SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 923

MODELIRANJE I VIZUALIZACIJA VOLUMETRIJSKIH PODATAKA

Željko Kelava

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 923

MODELIRANJE I VIZUALIZACIJA VOLUMETRIJSKIH PODATAKA

Željko Kelava

Zagreb, srpanj 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 3. ožujka 2025.

DIPLOMSKI ZADATAK br. 923

Pristupnik:	**Željko Kelava (0036529933)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Modeliranje i vizualizacija volumetrijskih podataka**

Opis zadatka:

Proučiti postupke vizualizacije volumetrijskih podataka. Razraditi implementaciju različitih postupaka vizualizacije i usporediti ostvarene performance. Posebice obratiti pažnju na optimizaciju implementiranih algoritama. Razraditi interaktivne tehnike skulpturiranja volumetrijskih podataka u svrhu digitalnog kiparenja. Na nizu primjera ostvariti testiranje dobivenih rezultata. Analizirati i ocijeniti postignute rezultate. Diskutirati upotrebljivost rješenja kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 4. srpnja 2025.

Sadržaj

Uvod

Sa sve većim računalnim resursima i većom dostupnošću metoda za prikupljanje volumetrijskih podataka raste i potreba za njihovom obradom i vizualizacijom. Volumetrijski podaci pojavljuju se u brojnim znanstvenim i inženjerskim područjima kao što su medicina, biologija, geologija, računalne igre i fizikalne simulacije.

Volumetrijski podaci su podaci koji predstavljaju informacije unutar prostora. Možemo ih zamisliti kao trodimenzionalnu rešetku volumnih elemenata. Najmanji element rešetke naziva se voksel (engl. *voxel*). Slično kao što je u slici najmanji element piksel (engl. *pixel*). Vokseli su vrijednosti koji mogu predstavljati neki parametar, primjerice temperaturu, gustoću, protok fluida i slično. Takvi podaci najčešće nastaju simulacijama fizikalnih procesa ili posebnim uređajima u medicini poput magnetske rezonance (*MRI*) ili računalnom tomografijom (*CT*).

Budući da volumetrijski podaci predstavljaju informacije u trodimenzionalnom prostoru postoje različiti načini i pristupi da se prikažu na ekranu.

U ovom radu objasnit će se neki postupci modeliranja i vizualizacije volumetrijskih podataka s naglaskom na njihovu optimizaciju i efikasno izvođenje u stvarnom vremenu. Podaci korišteni u primjerima ovog rada dobiveni su *CT* i *MRI* skeniranjem.

1. Korišteni alati

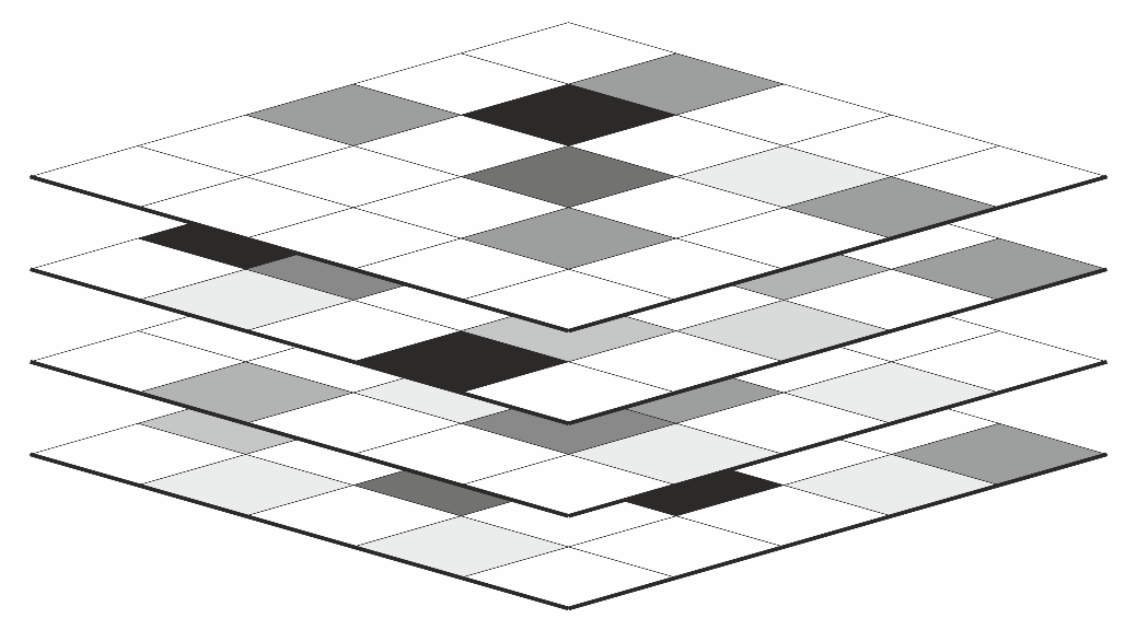
U sklopu ovog rada izrađena je programska aplikacija za demonstraciju metoda modeliranja i vizualizacije volumetrijskih podataka. Program je napisan u programskom jeziku C++. Za iscrtavanje poligona na ekranu korišteno je grafičko sučelje Vulkan. Odabrao sam Vulkan zato što je jedan od najmodernijih grafičkih sučelja, podržava veliki broj procesora i grafičkih kartica te podržava različite operacijske sustave.

Ostale biblioteke koje sam koristio:

- **Vulkan Memory Allocator** – olakšava stvaranje i kontroliranje memorijskih spremnika
- **GLFW** – kontrolira stvaranje prozora te detektiranje korisničkog unosa
- **GLM** – implementira operacije nad matricama i rad s vektorima
- **nifticlib** – čitanje *.nii* datoteka
- **gdcm** – čitanje *.dcm* datoteka
- **imgui** – prikaz korisničkog sučelja

2. Vizualizacija

Vizualizacija volumetrijskih podataka sastoji se od pretvaranja numeričkih informacija u nešto što čovjek može vidjeti. U ovom radu fokus će biti na vizualizaciji podataka dobivenih u medicini, odnosno podataka nastalih korištenjem *CT*-a i *MRI* skeniranja. Podaci nastaju uzimanjem poprečnih presjeka tako da ih možemo zamisliti kao niz 2D fotografija postavljenih jedne na druge. Pikseli predstavljaju točke u prostoru a intenzitet njihove vrijednosti (Slika 2.1).

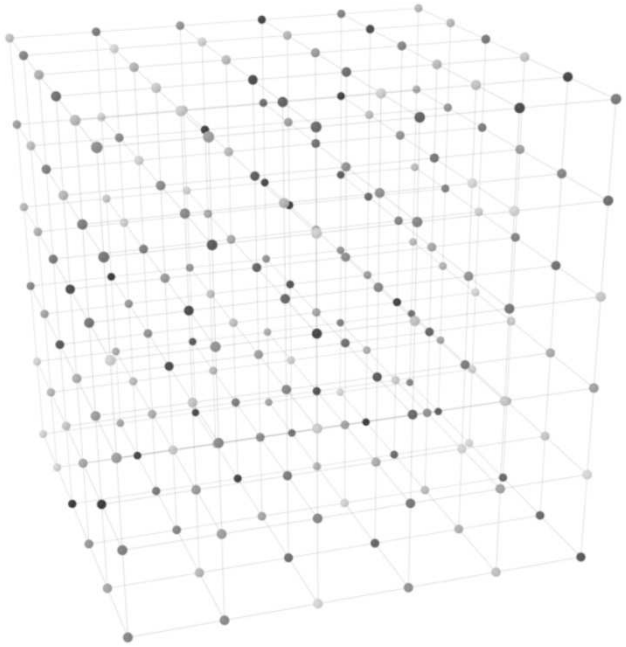


Slika 2.1 Vizualizacija volumetrijskih podataka

Postoji više algoritama koji se mogu koristiti u vizualizaciji, a ovaj rad najviše će pokriti algoritam pokretnih kocaka.

2.1. Algoritam pokretnih kocaka (*Marching Cubes*)

Jedan od algoritama za vizualizaciju volumetrijskih podataka je algoritam pokretnih kocaka. Algoritam kao ulaz prima trodimenzionalno polje, tj. skalarno polje. Skalarno polje može nastati raznim funkcijama koje preslikavaju trodimenzionalni ulaz u jednu vrijednost (skalar). Primjerice udaljenost od ishodišta može biti funkcija koja proizvodi skalarno polje. U našem slučaju vrijednosti skalarnog polja bit će upravo vrijednosti iz podataka dobivenim medicinskim uređajima.

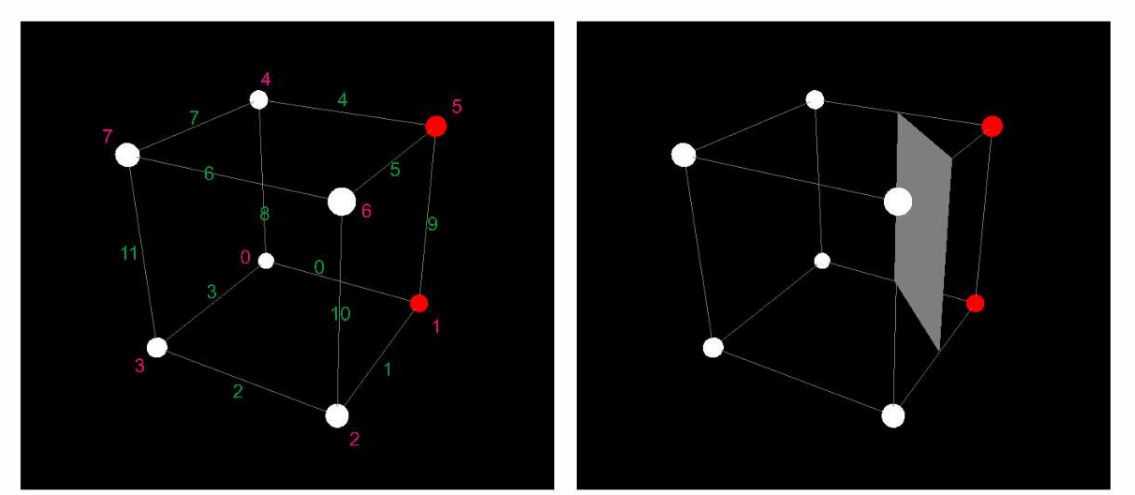


Slika 2.2 Primjer diskretnog skalarnog polja

Rezultat algoritma je poligonalna mreža izgrađena od trokuta, odnosno 3D objekt. Osim skalarnog polja algoritam kao ulaz prima još jedan broj. Taj broj predstavlja granicu (prag) koja određuje koje skalarne vrijednosti pripadaju odnosno ne pripadaju objektu. Objekt koji dobijemo kao rezultat može se shvatiti kao i površina gdje skalarno polje poprima točno neku vrijednost (*isosurface*). Vrijednosti veće od granice su dio objekta, a one manje nisu.

Algoritam se provodi lokalno na svakom 2 x 2 x 2 susjedstvu. Za početak možemo promatrati samo jedno susjedstvo, odnosno kocku. Kocka se sastoji od 8 točaka, a cilj algoritma je konstruirati trokute koji razdvajaju točke u dva skupa. U jednom skupu su točke koje imaju vrijednosti iznad granice, a u drugom one koje su ispod. Budući da postoji 8 točaka, postoji 256 (2^8) različitih kombinacija

Slika 2.3 prikazuje jedan primjer u kojem dvije točke imaju vrijednost veću od granice, a ostale manju. Također prikazuje i način kako bi konstruirali trokute koji odvajaju te dvije točke od ostalih.



