Praktikrapport 2017

Inlämnad 8.9.2017

Mikko Finell – 1600167

CASE står för Cellular Automata Simulation Engine och är resultatet av min praktik sommaren 2017. Rapporten börjar med en beskrivning av min motivation för göra ett projekt som detta, och en kort beskrivning av vad cellulär automata är samt exempel på klassiska automata. Efter det följer en genomgång av vilka tekniska mål jag hade när jag började, sedan ett rättfärdigande av dom teknologier jag valde använda mig av för att nå mina mål.

Huvudsakligen beskriver rapporten den överliggande strukturen hos CASE samt dom viktigaste algoritmerna som driver motorn. Vidare så analyseras motorns prestanda i olika typiska situationer, t.ex. hur effektivt den klarar av att simulera diverse enkla automata, och vad dess svaga och starka sidor är.

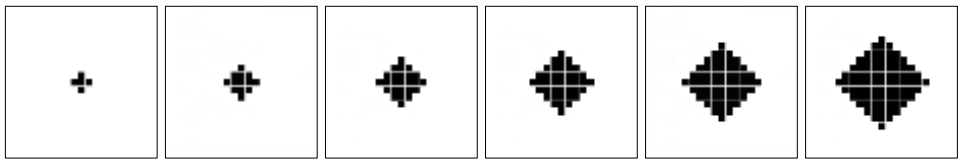
Avslutningsvis behandlas resultaten av den ovannämnda analysen, hur väl motorn möter mina mål. Jag diskuterar vilka lärdomar jag dragit av detta arbete, och ger en personlig utvärdering av projektet som helhet.

Innehåll

1 Cellulär Automata

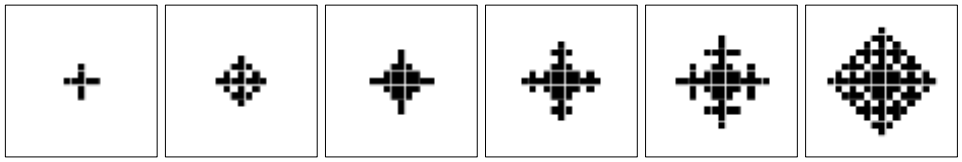
Cellulär Automata, härefter CA, är en datavetenskaplig modell som består av celler, där varje cell har någon regel om hur den reagerar till celler i sin omgivning. I en CA simulering uppdateras cellerna synkront och man undersöker vilka mönster och beteenden som uppstår från olika regler. Dom vanligaste CA modellerna, och i synnerhet dom som CASE är gjort att simulera, har sådan struktur att cellerna bildar ett tvådimensionellt rutnät. I teoretiska modeller sträcker sig detta nät ofta oändligt i alla riktningar, dock är detta i opraktiskt att implementera med begränsad hårdvara, så de flesta implementationer väljer att bestämma någon ändlig dimension som utgör världen, och kombinera detta med en regel hur gränsernas celler hanteras.

Exempel på simpel CA:



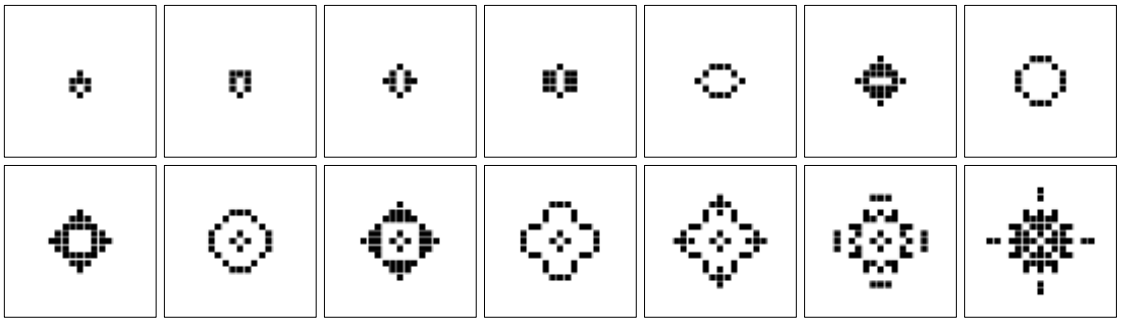
Figur 1. En CA med regeln att celler från början har vit färg, men om en av cellens grannar har svart färg så byter cellen själv till svart färg.

