An Investigation into Animating Plant Structures within Real-time Constraints

Matthew Caldwell
The University of Huddersfield
School of Computing and Engineering

September 2015

Abstract

This paper is an analysis of current developments in rendering botanical structures for scientific and entertainment purposes with a focus on visualising growth. The choices of practical investigations produce a novel approach for parallel parsing of difficult bracketed L-Systems, based upon the work of Lipp, Wonka and Wimmer (2010). Alongside this is a general overview of the issues involved when looking at growing systems, technical details involving programming for the Graphics Processing Unit (GPU) and other possible solutions for further work that also could achieve the project's goals.

Contents

Glossary

Grammar In computer science a formal grammar is a set of rules used to define a strings structure. 10, 12, 14, 15

Kernel A computer program managing input and output requests and instructions. 21, 25, 26, 29–31

Shader These are sets of instructions for the gpu to process. They normally come in different types for various stages of the gpu pipeline. 18–20

String A computer programming term for a collection of characters or letters. 5, 10, 11, 13, 14, 16, 18, 19, 21, 26, 27

Thread A thread is a term used to refer a set of instructions being carried out, typically multiple threads run in parallel. 21, 24, 26, 27, 29–31

Time Box An allotted space of time used in project management. 3, 4

Acronyms

API Application Programming Interface. 19, 20

CPU Central Processing Unit. 16, 19, 28, 30, 31

FPS Frames Per Second. 7

GPGPU General-Purpose Computing On Graphics Processing Units. 16, 18

GPU Graphics Processing Unit. 1, 6, 9, 10, 16–21, 24, 29, 31

HLSL High Level Shader Language. 20

IFS Iterated Function System. 7, 8

LBDP L-System Bracket Depth Parsing. 24, 26, 29–31

LOD Level Of Detail. 6

VBO Vertex Buffer Object. 21

1 Introduction

Animating plant life presents many problems for computer graphics; large amounts of detailed geometry, complex underlying mechanics and various visual effects that need to be applied for a realistic result. The problems have been tackled fairly well for some years with many techniques discovered and applied. There are two options for creating digital representations of plant life, they can be procedurally created by an engine at run time or they can be modelled by an artist in a separate software package then stored in memory for later use. Modelling in the past was tedious work as dense scenes contain large amounts of very complex geometry, commonly leading to a mass of identical foliage due to time constraints. The industry focus has been on allowing designers to quickly model plant life and then populate a scene which has been quite successful with various options available (Lintermann and Deussen, 1999; Interactive Data Visualization Inc, 2012).

Procedurally generated content is a more difficult scenario. Many systems will procedurally place pre-built models across a scene to various levels of complexity (Cohen, Deussen, Hiller and Shade, 2003) which is another method of creating rather vibrant scenes with minimal effort. The research done here will be focusing on procedurally creating the plants themselves.

This project's primary goal is a highly optimized technique focusing on the growth of botanical systems, working to a *'Firm real-time'* restriction (Mok, 1996). Growing systems present even greater challenges with developments in this area being either slow, heavy load simulations or detailed animations for pre-rendered scenes, likely due to the constantly shifting geometry and updates to the underlying data required.

There will be a balance in approaching this goal between how demanding any investigated/created technique is and what that technique can achieve; an optimal result being a large number of objects or systems that follow naturally accurate growth pattern while allowing for heavy user interaction. Regardless of how demanding any creation is, it should be focused on optimisation. This goal pushes the project's research towards the entertainment demographic where lowering the hardware demand is particularly important. However if an optimised technique can be found which allows for the editing of environmental parameters affecting growth and produces a botanically accurate result, then the application will perhaps engage a scientific demographic.

For a clear definition of the project goals the MoSCoW method is employed. With it four different levels of prioritisation are layed out which are to be completed within a time box. Miranda (2011) lists the MoSCoW rules as follows:

1. Must Have: Those features that the project, short of a calamity, would be able to deliver within the defined time box

2. Should Have: Those features that have a fair chance of being delivered within the defined time box

3. Could Have: Those features that the project could deliver within the defined time box if everything went extraordinarily well, i.e. if there were no hiccups in the development of requirements assigned to higher priority categories

4. Wont Have features, those for which there is not enough budget to develop them

For this project the time box will cover the entirety of research, implementaion and analysis. Laying out these goals makes evaluation of a chosen methodology clearer, in terms of how many and how important the achievable priorities are in relation to the MoSCow method.

1. Must Have: Animated growth of plant life.

2. Should Have: Innovation, optimisation or investigation of currently applied techniques.

3. Could Have: Scientifically accurate representations or applications. Ability to handle large loads of data or replicate the results.

4. Wont Have: Incorporation of modern computer graphic techniques based around lighting and texturing.

Possible clients might include; forestry commissions or studies of botany if simulation aspects are followed allowing them to display future possibilities, in contrast specific computer game applications where real time growth adds to the player's enjoyment in some fashion will find simpler low load techniques of use. Also those interested in architectural modelling and possible effects of plant life growth to an area may find value in the work. Research should not only cover the expansive subject matter of botanical simulation but also venture into current computer graphics developments. Relevant research will need to be individually assessed, understood and evaluated with the aim of finding room for innovation.

2 Research

2.1 L-Systems

L-Systems (short for Lindenmayer-Systems) were developed by Aristid Lindenmayer *'as a theoretical framework for studying the development of simple multicellular organisms'* (Lindenmayer, 1968), their application for modelling larger biological structures followed soon after. The systems consist of formal grammars based upon the process of rewriting. They start with an 'axiom' string and apply 'production rules' to each character. The rules replace the characters with a new string, this is done in parallel across the original string. The process is repeated as many times as required. These strings are used to form a geometry, each character representing a structure or change in direction. The strings can also represent other states of the plant which may not be visible.

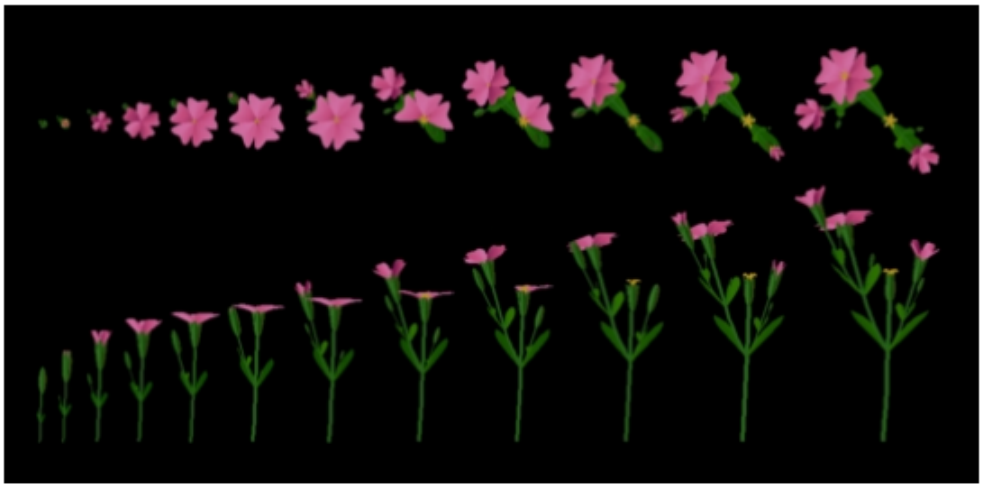

Figure 1: Development of *Lychnis coronaria* (Prusinkiewicz et al., 1993). An example of L-Systems used to model growth

They have undergone a large number of improvements to the basic underlying principles allowing for more adaptability; many of these changes build upon one another to produce different effects. For this projects goals the focus is on applications which apply L-System growth in real-time. Research involved with L-Systems and development of botanical structures are more often highly intensive with detailed processes such as plant light in-take (Soler, Sillion, Blaise and Dereffye, 2003) or root growth and distribution (Leitner, Klepsch, Bodner and Schnepf, 2010). These types of work don't aim for

user interaction as the simulation progresses, however with advancement of hardware the benefits of such a feature could provide unforeseen advantages.

An interesting adaptation is the introduction of Level Of Detail (LOD) data for L-System constructions (Lluch, Camahort and Vivó, 2003; Lluch, Camahort and Vivó, 2004), modern engines rely heavily on this data to construct expansive, detailed scenes while keeping a low demand. Other real-time focused research in this area advices interpretation of L-Systems on the GPU(Baele and Warzée, 2005) which will be assessed in section 4.

2.2 Billboarding

Billboards are flat, semi-transparent textured polygons which rely on keeping the image facing the viewer to simulate a full 3D model. They have become a useful tool for cheaply rendering forests, where a single billboard is repeated and oriented towards the viewer. Billboards struggle with an aerial perspective and can't hold the detail of a polygon model with closer inspections. Advanced implementations tackle these downfalls using a LOD system where trees near to the viewer have modelled skeletons with billboarded foliage and those far away are clusters of billboards (Colditz, Coconu, Deussen and Hege, 2005; Behrendt, Colditz, Franzke, Kopf and Deussen, 2005).

Figure 2: Here a breakdown from full model to clean billboard representation can be seen (Behrendt et al., 2005).

When looking for representations of growth, the process is limited to simple scaling of the texture (Zhang, Teboul, Zhang and Deng, 2007). The transformation process from model to billboard is far too expensive for a real-time adaptation of changing geometry. Even modern examples using optimised approaches, still rendering a static forest, achieve below 15 Frames Per Second (FPS) (Bao, Li, Zhang, Che and Jaeger, 2011). Growth could be represented by these systems using the near view model-billboard hybrids in the future with hardware advancement or large optimisations, yet how to smoothly transition between them and pre-built billboards of the entire tree is difficult to determine.

2.3 Iterated Function Systems, Fractals and Botanical Algorithms

There are a number of methods outside of L-Systems for tree construction which have been investigated in computer science to varying degrees.