Slika 2.3 Oznake vrhova i bridova (lijevo). Poligonalna mreža za vrhove 1 i 5 (desno)

U ovom slučaju to su trokuti koji spajaju središta bridova: [0, 5, 4] te [1, 5, 0]. Postavlja se pitanje kako odrediti koje trokute algoritam treba generirati. Budući da postoji 256 različitih kombinacija u kojima točke mogu biti iznad ili ispod granice, postoji i toliko različitih konfiguracija trokuta. Sve moguće konfiguracije trokuta potrebno je zapisati u *lookup* tablicu prije pokretanja algoritma.

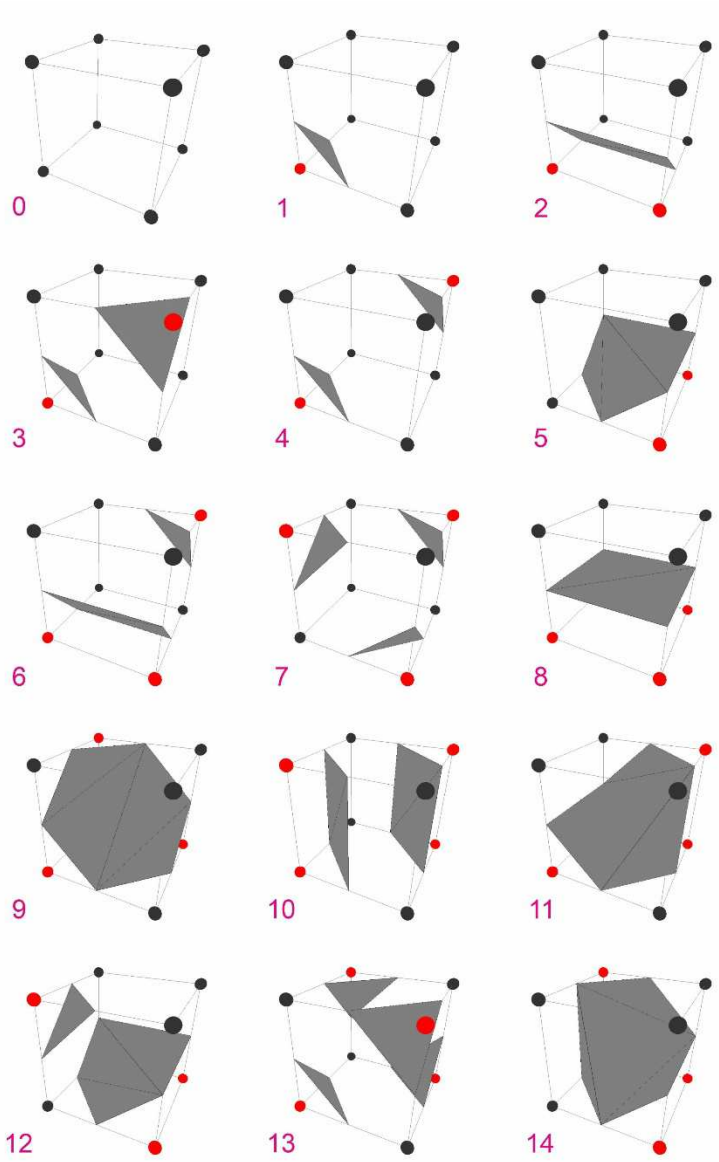
Tablica koja sadrži konfiguracije trokuta sastoji se od 256 redova. Svaki red u tablici sadrži listu bridova. Algoritam uzima prva tri brida iz odgovarajućeg retka i spaja njihova središta. Zatim uzima sljedeća tri brida i opet spaja njihova središta. To ponavlja sve dok ne iskoristi sve bridove iz retka u tablici. Zbog jednostavnosti implementacije svi redovi moraju biti jednake veličine. Ako su iskorišteni svi bridovi, ostatak tog retka ispunjen je s vrijednostima -1. Primjer retka iz prošlog primjera:

```
34:{0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
```

Pitanje je kako odrediti koji red u tablici algoritam treba čitati. To se određuje tako da se indeksima točaka čije su vrijednosti veće od zadane granice pridruže odgovarajuće binarne pozicije koje tretiramo kao binarni broj koji predstavlja indeks retka u tablici. U našem primjeru točke koje su iznad granice imaju indekse 1 i 5. Ako to zapišemo kao pozicije u binarnom broju dobit ćemo broj $00100010_{(2)}$. Taj binarni broj je broj 34 u dekadskom zapisu, odnosno indeks retka u *lookup* tablici.

Premda postoji 256 različitih kombinacija nisu svi slučajevi jedinstveni. Odnosno postoji 14 različitih konfiguracija trokuta, a ostale kombinacije zapravo su neka varijanta rotacije ili simetrije (Slika 2.4). Posebni su slučajevi kada niti jedna točka ne prelazi granicu,

te kada sve točke prelaze granicu. U ta dva slučaja nije potrebno generirati trokute jer ne moramo razdvajati točke. Sve točke se nalaze unutar ili izvan objekta.



Slika 2.4 Svi jedinstveni slučajevi

Generiranjem trokuta završava algoritam za jedno 2 x 2 x 2 susjedstvo. Sada je te korake potrebno ponoviti za sva susjedstva, a nastali trokuti čine poligonalnu mrežu 3D objekta.

Da bismo dobili bolje rezultate, umjesto spajanja središta bridova možemo u obzir uzeti razliku između granice i vrijednosti u točkama. Uzmimo za primjer brid koji s jedne strane ima točku vrijednosti 20, a s druge strane 100, te uzmimo da je granica postavljena na 30. Vrh koji ima vrijednost 100 puno više pripada 3D tijelu. Umjesto da vrh trokuta bude u sredini brida, bolje rezultate ćemo dobiti ako vrh postavimo bliže točki koja ima vrijednost

20. Kako bi točno odredili gdje postaviti vrh trokuta koristit ćemo interpolaciju. Neka t predstavlja relativni položaj vrha trokuta duž brida, a v_1 i v_2 vrijednosti na krajevima brida. Računamo:

$$t = \frac{granica - v_1}{v_2 - v_1}$$

Koordinate vrha trokuta p sada računamo linearnom interpolacijom između dvije točke brida p_1 i p_2:

$$\boldsymbol{p} = (1 - t)\boldsymbol{p}_1 + t\boldsymbol{p}_2$$

Sada više ne koristimo sredinu brida za koordinate vrhova trokuta. Time bolje aproksimiramo stvarnu površinu objekta (Slika 2.5).

Slika 2.5 Vrh trokuta bez interpolacije i s interpolacijom

2.2. Skulpturiranje podataka u algoritmu pokretnih kocaka

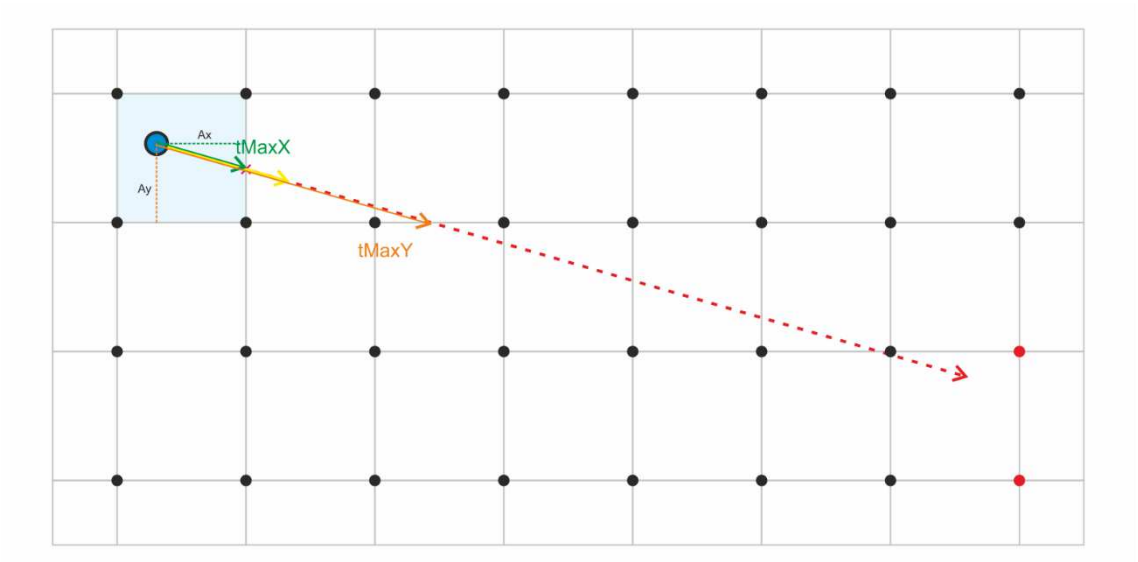
Skulpturiranje volumetrijskih podataka odnosi se na mijenjanje vrijednosti skalara u 3D prostoru kako bi ostvarili doživljaj digitalnog kiparenja. Za razliku od klasičnog modeliranja gdje se manipulira eksplicitno definirana mreža poligona, ovdje se manipuliraju skalarne vrijednosti unutar 3D prostora. Nakon što napravimo promjene u skalarnim vrijednostima potrebno je ponovno pokrenuti algoritam za vizualizaciju kako bi na tom mjestu ponovno izračunali kako geometrija treba izgledati. U sklopu ovog rada napravljen je jednostavan sustav za skulpturiranje volumetrijskih podataka koji koristi položaj i orijentaciju kamere.

Problem kod skulpturiranja skalarnog polja u 3D prostoru je što ne znamo koju vrijednost korisnik želi mijenjati. Nekako moramo odlučiti kako ćemo 2D ulaz korisnika pretvoriti u 3D koordinate koje želimo mijenjati. Pristup koji sam ja odabrao uključuje pucanje zrake iz kamere u smjeru gledanja.

Kada korisnik usmjeri kameru prema dijelu 3D objekta kojeg želi skulpturirati može povećati ili smanjiti vrijednost u skalarnom polju. Prvi korak je odrediti koju vrijednost mijenjati. Najbolji pristup bio bi ispucati zraku iz kamere i detektirati gdje sječe objekt. Zatim uzimamo skalarnu vrijednost koja je najbliža sjecištu zrake i trokuta i tu vrijednost mijenjamo. Problem kod tog pristupa je što svaki put moramo provjeriti da li zraka siječe trokut u 3D prostoru. To samo po sebi nije veliki problem, no stvari se zakompliciraju kada imamo više milijuna trokuta u sceni. Svaki put kada želimo promijeniti vrijednost u skalarnom polju moramo napraviti provjeru presjeka zrake sa svim trokutima u sceni. To je računski skupo i usporava interakciju.

