

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

David Fabris

Fraktalni modeli terena i vegetacije u
računalnoj grafici
DIPLOMSKI RAD

Varaždin, 2013.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

David Fabris

Matični broj: 39595/10–R

Studij: Informacijsko i programsko inženjerstvo

**Fraktalni modeli terena i vegetacije u
računalnoj grafici**

DIPLOMSKI RAD

Mentor:

Doc.dr.sc. Ivan Hip

Varaždin, rujan 2013.

Sadržaj

1. Uvod	1
2. Što su fraktali.....	2
2.1. Definicija	3
2.2. Svojstva fraktala	4
2.3. Podjela fraktala	7
2.4. Fraktali oko nas	9
3. Osnovne vrste fraktala	11
3.1. Poznati fraktali.....	20
4. WebGL	27
4.1. WebGL u praksi	28
4.2. Three.js	32
5. Fraktalni teren i vegetacija	34
5.1. Teren.....	35
5.1.1. Perlinov šum.....	35
5.1.2. Implementacija	37
5.2. Obalna linija	44
5.3. Drvo	48
5.3.1. Implementacija	48
5.3.2. Optimizacija	53
5.4. Paprat.....	56
6. Simulacija leta nad fraktalnim terenom, posađenim fraktalnom vegetacijom	60
6.1. Otok - teren i obala	60
6.2. Razmještanje vegetacije	63
6.3. Simulacija leta	66
7. Zaključak	68
8. Literatura	69

1. Uvod

Prirodu je nemoguće opisati koristeći tek standardne geometrijske likove poput kružnice, kvadrata ili trokuta. Koliko god kompleksnu kombinaciju tih likova napravili, planine nikada neće biti uvjerljivo opisane trokutima, obale krugovima, a biljke kombinacijom različitih geometrijskih likova. Dosadašnji pokušaji objašnjavanja svijeta oko nas svodili su se na krajnje pojednostavljivanje, na pokušaj pronalaska minimalne formule koja se može primijeniti na sve. Rezultat toga je i euklidska geometrija, koja itekako ima svoju primjenu, no u pokušaju opisivanja stvarnog svijeta ne može nam previše pomoći. Kako onda opisati prirodu?

Fraktali su u početku otkriveni kao intrigantne matematičke zagonetke, da bi kasnije bili prepoznati i u kontekstu pojave kaosa u dinamičkim sustavima. Konačno se shvatilo da upravo fraktalne strukture i oblici posjeduju slična svojstva kakva nalazimo u stvarnom svijetu. Koristeći matematičke formule ipak je moguće opisati složene oblike kakve nalazimo u prirodi, kao što su planine, oblaci, riječna korita, kristali, biljke i drugi. Pomoću fraktala njih je moguće uvjerljivo opisati!

Fraktali imaju svoju primjenu u znanosti pa i u umjetnosti, no nas zanima tek jedno područje, računalna grafika. Kada bi trebali nacrtati nekakvu planinu, mogli bi u nekom alatu za modeliranje iscrtati svaki rub, svako izbočenje, a to naravno iziskuje mnogo truda, strpljenja i vremena. Takav način rada je neefikasan, neprihvatljiv, a na kraju krajeva i nedovoljno dobar, jer uvećamo li prikaz prema jednom sitnom dijelu planine, on će biti prikazan kao nekakav n -terokut, bez svojih specifičnih utora, proreza, izbočina. Što ako bi mogli definirati osnovni oblik planine, a da se svi detalji koje je tako teško ručno iscrtati nekako sami naprave? Što ako bi mogli napraviti prikaz koji je neovisan o stupnju uvećanja, koji uvijek zadržava istu količinu detalja?

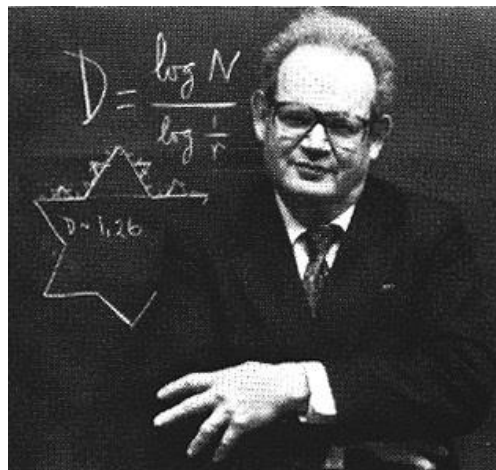
U ovom će radu biti opisani načini na koji se fraktali mogu iskoristiti u svrhu računalnog modeliranja stvarnog svijeta. Uz rad priložena je i računalna igra, odnosno simulacija letenja nad fraktalnim terenom i vegetacijom.

2. Što su fraktali

"Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line."

Benoit Mandelbrot

Početkom 20. stoljeća, Gaston Julia postavio je temelje za jednu novu granu geometrije, odnosno fraktale. Međutim, teorija je brzo zaboravljena, sve dok se njome opsežnije nije počeo baviti matematičar Benoit Mandelbrot, prikazan na slici 1.



Slika 1. Benoit Mandelbrot (Izvor: Nelson, 2010)

Upravo je Mandelbrot 1978. godine definirao naziv **fraktal**, riječ koja ima korijen u latinskoj riječi *fractus* (hrv. izlomljen). Detaljnija povijest fraktala je prikazana u tablici 1.

Tablica 1. Povijest fraktala (Šimac, 2007)

Vremenski okvir	Opis
1525.	• dokumentirani prikaz fraktala u priručniku za slikanje Albrechta Dürera
17. stoljeće	• Gottfried Wilhelm Leibniz definira ponavljanje samosličnosti
1872.	• Karl Weierstrass daje primjer apstraktne funkcije kojom se definira samosličnost
1904.	• Helge von Koch daje geometrijsku interpretaciju samoslične funkcije (Kochova pahuljica)
1915.	• Wacław Sierpiński kreira uzorak fraktala pomoću trokuta - <i>Sierpinskijev trokut</i> • Henri Poincaré, Felix Klein, Pierre Fatou, Gaston Julia i Georg Cantor daju doprinos kreiranjem fraktalnih skupova
1975.	• Benoit Mandelbrot definirao riječ fraktal i njezino značenje, a otkriće potkrijepio atraktivnim računalnim prikazima

2.1. Definicija

Benoit Mandelbrot je dao formalnu matematičku definiciju za fraktal, a prema njoj je fraktal skup za koji Hausdorff-Besicovitch dimenzija¹ strogo premašuje topološku dimenziju (Liming and Ting, s.a.). Detaljnije je objašnjena u poglavlju o svojstvima fraktala pod imenom fraktalna dimenzija.

Manje apstraktna definicija fraktala glasi: **fraktal** je geometrijski oblik koji se može podijeliti na beskonačno mnogo dijelova koji su na različitim skalama veličine sami sebi slični (Šimac, 2007).

Geometrijska konstrukcija fraktala

Geometrijski se fraktal sastoji od baze i motiva. I baza i motiv zapravo se sastoje od običnih linija. Motiv zamjenjuje bazu, a učestalim ponavljanjem tog postupka dobiva se fraktal. Na sljedećoj slici (slika 2) prikazan je primjer nastanka Kochove krivulje.



Slika 2. Primjer nastanka Kochove krivulje (Prema: e-škola, s.a.)

Baza je obična linija (_), dok je motiv takav da se središnji dio linije zamjeni sa trokutom bez donje stranice (_/_). U svakoj se iteraciji baza zamjenjuje motivom, čime nastaju nove umanjene verzije baze. Daljnjim ponavljanjem nove se baze zamjenjuju skaliranim motivima i tako u beskonačnost.

¹ Dimenzija u kojoj su dopušteni decimalni brojevi, a još je poznata i pod nazivom Hausdorff dimenzija i fraktalna dimenzija. Koncept je 1918. predstavio matematičar Felix Hausdorff, a Abram Samoilovitch Besicovitch je pružio veliki broj tehničkih poboljšanja za izračunavanje te dimenzije kod nepravilnih skupova.

2.2. Svojstva fraktala

Fraktali su geometrijski oblici poput onih koje poznajemo iz euklidske geometrije, no sa specifična tri svojstva koje ih razlikuju, a to su: **oblikovanje iteracijom**, **samosličnost** i **fraktalna dimenzija**.

Oblikovanje iteracijom

Svojstvo objekta da se generira nekim matematičkim ili geometrijskim postupkom koji se uzastopno ponavlja (Eklić, s.a.). Na taj način nastaju fraktali i većim brojem iteracija nastat će više detalja odnosno više umanjenih verzija cjeline. To možemo zaključiti i iz geometrijske strukture fraktala, objašnjene u prethodnom poglavlju.

Ako bismo gledali neku obalu iz svemira ona se sastoji od izbočina u more (rtova) i zaljeva. Uvećamo li prikaz prema jednom zaljevu, on se također sastoji od manjih verzija rtova, zaljeva, odnosno manjih uvala i slično. Koliko god "zumirali", efekt će uvijek biti isti, odnosno uvijek će postojati slična struktura manjih rtova i zaljeva.

Svojstvo samosličnosti

Govori da objekt sliči sam sebi bez obzira koji njegov dio promatrali i koliko ga puta uvećali. Uvijek ćemo dobiti sliku koja sliči početnoj (Eklić, s.a.).

Fraktalna dimenzija

Fraktalna dimenzija ili razlomljena dimenzija ne mora biti cijeli broj, kao što je to npr. euklidska dimenzija, a opisuje i neka svojstva objekta kao što su izlomljenost i hrapavost. Specifično za fraktalnu dimenziju je to što ona ostaje konstantna bez obzira na mjerilo (Eklić s.a.).

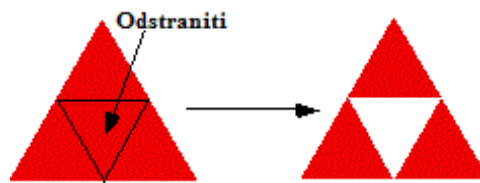
Mi vidimo svijet u 3 dimenzije. Svaku stvar moguće je opisati sa visinom, širinom i dubinom (euklidske dimenzije). Dvodimenzionalne su sjene ili crtež na papiru, makar crtež različitim tehnikama može davati privid trodimenzionalnosti. Jednodimenzionalni su pravci, a točke su objekti dimenzije nula. Međutim, dimenzija fraktala nije tako univerzalna. Fraktal koji se može prikazati u ravnini nije jednodimenzionalan kao pravac, niti dvodimenzionalan kao kvadrat, već je negdje između, koliko god to čudno zvučalo. Njegova dimenzija može biti npr. 1,23. Da se stvari koje mi vidimo ne mogu uvijek opisati striktno dvodimenzionalno ili trodimenzionalno pokazuje primjer britanskih znanstvenika koji su pokušavali izračunati duljinu britanske obale. Naime, ovisno o stupnju uvećanja, odnosno detaljima koji se uključuju u izračun, rezultat bi svaki puta bio drugačiji. Skaliranjem se gube ili dobivaju detalji koji značajno utječu na rezultat. Veća fraktalna dimenzija znači više detalja, a u primjeru obale, razvedeniju obalu.

Formula (dimenzija samosličnosti) za izračun fraktalne dimenzije je sljedeća:

$$d = \frac{\log(n)}{\log(s)}$$

gdje je "**d**" fraktalna dimenzija, "**n**" broj novih kopija odnosno samosličnih dijelova nakon skaliranja, "**s**" faktor skaliranja, a *log* prirodni algoritam.

Primjer izračuna je dan za Sierpinskijev trokut, za kojeg je prikazana njegova prva iteracija na slici 3.



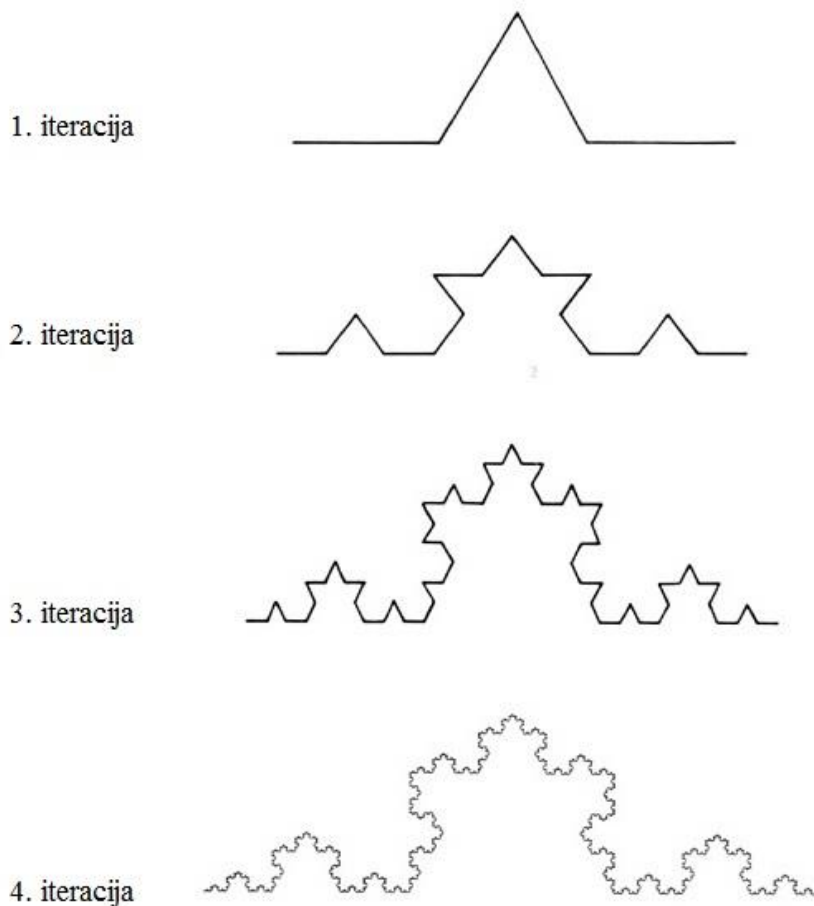
Slika 3. Kreiranje Sierpinskijevog trokuta (Prema: Riddle, 2013 c)

Kod Sierpinskijevog trokuta nastaju 3 nove kopije, stoga je "n" iz prethodne formule jednak 3. Promatramo li stranicu novonastale kopije vidimo da je ona duplo manja od originala, što znači da je "s" iz prethodne formule jednak 2.

Konačno:

$$\frac{\log(3)}{\log(2)} = 1.58$$

Kod Kochove krivulje na slici 4 svakom iteracijom nastaje 4 nova pravca. Svaki pravac je 3 puta manji.

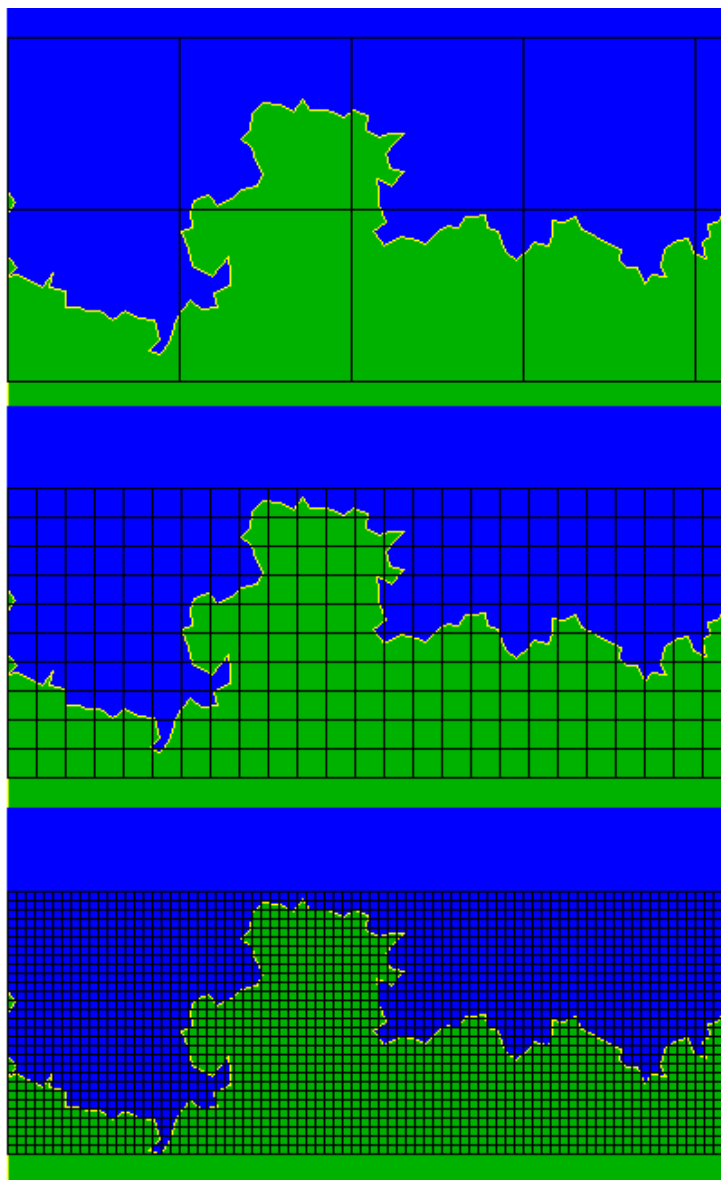


Slika 4. Kochova krivulja (Prema: Azí Arts, s.a.)

Fraktalna dimenzija Kochove krivulje je:

$$\frac{\log(4)}{\log(3)} = 1.26$$

Iako fraktalna tj. Hausdorffova dimenzija ima veliku teoretsku važnost, u praksi se slabo primjenjuje, a njoj sličan princip ima **Minkowski-Bouligand dimenzija**². Kod ove dimenzije fraktal se prekriva mrežom kvadratića (primjerice milimetarskim papirom) sve finije i finije razdiobe u svakom sljedećem koraku (prikazano na slici 5). Time dolazimo do jako jednostavnog algoritma pri kojem se prebrojavanjem kvadratića koji prekrivaju danu strukturu određuje dimenzija prema formuli sličnoj gore navedenoj za Hausdorffovu dimenziju. Ovako određena dimenzija ima samo aproksimativnu vrijednost (pri čemu je aproksimacija to bolja što upotrijebimo sitniju razdiobu) ali u praksi je to dovoljno (Blog Kaoslantida, 2007).



Slika 5. Minkowski-Bouligand dimenzija u primjeni (Izvor: Blog Kaoslantida, 2007)

² Poznata i pod imenom *Box-counting* dimenzija ili *Packing* dimenzija, u praksi najkorištenija formula za izračun fraktalne dimenzije.

2.3. Podjela fraktala

Fraktale možemo podijeliti na dva načina, koja su usko povezana s njihovim svojstvom samosličnosti i djelomično sa svojstvom oblikovanja iteracijom, odnosno prema:

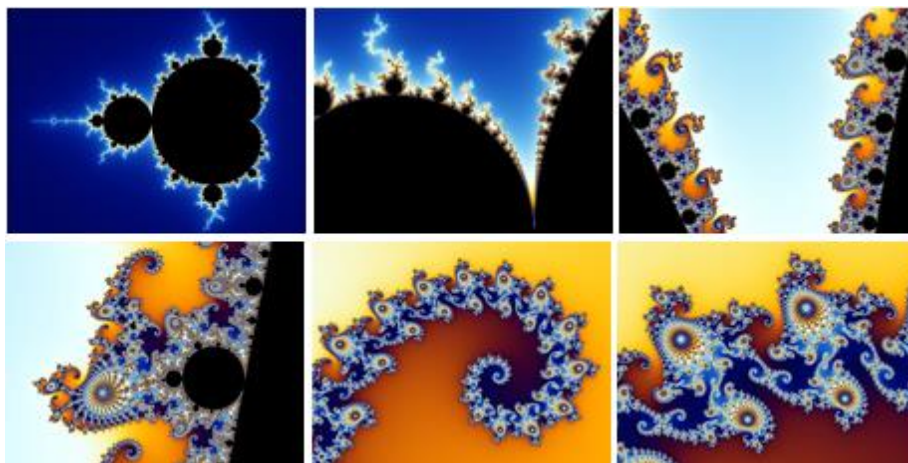
- a) **stupnju samosličnosti** i
- b) **načinu nastanka**.

Fraktali prema stupnju samosličnosti

Fraktali mogu biti **savršeno** ili **potpuno samoslični**, gdje je odmah vidljiva ta sličnost dijela i cjeline, kao što je primjer kod Sierpinskijevog trokuta i Kochove krivulje sa slike 3 i 4, te Hilbertove krivulje i Cantorovog skupa.

Ova vrsta fraktala nije korisna pri dizajniranju predmeta u svakodnevnom životu, pa se većinom koriste za kreiranje jednostavnih modela graničnih linija i obalnih linija, a nekada su se koristili za relativno dobre modele u ekonomiji (ahyco.uniri.hr, 2007 a).

Približno ili **kvazisamoslični** fraktali su fraktali gdje se na prvi pogled ne može uočiti sličnost. To su fraktali koji sadrže male kopije sebe koje nisu slične cijelom fraktalu, već se pojavljuju u iskrivljenom obliku (Brdar i sur., 2012).. Primjer takve samosličnosti je Juliaov skup i Mandelbrotov fraktal, koji je prikazan na slici 6.



Slika 6. Svojstvo samosličnosti na primjeru Mandelbrotovog fraktala (Izvor: Freiburger, 2009)

Statistički samosličan fraktal ne sadrži kopije samog sebe, ali neke njegove osobine (npr. fraktalna dimenzija) ostaju iste pri različitim mjerilima (Brdar i sur., 2012). Primjer takvog fraktala je Perlinov šum i plazma fraktal.

Statistička samosličnost ili Brownova³ samosličnost je ona koja se nalazi u prirodi, gdje ne postoji detalj koji je identičan cjelini. Detalji su kao i cjelina nepravilni, a upravo se takvi objekti nalaze u prirodi. Grančica nikad neće izgledati identično kao i drvo, no princip je uvijek isti. Primjer statističke samosličnosti je prikazan na slici 7, na primjeru drva.

³Robert Brown, zaslužan za otkriće *Brownovog kretanja*. Ako pratimo položaj mikroskopskih čestica u određenom vremenskom intervalu, dobit ćemo da se kreću po isprekidanoj putanji koja slijedi nasumce odabrane pravce. Krenemo li analizirati jedan od tih pravaca u kraćem vremenskom intervalu, dobit ćemo da se zapravo radi o većem broju isprekidanih dijelova pravca, isto tako nasumce odabranih (Brdar i sur., 2012).



Slika 7. Statistička samosličnost na primjeru drveta (Izvor: Freedesign4me, 2010)

Iako nijedna grana nije matematički ista kao i cijelo drvo, ili grančica kao grana, sličnost i dalje postoji.

Fraktali prema načinu nastanka

Fraktale prema načinu nastanka možemo podijeliti na sljedeći način (Brdar i sur. 2012):

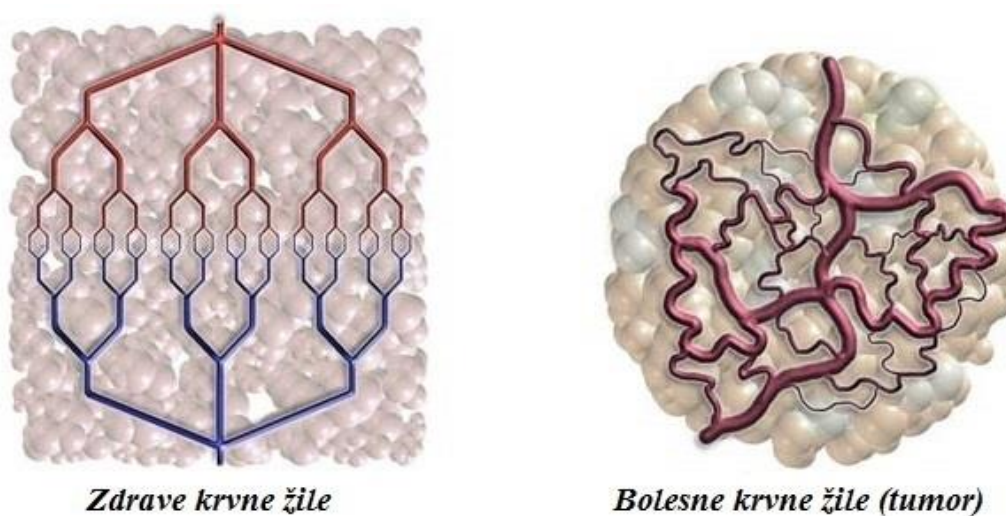
- a) kao što je već rečeno kod svojstva *oblikovanje iteracijom*, fraktali nastaju iteracijom tj. uzastopnim ponavljanjem nekog računskog postupka, a time dolazimo do **iterativnih fraktala**. Oni nastaju kopiranjem i rotiranjem i/ili translatiranjem kopije, te mogućim zamjenjivanjem nekog elementa tom kopijom.
- b) **rekurzivni fraktali** su određeni rekurzivnom matematičkom formulom koja određuje pripada li određena točka prostora određenom skupu ili ne,
- c) **slučajni fraktali** posjeduju najmanji stupanj samosličnosti i nalazimo ih često u prirodi, a to su drveće, obale, oblaci, munje i drugo.

2.4. Fraktali oko nas

Fraktali su stvarno sveprisutni i pronalaze primjenu u različitim sferama ljudske djelatnosti.

Ljudsko tijelo

Ako krenemo proučavati same sebe, uvidjet ćemo da u ljudskom tijelu postoje fraktalne strukture, pa su tako pluća, krvne žile i mozak najočitiji primjeri. Zašto je važno znati ovu činjenicu? Iz razloga što je danas uz medicinsko znanje, dostupnu tehnologiju i znanje o fraktalima moguće otkriti kada dođe do promjene u strukturi, tj. izgledu, primjerice, krvnih žila. Pod promjenom mislimo na razne bolesti, kao što su rak i tumor, a kod kojih rano otkrivanje i praćenje fraktalne strukture bolesnog dijela tijela oboljeloj osobi može uvelike pomoći. Na slici 8 je prikazan shematski dijagram normalnih krvnih žila i onih kod osobe s tumorom.



