Examensarbete

HÖGSKOLAN I SKÖVDE

KAN MAN ANVÄNDA AGENTER FÖR ATT SKAPA EN SVENSK TALLSKOG MED REALISTISK DISTRIBUTION?

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2018

Simon Heinonen

Handledare: Tom Ekblom
Examinator: Mikael Johannesson

Sammanfattning

Detta arbete undersöker hur vida agenter inom artificiell intelligens kan realistiskt distribuera en svensk tallskog. Landskapets mark där simuleringen kommer att ta plats är formad efter en referensskogs mark. I miljön kommer det även att finnas vindzoner som hjälper agenterna att färdas runt och sprida flera träd.

Resultatet visar att det är möjligt att skapa en svensk tallskog med agenter då simuleringen hade kvaliteter som liknar referensskogens. De kvaliteter som undersökts är täthetszoner och vart dessa hamnade. Simuleringen skapade täthetszoner för träden som var lika i antal som de som kunde mätas i referensskogen. Det gick att dra slutsatsen att zonerna inte behövde ligga exakt som i referensskogen. Med det sagt är agenterna dock inte bra för att återskapa en skog efter en referensbild.

I framtida arbeten skulle det vara intressant att kolla hur agenter kan återskapa skogar med flera trädarter samt annan vegetation. Marken skulle då kunnat få mer uppmärksamhet och dess tillstånd för att avgöra hur vida träd och andra växter kan växa där.

Nyckelord: Autonoma agenter, Artificiell intelligens, Procedurell generering, Landskap, Topologi.

Innehållsförteckning

1 Introduktion ... 1

2 Bakgrund ... 2
2.1 Positionering ... 2
2.2 Autonoma intelligenta agenter ... 4
2.3 Procedurell generering ... 5

3 Problemformulering ... 7
3.1 Metodbeskrivning ... 7
3.2 Problematisering ... 9

4 Relaterad/tidigare forskning ... 10
4.1 Prestanda ... 10
4.2 Population ... 10

5 Genomförande ... 12
5.1 Arbetets designval ... 12
5.2 Problem med designval ... 12

6 Utvärdering ... 14
6.1 Presentation av undersökning ... 14
6.2 Analys ... 22
6.3 Slutsatser ... 23

7 Avslutande diskussion ... 25
7.1 Sammanfattning ... 25
7.2 Diskussion ... 25
7.3 Framtida arbete ... 25

Referenser ... 27

1 Introduktion

Denna uppsats kommer att omfatta autonoma intelligenta agenter (AIA) och procedurell generering som grund för att utveckla en datorsimulerad spridning av träd. Uppsatsen kommer att omfatta vad AIA egentligen är och hur tekniken kan användas. I kapitlet procedurell generering kommer det att handla om tre tekniker för att skapa landskap. Samt hur dessa kan användas för att positionera ut träd och växter i en spelmiljö.

Allt eftersom tiden går ökar spelens standarder för hur mycket innehåll det finns. Innehåll som terräng är viktigt för vissa spelare då det ska gå att spela om och kunna upptäcka nya saker. I den verkliga världen är en skog full av träd med positioner som tilldelats genom slump. Denna slump grundar sig i olika faktorer som påverkat fröets resa som till exempel vind. Träd utsöndrar frön som har olika sätt att spridas, äppelträdets metod för att sprida frön är i form av fallfrukt. Trädets frukter kommer att locka till sig djur som hjälper spridningen. Vissa lönnträd har ett sätt att sprida sina frön långt utan att göra sig av med mycket energi. Dessa lönnträd har skapat löv som är bra på att segla i luften och fördriva sina frön långt bort med hjälp av vinden.

AIA kommer att användas för att agera som frön från träd. Dessa agenter kommer att flyga runt med hjälp av vindzoner för att göra spridningen av träden möjlig. När ett frö har landat får den en chans att gro till ett träd. Denna chans är grundad i hur många närliggande träd det finns, då flera träd ger en mindre chans att gro. Det har legat mycket fokus på hur distributionen kommer att bli efter en tid av spridning. Spridningen av träden har varigt influenserat av en artikel, *Random Scattering: Creating Realistic Landscapes*, skriven av West (2008) som handlar om hur man skapar realistiska landskap. West (2008) förlitar sig på slumptal inför sin generering, i denna uppsats kommer spridningen att använda sig av element från verkligheten som till exempel vind.

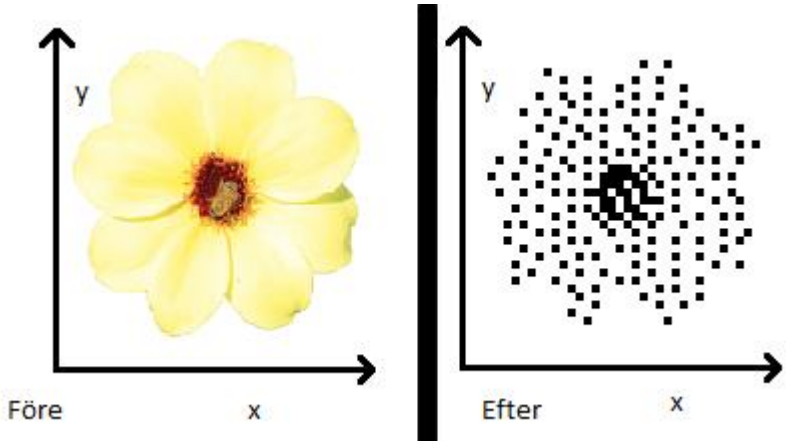
2 Bakgrund

Detta kapitel handlar om positionering av träd, autonoma intelligenta agenter och procedurell generering. Det finns olika metoder för att placera ut träd på en karta, många grundas i slump och kriterium om var träden blir placerade. Många saker kan anses vara en autonom agent, till exempel en kökstermometer eller en avancerad mekanisk robot. Manuella arbetsuppgifter som tar lång tid kan utföras fortare med hjälp av algoritmer. På så sätt kan man spara mycket tid och resurser.

2.1 Positionering

I spel kan man sträva efter att få en skog som påminner och liknar en skog från den verkliga världen. Det finns många olika sätt att placera ut växter i en spelmiljö, ofta är metoderna för att positionera ut träd och växter baserade på slumpade tal. Det finns även de algoritmer som använder efterbehandling för att skapa punkter som bestämmer positioner för träden. Dapper (2003, s.27) menar att "One algorithm that produces very good results for plant distributions is the Floyd-Steinberg Algorithm."

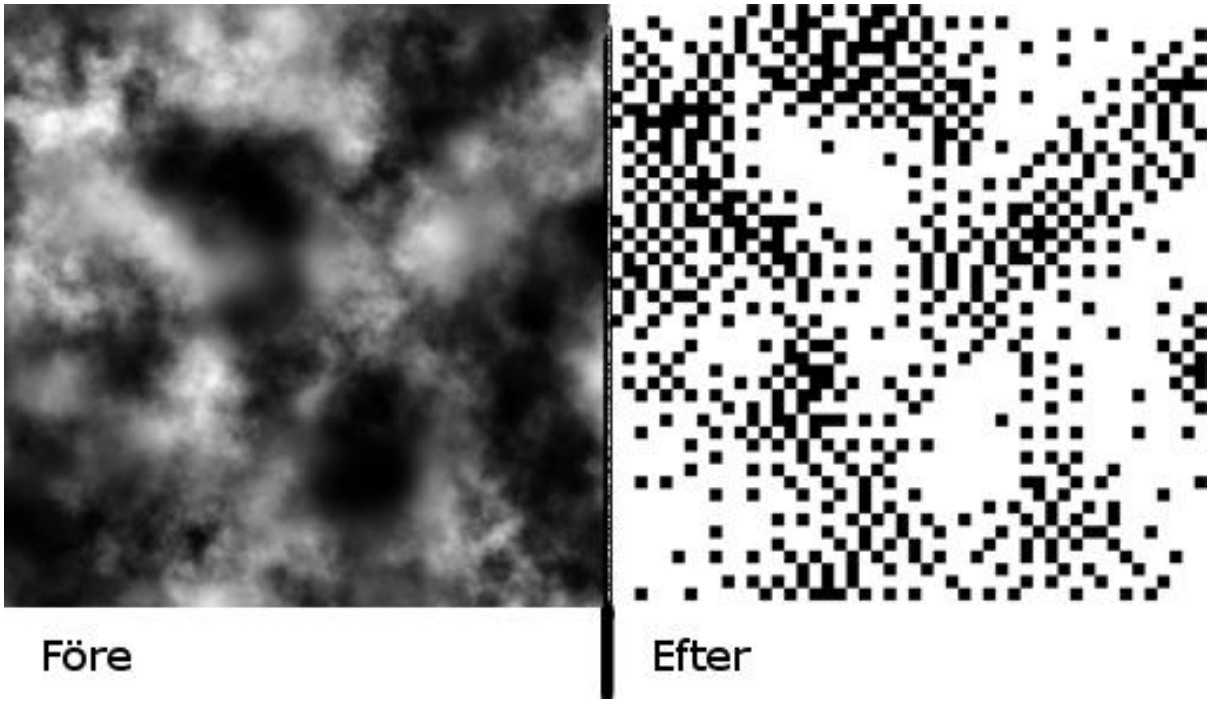
Floyd-Steinberg algoritm används till att reducera färger i bilder och kan användas till att positionera ut träd. Det som händer är att algoritmen modifierar alla pixlar i en bild genom att använda närliggande pixlar för att avgöra färgen för den nuvarande pixeln. Algoritmen reducerar antalet färger i bilden så att det bara finns svarta och vita pixlar kvar. Dessa svarta pixlar som finns kvar på bilden representerar då positioner för växter. Denna teknik arbetar i två dimensioner, x och y. I Figur 1 syns en bild på en blomma som sedan har blivit efterbehandlad med hjälp av Floyd-Steinbergs teknik. En spelbana är ofta i två eller tre dimensioner, efterbilden bestämmer positioner i x och y axeln. Efterbilden är en kartläggning på hur träden kommer att vara placerade. Detta är en illustration på hur algoritmen kan användas, ofta används egenmålade bilder för att sprida träd.



Figur 1 Bild som blivit behandlad med Floyd-Steinbergs metod

En annan användbar metod för att placera ut träd kallas *Dart Throwing* (Dapper, 2003) Algoritmen placerar ut helt slumpade punkter på till exempel en karta där träd kan positioneras. Träden får positionen enligt ett visst kriterium för till exempel hur nära andra träd befinner sig.

En *Fractal-based* metod för att skapa landskap påstår Shaker, Togelius och Nelson (2016, s.64) att det ger ett mer tidseffektivt alternativ snarare än estetiskt ”fractal-based methods are usually evaluated in term of efficiency rather than the aesthetic features of the terrains generate”. Fractal-based är ett sätt att försöka likna realistiska landskap genom att en algoritm böjar med en linje mellan två punkter för att sedan bygga från mitten av linjen. Den nya punkten från mitten skiftas av ett slumpat tal vertikalt. Denna process fortsätter då det skapas flera linjer mellan nya punkter och skapar en större terräng med fler detaljer. Detta är även ett sätt som går att användas för att bestämma positioner som är grundad i slumpade tal. Detta görs tillsammans med den föregående tekniken, Floyd-Steinbergs metod. Med hjälp av fractal-based metoden går det att skapa en altitud karta för terrängen. I Figur 2 är en altitud karta skapad av en fractal-based metod, illustrationen kan även kallas för ’height map’. Efterbilden är när bilden blivit behandlad med Floyd-Steinbergs metod för att få fram positioner för träd på kartan.



Figur 2 Karta för altituden över en terräng som sedan blivit efterbehandlad.

När träd positioneras ut är det viktigt att inte träd placeras för nära varandra, eftersom det anses vara onaturligt att små samlingar av träd förekommer i naturen. I verkligheten kan man säga att det finns så kallade ’dead zones’ runt träd. Dessa zoner bildas eftersom det finns begränsade resurser som ett träd behöver för att växa. Ett exempel på en sådan resurs är solljus. West (2008) påstår:

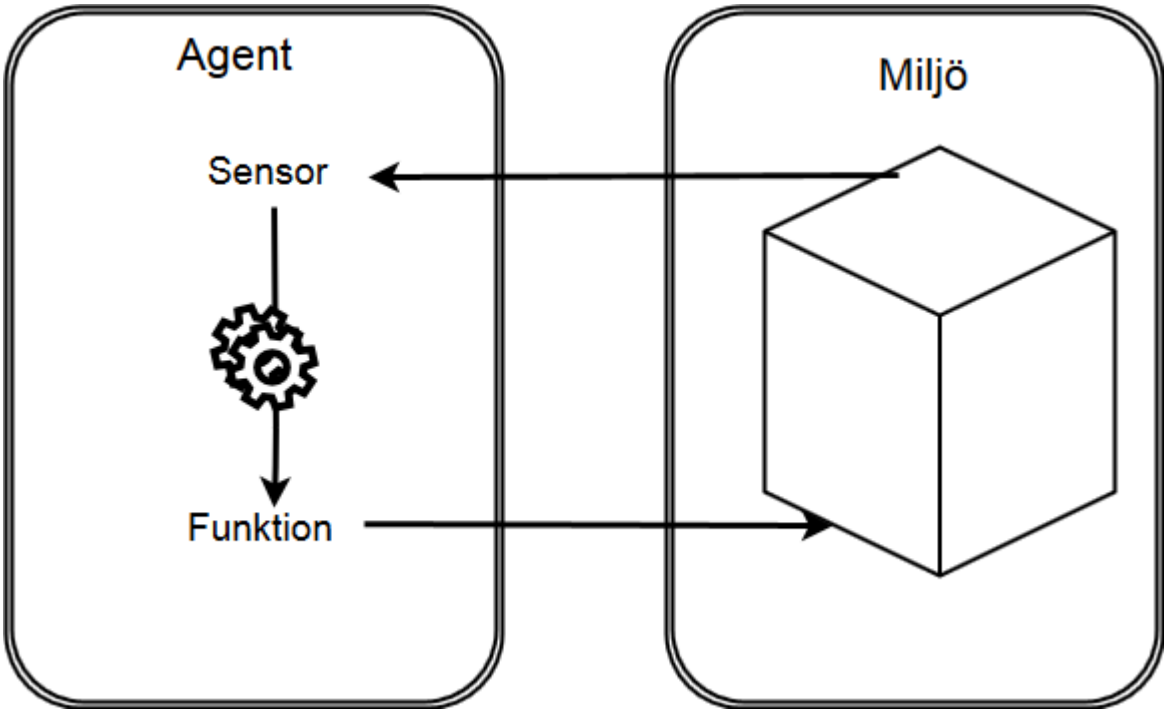
> “[...] obviously in real-life, trees do not have "kill zones" surrounding them. Rather they have something more like an inhibit field, where the successful growth of a tree is exponentially more unlikely the closer it is to other trees. The success of a tree is a function of the sum of all the other trees' inhibit fields at that point.”
>
> West, 2008, s.2

