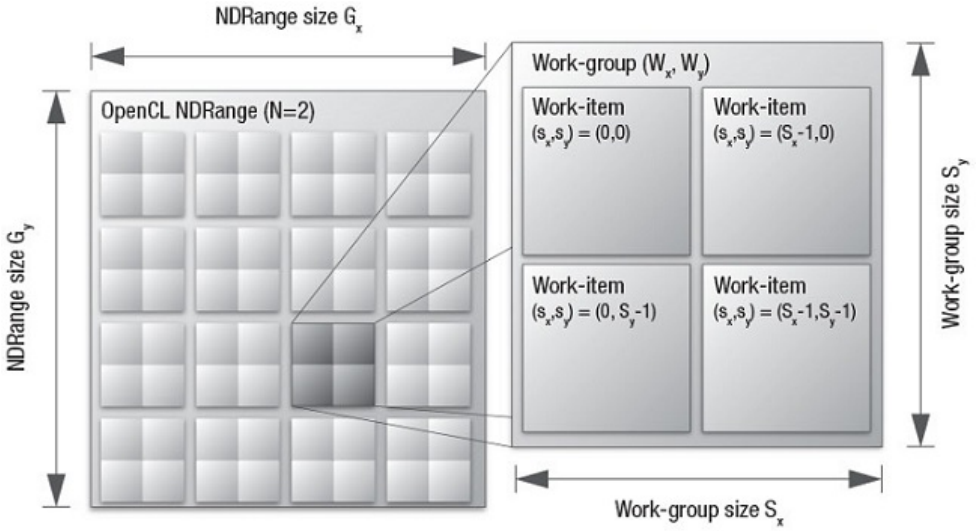
3.2 Deljenje problemov

Za čim boljšo izrabo paralelizacije je potrebno naš problem razbiti na čim bolj neodvisne podprobleme. Tako lahko za algoritem Marching cubes kot podproblem definiramo vsako kocko, saj je iskanje trikotnikov posamezne kocke neodvisno. V primeru binarizacije kot podproblem definiramo vsako slikovno točko, saj vrednosti slikovnih točk med seboj niso odvisne ipd. Pri problemu, ki ga ne moremo razbiti na neodvisne podprobleme, se poslužujemo sinhronizacije, opisane v prejšnem poglavju.

V OpenCL APIju vsak tak podproblem obdeluje ena delovna enota oz. nit (angl. work-item) [17, 18]. Delovne enote so združene v delovne skupine (angl. work-groups). Vsaka delovna skupina se izvede na svojem procesorju in delovne enote ene delovne skupine si delijo procesorske vire. V splošnem se delovne enote izvedejo paralelno. Vse delovne enote skupaj obdelujejo globalen problem. Velikost globalnega problema (in s tem število delovnih enot) moramo določiti sami, medtem ko velikost delovnih skupin lahko prepustimo prevajalniku.

Globalni problem lahko definiramo kot večdimenzionalen problem. Za primer vzemimo sliko velikosti 512x512, ki jo želimo binarizirati (kjer že imamo mejno vrednost). Binarizacija vsake slikovne točke je neodvisen podproblem. Torej imamo 512 * 512 oz. 262144 podproblemov. Iz tega sledi, da je najbolj naravno število vseh delovnih enot 262144 oz. je globalni problem velikosti 262144. Ta globalni problem lahko definiramo kot problem ene dimenzije. Pri tem bomo na podlagi globalne identifikacijske številke delovne enote (angl. global id) dobili vrednosti po eni dimenziji od 0 do 262143. Preko te številke bomo nato lahko izračunali, katero slikovno točko v sliki obdelujemo. V primeru da problem razdelimo v 2 dimenziji velikosti 512, bomo za vsako dimenzijo dobili globalno identifikacijsko številko od 0 do 511. S tem avtomatsko dobimo koordinati slikovne točke. Enako lahko definiramo tudi velikosti delovnih skupin, ki združujejo delovne enote (glej sliko 3.3).

Definiranje dimenzij globalnega problema in velikosti delovnih skupin



Slika 3.3: Razdelitev problema v dveh dimenzijah na delovne enote in delovne skupine [22]

lahko zelo pripomore k hitrosti izvajanja programa. V primeru grafičnih kartic bo v splošnem algoritem hitreje deloval, če naenkrat izkoristimo čim več procesnih enot. Zato lahko različne nastavitve delujejo različno hitro na različni strojni opremi in optimalna hitrost ni zagotovljena. Prav tako je odvisno od strojne opreme, kako velike so lahko delovne skupine za vsako dimenzijo. Velikost globalnega problema pa nima omejitev.

3.3 Prevajanje algoritmov

Ko imamo določeno velikost globalnega problema in kaj vsaka delovna enota obdeluje, lahko prevedemo algoritem za namene paralelizacije. Pri problemih, ki jih lahko enostavno razbijemo na neodvisne podprobleme, je paralelizacija enostavna. Za probleme, kjer je to težje, pa postane pisanje paralelnih algoritmov težje. Paralelizacija se izvede tako, da vsaka delovna enota izvrši akcijo napisano v kernelu OpenCL.

