DOLLYWOOD

"SUPER ADVANCED SHEEP SIMULATOR"

implementerat i Java

Operativsystem och multicoreprogrammering (1DT089) våren 2014.

Slutrapport för grupp 5

Viktor Andersson (900409-4471)

Jimmy Helmersson (920425-1970)

Marcus Münger (931007-5198)

Elin Parsjö (910915-5128)

Samuel Svensäter (831130-2957)

2014-06-07

Version 2.0

Innehållsförteckning

1. Inledning

Att försöka simulera naturen verklighetstroget kan vara fängslande och användbart för programmerare och animatörer. Detta är även vad som försökts åstadkommas i detta projekt. Tanken med projektet var att simulera det ekosystem som kan tänkas existera i en fårhage. Resultatet blev *DollyWood*, dock mer eller mindre verklighetstroget. I ekosystemet finns får som äter gräs, vargar som jagar och äter får, samt träd och gräs som växer eller torkar ut med avseende på vattenmängden i den begränsade världen. Tanken med en fårhage är enkel att visualisera och lämpar sig väl att skapas i 2D eller 3D. En viktig anledning till att detta valdes som projekt var att det fanns tydliga fördelar med att låta de komponenter som skulle simuleras köra på egna trådar. En annan anledning är att det enkelt gick att begränsa eller utöka antalet komponenter som ingår i systemet, samt att välja hur förfinade egenskaper dessa ska ha. Detta gjorde projektet lätt att anpassa efter olika situationer och det är därför open ended. Simulationen begränsades till får och vargar som de enda djuren och träd och gräs som de enda växterna, detta går dock att bygga ut vid behov.

I den här rapporten ges en översikt av systemet, det beskrivs först från en användares perspektiv och sedan beskrivs dess arkitektur mer ingående. Systemets implementation, datastrukturer och de algoritmer som använts förklaras ingående. Slutsatser om projektet diskuteras och det reflekteras över vad som gjorts och inte gjorts. Sist finns ett appendix innehållandes utvecklingsverktyg och övrig information om systemet.

2. Översikt av systemet

När systemet startas kommer användaren till en startmeny där inställningar kan göras för simuleringen. De inställningar som finns är rörande vatten, moln, dag/natt, gräs, träd, får, vargar, grafik och världsgenerering. De går bland annat att ställa in antal träd, får och vargar och hur mycket av världen som ska renderas. När inställningarna är gjorda kan användaren välja att gå in i *DollyWood*, vilket är själva simuleringen.

Simuleringen startar med att en svärm fjärilar flyger förbi och att en *Dollywood*skylt svävar bort i fjärran. Startvyn är i luften, ovanför land och inuti en *skydome* som är som ett skal runt världen. Det går att röra sig i världen som är uppbyggd av ett hexagonalt rutnät där varje ruta är jord som kan innehålla antingen vatten eller gräs. Rutnätet fortsätter oändligt för

användaren men egentligen är det samma värld som upprepar sig. I världen kan man se träd som växer eller vissnar beroende på den vattenmängd som finns i marken. Det finns dag och natt i världen och under natten blir världen mörk. Det finns även får och vargar vilka dricker vatten och antingen äter gräs eller får. Djuren kan både föröka sig och dö, och dör då av törst eller hunger. Fåren kan även dö genom att bli uppätna av vargarna som har förmågan att jaga fåren när de är hungriga. Fåren har ett flockbeteende vilket innebär att de prioriterar att gå nära andra får.

Genom att trycka på *TAB-tangenten* inne i simuleringen dyker det upp en dialogruta i vilken det går att se statistik och göra samma inställningar som i startmenyn i början. Statistiken beskriver antal får, träd och vargar och över vatten- och gräsnivå över tid.

2.1 Systemdesign

Systemet är hårt knutet kring två moduler, behovsystemet som finns i *NeedsController* samt *Settings* som ligger i *Globals*. Dessa tillåter en modulär simulering med möjlighet att lägga till fler simulerade objekt utan att behöva ändra i gammal kod.