Iterated Function Systems (IFSs) are similar to L-systems in that they are a repetition of a process on a set of data. Instead of strings IFSs apply purely to numeric sets, be it a set of real numbers, vectors or both. The data can then be interpreted geometrically and representations of botany can be produced. Fractal patterns are common occurrences in nature, leading Barnsley et al. (1986) to present a simple fern pattern using IFSs showing their ability to create botanical simulations. They have also been found to be interchangeable with L-Systems under certain circumstances (Prusinkiewicz and Hammel, 1994), around the same date visualisations of the two processes were brought together under 'The object instancing paradigm' (Hart, 1992) which is a modelling technique designed to handle recursive structures. When looking to simulate growth IFSs have a similar issue of only presenting large changes to geometry. For a smoother process either a separate system must interpret and visualise

Figure 3: Fractal Fern (Barnsley et al., 1986)

the distance between newly created points
or the IFSs must undergo alterations to the original process (Huaiqing and Min, 2009; Min, Huai-qing and Kang-ning, 2010). For a more in-depth look at fractals and IFSs see Barnsley and Rising (1993).

Different attempts at modelling are more focused constructions with simple variables. Bloomenthal (1985) is a well cited early demonstration which takes point data which is presumed to be manually declared and focuses on representing that data geometrically with 'splines', these consist of vector cylinders spaced across the branches of the tree that are angled to show bending and arching in the model instead of straight lines. Simulating growth in real-time using this technique is a simple case of hardware brute force to model changes. Oppenheimer (1986) shows a similar structure which is applied to animations; also shown is how fractals can be used for creating tree topologies.

The earliest look at growth in terms of simulation focuses on the structure and topology of the plant life. It is a detailed procedural process which is based upon observational study of how plants grow (de Reffye, Edelin, Françon, Jaeger and Puech, 1988). The results are surprisingly visually impressive with varying output for such a tailored process, see figure 4.

Figure 4: From SIGGRAPH '88

The issue with a system so heavily invested in biological data is that recreating these visuals requires a detailed knowledge of botany. Another data based implementation breaks down the visual aspect of tree topology into variables and constants (Weber and Penn, 1995). Here the hope is that one can build trees with less botanical knowledge, they also present ideas on handling level of detail across distance. The most impressive work found using botanical functions was that of Pirk, Niese, Deussen and Neubert (2012). They use a mesh contraction algorithm to create a skeleton from pre-constructed tree meshes, this is used as the starting point for the creation on an animation which represents the trees life span. The application allows for editing of the trees in real-time and quickly alters the animation data to accommodate, fulfilling many the goals this project aims to achieve. Investigating this application for optimisation would be difficult, there is little description of how data is processed, the research is focused on the botanical algorithms and how to produce accurate results. There are also downsides to this approach in that only a certain type of tree structure can be replicated.

More exotic are the particle based systems which require highly intensive computation for rich images (Reeves and Blau, 1985), their demand pushes them away from real-time applications. Finally when looking specifically at a large volume of growing structures Fan, Guan and Tang (2011) present an interesting set of controls for forest simulation, the visualisation is a set of different billboards represetning different ages o as previously mentioned in section 2.2. This could be used as a method for controlling other more elaborate visualisations to create a highly realistic forest simulation. Their suggestions for further work includes GPU implementations.

3 L-System Breakdown

After initial research the project moved towards an investigation into L-Systems on the GPU. To help display how the projects focus was decided below is a table listing each priority defined by the MoSCoW method in section 1 and each of our research topics. Justification for these short notes are in the previous section. The 'should have' item, 'Innovation, optimisation or investigation of currently applied techniques' will be evaluated with the projects final results. 'Won't have' priorites are ignored, all the techniques will likely be compatible with graphic shaders.

Research Topic	Animated Growth	Scientically accurate	Handling Large Load
L-Systems	Varying Complexity	Used by the scientific community for complex simulations	Can become unmanageable quickly
Billboards	Basic representations	Not Applicable	Can be used to render entire forests easily
IFS	Difficult to manage and design	Used to represent fractal patterns in nature	Simple data structure should be easily scaled

L-Systems are a versatile tool and are a key component in botanical simulations, they present the most flexible option in terms of what the research can achieve. Implementations that demand large amounts of geometry can take a single string and duplicate its output or keep the iterations of multiple systems low. More complex simulations can run detailed 'Open L-Systems' with outside variables affecting parameters. When specifically looking at growth, they again offer a range of computational demand with the various grammar rules allowing for multiple effects. Other techniques researched are just as valid options for creating plant life in computer graphics, however for the goals of this research with a focus on real-time growth it was felt that this direction was the best suited in the search for innovative optimisations.

The next sections are a breakdown of L-System properties and features which allow for such varying output. These variants were built into an early prototype and a majority of this writing was taken from previous research (Caldwell, 2013). A more in-depth explanation of these systems can be found in "The Algorithmic Beauty of Plants". This book is heavily cited when it comes to the topic of L-systems and is well regarded (Springer, 2013), it is the original reference for all of the L-system variants given here. The

website `http://algorithmicbotany.org` which is hosted by the University of Calgary (2013) also provides a wealth of resources on the subject of L-system progress.

3.1 Basic L-system

L-Systems come from a history of formal grammars (Chomsky, 1956). The key characteristic that defines them is that the rewriting is carried out in parallel rather than sequentially. Before detailing the significant types of L-System, first the basic terms must be outlined.

- 'Axiom' is the starting point of the string or the state before any changes are applied.
- 'Non-terminal Symbols' are letters or characters which can be replaced.
- 'Terminal Symbols' conversely are items which are not to be replaced.
- 'Production Rules' are a set of rules which dictate the conditions under which non-terminal symbols can be replaced and what replaces them.

The basic L-system begins with the axiom, ω and a set of rules, P.
The rules are applied to the axiom and then the output is stored for the next iteration. Here is a common example:

$\omega = A$
$P1 : A \rightarrow AB$
$P2 : B \rightarrow A$

The first parse will output AB after the rule $A \rightarrow AB$ is applied. On the second iteration AB is then rewritten again to produce ABA, moving across the string from left to right taking A through $A \rightarrow AB$ then B through $B \rightarrow A$. This process is repeated until a desired depth is reached. To transform these strings into some form of graphical representation they need to be parsed. In the early implementations for this project the wire frame was constructed using a process akin to turtle graphics. The *'turtle'* is a stored position and orientation starting at the base of the tree, upon reading a character F (or any other designated letter) the parser moves the turtle forward a set distance and records a new point. Different characters can change

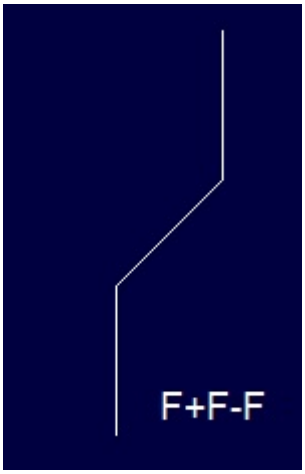

Figure 5: Screenshot from the prototype

the orientation (Here + and − are used to denote changes in angle/rotation) so that the next new point's course is altered, See figure 5 for a visual of the process.

3.2 Bracketed L-systems

The first step in creating tree like structures is the introduction of brackets. To create *'branches'* the *'turtle'* needs to revert to a previous point after reaching the end of a section. To do this a stack is created which stores the orientation and position of the *'turtle'* at these branching points. An L-System signifies this using brackets, `[` being a push onto a stack and `]` a pop.

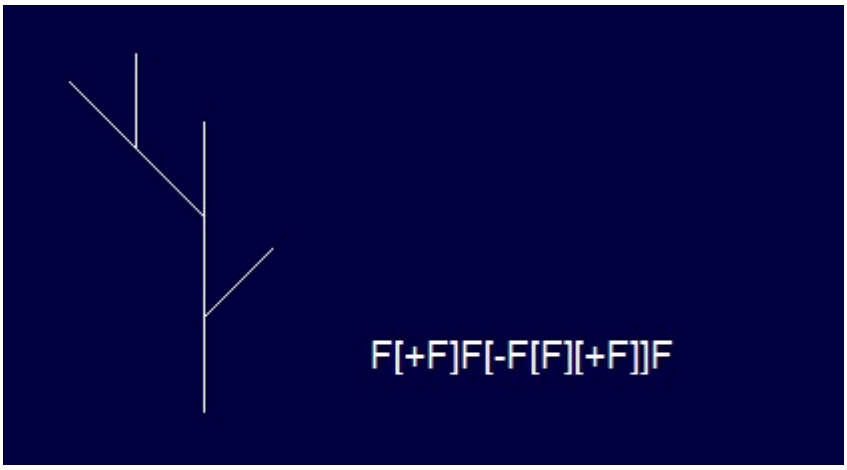

Figure 6: Take note of multiple brackets allowing for branches within branches

3.3 Stochastic L-systems

If the L-System as described so far were used to render a forest, the repeated identical trees would damage the scene's perceived realism. By giving letters in a grammar multiple rules and assigning each of these a probability, highly varying trees from a single L-system definition are possible. Here is a grammar where each rule has an equal chance of being applied ($\overrightarrow{0.33}$ meaning a $33.\dot{3}\%$ chance of application):

$\omega = F$
$P1 : F\overrightarrow{0.33}F[+F]F[-F]F$
$P2 : F\overrightarrow{0.33}F[+F]F$
$P3 : F\overrightarrow{0.33}F[-F]F$

The more iterations in a stochastic L-System the greater the variation in the final string as shown by the output in figure 7.

