Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Peter Gabrovšek

Proceduralno generiranje terena s Perlinovim šumom

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Saša Divjak
Somentor: doc. dr. Matija Marolt

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomskem delu preučite področje proceduralnega generiranja terena in tekstur. Izdelajte odprtokodno programsko knjižnico za generiranje terena in tekstur, ki naj temelji na uporabi Perlinovega šuma. Knjižnico evaluirajte v smislu časovne in prostorske zahtevnosti.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Peter Gabrovšek sem avtor diplomskega dela z naslovom:

Proceduralno generiranje terena s Perlinovim šumom

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Saše Divjaka in somentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 3. septembra 2015 Podpis avtorja:

Družini.

Kazalo

Povzetek

V okviru diplomskega dela smo ustvarili odprtokodno programsko knjižnico, ki nam omogoča proceduralno generiranje tekstur in terena. V zadnjem času opažamo skokovit razvoj in širitev obravnavanega področja. Predstavili smo principe in orodja, ki jih potrebujemo za implementacijo takega terena. Potrebujemo tako matematične pristope, kot je uporaba vektorskih in skalarnih polj ter njihovih odvodov, kot tudi nizkonivojsko uporabo grafičnih kartic, na katerih je izvajanje našega programa in grafike na sploh zaradi svojih lastnosti najbolj optimalno. Predstavili smo tudi pozitivne in negativne lastnosti takega pristopa generiranja terena, kjer smo spoznali, da več kot očitno prevladujejo pozitivne lastnosti.

Opisali smo rešitve za težave, ki se nam pojavljajo pri tovrstnem izdelovanju iger in aplikacij za vizualne predstavitve okolice ter ponudili rešitve za lažje soočanje s temi težavami.

V programski knjižnici, ki je rezultat tega diplomskega dela, smo se osredotočili na proceduralno generiranje tekstur in terena z uporabo nekaterih orodij in tehnik predvsem pa z implementacijo različnih tipov šumov.

Končni izdelek se zdi privlačen in predstavlja neko zaključeno celoto in je uporaben, dopušča pa številne možnosti nadaljnega razvoja.

Ključne besede: Proceduralno generiranje terena, Perlin, šum, igre.

Abstract

Within this thesis we have created open source software library which enables procedural terrain generation. In recent times, one can observe fast expansion of the field which is a subject of this thesis. We have presented principles and tools needed for implementation of procedurally generated terrain. Mathematical approaches, such as usage of vector and scalar fields and their derivatives, are also needed. The same goes for low-level usage of graphic cards for executing our program. Due to its characteristics, such approach gives optimal results. We presented both, positive and negative properties of procedurally generated terrain and came to a conclusion that the positive properties are prevailing in most cases.

We described problems and their solutions that that can be encountered during the process of making games and applications for visual presentations of region.

In the software library, which is the result of this thesis, we focused on procedurally generated textures and terrain by implementing certain tools and techniques. Different kinds of noises were mostly used.

The work was quite complex, but the end results seems to be quite attractive and represents a product which can be used in practice but also leaves a space for further development.

Keywords: Procedural terrain generation, Perlin, noise, games.

Poglavje 1

Uvod

Čeprav je že igra *Rogue*[1] okoli leta 1980 uporabljala proceduralno generiranje [13], šele sedaj razvijalci iger bolj pogosto posegajo po tem pristopu. Razlog je v tem, da je igro lažje oblikovati z orodjem za oblikovanje iger [5], kot pa napisati dober algoritem za avtomatično generiranje tekstur, terena in drugih objektov v igri.

Trenutno je potencial pri računalniški grafiki v funkcijskem ali proceduralnem generiranju terena in celotne okolice. Obstaja že veliko iger, ki so narejene po tem principu, recimo *Minecraft*, *No Man's Sky*, *Spore*, *Binding of Isaac*, *Don't Starve* in mnoge druge.

Slika 1.1: Minecraft, Binding of Isaac, Don't Starve

Tudi vsakdanji računalniki so že dovolj zmogljivi, da med samim delovanjem generirajo teren, ki je zelo podoben realnemu. V prihodnosti pa lahko zelo verjetno pričakujemo še večji razvoj v tej smeri, vsaj glede na trend,

[1] `https://en.wikipedia.org/wiki/Rogue_(video_game)`

ki ga lahko zasledimo na svetovnem spletu. Obstajajo spletne strani, ki ponujajo izdelavo prototipov senčilnikov, rezultat pa nemudoma vidimo na zaslonu. Primeri takih spletnih strani: glslsandbox[2], shadertoy[3] in chrome-experiments[4]. Poleg naštetih spletnih strani pa lahko poiščemo tudi mnogo drugih pod iskalnim terminom "shader sandbox" v spletnem iskalniku *Google*, ki ponujajo že v naprej pripravljene senčilnike, katerih rezultat so zelo zanimivi učinki, med drugimi tudi učinki, ki dajejo vtis naravnih struktur. Funkcijsko generiran teren je praktično neskončen in nismo omejeni z vnaprej pripravljenim in oblikovanim terenom. Če že imamo teren in se nam ni potrebno ukvarjati z oblikovanjem le-tega, se lahko bolj posvetimo sami igralnosti, kar pripomore k večji kvaliteti iger ali k hitrejšemu razvoju.

Poleg časa, ki ga privarčujemo, če je teren generiran funkcijsko, pa privarčujemo tudi pri prostoru, saj moramo hraniti le opis terena in ne terena samega. Dejansko ni pomembno kako velik je teren, pri proceduralnem generiranju je količina podatkov za generiranje vedno enaka, medtem ko pa pri ročno generiranemu terenu količina samih podatkov in s tem velikost igre zelo hitro narašča, še posebej, če želimo prikazati veliko podrobnosti.

Pri funkcijsko generiranem terenu pa nastopi težava, ko želimo imeti področje točno določene oblike. Takrat moramo teren oblikovati ročno in ga shraniti v pomnilnik.

Računalništvo se bolj in bolj seli v oblak, zato tudi igre niso izjema, saj si želimo, da bi do svojih iger lahko dostopali od kjerkoli. Poznamo že veliko programov in storitve v oblaku: *Steam*, *Good Old Games* in druge. Ker pa še nima vsakdo dovolj hitre internetne povezave, da bi lahko obsežne igre prenesel na svoj računalnik v zadovoljivem času, je rešitev v igrah, kjer se njeni sestavni deli generirajo proceduralno. Take igre ne zasedejo veliko prostora in se zato hitreje prenesejo, obseg take igre pa je lahko večji od ostalih iger.

Z vedno hitrejšim razvojem proceduralno generiranih iger se odpirajo nove

[2]http://glslsandbox.com/

[3]https://www.shadertoy.com/

[4]https://www.chromeexperiments.com/experiment/glsl-sandbox

možnosti igranja, saj je že trenutno stanje osupljivo. Takšen primer je igra *No Man's Sky*, ki vsebuje celotno vesolje različnih planetov, na katerih so vse vrste pokrajin in pojavov, na teh planetih pa se najde tudi prenekatero vrsto živali, da o rastlinah sploh ne govorimo.

Slika 1.2: Proceduralno generiran teren, rastline in živali v igri *No Man's Sky*, vir: `http://www.no-mans-sky.com/`

Cilj te diplomske naloge je ustvariti odprtokodno programsko knjižnico, ki bo omogočala enostavno proceduralno generiranje terena, ki bi ga lahko kdorkoli uporabil pri izdelovanju iger. Na tem mestu velja pripomniti, da se proceduralno generiranja terena množično uporablja pri izdelovanju iger, vse bolj pa prodira tudi v druge panoge, kot so arhitektura, animirani in reklamni filmi, geologija itd.

Naš teren je sestavljen iz travnatega in peščenega območja. Vsako območje je sestavljeno iz drugačnih tekstur, barv in oblik, za čimbolj pristen učinek.

Z nekaj parametri lahko enostavno prilagodimo teren svojim potrebam. Lahko nastavljamo stopnjo hribovitosti in razgibanosti pokrajine, nastavljamo lahko barvne prelive področij in njihovo velikost. Na primer: z manjšanjem travnatih področij lahko ustvarjamo oaze v puščavi, z manjšanjem peščenih območij pa dobimo prikupno igrišče za golf.

Dandanes si marsikdo želi narediti svojo igro, saj je na voljo veliko brezplačnih orodij, ki to omogočajo. Trenutno zelo priljubljen in precej zmogljiv igralni pogon *Unity* se lahko primerja z marsikaterim drugim plačljivim orod-

jem, kot je na primer *Unreal Engine*.

Unity je enostaven za uporabo in zanj obstaja veliko kvalitetnih učnih gradiv in virov. Zato smo se odločili, da bomo našo programsko knjižnico za proceduralno generiranje terena implementirali v njem. Naš namen je, da bi lahko razvijalci in oblikovalci iger uporabili našo knjižnico in se bolj usmerili v samo igralno izkušnjo ter si prihranili veliko časa in težav.

Poglavje 2

Orodja in tehnike za proceduralno generiranje terena

Pri proceduralnem oziroma funkcijskem pristopu generiranja okolice se uporablja veliko matematičnih prijemov [21]. Uporablja se šum [9], fraktale [19], matematične funkcije na splošno, postopke umetne inteligence ... Za prehod med področji uporabljamo prehode s prelivom.

