SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 221

# GENERIRANJE I PRIKAZ TERENA

Dario Filipović

Zagreb, veljača 2008.

Zahvaljujem se mentorici prof. dr. sc. Željki Mihajlović na savjetima i pruženoj pomoći prilikom pisanja ovog rada. Posebno se zahvaljujem roditeljima Gordani i Srećku na pruženoj podršci tokom dosadašnjeg studija.

# Sadržaj:

# 1. Uvod

Prikaz trodimenzionalnih terena već duže vrijeme je jedno od zanimljivijih područja računalne grafike, odnosno animacije. Što se ustvari želi postići je preslikavanje krajobraza iz našeg, stvarnog, svijeta na računalni ekran. To nipošto nije jednostavan proces ako uzmemo u obzir kompleksnost svega što nas okružuje.

Ne postoje dva identična genetska bića na ovoj planeti. Analogno tome ne postoje niti dva identična mjesta na Zemlji. Sve sadrži određenu količinu nasumičnosti, kako mi to volimo zvati. Točnije bi bilo reći da sve podliježe nekom pravilu, vidjeli mi to pravilo ili ne. Čovjekova težnja za shvaćanjem stvari oduvijek je bila motivacija za traženje tih pravila i njihovim korištenjem u svrhe predviđanja i opisivanja stvari i događaja. Ona su obično ili prilično intuitivna i jednostavna ili izrazito kompleksna do te granice da se ne mogu pratiti sve varijable koje se mijenjaju u realnosti. To nas prisiljava na pojednostavljivanje svari.

S terenima je ista priča. Zar bi bilo potrebno svaki atom zemlje upisati u memoriju i kasnije crtati? Vjerojatno bi bilo korisno za neke svrhe, no treba pogledati stvarnost. Prva i najvažnija stvar; za što je potreban prikaz terena u prostoru?

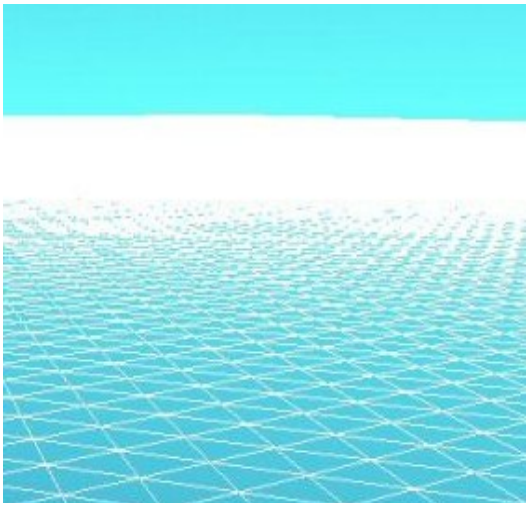
Trodimenzionalni prikaz terena ćemo često naći u ovim aplikacijama:

- Simulacijama (često vojne)
- Sustavima vizualizacija (npr. za prikaz topologije)
- Igre
- Virtualni turizam
- ...

Još jedno važno pitanje je: Koja razina detalja nam treba? Tu se treba osvrnuti na to za što nam je potreban teren. Ako je potrebno za scenu u filmu, u redu je odvojiti sat vremena za metodu praćenja zrake za generiranje jednog okvira. Češće je naglasak na animaciji terena u stvarnom vremenu (*engl. realtime*). U ovom radu će fokus biti na problematici generiranja terena i njihovo prikazivanje u stvarnom vremenu.

# 2. Problematika generiranja terena

Za početak bi bilo dobro spomenuti da za prikaz u stvarnom vremenu moramo izbaciti veliku većinu detalja iz stvarnog svijeta. Tako se stvarni krajobraz na računalu često zapisuje kao mreža pravilno raspoređenih točaka (zašto je to tako biti će riječi prilikom opisa metoda prikaza terena). Naziv za takvu mrežu je visinska mapa (*engl. heightmap*). Može se zamisliti da krećemo od mreže trokuta (*engl. triangle mesh*) koja predstavlja jednu plohu. Ta ploha je dakle predstavljena dvodimenzionalnim poljem točaka koje sve imaju jednu koordinatu 0. To bi izgledalo kao na s*lici 1*. Mi želimo postići efekt sličan onom na *slici 2*.



*Slika 1: Mreža trokuta na jednoj plohi*

*Slika 2: Deformirana mreža trokuta*

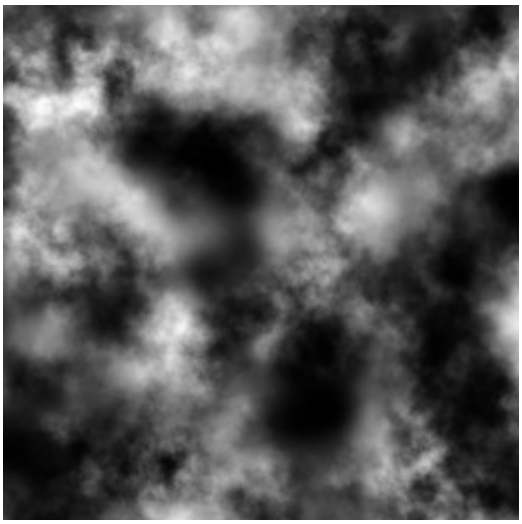
*Slika 2* prikazuje mrežu trokuta koja je deformirana tako da nalikuje 3D terenu koji je prikazan kao mreža linija. Na slici se vidi valovit oblik koji predstavlja brda i doline stvarnog terena i to prilično dobro. Postoji više metoda kojima je moguće stvoriti/učitati ovako deformiranu mrežu trokuta.

Osvrnuti ćemo se na neke od njih:

- Čitanje visinske mape iz datoteke
- Generiranje visinske mape dijeljenjem (metoda dijamant-kvadrat)
- Generiranje visinske mape pukotinama
- Generiranje visinske mape kružnicama

## *2.1 Čitanje visinske mape iz datoteke*

Visinska mapa se može predstaviti kao slika. Oboje su u osnovi dvodimenzionalno polje vrijednosti. Jedina je razlika što su slike kao datoteke zapisane kao cjelobrojne vrijednosti, a za prikaz u prostoru se najčešće koriste decimalni brojevi (*float*). Znači najveći problem je učitavanje podataka iz datoteke i skaliranje brojeva tako da nam odgovaraju. Na internetu se mogu pronaći brojni generatori terena koji generiraju npr. BMP datoteku čije nijanse sive boje predstavljaju visinu te točke u prostoru. Primjer takvog generatora je http://nemesis.thewavelength.net/index.php?c=2.
Visinska mapa se može vidjeti na *slici 3*.



*Slika 3: Primjer visinske mape*

Visinske mape su najčešće oblika takvog da crna boja predstavlja najniže razine terena, a bijela najviše razine. Razina terena je najčešće predstavljena Y-osi u prostoru, dok su X i Z osi koordinate te točke na visinskoj mapi.
Ovo je jedina spomenuta metoda koja ne stvara visinsku mapu već ju samo učitava.

## 2.2 *Generiranje visinske mape dijeljenjem*

Ovo je prvi od algoritama koji počinju s praznom visinskom mapom (sve vrijednosti visinske mape na 0), te određenim postupcima generiranja terena određuju vrijednosti za svaku točku. Za ovaj algoritam je poželjno da su visina i širina visinske mape jednake i djeljive sa 2. To znači da bi visinska mapa trebala biti četverokut. Osnovni razlog tome je što je algoritam dosta brži ako se radi s ovakvim dimenzijama.

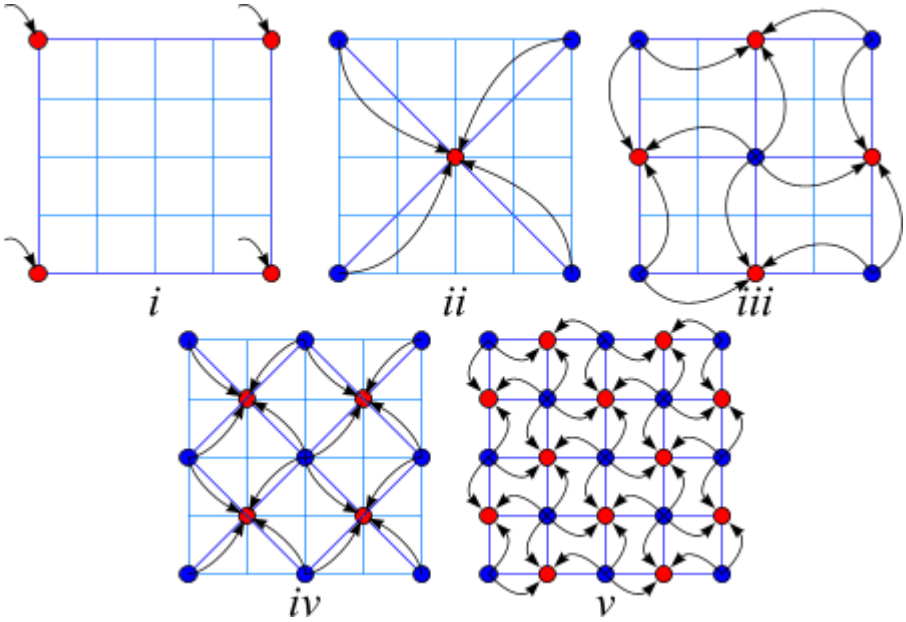
Prvi korak algoritma dodjeljuje krajnjim rubnim točkama nasumične vrijednosti visine. Slijedeći korak uzima točke na polovištu svake stranice i na sredini trenutnog četverokuta od srednje vrijednosti dobivene interpolacijom dodaje ili oduzima određenu nasumično odabranu vrijednost. Ova dva koraka se rekurzivno ponavljaju sa brojevima sve manjih nasumičnih vrijednosti koje se zbrajaju ili oduzimaju, tako da se utjecaj na deformaciju terena smanjuje svakom iteracijom. Algoritam završava nakon što je došao do zadane razine detalja ili kada nema više mogućih točaka za odabrati za slijedeći korak. Nekoliko koraka ovog algoritma na dvodimenzionalnom prikazu se može vidjeti na *slici 4.*

Prednost ovog algoritma je velika brzina, jer se svaka točka na visinskoj mapi mijenja samo jednom. Osim toga, rezultati dobiveni ovim algoritmom su prilično dobri.

Postoji više modifikacija ovog algoritma, tako je jedna od tih modifikacija Metoda *dijamant-kvadrata* (engl. *Diamond-square method* )[1]

Algoritam se u toj metodi svodi na to da se umjesto četiri rubne točke i središnje točke u drugom koraku prvo izračunaju središta kvadrata na toj razini detalja, te se nakon toga račun ponavlja koristeći nova središta i krajnje točke svake od stranica (koji sada čine dijamant) da bi se izračunala polovišta stranica. Koraci se mogu vidjeti na *slici 5.*[2]


*Slika 4: Dvodimenzionalni prikaz algoritma dijeljenja*



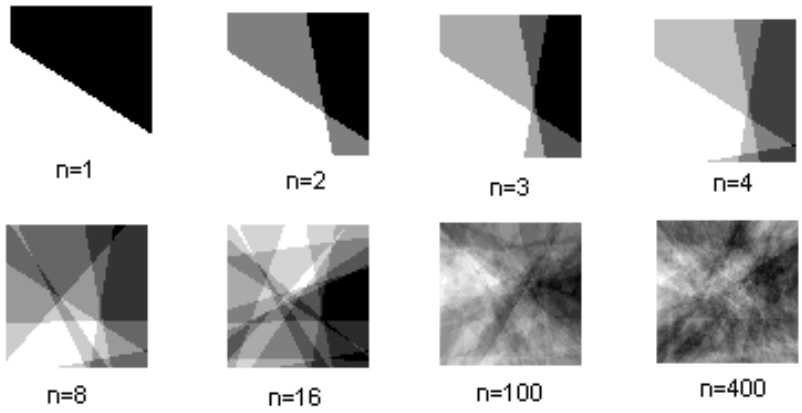

*Slika 5: Metoda dijamant-kvadrata*

Metoda *dijamant-kvadrat* je vrlo popularna zbog brzine i boljih rezultata od izvornika. Općenito, cijela ova obitelj algoritama se još često naziva i fraktalna generacija terena. Taj im je naziv dodijeljen zbog toga što se u algoritmu rekurzivno ponavljaju postupci koji stvaraju izmjene na terenu slične kao i u koraku prije, odnosno pojavljuje se samosličnost između različitih razina detalja.

## *2.3 Generiranje visinske mape pukotinama*

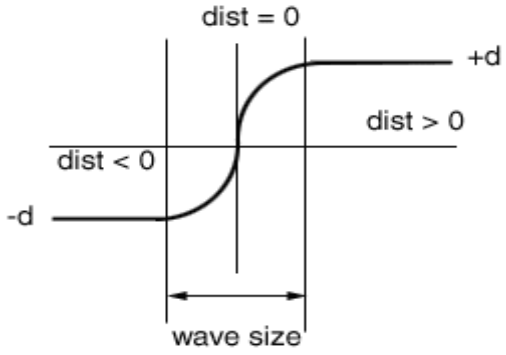
Generiranje visinske mape pukotinama kao osnovni element koristi pravac. Stvaranjem pravca na temelju početne točke i koeficijenta smjera određuje se linija presjeka. S jedne strane te linije visina svih točaka podiže se za određenu vrijednost, a s druge se strane ta ista vrijednost oduzima od trenutne vrijednosti točaka ako je to moguće.