2.2 Autonoma intelligenta agenter

Automona intelligenta agenter (AIA), är ett verktyg som har ett brett användningsområde. De är som själv-agerande individer som arbetar för att uppnå sitt mål. Buckland (2005) definierar en autonom agent som:

> "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what is senses in the future."
>
> Buckland, 2005, s. 85

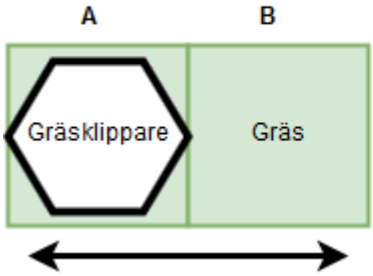
Agenter kan även lära sig och använda sig av kunskap för att uppnå sina mål. En agent kan vara väldigt komplex eller väldigt simpel beroende på arbetsuppgiften. Vad som helst kan anses vara en agent, ifall den kan läsa av en miljö och agera beroende av den. På samma sätt som en matematisk funktion fungerar även agenter. En agent tar indata som producerar utdata. I Figur 3 illustreras tanken bakom hur en agent fungerar.



Figur 3 Agenten läser av miljön och sedan agerar beroende på vad den läst av.

Indata i en matematisk funktion ger en viss utdata, på samma sätt beter sig agenten beroende på vad den läser in. Alla agentens beteenden är kartlagda till vad den läser in. Problemet är då att agenten kan läsa in oändligt många olika saker, vilket skulle leda till en kartläggning för oändligt många olika utfall. Det krävs en begränsning på inläsningen där man enbart bryr sig om det viktigaste och kartlägger dessa. Ett exempel på en agent är en självgående gräsklippare. För att göra det simpelt finns det en förbestämd miljö. I miljön finns det två rutor med gräs som gräsklipparen kan vara på. Agenten kan nu välja vad den ska göra; inget, klippa gräs, flytta till ruta A eller flytta till ruta B. Funktionaliteten hos gräsklipparen kan kartläggas med

hjälp av sensorer. I Figur 4 finns illustrationen för exemplet med gräsklipparen och de kartlagda funktionerna.



Funktioner	
Sensor	**Beteende**
A, Gräs	Klipp
A, Inget Gräs	Åk till B
B, Gräs	Klipp
B, inget Gräs	Åk till A

Figur 4 En gräsklippare i en miljö med två olika positioner den kan vara på.

2.3 Procedurell generering

Procedurell generering definierat av Togelius, Kastbjerg, Schedl och Yannakakis (2011) som ”Procedural content generation (PCG) in games refers to the creation of game content automatically using algorithms.” De menar att innehåll skapat med hjälp av algoritmer anses vara PCG.

Johnson, Yannakakis och Togelius (2010) har skapat ett automatiserat grottsystem där en oändlig grotta genereras i ett tvådimensionellt plan. Det skapas först ett plan i mitten som är startplanet, sedan byggs det på med plan runt om startplanet. Tekniken de använder kallas cellulär automata med regler som förändrar omgivningen till antingen väggar, stenar eller mark. Genereringen ser även till att det alltid finns kopplingar i tunneln så att grottan inte har mark som är oåtkomliga.

Ibland används agenter för procedurell generering och är inget nytt för spelindustrin. Tidigare forskning inom procedurell generering med agenter har växter inte varit i fokus, den mängden forskning det finns är väldigt liten. Procedurell generering är bra eftersom det kan skapa till exempel landskap under en kort tid. För att skapa landskap manuellt krävs mycket handarbete vilket är kostsamt. I vissa fall har agenter använts till att skapa miljöer, eftersom att de ger kontroll åt utvecklaren. Shaker, Togelius, och Nelson (2016) skriver i samband med deras arbete inom terränggenerering:

> “The main advantage of software-agent approaches to terrain generation over fractal-based methods is that they offer a greater degree of control while maintaining the other desirable properties of PCG methods.”
>
> Shaker, Togelius, och Nelson, 2016, s.64

Med hjälp av agenter går det att specifikt bestämma vad som ska göras och hur.

Agenter kan även användas för att skapa en stad. Med hjälp av flera olika typer av agenter kan dessa arbeta med olika saker samtidigt för att tillsammans uppnå ett gemensamt mål. Lechner, Watson och Wilensky (2003) har skapat en stadgenerator med hjälp av agenter. Även i det här arbetet har agenter använts eftersom det ger mer kontroll. Lechner, Watson och Wilensky (2003, s.2) skriver ”Our model differs from the previous work mentioned in its usage of agents rather than an L-system.”. Agenterna i prototypen har simpla regler på hur de ska

bete sig på grund av att det skulle skapa färre konflikter mellan varandra. Det finns flera olika agenter som gör olika saker som, bygga vägar, placera ut hus och koppla lösa vägar. Dessa agenter interagerar inte med varandra utan bara med miljön.

> "Our agents interact directly with their environment, andonly indirectly with each other, i.e. by observing the effects of other agents on the environment."
>
> Lechner, Watson och Wilensky, 2003 s.2

3 Problemformulering

Kan man använda autonoma intelligenta agenter för att skapa en skog med realistisk distribution? Arbetet grundar sig i procedurell generering av en miljö där fokus ligger på träd i en svensk tallskog och dess distribution för att kunna besvara denna frågeställning. Träd har växt fram genom en spridning av frön från andra träd. Denna spridning är påverkad av vind som gör att frön får en kraft applicerad åt det håll vinden är riktad. Ett frö glider fram i luften tills att den landat vilket gör att simuleringen jobbar i en 3D miljö snarare än befintliga metoder (beskrivna i kapitel 2), som ofta bara använder sig av ett plan i X och Y-led vid spridning av träd.

Fröna som har skapats i miljön kan inte flyga runt hur de vill eftersom det finns höjdskillnader som hindrar framkomligheten. Spridningen är då mer begränsad i jakt efter en mer verklig fördelning av träd. Miljön för simuleringen är formad med hjälp av topologi för att efterlikna den befintliga platsen i verkligheten. Detta för att resultatet från artefakten ska kunna jämföras med mer rättvisa mot den verkliga platsen.

Det finns flera olika sätt att placera ut träd på en spelplan. Många av metoderna grundar sig i slumptal och kriterium för hur träden blir placerade. Den aspekten som inte tas tillvara på är hur det går till i verkligheten, att ett träd utsöndrar frön som tar sig runt spelplanen med hjälp av verkliga faktorer som till exempel vind. De befintliga metoder som nämns i kapitel 2.1, där träd positioneras ut genom bland annat texturer, blir inte påverkade av någon vindfunktion eller liknande.

3.1 Metodbeskrivning

Agenterna har agerat som frön för att kunna skapa flera träd. Alla frön skapas ifrån befintliga träd på spelplanen. Fröna har arbetat med regler om hur de får skapa flera träd. Reglerna är till för att kontrollera hur lätt träden växer runt andra träd. En viktig regel är en så kallad ’dead zone’ där ett frö har väldigt liten chans att växa nära ett annat träd. Den regeln samspelar med en annan ”chans regel” där ett frö kollar avstånd med närliggande träd. Chansen bestäms med hjälp av avståndet till närmsta grannträd, desto längre bort närmsta träd ligger, ju högra chans blir det att det tillkommer ett nytt träd. Frön som hamnat på marken och inte har närliggande träd har större chans att växa än de frön som har träd bredvid sig. Det växer sällan två träd väldigt nära varandra eftersom de kämpar om att få till exempel mest solljus. I detta fall har inte solljus beräknats eftersom det går att få liknande resultat genom att ha så kallade ’dead zone’ för träd.

Till en början finns det bara ett träd i mitten på spelplanen. När den har växt till hälften av sin maximala höjd, kommer trädet att börja generera frön. Dessa frön har skapats på någon position i eller runt om trädet med en slumpad riktning.

Varje frö börjar med att färdas i den slumpade riktningen som den blivit tilldelad. Sedan finns det andra dragningskrafter som bestämmer vart fröet hamnar som till exempel vind och sammanhållning. Det som sammanhållningen gör är att den ger liv åt agenterna så att de dras samman och grupperas.

Det finns flera vindzoner runt om på kartan som förändras med tiden. Dessa vindzoner har inte varit hyperrealistiska eftersom realistisk vind är mycket komplext och ur fokus. Med relativt simpla vindzoner går det att få fröna att förflytta sig med restriktion av befintliga träd

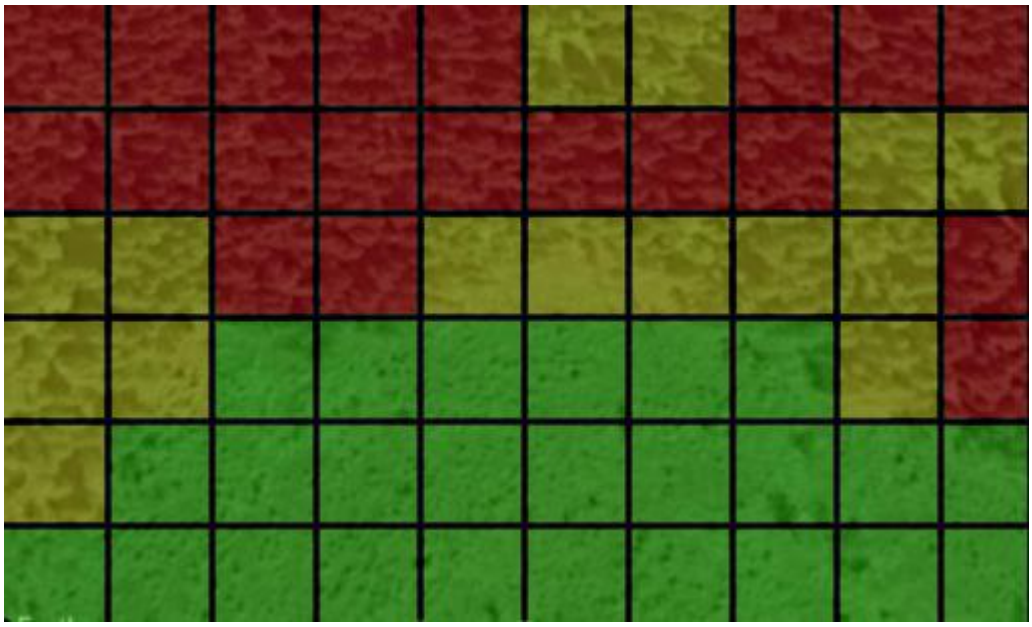
och terrängen som hinder. Eftersom att vinden har förändrats över tid blev distributionen av frön mer varierat. Problem som kunnat uppstå i och med att inte realistisk vind används, är att det inte skapar ett verkligt luftflöde som kunnat påverka distributionen på ett oväntat sätt.

Spelplanen har skapats i Unity 3D Engine där verktyg finns för att skapa terräng med hjälp av en 'height map'. Utifrån den verkliga miljön som jämförelsen kommer ske på, har en 'height map' hämtats. Denna är använd i prototypen för att topologin skall stämma överens med referensmiljön. Verktyget i Unity 3D har tagit en 'height map' som input som innehåller data för att forma landskapet enligt topologin från 'height mapen'.

Resultatet har granskats och undersökts ifall den tänkta metoden skapar en skog som eftertraktar en verklig skog i form av distribution. Mätningen gick till på det sätt att referensskogen samt de simulationer som gjorts, delades upp i ett godtyckligt antal lika stora zoner. Dessa zoner fick sedan färger som representerade varje zons täthet av träd. I Figur 5 illustreras tillvägagången för hur mätningen av trädens täthet gick till. Det fanns tre kategorier på täthet som uttrycks i färg i undersökningen, rött, gult och grönt. De tre färgerna avgörs genom en uppskattning ovanifrån. Ifall en zon visar bar mark tillsammans med något enstaka träd anses zonen ha grön täthet. När en zon börjar visa flera träd och marken blir svårare att se anses zonen ha gul täthet. Är marken svår att se och majoriteten i bilden är trädtoppar är rutan av röd täthet.

- Rött - Indikerar på mycket tätt.
- Gult - Mellan tätt.
- Grön - Mycket lite eller ingen täthet.

Denna mätningsmetod påminner om fotogrammetri som är en teknik för att mäta med hjälp av bilder. Fotogrammetri har används sedan 1960-talet och kan används till att mäta volym, avstånd, area och höjder (Wiselqvist och Ohlsson, 2016).



Figur 5 Bild på en skog från ovan som är indelad i tre kategorier.

3.2 Problematisering

De frön som skapats har varit påverkade av vindzoner som då kunnat skapa problem. Det kunde exempelvis ha varit att alla frön flög oavbrutet och aldrig landade. Det fanns ytterligare ett problem med vinden som var att den inte betedde sig realistiskt. Detta kunde ha orsakat oväntade resultat med distributionen av träd.