Od uporabljenih algoritmov sem na GPE implementiral algoritem Mar-

ching cubes, Gaussovo glajenje in Otsujevo metodo za iskanje praga. Dodatno sem za potrebe normalizacije vrednosti pri Otsujevi metodi in algoritmu Marching cubes implementiral iskanje maksimuma.

3.3.1 Iskanje maksimuma na GPE

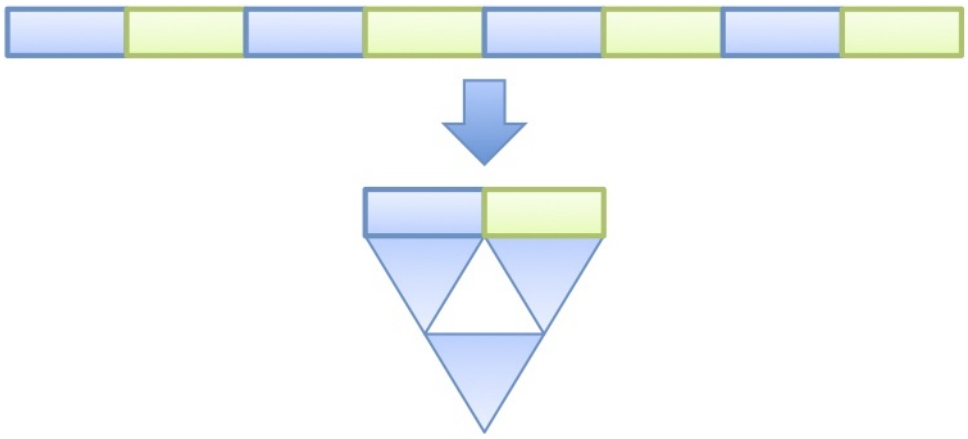
Iskanje maksimuma v vektorju števil, ki ni urejen po velikosti, je zelo preprost postopek. Na začetku nastavimo maksimum na neko majhno vrednost, ki nam zagotavlja, da je to število manjše ali enako maksimumu vektorja. Primeren je kar prvo število vektorja. Sprehodimo se linearno po vektorju in vsako število primerjamo s trenutnim maksimumom. V kolikor je število večje, ga postavimo za nov maksimum in nadaljujemo.

Ne moremo se izogniti zmanjšanju kompleksnosti iskanja maksimuma, pregledati moramo namreč vse elemente, vendar pa lahko postopek pohitrimo s paralelizacijo s pomočjo redukcije. Osnovna ideja redukcije je paralelno iskanje več lokalnih maksimumov. Med lokalnimi maksimumi nato poiščemo še globalni maksimum. Redukcija ni omejena samo na iskanje maksimumov oz. minimumov, lahko jo uporabimo tudi za npr. paralelno seštevanje [23].

Implementiral sem dvostopenjsko redukcijo [23]. Algoritem se izvede v treh stopnjah (glej sliko 3.4). V prvi stopnji se vsa števila razdeli v manjše skupine. Število skupin je enako številu delovnih skupin, recimo `N`. Vsaka delovna skupina je razdeljena na `n` delovnih enot. Vsaka `i`-ta enota nato poišče maksimum, in sicer na vsakem `i + k*n*N` mestu, kjer `k` povečujemo, dokler ne pridemo do konca. Tako dobimo maksimume za vsako delovno enoto.

V drugi stopnji nato poiščemo maksimum znotraj vsake delovne skupine. Tako smo dobili maksimum za vsako delovno skupino. V tretji stopnji samo še preiščemo vektor maksimumov vseh delovnih skupin skupaj. Zaradi redukcije je ta vektor veliko krajši, kot je vseh števil celotnega problema.

Slika 3.4 prikazuje redukcijo za dve delovni skupini.



Slika 3.4: Dvostopenjska redukcija za dve delovni skupini (zelena in modra) [23]

3.3.2 Algoritem Marching cubes in Gaussovo glajenje

Gaussovo glajenje in algoritem Marching cubes lahko enostavno razbijemo na neodvisne podprobleme. Algoritma sta tako enostavna za prenos na GPE.

Ker pri Gaussovem glajenju nove vrednosti slikovnih točk shranjujemo v novo sliko (matriko), se računanje za vsako izmed njih lahko izvede na svoji delovni enoti. Pri implementaciji, kjer povprečimo po vsaki dimenziji posebej, moramo upoštevati, da naenkrat povprečimo slikovne točke le po eni dimenziji.

Pri algoritmu Marching cubes lahko paralelno pregledamo vsako kocko zase, vsaka delovna enota tako pregleda svojo kocko po korakih opisanih v 2.3.

3.3.3 Otsujeva metoda iskanja praga

Pri Otsujevi metodi se paralelno splača implementirati le dele algoritma [24]. Tako sem na GPE implementiral iskanje histograma. Ostali deli algoritma se zaradi majhnega števila operacij in velike odvisnosti podatkov enostavneje hitro implementirajo na centralno procesni enoti (v nadaljevanju CPE).

