Aplikacija za iscrtavanje voxel prostora

Benčević, Bruno

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

APLIKACIJA ZA ISCRTAVANJE VOXEL PROSTORA

Diplomski rad

Bruno Benčević

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 21.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Bruno Benčević
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1185R, 07.10.2021.
OIB studenta:	89791554637
Mentor:	izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Tomislav Keser
Član Povjerenstva 1:	izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 2:	doc. dr. sc. Tomislav Galba
Naslov diplomskog rada:	Aplikacija za iscrtavanje voxel prostora
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	[Rezervirano: Bruno Benčević] Potrebno je napraviti aplikaciju za iscrtavanje voxel-prostora. Prostor treba generirati kao grid/voxel prostor u obliku zamišljenog terena, primjerice pomoću 3D Perlin Nose tehnike. Pri iscrtavanju koristiti različite tehnike optimizacija kod iscrtavanja objekata te usporediti te tehnike.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	21.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 10.10.2023.

Ime i prezime studenta:	Bruno Benčević
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1185R, 07.10.2021.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Aplikacija za iscrtavanje voxel prostora**

izrađen pod vodstvom mentora izv. prof. dr. sc. Alfonzo Baumgartner

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Razvojem računala kroz povijest pojavljivala se sve veća zainteresiranost za video igre i simulacije virtualnog svijeta koji nalikuje stvarnom. Ispočetka su video igre bile dvodimenzionalne, a kako je snaga računala i grafičkih kartica rasla, počele su se pojavljivati i trodimenzionalne video igrice koje simuliraju 3D prostor. Najčešći način implementacije prostora je bio učitavanje prethodno definiranih modela te iscrtavanje istih na zaslon- Takvi modeli su mali te konačnih dimenzija. Daljnjim napretkom tehnologije počele su se pojavljivati igre s proceduralno generiranim prostorima koji bi se generirali tijekom igranja. Jedan od najčešćih pristupa proceduralno generiranim terenima je implementacija kroz Voxel prostor u kojem se virtualni prostor dijeli u manje komade odnosno kocke čije se veličina prilagođava potrebama igre – od sitnih kocaka koje omogućuju detaljniji i realističniji prostor do velikih kocaka koje su idealne za reprezentaciju nasumičnog reljefa. U ovom diplomskom radu opisana je implementacija aplikacije za proceduralno generiranje prostora te metode iscrtavanja Voxel prostora. Prvi dio rada daje kratak pregled područja te opisuje načine implementacije aplikacije za iscrtavanje Voxel prostora iza čega slijedi opis korištenih tehnologija u ovom radu. Treće poglavlje rada opisuje arhitekturu aplikacije te razloge i načine razdjeljivanja poslova na više niti. Nakon toga se opisuje način generiranja Voxel prostora koristeći generatore šumova te metode ažuriranja prostora. U četvrtom dijelu ovog rada je objašnjen način iscrtavanja Voxel prostora te se uspoređuju tehnike optimizacije modela prostora i načina iscrtavanja prostora.

1.1. Zadatak završnog rada

Zadatak ovog diplomskog rada je napraviti aplikaciju za iscrtavanje Voxel-prostora. Prostor je potrebno razdijeliti u manje dijelovi te je potrebno generirati zamišljeni teren koristeći generatore dvodimenzionalnih i trodimenzionalnih šumova poput Perlin i Simplex šumova. Prilikom iscrtavanja Voxel-prostora, potrebno je implementirati različite tehnike optimizacije iscrtavanja modela na zaslon te usporediti performanse iscrtavanja Voxel-prostora primjenjivanjem svake od tehnika.

2. PREGLED PODRUČJA

Područje istraživanja načina generiranja i iscrtavanja Voxel prostora je popularizirala računalna igra Minecraft. Pokušavajući rekreirati stvarni svijet, programeri su na kreativne načine rješavali probleme implementacije prikazivanja Voxel prostora vezanih za korištenje memorije i iscrtavanje prostora. Služeći se raznim veličinama voxela unutar prostora, moguće je dobiti realističnije terene koji nalikuju na stvarni svijet. U nastavku ovog poglavlja su opisana rješenja u nekim od najpopularniji video igara zasnovanih na ideji Voxel prostora.

2.1. Minecraft

Minecraft je računalna video igra iz 2011. godine koja omogućuje igračima kretanje Voxel prostorom te uređivanjem istog po želji igrača. Zbog mogućnosti uređivanja prostora, Minecraft omogućuje brojne mogućnosti igračima za izražavanje kreativnosti gradeći razne građevine ili čak rekreiranje 8-bitnog računala unutar samo igre. Glavna karakteristika implementacije Voxel-prostora u Minecraftu je beskrajno proceduralno generirani teren kojeg igrač može istraživati. Implementacija beskrajnog svijeta je omogućena na način da se dijelovi svijeta koji su učitani u radnu memoriju, a daleko su od igrača, pohranjuju na disk te ako im se igrač približi, učitavaju se opet u radnu memoriju što zahtjeva manje radne memorije. Generiranje terena u Minecraftu je implementirano koristeći nekoliko raznih vrsta trodimenzionalnih šumova koji služe za odabir položaja bioma, špilja, vegetacije i ostalih struktura koje se pojavljuju unutar prostora. [1] Teren Minecraft je podijeljen u komade (engl. chunks), a model svakog komada je nadalje podijeljen u više pod-modela. Prilikom iscrtavanja modela provjerava se njihov položaj u odnosu na virtualnu kameru u svijetu te se model neće iscrtavati ukoliko nije vidljiv kameri.

Slika 2.1. *Prikaz generiranog Voxel prostora unutra računalne igre Minecraft*

2.2. Teardown

Teardown je računalna igra iz 2022. godine koja omogućava igračima istraživanje, uređivanje i uništavanje realističnog prostora kojeg čine sitni voxeli. Za razliku od Minecraft-a, Teardown simulira realistično ponašanje i reagiranje okoliša na razaranja koje igrači mogu učiniti. Svaki Voxel unutar okoline Teardown-a reagira na umjetno simuliranu gravitaciju i vanjske sile, te se voxeli međusobno mogu odbijati i sudarati. Za iscrtavanje voxel prostora implementirana je metoda Ray Tracing-a, gdje se umjesto iscrtavanja svakog voxela individualno zapravo iscrtavaju samo pikseli zaslona simuliranjem realističnog reflektiranja svjetlosti što omogućuje skoro konstantno vrijeme iscrtavanja neovisno o broju voksela na zaslonu ili unutar virtualne okoline te omogućava lakšu i realističniju implementaciju virtualnih izvora svjetlosti i sijena. [2]

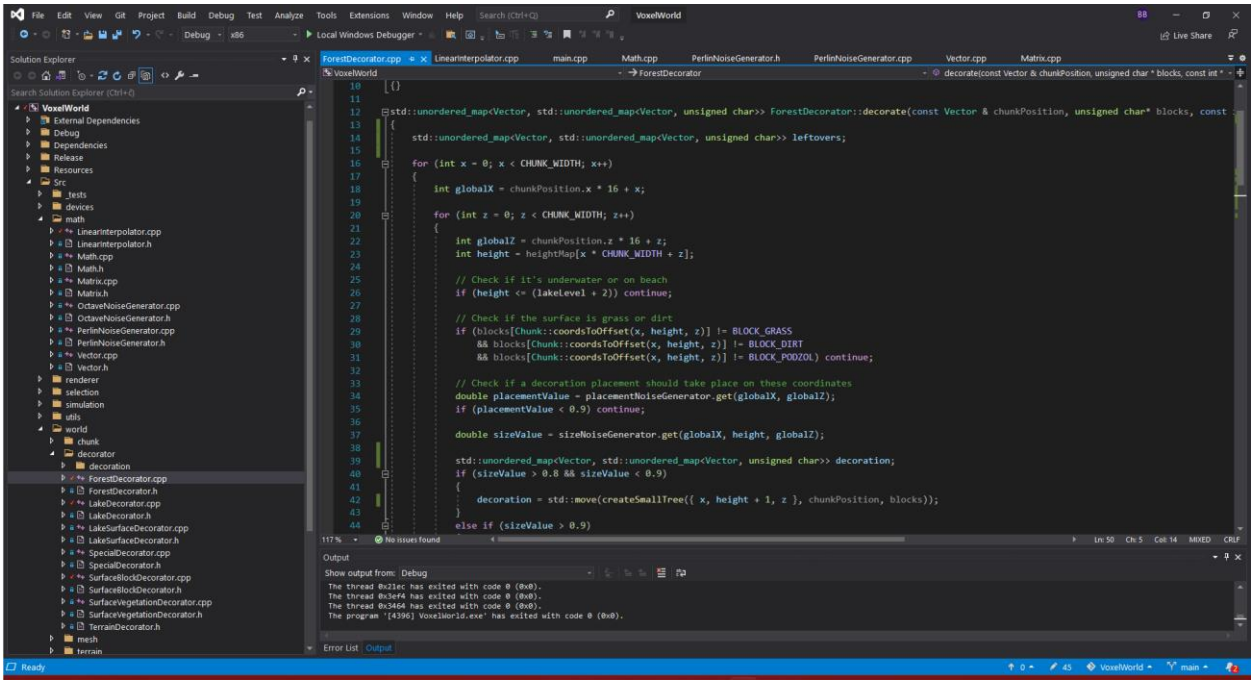
Slika 2.2. *Snimka zaslona računalne igre Teardown.*

3. KORIŠTENI ALATI I TEHNOLOGIJE

Aplikacija za ovaj diplomski rad je napisana u C++ programskom jeziku te se oslanja na OpenGL API za komunikaciju s grafičkom karticom. Za implementaciju osnovnih značajki aplikacije korištene su FastNoise i stb_image biblioteke. Aplikacija je rađena u Microsoft Visual Studio razvojnom okruženju, a sve slike korištene unutar aplikacije su izrađene koristeći Paint.NET program.

3.1. Microsoft Visual Studio, MSVC i C++

Microsoft Visual Studio je integrirano razvojno okruženje koje podržava razvoj projekata koristeći mnoge programske jezike i razvojne okvire. Osim mogućnosti pisanja i uređivanja koda, sadrži alate za pronalaženje grešaka (engl. debugging tools) poput provjeravanja vrijednosti lokalnih varijabli određene funkcije, vizualizacije memorije te osiguravanja pravilnog rukovanja iznimkama unutar koda. Za potrebe ovoga rada korišten je Microsoftov prevoditelj (eng. compiler) MSVC koji podržava C++ 2020 standard.



Slika 3.1. *Izgled integriranog razvojnog okruženja Microsoft Visual Studio.*

3.2. OpenGL, GLSL i GLFW

OpenGL je aplikacijsko programsko sučelje (engl. API) za komuniciranje s grafičkom karticom. Omogućuje kreiranje međuspremnika na grafičkoj kartici, prevođenje programa za sjenčanje te pozivanje raznih funkcija za iscrtavanje modela spremljenih u međuspremnike na zaslon. Kod aplikacija koje koriste više niti, korištenje OpenGL-a podrazumijeva pozivanje funkcija sučelja s niti koja je zatražila OpenGL kontekst.

Za pisanje programa za sjenčanje koristi se GLSL jezik (engl. OpenGL Shading Language). GLSL omogućuje prilagođavanje postupaka sjenčanja po zahtjevima na način da se uređuju koraci sjenčanja individualnih verteksa ili fragmenata, odnosno omogućuje pisanje prilagođenih programa sjenčanja verteksa i fragmenata (engl. Vertex and Fragment Shaders) što olakšava izradu animacija ili primjenjivanje efekata i filtera na konačni prikaz zaslona. [3]

GLFW (engl. Graphics Library Framework) je biblioteka javno dostupnog koda koja pruža pojednostavljeno sučelje za kreiranje prozora i konteksta OpenGL-a koji će služiti za iscrtavanje na prostor kreiranog prozora. GLFW također pruža sučelje za upravljanje korisničkim unosom preko tipkovnice, miša i joysticka te podršku za implementaciju odgovora na događaje vezane za trenutni prozor aplikacije.

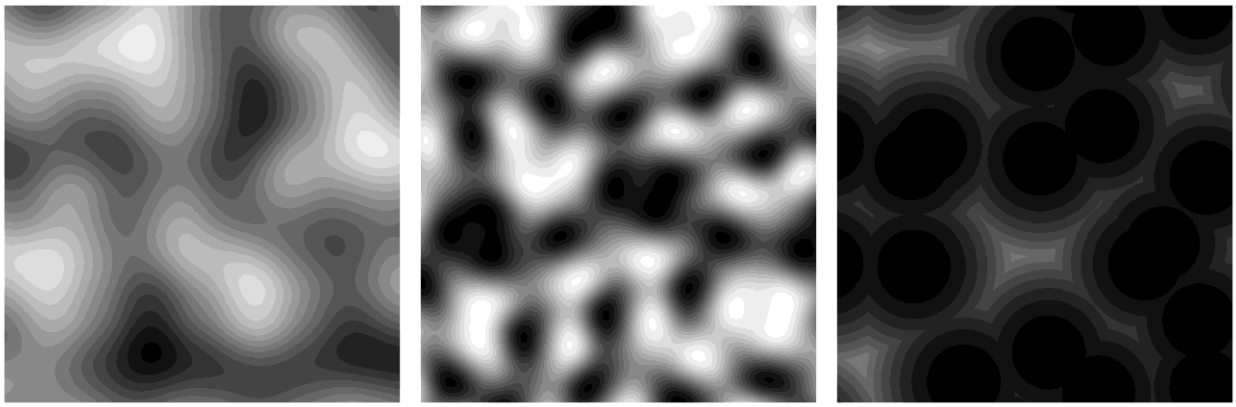
```
1   #version 330 core
2   layout (location = 0) in vec3 positionIn;
3   layout (location = 1) in vec2 uvIn;
4
5   uniform mat4 model;
6   uniform mat4 projection;
7
8   out vec2 uv;
9
10  void main()
11  {
12      gl_Position = projection * model * vec4(positionIn, 1.0);
13
14      uv = uvIn;
15  }
```

Slika 3.2. *Primjer jednostavnog programa za sjenčanje verteksa (engl. Vertex Shader) napisanog u GLSL jeziku za sjenčanje*

3.3. FastNoise i stb_image biblioteke

FastNoise je biblioteka za generiranje različitih vrsta šumova napisana za C++ programski jezik. Omogućava programerima generiranje Perlin, Simplex, kubičnog, gradijentnog, fraktalnog i ćelijskog šuma koristeći jednostavno sučelje. FastNoise je otvorenog koda te je dizajniran za upotrebu u sustavima realnog vremena u kojima generiranje šuma mora biti dovoljno brzo.

