SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 3033

PROCEDURALNO GENERIRANJE TERENA I OBJEKATA U VIRTUALNOM OKRUŽENJU

Marko Grozaj

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 3033

PROCEDURALNO GENERIRANJE TERENA I OBJEKATA U VIRTUALNOM OKRUŽENJU

Marko Grozaj

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3033

Pristupnik:	**Marko Grozaj (0036503270)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Maja Matijašević

Zadatak: **Proceduralno generiranje terena i objekata u virtualnom okruženju**

Opis zadatka:

Virtualna okruženja uobičajeno se koriste za pojednostavljeno prikazivanje stvarnih sustava. Cilj ovog diplomskog rada jest osmisliti i razviti 3D virtualno okruženje koje korisniku, kroz prikladne primjere, omogućuje intuitivni prikaz zamišljene bežične mreže senzora (primjerice, za potrebe precizne poljoprivrede) postavljene na računalno generirani teren. Za izradu terena (npr. ravnice, rijeke, polja), koristite proceduralno generiranje terena Perlinovim šumom i pogonski sustav Unity. Za bežičnu mrežu senzora koja se postavlja na tom terenu, objekti kojima su prikazani senzori mogu predstavljati neki uobičajenih u kontekstu precizne poljoprivrede - primjerice, za mjerenje temperature zraka, temperature tla, vlažnosti tla i slično (nije nužno da imaju realan izgled ili svojstva). Postavljanje mreže senzora na teren treba biti vođeno isključivo geometrijskim svojstvima (dimenzije terena, domet senzora i sl.) te vjerojatnošću pokrivanja. Korisničko sučelje treba sadržavati kontrole kojima se može birati među ponuđenim primjerima ili kreirati vlastiti, uz ponuđen izbor parametara.

Rok za predaju rada: 27. lipnja 2022.

Sadržaj

UVOD **1**

1. PROCEDURALNO GENERIRANJE U IGRAMA **2**

1.1. MOGUĆNOSTI PROCEDURALNOG GENERIRANJA U IGRAMA 3

1.2. FUNKCIJE ŠUMA 4

1.3. SVOJSTVA FUNKCIJE ŠUMA POGODNE ZA 2D-TEKSTURE 5

1.4. FUNKCIJA BIJELOG ŠUMA 5

1.5. FUNKCIJA PERLINOVOG ŠUMA 6

2. ANALIZA GEOMETRIJE TERENA **11**

2.1. VORONOIJEVI DIJAGRAMI 11

2.2. IMPLEMENTACIJA VORONOIJEVIH DIJAGRAMA 13

2.3. FORTUNE ALGORITAM 14

2.4. LLOYDOV ALGORITAM RELAKSACIJE 17

3. DIZAJN IMPLEMENTACIJSKOG RJEŠENJA **20**

3.1. AKCIJE 22

4. RAZVOJ IMPLEMENTACIJSKOG RJEŠENJA **25**

4.1. RAZVOJNA OKOLINA 25

4.2. ALATI 25

4.3. UNITY 25

4.4. UNITY UI 26

4.5. IZGLED KORISNIČKOG SUČELJA U IMPLEMENTACIJSKOM RJEŠENJU 27

4.6. UNITY MESH 34

4.7. GENERIRANJE TERENA 36

4.8. POSTAVLJANJE SENZORA 39

4.9. MOGUĆA POBOLJŠANJA 43

5. REZULTATI **44**

5.1. POSTAVLJANJE PARAMETARA GENERIRANOG TERENA 44

5.2. PRIMJERI GENERIRANOG TERENA 46

5.3. ZNAČAJKE PROGRAMSKOG RJEŠENJA 47

ZAKLJUČAK **49**

LITERATURA **50**

SAŽETAK **53**

SUMMARY **54**

Uvod

Proceduralno generiranje svoju primjenu nalazi u virtualnim okruženjima, ponajviše kod razvoja računalnih igara. Način koji je najjednostavniji i može imati vrlo široku primjenu jest korištenje funkcija šuma za proceduralno generiranje objekata. Postoje razne funkcije šuma i njima se vrlo precizno mogu opisati pojave iz realnog svijeta te tako u virtualno okruženje prenijeti što više sličnosti sa svijetom u kojem živimo. Kao i kod svake nasumičnosti tako i proceduralno generiranje treba ograničiti na određene načine. Upravo taj balans između pravila i nasumičnosti odvaja dobre tehnike proceduralnog generiranja od loših. Kroz ovaj rad analizirati će se funkcije šuma i njihova primjena u proceduralnom generiranju virtualnih okruženja. Osim analize, bit će predstavljeno i programsko rješenje koje kroz korisničko sučelje daje mogućnost proceduralnog generiranja terena pomoću funkcije Perlinovog šuma i omogućuje da se generirani teren koristi kao podloga za simulaciju precizne poljoprivrede pomoću mreže senzora. Rad je organiziran u pet poglavlja gdje se analiziraju mogućnosti proceduralnog generiranja u virtualnom okruženju i opisuje postupak izrade programskog rješenja te daju prijedlozi za moguća poboljšanja programskog rješenja.

1. Proceduralno generiranje u igrama

Pojam proceduralno generiranje jako je sličan pojmu proceduralnog programiranja. Proceduralno programiranje je programska paradigma koja se temelji na pozivu postupaka (procedura), utvrđuje se niz koraka za rješavanje određenog problema i točnog redoslijeda koji treba izvršiti kako bi se postigao željeni rezultat [2]. Korištenje proceduralnih tehnika pri dizajnu videoigara, pružaju alternativu stvaranju velikih kompleksnih virtualnih okruženja bez uključivanja ljudskog faktora (dizajnera) u sam proces izrade svakog pojedinog dijela okruženja. Glavna ideja jest da računala generiraju složeno virtualno okruženje. Proceduralnim generiranjem možemo kreirati likove, terene, poboljšanja koja igraču omogućuju napredovanje i time omogućiti održivost igre kroz duži period. Igrač može više puta odigrati istu igru, svaki puta s različitim sadržajem, i doživjeti tu igru na drugačiji način. Kako bi se izbjegao gubitak kontrole nad procesom dizajna poželjno je da dizajneri mogu promijeniti neke parametre proceduralnog generiranja. Tome služe pravila u proceduralnom generiranju. Primjerice, u igri preživljavanja, gdje igrač mora biti smješten na napušteni otok i glavni cilj mu je pronaći resurse na tom otoku kako bi napravio brod i vratio se na kopno, postojala bi sljedeća pravila:

- Otok mora biti odvojen od kopna i sa svih strana okružen vodom,
- Otok mora sadržavati potrebne resurse kako bi igrač mogao ostvariti svoj cilj

Bez tih pravila, igra bi bila, u nekim slučajevima, nerješiva. Proceduralnim generiranjem postiže se prividna nasumičnost, ali u toj nasumičnosti uvijek postoje neka pravila. Proceduralno generiranje već u početku određuje smjer razvoja igre upravo zbog pravila koja se moraju definirati. Ne mogu ni sve igre primijeniti ovaj način oblikovanja i kreiranja sadržaja. Primjerice, igre koje se temelje na priči i žele da igrač slijedi nekakav put kroz priču, nemaju prevelike koristi od proceduralnog generiranja, pogotovo ako je priča igre povezana s objektima koji se nalaze u virtualnom okruženju. S druge strane, igre koje nisu orijentirane na priču i nemaju neke striktne točke do kojih igrač mora doći kako bi pratio priču, imaju velike koristi od proceduralnog generiranja jer korisnik svaki puta može drugačije doživjeti igru. Godine 2016. izdana je igra *No man's sky*, koja se temelji upravo na proceduralnom generiranju. Posebnost ove igre je da prilikom stvaranja novog svijeta, igra proceduralno generira jedan planet i sve njegove elemente, od ekosustava do hranidbenog lanca. Igrač može uništavati taj svijet kako bi prikupio resurse, isto tako može ubijati

životinje. Problem koji se nameće jest kako igraču ponovno prikazati taj isti svijet, koji je na početku igre proceduralno generiran, bez pohrane velike količine podataka. Rješenje se temeljilo na tome da se sprema nasumični broj koji inicijalizira generator nasumičnih brojeva (engl. *random seed*) i svi parametri prema kojima se planet generirao i kako igra napreduje tako se spremaju samo događaji kojima igrač utječe na promjenu proceduralno generiranog svijeta. Prilikom pokretanja igre ili prikaza tog istog svijeta drugom igraču samo se primjene događaji kojima je igrač utjecao na proceduralno generirani svijet.

1.1. Mogućnosti proceduralnog generiranja u igrama

Kod korištenja proceduralnog generiranja u igrama postoji nekoliko mogućih smjerova za odabir jedinice računalne igre koja će se proceduralno generirati [4]:

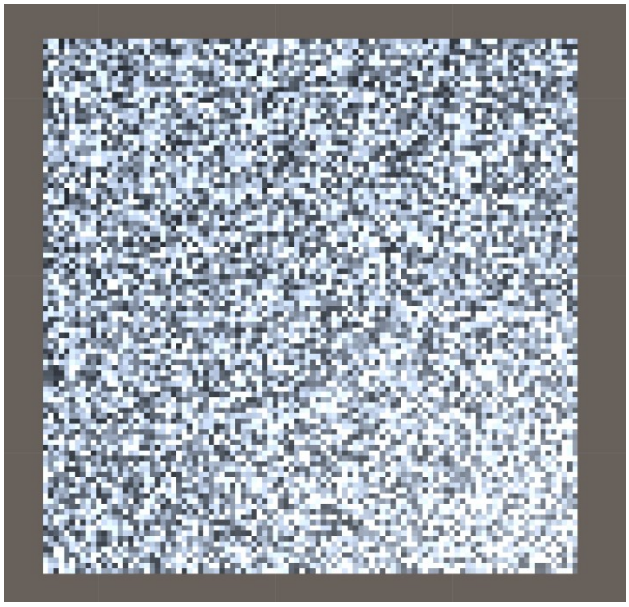
- objekti računalne igre,
- virtualno okruženje igre,
- sustavi igre

Objekti računalne igre mogu biti apstraktni i konkretni. Apstraktni objekti su oni s kojima igrač ne interaktira direktno, već kombiniranje više ovakvih apstraktnih objekata čini veću cjelinu koja igraču daje dojam uronjenosti u igru. Konkretni objekti su oni s kojima igrač može u bilo kojem smislu interaktirati. Primjerice, apstraktni objekti su teksture i zvukovi, a konkretni objekti su vegetacija, objekti (engl. *collectibles*) koje igrač može skupiti kako bi mu pomogli u napredovanju, valuta u igri. Više objekata postavljenih na nekom terenu čine virtualno okruženje igre. Virtualno okruženje može biti otvorenog ili zatvorenog tipa. Otvoreni tip virtualnog okruženja su igre s velikim otvorenim svijetom kojeg igrač može istraživati, a virtualno okruženje zatvorenog tipa ograničeno je područje u kojem se igrač može kretati. Odabir virtualnog okruženja ima izrazito velik utjecaj na tip igre, a potom i na način kojim će se virtualno okruženje izgraditi. Primjer otvorenog virtualnog okruženja koji koristi proceduralno generiranje kako bi generirao virtualno okruženje jest Minecraft. Minecraft koristi kompleksan način generiranja terena, ali u svojoj srži sve su to funkcije šuma [6]. Posljednji, vrlo bitan, dio igre koji je moguće proceduralno generirati jesu sustavi igre. U sustave igre možemo svrstati sve interakcije s drugim igračima, likovima u igri, životinjama. Sustave u igri možemo proceduralno generirati tako da definiramo striktna pravila, primjerice hranidbeni lanac. U hranidbenom lancu točno definiramo pravila prema kojima se životinje odnose jedna prema drugoj, a sve ostalo prepustimo proceduralnom

generiranju i tako svaki puta, prilikom simulacije hranidbenog lanca, možemo dobiti drugačije rezultate. Vrlo čest način proceduralnog generiranja u igrama i općenito računalnoj grafici jest korištenje funkcija šuma.

1.2. Funkcije šuma

Potreba za funkcijama šuma javila se sredinom 1980-ih godina u studijima koji su se bavili izradom 3D računalnih objekata [19]. Zbog ograničenih resursa, odnosno ograničenih performansi računalne opreme, nije bilo moguće koristiti kompleksne slike za teksturu objekata te su grafički dizajneri počeli koristiti jednostavne boje kao teksture. Korištenjem tih jednostavnih boja objekti su poprimili 2D izgled umjesto željenog 3D izgleda zbog nedostatka sjena i dubine objekta koja se može postići korištenjem slike kao teksture. Rješenje tog problema leži u funkcijama šuma. Prema definiciji, funkcija šuma je funkcija n varijabli koja vraća vrijednost s decimalnom točkom u intervalu [0,1]. Funkcija koja zadovoljava isključivo uvjet vraćanja nasumičnih vrijednosti u intervalu [0,1] je funkcija bijelog šuma. U prirodi se pojavljuju nasumični uzorci no ti uzorci su zapravo povezani, dvije točke tog uzorka, koje su blizu jedna drugoj, imaju gotovo ista svojstva, a točke u funkciji bijelog šuma nemaju nikakvu povezanost, a računalna grafika se želi što više približiti prirodi i svijetu oko nje pa su se definirale funkcije koje uspješno oponašaju pojave u prirodi, primjerice funkcija Perlinovog šuma.



Slika 1 Funkcija koja vraća nasumične brojeve između 0 i 1

1.3. Svojstva funkcije šuma pogodne za 2D-teksture

Na primjeru (Slika 1) može se primijetiti kako ideja o nasumičnim brojevima ne daje zadovoljavajuće rezultate jer su to nepovezani nasumični brojevi, a funkcije šuma za dvodimenzionalne teksture ipak ne isporučuju nepovezane brojeve. Koriste pseudo – nasumične brojeve te za isti ulaz uvijek trebaju dati isti izlaz odnosno te funkcije su determinističke, postoji jasan algoritam kojim se konačne vrijednosti generiraju. S obzirom na primjenu funkcija šuma u računalnim okruženjima, koja funkcioniraju na način da osvježavaju prikaz korisniku nekoliko puta u sekundi, determinizam je jako bitno svojstvo kako iz slike (engl. *frame*) u sliku ne bi dobili drugačiju teksturu kao posljedicu drugačijih vrijednosti funkcija šuma iz slike u sliku. Drugo bitno svojstvo je da funkcija šuma uvijek vraća jednu vrijednost s decimalnom točkom u intervalu [0,1]. Postoje razne implementacije funkcije šuma, od funkcija koje primaju jednodimenzionalni vektor do funkcija koje na ulazu dobivaju vektor od 4 dimenzije. Matematički, takve funkcije bi definirali na sljedeći način $f: R^n \rightarrow R$. Funkcije šuma bi trebale biti kontinuirane i diferencijabilne u svakoj točki. U svakoj točki funkcije, mora biti moguće pronaći tangentu, derivaciju funkcije u toj točki. Primjer funkcije koja je kontinuirana i diferencijabilna u svakoj točki jest $f(x) = \sin(x)$. Upravo diferencijabilnost u svakoj točki i kontinuitet daju povezanost između dvije točke, odnosno da dvije točke koje su blizu jedna drugoj imaju sličnu vrijednost funkcije. Primjerice funkcija $f(x) = |x|$ se ne može se iskoristiti za funkciju šuma jer nije diferencijabilna u svakoj točki. U $f(0)$ ne možemo pronaći derivaciju. Konačno, posljednji zahtjev koji bi idealna funkcija šuma trebala zadovoljavati jest da bude periodična ili da se uzorak ponavlja. Periodične funkcije, poput $f(x) = \sin(x)$ ponavljaju svoje vrijednosti nakon nekog vremena, što znači da kada se teksture, generirane pomoću funkcija šuman primjenjuju na niz površina od 100x100 piksela, a funkcija koja je generirali šum je periodična, dvije površine koje su jedna do druge neće imati iste vrijednosti i rezultat će biti potpuno drugačija tekstura, no već nakon nekoliko površina, vrijednosti bi se mogle ponavljati.

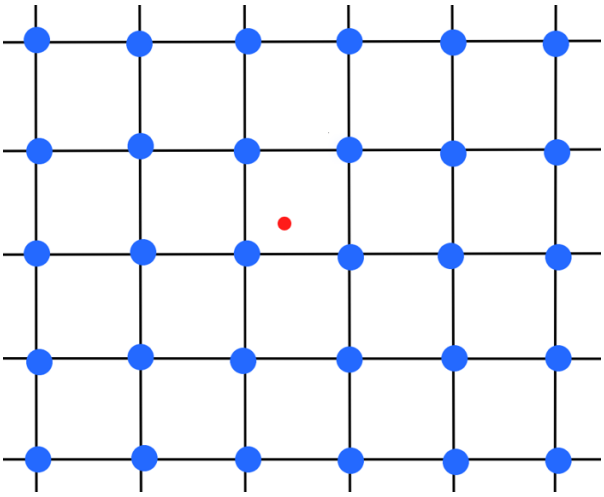
1.4. Funkcija bijelog šuma

Obilježje funkcije bijelog šuma (engl. *White noise function*) je da uzastopni uzorci nisu u korelaciji. Drugim riječima, autokorelacijska funkcija bijelog šuma je impuls s kašnjenjem nula. Budući da je spektralna gustoća snage Fourierova transformacija autokorelacijska

funkcije, Spektralna gustoća snage (engl. *Power spectral density*) bijelog šuma je konstantna. Stoga su sve frekvencijske komponente jednako prisutne – otuda i naziv bijela u analogiji s bijelom svjetlošću (koja se sastoji od svih boja u jednakim količinama) [24]. Funkcija bijelog šuma često se spominje u konotaciji sa zvučnim frekvencijama. Bijeli šum vrlo je sličan kiši, šumu dalekovidnice.