Mätningen av trädens täthet gjordes för hand och då kan misstag ha inträffat. En risk med att mäta själv kunde ha varit att jag missbedömt tätheten på grund av den mänskliga faktorn. Eftersom att det var jag som analyserade mätningarna fanns det även en risk att det blev ett inskränkt perspektiv på vad mätningarna visade. Men ifall flera personer skulle ha analyserat mätningarna kan det ha funnits en risk att det kan ha blivit subjektivt för testpersonerna.

Det har inte hittats något vetenskapligt arbete där agenter har använts till att sprida träd. De nyckelord som har använts för sökning är: agent, tree, procedurall och generation. Sökningen har gjorts på flera kända databaser, IEEE Xplore, CiteSeerX och arXiv. Det kunde ha funnits en anledning till varför detta område inte är undersökt av många andra. Det kan ha varit att agenter inte presterat bra nog att utföra uppgiften och att andra tekniker var att föredra framför agenter.

4 Relaterad/tidigare forskning

4.1 Prestanda

Det går att simulera ett helt ekosystem men det skulle ta väldigt mycket prestanda påstår West (2008); “While natural tree distribution over the course of several decades is something we could simulate, this is for most games unrealistically expensive.”. Det sägs vara orealistiskt på grund av att det skulle vara så prestandakrävande för de flesta spelen.

Boken *Programming Game AI by Example* av Buckland (2005) är en bok som handlar om artificiell intelligens inom spel som täcker området för agenter. Buckland (2005) använder inte många agenter i sina prototyper vilket fungerade bra, men när det är många agenter i världen blir det problem med prestandan. Buckland (2005) implementerade sin lösning så att alla agenter hade tillgång till varandra, vilket kan skapa problem med prestandan. Det finns en lösning Buckland (2005) skrivit som är att istället för att kolla alla agenter i hela spelrymden, går det att kolla inom en bestämd radie, vilket optimerar agenterna. Den källan anses vara trovärdig eftersom personen är mycket insatt inom området och har mycket erfarenhet. Buckland (2005) har skrivit flera böcker inom området spelprogrammering. Den prestandaoptimeringen har varit viktig när ett frö ska landa och kolla närliggande träd.

4.2 Population

West (2008) skriver i sin artikel att det inte krävs avancerade metoder för att få kontroll över distributionen:

> “Sometimes simple solutions work out better than complicated ones. In this case, to scatter a forest of trees, we found that applying a very simple exclusion zone around plants achieved a nice random distribution of different sized plants.”
>
> West 2008 s.3

West (2008) demonstrerade att det gick att placera ut träd eller växter slumpat på ett plan där växterna hade egna exklusiva zoner så att inte växter överlappade. På liknande sätt som den lösningen är implementerad, finns likheter beskrivet i 3.1 där det finns exklusiva zoner. Det är onödigt att tänka på alla aspekter inför genereringen av en skog enligt West (2008).

Det anses vara onaturligt att träd blir ihopklumpade enligt West (2008); “Problems with this first solution are immediately apparent. Several of the dots on Figure 1 overlap, meaning the geometry of the trees would intersect”. I West (2008) artikel finns Figur 1 som visar en bild med runda cirklar där några av dem överlappar, där cirklarna är hans representationer av träd. Dapper (2003) har även gjort en presentation om hur man växer en naturlig skog. I den presentationen skriver han:

> “In this simple simulation plants shown as small circles with slighly varying radius were distributed on a plane. Growth means increase in diameter. When two circles collide, the smaller one dies due to domination by the bigger one. Once plants have reached a certain maximum size don't grow any bigger and eventually die.”

Dapper, 2003

Dapper (2003) menar att det är viktigt att inte två cirklar får överlappa. I implementationen Dapper (2003) skapat så tävlar växter om plats. Växterna växer, blir större och tillslut dör för att ge plats åt nya växter. Implementationen är även skapad så att stora växter kan ta död på mindre närliggande växter. Det här systemet är mycket mer avancerat än det som West (2008) skapat. Även Stenberg (2012) har gjort liknande arbete. Dapper (2003) har arbetat med grafik och programmering i över 10 år, han har även eget företag som inriktar sig på grafik.

5 Genomförande

Detta kapitel omfattar förstudie, designval och problem som uppstod med designvalen. I förstudien kommer det handla om inspiration till arbetet och var den kom ifrån. Kapitlet designval kommer att handla om de designval som gjorts i projektet, samt de problem som uppstått med dessa designvalen.

5.1 Arbetets designval

Det finns väldigt få som har använt agenter till att generera skogar. Det är en utav anledningarna till att agenter valdes som designval för att generera en skog. Det finns närliggande områden där agenter används till att exempelvis generera en stad (Lechner 2003). Som det är förklarat i kapitel 3.2, finns det mycket begränsad vetenskaplig fakta om det här området. Agenter kan användas till väldigt mycket och potentialen fanns till att skapa en skog.

Fröna behövde skapas från något och någonstans, det första alternativet var att de skulle komma ovanifrån på slumpade positioner. Det alternativet var genomförbart men skulle leda till det som skrevs om i kapitel 2.1, skapade träd slumpvis som inte var målet med arbetet. Det blev istället att ett träd fanns på plats i mitten av världen som utsöndrade frön. Dessa frön utsöndrades beroende på en global variabel, som hade koll på hur många frön det redan befann sig på spelbanan. Det var en åtgärd för att inte skapa lagg, eftersom skapandet av frön ökade oerhört när antalet träd ökade. Det kunde finnas max 290 stycken frön samtidigt i spelvärden.

Ett frö från ett träd behövde på något sätt transportera sig i luften. Det skapades vindzoner utspritt på spelplanen för att kunna ge variation på vart frön hamnade. Ett rutnät bildades av vindzonerna för att få en jämn spridning. En vindzon kunde blåsa mellan två parametrar för att ge olika kraft. Parametrarna är framtagna empiriskt, minsta vindkraften var 15 och högsta var 42. Med dessa parametrar fick fröna en dragningskraft som inte kastade iväg dem men ändå gav dem en skjuts.

Varje vindzon hade en riktning som till en början var helt slumpad. Med hjälp av en algoritm mjukgjordes dessa riktningar så att de pekar åt liknande håll. Mjukgörningen fungerade på det sättet att den tog närliggande vindzoner och svängde dessa åt liknande håll beroende på en faktor. Ifall faktorn var 1 så blev dessa väldigt lika och om den var 0 så hände inget. De värdet som gav mjuka flöden men inte för lika riktningar, var värdet 0.75 och det blev testat fram. Resultatet av algoritmen blev att luftströmmar skapades. För att inte vinden skulle blåsa likadant hela tiden ändras riktningen slumpat hos vissa zoner för att det sedan skulle påverka hela vindsystemet.

5.2 Problem med designval

Agenterna var många vilket medförde problem. Ett problem uppstod när en agent kollade närliggande agenter när den förflyttade sig. Agenterna var menade att röra sig lite flocklikt, att de skulle dras samman. Då behövde agenterna veta vilka fler agenter som var i närheten, för att kunna påverka förflyttningen. I en tidig version av prototypen fanns alla agenter sparade i en lista. För att kolla närliggande agenter behövdes hela listan itereras igenom för att kolla ifall det var någon i närheten. Detta drog otroligt mycket beräkningskraft när antalet agenter ökade. Lösningen för detta problem var att kolla ifall det fanns några agenter inom en

viss radie runt agenten. Liknande lösning användes även av Buckland (2005 s.127), där han beskrev samma problem.

Ett annat problem som påverkade prestandan negativt var hur många vindzoner som fanns. Det fanns 30 stycken till en början vilket innebar en stor påfrestning på hela systemet. Påfrestningen var att varje zon var tvungen att kolla vilka frön som var innanför sin radie för att sedan lägga på en kraft på de fröna. Detta gjordes väldigt ofta och då tog det mycket beräkningskraft för alla vindzoner att kolla över alla frön. En lösning på detta problem var att minska antalet vindzoner, vilket fungerade bättre utan att påverka resultatet avsevärt.

6 Utvärdering

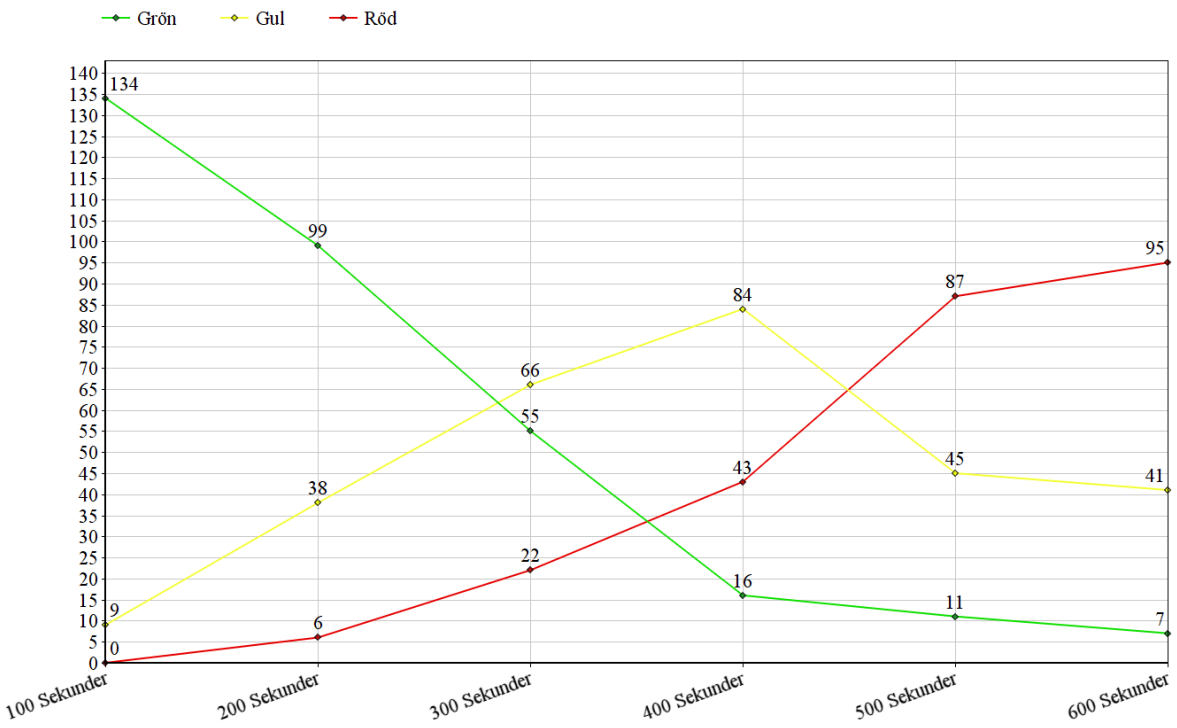
Målet med arbetet var att se ifall agenter kan skapa en skog med distribution som påminner om den svenska tallskogen. Detta kapitel kommer börja med att presentera körningar som gjorts samt olika mätningar för att sedan gå in på analys och ifall det går att dra några slutsatser.

6.1 Presentation av undersökning

När miljön skapas har träden en växande kurva på hur snabbt de ökar i antal som sedan avtar. I Figur 6 finns det ett diagram för de olika täthetszonerna i en körning på 600 sekunder.

- Rött - Indikerar på mycket tätt.
- Gult - Mellan tätt.
- Grön - Mycket lite eller ingen täthet.

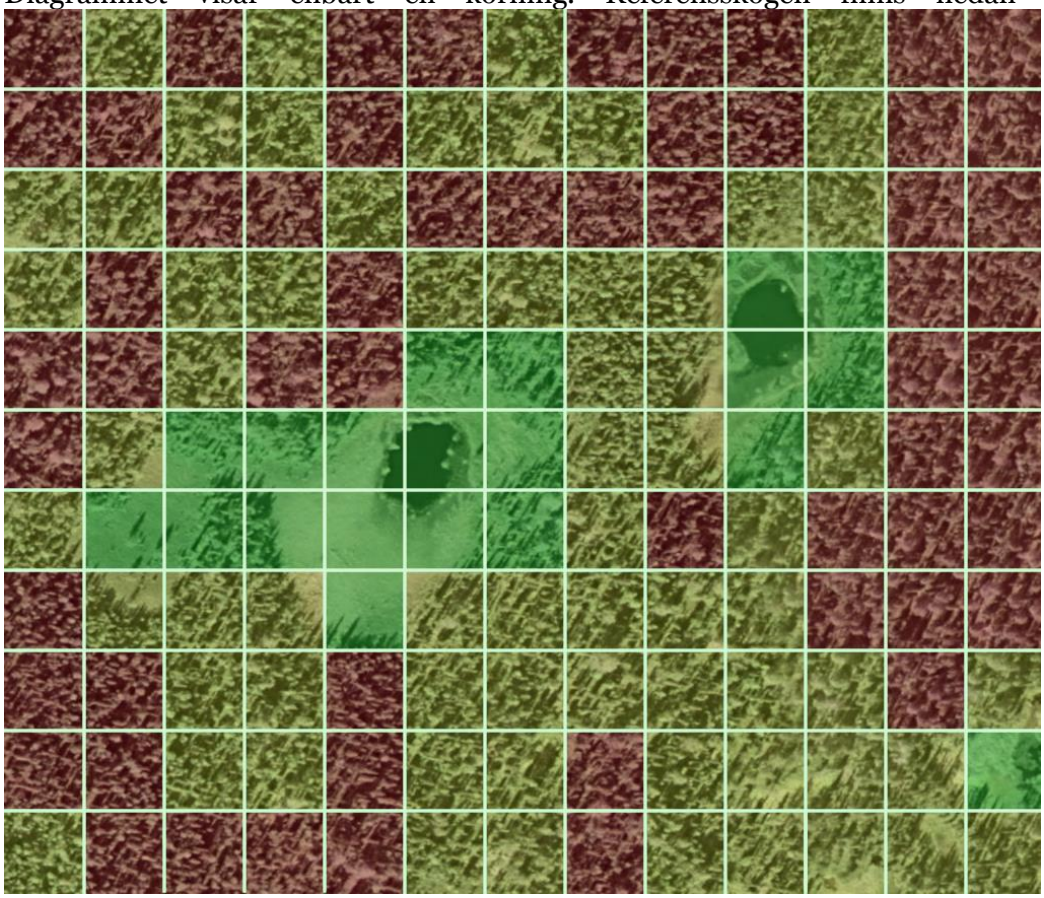
I diagrammet visas det hur de gröna zonerna är mycket överlägsna tills att simulationen kommit till drygt 300 sekunder då de gula och röda zonerna börjar ta över miljön.



