Examensarbete

HÖGSKOLAN I SKÖVDE

PRESTANDA HOS REALTIDSVÄXANDE TRÄD JÄMFÖRT MED STATISK MESH

PERFORMANCE OF REAL TIME GROWING TREES COMPARED TO STATIC MESH

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2016

Mattias Andersson

Handledare: Erik Sjöstrand
Examinator: Henrik Gustavsson

// Sammanfattning

Detta arbete undersöker prestandakraven hos realtidsväxande procedurellt genererande träd baserade på L-systemet genom att jämföra dessa med identiska träd representerade i form av statisk mesh.

Ett stokastiskt parametriskt kontextberoende L-system med hakparenteser har implementerats i en enkel testmiljö baserad på OpenGL (SGI 1997-2016), statisk mesh har implementerats i form av *display list* (SGI 1997-2016 kap. 5.4). Undersökningen har jämfört de två teknikerna baserat på renderingstid, minnesanvändning, och genereringstid av L-system, såväl som hur de båda teknikerna skalar med antal vertiser och L-systemslängd.

Resultaten visar på att de L-systemsbaserade träden har längre renderingstid än de representerade med statisk *mesh*, utöver tid för att växa träden via L-systemet före var rendering. De L-systemsbaserade träden visade dock på lägre minnesanvändning än sin statiska motsvarighet. Resultaten visar även på att de tekniker som jämförts inte nödvändigtvis var optimalt implementerade, och arbetet avslutas med en diskussion om hur detta skulle kunna åtgärdas i framtida arbeten.

Nyckelord: L-system, Lindenmayersystem, träd, växande, realtid.

Innehållsförteckning

1 Introduktion....1

2 Bakgrund....2

2.1 Procedurellt material....2
2.2 Användning....2
2.3 Motivation....2
2.4 Lindenmayersystemet....3
2.5 Deterministiska kontextfria L-system....3
2.6 Sköldpaddsgrafik....3
2.7 Stokastiska L-system....4
2.8 Kontextberoende L-system....4
2.9 Parametriska L-system....5
2.10 L-system med hakparenteser....6
2.11 Växande modellerad växtlighet....6

3 Problemformulering....9

3.1 Problem....9
3.2 Metodbeskrivning....9

4 Genomförande....11

4.1 Förstudie....11
4.2 Iterativ process....11
4.3 Programbibliotek....17
4.4 Pilotstudie....17
4.5 Användning av artefakt....19

5 Utvärdering....20

5.1 Undersökning....20
5.2 Resultat och analys....20

6 Avslutande diskussion....28

6.1 Sammanfattning....28
6.2 Diskussion....28
6.3 Framtida arbete....31

Referenser....36

1 Introduktion

Procedurell generation av material i datorspel gör det möjligt att producera stora mängder unikt material på kort tid och används för att generera allt från terräng och städer till texturer (Talton et. al. 2011). Ett användningsområde som lämpar sig bra för procedurell generering är växtlighet, då denna uppvisar stor självrepetition med mindre variation mellan individer (Lindenmayer & Prusinkiewicz 1990). En av de mer välanvända teknikerna för att generera procedurell växtlighet är L-systemet (Talton et. al. 2011), skapat av Arisitid Lindenmayer år 1968 med syfte att modellera alger (Lindenmayer & Prusinkiewicz 1990).

L-systemet är ett grammatiskt omskrivningssystem som skapar strängar av symboler baserat på ett axiom och produktionsregler. En grafisk representation av strängarna kan sedan skapas genom en teknik kallad sköldpaddsgrafik, där var tecken tolkas som en handling för en renderingsmekanism kallad för en sköldpadda.

Senare arbeten har expanderat L-systemet till att även simulera växt hos stående växter (Lam & King 2005), såväl som att generera träd i realtidsapplikationer. Baele och Warzee (2005) har producerat ett av det senare nämnda arbetena, och de hävdar att det är rimligt att använda realtidsgenererade träd även i tyngre realtidsapplikationer som till exempel spel.

Det här arbetet har strävat efter att undersöka en utvidgning av Baele och Warzees arbete inspirerat av Lam och Kings (2005) arbete om växande träd baserade på L-systemet. Målet har varit att applicera de viktigaste koncepten från Lam och Kings arbete på ett realtidssystem för att åstadkomma realtidsväxande L-systemsträd. Dessa har sedan utvärderats genom att jämföras med en av de vanligaste teknikerna för att representera träd i spel, det vill säga statisk mesh.

För att åstadkomma detta har en testmiljö skapats baserad på OpenGL (SGI 1997-2016) och SDL (SDL Community 2016). Ett stokastiskt parametriskt kontextberoende L-system med hakparenteser har implementerats tillsammans med ett system för att utföra sköldpaddstolkning av strängar. L-systemet tillåter en parameter per trädsegment som kan modifieras med de fyra räknesätten, dessa parametrar används för att kontrollera storleken på trädsegment och således används för att kontrollera växt hos trädet. Då systemet är stokastiskt används slump för att generera en stor mängd unika träd, slumpen använder sig av ett *seed* (eng. frö) för att göra det möjligt att generera samma träd flera gånger för testning. Systemets kontextberoende innebär att segment kan vara beroende av varandra, vilket används för att kontrollera hur stora träden tillåts växa samt tillåter avsmalnande grenar. Hakparenteser används för att skapa förgreningar i trädstrukturen och innebär att sköldpaddstolkningen reveterar till ett tidigare tillstånd. Detta används till exempel för att flytta tillbaka sköldpaddan till stammen efter att en gren renderats. Genom att växelvis iterera och rendera L-systemet är träden kapabla till att växa.

Statisk mesh har implementerats via OpenGLs *display list* (SGI 1997-2016 kap. 5.4), vilket är en teknik för att lagra en serie förkompilerade OpenGL-kommandon i grafikminne för snabb rendering. Applikationen genererar L-systemsbaserade träd baserat på ett *seed* (eng. frö). De träd representerade med statisk mesh skapas utifrån de L-systemsgenererade träden, för att på så vis tillåta en jämförelse mellan grafiskt identiska träd. Träden utgörs av kombinationer av simpel geometri i form av cylindrar, sfärer, och plan.

2 Bakgrund

2.1 Procedurellt material

Att skapa grafiskt material är en tidskrävande process, som till stor del behöver utföras för hand. Vissa typer av material kan dock med stor framgång procedurellt genereras. Exempel är texturer (Rhoades et. al. 1992), träd, landskap, städer och byggnader (Talton et. al. 2011), då dessa vanligen består av helt eller delvis repeterande mönster och former med mindre variationer.

Procedurellt genererat material har två huvudsakliga fördelar mot handgjort material. Först och främst tillåter det generering av stora mängder unikt material med lägre arbetsbörda än om det skapats för hand (Talton et. al. 2011). För det andra är det en möjlighet att utföra genereringen under körning av den användande applikationen, antingen i realtid eller under laddningstider, vilket kan resultera i lägre minnesanvändning både i primärminne och sekundärminne, samt att det är möjligt att påverka utseendet på det genererade materialet under körning (Magdics 2009).

2.2 Användning

SpeedTree (Interactive Data Visualization, 2015) är ett exempel på mjukvara för att skapa och använda procedurellt genererad växtlighet i bland annat spel. Mjukvaran erbjuder det producerade materialet i två huvudsakliga former: som ett format för användning i samband med SpeedTrees spelmotor-bibliotek, och i form av det mer traditionella formatet mesh, antingen statisk eller animerad, som används i majoriteten av moderna spelmotorer. Statisk mesh är geometri bestående av ett antal polygoner som kan lagras i grafikminne för effektiv rendering av komplexa former (Epic Games 2004-2015). SpeedTree har använts för att generera växtlighet till bland annat spel som Farcry 4 (Ubisoft Montreal 2014), The Witcher 3: Wild Hunt (CD Project RED 2015), och Saints Row IV (Deep Silver 2015).

2.3 Motivation

Spelare spenderar mycket tid i spel, ett exempel på ett populärt spel som det spenderats mycket tid på är The Elder Scrolls V: Skyrim (Bethesda Game Studios 2013), där medelspelaren spenderat 75 timmar på spelet (Todd Howard 2012). Tiden i spelet går snabbare än i verkligheten med ett förhållande på tre verkliga minuter per spel-timme. 75 timmar i verkligheten motsvarar därmed 62.5 dagar i spelet. Spelaren kan dessutom snabbspola spelet genom att till exempel sova och hoppa över restider, därmed kan hela dygn i spelet passera på ett par sekunder i verkligheten. Även om medelspelaren har 62.5 dagar effektiv speltid enligt spelets tideräkning är den förflutna tiden i spelet därmed säkerligen betydligt högre, och närmar sig magnituder där en verklig miljö skulle kunna genomgå en markant förändring i bland annat växtlighet.

Samtidigt är moderna spel fyllda med mekaniker utan direkt spelmekanisk påverkan. Träd som svajar i vinden, klädfysik, och att karaktärer lämnar fotsteg när de går i sand eller lera är alla exempel på effekter i spel som sällan tjänar ett syfte bortom att ge mer liv till spel. Att introducera växande växtlighet skulle vara ytterligare en sådan effekt.

2.4 Lindenmayersystemet

Ett av de vanligare teknikerna för att procedurellt generera material är Lindenmayersystemet (Talton et. al. 2011), även känt som L-systemet. L-systemet (Lindenmayer & Prusinkiewicz 1990) är ett grammatiskt omskrivningssytem, ursprungligen skapat med syfte att beskriva växtmönster hos alger, senare utvidgat även till större växtlighet. Systemet är dock kapabelt till att modellera även icke-biologiska strukturer, ett vanligt sådant användningsområde är fraktaler.

L-Systemet är ett iterativt system som agerar på en ursprungssträng, kallat ett axiom. Var iteration byter ut alla förekommande av specifika symboler baserat på ett antal omskrivningsregler, resultatet är en sträng vars uppbyggnad förändras under var iteration, tills iteration antingen upphör eller strängen består enbart av tecken för vilka det inte finns någon omskrivningsregel. Symboler för vilka det inte specificerats någon omskrivningsregel antas ersätta sig själva. De symboler som byts ut i omskrivningsregler kallas föregångare, medans symbolen de byts ut mot mot kallas efterträdare.

Den huvudsakliga skillnaden mellan L-systemet och andra grammatiska system är att omskrivningar sker parallellt, vilket innebär att samtliga förekommande av omskrivningsbara tecken ersätts samtidigt.

Följande beskrivningar av olika variationer på L-system, såväl som sköldpaddsgrafik, är sammanfattningar baserade på Lindenmayer och Prusinkiewicz (1990).

2.5 Deterministiska kontextfria L-system

Den mest grundläggande formen av L-system är deterministiska kontextfria system, som förkortas till DOL-system. I ett sådant system har var föregångare högst en efterträdare och var föregångare består av endast en symbol, efterträdare kan bestå av flera.

Följande exempel visar ett grundläggande deterministiskt L-system där ω är axiomet och *p1* och *p2* är produktionsregler:

$$\omega : a$$
$$p_1 : a \rightarrow ab$$
$$p_2 : b \rightarrow a$$

Utfallet över fem generationer blir då:

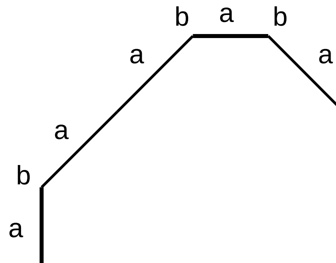
a
ab
aba
abaab
abaababa

2.6 Sköldpaddsgrafik

För att skapa en grafisk representation av den L-systemsgenererade strängen används vanligtvis sköldpaddsgrafik, där sköldpaddan är en representation av en position och en riktning i ett kartesiskt koordinatsystem. Var tecken i strängen tolkas som en handling av sköldpaddan, exempel på handlingar är att gå fram i nuvarande riktning med en viss

färdlängd, och att svänga ett specificerat antal grader i någon riktning. Sköldpaddan ritar därefter ut någonting när den rör sig, det kan vara en linje eller en geometrisk form, till exempel en cylinder med var sin ände vid sköldpaddans start respektive slutposition. Vanligtvis används även ett antal andra variabler, som till exempel tjocklek och färg, för att vidare kunna kontrollera renderingen.

Om sköldpaddsgrafik appliceras på strängen från L-systemet beskrivet under föregående rubrik, där a representerar ett steg rakt fram i nuvarande riktning av längd 1 och b en 45 grader rotation medurs, fås linjen i Figur 1.



