RĪGAS TEHNISKĀ UNIVERSITĀTE
DATORZINĀTNES UN INFORMĀCIJAS TEHNOLOĢIJAS FAKULTĀTE
Informācijas tehnoloģijas institūts

Elviss Strazdiņš

BEZGALĪGA RELJEFA IMITĀCIJAS ĢENERĒŠANA PIELIETOJUMIEM DATORSPĒLĒS UN VIZUĀLOS MODEĻOS

Maģistra darbs

Vadības informācijas tehnoloģijas katedra
Meža ielā 1/3
Tālrunis (+371) 7089515

Zinātniskais vadītājs:
Agris Ņikitenko
Docents, Dr. sc. ing.

Rīga – 2013

RĪGAS TEHNISKĀ UNIVERSITĀTE
DATORZINĀTNES UN INFORMĀCIJAS TEHNOLOĢIJAS FAKULTĀTE
Informācijas tehnoloģijas institūts

BEZGALĪGA RELJEFA IMITĀCIJAS ĢENERĒŠANA PIELIETOJUMIEM DATORSPĒLĒS UN VIZUĀLOS MODEĻOS

Elviss Strazdiņš

Anotācija

Mūsdienu datoru radītajās virtuālās vides tiek izmantoti arvien lielāki un detalizētāki reljefi, tāpēc reljefu izstrāde kļūst aizvien darbietilpīgāka, kā arī, tie aizņem aizvien vairāk vietas uz datu nesējiem. Šo problēmas risināšanai iespējams izmantot algoritmus, kas reljefus ģenerē procedurāli.

Darbā piedāvāts risinājums bezgalīgu reljefu ģenerēšanai pielietojumiem datorspēlēs un vizuālos modeļos. Tā kā reālistisku reljefu ģenerēšanai nepieciešams izmantot vairākus algoritmus, tad maģistra darbā tiek apskatīti visi izmantotie algoritmi. Balstoties uz teorētiskām atziņām par algoritmu pielietojumiem un darbības principiem, tika izstrādāts un pilnveidots risinājums, ko iespējams integrēt datorspēlēs un vizuālos modeļos.

Maģistra darbā ir 79 lappuses, 64 attēli, 1 tabula un 70 literatūras avoti.

RIGA TECHNICAL UNIVERSITY
FACULTY OF COMPUTER SIENCE AND INFORMATION TECHNOLOGY
Institute of Information Technology

INFINITE TERRAIN IMITATION GENERATION FOR APPLICATION IN COMPUTER GAMES AND VISUAL MODELS

Elviss Strazdiņš

Annotation

Modern virtual worlds are bigger and more detailed terrains than than they used to be, so their development is becoming more time consuming and they take up more place on storage devices. It is possible to solve these problems by procedurally generating terrains.

In this Master Thesis terrain generation system for usage in games and visual models is developed. A lot of algorithms and methods are needed to create realistic terrains, so in this paper all of the used ones are described. A procedural terrain generation system based on the researches that can be integrated in computer games and visual models was developed.

The volume of the Master Thesis is 79 pages, 64 figures, 1 table and 70 bibliographical sources.

РИЖСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Институт информационных технологий

BEZGALĪGA ZEMES RELJEFA IMITĀCIJAS ĢENERĒŠANA PIELIETOJUMIEM DATORSPĒLĒS UN VIZUĀLOS MODEĻOS

Элвисс Страздиньш

Аннотация

В современных виртуальных мирах используются всё большие и более детализированные рельефы, поэтому разработка таких рельефов становится всё более трудоёмкой, а так же занимает больше места на носителях данных. Для решения этой проблемы можно использовать алгоритмы процедурной генерации рельефов.

В данной работе описывается разработка решения для генерации бесконечного рельефа для компьютерных игр и визуальных моделей. Так как для генерации реалистичного рельефа необходимо использовать множество алгоритмов, в работе описаны все используемые алгоритмы. Руководствуясь полученными теоретически знаниями о применении алгоритмов и принципе их работы, было разработано и оптимизировано решение, которое можно интегрировать в компьютерные игры и визуальные модели.

Объём магистрской работы 79 стр., 23 рис., 1 таблица и 64 источников литературы.

Saturs

IEVADS

Līdz ar pirmo datorspēļu parādīšanos virtuālās pasaules ir nemitīgi augušas izmēros, gan arī to sarežģītībā. Augot prasībām pret virtuālo pasauļu satura kvalitāti un kvantitāti, arvien vairāk tehnoloģiju un resursu tiek veltīts to radīšanai. Šī problēma laikam ejot kļūst aizvien lielāka, tāpēc tiek meklēti dažādi rās risinājumi.

Viens no risinājumiem milzīgu virtuālo pasauļu veidošanai ir procedurāla satura ģenerēšana. Parasti procedurālā satura ģenerēšana tiek izmantota augstas sarežģītības satura ģenerēšanai [55]. Tā ļauj ar dažādu algoritmu un metožu palīdzību reālā laikā ģenerēt dažādus objektus, vizuālos efektos, mūziku vai pat scenāriju, prasot minimālu mākslinieku ieguldījumu. Tā kā saturs tiek ģenerēts programmas izpildes laikā, nevis pirms tā, tad pasaule uz datu nesējiem parasti aizņem daudz mazāk vietas. Lai arī procedurālā satura ģenerēšana patērē lielu apjomu datora resursu, tomēr datoru aparatūras straujā attīstība mūsdienās paver iespējas to darīt arī reālā laikā.

Darbā tiek pētīti un salīdzināti esošie reljefa imitācijas ģenerēšanas risinājumi. Balstoties uz esošajiem risinājumiem reljefu ģenerēšanā, maģistra darbā tiks izstrādāts risinājums, kas programmas izpildes laikā izmantojot dažādus algoritmus un metodes ģenerē šķietami bezgalīgu zemes reljefa imitāciju.

Maģistra darba mērķis ir izveidot risinājumu, kas reālā laikā ģenerē bezgalīgu zemes reljefa imitāciju lietojumiem datorspēlēs un vizuālos modeļos. Maģistra darba mērķa sasniegšanai tika izvirzīti šādi uzdevumi:

1. Iepazīties ar esošiem pētījumiem procedurāla satura ģenerēšanā pielietojumiem spēlēs un vizuālos modeļos;
2. Izpētīt un atlasīt piemērotākos algoritmus un tehnoloģijas reljefa ģenerēšanai reālā laikā;
3. Izveidot risinājumu, kas reālā laikā ģenerē reljefu un veido tā 3D projekciju;
4. Analizēt iegūtā pētījuma rezultātus.

Maģistra darba pirmajā nodaļā tiek izklāstīta vispārēja informācija par procedurālu

satura ģenerēšanu, kā arī, lai izprastu procedurāli ģenerēta satura pamatprincipus, tiek izpētīti esoši risinājumi procedurālu pilsētu, objektu un reljefu ģenerēšanai, kā arī to pielietojums datorspēlēs un citos lietojumos.

Otrajā nodaļā tiek atlasīti un apskatīti piemērotākie algoritmi bezgalīga reljefa ģenerēšanai, kā arī aprakstīts to darbības principi un lietojums risinājumā.

Savukārt trešajā nodaļā koda fragmentiem un ilustrācijām tiek aprakstīta risinājuma praktiskā implementācija.

Pēc trešās nodaļas seko secinājumi par bezgalīga zemes reljefa ģenerēšanu un tā pielietojumu datorspēlēs un vizuālos modeļos, kas balstīti uz iegūtajām zināšanām un izstrādātā risinājuma rezultātiem.

Darbā izmantoto terminu vārdnīca atrodas darba beigās.

1. BEZGALĪGA ZEMES RELJEFA IMITĀCIJAS ĢENERĒŠANA

1.1. Pamatjēdzieni

Procedurālā satura ģenerēšana izmanto dažādus algoritmus un metodes, lai veidotu virtuālus objektus. Ģenerējot bezgalīgos objektus, tiek izmantotas metodes to sadalīšanai un tikai vajadzīgo daļu ģenerēšanai. Lai procedurāli ģenerēti neizskatītos viendabīgi, to ģenerēšanai tiek izmantoti pseidonejaušu skaitļu ģeneratori. Zemes reljefu ģenerēšana parasti notiek deformējot 2D plakni, ko sauc arī par augstumu karti. Lai attēlotu uzģenerēto reljefu 3 dimensijās, uzģenerētā augstumu karte tiek pārveidota par trīsstūriem, kas tiek projicēti 3 dimensijās. Reljefu iespējams apskatīt no dažādām skatu punktiem pa 3D pasauli pārvietojot skatu punktu, kuru mēdz saukt par kameru (*camera*).

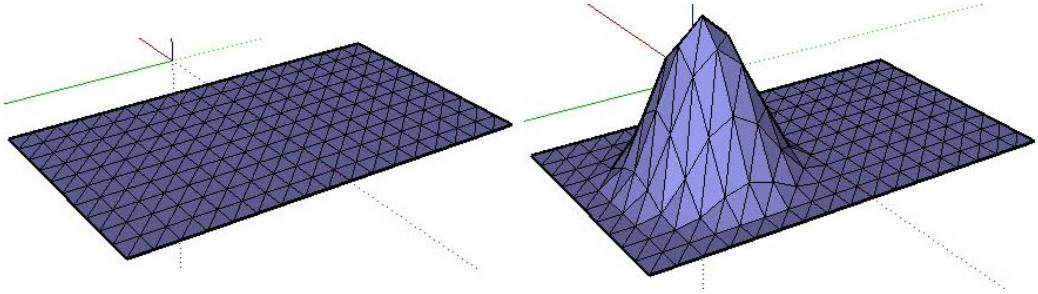
1.2. Pseidobezgalīgi skaitļi

Tā kā datorā katram skaitlim ir noteiktas robežas, tad par bezgalīgām tiek uzskatītas lietas, kas izmanto visu pieejamo skaitļu kopu. Piemēram, 32 bitu skaitļa robežas ir no −2147483648 līdz +2147483647 [31]. Ja tiek izmantota 3D telpa, kuras robežas ir 32 biti, un zemes reljefa virsotnes tiek projicēta ar izšķirtspēju, kas kamerai ir praktiski nesaskatāma, šīs robežas ir pilnībā pietiekamas, lai veidotu milzīgas 3D pasaules. Izmantojot 64 bitu skaitļus, var droši apgalvot, ka reljefa robežas ir nesasniedzamas. Tomēr, neveiksmīgu koordinātu izvēles dēļ, iespējams kameru nostādīt tieši uz pašas robežas, lai šādās situācijās radītu ilūziju, ka 3D pasaule ir bezgalīga, koordinātas tiek izmantotas cikliski t.i. katru reizi, kas tās sasniedz kādu no robežām, to vērtība tiek iestatīta kā pretējā robeža.

1.3. Zemes reljefs

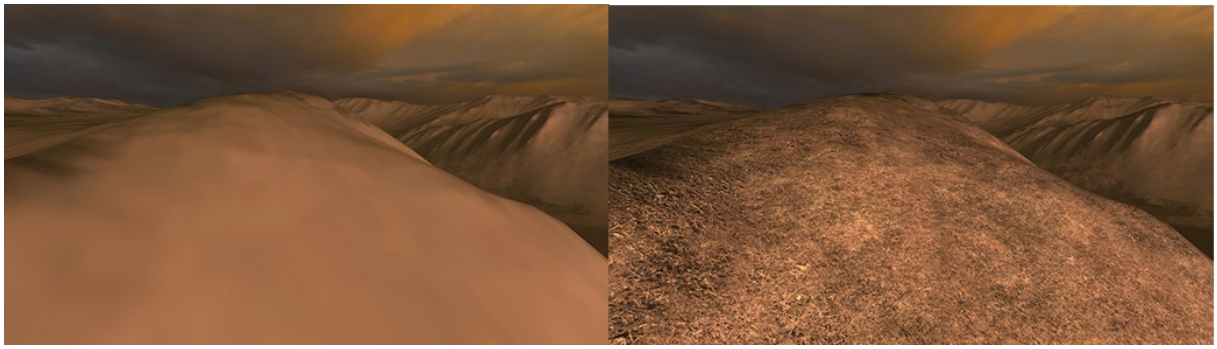
Ar terminu "zemes reljefs" datorzinātnēs saprot 3D modeli, kas reprezentē kalnus, ielejas un dažādus citus izliekumu veidus. Zemes reljefa glabāšanai parasti izmanto augstumu kartes. Lai attēlotu augstumu karti kā 3D reljefu, tiek izveidots režģis, kas stiepjas pa X un Z asīm. 3D režģis sastāv no četrstūriem, kas veidoti no diviem blakusesošiem trīsstūriem. Katrs 3D režģa četrstūru krustpunkts atbilst vienam punktam augstumu kartē. Katrs 3D režģa

krustpunkts tiek pacelts tādā augstumā, kāds glabājas augstumu kartē (skat. 1.3.1. att.) [47].



1.3.1. att. **Tukšs 3D režģis un deformēts 3D režģis**

Lai reljefs neizskatītos vienkrāsains, tam parasti tiek uzklāta tekstūra (*texture*). Bieži vien tekstūra tiek uzklāta vairākās kārtās, lai reljefs izskatītos reālistiskāks. Lai reljefs izskatītos detalizētāks, viens no tekstūras slāņiem tiek lietots kā detalizācijas karte (*detail map*) (skat. 1.3.2. att.). Papildus tekstūrām, reljefam iespējams aprēķināt katra punkta gaišumu, kas atkarīgs no šī punkta leņķa pret gaismas avotiem, piemēram, sauli [67].

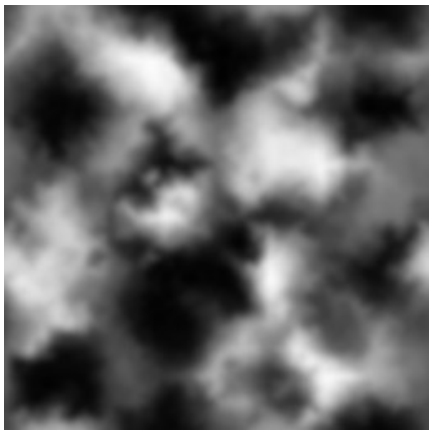


1.3.2. att. **Apgaismots reljefs bez un ar detalizācijas karti**

1.4. Augstumu karte

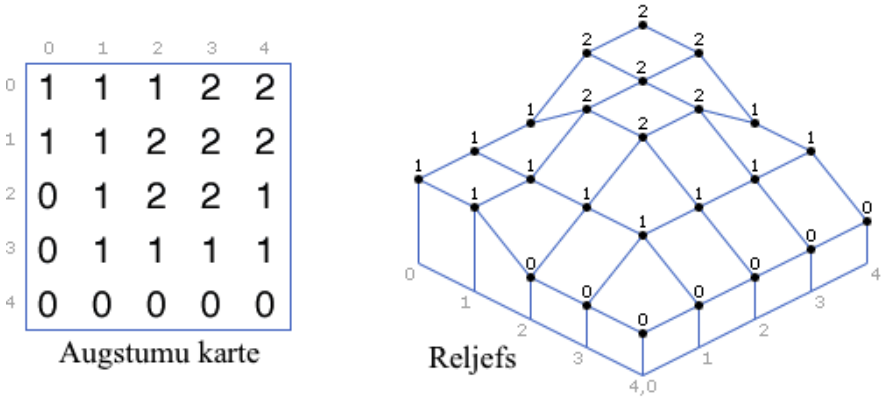
Datorgrafikā augstumu karte (*heightmap*) ir attēls (skat. 1.4.1. att.), kas tiek izmantots zemes reljefa virsotņu augstuma glabāšanai. Katrs augstumu kartes punkts apraksta vienu virsotni 3D režģī. Bitkartes (*bitmap*) katram pikselim ir noteikts skaitu krāsu kanālu, no kuriem katrs glabā vienas krāsa vērtību šajā pikselī. RGB (*red, green, blue – sarkans, zaļš, zils*) krāsu modelī katrs pikselis glabā sarkanas, zaļas un zilas krāsas kanālus, bet no visiem kanāliem parasti augstumu glabāšanai tiek izmantots tikai viens [57]. Lai uzskatāmi varētu redzēt katra punkta augstumu, katra pikseļa visi krāsu kanāli tiek uzstādīti vienādi, līdz ar to

augstumu karte izskatās melnbalta. Jo gaišāks pikselis augstumu kartē, jo augstāks ir punkts kuru tas apzīmē (skat. 1.4.2. att.).



1.4.1. att. **Augstumu karte**

Pēc augstumu kartes ielādes, tā tiek lietota, lai deformētu reljefa režģi (skat. 1.4.2. att.). Tā kā parasti augstumu karšu glabāšanai tiek izmantotas bitkartes, tad 24 bitu bitkartēs katra punkta augstumu apraksta 8 biti jeb viens baits, kura vērtība var būt robežās no 0 līdz 255. Lai palielinātu augstumu kartes precizitāti, iespējams izmantot bitkartes, kas glabā 16 bitus katrā kanālā [47].



1.4.2. att. **Augstumu kartes projekcija 3 dimensijās**

1.5. Procedurāla satura ģenerēšana

Procedurāla satura ģenerēšana (*procedural content generation*) ir programmatiska lietojumu satura veidošana izmantojot pseidonejaušus procesus [51]. Galvenais procedurālās

satura ģenerēšanas "Ahileja papēdis" ir tā kontrolējamība. Procedurāli ģenerētam objektam ir jāatbilst lietotāja prasībām, taču lietotājam nav iespēju kontrolēt uzģenerētā satura rezultātu [62].

Procedurāli ģenerēts saturs ir saturs, kas tiek ģenerēts programmas izpildes laikā, t.i. tas nav iepriekš sagatavots. Parasti procedurālā ģenerēšana tiek lietota datorgrafikā, kā arī datorspēļu līmeņu ģenerēšanai. Bieži vien tiek lietots jēdziens "procedurālā sintēze", jo, tieši tā pat, kā elektroniskā skaņa tiek sintezēta ar sintezatoru, procedurāli objekti tiek sintezēti ar algoritmu palīdzību. Tieši tāpēc, runājot par procedurālu satura ģenerēšanu, vārds "ģenerēšana" ir sinonīms vārdam "sintēze" [44]. Lai procedurāli ģenerēti objekti izskatītos reālistiski, to ģenerēšanā bieži vien tiek lietoti trokšņi. Piemēram, spēles lieto trokšņus lai padarītu cilvēku modelētus objektus reālistiskākus [8].

Procedurāli ģenerēta satura galvenās priekšrocības ir:

1. Tas ļauj daudz ātrāk sagatavot saturu grafiskiem lietojumiem
2. Tas ļauj veidot daudzveidīgu saturu, piemēram, katrā izpildes laikā ģenerējot unikālu saturu.
3. Tas ļauj samazināt satura aizņemto vietu uz datu nesējiem.

Procedurāla satura ģenerēšanu iespējams pielietot ne tikai datorgrafikas ģenerēšanai, bet ar tās palīdzību iespējam procedurāli ģenerēt arī, piemēram, datorspēles scenāriju, mūziku vai pat spēles noteikumus [53].

1.5.1. Procedurāla reljefu ģenerēšana

Līdzīgi kā procedurālā objektu ģenerēšanā, arī reljefu ģenerēšanā tiek izmantoti trokšņu algoritmi. Ja objektiem trokšņi tiek lietoti, lai tos padarītu reālistiskākus, tad reljefi parasti pilnībā balstās uz trokšņiem [8]. Tā kā reljefi parasti satur daudz trīsstūru, visa reljefa renderēšana patērē daudz grafiskā procesora (*Graphics processing unit*) laika. Lai samazinātu patērētā laika daudzumu, tiek pilnveidoti ģenerēšana un renderēšanas paņēmieni. Reljefu renderēšanas pilnveidošanai tiek izmantota viena vai vairākas no sekojošajām metodēm:

1. Viss reljefs tiek uzģenerēts algoritma sākumā un sadalīts apakšobjektos tiek glabāts

atmiņā, un tiek renderēts tikai redzamais apakšobjekts;

2. Tiek dinamiski ģenerēta un glabāta atmiņā tikai reljefa daļa, kas ir redzama kamerai;
3. Reljefa renderēšanai tiek izmantoti detalizācijas līmeņi (*LOD – level of detail*).