Učinkovito bi bilo na GPE implementirati tudi binarizacijo [24], ki pa je v našem postopku segmentacije ne potrebujemo.

Histogram na GPE

Najenostavnejša implementacija histograma na GPE bi lahko vsebovala samo dva koraka. Vsako slikovno točko volumna bi določili svoji delovni enoti. Nato bi v vsaki delovni enoti preverili vrednost slikovne točke in povečali histrogram za to vrednost.

Izkaže se, da tak algoritem ne deluje najbolje. Zaradi potrebe po sinhronizaciji in dostopu do istega dela pomnilnika je tak algoritem zelo počasen (glej 3.1.2). Da se izognemo pisanju na isto lokacijo, lahko priredimo algoritem podobno kot iskanje maksimuma. Glavna ideja je razdelitev dela med delovne skupine in iskanje lokalnega histograma za vsako delovno skupino. Dobljene lokalne histograme nato združimo v globalnega. Zaradi hitrejšega lokalnega pomnilnika in manjšega števila dostopov do istih lokacij s tem pohitrimo algoritem.

3.4 Učinkovitost implementacije na GPE

Narejena je bila primerjava med hitrostjo izvajanja implementacije segmentacije na CPE v programskem jeziku Java in implementacije na GPE z OpenCL APIjem. Rezultati so predstavljeni v tabelah 3.1 3.2 3.3 3.4 in prikazujejo čase izvajanja in koeficiente pohitritev. Vsa merjenja so se izvajala na prenosnem računalniku s specifikacijo: Intel Core 2 Duo T6600 2.20GHz, 4GB delovnega pomnilnika, Nvidia GT240M 1GB, operacijski sistem Windows 8.

V čas izvajanja algoritmov na GPE je bil upoštevan tudi čas za samo inicializacijo programa OpenCL in za delo s pomnilnikom (v oklepaju je naveden čas izvajanja algoritmov brez inicializacije in brez dela s pomnilnikom).

Dimenzija volumna	CPE (Java)	GPE (OpenCL API)	Pohitritev
512 x 512 x 390	81,652s	2,997s (1,7s)	27,24x (48,03x)
256 x 256 x 195	5,462s	0,595s (0,219s)	9,18x (24,94x)
128 x 128 x 97	0,673s	0,268s (0,031s)	2,51x (21,71x)

Tabela 3.1: Gaussovo glajenje ($\sigma = 0.5$)

Dimenzija volumna	CPE (Java)	GPE (OpenCL API)	Pohitritev
512 x 512 x 390	0,695s	1,422s (0,359s)	0,49x (1,94x)
256 x 256 x 195	0,111s	0,236s (0,041s)	0,47x (2,71x)
128 x 128 x 97	0,034s	0,127s (0,016s)	0,27x (2,12x)

Tabela 3.2: Otsujeva metoda iskanja praga

Dimenzija volumna	CPE (Java)	GPE (OpenCL API)	Pohitritev
512 x 512 x 390	22,186s	1,969s (0,689s)	11,27x (32,20x)
256 x 256 x 195	2,295s	0,439s (0,156s)	5,23x (14,71x)
128 x 128 x 97	0,454s	0,158s (0,031s)	2,87x (14,64x)

Tabela 3.3: Algoritem Marching cubes (prag je enak polovici maksimuma vrednosti)

Dimenzija volumna	CPE (Java)	GPE (OpenCL API)	Pohitritev
512 x 512 x 390	104,476s	3,561s	29,34x
256 x 256 x 195	7,973s	0,612s	13,02x
128 x 128 x 97	0,907s	0,275s	3,30x

Tabela 3.4: Celotna segmentacija

Analiza rezultatov

Iz tabele 3.4 lahko razberemo, da smo z implementacijo algoritmov na GPE zelo pohitrili celotno segmentacijo. Kljub slabši strojni opremi tako dosežemo zadovoljive čase segmentacije. Vidno je, da se koeficient pohitritve povečuje skupaj z velikostjo volumnov. Podobne rezultate dobimo tudi za posamezno izvajanje Gaussovega glajenja (tabela 3.1) in algoritma Marching cubes (tabela 3.3).

Slaba stran implementacije na GPE je, da zaradi dodatne inicializacije in dela s pomnilnikom, ter prenosom podatkov iz glavnega pomnilnika v pomnilnik GPE izgubimo nekaj časa. To prikazuje tabela 3.2. Opazimo, da se algoritem Otsujevega iskanja praga izvede hitreje na GPE, vendar pa zaradi dodatnega dela celoten klic in izvajanje programa OpenCL deluje počasneje kot v CPE implementaciji v programskem jeziku Java.