Exempel på en lite intressantare regel:



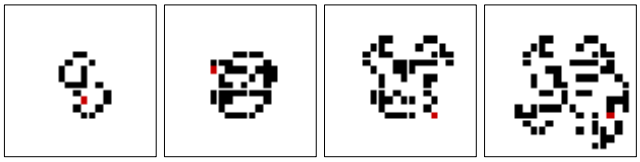
Figur 2. Denna CA har regeln att cellen är vit, men blir svart endast om 1 eller 4 av dess grannar är svarta. (Wolfram, 2002) Implementation finns i bilaga I.

Cellulär automata uppfanns i 1940 talet av Stanislaw Ulam och John von Neumann i deras studier av självreplikerande system. CA fick inte så mycket uppmärksamhet tills John Conway år 1970 uppfann reglerna som utgör den mest välkända automatan, kallad Game Of Life, eller Conway's Life (se bilaga H.)

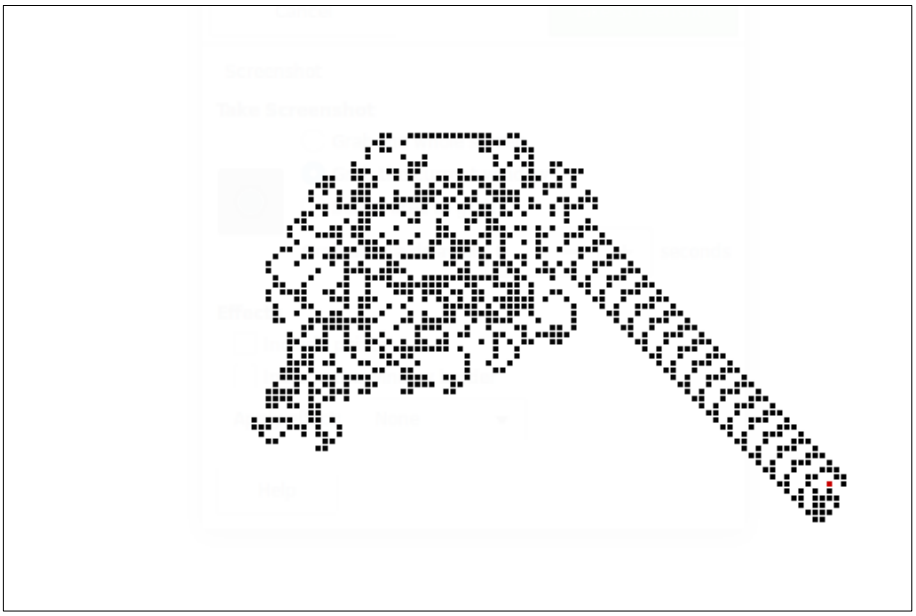


Figur 3. Life är en automata med reglerna att om en svart cell har färre än 2 svarta grannar eller fler än 3 svarta grannar blir vit, medans en vit cell med exakt 3 svarta grannar blir själv svart. Dessa simpla regler ger upphov till mönster som evolverar på ett mycket dynamiskt sätt.

Forskningsområdet fortsatte utvecklas och år 1986 uppfanns en annan välkänd CA av forskaren Chris Langton. Denna modell har kommit att kalls för Langton's Myra, och skiljer sig lite från dom tidigare modellerna vi har sett eftersom myran utgör en agent som flyttas runt i världen och manipulerar sin omgivning.



Figur 4. Myran som i dessa bilder representeras av den röda rutan följer regeln att om cellen den befinner sig på är vit så byter den färg till svart och myran vänder sig 90 grader till vänster och flyttar sig en cell framåt. Är cellen svart så blir cellen vit och myran vänder sig 90 grader till höger och flyttar sig en cell framåt. Se bilaga G.



Figur 5. Efter ungefär 10 000 uppdateringar hamnar myran i ett regelbundet mönster som upprepas i oändlighet.

Stephen Wolfram är en annan nämnvärd forskare som undersökt CA sedan mitten av 1980talet. Wolframs arbete ligger främst inom 1-dimensionella modeller med extremt enkla regler. Se bilaga J.

1.1 Dynamiska kontra Statiska CA

Man delar huvudsakligen in CA i två kategorier: Statiska och dynamiska.

Statiska modeller karakteriseras av att varje cell i världen består av en automaton, och tillståndet hos en enskild automaton är en funktion av granncellernas tillstånd. Exempel på statiska CA är Conway's Life.