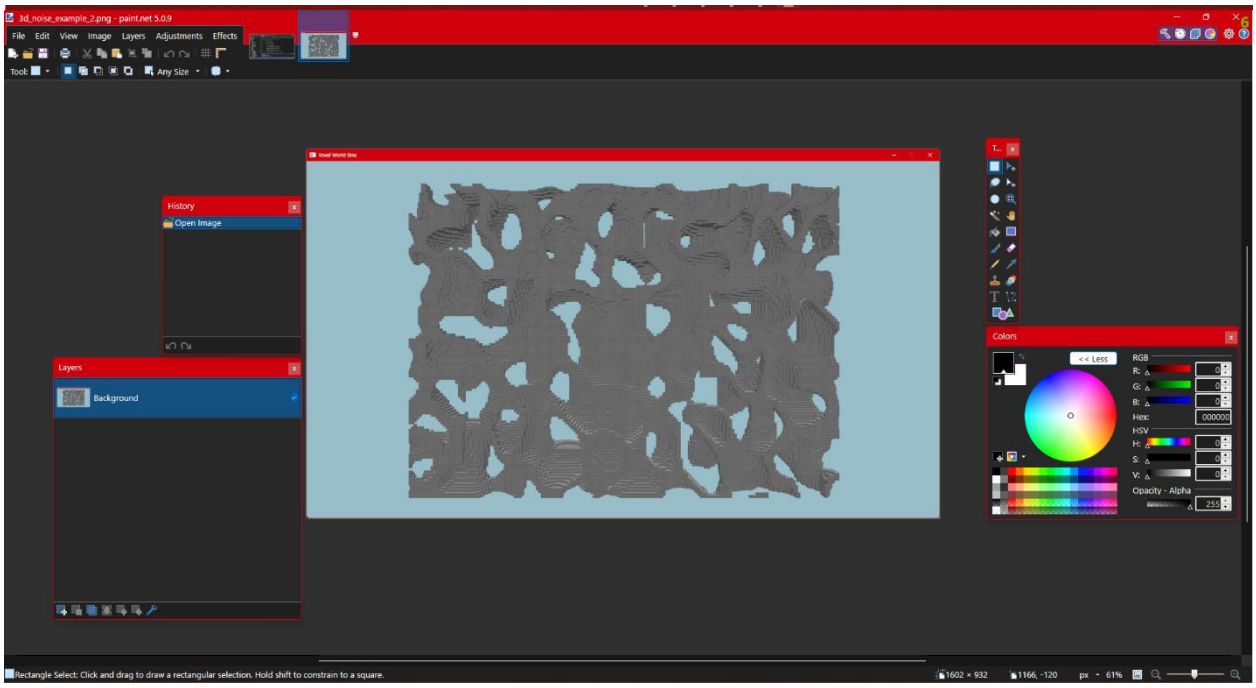
Biblioteka stb_image je biblioteka otvorenog koda napisana u C++ programskom jeziku koja pruža pojednostavljeno sučelje za učitavanje i zapisivanje slika različitih formata. Podržava *png, bmp* i *jpg* formate slika te podržava i rad na slikama poput mijenjanja veličine slike i uređivanje određenih piksela.



Slika 3.3. *Primjer slika generiranih koristeći FastNoise biblioteku za preslikavanje šuma u crno-bijelu sliku zapisanu koristeći stb_image biblioteku.*

3.4. Paint.NET

Paint.NET je napredni alat za uređivanje slika. Sadrži mogućnost uređivanja slika po slojevima koji se mogu preklapati jedan preko drugog te osim osnovnih alata za manipuliranje slikama pruža korisnicima generatore za jednostavno iscrtavanja šumova, fraktala, sijena i zamućivanja. Paint.NET je besplatno dostupan program javno dostupnog koda.



Slika 3.3. *Izlged programa za uređivanje slika Paint.NET*

4. GENERIRANJE VOXEL PROSTORA

Generiranje Voxel prostora podrazumijeva korištenje raznih algoritama i matematičkih funkcija za dobivanje virtualnog terena traženog oblika. Kako bi se prostorom lakše manipuliralo u računalnoj memoriji, prostor je potrebno razdijeliti u manje logičke cjeline koje predstavljaju jedan isječak prostora. Svaki isječak je definiran svojim položajem u 3D prostoru te terenom kojeg obuhvaća. Za potrebe ovog rada, teren svakog isječka je definiran kao 3D niz kocaka dimenzija 16x16x128 koje su predstavljene univerzalnim identifikatorom (engl. *id*) za kojeg se koristi podatkovni tip *unsigned char* čime je podržano maksimalnom 2^8, odnosno 256 različitih kocaka.

Ovim pristupom se problem generiranja Voxel prostora svodi na definiranje koja kocka, odnosno koji univerzalni identifikator kocke, se nalazi na određenim koordinatama unutar jednog isječka 3D terena. Prilikom generiranja terena je moguće da na određenim koordinatama ne postoji kocka te će se za taj slučaj smatrati da je na tim koordinatama kocka „zraka“ koja nema volumen, odnosno nije predstavljena nikakvim 3D modelom. Prilikom generiranja Voxel prostora, potrebno je generirati teren te napraviti model tog terena koji će se prikazivati unutar aplikacije.

4.1. Struktura aplikacije

Najveći problem prilikom generiranja Voxel prostora je potreba za definiranjem velikog broja kocaka unutar kratkog vremena. Kako je Voxel prostor trodimenzionalni prostor, bilo koji algoritam za generiranje terena će biti vremenske i prostorne složenosti n^3. Generiranje modela svakog isječka prostora je također iste vremenske i prostorne složenosti te sadrži dodatne uvijete i provjere koje mogu dodatno usporiti generiranje modela terena poput uvjetnog prikazivanja stranica kocaka koje čine isječak ili uvjetno prikazivanje stranica krajnjih kocaka koje se nalaze na rubovima isječaka te se prikazuju ovisno o postojanju susjednog isječka i kocaka unutar njega.

Kako bi se osigurale dobre performanse aplikacije, zadatke generiranje terena i modela prostora je potrebno odvojiti od iscrtavanja i upravljanja aplikacijom na način da se generiranje terena i modela odvija u odvojenim nitima procesa aplikacije od glavne niti koja je zaslužna za iscrtavanje prostora. Korištenje višenitnog programiranja zahtjeva dobar model komunikacije i sinkronizacije podatak između niti kako bi se spriječili neki od najčešćih problema u višenitnom programiranju poput potpunih zastoja i utrkama za podacima između niti. Zbog navedenih problema, potrebno je definirati zadatak svake niti, njezine podatke koji trebaju biti dostupni drugim nitima te sva moguća komunikacija između niti.

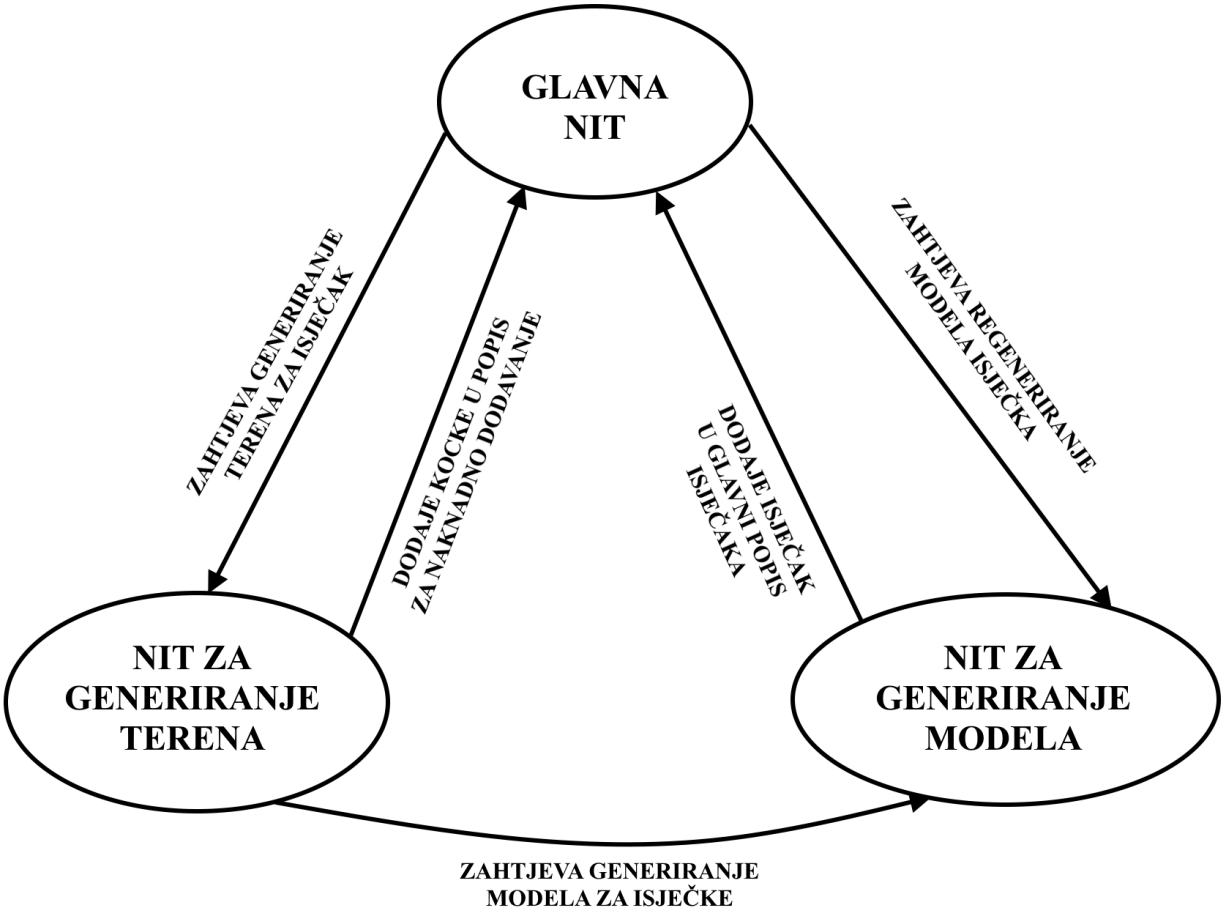
Implementacija aplikacije ovog rada se sastoji od jedne glavne niti, varijabilnog broja niti koje generiraju terena Voxel prostora te varijabilnog broja niti koje generiraju model isječaka Voxel prostora. Podržavanje varijabilnog broja niti za generiranje terena i modela isječaka omogućuje podešavanje brzine generiranja terena i modela u korist korisničkog iskustva.

Svaka od niti sadrži podatke koje dijeli između ostalih niti. Za implementaciju dijeljenja podataka između niti korišten je model kritičnih odsječaka koristeći mutekse koji omogućavaju da samo jedna nit pristupi određenom dijelu memoriju. U slučaju da više niti pokuša pristupiti istom kritičnom odsječku, ona nit koja je prva pristupila odsječku će imati pravo pristupa traženoj memoriji, dok će ostale niti biti suspendirane dok prva nit ne izađe iz kritičnog odsječka nakon čega sljedeća nit u redu ima pravo pristupa zatraženoj dijeljenoj memoriji. Na taj način je osigurano atomsko manipuliranje podacima (engl. *atomic data handling*). [4]

Glavna nit ove aplikacija služi kao početna točka cijelog programa. Ona je zaslužna za komuniciranje s grafičkom karticom preko OpenGL-a, učitavanje potrebnih resursa za aplikaciju poput tekstura i programa za sjenčanje, služi kao glavni izvor informacija o trenutnom stanju Voxel prostora, ažurira stanje terena te je zaslužna za učitavanje novih isječaka terena i ažuriranje modela već postojećih isječaka.

Niti za generiranje terena Voxel prostora, osim generiranja samog terena, komuniciraju glavnoj niti potrebna ažuriranja terena za isječke susjedne trenutnom isječku čiji se teren generira. Nakon što generiranje terena završi, ove niti šalju podatke isječaka za generiranje njihovih modela nitima za generiranje modela.

Niti za generiranje modela mogu generirati modele za novo-generirane isječke ili za već postojeće isječke prostora. U slučaju novo-generiranog isječka, ove niti šalju glavnoj niti zahtjev za dodavanje novog isječka u popis svih isječaka koje glavna nit ažurira i iscrtava. Ako ove niti generiraju model za već postojeći isječak, tada je potrebno samo zamijeniti model koji će se iscrtavati za traženi isječak.



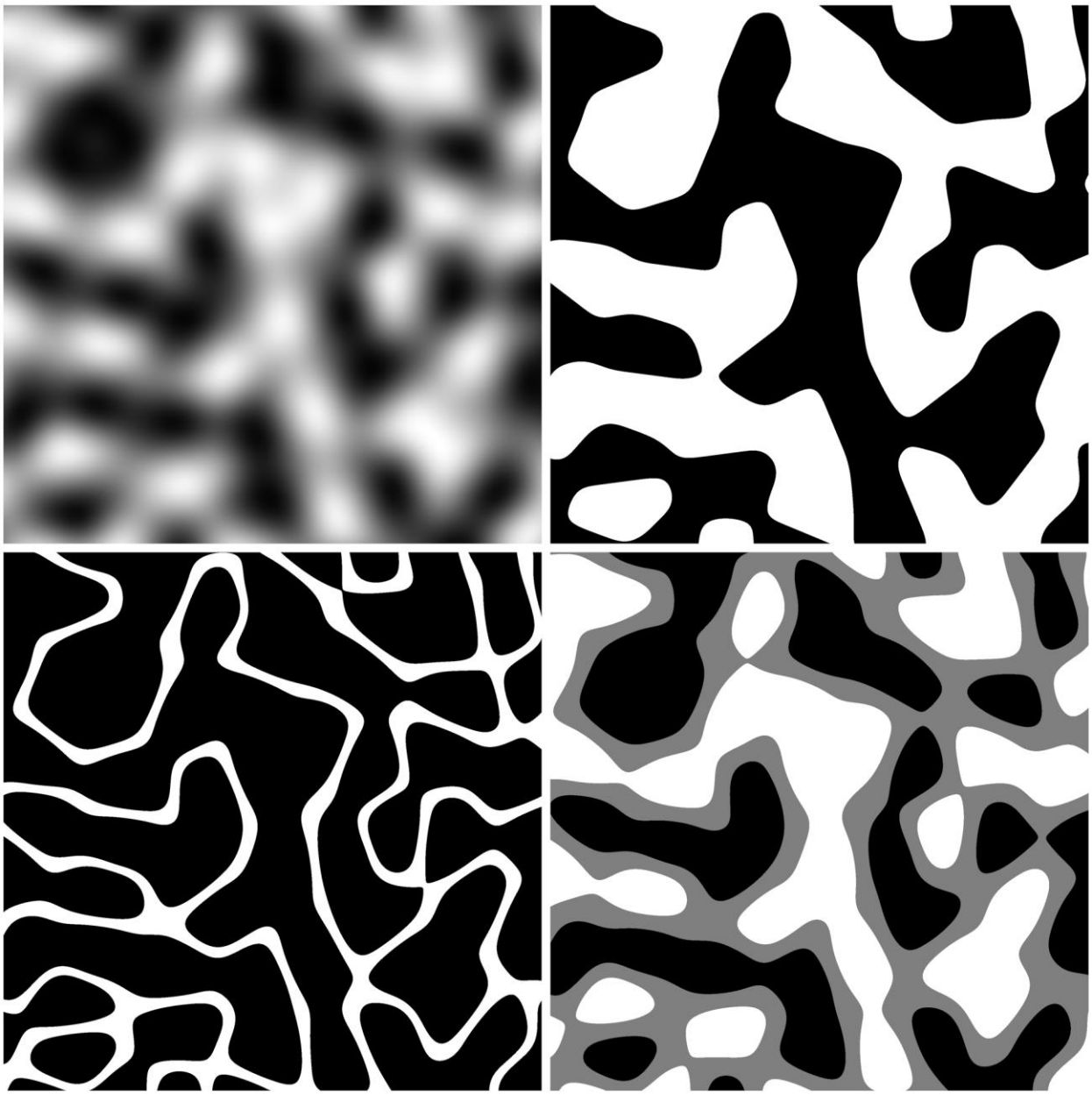
Slika 4.1. *Prikaz modela komunikacije glavnih dijelova aplikacije*

4.2. Generiranje šuma

Generiranje terena za Voxel prostor u ovom radu je implementirano preslikavajući izlaznih vrijednosti dvodimenzionalnog i trodimenzionalnog Perlin i Simplex šuma u određene kocke koje čine teren. Generatori šumova su funkcije koje za zadano sjeme, koje je predstavljeno podatkovnim tipom *unsigned int*, i predani im koordinatama vraćaju vrijednosti u intervalu [0, 1]. Mijenjajući početno sjeme generatora moguće je za iste koordinate promijeniti izlaznu vrijednost generatora što omogućuje veliku raznolikost krajnjih rezultata te mogućnost odabira izgleda nasumično generiranog terena. Osim izravnog korištenja i preslikavanja izlaza generatora šumova, često se koristi njihova modificirana vrijednost koristeći dodatne uvijete ili interpolatore.