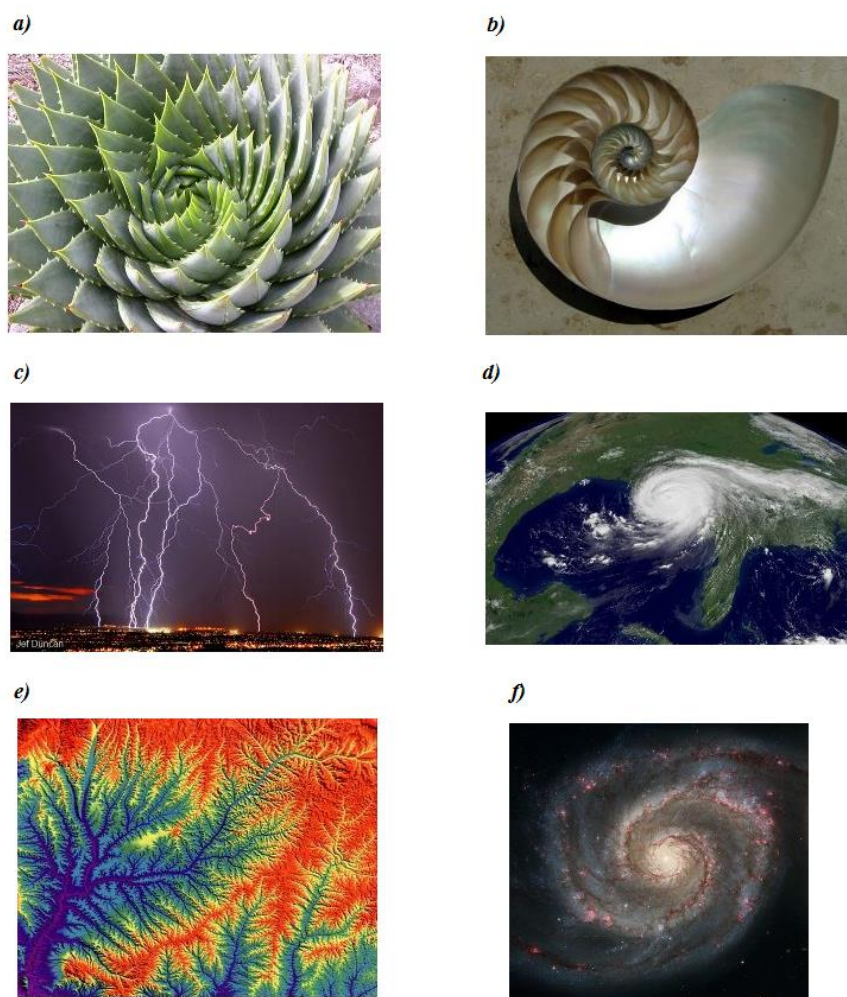
Slika 8. Zdrave i oboljele krvne žile (Prema: Fractal Foundation, s.a. d)

Priroda

U prirodi se na svakom koraku susrećemo s fraktalima, pa se tako fraktalni uzorci mogu vidjeti kod biljaka (drveće, kaktus, cvijeće, voće), rijeka, životinjskih oklopa, galaksija i prirodnih pojava kao što su munje i uragani.

Na slici 9 su prikazani primjeri fraktala u prirodi, a to su redom:

- a) agava kaktus (Fractal Foundation, s.a. g)
- b) nautilus školjka (Fractal Foundation, s.a. g)
- c) munja u gradu Albuquerque, u Novom Meksiku (Fractal Foundation, s.a. f)
- d) uragan Katrina (Fractal Foundation, s.a. g)
- e) samoslična rijeka iz Shaanxi provincije u Kini (Fractal Foundation, s.a. e)
- f) M51 „Vrtložna galaksija“ (engl. *Whirlpool Galaxy*)., (Fractal Foundation, s.a. g).



Slika 9. Primjeri fraktala u prirodi

Ostala područja primjene

U računalnoj grafici se fraktali koriste za kreiranje modela terena i vegetacije, što je i tema ovog diplomskog rada. Također, fraktali se mogu koristiti (vrlo ograničeno) za predviđanje stohastičkih procesa (potresi), slaganje optičkih vlakana, oponašanje rada neuronskih mreža za razvoj umjetne inteligencije, a za mobilne uređaje se proizvode antene u obliku fraktala kako bi mogle koristiti širok spektar frekvencija ne zauzimajući mnogo prostora (Brdar i sur. 2012). Na slici 10 je prikazan mobilni uređaj s antenom strukturiranom kao Sierpinskijev tepih.



Slika 10. Antena u mobilnom uređaju strukturirana kao Sierpinskijev tepih (Izvor: Fractal Foundation, s.a. h)

3. Osnovne vrste fraktala

Postoji mnogo različitih vrsta fraktala, od kojih su neki više, a neki manje poznati. Jedan mali dio sam spomenuo već u prethodnim poglavljima, ali sada ću dati pregled osnovnih vrsta fraktala.

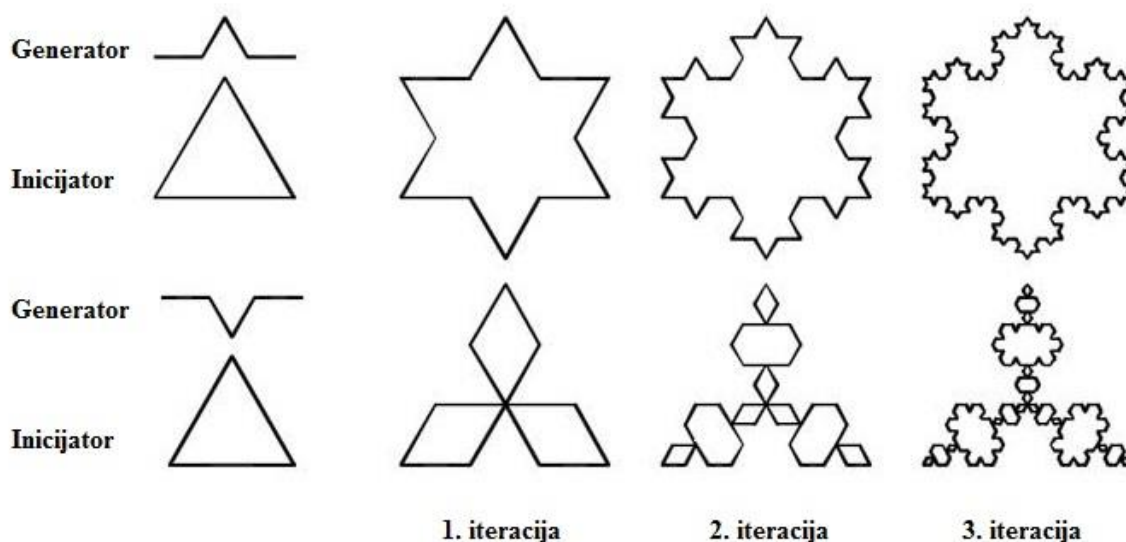
Baza-motiv fraktal

Najveći broj fraktala spada u ovaj tip fraktala, a njegov način nastanka je onda i način nastanka svih njegovih „inačica“.

Formira se na sljedeći način: potrebno je uzeti bilo koji oblik koji se sastoji od linijskih segmenata (baza) i odabrati drugi oblik (motiv) i nakon toga zamijeniti svaku bazu s motivom, kao i svaki sljedeći dobiveni oblik (ahyco.uniri.hr, 2007 a). Ove zamjene se mogu vršiti u beskonačnost.

Baza je najčešće dužina, četverokut i jednakostranični trokut, a motiv može imati mnogo različitih oblika. Što se tiče pozicioniranja motiva, ono može biti na sljedeće načine (ahyco.uniri.hr, 2007a):

- a) četverokut ili trokut se postavlja prema van (primjer je Kochova pahuljica, jedan od najpoznatijih baza-motiv fraktala, prikazana na slici 11 u prvom redu) ili prema unutrašnjosti lika (primjer je Kochova antipahuljica prikazana na slici 11 u drugom redu),
- b) motiv se postavlja na isti (pravilan baza-motiv fraktal) ili na različiti način (*Sweep* fraktal) tokom cijelog iterativnog postupka.



Slika 11. Kochova pahuljica i antipahuljica (Prema: Ibrahim i Krawczyk, 2006)

Prema dimenziji baza-motiv fraktali mogu biti (ahyco.uniri.hr, 2007 a):

- a) fraktalne krivulje ako im je dimenzija između broja 1 i broja 2,
- b) prašina ako im je dimenzija manja od broja 1,
- c) Peanova krivulja ako im je jednaka broju 2.

Prašina je jedan od tipova baza-motiv fraktala u kojemu su baza i motiv jednaki oblici s nekim odsječenim dijelovima. Najpoznatija prašina je Cantorov skup gdje je dužina razdijeljena u tri jednaka dijela, a srednja trećina je odstranjena (ahyco.uniri.hr, 2007 b).

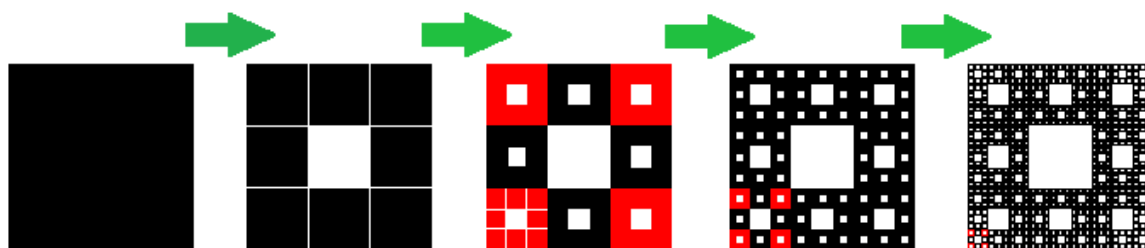
Skupine se mogu podijeliti u (ahyco.uniri.hr, 2007 b):

a) **2D skupina**

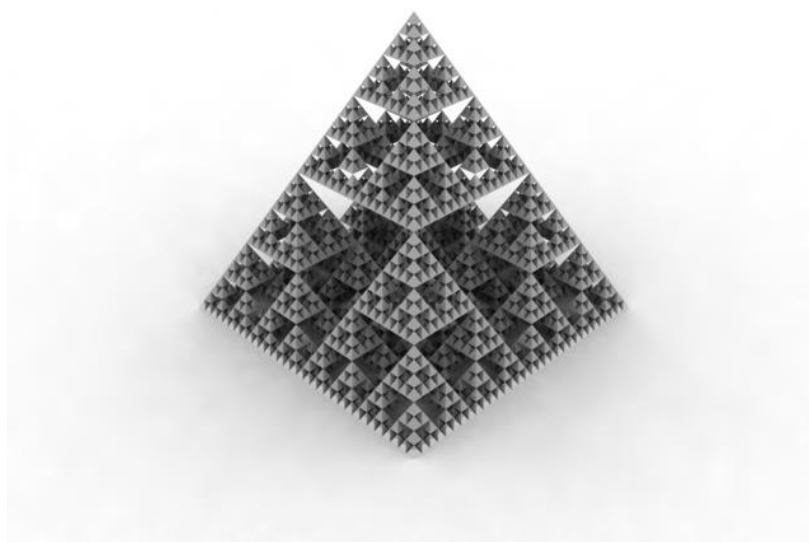
- umjesto dužine kao baze uzima se dvodimenzionalni lik, pa ako se uzme kvadrat dobivamo Sierpinskijev tepih (prikazan na slici 12) ili ako se uzme jednakostranični trokut dobivamo Sierpinskijev trokut (detaljnije objašnjen u podpoglavlju 3.1. *Poznati fraktali*),
- koriste se za modeliranje galaksija,

b) **3D skupina**

- umjesto dužine kao baze uzima se trodimenzionalni lik, pa ako se uzme piramida dobivamo Sierpinskijevu piramidu, prikazanu na slici 13.



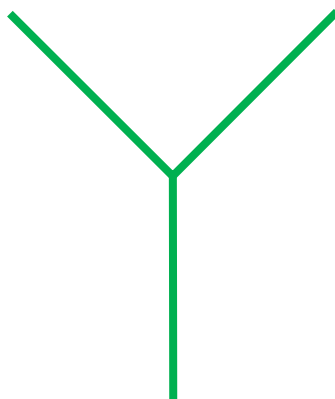
Slika 12. Sierpinskijev tepih (Prema: Riddle, 2013)



Slika 13. Sierpinskijeva piramida (Izvor: Tribe, 2006)

Fraktalni baldahini

Fraktali koje je najlakše kreirati jer je samo potrebno razdvojiti segment na njegovom kraju na dva manja segmenta i tako sa svakim sljedećim novim segmentom, kao što vidimo na slici 14.

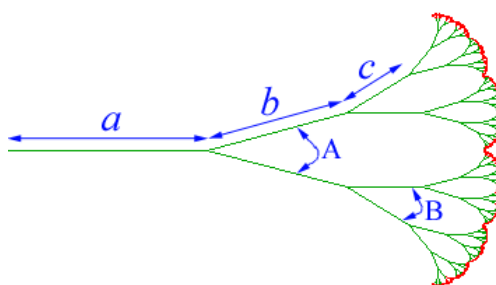


Slika 14. Nastanak fraktalnog baldahina

Fraktalni baldahin ima sljedeća svojstva (ahyco.uniri.hr, 2007 c):

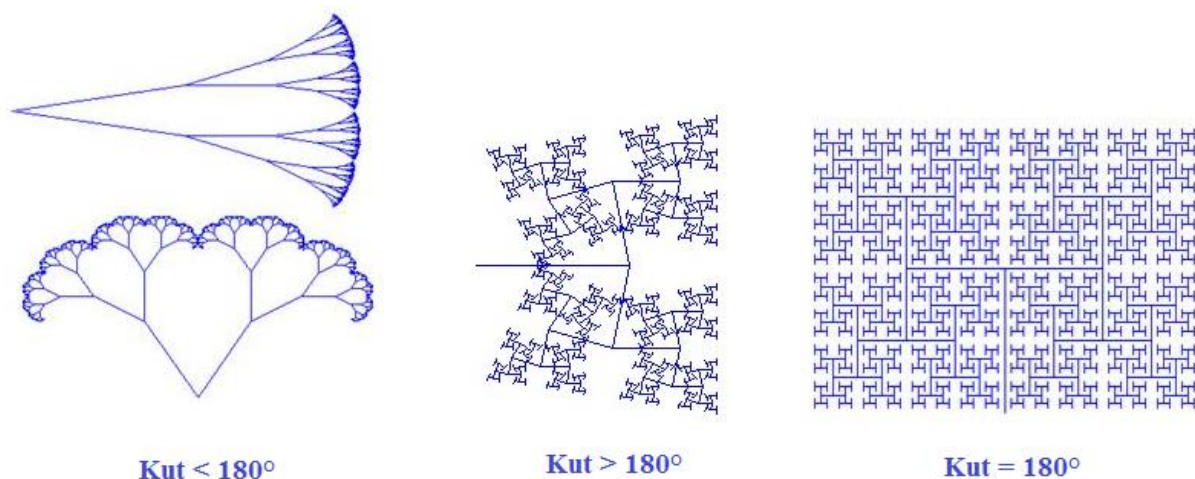
- 1) kut između bilo koja dva segmenta koje dobivamo razdvajanjem mora biti jednak u cijelom fraktalu ($A=B$);
- 2) omjer duljina uzastopnih dužina dobivenih razdvajanjem mora biti konstantan ($a/b=b/c$);
- 3) točke na krajevima najkraćih segmenta moraju biti međusobno povezane (crvena boja).

Dijagram na slici 15 pokazuje sva navedena svojstva.



Slika 15. Svojstva fraktalnog baldahina (Izvor: ahyco.uniri.hr, 2007 c)

Postoje različite vrste baldahina. Razlikuju se ovisno o kutu između segmenata, pa imamo one čiji je kut manji od 180° kao što je fraktalni kišobran, kut od 180° (H-fraktal) i one čiji je kut veći od 180° , a svi su prikazani na sljedećoj slici (slika 16).



Slika 16. Vrste fraktalnih baldahina (Prema: ahyco.uniri.hr, 2007 c)

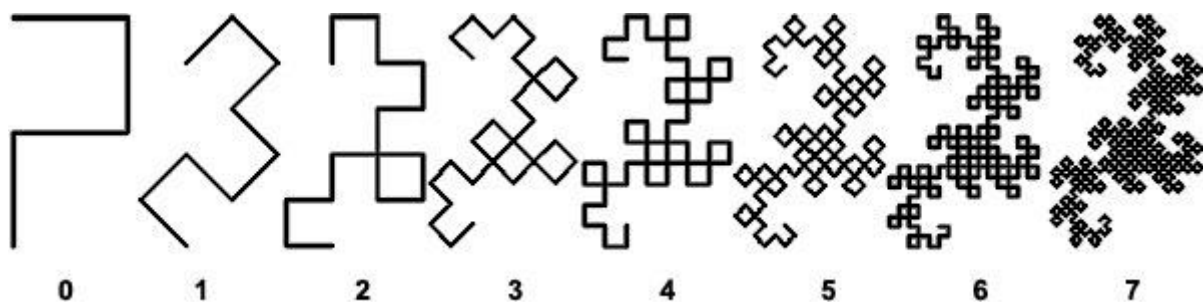
IFS fraktali

Najvažnija karakteristika *IFS* (engl. *Iterated Function Scheme*) fraktala je vrlo jasna samosličnost, pa se iz tog razloga primjenjuju kod izrade vrlo realističnih modela biljaka (ahyco.uniri.hr, 2007 d).

Formiraju se na način da se početna figura pomoću niza geometrijskih transformacija pretvori u nekoliko manjih figura i na tim manjim figurama se ponovi isti postupak, a taj postupak se onda može ponavljati u beskonačnost.

Najpoznatiji primjeri fraktala ovog tipa su paprat i fraktal *zmajeva krivulja*. Paprat će biti opisana u praktičnom dijelu rada (podpoglavlje 5.4), a *zmajeva krivulja* je ukratko opisana u nastavku.

Zmajeva krivulja se formira nizanjem zamjena jednu za drugom, kao što vidimo na sljedećoj slici (slika 17).



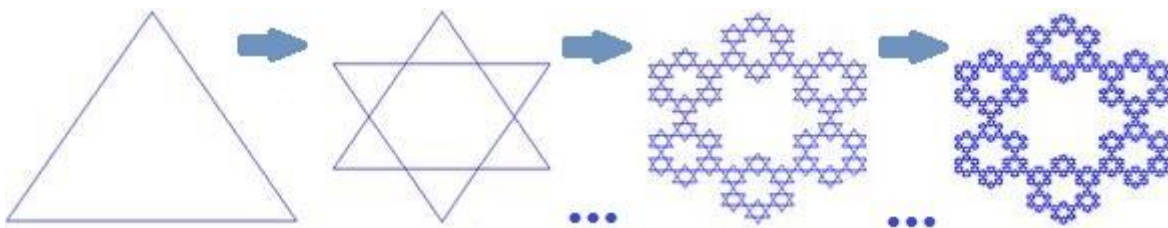
Slika 17. Nastanak zmajevе krivulje (Izvor: Fractal Foundation, s.a. a)

Početna slika, s oznakom nula se naziva generator. U prvoj iteraciji zamijeni se svaka polovica *zmajevе krivulje* s manjom kopijom istog oblika, rotiranu na način da pristaje. Postupak se ponavlja, a nakon svake iteracije ima dvostruko više kopija originalnog generatora (Fractal Foundation, s.a. a).

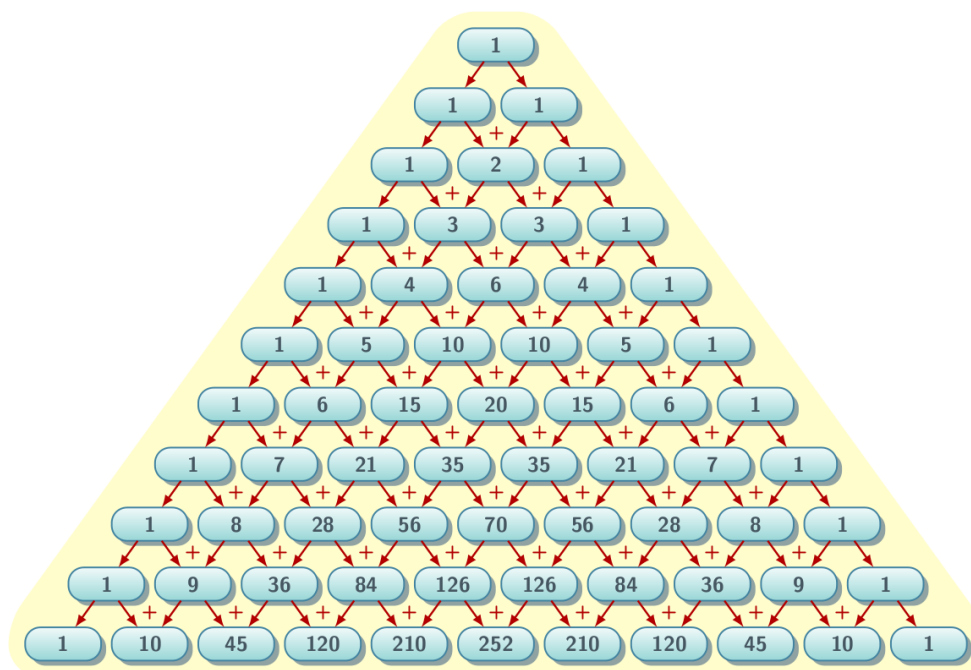
Nestandardni fraktali

U ovu skupinu fraktala spadaju sljedeće četiri vrste fraktala:

- 1) **lančani fraktali** gdje se koristi prsten umjesto standardnog segmenta, a postupak izrade je takav da se prsten u svakoj iteraciji zamijeni s lancem od određenog broja prstena spojenih u krug.
- 2) **Davidova zvijezda** je fraktal gdje se jednakokraničnom trokutu dodaje trokut okrenut u suprotnom smjeru (prikazan na slici 18).
- 3) **stanični automatizirani fraktal** kod kojeg iterativni postupak slijedi niz pravila, odnosno oponaša rast stanica (koristi se za simulaciju rasta bakterija), a najpoznatiji fraktal ovog tipa je *fraktal raspršenja* kod kojeg širenje ide od centra (ahyco.uniri.hr, 2007 e),
- 4) **Pascalov trokut** je fraktal kod kojeg se na mjesto vrhova prvog trokuta upisuje broj 1, a u svim sljedećim koracima se dodaje niz brojeva koji će vizualno stvarati dojam baze trokuta (brojevi su jednaki sumi dvaju brojeva iznad njih). Koristi se u matematici, a uzorak koji nastaje je isti kao i kod Sierpinskijevog trokuta (ahyco.uniri.hr, 2007 e). Prikazan je na slici 19.



Slika 18. Fraktal Davidova zvijezda (Prema: ahyco.uniri.hr, 2007 e)



Slika 19. Fraktal Pascalov trokut (Izvor: Olah, 2011)

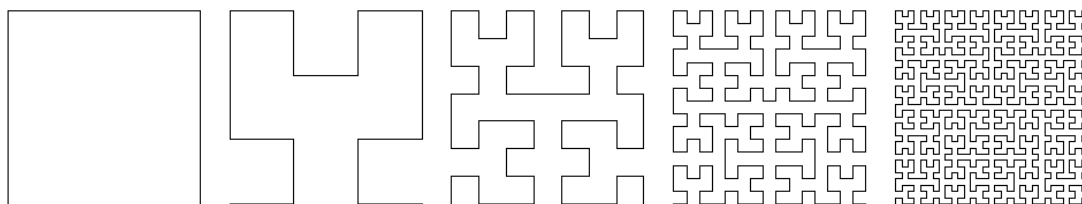
Fraktali dobiveni savijanjem papira

Kao što i samo ime kaže, to su fraktali koje dobijemo savijanjem papira. Papir možemo savijati uvijek u istom smjeru i pod pravim kutom i tada ćemo dobiti fraktal zmajeva krivulja koji je već objašnjen. Također, papir možemo savijati i pod nekim drugim kutem.

Svrstavaju se u Sweep fraktale koje ću objasniti u nastavku.

Peanove krivulje

Peano⁴ krivulja ili Hilbert⁵ krivulja je podskup obitelji takozvanih krivulja za popunjavanje prostora⁶ (engl. *space-filling curves*) koje su također fraktali. Kod Peano krivulje, dimenzija mora biti veća ili jednaka dva (jeff.tsai, s.a.). Na slici 20 je prikazano prvih pet iteracija Peano krivulje.



Slika 20. Prvih pet iteracija Peano krivulje (Izvor: jeff.tsai, s.a.)

To su fraktali tipa baza-motiv, kod kojih je baza segment, motiv je jednostavan i završna slika je četverokut (ahyco.uniri.hr, 2007 f).

Plazma fraktali

Fraktali koji su u praksi možda najkorisniji od svih, a primjenjuju se kod izrade specijalnih efekata u filmovima i kod izrada modela oblaka i krajolika.

Ako bismo htjeli napraviti fraktal na pravokutniku u ravnini, napraviti ćemo ga na sljedeći način (ahyco.uniri.hr, 2007 g):

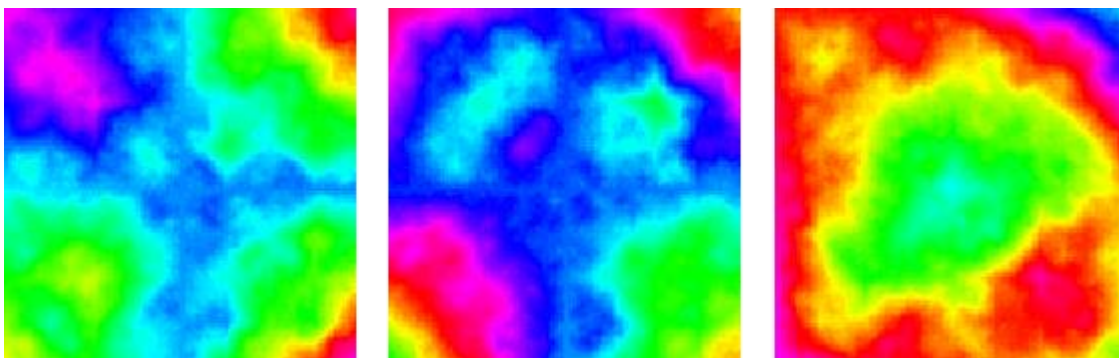
- 1) odaberu se vrijednosti koje će se pridružiti vrhovima pravokutnika,
- 2) izračuna se vrijednost koja će se pridružiti centru pravokutnika tako da se uzme prosjek vrijednosti pridruženih vrhovima i zbroji sa slučajnim brojem koji je pomnožen s namještenim parametrom grubosti,
- 3) izračunaju se vrijednosti središnjih točaka stranica pravokutnika tako da se uzme prosjek dva najbliža vrha i zbroji sa slučajnim brojem koji je pomnožen s namještenim parametrom grubosti,
- 4) sada su nastala četiri mala pravokutnika,
- 5) ponove se koraci od 2) do 4) za svaki od tih novonastalih pravokutnika,
- 6) na samom kraju se može nacrtati slika tako da se oboje točke ovisno o njihovoj vrijednosti.