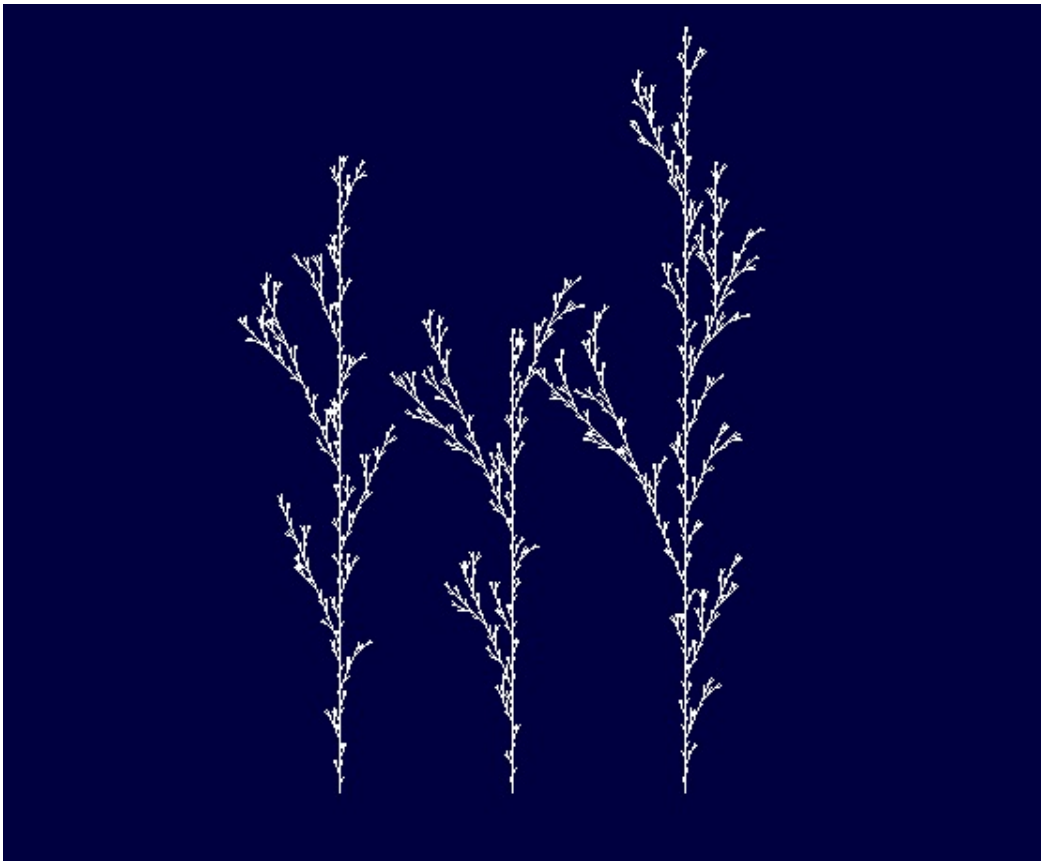

Figure 7: Three stochastic trees

There is an issue with stochastic L-systems however. To implement such a system would require the L-systems to be built in run time, otherwise the process has no value. At high depths, L-system parsing can become quite intensive with hundreds of thousands of characters being analysed. Normally around the sixth to seventh iteration is where most systems become unmanageable.

3.4 Context-sensitive L-systems

Context sensitivity is an addition to the rule declarations that gives them two new requirements before application. A left and right context is to be checked and only when these are both considered to be true can the rule be applied. These systems normally require an *ignore list*, which allows the

parser to skip over characters that aren't involved in the process (+ and − for example) to find the correct context. Bracketed systems also require jumps over closed sections and the letters within them to allow context. Here is the example given in (Prusinkiewicz and Lindenmayer, 1990):

$BC < S > G[H]M \rightarrow S$

Can be applied to S in the string

$ABC[DE][SG[HI[JK]L]MNO]$

To find BC the parse skips [as it has no effect, then $[DE]$ is skipped over entirely as it denotes a branch. To find $G[H]M$ the parser reaches $I[JK]L$ and as H has already been considered true within its bracketed segment the letters can be skipped to find M. Context sensitivity can make parsing much more consuming with the new requirements. One optimisation is to have a data structure where bracket locations are known so that parsing can skip large sections.

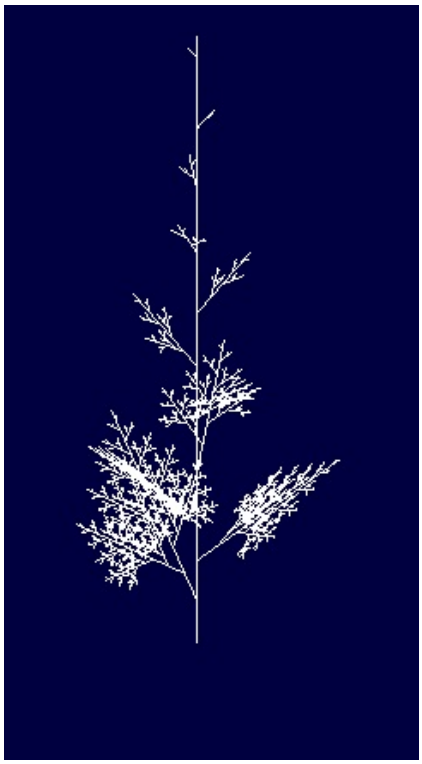

Figure 8: Parametric example. The branches decrease in length with each iteration

3.5 Parametric L-systems

To get a higher level of control Aono and Kunii (1984) built upon some of the early L-systems with the introduction of parameters allowing for varying geometry with a simpler grammar and a better ability to express underlying mechanics; this was well expressed in Prusinkiewicz, Lindenmayer and Hanan (1988), research with a focus on incremental development of Herbaceous Plants. To put it simply, this addition allows parentheses after a letter to denote input to the letters function. For example F in previous systems may have represented a movement of length 5 along the Y axis, parametric L-systems allow F to move any desired length with the string F(x), another application is angles.

This functionality becomes very powerful in allowing for more dynamic and varying tree structures. It can also be used to introduce new features like *'width'* to the L-systems grammar so that the tree can not only produce smaller branches but also thinner ones.

3.6 Timed L-Systems

L-Systems are normally given a specific iteration length and then the process is ended before the data involved becomes too large. This creates a very static, 'jumpy' process if one is looking to use L-systems for animation.

Timed L-systems (or differential L-systems)(Prusinkiewicz et al., 1993) introduce a new parameter to the system rules called *'terminal age'*, this means a rule can only be applied when the letter relating to it has a local age equal to or greater than the terminal age. In this paper's implementations these functions are applied using new rules with a zero terminal age. Rules are attempted in order so that the rules relating to a time are handled first and if they're successful no further rules are read. Other implementations use a growth function that is presumed to be applied in a separate stage to the rewrite process where only the parameters of letters are altered.

$\omega = A(1,1)$
$P1 : A(a,1) \rightarrow A(a,0)B(a,0)$
$P2 : A(a,t) \rightarrow A(a*t,t)$
$P3 : B(a,1) \rightarrow A(a,0)$
$P4 : B(a,t) \rightarrow B(a*t,t)$

Here a is used to represent geometric length when rendering. t is the local time of the letter, which is increased globally by a predetermined static amount or a system variable, called the *'time slice'* or $\triangle t$.

3.7 Environmentally-Sensitive L-Systems

The most recent extension to the grammar is the ability for environmental input and output. This is achieved through the introduction of another command to L-System grammar, the 'query module', normally symbolised by ?. This command returns the current turtle position for a specific letter, which can then be used to alter production rules using context sensitivity. Here is a simple example provided by Prusinkiewicz, James and Měch (1994) in their introduction of the command:

$\omega = A$
$P1 : A \rightarrow F(1)?P(x,y)$
$P2 : F(k) \rightarrow F(k+1)$

This command requires additional step in each rewrite pass for the query modules to be processed, here each step is listed with SN to show the changes to parameters and the string:

$S1 : A$
$S2 : F(1)?P(*,*) - A$
$S3 : F(1)?P(0,1) - A$
$S4 : F(2)?P(*,*) - F(1)?P(*,*) - A$
$S5 : F(2)?P(0,2) - F(1)?P(1,2) - A$
$S6 : F(3)?P(*,*) - F(2)?P(*,*) - F(1)?P(*,*) - A$
$S7 : F(3)?P(0,3) - F(2)?P(2,3) - F(1)?P(2,2) - A$

For output, which was introduced shortly after (Mech and Prusinkiewicz, 1996), the query module can be set with data when written, so $?P(3)$ would pass the value 3 to the environment when it is applied in a production rule. Overall these changes allow for a massive array of new applications. Light and soil can be properly simulated, separate systems can be informed of each other and properly avoid collision, even insect damage to plants can be created (Prusinkiewicz, Hanan, Hammel and Mech, 1997). Particularly impressive is the implementation of an animation system built entirely around L-system rules (Noser and Thalmann, 1996; Noser and Thalmann, 1999; Noser, 2002).

