Jämförelse mellan Perlinbrus och Diamond-square algoritmen för procedurell terränggenerering

Tony Lindberg
Åbo Akademi
Fakulteten för naturvetenskaper och teknik
Kandidatavhandling i Datavetenskap
Handledare: Mats Aspnäs
Våren 2021

Referat

Kommer senare...

Innehållsförteckning

1. Inledning 1
1.1. Arbetets avgränsning 1
2. Procedurell terränggenerering 2
2.1. Brus 3
2.1.1. Perlinbrus (Perlin Noise) 4
2.2. Fraktaler 6
2.2.1. Diamond-square algoritmen 8
3. Jämförelse av Perlinbrus och Diamond-square 10
3.1. Terränggenerering med Perlinbrus 10
3.1.1. Algoritmen 11
3.2. Terränggenerering med Diamond-square 14
3.2.1. Algoritmen 15
3.3. Jämförelse av resultat 16
3.3.1. Genererad Terräng 16
3.3.1. Prestanda och skalbarhet 17
4. Sammanfattning 19
Källförteckning 21
Figurförteckning 23
Bilaga A: Perlinbrus programkod
Bilaga B: Perlinbrus terrängsampel
Bilaga C: Diamond-square algoritmens programkod
Bilaga D: Diamond-square terrängsampel

Förkortningar

PCG	Procedurell innehållsgenerering *(Procedural Content Generation)*
PTG	Procedurell terränggenerering *(Procedural Terrain Generation)*
PRN	Pseudo-slumptal *(Pseudo-random Number)*
vnoise	Värdebrus *(Value Noise)*
gnoise	Lutningsbrus *(Gradient Noise)*

1. Inledning

Procedurell innehållsgenerering (*PCG, eng: procedural content generation*) innebär att man skapar innehåll algoritmiskt, istället för manuellt. Innehåll i detta sammanhang kan ta många olika former, men är oftast förknippat med multimedia, som datorspel, film och tv. Bland dessa handlar det vanligen om 3D-modeller och texturer, ofta i form av terräng eller landskap.

Trots att dessa algoritmiska metoder för att skapa innehåll ursprungligen uppstått p.g.a. hårdvarurestriktioner i hemdatorer under tidigt 80-tal, har det under de senaste 10–15 åren uppstått något av en renässans vad beträffar PCG [1]. Detta beror dels på att PCG tillåter att skapa mer komplexa spel för mindre arbete från speldesignernas sida (mindre företag har mindre resurser), och dels på att det skapar mervärde – i form av unicitet och slumpmässighet, och därmed omspelbarhet för varje nytt spel [2] [3].

1.1. Arbetets avgränsning

Detta arbete kommer behandla procedurell innehållsgenerering med fokus på terränggenerering. Definitionen på terräng är i detta fall landskap i form av berg, kullar, slätter, hav, sjöar och floder – inte hela planeter eller galaxer.

Arbetet kommer att jämföra och analysera två metoder för PTG: Perlinbrus *(eng: Perlin Noise)* och Diamond-square algoritmen (*eng: Diamond-square Algorithm*). Terrängen genereras enligt ett s.k. "uppifrån-ned"-tillvägagångssätt beskrivet nedan, eftersom en modellering som simulerar klimat- och naturfenomen är för omfattande för denna analys.

Målet med jämförelsen är att utreda i vilka fall man kan tänka sig använda den ena metoden framför den andra. I detta mål ingår att undersöka hur svåra metoderna är att implementera, samt hur mycket mänsklig input som egentligen krävs för att generera terrängen. Vidare utreds om resultatet genast är acceptabelt, eller om man måste iterera genom resultaten tills man

når ett sådant. Sedan utreds om det acceptabla resultatet ännu måste justeras innan användning.

Metoderna kommer att användas tillsammans med spelmotorn Unity för att skapa en terräng av godtycklig storlek. En av orsakerna bakom detta val är tidigare bekantskap, inte bara med C#, utan också med Unitys API och arbetsflöde. Unity är också en av de större och vanligare spelmotorerna. Företaget hävdar att över hälften av alla spel som skapats för mobiler, datorer och spelkonsoler under 2020 har skapats med deras spelmotor [4].

2. Procedurell terränggenerering

En vanlig form av PCG är procedurell generering av terräng *(PTG, eng: procedural terrain generation)*. Terränger är så allmänt förekommande att det nästan alltid är ett givet att det finns någon form av terräng i ett spel [1]. En stor fördel med PTG är den möjliga storleken av den terräng som genereras. Man kan skapa omfattande galaxer och solsystem, vars planeter alla har egna, unika terränger. Dessa världar kan vara så omfattande och komplexa, att det manuella arbetet motsvarande handgjorda världar skulle kräva, gör dem omöjliga i praktiken. [5].

PTG kan tillämpas till flera områden. Trots att denna metod för terränggenerering i huvudsak använts inom spelindustrin för utveckling av (digitala) spel har principerna för PTG (och PCG) utnyttjats för bordsrollspel som *Dungeons & Dragons*, om än i form av analoga tabeller som används i samband med tärningar [3]. Då dessa blivit allt vanligare i form av digitala eller virtuella bordsspel, har också innehållsgenerering i form av digitala (och ofta procedurella) slumpgeneratorer, blivit allt vanligare [6]. Även inom filmindustrin har PTG blivit allt vanligare. Exempelvis Disneys tv-serie *The Mandalorian* använder procedurellt genererade landskap, som skapats i spelmotorn Unreal Engine [7].

En metod för generering av terräng är en modell som simulerar klimat- och naturfenomen, såsom regn och plattektonik. Med hjälp av denna metod skapar man en mer verklighetstrogen terräng [5].

Ett alternativt sätt att generera terräng är med hjälp av en uppifrån och ned-metod. Man genererar landmassan i form av kontinenter och bergskedjor – man skapar alltså terrängen med hjälp av en algoritmiskt genererad höjdkarta. Höjdkartan skapas i form av ett tvådimensionellt rutnät *(eng: lattice, grid)*, där varje cell består av ett värde som beskriver jämnheten (höjden) för en cell. Vanligt förekommande algoritmer som Perlinbrus, Simplex-brus *(eng: Simplex Noise)* eller Diamond-square algoritmen använder denna metod för terränggenerering [5].

2.1. Brus

En höjdkarta kan skapas på flera sätt, men en vanlig metod är en höjdkarta baserad på brus *(eng: noise)* som är en icke-reguljär primitiv funktion. Eftersom generering av brus är (till synes) slumpmässigt skapas inte några mönster, vilket annars skulle leda till att terrängen verkar monoton. Det är värt att notera att genereringen är pseudo-slumpmässig – det handlar alltså inte om äkta slumpmässighet. Detta beror dels på att äkta slumpmässighet är rätt ovanligt inom datavetenskap, och dels på man inte egentligen vill ha äkta slumpmässighet vid PTG [8].

Ett enkelt exempel på brus är s.k. vitt brus *(eng: white noise)* som bl.a. uppstår hos tv-apparater då inget program sänds till deras analoga mottagare. Vitt brus kan lätt genereras med pseudo-slumptal *(PRN, eng: Pseudo-Random Numbers),* men denna form av brus lämpar sig inte så väl för PTG-ändamål [8].

