Examensarbete

HÖGSKOLAN I SKÖVDE

PROCEDURELLT GENERERADE DUNGEONKARTOR FÖR ROGUELIKESPEL

En jämförelse mellan Binary Space Partitioning och Delaunay Triangulation

PROCEDURALLY GENERATED DUNGEON MAPS FOR ROGUELIKE GAMES

A comparison between Binary Space Partitioning and Delaunay Triangulation

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2019

Oliver Karlsson

Handledare: Mikael Thieme
Examinator: Sanny Syberfeldt

Sammanfattning

Procedural Content Generation innebär att spelinnehåll automatiskt genereras för att både öka variationen i spel och minska arbetsbelastningen hos designers. Ett användningsområde för detta är rumbaserad bangenerering. Målet med detta arbete var att jämföra två algoritmer som gör just detta: Binary Space Partitioning och Delaunay Triangulation. De kriterier som algoritmerna utvärderades på var tidseffektivitet, variation, likhet och möjligheten att nå alla rum. Resultatet visade att Binary Space Partitioning hade snabbare genereringstid samtidigt som Delaunay Triangulation gav utvecklaren mer valmöjligheter. Vilken algoritm som var att föredra ifall tidsaspekten inte bar mest tyngd blev en mestadels subjektiv fråga där varje enskild utvecklares önskemål kommer påverka svaret. Ifall arbetet skulle fortsättas i framtiden skulle det vara intressant att utföra fler tester med flera olika mätvärden samt använda algoritmerna i ett spel och sedan låta testare bedöma kvalitén hos banorna som genereras.

Nyckelord: Procedural Content Generation, Binary Space Partitioning, Delauay triangulation, dungeonkartor, roguelike

Innehållsförteckning

1 Introduktion ... 1

2 Bakgrund ... 2

2.1 Procedural Content Generation ... 2

2.1.1 Online vs Offline ... 3

2.1.2 Egenskaper ... 3

2.1.3 Genereringsmetoder ... 3

2.1.4 Klassificering och styrning ... 4

2.1.5 Problem som PCG avser att lösa ... 4

2.2 Dungeon och generering av dungeon ... 5

2.2.1 Binary Space Partitioning ... 6

2.2.2 Delaunay Triangulation ... 8

3 Problemformulering ... 11

3.1 Metodbeskrivning ... 12

4 Implementation ... 15

4.1 Utvecklings- och experimentmiljö ... 15

4.2 Dungeonuppsättning och bit masking ... 15

4.3 Binary Space Partitioning ... 18

4.4 Delauney Triangulation ... 22

4.5 Utvärdering ... 26

4.5.1 Tidseffektivitet ... 26

4.5.2 Möjligheten att nå alla rum ... 26

4.5.3 Variation och likhet ... 26

5 Utvärdering ... 27

5.1 Presentation av undersökning ... 27

5.2 Resultat ... 28

5.2.1 Genereringstid ... 28

5.2.2 Genomsnittlig rumsstorlek ... 29

5.2.3 Genomsnittlig korridorlängd ... 30

5.2.4 Antal återvändsgränder ... 31

5.2.5 Antal icke nåbara rum ... 32

5.3 Analys ... 32

5.3.1 Genereringstid ... 32

5.3.2 Genomsnittlig rumsstorlek ... 33

5.3.3 Genomsnittlig korridorlängd ... 34

5.3.4 Antal återvändsgränder ... 34

5.4 Slutsatser ... 34

6 Avslutande diskussion ... 36

6.1 Sammanfattning ... 36

6.2 Diskussion ... 36

6.2.1 Trovärdighet ... 37

6.2.2 Samhälleliga och etiska aspekter ... 37

6.3 Framtida arbeten ... 38

Referenser ... 40

1 Introduktion

Procedural Content Generation (framöver förkortas detta till PCG), eller procedurellt genererat spelinnehåll, är en teknik som utvecklades inom spelindustrin för att framställa olika former av spelinnehåll. PCG är en metod som utvecklades för att motarbeta den ökade arbetsbelastningen, hos framförallt speldesigners och grafiker, som växte i takt med att spelindustrin utvecklades. PCG kan användas för att generera flera olika former av spelinnehåll, såsom miljöer, spelregler, kartor, musik och story. Av den anledningen kan PCG användas i många olika genrer och några spel som använder någon sorts PCG är Minecraft (Mojang, 2011), Spelunky (Mossmouth, 2008) och Spore (Maxis, 2008).

En sorts innehåll som genereras med hjälp av PCG är dungeons, där en dungeon ofta definieras som en rumsstruktur i form av fängelsehåla eller labyrintliknande. Det finns många olika algoritmer för generering av dungeons där man får olika resultat, men en populär version som används i spel som TinyKeep (Phigames, 2014) och Neon Chrome (10tons Ltd, 2016) går ut på att banor genereras helt slumpmässigt för att ge så stor mångfald som möjligt. Det är denna metod som arbetet fokuserade på.

Syftet med det här arbetet var att implementera och jämföra två olika algoritmer för slumpmässig rumbaserad bangenerering som kunde användas under körning av ett spel. Det fanns därmed krav på att algoritmerna skulle vara tillräckligt tidseffektiva och att de genererade banorna skulle vara av acceptabel kvalité. Även om algoritmernas uppbyggnad skiljer sig från varandra så är resultaten av liknande karaktär. Vidare kan de banor som algoritmerna genererar användas i samma spel där den sortens dungeon används.

För att utvärdera algoritmerna så jämfördes fyra olika aspekter: tidseffektivitet, likhet av resultat, variation av resultat och möjligheten att nå alla rum. För att mäta tidseffektivitet på algoritmerna användes hårdvaran i datorn för att mäta hur lång tid det tog att generera en dungeon. För att jämföra likhet och variation hos resultaten så utvärderades vardera algoritm baserat på ett antal utvalda aspekter. Dessa aspekter var specifikt rumsdimensioner, genomsnittlig korridorlängd och antal återvändsgränder. Avslutningsvis så implementerades en enkel kontroll för att säkerställa att alla rum går att nås.

2 Bakgrund

Följande kapitel innehåller en översiktlig genomgång till bakgrunden av frågeställningen där Procedural Content Generation är huvudfokus. Först beskrivs PCG överlag för att sedan skifta fokus till specifika typer av generering. Kommande underkapitel består av algoritmer för generering av dungeons. Avslutningsvis så redovisas vilka algoritmer som valdes samt tidigare forskning av området.

2.1 Procedural Content Generation

Procedural Content Generation (PCG) är ett algoritmiskt skapande av spelinnehåll med begränsad eller indirekt användarinmatning (Togelius, Kastbjerg, Schedl & Yannakakis, 2011). PCG refererar alltså till datormjukvara som kan skapa spelinnehåll på egen hand, eller tillsammans med en eller flera mänskliga spelare. Termen "innehåll" i PCG är ett nyckelord här. I detta arbete refererar innehåll till i stort sett allt som kan existera i spel. Några exempel skulle vara kartor, spelregler, musik och banor (Shaker, Nelson & Togelius, 2016).

PCG är dock ingen perfekt metod. På grund av faktorer som kvalitetssäkring och brist på kontroll så används PCG sällan till generering av fullständigt spelinnehåll i större kommersiella spel (Short & Adams, 2017). I det här fallet syftar kvalitetssäkring och kontroll på spelutvecklares förmåga att garantera spelinnehållets underhållningsvärde och utseende när det genereras av en algoritm istället för när innehållet skapats för hand. Trots detta finns det flertalet kända spel som använder PCG. Några exempel är Minecraft, se Figur 1, (Mojang, 2011) där PCG tekniker används för att generera hela spelvärldar, Spelunky (Mossmouth, 2008) där PCG tekniker skapar variationer av liknande spelbanor och Spore (Maxis, 2008) där designen skapad av spelarna animeras genom procedurella animationstekniker (Shaker m.fl., 2016).

Figur 1: En skärmdump från spelet Minecraft.

2.1.1 Online vs Offline

Online och offline är de två huvudsakliga sätt att generera spelinnehåll på. Online betyder att spelinnehåll genereras samtidigt som spelet spelas. Detta tillåter genereringen av ändlös variation vilket kan ge spelet oändligt med återspelningsvärde. Online skapar även möjligheten till att generera spelinnehåll anpassat till hur spelaren valt att ta beslut under spelets gång. Offline däremot sker under utvecklingen av spelet eller mellan spelsessioner som då även innefattar innan spelet startas. Det stora användningsområdet för offline PCG är exempelvis att generera större komplext spelinnehåll såsom miljö och vegetation (Shaker m.fl., 2016).

Ett bra exempel på online PCG kommer från spelet Left 4 Dead (Valve Corporation, 2008) där spelets motståndare anpassas efter spelaren. Spelarens beteende analyseras i realtid och spelets tillstånd ändras i samband med det genom PCG tekniker (Booth, 2009). Ett bra exempel på offline PCG kommer från racingspelet Forza Motorsport (Microsoft, 2005) där så kallade Non-Player Characters (NPCs) tränas offline för att imitera spelarens körsätt för att senare kunna användas för att köra åt spelaren (Shaker m.fl., 2016). NPC:s är icke-spelar karaktärer som spelaren inte kan ta kontroll över och inte står som motståndare i spelet. De finns främst till för att driva spelaren framåt i spelet, hjälpa spelaren eller ge mer liv i spelet.

2.1.2 Egenskaper

Trots nackdelarna med PCG så finns det flertalet kommersiella spel som använder tekniken. Detta syftar främst på delvis genererat spelinnehåll, och inte fullständigt. Anledningen till att PCG används är på grund av dess många arbetsområden och dess fördelar. Den första fördelen med PCG är minnesanvändning. Eftersom procedurellt representerat innehåll vanligtvis kan komprimeras och expanderas när det behövs (Browne, Stanley, Togelius & Yannakakis, 2011). Därför behöver inte spel som Spelunky (Mossmouth, 2008) lagra sina banor från start utan kan istället genererar dem när de behövs. Med andra ord räcker det att lagra de delar som krävs för att generera banorna, istället för att generera hela banorna direkt. Detta leder det till mindre minnesanvändning och är därför en fördel för PCG.

En annan fördel med PCG är de minskade kostnaderna gentemot att manuellt skapa spelinnehåll. Många stora företag använder metoder för att generera enorma områden av vegetation baserat på bara ett fåtal parametrar. Detta leder till att det sparas stora mängder resurser och hjälper vid utveckling av spel med stora, öppna världar (Browne m.fl., 2011). Ett sista argument för PCG är att det kan stå för skapandet av helt nya spel och genrer. Offline algoritmer kan bland annat skapa nya regeluppsättningar, banor och berättelser. Dessa kan senare komma att inspirera speldesigners och grunda basen för nya skapelser. På liknande sätt kan algoritmer skapa nya spel genom spelmekaniker som bygger på generering av spelinnehåll. Exempelvis så finns det en vision för PCG där ett spel kontinuerligt utvecklar sig själv för att alltid ha återspelningsvärde. Det vill säga ett spel som i praktiken skulle vara helt oändligt. PCG tekniken har inte nått denna punkt ännu men det är förhoppningar som denna som motiverar till att undersöka och utveckla området (Browne m.fl., 2011).

2.1.3 Genereringsmetoder

PCG algoritmer kategoriseras in i två olika grupper som på engelska kallas *constructive* och *generate-and-test*. I constructive PCG så genereras spelinnehåll en gång och avslutar exekveringen så fort algoritmen verifierat att innehållet är användbart. I generate-and-test däremot så genereras spelinnehåll som sedan testas. Efter testningen så växlar algoritmens status mellan generering och testning tills ett tillfredsställande resultat har nåtts (Shaker m.fl.,

2016). Den märkbara skillnaden är körtiden för algoritmerna. En constructive algoritm som körs en gång kan testas fram till en genomsnittlig körtid. I generate-and-test är det svårare att säga när den når ett tillfredsställande resultat och därför svårt att avgöra om körtiden blir lång eller kort. Dock eftersom constructive bara körs en gång och generate-and-test loopar tills den når ett bra tillstånd så blir körtiden för constructive till stor majoritet kortare än generate-and-test. Detta beror dock på hur den algoritmen implementerats.

2.1.4 Klassificering och styrning

Slutligen finns det två viktiga ämnen inom PCG att ta upp - klassificering och styrning. Med klassificering menas vilken typ av spelinnehåll som genereras, vilket delas in i två olika grupper: nödvändigt och valfritt spelinnehåll. Nödvändigt spelinnehåll är innehåll i spelet som måste användas av spelaren för att fortsätta i spelet. Detta betyder att nödvändigt spelinnehåll måste fungera korrekt för att spelet ska kunna användas. Valfritt spelinnehåll däremot är innehåll som spelaren inte måste interagera med för att avsluta spelsessionen. Det leder till att detta spelinnehåll inte har samma krav som det hade haft om det varit helt nödvändigt för spelet att fungera (Shaker m.fl., 2016). Styrning syftar till graden av kontroll som en algoritm har när den genererar spelinnehåll. Att en algoritm har kontroll betyder att en speldesigner har mindre kontroll. Vem som har mest kontroll bestäms genom hur mycket av algoritmen som styrs av parametrar valda av spelutvecklaren. Ju mer parametrar som kan modifieras av en utvecklare, desto mindre kontroll har algoritmen över resultatet (Shaker m.fl., 2016).

2.1.5 Problem som PCG avser att lösa

PCG är en metod som skapades för att lösa ett antal problem som ofta stöts på under spelutveckling. Följer gör en lista på fem av dom viktigaste problemen som PCG försöker lösa (Shaker m.fl, 2016).

- **Tidsaspekten:** Krav på hastighet varierar baserat på vad man vill åstadkomma med PCG. Allt ifrån millisekunder till månader kan vara acceptabelt. Detta beror främst på om genereringen sker under en spelsession eller under utvecklingen av spelet. Att välja rätt algoritm i rätt läge kan vara avgörande för utvecklingsprocessen. Likaså kan valet av algoritm också avgöra om genereringstiden är av acceptabel längd.

- **Pålitlighet:** Genereringsalgoritmer kan ge allt ifrån helt slumpmässiga till helt förväntade resultat. Detta blir viktigare för vissa sorters PCG än andra att det som genereras når acceptabel kvalité eller inte. Exempelvis så kan en dungeon utan början eller slut vara helt oanvändbart samtidigt som en blomma som ser lite udda ut knappt spelar någon roll.

- **Styrbarhet:** Det är ofta nödvändigt för genereringsalgoritmer att kunna styras på något sätt. Spelutvecklare kan ha ett väldigt specifikt syfte en PCG algoritm. Då är det viktigt att algoritmen anpassas för att garantera att resultatet uppfyller det syftet.

- **Mångfald:** Det kan finnas ett behov av att generera en mångfald av samma spelinnehåll. Detta för att undvika att olika spelinnehåll inte är för likt en annan. En algoritm skulle kunna generera banor som ser exakt likadana ut. Samtidigt skulle en annan algoritm kunna generera spelinnehåll som aldrig upprepar sig. Helt slumpmässiga banor kan dock leda till icke spelbart spelinnehåll och samtidigt kan

helt styrd generering ge ointressanta resultat. Mångfald blir på så vis en balans mellan slumpmässighet och styrning.

- **Kreativitet och trovärdighet:** PCG innehåll ska helst inte se ut som att det designats av en PCG algoritm. Det finns många tecken på att spelinnehåll har genererats av en algoritm istället för av en människa. Detta är något som man får motsträva på andra sätt men kan även styras av implementationen av algoritmen (Shaker m.fl., 2016).

2.2 Dungeon och generering av dungeon

Dungeon är en term som används i många RPG- och äventyrsspel som har en annan betydelse i spelvärlden än i vardagen. Den svenska översättningen är fängelsehåla men en dungeon i spelvärlden kan vara så mycket mer än det. En dungeon i spelvärlden är snarare en stängd plats där spelaren främst går in för att besegra motståndare och få belöningar. En dungeon kan ta många former och är inte avgränsad till en specifik miljö. Exempelvis så kan en dungeon vara ett antal rum ihopkopplade av korridorer som en dungeon i spelet Diablo III (Blizzard Entertainment, 2012) vid namnet *Depths of the Keep*, det kan vara ett tempel som en dungeon i spelet World of Warcraft, se Figur 2, (Blizzard Entertainment, 2004) vid namnet *Temple of the Jade Serpent*, en kloak som en dungeon i spelet Realm of the Mad God (Wild Shadow Studios, 2011) vid namnet *Toxic Sewers* eller ett grottsystem som en dungeon i spelet Torchlight (Runic Games, 2009) vid namnet *Tu'Tara Caverns*. Det finns en rad av exempel som skulle bli näst intill oändligt lång om alla skulle tas med. Det finns inte en exakt avgränsning för spelvärldens dungeon. Därför syftar ordet dungeon i detta arbeta till en miljö som har en början och ett slut med eller utan hinder på vägen. Dungeons fylls med flertalet objekt för att skapa underhållningsvärde för spelaren. Detta kan vara allt från monster till motståndare, skattkistor, vapen, dekorationer och fällor.

Figur 2: En skärmdump från dungeonen *Temple of the Jade Serpent* i spelet World of Warcraft.

Även om dungeons nästan aldrig är tomma så inkluderas inte objekt i begreppet dungeon-generering. Generering av dungeons syftar istället främst till skapandet av rumsstrukturen

och nödvändigt spelinnehåll (van der Linden, Lopes & Bidarra, 2014). Bangenerering som denna användes i spel som Diablo (Blizzard North. 1996) och Rogue (Artificial Intelligence Design, 1980).

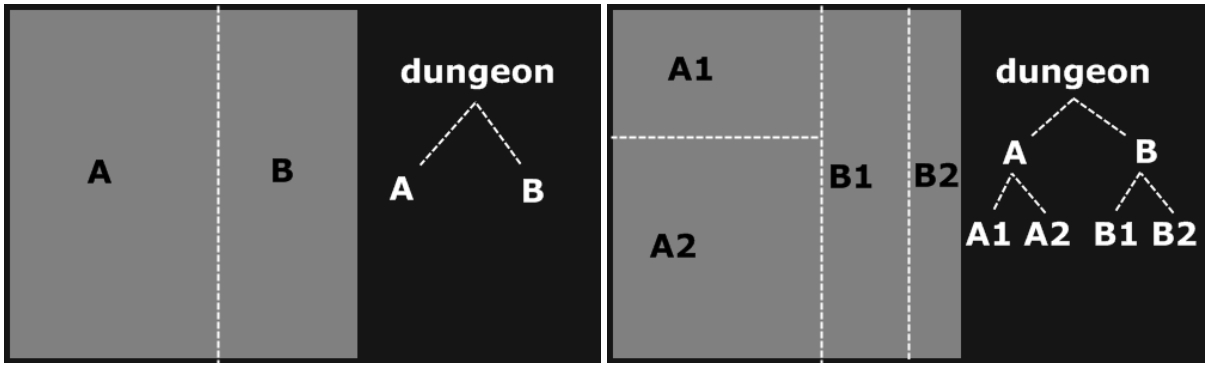
Att slumpmässigt generera rum och korridorer under genereringen är en undergrupp av algoritmer. Spelen nämnda ovan använder äldre exempel på denna metod. Metoden lever dock kvar och flertalet spel har på senare tiden använt liknande metoder, exempelvis Neon Chrome, se Figur 3, (10tons Ltd., 2016) och TinyKeep (Phigames, 2014). Det är denna form av bangenerering som detta arbete fokuserat på.

Figur 3: En skärmdump från spelet Neon Chrome.

Det finns flera algoritmer som används för rumbaserad dungeongenerering men det här arbetet fokuserar på två av dem. Båda algoritmerna har varsin genomgång som följer nedan. Avgränsningen till dessa två algoritmer baserades på ett fåtal nyckelfaktorer. Gemensamt för algoritmerna är att de kan använda slumpmässigt genererade rumsstorlekar och korridorer. Båda algoritmerna klassificeras som constructive algoritmer. Andra aspekter som algoritmerna har gemensamt är att all generering sker online, de är styrbara och allt spelinnehåll som genereras av algoritmerna ses som nödvändigt.

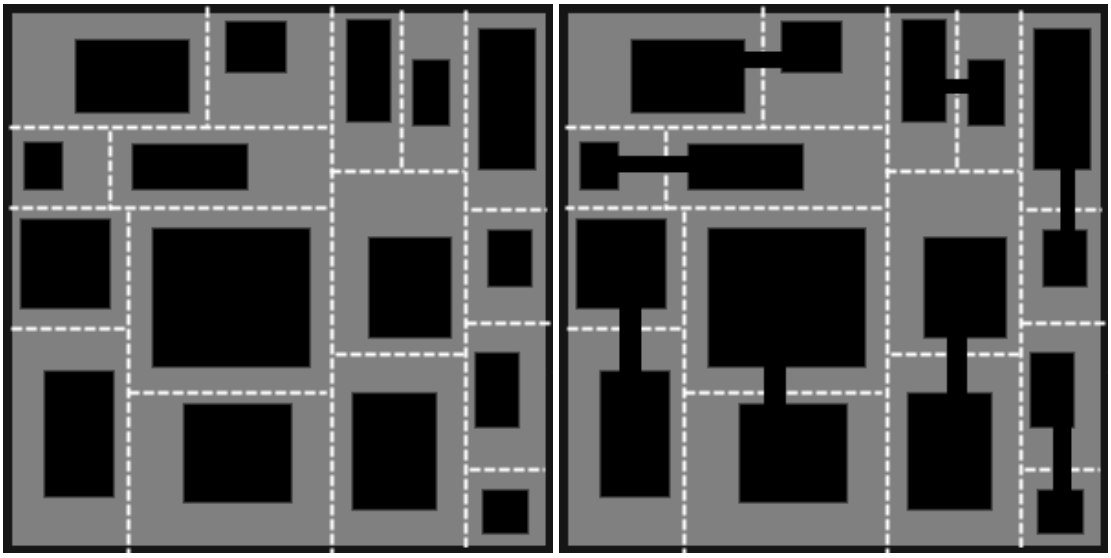
2.2.1 Binary Space Partitioning

Binary Space Partitioning (framöver förkortas detta till BSP) är en algoritm som rekursivt delar upp ett stort område i flera mindre områden. Processen lagras i ett så kallat BSP-träd där varje löv i trädet representerar ett av de mindre områdena. Rotnoden i BSP-trädet representerar hela startytan. Uppdelningen av områdena kan ske i höjdled eller sidled och de mindre områdena kan vara av olika storlek. För att få olika storlekar på rummen på de mindre områdena, slumpas en startposition i höjd- eller sidled. För att styra algoritmen kan flera parametrar sättas ifall målet är en mer specifik form av dungeon. Exempelvis så brukar en minimum storlek på rum sättas för att inte generera för små rum, se Figur 4, (Shaker m.fl., 2016).