4 Parallel Implementations

From the offset of this research it was expected that General-Purpose Computing On Graphics Processing Units (GPGPU) would be of interest, a sequential process is inherently inefficient for real time applications. It has become increasingly common for applications to offload tasks from the Central Processing Unit (CPU) to the GPU allowing for huge increases in speed across numerous tasks. SIGGRAPH have given examples of improvements with data showing the computational speed differences:

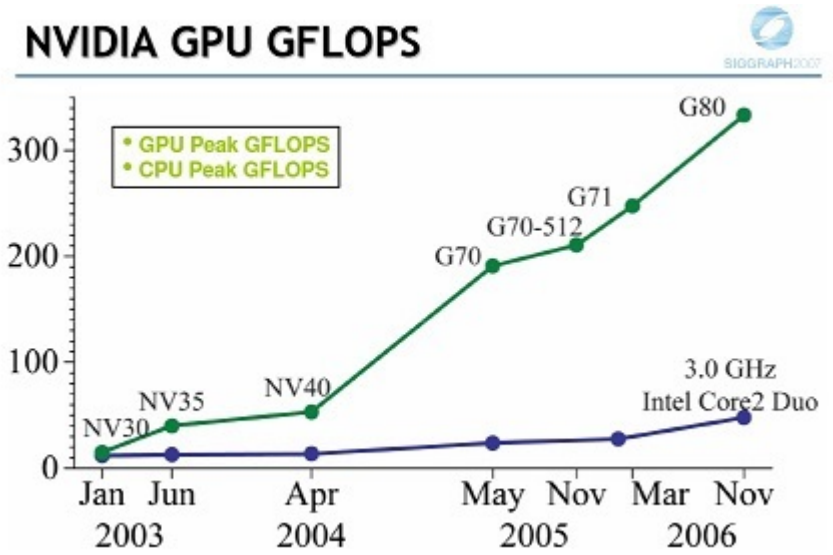

Figure 9: gigaFLOPS comparison (Green, 2007)

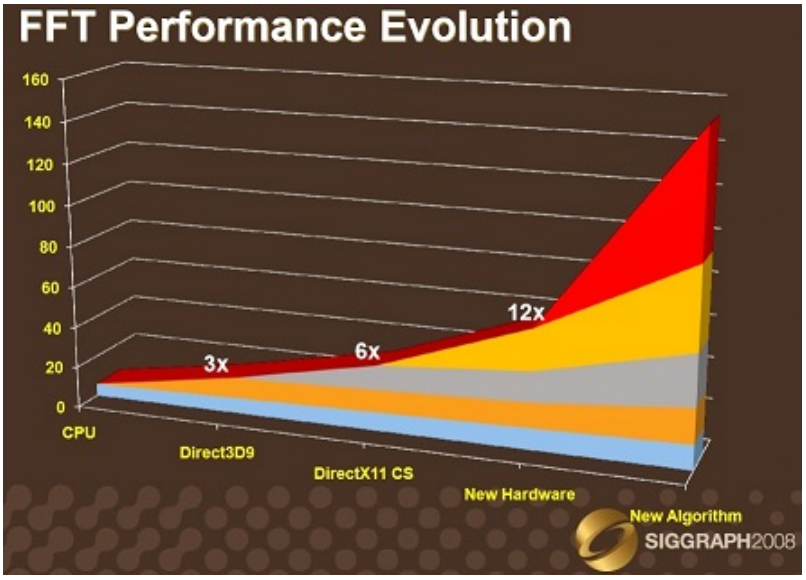

Figure 10: A comparison of FFT performance (Boyd, 2008)

The defining feature of L-systems is that the rewriting process is to be performed in parallel, with any context checks being based on the strings entirety before any changes occur. On the GPU this can be a rather difficult task, the process requires delicate control of memory with lots of shifting

data. A much larger issue however is how L-Systems are parsed, where 'branches' of the string are sequentially dependent on the results before them.

Research finds three key investigations into this subject area, each showing an interesting development with new hardware capabilities. The earliest work (Lacz and Hart, 2004) starts off by tackling how to handle bracketed L-Systems, they present a simple rule:

$L \rightarrow aLf[+L]Lf[L]L$

To parse this in a parallel system, they take each of the rules non-terminal symbols as a separate command (each L) and apply all prior terminal symbols (all $a + -$):

$L \rightarrow \{aL, af + L, afL, aff - L, affL\}$

The downside to this technique is the added calculation involved for each draw command. The greater issue for this early implementation was limited memory storage. Issues cited are a lack of a "render-to-vertex buffer" feature meaning data has to leave main GPU memory to be used in a draw cycle when handling large L-Systems.

Hardware progression and new techniques allowed Budapest (2009) to build upon Lacz and Hart's (2004) work. Their technique is fully GPU processed using the geometry shader. They achieve better memory control and avoid a sorting stage which was the root of many issues. However in doing so they lose the ability to compute context sensitive L-Systems.

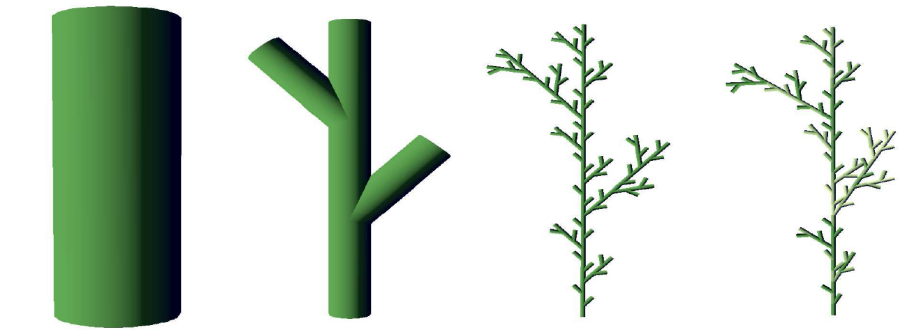

Figure 11: Shapes generated by the L-system given above. From left to right: the axiom (L); after 1 iteration; after 3 iterations; after 3 iterations with color and orientation perturbed by a random amount.

The latest development takes a different approach and doesn't change the underlying principle of the L-System. Lipp et al. (2010) tackle the bracketing with a work queue approach, based upon other GPGPU algorithms for bounding volume hierarchy construction (Lauterbach, Garland, Sengupta, Luebke and Manocha, 2009), Kd-Tree construction (Zhou, Hou,

Wang and Guo, 2008) and Reyes-style adaptive surface subdivision (Patney and Owens, 2008). Advantages of their approach include the return of context sensitive derivation and no longer requiring a shader compilation step present in earlier work:

> *we make explicit use of parallel primitives and do not rely on the graphics pipeline to deal with data amplification and other issues. We fully support productions having side-effects and thus do not need to rely on the specific side effect-free turtle commands presented by Lacz and Hart. Furthermore, we can directly use the productions without requiring a compilation or transformation step.*

Other more recent investigations are based on NVIDIA's CUDA platform. First derivation of the L-System (Liu, Zhang, Zeng, Zhu and Li, 2011) and then their Interpretation (Zhang, Zhu, Liu and Zeng, 2013). The system requires a CPU side scan of the string to prepare data for the GPU. Regardless of this hardware communication their results show a faster output than Lipp et al. (2010).

5 Working On The GPU

The project will be built upon on a C++ Direct3D core, a highly popular choice in the computer games industry meaning any discoveries can be quickly adapted to existing software. OpenGL is arguably the only other real competitor in graphics Application Programming Interfaces (APIs), which focuses on multi-platform applications while Direct3D is Microsoft Windows focused. The 'OpenGL vs Direct3D' debate is a hot topic that's ever changing, at the time of writing a quick look across the web brings up several viewpoints. Here is the Valve Linux team (2012) talking about the performance increase for their engine with OpenGL:

> *why does an OpenGL version of our game run faster than Direct3D on Windows 7? It appears that it's not related to multitasking overhead.We have been doing some fairly close analysis and it comes down to a few additional microseconds overhead per batch in Direct3D which does not affect OpenGL on Windows.*

And here's John Carmack speaking on how his opinions of the API have changed in an interview with bit-tech,"Direct3D handles multi-threading better, and newer versions manage state better." (Hardwidge, 2011) From

current findings, it's believed OpenGL may be a more suitable choice if the project were to move towards a computer simulation focus instead of computer games, due to portability. It has also been decided not to use any additional graphics rendering libraries as any features provided by these would move the project focus towards visual quality and fidelity which is outside of the initial aims. Another investigation was done for Ogre3D, a massive open source engine (Ogre3D, 2012). After some initial testing adoption was not taken.

With Direct3D the choice, developing on the GPU will be done using the compute shader. It is an almost separate stage from the classic pipelines focused on general purpose computing taking advantage of the GPU architecture of parallel processors (Microsoft, 2012). Introduced with the DirectX 11 iteration of the API, the programmer writes in High Level Shader Language (HLSL) and is able to perform most of the basic programming systems like looping, branching statements and structures. For L-System derivation and interpretation in particular, processes which require very different output and input, the compute shader offers a flexible environment for all stages of the implementation. Here are some of the tasks Boyd (2008) lists as possible algorithms which can be implemented on the compute shader:

- Ray-tracing, collision detection
- Inverse kinematics, Physics, AI, fluid simulation, radiosity
- Quad/octrees, irregular arrays, sparse arrays
- Linear Algebra

5.1 The Work Queue Approach

After a second stage of investigation and research the choice was made to focus on the work of Lipp et al. (2010). This project's practical investigations aims were to find improvements. Describing the limitations of their technique:

> *The varying results of the work-queue approach indicate that there may be future work necessary in creating more consistent speedups, maybe a more elaborate work-queue management can achieve this.*

First the work-queue process must be understood. As mentioned this technique is based upon algorithms for other processes (Lauterbach et al., 2009; Zhou et al., 2008; Patney and Owens, 2008). Here is a rough overview of these very different algorithms:

- A set of input data is processed. Perhaps each thread of the GPU taking one item.
- This data creates new items, a 'split', these are sent to the work queue.
- If no new items were created, end the routine. Otherwise begin again.

When applied to L-Systems the process begins at the start of the string which is considered as a single input. The parser moves along a set distance with 'splits' occurring at the final position if there is still data to be parsed and when ever a bracket is hit. The issue here is kernel calls, Steinberger, Kenzel, Kainz, Müller, Peter and Schmalstieg (2014) in their analysis of Lipp et al. (2010) describe the problem:

> *This approach requires a high number of kernel launches interrupted by read-backs from the GPU. Every kernel launch requires the current state to be flushed to slow global memory and read back from the GPU, which stalls the entire device.*

There are around five kernel launches for various processes before parsing can even begin and before that you may want to apply a rewrite which is two more. This situation is compounded by the parsing being such a varying process based upon the L-Systems structure. A rather contrived worst case scenario being a large string with no brackets occurring until the very end. In this situation the parsing suggested will repeatedly create a new kernel launch with only one thread. This may not be an issue that arises too often with most bracketed L-systems allowing for splitting quickly which greatly increases the amount of parsing completed across kernel calls.