Settings tillåter en klass att lagra och hämta variabler globalt, samt tillåter ändring av dessa dynamiskt i *GUI*:t.

NeedsController tillåter alla modulerna att påverka varandra utan att ha en direkt koppling mellan varandra. En klass registrerar sig själv för att påverka ett behov positivt eller negativt, exempelvis så har dag/natt-cykeln positiv inverkan på solljus medan molnen har negativ inverkan då de skapar en skugga under sig.

Som bilden, *fig 2.0*, nedan visar finns det många delar som är parallella. Alla dessa är trådar så det är ingen direkt skillnad på parallellismen mellan modulerna. Det har inte behövts några speciella *concurrency*-modeller för att få *concurrency*:n att fungera som den ska, utan lås av olika slag har visat sig lösa de problem som uppstått. Bilden visar inte det optimeringsdrag som gjorts i *Race* där en semafor har använts för att hindra att hundratals djur arbetade samtidigt och segade ner hela systemet. Detta var anledningen till att ett lås användes för att begränsa antalet aktivt körande trådar till 20 per ras. En *binär semafor* används även när djur ska förflytta sig för att de inte ska kunna ställa sig på samma ruta samtidigt.

Majoriteten av klasserna är i stora drag självförklarande; *Graphics3D* håller i all 3D-grafik; *Sheep, Tree* och *Wolf* är simulerade får, träd respektive vargar; *Model*-gruppen sköter

hantering och inläsning av 3D-modeller och så vidare. Detta gäller däremot inte för alla klasser, exempelvis *HexagonUtils* vilken underlättar konverteringen mellan ett rutnät och ett hexagonnät. Den gör detta främst genom att tillhandahålla en funktion som givet en punkt returnerar en lista med alla hexagoner som angränsar till den punkten. Klassen *Race* håller i alla djur som tillhör en och samma ras, exempelvis får eller vargar - den gör detta genom att hålla i en 2D-array med referenser till varje djur. Denna lösning skapar ett väldigt snabbt system då man kan ta ut ett djur på en specifik plats i konstant tid.