Izgled visinske mape nakon n iteracija ovog algoritma se može vidjeti na *slici 6.*




*Slika 6: Iteracije algoritma za generaciju visinskih mapa pukotinama*

S prijašnjih slika se vidi da je potrebno mnogo iteracija da bi se dobila dobra visinska mapa. Ako se uzme u obzir da se za svaku tu iteraciju prolazi kroz sve točke na visinskoj mapi, može se vidjeti da ovaj algoritam nije baš efikasan u iskorištavanju procesorskog vremena. Na mjestima gdje se dvije susjedne površine smiču dolazi do velikog naglog skoka u visinama, što je problem. Ova pojava je još poznata i kao step funkcija. To se može vidjeti u dvodimenzionanom prikazu na *slici 7.*




*Slika 8: Prirodniji pad s uzvišene plohe*

Ono što se želi postići je prirodniji pad s brijega na dolinu. *Slika 8* prikazuje taj efekt. Postoji više načina za ovo postići. Na www.lighthouse3d.com António Ramires Fernandes[3] predlaže korištenje sinusnog vala za interpolaciju. Na taj način bi se doista vrlo efektivno dobila puno realističnija visinska mapa. Još jedan način, koji je općenitiji i primjenjiv na bilo koju visinsku mapu je zamagljivanje slike (*engl. blur effect*).

Ovaj je algoritam dosta spor i u svom izvornom obliku ne daje baš dobre rezultate. Potrebno je puno iteracija da bi se stvorila zadovoljavajuća visinska mapa. Uz male varijacije u algoritmu, kao npr. korištenjem sinusoidnog vala može se dobiti puno prirodniji krajnji izgled 3D terena.

## *2.4 Generiranje visinske mape kružnicama*

Ako se malo prouče svojstva krajobraza, može se primjetiti da postoje nekakva pravila po kojima se mogu stvarati brežuljci. Seizmološke aktivnosti i vremenski uvjeti su glavni uzročnici koji definiraju izgled svih krajobraza oko nas. Erozijom tla dobivaju se prilično pravilne deformacije koje za brežuljke često znače eksponencijalno usporenje rasta visine kako se krećemo prema vrhu. Sinusne funkcije imaju svojstvo eksponencijalnosti, pa će one biti iskorištene za stvaranje visinske mape ovom metodom.

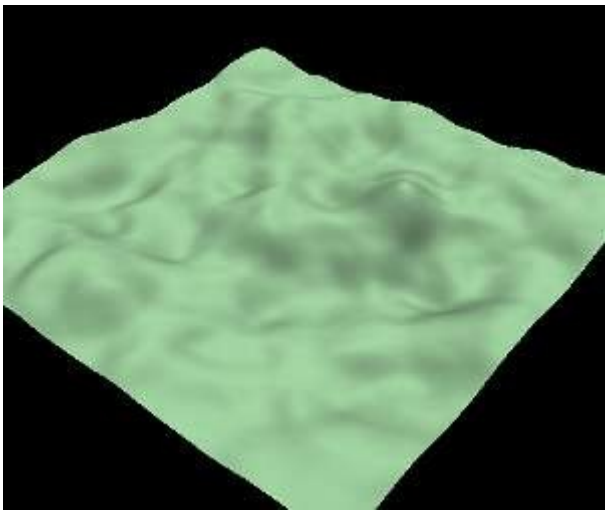
Izvorni algoritam[3] je dosta jednostavan i intuitivan. Prvi korak čini traženje središte brežuljka, odnosno kružnice kao dvodimenzionalnu, nasumično izabranu koordinatu. Potom je potrebno odabrati radijus, koji isto tako može biti nasumičan, ali u određenim granicama. Kroz nekoliko iteracija se visinska mapa deformira unutra te kružnice i stvara se brežuljak koji podsjeća na umjetni brežuljak.

Pseudokod algoritma za jednu iteraciju:

```
za svaku točku terena (tx,tz)
        pd=udaljenost_od_središta_kružnice*2/promjer_kružnice
        ako je(apsolutna_vrijednost(pd)<=1.0)
                visina(tx,tz) += odmak/2+cos(pd*3.14)*odmak/2;
```

Iz pseudokoda se vidi da je algoritam prilično spor, jer se mora prolaziti kroz svaku točku za svaku iteraciju, a osim toga koristi i *cos()* funkciju koja je jedna od sporijih funkcija. No, ako izuzmemo brzinu, stvoreni teren je prilično ugodan oku. U algoritmu *odmak* predstavlja najveću dopuštenu varijaciju točke terena .

Akumulacijom operacija stvaranja jednog brežuljka zbrajaju se visine svih brežuljaka koji se prostiru preko te točke i na taj se način dobivaju iznimno glatki prijelazi i stapanja više brežuljaka u jedan. To je posljedica zbrajanja kosinusnih funkcija koje su glatke, kontinuirane i njihovim zbrajanjem neće doći do diskontinuiteta ili nekakvih skokova na površini visinske mape. Primjer tako dobivenog terena nakon 1000 iteracija može se vidjeti na *slici 9*.



*Slika 9: Teren stvoren algoritmom generiranja visinske mape kružnicama*

Iz slike se može vidjeti da ovaj algoritam daje prilično impresivne rezultate. Naravno, kao i kod svih dosad spomenutih algoritama postoji bezbroj varijacija na osnovni algoritam, koje uglavnom koriste druge funkcije sličnih svojstava kao kosinus.

Dosad navedene tri metode generiranja terena su samo ideje kako stvoriti teren. Može ih se smatrati kosturima za daljnji razvoj jer ostavljaju velik prostor za fino ugađanje detalja koji su često jako bitni. Kod rješavanja problema generiranj terena teži se u što kraćem vremenu izračunati visine svake od točaka na visinskoj mapi i to tako da se dobije visinska mapa koja po izgledu podsjeća na stvarni krajobraz. To nije lako postići i po brzini i dobivenom rezultatu algoritam generiranja visinske mape dijeljenjem je najbliži. Njegova kompleksnost je O(n^2), a najveća mu je mana u tome što točke prve iteracije na rubovima obično imaju prevelik utjecaj na izgled terena, pa se često na prvi pogled može vidjeti da je teren generiran tom metodom.

Naravno, nemoguće je odrediti koji je algoritam najbolji, jer svaka od ovih ideja stvara visinsku mapu određenih svojstava koji se mogu više ili manje iskoristiti u određenu svrhu.

# 3. Dodavanje detalja

Dosad je uglavnom bila riječ o deformacijama visinske mape da bi dobili realistični izgled terena, što je najvažniji dio pri stvaranju 3D terena. No, nitko ne želi gledati sivu, neosjenčanu površinu koja bi trebala predstavljati teren. Svatko voli vidjeti zanimljive detalje u vanjskom prostoru. Na primjer, šuma se može kao jedna krajnost predstaviti samo kao tekstura ako znamo da je pozicija kamere uvijek visoko iznad terena (dobar primjer je Google Earth), no ako se radi o nekakvoj igri i igrač mora prolaziti kroz šumu, potrebno je dodati stvarne modele vegetacije, čak bi bilo dobro i animirati ih i slično.

U ovom poglavlju nećemo se baviti toliko kompleksnim stvarima, jer to već prelazi u problematiku stvaranja igara. Početi ćemo s jednostavnim načinom dodavanja detalja kao što su teksturiranje i osvjetljenje.

## *3.1 Teksture terena*

Teksturiranje je po definiciji pridruživanje teksture 3D modelu. Ovisno o obliku modela, odnosno načinu iscrtavanja, svakom vrhu se pridružuje *u* i *v* koordinata teksture, te se ta tekstura rasterizira u protočnom sustavu OpenGL-a prilikom rasterizacije.

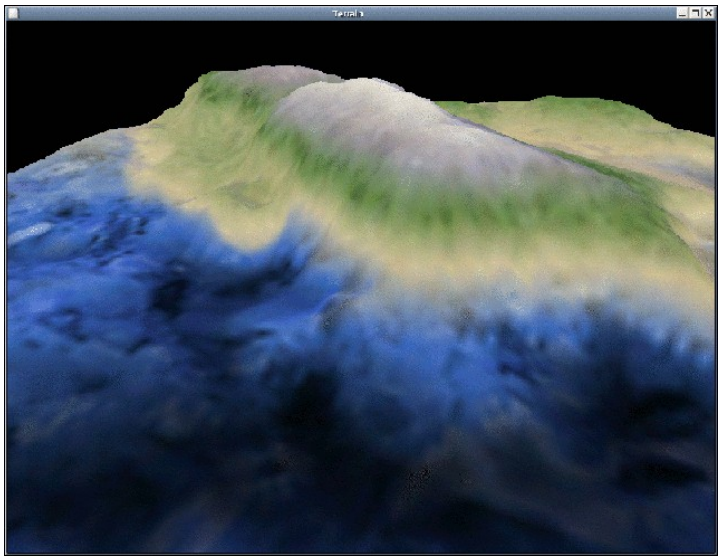
Tekstura daje dodatnu detaljnost generičkom poligonu u računalnoj grafici. Međutim, postoji više vrsta tekstura i više metoda teksturiranja. Tako će se ovdje spomenuti neke od njih koje se koriste da bi se stvorila više ili manje realistična tekstura terena.

### 3.1.1 Proceduralno generiranje tekstura

Proceduralno generiranje tekstura[4] je jednostavna metoda algoritamskog stvaranja teksture u računalnoj grafici. Kod stvaranja teksture za teren se za svaki vrh u prostoru proračunava boja. Znači ako imamo veličinu visinske mape 512x512, tih dimenzija će biti i generirana tekstura. Iz toga slijedi da će svaka točka teksture biti dodijeljena jednom vrhu u prostoru.

Sličan efekt se mogao postići postavljanjem boje pomoću funkcije *glColor3f()*. Razlog zašto se to ne radi, je to što tekstura ima brojne prednosti pred postavljanjem boje, kao što je npr. to da se nad teksturom mogu obavljati brojne operacije, tekstura se može komprimirati, kompaktno je zapisana, jednostavno ju je izračunati unaprijed i poslati u serversku memoriju, bez ikakvih problema se može zamijeniti s detaljnijom teksturom bez da se nešto mijenja u programskom kodu i sl. *Slika 11* prikazuje kako bi mogla izgledati tako generirana tekstura, a *Slika 10* visinsku mapu u prostoru na koju je zalijepljena ta tekstura[5].

*Slika 11: Proceduralno generirana tekstura*



*Slika 10: Teksturiran 3D teren*

Pseudokod algoritma za proceduralnu generaciju teksture terena[5]:

```
procedura
stvori_teksturu(visinska_mapa, podrucja, nova_tekstura)
za svaki (i,j ε visinska_mapa)
          h=visinska_vrijednost na visinska_mapa[i][j]
          v=0
          za svaki r u podrucja
                    v+=tezina(h,r)* vrijednst_boje r.tekstura na [i][j]
          upisi v u nova_tekstura na poziciju [i][j]


procedura
tezina (visina, podrucje) -> tezina ([0,1.0] float)
          w=(podrucje.domet – abs(visina – podrucje.max))/podrucje.domet
          vrati (w<0) ? 0 : w


struktura region{
          tekstura  # tekstura za ovaj tip terena
          domet    # podrucje pokriveno ovim tipom teksture
          max      # gornja granica ove teksture
          }
```

Algoritam uzima kao parametre visinsku mapu, teksture različitih područja (*engl. region*), te alociranu memoriju za novu teksturu. Prolazeći kroz visinsku mapu vrijednosti *v* se dodaje mješavina boja koje se nalaze u teksturama područja. Ta mješavina se određuje na temelju težina koje se računaju u proceduri *tezina*. Područje je struktura koja sadrži informaciju o teksturi, širini pojasa (varijabla *domet*), te vrijednosti gornje granice područja.

Postoji i jednostavnija varijanta ovog algoritma koja ne uzima u obzir težinu susjednih područja pa se događaju skokovi iz jedne teksture u drugu, što ne izgleda baš lijepo.

Područja se uglavnom definiraju prema visini, dakle jedno od mogućnosti je:

Tablica 1: Definicije pojasa pojedinih područja:

| |
|---|
| voda: [0, 50] |
| pijesak : [51, 101] |
| trava: [102, 152] |
| kamenje : [153, 203] |
| snijeg : [204, 255] |

Prema *tablici 1* se vidi ovisnost tipa terena o visini. Dodavanjem nasumičnog odstupanja tih granica na vrhovima terena se može dobiti još i bolji dojam, no algoritam se komplicira. Kompleksnost algoritma nije toliko bitna, jer se ovi proračuni obavljaju samo jednom, te se potom pohranjuju u memoriju, oslobađajući procesor tog zadatka prilikom izvršavanja (*engl. runtime*), što nam je i cilj. Iznimka za nasumično odstupanje od pojasa je voda, jer ne bi bilo baš u redu da tekstura vode postoji npr. na padini nekog brežuljka.