Ob upoštevanju trenutnega razvoja proceduralnega generiranja smo se osredotočili na uporabo Perlinovega šuma, belega šuma in prehodov s prelivom. Za razgibanost terena smo uporabili višinske preslikave in teksture z normalami. Naš namen je ustvariti teren, ki bo lahko kasneje uporabljen pri izdelovanju iger in drugih vizualnizacijsko usmerjenih aplikacij. Za implementacijo le-tega smo uporabili igralni pogon *Unity*, v katerem programiramo v programskem jeziku C#.

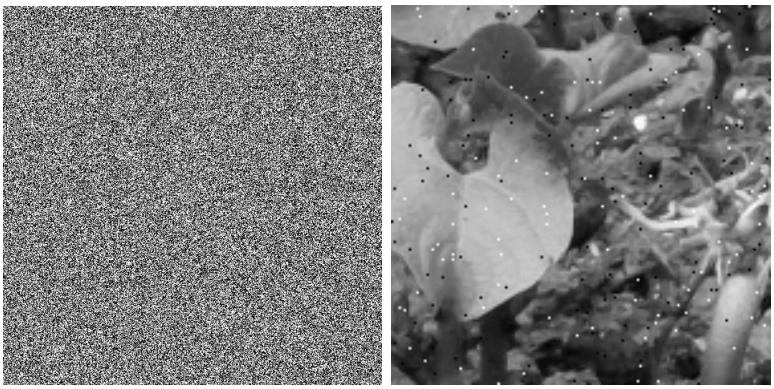
Najprej bomo na kratko opisali bolj pogosto uporabljena orodja in tehnike, nato pa bomo še bolj podrobno pristope, ki jih uporabljamo v tem delu.

2.1 Splošno uporabljena orodja in tehnike

2.1.1 Šum

Šum [4] je prisoten na veliko področjih fizike in tehničnih ved. Pojavlja se v prostorih s poljubnim številom dimenzij. Primer šuma v eni dimenziji je zvočni šum. V dveh dimenzijah nam je najbolj poznan v obliki snega, ki se pojavi na zaslonu televizije, ko ni signala. To je primer belega šuma. Do takega šuma prihaja zaradi različnih zunanjih dejavnikov, ki vpivajo na stanje določenega sistema. Matematično modeliranje takšnega šuma je potrebno za razvoj postopkov odstranjevanja šuma, ki so posledica zunanjih motenj.

Po drugi strani se lahko tako razviti modeli šuma uporabljajo tudi v drugačne namene, kot je na primer naš. Poleg prej omenjenih vrst šumov pa poznamo še veliko drugih vrst šuma, kot so roza šum, simpleksni, sol in poper in drugi ter seveda Perlinov šum, ki je glavna tema te diplomske naloge.



Slika 2.1: Beli šum ter šum sol in poper

Šum [25] je pojav, ki predstavlja signal, katerega vrednost v poljubni koordinati zavzema naključno vrednost. Drugače povedano, frekvenčni spekter šuma predstavlja neko bolj ali manj zvezno ter uniformno porazdelitev. Frekvenčni spekter pa se od šuma do šuma razlikuje. Beli šum[1] [7] ima na primer res uniformno porazdelitev frekvenc[2], pri Perlinovem šumu pa se po-

[1] http://www.its.bldrdoc.gov/fs-1037/dir-024/_3556.htm

[2] http://www.its.bldrdoc.gov/fs-1037/dir-040/_5873.htm

javi zgostitev pri frekvenci f blizu frekvence, ki je obratno sorazmerna z dolžino celice mreže, s pomočjo katere ustvarimo šum. Bolj kot se frekvence oddaljujejo od frekvence f, manjša je njihova zastopanost [8].

2.1.2 Fraktali

Fraktali [22, 10] so naravni pojavi oziroma matematične množice, ki predstavljajo isti ali podoben ponavljajoč se vzorec pri katerikoli povečavi. Čeprav si definicije fraktalov različnih avtorjev nasprotujejo, bi jih lahko opisali kot grobe ali razdrobljene geometrijske oblike, ki kažejo značilnosti samopodobnosti. Fraktale lahko ustvarimo na več načinov, en izmed njih pa so l-sistemi [16, 23].

L-sistemi v računalniški grafiki igrajo pomembno vlogo. Razviti so bili za opis bioloških sistemov, čeprav lahko z njimi opišemo tudi marsikaj drugega. L-sistemi so definirani z gramatiko, ki je pravzaprav sestavljena iz začetnega stanja in množic spremenljivk, konstant in pravil, po katerih se sistem gradi.

Mengerjeva spužva, Kochova snežinka in Mandelbrotova množica so primeri matematično izračunanih fraktalov. Fraktali pa se pojavljajo tudi v naravi in jih lahko opazujemo na primer pri cvetači, ananasu, storžih, snežinkah itd. Če fraktale uporabljamo z naključnostjo pa dobimo fraktalne šume, s katerimi lahko ustvarimo vzorce podobne celicam, tlakovanim potem, lesu ... Fraktalni šumi so mešanica med fraktali in šumom. Niso popolnoma pravilnih oblik, kot jih dobimo pri fraktalih, niso pa tudi popolnoma naključni, kot pri šumu.

2.1.3 Superformula

Pri proceduralnem generiranju se ponekod uporablja tudi t.i. superformula [12]. Z njeno pomočjo lahko ustvarjamo oblike primitivnih živih bitij, kot so bakterije ali morske živali, lahko pa ponazarjajo tudi pelod, cvetni prah.

Slika 2.2: Drevesa ustvarjena s pomočjo l-sistemov. vir: `https://en.wikipedia.org/wiki/L-system`

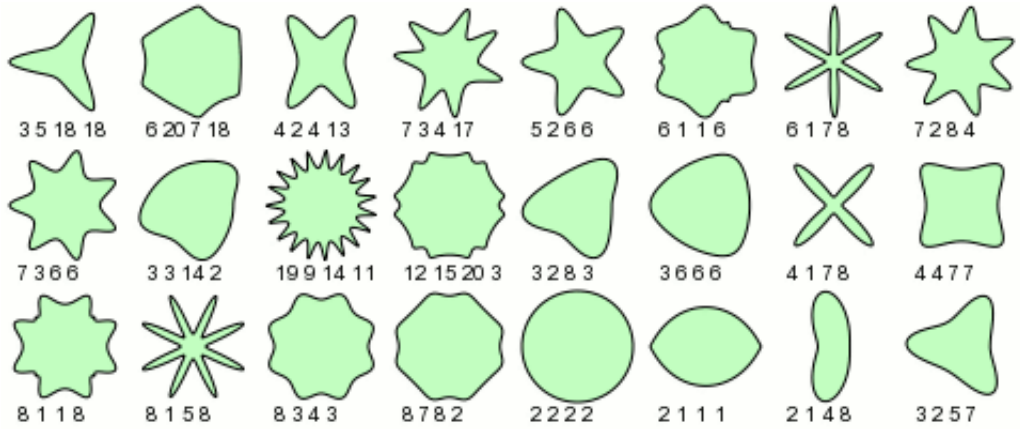
$$r(\varphi) = \left[\left| \frac{cos(\frac{m\varphi}{4}))}{a} \right|^{n_2} + \left| \frac{sin(\frac{m\varphi}{4}))}{b} \right|^{n_3} \right]^{-\frac{1}{n_1}} \tag{2.1}$$

Enačba (2.1) je superformula podana v parametrični obliki. Če spreminjamo koeficiente a, b, m, n_1, n_2, n_3, dobimo raznolike oblike, kot jih vidimo na sliki (spodaj). Na spodnji sliki so liki, kjer sta parametra a in b fiksna in enaka 1, φ pa je na intervalu $[0, 2\pi)$.

2.1.4 Senčilniki

Senčilniki so kratki namenski programi, ki tečejo na grafičnih karticah. Zaradi lastnosti računalniške grafike, je primerno paralelno programiranje. S tem namenom so pravzaprav ustvarjene grafične kartice. Grafične kartice imajo veliko jeder, ki so počasnejša od jeder v procesorju, toda jeder je več, zato je skupna hitrost računanja računalniške grafike z grafičnimi karticami večja, kot računanje na procesorju.

Senčilnikov v tej diplomski nalogi sicer ne uporabljamo, toda bili bi dobrodošla rešitev, ki bi pohitrila izrisovanje terena.



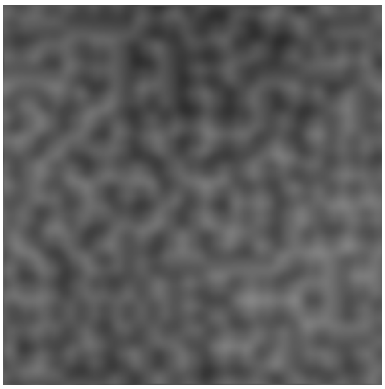
Slika 2.3: Primeri oblik ustvarjenih s *superformulo*, vir: `https://en.wikipedia.org/wiki/Superformula`

2.2 Perlinov šum

Ken Perlin je sodeloval pri ustvarjanju filma Tron, kjer je moral proceduralno generirati teksture. Te teksture je generiral s pomočjo šuma, ki so ga poznali tedaj, vendar mu je deloval preveč umetno, zato je leta 1982 za ta namen izumil Perlinov šum. Vizualni učinek Perlinovega šuma je bolj naraven in zelo lepo predstavi naravnim podobne strukture [14]. Bolj natančno, lepo opiše tako teksture kot tudi teren. Teren, zgrajen s pomočjo Perlinovega šuma, spominja na hribovitega. Za bolj ekstremen teren, kot je denimo kras ali pa so gorovja, bi bilo potrebno poseči po drugih orodjih, saj z dvodimenzionalnim Perlinovim šumom ne moremo ustvariti jam, strmih sten ali previsov.