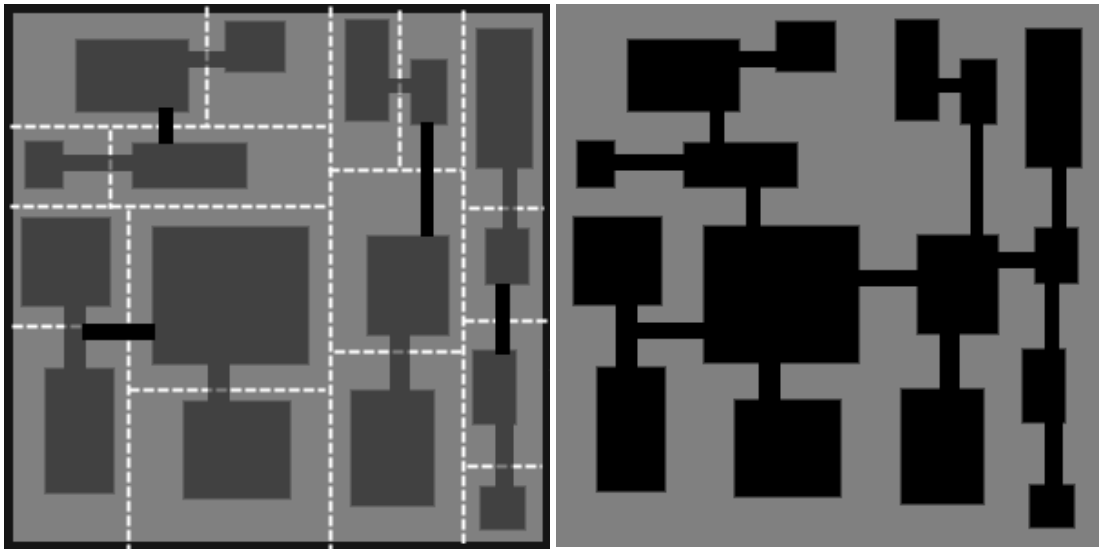


Figur 4: Startutrymmet delas upp i två delar, A och B, som sedan i sin tur delas upp i två delar, A1 och A2 respektive B1 och B2. Varje fortsatt uppdelning resulterar i två nya noder.

När uppdelningen är klar så placeras rum ut i de mindre områdena. Ett rum placeras i varje område med begränsningen att rummet ska få plats i området. När varje område fått ett rum kopplas rummen samman med korridorer. Genom att rekursivt gå igenom BSP-trädet och koppla ihop barnnoder med samma föräldranoder skapas en väg genom hela banan, se Figur 5. Algoritmen fortsätter med att koppla samman föräldranoder och rekursivt fortsätter det här tills rotnoden har nåtts, se Figur 6, (Shaker m.fl., 2016).



Figur 5: I bilden till vänster så syns hur rummen placerats ut i områdena. I bilden till höger så har syskonnoder kopplats samman med korridorer.



Figur 6: I bilden till vänster så syns föräldranoder som är syskon kopplas samman. I bilden till höger så ser man hur den färdiga kartan ser ut.

2.2.2 Delaunay Triangulation

TinyKeep (PhiGames, 2014) är ett spel där man går genom dungeon efter dungeon och klarar av nivå efter nivå. Spelet procedurellt genererar banor när spelaren klarat av en dungeon. Spelets utvecklare skapade en ny teknik för att generera dungeons i spelet. Utvecklaren Phi Dinh gick senare ut med hur tekniken fungerade och förklarade i detalj hur algoritmen genererar dungeons. Tekniken använder Delaunay Triangulation (framöver förkortas detta till DT) och Park-Miller Normal Distribution (framöver förkortas detta till PMND) för att generera dungeons (Phi Dinh, 2013).

PMND är en form av normalfördelning som anpassades från Park och Millers artikel om slumpgenerering. Artikeln föreslår en form av *Linear Congruential Generator* (framöver förkortas detta till LCG) med specifika parametrar valda (Park & Miller, 1988). LCG är en av de äldsta, simplaste och vanligaste pseudo slumpmässiga nummer generatorerna. Den beräknar varje nummer baserat på ett föregående värde, vilket startas från ett så kallat *seed*. Ett seed är ett startvärde som kan användas igen för att återskapa samma sekvens av slumpade värden (Dwyer & Williams, 1999). Användningen av PMND är ett val av utvecklarna för TinyKeep för att skapa den estetiska upplevelse de sökte. I detta arbete är inte den estetiska upplevelsen lika intressant och därför faller ämnet PMND utanför arbetets omfång.

Algoritmen börjar med att skapa ett antal celler och en rektangel med slumpmässig storlek i varje cell. Sedan används PMND istället för vanlig normalfördelning på rektanglarnas slumpmässighet. PMND förvränger storleken på cellerna så dom har större sannolikhet att vara av mindre storlek (Phi Dinh, 2013). Anledningen till att använda PMND är helt enkelt för att få mer kontroll över storlekarna på cellerna som skapas. För en vanlig implementation av Delaunay Triangulation så kan man skapa rum av samma eller olika storlek, och därav är normalfördelning inget som måste användas.

Figur 7: Rektanglar skapade i varje cell.

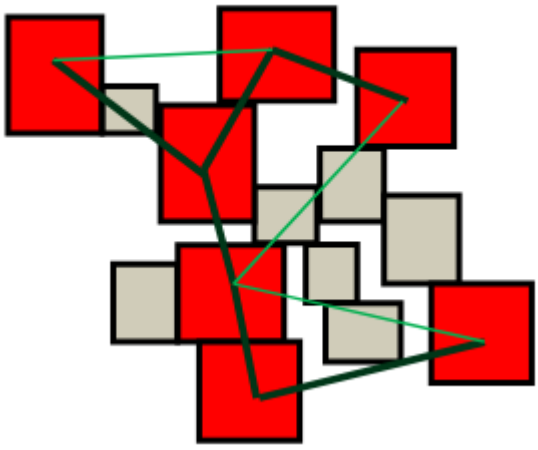
Nästa steg är att separera rektanglarna från varandra då de i nuläget överlappar, se Figur 7. Rummen separeras genom ett separationsstyrningsbeteende som baseras på den omvända kvadratlagen tills dess att inga rektanglar överlappar varandra. Den här metoden ser till att absolut inga rektanglar överlappar varandra, samtidigt som den försöker packa dem så tätt som möjligt. Efter styrningsbeteendet har separerat cellerna så omvandlas ett antal celler till rum baserat på satta kriterium, se Figur 8. I TinyKeep så är detta kriteriet att rummen ska vara över en viss storlek (Phi Dinh, 2013).

Figur 8: Cellerna utspridda efter styrningsbeteendet. De röda cellerna representerar utvalda rum.

De tomma vita områdena mellan cellerna fylls med 1*1 stora tomma celler. Utvecklarna använde sedan DT för att konstruera en graf över alla rummens mittpunkter. Grafen som genereras av DT har ett stort antal kopplingar mellan rummen. För att generera en graf som är bättre lämpad för en dungeon så används en algoritm för att hitta ett så kallat minimum spanning tree (MST). I TinyKeep så används Prims algoritm för detta (Phi Dinh, 2013).

Dock så kommer ett MST som producerats av Prims algoritm ta bort alla intressanta attribut från grafen, såsom cykler. För att behålla den mer intressanta dungeon uppbyggnaden så

skapas en tredje graf genom att kombinera båda graferna, se Figur 9. I TinyKeep så sparades runt 15 procent av DT grafen i den slutgiltiga grafen (Phi Dinh, 2013).



Figur 9: De mörkgröna kopplingarna kommer från MST grafen. De ljusgröna kopplingarna är sparade kopplingar från DT grafen.

Rummen kopplas sedan samman med korridorer med hjälp av den tredje grafen. Antingen placeras raka eller L-formade korridorer för att hålla korridorerna raka (Phi Dinh, 2013).

3 Problemformulering

I detta arbete implementerades och jämfördes två algoritmer för rumbaserad bangenerering. Algoritmerna var Binary Space Partitioning och Delaunay Triangulation. Arbetets mål var att avgöra vilken av algoritmerna som är mer tidseffektiv, att jämföra likheten hos resultaten från genererings-algoritmerna, att utvärdera variationen på banorna som algoritmerna skapar samt att utvärdera möjligheten att nå alla rum. Utöver det så mättes kvalitén genom de mätvärden som kunde nås med programmering. Mer specifikt så inkluderar det här två viktiga aspekter: den genomsnittliga längden på korridorerna och hur många rum som är återvändsgränder.

Valet av algoritmerna grundade sig i att jämföra tekniker vars resultat väl liknar varandra för att hitta viktiga skillnader. Båda algoritmerna är rumbaserade och genererar både rum och korridorer, båda är konstruktiva och båda klarar av att generera banor online. Vidare har båda algoritmerna implementerats i kommersiella spel och har därför existerande exempel. Det mest intressanta med algoritmerna är att de slumpmässigt genererar banor med hög mångfald. De används för att generera rum av slumpmässig storlek, som kopplas ihop med rum på slumpmässigt avstånd från varandra, vilket ger banorna hög mångfald. Detta arbete fokuserade därmed på algoritmer som behandlade slumpmässiga uppsättningar av specifikt rum och korridorer. Med andra ord är nyckelorden här *slumpmässigt genererade dungeons av klassisk uppsättning* och är anpassade för spelgenres som använder dessa, så kallade *dungeoncrawlers*.

Anledningen till att jämföra algoritmer som har liknande resultat är att avgöra vilken metod som är att föredra i vilken situation. Meningen är då att hitta situationer där en algoritm skulle vara att föredra över en annan. Vidare är DT en algoritm som inte fått lika mycket uppmärksamhet som andra algoritmer. Detta leder till att algoritmen inte utvärderats till den grad den borde och därför kan denna jämförelse komma att vara viktig för området. Syftet var således att tillföra mer forskning till området för att lägga en starkare grund för framtida spelutvecklare som är intresserade i algoritmerna.

Med tidseffektivitet menas hur lång tid genereringen tar. Som tidigare nämnt så kan tiden variera mycket beroende på vilken typ av generering som används. I detta arbete har generering skett online och då mäts ofta genereringstiden i millisekunder. En studie hade undersökt den acceptabla väntetiden på en så kallad *loading screen* och kom fram till ungefär två sekunder. Mer exakt så handlade det om en svart skärm som loading screen som varade i exakt 2182 millisekunder, men i detta arbete avrundades gränsen till ungefär två sekunder. Detta arbete har därav förhållit sig till ungefär samma värde (Zhao, Ge, Qu, Zhang & Sun, 2017).

Det viktiga med att jämföra likheten hos resultat mellan algoritmerna kom från att arbetet avsåg att hitta tillfällen där man bör välja den ena algoritmen över den andra. Ifall resultaten hos algoritmerna skiljer sig för mycket från varandra kan man av estetiska anledningar välja den algoritm som har lägre tidseffektivitet än den andra. Meningen var alltså att få ett mer övergripande resultat som tog hänsyn till mer än enbart tidsaspekten, till exempel speldesignaspekter.

Liknande argument fanns för variationen inbördes mellan banorna som algoritmerna genererar. Kort sagt så handlade det om resultatens mångfald. Denna jämförelse skedde

inbördes hos algoritmerna, det vill säga en algoritm jämförde sina egna resultat med varandra. Resultaten av detta jämfördes sedan mellan algoritmerna för att hitta viktiga skillnader. Även denna jämförelse var mer riktad mot speldesign än programmeringen.

Avslutningsvis så undersöktes möjligheten att nå alla rum. Detta var riktat mot problemområdet pålitlighet som nämndes tidigare. Detta borde inte kunna vara ett problem då utformningen av algoritmerna bör göra det omöjligt för detta att hända, dock testades det fortfarande för att verifiera att algoritmerna implementerades korrekt.

För att säkerställa att banorna som ska genereras var av god kvalité så hade det varit önskvärt med fler mätvärden, dock blev detta svårt att realisera. Eftersom arbetet utfördes ur ett programmeringsperspektiv så gav även detta ett konstigt fokus. Mått på kvalitén hos automatiskt genererade banor är subjektiva, vilket inte går att mäta genom programmering. Dock betydde inte detta att denna aspekt ignorerades helt. Ur ett kvalitetsperspektiv så är både möjligheten att nå alla rum och variationen på banorna intressanta aspekter. Vidare så mättes kvalitén genom de två aspekterna som nämndes tidigare: genomsnittlig korridorlängd och antalet återvändsgränder. Vare sig resultaten i dessa två aspekter är bra eller dåliga är subjektivt, men skillnaden mellan algoritmerna var fortfarande intressant. På vilket sätt jämförelsen och implementationen gick till beskrivs i metodbeskrivningen.

3.1 Metodbeskrivning

Som tidigare nämnt så refererade dungeongenerering främst till rumsstrukturen och spelspecifika objekt. Algoritmerna har därför implementeras baserats på detta, för att se konkreta skillnader i enbart genereringen av dungeonmiljöerna. Detta uppnåddes genom att inte skapa motståndare eller spelspecifika objekt i banorna, med andra ord skapades det enbart tomma rum sammankopplades med tomma korridorer. Denna undersökning syftade att hitta skillnader mellan algoritmerna för generering av dungeonmiljöer, och inte kompletta banor. I syfte att utvärderingen av algoritmerna skulle vara så rättvis som möjligt lades stor vikt på att det skulle gå att ställa in relevanta parametrar. Exempelvis så skapades en parameter som styrde antalet delningar av området för algoritmen BSP. Detta gjordes för att i algoritmen DT kunde ett exakt antal rum väljas. Med en parameter för antal delningar i algoritmen BSP kunde det garanteras att båda algoritmerna skapade banor med samma antal rum. Målet med parametrarna var att de olika algoritmerna skulle kunna anpassas för att få så lika förutsättningar som möjligt. Har algoritmerna inte samma grund så kommer de inte kunna användas som substitut mot varandra, av den anledningen behövde förutsättningarna jämnas ut.

Som tidigare nämnt så utvärderades algoritmerna baserat på fyra olika kriterier: tidseffektivitet, möjligheten att nå alla rum, likhet mellan resultaten av algoritmerna och variation på resultaten inbördes. Den viktigaste aspekten här brukar bli möjligheten att nå alla rum. Detta kommer från problemområdet pålitlighet, och inom den genren som arbetet riktats mot så skulle ett icke-nåbart rum kunna resultera i en oanvändbar bana. Därför blev denna aspekt viktigast att ha i åtanke i detta arbete. Nåbarheten var dock en av grunderna för båda algoritmerna som är i fokus och som nämndes tidigare så skulle en korrekt implementation aldrig att generera en bana med rum som inte går att nå. På grund av detta blev nåbarhet bara mer av en säkerhetskontroll i denna studie. För att mäta tidseffektivitet så mättes tiden det tar för algoritmen att generera en bana. Eftersom det fanns en risk för olika resultat mellan varje generering beroende på i vilken ordning operativsystemet valde att exekvera processer, så togs

flera mätvärden på varje dungeon-generering. Efter att ett större antal genereringar hade utförts på samma dungeon så kunde ett genomsnitt mellan exekveringstiden räknas ut. Eftersom förmågan att välja exakt antal rum fanns hos DT men inte BSP, så anpassades antalet rum till BSP:s förutsättningar. Antalet delningar sattes därför före generering av BSP och sedan sattes antalet rum till samma antal före generering av DT.

Den största anledningen för att jämföra tidseffektivitet var att undvika moment där spelaren tvingas vänta. Dock kommer en spelare troligtvis inte irriteras över att behöva vänta några sekunder mellan varje dungeon som spelas, med tanke på att i stort sett alla spel har laddningsskärmar vid något tillfälle. Alltså blev den exakta tiden det tar att ladda en bana inte så intressant så länge som den var under gränsen för acceptabel väntetid. På liknande sätt blev inte för långa väntetider heller speciellt intressanta. Detta eftersom en spelare troligtvis inte blir mer irriterad om det tar tre minuter än fem, det är oavsett för lång tid att vänta. Som nämndes tidigare ha en gräns för väntetiden på ungefär två sekunder, vilket presenterades av den nämnda studien som undersökt ämnet (Zhao m.fl., 2017). Algoritmernas resultat var av acceptabel kvalité enbart om genereringstiden låg under den gränsen. Dock blev det inte lika viktigt att diskutera varken det bästa eller sämsta resultatet. Istället behövde diskussionen kring tidseffektivitet vara fokuserad på hur ofta den bästa tiden nås och hur bra den faktiskt är. En spelare kommer troligtvis inte märka några hundradelars skillnad så länge som genereringen sker på acceptabel tid. I dessa fall blev det mer intressant att diskutera kvalitén hos banorna som genererats och jämföra den aspekten.

Att mäta möjligheten för att nå alla rum skedde genom en enklare kontroll för vardera algoritm. Hur detta gick till skiljde sig mellan algoritmerna eftersom de genererade banor på olika sätt. Som nämndes tidigare kommer en korrekt implementation av algoritmerna aldrig generera icke-nåbara rum. Dock var det viktigt att avgöra ifall det här stämde för arbetets genereringsalgoritmer. Av den anledningen implementerades den enklare kontrollen.

Vidare så skulle mätning av likhet och variation att ta plats. Som nämndes tidigare kommer banorna att genereras med samma mängd rum så den aspekten berörs inte. Likheten syftade till hur lika de två algoritmernas genererade banor med samma mängd rum var. Eftersom algoritmerna genererade banor på olika sätt blev en jämförelse av utseendet och kvalitén subjektivt. Av den anledningen strukturerades undersökningen med detta i åtanke, för att kunna samla objektiv data och att göra jämförelsen så icke-subjektiv som möjligt.

Mätningen av likhet gick ut på att jämföra hur många rum av samma, eller likartade, dimension som algoritmerna genererade. I mätningen av variation däremot så jämfördes samma data fast inbördes i algoritmerna för sig. Varje algoritm genererade flera banor med samma antal rum, som sedan jämfördes mot varandra för att mäta hur stor variation varje algoritm bidrog med. Vidare som tidigare nämnt så jämfördes genomsnittlig korridorlängd och antalet återvändsgränder. Antalet genererade återvändsgränder beräknades från antalet rum som var ihopkopplat med bara ett annat rum. Korridorlängden räknades ut under generering och ett genomsnitt beräknades när genereringen är klar.

Variation användes som mätvärde i en tidigare studie (Evertsson, 2014). För att mäta variation tog studien upp två mätvärden för variation: antal rum och rumsvariation. Eftersom antalet rum sattes till bestämda värden i detta arbete kunde inte det mätvärdet användas. Rumsvariation i den nämnda studien baserades på vilka av de fördefinierade rummen, vilka användes i studien, som skapades av algoritmen. Eftersom rumsdimensioner slumpades fram i detta arbete kunde inte heller detta mätvärde användas. Av den anledningen togs istället

andra mätvärden fram i detta arbete: genomsnittlig korridorlängd, antal återvändsgränder och rumsdimensioner.

I en perfekt situation hade det funnits exakta mätvärden för att mäta kvalitén av en bana, men eftersom detta är ett subjektivt ämne så går inte det. Det fanns förhoppningar att lösa problemen med att mäta utseendet och kvalitén, dock finns det inget sätt att göra det utan att samla in subjektiva åsikter från exempelvis en speltestning. Metoden som användes i detta arbete är inte på något sätt en perfekt lösning för att mäta utseendet och kvalitén på banorna. Trots detta fanns förhoppningar om att det skulle gå att säga något om hur lämpliga algoritmerna är i olika situationer. Detta skede genom en jämförelse av aspekter som är kopplade till utseendet och kvalitén. Som nämndes tidigare var dessa tre i antal: korridorlängder, antal återvändsgränder och rumsdimensioner.

Experiment i form av simulering (Zelkowitz & Wallace, 1998) valdes som metod för att besvara frågeställningen då metoden konstaterades vara bäst lämpad för arbetet. En fallstudie hade kunnat väljas ifall undersökningens fokus hade varit på subjektiva egenskaper, exempelvis utseende och kvalité. Men eftersom denna studie istället främst fokuserar på objektiva egenskaperna, där en fallstudie potentiellt skulle kunna bidra med felkällor, så valdes denna metod bort. En annan fördel med experiment var att de är lätta att återskapa, vilket är en fördel i denna studie då genereringarna ska exekveras flera gånger. Att experiment användes gav också en fördel vid jämförelsen då datan som samlades in var kvantitativ, vilket stödjer jämförelser och statistiska analyser (Wohlin, Runeson, Höst, Ohlsson, Regnell & Wesslén, 2012). Ytterligare argument för metoden var att i stora delar av källmaterial och tidigare studier var experimentbaserade. Såväl kandidat arbeten (Andersson, 2013; Evertsson, 2014; Helge, 2018) som forskningsstudier (Sanamaría-Ibirika, Cantero, Salazar, Devesa, Santos, Huerta & Bringas, 2013; Luo, Yin, Cai, Lees, Othman & Zhou, 2014) använde samma metod. För att detta arbete och resultatet från arbetet ska vara en del av forskningsområdet så borde metoden vara anpassad efter området. Genom att göra arbetet experimentbaserat så anpassas undersökningen till områdets tidigare forskning och försöker besvara frågor relaterade till liknande synpunkter.

