Proceduralna generacija: od naključnih števil do neskončnih svetov

Blaž Stojanovič

→ **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? Računalnisko 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. Praktična uporaba te metode je predvsem v računalniški grafiki, kjer se uporablja za generacijo tekstur in v računalniških igrah. Zgodovinski mejnik uporabi 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 imeli 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 katerikoli pomnilnik tistega časa.

Ena izmed glavnih prednosti proceduralnih pristopov je doseganje velikih razsežnosti in majhnih podrobnosti. Tak pristop lahko prihrani ogromno časa, poslužujejo se ga igre z zelo velikimi svetovi, kot npr. *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 rastlinje, naselja in prebivalstvo, generirajo proceduralno. Še eno 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, vseeno še 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 neskončno velikih svetov lahko naveličamo, če ni v njih nič zanimivega. To je razlika med zelo uspešnim *Minecraftom* in malo manj uspešnim *No Man's Sky*.

Proceduralna generacija ima 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. Ustvarjenje 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 svojevrstna umetnost in bralci, ki jih to področje zanima, si lahko več o tem preberejo v izvrstni knjigi *Procedural Generation in Game Design* [1].

Generiranje 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

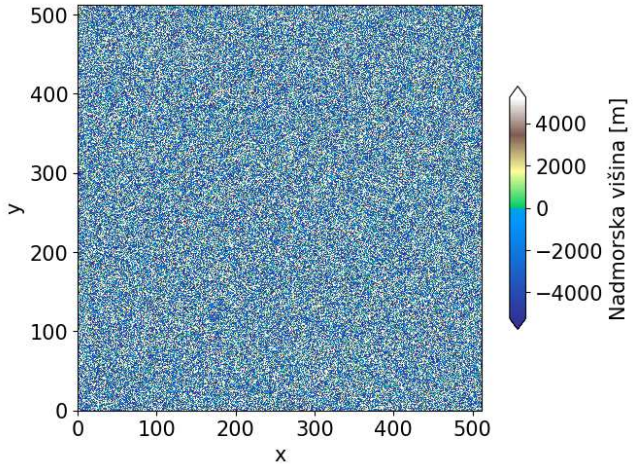
TABELA 1.
Primer 5×5 višinske razpredelnice

Takšno razpredelnico lahko s procesom upodabljanja (*rendering*) spremenimo v 3D sliko reliefa. Ker takšna razpredelnica vsebuje vse želene 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 (npr. globoko morje in visokogorje, ki sta postavljena na sosednji točki, vidno na sliki 1). Treba je poiskati način, kako naključnost oblažiti.

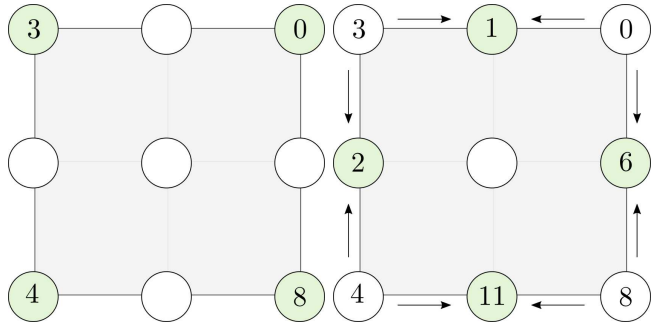
Pomik središča

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 razlika 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 pomik središča (angl. *midpoint-displacement*) [2]. Ta deluje tako, da razpredelnico najprej polni na grobo, z vrednostmi, ki se lahko zelo razlikujejo, potem pa jo polni vedno bolj podrobno z vrednostmi, med seboj vedno manj razlikovale. Poglejmo, kako natančno se to zgodi. Začnemo s kvadratno razpredelnico, ki ima stranico dolgo $2^n + 1$; takšna dolžina stranice olajša deljenje razpredelnice na kvadratne podrazpredelnice. Vse skupaj bomo ilustrirali za primer, kjer je $n = 2$. Najprej naključno izberemo vrednosti v ogliščih razpredelnice, te bomo izžrebali kar iz intervala $[0, 10]$.



SLIKA 1.
Popolnoma naključni relief.



