Real-time Generation of Procedural Forests

Julian Kenwood
jkenwood@cs.uct.ac.za

Supervised By:

James Gain
jgain@cs.uct.ac.za

Patrick Marais
patrick@cs.uct.ac.za

DISSERTATION PRESENTED FOR THE DEGREE OF MASTER OF SCIENCE IN THE DEPARTMENT OF COMPUTER SCIENCE

Department of Computer Science

University of Cape Town

February 2013

I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own.

Abstract

The creation of 3D models for games and simulations is generally a time-consuming and labour intensive task. Forested landscapes are an important component of many large virtual environments in games and film. To create the many individual tree models required for forests requires a large numbers of artists and a great deal of time. In order to reduce modelling time procedural methods are often used. Such methods allow tree models to be created automatically and relatively quickly, albeit at potentially reduced quality. Although the process is faster than manual creation, it can still be slow and resource-intensive for large forests.

The main contribution of this work is the development of an efficient procedural generation system for creating large forests. Our system uses L-Systems, a grammar-based procedural technique, to generate each tree. We explore two approaches to accelerating the creation of large forests. First, we demonstrate performance improvements for the creation of individual trees in the forest, by reducing the computation required by the underlying L-Systems. Second, we reduce the memory overhead by sharing geometry between trees using a novel branch instancing approach.

Test results show that our scheme significantly improves the speed of forest generation over naive methods: our system is able to generate over $100,000$ trees in approximately 2 seconds, while using a modest amount of memory. With respect to improving L-System processing, one of our methods achieves a 25% speed up over traditional methods at the cost of a small amount of additional memory, while our second method manages a 99% reduction in memory at the expense of a small amount of extra processing.

Keywords

- Procedural Forest Generation
- L-Systems

Acknowledgements

I would like to thank my supervisors, James and Patrick, for all their assistance throughout this project. Without their help, especially during the final months, this project would not have been finished. Thank you and may you continue to guide other students successfully through the dungeon of postgraduate research.

To all my fellow students, thank you for making this experience far less painful than it could have been. Especially with the camaraderie and assistance, not just of a technical nature. I wish you all the best of luck with your future endeavours. To those still working hard to finish their theses, I hope that remainder of the struggle goes without incident and you finish strong.

I would like to thank my friends and family for their support over the past three years. Thank you for reminding me that I should be social some of the time and not just have my head buried in work. To people I met during my internships in the US, thank you showing me that awesome people exist everywhere in the world. Also, to the people that I worked with over the course of my internships, thank you for being patient with me and teaching me a mountain of new ideas.

Finally, thank you to my loving girlfriend. Without your help there is no way I would have finished this work. Thank you for making me work till 4AM when I needed to. Without your love and friendship I would have probably would have broken down a long time ago. Thank you so much!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The trend towards ever increasing realism in games and visual effects has placed greater pressure on both the modelling and rendering pipelines. Forests are a particular case in point; requiring the modelling of thousands of trees, each exhibiting significant geometric complexity.

The entertainment sector (movies and games) leads the demand for virtual environments, many of which contain forests. An example of a large forest can be seen in the movie Avatar[1], in which the entire planetary surface landmass is covered in trees. Besides entertainment, other uses for computer generated forests would include training in forest management and accurate simulation for regions surrounding the forest.

In spite of the complexity of forests, game designers and movie producers wish to retain as much visual fidelity as possible, while still keeping the computational costs low enough to be displayed to the user (or rendered to the movie scene) in an efficient manner. This is especially important for games, as they must also update the game world so that the user has an interactive experience. Traditional modelling fails in when creating large forests as they are too complicated for artists to create on a tree by tree basis. Traditionally artists and modellers were free to specify the position (and exact appearance) of every tree in the scene. While this does allow complete creative freedom, it is a time consuming task and limits the complexity of the forests that can be constructed. In order to reduce the amount of time required to model an object, procedural methods are often used. Procedural methods have their drawbacks, however, such as a potentially increased memory footprint once fully processed. Artists often use design tricks to decrease the size of the models while maintaining its quality. This work explores the problem of procedurally generating entire forests. In particular we investigate techniques and optimisations that enable the creation of very large forests, in the range of $10,000$ trees or more, in real-time. In the context of this research, we define real-time forest creation as only having to wait a few seconds for the entire forest to generated.

1.1 Procedural Methods

Procedural methods are algorithms that are able to produce content by automatic processes. Content is any form of media like art [70, 14] or music [12]. In the realm of com-

[1] http://www.avatarmovie.com

puter graphics, procedural methods are commonly used to create landscapes[44, 46], plants[54], buildings[42, 43, 31], cities[49] and textures[68].

Often the automatic processes take the form of a collection of rules that are applied to some small amount of input. From these rules, complex scenes may be created using only a few input parameters and can be applied to various objects such as landscapes, buildings and cities. A potential weakness of procedural methods is that the ability to fine tune the aesthetics is limited; changing parameters even slightly can have unintended side-effects on the created scene. Furthermore, although procedural content generation is faster, it results in an increase in the amount of processing required.

1.1.1 Procedural Forest Generation

Procedural Forest Generation is the creation of forests (most often trees but it can include the environment around the forest as well) by procedural methods. The creation of forests by procedural means is often a necessity as the manual modelling of vast numbers of trees is both costly and time-consuming. A forest the size of the Amazon Rainforest, for instance, contains approximately 7.5 trillion trees, which would be impossible to reproduce through manual labour alone.

So far, two major systems have been developed for the creation for forests: *EcoSys* and *SpeedTree*.

Figure 1.1: A forest rendering created using the EcoSys system[17].

EcoSys: *EcoSys*, the earliest such system, was created by Deussen et al. [17]. EcoSys is able to generate realistic looking forests, including plants and other foliage, from a relatively small amount of input, such as a heightmap of the landscape. The system allows for interactive editing of the parameters with built-in editors. Each individual plant is created using a procedural technique known as an L-System. L-Systems is a

set of match-replace rules that specify the appearance of the tree. These rules control everything from the texture and colour of the tree and its leaves to how the tree branches and how branches lengthen. L-Systems allow modellers to present a set of rules that describe a particular species of trees.

Unfortunately, the amount of data generated for a single tree in EcoSys can be as much as 10MB. A relatively small forest of 1000 trees results in a total of 10GB of data which cannot be rendered efficiently using current technology. The Amazon Rainforest, for example, requires orders of magnitude more memory than modern computers possess. In order to reduce the amount of memory required the system uses instancing. Instancing allows the system to create a single tree and use it in multiple places, which saves memory but can reduce realism. EcoSys only uses instances in cases where the resulting trees are likely to be similar. EcoSys is able to render forests interactively using points and lines, however, it cannot achieve high enough visual quality and frame rates for games even when executed on modern hardware [16].

A second issue with EcoSys is that it does not facilitate fast generation of forests making it unsuitable for use in games. For example, a forest (Figure 1.1) containing a large amount of duplicated geometry, and only 56 unique plants took approximately 50 minutes to create and roughly 8 hours to render, albeit on late-90's hardware. Even on the faster machines available today, we estimate this relatively simple forest would require at least several minutes to create and render.

Figure 1.2: A screenshot from the MMO *World of Warcraft*.[1] The red circles represent areas where the same tree model has been used. This artifact can lead to a break in game immersion.

[1]This game does not use SpeedTree.

SpeedTree: The trees that are presented in modern games are usually made with a third-party library called *SpeedTree*[1]. SpeedTree generates trees using an offline process and provides features, such as billboarding and mesh simplification, that allow efficient rendering even at a distance. These trees can be loaded in by the game and rendered in real-time. Unfortunately, SpeedTree is proprietary software, meaning that companies have to pay a license fee to use it. The modeller is given freedom about which trees to use, how many of each to use, and where to place them. However, since the library of trees that an artist works with can be small, the same tree could be repeated in an unrealistic fashion as can be seen Figure 1.2. This can break game immersion.

1.2 System

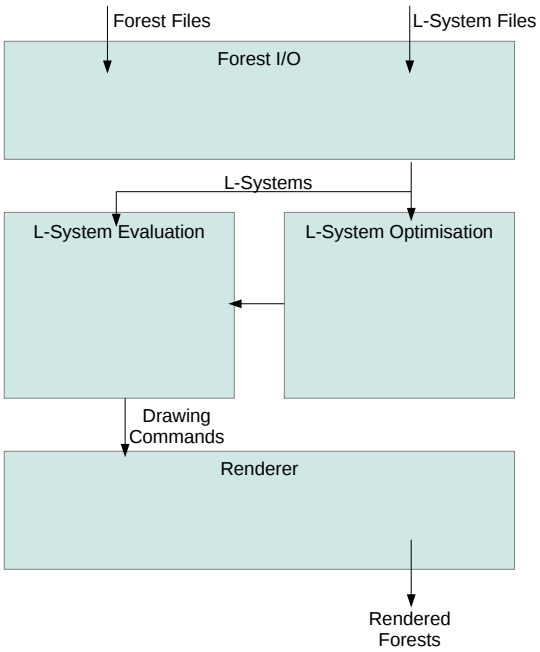


Figure 1.3: The dataflow diagram for our system, which is explained in Chapter 3.

The main purpose of this project is the development of a system for efficient *online generation* of forests. Like *EcoSys*, our system uses the L-System technique to procedurally generate individual trees in the forest. L-Systems are explained in more detail in Section 2.1.1. For now they can be described as a set of fractal-like rules that can be used to create drawing instructions for an object. L-Systems are useful and incredibly flexible in what results they can compute. which ranges from trees (of which a wide variety can be created), buildings and cities. Our system focuses only on trees as they are the chief component of forests. Constructing trees from L-Systems is a slow process that we would like to optimise. While it is possible to create realistic trees, we do not use such L-Systems. We place less emphasis on the rendering aspect of L-Systems so

[1]http://www.speedtree.com

our system uses lower quality data. It is possible, for instance, to increase the quality of the trees created in our system by supplying higher quality textures.

The system is divided up into four separate components that process necessary input from files, evaluates the L-System rules and renders the results to the screen. Finally, we provide a component that optimises L-Systems for better use with our system.

1.3 Research Questions

Our research determines whether it is possible to procedurally generate large forest ($10,000$ trees or more) in real-time (only a few seconds of generation time) on modern computers. The key success factor is whether our optimisations can effectively reduce the creation time and memory requirements to within acceptable limits.

1.4 Contribution

Our system presents two novel algorithms that process L-System more efficiently than the naïve methods. The *Automaton Writer*, Section 4.2.2, applies pattern searching algorithms to the area of L-System rule matching to achieve better theoretical run-time complexity. The *Automaton Chain Writer* 4.2.3 reduces memory requirements by introducing lazy evaluation. This modification allows trees to be generated using less than half as much memory as before. In our experiments, we find that the Automaton Writer runs 25% faster at the cost of a little extra memory. Additionally, the Automaton Chain Writer uses two orders of magnitude less memory while running only slightly slower.