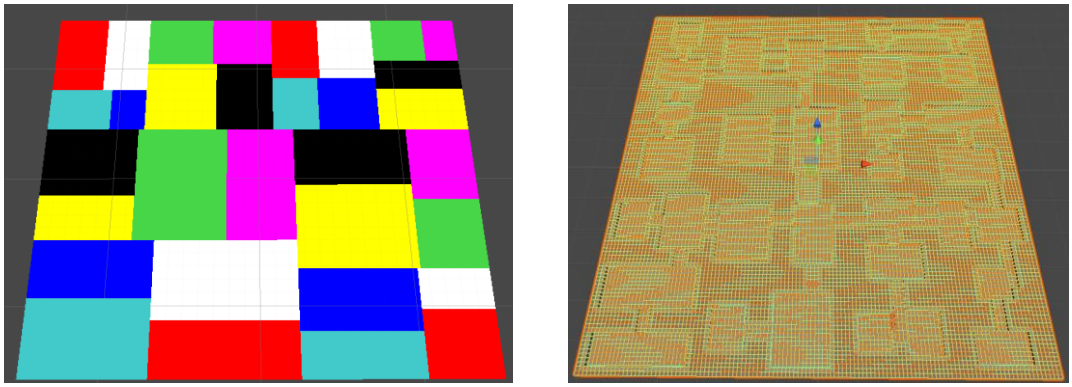
4 Implementation

I detta kapitel följer en genomgång av implementationen av de två algoritmerna och utvärderingarna för dessa. Först kommer dungeonuppsättningen och så kallade *bit masking* presenteras vilket sedan följs av en genomgång av implementationen för vardera algoritmen. Avslutningsvis presenteras implementationen av utvärderingarna.

4.1 Utvecklings- och experimentmiljö

Artefakten utvecklas i spelmotorn Unity 4 vilket använder programmeringsspråket C#. Implementationen skulle ha kunnat gjorts i en 2D-miljö eftersom placeringen av rummen enbart placeras i två dimensioner. Dock eftersom man kan använda 2D-uppsättningen till att skapa en 3D-miljö i Unity så fungerade detta lika bra. När ytterligare information behövdes för spelmotorn och programmeringsspråket användes ett online API (Unity, 2018).

För att kunna generera dungeons med hjälp av algoritmerna behövdes något sätt för rummen att kopplas ihop. Vidare så behövdes något sätt för dungeonuppsättningen, som förklaras i kapitel 4.2, att kunna placeras ut på dungeonmiljön. Slutligen behövdes ett sätt att bestämma storlek på hela dungeonmiljön för att verifiera att rummen från BSP ska kunna få en storlek. För att lösa detta så placerades ett rutnät ut innan dungeonmiljöerna omvandlas till 3D. Genom att först skapa en yta som är uppbyggt av ett antal 1x1 rutor så kan man sedan använda bit masking metoden, som beskrivs nedan i 4.2, för att skapa en 3D-miljö. Ta BSP som exempel och se till Figur 10. Detta fungerade som grunden för båda algoritmerna och lade grund för stora delar av experimentmiljön.

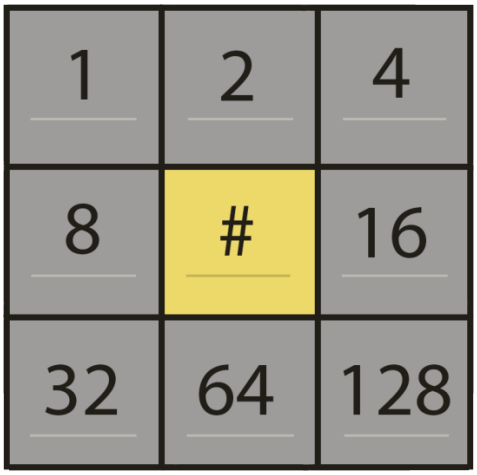


Figur 10: I bilden till vänster syns hur startytan är utplacerad och uppdelad i partitioner, i bilden till höger syns den färdiga dungeonmiljön utplacerad med det nu underliggande rutnätet markerat.

4.2 Dungeonuppsättning och bit masking

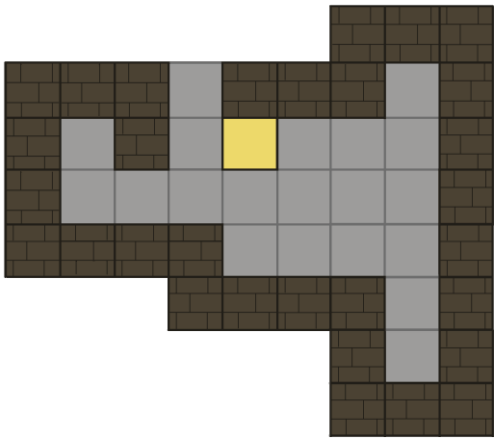
För att ge banorna ett mer representativt utseende för spelindustrin omvandlades de till en 3D-miljö. För att undvika att skapa en dungeon med bara vita kuber som objekt så användes en royalty fri uppsättning av 3D-texturer. Uppsättningen innehöll ungefär 50 olika texturer för att kunna skapa varje 3D-objekt. Uppsättningen av texturerna kan ses i Appendix A. För att beräkna vilken textur som skulle användas för varje individuell rutnätscell så användes en teknik som kallas bit masking.

Bit masking betyder att man räknar ut ett hashvärde för varje vägg, sedan används det hashvärdet för att avgöra vilken 3D-textur som ska användas. Hashvärdet beräknades genom att kolla på de åtta intill liggande positionernas värde, vilka var baserade på om det finns en vägg där eller inte samt vart de var placerade (Driver, 2010), se Figur 11.

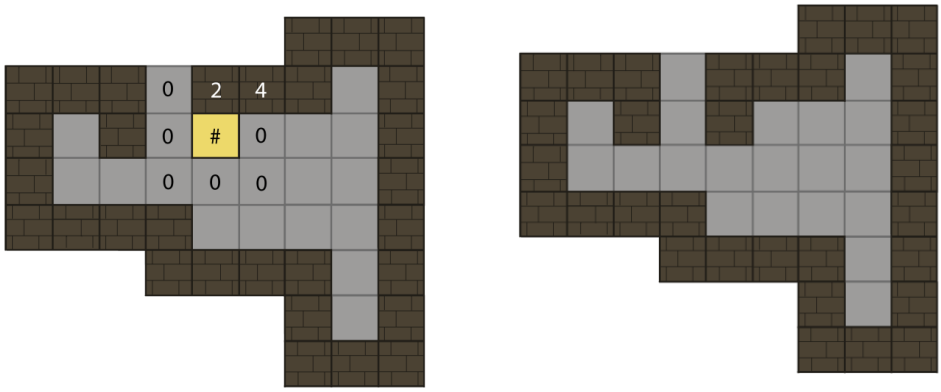


Figur 11: Hashvärden i ett rutnät.

Ifall den gula rutan märkt med "#" i Figur 11 är rutan som hashvärdet ska beräknas för så inspekteras de åtta intill liggande positionerna. Om någon av positionerna innehåller en vägg så kommer värdet av den positionens hash att adderas till den gula rutans hash, se Figur 12, och 13.



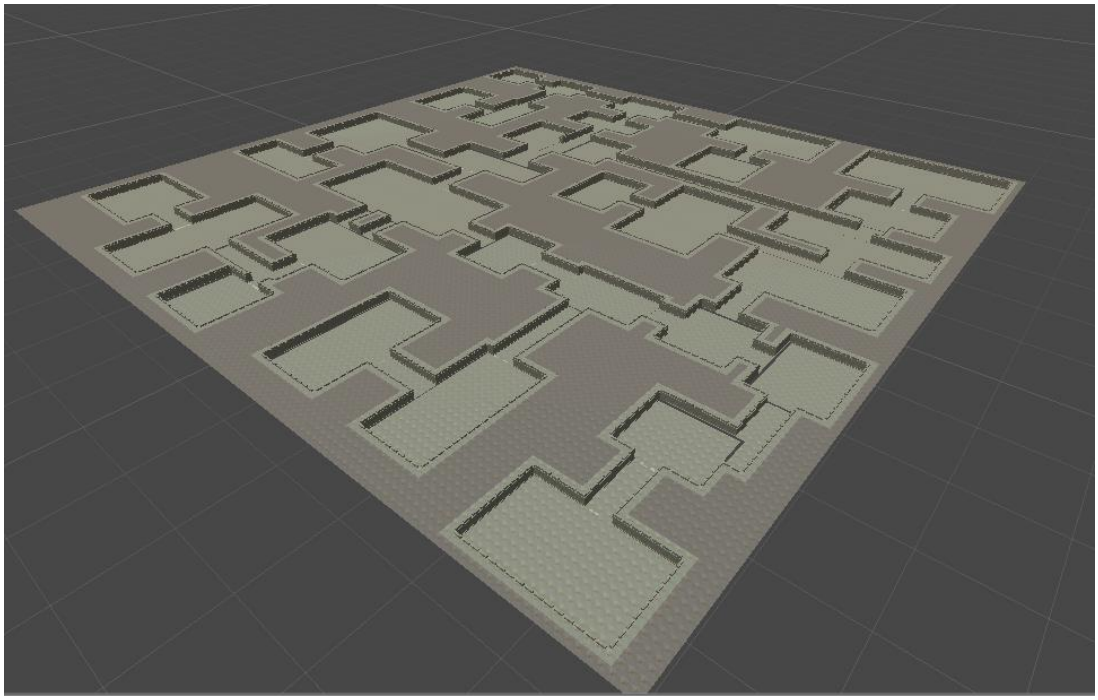
Figur 12: Rutnät representation av positioner i en dungeon.



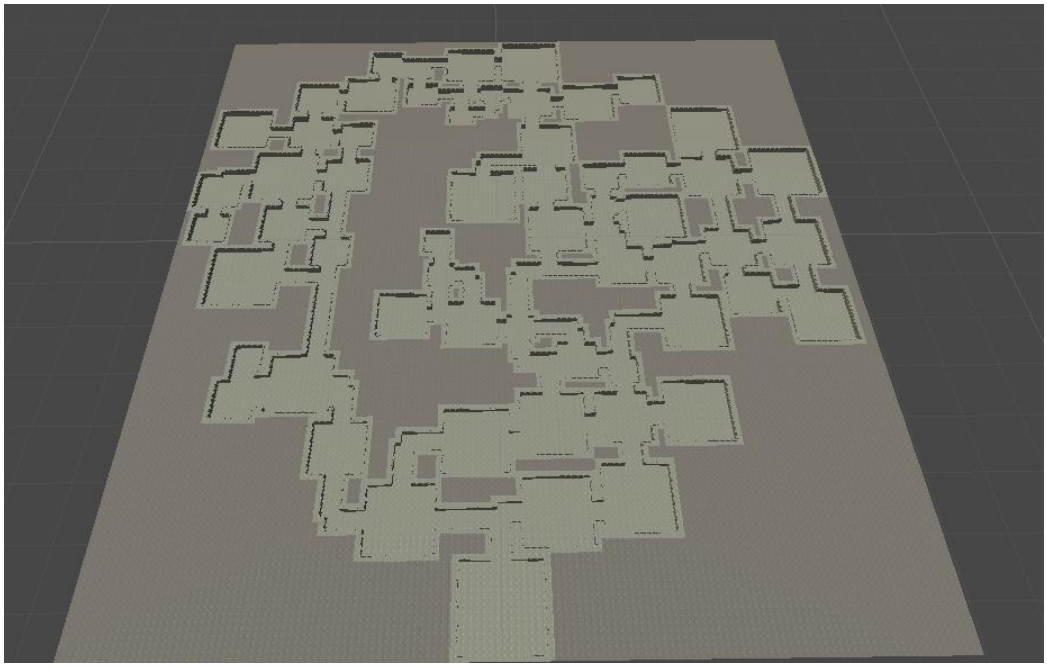
Figur 13: Bilden till vänster visar en rutnäts representation av positioner i en dungeon med hash värden. Bilden till höger visar resultatet av hash beräkningen.

De bruna rutorna är väggar, den gula rutan är positionen där hash värdet ska räknas ut och de gråa rutorna är golv. Enbart de intill liggande rutorna som är väggar har sina värden adderade till hash värdet. I scenariot i Figur 13 kommer den gula rutans hash värde räknas ut till 2 + 4 = 6. I detta fallet skulle det betyda att den gula rutan ska vara en hörnvägg.

Denna dungeonuppsättning och bit masking använde båda algoritmerna för att skapa 3D-miljöer. Meningen med dessa var att skapa banor som ser ut som vad man skulle kunna se i ett spel utvecklat idag. I Figur 14 och Figur 15 finns exempel på hur de genererade kartorna ser ut när texturerna från Appendix A används (Skorpio, 2012).



Figur 14: Skärmdump från en bana skapad av BSP som använder texturuppsättningen.

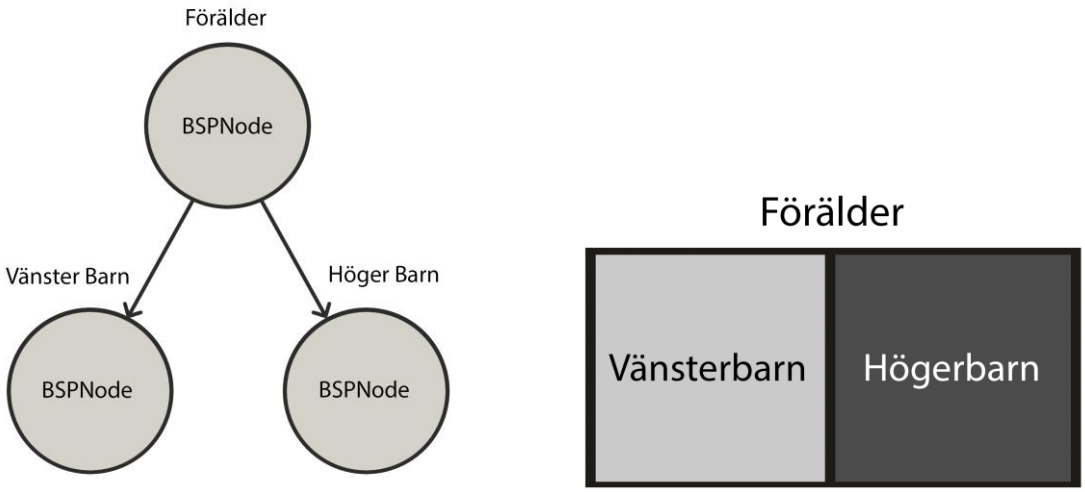


Figur 15: Skärmdump från en bana skapad av DT som använder texturuppsättningen.

4.3 Binary Space Partitioning

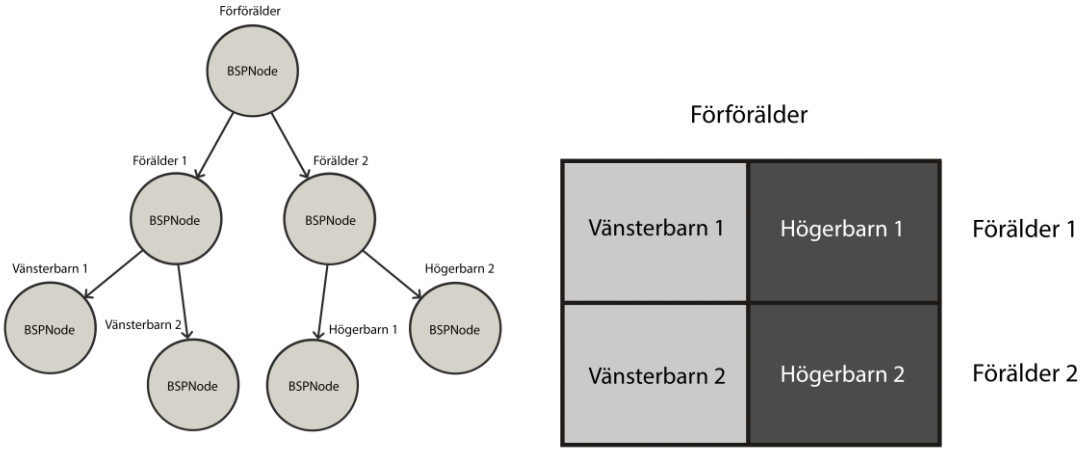
Implementationen av BSP baserades på en beskrivning av algoritmen från boken Procedural Content Generation in Games (Shaker, m.fl. 2016). Algoritmen började med att skapa en stor kvadrat och placerar in den i scenen. Nästa steg var att implementera en funktion som kan hantera uppdelning av startytan i mindre delar. Delningsfunktionen valde slumpmässigt mellan att dela horisontellt eller vertikalt innan sedan valde den slumpmässigt en position att dela på. Sedan utförde den samma process som på den första rutan för de två nya partitionerna. Detta upprepade sig tills dess att ett kriterium hade uppfyllts, i detta arbete var kriteriet att ett antal delningar ska göras för att skapa ett valt antal rum.

När det fanns en funktion för att dela partitioner så behövdes en datastruktur för att lagra BSP trädet. En BSP nodklass skapades för att lagra datan varje partition hade och en referens till tre andra BSP noder: en förälder, ett höger barn och ett vänster barn, se Figur 16.



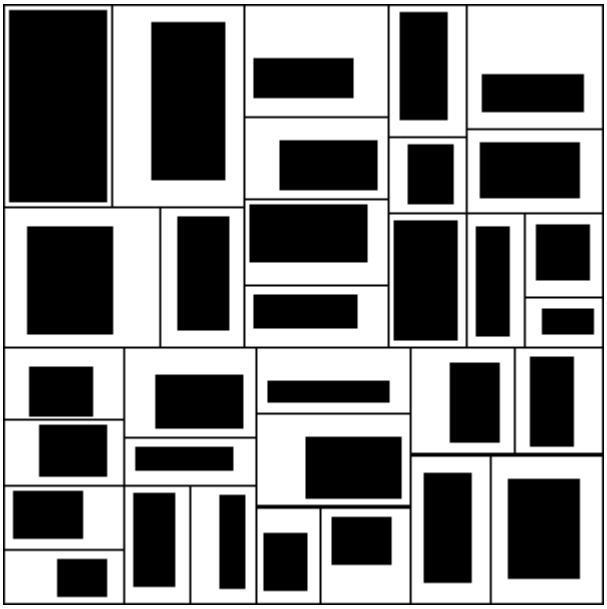
Figur 16: Till vänster syns en representation av BSPNode klassen. Till höger visas hur algoritmens output ser ut i 2D.

När denna datastruktur användes kunde BSP trädet konstrueras. För varje ny delning som genomfördes skapades två nya BSP noder. Dessa två nya noder sattes sedan som barn till partitionen som delades. Eftersom det redan skapats en tom yta, som nämndes i kapitel 4.1, så sattes denna till rotnod. På denna yta utförs sedan den första delningen vilket skapar två nya ytor. Dessa två ytor delas sedan igen på likartat sätt, se Figur 17.



Figur 17: Till vänster syns en representation av BSPNode klassen efter två delningar. Till höger visas hur algoritmens output ser ut i 2D.

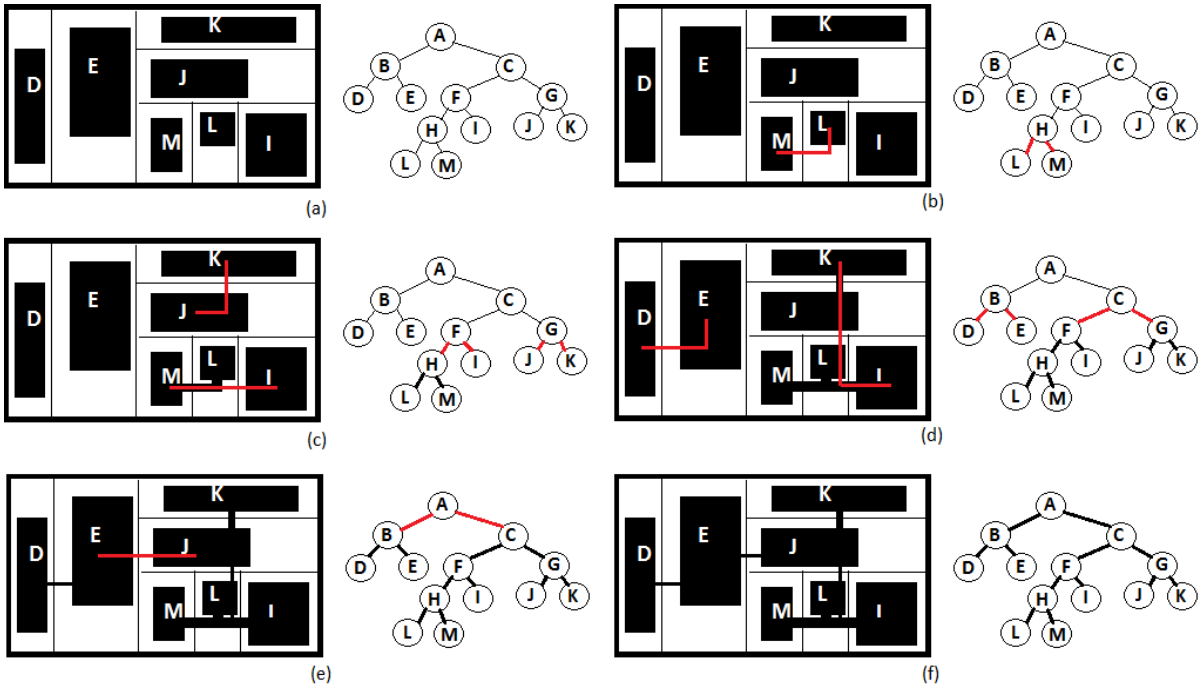
Delningarna fortsätter sedan tills antalet delningar har genomförts. Efter fem delningar av startytan kan man se resultatet i Figur 18.



