HÖGSKOLAN I SKÖVDE

Examensarbete

Procedurell generering av terräng Perlin noise eller Diamond-Square

med fokus på exekveringstid och framkomlighet

Procedural generation of terrain Perlin noise or Diamond-Square

With focus on execution time and good exploration

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2016

Johan Lautakoski

Sammanfattning

Arbetet handlar om vilken algoritm som är bäst för att procedurellt generera terräng. Är Diamond-Square bättre eller sämre än vad Perlin noise när de jämförs på exekveringstid och framkomlighet. Algoritmerna är implementerade i Unity där de körs för att få fram exekveringstid och Flood fill används för att ta reda på framkomligheten. Algoritmerna kördes 1000 gånger var på tre olika kartstorlekar för att få fram ett genomsnitt. Resultatet visar att Diamond-Square är snabbare än vad Perlin noise är men Perlin noise har bättre framkomlighet.

Nyckelord: Perlin noise, Diamond-Square, procedurell generering, terräng, framkomlighet

Abstract

This project deals with which algorithm is best for procedural terrain generation. Is Diamond Square better or worse than Perlin noise when they are compared on execution time and exploration. The algorithms are implemented in Unity, where they are tested to get the execution time and Flood Fill is used to determine exploration. The algorithms were run 1000 times each on three different map sizes to obtain an average. The results show that Diamond Square is faster than Perlin noise is but Perlin noise has better exploration.

Keywords: Perlin noise, Diamond-Square, procedural generation, terrain, exploration

Innehållsförteckning

1 Introduktion 1

2 Bakgrund 2

2.1 PCG av banor 2

2.1.1 Utplacering av objekt (artefakt) i banor 3

2.2 PCG av terräng 3

2.2.1 Fraktaler 4

2.2.2 Perlin noise 5

2.3 Evaluering av banor 5

2.4 Relaterade arbeten 6

2.4.1 PCG av terräng 6

2.4.2 Tidigare examensarbeten på högskolan i Skövde 6

3 Problemformulering 7

3.1 Problem 7

3.2 Metodbeskrivning 7

3.2.1 Metod 7

3.2.2 Genomförande 8

3.2.3 Metoddiskussion 8

4 Projektbeskrivning 10

4.1 Miljö 10

4.2 Implementation 10

4.2.1 Perlin noise 10

4.2.2 Diamond-Square 10

4.2.3 Framkomlighet 11

4.2.4 Programvariabler 11

4.3 Progressionsexempel 11

5 Utvärdering 14

5.1 Presentation av undersökning 14

5.2 Analys 14

5.2.1 50x50m terrängstorlek 14

5.2.2 75x75m terrängstorlek 15

5.2.3 100x100m terrängstorlek 16

5.3 Slutsatser 17

6 Avslutande diskussion 18

6.1 Sammanfattning 18

6.2 Diskussion 18

6.3 Framtida arbete 19

Referenser 20

1 Introduktion

Procedurell generering refererar till automatiskt eller semi-automatiskt skapande av innehåll med hjälp av en eller flera algoritmer. I det här arbetet undersöks hur procedurell generering kan användas för att skapa terrängformationer.

Bakgrunds-delen består av insamlad forskning som är relaterad till ämnet. Den är upplagd så att den börjar med en överblick av procedurell generering banor/terränger för att sen fördjupa sig i hur Perlin noise och Diamond-Square fungerar och för att avslutningsvis presenterar andra relaterad arbeten och ta upp hur validering av banor fungerar.

I problem-delen kommer det att tas upp några problem som finns när den här typen av arbete utförs, samt viktiga detaljer att ha i åtanke. En hypotes och en frågeställning kommer också att vara med, som förhoppningsvis besvaras med kommande implementation.

Metod-delen kommer att innehålla en diskussion och hur arbetet ska göras för att uppfylla de krav som ställts. Det är också en överläggning om den valda metoden är den bästa för att uppfylla kraven och vilka alternativ som finns. Det kommer även vara ett segment där det står vad som vill uppnås med implementeringen av arbetet och vad för resultat den kommer att ge.

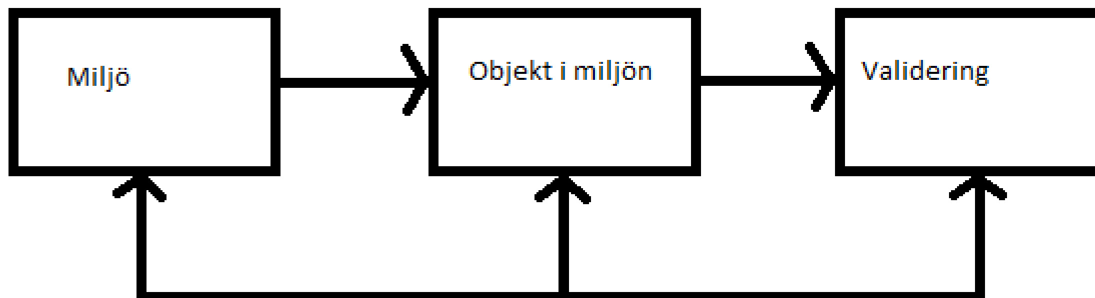
Implementerings-delen kommer att beskriva hur arbetet implementerats i form av vilken miljö och vilka designval som gjorts och varför. Den kommer även att visa hur arbetet har fortskridit under utvecklingen av implementationen.

Analys-delen kommer att beskriva hur utförandet av undersökningen gick till samt vilken information som togs fram. Informationen kommer att presenteras i form av tabeller som det sen görs en analys på. Analys-delen avslutas med ett kort stycke om vilka slutsatser som går att dra från informationen.

Diskussions-delen kommer innehålla en längre sammanfattning över hela arbetet. Det kommer finnas en diskussion över det resultat som kommer att tas fram i form av vad som skulle ha gått att göra bättre, varför resultat blev som det blev samt reflektioner kring hur arbetet ställer sig till etik och samhällelig nytta. Den kommer att avslutas med förslag på vad som går att göra för att utöka detta specifika arbete samt förslag på andra arbeten som är relaterad till samma ämne.

2 Bakgrund

Vid utvecklingen av spel går det att använda sig av procedurellt genererat innehåll för att skapa unika banor varje gång spelet spelas. Enligt Georgios m.fl. (2011) och Georgios m.fl. (2015) är procedurellt genererat innehåll, procedural content generation (PCG), ett allt viktigare tekniskt område inom modern människa-dator interaktion.



Figur 1 Översikt om hur PCG går till när en bana skapas. (egen bild)

2.1 PCG av banor

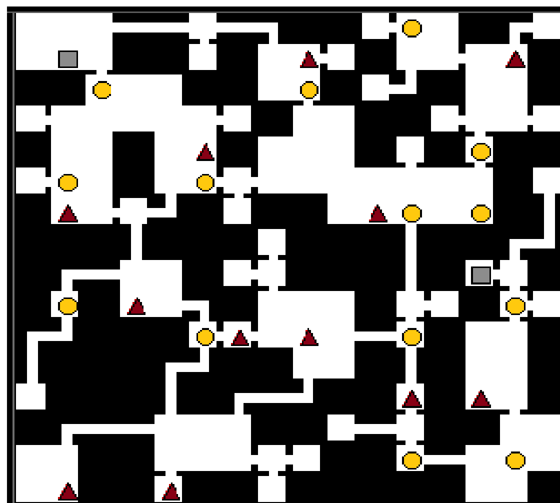
När PCG används för att generera banor går processen igenom flera steg. Det börjar med att miljön skapas och valideras mot kraven för spelet. Till exempel i ett openworld spel kan framkomlighet vara ett krav. Efter att miljön är godkänd genereras objekten (artefakt) som ska ingå i spelet t.ex. lådor, träd, hus eller bilar. Efter det valideras banan i sin helhet. Det som undersöks är om banan är inbjudande d.v.s. vill man spendera tid i den här miljön, är den framkomlig, finns resurser i närheten av starten eller liknande? Förutom att validera en sak i taget går det att göra all validering i slutet d.v.s. att man genererar miljön och fyller den med objekt för att slutligen validera allt. Se Figur 1 för att få en bild över processen till PCG av en bana.