⁴ Guiseppe Peano je talijanski matematičar koji je prvi otkrio tip krivulje za popunjavanje prostora 1890. godine.

⁵ David Hilbert je njemački matematičar koji je 1891. godine napravio varijantu Peanove krivulje.

⁶ Krivulja koja može proći kroz svaku točku u određenom segmentu n-dimenzionalnog prostora.

Na slici 21 je prikazan fraktal s različito obojenim točkama ovisno o njihovoj vrijednosti.



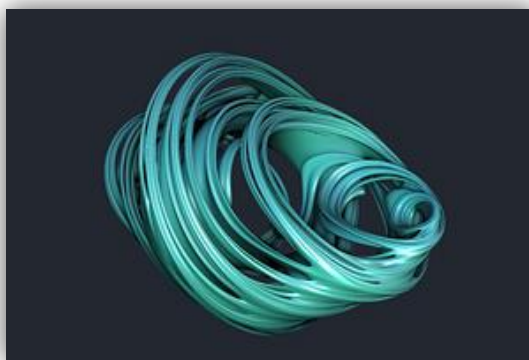
Slika 21. Fraktal s različito obojenim točkama ovisno o njihovoj vrijednosti (Prema: Sedgewick i Wayne, 2012)

Kvaternion

Fraktali koje je 1847. godine otkrio William Rowan Hamilton, irski matematičar koji je prvi opisao istoimeni numerički sustav⁷.

To su 3D verzije Juliaovog skupa, koji ću objasniti u sljedećem poglavlju.

Primjer jednog takvog fraktala je prikazan na slici 22.



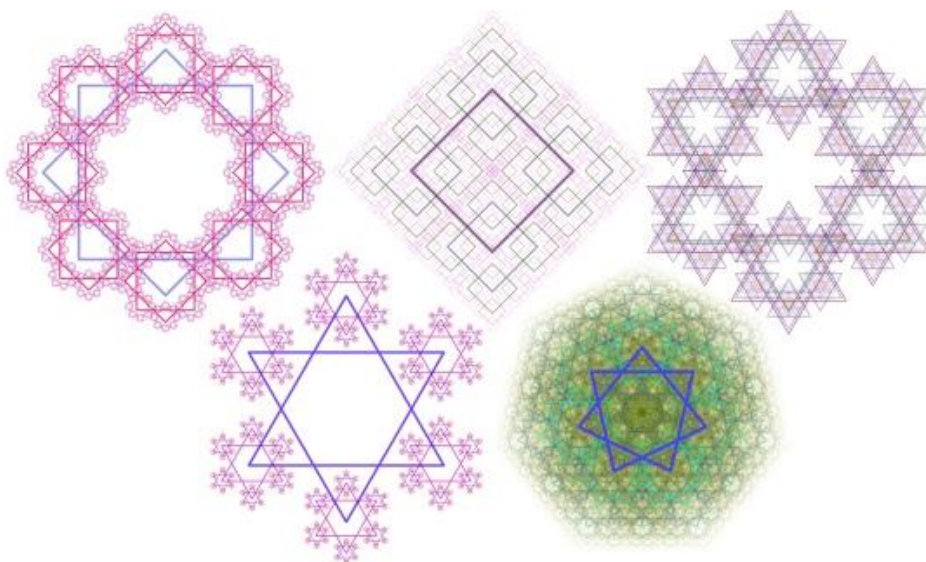
Slika 22. Kvaternion (Izvor: Beddard, 2009)

Zvezdani fraktali

Fraktal koji nastaje na način da se odabere geometrijski lik i nakon toga se manje verzije tog istog geometrijskog lika „lijepe“ na originalni (početni) geometrijski lik.

Na slici 23 su prikazane različite verzije zvjezdanih fraktala.

⁷ Brojevni sustav koji proširuje kompleksne brojeve tako da umjesto jedne, postoje tri imaginarne jedinice i , j , i k za koje vrijedi sljedeće: $i^2 = j^2 = k^2 = ijk = -1$.



Slika 23. Primjeri zvjezdanih fraktala (Izvor: Stone Design, s.a.)

Sweeps

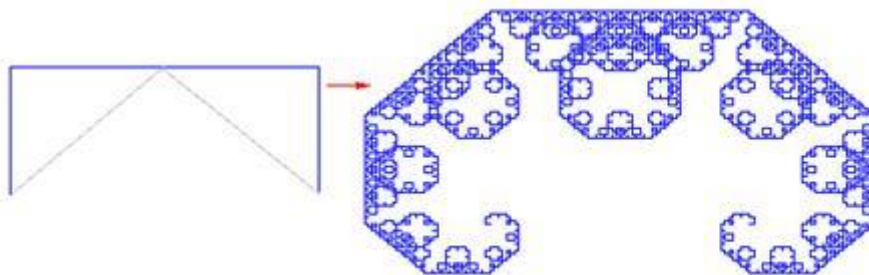
Kod ove vrste fraktala se tokom iteracije motiv smješta na drugačiji način, a moguća su tri načina smještanja, i to koristeći:

- 1) preokrenutu sliku motiva,
- 2) zrcalnu sliku motiva,
- 3) preokrenutu zrcalnu sliku motiva.

Isto tako, moguće su tri metode kreiranja, pomoću kojih se mogu dobiti četiri različita motiva s istom bazom i motivom, a te metode su (ahyco.uniri.hr, 2007 h):

- 1) izmjenjivati normalnu i izmijenjenu verziju istog motiva u koracima iteracije (primjer je Cesarova krivulja),
- 2) započeti sve korake iteracije istom verzijom motiva, ali izmijeniti verziju između različitih linijskih segmenata (primjer je zmajeva krivulja),
- 3) započeti različite korake iteracije različitim verzijama (primjer je Polyina krivulja).

Kada se ne upotrebljava nijedna metoda dobije se Levijeva krivulja, prikazana na slici 24.



Slika 24. Levijeva krivulja (Izvor: ahyco.uniri.hr, 2007 h)

Neobični atraktori

Neobični atraktor je uzorak koji postoji u kompleksnom matematičkom prostoru i ima fraktalna svojstva. On predstavlja parametarski prostor kaotičnog sustava tako da rezultat svake jednadžbe postaje parametar u sljedećoj jednadžbi. Također, neobični atraktor može se shvatiti kao rješenje nelinearne jednadžbe, dinamičnog ili kaotičnog sustava (Gardi, 2009).

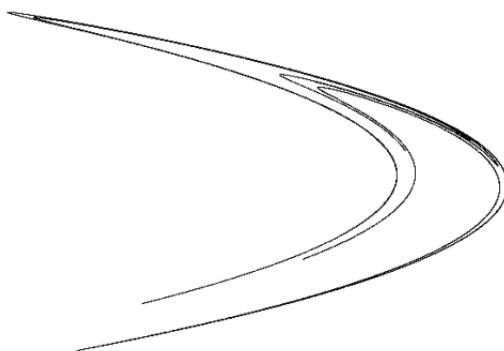
Kvadratni atraktori su najčešći oblik neobičnih atraktora, a najpoznatiji kvadratni atraktor je Henonov atraktor, prikazan na slici 25.

Henonov atraktor izračunava se po formulama (Weisstein, s.a.):

$$x_{n+1} = y_n + 1 - ax_n^2$$

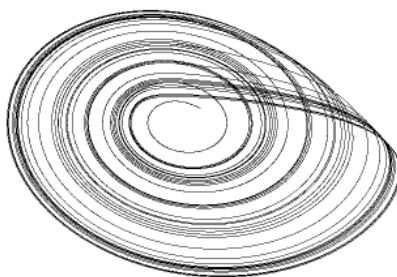
$$y_{n+1} = bx_n$$

Rezultat ovisi o vrijednosti parametra a i b , koji prilikom izračuna zadržavaju konstantnu vrijednost. Za klasičan Henonov atraktor $a = 1.4$, dok je $b = 0.3$ (Weisstein, s.a.).

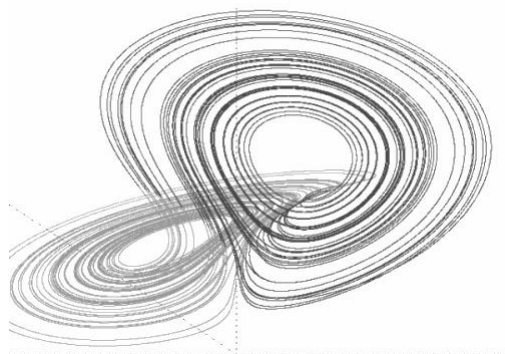


Slika 25. Henonov atraktor (Prema: Bradley, 2010)

Neobični atraktori se koriste u izučavanju promjena kod populacija, vremenskih prilika i kemijskih reakcija te su mnogi uzorci za fraktale pronađeni upravo u ovim prirodnim i društvenim pojavama. Fraktali kao što je Rosslerov atraktor, prikazan na slici 26 i Lorenzov atraktor, prikazan na slici 27, su otkriveni proučavajući prirodu (ahyco.uniri.hr, 2007 i).



Slika 26. Rosslerov atraktor (Izvor: Bradley, 2010)



Slika 27. Lorenzov atraktor (Izvor: Bradley, 2010)

3.1. Poznati fraktali

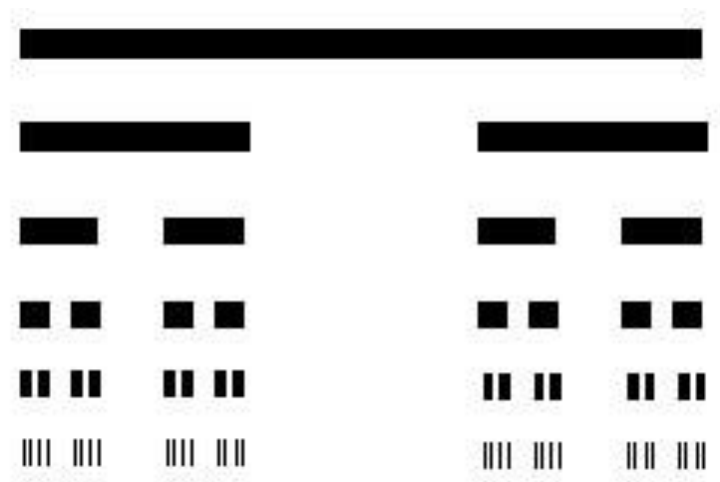
U ovu skupinu fraktala spadaju fraktali koji su poznati zbog svojih autora i najčešće se primjenjuju u različitim sferama ljudske djelatnosti.

Cantorov skup

Smatra se najpoznatijim i najjednostavnijim fraktalom, a ime je dobio prema njemačkom matematičaru Georgu Cantoru, koji je predstavio istoimeni skup 1883. godine.

Spada u baza-motiv fraktale gdje ravnu liniju smatramo bazom fraktala, a prazninu koja nastaje odstranjivanjem srednje trećine te linije smatramo motivom fraktala. To je ujedno i način nastanka ovog tipa fraktala. Postupak počinje odstranjivanjem srednje trećine linijskog segmenta, zatim odstranjivanjem srednje trećine preostala dva linijska segmenta i tako dalje sa svakim novonastalim (preostalim) segmentom. Nakon beskonačno mnogo dijeljenja, ostaje podskup realnih brojeva zvan *Cantorova prašina* (Platonic Realms, s.a.).

Naden u Saturnovim prstenima i u nekim molekulama (ahyco.uniri.hr, s.a. I), a njegovih prvih pet iteracija je prikazano na slici 28.



Slika 28. Cantorov skup (Izvor: R.C.L., 2010)

Juliaov skup

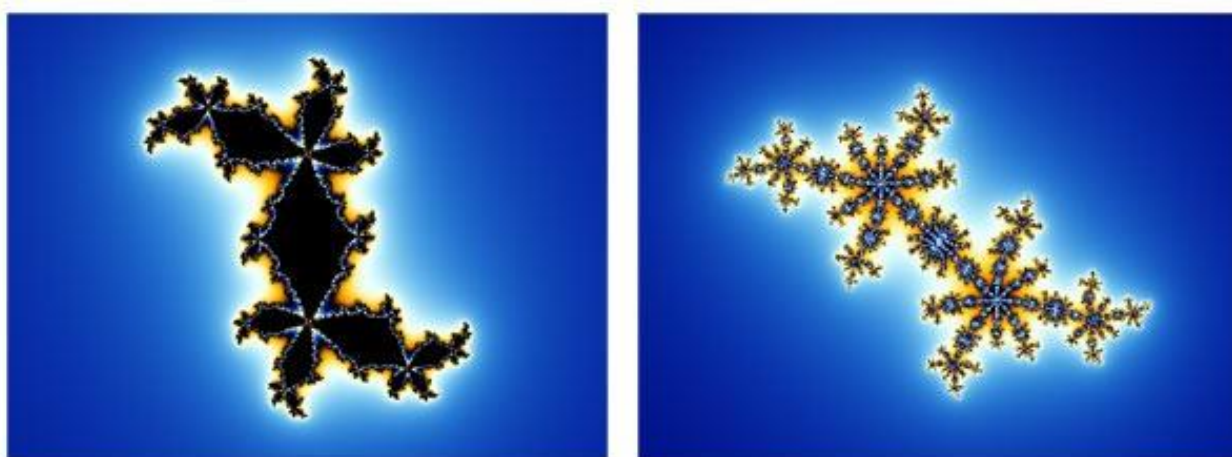
Fraktal je dobio naziv po francuskom matematičaru Gaston Julia koji je otkrio te skupove početkom 20. stoljeća, a detaljniju analizu su doživjeli razvojem računala. Može koristiti vrlo jednostavno preslikavanje, a biti vrlo složen (ahyco.uniri.hr, 2007 j).

Formira se na sljedeći način (ahyco.uniri.hr, 2007 j):

- 1) izabere se bilo koji kompleksan broj i i izjednači sa c (konstanta Juliaovog skupa⁸),
- 2) izabere se neki drugi kompleksan broj i i izjednači se sa z ,
- 3) z se izjednači sa $z^2 + c$ (ova promjena se ponovi više puta),
- 4) ako se broj ubrzano povećava prema beskonačnosti, ne treba označiti odgovarajuću točku u kompleksnoj ravni, dok suprotno znači da pripada skupu i onda se označi,
- 5) koraci od 2) do 5) se ponove za različite brojeve (dok sve točke u ravni nisu provjerene).

Izgled Juliaovog skupa ovisi o tome da li je njegova konstanta c element Mandelbrotovog skupa (objašnjen u nastavku). U slučaju da je, tada će Juliaov skup biti u potpunosti povezana figura, a ako nije, dobiva se skupina nepovezanih točaka (ahyco.uniri.hr, 2007 k).

Na slici 29 su prikazana dva različita Juliaova skupa.



Slika 29. Primjeri Juliaovih skupova (Prema: Fractal Foundation, s.a. b)

Mandelbrotov skup

Fraktal je dobio ime prema već spomenutom, matematičaru Benoit Mandelbrotu, koji ga je proučavao i popularizirao. Kao i Juliaov skup, spada u fraktal koji nastaje preslikavanjem i koristi slične algoritme kao i taj skup.

⁸ Konstanta c može biti bilo koji kompleksan broj, i svaki stvara različit Juliaov skup. Ista formula s različitim konstantama može proizvesti fraktale koji uopće ne nalikuju jedan drugome (ahyco.uniri.hr, 2007 j).

Formira se na sljedeći način (ahyco.uniri.hr, 2007 k):

- 1) neka je $z = 0 + 0i$,
- 2) odabere se neka točka u kompleksnoj ravnini i njene koordinate izjednače sa c ,
- 3) zatim neka je $z = z^2 + c$ (najčešće korištena formula) i ponovi se ta promjena više puta i to preslikavanje se iterira više puta,
- 4) ako broj ne ode u beskonačnost, pripada skupu i označi se, a ako ode onda se točka oboji ovisno o tome koliko brzo ide u beskonačnost,
- 5) prethodni koraci se ponove za sve točke u ravnini.

Mandelbrotov skup nije savršeno samosličan i stilovi objekata s različitih dijelova ovog fraktala su poprilično različiti, a uzorci postaju sve kompleksniji i vizualno privlačniji što više se uvećava taj određeni dio na fraktalu (Fractal Foundation, s.a. c).

Gore navedeno se vidi na sljedećoj slici (slika 30), gdje su prikazani detalji s četiri različita područja unutar Mandelbrotovog skupa (uvećana područja).



Slika 30. Detalji sa četiri različita područja unutar Mandelbrotovog skupa (Prema: Fractal Foundation, s.a. c)

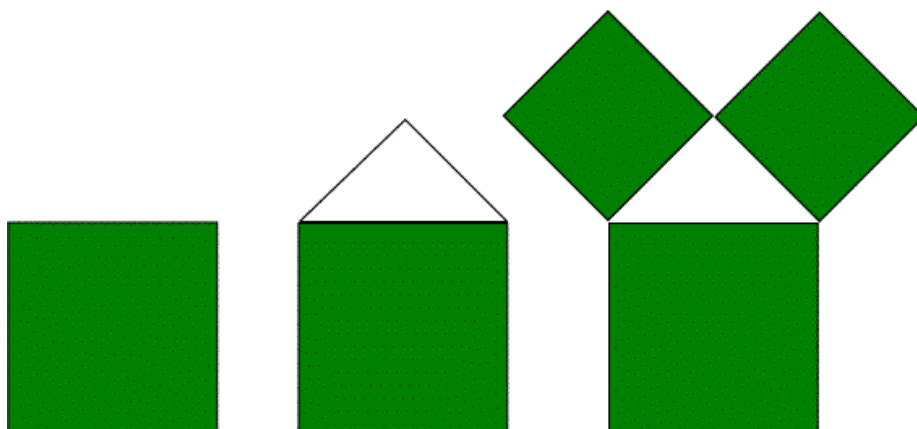
Pitagorino stablo

Fraktal je dobio naziv po grčkom matematičaru Pitagori iz razloga što konstrukcija fraktala prikazuje geometrijski dokaz Pitagorinog teorema⁹ (Riddle, 2013 b).

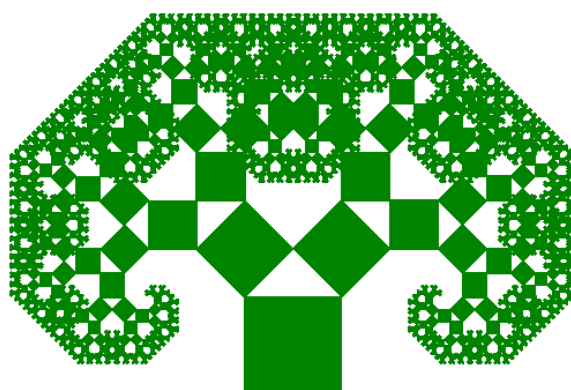
Moglo bi se reći da je inačica fraktalnog baldahina jer umjesto razlomljenih linija koristi kvadrate i jednakokračne trokute.

⁹ Kvadrat nad hipotenuzom pravokutnog trokuta jednak je zbroju kvadrata nad dvije katete.

Način formiranja je prikazan na slici 31, a fraktal je prikaza na slici 32.

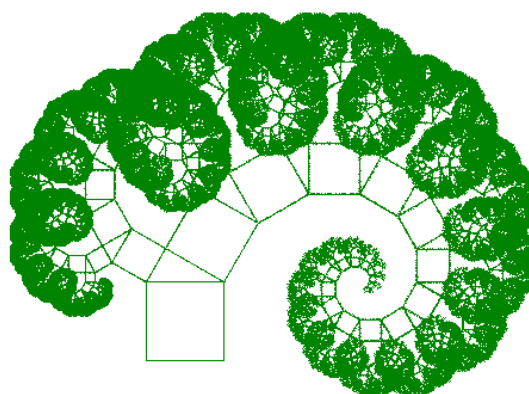


Slika 31. Formiranje Pitagorinog stabla (Izvor: Riddle, 2013 b)



Slika 32. Pitagorino stablo (Izvor: Riddle, 2013 b)

S obzirom na kut u trokutu, može se kreirati i stablo gdje su kutovi u trokutu na primjer 60° i 30° , a prikazano je na slici 33 (nakon 20 iteracija).



Slika 33. Asimetrično Pitagorino stablo (Izvor: Riddle, 2013 b)

Najčešće nalazi primjenu kod modeliranja biljaka.

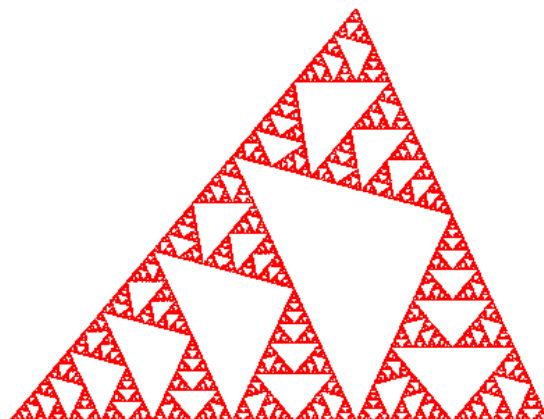
Sierpinskijev trokut sam već spomenuo u prethodnim poglavljima, a ovdje ću još spomenuti načine nastanka različitih verzija ovog fraktala (ima ih pet) i prikazati ih.

1. iteracija 2. iteracija 3. iteracija

Dakle, način formiranja različitih verzija Sierpinskijevog trokuta se može promatrati s geometrijskog gledišta (odstranjivanje dijelova trokuta) i sa *IFS* gledišta (skaliranje, rotiranje i prijenos) (Riddle, 2013 c).

1. kreira se šiljasti trokut T s vrhovima A , B i C ,
2. *pedal* trokut od T je trokut omeđen s točkama A_1 , B_1 i C_1 , odnosno točkama u koje se spuštaju okomice trokuta T iz vrhova A , B i C na suprotne stranice,
3. točka E je točka u kojoj se sijeku te okomice i ortocentar trokuta T ,
4. dobiveni *pedal* trokut dijeli trokut T u četiri manja trokuta,
5. daljnji postupak dobivanja Sierpinskijevog *pedal* trokuta je isti kao i kod klasičnog Sierpinskijevog trokuta (odstranjivanjem središnjeg trokuta).

¹⁰ Trokut dobiven projiciranjem točke na stranice trokuta.

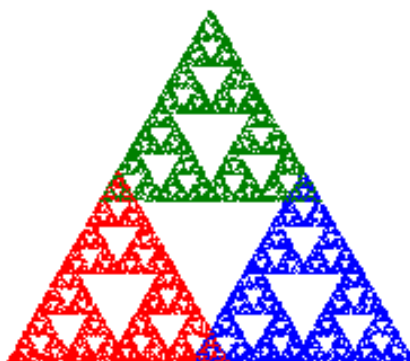


Slika 36. Sierpinskijev pedal trokut (Izvor: Riddle, 2013 d)

Sierpinskijev trokut se može formirati i na sljedeći način (Riddle, 2013 e):

1. kreira se istostraničan trokut i skalira se ili:
 - a. faktorom $r = \frac{1}{2}$ ili
 - b. brojem λ s vrijednostima između 0 i 1,
2. naprave se tri kopije tog skaliranog trokuta,
3. trokuti se transliraju na način da se uklapaju u originalan trokut,
4. postupak se ponavlja u beskonačnost na novo dobivenim trokutima.

Oblik Sierpinskijevog trokuta se mijenja s obzirom na vrijednost broja λ , a ako je $\frac{1}{2} < \lambda < \frac{2}{3}$ onda se naziva **debeli Sierpinskijev trokut**, a prikazan je na slici 37.

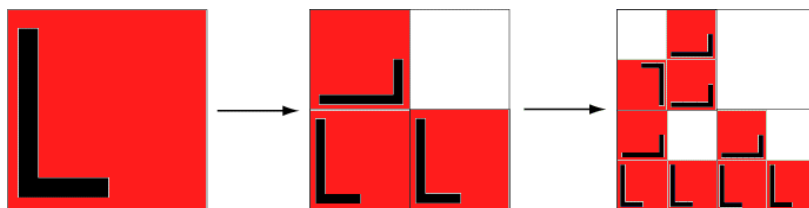


Slika 37. Debeli Sierpinskijev trokut (Izvor: Riddle, 2013 c)

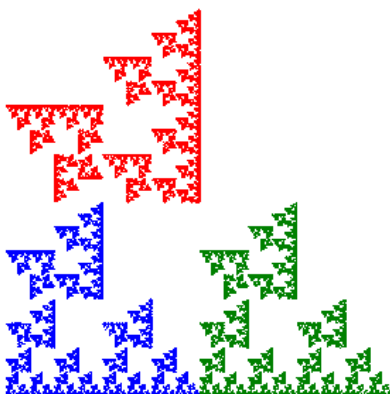
Ostale verzije ovog trokuta podrazumijevaju:

1. rotaciju jednog od trokuta,
2. upotrebu kvadrata i transformacije nad tim kvadratom, tj. podjelu, odstranjivanje i rotaciju (formiranje ovog tipa trokuta je prikazano na slici 38, a konačan izgled na slici 39),

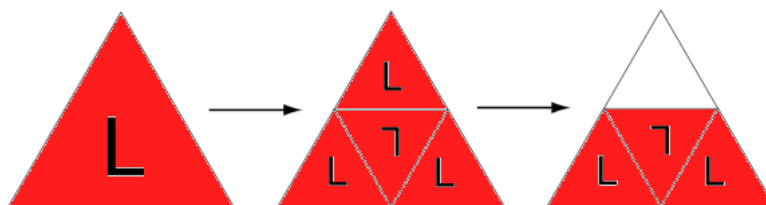
3. odstranjivanje gornjeg trokuta umjesto središnjeg i rotiranje novonastale konstrukcije (formiranje ovog tipa trokuta je prikazano na slici 40, a konačan izgled na slici 41) .