Ja reljefs nav liels, tad tā dalīšana apakšobjektos un redzamā apakšobjekta attēlošana ir vispārpieņemts paņēmiens, jo netiek tērēts procesora laiks reljefa ģenerēšanai izpildes laikā (*runtime*).

Ja visu reljefu nav iespējams glabāt atmiņā, tad tas tiek ģenerēts dinamiski tad, kad tas ir redzams kamerai. Dinamiska reljefa ģenerēšana parasti izmanto maz atmiņas, taču salīdzinoši daudz procesora laika.

Izmantojot detalizācijas līmeņus, tālāk esošie reljefa trīsstūri tiek sapludināti un pārveidoti par vienu trīsstūri. Jo tālāk no kameras atrodas trīsstūri, jo vairāk trīsstūru tiek sapludināti. Šī metode ļauj ietaupīt grafiskā procesora laiku, jo tiek attēlots mazāks skaits trīsstūru, taču tiek patērēts lielāks daudzums datora atmiņas, jo tajā tiek glabāts vairāk informācijas par katru trīsstūri [10].

1.6. Saistītie pētījumi un izmantošana

Procedurālai satura ģenerēšanai ir salīdzinoši gara vēsture, pirmais komerciālais procedurālā satura lietojums ir 1993. gada spēlē Frontier: Elite II, kurā procedurāli tika ģenerētas zvaigžņu sistēmas [14]. Procedurālo metožu sarakstā ir tādas tehnoloģijas kā dažādi trokšņu algoritmi, fraktāļi, L sistēma (*Lindenmayer system*) un ģeometrisko figūru ģeneratori. Šīs tehnoloģijas tiek bieži lietotas tādās sistēmās kā mākoņu, koku un citu dabas elementu ģenerēšana [55].

Makrī (*McCree*) un Palisters (*Pallister*) savā pētījumā apraksta procedurālu 3D dabas skatu ar kokiem un mākoņiem ģenerēšanu izmantojot Perlina troksni. Prototips demonstrē procedurāli ģenerētu reljefu ar kokiem un divdimensionālu mākoņu slāni. Zemes reljefs var tikt brīvi apskatīts horizontālā plaknē reālā laikā, un tas tiek ģenerēts tikai noteiktā rādiusā ap kameru, tādā veidā netiek tērēts laiks neredzamās reljefa daļas ģenerēšanai un renderēšani [50].

Maurīcs Danahers savā pētījumā "Dynamic landscape generation using page management" apraksta kā attēlot milzīgu, dinamiski ģenerētu reljefu tā, lai tas izmantotu pēc iespējas mazāk atmiņas un datora grafiskā procesora laiku. Viņš savā pētījumā izmanto lapu pārvaldības (*page management*) tehniku, ar kuras palīdzību tiek ģenerētas noteikta daudzuma lapas ar reljefiem ap kameru. Kad kamera pārvietojas, neredzamās lapas tiek dzēstas, un tiek ģenerētas jaunas [12].

Jakobs Olsens pētījumā "Realtime Procedural Terrain Generation" apraksta algoritmus dinamisku reljefu ģenerēšanai, kas izmanto Voronoja diagrammu, Perlina troksni, perturbāciju un erozijas algoritmus pielietojumiem datorspēlēs [56]. Sistēma, kas saucas CityEngine izmantot uzlabotu L sistēmu, lai ģenerētu pilsētas modeļus. Sistēma izmanto hierarhisku likumu kopu, lai ģenerētu ielu un ēku modeļus. Tomēr CityEngine neatbalsta reālā laika ģeometrijas ģenerēšanu [43].

Lielākoties tomēr procedurālo procesu pielietošana komerciālos produktos un spēlēs nav populārs, jo izveidot algoritmu, kas ģenerē reālistisku un kvalitatīvu saturu ir ļoti sarežģīti. Liela daļa procedurālo metožu, piemēram, tekstūru ģenerēšana, paļaujas uz pseidonejaušiem skaitļiem [50].

1.7. Izmantošana projektos

Nelielā pieejamās atmiņas apjoma uz vecākiem datoriem dēļ, procedurālā satura ģenerēšana jau daudzus gadus ir ļoti populāra arī datorspēlēs. Spēles The Sentinel veidotāji apgalvo, ka spēlei ir desmitiem tūkstoši unikālu līmeņu, kas glabājas tikai pāris kilobaitos [58]. Tāds piemēri kā .theprodukkt izstrādātā spēle .kkrieger spilgti ataino procedurālās satura ģenerēšanas iespējās, tas spēj ģenerēt 3D pasauli, muzikālo pavadījumu, tekstūras, gaismas un pat ierobežotu pirmās personas šaušanas spēles mehāniku. Pie tam .theprodukkt un .kkrieger aizņem attiecīgi tikai 64 un 96 kilobaitus uz datu nesējiem. Vislabākais procedurāli ģenerētā satura piemērs komerciālā produktā ir spēlē Spore, kurā varoņu animācijas tiek ģenerētas procedurāli, balstoties uz lietotāja ievadītajiem parametriem [54].

Spēlē Minecraft vienspēlētāja (*single player*) režīmā saturs tiek procedurāli ģenerēts izmantojot izstieptu 3D Perlina troksni ar lineāru interpolāciju. Izstiepšana tiek izmantota, lai

taupītu datora resursus, taču tā arī izslēdz iespēju, ka pasaulē parādīsies fragmenti, kas sastāv tikai no viena bloka. Minecraft pasaule sastāv no biomiem (*Biome*). Katram biomam ir savas ģeogrāfiskas īpašības, piemēram, flora, augstums, temperatūra, mitrums, debesu krāsa un pat lapu krāsa. Biomi Minecraft tiek būvēti izmantojot Vitakera diagrammu (*Whittaker diagram*), kuras pamatā ir mitruma un temperatūras piešķiršana katram uzģenerētajam biomam [48].

Codemasters izstrādātā spēle FUEL 2009. gadā ieguva Ginesa rekordu kā spēļu konsoļu spēle ar vismilzīgāko pasauli, kuras izmēri ir 5560 kvadrātjūdzes (14400km^2) [19]. Lai arī Codemasters . Tika aprēķināts, ka, lai glabātu šādu reljefu un tā pārklājuma tekstūras, būtu nepieciešami 40 DVD diski, līdz ar to var secināt, ka lielākā daļa reljefa tiek ģenerēta procedurāli [20].

Vietnē, kas veltīta procedurālā satura ģenerēšanai var redzēt sarakstu ar lietojumiem, kas saturu ģenerē dinamiski. Tā kā mūsdienās dinamiska satura ģenerēšana spēlēs nav ieguvusi lielu popularitāti, tad saraksts ir salīdzinoši īss, tajā atrodas tikai 5 (4 maksas un 1 bezmaksas) komerciāli produkti, 7 nekomerciāli produkti un 8 atvērta pirmkoda risinājumi [52].

1.8. Lietojums

Galvenokārt zemes reljefa ģenerēšana tiek pielietota datorspēlēs vai dabas skatu ģenerēšanā. Praktiski jebkuru dabas elementu, piemēram, zemi, mākoņus, laikapstākļus, augus un dažādus efektu iespējams ģenerēt procedurāli [47]. Spēlēs procedurālā satura ģenerēšana arī ļauj pagarināt spēles ilgumu, ģenerējot jaunus līmeņus, objektus un citu saturu. Bieži vien procedurāli ģenerēts saturs ir daudz interesantāks par saturu, ko ir veidojuši mākslinieki [51]. Procedurāli ģenerēti reljefi būtiski samazina laiku, kas vajadzīgs māksliniekiem, lai sagatavotu saturu iepriekš [54].

1.9. Kopsavilkums

Apskatītie lietojumi procedurālo satura ģenerēšanu izmato, lai samazinātu laiku, kas tiek veltīts satura sagatavošanai, lai ģenerētu unikālu saturu, kā arī, lai samazinātu programmas aizņemto vietu uz datu nesēja. Procedurālā ģenerēšana arī dod iespēju ietaupīt

grafikas adapteru resursus, jo tiek ģenerēti tikai tie objekti, kas atrodas kameras redzamajā daļā. Tomēr reljefu ģenerēšanas process aizņem daudz procesora laika, tāpēc nepieciešami dažādi pilnveidošana veidi to ģenerēšanai un renderēšanai. Procedurāli ģenerēti reljefi bieži vien neizskatās reālistiski, tāpēc to ģenerēšanai jāizmanto vairāki algoritmi vienlaicīgi, kas savukārt izmanto daudz aparatūras resursu.

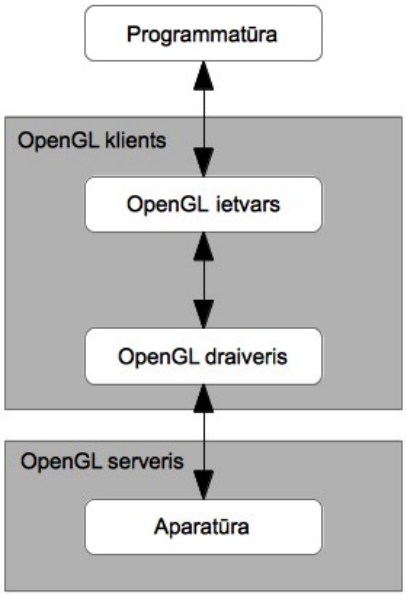
Kopumā pētījumu par procedurāli ģenerētiem objektiem un reljefiem netrūkst, taču tie datorspēlēs un citos vizuālos modeļos tiek izmantoti reti, jo ir sarežģīti, patērē daudz resursu un rezultāts bieži vien nav apmierinošs.

2. PIEDĀVĀTAIS RISINĀJUMS

Lai izveidotu reālistisku reljefu, tiek izmatotas vairākas tehnoloģijas un algoritmi.

2.1. OpenGL

OpenGL (*Open Graphics Library*) ir C valodā rakstīta starpplatformu bibliotēka, kas ļauj programmētājiem veidot programmatūru, kas kas izmanto grafisko aparatūru [5]. OpenGL paredzēts gan 2D, gan 3D grafikas renderēšanai [27]. Programmas, kas izmanto OpenGL bibliotēku komunicē ar OpenGL ietvaru izmantojot C vai kādā citā valodā rakstītu OpenGL implementāciju. Katrs grafiskās aparatūras ražotājs veido aparatūras draiverus, kas tiek izmantoti ietvara komunikācijai ar grafisko aparatūru. Grafiskā aparatūra var uz ekrāna zīmēt OpenGL bibliotēkas izsaukto komandu rezultātu vai arī dot aprēķināto rezultātu atpakaļ programmai (skat. 2.1.1. att.) [37].



2.1.1. att. **OpenGL arhitektūra**

2.1.1. Vēsture

Silicon Graphics radīja OpenGL kā alternatīva viņu pašu izstrādātajam IrisGL

(*Integrated Raster Imaging System Graphics Library*), kas 1990. gadu sākumā bija industrijas standarts. Lai arī IrisGL šajā periodā bija populārs, tā kā tam nebija atvērta standarta, arvien lielāku tirgus daļu iekaroja PHIGS (*Programmer's Hierarchical Interactive Graphics System*). Lai nezaudētu savu tirgus daļu konkurentiem, Silicon Graphics nolēma IrisGL pārveidot par atvērta standarta tehnoloģiju.

1992. gadā Silicon Graphics izveidoja arhitektūras pārskata padomi (*architectural review board*) jeb ARB, kurā ietilpa vadošie industrijas uzņēmumi. ARB mērķis ir uzturēt un paplašināt OpenGL specifikāciju.

OpenGL ir piedzīvojis vairākas versijas, kas lielākoties tikai pievienoja jaunus paplašinājumus esošajam API (*Application Programming Interface*). Piemēram, OpenGL 1.1 pievienoja *glBindTexture* paplašinājumu kodola API.

OpenGL 2.0 ietver būtisku papildinājumu – OpenGL Shading Language jeb GLSL. GLSL ir programmēšanas valoda, kas līdzīga C valodai, un tā ļauj izstrādātājiem programmēt transformācijas un fragmentu konveijeri (*pipeline*). 2.1. versija pievienoja jaunas funkcijas GLSL valodai.

OpenGL 3.0 galvenokārt marķēja lielu daļu funkciju kā novecojušas. Savukārt 3.1. versija pilnībā izvāca novecojušās funkcijas no OpenGL standarta. 3.2. versija ieviesa versiju savietojamības kontekstu [42].

Vai vajag minēt kāpēc es izmantoju OpenGL?

OpenGL tiek izmantots tāpēc, ka tas jau kļuvis par starpplatformu 3D grafikas standartu. Taču risinājumu var pielāgot arī citām 3D grafikas bibliotēkām.

2.1.2. Galvenās OpenGL iezīmes

Galvenā OpenGL iezīme, kas to padarīja par tirgus standartu ir atvērtais standarts, kuru izstrādā lielākie industrijas uzņēmumi. Tā kā OpenGL standarts ir pieejams visiem, šo bibliotēku ir pieejama uz praktiski jebkuras operētājsistēmas un tas ir savietojams ar lielāko daļu modernās grafikas aparatūras [5].

2.1.3. Grafiskās adapteru programmas

Grafisko adapteru programmas mūsdienās aizstāj fiksēto funkciju konveijeri (*Fixed Function Pipeline*), kas automātiski izpildīja tādas darbības kā virsotņu transformācija un apgaismojuma un miglas aprēķināšana [41]. Programmas, kas tiek izpildītas uz grafiskās aparatūras bieži vien tiek sauktas par ēnotajiem (*Shaders*) jeb grafiskajām programmām. OpenGL 1.3 bija pirmā versija, kas ļāva veidot grafikas adapteru programmas. Pirmā valoda, ar kuru bija iespējams veidot grafisko adapteru programma bija ARB assembly valoda, kuru 2002. gadā izveidoja OpenGL ARB (*Architecture Review Board*) [7]. Tā bija zema līmeņa valoda, kas ļāva operēt ar datiem uz grafikas adapteriem, taču līdz ar GLSL ieviešanu, ARB assembly valodas izstrāde un attīstīšana tika apstādināta.

2004. gadā ARB assembly valodu aizstāja GLSL [66]. GLSL (*OpenGL Shading Language*) ir valoda, kas līdzīga C un paredzēta grafiskās aparatūras programmu rakstīšanai [40]. Sākot ar OpenGL 3.1 versiju, GLSL ir vienīgā tehnoloģija, kas ļauj grafiskās aparatūras izmantošanai ar OpenGL bibliotēku [36].

2.1.4. Alternatīvas

Vienīgā alternatīva, ar kuras palīdzību iespējams izmantot grafisko aparatūru ir Microsoft izstrādātā 3D bibliotēka Direct3D. Direct3D ir bibliotēku kopas DirectX sastāvdaļa. Direct3D galvenā priekšrocības ir tāda, ka Microsoft ir izstrādājis daudz rīku, lai atvieglotu programmatūras izstrādi ar Direct3D [39]. Arī grafiskās aparatūras ražotāji veicina Direct3D attīstību veidojot dažādus rīkus priekš tā un palīdzot Microsoft ar savām atsauksmēm par jaunieviestu funkcionalitāti un kļūdu ziņojumiem [69]. Taču galvenais Direct3D trūkums ir tas, ka Direct3D iespējams izmantot tikai uz Microsoft Windows operētājsistēmas [35].

2.1.5. Lietojums risinājumā

Risinājuma izstrādei ir vajadzīga reljefa 3D attēlojums, tāpēc vajadzīga bibliotēka, kas spēj to darīt izmantojot grafikas aparatūru. Tā kā risinājums tiek izstrādās uz OSX operētājsistēmas un OpenGL ir vienīgā pieejamā 3D bibliotēka, kas pieejams šai operētājsistēmai, tā tiks izmantota pētījumā. GLSL tiks izmantota, lai izveidot grafiskās

aparatūras programmas, kas paredzētas tekstūru uzklāšanai uz reljefa.

2.2. Irrlicht 3D dzinis

Irrlicht ir atvērta pirmkoda reāla laika 3D dzinis (*3D engine)*, kas rakstīts C++ programmēšanas valodā. Tas veidots kā starpplatformu bibliotēka, kas izmanto OpenGL, programmatūras (*software),* kā arī uz Windows sistēmām Direct3D renderēšanas API (*Application Programming Interface*) [25].

Irrlich dziņa izstrādi 2003. gadā uzsāka vācu programmētājs Nikolauss Gebhards. 3 gadu laikā dziņa izstrādātāju komanda izauga līdz 10 cilvēkiem. Nikolauss nolēma radīt šo dzini galvenokārt tāpēc, ka populārākie tā laika 3D dziņi bija pārāk sarežģīti, kaut, viņaprāt, tiem tādiem nevajadzēja būt [65].

2.2.1. Lietojums risinājumā

Risinājuma izstrādei nepieciešama vienkārša bibliotēka, kas spēj attēlot 3D objektu projekcijas. Irrlicht 3D ir viegli saprotams un atbalsta tādas populārākās platformas kā Windows, OSX un Linux, tāpēc tas tiks izmantots risinājuma izstrādē.

Galvenokārt Irrlicht 3D tiks izmantots, jo tas atvieglo virsotņu un indeksu buferu izveidi un renderēšanu. Ar Irrlicht 3D tiks izveidots režģis (skat. 1.3.1. att.), tas tiks deformēts atbilstoši uzģenerētajai augstumu kartei un uz tā tiks uzklātas vairākas tekstūras, lai reljefs izskatītos reālistiskāk.

2.3. Pseidonejauši skaitļi

Pseidonejauši skaitļu ģeneratori (*pseudo random number generators*) ir svarīga procedurālo sistēmu sastāvdaļa. Pseidonejaušu skaitļu ģenerators ģenerē virkni ar skaitļiem, kuri atkarīgi no sākotnējās ievadvērtības, sauktas arī par sēklu (*seed*). Procedurālu objektu kontekstā šī sēkla tiek izmantota, lai vienmēr vienā un tajā pašā vietā tiktu uzģenerēts viens un tas pats pseidonejaušs skaitlis, nodrošinot to, ka vieni un tie paši procedurāli ģenerēti objekti vienmēr izskatīsies vienādi [55].

GNU C bibliotēka (*glibc*) ģenerē pseidonejaušus skaitļus izmantojot lineāru algoritmu.

Glibc aprēķina nejaušo skaitļu virknes elementus izmantojot sekojošās formulas, tajās mainīgais s ir sēkla, r_i ir inicializācijas vektori, $2147483647 = 2^{31} - 1$, $4294967296 = 2^{32}$, bet o_i ir rezultāta vērtība:

- $r_0 = s$
- $r_i = (16807 * (\text{signed int}) r_{i-1}) \bmod 2147483647$ (for $i = 1...30$)
- $r_i = r_{i-31}$ (for $i = 31...33$)
- $r_i = (r_{i-3} + r_{i-31}) \bmod 4294967296$ (for $i \geq 34$)
- $o_i = r_{i+344} >> 1$

Glibc nejaušo skaitļu ģenerators izvada 31 bitu pozitīvu skaitli, 32. bits vienmēr tiek atmests t.i. tas ir 0 [64].

2.4. Jaucējfunkcijas

Jaucējfunkcijas (*hash functions*) ir algoritmi, kas attēlo dažāda garuma datu kopas fiksēta izmēra datu kopās. Galvenokārt Jaucējfunkcijas tiek izmantotas datu šifrēšanai, elektroniskajiem parakstiem kā arī, lai pārveidotu kādu datu kopu, sauktu arī par atslēgu, par norādi uz vietu noteiktā datu struktūrā, kurā šos datus atrast [1].

Populārākais jaucējfunkciju pielietojums ir datu glabāšana tabulā, kurai ieraksta adrese tiek aprēķināta izmantojot jaucējfukciju. Taču lielākā problēma ar šādu adrešu aprēķināšanu ir tā, ka vairākām datu kopām var tikt aprēķināta viena un tā pati adrese, saukta arī par sadursmi (*collision*) [23]. Lai maksimāli izvairītos no sadursmēm, un efektīvi izmantotu atmiņu, jaucējfunkciju algoritmi parasti tiek veidot tā, lai vērtības tiktu izkaisītas vienlīdzīgi pa visu atļauto vērtību kopu [26].

