

Proceduralna generacija: Od naključnih števil do neskončnih svetov

Blaž Stojanovič

7. oktober 2019

Ali lahko ustvarimo neskončen svet, ne da bi za to porabili neskončno truda? Na prvi pogled se to zdi nemogoče. Kaj pa če sveta ne bi ustvarjali ročno, ampak algoritmično? Računalniško orodje, ki se ukvarja s takšnimi in podobnimi problemi, se imenuje *proceduralna generacija*.

Proceduralna generacija s pomočjo nekaj ročno zapisanih pravil in računalniško generiranega naključja ustvarja gromozanske količine vsebine. Praktično uporabo so metode našle predvsem v računalniški grafiki, kjer se uporabljajo za generacijo tekstur in v računalniških igrah. Zgodovinski mejnik za uporabe proceduralne generacije v računalniških igrah sta postavili igri *Beneath Apple Manor* (1978) in *Rogue* (1980), ki sta prvi na tak način ustvarjali svetove, v katerih se znajde igralec. Glavna prednost, ki sta jo pred svojimi tekmeci imel igri davnega leta 1980, je velikost pomnilnika. Ker svetov, ustvarjenih s pomočjo proceduralne generacije, ni treba spraviti v pomnilnik ampak jih lahko generiraš sproti, sta imeli *Beneath Apple Manor* in *Rogue* svetove večje, kot bi jih lahko spravili v pomnilnik v tistem času.

Dandanes se metode proceduralne generacije uporabljajo zelo pogosto, ustvarjanje iger na tak način ima kar nekaj praktičnih

prednosti. Ena izmed glavnih prednosti proceduralnih pristopov je doseganje velikih skal in majhnih podrobnosti. Tak pristop lahko prihrani ogromno časa, poslužujejo se ga igre z zelo velikimi svetovi, kot na primer *Skyrim*. Proceduralno generacijo uporabljajo na dva načina, proceduralna orodja lahko uporabijo za ustvarjanje grobega reliefa, ki ga potem dovršijo ročno, ali pa pokrajino ustvarijo ročno in potem malenkosti, kot so rastlinstvo, naselja in prebivalstvo generirajo proceduralno. Še ena prednost je modularnost, novo vsebino lahko v igro dodajamo z veliko manj truda, vsaka majhna sprememba pa se pozna na celotni skali igre. Poleg tega lahko en sam generator ustvari neskončno podobnih ampak še vseeno raznolikih svetov, kar močno zmanjša možnost, da se bo igralec igre naveličal. To mojstrsko izkoristi igra *Spelunky*.

Seveda ima proceduralna generacija kot vsaka metoda, tudi svoje slabosti. Zagotavljanje kakovosti postane težavno, sama metoda vpelje v igro negotovost in praktično nemogoče je preizkusiti vse izide. Tako vedno obstaja majhna možnost za nepredvidljivo in nezaželeno obnašanje igre. Poleg tega proceduralna generacija ne sme biti edina zanimiva reč v računalniški igri, saj se tudi neškončno velikih svetov da naveličati, če ni v njih nič iskanja vrednega. To je razlika med zelo uspešnim *Minecraft*-om in malo manj uspešnim *No Man's Sky*.

Proceduralna generacija ima zares dve plati, prva je delovanje samega algoritma, s katerim generiramo vsebino, druga pa je uporaba algoritma z jasnim namenom, naj bo to kot poseben efekt v filmski uspešnici ali pa kot pomemben del videoigre. Ustvarjanje vsebine s proceduralno generacijo je večna bitka med kaosom in dolgčasom; parametre generatorja želimo nastaviti tako, da dobimo nekaj, kar je dovolj naključno, da ni

dolgočasno/monotono, in ne tako naključno, da je nesmiselno. Grajenje proceduralnih sistemov je tako svojevrstna umetnost in bralci, ki jih to področje zanima, si lahko več o tem preberejo v izvrstni knjigi *Procedural Generation in Game Design* [1].

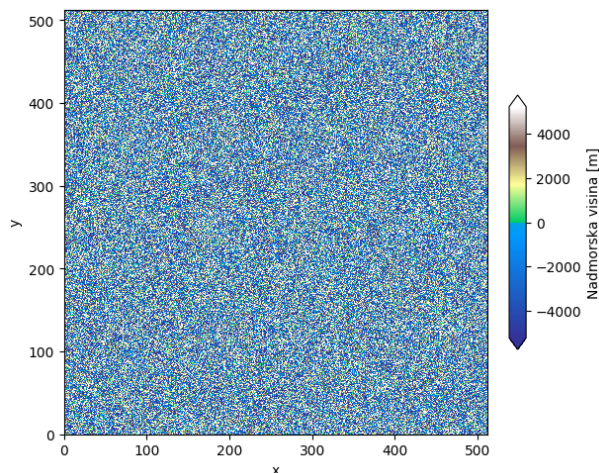
Gereriranje terena

Uporabo proceduralnih metod si bomo ogledali na najbolj osnovnem primeru, generiranju reliefov. Relief bomo predstavili kot višinsko karto (*angl. heightmap*), razpredelnico v kateri je v vsaki celici zapisana nadmorska višina.