Den andra typen är vad vi kallar dynamiska CA. I dynamiska simuleringar pratar man om agenter istället för automatons. Skillnaden är att agenter inte är bunden till någon cell utan kan flytta sig mellan celler. Vidare så kan agenter mutera andra agenters tillstånd. Exempel på dynamiska CA är Langton's Myra.

Dessa två modeller skiljer sig implementationsmässigt på följande vis: Statiska CA så uppdateras hela världen samtidigt, dvs från tillstånd t_0 så utför varje automaton sin uppdatering parallellt, vilket returnerar en ny automaton som funktion av sin omgivning. Dessa nya, uppdaterade, automatons samlas då ihop och vi kallar dom världens tillstånd t_1.

Hos dynamiska modeller måste varje agent nödvändigtvis bli uppdaterad i sekventiell ordning eftersom agenten kan mutera sin omgivning, vilket skulle leda till odefinierat beteende om agenterna uppdaterades parallellt.

2 Målsättning

Mitt främsta mål med CASE var att skapa ett verktyg som låter användaren enkelt skapa simuleringar av godtyckliga CA. Vad som menas i detta sammanhang med "skapa" innebär att användaren inkluderar i sin C++ källkod vissa moduler som utgör CASE, varefter användaren endast behöver definiera vilka regler simuleringens automata följer och hur dom skall renderas, samt några detaljer som t.ex. dimensionerna av rutnätet som utgör simuleringsvärlden. Allt annat, som att driva uppdateringen framåt, hantera mus och tangentbordsinput, att skapa det fönster var i simuleringen renderas, och annat dylikt, allt sådant skall hända automatiskt.

En annan kritiskt viktig men aningen mera subtil aspekt av användarvänlighet är huruvida användarna ändra hur själva motorn fungerar. I detta avseende har jag försökt maximera användarnas potential att anpassa produkten till deras behov genom att licensiera CASE under s.k. public domain, samt att göra källkodens funktionsmekanismer så uppenbara som möjligt.

Jag hade några sekundära mål gällande motorns prestanda. Att förutspå vad som är realistiska förväntningar utan att tidigare ha jobbat med dylika program är svårt, men jag bestämde att som minimum skall motorn kunna simulera en värld bestående av 192 gånger 108 (20736st) celler med 60Hz där automatan består av någon intressant modell. Vad som utgör en intressant sådan ligger i betraktarens ögon, men rimligen kan komplexiteten befinner sig i närheten av dom klassiska modeller som nämndes i föregående kapitel.

2.1 Val av teknologi

CASE är skriver i C++ version ISO/IEC 2014. Valet av programmeringsspråk baserades huvudsakligen på att jag redan är bekant med språket samt att C++ levererar den prestanda som krävs för att uppnå specifikationens mål. CASE använder mediabiblioteket SFML för att samla mus- och tangentbordsinput, samt som abstraktionslager över openGL för rendering.

Motorn är skapad och testad under operativsystemet GNU/linux Mint 17, och officiellt har jag bestämt att produkten är ämnad att fungera under en modern linux distro och en kompilator som stöder C++ version 14 och senare. Det är i nuläget okänt huruvida CASE går att kompilera under något annat operativsystem som t.ex. Windows, men det finns i princip inga hinder för det eftersom CASE egen källkod innehåller ingenting specifikt till linux, och SFML är portat till både Windows och OS X. Någon mobilversion är inte aktuell eftersom CASE är mest ämnat att fungera som ett datavetenskapligt forskningsverktyg för arbetsstationer.

3 Implementation

CASE är indelat i moduler som var för sig innehåller diverse C++ klasser och funktioner, vilka användaren kombinerar för att erhålla den funktionalitet som krävs till att skapa en simulering. Vissa moduler är främst ämnade för internt bruk i motorn, medans vissa andra existerar endast som byggstenar för simuleringar. Här följer en lista av alla moduler med en kort förklarning om deras innehåll.