We provide several algorithms that optimise L-System rules without changing their overall behaviour. We introduce the *Identity Function Optimisation*, *Orientation Optimisation*, *Growth Optimisation* and *Branch Optimisation* algorithms. These optimisations broadly attempt to improve the characteristics of the input L-System. For instance, the Branch optimisation is able to cache the geometry from a given tree branch and use it as an instance later. With this optimisation we can produce a large forest in only a few second and only modest memory requirements.

1.5 Outline

This work is divided up into four main areas. First, background knowledge about other procedural methods are discussed in Chapter 2, which includes a description of the various types of L-Systems that exist. We provide a brief overview of the system that we develop in Chapter 3. This chapter outlines the three major components of our system. These components are *L-System Evaluation*, *L-System Optimisation* and *L-System Rendering* which are discussed in Chapters 4, 5 and 6, respectively. L-System Evaluation performs the necessary work to produce usable model information from the set of rules using our novel *Automaton* and *Automaton Chain* algorithms. L-System Optimisation is an optional component that transforms provided L-Systems into a more efficient form before being used in evaluation. Finally, we end off with results and analysis in Chapter 7 and conclude the work with thesis with possible future work in Chapter 8. We test the limits of the system as the size of the created forest increases.

Chapter 2

Background

The following chapter describes the background necessary for an understanding of our forest generation algorithms. Since the most important objects in a forest are the trees and plants that constitute it, we discuss common procedural methods for trees. This is followed by an investigation of techniques for generating entire forest ecosystems.

2.1 Tree Generation

Proceduralism is a computer graphics discipline dedicated to the abstraction of form to succinct procedures and algorithms[44]. Large, detailed models and images are transformed into methods that can create the objects in question.

Procedural methods are the results of proceduralism. These are the algorithms that are able to produce content by automatic processes.

Typically, this content is media in the form of images and models but it can also take the form of music. Procedural methods, in the graphics community, are commonly used to create landscapes[44], plants[54], buildings[42, 43], cities[49] and textures[68].

Most procedural methods have several properties in common:

- Abstraction:
  Procedural methods hide details that might not be relevant to many users. These details are instead described by the algorithm itself. This layer of abstraction means that one does not have to worry about creating small details and can concentrate on large-scale appearance.

- Flexibility:
  Procedurally generated objects can be modified easily and allow quick experimentation with different parameters and settings. Manually created models do have a greater degree of freedom in the allowable modifications, but typically require more time to facilitate these modifications.

- Parameter control:
  The output of procedural methods are usually dictated by a small number of input parameters. These parameters change properties such as the roughness of terrains or the number of branches on a tree. Many procedural methods have the disadvantage that appearance is not related to parameters in a straight-forward

manner. For instance, lowering the 'hilliness' parameter of the terrain could have the unintended side effect that a mountain disappears. This makes it difficult to fine-tune output to the designer's exact requirements.

Procedural methods have several advantages over manual methods. For example, procedural methods can create extensive, rolling landscapes in a matter of seconds as opposed to weeks necessary by an artist[13, 25].

Procedural methods tend to have problems in the area of control. It is often significantly more difficult to effect minor changes. The changes tend to require a full re-run. Manual methods do not suffer from this and small modifications can be easily carried out in a 3D editor.

The procedural generation of trees is the most necessary component for creating entire forests. Generating these objects with increasingly realistic methods have been discussed by several authors. The following sections discuss various asset generation techniques used in forest environments.

2.1.1 Lindenmayer Systems

Lindenmayer Systems (L-Systems) are a technique that is used extensively in Procedural Graphics[54, 55, 22]. These rules for these systems are capable of describing complex structures such as plants and buildings yet simple enough to be created by modellers[42].

This section is divided into three areas. The first area details some of the history to L-Systems as well as some background knowledge that is relevant. The second shows the process of using standard L-Systems to create trees. Finally, the last area introduces less common L-Systems and their respective modifications and benefits.

Overview

L-Systems were first formalised in 1968 by Aristid Lindenmayer, a biologist, as an axiomatic approach to describe plant growth[32, 33]. L-Systems were later re-discovered by Przemyslaw Prusinkiewicz who realised their importance for computationally modelling plant growth .

L-Systems are a type of grammar similar to the formal grammars of Compiler Theory and Linguistics[11]. A formal (Chomsky) grammar is a collection of productions and an associated axiom. The axiom is the initial symbol and the productions are simple match-replace rules that act on symbols. Table 2.1 shows a simple grammar. These grammars form the basis of many computer languages, but with modifications may be adapted to other uses.

Two major differences exist between L-Systems and formal grammars. First, the rules in L-Systems are applied differently. In formal grammars exactly one production may be expanded at a time. In L-Systems, however, all productions that match are expanded in parallel. Figure 2.2 demonstrates this difference. The number of times this process has been repeated is called the *generation* and the set of symbols that is created is called the *string*.

Each symbol that is located in the string may be thought of as a segment of the object being modelled (bacteria, trees, and other biological objects) that still needs to grow. For this reason, parallel rewriting is more useful for organic objects as parts tend

$\rightarrow A$	*axiom*
$A \rightarrow B\ A$	
$B \rightarrow C\ B$	*productions*
$C \rightarrow C\ C$	

Table 2.1: A simple L-System grammar. The L-System contains the set of symbols to start with called the axiom. The L-System contains three rules for transforming the symbols, A, B and C into different sets of symbols.

to grow simultaneously. Each generation has the effect of growing every part of the organism by some fixed amount.

	Formal	L-System
Axiom	A	A
Generation 1	B A	B A
Generation 2	B B A	C B B A
Generation 3	B C B A	C C C B C B B A

Table 2.2: Comparison of formal grammar and L-System applications. Each underlined symbol shows the symbols that are to be expanded in the following generation. In this example, a random matching symbol is expanded for formal grammars. For L-Systems, every matching symbol is expanded.

Second, while the main purpose of formal grammars is to produce parse trees[3], L-Systems produce one dimensional strings. Parse trees, as shown in Figure 2.1, explicitly describe the structure of a program. L-Systems embed this information directly in the string, although similar structures called derivation trees are sometimes created[56]. L-System derivation trees may be thought of as the extension of parse trees to parallel grammars.

The corresponding structural information is embedded in the strings generated by L-Systems. These strings need to go through an additional process called interpretation. This step gives a physical meaning to the strings. The most common use for interpretation is the creation of geometry that may be rendered in a game or saved to disk for later use[55].

The following section describes the various classes of L-Systems and their similarities and differences in relation to the usual classes of Compiler Theory.

L-System Grammars

Several types of L-Systems exist. Their differences correspond to modifications in the grammars and interpretations techniques that are carried out. The Chomsky Hierarchy for the main sets in Compiler Theory are given in Figure 2.2. These sets represent types

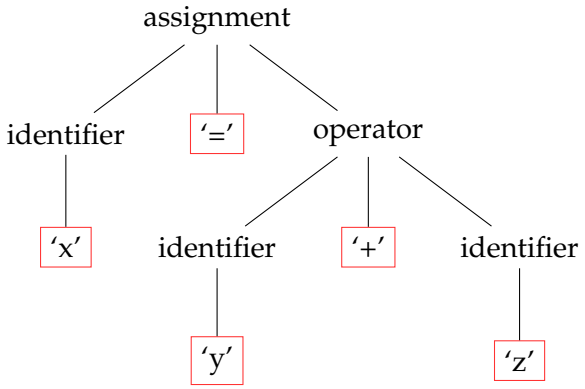


Figure 2.1: An example parse tree created for the sentence 'x = y + z'. The boxed nodes indicate symbols that were present in the sentence. The remaining nodes indicate that the corresponding texts were either identifiers, assignments or operators. Finally, the parse tree has a sense of order; an in-order walk of the tree yields the original sentence.

of grammars commonly used.

In decreasing order of ability these are: Recursively enumerable (a super set describing all grammars that can be computed), context sensitive (a set of grammars that can describe natural languages), context free (a set of grammars that are frequently used to describe programming languages), regular (used for string processing and to describe programming language) and finite (a set of grammars that can only describe finite sets).

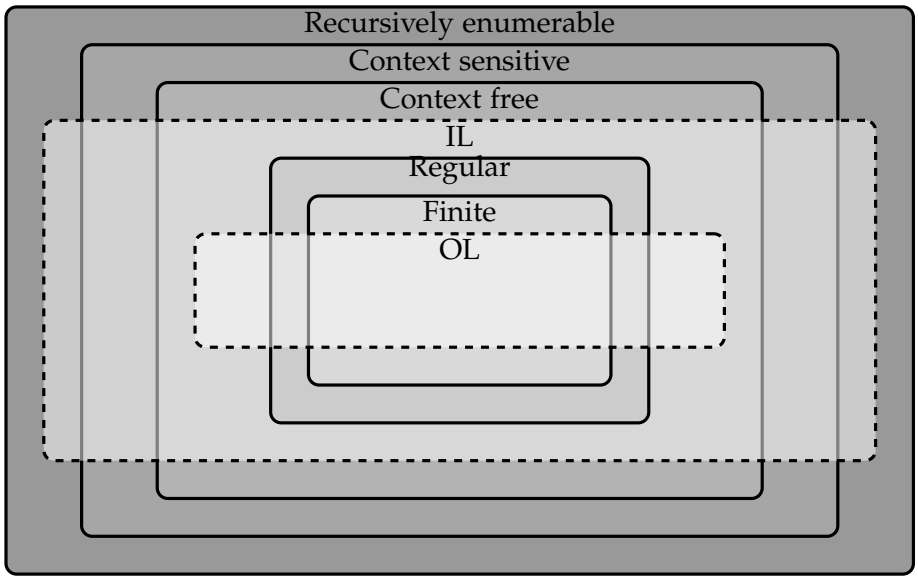


Figure 2.2: Chomsky Hierarchy including the main L-System classes. The weakest type of L-System is the OL-System, while the IL-System is significantly more powerful.'

L-Systems do not fit neatly into this hierarchy; they can be grouped into two main classes, OL and IL, which partially intersect several main classes.

The least able of the L-System classes is OL-Systems[1], or *non-interactive L-systems*[54, 60]. This class describes L-systems that use context free grammars. A context free grammar is a grammar where the left hand side of each production has empty left and right contexts. A context is a sequence of symbols on the left or right of the strict predecessor which must be matched for the rule to be applied. Table 2.3 shows the different sections of a production rule.

