SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6643

PRIKAZ TERENA ALGORITMOM POKRETNE KOCKE

Danijel Bajlo

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6643

PRIKAZ TERENA ALGORITMOM POKRETNE KOCKE

Danijel Bajlo

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6643

Pristupnik:	**Danijel Bajlo (0036505511)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak: **Prikaz terena algoritmom pokretne kocke**

Opis zadatka:

Proučiti prostore zadane volumnim elementima (engl. voxel) te mogućnost generiranja ovakvog prostora Perlinovim šumom. Proučiti algoritam pokretne kocke. Razraditi postupak generiranja scene Perlonovim šumom te prikaza prostora primjenom algoritma pokretne kocke. Razmotriti mogućnosti implementacije algoritma na grafičkom procesoru. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafički pogon Unity i programski jezik C#. 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: 12. lipnja 2020.

Sadržaj

Uvod

Proceduralno generiranje je postupak kojim se mogu stvarati slučajne varijacije podataka u razne svrhe. U prošlosti, proceduralno generiranje koristilo se uglavnom zato što je štedjelo memorijski prostor, jer se na temelju malenog skupa podataka moglo generirati puno veći broj raznovrsnih objekata i prostora. S padom cijene memorije, taj problem se smanjio, ali divovski, otvoreni svjetovi postali su standard modernih videoigara. Brojne igre trude se igraču dati veliki stupanj slobode i stvoriti realistične krajolike, a interaktivnost s okruženjem jedan je od preduvjeta. S veličinom virtualnog svijeta raste i vrijeme te cijena njegove izrade. Iz tog razloga proceduralna generacija vrlo je zastupljena metoda efikasnog stvaranja neiscrpnog broja objekata, atributa, svjetova i bilo čega zamislivog. Koristeći proceduralnu generaciju, dizajneri videoigara mogu u vrlo kratkom vremenu stvoriti na desetke pa čak i stotine raznovrsnih objekata. Neke od prednosti proceduralne generacije su manje datoteke te velike količine sadržaja i varijacija. Osim za videoigre, proceduralna generacija koristi se i u industriji specijalnih efekata za stvaranje scena s velikim brojem sličnih objekata.

U ovom radu predstaviti će se osnovne ideje i implementacije proceduralne generacije terena u programskom alatu Unity. Prvo na jednostavnom primjeru terena prikazanog volumnim elementima, a potom na sofisticiranijem primjeru terena prikazanog algoritmom pokretne kocke. Prikazat će se mogućnost stvaranja terena korištenjem dvodimenzionalnog i trodimenzionalnog šuma.

U sklopu rada izrađen je i jednostavan uređivač terena koji omogućava dinamičku izmjenu terena u sve tri dimenzije, dodavanjem ili oduzimanjem materijala. Napisan je i jednostavni triplanarni sjenčar za uljepšavanje prikaza terena.

Rad će se i dotaknuti teme ubrzanja algoritma implementacijom na grafičkoj kartici korištenjem sjenčara računanja (engl. *Compute shader*).

1. Osnove proceduralne generacije

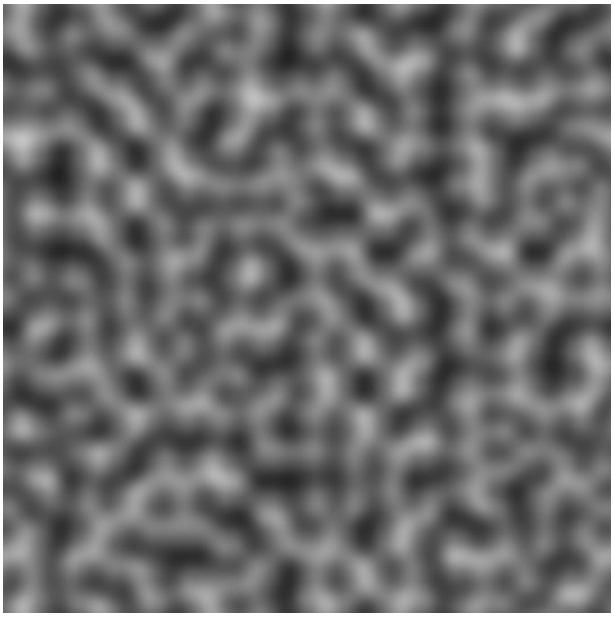
Proceduralna generacija postupak je generiranja podataka algoritamski, umjesto ručno. [1] U tu svrhu koriste se pseudo-slučajni izvori informacija u kombinaciji s unaprijed ručno napravljenim objektima. Ideja proceduralne generacije nije da bude potpuno nasumična, već da stvara iskoristive varijacije unutar ograničenja postavljenih od strane programera.

U ovom radu, pojednostavljeni cjevovod za generaciju terena prikazuje Slika 1.1. Sastoji se od generacije pseudo-slučajnih brojeva, manipulacije i interpretacije tih brojeva te prikaza interpretacije. Kao izvor brojeva korišten je Perlinov šum. Manipulacija brojeva svodi se na njihovo skaliranje i izmjenu graničnih vrijednosti. Interpretacija iskorištava te brojeve u kontekstu visine ili gustoće, o čemu će više biti rečeno u poglavljima 2.2 i 3.2. Prikaz odgovara iscrtavanju terena određenog interpretacijom podataka, u prostoru.

Slika 1.1. - Pojednostavljeni cjevovod generacije terena

1.1 Perlinov šum

Perlinov šum vrsta je gradijentnog šuma kojeg je smislio Ken Perlin 1983. Naziv gradijentni dolazi od činjenice da se koriste nasumično generirani gradijentni vektori kako bi se stvorio šum. Za razliku od npr. Bijelog šuma, koji je potpuno nasumičan, Perlinov šum ima glatke prijelaze između vrijednosti. To svojstvo može se vizualizirati ako se vrijednost šuma interpretira kao intenzitet svjetlosti slikovnog elementa. Tada se dobije tekstura koju prikazuje Slika 1.2. [2]



Slika 1.2. - 2D tekstura Perlinovog šuma

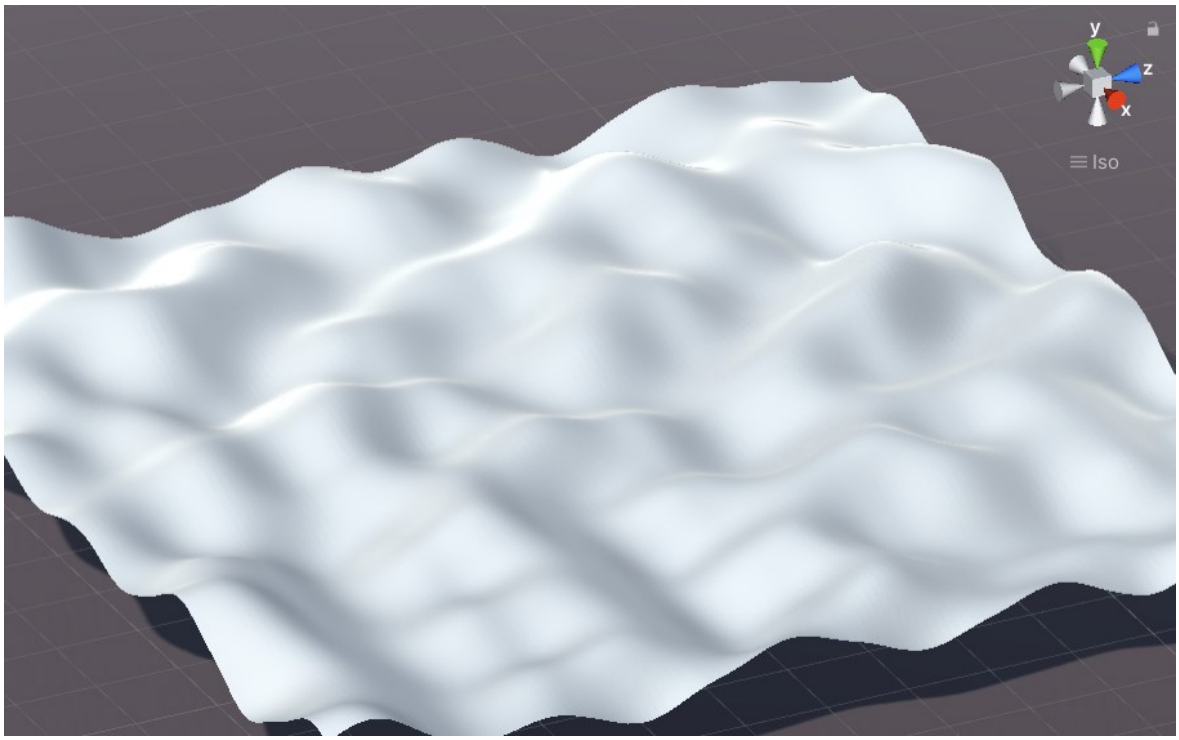
Iako se danas u računalnoj grafici češće koriste noviji, poboljšani šumovi, poput *simplex noise* i *fractal noise*, koristio sam Perlinov šum jer je on ugrađen u programski alat Unity, pa ga je izrazito lako koristiti.

U alatu Unity postoji funkcija *PerlinNoise*, koja kao argument prima 2 realna broja. Ti brojevi mogu se interpretirati kao x i y koordinate na 2D teksturi Perlinovog šuma. Vrijednosti koje vraća funkcija su realni brojevi u rasponu [0, 1], iako dokumentacija spominje da vrijednosti mogu malo izaći izvan tih granica. [3] Radi jednostavnosti u ostatku rada pretpostavljat će se da je vrijednosti ne izlaze iz tih granica. Za potrebe generiranja 3D terena koji neće biti samo visinska mapa, potreban je 3D šum. Iako Unity nema ugrađeni takav šum, jednostavan način da ga se stvori jest da se izračunaju dvije vrijednosti 2D šuma, npr. za koordinate X, Y i X, Z te se vrati njihova aritmetička sredina. To se može interpretirati kao korištenje dviju okomitih ploha s teksturama šuma za generiranja vrijednosti trodimenzionalne točke.

Bitno je spomenuti da funkcija Perlinovog šuma u alatu Unity ne radi dobro s cjelobrojnim vrijednostima, tako da je potrebno pretvoriti cjelobrojne koordinate u zapis s pomičnim zarezom.

1.2 Korištenje šuma za generiranje podataka o terenu

Dvije su osnovne interpretacije brojčane vrijednosti šuma kod generacije terena: visina i gustoća. Korištenje 2D teksture šuma kao visinske mape svodi se na postavljanje Y koordinate svakog vrha mreže na vrijednost koja odgovara vrijednosti šuma u točki X, Z. Množenjem te vrijednosti nekom konstantom skalira se amplituda terena. Primjer takve mreže prikazuje Slika 1.3.

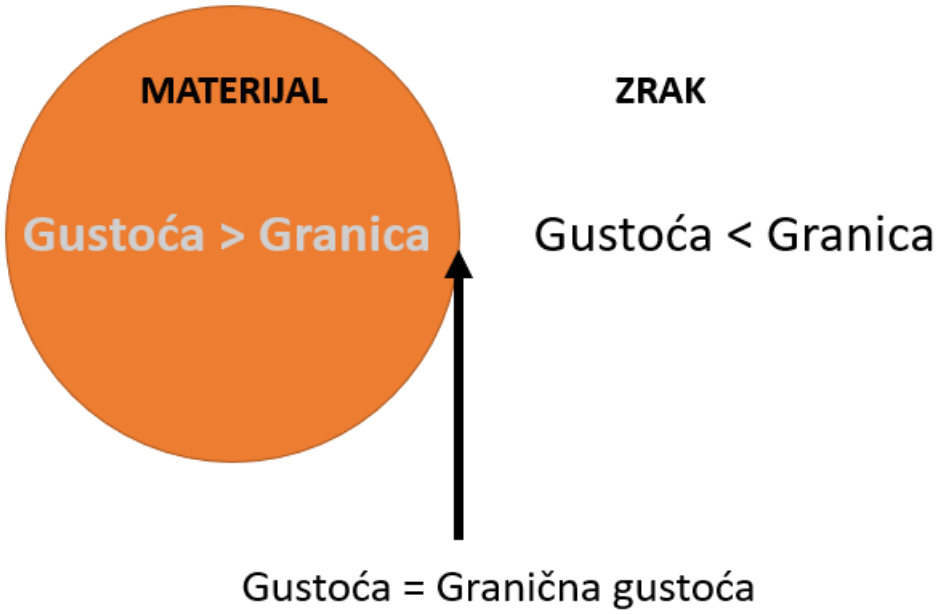


Slika 1.3. Mreža koja koristi Perlinov šum kao visinsku mapu

U nekim slučajevima, ovaj pristup je dovoljno dobar za stvaranje terena, ali ima jedan veliki problem: svaka točka na mreži može imati samo jednu Y vrijednost, stoga je nemoguće generirati ili izmijeniti teren tako da se dobiju konkavni oblici, poput pećine ili tunela.