agent_manager.hpp	Innehåller klassen AgentManager som används internt i dynamiska simuleringar för att uppdatera simuleringens agenter.
cell.hpp	Innehåller klassen ZCell vilken innehåller funktionalitet för att representera en cell med godtyckligt djup i bemärkelsen om hur många agenter som kan uppta samma cell.
dynamic_sim.hpp	Innehåller en funktion kör dynamiska simuleringar.
events.hpp	Används internt av modulerna dynamic_sim.hpp och static_sim.hpp för att hantera mus- och tangentbords input när en simulering körs.
grid.hpp	Används internt av dynamic_sim.hpp som abstraktionslager för att hantera celler.
helper.hpp	Här finns funktionerna clamp och wrap. Clamp ställer en given parameter att hålla sig inom gränsvärden. Wrap vecklar ett värde inom givet domän.
index.hpp	Innehåller funktionen index vilken konverterar x och y koordinater till ett index i en lista sorterad enligt rad-storleksordning.
job.hpp	Innehåller en abstrakt basklass kallad Job ämnad för att ärvas och specialiseras som byggsten i algoritmer som implementerar parallellism.
log.hpp	Innehåller klassen Log vilken är ett abstraktionslager som öppnar en fil och dumpar dit data. Ämnad för att samla data från simuleringar.
neighbors.hpp	Innehåller klasserna Adjacent och Neighbors vilka har bekvämlighetsmetoder för att få tillgång till en cells grannar.
pair.hpp	En abstraktion där vilket av två objekt som nås bestäms av en metod som vänder en variabels tillstånd.
quad.hpp	Bekvämlighetsfunktioner för rendering.
random.hpp	Innehåller diverse konfigurationer av slumpmässighetsmotorer.
static_sim.hpp	Innehåller en funktion kör statiska simuleringar.
timer.hpp	Innehåller en klass som bildar ett abstraktionslager för tidmätning med hög-precision.

Här följer en genomgång av fyra viktiga algoritmer som driver motorns funktionalitet.

3.1 Snabb tillgång till cellers grannar

Minneshanteringen av automatons, celler, och agenter har optimerats för cachevänlighet genom att dessa typer av objekt packas i kontinuerliga arrays. Man vill ofta i simuleringar komma åt en viss cell som funktion av dess x- och y-koordinater i världen, så frågan uppstår hur detta kan göras så effektivt som möjligt. I CASE har detta lösts genom att skapa en array av längden bredd × höjd, vari ett element identifieras av sitt index från följande funktion:

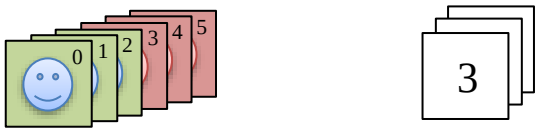
```
int index(const int x, const int y, const int columns) {
    return y * columns + x;
}
```

På detta sätt undviks något algoritmiskt sökande, och cellen nås på ett optimalt sätt.

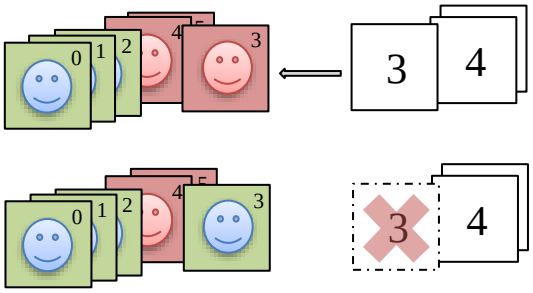
3.2 Minneshantering av agenter i dynamiska simuleringar

I dynamiska simuleringar finns möjligheten för agenter att skapas och förstöras, motorn behöver således en mekanism som hanterar denna allokering. I CASE har problemet lösts på följande vis: När motorn initialiseras skapas en array med default-konstruerade agent-objekt. Strax därefter görs en lista som innehåller ett index till varje agent, där indexet motsvarar agentens position i arrayen. När en ny agent skapas tas ett index ur index-listan och den agent vars position indexet pekar till blir aktiverad som en del av simuleringen.

När en agent skall skapas men listan av tillgängliga index är tom så itereras alla agenter, och indexen sparas för dom som är inaktiva.



Figur 6. Någon andel av agenterna är aktiverade (gröna), och dom inaktiverade (röda) agenternas index finns tillgängliga i en lista.



Figur 7. Simuleringens logik leder till att en ny agent skapas, ett index väljs och motsvarande agent aktiveras.

Figur 8. Den nyskapta agenten är nu aktiv i simuleringen, vilket betyder att den kommer uppdateras och renderas tillsammans med resten av dom aktiva agenterna. Dess index har tagits bort ur index-listan.

Genom att hantera agenternas livstid på detta vis vinner man två saker:

1. Cachevänlighet, eftersom objekten är tätt packade i minnet.
2. Undviker kostnaden att allokera ett nytt objekt när en agent skapas.

Nackdelen är att index-listan förbrukas periodvis vilket betyder att alla agenter måste undersökas huruvida dom är aktiva, och om antalet är stort kan processen ta lång tid.