Figur 6 Diagram över trädens tillväxt med tre zoner som beskriver tätheten i miljön. Max antal frön: 290 stycken. Trädets livslängd: 45 sekunder.

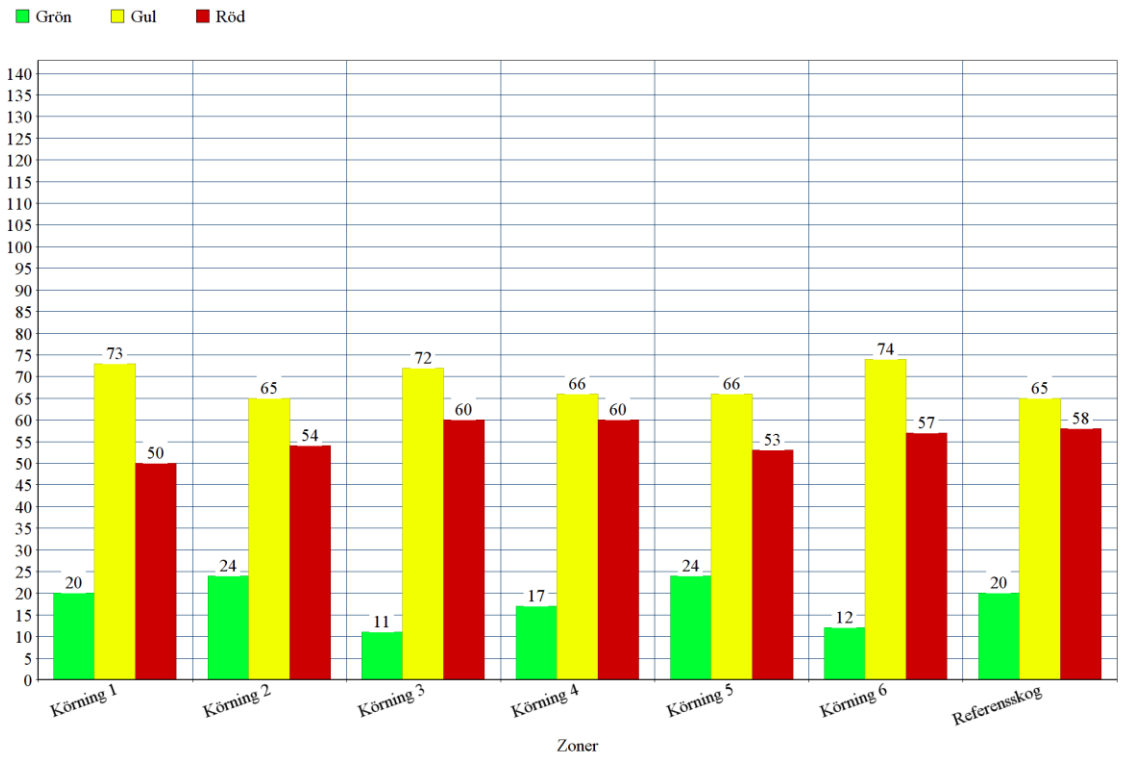
Det finns områden i miljön där träd har svårare att spridas till. Efter 600 sekunder visar diagrammet i Figur 6 att det fortfarande finns zoner där det är lite eller inga träd alls.

Diagrammet visar enbart en körning. Referensskogen finns nedan i Figur 7.



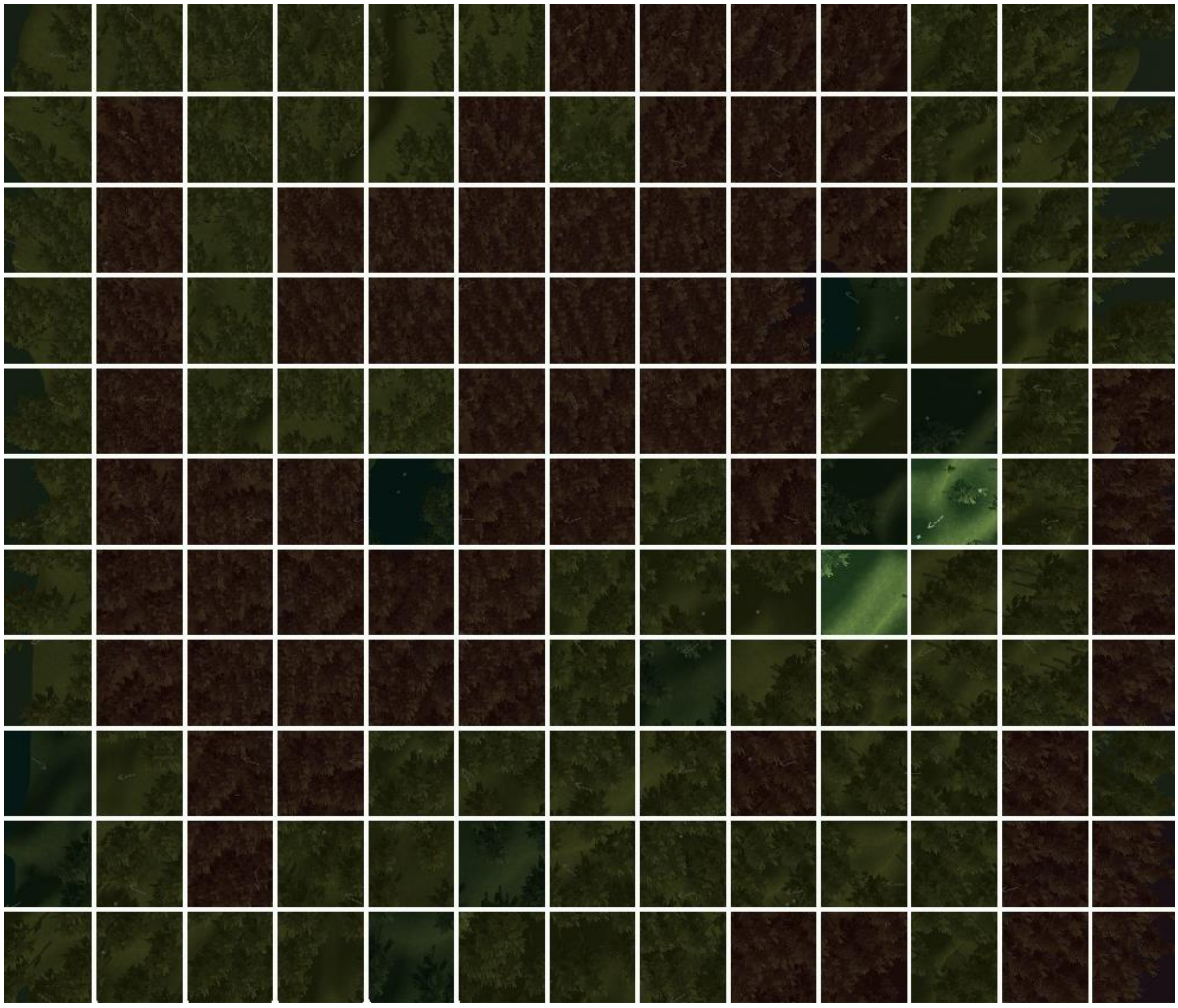
Figur 7 Referensskogen indelad i zoner.

Nedan i Figur 8 är ett diagram för 6 stycken körningar och referensskogen.



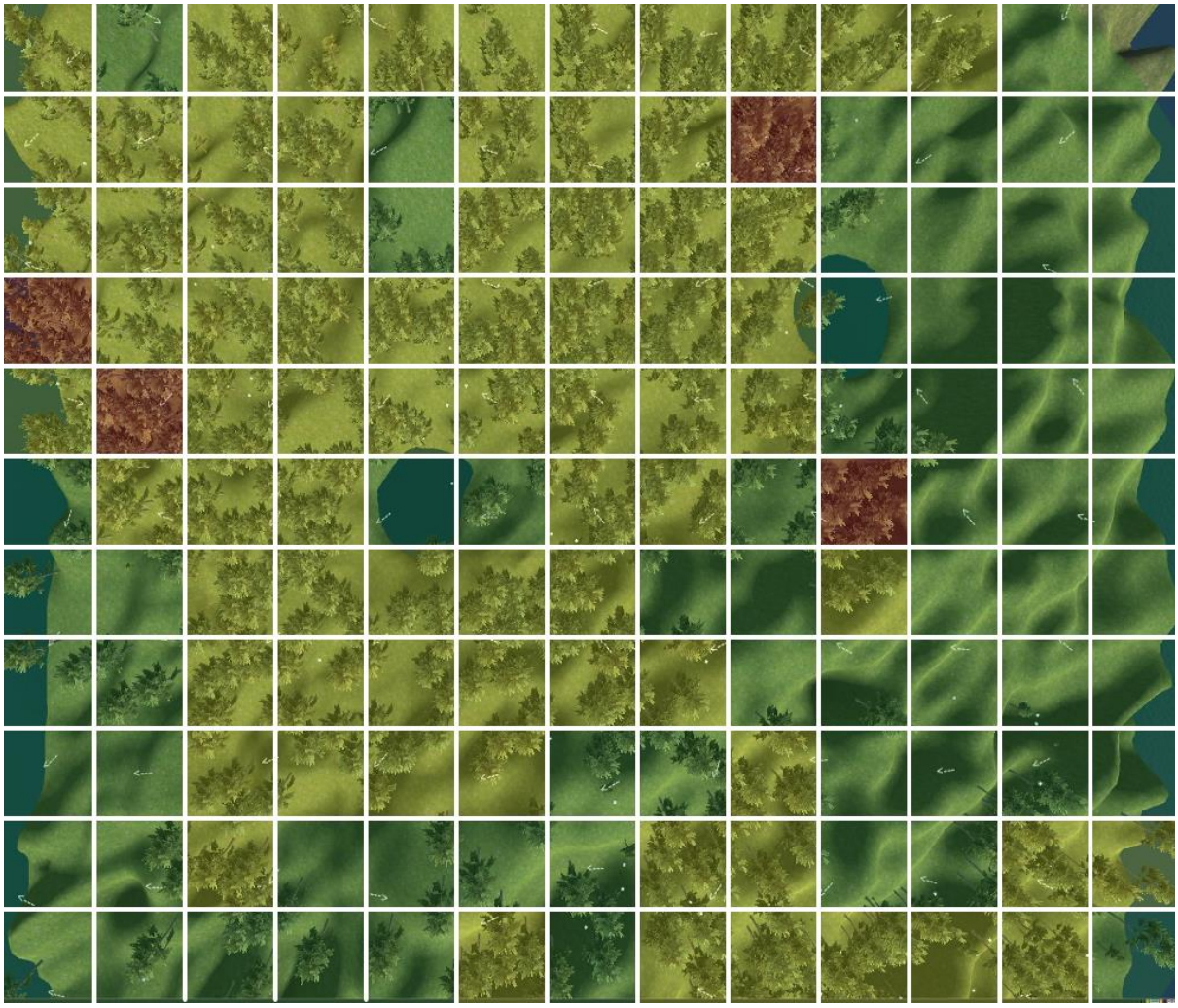
Figur 8 6 körningar på 600 sekunder tillsammans med referensskogen. Max antal frön: 290 stycken. Trädets livslängd: 45 sekunder.

I Figur 9 visas två rutor som båda varit gröna 5/6 körningar efter 600 sekunder.



Figur 9 Illustration av återkommande gröna zoner på platserna (4,9) & (5,10) då (0,0) är nere till vänster. Max antal frön: 290 stycken. Trädets livslängd: 45 sekunder.

I Figur 6, Figur 8 och Figur 9 har körningar haft ett max antal frön som får befinna sig i världen samtidigt, max antalet var satt till 290 stycken. Ifall max antalet frön var ökat skapades lagg och det blev mycket tätt överallt. När max antalet var lägre, spred sig inte träden och de var mycket färre i antal. I Figur 10 och Figur 11 ser vi hur resultatet blev påverkat när max antalet höjts och sänkts.



Figur 10 Körning på 600 sekunder med max 100 frön samtidigt i miljön. Trädets livslängd: 45 sekunder.