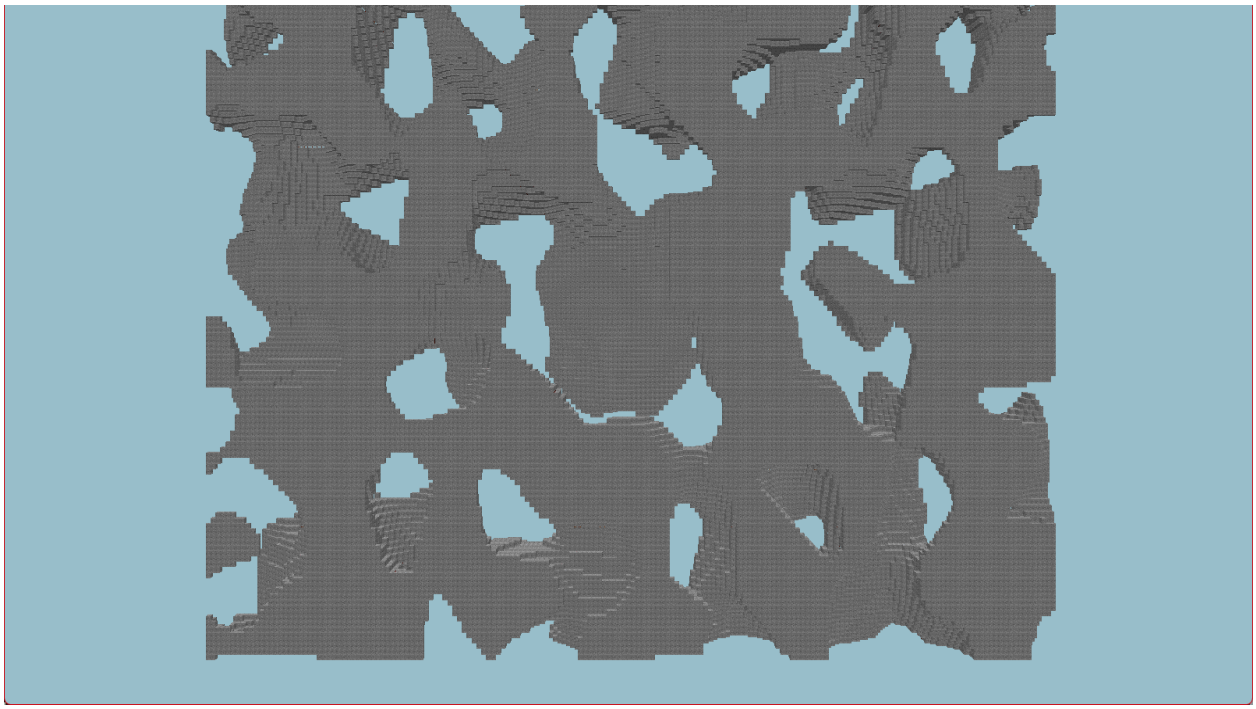
Slika 4.2. *Prikaz preslikavanja dvodimenzionalnog Simplex šuma u crno-bijelu sliku. Ulazi generatora šuma su koordinate piksela na slici, a izlazna vrijednost se preslikava iz intervala [0.0, 1.0] u interval [0, 255]. Crna boja predstavlja područja blizu vrijednosti 0, dok bijela područja predstavljaju vrijednosti bluzu 1. Korišteni brojevi za sjeme slika su 0, 123 i 456.*

Prilikom preslikavanja izlaza generatora šuma dodatnim uvjetima, najčešće se provjerava pripada li izlaz generatora šuma određenom intervalu nakon čega se ta vrijednost preslikava u prethodno definiranu vrijednost. Na primjeru crno-bijelih slika, ovakav način preslikavanja zamjenjuje postepen prelazak iz crne u bijelu boju s oštrim prijelazima boja. Na slici 4.3. prikazani su rezultati korištenja preslikavanja izlaza generatora šuma čije je sjeme postavljeno na 0. Na prvoj slici je direktno preslikan izlaz generatora šuma bez uvjeta. Na drugoj slici su pikseli za čije koordinate generator šuma daje vrijednost manju od 0.5 obojani crnom bojom, a ostali su obojani bijelom bojom. Na trećoj slici su pikseli za čije koordinate generator šuma da vrijednost u intervalu [0.4, 0.6] obojani bijelom bojom, a ostali pikseli su obojani crnom bojom. Na zadnjoj slici su pikseli čije koordinate daju kao izlaz generatora šuma vrijednost veću od 0.66 obojani bijelo, pikseli čiji koordinate daju vrijednost manju od 0.33 su obojani crnom bojom, a ostali su obojani sivom bojom.

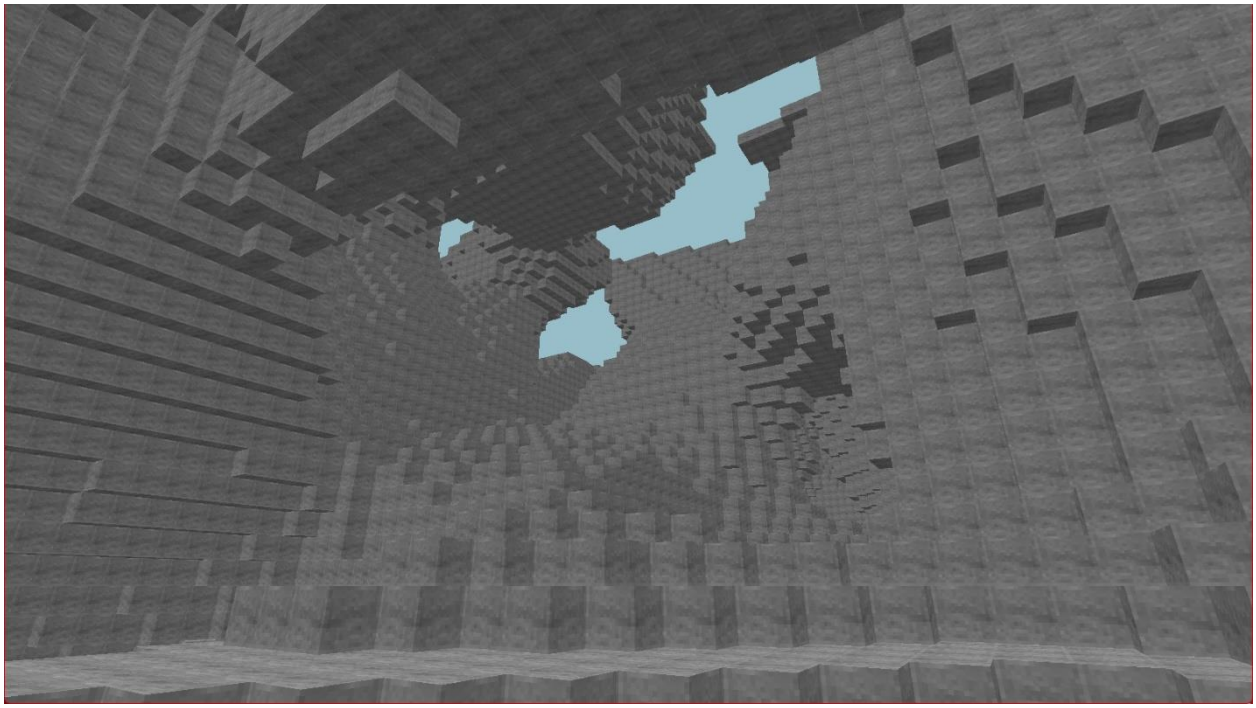


Slika 4.3. *Primjer preslikavanja izlaza generatora Simplex šuma koristeći dodatne uvijete*.

Primjenjivanje uvjetnog preslikavanja je moguće i kod trodimenzionalnih šumova. Slika 4.4. prikazuje preslikavanje trodimenzionalnog šuma na način ako je izlaz generatora šuma za dane koordinate manja od 0.5, biti će postavljena određena kocka. Za koordinate za koje izlaz generatora šuma daje vrijednost veću ili jednaku 0.5, neće biti postavljena kocka te će izgledati kao prazan prostor.



Slika 4.4. *Vertikalni presjek trodimenzionalnog šuma preslikanog koristeći dodatne uvijete.*



Slika 4.5. *Prikaz unutrašnjosti trodimenzionalnog prostora s terenom generiranim koristeći trodimenzionalni Simplex šum preslikanog koristeći dodatne uvijete*

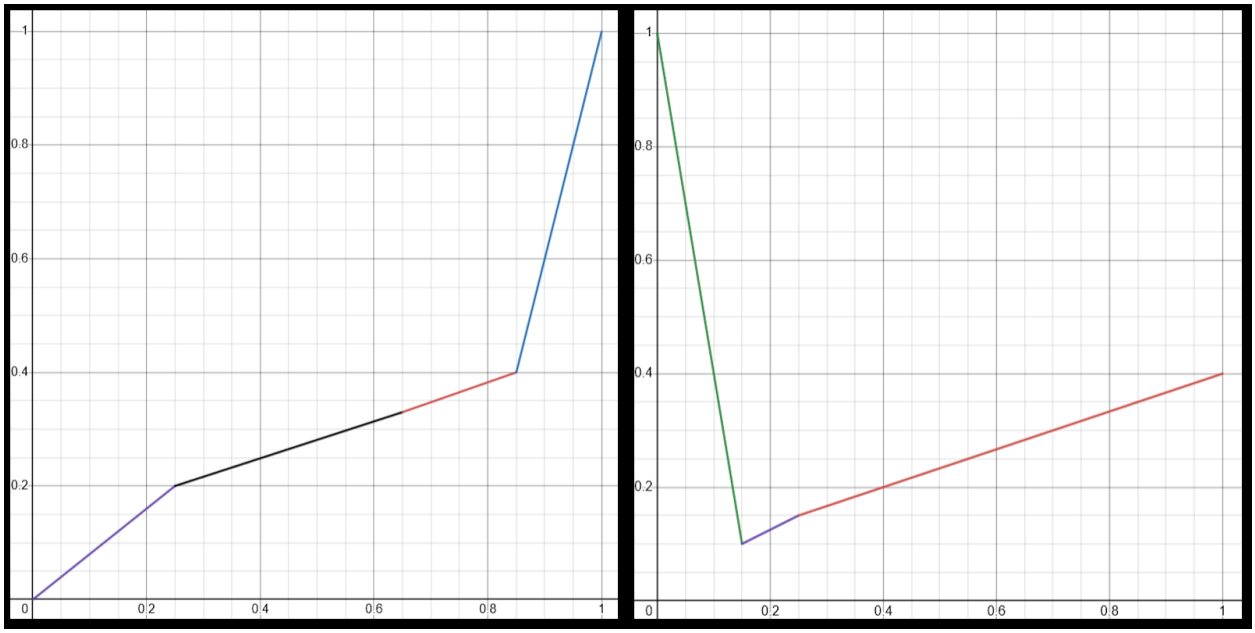
Interpoliranje vrijednosti izlaza generatora šuma se odnosi na primjenjivanje određene funkcije ili skupa funkcija na izlaz generatora što omogućuje detaljnije upravljanje izlaznih vrijednosti generatora. Za potrebe ovoga rada koriste se linearni interpolatori koji preslikavaju izlaze generatora šuma koristeći linearne funkcije u zadane vrijednosti. Kako bi bilo moguće linearno interpolirati vrijednosti u intervalu [0.0, 1.0], potrebno zadati minimalno 2 para brojeva pri čemu prvi broj u paru predstavlja osnovni izlaz generatora šuma, dok drugi broj u paru predstavlja vrijednost u koju se izlaz treba preslikati.

```
double LinearInterpolator::interpolate(double value) const
{
    size_t lowerIndex = 0;

    for (size_t i = 0; i < pairs.size(); ++i)
    {
        if (std::get<0>(pairs[i]) <= value)
        {
            lowerIndex = i;
        }
        else
        {
            break;
        }
    }

    const auto& [x1, y1] = pairs[lowerIndex];
    const auto& [x2, y2] = pairs[std::min(lowerIndex + 1, pairs.size() - 1)];

    double coefficient = (y2 - y1) / (x2 - x1);
    double interpolation = y1 + (value - x1) * coefficient;

    return interpolation;
}
```

Slika 4.6. *Implementacija algoritma za linearnu interpolaciju brojeva između parova točaka.*