2.4.1. FNV jaucējfunkcija

FNV algoritmu izgudroja Glens Fovlers (*Glenn Fowler*) un Fongs Vo (*Phong Vo*) 1991

gadā, kad tas tika aizsūtīts kā pēcskata komentārs IEEE POSIX P1003.2 komitejai. Vēlāk Lendons Kurts Nolls (*Landon Curt Noll*) uzlaboja šo algoritmu. Vēstulē Londonai algoritma autori to nosauca par Fowler/Noll/Vo algoritmu. FNV algoritma pirmā versija tika nosaukta par FNV-1. FNV pseidokods redzams 2.4.1.1. attēlā, *FNV_prime* vērtība pie 32 bitiem ir 16777619, bet *offset_basis* vērtība 2166136261, bet *hash* sevī satur jaucējfunkcijas starprezultātus un rezultātu [16].

```
hash = offset_basis
for each octet_of_data to be hashed
     hash = hash * FNV_prime
     hash = hash xor octet_of_data
return hash
```

2.4.1.1. att. **FNV-1 jaucējfunkcijas pseidokods**

Vēlāk algoritms tika nedaudz uzlabots un nosaukts par FNV-1a versiju. Jaunajā versijā vienkārši tika vietām apmainītas XOR un reizināšanas darbības. Tā kā FNV-1a versijai ir nedaudz labāka izkliede (*dispersion*), tā tiek lietota biežāk kā FNV-1 versija. FNV-1a pseidokods redzams 2.4.1.2. attēlā [16].

```
hash = offset_basis
for each octet_of_data to be hashed
     hash = hash xor octet_of_data
     hash = hash * FNV_prime
return hash
```

2.4.1.2. att. **FNV-1a jaucējfunkcijas pseidokods**

FNV algoritma galvenā priekšrocība ir tā ātrums. Taču šim algoritmam ir arī vairāki būtiski trūkumi, kas neļauj to izmantot kriptogrāfijā:

1. Tā kā FNV algoritms ir ļoti ātrs, tad rupja spēka (*brute force*) uzbrukums tam ir ļoti viegls, un tam var ātri atrast sadursmes.
2. Tā kā algoritmā galvenokārt tiek izmantotas XOR un reizināšanas darbības, tad tas ir ļoti jūtīgs pret 0 vērtībām. Ja kaut vienā iterācijā tā vērtība ir 0, visas pārējās

iterācijas atgriezīs 0.

3. Ideālā jaucējfunkcijā visiem ievades baitiem būtu vienādi jāietekmē rezultātu, taču FNV algoritmā labās puses baiti rezultātu ietekmē vairāk par kreisās puses baitiem [63].

2.4.2. Vanga jaucējfunkcija

Tomass Vangs (*Thomas Wang*) ir izgudrojis ātru jaucējfunkciju viena skaitļa pārveidošanai par citu, kuru viņš nosauca par Vanga jaucējfunkciju (*Wang's Mix Function*). Viņš piedāvā ievadvērtību noteiktu skaitu reižu pārbīdīt par noteiktu skaitu bitu un izmantot XOR un NOT bitu operācijas [2]. Šī funkcija ļoti labi izkliedē vērtības pa visu vērtību kopu, tāpēc labi noder nejaušu skaitļu sēklas (*seed*) ģenerēšanai [55]. Attēlā 2.4.2.1. redzams Vanga 32 bitu jaucējfunkcijas pseidokods, kurā *key* ir ievadvērtība, bet *output* ir funkcijas izvadvērtība.

```
output += ~(key << 15);

output ^= (output >> 10);

output += (output << 3);

output ^= (output >> 6);

output += ~(output << 11);

output ^= (output >> 16);
```

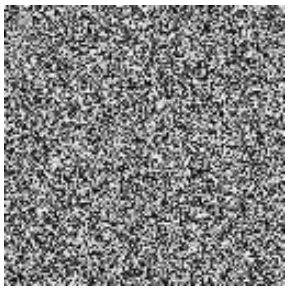
2.4.2.1. att. **Vanga jaucējfunkcijas pseidokods**

2.4.3. Lietojums risinājumā

Jaucējfunkcija risinājumam ir nepieciešama, lai ģenerētu katram reljefa punktam savu pseidonejaušu skaitli. Jaucējfunkcijas rezultāts tiks padots pseidonejaušu skaitļu ģeneratoram kā sēkla. Tā kā Vanga jaucējfunkcija ir ātrāka par FNV jaucējfunkciju [22], un tās ģenerētās vērtības ir labi izkliedētas [55], tad tā tiks izmantota risinājuma izstrādē.

2.5. Trokšņu algoritmi

Trokšņa algoritmi mākslīgu trokšņu, kas bieži tiek izmantoti tekstūru ģenerēšanai. Trokšņu algoritmi ir paredzēti koherentu trokšņu ģenerēšanai telpā. Koherents nozīmē, ka starp 2 punktiem trokšņa vērtība, pārvietojoties no viena punkta uz otru, mainās gludi (*smoothly*). Tas nozīmē, ka starp diviem telpas punktiem nekad nebūs liela vērtības "lēciena". Attēlā 2.5.1. redzams nekoherents troksnis.



2.5.1. att. **Nekoherents troksnis**

Trokšņa funkcijām parasti tiek padots viens vai vairāki argumenti. Trokšņa argumentu skaits ir vienāds ar datu dimensiju skaitu [62].

2.5.1. Režģu trokšņi

Režģu trokšņi ir viena no populārākajām trokšņu algoritmu kategorijām. Režģa trokšņu pamatā ir plaknes vai telpas sadalīšana režģī. Režģu trokšņi dotajiem punktiem atrod režģa kvadrātu (skat. 2.5.3.1. att.), katram šī kvadrātam piešķir pseidonejaušu vērtību un interpolē starp šīm vērtībām [62].

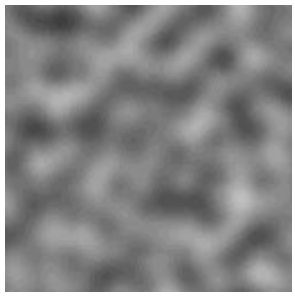
2.5.2. Perlina troksnis

Ņujorkas universitātes profesors Kens Perlins (*Ken Perlin*) pasauli ar Perlina trokšņa algoritmu iepazīstināja 1985. gadā [34]. Šis algoritms paredzēts vizuālo efektu ģenerēšanai ar datora palīdzību [28]. Perlina trokšņa algoritms ir viens no populārākajiem režģu trokšņu algoritmiem [62].

Perlina troksnis jau ir kļuvis par standartu "trokšņu industrijā", un tam ir vairāki iemesli. Perlina trokšņa algoritms ir ātrs, izmanto maz atmiņas, un tas ģenerē augstas

kvalitātes rezultātu. Taču tas ir nedaudz sarežģītāks par citiem populārajiem trokšņa algoritmiem [8].

Dimensiju skaitam nav ierobežojumi, 2.5.2.1. attēlā redzams divu dimensiju Perlina troksnis [8].



2.5.2.1. att. **Perlina troksnis**

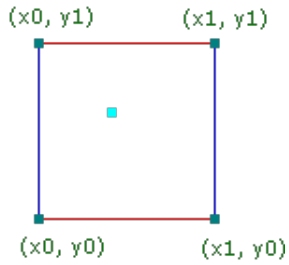
2.5.3. Perlina trokšņa algoritms

Perlina trokšņa algoritmam kā parametrs tiek padots viens vai vairāki decimāli skaitļi, no kuriem tiek izrēķināta trokšņa vērtība attiecīgajā punktā. Perlina trokšņa algoritms atgriež vērtību, kas ir intervālā no -1 līdz 1. Pseidonejaušu skaitļu ģenerēšanai Perlina trokšņa algoritmā tiek izmantota iepriekš izveidota permutāciju tabula [71]. Permutāciju tabula Perlina trokšņa algoritmā ir virkne, kas sastāv no 256 8 bitu skaitļiem. Permutāciju tabula tiek veidota secīgi aizpildot to ar skaitļiem no 0 līdz 255, kas pēc tam tiek sajaukti ar pseidonejaušu skaitļu ģeneratora palīdzību. Permutāciju tabula tiek izmantota, lai tiktu ģenerēti pēc iespējas dažādāki skaitļi, jo divās blakus koordinātās nedrīkst atrasties divi vienādi pseidonejauši skaitļi [46].

Lai aprēķinātu trokšņa vērtību dotajā punktā, tiek izpildīti sekojoši soļi:

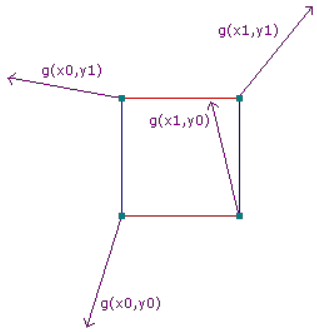
1. Tiek atrasti dotā punkta apkārtējie punkti, kas veido ieskaujošo kvadrātu vai kubu;
2. Katram apkārtējam punktam tiek atrasts slīpuma (*gradient*) vektors;
3. Katram apkārtējam punktam tiek izrēķināts skalārais reizinājums starp punkta slīpuma vektoru un distances vektoru līdz šim dotajam punktam;
4. Rezultāts tiek aprēķināts lineāri interpolējot starp vektoru skalārajiem reizinājumiem.

Pirmajā solī, katra dotā punkta koordināte tiek apaļota uz augšu un uz leju, tādā veidā iegūstot kvadrātu vai kubu, kas ieskauj šo punktu. 2.5.3.1. attēlā redzams piemērs, kā divās dimensijās izskatās ieskaujošais kvadrāts.



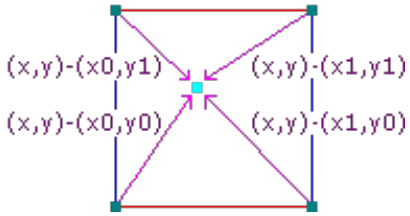
2.5.3.1. att. **Dotā punkta apkārtējie punkti**

Kad visi apkārtējie punkti atrasti, katram tiek izveidots pseidonejaušs slīpuma vektors. Slīpuma vektora piemērs redzams 2.5.3.2. attēlā.



2.5.3.2. att. **Ieskaujošā kvadrāta slīpuma vektori**

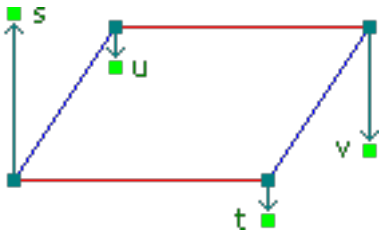
Pēc slīpuma vektoru iegūšanas, tiek aprēķināti vektori no katra apkārtējā punkta līdz dotajam punktam (skat. 2.5.3.3. att.), un katrs no šiem vektoriem tiek skalāri sareizināts ar attiecīgā apkārtējā punkta slīpuma vektoru.



2.5.3.3. att. **Apkārtējo punktu vektori līdz dotajam punktam**

Skalāri sareizinot katra apkārtējā punkta slīpuma vektoru un vektoru līdz dotajam

punktam, tiek iegūts katra apkārtējā punkta svars (skat. 2.5.3.4. att.).

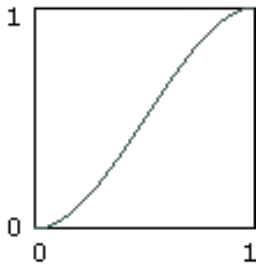


2.5.3.4. att. **Apkārtējo punktu svari**

Lai iegūtu Perlina trokšņa vērtību, visi apkārtējo punktu svari savā starpā ir jāinterpolē. Ja dimensiju skaits ir lielāks par 2, tad apkārtējo punktu skaits ir vismaz 4, līdz ar to, tiek veidota interpolācija starp vairākām vērtībām. Lai to izdarītu, tiek ņemtas pirmās divas vērtības, interpolētas savā starpā, un rezultāts tiek interpolēts ar nākošo divu vērtību interpolācijas rezultātu. Lai vērtību pāreja būtu pēc iespējas gludāka, tiek izmantota lineārā interpolācija (skat. 2.5.3.5. att.), kurai kā parametrs tiek izmantots S līknes (*S-curve*) rezultāts. Perlina trokšņa algoritms S līknes aprēķināšanai izmanto formulu $3p^2 - 2p^3$, kas izvada samērā lēzenu rezultātu (skat. 2.5.3.6. att.) [71].

```
Sx = 3*(x - x0)*(x - x0) - 2*(x – x0)*(x – x0)

a = s + Sx(t - s)
```

2.5.3.5. att. **Lineāra interpolācija izmantojot S līkni**



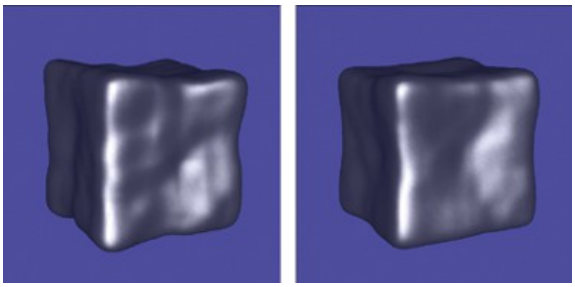
2.5.3.6. att. **S līkne**

2.5.4. Uzlabotais Perlina troksnis

2002. gadā Kens Perlins izveidoja Perlina trokšņa algoritma uzlabotu versiju. Jaunais algoritms ģenerē troksni par aptuveni 10% ātrāk, kā arī novērš problēmas, kas saistītas ar

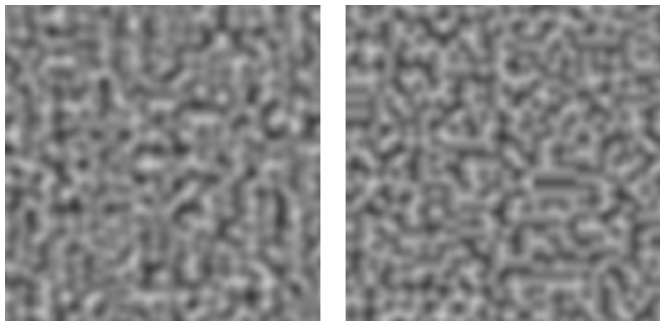
trokšņa matemātiska atvasinājuma izmantošanu, piemēram, tekstūru ģenerēšanā.

Uzlabotais Perlina S līknei izmanto formulu $6t^5 - 15t^4 + 10t^3$, tādā veidā formulas otrās pakāpes atvasinājums ir 0, ja $t = 0$, tāpēc, matemātiski atvasinot Perlina trokšņa algoritmu, rezultātā neveidojas kropļojumi (skat. 2.5.4.1. att.) [74].



2.5.4.1. att. **Oriģinālā (pa kreisi) un uzlabotā (pa labi) algoritma atvasinājums**

Uzlabotais Perlina trokšņa algoritms kā slīpuma vektorus vienmēr izmanto noteiktas konstantes, nevis katru reizi tās aprēķina. Izmantojot konstantes slīpuma vektoriem izmanto mazāk procesora resursu, taču rezultāts ir ļoti līdzīgs oriģinālajam algoritmam (skat. 2.5.4.2. att.).



2.5.4.2. att. **Oriģinālā (pa kreisi) un uzlabotā (pa labi) algoritma rezultāts**

2.5.5. Slīpumu troksnis

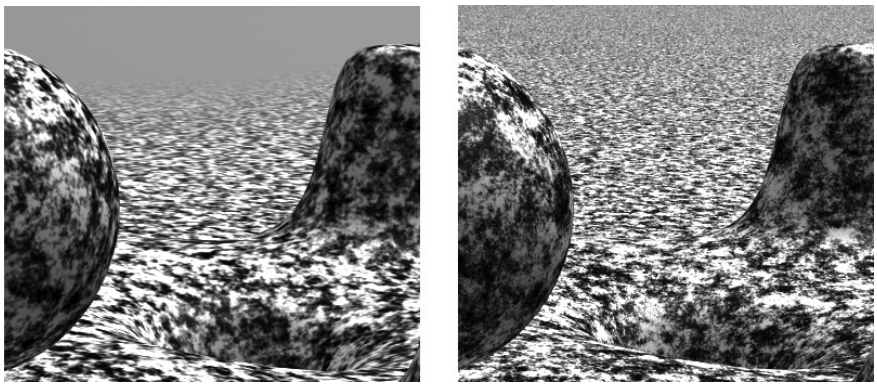
Slīpumu trokšņa (*Gradient noise*) algoritms, tā pat kā Perlina trokšņa algoritms, ir režģa troksnis. Līdzīgi kā Perlina trokšņa algoritmā, slīpumu trokšņa pamatā ir dotā punkta apkārtējo punktu atrašana (skat. 2.5.3.1. att.) un slīpuma vektora piešķiršana katram no tiem (skat. 2.5.3.2. att.). Atšķirībā no Perlina trokšņa, slīpumu troksnis rezultāta aprēķināšanai interpolē starp apkārtējo punktu slīpuma vektoriem. Tā kā slīpumu troksnis izmanto mazāk

operāciju, tas arī ir daudz ātrāks par Perlina troksni, taču tā rezultāts ir nekvalitatīvāks (skat. 2.5.5.1. att.) [62].

2.5.5.1. att. **Slīpumu trokšņa rezultāts**

2.5.6. Vilnīša troksnis

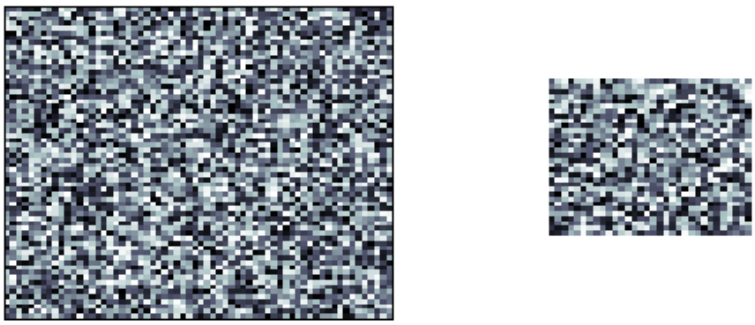
Vilnīša troksni (*wavelet noise*) ir izveidojusi animācijas studija Pixar. Vilnīša troksni galvenokārt tika veidots kā Perlina trokšņa aizstājējs [3]. Viens no lielākajiem Perlina trokšņa trūkumiem ir zemā detalizācija, kas pamanāma kā defekti, kad trokšņa attēla detalizācija tiek samazināta. Perlina trokšņa trūkumus var redzēt, ja tas tiek izmantots kā tekstūra, jo lielā attālumā tekstūras kvalitāte tiek samazināta, un Perlina trokšņa tekstūrā veidojas vizuāli defekti (skat. 2.5.6.1. att.) [68].



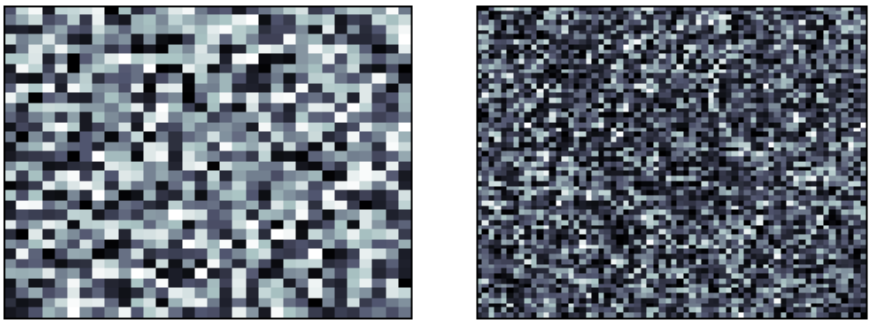
2.5.6.1. att. **Vilnīša troksnis (pa kreisi) un Perlina troksnis (pa labi)**

Vilnīša trokšņa algoritms nav režģa algoritms, t.i. tas gala vērtības aprēķināšanai neizmanto režģi. Vilnīša trokšņa algoritms tā darbības inicializācijas posmā uzģenerē trokšņa tekstūru. Šī trokšņa tekstūra tiek samazināta, pēc tam palielināta līdz oriģinālajam izmēram,

tādā veidā iegūstot tekstūru, kas sastāv no lieliem kvadrātiem (skat. 2.5.6.2. att.). Lai aprēķinātu gala vērtību, samazinātā un atkal palielinātās tekstūras katra pikseļa vērtība tiek matemātiski atskaitīta no oriģinālās tekstūra (skat. 2.5.6.3. att.).