Kljub temu je uporaba GPE implementacije v segmentaciji smiselna. V postopku segmentacije namreč inicializiramo program OpenCL enkrat za več algoritmov. Poleg tega je potrebno prenesti podatke na GPE le na začetku segmentacije. Med segmentacijo podatki ostanejo na grafični kartici, s čimer se izognemo nepotrebni izgubi hitrosti.

3.5 Težave pri implementaciji

Prenos algoritmov iz programskega jezika Java v OpenCL C zaradi podobnosti v sintaksi ni bil preveč problematičen. Glavne težave so se pojavile ob slabem razumevanju delovanja OpenCL APIja in s strojno opremo.

3.5.1 Pomanjkanje pomnilnika

Ker OpenCL ne omogoča dinamične alokacije pomnilnika, je potrebno vnaprej predvideti, koliko pomnilnika bo zasedla rešitev. To se je izkazalo za težavno pri implementaciji algoritma Marching cubes. Zaradi paralelne implementacije namreč ne moremo vedeti kam v pomnilnik lahko zapišemo

trikotnike. Najenostavnejša rešitev bi bila, da bi upoštevali, da ima vsaka kocka največje možno število trikotnikov. S tem bi enostavno izračunali mesta za zapis trikotnikov za vsako kocko. Izkaže pa se, da je ta rešitev preveč prostorsko potratna.

Za obdelavo sem dobil testne 3D volumne velikosti 512 x 512 x 390. Če upoštevamo, da je največje možno število trikotnikov na kocko v algoritmu Marching cubes enako 4, potem imamo za vse kocke v najslabšem primeru 512 * 512 * 390 * 4 trikotnikov. Vsak trikotnik ima 3 oglišča, vsako oglišče pa ima 3 vrednosti x, y in z, ki jih zapisujemo v float natančnosti velikosti 4B. To skupaj nanese 512 * 512 * 390 * 4 * 3 * 3 * 4B oz. 14,7GB za zapis. Tu ne upoštevamo, da je za vsak trikotnik potrebno izračunati tudi normale. Današnje grafične kartice dosegajo do 4GB pomnilnika. Grafična kartica, na kateri sem delal pa ima 1GB pomnilnika. Da bi zmanjšali prostorsko kompleksnost, bi lahko združili podvojena oglišča in zapisali trikotnike bolj kompaktno. Vendar pa bi težko zmanjšali problem tako, da bi zasedel manj kot 1GB pomnilnika.

Druga možnost bi bila razdelitev volumna na ustrezno manjše volumne. Vsak tak volumen bi morali posebej obdelati in rezultat združiti v večji pomnilnik v gostiteljskem programu. Slaba lastnost tega postopka je, da zahteva veliko prenosov podatkov med gostiteljskim programom in OpenCL napravo.

Nekatere implementacije algoritma Marching cubes imajo dodan proces, ki pred samo izvedbo algoritma prešteje in vrednoti vsako kocko in izloči vse kocke, ki ne vsebujejo trikotnikov. Ta implementacija je zanesljiva in učinkovita, vendar pa naredi algoritem bolj kompleksen. Veliko enostavnejša, a vseeno učinkovita rešitev je uporaba globalnega števca trikotnikov z uporabo atomarnih funkcij. Izkaže se, da uporaba atomarne funkcije za števec v tem primeru minimalno vpliva na čas izvedbe, implementacija pa je preprosta, zato je uporabljena tudi v implementaciji segmentacije.

Če bi želeli izdelati 3D model z večjim volumnom, npr. dimenzij 1024 x 1024 x 512, to ne bi več zadostovalo, saj že 3D volumen podatkov zasede

več prostora, kot ga omogoča večina grafičnih kartic. Najenostavneje bi bilo združiti možnost razdelitve volumna in implementacije z globalnim števcem.

3.5.2 Razhroščevanje

Veliko težav mi je povzročilo razhroščevanje. Enostavno je bilo odpraviti napake, ki so se zgodile pri prevajanju programa zaradi sintaktičnih napak. Potrebno je bilo preverjati za napako tipa CL_BUILD_PROGRAM_FAILURE in opisno sporočilo napake izpisati.

Težje je bilo odkriti napake, ki so se zgodile med samim izvajanjem programa. Naprava pogosto ne vrača opisnih sporočil o napaki, mnoge napake pa vračajo enake vrednosti. Tako napake za predolgo izvajanje programov (angl. timeout), napake pri inicializaciji in napake s spominom vračajo enako vrednost - CL_INVALID_COMMAND_QUEUE. Dodatna težava ob napakah pri delovanju pa predstavlja sesutje programa ali celotnega operacijskega sistema.

Poglavje 4

Implementacija segmentacije v programu Neck Veins

V naslednjem poglavju bom opisal in prikazal delovanje programa za segmentacijo, s katerim sem razširil program Neck Veins, ki ga je za diplomsko nalogo implementiral Simon Žagar. Opisal bom tudi urejanje programa in prevajanje programa.