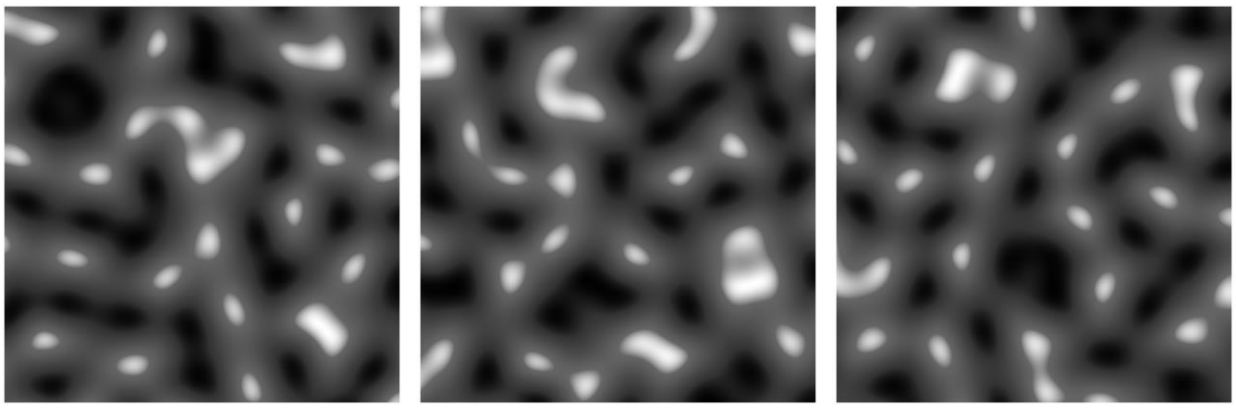
Vizualni prikaz interpolatora je graf s najmanje jednim omeđenim pravcem čija x os predstavlja ulaznu vrijednost interpolatora, a y os izlaznu vrijednost. Na slici 4.7. se nalazi vizualni prikaz dva različita interpolatora. Prvi interpolator preslikava vrijednosti do broja 0.85 u vrijednosti u intervalu [0.0, 0.4], a za brojeve veće od 0.85, preslikava vrijednosti u interval [0.4, 1.00]. Drugi interpolator vrijednosti manje od 0.15 preslikava u vrijednosti u interval [0.1, 1.0] na način da manje vrijednosti preslikava u veće i obratno. Vrijednosti iznad 0.15 su ravnomjerno

preslikane u vrijednosti u intervalu [0.1, 0.4]. Korištenjem interpolatora na izlaz generatora moguće je napraviti pristranost vrijednosti izlaza oko traženih vrijednosti.



Slika 4.7. *Vizualni prikaz interpolatora.*

Primjenjivanjem prvog linearnog interpolatora sa slike 4.7. na izlaz generatora šumova sa slike 4.2. dobije se sljedeći prikaz:

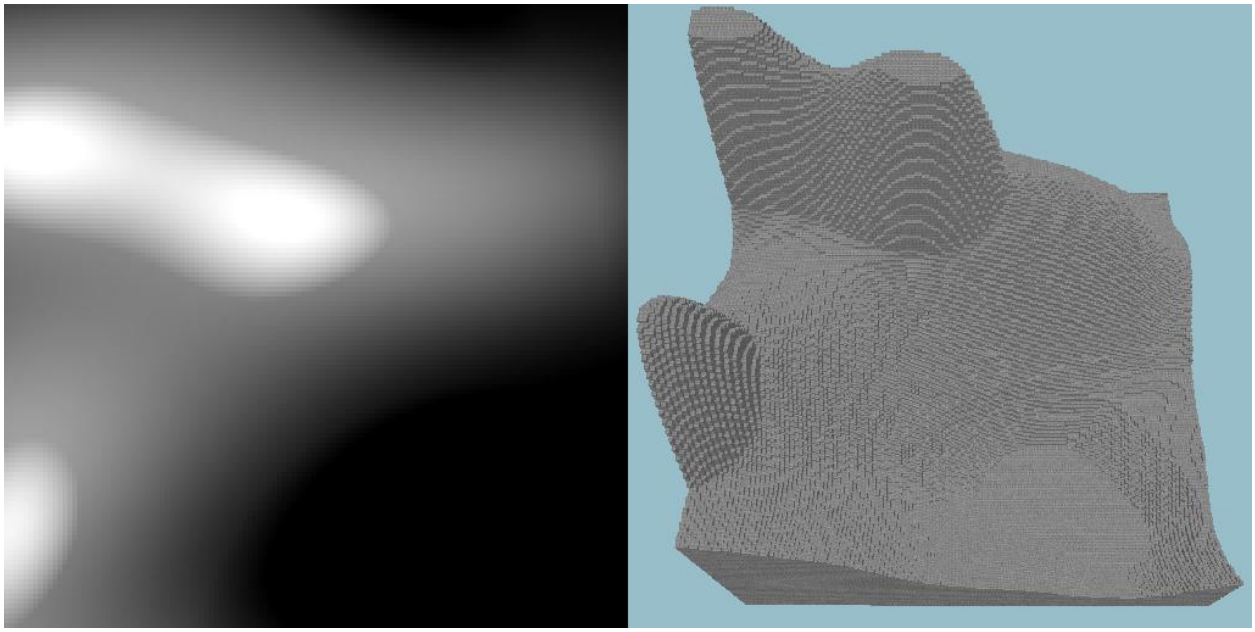


Slika 4.8. *Prikaz primijenjenog linearnog interpolatora na izlaz generatora šuma preslikano u crno-bijelu sliku*

4.3. Generiranje terena

Implementacija generiranja terena u ovom radu se odvija za svaki isječak u tri koraka: generiranje visinske karte, generiranje špilja te ukrašavanje generiranog terena.

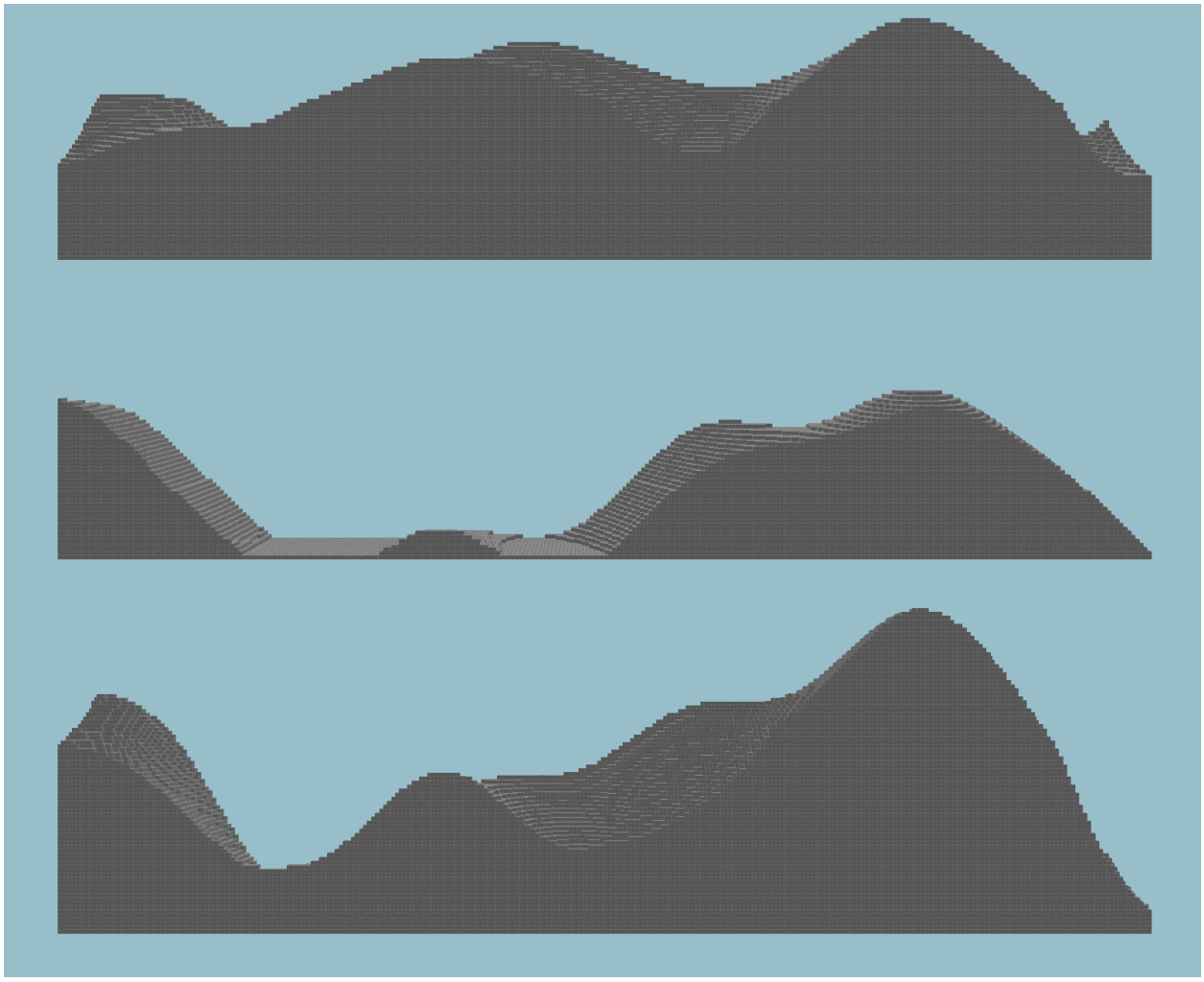
Generiranje visinske karte odnosi se na generiranje dvodimenzionalnog polja popunjenog izlaznim vrijednostima generatora šuma te naknadnim interpoliranjem i preslikavanjem u interval [0, 127], što odgovara prethodno definiranim zahtjevima navedenim na početku ovog poglavlja. Prilikom generiranja terena, sve kocke unutar isječka čija je y koordinata manja ili jednaka vrijednosti u visinskoj karti na istim x i z koordinatama je popunjena određenim tipom kocke, u ovom radu kamenom, a sve kocke čija je y koordinata veća od one vrijednosti u visinskoj karti se popunjavaju zrakom.



Slika 4.8. *Primjer dvodimenzionalnog šuma preslikanog u trodimenzionalni teren*

Za potrebe ovog rada, korištena su dva generatora šuma za generiranje visinske karte čiji se rezultati nakon interpoliranja zbrajaju te preslikavaju u interval [0, 127]. Prvi interpolirani generator šuma služi za generiranje terena koji sadrži ravnija područja i nasumično postavljene brežuljke te se vrijednosti ovog generatora nazivaju vrijednosti kontinentalnosti. Drugi interpolirani generator šuma služi za dodavanje nasumičnosti prethodno generiranom šumu. Interpoliranjem ovog šuma u interval [-0.25, 0.25] postiže se efekt erozije terena te se vrijednosti

ovog generatora šuma nazivaju vrijednosti erozije. Kombinirajući vrijednosti kontinentalnosti i erozije postiže se izgledom realno nasumičan teren.

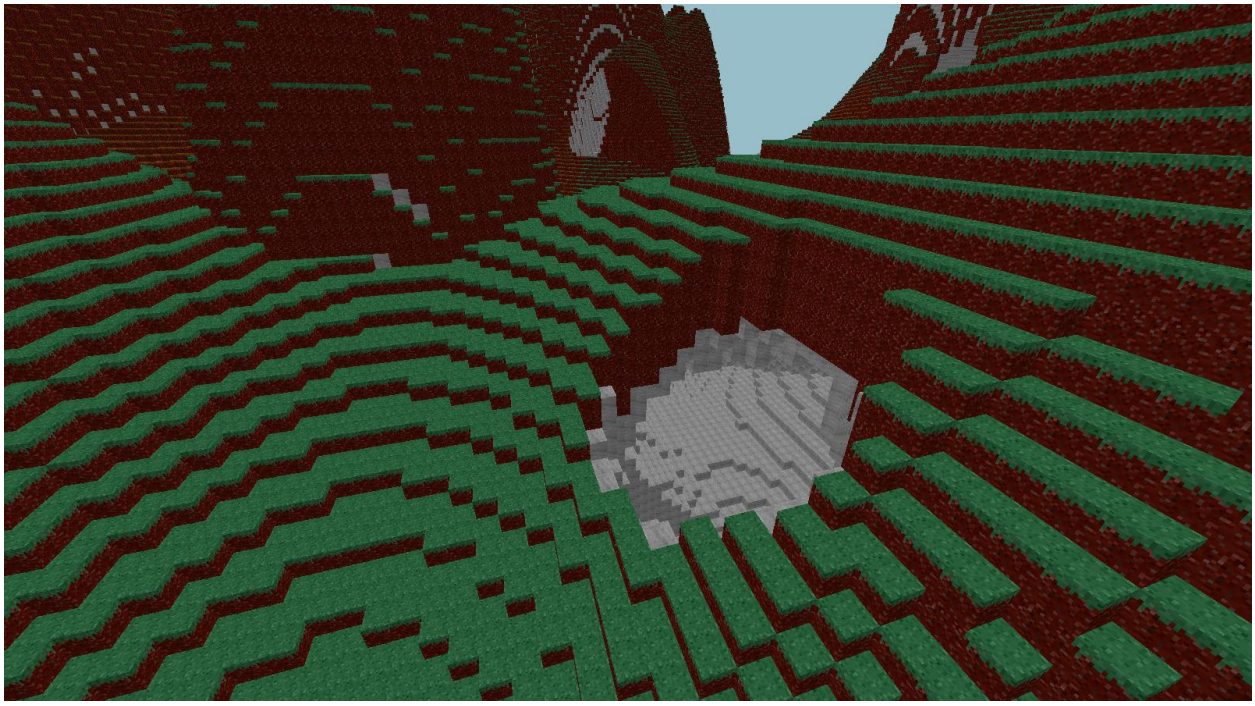


Slika 4.9. *Prikaz terena generiranih koristeći interpolirani šum kontinentalnosti, interpolirani šum erozije i kombinacije interpoliranih šumova kontinentalnosti i erozije*

Prilikom generiranja terena za potrebe ovoga rada implementirano je i generiranje špilja koje daje dinamičan izgled terenu ispod površine na način da se prethodno generirane kocke terena zamijene zrakom. Implementirane su dvije vrste špilja: površinske i podzemne.