2.5.6.2. att. **Oriģināla un samazinātā trokšņa tekstūra**



2.5.6.3. att. **Palielināta samazinātā trokšņa tekstūra un gala trokšņa tekstūra**

Kad vilnīša trokšņa tekstūra ir uzģenerēta, to iespējams izmantot trokšņa vērtību ģenerēšanai. Trokšņa vērtības tiek ģenerētas interpolējot starp dotā punkta blakus esošiem punktiem trokšņa tekstūrā [49].

2.5.7. Pielietojums

Trokšņa algoritmus bieži lieto datorgrafikas vizuālo efektu ģenerēšanai. Populārākais trokšņa pielietojums ir tekstūru ģenerēšana (skat. 2.5.7.1. att.). Ar trokšņu algoritmu palīdzību var ģenerēt ne tikai statiskas tekstūras, bet to iespējams lietot arī tekstūru animāciju ģenerēšanai [28].

2.5.7.1. att. **Ar Perlina troksni ģenerēta uguns tekstūra**

Perlina troksnis arī tiek lietots virtuālas dzīvības simulēšanai. Piemēram, ar 1D Perlina trokšņa palīdzību tiek ģenerētas cilvēku locītavu kustības [45].

Viens no populārākajiem trokšņa pielietojumiem ir dabas skatu ģenerēšanā. Ar trokšņa algoritmu palīdzību iespējams ģenerēt zemes reljefus, dažādus augus un mākoņus (skat. 2.5.7.2. att.). Tā kā Perlina troksnis ir neatkarīgs no dimensiju skaita, tad ar to var ģenerēt gan 2D, gan 3D dabas skatus [71].

2.5.7.2. att. **Ar Perlina troksni ģenerēti 2D mākoņi**

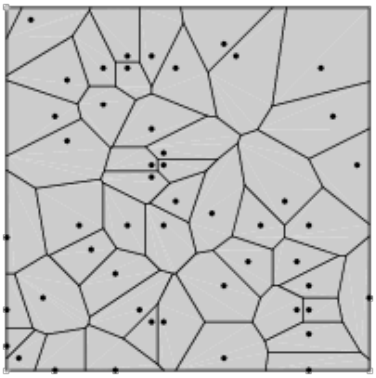
2.5.8. Lietojums risinājumā

Lai arī Vilnīša troksnim ir mazāk defektu samazinot tā kvalitāti, tomēr tas ir nedaudz lēnāks par uzlaboto Perlina troksni, jo Vilnīša troksnis veic 27 aritmētiskās darbības, bet uzlabotais Perlina troksnis tikai 24 [68]. Tā kā detalizācijas samazināšana risinājumam nav svarīga, bet svarīgs ir algoritma ātrums, tad pamata slāņa ģenerēšanai risinājumā tiks izmantots uzlabotais Perlina troksnis. Pamata slānis netiks ģenerēts ar slīpumu trokšņa algoritmu, jo tā rezultāts ir pārāk nekvalitatīvs. slīpumu troksnis tiks izmantots perturbācijas algoritmā, jo eksperimentos tā rezultāts uzrādīja augstāku reljefa erozijas koeficientu un

veiktspēju.

2.6. Voronoja diagramma

Voronoja (*Voronoi*) diagramma sadala plakni ar *n* punktiem izliektos daudzstūros tā, ka katrs daudzstūris satur tieši vienu no iepriekš uzģenerētiem punktiem un katrs punkts šajā daudzstūrī atrodas tuvāk savam uzģenerētajam punktam kā citiem. Dažkārt Voronoja diagrammu sauc arī par Dirihleta (*Dirichlet*) dalīšanu. Daudzstūri jeb šūnas reizēm tiek saukti arī par Dirihleta reģioniem, Tisena (*Thiessen*) politipiem vai Voronoja daudzstūriem [29]. Voronoja diagrammu iespējams izmantot ne tikai plaknes dalīšanai daudzstūros, bet to var izmantot arī vairāku dimensiju telpas dalīšanai. Voronoja diagrammas rezultāts plaknē redzams 2.6.1 attēlā.



2.6.1. att. **Voronoja mozaīka**

2.6.1. Vēsture

Voronoja (*Voronoi*) dalīšana (*partitioning)* jau 1644. gadā izmantoja Renē Descartes (*René Descartes*), bet 1850. gadā Dirihlets (*Dirichlet*) tās izmantoja, lai pētītu pozitīvas kvadrātiskās formas. Pats Voronojs tās pētīja 1907. gadā un pacēla Voronoja diagrammu pētīšanu pavisam jaunā līmenī. Voronoja diagrammām ir plašs pielietojums datorgrafikā, epidemioloģijā, ģeofizikā un meteoroloģijā. Viens no nozīmīgākajiem Voronoja diagrammu pielietojumiem bija 1854. gada Londonas holēras epidēmijas analīzē, kad fiziķis Džons Snovs (*John Snow*) atklāja stingru korelāciju starp inficēšanās un nāves gadījumiem un ūdens pumpi

Brodstrītā [29].

2.6.2. Pielietojums

Voronoja diagrammas parasti tiek pielietotas tādās ar dabu un zinātni saistītās nozarēs kā antropoloģija, arheoloģija, bioloģija, kartogrāfija, ķīmija, ģeogrāfija, ģeoloģija, ģeometrija, mārketings, matemātika, meteoroloģija, fizioloģija, robotika , statistika un zooloģija. Galvenokārt visās šajās nozarēs Voronoja diagrammas tiek lietotas, lai sadalītu reģionu vai datus noteiktās daļās un analizētu šo reģionu ietekmi vienam uz otru.

Voronoja diagrammas arī tiek lietotas dažādu efektu ģenerēšanai, tai skaitā arī tekstūru ģenerēšanai. Parasti Voronoja diagrammas tiek izmantotas, lai panāktu izskatu, kas līdzinās šūnām [62]. IT nozarē Voronoja diagrammas tiek izmantotas galvenokārt, lai risinātu problēmas kas saistītas ar tuvāko kaimiņu meklēšanu [21]. Taču nereti Voronoja diagrammas arī tiek lietotas, lai modificētu augstuma kartes (*heightmap*), tādā veidā iegūstot mazāk viendabīgus rezultātus.

2.6.3. Algoritms

Pastāv vairāki algoritmi kā aprēķināt Voronoja diagrammu, taču šajā pētījumā tiks izmantots sekojošs algoritms. Katram punktam plaknē, izmantojot tuvakā kaimiņa meklēšanas algoritmu, tiek atrasts tuvākais ģenerētais punkts, un plaknē tiek saglabāts attālums līdz šim punktam. Voronoja algoritms 2 dimensiju plaknei redzams 2.6.3.1 attēlā [11].

```
for y = 0 to sizeY - 1
  for x = 0 to sizeX - 1
    point = NearestNeighbor(x, y)
    data[x][y] = point.distance
```

2.6.3.1. att. **Voronoja algoritms**

2.6.4. Lietojums risinājumā

Voronoja diagramma tiks lietota risinājuma izstrādē, lai deformētu augstumu karti. Voronoja diagramma tiks izmantota, lai sadalītu augstumu karti reģionos. Katram augstumu reģionam tiks piešķirts augstums, kas pacēlumus un ieplakas augstumu kartē.

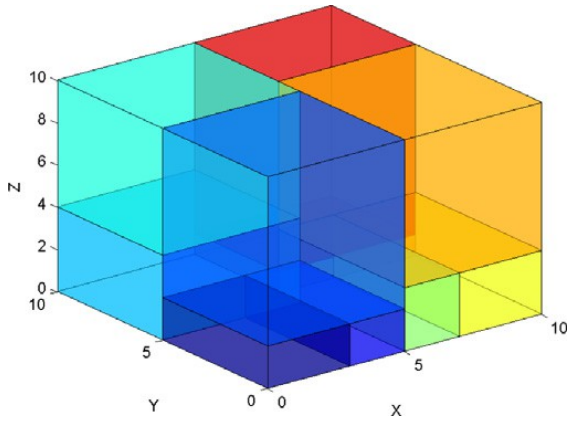
2.7. Tuvākā kaimiņa atrašana

Tuvāko kaimiņu (*NN – Nearest Neighbour*) meklēšanas algoritmi mērķis ir apstrādāt objektu kopu un atrast no tiem objektu, kas atbilst dotajiem kritērijiem. Parasti tuvāko kaimiņu meklēšanu izmanto, lai risinātu ģeometriskas problēmas, t.i. objekta atrašanai, kas atrodas vistuvāk dotajam punktam. Taču tuvako kaimiņu meklēšanu var izmantot arī informācijas iegūšanai, meklēšanai attēlu datubāzēs, vienādu failu meklēšanai mājaslapās, u.c. [32].

2.7.1. KD koks

KD (*k dimensional*) koka algoritms tiek izmantots tuvāko kaimiņu atrašanai. Šī algoritma pamatā ir datu struktūra, kas glabā galīgu skaitu punktu *k* dimensiju telpā. To 1975. gadā izgudroja Džons Luiss Bentlijs [30]. S. Omuhundro izteicis viedokli, ka KD koks ir labākais veids kā paātrināt neironu tīklu apmācību [13].

KD koks ir binārs koks. KD koka algoritms paredzēts, lai atrastu dotajām koordinātām tādu elementu no doto elementu saraksta, par kuru nav neviena tuvāka elementa šīm koordinātām. Jāņem vērā, ka vairāki elementi var būt vienādā attālumā no dotajām koordinātām, tādā gadījumā KD koka tuvākā kaimiņa meklēšanas algoritms atgriež vienu no abiem elementiem. Algoritma pamatā ir telpas sadalīšana reģionos ar noteiktām plaknēm, lai pēc iespējas vieglāk atrastu reģionu, kurā atrodas tuvākais elements (skat. 2.7.1. att.).



2.7.1. att. **Telpa, kas sadalīta reģionos ar KD koka algoritmu**

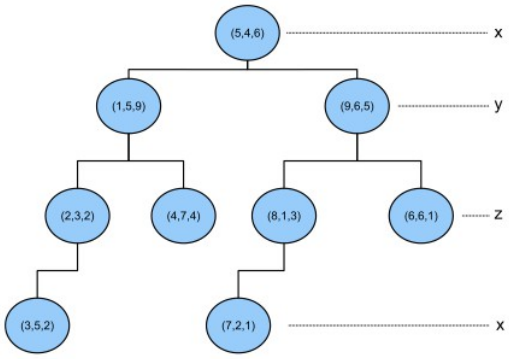
2.7.2. KD koka konstruēšana

Konstruējot KD koku, elementi, kas nesatur apakšelementus tiek saukti par lapām (*leaf*), bet elementi, kas satur apakšelementus tiek saukti par apakškokiem (*sub-tree*). Konstruējot koku, katrs elements, kas nav lapa, izveido plakni, kas sadala telpu divās daļās. Kreisais apakškoks satur elementus, kas atrodas vienā atdalošās plaknes pusē, bet labais apakškoks satur elementus, kas atrodas tās otrā pusē. Atkarībā no dziļuma, katram elementam tiek izvēlēta vienas dimensijas ass. Sadalošā plakne ir perpendikulāra izvēlētās dimensijas asij [6].

Lai izveidotu KD koku, katram eksemplāram tiek izveidots elements KD kokā, pēc tam no šiem elementiem tiek veidots koks, katram no elementiem izpildot sekojošas darbības:

1. Saraksts tiek sakārtots pēc vienas koordinātas;
2. Tiek aprēķināta sadalošā plakne;
3. Visi apakšelementi tiek sadalīti divās daļās atkarībā no tā, kurā sadalošās plaknes pusē tie atrodas;
4. Katras puses tuvākais elements tiek saglabāti kā kreisā apakškoka stumbrs un labā apakškopa stumbrs.

Visas šī darbības tiek atkārtotas tik ilgi, kamēr visiem elementiem izņemot lapām tiek atrasti apakškoki [30]. Attēlā 2.7.2.1. redzams 3D KD koka elementi.



2.7.2.1. att. **3D KD koka elementi**

2.7.3. Tuvākā kaimiņa meklēšana KD kokā

Tuvākā kaimiņa meklēšanas (*nearest neighbor search*) algoritma mērķis ir atrast tuvāko kaimiņu dotam punktam KD kokā. KD algoritms tuvākā kaimiņa atrašanai patērē *O(log n)* laiku. Ar šī algoritma palīdzību var ātri izslēgt lielus telpas laukumus, kuros noteikti meklējamais elements neatrodas.

Meklēšana KD kokā sastāv no šādiem soļiem:

1. Sākot ar saknes elementu, jāpārvietojas lejup pa koku tādā pat virzienā, kādā elementi tika pievienoti kokam t.i. Sākumā jāapmeklē elements, kas atrodas pa kreisi no pārdalošās plaknes, pēc tam jāapmeklē elements, kas atrodas pa labi;
2. Tikko algoritms sasniedz lapu, tas saglabā to kā pagaidu atbilstošāko elementu;
3. Sākot no lapas elementa, jāpārvietojas atpakaļ augšup un katram elementam jāizdara sekojošas darbības:
    - Ja tekošais punkts ir tuvāks par pagaidu atbilstošāko elementu, tas tiek saglabāts kā pagaidu atbilstošākais elementu;
    - Algoritms pārbauda, vai pārdalošās plaknes otrā pusē atrodas kāds elements, kas ir tuvāks par pagaidu atbilstošāko elementu;
4. Kad algoritms nonāk pie saknes elementa, tad tuvākais kaimiņš ir atrasts un algoritms pārtrauc darbību.

Lai nebūtu jārēķina kvadrātsakni, kas ir laikietilpīga operācija, algoritms distanču salīdzināšanai izmanto kvadrātiskās distances [30].

2.7.4. Alternatīvas

Tā kā risinājumam nepieciešams atrast vairākus tuvākos punktus, tad iespējams izmantot algoritmus, kurus paredzēts modificēt, vai kas ir domāti šādai operācijai. Viena no populārākajām alternatīvām KD kokam ir KNN (*K-Nearest Neighbor*) algoritms.

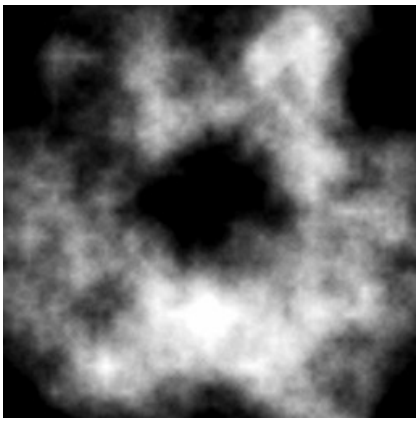
KNN algoritms ir paredzēts datu meklēšanai telpā ar noteiktu skaitu dimensiju. Starp visiem datiem jābūt funkcijai, kas nosaka distanci starp tiem, tai nav obligāti jābūt Eiklīda distancei. Burts K algoritma nosaukumā norāda to, ka algoritms izmanto koeficientu *k*, kas nosaka, cik tuvākos kaimiņus algoritmam būtu jākvalificē. KNN algoritms patērē *O(dn)* laiku, lai atrastu tuvāko kaimiņu, kur *d* ir laiks, kurā tiek aprēķināta distance starp objektiem. [4]

2.7.5. Lietojums risinājumā

Tuvākā kaimiņa meklēšanas algoritms tiks izmantots risinājumā, lai ģenerētu Voronoja diagrammu. KD koka algoritms patērē *O(log n)* laiku, lai atrastu tuvāko kaimiņu, taču KNN algoritms izmanto *O(dn)* laiku, no kā var secināt, ka KD koka algoritms ir ātrāks tuvāko kaimiņu meklēšana. Tā kā KD koka algoritms ir ātrāks par KNN algoritmu, risinājumā tiks izmantots KD koka algoritms.

2.8. Perturbācija

Pertrubācija un sagrozīšana jeb turbulence tiek izmantota, lai deformētu datus un mainītu to īpašības. Perturbācijas pielietojums ir ļoti plašs, piemēram, mākoņu, dūmu, uguns ģenerēšana, ūdens virsmas atspulga deformēšana, plazmas simulēšana, kā arī zemes reljefa deformēšana. 2.8.1. attēlā redzama ar perturbāciju ģenerēta bitkarte, kas paredzēta mākoņu simulācijai [9].



2.8.1. att. **Ar perturbāciju ģenerēti mākoņi**

Zemes reljefa deformācijai perturbācija tiek izmantota, lai izkliedētu augstumu kartes virsotnes ar lielāku frekvenci un veidotu reālistiskāku virsmas imitāciju. Zemes virsmas perturbācijas būtība ir līdzīga okeāna viļņu simulācija, izkliedējot tos ar perturbāciju [15].

Parasti perturbācijas algoritms tiek pielietots grafiskajām tekstūrām (kas sastāv no bitkartes), taču algoritmu pēc tāda paša principa var pielietot arī augstumu kartēm. Perturbācijas funkcijai ir viens parametrs – perturbācijas distance. Perturbācijas algoritms sastāv no divām darbībām, kas tiek izdarītas visiem bitkartes punktiem:

1. Avota punkta aprēķināšana;
2. Avota punkta vērtības saglabāšana rezultāta kartē

Lai aprēķinātu avota punktu, var izmantot pseidonejaušu skaitli, vai kādu no trokšņu algoritmiem [62].

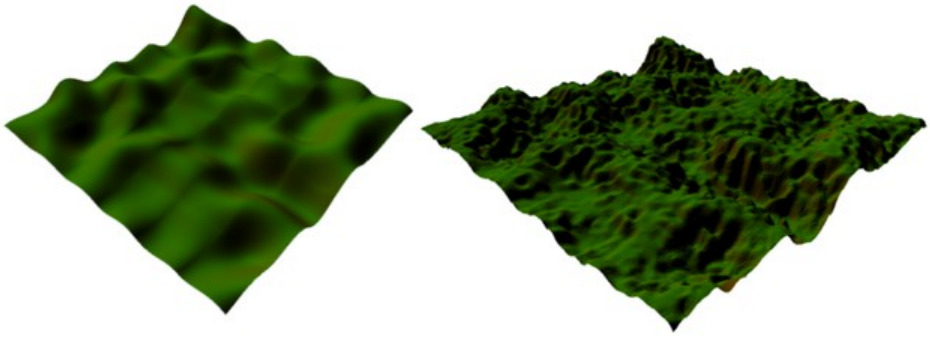
2.8.1. Perturbācija izmantojot Perlina troksni

Perturbācijai izmantojot Perlina troksni avota punkta aprēķināšanai tiek izmantots 2.8.1.1. attēlā redzamais algoritms, kurā *d* ir distances parametrs, *x* un *y* ir tekošā punkta koordinātas, *sizeX* un *sizeY* ir bitkartes izmēri, bet *u* un *v* ir avota punkta koordinātas. Ar funkciju *clamp*, *u* un *v* koordinātas tiek ierobežotas tā, lai koordinātas neatrastos ārpus bitkartes.

```
u = x + (Perlin.Noise(x, y) * d)

v = y + (Perlin.Noise(x, y) * d)

u = clamp(u, 0, sizeX - 1)

v = clamp(v, 0, sizeY - 1)
```

2.8.1.1. att. **Avota punkta atrašana**

Kad aprēķinātas avota punkta koordinātas, vērtība, kas atrodas šajā punktā tiek ierakstītas tekošajās koordinātās. Visi rezultāti tiek rakstīti pagaidu buferī, lai neietekmētu punktus, kas vēl tiks apstrādāti. Kad visi punkti ir apstrādāti, pagaidu buferis tiek pārkopēts uz avota buferī. 2.8.1.2. attēlā redzams zemes reljefs pirms un pēc perturbācijas [15].