1.5. Funkcija Perlinovog šuma

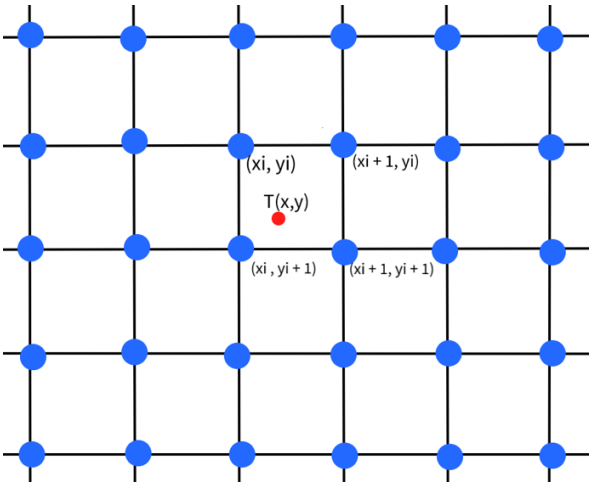
Funkciju Perlinovog šuma razvio je Ken Perlin tijekom rada na originalnom filmu Tron iz ranih 80-ih te je i 1997. godine dobio filmsku nagradu Oscar za tehnička dostignuća. Svoju novu funkciju koristio je za stvaranje proceduralnih tekstura za računalno generirane efekte [15]. Postoje dvije verzije funkcije Perlinovog šuma, jedna je originalna, definirana 80-ih godina. Druga, poboljšana verzija [7] definirana je 2002.godine. Obje verzije funkcije temelje izračun vrijednosti na jediničnim vektorima gradijentima [7].



Slika 2 Primjer vektora za kojeg želimo pronaći vrijednost Perlinovog šuma

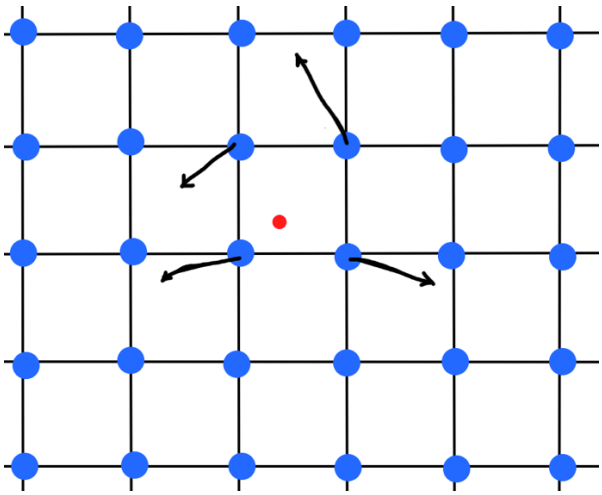
Na slici (Slika 2) prikazana je točka T(x, y) za koju je potrebno pronaći vrijednost Perlinovog šuma. Postupak pronalaženja vrijednosti Perlinovog šuma za točku je sljedeći:

- Pronaći vrijednosti plavih vektora koji okružuju traženu točku T. Formule za izračun vrijednosti plavih vektora u koordinatnom prostoru dane su prema sljedećim formulama
    - $xi = floor(x),\ yi = floor(y)$



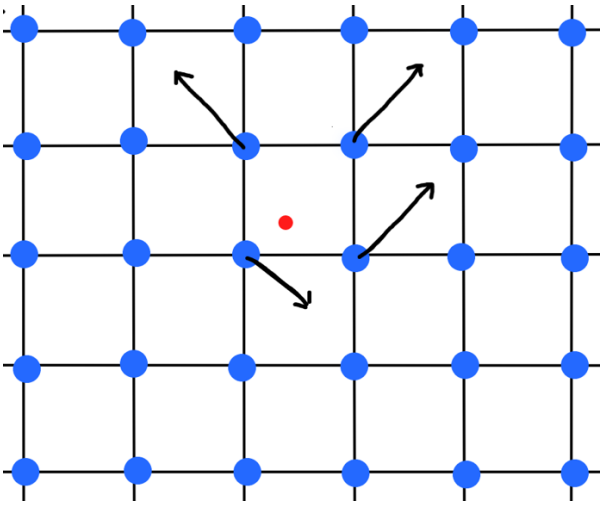
Slika 3 Formule za izračun koordinata plavih vektora

- Sljedeći korak ovisi o verziji funkcije. Osnovna verzija Perlinovog šuma iz svakog od 4 plava vektora odabire slučajan jedinični vektor koji predstavlja gradijent (primjer je prikazan na slici (Slika 4))



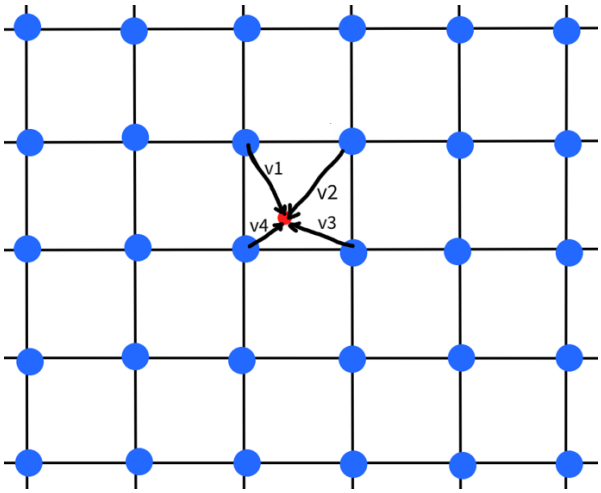
Slika 4 Gradijenti osnovne verzije Perlinovog šuma

Poboljšana verzija ne odabire slučajne vektore već odabire jedan od dijagonalnih vektora koji mogu ići u bilo kojem smjeru dijagonalno, dakle iz skupa vektora ((1, 1), (-1, 1), (1, -1), (-1, -1)).



Slika 5 Gradijenti poboljšane verzije Perlinovog šuma

- Nakon pronalaženja vrijednosti plavih okolnih vektora i slučajne vrijednosti gradijenta u tim vektora, potrebno je pronaći i vektor udaljenosti između svakog plavog vektora i vektora za koju se traži vrijednost Perlinovog šuma.



Slika 6 Prikaz vektora udaljenosti plavih vektora i željenog vektora

Vektori se računaju prema sljedećim formulama

  - $v1 = (x, y) - (xi, yi)$
  - $v2 = (x, y) - (xi + 1, yi)$
  - $v3 = (x, y) - (xi + 1, yi + 1)$
  - $v4 = (x, y) - (xi, yi + 1)$

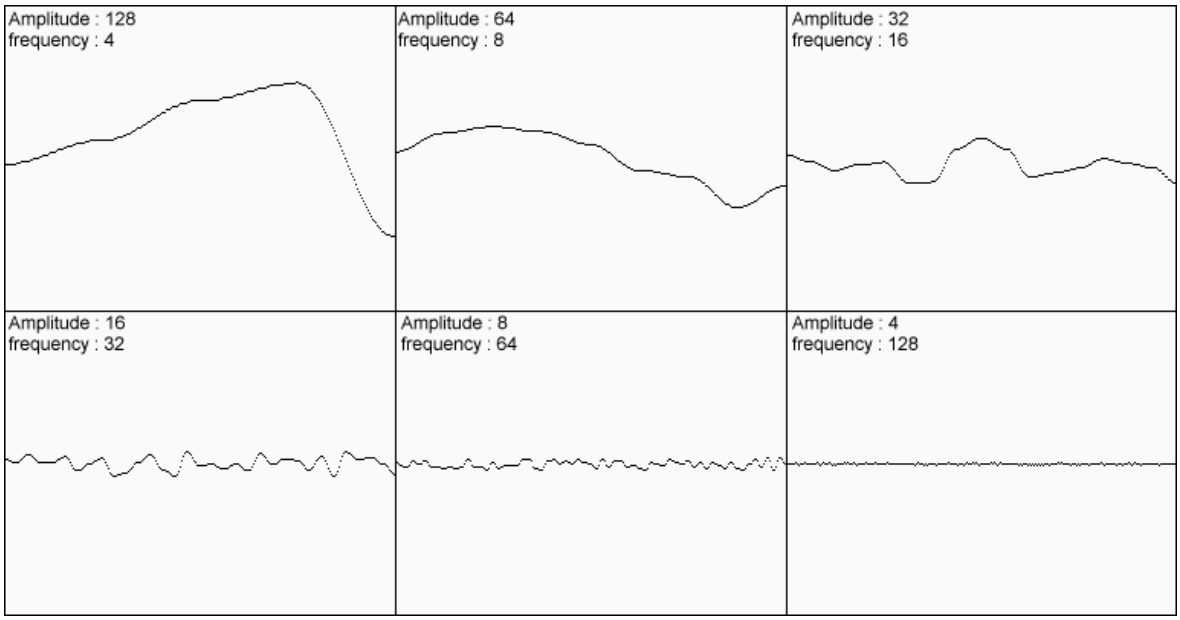
- Uz definirane vektore gradijenta i vektore udaljenosti moguće je izračunati skalarni umnožak ta dva vektora kako bi se dobila vrijednost utjecaja pojedinih plavih vektora na vrijednost tražene točke. Zaključno, vrijednost tražene točke bit će prosjek

vrijednosti utjecaja pojedinih plavih točaka. S obzirom na taj zaključak potrebno je izračunati prosjek utjecaja.

```
function(vector inputPoint) {
    number x = floor(inputPoint.x);
    number y = floor(inputPoint.y);

    //definiramo okolne vektore (plavi vektori)
    vector i1 = new Vector(x, y);
    vector i2 = new Vector(x+1, y);
    vector i3 = new Vector(x, y+1);
    vector i4 = new Vector(x+1, y+1);

    //izracun gradijentnih vektora u plavim vektorima
    vector g1 = selectRandomGradientVector(i1.x, i1.y);
    vector g2 = selectRandomGradientVector(i2.x, i2.y);
    vector g3 = selectRandomGradientVector(i3.x, i3.y);
    vector g4 = selectRandomGradientVector(i4.x, i4.y);

    //izracun vektora udaljenosti izrazenog vektora do plavih vektora
    vector d1 = subtractTwoVectors(inputPoint, g1);
    vector d2 = subtractTwoVectors(inputPoint, g2);
    vector d3 = subtractTwoVectors(inputPoint, g3);
    vector d4 = subtractTwoVectors(inputPoint, g4);

    //izracun utjecajnih vrijednosti svakog plavog vektora
    number influenceValue1 = dotProduct(g1, d1);
    number influenceValue2 = dotProduct(g2, d2);
    number influenceValue3 = dotProduct(g3, d3);
    number influenceValue4 = dotProduct(g4, d4);

    //interpolacija vrijednosti uz primjenu fadeout funkcije
    return lerp(lerp(influenceValue1, influenceValue2, fadeout()), lerp(influenceValue1, influenceValue4, fadeout()), fadeout());
}
```

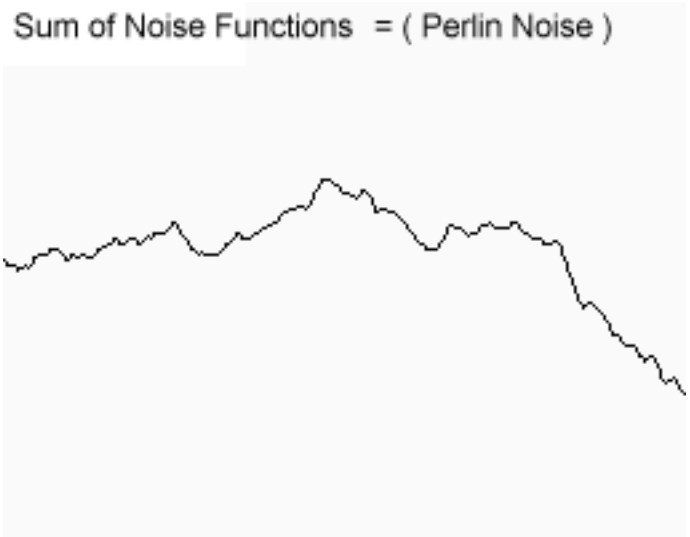
Isječak koda 1 Pseudokod za izračun vrijednosti Perlinovog šuma

Funkcija Perlinovog šuma je zbroj nekoliko funkcija koherentnog šuma stalno rastućih frekvencija i stalno padajućih amplituda Na slici (Slika 7) prikazano je 6 funkcija, sa svakom novom funkcijom amplituda je upola manja, a frekvencija dvostruko veća. Svaka od tih 6 funkcija predstavlja jednu oktavu Perlinovog šuma, dakle oktava je svaka funkcija koherentnog šuma koja je dio Perlinove funkcije šuma. Takve funkcije nazivaju se oktava zato što svaka sljedeća ima dvostruku frekvenciju od prethodne, a to isto svojstvo imaju i glazbeni tonovi [8].



Slika 7 Prikaz funkcija s različitim amplitudama i frekvencijama, izvor: [15]

Rezultat zbrajanja funkcija sa prethodne slike je prikazan na slici (Slika 8)



Slika 8 Funkcija Perlinovog šuma, izvor: [15]

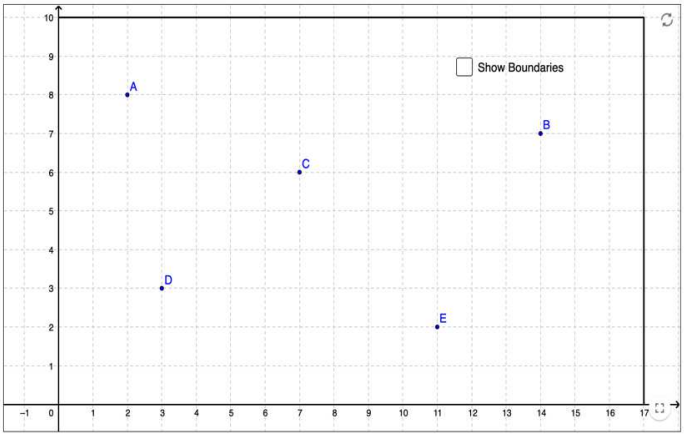
S pojmom oktava posljedično dolaze dva pojma, koja su također bitna za funkciju Perlinovog šuma i za potrebe generiranja virtualnog okruženja pomoću iste, *lacunarit*y i *persistence*. *Lacunarity* je faktor koji određuje brzinu povećavanja frekvencije za svaku sljedeću oktavu. Frekvencija svake sljedeće oktave jednaka je umnošku frekvencije prethodne oktave i *lacunarity* faktora. *Persistence* je faktor koji određuje koliko brzo se amplitude smanjuju za svaku sljedeću oktavu. Svaka sljedeća amplituda računa se na identičan način kao i *lacunarity* faktor, dakle amplituda sljedeće oktave je umnožak prethodne i *persistence* faktora.

2. Analiza geometrije terena

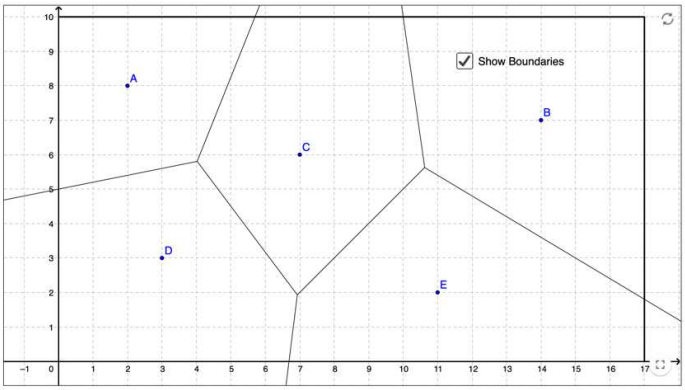
Generiranje terena Perlinovim šumom samo po sebi je vrlo zanimljiv problem i zahtijeva razumijevanja raznih koncepata i konačno generirani teren se može iskoristiti za razne analize svojstava takvog terena. Primjerice, nad generiranim terenom mogu se analizirati svojstva koje takvo tlo ima u vidu poljoprivrede ili pak bi se moglo pokušati pronaći najmanji optimalni broj senzora, za analizu svojstava tla, koji bi pokrivali svojim dometom najveću moguću površinu terena kako bi za svaki komadić terena postojala informacija o svojstvima tog dijela terena.

2.1. Voronoijevi dijagrami

Na 2D površini je raspoređeno N nasumičnih točaka (Slika 9). Svaka nasumična točka naziva se „site“. Voronoijev dijagram rasporedit će dvodimenzionalnu površinu na N poligona koji okružuju točke čija je udaljenost do referente točke (engl. *site*), unutar istog poligona, manja nego udaljenost do bilo koje druge referentne točke.