Various fractal curves, such as the *Koch* curve and *Dragon* curve, can be represented using a succinct OL-System grammar. The growth of various bacteria, such as *Anabaena catenula*[24], and simple trees[50], can also be simulated by an OL-System.

$$\overbrace{A\ B\ C}^{Left\ context} < \underbrace{O}_{Strict\ predecessor} > \overbrace{X\ Y\ Z}^{Right\ context} \rightarrow \underbrace{F\ F\ F\ F\ F\ F}_{Successor}$$

Table 2.3: The various parts of a production. The rule has two parts: the left- hand side, or head, and the right-hand side, or successor. The strict predecessor is a single symbol on the left-hand side that must be matched at the current position. The left and right contexts, which may be empty, indicate the symbols that must occur on either side of the strict predecessor. While the left-hand side indicates what should be matched, the right-hand side indicates what the strict predecessor should be replaced by.

When an OL-System is being applied to a string, only the current symbol (also known as the strict predecessor) being examined is used to determine which rule is applied; i.e both contexts must be empty. Otherwise, If no rule matches, the current symbol is copied over unmodified into the next generation's string. Each symbol in an OL-system in the string grows in an independent fashion.

IL-Systems, or *Interactive L-systems*, are similar to OL-Systems but without the restriction that both contexts be empty. Contexts add another layer of difficulty when it comes to applying rules. Both the strict predecessor and the contexts must match. The left context and right context must align with the preceding and following symbols, respectively.

Mathematically-focused literature further subdivides this class into 1L-Systems and 2L-Systems[54, 66]. 1L-Systems are allowed one non-empty context; 2L-Systems must have both contexts being non-empty. This distinction is typically not useful for other areas of study.

IL-Systems are a superset of OL-Systems[34]. Not only can they create the curves, bacteria and trees from OL-Systems but they are able to simulate more biologically-inspired processes.

For plants, IL-Systems are able to send information to different parts of the tree using the contexts to specify direction. Typically the left context corresponds with sending to lower (earlier grown) sections and the right context sends to higher (later grown) sections.

Table 2.4 shows a simple context-sensitive L-System grammar and the string it generates. The first production (after the axiom) rule allows a single 'B' token to be moved

[1]OL is not an acronym, the 'O' is meant to represent a biological cell.

past subsequent tokens. This extra communicative ability allows biological control signals to be simulated in an L-System.

$$\begin{aligned} &\rightarrow BAAAAA \\ B < A &\rightarrow B \\ B &\rightarrow A \end{aligned}$$

	String
Axiom	BAAAAA
Generation 1	ABAAAA
Generation 2	AABAAA
Generation 3	AAABAA
Generation 4	AAAABA
Generation 5	AAAAAB

Table 2.4: A grammar that shows the communicative abilities of IL-Systems. The 'A' symbol represents a segment of tree while the 'B' symbol represents a hypothetical signal symbol. The first rule moves the signal symbol one symbol along the string. The second rule destroy signals that are more than a single generation old. These rules together result in a signal propagation.

Although the grammar portion is significant, the next process, called *interpretation*, adds a physical aspect to L-Systems and produces useful output.

Interpretation

Interpretation is the process of adding a visual component to the output string of an L-System. Several different forms of interpretation exist; each tailored to different forms of output. For instance, L-Systems simulating bacterial growth may be better visualised using 1D-bitmaps as shown in Figure 2.3.

Since we are interested in trees, we discuss the turtle interpretation method for L-Systems. This process uses a drawing paradigm, similar to Logo's turtle[1] paradigm, to create geometry. A hypothetical 'turtle' contains various state information such as a position and orientation[55].

The interpreter represents the string as a sequence of drawing commands that are fed into the turtle from left to right. Cylinders, representing trunks or branches, are placed as the turtle moves through space to cover its path.

Several symbols exist that have specific roles in this process. The following is a list of these symbols and a description of how they work and their importance in tree creation:

- F: This symbol is used to move the turtle forward by some small amount. A cylinder is placed as it goes. All of the properties for the cylinder, such as colour and texture, are inferred from the internal state of the turtle. Often L-Systems have rules that cause the number of F symbols in the string to grow exponentially with each generation, useful for trees which grow quickly.

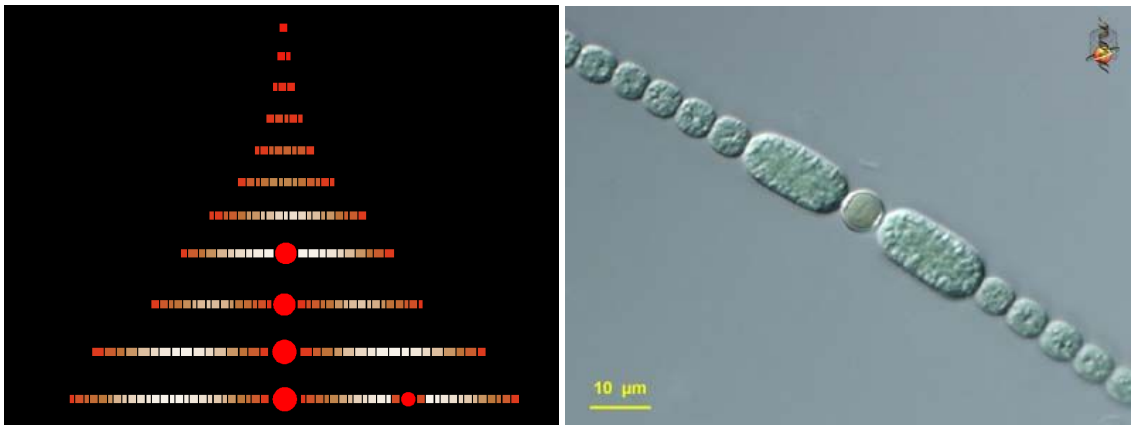


Figure 2.3: The image on the left[54] shows growth of the Anabaena Catenula bacteria simulated by an L-System. The image on the right[6] shows an actual Anabaena Catenula bacterium.

- $+, -, \&, \hat{}\,, /, \backslash, |$: These symbols are used to change the orientation of the turtle by some predefined angle around different axes. Changing the turtle's orientation is important for producing realistic trees.
- {, }, f: These symbols allows the user to draw complex polygonal shapes instead of just cylinders. When the { symbol is encountered an empty list of points is created. The f symbol now moves the turtle in the same way as F but does not place a cylinder. Instead, it adds the current position of the turtle to a list. Finally, when } is read the list of points is treated as the vertices of a polygon allowing complex shapes, such as those seen in leaves, to be drawn. Figure 2.4 shows a leaf constructed using these symbols.

Figure 2.4: A simple polygonal leaf. This example leaf is constructed from the symbols: { & f ^ f ^ f & | & f ^ f ^ f & }. Each of the 'f' symbols places a single at current position and then move the turtle forward some small amount. The & symbol and ^ (its opposite) change the orientation of the turtle to match the required leaf shape.

- [,]: The bracket symbols are important when it comes to creating branches. Productions in the L-Systems, which contain them, indicate that a branch may be constructed. The [symbol creates a copy of the turtle's state (position, orientation, etc) and pushes it onto a stack. When] is encountered it copies the state on top of the stack back to the turtle and pops the stack. Figure 2.5 shows how these symbols help isolate any state changes that were made when creating a branch.

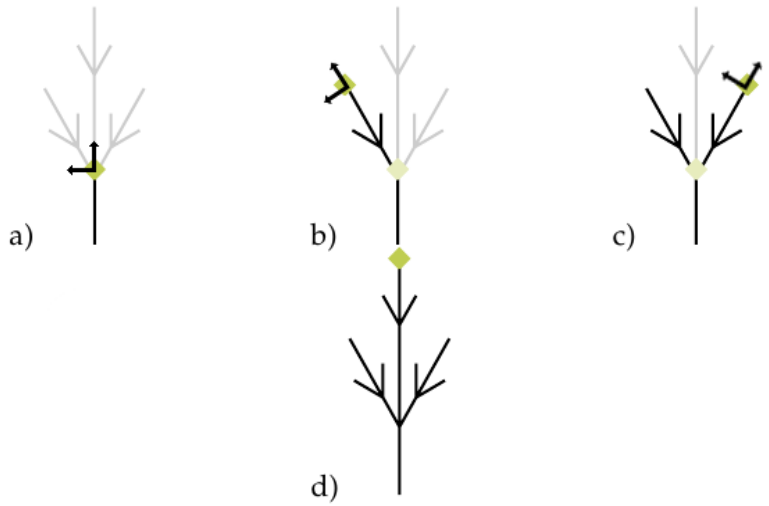


Figure 2.5: Image a) shows the tree before any branches have been made. Before the branch is constructed, shown in image b), the state is saved which is indicated by the transparent green block. After construction the state is restored so that the second branch can be made as shown in image c). Finally, the state is restored to allow the tree to grow vertically once more as shown in image d).

Once the string has been read completely the turtle is discarded and all geometry that has been generated is ready to be used by a renderer.

The L-System creator is free to define new symbols and give a new interpretation to them. Not all symbols in the input have to be consumed by the turtle. Some symbols are just used to make the grammar easier to write or more readable. These extra symbols are skipped over by the turtle.

Even though standard L-Systems are useful, improvements have arisen that change the way strings are generated and interpreted. Since these may be useful to the creation of trees they are described next.

Stochastic L-Systems

Although standard L-Systems allow one to create a large quantity of trees very quickly, they do not allow for much variation to take place. Strings can differ only when a new axiom is used or when run for a different number of generations.

Using new axioms is an uncommon and restrictive solution since they are generally given alongside the rules. This solution only delays the problem until the axioms have been depleted. Running the L-System for different numbers of generations is also problematic, since this is often linked to the maturity of the output tree.

Stochastic L-Systems solve this problem by allowing rules to have multiple successors[54, 19, 51]. When a rule is found to match, a successor is chosen at random from the right-hand side. These successors may be given different probabilities so that certain right-hand sides are more likely than others.

This is useful for trees that have a non-constant amount of branching. One could, for instance, specify several right-hand sides that express creating zero to three branches. Another example is adding symbols to change the orientation of the turtle. These

changes add both variation and realism to the output.

Figure 2.6 shows the differences between two three-dimensional trees constructed with and without non-determinism. Non-deterministic trees often exhibit jarring self-similarity.

Figure 2.6: The branching structure on the left has is purely deterministic. The branching structure on the right has been created with a stochastic L-System. The rules governing branching and growing have been given a 50% chance of occurring.

Parameterised L-Systems

An important aspect of L-Systems is achieving branch growth that is natural. This involves modelling not only branching angles and the number of branches, but also their lengths.

Modelling lengths in a standard L-System tends to be very awkward. Creating branches that grow in a pre-determined fashion (eg, linearly or exponentially) requires that the rules be specified in an intricate and non-obvious manner.

Sometimes it may be impossible to create a standard L-System that exhibits the desired behaviour. It has been proven that the growth functions achievable by standard OL-System grammars are combinations of polynomial and exponential functions[60]. It is undecidable whether a function can be represented by a standard IL-System grammar[65].

