SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7153

OSTVARIVANJE PRIKAZA TEMELJENOG NA PROSTORU VOLUMNIH ELEMENATA

Karlo Miličević

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7153

OSTVARIVANJE PRIKAZA TEMELJENOG NA PROSTORU VOLUMNIH ELEMENATA

Karlo Miličević

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 7153

Pristupnik:	**Karlo Miličević (0036506449)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak: **Ostvarivanje prikaza temeljenog na prostoru volumnih elemenata**

Opis zadatka:

Proučiti grafičke scene koje se temelje na prostoru volumnih elemenata. Proučiti načine ostvarivanja prikaza temeljenih na prostoru volumnih elemenata. Posebice obratiti pažnju na metode koje transformiraju modele temeljene na volumnim elementima u geometrijske primitive. Razmotriti mogućnosti implementacije na grafičkom procesoru. Ostvariti prikaze dobivenih rezultata. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++, odgovarajuću grafičku biblioteku te biblioteku za izradu grafičkog sučelja Dear ImGui. 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: 11. lipnja 2021.

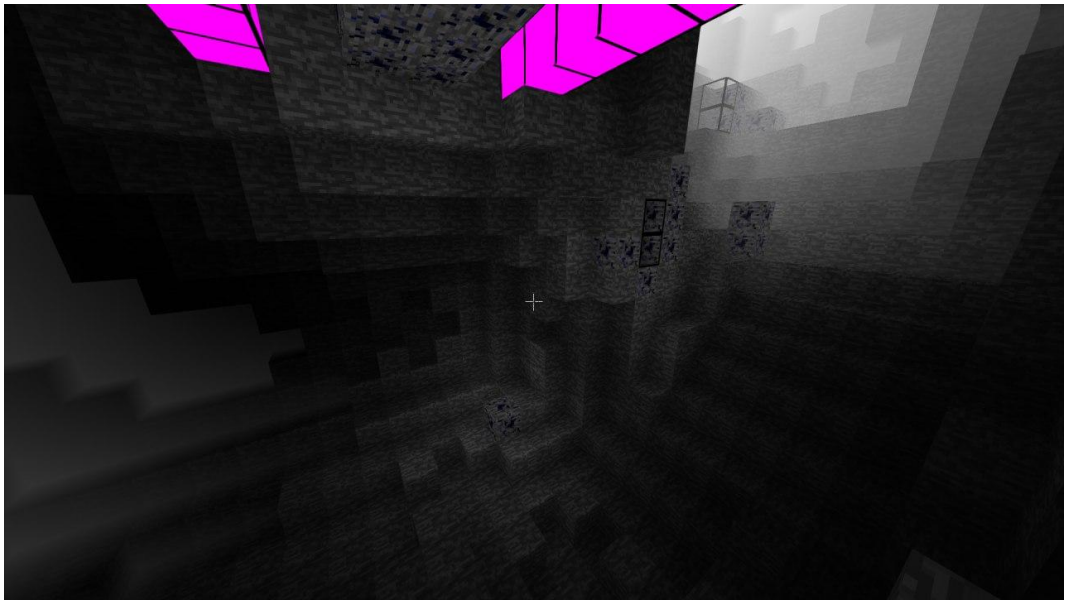
SADRŽAJ

1. Uvod

Predstavljanje i prikazivanje trodimenzionalnog prostora nema jedno rješenje. Kroz godine razvile su se razne tehnike i prikaza i predstavljanja.

Ovaj rad je usredotočen na implementacije predstavljanja i prikazivanja prostora volumnim elementima – prvenstveno prikaz takvog prostora "kockastim" modelom. Prvo ćemo opisati načine pamćenja svijeta, a zatim načine prikazivanja svijeta. Kod prikazivanja ćemo pokazati razna poboljšanja kojima smanjujemo količinu memorije koja je potrebna, te poboljšanja kojima smanjujemo broj lica koje prikazujemo.

Prokaz svijeta volumnim elementima nije toliko često kao trokutima, ali je svakako zanimljiv pristup jer nudi različite mogućnosti za implementaciju drugih algoritama i optimizacija. Čest je odabir kod simulacije fluida.



Slika 1.1: Rezultat

Na slici 1.1 prikazan je produkt praktičnog dijela.

2. Korišteni alati

2.1. Programski jezik C++

Kao jezik implementacije algoritama odabran je C++. C++ je čest izbor u razvoju računalnih igara zbog svoje raširenosti i brzine. Pogodan je i jer dopušta određivanje rasporeda dijelova struktura u memoriji i time olakšava ispravno prenošenje podataka na grafičko sklopovlje. U implementacijama je korišten dio C++ koji se gotovo izravno može prenijeti u programski jezik C.

2.2. Biblioteka Vulkan

Vulkan je moderna biblioteka za grafiku. Korištena je inačica 1.2 u svrhu ostvarivanja prikaza uz korištenje grafičkog sklopovlja za tu namjenu. Biblioteka Vulkan pruža API varijantu za programski jezik C i za programski jezik C++. Korištena je varijanta za programski jezik C prvenstveno zbog brzine prevođenja u odnosu na C++ varijantu. Osim zbog brzine prevođenja, veća je pokrivenost i bolja dokumentacija za C varijantu.

Moguće je izvesti iste rezultate i pomoću biblioteke OpenGL, ali ona pruža manje mogućnosti od biblioteke Vulkan kad je potrebno direktnije upravljati sklopovljem.

2.3. Biblioteka SDL2

SDL2[1] je biblioteka za olakšavanje koraka pri stvaranju grafičkog prozora, puštanju zvukova, dohvaćanja stanja tipkovnice i drugih sličnih poslova posebnih svakom operacijskom sustavu. Korišena je kako bi se ubrzao razvoj i prenošenje na različite platforme.

[1] `https://www.libsdl.org/index.php`

2.4. Biblioteka Dear ImGui

Dear ImGui[2] biblioteka korištena je za stvaranje grafičkog korisničkog sučelja. Otvorenog je izvornog koda te korištena u mnogim komercijalnim i nekomercijalnim projektima[3]. Temelji se na "immediate mode" pristupu stvaranja sučelja koji je analogan načinu crtanja u starijim inačicama biblioteke OpenGL[4].