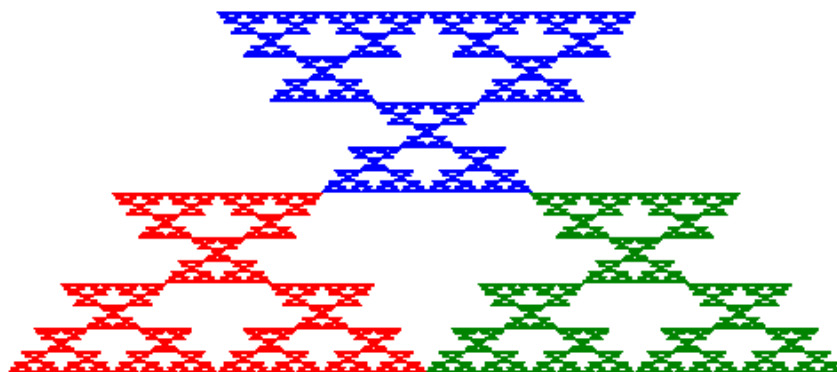
Slika 38. Formiranje Sierpinskijevog trokuta upotrebom kvadrata (Prema: Riddle, 2013 f)



Slika 39. Sierpinskijev trokut ako je korišten kvadrat (Izvor: Riddle, 2013 f)



Slika 40. Formiranje Sierpinskijevog trokuta odstranjivanjem gornjeg trokuta (Prema: Riddle, 2013 g)



Slika 41. Sierpinskijev trokut ako je odstranjen gornji trokut (Izvor: Riddle, 2013 g)

4. WebGL

WebGL je Javascript API (programsko sučelje) za kreiranje 2D i 3D grafike. Integriran je u najnovije web standarde, te je neovisan o platformi odnosno operacijskom sustavu. Baziran je na OpenGL ES 2.0 (Wikipedia, 2013 a).

Počeci WebGL-a datiraju iz 2006. godine kada je Vladimir Vukičević počeo eksperimentirati s 3D Canvasom. 2009. godine, neprofitabilna organizacija Khronos počinje intenzivnije razvijati WebGL, u suradnji s Apple, Google i drugim velikim korporacijama. Google Maps jedna je od prvih značajnih aplikacija koja je koristila WebGL (Wikipedia, 2013 a).

Iako nije jedini način za prikazivanje 3D grafike u mrežnim preglednicima, relativno je najbolji s obzirom na mogućnosti i činjenicu da ne zahtijeva instalaciju nikakvih dodatnih programa. Jedino ograničenje je sam preglednik koji mora podržavati WebGL. Trenutno ga od najčešće korištenih preglednika jedino Internet Explorer (verzija 10) ne podržava.

Pojednostavljena arhitektura sustava, odnosno način na koji WebGL pretvara programski kod u sliku na ekranu je prikazana na slici 42.



Slika 42. WebGL arhitektura

Prilikom poziva metode za iscrtavanje, za svaki definirani vrh (engl. *vertex*) učitava se program za sjenčanje vrhova (engl. *vertex shader*) kojemu se proslijeđuje pozicija vrha u koordinatnom sustavu, boja i slično, te uniformne varijable kao što su projekcijska matrica i matrica modela. One se ne mijenjaju prilikom iscrtavanja i služe kako bi se vrh ovisno o poziciji i rotaciji (matrica modela) te položaju kamere i samoj vrsti projekcije (projekcijska matrica) transformirao u koordinatni sustav kakav je pogodan za prikazivanje na ekranu (zaslonske koordinate). Program za sjenčanje vrhova obavezno na kraju svog rada mora imati postavljenu vrijednost varijable *gl_Position*, odnosno poziciju vrha na ekranu. Osim te obavezne varijable moguće je definirati, postaviti i/ili proslijediti druge varijable kao što je boja i slično, a one se proslijeđuju programu za sjenčanje točaka (engl. *fragment shader*) (Thomas, 2009).

Program za sjenčanje točaka se poziva za svaki slikovni element (engl. *pixel*) na slici. To znači da će se program pozvati i za slikovne elemente na kojem nema niti jednog vrha. U tom slučaju program za sjenčanje točaka će odrediti boju trenutnog slikovnog elementa linearnom interpolacijom između slikovnih elemenata koji sadrže vrhove, a okružuju trenutni (Thomas, 2009). Uz to, program za sjenčanje točaka primjenjuje različite efekte, kao što je magla i sl.

Konačan rezultat obrade programa za sjenčanje točaka je slika koja će se prikazati na ekranu (Thomas, 2009).

4.1. WebGL u praksi

WebGL se prikazuje unutar HTML elementa, a sam programski kod se izvodi u Javascriptu. HTML dio je prilično jednostavan. Potrebno je definirati jedino `<canvas>` element unutar kojeg će se iscrtati WebGL program:

```
<body onload="webGLStart();">
  <canvas id="webgl-canvas" width="640" height="480"></canvas>
</body>
<script>
  function webGLStart() {
    initGL();
    initShaders();
    initObjects();

    gl.clearColor(1.0, 1.0, 1.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    animate();
  }
</script>
```

Element `<canvas>` dio je HTML5 specifikacije. Uz to, definira se i metoda koja će se pozvati kada se stranica učita, a to je metoda koja će učitati WebGL, odnosno `webGLStart()`. Spomenuta metoda učitava WebGL, programe za sjenčanje (engl. *shader*) i definira objekte koji će se iscrtati, postavlja generalne postavke (boja pozadine, test dubine itd.), te konačno pokreće animaciju odnosno iscrtavanje.

Učitavanje WebGLa vrši se na način da se HTML elementu dodjeljuje kontekst kroz naredbe:

```
<script>
  function initGL() {
    var canvas = document.getElementById("webgl-canvas");
    gl = canvas.getContext("experimental-webgl");
    gl.viewportWidth = canvas.width;
    gl.viewportHeight = canvas.height;
  }
</script>
```

Ako kojim slučajem dođe do greške prilikom izvođenja te naredbe (npr. preglednik ne podržava WebGL) programski kod koji slijedi se ne bi trebao izvršavati. Dodatno, određuje se visina i širina prozora unutar kojeg će se program iscrtati (definirano pomoću *width* i *height* atributa na `<canvas>` elementu).

Sljedeći korak je učitavanje programa za sjenčanje. Programi za sjenčanje su pisani u GLSL-u (Open GL Shading Language), i izvode se direktno na grafičkom procesoru (GPU):

```
<script id="shader-vs" type="x-shader/x-vertex">
    attribute vec3 aVertexPosition;
    attribute vec4 aVertexColor;
    uniform mat4 uMVMMatrix;
    uniform mat4 uPMatrix;
    varying vec4 vColor;

    void main(void) {
        gl_Position = uPMatrix * uMVMMatrix * vec4(aVertexPosition, 1.0);
        vColor = aVertexColor;
    }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
    precision mediump float;
    varying vec4 vColor;

    void main(void) {
        gl_FragColor = vColor;
    }
</script>
```

Programi za sjenčanje se uključe kroz `<script>` element ali sa specifičnim "type" atributom kako se ne bi izvršili kao javascript kod. U gornjem programskom isječku prikazani su osnovni programi za sjenčanje. Kod programa za sjenčanje vrhova, WebGL očekuje da se postavi varijabla `gl_Position` i ona označava poziciju na ekranu, a u najjednostavnijem slučaju se dobije umnoškom projekcijske matrice (`uPMatrix`), matrica modela i pogleda (`uMVMMatrix`) te proslijeđenog vektora koji sadrži koordinate vrha (`aVertexPosition`). Slično tome, kod programa za sjenčanje točaka očekuje se da bude postavljena vrijednost varijable `gl_FragColor`. Programi za sjenčanje također sadrže definiciju uniformnih varijabli (varijable koje su dostupne izvan samih programa za sjenčanje - `uPMatrix`, `uMVMMatrix` u primjeru), te atributa. S obzirom da se programi za sjenčanje izvode na grafičkom procesoru, koji je dizajniran specifično za izračune vezane za grafiku (množenje matrica itd.), oni uvelike pomažu u optimiziranju samog programa, kao i kod prikaza kompleksnijih specijalnih efekata.

Samo uključivanje programa za sjenčanje u WebGL program vrši se na sljedeći način:

```
<script>
    function initShaders() {
        var fragmentShader = getShader(gl, "shader-fs");
        var vertexShader = getShader(gl, "shader-vs");

        shaderProgram = gl.createProgram();
        gl.attachShader(shaderProgram, vertexShader);
        gl.attachShader(shaderProgram, fragmentShader);
        gl.linkProgram(shaderProgram);
        gl.useProgram(shaderProgram);

        shaderProgram.vertexPositionAttribute =
gl.getAttribLocation(shaderProgram, "aVertexPosition");
        gl.enableVertexAttribPointer(shaderProgram.vertexPositionAttribute);
        ...
    }
</script>
```

Programi za sjenčanje kreiraju se kroz metodu *getShader()* koja je prikazana u nastavku, te se uključuju u glavni program. Potrebno je i referencirati attribute i uniformne varijable (na primjeru prikazano samo za *aVertexPosition*).

Metoda *getShader()* izgleda ovako:

```
<script>
function getShader(gl, id) {
    var shaderScript = document.getElementById(id);
    var str = //getScriptContent;
    var shader;
    if (shaderScript.type == "x-shader/x-fragment")
        shader = gl.createShader(gl.FRAGMENT_SHADER);
    else if (shaderScript.type == "x-shader/x-vertex")
        shader = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(shader, str);
    gl.compileShader(shader);
    return shader;
}
</script>
```

U prikazanoj metodi, preko proslijeđenog ID-a pronalazi se skripta koja sadrži program za sjenčanje, ovisno o tipu skripte sadržaj se prevodi (engl. *compile*) te vraća natrag kao objekt u pozivajuću funkciju.

Sada kada su WebGL i programi za sjenčanje učitani možemo definirati objekt (ili više njih) koji će se iscrtavati na ekranu, a to će se učiniti pomoću metode *initObjects()* na sljedeći način:

```
<script>
function initObjects() {
    triangleVertexPositionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
    var vertices = [
        0.0, 1.0, 0.0,
        -1.0, -1.0, 0.0,
        1.0, -1.0, 0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
gl.STATIC_DRAW);
    triangleVertexPositionBuffer.itemSize = 3;
    triangleVertexPositionBuffer.numItems = 3;

    ...
}
</script>
```

U ovom slučaju objekt kojeg iscrtavamo je običan trokut i kroz varijablu *vertices* definiramo njegova 3 vrha. Pozicija je definirana s 3 vrijednosti za svaku koordinatnu os. Na identičan način postavile bi se i boje za naš objekt, jedina razlika je što bi polje bilo 3*4, jer se boja definira sa 4 vrijednosti, i shodno tome, *itemSize* bi bio 4, a *numSize* bi ostao 3.

Konačno, sve što preostaje je pokretanje animacije:

```
<script>
    function animate() {
        requestAnimFrame(animate);
        drawScene();
        updateScene();
    }
</script>
```

requestAnimFrame() je metoda koja periodički poziva proslijeđenu metodu. Slično kao i Javascript *setTimeout()* metoda, s razlikom da se posljednja navedena poziva čak i kada prozor nije aktivan, i vrijedi za sve otvorene kartice. S druge strane, *requestAnimFrame()* radi samo kada je kartica i prozor u kojem se izvršava program aktivan. Metoda *updateScene()* ažurira varijable koje se mijenjaju unutar animacije. U ovom slučaju gdje se radi o rotirajućem trokutu, jedina varijabla koju je potrebno mijenjati je sama rotacija, što vidimo u sljedećem programskom isječku:

```
<script>
    function updateScene() {
        var timeNow = new Date().getTime();
        if (lastTime != 0) {
            var elapsed = timeNow - lastTime;

            rTri += (90 * elapsed) / 1000.0;
        }
        lastTime = timeNow;
    }
</script>
```

Kao što je već rečeno, varijabla *rTri* je jedina koja se ažurira, dok ostatak koda služi čisto radi osiguravanja jedinstvenosti brzine rotiranja, bez obzira na računalo na kojem se program izvodi.

Samo iscrtavanje izvodi metoda *drawScene()*:

```
<script>
    function drawScene() {
        ...
        mat4.translate(mvMatrix, [0, 0.0, -7.0]);
        mat4.rotate(mvMatrix, degToRad(rTri), [0, 1, 0]);
        gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
        gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
            triangleVertexPositionBuffer.itemSize, gl.FLOAT, false, 0, 0);
        ...
        setMatrixUniforms();
        gl.drawArrays(gl.TRIANGLES, 0,
            triangleVertexPositionBuffer.numItems);
    }
</script>
```

Ono što je najvažnije u ovoj metodi je da se korištenjem matrice modela objekt translata i rotira na odgovarajući način, a zatim se iskoriste programi za sjenčanje za generiranje slike, te konačno se poziva samo iscrtavanje.

Naravno, prikazani programski kod nije kompletan da bi ovaj jednostavan primjer radio, no prikazuje princip na kojem WebGL počiva. Zadatak ovog rada je generirati teren i vegetaciju pomoću fraktala, a to iziskuje kompleksne proračune i iscrtavanja, što ovakav jednostavan program ne bi mogao izvesti. On niti neće biti dalje razvijan i nadograđivan već će se iskoristiti postojeća biblioteka koja ima implementirane mnoge metode, sve u svrhu olakšavanja izrade kompleksnijih zadataka.

4.2. Three.js

Three.js je Javascript biblioteka/API, neovisna o pregledniku, koja se koristi za kreiranje i animaciju 3D grafike. Kao i WebGL, Three.js ne zahtijeva instalaciju nikakvih dodatnih programa ili plug-inova.

Biblioteku je 2010. godine izdao glavni autor Ricardo Cabello. Tokom godina niz se drugih autora pridružio projektu, od kojih treba izdvojiti sljedeće: Paul Brunt, Branislav Ulicny i Joshua Koo.

Pomoću Three.js biblioteke lako se definiraju i kontroliraju mnogi aspekti 3D grafike:

- kamere (perspektivna i ortogonalna projekcija)
- osvjetljenje (ambijentalno, usmjereno, reflektor) i sjene
- različite vrste materijala, teksture i sl.
- geometrija - Three.js ima predefiniran veliki skup geometrijskih objekata (kugla, kocka, valjak itd.)
- objekti - "mesh", "particles", "sprites" itd.

Three.js također ima mogućnost uvoza i izvoza objekata u JSON formatu, što znači da se 3D model može napraviti u nekom specijaliziranom programu za 3D modeliranje (Blender, OpenCTM, FBX itd.) te učitati u Three.js program.

Jednostavan Three.js program izgleda ovako:

```
<script>
  renderer = new THREE.WebGLRenderer();
  renderer.setSize(SCREEN_WIDTH, SCREEN_HEIGHT);
  scene = new THREE.Scene();

  container = document.getElementById('webgl-container');
  container.appendChild( renderer.domElement );

  var obj = new Object();
  obj.position.z = -7;
  scene.add(obj);
  ...
</script>
```

Učita se iscrtavač (engl. *renderer*) i scena, od kojih se prvi pridruži željenom HTML elementu. Konačno, bilo koji objekt koji se želi dodati u program, bila to kamera, svjetlo, geometrija itd., se dodaje kroz naredbu *scene.add()*.

Objekte se uređuje kroz njihova svojstva (u primjeru: *obj.position.z* određuje njegovu poziciju na osi Z).

Struktura programa svakoga od primjera koji će biti prikazani u sljedećem poglavlju je sljedeća:

```
<script>
  /**
    definicija globalnih varijabli
  */

  init();
  animate();
</script>
```

Globalne varijable definiraju se iz razloga da možemo uređivati objekte iz bilo koje funkcije. Kamera će se, primjerice, na scenu dodati u metodi *init()*, no u simulaciji leta biti će potrebno mijenjati poziciju kamere, što će se raditi u drugoj metodi. Varijabla koja predstavlja objekt kamere, iz tog razloga, mora biti definirana kao globalna varijabla.

Metoda *init()* se pokreće samo jedamput. Dio njenih naredbi je prikazan u prethodnom programskom isječku. Osim učitavanja WebGL, kamere, svjetla i slično, u toj metodi će se i generirati primjer fraktala.

Metoda *animate()* slična je istoimenoj metodi iz primjera gdje se objašnjava WebGL bez upotrebe Three.js, a izgleda ovako:

```
<script>
  function animate()
  {
    requestAnimationFrame( animate );
    renderer.render( scene, camera );
    update();
  }
</script>
```

Kao i prije, za osiguravanje animacije koristi se metoda *requestAnimationFrame()*.

Naredba

```
renderer.render( scene, camera );
```

iscrtava scenu, dok se unutar metode *update()* mogu modificirati varijable koje na neki način utječu na scenu. S obzirom da mi generiramo teren i vegetaciju koja je po prirodi statična, unutar te metode mijenjat će se jedino pozicija kamere.

Navedene naredbe zajedničke su svim primjerima koji će biti prikazani u sljedećem poglavlju, te će se prikazati samo specifični programski isječki za svaki primjer.

5. Fraktalni teren i vegetacija

U ovom poglavlju će biti opisani različiti objekti koje susrećemo u prirodi i na koji način se oni mogu opisati pomoću fraktala. Naglasak je na implementaciji u WebGL-u odnosno Three.js. Bit će opisan:

- teren,
- obala, odnosno obalna linija,
- drvo,
- paprat.

Prije detaljnog objašnjenja implementacije svakog od ovih primjera potrebno je navesti nekoliko principa na kojima oni svi počivaju.

Rekurzija

Svaki od ovih objekata ima mnogo verzija sebe unutar samog sebe (samosličnost). Drvo ima grančice koje su manji oblik drva, teren ima kamenje i litice, koje su zapravo manje verzije cijele konfiguracije terena. Taj oblik ponavljanja u programskom kontekstu se zove rekurzija. Rekursivna metoda je ona koja poziva samu sebe. U prirodi ponavljanja mogu biti naizgled beskonačna i, kao što je već rečeno, bez obzira na razinu uvećanja, postoje mnogi detalji. Programski to, nažalost, nije tako lako izvedivo. Rekurzije su iznimno zahtjevne za računalo, i s obzirom da treba voditi računa o optimizaciji, rekurzija mora biti limitirana. To u konačnici znači da postoji razina uvećanja gdje se počinje gubiti na detaljima.

Skaliranje

Kada bi se samo redalo drvo na drvo, ili neki novi vrh iste visine u planinsku liniju, rezultat ne bi bio zadovoljavajući. Ideja rekurzije je dobra, no ona poziva istu metodu unutar sebe, a grančica ipak nije ista kao i drvo. Možda i je oblikom, no sigurno nije veličinom. Svaka grančica je manja nego ona roditeljska, i svaki sljedeći dodijeljeni vrh u planinskoj liniji je manje amplitude nego onaj dodijeljen u nadređenoj razini rekurzije. Programerski gledano, skaliranje će se riješiti prosljeđivanjem atributa, pri čemu će se oni skalirati određenim faktorom.

Slučajnost

Svaka sljedeća grana je manja od one iz nadređene razine rekurzije, no rezultat i dalje nije sličan kakav bi bio u prirodi. Grančica je posve identična verzija drva, samo skalirana, a to ipak nije slučaj u stvarnosti. Priroda je nepredvidiva i slučajna, što i mi trebamo simulirati u programu. Kada se prosljeđuje atribut koji primjerice označava dužinu, onda ga treba skalirati ali i pomnožiti s nekom slučajnom vrijednošću. Generalni oblik će ostati isti, ali opet različiti, a ukupni dojam će konačno sličiti onom iz stvarnosti.

Svaki od primjera koji će biti prikazani koristi navedene principe, pa će algoritmi za njihovo generiranje upravo kroz njih biti objašnjeni.

Iznimno je bitno napomenuti da programski isječci koji će biti prikazani u svim sljedećim primjerima nisu dovoljni za funkcioniranje programa. Oni samo prikazuju najvažnije stvari na koje je potrebno obratiti pažnju. Cjelokupni programski kod, odnosno aplikacija koju je moguće pokrenuti priložen je uz rad.

5.1. Teren

Ako pogledamo konačni zadatak ovog rada, a to je izrada simulacije letenja nad fraktalnim terenom posađenim fraktalnom vegetacijom, onda je teren vjerojatno najvažniji dio, s obzirom da će na pogled iz neba vegetacija biti prilično mala. Teren je manje očiti fraktal od vegetacije i njegova uvjerljivost u simulaciji najviše se bazira na slučajnosti.

Od prethodno opisanih fraktala, teren u prirodi najviše odgovara plazma fraktalima, ali može se napraviti i pomoću baza-motiv fraktala, posebno ako liniju zamijenimo trokutom, gdje bi treća točka bila neka slučajno generirana točka u rasponu ovisnom u stupnju rekurzije.

Ipak, najpoznatiji način generiranja terena je pomoću Perlinovog šuma.

5.1.1. Perlinov šum

Perlinov šum (engl. *noise*) je matematička funkcija koju je osmislio Ken Perlin 1983.godine.

Šum se može koristiti za kreiranje puno različitih tekstura prirodnog izgleda, a u kombinaciji s različitim matematičkim izrazima postaje proceduralna tekstura. Ovaj tip teksture ne zahtijeva izvornu sliku teksture, pa su za prijenos ili spremanje proceduralnih tekstura zahtjevi na mrežu minimalni. Također, može se primijeniti direktno na 3D objekt. (Perlin, 1999 a).

Perlinov šum je, dakle, funkcija za kreiranje dosljednog (engl.*coherent*) šuma preko određene površine/prostora. Dosljedni šum znači da za bilo koje dvije točke u prostoru se vrijednost funkcije šuma mijenja glatko (tokom kretanja od jedne do druge točke), odnosno nema nepovezanosti. Funkcija u ovom kontekstu radi sljedeće: uzme koordinatu u nekom prostoru i poveže je s realnim brojem u intervalu od -1 do 1. (Zucker, 2001).

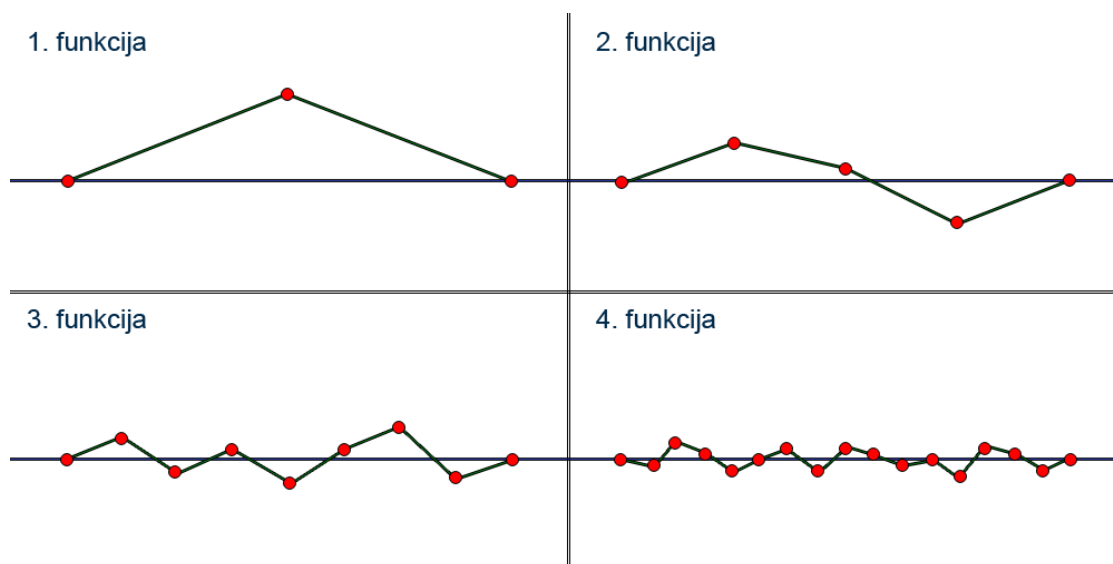
Koristio se u raznim filmovima, a jedan mali dio filmova u kojima je korišten je prikazan na slici 43.



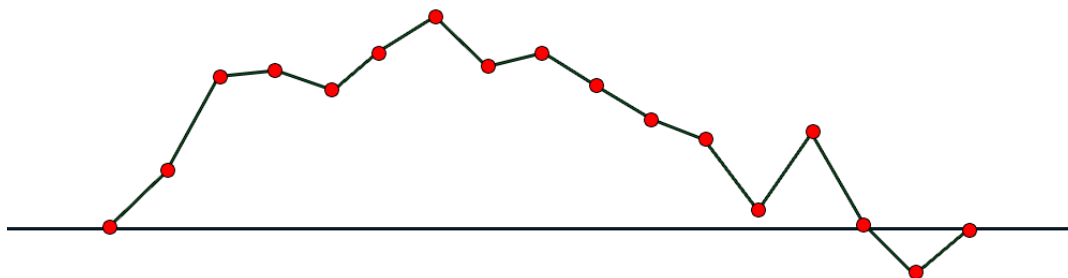
Slika 43. Filmovi u kojima je korišten Perlinov šum (s lijeva na desno, prema godini izlaska: *Terminator 2*, *Kralj lavova*, *Titanik* i *Zvjezdani ratovi: Epizoda I*) (Prema: IMDB, s.a.)

Pri svakom ponavljanju amplituda (raspon na osi Y) je dvostruko manja, dok je frekvencija dvostruko veća. Perlinov šum ne zahtijeva nužno korištenje rekurzivnih metoda kako bi se implementirao, već je dovoljna obična iteracija. Ovisno o količini detalja (n) generira se n funkcija, a svaka sljedeća funkcija generira duplo više slučajnih točaka u duplo manjoj amplitudi nego prethodna.

Na slici 44 su prikazane 4 funkcije koje će se iskoristiti u dobivanju konačnog rezultata. Konačan rezultat je zbroj svih funkcija. Na slici 45 je prikazan konačan rezultat.



Slika 44. Funkcije Perlinovog šuma



Slika 45. Konačan rezultat Perlinovog šuma

Najbolja alternativa Perlinovom šumu je algoritam kojeg je izmislio isti autor, Ken Perlin kao nadogradnja originalnog algoritma, a radi se o **simplex šumu**.