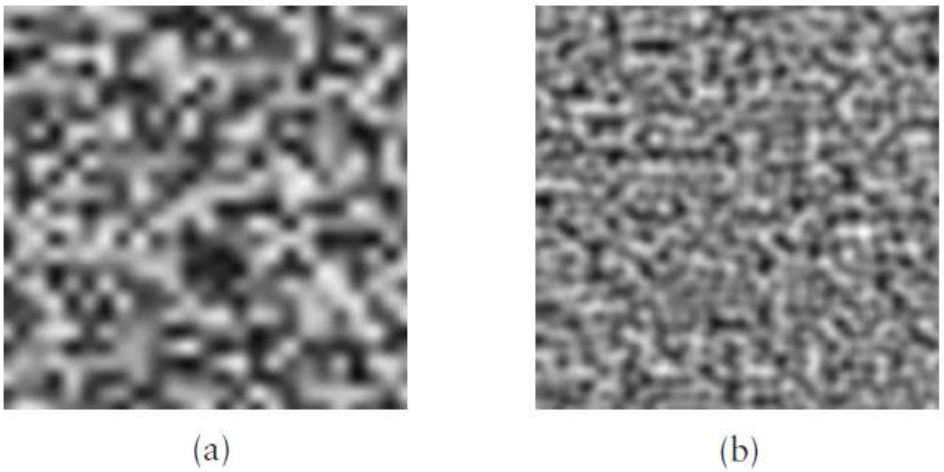
För PTG genereras brus ofta i form av ett rektangulärt rutnät. Denna form av brus kallas gitterbrus *(eng: lattice noise)* och varje punkt kallas för en gitterpunkt (*eng: lattice point*). I denna form av rutnät, representerar

ljusstyrkan på var gitterpunkt dess värde; desto ljusare punkt, desto högre värde. Ett rutnät som genererats för detta ändamål kallas för en intensitetskarta *(eng: intensity map)*. Om man också representerar terrängen i form av ett tvådimensionellt rutnät, där X- och Y-koordinaterna representerar longitud och latitud, representerar värdena i intensitetskartans gitterpunkter höjden på terrängen [1] [8].

Eftersom vitt brus är en form av värdebrus (*eng: value noise, vnoise*), är alla värden i rutnäten oberoende av varandra. Detta gör att värdebrus inte lämpar sig för terräng, eftersom elevationen på verklig terräng är statistisk beroende på närliggande terräng. Terräng på punkter närliggande till en bergstopp med hög elevation, har sannolik också hög elevation [1] [8].

2.1.1. Perlinbrus (Perlin Noise)

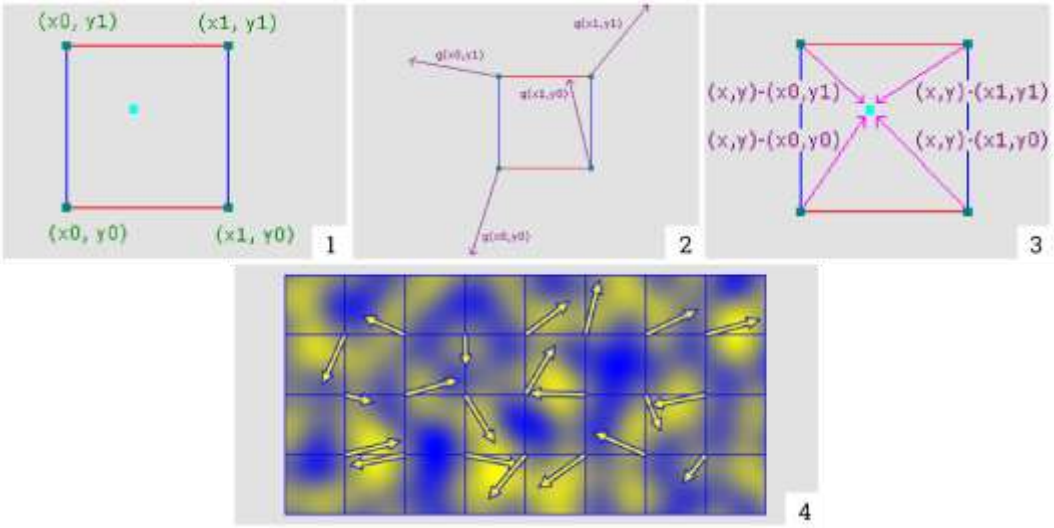
Perlinbrus är en form av lutningsbrus *(eng: gradient noise, gnoise)* skapat av Ken Perlin i 1983. Till skillnad från värdebrus, har lutningsbrus smidigare övergångar mellan gitterpunkterna eftersom värdena interpoleras på basis av sina grannar [8] [9].



Figur 1: Jämförelse mellan vnoise (a) och (b) gnoise i form av Perlinbrus [8].

Perlinbrus ses som en vanlig och lättimplementerad algoritm som ofta används vid procedurell texturgenerering [10]. Algoritmen definierar ett

jämnt fördelat rutnät där punkten för varje hörn antar ett heltalsvärde. För varje punkt definieras pseudo-slumptalsvektorer. För varje ruta på nätet väljs en godtycklig punkt inne i rutan, mot vilken distansvektorer definieras från hörnpunkten till den godtyckliga punkten. Lutningen uträknas genom skalärprodukten mellan de två vektorerna i en punkt. Slutligen interpoleras mellan alla fyra lutningsvärden kring en ruta. Detta upprepas för alla rutor i rutsystemet. [8].



Figur 2: Steg-för-steg Perlinbrus [9].

Ett viktigt koncept för generering av brus är oktaver *(eng: octaves)*. Oktaver är alla skilda "skikt" av brus som kombineras. Varje på varandra följande oktav har allt mindre inflytande på det slutliga resultatet. Brusen har vanligen olika frekvenser (förändringshastighet) och amplituder (förändringsmängd) [8] [9].

Ihållighet *(eng: persistence)* bestämmer avtagningshastigheten i amplituden hos varje på varandra följande oktav, och bestämmer hur mycket inflytande varje oktav har på slutresultatet [8] [9].

Frekvensen ökar hur ofta förändringar sker längs en definierad enhetslängd, och därmed även storlek på särdrag i terrängen. Frekvensen påverkas av en parameter kallad *lacunarity* som ändrar på frekvensen hos varje oktav [8] [9].

Procedurellt brus i allmänhet ses som ett grundläggande verktyg inom datorgrafik, och Perlinbrus är inte ett undantag. En av de största fördelarna med denna metod är att det är en lättimplementerad och väldokumenterad metod för PTG. Perlinbrus, likt andra brusmetoder, evalueras snabbt och använder inte mycket minne under exekvering [11].

Perlinbrus är rätt simpelt och då den används för terränggenerering har metoden inte så många avsevärda nackdelar. En nackdel är att man inte har någon direkt kontroll över det producerade bruset – det är slumpmässigt. Detta kan dock någorlunda motverkas om man i koden tillåter justering på frekvensen och ihålligheten Genom att öka dessa två parametrar skapas en ”grövre” och fragmenterad terräng [9].

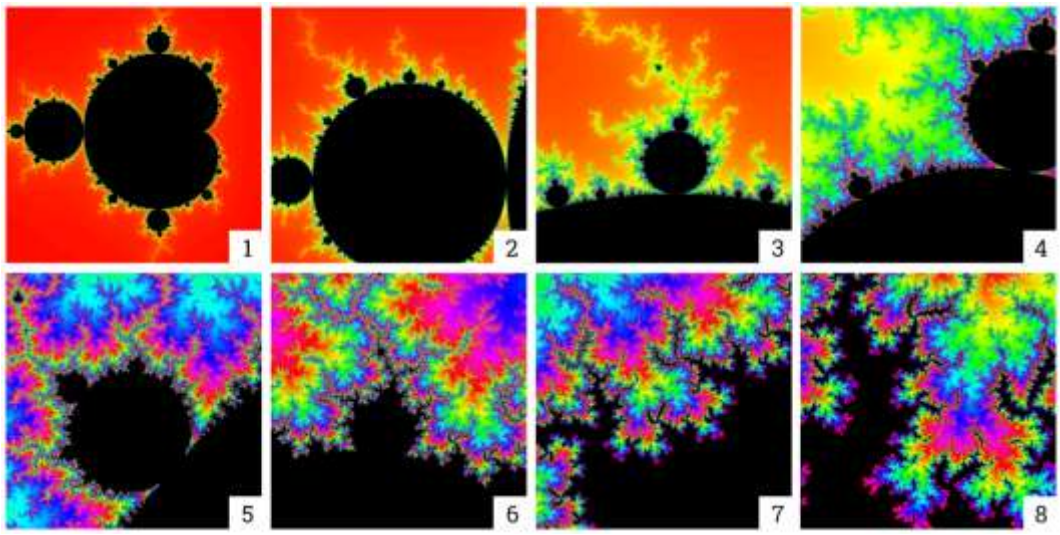
Om man jämför Perlinbrus med sin uppföljare, Simplex, kan man extrapolera ytterligare för- och nackdelar med metoden. Perlinbrus har högre komplexitet, $O(2^n)$, jämfört med Simplex, $O(n^2)$. Perlinbrus skapar också s.k. direktionella artefakter *(eng: dimensional artifacts)* som skapar ett rätt märkbart rutmönster. Detta beror på att Perlinbrus alltid använder ett rutnät oberoende av dimension, medan Simplex använder ett rutnät som byggs upp av n-simplex, beroende på vilken n-dimension som används. I två dimensioner är rutnätet 2-simplex – alltså trianglar (som skapar färre direktionella artefakter är kvadrater) [12].