3.3 Optimering via jämlöpande trådar

I statiska simuleringar används parallellism för att optimera uppdateringsprocessen, vilket i praktiken implementeras via C++ modulen std::thread. De två kritiska aspekterna av denna algoritm är först hur dom jämlöpande trådarna synkroniseras med huvudprocessen, och sedan hur trådarna sinsemellan delar arbetet på ett matematiskt sätt.

Trådarna synkroniseras med huvudprocessen genom vad jag kallar Klara-Färdiga-Gå mekanismen. Klara-Färdiga-Gå är ett signaleringsprotokoll där en styrande process säkert och effektivt hanterar ett antal subprocesser som utför något godtyckligt arbete. Protokollet indelas i tre stadier:

Stadie 1, Klara:
Styrprocessen väntar tills alla arbetare skickar signalen "Klar". När signalen mottagits från alla arbetare inleds nästa stadie.

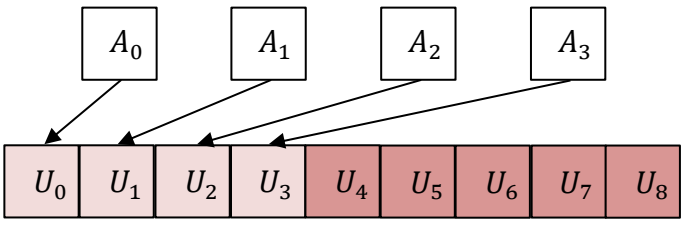
Stadie 2, Färdiga:
I detta stadie garanteras att alla arbetare ligger i ett väntande tillstånd, vilket betyder att det nu är säkert att från huvudtråden mutera deras tillstånd. Nu definieras arbetsbördan som kommer utföras, och den informationen sparas hos varje arbetare. Arbetarna väntar på signalen "Gå".

Stadie 3, Gå:
När styrprocessen skickar signalen "Gå" så börjar alla arbetare utföra det arbete som definierats i det föregående stadiet. När en arbetare har fullbordat sitt jobb skickar den signalen "Klar" till styrprocessen.

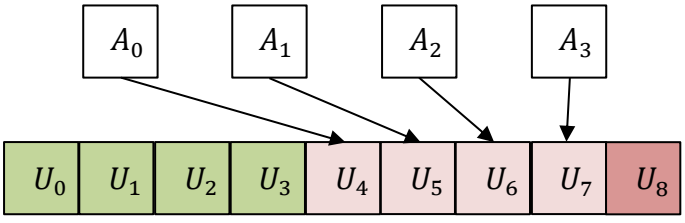
Under simuleringens gång upprepas dessa tre steg kontinuerligt för att driva världens tillstånd framåt. Arbetet som nämns i stadie 3 är alltså själva uppdateringen av automatons. Vi skall nu se hur arbetarprocesserna delar upp bördan sinsemellan på ett effektivt sätt. Frågan är om det finns ett antal arbetarprocesser, och ett stort antal uppgifter att utföra, hur vet en enskild arbetare exakt vilka uppgifter som hör till den? Man vill undvika följande problem:

Flera arbetare utför samma uppgift samtidigt → Leder till odefinierat beteende.
Samma uppgift utförs flera gånger sekventiellt → Slösar processortid.
Någon uppgift lämnar ogjord → Simuleringen är inkorrekt.

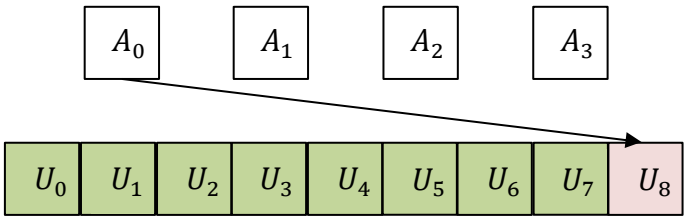
Vi undviker dessa problem genom att matematiskt definiera det subset av uppgifter en enskild arbetare skall utföra. Den information vi behöver är antalet arbetare som vi kan kalla A_{tot}, och antalet uppgifter som vi kallar U_{tot}, samt att uppgifterna existerar i en lista med bestämd ordning. Då kan problemet lösas genom att enumerera arbetarna från $A_0 \longrightarrow A_{tot-1}$ så att den n:tte arbetaren utför alla uppgifter från n, med n mellansteg, tills vi kommer förbi U_{tot}.