Simplex šum je zapravo mnogo efikasniji način da se postigne isti efekt. Prednosti ovog algoritma su sljedeće (Gustavson, 2005):

- a) kompleksnost je $O(N^2)$ za N dimenzija (umjesto $O(2^N)$),
- b) jednostavno se implementira u hardver.

5.1.2. Implementacija

Ideja je sljedeća: generirati n funkcija te zbrojiti ih u konačno rješenje. Visinska vrijednost (os Y) bit će pohranjena u dvodimenzionalnom polju ($polje[osX][osZ] = osY$), stoga je prvi korak inicijalizacija tog polja (varijabla *main_array*), te nakon toga slijedi petlja koja će generirati točke za n funkcija i spremiti ih u to polje.

Sve navedeno, u kodu, izgleda ovako:

```
<script>
  var length = 2000;
  var main_amplitude = 1200;
  var max_lvl = 8;

  init_main_array(max_lvl, max_lvl, length);
  for(var lvl = 1; lvl <= max_lvl; lvl++) {
    generate_points(lvl, max_lvl, lvl,
main_amplitude*Math.pow(0.5, lvl), 0, length, 0, length);
  }
</script>
```

Za jednu funkciju je dovoljno dvodimenzionalno polje, no kako imamo n različitih funkcija, metoda *init_main_array()* inicijalizira trodimenzionalno polje koje bi onda izgledalo ovako: $polje[funkcija][osX][osZ] = osY$.

Metoda *generate_points()* je rekurzivna metoda koja poziva samu sebe, i unutar prikazane *for* petlje njen je prvi poziv. Ta metoda generira točke za jednu funkciju, a kako je ideja Perlinovog šuma generiranje više različitih funkcija, taj se postupak mora ponoviti više puta kroz iteraciju.

Važne varijable su *length* koja označava raspon na osi X i osi Z (bitna za određivanje frekvencije), *main_amplitude* koja označava raspon unutar kojeg će se generirati slučajna točka na osi Y i *max_lvl* koja označava broj funkcija koje će se generirati, odnosno n . Unutar *for* petlje koja će se ponoviti puta *max_lvl*, mijenja se varijabla *lvl* odnosno indeks svake od funkcija.

Definicija metode *generate_points()* izgleda ovako:

```
<script>
  function generate_points(perm_lvl, max_lvl, lvl, amplitude, x1, x2,
z1, z2) {...}
</script>
```

Argumenti koje metoda prihvća su *perm_lvl*, odnosno indeks trenutne funkcije koji se neće mijenjati prilikom rekurzije, *max_lvl* odnosno ukupni broj funkcija koje će se generirati, i koja će se umanjivati za jedan pri svakom sljedećem rekurzivnom pozivu, te varijabla *lvl* koja kao i *perm_lvl* označava indeks trenutne funkcije. Za razliku od varijable *perm_lvl*, varijabla *lvl* se umanjuje za jedan pri svakom sljedećem rekurzivnom pozivu. Zašto se neke varijable umanjuju na taj način bit će naknadno objašnjeno. Varijabla *amplitude* označava raspon unutar kojeg će se generirati vrijednost na osi Y, dok *x1*, *x2*, *z1* i *z2* označavaju raspon na osi X i Z unutar kojeg će se generirati nove točke.

Sada kada znamo što znače varijable koje metoda *generate_points()* prima, možemo objasniti koje vrijednosti se šalju unutar *for* petlje pri pozivu te metode.

Varijable *perm_lvl* i *lvl* metode *generate_points()* dobivaju vrijednost *lvl* varijable iz *for* petlje koja se mijenja kroz iteraciju i označava indeks funkcije za koju će se generirati točke.

Varijabla *max_lvl* označava ukupan broj funkcija koje će se generirati.

Varijabla *amplitude* preuzima: *main_aplitude * Math.pow(0.5, lvl)*, što odgovara ideji Perlinovog šuma da svaka sljedeća funkcija ima dvostruko manju amplitudu od prethodne. Varijable *x1*, *x2*, *z1* i *z2* (od 0 do maksimalne vrijednosti, odnosno vrijednosti varijable *length*) su iste za svaku funkciju, jer se svaka funkcija generira u istom rasponu na osi X i Z, a jedina razlika je u frekvenciji.

Metoda *generate_points()* izgleda ovako:

```
<script>
    function generate_points(perm_lvl, max_lvl, lvl, amplitude, x1, x2,
y1, y2){
        var xm = (x1+x2)/2;
        var zm = (z1+z2)/2;

        if(lvl>0){
            /**
                generate points
            */
        }else{
            /**
                calculate points
            */
        }
        if(max_lvl > 1){
            generate_points(perm_lvl, max_lvl-1, lvl-1, amplitude, x1, xm,
z1, zm);
            generate_points(perm_lvl, max_lvl-1, lvl-1, amplitude, xm, x2,
z1, zm);
            generate_points(perm_lvl, max_lvl-1, lvl-1, amplitude, x1, xm,
zm, z2);
            generate_points(per_lvlm, max_lvl-1, lvl-1, amplitude, xm, x2,
zm, z2);
        }
    }
</script>
```

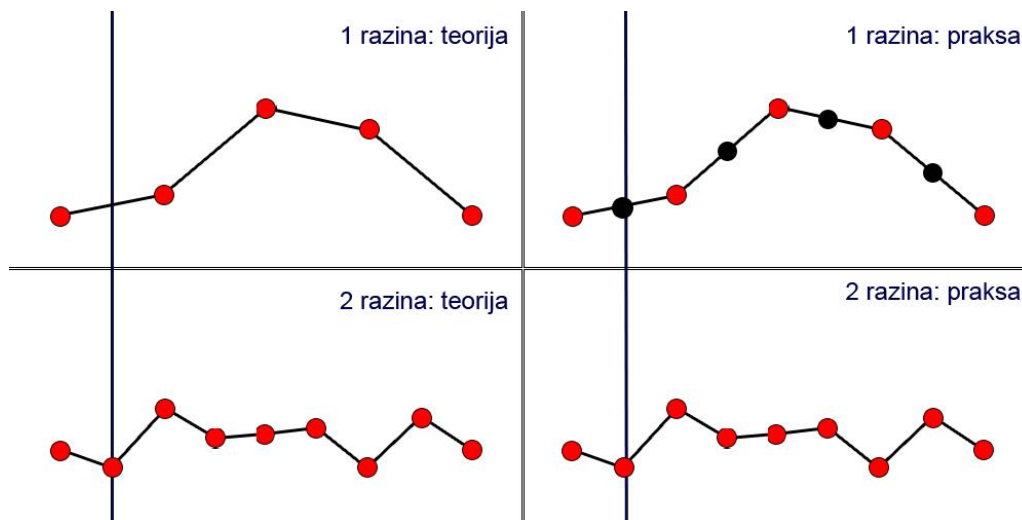
Najprije se definiraju varijable *xm* i *zm* čija je vrijednost aritmetička sredina od *x1* i *x2*, odnosno *z1* i *z2*. Svaki rekursivni poziv generira jednu točku, a to je ona u sredini danog raspona, dakle točka (*xm*, *zm*).

Prvi problem koji se pojavljuje prilikom implementacije je sljedeći: svaka sljedeća funkcija treba imati duplo veću frekvenciju, odnosno generirati duplo više točaka od prethodne. No, ako pogledamo naredbu:

```
if(max_lvl > 1){...}
```

možemo zaključiti da će se za svaku funkciju Perlinovog šuma napraviti isti broj točaka, jer varijabla *max_lvl* označava ukupan broj funkcija, a ne indeks trenutne. Razlog je taj što kada bi nekoliko koraka unaprijed morali zbrajati vrijednosti svih funkcija, ne bi mogli zbrojiti vrijednosti koje ne postoje, ali su ipak jasne iz grafičkog prikaza (npr. na slici 46).

Dakle, trebali bi generirati određeni broj točaka u skladu s trenutnim indeksom, ali i izračunati sve ostale međuvrijednosti na najvećoj mogućoj frekvenciji.



Slika 46. Rupe u polju koje nastaju na funkcijama različitih frekvencija

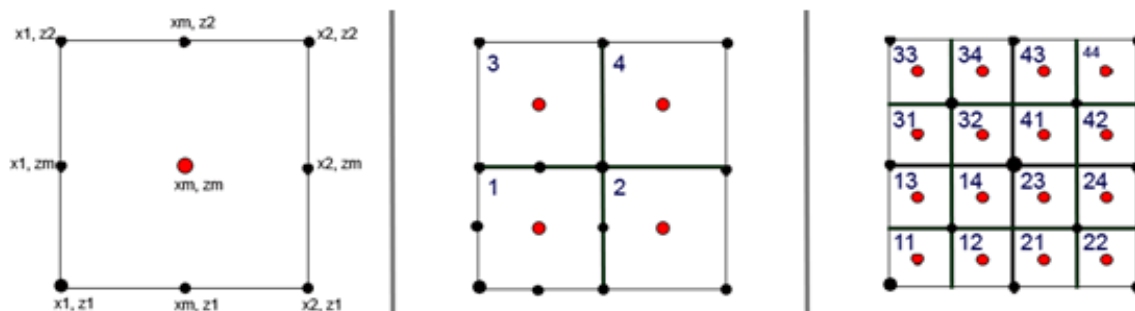
Kako vrijednosti spremamo u polje, pri zbrajanju vrijednosti svih funkcija ne bi mogli zbrojiti vrijednosti na točki označenoj plavom crtom na slici 46 (lijevi stupac: teorija) jer ona za razinu 1 ne postoji. Umjesto toga tu točku trebamo izračunati kao međuvrijednost dvije generirane točke. Na taj način moguće je zbrojiti vrijednosti dvije različite funkcije jer za istu točku (indeks u Javascript polju) postoji vrijednost.

Iz tog razloga imamo odvojeno varijable *lvl* i *max_lvl* unutar metode *generate_points()*. Rekurzija će se odvijati za svaku funkciju maksimalan broj puta što je određeno naredbom:

```
if (max_lvl > 1) {...}
```

ali isto tako točke će se generirati smo *lvl* broj puta, odnosno broj puta jednak indeksu trenutne funkcije. Kako se varijabla *lvl* smanjuje pri svakom sljedećem rekurzivnom pozivu, a ta ista varijabla je obavezno manja ili jednaka varijabli *max_lvl*, sigurno je da će u jednom trenutku varijabla *lvl* biti manja od nule (osim pri generiranju posljednje funkcije) i tada će se početi računati međutočke umjesto generiranja novih slučajnih točaka.

Kako točno izgleda dio generiranja i računanja točaka bit će prikazano malo kasnije, prvo je potrebno pogledati na koji način se rekurzivno pozivaju metode.



Slika 47. Generiranje novih točaka u XZ ravni

Metoda poziva četiri nove verzije na način kako je to prikazano na prethodnoj slici 47. Verzije se međusobno razlikuju u $x1$, $x2$, $z1$ i $z2$ varijablama. Četiri rekurzivna poziva dijele prostor na četiri dijela, koji se opet dijele na četiri dijela, sve dok se ne dostigne maksimalna frekvencija.

Nakon prve podjele, točke po dijelovima su dodijeljene ovako (dvije točke predstavljaju donji lijevi i gornji desni vrh pravokutnika):

- 1. dio: $(x1, z1)$, (xm, zm)
- 2. dio: $(xm, z1)$, $(x2, zm)$
- 3. dio: $(x1, zm)$, $(xm, z2)$
- 4. dio: (xm, zm) , $(x2, z2)$.

Na slici 47 crvene točke se generiraju, crne točke su izgenerirane točke iz prethodne razine rekurzije, dok su presjeci linija - točke koje se moraju izračunati na osnovu susjednih točaka. Konačno, generiranje i računanje točaka izgleda ovako:

```
<script>
    if(lvl>0){
        main_array[perm_lvl][xm][zm]['val'] = Math.random()*amplitude -
(amplitude/2);

        if(!main_array[perm_lvl][x1][zm]['set']){
            if(x1!=0) main_array[perm_lvl][x1][zm]['val'] =
Math.random()*amplitude - (amplitude/2);
            main_array[perm_lvl][x1][zm]['set'] = true;
        }
        ...

    }else{
        main_array[perm_lvl][x1][zm]['val'] =
determine_mid_point(ym,y1,y2,main_array[perm_lvl][x1][z1]['val'],
main_array[perm_lvl][x1][z2]['val']);
        ...

        main_array[perm_lvl][xm][zm]['val'] =
determine_mid_point(zm,z1,z2,main_array[perm_lvl][xm][z1]['val'],
main_array[perm_lvl][xm][z2]['val']);
    }
</script>
```

Metoda *determine_mid_point()* jednostavna je matematička funkcija koji traži srednju točku između dvije u ravnini.

Varijabla *perm_lvl* je ona koja se ne mijenja prilikom rekurzivnih poziva i zapravo samo označava indeks u Javascript polju trenutne funkcije.

Ovaj isječak je logički podijeljen na dva djela *if* naredbom, kako je već objašnjeno. Jedan dio se izvršava u fazi dok se generiraju točke neke funkcije, a drugi dio se izvršava dok se računaju međutočke do najviše moguće frekvencije radi logičke konzistencije, koja je nužna za računanje zbroja svih funkcija.

Generira se samo jedna slučajna točka u rasponu određenom amplitudom i to je točka (x_m, z_m) .

Točke (x_1, z_1) , (x_1, z_2) , (x_2, z_1) i (x_2, z_2) su vrhovi, i njihova vrijednost na osi Y je već određena u nadređenim rekurzijama ili imaju početnu vrijednost 0, no i dalje ostaje za odrediti vrijednost točaka poput (x_m, z_1) i sl. Njihova vrijednost se određuje na način da se izračuna međuvrijednost dvije susjedne točke. U primjeru s točkom (x_1, z_m) to bi bile npr. (x_1, z_1) i (x_1, z_2) .

U drugom dijelu koda izračunavaju se međuvrijednosti svih točaka, uključujući i središnje (x_m, z_m) točke.

Jedino što treba objasniti je ovaj dio:

```
if(!main_array[perm_lvl][x1][zm]['set']){...}
```

Razlog zašto se provjerava da li je vrijednost postavljena je sljedeći. Naime, na slici 47 se može vidjeti da je u 2. razini (središnji dio) točka (x_2, z_m) sekcije 1 zapravo ista točka kao i točka (x_1, z_m) sekcije 2. Na taj način neke točke bi se izračunavale više puta. Također te su točke uvijek one kojima se izračunava međuvrijednost na osi Y, pa bi zato dolazilo do anomalija.

Rezultat funkcije *generate_points()* koja se poziva kroz *for* petlju iz samoga početka je sljedeći. Imamo generirane točke za n funkcija u skladu s idejom Perlinovog šuma. Svaka funkcija ima isti broj ukupnih točaka (maksimalna frekvencija) no svaka sljedeća funkcija ima duplo više slučajno generiranih točaka u duplo manjem rasponu, također u skladu s idejom Perlinovog šuma. Ono što preostaje je zbrojiti vrijednosti svih funkcija po točkama na osi X i Z, te konačno generirati teren na osnovi dobivenog rezultata.

```
<script>
var plane_geo = new THREE.PlaneGeometry(length, length,
Math.pow(2,max_lvl), Math.pow(2,max_lvl));

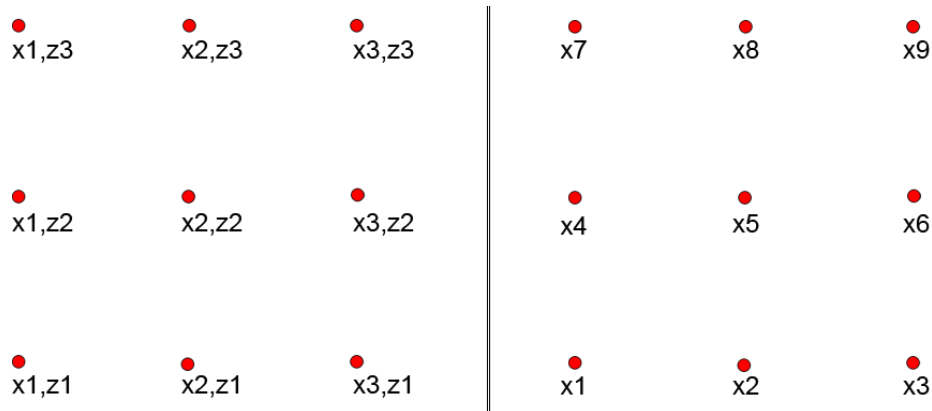
var step = length/Math.pow(2,max_lvl);
var counter = 0;
for(var i = 0; i <= length; i += step){
  for(var j = 0; j <= length; j += step){
    var total = 0;
    for(var k=1; k<=max_lvl; k++){
      total+= main_array[k][i][j]['val'];
    }
    plane_geo.vertices[counter].y = total;
    counter++;
  }
}
</script>
```

Prvi korak je kreiranje ravnine dimenzija $length*length$, te s 2^{max_lvl} točaka što odgovara maksimalnoj frekvenciji. Na sličan način se određuje i varijabla *step* koja zapravo i predstavlja frekvenciju.

Slijedi dvostruka *for* petlja koja će prolaziti po svim točkama na osi X (varijabla *i*) i Z (varijabla *j*). Unutar tih petlji inicijalizira se varijabla *total* s početnom vrijednosti 0. Ulazi se u treću petlju, u kojoj varijabla *k* predstavlja indeks funkcije, i kojoj se na vrijednost varijable *total* dodaje visinska vrijednost (osY) točke (*i*, *j*). Nakon što se treća petlja izvrši, varijabla *total* sadrži konačnu visinsku vrijednost za neku točku.

Ta vrijednost se pridružuje jednoj od točaka ravnine koju smo prethodno kreirali.

Jedina razlika je u strukturi polja u kojoj smo mi spremali vrijednosti, i kakvu kreira Three.js za geometrijski objekt. Naime, mi smo koristili dvodimenzionalno polje, dok *vertices* objekta *plane_geo* je jednodimenzionalno polje kako je prikazano na sljedećoj slici.



Slika 48. Razlika u koordinata generiranog Javascript polja i Three.js geometrije

It tog razloga koristimo varijablu *counter* kao sredstvo kojim preslikavamo koordinate iz jednog oblika u drugi.

Ravnina sada ima visinske vrijednosti za sve njene točke i prije dodavanja na scenu moramo napraviti još par modifikacija prikazanih na sljedećem isječku:

```
<script>
    plane_geo.computeFaceNormals();
    plane_geo.computeVertexNormals();
    plane_geo.__dirtyVertices = true;
    plane_geo.__dirtyNormals = true;

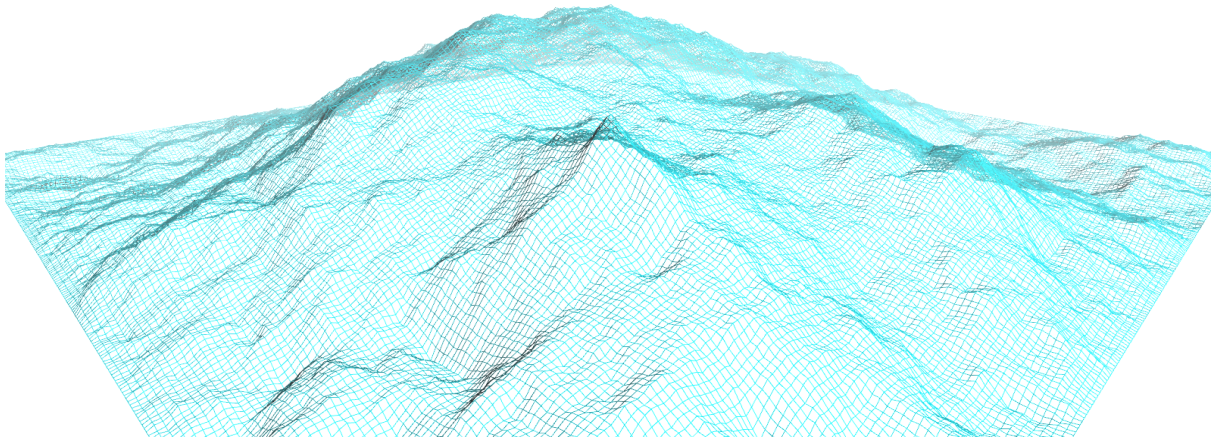
    var plane = new THREE.Mesh(plane_geo, new THREE.MeshLambertMaterial
    ({options}));
    plane.rotation.x = 270 * Math.PI / 180;

    scene.add(plane);
</script>
```

Kada smo dodijelili visinsku vrijednost svim točkama u ravnini, promijenili smo poziciju točke na osi Y. Kako bi se to pravilno isctalo potrebno je izračunati normale, odnosno kut pod kojim će se svjetlost odbijati od dijela ravnine koje čine 3 ili 4 točke (lice ili *face*). Three.js ima ugrađene funkcije koje to računaju pa tako prvi set naredbi u prikazanom isječku služi upravo tome.

Varijabla *plane_geo* tek je geometrijski objekt i njemu se mora pridružiti nekakav materijal. "{options}" određuje kakav će taj materijal biti (boja, tekstura) i slično.

Konačno, objekt se mora rotirati po osi X jer Three.js općenito isctava ravninu u X i Y ravnini, umjesto X i Z kako nama treba, te se objekt dodaje u scenu. Rezultat je prikazan na sljedeće tri slike, jedna pomoću žičanog modela (engl. *wireframe*) gdje se može vidjeti raspored svih točaka, jedna koristeći osnovnu boju i zadnja koristeći teksturu za realniji prikaz.



Slika 49. Generirani teren (žičani model)



Slika 50. Generirani teren (osnovna boja)



Slika 51. Generirani teren (tekstura)

5.2. Obalna linija

Obalna linija je oblik fraktala vrlo sličan terenu, uz razliku da je obalna linija striktno dvodimenzionalan fraktal. Kako su oblikom slični, tako su slični i načini implementacije. Obalnu liniju možemo dakle generirati pomoću:

- baza-motiv fraktala,
- 2D Perlinov šum,
- presjek 3D Perlinovog šuma (teren).

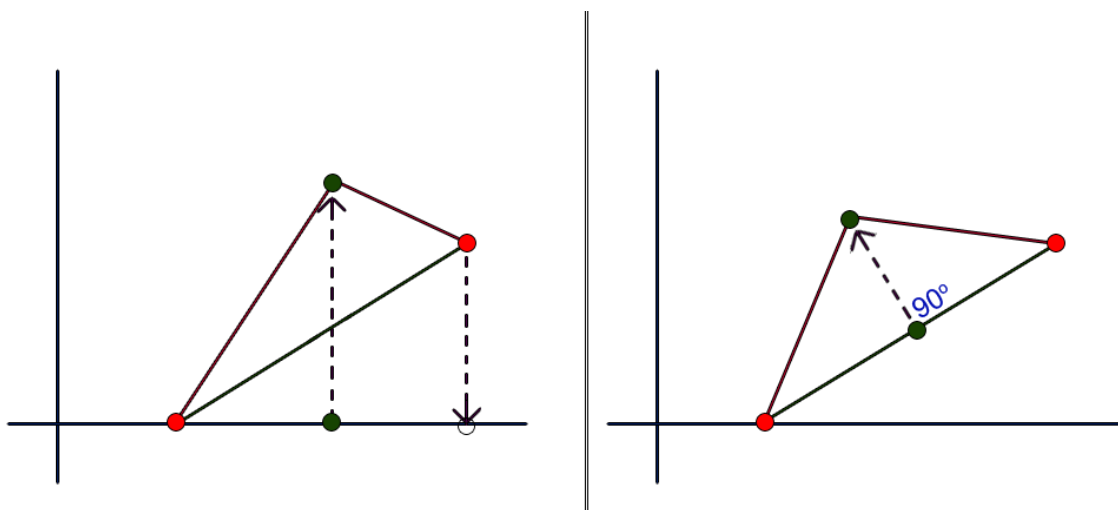
Perlinov šum je objašnjen na primjeru drva (podpoglavlje 5.3), pa se ovdje nećemo njime baviti. Umjesto toga obalnu liniju ćemo najprije implementirati pomoću baza-motiv fraktala:

```
<script>
  var x1 = 0;
  var x2 = length;
  var y1 = Math.random()*main_amplitude - (main_amplitude/2);
  var y2 = Math.random()*main_amplitude - (main_amplitude/2);
  generate_points(max_lvl, main_amplitude, x1, x2, y1, y2);
</script>
```

Za razliku od terena, ovdje imamo samo jedan skup dvodimenzionalnih točaka (funkcija), pa ne treba pripremati posebno polje u kojima ćemo spremati podatke. Dapače, i generiranje i iscrtavanje točaka odvijat će se u glavnoj *generate_points()* rekurzivnoj metodi. Za početak definiramo dvije početne točke između kojih će se generirati obalna linija. Na osi X između 0 i maksimalne duljine, te na osi Y dvije slučajno generirane točke.

Baza-motiv fraktal već je objašnjen u 3. poglavlju, pa ćemo samo spomenuti općenitu ideju. Cilj je, dakle, pri svakom rekurzivnom pozivu na osnovi dvije proslijeđene točke generirati treću. Sve tri točke činile bi jednakokračni trokut, i na taj način dužina (baza) bi bila zamijenjena trokutom (motiv).

Pri tome imamo problem prikazan na sljedećoj slici:



Slika 52. Generiranje vrha trokuta

Naime, kod implementacije terena našli smo središte između dvije točke na osi X, i onda direktno dodijelili vrijednost na osi Y. Kada bi isto radili i sada, tri točke ne bi činile jednakokračni trokut kao što se može vidjeti na slici 52. Takav slučaj nije pogrešan već jednostavno neprikladan za ono što želimo postići. Umjesto toga moramo pronaći središte na dužini između dvije proslijeđene točke, pravac okomit na tu dužinu (normala) i točku generirati na pravcu određenom središnjom točkom i vektorom smjera (normala).