När PCG används för att göra banor är ett av målen att avlasta designern genom att generera banorna istället för att designern ska göra det manuellt. Det gäller framförallt för stora open world banor som t.ex. Skyrim eller Borderlands. PCG av banor används även för att skapa variation vilket ger ökad levnadstid för spelet genom att banorna blir unika som t.ex. Roguelike-generen eller XCOM 2.

Ett sätt för att procedurellt skapa variation är att använda sig av slumptal. Det finns två typer av slumptal, rena slumptal eller en sekvens av slumpade tal ett så kallat frö ”Seed”. Ett frö garanterar att du alltid får en likadan bana varje gång du använder just det fröet. Det finns nackdelar med att använda slumptal, exempelvis att framkomlighet inte kan garanteras. Det finns olika sätt att hantera det och olika algoritmer som går att använda. Ett exempel för att göra en framkomlig bana är att fylla hela banan med objekt och sedan ta bort objekt för att skapa rum och vägar.

2.1.1 Utplacering av objekt (artefakt) i banor

För att skapa en bana med variation kan man, förutom att slumpgenera terrängens eller miljöns utseende, också slumpa ut objekt t.ex. fiender, skatter, kameror eller NPC:er/civilpersoner. Till exempel i XCOM 2 där slumpas fienderna ut varje gång du laddar om en bana eller i Payday 2 där kameror och fiendens patrullmönster slumpas ut vid start av varje uppdrag. En fördel är att en ny miljö skapas varje gång utan risken att få en dålig layout. Istället erhålls en bra miljö som måste navigeras med vaksamhet. Nackdelar som kan uppstå är att det inte går att göra en s.k. ”speedrun” eftersom att objekten inte har samma plats. Detta diskuterar även Liapis m.fl. (2014) när de pratar om hur objekt kan slumpas in i en bana till Roguelike-dungeonspel och hur den slumpmässiga positioneringen av objekten används det för att skapa bra och unika banor genom att ha samma karta med slumpade positioner för fiender och skatter. I Figur 2 är det skatterna (gula cirklarna) och fienderna (röda trianglar) som är slumpmässigt utplacerade på banan.



Figur 2 Bana där de gula cirklarna motsvarar skatter och de röda trianglarna motsvara fiender. (efter Liapis m.fl. 2014)

2.2 PCG av terräng

Terräng representeras i spel med hjälp av polygoner. Polygonerna är vanligtvis uppbyggda av trianglar som i sin tur är uppbyggda av punkter och streck (Wikipedia, 2015). Det vanligaste sättet att spara punkternas koordinater är att använda sig av en höjdkarta.

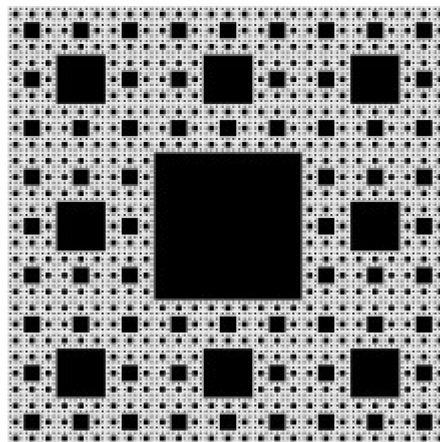
Vid procedurell generering av terräng finns det olika moment att tänka på för att skapa en omväxlande miljö. Några saker som ska övervägas är hur terränggenereringen ska hantera höjdskillnader för att den inte ska vara platt, hur vatten hanteras som t.ex. floder eller sjöar samt hanteringen av vegetation. I en undersökning av Smelik m.fl. (2009) har de tittat på viktiga egenskaper hos algoritmer inom de nämnda områdena. Det som har undersökts är hur realistisk terrängen blir, hur effektiv den är och hur mycket utvecklaren kan anpassa den. När det kommer till höjdskillnader skriver de om algoritmer som fraktaler och Perlin noise (se nedan 2.2.1 och 2.2.2 för mer information). När det kommer till vatten kan genereringen antingen äga rum i samband med höjdalgoritmen eller så kan det göras efteråt. Om det görs efteråt är det möjligt att använda samma princip som höjdalgoritmerna. Det som sker då är att man börja med en rak linje för att sedan dela ut den i grenar och där efter interpolera den för att få kurvor. Detta gäller bara när floder eller bäckar hanteras. I vegetationsdelen nämner de hur det går att procedurellt generera träd eller liknande

växtlighet. De förslag som ges är algoritmer som börjar från roten för att sedan lägga till mindre grenar och i slutet lägga till löven.

Det finns även andra metoder för att skapa terräng än vad Smelik m.fl. (2009) tog upp. Bevilacqua m.fl. (2011) undersökte algoritmer som Lindenmayersystemet förkortat L-systemet och Bezier/diffusions-kurvor. L-systemet går rekursivt igenom punkterna i terrängen och modifierar dem. L-systemet används t.ex. vid procedurell generering av växtlighet. Bezier/diffusions-kurvor används vanligtvis med andra algoritmer som t.ex. fraktaler och/eller Perlin noise men kan även användas självstående. Bezier/diffusions-kurvor är bra för att sätta gränser på terrängen för att den inte ska bli för brant. Kurvorna måste anpassas manuellt vilket gör att det inte är en automatisk process. Detta kan leda till att mycket tid går åt att anpassa terrängen. Enligt Bevilacqua m.fl. (2011) kan processen att genera och anpassa terrängen ta 45 minuter men resultatet blir mer förutsägbart.

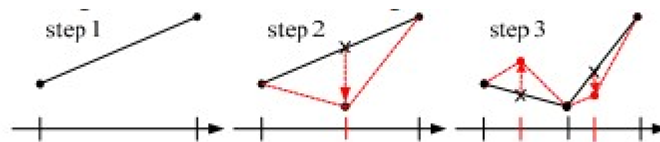
2.2.1 Fraktaler

Fraktaler är ett självlikformigt mönster i alla skalor d.v.s. ett mönster som upprepas om och om igen Wikipedia (2016). Genom att titta på Figur 3 syns ett mönster av kvadrater som blir mindre och mindre. Algoritmen utgår från en kvadrat som rekursivt delar sig i mindre bitar. Det finns klassiska fraktaler som mandelbrot eller Sierpinskis matta från Figur 3.