2.8.1.2. att. **Zemes reljefs pirms (pa kreisi) un pēc (pa labi) perturbācijas**

2.8.2. Perturbācija izmantojot slīpumu troksni

Perturbācija izmantojot slīpuma vektoru tiek veikta līdzīgi kā izmantojot Perlina troksni. Attēlā 2.8.2.1. att. redzams avota aprēķināšanas algoritms, kurā *d* ir distances parametrs, *x* un *y* ir tekošā punkta koordinātas, *sizeX* un *sizeY* ir bitkartes izmēri, bet *u* un *v* ir avota punkta koordinātas. Atšķirībā no perturbācijas ar Perlin troksni, pirms avota punktu aprēķināšanas ar slīpumu trokšņa palīdzību tiek uzģenerēta kopēja sāls (*salt*). Šī sāls vērtība tiek saskaitīta ar konstantēm 912 un 333 un padota trokšņa funkcijai, kas aprēķina avota punktus *u* un *v* [59].

```
salt = Gradient.noise(x, y);

u = x + Gradient.noise(salt + 912) * d;

v = y + Gradient.noise(salt + 333) * d;

u = clamp(u, 0, sizeX - 1)

v = clamp(v, 0, sizeY - 1)
```

2.8.2.1. att. **Avota punkta atrašana**

2.8.3. Lietojums risinājumā

Testējot perturbācijas algoritmu, tika secināts, ka perturbācijas algoritma izmantošana avota punkta aprēķināšanai patērāja par 0,6% mazāk procesora laika, kā slīpumu trokšņa izmantošana. Taču Perlina trokšņa izmantošana samazināja reljefa erozijas vērtību, bet slīpumu troksnis to palielināja. Tā kā 0,6% procesora laika ir salīdzinoši mazs zaudējums, bet erozijas vērtība ir svarīgs faktors, tad risinājuma izstrādē avota punkta aprēķināšanai tiks izmantots slīpumu trokšņa algoritms.

2.9. Erozijas algoritmi

Lai arī trokšņu algoritmi spēj ģenerēt ļoti gludas augstumu kartes, tomēr Voronoja diagrammas un perturbācijas rezultātā rodas dažādi defekti un asas vērtību pārejas, kas 3D projekcijā izpaužas kā asas klintis. Lai nogludinātu augstumu kartes, tiek izmantoti erozijas algoritmi, kas straujās pārejas augstumu kartēs padara lēzenākas [8].

2.9.1. Erozijas vērtība

Spēlēs un virtuālās realitātes lietojumos parasti ir vēlams tāds zemes reljefs, pa kuru iespējams pārvietoties, tas nozīmē, ka kalnu nogāzes nedrīkst būt pārāk stāvas. Šis likums viens pats nozīmētu, ka pilnīgi plakana karte ir ideāla, taču, lai tā nebūtu, tiek ieviests otrs mainīgais – standarta novirze, kuras aprēķin formula redzama 2.9.1.1. att. [70].

$$\bar{s} = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} s_{i,j}$$

2.9.1.1. att. **Standarta novirzes aprēķināšanas formula**

Lai aprēķinātu vidējo klinšu stāvumu zemes reljefā, tiek izmantota 2.9.1.2. att. redzamā formula. Abās formulās $s_{i,j}$ apzīmē tekošā punkta un zemākā kaimiņa augstumu starpību.

$$\sigma_s = \sqrt{\frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (s_{i,j} - \bar{s})^2}$$

2.9.1.2. att. **Vidējā klinšu stāvuma aprēķināšana**

Erozijas vērtība (*erosion score*) ir galvenais kritērijs, ar kuru iespējams noteikt, cik stāvas klintis ir dotajā zemes reljefā un to aprēķina izdalot vidējo klinšu stāvumu ar standarta novirzi [8].

2.9.2. Termālā erozija

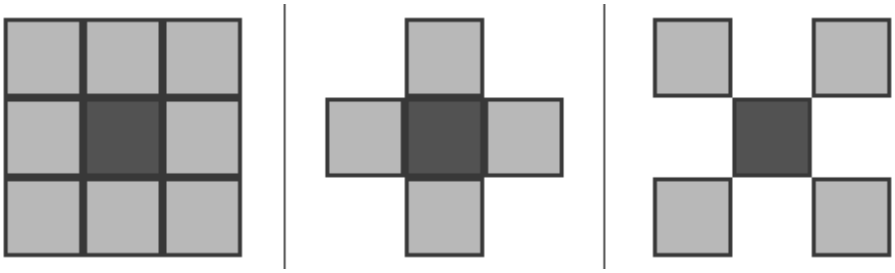
Termālās erozijas algoritms simulē materiāla atdalīšanos un slīdēšanu lejā pa nogāzēm, sakrājoties apakšā. Erozijas process notiek sekojoši: noteikta daļa materiāla, kas atrodas uz nogāzēm, kuru slīpums pārkāpj noteiktu slieksni (sauktu par talus leņģi jeb *T*) tik ilgi, kamēr nogāze sasniedz leņķi *T*. Algoritma kods redzams 2.9.2.1 attēlā, kurā *hi* ir virsotnes augstums, *di* ir augstuma atšķirība starp šo tā zemāko kaimiņu, *dmax* ir starpība starp šo un viszemāko kaimiņu virsotni, *dtotal* ir virsotņu skaits, kuru slīpums pārsniedz *T*, bet *c* ir parametrs, kas norāda, cik daudz materiāla tiek pārvietots lejup pa nogāzi katrā iterācijā, kas parasti ir 0,5 [70].

```
hi = hi + c * (dmax    T) * di / dtotal
```

2.9.2.1. att. **Termālās erozijas formula**

Parasti termālās erozijas algoritms izmanto Mūra kaimiņus (*Moore neighborhood*),

taču, lai to paātrinātu, bieži vien visu blakus esošo kaimiņu virsotņu apskatīšanas vietā tiek apskatītas tikai 4 virsotnes. Fon Noimana kaimiņu (*Von Neumann neighborhood*) virsotnes ir tādas virsotnes, kas atrodas ne tekošās virsotnes augšup, lejup, pa labi vai pa kreisi (skat. 2.9.2.2. att.). Iespējams izmantot arī pagriestās Fon Noimana kaimiņu (Rotated *Von Neumann neighborhood*) virsotnes, kas atrodas labajā augšējā, labajā apakšējā, kreisajā augšējā vai kreisajā apakšējā tekošās virsotnes stūrī (skat. 2.9.2.2. att.) [17].



2.9.2.2. att. **Mūra, Fon Noimana un pagriestā Fon Noimana kaimiņi**

2.9.3. Inversā termālā erozija

Galvenā problēma ar erozijas algoritmu ir tā, ka tā nespēj sasniegt augstu erozijas vērtība, taču datorspēlēs un vizuālos modeļos parasti ir vajadzīgs algoritms ar augstu erozijas vērtība. Inversā erozijas algoritma būtībā ir pārvietot materiālu uz leju pa nogāzi tad, ja nogāzes leņķis ir mazāks par *T*, nevis lielāks, kā tas ir termālās erozijas algoritmam.

Šī algoritma pielietošanas rezultātā zemes reljefs iegūst augstu erozijas vērtību 2 iemeslu dēļ:

1. Gandrīz plakanas virsmas mēdz kļūst pilnīgi plakanas. Šī izlīdzināšana samazina nogāžu skaitu, līdz ar to palielinās erozijas vērtība.
2. Lai arī kopējais reljefa augstums nemainās, taču izlīdzināšana padara kalnu aizņemto laukumu daudz mazāku, līdz ar to kalni kļūst stāvāki. Tā kā kalni kļūst stāvāki, palielinās standarta novirze, kas savukārt palielina erozijas vērtību [70].

2.9.4. Hidrauliskā erozija

Hidrauliskā erozija simulē kalnu šķīdināšanu un pārvietošanu, ko izraisa ūdens.

Hidrauliskās erozijas algoritms sastāv no sekojošiem soļiem:

1. Ūdens parādīšanās.
2. Zemes virsmas šķīdināšana.
3. Ūdens un materiāla pārvietošana.
4. Ūdens iztvaikošana un materiāla nogulsnēšanās.

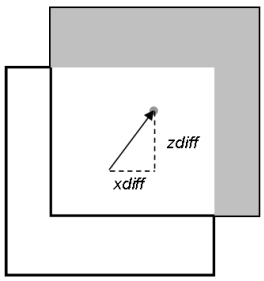
Hidrauliskā erozija paralēli augstumu kartei atmiņā glabā arī ūdens karti un nogulšņu karti, kas glabā informāciju par ūdens un materiālu pārvietošanos [56].

2.9.5. Lietojums risinājumā

Hidrauliskās erozijas algoritms patērē daudz procesora laika, jo tiek veiktas vairākas darbības katram augstumu kartes punktam. Uzlabotais termālās erozijas algoritms patērē tikpat daudz laika, cik termālais erozijas algoritms, taču tas padara kalnus stāvākus un līdzenumus plakanākus, kas noder reljefu ģenerēšanai priekš spēlēm un vizuāliem modeļiem, kam vajadzīgi kalnaini reljefi. Risinājumā tiks izmantots uzlabotais termālās erozijas algoritms.

2.10. Toroidālie masīvi

Toroidāls masīvs ir tāds masīvs, kas izmanto apkārtejošu (*wraparound*) adresāciju. Apkārtejoša adresācija ir vajadzīga, lai masīvu varētu atjaunot inkrementāli. Attēlā 2.10.1. redzams, ja toroidāls masīvs tiek pārvietots par *xdiff* un *zdiff* vienībām, tad veidojas pelēkā L veida daļa, kura tiks atjaunota [61].



2.10.1. att. **Masīva novirze**

Toroidāliem masīviem raksturīgas divas īpašības:

1. Elements *i* vienmēr tiks kartēts vienā un tajā pašā pozīcijā, kuru var iegūt ar funkciju *f(i)*;
2. Katram elementam toroidālā masīvā atbilst tieši viena koordināta, un starp *n* kaimiņiem nav neviena starppunkta.

Pirmā no šīm īpašībām ļauj toroidālos masīvus atjaunot inkrementāli, jo, ja kāds punkts ir masīvā, pie masīva pārbīdes šis punkts nav jāpārbīda, tikai jāuzstāda kopējā masīva nobīde. Otrā īpašība nodrošina to, ka neviens no masīva elementiem nekad netiks pārrakstīts, jo masīva izmērs ir minimāls [61].

Lai atmiņā glabātu mainīga izmēra toroidālo masīvu, atmiņā jāglabā visi tā elementi, kā arī visu stūru koordinātas, piemēram, 2 dimensiju toroidālajiem masīviem atmiņā jāglabā visu 4 stūru koordinātas. Attēlā 2.10.2. redzams pseidokods vērtības iegūšanai no 2D toroidāla masīva, *leftTop* ir vektors, kas satur augšējā kreisā stūra koordinātas, bet *rightBottom* ir vektors, kas satur labā apakšējā stūra koordinātas, *values* ir 2D masīvs, kas satur visas masīva vērtības.

```
function getValue(x, y)
     posX = (x + leftTop.x) % (leftTop.x – rightBottom.x)
     posY = (y + leftTop.y) % (leftTop.y – rightBottom.y)
     return values[posX][posY]
end
```

2.10.2. att. **Vērtības iegūšana no 2D toroidālā masīva**

2.10.1. Lietojums risinājumā

Toroidālie masīvi tiks izmantoti risinājumā augstumu karšu glabāšanai. Tā kā pārvietojoties kamerai, augstumu karte visu laiku ir jāpārbīda, divu koordināšu nomainīšana ir daudz ātrāka kā visa bufera pārbīdīšana atmiņā.

2.11. Kopsavilkums

Šajā nodaļā tika aplūkoti visi algoritmi un metodes, kas tiks izmantoti risinājuma izstrādē, kā arī tika aprakstīts to pielietojums.

Lai reljefs pie viena un tās pašas sēklas vienmēr izskatītos vienādi, pseidonejaušo skaitļu sēklai kalpos Vanga jaucējfunkcija. Vanga jaucējfunkcijai kā parametri tiks padoti tekošās augstumu kartes koordinātes.

Perlina trokšņa algoritms ir salīdzinoši ātrs un ģenerē pietiekami kvalitatīvu rezultātu, lai to varētu izmantot reljefa pamata ģenerēšanai. slīpumu troksnis netika izvēlēts risinājuma izstrādei, jo tā uzģenerētais rezultāts ir pārāk nekvalitatīvs, taču vilnīša algoritms ir nedaudz lēnāks par Perlina algoritmu un tā priekšrocības esošanam risinājumam ir nenozīmīgas.

Atsevišķu reljefa reģionu pacelšanai tiks izmantota Voronoja diagramma. Tuvāko kaimiņu meklēšanai tika izvēlēts KD koka algoritms.

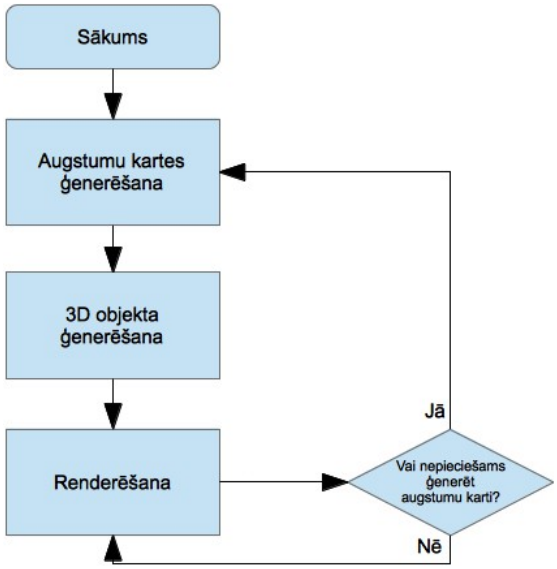
Lai salabotu problēmas, ko rada Voronoja diagramma, un uzlabotu reljefa erozijas vērtību, tiks izmantota perturbācija. Tā kā perturbācija, kas izmantoja Perlina troksni pasliktināja erozijas vērtību, bet, izmantojot slīpumu troksni, tā uzlabojās, tad perturbācija tiks ģenerēta izmantojot slīpumu troksni.

Tā kā kombinējot visus algoritmus un metodes tiek uzģenerēts rezultāts, kas neizskatās reālistiski, tiks izmantots erozijas algoritms. Erozijas algoritms gan nolīdzina asās virsotnes, gan arī uzlabo reljefa erozijas vērtību. Kā vispiemērotākais erozijas algoritmu risinājuma izstrādei tika izvēlēts uzlabotais termālās erozijas algoritms, jo tas, saglabājot uzģenerētos kalnus, līdzenumus padara līdzenākus.

3. PRAKTISKĀ IMPLEMENTĀCIJA

3.1. Risinājuma arhitektūra

Darbā tika izstrādāta sistēma, kas paredzēta bezgalīgu reljefu ģenerēšanai un tā 3D attēlošanai, kuru iespējams aplūkot izmantojot pirmās personas kameru. Risinājums pašā izpildes sākumā ģenerē augstumu karti un tai atbilstošu 3D objektu. Uzģenerētais 3D objekts izmantojot Irrlicht 3D dzini katru kadru tiek renderēts uz ekrāna. Visi risinājuma soļi redzami 3.1.1. attēlā.



3.1.1. att. **Risinājuma arhitektūra**

3.2. Augstumu kartes ģenerēšana

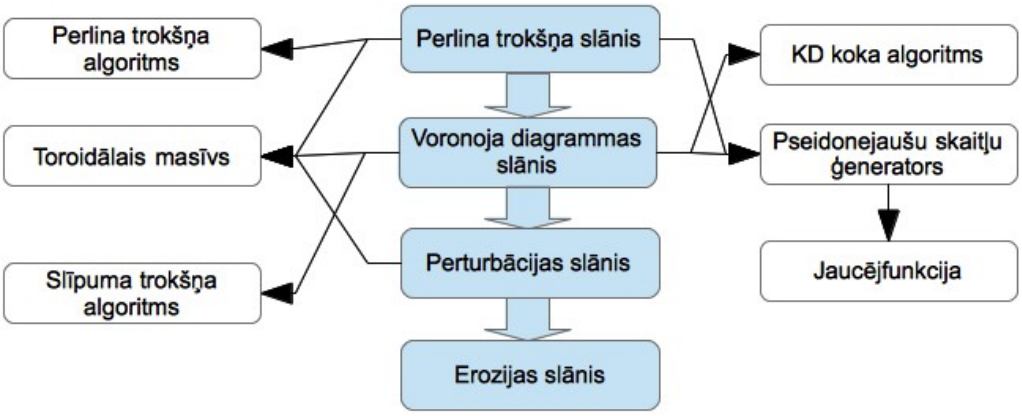
Katra virsotne reljefā tiek ģenerēta tikai tad, kad tā atrodas noteiktā attālumā no kameras. Katras virsotnes augstums ir atkarīgs no tās *X* un *Z* koordinātām, tādā veidā, izmantojot vienu un to pašu sēklu, tiek iegūts viens un tas pats reljefs. Attiecīgi, visas virsotnes, kas vairs nav kameras redzes attālumā tiek dzēstas. Tā kā atmiņā tiek glabāts konstants skaits virsotņu, tad algoritms darbības laikā izmantot konstantu atmiņas daudzumu,

lai arī uzģenerētajam reljefam nav robežu. Ja pieņemtu, ka atstarpe starp 2 virsotnēm ir 1cm, tad ar šo algoritmu 32 bitu sistēmā varētu izveidot vairāk kā 4 miljonu km^2 plašu reljefu. Viss reljefa ģenerēšanas algoritms tika veidots tā, lai tas darbotos ar jebkāda izmēra skaitļiem, teorētiski padarot to par bezgalīgu. Ja kamera sasniedz koordinātas, kas ir izmantojamā skaitļa robežas, tad koordinātu tiek apmestas (*wrap*) un sāktas no sākuma, tādā veidā reljefs visu laiku atkārtojas, bet kamerai tas izskatās bezgalīgs.

Lai būtu iespējams uzģenerēt vairāk kā vienu reljefu, tā ģenerēšanas procesā tiek izmantota sēkla, kura tiek padota algoritmam tā inicializācijas laikā. Ja kā sēkla tiek izmantota 32 bitu vērtība, tad iespējams iegūt 4,2 miljardus dažādu reljefu, jo sēklas vērtībai tiek ģenerēts unikāls reljefs.

Lai taupītu pieejamo datora atmiņu, algoritms izmanto datu slinko rēķināšanu (*lazy evaluation*) t.i. tiek aprēķināta augstumu karte tikai kamerai redzamā daļa. Algoritms sadala reljefu *n* x *n* blokos, savukārt katrs no blokiem sastāv no *m* x *m* virsotnēm (*vertex*). Kamera vienmēr redzami n-2 x n-2 bloki, pārējie bloki redzami daļēji. Kad kamera pārvietojoties par *m* šūnām, tiek ģenerētas jaunās virsotnes.

Augstumu kartes ģenerators ģenerē 4 slāņus, kas tiek kombinēti noteiktā secībā. Katrs no slāņiem izmantot vienu vai vairākus algoritmus un tehnoloģijas tā ģenerēšanai, kā arī glabāšanai atmiņā. Attēlā 3.2.1. redzami augstumu kartes ģenerēšanas soļi un katra soļa izmantotie algoritmi.



3.2.1. att. **Augstumu kartes ģenerēšana**

Augstumu kartes ģenerēšana tiek ietekmēta ar dažādu parametru palīdzību. Katrs no

parametriem ietekmē kādu no algoritma darbībām. Tabulā NN redzami visi parametri, kas tiks padoti risinājumam.

3.2.1. tabula

Lietotie parametri

Mainīgā nosaukums	Apraksts	Pielietojums
c_1	Koeficients, kas tiek reizināts ar attālumu līdz tuvākajam kaimiņam	Voronoja diagrammā
c_2	Koeficients, kas tiek reizināts ar attālumu līdz otram tuvākajam kaimiņam	Voronoja diagrammā
P	Perturbācijas algoritma ietekme	Perturbācijas un Voronoja diagrammas kombinācija
d	Maksimālā distance, par kuru punkts var tikt pārvietots	Perturbācijā
c	Vērtība, par kuru tiek samazināts virsotnes augstums, ja tā ir par augstums	Erozijas algoritmā
T	Talus leņķis, kuru pārsniedzot virsotnes augstums tiek samazināts	Erozijas algoritmā
i	Erozijas iterāciju skaits	Erozijas algoritmā

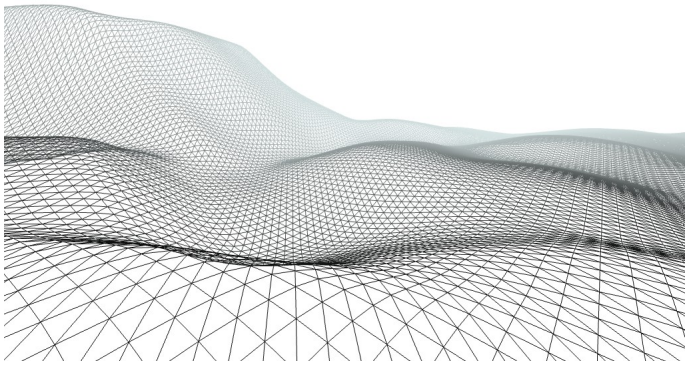
3.3. Datu attēlošana

Lai aprakstītu zemes reljefu, algoritms kā augstumu karti izmanto divu dimensiju masīvu, kas sastāv no 32 bitu peldošā punkta tipa mainīgajiem, kuru vērtība var būt no 0 līdz 1. Augstumu kartes malas tiek apzīmētas ar *sizeX* un *sizeZ*.