Figur 1 Grafisk tolkning av strängen “abaababa”

2.7 Stokastiska L-system

Stokastiska L-system skiljer sig från sin deterministiska motsvarighet i att det för var företrädare kan finnas mer än en gällande produktionsregel med samma kontextkrav. Vilken produktion som skall appliceras kontrolleras därmed av en slumpfaktor, där var produktionsregel har tilldelats ett sannolikhetsvärde. Sannolikheten skall vara uppdelad på ett sådant sätt att någon av produktionerna alltid sker.

Exempel på produktionsregler för stokastiskt L-system där det upphöjda värdet efter pilen är sannolikheten för att produktionsregeln appliceras:

$$\omega : a$$
$$p_1 : a \xrightarrow{0.5} ab$$
$$p_2 : a \xrightarrow{0.5} ac$$

Följande fem generationer demonstrerar ett av flera möjliga utfall:

a
ac
abc
abbc
acbbc

2.8 Kontextberoende L-system

Genom att basera produktionsregler på symboler runtomkring föregångaren såväl som föregångaren själv kan betydligt mer komplexa beteenden modelleras än för kontextfria system. Det är dock fortfarande endast föregångaren som substitueras vid applicering av

produktionsregler. Om både en kontextberoende och en kontextfri produktionsregel gäller för en given symbol har den kontextberoende företräde. Kontexten kan sträcka sig över en eller flera symboler och i båda riktningar. Detta kallas för ett 2L-system, medans det kallas ett 1L-system om kontexten enbart gäller i en riktning. Mer generellt faller båda dessa varianter av systemet under kategorin (k, l)-system, där k är längden på kontexten åt vänster och l åt höger.

Exempel på produktionsregler för kontextberoende L-system där produktion endast sker om *a* finns till vänster om *b*:

$$\omega : abb$$
$$p_1 : a < b \rightarrow ab$$

Utfallet över fem generationer blir då:

abb
aabb
aaabb
aaaabb
aaaaabb

Värt att notera är att det är det första *b* från vänster i exemplet som substitueras, med *b* som den vänstra kontexten. Det högra *b* substitueras aldrig eftersom dess vänstra kontext alltid är ett *b*. Den öppna sidan på tecknet < är vänd mot föregångaren. En produktionsregel gällande den högra kontexten hade kunnat se ut på följande vis:

$$p_1 : b > a \rightarrow ab$$

Slutligen för ett 2L-system:

$$p_1 : a < b > a \rightarrow ab$$

2.9 Parametriska L-system

I parametriska L-system har var symbol en associerad variabel i form av ett reellt tal. Talet kan användas som ett kriterium för produktionsregler, och dess värde modifieras av produktionsregler.

Exempel på produktionsregler för parametriskt L-system:

$$\omega : a(0)$$
$$p_1 : a(x) : x < 3 \rightarrow a(x+1)$$
$$p_2 : a(x) : x \geq 3 \rightarrow b$$

Utfallet över fem generationer blir då:

$a(0)$
$a(1)$
$a(2)$
$a(3)$
b

Likt demonstrerat för b i exemplet ovan är det inte ett krav att alla symboler har en associerad parameter. För kontextberoende parametriska L-system kan kontexten inkludera parametrar såväl som symboler i kontextområdet.

2.10 L-system med hakparenteser

För att representera trädstrukturer med L-system kan en tillståndsstack användas. Genom att spara sköldpaddans tillstånd på stacken vid förgreningar i trädstrukturen blir det möjligt att återvända till förgreningspunkten efter att änden på en gren nåtts. Symbolerna *[* och *]* används för att lagra på respektive läsa från stacken, och placeras därmed i början och slutet av en gren.

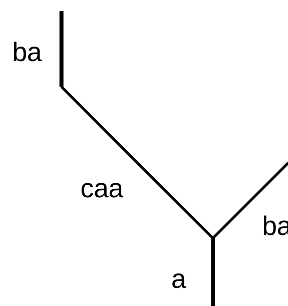
Exempel på produktionsregler för L-system med hakparenteser:

$\omega : a$
$p_1 : a \rightarrow a[ba]$
$p_2 : b \rightarrow ca$

Utfallet över tre generationer blir då:

a
$a[ba]$
$a[ba][caa[ba]]$

När strängen tolkas med sköldpaddsgrafik där a representerar ett steg rakt fram i nuvarande riktning av längd 1, b en 45 grader rotation medurs, och c en 45 grader rotation moturs, fås trädstrukturen i Figur 2.



Figur 2 Grafisk tolkning av strängen ”a[ba][caa[ba]]”

2.11 Växande modellerad växtlighet

Prusinkiewicz, Lindenmayer & Hanan (1988) redogör för vikten av utvecklingsprocessen för utseende på växtlighet, och menar på att genom att simulera hela växtens uppväxt kan ett mer realistiskt resultat uppnås än om enbart den önskade slutprodukten genereras direkt.

Arbetet fokuserar på örtartade växter och de naturliga kontrollmekanismer som påstås vara den huvudsakliga faktorn för dess utveckling, medan vedartade växter, för vilka miljö och konkurrens om utrymme är den kontrollerande faktorn, utelämnas. En lösning baserat på L-systemet, innehållande ett antal biologiska funktioner, redovisas, såväl som exempel på hur organ, däribland löv och blommor, kan implementeras med L-system.

Lam och King (2005) presenterar en variation på L-system för att simulera växt hos träd. Systemet använder sig av faktorer som spatial konkurrens, tillgång till ljus och vatten, hormoner, samt energiproduktion, för att framkalla realistiskt beteende hos träd under uppväxt.

Fototropism beskrivs och dess markanta påverkan på trädstrukturer poängteras. Fototropism är det fenomen som leder till att träd växer mot ljus, såväl som att delar av träd som tar emot mindre mängder ljus växer med ökad hastighet. Fototropism tros även ha en påverkan på träds förmåga att undvika kollision vid växt samt dess tendens att fylla tillgängligt utrymme.

Castellanos, Ramos och Ramos (2014) konstaterar avsaknaden av arbeten kring död hos simulerade växter, och hävdar att detta är av stor vikt för att åstadkomma en komplett modell av artificiella växter. De presenterar hur växtdöd kan representeras i fyra av de vanligaste variationerna på Lindenmayersystem, deterministiska, parametriska, stokastiska, och kontextberoende Lindenmayersystem.

För deterministiska Lindenmayersystem föreslås ett system där var segment efter ett givet antal iterationer övergår till ett dött tillstånd. Övergången åstadkoms genom att representera det döda tillståndet med en symbol, samt introduktionen av en regel för att gå från axiomet till denna symbol.

Lösningen för parametriska Lindenmayersystem resulterar i ett beteende mycket likt det för deterministiska system, men med en mer praktisk implementation, då parametrar kan användas för att hålla reda på antalet iterationer, istället för att en motsvarande symbol krävs för var iteration.

Stokastiska Lindenmayersystem är baserade på en slump som väljer mellan två eller fler alternativ vid var iteration. Den stokastiska lösningen för noddöd fungerar därmed genom att det vid var iteration finns en tio procent chans att noden dör. Författarna påpekar att denna lösning dock ej ger ett biologiskt korrekt resultat, med omväxlande döda och levande segment, och därmed även en grafiskt oönskad effekt.

En mer biologiskt motiverad lösning är kontextberoende Lindenmayersystem. Lösningen erbjuder ingen introduktion av döda segment i ett fullt levande träd, istället beskrivs en spridning av döda segment om minst ett sådant redan existerar. Spridningen sker i riktning från trädets rot, och motiveras biologiskt med att döda segment inte är kapabla till att transportera näring, segment av trädet separerade från roten via döda segment kommer därmed även de att dö.

Arbetet avslutas med ett exempel baserat på en kombination av de tidigare redovisade teknikerna. Exemplet redovisar uppbyggnaden av Lindenmayersystemet såväl som en grafisk representation av ett simulerat träd. Det noteras att inget av exemplen ger en helt biologiskt korrekt bild av växtdöd och att den grafiska representationen är grovt förenklad.

Baele och Warzee (2005) presenterar en metod för att hybridexekvera generation och rendering av träd baserade på Lindenmayersystemet med syfte att användas i realtidsapplikationer. Metoden använder CPU respektive GPU baserat på vilken enhet som är mest lämpad för var uppgift i systemet. Lösningen leder till att de två enheterna till stor del kan arbeta parallellt, vilket enligt författarna minskar den totala beräkningstiden gentemot seriell exekvering.

Arbetet täcker även ett antal biologiska koncept, framförallt angående förgreningsstruktur och grenkrökning. Det huvudsakliga fokuset för arbetet ligger dock på prestandaeffektivitet, och den biologiska delen av arbetet är därmed något förenklad. Arbetet går även in på strikt 3D-grafiska områden som detaljnivåer och ljussättning.

3 Problemformulering

3.1 Problem

Baele och Warzée (2005) presenterar tidsbaserade mätvärden på deras implementation av L-systemet, varpå de hävdar att denna typ a realtidsgenerering är lämplig för implementation i tyngre realtidsapplikationer som till exempel spel. Att förutspå teknikens prestandapåverkan vid faktisk tillämpning är dock inte helt trivialt eftersom resultaten inte sätts i perspektiv mot de tidseffektivitetskrav som finns på tyngre spel, där mycket mer än enbart grafiskt material behöver hanteras.

L-systemet som presenteras i det här arbetet är dock en teknik med syfte att representera träd i realtidsapplikationer, för vilket det redan finns en väl etablerad och dominerande teknik: statisk mesh. Då statisk mesh är en av de tekniker efter vilka realtidsapplikationers grafiska komplexitet måste anpassas efter idag, såväl som den teknik L-systemet skulle substituera, är det ett bra mål för en jämförelse med syfte att utvärdera substitutets prestandapåverkan i form av exekveringstid på CPU och GPU, såväl som minnesanvändning.

Målet med det här arbetet kan därmed summeras till att utvärdera utsträckningen för vilket L-systemsbaserade procedurellt genererade realtidsväxande träd kan användas som ett substitut för träd av motsvarande komplexitet i form av statisk mesh, med frågeställningen:

Hur förhåller sig L-systemsbaserade realtidsväxande träd till träd av motsvarande komplexitet representerade i form av statisk mesh, med avseende på prestandapåverkan i form av exekveringstid och minnesanvändning?

3.2 Metodbeskrivning

3.2.1 Resultat

Resultaten består av mätningar av exekveringstid såväl som minnesanvändning på CPU och GPU vid generering och rendering, respektive enbart rendering, för de två teknikerna. Resultaten används sedan för att få en bild över hur mycket mer krävande de realtidsgenererade versionerna av träden är jämfört med sina förgenererade motsvarigheter. Det blev även möjligt att utvärdera om realtidsträden har någon optimal komplexitet, såväl som vad dessa har för tidskomplexitet, det vill säga om dubbla komplexiteten på trädet tar dubbla tiden att behandla, eller om dubbla komplexiteten leder till mer eller mindre än dubbla tiden.

3.2.2 Testmiljö

Testerna har genomförts i en grundläggande renderingsmiljö baserad på OpenGL (SGI 1997-2016). Genom att spara de genererade träden som statiska modeller kan jämförelse mellan de två teknikerna ske med exakt samma trädstruktur.

Extern mjukvara som till exempel SDL (SDL Community 2016) användes för att hålla utvecklingstiden av artefakten inom en rimlig tidsram, introduktionen av extern mjukvara har dock minimeras eftersom prestandapåverkan av dessa kan vara problematisk att mäta och kontrollera.

3.2.3 Trädspecifikationer

Fokus har legat på trädstruktur och placering av lövverk, där segment i stammen representeras av primitiva cylindrar och löv av enkla polygoner. Eftersom träd varierar kraftigt mellan olika applikationer baserat på grafisk stil krävs det att systemet testas på träd av varierande komplexitet. SpeedTree (Interactive Data Visualization, 2015) är en väl etablerad mjukvara för att generera träd skapat av Interactive Data Visualization, som utöver att erbjuda mjukvara för att generera träd även erbjuder färdiga trädmodeller. Modeller producerade för användning i applikationer på persondatorer erbjuds i ett intervall där de består av mellan cirka 4000 och 30 000 trianglar. Baserat på SpeedTrees plats i spelbranschen görs därmed ett antagande om att detta är intervallet inom vilket majoriteten av spel placerar sin trädkomplexitet, och det är därmed intervallet som undersökts i det här arbetet.