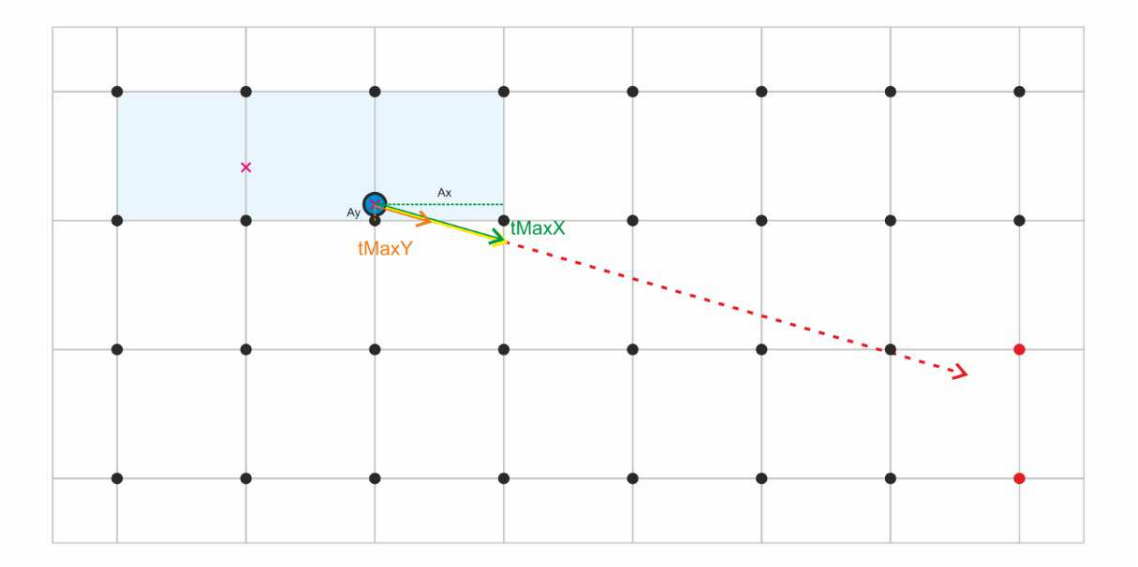
Ovaj pristup moguće je značajno ubrzati koristeći bolju organizaciju podataka, odnosno trokuta. Umjesto da testiramo presjek zrake sa svim trokutima, koristi se hijerarhijska struktura podataka koja grupira trokute i omogućuje brzi prekid provjere velikih dijelova scene. Tipični primjeri su BVH (*Bounding Volume Hierarchy*) ili oktalno stablo. Trokuti su raspoređeni u čvorove stabla i kompleksnost provjere prelazi iz linearne $O(n)$ u logaritamsku $O(log(n))$ složenost. Budući da je implementacija ovakvog algoritma složena, odlučio sam koristiti jednostavniji pristup.

Drugi pristup također koristi kameru i pucanje zrake u smjeru gledanja. Umjesto da detektiramo presjek zrake i trokuta, kretat ćemo se u smjeru zrake sve dok ne dođemo do voksela koji je od interesa. Kretanje je lakše objasniti u 2D prostoru, ali isto vrijedi i u tri dimenzije (Slika 2.6). Algoritam se zove DDA (*Digital Differential Analyzer*).



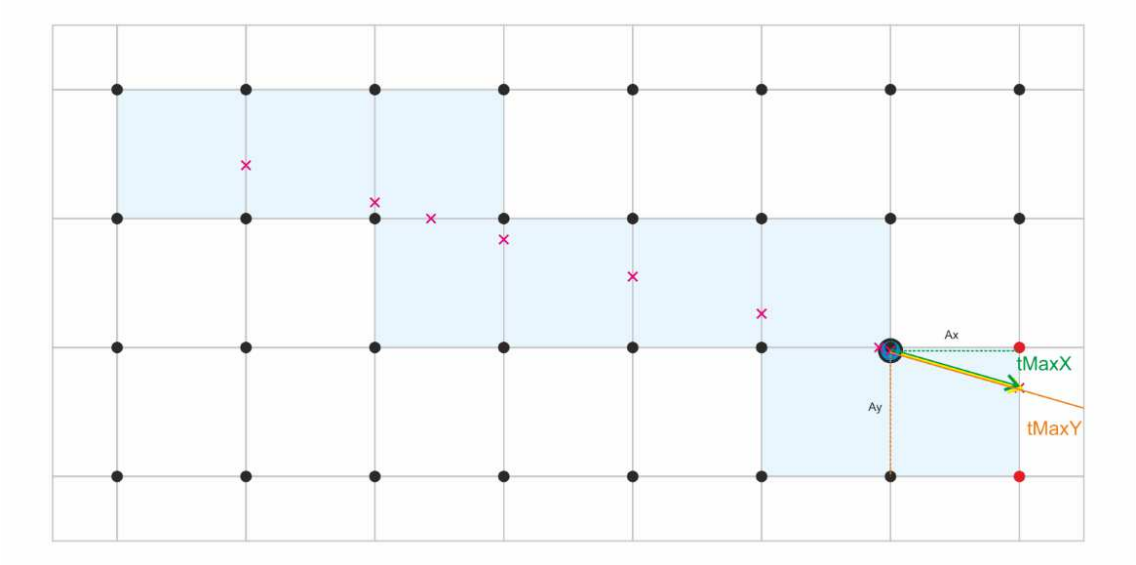
Slika 2.6 Oznake i vizualizacija DDA algoritma

Početak zrake (kamera) nalazi se u nekom pikselu (vokselu) i zraka ima svoju orijentaciju. Želimo se kretati iz kamere u smjeru zrake i obilježiti sve piksele kroz koje prolazimo. U prvom koraku algoritma pozicija kamere nije nužno poravnata s koordinatnim sustavom. Uvodimo oznake A_x i A_y koje predstavljaju duljine do prve promjene piksela (voksela). Budući da znamo nagib (orijentaciju) zrake možemo projicirat duljine A_x i A_y na orijentaciju gledanja kamere i dobiti udaljenosti $tMaxX$ i $tMaxY$. Smjer kretanja određujemo tako da uzmemo broj koji je manji. U prvom koraku to je $tMaxX$ i prelazimo jedan piksel u x smjeru. Zatim ponovno računamo duljine A_x i A_y, te ih projiciramo na vektor zrake. Opet uzimamo manju duljinu i krećemo se u tom smjeru. Nakon par koraka $tMaxY$ će postati manja udaljenost pa ćemo prijeći u novi piksel u y smjeru (Slika 2.7).



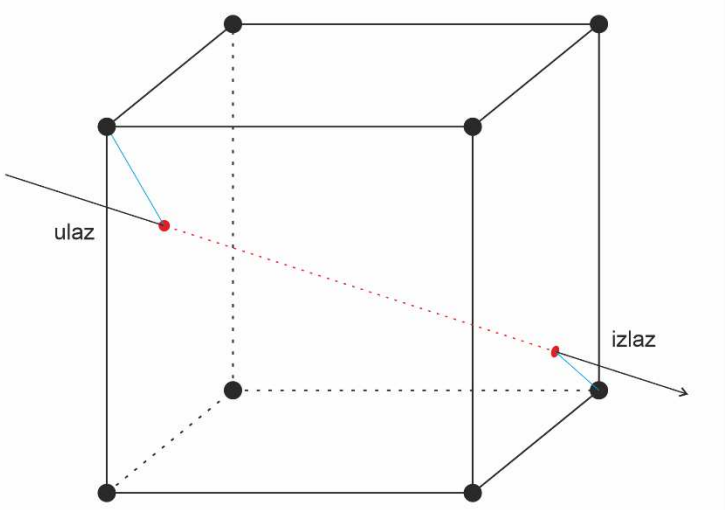
Slika 2.7 Jedan trenutak u izvođenju DDA algoritma

Ovaj postupak ponavljamo sve dok ne dođemo do piksela (voksela) koji u svojim vrhovima sadrži barem jednu vrijednosti koja je veća od granice. Nakon što nađemo takav piksel, računamo još jedan korak algoritma kako bi mogli dobiti koordinate izlaza zrake iz piksela (Slika 2.8). Ulazne i izlazne koordinate zrake zadnjeg piksela potrebne su da bi preciznije mogli odrediti koju skalarnu vrijednost korisnik želi mijenjati. Algoritam se također zaustavlja ako prođemo neku zadanu udaljenost, a nikad ne pogodimo piksel s vrijednostima većima od granice. Svi koraci analogno vrijede i u 3D prostoru osim što tražimo najmanju vrijednost od 3 udaljenosti $tMaxX$, $tMaxY$, $tMaxZ$.



Slika 2.8 Kraj DDA algoritma

Ovim algoritmom dobili smo voksel u 3D prostoru kojeg želimo skulpturirati. Osim voksela dobili smo i koordinate ulaska i izlaska zrake iz voksela. Ovo će nam pomoći odlučiti koji od 8 vrhova tog voksela želimo mijenjati. Računat ćemo udaljenost svih točaka do mjesta ulaska i izlaska u voksel. U obzir ćemo uzeti 2 točke koje su bile najbliže (Slika 2.9). Te točke mogu, ali ne moraju biti iste.



Slika 2.9 Prikaz zrake koja siječe voksel

Još preostaje odlučiti kojoj točki ćemo mijenjati vrijednost. Da bi dobili prirodniji osjećaj kod skulpturiranja odluku ćemo rastaviti na slučajeve.

U slučaju da povećavamo vrijednost u obzir prvo uzimamo točku bližu izlazu zrake. Ako ta točka već ne zadovoljava granicu onda ćemo povećati njenu vrijednost. Inače, ako je

ta točka već zadovoljila granicu onda povećavamo vrijednost u točki koja je bliža ulazu zrake.

Drugi slučaj je da smanjujemo vrijednost u točki. Onda u obzir prvo uzimamo točku koja je bliža ulazu zrake. Ako je vrijednost u toj točki iznad granice, smanjujemo njenu vrijednost. Inače, ako je vrijednost točke već ispod granice, smanjujemo vrijednost u točki bližoj izlazu zrake.

Nedostatak ovog pristupa je što uvijek moramo graditi nad postojećom geometrijom. Zraka uvijek mora pogoditi voksel koji već sadrži skalarne vrijednosti iznad granice. Ne možemo započeti graditi objekt iz nule.

Nakon mijenjanja skalarnih vrijednosti potrebno je ponovno provesti algoritam za generiranje geometrije jer ovim postupkom nismo eksplicitno mijenjali geometriju, već ulazne parametre koji generiraju trokute.

3. Implementacija

Programska aplikacija izrađena je u programskom jeziku C++. Program kao ulaz prima datoteku formata *.nii* ili mapu koja sadrži niz *.dcm* datoteka. Zatim na temelju volumetrijskih podataka računa poligone i prikazuje ih na ekranu.

3.1. Inicijalizacija Vulkan sučelja.

Prvi korak u Vulkan programu je inicijalizacija samog grafičkog sučelja. Vulkan je *low-level* sučelje, pa zahtjeva znatno više kôda nego OpenGL. Potrebno je s interneta preuzeti Vulkan SDK i uključiti ga u program. To je set alata koji pomaže pri izradi aplikacije. Sadrži potrebna zaglavlja, validacijske slojeve i alate za provjeru grešaka. Također sadrži programe koji služe za kompajliranje sjenčara (engl. *shader*). Prikaz prozora na ekranu ostvaren je GLFW bibliotekom. Ta biblioteka komunicira s operacijskim sustavom, a Vulkan je zadužen za crtanje u tom prozoru.