Slika 9 Raspored referentnih točaka prije generiranja Voronijeva dijagrama, izvor:[21]



Slika 10 Konstruiran Voronoijev dijagram prema referentnim točkama, izvor: [21]

Voronoijevi dijagrami još se nazivaju i „Dirichletova teselacija“ ili „Thiessenovi poligoni“. Primjeri Voronoijevih dijagrama vrlo često su prisutni u prirodi. Tvore učinkovite oblike jer prekrivaju cijelu površinu i moguće je vrlo malen prostor podijeliti na velik broj ćelija pa se struktura Voronoijevih dijagrama može pronaći primjerice u pčelinjim košnicama, mišićnim vlaknima. Osim stvarnih okruženja, Voronoijevi dijagrami korisni su i u virtualnim okruženjima pa se tako primjerice mogu iskoristiti za simulaciju uništavanja raznih materijala (primjerice stakla) ili za uništavanje samih modela unutar virtualnog okruženja. Na početku poglavlja spomenuto je kako Voronoijevi dijagrami prekrivaju dvodimenzionalnu površinu no to ne znači da se oni isključivo mogu koristiti u dvodimenzionalnom prostoru, dapače, njihovu matematičku interpretaciju moguće je primijeniti i na *N* dimenzija. U skupu $p = \{p1, p2, p3, \dots, m\}$ nalazi se m referentnih točaka prema kojima se konstruira m ćelija, Ci, koje sadrže sve točke u prostoru čija je udaljenost do referentne točke pi manja od bilo koje druge točke iz skupa p. Dakle skup Voronoijevih ćelija definiran je na sljedeći način:

$$Ci = \{x: \forall j \neq i \; uz \; d(x, pi) \leq d(x, pj)\}, i, j \in \{1,2,3, \dots, m\},$$

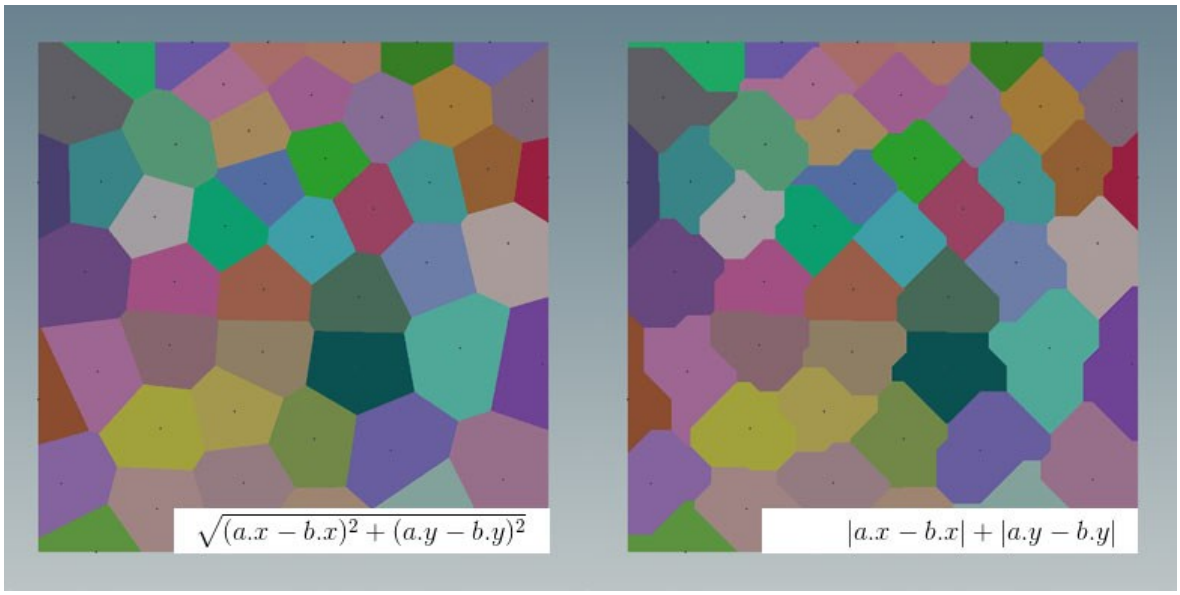
gdje x predstavlja točku u N-dimenzionalnom prostoru, a funkcija d(x,y) predstavlja funkciju udaljenosti između dvije točke u prostoru. Za funkciju udaljenosti najčešće se koristi Euklidska udaljenost čija formula je sljedeća:

$$\sqrt{\{(x1 - y1)^2 + (x2 - y2)^2 + (x3 - y3)^2 + \cdots + (xN - yN)^2\}}$$

Osim euklidske udaljenosti za udaljenost točaka još se može koristiti i formula za izračun Manhattan udaljenosti:

$$|x1 - y1| + |x2 - y2| + |x3 - y3| + \cdots + |xN - yN|$$

S obzirom na odabir formule za izračun udaljenosti između dvije točke, poligoni ćelija Voronoijevih dijagrama su također drugačiji. Na slici (Slika 11) prikazana je razlika Voronijevih dijagrama uz računanje udaljenosti između dvije točke korištenjem Euklidske udaljenosti i korištenjem formule za izračuna Manhattan udaljenosti.



Slika 11 Prikaz Voronoijevih dijagrama uz različite funkcije udaljenosti, izvor: [11]

2.2. Implementacija Voronoijevih dijagrama

Voronoijevi dijagrami često se koriste kao heuristika umjetne inteligencije u računalnim igrama. Heuristika je način rješavanja problema u kojem je cilj pronaći rješenje problema unutar razumnog vremenskog okvira nauštrb savršenog rješenja. Kod heurističkog pristupa naglasak je na brzini, a ne na točnosti rezultata [5]. Ključ dobre heuristike je upravo brzina rješavanja problema. Prva ideja rješenja problema Voronoijevih dijagrama jest za svaku točku, površine koju analiziramo, izračunati udaljenost do referentnih točaka pa onda odrediti kojoj referentnoj točki je najbliža. Takva vrsta pristupa nije zahtjevna za izračun sve dok je površina, nad kojoj je potrebno izračunati Voronoijev dijagram, mala i dok je broj referentnih točaka relativno malen.

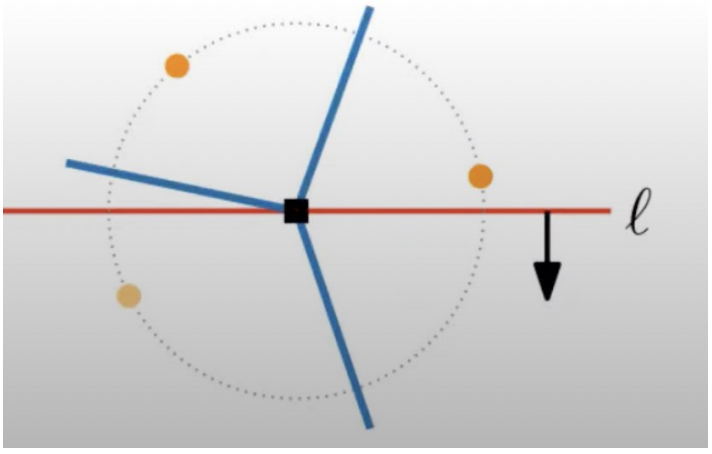
```
for(point in plane) {
    for(site in sites) {
        float number = distance(point, site);
        if(Objects.isNull(voronoi[point])) {
            voronoi[point] = site;
        }

        if(number < distance(point, voronoi[point])) {
            voronoi[point] = site;
        }
    }
}
```

Isječak koda 2 Pseudokod izračuna Voronoijevih dijagrama naivnom metodom

Složenost ovakvog naivnog pristupa je O(p*s), gdje je p broj točaka, a s je broj referentnih točaka [20]. U nastavku je opisan Fortune algoritam koji je manje kompleksan, razvio ga je Steven Fortune i objavio ga u svom radu „A sweepline algorithm for Voronoi diagrams“ 1986.godine [1].

2.3. Fortune algoritam

Fortune algoritam, nije neefikasan kao naivan algoritam iz prošlog poglavlja, već svoje temelje izvlači iz algoritma pomicanja linija. Fortune algoritam uzima svaku referentnu točku i sprema je u listu događaja. Događaj u Fortune algoritmu je indikacija da će se nešto promijeniti u izgledu Voronoijevih dijagrama. Dvije su vrste događaja, jedan događaj je kada linija naiđe na referentnu točku, a drugi događaj je kada linija naiđe na Voronoijev vektor. Voronijev vektor je točka koja je jednako udaljena od tri najbliže referentne točke (Slika 12).



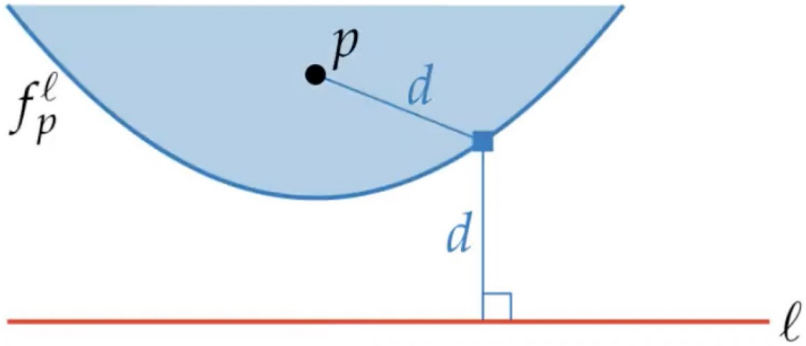
Slika 12 Primjer Voronoijevog vektora

Osim događaja, pratimo i ćelije koje se dalje mogu razvijati, odnosno one koje nisu okružene s drugim ćelijama. Fortune algoritam definira pojam za ćelije koje još mogu rasti, a to je „beachline“. Beachline predstavlja točke površine koje još moramo analizirati i odrediti kojoj Voronoijevoj ćeliji pripadaju.

Slika 13 Primjer beachline-a u Fortune algoritmu

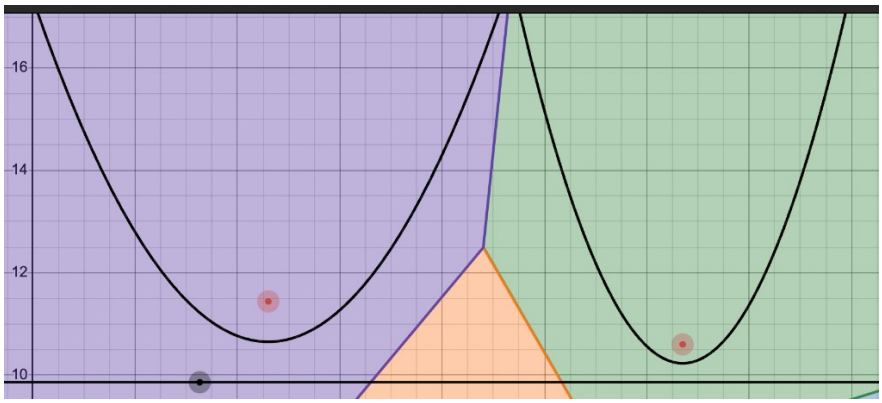
Ćelije koje „rastu“ predstavljene su parabolama. Parabola Voronoijevih ćelija konstruira se uzimajući referentnu točku kao fokus parabole i „sweep line“ kao directrix parabole. Directrix je linija koja je okomita na simetralu parabole i ne dodiruje samu parabolu. Tako definirana parabola na svojim rubovima daje točke koje su jednako udaljene od fokusa parabole i od horizontalne linije (engl. *directrix)* parabole. Upravo to zadovoljava zahtjev Voronoijevih ćelija. Voronoijeva ćelija okružuje točke čija je udaljenost između referentne točke te ćelije manja od udaljenosti do drugih referentnih točaka. S obzirom na to da Fortune algoritam sortira događaje tako da ih posjećuje od vrha prema dnu, kada definiramo parabolu sigurni smo da će ona u konačnici tvoriti jednu Voronoijevu ćeliju. Formula za računanje parabole pomoću fokusa i directrixa je dana u nastavku:

$$y = \frac{1}{2(yf - yd)}(x - xf)^2 + \frac{yf + yd}{2}$$



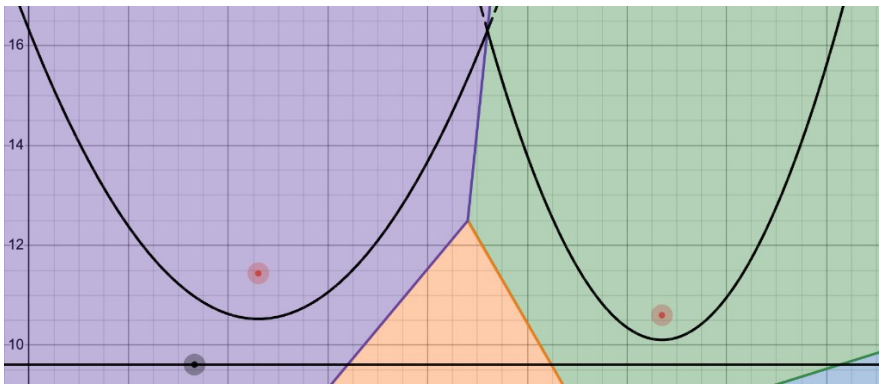
Slika 14 Prikaz matematičke pozadine ćelija u Voronoijevu dijagramu

Pomicanje linije od jedne referentne točke do druge uzrokuje da već konstruirane parabole rastu i mijenjaju svoj oblik. S obzirom da postoji više parabola, događa se da se parabole počnu presijecati. Upravo te točke presjeka parabola definiraju rubove Voronoijevih ćelija. Točke na rubovima Voronoijevih ćelija jednako su udaljene od dvije referentne točke pa je to i razlog zašto se Voronoijev rub dobiva presijecanjem parabola kako one rastu. Na primjeru slike (Slika 15) prikazane su dvije parabole. Jedna koja u konačnici tvori ljubičastu Voronoijevu ćeliju i drugu koja u konačni tvori zelenu Voronoijevu ćeliju. Sve točke koje su iznad parabole u ljubičastoj Voronoijevoj ćeliji manje su udaljene do referentne točke u ljubičastoj ćeliji nego do referentne točke u zelenoj ćeliji. U ovom trenutku parabole još nisu došle do točke presjeka, to se događa na slici (Slika 18) gdje je vrlo lako primijetiti točku presjeka ove dvije parabole.



Slika 15 Prikaz parabola koje tvore Voronoijeve ćelije, [3]

Na slici (Slika 16) točka presjeka jednako je udaljena od referentne točke u ljubičastoj ćeliji kao i od referentne točke u zelenoj ćeliji. Upravo je to svojstvo Voronoijevog ruba pa se i iz tog razloga svaka točka presjeka parabola tumači kao točka ruba Voronoijeve ćelije.



Slika 16 Prikaz točke presjeka dviju parabola Voronoijevih ćelija, [3]

Vremenska složenost Fortune algoritma jest O(nlog(n)), a memorijska složenost je O(n).

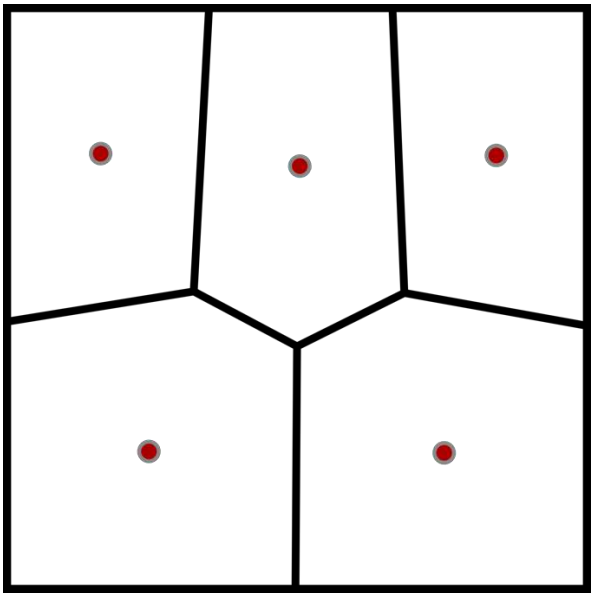
2.4. Lloydov algoritam relaksacije

Rezultat Fortune algoritma jesu Voronoijeve ćelije koje nerazmjerno prekrivaju površinu. Ovisno o slučaju korištenja Voronoijevih dijagrama za analizu problema, postoji mogućnost za potrebom jednolikog prekrivanja površine Voronoijevim ćelijama. S dva različita problema vrlo lako je shvatiti primjenu jednog i drugog algoritma. Prvi problem definiran je kao problem vatrogasnih postaja. U gradu je N vatrogasnih postaja, računalni program prilikom dojave požara mora obavijestiti najbližu vatrogasnu postaju da izađe na teren i ugasi požar. Drugi problem je problem broja senzora za očitavanje vlažnosti tla. U ovom slučaju potrebno je izračunati najmanji broj senzora koji će se postaviti na poljoprivredno tlo. Senzori imaju neka specifična svojstva koja sprječavaju da se postavi samo jedan senzor na tlo. Svaki senzor definira površinu u kvadratnim metrima za koju može bez pogreške izmjeriti vlažnost tla te ne može komunicirati s drugim senzorima. Rješenje prvog problema vrlo je jednostavno rješivo s normalnim Fortune algoritmom uz Mahnattan udaljenost kao funkcijom udaljenosti. Na slici (Slika 17) prikazano je rješenje problema. Uvijek će jedna vatrogasna postaja biti najbliže požaru.

Slika 17 Rješenje problema vatrogasnih postaja, izvor: [22]

Drugi problem zahtijeva malo unaprjeđenje običnog Fortune algoritma. Naime, običan Fortune algoritam zahtijeva početne referentne točke od kojih može konstruirati Voronoijev dijagram, a definicija drugog problema zahtijeva broj potrebnih senzora, a ne područja koja pokrivaju referentne točke. Za početak moguće je uzeti neki nasumičan broj senzora n, neka je u ovom slučaju n = 5 i nasumično odabrati njihove lokacije na površini koju analiziramo. Običan Fortune algoritam konstruirat će nepravilan Voronoijev dijagram, a ono što bi potencijalno moglo biti rješenje problema je struktura koja se zove centralni Voronoijev dijagram. Centralni Voronoijev dijagram je optimalan raspored Voronoijevih ćelija,

odnosno raspored ćelija takav da sve ćelije pokrivaju približno jednak prostor površine. Upravo to je rezultat algoritma koji nosi naziv Lloydov algoritam relaksacije.