Prvi korak je pronalaženje središta $Tm(xm, ym)$ na dužini između dvije točke $T1(x1, y1)$ i $T2(x2, y2)$:

```
<script>
    var xm = (x1+x2)/2;
    var ym = (y1+y2)/2;
</script>
```

Sljedeći korak je određivanje normale na dužinu između $T1$ i $T2$:

```
<script>
    if(y2 - y1 == 0 || x2 - x1 == 0){
        if(x2 - x1 == 0){
            vx = 1;
            vy = 0;
        }
        if(y2 - y1 == 0){
            vx = 0;
            vy = 1;
        }
    }else{
        k1 = (y2-y1)/(x2-x1);
        k2 = -1/k1;

        vx = 1;
        vy = k2;

        var v_length = Math.sqrt(Math.pow(vx,2) + Math.pow(vy,2));
        vx = vx/v_length;
        vy = vy/v_length;
    }
</script>
```

U tu svrhu poslužit će nam koeficijent smjera pravca određenog dvjema točkama koji se izračunava po sljedećoj formuli:

$$k1 = \frac{y2 - y1}{x2 - x1}$$

dok pravac okomit na taj pravac ima koeficijent smjera:

$$k2 = -\frac{1}{k1}$$

Nedozvoljeno dijeljenje s nulom može se dogoditi ako će:

- $y2 - y1$ biti jednako nula, jer će se odraziti na formulu za izračunavanje koeficijenta smjera okomitog pravca $k2$ s obzirom da će $k1$ biti nula,

- $x_2 - x_1$ biti jednako 0 u formuli za izračunavanje koeficijenta smjera pravca određenim dvjema točkama.

Kako ne bi došlo do tih nedozvoljenih dijeljenja s nulom ti slučajevi su u kodu odvojeni *if* naredbom gdje se vektor smjera $v(v_x, v_y)$ direktno određuje kao $v(1, 0)$ ili $v(0, 1)$.

Za svaki drugi slučaj koeficijenti smjera se izračunavaju i na osnovu njih se dobiva vektor smjera. Naime ako je koeficijent smjera 2, to znači da se za svaki x (uvećavanje za 1), y povećava za 2. Iz toga možemo definirati vektor v kao $v(1, k_2)$.

Dobiveni vektor moramo svesti na jedinični vektor, odnosno vektor čiji je iznos ili duljina jednaka 1. Stoga izračunavamo duljinu vektora, te vektor dijelimo s dobivenom duljinom. Time smo dobili jedinični vektor okomit na dužinu određenu s dvije početne točke.

```
<script>
    var rand = Math.random()*amplitude - (amplitude/2);

    vx = vx*rand;
    vy = vy*rand;

    var fx = xm + vx;
    var fy = ym + vy;
</script>
```

Konačno, generiramo slučajnu vrijednost u rasponu amplitude, te tu vrijednost množimo s jediničnim vektorom, čime mu zapravo određujemo novu dužinu, eventualno orijentaciju, ako je generirana vrijednost manja od 0.

Dobiveni vektor zbrojimo sa središtem dužine, točkom T_m , i dobivamo točku $F(f_x, f_y)$, tj. treći vrh jednakokračnog trokuta.

Slijedi rekurzivno pozivanje metoda i iscrtavanje:

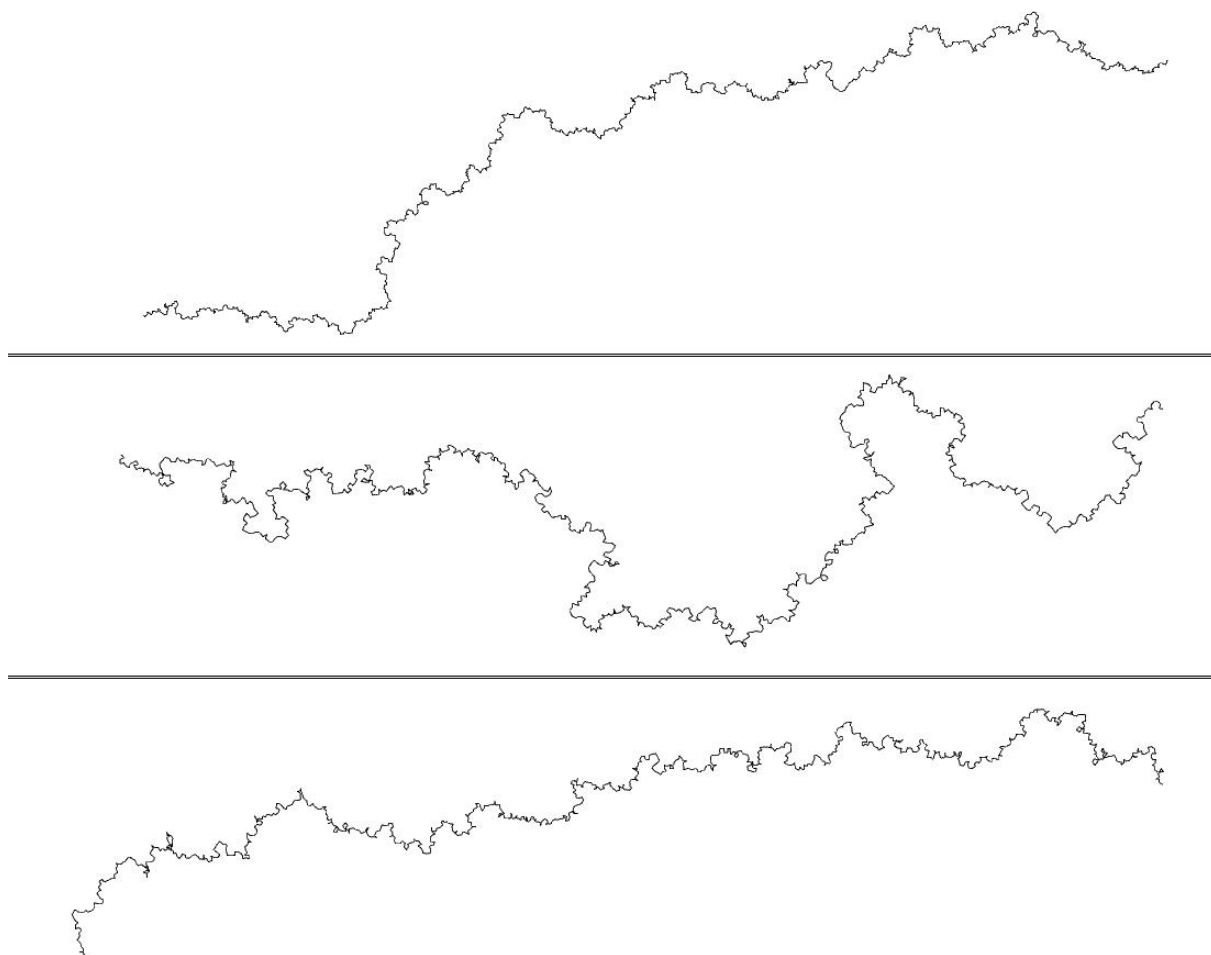
```
<script>
    if(lvl > 1){
        generate_points(lvl -1, amplitude/1.6, x1, fx, y1, fy);
        generate_points(lvl -1, amplitude/1.6, fx, x2, fy, y2);
    }else{
        var geometry = new THREE.Geometry();
        geometry.vertices.push(new THREE.Vector3(x1, y1, 0));
        geometry.vertices.push(new THREE.Vector3(fx, fy, 0));
        geometry.vertices.push(new THREE.Vector3(x2, y2, 0));

        var line = new THREE.Line(geometry, new THREE.MeshBasicMaterial
        ({ color: 0x000000}));
        scene.add(line);
    }
</script>
```

Ako je trenutni stupanj rekurzije (varijabla *lvl*) veći od 1, onda je potrebno generirati još točaka. Pozivamo dvije rekurzivne funkcije kojima ćemo proslijediti točke T_1 i F te F i T_2 . Također umanjujemo varijablu *lvl*, kako bi osigurali eventualni izlazak iz rekurzije, te skaliramo varijablu *amplitude*.

Dok smo kod terena amplitudu umanjili za duplo, ovdje radimo nešto manje ublažavanje upravo zato kako bi rezultat bio grublji, što nekako više priliči obalnoj liniji. Općenito, što blaže umanjujemo amplitudu, to je obalna linija razvedenija i obrnuto.

Kada smo došli do konačne razine rekurzije onda iscrtavamo liniju između 3 točke koje će činiti trokut, odnosno njegov "gornji" dio. Na konačnoj razini rekurzije postoji tisuće različitih poziva od kojih će svaki iscrtati svoj mali trokut, dio konačnog rezultata koji je prikazan na slici 53.



Slika 53. Obalne linije

Ipak, najzanimljiviji, ujedno i najefikasniji način za generiranje obalne linije je presjek 3D Perlinovog šuma. Naime, mi smo već kreirali obalnu liniju iako to onda nismo znali, ili barem nismo računali na to.

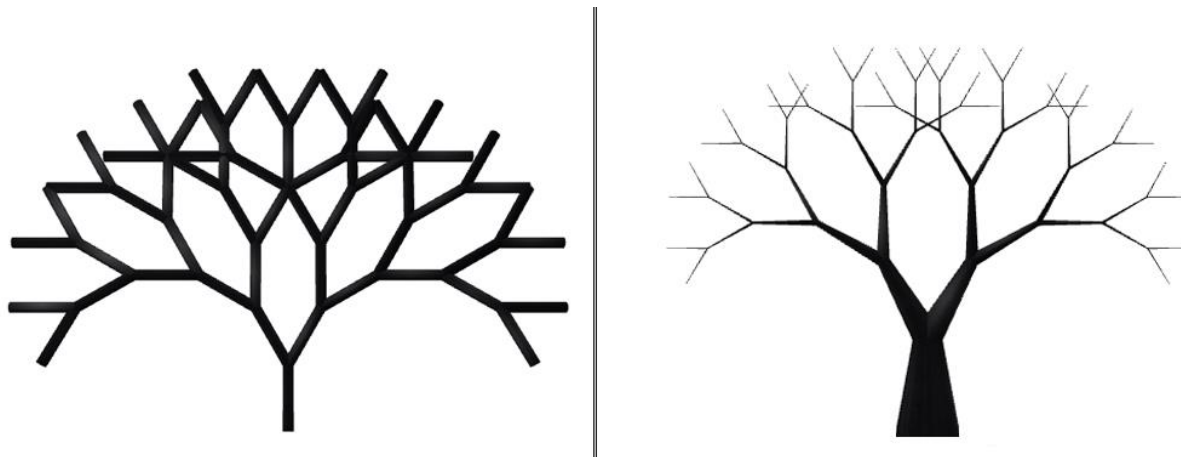
Uzmemo li ravninu i presječemo li 3D Perlinov šum, odnosno teren koji smo generirali u prethodnom podpoglavlju na nekoj visini (os Y), presjek te ravnine i terena dat će obalnu liniju. Ako nam je cilj napraviti 3D prikaz terena i obale, onda niti nema smisla posebno računati obalnu liniju, jer bi bilo prilično kompleksno generirati i iscrtati teren u rasponu obalne linije. Jednostavnije je generirati ravninu, koja bi predstavljala vodu, i pozicionirati je na osi Y negdje iznad najniže, a ispod najviše točke terena.

Implementacija je tako iznimno jednostavna, a bit će realizirana u 6. poglavlju kada će se svi fraktali kombinirati u jedan prikaz - fraktalni teren posađen fraktalnom vegetacijom.

5.3. Drvo

Po vrsti fraktala, drvo bi bilo fraktalni baldahin, odnosno Pitagorino stablo. Od svih primjera fraktala u prirodi, za njega je vjerojatno najlakše objasniti povezanost s fraktalima. Također, principi koje smo naveli na početku 5. poglavlja najočitiji su na primjeru drva.

Princip rekurzije je jasan, jer savršeno opisuje svojstvo samosličnosti, no sljedeća slika jasno ilustrira na primjeru drva kakav bi rezultat bio bez upotrebe principa skaliranja i slučajnosti.



Slika 54. Drvo bez skaliranja (lijevo) i slučajnosti (desno)

Kod lijevog primjera, pri svakom rekurzivnom pozivu, iscrtava se identični objekt različito transliran i rotiran, što je tehnički dozvoljeno no u suprotnosti s teorijom fraktala. Desni primjer je zapravo matematički savršen primjer, no neprirodan baš zato jer je savršen. Mi želimo simulirati prirodu pa koristimo slučajnost iako se time udaljavamo od matematičke definicije fraktala.

Algoritam koji će se iskoristiti za implementaciju drva je primjenjiv i za ostale oblike vegetacije. Također, u konačnoj kombinaciji svih primjera, teren i obala će se generirati samo jednom, dok će za sadenje terena vegetacijom biti potrebno višestruko izvođenje metoda za generiranje drva. Zato, generiranje i iscrtavanje drva mora biti brzo i efikasno, pa nakon opisa implementacije slijedi i opis korištenih optimizacija.

5.3.1. Implementacija

Implementacija drva počinje sljedećim programskim isječkom:

```
<script>
    group = new THREE.Object3D();
    tree(max_lvl, branch_length, branch_width, 0, 0, group,
branch_length_factor);
    scene.add( group );
</script>
```

Varijabla *group* je varijabla koja predstavlja skup grana, odnosno drvo. Pri svakom rekurzivnom pozivu u grupu će se dodati jedna grana.

Definicija metode *tree()* izgleda ovako:

```
<script>
    function tree(lvl, length, width, ry, rz, parent, counter){...}
</script>
```

Varijabla *lvl* označava trenutni stupanj rekurzije. Naravno, pri svakom sljedećem rekurzivnom pozivu ona se umanjuje za 1. Kao što je vidljivo u prethodnom programskom isječku, pri prvom pozivu prosljeđuje se vrijednost varijable *max_lvl*, odnosno maksimalna razina rekurzije. Veća razina rekurzije generirat će više grana i tako napraviti drvo s više detalja. Varijable *length* i *width* označava duljinu odnosno širinu grane, ili bolje rečeno nadređene grane. Varijable *ry* i *rz* označavaju rotaciju oko osi Y i Z te zajedno usmjeruju granu. Varijabla *parent* je grana koja je nadređena trenutnoj grani. Svaka grana ima granu iz koje se ona izdvojila, osim debla, kojeg zapravo predstavlja varijabla *group* koja se prosljeđuje u prvom pozivu. Ta varijabla služi nam kako ne bi morali rotirati i translirati svaku granu apsolutno, već relativno u odnosu na nadređenu granu. Varijablu *counter* koristimo kako bi skalirali grane. Njena vrijednost se uvećava pri svakom rekurzivnom pozivu, a služi tome kako bi sve grane na nekoj razini rekurzije bile približno iste duljine. Ovakav način nije jedini logički ispravan nego jednostavno sredstvo da dobijemo drvo željenog oblika.

```
<script>
    var height = length;
    length = length*((1/counter)+(1-
(1/branch_length_factor)))*rand_length();

    var branchGeom = new
THREE.CylinderGeometry(width*branch_width_factor, width, length, 20);
    branchGeom.applyMatrix( new THREE.Matrix4().makeTranslation( 0,
length/2, 0 ) );

    var branchMaterial = new THREE.MeshLambertMaterial( { map:
branchTexture } );
    var branch = new THREE.Mesh( branchGeom, branchMaterial );
</script>
```

Prvi isječak koda metode *tree()* prikazuje kreiranje grane. Najprije vrijednost varijable *length* (duljina grane) pridružujemo varijabli *height*, te onda skaliramo varijablu *length* za faktor ovisan o brojaču i slučajnom broju (metoda *rand_length()*). Faktor smanjivanja grane za koji sam se odlučio je prikazan na slici 55 a zapravo predstavlja funkciju:

$$y = \frac{1}{x} + 1 - \left(\frac{1}{factor} \right),$$

gdje *factor* predstavlja vrijednost na osi X od koje će se početi izračunavati ukupan faktor skaliranja. Sama priroda bazne funkcije:

$$y = \frac{1}{x}$$

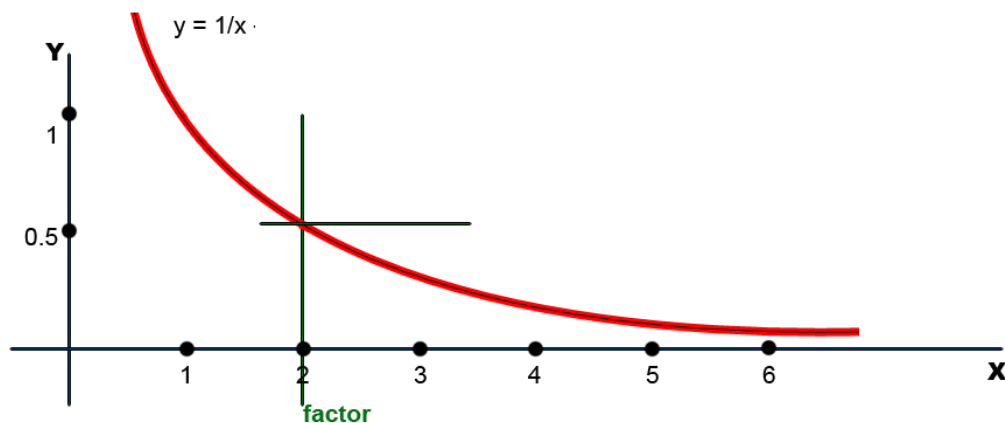
je takva da veća vrijednost na osi X znači manji pad, tj. ukupno teži nuli, ali je nikada ne dostigne. Zato služi faktor koji definira kakav pad, odnosno skaliranje grane, želimo.

Mali faktor (vrijednost na osi X) znači da će svaka grana biti dosta manja od prethodne, dok suprotno veći faktor bi značio da bi grane bilo skoro iste veličine.

Korigirajući dio funkcije:

$$+ 1 - \left(\frac{1}{factor} \right)$$

zapravo samo osigurava da bez obzira na kojem djelu osi X želimo početi s funkcijom (faktor), prva vrijednost uvijek bude što bliža, ali uvijek manja od 1.



Slika 55. Grafički prikaz funkcije za skaliranje dužine grane

Razlog iz kojeg pridružujemo varijabli *height* vrijednost varijable *length* na samom početku je sljedeći. Varijabla *length* predstavlja duljinu nadređene grane, a rekli smo da koristeći varijablu *parent*, granu translatiramo relativno u odnosu na nadređenu granu. Stoga moramo uvećati poziciju nove grane na osi Y za duljinu nadređene grane. Kako odmah u sljedećoj liniji skaliramo varijablu *length*, tako osiguravajući da ona bude manja nego nadređena, staru vrijednost moramo spremiti u novu varijablu, jer će nam ona kasnije još trebati.

Sljedeće se kreira valjak odgovarajuće duljine i širine koji predstavlja granu. Također njegovo se "sidro", odnosno točka oko koje se on rotira, postavlja na dno. Nakon toga se kreira objekt *branch* s pridruženim materijalom kojeg dalje možemo translatirati i rotirati po želji, kao što je prikazano u sljedećem programskom isječku:

```
<script>
    branch.rotation.y = (ry*rand_rotate()) * Math.PI / 180;
    branch.rotation.z = (rz*rand_rotate()) * Math.PI / 180;
    branch.position.y = height;

    parent.add( branch );
    counter++;
</script>
```

Granu rotiramo po Y i Z osima prema vrijednostima varijabla *ry* i *rz*, naravno pomnoživši ih s nekom slučajnom vrijednosti kako one ne bi uvijek bile pravilno rotirane. Također, translatiramo je na Y osi u skladu s vrijednošću varijable *height*.

Novokreirani objekt *branch* koji predstavlja granu dodamo kao dijete varijable *parent*. Konačno, uvećamo varijablu *counter* za jedan kako bi se grane, u rekurzivnim pozivima koji slijede, pravilno skalirale.

```
<script>
    if(lvl>0){
        var y_rotation = 0;
        for(var i = 0; i < branch_number ; i++){
            tree(lvl-1, length, width*branch_width_factor,
y_rotation, branch_rotate, branch, counter);
            y_rotation += 360/branch_number;
        }
    }else{
        /**
         * draw leaves?
         */
    }
</script>
```

Znači, ako je stupanj rekurzije veći od nule možemo pozvati sljedeće rekurzivne pozive, a ako je jednak nuli onda se radi o najmanjim granama na vrhu drva gdje bi mogli iscrtati lišće, odnosno krošnju.

Unutar *if* naredbe postoji *for* petlja koja metodu poziva rekurzivno onoliko puta koliko grana želimo generirati, odnosno broj grana na koje će se trenutna podijeliti. To može biti i neka slučajna vrijednost, no kod ovog drva grana se uvijek dijeli na tri manje. Pri tome, za svaku granu određuje se rotacija na osi Y. Rotacija na osi Z se nikada ne mijenja i vrijednost joj je postavljena u postavkama na globalnoj razini. Ona se ipak proslijeđuje kroz rekurzivne pozive u slučaju da se i nju želi mijenjati. Mijenjajući faktore smanjenja širine, duljine, rotacije i ostale, moguće je dobiti drastično različite oblike.

Pri samom rekurzivnom pozivu naravno umanjujemo i širinu grane za neki faktor (mora biti manji od 1).

Trenutnu granu odnosno objekt *branch* proslijeđujemo kao *parent* granama koje će se generirati u rekurzivnim pozivima. Javascript općenito proslijeđuje objekt kao referencu a ne kao vrijednost, što je u ovom slučaju korisno, jer se na taj način stvarno radi hijerarhijska struktura objekata.

Objekt *group*, iz samog početka primjera, sada sadrži sve moguće grane. Preostaje jedino dodati taj objekt na scenu pozivom metode *scene.add()*.

Nekoliko primjera prikazano je na sljedećim slikama.



Slika 56. Generirano drvo (stupanj rekurzije 6)



Slika 57. Generirano drvo (stupanj rekurzije 7)

5.3.2. Optimizacija

Na programskom primjeru koji je priložen uz rad vidljivo je da drvo maksimalne dubine rekurzije 7 (takvo drvo sastoji se od preko 1000 grana) radi na negdje 15-20 FPS (engl. *Frames Per Second*). Uzevši u obzir da u konačnoj kombinaciji fraktala treba cijeli teren posaditi drvećem, a samo jedno drvo značajno usporava rad animacije, potrebno je optimizirati način na koji se drvo iscrtava.

Trenutno radimo preko 1000 poziva grafičkom procesoru za iscrtavanjem, u čemu i jest problem. Manji broj poziva znači brže izvođenje, no kako to postići?

Jedna grana predstavlja jedan poziv za iscrtavanje, no grana odnosno valjak koji koristimo u tu svrhu nije najosnovniji element. Valjak se sastoji od dvije baze s 20-tak i više vrhova, i isto toliko pravokutnika koji čine plašt valjka. No ipak, za iscrtati valjak dovoljan je samo jedan poziv za iscrtavanje.

Naime, svi ti osnovni elementi (vrhovi i stranice) su grupirani u jedan element, mnogo kompleksniji od osnovnih, jer efikasnije je jedanput pozvati grafički procesor da iscrtaja jedan kompleksni objekt nego mnoštvo jednostavnih.

To je i naša ideja, sve grane u jednom drvu pokušati ćemo kombinirati u jedan jedinstveni objekt i njega dodati na scenu. Three.js nudi metode koje možemo koristiti u tu svrhu.

```
<script>
    combined = new THREE.Geometry();
    var matrix = new THREE.Matrix4();

    tree(max_lvl, branch_length, branch_width, 0, 0,
    branch_length_factor, matrix.clone());

    var mesh = new THREE.Mesh( combined, new THREE.MeshLambertMaterial(
    {map: branchTexture } ) );
    scene.add( mesh );
</script>
```

Varijabla *combined* sada predstavlja objekt, odnosno geometriju cijelog drva. Nakon generiranja drva (poziva metode *tree()*) kreira se objekt s generiranom geometrijom, pridruži mu se materijal, te se dodaje na scenu.

Novost je varijabla *matrix* koja će predstavljati matricu modela, a inicijalna vrijednost joj je četverodimenzionalna jedinična matrica. Također, iz definicije metode *tree()* koja je prikazana u sljedećem programskom isječku, vidljivo je da je varijabla *parent* maknuta, i na njeno mjesto dolazi varijabla *matrix*.

```
<script>
    function tree(lvl, length, width, ry, rz, counter, matrix){...}
</script>
```

Prije smo varijablu *parent* odnosno hijerarhijsko organiziranje grupe objekata koristili kako bi svaku granu translatirali i rotirali relativno u odnosu na nadređenu granu.

Nažalost takav sustav ne možemo iskoristiti u ovom slučaju jer ovdje svaku granu koju kreiramo dodajemo u jednu kompleksnu geometriju iz koje je nemoguće identificirati nadređenu granu. No, to ne znači da mi moramo izračunati apsolutnu poziciju svake grane. Upravo zato koristimo matricu modela. Pri svakom sljedećem rekurzivnom pozivu prosljeđivati će se vrijednost matrice modela trenutne grane, tako da će grane koje se izdvajaju iz trenutne ipak imati relativnu poziciju nadređene grane. Kako je to izvedeno, prikazano je u sljedećem programskom isječku:

```
<script>
...
var branch = new THREE.Mesh( branchGeom );

var t_matrix = new THREE.Matrix4();
t_matrix.makeTranslation(0,height,0);
matrix.multiply(t_matrix);

var ry_matrix = new THREE.Matrix4();
ry_matrix.makeRotationY((ry*rand_rotate()) * Math.PI / 180);
matrix.multiply(ry_matrix);

var rz_matrix = new THREE.Matrix4();
rz_matrix.makeRotationZ((rz*rand_rotate()) * Math.PI / 180);
matrix.multiply(rz_matrix);

branch.applyMatrix(matrix);

THREE.GeometryUtils.merge( combined, branch );
...
</script>
```

Prikazani programski isječak ne prikazuje početni dio metode *tree()* gdje se određuje vrijednost varijable *height*, skalira varijabla *length*, kreira geometrija valjka, te mu se definira sidro. Taj dio ostaje isti kao i prije.