Korištena je kako bi se olakšao razvoj sučelja potrebnog za prikazivanje informacija koje računamo, a koje nisu direktno vidljive iz samih rezultata, odnosno prikaz inače skrivenih podataka programa.

[2] `https://github.com/ocornut/imgui`

[3] `https://github.com/ocornut/imgui/wiki/Software-using-dear-imgui`

[4] prije inačice OpenGL 3

3. Volumni element

Volumni element (engl. voxel) je trodimenzionalna nadogradnja na slikovni element (engl. pixel). Opisivanje prostora volumnim elementima pogodno je zbog često jednostavnijeg rukovanja i mijenjanja istoga. Zbog toga što se razlikuje od ustaljenog pristupa opisivanjem omeđujućim politopima, donosi i različite probleme. Dva značajna problema su opisivanje takvog prostora računalnim primitivima i prikaz takvog prostora.

3.1. Opisivanje prostora računalnim primitivima

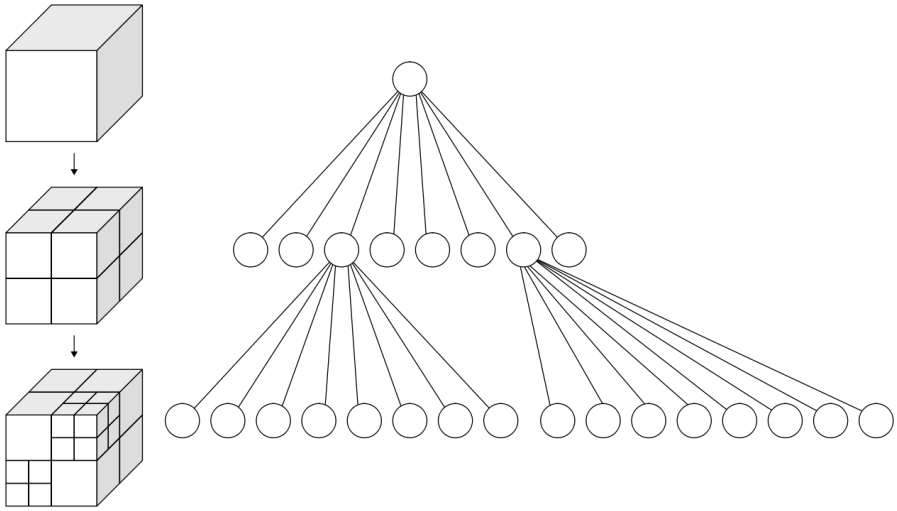
3.1.1. Oktalno stablo

Struktura oktalno stablo (engl. octree) je stablo bez ijednog ili s osam oktalnih stabala stabala djece. U isječku koda 3.1 možemo vidjeti kako jednostavno u programskom jeziku C++ prikazati strukturu oktalnog stabla.

```
struct octree_node
{
    voxel_data   Data;
    octree_node *Children[8];
};
```

Isječak 3.1: Stuktura oktalnog stabla

U računalnoj grafici se često koristi za sadržavanje informacije o volumnim elementima. Interpretira se kao trodimenzionalna struktura koja je binarna u 3 dimenzije. Na slici 3.1 vidimo vizualizaciju strukture oktalnog stabla.



Slika 3.1: Trodimenzionalnost oktalnog stabla [3]

Oktalno stablo omogućuje lako sažimanje podataka za regije koje sadrže jednolike podatke. Ako cijela regija koju trenutni čvor stabla opisuje sadrži istu informaciju, informaciju zapisujemo u čvor. Inače, stvaramo osam čvorova za kao djecu te regije i ponavljamo za svako dijete čvor isti postupak.

Korištenje strukture oktalnog stabla omogućuje proizvoljnu veličinu detalja mijenjanjem maksimalne dubine stabla i omogućuje lako prostorno pretraživanje.

Nezgodna stvar kod stabla je što je brzina pristupa elementu prostora proporcionalna njegovoj dubini u stablu. Kod jednostavnijih i manjih svjetova to ne predstavlja problem, ali stvaranje modela za prikaz tog svijeta to zahtjeva prolazak po cijelom svijetu. Zato što imamo i primitivan model svjetlosti, koji ćemo kasnije pokazati, prolazak bi zahtijevao i obilazak svih susjeda za svaki element svijeta. Algoritam izrade modela iz ovakve reprezentacije svijeta nije jednostavan za izvesti ni povoljan za izračunati.

S druge strane, ovakav svijet bi mogao biti dobar za metodu praćenja zrake. Sudar zrake se tada može provjeravati po ulasku u element stabla, i time bismo imali veće korake zrake za područja svijeta s malom dubinom stabla.

3.1.2. Koordinatna mreža

Koordinatna mreža (engl. grid) je jednostavniji zapis svijeta. U tom pristupu je svijet trodimenzionalno polje elemenata. Pristup pojedinom elementu i dohvaćanje susjeda elementa je konstantne složenosti. U odnosu na oktalno stablo, ovaj pristup ne može prikazivati proizvoljno sitne detalje i nema ugrađen mehanizam za sažimanje podataka za veće jednolike regije. Najjednostavniji način za opisati takav svijet u programskom jeziku C++ dan je u isječku 3.2.

```
struct world {
    voxel_data Data[WORLD_SIZE][WORLD_SIZE][WORLD_SIZE];
};
```

Isječak 3.2: Jednostavna struktura svijeta

Osim toga, problem je i ako pokušamo držati informacije o većim svjetovima. Za svijet veličine $1024 \times 1024 \times 1024$ bismo morali imati u memoriji preko milijardu elemenata.