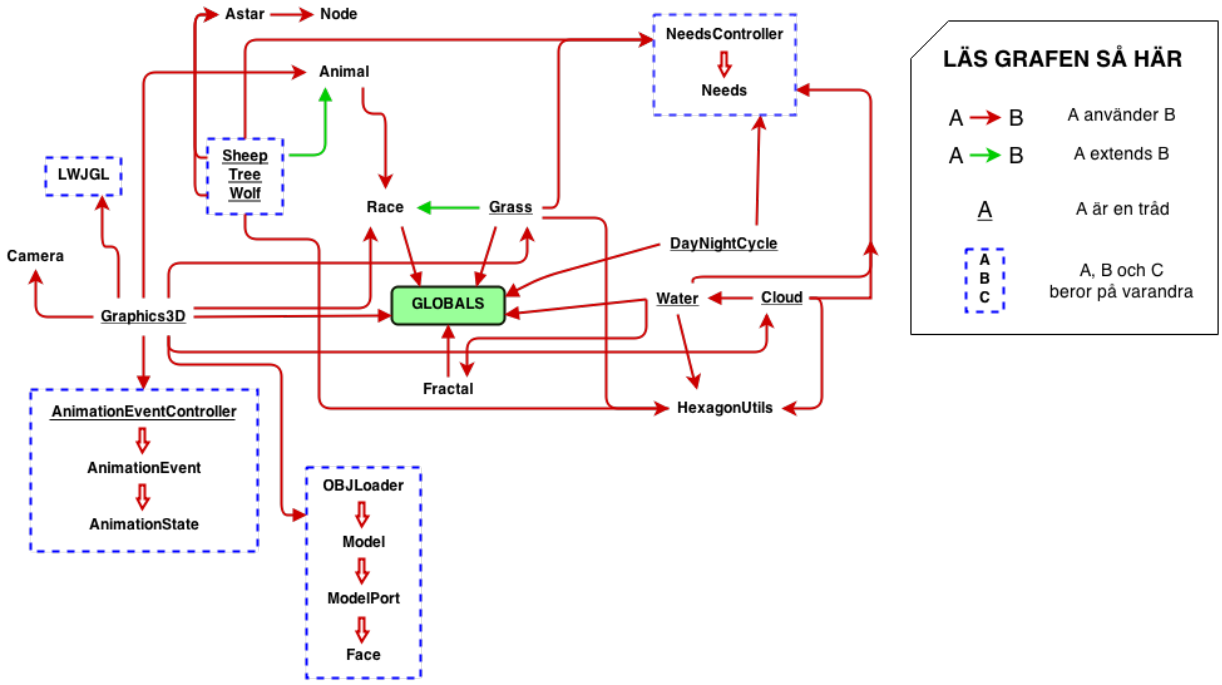

Fig 2.0 Systemarkitektur

3. Implementation

Projektet är implementerat i *Java* och som *IDE* har programmet *Eclipse* använts. Samtliga gruppmedlemmar hade erfarenhet av *Java* sedan tidigare vilket gjorde att arbetet snabbt kunde påbörjas utan för hög inlärningströskel. Ytterligare anledningar till att *Java* valdes var att det lämpar sig väl för grafiska applikationer, har bra bibliotek för concurrency samt är väldokumenterat. Det finns programmeringsspråk som lämpar sig bättre för concurrency men då projektet till stor del även inriktades på grafik valdes *Java* då detta har relativt enkel hantering av båda. Initialt var tanken att använda *actors* som concurrencymodell men detta visade sig svårt att

lösa i *Java*. Som actorbibliotek testades både *Akka toolkit* och *Jumi*, men på grund av komplikationer att få det att fungera som önskat togs beslutet att använda trådar istället. Simulerade objekt placerades på trådar vilket ledde till behov av synkronisering då objekt interagerade med varandra på olika sätt.

3.1 A* - kortaste vägen algoritm

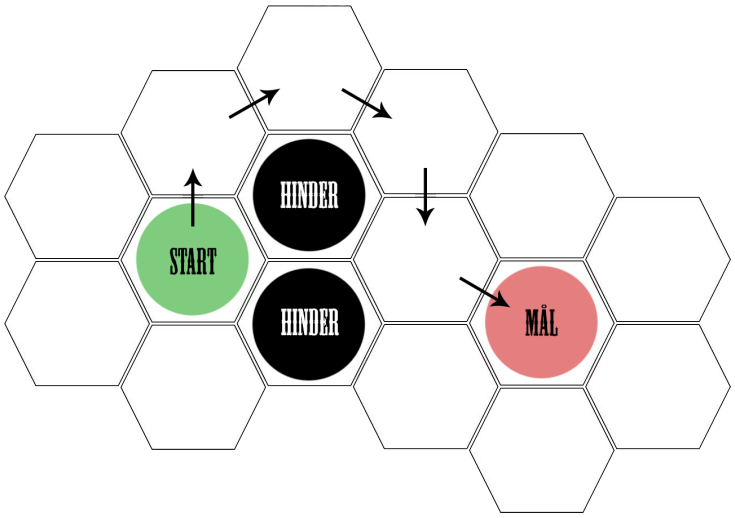

Fig 3.0 visar kortaste vägen mellan start och mål med hänsyn till hinder

A*-algoritmen valdes för att hitta det kortaste avståndet mellan två punkter, se *fig 3.0*, på engelska kallat *pathfinding*. A* används för att beräkna avstånd på ett statiskt rutnät, varför den passade projektet bra. Till en början ges en startposition och en slutposition. Startpositionen läggs till i en lista, som kallas *open-listan*, vilken innehåller positioner som inte är genomgångna. Algoritmen är som följer:

- Startnodens intilliggande noder läggs till i *open-listan*, om dessa inte är hinder som det inte går att ta sig igenom, exempelvis träd.
- Därefter tas startnoden bort från *open-listan* och läggs till i en *closed-lista* som innehåller besökta noder.
- Nästa steg är att välja den nod i *open-listan* med lägst totalkostnad (*), och göra denna till aktuell nod.