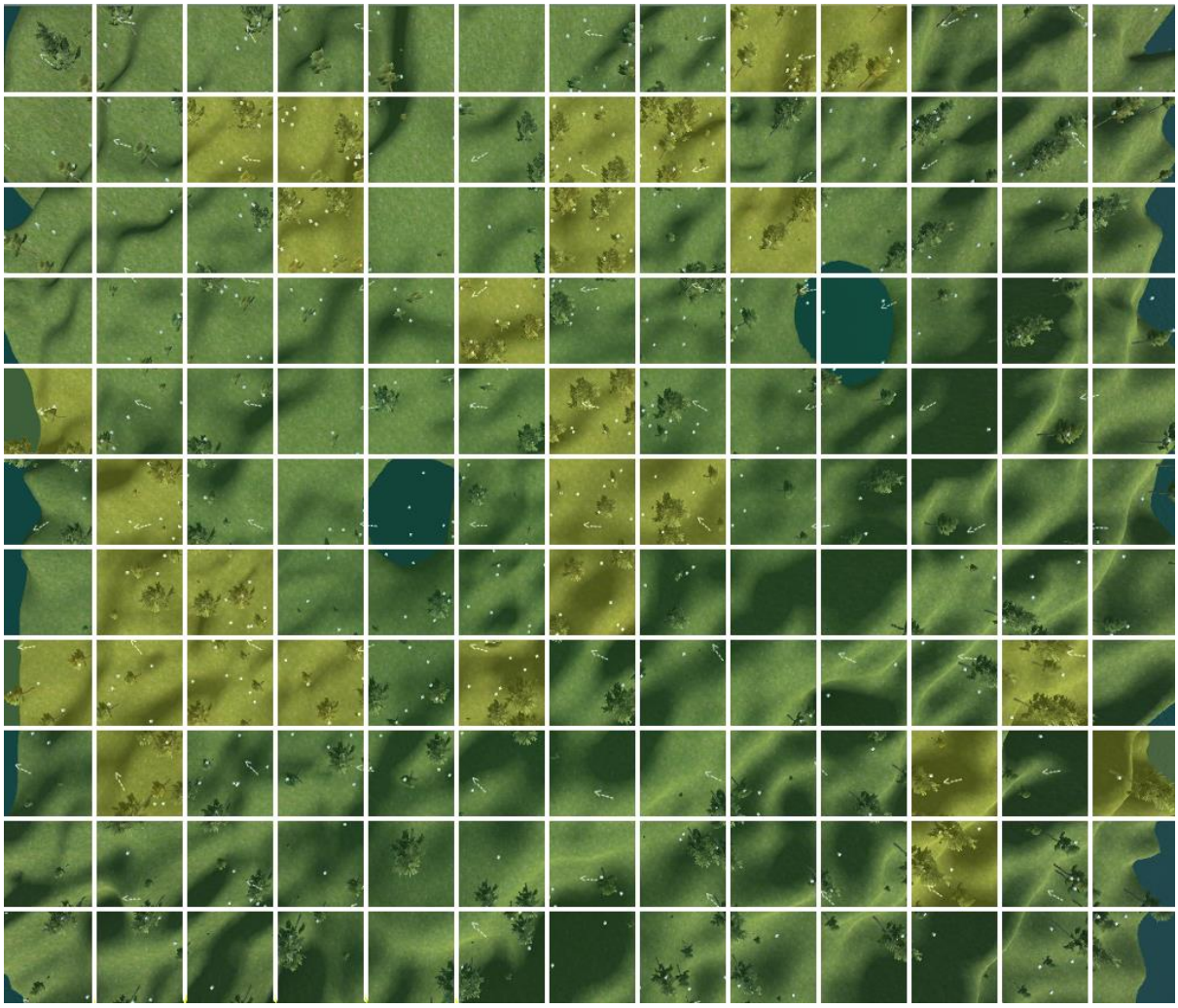
När max antal frön var satt till att vara 100 stycken spreds inte träden fort. I Figur 10 visas en stor majoritet för gula och gröna zoner som är 139 stycken tillsammans mot röda zoner som endast är 4 stycken.

Figur 11 Körning på 600 sekunder med 600 max antal frön i miljön samtidigt. Trädets livslängd: 45 sekunder.

När max antal frön var satt till att vara 600 stycken blir resultatet motsatsen till det som visades i Figur 10, skogen blir mycket mer tät. De röda zonerna har stor majoritet på 98 stycken mot gul och grön som tillsammans är 45 stycken.

Efter att ett träd har skapats har den en viss tid att leva som var angivet till att vara 45 sekunder. Värdet som den hade blev framtaget empiriskt och likt max antal frön påverkar den tätheten i miljön.

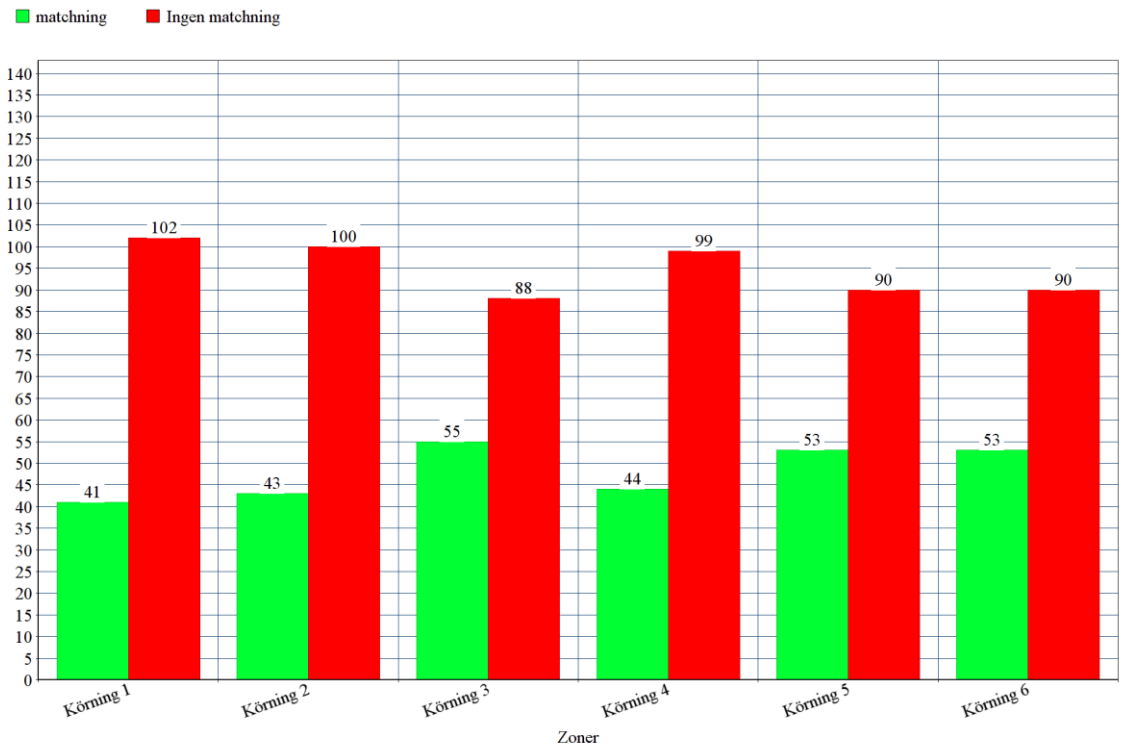
Ifall max antal frön var motsatt till livslängden så att den ena var hög och andra låg och vice versa, blev resultatet som i Figur 12 och Figur 13.



Figur 12 Max antal frön var 600 och livslängden för träd var 25 sekunder.

Figur 13 Max antal frön var 150 och livslängden för träden var 65 sekunder.

Körningarna och referensskogen kan jämföras när det kommer till hur zonerna är placerade och vad de har för likheter. Det kunde vara så att till exempel zon (0,0) var av samma täthet i simulationen som i referensskogen. I Figur 14 visas hur många zoner som stämde överens mellan de 6 körningarna och med referensskogen.



Figur 14 Översikt för hur många matchande zoner de 6 körningarna hade med referensskogen. Max antal frön: 290 stycken. Trädets livslängd: 45 sekunder.

6.2 Analys

I de olika körningarna är max antal frön viktigt, eftersom det avgör hur fort träden sprider sig. Det finns ett max antal eftersom att det inte ska vara möjligt att det blir oändligt antal frön samtidigt. Diagrammet i Figur 6 visar tillväxten av träd över en tid. Det som påverkar tillväxten är som nämnt tidigare, max antal frön och även hur tätt träden hamnar. De röda zonerna blir inte allt för täta, eftersom chansen för frön att skapa flera träd i dessa zoner är liten. I Figur 10 och Figur 11 ser vi hur resultatet blev när max antal ökat och minskat. När max antal frön endast var 100 stycken spred sig inte träden lika mycket och antalet träd var även reducerat. När det endast fick vara 100 stycken frön samtidigt är antalet frön som blev träd bara lite större än de träd som försvinner efter en tid. När max antal frön är satt till att vara 600 blir chansen att ett nytt träd skapas större, eftersom det får produceras mer frön och det resulterar i vad som visas i Figur 11.

Variablen timeAlive påverkar hur länge ett träd lever vilket är betydelsefullt för simuleringen men resultatet blir det samma som när max antal frön ändrades (Figur 10 och Figur 11). Det blir liknande resultat eftersom ett träd har en potential att skapa ett visst antal frön under sin livstid. När livstiden för trädet förkortas, minskas de potentiella antal frön den kan skapa och det är bland annat därför antal träd minskas. Ifall livslängden för trädet ökar, blir antalet frön den kan skapa fler som resulterar i flera träd.

I Figur 12 och Figur 13 var max antal frön motsatt till livslängden för träden alltså den ena högre än den andra. Figur 12 visar när max antal frön var många men livslängden var låg. Resultatet blev att spridningen var sämre samtidigt som att det inte fanns många träd. Detta beror på att träden inte hann leva länge nog för att kunna skapa många träd. För att det skulle bli tätt behövdes det återplanteras ett nytt träd där ett träd nyligen försvunnit vilket inte gjordes alla gånger. När max antal frön var färre men livslängden för träden var lång blev resultatet att skogen var mindre och tät. Träden placerades tätt eftersom när det inte fanns många frön blev chansen för spridning försämrad. Flera frön ökar chansen att något frö når svåråtkomliga platser.

Det fanns en chans att till exempel zon på plats (0,0) var av samma täthet i simulationen som i referensskogen, men den chansen var inte stor. Detta hände eftersom det finns vattenhål i referensskogen, där det är mycket gröna runt om. En gissning är att marken runt vattnet inte uppfyller de krav träden har för att växa, därför blir det gröna zoner runt det området. I simulationen finns det inte någon algoritm implementerad för att skapa detta fenomen och träden kan placeras överallt förutom i vatten.

Det har nämnts tidigare om två zoner där de har varit väldigt få träd under flera körningar, se Figur 9. Detta kan bero på den höjdskillnad som finns där, eftersom på de två platser är det kullig mark.

6.3 Slutsatser

Kan agenter skapa en skog där distributionen påminner om en svensk tallskog? Flera körningar har gjorts som skapat miljöer där skogarnas zoner liknat referensskogen i antal, se Figur 8. Denna statistik visar endast hur många zoner av de olika kategorier som finns och säger inget om var dessa zoner är placerade. Diagrammet i Figur 8 visar att gröna zoner går mellan att vara 11 – 24, gula zoner; 65 – 74 och röda zoner; 50 – 60. Referensskogen har 20 stycken gröna zoner, 56 gula zoner och 68 röda zoner. Detta säger att referensskogen befinner sig inom de spann som skapats av simulationer.

När en jämförelse mellan hur simulationernas och referensskogens zoner är placerade i sina världar visas mycket lite gemenskap, se Figur 14. En gissning som nämnt i tidigare kapitel är att marken har ett särskilt tillstånd som gör att träd har svårt att växa där. Simulationen är inte implementerad på det vis att den tar hänsyn till detta fenomen, men det behöver inte göra den simulerade skogen mindre intressant.

För att svara på frågeställningen: ”Kan man använda agenter för att skapa en svensk tallskog med realistisk distribution?” är det två saker som undersökts, antal zoner för täthet och var dessa zoner är placerade. Det första; är antalet olika zoner för täthet något likt referensskogens antal zoner? I de olika körningarna som gjorts har det skapat zoner där antalet för de olika tätheterna ligger runt om, eller väldigt nära referensskogens antal. Detta innebär att miljöerna som simulerats fram hade liknande antal täthetszoner som referensskogen, det är ett viktigt steg för att uppnå en verkligare distribution. Det andra; vad för gemensamma zoner har körningarna till referensskogen. I detta fall fanns inte jättestora likheter men det betyder inte att körningarna skapar skogar som inte har realistisk distribution. Utöver gemensamma zoner så har referensskogen sällan en eller två gröna zoner bredvid en röd zon. Precis som referensskogen gäller det även körningarna, det var ovanligt att det låg en röd zon bredvid en eller två gröna zoner. Det går även att se likheter mellan körningarna och referensskogen att

det inte förekommer ofta att det bildas ensamma röda zoner. Med det sagt kan jag dra slutsatsen att agenter kan användas för att skapa en realistisk distribuerad svensk tallskog.

7 Avslutande diskussion

7.1 Sammanfattning

Agenter inom artificiell intelligens kan användas till många saker och kommer att sättas på prov i att distribuera träd i en miljö. När en verklighetstrogen miljö skapas för bland annat spel, är positioneringen viktig. Tidigare metoder för att placera ut träd är exempelvis slumpen använd tillsammans med kriterier som kan vara att träd inte får överlappa. Med hjälp av slumpade positioner och kriteriet skapas träd därefter.

Detta arbete hade som mål att använda metoder inspirerat från verkligheten tillsammans med agenter för att distribuera träden i en verklighetstrogen form. De metoder som använts är agenter som agerar som frön och vindzoner som var placerade runt om miljön. För att göra undersökningen fanns en referensskog från verkligheten. Där denna referens är tagen är dessutom en höjdkarta hämtad för att testmiljön skulle se likvärdig ut.

7.2 Diskussion

Detta arbete är grundat i problemformuleringen som beskriver problemet med att skapa en skog med procedurell generering, där agenter låg i fokus. Trovärdigheten för resultatet kan vara slumpad och inte ge ett säkert resultat varje gång. Eftersom det finns många olika faktorer som spelar in när det kommer till procedurell generering finns det alltid fall där resultatet anses vara misslyckat. Ifall någon skulle använda detta arbete i hopp om att få en realistisk distribuerad svensk tallskog, kan det hända att resultatet inte blir lyckat. Personen som ville ha den svenska tallskogen kanske fick ett extremfall där det inte alls liknar en skog. Arbetet täcker viktiga delar som till exempel variablerna för max antal frön och trädens livslängd nämnt i kapitel 6.2. De kan vara till hjälp för att skapa en svensk tallskog.