Taj problem rješavamo tako da svijet dijelimo na regije, a zatim ih učitavamo ili stvaramo po potrebi. Kada prestane potreba za nekom regijom, brišemo ju iz memorije i, ako ima potrebe za tim, zapisujemo na disk.

```
struct chunk {
    ivec3 Position;
    voxel_data Data[CHUNK_SIZE][CHUNK_SIZE][CHUNK_SIZE];
};
struct world {
    hashmap RegionMap; // key = Position, value = Chunk
};
```

Isječak 3.3: Struktura svijeta podijeljenog na regije

U isječku 3.3 se vidi struktura dovoljna za implementaciju te ideje. Svijet je u tom slučaju preslikavanje pozicija regija na same regije, a regije poprimaju sličan oblik kao što je svijet imao u isječku 3.2.

Možemo napraviti i razne hibridne pristupe – na primjer u kojemu su regije prikazane strukturama oktalnog stabla, a regije kao preslikavanja s pozicije na segment oktalnog stabla koji opisuje tu regiju.

4. Prikazivanje prostora

Prikazivanje svijeta se može ostvariti na razne načine. Ovisno o stilu koji želimo postići, možemo prikazivati volumne elemente kockama, a možemo i raditi zaglađivanje između podataka koje imamo.

Prikaz kockama je jednostavniji i češće uočljiv zbog svojeg karakterističnog kockastog izgleda. Na slici 4.1 vidimo kocaksti izgled prostora opisanog volumnim elementima u računalnoj igri Minecraft.

Slika 4.1: Minecraft, kockasti izgled [7]

Mogu se i miješati ta dva stila – primjer toga bi bio u računalnoj igri Roblox.

Slika 4.2: Miješanje glatkog i kockastog pristupa [4]

Na slici 4.2 možemo vidjeti teren koji izgleda "glatko", a u pozadini dio kuće, za koju volumni elementi izgledaju "oštro".

U ovom radu ćemo proći kroz neke metode i pobliže opisati jednu.

4.1. Naivan pristup cijelih kocaka

Najjednostavnije rješenje je crtati cijeli svijet kockama. Potrebno je generirati model kocke za svaki volumni element koji nije prazan, a za prikazati svijet je onda potrebno prikazati svaku od generiranih kocaka.

Količina podataka za prikaz je tada jednaka $N_{kocaka} \times N_{kocka,trokuta} \times N_{trokut,tocaka} \times Velicina_{tocka}$. Ako uzmemo da je za jednu kocku potrebno 6 lica, a za svako lice po 2 trokuta s po 3 točke, te za svaku točku njena pozicija, koordinate teksture i jedan broj za količinu osvjetljenja – odnosno da nam je točka velika 8×4 bajta (4 bajta je uzeto zbog prikaza broja s pomičnom točkom IEEE754) – dolazimo do veličine od 1152 bajta po kocki. U isječku 4.1 je prikazana implementacija takve točke u programskom jeziku C++.

```
struct vertex {
    vec3 Position;
    float LightIntensity;
    struct {
        vec2 Min;
        vec2 Max;
```

```
    } Texture;
};
```

Isječak 4.1: Struktura vrha kocke

Korištenje indeksa za točke je moguće. Nedostatak je što bismo tako izgubili mogućnost osvjetljenja lica neovisno jedno o drugomu ukoliko dijele točku. Osim osvjetljenja gubi se i mogućnost različitih tekstura po licu. Prednost je što bismo uštedjeli značajnu količinu memorije. Uz ponovno korištenje točaka pomoću indeksa bismo imali 256 bajta za podatke o točkama (račun je isti kao i gore, ali uz samo 8 točaka) i 144 bajta (6 lica po 6 točaka – 3 za svaki trokut) za indeksiranje koje bi se moglo i prepoloviti za dovoljno malene modele (gdje indeksi ne prelaze 2^{16}. To daje ukupno 336 bajta po kocki pa je ušteda od 528 bajta po kocki.

Lako se uočava da se model kocke ponavlja uz samo drugačiju poziciju u svijetu. Zbog toga možemo napraviti samo jedan model kocke, a za prikaz koristiti instanciranje. Time se smanjuje količina za sam model – koja tada postaje konstanta i ne ovisi o broju kocaka. Koordinate tekstura se mogu svesti na preddefinirane teksture koje će se samo izabirati pomoću indeksa. Uz pretpostavku da želimo održati sposobnost neovisnog osvjetljenja lica, potrebno je imati 6 (1 po licu) brojeva koji opisuju količinu osvjetljenja.

Količina podataka za prikaz je tada jednaka $N_{kocaka} \times Velicina_{instanca}$, a $Velicina_{instanca}$ je tada jednaka 6×4 bajta (osvjetljenje) $+16$ bajta (indeks teksture) $+3 \times 4$ bajta (pozicija) – što je 52 bajta. U isječku 4.1 je prikazana implementacija opisane instance u programskom jeziku C++.

```
struct instance {
    vec3  Position;          // pozicija instance
    float LightIntensity[6]; // za svako od lica
    struct {
        vec2 Min;
        vec2 Max;
    } Texture;
};
```

Isječak 4.2: Struktura instance kocke

Korištenje indeksa za točke je moguće, ali ne bi se time uštedjelo značajno u slučaju instanciranja. Ušteda bi bila konstantna u odnosu na količinu kocaka.

Metoda	Veličina po kocki	Veličina za 256^3 kocaka
1	1152 B	18 GiB
1 uz indekse	528 B	8.25 GiB
2	52 B	832 MiB
2 uz indekse	52 B	832 MiB

Tablica 4.1: Usporedba veličina za prethodne, naivne, metode

Iz tablice 4.1 vidimo da prethodne metode nisu pogodne za imalo veće svjetove.

4.1.1. Optimizacije prostora

Kod optimiranja količine podataka, dobar prvi korak je tražiti mjesta koja pružaju više slobode nego što se od podataka očekuje. U prelasku na instanciranje smo zapravo napravili tako jednu optimizaciju – prije instanciranja smo mogli imati svaki model drugačiji (za čime nema potrebe u slučaju kocaka), i zapravo uklanjanjem te slobode dobivamo smanjenje količine podataka.