En viktig fördel hos Perlinbrus i jämförelse till Simplex, är att det inte är patenterat. Den öppna källkoden bakom Perlinbruset är sannolikt en faktor bakom dess popularitet, trots att Simplex är en rätt objektiv förbättring. [5] [12].

2.2. Fraktaler

Fraktaler är geometriska objekt med en upprepande symmetri. De kan beskrivas som oändligt komplexa rekursiva mönster. Oberoende av fraktalernas skala, är ett fraktals mönster alltid sig själv likt – denna egenskap kallas självlikformighet *(eng: self-similarity).* Termen *"fractal"*

myntades av Benoît Mandelbrot i 1975, och en av de mest kända fraktalerna, Mandelbrotmängden, är uppkallad efter honom. Abstrakta fraktaler som Mandelbrotmängden kan genereras av en dator genom att iterera genom en enkel ekvation. I fallet med Mandelbrotmängden definieras ekvationen med talföljden z_n: $z_{n+1} = z_n^2 + c$, där c är de komplexa tal ur talmängden där talföljden inte går mot oändligheten då $z_0 = 0$. [13] [14].



Figur 3: Ett ytskrap av Mandelbrotmängden.

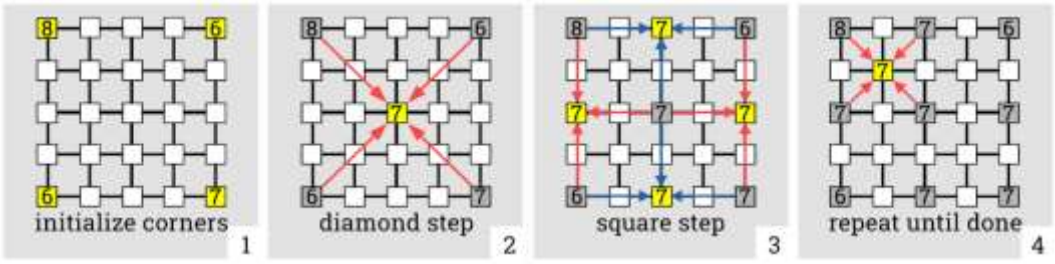
Mönster lika fraktaler förekommer i naturen, dock på ändlig skala. Allt från löv, blixtar, kustlinjer, till proteiner och DNA har fraktalegenskaper. Detta möjliggör att man kan modellera s.k. fraktallandskap *(eng: fractal landscapes)* med hjälp av fraktalytor och få naturliga terränger. Algoritmer som tillämpar fraktalgeometri för terränggenerering är t.ex. *Midpoint Displacement Method*, eller Diamond-square algoritmen (som är en variant av den förstnämnda) [13].

Fraktaler kan tillämpas inom terränggenerering i form av fraktallandskap. Eftersom många naturliga fenomen har någon form av självlikformighet som kan modelleras med fraktalytor, kan man med hjälp av stokastiska algoritmer generera fraktallandskap vars mål är att efterlikna naturlig terräng. Vid PTG kan man exempelvis skapa en kustlinje, som blir allt mer detaljerad desto mer man itererar. Denna metod kan även appliceras till t.ex. en höjdkarta [13].

2.2.1. Diamond-square algoritmen

Diamond-square algoritmen är en fraktalbaserad metod för generering av höjdkartor. Algoritmen presenterades i 1982 av Alain Fournier, Don Fussell och Loren C. Carpenter [15] .

Diamond-square algoritmen fungerar genom att iterera mellan två steg. Innan algoritmen körs, skapar man en tvådimensionell matris, ett rutnät, dit man matat in fyra frövärden *(eng: seed values)* i var hörn. Matrisens höjd och bredd måste vara 2n + 1. I det första steget *(diamond)* kalkyleras ur matrisen ett mittpunktsvärde som medeltalet mellan de fyra hörnpunkterna. Efter detta, körs det andra steget *(square)* som skapar "nya" hörnvärden. De värden som ligger exakt i mitten mellan två hörnvärden (i samma kolumn/rad) genereras som medeltalet av de två närliggande hörnvärden och det nya mittpunktsvärden – som tillsammans med det nya värdet formar en ny, mindre kvadrat. Diamantsteget körs fyra gånger, varefter nästan iteration börjar och körs tills alla celler tilldelats värden [15] [16] [17].

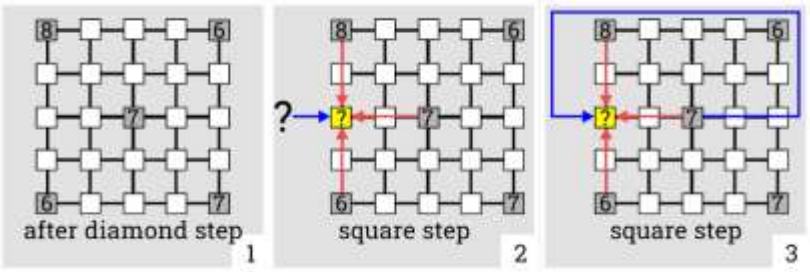


Figur 4: Steg-för-steg Diamond-square algoritm.

Eftersom Diamond-square algoritmen tillåter (och kräver) fyra frövärden, har man en del kontroll över det slutliga resultatet som räknas ut baserat på ursprungsvärdena. Man kan även avsäga sig denna kontroll genom att använda sig av PRN som frövärden, om man vill ha en ännu mindre deterministisk terräng [17].

Då Diamond-square metoden genererar kantvärden vid *square*-steget, använder metoden bara tre värden – eftersom det inte existerar ett fjärde värde. Man kan dock skapa upprepbara "brickor" *(eng: tiles)* om man

använder sig av det motsatta värdet på andra sidan höjdkartan. Detta kan, om terrängen är tillräckligt stor, skapa illusionen av en oändlig terräng. Implementationen är dock en hel del svårare, speciellt om man inte vill att kanterna på ”brickorna” ska vara alldeles för synliga. Detta kan ses som en nackdel, om man vill ha verklig oändlig terräng [17].



Figur 5: Hur man kunde skapa upprepbar terräng genom att ta ett fjärde värde från från andra sidan höjdkartan under square-steget.

Jämfört med Perlinbrus, borde Diamond-square ha kortare genereringstider (enligt en jämförelse mot Perlinbrus med fyra oktaver) trots att man ökar på upplösningen. Detta innebär att metoden borde ha bättre skalbarhet även på högre detaljnivå. För PTG med hög upplösning, som t.ex. på planetär skala, borde Diamond-square generera terrängen på kortare tid [18].

3. Jämförelse av Perlinbrus och Diamond-square

För att se hur algoritmerna presterar mot varandra, kommer två varandra liknande PTG-implementationer i spelmotorn Unity att jämföras.