Parameterised L-Systems provides a simpler alternative to specifying growth functions directly in the rules. Each token in the original grammar is allowed to have an optional parameter list. Tokens with parameter lists are called *modules*. Typically these parameters consist of textual strings, integers or floating point numbers, but conceivably any data type is possible.

Specifying rules changes slightly in a Parameterised L-System. Modules on the left hand side of a production rule are annotated with variable names. Modules in the successors are allowed to reference the values indicated by those variable names and use them in calculations.

Each rule is also allowed a conditional guard that refers to these variables. If the conditional statement does not evaluate to true the rule is prevented from being matched further. The string creation process is made significantly more complex by this addition as can be seen in Table 2.5.

$$
\begin{aligned}
&\rightarrow A(1000) && \\
A(t) &\rightarrow A(t)\ A(t/2) && \textbf{if } \text{t mod 2} = 0 \\
A(t) &\rightarrow A(t)\ A(3 \times t + 1) && \textbf{if } \text{t mod 2} = 1
\end{aligned}
$$

Table 2.5: The above L-System produces a sequence of 'A' symbols whose parameters are the hailstone sequence starting at N = 1000. The LHS of both rules are identical except for their conditions. This means that the LHS is not necessarily guaranteed to be unique for Parameterised L-Systems which complicates rule matching significantly.

The interpretation of Parameterised L-Systems is modified to take into account the parameters of a module. The symbol name of a module is still used to determine what function should be called, but the functions are allowed to use the parameters to modify their behaviour.

For instance, feeding numerical parameters to an orientation modifying symbol allows one to specify the amount of rotation to apply. Using numerical parameters for geometry creating symbols allows a variable amount of geometry to be instantiated.

This modification unlocks a much simpler way to specify growth functions. Branching functions simply need to specify an algebraic function for growth and let the interpretation part of the L-System handle the rest. This also tends to ease memory constraints, as large strings of identical symbols are replaced by a single symbol and a number.

Parameterised L-Systems can also be used to improve the appearance of the generated data. An example of this occurs in texturing. Texture images can be changed by a simple custom module with the name of the new texture as a parameter. To facilitate this, however, one would need to add the desired functionality to the interpreter.

L-System Extensions and Other Procedural Grammars

The standard grammar-based model is not necessarily the best choice for every problem in procedural graphics. Some problems need only make minor changes to the model. This section describes common modifications that are used for specific problems.

Timed L-Systems Parameterised L-Systems allow greater flexibility over the development processes, but they are not perfect. Objects created with L-Systems can only be created in finite time steps via the generation count. Running an extra generation of an L-System might result in the addition of an entire year of growth, for instance, and it may not be possible to create an object which falls in between two generations (eg: a two and a half year old tree).

Timed L-Systems solve this problem by allowing fractional amounts of the generation count to be created[52]. In traditional L-Systems the lengths of successors are coupled to the generation in which they first occurred. For, instance the length of a branch of a tree could be exponential in the number of generations since it was created.

Timed L-Systems allow symbols to have an explicit real-valued time parameter, called the *terminal age*. The terminal age limits the time during which symbols can grow. In terms of trees, this means growing branches or creating new branches is limited. Timed symbols are created with initial time values that are synchronised to a global time for the entire L-System and which can be referenced in successors. This allows branches to grow in accordance with the tree as a whole.

Timed L-Systems allow different successor functions to be used instead of the standard L-System pattern matching method. The new methods are better able to capture various growth patterns seen in nature. However, the L-System creator has to do additional work by now defining the growth using mathematical equations. This is usually substantially easier than finding the appropriate rules to implement.

Timed L-Systems result in a continuous, rather than discrete, growth pattern for objects. This is especially useful when creating animations of growing trees as well as simulating more accurate growth. The main benefit lies in providing better control over the L-Systems output. Figure 2.7 shows a continuous growth pattern for the *Anabaena Catenula* bacterium generated using Timed L-Systems.

Figure 2.7: The growth of a Anabaena Catenula bacterium using Timed L-Systems. Each orange line in the bitmap represents a valid time in its growth[52].

Timed L-Systems, however, are significantly more complex to implement than traditional L-Systems. They require all of the features of a traditional L-System, as well as the timed features. Common implementations of timed L-Systems grow objects in small deltas requiring more work for older objects.

Shape Grammars Shape grammars are similar to ordinary L-Systems except that the grammar has been modified to work on shapes[62]. Rules in a shape grammar involve matching configurations of shapes in three-dimensional space and creating new or modifying existing geometry as a result. These operations, especially matching, are difficult, if not impossible, to efficiently implement. As a result, simpler systems are used instead.

Split grammars were developed as a way of making the problem more tractable [71]. They place restrictions on the types of allowable rules. The main operation of the grammar is the splitting of closed, convex polyhedra. The axiom of an L-System is replaced by an initial set of shapes. The matching can only be done on individual

shapes. This restriction makes it possible to design the geometry of simple buildings easily and this is the main application area.

Computer Generated Architecture (CGA) grammars are an extension of split grammars that alleviate certain issues[42]. This grammar addresses difficulties in creating certain buildings that would otherwise require intricate rules to manage or create intersecting geometry that is visually unpleasant.

The CGA grammar is a sequential grammar, similar to Chomsky grammars, with additional rules added. A further step, called façade generation[43], is emphasised for more visually pleasing aesthetics. Façades, the visible faces of a building, can be identified and marked with symbols (wall, roof, etc) in the grammar. These symbols determine what texture is applied to the façade.

CGA grammars have been used to create buildings for procedural city generation. They have been employed for urban environments, such as a historical recreation of Pompeii(Figure 2.8).

Figure 2.8: A reconstruction of Pompeii where each building has been created from a CGA shape grammar[42].

Environmentally-sensitive L-Systems Environmentally-sensitive L-Systems augment the ability of standard L-Systems by providing feedback as to the surroundings of an object[53, 48]. For example, adjacent trees in a forest can detect each other if nearby.

Environmentally-sensitive L-Systems allow the creation of special modules in a grammar, called query modules. Query modules can query certain state of the interpretation such as the position and orientation of the current turtle. The query module imports these values directly into the current string via its parameter list. With the use of conditional guards rules can be disabled based on position and orientation data.

The main application of Environmentally-sensitive L-Systems is to avoid intersecting geometry. In conventional L-Systems no attention is given to whether the L-System will collide with another tree, or even itself. Environmentally-sensitive L-Systems have also been used to create a form of digital topiary by growing trees into predefined shapes, as shown in Figure 2.9.

A deterrent to using these forms of L-Systems is that they require a small part of the interpretation to be done 'online', instead of as a final process. These extra steps leads to an overall slowdown, as they contain expensive geometric operations.

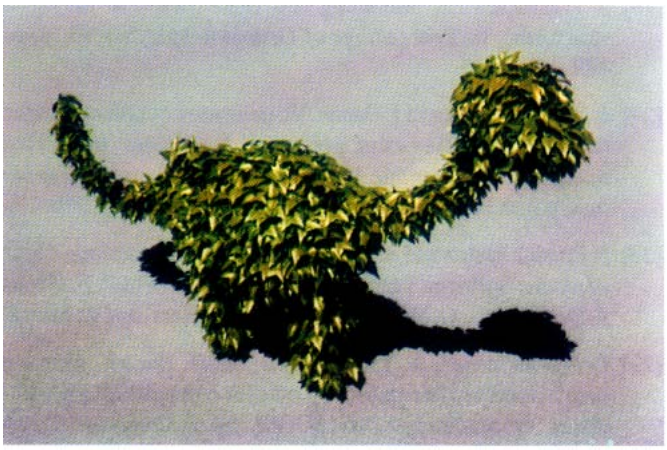


Figure 2.9: A plant grown into the shape of a dinosaur. A crude mesh of a dinosaur is used to constrain the shape of the plant. Newly constructed geometry is tested to see if it is inside the mesh and deleted upon failure[53].

Open L-Systems are a generalisation of Environmentally-sensitive L-Systems [40, 59]. Communication modules are able to query any part of the environment. An example where this would be useful is in a forest. This is useful, for example, in a forest where two nearby trees can query the positional data of each other and grow accordingly as they compete for light resources. Trees could also query their existing shape to minimise self-intersections.

While Open L-Systems allow a significant amount of flexibility when developing L-Systems, they are inherently slow as interpretations must be undertaken for all objects in the scene. The creation of such objects is much slower than simply creating a string as it may require constructing data structures to accelerate collision detection, for instance. Even worse, the L-System could require much more computationally-intensive work to be performed such as detecting the amount of sunlight that reaches a plant. This makes Open L-Systems inappropriate for online purposes.

A practical example of Open L-Systems being used is in procedural city generation[49]. Open L-Systems decide on the placement of roads based on desired parameters and rules. Closed road polygons are subdivided and labelled as plots for their interiors to be filled with buildings. The construction of the buildings is deferred to the more suitable CGA grammar. Figure 2.10 shows a procedurally generated city constructed by the software package CityEngine[1].

Summary

In this section we discussed the various L-Systems that are in common use. Standard L-Systems are a good starting point from a procedural graphics standpoint, but lack many features.

Stochastic and Parameterised L-Systems are improvements and increase the variation of objects that can be created, while only increasing complexity marginally. Additionally, Parameterised L-Systems provide greater flexibility by allowing changes to the

[1] http://www.esri.com/software/cityengine

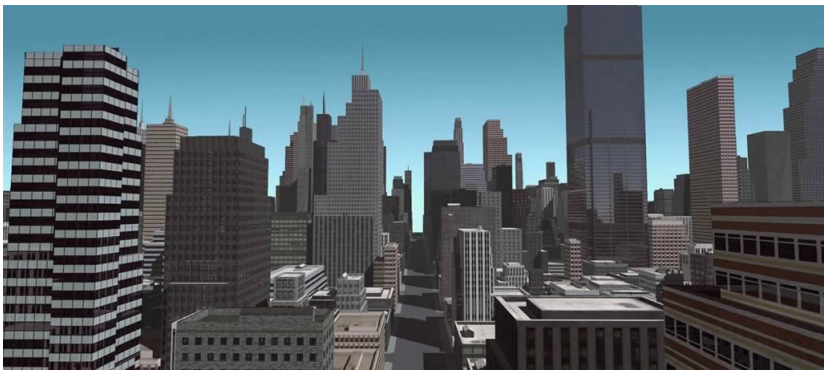


Figure 2.10: A procedurally generated New York City made by CityEngine[49].

visual aspects of the output.

Timed L-Systems provide even greater control over output but lead to high levels of complexity. Timed L-Systems provide a more continuous approach to growing trees, as opposed to the discrete methods in standard L-Systems.

Shape grammars approach the problem from the viewpoint of manipulating shapes. The technique can be very powerful if the problem being tackled is appropriate. These grammars are most commonly used for generating buildings.