Nakon same inicijalizacije potrebno je kreirati Vulkan instancu. Ona predstavlja vezu između aplikacije i pokretačkog programa (engl. *driver*). U instanci navodimo koje ekstenzije i validacijske slojeve želimo koristiti. Primjerice ekstenzija za crtanje u GLFW prozoru i slojevi koji provjeravaju greške pri pokretanju programa. Poveznica između prozora i Vulkana naziva se *surface*. GLFW napravi objekt *surface*, a Vulkan ga koristi za crtanje.

Dalje slijedi odabir fizičkog i logičkog uređaja. To znači da je prvo potrebno dohvatiti popis svih grafičkih kartica u sustavu, provjeriti da li podržavaju ekstenzije koje smo prije odlučili koristit i odabrati jednu karticu. Zatim stvaramo logički uređaj koji nam omogućuje izradu redova zadataka (engl. *queue*). Možemo imati više redova zadataka s različitim ulogama i različitim prioritetima. Primjerice *graphics queue* (za iscrtavanje) i *present queue* (za prikaz sadržaja u prozoru).

Prikazivanje na ekranu sastoji se od prikazivanja niza 2D slika. Izmjenom slika upravlja mehanizam koji se zove *swapchain*. *Swapchain* se obično sastoji od skupa slika koje se izmjenjuju, najčešće dvije ili tri. Dok se jedna slika prikazuje korisniku, druga se istovremeno popunjava podacima s grafičke kartice. Na taj način postižemo neprekidno osvježavanje prozora.

Da bi definirali na koji način popunjavamo slike iz *swapchaina* potrebno je napraviti *render pass*. *Render pass* određuje što se događa s prikazom na početku i kraju crtanja te koje dodatke (engl. *attachments*) koristimo, primjerice za boju ili dubinu. Točan način i redoslijed koji opisuje svaki korak procesa iscrtavanja zove se *pipeline*. U *pipelineu* detaljno određujemo koji sjenčari se koriste, kakva je topologija podataka (trokuti, linije ...), kako se provodi rasterizacija, na koji način se testira dubina itd. Unaprijed ga je potrebno definirati i nije ga moguće mijenjati tijekom rada programa, ali ako imamo više *pipelineova* onda možemo odabrati koji ćemo kada koristiti.

Preostaje još izraditi spremnik u koji spremamo naredbe koje se šalju grafičkoj kartici. Spremnik se zove *command buffer* i u njega šaljemo naredbe za početak *render passa*, povezivanje *pipelinea*, određivanje izvora podataka, naredbe za crtanje itd. Da bi sve radilo pouzdano koriste se ograde i semafori za sinkronizaciju procesora i grafičke kartice.

Početni kôd koji služi kao kostur za izradu ovih objekata i njihovu sinkronizaciju preuzet je s web stranice [3].

3.2. Algoritam pokretnih kocaka

Program se sastoji od dvije glavne klase. Klasa `VolumeRenderer` odgovorna je za inicijalizaciju Vulkan sučelja, iscrtavanje slika na ekran, dodjeljivanje i računanje memorije i slično. Sadrži poziciju i orijentaciju kamere te funkcije za iscrtavanje korisničkog sučelja.

Druga klasa `MarchingCubesModel` odgovorna je za sve podatke koji imaju veze s algoritmom pokretnih kocaka. Pohranjuje volumetrijske podatke, parametre kao što su veličina i skaliranje u 3D prostoru te granicu pripadnosti. Osim toga sadrži i pokazivače na memoriju u kojoj su zapisani svi podaci koji su potrebni da bi pokrenuli algoritam pokretnih kocaka na grafičkoj kartici. Također sadrži još jednu važnu strukturu podataka - `Chunk`.

Zbog brzine i efikasnosti algoritam pokretnih kocaka pokrenut ćemo na grafičkoj kartici. Budući da je za svaki voksel moguće generirati trokute bez ovisnosti o susjednim vokselima algoritam je moguće vrlo efikasno paralelizirati. Kako bi se algoritam mogao pokrenuti na kartici potrebno je kopirati volumetrijske podatke iz sistemske memorije u memoriju grafičke kartice. Osim samih podataka također kopiramo *lookup* tablicu te neke parametre algoritma. Zadatak grafičke kartice je obraditi podatke, generirati trokute i zapisati ih u memoriju.

3.2.1. Dodjeljivanje i rezerviranje memorije

Da bi podatke kopirali na grafičku karticu potrebno je prvo dodijeliti (engl. *allocate*) memoriju. To znači zatražiti i rezervirati memoriju od operacijskog sustava. Biblioteka Vulkan Memory Allocator olakšava taj proces. Potrebno je napraviti objekte `VkBuffer` i `VmaAllocation`, popuniti par struktura podataka koje definiraju na koji način će se koristiti memorija i pozvati konstruktor `vmaCreateBuffer()`. Primjer jednostavnog kôda koji rezervira memoriju:

```
VkBuffer buffer;
VmaAllocation allocation;
VkDeviceSize bufferSize = totalSize;

VkBufferCreateInfo bufferInfo = {};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = bufferSize;
bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_SRC_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT;
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

VmaAllocationCreateInfo allocInfo = {};
allocInfo.usage = VMA_MEMORY_USAGE_CPU_TO_GPU;
allocInfo.flags = VMA_ALLOCATION_CREATE_MAPPED_BIT;

VmaAllocationInfo allocationInfo;
vmaCreateBuffer(vmaAllocator, &bufferInfo, &allocInfo,
&buffer, &allocation, &allocationInfo);
```

Iz objekta `VmaAllocationInfo` možemo dohvatiti pokazivač na početak rezervirane memorije i kopirati ili čitati podatke. Ovaj proces ponavljamo svaki put kada želimo dodijeliti memoriju programu.

3.2.2. Izrada sjenčara (*shader*)

Program koji se pokreće na grafičkoj kartici zove se sjenčar (engl. *shader*). Sjenčari se izvode nad podacima koji putuju kroz *pipeline*. Postoje različite vrste sjenčara. Primjerice *vertex shader* obrađuje vrhove (točke) 3D modela. Preslikava ih iz 3D prostora u 2D sliku. *Fragment shader* izračunava boju svakog piksela uzimajući u obzir teksture, svjetlo i

materijale. Sjenčar koji se koristi za paralelne izračune na grafičkoj kartici, koji nisu nužno povezani s grafikom, zove se *compute shader*. U ovoj implementaciji korišten je *compute shader* kako bi izračunali vrhove generiranih trokuta i zapisali ih u memoriju.

Kako bi *compute shader* mogao čitati memoriju koju smo rezervirali moramo napraviti još nekoliko koraka.

3.2.2.1 Objekt tipa *Descriptor Set Layout*

Kako bi sjenčar znao koje resurse očekuje potrebno je prvo napraviti objekt tipa `VkDescriptorSetLayout`. U tom objektu zapisano je kakav tip memorije sjenčar očekuje na pojedinim pozicijama (engl. *binding*). Primjerice ako sjenčar treba očekivati 5 različitih pokazivača na memoriju kôd bi izgledao ovako:

```
VkDescriptorSetLayout marchingCubesDescriptorSetLayout;
std::array<VkDescriptorSetLayoutBinding, 5> layoutBindings{};

layoutBindings[0].binding = 0;
layoutBindings[0].descriptorCount = 1;
layoutBindings[0].descriptorType =
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
layoutBindings[0].pImmutableSamplers = nullptr;
layoutBindings[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
...

if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr,
&marchingCubesDescriptorSetLayout) != VK_SUCCESS) {
    throw std::runtime_error("failed to create compute
descriptor set layout!");
}
```

Iz isječka kôda sjenčar očekuje 1 pokazivač na memoriju tipa `STORAGE_BUFFER` na poziciji 0. Slično popunjavamo i ostatak, ali mijenjamo pozicije na kojima će se nalaziti. Ovdje još nismo odredili koji su to točno pokazivači, već samo kojeg su tipa i gdje će se nalaziti.

3.2.2.2 Objekt tipa *Descriptor Pool*

Da bismo mogli odrediti tip memorije koju koristimo u sjenčaru potrebno je napraviti skup u kojem moramo nabrojati sve vrste memorije i koliko kojih ima. Objekt koji sadrži te podatke zove se `VkDescriptorPool`. Primjer kôda za prijašnji primjer:

```
std::array<VkDescriptorPoolSize, 1> poolSizes{};
poolSizes[0].type = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
poolSizes[0].descriptorCount = 3;

poolSizes[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
poolSizes[1].descriptorCount = 2;
...

if (vkCreateDescriptorPool(device, &poolInfo, nullptr,
&marchingCubesDescriptorPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create compute
descriptor pool!");
}
```

Ovdje smo rekli da ćemo koristiti tri pokazivača na memoriju tipa `STORAGE_BUFFER` i dva pokazivača tipa `UNIFORM_BUFFER`. Razlika između ova dva tipa memorije je što u prvi možemo čitati i pisati iz sjenčara, a iz drugog samo čitati. Drugi tip obično ima limit na veličinu (npr. 16-64 KB) pa u njega stavljamo količinski manje podatke kao neke konstante ili parametre koje smo zapisali iz procesora.

3.2.2.3 Objekt tipa *Descriptor Set*

Vrijeme je da izradimo instancu objekta `VkDescriptorSet` u kojeg ćemo zapisati prave pokazivače na memoriju. Za svaki pokazivač potrebno je definirati njegovu adresu, pomak, na koje odredište treba ići i njegov tip. Primjer za jedan pokazivač:

```
VkDescriptorBufferInfo volumeBufferInfo{};
volumeBufferInfo.buffer = marchingCubesModel.volumeBuffer;
volumeBufferInfo.offset = 0;
volumeBufferInfo.range = VK_WHOLE_SIZE;

std::array<VkWriteDescriptorSet, 5> descriptorWrites{};
...
descriptorWrites[1].sType =
VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = marchingCubesDescriptorSet;
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].descriptorType =
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
descriptorWrites[1].pBufferInfo = &volumeBufferInfo;
```

```
...
vkUpdateDescriptorSets(device, 5, descriptorWrites.data(), 0,
nullptr);
```

Pokazivač spremljen u `marchingCubesModel.volumeBuffer` kopirat će se na odredište 1 u sjenčaru i bit će tipa `STORAGE_BUFFER`. Isto ponavljamo za ostale pokazivače.

3.2.2.4 Objekt tipa *Pipeline*

Ostaje još napraviti `VkPipeline` u kojem definiramo koji sjenčar pokrećemo. Potrebno je zadati putanju do datoteke koja sadrži kompajliran kôd sjenčara. Osim toga potrebno je popuniti neke strukture koje pokazuju na objekt `VkDescriptorSet` i slično.

```
auto computeShaderCode = readFile("marchingCubes.spv");
VkShaderModule computeShaderModule =
createShaderModule(computeShaderCode);

VkPipelineShaderStageCreateInfo computeShaderStageInfo{};
computeShaderStageInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
computeShaderStageInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
computeShaderStageInfo.module = computeShaderModule;
computeShaderStageInfo.pName = "main";

VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
...

vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr,
&marchingCubesPipelineLayout);

VkComputePipelineCreateInfo pipelineInfo{};
...

vkCreateComputePipelines(device, VK_NULL_HANDLE, 1,
&pipelineInfo, nullptr, &marchingCubesPipeline);
vkDestroyShaderModule(device, computeShaderModule, nullptr);
```

Sada možemo pozvati funkcije `vkCmdBindPipeline(...)` i `vkCmdBindDescriptorSets(...)` te pokrenuti sjenčar.