Prusinkiewicz, Lindenmayer & Hanan (1988) såväl som Lam och King (2005) hävdar att formen för vedartad växtlighet huvudsakligen beror på spatiala förhållanden som tillgängligt utrymme och konkurrens mellan grenar från trädet själv såväl som andra träd. Lam och King (2005) beskriver och demonstrerar även ljus påverkan på träds struktur, framförallt genom dess tendens att växa i riktning mot ljuskällor, men påpekar även att ljuset i sig kan vara anledningen till att träd undviker att växa in i objekt, då det är mörkare närmare dessa. Metoderna de använder är dock allt för krävande för realtidsapplikationer.

För att praktiskt kunna använda växande träd i spel där nya träd befolkar världen krävs att gamla träd försvinner. Castellanos, Ramos och Ramos (2014) system för att representera växtdöd bygger på introduktionen av en symbol som representerar ett dött tillstånd. Mekaniskt är döden snarlik växtprocessen, och bör ha liknande prestandapåverkan.

3.2.4 Vetenskaplig metod

Tester har skett som ett experiment i en grundläggande kontrollerad miljö. Att utföra en fallstudie där systemet testas i ett fullt spel har potential att ge en bättre bild över dess möjlighet till integration med krävande realtidsapplikationer, men introducerar även ett antal problem. Först och främst riskeras resultatet bli svårtolkat eftersom det gäller för en specifik applikation som inte nödvändigtvis speglar de förhållanden som gäller för majoriteten av producerade spel. För det andra försvårar det framställningen av tillförlitliga mätvärden eftersom mekaniker i spelet orelaterade till trädgenereringen sannolikt kommer påverka dess exekveringstider genom oförutsägbar belastning av hårdvaran. Slutligen kräver en fallstudie antingen skapandet av ett spel, eller introduktionen av trädsystemet till ett existerande spel, vilket i båda fall sannolikt skulle resulterat i att arbetet expanderat utanför den tillgängliga tidsramen.

3.2.5 Hypotes

Hypotesen var att den statiska representationen av träd skulle vara effektivare, både gällande exekveringstid och minnesanvändning. Procedurellt genererat material har potential att använda mycket lite minne om det produceras inför var rendering, det är dock här inte fallet. Information angående trädets struktur lagras mellan renderingar för att minska antalet beräkningar som behöver repeteras, den L-systemsbaserade lösningen förväntades därmed kräva mer minne än den statiska. Skillnaden mellan de två teknikerna förväntades vara mindre för små träd med låg komplexitet, men när träden blev större och mer komplexa förväntades L-systemsrepresentationen, senare refererad till som den dynamiska representationen, kunna konkurrera allt sämre.

4 Genomförande

4.1 Förstudie

4.1.1 Inspiration

Den huvudsakliga inspirationskällan till arbetet kommer från ett examensarbete av Stefan Enberg (2007) kallat: *”Kontextberoende generering av träd för dataspel”*. Arbetet handlar om utforskning av möjligheter till att använda L-system för att generera visuellt tilltalande och trovärdiga träd som beror på sin omgivning. Träden påverkas av aspekter som vilken riktning ljus kommer ifrån, vind, och gravitation.

Enbergs L-system lagrades i form av datastrukturen sträng, för vilket han konstaterar att ett stort antal konversioner mellan strängar och numeriska värden krävdes under generering och påpekar att högre effektivitet kan uppnås med en alternativ datastruktur, vilket är en bidragande faktor till uppkomsten av den datastruktur som använts för det här arbetet.

Baele och Warzees (2005) uttalande om att CPU är effektivare för att generera och tolka L-system än GPU ligger som grund för metoden som använts i det här arbetet. Deras lösning utför däremot geometrigenerering på GPU, vilket för det här arbetet sker på CPU baserat på ett antagande om att ett växande träd resulterar i mer information än Baele och Warzees träd. Det antas därmed vara enklare att överföra grafisk information till GPU än att generera den på GPU.

Implementationen av L-systemet i sig baseras huvudsakligen på definitionerna för L-system. Implementationsdetaljer och krav skiljer sig mycket från L-system till L-system baserat på vad implementationen behöver vara kapabel till. För ett deterministiskt kontextfritt L-system kan till exempel substitutioner ske i en enda datastruktur, till exempel en list. För ett kontextberoende system skulle detta resultera i att kontexten förändras under seriell behandling, därmed behövs två datastrukturer: en som innehåller föregångaren och en som ändringar kan utföras i.

Operationerna som krävs för att generera ett L-system är generellt inte komplexa och majoriteten består av att jämförelser av parametrar och symboler samt addition och multiplikation av parametrar, men de är många. En av de huvudsakliga uppgifterna är därmed att använda rätt datastrukturer och operationer för rätt uppgift. Ett av enklaste sätten att åstadkomma detta är genom att referera till dokumentationen för programspråket och de bibliotek som används. För det här arbetet inkluderar det dokumentationen för OpenGL (SGI 1997-2016) och två separata dokumentationer av C++ (Microsoft 2016; cplusplus 2016).

4.2 Iterativ process

Låga prestandakrav och hög funktionalitet är två koncept som ofta drar i olika riktningar. För utvecklingen av artefakten har därför en iterativ process valts, med syfte att finna det enklaste och minst prestandakrävande L-systemet kapabelt till att producera träd av acceptabel kvalité.

4.2.1 Deterministiskt kontextfritt L-system med hakparenteser

Den fösta implementationen av L-systemet var så enkelt som möjligt, och innehöll därför endast en produktionsregel per symbol, utan kontext. I ett sådant system kan hakparenteser användas utan explicit hantering, och blir relevanta först när en grafisk tolkning a systemet ska ske, vilket implementeras i ett senare steg.

L-systemets natur innebär in stor mängd insättningar av symboler över hela den lagrande datastrukturen såväl som iteration över intervallet, men kräver få oordnade indexeringar. Datastrukturen list erbjuder mer eller mindre precis denna funktionalitet, med konstant tid för insättning över hela strukturens intervall. En av de utmärkande egenskaperna för L-system är dock dess parallellism, vilket innebär att alla substitutioner sker samtidigt. Tekniska begränsningar gör det opraktiskt att faktiskt utföra alla substitutioner samtidigt, en mer praktisk lösning är att utföra dem seriellt på ett sätt som ger samma resultat som en parallell substitution. Att utföra alla substitutioner i en och samma lagringsentitet påverkar resultatet eftersom det innebär att kontexten förändras vid var substitution. Genom att placera efterträdare i en ny lagringsenhet för att sedan ersätta originalet med den nya lagringsenheten undviks introduktionen av inkorrekt kontext. Metoden leder även till att alla insättningar i den nya lagringsenheten kan ske i slutet, vilket tillåter effektiv användning av lagringsenheter som ligger linjärt i minnet, och därmed inte kräver lika mycket minne som en länkad struktur. I det här fallet användes en vector. Tester på L-systemet visade att för alla testade träd reducerades genereringstiden med ungefär 30 procent när en vector användes istället för en list, där i båda fall alla insättningar skedde i slutet. Symbolerna representeras med datastrukturen *char*.

Produktionsregler lagras i en *unorderd multimap* med föregångaren som nyckel och efterträdaren som värde. *Unorderd multimap* använder hashindexering för snabbt uppslag av nycklar och tillåter både flera identiska och unika element med samma nyckel.

4.2.2 Stokastiskt kontextfritt L-system med hakparenteser

Som tidigare nämnt innebär ett stokastiskt system att flera produktionsregler kan vara applicerbara på samma föregångare, i vilket fall slump används för att välja mellan dem. *Unorderd multimap* tillåter uppslag av alla värden associerade med en nyckel, vilket i det här fallet innebär alla produktionsregler associerade med en viss symbol. Selektion sker via slumptalsgenerering av index i intervallet för giltiga produktionsregler. Lösningen är naiv och resulterar i att alla produktionsregler har samma sannolikhet att inträffa, *unorderd multimap* tillåter dock att flera identiska produktionsregler matas in för att på så sätt modifiera sannolikheten. Fördelen med denna naiva lösning är snabbare exekvering eftersom det inte behövs några sannolikhetsberäkningar, nackdelarna är minskad kontroll och ökad minnesanvändning.

Elementen i *unorderd multimap* är som namnet antyder inte sorterade enligt någon speciell ordning, men en konstant ordning bibehålls mellan iterationer av träd såväl som programstarter, förutsatt att produktionsreglerna matas in i samma ordning. Identiska träd kan därmed genereras flera gånger så länge samma *seed* (eng. frö) används i slumptalsgenerator för att indexera produktionsreglerna.

4.2.3 Sköldpaddsgrafik

L-systemet är vid det här laget kapabelt att generera grundläggande grenstrukturer med fasta vinklar, vilket gör det till ett lämpligt tillfälle att implementera den grafiska tolkningen

av systemet. Tolkningen sker via iteration av char-listan med en switch-case-sats för att identifiera symboler och förmedla vad de motsvarar till OpenGL (SGI 1997-2016).

Sköldpaddan, som tillståndet för renderingen kallas, delar många likheter med OpenGLs egna tillstånd. Av rent praktiska skäl, såväl som av effektivitetsskäl, har därför OpenGLs tillstånd använts som sköldpadda.

4.2.4 Stokastiskt parametriskt kontextfritt L-system med hakparenteser

För att skapa träd som växer mjukt krävs parametrar som tillåter segment av trädet att bli större, på så sätt kan nya segment växa fram istället för att dyka upp fullvuxna när L-systemet blir längre. Antalet parametrar är begränsat till en parameter per symbol för att förenkla systemet och tillåta snabbare exekvering. En egen klass kallad TreeSegment innehållande symbolen och en parameter har skapats för att ersätta de chars som använts för att tidigare lagra symbolen, likt de chars som ersätts lagras den nya TreeSegment i en list.

Produktionsreglerna har fått en liknande omarbetning där en ny klass kallad ProductionRule skapats för att lagra eventuella efterträdare, i form av TreeSegment, såväl som för att beskriva regelns modifikation av parametervärden och eventuella kriterier som dessa ska uppfylla. Ett additionsvärde och ett multiplikationsvärde används för att modifiera parametervärden, dessa appliceras vid var iteration. Om parametervärdet inte önskas förändras sätts additionsvärdet till noll och multiplikationsvärdet till ett. Förväntan är att det i de flesta fall kommer ske en förändring av parametern och att en kontroll för att urskilja de få tillfällen då det inte händer skulle vara mer krävande än att göra ett litet antal extra beräkningar, speciellt eftersom det handlar om mycket enkla beräkningar.

Det är här värt att påpeka att det här L-systemet endast beskriver topologin hos trädet, vilket betyder att vinklarna för grenar och böjar i trädstrukturen inte direkt är ett resultat av traditionella L-systemsmekanismer utan tillkommer som en del av tolkning via sköldpaddsgrafik. I användarvänlighetssyfte hanteras tilldelning av vinklar som en del av L-systemsimplementationen.

Bild 1 visar ett exempel på träd genererat med den här typen av L-system, notera att trädet är växande. Som synligt i bilden har segment inget sätt att avgöra hur tjockt segmentet det är fäst vid är, vilket resulterar i att grenar växer till en övre gräns. Det är därmed problematiskt att skapa avsmalnade grenar. Grenen längst till vänster är avsmalnande på grund av att den vid tillfället då bilden togs inte vuxit till sin fulla storlek. På bilden syns inte heller hela trädet. Hur trädet växer och när det ska sluta växa avgörs helt och hållet av hur stor chans var produktionsregel har att väljas, som ett resultat är det mycket svårt att kontrollera storleken på träd. Stammen har i det här fallet vuxit väldigt lång och utanför programmets fasta vy.

Bild 1 Exempel på träd genererat med stokastiskt parametriskt kontextfritt L-system med hakparenteser

4.2.5 Stokastiskt parametriskt kontextberoende L-system med hakparenteser

L-systemet som har implementerats så här långt är kapabelt att generera växande grenstrukturer, praktisk användning av algoritmen visar dock på ett antal begränsningar. Först och främst innebär avsaknaden av kontext att grensegment växer oberoende av varandra upp till en förutbestämd övre gräns, att skapa träd med avsmalnande grenar kräver därmed komplexa och oflexibla produktionsregler.