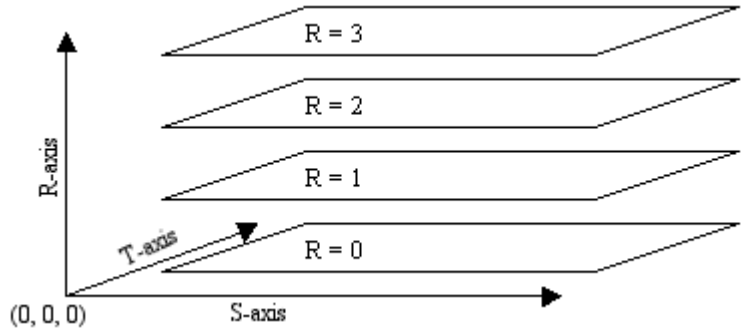
Iz *slike 10* se vidi da smo uspješno dobili osnovnu boju na terenu, no, nitko ne može reći da je to lijep teren, jer ne postoje detalji. Detalji se proceduralnim teksturama često dodaju dodatnim teksturama sa šumom. Naime, današnje grafičke kartice imaju više od jedne jedinice za teksturiranje, pa je moguće jednu od njih zadužiti za operacije s proceduralno generiranom teksturom, a drugu za dodatnu, detaljniju teksturu koja sadrži šum i koja se može ponavljati preko svih područja.

Moguće je i rješenje s proceduralnom teksturom koja je veće dimenzije od same visinske mape, no to je vrlo neelegantan način teksturiranja, jer se okupira ogroman prostor memorije samo za tu teksturu.

### 3.1.2 Teksturiranje 3D teksturama

Jedna zanimljiva mogućnost OpenGL API-ja je to da postoje volumne teksture koje nemaju samo *s* i *t* koordinatu(koje odgovaraju koordinatama *u* i v u 2D teksturama), već i dubinsku koordinatu *r*. To omogućava programeru da svakom vrhu u prostoru dodijeli ekvivalentnu točku u prostornoj teksturi.

Dobar članak koji opisuje korištenje 3D tekstura je od Doug Sheetsa, pod nazivom: *An OpenGL 3d Texturing Tutorial* [6], *slika 12* prikazuje na koji način se može zamisliti 3D tekstura.




*Slika 12: Izgled prostorne teksture*

Na slici se može vidjeti da se koriste četiri teksture kao osnova prostorne teksture. *r* koordinata predstavlja težinski faktor za miješanje dvaju susjednih tekstura.

Teksture izmješane na takav način su vrlo pogodne za korištenje prilikom iscrtavanja terena.

Stvaranje ove teksture se obavlja na slijedeći način:

1. učitavanje ili stvaranje tekstura
2. traženje imena za teksturu od OpenGL-a
3. vezanje na ime
4. određivanje parametara teksture kao što je omatanje
5. pozivanje *glTexImage3D()*
6. korištenje teksture prilikom izravnog crtanja (*glTexCoord3f()*)ili pohranjivanje u serversku memoriju za kasnije pozivanje

Potrebno je reći nekoliko napomena u vezi stvaranja 3D teksture; prvi korak podrazumijeva da se niz tekstura koje želimo koristiti mora smjestiti u polje jedna iza druge najčešće kao niz bajtova. To bi značilo za prošlu ilustraciju da se prvo pohrani tekstura s $r = 0$, pa s $r = 1$ itd. Zauzeće meorije ovako stvorene teksture iznosi (širina*visina*dubina*bajtova po pikselu) bajtova. I za ovaj tip teksture vrijedi da sve dimenzije moraju biti vrijednosti oblika 2^n i da je *r* koordinata ukliještena između 0 i 1, isto kao i s i t.

*glTexImage3D()* je funkcija koja nije dio OpenGL 2.1, već je njegova ekstenzija. Prije korištenja se mora dohvatititi pokazivač na tu funkciju. Operacija je specifična za platformu, no postoje biblioteke koje to obavljaju prenosivo kao što je GLEW (*OpenGL Extension Wrangler Library*) ili GLEE (*GL Easy Extension library*). Jednom kada je to napravljeno, funkcija se koristi kao i svaka druga. Gore navedene biblioteke sadržavaju i neke makro definicije kao što je *GL_TEXTURE_3D*, koja je potrebna da bi se 3D tekstura uopće mogla u trećem koraku vezati na međuspremnik.

*Slika 13* prikazuje kako može izgledati teren teksturiran pomoću 3D tekstura. Rezultat je jako dobar, jer se koriste samo nekoliko detaljnih tekstura koje se miješaju sklopovski na grafičkoj kartici. Sličan rezultat smo mogli dobiti i proceduralnim teksturiranjem, samo ne bi postojalna takva detaljnost kada bi se približili poligonu.

Lijepljenje teksture za poligone se vrši slično kao i kod proceduralnog stvaranja teksture, znači na temelju visinskih pojasa se određuje koja se od nekoliko tekstura koje predstavljaju konfiguracije terena vidi i koliko, ovisno o parametru *r*.

Ovo bi bio vjerojatno najbolji način dodavanja tekstura na teren da ne postoji jedan veliki problem. Kod prikazivanja terena često koriste LOD algoritmi koji izbacuju pojedine vrhove koji su suvišni u datom trenutku za trenutnu poziciju kamere. Događa to da se, kada se izbaci vrh, izbaci i tekstura koja mu je bila dodijeljena. To se u svakom slučaju želi izbjeći, jer je sama svrha teksture da na neki način zamaskira nestajanje vrhova koji nisu bitni. Ovaj problem se pravilnim teksturiranjem ne javlja kod proceduralno generirane teksture.

*Slika 13: Rezultat korištenja 3D teksture*

### 3.1.3 Teksturiranje miješanjem teksturama

Ova metoda predstavlja svojevrsnu zlatnu sredinu između proceduralno generirane i 3D teksture. Miješanje tekstura je tehnika koja gotovo ne koristi CPU, no zato dosta opterećuje grafičku karticu. Ulazni podaci su opet teksture koje predstavljaju pojedino visinsko područje, te mapa prozirnosti, koja je ništa drugo do tekstura s nijansama sive boje koja na svojim koordinatama sadrži težinske faktore za miješanje dviju tekstura. Crna boja predstavlja jednu, a bijela drugu teksturu, ostale vrijednosti nam govore koliki je udio svake od tih tekstura na određenoj *u*-*v* poziciji.

Prema Charlesu Bloomu[7] najbolje je podijeliti teren u blokove 32x32 sa 33x33 vrha. Za svaki blok se mora pronaći koje teksture utječu na njega, odnosno teksture koje se nalaze na susjednim elementima. Svaki se blok od 33x33 vrha sprema u spremnik vrhova. Isto tako se grupiraju svi ti elementi spremnika vrhova na kojima se nalazi ista tekstura, te se spremaju u listu trokuta.

Za svaki blok se pošalje element iz spremnika vrhova koji mu odgovara. Nakon toga se pošalje svaka od tekstura za miješanje i nakon svake izmjene teksture se poziva lista trokuta koja odgovara toj teksturi.

Za miješanje je potrebna mapa prozirnosti. Ona se mora generirati iz informacija koje imamo o našem terenu. Informacije koje su nam potrebne su one o teksturama na susjednim blokovima. Za svaku listu trokuta se stvara tekstura koja prekriva cijeli blok, a dvostruko je veće dimenzije od broja elemenata bloka. Za svaki se teksturni element mora odrediti težina prozirnosti na temelju susjednih elementa u bloku koji čine matricu 3x3 elemenata. Što je trenutno promatrani blok dalje od središnjeg, to je veća prozirnost teksture centralnog elementa. Formula po kojoj se to računa izgleda ovako:

$$tezina(tocka1, tocka2) = 1 - \frac{(tocka1^2 + tocka2^2)}{1.75^2}$$

*Formula 1: Težina za mapu prozirnosti na temelju udaljenosti*

Iz *formule 1* se vidi da će za udaljenost od 1.75 težina iznositi 0, isto kao i za sve veće vrijednosti udaljenosti.

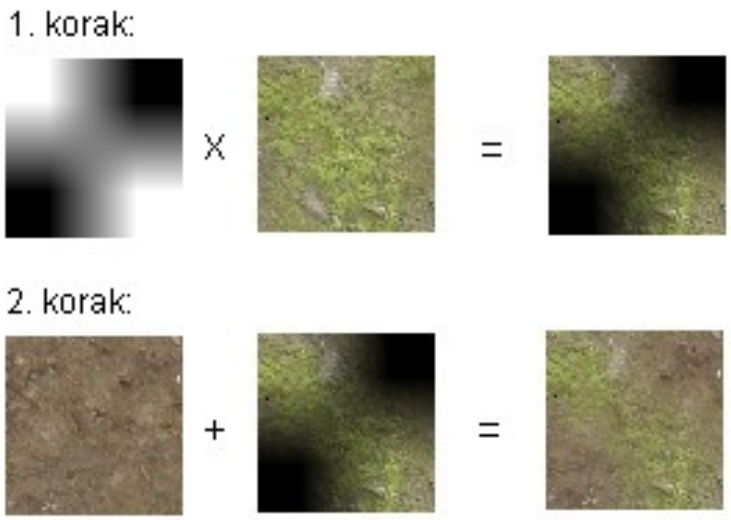
Mapa prozirnosti se mora proračunavati sa svaku novu teksturu za miješanje. Pritom se rezultat prošle operacije miješanja mora normalizirati i tek potom miješati sa novom teksturom.

Već se tu vidi da je potrebno više prolaza da bi se dobio željeni efekt, što je za grafičku karticu jako intenzivno. Gotovo se sav posao nalazi na rasterizacijskoj jedinici. Ako se želi implementirati u fiksnoj protočnoj strukturi, ovaj postupak zahtijeva dvije jedinice za teksturiranje; jedna za mapu prozirnosti, a drugu za teksturu za miješanje.

Implementacijom u programu za sjenčanje vrhova[9] se omogućava korištenje RGB kanala umjesto mape prozirnosti. Pošto mapa prozirnosti koristi sva tri kanala i tako stvara samo nijanse sive, bespotrebno se troši memorija, a i potrebno je više prolaza, jer bi se za svaki prijelaz između dvije teksture za miješanje mogla koristiti jedna komponenta boje. Posljedica svega toga je manje zauzeće memorije i što je još važnije, mogu se sve četiri teksture za miješanje pomiješati u samo jednom prolazu.

Proces teksturiranja miješanjem tekstura se za dvije teksture vrši u dva koraka[8]:

- mapa prozirnosti se pomnoži jednom od tekstura za miješanje
- druga tekstura za miješanje se zbraja s dobivenom teksturom




*Slika 14: Postupak stvaranja teksture miješanjem dvaju tekstura*

*Slika 14* pokazuje na koji način se stvara nova tekstura operacijama nad teksturama. Za ove operacije je najbolje koristiti jedinicu za kombiniranje tekstura (*OpenGL texture combiner*) koji omogućuje manje prolaza korištenjem više jedinica za teksturiranje. Bijela boja ima vrijednost (1.0,1.0,1.0) pa se množenjem s njom zadržava vrijednost druge teksture, a zbrajanjem s crnom bojom (0.0,0.0,0.0) opet daje za rezultat teksturu koja nije maska.

Dobre strane ove metode su da je prilično jednostavna i daje jako dobre rezultate. Koristi relativno malo memorije, jer se puno puta ponovo iskorištavaju iste teksture, jedino se za svaki element u bloku mora stvarati mapa prozirnosti za svaku teksturu za miješanje(ili samo jednu ako se koristi jedinica za sjenčanje) što je prihvatljivo. Današnje grafičke kartice su dovoljno snažne čak i za više prolaza teksturiranja, no ipak je dosta bolje rješenje koristiti jedinicu za sjenčanje, jer se štedi na prolazima i na memoriji.

Korištenjem jedinice za sjenčanje sve se mape prozirnosti stapaju u jednu koja nije u nijansama sive već u RGB bojama. Primjer kombinacije dvije mape prozirnosti u jednu RGB se može vidjeti na *slici 15*.


*Slika 15: Kombiniranje dvije mape prozirnosti u jednu*

## *3.2 Sjene na terenu*

Sjene su oduvijek bile problem u računalnoj grafici, prvenstveno zbog broja operacija koje je potrebno obaviti da bi se uspješno bacila sjena na scenu. Osnovna ideja je metoda praćenja zrake. Kako je to praktički neizvedivo u stvarnom vremenu, postoje druge metode kao što su volumeni sjene i mape sjena

### 3.2.1 Volumeni sjena

Ideja kod volumena sjena je da svaki objekt koji baca sjenu na područje u prostoru definiranom kao volumen. Spremnik maske (*engl. Stencil Buffer*) se potom koristi ta pronalaženje presjecišne točke između objekata u sceni i sjene. Volumeni sjena su konstruirani od zraka svjetlosti koji na rubovima presijecaju objekt koji baca sjenu, te potom nastavljaju svoj put izvan scene. To nam daje poligonalnu površinu objekta čiju sjenu želimo baciti. Stenicl buffer se koristi za za računanje koji dio objekta je unutar volumena. Za svaki piksel na sceni vrijednost maske će se povećati ako je granica za volumen sjene presječena na putu u sjenu i smanjiti ako je granica presječena na putu van. Operacija spremnika maske je namještena samo za povećavanje/smanjivanje kada je test na dubinu prošao. To znači da će svi pikseli čija vrijednost maske nije jednaka nuli biti identificirani kao dijelovi objekta koji su u sjeni. Na taj se način stvara maska.