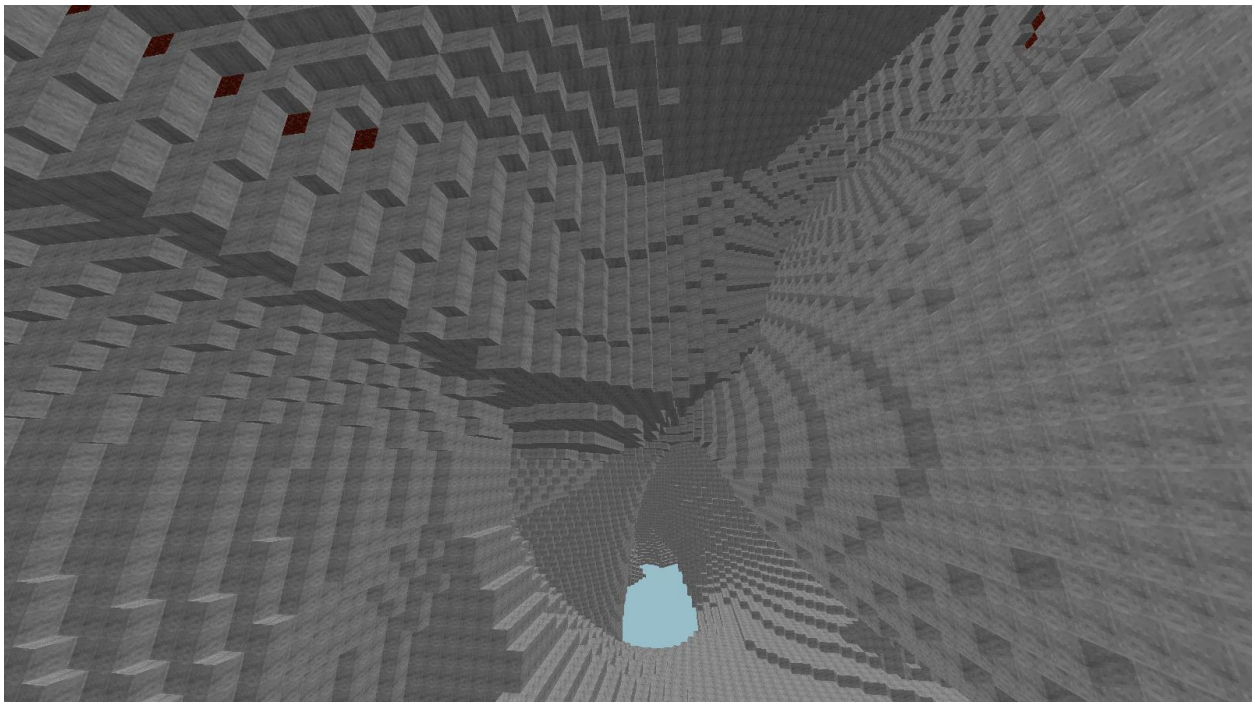
Površinske špilje su špilje čiji ulaz se nalazi na površini prethodno generiranog terena te stvara prolaz do podzemnih špilja. Implementirane su koristeći generator šuma uz dodatni uvjet pri kojem ako je izlaz generatora šuma za zadane koordinate kocke veći od 0.9, tada će se ta kocka zamijeniti kockom zraka. Jedna od karakteristika površinskih špilja je njihova razvučenost po y

osi. Način dobivanja ovakve razvučenosti je množenje y-koordinate s određenim faktorom prilikom generiranja šuma. Za vrijednosti faktora manjih od 1, površinske špilje će biti razvučene po y-osi, a za vrijednosti faktora većih od 1, površinske špilje će biti kraće po y-osi.



Slika 4.10. *Primjer površinske špilje u prostoru*

Podzemne špilje se generiraju uz uvjet da su 10 kocaka ispod površine terena te su implementirane koristeći jednostavni trodimenzionalni generator šuma. Ako je izlaz generatora šuma u intervalu <0.3, 0.45> za zadane koordinate kocke, tada će se ta kocka zamijeniti kockom zraka što daje konačni izgled podzemnim špiljama.



Slika 4.11. *Primjer generiranih podzemnih špilja*

4.4. Ukrašavanje terena

Nakon što generiranje terena isječka završi, spremno je za ukrašavanje. Ukrašavanje terena se odnosi na dodavanje, brisanje i mijenjanje kocaka unutar isječka kako bi se dobili traženi rezultati. Prilikom ukrašavanja terena dostupni su podaci o glavnom sjemenu svijeta, položaju isječka kojeg se ukrašava, visinska karta te prethodno generirane kocke terena.

Primjeri ukrašavanja terena bi bili dodavanje kocaka vode ispod određene visine, zamjenjivanje površinskih blokova s blokovima trave ili dodavanje ukrasa na površini vode poput lopoča i šaša.

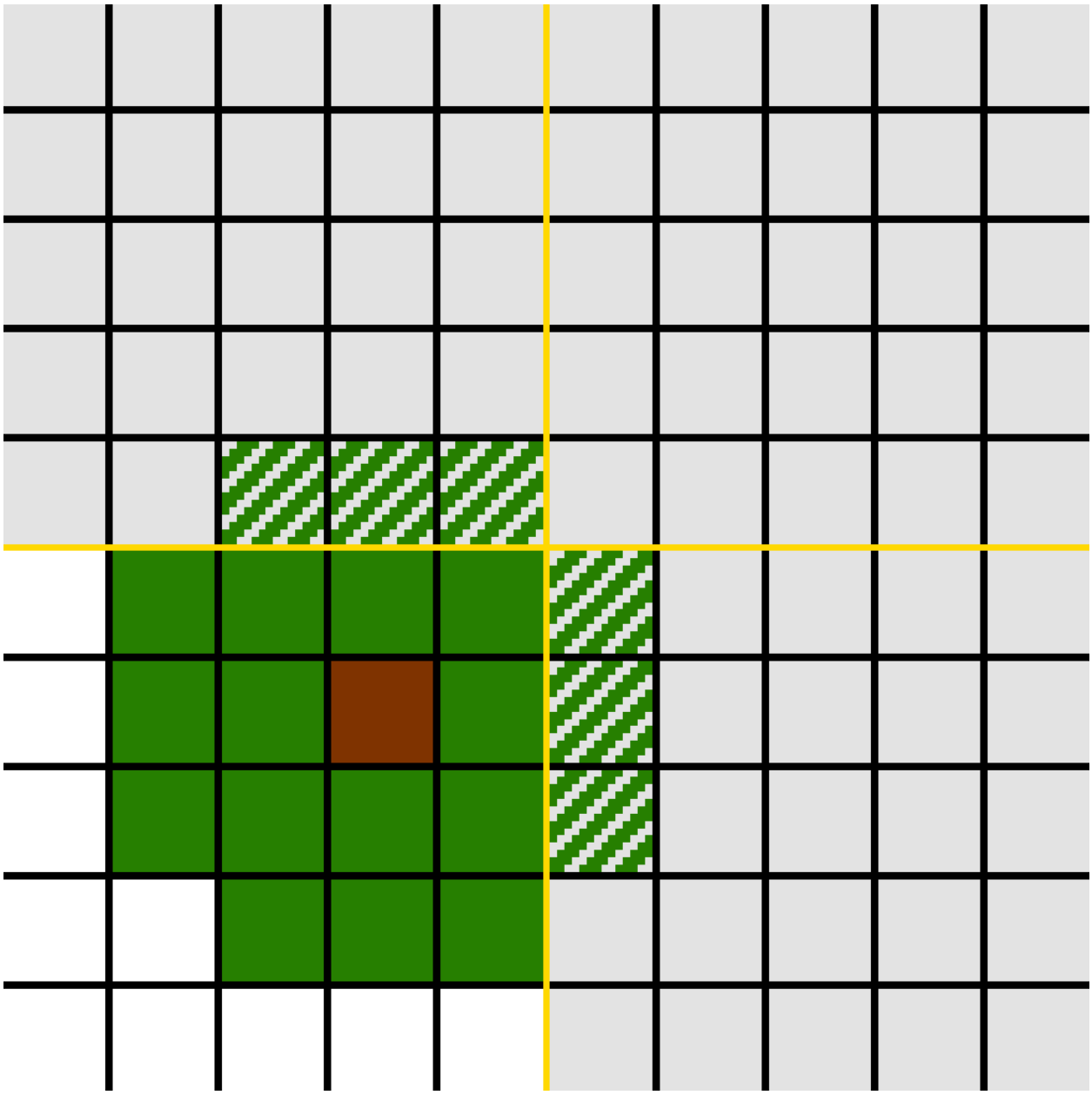
```
std::unordered_map<Vector, std::unordered_map<Vector, unsigned char>>
    LakeDecorator::decorate(const Vector& chunkPosition, unsigned char* blocks, const int* heightMap) const
{
    for (int x = 0; x < CHUNK_WIDTH; x++)
    {
        for (int z = 0; z < CHUNK_WIDTH; z++)
        {
            for (int y = heightMap[x * CHUNK_WIDTH + z]; y <= lakeLevel; y++)
            {
                blocks[Chunk::coordsToOffset(x, y, z)] = y == lakeLevel ? BLOCK_WATER_SURFACE : BLOCK_WATER;
            }
        }
    }

    return {};
}
```

Slika 4.12. *Algoritam za ispunjavanje isječka vodom do zadane razine*

```
void LakeSurfaceDecorator::placeSedges(int x, int z, unsigned char* blocks, const int* heightMap) const
{
    unsigned char blockBelow = blocks[(lakeLevel - 1) * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z];
    unsigned char blockBelow2 = blocks[(lakeLevel - 2) * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z];

    bool isOneBlockDeepWater = blockBelow != BLOCK_WATER;
    bool isTwoBlockDeepWater = blockBelow == BLOCK_WATER && blockBelow2 != BLOCK_WATER;

    if (!(isOneBlockDeepWater || isTwoBlockDeepWater)) return;

    blocks[lakeLevel * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z] = BLOCK_SEDGE;
    blocks[(lakeLevel + 1) * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z] = BLOCK_SEDGE_TOP;
}

void LakeSurfaceDecorator::placeLilyPads(int x, int z, unsigned char* blocks, const int* heightMap) const
{
    unsigned char blockAboveWater = blocks[(lakeLevel + 1) * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z];

    if (blockAboveWater != BLOCK_AIR) return;

    blocks[(lakeLevel + 1) * CHUNK_WIDTH * CHUNK_WIDTH + x * CHUNK_WIDTH + z] = (std::abs(x + z) % 10 < 5) ? BLOCK_LILY_PAD : BLOCK_LILY_PADS;
}
```

Slika 4.13. *Isječak algoritma za ukrašavanje vodenih površina isječka*

Problem koji se pojavljuje prilikom ukrašavanja isječaka je potreba za ukrašavanjem susjednih isječaka, odnosno postavljanjem kocaka na određenim koordinatama susjednih isječaka. Jedan od primjera ovakvog ukrašavanja implementiranih u ovom radu je generiranje drveća unutar jednog isječka. Deblo drveta se rasprostire unutar jednog isječka duž y osi, dok se krošnja drveta rasprostire po horizontalnoj ravnini isječka. Ukoliko se drvo nalazi na rubu isječka, dio krošnje je potrebno generirati van granica trenutnog isječka. Zbog toga, prilikom generiranja ukrasa isječaka, potrebno je pratiti na koje koordinate se postavljaju određeni tipovi kocke, te ukoliko koordinate predstavljaju poziciju izvan isječka potrebno ih je spremiti u popis kocaka koje je potrebno naknadno postaviti za ostale isječke o čemu se brine glavna nit.

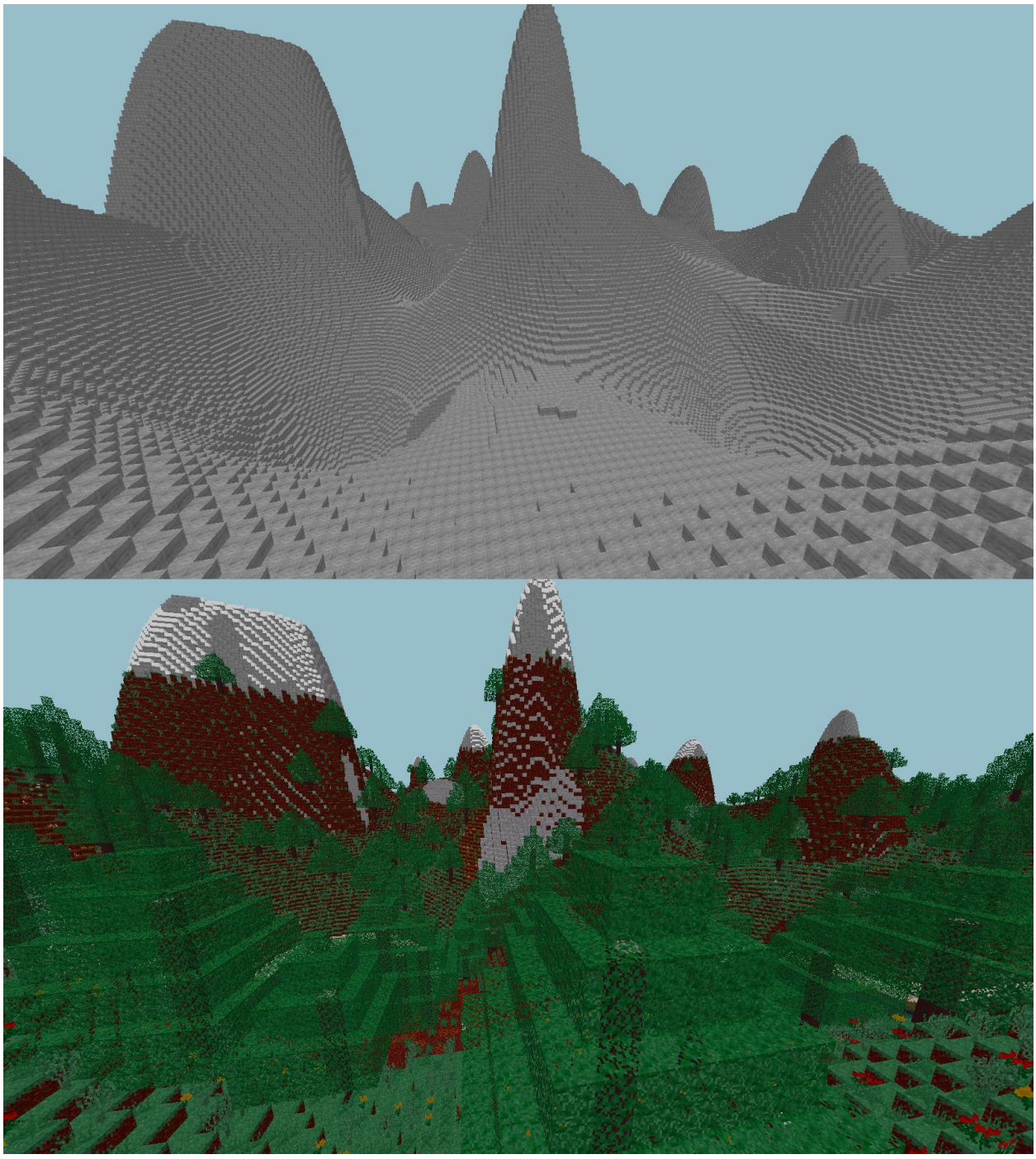


Slika 4.14. *Vizualni prikaz problema ukrašavanja susjednih isječaka. Žutom bojom su označeni rubovi isječaka, zelenom bojom kocke krošnje drveta, smeđe deblo drveta, a osjenčano zelenom bojom su označene kocke krošnje drveta koje treba postaviti u susjednim isječcima*