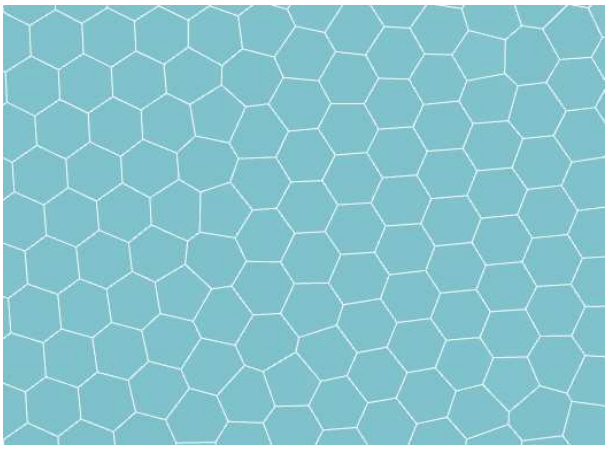


Slika 18 Centralni Voronoijev dijagram, izvor: [9]

Za razliku od Fortune algoritma gdje se jednom definirane referentne točke ne mijenjaju, Lloydov algoritam se temelji na pomicanju referentnih točaka prema centru mase Voronoijeve ćelije. Koraci Lloydovog algoritma su sljedeći:

- Izračunaj Voronoijeve ćelije
- Izračunaj centar mase svake Voronoijeve ćelije
- Pomakni referentnu točku na centar mase svake Voronoijeve ćelije

Iz iteracije u iteraciju ovog algoritma, Voronoijeve ćelije postaju sve sličnije svojom površinom te nakon divergencije Voronoijev dijagram postaje kao dijagram sa slike (Slika 19).

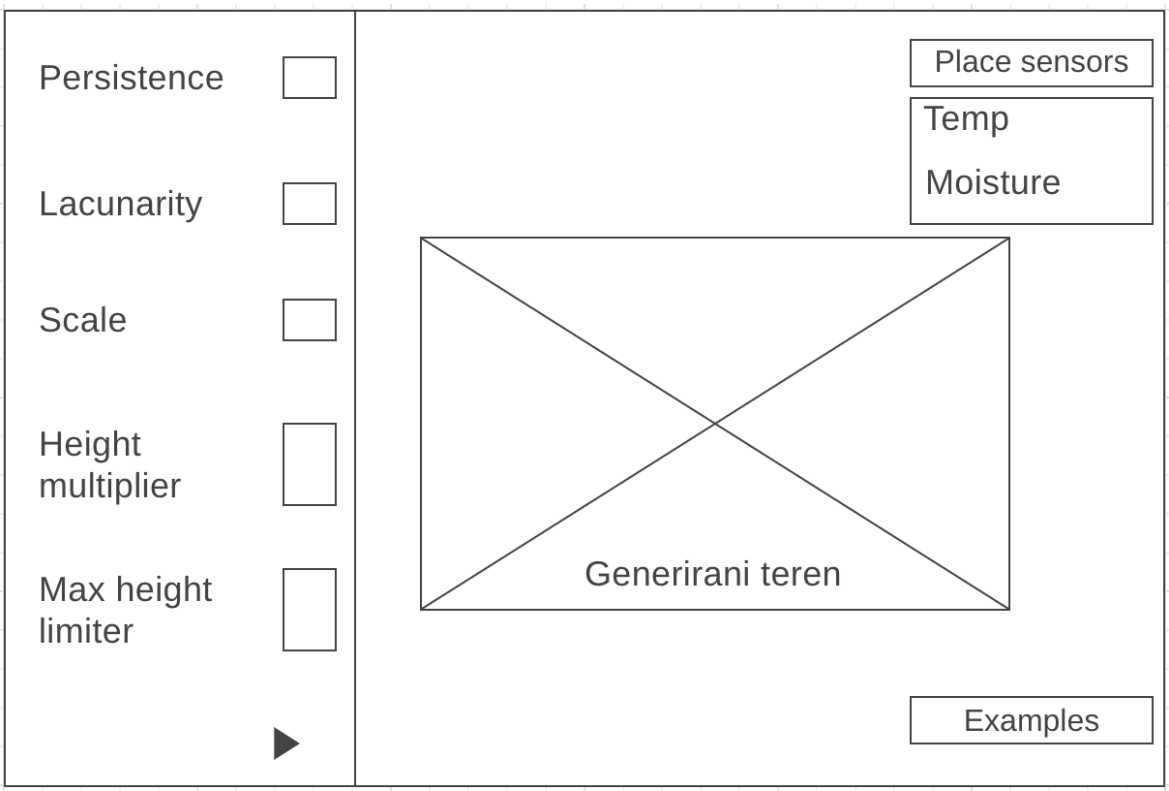


Slika 19 Centralni Voronoijev dijagram, izvor: [10]

Vremenska složenost Lloydovog relaxation algoritma jest O($n * nlog(n)$), jer moramo n puta izvršavati Fortune algoritam.

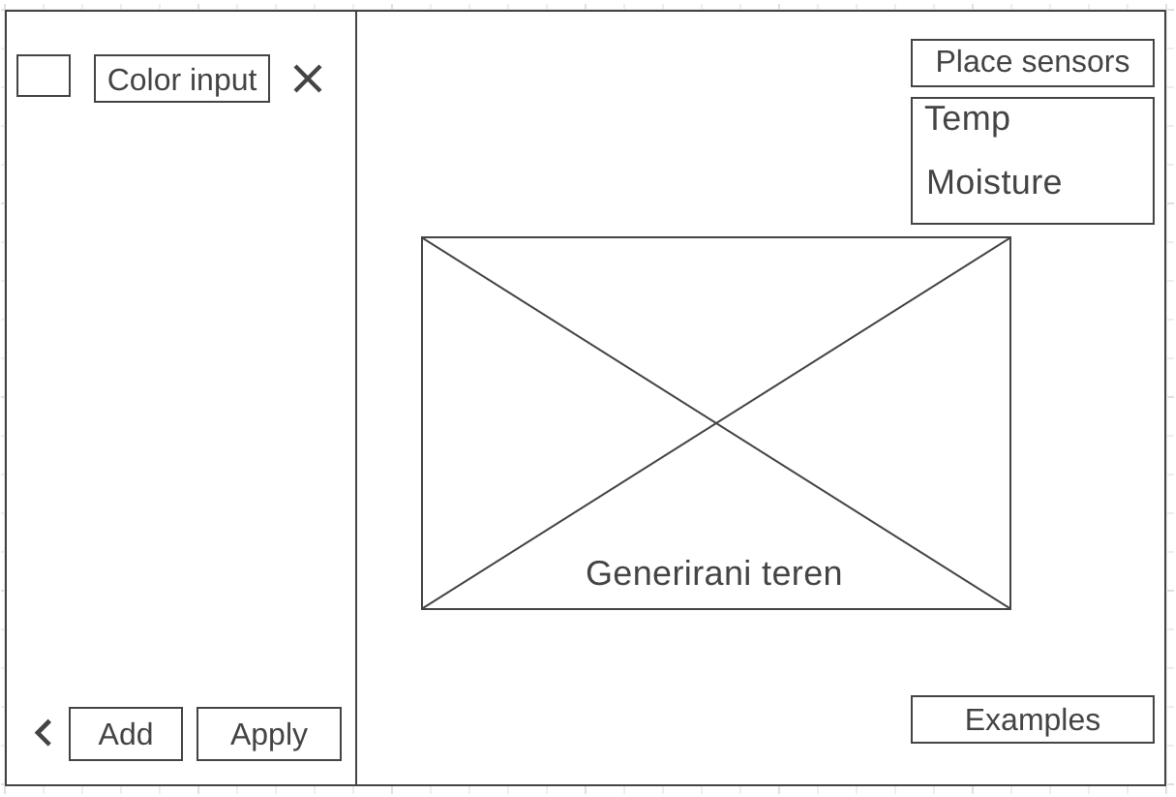
3. Dizajn implementacijskog rješenja

Implementacijsko rješenje mora biti predstavljeno kroz sučelje, koje korisniku daje upravljačku mogućnost nad generiranjem terena. Rješava se problem proceduralnog generiranja terena i objekata. Korisnik mora moći upravljati izgledom terena te isto tako utjecati na proceduralno generiranje objekata zbog drugačije generiranog terena. Konačno rješenje sačinjavat će tri ekrana te će na svakom ekranu u fokusu biti generirani teren. Ekran 1 prikazuje inicijalne opcije koje korisnik može kontrolirati. S desne strane nalazi se generirani teren, a s lijeve strane nalaze se opcije koje su implementirane kao klizači te sa svoje desne strane imaju polje za unos koje prikazuje vrijednost klizača. Isto tako unošenjem vrijednosti u polje za unos kontrolira se vrijednost klizača.



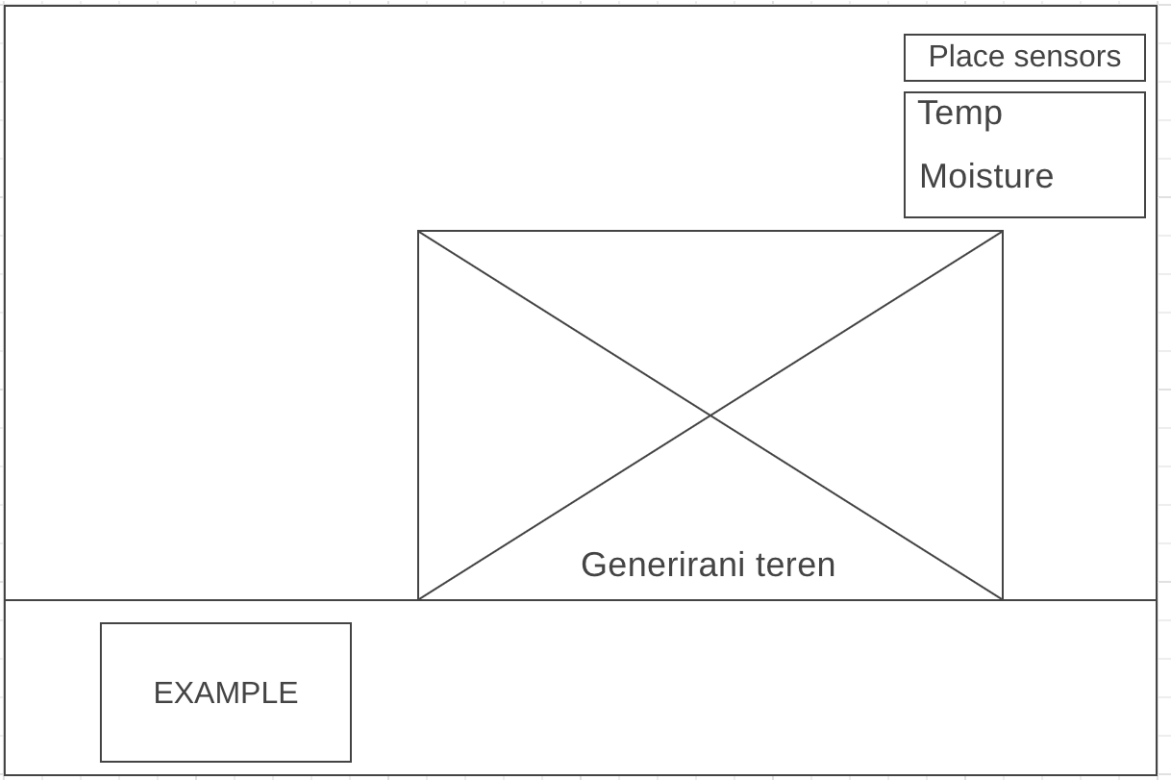
Slika 20 Ekran 1 korisničkog sučelja

Na ekranu 2 mijenja se jedino prikaz opcija, koje sad umjesto opcija Perlinove funkcije, prikazuju vrijednosti boje terena. Panel s opcijama za boje sastoji se od više polja za unos boja te gumba za dodavanje novih polja za unos i gumba za primjenjivanje boja na generirani teren.



Slika 21 Ekran 2 korisničkog sučelja

Ekran 3 prikazuje panel s predefiniranim primjerima čijom primjenom se generirani teren mijenja prema opcijama koje su zadane u predefiniranom primjeru. Korisnik će moći vidjeti popis predefiniranih primjera te sliku svakog primjera koja mu indicira kako će generirani teren izgledati.



Slika 22 Ekran 3 korisničkog sučelja

3.1. Akcije

U nastavku su popisane akcije, koje su podijeljene u tri grupe. Prva grupa su akcije navigacije po korisničkom sučelju, druga grupa su akcije koje utječu na generiranje terena i treće su akcije kojima se upravlja generiranim terenom.

Akcija	**Rezultat akcije**
Klik na gumb 'Examples'	Sakriva se panel s opcijama i pojavljuje se panel koji sadrži već generirane primjere terena.
Klik na gumb za navigaciju koji se nalazi u donjem dijelu ekrana s opcijama	Ako se korisnik nalazi na ekranu s opcijama šuma tada ova akcija rezultira prikazom ekrana s unosom boja, a inače je obrnuto.

Tablica 1 Grupa akcija navigacije po korisničkom sučelju

Druga grupa akcija uključuje prva dva ekrana. Na prvom ekranu (Slika 20) s lijeve strane su popisane opcije kojima se upravlja parametrima za generiranje terena, svaka od tih opcija je kombinacija klizača (engl. *slider*) i tekstualnog unosa. Na drugom ekranu (Slika 21) prikazane su opcije s odabirom boja koje su tekstualni unos.

Akcija	**Rezultat akcije**
Pomicanje vrijednosti klizača „*Persistence*“	Ponovno generiranje terena te postavljanje odgovarajućeg tekstualnog unosa na vrijednost klizača
Upisivanje vrijednosti u tekstualni unos u polje „Persistence“	Ponovno generiranje terena te postavljanje upisane vrijednosti klizača
Pomicanje vrijednosti klizača „*Lacunarity*“	Ponovno generiranje terena te postavljanje odgovarajućeg tekstualnog unosa na vrijednost klizača
Upisivanje vrijednosti u tekstualni unos u polje „*Lacunarity*“	Ponovno generiranje terena te postavljanje upisane vrijednosti klizača
Pomicanje vrijednosti klizača „*Scale*“	Ponovno generiranje terena te postavljanje odgovarajućeg tekstualnog unosa na vrijednost klizača
Upisivanje vrijednosti u tekstualni unos u polje „*Scale*“	Ponovno generiranje terena te postavljanje upisane vrijednosti klizača
Pomicanje vrijednosti klizača „*Height multiplier*“	Ponovno generiranje terena te postavljanje odgovarajućeg tekstualnog unosa na vrijednost klizača
Upisivanje vrijednosti u tekstualni unos u polje „*Height multiplier*“	Ponovno generiranje terena te postavljanje upisane vrijednosti klizača
Pomicanje vrijednosti klizača „*Max height limiter*“	Ponovno generiranje terena te postavljanje odgovarajućeg tekstualnog unosa na vrijednost klizača

Upisivanje vrijednosti u tekstualni unos u polje „Max height limiter“	Ponovno generiranje terena te postavljanje upisane vrijednosti klizača
Klik gumba „Add“ na ekranu s bojama	Dodavanje novog unosa za boju
Unos boje u tekstualni unos „Color Input“	Promjena boje tekstualnog unosa da odgovara upisanoj boji
Klik gumba „Apply“ na ekranu s bojama	Primjena boja na generirani teren

Tablica 2 Grupa akcija koje utječu na generiranje terena

Treća i posljednja grupa akcija sastoji se od jedne akcije kojom je moguće upravljati generiranim terenom i akcije postavljanja senzora.

Akcija	**Rezultat akcije**
Pomicanje terena pomoću miša	Rotacija generiranog terena
Klik na gumb „Place sensors“	Generira N senzora te ih postavlja po terenu

Tablica 3 Grupa akcija kojima se upravlja generiranim terenom

4. Razvoj implementacijskog rješenja

4.1. Razvojna okolina

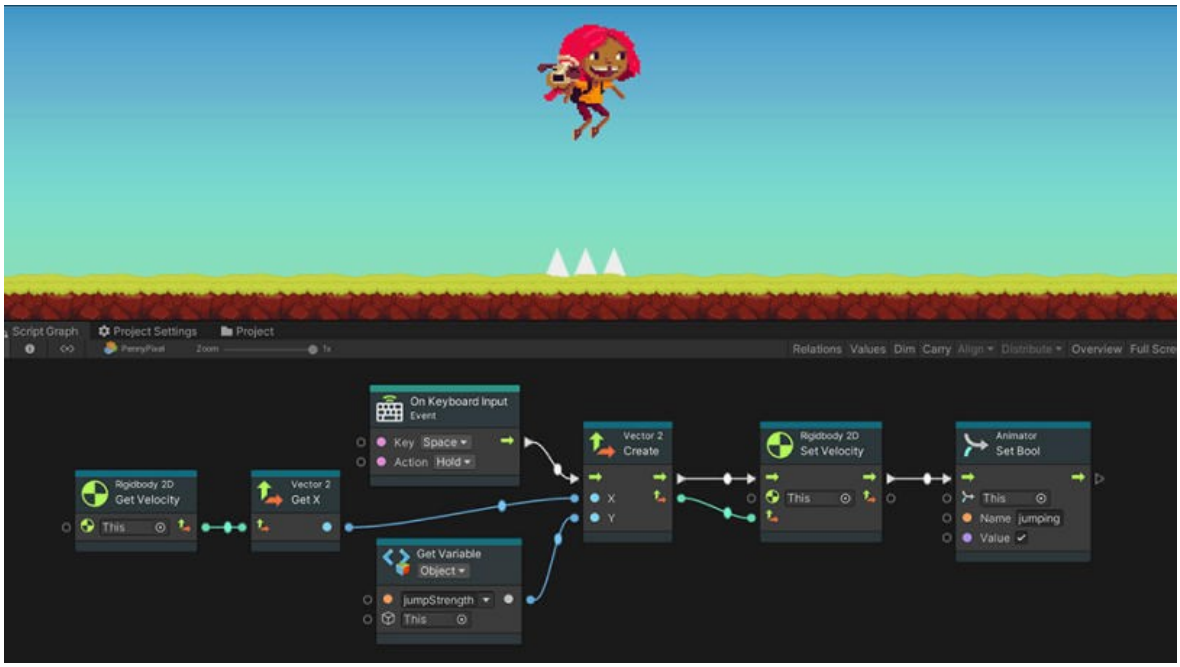
Rješenje problema implementirano je na osobnom računalu sljedećih specifikacija: 16GB radne memorije, procesor Apple M1 Pro s 10 jezgri uz integriranu grafičku od 16 jezgri. Operacijski sustav na osobnom računalu jest macOS Monterey verzije 12.5. Za potrebe razvojnog okruženja korišten je Unity verzije 2020.3.25f1.