Augstuma karte tiek saukta par *data*, bet tās elementi ir *data[x][z]*, kur *x* un *z* ir augstuma kartes koordinātas, kuras var būt attiecīgi no 0 līdz *sizeX* un no 0 līdz *sizeZ*. Ja *x* vai *z* koordinātas atrodas ārpus atļautās robežas, tad tās tiek samazinātas vai palielinātas tā, lai tās atrastos šajās robežās.

3.4. 3D attēlošana

Reljefa grafiskai attēlošanai tiek izmantots Irrlicht 3D dzinis. Lai vizuāli attēlotu uzģenerēto augstuma karti, tiek izveidots indeksu un virsotņu buferi, kuros tiek glabātas 3D režģis (3.4.1. att.), kas tiek deformēts atkarībā no uzģenerētās augstumu kartes. Tā kā augstumu karte tiek regulāri mainīta, tas pats jādara arī ar risinājuma uzģenerēto virsotņu buferi. Virsotņu buferi no grafiskā adaptera atmiņas nolasīt nav iespējams [38], tāpēc tā kopija tiek glabāta arī datora galvenajā atmiņā. Kad vajadzīgs, režģa virsotņu buferis tiek atjaunots ar jaunajām vērtībām un pārkopēts uz grafiskās aparatūras atmiņu.



3.4.1. att. **Risinājuma uzģenerētais režģis**

3.4.2. att. **Augstumu kartes 3D attēlojums**

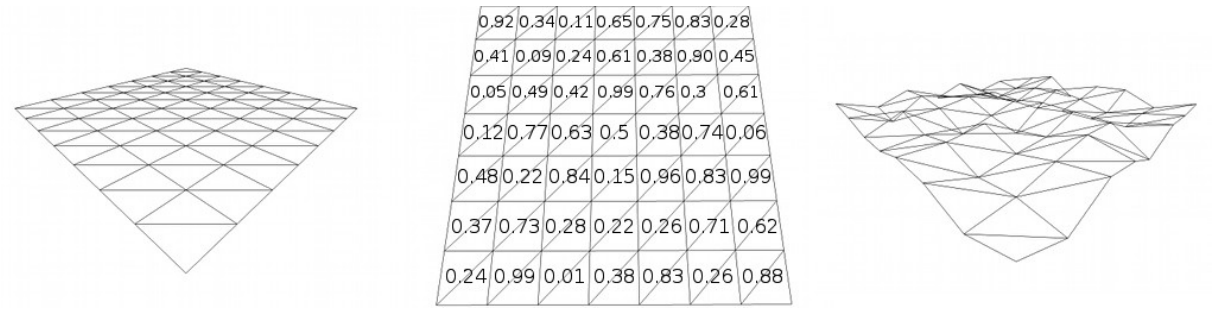
Lai uzģenerētā risinājuma rezultātu varētu pēc iespējas labāk apskatīt, tam tiek uzklātas tekstūras. Risinājuma uzģenerētā reljefa 3D projekcija redzama attēlā 3.4.2.

3.5. Testēšanas metode

Visi testi tiek izpildīti 10 reizes, rezultāts tiek aprēķināts izrēķinot vidējo vērtību no šīm tām. Lai testos nerastos kļūmes, visi testi tiek veikti vienās un tajās pašās *X* un *Z* koordinātās, pie tam, koordinātas tiek izvēlētas tā, ka viss reljefs tiek pārģenerēts no jauna. Testos tiek mērīts tikai augstumu kartes ģenerēšanas laiks, jo tiek pieņemts, ka 3D reljefa ģenerēšana vienmēr aizņem konstantu laiku. Testi tika veikti uz OSX datora ar 8GB RAM un 2.8 GHz Intel Core i7 procesoru.

3.6. Pseidonejaušu skaitļu ģenerēšana

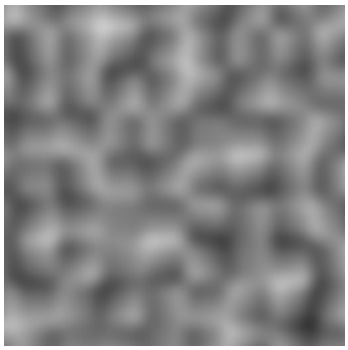
Katra augstumu kartes punkta ģenerēšanā tiek iesaistīti vairāki pseidonejauši skaitļi. Lai vienā un tajā pašā lokācijā vienmēr būtu viens un tas pats augstums, visi pseidonejaušie skaitļi šajā punktā tiek ģenerēti ņemot vērā tikai punkta *x* un *y* koordinātas, kā arī kopējo kartes sēklu. Visi šie skaitļi tiek sajaukti izmantojot FNV jaucējfunkciju. FNV jaucējfunkcija tiek izmantota tādēļ, ka tā ir samērā ātra un labi izkliedē vērtības pa visu skaitļu kopu. Lai iegūtu pseidonejaušu skaitli, kas atbilst konkrētajām augstumu kartes koordinātām, FNV jaucējfunkcijas rezultāts tiek padots pseidonejaušu skaitļu ģeneratoram. 32 bitu sistēmā pseidonejaušo skaitļu ģenerators izvada 32 bitu skaitli, taču algoritmam nepieciešami skaitļi no 0 līdz 1, tāpēc pseidonejaušo skaitļi tiek izdalīti ar lielāko 32 bitu skaitli. Attēlā 3.6.1. redzams, kā, piešķirot pseidonejaušu skaitli katrai režģa koordinātai, tiek uzģenerēts reljefs.



3.6.1. att. **Pseidonejauši skaitļi atkarībā no režģa koordinātām**

3.7. Perlina trokšņa pielietojums

Augstumu kartes pamatā tiek izmantots Perlina troksnis, jo tas ir pietiekami ātrs un ģenerē kvalitatīvu troksni. Perlina troksnis tiek izmantots, lai piedotu reljefam izliekumus un bedres. Perlina trokšņa augstumu karte redzama attēlā 3.7.1. att.



3.7.1. att. **Perlina trokšņa augstumu karte**

3.7.1. Pilnveidošana

Lai netiktu rēķināta Perlina trokšņa vērtība katram punktam katru reizi, kad tiek pārģenerēta karte, šis slānis atmiņā tiek glabāts kā atsevišķs slānis. Kad tiek ģenerēts jaunais reljefs, tiek aprēķināta nobīde pret iepriekš ģenerēto reljefu, un Perlina troksnis tiek rēķināts tikai tiem punktiem, kas atrodas nobīdītajā zonā.

Oriģinālā Perlina trokšņa vietā tiek izmantots uzlabotais Perlina troksnis. Reljefa ģenerēšanas algoritms izmantojot oriģinālo Perlina troksņa algoritmu aizņem 0,102 sekundes, taču izmantojot uzlaboto Perlina trokšņa algoritmu, tas aizņem 0,091 sekundi, kas pātrina algoritma izpildi par 0,01 sekundi. Tomēr izmantojot uzlaboto Perlina trokšņa algoritmu, reljefa erozijas vērtība samazinās par 0,179, no 1,505 uz 1,133. Tas skaidrojams ar to, uzlabotais Perlina troksnis vienmēr izmanto vienus un tos pašus slīpuma vektorus, tas ģenerē troksni ar lielām nogāzēm.

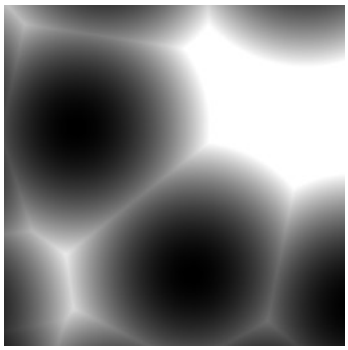
3.8. Voronoja diagrammas pielietojums

Lai arī Perlina troksnis bieži tiek izmantots reljefu ģenerēšanai, tomēr reljefs, kas ģenerēts ar to ir ļoti viendabīgs. Lai padarītu reljefu neviendabīgāku, tiek izmantota Voronoja

diagramma, tā piedod reljefam kalnus, ielejas un līdzenumus.

Sākotnēji tiek uzģenerēts pseidonejaušs skaits ar kontrolpunktiem, pie tam katram uzģenerētajam kontrolpunktam tiek uzģenerēts pseidonejaušs augstums. Kontrolpunkti tiek ģenerēti iterējot cauri visiem punktiem, kur iespējams varētu atrasties kontrolpunkts un ar noteiktu varbūtību tiek noteikts, vai attiecīgajā punktā atrodas kontrolpunkts vai arī nē. Vispiemērotākais kontrolpunktu daudzums tika sasniegts ar varbūtību 0,01%, jo pie šādas varbūtības augstumu kartē vienlaicīgi atrodas ap 10 kontrolpunktiem.

Visiem uzģenerētajiem kontrolpunktiem tiek uzģenerēts KD koks, lai pēc iespējas ātrāk varētu atrast tuvākos kontrolpunktus. Pēc punktu ģenerēšanas, katram augstumu kartes punktam tiek uzģenerēta vērtība, kas atkarīga no attāluma līdz tuvākajiem kontrolpunktam. Katra punkta vērtība tiek aprēķināta pēc formulas: $h = c_1d_1 + c_2d_2 + ... + c_nd_n$, kurā d_n ir attālums līdz n-tajam tuvākajam punktam, bet c_n ir brīvi izvēlēts koeficients, bet h ir rezultāta augstums. Attēlā 3.8.1. redzama Voronoja diagramma, kurā parametr $c_1 = 1$.



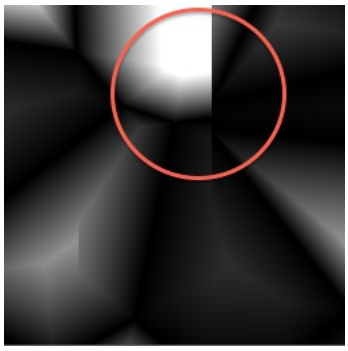
3.8.1. att. **Voronoja diagramma ar koeficientu $c_1 = 1$**

Pēc Voronoja diagrammas uzģenerēšanas, tā tiek kombinēta kopā ar Perlina troksni, lai iegūtu kalnus, ielejas un līdzenumus (skat. 3.12.2.1. att.). Kombinēšana tiek veikta pēc formulas $h = p * h_v + (1 - p) * h_p$, kurā h_v ir Voronoja augstums, h_p ir Perlina troksnis attiecīgajā punktā, bet p ir Perlina trokšņa intensitāte attiecīgajā punktā, kura vērtība var būt no 0 līdz 1.

3.8.1. Pilnveidošana

Lai ierobežotu distanci kādā nepieciešams ģenerēt Voronoja diagrammas

kontrolpunktus, tiek noteikta maksimālā distance, kurā tuvākais kaimiņš skaitās atrasts. Ja kāds no tuvākajiem kaimiņiem atrodas tālāk par šo robežu, tas netiek ņemts vērā. Tādā veidā iespējams ģenerēt kontrolpunktus, kas atrodas noteiktā attālumā no reljefa malas, pārējos punktus nav vajadzības ģenerēt. Šis paņēmiens dažreiz rada lielus lēcienus (skat. 3.8.1.1. att.) Voronoja diagramma, taču tie tiek nolīdzināti ar perturbāciju un erozijas algoritmu.



3.8.1.1. att. **Voronoja diagrammas problēmas**

Tā kā kvadrātsaknes rēķināšanas operācija patērē daudz procesora laika, tad distances rēķināšanai tiek izmantota kvadrātiskā distance. Kvadrātiskā distance sniedz praktiski tādu pašu rezultātu kā Eiklīda distance.

Līdzīgi kā Perlina trokšņu karte, arī Voronoja diagramma tiek glabāta atmiņā kā atsevišķs slānis, kuram klāt tiek ģenerēti tikai vajadzīgi punkti, tādējādi iespējams ietaupīt procesora resursu pārrēķinot tikai to slāņa daļu, kas ir nākusi klāt.

3.9. Virsmas perturbācija

Lai atbrīvotos no dažādām vizuālām problēmām augstumu kartē, virsmai tiek pielietota perturbācija (skat att. 3.12.3.1. att.). Izmantojot perturbāciju, reljefs kļūst reālistiskāks un tam paaugstinās erozijas vērtība. Perturbācija ļauj arī atbrīvoties no Voronoja diagrammas radītajām vizuālajām problēmām, izpludinot tās radītās taisnās līnijas, kā arī pārāk lielās distances dēļ radītās stāvās sienas (skat. 3.8.1.1. att.).

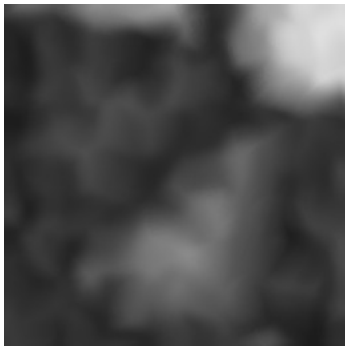
Perturbācijai ir tikai viens konfigurējam parametrs ir perturbācijas distance *d*. Perturbācijas distance norāda maksimālo vērtību pikseļos, par kuru kāds pikselis var tikt nobīdīts augstumu kartē.

3.9.1. Pilnveidošana

Tā kā zināma distance, kurā perturbācija spēj ietekmēt kaimiņu punktus, tad algoritms tiek pārveidots tā, ka tas ģenerē tikai šos punktus. Līdzīgi kā Voronoja un Perlina trokšņu slāņi, arī Perturbācijas slānis tiek glabāts atsevišķā buferī atmiņā, un tam tiek pārģenerēti tikai vajadzīgie punkti.

3.10. Virsmas deformēšana ar erozijas algoritmu

Lai nolīdzinātu reljefa asās šķautnes un palielinātu tā erozijas vērtību, reljefs tiek erodēts ar termālās erozijas algoritma palīdzību. Tā kā spēlēs un vizuālos modeļos ir svarīgi, ka reljefs ir ar kalniem un ieplakām, tad tā ģenerēšanai tiks izmantots inversās erozijas algoritms. Galvenie erozijas algoritma parametri ir erozijas iterāciju skaits *i*, *talus* leņķis *T* un izmaiņu vērtība *c*. Attēlā 3.10.1. redzams reljefs pēc erodēšanas ar erozijas algoritmu.



3.10.1. att. **Erozijas algoritms**

3.10.1. Pilnveidošana

Lai taupītu resursu, tiek izmantota Fon Noimana kaimiņi Mūra kaimiņu vietā. Tādā veidā tiek veikts divas reizes mazāk kalkulāciju un ietaupīts procesora laiks. Izmantojot Fon Noimana kaimiņus erozijas kalkulācija aizņem 0,003 sekundes jeb par 3.6% mazāk laika.

Lai nebūtu jāveido vairāki buferi, kas glabā augstumu kartes, erozijas algoritms tika pārveidots tā, ka tas modificē augstumu karti uzreiz, nevis kopē starpvērtības uz citu augstumu karti.

Atšķirībā no pārējiem slāņiem, erozijas algoritma rezultātu nav iespējams glabāt

atmiņā un pārģenerēt tikai vajadzīgo daļu, jo erozijas rezultātā, praktiski jebkurš punkts var ietekmēt jebkuru citu punktu uzģenerētajā reljefā.

3.11. Risinājuma uzlabošana

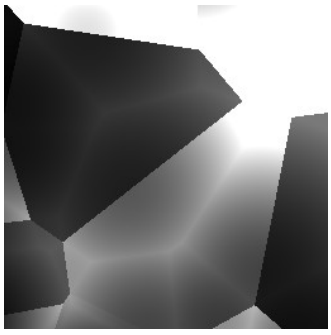
Sākotnēji augstumu glabāšanai algoritms izmantoja dubultās precizitātes jeb 64 bitu peldošo punktu, taču izmantojot 32 bitu peldošo punktu netika novērotas nekādas acīmredzamas izmaiņas reljefā. 32 bitu peldošā punkta izmantošana paātrināja risinājuma izpildi par 0,01 sekundi.

Lai paātrinātu masīvu pārbīdi, tika izmantoti toroidālie masīvi. Pārbīdot kameru, tiek pārbīdīts arī visas augstumu kartes. Izmantojot toroidālos masīvus, katra augstumu kartes punkta kopēšanas uz jauno pozīciju vietā tiek pārbīdīts toroidālā masīva nobīdes vektors, kas izmanto daudz mazāk procesora resursu.

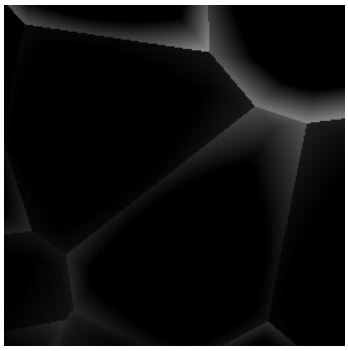
3.12. Parametru noskaņošana

3.12.1. Voronoja trokšņa noskaņošana

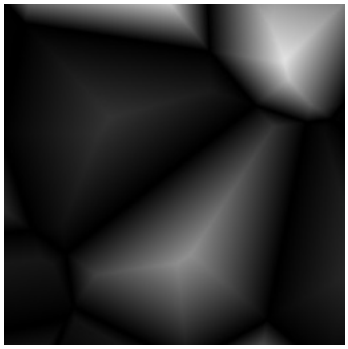
Tā kā ar koeficientu izmantojot tikai vienu koeficientu tiek ģenerēti pārāk nereālistiski rezultāti (skat. 3.8.1. att.), risinājumā tiek izmantoti 2 koeficienti. Abi koeficienti nevar būt negatīvi, jo tad rezultātā būs negatīvs skaitlis, taču augstumu kartei vajadzīgs tikai pozitīvs skaitlis. Ņemot vērā, ka tuvākais kontrolpunkts atrodas daudz tuvāk kā otrs tuvākais, ja c_1 un c_2 ir atšķirīgām zīmēm, tad jāizvelas tādas vērtības, lai to summa nebūtu negatīva, piemēram, $c_1 = 1$, $c_2 = -0,5$ (skat. 3.12.1.2. att.).



3.12.1.1. att. **Voronoja diagramma ar koeficientiem $c_1 = 1$ un $c_2 = 1$**



3.12.1.2. att. **Voronoja diagramma ar koeficientiem $c_1 = 1$ un $c_2 = -0.5$**

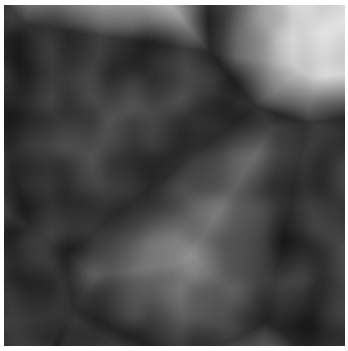


3.12.1.3. att. **Voronoja diagramma ar koeficientiem $c_1 = -1$ un $c_2 = 1$**

Kā redzams attēlos, tad vienīgais rezultāts, kas nesatur izteiktas taisnas līnijas un atgādina kalniem līdzīgus objektus ir pie koeficientiem $c_1 = -1$ *un* $c_2 = 1$, tas arī tiks izmantots pētījumā.

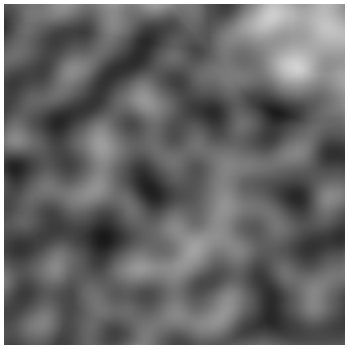
3.12.2. Perlina trokšņa un Voronoja diagrammas noskaņošana

Voronoja diagrammas galvenais pielietojums ir padarīt reljefu kalnainu. Ar parametr *p* palīdzību iespējams pielāgot tā intensitāti, tādā veidā palielinot vai samazinot reljefa kalnu augstumu.