Možemo uočiti je da pozicije kocaka imaju nepotrebnu slobodu. Možemo imati pozicije kocaka koje nisu na cjelobrojnim koordinatama, a za to nema potrebe. Ako to promijenimo na cjelobrojne brojeve tipa int, nećemo ništa uštedjeti jer je jednake veličine kao i $float$. No, ako pretpostavimo da se svijet drži u regijama, možemo sve tri koordinate (ako su regije dovoljno malene) staviti u jedan broj. Na primjer, za regije veličine 32^3 dovoljno nam je $\lceil log_2 32 \rceil$ bitova za podatke o poziciji za jednu koordinatnu os – to jest $3 * \lceil log_2 32 \rceil$, odnosno 15 bitova za sve tri. Tada umjesto da poziciju držimo pomoću 12 bajta, držimo pomoću samo 2 bajta. Veličina jedne instance tada je 42 bajta. U isječku 4.3 je dana implementacija takve instance s cjelobrojnim koordinatama.

```
struct instance {
    struct {
        u16 X : 5;
        u16 Y : 5;
        u16 Z : 5;
    } Position;
    float LightIntensity[6];
    struct {
        vec2 Min;
        vec2 Max;
```

```
    } Texture;
};
```

Isječak 4.3: Struktura instance kocke s cjelobrojnim koordinatama

Slično, možemo i teksturu izabirati ne pomoću koordinata, nego imati mapiranje tekstura indeks $\rightarrow$ tekstura. Uz pretpostavku da imamo do 2^{16} različitih tekstura, možemo količinu podataka za odabir teksture smanjiti s 16 bajta na 2. Veličina jedne instance tada je 28 bajta. U isječku 4.4 vidimo strukturu instance kocke nakon promjene načina opisivanja teksture.

```
struct instance {
    struct {
        u16 X : 5;
        u16 Y : 5;
        u16 Z : 5;
    } Position;
    float LightIntensity[6];
    u16 Texture;
};
```

Isječak 4.4: Struktura instance kocke uz indeksiranje teksture

Dominantni dio podataka je sada u količini osvjetljenja. Imati manje distinktnih razina osvjetljenja je nešto što možemo odabrati. Ako odaberemo 16 razina, možemo ih reprezentirati pomoću 4 bita. Time imamo samo 3 bajta umjesto 24 za količinu osvjetljenja za sva lica. Veličina instance je 6 bajta. U isječku 4.5 promijenjen je način opisivanja količine osvjetljenja.

```
struct instance {
    struct {
        u16 X : 5;
        u16 Y : 5;
        u16 Z : 5;
    } Position;
    struct {
        u8 _0:4, _1:4, _2:4, _3:4, _4:4, _5:4;
    } LightIntensity;
    u16 Texture;
```

```
};
```

Isječak 4.5: Struktura instance kocke uz smanjenu količinu razina osvjetljenja lica

Metoda	Veličina po kocki	Veličina za 256^3 kocaka
cjelobrojne koordinate	42 B	672 MiB
indeks teskture	28 B	448 MiB
16 razina osvjetljenja	6 B	96 MiB

Tablica 4.2: Optimizacije naivnog pristupa kockama

U tablici 4.3 vidimo da su svjetovi te veličine prostorno izvedivi uz optimizacije.

4.1.2. Optimizacije brzine

U prijašnje opisanim koracima smo koristili potpune kocke. Potpune kocke nisu idealne jer sadrže lica koja nisu vidljiva. Ukoliko se nalaze dvije kocke jedna do druge, njihova lica kojima se diraju neće biti vidljiva i zato ih nema potrebe crtati.

Instance kocaka u ovom slučaju ne mogu više pomoći jer zapravo, iako su naizgled modeli nam kocke – modeli bi nam trebali biti samo vidljiva lica. Za postići takvo crtanje trebamo pažnju prebaciti na sama lica kocaka. Lice možemo reprezentirati jednom instancom. Za lice na jedinstven način zapisati unutar jedne regije nam treba pozicija kocke i orijentacija. Za poziciju smo već vidjeli da nam je dovoljno 15 bitova – po 5 sa svaku os. Za orijentaciju lica nam je dovoljno 3 bita – pozitivan i negativan smjer na svakoj osi. Za osvjetljenje nam je sada dovoljno samo jednu vrijednost držati jer za razliku od maloprije instance su točno ono za što želimo. Preostaje nam 10 slobodnih bitova po licu koje možemo iskoristiti za druge stvari.

Time smo sveli jednu instancu na 6 bajta. Implementaciju te strukture možemo vidjeti u isječku 4.6.

```
struct instance {
    u32 PositionX : 5;
    u32 PositionY : 5;
    u32 PositionZ : 5;
    u32 Direction : 3; // +X, -X, +Y, -Y, +Z, -Z
    u32 LightIntensity : 4;
    u16 Texture;
```

```
};
```

Isječak 4.6: Struktura instance lica

Ukoliko je dovoljno 2^{10} različitih tekstura, možemo izbaciti još i dva bajta i dobili bismo 4-bajtnu instancu. Takvu strukturu možemo vidjeti u isječku 4.7.

```
struct instance {
    u32 PositionX : 5;
    u32 PositionY : 5;
    u32 PositionZ : 5;
    u32 Direction : 3; // +X, -X, +Y, -Y, +Z, -Z
    u32 LightIntensity : 4;
    u32 Texture   : 10;
};
```

Isječak 4.7: Struktura instance lica

Za prikazati jednu kocku je potrebno 6 instanci. Samo po sebi ovo je korak unazad, no ovo nije direktno usporedivo s prošlim pristupom jer nećemo prikazivati cijele kocke. Iz strukture koja opisuje volumne elemente ćemo stvarati jedan model po regiji, a model će biti napravljen samo od vidljivih lica kocaka. Najgori slučaj je da se vidi svaka kocka, a to možemo konstruirati tako da naizmjenično popunimo volumne elemente. Broj volumnih elemenata za taj slučaj je pola od skroz popunjene regije.