Avsaknaden av kontext innebär också att det är problematiskt att avgöra trädets storlek, vilket orsakar problem när det gäller att kontrollera trädets växthastighet, mer specifikt handlar det om hur ofta trädet förgrenar sig. Då förgreningar och beslutet om en gren ska fortsätta eller sluta växa är beroende på statistisk sannolikhet innebär en låg sannolikhet för växt att trädet sällan skapar förgreningar och slutar växa efter få iterationer, vilket innebär små enkla träd. En hög växtsannolikhet innebär att trädet bildar långa grenar och många förgreningar. I värsta fall slutar trädet aldrig växa eftersom det för varje gren som slutar växa skapas ett större antal nya undergrenar, resultatet är exponentiell tillväxt som snabbt överbelastar systemet.

Genom att implementera kontextberoende åt vänster, det vill säga mot roten av trädet, kan tjockleken på segment anpassas efter tjockleken hos segmentet som det är fäst vid, därmed kan avsmalnande strukturer skapas. På samma sätt kan ett vänsterriktat kontext användas för att introducera nya produktionsregler när trädet uppnått en viss storlek, för att till exempel göra det mer sannolikt för en lång gren att sluta växa än en kort gren, eller för att garantera att ett träd inte kan sluta växa innan det uppnått en viss storlek.

Kontext har implementerats med en vector innehållande en ny klass kallad TreeContext. TreeContext innehåller information om kontextkrav på symbol och parametervärde, såväl som information om parametrar ska jämföras med varandra eller mot ett fast värde.

Introduktionen av vänsterkontext visade en klar förbättring i kontroll över trädets struktur såväl som tillväxt. Tester visade dock att det uppstod situationer där korta grenar växte mycket tjocka, vilket resulterade i klumpar på trädet. Även ett högerkontext introducerades för att lösa problemet, då ett sådant kan användas för att hindra ett segment från att växa över en viss storlek om det inte finns ytterligare ett följande segment. Med andra ord innebär detta att en gren inte kan växa tjock utan att också växa lång.

Notationen med hakparenteser för att representera grenar betyder att en genstruktur representeras via ett linjärt medium. Två symboler som ligger bredvid varandra i listan kan därmed befinna sig på helt olika ställen i trädet. Kontexten måste därmed ta *[* och *]* i åtanke. För vänsterkontexten ökar varje *]* en räknare och *[* minskar den. En loop itererar genom symbolerna tills räknaren står på noll igen och den följande symbolen inte är en *]*. Högerkontexten fungerar praktiskt taget identiskt, med undantag för att de två hakparenteserna är omvända, eftersom vilken parentes som är öppnande och stängande är beroende på iterationsriktningen.

Det kan även finnas situationer där vissa symboler inte önskas vara en del av kontextkontrollen. En produktionsregel som ser till att ett segment i en gren inte växer tjockare än segmentet innan är till exempel inte intresserad av att veta att det finns en symbol som representerar en böj i grenen mellan de två segmenten. Var produktionsregel innehåller därmed en vector med symboler som skall ignoreras vid kontextsökning. Kontrollen sker i samma loop som kontextkontrollen för hakparenteser och fungerar enligt en liknande princip där den aktuella symbolen itereras tills en symbol som inte finns med i vectorn hittas. Bild 2 visar ett exempel på träd som kan genererar med det här L-systemet.

Bild 2 Exempel på träd genererat med stokastiskt parametriskt kontextberoende L-system med hakparenteser

Nedan visas pseudokod för att hitta relevant kontext:

```
while (symbol = opening bracket or open brackets > 0 or symbol found in context
exeptions)
   if (symbol = opening bracket)
      open brackets + 1
   else if(symbol = closing bracket)
      open brackets - 1
   end if
   symbol = next symbol
end while
```

Pseudokod för att avgöra om produktionsregler är applicerbara:

```
if(isEnforceingRequirement)
   if(parameter requrement is met)
      while(more symbols in string and more symbols in context)
         find relevant context
         if(symbol = context)
            if(lower requirement < context parameter < upper requirement)
               if(compare parameters)
                  if(parameter is supposed to be greater than context parameter)
                     if(parameter < kontext parameter)
                        context is not met
                     end if
                  else
                     if(parameter > context parameter)
                        context is not met
                     end if
                  end if
               end if
            else
               context is not met
            enf if
         else
            context is not met
         end if
      end while
   end if
end if
context is met
```

4.2.6 Statisk representation med geometrisk data

Den ursprungliga implementationen av statisk representation av träd baserades på lagring av geometriska former med en gemensam basklass innehållande en renderingsfunktion som anropade lämpliga OpenGL-funktioner (SGI 1997-2016). Formerna innehöll även färginformation, geometrisk information, och en matris beskrivande position och rotation hos formen. Lösningen visade sig dock vara allt för långsam för att vara en representativ implementation av statisk modell för användning i realtidsapplikationer eftersom den skickade stora mängder information till GPU vid var bilduppdatering. Lösningen lagrade geometri som former ett antal former: cylindrar, sfärer, och plan. Därmed åstadkoms en lägre minnesanvändning, men det krävdes en ökad mängd beräkningar vid var bilduppdatering för att generera geometri utifrån dessa former.

4.2.7 Statisk representation med display list

En andra lösning på statisk representation av träd implementerades via användningen av *display list* (SGI 1997-2016 kap. 5.4). *Display list* fungerar genom att lagra en serie förkompilerade OpenGL-kommandon (SGI 1997-2016) i grafikminne, serien kan sedan kalls på flera gånger utan att behöva överföras igen. Denna lösningen resulterade i betydligt snabbare renderingstider. Processen för att konvertera det genererade trädet till en dynamisk modell blev också betydligt snabbare, en trevlig sidoeffekt, även om det inte är en del av frågeställningen för arbetet. Lösningen har endast en nackdel jämfört med den tidigare tekniken, den använder betydligt mer minne då alla vertiser lagras direkt i grafikminne.

4.3 Programbibliotek

4.3.1 SDL

SDL (SDL Community 2016) är ett *open-source* programbibliotek som förenklar och abstraherar användning av lågnivåfunktioner hos till exempel grafikhårdvara och inputenheter över en rad plattformar. För det här projektet används SDL för att förenkla skapandet av ett grafiskt fönster såväl som för att skapa ett OpenGL-kontext (SGI 1997-2016).

4.3.2 GLM

GLM (G-Truc Creation 2016) är ett headerbibliotek designat för användning med OpenGL (SGI 1997-2016) och erbjuder en stor mängd matematiska funktioner, huvudsakligen gällande matris och vektor-operationer.

4.4 Pilotstudie

En pilotstudie har utförts där tester har utförts på ett godtyckligt träd. Trädet är i övre skiktet av komplexitetsintervallet inom vilket systemet skapar träd med nuvarande konfiguration. Testet har som syfte att skapa ett medelvärde på tiden det tar att växa den generation där det dynamiska trädet når samma generation som det statiska trädet skapats från, för att minimera effekten av oförväntad belastning av CPU och GPU från bakgrundsprocesser på datorn har testet utförts 1000 gånger. Generationen för det statiska trädet är nummer 10000, vilket innebär att mätningen har skett när det dynamiska trädet gått från generation 9999 till 10000. Antalet generationer är väldigt mycket högre än de skulle vara vid faktisk användning, normal sett nås samma växtstorlek runt generation 600.

För att utföra mätningar av tiden på grafikkortet behöver dock antalet bilduppdateringar vara olåst och växthastigheten har anpassats därefter. Vid 60 bilder per sekund, vilket är en vanlig bilduppdateringshastighet för spel, tar var bildruta 16 millisekunder att beräkna, under testet tar en bildruta mellan 0.1 och 3 miliskunder, beroende på komplexiteten hos trädet som genereras. Nedan visas utdata från programmet:

Dynamic average total time in ms: 6.37304	Static average total time in ms: 1.25207
Dynamic average grow time in ms: 0.193643	
Dynamic average render time in ms: 6.1794	
Dynamic tree size in bytes: 23544	Static tree size in bytes: 106380
Number of vertices: 24160	Number of vertices: 24160
L-system length: 981	

Testet visar att det dynamiska trädet likt förväntat är betydligt långsammare. En del av anledningen bör vara att medans de statiska träden enbart behöver innehålla synliga segment och deras position, innehåller de dynamiska ett antal segment som representerar saker som inte är direkt är synliga, men som påverkar trädets struktur, till exempel böjar i stammen.

Något överraskande är att rendering för båda tekniker tar avsevärt längre tid än generering av det dynamiska trädet. Det beror sannolikt på att även om trädet består av enklare former som sfärer och cylindrar är antalet vertiser hos trädet ganska högt. Till skillnad mot vad som beskrevs som väntat resultat i hypotesen kräver den statiska lösningen betydligt mer minne, vilket har att göra med sättet som de två teknikerna beskriver geometri. De statiska träden lagrar geometrin i form av vertiser medans de dynamiska träden beskriver den som former, de cylindrar som används för att representera stammen kan till exempel beskrivas med två punkter och diametrar. I fallet för de dynamiska träden beror dessutom dessa fyra parametrar på en enda variabel, vilket vidare reducerar mängden data som behöver lagras. Detta innebär att en betydligt mindre mängd information behöver lagras, men också att fler beräkningar behöver utföras vid rendering för att generera geometri från formerna. Bild 3 visar trädet som generats för studien, vid generation 10 000.

Bild 3 Träd genererat i pilotstudie

4.5 Användning av artefakt

Vid start av artefakten instrueras användaren skriva in i ett konsolfönster hur många gånger testet önskas genomföras. Runt tusen tester rekommenderas då medel och medianvärden baserade på färre värden än detta har en tendens att variera mellan testomgångar. Programmet kommer sedan fråga efter ett *seed (eng. frö)* för genereringen, antalet segment var cylinder i trädet skall bestå av, och hur många tester som skall utföras. Under testningen kan två exemplar av det genererade trädet synas i huvudfönstret. Notera att ingen växt av trädet är synlig under testning. Detta beror på att trädet renderas efter att det växt var bildruta, för att sedan återställas en generation för att växa igen. Trädet växer därmed vid var bilduppdatering till samma tillstånd som det hade i slutet av förra bilduppdatering. När testet är klart skrivs resultaten ut i konsolfönstret. Programmet kan stängas genom att mata in valfritt tecken i konsolfönstret efter att testet är klart. Träden som genereras beror på slump och produktionsreglerna ger varierande resultat, träden kan därmed skilja mycket i storlek och komplexitet mellan olika *seeds (eng. frö)*.

5 Utvärdering

5.1 Undersökning

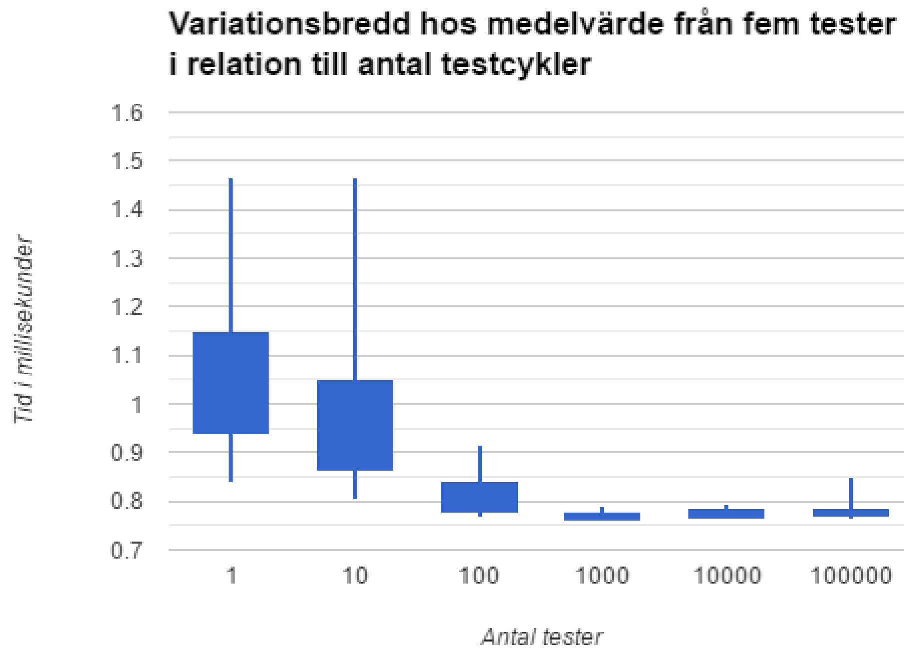
Alla tester har utförts på en persondator, utöver testprogrammet exekverar även ett operativsystem såväl som ett antal drivrutiner på enheten. Dessa kan orsaka oförutsägbar belastning på diverse beräkningsenheter, minnen, och bussar, vilket kan leda till inkorrekta mätvärden. För att minimera påverkan av dessa belastningar baseras utvärderingen på medel samt medianvärde från en serie repeterade tester. Den första delen av undersökningen går ut på att undersöka hur många tester som behöver utföras för att ge pålitliga värden. Detta har gjorts genom att utföra ett experiment där medelvärden togs fem gånger var för: ett, tio, hundra, tusen, tiotusen, och hundratusen tester på ett och samma träd. Spridningen av resultaten kunde sedan användas för att avgöra hur många tester som behövs för att ge konsekventa resultat.