S Perlinovim šumom dosežemo precej boljše učinke, kot če bi uporabili katerega od drugih, enostavnejših šumov. Še boljši učinek pa dosežemo s tako imenovano turbulenco [1, 11], kjer združujemo posamezne plasti Perlinovih šumov različnih frekvenc. V tem diplomskem delu uporabljamo 6 plasti različnih frekvenc. 3 različne plasti uporabimo pri višinskih preslikavah, 3 pa pri lepljenju normal, toda več o tem v poglavju, kjer opisujemo implementacijo generatorja terena.

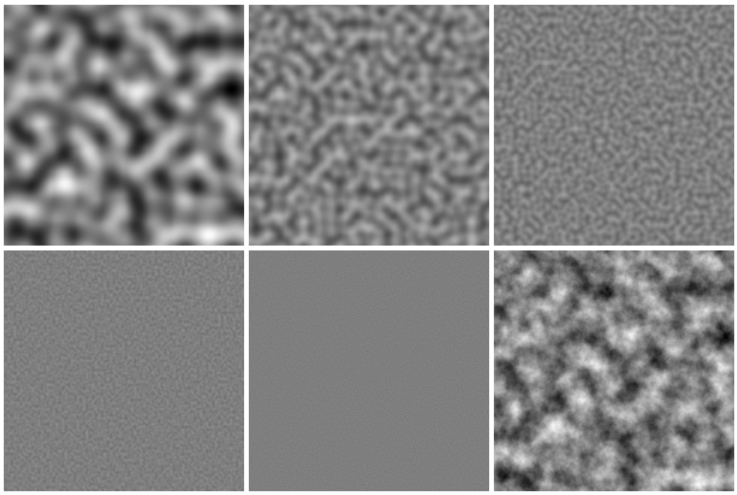
Podoben pojav najdemo tudi na področju zvoka, kjer osnovni frekvenci



Slika 2.4: Osnovni Perlinov šum

zvoka dodamo višje harmonike, celoštevilske mnogokratnike, in tako dobimo prijetnejši, naravnejši zven [26] namesto nadležnega piska. Kot v zvoku, tudi pri zlivanju različnih plasti Perlinovega šuma dobimo bolj prijeten in naraven učinek.

V naravi gorovja predstavljajo zelo nizke frekvence spreminjanja področja, hribovje in gričevje višjega, še višje frekvence pri spreminjanju področja pa najdemo v skalah, kamnih ... Ravno zaradi turbulence, s tem pristopom generiran teren dobi tako naraven učinek.



Slika 2.5: Posamezne plasti z različnimi frekvencami in spodaj desno vsota le-teh

Kako pa Perlinov šum[3] sploh deluje? Recimo, da ga želimo generirati v n dimenzijah. Perlinov šum se tipično generira v treh korakih. Najprej ustvarimo n dimenzionalno mrežo in vsaki točki te mreže določimo naključno usmerjen enotski vektor – vektor gradienta [2], znotraj posamezne celice za vsako točko izračunamo 2^n skalarnih produktov – za vsako oglišče, na koncu pa še interpoliramo rezultate skalarnih produktov in dobimo končno vrednost šuma v določeni točki.

Za začetek ustvarimo n dimenzionalno gradientno **mrežo**. S to mrežo ustvarimo n dimenzionalne kocke, s stranicami dolžine 1. Na ogliščih t teh kock bomo inicializirali naključno usmerjene normalne vektorje g_i. To so t.i. gradientni vektorji. Gradient je diferencialna operacija, nad vektorskim ali skalarnim poljem in nam pove, v kateri smeri se polje v dani točki najbolj spreminja. V naravi si lahko kot gradient predstavljamo veter, katerega vektorsko polje je zračni tlak. Smer pihanja vetra nam pove, v katero smer se zračni tlak najbolj spreminja, niža.

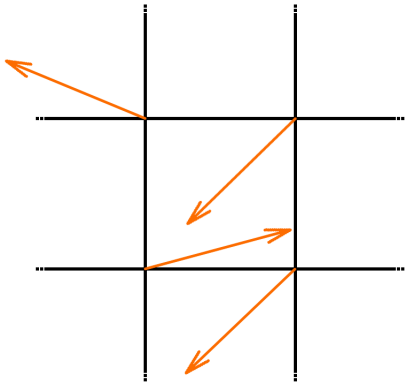
$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \tag{2.2}$$

Matematično se gradient zapiše kot operator nabla [24] nad skalarnim poljem f. To skalarno polje predstavlja funkcijo, ki vsaki točki v prostoru pripiše neko skalarno vrednost. $f(k)$ je skalarno polje, ki je odvisno od krajevnega vektorja $k = (x, y, z)$, komponente vektorja k pa predstavljajo parcialne odvod po vsaki od koordinat.

V primeru Perlinovega šuma ne potrebujemo iz skalarnega polja izračunati gradienta, vendar uporabimo obratni pristop. V ogliščih celic gradientne mreže naključno izberemo vektor in s tem dobimo gradientne vektorje. Glede na te vektorje v nadaljnjem postopku izračunamo končne vrednosti.

Naslednji korak je računanje **skalarnih produktov** za točko, v kateri želimo izračunati vrednost šuma. Za vsako oglišče t_i celice, v kateri je točka x_j, ki ji želimo izračunati vrednost šuma, izračunamo vektor k_{ij}.

[3]Originalna izvorna koda Perlinovega šuma: `http://mrl.nyu.edu/~perlin/doc/oscar.html#noise`



Slika 2.6: Naključni vektorji gradienta v posameznih točkah

$$k_{ij} = t_i - x_j \tag{2.3}$$

Za vsako točko $x_j = (x_1, x_2, \ldots, x_n)$ torej izračunamo 2^n vektorjev k_{ij} v n dimenzionalnem prostoru. Vsakega od teh vektorjev k_{ij} skalarno zmnožimo z gradientnim vektorjem g_i. Dobimo 2^n skalarnih produktov s_{ij} za vsako točko x_j. V dvodimenzionalnem prostoru je časovna zahtevnost tega algoritma $O(4)$, v splošnem pa $O(2^n)$, kjer je n dimenzija prostora. V nizkih dimenzijah to ne predstavlja nikakršnega problema.

$$s_{ij} = k_{ij} \cdot g_i \tag{2.4}$$

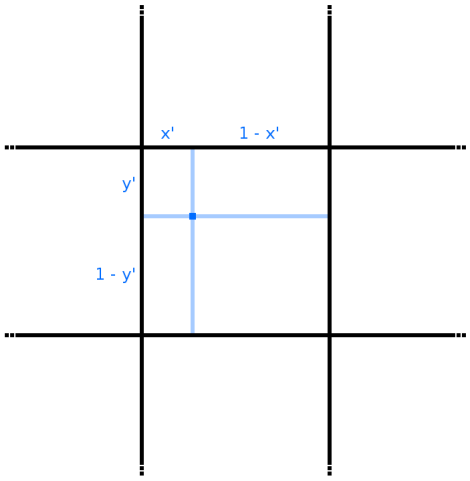
V zadnjem koraku pa izvedemo **bilinearno interpolacijo**, izračunane skalarne produkte uteženo seštejemo v končno vrednost. Imamo točko $x = (x, y)$, iz nje izračunamo pomožne koeficiente $x' = x - \lfloor x \rfloor$ in $y' = y - \lfloor y \rfloor$. Imamo tudi 4 skalarne produkte s_1, s_2, s_3, s_4. Sedaj le primerno uteženo seštejemo le-te in dobimo želeno vrednost z točke x_{ij}, kateri želimo pripisati vrednost Perlinovega šuma na intervalu $[0, 1]$.

$$z_1 = (1 - x') * s_1 + x' * s_2 \tag{2.5}$$

$$z_2 = (1 - x') * s_3 + x' * s_4 \tag{2.6}$$

$$z = (1 - y') * z_1 + y' * z_2 \tag{2.7}$$

Ta postopek ponovimo za vsako točko. V višjih dimenzijah je postopek popolnoma enak, le večje število izračunov je potrebnih, za optimizacijo interpolacije pa se lahko uporablja pristop Monte Carlo[4], kjer toliko časa ponovno generiramo vektor g_i okoli posamezne točke, da leži znotraj enotske sfere v trenutni točki [17], veljati mora torej: $\sum_{k=1}^{n} x_k \leqslant 1$, kjer je x_k koordinata vektorja g_i.