(*) Kostnaden att ta sig från startnoden till den aktuella noden kallas *G-kostnaden*, och kostnaden från den aktuella noden till slutpositionen kallas *H-kostnaden*. Summan av dessa två kallas totalkostnaden (*F*). Se *fig. 3.1.*

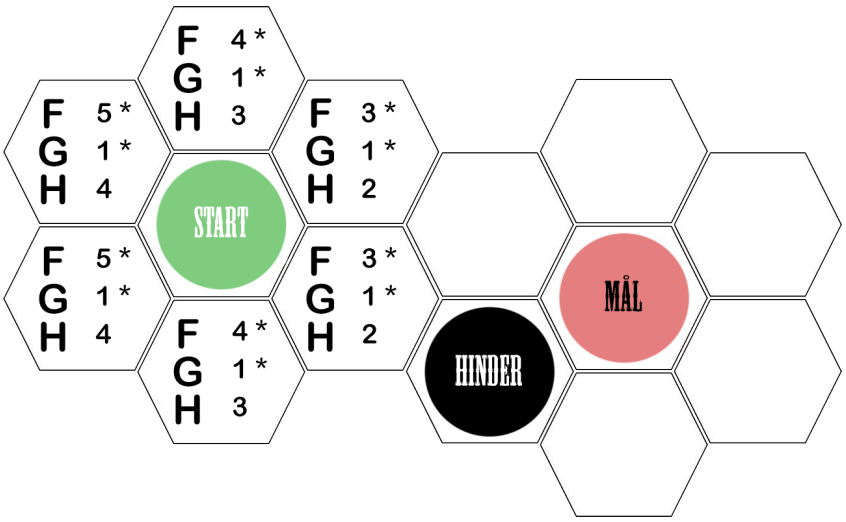

Fig 3.1

Normalt beräknas kostnaden att röra sig från en nod till en annan i diagonal riktning till 14 och att röra sig rakt uppåt eller rakt åt sidan till 10. Detta då rutnätet består av kvadrater och kostnaden beräknas med hjälp av Pythagoras sats. I simuleringen representeras världen av ett hexagonalt rutnät och kostnaden att röra sig i alla riktningar är därför lika stor. Däremot är en extra kostnad adderad för att röra sig i svår terräng, beroende på om beräknad rutt går upp eller ner för berg/sluttning. Observera att denna extra kostnad inte är med i *fig 3.1.*

Det finns olika sätt att beräkna *H-kostnaden* och en vanlig metod, som används i denna implementation, kallas Manhattan-metoden vilken innebär att hänsyn inte tas till eventuella hinder när kortaste vägen till slutpositionen beräknas.

- När noden med den lägsta *F-kostnaden* valts läggs den till *closed-listan*.
- De intilliggande noderna placeras i *open-listan*, om de inte redan finns där eller är hinder som inte går att passera. Den valda noden blir nu förälder till de nya noderna.