3.2.3. Organizacija podataka u memoriji

Da bi sjenčar mogao generirati trokute potrebno je rezervirati memoriju u koju će zapisivati rezultate. Problem je što mi ne znamo unaprijed koliko memorije je potrebno da bi zapisali sve trokute. Neki od pristupa su:

- pretpostaviti koliko memorije je potrebno u najgorem slučaju i onda rezervirati dovoljno. Uzmemo li u obzir da su podaci organizirani u 3D prostoru, to znači da veličina memorije kubno ovisi o dimenzijama. Najgori slučaj bio bi kada bi svaki voksel generirao 5 trokuta. Trokut se sastoji od 3 vrha, a svaki vrh od 3 koordinate u prostoru i boje. Znači da bi jedan voksel zauzeo 5 x 3 x 4 = 60 jedinica memorije tipa *float*. Svaki *float* je 4 bajta što znači da svaki voksel zauzima 60 * 4 = 240 bajtova
- pokrenuti algoritam i samo prebrojati koliko trokuta bi nastalo, ali nigdje ih zapravo ne zapišemo. Zatim bi rezervirali dovoljno memorije i pokrenuli sjenčar koji bi popunio memoriju trokutima. Ovako točno znamo koliko memorije nam je potrebno i ne rezerviramo višak.

Teško je napraviti usporedbu ova dva pristupa jer u drugom slučaju memorijski zahtjevi ovise o granici koju smo zadali algoritmu. Izabrane su vrijednosti granice koje imaju smisla. Tablica 3.1 prikazuje usporedbu između ova dva pristupa na pravim podacima dobivenim *MRI* ili *CT* uređajima.

Tablica 3.1 Usporedba metoda rezerviranja memorije

Dimenzija skalarnog polja	Prva metoda (najgori slučaj)		Druga metoda (izračun unaprijed)	
	Broj trokuta	Veličina memorije	Broj trokuta	Veličina memorije
176 x 256 x 256	56,896,875	2,605 MB	2,725,972	125 MB
512 x 512 x 113	146,227,760	6,694 MB	1,455,746	67 MB
512 x 512 x 258	335,540,485	15,360 MB	3,989,990	183 MB

Vidimo da je za velike dimenzije skalarnog polja gotovo nemoguće koristiti pristup u kojem uzimamo najgori slučaj. Drugi pristup zauzima red veličine manje memorije.

Naš program prvo će pokrenuti *compute shader* koji će izbrojati koliko trokuta bi nastalo pokretanjem algoritma pokretnih kocaka, ali te trokute neće nigdje zapisati. Zatim

pokrećemo drugi *compute shader* koji radi sličan posao i zapisuje trokute u memoriju. Za potrebe vizualizacije ovo je dovoljno. Na ekran možemo iscrtati generirane trokute.

Sjenčar koji provodi algoritam radi u paralelnom načinu rada. To znači da u jednom ciklusu instrukcija grafička kartica obradi veći broj podataka od jednom. Svi dijelovi algoritma mogu raditi paralelno neovisni jedni o drugima sve dok ne dođemo do dijela gdje trebamo zapisati vrhove trokuta u memoriju. Zapisivanje se provodi tako da postoji globalna varijabla koja sadrži indeks zadnjeg trokuta u memoriji. Trenutna dretva sjenčara pročita indeks te ga zatim poveća i svoje trokute zapiše na novo mjesto. Budući da čitanje i zapisivanje traje duže od jednog ciklusa, moguće je da dvije dretve pročitaju isti indeks i zapišu svoje trokute na isto mjesto. Ovo je problem, ali lagano se rješava korištenjem atomičkih operacija. Primjerice funkcija `atomicAdd()` osigurava da samo jedna dretva piše na memorijsku lokaciju u jednom trenutku. Problem se pojavljuje kad u sustav želimo ubaciti skulpturiranje, odnosno modeliranje podataka.

Nakon što sjenčar generira trokute zapravo ne znamo gdje je koji trokut završio u memoriji. Znamo koliko ih ima i da su složeni jedan iza drugog, ali vrlo teško je povezat koji trokut je nastao iz kojeg voksela. To je problem jer kod skulpturiranja mijenjamo vrijednosti skalarnog polja i potrebno je ponovno izračunati kako izgleda geometrija na tom dijelu 3D modela. Ne možemo znati koje trokute obrisati i gdje se nalaze.

Najjednostavnije rješenje je ponovno pokrenuti sjenčar za računanje memorije i ponovno generirati sve trokute. Ovaj pristup nije prihvatljiv jer taj postupak traje oko jednu sekundu. To bi značilo da svaki put kad skulpturiramo model moramo čekat jednu sekundu da vidimo rezultate. Interaktivnost je jako loša i skulpturiranje je sporo.

Bilo bi dobro kad bi mogli ponovno izračunati samo onaj dio koji se promijenio. To je teško jer ne znamo gdje su trokuti u memoriji. Mogli bi pamtiti gdje je početak svakog voksela, ali to bi zauzimalo jako puno memorije. Ideja je da prostor podijelimo na veće dijelove odnosno blokove (engl. *chunk*) i onda pamtimo početak svakog bloka u memoriji. Kada skulpturiramo označit ćemo koje blokove ponovno treba generirati algoritmom. Ovako osiguravamo da računanje algoritma možemo provesti u stvarnom vremenu, odnosno da ne primjećujemo vrijeme koje je potrebno da se provede algoritam.

3.2.3.1 Struktura `Chunk`

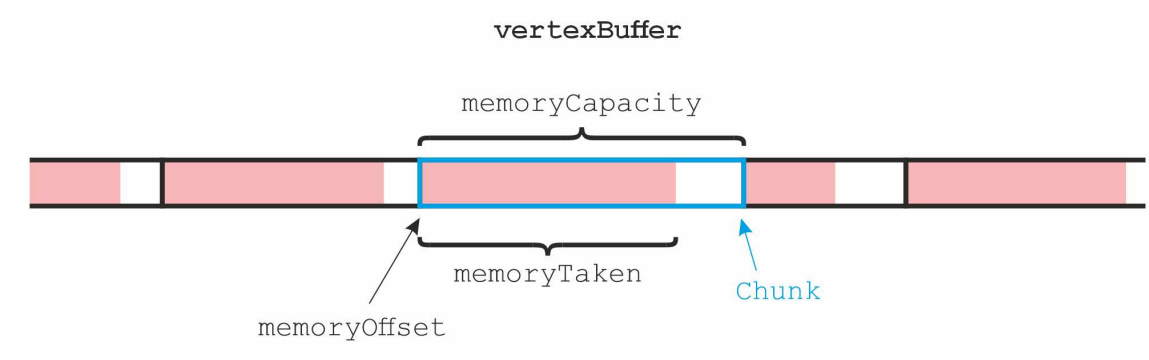
Klasa `MarchingCubesModel` sadrži vektor u koji su spremljeni svi blokovi tipa `Chunk`. Struktura `Chunk`:

```
struct Chunk {
    int32_t sizeX, sizeY, sizeZ;
    int32_t triangleCount;

    int32_t memoryOffset;   // Offset in the GPU memory
    int32_t memoryTaken;    // Memory taken in the GPU memory
    int32_t memoryCapacity; // Capacity in GPU memory

    int32_t indexOffset;    // Offset in the index buffer
    int32_t indexCapacity;  // # of indices in index buffer

    int32_t isDirty;
};
```

U strukturi postoje varijable `sizeX`, `sizeY` i `sizeZ` koje opisuju dimenzije bloka. Iako su svi blokovi u pravilu iste dimenzije, ipak postoje slučajevi u kojima trebamo znati njegove dimenzije. Primjerice na rubovima skalarnog polja blok ne mora biti istih dimenzija kao na sredini. Varijabla `triangleCount` sadrži broj trokuta u pojedinom bloku. Tu varijablu koristimo na mnogo mjesta, a ponajviše da bi znali koliko memorije je potrebno rezervirati. U nju prvi sjenčar zapisuje koliko trokuta bi nastalo algoritmom pokretnih kocaka.

Ostale varijable `memoryOffset`, `memoryTaken` i `memoryCapacity` služe da bi mogli znati gdje je blok u memoriji. Također pokazuju koliko prostora je rezervirano za pojedini blok i koliko prostora je on stvarno zauzeo. Time omogućujemo da se memorija ne mora rezervirati svaki put kad se poveća broj trokuta u bloku, već samo onda kad ponestane prostora. Slika 3.1 prikazuje što znače pojedine varijable u memoriji koja sadrži generirane trokute.

Postoje još tri varijable, dvije su manje bitne i odnose se na poziciju i veličinu u memoriji koja sadrži indekse potrebne da se iscrtaju linije između trokuta. Varijabla `isDirty` je zastavica koja označava je li taj blok potrebno ponovno izračunati. Nju postavljamo kad promijenimo vrijednost u skalarnom polju. Treba paziti da označimo susjedne blokove ako je vrijednost koju mijenjamo na granici.



Slika 3.1 Prikaz organizacije memorije u grafičkoj kartici

U nastavku je popis svih memorijskih spremnika (engl. *buffer*), te njihov opis, koje sadrži klasa `MarchingCubesModel`:

- `chunkBuffer` – spremnik koji sadrži popis svih blokova i njihove podatke. U njega prvi sjenčar (*calculateMemory.comp*) zapisuje koliko će trokuta nastati u svakom bloku. Koristi se još kod detektiranja da li bloku ponestaje memorije, odnosno jesu li se varijable `memoryTaken` i `memoryCapacity` dovoljno približile
- `volumeBuffer` – spremnik u kojem su zapisane skalarne vrijednosti. Skulpturiranjem direktno mijenjamo vrijednosti u njemu
- `triTableBuffer` – spremnik koji sadrži *lookup* tablicu
- `paramsBuffer` – spremnik u kojem su zapisani parametri potrebni za provođenje algoritma: granica, veličina bloka, dimenzija prostora u XYZ, zastavica za interpolaciju
- `vertexBuffer` – u ovaj spremnik sjenčar zapisuje vrhove trokuta. Trokuti istog bloka zapisani su slijedno, a iza njih dolazi prazan prostor koji je rezerviran za taj blok. Dok ima prostora on se popunjava, a ako ponestane ponovno je potrebno rezervirati memoriju
- `indexBuffer` – sadrži jednostavan popis indeksa trokuta koje treba spojiti da bi dobili linije između trokuta, primjerice: [0, 1, 1, 2, 2, 0, 3, 4, 4, 5, 5, 3, ...]. To je lista u kojoj se spajaju vrhovi dva po dva. Prvo 0-1, pa 1-2 i na kraju 2-0 da bi „zaokružili“ trokut. Ne postoji jednostavniji način da se dobiju linije između trokuta. Treba napraviti novi *pipeline* i navesti popis vrhova koje spajamo. Jedan od problema je što ovim pristupom svaku liniju crtamo dva puta jer trokuti dijele stranice.