Drugi pristup je interpretacija vrijednosti šuma kao gustoće. Svakoj točki u 3D prostoru potrebno je dodijeliti vrijednost koja predstavlja gustoću. To nije gustoća u znanstvenom smislu omjera mase i volumena, već mjera koliko je ta točka duboko ispod površine, što ilustrira Slika 1.4. Tada je potrebno usporediti gustoću svake točke s proizvoljnom graničnom gustoćom i time odrediti je li ta točka unutar ili

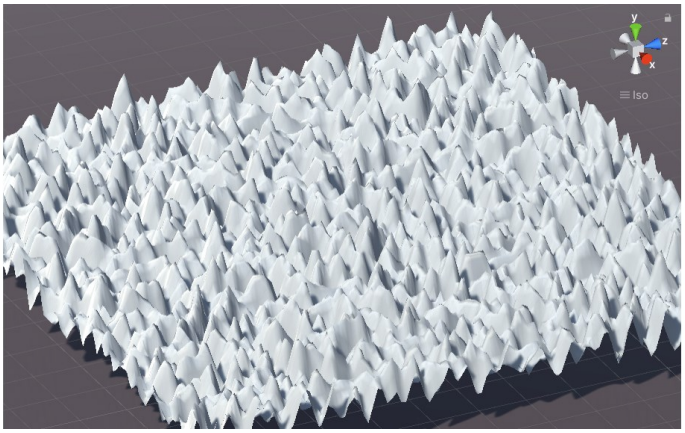
izvan terena. Te informacije se kasnije koriste različito ovisno o načinu na koji se prikazuje teren.



Slika 1.4. Ideja prikaza terena na temelju gustoće

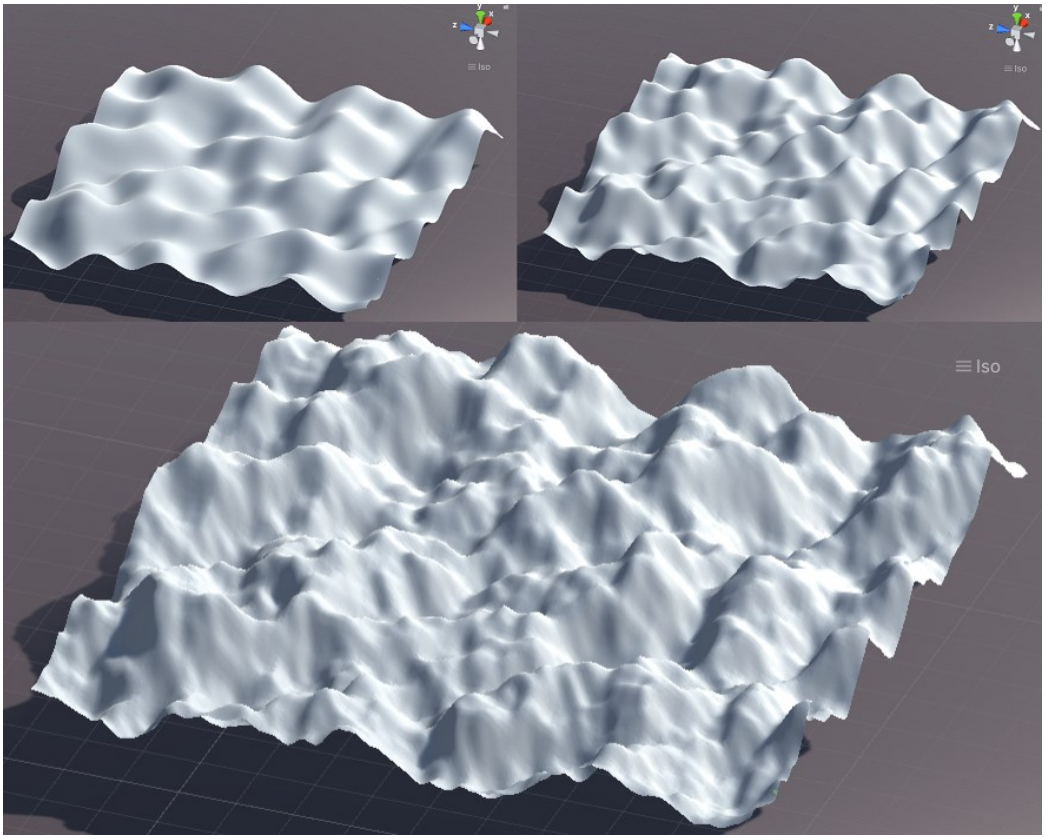
1.3 Dodavanje detalja korištenjem oktava

Iako je 2D tekstura Perlinovog šuma beskonačno velika te stalno generira nove, pseudo-slučajne uzorke, svi ti uzorci izgledaju slično jer variraju sličnom frekvencijom među sličnim vrijednostima. To se najbolje vidi na velikim terenima, koji izgledaju kao da se stalno ponavlja isti uzorak, umjesto da je nasumično generirano. Takav teren prikazuje Slika 1.5. Iako su sve vrijednosti nasumične, teren izgleda dosta monotono i dosadno.



Slika 1.5. Veliki teren s mnogo vrhova

Taj problem može se riješiti dodavanjem viših oktava šuma, koje će imati veću frekvenciju, a manju amplitudu. Prvu oktavu možemo zamisliti kao onu koja generira planine i doline. Druga oktava tada oblikuje vrhove planine i stvara brjegove, treća stvara velika kamenja, četvrta stvara šljunak itd. Način na koji se oktave mogu implementirati u kodu je vrlo jednostavan. Za svaku točku ponavlja se petlja u kojoj se računa vrijednost Perlinovog šuma sa zadanom frekvencijom. Ta vrijednosti pomnoži se s amplitudom i pribroji prethodno izračunatoj vrijednosti šuma u toj točki. Potom se vrijednosti amplitude i frekvencije skaliraju kako bi u slijedećoj iteraciji odgovarali višoj oktavi. Tako se amplituda množi s pozitivnom vrijednosti manjom od 1, a frekvencija s vrijednosti većom od 1. Te vrijednosti su proizvoljne i ovise o kontekstu u kojem se koriste i osobnoj preferenciji. Slika 1.6 prikazuje teren generiran korištenjem viših oktava.

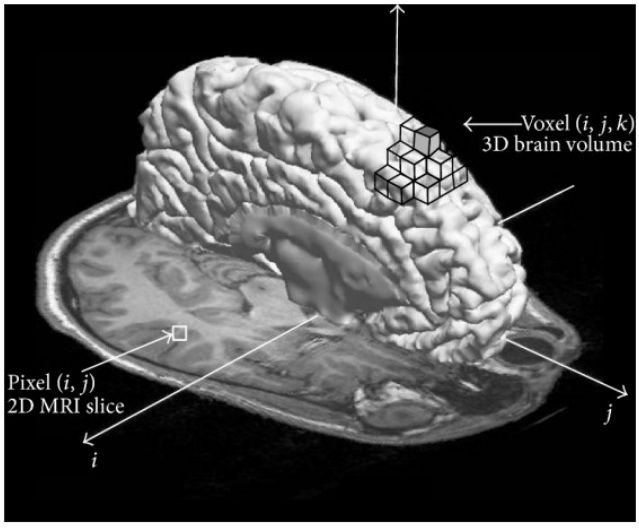


Slika 1.6. Teren generiran s 1, 2 i 4 oktave

2. Prikaz terena volumnim elementima

2.1 Općenito o volumnim elementima

Volumni elementi (engl. *Voxel*) predstavljaju vrijednosti u regularnoj 3D koordinatnoj mreži. Oni su 3D analogija dvodimenzionalnih slikovnih elemenata (engl. *Pixel*). Sami elementi nemaju eksplicitno pohranjenu poziciju u svojoj vrijednosti, već je ona određena relativno prema poziciji ostalih elemenata. [4] Volumni elementi često se koriste za prikazivanje medicinskih i znanstvenih podataka te su osnovni element slike kod volumetrijskih prikaza. Korištenje volumnih elemenata za vizualizaciju podataka magnetske rezonancije prikazuje Slika 2.1. [5] U videoigrama često se koriste kao alternativa visinskim mapama zbog mogućnosti jednostavnog prikaza konkavnih terena. Glavna prednost volumnih elemenata u odnosu na algoritam pokretne kocke je jednostavnost implementacije i izračunavanja. S druge strane, najveći nedostatak je nemogućnost izglađivanja terena, iz čega slijedi taj karakteristični „kockasti“ izgled pri nižoj frekvenciji uzorkovanja.



Slika 2.1. Snimka ljudskog mozga prikazana volumnim elementima

2.2 Prikaz terena generiranog Perlinovim šumom

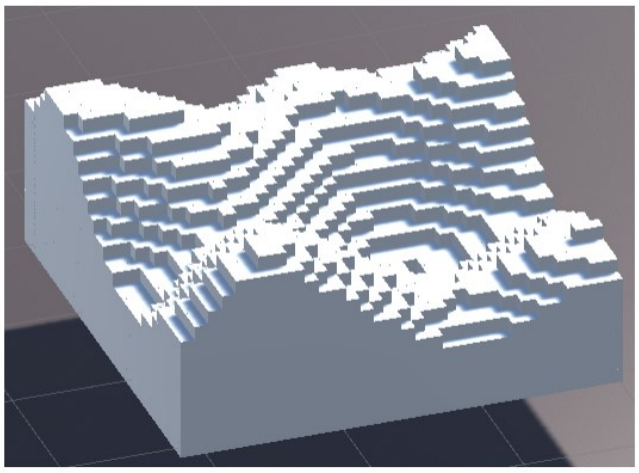
U prvom primjeru vrijednosti Perlinovog šuma interpretirat će se u kontekstu visine, kao u prethodnom poglavlju. U drugom primjeru koristit će se 3D Perlinov šum interpretiran kao gustoća, za generiranje terena nalik na sustav špilja.