Undersökningen fokuserar på tre huvudsakliga aspekter: genereringstid, renderingstid, och minnesanvändning. Intressant för undersökningen är att utvärdera hur de respektive teknikerna beror på L-systemslängd och antal vertiser. För att utföra en rättvis jämförelse mellan de två teknikerna används en representation av samma träd i båda fall, vilket innebär att en tolkning av det dynamiska trädet har skett för att skapa det statiska, därmed är det också relevant för undersökningen att avgöra närvaron av eventuella artefakter som uppstått som ett resultat av denna tolkning.

5.2 Resultat och analys

Nedan redovisas och analyseras resultat från tester av genereringstid för en generation av L-system, renderingstid av dynamiska och statiska träd, samt minnesanvändning för de båda teknikerna. Generering av L-system sker på CPU och mätning av tid sker genom att mäta skillnaden i tid mellan en tidpunkt innan generering och en efter. OpenGL (SGI 1997-2016) version 2.1 tillåter inte direkt tillgång till mätning av tid på GPU, båda tidpunkter tas därmed på CPU. Den första tidtagningen sker precis innan det första renderingskommandot sker, den andra tidtagningen sker efter att OpenGL returnerat att alla instruktioner exekverat genom användandet av en funktion kallad glFinish. Minnesanvändning beräknas genom att multiplicera antalet objekt av var typ med dess storlek, som fås från C++ sizeof-operator.

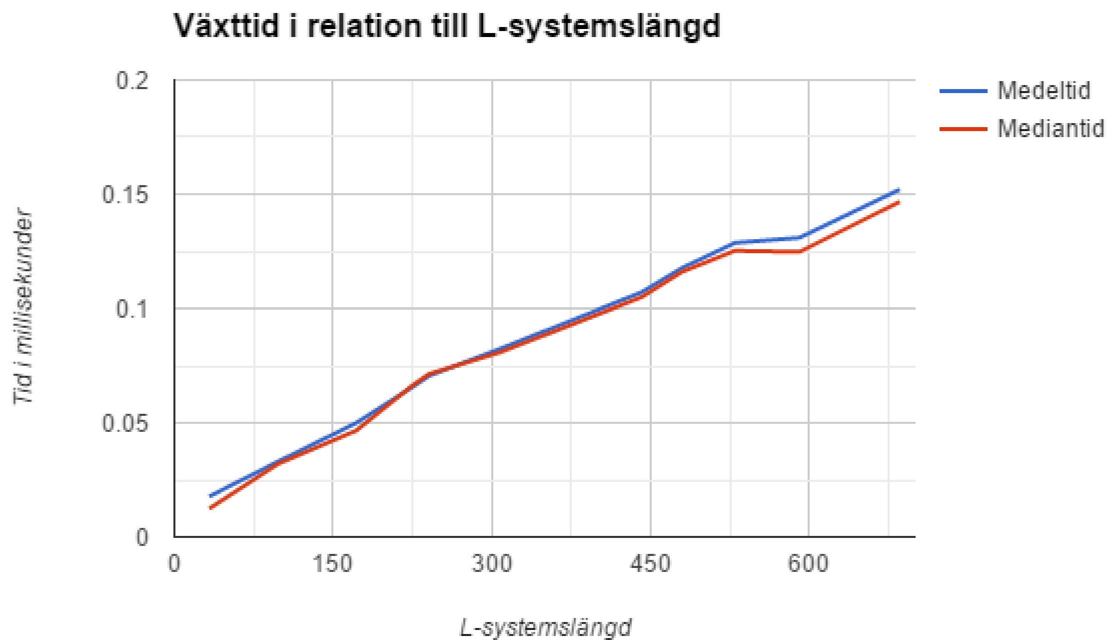
5.2.1 Antal tester



Figur 3 Variationsbredd hos medelvärde från fem tester i relation till antal testcykler

Figur 3 visar på spridningen av resultat för de respektive testerna, där linjen motsvarar det högsta respektive lägsta värdet, och den ifyllda lådan det nästa högsta respektive lägsta värdet. tusen tester, som oväntat nog presterade snarlikt tiotusen tester och bättre än hundratusen tester, ansågs ge goda resultat. Så få tester som möjligt är dessutom att föredra eftersom tiden det tar att utföra testen är direkt beroende av hur många tester som utförs.

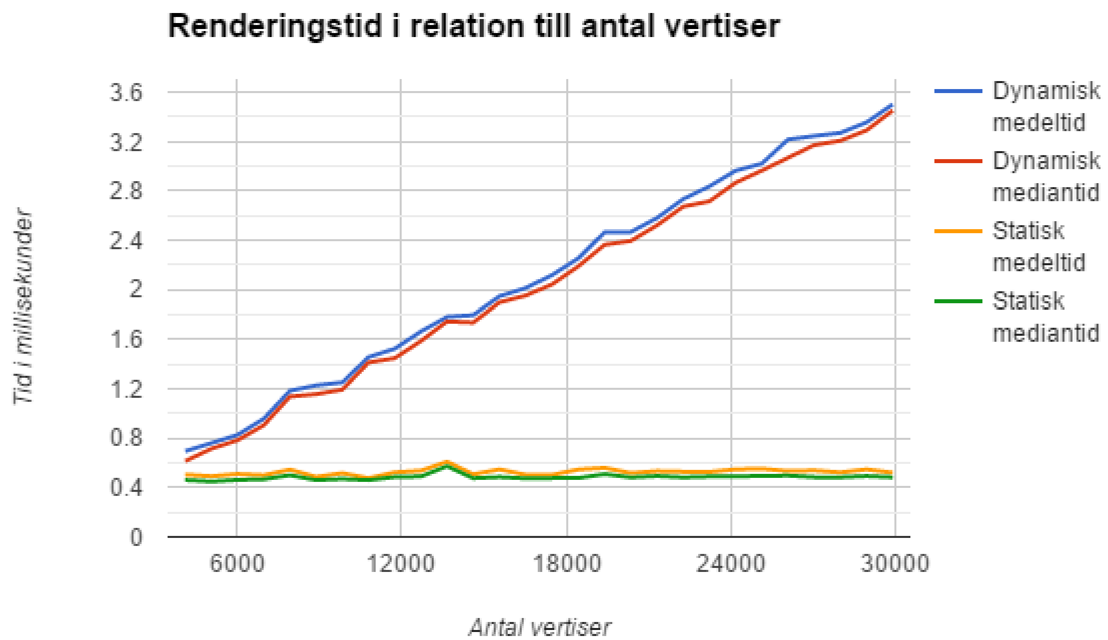
5.2.2 Växttid



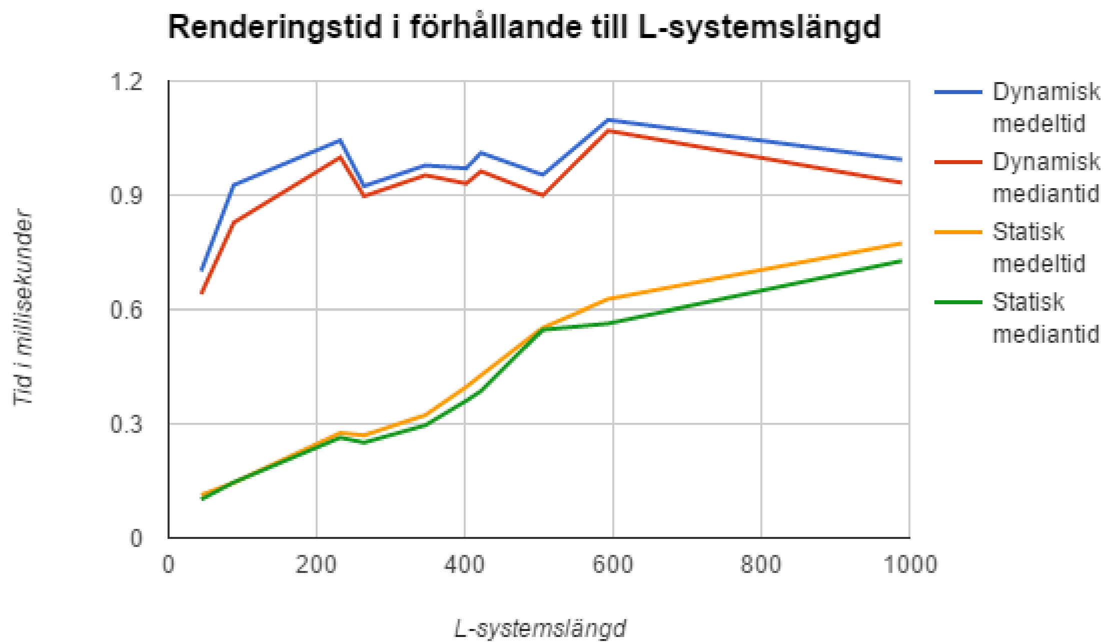
Figur 4 Växttid i relation till L-systemslängd

Växttiden, det vill säga tiden det tar att iterera en generation av L-systemet, är oberoende av trädets geometriska detaljnivå och beror istället på L-systemets längd och komplexiteten hos dess produktionsregler. I Figur 4 visas växttidens relation till L-systemets längd.

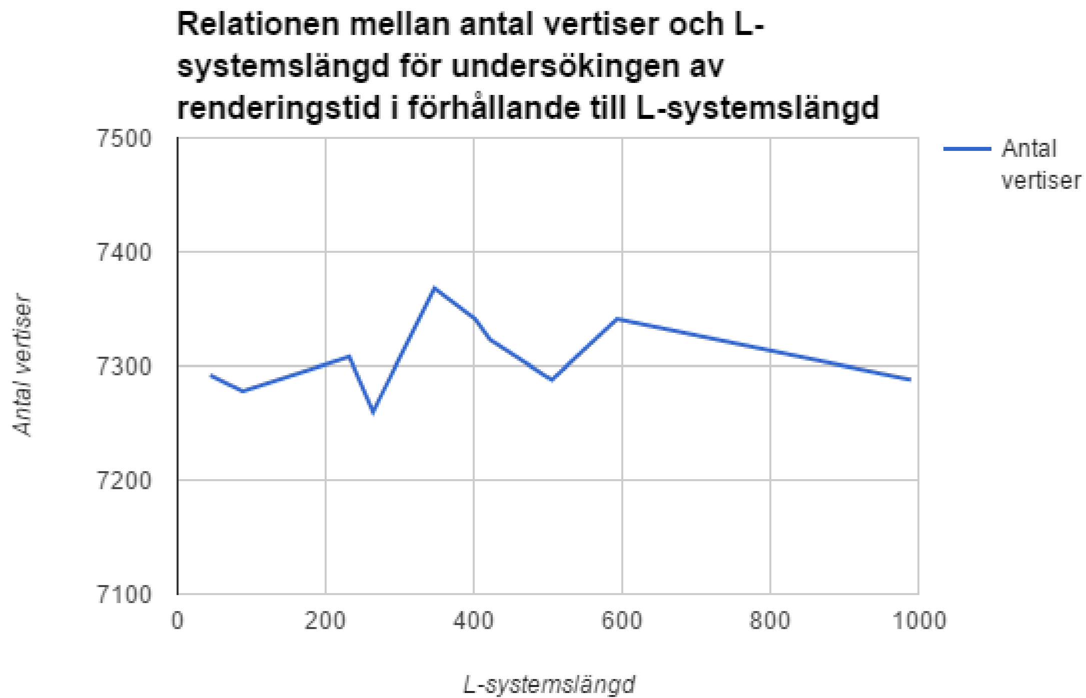
5.2.3 Renderingstid



Figur 5 Renderingstid i relation till antal vertiser



Figur 6 Renderingstid i förhållande till L-systemslängd



Figur 7 Relationen mellan antal vertiser och L-systemslängd för undersökingen av renderingstid i förhållande till L-systemslängd