Finally, Open L-Systems augment the grammar with symbols that query current state. These modifications allow objects to be grown to fit an overall shape. Open L-Systems extend this by allowing the L-System to query any part of the environment it is contained within. Although they allow a very high degree of freedom they result in significantly slower performance.

2.1.2 Semi-Procedural Tree Generation Methods

The earliest forms of procedural tree generation were not fully automated. *SynthaVision* is a system developed by research from the US military[9] and is based on a mathematical and statistical analysis of vegetation. In this system, trees are broken down into their component pieces: branch, stem or leaf. These primitives are stored in a database for future look up. Trees are manually specified on the basis of which primitives are used and where they are placed. Simple primitives (leaves) can be defined in terms of a few attributes (stem length and diameter). These primitives are stitched together in a realistic fashion based on common tree statistics to create a full tree model.

There are several drawbacks to this approach. A lot of the work is still undertaken manually: primitives are hand chosen and hand placed. Furthermore, the system has a limited set of geometry with which to specify primitives as a consequence of the lack of computing power at the time. The geometry was created for the sole purpose of ray tracing. Limitations in the created geometry were exacerbated by the lack of appropriate display technology. Figure 2.11 shows a tree created by the system, albeit in black-and-white.

Figure 2.11: A black-and-white tree image produced from SynthaVision[9]. One can see various discontinuities where the stitching process is not realistic. For instance, the 'right' branch coming off the main stem appears to be disconnected.

2.1.3 Biologically Inspired Plant Growth

Although computer-generated plants may appear to mimic real trees, many created trees are not completely realistic. Generated trees suffer from a variety of unrealistic features. Illegal branching positions, growth patterns or leaf placement are common.

Reffye et al. propose a system of creating plants and trees that more faithfully replicates real trees[15]. This system models the growth of so called *meristem* cells from known biological principles.

Meristem cells are groups of undifferentiated cells akin to stem cells in humans and other animals. These cells form the core of the plant buds. At each time step these buds are given a chance of doing several actions:

- Die: This stops the bud from growing any further.
- Flower: This causes the bud to grow a single flower or leaf at its position.
- Inflorescence: This causes the bud to create an inflorescence. An inflorescence is a cluster of flowers or leaves arranged on a stem. This has the added effect of killing the bud since no further growth is allowed.
- Internode: The bud creates an internode at its current position. The main consequence of internode creation is the subsequent creation of a node, a place where additional leaves and flowers can occur.

Each action has a certain probability of occurrence, which is a function of the age of the bud and pre-gathered tree statistics. This processes resolution is the stem, the smallest part of the tree that can be simulated. The stem's tip, called the *apical bud*, is

the main source of growth for the stem. The process is repeated multiple times, each time increasing the age of the tree by a small time step.

Branching is controlled in one of two ways: The branch can start growing immediately, or with some delay. This is chosen based on the characteristics of the plant. Specialised branching structures are catered for with rules. Branches can grow in a parallel (*orthotropy*) or in a perpendicular (*plagiotropy*) fashion.

Although these rules are able to create very realistic trees, the system as proposed has a few flaws. It is very inflexible; the code cannot easily be adapted to create trees not covered by the list of changeable parameters. To create a tree one must know all of its parameters.

The visualisation aspect of the system is also lacking as it does not cover such necessities as texturing and anti-aliasing. This is a consequence of the generation phase of the algorithm as the trees are highly detailed. The visualised output is simply a large sets of coloured lines representing geometry. Figure 2.12 shows two trees created from the system.

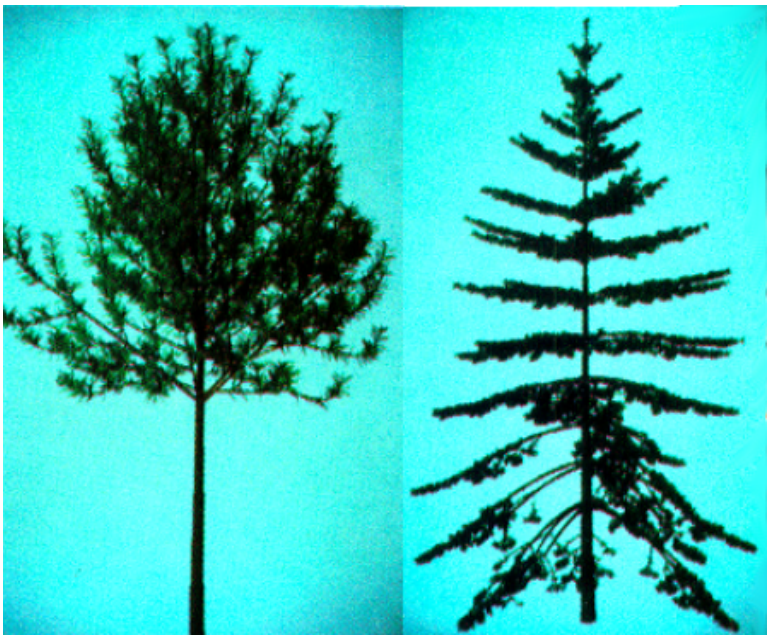


Figure 2.12: Two examples of trees created with Reffye et al.'s system. The illustration shows a pine tree on the left and a fir tree on the right[15].

Tree creation is a slow process, with reported times of a few minutes for small trees and a few tens of minutes for large trees. These times were, however, reported in 1988 and since then the increase in commodity processing power is likely to bring the figure into the realm of a few seconds.

Another method for producing realism in trees is the *A-System*[4], proposed by Aono and Kunii. A-Systems are able to create trees from a series of commands using a specialised mini-language. The language is able to explicitly describe details of the created tree; branching factors and various angles are all easily customisable. They present four different algorithms for generating trees from these systems labelled *GMT1*, *GMT2*,

GMT3 and *GMT4*. These Geometric Models for Trees are based on mimicking statistical features present in nature. Each one of these systems is thus suitable for different tree types.

GMT1 is a model for generating trees having simple characteristics. Each branch must be a bifurcation; the trunk (and subsequent branches) must split off in two separate directions. A large group of plants, *dichotomous branching plants*, fall into this category. The GMT1 model also describes the dimensions and positions of subsequent branches. Successive branches decrease in length and diameter by constant ratios. Branching angles remain constant throughout the entire tree, but two distinct angles are allowed for each branch of the bifurcation. Lastly, there is a divergence angle which applies only in special cases. If one of the branching angles is zero (or close to zero), the divergence angle is added in increasing amounts to create a spiralling effect.

The GMT2 model builds on the GMT1 model by adding attractors and inhibitors allowing the tree to be shaped. Attractors and inhibitors describe a simple force field originating at a single point in space. These points emit forces which either pull branches inwards (for instance gravity) or push branches outwards from it (for instance wind). The force is inversely proportional to the distance from the point and proportional to a 'strength' constant.

GMT3 is a model that adds ternary branching to the generation process. The bifurcation-based branching process continues, as with GMT1, but the original branch is allowed to continue growing. This gives the impression of three branches being created. The center 'branch' that continues retains the same characteristics of its other sibling branches in that the lengths and diameter contract by the same ratio.

Finally, GMT4 allows trees to have some variation. The authors use gathered statistics about the shape and branching characteristics of Japanese plants[21]. Aono and Kunii observed that the branching angles of a tree tend to decrease as the tree grows older. They developed GMT4 to test the various ways this can be accomplished via their A-System. Aono and Kunii evaluated the realism and effectiveness of each model with respect to the following criteria:

- Phyllotaxis: The arrangement of leaves or branches (phyllotaxis) on the main stem or trunk should reflect a natural plant as far as possible.
- Apical dominance: The main stem or trunk should be clearly differentiable from the other branches in the plant.
- Shape: The overall appearance of the model should resemble a natural tree.

Although Aono and Kunii's models pass each test, there are several other issues. GMT1 is able to generate a wide range of plants despite its simplicity but GMT1's main flaw is that the branching angle and diameter contraction ratios are kept constant throughout the trees development, which does not occur often in the real world.

GMT2 attempts to hide this by allowing natural (force-like) phenomenon to shape the tree's growth. This, combined with the ternary branching of GMT3, is able to produce realistic looking trees, according to the authors. Finally, Aono and Kunii note that the variation added by GMT4 does not significantly add to realism. This is more affected by the branching angles themselves than their random deviations.

Figure 2.13: The *Zelkova serrata* tree with a real tree on the left and a manually parameter-fitted tree on the right[4].

The A-System framework contains many features that are useful for artists. It can generate multiple trees and 'glue' them together to form a single tree. The visualisation aspect framework is able to produce accurate tree shadows given the position of the light source. The framework also opens up the area of parameter fitting for trees. Using photographs as a reference, parameters can be found for a given tree. Although the process remains manual, it is significantly faster than other alternatives. Figure 2.13 shows a parameter-fitted example of the *Zelkova serrata* tree. Although no timings are presented, the system is significantly more efficient than previous schemes as entire branches are grown in only a few steps.

There are a few caveats. For instance, Leaves are treated as small, opaque disks. This reduces output complexity as real leaves can be complicated curved objects. The system is rather unintuitive as many parameters need to be set in the language in order to draw trees of the correct structure and colour. Trees are only flat shaded in a single colour, while current industry practice involves realistic texturing. Finally, the system is not very flexible; trees must be forced into GMT1 to GMT4, even if this is not an appropriate fit.

2.1.4 Fractal Tree Generation

Fractals are patterns or models which exhibit some degree of *self-similarity*[38]. It has long been noted that many designs in nature exhibit self-similarity[63, 37]. However, trees lack *strict self-similarity* inherent in fractals such as the Mandelbrot set. Many trees exhibit a form called *statistical self-similarity*, branches bear only a resemblance (as opposed to being exact transformed copies) to the the tree as a whole.

An early algorithm to exploit this was proposed by Oppenheimer[47]. He uses a model that treats trees as recursive objects. A tree is defined as a piece of branch geometry with one or more trees attached. Attached trees each have associated linear transformations that describe how to orient each sub-tree. The model is only able to create strictly self-similar trees. By varying parameters (width, length, branching

angles, etc) across the levels of the tree one can create more sensible, statistically self-similar trees.

Using this method, Oppenheimer shows how to create various types of branching seen in nature. Cylindrical branches can be mimicked when the transformation is a translation and scale, while randomly changing the transformation at each step creates a curve that loops and curls in irregular ways.