1	2	2	4	10
0	1	2	1	6
-1	2	2	0	3
-2	0	0	3	6

Slika 1: Primer 5×5 višinske razpredelnice

Takšno razpredelnico lahko s procesom upodabljanja (*rendering*) spremenimo v 3-D sliko reliefa. Ker takšna razpredelnica vsebuje vse željene informacije o terenu, lahko na nov način definiramo problem. Zanima nas, kako ustvariti višinsko razpredelnico, da bo relief podoben nečemu kar bi srečali v naravi. Jasno je, da vrednosti nadmorske višine ne moremo kar naključno žrebati, saj tako dobimo nekaj nesmiselnega. Z naključnim žrebanjem se vrednosti lahko prehitro spreminjajo, kar lahko vodi do prevelike višinske razlike med sosednjimi točkami. Naprimer globoko morje in visokogorje, ki sta postavljena na sosednji točki. Treba je poiskati način, kako naključnost oblažiti.

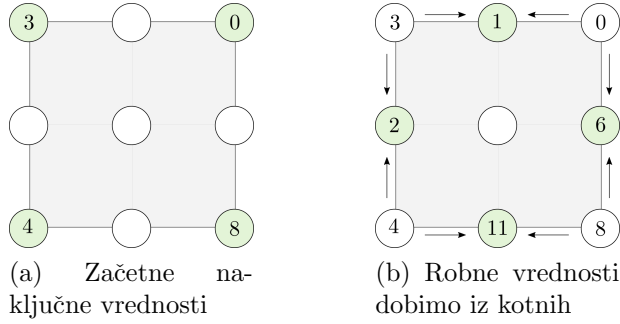


Slika 2: Popolnoma naključen relief

Midpoint displacement

Želimo si ustvariti relief, kjer obstaja možnost, da imata dve točki zelo različno nadmorsko višino, ampak ti dve točki ne smeta biti preveč blizu skupaj. Z drugimi besedami, želimo si relief, kjer je variacija v višini dveh točk manjša, če sta ti dve točki blizu skupaj, kot če sta daleč narazen.

Zelo preprost algoritem, ki poskrbi za take lastnosti reliefa, je *midpoint-displacement* [2]. Ta deluje tako da razpredelnico najprej polni na grobo, z vrednostmi, ki se lahko veliko razlikujejo, potem pa jo polni vedno bolj podrobno z vrednostmi, med seboj vedno manj razlikovale. Poglejmo kako natančno se to zgodi. Začnemo tako da naključno izberemo vrednosti v ogljiščih razpredelnice, katere velikost je enaka $2^n + 1$.



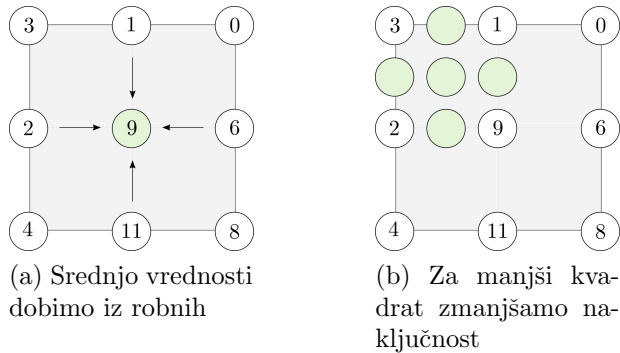
Slika 3

Potem pa ogljiščne vrednosti uporabimo za generiranje robnih vrednosti tako, da izračunamo povprečje dveh ogljiščnih vrednosti in mu dodamo neko naključno vrednost iz intervala $[-r, r]$. Na spodnjem robu slike 3b tako dobimo:

$$\text{spodnji rob} = (4 + 8)/2 + 5 = 11$$

Prost parameter r bomo imenovali *naključnost*, ki nadzoruje nagubanost reliefa. Večji kot je r , večja bo razlika med najvišjo in najnižjo točko na reliefu. Ko imamo vrednosti na robu kvadrata, lahko izračunamo še vrednost v sredini:

$$\text{sredina} = (2 + 1 + 6 + 11)/4 + 4 = 9$$

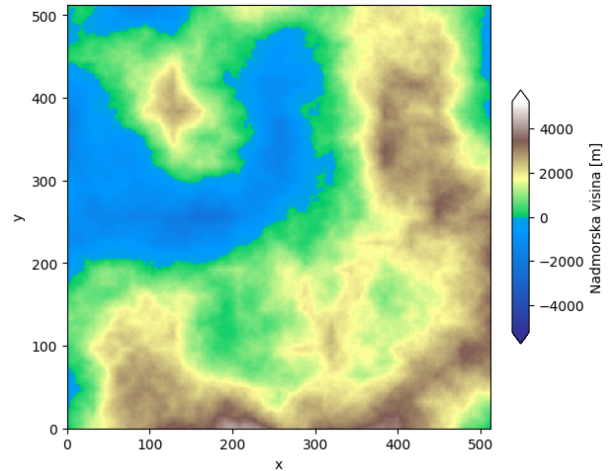


Slika 4

Ko izračunamo vrednost v sredini nadaljujemo z napolnitvijo manjših kvadratov.

Ključni korak pa je, da naključnost r zmanjšamo vsakič, ko polnimo manjši kvadrat. Če smo pri polnjenju velikega kvadrata na slikah 3a, 3b in 4a žrebali vrednosti na intervalu $[-5, 5]$, bomo pri polnjenju kvadrata na obarvanega na sliki 4b žrebali vrednosti na intervalu $[-2, 2]$. Tako poskrbimo, da bo razlika med vrednostmi v majhnem kvadratu manjša kot tista med vrednostmi v velikem kvadratu. Tako polnimo čedalje manjše kvadrate in v naš relief dodajamo vedno bolj fine podrobnosti. Relief ustvarjen, na zgoraj opisan način je predstavljen na sliki 5.

Seveda je tudi hitrost padanja naključnosti pomembna lastnost generatorja, če si želimo naprimer ustvariti strma gorovja, obkrožena z ravninami bomo, bomo v takem primeru r hitro zmanjševali.



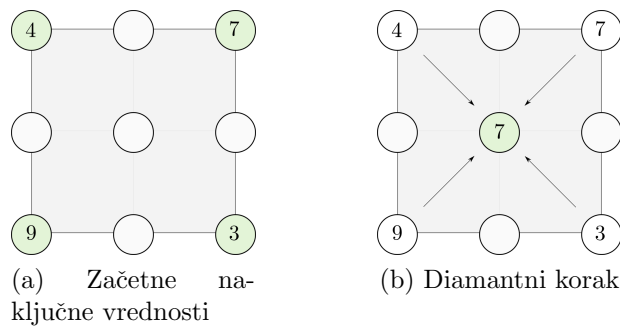
Slika 5: Relief ustvarjen z algoritmom midpoint displacement. Velikost razpredelnice je 1025×1025 , $r = 10250$, r se v vsaki iteraciji razpolovi in začetne ogljiščne točke so izžrebane iz intervala $[-205, 205]$.

Diamond Squares

Ko z midpoint displacement-om nekajkrat na tak način pri različnih semenih ustvarimo re-

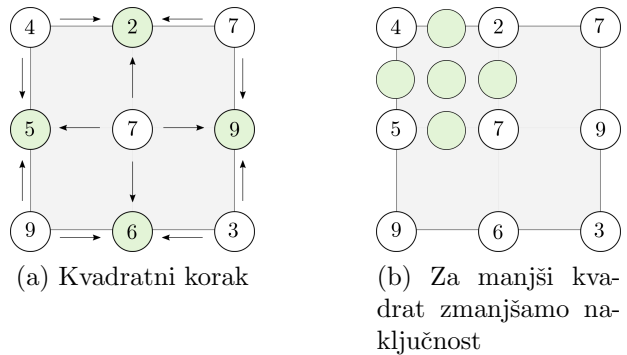
lief, opazimo, da nastanejo v reliefu razpoke. Te razpoke vedno potekajo navpično ali pa vodoravno. Takšne razpoke so nezaželjene, saj jih lahko nekdo z ostrim očesom opazi in ugotovi, da relief ni resničen. Generirano vsebino, ki ni zaželjena, imenujemo *artefakti*.

Izboljšava midpoint displacement-a, ki delno odpravi artefakte se imenuje diamond-squares algoritem. Ime sledi, iz diamantnega in kvadratnega koraka algoritma. Izkaže se da ni dobro, če vrednosti v robovih kvadratov računamo le iz dveh ogliščnih vrednosti.