Slika 2.7: Uteži, ki jih vzamemo pri linearni interpolaciji

2.2.1 Simpleksni šum

Ken perlin je leta 2001 razvil simpleksni šum [6], ki je izboljšana različica Perlinovega šuma. Simpleksni šum ima časovno zahtevnost le $O(n^2)$, za razliko od prelinovega šuma, katerega časovna zahtevnost je $O(2^n)$, v našem delu $O(4)$. Vendar, kot je že omenjeno, v nizkih dimenzijah to ne predstavlja težav. Poleg tega se pri simpleksnem šumu boljši učinek opazi šele v višjih dimenzijah. Za naš namenu popolnoma zadostuje Perlinov šum, saj

[4] `http://www.noisemachine.com/talk1/17b.htm`

je matmatično manj kompleksen, poleg tega pa *Unity* privzeto ne podpira simpleksnega šuma.

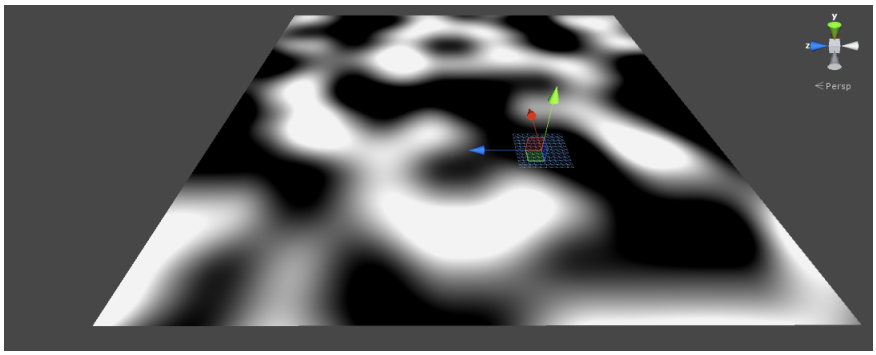
2.3 Beli šum

Za razliko od Perlinovega šuma, ki je gradientni šum, je beli šum vrednostni šum. Pri gradientnemu šumu naključno določimo smeri vektorjev gradienta skalarnega ali vektorskega polja, nato pa glede na te vektorje s pomočjo interpolacije izračunamo skalarno oziroma vektorsko polje, glede na to, katera točka pripada kateremu vektorju.

Pri vrednostnem šumu pa ni potrebnih tolikšnih priprav in izračunov, saj ni potrebno gledati, katere so sosednje točke in njihove vrednosti, temveč le določimo naključno vrednost. V tej diplomski nalogi beli šum uporabljamo kot enega od sestavnih delov teksture trave.

2.4 Prehodi s prosojnostjo

Pri prikazovanju barv na zaslon najpogosteje uporabljamo sistem RGB, ki je sestavljen iz treh kanalov: rdečega (R), zelenega (G), in modrega (B). Ponekod pa potrebujemo tudi četrti kanal, ki pa predstavlja prosojnost in se imenuje kanal prosojnosti (A).



Slika 2.8: Tekstura za pomoč pri zlivanju pokrajin. Za orientacijo je označena ena plošča.

Na princip kanala prosojnosti, smo implementirali tudi prehod med različnimi pokrajinami. Naredili smo teksturo s pomočjo Perlinovega šuma. Določili smo mejo dejanskega prehoda med pokrajinami. Tekstura Perlinovega šuma zavzema vrednosti na intervalu $[0, 1]$, odločili smo se, da bo prehod med pokrajinama pri vrednosti $k \in [0, 1]$, ki ima privzeto vrednost $0,5$. Potem pa določimo tudi širino prehoda, področij Δx s privzeto vrednostjo $0,1$. Vse, kar je izven tega intervala je bodisi trava, bodisi pesek.

2.5 Preslikave

3D predmete predstavimo z množico mnogokotnikov. Zaradi svojih geometrijskih lastnosti so za prikaz 3D predmetov najpogosteje uporabljeni trikotniki [18]. Geometrijske operacije s trikotniki so zelo enostavne. Trikotnik je planaren, kar pomeni da oglišča trikotnika in točke med njimi vedno ležijo na isti ravnini. To zagotavlja, da vedno vemo, kako bo postavljen mnogokotnik in tekstura na njem.

V računalniških igrah se za upodabljanje elementov v prostoru večinoma uporablja trikotnike. Obstajajo tudi drugi pristopi pri upodabljanju, na primer z uporabo vokslov, ki bi predstavljajo tridimenzionalne piksle, polnih modelov, oblak točk .... Uporabo vokslov opazimo pri igri *Minecraft*.

Trikotniki sami v računalniški grafiki niso dovolj. Potrebno jih je premikati, vrteti in raztezati. V tej diplomski nalogi bomo za deformacijo uporabljali višinske preslikave (anlg. mapping). Poleg višinskih preslikav pa bomo uporabljali tudi normalne preslikave, ki pa ne vplivajo na samo geometrijo, vplivajo pa na odboj svetlobe, kar ustvari le vizualni izgled deformirane geometrije, čeprav je v ploskev popolnoma ravna.

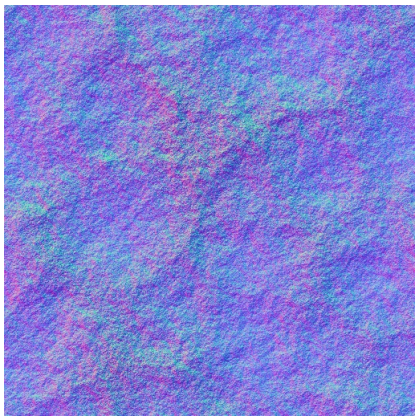
Višinske preslikave [15] deformirajo samo obliko sestavnih delov. Višinsko preslikavo naredimo s pomočjo sivinske slike, ki ji rečemo tudi višinska mapa. Pri višinski mapi določena sivina predstavlja določeno višino terena, bela najvišjo točko, črna najnižjo in vmesne sive primerno višino. Teren s pomočjo te slike deformiramo tako, da vsako oglišče trikotnikov, ki sestavljajo te-

ren, prestavimo v želeno smer glede na vrednost pripadajoče točke na sliki. Tipično se teren deformira v vertikalni smeri, ni pa popolnoma nobene ovire, da bi to storili v poljubni smeri.

Ker v računalniških igrah in programih za simulacijo okolice kamero premikamo, moramo za vsak premik izračunati nov položaj trikotnikov, kar pa je potratna operacija. Zato si želimo čim manj trikotnikov, kjub temu, da bi želeli zelo podroben teren, ki bi izgledal kar se da naravno. Za ta namen uporabimo normalne teksture - polja vektorjev, ki vplivajo na odboj svetlobe. Izračun svetlobe poteka na grafični kartici s pomočjo senčilikov, kar nam zagotavlja hitrost, ki si jo želimo.

Normalna tekstura v vsaki točki hrani zapis o smeri vektorja, ki ga grafična kartica upošteva pri izračunu odboja svetlobnega žarka. Ker zapis o barvi hranimo s tremi komponentami, vektor v trodimenzionalnem prostoru pa ima tri koordinate, lahko kar barvni zapis uporabimo za opis teh normalnih vektorjev. Normalni vektorji pa se tako imenujejo zato, ker so postavljeni pravokotno na teren, ki ga želimo prikazati.

Z normalnimi teksturami lahko dosežemo le rahlo nagubanost, za večje premike terena pa moramo vseeno premikati in deformirati podlago samo.

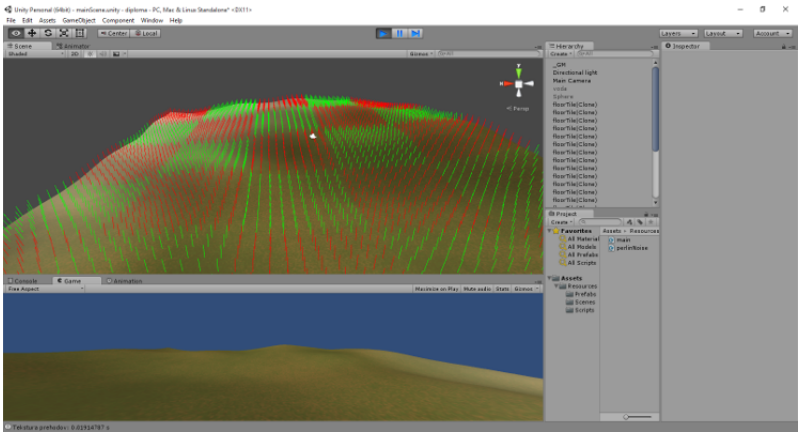


Slika 2.9: Primer normalne teksture.

2.6 Unity

To diplomsko nalogo smo implementirali v igralnem pogonu *Unity*[5], programiramo pa v programskem jeziku *C#*.

Za igralni pogon *Unity* smo se odločili, ker lahko v brezplačni različici uporabljamo vse funkcionalnosti tega pogona, le predstavitveno okno ob zagonu igre prikazuje logotip igralnega pogona *Unity*.



Slika 2.10: Grafični vmesnik urejevalnika *Unity*.

2.6.1 Kratka zgodovina in lastnosti igralnega pogona Unity

Leta 2005 je igralni pogon *Unity* deloval na operacijskem sistemu *OS X*, izrisoval je z rendererjem *OpenGL* (angl. Open Graphic Library), podpiral je programski jezik *C#*. Kasneje so dodali še podporo za *Windows* in izvoz za mnogo drugih popularnih platform, dodali so podporo za programska jezika *JavaScript* in *Boo*. Vključili so fizikalni pogon *PhysX* podjetja *Nvidia*.