SLIKA 2.
Prvi in drugi korak pomika središča. Levo: Začetne naključne vrednosti, desno: robne vrednosti dobimo iz kotnih.

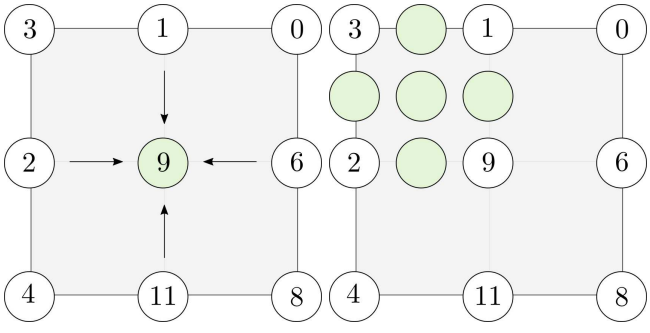
Oglišče vrednosti bomo uporabili za generiranje vrednosti na robovih, ki povezujejo pare oglišč. To

storimo tako, da izračunamo povprečje vrednosti v ogliščih in povprečni vrednosti dodamo še naključno vrednost iz intervala $[-r, r]$. Izračunajmo vrednost na spodnjem robu za primer, prikazan na sliki 2, če vzamemo $r = 5$. V ogliščih imamo vrednosti 4 in 8, potem pa naključno izžrebamo 5, vrednost v sredini spodnjega roba tako znaša:

- spodnji rob $= (4 + 8)/2 + 5 = 11$.

Prosti parameter r bomo imenovali *naključnost*, ki nadzoruje razgibanost 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. Postopamo enako kot prej, le da sedaj upoštevamo vrednosti iz robov. Izračunamo povprečje števil $1, 6, 11$ in 2 in mu dodamo naključno vrednost iz intervala $[-5, 5]$, npr. 4. Tako dobimo:

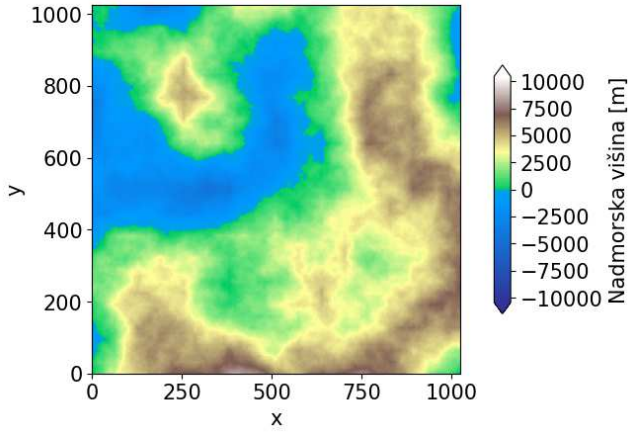
- sredina $= (2 + 1 + 6 + 11)/4 + 4 = 9$.



SLIKA 3.
Tretji in četrti korak pomika središča. Levo: Srednjo vrednosti dobimo iz robnih, desno: tretji in četrti korak pomika središča.

Ko izračunamo vrednost v sredini, nadaljujemo z napolnitvijo manjših kvadratov. Pomembno pa je, da naključnost r zmanjšamo vsakič, ko polnimo manjši kvadrat. Če smo pri polnjenju velikega kvadrata na slikah 2 levo, 2 desno in 3 levo žrebali vrednosti na intervalu $[-5, 5]$, bomo pri polnjenju kvadrata, na obarvanega na sliki 3 desno, žrebali vrednosti na intervalu $[-2, 2]$. Na ta način poskrbimo, da bo razlika med vrednostmi v majhnem kvadratu manjša kot tista med vrednostmi v velikem kvadratu. Tako polnimo čedalje manjše kvadrate in v relief dodajamo vedno večje podrobnosti. Relief ustvarjen na zgoraj opisani način je predstavljen na sliki 4.

