

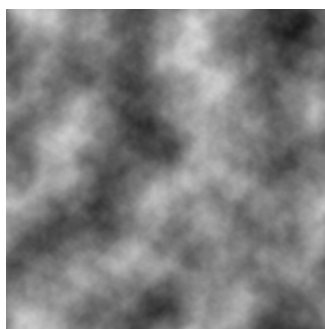
Proceduralno generisanje terena

Predlog projekta iz predmeta Numericki algoritami i numericki softver

Height Mape

Kod generacije terena pomocu Height Mape se uzimima da svaki piksel ima boju, ako se radi u RGB formatu za skorz beo piksel koji sadrzi mesavinu svih broja vrednost je jednaka $256 \times 256 \times 256$ sto je i maksimalna moguca vrednost, a skroz crni ima vrednost $0 \times 0 \times 0$ ovaj interval se moze iskoristi tako sto se za svaku tacku generisanog terena koja odgovara poziciji piksela slike dodeli vrednost visine tog piksela. Naravno ovaj interval se moze skalirati da nam maksimalna visine ne bude $256 \times 256 \times 256$. Skaliranje se obicno uvek vrsi na vrednosti od -1 do 1 koje se kasnije mnoze sa maksimalnom visinom. -1 jer se predvidja da postoji neka voda ili zamisljena granica koja ogranicava normalnu visinu, u radu ce to biti standardno 0 i na njoj ce biti voda.

Ovakva generacija terena preko slike nije uvek pogodna jer prelazi piksela nisu uvek blagi i ne prate ono sto bi se nazvalo prirodnim. Zato se prave specijalne slike koje se zovu HeightMape, a medju popularnijima je najcesca upotreba Perlin noisa za HeightMape.



Primer Perlin noise. Bitne cinjenice su da su prelazi izmedju najtamnijeg piksela i najsvetlijeg blagi, ali sama funkcija nije po sebi glatka jer sadrzi puno manjih variranja vrednosti, ovo je bitno jer daje na prirodnosti izgleda generisanog terena.

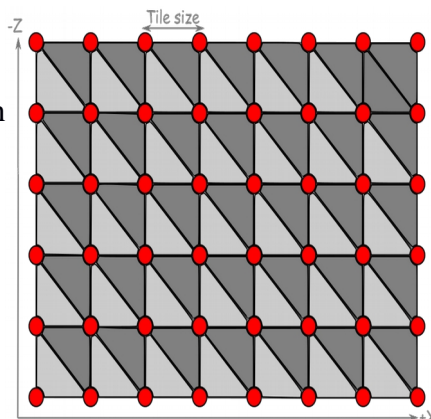
Celo nase bavljenje HeightMapama i Perlin noisom jeste zato sto cemo mi proceduralno pokusati da izmitiramo ovu funkciju. To moze da se postigne na par nacina, dalje je opisan korisceni nacin.