L-Systems can be parsed in parallel quite efficiently when brackets are not in use (see figure 12). Each thread parses a section of the string, storing the number of objects and a final matrix representing the position and rotation of the turtle after the last character. This information is then summed across threads and groups. When each thread knows the number of objects and the turtle position before it, data can be output to the Vertex Buffer Object (VBO).

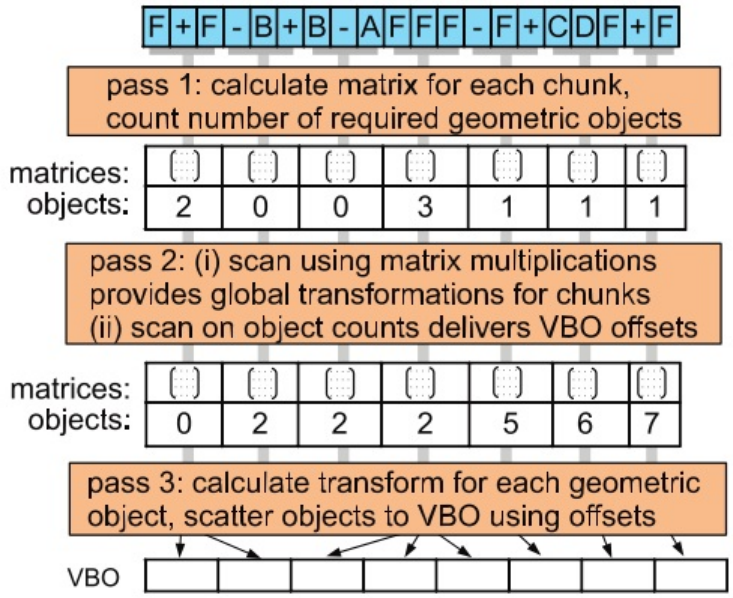

Figure 12: Visualisations of unbracketed L-System parsing (Lipp et al., 2010)

The work queue process traverses almost sequentially, using brackets as a opportunity to begin parallel processing. Knowing this, an approach was taken to parse the L-System in parallel at each depth of the brackets within it. Before explaining this approach lets explain the 'Scan' sorting algorithm, also known as the 'Prefix Sum'. It is a key element used throughout the parsing process.

5.2 Prefix Sum

The 'Prefix sum' algorithm is used to calculate offsets in data when the writing is to be preformed, it is a simple addition of items in a set together as they are iterated. For example:

$Set : 1, 3, 4, 11, 0, 2$

$Output : 0, 1, 4, 8, 19, 19$

Performing this algorithm in parallel also presents challenges. The sequential implementation has a complexity of O(n), to achieve a similar result in parallel the additions are broken down and the entire process is done in two steps.

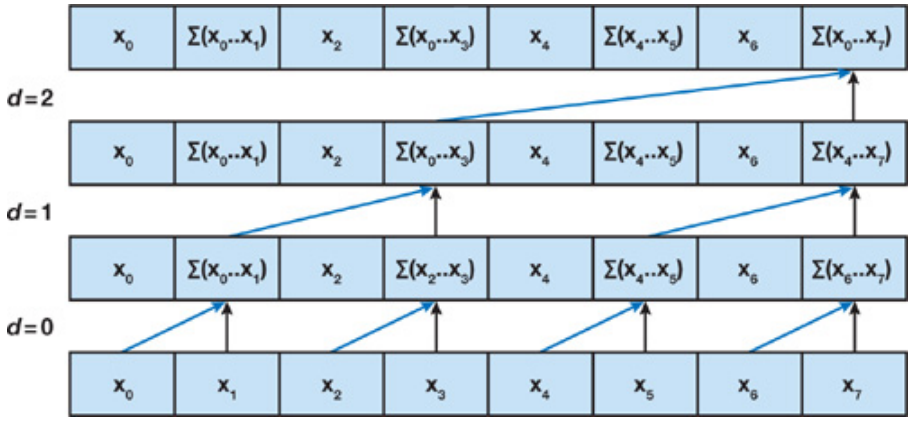

Figure 13: Stage 1, called the 'reduce phase' sums interleaved nodes at over a delimiter

The first stage is a summing of all the values. The set is looped for $\log_2$ of the total, where each loop takes a different total across the set until a final value is found. The second stage begins by setting this final value to zero, then in the same loop goes back across the set making comparisons and swaps of data, creating a gradual sum of values across the set.

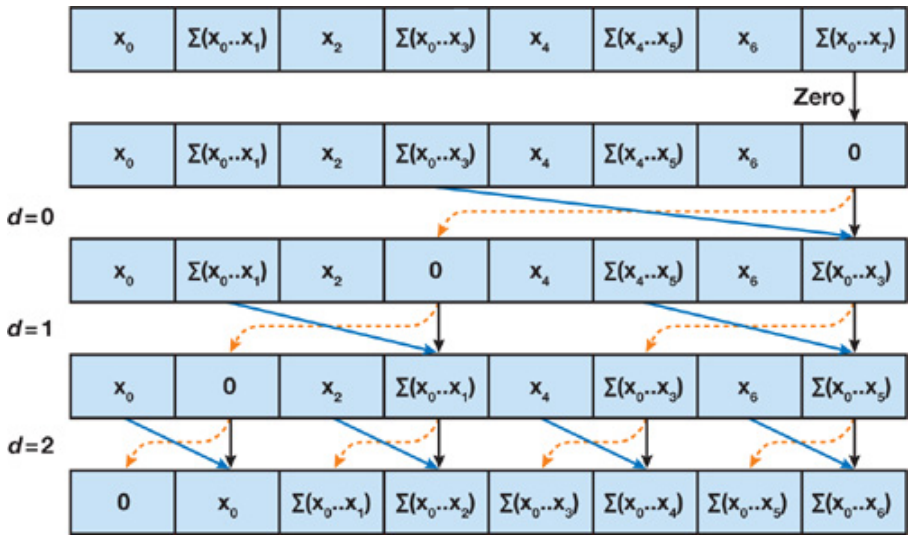

Figure 14: Stage 2, called the 'down-sweep phase' first zeros the last value, then moves through the data swapping and summing the other half of the interleaved nodes with that on the left

This process is only applicable to sets of data which total a power of 2. For an uneven or sizeable amount the process must be padded to a power

of 2, or to take advantage GPU structure, the process is broken down into smaller powers of 2. The final optimisation is the avoidance of an issue called 'Bank conflicts' (Thibieroz and Cebenoyan, 2010). When multiple threads try and access shared memory within the GPU these bank conflicts occur which stall further calculations. Avoiding this is done by padding the data in relation to the *'degree of bank conflict'*.

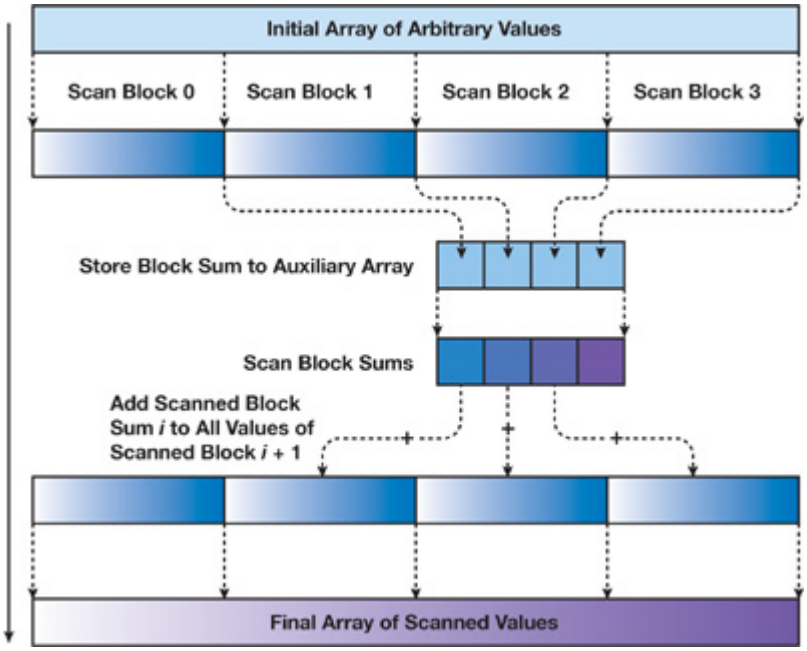

Figure 15: A breakdown of how the algorithm applied to multiple sets in sequence

For a more in-depth explanation of this process and the problems it presents refer to Mark Harris (2007) in 'GPU Gems 3' (Nguyen, 2007), or for layout of the prefix sum algorithm described (Sengupta, Lefohn and Owens, 2006).

5.3 L-System Bracket Depth Parsing

L-System Bracket Depth Parsing (LBDP) is this paper's method of parsing sets of brackets at specific depths in parallel. It is built upon the known method of parallel parsing L-Systems and applies it to the data within brackets separately. It begins by scanning the spaces in-between brackets at each depth, then summing the number of items to create a set of positional data for parsing at each depth.

Before this can be achieved the *Parallel hierarchy extraction* algorithm must be understood (Lipp et al., 2010), a preliminary to the work queue approach. It is used to find bracket positions which allows parsing to continue once a bracket is reached.