Pri drugom prolazu samo se osvježavaju pikseli s vrijednošću maske 0. Na taj se način dijeli scena na dijelove koji su u sceni i dijelove koji nisu.

Potpuni redoslijed promjene stanja izgleda ovako[10]:

1. Spremnik boje (*engl. Color Buffer*) i spremnik dubine (*engl. Depth Buffer*)se otvaraju za pisanje i testiranje dubine se aktivira
2. Postavljaju se atributi za crtanje sa sjenčanjem, izvor svjetlosti ugašen
3. Iscrtava se cijela scena
4. Pronalaze se poligoni koji su unutar volumena sjene
5. Isključuje se spremnik boje i spremnik dubine ali se testiranje dubine vrši i dalje
6. Spremnik maske se postavlja na 0 ako je oko izvan volumena sjene, 1 ako je unutar
7. Postavljanje funkcije maske da uvijek prolazi
8. Postavljanje operacije nad maskom da se poveća ako test prođe
9. Uključivanje izrezivanje stražnjeg dijela poligona
10. Iscrtavanje poligona volumena sjene
11. Postavljanje operacije nad maskom da se smanji ako test prođe Uključivanje izrezivanja prednjeg dijela poligona
12. Iscrtavanje poligona volumena sjene
13. Postavljanje funkcije maske da provjerava ako je vrijednost jednaka 0
14. Postavljanje operacije nad maskom da ne radi ništa
15. Uključivanje svjetala
16. Iscrtavanje scene

Ova metoda je vrlo efikasna za stvaranje točnih sjena u realnom vremenu, koristi se u Doom 3 pogonu i baš se u toj igri može vidjeti fantastične rezultate igre svjetla i sjene na sceni. Proces se sastoji od dosta koraka i svi oni koriste grafičku karticu kao jedinicu za obavljanje operacija, dok mi želimo rasteretiti grafičku karticu. Usprkos tome, realno gledano iz stajališta opterećenja grafičke kartice ne postoji pravi razlog za ne korištenje ove metode iscrtavanja sjena. Današnje su grafičke kartice iznimno brze i u potpunosti su optimizirane baš za ovakav posao.

Korištenja volumena nije pogodno za korištenje zajedno s terenima zato što nije prijateljski za implementaciju s LOD algoritmima za prikazivanje terena. U suprotnom se ponovno javlja problem izrezivanja nevažnih vrhova s kojima nestaje i njihova sjena. U fiksiranom sustavu gledanja na teren iz ptičje perspektive u kojem se ne koriste LOD algoritmi je ovakva metoda odlična jer ne samo da će osjenčati teren već i sve aktivne i pasivne objekte na terenu bez posebnih promjena u kodu, te je moguće animirati pomicanje izvora svjetlosti (npr. simuliranje dana i noći).

### 3.2.2 Mape sjena

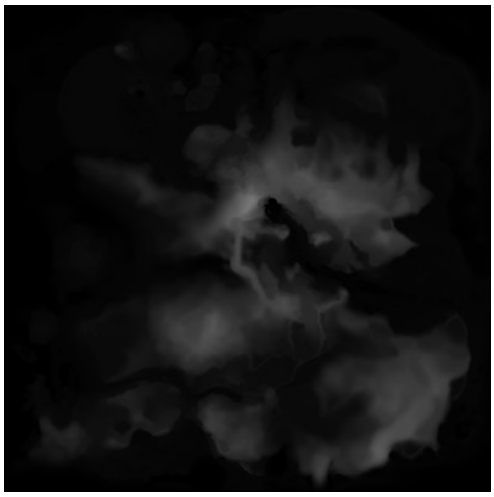
Mape sjena su statička simulacija osvjetljenja teksturom. Računanjem efekata svjetlosti unaprijed dobivaju se teksture koje se samo zalijepe na poligone i kao takve daju osjećaj osvjetljenosti tako teksturiranog objekta. Jedina mana mapa sjena je to što nisu dinamičke i nije ih moguće brzo izračunati, pa se tako unaprijed izračunate najčešće ne mijenjaju u realnom vremenu, iako postoje i pokušaji toga. Gledano iz zahtjeva koji nama trebaju za osvijetljenje, ovo je odličan način simulacije svjetlosti. Takva tekstura je u stanju maskirati nestajanje naših vrhova i potencijalno sakrila efekt iskakanja (*engl. popping effect*) koji se javlja pri promjeni razine detalja terena.

Za mape sjena nam je potrebna pozicija izvora svjetlosti, te poznavanje visinske mape. Postoje dvije komponente sjene:

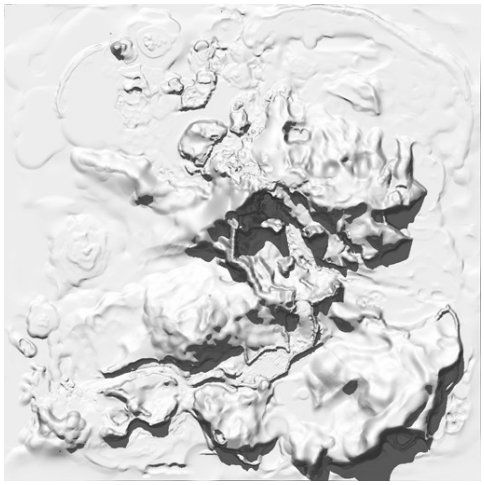
- sjene dobivene na temelju nagiba u odnosu na izvor svjetlosti (obično sjenčanje)
- sjene dobivene bacanjem sjene na samog sebe

Prva komponenta sjene je identična kao i sjenčanje u OpenGL-u korištenjem ugrađenih funkcija osvijetljenja.

Druga komponenta predstavlja veći izazov za izračunati, no tehnički gledano ne unosi puno poboljšanja u osjećaj prostornog osvijetljenja terena zato što se efekt maskiranja i osjećaj dubine terena može sasvim solidno postići samim sjenčanjem. Ipak, stvarne sjene daju pravu ljepotu terenu, naročito pri scenama gdje je izvor svjetlosti nisko, pa višlji dijelovi terena bacaju sjenu na niže dijelove.[12]



*Slika 16: Primjer generirane sjene na temelju nagiba terena*



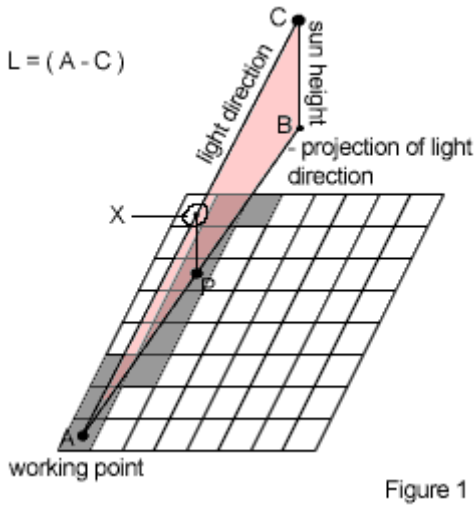
*Slika 17: Primjer generirane mape sjene na temelju samozasjenjivanja*

Algoritam koji obavlja generiranje mape sjena zahtijeva mrežu, odnosno dvodimenzionalno polje. Za svaki element mreže se provjerava da li zraka iz izvora svjetlosti prema toj točci presijeca visinsku mapu terena. Da bismo provjerili događa li se to moramo stvoriti projiciran pravac na teren po kojem se ispituju točke mreže.

Da bi pojasnili o čemu je riječ potrebno je definirati neke parametre (*slika 18*):

- *A* će biti trenutna točka u mreži
- *B* – koordinate pozicije izvora svjetlosti s visinom jednakom 0

  (vector *B* = vector(poz_svjetla.x, 0, poz_svjetla.z) );
- *C* – koordinate pozicije izvora svjetla
- *L* – vektor smjera svjetlosti ( *L*=*A*-*C*)
- *P* – točka koja leži na projekciji vektora *L*
- *X(p)* – koordinate na vektoru *L* čija projekcija je točka *P*




*Slika 18: Postupak provjere zasjenjenosti*

Niz točaka *P* leži na 2D pravcu, koji se može generirati algoritmom iz točaka *A* i *B*. Za svako polje *P* postoji odgovarajuće polje u visinskoj mapi.

Ako je vrijednost visinske mape na lokaciji $A > X(P)$, može se prekinuti postupak za trenutnu točku *A* i postavlja vrijednost sjene na tu lokaciju pošto je ta točka mreže u sjeni. U suprotnom, osvjetljenje se računa sjenčanjem formulom:

$$osvjetljenje(A) = ambijentalnaSvjetlost + (L \times N)$$

*Formula 2: Računanje ambijentalne i difuzne komponente*

,

gdje je *N* vektor normale na površinu terena, a ambijentalna svjetlost predstavlja trokomponentnu boju ambijentalne svjetlosti u RGB formatu.

Mirce Marghidanu u svom članku *Fast Computation of Terrain Shadow Maps* [11] predlaže implementaciju kao dvije funkcije. Prva funkcija provjerava da li je točka *A* zaklonjena i poziva se za svaku točku u visinskoj mapi. Druga funkcija koristi prvu da bi obavila proračun osvjetljenja u točci *A* na temelju vraćene vrijednosti iz prve funkcije.

Pseudokod funkcije za pronalazak presjecišta [11]:

```
int presjeciste_mape(vektor iv, zraka r, visinska_mapa *hm, float
skalirana_visina){
          int pogodak
          float D,d,h
          vektor dir,v=iv+r.smjer

          dok_su(v.x i v.z unutar granica){
          D=duzina(vektor(v.x,0,v.z)-vektor(r.izvor.x,0,r.izvor.z))
          d=duzina(iv-v)
          h=iv.y+(d*r.izvor.y)/D          //X(P)
          ako_je(hm->podaci[ifloor(v.z)* visinska_mapa.sirina +

          ifloor(v.x)]*skalirana_visina> h){
                    pogodak++
                    break
          }
          dir=r.smjer
          dir.y=0
          v+=normaliziraj(dir)
          }
          vrati pogodak;
}
```

Pseudokod funkcije za stvaranje mape sjene na temelju prolaska kroz visinsku mapu:

```
visinska_mapa *stvoriMapuSjene(char * normala, visinska_mapa *hm, vektor izvor_svjetla, int w, float ambijentalno ){
          int i,j, pogodak
          float f,dot
          vektor n,vrh
          visinska_mapa *mapa_sjene
          zraka r
          float  skalirana_visina=10/255;
          alociraj(mapa_sjene,w*w)
          za_svaki(j<w)
          za_svaki(i<w)
          {
          vrh.x=i
          vrh.y=hm->podaci[j*w+i] * fHeightScale
          vrh.z=j

          f=ambijentalno
          r.izvor=vrh+izvor_svjetla *2000
          r.smjer=izvor_svjetla.smjer

          ako(!presjeciste_mape(vrh,r,hm, skalirana_visina)){
          n.x = (float)(normal[3*(j*w+i)+0]);
          n.y = (float)(normal[3*(j*w+i)+1]);
          n.z = (float)(normal[3*(j*w+i)+2]);
          f+=0.5*(1.0+skalarni_umnozak(normaliziraj(n),

          normaliziraj(izvor_svjetla)));
          ako_je(f>1.0)
                    f=1.0

          }
          dot=f*255
          mapa_sjene[j*w+i] = (unsigned char)dot;

          }
          vrati mapa_sjene;
}
```

# 4. Postupci prikazivanja 3D terena

Tereni se često sastoje od ogromnog broja točaka u visinskoj mapi, odnosno vrhova u grafičkoj protočnoj strukturi. Karakteristika čovjeka je da mu je dovoljna konačna količina vizualnih detalja da bi prepoznao neki oblik. To se svojstvo jako često koristi, ne samo u grafici, već i u obradi zvuka, kod npr. mp3 formata, gdje se filtriraju zanemarive frekvencije. Tako se smanjuje veličina datoteka, i što je još bitnije za naš kontekst, smanjuje se protok informacija u stanju izvršavanja. Važno je naglasiti da u slučaju prikazivanja terena nije toliko bitna veličina terena u datoteci na disku koliko je bitno da se u stvarnom vremenu prilikom prikazivanja terena ne koriste čovjeku zanemarive informacije. S obzirom da je dojam subjektivan, više detalja je uvijek bolje.

Algoritmi za prikazivanje 3D terena obavljaju na mreži točaka, odnosno na dvodimenzionalnoj visinskoj mapi za koju se podrazumijeva da su po X i Z osi točke pravilno razmaknute za točno određenu vrijednost. To je mana iz razloga što smo diskretizirali te dvije osi, gubeći detalje, no to je *de-facto* postao standard prikazivanja terena. Naravno, postoje i programi koji koriste svoje formate zapisa terena koji odstupaju od ovog pravila.