4.2. Alati

S obzirom na to da se problem generiranja terena rješava pomoću funkcija šuma koje su striktno matematički zadane i predstavljene kao brojevi, potrebno je osmisliti način na koji bi se funkcija šuma mogla vizualizirati. Za potrebe vizualizacije zaista postoji mnogo mogućnosti, primjerice dvodimenzionalni grafovi, kreiranje tekstura za objekte, no za potrebe ovog rada potrebno je generirati trodimenzionalni teren. S obzirom na to da bi se takav teren mogao također koristiti u računalnim igrama, za razvoj implementacijskog rješenja odabran je programski alat Unity.

4.3. Unity

Unity je softverski okvir dizajniran za izradu računalnih igara. Besplatan je za korištenje, ako se ne koristi u komercijalne svrhe. Ima podršku za nekoliko desetaka platformi, uključujući Android, PC igre, igre za konzole, igre u virtualnoj stvarnosti. Prema blogu [14] uključen je među sedam najboljih programa za izradu računalnih igara. U Unityju kontrola toka igre događa se kroz primjenjivanje komponenti na objekte. Objekti mogu biti trodimenzionalni ili dvodimenzionalni, ovisno o implementaciji računalne igre. Komponente koje se primjenjuju na objekte, mogu biti odabrane iz liste postojećih komponenti koje dolaze zajedno u paketu s Unityjem te rješavaju problem primjerice fizike objekta, poput padanja ili rješavaju problem kretanja igrača i usklađivanja animacija kretanja s položajem objekta u virtualnom okruženju. Ako se razvojni programer susretne s nekim problem koji ne može riješiti pomoću već postojeće komponente, također može kreirati svoju komponentu. Kreiranje komponenti može biti izvedeno kroz pisanje programskog koda u objektno orijentiranom jeziku C# ili kroz vizualno programiranje uz Bolt.

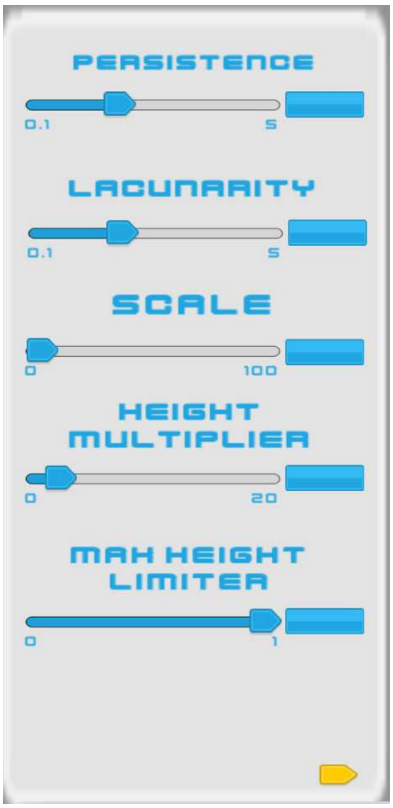


Slika 23 Vizualno programiranje u Unityu, izvor: [18]

4.4. Unity UI

Bitna komponenta implementacijskog rješenja je korisničko sučelje kojim će se moći promijeniti konfiguracija terena. Za takve potrebe iskorištene su komponente koje Unity pruža u svojoj UI knjižnici. Korisničko sučelje je skup elemenata kojim korisnik interaktira kako bi se kretao po aplikaciji ili utjecao na nekakve događaje u aplikaciji. Unity UI je korisničko sučelje temeljeno na komponentama [17], razvijene su posebne komponente koje se mogu koristiti samo za stvaranje korisničkog sučelja poput gumba, klizača, polja za unos i drugo. Sve komponente koje su dio korisničkog sučelja igre moraju biti smještene na komponenti Canvas. Kako bi se elementi lakše rasporedili, Canvas komponenta prikazana je kao pravokutnik, te se sve komponente, koje su smještene na Canvas, prikazuju s pozicijom relativnom na Canvas komponentu.

4.5. Izgled korisničkog sučelja u implementacijskom rješenju



Slika 24 Izgled korisničkog sučelja, panel s opcijama koje utječu na vrijednosti šuma

Za potrebe izgleda korisničkog sučelja korišten je paket besplatnih uređenih komponenti pronađen na poveznici: *https://www.kenney.nl/assets/ui-pack*. Kako bi korisnik mogao interaktirati s korisničkim sučeljem, potrebno je definirati događaje/akcije koje će rezultirati nekim posljedicama u vidu interkacije s korisničkim sučeljem. Akcije i rezultati popisani su u poglavlju 4.1, stranica 22. Korištene komponente za interakciju s korisničkim sučeljem su klizači (engl. *slider*), tekstualni unos i gumbi. Svaka od tih komponenta ima definirane slusače (engl. *listener*) koji na određeni događaj pozivaju definirane funkcije. Klizač tako ima listener „On Value Changed“ kojim je poznato da se dogodila promjena na unosu i koja je nova vrijednost tog unosa. U korisničkom sučelju implementacijskog rješenja svaka promjena klizača uzrokuje i promjenu na odgovarajućem tekstualnom unosu s desne strane klizača.

```
public void Change_Lacunarity(float lacunarity)
{
    noiseTexture.lacunarity = lacunarity;
    noiseTexture.generateElevatedTexture();
    lacunarityInput.text = "" + lacunarity;
}
```

Isječak koda 3 Funkcija koja se poziva kada se promijeni vrijednost na klizaču


```
public void Change_Lacunarity(float lacunarity)
{
    noiseTexture.lacunarity = lacunarity;
    noiseTexture.generateElevatedTexture();
    lacunarityInput.text = "" + lacunarity;
}
```

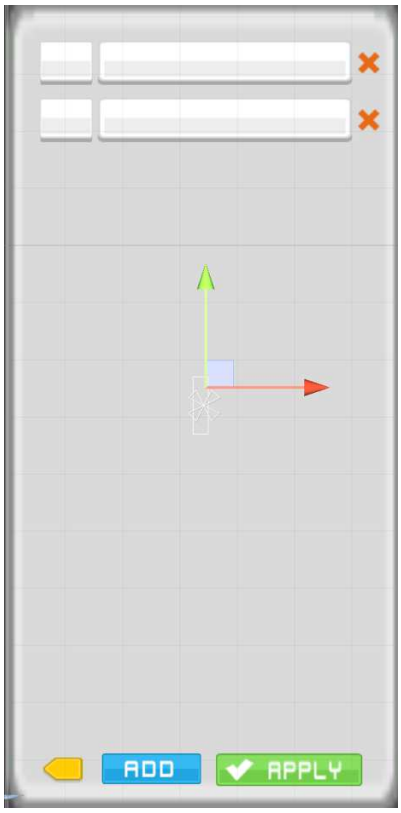

Na slici

Isječak koda 3 prikazan je princip promjene vrijednosti kada se okine događaj za promjenu vrijednosti klizača, na istom principu implementirane su i funkcije za promjenu vrijednosti *lacunarity, height multiplier, scale i max height limiter*.

Isto tako vrijednosti se mogu promijeniti upisom željene vrijednosti u tekstualni unos, upis u tekstualni unos okida događaj 'On End Edit' koji zatim poziva funkciju za promjenu vrijednosti tekstualnog unosa i klizača.

```
public void Change_Lacunarity_Input(string lacunarity)
{
    float temp = float.Parse(lacunarity);
    noiseTexture.lacunarity = temp;
    noiseTexture.generateElevatedTexture();
    lacunaritySlider.value = temp;
}
```

Isječak koda 4 Funkcija koja se poziva kada se promijeni vrijednost u tekstualnom unosu



Slika 25 Izgled korisničkog sučelja, panel s bojama koje utječu na boju određenih dijelova terena

Ako klasični slušači ne zadovoljavaju potrebe, također je moguće implementirati i vlastite. Za potrebe unosa boje, iskorišten je pristup direktnog pozivanja željene funkcije, koja mijenja boju samog tekstualnog unosa, kada korisnik završi s unosom.

```
void Start()
{
    inputField.onEndEdit.AddListener(delegate{ChangeColor(inputField);});
}

void ChangeColor(InputField inputField)
{
    Color color = new Color();
    if(ColorUtility.TryParseHtmlString(inputField.text, out color))
    {
        inputField.image.color = color;
    }
}
```

Isječak koda 5 Funkcija koja mijenja boju tekstualnog unosa s obzirom na unesenu boju

Na ekranu s bojama postoje i dva vrlo bitna gumba, Add i Apply. Gumbom Add korisnik može dodati novo tekstualno polje za unos boje. Za dodavanje novog tekstualnog polja potrebno je izračunati poziciju novo dodanog polja. Pozicija se izračunava relativno na vršnu komponentu koja sadrži svaki tekstualni unos.

```
public void Add_Button_Clicked()
{
    InputField[] inputFields = listOfColors.GetComponentsInChildren<InputField>();
    if(inputFields.Length < 24)
    {
        GameObject gameObject = Instantiate(colorInputPrefab, new Vector3(0, 0, 0), new Quaternion());
        gameObject.transform.parent = listOfColors.transform;

        gameObject.transform.localScale = new Vector3(10.29651f, 2.5877f, 1f);
        Vector3 position =
            listOfColors.transform.position +
            new Vector3(43f, (inputFields.Length/2 + 1)*-100, 0);
        gameObject.transform.position = position;
    }
}
```

Isječak koda 6 Funkcija za dodavanje novog tekstualnog unosa

Broj tekstualnih unos ograničen je na dvanaest, a funkcija provjerava postoji li dvadeset i četiri polja za unos zato što polje za unos boje također sadrži još jedno tekstualno polje. Gumbom Apply unesene boje primjenjuju se na generirani teren. Svaka tekstualni unos za boju uz sebe sadrži i tekstualni unos za unos decimalnog broja koji reprezentira granicu u intervalu [0,1] ispod koje će se primijeniti unesena boja. Razlog tog intervala krije se u definiciji funkcije šuma koja vraća vrijednosti upravo u tom intervalu.

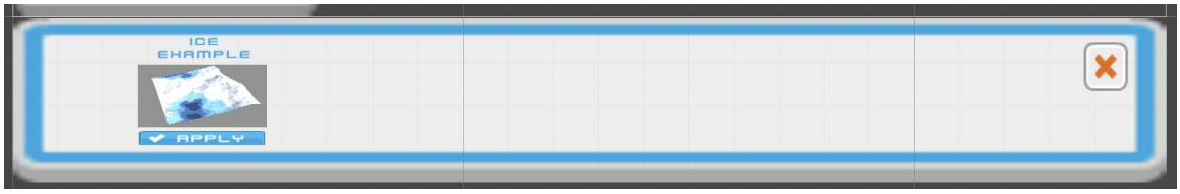
```
public void Apply_Colors_Clicked()
{
    InputField[] inputFields = listOfColors.GetComponentsInChildren<InputField>();
    List<TextureColors> textureColors = new List<TextureColors>();
    for(int i = inputFields.Length - 1; i > 0; i -= 2)
    {
        string value = inputFields[i].text;
        Color color = inputFields[i - 1].image.color;
        if(!value.Equals(""))
        {
            textureColors.Add(new TextureColors(value, color));
        }
    }
    textureColors.Sort((t1, t2) => t1.height.CompareTo(t2.height));
    noiseTexture.textureColors = textureColors.ToArray();
    noiseTexture.generateElevatedTexture();
}
```

Isječak koda 7 Funkcija koja primjenjuje unesene boje i njihove granice na generirani teren

Unesene boje i njihove granice čine strukturu TextureColors, koja sadrži boju i granicu te boje kako bi se kasnije mogla definirati boja teksture generiranog terena. Struktura također sadrži funkciju generiranja polja boja za teksturu i vraća izlaz koji se može primijeniti na teksturu generiranog terena.

```
[Serializable]
public class TextureColors
{
    public float height;
    public Color color;

    public TextureColors(string value, Color color)
    {
        this.height = float.Parse(value);
        this.color = color;
    }

    public static Color[,] generateColorMap(
        int height,
        int width,
        TextureColors[] textureColors,
        float[,] noiseValues)
    {
        Color[,] colorMap = new Color[height, width];
        for (int i = 0; i < width; i++)
        {
            for (int j = 0; j < height; j++)
            {

                for (int colorIndex = 0; colorIndex < textureColors.Length; colorIndex++)
                {
                    if (noiseValues[i, j] < textureColors[colorIndex].height)
                    {
                        colorMap[i, j] = textureColors[colorIndex].color;
                        break;
                    }
                }
            }
        }
        return colorMap;
    }
}
```

Isječak koda 8 Struktura TextureColors



Slika 26 Izgled korisničkog sučelja, ekran s unaprijed definiranim primjerima

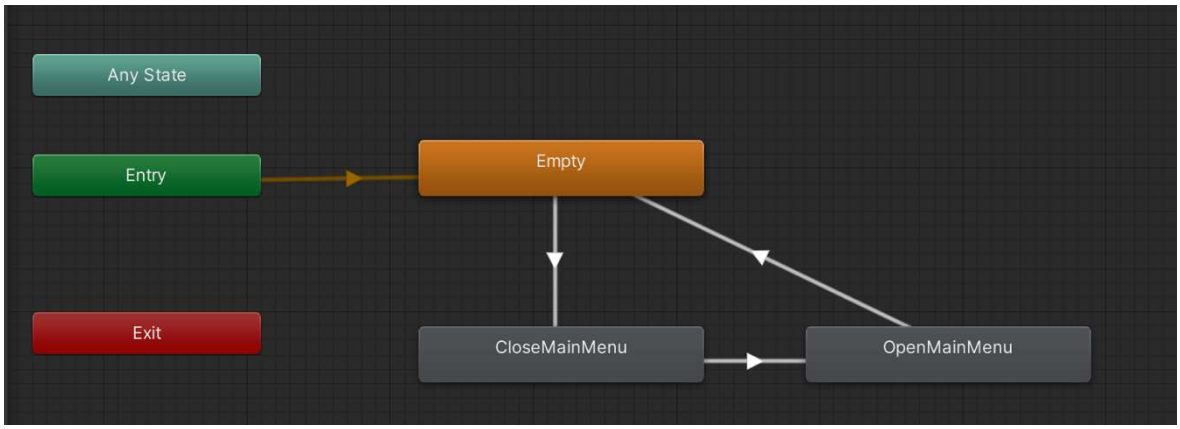
Unaprijed definirani primjeri sadrže sve informacije koje su potrebne kako bi se generirao potpuno isti teren, što znači da su tu uključeni svi parametri potrebni za izgradnju generiranog terena pomoću Perlinovog šuma: *persistence, lacunarity, scale, max heigth limiter i height multiplier*. Pritiskom na gumb „Apply“, ispod same slike generiranog primjera, korisniku se generira teren prema definiranim parametrima te se vrijednosti parametara postavljaju na korisničkom sučelju, u odgovarajuća polja za unos.

```
public class ApplyPredefinedExample : MonoBehaviour
{
    public PredefinedExample predefinedExample;
    public NoiseTexture noiseTexture;
    public GameObject colorInputs;
    public Slider persistenceSlider;
    public Slider lacunaritySlider;
    public Slider scaleSlider;
    public Slider heightMultiplierSlider;
    public Slider maxHeightLimiterSlider;
    public InputField persistenceInput;
    public InputField lacunarityInput;
    public InputField scaleInput;
    public InputField heightMultiplierInput;
    public InputField maxHeightLimiterInput;
    public GameObject colorInputPrefab;

    public void Apply_Example_Clicked()
    {
        noiseTexture.lacunarity = predefinedExample.lacunarity;
        noiseTexture.persistance = predefinedExample.persistence;
        noiseTexture.heightMultiplier = predefinedExample.heightMultiplier;
        noiseTexture.noiseHeightMaxValue = predefinedExample.maxHeightLimiter;
        noiseTexture.scale = predefinedExample.scale;
        noiseTexture.textureColors = predefinedExample.textureColors;
        noiseTexture.generateElevatedTexture();

        persistenceSlider.value = predefinedExample.persistence;
        lacunaritySlider.value = predefinedExample.lacunarity;
        scaleSlider.value = predefinedExample.scale;
        heightMultiplierSlider.value = predefinedExample.heightMultiplier;
        maxHeightLimiterSlider.value = predefinedExample.maxHeightLimiter;
        persistenceInput.text = "" + predefinedExample.persistence;
        lacunarityInput.text = "" + predefinedExample.lacunarity;
        scaleInput.text = "" + predefinedExample.scale;
        heightMultiplierInput.text = "" + predefinedExample.heightMultiplier;
        maxHeightLimiterInput.text = "" + predefinedExample.maxHeightLimiter;

        InputField[] inputFields = colorInputs.GetComponentsInChildren<InputField>();
        TextureColors[] textureColors = noiseTexture.textureColors;

        if (textureColors.Length > 2 && (inputFields.Length / 2) < textureColors.Length)
        {
            for (int i = (inputFields.Length / 2); i < textureColors.Length; i++)
            {
                GameObject gameObject = Instantiate(colorInputPrefab, new Vector3(0, 0, 0), new Quaternion());
                gameObject.transform.parent = colorInputs.transform;

                gameObject.transform.localScale = new Vector3(10.29651f, 2.5877f, 1f);
                Vector3 position = colorInputs.transform.position + new Vector3(43f, (i+1) * -100, 0);
                gameObject.transform.position = position;
            }

        }

        inputFields = colorInputs.GetComponentsInChildren<InputField>();
        Debug.Log(Color.cyan);
        for(int i = 0; i < textureColors.Length; i++)
        {
            inputFields[i * 2 + 1].text = "" + textureColors[i].height;
            inputFields[i*2].image.color =
                new Color(
                    textureColors[i].color.r,
                    textureColors[i].color.g,
                    textureColors[i].color.b
                );
        }
```