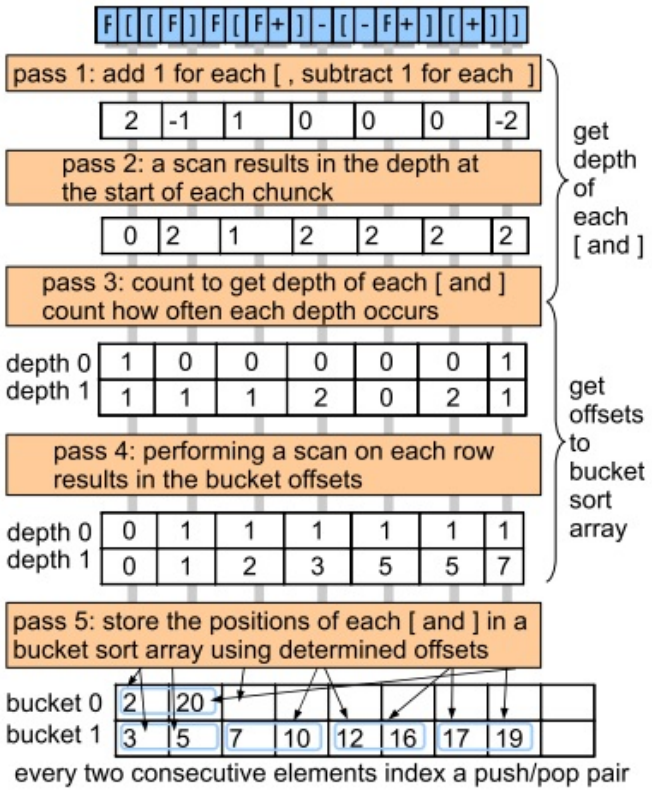

Figure 16: The *Parallel hierarchy extraction* algorithm(Lipp et al., 2010).

The algorithm requires five kernel calls but is only applied once. The first pass calculates the depth of each assigned section, the next pass scans the depth values so that a depth table can be constructed and so that each thread knows the depth it is starting from. The depth table is then filled, and then scanned. The final pass stores the positions of brackets in the bucket using the table's determined offsets.

At this stage of analysis an issue was discovered with this approach which impeded the implementation. The system presumes a known depth maximum. Looking at figure 16 one might presume the depth could be determined from the second pass, however if a pair of brackets open and close within a parse block they become lost. Depth maximum could be found at the rewriting stage based upon the iteration depth with rules that apply

brackets (Ignoring the issue of stochasticism and context sensitivity effecting this) however for a timed L-System where the rewrite process is to be applied every frame this isn't possible. To remedy this an alteration to the process is applied. On the first call the system also records the maximum depth discovered in each block. This value is taken and it is added to the summed value, the maximum in the group is found with a process of similar complexity to the 'reduce stage' of the prefix sum algorithm. In this implementation the data is scanned in the first stage then the second is used to scan the group totals. It's here the final 'true' maximum is deduced.

Another alteration made to accommodate LBDP is to the bucket, a one dimensional array of unsigned integers which is used extensively. The size is increased to hold not only the bracket positions but also the number of previous draw commands, the depth and the count 'above' so the number of brackets within a pair at the next depth is known.

With this extra data the next stage is a calculation of the space in-between brackets. Each bracket pair is given its own thread group (The first group considers the beginning and end of the entire string as a pair of brackets) and each thread finds a bracket within that pair and records the space in-between it and the next bracket pair, storing that value in group memory. Then only the first stage of the prefix sum algorithm is used to find the group total. This total along with other data is stored and then summed again in the next kernel launch which gives each groups an offset telling it how many positional data points are used in the preceding bracket pairs.

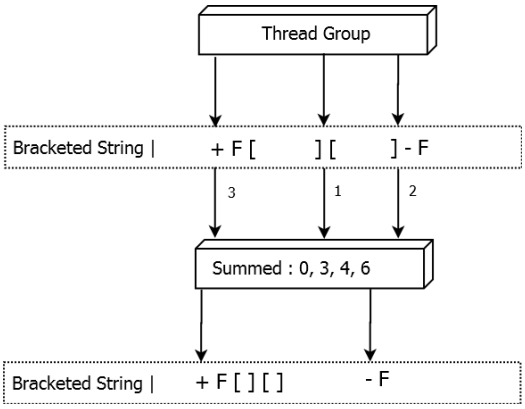

Figure 17: A single thread group's view of finding spacing data, summing it and then creating positional data for parsing

The first step of this is then repeated, using the newly calculated offset to store the positional data which will be used in parsing, see algorithm 1.

Algorithm 1: Setting thread parse positions

```
foreach thread with spatial data to read do
    get relevant bracket data, used as a ParsePosition;
    find the spacing(SpVal) and store in local group;
end
Prefix Sum local group data;
if thread in use then
    get local group data, TrailingVal;
    /* ParseChunk is the number of allotted items each
       thread will parse in later stages                    */
    TrailingNum ← TrailingVal/ParseChunk;
    EndNum ← (TrailingVal + SpVal)/ParseChunk;
    if EndNum > TrailingNum then
        mResult ← TrailingVal mod ParseChunk;
        /* If the items before don't fit the parseChunk, the
           first parse position stored is offset            */
        if mResult ≠ 0 then
            TrailingNum ← TrailingNum + 1;
            mResult ← ParseChunk − mResult;
        end
        ParsePosition ← ParsePosition + mResult;
        write ParsePosition data;
        /* With an inital position set, parse positions are
           written across the space between this threads
           alloted bracket space                            */
        set new variable i to 1;
        while TrailingNum < EndNum do
            TrailingNum ← TrailingNum + 1;
            ParsePosition ← ParsePosition + (i * ParseChunk);
            write ParsePosition data to TrailingNum;
            i++;
        end
    end
end
```

With specific parse positions discovered the process can finally begin calculating the rotational and positional data that the string represents. Much like the previous stage each thread group parses and sums its data, a global sum is applied then a final pass uses this data to write the final results. The difference being data now consists of matrices, similar to figure 12.

This method of parallel parsing must be applied to each depth separately, the overall process looking like this:

1. Parse entire L-System. Count number of draw items, brackets and find maximum bracket depth. Have each group sum this data and pass results into global buffer.
2. Sum the global buffer and transfer results to the CPU for depth looping and buffer sizes.
3. Parse entire L-System again, creating the depth table with brackets. Depth table is then summed.
4. Sum the depth table across groups
5. Parse again, storing the positions, depth and depth table value for each bracket using the depth table to find the offset
6. Now each bracket pair is given a thread group and the total items to be parsed are calculated
7. The total for each bracket pair is summed in a single group, giving each group an offset
8. Using the group offsets, the bracket spaces are read again and positional data is written
9. Now loop across each depth of the L-System.
   (a) Each thread parses the L-System starting at the given position and stores matrix data.
   (b) Matrix data is summed for groups within the same sequential brackets.
   (c) Last parse, setting vertice and indice data
10. L-System is now parsed and the structure can be rendered.

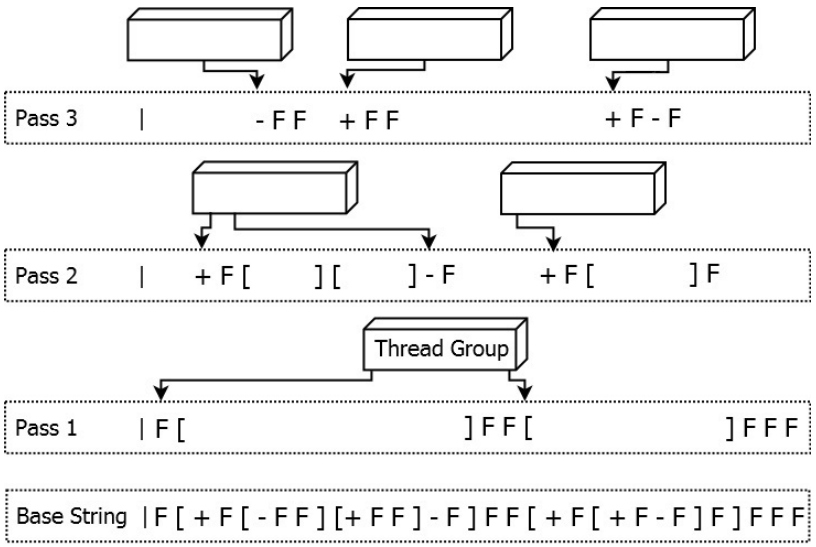

Figure 18: Here is a visualisation of the process. The arrows represent individual threads, take note that in this example the parse block is four items long and bracket jumps are considered to be one item.

6 Conclusion

The aims of the practical investigations were to produce a more efficient parallel parsing solution for the GPU. A working prototype was developed which produced basic vertex data. It would also have been of interest to have built a work queue approach alongside the implementation to compare speeds of parsing. What can be assessed to some degree is whether a reduction of kernel calls in comparison to a work queue has been achieved. How many calls each method requires in practice relies entirely on the L-System being parsed and how many items either system reads per thread. LBDP requires a preliminary 7 kernels over the work queue's 5. Each pass of the work queue will require 2-3 launches based upon Lipp et al.'s (2010) description of their implementation, LBDP uses 3. This flat performance loss is to be gained back across large L-systems where the work queue takes more passes to clear.

Creating a successful GPU based algorithm requires lots of delicate control of memory and good design. Terms commonly used to describe how algorithms perform are occupancy (how much of the GPU is in use), concurrency (making sure all threads are active) and divergence (a term describing a process used to handle if-else statements (Micikevicius, 2012; Woolley, 2013)).

Divergence is the branching within a GPU, normally based around an if statement (Han and Abdelrahman, 2011). This project's implementation

contained a number of branching statements across the process. The implementation did not reach the stage of investigating optimisations presented in Han and Abdelrahman (2011).