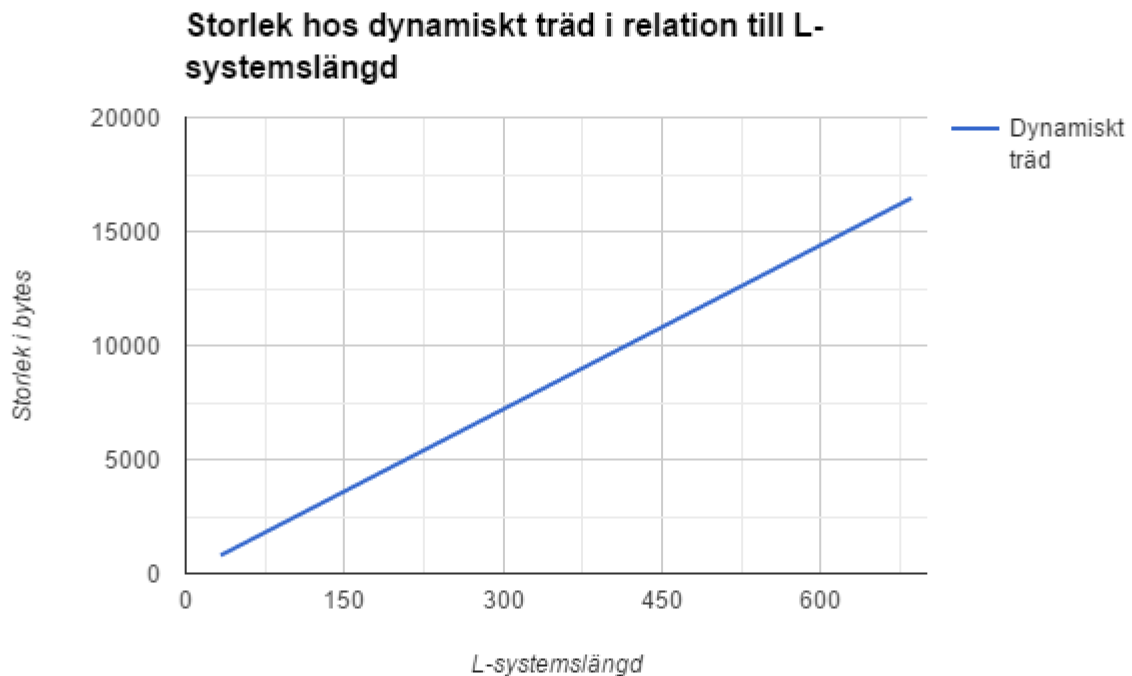
Den dynamiska lösningens renderingstid beror på två huvudsakliga faktorer: antalet vertiser och L-systemets längd. Figur 5 visar på relationen mellan den dynamiska och den statiska implementationen och antalet vertiser för ett L-system bestående av 506 symboler renderat med olika antal vertiser. Den dynamiska lösningen skalar betydligt sämre med antal vertiser än den statiska, då den dynamiska behöver både beräkna och överföra vertispunkter till grafikenheten innan dessa kan renderas. Grafen är något bedragande i att det ser ut som att den statiska lösningen har en konstant renderingstid, detta är inte fallet. Även den statiska renderingstiden ökar med antalet vertiser, ökningen är dock liten vilket innebär att den överskuggas av den mycket större ökningen hos den dynamiska lösningen samt döljs av brus orsakat av externa processer.

Figur 6 visar förhållandet mellan renderingstid och L-systemslängd. L-systemets procedurella natur gör det svårkontrollerat, att generera unika träd med varierande L-systemslängd trots samma antal vertiser har därför varit problematiskt. Som ett resultat har de träd som använts i Figur 6 en skillnad på upp till cirka 0.7% från medelantalet för vertiser och en skillnad på upp till cirka 1.5% mellan det största och minsta antalet vertiser. Relationen mellan L-systemslängden och antalet vertiser redovisas i Figur 7, som uppvisar beteende liknande det hos de två kurvorna för dynamisk renderingstid i Figur 6, och därmed visar på dess påverkan på testresultatet.

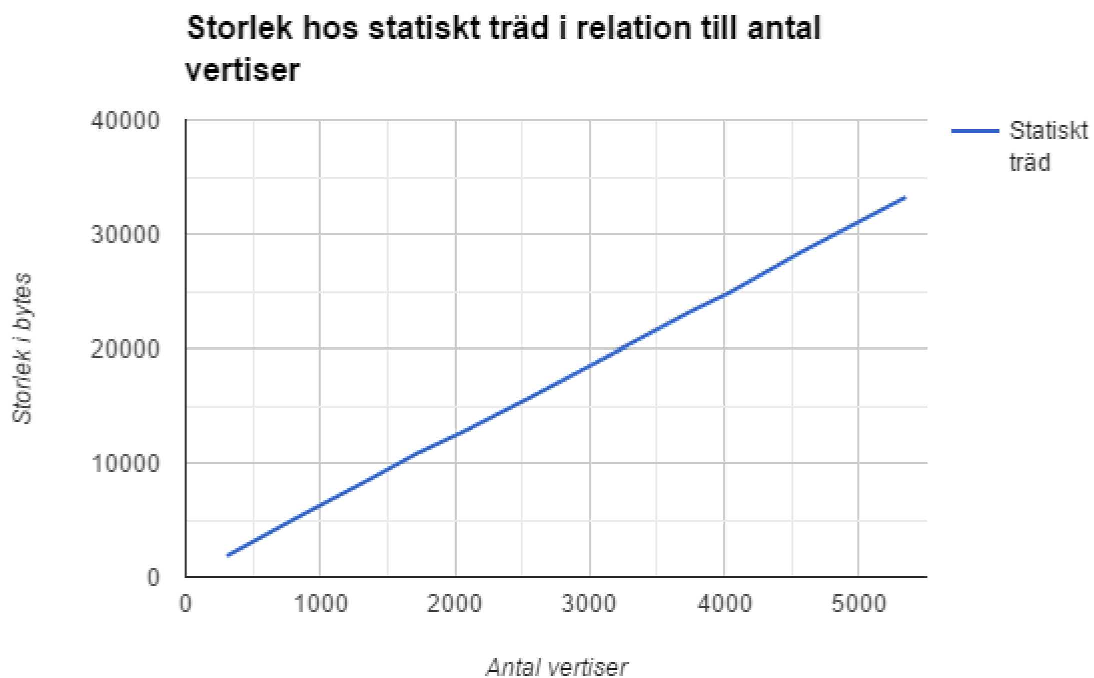
Resultaten visar övrigt inte på någon stark trend hos resultaten för den dynamiska renderingen. För den statiska renderingen syns en tydlig ökning i renderingstid med längre L-system. Detta är sannolikt ett resultat av användandet av tekniken *display list* (SGI 1997-2016 kap. 5.4) som används för att rendera de statiska träden. *Display lists* lagrar en serie förkompilerade OpenGL-kommandon (SGI 1997-2016) i grafikminne, för att ge ett identiskt träd som för den dynamiska tekniken lagras exakt samma kommandon som används för att

rendera detta. För det dynamiska trädet renderas var segment, i det här fallet synonymt med symbol, som en serie vertiser skiljt från andra segment av trädet, vilket bär över till det statiska representationen. För den dynamiska representationen verkar denna separation inte ha en speciellt tydlig påverkan, istället är antalet vertiser den huvudsakliga faktorn i hur lång tid rendering tar. Den statiska lösningen däremot beter sig precis tvärtemot, med liten påverkan från antalet vertiser men med en tydlig koppling mellan L-systemslängd och renderingstid.

5.2.4 Minnesanvändning



Figur 8 Storlek hos dynamiskt träd i relation till L-systemslängd



Figur 9 Storlek hos statiskt träd i relation till antal vertiser

Figur 8 visar den dynamiska implementationens minnesanvändning i förhållande till L-systemslängd. Då implementationen beskriver geometri som längder och radier för vilka vertiser beräknas under rendering är minnesanvändningen orelaterad till detaljnivån i vilket trädet renderas. Minnesanvändningen beror därmed endast på längden hos L-systemet, för vilket Figur 8 visar på ett linjärt förhållande.

Den statiska representationens baseras på att lagra geometri som vertiser. Minnesanvändningen är därmed direkt beroende av antalet vertiser, vars förhållande kan ses i Figur 9. Även här tyder grafen på ett linjärt förhållande. Inte representerat i graferna är skillnaden i minnesanvändning mellan de två teknikerna. Den dynamiska lösningens lagring av geometri innebär en mycket låg användning av minne, och lagring av ett trädsegment, här synonymt med L-systemssymbol, kräver 21 bytes, oavsett hur många vertiser som används för att rendera segmentet. Ett segment består av två geometriska komponenter: en cylinder och en sfär. Motsvarande segment representerat i ett statiskt format med fem horisontella och ett vertikalt segment för cylindern respektive fem horisontella och fem vertikala segment för sfären, vilket är den minsta mängden vertiser som krävs för att skapa runda objekt snarare än rätblock, resulterar i 420 bytes information.

5.3 Slutsatser

Syftet med undersökningen har varit att undersöka prestandakraven hos procedurella realtidsväxande träd baserade på L-systemet genom att jämföra dessa med motsvarande träd representerade i form av statisk mesh. Förväntan var aldrig att de procedurella träden skulle prestera bättre eller lika med de statiska, undersökningens syfte har snarare varit att undersöka hur mycket mer krävande dessa är.

Genereringstiden för att växa det dynamiska trädet en generation beror på L-systemslängden och är likt renderingstiden för samma träd, som beror på antal vertiser, linjär.

Genereringstiden är dock generellt betydligt kortare än renderingstiden. Renderingstiden för dynamiska träd skalar kraftigt med antal vertiser och presterar alltid betydligt sämre än rendering av sin statiska motsvarighet, men skillnaden blir mindre när färre vertiser renderas.

Resultaten skiljer sig något från hypotesen som beskrivs i kapitel 3.2.5. Förväntan var att de dynamiska träden skulle ta upp mer minne på grund av L-systemsinformationen som lagrades, vilket visade sig inte var fallet. I övrigt stämde hypotesen ganska väl med längre renderingstider för den dynamiska lösningen såväl som större skillnad mellan de två teknikerna vid mer komplexa träd.

Metoden som används för att skapa statisk representation av träd från L-system resulterar i att träden för de båda teknikerna blir identiska, men även i att segmentering av vertiser från den linjära tolkningen och renderingen av L-system bär över till den statiska representationen. Renderingstiden för de statiska träden visar ett tydligt beroende på L-systemslängden, vilket sannolikt beror på segmenteringen. Det är därför troligt att den statiska representationen skulle kunna prestera betydligt bättre om segmentering kunde undvikas vid dess skapande, och därmed öka prestandaskillnaden mellan de två teknikerna. Den dynamiska lösningen erbjuder dock lägre minnesanvändning än den statiska, men till priset av att fler beräkningar sker vid rendering. Minnesanvändningen för den dynamiska lösningen är dessutom oberoende av antalet vertiser som trädet består av vid rendering.

6 Avslutande diskussion

6.1 Sammanfattning

Arbetet har haft som mål att utvärdera prestandakraven hos realtidsväxande procedurella träd baserade på L-systemet genom att jämföra dessa med motsvarande träd i ett statiskt format. Ett stokastiskt parametriskt kontextberoende L-system med hakparenteser samt en statiskt representation baserad på *display list* (SGI 1997-2016 kap. 5.4) har implementerats i en testmiljö baserad på OpenGL (SGI 1997-2016).

En undersökning med syfte att mäta genereringstid vid trädväxt, renderingstid, och minnesanvändning respektive renderingstid och minnesanvändning för de båda teknikerna har utförts. Likt förväntat visar undersökningen på att den statiska representationen erbjuder betydligt snabbare rendering, men att skillnaden mellan de två teknikerna är mindre för träd bestående av färre vertiser. Genereringstiden för de växande träden är relativt liten jämfört med renderingstiden då tolkningen från L-system till geometri kräver ett stort antal vertisberäkningar. Som ett resultat av att vertiser beräknas vid rendering kräver de växande träden betydligt mindre minne än de statiskt representerade träden.

Resultaten visar att de växande träden är betydligt mer krävande än de statiska, praktiskt användning av dem som ett substitut inom realtidsapplikationer är inte helt orimligt, men kommer inte ske utan uppoffringar på dess antal och grafiska kvalité.