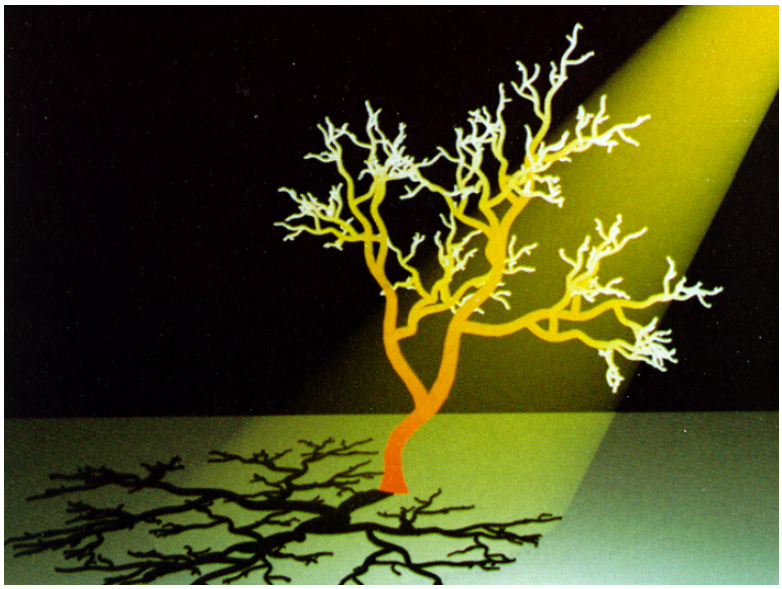


Figure 2.14: A false-coloured tree generated using a fractal approach[47]. By randomly perturbing the transformation matrix at each step, the tree can be made to have a more natural look.

Geometry created using this method is rendered in real-time using dedicated hardware as shown figure 2.14. Advanced features, such as texturing and bump-mapping are added to give the tree a bark-like surface. Each of these components are, themselves, generated using other procedural methods (specifically fractional Brownian motion[39]).

A similar approach is described in the Growth Rationale Object Work Theorem system(GROWTH) described by Kawaguchi[26]. This system is able to model the growth of not only trees but shells and corals as well. The system's main benefit is having in-built rules for modelling natural objects based on mathematical laws. Shells are modelled via a logarithmic spiral. Trees and corals are modelled by a mathematical examination to how trees branch. The resulting research is similar to the fractal system above.

2.1.5 Particle based Tree Generation

Traditionally, objects in Computer Graphics are made up of meshes of interconnected triangles. These triangles each represent a single flat surface in the object. Generally, the number of triangles grows with detail; objects with a lot of detail may need more triangles and storing the triangles' metadata can be costly.

The *Particle systems* model is a technique in Computer Graphics that can represent immensely detailed objects more compactly. Each particle is an independent sphere that has colour and other visual information but no connectivity. This helps reduce the overhead for intricate objects.

Particle systems were applied by Reeves et al.[57] to the problem of creating detailed tree geometry. As with fractal methods, a tree is described as a recursive entity. It is, however, less rigid as random parameters apply throughout. Various properties of the tree are described by the following equations:

$$
\begin{aligned}
h &= \bar{h} + rand() \times \Delta h \\
w &= h \times (\bar{w} + rand() \times \Delta w) \\
t &= t_b \times \sqrt{\frac{length - distance}{length}}
\end{aligned}
$$

The terms h and w describe the width the height of the whole tree. The $\bar{h}$, Δh, $\bar{w}$ and Δw terms are all constants that relate to each particular species of tree. Deciduous trees, for instance, have $\bar{h} = 60$, $\Delta h = 12$, $\bar{w} = 0.6$ and $\Delta w = 0.05$. The third equation describes the thickness of the branch as a function of distance to the base. As the length increases the segment will be smaller as a function of the square root of the distance.

Branching is handled by stochastically choosing a single starting point for the lowest branch. The algorithm then creates subsequent branches by choosing a random vertical displacement from the previous branch. The angle that each sub-branch forms with the tree is chosen randomly from a distribution for each species, as is the fractional diameter change. Parameters which are not updated are inherited from their parent branches.

Particles are placed on the positions chosen for branches, while solid geometry is used for the much simpler trunks. The number of particles is dependent on size; enough particles are placed to cover any gaps. A single tree could create as many as $100,000$ particles. The colours of the particles are chosen randomly from a colour table. There are separate tables for leaf colours and bark colours.

The recursion stops when a minimum branch thickness or a maximum depth is met. Leaves and needles are added to branches that have no sub-branches, to give the impression of a full tree. The trees often have a very regular structure, so environmental effects, such as wind and gravity, are added as a post-processing step.

Due to the large amount of output, stochastic algorithms are used to shade the particles. It was computationally infeasible at the time to determine which particles were in shadow and which lit. They address this issue by assigning a probability of being in shadow to each particle. This probability is associated with the depth of the particle within the tree, calculated using a metric involving the particle's position in the tree.

Rendering trees created by this algorithm is not a simple process. The particles in each tree are rendered from back to front. The main computation involves finding this ordering and this often involves sorting. With the advent of modern hardware-based depth buffers this restriction can be ignored.

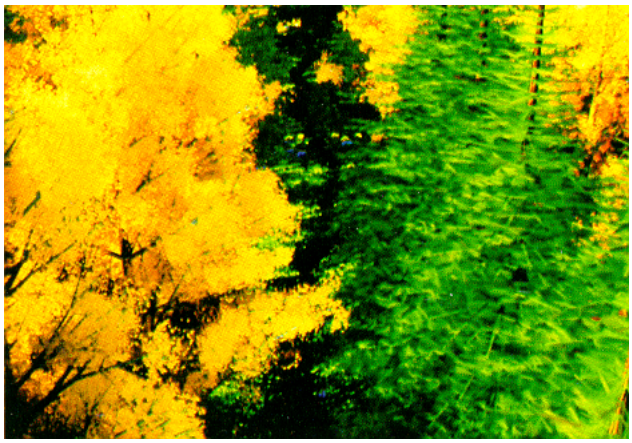


Figure 2.15: A close up of a forest scene from *The Adventures of André and Wally B*[57]. Millions of particles were required for this scene.

The algorithm produces compelling trees and was used in the movie *The Adventures of André and Wally B.*(shown in Figure 2.15) for *Lucasarts*. Although, the geometry creation part of the algorithm is intended for particles, there is little reason why it could not be adapted to work with standard triangle meshes.

2.1.6 Image-based Tree Generation

Image-based techniques use pictures and photographs to direct tree creation. Often these algorithms involve several steps, such as feature extraction, before any geometry is created. As such, they tend to be slow compared to other techniques. However, since the source tree can be, they can produce more realistic looking trees.

Shlyakhter et al.[61] introduces a technique that can create trees from multiple images. A series of between 4 and 15 photographs are taken from known poses and positions for a given tree.

Each of these images undergoes a process called *image segmentation* to separate out tree pixels from non-tree pixels. This process is manual (the authors do mention that an automatic version was tested and rejected), and a user must identify the tree in an editor.

The output of the image segmentation is a list of tree *silhouettes* or boundaries. These two dimensional silhouettes are merged to create a three dimensional *visual hull* representing an approximation of the boundary of the tree. The process used to create the hull is a projection algorithm. The silhouette is converted into a polygon with no intersecting edges. From the camera's viewpoint, the edges of the polygon are projected back through space to produce a *projection volume*. This volume is a three-dimensional representation of the all possible tree models that can create the silhouette. The process is repeated for each silhouette and their intersection is taken as an approximation of the tree. The visual hull and the silhouettes coincide when looking at hull from the appropriate position and view direction.

The visual hull is too coarse and error-prone to be used as the tree model itself so more meaningful information is extracted in the third step. The *medial axis* is de-

termined from the hull. The medial axis of a 3D volume is defined as the centres of maximal spheres that fit inside the volume. The centres form continuous curves at what can be considered the skeleton of the 3D volume. The problem of finding the medial axis of a 3D object is difficult (current solutions are slow) so an approximation algorithm is used[64].

The approximation algorithm is further modified to automatically ensure certain vertices remain in the skeleton. These vertices correspond to the tips of branches in the silhouette which are detected by computing a series of two dimensional convex hulls on the silhouettes. To reduce the running time of the medial skeleton approximation algorithm, only 30% of the interesting vertices are used. An interesting vertex is defined as a vertex at which the convex hull makes a sharp enough angle or a vertex which is adjacent to an interesting edge. An interesting edge is an edge which is long enough in proportion to total convex hull perimeter or an edge which does not closely match the underlying geometry of the tree.

The skeleton is used as an input axiom to an Open L-System, which generates additional geometry. The L-System contains rules for further growing the branches of the tree and producing leaves or flowers. Two environmental processes are then applied to grow tree: shape constraints and sunlight competition. The shape constraint process ensures that the tree does not outgrow the shape defined in the original picture. If this is detected, the offending branch is removed and replaced by a bud which later grows into leaves. The visual hull is used as a means of computationally testing whether the shape of the tree is being adhered to.

The sunlight competition process controls the placement of leaves on the tree itself. In nature, trees grow leaves so as to maximise the total sunlight received. For each branch, the L-System determines the amount of light hitting its leaves (including those further down the tree). Then a heuristic is used to determine the amount of sunlight needed to support the branch. Branches are removed if they do not receive sufficient sunlight, provided they are not significant to the structure of tree.

Finally, once the L-System has run its course the tree is coloured according to the original photographs. Appropriate colours are determined based on whether it is a leaf or trunk geometry. The system maps the three dimensional positions of the leaves to the two dimensional positions in the photographs. If the mapped position does not correspond to the tree, the geometry is deleted. The output of this final process is a model that can be rendered or saved for later use.

The system, although ground-breaking at the time, has its flaws. Multiple constraints are imposed on the images that can be used. The images must cover at least 135°, which necessitates taking many photographs. Each image must undergo several time-consuming steps to extract useful information.

Segmentation and visual hull construction can take anywhere from five to fifteen minutes per image. The remaining run-time is dominated by the Open L-System, which requires approximately forty minutes. The total running time is around four hours for between seven to fourteen images.

On the other hand, the system is able to combine the best of biologically inspired rendering and geometrical rendering. Biological and environmental aspects of tree creation are left for nature, while the modelling aspects allow fine-grain tweaking of the high-fidelity output model.

Neubert et al.[45] propose a different approach to recovering tree information from

photographic images. As input, their system also uses a set of pictures. However, the trees are automatically separated from their surroundings. The system does not need any form of exact alignment, unlike the visual hull construction of Shlyakhter et al, making it less of a burden to the user.

The first step involves extracting relevant information from each segmented image. As before, a skeleton is extracted to represent the trunk and first-order branches of the tree. The Livewire image segmentation algorithm is used to automatically extract the skeleton in a fraction of the time.

The images are then used to construct a low-fidelity voxel map. This voxel map stores the amount of *biomass* present at each relevant portion of the tree. The density values are estimated from each photograph, while least-squares and other algorithms are used to refine the estimates. The density map and the skeleton are key in producing a tree model.

The creation process uses particles to determine the placement of branches. Particles are randomly placed at positions close to the skeleton in the same distribution as the density map. For medium-sized trees between five hundred and one thousand particles are used, while larger models can use considerably more. Each particle is initialised with a uniform mass and velocity.