Figur 3 Sierpinskis matta nivå 6, en tvådimensionell fraktal (efter Wikipedia)

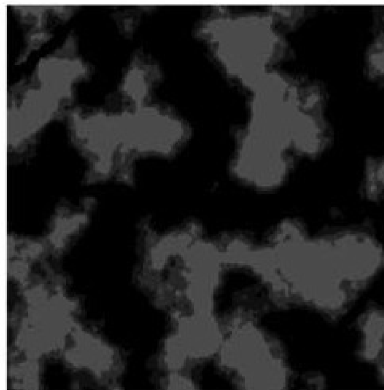
Fraktaler inom datorspel används framförallt inom terränggeneration men utöver det kan fraktaler appliceras på grafiska element som bilder eller 3D-volymer. Hong-Rui Wang m.fl. (2010) beskriver fraktalteori med två punkter. Den första är Fractal Brownian motion (FBM) vilket enligt dem är den bästa slumpgenereringsprocessen för att skapa en realistisk terräng. Den andra punkten är slumpmässig mittpunktsförflyttning, Random midpoint displacement (RMD), vilket är en metod för simulering av FBM. RMD bygger på att man tar mittpunkten på en linje och flyttar den samtidigt som linjen delas. Det upprepas det antal gånger som önskas se Figur 4 för en tydligare bild av hur det går till. Hong-Rui Wang m.fl. (2010) tar sedan upp en metod för att implementera RMD genom att använda algoritmen Diamond-Square. Diamond-Square har som inparameter antal rekursioner även kallat steglängd. Den har även en slumpfaktor som parameter. Förutom Diamond-Square finns det andra fraktal-algoritmer t.ex. L-systemet.



Figur 4 Slumpmässig mittpunktsförflyttning (efter Hong-Rui Wang m.fl. 2010).

2.2.2 Perlin noise

Perlin noise är en algoritm som används inom många fält relaterade till datorspel t.ex. shadergrafik, 2D-grafik och terränggenerering. För att skapa den klassiska Perlin noise algoritmen behövs generellt två funktioner en noisefunktion och en interpoleringsfunktion. Noisefunktionen tar ett heltal och returnerar ett närliggande tal. Det är viktigt att noisefunktionen alltid returnerar samma tal på samma inparameter annars är funktionen oanvändbar. För att sätta samman alla returnerade tal behövs interpoleringsfunktionen. Det finns tre metoder för att interpolera. Den första är linjärinterpolering den andra är cosinusinterpolering och den sista är kubisk interpolering enligt Xinyan Zhang m.fl. (2010). Perlin noise har inparametrar som antalet rekursioner d.v.s. hur många gånger det ska köras och vilken frekvens och amplitud. Genom att använda sig av Perlin noise skapas terräng som är har en mjuk övergång mellan plattmark och berg. Det är inte helt ovanligt att man matar in värdena från funktionen i en textur. Värdena representeras då av färgerna svart/vitt och skalor av grått (Figur 5 är ett exempel på det).



Figur 5 Perlin noise genererad bild (efter Schetinger m.fl. 2014).

2.3 Evaluering av banor

Enligt Liapis m.fl. (2014) finns tre attribut att undersöka vid evaluering procedurellt genererade banor. De tre attributen är symmetri, områdeskontroll och möjligheten att utforska. I t.ex. strategispel är symmetriaspekten viktig för att inte någon sida av bana ska få fördel. Områdeskontroll i strategispel evalueras för att kartan inte ska bli obalanserad dvs. att ena sida har fördel i förhållande till resurser. Utforskning undersöks för att se att man enkelt hitta resurser på banan och hur man kan skapa en stabil bas. För att evaluera dessa attribut finns det enligt Liapis m.fl. (2014) algoritmer för att beräkna hur bra en bana är. Förutom strategispel kan attributen appliceras på andra spel-genrer. De säger att evaluering av dessa attribut kan användas för banor som används till spel som involverar mer än en spelare som t.ex. League of Legends eller Counter Strike. Det går även att använda i Roguelike-spel där kartan förändras varje gång. I League of Legends fokuseras det på

symmetridelen för att ena laget inte ska få någon fördel medan i Roguelike-spel är utforskning huvudområdet.

2.4 Relaterade arbeten

2.4.1 PCG av terräng

Det finns många studier gjorda inom terränggenerering och olika sätt att göra det på. Till exempel Galin m.fl. (2015) undersökte huruvida hydrologi går att använda för att skapa terräng. Deras resultat visar att det är möjligt att skapa terräng och resultatet blir terrängformationer som har en bra övergång från plan mark till berg. Det finns också studier som utnyttjar erosion t.ex. en studie av Schetinger m.fl. (2014) som går ut på att skapa dalar medhjälp av erosion. Deras resultat är att det går att skapa realistiska dalar.

2.4.2 Tidigare examensarbeten på högskolan i Skövde

Det finns en del examensarbeten som har gjorts på högskolan i Skövde gällande procedurellt generad terräng eller kartor. Alexander Sällström (2008) har tittat på generering av höjdkartor. Han använde höjdkartor för att genera terräng som efterliknar områden i USA. Han kom fram till att två av teknikerna som han hade gjort var bättre än de andra. Den första tekniken som var bra var en kombination av FBM och voronoi med en egen funktion som väljer ett värde. Tekniken hade en bra representation av terräng och kunde ge ett stort utbud av terrängformationer med små justeringar av parametrarna. Den andra tekniken var mer komplex och gav ett liknande resultat som den första tekniken.

Rasmus Björk (2015) har gjort ett examensarbete som handlar procedurellt generering av banor med höjdskillnader. Han hade framkomlighet som prioritet. Han använde sig av sinuskurvor för att skapa sin terräng och resultatet var att två av funktionerna gav bra resultat. Eftersom att hans fokus låg i framkomlighet så är resultatet baserat på hur stor del av kartan som är framkomlig.

3 Problemformulering

3.1 Problem

Som tidigare nämnts finns det problem med att procedurellt generera terräng och problemen är olika till vilket typ av spel terrängen är till för. Ett strategispel behöver t.ex. en karta som är symmetrisk där resurser är jämnt fördelade och terräng är identisk på båda sidorna av kartan (Liapis m.fl. 2014). Ett openworld spel har t.ex. mer fokus på att miljön ska var utforskningsbar och har mindre fokus på fördelningen av resurser. I ett realtidsspel där terrängen genereras allt eftersom krävs det att algoritmen är effektiv för att spelet inte ska stanna upp under genereringen av terrängen. I ett spel som Minecraft där kartan kan få stora proportioner upp till miljontals meter krävs att terrängen har en upplevd variation.

Om en bana helt slumpmässigt genereras kan problem uppstå t.ex. den blir osammanhängande. Ett exempel på osammanhängande karta är i ett labyrintspel där det inte finns en väg mellan start och mål och/eller det är för många återvändsgränder (Liapis m.fl. 2014). Vid ren slumpgenerering av terräng kan det uppstå terrängformationer som går upp och ner utan struktur.

Vid användning av en existerande algoritm som t.ex. Perlin noise eller Diamond-Square är det viktigt att ha värden på algoritmens inparametrar som är anpassade för den terräng som ska generas. Som exempel har Perlin noise en styrparameter som anger antalet rekursioner för att avgöra graden av lutning på branter.

Problem med styrparameter till existerande algoritmer är att de måste evalueras och justeras för att terrängen skall anpassas till spelets krav. Evalueringen kan ske på två olika sätt antingen manuellt där en utvecklare inspekterar terrängen eller på automatisk väg med hjälp av en algoritm t.ex. Flood fill.

Frågeställningen lyder: Är Perlin noise bättre än fraktalalgoritmen Diamond-Square med avseende på tid och framkomlighet vid generering av terräng?

Hypotesen är att Perlin noise är effektivare men skapar terräng som är enformig i förhållande till fraktaler, som skapar större variation av terränger, men är mer beräkningskrävande. Båda algoritmerna kan lida av framkomlighetsproblem om parametrarna till algoritmerna inte är lämpliga.