Isječak koda 9 Funkcija koja se poziva kada se primjenjuje predefinirani generirani teren

Konačno, potrebno je izvesti prijelaz između panela s opcijama i panela s primjerima. Za te potrebe postoji gumb „Examples“ na glavnom ekranu te gumb za uklanjanje panela s primjerima na panelu s primjerima. Klikom na gumb „Examples“ ili gumb za uklanjanje panel s primjerima aktivira se jedna od definiranih animacija. Definirana animacija mijenja položaj pojedinog panela ovisno o događaju pa tako pritisak na gumb „Examples“ aktivira

animaciju koja mijenja poziciju panela s opcijama ulijevo kako ga korisnik više ne bi vidio te mijenja poziciju panela s primjerima kako bi korisnik mogao odabrati primjer. Za upravljanje animacijama korišteni su animation upravljači Unity programskog alata. Animation upravljač je zaslužan za kontroliranje stanja animacije ili pojedinog objekta, a definiran je poput automata s konačnim brojem stanja gdje postoje određeni prijelazi iz jednog stanja u drugo. Prijelazi stanja se kontroliraju pomoću uvjeta, a uvjete prijelaza obilježavaju parametri.



Slika 27 Animation controller animacije zatvaranja panela s opcijama

Primjerice, na slici (Slika 27), prijelaz iz stanja „Empty“ u stanje „CloseMainMenu“ definirano je uvjetom da je parametar shouldCloseMenu (tip *boolean*) true.

Slika 28 Primjer uvjeta za prijelaz iz jednog stanja u drugo stanje

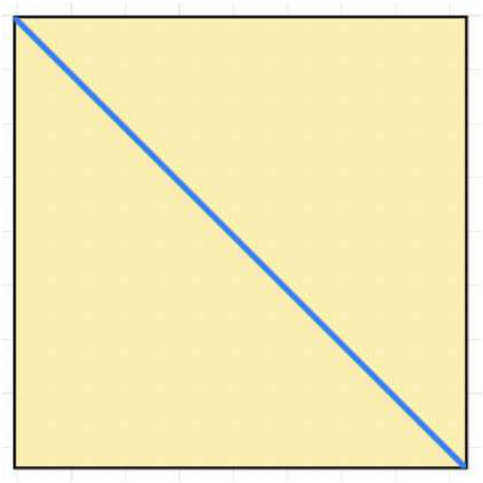
Stanja indiciraju koja animacija bi se trebala odvijati u tom stanju pa tako se ta animacija može i ponavljati, a može biti prikazana samo jednom. U slučaju animacije korisničkog sučelja, dovoljno je da je animacija prikazana samo jednom. Animacija korisničkog sučelja zapravo je podijeljena u dva odvojena događaja spojena u jedan. Prvi je zatvaranje ili otvaranje panela s opcijama, a drugi je otvaranje ili zatvaranje panela s primjerima.

```
public void showExamplesClicked()
{
    Animator animator = mainMenuPanel.GetComponent<Animator>();
    if(animator != null)
    {
        animator.SetBool("shouldCloseMenu", true);
    }

    Animator animator1 = examplesPanel.GetComponent<Animator>();
    if(animator1 != null)
    {
        animator1.SetBool("shouldOpenExamplesMenu", true);
    }

}

public void backArrowOnExamplesPanelClicked()
{
    Animator animator1 = examplesPanel.GetComponent<Animator>();
    if (animator1 != null)
    {
        animator1.SetBool("shouldOpenExamplesMenu", false);
    }
    Animator animator = mainMenuPanel.GetComponent<Animator>();
    if (animator != null)
    {
        animator.SetBool("shouldCloseMenu", false);
    }
}
```

Slika 29 Funkcije koje kontroliraju promjenu animacija korisničkog sučelja

4.6. Unity Mesh

S obzirom na to da generirani teren treba biti trodimenzionalni, potrebno je uz generiranje vrijednosti šuma, napraviti i mrežu vektora koji će predstavljati trodimenzionalni teren. Za te potrebe iskorištena je komponenta Mesh. Slika 30 prikazuje pravokutnik koji sadrži četiri točke. Svaka točka označena je koordinatama, krenuvši od gornjeg lijevog kuta u smjeru kazaljke na satu, točke redom su: (0,1), (1,1),(1,0),(0,0).



Slika 30 Pravokutnik podijeljen na dva trokuta

S obzirom na to da je mreža sačinjena od trokuta, svaki objekt se može razdijeliti na skup trokuta. Trokut definiraju tri točke pa bi tako donji trokut bio definiran točkama (0,1) – (1,0) – (0,0), a gornji trokut bi bio definiran točkama (0,1) – (1,1) – (1,0). Ove definicije trokuta nisu jedine koje definiraju isti trokut. Primjerice, gornji trokut također može biti definiran točkama (0,1) – (1,0) – (1,1). Predstavljanje objekta trokutima vrlo brzo postaje jako kompleksno, posebice u virtualnim okruženjima visoke rezolucije. U trodimenzionalnim okruženjima nije potrebno crtati sve trokute, obzirom na perspektivu neki trokuti nisu vidljivi. Taj problem moguće je riješiti primjenom postupka uklanjanja stražnje strane objekta (engl. *Backface culling*). Backface culling je postupa određivanja koja strana objekta je vidljiva ili nije kako bi se spriječilo pretjerano iscrtavanje trokuta koji zapravo niti ne trebaju biti iscrtani jer nisu vidljivi igraču/korisniku. U Unityju backface culling možemo kontrolirati sa smjerom definiranja trokuta pomoću vektora. U gornjem primjeru gdje je gornji trokut definiran točkama (0,1) – (1,1) – (1,0) Unity bi interpretirao kao trokut koji je vidljiv korisniku dok gornji trokut definiran točkama (0,1) – (1,0) – (1,1) Unity interpretira kao stražnju stranu objekta, dakle trokuti definiranu točkama u smjeru kazaljke na satu su usmjereni prema kameri, a trokuti definirani točkama u smjeru suprotnom od kazaljke na satu su sa stražnje strane objekta pa ih Unity i ne iscrtava [12]. Uz vektore i trokute vrlo bitna stavka, za izgled objekta koji se prikazuje pomoću mreže, jesu normale. Normala je vektor koji je okomit na površinu. U mreži svaki vektor treba imati svoju normalu kako bi se pravilno izračunalo osvjetljenje objekta. Uz normale koje omogućavaju pravilno osvjetljenje objekta, objekt nije potpun bez teksture. Informacije o teksturi pohranjene su kao UV koordinate. UV koordinate predstavljaju položaj u slici, primjerice 0 je lijevi kut slike, a 1 je gornji desni kut slike [16]. Generirana mreža, dakle, treba biti sastavljena od

trokuta i vektora, a vektori moraju reflektirati generirane vrijednosti Perlinovog šuma kako bismo dobili trodimenzionalni teren. Vrijednosti Perlinovog šuma zapravo predstavljaju visinsku mapu koju možemo iskoristiti kako bismo u generiranoj mreži dobili željene vektore i postigli trodimenzionalnost terena. Broj vektora koje generirani teren ima može se dobiti ako se pomnoži visina i širina terena, jednako kao i broj vrijednosti koje generira Perlinov šum. Prolazeći kroz dvodimenzionalno polje, vektorima se pridjeljuje vrijednost Perlinovog šuma, prema trenutnoj poziciji u dvodimenzionalnom polju, kao y koordinata što će predstavljati visinu tog vektora i rezultat će biti trodimenzionalno generirani teren.

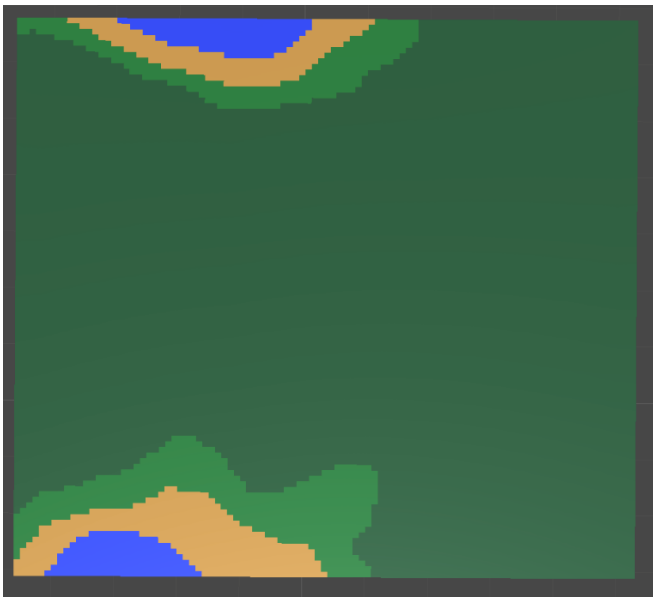
4.7. Generiranje terena

Generiranje terena odvija se na svaku promjenu parametara kako bi korisnik u realnom vremenu mogao vidjeti učinak pojedinih parametara na generirani teren. Prvi korak kod generiranja terena jest za određene dimenzije, visinu i širinu, generirati Perlinov šum koji ima $h \times w$ točaka gdje je h visina, a w širina određenog objekta/plohe. Dodatan sloj nasumičnosti čini random seed prema kojem će funkcija Perlinovog šuma, definirana u Unityju, uvijek dati drugačiji set vrijednosti. Random seed je vrijednost koja se koristi za inicijalizaciju pseudo nasumičnih generatora ili u ovom slučaju funkcije Perlinovog šuma.

```
public static float[,] generateNoise(
    int height, int width, int seed,
    int octaves, float persistance,
    float lacunarity, float scale,
    float noiseHeightMaxValue)
{
    float[,] noiseValues = new float[height, width];
    Random.InitState(seed);
    var randomOffsetX = Random.value;
    var randomOffsetY = Random.value;

    float maxNoise = float.MinValue;
    float minNoise = float.MaxValue;

    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {

            float noiseHeight = 0;
            float amplitude = 1;
            float frequency = 1;

            for (int k = 0; k < octaves; k++)
            {
                float sampleX = i / scale * frequency;
                float sampleY = j / scale * frequency;

                float perlinValue = Mathf.PerlinNoise(sampleX + randomOffsetX, sampleY + randomOffsetY) * 2 - 1;
                noiseHeight += perlinValue * amplitude;

                amplitude *= persistance;
                frequency *= lacunarity;
            }

            if (noiseHeight > maxNoise)
            {
                maxNoise = noiseHeight;
            }
            else if (noiseHeight < minNoise)
            {
                minNoise = noiseHeight;
            }

            noiseValues[i, j] = noiseHeight;
        }
    }


    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            float value = Mathf.InverseLerp(minNoise, maxNoise, noiseValues[i, j]);
            noiseValues[i, j] = value > noiseHeightMaxValue ? noiseHeightMaxValue : value;
        }
    }
    return noiseValues;
}
```

Isječak koda 10 Funkcija koja generira dvodimenzionalno polje vrijednosti Perlinovog šuma

Nakon generiranja Perlinovog šuma, primjenjuje se tekstura, odnosno za svaku točku u mreži definira se vrijednost teksture, u ovom slučaju to je isključivo boja teksture. Izlaz tog procesa je dvodimenzionalni izgled generiranog terena. Izgled dvodimenzionalnog terena prikazan je na slici (Slika 34).



Slika 31 Izgled dvodimenzionalnog terena prije generiranja trodimenzionalne mreže

Dvodimenzionalni teren je i dalje ravan kao što je i bio prije generiranja Perlinovog šuma, stoga je potrebno kreirati mrežu kako bi svaka točka poprimila i određenu visinu i time se dobiva konačno generirani teren.

```
private Texture2D setTextureColor(Texture2D texture)
{
    noiseValues = new float[height, width];
    if (noiseType.Equals(NoiseTypeEnum.PERLIN_NOISE))
    {
        noiseValues = PerlinNoiseUtil.generateNoise(height, width, seed, octaves, persistance, lacunarity, scale, noiseHeightMaxValue);
        Color[,] colorMap = TextureColors.generateColorMap(height, width, textureColors, noiseValues);
        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                texture.SetPixel(i, j, colorMap[i, j]);
            }
        }
    }
    else if (noiseType.Equals(NoiseTypeEnum.WHITE_NOISE))
    {
        noiseValues = WhiteNoiseUtil.generateNoise(height, width);
        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                texture.SetPixel(i, j, Color.Lerp(Color.black, Color.white, noiseValues[i, j]));
            }
        }
    }
    return texture;
}
```

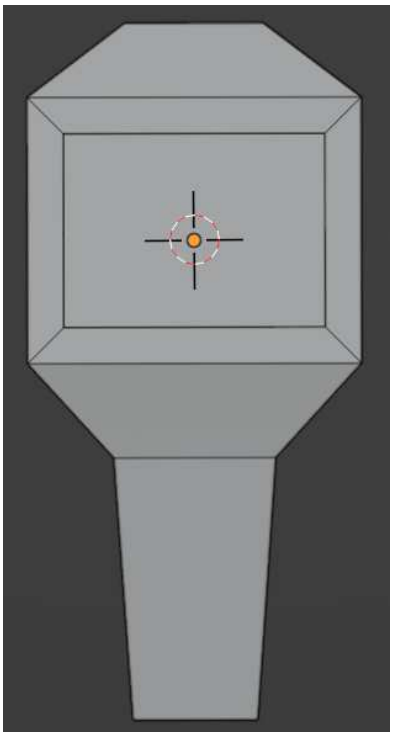
Isječak koda 11 Funkcija koja postavlja boju teksture za generirani teren

```
public void generateElevatedTexture()
{

    SensorHandler[] handlers = gameObject.GetComponentsInChildren<SensorHandler>();
    foreach (SensorHandler handler in handlers)
    {
        handler.destroy();
    }
    texture = setTextureColor(new Texture2D(height, width));

    texture.filterMode = FilterMode.Point;
    texture.Apply();

    MeshData meshData = MeshGenerator.GenerateTerrainMesh(noiseValues, heightMultiplier, meshHeightCurve);

    Mesh mesh = meshData.CreateMesh();
    meshFilter.sharedMesh = mesh;
    meshRenderer.sharedMaterial.mainTexture = texture;
    meshCollider.sharedMesh = mesh;
    placeSensorsButton.SetActive(true);
}
```

Isječak koda 12 Funkcija koja generira konačan trodimenzionalni teren

4.8. Postavljanje senzora

Nakon generiranog terena, korisniku se pojavljuje opcija za postavljanje senzora. Senzori su modelirani u Blenderu i prikazani su na slici (Slika 32). Svrha senzora je da mjere temperaturu i vlažnost tla u definiranom radijusu. Vrijednosti koje pojedini senzora očitava su nasumične i mijenjaju se u intervalima očitavanja senzora, koji je za potrebe implementacijskog rješenja postavljen na 10 sekundi.