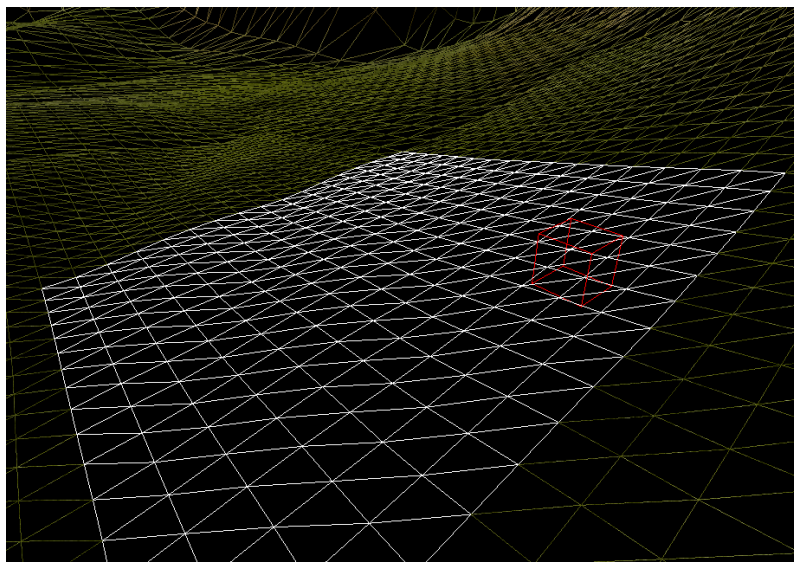
Proceduralna generacija

Pocetak naseg generisanja ce se zasnivati na nasumicnosti ili mozda bolje receno na pseudo nasumicnosti (pseudo jer cemo vrsiti vise korekcija na tome sto je nasumicno, a i nista nije stvarno nasumicno). Da bi objasnili pojam pseudo nasumicnosti prvo cemo se vratiti na neke osnove 3D prikaza i programiranja. Dakle mi imamo 3 ravni od kojih smo generisali nesto u ravnima X i Z ravanima, Y je visina i na njoj je 0. Principi generisanja se zasnivaju na trouglovima odnosno svako telo je generisano iz gomile malih trouglica.

Zasto trouglovi? Pa zato sto trouglove je nemoguće povezati drugacije, odnosno kad su tacke trougla povezane, povezane su na jedini moguci nacin. To nam je bitno kasnije za mapiranje istih tacaka jer se nase tacke preklapaju (u prevodu svaka dva susedna trougla imaju najmanje 2 iste tacke ovo kasnije ima uticaja i kod generacije visine).

Primer koordinatnog sistema predstavljaju gore receno u praksi. Ovakava mreza se generise bez vecih problema sa jendakim



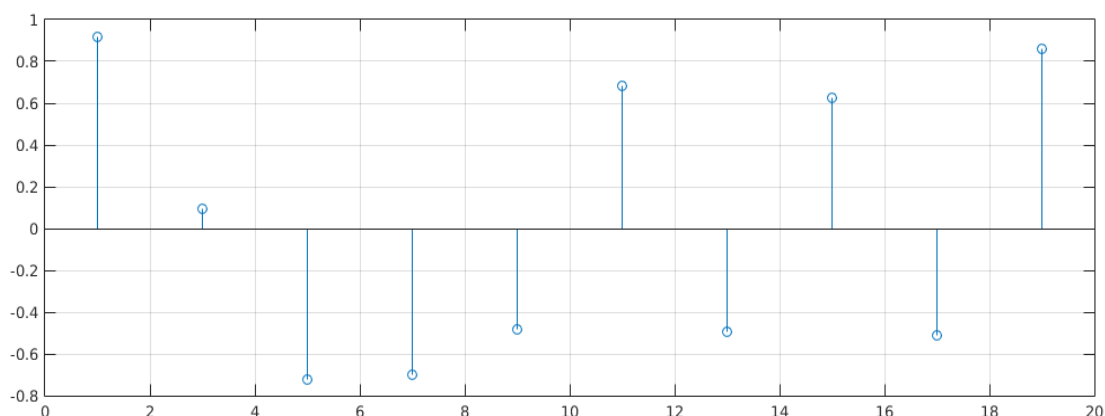


korakom za svaku tacku, ali tu nailazimo na prvu stvar zbog koje nije proces nasumican, svaki ovakam sistem se generise preko semena (seed), sto je jako bitno zbog optimizacije kasnije i renderovanja . Naime ako generisemo samo teren koji je u fokusu onda kad bi se vratili na pocetnu tacku bio bi izgenerisan neki skroz drugaciji teren sto nije ono sto zelimo. Te uvodimo u generisanje nasumicnih brojeva seme. Odnosno za dva ista broja nasumicni generator sa

istim semenom uvek vraca istu vrednost. Primer igre koja koristi ovo je minecraft. Znaci kada je prvi put nasumicni (random) generator naisao na broj 3 i recimo izgenerisao broj 51 svaki sledeci put kad dobije broj 3 on mora da generise broj 51. Druga stvar koja nam smanjuje nasumicnost jeste nase ogranicenje intervala u kojem se generisu nasumicni brojevi. A treca stvar je to sto kod nasumicnog generisanja javlja se isti problem da dve susedne kordinate (recimo (x,y) i $(x+1,y)$) mogu da imaju ogroman skok (npr. -97, i 100) sto nije prirodno, te da bih to smanjio koristim u kodu osluskivanje okoline (ili ti neki filter) odnosno tacke okoline imaju uticaj na vrednost kordinate visine. Npr. uzimamo da samo okolne tacke uticu na dobijanje visi nase tacke tako da imamo 8 tacaka koje okružuju nasu. Tacke u coskovima imaju recimo uticaj $1/16$, tacke iznad, levo, desno i ispod $1/8$, a nasa nasumicno generisana tacka (centralna) utice sa $1/4$. Ako se saberu ove verovatnoce zbir tih uticaja bi trebao da bude 1 ukupno.