Nakon što završi ukrašavanje isječka, nit za generiranje isječaka razmjenjuje podatke s glavnom niti o kockama koje treba postaviti u susjednim isječcima. Prilikom razmijene tih podataka, potrebno je i dohvatiti one kocke koje je potrebno naknadno postaviti u trenutni isječak zbog mogućnosti da je prije nego što se generirao isječak zatražilo postavljanje kocaka u isti.

```
void ChunkGenerator::generateTerrain(const Vector& chunkPosition)
{
    int* heightMap = terrainGenerator->generateHeightMap(int(chunkPosition.x), int(chunkPosition.z));

    // 1) Generate the terrain
    unsigned char* blocks = terrainGenerator->generateTerrain(seed, chunkPosition, heightMap);

    // 2) Add decorations to this chunk
    std::unordered_map<Vector, std::unordered_map<Vector, unsigned char>> neighbourDecorations;

    for (TerrainDecorator* decorator : terrainDecorators)
    {
        auto decoration = decorator->decorate(chunkPosition, blocks, heightMap);

        for (auto& [chunkPosition, blocks] : decoration)
        {
            neighbourDecorations[chunkPosition].insert(blocks.begin(), blocks.end());
        }
    }

    delete[] heightMap;

    // 3) Add any blocks that are waiting to be added to this chunk
    std::unordered_map<Vector, unsigned char> pendingBlocks = world->exchangePendingBlocks(chunkPosition, neighbourDecorations);
    for (auto& [position, blockID] : pendingBlocks)
    {
        blocks[Chunk::coordsToOffset(position)] = blockID;
    }

    Chunk* newChunk = new Chunk(chunkPosition, world, blocks);
    chunkMesher->requestMesh(newChunk);
}
```

Slika 4.15. *Osnovni dio algoritma za generiranje i ukrašavanje isječaka*



Slika 4.16. *Usporedba generiranog prostora prije i poslije ukrašavanja*

4.5. Glavna nit

Glavna nit ove aplikacije je zaslužna za ažuriranje stanja isječaka, zahtijevanje novih isječaka te iscrtavanje isječaka. Prilikom ažuriranja stanja isječaka, glavna nit izvršava sljedeće zadatke:

1) ažuriranje modela isječaka
2) postavljanje kocaka u zatražene isječke
3) dodavanje novih isječaka u popis isječaka za ažuriranje
4) slanje zahtjeva niti za generiranje novih modela i
5) ažuriranje varijable koja predstavlja vrijeme unutar aplikacije.

Prilikom rukovanja stanjima isječaka, koristi se termin prljavog isječka (engl. *dirty chunk*) koji označava isječak čije se stanje promijenilo, odnosno isječak unutar kojeg je došlo do promjene kocaka, te je potrebno ažurirati model isječka kako bi predstavljao ažurno stanje isječka. Glavna nit stalno provjerava popis svih isječaka te za one koji su označeni prljavima zahtjeva od niti za generiranje modela da generira novi model za određeni isječak.

```
void World::handleDirtyChunks()
{
    std::vector<Chunk*> dirtyChunks;

    for (const auto& [position, chunk] : chunks)
    {
        if (chunk->isDirty())
        {
            dirtyChunks.push_back(chunk);
            chunk->markDirty(false);
        }
    }

    chunkMesher.requestRemeshes(dirtyChunks);
}
```

Slika 4.17. *Algoritam za zahtijevanje novih modela za prljave isječke*

Nakon što nit za generiranje modela završi s generiranjem modela za zatraženi isječak, glavna nit je zadužena za ažuriranje modela isječka te učitavanja podataka modela na grafičku karticu.

```
void World::updateCleanedChunkMeshes()
{
    cleanedChunks.perform([this](std::vector<std::tuple<Chunk*, ChunkMesh*>>& cleanedChunks) -> void
        {
            for (auto& [chunk, chunkMesh] : cleanedChunks)
            {
                chunk->updateChunkMesh(chunkMesh);
                chunk->markDirty(false);
            }

            cleanedChunks.clear();
        });
}
```

Slika 4.18. *Algoritam za ažuriranje modela isječaka nakon završenog generiranja novih modela*

Glavna nit se također brine o novo-generiranim isječcima ubacujući ih u popis svih isječaka. Problem koji se pojavljuje nakon dodavanja novog isječka je neispravnost modela susjednih isječaka na rubovima koji graniče s novim isječkom. Zbog toga se nakon dodavanja novog isječka svi susjedni isječci koji su učitani u memoriju označavaju prljavima za koje će se kasnije zatražiti ažuriranje modela isječka.

```
void World::addPendingChunks()
{
    pendingChunks.perform([this](std::vector<std::tuple<Chunk*, ChunkMesh*>>& pendingChunks) -> void
        {
            for (auto& [chunk, chunkMesh] : pendingChunks)
            {
                chunk->updateChunkMesh(chunkMesh);
                this->chunks[chunk->getPosition()] = chunk;
            }

            for (auto& [chunk, chunkMesh] : pendingChunks)
            {
                for (const Vector& position : chunk->getNeighbouringPositions())
                {
                    if (Chunk* neighbourChunk = this->getChunkAt(position))
                    {
                        neighbourChunk->markDirty(true);
                    }
                }
            }

            pendingChunks.clear();
        });
}
```

Slika 4.19. *Algoritam za dodavanje novih isječaka u popis učitanih isječaka te za označavanje susjednih isječaka prljavima*

Osim što dodavanje novih isječaka može uzrokovati neispravnosti modela na rubovima sa susjednim isječcima, uređivanja stanja isječka, odnosno mijenjanje kocaka isječka, također zahtjeva ažuriranje modela isječka. Glavna nit nakon dodavanja kocaka koje su nastale kao dekoracije proizašle iz ukrašavanja drugih isječaka mora označiti iste isječke prljavim kako bi se naknado zahtijevalo ažuriranje novog modela isječka.

```
void World::handlePendingBlocks()
{
    pendingBlocksMutex.lock();

    std::vector<Vector> checkedPositions;

    for (auto& [chunkPosition, blocks] : pendingBlocks)
    {
        if (Chunk* chunk = getChunkAt(chunkPosition))
        {
            for (auto& [blockPosition, blockID] : blocks)
            {
                chunk->setBlock(blockPosition, blockID);
            }

            checkedPositions.push_back(chunkPosition);
            blocks.clear();
            chunk->markDirty(true);
        }
    }

    for (const Vector& checkedPosition : checkedPositions)
    {
        pendingBlocks.erase(checkedPosition);
    }

    pendingBlocksMutex.unlock();
}
```

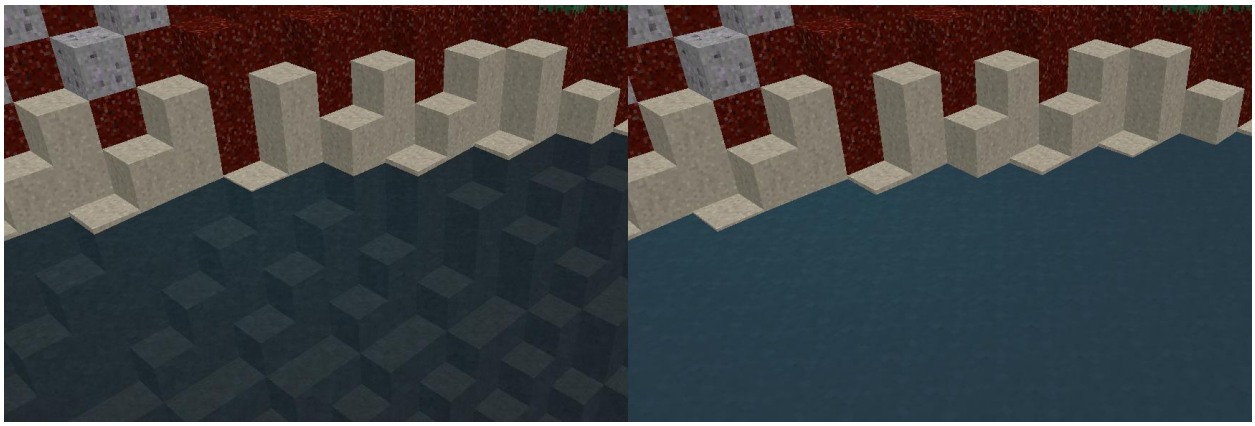
Slika 4.20. *Algoritam za dodavanje kocaka u isječke nastalih nakon generiranja isječaka*

5. ISCRTAVANJE VOXEL PROSTORA

Iscrtavanje voxel prostora obuhvaća dva područja – generiranje modela i iscrtavanje modela. Kako je u ovom radu prostor podijeljen u isječke, problem generiranja modela i iscrtavanja prostora se razlaže na generiranje modela zasebnih isječaka i njihovog iscrtavanja.

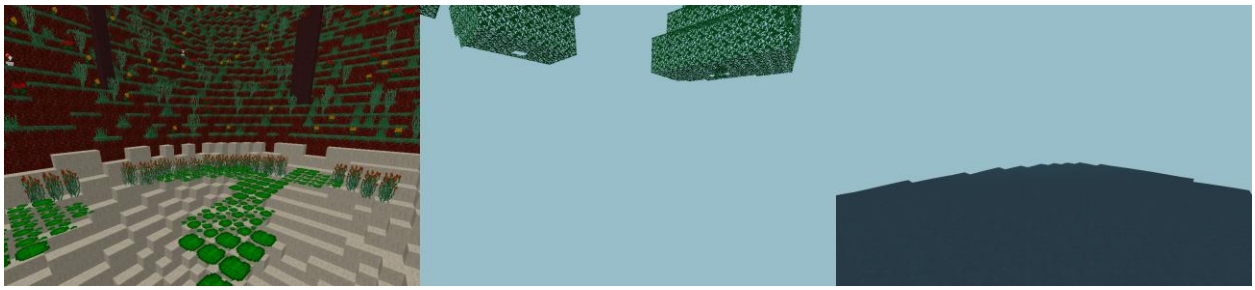
5.1. Metode iscrtavanja prostora

Prilikom iscrtavanja 3D modela na zaslon, bitno je poštivati redoslijed iscrtavanja modela. Najprije se iscrtavaju neprozirni dijelovi modela, a zatim prozirni dijelovi. Na taj način se dobiva točan prikaz transparentnih tekstura i slika unutar prostora.



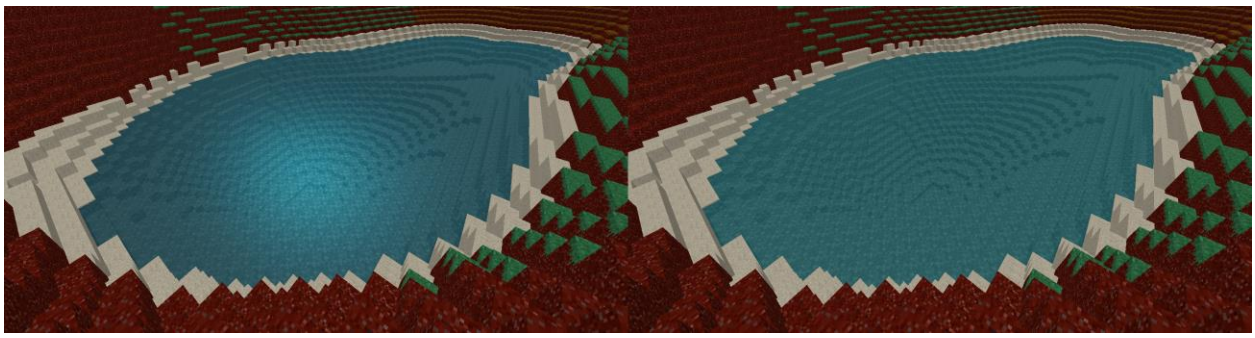
Slika 5.1. *Primjer pravilnog redoslijeda iscrtavanja modela (lijevo) i nepravilnog redoslijeda iscrtavanja (desno)*

Kako je prostor podijeljen na isječke, za svaki isječak je potrebno generirati model koji će se iscrtavati. Za potrebe ovog rada, model isječka je podijeljen u 3 pod-modela: model koji sadrži neprozirne dijelove, model koji sadrži prozirne dijelove i model koji sadrži modele kocaka vode. Model koji sadržava neprozirne dijelove čine kocke koje predstavljaju travu, kamen, pijesak ili drvo, model koji sadrži prozirne dijelove čine kocke koje predstavljaju cvijeće, lišće i slično. Kocke koje predstavljaju zrak se, iako su teoretski prozirne jer se ne vide, prilikom iscrtavanja zanemaruju.