Postoji nekoliko algoritama koji su se kroz zadnjih desetak godina pokazali kao najbolji i čije su specifikacije poslužile za stvaranje mnogih hibrida:

- Optimalno prilagođavajuće mreže u stvarnom vremenu – ROAM
- Geometrijsko MIP preslikavanje

## 4.1. Optimalno prilagođavajuće mreže u stvarnom vremenu - ROAM

ROAM algoritam se sastoji od predprocesne komponente i nekoliko komponenti koji se izvršavaju prilikom prikazivanja. Predprocesna komponenta stvara ugniježđene granice pogrešaka za binarno stablo trokuta.

Ostale faze su:

- rekurzivno, inkrementalno osvježavanje vidnog polja
- prioritetno osvježavanje samo za izlazne trokute koji se potencijalno mogu podijeliti/spojiti u 3. fazi
- triangulacija korištenjem pohlepnih koraka dijeljenja i spajanja u dva prioritetna reda
- osvježavanje traka trokuta koje su zahvaćene promjenama izrezivanja iz faza 1 i podjelama/spajanjima iz faze 3

Osim navedenih faza, postoji i 12 kriterija opće primjene:

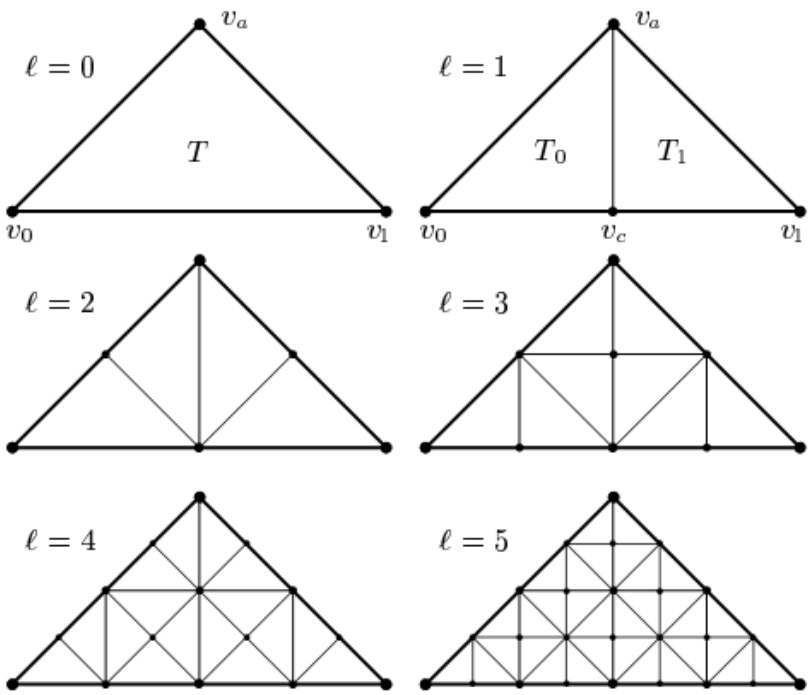
1. Vrijeme potrebno za postizanje zadanog broja trokuta – ova implementacija može održavati optimiziranu mrežu koja sadrži tisuće vrhova pri 30 fps za brze letove na niskoj visini iznad terena.
2. Fleksibilnost u odabiru načina određivanja pogreške na temelju pogleda – koristi se maksimalno geometrijsko izobličenje u zaslonskom prostoru. Moguće je jednostavno poboljšati račun tako da bi se osigurala pravilna vidljivost.
3. Prikazivanje mreže – binarno stablo je jezgra operacija dijeljenja/spajanja
4. Jednostavnost algoritma – jednostavnost proizlazi iz toga što se algoritam okreće oko operacija spajanja i dijeljenja trokuta u binarnom stablu. Sve se operacije obavljaju na trokutima, koje kao osnovne poligonske jedinice u računalnoj grafici dijeljenjem i spajanjem ne proizvode artefakte kao što su pukotine (*engl. cracks*) i

općenito su jednostavni za manipulaciju.

5. Kvaliteta mreže za zadani broj trokuta – stvara se optimalna mreža u smislu smanjivanja najveće pogreške za trenutne monotone granice
6. Upravljanje brojem trokuta – ROAM stvara mreže sa specifičnim brojem trokuta
7. Ograničavanje frekvencije izmjene okvira (*engl. framerate*) – na temelju danog vremena za svaki okvir mogu se ograničiti operacije ROAM-a
8. Garantirane granice pogrešaka – ROAM proizvodi garantirane granice pogrešaka na temelju izobličenja zaslonskog prostora. Te se granice dobivaju lokalno pretvaranjem iz predprocesiranih granica u granice zaslonskog prostora ovisnog o poziciji.
9. Memorijski zahtjevi – Predprocesna veličina je jednaka broju uzoraka visinske mape plus jednostruka "debljina" vrijednosti po trokutu u binarnom stablu. Prilikom izvršavanja strukture zahtijevaju prostor proporcionalan izlaznoj veličini mreže.
10. Dinamičnost terena – mogućnost deformiranja terena uz uvjet da se ponovo obavi predprocesiranje.
11. Smanjeno iskakanje vrhova (*engl. popping*) – korištenjem mjere za zaslonsko izobličenje i tendencija da se obavlja malo promjena na mreži po okviru smanjuje efekt iskakanja vrhova. Moguće je i dodati stapanje vrhova (algoritam kojim se postiže gladak prelazak između dvije pozicije vrha).
12. Općenite ulazne mreže – ROAM nije usko vezan uz terene, već se može koristiti i za generičke oblike mreža vrhova

### 4.1.1 Binarna stabla trokuta

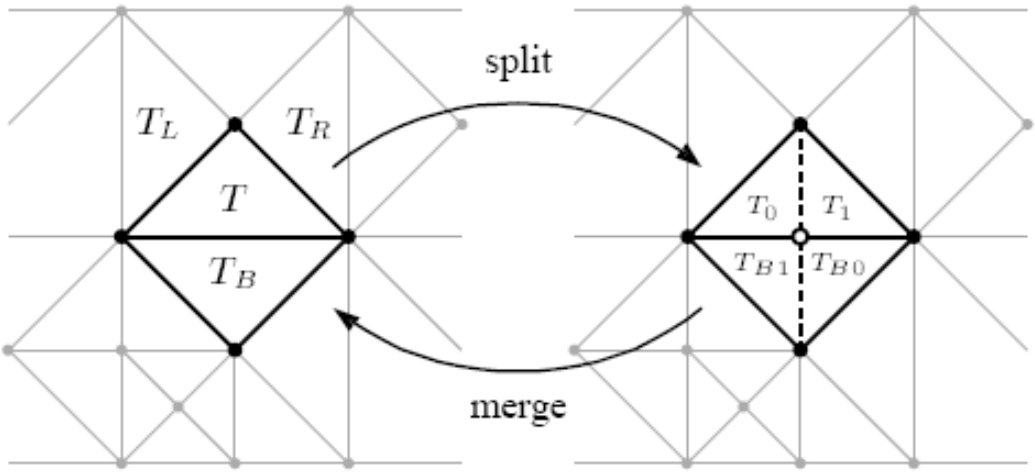
Operacije nad binarnim stablima trokuta predstavljaju srce ROAM algoritma. Binarno stablo trokuta je struktura podataka koja sadrži informacije o trokutima, njihovoj djeci i njihovim susjedima. Ako označimo korijenski čvor stabla s $T = (v_a,v_0,v_1)$ na dubini $l = 0$, na slijedećoj dubini $l = 1$ ovaj se trokut dijeli na svoja dva djeteta. Prije toga potrebno je stvoriti novu točku na polovištu hipotenuze trokuta (između vrhova $v_0$ i $v_1$), vrh $v_c$. Lijevo dijete $T_0$ će se sastojati od skupa vrhova ($v_c$,$v_a$,$v_0$), a desno dijete $T_1$ od ($v_c$,$v_1$,$v_a$). Rekurzivnim pozivanjem ovih operacija se dolazi do listova stabla.
Izgled binarnog stabla trokuta na pojedinim razinama vidi se na slici 19.




*Slika 19: Izgled binarnog stabla po razinama*

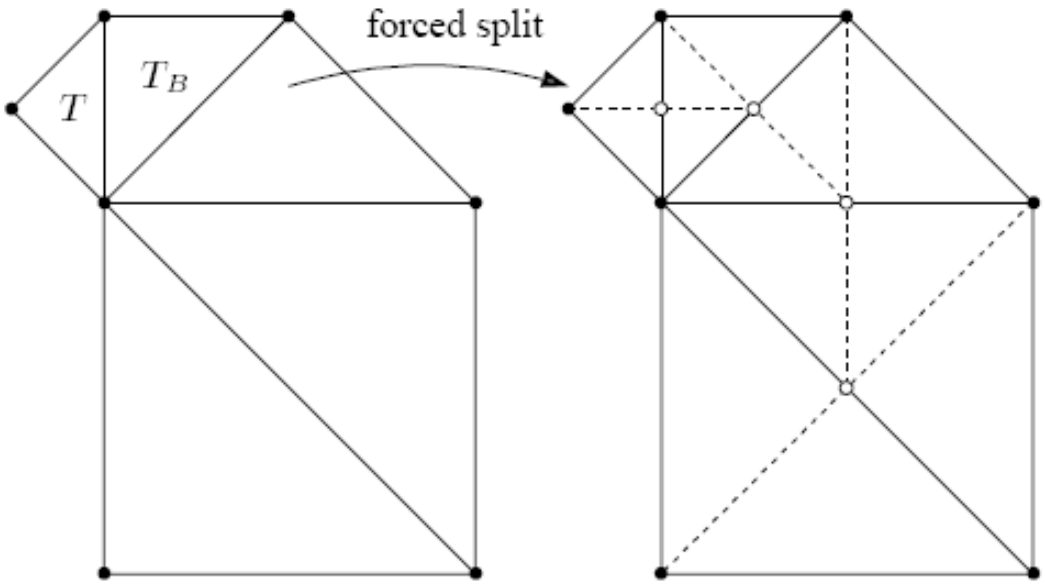
### 4.1.2 Dinamička kontinuirana triangulacija

U binarna stabla trokuta pohranjuju se vrijednosti visinske mape. Skup binarnih trokuta čini kontinuiranu mrežu gdje se bilo koja dva trokuta ne preklapaju, preklapaju na zajedničkom vrhu ili na zajedničkom rubu. Takve mreže nazivaju se *triangulacije.* Na *slici 20* vidi se tipično okruženje trokuta $T$ unutar triangulacije. $T_B$ je njegov susjed po hipotenuzi, te dijele rub $(v_0, v_1)$. $T_L$ je lijevi susjed ruba $(v_a, v_0)$, a $T_R$ desni susjed koji dijeli rub $(v_1, v_a)$. Susjedi trokuta $T$ mogu biti iste razine, više ili niže razine od trokuta $T$. Ako su $T$ i $T_B$ iste razine $l$, oni čine *dijamant*. Operacija dijeljenja dijamanta je jednostavna, jer se $T$ i $T_B$ samo zamjene svojom djecom $(T_0, T_1)$ i $(T_{B0}, T_{B1})$ i ujedno se aktivira točka u središtu dijamanta. *Slika 20* prikazuje taj slučaj. Za slučaj kada $T$ nema susjeda, samo se podjeli na $(T_0, T_1)$. Spajanje se može obaviti samo ako $T$ i $T_B$ tvore dijamant.




*Slika 20: Operacije dijeljenja i spajanja u tipičnom okruženju*

Bilo koja triangulacija se može postići slijedom podjela i spajanja. Trokut $T$ u triangulaciji ne može biti podijeljen odmah ako je $T_B$ grublje razine. Tu se uvodi prisiljavanje dijeljenja. $T_B$ mora biti prisiljen na dijeljenje, što može prouzročiti rekurzivno dijeljenje kroz strukturu. *Slika 21* prikazuje jedan takav scenarij.




*Slika 21: Prisiljeno dijeljenje*

### 4.1.3 Redovi spajanja i dijeljenja

Redovi spajanja i dijeljenja se baziraju na ideji održavanja prioriteta za svaki trokut u triangulaciji. Prioriteti opadaju s dubinom u stablu, što bi značilo da dijete ne može imati veći prioritet od roditelja. Postojanje reda spajanja omogućuje koherenciju između okvira, omogućujući i spajanje spojivih dijamanata gdje je potrebno.

Red dijeljenja koristi pohlepni algoritam koji pretpostavlja da su prioriteti monotono padajući po razinama.

Pseudokod reda dijeljenja:

```
T = korjen_triangulacije
za_sve T€T, umetni T u Qs
dok_je T previše mali ili neprecizan{
        identificiraj najveći prioritet T u Qs
        prisili dijeljenje T
        osvježi djeljenje:{
                ukloni T i druge podijeljene trokute iz Qs
                dodaj bilo koje nove trokute iz T u Qs
        }
}
```

Gdje $T$ predstavlja svaki trokut binarnog stabla, $Q_s$ red prioriteta koji sadrži sve trenutne trokute unutar ***T***, dok je ***T*** triangulacija.