Najgori slučaj je u praksi dosta nezanimljiv. Za "stvarne" prostore je dosta velika količina nevidljivih lica po volumnom elementu. Ova optimizacija, iz očitih razloga, dosta i smanjuje broj piksela koji se nepotrebno crtaju. Iako se ova optimizacija ne očituje direktno toliko memorijom, no ubrzanje crtanja je značajno.

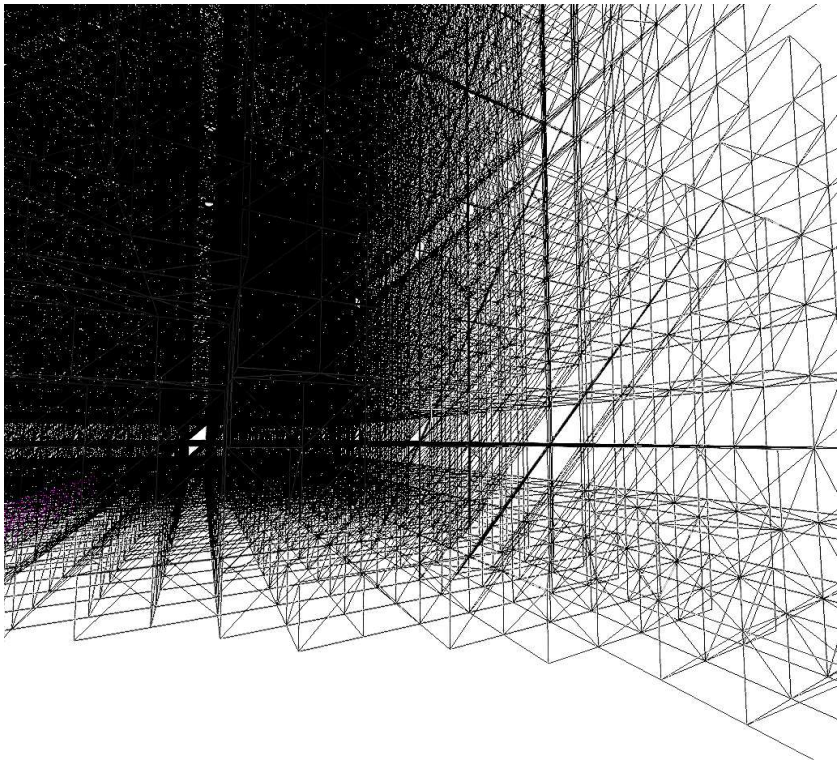
Najbolji slučaj bi bilo da svaka kocka ima sve susjede prisutne. Takav svijet isto nije baš zanimljiv.

Najbolji slučaj za 256^3 kocaka	Najgori slučaj za 256^3 kocaka
393.216 lica	$50.331.648$ lica
2.25 MiB	288 MiB

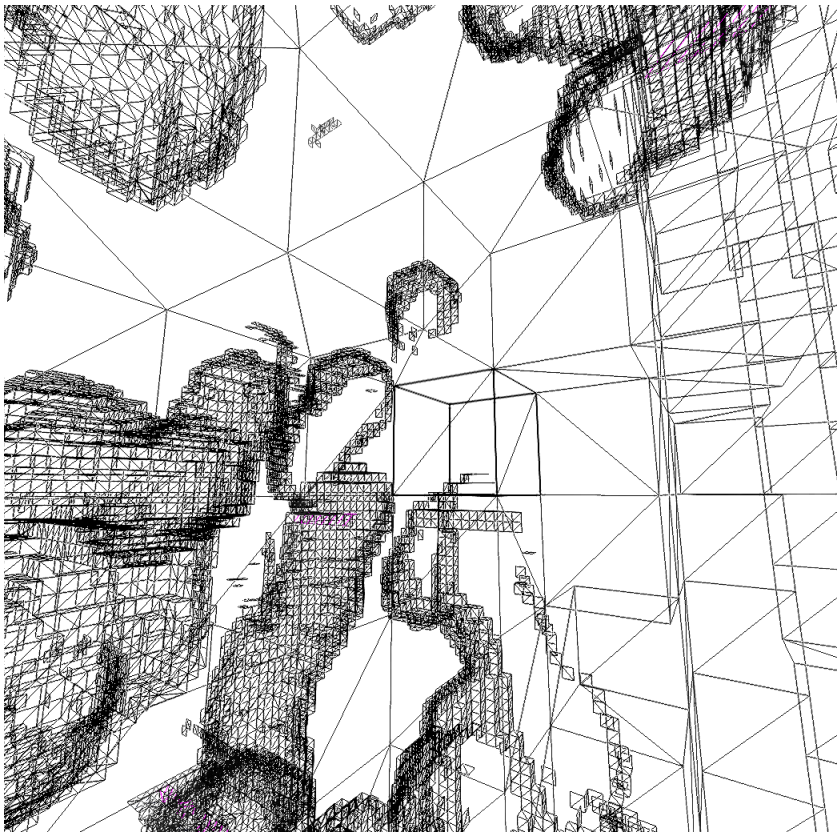
Tablica 4.3: Optimizacije naivnog pristupa kockama

U tablici 4.3 možemo vidjeti za najbolji i najgori slučaj količinu memorije koju bi modeli zauzimali i koliko bi lica prikazali.

Na slici 4.3 se vidi dosta lica koje se nepotrebno crtaju, a na slici 4.4 vidimo kako to izgleda nakon skrivanja lica koja nisu vidljiva.



Slika 4.3: Prikaz bridova – iz praktičnog dijela rada



Slika 4.4: Prikaz bridova bez prikazivanja nevidljivih lica – iz praktičnog dijela rada

4.1.3. Još optimizacija brzine

Zbog toga što je svijet podijeljen u regije, možemo regije za koje znamo da se ne mogu vidjeti u cijelosti iz perspektive kamere uopće ne prikazivati. Sve što je izvan vidnog polja se može ne prikazivati.