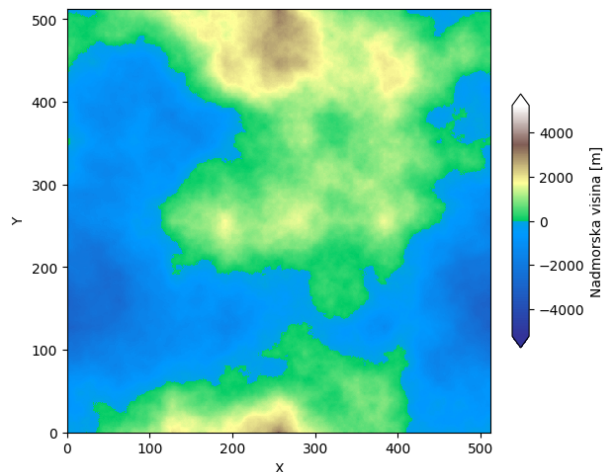


Slika 6

Lahko postopamo v drugem vrstnem redu, najprej iz vseh ogliščnih vrednosti izračunamo srednjo vrednost, to imenujemo diamantni korak. Potem pa s pomočjo srednje vrednosti izračunamo vrednosti na robovih kvadrata, kar imenujemo kvadratni korak. Na tak način se nikoli ne zanašamo na samo vrednosti pri generiranju nove. Algoritem je malce boljši od midpoint displacementa, ustvarjen relief je prikazan na sliki 8. Seveda obstajajo tudi nadaljnje izboljšave [3], ki se artefaktov znebijo tako, da pri računanju novih vrednosti iz starih le-te obtežijo. Uteži izračunajo iz majhnega linearne sistema, ki se ga dobi s pomočjo teorije približkov (angl. *approximation theory*).



Slika 7



Slika 8: Relief ustvarjen z algoritmom diamond-squares. Velikost razpredelnice je 1025×1025 , $r = 10250$, r se v vsaki iteraciji razpolovi in začetne ogliščne točke so izžrebane iz intervala $[-205, 205]$.

Perlinov šum

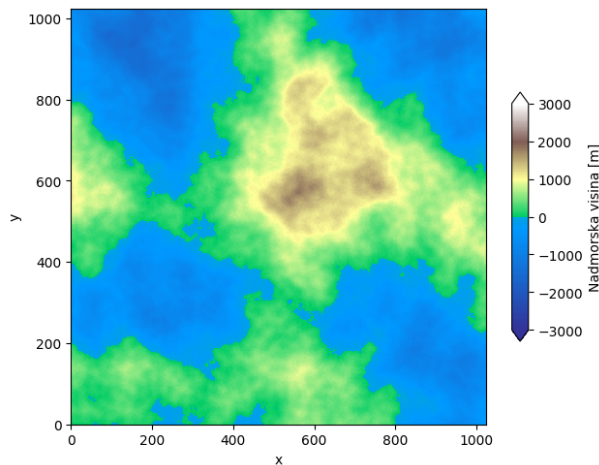
Za konec si oglejmo še, kako bi lahko ustvarili relief s posebno vrsto šuma, imenovano Perlinov šum. Šum je iznajdel Ken Perlin [4], ko je delal na posebnih učinkih za film Tron (1982). Za svojo iznajdbo je leta 1997 prejel tudi Oskarja, Perlinov šum in njegove izboljšave pa se še danes pogosto uporabljajo.

Ne bomo se spuščali v to, kako sam šum generiramo, osredotočili se bomo na njegovo uporabo. Vse kar moramo vedeti je, da gre za *koherenten* šum, to je šum, v katerem se spremembe v vrednostih od mesta do mesta dogajajo postopoma. Do realističnega terena bomo prišli tako, da bomo sešteli več sumov z različnimi frekvencami in amplitudami. Preden storimo to, pa se moramo naučiti terminologije v zvezi s Perlinovim šumom:

1. *Oktave*: Koliko šumov z različnimi frekvencami bomo uporabili. Posamezno oktavo označimo z naravnim številom, n .
2. *Lacunarity*: Kako se med oktavami spreminja frekvenca, koliko podrobnosti dodamo/odvzamemo pri vsaki oktavi. Označimo jo z l . Frekvenca v n -ti oktavi je enaka l^{n-1} .
3. *Persistenca*: Kako se med oktavami spreminja amplituda, koliko prispeva posamezna oktava. Označimo jo s p . Amplituda v n -ti oktavi je enaka p^{n-1} .