Pretpostavimo da imamo prioritete promjenjive u vremenu $p_f(T) \in [0,1]$ za okvire $f \in (0,1,...)$ i zahtijeva se izgradnja triangulacija.Ako se prioriteti mijenjaju polagano i glatko, onda su optimalne triangulacije za bilo koja dva slijedna okvira slične. Zbog toga postoji i red spajanja. Spajanje će se u njemu događati na spojivom dijamantu ($T$, $T_B$) čiji je prioritet $\max\{p_f(T),p_f(TB)\}$

Algoritam slijedi:

```
ako_je f=0{
        T=bazna_triangulacija
        očisti Qs,Qm
        izračunaj prioritete za T-ove trokute i dijamante, pa
umetni u Qs i Qm prema potrebi
}inače{
        nastavi obrađivati T=Tf-1
        osvježi prioritete za sve elemente Qs, Qm
}
dok_je T manji od ciljane_veličine/preciznosti, ili je
maksimalni_prioritet
        podjela veći od minimalnog prioriteta spajanja{
        ako_je T prevelik ili previše precizan{
        nađi par najmanjeg prioriteta(T, TB) u Qm
        spoji(T,TB)
        osvježi redove:{
                ukloni svu spojenu djecu od Qs
                dodaj roditelje parova T,TB u Qs
                ukloni(T,TB) iz Qm
                dodaj sve novospojene dijamante u Qm
        }
}inace{
        identificiraj T najveceg_prioriteta u Qs
        prisili dijeljenje T
        osvjezi redovi:{
                ukloni T i ostale podijeljene trokute iz Qs
                dodaj nove trokute iz T u Qs
                ukloni iz Qm sve dijamante čija su djeca
podijeljena
                dodaj sve novospojene dijamante u Qm
                }
        }
}
postavi Tf = T
```

*Qm* je red spajanja, *Qs* je red dijeljenja, $\boldsymbol{T_f}$ je triangulacija u okviru *f*.
Inkrementalni pohlepni algoritam stvara optimalnu mrežu $\boldsymbol{T_f}$ koja ima iste prioritete kao da je algoritam dijeljenja od gore prema dolje obavljen na mreži.

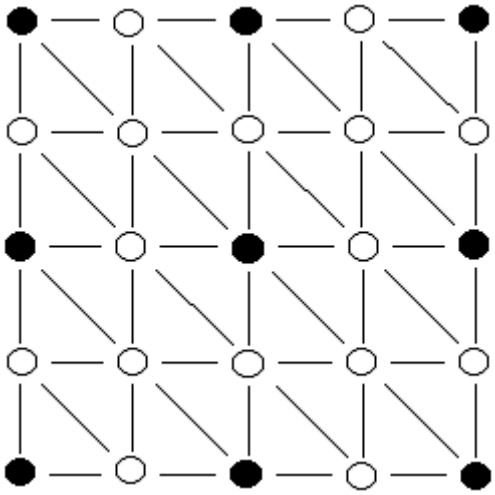
Općenito, prioriteti se računaju na temelju pogreške na svakoj od triangulacija. Mjere za pogrešku mogu se dobiti na više načina i u svojoj osnovi predstavljaju najveće odstupanje te triangulacije od pravih vrijednosti visine na visinskoj mapi, te uzimaju u obzir položaj kamere i perspektivu.

## 4.2. Geometrijsko MIP preslikavanje

Geometrijsko MIP preslikavanje[14] je sustav pločica različitih razina detalja koje se slažu jedna do druge, zajedno stvarajući teren. Opet se sve operacije obavljaju na mreži pravilno razmaknutih točaka po X i Z osi dimenzija 2^n+1 po obje osi, dok je Y os vrijednost visinske mape na toj poziciji. U originalnoj specifikaciji koriste se pločice dimenzija 17x17, uz uvjet da je dimenzija cijele visinske mape 257x257. Promjene dimenzija pločica utječu na brzinu iscrtavanja i subjektivni dojam. Osnovna struktura podataka za preslikavanje geometrijskih isječaka je stablo s četiri djeteta (*engl. quadtree*).

Stablo sadrži prostorne obujmice usporedne s osima (*engl. AABB bounding box*). Obujmice omogućuju izrezivanje nevidljivih pločica raznim algoritmima za određivanje vidljivosti (*engl. frustum culling*). Samo izrezivanje na ovaj način nije dovoljno dobro rješenje za smanjivanje detalja i brzo prikazivanje scene koja sadrži velik teren.

Geometrijski isječci terena se, ovisno o udaljenosti od oka/kamere, prikazuju kao isječci određene razine detalja. Lanac isječaka je naziv za slijedni skup isječaka, kod kojih je svaki isječak upola manji od svog prethodnika. Lanac isječaka se može unaprijed stvoriti i pohraniti u memoriju ili se može stvarati izravno iz memorije prema razini detalja po potrebi u vrijeme izvođenja programa. *Slika 22* prikazuje kako izgleda isječak 5x5 i njegov sljedbenik u lancu. Razina 0 je najdetaljnija razina, svaka slijedeća razina(1, 2, 3, ...) je manje detaljna.

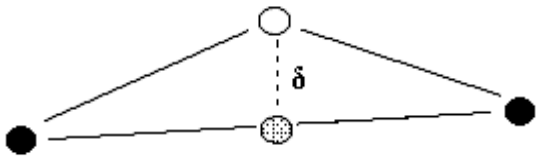


*Slika 22: Aktivirane točke u mreži 5x5 vrhova za geometrijski isječak razine 1*

### 4.2.1 Odabir razine geometrijskog isječka terena

Označimo s *d* udaljenost odsječka od kamere. Ako se koriste fiksirane vrijednosti *d* za svaku razinu detalja, događaju se neželjena iskakanja vrhova prilikom prelaska s jedne na drugu razinu. Svaka promjena razine naviše unosi pogrešku u teren, a ona ovisi o konfiguraciji terena.

Potrebno je definirati $\delta$ kao promjenu visine prilikom uklanjanja nekih od vrhova. $\delta$ je usko vezan uz *d*, jer, što je *d* veći to je $\delta$ više zanemariv. Uvođenjem $\varepsilon$ kao projekcije $\delta$ iz prostornih u koordinate zaslona za svaki vrh koji dodajemo/uklanjamo dobivamo bolju mjeru o iskakanju. Ako imamo $\varepsilon$, imamo informaciju što korisnik vidi i definiranjem da $\varepsilon$ ne smije biti veći od npr. nekoliko piksela ograničavamo efekt iskakanja.




*Slika 23: Uklanjanjem bijelog vrha pri prelasku na višu razinu(manje detalja), stvara se pogreška iznosa δ između bijele točke i njezine projekcije*

Postupak pronalaženja $\varepsilon$ počinje pronalaženjem $\delta_M = \max\{\delta_0, \ldots, \delta_{n-1}\}$, gdje je $n$ broj uklonjenih vrhova. Dobiveni $\delta_M$ se pretvara u $\varepsilon$, nakon čega uspoređujemo $\varepsilon$ s $\tau$, $\tau$ predstavlja prag pogreške u pikselima zaslona. Time je pokriven najgori slučaj za pločicu. Nadalje, ako je $\varepsilon < \tau$ znači da je pogreška unutar granica prihvatljivoga, te je prijelaz na višu razinu prihvatljiv.

Ovaj je postupak intenzivan za procesor. Zanemarivanjem visine kamere (Y-osi) moguće je unaprijed izračunati sve $\varepsilon$. Osim te optimizacije, postoji mogućnost definiranja minimalne udaljenosti $D_n$ na kojoj se razina $n$ može koristiti.

Usporedbom $L$ (udaljenost kamere od centra pločice) i $D_n$ za svaki $n$ iz razina možemo odrediti je li ta razina zadovoljavajuća. $D_n$ se računa prema *formuli 3*.

$$D_n = |\delta| * C \,,\; C = \frac{A}{T} \,,\; A = \frac{n}{|t|} \,,\; T = \frac{2 * \tau}{v_{res}}$$

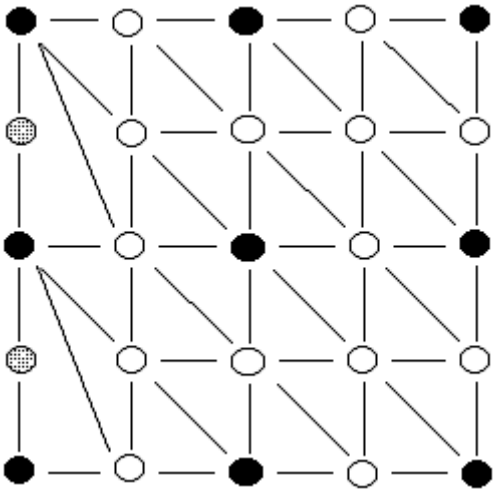
*Formula 3: Skup formula za izračunavanje Dn;* δ je pogreška (Ilustracija 23), n je bliža izrezujuća ploha, t je gornja koordinata od n, $v_{res}$ je vertikalna rezolucija u pikselima, τ je prag pogreške

Na nivou pločica sve je definirano. Ako promatramo cjelokupni teren iscrtan tako generiranim pločicama može se primijetiti da se stvaraju pukotine između pločica različitih razina. Važno je imati na umu da nikad dvije susjedne razine ne smiju biti po detaljima udaljeni više od jedne razine.

Jedno od mogućih rješenja ovog problema jest uključivanje dodatnih vrhova na rub pločice niže razine detalja. To se mora obavljati dinamički ovisno o susjedima sa sve četiri strane od pločice. Takvo rješenje nije baš dobro jer je zahtjevno i sa stajališta procesora i memorije.

Još jedno rješenje bilo bi poravnanje točaka pločice s više detalja sa srednjom vrijednošću dvaju susjednih vrhova, što opet nije dobro jer se zbog decimalne preciznosti ipak javljaju 'rupe' na mjestima spoja.

Autor ove metode iscrtavanja kao najbolje rješenje nudi promjenu načina povezivanja. Ideja je da se na pločici više razine detalja jednostavno izostave vrhovi koji uzrokuju pukotine. To iziskuje promjenu u povezivanju vrhova. Na *slici 24* se može vidjeti jedno rješenje promjene povezivanja.



*Slika 24: Povezivanje pločica različitih razina promjenom indeksiranja vrhova; gornji lijevi crni vrh se povezuje sa susjednim bijelim i crnim ispod, za koji vrijedi isto*

# 5. Programsko rješenje

Dosad su spomenute neke mogućnosti rješenja danog problema. Postoji još mnogo rješenja, no uglavnom su bazirana na spomenutima. Sada bi bilo dobro još se jednom osvrnuti na glavne zahtjeve dane na izradu ovog rada:

- generiranje terena fraktalnom tehnikom
- prikazivanje velikih terena
- preslikavanje tekstura na teren na temelju konfiguracije terena
- koristiti OpenGL i C++

## *5.1. Analiza zahtjeva i izrada specifikacije*

Zahtjev na generiranje terena fraktalnom metodom ne ostavlja puno prostora za odabir najboljeg tehničkog rješenje. Odabrano rješenje koje će se koristiti u ovom radu je generiranje terena metodom *dijamant-kvadrat.*

Teško se bilo odlučiti za odabir algoritma koji bi najbolje riješio problem prikazivanja velikih terena. Nakon ispitivanja nekoliko već gotovih programskih rješenja besplatno dostupnih na internetu, odluka je pala na ROAM implementaciju Bryana Turnera[15]. Obrazloženje ove odluke leži u svojstvu ROAM algoritma da se ograniči broj trokuta dodijeljenih za prikazivanje. To je odlično svojstvo, jer uvijek imamo kontrolu nad tom varijablom. Ako bi koristili geometrijske isječke za prikazivanje terena, uzmimo da je *r* radijus oko kamere unutar kojeg se nalaze svi geometrijski isječci razine detalja veće od najmanje razine detalja. Za prostor unutar te kružnice znamo broj poligona, no izvan tog dijela broj poligona se linearno povećava ovisno o veličini terena. Drugi razlog vezan je uz dojam koji ostavlja pojedino rješenje. Razlika između ROAM-a i korištenja geometrijskih isječaka je u tome što geometrijski isječci ne dodaju detalje tamo gdje je potrebno više detalja, već su fokusirani isključivo na pojednostavljivanje promatrane pločice na što manje intenzivan način za CPU. ROAM s druge strane nastoji dinamički odrediti gdje iz korisničke perspektive treba više detalja i tamo preraspodjeljuje više trokuta.

Turnerov ROAM koristi podjelu na pločice slično kao i kod iscrtavanja terena geometrijskim isječcima. Time dobivamo najbolje i od te metode, a to je da se može obavljati izrezivanje nevidljivih pločica korištenjem AABB obujmica. Dva reda iz originalne specifikacije su zamijenjeni jednim redom i to za dijeljenje. Ako se pogleda pseudokod reda za spajanje ROAM specifikacije, može se primijetiti da je dosta kompleksiji od reda za dijeljenje, pa je takva modifikacija potpuno smislena. Međutim, time se gubi koherencija između okvira. Jedina mana Turnerove implementacije ROAM algoritma je to što je predviđen za kameru koja lebdi iznad terena, a ne za pogled s visine, što se lagano korigira.