3.2 Metodbeskrivning

3.2.1 Metod

För att evaluera hur effektiv lösningen är kommer tiden det tar att genererar terrängen mätas och jämföras mellan en implementation av Perlin noise och Diamond-Square. Anledningen till att tid mäts är för att algoritmen ska kunna vara användbar vid procedurell generering i realtid. För att något ska vara användbart i realtid bör genereringen ej ta mer än en sekund för att spelet inte ska saktas ner och för att det inte ska vara hål i terrängen.

Framkomligheten kommer mätas genom att undersöka hur stor del av banan det går att traversera med hjälp av en algoritm som heter Flood fill. Resultatet jämföras sedan mellan algoritmerna för att vilken som har bäst framkomlighet. För att en punkt i kartan ska räknas

som onåbar måste den ha ett värde som är avsevärt högre i förhållande till punkterna runt den. Anledningen till att framkomlighet mäts är för att ta reda på hur bra miljön är i förhållande till attributet utforskande från Liapís m.fl. (2014) studie.

Det här arbetet kommer att matematisk evalueras eftersom vi får ut numeriska värden. Det passar då bättre att jämföra dem matematiskt istället för att en person berättar sin åsikt om terrängerna. Det går även att göra många tester eftersom Flood fill snabbt går igenom terrängen och returnerar hur stor del av kartan som är traverserbar. Eftersom att Flood fill är snabbare än att fråga vad någon tycker om terrängen kan antalet tester som kan köras ökas och det ger möjlighet till parameterjusteringar och ett större underlag för genomsnittsvärdena erhålls.

3.2.2 Genomförande

För att undersöka om Perlin noise är bättre än Diamond-Square kommer algoritmerna att implementeras i Unity. Algoritmerna kommer påverka höjdkoordinaterna för punkterna in en polygon för att skapa terräng. Tiden det tar att exekvera algoritmerna kommer att sparas. Terrängen kommer sedan att evalueras på sin framkomlighet mot andra terrängstrukturer. Algoritmerna kommer att köras ett flertal gånger på olika kartstorlekar för att få fram ett genomsnitt över exekveringstiden och framkomligheten. Algoritmernas inparametrar kommer att justeras och den genererade terrängen kommer att manuellt evalueras för att avgöra om det är acceptabla inparametrar till testerna som ska köras. Inparametrarna är antalet rekursioner, steglängd och slumpvärdet som läggs på.

3.2.3 Metoddiskussion

Sättet som evalueringen sker på i det här arbetet är inte det enda sättet att göra det på. Ett annat sätt att evaluera framkomlighet är att importera kartan i ett existerande spel för att sedan gå runt med karaktären och se vart det är möjligt att ta sig. Genom att importera det till ett existerande spel får man regler för hur en karaktär rör sig i terrängen och vad som är för högt eller för brant. Anledningen till att det inte kommer att ske i det här arbetet är att tiden det tar att traversera banan i ett spel blir längre än om man använder algoritmen Flood Fill för att beräkna framkomligheten. Förutom att beräkningstiden för framkomligheten blir större så ger Flood Fill möjligheten till att sätta ett värde på vad som är för högt. Det går även att enkelt få fram en procentsats över hur mycket av kartan som går att traversera och det går att grafiskt visa var det är framkomligt.

Att använda Unity för att implementera algoritmerna och skapa terränger är inte det enda sättet att göra det på. Genom att använda Unity får man sämre tider för generationen på grund av att spelmotorn själv kräver prestanda, resultatet påverkas inte eftersom att algoritmerna jämförs mot varandra och prestandaförlusten är den samma för båda algoritmerna. De tiderna som registreras blir beroende av den hårdvara och mjukvara som används vid testtillfället och kan inte tas som generella för andra system.

Att ha en testgrupp för att manuellt evaluerar inparametrarna till algoritmerna är i det här fallet inte ett effektivt alternativ då det blir för tidskrävande och antalet parameter kombinationer är för många. Eftersom arbetet går ut på att undersöka vilken algoritm som är bättre i förhållande till exekveringstid och framkomlighet och inte på vilka inparametrar som ger bäst terräng.

Studien gör en matematisk jämförelse mellan tid och framkomlighet mellan algoritmerna för att avgöra vilken som har bäst framkomlighet och tidseffektivitet. Det är möjligt att ha en

användarstudie för att evaluera terrängernas framkomlighet. Dock ger en användarstudie ett subjektivt resultat vilket betyder att det är beroende på testpersonens förutsättningar t.ex. spelvana, preferens av spel-genre o.s.v. En användarstudie hade fungerat bättre för ett arbete som är mer grafisk eller äventyrs-orienterat. I det här arbetet är en matematisk analys att föredra eftersom det är numeriska värden som produceras.

4 Projektbeskrivning

4.1 Miljö

Som tidigare nämnts kommer implementeringen att ske i Unity (Unity Technologies 2015) med hjälp av programmeringsspråket C#. Tanken med arbetet är att det ska kunna användas som grund till liknande implementationer där ett spel kan använda procedurellt genererad terräng i realtid.

Programmet använder sig av 3D-koordinater från X, Y, Z axlarna, vilket är standard i Unity. X- och Z- axlarna motsvarar sidled och djupled, Y-axeln motsvarar höjdled. Perlin noise och fraktalalgoritmen Diamond-Square manipulerar Y-koordinaten för varje punkt till varje triangel i meshen. Y-värdet varierar mellan noll till oändligheten. Alla Y-värden som är lägre än noll sätts till noll eftersom sjöar inte hanteras i det här arbetet. Det är antalet punkter som avgör storleken till meshen. Avståndet mellan punkterna i X- och Z- led kommer vara 0,5 Unity-längdenhet vilket motsvarar 0,5 meter. Alltså är storleken baserad på hur många punkter det är i meshen t.ex. 101x101 punkter blir en karta på 50x50m eller 2500 kvadratmeter.

4.2 Implementation

För att skapa meshen används Unitys inbyggda system. För att bygga upp meshen har programmet två nästlade for-loopar som placerar alla punkter och sen ytterligare två nästlade for-loopar som länkar ihop punkterna till trianglarna. Punkterna är av Unitys inbyggda datastruktur Vector3 som sparas i en lista. Varken Perlin noise eller Diamond-Square använder egna datastrukturer utan manipulerar bara Y-koordinaterna i listan.

4.2.1 Perlin noise

Perlin noise är implementerad med hjälp av linjärinterpolering (Xinyan Zhang m.fl. 2010) och en funktion som slumpar tal istället för en hashtabell. Anledning till att en slumpfunktion används istället för en hashtabell är för att hashtabellen har en begränsning på storlek och är mer minneskrävande. Linjärinterpolering valdes för att den är effektivast att exekverar och det märkbara grafiska resultatet inte var tillräckligt stort för att kunna motivera användandet av en mer prestandakrävande interpoleringsfunktion. Pseudokoden för algoritmen finns i Appendix A.

4.2.2 Diamond-Square

Diamond-Square är implementerad så att den har två nästlade for-loopar som ger ett värde till mittpunkterna för en nivå t.ex. nivå 1 ger en mittpunkt, nivå 2 ger fyra mittpunkter o.s.v. Värdet som appliceras på mittpunkterna är ett genomsnitt av hörnen adderat med ett slumptal. Efteråt sker ett liknande steg för att ta fram värdet på mitten av kanterna till kvadraterna, värdet för mitten av kanten fås genom att ta genomsnittet av hörnen för den kanten och mittpunkten, adderat med ett slumptal. Detta sker rekursivt för antalet nivåer som önskas, ju fler nivåer desto mjukare terräng skapas. Pseudokoden för algoritmen finns i Appendix B.