Vulkan pruža mogućnost crtanja bez izravnog pozivanja naredbi za crtanje. Zamišljeno je da smanje nepotrebne sinkronizacije između grafičkog sklopovlja i procesora. Umjesto da se poziva više naredbi za crtanje, možemo napraviti jedan spremnik (engl. buffer) naredbi s njihovim argumentima i onda pozvati naredbu za neizravno crtanje koja će direktnije sklopovlju ostvariti željene pozive. Taj spremnik naredbi možemo čak i na grafičkom sklopovlju direktno stvoriti. To nam omogućuje da prikazujemo svijet pomoću konstantnog broja poziva. Taj poziv onda iz pozicija regija i pozicije i orijentacije kamere stvara potrebne pozive za vidljive regije, a zatim izvršavamo indirektno prikazivanje nad tim stvorenim pozivima.

Algoritam 1: CRTANJE SVIJETA S GLEDIŠTA CPU

1 $L \leftarrow GenerirajPoziveZaCrtanjeVidljivihModela(ModeliRegija)$;
2 $PosaljiListuPozivaNaGPU(L)$;
3 $IzvrsiUcitanePoziveNaGPU()$;

4.1.4. Izgradnja modela

Prelaskom na model sa samo vidljivim licima gubimo jednostavnost izgradnje modela. Više nije moguće direktno prevesti zapis volumnih elemenata regije u model regije.

Potrebno je za svaku kocku, za svako lice pogledati postoji li neprozirna kocka u smjeru tog lica. Ako ne postoji, treba dodati lice. Generiranje modela nije trivijalan korak i zahtjeva dosta vremena.

Možemo znatno ubrzati postupak izgradnje tako da napravimo strukturu uz onu u kojoj držimo podatke o volumnim elementima, ali umjesto o popunjenosti i vrsti elementa držimo podatak o tome koje su susjedne neprozirne kocke prisutne. Time možemo značajno smanjiti broj upita za volumne elemente jer možemo, umjesto da za svaki volumni element gledamo na njegovoj poziciji i na pozicijama susjeda što je prisutno kako bismo odredili koja lica i kako su vidljiva, gledati samo one susjede za lica za koje znamo da su vidljivi.

4.1.5. Svjetlost

Dosta jednostavan model svjetlosti za uvesti u takav svijet i izgradnju modela je svjetlost intenziteta proporcionalnog udaljenosti od izvora. Za udaljenost uzimamo zbroj udaljenosti po osima, gdje je udaljenost po osi apsolutna vrijednost razlike pozicija, odnosno L^1 norm.

$$I = I_0 - \sum_{i=1}^{n} |p_i^{(1)} - p_i^{(2)}| \tag{4.1}$$

Osim udaljenosti treba uzeti u obzir i da predmeti na putu do izvora svjetlosti blokiraju zrake.

Izgradnja modela s takvom svjetlošću je konceptualno jednostavna, ali bez dodatnih optimizacija će biti ju problem u stvarnom vremenu izvesti. Promjena u procesu izgradnje modela je to što za svaku vidljivo lice trebamo odrediti koliko je osvijetljeno – dakle, za svako vidljivo lice moramo pogledati za sve izvore svjetlosti koliko su udaljeni i odrediti koliko bi svjetlosti ukupno to bilo. Da ne postoji uvjet da je svjetlost blokirana neprozirnim elementima, odrediti količine svjetlosti bi bilo moguće u stvarnom vremenu, za mali broj izvora. Doduše, ni u tom slučaju broj koraka nije malen – jednak je broju vidljivih lica pomnoženim s brojem svjetala. Za slučaj da gledamo i zaklonjenost neprozirnim objektima, morali bismo i tražiti put između izvora i lica – što ovisi o udaljenosti i relativno je skupa operacija.

Kako bismo sve to izbjegli, možemo količinu svjetlosti računati jednom prilikom postavljanja izvora ili promjeni nekog neprozirnog elementa prostora. Za to moramo svesti informacije na nešto jednostavnije i manje realistično, a to je da svakom elementu prostora označimo kolika je količina svjetlosti "u njemu". Ako uzmemo 4 bita za sadržavanje informacija o količini svjetlosti na nekom elementu, imali bismo 2^4 razina svjetlosti. Maksimalan broj elemenata kojima trebamo označiti količinu svjetlosti je tada volumen u L^1 prostoru koji za intenzitet R iznosi $((R*(R*((R*2)-3)+4)*1)/3-1)$ za $R > 0$, odnosno 4991 za $R = 16$. Izraz je dobiven brojanjem volumnih elemenata koje svjetlost može posjetiti. Možemo po nekoj od osi podijeliti na plohe. Za plohu u kojoj se nalazi izvor svjetlosti ćemo imati slučaj isti problem kao i s kojim smo počeli, no ovaj put u dimenziji manje. Dvodimenzionalni slučaj se lako rješava tako da se podijeli na 4 identična dijela zakrenuta oko središta uz jedan element za samo središte. Time dobivamo $1+4\times(R+1)R/2$. Za plohe iznad i ispod plohe u kojoj se nalazi izvor će radijus biti manji točno za onoliko koliko su udaljene od plohe u kojoj je izvor zbog načina na koji se intenzitet svjetlosti smanjuje. Zbrajanjem broja volumnih elemenata u svim plohama koje čine taj prostor dobivamo formulu koji tražimo.

Da bismo proširili svjetlost krećemo s dodavanjem volumnog elementa izvora

svjetlosti, uz intenzitet izvora svjetlosti, u listu koju treba obraditi. Propagiramo svjetlost dok ne ispraznimo listu za obradu.

Na svakom koraku propagacije postavljamo intenzitet trenutnog elementa na onaj koji smo dobili, a zatim sve susjede koji imaju intenzitet za dva ili više manji od onoga na trenutnom elementu, uz intenzitet koji će imati zbog nas (odnosno trenutni intenzitet umanjen za 1), dodajemo u listu elemenata koje još treba posjetiti i obraditi na isti način.