2.2.1 Perlinov šum kao visinska mapa

Za razliku od primjera iz prošlog poglavlja, gdje je visina svakog vrha mreže bila postavljena na iznos određen šumom, kod volumnih elemenata potrebno je stvoriti element na svakoj točki koja je ispod razine terena. Algoritam kojim se to ostvaruje glasi:

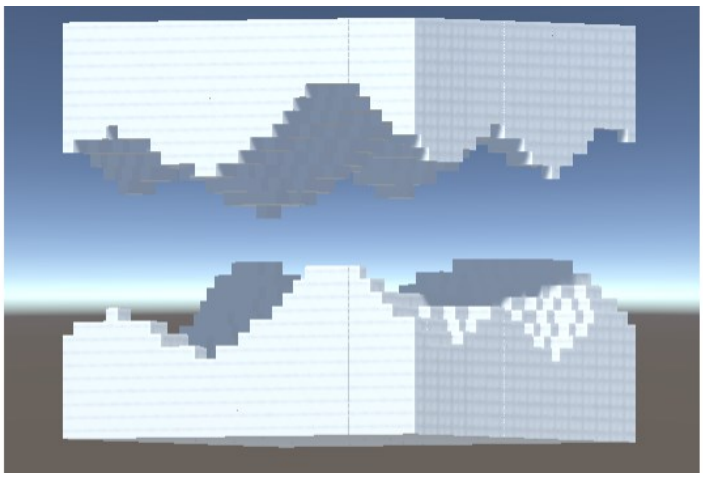
1. Iterirati po svim vrijednostima X, Y i Z koordinate u zadanom rasponu
2. Za svaki X i Z potrebno je izračunati vrijednost Perlinovog šuma
3. Tu vrijednost, pomnoženu s proizvoljnom amplitudom, usporediti s Y koordinatom točke
4. Ako je točka ispod razine terena, na tom mjestu stvoriti instancu volumnog elementa

Teren koji se dobije ovim jednostavnim algoritmom prikazuje Slika 2.2. Udaljenost koordinata treba biti jednaka veličini volumnog elementa, kako ne bi došlo do preklapanja ili razmaka između elemenata. Također, s obzirom da će sve točke s istim X i Z koordinatama imati istu vrijednost šuma, moguće je unaprijed izračunati te vrijednosti kako se ne bi nepotrebno računale za svaki Y.



Slika 2.2. Visinska mapa prikazana volumnim elementima

Ovim pristupom zapravo se mogu stvoriti tereni koji nisu samo visinske mape, ali nepraktično je, jer programer mora ručno određivati granice unutar kojih će se nalaziti teren. U primjeru koji prikazuje Slika 2.3. volumnim elementima prikazane su koordinate koje se nalaze ispod, ali i iznad neke visine određene Perlinovim šumom. Moguće je napisati mnogo pravila i time stvoriti zanimljivi teren, no očito je da taj pristup nije jednostavno izmijeniti i nadograditi. Čitava bit proceduralne generacije je omogućiti beskonačne varijacije s minimalnim skupom pravila, a ovdje je broj varijacija proporcionalan broju pravila.



Slika 2.3. Prikaz blokova iznad i ispod neke visine

Kada bi se koristio još jedan šum koji bi određivao ta pravila za nas, dobili bismo beskonačan broj varijacija u 3D prostoru. Upravo na toj ideji bazira se sljedeći primjer.

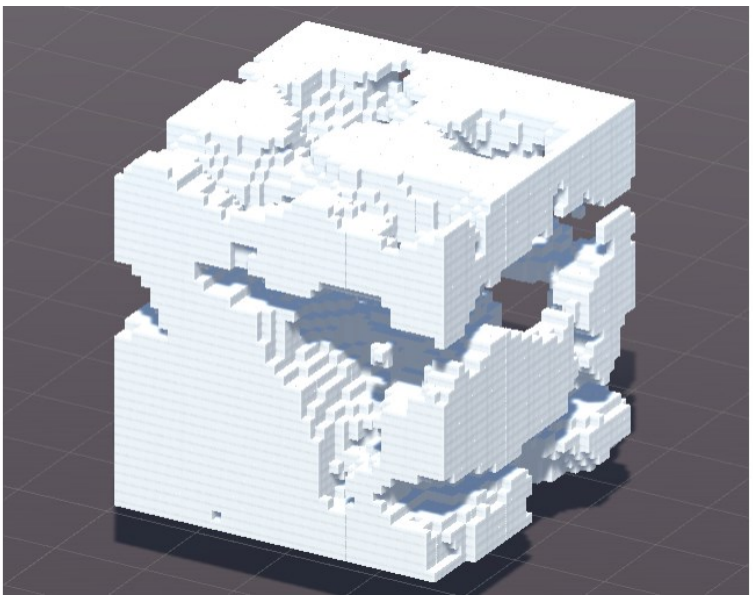
2.2.2 Perlinov šum kao gustoća terena

U prošlom primjeru svaki vrh imao je tri koordinate od kojih je jedna ovisila o vrijednosti šuma. Samo vrhovi čija je Y koordinata bila ispod granične razine terena određene šumom bili su prikazani. Tu ideju, uz manje izmjene, moguće je preslikati u višu dimenziju. Moguće je svakoj točki pridijelili vrijednost na temelju svih triju koordinata, te na temelju usporedbe vrijednosti s graničnom, odrediti hoće li točka biti prikazana. Ta vrijednosti je gustoća. Ovaj primjer se može zamisliti kao projekcija dijela 4D terena kojem je četvrta koordinata manja od vrijednost granične gustoće, u trodimenzionalni prostor.

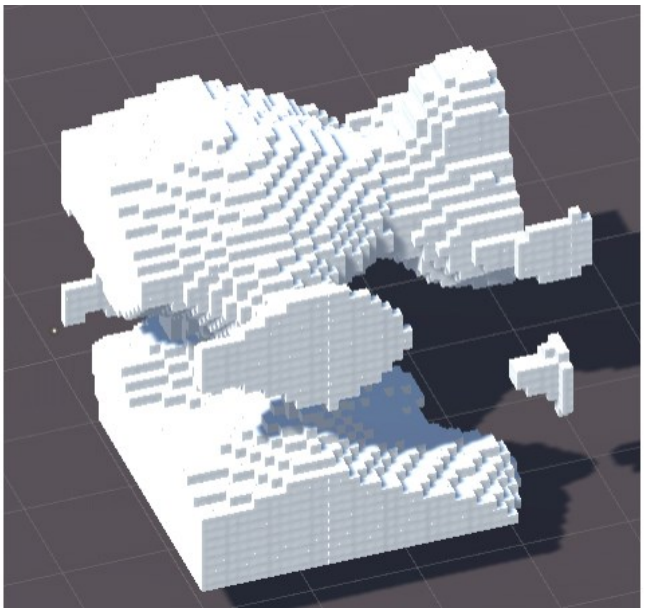
Algoritam za prikaz terena interpretacijom šuma kao gustoće glasi:

1. Za svaku koordinatu X, Y i Z izračunati vrijednost šuma
2. Usporediti vrijednost s graničnom
3. Ako je manja od granične, stvoriti instancu volumnog elementa na tim koordinatama

Primjer terena generiranog i prikazanog tim algoritmom ilustrira Slika 2.4. U tom primjeru granična gustoća postavljena je točno na 0.5, što je polovina raspona vrijednosti šuma pa će u prosjeku pola točaka biti prikazano. Kada se vrijednost granične gustoće smanji, manje točaka bude prikazano pa se dobije pregledniji teren, kakav prikazuje Slika 2.5.



Slika 2.4. Teren generiran 3D Perlinovim šumom



Slika 2.5. Teren kod kojeg je granična gustoća 0.4

U oba algoritma prikazana u ovom poglavlju navedeno je da se na točki koja zadovoljava uvjet stvori instanca volumnog elementa. Taj pristup je izrazito neefikasan, jer većina objekata nije uopće vidljiva. Primjeri na prijašnjim slikama su dimenzija 38x38x38. Ako pretpostavimo da su u prvom slučaju stvoreni volumni elementi na polovini točaka, tada na sceni postoji 27436 instance objekta kocke,

što znatno utječe na performanse. Ovaj problem može se riješiti spajanjem njihovih mreža u zajedničku mrežu, ali taj dio nije obrađen u ovom radu.

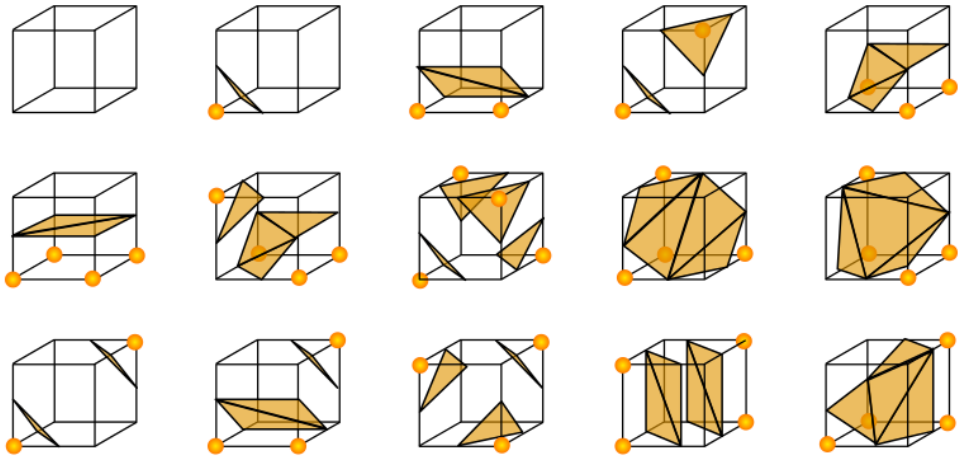
3. Prikaz terena algoritmom pokretne kocke

U prošlom poglavlju predstavljen je način prikaza terena korištenjem volumnih elemenata. Takvi tereni izgledaju diskretizirano i „kockasto“. To svojstvo osim estetskog značenja, također otežava kretanje po terenu, pogotovo vozilima poput automobila. Zato u nekim primjenama poželjno je imati glatki teren. Jedan od najčešćih načina ostvarivanja takvog terena je korištenje algoritma pokretne kocke.

3.1 Općenito o algoritmu

Algoritam pokretne kocke (engl. *Marc*hing *cubes algorithm*) razvili su William E. Lorensen i Harvey E. Cline s ciljem da efikasno vizualiziraju podatke dobivene magnetskom rezonancijom i računalnom tomografijom. [6]

Algoritam se temelji na ideji da se svakom od vrhova kocke dodijeli vrijednost na temelju koje se može odrediti nalazi li se vrh unutar ili izvan nekog volumena. Brid koji spaja susjedne vrhove od kojih se jedan nalazi unutar, a drugi izvan volumena tada mora biti presječen površinom samog volumena, jer ona prolazi između tih dviju točaka. Stoga je na tom bridu potrebno stvoriti novi vrh te na kraju spojiti novonastale vrhove u trokute. Svaki od osam vrhova kocke može biti unutar ili izvan volumena što znači da postoji 256 mogućih kombinacija. Uklanjanjem slučajeva koji su rotacijske i zrcalne simetrije jedni drugih, Lorensen i Cline došli su do originalnih 15 konfiguracija kocke koje prikazuje Slika 3.1. [7] Vrhovi prikazani narančastom bojom nalaze se ispod površine. Slučajevi u kojima su svih 8 vrhova ili ispod ili iznad površine su ekvivalentni, jer u oba slučaja nema presjeka između površine tijela i bridova kocke pa ne treba ništa prikazati.



Slika 3.1. Originalno objavljenih 15 konfiguracija kocke

3.2 Implementacija algoritma u alatu Unity

U primjeru implementacije koji sam koristio kao referencu, korišten je ranije spomenuti pristup interpretacije Perlinovog šuma kao visine. [8] To je rezultiralo terenom nalik na visinsku mapu. Zbog jednostavnosti prvo će implementacija biti ilustrirana na primjeru interpretacije Perlinovog šuma kao visine, a zatim će biti proširena na tri dimenzije.