Osim ovih memorijskih spremnika postoji još par u klasi `VolumeRenderer`:

- `fragmentParamsBuffer` – sadrži parametre koje korist *fragment shader*. Parametri su boja i prozirnost
- `indirectBuffer` – spremnik u kojem su spremljene naredbe za crtanje. Inače je potrebno pozvati funkciju `vkCmdDraw(...)` i popuniti parametre kao što su broj točaka i pokazivač na spremnik s podacima. Problem je što naša implementacija koristi prazan prostor u memoriji da bi osigurali dovoljno prostora. Ako pozovemo `vkCmdDraw()` dobit ćemo neželjene rezultate jer će se iscrtavati podaci koji nisu dio modela već smeće u memoriji. Mogli bi pozvati `vkCmdDraw()` jednom za svaki blok, ali to je računski skupo. Rješenje je funkcija `vkCmdDrawIndirect()`. Kao parametre prima memorijski spremnik koji sadrži elemente strukture `VkDrawIndirectCommand`, a svaki element onda sadrži početak u memoriji, broj točaka i broj instanci. To je zapravo popis naredbi za crtanje. Popis je potrebno izgraditi svaki put kad skulpturiramo model
- `indirectIndexBuffer` – slično kao i `indirectBuffer`. Umjesto naredbi za crtanje trokuta u njega spremamo naredbe za crtanje linija.

3.2.4. Kôd sjenčara (*shader*)

Budući da sjenčari svoj kôd izvode paralelno potrebno je unaprijed odrediti koliko dretvi želimo pozvati. Moramo pozvati dovoljno dretvi da svaki voksel bude dretva za sebe. Potrebno je odrediti veličinu radnih grupa i koliko ih pozvati. Ove brojeve odredio sam testirajući performanse s različitim kombinacijama:

Tablica 1 Ovisnost broja sličica u sekundi o veličini radne grupe

Veličina radne grupe (X, Y, Z)	Ukupna veličina radne grupe	Broj sličica u sekundi (FPS)
1, 1, 1	1	~88
2, 2, 2	8	~370
4, 4, 2	32	~651
4, 4, 4	64	~704
8, 4, 4	128	~730
8, 8, 4	256	~713
8, 8, 8	512	~714

3.2.4.1 Sjenčar *calculateMemory.comp*

Sjenčar *calculateMemory.comp* služi da bi prebrojao koliko trokuta će nastati provođenjem algoritma pokretnih kocaka. Sjenčar kao ulaz prima 4 memorijska spremnika: `chunkBuffer`, `volumeBuffer`, `triTableBuffer`, `paramsBuffer`.

```
uvec3 globalVoxel = gl_GlobalInvocationID;

if (globalVoxel.x >= uint(volumeSizeX - 1) ||
    globalVoxel.y >= uint(volumeSizeY - 1) ||
    globalVoxel.z >= uint(volumeSizeZ - 1)) {
    return;}
```

Sjenčar započinje dohvaćanjem vrijednost svoje invokacije. To su zapravo koordinate u 3D prostoru jer je veličina radne grupe također zadana u 3 dimenzije. Sjenčar će u varijablu `globalVoxel` dobiti neke brojeve kao što su (132, 45, 77) i oni zapravo predstavljaju voksel na tim koordinatama. Zatim slijedi provjera je li taj voksel unutar dimenzija skalarnog polja. Ako dimenzije skalarnog polja nisu višekratnik veličine radne grupe onda će se na rubovima pokrenuti dretve za voksele koji ne postoje.

```
int cubeValues[8];
for (int i = 0; i < 8; ++i) {
    ivec3 corner = ivec3(globalVoxel) + cornerOffsets[i];
    int index = corner.x +
            corner.y * volumeSizeX +
            corner.z * volumeSizeX * volumeSizeY;
    cubeValues[i] = volumeData[index];
}
```

U polje `cubeValues[8]` ćemo zapisati vrijednosti svih vrhova trenutnog voksela. To radimo tako da prolazimo kroz petlju i u varijablu `corner` računamo koordinate vrha kojeg trenutno gledamo. Računamo tako da na koordinate voksela dodajemo pomake spremljene u polje `cornerOffsets`. Računamo njegov indeks u jednodimenzionalnom memorijskom spremniku, te čitamo i zapisujemo vrijednost.

```
const ivec3 cornerOffsets[8] = ivec3[](
    ivec3(0,0,0), ivec3(1,0,0), ivec3(1,1,0), ivec3(0,1,0),
    ivec3(0,0,1), ivec3(1,0,1), ivec3(1,1,1), ivec3(0,1,1)
);
```

Ostaje nam pronaći taj slučaj u *lookup* tablici i prebrojati trokute. Da bi dobili indeks retka u tablici računamo `cubeIndex`. To radimo tako da provjeravamo je li pojedini vrh unutar granice vrijednosti `limit`. Ako je unutar granice napravit ćemo bitmasku koja ima jedinicu na `i`-toj poziciji - `(1 << i)`. Na kraju radimo bitovnu `OR` operaciju nad `cubeIndex` i bitmaskom. Budući da je `cubeIndex` varijabla tipa `int` u njoj je već zapisan indeks retka *lookup* tablice.

```
int cubeIndex = 0;
for (int i = 0; i < 8; ++i) {
    if (cubeValues[i] >= limit) {
        cubeIndex |= (1 << i);
    }
}
```

Brojanje trokuta ostvarujemo tako da će svaka dretva zapisati koliko trokuta je nastalo za njen voksel. Pomoćna funkcija `triTableValue(int i, int j)`, vraća vrijednost tablice u retku `i` na mjestu `j`. Gledamo svaki treći element te ako je različit od -1 znamo da on i sljedeća dva tvore trokut. Kada nađemo vrijednost -1 došli smo do kraja retka.

```
int triangleCount = 0;
for (int i = 0; i < 5; ++i) { // max 5 triangles per cube
    int a = triTableValue(cubeIndex, i * 3);

    if (a == -1) {
        break;
    }
    ++triangleCount;
}
```

Broj trokuta zapisujemo u varijablu koja se nalazi u `chunkBuffer`-u. Treba paziti da koristimo funkciju `atomicAdd()` kako bi pravilno izbrojali trokute.

```
...
atomicAdd(chunks[chunkIndex].triangleCount, triangleCount);
```

Ovime završava prvi sjenčar i sada je moguće rezervirati dovoljno memorije za spremnik `vertexBuffer`.

3.2.4.2 Sjenčar *marchingCubes.comp*

Drugi sjenčar (*marchingCubes.comp*) zadužen je da u memoriju zapiše vrhove trokuta. Prvo što sjenčar radi je provjerava je li njegov blok potrebno ponovno računati. Na početku sve blokove treba računati.

Sljedeći dio sjenčara je identičan kao i kod prošlog. Provjerava je li unutar dimenzija skalarnog polja, dohvaća vrijednosti vrhova voksela, računa redak u tablici te računa koliko trokuta će generirati.

Da bi dretva sjenčara znala gdje u memoriju zapisati svoje trokute treba nekako komunicirati s ostalim dretvama tako da dvije dretve ne bi pisale na isto mjesto. U strukturi `Chunk` postoji varijabla `memoryTaken`. Njena vrijednost je trenutno 0. Svaka dretva sjenčara će pročitati vrijednost u toj varijabli i onda ju povećati za broj trokuta koja ta dretva generira. To znači da će prva dretva pročitati vrijednost 0 i povećati ju za neku vrijednost. Prva dretva svoje trokute počinje pisati na pomaku (engl. *offset*) od 0, a sljedeća će imati pomak koji odgovara vrijednosti što je prva dretva povećala. Treća će onda uzeti broj koji je zapisala druga.

```
int localOffset = atomicAdd(chunks[chunkIndex].memoryTaken,
triangleCount * 3);
```

Vrijednost pomaka spremljena je u varijablu `localOffset`. Povećavamo vrijednost `memoryTaken` za `triangleCount` x 3 jer svaki trokut ima tri vrha. Dalje ulazimo u petlju i dohvaćamo prva tri brida koja trebamo spojiti. Bridovi su zapravo vrijednosti od 0 do 11. Na primjer brid 9 je brid između vrhova 1 i 5 (Slika 2.3). Te indekse je potrebno pretvorit u koordinate spremljene u varijable `pos0`, `pos1` i `pos2`. Pomoćna funkcija `interpolateEdge(uvec3 voxel, int edge)` prima koordinate voksela i indeks brida, te vraća interpolirane koordinate između vrhova pojedinog brida.

```
for (int tri = 0; tri < triangleCount; ++tri) {
    int edge0 = triTableValue(cubeIndex, tri * 3 + 0);
    int edge1 = triTableValue(cubeIndex, tri * 3 + 1);
    int edge2 = triTableValue(cubeIndex, tri * 3 + 2);

    vec3 pos0 = interpolateEdge(globalVoxel, edge0);
    vec3 pos1 = interpolateEdge(globalVoxel, edge1);
    vec3 pos2 = interpolateEdge(globalVoxel, edge2);
```

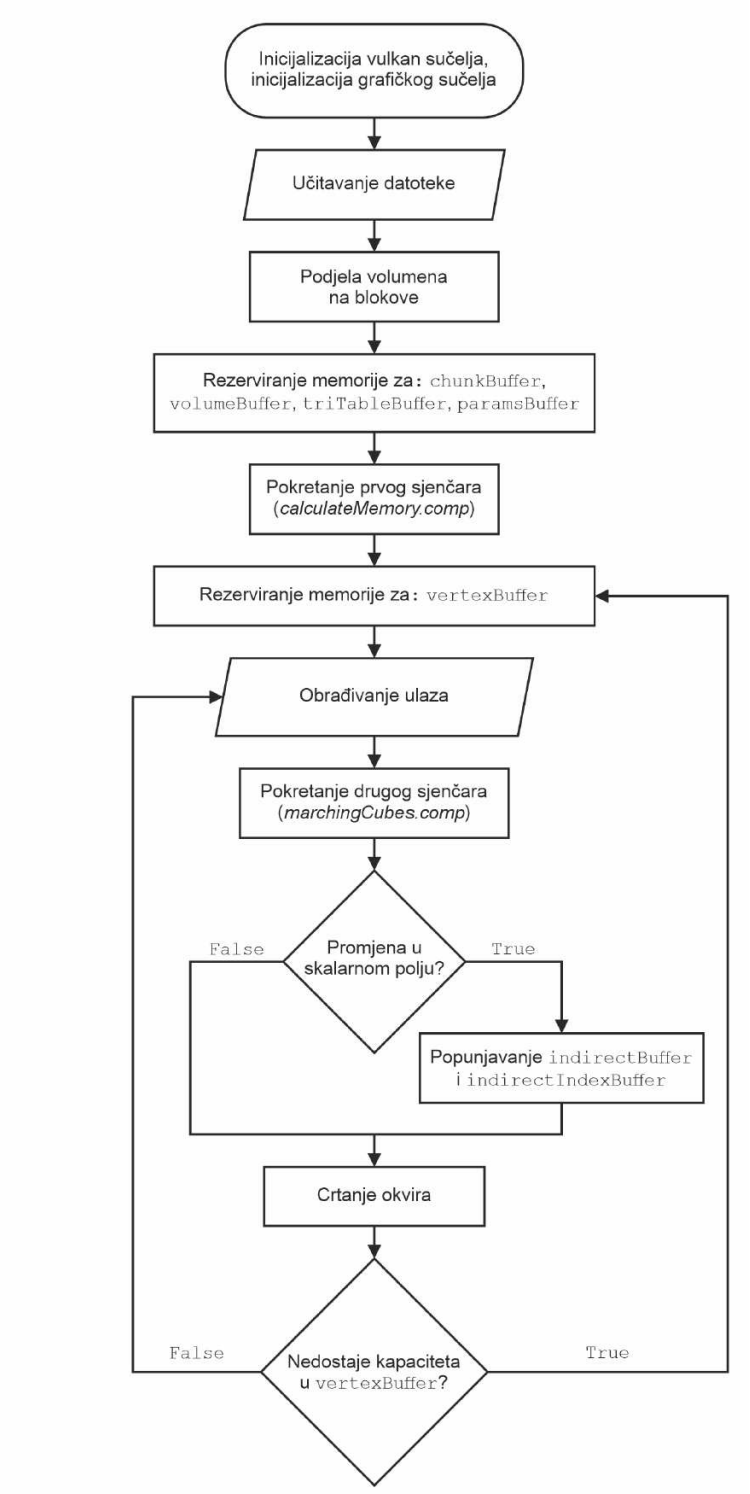
Ostaje još zapisati trokute u memoriju, varijabla `baseIndex` sadrži indeks u memoriji `indexBuffer`. Indeks se računa iz pomaka pojedinog bloka, lokalnog pomaka dretve te pomaka što je dretva već upisala.

```
        int baseIndex = (chunks[chunkIndex].memoryOffset / 16) +
                         localOffset + tri * 3;

        vertices[baseIndex + 0] = vec4(pos0, 1.0);
        vertices[baseIndex + 1] = vec4(pos1, 1.0);
        vertices[baseIndex + 2] = vec4(pos2, 1.0);
    }
```