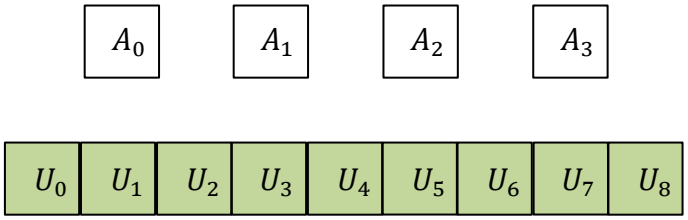
Figur 9. Om vi låter antalet arbetare $A_{tot} = 4$ och mängden uppgifter $U_{tot} = 13$ så börjar algoritmen genom att arbetare A_n utför uppgift $n + A_{tot}$



Figur 10. När arbetare A_n är klar med uppgift n utför arbetaren uppgift $U_{2n+A_{tot}}$.



Figur 11. Arbetarna fortsätter enligt samma mönster och väljer nästa uppgift genom att hoppa A_{tot} steg framåt i listan. Om hoppet tar arbetaren förbi U_{tot} så signalerar arbetaren att den är klar och lägger sig i stadie 2.



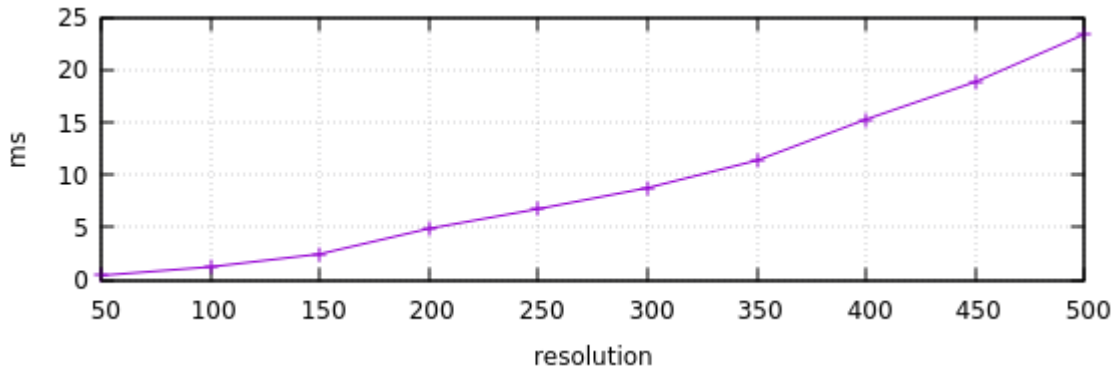
Figur 12. Alla uppgifter är slutförda och alla arbetare väntar på att styrprocessen skall signalera nästa steg i protokollet.

4 Prestanda

Enligt målsättningarna utvärderas motorns prestanda på basis av huruvida en given simulering inom viss resolution kan köras med 60Hz. Här måste beaktas hur komplex simuleringens automaton eller genomsnittliga agent är. Med samma resolution kommer en simulering med väldigt enkla agenter ha bättre prestanda än en simulering där agenternas uppdateringsfunktion innehåller komplex logik med många beräkningar, t.ex. tung användning av slumpmässighetsmotor.

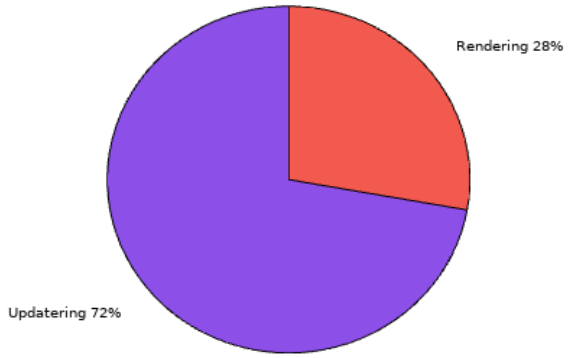
$$Prestanda = \frac{1}{Resolution * Komplexitet}$$

Prestandan har således ett invers förhållande till produkten av resolutionen och genomsnittliga komplexiteten. I följande diagram kan vi se hur detta förhållande uppenbarar sig i praktiken.



Figur 13. Genomsnittlig tid av 1000 uppdateringar som funktion av simuleringens resolution. Brians Brain automata (se bilaga B.)