4.1 Program Neck Veins

Neck Veins je program, ki omogoča prikaz 3D objektov iz datotek vrste `.obj`. V osnovi je namenjen prikazu 3D žil, vendar se lahko uporablja tudi za druge objekte. Napisan je v Javi in uporablja OpenGL API za komunikacijo z grafično kartico. Za grafični vmesnik je uporabljena knjižnica TWL.

Program Neck Veins omogoča uporabniku pogled na 3D objekte z možnostjo rotacije objekta, premikanja kamere, spreminjanja svetlobe ipd. Z mojo razširitvijo lahko objekte prikažemo direktno iz podatkov pridobljenih s 3D DSA. Prebrane objekte lahko tudi izvozimo v `.obj` datoteko za kasnejšo uporabo.

4.2 Urejanje programske kode

Pred samo implementacijo segmentacije sem najprej uredil programsko kodo programa. Glavni del programa Neck Veins je bil napisan v enem samem razredu. Ta razred je združeval grafični vmesnik, logiko za delo z grafiko, logiko za delo z vhodno-izhodnimi napravami ipd. Implementacija segmentacije v tako obsežen razred bi bila težja, povečala bi se kompleksnost programa, posledično pa bi bila koda težje razumljiva.

Program sem razdelil v tri sklope, ki sem jih dodal že narejenemu sklopu grafičnih modelov. Vse štiri sklope sem ločil v pakete ter jih razdelil v logične razrede. Tako so nastali štirje glavni paketi - paket z razredi grafičnega vmesnika, paket z razredi za delo z grafiko, paket z grafičnimi modeli, ki so predstavljali 3D objekte in paket z orodji.

Samo arhitekturo programa sem poskusil čim bolj približati že znanim arhitekturam grafičnih vmesnikov iz Jave. Zato sem ustvaril logično hierahijo razredov, podobno tisti, ki jo pozna Swing. Tako imamo glavni razred VeinsWindow, ki skrbi za inicializacijo nastavitev, kjer je glavna zanka in ki skrbi za delo z vhodno-izhodnimi napravami. VeinsWindow vsebuje in inicializira tri glavne podrazrede. VeinsRenderer razred skrbi za izrisovanje 3D modela s pomočjo OpenGL APIja. VeinsFrame inicializira in vsebuje vse gumbe, ki so del knjižnice TWL. Razred HUD pa izrisuje grafični vmesnik, ki je izrisan s pomočjo OpenGL APIja.

4.3 Prevajanje uporabniškega vmesnika

Programu sem dodal možnost spreminjanja jezika grafičnega vmesnika. Neck Veins program omogoča preklapljanje med slovenščino in angleščino.

Možnost menjave jezika grafičnega vmesnika je implementirana s pomočjo Javinega razreda ResourceBundle. Ta omogoča enostavno lokalizacijo s pomočjo pomožnih datotek, v katerih je napisan tekst za vsak jezik posebej. Na ta način se loči programska koda in tekst uporabljen v programih. Za prevajanje tako ni potrebno znanje programiranja, poleg tega je dodajanje novih

prevodov enostavno.

Vsaka pomožna datoteka predstavlja določeno lokalizacijo. Vsaka pomožna datoteka ima k imenu datoteke dodano končnico, ki predstavlja lokalizacijo - za slovenščino je to _sl_SI. Preko teh končnic program razpozna, v kateri datoteki se nahaja klicana lokalizacija. V vsaki pomožni datoteki se nahajajo pari oznak in tekstov. S pomočjo oznak pridobimo tekst v želenem jeziku.

4.4 Dodajanje segmentacije v program

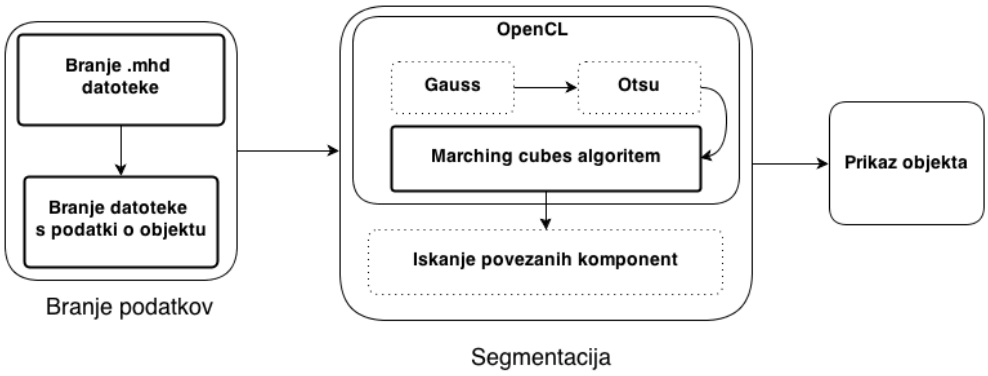
Glavna naloga mojega diplomskega dela je bila razširitev programa Neck Veins z možnostjo branja datotek, v katerih je zapisan 3D objekt in konstruiranje tega objekta s pomočjo algoritmov, ki so opisani v prejšnih poglavjih.