6.2 Diskussion

6.2.1 Renderingstid

Som ett resultat av implementationen av tolkning av L-system till grafisk representation skapas var segment, här synonymt med L-systemssymbol, av trädet som en separat serie vertiser. Detta visade sig under testning ha en tydlig påverkan på renderingstiden hos de statiska representationerna av träd, och även en möjlig påverkan på de dynamiska representationerna. Fler separata serier av vertiser innebär fler *draw calls*, ett rimligt antagande är därmed att snabbare rendering kan åstadkommas genom att gruppera vertiser för att minimera antalet *draw calls*.

Det är även värt att påpeka att effekten av antalet *draw calls* var betydligt mindre synlig för renderingen av det dynamiska träden, vilket till stor del beror på grafens erratiska utseende och dess tydliga beroende på variationen i antalet vertiser. Renderingstiden för de statiska träden visar ett beroende på antalet *draw calls. Då* vertiser för det statiska trädet lagras i grafikminne och renderas på GPU är ett rimligt antagande att variationer i renderingstid beroende på antalet *draw calls* orsakas av GPU. Dynamiska träd beräknas på CPU för att sedan överföras till grafikminne innan de kan renderas, denna process är betydligt långsammare än att rendera baserat på information redan lagrad i grafikminne. En möjlighet är därmed att effekten av variationer i antalet *draw calls* inte visar sig lika tydligt vid rendering av dynamiska träd eftersom processen av att beräkna vertiser på CPU för att sedan överföra dem till grafikminne för rendering är en större begränsning på systemet.

Valet att utföra tolkning av L-system på CPU grundas i Baele och Warzees (2005) påstående om att tolkning på GPU är problematiskt på grund av hårdvarubegränsningar i upplösning

hos *framebuffer*. Baele och Warzee (2005) skapade dock statiska träd med mer komplex geometri än vad som gjorts i det här arbetet. Som ett resultat gäller inte nödvändigtvis deras påstående för det här scenariot. Sedan Baele och Warzees (2005) arbete publicerades har hårdvara hos hemdatorer dessutom utvecklats, och det finns en möjlighet att att de utvecklats tillräckligt för att upplösningen hos *framebuffer* inte längre skall vara ett problem vid tolkning av L-system av den här storleken på GPU. Med undantag för begränsningarna introducerade av *framebuffer* beskriver Baele och Warzee (2005) tolkning av L-system på GPU som det ideala scenariot. Det är därmed möjligt att en lösning baserad på modern hårdvara som utnyttjar GPU för tolkning kan ge bättre resultat än lösningen som redovisats här.

6.2.2 Samhällsperspektiv

Som ett resultat av de begränsningar som placerats på L-systemet för att tillåta exekvering i realtid har stora delar av dess biologiska användbarhet begränsats. Resultatet är ett simplifierat system kapabelt till att generera strukturer som uppvisar grovt approximerade biologiska koncept. Med andra ord kan systemet skapa strukturer som ser ut som träd och uppvisar biologiska koncept, men som inte gör det som ett resultat av biologiskt korrekt simulation. Systemet lämpar sig därmed för applikationer som till exempel spel, där det finns en önskan till att ha träd som ser realistiska ut, men som inte kräver biologisk korrekthet.

Fokus för designen av systemet har varit realtidsanpassning, att få det enkelt och effektivt nog att kunna exekvera i realtid. Detta innebär dock inte att systemet endast är lämpat för att exekvera i realtid, ett snabbt system kan vara önskvärt även i en rad situationer där generering inte sker i realtid. Ett möjligt applikationsområde skulle kunna vara landskapsarkitektur. Där skulle det kunna tillåta arkitekten att visualisera hur ett område skulle kunna se ut om ett antal år då träd av en viss art planteras, hur stora träden blir i förhållande till området, om de skulle blockera vyn för närliggande hus, och hur träden skulle skugga området. Ett annat exempel är att kombinera trädsystemet med ett simulationssystem för eld, för att på så vis simulera spridning av skogsbrand. I båda dessa fall kan det vara önskvärt med träd som uppvisar realistiskt utseende eller beteende, men som inte kräver samma nivå av biologiskt korrekthet som presenteras av Lam och King (2005). Därmed skulle högre exekveringshastighet med rimlig korrekthet kunna vara att föredra över ett långsammare system med hög biologisk korrekthet.

6.2.3 Begränsningar hos systemet

Då genereringstiden för dynamiska träd visade sig utgöra en relativt liten del av den totala exekveringstiden och då det sannolikt går att utföra markanta förbättringar på renderingstiden, är det säkerligen möjligt att i någon mån expandera funktionaliteten hos L-systemet. Implementationen som redovisats här har två huvudsakliga begränsningar som försvårar skapandet av biologiskt korrekta träd. Den första är begränsningen till en parameter och dess associerade parameteroperationer. Genom att utöka antalet tillåtna parametrar och parameteroperationer skulle systemet kunna beskriva mer komplexa beteenden, och därmed ge stöd för mer komplexa beteenden. Den andra stora begränsningen är avsaknaden av extern påverkan på systemet i form av till exempel kollisionsdetektering och tillgång till resurser som ljus, vatten, och näringsämnen. Kollisionsdetektering är av speciellt värde för spel i form av kollision mellan träd och spelentiteter, kollision mellan träd i syfte att hindra grenar och träd från att växa in i

varandra kan dock sannolikt vara för krävande för spel, men kan vara rimligt att använda i andra typer av realtidsapplikationer.

6.2.4 Forskningsetiska aspekter

Resultaten i arbetet redovisas som de mättes, inga värden inom testserierna har sorterats bort som ett resultat av avvikande eller oförväntade värden. De träd som representeras i serien för renderingstid beroende på L-systemslängd som redovisas i Figur 6 har däremot selektivt valts ut baserat på dess L-systemslängd och hur väl dess vertisantal kunde anpassas till ett gemensamt värde. Systemet som implementerats i detta arbete har en tendens att generera fler träd med L-systemslängder i den nedre halvan av det testade intervallet än i den övre. Mätprocessen för det här testet är dessutom mer tidskrävande än övriga tester som redovisas i arbetet. Utöver att generera träden har även antalet vertikala segment hos cylindrarna som utgör stammen hos träden justerats för att alla träd i testet ska ha ett så likt antal vertiser som möjligt. Justeringen agerar globalt och identiskt på alla cylindrar i trädet, men har inte automatiserats, och måste därmed utföras manuellt. Utöver detta tar testet i sig en ej försumbar tid att utföra, det har därmed inte varit realistiskt att testa alla träd med kortare L-system som genereras som en biprodukt av att hitta ett *seed* (eng. frö) som producerar de mer sällan förekommande träden med längre L-system.

För att inkludera de statistiskt underrepresenterade träden med längre L-system har ett stort antal träd genererats så att dess L-systemslängd kunnat utvärderas. Träd med L-systemslängder redan representerade i undersökningen har ignorerats medans orepresenterade träd har vertisjusterats. I fall där den minsta möjliga vertisskillnaden överskridit hundra vertiser från något redan testat träd har dessa uteslutits från undersökningen. För orepresenterade träd där antalet vertiser går att justera inom det önskade intervallet har testning skett på samma sätt som för övriga tester.

Både pilottestet och testet för renderingstid i förhållande till antal vertiser har utförts två gånger. Resultaten från det första genomförandet har i båda fall avfärdats på grund av påverkan på resultaten från tvingad vertikalsynkronisering som ett resultat av grafikkortsinställningar, då dessa förbisetts innan testernas utföranden. Då renderingstiden mäts genom att grafikenheten signalerar när alla grafikoperationer har utförts innebär tvingad vertikalsynkronisering att tidsmätningen synkroniseras med uppdateringsfrekvensen hos den bildskärm som grafikenheten anpassar sig efter. Testerna från vilka resultaten som redovisas i arbetet härstammar har därmed skett utan vertikalsynkronisering.

För att minimera effekten av oförutsägbar belastning av hårdvaran på resultaten används medel och medianvärden baserade på ett antal tester. Antalet tester som medel och medianvärdena grundades i baserade på undersökningen som presenterats i Figur 3. Då tusen tester gav snarlika resultat som både tiotusen och hundratusen tester anses tusen tester vara tillräckligt många för att statistisk pålitliga resultat.

Hur L-system fungerar och beskrivs med formella termer är väl definierat. Det finns däremot ingen sådan formell beskrivning för hur L-system skall realiseras i ett digitalt medium. L-system kan därmed implementeras på många olika sätt utan att bryta mot L-systemsdefinitionerna. Implementationen som redovisas i det här arbetet är därmed inte representativ för alla möjliga L-systemsimplementationer eller L-systemets potential. Den utförda undersökning kan därmed inte svara på hur effektivt ett optimalt implementerat L-

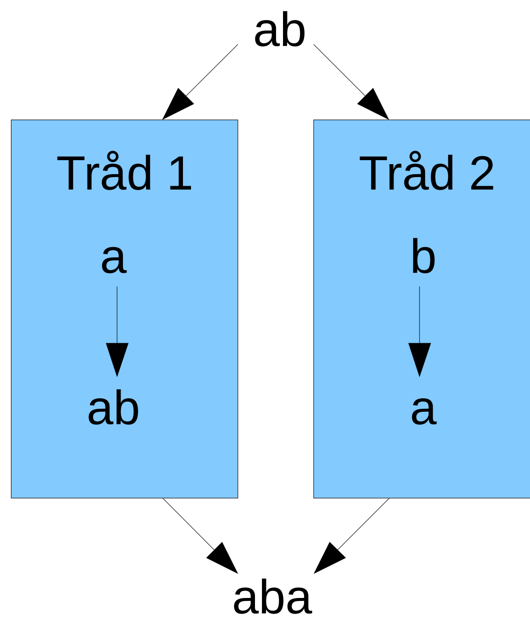
system kan prestera. Den kan däremot visa på att L-system kan prestera minst på samma nivå som lösningen presenterad i detta arbete, samt att det finns ett antal tydliga problem med implementationen som om lösta sannolikt skulle kunna få den att prestera betydligt bättre.

6.3 Framtida arbete

Då resultaten tydligt visar på att renderingstiden för den statiska representationen av träd beror på antalet segmenteringar av vertiser skulle fortsatt arbete framförallt fokusera på att konvertera vertiserna till en serie. Detta är huvudsakligen relevant för den statiska representationen men skulle även kunna ha påverkan på den dynamiska lösningen, då det skulle tillåta att hela träd överförs i ett stycke till grafikkortet vid rendering. Även om båda fall skulle optimeras lika mycket, då de i nuvarande läge har identisk segmentering, skulle det innebära att jämförelsen blir mer relevant.

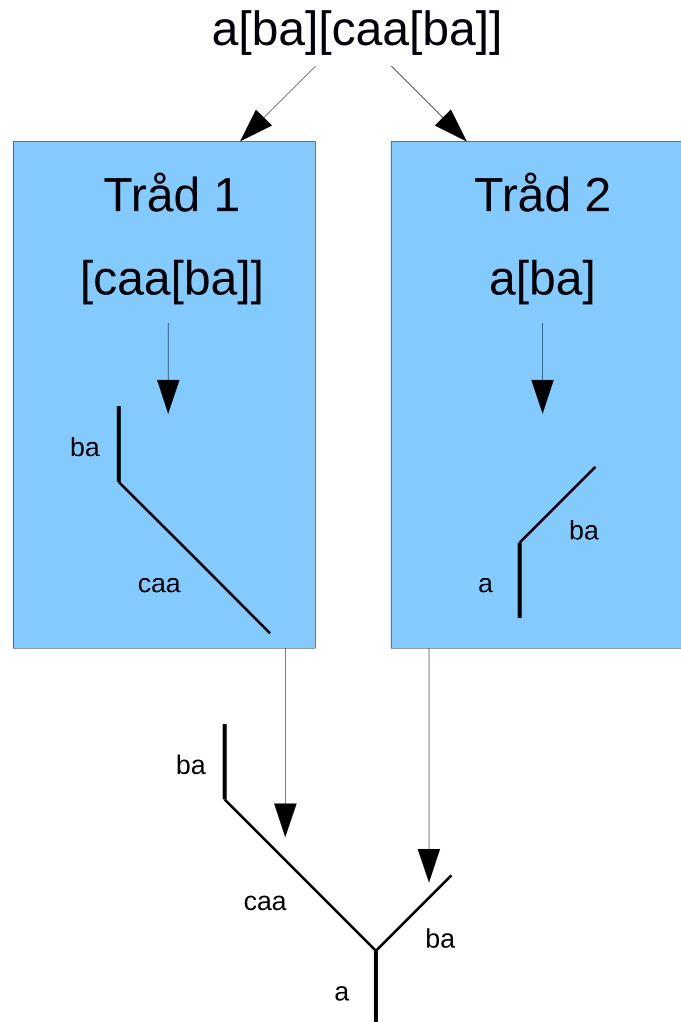
Baele och Warzee (2005) valde att utföra sin tolkning av L-systemet på CPU då de påstod att detta skulle vara den mest praktiska lösningen, vilket ledde till att samma beslut fattades för det här arbetet. Baele och Warzee beskriver dock tolkning av L-system på GPU som den optimala lösningen, då detta tillåter skapandet av geometri helt utan användning av CPU, vilket därmed leder till mindre dataöverföring mellan de två enheterna och snabbare renderingstider. De beskriver en sådan lösning som lovande, men problematisk på grund av hårdvarubegränsningar i upplösningen hos grafikenhetens *framebuffer*. Teknikens potential gör att det är intressant att undersöka även träd baserade tolkning av L-system på GPU.