Slika 5.2. *Dekompozicija prostora na tri vrste modela: netransparentni model, transparentni model i model vode*

Kao dodatan detalj u ovom radu, kocke koje predstavljaju vodu se zasebno iscrtavaju od ostatka terena kako bi se omogućile dodatne refleksije svjetla na površini vode te efekt gibanja vode, odnosno valova. Refleksija svjetla na površini vode je izvedena koristeći Phongov model osvjetljenja koji se sastoji od tri komponente: ambijentalnog, difuznog i spekularnog osvjetljenja. Ambijentalna rasvjeta predstavlja temeljno osvjetljenje određenog modela, difuzno osvjetljenje predstavlja osvjetljenje koje površine modela dobivaju od izvora svjetlosti ovisno o orijentacija svjetla i površine, a spekularno osvjetljenje predstavlja odsijevanje izvora svjetlosti na površini modela. Phongov model osvjetljenja se implementira u programu za sjenčanje fragmenata (engl. Fragment Shader) koji omogućuje dodatno manipuliranje izlaznog prikaza zaslona na bazi fragmenata. Koristeći Phongov model osvjetljenja na površini vode dobije se realističniji prikaz vode koji daje dodatnu razliku kockama vode od ostatka terena. [6]



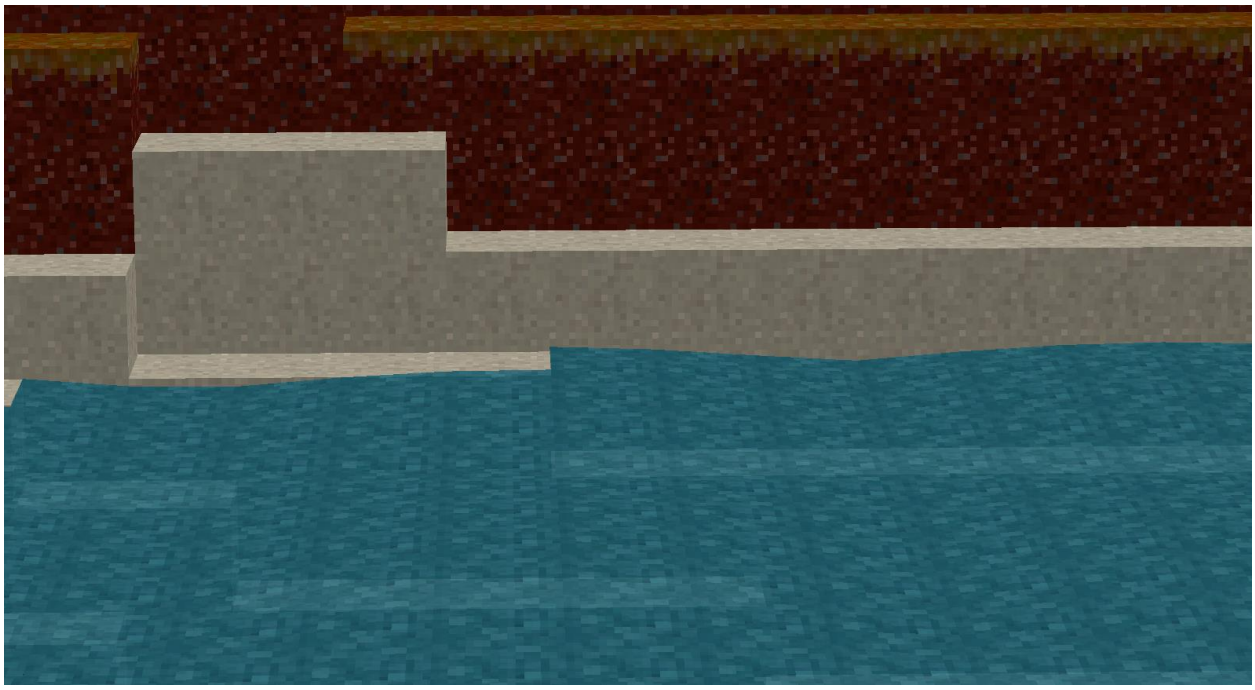
Slika 5.3. *Usporedba izgleda vodene površine iscrtane koristeći Phongov model osvjetljenja (lijevo) i bez Phongovog modela osvjetljenja (desno)*

```
#version 330 core

in vec2 uv;
in vec3 normal;
in vec3 fragment_position;
in vec4 color;

uniform vec3 view_position;
uniform sampler2D object_texture;
uniform bool enablePhong;;

out vec4 FragColor;

const vec3 LIGHT_POSITION = vec3(10.0);

vec4 getAmbientLight()
{
    const float AMBIENT_STRENGTH = 1.0;
    const vec3 AMBIENT_LIGHT = vec3(0.2, 0.2, 0.3);

    return vec4(AMBIENT_LIGHT * AMBIENT_STRENGTH, 1.0);
}

vec4 getDiffuseLight(vec3 norm, vec3 lightDirection)
{
    const vec3 DIFFUSE_LIGHT = vec3(1.0);

    float diff = max(dot(norm, lightDirection), 0.0);

    return vec4(diff * DIFFUSE_LIGHT, 1.0);
}
```

```
vec4 getSpecularLight(vec3 norm, vec3 lightDirection)
{
    const float SPECULAR_STRENGTH = 1.0;
    const vec3 SPECULAR_LIGHT = vec3(1.0);
    const int SPECULAR_INTESITY = 2 << 3;

    vec3 viewDir = normalize(view_position - fragment_position);
    vec3 reflectDir = reflect(-lightDirection, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), SPECULAR_INTESITY);

    return vec4(SPECULAR_STRENGTH * spec * SPECULAR_LIGHT, 1.0);
}

void main()
{
   vec3 norm = normalize(normal);
        vec3 lightDirection = normalize(LIGHT_POSITION - fragment_position);

        vec4 ambientLight = getAmbientLight();
        vec4 diffuseLight = getDiffuseLight(norm, lightDirection);
        vec4 specularLight = getSpecularLight(norm, lightDirection);
        vec4 light = ambientLight + diffuseLight + specularLight;

        vec4 outputColor = light * texture(object_texture, uv) * color;

        FragColor = vec4(outputColor.x, outputColor.y, outputColor.z, 0.75);
}
```

Slika 5.4 *Implementacija Phongovog modela osvjetljenja u programu za sjenčanje fragmenata*

Za simuliranje efekta valova na modelu vode, prilikom iscrtavanja samog modela koristi se poseban program za sjenčanje verteksa (engl. Vertex Shader). Efekt valova se dobiva dodavanjem naizgled nasumične vrijednosti y koordinati svakog verteksa ovisno o x i z koordinatama verteksa. U ovom radu koristi se kombinacija funkcije sinusa i kosinusa, gdje se koristeći vrijednost vremena simuliranog prostora kao parametar tih funkcija dobivaju valovi koji se mijenjaju kroz vrijeme.

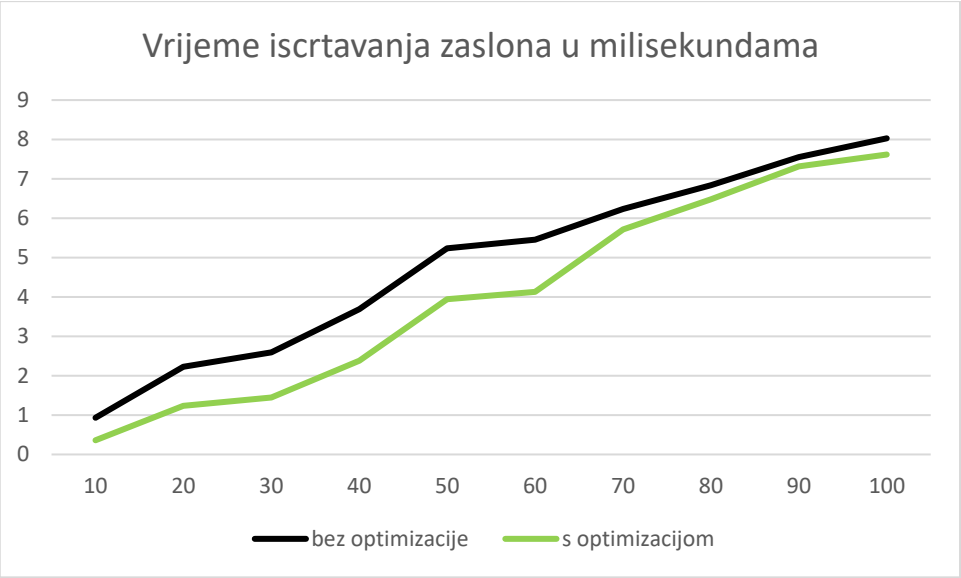


Slika 5.5 *Prikaz efekta valova na površini vode*

5.2. Optimiziranje modela isječaka

Prilikom iscrtavanja modela na zaslon, poželjno je dobiti što bolje performanse odnosno potrošiti što manje vremena iscrtavajući sve modele na zaslonu. U nastavku ovog poglavlja nabrojane su metode poboljšavanja performansi iscrtavanja uz koje se nalaze grafovi s prosječnim vremenima potrebnim za iscrtavanje određenog broja isječaka. Isječci na kojim je provedeno mjerenje su generirani u prostoru čije je početno sjeme broj 0.

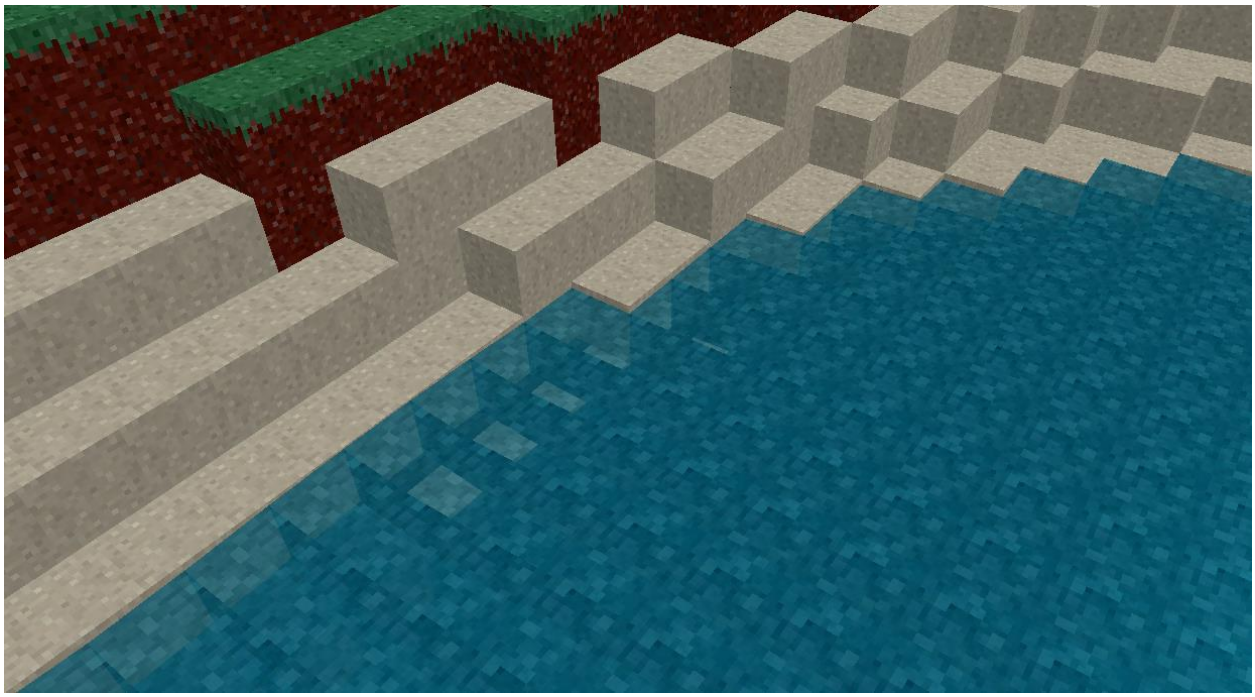
Prvi korak prilikom optimiziranja iscrtavanja 3D modela je uklanjanje stranica (engl. Face Culling) koje su okrenute suprotno od smjera gledanja kamere unutar prostora. Na taj način se osigurava da iscrtavanje modela ne troši vrijeme iscrtavajući stranice koje nisu vidljive.



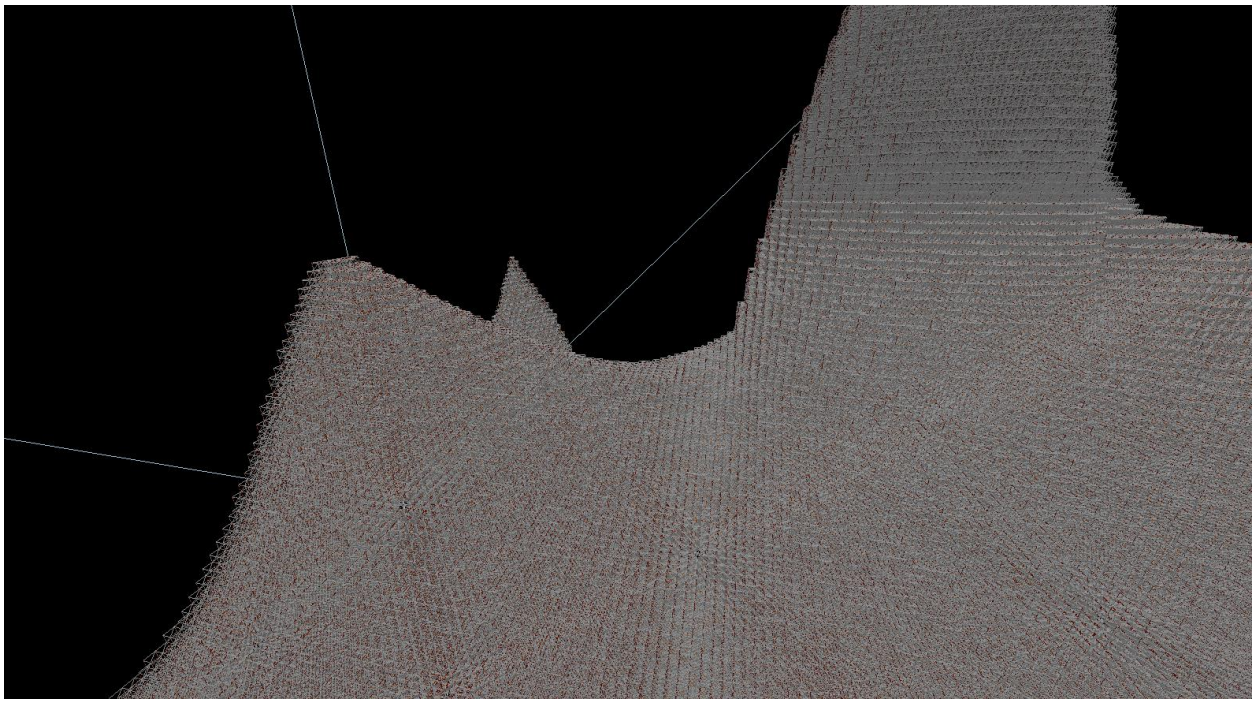
Slika 5.6 *Graf odnosa vremena iscrtavanja isječaka na zaslon u odnosu na broj učitanih isječaka*

Glavna metoda za poboljšavanje performansi iscrtavanja 3D prostora je smanjivanje broja stranica modela, odnosno trokuta koji čine tu stranicu, koje je potrebno iscrtati. Kako se u ovoj aplikaciji radi o proceduralno generiranom terenu, nije moguće unaprijed znati koliko kocaka će biti potrebno iscrtati što predstavlja izvor problema vezanih za performanse prilikom iscrtavanja. Svaki isječak generiranog terena je unikatan te ukoliko se nalazi planinski teren u određenom isječku, on može biti većinski popunjen kockama, a u najgorem slučaju i cijeli popunjen. Jedino područje koje preostaje kao mogućnost optimiziranja iscrtavanja je generiranje modela isječka.

Glavni pristup generiranju modela isječka je grupiranje modela kocaka za svaku kocku pojedinog isječka u jedan veliki model. Iako se čini kao korektan pristup generiranju modela, vrlo je neučinkovit i netočan. Ukoliko je isječak cijeli popunjen kockama, broj potrebnih stranica, odnosno trokutova koje je potrebno iscrtati, prelazi preko 100 000 što čini iscrtavanje planinskog područja vrlo sporim. Osim toga, ukoliko se dvije neprozirne kocke nalaze jedna pokraj druge, njihove susjedne stranice nisu vidljive te se bespotrebno traži iscrtavanje tih stranica. Ovaj problem se proteže kroz sve susjedne stranice svih kocaka unutar jednog isječka. Nadalje, iscrtavanje kocaka vode daje vizualno netočne rezultate. Model vode unutar isječka treba predstavljati model jednog kontinuiranog tijela, a kako se radi o prozirnim modelima, čest je problem netočnog i nekonzistentnog iscrtavanja prozirnih stranica jedne preko druge.