Moramo `chunks[chunkIndex].memoryOffset` podijeliti sa 16 jer je u njoj zapisana veličina u bajtovima, a nama je potreban indeks. Svaka točka sadrži 4 vrijednosti (poziciju i boju) i svaka vrijednost je tipa *float* veličine 4 bajta.

Ovime završava sjenčar i u spremniku `vertexBuffer` imamo zapisane koordinate trokuta koji čine 3D objekt.

3.3. Tok programa (engl. *flowchart*)

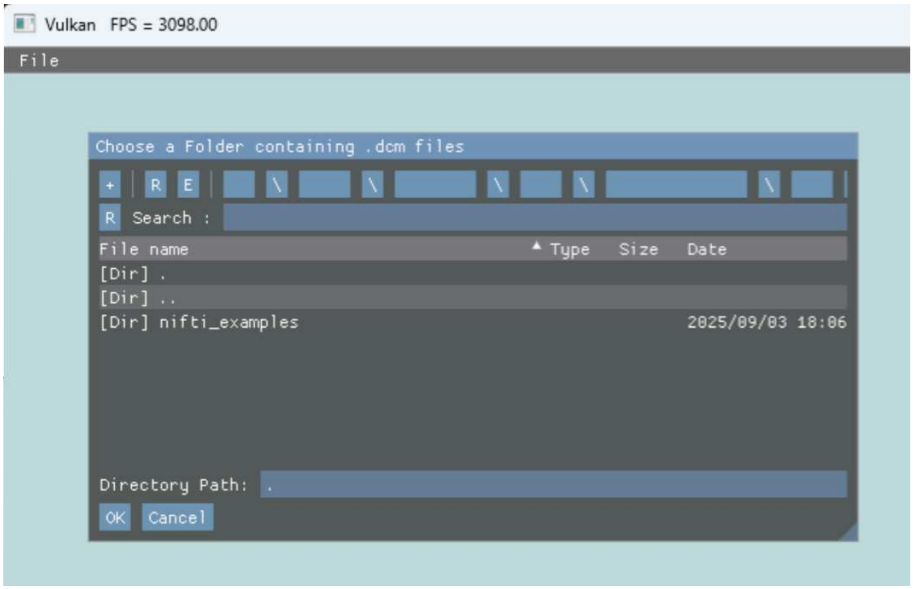


Slika 3.2 Slika prikazuje vizualizaciju tôka programa

Vizualizacija na slici 3.2 prikazuje samo ključne elemente programa. Nisu sve funkcije točno tim redoslijedom pozvane.

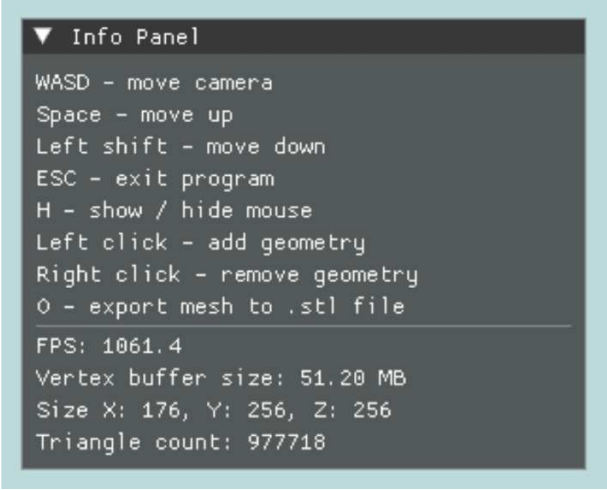
4. Rezultati

Pri pokretanju programa prvo je potrebno izabrati datoteku koju želimo otvoriti. Pritiskom na gumb **File** otvara se izbornik gdje je moguće izabrati **Open file** ili **Open folder**. Ako želimo otvoriti *.nii* datoteku ili *.dcm* datoteku onda odabiremo **Open File**, a ako otvaramo direktorij koji sadrži više *.dcm* datoteka onda odabiremo **Open folder**.



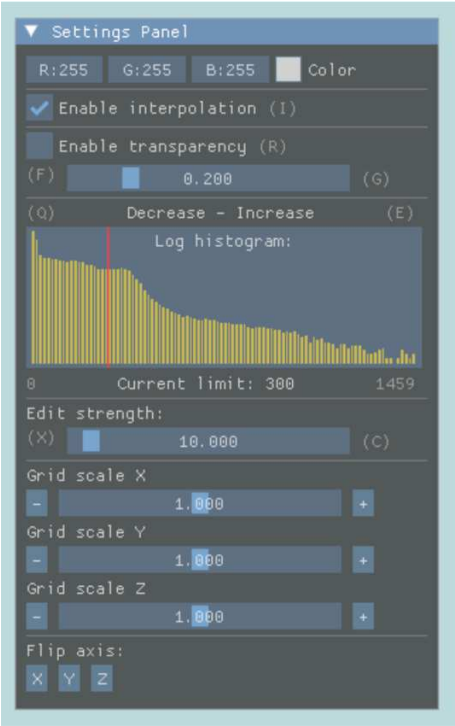
Slika 4.1 Izbornik datoteka

Nakon toga pokreće se algoritam i na ekranu se iscrtava 3D model. S lijeve strane ekrana prikazan je prozor koji objašnjava kontrole programa. Također navodi neke parametre programa: broj sličica u sekundi, veličinu memorije koju zauzimaju trokuti, dimenzije skalarnog polja te broj trokuta u sceni.



Slika 4.2 Prozor s kontrolama i parametrima

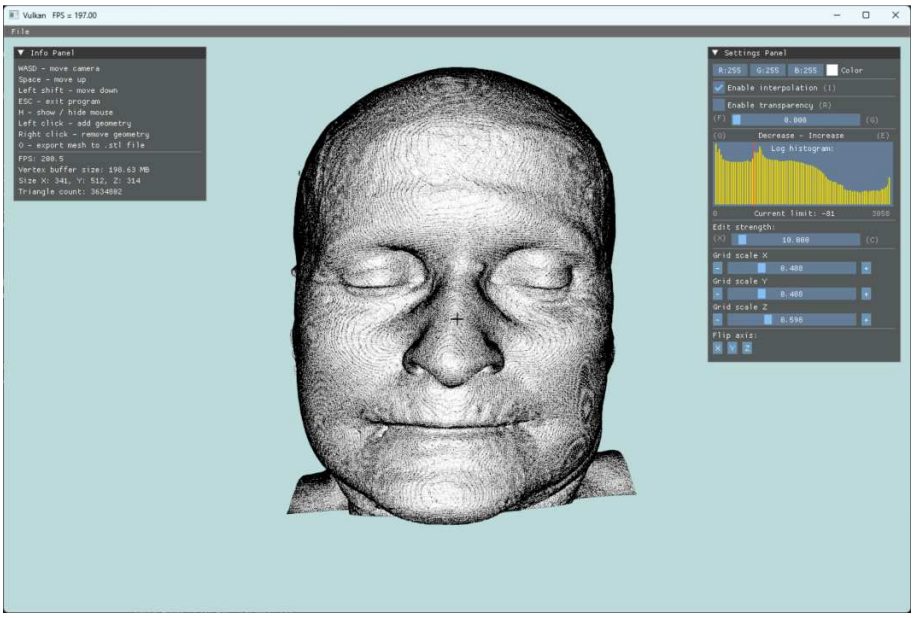
S desne strane ekrana nalazi se prozor u kojem je moguće podešavati parametre algoritma. Moguće je mijenjati boju trokuta, uključiti ili isključiti interpolaciju i podesiti transparentnost. Također prikazan je graf koji pokazuje histogram svih vrijednosti skalarnog polja. Klikom miša na graf moguće je pomaknuti granicu pripadnosti. Osim toga prikazuje i snagu kojom skulpturiramo podatke.



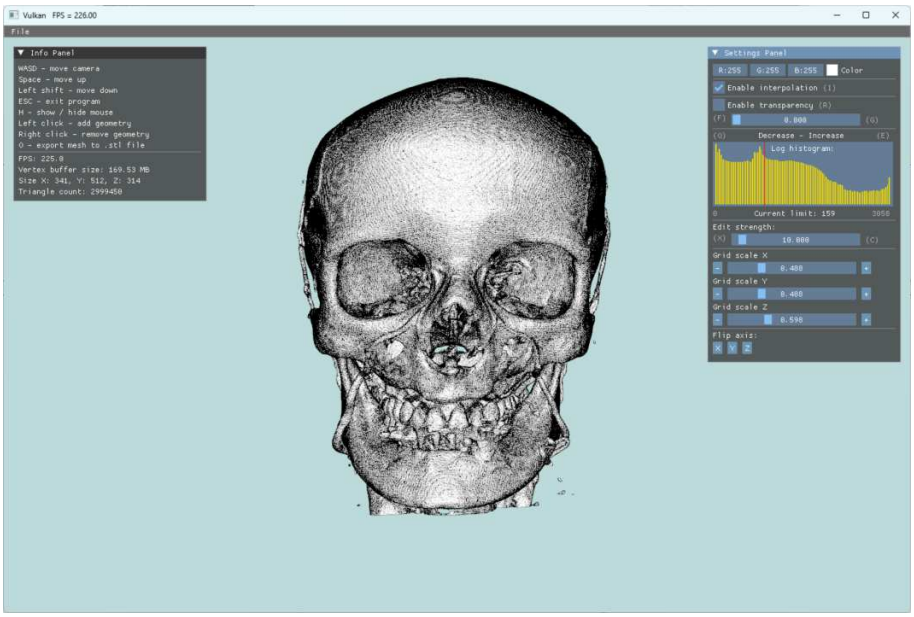
Slika 4.3 Prozor s parametrima algoritma

Na dnu se nalaze opcije za podešavanje razmaka između voksela. Inicijalne vrijednosti tih parametara pročitaju se iz ulazne datoteke, ali u nekim slučajevima tih podataka uopće nema ili su pogrešni. Zbog toga ih je moguće ručno podesit. Također postoje gumbi za okretanje smjera osi.