Figur 18: Skärmdump av outputen efter fem BSP delningar med rum utplacerade.

Nästa steg av implementationen var att lägga till rum till alla löv i BSP trädet. För att hitta löv utfördes en så kallad *breadth first traversal* (BFT) på trädet. När lövnoder hittades så verifierades att de inte hade några barn, ifall de inte hade några barn så placerades ett rum i lövnoden. Rummen som skapades var av slumpad storlek med en rutas marginal runt rummet för väggar. Efter att rummen lagts till så skapas ett 2D rutnät av kartan. Rutnätet av dungeonen omvandlades sedan till en 3D spelvärld. För att börja omvandlingen så konverterades rutor som var markerade som rum till 3D. Rummen gavs en minimum storlek i dimension för att de inte skulle bli för små.

Ett av de mer komplicerade problemen med 3D omvandlingen var metoden för att koppla ihop rum. Det första steget för att koppla ihop rummen var att koppla alla lövnoder till sina syskon. Noder räknades som syskon ifall de har samma förälder. När syskonnoder hade kopplats ihop så användes en rekursiv funktion som fortsatte på BFT och kopplade ihop syskonnoder högre upp i trädet, tills dess att rotnoden nåddes. Ett problem var att så fort funktionen nådde syskonnoder som är högre upp än föräldrar till löv, så innehöll inte noderna längre ett specifikt rum. Av den anledningen så gick algoritmen för en sådan nod neråt i trädet genom nodens barn tills ett rum hittades. Den valde här ingen direkt väg utan gick slumpmässigt neråt i trädet tills den nådde en lövnod. Oavsett hur den kopplade så garanterade denna metod att alla noder var sammankopplade, vilket betydde att den garanterade att alla rum var nåbara. De hittade lövnoderna användes sedan för att koppla ihop noderna högre upp i trädet, se Figur 19.



Figur 19: Visualisering av kopplingsalgoritmen.

Visualiseringen i Figur 19 visar hur tekniken fungerade, med början i (a) och den färdiga dungeonkartan i (f). Denna teknik som användes krävde skapandet av en grävare, vilken skulle placera sig i mitten av ett rum och gräva till mitten av ett annat rum. Under grävningen så omvandlades alla rutor som grävaren passerade till golv omringade av väggar. Grävaren använde rutnätet och satte rutnätsceller till golv och intilliggande rutor till väggar. Det är även här längderna av korridorer beräknas. Varje gång en cell omvandlades till golv ökade längden av korridoren. När alla korridorer hade grävts fram så omvandlades resterande delar av kartan till 3D. Kopplingsalgoritmen kunde röra sig över rutnätet och markera celler som golv enligt beskrivningen nedan, se Figur 20.

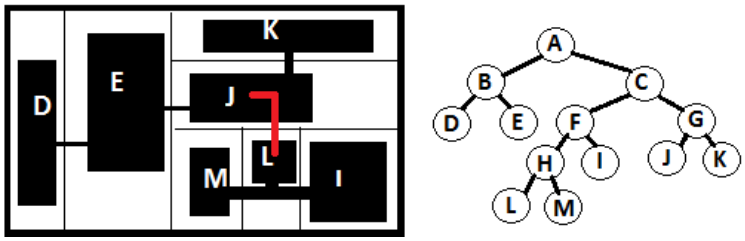
1. **If** startPosition.x < targetPosition.x **then** x += 1 until x = targetPosition.x
   else
   x -= 1 until x = targetPosition.x

2. **If** startPosition.z < targetPosition.y **then** z += 1 until z = targetPosition.z
   else
   z -= 1 until z = targetPosition.z

Figur 20: Illustrering av kopplingsalgoritmen.

Vid det här laget så var så var dungeonen klar, dock så märktes det att användningen av kopplingsalgoritmen gav oönskade resultat. Problemet som uppstod var att ibland kopplades två rum ihop som inte låg intill varandra. Det betyder att grävaren grävde genom ett annat rum för att koppla ihop två rum. Ser man till Figur 19 så syns detta tydligt i kopplingen mellan K och I i (d). Lösningen på detta problem var att implementera en sökalgoritm som utökade

algoritmen i Figur 19. Sökalgoritmen gick ut på att hitta de två rum som låg närmast varandra vilka även var kopplade till rummen som algoritmen utgick ifrån. I fallet med kopplingen i Figur 19 (d) så hade den nya grävaren istället kopplat J till L, se Figur 21.



Figur 21: Illustrering av den utökade kopplingsalgoritmen.

Som syns i Figur 21 så skapades en koppling mellan rum F och G genom att istället koppla ihop rum J och L med hjälp av sökningsalgoritmen. Lösningen var inte perfekt då den inte helt tog bort möjligheten för en korridor att gå igenom ett rum. Dock så ansågs resultatet vara mycket bättre än innan och dugligt nog för denna implementation. På så vis var algoritmen klar och resultatet syns sen tidigare i Figur 14.

4.4 Delauney Triangulation

Delauney generering började med att ett antal rum med slumpmässig storlek skapades och placerades runt mitten av scenen. Efter det separerades rummen från varandra med hjälp av *den omvända kvadratlagen*. Rummen flyttades från varandra tills dess inga rum överlappade. Formeln som användes för den omvända kvadratlagen beskrivs i Figur 22.

Strength = min(k / (distance * distance), maxAcceleration)

Distance = the distance between overlapping rooms

K = 2000

maxAcceleration = 5 (units per frame)

Figur 22: Den omvända kvadratlagens formel (Millington, 2009).

Värdena på **K** och **maxAcceleration** i Figur 22 bestämdes genom att experimentera fram vad som gav ett fungerande resultat. Förutom att flytta rummen för att undvika överlapp så var det viktigt att se till att rummen passade in i det underliggande rutnätet. Detta för att verifiera att senare steg i implementationen fungerade korrekt.

Nästa steg var att skapa Delaunay trianguleringen över rummen. Det fanns flera olika välkända algoritmer för att skapa Delaunay trianguleringen av en grupp med noder (Su & Drysdale, 1997). Valet blev att använda en så kallad *incremental algorithm*, trots att andra algoritmer (exempelvis QuickHull) hade lägre genomsnittlig komplexitet. Incremental var enklare att förstå och implementera, av den anledningen valdes det att implementera denna för att vara säker på både att det fanns tid och att algoritmen fungerade korrekt. För att påbörja Delaunay triangulering så skapas en stor triangel som innehöll alla rum som har skapats.

Algoritmen fortsatte sedan med att lägga till en nod i taget till trianguleringen, samtidigt som den behåller Delaunay egenskaperna av trianguleringen hela tiden. Varje nod i det här laget var ett av rummen. Första steget för att lägga till ett rum till trianguleringen var att hitta vilken deltriangel den ligger i. För att kunna hitta en plats inuti en triangel så hämtades en algoritm som kan avgöra om en punkt är inuti ett konvext eller konkavt skrov, se Figur 23.

1. Create a point that is outside the hull (point Y).
2. Produce a line segment connecting the point in question (point X) to the new point Y.
3. Loop around all edge segments of the hull. Check if the segment intersects with XY.
4. If the number of intersections counted is even, X is outside the hull, Otherwise X is inside the hull.

Figur 23: Algoritm som avgör vilken typ av skrov ett rum ligger i (firemana, 2013).

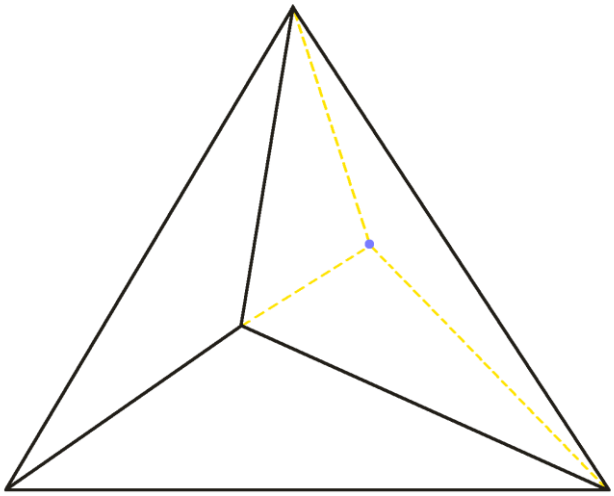
Trots att algoritmen i Figur 23 fungerade så var komplexiteten av implementationen högre än den behövde vara, främst eftersom flertalet hjälpfunktioner för linjekorsningar krävdes för att få det att fungera. Istället hittades en matematisk funktion som löste samma problem, se Figur 24.

```
function SameSide(p1,p2, a,b)
    cp1 = CrossProduct(b-a, p1-a)
    cp2 = CrossProduct(b-a, p2-a)
    if DotProduct(cp1, cp2) >= 0 then return true
    else return false

function PointInTriangle(p, a,b,c)
    if SameSide(p,a, b,c) and SameSide(p,b, a,c)
        and SameSide(p,c, a,b) then return true
    else return false
```

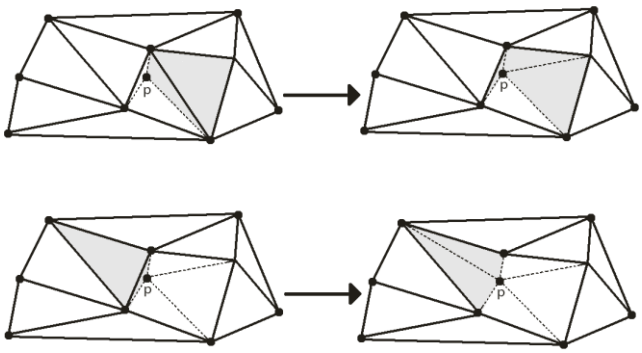
Figur 24: Algoritm som avgör om ett rum är inuti en triangel (blackpawn, 2013).

Eftersom komplexiteten var lägre i algoritmen i Figur 24 är den i Figur 23 så byttes algoritmen ut. Anledningen till bytet kom från bekymmer över att kodbasen snabbt ökade i komplexitet och blev svårare att förstå. Så fort det avgjorts vilken triangel som rummets mittpunkt ligger i kunde trianguleringen fortsätta. Rummets mittpunkt blev en nod som kopplade till de tre hörn av triangeln som den låg i. Nu fanns det tre trianglar istället för en, se Figur 25.



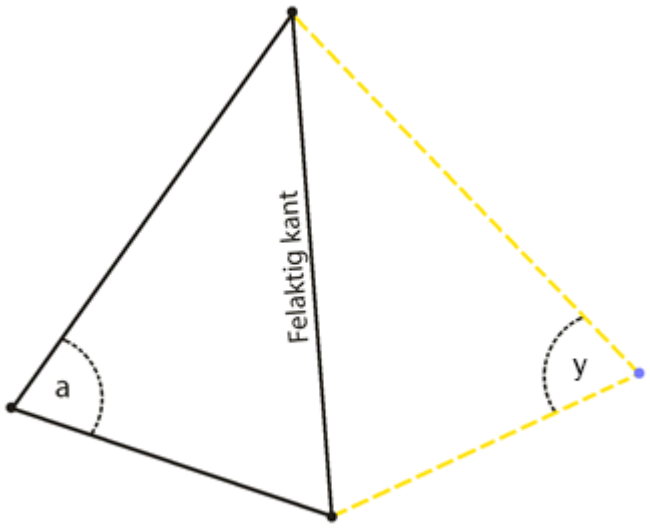
Figur 25: Rum som läggs till i trianguleringen.

När den nya triangeln hade lagts till i trianguleringen så fanns det en risk för att felaktiga kanter existerar. Ett sätt att avgöra om sådana fanns och för att ta bort dem kallas för *Lawson flip*, vilken var den metod som användes i detta arbete (Lawson, 1971). Lawson flip ändrar de onödiga kanterna i de fyrsidiga formerna av de två trianglarna som delar den onödiga kanten, se Figur 26.



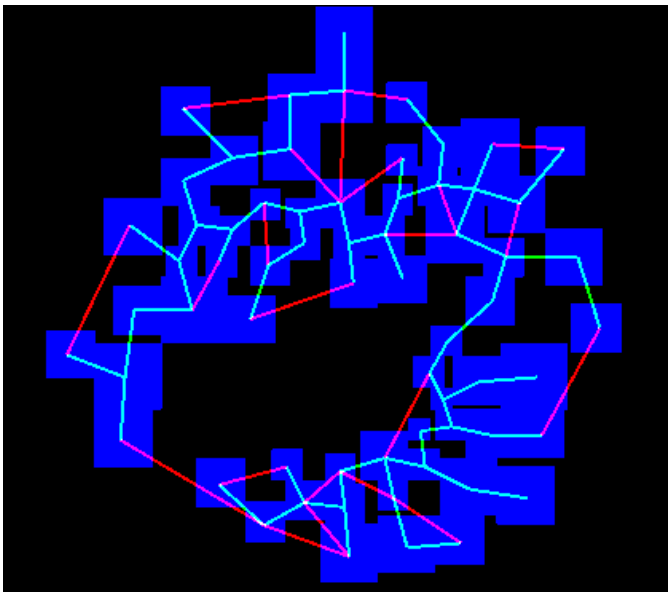
Figur 26: Illustrering av Lawson flip.

För att avgöra ifall en kant är felaktig ar så användes en metod som undersöker hörn som ligger på var sin sida om kanten (Berg, 2010). Ifall summan av vinkel a och vinkel y i Figur 27 är större än 180 grader så vet man att kanten är felaktig. Om en felaktig kant upptäcktes så utfördes en Lawson flip för att verifiera att alla kanter bar Delaunay egenskaper, se Figur 27. Implementationen i detta arbete gjorde det möjligt att ta fram längden på alla kanter. Med hjälp av kanternas längd kunde vinklarna beräknas och då kunde även metoden ovan användas.



Figur 27: Illustrering av en potentiellt felaktig kant.

Nästa steg var att skapa ett så kallat *minimal spanning tree* (framöver förkortas detta till MST) av Delaunay trianguleringen. Det finns två välkända och ofta använda algoritmer för att hitta MST, Prims algoritm och Kruskals algoritm. Båda algoritmerna har samma komplexitet men de beror på olika datastrukturer. För att avgöra vilken som är mest lämpad kontrolleras istället datastrukturerna algoritmen ska arbeta på (Erkan 2018). Dock fanns det ingen metod i Erkans arbete som gick att applicera på detta arbete, av den anledningen söktes en alternativ metod. Svaret hittades istället på forumet Stackoverflow där användaren Todd Gamblin gav svaret att Prims algoritm är det korrekta valet när man har fler kanter än noder (Gamblin, 2009). Eftersom detta arbetets lösning på DT skapade tre kanter för varje nod som lades till så var valet mellan algoritmerna lätt. Prims algoritm ansågs vara mest lämpad för problemet i detta arbete. Efter att Prims algoritm var implementerad så var i stort sett implementationen klar, se Figur 28.



Figur 28: Visualisering av en Delaunay triangulering.

Det enda som fanns kvar att göra var att konvertera 2D kartan till 3D. Detta gjordes med hjälp av dungeonuppsättningen och bit masking som nämndes tidigare. Hela processen för konvertering skedde med samma lösning som i BSP, dock eftersom korridorerna här ligger i MST uppsättningen så behövdes inte den förbättrade kopplingsstrategin. Ett exempel på en output från Delaunay triangulering kan ses i Figur 15 tidigare i dokumentet.

4.5 Utvärdering

Som tidigare nämnt så var de fyra kriterier som algoritmerna utvärderades med följande: tidseffektivitet, möjligheten att nå alla rum, likhet mellan algoritmerna och variation på resultaten. Hur mätningen av dessa kriterier implementerades beskrivs nedan.

4.5.1 Tidseffektivitet

Det första kriteriet som utvärderades för algoritmerna är tidseffektiviteten. För att mäta tiden det tar för varje exekvering så användes Unity klassen Time, och mer specifikt funktionen *realtimeSinceStartup*. Denna funktion returnerar tiden som gått sen start av exekveringen enligt hårdvarans systemklocka. Med andra ord returnerade funktionen tiden det tog för exekveringen att bli klar oavsett om artefakten fick oväntade fördröjningar eller inte. Resultatet skrevs ut till en textfil när exekveringen var klar.

4.5.2 Möjligheten att nå alla rum

Det andra kriteriet för algoritmerna var möjligheten att nå alla rum. Dock som nämndes tidigare så bör resultatet här alltid vara samma. Eftersom detta är ett grundläggande krav för algoritmerna så kommer en korrekt implementation inte skapa banor med rum som inte går att nå. Men för att säkerställa att detta stämde så har en enklare koll implementerats.

4.5.3 Variation och likhet

Som nämndes tidigare så mättes variation och likhet utifrån på tre kriterier: rumsdimensioner, genomsnittlig korridorlängd och antalet återvändsgränder. Under genereringen av en dungeon så skrevs rumsdimensionerna ut när rummen genereras. Eftersom korridorerna grävs ut av 3D grävaren i båda algoritmerna så räknade den hur många celler den omvandlade till golv.

Antalet återvändsgränder räknades på olika sätt för båda algoritmerna. I BSP så markerades rum när de kopplades ihop med andra rum, en ökning i antal grannar varje gång en koppling tog plats. När exekveringen sedan var klar så undersöktes hur många grannar varje rum hade. På likartat sett gick det till i Delaunay algoritmen. När MST räknades ut så noterades antalet grannar för varje rum. För att se om ett rum var en återvändsgränd eller inte så undersöktes antalet grannar, ifall ett rum bara hade en granne så ansågs det vara en återvändsgränd. Alla tre olika värden från de tre kriterierna skrevs ut till en textfil för senare undersökning.

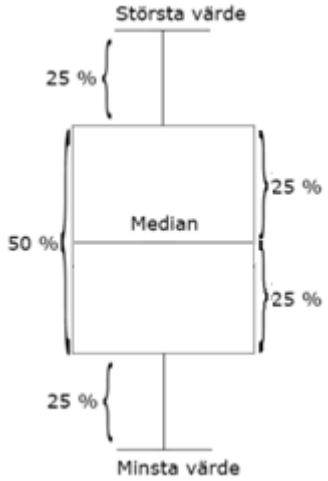
5 Utvärdering

I detta kapitlet presenteras resultatet av utvärderingarna. Först kommer resultaten från undersökningen. Efter det kommer en analys om varför resultaten ser ut som de gör. Avslutningsvis dras slutsatser baserat på analysen.

5.1 Presentation av undersökning

Undersökningens resultat baserades på 100 utförda exekveringar på varje algoritm på fem olika storlekar av banor. Storlekarna som användes i undersökningen är 8, 16, 32, 64 och 128 antal rum. Färre än 8 ansågs vara för lite för en dungeon och resultat över 128 uteslöts då genereringstiden alltid låg för högt över den acceptabla gränsen. Vidare så hade Delaunay Triangulering algoritmen problem när den skulle generera dungeons på 128 rum och exekvering avbröts halvvägs igenom. Problemet låg i sökalgoritmen men det valdes att inte lösas då genereringstiden, även utan sökalgoritmen och sammankopplingen av rummen, var drygt fyra gånger så hög som gränsen. Vidare så låg majoriteten av exekveringen i DT hos den omvända kvadratlagen. Det betyder att även om buggen hade åtgärdats så hade banor med 128 som genererats av DT inte vara användbara ändå. Eftersom inga argument kunde hittas för att rättfärdiga insamlingen av just denna datan så exkluderas den. Den påverkade datan är genomsnittlig korridorlängd och antalet återvändsgränder på DT generering av 128 rum. Genereringstid och rumsstorlekar gav fortfarande resultat och sparades för att gen en insikt i DT generering av banor med 128 rum. Genereringsresultaten sparades ner i textdokument vilka sedan användes för att hitta högsta, lägsta och genomsnittliga värden av varje egenskap hos respektive algoritm och storlek.

Resultaten från utvärderingarna presenterades i så kallade lådagram (Kendrick, Parker & Williamson, 1989). Ett lådagram presenterar data i form av lådor som markerar en majoritet av datafördelningen. Ett exempel på ett lådagram visas i Figur 29.



Figur 29: Exempel på hur ett lådagram kan se ut och hur datan fördelas.

50% av datan ryms i lådan och delningen av lådan representerar medianen av hela dataintervallet. Strecket som går vertikalt genom lådan representerar hela dataintervallet, där den översta punkten visar det största värdet och den nedersta punkten visar det minsta värdet. Värden som ligger längre ifrån lådan än 1,5 gånger avståndet mellan kanter, så kallade

kvartiler, kallas för uteliggare och markeras med en punkt eller liknande (Kendrick m.fl., 1989). Meningen med att använda lådagram är att det ger en bra överblick på dataintervallets spridning, hur uppdelningen ser ut och hur mycket det skiljer sig mellan största och minsta värde. Testerna utfördes på en dator med Intel Core i7-4770K processor med en klockhastighet på 4.0 GHz och 24 GB DDR3 RAM internminne med en minneshastighet på 1600 MHz.