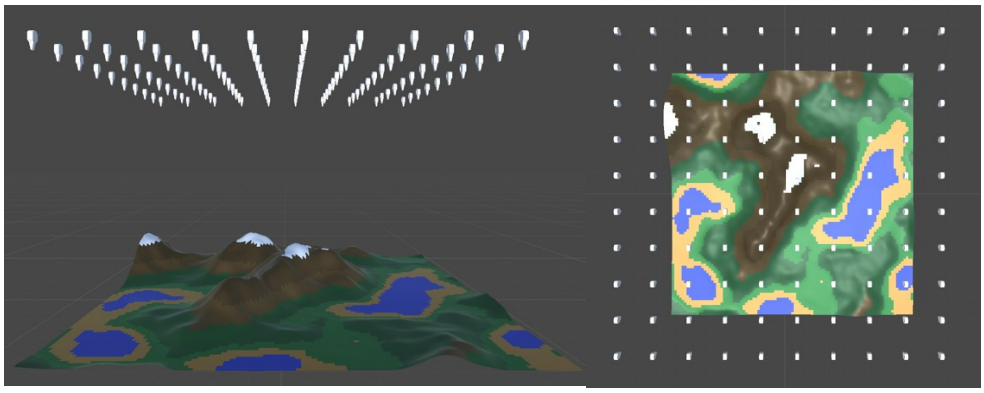


Slika 32 Model senzora prikazan u Blenderu

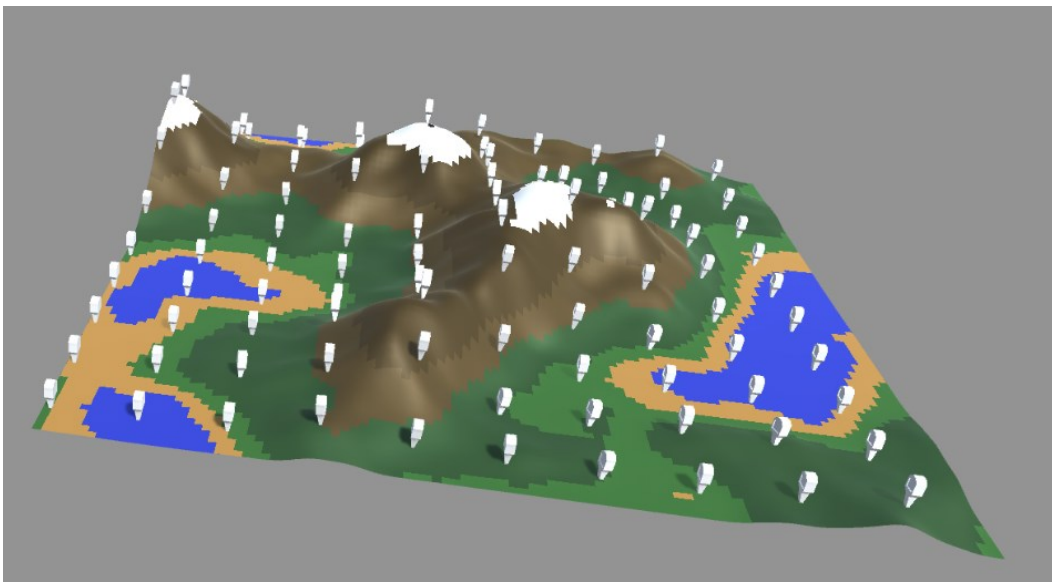
Senzori se postavljaju u pravokutnu mrežu senzora, uzimajući u obzir radijus očitanja koje senzor pokriva. Uzimajući u obzir da je radijus senzora 10, a duljina i širina terena 100, moguće je izračunati koliko će biti potrebno senzora kako bi se teren prekrio sa najmanjim brojem senzora, a da svejedno postoji informacija za svaki dio terena o njegovoj temperaturi i vlažnosti. Svaki senzor će pokrivati jedan kvadrat duljine i širine jednake radijusu senzora, a pozicija senzora bit će u sredini tog kvadrata. Broj senzora, za teren širine i duljine 100, koji će se postaviti na teren je 100. U nastavku su dane formule za izračun broja senzora:

$$brojSenzoraNaXKoordinati = širinaTerena \ / \ radijusSenzora$$

Svakom senzoru se prilikom stvaranja dodjeljuje komponenta *RigidBody* kako bi se senzor mogao postaviti na teren prilikom kolizije s terenom. Ako već postoje neki senzori, prvo se ti isti uništavaju i generiraju se novi.



Slika 33 Mreža senzora prije kolizije s terenom



Slika 34 Mreža senzora nakon kolizije s terenom

```
public void createSensors()
{
    Random.InitState((int)System.DateTime.Now.Ticks);
    SensorHandler[] handlers = terrain.GetComponentsInChildren<SensorHandler>();
    foreach(SensorHandler handler in handlers)
    {
        handler.destroy();
    }
    int numberOfSensorsOnXAxis = noiseTexture.width / sensorRadius;
    int numberOfSensorsOnYAxis = noiseTexture.height / sensorRadius;
    List<Vector3> sensorPositions = new List<Vector3>();
    for(int i = Mathf.RoundToInt(numberOfSensorsOnYAxis/2f) - 1; i >= Mathf.RoundToInt(numberOfSensorsOnYAxis/-2f); i--)
    {
        for(int j = Mathf.RoundToInt(numberOfSensorsOnXAxis/-2f); j <= Mathf.RoundToInt(numberOfSensorsOnXAxis/2f) - 1; j++)
        {
            Vector3 position = new Vector3
                (
                    i * sensorRadius + sensorRadius/2,
                    90,
                    j * sensorRadius + sensorRadius/2
                );
            sensorPositions.Add(position);
            instantiateSensor(position);
        }
    }
    gameObject.SetActive(false);
}

private void instantiateSensor(Vector3 position)
{
    GameObject newSensor = Instantiate(sensor);
    SensorHandler sensorHandler = newSensor.GetComponent<SensorHandler>();
    sensorHandler.panel = panel;
    sensorHandler.temp = tempText;
    sensorHandler.moisture = moistureText;
    newSensor.GetComponent<Rigidbody>().isKinematic = false;
    newSensor.transform.SetParent(terrain.transform);
    newSensor.transform.position = position;

}
```

Isječak koda 13 Funkcije koje se brinu o uništavanju i postavljanju senzora

Svaki senzor brine o svojoj vrijednosti temperature i vlažnosti tla. Korisnik može očitati vrijednost senzora na način da prijeđe mišem preko senzora pa će mu se prikazati panel s informacijom o tom senzoru.

Slika 35 Izgled korisničkog sučelja koje prikazuje informacije o senzoru

```
private void Start()
{
    tempValue = Random.Range(20, 40);
    moistureValue = Random.Range(20, 25);
}

private void Update()
{
    time += Time.deltaTime;
    if(time % 60 > 10f)
    {
        tempValue = Random.Range(20, 40);
        moistureValue = Random.Range(20, 25);
        time = 0;
    }


}

private void OnCollisionEnter(Collision collision)
{
    gameObject.GetComponent<Rigidbody>().isKinematic = true;
}

private void OnMouseOver()
{
    panel.SetActive(true);
    temp.text = "" + tempValue;
    moisture.text = "" + moistureValue + "%";

}

private void OnMouseExit()
{
    panel.SetActive(false);
}

public void destroy()
{
    Destroy(gameObject);
}
```

Isječak koda 14 Funkcije senzora

4.9. Moguća poboljšanja

Programsko rješenje vrlo je jednostavno, vrlo dobro ilustrira postupak proceduralnog generiranja, no svejedno ima mjesta za poboljšanja. Generiranje Perlinovog šuma za N broj točaka gdje je N >>1000 zahtijeva jako puno računalnih resursa i iz tog razloga nemoguće je proceduralno generirati veliko okruženje odjednom. Postoji nekoliko rješenja problema generiranja velikih terena odjednom. Prvo rješenje bilo bi smanjivanje detalja, odnosno smanjivanje rezolucije mreže trokuta koja se generira. Programsko rješenje koristi svaki vektor iz dvodimenzionalnog polja veličine $visina * širina$, i time se dobiva jako puno trokuta koji se generiraju prilikom generiranja mreže trokuta. Za teren dimenzija $100 * 100$ broj trokuta iznosi 19602, taj broj se može dobiti prema sljedećoj formuli:

$$Broj\ trokuta = (visina - 1) * (širina - 1) * 2$$

U slučaju smanjivanja rezolucije, primjerice da se umjesto svakog vektora koristi svaki drugi vektor, efektivno dobije se 4 puta manja rezolucija i time se smanjuje broj trokuta za 4 puta. Smanjivanjem rezolucije i teren gubi svoj oblik pa je tu uvijek dobro odmjeriti pluseve i minuse jednog i drugog pristupa. Drugo rješenje generiranja velikih terena jest kreiranje manjih dijelova terena ovisno o kretanju vidnog polja virtualne igre. Igrač ima ograničeno vidno polje tako da nije potrebno kreirati dio terena sve dok dio teren nije u vidnom polju koje se prikazuje na ekranu. Takvo rješenje zapravo povlači još dva problema, jedan problem je višedretvenost i računalni resursi, postoji rizik od presporog generiranja, a drugi problem je spajanje više tekstura, događa se problem nekonzistentnosti između dva dijela terena, zbog normaliziranja vrijednosti svakog komadića terena. Svaki dio terena prati svoju najvišu točku i prema njoj normalizira vrijednosti, no kada bi se pratila najviša točka globalno i kada bi se normalizirale vrijednosti u odnosu na globalnu točku izbjegao bi se problem spajanja tekstura. Isto tako, korištena tekstura vrlo je primitivna. Cilj programskog rješenja nije generirati najrealističniji mogući teren već ilustrirati kreiranje teksture. Ovisno o konačnoj igri, ovakav pristup možda nije zadovoljavajuće pa bi se umjesto trenutne teksture u programskom rješenju trebalo osmisliti novi način bojanja teksture, primjenom tehnike sjenčanja i shadera u Unityju. Posljednje poboljšanje bi bilo u vidu senzora, trenutno se senzori postavljaju u pravokutnu mrežu koja ne prekriva najveću moguću površinu, već neki dijelovi terena ostaju nepokriveni senzorima pa tako ne postoji mogućnost preciznog mjerenja svih parametara. U poglavlju 2.4 dano je rješenje ovog problema.

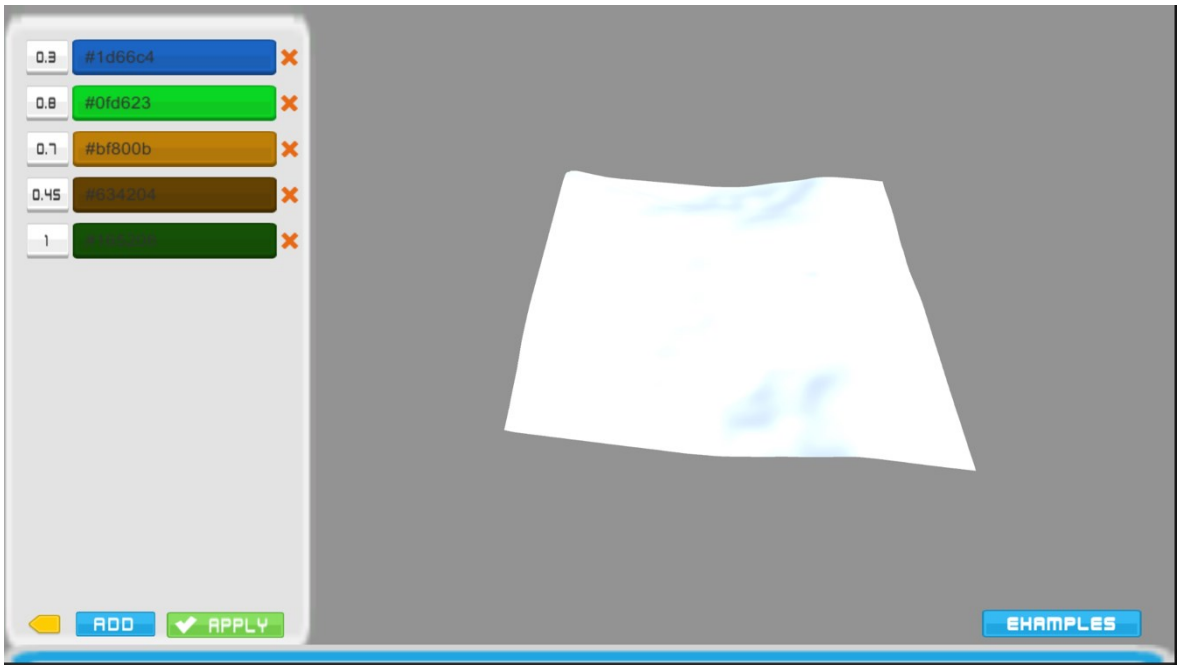
5. Rezultati

5.1. Postavljanje parametara generiranog terena

Kako se programsko rješenje ne bi koristilo na pogrešan način, u nastavku su dane upute i smjernice koje korisniku pomažu da se koristi programskim rješenjem intuitivno. Kada se programsko rješenje prvi puta otvori, pred korisnikom se nalazi ekran (Slika 20) i korisnik može promijeniti bilo koju od opcija. Nakon promjene bilo koje od opcija ponovno se generira novi teren kako bi korisnik bio svjestan kako promjena određenih parametara mijenja generirani teren. Prilično je intuitivno mijenjanje tih parametara no ekran 2 (Slika 21) zahtijeva da korisnik razumije način na koji se primjenjuju boje. Pred korisnikom se nalaze dva unosa, svaki od unosa sadrži dva tekstualna unosa, prvi tekstualni unos zahtijeva unos decimalnog broja, a u drugi tekstualni unos korisnik unosi heksadecimalni kod željene boje. Svaki od tih unosa predstavlja gornju granicu za neku boju. Perlinov šum daje vrijednosti u intervalu [0,1]. Primjerice, Perlinov šum generira 10 vrijednosti:

$$[0.15\,,0.72\,,0.92\,,0.14\,,0.24\,,0.62\,,0.32,0.54\,,0.8\,,0.4]$$

A korisnik je definirao boje na način prikazan na slici (Slika 36).



Slika 36 Primjer unosa boja

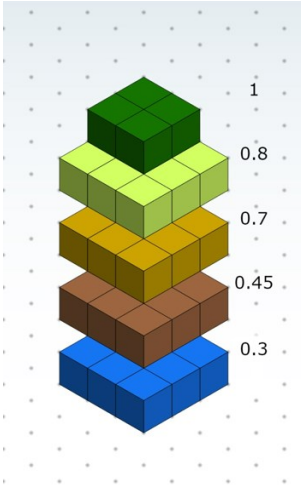
Prvi unos boje kao granicu boje ima 0.3, to znači da sve vrijednosti Perlinovog šuma koje su manje od 0.3 imaju na svojoj teksturi primijenjenu navedenu plavu boju. Prema unesenim bojama granice su sljedeće:

- [0, 0.3> - tekstura će imati plavu boju
- [0.3, 0.45> - tekstura će imati smeđu boju
- [0.45, 0.7> - tekstura će imati svjetlo smeđu boju
- [0.7, 0.8> - tekstura će imati svjetlo zelenu boju
- [0.8, 1] – tekstura će imati tamno zelenu boju

Prema definiranim granicama generirane vrijednosti pripadaju u sljedeće granice:

- [0, 0.3> - [0.15, 0.14, 0.24]
- [0.3, 0.45> - [0.32, 0.4]
- [0.45, 0.7> - [0.54, 0.62]
- [0.7, 0.8> - [0.72]
- [0.8, 1] – [0.8, 0.92]

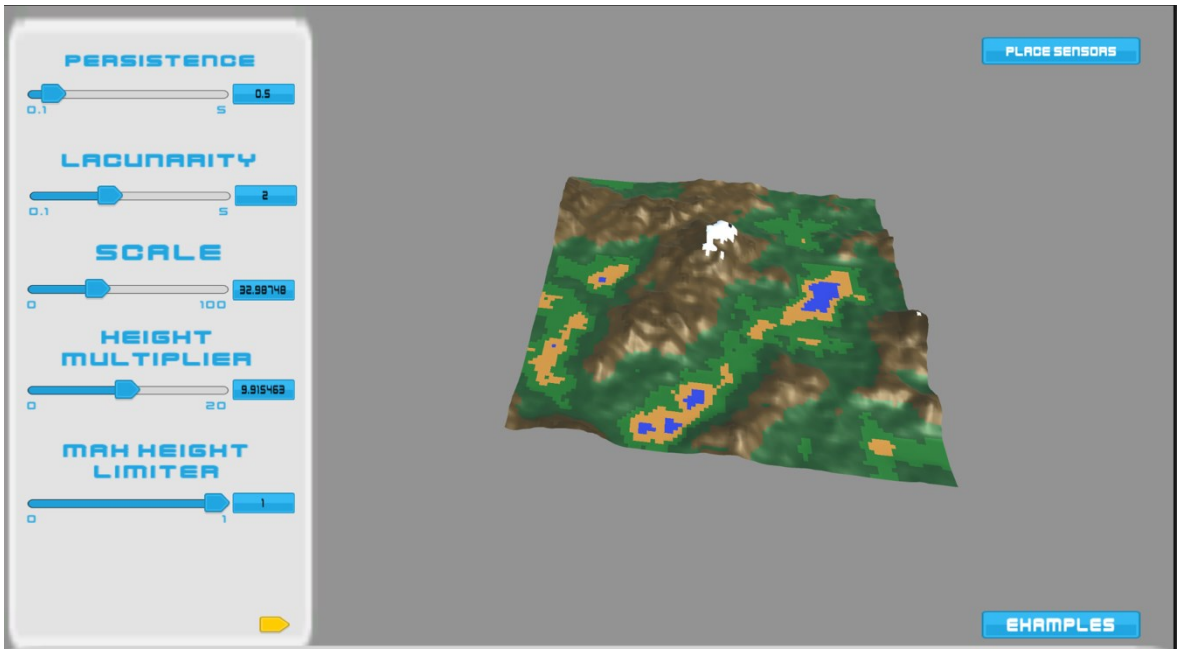
Ako u polje za unos broja ne postoji vrijednost to polje s bojom će se zanemariti prilikom generiranja novog terena. U korisničkom sučelje boje ne trebaju biti poredane ni u kojem poretku, već mogu biti nasumično unesene, a programsko rješenje će se pobrinuti o sortiranju. U nastavku je prikazan još jedna reprezentacija boja u vidu granicu i blokova boja.