3.2.1 Prikaz terena generiranog 2D Perlinovim šumom

Prva funkcija koja se poziva zadužena je za dodjeljivanje visine terena svakoj točki u XZ ravnini, isto kao u prijašnjim primjerima. Sljedeća funkcija `CreateMeshData` koju prikazuje Slika 3.2 grupira po osam susjednih točaka u jednu kocku i poziva provođenje algoritma pokretne kocke nad njima. Ta kocka je zapravo polje realnih vrijednosti koje odgovaraju razlici visine terena i Y koordinate te točke. Za dobivanje koordinata svih vrhova kocke koristi se pomoćna tablica `CornerTable` koja sadrži cjelobrojne vektore od [0 0 0] do [1 1 1].

```
void CreateMeshData()
{
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            for (int z = 0; z < width; z++)
            {
                float[] cube = new float[8];
                for(int i = 0; i < 8; i++)
                {
                    Vector3Int corner = new Vector3Int(x, y, z) + CornerTable[i];
                    cube[i] = corner.y - terrainMap[corner.x, corner.z];
                }
                MarchCube(new Vector3(x, y, z), cube);
            }
        }
    }

}
```

Slika 3.2. Funkcija CreateMeshData

Funkcija `MarchCube` (Slika 3.4) koristi pomoćnu funkciju `GetCubeConfig` (Slika 3.3) kako bi dobila konfiguracijski indeks kocke. Taj indeks koristi se za dobivanje konfiguracije kocke iz tablice trokuta (`Triangle table`), koja sadrži svih 256 konfiguracija. Primjer tablice dostupan je na referenci [9]. Svaki redak u tablici sadrži 15 elemenata koji predstavljaju indekse bridova koje siječe površina u danoj konfiguraciji. Na indeksiranim bridovima stvaraju se novi vrhovi iz kojih se gradi mreža.

```
int GetCubeConfig(float[] cube)
{
    int configIndex = 0;
    for(int i = 0; i < 8; i++)
    {
        if(cube[i] > 0)
        {
            configIndex |= 1 << i;
        }
    }
    return configIndex;
}
```

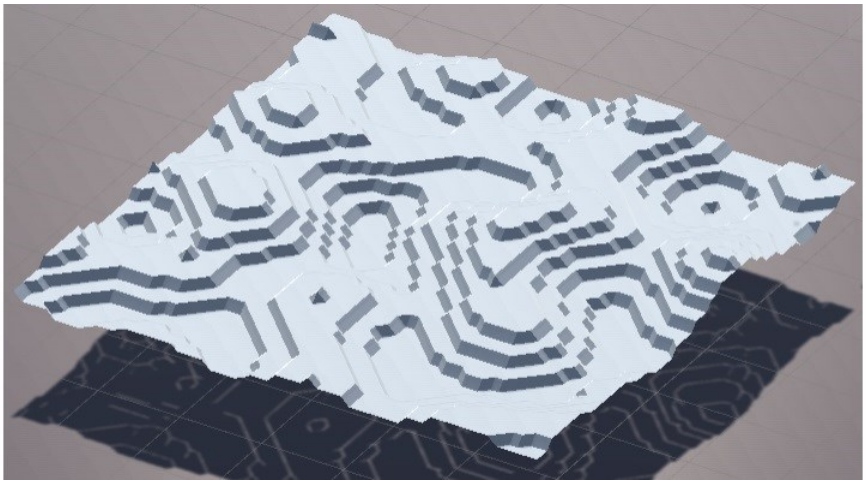
Slika 3.3. Funkcija koja određuje konfiguracijski indeks

Za svaki od 8 vrhova kocke provjeri se jesu li ispod ili iznad površine. Negativna vrijednost znači da je ispod, a pozitivna iznad, jer vrijednost koju uspoređujemo predstavlja razliku Y koordinate i razine terena u točki. Ono što slijedi u slučaju ispunjenog uvjeta je postavljanje bita na indeksu `i` u varijabli `configIndex` na 1. Kada se niz bitova interpretira kao dekadski broj, dobije se indeks konfiguracije u tablici trokuta. U ovom slučaju svi bitovi 0 znači da su svi vrhovi ispod površine, a svi bitovi 1 znači da su svi vrhovi u zraku.

```
void MarchCube(Vector3 position, float[] cube)
{
    int configIndex = GetCubeConfig(cube);
    int edgeIndex = 0;
    for(int i = 0; i < 5; i++)
    {
        for(int p = 0; p < 3; p++)
        {
            int index = TriangleTable[configIndex, edgeIndex];
            if(index == -1)
            {
                return;
            }
            Vector3 vert1 = position + EdgeTable[index, 0];
            Vector3 vert2 = position + EdgeTable[index, 1];
            Vector3 vertPosition = (vert1 + vert2) / 2;
            vertices.Add(vertPosition);
            triangles.Add(vertices.Count - 1);
            edgeIndex++;
        }
    }
}
```

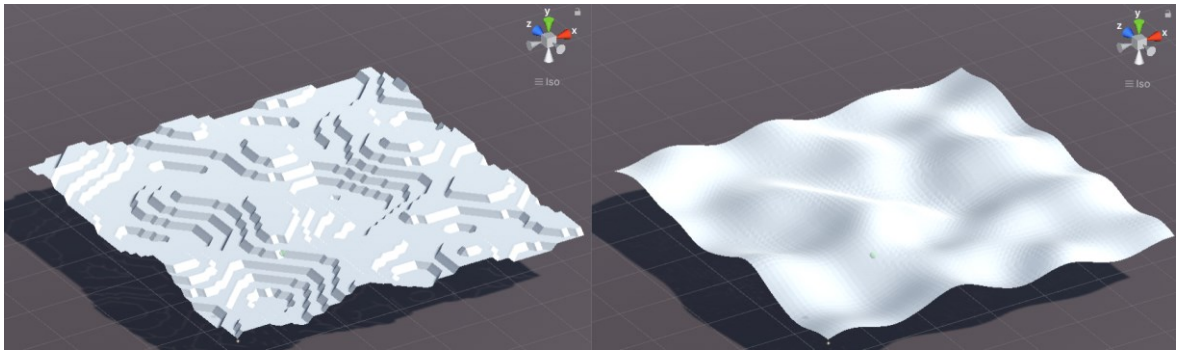
Slika 3.4. Funkcija koja provodi algoritam pokretne kocke

U funkciji `MarchCube` prva petlja ima pet iteracija jer u svakoj konfiguraciji kocke može biti najviše pet trokuta. Druga petlja ima tri iteracije i predstavlja tri vrha trokuta. Oni redci u tablici trokuta koji imaju manje od pet trokuta na preostalim poljima imaju vrijednosti -1, koja označava završetak. Na temelju indeksa brida i pozicije jednog vrha kocke dobivaju se koordinate vrhova koje spaja taj brid, uz pomoć tablice bridova `EdgeTable` u kojoj su pohranjeni cjelobrojni vektori. U ovom jednostavnom primjeru novi vrh mreže stvara se točno na polovini brida, zbog čega teren ima terasasti izgled. Naposlijetku, vrhovi se dodaju u listu vrhova mreže, a njihovi indeksi u listu trokuta. Liste se pretvore u polja i predaju mreži koja se potom iscrta na ekranu, što prikazuje Slika 3.5.



Slika 3.5. Teren generiran 2D šumom prikazan algoritmom pokretne kocke

Kada bi se, umjesto stvaranja vrhova mreže točno na polovini brida, napravila linearna interpolacija vrijednosti susjednih vrhova, dobiveni teren bio bi zaobljen. Usporedbu terasastog i zaobljenog terena prikazuje Slika 3.6.



Slika 3.6. Terasasti i zaobljeni teren

3.2.2 Prikaz terena generiranog 3D Perlinovim šumom

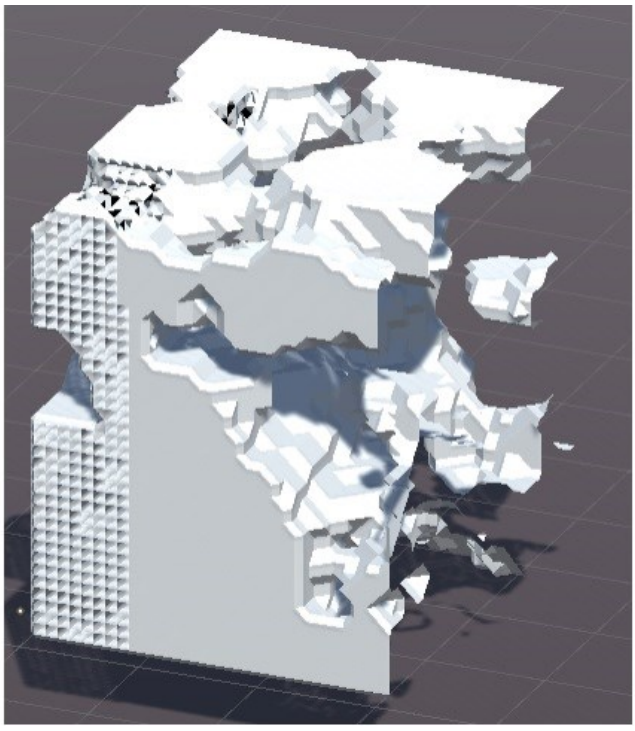
Prošli primjer stvorio je rezultate slične visinskoj mapi, što nije baš dobra ilustracija mogućnosti algoritma pokretne kocke. Kako bi se stvorio teren s konkavnim elementima, potrebno je izmijeniti funkciju koja računa vrijednost vrhova kocke. Slika 3.7. prikazuje funkciju korištenu za računanje gustoće vrha na koristeći 3D Perlinov šum.

```
public static float GetPointDensity(Vector3 coords, int octaves, float scale, float persistence,
    float lacunarity, Vector3 offset, int perlinNum)
{
    float density = 0;
    float amplitude = 1;
    float frequency = 1;
    float noise = 0;
    for (int i = 0; i < octaves; i++)
    {
        noise = Perlin3D(((float)coords.x + offset.x) / scale * frequency,
                         ((float)coords.y + offset.y) / scale * frequency,
                         ((float)coords.z + offset.z) / scale * frequency, perlinNum) * amplitude;
        if (i == 0)
        {
            density += noise;
        } else
        {
            //scale noise to range [-0.5A, 0.5A]
            density += noise - amplitude * 0.5f;
        }
        amplitude *= persistence;
        frequency *= lacunarity;
    }
    return density;
}
```

Slika 3.7. Funkcija za računanje gustoće vrha

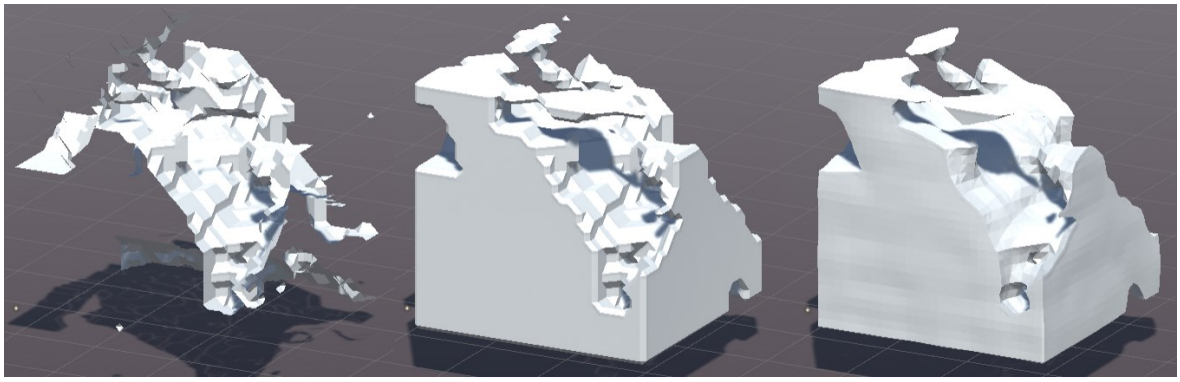
Funkcija podržava proizvoljan broj oktava i broj šumova koji će se koristiti za izračunavanje vrijednosti. Za prvu oktavu vrijednost šuma je u rasponu [0, 1], dok za više oktave se skalira na raspon [-0.5*A, 0.5*A] gdje je A amplituda te oktave. Razlog tomu jest taj da kad bi se samo dodavale pozitivne vrijednosti šuma, većina vrhova bi prešla graničnu vrijednost i takav teren bio bi samo zrak. Na ovaj način varijacije se stvaraju oko vrijednosti određenih prvom oktavom pa oblik terena ostaje prepoznatljiv, ali uz dodane detalje.