4.2.3 Framkomlighet

Flood fill används för att beräkna framkomligheten. Flood fill är implementerad med hjälp av en stack som håller reda på vilka punkter som kan besökas. Stacken fylls på med grannarna till punkten som är nåbara d.v.s. inte är för högt upp eller för långt utan ligger inom gränsvärdet. Det finns också en lista som håller reda på alla punkter som redan besökts för att samma punkt inte ska besökas flera gånger. Höjdskillnadens gränsvärde bestäms av en variabel. Implementationen av Flood fill är förenklad i det här arbetet: slutpunktsvariabeln såväl som färgläggning av punkterna som går att besöka har tagits bort. I det här arbetet ska framkomligheten beräknas för hela terrängen vilket gör att det inte behövs en slutpunkt, utan algoritmen körs tills det inte finns fler punkter som går att besöka. Eftersom att det är numeriska värden som jämförs kommer en grafisk representation över framkomligheten inte att implementeras. När det kommer till startpunkterna används fyra stycken, en som är fast och är placerad på koordinaterna (0,0) och de övriga slumpas ut. Anledningen till att tre till startpunkter slumpas ut är för att minska risken för att algoritmen börjar på en position som resulterar i utebliven framkomlighet. Pseudokoden för algoritmen finns i Appendix C.

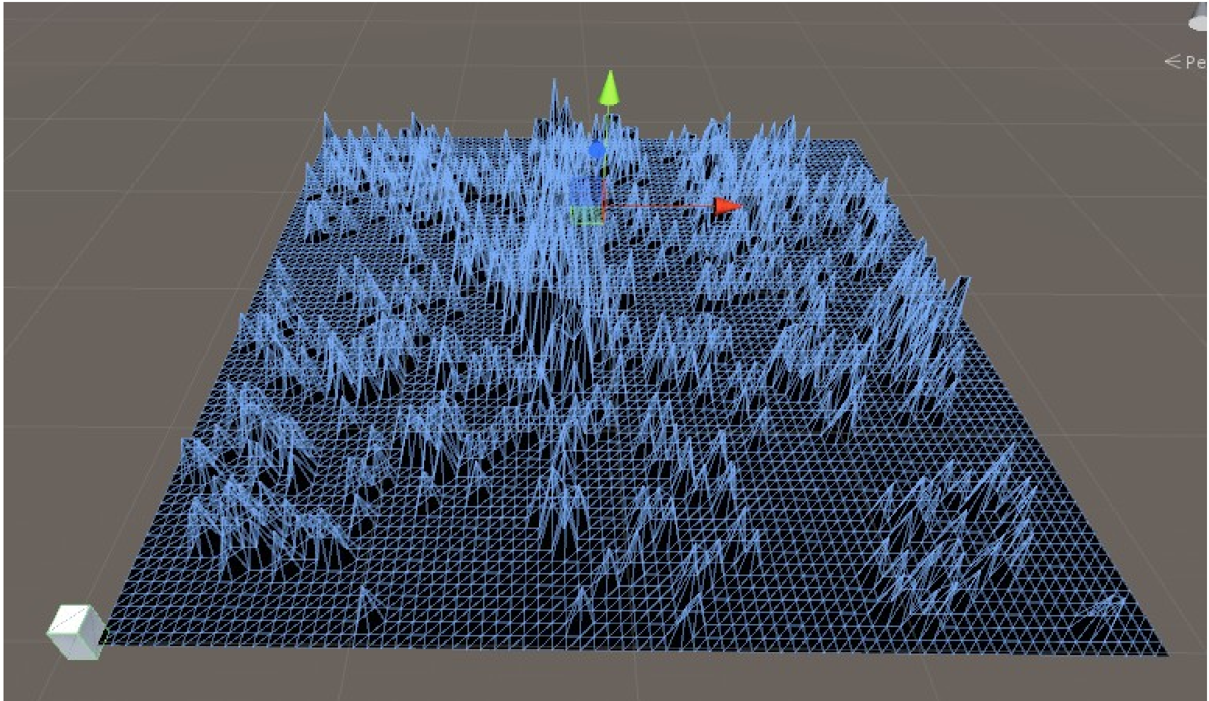
4.2.4 Programvariabler

Programmet har en del styrparametrar. En parameter är hur stor terrängen ska vara, det finns tre alternativ: liten 50x50m, mellan 75x75m och stor 100x100m. Det finns även en parameter för att välja vilken algoritm som ska användas. Diamond-Square styrs av två parametrar, den första är hur många iterationer som ska göras, den andra är hur stor slumpfaktor som ska adderas. För Perlin noise finns det en ställbar parameter och det är slumpfaktorn. Som tidigare nämnts om framkomlighet har Flood fill en parameter för höjd.

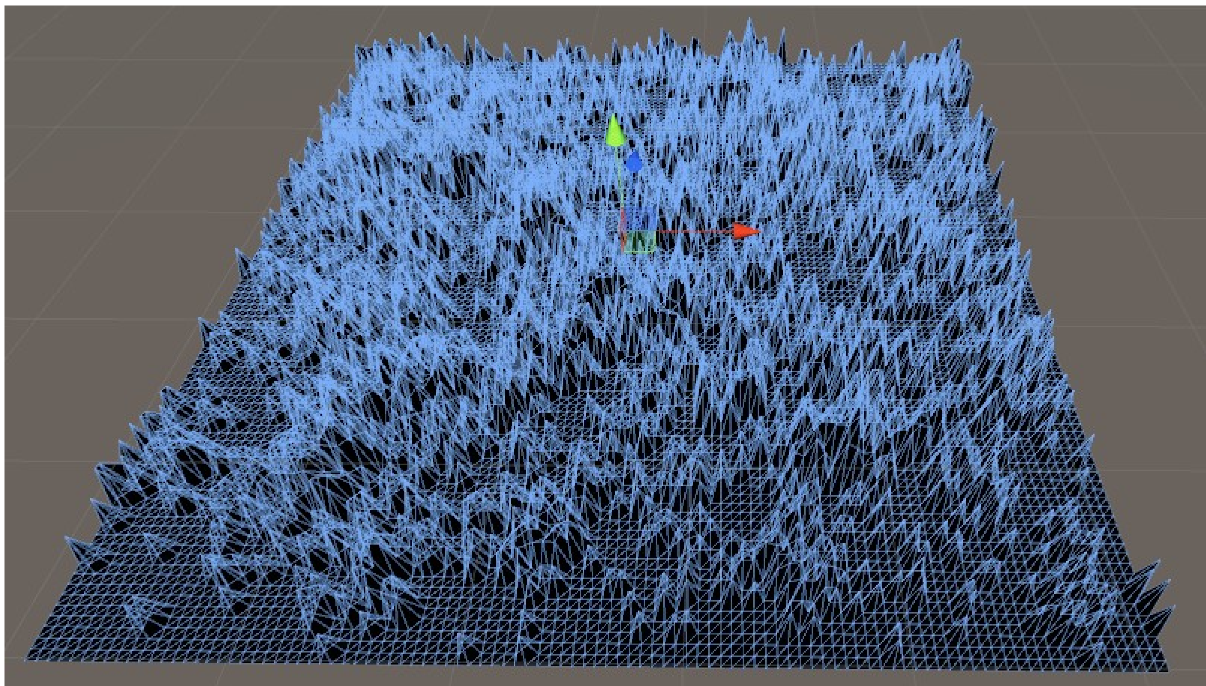
Det programmet ger som utdata är en procentsats över hur mycket av terrängen som går att traversera, hur lång tid det tar att exekvera en av algoritmerna, en standardavvikelse (Statistiska centralbyrån, 2016) för tid och framkomlighet samt en grafisk representation av terrängen.