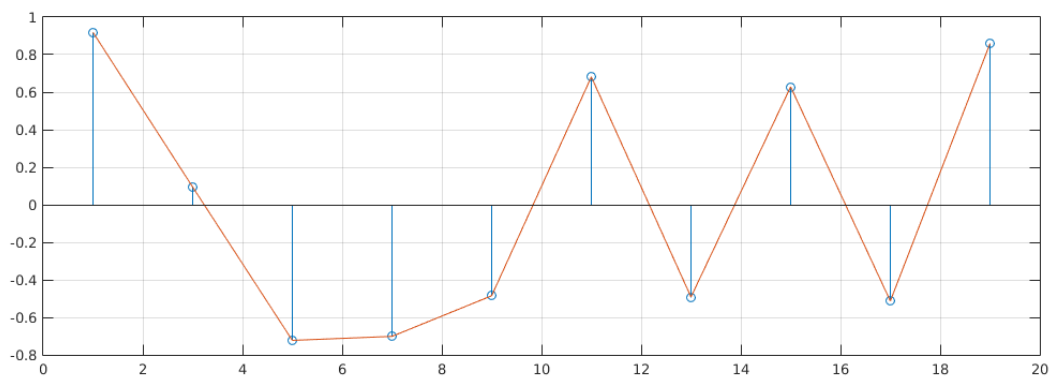
Princip nasumicne generacije prikazani u 2D-u

Prvo se generisu nasuminci projevi od -1 do 1 za svaku tacku.

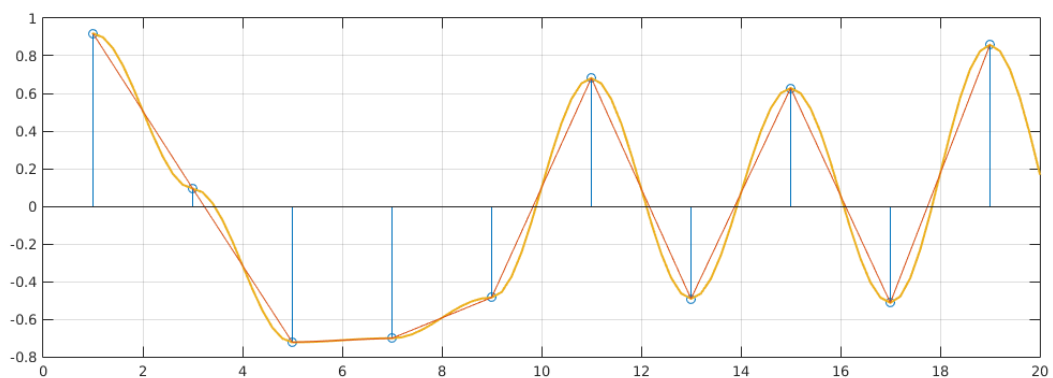
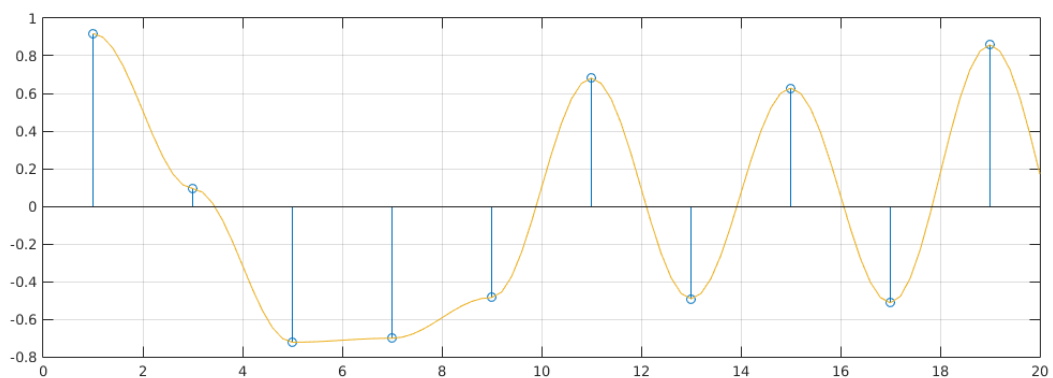


U ovom primeru nije koriscenjo osluskivanje okoline, ali u kodu ce biti.