- För de noder som redan finns i *open-listan* måste nu en speciell kontroll göras. Detta steg är lättast att förstå genom att titta på *fig 3.1* och gå igenom algoritmen från början. Först ska den nod med lägst *F-kostnad* av de till startnoden intilliggande noderna väljas till nästa nod att besöka. *Fig 3.1* visar att *F-kostnaden* för den, sett från startpositionen, ”sydöstra” noden och den ”nordöstra” noden är samma, nämligen 3. Låt säga att den ”sydöstra” noden väljs till nästa nod (den nu aktuella noden). Två av de intilliggande noderna till denna nod finns redan på *open-listan*, nämligen den ”sydväst” respektive den ”norr” om den aktuella noden. Nu måste en koll göras för att kontrollera om vägen från startnoden genom den aktuella noden till någon av dessa noder är bättre än den väg som noderna för tillfället känner till. Den bästa vägen beräknas med hjälp av *G-kostnaden*. *Fig 3.1* visar att vägen från startpositionen, genom den aktuella noden, till noden ”sydväst” om den aktuella noden är 1+1=2. På samma sätt blir vägen genom den aktuella noden till noden ”norr” om den aktuella noden 1+1=2. Det fanns alltså ingen bättre väg till någon av noderna då *G-kostnaden* för att gå direkt till respektive nod från startnoden är 1. I detta fall behöver ingenting göras och algoritmen fortsätter genom att leta efter nästa nod att besöka. Om det i stället hade varit så att vägen till exempelvis den ”norra” noden hade varit bättre om man hade gått genom den aktuella noden, så hade den aktuella noden satts till förälder, och därigenom pekat, till den ”norra” noden. Den ”norra” noden hade då också blivit tilldelad en ny *F-* och *G-kostnad.*

Denna procedur upprepas tills det att antingen slutpositionen är hittad och läggs till i *closed-listan*, eller att slutmålet inte hittats samtidigt som *open-listan* är tom. Se *fig 3.2* för en bild där *F*, *G* och *H*-värden är beräknade för noder från start till mål. *Closed-listan* innehåller kortaste vägen från start till mål.

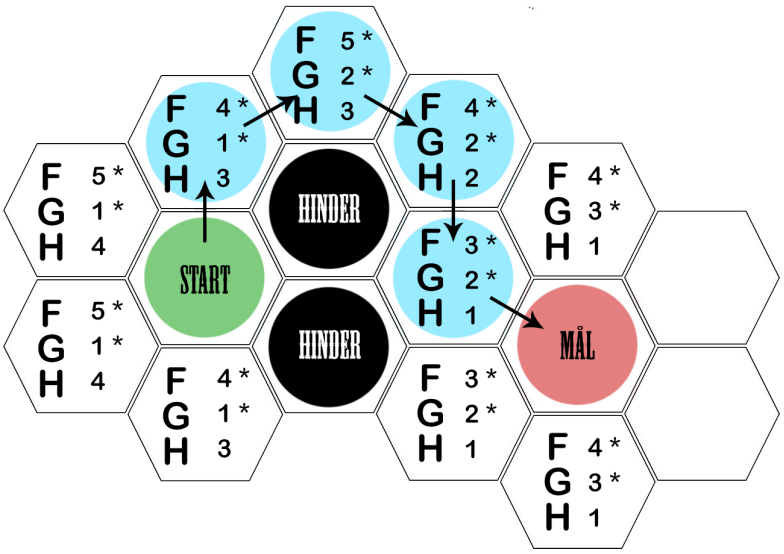

Fig 3.2

3.2 Diamond-square - Generera slumpmässiga fraktaler

Diamond-square är en algoritm som fyller en 2D-array med flyttal. Dessa flyttal är inte helt slumpmässiga då deras värden beräknas genom medelvärdet av andra tal på specifika positioner i arrayen. Detta medför att arrayen som genereras kan användas för att simulera höjdskillnaden i en värld, vilket är precis vad den huvudsakligen används till i detta projekt. Anledningen till att *Diamond-square* valdes var att den är relativt enkel att implementera medan den ger ett resultat som passar detta syfte perfekt. Eftersom denna algoritm enbart används under initialiseringen av världen så finns det ingen större anledning att tillämpa concurrency, då simulationen inte skulle bli lättare under körning av detta.