Particles are simulated according to the law of gravity. When two particles are considered 'close enough' they join into a single particle with an increased mass and altered momentum. The skeleton is the main source of attraction for the particles. The force field generated by the skeleton is pre-computed by using a distance transform algorithm. When particles reach the main skeleton they stop being simulated.

The paths of the particles that reach the skeleton represents the layout of a branch in space. Sometimes the chaotic nature of particle movement results in some unwanted anomalies, and to remedy this a blending function is applied to reduce artifacts. Finally, foliage is placed, in a similar fashion to the particles, according to the calculated biomass density maps.

The method uses considerably less resources and places fewer restrictions on the input images than Shlyakhter et al.'s system. A tree image can be created in only a few seconds on modern hardware as opposed to a few hours. The system does produce high fidelity trees, but less emphasis is placed on matching the tree in the photograph. For instance, a group of leaves is represented by a single flat 'billboard' with leaf instead of high quality polygonal data.

2.1.7 Other Methods for Tree Generation

Weber and Penn introduce a method for creating tree geometry with emphasis on having a wide variation of tree species[67]. Trees are specified from a single list of parameters. A list of built-in rules act to produce the tree geometry by doing different things depending on parameter values. Generalised cylinders, a simple method to model curved cylindrical geometry, are used for trunk and branch geometry. The path and cross-sections of a generalised cylinder are represented using splines resulting in greater detail and more convincing models.

In Weber and Penn, trees are described recursively using no fewer than fifty parameters. Many of the parameters simply make it possible to override the features at certain levels of the recursion. As a result, explicitly defining each parameter is not

Figure 2.16: A comparison between generated models of both systems. The left two images are made from Shlyakhter et al.'s system[61] while the right two use Neubert et al.'s system[45]. In each case the generated model is on the right.

necessary. Almost a dozen parameters control the overall shape of the tree, a single parameter controls the maximum recursion depth, and five parameters are used to control the shape of the cross-section. The positions of branches is controlled using parameters that specify their branching angle and interbranch distance.

The system supports four forms of environmental constraints: pruning, wind deflection, vertical attraction and leaf orientation. Special parameters, called shape identifiers, can be specified to ensure that leaves and branches are created in the desired manner. For instance, a value of 3 indicates that the tree's shape should be similar to that of the *Weeping Willow*. Pruning is used to ensure that the shape constraints are adhered to, and for this purpose the shape identifier is used as an index into a shape lookup table. Branches are disallowed if adding them results in the tree growing outside the specified shape. The dimensions of the shapes are determined by a further list of parameters.

Deflection via the wind is handled by treating each branch as a bendable rod with a uniform force applied to it. This problem has been well studied by physicists and several solutions are available. Vertical attraction is used to bend tree branches towards or away from the ground. Many trees grow upwards in order to best capitalise on the amount of sunlight hitting their leaves. Some trees, such as the Weeping Willow, grow upward initially and start to droop with the attraction being towards the ground. This effect is modelled using more system parameters to control the strength of attraction. In order to keep trees growing upwards initially, this environmental factor is only applied to second level branches and deeper.

Finally, the leaf orientation is optimised in a way that the faces of the leaves point upwards, towards the sun and away from the trunk. Leaves are rotated once they are created to obey this constraint.

Using these rules, the system is able to create a large range of trees from a relatively small space of parameters. Figure 2.17 shows a highly detailed *Black Tupelo* tree created using the system. Additionally, the authors provide a list of known parameters values

Figure 2.17: A highly detailed *Black Tupelo* tree created from the *Weber and Penn* system[67]. The tree is created by changing approximately twenty parameters from their defaults.

to reproduce some species of plants. In addition to the rules, the rules have an in-built notion of distance from the viewer. The system can create either high-detail models (polygonal data), low-quality models (lines and points) or a mixture of the two. This is advantageous as the system can produce trees with up to one million vertices which can put strain on even modern hardware.

A downside to the system is that is not easily modifiable. The system requires complex rules in order to function and is heavily reliant on the use of 'magic numbers' (numerical constants that are littered throughout to produce desired behaviour) and 'magic formulae.' On the other hand, the system is able to produce very natural looking objects without the user having to fine-tune parameters themselves.

Changing certain parameters can affect other parameters in subtle ways. For instance, if the **nCurveBack** parameter is set to zero, rotation occurs to the entire stem of a plant while if it is non-zero, rotation occurs for only the first half of the stem. If **nCurveBack** is negative, the stem is rotated so that it resembles a helix which is immediately obvious as to the user. Another example which produces subtle changes, is specifying the *Weeping Willow* shape identifier affects both the shape and vertical attraction of the tree. On the other hand, a user can create convincing trees without any knowledge of the botanic principles at hand.

Livny et al.[35] introduce a new representation for trees that significantly reduces space requirements. Given an input point cloud representation of a tree (such as those produced from a laser scan), the method creates a collection of shapes called lobes for the tree. A lobe is a simply-defined volume that encapsulates some part of the tree. The lobes are positioned around the skeleton (trunk and branches) to form an intermediate representation of the tree, approximating its original shape. Each lobe is then textured with an appropriate image so that a tree resembling the original is produced. The results of this process are shown in Figure 2.18.

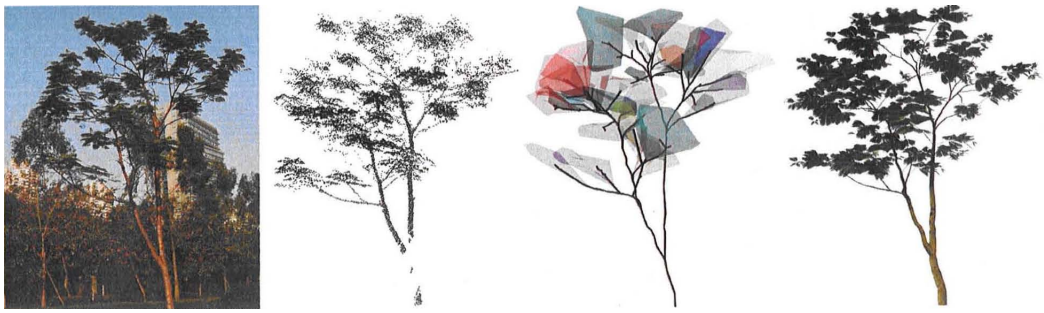


Figure 2.18: A tree created with the Livny et al's method[35]. From left to right the images represent: the original tree in its environment, the point cloud representation of the tree, the skeleton of the tree with attached lobe shapes, and the final textured tree.

Before any lobe representation can be constructed, however, the method requires a pre-generated species library. The library contains three parameters for each species: β, γ and f_s. The β parameter is related to the compactness of the tree. γ controls how the diameter of the tree decreases as its distance from the base increases. For most trees, this value is $\gamma \sim 1.5$, however it may be affected by several environmental conditions such as wind strength. Finally, the f_s parameter determines the size of the lobes; the larger f_s, the larger the lobes. Values close to 0 are more appropriate for trees with fewer leaves, while values close to 1 are used for leafy trees. Each of these parameters must be determined manually when creating the library.

Finally, the library contains graphical information about the tree species. Specifically, between twenty to thirty 'branchlets' are created and stored for each tree species. Each branchlet is textured geometry of a small branch of the tree and is created using standard L-System techniques. Each branchlet is further partitioned into incomplete branchlets which can be combined with other incomplete branchlets in order to create a new geometry without running any L-Systems. When the branchlets are partitioned, the partition points, called docking points, and their thickness and direction, are recorded in order to properly transform incomplete branchlets into a combined branchlet.

Lobe representation construction begins by determining the trunk and main branches of the tree from the point cloud using Dijkstra's algorithm on the graph induced from the point cloud. The distance between two vertices is given by $\|a - b\|^\beta$. The β parameter makes longer distances count much more than shorter distances. With larger β the tree skeleton becomes more compact. This shortest path tree becomes the basis of the skeleton of the tree.

Each point is given a diameter based on its position from the root along the shortest path tree. The diameter is given by the formula: $d(u) = d_{root}(\frac{l(u)}{l(root)})^\gamma$ where d_{root} is the diameter at the base of the tree and the function $l(u)$ returns the sum of the lengths of all the edges that are below u in the rooted tree. Similarly, $l(root)$ returns the sum of the lengths of all edges in the tree. The use of γ allows one to tailor the rate of decay of the branch diameter in the tree.

Finally, each edge is potentially marked if it is not likely to be part of the main structure of the tree. This is determined by comparing the average distances of the endpoints

of the edge against the the computed diameter of the edge (its associated branch). If the average distance is greater, the edge is labelled as low confidence. Finally, the tree is traversed from root outwards in a depth-first search fashion. If a low confidence edge is encountered, the traversal stops and continues from a different possible edge.

The set of all points and edges visited in the search are chosen to be the skeleton of the tree. The points that were not visited are grouped into lobes. The parameter f_s can be used to increase or decrease the number of lobes that are created from the algorithm. The lobes, themselves, are modelled using the α-Shapes approach of [18]. The points on the skeleton where traversal was halted due to a low confidence edge are recorded as seed points. These seed points denote where additional geometry should be placed. The seed points have two important properties: a direction vector (the direction of the connected low confidence edge), and an initial thickness (the thickness at the start of the low confidence edge).

Finally, the generated lobe volumes are textured using the L-System branchlets from the library. The branchlets are chosen based on a formula based on the thicknesses and directions of the branchlets. The algorithm is recursive in that the selected incomplete branchlets may need more branchlets to be combined together. Using the textured lobes instead of the branchlet geometry allows for a progressive level-of-detail method to be developed based on the addition or removal of lobes based on distance. Secondly, the lobes tend to be simpler to render as lots of small textured geometry (small twigs and leaves) is replaced by a single, larger texture lobe.

The results of this technique are impressive. Trees may be constructed at a rate of between 3ms per tree and 18ms per tree depending on the species. Simpler trees, such as palm trees, tend to be constructed very quickly as they need very few lobes. On the other hand, more complicated trees, such as the pine tree, require on the order of a hundred lobes resulting in lower performance. Excluding very simple tree species, the texture lobe representation requires the small amount of approximately 20kb memory per tree. This small memory footprint per tree and low construction time makes the algorithm suitable for using it to construct forests.

The method does have a limitation, however, in that it requires a point cloud of the tree as input. These point clouds are typically captured from expensive laser scanners and not only store the tree, but the entire environment. The point cloud must be manipulated (either by hand or by some automatic process) to separate out the tree points from its surroundings. Finally, the point cloud may not contain sufficient information to reconstruct the tree accurately. For instance, for trees with dense foilage the laser scan may not penetrate to the internal parts of the tree and provide incomplete information to reconstruct the skeleton of the tree. On the other hand, once the lobe representation of the tree has been successfully computed multiple different trees may be synthesised from differing branchlets.