Trenutno pa podpira zvočne učinke in njihovo filtriranje v realnem času. Enostavno se zgradi uporabniški vmesnik, ki se uporablja za menije v igri, trenutno igralčevo stanje in stanja nasprotnikov, kot je količina preostalega življenja in podobno.

[5] `https://unity3d.com/`

Unity vsebuje enostavno orodje za animacije, kjer lahko vsakemu objektu animiramo položaj, velikost in rotacijo, lahko pa tudi ustvarimo končni avtomat, v kateremu določimo stanja objektov in sprožilce za prehod med stanji. Poleg diskretnih prehodov pa igralni pogon *Unity* podpira tudi zvezne prehode med stanji, kar pripomore k bolj naravnemu učinku pri animiranju naravnih gibajočih se likov, kot so živali, ljudje, roboti itd.

Poleg naštetih funkcionalnosti pa *Unity* ponuja tudi uporabo sistemov delcev, luči, ki podpirajo svetlobno mapiranje, omogoča že osnovne fizikalne interakcije, kot je gravitacija, detekcija trkov, trenje, itd.

Ima tudi veliko vgrajenih matematičnih funkcij, ki nam močno olajšajo delo. Kot najpomembnejša funkcija za nas je funkcija, ki generira Perlinov šum:

```
Mathf.PerlinNoise(x, y);
```

Parametra x in y predstavljata koordinati v svetu, v katerih želimo izračunati vrednost Perlinovega šuma. Vrne nam pa število s plavajočo vejico (angl. float) na intervalu $[0.0, 1.0]$. Poleg Perlinovega šuma *Unity* podpira še računanje z vektorji in matrikami, ponuja funkcije glajenja, omejevanje vrednosti, skalarni produkt, trigonometrične in druge matematične funkcije.

2.6.2 Platforme, ki jih podpira Unity

Unity trenutno podpira več kot 21 platform, nekatere med njimi pa so: *Windows*, *OS X*, *Linux*, *Xbox One* in *360*, *Playstation*, glavne mobilne platforme: *android*, *Windows phone*, *iOS* itd.

Izvoz za posamezno platformo je zelo enostaven, saj v pogovornem oknu izberemo le želeno platformo, pot do končne datoteke in kliknemo na gumb "Build".

2.6.3 Trgovina sredstev in učna gradiva

Unity ima tudi trgovino sredstev[6] (angl. asset store), kamor lahko razvijalci naložijo svoje izdelke, ki jih drugi razvijalci lahko kupijo, nekateri izdelki pa so tudi brezplačni. Na voljo so 3D modeli, skripte, senčilniki, teksture in materiali, sistemi delcev ali pa celo celotni projekti. Najdemo niz, od zelo preprostih pa do zelo zapletenih rastlin, bitij, robotov in pokrajin.

Poleg vedno večjega nabora sredstev pa se iz dneva v dan pojavlja več in več učnih gradiv (angl. tutorial) v obliki videev, ki počasi, korak za korakom razložijo postopek.

2.6.4 C#

Za implementacijo projekta smo si izbrali programski jezik C# zato, ker je eden izmed jezikov, ki ga podpira igralni pogon *Unity*, je zelo razširjen in se koda napisana v tem jeziku izvaja hitro. C# je med orodji za razvijanje iger bolj razširjen kot ostali jeziki, ki jih podpira *Unity*. To omogoča, da je programska koda napisana v njem prenosljiva in jo lahko nadaljno uporabljamo (angl. reusability).

C# je statično tipiziran jezik[7], kar pomeni, da se tipov spremenljivk ne da spreminjati med samim izvajanjem programa. To se pri večjih programih izkaže za prednost, saj nas sili v konsistentnost in je zato koda bolj obvladljiva, med samim programiranjem pa nastane manj skritih napak ter hroščev.

[6] `https://www.assetstore.unity3d.com/en`

[7] `https://en.wikipedia.org/wiki/C_Sharp_(programming_language)`

Poglavje 3

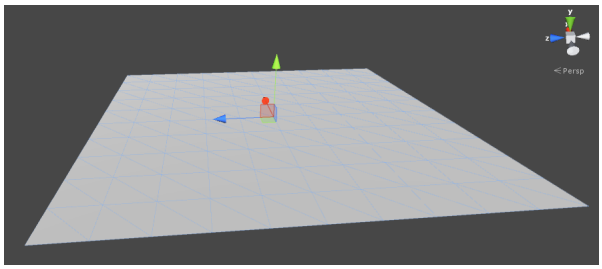
Implementacija generatorja terena

Končni izdelek je narejen popolnoma programsko, uporabljenih ni nikakršnih slik ali tekstur. Uporabnik le določi nekaj parametrov, kot so nagubanost terena, barvni preliv posameznih področij, višina hribov, velikost terena ter seme. Algoritem pa iz podanih parametrov izračuna teksture velikosti 128 × 128, ki se nato uporabijo na posameznih ploščah.

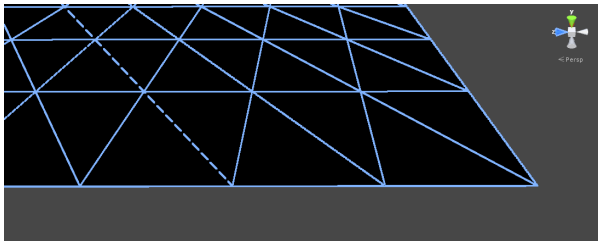
3.1 Plošče

Teren je sestavljen iz kvadratnih plošč, ki jih razporedimo po prostoru tako, da se z robovi stikajo ena z drugo. Na začetku jih postavimo na višino 0, kasneje pa jih deformiramo in premikamo v vertikalni smeri. Vsaka plošča je sestavljena iz 121 točk, oziroma iz 200 trikotnikov. Razdalja med dvema sosednjima točkama pa predstavlja približno 1 meter.

Vsaka plošča je sestavljena iz dveh delov. Prvi del je viden uporabniku in je namenjen, da se nanj izriše tekstura. Drugi del pa ni viden, vendar je namenjen detekciji trkov. Omogoča nam, da se elementi v igri ob njem ustavijo in z njim kako drugače interagirajo. V tej diplomski nalogi upravljamo z obema deloma plošče. Na prvi del plošče izrisujemo teksture in premikamo



Slika 3.1: Plošča v urejevalniku igralnega pogona Unity.



Slika 3.2: Trikotniki, ki sestavljajo ploščo.

točke za končni učinek nagubanosti. Drugi del plošče pa prilagajamo prvemu delu, saj ne želimo, da se objekti, ki se po terenu premikajo, vanj ugreznejo.

3.2 Tekstura peska

Naslednji korak je izdelava teksture peska. Vsaka tekstura je sestavljena iz treh plasti. Tekstura peska je sestavljena iz treh različnih plasti Perlinovega šuma različnih velikosti, zlitih skupaj.

Program kot vhodni parameter dobi dva vektorja $s = (s_1, s2, s_3)$, $p = (p_1, p_2, p_3)$ in preliv g. Posamezna komponenta s_i vektorja s nam pove, kako skrčena bo tekstura posamezne plasti: pri koeficientu s_i bo velikost d stranice celice Perlinovega šuma $d = \frac{1}{s_i}$. Če želimo povečan videz teksture pridobljene s Perlinovim šumom, uporabimo parameter na intervalu $[0, 1]$.

Komponente vektorja p pa so uteži, s katerimi izračunamo uteženo povprečje pri zlivanju posameznih plasti v končno plast. Vektor p moramo pred uporabo pomnožititi z ustreznim koeficientom, da bo vsota vseh komponent

vektorja p enaka $p_1 + p_2 + p_3 = 1$. Za zgled smo uporabili sledeče koeficiente skrčitve in pripadajoče uteži:

$$s = \left(\frac{3}{10}, 1, 7\right) \tag{3.1}$$

$$p = \left(\frac{5}{9}, \frac{1}{3}, \frac{1}{9}\right) \tag{3.2}$$

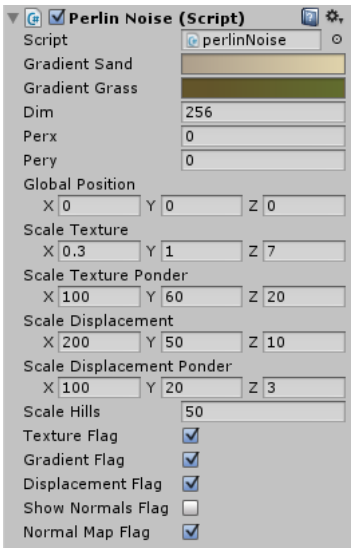
Po združevanju teh treh plasti dobimo črnobelo teksturo tbw. Z barvnim prelivom g jo nato obarvamo. Barvni preliv je še eno orodje, ki nam ga ponuja Unity in zelo poenostavi rokovanje z barvami, saj nam prelivov ni potrebno predhodno ročno izračunati. V tem orodju le preko grafičnega vmesnika določimo ključne točke in barve v teh točkah, pogon *Unity* pa nam potem sam interpolira vrednosti med temi barvami. Želeno barvo c iz gradienta pridobimo tako, da izvršimo ukaz *Evaluate*:

```
Color c = g.Evaluate(0.25f);
```

Obarvano teksturo peska dobimo, da barvo iz črnobele teksture pretvorimo v število v. Ker igralni pogon *Unity* hrani posamezno komponento zapisa barve na intervalu $[0, 1]$, si izberemo poljuben kanal r izmed rdečega, zelenega in modrega. Vrednost izbranega kanala podamo kot parameter metodi Evaluate, iz katere dobimo iskano barvo, ki jo nato vstavimo na primerno mesto v končni teksturi t.

```
Float v = t_{bw}(x, y)(r);
Color c = g.Evaluate(v);
t(x, y) = c;
```

Na začetku smo poskušali imeplementirali rešitev, ki je ponujala večje število plasti, vendar se je izkazalo, da ne pridobimo veliko pri končnem učinku, saj je pri več kot treh plasteh razlika komaj opazna. Poleg tega *Unity* podpira vektorje dveh, treh in štirih dimenzij, kar naredi uporabniški vmesnik bolj pregleden, če za hranjenje koeficientov uporabljamo katerega od



Slika 3.3: Pregledna plošča za nastavljanje parametrov terena.

teh konstruktov. Orodje za urejanje vrednosti vektorjev je bolj pregledno, kot če bi vsako vrednost urejati posebej.