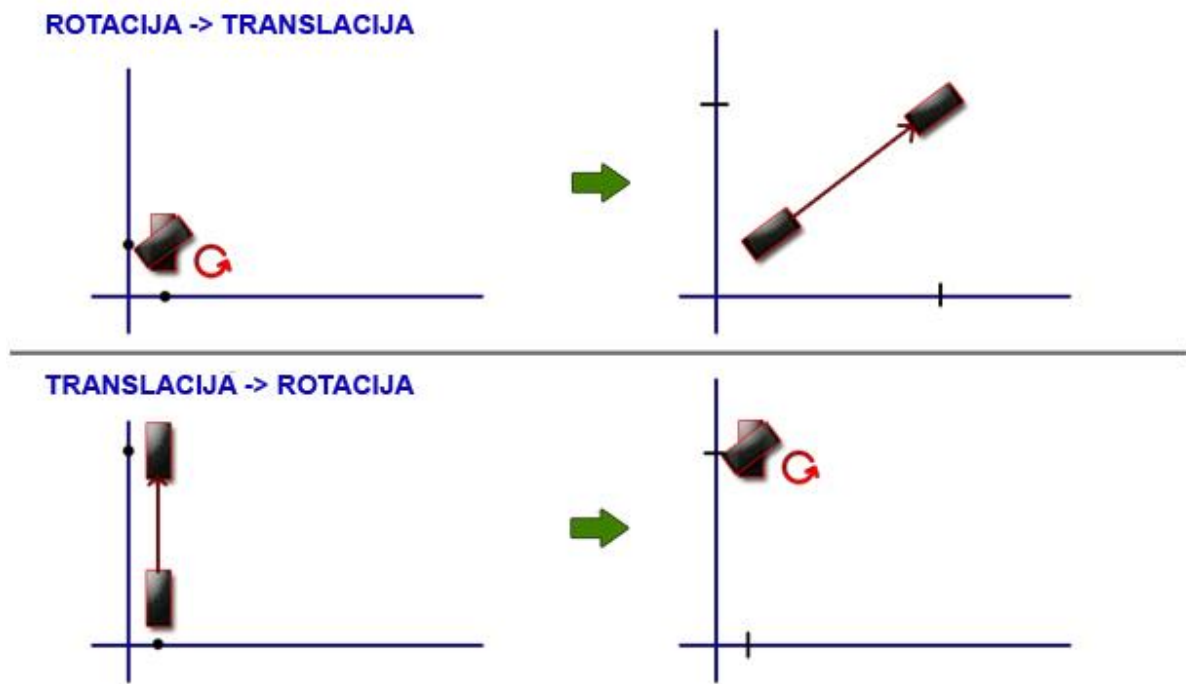
Razlika je u kreiranju objekta grane, gdje nema smisla da mu se pridružuje materijal, no objekt moramo kreirati kako bi bili u mogućnosti odrediti mu poziciju i rotaciju.

Nakon toga vršimo translaticiju i rotiranje i to na način da kreiramo novu matricu za svaku rotaciju ili translaticiju te je množimo s proslijeđenom matricom, odnosno matricom modela nadređene grane. Matrica modela nadređene grane sadrži njenu apsolutnu poziciju i rotaciju, pa na taj način zapravo trenutnu granu rotiramo i translaticiramo relativno na nadređenu.

Prilikom translaticiranja i rotiranja treba paziti na poredak operacija. Kao što se može vidjeti na sljedećoj slici (slika 58), nije isto najprije rotirati pa translaticirati i obrnuto. Kako bi granu uskladili s nadređenom, mi ju najprije trebamo rotirati po Z i Y osi te onda translaticirati po Y osi. Zbog načina kako množenje matrica funkcionira u stvarnosti, taj postupak mora biti obrnut. Znači, najprije se kreira matrica translaticije kojom se određuje pozicija na Y osi. Ta matrica množi se proslijeđenom matricom modela.

Po istom principu kreiraju se matrice najprije za rotiranje po Y osi, a zatim po Z osi. Nakon toga se napravi željena rotacija i množe se s matricom modela.

Sada u matrici modela *matrix* imamo apsolutnu poziciju trenutne grane, a opet relativnu na nadređenu, te ju možemo primijeniti na objekt *branch*.



Slika 58. Rezultat ovisi o poretku operacije translacije i rotacije

Preostaje spojiti objekt trenutne grane u geometriju drva (varijabla *combined* iz samog početka), što se čini pozivom statične metode *GeometryUtils.merge()*.

```
<script>
  if(lvl>0){
    var y_rotation = 0;
    for(var i = 0; i < branch_number ; i++){
      tree(lvl-1, length, width*branch_width_factor,
y_rotation, branch_rotate, counter, matrix.clone());
      y_rotation += 360/branch_number;
    }
  }
</script>
```

Metoda *tree()* se rekurzivno poziva na isti način kao i prije, s time da se umjesto varijable *parent* sada prosljeđuje trenutna modelska matrica *matrix*, ali uz jednu bitnu napomenu. Prije smo direktno prosljeđivali varijablu *parent*, a ako je prosljeđena varijabla objekt, onda Javascript općenito prosljeđuje referencu na objekt umjesto njene vrijednosti. To nam je odgovaralo prije, jer su sve grane sljedećeg rekurzivnog objekta morale biti djeca trenutnog objekta. Međutim kada bi sada na isti način prosljedili varijablu *matrix* koja je također objekt, onda bi sve grane sljedećeg rekurzivnog poziva obrađivale istu matricu, i na kraju bi sve bile isto translirane i rotirane. Iz tog razloga moramo napraviti duplikat objekta trenutne matrice pozivom metode *clone()*, i njega proslijediti u rekurzivnom pozivu.

Dakle, modifikacijom algoritma drva napravili smo da se sve grane spoje u jedinstvenu geometriju, za koju je potreban samo jedan poziv za iscrtavanje grafičkom procesoru. Novi algoritam stvarno i funkcionira jer se drvo maksimalne dubine rekurzije 8 iscrtava na maksimalnih 60 FPS, a pritom sadrži preko 4000 pojedinačnih grana.

5.4. Paprat

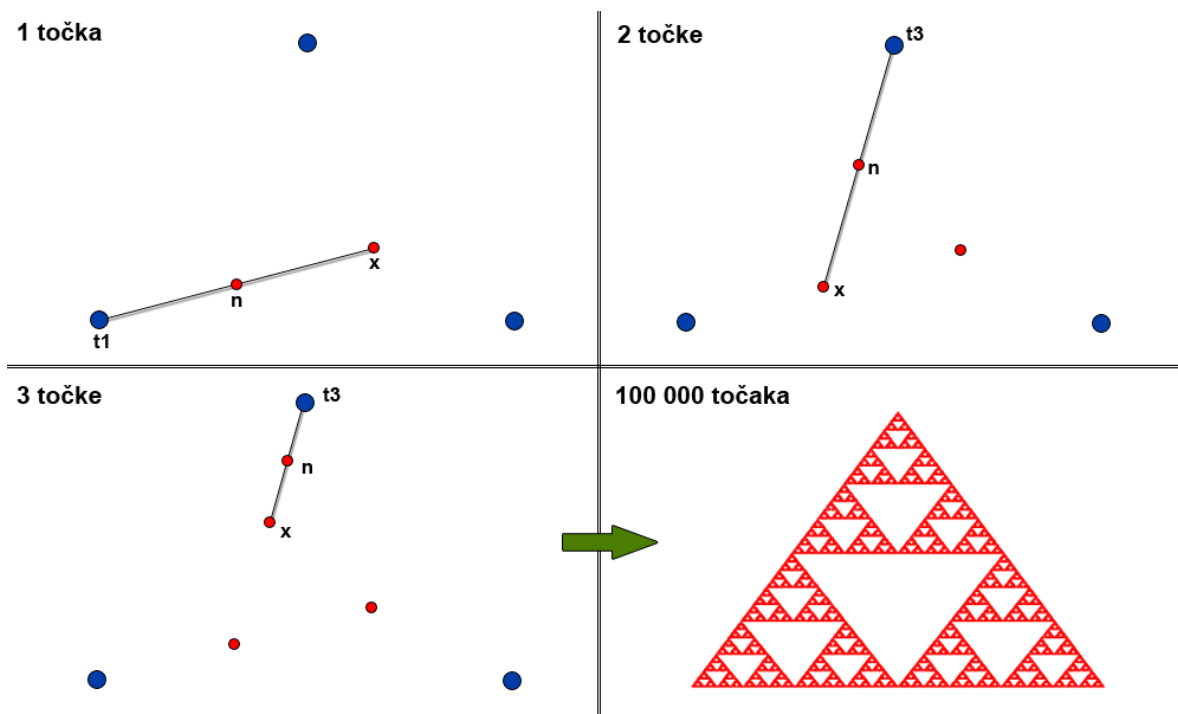
Paprat je oblik IFS fraktala. Kao što je već spomenuto, postoji više načina na koji je moguće generirati takav fraktal. Možemo ga napraviti koristeći bazu i motiv, no taj je primjer već obrađen pri izradi obalne linije. Najpoznatiji primjer kako generirati paprat je korištenjem metode srednje točke.

Sam algoritam za generiranje paprati je prilično specifičan i malo teži za razumjeti, pa će osnovna ideja biti prikazana na primjeru Sierpinskijevog trokuta.

IFS fraktali su općenito poznati po očiglednoj samosličnosti, što u programerskom kontekstu predstavlja rekurzija. Međutim, metoda srednje točke ne zahtijeva rekurziju već običnu iteraciju. Također, po prvi puta ne generiraju se geometrijski objekti koji prezentiraju neki dio fraktala, već ukupan lik čini ogroman skup točaka. Ključ u dobivanju različitih oblika je određivanje pozicije točke.

Algoritam za generiranje Sierpinskijevog trokuta je sljedeći (popraćeno vizualnim primjerom na slici 59):

1. odabiru se tri točke (t_1 , t_2 , t_3) u ravni, te jedna točka x koja se nalazi unutar trokuta kojeg čine točke t_1 , t_2 i t_3 .
2. slučajno se odabere jedna od točaka (samo t_1 , t_2 , i t_3), te odredi nova točka "n" koja se nalazi između slučajno odabrane točke i točke "x".
3. novonastala točka "n" postaje točka "x".
4. ponavljati korake 2 i 3 onoliko puta koliko točaka želimo generirati.



Slika 59. Generiranje Sierpinskijevog trokuta

Specifičnost ovakvih algoritama je da je potrebna iznimno velika količina točaka da bi konačan oblik bio vidljiv. Na slici 59 su u prva tri koraka vidljive 4 točke iz kojih je nemoguće zaključiti kakav konačni oblik one predstavljaju, a ništa bolje ne bi bilo ni sa stotinu točaka.

Na sličnom principu počiva i algoritam za generiranje paprati, gdje se točka određuje zbrajanjem i množenjem prethodno generirane točke sa specifičnim faktorima.

Algoritam je osmislio britanski matematičar Michael Barnsley, a bit će objašnjen kroz programski primjer.

```
<script>
    var particles = generate_points(count);

    var particle_system = new THREE.ParticleSystem(particles, new
THREE.ParticleBasicMaterial({ color: 0xff0000, size: 5}));
    scene.add(particle_system);
</script>
```

Kao točke koristit ćemo čestice (engl. *particles*) odnosno sustav čestica (engl. *Particle System*) koji je implementiran u Three.js biblioteci. Metoda *generate_points()* vratit će onoliko točaka, grupiranih u jednu varijablu, kolika je vrijednost varijable *count*. Kao i do sada, na osnovi geometrije (skupa točaka) kreira se objekt s pridruženim materijalom te se dodaje na scenu.

Unutar funkcije *generate_points()* odvijaju se sljedeće naredbe:

```
<script>
    var particles = new THREE.Geometry();
    var x = [0,0];
    var n = [0,0];

    for (var k = 0 ; k < count ; k++){
        /**
            determine n points
        */

        var vertex = new THREE.Vector3();
        vertex.x = n[0]*100;
        vertex.y = n[1]*100;
        vertex.z = -Math.pow(n[1],2)*5;

        particles.vertices.push(vertex);

        x[0] = n[0];
        x[1] = n[1];
    }
    return particles.clone();
</script>
```

Za početak se kreira nova geometrija u koju će se dodavati generirane točke. Također, određuju se dvije početne točke (obje imaju koordinate 0,0 na X i Y osi). Prisjetimo se, u primjeru s Sierpinskijevim trokutom odredili smo 3 početna vrha trokuta i još jednu točku koja se nalazi u središtu trokuta. Ovdje nam je dovoljno odrediti dvije početne točke.

Zatim kreće iteracija gdje se na neki način izračunava nova točka, što će biti prikazano kasnije u posebnom programskom isječku. Nakon što je generirana točka, ona se pridružuje geometriji, odnosno varijabli *particles* koju na kraju vraćamo glavnom programu.

Prije toga koordinate novokreirane točke n pridružujemo točki x koja će u sljedećoj iteraciji služiti za ponovno izračunavanje točke n .

Također treba napomenuti još jednu sitnicu. Naime, paprat je dvodimenzionalni fraktal, stoga bi pozicija na Z osi uvijek trebala biti 0. No, kako bi barem prividno davao dojam trodimenzionalnosti, pozicija na osi Z se modificira tako da iznosi kvadratnu vrijednost pozicije na osi Y . Kako kvadratna funkcija raste, tako se i paprat "uvija" što će biti vidljivo kasnije na slikama.

Samo generiranje točke n izgleda ovako:

```
<script>
  rand = Math.random();
  //section 1
  if ( (rand >= 0.0) && (rand < 0.83) ){
    n[0] = (( 0.86* x[0]) + ( 0.03 * x[1]) + 0.0);
    n[1] = ((-0.03* x[0]) + ( 0.86 * x[1]) + 1.5);
  }
  //section 2
  if ( (rand >= 0.83) && (rand < 0.91) ){
    n[0] = ( 0.02* x[0]) + (-0.25 * x[1]) + 0.0;
    n[1] = ( 0.21* x[0]) + ( 0.23 * x[1]) + 1.5;
  }
  //section 3
  if ( (rand >= 0.91) && (rand < 0.99) ){
    n[0] = (-0.15* x[0]) + ( 0.27 * x[1]) + 0.0;
    n[1] = ( 0.25* x[0]) + ( 0.26 * x[1]) + 0.45;
  }
  //section 4
  if ( rand >= 0.99){
    n[0] = ( 0.00* x[0]) + ( 0.00 * x[1]) + 0.0;
    n[1] = ( 0.00* x[0]) + ( 0.17 * x[1]) + 0.0;
  }
</script>
```

Generira se slučajni broj i ovisno o tom broju, nova točka se generira na 1 od 4 moguća načina. Jedan način izводи se 83% puta, drugi 8% puta, treći također 8%, dok se četvrti način izводи samo 1% puta.

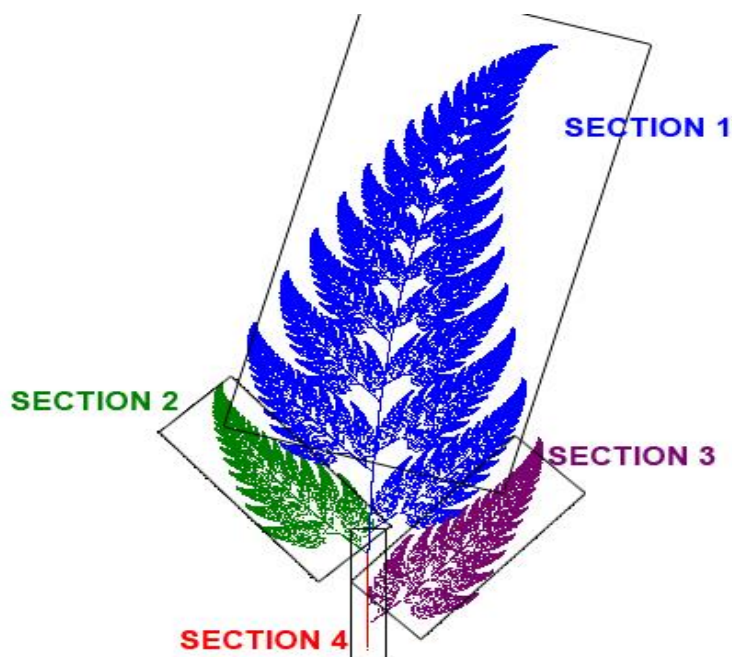
Bez obzira na način na koji se točka generira, njena vrijednost izračunava se po sljedećoj funkciji:

$$n_x = \frac{x_x}{factor1} + \frac{x_z}{factor2} + factor3$$

$$n_z = \frac{x_x}{factor4} + \frac{x_z}{factor5} + factor6$$

Faktori su specifični, no sitna promjena može u konačnici generirati različite oblike paprati ili lišća.

Na sljedećoj slici prikazano je koji dio koda (u programskom isječku označen kao "section") generira koji dio paprati.



Slika 60. Generirana paprat po sektorima

Iako "section 1" generira najveći dio fraktala, dok npr. "section 4" koji se izvodi tek u 1% slučajja generira tek crvenu liniju na samom dnu, izostavljanjem bilo kojeg djela koda, algoritam bi generirao sasvim drugačiji oblik.

Konačno, na sljedećoj slici moguće je vidjeti različite varijacije koje se mogu postići promjenom faktora.



Slika 61. Različite modifikacije paprati

6. Simulacija leta nad fraktalnim terenom, posađenim fraktalnom vegetacijom

Sada kada su implementirani svi individualni primjeri, vrijeme je da ih sve spojimo u jedan program. U ovom poglavlju bit će prikazan postupak stvaranja otoka (teren + obalna linija), sadenje otoka vegetacijom (drvo i paprat), te problemi koji se mogu pojaviti u sličnim situacijama, ponajviše vezanim za optimizaciju.

Najprije će se generirati vegetacija. Slijedi teren, a tokom generiranja terena posadit ćemo vegetaciju koju smo prethodno generirali. Na kraju se kreira vodena površina koja će u kombinaciji s terenom napraviti obalnu liniju, i ukupno otok, ili otočje.

6.1. Otok - teren i obala

U postupku generiranja terena neće se skoro ništa mijenjati. Radit će se rekurzija dubine 8, radi velike količine detalja, a opet pristojnim vremenom generiranja (5-6 sekundi). Tako će teren činiti ukupno 65536 točaka.

Ono što ćemo promijeniti u generiranju terena je da ćemo osigurati da napravimo otok, odnosno teren koji bi imao dovoljan broj točaka iznad površine vode. Naime, svaka točka ovisno o amplitudi može biti u rasponu od točaka $xmin$ i $xmax$ koje se izračunavaju na sljedeći način:

$$xmin = -\frac{amplitude}{2}, xmax = \frac{amplitude}{2}$$

To praktički znači da svaka točka ima jednake šanse biti manja ili veća od nule. Uzevši u obzir da ćemo teren presjeći negdje iznad nule, točka ima veće šanse da konačno bude ispod vodene površine nego iznad.

Najjednostavniji način da osiguramo da će dovoljan broj točaka biti iznad razine vode, a da opet ne modificiramo previše trenutni algoritam, je da utječemo na generiranje samo prve točke:

```
<script>
    if(first){
        main_array[perm_lvl][xm][zm]['val'] = Math.random()*amplitude -
(amplitude/4);
        first = false;
    }else{
        main_array[perm_lvl][xm][zm]['val'] = Math.random()*amplitude -
(amplitude/2);
    }
</script>
```

Dakle, varijabla *first* je inicijalno postavljena na *true* i odmah nakon što se prva točka generira na taj poseban način, varijabla *first* se postavlja na *false* tako osiguravajući da se taj poseban način generiranja točke nikad više ne pokrene. Razlika je u tome što se od slučajnog generiranog broja oduzima manji dio jer ga dijelimo s većim brojem. Na taj način točke $xmin$ i $xmax$ imaju sljedeće vrijednosti:

$$x_{min} = -\frac{amplitude}{4}, x_{max} = \frac{3}{4} amplitude$$

Time smo osigurali da ipak većina točaka bude iznad površine vode.

S druge strane, obalu nećemo posebno generirati. Kao što je napomenuto u prethodnom poglavlju gdje je obrađivana obalna linija, nju se može dobiti i presjekom 3D Perlinovog šuma, odnosno terena kojeg smo kreirali.

Ravnina koja će presjeći teren predstavlja vodu. Kako izgleda dodavanje vode pokazano je u sljedećem programskom isječku:

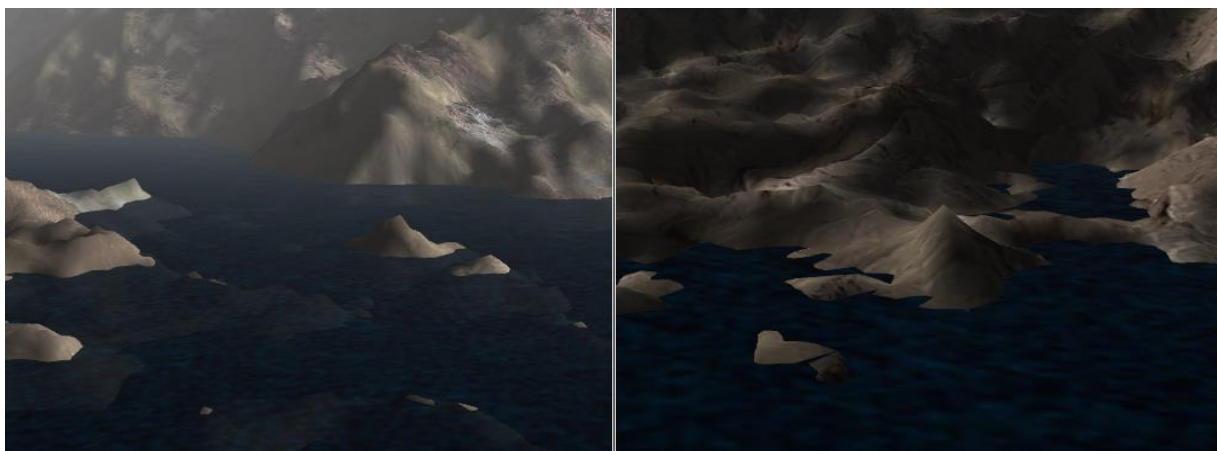
```
<script>
    var plane_geo = new THREE.PlaneGeometry(10000, 10000, 1,1);
    var texture = THREE.ImageUtils.loadTexture( 'images/sea_tex.jpg' );
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    texture.repeat.set(16,16);

    var plane = new THREE.Mesh(plane_geo, new THREE.MeshLambertMaterial
    ({ map: texture, transparent: true, opacity: 0.9 })))
    plane.position.y = 20;
    plane.rotation.x = 270 * Math.PI / 180;
    scene.add(plane);

    var plane2 = new THREE.Mesh(plane_geo, new THREE.MeshLambertMaterial
    ({ map: texture}));
    plane2.position.y = 5;
    plane2.rotation.x = 270 * Math.PI / 180;
    scene.add(plane2);
</script>
```

Kreira se velika ravnina (10000 x 10000 u usporedbi sa 2000 x 2000 koliku površinu ima generirani teren) kako bi se dobio dojam beskonačnosti oceana.

Nakon što se definira tekstura, kreiraju se dva objekta (vodena površina) na osnovu ravnine. Jedan objekt se pozicionira više od drugog (*position.y = 20 | 5*), a onom koji je više pozicioniran postavlja se i prozirnost. Ono što želimo postići s dvije ravnine je da se vidi i dio terena ispod razine vode, odnosno plićak, što se može vidjeti i na sljedećoj slici.



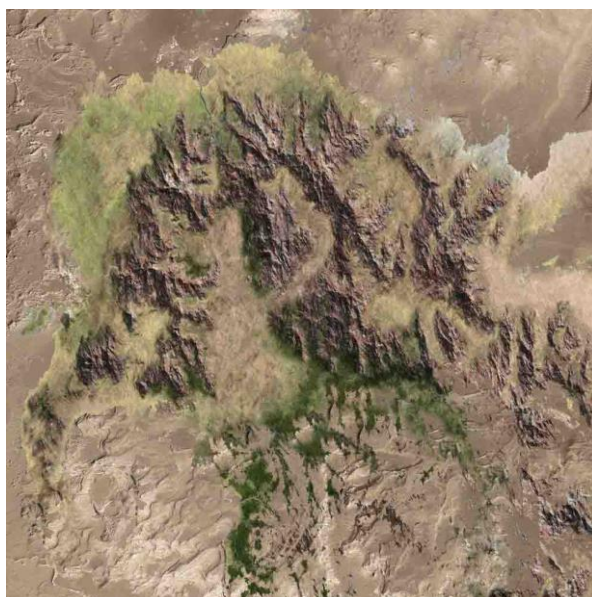
Slika 62. Razlika u vodenoj površini sa dvije ravnine i samo s jednom.

Kao što se može vidjeti na slici 62, korištenjem prozirnosti i dvije ravnine dobije se lijepi efekat i dojam plićaka. Slično se moglo postići i korištenjem samo jedne djelomično prozirne ravnine, no kako na jednom djelu površina terena prestaje tako bi i prelazak bio prenapli i estetski neprivlačniji.

Pitanje koje bi se još moglo pojaviti je na koji način će se sam teren prikazati. Moguće je generirati boju točke terena ovisno o njegovoj visini, no ipak konačan rezultat nije impresivan ili uvjerljiv, jer stvarni teren ne mogu opisati osnovne boje i jednostavan gradijent između njih. Može se uzeti mala tekstura te je višestruko ponavljati preko površine, kao što je napravljeno s vodom, no onda je konačna slika previše jednolična. Također, potrebno je pronaći teksturu koja se savršeno ponavlja i ne stvara čudne prelaze, što je prilično teško ako tražimo teksturu kamena i slično.

Zato smatram da je najbolja opcija korištenje velike teksture, koja se prevlači preko terena bez da ju se ponavlja. Slika koja se koristi ima rezoluciju 4000 x 4000 slikovnih elemenata, a velika je 1MB, što je u razumnim okvirima uzevši u obzir da se cijela aplikacija učitava preko web-a.