Preslikavanje tekstura na teren na temelju konfiguracije terena je prilično nezgodno područje gdje je sve stvar kompromisa. Koji god način teksturiranja odabrali, nešto dobivamo, a nešto gubimo. Općenito, zbog korištenja bilo kojeg LOD algoritma, nije preporučljivo koristiti 3D teksture, jer su teksture tada vezane za vrhove, a vidljivost vrhova je upitna. Preostaje proceduralno generiranje vrhova ili koristiti miješanje tekstura. Miješanje tekstura u svakom slučaju daje bolje rezultate, ali, pošto se radi o velikim terenima, nije svejedno koliko prolaza teksturiranja je potrebno i koliko jedinica za teksturiranje je potrebno. Isključivo zbog pitanja brzine koristi se proceduralno generiranje tekstura. Karakteristike ovog teksturiranja su da je unaprijed generirana, zauzima relativno malo prostora, svaki vrh ima dodijeljen točno jedan piksel teksture, lagano je moguće pomiješati ovu teksturu s mapom osvjetljenja jednostavnom operacijom množenja. Rezultat ove operacije je teren prilično lijepo teksturiran ako ga promatramo izdaleka, iako i to ovisi o odabiru baznih tekstura za proceduralno generiranje.

C++ je objektni jezik i zahtjeva objektno orijentiran pristup organizaciji koda. Kao pripomoć u stvaranju objektno orijentiranog programskog rješenja poslužio je alat BOUML, odličan besplatni program za izradu UML dijagrama. Korišten je samo dijagram klasa, da bi se okvirno konstruirala funkcionalnost i međuoperabilnost objekata. Više informacija o BOUML-u se može pronaći na stranici http://bouml.free.fr/. UML dijagram nalazi se u prilogu A.

Program bi trebao raditi s OpenGL verzijom 1.5 i više. Poveznica između sustava prozora i OpenGL konteksta bio je SDL. SDL je portabilna multimedijska biblioteka koja nudi pristup audio, video i drugim ulazno-izlaznim jedinicama na niskoj razini.

Code::Blocks je besplatno integrirano razvojno sučelje (*engl. Integrated Development*

*Environment*) koje se može pronaći na www.codeblocks.org. Code::Blocks je poslužio za sam razvoj programa.

## 5.2 Detalji implementacije i rad programa

Prilikom implementacije svih elemenata navedenih u specifikaciji bilo je potrebno napraviti neke promjene i implementirati dodatne stvari da bi sve zajedno dobro radilo. Na kraju procesa stvaranja terena dodan je filtar čija je uloga smanjivanje naglih visinskih skokova. Filtar zamućivanja (*engl. blur filter*) bio je očit izbor, jer se izvor informacija o visini može predstaviti kao slika, a ovaj filtar upravo je namijenjen za operacije nad slikama. Nažalost, problem je u tome što se u originalnom algoritmu koristi samo 256 razina diskretizacije da bi se sačuvala memorija i da bi operacije bile brze. Za posljedicu imamo primjetne diskretne skokove na visinskoj mapi koji se ne mogu filtrirati na ovaj način. Rješenje bi bilo pretvaranje cijele visinske mape u decimalne vrijednosti.

Prilikom generiranja proceduralne teksture dodano je jednostavno sjenčanje, koje pretpostavlja da je izvor svjetla iznad jednog kuta terena, a intenzitet sjene ovisi o nagibu. Rezultat sjenčanja se upisuje izravno u teksturu i tako daje osjećaj dubine terena.

Najvažniji dodatak u implementaciji bilo je izrezivanje nevidljivih pločica. Za to je korišten sustav AABB obujmica i detekcije vidljivosti obujmica pomoću stabla s četvero djece. Stablo je izgrađeno prilikom stvaranja pločica, a u svakom okviru se prolazilo kroz cijelo stablo i provjeravao se presjek vidnog polja s obujmicom. Ako je obujmica unutar, išlo se na slijedeću razinu pod uvjetom da ona postoji, ako ne postoji, znači da smo na pločici, koju aktiviramo da je vidljiva.

Procedura rada programa izgleda nekako ovako:

- odabir parametara postavljanjem sklopki prilikom pokretanja programa
- učitavanje visinske mape i odgovarajuće teksture iz datoteke ili stvaranje visinske mape i vlastite teksture sa sjenama
- stvaranje pločica od visinske mape, svakoj se pločici dodjeljuje memorijski prostor s njezinim dijelom visinske mape
- računanje pogrešaka u binarnom stablu trokuta i postavljanje veza između susjednih trokuta i dijamanata

- prelazak u način rada u stvarnom vremenu
- za svaki okvir se u ROAM algoritmu ponovno postavljaju veze između trokuta na početne vrijednosti, prolazi se kroz stablo i računaju pogreške, izrezuju nevidljive površine i sve vidljive iscrtava

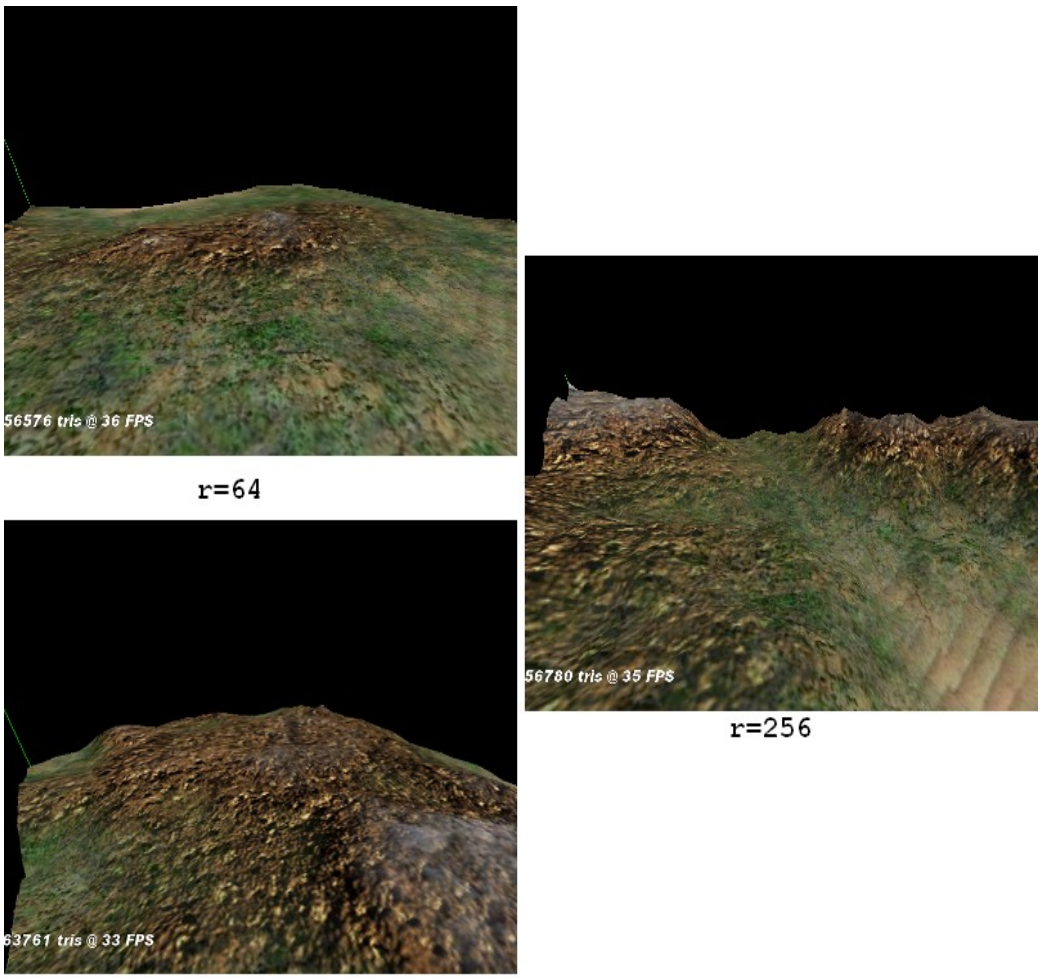
## *5.3 Analiza algoritama*

Analiza će biti podijeljena na dva dijela; u prvom će se analizirati fraktalni generator terena, a u drugom će biti analiziran ROAM algoritam. Promatrati će se neke od karakteristika zanimljivih za analizu iz perspektive zahtjeva na rad.

### 5.3.1 Analiza fraktalnog generatora

Kod fraktalnog generatora koristi se samo jedna varijabla za kontrolu kakav će se rezultat ovim algoritmom dobiti, a to je grubost (*engl. roughness*). Promjenom te varijable određuje se najveće odstupanje od početne visine. Grubost se monotono smanjuje pri svakoj rekurzivnoj propagaciji.

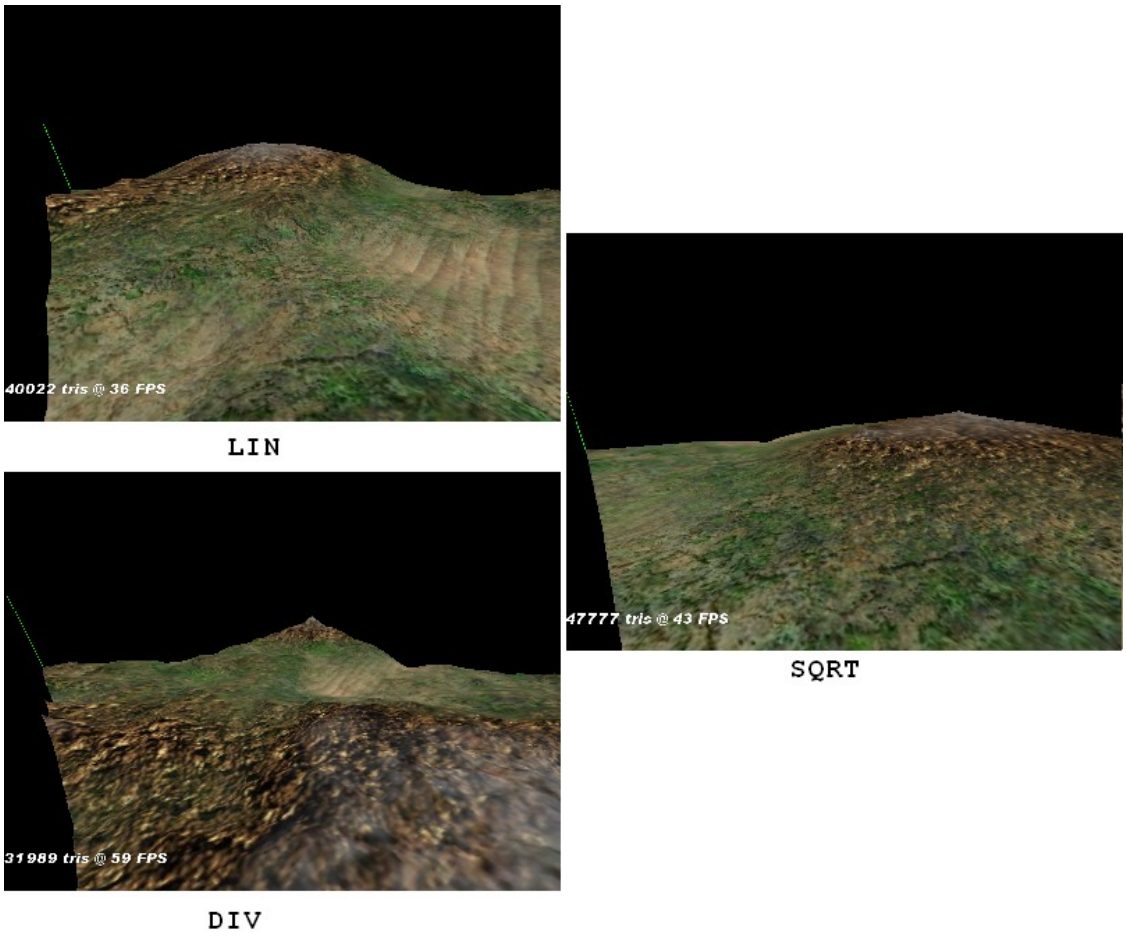
Mijenjanjem grubosti mogu se dobiti različiti tipovi terena koji se mogu vidjeti na *slici 25*. Očito je da za najveći *r*, teren je najgrublji, odnosno visinske razlike su najveće.



*Slika 25: Primjeri dobivenih terena za grubost r*

Zanimljivije je uočiti da nije samo bitna početna vrijednost grubosti terena. Jako je bitno na koji način opada ta vrijednost. Naime, nije svejedno je li to linearno, dijeli li se u svakom koraku, traži li se korjen u svakom koraku ili neka druga operacija. Stoga je implementirana i mogućost promjene tih parametara, a rezultati se mogu vidjeti na *slici 26*.

Linearno opadanje grubosti je ponešto sporo, no izgleda prilično prirodno. Opadanje dijeljenjem s 2, također u svakom slučaju izgleda vrlo prirodno. Korištenjem korjena kao funkcije opadanja grubosti terena jako se brzo guše visoke frekvencije u terenu, tako da nam ostaje utjecaj samo prvih nekoliko koraka algoritma.