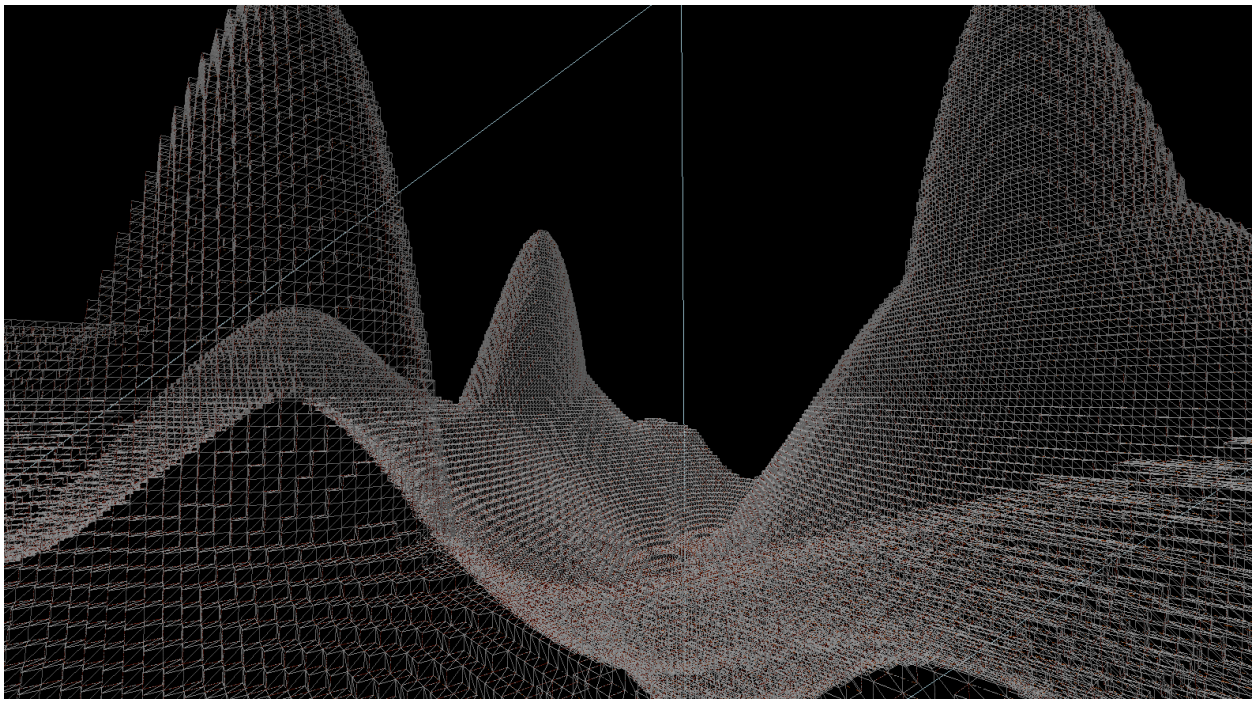
Slika 5.7 *Primjer netočno iscrtane površine vode*



Slika 5.8 *Prikaz okvira svih stranica kocaka koje se iscrtavaju bez upotrebe optimizacija modela*

Bolji pristup generiranju modela isječka je grupiranje modela kocaka isječka uz provjeru susjednih kocaka. Provjeravanjem susjednih kocaka prilikom generiranja modela isječka omogućuje izbacivanje stranica dvaju susjednih neprozirnih kocaka koje se ne vide. Na taj način

se smanjuje broj trokutova koje je potrebno iscrtati te tereni koji predstavljaju planinska područja postaju lakši za iscrtati, odnosno brže se iscrtavaju. Osim toga, ukoliko se generira model za kocke vode, moguće je izbjeći vizualno netočne rezultate na način da se ne iscrtava stranica vode koja je susjedna drugoj stranici vode što kao rezultat daje iscrtavanje samo površinskih stranica kocaka vode te onih koje su izložene drugim prozirnim kockama. Dodatno poboljšanje na prethodno objašnjen pristup je ne iscrtavanje stranica kocaka koje se nalaze na rubovima isječaka koji se nalaze na krajevima učitanih isječaka prostora. Iscrtavanje tih stranica nije potrebno jer se nikada ne vide te vizualno njihova odsutnost ne čini razliku u krajnjem prikazu prostora. [5]

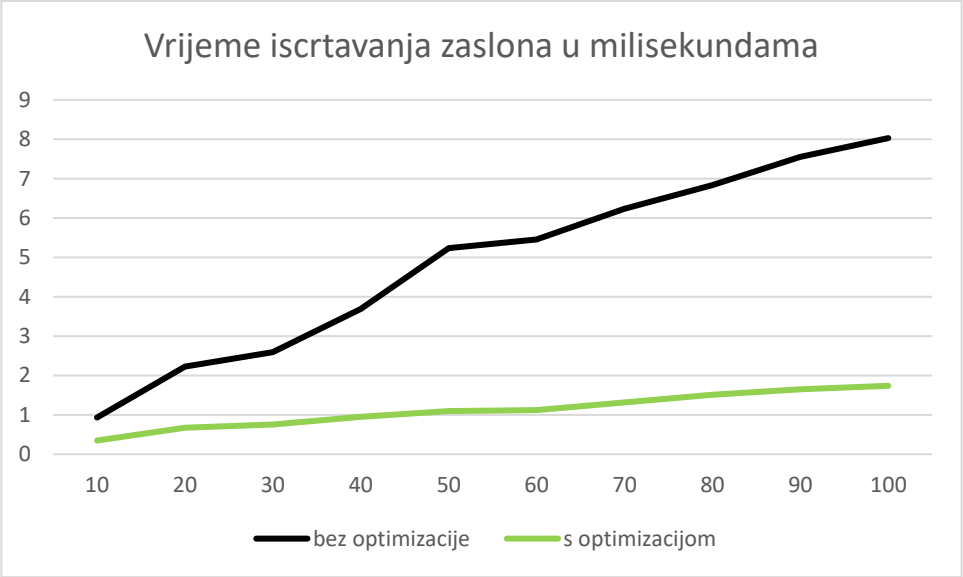


Slika 5.9 *Prikaz okvira svih stranica kocaka koje se iscrtavaju nakon primjene optimizacije modela*

Jedan od problema koji nastaje prilikom implementiranja prethodnog poboljšanja je nedostatak naizgled nasumičnih stranica kocaka na granicama isječaka učitanog terena. Kako se prostor proceduralno generira na način da se na jednom skupu niti generira teren, a na drugom skupu niti se generiraju modeli isječaka terena, nije zagarantiran redoslijed generiranja isječaka te prilikom provjere susjednih kocaka tijekom dodavanja stranica kocaka u završni model isječka nedostaju informacije o susjednim kockama. Kako bi se izbjegao nedostatak određenih stranica kocaka na rubovima isječka, prilikom dodavanja novog isječka u popis svih isječaka aplikacija

zahtjeva ponovno generiranje modela za susjedne isječke kako bi se dobio točan model susjednih isječaka ovisno o njihovim susjednim isječcima.

Uporabom prethodno navedenih optimizacija modela isječaka dobiju se znatna poboljšanja u performansama zbog manjeg broja stranica koje je potrebno iscrtati. Osim što se na taj način modeli mogu brže iscrtati, smanjenjem broja stranica modela se smanjuje i veličina memorije koju model zauzima u radnoj memoriji što omogućava prikaz većeg broja isječaka na zaslon i iscrtavanje prostora većih dimenzija.



Slika 5.10. *Graf odnosa vremena iscrtavanja isječaka na zaslon u odnosu na broj učitanih isječaka bez i s optimizacijom modela isječaka*

5.1. Poboljšavanje performansi iscrtavanja

Kako bi se dodatno poboljšale performanse iscrtavanja Voxel prostora, moguće je implementirati skrivanje isječaka koji se nalaze van okvira prozora unutar kojeg se iscrtava (engl. Face Culling). Korištenjem ovakve vrste skrivanja, nije potrebno pozivati funkcije za iscrtavanje modela onih isječaka koji se nalaze iza kamere unutar prostora što može značajno skratiti vrijeme iscrtavanja jednog zaslona.

Klasičan pristup iscrtavanja modela isječaka voxel prostora omogućava implementiranje jednostavnog iscrtavanja terena na zaslon. Ukoliko se želi dodati realistično osvjetljenje unutar prostora na način da teren može bacati sjenu na ostatak terena ukoliko se promjeni položaj izvora svjetlosti, tada dolazi do vizualnih problema prilikom implementacije. Kako bi se stvorio efekt sjena unutar prostora, potrebno je puno više resursa i računanja prilikom iscrtavanja. Implementacije sjena također zahtijevaju puno više resursa ako ih je potrebno napraviti što realističnijima. Dodavanjem sjena i iscrtavanjem velikog prostora koji sadrži veliki broj voxela se povećava vrijeme potrebno za iscrtavanje prostora na čemu se gube performanse i dobiva lošije iskustvo. Način iscrtavanja koji ne ovisi o broju voxela u prostoru i o kvaliteti sjena unutar prostora je iscrtavanje praćenjem svjetlosnih zraka (engl. Ray Tracing). Koristeći Ray Tracing, za svaki piksel prozora unutar kojeg se iscrtava aplikacija potrebno je pratiti na koji način se svjetlosti odbija iz svakog piksela na zaslonu prema ostatku prostora. Ovakav način iscrtavanja omogućuje laganu i brzu implementaciju sjena u prostoru, ne iscrtava nevidljive stranice i dijelove prostora te omogućava dodatne efekte poput refleksije svjetla o teren ili refrakcije svjetla u vodi. Iako ovaj pristup omogućava bolje vizualno iskustvo, zahtjeva veću računalnu snagu i hardversku podršku na grafičkoj kartici za računanje loma svjetlosti ukoliko se želi implementirati iscrtavanje prostora u realnom vremenu.

6. ZAKLJUČAK

Generiranje i iscrtavanje Voxel prostora zahtjeva dobru arhitekturu aplikacije i kompleksne tehnike optimizacije prilikom generiranja modela za iscrtavanje. Generiranje trodimenzionalnog terena zahtjeva podjelu terena u manje isječke kako bi rukovanje isječcima u memoriji bilo lakše, a sam postupak definiranja izgleda terena isječaka zahtjeva naprednije korištenje računalnih resursa korištenjem višenitnog programiranja kako bi se aplikacija mogla izvoditi u realnom vremenu bez blokiranja. Potrebno je kreirati dobar model komunikacije između niti koje odrađuju posao generiranja terena prostora i modela kako bi se osigurao sinkroni pristup dijeljenoj memoriji, smanjilo vrijeme potrebno na komunikacije između niti te kako bi se prevenirao zastoj više niti. Za dobivanje određenog oblika terena koriste se razni generatori šuma i matematičke funkcije čiji se izlazi preslikavaju u tražene oblike terena. Za dobre performanse prilikom iscrtavanja voxel prostora, potrebno je optimizirati modele isječaka na način da se smanji broj trokutova koje čine sami model koristeći razne tehnike uklanjanja nepotrebnih trokutova poput Face Culling-a i uklanjanja nevidljivih stranica kocaka.

LITERATURA

[1] A. Zucconi, The World Generation of Minecraft, 5.6.2022., dostupno na: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation [15.9.2023.]

[2] S. Wittens, Teardown Frame Teardown, 24.1.2023., dostupno na: https://acko.net/blog/teardown-frame-teardown/ [15.9.2023]

[3] Core Language (GLSL), The Kronos Group Inc., 19.3.2011., dostupno na: https://www.khronos.org/opengl/wiki/Core_Language_(GLSL) [16.9.2023.]

[4] Jakob Jenkov, Race Conditions and Critical Sections, 28.10.2020., dostupno na: https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html [17.8.2023.]

[5] Mikola Lysenko, Meshing in a Minecraft Game, 30.6.2012., dostupno na: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ [22.8.2023.]

[6] Joey de Vries, Basic Lighting, 27.9.2014., dostupno na: https://learnopengl.com/Lighting/Basic-Lighting [3.9.2023.]

SAŽETAK

Generiranje i iscrtavanje terena Voxel prostora zahtjeva rastavljanje prostora na manje isječke za lakše ažuriranje i manipuliranje prostorom. Kako su algoritmi za generiranje terena Voxel prostora i modela isječaka velike složenosti i računski zahtjevni, potrebno je strukturirati aplikaciju na način da se rastereti glavna nit koja iscrtava prostor prebacujući zadatak generiranja terena prostora i modela isječaka na druge niti kako bi se osigurale dobre performanse aplikacije. Pri tome je važno osmisliti model komunikacije niti koji osigurava sigurni i sinkroni pristup podacima o isječcima. Generiranje realističnog terena Voxel prostora je moguće koristeći generatore 2D i 3D šumova te preslikavajući njihove izlazne vrijednosti koristeći dodatne uvijete ili linearne interpolatore za dodatno oblikovanje terena. Koristeći generatore šuma moguće je generirati špilje unutar terena te ukrasiti isječke postavljajući dekoracije čiji položaj ovisi o izlazu generatora šuma. Zbog proceduralno generiranog terena koji se razlikuje za svaki isječak, potrebno je optimizirati modele isječaka koji se iscrtavaju na zaslon na način da se uklanjaju stranice kocaka koje nisu vidljive virtualnoj kameri unutar prostora čime se smanjuje vrijeme potrebno za iscrtavanje cijelog prostora na zaslon te omogućuje učitavanje većeg broja isječaka u memoriju i iscrtavanje istih. Za prilagođavanje izgleda vodenih površina u prostoru koriste se prilagođeni programi za sjenčanje verteksa i fragmenata koji omogućuje dodavanje efekata valova i efekta refleksije izvora svjetlosti na površini.

Ključne riječi: generiranje, iscrtavanje, optimiziranje, šum,

ABSTRACT

Generating and rendering Voxel terrain requires splitting the Voxel space into smaller chunks which allows for easier terrain updating and manipulating. The algorithms used for generating Voxel space terrain and its model are complex and require lots of computing power. Structuring the application in a proper way is key to good performance of the application and is achieved by separating rendering from terrain and model generation into separate thread which require a good thread-communication model to ensure chunk data safety and synchronization. Genereating realistic Voxel terrain is posibble by using 2D and 3D noise generator and mapping their output using linear interpolators and additional conditions which can additionaly shape the terrain. Nosie generators are also used for generating caves inside the terrain and are also used to determine the placement of decorations. Because the terrain is procedurally generated, it is important to optimize chunk models by removing faces that are invisbile to the virtual camera inside the Voxel space. This ensures lower frame drawing time and allows for more chunks to be loaded into the memory and rendered. For modifying the appearance of water bodies across the Voxel terrain, custom Vertex and Fragment Shaders are used to create the effect of waves and light source reflection on the surface of the water.

Keywords: generating, noise, optimizing, rendering, Voxel