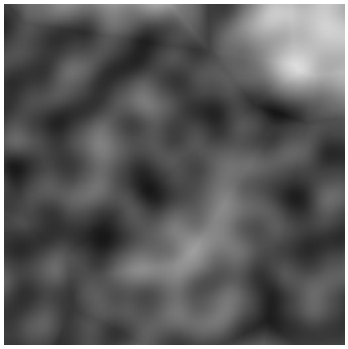


3.12.2.1. att. **Voronoja diagrammas un Perlina troksņa kombinācija pie p = 0,33**

Ja Perlina trokšņa slānis nav pārāk spēcīgs (skat. 3.12.2.1. att.), tad tiek iegūts rezultāts, kurā redzamas Voronoja diagrammas radītās taisnās līnijas. Taču Voronoja diagrammas piešķir reljefam lielus kalnus un ielejas.

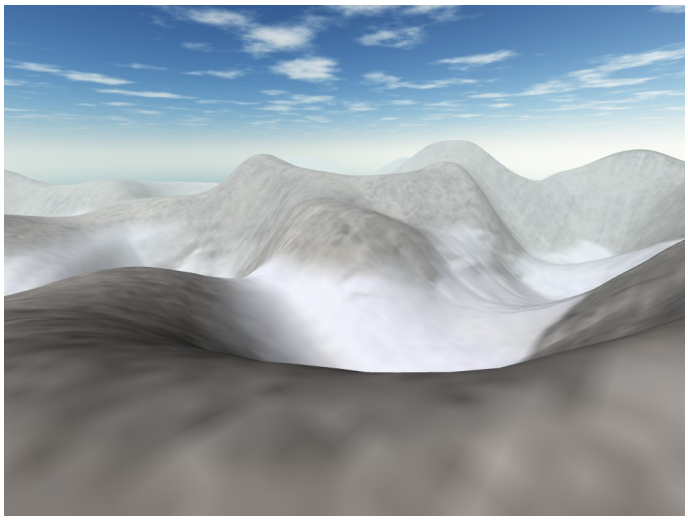


3.12.2.2. att. **Voronoja diagrammas un Perlina troksņa kombinācija pie p = 0,66**



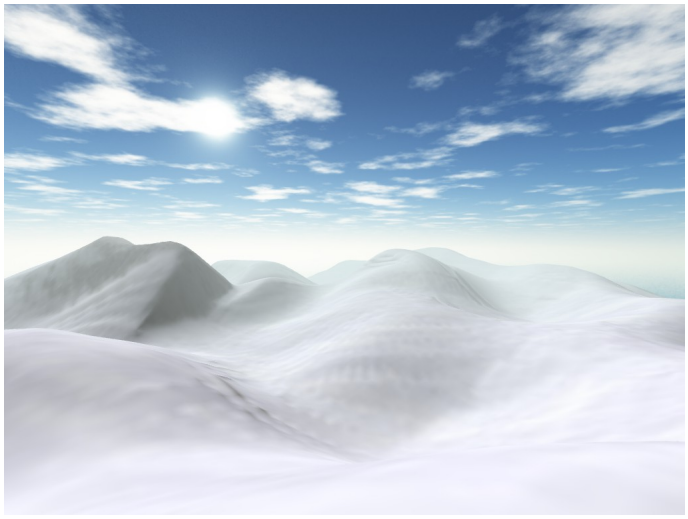
3.12.2.3. att. **Voronoja diagrammas un Perlina troksņa kombinācija pie p = 0.5**

Kā redzams, tad, ja parametrs *p* ir lielāks (3.12.2.2. att.) vai vienāds ar 0,5 (3.12.2.3. att.), tad Perlina troksnis ir pārāk intensīvs un reljefs izskatās trokšņains. Tiek secināts, ka parametram *p* jābūt mazākam par 0,5, piemēram, 0,33.



3.12.2.4. att. **3D reljefs pie parametra p = 0,66**

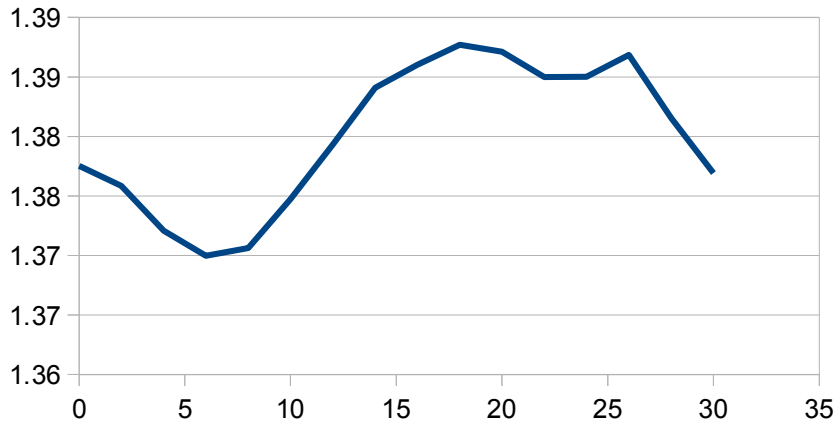
Attēlā 3.12.2.4. var redzēt, ka, ja *p = 0,66*, tad reljefs izskatās pārāk kalnains, taču, ja *p = 0,33* (skat. 3.12.2.5. att.), tad reljefs ir pietiekami līdzens, lai būtu pielietojams, piemēram, datorspēlēs.



3.12.2.5. att. **3D reljefs pie parametra p = 0,33**

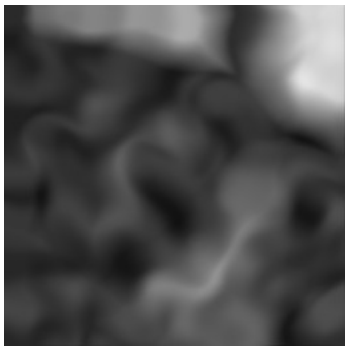
3.12.3. Perturbācijas noskaņošana

Izslēgtas perturbācijas gadījumā reljefa erozijas vērtība ir 1,378. 3.12.3.1. attēlā redzama reljefa erozijas vērtība atkarībā no erozijas distance *d*.

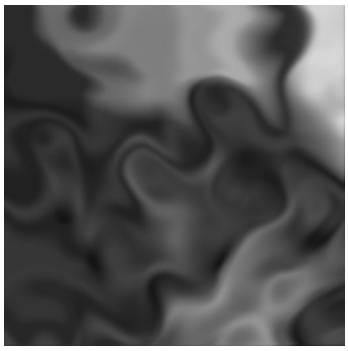


3.12.3.1. att. **Erozijas vērtība atkarībā no perturbācijas distances**

Perturbācija pie mazas *d* vērtības samazina reljefa erozijas vērtība, taču ja *d* ir lielāks par 12, reljefa erozijas vērtība kļūst lielāka. Visaugstāko erozijas vērtību reljefs sasniedz pie *d = 18*. Ja *d* vērtību uzstāda lielāku par 18, reljefs sāk izskatīties nereālistiski. Lielu perturbācijas distanci varētu pielietot, piemēram, fantāziju pasaules ģenerēšanai, taču modeļos, kas simulē dabas skatus, šāda vērtība nav piemērota. Perturbācija ne tikai palielina reljefa erozijas vērtību, bet arī padara to reālistiskāku un novērš vairākas Voronoja diagrammas radītās problēmas, kas redzamas 3.9.1. attēlā.



3.12.3.2. att. **Reljefs ar perturbācijas distanci 18**



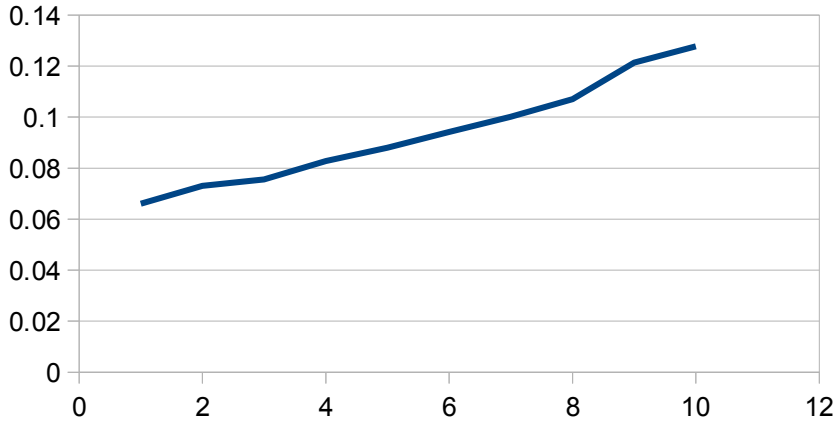
3.12.3.3. att. **Reljefs ar perturbācijas distanci 50**

Visaugstāko erozijas vērtību reljefs sasniedz pie perturbācijas distances 18, kā arī pie šīs vērtības reljefs izskatās reālistiski, un Voronoja diagrammas radītie defekti vairs nav redzami, tāpēc risinājumā tika izmantota šī vērtība.

3.12.4. Erozijas noskaņošana

3.12.4.1. attēlā redzams risinājuma patērētais laiks atkarībā no erozijas iterāciju skaita. Kā redzams, tad izpildes laiks aug viendabīgi, taču neliels lēciens redzams pie 9 iterācijām. Līdz 3 iterācijām izpildes laiks pieaug salīdzinoši lēni.

Atslēdzot eroziju, algoritms izpildes laiks ir 0,06 sekundes.

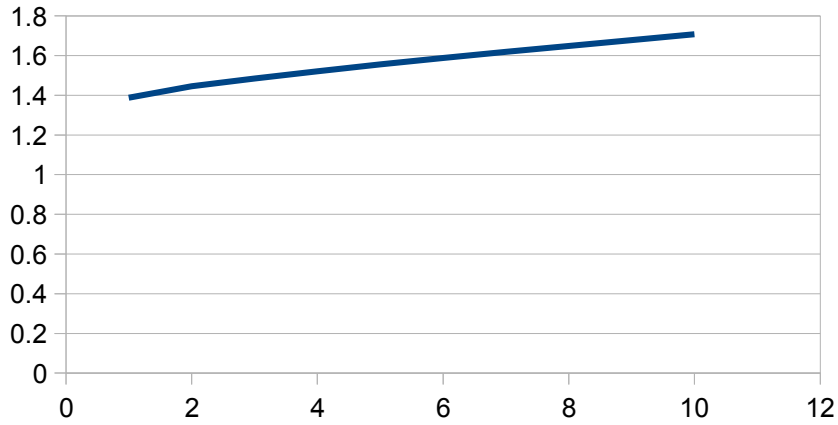


3.12.4.1. att. **Algoritma izpildes laiks atkarībā no erozijas iterāciju skaita**

Attēlā 3.12.4.1. redzams reljefa erozijas vērtība atkarībā no erozijas iterāciju skaita. Kā redzams, tad arī erozijas vērtība aug vienmērīgi, visstraujākais pieaugums novērojams līdz

3 iterācijām. Ja veiktspēja ir svarīga, tad ieteicams izmantot 3 erozijas iterācijas, jo pie 3 iterācijām novērojams viszemākais veiktspējas pieaugums, taču visstraujākais erozijas vērtības pieaugums.

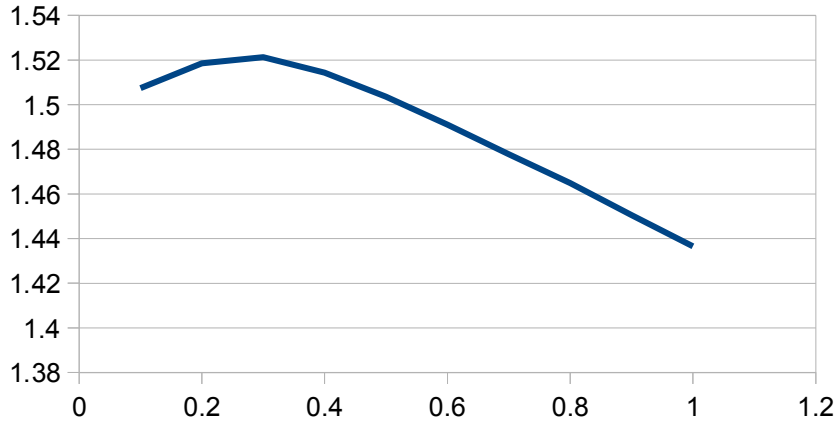
Pavisam izslēdzot eroziju, reljefa erozijas vērtība ir 1,23. Attēlā 3.12.4.2. redzams reljefa erozijas vērtība atkarībā no erozijas iterāciju skaita *i*. Grafikā redzams, ka jau pie vienas erozijas iterācijas, tiek iegūts par 9,4% lielāka erozijas vērtība, jeb 1,38. Erozijas vērtība palielinot iterāciju skaitu aug vienmērīgi.



3.12.4.2. att. **Erozijas vērtība atkarībā no i**

Tā kā risinājumā nepieciešama laba ātrdarbība, tad iterāciju skaits jāizmanto pēc iespējas mazāks. Risinājuma veiktspēja ir līdzīga pie 2 un 3 iterācijām, tāpēc risinājumā tiks izmantotas 3 erozijas algoritma iterācijas.

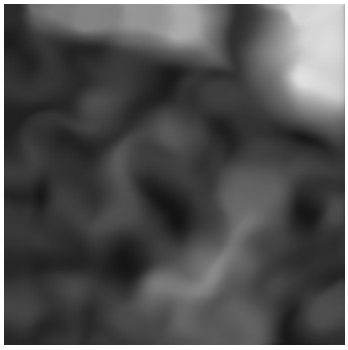
3.12.4.3. attēlā redzamajā grafikā var redzēt, ka visaugstākā erozijas vērtība tiek sasniegta pie *T* vērtības 0,3, tāpēc risinājumā tiek izmantota šī vērtība.



3.12.4.3. att. **Erozijas vērtība atkarībā no T**

Erozijas izmaiņu vērtība *c* risinājumā tika izvēlēta 1,0 (skat. 3.12.4.5. att.), jo pie šādas vērtības rezultāta augstumu karte netika būtiski deformēta.

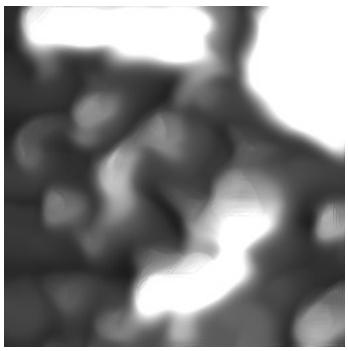
3.12.4.4. att. **Augstumu karte pie c = 0,5**



3.12.4.5. att. **Augstumu karte pie c = 1,0**

Ja *c* vērtība ir mazāka par 1,0 (skat. 3.12.4.4. att.), rezultātā tiek uzģenerēta pārāk

sapludināta augstumu karte. Taču, ja *c* vērtība lielāka par 1,0 (skat. 3.12.4.6. att.), rezultātā tika uzģenerēta augstumu karte, kas tika pārāk strauji erodēta, tāpēc tajā ir daudz punktu, kurā augstuma vērtība ir maksimāla.



3.12.4.6. att. **Augstumu karte pie c = 1,5**

Erozijas algoritmam tika izvēlētas šādas vērtības: *i = 3, T = 0.3, c = 1.0*, jo pie šīm vērtībām risinājums patērē apmierinošu laika apjomu augstumu kartes ģenerēšanai, kā arī reljefa erozijas vērtība ir pieņemama.

3.13. Novērtējums

Risinājumā kopumā tika izmantoti 10 algoritmu un tehnoloģijas augstumu kartes ģenerēšanai reālā laikā, kā arī ar Irrlicht 3D dzinis augstumu kartes renderēšanai uz ekrāna.

Risinājums kā pirmo ar Perlina trokšņa palīdzību uzģenerē pamata slāni. Nākošais slānis tiek ģenerēts no Voronoja diagrammas ar parametriem $c_1 = 1$, $c_2 = -0,5$. Pamata slānis tiek kombinēts kopā ar Voronoja diagrammu, izmantojot parametru $p = 0.3$, jo tika izpētīts, ka šādi koeficienti ģenerē vispiemērotāko rezultātu pielietojumiem datorspēlēs un vizuālos modeļos. Pamata slāņa un Voronoja diagrammas kombinācija tiek perturbēta ar perturbācijas algoritmu, kas izmanto slīpumu troksni, ar parametru $d = 18$. Perturbācijas rezultāts tiek erodēts ar parametriem $i = 3, T = 0.3, c = 1.0$.

Izmantojot minētos parametrus, risinājums augstumu karti ģenerē vidēji 0.072 sekundes, bet deformē iepriekš uzģenerētu režģi 0.017 sekundes. Augstumu karte tiek ģenerēta tikai tad, ka kamera ir sasniegusi punktu, kurā iespējams redzēt vēl neuzģenerētu reljefa reģionu. Kopā ģenerēšanas brīdi risinājums patērē 0.091 sekundes, jeb gandrīz simto

daļu no sekundes. Mūsdienu spēlēs pieņemams kadru skaits sekundē ir 30 [18], izstrādātais algoritms spētu ģenerēt augstumu karti 100 reizes sekundē. Taču spēles parasti patērē daudz laika spēles loģikas aprēķināšanai, un izstrādātais risinājums reljefu ģenerē samērā reti (tikai tad, kad lietotājs ir sasniedzis vel neuzģenerētu reljefa daļu) un tā ģenerēšana aizņem salīdzinoši maz laika.

4. SECINĀJUMI

Maģistra darba mērķis bija izveidot risinājumu, kas ģenerē bezgalīgu zemes reljefu pielietojumiem datorspēlēs un vizuālos modeļos. Darbā tika aplūkoti jau esoši pētījumi par procedurāli ģenerētiem reljefiem, kā arī veikts algoritmu apskats, kas varētu noderēt zemes reljefa ģenerēšanai. Tika izveidots algoritms, kuru iespējams integrēt gan datorspēlēs, gan vizuālos modeļos.

Lai arī procedurāla satura ģenerēšana tiek pētīta samērā ilgi, tomēr tie reti tiek izmantoti dažādos projektos, jo ir sarežģīti izstrādājami, reljefa ģenerēšanas process patērē daudz aparatūras resursu un bieži vien tas nedod reālistisku rezultātu. Izstrādātais risinājums tika veidots tā, lai būtu izmantojams datorspēlēs reljefa ģenerēšanai reālā laikā, un šo uzdevumu arī pilda.

Izstrādātais risinājums reljefu ģenerē procedurāli, un, lai samazinātu ģenerēšanai patērēto laiku, tiek ģenerēta tikai tā reljefa daļa, kas atrodas noteiktā distancē līdz kamerai. Reljefs tiek ģenerēts izmantojot 10 algoritmus un tehnoloģijas, kā arī tiek izmantotas 3D tehnoloģijas tā renderēšanai uz ekrāna. Reljefs tiek ģenerēts pa slāņiem, kā pirmais slānis tiek ģenerēts pamata slānis, kas izmanto Perlina trokšņa algoritmu. Pamata slānis tiek kombinēts kopā ar Voronoja diagrammu, un šīs kombinācijas rezultāts tiek perturbēts ar slīpumu trokšņa palīdzību. Lai rezultāta reljefā nebūtu dažādu vizuālu defektu un asu virsotņu, tas tiek izlīdzināts ar erozijas algoritma palīdzību. Risinājums visus slāņus glabā atmiņa izmantojot toroidālos masīvus un nejaušu skaitļu ģenerēšanai izmanto nejaušu skaitļu ģeneratoru, kas kā sēklu lieto jaucējfunkcijas rezultātu.

Darba mērķis tika sasniegts, risinājums reāla laika bezgalīga reljefa imitācijas ģenerēšanai pielietojumiem datorspēlēs un vizuālos modeļos tika izveidots. Lai sasniegtu izvirzīto mērķi, tika tika izpildīti sekojoši uzdevumi:

1. Tika izpētīti esošie risinājumi, kas paredzēti procedurālā satura ģenerēšanai spēlēs un vizuālos modeļos;
2. Tika izpētīt un atlasīti piemērotākie algoritmi un tehnoloģijas reljefa ģenerēšanai reālā laikā;

3. Tika izveidot risinājums, kas reālā laikā ģenerē reljefu un veido tā 3D projekciju;
4. Tika izanalizēt iegūtā pētījuma rezultāti.