4.3 Progressionsexempel

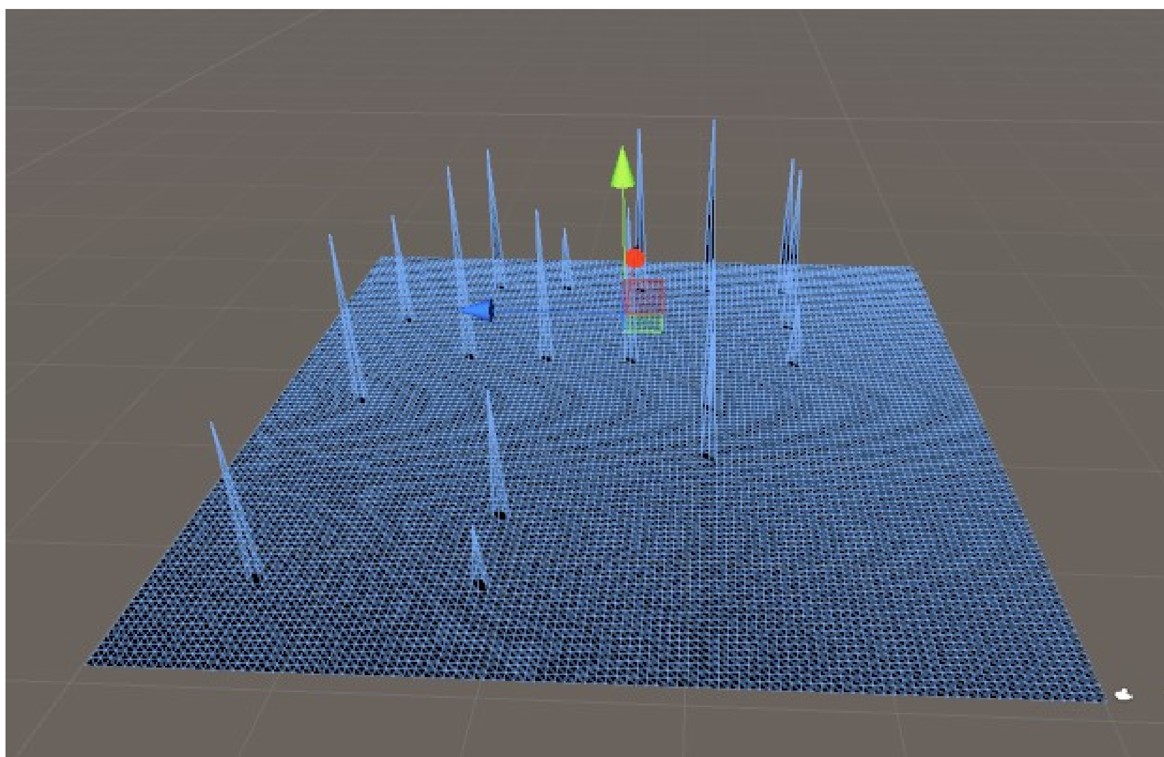
Under programmets utveckling har det skett några ändringar i hur programmet fungerar. Vid implementeringen av Perlin noise syntes det att när man har en högre frekvens får man ett resultat som inte är garanterat att fungera för den här typen av terränggenerering (se Figur 9 för ett exempel), på grund av det kommer frekvensen att vara ett i det här arbetet. Vid implementering av Diamond-Square fanns det inte några alternativ gällande hur den implementeras, utan den är implementerad enligt originaldesignen. Det som har tagit upp mycket av implementationstiden är modifieringar av parametrarna som styr algoritmen för att se till så att resultatet liknar någon form av terräng och inte blir för platt eller för bergig (se Figur 6-8 för exempel).



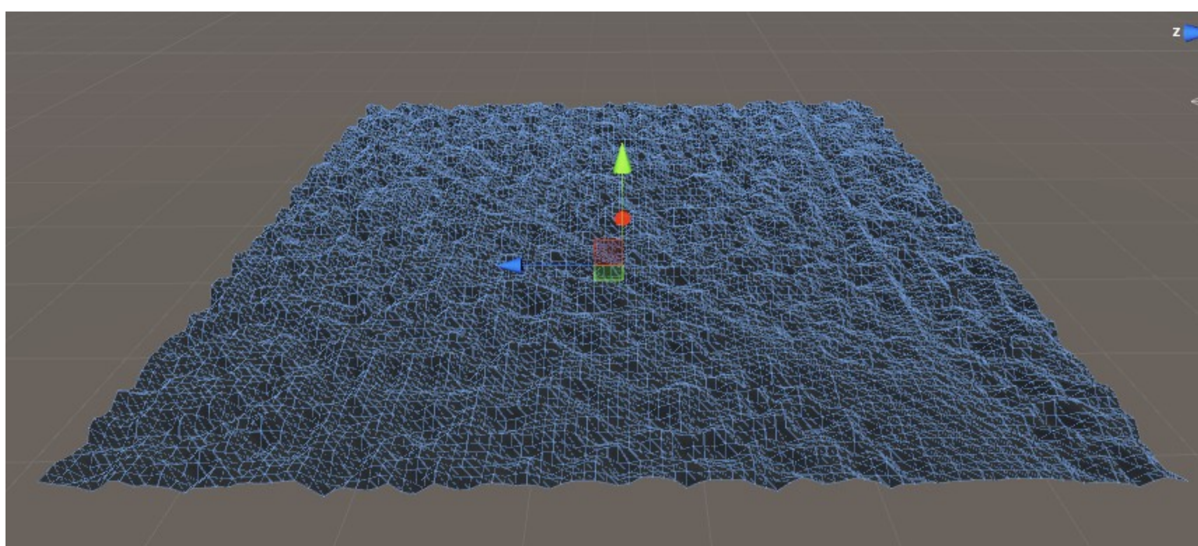
Figur 6 Ett acceptabelt resultat (Diamond-Square)



Figur 7 För mycket berg (Diamond-Square)



Figur 8 För lite berg (Diamond-Square)



Figur 9 Ett oönskat resultat (Perlin noise)

5 Utvärdering

5.1 Presentation av undersökning

För att undersöka vilken av Perlin noise och Diamond-Square som är bäst med avseende på exekveringstid och framkomlighet har algoritmerna exekverats 1000 gånger/terrängstorlek för att få fram ett så bra genomsnittsvärde som möjligt. De värden som har sparats under exekveringen är bästa, sämsta, genomsnitt och vad standardavvikelsen är för både tid och framkomlighet. Värdet för framkomligheten sparas i procent och tid sparas i ticks. Tio miljoner ticks motsvarar en sekund.

5.2 Analys

5.2.1 50x50m terrängstorlek

Genom att göra en analys på informationen i Tabell 1, går det avgöra att Perlin noise har bättre framkomlighet än vad Diamond-Square har. Den har även en lägre standardavvikelse. Diamond-Square är dock snabbare än vad Perlin noise är.