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



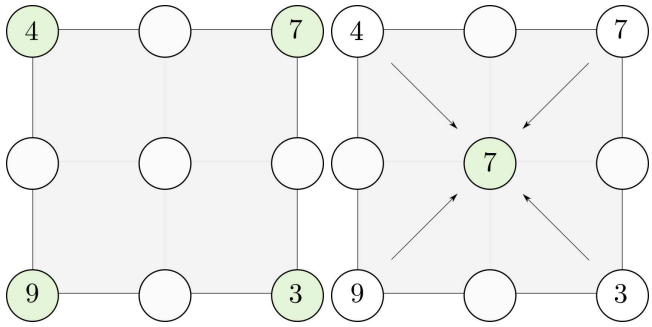
SLIKA 4.
Relief, ustvarjen z algoritmom pomika središča. Velikost razpredelnice je 1025×1025, $r = 10250$, r se v vsaki iteraciji razpolovi in začetne ogliščne točke so enakomerno naključno izžrebane iz intervala [-205, 205].

Karo-kvadrat

Ko s pomikom središča nekajkrat na tak način pri različnih semenih ustvarimo relief, opazimo, da nastanejo v reliefu razpoke. Te razpoke vedno potekajo navpično ali pa vodoravno. Takšne razpoke so nezaželene, saj jih lahko kdo z ostrim očesom opazi in ugotovi, da relief ni resničen. Generirano vsebino, ki ni zaželena, imenujemo *artefakti*.

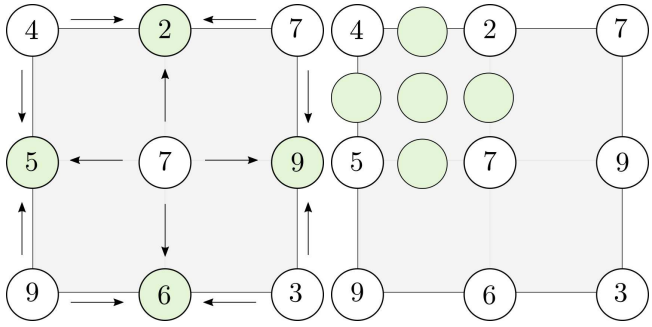
Izboljšava pomika središča, ki delno odpravi artefakte, imenujemo karo-kvadrat (angl. *diamond-square*) algoritem. Ime izvira iz karo in kvadratnega koraka algoritma. Izkaže se, da ni dobro, če vrednosti v robovih kvadratov računamo le iz dveh ogliščnih vrednosti.

Lahko postopamo v drugem vrstnem redu. Najprej iz vseh ogliščnih vrednosti izračunamo srednjo vrednost, to imenujemo karo korak. Potem pa s pomočjo srednje vrednosti izračunamo vrednosti na



SLIKA 5.
Prvi in drugi korak algoritma karo-kvadrat. Levo: Začetne naključne vrednosti, desno: karo korak.

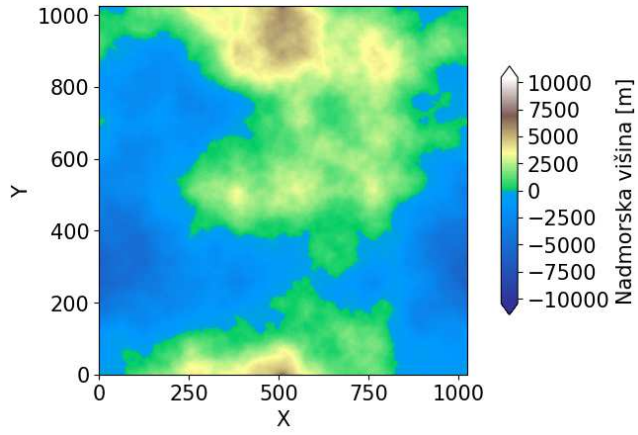
robovih kvadrata, kar imenujemo kvadratni korak. Na tak način se nikoli ne zanašamo na samo dve vrednosti pri generiranju nove. Algoritem je malce boljši od pomika središča, ustvarjeni relief je prikazan na sliki 7. Seveda obstajajo tudi nadaljnje izboljšave [3], ki se artefaktov znebijo tako, da pri računanju novih vrednosti iz starih le-te obtežijo. Uteži izračunajo iz majhnega linearnega sistema, ki ga dobijo s pomočjo teorije približkov (*angl.* approximation theory).