Zaradi take omejitve je posledično tudi izvajanje programa hitrejše, robustnejše in bolj zanesljivo. Te pridobitve na koncu odtehtajo svobodo, ki bi jo imeli v nasprotnem primeru, ki pa ne prinese velikih izboljšav.

Slika 3.4: Končni videz teksture peska.

3.3 Tekstura trave

Postopek izdelave teksture trave je podoben postopku izdelave teksture peska. Razlika je le v prvi plasti. Za razliko od teksture peska, smo se odločili, da namesto Perlinovega šuma uporabimo kar beli šum. Beli šum smo implementirali tako, da smo za vsako točko $T(x, y)$ izbrali naključno vrednost na intervalu $[0, 1]$:

```
float p = Random.Range(0.0f, 1.0f);
```

Ker je v naravi trava pravzaprav sestavljena iz veliko vrst rastlin, je od blizu heterogenega izgleda. Zaradi tega razmisleka smo se odločili, da je beli šum primeren za upodobitev vsaj ene plasti teksture trave. Za zgled smo pri teksturi trave uporabili sledeče parametre skrčitve in pripadajoče uteži:

$$s = (x, 1, 7); x \in \mathbb{R} \tag{3.3}$$

x ne vpliva na končni rezultat.

$$p = \left(\frac{5}{9}, \frac{1}{3}, \frac{1}{9}\right) \tag{3.4}$$

Slika 3.5: Končni izgled teksture trave.

3.4 Teksture normal

Naslednji korak je računanje teksture normal oziroma lepljenje normal [20]. Vsak tip terena ima drugačne lastnosti odboja svetlobe, zato moramo za vsak tip terena ustvariti drugačno teksturo normal. Ker smo želeli ohraniti majhno število parametrov, ki jih lahko uporabnik spreminja, smo se odločili, da se parametrov teh tekstur ne bo dalo spreminjati, saj smo bili z učinki, ki smo jih ustvarili, zelo zadovoljni.

Učinek, ki je posledica lepljenja normal, bi se dalo občutno izboljšati, če ne bi uporabili le Perlinovega šuma, temveč kakšen drug pristop. En od možnih pristopov, ki smo jih našli, je generiranje tekstur s pomočjo turbulence[1]. Ta pristop uporablja teksturo, ki se ji barva spreminja sinusno v odvisnosti od koordinat, nato pa na tej teksturi uporabi turbulenco črt v naključni smeri. Učinek, ki nastane pri tem pristopu z uporabo turbulence, je presenetljivo podoben pesku v puščavi.

V tej diplomski nalogi smo teksturo normal peska implementirali v dveh korakih in sicer: generiranje črnobele teksture ter pretvarjanje te teksture v teksturo normal. Prvi korak, se pravi generiranje sivinske teksture, poteka enako kot priprava sivinske teksture pri generiranju teksture peska in teksture trave. Edina razlika so nastavitve parametrov, kjer uporabimo bolj fine plasti. Posamezne plasti smo bolj skrčili kot pri barvnih teksturah, se pravi, uporabili smo večje koeficiente skrčitve.

Kot primer smo uporabili sledeče parametre:

$$s_{np} = (1, 4, 32) \tag{3.5}$$

$$p_{np} = \left(\frac{1}{13}, \frac{2}{13}, \frac{10}{13}\right) \tag{3.6}$$

$$s_{nt} = (1, 4, 32) \tag{3.7}$$

$$p_{nt} = \left(\frac{10}{13}, \frac{2}{13}, \frac{1}{13}\right) \tag{3.8}$$

Kjer indeks np predstavlja parametre teksture normal pseka, indeks nt pa parametre teksture normal trave.

[1] `http://lodev.org/cgtutor/randomnoise.html`

Naslednji korak je pretvorba v prejšnjem koraku pridobljene teksture v dejansko teksturo normal. V vsaki točki T teksture t, ki smo jo izračunali v prejšnjem koraku, moramo izračunati normalni vektor. Ta vektor določimo s pomočjo štirih sosednjih točk. Ker vrednost vsake točke teksture predstavlja višino, lahko iz teh sosednjih točk izračunamo normalni vektor.

Iz vrednosti a leve točke in vrednosti b desne točke izračunamo x koordinato normalnega vektorja n v točki T. Koordinato y normalnega vektorja n pa izračunamo iz vrednosti c zgornje točke ter vrednosti d spodnje točke. Koordinati z priredimo vrednost 1:

$$x = ((a - b) + 1) * 0,5 \tag{3.9}$$

$$y = ((c - d) + 1) * 0,5 \tag{3.10}$$

$$z = 1 \tag{3.11}$$

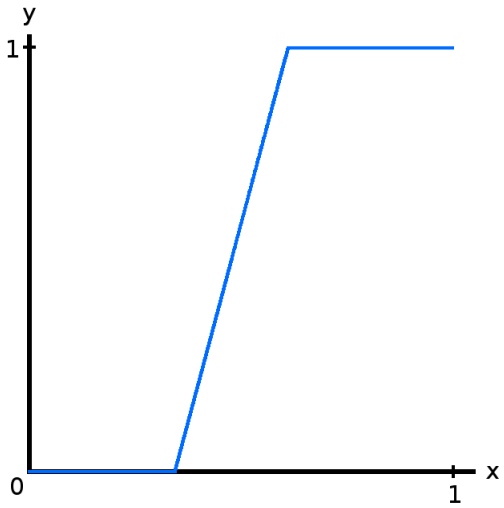
Koordinati x in y moramo korigirati, saj velja: $a - b \in [-2, 2]$, mi pa želimo vrednosti na intervalu $[0, 1]$. Tej razliki zato prištejemo 1 in dobljeno vrednost razpolovimo, da zagotovimo, da bo končna vrednost na intervalu $[0, 1]$.

Te komponente nato zložimo v vektor $n = (x, y, z)$. Ker imata tako normalni vektor, kot tudi zapis barv v računalništvu tri komponente, se barvni zapis uporablja za predstavitev normalnih vektorjev. Koordinata x je predstavljena z rdečim kanalom, y z zelenim in z z modrim.

3.5 Zlivanje tekstur

Ker želimo imeti v naši igri oziroma drugi aplikaciji, ki bi uporabljala naš teren, različne tipe pokrajin naenkrat, smo implementirali tudi zlivanje tekstur, kar upodablja prehode med posameznimi pokrajinami. Za ta namen smo ustvarili novo teksturo, prav tako s pomočjo Perlinovega šuma, s katero smo implementirali zlivanje tekstur. Bela barva na tej teksturi predstavlja eno pokrajino, črna barva pa drugo pokrajino. Sivina med belo in črno barvo pa predstavlja razmerje med mešanico dveh pokrajin.

Z običajno teksturo pridobljeno s pomočjo Perlinovega šuma praktično ni popolnoma belih oziroma popolnoma črnih področij. Zaradi tega, moramo vrednosti te teksture preslikati, tako da dobimo kratka prehodna območja in velika območja črne ter bele barve. Za ta namen smo uporabili preslikavo, ki deluje na sledeč princip: $v = min(1, max(0, v * (k * 2 + 1) - k))$, kjer je k koeficient sorazmeren s hitrostjo prehoda. Postopek je podoben povečanju kontrasta slike.



Slika 3.6: Slika prikazuje preslikavo iz dolgih v kratke prehode črnih (0) in belih (1) področij, kjer x predstavlja prvotno, y pa končno stanje.

Barvo (vrednost) vsake točke T_{fc} končne teksture in točke T_{fn} normalne teksture dobimo tako, da uteženo seštejemo vrednosti v ustreznih točkah T_{pc} in T_{pn} tekstur peska in vrednosti v točkah T_{tc} in T_{tn} tekstur trave. Vrednost v, izračunana s preslikavo, ki je prikazana na sliki 3.6, predstavlja iskano razmerje r, ki ga uporabimo kot utež pri seštevanju:

$$T_{fc} = r * T_{pc} + (1 - r) * T_{tc} \tag{3.12}$$

$$T_{fn} = r * T_{pn} + (1 - r) * T_{tn} \tag{3.13}$$

S tem je generiranje tekstur končano.