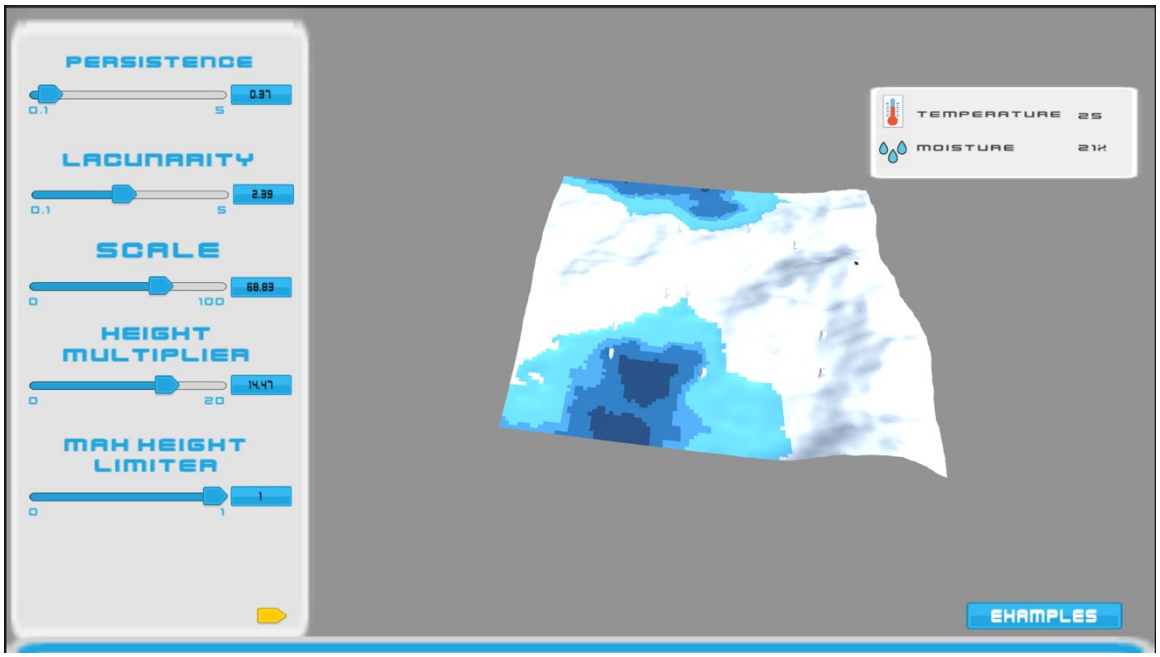
Slika 37 Prikaz granica boja

5.2. Primjeri generiranog terena

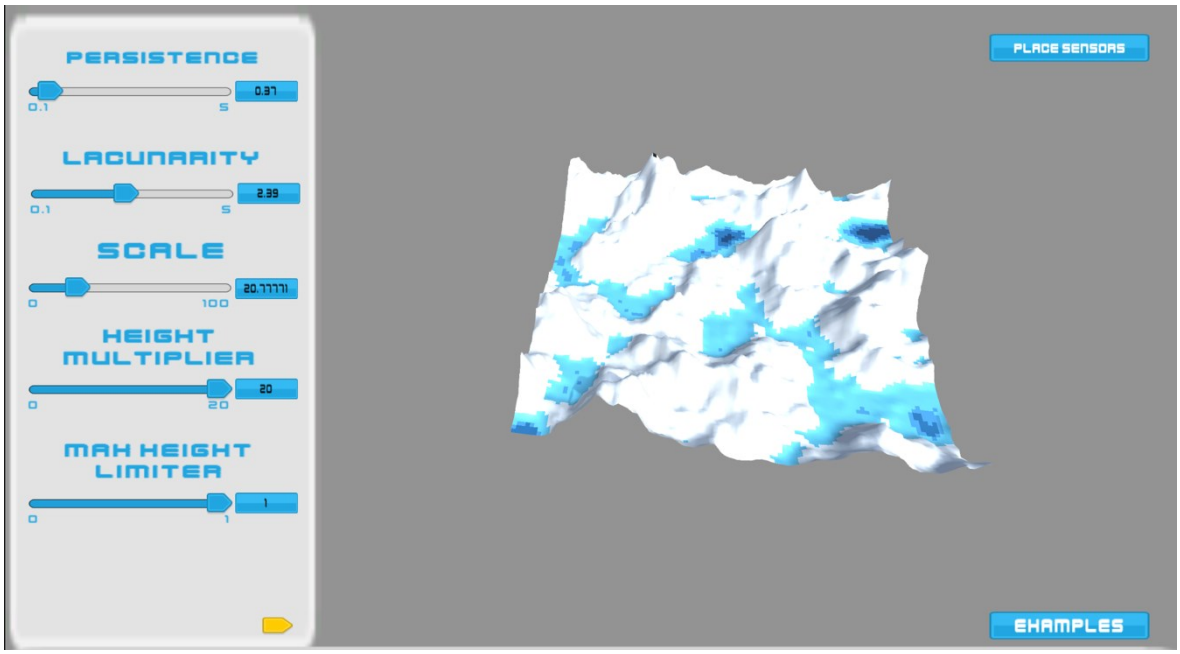
U nastavku su dane slike ekrana koje služe kao rezultati koji su dobiveni ovim programskim rješenjem. Dani primjeri su različiti tereni i parametri se mogu koristiti kako bi se dobili isti tereni.



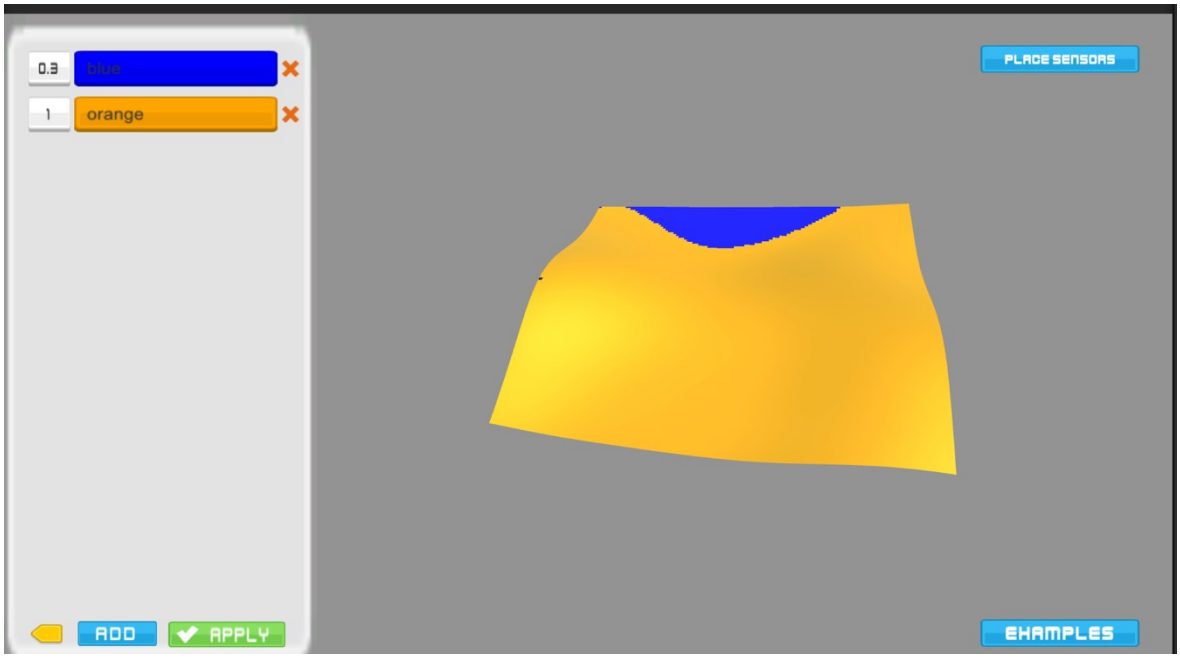
Slika 38 Primjer mogućeg rezultata: šumski teren s planinama



Slika 39 Primjer mogućeg rezultata, primjer ledene površine



Slika 40 Primjer mogućeg rezultata: ledenjak s izraženim planinama



Slika 41 Primjer mogućeg rezultata: pustinjski teren

5.3. Značajke programskog rješenja

Na performanse programskog rješenja uvelike utječe veličina terena koji se generira jer se za svaku točku duljine i širine terena računa odgovarajuća vrijednost Perlinovog šuma za tu

točku. U nastavku je dana tablica koja prikazuje vrijeme generiranja terena za određene veličine terena.

Veličina terena (širina, duljina)	**Vrijeme generiranja terena (sekunde)**
100, 100	0.012
110, 110	0.014
190, 190	0.046
250, 250	0.078
500, 500	0.22
700, 700	0.40
900, 900	0.65

Tablica 4 Vrijeme generiranja terena

Na tablici (Tablica 4) vidljivo je da vrijeme generiranja raste linearno kako se mijenja veličina terena koji se želi generirati. Prema tablici također je moguće i odrediti linearnu funkciju kojom se zatim može predvidjeti vrijeme generiranja terena. Dana je funkcija u ovisnosti o x, gdje x predstavlja vrijednost širine/duljine generiranog terena.

$$f(x) = 0.00075908x - 0.0953565$$

Zaključak

Proceduralno generiranje virtualnog okruženja zahtijeva duboko razumijevanje matematičkih koncepata funkcija šuma. Funkcije šuma deterministički su generatori pseudo slučajnih brojeva koji se zatim mogu interpretirati na mnogo različitih načina, jedan od tih načina jest za generiranja terena u virtualnom okruženju. Mnoge popularne igre, poput *Minecrafta i No Man's Sky*, koriste proceduralno generiranje kako bi igraču dale mogućnost višestrukog igranja iste igre ispočetka bez ponavljanja okoline. Proceduralno generiranje zahtijeva određivanje pravila kako bi se postigla kontrola nad onim što se generira. Ta pravila mogu biti implementirana u samo proceduralno generiranje ili pak mogu bili dio odbacivanja nekih vrijednosti nakon proceduralnog generiranja. U sklopu ovog rada izrađeno je programsko rješenje generiranja terena koristeći vrijednosti dobivene Perlinovim šumom za potrebe analize geometrije terena u vidu postavljanja senzora za preciznu poljoprivredu uzimajući u obzir položaj i domet svakog senzora. Za potrebe postavljanja mreže senzora provedena je analiza Voronoijevih dijagrama te je dan prijedlog za idealno rješenje, koristeći Lloydov algoritam relaksacije koji ujednačava Voronoijeve ćelije u Voronoijevom dijagramu i maksimizira pokrivenost terena što sličnijim Voronoijevim ćelijama. Konačni rezultat prezentiran je u vidu simulacijske igre izrađene u Unity programskom okviru, gdje korisnik može djelovati na generirani teren mijenjajući parametre funkcije Perlinovog šuma te tako svaki puta dobiti drugačije generirani teren. Na generirani teren se mogu postavljati senzori koji u realnom vremenu simuliraju promjenu dvaju parametara: temperature i vlažnosti tla.

Literatura

1. Steven Fortune. *A sweepline algorithm for Voronoi diagrams*. Proceedings of the second annual symposium on Computational geometry SCG ’86, pp. 313–322, ACM, 1986.

   Poveznica: https://dl.acm.org/doi/pdf/10.1145/10515.10549; pristupljeno 11. kolovoza 2022.

2. Randy Alexander, *Razlika između objektno orijentiranog programiranja i proceduralnog programiranja*. [Online] strepsonsays.com, travanj 2021.

   Poveznica: https://hr.strephonsays.com/object-oriented-programming-and-vs-procedural-programming-13453; pristupljeno 10. srpnja 2022.

3. Fortune's algorithm. [Online] desmos.com, 2022.

   Poveznica: https://www.desmos.com/calculator/0b96jkosm2; pristupljeno 21. kolovoza 2022.

4. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. *Procedural content generation for games: A survey.* ACM Trans. Multimedia Comput. Commun. Appl. 9, 1, Article 1 (February 2013), 22 pages. https://doi.org/10.1145/2422956.2422957

5. Heuristics. [Online] techopedia.com, 2022.

   Poveznica: https://www.techopedia.com/definition/5436/heuristic; pristupljeno 2. kolovoza 2022.

6. Caleb Compton, *How Mincraft generates massive virtual worlds from scratch.* Rempton games, 2021.

   Poveznica: https://remptongames.com/2021/02/28/how-minecraft-generates-massive-virtual-worlds-from-scratch/; pristupljeno 11. srpnja 2022.

7. Ken Perlin. *Improving noise*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques (SIGGRAPH '02), July 2002, pp. 681–682.

   Poveznica: https://doi.org/10.1145/566570.566636 ; ili na web stranici autora: https://mrl.cs.nyu.edu/~perlin/paper445.pdf; pristupljeno 11. srpnja 2022.

8. Libnoise. A portable, open-source, coherent noise-generating library for C++. [Online] sourceforge.net, 2013.

   Poveznica: http://libnoise.sourceforge.net/glossary; pristupljeno 9. srpnja 2022.

9. Lloyd, Stuart P. (1982), *Least squares quantization in PCM*, IEEE Transactions on Information Theory, 28 (2): 129–137, doi:10.1109/TIT.1982.1056489. (Pojednostavljeni opis na: https://en.wikipedia.org/wiki/Lloyd%27s_algorithm; pristupljeno 21. kolovoza 2022.)

10. Jason Davies, Lloyd's relaxation. [Online] jasondavies.com.

    Poveznica: https://www.jasondavies.com/lloyd/; pristupljeno 21. kolovoza 2022.

11. Matt Tillman. Manhattan voronoi approximation. [Online] www.pixelninja.design, 2020.

    Poveznica: http://www.pixelninja.design/manhattan-voronoi-approximation; pristupljeno 15. kolovoza 2022.

12. Mesh Generation in Unity.

    Poveznica: https://www.youtube.com/watch?v=eJEpeUH1EMg&ab_channel=Brackeys; pristupljeno 31. kolovoza 2022.

13. Noise functions.

    Poveznica: https://www.ronja-tutorials.com/noise.html; pristupljeno

    19. srpnja 2022.

14. Top 7 gaming engines you should consider.

    Poveznica: https://www.incredibuild.com/blog/top-7-gaming-engines-you-should-consider, pristupljeno 4. rujna 2022.

15. Understanding Perlin noise.

    Poveznica: https://adrianb.io/2014/08/09/perlinnoise.html; pristupljeno

    10. kolovoza 2022.

16. Unity documentation.

    Poveznica: https://docs.unity.cn/560/Documentation/Manual/AnatomyofaMesh.html; pristupljeno 31. kolovoza 2022.

17. Unity UI Documentation.

    Poveznica: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html, pristupljeno 2. rujna 2022.

18. Unity visual scripting.

    Poveznica: https://unity.com/features/unity-visual-scripting, pristupljeno

    4. rujna 2022.

19. Value noise and procedural patterns.

    Poveznica: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/introduction; pristupljeno 18. srpnja 2022.

20. Voronoi diagram: distance functions.

Poveznica: https://www.codingame.com/playgrounds/243/voronoi-diagrams/voronoi-diagrams-distance-functions; pristupljeno 12. kolovoza 2022.

21. Voronoi diagrams.

    Poveznica: https://www.geogebra.org/m/ZVQvCWEV#material/Bmtt7nHj; pristupljeno 13.kolovoza 2022.

22. What are voronoi diagrams?.

    Poveznica: https://www.codingame.com/playgrounds/243/voronoi-diagrams/what-are-voronoi-diagrams; pristupljeno 21. kolovoz 2022.

23. What is white noise and why is it so powerful.

    Poveznica: https://blog.noisli.com/what-is-white-noise-and-why-is-it-so-powerful/; pristupljeno 12. srpnja 2022.

24. White noise.

    Poveznica: https://ccrma.stanford.edu/~jos/sasp/White_Noise_I.html; pristupljeno 12. srpnja 2022.

Sažetak

Naslov: Proceduralno generiranje terena i objekata u virtualnom okruženju

Sažetak:

Proceduralno generiranje temeljeno na funkcijama šuma može poslužiti za generiranje terena i virtualnih okruženja u računalnim igrama. U okviru ovog rada opisana su svojstva funkcija šuma, posebice Perlinove funkcije šuma koja je korištena za generiranje vrijednosti za generiranje 3D terena. Dizajnirana je simulacijska igra u kojoj se putem korisničkog sučelja može upravljati parametrima funkcije šuma te je implementirana pomoću razvojnog okvira Unity. Također je i razmatrana moguća primjena Voronoijevih dijagrama za postavljanje objekata na proceduralno generiranom terenu, na zamišljenom primjeru postavljanja mreže senzora za preciznu poljoprivredu. Na kraju rada analizirane su prednosti i nedostaci razvijenog rješenja i dani su prijedlozi kako bi se problemi mogli riješiti u mogućim budućim inačicama.

Ključne riječi:

računalna grafika, proceduralno generiranje, Perlinov šum, Voronoijev dijagram, virtualno okruženje, pogonski sustava za igre Unity

Summary

Title: Procedural generation of terrain and objects in virtual environment

Summary:

Procedural generation based on noise functions can be applied to terrain generation and virtual environments in computer games. This work contains a description of noise functions, in particular the Perlin noise that was used to generate 3D terrain values. A simulation game has been designed and implemented by using Unity game engine, in which a user can manipulate the parameters of noise functions through a user interface. An application of Voronoi diagrams has been considered for object placement on a generated terrain, using as an example an imaginary problem of sensor placement for precision agriculture. Finally, advantages and disadvantages of the developed solution have been analyzed, and suggestions have been proposed for possible future work.

Key words:

computer graphics, procedural generation, Perlin noise, Voronoi diagram, virtual environment, Unity game engine