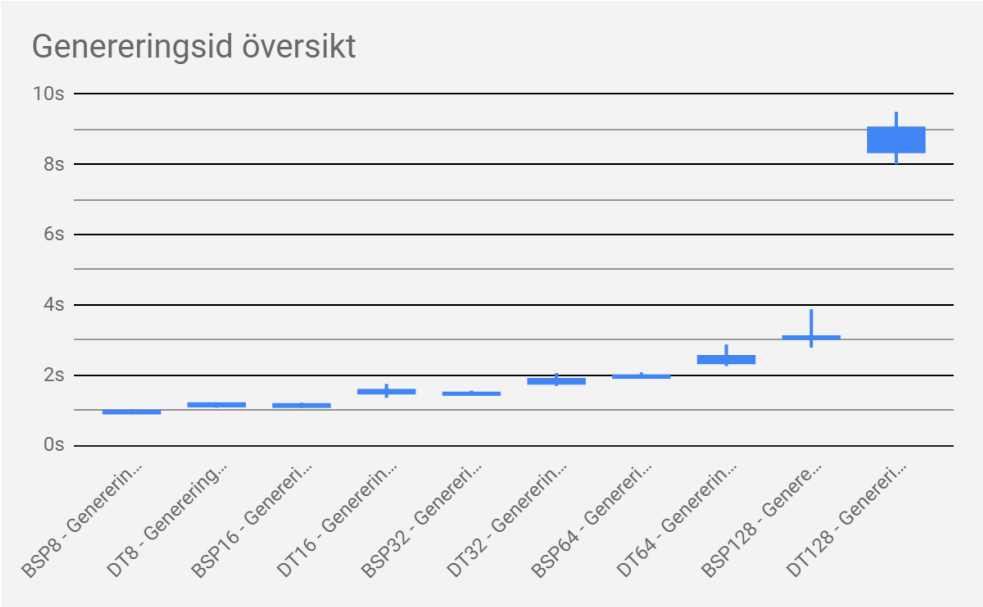
Buggen som nämndes ovan löstes efter att datainsamlingen var gjord. För att verifiera att samlad data var användbar utfördes en test på alla rumsantal som i arbetet, fast med färre genereringar. Resultaten från dessa test var omöjliga att urskilja när de jämfördes med den redan insamlade datan. Av den anledningen så genomfördes inte datainsamlingen på nytt då buggen inte ansågs påverka resultatet.

5.2 Resultat

Nedan följer en genomgång av alla resultat som producerades av artefakten. Varje delkapitel är en av de fem mätvärden som algoritmerna producerar. Efter kapitlet följer en analys av resultaten som presenteras i detta kapitel.

5.2.1 Genereringstid

Nedan följer ett lådagram där algoritmernas genereringstid jämförts. Först kommer en översikt över hela genereringstid undersökningen, se Figur 30, sedan följer en kort genomgång av varje enskilt testfall. För fullständiga resultat i datainsamlingen för genereringstid se Appendix B. Genereringstidens mätvärde är sekunder.



Figur 30: Överblick över genereringstiderna för båda algoritmerna på alla olika rumsantal som användes i undersökningen.

För generering av banor med 8 rum hade BSP minsta värde 0,887 s, största värde 1,026 s och genomsnittligt värde 0,952 s. På samma antal rum hade DT minsta värde 1,066 s, största värde 1,374 s och genomsnittligt värde 1,165 s.

För generering av banor med 16 rum hade BSP minsta värde 1,059 s, största värde 1,213 s och genomsnittligt värde 1,143 s. På samma antal rum hade DT minsta värde 1,350 s, största värde 1,746 s och genomsnittligt värde 1,542 s.

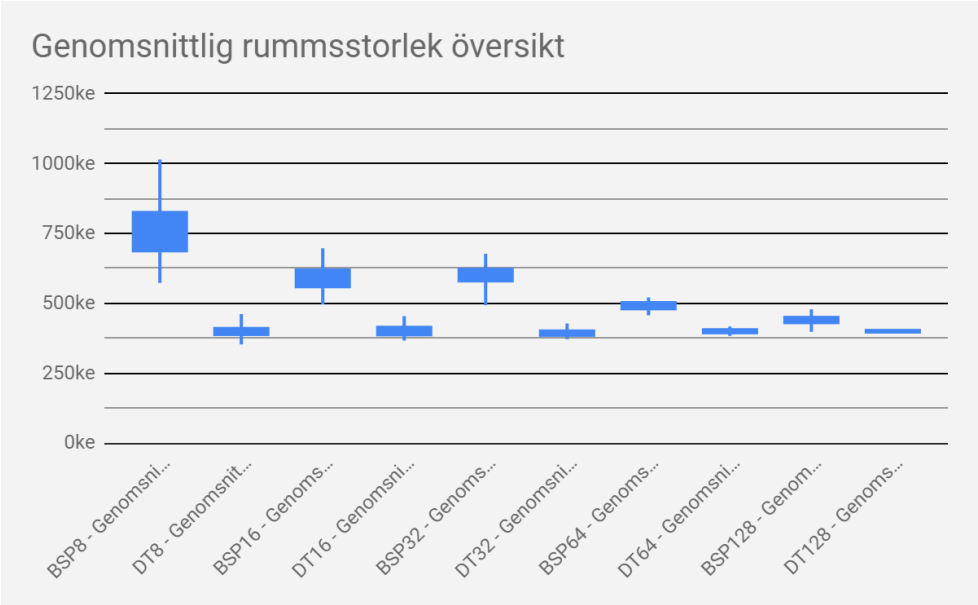
För generering av banor med 32 rum hade BSP minsta värde 1,444 s, största värde 1,562 s och genomsnittligt värde 1,488 s. På samma antal rum hade DT minsta värde 1,691 s, största värde 2,068 s och genomsnittligt värde 1,845 s.

För generering av banor med 64 rum hade BSP minsta värde 1,901 s, största värde 2,073 s och genomsnittligt värde 1,985 s. På samma antal rum hade DT minsta värde 2,254 s, största värde 2,868 s och genomsnittligt värde 2,478 s.

För generering av banor med 128 rum hade BSP minsta värde 2,782 s, största värde 3,871 s och genomsnittligt värde 3,105 s. På samma antal rum hade DT minsta värde 8,007 s, största värde 9,498 s och genomsnittligt värde 8,802 s.

5.2.2 Genomsnittlig rumsstorlek

Nedan följer ett lådagram där algoritmernas genomsnittliga rumsstorlek jämförs. Först kommer en översikt över hela genomsnittliga rumsstorleks undersökningen, se Figur 31, sedan följer en kort genomgång av varje testfall. För fullständiga resultat i datainsamlingen för genomsnittlig rumsstorlek se Appendix C. Genomsnittlig rumsstorleks mätvärde är kvadratenheter (kommer härifrån förkortas till ke), där en kvadratenhet är 1X1 ruta av det underliggande rutnätet.



Figur 31: Överblick över genomsnittlig rumsstorlek för båda algoritmerna på alla olika rumsantal som användes i undersökningen.

För generering av banor med 8 rum hade BSP minsta värde 573,062 ke, största värde 1014,5 ke och genomsnittligt värde 767 ke. För DT var minsta värde 352,44 ke, största värde 461,76 och genomsnittligt värde 403, 04 ke.

För generering av banor med 16 rum hade BSP minsta värde 496,093 ke, största värde 697,27 ke och genomsnittligt värde 594,639 ke. För DT var minsta värde 366,6 ke, största värde 454,2 och genomsnittligt värde 403, 22 ke.

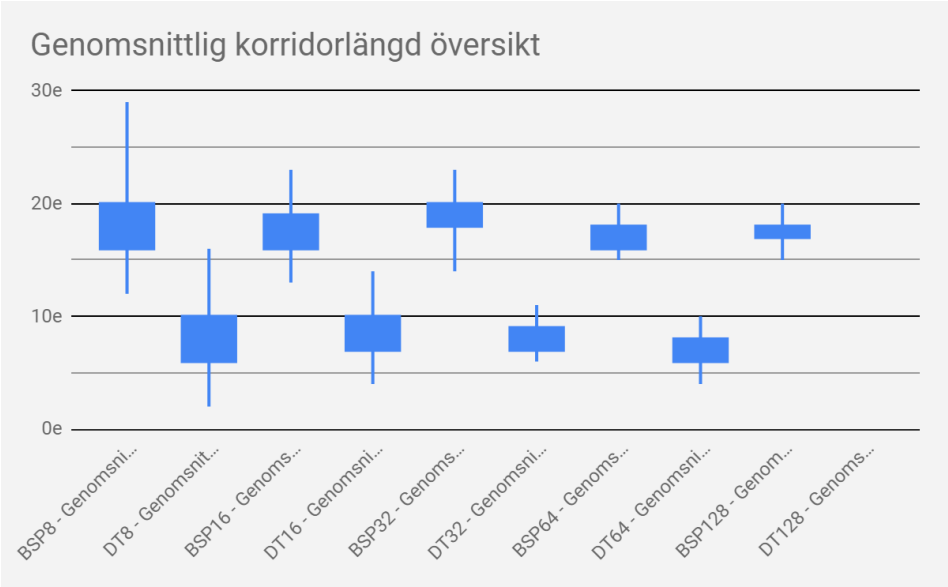
För generering av banor med 32 rum hade BSP minsta värde 493,805 ke, största värde 677,32 ke och genomsnittligt värde 606,272 ke. För DT var minsta värde 371,525 ke, största värde 428,288 och genomsnittligt värde 397, 805 ke.

För generering av banor med 64 rum hade BSP minsta värde 457,324 ke, största värde 521,304 ke och genomsnittligt värde 494,575 ke. För DT var minsta värde 383,67 ke, största värde 417,18 och genomsnittligt värde 404, 672 ke.

För generering av banor med 128 rum hade BSP minsta värde 398,246 ke, största värde 478,051 ke och genomsnittligt värde 444,994 ke. För DT var minsta värde 395 ke, största värde 404,827 och genomsnittligt värde 403, 927 ke.

5.2.3 Genomsnittlig korridorlängd

Nedan följer ett lådagram där algoritmernas genomsnittliga korridorlängd jämförs. Först kommer en översikt över hela genomsnittliga korridorlängds undersökningen, se Figur 32, sedan följer en kort genomgång av varje testfall. För fullständiga resultat i datainsamlingen för genomsnittlig korridorlängd se Appendix D. Genomsnittlig korridorlängds mätvärde är enheter (kommer härifrån förkortas till e), där en enhet är längden av en 1x1 ruta av det underliggande rutnätet.



Figur 32: Överblick över genomsnittlig korridorlängd för båda algoritmerna på alla olika rumsantal som användes i undersökningen.

För generering av banor med 8 rum hade BSP minsta värde 12 e, största värde 29 e och genomsnittligt värde 18,586 e. För DT var minsta värde 2 e, största värde 16 e och genomsnittligt värde 8,01 e.

För generering av banor med 16 rum hade BSP minsta värde 13 e, största värde 23 e och genomsnittligt värde 17,788 e. För DT var minsta värde 4 e, största värde 14 e och genomsnittligt värde 8,636 e.

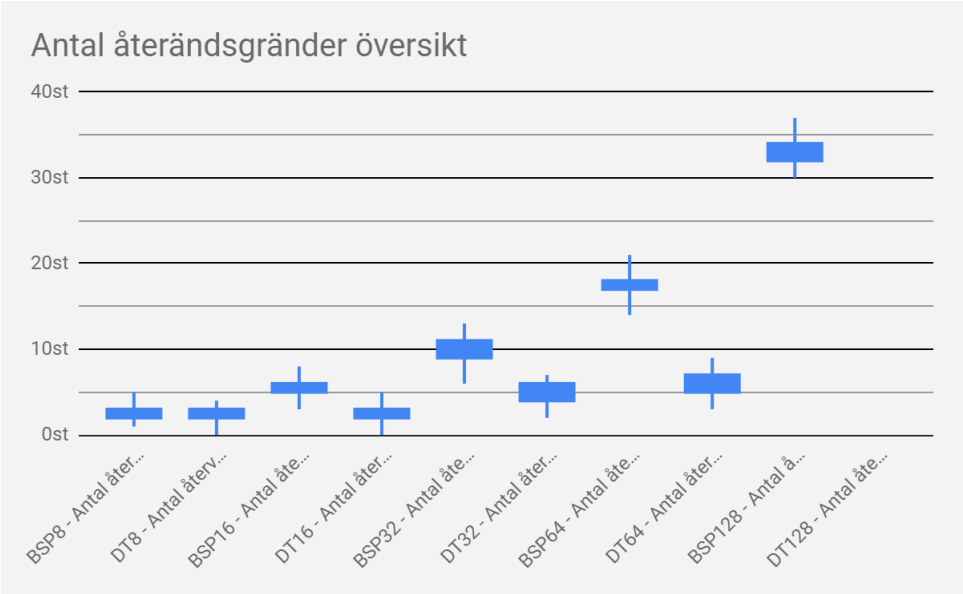
För generering av banor med 32 rum hade BSP minsta värde 14 e, största värde 23 e och genomsnittligt värde 19,52 e. För DT var minsta värde 6 e, största värde 11 e och genomsnittligt värde 8,44 e.

För generering av banor med 64 rum hade BSP minsta värde 15 e, största värde 21 e och genomsnittligt värde 17,6 e. För DT var minsta värde 4 e, största värde 12 e och genomsnittligt värde 7,3 e.

För generering av banor med 128 rum hade BSP minsta värde 15 e, största värde 20 e och genomsnittligt värde 17,67 e. För DT samlades ingen data, vilket förklarades ovan.

5.2.4 Antal återvändsgränder

Nedan följer ett lådagram där algoritmernas antal återvändsgränder jämförs. Först kommer en översikt över hela genomsnittliga korridorlängds undersökningen, se Figur 33, sedan följer en kort genomgång av varje testfall. För fullständiga resultat i datainsamlingen för antalet återvändsgränder se Appendix E. Antalet återvändsgränders mätvärde är styck (kommer härifrån förkortas till st).



Figur 33: Överblick över genomsnittlig korridorlängd för båda algoritmerna på alla olika rumsantal som användes i undersökningen.

För generering av banor med 8 rum hade BSP minsta värde 1 st, största värde 5 st och genomsnittligt värde 2,98 st. För DT var minsta värde 0 st, största värde 4 st och genomsnittligt värde 2,56 st.

För generering av banor med 16 rum hade BSP minsta värde 3 st, största värde 8 st och genomsnittligt värde 5,62 st. För DT var minsta värde 0 st, största värde 5 st och genomsnittligt värde 2,44 st.

För generering av banor med 32 rum hade BSP minsta värde 6 st, största värde 13 st och genomsnittligt värde 9,83 st. För DT var minsta värde 2 st, största värde 7 st och genomsnittligt värde 4,52 st.

För generering av banor med 64 rum hade BSP minsta värde 14 st, största värde 21 st och genomsnittligt värde 17,74 st. För DT var minsta värde 3 st, största värde 9 st och genomsnittligt värde 6,44 st.

För generering av banor med 128 rum hade BSP minsta värde 30 st, största värde 37 st och genomsnittligt värde 33,44 st. För DT samlades ingen data, vilket förklarades ovan.

5.2.5 Antal icke nåbara rum

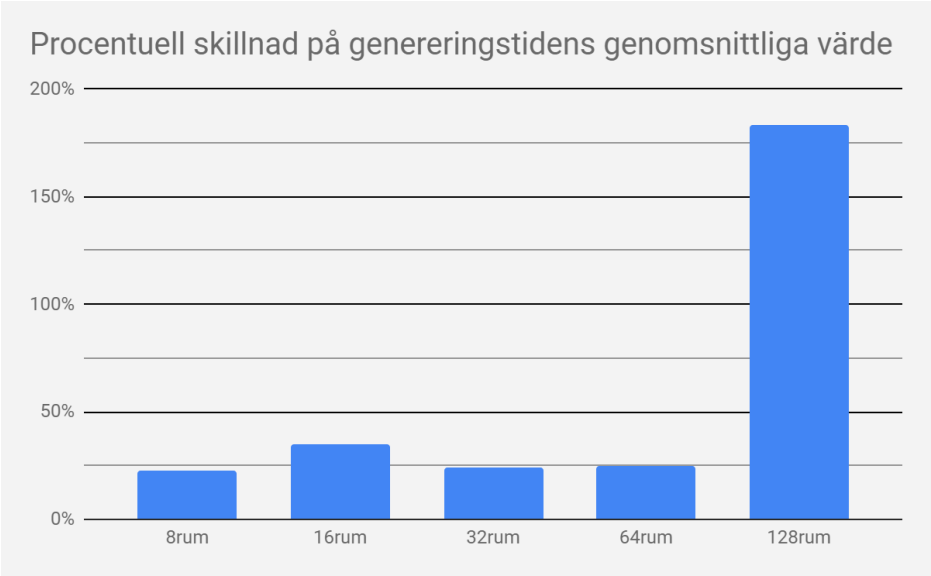
Detta test var som nämnt tidigare mest en verifiering på att algoritmerna fungerar som de ska, vilket visar sig i att siffran alltid var noll på alla algoritmer och storlekar. Enda avvikelsen från detta var i DT på 128 rum, dock beror detta på att exekveringen avbröts och av den anledningen exkluderas detta resultat.

5.3 Analys

I följande kapitel presenteras en analys av datan som samlades in i undersökningen som beskrevs i föregående kapitel. Analysen pekade ut viktiga skillnader och jämförde egenskaperna hos innehållet som algoritmerna genererar. Vidare så analyserades och jämfördes resultat och skillnader mellan olika storlekar på banor inbördes hos algoritmerna.

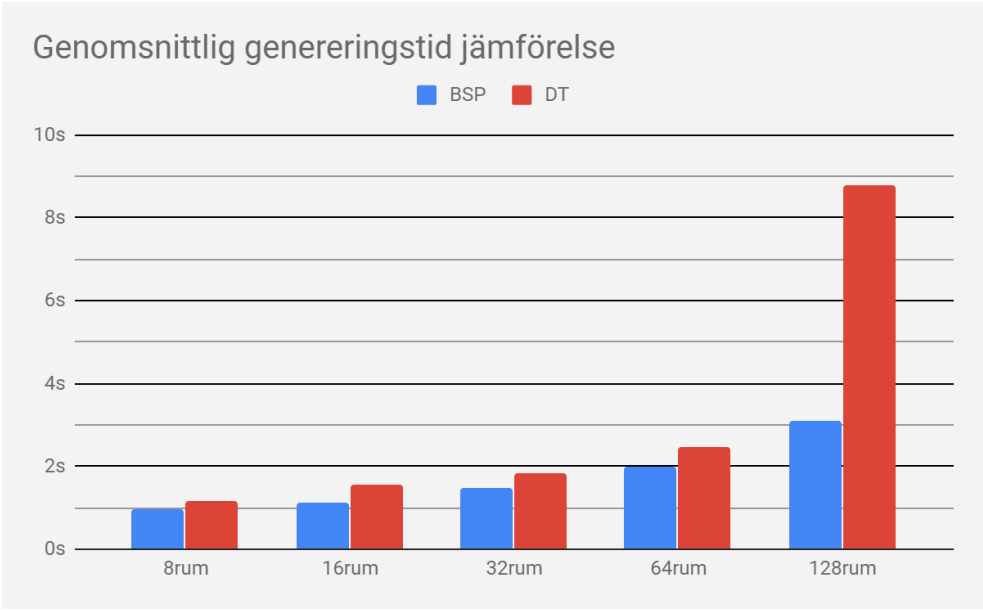
5.3.1 Genereringstid

Undersökningens resultat visade tydligt på att genereringstiden för en dungeon av DT är högre jämfört med en dungeon av samma storlek genererad av BSP. Genereringstiden är konsekvent lägre hos BSP än DT på alla storlekar på banor. Skillnaden plottades ut på en graf för att ge en mer översiktlig bild, se Figur 34.



Figur 34: Procentuell skillnad på genereringstidens genomsnittliga värde mellan algoritmerna. Hur mycket högre genomsnittlig genereringstid DT har än BSP procentuellt.

Som visades i Figur 34 så låg den procentuella skillnaden runt 25% fram tills antalet rum ökar till 128 då skillnaden drar iväg kraftigt. Vidare i Figur 35 syntes den genomsnittliga genereringstiden på varje storlek för båda algoritmerna jämförda mot varandra, se Figur 35.



Figur 35: Jämförelse av genomsnittlig genereringstid.

Resultatet i Figur 35 visade på att DT har kraftigare tillväxt i tid än BSP, vilket talade för att BSP alltid kommer vara första valet ifall tidsaspekten bär mest tyngd. Dock så sattes en gräns för godkänt resultat tidigare, ungefär två sekunder. Denna gräns klarar både DT och BSP av upp till 32 rum, efter detta klarade BSP gränsen på 64 rum men DT var precis ovanför gränsen. Viktigt att peka ut dock var att BSP alltid var under gränsen på 64 rum, även på algoritmens sämsta resultat, samtidigt som DT aldrig var under gränsen, även på algoritmens bästa resultat.

Eftersom resultaten mättes av systemklockan, genomfördes en algoritmanalys för att avgöra huruvida resultatet ovan borde stämma på alla testmiljöer. Resultatet från algoritmanalysen var att BSP algoritmen som implementerats i detta arbete kan summeras i komplexiteten $O(n \log_2 n)$. DT däremot hade komplexiteten $O(n^2)$. Detta betydde att algoritmerna borde ge likartade resultat, alltså att DT tog längre tid än BSP, oavsett testmiljö ifall de implementeras på samma sätt som i detta arbete.

Huruvida de större rumsantalen, 64 och 128 rum, var relevanta för spelutvecklingens värld var svårt att avgöra. Det hittades ingen källa som avgjorde vad ett genomsnittligt rumsantal var, och av den anledningen genomfördes en mindre undersökning. Flera relevanta spel med procedurellt genererade dungeonkartor som hade en aktiv spelarbas testades, Diablo 3, Path of Exile och Realm of the Mad God, för att skapa översiktlig bild av ämnet. Resultatet visade på att det hela var en subjektiv fråga. Alla tre spelen som testades hade allt från 10 till 100 rum, dock låg genomsnittet närmare 20 till 30 med viss variation mellan de olika spelen. Resultatet verkade tyda på att antalet rum beror på vilken upplevelse spelutvecklaren vill ge spelaren. Vilket i sin tur betydde att de större rumsantalen kan vara relevanta, ifall det var en önskvärd egenskap för spelutvecklaren.