Papildus darba gaitā radās sekojoši secinājumi:

1. Reljefa ģenerēšana spēlēs tiek izmantota reti, jo tā ģenerēšana patērē daudz laika un nepieciešams liels darbs, lai rezultāts izskatītos reālistiski;
2. Lielākā daļa algoritmu, kas paredzēti reljefu ģenerēšanai, izmanto trokšņu algoritmus, jo tie padara reljefu daudzveidīgāku un reālistiskāku;
3. Ģenerējot reljefu pa daļām, iespējams būtiski samazināt tā ģenerēšanas laiku;
4. Lielāko daļu reljefu ģenerēšanai izmantojamos algoritmus un tehnoloģijas iespējams pilnveidot, būtiski samazinot laiku, kas tiek patērēts to izpildei.

5. TURPMĀKIE PĒTĪJUMI

Darbā tika izstrādāts risinājums, kas paredzēts bezgalīgu reljefu ģenerēšanai. Uz mūsdienīgas tehnikas risinājuma patērētais laiks ir pieņemams, taču izmantošanai spēlēs, kurās liela daļa no resursiem tiek atvēlēta spēles loģikas realizēšanai, šis algoritms varētu nederēt. Risinājumu iespējam pilnveidot vēl vairāk, iespējams zaudējot reljefa kvalitāti vai reālistiskumu. Piemēram, viens no veidiem, kā pilnveidot risinājumu, būtu glabājot atmiņā iepriekš aprēķinātu erozijas slāni, un rēķinot eroziju tikai jaunajiem reljefa punktiem.

Lielu daļu no izstrādātās sistēmas iespējams darbināt uz grafiskās aparatūras, kas ir paredzēta matemātisko darbību izpildīšanai. Rēķinot reljefu uz grafiskās aparatūras, paralēli būtu iespējams arī ģenerēt dažādus reljefa grafiskos efektus un tekstūras.

Šobrīd reljefa rezultātu iespējams ietekmēt tikai mainot algoritmam padoto sēklu, perturbācijas, erozijas un Perlina trokšņa parametrus. Taču algoritmu varētu pārveidot tā, lai tam varētu regulēt tādus parametrus, kā kalnu un ieplaku biežumu, vidējo kalnu augstumu, nogāžu stāvumu un dažādus citus parametrus. Tādā veidā risinājums būtu pielāgojams plašākam pielietojumam.

Lai pilnība varētu kontrolēt uzģenerēto reljefu, algoritmu varētu papildināt ar iespēju rediģēt atsevišķas tā vietas. Tādā veidā, ja mākslinieks nolemtu, ka konkrētajā vietā lietojumam vajadzīgs līdzenumu, tad izmantojot specializētus rīkus, būtu iespējams izmainīt algoritma ģenerētās virsotnes noteiktajos punktos.

Lai paplašinātu algoritma pielietojumu spēlēs un dažādos citos dabas skatu ģeneratoros, algoritmu varētu papildināt ar iespējamu dinamiski uz reljefa izklāt dažādus objektus, piemēram, kokus, zāli vai pat veselas pilsētas. Pārbūvējot algoritmu, būtu iespējams procedurāli ģenerēt izklājamos objektus, tādējādi būtiski samazinot laika apjomu, kas māksliniekam būtu jāveltī pasaules izstrādei.

Lai padarītu zemes reljefu vēl reālistiskāku, pirms tā ģenerēšanas varētu simulēt dažādus ģeoloģiskus procesus, piemēram, plātņu kustību. Izanalizējot kā kustējušās plātnes, būtu iespējams ģenerēt reālistiskākus kalnus un ieplakas. Izmantojot ģeoloģisko procesu simulēšanu, būtu iespējams arī noteikt, kurā reljefa daļā ir atsegta klints, bet kurā atrodas,

piemēram, zālājs. Zinot kāda tipa virsma atrodas katrā reljefa punktā, būtu iespējams uz reljefa uzklāt atbilstošu tekstūru, tādā veidā padarot to daudz reālistiskāku un daudzveidīgāku.

6. TERMINU VĀRDNĪCA

Bitmap – bitkarte – 2 dimensiju masīvs, kas paredzēts attēlu glabāšanai

Camera – kamera – vektors 3D pasaulē, kas nosaka kā 3D pasaule tiek attēlota grafiskajā logā

Euclidean distance – Eiklīda distance – distance starp diviem objektiem, ko var izrēķināt ar Pitagora teorēmu

Fragment – fragments – pikselis, kas pieder kādam no trīsstūriem, kas atrodas 3D pasaulē

Frame – kadrs – programmas ģenerēts attēls, kas tiek regulāri atjaunots un rādīts uz ekrāna

Incremental – inkrementāls – tāds, kas pakāpeniski palielinās

Interpolation – interpolācija – starpvērtību noteikšana pēc doto funkcijas vērtību virknes

Iteration – iterācija – atkārtota kādas matemātiskas operācijas izpildīšana

Permutation – permutācija – permutēšana, pārstatīšana, mainīšana vietām

Perturbation – perturbācija – datu deformēšana

Pixel – pikselis – maza daļa no attēla, kas izpaužas kā apgaismots laukums uz ekrāna

Rendering – renderēšana – 3D objektu projekcija 2D grafiskajā logā

Runtime – izpildes laiks – laiks, kurā strādā programma

Screen – ekrāns – datora monitora daļa, kas paredzēta attēla rādīšanai

Terrain – reljefs – zemes virsma

Texture – tekstūra – attēls, kas nosaka objekta virsmas izskatu

Toroid – toroīds – objekts, kura forma atgādina toru

Vertex – virsotne – punkts, kurā sastopas 2 taisnes, piemēram, daudzstūra stūrī

7. LITERATŪRA

1. Terminoloģijas portāls: Letonika [Elektroniskais resurss] / Hash. - Tiešsaistes pakalpojums. - Rīga: 1999. - Pieejas veids: tīmeklis WWW. URL: http://termini.letonika.lv/Term.aspx?search=hash. - Resurss aprakstīts 27.08.2012.
2. 4-byte Integer Hashing: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://burtleburtle.net/bob/hash/integer.html. - Resurss aprakstīts 29.09.2012.
3. 64. What Is Wavelet Noise?: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://www.wisegeek.com/what-is-wavelet-noise.htm. - Resurss aprakstīts 10.112.2012.
4. A Detailed Introduction to K-Nearest Neghbor (KNN) Algorithm: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2010. - Pieejas veids: tīmeklis WWW. URL: http://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/. - Resurss aprakstīts 17.12.2012.
5. A Free OpenGL Programming Book: [Elektroniskais resurss] / What is OpenGL? - Tiešsaistes pakalpojums. – Pieejas veids: tīmeklis WWW. URL: http://openglbook.com/the-book/preface-what-is-opengl/. - Resurss aprakstīts 15.09.2012.
6. A multi-objective approach for Dynamic Airspace Sectorization using agent based and geometric models / Tang J., Alam S., Lokan C., Hussein A. A. - Canberra: University of New South Wales at the Australian Force Academy, 2011. - 33 p.
7. ARB Vertex Program Specification – SGI: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2004. - Pieejas veids: tīmeklis WWW. URL: http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt. - Resurss aprakstīts 30.08.2012.
8. Archer T. Procedurally Generating Terrain / Sioux CitY, Iowa: Morningside College. - 14 p.

9. Cite SeerX: Texture Perturbation Effects, Isidoro J., Riguer G. [Elektroniskais resurss] / Pieejas veids: .pdf: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.192.226. - Resurss aprakstīts 29.12.2012.er

10. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. Thatcher U. / [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2000. - Pieejas veids: tīmeklis WWW. URL: http://www.gamasutra.com/view/feature/131841/continuous_lod_terrain_meshing_.php?page=3. - Resurss aprakstīts 14.12.2012.

11. Department of Computing Science: [Elektroniskais resurss] / Nearest Neighbor Search - Tiešsaistes pakalpojums. - Umea Universitet - Pieejas veids: tīmeklis WWW. URL: http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK4/NODE188.HTM. - Resurss aprakstīts 07.12.2012.

12. Dynamic landscape generation using page management/ Danaher M., - Western Australia: School of Computer and Information Science, Edith Cowan University. - 4

13. Efficient Algorithms with Neural Network Behaviour / Omohundro S. M. - Champaign: University of Illionois atUrbana-Champaign. Department of Computer Science and Center for Complex Systems Research, 1987.- 74 p.

14. First use of Procedual generation in a video game: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://www.guinnessworldrecords.com/records-6000/first-use-of-procedural-generation-in-a-video-game/. - Resurss aprakstīts 17.12.2012.

15. Float4x4: Computer graphics andd stuff [Elektroniskais resurss] / Generating realistic and playable terrain height – maps / Tiešsaistes pakalpojums. - 2010. - Pieejas veids: tīmeklis WWW. URL: http://www.float4x4.net/index.php/2010/06/generating-realistic-and-playable-terrain-height-maps/. - Resurss aprakstīts 15.12.2012.

16. FNV Hash: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Sunnyvale, 2012. - Pieejas veids: tīmeklis WWW. URL: http://isthe.com/chongo/tech/comp/fnv/. - Resurss aprakstīts 28.09.2012.

17. Fractalterraingeneration: Investigation and implementation of various PGC techniques [Elektroniskais resurss] / Erosion Models; Explanatation of three Erosion Models. - Tiešsaistes pakalpojums. - 2010. - Pieejas veids: tīmeklis WWW. URL: http://code.google.com/p/fractalterraingeneration/wiki/Erosion_Models. - Resurss aprakstīts 17.11.2012.

18. Frame Rate (video game concept) [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2009. - Pieejas veids: tīmeklis WWW. URL: http://www.giantbomb.com/frame-rate/92-4761/

19. Fuel sets Guinness record as biggest console game ever: Dobson J. [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2009. - Pieejas veids: tīmeklis WWW. URL: http://www.joystiq.com/2009/05/22/fuel-sets-guinness-record-as-biggest-console-game-ever/. - Resurss aprakstīts 07.01.2013.

20. Fuel:Terrain [Elektroniskais resurss] / Tiešsaistes pakalpojums. 2009. - Pieejas veids: tīmeklis WWW. URL: http://www.shamusyoung.com/twentysidedtale/?p=5267. - Resurss aprakstīts 07.01.2013.

21. Geometry in Action: [Elektroniskais resurss] / Voronoi Diagrams: Applications from Archaology to Zoology - Tiešsaistes pakalpojums. – Regional Geometry Institute, Smith College: 1993. - Pieejas veids: tīmeklis WWW. URL: http://www.ics.uci.edu/~eppstein/gina/scot.drysdale.html. - Resurss aprakstīts 24.11.2012.

22. Good Hash Functions: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2007. - Pieejas veids: tīmeklis WWW. URL: http://blog.clawpaws.net/post/2007/04/22/Good-Hash-Functions. - Resurss aprakstīts 08.112.2012.

23. Hash Collisions (The Poisoned Message Attack): “The Story of Alice an her Boss” [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/. - Resurss aprakstīts 19.12.2012.

24. Improving Noise / Perlin K. - New York: New York University, 2002. - 2 p.

25. Irrlicht Engine - A free open source 3D engine: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://irrlicht.sourceforge.net/. - Resurss aprakstīts 20.10.2012.

26. Knott G. D. Hashing functions // The Computer Journal – Volume 18, Nr 3 (1975), p. 265-278.

27. Mac Developer Library: [Elektroniskais resurss] / About OpenGL for OS X - Tiešsaistes pakalpojums. - Cupertino: Apple Inc., 2012. - Pieejas veids: tīmeklis WWW. URL: https://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_intro/opengl_intro.html. - Resurss aprakstīts 22.12.20

28. Making Noise: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://www.noisemachine.com/talk1/index.html. - Resurss aprakstīts 25.11.2012.

29. MathWold: [Elektroniskais resurss] / Voronoi diagrams. Weisstein E. - Tiešsaistes pakalpojums. – Pieejas veids: tīmeklis WWW. URL: http://mathworld.wolfram.com/VoronoiDiagram.html. - Resurss aprakstīts 24.11.2012.

30. Moore A.W. An introductory tutorial on kd-trees / Efficient Memory-based Learning for Robot Control PhD. Thesis; Technical Report No. 209 – University of Cambridge, 1991. - 19 p.

31. Msdn: Data Type Ranges [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://msdn.microsoft.com/en-us/library/s3f49ktz%28v=vs.80%29.aspx. - Resurss aprakstīts 12.12.20

32. Nearest Neighbor Search: the Old, the New, and the Impossible / Andoni A. - Cambridge: Massachusetts Institute of Technology, 2009. - 178 p.

33. Nquyen H. GPU Gems 3 – Boston: NVIDIA Corporation, 2007. - 1008 p.

34. NYU Media Research Lab: [Elektroniskais resurss] / Brief Biography - Tiešsaistes pakalpojums. - Manhattan: 2012. - Pieejas veids: tīmeklis WWW. URL: http://mrl.nyu.edu/~perlin/doc/bio.html. - Resurss aprakstīts 25.08.2012.

35. OpenGL 3 & DirectX 11: The War Is Over [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2008. - Pieejas veids: tīmeklis WWW. URL: http://www.tomshardware.com/reviews/opengl-directx,2019.html. - Resurss aprakstīts 17.09.2012.

36. OpenGL ES Shading Language: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://linux-sunxi.org/OpenGL_ES_Shading_Language. - Resurss aprakstīts 02.12.2012.

37. OpenGL Programming Guide for Mac: [Elektroniskais resurss] / About OpenGL for OSX. - Tiešsaistes pakalpojums. – Cupertino: Apple Inc., 2012. - Pieejas veids: tīmeklis WWW. URL: https://developer.apple.com/library/mac/#documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_intro/opengl_intro.html. – Resurss aprakstīts 15.09.2012.

38. OpenGL Vertex Buffer Object (VBO): [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://www.songho.ca/opengl/gl_vbo.html. - Resurss aprakstīts 10.01.2012

39. OpenGL vs DirectX: The War Is Far From Over [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://rastergrid.com/blog/2011/10/opengl-vs-directx-the-war-is-far-from-over/. - Resurss aprakstīts 17.09.2012.

40. OpenGL: Core Language (GLSL) [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://www.opengl.org/wiki/Core_Language_%28GLSL%29. - Resurss aprakstīts 19.09.2012.

41. OpenGL: Fixed Function Pipeline [Elektroniskais resurss] / Tiešsaistes pakalpojums.

- 2012. - Pieejas veids: tīmeklis WWW. URL: http://www.opengl.org/wiki/Fixed_Function_Pipeline. - Resurss aprakstīts 21.09.2012.

42. OpenGL: Industry's Foundation for High Performance Graphics [Elektroniskais resurss] / History of OpenGL - Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://www.opengl.org/wiki/History_of_OpenGL. - Resurss aprakstīts 14.09.20
43. Parish Y., Müller P. Procedural Modeling of Cities // Proceedings of The 28th annual conference on Computer graphics and interactive techniques, New York, 2001.- 301-308 p.
44. Pentopia: [Elektroniskais resurss] / What is Procedual generatiion?; Procedual generation - Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://encycl.opentopia.com/term/Procedural_generation. - Resurss aprakstīts 07.12.2012
45. Perlin Noise [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm. - Resurss aprakstīts 26.11.2012.
46. Perlin Noise: Explanation of Perlin Noise [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2010. - Pieejas veids: tīmeklis WWW. URL: http://code.google.com/p/fractalterraingeneration/wiki/Perlin_Noise. - Resurss aprakstīts 05.12.2012.
47. Polack T. Focus On 3D Terrain Programming (Focus on Game Development). - USA: Premier Press, 2002. - 329
48. Procedual Content Generation Wiki: [Elektroniskais resurss] / Minecraft - Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://pcg.wikidot.com/pcg-games:minecraft. - Resurss aprakstīts 06.01.2013.
49. Procedual Noise Categories: Explicit Noises [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2011. - Pieejas veids: tīmeklis WWW. URL: http://physbam.stanford.edu/cs448x/old/Procedural_Noise%282f

%29Categories.html#ExplicitNoises. - Resurss aprakstīts 06.12.2012.

50. Procedural 3D content generation: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Tech. rep., Intel Developer Service., 2000. - Pieejas veids: tīmeklis WWW. URL: http://developer.intel.com. - Resurss aprakstīts 03.11.2012.
51. Procedural Content Generation Wiki: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://pcg.wikidot.com/. - Resurss aprakstīts 03.11.2012.
52. Procedural Content Generation Wiki: [Elektroniskais resurss] / List Of Pcg Games By License. - Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://pcg.wikidot.com/list-of-pcg-games-by-license. - Resurss aprakstīts 03.11.2012.
53. Procedural Content Generation: Thinking with Modules. Lambe I. [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://www.gamasutra.com/view/feature/174311/procedural_content_generation_.php. - Resurss aprakstīts 12.12.2012.
54. Procedural Terrain Generation and Object Propagation / York S. - Philadelphia: Univesity of Pennsylavania, 2007. - 14 p.
55. Real-time Procedural Generation of 'Pseudo Infinite' Cities / Greuter S., Parker J., Stewart N., Leach G. - Melbourne, Victoria, Australia: RMIT University, 2003. - 8 p.
56. Realtime Procedural Terrain Generation: Realtime Synthesis of Eroded Terrain for Use in Computer Games. Olsen J. / University of Sousthern Denmark. - 2004. - 20 p.
57. RGB pixel formats: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2011. - Pieejas veids: tīmeklis WWW. URL: http://www.fourcc.org/rgb.php. - Resurss aprakstīts 05.01.2013.
58. Sentinel, The Download (Puzzle Game): [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://www.old-games.com/download/3779/sentinel-the. - Resurss aprakstīts 20.12.2012.
59. Stochastic Patterns Perturbed Regular Patterns: [Elektroniskais resurss] / Tiešsaistes

pakalpojums. - Pieejas veids: tīmeklis WWW. URL: http://www.mymentalray.com/component/content/article/58-passthrough-bump-map.html?itemid=55&start=9. - Resurss aprakstīts 06.01.2013.

60. Swiftcoding: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2008. - Pieejas veids: tīmeklis WWW. URL: http://swiftcoder.wordpress.com/page/4/. - Resurss aprakstīts 26.08.2012.
61. Terrain Rendering Using Geometry Clipmaps / Bartell N. - New Zealand: University of Canterbury, 2005. -36 p.
62. Texturing and Modeling, Third Edition: A Procedural Approach / D.S. Ebert, F.K. Musgrave, D. Peachey ... [u.c.]. - San Francisco: Elsevier Science (USA), 2003. - 687 p.
63. The FNV Non-Cryptographic Hash Algorithm: [Elektroniskais resurss] / Fowlwr G., Noll L., Vo K., Eastlake D. Why is NFV Non-Cryptographic? - Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://tools.ietf.org/html/draft-eastlake-fnv-03#section-6.1. - Resurss aprakstīts 28.09.2012.
64. The GLIBC random number generator: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - Halifax: Dalhousie University., 2007. - Pieejas veids: tīmeklis WWW. URL: http://www.mathstat.dal.ca/~selinger/random/. - Resurss aprakstīts 24.11.2012.
65. The Irrlicht Engine: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - O'Reilly Media, Inc., 2005. - Pieejas veids: tīmeklis WWW. URL: http://linuxdevcenter.com/pub/a/linux/2005/06/02/irrlicht.html. - Resurss aprakstīts 20.10.2012.
66. The OpenGL® Shading Language / Kessenich J., Baldwin D., Rost R. - 3Dlabs, Inc. Ltd., 2004. - 112 p.
67. Unify Community: Terrain tutorial [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2012. - Pieejas veids: tīmeklis WWW. URL: http://wiki.unity3d.com/index.php?title=Terrain_tutorial. - Resurss aprakstīts 11.12.2012.

68. Wavelet Noise / Cook R. L., DeRose T. - Association for Computing Machinery, 2005. - 9 p.

69. Why you should use OpenGL and not DirectX: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2010. - Pieejas veids: tīmeklis WWW. URL: http://blog.wolfire.com/2010/01/Why-you-should-use-OpenGL-and-not-DirectX. - Resurss aprakstīts 17.09.2012.

70. Zhu J. Real Time Procedural Terrain Generation – 5 p.

71. Zucker M. The Perlin noise math FAQ: [Elektroniskais resurss] / Tiešsaistes pakalpojums. - 2001. - Pieejas veids: tīmeklis WWW. URL: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. - Resurss aprakstīts 08.12.2012.