Det finns särskilda områden i referensbilden där det inte växer några träd. Detta kan bero på markens tillstånd att de inte växer några träd där. Hur detta påverkar resultatet av undersökningen är betydelsefullt och kan vara en viktig del i att skapa en verkligare miljö. Ifall denna metod skulle användas för att skapa till exempel spelbanor råder jag att sköta denna sorts specialfall för hand.

Detta arbete kan hjälpa till med ekonomisk hållbarhet eftersom det kan producera en skog med valfritt ändamål. Det som artefakten gör är att den placerar ut hundratals träd som skulle tagit tid och arbetskraft ifall någon person skulle göra det. Med hjälp av detta arbete hoppas jag att läsaren kan få hjälp med liknande problem. En annan aspekt på hållbarhet är den sociala hållbarheten. Tanken är att efter att ha läst detta arbete kan läsaren utan svårigheter implementera liknande artefakt. En bidragande faktor till att det stödjer social hållbarhet är lättläslig kod, det vill säga berättande variabelnamn som jag hoppas jag lyckats med. Detta arbete har inte haft några deltagare och därför finns inte någon deltagares integritet att beskydda eller några andra forskningsetiska aspekter att ta hänsyn till.

7.3 Framtida arbete

Ifall det här arbetet skulle fortsättas att arbeta på finns det flera saker som går att förbättra beroende på vad man vill göra. Är målet att skapa en skog som liknar en referensbild på pricken eller att skapa en skog som har verkliga aspekter som då påminner om en verklig skog?

Ifall målet är att skapa en skog efter en bild finns det bättre metoder för detta som till exempelvis nämnt i kapitel 2.1 där positionering av träd är gjort med hjälp av bilder. Det målet som detta arbete hade var att framställa en skog som skulle kunnat ha varit en verklig skog. De saker som kan förbättras är till exempelvis använda sig av marken. I bilden på den verkliga skogen fanns det områden där inga träd växte alls. Marken avgör väldigt mycket om vad som kan växa där och det kan implementeras specialfall för träden när de landar på speciella underlag. Det går att göra väldigt många specialfall och det går även att kombinera med olika trädarter så att det blir regioner av olika träd. Altitud är en aspekt som också är viktig. Som det är nu är det redan svårt för träd att klättra men inte omöjligt. Ifall det skulle önskas ännu mer restriktion går det även med altitud att avgöra huruvida träden kan växa.

Ifall ett företag skulle använda detta för ekologi och se tillväxt bland träd kommer det att krävas mer exakt data kring vind, frön och markens tillstånd. Mycket skulle behövt göras om och skulle bara vara dåligt för deras ekonomi. Däremot skulle ett spelföretag kunna använda detta för att skapa spelbanor. I en spelbana är inte verkligheten lika viktig eftersom att det ska finnas rum för balansering av spelbanan. Ett exempel på när en balansering krävs är när ett område har fått överlägset bra positionerade träd som kan ge övertaget till en spelare.

Referenser

arXiv (1991) Cornell University. arXiv.org [2018-03-09]

Buckland, M. (2005) Programming Game AI by Example. Jones och Bartlett Publishers, LCC.

CiteseerX (1997) NEC Research Institute, Princeton, New Jersey. http://citeseerx.ist.psu.edu [2018-03-09]

Dapper, T. (2003) Practical Procedural Modeling of Plants, Gilles Tran. Universität Bremen. http://www.td-grafik.de/artic/talk20030122/overview.html [2017-12-28]

Gamasutra, The art & buisness of making games, UBM. https://www.gamasutra.com [2017-12-28]

IEEE Xplore (u.å) Institute of Electrical and Electronics Engineers. http://ieeexplore.ieee.org/Xplore/home.jsp [2018-03-09]

Johnson, L., Yannakakis, G. N., & Togelius, J. (2010) Cellular automata for real-time generation of infinite cave levels. IT University of Copenhagen. PCGames, Monterey, CA, USA

Lantmäteriet, Referensskogen. https://kso.etjanster.lantmateriet.se/?e=689946&n=7447935&z=13&profile=default_terrangskuggning_noauth [2018-05-18]

Lechner, T., Watson, B. & Wilensky, U.: Procedural city modeling. In: Proceedings of the 1st Midwestern Graphics Conference (2003).http://ccl.northwestern.edu/2004/ProceduralCityMod.pdf [2018-09-02]

Nelson, M., Shaker, N. & Togelius, J. (2016). Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer. ISBN 978-3-319-42714-0.

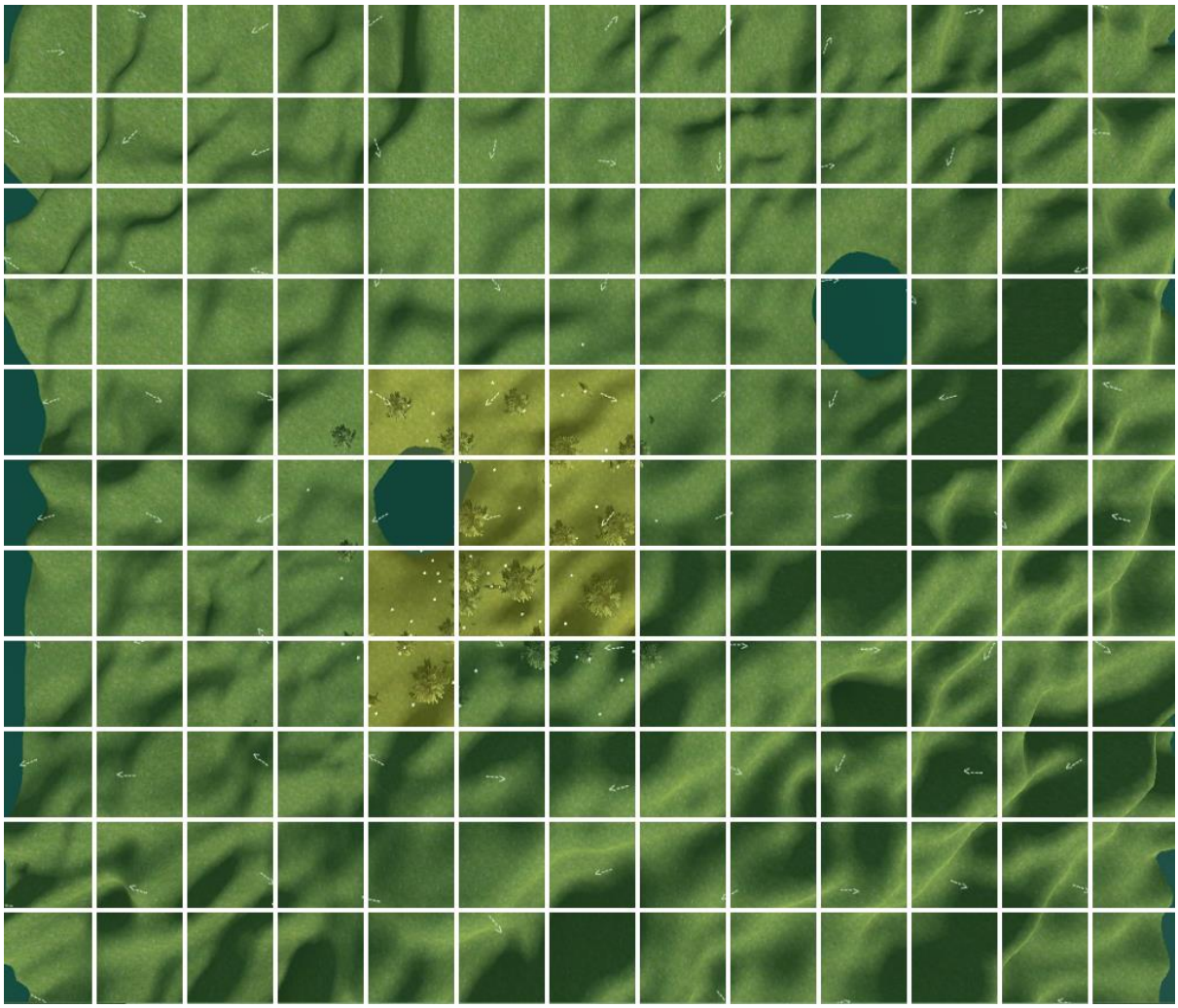
Stenberg, C. (2012) Co-evolving niches in virtual Plant species: Exploring the niche forming capabilities of coevolving plants in a virtual environment. Högskolan i Skövde.

Togelius, J., Kastbjerg, E., Schedl, D. & Yannakakis, G, N. (2011) What is Procedural Content Generation? Mario on the borderline. In: Proceedings of the 2nd Workshop on Procedural Content Generation in Games. http://julian.togelius.com/Togelius2011What.pdf [2018-09-02]

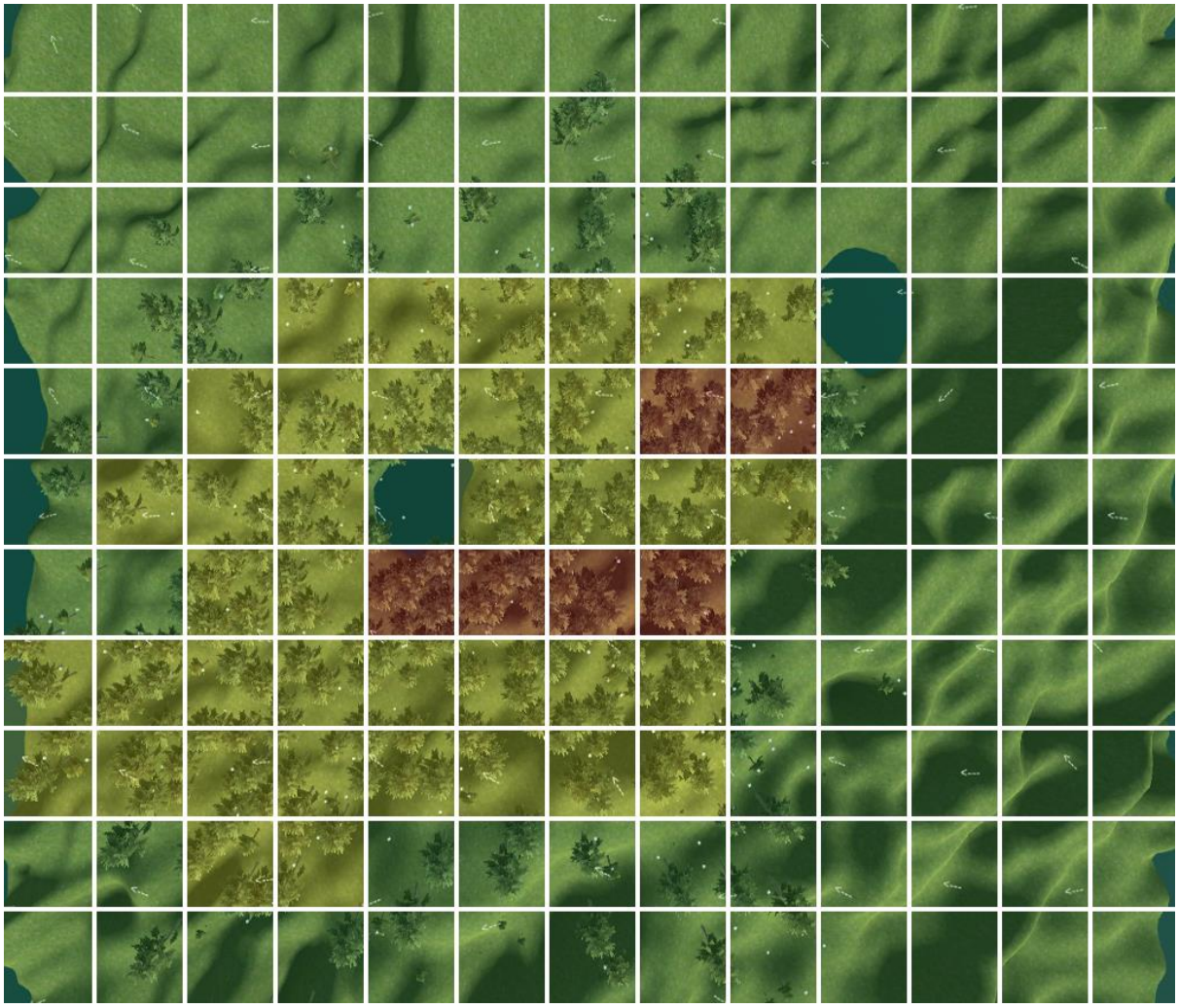
West, M. (2008) Random Scattering: Creating Realistic Landscapes, Gamasutra's editors. https://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php [2017-12-28]

Wiselqvist, D. och Ohlsson, E. (2016) Drönarfotogrammetri som komplement till den nya natilnella höjdmodellen, En komparativ översvämningsanalys. Department of Earth Sciences. University of Gothernburg. https://gvc.gu.se/digitalAssets/1588/1588347_b925.pdf [2018-03-09]