3.6 Višinske preslikave

Poleg drobnega nagubanja terena, ki ga dobimo z uporabo normalnih tekstur, tipično želimo tudi večje hribe in doline. Za ta namen uporabimo višinske mape, tipično črnobele slike, ki jih uvozimo v program, ki teren deformira glede na barvo v primerni točki uvožene slike. Črna barva predstavlja najnižjo točko terena, bela najvišjo, vmesne višine pa so predstavljene z vrednostmi sivih barv, enako kot pri generiranju normalnih tekstur.

Ker je naš teren generiran funkcijsko, ne pride v poštev, da bi najprej ustvarili sliko, ki bi predstavljala višinsko mapo. Ker je teren sestavljen iz posameznih plošč, plošča pa iz točk, smo se odločili, da bomo deformirali plošče neposredno. Se pravi, prestavljamo posamezne točke na primerno višino.

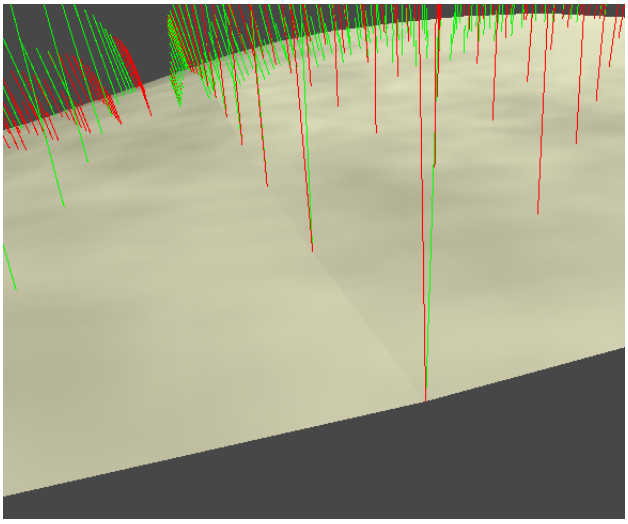
Za implementacijo višinskih map smo ponovno uporabili tri različne plasti Perlinovega šuma, ki smo jih primerno sešteli. Vsaka plast je namenjena za različno nagubanost terena. Postopek je popolnoma enak kot pri generiranju normalnih map in teksture peska, le koeficiente si moramo izbrati tako, da so hribi primernih velikosti.

Najprej smo za vsako točko T plošče p ugotovili njen položaj v prostoru, nato pa glede na njen položaj (x, y) s pomočjo preslikave f, ki uporablja tri plasti Perlinovega šuma, izračunali nov položaj $T'(x, y, z')$:

$$z' = k * f(x, y) \tag{3.14}$$

S preslikavo f določimo novo vrednost koordinate z. Glede na položaj točke v prostoru ji določimo nov, končni položaj, se pravi premaknemo jo gor ali dol za primerno vrednost. Ker je zaloga vrednosti preslikave f na intervalu $[0, 1]$, smo vpeljali nov koeficient k, s katerim določamo največjo višino hribov.

Pri tej fazi generiranja terena se je pojavila težava. Pri premikanju točk posamezne plošče igralni pogon spreminja tudi normale, ki so pripete na točke. Pri izračunavanju normal pogon upošteva točke, ki so sosednje točki, ki jo premikamo. Ker robne točke plošč nimajo vseh sosedov, jih pogon ne more upoštevati, zato normale na robovih niso poravnane, kar povzroči, da se na robovih plošč pojavijo prelomi.



Slika 3.7: Zelo opazen prelom in neporavnani normalni vektorji.

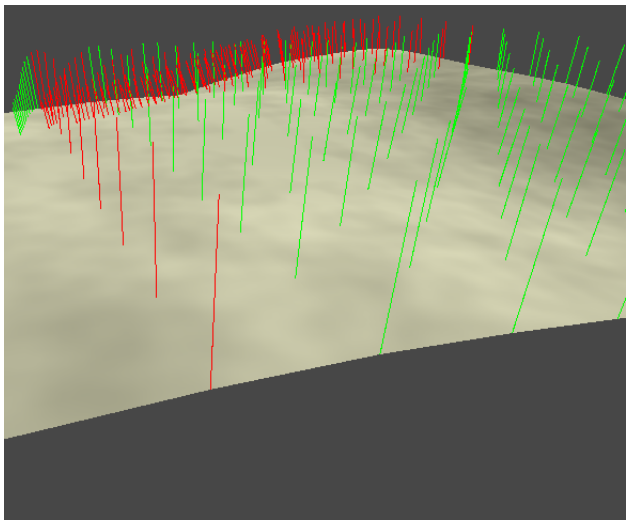
Težavo smo odpravili tako, da na robovih plošč ročno poravnamo normale. Za vsako robno točko $T(x, y, z)$ s pomočjo preslikave f izračunamo dve sosednji točki A in B. Iz točk T, A in B nato izračunamo vektorja a in b. vektorja a in b zmnožimo z vektorskim produktom in dobljen rezultat je nova, popravljena normala, ki jo priredimo točki T. Ker v vsaki točki izvedemo isti postopek, smo zagotovili, da na istoležnih točkah vedno dobimo isti normalni vektor in posledično gladek prehod.

$$A = (x + 1, y, f(x + 1, y)) \tag{3.15}$$

$$B = (x, (y + 1), f(x, y + 1)) \tag{3.16}$$

$$a = A - T, b = B - T \tag{3.17}$$

$$n' = a \wedge b \tag{3.18}$$



Slika 3.8: Sovpadajoči normalni vektorji in posledično gladek prehod.

Poglavje 4

Evalvacija in rezultati

4.1 Prostorska analiza generiranja terena

Z namenom analize porabe prostora smo desetkrat zagnali generiranje 100 plošč in desetkrat generiranje 2500 plošč. Za merjenje uporabljenjega prostora smo uporabili program *Upravljalec procesov* (*Task manager*). Pri generiranju 100 plošč se je v povprečju zasedenost pomnilnika povečala za $S_{100} = 200$ MB, pri generiranju 2500 plošč pa se je zasedenost pomnilnika v povprečju povečala za $S_{2500} = 4200$ MB.

Porabo prostora za različna števila plošč smo izmerili zato, ker smo predvidevali, da za zagon postopka igralni pogon *Unity* potrebuje nekaj začetnega prostora. Vrednosti tega začetnega prostora se znebimo tako, da izmerjene vrednosti preprosto odštejemo in dobimo le prostor, ki ga porabi razlika plošč. Na koncu lahko še s pomočjo začetnih izmerjenih količin in končnih izračunanih vrednosti, izračunamo koliko začetnega prostora potrebuje igralni pogon *Unity* za zagon.

$$S_{100} = s_0 + s_{100} \tag{4.1}$$

$$S_{2500} = s_0 + s_{2500} \tag{4.2}$$

$$s_{2400} = S_{2500} - S_{100} = s_0 + s_{2500} + (s_0 + s_{100}) = s_0 - s_0 + s_{2500} - s_{100} \tag{4.3}$$

$$s_{2400} = s_{2500} - s_{100} = S_{2500} - S_{100} \tag{4.4}$$

$$s_{2400} = 4200\,\text{MB} - 200\,\text{MB} = 4000\,\text{MB} \tag{4.5}$$

$$s_1 = \frac{4000\,\text{MB}}{2400} = \underline{1.67\,\text{MB}} \tag{4.6}$$

$$s_0 = S_{2500} - s_{2500} = 4200\,\text{MB} - 2500 * 1.67\,\text{MB} = \underline{33\,\text{MB}} \tag{4.7}$$

Prostor v pomnilniku, ki ga zasede posamezna plošča torej znaša približno 1.67 MB. Začetni prostor, ki ga igralni pogon *Unity* potrebuje za zagon pa je približno 33 MB.

4.2 Časovna analiza generiranja terena

Trajanje posameznih stopenj pri generiranju terena za posamezno ploščo:

Faza	**Trajanje** $[s]$
Tekstura peska	0,014
Tekstura trave	0,012
Višinska preslikava	0,001
Normalna tekstura peska	0,043
Normalna tekstura trave	0,013
Tekstura prehodov	0,019
Skupaj	**0,102**

Tabela 4.1: Trajanje generiranja posameznih stopenj.

Očitno največ časa porabi generiranje normalnih tekstur, kar 55% celotnega časa. Pohitritev izvajanja izračuna normalnih map bi lahko izvedli z izbiro boljšega algoritma. V našem primeru uporabljamo naiven pristop. Za vsako točko izračunamo normalni vektor, tako da interpoliramo vrednosti štirih sosednjih točk.

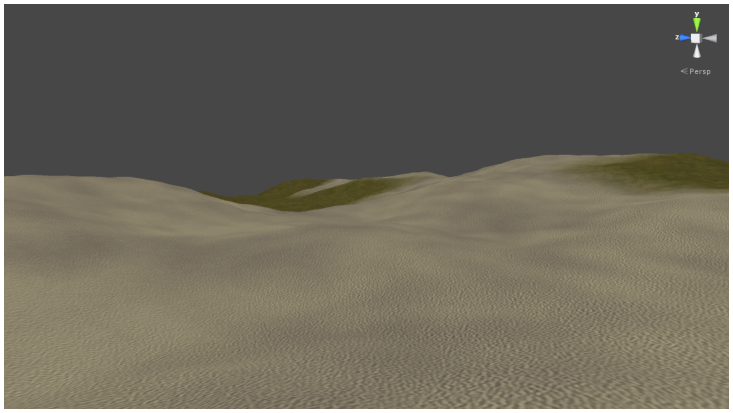
Pravzaprav vse stopnje izgradnje terena porabijo veliko časa, razen višinskih map, saj izračunamo le 121 točk, namesto približno 16000 točk, kot jih

računamo pri teksturah. Eden izmed načinov, kako bi postopek generiranja lahko deloma pohitrili je, da ne bi računali vsake faze posebej, temveč bi vse takoj izračunali v eno teksturo. Namesto petih tekstur, ki jih pridobimo med postopkom, bi dobili le eno teksturo. Operacije, ki jih uporabimo za združevanje bi se skrajšale, saj bi prihranili mnogo operacij, ki jih uporabljamo zdaj.