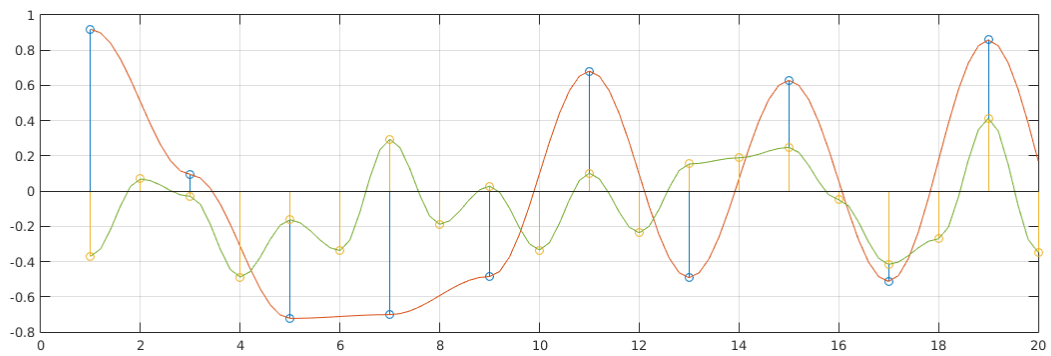
Sledi korak interpolacije. Ovaj korak se moze vrsiti na milion nacina. ispod je dat linearna.



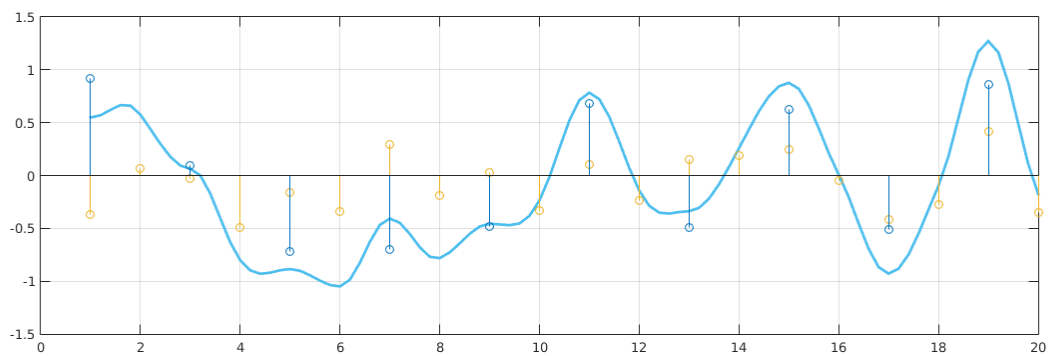
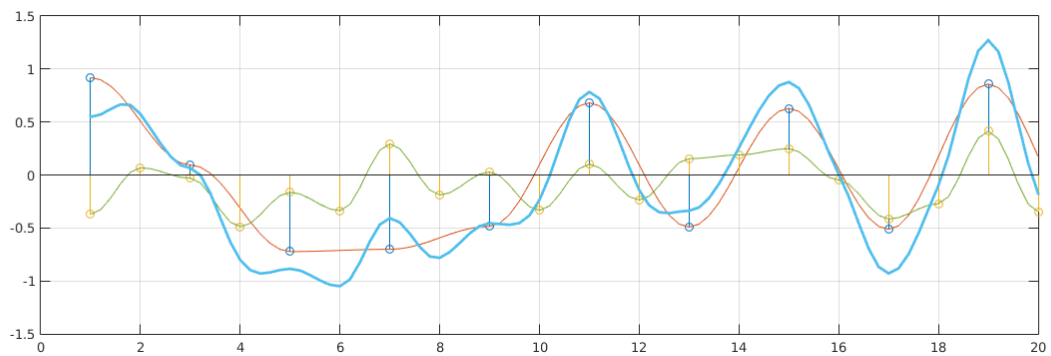
Kao što se da primetiti ovo nije baš prirodno te ćemo pokušati npr. sa kosinusnom interpolacijom gde dobijamo i krivine. Sve interpolacije su kroz 2 tačke, moguće je implementirati i kroz više tačaka.