Ostale funkcije su identične onima iz prijašnjeg primjera, uz zamjenu visine terena gustoćom. Teren generiran 3D Perlinovim šumom prikazan algoritmom pokretne kocke prikazuje Slika 3.8. Na slici vidljiva je greška u prikazu terena uzrokovana prevelikim brojem vrhova. U alatu Unity mreža može imati najviše 65536 vrhova, jer koristi 16-bitne indekse. Iako prikazani teren ima dimenzije 38x38x38 za što je bilo očekivano da će rezultirati u najviše 54872 vrha, mreža se sastoji od 87816 vrhova. Razlog je u tome što se vrhovi mreže stvaraju na bridovima kocke pa će biti maksimalno 12 vrhova po kocki, a ne 8. Također, za svaku se kocku u algoritmu ponovno stvaraju vrhovi, što znači da susjedne kocke koje dijele bridove stvaraju višestruke vrhove koji se poklapaju.



Slika 3.8. Teren prikazan algoritmom pokretne kocke s previše vrhova

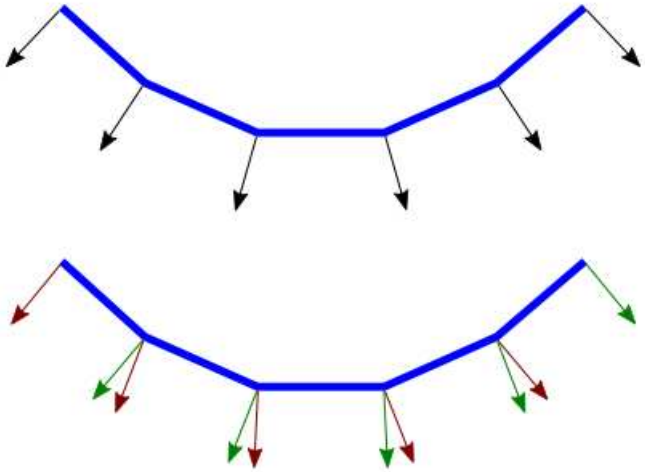
Uz smanjene dimenzije, broj vrhova mreže može se održati unutar granica te tada nema greške u prikazu. Vrhovima kocke čije koordinate odgovaraju granici terena, vrijednost gustoće postavljena je na 1, što odgovara zraku, kako bi se između graničnih i unutarnjih vrhova stvorila površina. U suprotnom bi vanjske površine bile prozirne. Usporedbu prikazuje Slika 3.9.



Slika 3.9. Teren bez granica, s granicama te s interpolacijom vrijednosti

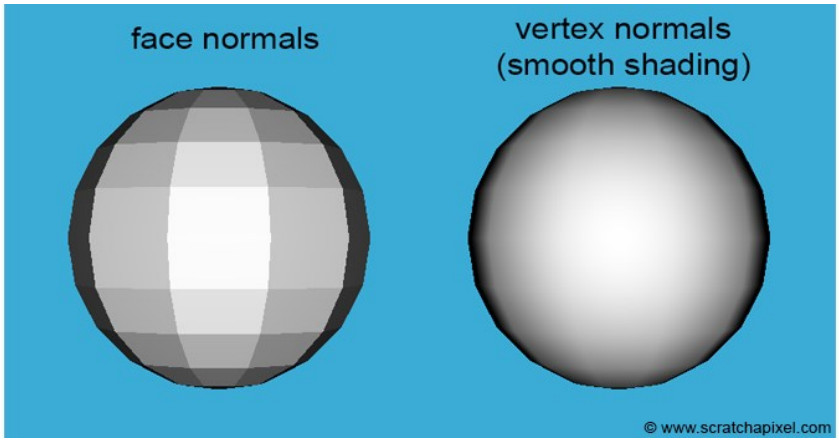
3.3 Uklanjanje višestrukih vrhova

Jedan od načina rješavanja problema prevelikog broja vrhova u mreži jest uklanjanje onih koji se preklapaju. Iako isprva zvuči kao da bi teren nakon uklanjanja višestrukih vrhova trebao izgledati isto kao i prije, zapravo će izgledati dodatno zaglađeno. Razlog tomu je interpolacija normala između susjednih trokuta, koju prikazuje Slika 3.10. [9]



Slika 3.10. Dijeljeni vrhovi (gore) i višestruki vrhovi (dolje)

Dosad je korišteno konstantno sjenčanje (engl. *Flat shading*), kod kojeg susjedni trokuti ne dijele vrhove. Iz tog razloga normale svih vrhova trokuta odgovarati će normali površine trokuta. Kad bi se uklonili višestruki vrhovi, tada bi normala dijeljenog vrha bila jednaka interpolaciji normala svih susjednih trokuta, što mreži daje zaobljeni izgled. Taj postupak naziva se glatko sjenčanje (engl. *Smooth shading*). Usporedbu prikazuje Slika 3.11. [11]



Slika 3.11. Normale plohe (lijevo), normale vrhova (desno)

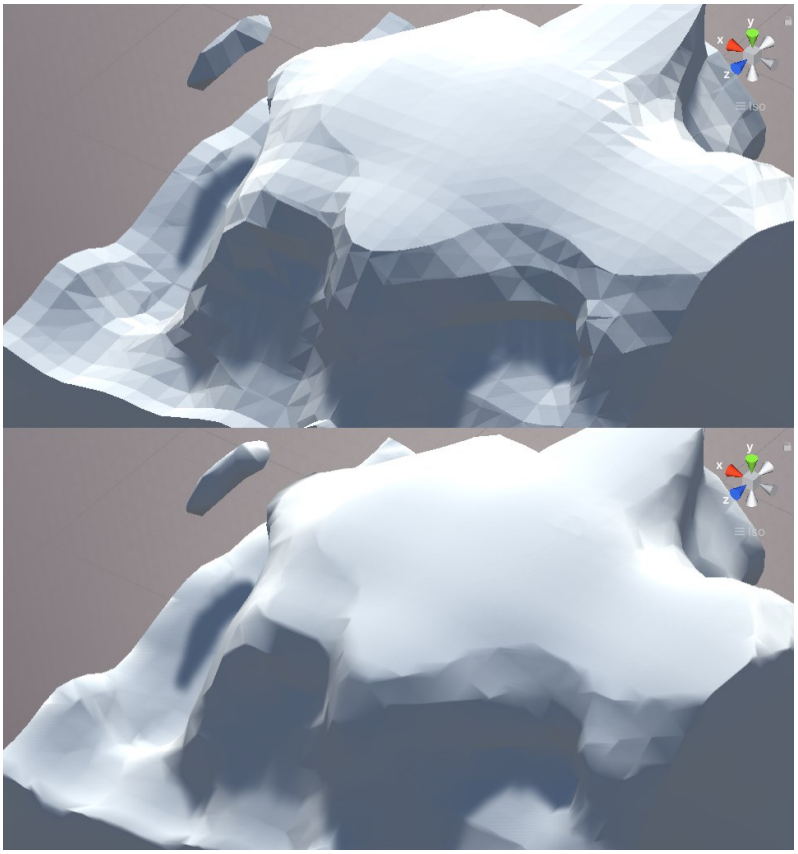
Najjednostavniji način uklanjanja višestrukih vrhove je provjera nalazi li se taj vrh već u listi prije dodavanja. To je jako neefikasan način, jer ima kvadratnu asimptotsku složenost. Rješenje s manjom vremenskom, ali većom prostornom

složenosti zahtjeva stvaranje nove strukture podataka. Konstantno vrijeme pristupa u programskom jeziku C# može se ostvariti rječnikom (engl. *Dictionary*). Ključ rječnika je vrh, a vrijednost njegov indeks u listi vrhova mreže. Slika 3.12. prikazuje funkciju korištenu za uklanjanje višestrukih vrhova.

```
int IndexForVertex(Vector3 vertex)
{
    if (vertexIndexDict.ContainsKey(vertex))
    {
        return vertexIndexDict[vertex];
    }
    //the vertex doesn't exist so we create it
    vertices.Add(vertex);
    int index = vertices.Count - 1;
    vertexIndexDict[vertex] = index;
    return index;
}
```

Slika 3.12. Funkcija koja vraća indeks vrha u mreži

Izgled terena nakon uklanjanja višestrukih vrhova prikazuje Slika 3.13. Teren izgleda glatko, od čega i dolazi ime *Smooth shading*. Mreža s višestrukim vrhovima prikazana na slici sadrži 22524 vrha, dok ona s dijeljenim sadrži 3762 vrha, što je skoro 6 puta manje vrhova. Taj broj je neobičan jer se svaki vrh stvara na bridu kocke, a jedan brid mogu dijeliti najviše 4 kocke, iz čega bi trebalo slijediti da će uklanjanjem višestrukih biti oko 4 puta manje vrhova. Ipak, vrhove ne dijele samo trokuti susjednih kocaka, već i unutar jedne kocke trokuti mogu imati isti vrh. To je razlog zašto je više od tri četvrtine vrhova uklonjeno. Vrijeme potrebno za prikaz terena u oba slučaja je identično zbog konstantnog vremena pristupa elementima rječnika.



Slika 3.13. Teren s višestrukim vrhovima (gore) i s dijeljenim vrhovima (dolje)

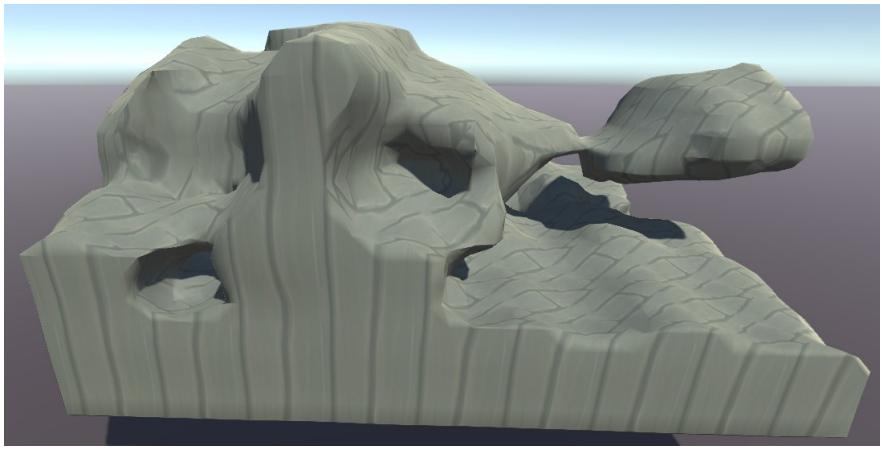
Razlog zašto na nekim dijelovima teren izgleda zgužvano je zbog viših oktava, koje dodaju detalje u obliku naglih promjena. To se može ublažiti smanjivanjem broja oktava, ili povećavanjem skale terena što je ekvivalentno uzorkovanju manjeg terena jednakim brojem točaka.

3.4 Uljepšavanje terena

Dosad je teren bio bezbojan i monoton. Kako bi se uljepšao izgled i olakšalo razumijevanje prikazanih oblika, napisan je takozvani triplanarni sjenčar (engl. *Triplanar shader*). Također, kako bi se teren pretvorio u površinu po kojoj se može hodati, umjesto oblika nalik sustavu pećina, napisana je funkcija koja će mijenjati vrijednost gustoće tako da više materijala bude na manjim visinama.

3.4.1 Triplanarni sjenčar

Jedan od načina dodavanja tekstura tijelu je projiciranje teksture iz nekog smjera. Triplanarni sjenčar naziva se tako jer teksturu projicira iz triju ravnina. Razlog je to što kada bi se tekstura projicirala iz samo jedne ravnine, došlo bi do razvlačenja na dijelovima koji su pod velikim kutom u odnosu na ravninu iz koje se projicira. Do najvećeg razvlačenja došlo bi na površinama koje su okomite na ravninu projekcije što prikazuje Slika 3.14. Tekstura prikazana na slici je besplatna tekstura preuzeta s Interneta. [13] Jednostavni sjenčar korišten u ovom radu napisan je po uzoru na referencu [12].