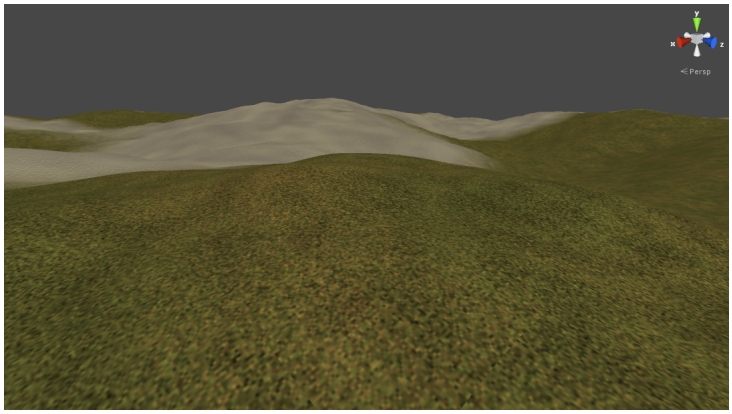
Še bolj kot na času pa bi s tem pristopom prihranili na prostoru. Vendar tudi ta izboljšan postopek ne bi občutno zmanjšal časa potrebnega za generiranje terena. Za večjo pohitritev bi morali celoten program prepisati v senčilnike.

Poleg tega pa se senčilniki učinkoviteje prevedejo v nižjenivojski jezik, saj je nabor najbolj uporabljenih postopkov, ki se izvajajo na grafični kartici, majhen. Zaradi tega se senčilniki zelo učinkovito prevedejo v jezik, ki se izvaja na grafičnih karticah in je zaradi tega čas izvajanja programa na grafičnih karticah še manjši.

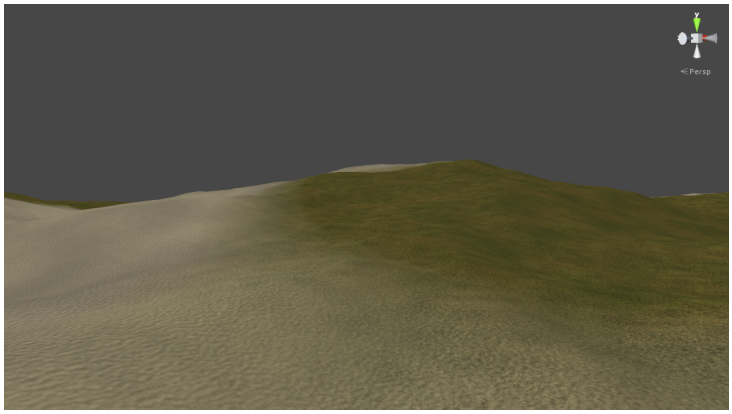
4.3 Zaslonske slike končnega rezultata



Slika 4.1: Peščeni teren.



Slika 4.2: Travnati teren.



Slika 4.3: Prehod med peskom in travo.

Poglavje 5

Zaključki in nadaljnje delo

S končnim videzom našega izdelka smo zelo zadovoljni, saj da kot celota zelo lep in privlačen videz. Posebej fascinantno je dejstvo, da lahko z relativno majhnim številom nastavljivih parametrov dosežemo učinek, ki se zdi podoben naravnemu. Do rezultata, ki bi bil funkcijsko generiran in bi izgledal enako kot pokrajina v naravi pa je seveda še dolga pot.

Med samim implementiranjem zastavljenih ciljev so se odpirale nove ideje za razširitev te diplomske naloge. Naslednji korak bi bil implementacija dinamično generiranega terena. To pomeni, da bi se teren generiral v okolici igralca in se ne bi predhodno izračunal, tako kot se računa zdaj. Poleg tekstur, ki so statične, se ne premikajo, bi lahko ustvarili tudi dinamične teksture. S temi teksturami bi lahko upodobili puščavski pesek, ki se preoblikuje v vetru, ali pa bi ustvarili učinek vode.

Kasneje bi se lahko dodalo tudi rastline in drevesa, skratka potencialnih razširitev je veliko in dobili smo občutek, da ima ta izdelek potencial v uporabi za marsikatero področje, med katerim so igre in predstavitvene aplikacije arhitekturnih ter podobnih projektov.

Poleg razširitve na vsebinskem nivoju pa dodatno izboljšavo prinesla uporaba senčilnikov, s katerimi bi pridobili tako na porabljenem prostoru v pomnilniku, kot tudi na času generiranja. Naslednji pristop k optimizaciji, poleg senčilnikov, bi bila uvedba teselacije [3]. Ker je dovolj, da uporabnik

vidi podrobnosti le blizu svojega položaja, bi za zelo oddaljen teren uporabili večje plošče, ki bi potrebovale manjšo gostoto izračunov za posamezne teksture.

Skratka poti so odprte za nadaljnje delo in menimo, da bi bil izdelek lahko uporabljen v drugih izdelkih zaradi svoje robustnosti in enostavnosti za uporabo.

Literatura

[1] Paul Bourke. Perlin Noise and Turbulence. `http://paulbourke.net/texture_colour/perlin/`, 2000. [Online; accessed 10. 9. 2015].

[2] Ilja Nikolajevič Bronštejn, Konstantin Adolfovič Semendjajev, and Albin Žabkar. *Matematični priročnik*. Tehniška založba Slovenije, 1990.

[3] Harold Scott Macdonald Coxeter. *Regular polytopes*. Courier Corporation, 1973.

[4] Wim Etten. *Introduction to random signals and noise*. Wiley, Chichester, England Hoboken, NJ, 2005.

[5] Miguel Frade, Francisco Fernandéz de Vega, and Carlos Cotta. Breeding terrains with genetic terrain programming: the evolution of terrain generators. *International Journal of Computer Games Technology*, 2009, 2009.

[6] Stefan Gustavson. Simplex noise demystified. `http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, 2005. [Online; accessed 8. 9. 2015].

[7] ER Hapeman, WR Zeuch, JA Crandall, SM Carioti, SD Barclay, and C Underkoffler. Telecom glossary 2000–american national standard t1. 523-2001. http, 2001.

[8] Andrew Kensler, Aaron Knoll, and Peter Shirley. Better gradient noise. Technical report, Tech. Rep. UUSCI-2008-001, SCI Institute, University of Utah, 2008.

[9] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S Ebert, JP Lewis, Ken Perlin, and Matthias Zwicker. A survey of procedural noise functions. In *Computer Graphics Forum*, volume 29, pages 2579–2600. Wiley Online Library, 2010.

[10] Benoit B Mandelbrot. *The fractal geometry of nature*, volume 173. Macmillan, 1983.

[11] Microsoft. Turbulence effect. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh706378(v=vs.85).aspx`, 2000. [Online; accessed 10. 9. 2015].

[12] Sudhanshu K Mishra. On estimation of the parameters of gielis superformula from empirical data. *Available at SSRN 905051*, 2006.

[13] Jacob Olsen. Realtime procedural terrain generation. 2004.

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

[15] Gabriele Peters and Jochen Kerdels. Image segmentation based on height maps. In *Computer Analysis of Images and Patterns*, pages 612–619. Springer, 2007.

[16] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[17] Christian Robert and George Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[18] William J Schroeder, Jonathan A Zarge, and William E Lorensen. Decimation of triangle meshes. In *ACM Siggraph Computer Graphics*, volume 26, pages 65–70. ACM, 1992.

[19] Jason Shankel. Fractal terrain generation–midpoint displacement. *Game Programming Gems*, 1:503–507, 2000.

[20] Peter-Pike Sloan. Normal mapping for precomputed radiance transfer. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 23–26. ACM, 2006.

[21] Ruben M Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. A proposal for a procedural terrain modelling framework. In *Poster Proceedings of the 14th Eurographics Symposium on Virtual Environments EGVE08*, pages 39–42, 2008.

[22] Wikipedia. Fractal. `https://en.wikipedia.org/wiki/Fractal`. [Online; accessed 8. 9. 2015].

[23] Wikipedia. L-system. `https://en.wikipedia.org/wiki/L-system`. [Online; accessed 8. 9. 2015].

[24] Wikipedia. Nabla. `https://en.wikipedia.org/wiki/Nabla_symbol`. [Online; accessed 8. 9. 2015].

[25] Wikipedia. Noise (electronics). `https://en.wikipedia.org/wiki/Noise_(electronics)`. [Online; accessed 8. 9. 2015].

[26] Wikipedia. Zven. `https://sl.wikipedia.org/wiki/Zven`. [Online; accessed 11. 9. 2015].