Algoritam 2: ŠIRENJE SVJETLOSTI

Data: P_0 – voxel position, I_0 – source light intensity

$L \leftarrow \{(@P \leftarrow P_0, @I \leftarrow I_0)\}$;
for $i \leftarrow L$ **do**
  if $I_i > LightIntensity_{P_i}$ **then**
    $LightIntensity_{P_i} \leftarrow I_i$;
    for $n \leftarrow VisibleNeighbors(P_i)$ **do**
      if $LightIntensity_n + 1 < I_i$ **then**
        $L \leftarrow L \cup \{(@P \leftarrow n, @I \leftarrow I_i - 1)\}$;
      end
    end
  end
end

U stvarnoj implementaciji algoritam je poboljšan tako što prije traženja susjeda gledamo je li intenzitet takav da ga ima smisla širiti, odnosno da je veći od 1. Osim toga je i nepotrebno provjeravati intenzitet trenutnog koraka veći od već postavljenog elementa ako obrađujemo elemente u širinu, a uz to još pratimo i posjećenost elemenata tako da nema višestrukog obrađivanja.

Intenzitet je potrebno pamtiti za svaki volumni element, i njega se onda koristi prilikom izgradnje za vidjeti koliko je lice osvijetljeno. Za to je potrebno izmijeniti strukturu regije. U isječku 4.9 vidimo strukturu regije svijeta u kojoj je dodan prostor predviđen za podatke o intenzitetu svjetlosti.

```
struct chunk {
    ivec3 Position;
    metadata   Meta[CHUNK_SIZE][CHUNK_SIZE][CHUNK_SIZE];
    voxel_data Data[CHUNK_SIZE][CHUNK_SIZE][CHUNK_SIZE];
};
```

Isječak 4.8: Struktura regije svijeta uz prostor predviđen za metapodatke

$Meta$ se koristi za spremanje podataka o prostoru koji se mogu izračunati iz $Data$, a ideja je ubrzati skupe izračune pamteći njihove rezultate. Dodavanje takve strukture je opravdano svojim doprinosom ubrzanja raznih operacija nad svijetom.

Uz intenzitet svjetlosti možemo i pamtiti koja lica tog volumnog elementa su vidljiva – po 1 bit za svako lice, odnosno 6 bitova ukupno za trodimenzionalnu kocku. Implementaciju strukture za praćenje vidljivosti lica i intenzitet svjetlosti možemo vidjeti u isječku 4.9.

```
struct metadata {
    u16 LightIntensity : 4;
    u16 X  : 1;
    u16 X_ : 1;
    u16 Y  : 1;
    u16 Y_ : 1;
    u16 Z  : 1;
    u16 Z_ : 1;
};
```

Isječak 4.9: Struktura metapodataka za jedan volumni element

Traženje vidljivih susjeda se u tom slučaju svodi na gledanje koji bit je postavljen na samo trenutnom elementu – time nemamo dodatne upite nad memorijom gdje provjeravamo jesu li susjedi prisutni.

Uništavanje izvora svjetlosti je skuplja operacija od samog širenja. Potrebno je na sličan način kao kod algoritma širenja svjetlosti proširiti se, ali ovaj put na sve one elemente koji su potencijalno osvijetljeni od izvora kojeg uništavamo i za sve te elemente koji su time obuhvaćeni postaviti intenzitet na 0. Nakon toga je potrebno uzeti vanjsku ljusku tog područja, i za sve one elemente kojima je intenzitet takav da ima smisla ga širiti, obraditi kao i kod prvog algoritma širenja svjetlosti. Ovo je očito skuplja operacija od samog širenja svjetlosti jer, osim sličnog koraka kao cijeli algoritam širenja svjetlosti, poziva algoritam širenja svjetlosti za svaki element ljuske područja koji je obrađen.

Algoritam 3: UNIŠTAVANJE SVJETLOSTI

Data: P_0 – voxel position

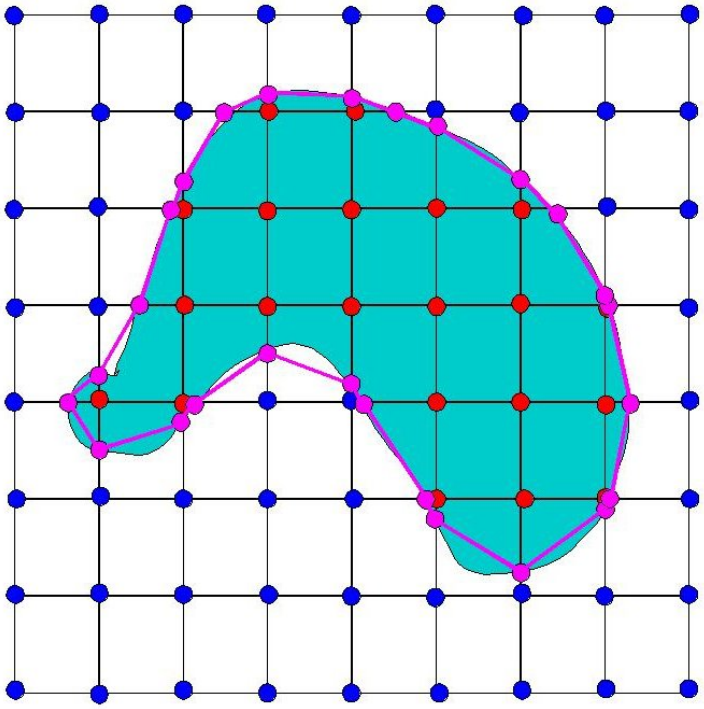
$B \leftarrow \emptyset$;
$L \leftarrow \{(@P \leftarrow P_0, @I \leftarrow LightIntensity_P)\}$;
for $i \leftarrow L$ **do**
    if $I_i = 0$ **then**
        $B \leftarrow B \cup \{P_i\}$;
    end
    else
        if $LightIntensity_{P_i} = I_i$ **then**
            $LightIntensity_{P_i} \leftarrow 0$;
            for $n \leftarrow VisibleNeighbors(P_i)$ **do**
                if $LightIntensity_n + 1 < I_i$ **then**
                    $L \leftarrow L \cup \{(@P \leftarrow n, @I \leftarrow I_i - 1)\}$;
                end
            end
        end
    end
end
for $i \leftarrow B$ **do**
    $SpreadLight(@P_0 \leftarrow i, @I_0 \leftarrow LightIntensity_i)$;
end

Za postavljanje ili uništavanje elementa je potrebno raditi ove izračune, odnosno maknuti svjetlost iz samog elementa ako je neproziran (time zapravo uništiti izvor koji je na poziciji na koju postavljamo element) odnosno proširiti svjetlost iz susjednih elemenata nakon uništavanja elementa.