5.3.2 Genomsnittlig rumsstorlek

Genomsnittlig rumsstorlek hade BSP konstant högre än DT. BSP hade också konsekvent större spridning än DT. Dock är dessa resultat missvisande och kan därför anses inte vara ett helt korrekt resultat. Problemet ligger i att de slumpgenererade rummen begränsas av hur

algoritmerna är uppbyggda. Eftersom BSP delade upp en startyta i partitioner så var rumsstorleken direkt kopplad till hur många rum som skapas på ytan. Detta kan fortfarande styras av utvecklaren till viss grad. I DT så skapades ett visst antal rum med en bestämt storlek, därav så sätts den genomsnittliga rumsstorleken indirekt av skaparen av algoritmen. Detta gör inte resultatet ointressant, snarare tvärt om. Eftersom rumsstorleken sattes på olika sätt i algoritmerna så blev detta istället centrum av diskussionen.

BSP hade högre spridning vilket tydde på att den gav ett mindre förutsägbart resultat. I DT däremot kunde man sätta exakt rumsstorlek, eller exakta intervallet som rumsstorleken baserades på. Med andra ord tydde resultaten på att BSP hade mer sporadiska rumsstorlekar medan DT hade mer specifika rumsstorlekar baserat på vad utvecklaren efterfrågade. Den mer sporadiska fördelningen i BSP tydde även på att den bestod högre variationsmöjligheter jämfört med DT. Eftersom båda algoritmerna kunde sätta ungefärliga intervall för de slumpgenererade rummen, så fanns data som talade för viss likhet mellan algoritmerna.

5.3.3 Genomsnittlig korridorlängd

Genomsnittlig korridorlängd hade BSP högre än DT. Båda algoritmernas genomsnittliga korridorlängd verkade vara någorlunda konsekvent. BSP låg någonstans mellan 15 och 20 på alla rumsantal medan DT låg mellan 5 och 10 på alla rumsantal. När det kom till spridningen så såg algoritmerna väldigt lika ut. Lådorna i lådagramen hade ungefär samma storlek, vilket betydde att cirka 50% av alla genereringar mellan algoritmerna låg inom lika stort intervall.

När det kom till variationsaspekten så betydde detta att de olika algoritmerna hade likvärdiga egenskaper då de hade ungefär lika stor spridning. Detta ledde till att viss likhet fanns mellan algoritmerna även om värdena i intervallen inte är samma. Hade spridningen varit större i en av algoritmerna hade den varit att föredra ifall man eftersökt större variation, men resultatet talade för att båda algoritmerna gav likartade resultat i den aspekten.

5.3.4 Antal återvändsgränder

Även om algoritmerna började på ungefär samma antal återvändsgränder så steg BSP i antal samtidigt som DT höll sig ganska lågt. Med tanke på att BSP kopplade ihop alla rum med minst ett annat och DT skapade ett MST över rummen samtidigt som den sparar cykler, var det här ett förväntat resultat. Antalet återvändsgränder som DT genererade stannade ganska nära samma värde hela tiden, även om det steg aningen för varje ökning i rumsantal. BSP däremot näst intill dubblade i genomsnitt för varje tvåpotens ökning i rumsantal.

När det kom till spridning låg båda algoritmerna ganska nära varandra. De få ökade spridningarna kunde argumenteras för att vara tillfälligheter. Spridningen höll sig i båda algoritmerna på alla storlekar runt ungefär fem till sex. I likhetsaspekten så var algoritmerna ganska olika med tanke på antalet, dock var de snarlika i spridningen. Ifall man såg till variationsaspekten så höll sig båda algoritmernas låda i lådagramen inom två till tre i skillnad på antal.

5.4 Slutsatser

Slutsatser drogs utifrån de fyra punkterna som arbetet tog sig an att besvara: att hitta viktiga skillnader i tidsaspekten, likhet, variation och möjligheten att nå alla rum. En direkt slutsats som kunde dras var båda algoritmerna inte skapade några icke avklarbara banor. Det faktum att båda algoritmerna alltid skapade noll icke nåbara rum gav ett tydlig resultat.

Ser man till tidsaspekten så fanns det en enkel slutsats att dra. Som tidigare nämnt, ifall genereringstiden bär störst vikt så är det BSP som är det uppenbara valet. Att algoritmen inte hade några problem med någon storlek upp till 64, och gränsade på att klara av 128, samtidigt som DT inte klarade av att generera en enda bana på storlek 64 inom tidsramen, ger BSP en stor fördel. Utöver detta gav även BSP ett bättre resultat på alla rumsstorlekar, där den som minst hade 23% lägre genomsnittlig genereringstid. Enligt artefakterna och testresultaten framtagna i detta arbete så kan man avgöra att BSP generera dungeons snabbare än DT.

Variationsaspektens syfte var att avgöra ifall någon av algoritmerna bar fördel ifall en större variation mellan banorna eftersöktes. Detta mättes med tre framtagna mätvärden: genomsnittlig rumsstorlek, genomsnittlig korridorlängd och antal återvändsgränder. För genomsnittlig rumsstorlek drogs slutsatsen att det är upp till utvecklaren att avgöra vilka intervall som rummen ska falla inom. Av den anledningen fick ingen av algoritmerna någon direkt fördel. Dock kan man direkt sätta ett rumsstorleks intervall för DT medan man får använda lite matematik för att avgöra rumsstorlekarna för BSP. Ifall målet är en enklare lösning för denna aspekten så är DT det uppenbara valet.

Genomsnittlig korridorlängd däremot fanns det tydligare skillnader. Båda algoritmerna skapade korridorlängder med likartade längd mellan alla olika rumsantal. Valet mellan algoritmerna ur denna aspekten faller snarare på hur långa korridorer som önskas. Ifall målet är längre korridorer så är BSP det rätta valet, dock önskas kortare korridorer så är DT att föredra.

Slutligen antalet återvändsgränder fanns det inga stora skillnader mellan algoritmerna. Båda algoritmerna hade ungefär samma spridning i antalet återvändsgränder, oavsett vilken dungeon storlek som genererades. Den enda skillnaden var att antalet återvändsgränder ökade med antalet rum i BSP, samtidigt som ökningen för DT var minimal. Slutsatsen för vilken algoritm som gav mer variation drogs till fördel för BSP. Algoritmen hade aningen större intervall på alla tre aspekterna och av den anledningen drogs följande slutsats: ifall variationsaspekten är viktigast så är BSP det bättre valet. Dock ifall man tvärtemot önskar låg variation så blir DT rätt val.

Vidare syftade arbetet att avgöra ifall algoritmerna BSP och DT var lika nog i resultatet att kunna ersätta varandra. Som nämndes ovan så fanns det skillnader mellan algoritmerna och de genererar inte exakt samma banor. Dock ifall man ska nämna likheterna mellan algoritmerna, så kan båda kan användas för att sätta storlek på rummen i dungeonen, båda har fasta genomsnittliga korridorlängder även om de har olika värden och båda algoritmerna har ungefär samma intervall för hur många återvändsgränder som skapas. I slutändan så är dessa skillnader subjektiva frågor som en utvecklare måste ta ställning till för att avgöra vilken som är att föredra. Men ska en rak slutsats dras på ifall algoritmerna kan ersätta varandra så är svaret ja, så länge banorna håller sig under 64 rum i antal. Ifall önskemålet är fler än 64 rum så kan inte DT användas, då algoritmen inte genererar rum snabbt nog för att stanna under den acceptabla gränsen.

6 Avslutande diskussion

I följande kapitel presenteras först en kort sammanfattning för hela arbetet. Sedan följer en större diskussion om arbetets resultat, alternativa användningsområden, forskningsetik samt arbetets påverkan på samhället. Avslutningsvis presenteras några förslag på hur arbetet skulle kunna utvecklas i framtiden.

6.1 Sammanfattning

Syftet med arbetet var att jämföra två algoritmer som kan användas för att procedurellt generera en dungeon online. De två algoritmerna som undersöktes var Delaunay Triangulation (DT) och Binary Space Partitioning (BSP) där egenskaperna som jämfördes var genereringstid, likhet, variation samt pålitlighet. Genereringstid beskriver den tid det tar för algoritmen att generera en dungeon, likhet mäter hur lika algoritmernas resultat är gentemot varandra, variation mäter skillnaden mellan algoritmernas resultat inbördes och pålitlighet beskriver ifall en dungeon är fullt sammankopplad eller inte.

Slutsatsen från undersökningen var att BSP genererade dungeons på kortare tid än DT, även om båda algoritmerna genererade dungeons på acceptabel tid. När det kommer till likhet så visades att det finns tydliga likheter mellan resultaten hos algoritmerna. Ur variationsaspekten drogs slutsatsen att BSP gav aningen mer sporadiska resultat än DT. När det kommer till pålitlighet genererade båda algoritmerna fullt kopplade dungeons i alla tester. Vid en korrekt implementation av båda algoritmerna så ska både BSP och DT generera fullt sammankopplade dungeons, vilket också visade sig i resultaten.

6.2 Diskussion

Procedurell generering är en växande del av spelutvecklingen och har varit en del av den i mer än tre decennier, dock är det först på senare år som den undersökts genom akademisk forskning. I samband med att ämnet tar mer plats i den akademiska världen har nya metoder, såväl som nya varianter och kombinationer av äldre metoder, skapats. Bland de nyare metoderna har vissa en stor efterfrågan på mer forskning och utveckling innan de kan användas inom spelutveckling (Shaker m.fl., 2016). Av algoritmerna som undersöktes var det Delaunay Triangulering som var av störst behov av mer forskning. Det hittades ytterst få arbeten som nämnde Delaunay och de få som fanns uttryckte ett behov av mer forskning.

Detta arbete har som mål att ge större insikt i egenskaper och skillnader mellan två algoritmer. Metoden som valdes för studien ledde till ett resultat som tydligt framhäver skillnader i egenskaperna hos algoritmerna. Detta kan förhoppningsvis komma att vara en avgörande faktor i potentiella framtida spelutvecklingsprojekt, då de kan användas för att avgöra vilken algoritm man som utvecklare föredrar. Dock mättes alla egenskaper i värden som en dator kan hantera, av den anledningen visar inte arbetet hur en spelare hade upplevt skillnaderna. Att genomföra en teknisk undersökning kommer inte utan sin problematik (Wohlin m.fl., 2012).

Det främsta problemet med en teknisk undersökning utan testpersoner är svårigheten med att avgöra vad de olika resultaten betyder för en spelare. Att den genomsnittliga korridorslängden är fem eller femton säger inte så mycket om hur en individ hade upplevt banorna olika. För att öka arbetets användbarhet bör en kompletterande undersökning med testpersoner utföras. Eftersom detta arbete fokuserar på de tekniska aspekterna, skulle en kompletterande

undersökningen därför behöva fokusera på spelarperspektivet. Resultat från det framtida arbetet bör lyfta fram hur samma skillnader som presenterades i detta arbete upplevs av spelare.

6.2.1 Trovärdighet

I undersökningen som beskrivs i rapporten finns det ett flertal aspekter som behöver diskuteras i form av för- och nackdelar när det kommer till arbetets trovärdighet. Det första problemet är att båda algoritmerna har implementerats utifrån en generell beskrivning. Majoriteten av arbetets kod har skrivits specifikt för det här arbetet och inte av de som uppfann algoritmerna. Detta leder till att implementationerna eventuellt inte är helt optimala och kan därför komma att ge missvisande resultat. Vidare så har detta arbete valt att implementera skapandet av 3-dimensionella dungeons där genereringen av spelvärlden även tar upp genereringstid. Anledningen, som nämnt tidigare i arbetet, var att jämförelsen specifikt inriktade sig på 3D-miljögenerering och tanken var då att resultaten främst riktas mot generering av 3D-spel. Problematiken som uppstår här är att resultatet i denna studie blir svårt att jämföra med andra studier. Ifall implementationerna sett likadant ut som i andra studier på samma algoritmer hade det varit lättare att avgöra ifall implementationen är korrekt. Vidare så har de flesta studier olika testmiljöer, med andra ord så utförs experimenten på olika datorer. Även det leder till att resultaten är svåra att jämföra och att valideringen mellan studier blir missvisande.

Ytterligare problem med jämförelse av andra studier kommer från att DT inte har tillräckligt med forskning i jämförelse med BSP. Detta leder till att det inte finns så mycket att jämföra med, vilket i sin tur leder till att det blir svårt att validera huruvida implementationen är korrekt eller inte. Utöver detta så genomförs de flesta studier som detta arbetet utgått från med ytterst lite återkoppling till spelindustrin. De flesta genererar dungeons som enbart är dungeonuppsättningar med platta 2D objekt. Som tidigare nämnt, så var tanken att genomföra en studie som tar fram resultat som direkt kan stödja spelutvecklare i industrin. Genom att generera kartor och inte spelmiljöer så försvinner stor potential till användbarhet. Det finns två utfall av det här, till att börja med kommer detta arbete direkt kunna användas för att avgöra vilken av algoritmerna som bäst lämpar sig för generering av 3D-miljöer. Däremot ifall frågan om vilken av algoritmerna som lämpar sig för en 2D-miljö skulle ställas, så finns inte samma garanti.

En annan aspekt värd att diskutera är det faktum att 100 tester genomfördes för varje algoritm på varje dungeonstorlek. Ifall fler tester istället hade genomförts så hade det kunnat gett en tydligare indikation på hur stor spridning algoritmerna har. Fler avvikande resultat och en eventuellt större spridning är båda potentiella resultat från detta.

I Delaunay Triangulation algoritmen som implementerades i detta arbete valdes det att använda en *incremental algorithm*. Detta trots att det fanns alternativa algoritmer med lägre teoretisk komplexitet. Anledningen var tidsbrist och för att inte riskera en felaktig implementation, dock kan detta leda till ett vilseledande resultat. I ett framtida arbete där samma algoritmer implementeras kan det vara värt att testa en annan algoritm med lägre teoretisk komplexitet, såsom QuickHull.

6.2.2 Samhälleliga och etiska aspekter

PCG i helhet skulle kunna bidra till utveckling av utbildningsmiljöer och träningssimulering, dock när det kommer till generering av dungeons finns det främst ett stort problem. Den stora

samhälleliga frågan är spelberoende. Spelberoende har varit ett växande problem i samhället under de senare åren och det är i många fall i samband med spelgenren rollspel. Problemet är som störst bland ungdomar och barn. Anledningen att det är beroendeframkallande är att spelet egentligen aldrig tar slut och att man alltid kan göra sin karaktär starkare (Young, 2009). De algoritmer som presenteras i detta arbete har sin främsta användning inom samma genre även om de tekniskt sett skulle kunna användas för att skapa andra spel. Problemet med rollspel är att de aldrig tar slut och det faktum att man skulle kunna fortsätta spela hela tiden har en direkt koppling till detta arbete. När PCG likt den som är undersökt i detta arbete används i ett spel är tanken ofta att öka variationen. Ifall spelupplevelsen kontinuerligt uppdateras leder det till att en spelare inte tröttnar lika fort. Detta skulle då kunna komma att bidra till att en användare blir beroende av ett spel (Young, 2009).

Det har framkommit studier som talar för att spelindustrin och spelutvecklare borde ta ett större ansvar för problemtillväxten (Griffiths, Meerkerk, Schoenmakers, van de Mheen & van Rooij, 2009). I och med att detta arbete söker att utöka forskningen kring ämnet, kan detta anses bidra till ökningen av spelberoendet och därför ha en negativ inverkan på samhället. Därför ställs frågan ifall det är etiskt korrekt att publicera detta arbete när det inte garanterat tillför något positivt till samhället. Spel som utvecklats med hjälp av rumbaserad bangenerering har varit framgångsrika (Bidarra m.fl., 2014), dock trots detta har inte tekniken fått sitt stora genombrott. Som nämnt tidigare sökte detta arbete att utöka forskningen till området och bidra med viktig information till framtida spelutvecklare. Den positiva tillökningen som detta arbete tillför är då en breddad grund för spelskapare som är intresserade att använda PCG tekniker. Bortsett från denna punkt är den samhälleliga nyttan inte speciellt stor.

För att återgå till den viktigaste frågan, huruvida det är etiskt korrekt att publicera ett arbete som skulle kunna bidra till spelberoende, är det svårt att ge ett rakt svar på. Den viktiga frågan är ifall den positiva tillförseln är större än den negativa. Svaret på en fråga som denna har en aningen subjektiva och alla kommer inte besvara den på samma sätt. Ska man dock se det hela lite större så är det ju inte direkt PCG som står som den stora boven i tillväxten av spelberoende. Snarare är det hur ett spel formas, bland annat vilka spelregler och hur mycket spelaren uppmuntras fortsätta spela. Att inte skapa spel som direkt uppmuntrar till förlängda spelsessioner är inte heller den enda lösningen. En studie föreslog att spel borde innehålla fler varningstexter som informerar om problemet och uppmuntrar spelaren att byta aktivitet till och från (Griffiths m.fl., 2010). Vidare föreslår studien att spelarens speltid ska mätas genom spelet och om en spelare blir flaggad för att spela för mycket bör kontakt tas med denna och information om hjälp ska ges. På så vis kan man fortsätta utveckla spel som skulle kunna uppfattas som beroendeframkallande med hjälp av metoder som de nämnda i detta arbetet. Det viktiga är att användarna av produkten är medvetna om riskerna och vart den söker hjälp.

6.3 Framtida arbeten

Det första och viktigaste framtida arbetet hade varit att genomföra en studie på spelarperspektivet. Det vill säga att genomföra samma studie fast där fokus är på upplevelsen hos en användare. Att se de tekniska skillnaderna har sina fördelar, men som nämndes tidigare kan man inte helt avgöra vilken algoritm som är att föredra. Utan en studie som talar för hur skillnaderna i egenskaper mellan algoritmerna påverkar användarens upplevelse, kommer inte denna studie garanterat kunna hjälpa i avgörandet av vilken algoritm som är att föredra.

Vidare så nämndes det tidigare att denna studie är direkt anknuten till 3D-miljögenerering. Ett framtida arbete hade kunnat vara att implementera samma algoritmer och testa ifall likartade resultat uppkommer på generering av enbart dungeonkartorna. Meningen med detta hade varit att ge arbetets resultat mer bredd samt för att validera arbetets resultat inför spelindustrin. Ifall det visar sig att samma resultat inte skulle uppkomma för enbart generering av kartor, så skulle det kunna betyda att resultatet från denna studie är nischat till 3D-miljön och genereringen av mer fullständiga dungeons.

Ytterligare framtida arbete hade varit att genomföra samma undersökning i en annan testmiljö eller med större antal tester. Exempelvis med en mer kraftfull dator att genomföra testerna på eller ifall man genomför fler tester på varje algoritm och rumsantal. Den undersökning kan, som nämndes tidigare, ge ett mer trovärdigt och användbart resultat. Att köra fler tester är direkt kopplat till ökad trovärdighet i resultatet. Att testa undersökningen i en annan testmiljö hade kunnat utöka undersökningens användbarhet, eftersom ett likartat resultat hade talat för en mer generell användbarhet.

Ett slutgiltigt framtida arbete kan vara att implementera en annan algoritm än *incremental*. Algoritmen incremental hade inte den lägsta komplexiteten enligt en refererad studie (Su & Drysdale, 1997). Det hade därav varit intressant att testa att implementera den algoritm som Su och Drysdale visade vara snabbast, istället för den som användes i detta arbetet. Även om skillnaderna i tid mellan BSP och DT inte berodde på millisekunder så finns ändå möjligheten av ett annat resultat.

Referenser

10tons Ltd. (2016) *Neon Chrome* (Version: 1.0) [Datorprogram]. Tillgänglig: http://neonchromegame.com/

Andersson, F. (2013) *Procedurellt genererade provinskartor för strategispel: En jämförelse mellan Voronoidiagram och Cellular Automata*. Kandidatuppsats, Datalogi. Skövde: Högskolan i Skövde. urn:nbn:se:his:diva-8185

Artificial Intelligence Design. (1980) *Rogue* (Version okänd) [Datorprogram]. Tillgänglig: https://archive.org/details/msdos_Rogue_1983

Berg, M. de (2010). *Computational geometry: algorithms and applications (3rd edition)*. Cham: Springer.

Blackpawn. (Okänt datum) Point in triangle test. *Blackpawn* [blogg]. http://blackpawn.com/texts/pointinpoly/ [2019-04-08]

Blizzard Entertainment. (2004) *World of Warcraft* (Version 8.1.0) [Datorprogram]. Tillgänglig: https://worldofwarcraft.com/en-us/

Blizzard Entertainment. (2012) *Diablo III* (Version 2.6.4) [Datorprogram]. Tillgänglig: https://eu.diablo3.com/en/

Blizzard North. (1996) *Diablo* (Version: 1.0) [Datorprogram]. Tillgänglig: https://archive.org/details/Diablo_1996_Blizzard

Booth, M. (2009) The AI systems of Left 4 Dead. *Keynote: Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. Stanford, California, USA.

Browne, C., Stanley, K., Togelius, J. & Yannakakis, G. (2001) Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transaction on*. 3 (3), s. 172-186.

Dinh, Phi. (2013) Procedural Dungeon Generation Algorithm Explained. *Reddit* [forum], 14 februari. https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/

Driver, S. (2010) A Bitwise Method For Applying Tilemaps. *saltgames* [blogg], 26 januari. https://web.archive.org/web/20130410235113/http://www.saltgames.com/2010/a-bitwise-method-for-applying-tilemaps/ [2019-04-04]

Dwyer, G. Williams, K. B. (1999) *Portable Random Number Generators*.