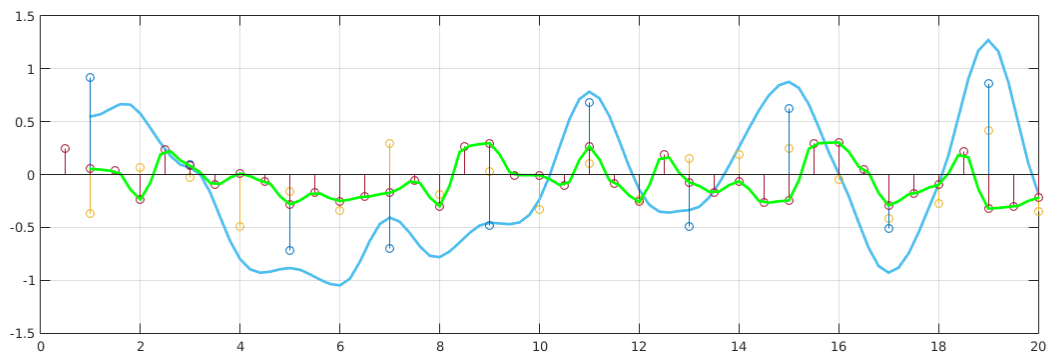
iznad data je bolja kosinusna interpolacija i uporedo linearnom interpolacija. No idalje ovo nije baš prirodno te se pribegava sabiranju funkcija, da bi se dobile sitne promene funkcije koje su prirodne. To se vrši tako što se frekvencija funkcije poveća ali se njena amplituda smanji.

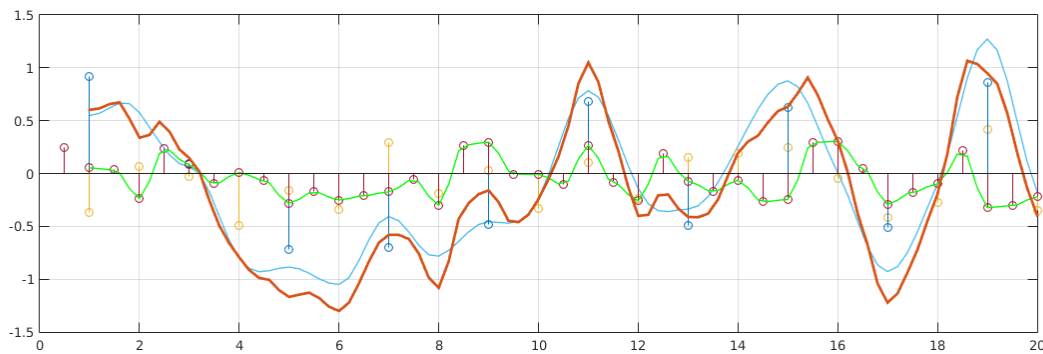


Gore je prikaz dotate funkcija sa duplo vecom frekvencijom i duplo manjom amplitudom. Ovo inace treba da su iste funkcije (na slici nisu). Dalje se saberu te dve funkcije i dobija se:



Vec se dobija malo realnija slika terena sada ovaj postupak se ponavlja dok nismo zadovoljni.





Uzeti u obzir da ovde nema osluskivanja okoliše i da nije korišćena ista funkcija, no to je princip nasumičnog generisanja (Perlin noise) suma i proceduralnog terena

Bitno za izvući odatle jeste to da povećanjem amplitude dobijamo detaljnost funkcije, ali ne želimo da nam ta detaljnost kvari naš prvobitni oblik, te smanjujemo njen uticaj na globalni oblik tako što joj smanjujemo amplitudu. Ove korake možemo ponavljati više puta.

Sada za razne načine interpolacije konsultovati sledeći sajt odakle su uzete neke varijante interpolacije između 2 tačke: <https://codeplea.com/simple-interpolation>