In LBDP when thread groups are assigned, any threads that aren't in use are left to stall, creating poor concurrency. The work queue approach can launch just the right number of thread groups using an extra call to scan new items and align pointers, meaning that nearly all threads are in use. For LBDP this issue can be mitigated by small thread groups or possibly by aggregating the data for multiple pairs of brackets that are known to contain small amounts of data. If the data required summing then it would need to be done on a single thread. This could be a solution to be investigated in future work. Another concurrency issue that effects both systems is the natural variation in tasks that the L-System will create, each thread will unlikely be covering an identical load.

Occupancy is the area where LBDP aims to greatly improve on the work queue approach. As mentioned, it is expected that the L-System being parsed decides how well these algorithms perform. For LBDP a small depth with large amounts of data between brackets is optimal. A work queue will perform better with L-Systems which contain large numbers of brackets that are reached quickly. In general the larger the L-System the better LBDP should perform as more items can be processed in the initial kernel launches.

Looking back to section 1 and the MoSCoW priorities listed, we achieved animated growth of plant life. The L-System parser can handle timed L-Systems allowing for smooth changes to geometry. Looking at the 'Should have' priorities, the investigation has approached the specific difficulties of real-time growth and provided a possible new parsing technique to improve performance for a specific implementation. Research should provide those interested in the project aims a wealth of possible avenues for development depending on their choice of approach. 'Could have' priorities like scientific accuracy are dependent on the L-System fed into the parser, L-system parsing can be highly complex as seen in this work, the projects implementation should easily be able to adress most challenges although 'Open L-Systems' could prove difficult. How well LBDP handles large load is dependent on hardware and the data involved, output of any parsed system can easily be duplicated.

7 Future Work

Another method of reducing CPU interaction is a recently introduced technique called 'Dynamic Parallelism' (Jones, 2012). This is a new hardware

capability, where the GPU is able to relaunch threads as if a new dispatch call is made without CPU interaction. A kernel can include a specific command that will dictate a variable number of new thread groups. The choice of platform (Direct3D) unfortunately didn't allow testing of 'Dynamic Parallelism'. Logistically the difficulty with this is how to adapt the system to work within a single call, the work queue system has a break in each pass to scan the new work queue items for the global queue, LBDP had a group summing kernel launch that would likely be an issue. If the technique could be applied to either algorithm it could provide a considerable boost in performance due to the slow delays caused by GPU dispatch calls.

An area that was originally planned to be investigated is 'Skinned/Crowd Instancing' (Dudash, 2007). These systems take a single object, normally along with animation information; and instance that object multiple times. The aims were to build a GPU based animation system using L-Systems where instanced objects would all have their own unique time value representing a certain iteration point of that L-System. A forest of identical trees could easily be achieved with a pre-built animation, including L-System data could allow for stochastic variation, each tree could be given a 'seed' for a random number generator meaning it would always produce the same tree from stochastic L-System data. Instancing is already heavily used to render plant geometry, as either small structures to create basic botanical representations or artist pre-built models instanced to create large biomasses. Crowd instancing has been adapted in interesting ways (Zhou, Tang and Ji, 2013; Peng, Park, Cao and Tian, 2011), if the described technique was successful it would allow for large numbers of growing trees with comparably small computational costs.

Other work which wasn't fully evaluated are techniques that look for short-cuts in the foundations of the systems used to render tree like structures. The first is a look at applying binary trees to L-System rewriting. The rewriting process in the implementation was always far quicker than parsing so investigations into optimisations were not followed. The work of Yang, Huang, Lin, Chen and Ni (2007) could prove to be of some value to an L-System backed implementation if the process described could be applied to the GPU and merged with other systems. Another more recent piece looks at tree's self nested properties, similar to the Hart's (1992) 'Object Instancing Paradigm'. This work is less focused on real-time interpretation of tress and more on compression of data (Godin and Ferraro, 2010). If memory issues ever became an issue this is an avenue which could provide a solution.

8 References

Aono, M. and Kunii, T.: 1984, Botanical Tree Image Generation, *IEEE Computer Graphics and Applications* **4**(5).

Baele, X. and Warzée, N.: 2005, Real time L-system generated trees based on modern graphics hardware, *Proceedings - International Conference on Shape Modeling and Applications, SMI'05* **2005**, 186–195.

Bao, G., Li, H., Zhang, X., Che, W. and Jaeger, M.: 2011, Realistic real-time rendering for large-scale forest scenes, *ISVRI 2011 - IEEE International Symposium on Virtual Reality Innovations 2011, Proceedings* pp. 217–223.

Barnsley, M. F., Ervin, V., Hardin, D. and Lancaster, J.: 1986, Solution of an inverse problem for fractals and other sets, *Proceedings of the National Academy of Sciences* **83**(7), 1975–1977. [online] Available at: `http://www.pnas.org/content/83/7/1975.abstract`, last accessed 19^{th} December 2014.

Barnsley, M. and Rising, H.: 1993, *Fractals everywhere*, Academic Press, Boston.

Behrendt, S., Colditz, C., Franzke, O., Kopf, J. and Deussen, O.: 2005, Realistic real-time rendering of landscapes using billboard clouds, *Computer Graphics Forum* **24**(3), 507–516.

Bloomenthal, J.: 1985, Modeling the Mighty Maple, *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, pp. 305–311.

Boyd, C.: 2008, *DirectX 11 Compute Shader*. [online] Available at: `http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf`, last accessed 24^{th} April, 2013.

Budapest, M. T. U.: 2009, Real-time Generation of L-system Scene Models for Rendering and Interaction, **1**(212), 67–74.

Caldwell, M.: 2013, An investigation into real-time rendering of tree growth.

Chomsky, N.: 1956, Three models for the description of language, *IRE Transactions on Information Theory* **2**, 113–124.

Cohen, M. F., Deussen, O., Hiller, S. and Shade, J.: 2003, Wang Tiles for image and texture generation, *ACM Transactions on Graphics* **22**(3), 287.

Colditz, C., Coconu, L., Deussen, O. and Hege, C.: 2005, Real-time rendering of complex photorealistic landscapes using hybrid level-of-detail approaches, *Trends in Real-time Landscape Visualization and Participation: Proceedings at Anhalt University of Applied Sciences*, Herbert Wichmann, chapter 39.

de Reffye, P., Edelin, C., Françon, J., Jaeger, M. and Puech, C.: 1988, Plant models faithful to botanical structure and development, *SIGGRAPH Comput. Graph.* **22**(4), 151–158. [online] Available at: `http://doi.acm.org/10.1145/378456.378505`, last accessed 13^{th} June 2014.

Dudash, B.: 2007, Skinned instancing. [online] Available at:`http://developer.download.nvidia.com/SDK/10/direct3d/Source/SkinnedInstancing/doc/SkinnedInstancingWhitePaper.pdf`, last accessed 13^{th} October 2014.

Fan, J., Guan, X. X. and Tang, Y.: 2011, The dynamic simulator of forest evolution, *ISVRI 2011 - IEEE International Symposium on Virtual Reality Innovations 2011, Proceedings* pp. 225–229.

Godin, C. and Ferraro, P.: 2010, Quantifying the degree of self-nestedness of trees: Application to the structural analysis of plants, *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **7**(4), 688–703.

Green, S.: 2007, *GPU Physics*. [online] Available at: `http://gpgpu.org/static/s2007/slides/15-GPGPU-physics.pdf`, last accessed 23^{rd} April, 2013.

Han, T. D. and Abdelrahman, T. S.: 2011, Reducing branch divergence in GPU programs, *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-4* p. 1. [online] Available at: `http://portal.acm.org/citation.cfm?doid=1964179.1964184`, last accessed 19^{th} December 2014.

Hardwidge, B.: 2011, *Carmack: Direct3D is now better than OpenGL*. [online] Available at: `http://www.bit-tech.net/news/gaming/2011/03/11/carmack-directx-better-opengl/`, last accessed 30^{th} October, 2012.

Hart, J. C.: 1992, The object instancing paradigm for linear fractal modeling, *Proceedings of the Conference on Graphics Interface '92*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 224–231. [online]

Available at: http://dl.acm.org/citation.cfm?id=155294.155320, last accessed 13^{th} June 2014.

Huaiqing, Z. and Min, L.: 2009, Tree growth simulation method based on improved ifs algorithm, *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pp. 1–5.

Interactive Data Visualization Inc: 2012, *SpeedTree*. [online] Available at: http://www.speedtree.com/, last accessed 2^{nd} November, 2012.

Jones, S.: 2012, Introduction to dynamic parallelism, *GPU Technology Conference* . [online] Available at: http://on-demand.gputechconf.com/gtc/2012/presentations/S0338-GTC2012-CUDA-Programming-Model.pdf, last accessed 07^{th} October 2014.

Lacz, P. and Hart, J. C.: 2004, Procedural geometry synthesis on the gpu, *nation* **8**, 21.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D. and Manocha, D.: 2009, Fast bvh construction on gpus, *IN PROC. EUROGRAPHICS 09*.

Leitner, D., Klepsch, S., Bodner, G. and Schnepf, A.: 2010, A dynamic root system growth model based on l-systems, *Plant and Soil* **332**(1-2), 177–192. [online] Available at:http://dx.doi.org/10.1007/s11104-010-0284-7, last accessed 13^{th} November, 2014.

Lindenmayer, A.: 1968, Mathematical models for cellular interactions in development i. filaments with one-sided inputs, *Journal of Theoretical Biology* **18**(3), 280 – 299. [online] Available at: http://www.sciencedirect.com/science/article/pii/0022519368900799, last accessed 19^{th} December, 2014.

Lintermann, B. and Deussen, O.: 1999, Interactive modeling of plants, *IEEE Computer Graphics and Applications* **19**(1), 2–11.

Lipp, M., Wonka, P. and Wimmer, M.: 2010, Parallel generation of multiple l-systems, *Computers & Graphics* **34**(5), 585–593. [online] Available at: http://www.cg.tuwien.ac.at/research/publications/2010/LIPP-2010-PGMS/, last accessed 3^{rd} july, 2012.