I denna del av avhandlingen kommer Perlinbrus-implementation redogöras för först, följt av implementationen för Diamond-square algoritmen, varefter de slutligen kommer att jämföras sinsemellan. Algoritmerna jämförs på ett flertal områden som t.ex. uppfattad implementeringssvårighet, exekveringstid, och skalbarheten. Exekveringstiden mäts i ticks (1/10 000 000 sekund eller 1/10 000 millisekund), men eftersom ett flertal olika faktorer kan får det absoluta resultatet att variera, kommer resultaten presenteras i procentenheter. För att mäta skalbarheten kommer storleken på terrängen att ökas gradvis. Denna ökning följer tvåpotenser, eftersom Diamond-square algoritmen kräver en tvåpotens som inputparameter för storlek. Ytterligare kommer själva terrängerna att jämföras visuellt, eftersom om ena algoritmen är väldigt snabb men producerar helt osammanhängande terräng, uppstår det argument för att metoden i själva verket är sämre för PTG. Verklighetstrogen terräng är ofta ett viktigt krav för praktisk tillämpning av PTG.

Utöver själva genereringen av höjdkartan använder båda PTG-implementationerna samma programkod för att visa resultatet på skärmen, färglägga terrängen, etc. På detta vis bör inga skillnader som inte direkt hänför sig till genereringen av höjdkartor uppkomma.

3.1. Terränggenerering med Perlinbrus

Metoden för generering av Perlinbrus behöver i princip inga andra parametrar än storleken på kartan. Men med hjälp av ytterligare parametrar som antalet oktaver, ihållighetsvärdet, *lacunarity-värdet*, brusskalan *(eng: noise scale)* och kompensationsvärdena *(eng: offset)*.

Terrängen genereras i två dimensioner, men en tredimensionell implementation är också fullständigt möjlig med relativt lite kodmodifikation. I en tredimensionell terränggenerering skulle punkterna som genereras i metoden fungera som hörn och brusvärdena skulle appliceras till polygonytornas höjdkoordinater som framställs mellan hörnen (trianglar är ett vanligt val av polygon för detta ändamål). En 3D-implementation skulle också jämförts i detta arbete om tidsramarna tillåtit.

3.1.1. Algoritmen

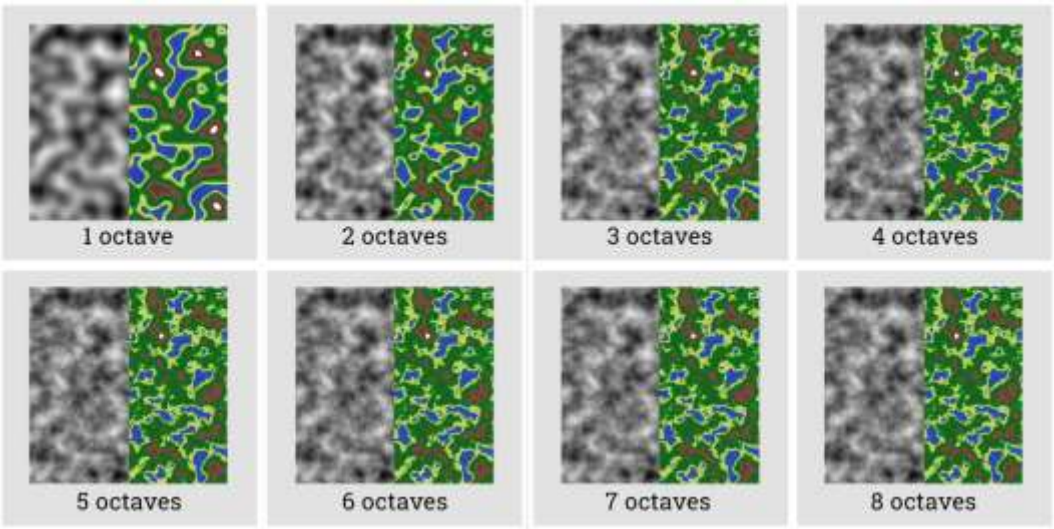
I korthet, returnerar algoritmen en flyttalsmatris vars storlek är baserad på två insättningsparametrar för höjd och bredd. Detta görs genom två nästlade for-loopar, en för y-axeln och en för x-axeln. Ytterligare, om det genereras värden för fler än en oktav, körs ytterligare en loop.

Värdena genereras med hjälp av metoden Mathf.PerlinNoise() ur Unitys API som tar två flyttal (sampelvärde för x och y) och returnerar ett flyttal mellan 0.0 och 1.0. Mathf.PerlinNoise() genererar alltså ett Perlinbrus och värdet som returneras tas fram med hjälp av input-parametrarna, dvs. x- och y-koordinaterna för en punkt i bruset. Detta görs för flera punkter och sparas i en flyttalsmatris, vars element tillsammans bildar en bruskarta *(eng: noise map)*. En enskild punkt skulle bara producera ett flyttal som representerar ett individuellt brusvärde *(eng: noise value)*.

X- och Y-inputvärdena kan justeras med hjälp av andra parametrar som t.ex. brusskalan eller kompenseringsvärden för att manipulera själva bruset i Mathf.PerlinNoise()-funktionen. Brusskalan ökar storleken på bruset, vilket i praktiken betyder att man ”zoomar” in eller ut på bruset. Kompenseringsvärdena flyttar brusets position på X- och/eller Y-axlarna, och ger illusionen att man flyttar kameran enligt väderstrecken.

Implementationen tillåter även för användaren att välja hur många oktaver används. Dock eftersom varje extra oktav i princip genererar ett nytt brus,

påverkar detta rätt linjärt på prestandan. Dessutom, om ihållighetsvärdet är tillräckligt lågt, blir effekten också mindre märkbar för varje på varandra följande oktav. Ett oktav-värde kring 4–8 verkar producera önskvärt resultat med valda parametrar – och efter 6–8 oktaver blir det allt svårare att se förändringar, som man kan se i Figur 6. Genom att justera på antalet oktaver, kan man i praktiken bestämma hurdan detaljnivå man vill ha på bruset.



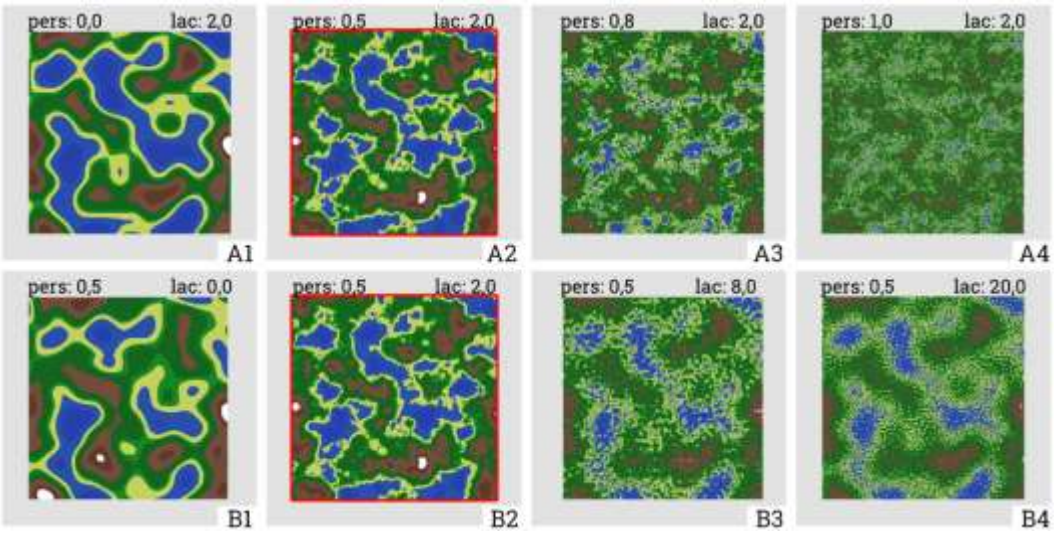
Figur 6: Hur antalet oktaver påverkar bruset. (1024x1024, Persistence: 0,5; Lacunarity; 2)

Exekveringstiden ökade med ungefär 25% för varje oktav, med undantag för bytet från 1 till 2 oktaver, där exekveringstiden i medeltal ökade med 54%.