Evertsson, J. (2014) *Rumsbaserad bangenerering: En jämförelse av procedurella tekniker*. Kandidatuppsats, Datalogi. Skövde: Högskolan i Skövde. urn:nbn:se:his:diva-9431

Erkan, A. (2018) The educational insights and opportunities afforded by the nuances of Prim's and Kruskal's MST algorithms. *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. Larnaca, Cyprus, ACM. s. 129–134.

Firemana. (2013) What's an efficient way to find if a point lies in the convex hull of a point cloud?. *Stackoverflow* [forum], 4 juni. http://stackoverflow.com/a/16909956 [2019-04-08]

Gamblin, T. (2009). When should I use Kruskals as opposed to Prim (and vice versa)? *Stackoverflow* [forum], 28 juli. https://stackoverflow.com/a/1195932 [2019-04-09]

Helge, A. (2018) *Procedurell generering - rum och korridorer: En jämförelse av BSP och Bucks algoritm som metoder för procedurell generering av dungeons*. Kandidatuppsats, Informationsteknologi. Skövde: Högskolan i Skövde. urn:nbn:se:his:diva-15288

Kastbjerg, E., Schedl, D., Togelius, J. & Yannakakis, G. (2011) What is procedural content generation? Mario on the borderline. *PCGames `11: Proceedings of the 2nd International Workshop on Procedural Content Generation in Games. New York*, NY, USA, ACM.

Kendrick, J. S., Parker, R. A. & Williamson, D. F. (1989) The box plot: Visual method to interpret data. *Annuals of International Medicine*. 110(11), s. 916-921.

Lou, L., Yin, H., Cai, W., Lees, W., Othman, N. B. & Zhou, S. (2014) Towards a data-driven approach to scenario generation for serious games. *Computer animation & virtual worlds*. 25(3-4), s. 393-402.

Maxis. (2008) *Spore* (Version: 5.1) [Datorprogram]. Tillgänglig: https://www.spore.com/

Microsoft Game Studios. (2005) *Forza Motorsport* (Version: 1.0) [Datorprogram].

Millington, I. Funge, J. (2009). *Artificial Intelligence for Games*. 2. uppl., Massachusetts: Elsevier. ISBN: 978-0-12-374731-0

Mojang. (2011) *Minecraft* (Version: 1.13.2) [Datorprogram]. Tillgänglig: https://minecraft.net/sv-se/

Mossmouth. (2008) *Spelunky* (Version: 1.1) [Datorprogram]. Tillgänglig: http://www.spelunkyworld.com/

Park, S. K., Miller, W. (1988) Random number generators: Good ones are hard to find. *Communications of the ACM*. 32 (10), s. 1192-1201.

Phigames. (2014) *TinyKeep* (Version: 1.0) [Datorprogram]. Tillgänglig: http://tinykeep.com/

Runic Games. (2009) *Torchlight* (Version: 1.15) [Datorprogram]. Tillgänglig: https://www.runicgames.com/torchlight/

Santamaría-Ibirika, A., Cantero, X., Salazar, M., Devesa, J., Santos, I., Huerta, S. & Bringas, P. G. (2013) Procedural approach to volumetric terrain generation. *The Visual Computer*. 30(9), s. 997-1007.

Shaker, N., Togelius, J., Nelson, M. (2016) *Procedural Content Generation in Games*. Cham: Springer.

Short, T., Adams, T. (2017) *Procedural Generation in Game Design*. Miami: A K Peters/CRC Press.

Skorpio. (2012) 3D Dungeon Tileset. *OpenGameArt* [forum], 31 march. https://opengameart.org/content/3d-dungeon-tileset [2019-03-20]

Su, P. Drysdale, R. L. S. (1997). A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry*. 7 (5-6), s- 361-385. doi:10.1016/s0925-7721(96)00025-9

Unity. (2018) *Scripting API*. https://docs.unity3d.com/ScriptReference/ [2019-03-25]

Valve Corporation. (2008) *Left 4 Dead* (Version: 2.) [Datorprogram]. Tillgänglig: http://www.l4d.com/blog/

Van der Linden, R. Lopes, R. Bidarra, R. (2014) Procedural generation of dungeons. *Computational Intelligence and AI in Games, IEEE Transacrions on*. 6 (1), s. 78-89.

Wild Shadow Studios. (2011) *Realm of the Mad God* (Version X.31.3.0) [Datorprogram]. Tillgänglig: https://www.realmofthemadgod.com/

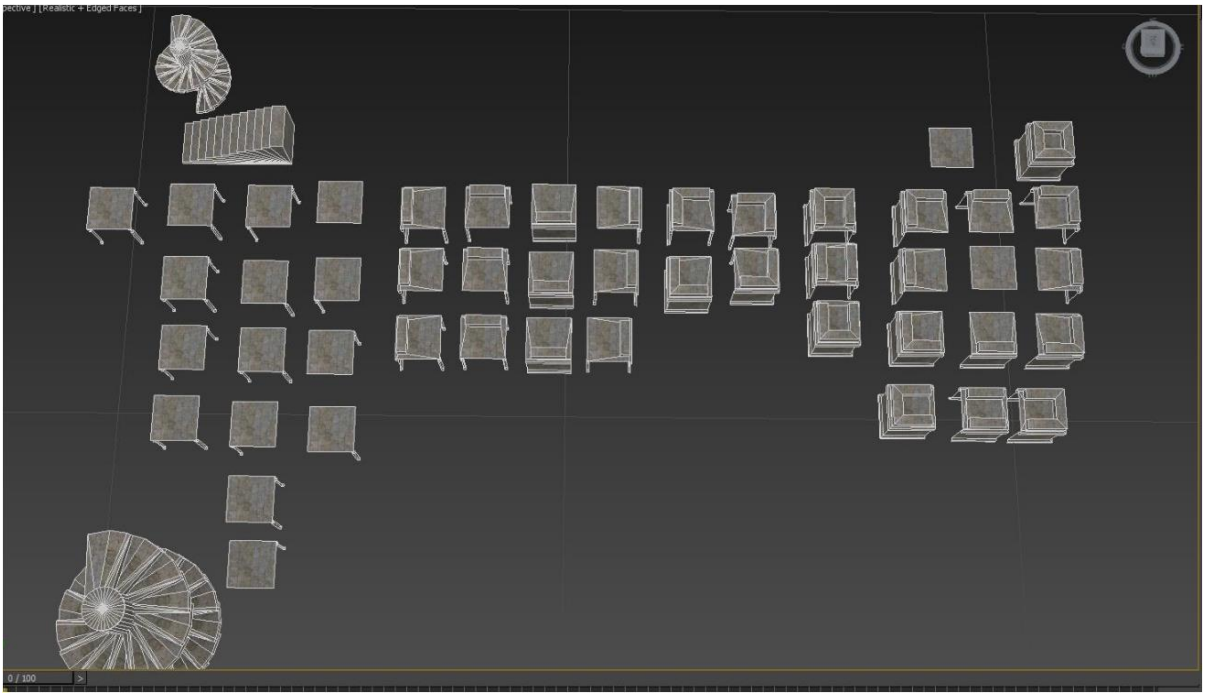
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A. (2012) *Experimentation in Software Engineering*. Springer.

Young, K. (2009) Understanding online gaming addiction and treatment issues for adolescents. *The American Journal of Family Therapy*. 37 (5), s. 355-372.

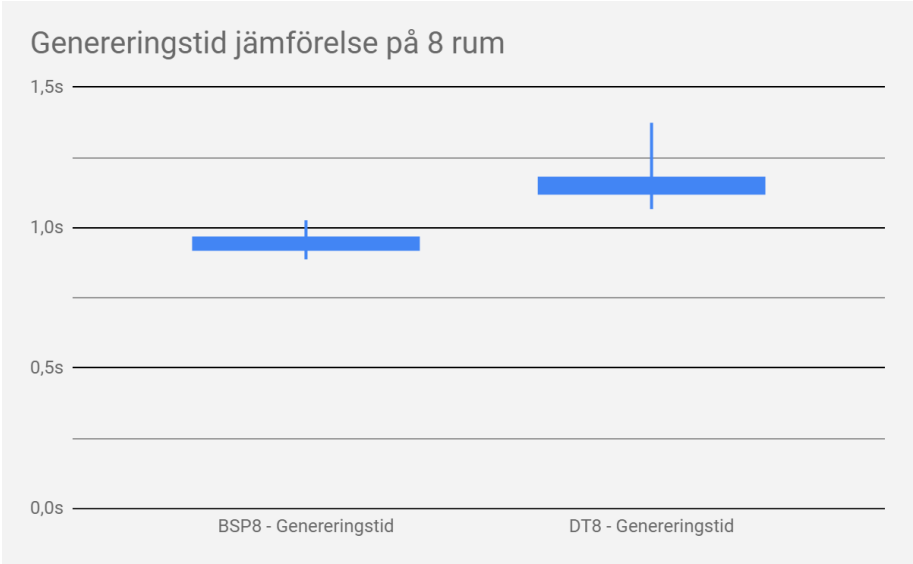
Zhao, W., Ge, Y., Qu, W., Zhang, K., Sun, X. (2017) The duration perception of loading applications in smartphone: Effects of different loading types. *Applied Ergonomics*. 65, s. 223-232.

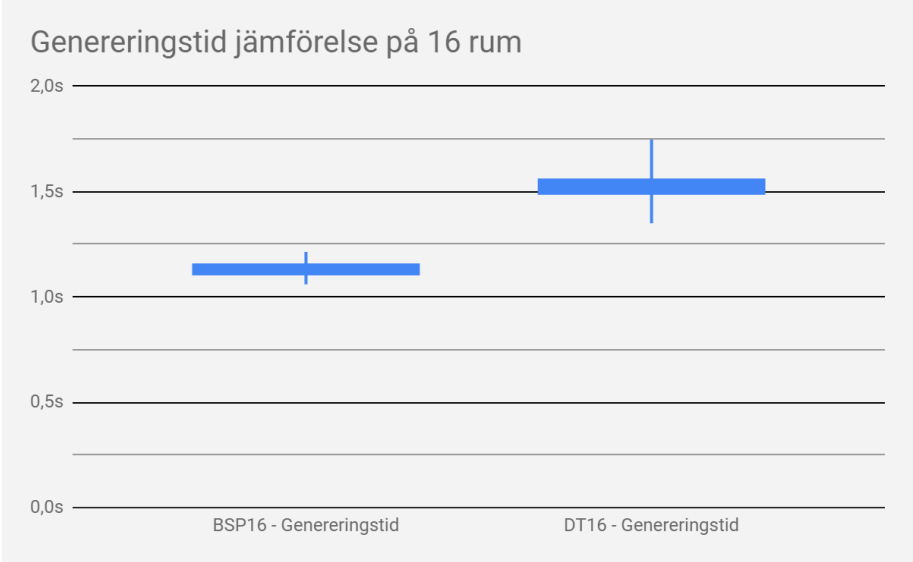
Appendix A - Dungeonuppsättning

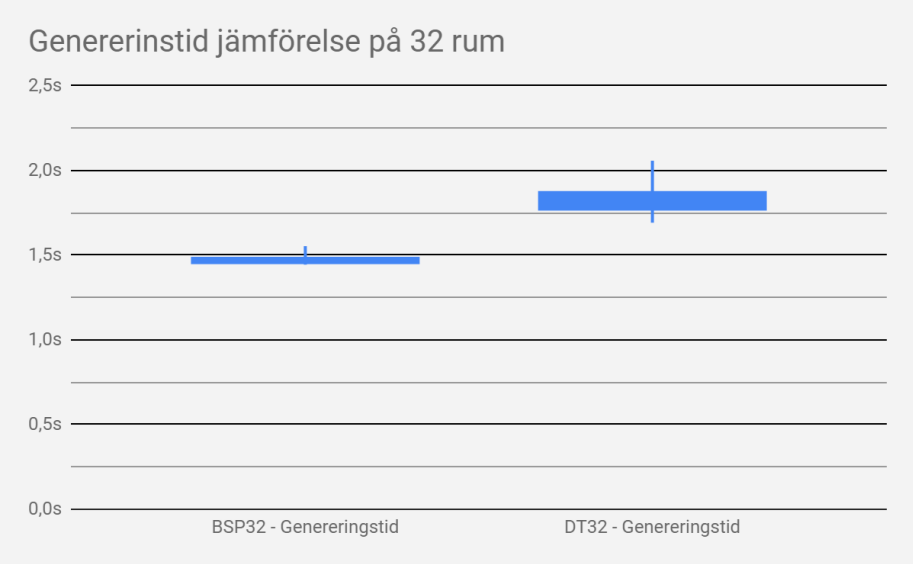
Dungeonuppsättningen som användes vid generering av dungeonkartorna hämtades från http://opengameart.org/content/3d-dungeon-tileset (20-03-2019) Stort tack till OpenGameArt.org medlemmen 'Skorpio' för att ha skapat denna dungeonuppsättning och gett ut den gratis och till Liam Bower för att ha exporterat de individuella modellerna.

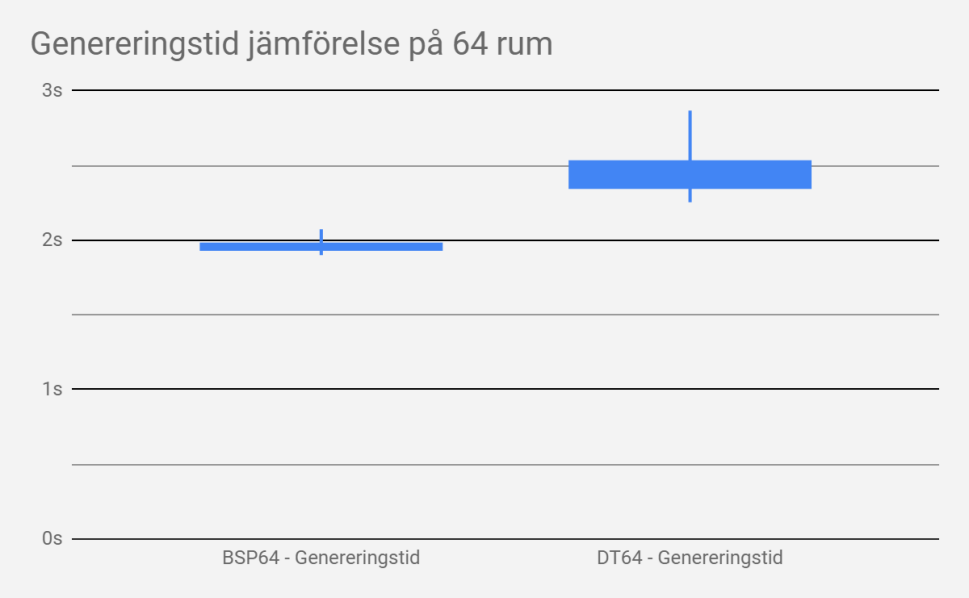


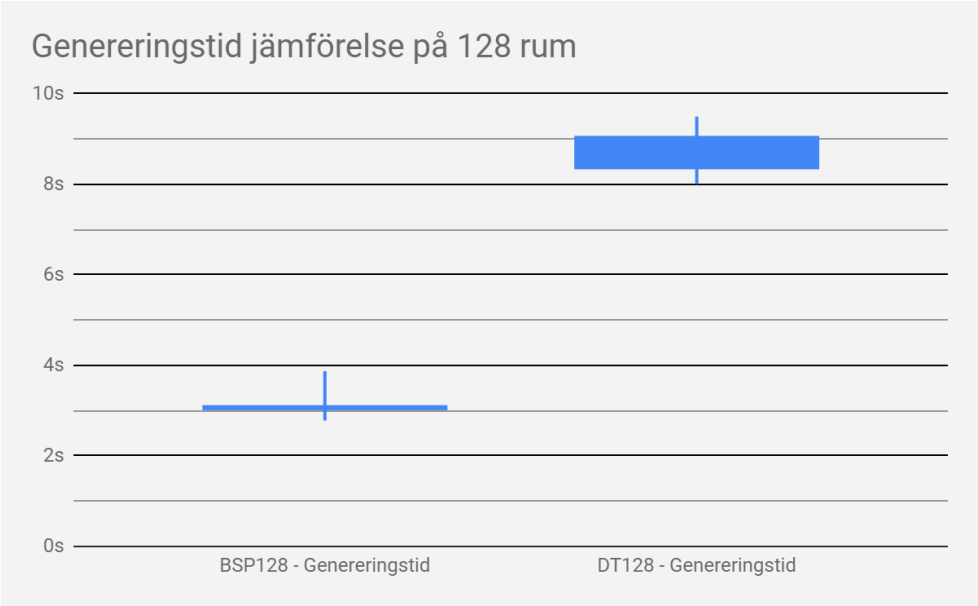
Appendix B - Genereringstid resultat



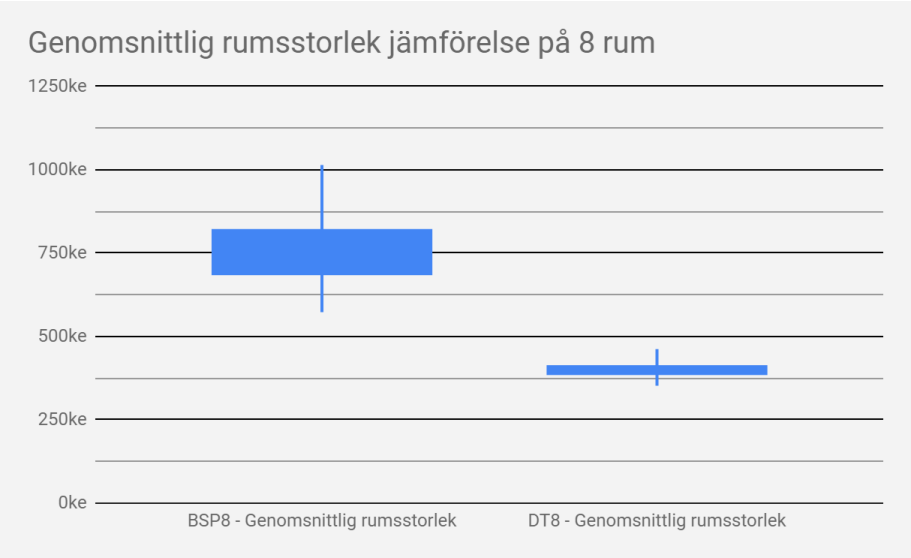


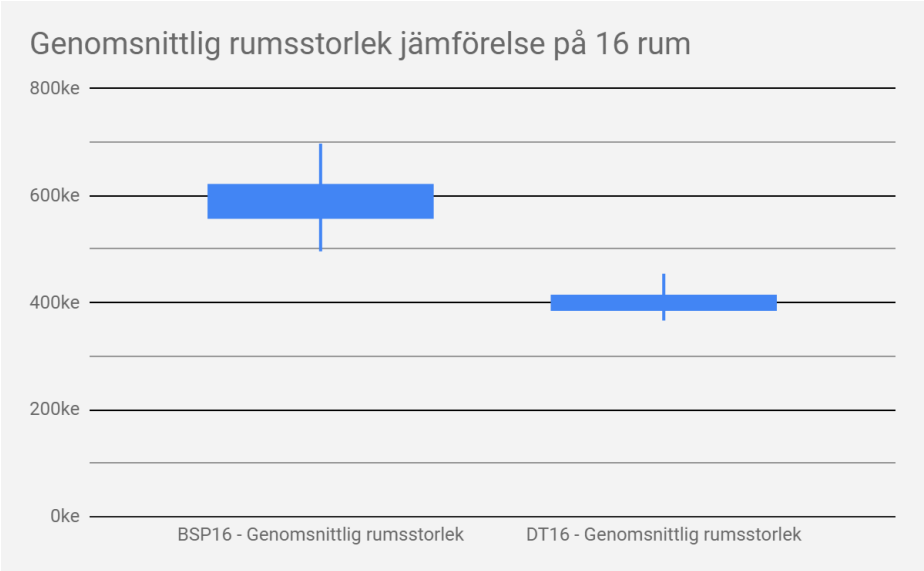


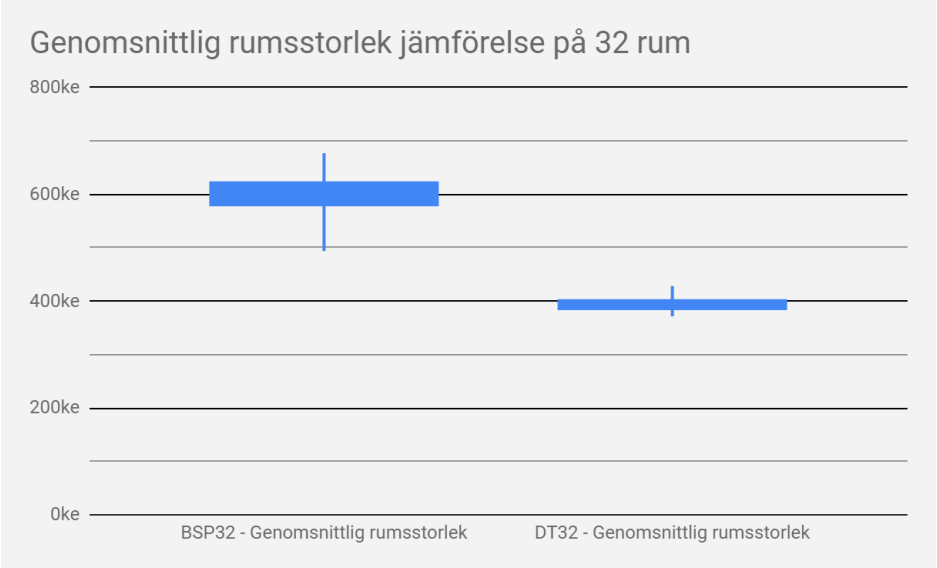

Genereringstid jämförelse på 64 rum
3s
2s
1s
0s
BSP64 - Genereringstid
DT64 - Genereringstid