Segmentacija je bila najprej implementirana v jeziku Java, v katerem je napisan tudi program Neck Veins. Zaradi počasnosti je bila prenesena na grafično kartico z uporabo OpenCL APIja. Zaradi lažje implementacije algoritmov in lažjega odpravljanja napak, je bila segmentacija prvotno implementirana kot ločen program, ki je omogočil samo branje datotek ter konstruiranje objektov ter pisanje v .obj datoteko. Naknadno je bila dodana v program Neck Veins, kjer je mogoče klicanje funkcij s pomočjo grafičnega vmesnika.

4.4.1 Postopek obdelave in prikaz

Celoten postopek obdelave je razdeljen na tri glavne dele. To so branje podatkov, segmentacija in prikaz modela. Celoten postopek je prikazan na sliki 4.1.

Program dobi vhodne podatke v obliki dveh datotek. Datoteka s končnico `.mhd`, kjer so zapisani meta podatki o 3D volumnu in datoteka, v katerem je zapisan 3D volumen v obliki zaporedja števil. Najprej preberemo meta podatke: velikost 3D volumna za vsako dimenzijo posebej, orientacijo, tip vrednosti podatkov, v kateri datoteki se volumen nahajajo ipd. Nato se iz datoteke, kjer je zapisan 3D volumen, preberejo vrednosti v pomnilnik.



Slika 4.1: Shema postopka

Ko je branje podatkov končano, se začne postopek segmentacije. Ustvari se OpenCL kontekst in podatki se prenesejo na grafično kartico. Glede na izbrane nastavitve uporabnika se nato izvedejo algoritmi. Najprej se izvede Gaussovo glajenje, da odpravimo šumne podatke. Nato se izvede Otsujeva metoda za iskanje praga. Nazadnje se poiščejo vsi trikotniki z algoritmom Marching cubes.

Tu se zaključi delo na grafični kartici, počistijo se vsi viri na grafični kartici in podatki o trikotnikih se prenesejo v glavni program. V glavnem programu se izvede algoritem za iskanje povezanih objektov. Nefiltrirani objekti se nato pošljejo za prikaz.

Trikotnike se prikaže z že implementiranimi metodami programa Neck Veins. Ker se za prikaz trikotnikov uporablja grafična kartica, to pomeni, da se trikotniki nepotrebno prenašajo iz grafične kartice v glavni program in nato zopet na grafično kartico. Možno bi bilo prikazati trikotnike direktno na grafični kartici takoj po segmentaciji. S tem bi pohitrili celoten postopek obdelave.

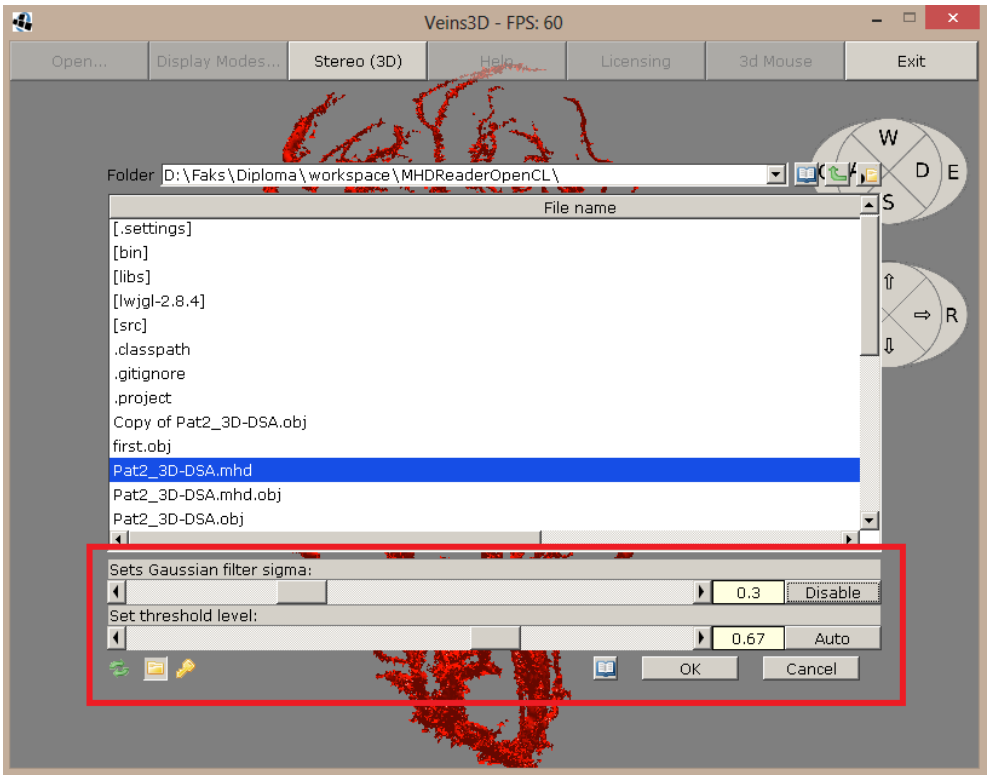
4.4.2 Grafični vmesnik

Grafični vmesnik uporabniku omogoča enostavnejšo uporabo programov in delo z njimi. Za delovanje segmentacije sem v program Neck Veins dodal manjši grafični vmesnik.