Slika 3.14. Razvlačenje teksture

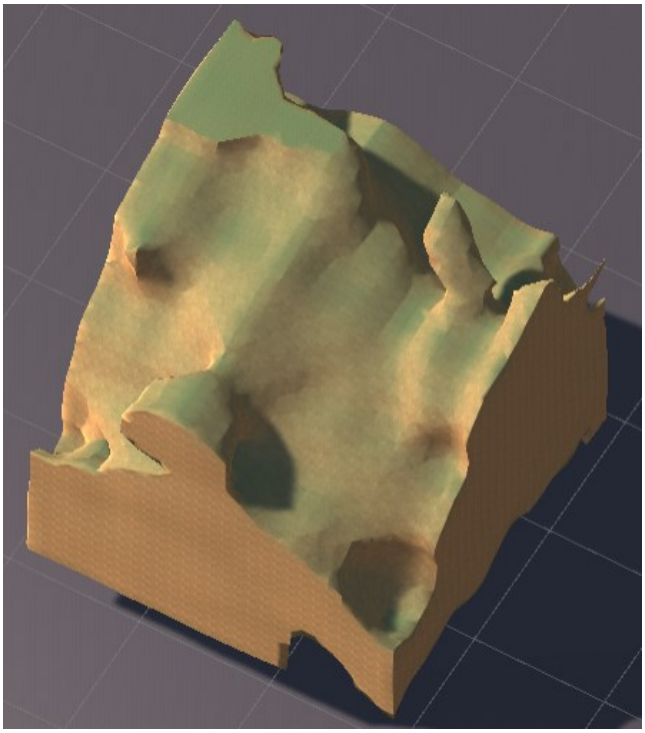
Dio koda sjenčara koji je zadužen za projiciranje teksture prikazuje Slika 3.15. Prikazana funkcija poziva se za svaki piksel terena. U njoj se na temelju pozicije piksela odredi koji piksel iz teksture će se preslikati na njega. Broj tekstura je proizvoljan, a u navedenom primjeru korištene su dvije teksture, jedna koja će se projicirati odozgo i predstavljati pod te druga, koja će se projicirati sa strane i predstavljati zidove. Na temelju normale na točku terena određuje se koliko će koja tekstura biti zastupljena. Rezultat je boja piksela koja je interpolacija svih tekstura koje se projiciraju.

```
void surf(Input IN, inout SurfaceOutputStandard o){
    float3 scaledWorldPos = IN.worldPos / _TexScale;
    float3 pWeight = abs(IN.worldNormal);
    pWeight /= pWeight.x + pWeight.y + pWeight.z;
    float3 xP = tex2D(_WallTex, scaledWorldPos.yz)*pWeight.x;
    float3 yP = tex2D(_MainTex, scaledWorldPos.xz)*pWeight.y;
    float3 zP = tex2D(_WallTex, scaledWorldPos.xy)*pWeight.z;
    o.Albedo = xP + yP + zP;

}
```

Slika 3.15. Funkcija površinskog sjenčara

Rezultat projiciranja teksture zemlje sa strane i teksture trave odozgo prikazuje Slika 3.16. Kao izvor tekstura korišteni su besplatni primjerci s Interneta. [14]



Slika 3.16. Teren prikazan korištenjem triplanarnog sjenčara

3.4.2 Funkcija opadanja volumena terena s visinom

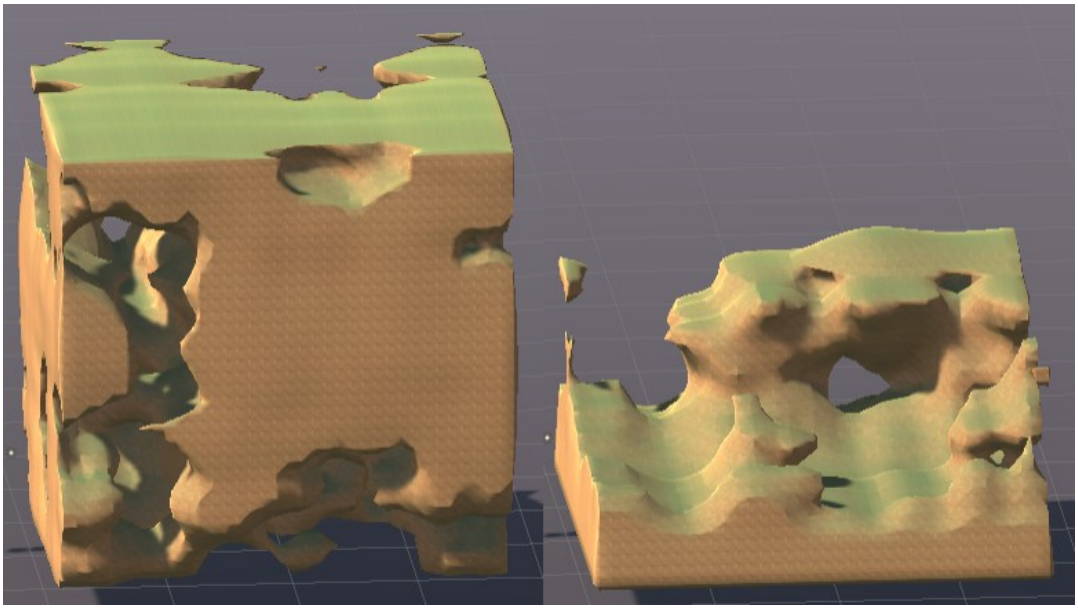
Dosada, postojala je prosječno jednaka količina materijala na svim visinama. To je bilo pogodno za generiranje nekakvog podzemnog sustava špilja, ali nije poželjno za generiranje tla. Kako bi se povećala vjerojatnost da će se materijal pojaviti na

nižim visinama, a smanjila na višim, napisan je dio koda kojeg prikazuje Slika 3.17.

```
pointDensity = GetPointDensity(new Vector3(x, y, z), octaves, scale,
    persistence, lacunarity, offset, perlinNum, usesFalloff);
if (usesFalloff)
{
    pointDensity *= falloffPower * ((float)y / (height));
}
```

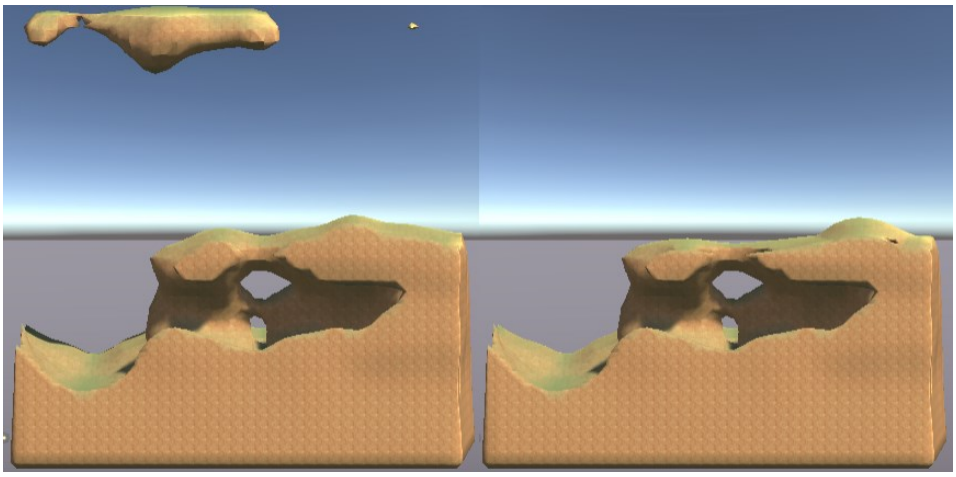
Slika 3.17. Dio koda koji smanjuje količinu materijala na većim visinama

Nakon dohvaćanja gustoće vrha, u slučaju da se koristi opadanje, ta vrijednost pomnoži se s faktorom snage opadanje te omjerom visine vrha i visine terena. Time će gustoća biti veća što je veća Y koordinata vrha. Podsjećam da je tablica trokuta korištena u ovom radu takva da 0 predstavlja da je točka unutar terena, a 1 da je izvan. Zato povećanjem gustoće teren na većim visinama prelazi u zrak. Faktor snage opadanja je proizvoljan, ali prema mom mišljenju daje najbolje rezultate za vrijednosti oko 4 i 5. Slika 3.18 prikazuje usporedbu terena bez opadanje i s faktorom snage opadanje 5.



Slika 3.18. Usporedba terena bez i s opadanjem

Jedan od problema ovog pristupa je stvaranje lebdećih elemenata. Dijelovi terena koji imaju vrlo nisku gustoću će čak i nakon množenja biti ispod granične gustoće te će biti vidljivi u zraku. Kad bi odrezali teren nakon neke visine smanjila bi se varijacija i narušio izgled terena. Zakrpa kojom se ublažio taj problem je uklanjanje viših oktava nakon određene visine. Još uvijek postoji mogućnost nastanka lebdećih elemenata, ali smanjuje se njihova učestalost, pogotovo za malene elemente. Usporedbu terena bez i s uklanjanjem viših oktava na većim visinama prikazuje Slika 3.19. Na desnoj strani druge slike vidljiva je prva oktava koja naizgled izbija iz terena, jer na tom dijelu nestaju više oktave.



Slika 3.19. Teren s lebdećim elementom i teren bez elementa

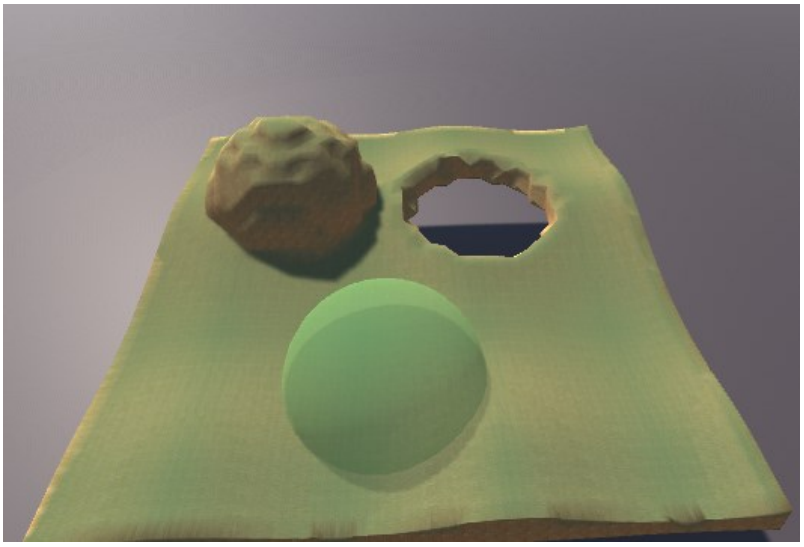
3.5 Izrada jednostavnog uređivača terena

Jedna od primjena algoritma pokretne kocke je kod uređivača terena, alata koji omogućava korisniku da u trodimenzionalnom prostoru stvara i uređuje elemente terena. Uređivač napravljen u ovom radu nije efikasan, ali služi kao ilustrativni primjer mogućnosti algoritma.

Za potrebe uređivača stvoren je objekt igrača, kojim se može upravljati i pomicati u prostoru. Za model igrača korištena je obična kapsula radi jednostavnosti. Mreži je dodan sudarač kako bi se omogućilo hodanje po terenu.

Uređivač radi na principu da se iz kamere vezane za igrača baci zraka (engl. *Raycast*) u smjeru kursora na ekranu. Kada korisnik pritisne lijevi ili desni klik,

pronađe se presjek zrake i terena, ako postoji. Ta koordinata se zatim pohrani u strukturu zajedno s radijusom promjene te informacijom je li materijal dodan ili uklonjen iz radijusa. Zatim se za sve vrhove ponovno provede algoritam pokretne kocke te iscrta nova mreža. Korisniku je omogućena promjena radijusa izmjene korištenjem kotačića za pomicanje, a radi lakše vizualizacije koji dio terena će biti izmijenjen, dodana je djelomično prozirna zelena sfera koja prati pokazivač, čiji radijus odgovara radijusu promjene. Izmijenjen teren prikazuje Slika 3.20.