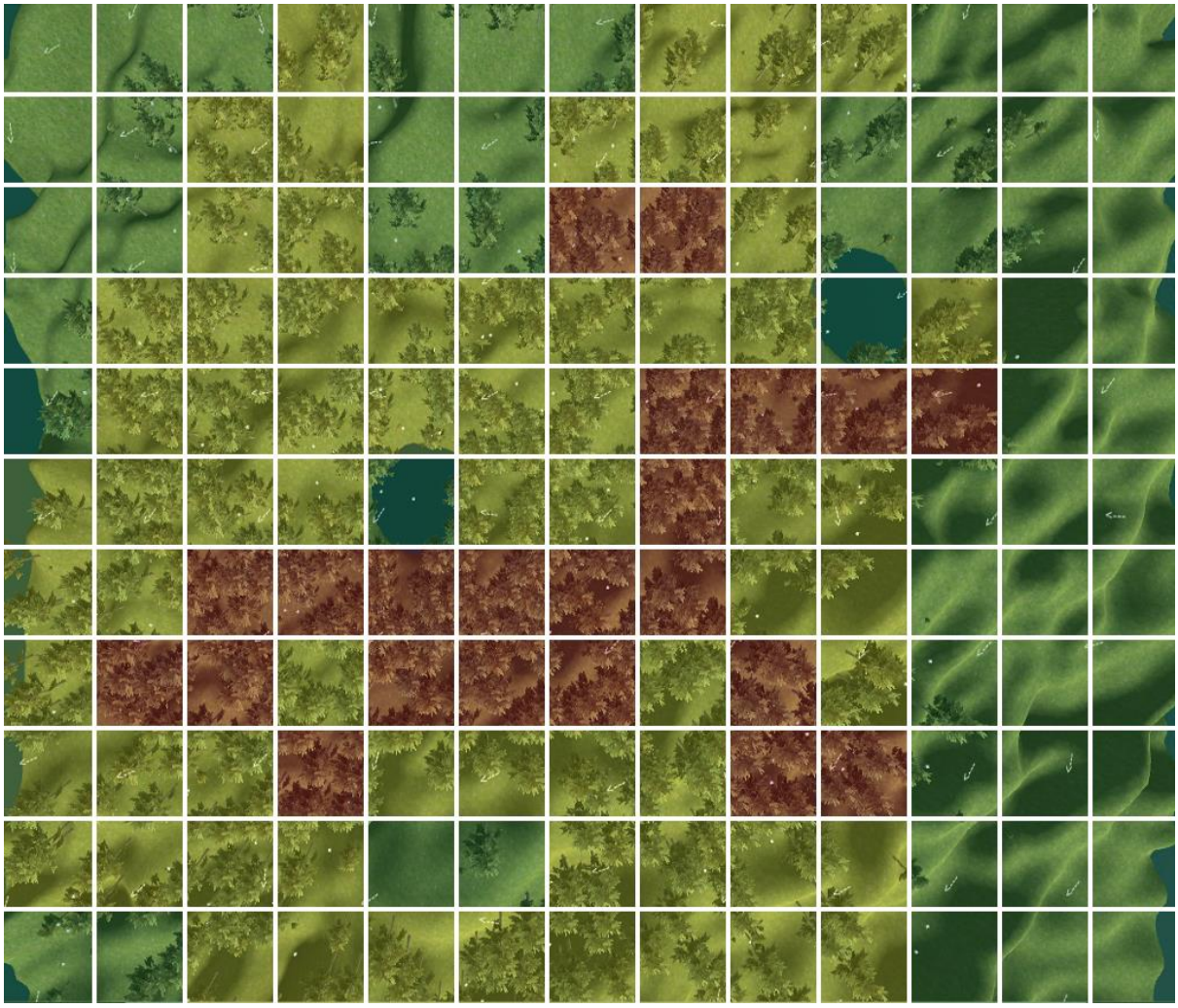
Appendix A - Bilagor



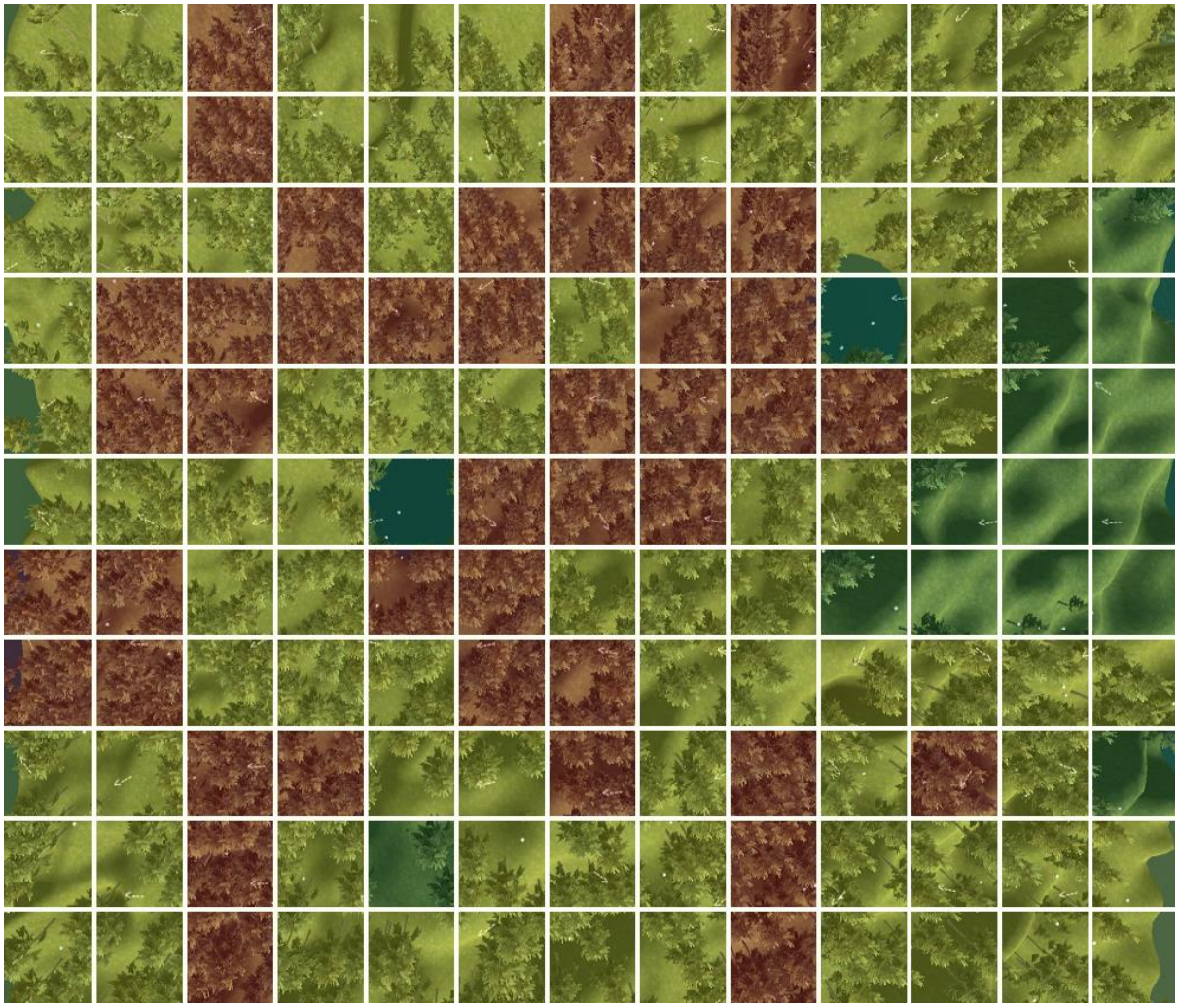
Figur 15 Körning – 100 sek.



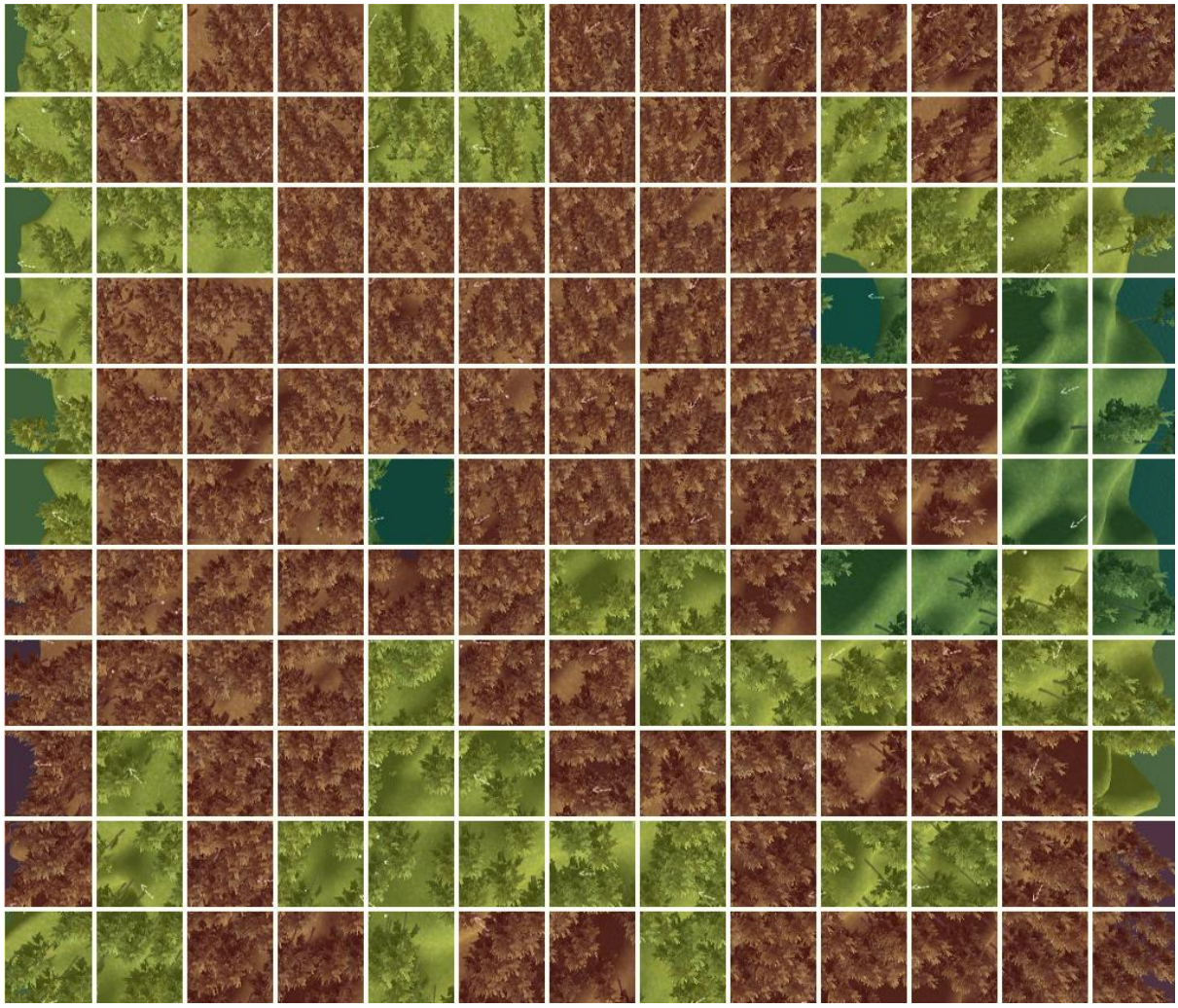
Figur 16 Körning – 200 sek.



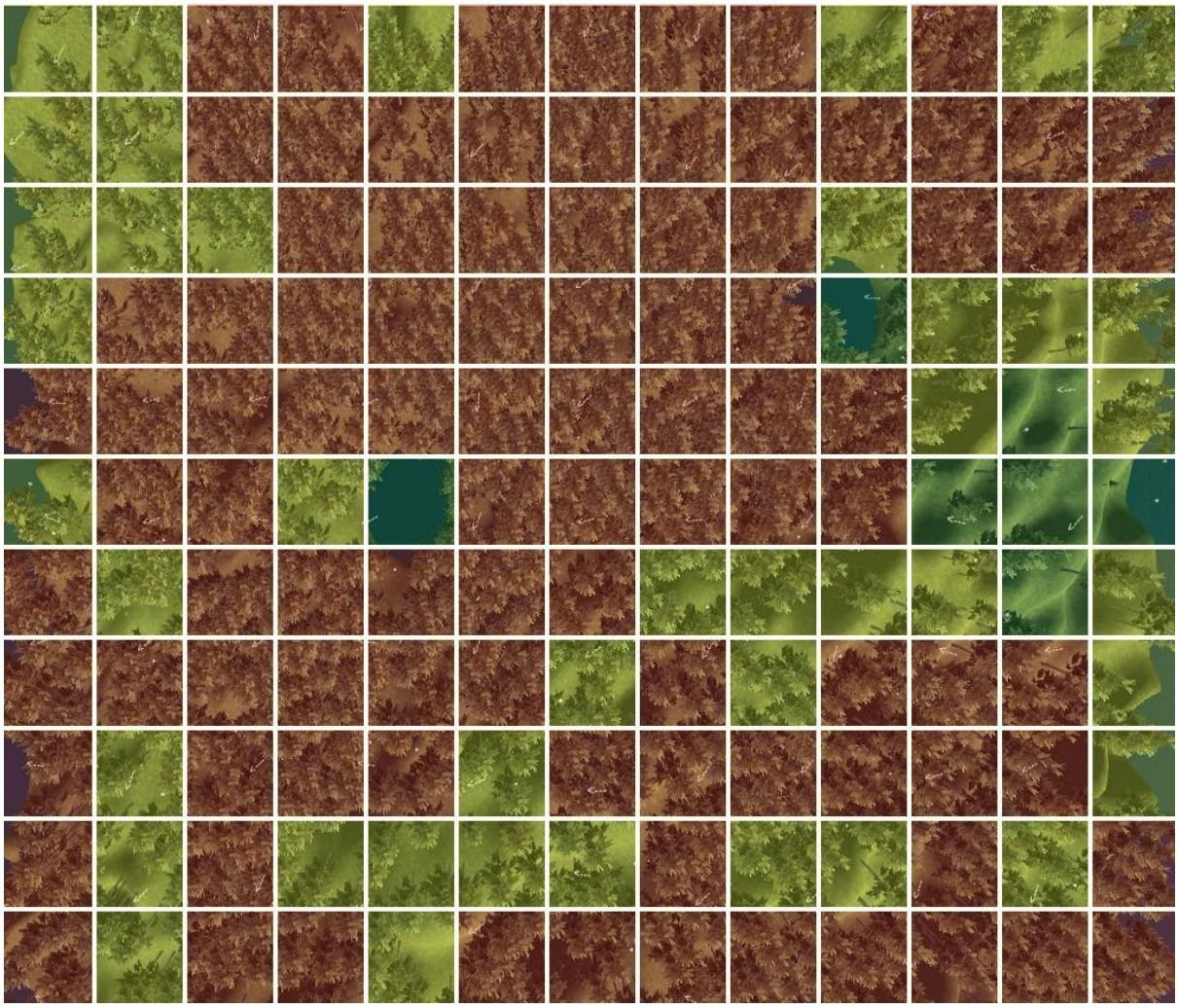
Figur 17 Körning – 300 sek.