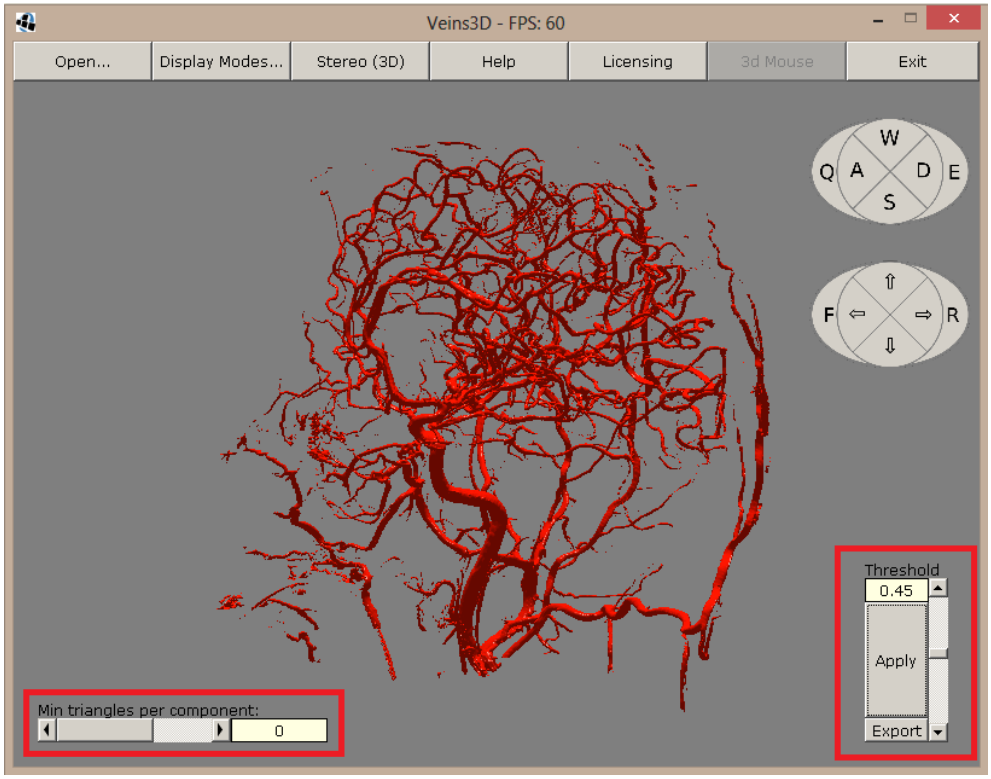
Grafični vmesnik za uporabo segmentacije je bil dodan v oknu za odpiranje datotek in v glavnem oknu programa Neck Veins. Ko uporabnik odpre okno za izbiro vhodne datoteke, pod oknom za pogled na datoteke vidi drsnik za izbiro vrednosti sigma Gaussovega glajenja in možnost izbire stopnje praga (glej sliko 4.2). Poleg drsnika za Gaussovo glajenje se nahaja gumb, s katerim lahko uporabnik izključi ali vključi funkcijo za Gaussovo glajenje. Poleg drsnika za prag je možnost za nastavitev avtomatskega praga z uporabo Otsujeve metode. Izbira vrednosti je mogoča le, če ima trenutno izbrana datoteka končnico `.mhd`.

Ko uporabnik nastavi želene vrednosti in pritisne gumb Odpri (angl. Open), se zažene program za segmentacijo. Rezultat algoritma se prikaže v glavnem oknu. Poleg tega se pojavi drsnik za dinamično izbiro praga in možnost izbire minimalnega števila trikotnikov za filtriranje premajhnih objektov (glej sliko 4.3).

V glavnem oknu lahko spremenimo prag. Ob pritisku gumba Uporabi (angl. Apply) se ponovno zažene segmentacija z novo nastavljenim pragom in v glavnem oknu opazimo isti 3D model z drugim pragom. Ko smo zadovoljni z izbranim pragom, lahko objekt izvozimo v datoteko .obj, ki jo kasneje ponovno uporabimo.



Slika 4.2: Izbira praga in vrednosti sigma v oknu za izbiro vhodne datoteke



Slika 4.3: Izbira praga in minimalnega števila trikotnikov komponent v glavnem oknu

Poglavje 5

Zaključek

V diplomskem delu sem implementiral algoritme za prikaz 3D objektov iz podatkov zajetih s 3D DSA. Algoritme sem implementiral v Javi, vendar sem jih kasneje prenesel na OpenCL API z namenom pohitritve. Program sem implementiral kot razširitev za že obstoječ program prikaza 3D objektov Neck Veins.