Slika 3.20. Uređeni teren i sfera radijusa promjene

Funkciju koja mijena vrijednosti onim vrhovima zahvaćenim promjenom prikazuje Slika 3.21. Kao argumente prima informacije o promjeni i terenu. Ovisno o veličini terena i radijusu promjene, određuje granice unutar kojih će se provjeravati udaljenost vrhova od centra promjene. Razlog tomu je to što bi u suprotnom trebalo iterirati po svim vrhovima i računati njihovu udaljenost od centra promjene, što bi bilo jako neefikasno. Na ovaj način iterirati će se samo po vrhovima koji se nalaze unutar kocke duljine stranice dva radijusa te će se za njih provjeravati nalaze li se unutar sfere promjene. Pri provjeri koriste se kvadratne udaljenosti jer je računanje korijena skupa funkcija, a u ovom slučaju nije potrebno. Ovisno o vrsti promjene, točkama se vrijednost postavi na 0 ako se dodaje ili 1 ako se uklanja materijal.

```
public static void EditValues(Structs.TerrainChange change, int width, int height,
    int length, float[,,] noiseMap)
{
    Vector3 position = change.getPosition();
    float radius = change.getRadius();
    float sqrRadius = radius * radius;
    Vector3Int upperLimit = new Vector3Int((int)Mathf.Min(width, position.x + radius),
                                           (int)Mathf.Min(height, position.y + radius),
                                           (int)Mathf.Min(length, position.z + radius));
    Vector3Int lowerLimit = new Vector3Int((int)Mathf.Max(0, position.x - radius),
                                           (int)Mathf.Max(0, position.y - radius),
                                           (int)Mathf.Max(0, position.z - radius));
    for (int i = lowerLimit.x; i <= upperLimit.x; i++)
    {
        for (int j = lowerLimit.y; j <= upperLimit.y; j++)
        {
            for (int k = lowerLimit.z; k <= upperLimit.z; k++)
            {
                float sqrDistance = SquareDistance(new Vector3(i, j, k), position);
                if (sqrDistance < sqrRadius)
                {
                    if (change.getMaterial())
                    {
                        noiseMap[i, j, k] = 0;
                    }
                    else
                    {
                        noiseMap[i, j, k] = 1;
                    }
                }

            }
        }
    }
}
```

Slika 3.21. Funkcija koja mijenja vrijednosti vrhova

4. Implementacija algoritma na grafičkoj kartici

Dosad su se sve naredbe izvršavale slijedno na procesoru. Iz tog razloga posljednja točka morala je čekati da se izračuna vrijednost svih prijašnjih točaka prije nego se izračuna njena vrijednost. S obzirom da su vrijednosti u točkama međusobno neovisne, ne postoji logička prepreka zbog koje se ne bi mogle sve vrijednosti izračunati odjednom.

Iz prirode samog problema proizlaze tri bitne činjenice:

1. Dimenzije terena ne moraju biti promjenjive. Ako želimo veći teren moguće je stvoriti više komada terena (engl. *Chunks*) istih dimenzija.
2. Vrijednosti gustoće u svim točkama su međusobno nezavisne.
3. Algoritam pokretne kocke provodi se nad vrhovima iste kocke, što znači da su vrhovi različitih kocaka međusobno nezavisni.

Iz tih činjenica očito je da nema potrebe da se računanje gustoće i provođenje algoritma pokretne kocke odvija slijedno, jer se može paralelizirati. Za izvođenje velikog broja operacija istovremeno zadužena je grafička kartica (engl. *Graphics card*). Iako su originalno namijenjene za stvaranje izlaznog prikaza, što i implicira njihovo ime, posljednjih godina grafičke kartice koriste se sve više za probleme koji nisu grafičke prirode, poput treniranja umjetne inteligencije. [15]

U alatu Unity za preusmjeravanje cjevovoda izvođenja na grafičku karticu mogu se koristiti sjenčari računanja (engl. *Compute shaders*).

4.1 Sjenčar računanja

Sjenčar računanja je program koji se izvršava na grafičkoj kartici, izvan uobičajenog cjevovoda za prikaz. U alatu Unity pišu se jezikom HLSL u stilu DirectX11. [16] Sjenčar se sastoji od minimalno jednog računskog zrna (engl. *Compute kernel*). Zrna su funkcije u sjenčaru koje će se pozvati kada se sjenčar otpremi (engl. *Dispatch*). Svako zrno mora biti označeno s `#pragma kernel` direktivom.

Osnove korištenja sjenčara računanja bit će prikazane na jednostavnom primjeru generiranja visinske mape na temelju šuma. S obzirom da sjenčari nemaju pristup strukturama i razredima koje nudi alat Unity, kao izvor šuma korišten je *Simplex noise* napisan u jeziku GLSL, koji je preuzet s interneta. [17]

Važno je spomenuti da je *Simplex noise* u pravilu brži od Perlinovog šuma. Također, Perlinov šum u alatu Unity je dvodimenzionalan, dok je korišteni *Simplex noise* trodimenzionalan. Pri usporedbi brzine slijednog i paralelnog izvođenja programa treba ove činjenice uzeti u obzir.

```
[numthreads(numOfThreads,numOfThreads,1)]
void CSMain (uint3 id : SV_DispatchThreadID){
    float3 position = {id.x + offsetX, id.y+3.1, 0};
    float value = 0;
    for(int i = 0; i < octaves; i++){
        //snoise vraca vrijednosti u rasponu -1,1 pa skaliramo na raspon 0,1
        value += (2+snoise(position*frequency))*amplitude/2;
        amplitude*=persistence;
        frequency*=lacunarity;

    }
    points[id.x * numOfThreads*numThreadGroups + id.y] = value;
}
```

Slika 4.1. Glavno zrno sjenčara računanja

Glavno zrno sjenčara računanja za dodjeljivanje vrijednosti šuma vrhu prikazuje Slika 4.1. Ono što nije prikazano na slici jest direktiva `#pragma kernel CSMain`, koja označava funkciju kao početnom za to zrno, te deklaracija varijabla koje se pojavljuju u samoj funkciji. Vrijednosti tih varijabla postavlja pozivatelj prije otpremanja sjenčara.

Konstanta `numOfThreads` je statička, jer se unaprijed mora znati koliko dretvi će biti u jednoj grupi. Maksimalni broj koji dozvoljava ovaj sjenčar je 1024 pa je vrijednost konstante postavljena na 32, što je drugi korijen maksimalne vrijednosti. Najveći mogući broj dretava u grupi ne mora nužno dati najbolje rezultate, jer brzina izvođenja ovisi o grafičkoj kartici na kojoj se izvodi. U sjenčaru računanja dretve su organizirane u obliku trodimenzionalnog vektora te svaka od njih ima

svoj identifikacijski broj `id`. Takva organizacija omogućava da se identifikacijski broj dretve koristi kao koordinata točke za koju se računa vrijednost.

Ostatak funkcije računa vrijednost na identičan način kao i dosad, ali umjesto korištenja strukture `Vector3` koje nema u jeziku sjenčara, koriste se postojeće strukture `int3` te `float3` koje su joj identične. Na kraju funkcije izračunata vrijednost pohranjuje se u strukturu `points` na indeks određen koordinatom točke. Struktura `points` je tipa `RWStructuredBuffer<float>`. To je strukturirani međuspremnik za čitanje i pisanje koji u sebi pohranjuje realne brojeve. Za prijenos početnih i izračunatih podataka između pozivatelja i sjenčara koriste se računski međuspremnici (engl. *Compute buffers*). Primjer stvaranja i korištenja takvog spremnika prikazuje Slika 4.2.

```
void GetNoiseMap()
{
    int numEle = numOfThreads * numOfThreads * 1 * numThreadGroups * numThreadGroups;
    ComputeBuffer buffer = new ComputeBuffer(numEle, sizeof(float));

    populateNoiseMap2D.SetBuffer(0, "points", buffer);
    populateNoiseMap2D.SetInt("numThreadGroups", numThreadGroups);
    populateNoiseMap2D.SetInt("octaves", octaves);
    populateNoiseMap2D.SetFloat("lacunarity", lacunarity);
    populateNoiseMap2D.SetFloat("persistence", persistence);
    populateNoiseMap2D.SetFloat("frequency", frequency);
    populateNoiseMap2D.SetFloat("amplitude", amplitude);
    populateNoiseMap2D.SetInt("offsetX", offsetX);

    populateNoiseMap2D.Dispatch(0, numThreadGroups, numThreadGroups, 1);
    noiseMap = new float[numEle];

    buffer.GetData(noiseMap);
    buffer.Release();
}
```

Slika 4.2. Postavljanje varijabli i otpremanje sjenčara računanja

Pri stvaranju računskog međuspremnika tip spremnika je pretpostavljeni, a to je upravo `RWStructuredBuffer`. Njegova veličina mora biti jednaka broju elemenata koji će rezultirati pozivom sjenčara. U ovom slučaju ta veličina odgovara kvadratnoj matrici realnih brojeva dimenzija određenih brojem grupa dretava i brojem dretva u grupi. Ako je broj izračuna višekratnik broja dretava u grupi, tada nijedna dretva neće biti potraćena pa se dobije veća efikasnost.

Slijedeće naredbe postavljaju vrijednosti varijabla u sjenčaru uključujući i referencu na međuspremnik. Nakon inicijalizacije varijabli, poziva se metoda `dispatch` kojom se otprema sjenčar. Argumenti te metode su redni broj zrna koje se poziva te broj grupa dretva na svakoj osi. Kako se u ovom primjeru generira visinska mapa, broj grupa po Z osi postavljen je na 1.

Nakon otpreme sjenčara stvara se polje u koje se prekopiraju podatci iz međuspremnika, jer im program ne može pristupiti direktno kroz međuspremnik. S obzirom da je grafičkoj kartici potrebno neko vrijeme da obavi izračune, preporučljivo je obavljati neke korisne operacije između otpreme sjenčara i dohvaćanja podataka, jer će u suprotnom procesor samo čekati grafičku karticu. U ovom slučaju za to vrijeme alocira se polje za pohranu elemenata. Na kraju se međuspremnik oslobodi.

4.2 Usporedba slijednog i paralelnog izvođenja

Za efikasno korištenje sjenčara računanja potrebno je detaljno poznavanje arhitekture grafičkih kartica, stoga je teško ostvariti maksimalno teoretsko ubrzanje. Za usporedbu sa sjenčarom računanja napisana je funkcija koja slijedno generira i pohranjuje vrijednosti svih vrhova koristeći funkciju Perlinovog šuma.

Rezultate mjerenja prikazuje Tablica 1. Za posljednja dva slučaja broj iteracija smanjen je zbog predugog trajanja pa su ti rezultati manje precizni. Iz mjerenja je vidljivo da se ubrzanje paralelnog izvođenja u odnosu na slijedno povećava s brojem vrhova. Razlog tomu je smanjenje udjela linija koda koje se izvode slijedno u cjelokupnom trajanju izvedbe funkcije. Što je veći udio operacija koje se mogu paralelizirati, to će veće ubrzanje biti ostvareno. S obzirom da se u jednog grupi nalazi 1024 dretve tada bi teoretska granica ubrzanja bila 1024 puta, kada ne bi bilo slijednih dijelova programa. Dio prikazanog ubrzanja proizlazi iz činjenice da je korišteni *Simplex noise* brži od Perlinovog šuma, ali taj dio ubrzanja nije proporcionalan broju vrhova. Mjerenje je provođeno na procesoru Intel i5 9300H te grafičkoj kartici Nvidia GTX 1650.

Tablica 1. Usporedba trajanja paralelnog i slijednog izvođenja

Broj iteracija	Broj vrhova	Prosječno trajanje paralelno [s]	Prosječno trajanje slijedno [s]	Ubrzanje
100	262.144	0,002566 +/- 0,000405	0,063034 +/- 0,001504	24.56
100	1.048.576	0,007140 +/- 0,001208	0,239378 +/- 0,003768	33.52
100	4.194.304	0,017767 +/- 0,005529	0,982552 +/- 0,136805	55,30
10	16.777.216	0,066003 +/- 0,022387	3,992042 +/- 0,25245	60,48
10	67.108.864	0,235787 +/- 0,059348	15,224856 +/- 0,555886	64,57

4.3 Implementacija algoritma pokretne kocke korištenjem sjenčara računanja

Sjenčar za generiranje vrijednosti šuma već je napisan u prošlom primjeru. Uz manje promjene može se prilagoditi da radi za trodimenzionalne točke, tako da ono što ostaje je napisati sjenčar računanja koji će provoditi algoritam pokretne kocke. Sjenčar u ovom radu napisan je po uzoru na referencu [18].