Inställningarna som användes vid exekvering av algoritmerna var:

Diamond-Square

- slumptalsfaktor 60
- antalet nivåer är 7

Perlin noise

- slumptalsfaktor 100

Flood fill

- höjdskillnad 0,5m

Tabell 1 Sammanställning av data från 50x50m terrängstorlek

Mätvärde	Diamond-Square	Perlin noise
Bästa tid	23 678 ticks	26 024 ticks
Sämsta tid	39 084 ticks	45 394 ticks
Genomsnittlig tid	24 854 ticks	27 127 ticks
Standardavvikelse tid	1 391 ticks	1 658 ticks
Bästa framkomlighet	90%	97%
Sämsta framkomlighet	51%	87%
Genomsnittlig framkomlighet	70%	92%
Standardavvikelse framkomlighet	6 procentenheter	2 procentenheter

5.2.2 75x75m terrängstorlek

I Tabell 2 kan man se att Diamond-Square är snabbare än vad Perlin noise är. Däremot är Perlin noise bättre när det gäller framkomlighet både i genomsnitt och när det kommer till bästa framkomlighet.

Inställningarna som användes vid exekvering av algoritmerna var:

Diamond-Square

- slumptalsfaktor 80
- antalet nivåer är 7

Perlin noise

- slumptalsfaktor 100

Flood fill

- höjdskillnad 0,5m

Tabell 2 Sammanställning av data från 75x75m terrängstorlek

Mätvärde	Diamond-Square	Perlin noise
Bästa tid	19 287 ticks	58 022 ticks
Sämsta tid	34 042 ticks	86 196 ticks
Genomsnittlig tid	20 064 ticks	60 785 ticks
Standardavvikelse tid	1 264 ticks	2 581 ticks
Bästa framkomlighet	91%	96%
Sämsta framkomlighet	88%	88%
Genomsnittlig framkomlighet	90%	92%
Standardavvikelse framkomlighet	0,5 procentenheter	2 procentenheter

5.2.3 100x100m terrängstorlek

Även när terrängstorleken är 100x100m så syns det att Diamond-Square är snabbare än vad Perlin noise är. Perlin noise har fortfarande ett bättre bästa resultat än vad Diamond-Square har när det kommer till framkomlighet. Båda algoritmerna har låg standardavvikelse när det kommer framkomligheten, och när det gäller tiden är standardavvikelsen relativt låg i förhållande till exekveringstiden.

Inställningarna som användes vid exekvering av algoritmerna var:

Diamond-Square

- slumptalsfaktor 110
- antalet nivåer är 7

Perlin noise

- slumptalsfaktor 100

Flood fill

- höjdskillnad 0,5m

Tabell 3 Sammanställning av data från 100x100m terrängstorlek

Mätvärde	Diamond-Square	Perlin noise
Bästa tid	38 187 ticks	103 253 ticks
Sämsta tid	67 555 ticks	157 819 ticks
Genomsnittlig tid	44 045 ticks	107 975 ticks
Standardavvikelse tid	2 345 ticks	5 070 ticks
Bästa framkomlighet	84%	95%
Sämsta framkomlighet	81%	88%
Genomsnittlig framkomlighet	82%	91%
Standardavvikelse framkomlighet	0,7 procentenheter	1 procentenhet

5.3 Slutsatser

De slutsatser som går att göra av den insamlade datan är att Diamond-Square är snabbare än Perlin noise. Ju större terrängen är desto större blir tidsskillnaden mellan algoritmerna. Standardavvikelsen är mellan 1000 och 5000 ticks för algoritmerna vilket kan kännas som att ett högt värde men det man måste ha i åtanke är att standardavvikelsen jämförs mot värden som är mellan 20 000 och 150 000 ticks och att 1 tick är 1/10 000 000 sekund. När det kommer till framkomligheten är Perlin noise bättre än Diamond-Square och standardavvikelsen är så pass låg att det går tydligt att avgöra att algoritmerna producerar ett konsekvent resultat.

Frågeställningen för det här arbetet är om Perlin noise är bättre eller sämre än Diamond-Square för att procedurellt generera terräng när exekveringstid och framkomlighet är bedömningsmallen. Utgår man från den insamlade informationen syns det tydligt att om exekveringstid är huvudkravet för algoritmen ska Diamond-Square användas. Är framkomlighet det viktigaste är Perlin noise ett bättre alternativ.

Slutsatsen är att: välj algoritm efter det som är viktigast, Diamond-Square för snabbhet, Perlin noise för framkomlighet.

6 Avslutande diskussion

6.1 Sammanfattning

Målet med det här arbetet var att avgöra vilken algoritm som är bättre Perlin noise eller Diamond-Square med avseende på exekveringstid och framkomlighet. Arbetet gick ut på att skapa en testmiljö för att kunna avgöra vilken av algoritmerna som är bäst med avseende på det tidigare nämnda kraven. Framkomligheten evalueras för att det är en viktig del av terräng/banskapande enligt Liapis m.fl. (2014). Tid evalueras för att få en uppfattning om algoritmerna är användbara till procedurell generering i realtid.

Implementeringen bestod av två moment, det första var implementeringen av rutnätet till terrängen, det andra var implementeringen av algoritmerna. Rutnät skapades med hjälp av en lista med koordinater för punkterna som byggde upp terrängen som sedan manipulerades av algoritmerna. Flood fill användes för att beräkna ur många procent av terrängen som är traverserbar.

Under undersökningens gång testades båda algoritmerna 1000 gånger var på tre olika terrängstorlekar för att få fram ett genomsnitt över hur lång exekveringstid algoritmerna har och hur bra framkomlighet terrängen som skapats har. Resultatet som togs fram under testningen sparades i en tabell för analys. Under analysen av informationen som togs fram går det att se hur Diamond-Square är en snabbare algoritm och att Perlin noise har bättre framkomlighet.

6.2 Diskussion

Det resultat som har tagits fram vid utförandet av exekveringen av artefakten visar att Diamond-Square är snabbare när nivå 7 är den högsta nivån som uppnås. Hade en annan nivå används kanske resultatet hade blivit annorlunda t.ex. en högre nivå hade resulterat i längre exekveringstid.

Parametrarna som har valts och används till testerna har tagits fram genom manuell granskning av de producerade terrängerna. Detta är inte ett vetenskapligt sätt att göra det på. Då arbetet inte går ut på att avgör vilka inparametrar algoritmerna ska ha för att skapa den bästa terrängen, utan vilken algoritm som är snabbast och/eller har bäst framkomlighet, går arbetet fortfarande att ha som grund när algoritmerna ska jämföras.

Resultatet som någon annan kan få vid implementering av algoritmerna kan skilja sig från den här undersökningen genom att de gör en annan implementering och tillämpning av algoritmerna. Resultatet från den här undersökningen är fortfarande relevant eftersom pseudokoden till implementeringarna av algoritmerna finns i Appendix A-B. Om pseudokoden används som en referensmall när algoritmerna implanteras till ett annat projekt kommer deras resultat bli liknande.

Terrängerna har bara bedömts med avseende på sin framkomlighet och inte någon av de två andra aspekterna (utforskande, symmetri) som Liapis m.fl. (2014) tar upp när de pratar om procedurell generering av terräng/banor. Detta gör att terrängerna inte med säkerhet är garanterade att fungera till alla typer av spel.