Prikaz 3D objektov iz 3D volumnov ima široko možnost uporabe v medicini. S tem olajša delo zdravnikom in izboljša napovedovanje in diagnozo bolezni, kot so rakava obolenja, obolenja žil ipd. Možno bi bilo enak postopek aplicirati v drugih panogah in z drugačnimi načini zajetja podatkov. Morda bi bila možna uporaba za odkrivanje podzemnih jam ali za odkrivanje arheoloških ostankov.

Sam postopek segmentacije bi lahko še izboljšali in optimizirali. Zanimivo bi bilo segmentiranje v realnem času. To bi vsekakor že bilo možno ob manjših vhodnih podatkih in z direktnim prenosom iz OpenCL na OpenGL API. Z boljšo strojno opremo bi to postalo mogoče tudi za večje vhodne podatke.

Postopek segmentacije bi lahko izboljšali tudi s tem, da bi podpiral vhodne podatke, ki presegajo velikost pomnilnika grafičnih kartic ali da bi uporabnik sam segmentiral in ročno odstranil dele 3D objektov, ki jih ne potrebuje za prikaz.

Literatura

[1] (2013, Avgust) Medical radiography - Wikipedia, the free encyclopedia. [Online]. Dostopno na: http://en.wikipedia.org/wiki/Medical_radiography

[2] (2013, September) X-ray computed tomography - Wikipedia, the free encyclopedia. [Online]. Dostopno na: http://en.wikipedia.org/wiki/X-ray_computed_tomography

[3] J. H.-C. Yu. 3D Digital Substraction Angriography. [Online]. Dostopno na: http://www.bme.vn/bmevn/bme/tai-tai-lieu/xu-ly-tin-hieu/51-3d-digital-subtraction-angiography/download/

[4] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, zv. 9, str. 62—-66, 1979.

[5] (2013, Avgust) Otsu's method - wikipedia, the free encyclopedia. [Online]. Dostopno na: http://en.wikipedia.org/wiki/Otsu's_method

[6] (2012, Marec) Separating Bimodal Distributions with Otsu Threshold. [Online]. Dostopno na: http://www.idlcoyote.com/code_tips/otsu_threshold.php

[7] (2013, Avgust) Gaussian blur - wikipedia, the free encyclopedia. [Online]. Dostopno na: http://en.wikipedia.org/wiki/Gaussian_blur

[8] M. K. Chung, "The Gaussian kernel." [Online]. Dostopno na: http://www.stat.wisc.edu/~mchung/teaching/MIA/reading/diffusion.gaussian.kernel.pdf

[9] M. Kristan, "UZ Procesiranje Slik 2," 2013, prosojnice pri predmetu Umetno zaznavanje na FRI.

[10] (2013, September) Gaussian function - Wikipedia, the free encyclopedia. [Online]. Dostopno na: http://en.wikipedia.org/wiki/Gaussian_function

[11] P. Bourke. Polygonising a scalar field (Marching cubes). [Online]. Dostopno na: http://paulbourke.net/geometry/polygonise/

[12] D. Teixeira. GPU Gems 3 - Chapter 22. Baking Normal Maps on the GPU. [Online]. Dostopno na: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch22.html

[13] W. E. Lorensen in H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM Computer Graphics*, zv. 21, št. 4, str. 163–169, 1987.

[14] R. Geiss. GPU Gems 3 - Chapter 1. Generating Complex Procedural Terrains Using the GPU. [Online]. Dostopno na: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html

[15] V. M. A. Oliveira in R. A. Lotufo, "A Study on Connected Components Labeling algorithms using GPUs," v *Graphics, Patterns and Images (SIBGRAPI)*, 2010.

[16] Graphics processing unit. [Online]. Dostopno na: https://en.wikipedia.org/wiki/Graphics_Processing_Unit

[17] J. Thompson in K. Schlachter, "An Introduction to the OpenCL Programming Model," accesible at http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf.

[18] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, in D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley, 2011, Dostopno na: http://it-ebooks.info/book/1602/.

[19] AMD. An Introduction to OpenCL™ . [Online]. Dostopno na: http://www.amd.com/uk/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx

[20] *OpenCL 1.0 Reference Pages*, Khronos group, Dostopno na: http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/.

[21] R. Farber. (2011, January) OpenCL - Part 3: Work-Groups and Synchronization. [Online]. Dostopno na: http://www.codeproject.com/Articles/143395/Part-3-Work-Groups-and-Synchronization

[22] E. Prairie. (2012, Julij) Developing Embedded Hybrid Code Using OpenCL — RTC Magazine. [Online]. Dostopno na: http://www.rtcmagazine.com/articles/view/102657

[23] B. Catanzaro. (2013) OpenCL™ Optimization Case Study: Simple Reductions. [Online]. Dostopno na: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/

[24] B. M. Singh, R. Sharma, A. Mittal, in D. Ghosh, "Parallel Implementation of Otsu's Binarization Approach on GPU," *International Journal of Computer Applications*, zv. 32, št. 2, str. 16–21, October 2011.