Som diskuterades i kapitel 5 så uppdateras statiska simuleringar genom användning av parallellism. Detta är i praktiken arrangerat så att om processorn har n fysiska kärnor så används n-1 till uppdatering medans den sista kärnan används till att sekventiellt rendera simuleringens tillstånd. Detta försäkrar att alla processorkärnor utnyttjas på ett effektivt sätt.



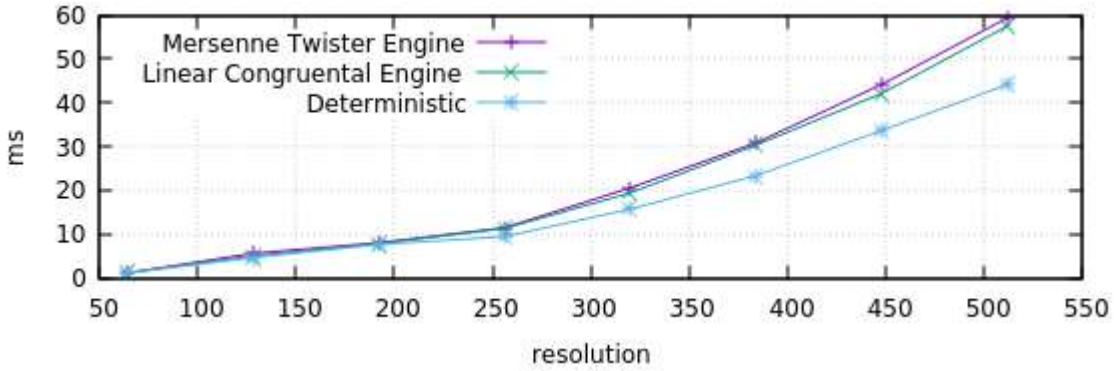
Figur 14. Genomsnittlig tid för uppdatering kontra rendering från 1000 generationer i Foxes And Rabbits.

I dynamiska simuleringar behöver uppdateringen nödvändigtvis ske sekventiellt eftersom agenter kan mutera andra agenters tillstånd. Prestandan hos dynamiska simuleringar är därför bestämd av två faktorer: Summan av varje agents uppdatering och varje agents rendering. Om vi godtyckligt väljer simuleringen Foxes And Rabbits (bilaga E) så illustrerar följande diagram faktorernas förhållande.

Prestandan är en funktion av komplexiteten hos simuleringens genomsnittliga cell, och cellens komplexitet är i sin tur en funktion av ett flertal faktorer. En nämnvärd faktor som spelar avgörande roll i många typer av simuleringar är valet av slumpmässighetsmotor. Standardbiblioteket i C++ erbjuder ett flertal implementationer, och CASE låter användaren välja mellan två slumpmotorer, samt valet att stänga av slumpmässighet:

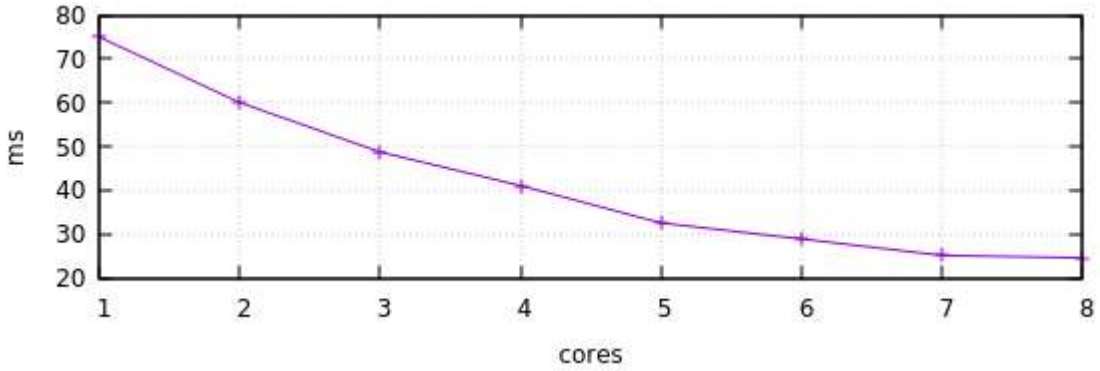
std::minstd_rand	"Linjär kongruensgenerator" är väldigt snabb slumptalsgenerator, men producerar slumpmässiga tal av låg kvalitet.
std::mt19937	"Mersenne Twister" producerar slumpmässiga tal av hög kvalité, men kräver en tyngre beräkning för att driva slumpmotorns tillstånd framåt.
CASE_DETERMINISTIC	Stänger av all dynamisk hämtning av frö-objekt, samt skippar mycket av motorns interna användning av slumpmässighet.