I dagsläget är procedurell generering av terränger begränsat till den virtuella världen och arbetet har där med inga direkta samhälleliga påverkningar i sin helhet. Några etiska aspekter går inte att identifiera i det här arbetet. Däremot går Flood fill-algoritmen att applicera på verkligheten, om man virtualiserar riktiga miljöer och exekverar sedan Flood fill på dem går det att simulera t.ex. ett snöskred. För att göra det måste Flood fill-algoritmen ändras. En ändring som skulle behövas göras är att nivån för vad som räknas som för högt måste gradvis ändras under exekvering och det får inte finnas en lägre nivå. Det går även att använda Flood fill för att beräkna framkomlighet på terränger som existerar i verkligheten.

6.3 Framtida arbete

Framtida projekt eller utökande av det här skulle kunna vara att göra fler tester med algoritmer som att öka antalet nivåer för Diamond-Square eller andra storlekar på kartorna. Det går även att implementera andra terränggenereringsalgoritmer t.ex. Lindenmayersystemet för att se hur de ställer sig mot Perlin noise och/eller Diamond-Square.

De procedurellt genererade terrängerna som har skapats är grundläggande. För att kunna använda terrängerna i ett riktigt spel måste ett antal saker läggas till. Bättre hantering av höjdskillnaderna är en av de viktigare delarna. I nuläget är terrängen kantig och ger ingen känsla av realism. Terränggenereringen bör även kunna hantera vattendrag vilket inte görs i det här arbetet.

Något annat som skulle gå att undersöka är om att det är möjligt att ta en existerande terrängformation, verklig eller procedurellt genererad, och implementera en algoritm som procedurellt generar områden för skog, städer eller liknande.

Referenser

Björk, R. (2015) *Procedurellt genererade banor med höjdskillnader och bra framkomlighet*. Tillgänglig på Internet: http://his.diva-portal.org/smash/get/diva2:819060/FULLTEXT01.pdf

De Carli, D.M., Pozzer, C.T., Schetinger, V. (2014). *Procedural generation of 3D canyons*. SIBGRAPI Conference on Graphics, Patterns and Images.

Firaxis Games (2016) XCOM 2 (Version 1.0) [Datorprogram]. Firaxis Games Tillgänglig på internet: https://xcom.com/.

Génevaux, J., Galin, E., Guérin, E., Peytavie, A., Benes, B. (2013). *Terrain Generation Using Procedural Models Based on Hydrology*. Tillgänglig på Internet: http://arches.liris.cnrs.fr/publications/articles/SIGGRAPH2013_PCG_Terrain.pdf

Liapis, A., Yannakakis, G.N., Togelius, J. (2014) *Towards a Generic Method of Evaluating Game Levels*. Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference

Mojang (2015) Minecraft (Version 1.9) [Datorprogram]. Mojang. Tillgänglig på internet: HTTPs://minecraft.net/.

Olsen, J (2004) *Real-time Procedural Terrain Generation*. Tillgänglig på Internet: http://web.mit.edu/cesium/Public/terrain.pdf

Overkill Software (2015) Payday 2 (Version 5.3.2) [Datorprogram]. Overkill Software. Tillgänglig på internet: http://www.crimenet.info/gate?returnUrl=/.

Smelik, R.M, de Kraker, K.J., Groenewegen, S.A. (2009). *A Survey of Procedural Methods for Terrain Modelling*. Tillgänglig på Internet: http://www.cg.its.tudelft.nl/Publications-new/2009/SDGTB09a/SDGTB09a.pdf

Statistiska centralbyrån. (2016). *Standardavvikelse och kvartiler*. Tillgänglig på internet: http://www.scb.se/sv_/Dokumentation/Statistikguiden/Rakna-ratt/Standardavvikelse-och-kvartiler/

Sällström, A. (2008). *Generering av höjdkartor Och dess användbarhet*. Tillgänglig på internet: *http://his.diva-portal.org/smash/get/diva2:113674/FULLTEXT01.pdf*

Unity Technologies (2015) Unity Engine (Version 5.3.2) [Datorprogram]. Unity Technologies. Tillgänglig på internet: https://unity3d.com/.

Wang, HR., Chen, WL., W., Liu, XL., Dong, B. (2010). *An improving algorithm for generating real sense terrain and parameter analysis based on fractal*. Proceedings of the Ninth International Conference on Machine Learning and Cybernetics.

Wikipedia. (2015). *Fraktal*. Tillgänglig på internet: https://sv.wikipedia.org/wiki/Fraktal

Wikipedia. (2007). *Terrain rendering*. Tillgänglig på internet: https://en.wikipedia.org/wiki/Terrain_rendering

Yannakaki, G.N., Togelius, J. (2011). *Experience-Driven Procedural Content Generation.* (Volume: 2, Issue: 3) s.147 - 161.

Yannakaki, G.N., Togelius, J. (2015). *Experience-Driven Procedural Content Generation.* International Conference on Affective Computing and Intelligent Interaction

Appendix A - Perlin noise pseudokod

```
function Noise1(integer x, integer y)
    n = (x * 57 % y;
    return(1.0 - ((n* (n *n * 15731 + 789221) + 1376312589) & 7fffffff) / 1073741824.0);
  end function
function Interpolate(a, b, x)
      return (a * (1 - x)) + b * x
  end of function
  function InterpolatedNoise_1(float x, float y)
     integer_X    = int(x)
     fractional_X = x - integer_X
    integer_Y    = int(y)
     fractional_Y = x - integer_Y
     v1 = Noise1(integer_X, integer_Y)
     v2 = Noise1(integer_X + 1, integer_Y )
     v2 = Noise1(integer_X , integer_Y +1)
     v2 = Noise1(integer_X + 1, integer_Y +1)
    i1 = Interpolate(v1 , v2 , fractional_X)
    i2 = Interpolate(v1 , v2 , fractional_X)
     return Interpolate(i1 , i2 , fractional_Y)
  end function
  function PerlinNoise_1D(float x, float y)
     total = 0
     p = persistence
     n = Number_Of_Octaves - 1
     loop i from 0 to n
        frequency = 2^i
        amplitude = p^i
```

total = **total** + **InterpolatedNoise**$_i$(**x** * **frequency**, **y** * **frequency**) * **amplitude**

end of **i** loop

return **total**

end function

Appendix B - Diamond-Square pseudokod

A
let avg = average of square corners step_size apart
let avg = average of four corners of average of four corners of diamond

```
function DiamondSquare(int xmin, int zmin, int xmax, int zmax, float maxrandomvalue,
int max stepsizelevel, int maxLevel, int currentLevel)
    {
        if (maxLevel < currentLevel) return;

        if (level < 1) return;

        // diamonds
        for (int i = z1 + level; i < z2; i += level)
        {
            for (int j = x1 + level; j < x2; j += level)
            {
                let avg = average of square corners step_size apart
                middle =  avg+ random value);
            }
        }

        // squares
        for (int j = x1 + 2 * level; j < x2; j += level)
            for (int i = z1 + 2 * level; i < z2; i += level)
            {
                    let avg = average of four corners of average of four corners of diamond
                    middle of top side = avg + random value
                    middle of left side = avg + random value
            }
        DiamondSquare(x1, z1, x2, z2, range / 2, level/2, maxLevel, currentLevel+1);
    }
```

Appendix C - Flood fill pseudokod

Function Floodfill

Set Q to the empty stack.

Set L to empty list

Add node to the top of Q.

While Q is not empty:

Set n equal to the top element of Q.

Remove first element from Q.

If the height of n to the surrounding nodes is within the limits:

Add west node to top of Q if west is not in list L.

Add east node to top of Q if east is not in list L.

Add north node to top of Q if north is not in list L.

Add south node to top of Q if south is not in list L.

End while

Return list size / max nodes.

End function