Man kan även justera ihålligheten och *lacunarity*-värdet för att ytterligare manipulera terrängen. Genom att justera ihållighetsvärdet, bestämmer man hur mycket varje oktav bidrar till den övergripande formen av bruset. Som man kan se i Figur 7: A1-A4Figur 1: Jämförelse mellan vnoise (a) och (b) gnoise i form av Perlinbrus ., bidrar ett högre ihållighetsvärde till en allt mer fragmenterad terräng. Då varje oktav är ett skilt skikt av brus, helt olika varandra, skapar ett för högt ihållighetsvärde en osammanhängande terräng – terrängerna har knappt några gemensamma särdrag, och om de blandas för mycket, fås en terräng som saknar koherens. Även i fallet A4, trots att ihållighetsvärdet är på det högsta tillåtna, kan man ändå vagt urskilja den

första oktavens särdrag. Ett ihållighetvärde på 0,5 visade sig producera tillräckligt realistisk och önskvärd terräng.

Lacunarity-variabeln justerar frekvensen, hur mycket detalj varje oktav bidrar med för slutprodukten. Då *Lacunarity* modifierar oktaver, krävs ett nollskilt värde på ihålligheten. Eftersom ett högre *lacunarity*-värde leder till allt högre frekvenser i oktaverna, betyder detta i praktiken att detaljnivån ökar och terrängen verkar ”grövre” eller ”skarpare”. Om man här ser på Figur 7 och jämför med hur ihålligheten påverkar själva terrängens uppbyggnad, ser man hur *lacunarity* inte påverkar den övergripande formen av terrängen, utan bara ”konturerna”, t.ex. vid kustlinjer och där regionerna (färgerna) byts. Ett *lacunarity*-värde på 2 producerade önskvärt resultat (i kombination med ovannämnda ihållighetsvärde på 0,5).



Figur 7: A-raden visar hur ihålligheten förändrar terrängen, medan B-raden visar hur lacunarity förändrar terrängen. A2 och B2 är samma bild.

Terrängen färgläggs på basis av en flyttalsskala som definieras av användaren. Antalet terrängtyper eller regioner definieras också. Eftersom Perlinbrus implementationen består av en tvådimensionell matris med flyttalsvärden från 0 till 1, definieras även de olika terrängnivåerna enligt samma skala. I denna implementation måste flyttalsvärdena för gränserna justeras manuellt, så terrängens biom blir deterministisk. Ett alternativ hade varit att skapa en matris med flera biomer (varje biom skulle vara en lista

med n flyttal) och välja slumpmässigt – men eftersom terrängmålningen inte har någon egentlig relevans i jämförelse (ingen påverkan på prestandan) lämnades den rätt outvecklad.

Eftersom terrängen konstant modifieras vid parameterförändring, blir jämförelse av en viss genererad terräng omöjlig. Det vill säga, det går inte att bara aningen modifiera terrängen, eftersom genast en parameter ändrar värde, genereras ett nytt brus. Detta väghinder övervanns genom att för varje genererat brus skapa ett frövärde *(eng: seed)* som hålls konstant, trots att parametrarna förändras. På detta sätt kan man återskapa ett brus om man har rätt frövärde, vilket tillåter att för minimala justeringar (t.ex. antalet oktaver, ihållighetsvärde, kompenseringsvärden) utan att terrängen slumpmässigt genereras på nytt. Detta tillåter en tydligare jämförelse av hur de olika parametrarna påverkar terrängen.

3.2. Terränggenerering med Diamond-square

Likt terrängen genererad av Perlinbrus-implementationen, kommer terrängen genererad av Diamond-square också vara i två dimensioner. Trots att metoden i princip inte kräver mer än en storleksparameter, kan man med hjälp av *height*- och *offset*-kompenseringsvärdena bestämma hur mycket alla genererade värden varierar. Ytterligare, om *randomCorners* sanningsvärdet är sant, kan man definiera hörnvärden som bör ha inflytande över all genererad terräng. Terrängen kommer här att genereras både deterministiskt och stokastiskt, för att se hur mycket kontroll man egentligen har över genereringen. Dessa parametrar är de enda kontrollverktygen som metoden har över terrängen på helhetsnivå, och eftersom Diamond-square algoritmen inte använder sig av oktaver, existerar inte samma kontroll över detaljnivån.

Ett frövärde som sparar terrängen, likt det i Perlinbrus-implementeringen, lyckades inte implementeras, och lämnades på grund av hur liten inverkan det skulle ha för jämförelsen, samt på grund av tidsbrist. Detta leder till att

varje genererad terräng är helt olika den andra, till skillnad från Perlin-implementationen där man kan återskapa en terräng med rätt parametrar och frövärde.

3.2.1. Algoritmen

Diamond-square algoritmen tar som input ett storleksvärde, som sedan används som exponent n i formeln 2^n+1. Värdet ur formeln används som längd och bredd för höjdkartan och den tvådimensionella flyttalsmatrisen som håller höjdvärdena.

Efter detta bestäms de fyra frövärdena för hörnen. Detta görs antingen deterministiskt, då flyttalen anges som fyra separata inputvärden, eller stokastiskt med PRN-värden. Om värdena genereras slumpmässigt, antar de flyttal inom ett definierat intervall, vars övre och nedre gränser är *height*-inputparametern och dess negativa tal.

Före diamond- och square-stegen definieras ett grovhetsvärde och ett styckesstorleksvärde *(eng: chunk size)*. Grovhetsvärdet är ett kompenseringsvärde för höjdpunkterna, som hjälper producera en mer slumpmässig terräng. Som namnet säger, bestämmer värdet hur ”grov” terrängen är. Styckesstorleken är längden och bredden på det ”stycke” som algoritmen körs på; gränserna för diamond-steget.

Sedan påbörjas diamond-steget, som med hjälp av medeltalet av hörnvärdena, (samt med ett PRN-kompensationsvärde, vars intervall ges som inputparameter) räknar ut ett mittpunktsvärde för ’diamanten’.

Efter detta utförs square-steget, som räknar ut nya hörnvärden, halvvägs mellan varje hörnpunkt längs det valda stycket. Detta värdes räknas ut som ett nytt medeltal med existerande hörnvärden, det nya mittpunktsvärdet och ett nytt PRN-värde med samma kompensationsvärde som intervall.

Slutligen halveras grovheten och styckesstorleksvärdet, och om loopen nått sitt slut, returneras höjdkartan för visning.

Ett problem med denna implementation är att eftersom höjdvärdena som returneras i höjdkartan inte är normaliserade mellan 0 och 1, måste gränsvärdena för terrängen justeras manuellt om man vill ha en färglagd terräng. Dessa terrängvärden är heller inte lika intuitiva att bestämma, då intervallet inte är normaliserat.

3.3. Jämförelse av resultat

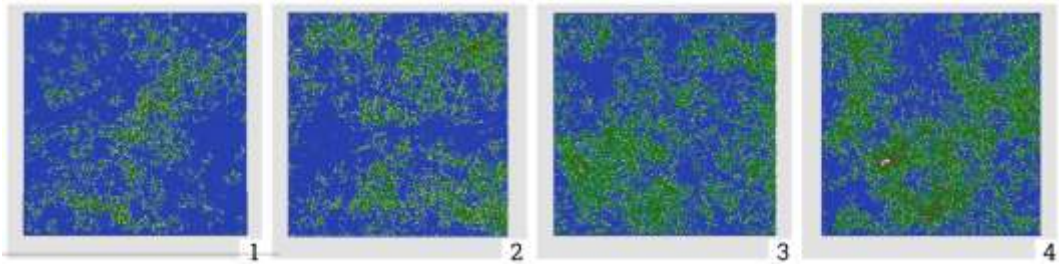
Eftersom Diamond-square metoden kräver en höjd/breddparameter i form av tvåpotenser, kommer också de kartor som genereras med hjälp av Perlinbrus för jämförelsen skull vara tvåpotenser (även om metoden inte kräver detta). Utöver detta, eftersom Diamond-square alltid producerar en höjdkarta med samma bredd och höjd, kommer metoden för Perlinbrus också tvingas producera höjdkartor i kvadratform (trots att detta igen inte är ett krav på metoden).