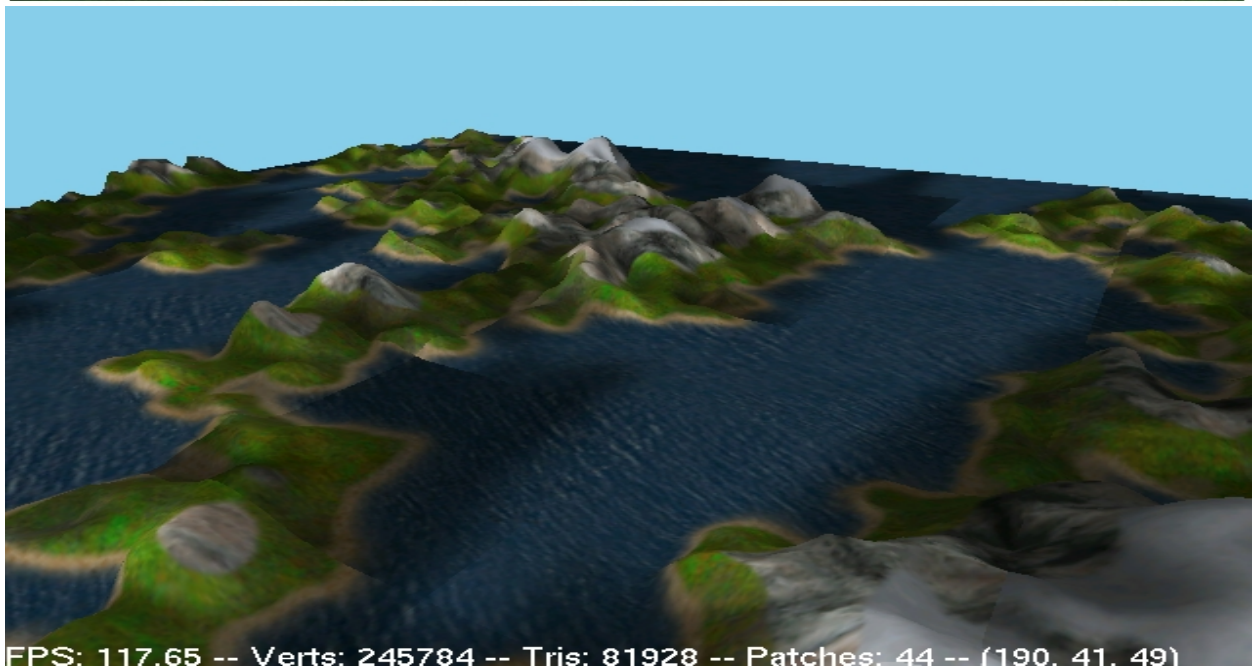
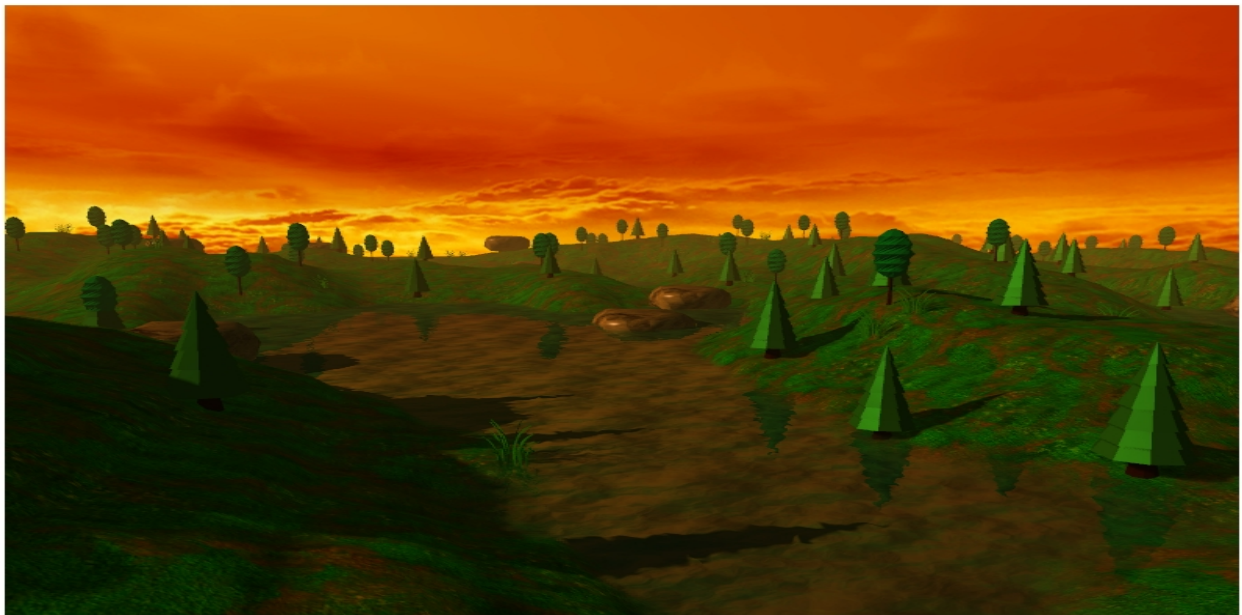
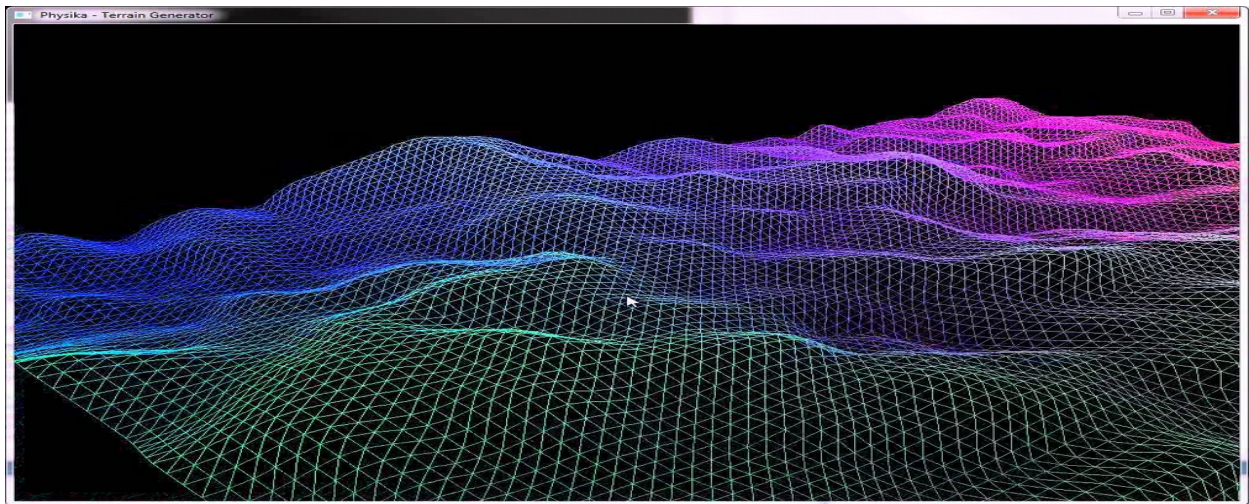
Specifikacija alata