Figur 18 Körning – 400 sek.



Figur 19 Körning – 500 sek.



Figur 20 Körning – 600 sek.

Appendix B - Kod

ArrayLayout.CS

```
using System.Collections;
using UnityEngine;
using System.Collections.Generic;
[System.Serializable]
public class ArrayLayout {
    [System.Serializable]
    public struct rowData
    {
        public List<GameObject> row;
    }
    public List<rowData> rows = new List<rowData>();

    public ArrayLayout(int x)
    {
        for (int i = 0; i < x; i++)
        {
            rows.Add(new ArrayLayout.rowData());
        }

    }
}
```

Globals.CS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Globals : MonoBehaviour
{
    public static Globals Instance { get; private set; }

    [Range(0.0f, 25.0f)]
    public float TimeScale = 1.0f;
    public float timer = 0f;
    public float tim;

    public float timeAlive = 5f;
    public int seedCount = 0;
    public int maximumSeeds = 400;
    private void Awake()
    {
        tim = Time.time;
        if (Instance == null)
        {

            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }

    }
    private void Update()
    {
        if (tim + timeAlive <= Time.time)
        {
            TimeScale = 0f;
        }

        timer = Time.time;
        Time.timeScale = TimeScale;
    }
}
```

WindEditor.CS

```
using System.Collections;
using System.Collections.Generic;
using UnityEditor;
using UnityEngine;

[CustomEditor(typeof(WindZoneManager))]
public class Utility : Editor {

    public override void OnInspectorGUI()
    {
        WindZoneManager manager = target as WindZoneManager;
        if (DrawDefaultInspector())
        {
            manager.Lerp();
        }
        if (GUILayout.Button("Lerp"))
        {
            manager.Lerp();
        }
    }
}
```

WindSimulate.CS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WindSimulate : MonoBehaviour
{
    private float windStrenght = 4f;
    public bool showRadius = true;
    public bool showDirection = true;

    public float radius = 15f;
    public float windStrengthMin = 0f;
    public float windStrengthMax = 5f;
    public Transform windTransformPosition;
    public Transform windTransformRotation;

    private void Start()
    {
        windTransformRotation.Rotate(0f, Random.Range(0f, 360f), 0f);
    }

    void Update()
    {
        if (windTransformPosition != null && windTransformRotation != null)
        {
            windStrenght = Random.Range(windStrengthMin, windStrengthMax);
            windTransformRotation.rotation = transform.rotation;

            Collider[] hitColliders;
            hitColliders =
Physics.OverlapSphere(windTransformPosition.transform.position, radius);
            for (int i = 0; i < hitColliders.Length; i++)
            {
                if (hitColliders[i].GetComponent<Rigidbody>() != null)
                {
                    Vector3 rayDirection =
hitColliders[i].GetComponent<Rigidbody>().gameObject.transform.position -
windTransformPosition.transform.position;

hitColliders[i].GetComponent<Rigidbody>().AddForce(windTransformPosition.transform.for
ward * windStrenght, ForceMode.Acceleration);
                }
            }

        }
    }
    private void OnDrawGizmosSelected()
    {
        if (showRadius)
        {
            Gizmos.color = Color.yellow;
            Gizmos.DrawWireSphere(transform.position, radius);
        }

    }
}
```

WindZoneManager.CS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WindZoneManager : MonoBehaviour {
    public GameObject windzonePrefab;
    public Terrain terrain;
    public ArrayLayout zoneList = new ArrayLayout(5);
    public int zonesX = 5;
    public int zonesY = 5;
    public float scaleX = 5f;
    public float scaleY = 5f;
    public string holdingName = "Windzones";
    public Transform Windzones;

    // Use this for initialization
    void Start ()
    {
        scaleX = terrain.terrainData.size.x / zonesX;
        scaleY = terrain.terrainData.size.z / zonesY;
        Windzones = new GameObject(holdingName).transform;

        InvokeRepeating("Lerp", 0, 2f);
        InvokeRepeating("windChange", 0, 5f);

        for (int x = 0; x < zonesY; x++)
        {
            for (int y = 0; y < zonesY; y++)
            {
                float altitude = terrain.terrainData.GetHeight(Mathf.RoundToInt( x *
scaleX ), Mathf.RoundToInt(y * scaleY));
                GameObject newZone = Instantiate(windzonePrefab, new Vector3(x *
scaleX, altitude, y * scaleY), Quaternion.identity);
                zoneList.rows[x].row.Add(newZone);

                newZone.transform.parent = Windzones;
            }
        }
    }
    public void windChange()
    {
        float lerp = 0.55f;
        int dirChange = Random.Range(0, 4);
        Debug.Log(dirChange);
        for (int x = 0; x < zonesX; x++)
        {
            for (int y = 0; y < zonesY; y++)
            {
                Vector3 temp = new Vector3();

                switch (dirChange)
                {
                    case 0:
                        temp = new Vector3(0f, Random.Range(200f, 500f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);
```

```
zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);

                        break;
                    case 1:
                        temp = new Vector3(0f, Random.Range(280f, 600f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);
                        break;
                    case 2:
                        temp = new Vector3(0f, Random.Range(50f, 360f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);
                        break;

                    case 3:
                        temp = new Vector3(0f, Random.Range(200f, 440f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);
                        break;

                    case 4:
                        temp = new Vector3(0f, Random.Range(0f, 360f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);
                        break;
                    default:
                        temp = new Vector3(0f, Random.Range(0f, 360f), 0f);

                        temp =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s, temp, lerp);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, temp.y, 0);
                        break;
```

```
                }


            }
        }

    }

public void Lerp()
    {
        int cases = 100;
        float lerpFactor = 0.15f;
        Vector3 tempVector;
        for (int x = 0; x < zonesX; x++)
        {
            for (int y = 0; y < zonesY; y++)
            {
                tempVector = new Vector3();
                if (x != 0 && x != zonesX - 1 && y != 0 && y != zonesY - 1)
                    cases = 8;
                else if (x == 0 && y == 0)
                    cases = 0;
                else if (x == 0 && y == zonesY - 1)
                    cases = 1;
                else if (x == zonesX - 1 && y == zonesY - 1)
                    cases = 2;
                else if (x == zonesX - 1 && y == 0)
                    cases = 3;
                else if (x == 0 && y != 0 && y != zonesY - 1)
                    cases = 4;
                else if (x != 0 && x != zonesX - 1 && y == 0)
                    cases = 5;
                else if (x != 0 && x != zonesX - 1 && y == zonesY - 1)
                    cases = 6;
                else if (x == zonesX - 1 && y != zonesY - 1 && y != 0)
                    cases = 7;

                if (cases == 100)
                    Debug.LogError("Cases fault");

                switch (cases)
                {
                    case 0:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                zoneList.rows[x + 1].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x +
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);



zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);
                        break;
```

```
                    case 1:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                zoneList.rows[x +
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x + 1].row[y-
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);



zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);
                        break;
                    case 2:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                 zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x - 1].row[y-
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);



zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);
                        break;
                    case 3:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                 zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x - 1].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;
                    case 4:
```

```
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,

zoneList.rows[x+1].row[y].GetComponent<WindSimulate>().transform.eulerAngles,
lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;
                    case 5:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                  zoneList.rows[x +
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;
                    case 6:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                  zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x +
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                  zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;
                    case 7:
```

```
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;
                    case 8:
                        tempVector =
Vector3.Lerp(zoneList.rows[x].row[y].GetComponent<WindSimulate>().transform.eulerAngle
s,
                                                zoneList.rows[x +
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x -
1].row[y].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y +
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);

                        tempVector = Vector3.Lerp(tempVector,
                                                zoneList.rows[x].row[y -
1].GetComponent<WindSimulate>().transform.eulerAngles, lerpFactor);


zoneList.rows[x].row[y].GetComponent<WindSimulate>().windTransformRotation.eulerAngles
= new Vector3(0, tempVector.y, 0);

                        break;

                    default:
                        Debug.LogError("Defult case");
                        break;


                }

            }
        }

    }
}
```

Tree.CS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Tree : MonoBehaviour {
    public GameObject seed;
    public Vector3 spawnPos;
    private float scale = 0.1f;
    public float timeAlive = 45f;
    private float timer;
       // Use this for initialization
       void Start ()
    {
        timer = Time.time;
        spawnPos = transform.position;
        spawnPos.y += 35f;
        InvokeRepeating("SpawnSeed", 10f, 0.7f);
        InvokeRepeating("updateSize", 0f, 2f);

       }

       // Update is called once per frame
       void Update () {

       }
    void SpawnSeed()
    {
        if(Globals.Instance.seedCount < Globals.Instance.maximumSeeds)
        {
            if (timer + timeAlive <= Time.time)
                Destroy(gameObject);

            float randomDirection = UnityEngine.Random.Range(0f, 360f);
            GameObject newSeed;
            spawnPos.y = Random.Range(transform.position.y + 10f, transform.position.y
+ 30f);
            newSeed = Instantiate(seed, spawnPos, Quaternion.identity);
            newSeed.transform.Rotate(new Vector3(0f, randomDirection, 0f));
            Globals.Instance.seedCount++;
        }
    }
    void updateSize()
    {
        if (scale > 1.5f)
            CancelInvoke("updateSize");

        gameObject.transform.localScale = new Vector3(scale, scale, scale);
        scale += 0.1f;
    }
}
```

SeedAgent.CS

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SeedAgent : MonoBehaviour
{
    public GameObject tree;

    public float maxSpeed = 10f;
    private float mass;
    public float separationWeight = 1f;
    public float AlignmentWeight = 0.5f;
    public float cohesionWeight = 0.6f;
    public float wanderWeight = 0.6f;
    public float wanderRadius;
    public float wanderJitter;
    public float wanderDistance;
    public float evadeWeight;
    public float findNeighborRange;
    public float timeAlive = 45f;
    private float timer;

    public float flyWeight = 1f;

    Vector2 wanderTarget;
    Vector3 velocity;
    private List<Tree> treeNeighbors = new List<Tree>();
    protected void Start()
    {
        timer = Time.time;

        mass = gameObject.GetComponent<Rigidbody>().mass;
        flyWeight = UnityEngine.Random.Range(0.2f, 0.45f);
    }

    protected void FixedUpdate()
    {
        Vector3 steeringForce = Vector3.zero;
        steeringForce += Wander() * wanderWeight;

        steeringForce = new Vector3(steeringForce.x, 0, steeringForce.z);
        Vector3 acceleration = steeringForce / mass;
        velocity += acceleration * Time.deltaTime;
        velocity = Vector3.ClampMagnitude(velocity, maxSpeed);

        transform.position += velocity * Time.deltaTime;
        if (timer + timeAlive <= Time.time)
        {
            Globals.Instance.seedCount--;
            Destroy(gameObject);

        }
    }
    private void OnTriggerEnter(Collider collision)
    {
        if (collision.gameObject.name == "Terrain")
        {

            PlantTree();
            Destroy(gameObject);
```

```
            Globals.Instance.seedCount--;
        }
        if (collision.gameObject.tag == "EndOfMap")
        {
            Debug.Log("Endofmap!");
            Destroy(gameObject);
            Globals.Instance.seedCount--;
        }
    }

    private void PlantTree()
    {
        float distance = FindNeighbors();
        if (distance > 10f)
        {
            float chance = Mathf.InverseLerp(0f, findNeighborRange, distance);
            float arrow = UnityEngine.Random.value;

            if (distance == 0 || arrow < chance)
            {
                Instantiate(tree, transform.position, Quaternion.identity);
            }
        }
    }

    private Vector3 Flee(Vector3 target)
    {
        Vector3 desiredVelocity = (transform.position - target).normalized * maxSpeed;

        return desiredVelocity - velocity;

    }

    private Vector3 Wander()
    {
        wanderTarget += new Vector2(UnityEngine.Random.Range(0, wanderJitter),
UnityEngine.Random.Range(0, wanderJitter));
        wanderTarget = wanderTarget.normalized * wanderRadius;
        Vector3 targetLocalSpace = wanderTarget + new Vector2(wanderDistance, 0);
        Vector3 target = transform.TransformPoint(targetLocalSpace);
        return target - transform.position;
    }

    private Vector3 Seek(Vector3 target)
    {
        Vector3 desirecVelocity = (target - transform.position).normalized * maxSpeed;
        return desirecVelocity - velocity;
    }

    private float FindNeighbors()
    {
        Collider[] collisionSpheres = Physics.OverlapSphere(transform.position,
findNeighborRange, LayerMask.GetMask(new[] { "Tree" }));
        float shortestRange = 100f;
        if (collisionSpheres.Length == 0)
            return 0f;

        foreach (Collider collisionSphere in collisionSpheres)
        {
            if (collisionSphere.gameObject != gameObject)
            {
```

```
                float dist = Vector3.Distance(collisionSphere.transform.position,
transform.position);
                if(dist < shortestRange)
                {
                    shortestRange = dist;
                }

            }
        }
        return shortestRange;
    }
}
```