3.3.1. Genererad Terräng

Terrängen som genereras av denna implementering av Perlinbrus, är genast sammanhängande och koherent. Perlinbrus-implementationen framträder som om den genererar bättre terräng med ett lägre antal flyttal.

Den terräng som produceras av Diamond-square implementationen, däremot, är väldigt fragmenterad, se Figur 8 nedan. Det verkar existera många enstaka flyttalsvärden som verkar rätt ologiskt placerade. Då höjdkartan färgläggs med terräng, uppstår den som väldigt ”prickig” och inte värst verklighetstrogen. Då man ökar storleksparametern till det högsta Unity tillåter innan minnet tar slut, börjar man få något som lite mer börjar likna en realistisk terräng. Frågan uppstår då om problemet är algoritmen i sig själv eller om det är fast i implementationen, men en snabb nätsökning visar att många lyckats åstadkomma klart bättre resultat med egna implementationer

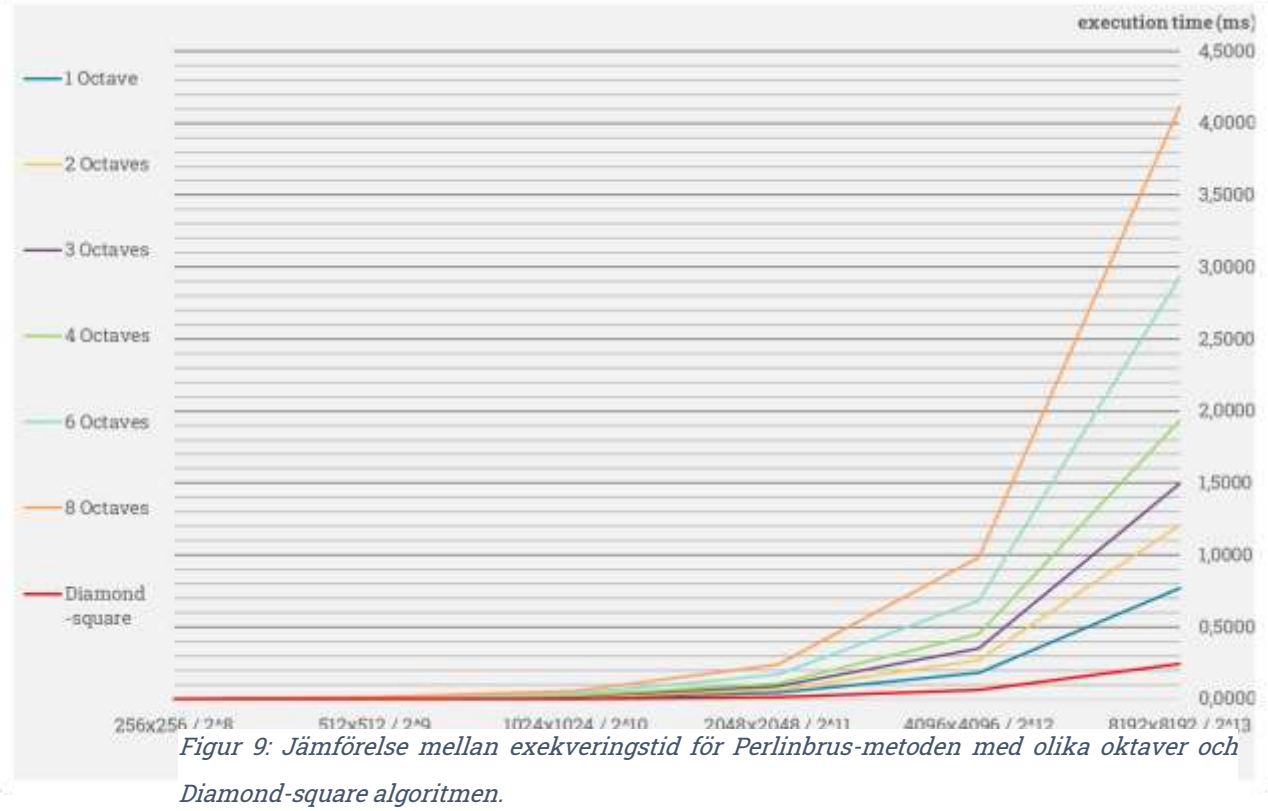
av samma algoritm. Detta tyder på att problemet är en bristfällig implementation, och med detta i åtanke kan man inte med säkerhet påstå att Diamond-square algoritmen producerar en mindre verklighetstrogen terräng.



Figur 8: Terräng genererad med Diamond-square.

3.3.1. Prestanda och skalbarhet

För exekveringstiden mättes hur länge det tog för var funktion att generera värdena för terrängen. För sex stycken storlekar på kartorna genererades 15 stycken för var storlek. Ytterligare, för Perlinbrus genererades också 15 höjdkartor för varje oktav. Från dessa tas medelvärdet som jämförelsevärde. Resultaten kan ses i Figur 9 nedan.



Figur 9: Jämförelse mellan exekveringstid för Perlinbrus-metoden med olika oktaver och Diamond-square algoritmen.

Denna implementation av Diamond-square algoritmen genererade sin terräng betydligt snabbare än Perlinbrus-metoden. Diamond-square genererade sina kartor i medeltal över 200% snabbare än den snabbaste Perlinbrus-iterationen, dvs. den som enbart använde sig en oktav.

Diamond-square algoritmen nådde minnesprogramfel då storleksparametern var över 13, dvs. då pixellängden för en sida översteg $2^{13}+1$; maximalstorleken blir 8193x8193 pixlar.

Perlinbrus-implementationen däremot, trots att exekveringstiden sköt i höjden, klarade av att producera en höjdkarta som var över 1,8 gånger större än den som producerats med Diamond-square, både vad beträffar höjd och bredd. Exekveringen nådde sitt minnesprogramfel före den genererat en höjdkarta som var 2 gånger större. Den högsta pixelstorleken ligger alltså mellan 14746x14746 och 16384x16384.

Den relativa ökning i exekveringstiden hos båda metoderna då höjdkartans storlek dubblades, steg på en rätt lika nivå. För varje steg, ökade exekveringstiden med runt 300%, oberoende algoritm, eller i Perlinbrus-metodens fall, oberoende antal oktaver.

Trots att man i praktiken hellre använder flera mindre terränger som lappas ihop, istället för en enorm terräng, tyder ovanstående resultat tyder på att Perlinbrus-metoden har bättre skalbarhet (åtminstone vad beträffar dessa implementationer). Perlinbrus-metoden i detta fall använde sig även av fyra oktaver, vilket i princip betyder att den genererade fyra stycken skilda brus; Perlinbrus genererade fyra skilda bruskartor som alla för sig är större än en enda Diamond-square höjdkarta.

4. Sammanfattning

Det finns otaliga metoder för procedurell terränggenerering och bland alla dessa metoder finns det även otaliga sätt att implementera dem. I takt med procedurell innehållsgenerering införs allt mer i allt fler sorters medier, lär också PTG följa denna trend. Som en följd av detta, är det sannolikt att det blir allt viktigare att hitta en metod för PTG som producerar tillräckligt verklighetstrogen terräng med tillräcklig effektivitet.

I jämförelsen hos dessa två implementationer genererade Perlinbrus en bättre, mer koherent terräng. Det krävs inte mycket fantasi hos användaren för att se att det som genererats är en karta. Implementationen hade en större maxstorlek och mer kontroll över detaljnivån, men exekveringstiden var alltid högre än Diamond-square algoritmen.