SLIKA 6.
Tretji in četrti korak algoritma karo-kvadrat. Levo: Kvadratni korak, desno: za manjši kvadrat zmanjšamo naključnost

Perlinov šum

Za konec si oglejmo še, kako bi lahko ustvarili relief s posebno vrsto šuma, imenovano Perlinov šum. Šum je iznašel Ken Perlin [4], ko je delal posebne učinke za film Tron (1982). Za svojo iznajdbo je leta 1997



SLIKA 7.
Relief, ustvarjen z algoritmom karo-kvadrat. Velikost razpredelnice je 1025×1025, $r = 10250$, r se v vsaki iteraciji razpolovi in začetne ogliščne točke so enakomerno naključno izžrebane iz intervala [-205, 205].

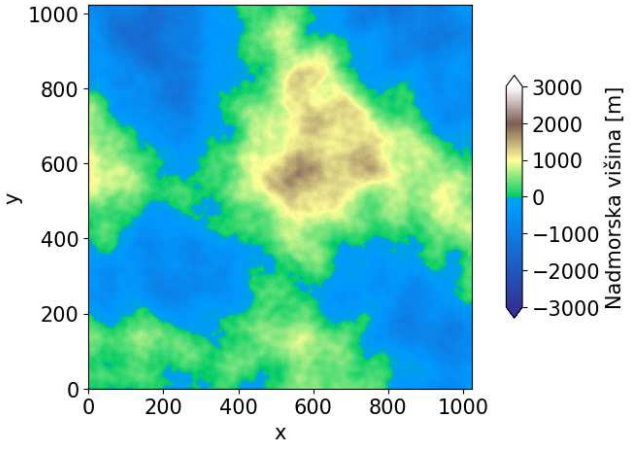
prejel tudi Oskarja, Perlinov šum in njegove izboljšave se še danes pogosto uporabljajo.

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 šumov z različnimi frekvencami in amplitudami. Preden storimo to, pa se moramo naučiti terminologije v zvezi s Perlinovim šumom:

- *Oktave.* To pomeni, koliko šumov z različnimi frekvencami bomo uporabili. Posamezno oktavo označimo z naravim številom, n.
- *Lakunarnost.* To pomeni, 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}.
- *Persistenca.* To pomeni, 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 pove, 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 vidimo na sliki 8.



SLIKA 8.
Relief, ustvarjen s Perlinovim šumom. Velikost razpredelnice je 1024×1024, uporabili smo osem oktav, persistenco 0.3 in lakunarnost 4.

Zaključek

Metode proceduralne generacije smo ilustrirali na preprostem primeru ustvarjanja umetnih 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.

Implementacijo algoritmov karo-kvadrat in pomik središča v programskem jeziku Python ter kodo, potrebno za ustvarjanje slik, lahko najdete na `github.com/BlazStojanovic/ProceduralnaGeneracija`. Vsekakor predlagamo, da se poigrate s parametri algoritmov in poskusite ustvariti čim bolj zanimive reliefe tudi sami.

Literatura

[1] N. Shaker, J. Togelius in M. J. Nelson, *Procedural content generation in games*, Springer, 2016.

[2] A. Fournier, D. Fussell in L. Carpenter, *Computer rendering of stochastic models*, Communications of the ACM, 25(6): 371–384, 1982.

[3] G. S. P. Miller, *The definition and rendering of terrain maps*, In ACM SIGGRAPH Computer Graphics, volume 20, 39–48, ACM, 1986.

[4] K. Perlin, *An image synthesizer*, ACM Siggraph Computer Graphics, 19(3): 287–296, 1985.

Barvni sudoku

REŠITEV BARVNI SUDOKU

4	**6**	1	8	**5**	3	**7**	**2**
5	**3**	**2**	**7**	**1**	**6**	**4**	**8**
1	**5**	7	**2**	**3**	8	**6**	**4**
6	**8**	**3**	**4**	**7**	**2**	1	**5**
2	**4**	**5**	3	**6**	1	**8**	7
8	**7**	**6**	1	4	**5**	2	**3**
7	**1**	**8**	**5**	2	**4**	**3**	6
3	2	4	**6**	8	**7**	**5**	**1**