Liu, J., Zhang, S., Zeng, L., Zhu, Q. and Li, S.: 2011, CUDA based Parallel Derivation of Parametric L-system, *International Journal of Digital Content Technology and its Applications* **5**(7), 80–90. [online] Available

at:http://www.aicit.org/jdcta/paper_detail.html?q=567, last accessed 16^{th} October, 2014.

Lluch, J., Camahort, E. and Vivó, R.: 2003, Procedural multiresolution for plant and tree rendering, *Proceedings of the 2nd international conference on computer graphics, virtual reality, visualisation and interaction in Africa - AFRIGRAPH '03* p. 31. [online] Available at:http://dl.acm.org/citation.cfm?id=602330.602336, last accessed 20^{th} October 2014.

Lluch, J., Camahort, E. and Vivó, R.: 2004, A dynamic root system growth model based on l-systems, *Journal of WSCG* **12**(1-3), 507–514. [online] Available at:http://wscg.zcu.cz/wscg2004/Papers_2004_Full/I47.pdf, last accessed 13^{th} November, 2014.

Mark Harris, Shubhabrata Sengupta, J. D. O.: 2007, Parallel prefix sum (scan) with cuda, *Gpu Gems 3*, first edn, Addison-Wesley Professional, chapter 39.

Mech, R. and Prusinkiewicz, P.: 1996, Visual models of plants interacting with their environment, *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* **96**, 397–410. [online] Available at:http://dl.acm.org/citation.cfm?id=237279, last accessed 20^{th} October 2014.

Micikevicius, P.: 2012, Gpu performance analysis and optimization, *GPU Technology Conference* . [online] Available at: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf, last accessed 1^{st} October 2014.

Microsoft: 2012, *Compute Shader Overview*. [online] Available at: http://msdn.microsoft.com/en-gb/library/windows/desktop/ff476331%28v=vs.85%29.aspx, last accessed 24^{th} April, 2013.

Min, L., Huai-qing, Z. and Kang-ning, L.: 2010, Research on three-dimensional simulation of tree's morphology based on tree-crown growth model, *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pp. 1–4.

Miranda, E.: 2011, *Time boxing planning: buffered moscow rules*. [online] Available at: http://dl.acm.org.libaccess.hud.ac.uk/citation.cfm?id=2047428, [Accessed 12^{th} April, 2013].

Mok, A.: 1996, Firm real-time systems, *ACM Computing Surveys* **28**.

Nguyen, H.: 2007, *Gpu Gems 3*, first edn, Addison-Wesley Professional.

Noser, H.: 2002, Lworld: an animation system based on rewriting, *10th Pacific Conference on Computer Graphics and Applications, 2002. Proceedings.* pp. 0–1.

Noser, H. and Thalmann, D.: 1996, The animation of autonomous actors based on production rules, *Proceedings Computer Animation '96* pp. 47–57.

Noser, H. and Thalmann, D.: 1999, A rule-based interactive behavioral animation system for humanoids, *IEEE Transactions on Visualization and Computer Graphics* **5**(4).

Ogre3D: 2012, *Ogre3D*. [online] Available at: `http://www.ogre3d.org/`, last accessed 2^{nd} November, 2012.

Oppenheimer, P. E.: 1986, Real time design and animation of fractal plants and trees, *SIGGRAPH Comput. Graph.* **20**(4), 55–64. [online] Available at: `http://doi.acm.org/10.1145/15886.15892`, last accessed 13^{th} June 2014.

Patney, A. and Owens, J. D.: 2008, Real-time reyes-style adaptive surface subdivision, *ACM Trans. Graph.* **27**(5), 143:1–143:8. [online] Available at: `http://doi.acm.org/10.1145/1409060.1409096`, last accessed 19^{th} December 2014.

Peng, C., Park, S. I., Cao, Y. and Tian, J.: 2011, A Real-Time System for Crowd Rendering : Parallel LOD and Texture-Preserving Approach on GPU, pp. 27–38.

Pirk, S., Niese, T., Deussen, O. and Neubert, B.: 2012, Capturing and animating the morphogenesis of polygonal tree models, *ACM Transactions on Graphics* **31**(6), 169:1–169:10.

Prusinkiewicz, P. and Hammel, M.: 1994, Language-restricted iterated function systems, koch constructions, and l-systems, *New Directions for Fractal Modeling in Computer Graphics* pp. 115–143. [online] Available at: `http://algorithmicbotany.org/papers/lrifs.sigt94.pdf`, last accessed 13^{th} June 2014.

Prusinkiewicz, P., Hammel, M. and Mjolsness, E.: 1993, Animation of Plant Development, *Conference on Computer Graphics and Interactive Techniques* **93**, 351–360.

Prusinkiewicz, P., Hanan, J., Hammel, M. and Mech, R.: 1997, L-systems : from the Theory to Visual Models of Plants, pp. 1–12.

Prusinkiewicz, P., James, M. and Měch, R.: 1994, Synthetic topiary, *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, ACM, New York, NY, USA, pp. 351–358. [online] Available at:`http://doi.acm.org/10.1145/192161.192254`, last accessed 13^{th} November, 2014.

Prusinkiewicz, P. and Lindenmayer, A.: 1990, *The Algorithmic Beauty of Plants*, Springer.

Prusinkiewicz, P., Lindenmayer, A. and Hanan, J.: 1988, Development models of herbaceous plants for computer imagery purposes, *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, ACM, New York, NY, USA, pp. 141–150. [online] Available at: `http://doi.acm.org/10.1145/54852.378503`, last accessed 19^{th} December, 2014.

Reeves, W. T. and Blau, R.: 1985, Approximate and probabilistic algorithms for shading and rendering structured particle systems, *SIGGRAPH Comput. Graph.* **19**(3), 313–322. [online] Available at: `http://doi.acm.org/10.1145/325165.325250`, last accessed 13^{th} June 2014.

Sengupta, S., Lefohn, A. and Owens, J. D.: 2006, A work-efficient step-efficient prefix sum algorithm, *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pp. D–26–27.

Soler, C., Sillion, F. X., Blaise, F. and Dereffye, P.: 2003, An efficient instantiation algorithm for simulating radiant energy transfer in plant models, *ACM Trans. Graph.* **22**(2), 204–233. [online] Available at:`http://doi.acm.org/10.1145/636886.636890`, last accessed 13^{th} November, 2014.

Springer: 2013, *Springer Book Listing*. [online] Available at: `http://www.springer.com/computer/book/978-0-387-97297-8`, last accessed 3^{rd} March, 2013.

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Peter, W. and Schmalstieg, D.: 2014, Parallel generation of architecture on the gpu, *Computer Graphics Forum* **33**(2), 73–82. [online] Available at: `http://dx.doi.org/10.1111/cgf.12312`, last accessed 19^{th} December 2014.

Thibieroz, N. and Cebenoyan, C.: 2010, Directcompute performance on dx11, *Game Developers Conference* .

University of Calgary: 2013, *Algorithmic Botany*. [online] Available at: `http://www.algorithmicbotany.org`, last accessed 8^{th} March, 2013.

Valve Linux team: 2012, *Faster Zombies!* [online] Available at: `http://blogs.valvesoftware.com/linux/faster-zombies/`, last accessed 30^{th} October 2012.

Weber, J. and Penn, J.: 1995, Creation and rendering of realistic trees, *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, ACM, New York, NY, USA, pp. 119–128. [online] Available at: `http://doi.acm.org/10.1145/218380.218427`, last accessed 13^{th} June 2014.

Woolley, C.: 2013, Gpu optimization fundamentals, *GPU Technology Conference* . [online] Available at: `https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf`, last accessed 09^{th} October 2014.

Yang, T., Huang, Z., Lin, X., Chen, J. and Ni, J.: 2007, A parallel algorithm for binary-tree-based string rewriting in L-systems, *Proceedings - 2nd International Multi-Symposiums on Computer and Computational Sciences, IMSCCS'07* pp. 245–252. [online] Available at:`http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4392608`, last accessed 20^{th} October 2014.

Zhang, S., Zhu, Q., Liu, J. and Zeng, L.: 2013, Parallel Interpretation of L-system Based on CUDA , **2**, 415–424.

Zhang, Y. K., Teboul, O., Zhang, X. P. and Deng, Q. Q.: 2007, Image based real-time and realistic forest rendering and forest growth simulation, *Proceedings - Second International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications, PMA 2006* pp. 323–327. [online] Available at:`http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4548390`, last accessed 20^{th} October 2014.

Zhou, K., Hou, Q., Wang, R. and Guo, B.: 2008, Real-time kd-tree construction on graphics hardware, *ACM Trans. Graph.* **27**(5), 126:1–126:11. [online] Available at: `http://doi.acm.org/10.1145/1409060.1409079`, last accessed 19^{th} December 2014.

Zhou, W., Tang, H. and Ji, Z.: 2013, GPU Instancing Method for Crowd Animation Based on Motion Capture Data, *Journal of Computaional Information Systems* **9**(11), 8. [online] Available at:`http://www.jofcis.com/publishedpapers/2013_9_11_4459_4467.pdf`, last accessed 13^{th} October 2014.

9 Other Reading

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D.: 2006, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Deussen, O. and Lintermann, B.: 2010, *Digital Design of Nature: Computer Generated Plants and Organics*, 1st edn, Springer Publishing Company, Incorporated.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K. and Worley, S.: 2002, *Texturing and Modeling: A Procedural Approach*, 3rd edn, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Engel, W.: 2013, *GPU PRO 4: Advanced Rendering Techniques*, 1st edn, A. K. Peters, Ltd., Natick, MA, USA.

Jibitesh Mishra, S. M.: 2007, *L-System Fractals*, Elsevier, Amsterdam, Netherlands.