Rezultati algoritma prikazani su na slikama 4.4 i 4.5. Modeli su nastali iz istih volumetrijskih podataka, jedina razlika je vrijednost granice koju smo koristili. Na slici 4.4 granica je postavljena na -81 i pokazuje kožu lica. Isti volumen je na slici 4.5, ali vrijednost granice je 159, a rezultat algoritma pokazuje lubanju. Vidimo da različitim odabirom granice možemo dobiti potpuno različite rezultate.

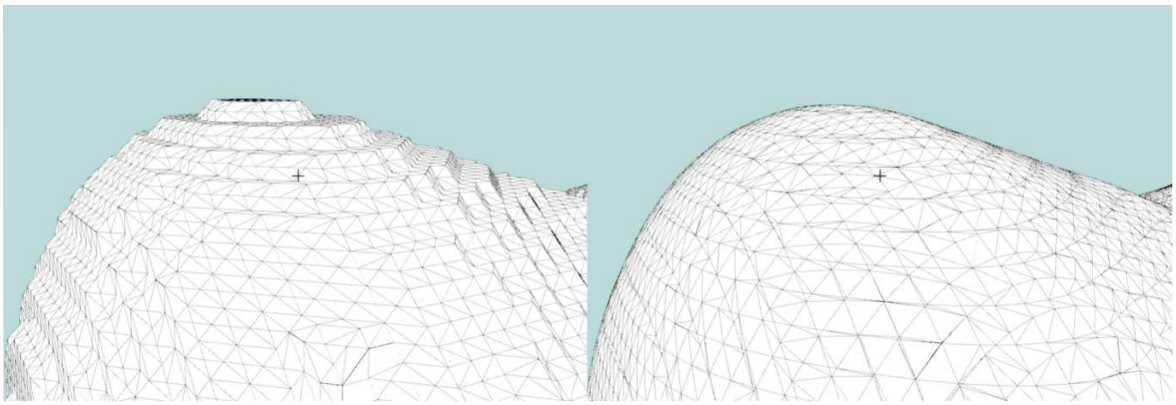


Slika 4.4 Prikaz volumena s granicom postavljenom na -81



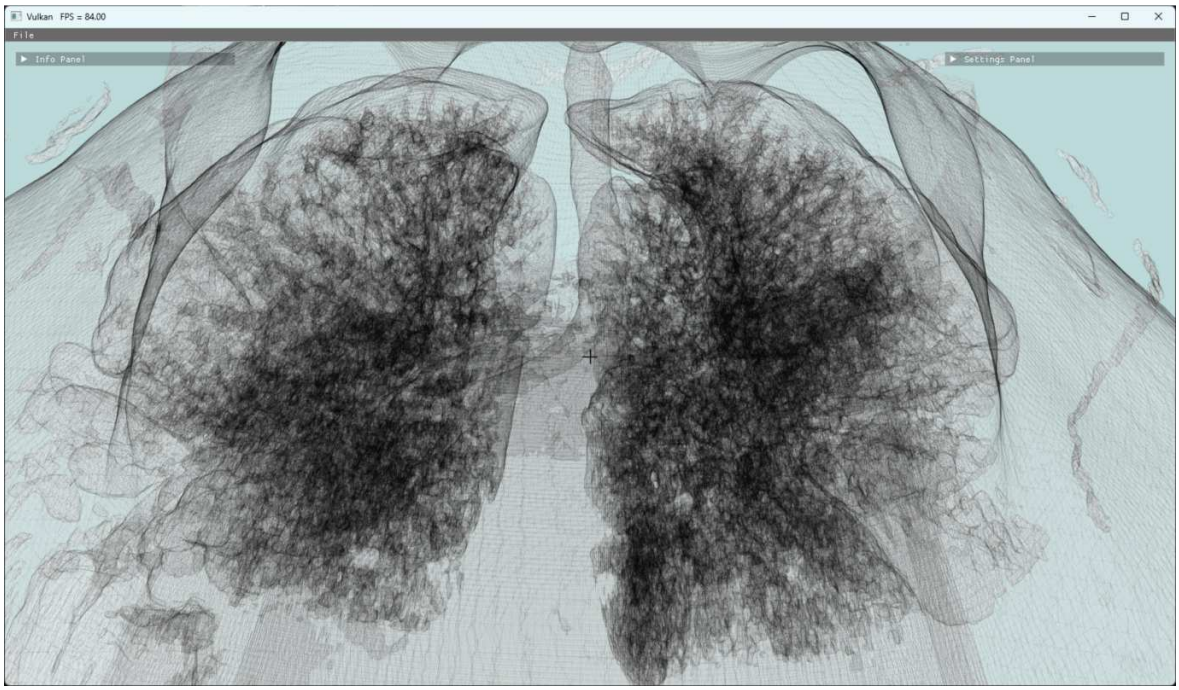
Slika 4.5 Prikaz volumena s granicom postavljenom na 159

Na slici 4.6 prikazana je razlika u načinu izračuna vrhova točaka. S lijeve strane na slici prikazan je izračun bez korištenja interpolacije, a s desne strane je interpolacija uključena. Na djelu slike bez interpolacije svi trokuti imaju nagib od 0°, 45° ili 90°. Možemo čak i prepoznati pojedine voksele. S druge strane na dijelu s interpolacijom trokuti mogu biti pod bilo kojim nagibom, ovisno o vrijednostima u vrhovima voksela. Rezultati su puno bolji s interpolacijom.



Slika 4.6 Isti dio modela bez interpolacije (lijevo) i s interpolacijom (desno)

Zadnja postavka odnosi se na način kako se trokuti iscrtavaju. Opcija **Enable Transparency** postavlja vrijednost alfa kanala trokuta na broj koji smo zadali. Time možemo simulirati neku vrstu rendgena. Nema utjecaja na poligonalnu mrežu već samo na vizualizaciju. Primjer koji pokazuje pluća je na slici 4.7.



Slika 4.7 Rezultat algoritma s prozirnim trokutima

Zaključak

U ovom radu prikazani su postupci modeliranja i vizualizacije volumetrijskih podataka s naglaskom na podatke dobivene u medicini. Objašnjen je algoritam pokretnih kocaka. Njime je ostvarena konstrukcija trodimenzionalnih objekata iz volumetrijskih podataka. Skulpturiranje podataka ostvareno je detekcijom presjeka zrake i volumena. Opisani su neki pristupi i algoritmi, a implementiran je algoritam DDA.

Pokazani su koraci koje treba napraviti za izradu Vulkan aplikacije. Detaljno su objašnjeni detalji vezani uz stvaranje sjenčara. Opisana ja je efikasna implementacija algoritma pokretnih kocaka koja se izvodi u stvarnom vremenu. Prikazan je efikasan način organizacije memorije u memorijskom spremniku što je ključno za skulpturiranje u stvarnom vremenu. Volumen je podijeljen na blokove, a algoritam ponovno pokrećemo samo na bloku koji je mijenjan.

Rezultat implementacije je program u kojem je moguće interaktivno mijenjati parametre algoritma pokretnih kocaka. Algoritmom nastaje poligonalna mreža koju je moguće skulpturirati i izvesti u *.stl* formatu za daljnju obradu ili korištenje.

Literatura

[1] Paul Bourke, svibanj 1994., *Polygonising a scalar field*. Poveznica: https://paulbourke.net/geometry/polygonise/; pristupljeno 3. ožujka 2025.

[2] William E. Lorensen i Harvey E. Cline, 01. kolovoza 1987., *Marching cubes: A high resolution 3D surface construction algorithm*. Poveznica: https://dl.acm.org/doi/10.1145/37402.37422; pristupljeno 3. ožujka 2025.

[3] Overvoorde A., *Vulkan Tutorial*. Poveznica: https://vulkan-tutorial.com/; pristupljeno 24. travnja 2025.

[4] *Vulkan Documentation*. Poveznica: https://docs.vulkan.org/spec/latest/index.html; pristupljeno 24. travnja 2025.

[5] javidx9, 28. veljače 2021., *Super Fast Ray Casting in Tiled Worlds using DDA*. Poveznica: https://www.youtube.com/watch?v=NbSee-XM7WA; pristupljeno: 25. lipnja 2025.

[6] OpenNEURO. Poveznica: https://openneuro.org/; pristupljeno 1. lipnja 2025.

[7] embodi3D. Poveznica: https://www.embodi3d.com/; pristupljeno 1. lipnja 2025.

[8] The Cancer Imaging Archive. Poveznica: https://www.cancerimagingarchive.net/; pristupljeno 1. lipnja 2025.

[9] Vulkan Memory Allocator. Poveznica: https://gpuopen.com/vulkan-memory-allocator/; pristupljeno 7. svibnja 2025.

[10] GLFW. Poveznica: https://www.glfw.org/; pristupljeno 16. travnja 2025.

[11] nifticlib. Poveznica: https://sourceforge.net/projects/niftilib/files/nifticlib/; pristupljeno 28. travnja 2025.

[12] ImGui. Poveznica: https://github.com/ocornut/imgui; pristupljeno 29. lipnja 2025.

Sažetak

Modeliranje i vizualizacija volumetrijskih podataka

Ovaj rad objašnjava neke postupke modeliranja i vizualizacije volumetrijskih podataka primarno dobivenih u medicini. Detaljno je opisan algoritam pokretnih kocaka za vizualizaciju podataka. Za detekciju presjeka zrake s volumenom korišten je algoritam DDA. Programska implementacija napisana je koristeći grafičko sučelje Vulkan i programski jezik C++. Objašnjeni su postupci potrebni za pisanje sjenčara. Opisana je organizacija podataka u memoriji te načini upravljanja memorijom. Rezultati pokazuju da algoritam uspješno generira poligonalnu mrežu koja se može izvesti u *.stl* formatu za daljnju upotrebu.

Ključne riječi: modeliranje, vizualizacija, pokretne kocke, DDA, vulkan, sjenčar

Summary

Modeling and visualization of volumetric data

This paper explains some procedures for modeling and visualization of volumetric data primarily obtained in medicine. It provides a detailed description of the marching cubes algorithm for data visualization. The DDA algorithm was used to detect the intersection of a ray with a volume. The implementation was developed in C++ using the Vulkan graphics API, with emphasis on shader development, data organization in memory and memory management techniques. The results demonstrate that the algorithm successfully generates a polygonal mesh, which can be exported as an .stl file for further use.

Keywords: modeling, visualization, marching cubes, DDA, vulkan, shader

Privitak

Izvorni kod dostupan je na poveznici: https://gitlab.com/zzee/diplomski_rad