Tekstura je prikazana na sljedećoj slici:

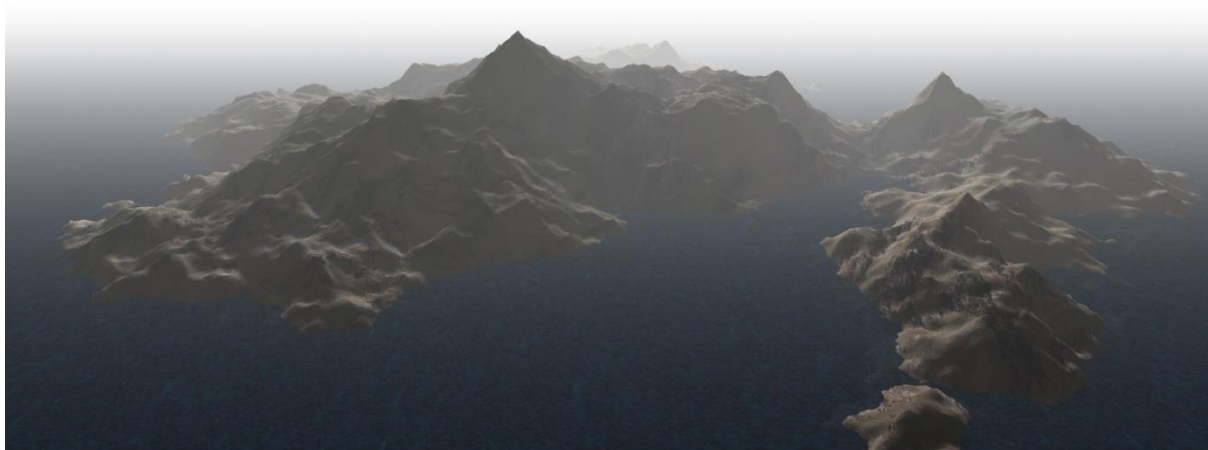


Slika 63. Tekstura korištena za prikaz terena

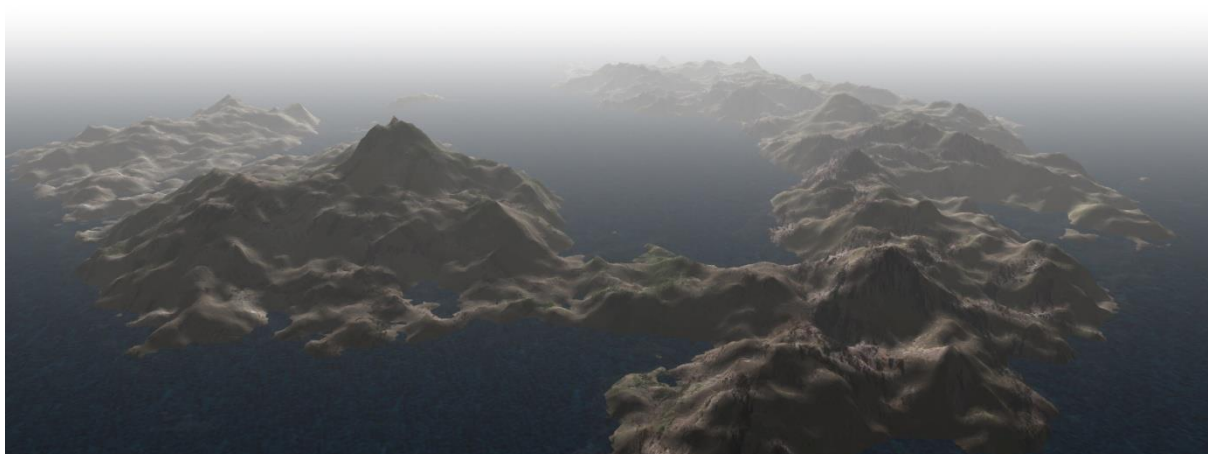
Tekstura je ručno napravljena u Adobe Photoshopu kao kombinacija nekoliko tekstura nađenih na internetu, te obrađena kako bi se stijene, trava i ostali dijelovi pozicionirali na određeni dio teksture. Kako uglavnom očekujemo najvišu točku negdje u sredini terena tako su tamo pozicionirane stijene, okolo njih trava, a uz same rubove pijesak koji može poslužiti i kao obala.

Naravno, najimpresivniji pogled je iz daljine, dok zumiranje otkriva pikselizaciju što je i normalno za ovakav način korištenja teksture.

Konačno, sceni dodajemo i maglu za dodatni efekt. Mogući ukupan rezultat prikazan je na sljedećim slikama.



Slika 64. Primjer terena i obale



Slika 65. Drugi primjer terena i obale

6.2. Razmještanje vegetacije

Prije nego što se vegetacija ustvari posadi po terenu potrebno ju je generirati. Razlog je taj što će se samo razmještanje vršiti tijekom generiranja terena pa vegetacija mora već biti pripremljena. Na samom početku, generirati će se tri-četiri različite verzije drva i paprati i onda će se prilikom sadnje kopirati jedna od njih. Njoj će se odrediti slučajna rotacija, što je dovoljno da se u skupini od tisuće drva ne primijeti nekakvo ponavljanje.

Svi faktori koji utječu na generiranje drva, uključujući širinu grane, rotacije, faktore skaliranja, odrede se slučajno, i na osnovu njih generira se drvo. Taj postupak se ponavlja onoliko puta koliko različitih verzija želimo napraviti. Također svakoj verziji se može pridružiti drugačija tekstura, odnosno materijal.

Jedino što je promijenjeno u postupku generiranja drva je broj grana na koje se jedna grana dijeli. Prije se grana uvijek dijelila na tri manje dok se sada i broj grana u svakom rekurzivnom pozivu slučajno generira, sve u svrhu generiranja različitijih drva.

Razina rekurzije koja će biti upotrijebljena za generiranje drva je 4. Iako je to malo detalja za pojedinačno drvo, s obzirom na zračni i udaljeni pogled, to će biti dovoljno. U poglavlju s optimizacijom drva postigli smo da je za iscrtavanje jednog drva dovoljan jedan poziv. Kada je samo jedno drvo na ekranu svejedno je da li je to drvo sastavljeno od 100 ili 2000 grana. No, stavimo li 2000 drva na ekran počinje se primjećivati razlika. Lakše je iscrtati 2000 jednostavnijih objekata nego isti broj kompleksnijih bez obzira što se vrši isti broj poziva za iscrtavanje grafičkom procesoru.

Postavlja se pitanje zašto se ne bi moglo spojiti sva drva u jednu geometriju. Odgovor je taj da je proces spajanja geometrije iznimno spor. Jedno je dodati jedan jednostavan objekt (valjak) koji predstavlja granu u rastuće kompleksan objekt (cijelo drvo), a drugo je spajati dva kompleksna objekta. Čak i da je prihvatljivo čekati nekoliko minuta koliko traje proces spajanja tek 100-tinjak gotovih drva, performanse su tek neznatno bolje. Spajanje geometrije je korisna tehnika, no ima svojih granica.

U vegetaciju spada i paprat te generiramo tri-četiri predefinirane verzije paprati, svaku s 5000 točaka. Razmještanje vegetacije (metoda *plant_tree()*) izvodi se u djelu gdje se točkama ravnine dodjeljuje ukupna visinska vrijednost iz polja koje sadrži sve funkcije Perlinovog šuma.

Metodi *plant_tree()* proslijeđuju se koordinate na kojoj će se posaditi jedan od oblika vegetacije. Ovako izgleda navedena metoda:

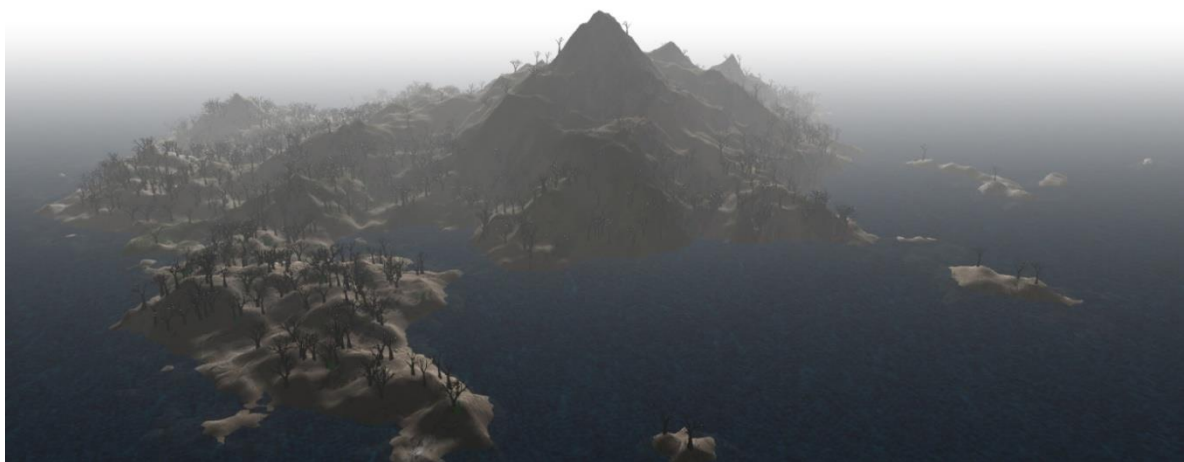
```
<script>
function plant_tree(x,y,z){
    var tree_root = y - branch_length - (branch_length/3);
    if(tree_root > 20 && tree_root < 100){
        if(Math.random() > 0.9){
            var mesh = // slučajno odrediti tip vegetacije (4 vrste
drva i 3 paprati)
            // .clone()
            mesh.position.x = x - (length/2);
            mesh.position.y = tree_root;
            mesh.position.z = z - (length/2);
            mesh.rotation.y = Math.random()*360 * Math.PI / 180;
            scene.add(mesh);
        }
    }else if( tree_root > 100){
        if(Math.random() > 0.995){
            // isto kao i gore
        }
    }
}
}</script>
```

Vegetacija se sadi samo ako je točka na kojoj se treba nešto posaditi iznad površine vode (20 na osi Y, kao što se može vidjeti u prethodnom podpoglavlju). Također, visina određuje i gustoću vegetacije. Ako je visina manja od 100, onda se vegetacija sadi u 10% slučajja, a ako je visina veća od 100, vegetacija se sadi tek u 0.5% slučajja.

Ovisno o generiranom terenu to znači prosječno 1500 do maksimalno 2000-2200 posađenih drva ili paprati. U "najgušćem" slučaju to znači nekih 15 FPS, što nije baš optimalno, ali je opet pristojno za izvođenje programa.

Samo razmještanje je jednostavno. Na osnovi nekog slučajnog broja odabire se jedna verzija vegetacije, dodjeljuje joj se pozicija na osnovi proslijeđenih koordinata, te se rotira za neki slučajan kut na Y osi.

Naravno, ponovno se može igrati s koeficijentima gdje će se nešto generirati i što će se generirati, a može se dodati i slučajno skaliranje kako bi se dobila još raznovrsnija vegetacija. Sve ovisi o tome kakvu konačnu sliku želimo dobiti. Kako to konačno izgleda moguće je vidjeti na sljedećim slikama.



Slika 66. Teren posut vegetacijom



Slika 67. Teren posut vegetacijom- drugi prikaz



Slika 68. Teren posut vegetacijom - treći prikaz

6.3. Simulacija leta

Upravljanje simulacijom će se vršiti iz trećeg lica. To znači da se neće direktno kontrolirati kamera, što bi bilo upravljanje iz prvog lica, već će se kontrolirati nekakav objekt, a kamera će uvijek pratiti taj objekt. Taj objekt može biti nekakav model aviona, helikoptera, no mi ćemo dodati jedan jednostavan kursor koji će pomagati pri snalaženju u prostoru, što je dosta korisno s obzirom na avionski model kontrola koje ćemo implementirati.

Dakle prije samog ulaženja u animaciju (metoda *init()* gdje se generiraju svi objekti na sceni) potrebno je generirati naš "avion" i dodati ga na scenu. Ostatak posla odvija se u metodi *update()* koja se periodično poziva.

Prvi korak je određivanje stupnja rotacije koja će se primijeniti pritiskom na neku tipku:

```
<script>
var delta = clock.getDelta();

var moveDistance = 100 * delta;
var rotateAngleX = Math.PI / 8 * delta;
var rotateAngleZ = Math.PI / 2 * delta;
</script>
```

Vrijednost varijable *delta* je stupanj za koji moramo skalirati translaticiranje ili rotiranje, a ovisan je o vremenu proteklom od posljednjeg iscrtavanja. Što je iscrtavanje brže (veći FPS) to je *delta* manji, i suprotno. Na taj način osiguravamo da se avion miče istom brzinom bez obzira na brzinu iscrtavanja, kompleksnost scene ili performanse računala na kojem se aplikacija izvodi.

Avion mičemo brzinom od 100 slikovnih elemenata po sekundi (*delta* = 1 znači 1px/s), i ona je konstantna, odnosno neće postojati način da se modificira brzina aviona u animaciji. Također rotiramo ga brže po osi Z nego po osi X.

Spomenuli smo da avion ima avionske kontrole (pojednostavljene), što znači da se avionu prvo kontrolira rotacija na osi kojom se kreće (os Z). Na taj način mijenja se nagib aviona ali se uopće ne utječe na njegov smjer. Tu rotaciju je moguće izvoditi dosta brzo. Druga rotacija je rotacija oko osi X. Ona je sporija i mijenja smjer aviona ovisno o rotaciji na osi Z.

Implementacija kontrole prikazana je u sljedećem programskom isječku:

```
<script>
    MovingCube.translateZ( -moveDistance );

    if ( keyboard.pressed("S") )
        MovingCube.rotateOnAxis( new THREE.Vector3(1,0,0), -
rotateAngle1);
    if ( keyboard.pressed("W") )
        MovingCube.rotateOnAxis( new THREE.Vector3(1,0,0), -
rotateAngle1);
    if ( keyboard.pressed("A") )
        MovingCube.rotateOnAxis( new THREE.Vector3(0,0,1), -
rotateAngle2);
    if ( keyboard.pressed("D") )
        MovingCube.rotateOnAxis( new THREE.Vector3(0,0,1), -
rotateAngle2);
</script>
```

Dakle koriste se tipke "S" i "W" za rotiranje po osi X (gore i dolje), te "A" i "D" za rotiranje aviona po osi Z (lijevo i desno). Također, odmah u prvoj liniji vidljivo je da se avion uvijek translata po Z osi. Kretanje aviona je riješeno, preostaje samo namjestiti kameru da prati avion, i uvijek gleda u njega:

```
<script>
    var relativeCameraOffset = new THREE.Vector3(0,0,150);
    var cameraOffset = relativeCameraOffset.applyMatrix4(
MovingCube.matrixWorld );

    camera.position.x = cameraOffset.x;
    camera.position.y = cameraOffset.y;
    camera.position.z = cameraOffset.z;
    camera.lookAt( MovingCube.position );
</script>
```

Time je zaključen postupak generiranja prirodnih elemenata pomoću fraktala i spajanja u jedinstvenu aplikaciju. Rezultat je vizualno atraktivna aplikacija koja je kreirana koristeći isključivo matematičke formule i slučajnost.

Ono na čemu bi se moralo poraditi je daljnja optimizacija, kako bi se na scenu mogli dodati i drugi detaljniji fraktali, no to je tema za neki drugi rad.

7. Zaključak

Fraktalne strukture se pojavljuju u mnogim oblicima u prirodi. To nam daje mogućnost da iskoristimo matematičke formule kako bi u računalnoj grafici simulirali prirodu.

Danas se u računalnoj grafici fraktali često i koriste na način da se osmisli osnovni koncept, a pomoću fraktala se dodaje velika količina detalja. To posebno vrijedi za teren. S druge strane, fraktalno drvo i nema neku veliku primjenu, no bez obzira na to svakako ima potencijal za generiranje uvjerljivih rezultata s minimalnim učešćem dizajnera.

Potencijal ima i WebGL, još uvijek eksperimentalna tehnologija koja donosi napredne grafičke mogućnosti u web standard. Time postaje masovno dostupna, iako, ne treba se zavaravati, i dalje je potrebno posjedovati solidno računalo kako bi se aplikacija mogla efikasno izvoditi. Također, performanse i mogućnosti WebGL-a nisu ni blizu onome što danas nudi npr. DirectX, no mislim da barem zasad WebGL i nije namjenjen izvođenju igara u čiji je razvoj utrošeno milijune dolara. U budućnosti, tko zna?

Za sada, WebGL je odlično proširenje trenutnim mogućnostima weba, koji sve više teži interaktivnosti i atraktivnosti.

Primjeri prikazani u ovom radu pokazuju što može učiniti kombinacija teorije fraktala u prirodi i WebGL-a. No, time je samo zagrebana površina svih mogućnosti koje nam se pružaju. U primjerima nije iskorištena mogućnost direktnog programiranja u jeziku za sjenčanje (engl. *shader language*), što bi moglo rezultirati znatno boljim performansama i boljom grafikom. Koristili smo tek četiri fraktalne pojave kako bi simulirali prirodu. Pomoću fraktala može se simulirati mnogo više vegetacije od drva i paprati, mogu se simulirati i oblaci, pahuljice, kristali itd.

Sve u svemu uživao sam raditi na ovom diplomskom radu iako se s računalnom grafikom nisam sretao prije ovog diplomskog rada, ili bolje rečeno, kolegija Računalna grafika. Obradeni primjeri prikazuju vezu između teorije (matematika) i same prakse (implementacija u specifičnoj tehnologiji), što je odličan način na koji se korisnik s dovoljnim iskustvom u programiranju može početi učiti principima izrade 3D grafike i animacije.

8. Literatura

- [1] ahyco.uniri.hr (2007 a), *Baza-motiv fraktal*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/baza_motiv.html
- [2] ahyco.uniri.hr (2007 b), *Prašina i skupina*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/baza_motiv.html
- [3] ahyco.uniri.hr (2007 c), *Fraktalni baldahini*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/fraktalni_baldahini.html
- [4] ahyco.uniri.hr (2007 d), *IFS fraktali*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/ifs_fraktali.html
- [5] ahyco.uniri.hr (2007 e), *Nestandardni fraktali*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/nestandardni_f.html
- [6] ahyco.uniri.hr (2007 f), *Peano krivulje*, Preuzeto 6. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/peanove_krivulje.html
- [7] ahyco.uniri.hr (2007 g), *Plazma fraktali*, Preuzeto 7. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/plazma_f.html
- [8] ahyco.uniri.hr (2007 h), *Sweeps*, Preuzeto 7. kolovoza 2013. s <http://ahyco.uniri.hr/seminari2007/fraktali/sweeps.html>
- [9] ahyco.uniri.hr (2007 i), *Neobični mamitelji*, Preuzeto 12. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/neobicni_mam.html
- [10] ahyco.uniri.hr (2007 j), *Julijin skup*, Preuzeto 13. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/julijin_s.html
- [11] ahyco.uniri.hr (2007 k), *Mandelbrotov skup*, Preuzeto 13. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/Mandelbrotov_skup.html
- [12] ahyco.uniri.hr (2007 l), *Cantorov skup*, Preuzeto 13. kolovoza 2013. s http://ahyco.uniri.hr/seminari2007/fraktali/kantorov_s.html
- [13] Azí Arts (s.a.), *Fractals*, Preuzeto 3. kolovoza 2013. s <http://aziarts.com/AIR/Fractal1.html>
- [14] Beddard T (2009), *4D Quaternion Julia-set Ray Tracer with Ambient Occlusion*, Preuzeto 11. kolovoza 2013. s <http://www.adobe.com/cfusion/exchange/index.cfm?event=extensionDetail&extid=1888022>
- [15] Blog Kaoslantida (2007), „Trivijalne“ činjenice o pojmu dimenzije (III.), Preuzeto 3. kolovoza 2013. s <http://elgrunon.wordpress.com/2007/05/09/%E2%80%9Ctrivijalne%E2%80%9D-cinjenice-o-pojmu-dimenzije-iii/>
- [16] Bradley L (2010), *Strange Attractors*, Preuzeto 11. kolovoza 2013. s <http://www.stsci.edu/~lbradley/seminar/attractors.html>
- [17] Brdar K, Dobrinić M, Joskić R (2012), *Fraktali*. Seminarski rad, Sveučilište u Zagrebu, Fakultet kemijskog inženjerstva i tehnologije, Zavod za matematiku
- [18] Eklić H (s.a.), *Eksperimentalno određivanje fraktalne dimenzije*, Preuzeto 2.kolovoza 2013. s <http://e.math.hr/old/frakdim/index.html>
- [19] e-škola (s.a.), *Fraktali – čudesne slike kaosa*, Preuzeto 1. kolovoza 2013. s http://eskola.hfd.hr/mini_projekt/mp7/fraktali_3.htm
- [20] Fractal Foundation (s.a. a), *Geometric Fractals, Iterated Functions*, Preuzeto 8. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [21] Fractal Foundation (s.a. b), *Mandelbrot Magic, Julia Sets*, Preuzeto 14. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [22] Fractal Foundation (s.a. c), *The Mandelbrot Set, Exploring Infinity*, Preuzeto 14. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>

- [23] Fractal Foundation (s.a. d), *Fractal Applications, Medical Fractals*, Preuzeto 18. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [24] Fractal Foundation (s.a. e), *Fractals in Nature, Rivers*, Preuzeto 18. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [25] Fractal Foundation (s.a. f), *Fractals in Nature, Lightning*, Preuzeto 18. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [26] Fractal Foundation (s.a. g), *Fractals in Nature, Spirals*, Preuzeto 18. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [27] Fractal Foundation (s.a. h), *Fractal Applications, Fractal Devices*, Preuzeto 18. kolovoza 2013. s <http://fractalfoundation.org/OFC/OFC-index.htm>
- [28] Fraiberger M (2009), *Pandora's 3D box*, Preuzeto 5. kolovoza 2013. s <http://plus.maths.org/content/pandoras-3d-box>
- [29] Freedesign4me (2010), *Free Vector Tree Branches*, Preuzeto 4. kolovoza 2013. s <http://www.freedesign4me/vectors/free-vector-tree-branches/>
- [30] Gardi L (2009), *Strange Attractors*, Preuzeto 15. rujna 2013. s <http://www.butterflyeffect.ca/Close/Pages/StrangeAttractors.html>
- [31] Gustavson S (2005), *Simplex noise demystified*, Preuzeto 20. kolovoza 2013. s <http://www.google.hr/url?sa=t&rct=j&q=simplex%20noise&source=web&cd=4&ved=0CDkQFjAD&url=http%3A%2F%2Fwww.itn.liu.se%2F~stegu%2Fsimplexnoise%2Fsimplexnoise.pdf&ei=I6EvUr2UPMHPAfKxYGQCQ&usg=AFQjCNEVzOM03haFrTgLrjJp-jPkQyTOKA&bvm=bv.51773540,d.bGE&cad=rja>
- [32] Ibrahim M, Krawczyk R (2006), *Generating Fractals Based on Spatial Organizations*, Preuzeto 7. kolovoza 2013. s <http://cgg-journal.com/2006-2/02/index.html>
- [33] jeff.tsai (s.a.), *Peano curves*, Preuzeto 10. kolovoza 2013. s <http://pzacad.pitzer.edu/~jtsai/peano.html>
- [34] Liming H, Ting C (s.a.), *World of Fractal*, Preuzeto 2. kolovoza 2013. s http://www.google.hr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&ved=0CFIQFjAG&url=http%3A%2F%2Fwww.math.nus.edu.sg%2Faslaksen%2Fgem-projects%2Fmaa%2FWorld_of_Fractal.pdf&ei=2tshUq-fNInXswani4CgAQ&usg=AFQjCNEXFkEnJchF7b8T0-Ty0aLXYHCBPw&sig2=oi8gUcCwpKCz0TFdVAXJ5g&bvm=bv.51495398,d.Yms&cad=rja
- [35] Nelson P (2010), *Benoit Mandelbrot*, Preuzeto 1. kolovoza 2013. s <http://hilobrow.com/2010/11/20/benoit-mandelbrot/>
- [36] Olah C (2011), *Understanding Pascal's Triangle*, Preuzeto 9. kolovoza 2013. s <http://christopherolah.wordpress.com/2011/08/29/understanding-pascals-triangle/>
- [37] Perlin K (1999 a), *Making Noise, What's noise?*, Preuzeto 20. kolovoza s <http://www.noisemachine.com/talk1/3.html>
- [38] Perlin K (1999 b), *Making noise, Meanhwiile, back at the Ranch*, Preuzeto 20. kolovoza 2013. s <http://www.noisemachine.com/talk1/12.html>
- [39] Platonic Realms (s.a.), *Cantor Set*, Preuzeto 11. kolovoza 2013. s <http://platonicrealms.com/encyclopedia/Cantor-set>
- [40] R.C.L. (2010), *Law and Disorder*, Preuzeto 12. kolovoza 2013. s <http://www.lawsofwisdom.com/course-overview/statement-of-the-law/law-and-disorder/>
- [41] Riddle L (2013 a), *Classic Iterated Function Systems, Sierpinski Carpet*, Preuzeto 7. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/carpet/carpet.htm>

- [42] Riddle L (2013 b), *Classic Iterated Function Systems, Pythagorean Tree*, Preuzeto 16. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/pythagorean/pythTree.htm>
- [43] Riddle L (2013 c), *Classic Iterated Function Systems, Sierpinski Gasket*, Preuzeto 17. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/siertri.htm>
- [44] Riddle L (2013 d), *Classic Iterated Function Systems, Sierpinski Pedal Triangle*, Preuzeto 17. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/sierpedal.htm>
- [45] Riddle L (2013 e), *Classic Iterated Function Systems, Fat Sierpinski Triangle*, Preuzeto 17. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/sierfat.htm>
- [46] Riddle L (2013 f), *Classic Iterated Function Systems, Sierpinski Relatives*, Preuzeto 17. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/boxVariation.htm>
- [47] Riddle L (2013 g), *Classic Iterated Function Systems, Triangle Fractals*, Preuzeto 17. kolovoza 2013. s <http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/triangleVariation.htm>
- [48] Sedgewick R, Wayne K (2012), *Recursion*, Preuzeto 11. kolovoza 2013. <http://introcs.cs.princeton.edu/java/23recursion/>
- [49] Stone Design (s.a.), *StarMores*, Preuzeto 15. rujna 2013. s <http://www.stone.com/StarMores/>
- [50] Šimac M (2007), *Teorija kaosa – novi pogled na svijet*, Preuzeto 1. kolovoza 2013. s <http://zvjezdarnica.com/znanost/nas-planet/teorija-kaosa-novi-pogled-na-svijet/157>
- [51] Thomas G (2009), *Learning webGL, WebGL Lesson 2 – Adding colour*, Preuzeto 27. kolovoza 2013. s <http://learningwebgl.com/blog/?p=134>
- [52] Tribe (2006), *Sierpinski's Gasket : The Tetrix & Tetractys*, Preuzeto 15. rujna 2013. s http://people.tribe.net/white_wizard/blog/e04e79ce-b876-4e73-82d2-0ddd5f877136
- [53] Weisstein E (s.a.), *Hénon map*, Preuzeto 15. rujna 2013. s <http://mathworld.wolfram.com/HenonMap.html>
- [54] Wikipedia (2013 a), *WebGL*, Preuzeto 27. kolovoza 2013. s <http://en.wikipedia.org/wiki/WebGL>
- [55] Zucker M (2001), *The Perlin noise math FAQ*, Preuzeto 20. kolovoza 2013. s <http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html>