*Slika 26: Primjeri terena za linearno smanjenje grubosti(LIN), smanjenje dijeljenjem(DIV), te smanjenje pomoću korjena(SQRT)*

Ovo su korištene operacije kao zakon opadanja za pojedini tip

```
case LIN:
roughness-=40;
case DIV:
roughness>>=1;
case  SQRT:
roughness=sqrt(roughness);
```

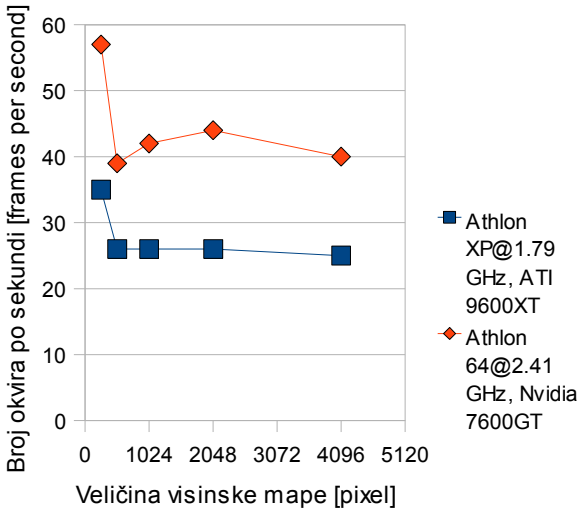
, gdje je *roughness* vrijednost grubosti terena za tu razinu

Jednostavnom analizom se dolazi do spoznaje da za linearno opadanje grubosti, za dubinu od 7 grubost iznosi 0. Kod opadanja dijeljenjem za to je potrebna dubina od 8, a za opadanje pomoću korjena, zbog zaokruživanja brojeva, na 1 se pada već nakon četvrte razine. To znači da se sve niže razine od ovdje navedenih mogu smatrati zanemarenima za određeni slučaj.

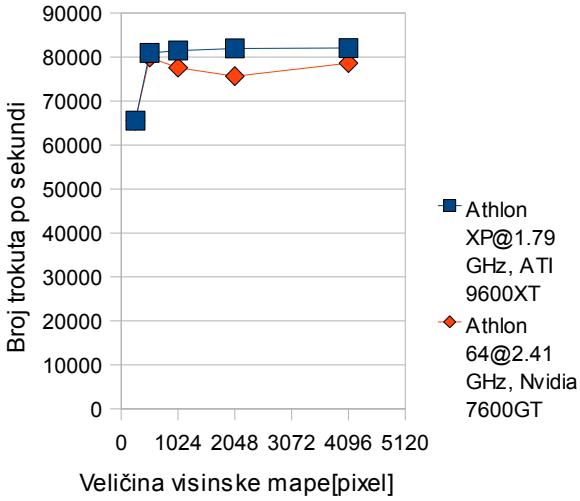
Zaključak je da odabiranje zakona opadanja grubosti terena daje još jednu dimenziju fraktalnom generatoru terena.

### 5.3.1 Analiza ROAM algoritma

Kod ROAM algoritma najzanimljivije je za promatrati ovisnost broja okvira po sekundi (*engl. frames per second, FPS*) o veličini visinske mape.




*Slika 27: Ovisnost broja okvira po sekundi o veličini visinske mape*



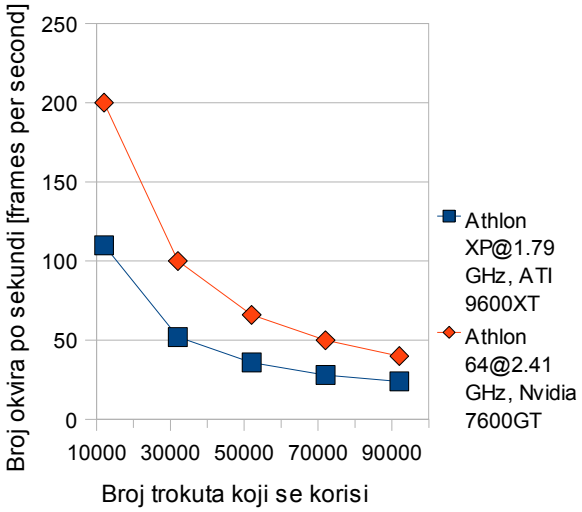

*Slika 28: Ovisnost broja trokuta o veličini visinske mape*

*Slika 27 i slika 28* se moraju gledati zajedno, inače je moguće doći do pogrešnih zaključaka. Dakle, broj okvira po sekundi usko je vezan s brojem trokuta po sekundi. Broj trokuta koji se koristi je postavljen na 80 000. Za vrijeme rada programa, taj se broj smije prijeći, ali će se povratnom vezom pokušati vratiti na vrijednost od oko 80 000.

Veličina terena ne utječe na broj sličica u sekundi, osim ako je veličina terena previše malena za zadani broj trokuta, kao u slučaju visinske mape 256x256. Takav slučaj je besmislen, jer je to ekvivalentno iscrtavanju bez ikakvog algoritma za upravljanjem detaljima.

Pomoću ova dva grafa dokazano je da je ROAM algoritam pogodan za prikazivanje velikih terena po pitanju brzine, jer veličina visinske mape ne utječe na rad algoritma.

Slijedeća korisna informacija je ovisnost broja okvira po sekundi o broju trokuta koji se koristi. *Slika 29* nam daje tu informaciju.




*Slika 29: Ovisnost broja okvira o broju korištenih trokuta prilikom iscrtavanja*

Vidimo da broj okvira po sekundi brzo konvergira prema nuli, što je i očekivano, jer svaki dodatni trokut predstavlja dodatno opterećenje i glavnom procesoru koji mora analizirati pogrešku za trokut i odlučiti treba li ga crtati i za grafičku karticu koja taj trokut mora iscrtati.

Algoritam je pogodan za dodavanje sustava za određivanje vidljivog volumena. Nakon implementacije tog sustava je ustanovljeno da on stavlja preveliki pritisak na glavni procesor i da performanse padaju. Iz toga razloga je on isključen predprocesorskom naredbom. Informacija koju možemo izvući iz ovoga je da je glavni procesor usko grlo ovog algoritma, usprkos korištenju cjelobrojnih varijabli da bi pojednostavili operacije. Ovaj rezultat se u zadnje vrijeme često javlja, a uzrok tomu je nesrazmjeran razvoj grafičkih procesora i klasičnih procesora. Današnji grafički procesori su veoma brzi i optimiziran, u stanju isporučiti velik broj obrađenih poligona u sekundi. Ovaj je algoritam je iz doba dok to nije bilo tako. Tada se na glavnom procesoru radilo sve što je bilo moguće da bismo rasteretili grafičku karticu, dok je danas to često suvišno.

# 6. Zaključak

Uzevši u obzir zadane zahtjeve i programsko rješenje ovog rada, može se zaključiti da su sve bitne komponente zahtjeva ispunjene. Ustanovljeno je da je glavni procesor usko grlo ROAM algoritma, što je bilo za očekivati. Pojavio se problem korištenja sustava za izrezivanje pločica izvan vidnog polja. Unatoč tomu, razina detalja visinske mape je prilično dobra čak i za dimenzije visinske mape 4096x4096 slikovnih elemenata.

Fraktalni generator terena jednostavan je i iznimno brz model za rješavanje problema stvaranje visinske mape. Pokazalo se da promjenom parametara utječemo na stvoreni teren na razne načine.

Korišteni način teksturiranja ne može se smatrati najboljim, no kao ilustrativan primjer je dobra ideja koja se još može razviti. Prilikom razvoja programskog rješenja nije bio naglasak na detaljnim teksturiranjem, iz razloga što nije postojao zahtjev na to, a zahtjev na teksturiranje na temelju konfiguracije terena je uspješno ispunjen. Nedostatak detalja dodaje opterećenje i stoga je problematičan, a može se riješiti na nekoliko predloženih načina u poglavlju o teksturiranju.

Turnerov ROAM algoritam se pokazao vrlo dobar za prikazivanje terena velikih dimenzija. Fleksibilnost, proširivost i velik prostor za eksperimentiranje s ovim algoritmom karakteriziraju ga kao potencijalni prostor za daljnji razvoj metoda za prikazivanje terena. Dokaz tome leži u činjenici da se aktivno radi na razvoju ROAM 2.0 algoritma za koji najavljuju da će donijeti mnoga poboljšanja izvornika.

# 7. Popis slika

# 8. Popis kratica

AABB – *engl. Axis-Aligned Bounding Box* – obujmice u obliku kvadra poravnate s osima

API - *engl. Application Programming Interface* – programsko sučelje

LOD – *engl. Level Of Detail* – razina detalja

MIP – *lat. Multum In Parvo* – mnogo na jednom mjestu, u računalnoj grafici označava skup slika raznih razina detalja

RGB - *engl. Red Green Blue* – standard prikazivanja boja

# 9. Popis tablica

# 10. Literatura

[1]Paul Martz , Generating Random Fractal Terrain, http://www.gameprogrammer.com/fractal.html
[2]Miller, Gavin S. P., The Definition and Rendering of Terrain Maps. SIGGRAPH 1986 Conference Proceedings (Computer Graphics, Volume 20, Number 4, August 1986).
[3]António Ramires Fernandes, http://www.lighthouse3d.com/opengl/terrain/index.php3?fault

[4]Tobias Franke, Terrain Texture Generation, http://www.flipcode.com/archives/Terrain_Texture_Generation.shtml, travanj 2001.
[5]Procedural Texture Generation for Height Mapped Terrain - By Perspective, http://www.cprogramming.com/discussionarticles/texture_generation.html
[6]Doug Sheets, An OpenGL 3d Texturing Tutorial, http://gpwiki.org/index.php/OpenGL:Tutorials:3D_Textures, 9. studeni 2004.
[7]Charles Bloom, Terrain Texture Compositing by Blending in the Frame-Buffer, http://www.cbloom.com/3d/techdocs/splatting.txt, 2. studeni 2000.
[8]Tim Jenks , Terrain Texture Blending on a Programmable GPU,http://www.jenkz.org/articles/terraintexture.htm, rujan 2005
[9]Nate Glasser, Texture Splatting in Direct3D, http://www.gamedev.net/reference/articles/article2238.asp, 23. travnja 2005.
[10]Øivind Liland, Shadow, http://www.ia.hiof.no/~borres/cgraph/explain/shadow/p-shadow.html, 2002.
[11]Mircea Marghidanu, Fast Computation of Terrain Shadow Maps, http://www.gamedev.net/reference/articles/article1817.asp, 7. svibnja 2002.
[12]Faster Ray Traced Terrain Shadow Maps, http://gpwiki.org/index.php/Faster_Ray_Traced_Terrain_Shadow_Maps
[13]Duchaineau Mark, Wolinsky Murray, Sigeti David E., Miller Mark C., Ladrich Charles, Mineev-Weinstein Mark B., ROAMing Terrain: Real-time Optimally Adapting
Meshes
[14]De Boer Willem H., Fast Terrain Rendering Using Geometrical MipMapping, kolovoz 2000., http://www.connectii.net/emersion
[15]Bryan Turner,Real-Time Dynamic Level of Detail Terrain Rendering with ROAM, http://www.gamasutra.com/features/20000403/turner_01.htm, 3. Travnja 2000.

# 11. Sažetak

U ovom radu pokušano je obuhvatiti nekoliko načina generiranja i prikazivanja 3D terena. Od metoda generiranja visinskih mapa dojam najveće uporabivosti pokazala je fraktalna metoda generiranja terena zbog jednostavnosti, brzine i rezultata.

Teksturiranje velikih terena je problematično zbog količine podataka potrebnih za dobivanje detalja na terenu. Obično se koristi više manjih tekstura koje služe kao predlošci za miješanje u jednu teksturu za vrijeme izvođenja programa (miješanje tekstura) ili prije pokretanja programa (proceduralno generiranje tekstura). Sjene terenu daju dubinu i jako često se simuliraju lijepljenjem unaprijed izračunate mape sjene na teren. Volumne sjene se izbjegavaju kod prikazivanja terena.

Postoji mnogo ideja na koji način prikazati visinsku mapu u obliku 3D terena pomoću zadovoljavajuće razine detalja. Izdvojio bih ROAM i geometrijsko MIP preslikavanje kao dvije svojevrsne krajnosti u pristupima problemu. ROAM dinamički određuje razinu detalja potrebnu na određenom prostoru, dok se geometrijski MIP isječci mogu stvoriti unaprijed s manje proračuna, ali i lošijom distribucijom vrhova.

**Ključne riječi:** teren, fraktalna generacija, teksturiranje, sjene, miješanje tekstura, ROAM, geometrijsko MIP preslikavanje,C++, OpenGL, prikazivanje

# 12. Abstract

This paper has tried to enfold a few ways of generating and rendering 3D terrains. By analyzing methods for heightmap generation, fractal terrain generation method left best impression of usability, mainly because of its simplicity, generation speed and results.

Texturing of large terrains is problematic in terms of big data requirements needed for detailed terrain. Usual way of handling terrain texturing is to use a number of smaller textures as a template for blending into one texture either at runtime (texture splatting method) or at startup (procedural texture generation method). Shadows add depth to the terrain and they are often simulated as a precalculated shadow maps. Volume shadows are avoided when rendering terrain.

There are many ideas of rendering heightmaps at satisfactory level of detail. I would like to point out ROAM and Geometrical MipMapping as two kinds of extremes in approach to solving this problem. ROAM dynamically sets level of detail needed at certain area, while Geometrical MipMaps can be generated in advance with less calculations but not so good vertex distribution.

**Keywords:** terrain, fractal terrain generation, texturing, shadows, texture blending, ROAM, Geometrical MipMapping, C++, OpenGL, rendering

# 13. Privitak A – UML dijagram programa