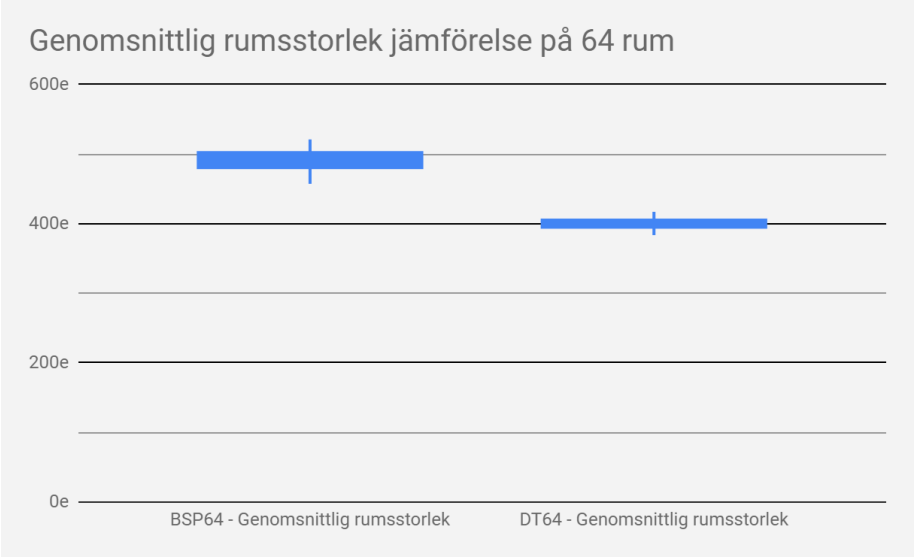

Genereringstid jämförelse på 128 rum
10s
8s
6s
4s
2s
0s
BSP128 - Genereringstid
DT128 - Genereringstid

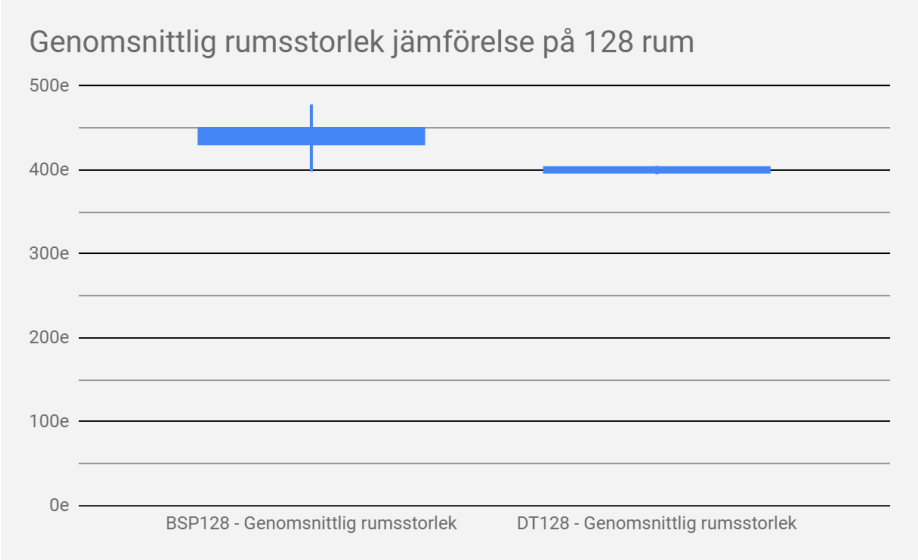
Appendix C - Genomsnittlig rumsstorlek resultat



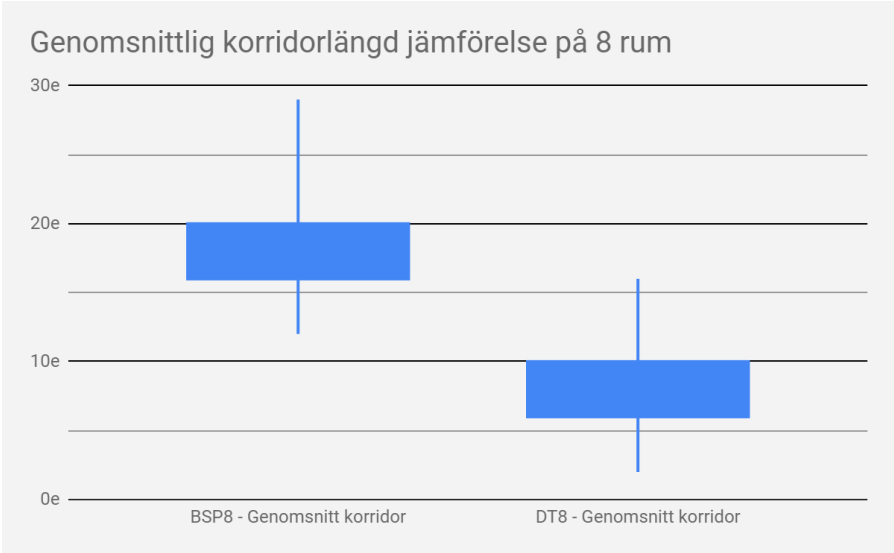


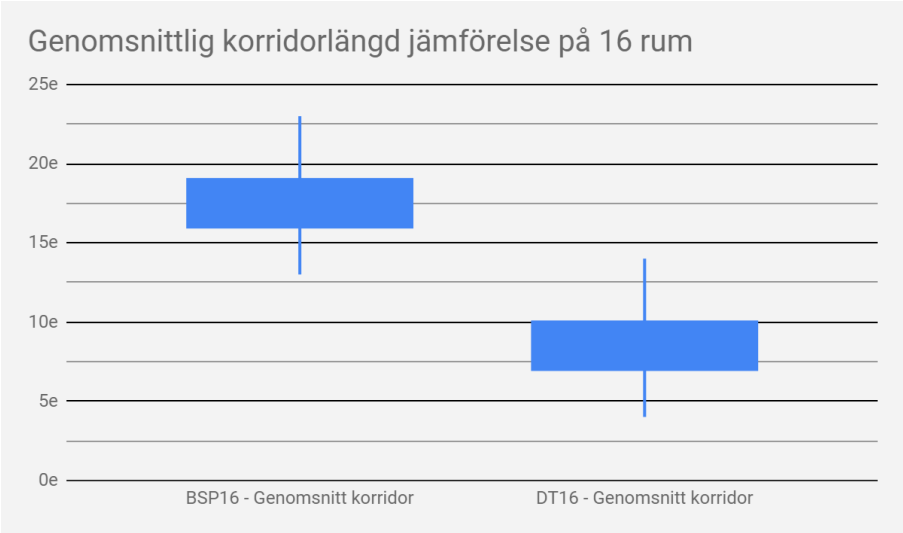


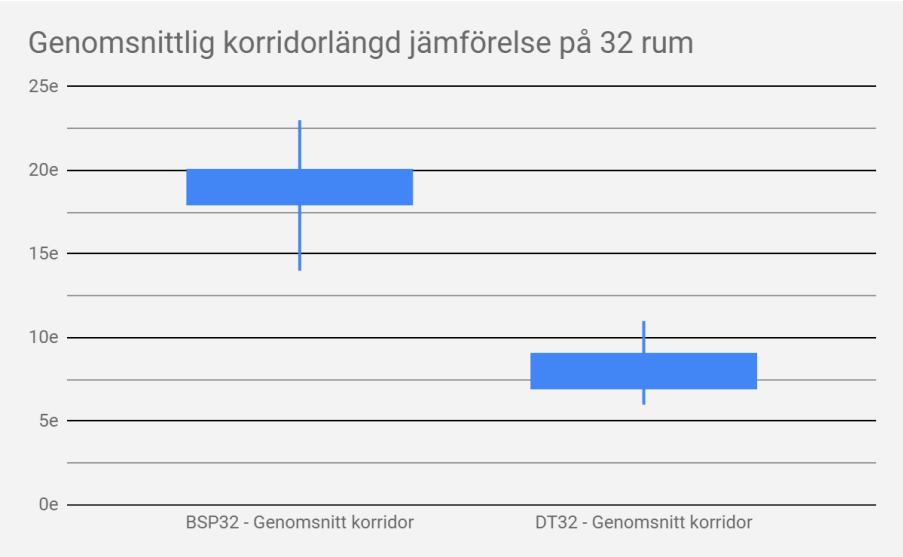

Genomsnittlig rumsstorlek jämförelse på 64 rum
600e
400e
200e
0e
BSP64 - Genomsnittlig rumsstorlek
DT64 - Genomsnittlig rumsstorlek

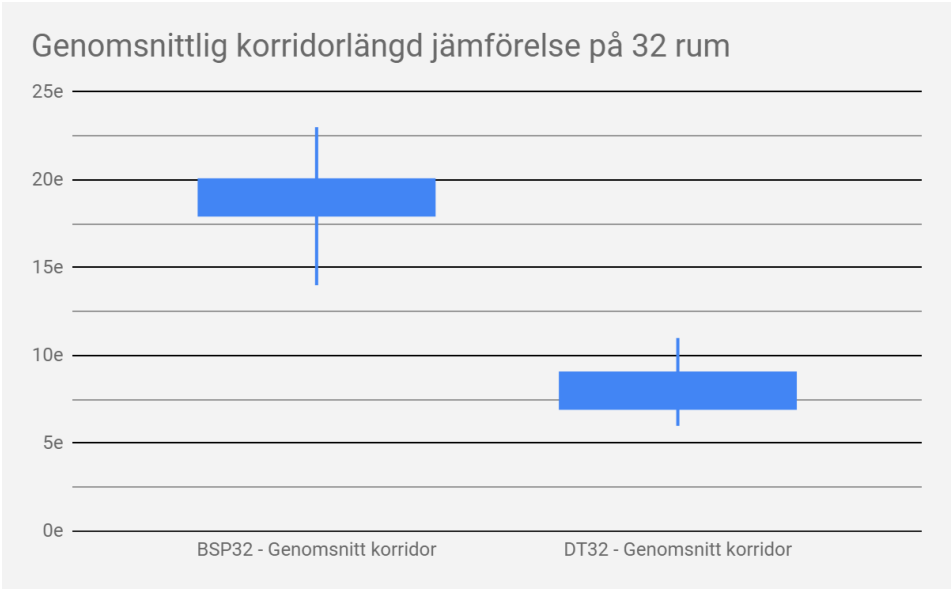

Genomsnittlig rumsstorlek jämförelse på 128 rum
500e
400e
300e
200e
100e
0e
BSP128 - Genomsnittlig rumsstorlek
DT128 - Genomsnittlig rumsstorlek

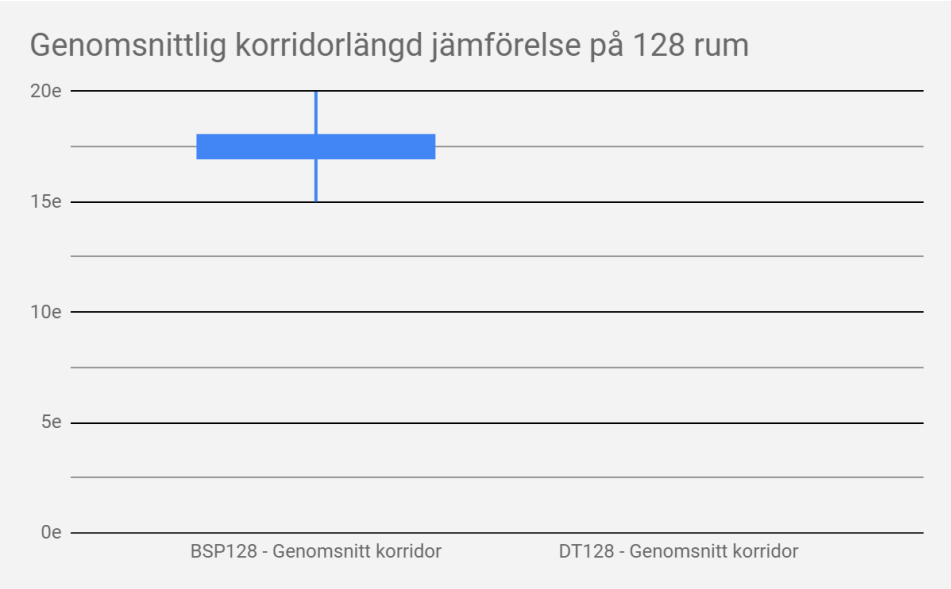
Appendix D - Genomsnittlig korridorlängd resultat



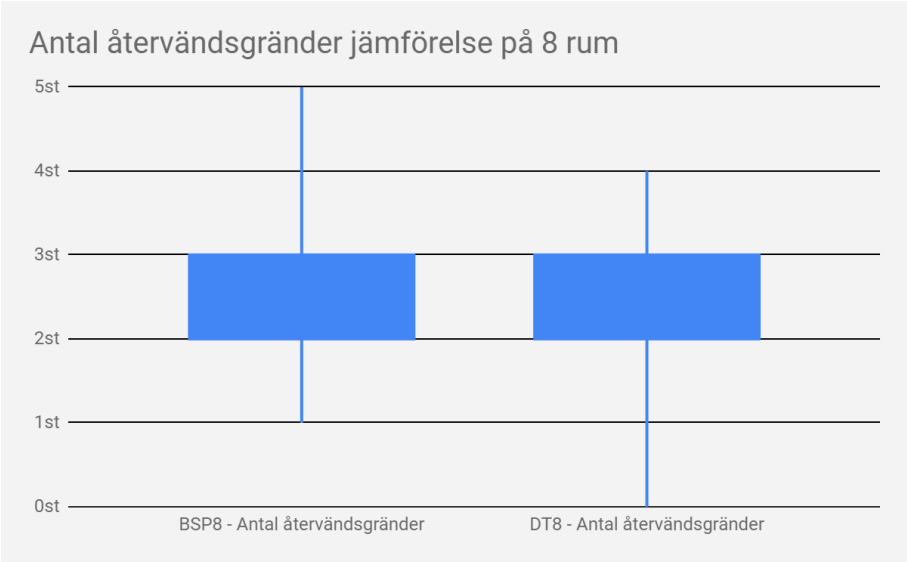


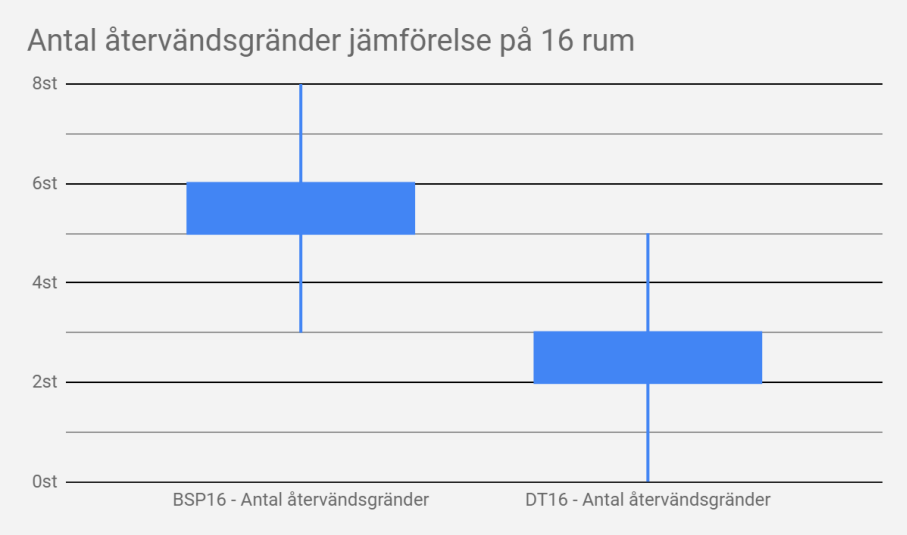


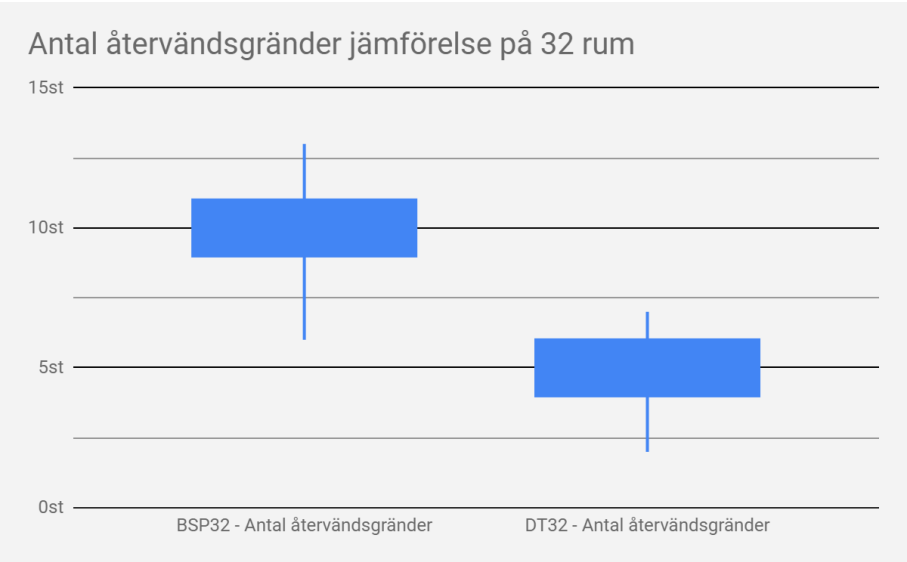

Genomsnittlig korridorlängd jämförelse på 32 rum
25e
20e
15e
10e
5e
0e
BSP32 - Genomsnitt korridor
DT32 - Genomsnitt korridor

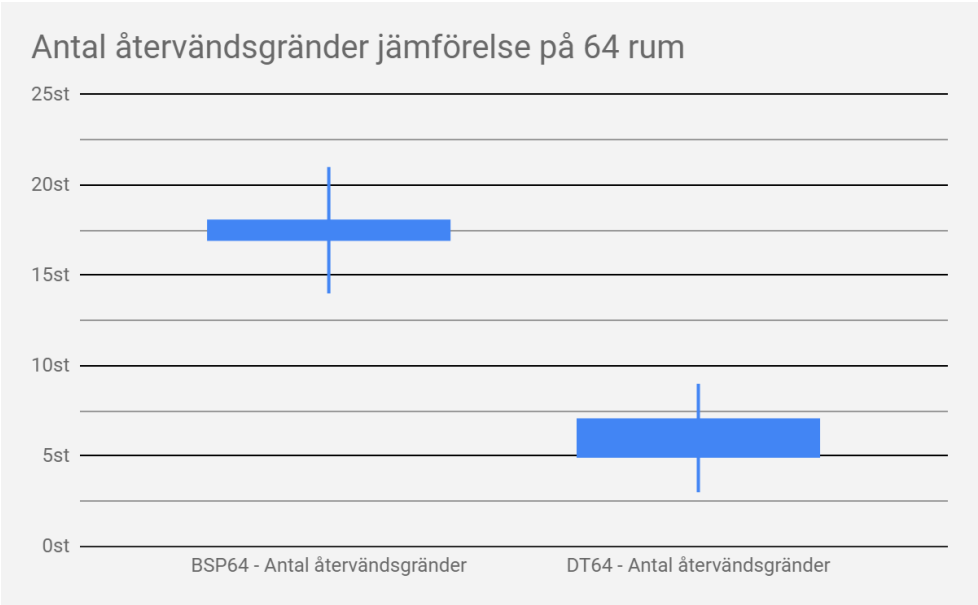

Genomsnittlig korridorlängd jämförelse på 128 rum
20e
15e
10e
5e
0e
BSP128 - Genomsnitt korridor
DT128 - Genomsnitt korridor

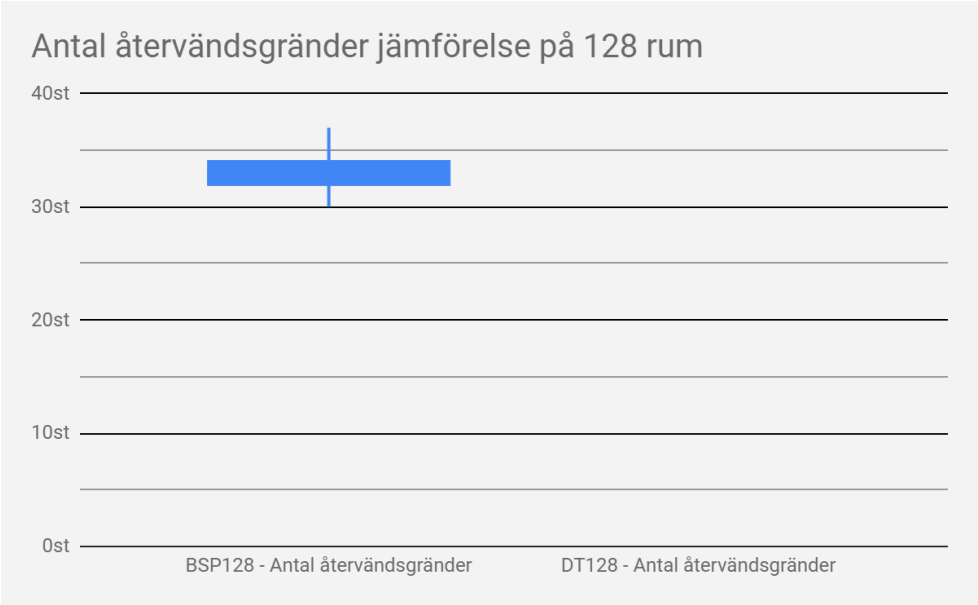
Appendix E - Antal återvändsgränder resultat








Antal återvändsgränder jämförelse på 64 rum
25st
20st
15st
10st
5st
0st
BSP64 - Antal återvändsgränder
DT64 - Antal återvändsgränder


Antal återvändsgränder jämförelse på 128 rum
40st
30st
20st
10st
0st
BSP128 - Antal återvändsgränder
DT128 - Antal återvändsgränder