Glavna ideja generiranja tekstur s Perlinovim šumom je ta, da seštejemo med sabo več šumov, ki pa imajo različne frekvence in amplitude. Frekvenca pove, kako hitro se vrednosti šuma spreminjajo od točke do točke, medtem ko amplituda, kako velike so te spremembe. Šumi z nizkimi frekvencami in velikimi amplitudami ustvarijo makroskopske tvorbe, v našem primeru so to hribi, gore in doline. Šumi z visokimi frekvencami in majhnimi amplitudami pa ustvarijo majhne hitre spremembe površja in v našem primeru predstavljajo manjše skale in luknje na površju. Če želimo ustvariti realističen relief, morajo biti prispevki visokih frekvenc manjši kot prispevki nizkih frekvenc; gore in doline so večje, kot pa skale na njih. Od tod sledi, da če se frekvenca iz oktave v oktavo narašča ($l > 1$),

mora amplituda padati in mora tako biti $p < 1$. Relief ustvarjen s Perlinovim šumom je viden na sliki 9.



Slika 9: Relief ustvarjen s Perlinovim šumom. Velikost razpredelnice je 1024×1024 , uporabili smo osem oktav, persistenco 0.3 in lacunarity 4.

Zaključek

Metode proceduralne generacije smo ilustrirali na preprostem primeru ustvarjanja umečnih reliefov, a to še zdaleč ni vse kar proceduralna generacija lahko ponudi. Z njo lahko ustvarimo svetove polne rastlinstva in živalstva. Svetove lahko napolnimo z vasmi in mesti polnimi ljudi, vsak izmed njih pa ima lahko svojo videz in svojo zgodbo. Še več kot to, v našem svetu lahko dežuje in sneži, erozija pa počasi preoblikuje pokrajino. Seveda so možne tudi interakcije med prebivalci sveta, plenilci tako lovijo svoj plen, rastlinojedci pa jedo rastlinje. Metode proceduralne generacije nam tako omogočajo, da je edina stvar, ki nas omejuje pri ustvarjanju svetov, naša domišljija.

Literatura

- [1] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.
- [2] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.
- [3] Gavin SP Miller. The definition and rendering of terrain maps. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 39–48. ACM, 1986.
- [4] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

Midpoint displacement v Pythonu

```
import random as rand

# n - velikost stranice kvadratne mreze (privzamemo da je  $n = 2**a + 1$ )
# r - nakljucnost

rand.seed(0)
def midpoint_displacement(n, r, visina=256):

    # Naredimo mrezo
    mreza = [[0]*n for i in range(n)]

    # Nakljucno nastavimo kotne tocke
    mreza[0][0] = rand.randint(0, visina)
    mreza[n - 1][0] = rand.randint(0, visina)
    mreza[0][n - 1] = rand.randint(0, visina)
    mreza[n - 1][n - 1] = rand.randint(0, visina)

    # naredimo seznam tock, v katerih znamo izracunati nove vrednosti
    # shranjevali bomo stiri robne tocke in nakljucnost
    q = list()
    q.append((0,0,n-1,r))

    # dokler imamo na voljo tocke, racunamo srednjo vrednost
    while q:
        zgoraj, levo, spodaj, desno, r = q.pop(0)

        # izracunamo koordinate srednje tocke
        centerX = (levo + desno) // 2
        centerY = (zgoraj + spodaj) // 2

        # nastavimo vrednosti v sredinah stranic
        mreza[centerX][zgoraj] = (mreza[levo][zgoraj] + mreza[desno][zgoraj]) // 2 + rand.randint(-r, r)
        mreza[centerX][spodaj] = (mreza[levo][spodaj] + mreza[desno][spodaj]) // 2 + rand.randint(-r, r)
        mreza[levo][centerY] = (mreza[levo][zgoraj] + mreza[levo][spodaj]) // 2 + rand.randint(-r, r)
        mreza[desno][centerY] = (mreza[desno][zgoraj] + mreza[desno][spodaj]) // 2 + rand.randint(-r, r)

        # nastavimo vrednost v sredini kvadrata
        mreza[centerX][centerY] = (mreza[levo][zgoraj] + mreza[levo][spodaj]
        + mreza[desno][zgoraj] + mreza[desno][spodaj]) // 4 + rand.randint(-r, r)

        # Ce imamo prostor, dodamo kote novega kvadrata
        if (desno - levo) > 2:
            q.append((zgoraj, levo, centerY, centerX, r // 2))
            q.append((zgoraj, centerX, centerY, desno, r // 2))
            q.append((centerY, levo, spodaj, centerX, r // 2))
            q.append((centerY, centerX, spodaj, desno, r // 2))

    return mreza
```