Förklaring av algoritmen:

Diamond-square består egentligen av två steg, efter att ytterst lite förarbete har utförts. Till att börja måste hörnen i arrayen initieras till några värden. Dessa värden kan vara helt slumpmässiga, inom ett intervall som passar situationen, eller så kan dessa fyra värden vara identiska. I regel används fyra identiska värden om *diamond-square* ska implementeras så att flera fraktaler kan läggas bredvid varandra och fortfarande se sammanhängande ut. Efter detta finns det två steg som ska upprepas, som kommer att förklaras nedan. De olika färgerna på

bilderna tolkas så att rött är värden som finns innan steget börjar och grönt är värden som beräknas i steget.

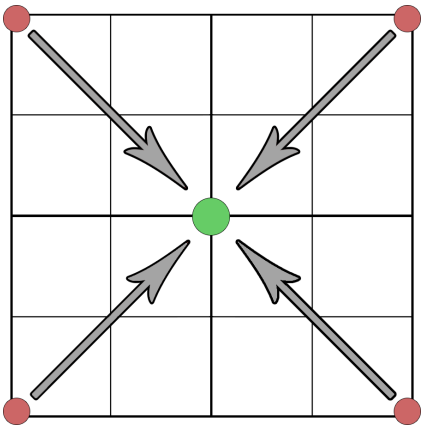

Fig 3.3 - The Diamond Step

The Diamond Step: Fyra hörn från en kvadrat adderas och divideras sedan med fyra. Detta skapar medelvärdet av hörnen. Detta värde plus ett litet slumpmässigt tal sparas sedan på positionen som är exakt i mitten av kvadraten, dvs. där diagonalerna möts.

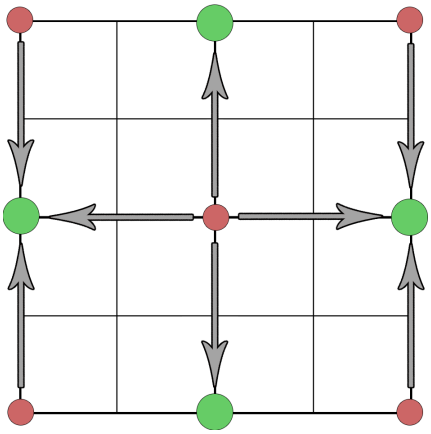

Fig 3.4 - The Square Step

The Square Step: Efter föregående steg har så kallade diamanter skapats och i detta steg beräknas mittpunkterna på diamanterna. Dessa värden beräknas på liknande sätt som tidigare, medelvärdet av hörnen på diamanten beräknas sedan adderas ett slumpmässigt tal. Detta tal sparas i mitten av diamanten. Värt att tillägga är att det kommer att genereras diamanter vars mittpunkt ligger exakt på en kant av arrayen, detta medför att dessa diamanter enbart har tre hörn, så då beräknas medelvärdet av dessa tre istället.

En sekvens för att fylla en array som är 5x5 står ser ut enligt följande:

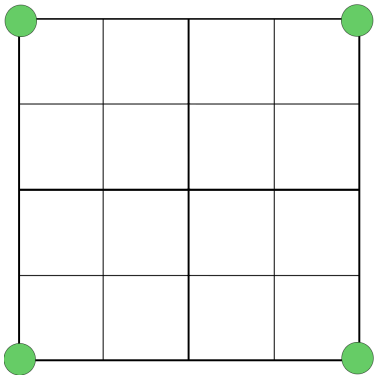

Fig 3.5 Initialisering av hörnen

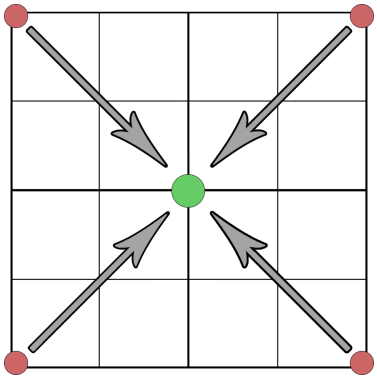

Fig 3.6 The Square Step

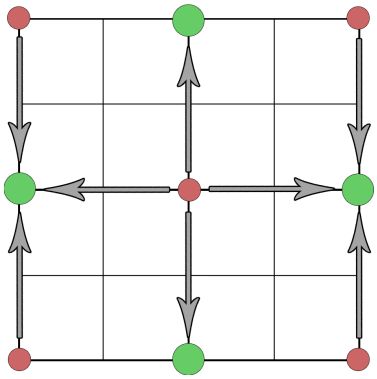

Fig 3.7 The Diamond Step

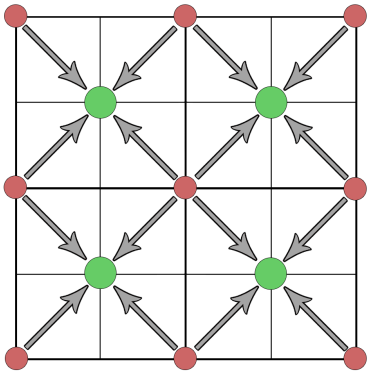

Fig 3.8 The Square Step

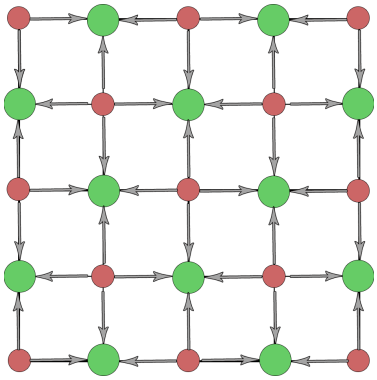

Fig 3.9 The Diamond Step

I *fig 3.5* sparas värden i hörnen på arrayen och i *fig 3.6* utförs det första steget, *the square step*. Sedan utförs *the diamond step* i *fig 3.7.* Detta upprepas i *fig 3.8* och *fig 3.9* tills arrayen är helt fylld med tal. För att det ska fungera att använda detta tillvägagångssätt måste arrayen vara helt kvadratisk och sidorna måste även vara av längden $2^n + 1$, dvs. 3, 5, 9, 17, 33 etc.

3.3 Concurrency

Concurrency har tillämpats i detta projekt utanför kända algoritmer helt enkelt eftersom det inte hade påverkat slutresultatet på något positivt sätt om dessa kända algoritmer hade parallelliserats. Som nämnt tidigare har *concurrency* tillämpats på andra ställen, tillexempel på alla djur och träd. Alla dessa objekt

körs på enskilda trådar. Detta medför att allt slags minne som delas måste synkroniseras. Det största delade minnet som måste synkroniseras är världen som alla objekt rör sig på. Men om hela denna array hade synkroniserats hade det blivit en enorm flaskhals i projektet och stoppat nästintill all *concurrency* och istället blivit sekventiellt till stor del. Lösningen på detta var att skapa en 2D-array av lås som har samma storlek som världen, där alla objekt måste låsa enskilda celler i denna array innan de flyttar på sig. Detta förhindrar olika objekt att existera på samma position.

De olika objekten i världen är även i sig själva ett slags delat minne då de måste kunna kommunicera med varandra. En binär semafor implementerades i varje *animalobjekt*, och denna låses upp medan objekt på något sätt sover, det vill säga när objekten står stilla och inte behöver utföra några beräkningar. Detta händer när simulationen kräver att objekten tar en paus för att det ska se mer verklighetstroget ut. Detta görs till exempel efter varje steg objekten tar så de inte går alldeles för snabbt. Det görs också när de äter och dricker. Denna binära semafor garanterar att objekt inte håller på med någonting annat när de börjar kommunicera med varandra.