Diamond-square algoritmen presterade klart mycket snabbare med ungefär hälften av 1-oktavs Perlinbrus exekveringstid. Eftersom man vill generera en terräng med någon form av detaljnivå, är det dock mer lämpligt att jämföra med en Perlinbrus-karta som genererats med fler oktaver. Om man då jämför med hur snabbt Diamond-square algoritmen genererar sin höjdkarta mot 4-oktavs Perlinbrus, genererar Diamond-square sin terräng ca 87% snabbare. Båda metoderna hade ungefär samma procentuella ökning för varje steg upp i terrängens pixelstorlek; för varje gång bredden och höjden förbubblades, ökade genereringstiden med ungefär 300%. Höjdkartorna som genereras av Diamond-square är dock mer fragmenterade och osammanhängande, vilket gör dem mindre verklighetstrogna. Verklighetstrogenheten är dock inte något som kan mätas objektivt, och det verkade som om problemen till detta låg i implementationen snarare än algoritmen själv.

Enligt mina resultat är Diamond-square algoritmen ett bättre alternativ om exekveringstiden är den absolut viktigaste faktorn. Perlinbrus-metoden är att föredra om man vill generera större kartor med högre detaljnivå. Huruvida den genererade terrängen verkligen är mer verklighetstrogen eller ej hos

endera metoden, kan jag inte direkt säga – men bland dessa två implementationer är Perlinbrus-metoden klart bättre.

Överlag verkar som om det krävs en bättre förståelse över Diamond-square algoritmen, eftersom en potentiellt medelmåttig implementation kastar lite mörker över delar av resultatet.

Det hade varit väldigt intressant att se hur en tredje dimension hade påverkat resultaten. I en tidig prototypimplementering av en 3D Diamond-square terräng, fick jag klart bättre resultat vad beträffar hur verklighetstrogen terrängen var – vilket tyder på att implementering inte hade några större brister (som 2D-implementation av Diamond-square verkar tyda på). Ursprungsplanen var att generera terräng både i 2D och 3D, men på grund av tidsbrist blev planera skrinlagda.

Källförteckning

[1] N. Shaker, J. Togelius and M. J. Nelson, Procedural Content Generation in Games, Cham: Springer International Publishing Switzerland, 2016.

[2] M. Hendrikx, S. Meijer, J. van deer Velden and A. Iosup, "Procedural Content Generation for Games: A Survey," *ACM Transactions on Multimedia Computing, Communications, and Applications,* pp. 1-22, February 2013.

[3] G. Smith, "An Analog History of Procedural Content Generation," in *Foundations of Digital Games 2015*, Pacific Grove, CA, 2015.

[4] Unity Software Inc., "Our Company," Unity Software Inc., 13 February 2021. [Online]. Available: https://web.archive.org/web/20210213090802/https://unity.com/our-company. [Accessed 26 February 2021].

[5] T. X. Short and T. Adams, Procedural Generation in Game Design, Boca Raton, FL: CRC Press, 2017.

[6] J. Lai and A. Chen, "The Digital Future of Tabletop Games," andreessen horowitz, 1 September 2020. [Online]. Available: https://web.archive.org/web/20210124004804/https://a16z.com/2020/09/01/tabletop-games-go-digital. [Accessed 26 February 2021].

[7] C. Gartenberg, "How The Mandalorian teamed up with Fortnite creator Epic Games to create its digital sets," The Verge, 20 February 2020. [Online]. Available: https://www.theverge.com/2020/2/20/21145671/mandalorian-sets-stagecraft-epic-games-ilm-fortnite-baby-yoda-digital. [Accessed 25 February 2021].

[8] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley, Texturing & Modeling - A Procedural Approach, San Francisco: Morgan Kaufmann Publishers, 2003.

[9] A. Biagioli, "Understanding Perlin Noise," Adrian's Soapbox, 9 August 2014. [Online]. Available: https://web.archive.org/web/20210107091057/https://adrianb.io/2014/08/09/perlinnoise.html.. [Accessed 25 February 2021].

[10] X. Zhang, C. Zhang and W. Dong, "Generation of Cloud Image Based on Perlin Noise," in *2010 International Conference on Multimedia Communications*, Hong Kong, 2010.

[11] A. Lagae, S. Lefebvre, R. Cook, T. Derose, G. Drettakis, D. S. Ebert, J. P. Lewis, K. Perlin and M. Swicker, "A Survey of Procedural Noise Functions," *Computer Graphics Forum,* vol. 29, no. 8, pp. 2579-2600, 2010.

[12] S. Gustavson, "Simplex Noise Demystified," 22 March 2005.

[13] B. B. Mandelbrot, The Fractal Geometry of Nature, New York: W.H. Freeman and Company, 1982.

[14] Fractal Foundation, "What Are Fractals?," Fractal Foundation, [Online]. Available: https://fractalfoundation.org/resources/what-are-fractals/. [Accessed 15 March 2021].

[15] G. S. P. Miller, "The Definition and Rendering of Terrain Maps," *ACM SIGGRAPH Computer Graphics,* vol. 20, no. 4, p. 39–48, 1986.

[16] N. O'Brien, "Diamond-Square Algorithm Explanation and C++ Implementration," 10 August 2018. [Online]. Available: https://web.archive.org/web/20191124081728/https://medium.com/@nickobrien/diamond-square-algorithm-explanation-and-c-implementation-5efa891e486f. [Accessed 9 March 2021].

[17] S. Losh, "Terrain Generation with Diamond Square," 27 June 2016. [Online]. Available: https://web.archive.org/web/20201108100027/https://stevelosh.com/blog/2016/06/diamond-square/. [Accessed 9 March 2021].

[18] R. J. Vitacion, "Procedural Generation of Planetary-Scale Terrains in Virtual Reality," CALIFORNIA STATE UNIVERSITY, Northridge, 2018.