Sadržaj sjenčara koji provodi algoritam pokretne kocke identičan je dosad korištenim funkcijama. Jedina razlika su koordinate vrhova koje se u sjenčaru dobivaju preko identifikacijskog broja dretve. U ovom primjeru na svakoj osi grupe dretava ima 8 dretvi, pa je ukupan broj dretvi u grupi 512. Osim međuspremnika `points` koji je bio prisutan u prošlom primjeru, u ovom sjenčaru prisutan je i međuspremnik za dodavanje (engl. *Append buffer*) koji u sebi pohranjuje podatke tipa `Triangle`. To je struktura koju sam definirao u sjenčaru i u programu

pozivatelja, a sadrži 3 trodimenzionalna vektora realnih brojeva koji predstavljaju koordinate vrhova trokuta.

Kada sjenčar provede algoritam nad jednom kockom, svaki od dobivenih trokuta doda u međuspremnik. Korištenje te strukture podataka potrebno je radi očuvanja redoslijeda vrhova u trokutima, jer se dretve izvršavaju istovremeno pa nema garancije da će jedna dretva uspjeti dodati sva tri vrha trokuta bez da neka druga dretva umetne svoj vrh između. Na ovaj način sva tri vrha dodaju se zajedno kao dijelovi iste strukture. Korištenje međuspremnika za dodavanje potrebno je jer se ne zna unaprijed koliko vrhova će biti u mreži.

```
int width = numOfThreads * numThreadGroups - 1;
int maxNumTriangles = 5 * width * width * width;
triangleBuffer = new ComputeBuffer(maxNumTriangles,
    sizeof(float) * 3 * 3, ComputeBufferType.Append);

triangleBuffer.SetCounterValue(0);
MarchCubes.SetFloat("thresholdDensity", thresholdDensity);
MarchCubes.SetInt("numThreadGroups", numThreadGroups);
MarchCubes.SetBuffer(0, "points", pointBuffer);
MarchCubes.SetBuffer(0, "triangles", triangleBuffer);

MarchCubes.Dispatch(0, numThreadGroups, numThreadGroups, numThreadGroups);

counterBuffer = new ComputeBuffer(1, sizeof(int), ComputeBufferType.Raw);
ComputeBuffer.CopyCount(triangleBuffer, counterBuffer, 0);

int[] triCountArray = { 0 };
counterBuffer.GetData(triCountArray);
int numTris = triCountArray[0];

Structs.Triangle[] triangleArray = new Structs.Triangle[numTris];
triangleBuffer.GetData(triangleArray, 0, 0, numTris);
```

Slika 4.3. Poziv sjenčara računanja za algoritam pokretne kocke

Dio koda koji otprema sjenčar za algoritam pokretne kocke prikazuje Slika 4.3. Prvo se stvara međuspremnik za dodavanje kojem se maksimalni broj elemenata postavlja na maksimalni broj trokuta kojima može rezultirati algoritam pokretne kocke. Za veličinu pojedinog elementa međuspremnika postavlja se veličina 9 realnih brojeva, jer se struktura `Triangle` sastoji od 3 varijable `Vector3` u

pozivatelju, odnosno `float3` u sjenčaru. Postavljaju se vrijednosti varijabla kao i dosad te se otprema sjenčar.

Nakon otpreme stvara se računski međuspremnik za brojanje, koji se koristi za prebrojavanje elemenata dodanih u međuspremnik za dodavanje. Ta informacija potrebna je za stvaranja polja trokuta potrebnih dimenzija kako bi se u njega naposlijetku upisali podatci iz međuspremnika.

Posljednji korak je dodavanje svih vrhova i trokuta u mrežu te iscrtavanje. Ovaj korak zahtijeva slijedni prolazak kroz sve trokute čime se smanjuje ubrzanje zbog implementacije na grafičkoj kartici.

Rezultati mjerenja nisu baš izvrsni. U prosjeku je brzina čitavog procesa generacije šuma i provođenja algoritma korištenjem sjenčara jedva red veličine brža nego slijedno, za komad terena dimenzija 24 x 24 x 24. Povećanjem dimenzija ubrzanje raste kao i u prošlom primjeru, ali veličina jednog komada ograničena je maksimalnim brojem vrhova u mreži, tako da već kod dimenzija 32 x 32 x 32 dolazi do prevelikog broja vrhova.

Velik dio vremena potroši se na prepisivanje podataka u polje nakon što ih sjenčar izračuna. Dobavljanje sadržaja međuspremnika koji sadrži trokute i iteriranje po trokutima kako bi se dodali u mrežu čini u prosjeku polovinu trajanja čitavog procesa.

Moguće poboljšanje performansi potencijalno bi bilo ostvarivo korištenjem sinkronizacije dretvi pri dodavanju vrhova trokuta, zbog čega ne bi bilo potrebno koristiti strukturu `Triangle`. Umjesto toga sadržaj međuspremnika mogao bi se direktno kopirati u polje vrhova mreže, a polje trokuta bilo bi lako generirati jer su svi vrhovi već poredani.

Zaključak

Glavni cilj rada bio je dinamičko stvaranje raznovrsnih terena te omogućavanje interakcije s terenom u obliku dodavanja i uklanjanja materijala.

Prvi zahtjev ostvaren je metodom proceduralnog generiranja koristeći Perlinov šum kao izvor pseudo-slučajnih informacija. Obrađena su svojstva takvog šuma te jednostavni postupak pretvaranja dvodimenzionalnog šuma u trodimenzionalni. Omogućena je promjena parametara za generaciju i interpretaciju šuma za vrijeme izvođenja programa, čime je ostvarena vizualizacija utjecaja različitih parametara na oblik terena u stvarnom vremenu.

Interakcija s terenom ostvarena je izradom uređivača terena. Korisniku je omogućeno dodavanje i uklanjanje materijala u određenom radijusu oko odabrane točke. Za vizualizaciju radijusa promjene korištena je poluprozirna sfera, a korisniku je omogućena promjena radijusa koristeći kotačić za pomicanje.

Jedan od početnih zahtjeva bio je stvaranje terena s konkavnim strukturama poput pećina i tunela. U tu svrhu istraženi su volumni elementi i algoritam pokretne kocke. Obrađeni su osnovni postupci prikaza terena tim tehnikama te su iznesene dobre i loše strane svakog od pristupa.

Naposlijetku, istraženi su sjenčari računanja kao jedan od pristupa implementaciji algoritama na grafičkoj kartici, s ciljem poboljšanja performanse paralelnim izvođenjem dijelova koda.

Jedno od mogućih poboljšanja rada bilo bi odvajanje igrača u vlastitu dretvu kako se ne bi zamrznuo za vrijeme generiranja terena. U ovom radu omogućeno je generiranje samo jednog komada terena. Kako bi se generirao čitav svijet potrebno je omogućiti dinamičko generiranje novih komada terena. Dodavanje sustava za promjenu razine detalja bi tada pomoglo s performansama.

Literatura

[1] Wikipedia, Procedural generation, datum pristupa 6.5.2020. https://en.wikipedia.org/wiki/Procedural_generation

[2] Pleek, Procedural noise generation, datum pristupa 6.5.2020. https://pleek.net/16_Noise.php

[3] Unity documentation, Mathf.PerlinNoise, datum pristupa 6.5.2020. https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

[4] Wikipedia, Voxel, datum pristupa 6.5.2020. https://en.wikipedia.org/wiki/Voxel

[5] Research gate, MRI Segmentation of the Human Brain: Challenges, Methods, and Applications, datum pristupa 6.5.2020. https://www.researchgate.net/figure/Illustration-of-image-elements-in-the-MRI-of-the-brain-An-image-pixel-i-j-is_fig12_276836180

[6] Wikipedia, Marching cubes, datum pristupa 6.5.2020. https://en.wikipedia.org/wiki/Marching_cubes

[7] Wikimedia, Marching cubes, datum pristupa 6.5.2020. https://commons.wikimedia.org/wiki/File:MarchingCubes.svg

[8] B3agz, How to Make 7 Days to Die in Unity - 01 - Marching Cubes, datum pristupa 2.3.2020. https://youtu.be/dTdn3CC64sc

[9] Paul Bourke, Polygonizing a scalar field, datum pristupa 6.3.2020. http://paulbourke.net/geometry/polygonise

[10] Graphicsbook, Introduction to Lighting, datum pristupa 13.3.2020. http://math.hws.edu/graphicsbook/c4/s1.html

[11] Scratchapixel, Introduction to Shading, datum pristupa 5.5.2020. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/shading-normals

[12] B3agz, How to Make 7 Days to Die in Unity - 03 - Triplanar Texturing, datum pristupa 16.4.2020. https://www.youtube.com/watch?v=Y2qS-c67NzU

[13] Inigmas Studios, Fantasy forest set, datum pristupa 3.5.2020. https://assetstore.unity.com/packages/3d/environments/fantasy/fantasy-forest-set-free-70568

[14] A dog's life software, Outdoor ground textures, datum pristupa 16.4.2020. https://assetstore.unity.com/packages/2d/textures-materials/floors/outdoor-ground-textures-12555

[15] Wikipedia, Video card, datum pristupa: 27.5.2020. https://en.wikipedia.org/wiki/Video_card

[16] Unity documentation, Compute shader, datum pristupa 27.5.2020. https://docs.unity3d.com/Manual/class-ComputeShader.html

[17] Github, keijiro, datum pristupa 27.5.2020.
https://github.com/keijiro/NoiseShader

[18] Github, Sebastian Lague, datum pristupa 27.5.2020.
https://github.com/SebLague/Marching-Cubes

Prikaz terena algoritmom pokretne kocke

Sažetak

Ovaj završni rad opisuje postupak proceduralnog generiranja terena u programskom alatu Unity. Kao osnovni primjer korišten je teren određen visinskom mapom. Za dodavanje detalja terenu korišten je veći broj oktava šuma. Opisan je način dobivanja trodimenzionalnog šuma od dvodimenzionalnog, u svrhu generiranja terena s konkavnim strukturama. Takav teren prikazan je na dva načina: volumnim elementima i algoritmom pokretne kocke. Izgled terena je poboljšan korištenjem glatkog sjenčanja i izradom triplanarnog sjenčara. Dodavanje i uklanjanje dijelova terena omogućeno je izradom jednostavnog uređivača terena. Algoritam pokretne kocke implementiran je na grafičkoj kartici koristeći sjenčare računanja. Provedena su mjerenja trajanja postupka generiranja visinske mape paralelnom i slijednom implementacijom. Nad dobivenim vremenima provedena je usporedba.

Ključne riječi: proceduralno generiranje; volumni elementi; uređivač terena; sjenčar računanja; algoritam pokretne kocke; perlinov šum;

Terrains Rendering by Marching Cubes Algorithm

Abstract

This thesis describes the process of procedural terrain generation in the game engine Unity. A terrain defined by a heightmap was used as a base example. Greater number of octaves were used for adding details to the terrain. A method of creating 3D noise from 2D noise was shown for the purpose of generating terrain with concave structures. Such a terrain was rendered in two way: using voxels and using the marching cubes algorithm. The appearance of the terrain was improved using smooth shading and a triplanar shader. Removing and adding parts of terrain was achieved by writing a simple terrain editor. The marching cubes algorithm was implemented on the graphics card using a compute shader. Measurements of heightmap generation times were carried out on both parallel and sequential implementations. A comparison was made on the resulting times.

Keywords: procedural generation; voxels; terrain editor; compute shader; marching cubes algorithm; perlin noise;