4.2. Ostali pristupi

4.2.1. Algoritam pokretne kocke

Algoritam pokretne kocke pokušava pobliže aproksimirati originalne modele gdje koristi više informacija, a to mjesta sjecišta modela i koordinatne mreže. Za razliku od samo kocaka, ovdje plohe kojima se aproksimira model nisu poravnate s koordinatnim osima.



Slika 4.5: Aproksimacija originalnog modela pomoću sjecišta s koordinatnom mrežom [1]

Na slici 4.5 se ilustrira model dobiven ovim pristupom.

Ovaj algoritam nije fokus rada pa nećemo više od ovoga ulaziti u detalje implementacije. Više o samom algoritmu možete pronaći u radu [2].

4.2.2. Metoda praćenja zrake

Metoda praćenja zrake se izvodi tako da iz virtualne kamere pratimo putanju zraka kakve bi svjetlost i u stvarnosti imala. Najjednostavniji način za to izvesti je iz kamere projecirati zraku kroz zaslon na svijet kojeg pokušavamo prikazati, a to je potrebno napraviti za svaki element slike koju stvaramo. Zbog toga što količina zraka koje pratimo ovisi o veličini slike koju generiramo, ova metoda može biti pogodna za ogromne svjetove za koje želimo na proceduralan način generirati. Osim toga, moguće je lako dodati neka realistična ponašanja svjetlosti – poput refleksije i loma.

Slika 4.6: [6]

Postoje razne računalne igre s implementacijom metode praćenja zrake, ali trenutno jedna od najpotpunijih koristivih implementacija je od strane Nvidia u računalnoj igri Minecraft koju možemo vidjeti na slici 4.6.

U radu se nećemo detaljnije baviti ovom metodom, ali pročitati više o istoj možete u radu [5].

5. Zaključak

Prikazivati prostor je moguće izvesti na razne načine. Svaki pristup ima svoje pozitivne strane. Jedan od načina prikaza prostora je pomoću volumnih elemenata (engl. voxel). Ta metoda nije raširena koliko i prikaz modela pomoću trokuta. Ideja rada je bila razmotriti prostor opisan volumnim elementima, opisati prikaz istoga i vidjeti implementaciju na sklopovlju.

Pobliže je opisana implementacija kockastog prikaza prostora opisanog volumnim elemntima. Opisane su i implementirane optimizacije kako bi izvedba na sklopovlju bila interaktivna. Opisani su doprinosi optimizacija. Optimizacije uključuju smanjenje memorijskog prostora potrebnog za opisati model regija, smanjenje broja lica koja prikazujemo i ubrzanje izgradnje modela. Opisan je jednostavan model svjetlosti i ubrzanje računanja istog. Objavljeni su i javno dostupni opisani algoritmi u programskom jeziku C++.

Dobiveni rezultati rada bi se mogli još unaprijediti. Implementacija bi se mogla još ubrzati tako da spajamo susjedna lica koja dijele stranicu u ravnini u jedno veće lice i time smanjiti broj lica koje je potrebno prikazati. Osim toga, mogli bismo u stvarnom vremenu odrediti za svaku regiju koji pristup najviše donosi i njega za tu regiju koristiti. Mogao bi se više istražiti i implementirati svijet ostvaren oktalnim stablom, napraviti implementacija simulacije fluida u prostoru opisanom volumnim elementima, istražiti metoda praćenja zrake i metoda pokretne kocke. Mogao bi se istražiti i zakrivljeni prostor opisan volumnim elementima.

LITERATURA

[1] Marching cubes. URL https://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html.

[2] Danijel Bajlo. Prikaz terena algoritmom pokretne kocke, 2020.

[3] Wikimedia Commons. Octree. URL https://upload.wikimedia.org/wikipedia/commons/2/20/Octree2.svg.

[4] Arseny Kapoulkine. Roblox rendering. URL https://zeux.io/2017/12/30/voxel-terrain-physics/.

[5] Ivan Karlović. Interaktivan prikaz vokseliziranog prostora s vulkanom uz sklopovski ubrzano praćenje zrake, 2020.

[6] Nvidia. Minecraft rtx. URL https://www.nvidia.com/en-us/geforce/news/minecraft-rtx-dlss-official-release/.

[7] zzbking. Minecraft voxels. URL https://pixabay.com/illustrations/game-my-world-landscape-digital-1721096/.

Ostvarivanje prikaza temeljenog na prostoru volumnih elemenata

Sažetak

Postoje razni načini za prikazati trodimenzionalni prostor. Jedan od načina je pomoću volumnih elemenata (engl. voxel). Volumni elementi nisu rašireni koliko prikaz trokutima, ali su zanimljivi jer pružaju različite mogućnosti za optimizacije prikaza. U ovo radu su uspoređeni različiti načini prikaza svijeta ostvarenog volumnim elementima i opisano je nekoliko optimizacija. Napravljena je implementacija opisane metode i razne optimizacije kako bi se olakšala izvedba sklopovljem.

Ključne riječi: volumni element, optimiranje, voxel, stvaranje modela, prikaz svijeta, 3D

Voxel meshing

Abstract

There are many ways to represent a 3-dimensional world. One of the ways to represent it is using voxels. Unlike triangles, voxels are not as used in world representation. Regardless of that, they are interesting because of vastly different optimisations they allow us to make. The goal of this thesis was to compare different methods of representing world and describe a few optimisations that can be made to make it faster to run or hardware. Described algorithms and optimisations were implemented.

Keywords: voxel, optimisations, world meshing, world rendering, 3D