L-system är parallella system, det vill säga att alla förändringar mellan två iterationer av systemet sker under samma tidssteg. I det här arbetet emulerades detta genom att använda två strängar: en arbetssträng i vilket den nya iterationen av L-systemet skapas, och en referenssträng innehållande den tidigare iterationen från vilket den nya baseras på. Ett givet framtida arbete är att implementera ett system där dessa operationer faktiskt sker parallellt via användandet av flera trådar. En sådan lösning kan sannolikt producera betydligt bättre resultat än den som redovisats här, skillnaden bör vara av speciellt värde för långa L-system. Av speciellt intresse är att undersöka i vilken grad systemet skulle kunna parallelliseras och vilka prestandakrav som introduceras av att administrera parallelliseringen.



Figur 10 Grafisk illustration av uppdelning av en L-systemssträng över två trådar

Figur 10 visar hur genereringen av L-systemssträngar skulle kunna delas upp över flera trådar för att minska tiden det tar att applicera produktionsregler på strängen. Då var föregångare i produktionsreglerna består av endast en symbol kan L-systemssträngen teoretiskt delas upp över lika många trådar som det finns symboler, i praktiken skulle detta dock sannolikt resultera i allt för mycket administration av trådar och för många kontextbyten för att vara effektivt. En bra balans mellan antalet trådar och L-systemslängden kan dock ge goda resultat. För kontextberoende system kan en referenssträng användas på samma sätt som det används i det här arbetet för att hantera kontext, då informationen från denna sträng endast läses utan att modifieras bör alla trådar kunna läsa från samma sträng.



Figur 11 Illustration som visar hur delar av ett L-system kan genereras separat för att sedan renderas tillsammans

Figur 11 visar hur parallellisering skulle kunna appliceras även på sköldpaddstolkning av L-system genom att generera L-systemet i stycken, som sedan antingen kombineras till en entitet eller renderas som separata objekt på ett sätt som får det att se ut som att de tillhör samma entitet. För att detta skall fungera kan det dock vara önskvärt att instruktioner i L-systemet till så stor del som möjligt antingen är självständiga eller grupperade med tillhörande instruktioner. Det vill säga att om flera instruktioner, här synonymt med symboler, beror på varandra bör de behandlas av samma tråd. Ett bra exempel är *[* och *]* där en stängande hakparentes innebär att sköldpaddan laddar ett tidigare tillstånd, vilket kräver att denna känner till tillståndet. Situationer där trådar skulle behöva dela sådan information med varandra kan vara svåra att hantera och kan resultera i försämrad prestanda. Uppdelningen i Figur 11 leder till att trådarna hanterar olika långa strängar för att hålla dem oberoende av varandra, men detta är sannolikt att föredra över att dela information mellan dem. Vid parallellisering av generering såväl som tolkning av L-system kommer någon form av administration att behövas för att dela upp strängar mellan trådar och kombinera resultat från dem, det är sannolikt denna administration som kommer vara den största utmaningen med en parallelliserad implementation av L-system, både ur en prestandasynpunkt och ur en implementationssynpunkt.

Som vidare arbete hade det dessutom varit intressant att undersöka möjligheten att arbeta in vertiser som en del av L-systemets växtfunktion, för att på så vis minska antalet beräkningar

utförs vid rendering. I sitt nuvarande utförande beräknas alla vertiser om från grunden vid var rendering, om vertispunkter istället kunde flyttas allteftersom trädet växer skulle viss data kunna återanvändas. Resultatet skulle bli ökad minnesanvändning till fördel för minskad renderingstid, vilket sannolikt kan vara att föredra i många praktiska applikationer av tekniken.

Genom att utöka funktionaliteten hos L-systemet till att tillåta fler parametrar och mer komplexa parameteroperationer samt att introducera påverkan från externa variabler och funktioner, till exempel ljusnivåer och kollisionsdetektering, finns det mer potential för att generera träd som är både mer biologiskt korrekta och som ser mer realistiska ut.

Målet med arbetet har varit att jämföra samma träd representerat med två olika tekniker. Det viktiga har därmed varit att träden är lika snarare än hur tekniskt komplexa de är. Träden som presenterats här saknar därmed vissa egenskaper som är önskvärda vid användning i spel, den viktigaste av vilken är texturering. Ett alternativ till cylindrarna som utgör stammen på träden i det här arbetet är att använda en eller ett flertal modeller som trädsegment, för att på så vis ge trädet ett mer realistisk utseende.

Ytterligare en aspekt av träd i realtidsapplikationer som inte utforskas i det här arbetet är *level of detail* (eng. detaljnivå), vilket refererar till rendera entiteter i olika detaljnivåer beroende på ett antal faktorer gällande synlighet, den vanligaste av vilka är avstånd från kameran. Då de L-systemsbaserade träden genereras vid var rendering lämpar sig systemet bra för variabel detaljnivå genom att till exempel rendera de cylindrar och sfärer som utgör stammen med olika antal segment beroende på avstånd från kameran. Om modeller används för att representera segment i stammen skulle modeller i olika detaljnivå kunna användas istället. Det skulle även kunna vara möjligt att tolka symboler annorlunda, till exempel ersätta en cylinder med en *billboard* om avståndet är tillräckligt stort. En *billboard* är en tvådimensionell textur som alltid är vänd mot spelaren. Det skulle även vara möjligt att ignorera delar av L-systemet beroende på parametervärden, för att på så vis till exempel enbart rendera grenar som är stora nog för att synas från det aktuella avståndet. Prestandavinsterna av att använda olika detaljnivåer har inte undersökts i det här arbetet, men är något som är av stor relevans för realtidsapplikationer och därmed lämpligt att undersöka i ett framtida arbete.

I de träd som presenterats här representeras individuella löv renderade i form av dubbelsidiga polygoner, som ett resultat är antalet löv lågt för att hålla nere komplexiteten. Ett sätt att representera löv i spel är genom att applicera en delvis transparent textur illustrerande löv på ett tvådimensionellt dubbelsidigt plan. På så sätt kan ett stort antal löv renderas med liten prestandapåverkan, då det ger det dynamiska trädet ett stort antal löv med ett lågt polygonantal såväl som ett simpelt L-system. Som framtida arbete skulle det vara intressant att expandera systemet med dessa tekniker för att få en full bild av hur träden skulle prestera och se ut i ett antal format önskvärda för praktiskt användning i realtidsapplikationer.

L-systemet som presenterats i det här arbetet har skapats med syftet att generera träd, det finns dock inget utöver produktionsreglerna som gör det mer lämpat för att skapa träd över någon annan typ av växtlighet. Talton et. al. (2011) nämner dessutom städer, byggnader, och landskap som saker som kan genereras med hjälp av procedurella tekniker som till exempel L-systemet. Sköldpaddstolkningen som implementerats i det här arbetet är gjort specifikt för de trädstrukturer som demonstrerats, med mycket lite flexibilitet. L-systemet är däremot ett

generellt L-system med anpassningsbart axiom och produktionsregler, det kan därmed användas till att generera en rad olika former, mönster, och strukturer. Systemet är designat för användning i realtid, effekten av detta är att det är något simplifierat för att åstadkomma snabbare exekvering, men det finns ingen anledning till att en snabb algoritm inte skulle vara av nytta även när generering inte sker i realtid. Möjliga framtida arbeten skulle därmed kunna utforska möjligheten att använda systemet till att generera andra saker än träd, såväl som att testa systemets användbarhet utanför realtidsapplikationer.

Referenser

Baele, X. & Warzee, N. (2005) Real time L-system generated trees based on modern graphics hardware. 2005 International Conference on Shape Modeling and Applications, ss. 184 - 193. DOI: 10.1109/SMI.2005.38

Castellanos, E., Ramos, F. & Ramos, M. (2014) Semantic death in plant's simulation using Lindenmayer systems. Natural Computation (ICNC): 2014 10th International Conference on Natural Computation, ss. 360 - 365. DOI: 10.1109/ICNC.2014.6975862

CD Project RED. (2015). The Witcher 3: Wild Hunt(1.08)[Datorprogram].Tillgängligt: thewitcher.com/witcher3 [2016-02-18]

cplusplus (2000-2016). Standard C++ Library reference. Tillgänglig: http://www.cplusplus.com/ [2016-04-06]

Deep Silver. (2015). Saints Row IV(1.0.6.1)[Datorprogram] Tillgänglig: https://www.saintsrow.com/ [2016-02-18]

Enberg, S. (2007) Kontextberoende generering av träd för dataspel. Skövde: Institutionen för kommunikation och information.

Epic Games (2004-2015). Static Meshes. Tillgänglig: https://docs.unrealengine.com/latest/INT/Engine/Content/Types/StaticMeshes/index.htm [2016-02-18]

G-Truc Creation. (2005-2016). GLM(0.9.7.4)[Datorprogram] Tillgänglig: http://glm.g-truc.net/0.9.7/index.html

Interactive Data Visualization. (2002-2015).SpeedTree(7.1.2)[Datorprogram] Tillgänglig: www.speedtree.com/ [2016-02-18]

Lam, Z. & King, S. A. (2005) Simulating tree growth based on internal and environmental factors. GRAPHITE '05 Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, ss. 99-107. DOI: 10.1145/1101389.1101406

Magdics, M. (2009) Real-time generation of L-system scene models for rendering and interaction. SCCG '09 Proceedings of the 25th Spring Conference on Computer Graphics, ss. 67-74. DOI: 10.1145/1980462.1980478

Microsoft (2016). Visual C++ in Visual Studio 2015. Tillgänglig: https://msdn.microsoft.com/en-us/library/60k1461a.aspx [2016-04-06]

SGI (1997-2016). OpenGL API Documentation Overview. Tillgänglig: https://www.opengl.org/documentation/ [2016-04-06]

Prusinkiewicz, P. & Lindenmayer, A. (1990) The Algorithmic Beauty of Plants. New York: Springer-Verlag.

Prusinkiewicz, P., Lindenmayer, A. & Hanan, J. (1988). Development models of herbaceous plants for computer imagery purposes. SIGGRAPH '88 Proceedings of the 15th annual

conference on Computer graphics and interactive techniques, ss. 141-150. DOI: 10.1145/54852.378503

Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. & Varshney, A. (1992). Real-time procedural textures. I3D '92 Proceedings of the 1992 symposium on Interactive 3D graphics, ss. 95-100. DOI: 10.1145/147156.147171

SDL Community. (2016). SDL(2.04) [Datorprogram]. Tillgängligt: https://www.libsdl.org/index.php [2016-04-06]

Talton, J. O., Lou, Y., Lesse, S., Duke, J., Měch, R. & Koltun, V. (2011). Metropolis procedural modeling. ACM Transactions on Graphics, Volume 30 Issue 2, Article No. 11. DOI: 10.1145/1944846.1944851

Todd Howard DICE 2012 Keynote (2012) [video]. GameSpot. Tillgänglig: https://www.youtube.com/watch?v=7awkYKbKHik [2016-02-17]

Ubisoft Montreal. (2014). Far Cry 4(1.9.0) [Datorprogram]. Tillgänglig: far-cry.ubisoft.com/fc4/en-gb/home/ [2016-02-18]