När synkroniseringen sker med hjälp av lås finns det risk för *deadlock*. I detta projekt har kriteriet *Circular Wait* försökts undvikas genom att aldrig låta ett objekt hålla i ett lås medan det försöker låsa upp ett annat. Dock måste detta i någon mening ske, till exempel när de olika objekten rör sig i världen. De håller då sitt eget lås, medan de försöker låsa upp en cell i arrayen som representerar världen. Lösningen på detta är att ett objekt bara försöker gå till en viss plats ett begränsat antal gånger, och lyckas det inte efter dessa försök “ger den upp”. När objektet ger upp låser den upp sitt eget lås för att kontrollera om någon vill kommunicera innan den försöker röra på sig igen. Detta gör att objekten aldrig kommer att fastna i *deadlock*.

4. Slutsatser

Under projektets gång har en simulation av en fårhage skapats. Simulationen består av växter och djur med olika behov som på olika sätt kan interagera med varandra. Användaren kan innan simulationens start och under simulationens körning påverka olika värden för att styra eller ändra parametrar såsom väder, dag/nattcykel, gräs, får etcetera.

Att automatiskt testa en simulering är svårt då mycket beror på slumpvärden och realtidsbeteenden som uppkommer under simuleringens gång. Detta har gjort att vi inte har fullständiga tester, dock har statistiken, vilken uppdateras i realtid, använts för tester.

En av de svårare aspekterna i detta projektarbete har varit att få de olika delarna som ingår i projektet så effektiva som möjligt, så att slutprodukten inte blir för beroende av bra hårdvara. En huvudsaklig lösning för oss var att begränsa antalet objekt som jobbar samtidigt under simulationen med semaforer.

För den artificiella intelligensen till de olika objekt som simuleras finns det en hel del logik som kan skrivas smartare och effektivare. Det finns även några algoritmer som skulle kunna implementeras bättre, istället för några av de lösningar som vi har gjort. Det finns t.ex. en algoritm för flockbeteende som vi gärna hade använt oss av istället för vår lösning, men som blev utelämnad på grund av tidsbrist.

Något vi önskar att vi börjat med tidigare är projektrapporten. Det hade varit bättre att skriva om det vi gjort, så som algoritmerna, precis efter vi gjort dem. Det hade spridit ut arbetet med rapporten och vi hade haft färskare minne om vissa saker.

Detta har varit ett relativt stort projektarbete och genom att vi till största delen har arbetat tillsammans som grupp i skolan har vi enkelt kunnat ta del av varandras kod och arbetat tillsammans. Vi har kunnat bolla saker med varandra och utbytt tankar om möjliga lösningar kring problem som uppkommit under arbetets gång. Vi upplever att vi har fått mycket gjort och är nöjda med vad vi har åstadkommit.

Appendix: Installation och utveckling

Utvecklingsverktyg:

- Java version 7 användes som programmeringsspråk.
- Eclipse användes som IDE.
- Egit användes för hantering och delning av kod. Det är ett plugin som förser Eclipse med git som är ett versionhanteringssystem.
- JUnit användes för att testa kod.
- Javadoc användes för dokumentation.
- Google Drive användes för lagring av planering och övrigt material.

Projektet kan laddas ner från: https://github.com/Peter-Odd/DollyWood.

Systemet kompileras i Eclipse om man importerar det projekt som laddats ner från git. Systemet startats i Eclipse genom att trycka på run-knappen eller genom att köra följande kommando i terminalen "*java -Djava.library.path=lib\native -jar DollyWood.jar*" med *DollyWood/* som aktivt biblotek. Det finns även en *run.bat* fil för Windows-användare som gör just detta.

Katalogstruktur:

Det finns en src/ mapp och i denna mapp ligger det undermappar med namn som representerar varje klasspaket. I res/ mappen ligger projektets externa resurser, som exempelvis alla 3D-modeller och bilder. Det finns även en lib/ mapp som innehåller externa paket, i detta fall dock endast LWJGL. Den sista mappen av relevant vikt är doc/ i vilken man kan hitta dokumentation för alla klasser. I den mappen kan man hitta en *index.html*-fil som är utgångspunkten för all dokumentation.