Kod će biti u jeziku Java (Java) sa korišćenjem biblioteka LWJGL (Light Weight Java Game Library) koja koristi OpenGL API. Korišćeni se LWJGL 2.9.1 iako postoji novija verzija, zato što ima najviše podrške na netu i usput će se konsultovati tutorijali pravljenja game engine od scratch-a jednog programera. U kodu su implementirani i neke njegove metode za shadere (Vertex shader i Fragment shader - koji umnogome upotpunjuju izgled generisanog terena), takođe plan je da se ubaci i voda (sa njenim osobinama refrakcije i refleksije ali tema ovog rada neće biti kako funkcioniše ta voda i neće biti objašnjeno). Grafika će se bazirati na njegovom kodu. On se takođe bavio malo i sa generacijom terena. Sav deo oko generacije terena biće obradjen u kodu i podrazumeva se u odbrani projekta. Link njegovog kanala:

https://www.youtube.com/channel/UCUkRj4qoT1bsWpE_C8lZY0Q

Takođe pored LWJGL biblioteka koristi se i Slick-Utils biblioteku kao i PNG-Decoder koji je potreban za implementaciju SkyBoxa.

Sto se tiče menija koji je zadat u specifikaciji, napraviće se napor da bude pokriveno što više parametara koji su bitni: tipa izbor interpolacione metode (biće implementirana step, linear, cosine, smooth step...), izbor načina generisanja terena (preko HeightMapa ili proceduralno), izbor broja funkcija koje će se sabirati i parametar povećanja frekvencije i parametar smanjenja amplitude no ne garantuje se da će to biti vidljivo u real time-u. Shaderi su kompleksna stvar uopšte za samu generaciju terena, veliki broj proračuna tačaka terena, vode, objekata, gde za svaki objekat mora da se uzmu pozicija izvora svetlosti i normale da bi se pravile senke, refleksije, refrakcije i uopšte pozadinske interpolacije za boje i teksture koje vrsi GPU nisu nimalno naivne, zbog kojih možda neće biti moguće vrsiti promene u real time-u, tu se još dodaje ugao kamere sa kojim kamera gleda na nešto. Primer očekivanog rezultata:



FPS: 117.65 -- Verts: 245784 -- Tris: 81928 -- Patches: 44 -- (190, 41, 49)