Valet av slumpmotor påverkar prestandan således:



Figur15. Jämförelse av de två slumpnummergeneratorerna som erbjuds i CASE. Diagrammet visar genomsnittlig tid i millisekunder hos 1000 generationer av simuleringen Foxes and Rabbits som funktion av ökande resolution. Vartefter resolutionen ökar så blir det deterministiska alternativet jämförelsevis mer effektivt, ca 15ms vid resolution 512×512. Simuleringen innehåller inte tillräckligt mycket slumpgenerering för att märka stor skillnad mellan std::minstd_rand och std::mt19937, ca 2ms vid 512×512.

Slutligen är prestandan i statiska simuleringar beroende på antalet fysiska processorkärnor som finns tillgängliga. Nedanstående graf demonstrerar inverkan på prestanda hos samma simulering när antalet kärnor ökar.



Figur 16. Genomsnittstid för 1000 generationer av simuleringen Conway's Life med av ökande antal kärnor som den oberoende variabeln. (Resolution = 512×512.)

Det är okänt hur länge förhållandet i grafen ovan fortsätter vartefter antalet kärnor ökar. Eftersom renderingen sker sekventiellt så förmodas det finnas en gräns när tidskostnaden för rendering dominerar, eller att antalet trådar blir så stort att tidskostnaden för synkroniseringen är större än själva uppdateringsarbetet.

5 Slutsats

Om man utgår från att en genomsnittlig simulering har en komplexitet som någorlunda liknar komplexiteten hos det godtyckliga urval av demo-simuleringar som visas i rapportens bilagor, så kan man gott säga att prestandan möter målsättningarna. Det främsta målet är att CASE kan användas som verktyg för att enkelt kunna definiera diverse automata och köra simuleringar. Det är svårt att bedöma objektivt huruvida jag har lyckats med det eftersom feedback från andra användare saknas. Den enda data jag kan använda för att utvärdera användarvänligheten är min egen uppfattning, och jag personligen tycker att CASE kan mycket väl nyttjas som hjälpmedel till att simulera en stor klass intressanta CA. Naturligtvis är jag partisk eftersom jag är intimt bekant med kodbasen och dom bakomliggande idéerna, men tills dess att jag kan få kritik från utomstående så kommer jag att vara nöjd med resultatet.

5.1 Vad jag har lärt mig

Jag har lärt mig en hel del om mjukvaru-utveckling från att arbeta med detta projekt. Vad jag anser är den viktigaste lärdomen är att analysera mina misstag. Ungefär vid mitten av projektets gång började jag dokumentera svårlösta buggar som upptäcktes i min kod. Jag dokumenterade varje bug från tre perspektiv:

Manifestation	Hur buggen upptäcktes och vilka symptom som uppenbarades i programmets exekvering.
De facto orsak	Hur buggen fungerar mekaniskt kontra hur koden är tänkt att fungera.
Underliggande orsak	En analys på mer abstrakt nivå om varför jag gjorde det logiska misstag som ledde till att jag implementerade en bug.

Att reflektera över dessa tre perspektiv anser jag har hjälpt mig enorm att bli en bättre programmerare.

Min strävan för att undvika misstag, i samband med dom insikter jag dragit från ovannämnda analys, har lett mig till lärdomen att programkod bör i första hand optimeras för robusthet, och först senare optimeras för prestanda. Eftersom jag ville ha god prestanda så gjorde jag många val angående kodens arkitektur som ledde till invecklade algoritmer och kopplingsanordningar. I framtida projekt kommer jag prioritera robusthet framför prestanda i planeringsskedet.

5.2 Vad jag skulle vilja lära mig inför nästa praktikperiod

Jag skulle vilja kunna beräkna hur komplex en algoritm är på ett matematiskt sätt. Förmågan att kunna objektivt bedöma komplexitet skulle ha varit till stor hjälp i planeringsskedet. Relaterat till detta vore förmågan att förstå assembly kod, så att jag kunde utvärdera vilken implementation av samma mekanism som blir mest effektiv efter att kompilatorn har gjort sin optimering. Felsökning och bugfixande hade gått smidigare om jag visste hur man använder en debugger, så det skulle jag verkligen vilja lära mig.

6 Referenser

Wolfram, S. (2002). i *A New Kind Of Science* (s. 171).