Figurförteckning

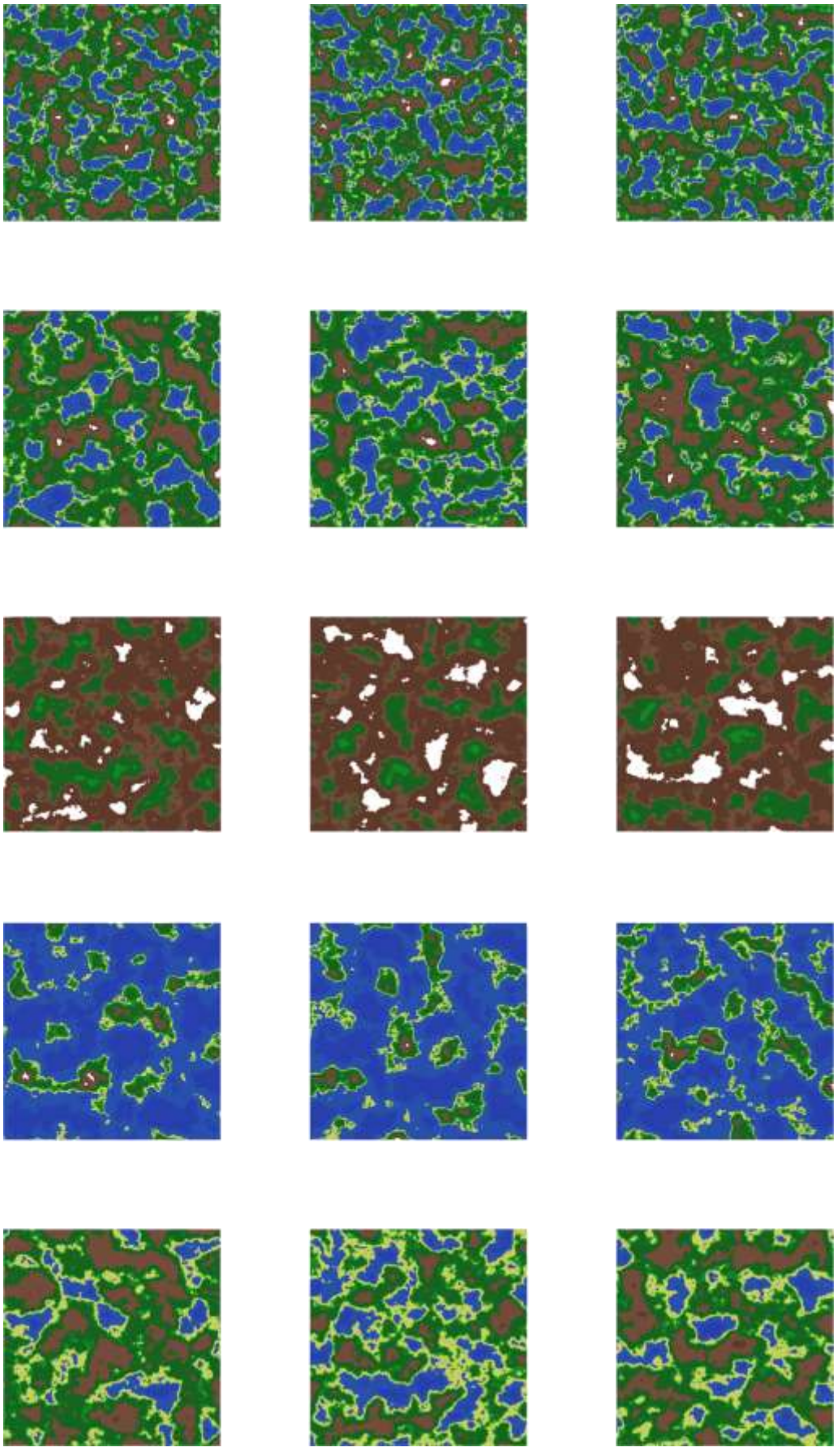
Figur 1: Jämförelse mellan vnoise (a) och (b) gnoise i form av Perlinbrus [8].4
Figur 2: Steg-för-steg Perlinbrus [9].....5
Figur 3: Ett ytskrap av Mandelbrotmängden 7
Figur 4: Steg-för-steg Diamond-square. 8
Figur 5: Upprepbar terräng under square-steget. 9
Figur 6: Hur antalet oktaver påverkar bruset.....12
Figur 7: Ihållighet och Lacunarity. 13
Figur 8: Terräng genererad med Diamond-square 17
Figur 9: Jämförelse mellan exekveringstid.....17

Bilaga A: Perlinbrus programkod

```
public static float [,] GenerateNoiseMap(int mapWidth, int mapHeight, int seed, float noiseScale,
int octaves, float persistence, float lacunarity, Vector2 offset)
{
    float[,] noiseMap = new float[mapWidth, mapHeight];

    System.Random prng = new System.Random(seed);

    // Allows for vertical and horizontal offsetting of noise
    Vector2[] octaveOffsets = new Vector2[octaves];
    for (int i = 0; i < octaves; i++)
    {
        // Offset limits set +-100000 since float has ~7 decimals of accuracy
        float offsetX = prng.Next(-100000, 100000) + offset.x;
        float offsetY = prng.Next(-100000, 100000) + offset.y;
        octaveOffsets[i] = new Vector2(offsetX, offsetY);
    }

    // Ensures that noiseScale isn't 0
    if(noiseScale <= 0)
    {
        noiseScale = 0.0000001f;
    }

    // Instatiating to their lowest/highest values, ensuring they'll always be updated further down
    float maxNoiseHeight = float.MinValue;
    float minNoiseHeight = float.MaxValue;

    // Using these allows for centered 'zooming' when modifying noise scale
    float halfWidth = mapWidth / 2f;
    float halfHeight = mapHeight / 2f;

    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            // These are modified by the persistence and the lacunarity parameters
            float amplitude = 1;
            float frequency = 1;
            float noiseHeight = 0;

            // Octaves loop
            for (int i = 0; i < octaves; i++)
            {
                // Higher frequency <=> sample points further apart <=> height values faster
                float sampleX = (x - halfWidth) / noiseScale * frequency + octaveOffsets[i].x;
                float sampleY = (y - halfHeight) / noiseScale * frequency + octaveOffsets[i].y;

                // Multiplying by 2 and subtracting 1 allows for negative height
                float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;

                // Increase noiseHeight, based on Perlin value of each octave
                noiseHeight += perlinValue * amplitude;

                amplitude *= persistence;
                frequency *= lacunarity;
            }
```

```
            // Updates the range of values for noise map
            if (noiseHeight > maxNoiseHeight)
            {
                maxNoiseHeight = noiseHeight;
            }
            else if (noiseHeight < minNoiseHeight)
            {
                minNoiseHeight = noiseHeight;
            }

            noiseMap[x,y] = noiseHeight;
        }
    }

    // Normalizes noiseMap to return values between 0 and 1
    for (int y = 0; y < mapHeight; y++)
    {
        for (int x = 0; x < mapWidth; x++)
        {
            noiseMap[x,y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap [x,y]);
        }
    }
    return noiseMap;
}
```

Bilaga B: Perlinbrus terrängsampel



Bilaga C: Diamond-square algoritmens programkod

```
public static float[,] GenerateDSquareMap(int mapSize, float height, float offset, float topLeft,
float topRight, float botLeft, float botRight, bool randomCorners)
{
    // Sets the mapsize to 2^n+1 (ensuring its always uneven)
    int size = (int)(Mathf.Pow(2f, mapSize) + 1);

    // Easier to use the divisions between cells than writing "size-1" everywhere
    int divisions = size - 1;

    float[,] heightMap = new float[size, size];

    if (randomCorners == false)
    {
        //-- Assign defined values for the corners.

        ///  Top right
        heightMap[0, 0] = topRight;

        /// Bottom right
        heightMap[0, size - 1] = botRight;

        /// Top left
        heightMap[size - 1, 0] = topLeft;

        /// Bottom left
        heightMap[size - 1, size - 1] = botLeft;

    } else {
        //-- Assign random values for the corners.
        ///  Top right
        heightMap[0, 0] = Random.Range(-height, height);

        /// Bottom right
        heightMap[0, size - 1] = Random.Range(-height, height);

        /// Top left
        heightMap[size - 1, 0] = Random.Range(-height, height);

        /// Bottom left
        heightMap[size - 1, size - 1] = Random.Range(-height, height);
    }

    int chunkSize = size - 1;

    // Adjustable, could be put as a input parameter
    float roughness = 0.8f;


    while (chunkSize > 1)
    {
        int half = chunkSize / 2;

        //-- Diamond Step
        for (int y = 0; y < divisions; y+=chunkSize)
        {
            for (int x = 0; x < divisions; x+=chunkSize)
            {
                heightMap[y + half, x + half] =
                    (heightMap[y,               x]+
                        heightMap[y,               x + chunkSize] +
                        heightMap[y + chunkSize,   x] +
```

```
                    heightMap[y + chunkSize,    x + chunkSize])
                    / 4 + Random.Range(-offset, offset);
            }
        }

        //-- Square Step
        for (int y = 0; y < divisions; y+=half)
        {
            for (int x = ((y+half)%chunkSize); x < divisions; x+=chunkSize)
            {
                float sum = 0;
                int count = 0;

                if (x == 0)
                {
                    x = chunkSize;
                }
                if (x - half > 0)
                {
                    sum += heightMap[y, x - half];
                    count += 1;
                }
                if (x + half <= divisions)
                {
                    sum += heightMap[y, x + half];
                    count += 1;
                }
                if (y - half > 0)
                {
                    sum += heightMap[y - half, x];
                    count += 1;
                }
                if (y + half <= divisions)
                {
                    sum += heightMap[y + half, x];
                    count += 1;
                }
                heightMap[y, x] = sum / count + Random.Range(-offset, offset);
            }
        }
        chunkSize /= 2;
        roughness /= 2;
    }
    return heightMap;
}
```

Bilaga D: Diamond-square terrängsampel