2.2 Forest Generation

Forests represent a significantly larger challenge than the creation of individual trees. Not only are they more difficult to construct, partially owing to their size, but they are also more characteristic of natural environments. It is rare to see only a single tree in a field. A naïve approach to creating forests is simply to perform the tree creation for

each (randomly placed) tree in the forest. However, this approach ignores the interplay between the environment and the trees. As trees grows they alter soil and sunlight conditions which affects the growth of other trees.

Although little work has been done in the field of generating entire forests, Deussen et al.[17] propose a monolithic system that is able to procedurally generate entire forest scenes. The system, called *EcoSys*, gathers data about the environment in which the forest is to be placed. Terrain data, including heights, normal vectors and environmental conditions, is read in from a list of files. These conditions can include information such as soil humidity and amounts of soil nutrients. A terrain editor is included to make editing of terrains easier. The editor is able to 'paint' features and conditions onto the heightmap, distribute water according to the terrain, and model water's effects on the terrain itself.

The system then attempts to distribute plants via an iterative growth algorithm. Using a selection of plant species placed in an initially random distribution, the system attempts to simulate the whole ecosystem. At this stage, trees are modelled as disks on the height field, positioned at the plant's location. The size of each disk is dependent on the age and species of each tree and represents a tree's *ecological neighbourhood*. Tree species are annotated with different metadata detailing their preferred conditions, growth rate and number of new trees spawned per iteration. Every step of the algorithm grows each tree by a small amount, thus expanding the size of their disks. When the ecological neighbourhoods of two plants overlap, the plants are said to be competing. Biologically, this means that they are competing for the ground and light resources in the environment. The ability of a plant to fight off another plant is determined by its *vigour* which is represented by a combination of the suitability of the plant to its current conditions and its age. Plants can kill off competing plants with lower vigour scores, based on species-tailored probabilities. The result after many iterations is a list of disks representing the age and positions of plants. Recent research has improved the distribution algorithm by introducing a probability distribution function for the entire forest and deforming it with each placed tree[30].

Each individual plant model is created using an Open L-System. This L-System simulates various environmental factors, such as gravity, sunlight gathering and intersection with other trees. The models generated by this scheme are too highly detailed to be used throughout the forest scene. Indeed, a single plant specimen can take upwards of 10MB of space and the forest may contain tens of thousands of plants resulting in over 100GB of space which, until recently, was unmanageable.

Instead, Ecosys uses a method called *instancing* to solve this problem. A smaller set of exemplar plants and trees are used to represent the entire the forest. These exemplar trees are cloned throughout the forest when a model is needed. This method can be used to decrease the memory overhead by a desired amount at the cost of visual quality. Too much instancing can cause identical models to be used throughout the forest, thus ruining the overall impression.

Deussen et al. augment the instancing approach with information about plant clusters. Plants are mapped back to the set of parameters that created them. The parameters are treated as points in a high-dimensional space. The system proceeds to pick a smaller representative sample of parameters using a multi-dimensional clustering algorithm.

The resulting set of points indicates which trees to use as exemplars with the assumption that similar attributes (closer points in the parameter space) result in similar

trees. This assumption is not necessarily true, however since changing parameters even slightly may result in vastly different output. By extension, plants and groups of plants can be used as exemplars which help minimise overall running time. The technique of instancing brings the amount of data required for the forest to a more manageable size. A typical forest scene of $100,000$ plants and trees generated by EcoSys was estimated to be on the order of 200GB in size. Instancing reduced this to approximately 150MB or by a compression ratio of $1,300 : 1$. Figure 2.19 shows an image generated by EcoSys. Originally, each plant was designated a unique model but using instancing constraining memory is feasible without sacrificing visual quality. The system that we propose also uses instancing to reduce memory and computational requirements. Our system, however, using instancing to cache individual trees at the branch level. This allows our system to construct new trees out of these branches at a later time, without requiring an L-System. This aspect of our system is described in more detail in Section 3.5.4 and Chapter 6.

Figure 2.19: A scene created using EcoSys which took approximately three days to create and 75 minutes to render[17]. The largest savings from instancing were the grass tufts where 15 exemplars were used $2,577$ times and yellow flowers where 10 exemplars were used $2,751$ times. In this scene, the apple trees were all represented by the same exemplar which is presents significant saving. With instancing turned on, the number of polygons were decreased by a ratio of $16.7 : 1$.

EcoSys, as a whole, is a major development in the field of generating entire forests procedurally. The system is very modular and offers an open architecture, however, little work has been done in the field subsequently. One area of improvement for the system would be in using a more accurate simulation of the ecosystem. The system also

uses instancing at a very broad level; entire plants may be instanced which can produce jarring repetition. Finally, the system is slow, even on current hardware, owing to the sheer volume of data that must be produced.

2.3 Summary

We have discussed several methods for the procedural creation of trees and forests. The process of creating individuals trees is not an easy one. Many factors need to be taken into consideration and these are prioritised differently in different systems. Trees that convey properties that are biologically accurate can be generated by Reffye et al.'s algorithm of bud growing and Aono et al.'s A-Systems.

An earlier, more primitive, version of tree creation is the US Army's SynthaVision system. This system is able to create trees for simple simulation purposes, but is a partially manual work flow and lacks fidelity. Fractal-based tree generation is another early form of tree generation which is able to produce certain trees using fractal behaviour.

L-Systems are another long-standing technique for producing trees. The field of L-Systems is rooted in mathematics and language theory and has been studied thoroughly. L-Systems have constantly evolved to meet several challenges presented in tree creation.

The particle-based method of Reeves et al. creates trees that have a more suitable geometry for highly detail models. While particles are used in creation, modern advances in computer hardware facilitates more traditional modelling approaches, such as the more common triangle mesh.

Image-based tree generation techniques attempt to recreate trees from photographic images. These algorithms extract only partial information from the photographs before falling back on other approaches. Shlyakhter et al. use a traditional L-System, while and Neubert et al. use physically simulated particles to grow a trees outward.

Weber and Penn introduce a parameter driven approach to creating tree models. The system is sufficiently powerful that many different tree species can be modelled. The system is driven by a list of physics-oriented rules making it impossible to specify esoteric or apparently contradictory trees. Livny et al. provided a novel method for representing trees using texture lobes from their point clouds and a species library. The algorithm is efficient and can represent trees with a small memory overhead.

Finally, Deussen et al. introduces the EcoSys system. This system attempts to simulate the ecosystem of the forest as whole. Positions and ages of trees are determined by an intricate positioning algorithm that handles competition for space and resources and takes into account information about different species. The system uses an instancing approach in determining which models should be drawn at their appropriate locations.

Table 2.6 shows a breakdown of each system discussed in this system. Our system should be flexible in the species of trees it can create. It should also be compatible with real-time rendering. L-Systems fit these prerequisites, however L-Systems are not particularly fast, which makes them less suitable to the process of forest generation. Our system rectifies this by introducing novel algorithms to process L-Systems more efficiently and optimisations that are applied to the L-System rules, themselves. Like EcoSys, we use instancing to improve performance and memory issues. While EcoSys provides a realistic method for placing trees, it is a slow process and not suitable for

	Realism	Automatic	Flexibility	Performance	Rendering	Forests
L-System	average/high	yes	high	slow/average	real-time/raytracer	naïve
SynthaVision	low	semi-automatic	low	fast	raytracer	naïve
Reffye et al.	high	yes	low	average	real-time	naïve
Aono and Kunii	low/average	yes	average	fast	real-time	naïve
Oppenheimer	low	yes	low	fast	real-time	naïve
GROWTH	low	yes	low	fast	real-time	naïve
Reeves et al.	average	yes	average	average	real-time	naïve
Shlyakhter et al.	high	manual input	low	slow	real-time	naïve
Neubert et al.	average/high	manual input	low	fast	real-time	naïve
Livny et al.	high	manual input	average	fast	real-time	naïve
Weber and Penn	high	yes	low/average	fast	real-time	naïve
EcoSys	high	yes	average/high	slow	raytracer	realistic

Table 2.6: A comparison between each of the systems discussed in this chapter. The table scores the how realistic the results are, if manual intervention is required, how many different types of trees can be made, how fast the system is, what renderer is used and if the system deals with forest generation in a non-naïve way.

creating forests in a real-time scenario. Our system has the option of randomly placing the trees independently of each other, like the naïve, or reading the tree positions in from a file of previously generated tree positions.

Chapter 3

System Design

The aim of the project is the investigation of new forest generation methods. We use the framework of L-Systems to generate trees. As such the system that we propose is designed to test new methods for generating large forests. We intend to measure the effectiveness of our forest creation schemes in both speed and memory usage among other metrics. Each component in the system can report its memory usage and total time spent within itself. In the remainder of this chapter we discuss some of the decisions that affected our design and follow with a system design breakdown. Finally, we introduce each component in brief before they are presented in more detail in corresponding chapters.

3.1 Design and Scope Considerations

A crucial component to the procedural generation of forests is the ability to create each tree procedurally. Our L-Systems allow the rules to be stochastic, context-sensitive and parameterised. While not as powerful as Open L-Systems, they can still produce high quality trees and are considerably faster.

These L-Systems provide the ability to systematically produce trees according to a specification. Each L-System can be specified entirely in a single human-readable text file, save for the textures and other graphics necessary to render a tree properly.

A second important aspect to forest generation is the placement of trees on the terrain. Our system, however, does not perform such tasks. Placement of trees has been investigated by other systems such as *EcoSys*[17]. On the other hand, we wish to allow custom placements of trees in a forest such as those manually specified by users. The system has been designed to read in a list of tree positions and other characteristics from a file. From this file, the system creates and ultimately renders the forest.

We approach the task of producing tree models quickly from multiple angles. First, we optimise the string creation phase of L-Systems by introducing several new algorithms. While it is not necessarily the slowest part of the system, it does contribute significantly. String creation is important as it is responsible for modelling the various properties of a tree via the resultant drawing commands. We introduce two algorithms for string creation with better asymptotic performance than the standard algorithm. Our final algorithm transforms the problem of generating L-Systems into a recursive one allowing us to achieve constant memory overhead for string generation, a signif-

icant improvement. Reducing the time and memory overheads for string generation were chosen as enhancements due its prominence in creating the final model.

Next we improve upon the L-Systems themselves via a series of transformations intended to target two problems of L-Systems: exponential behaviour, and non-parameterised rules. The first, and more serious, problem is exponential behaviour. This occurs when rules are grown using rules which result in an exponential number of symbols in the output string. We have identified two types of rules that exhibit this behaviour. The first is when subbranches are created, and the second is when branches lengthen. By eliminating exponential behaviour in the system, we can improve overall performance dramatically. Of the optimisations used, the ones that specificaly tackle the exponential behaviour prove to be the most effective. We shall explain these two scenarios in chapter 5. The second problem area of L-Systems, non-parameterised rules, occurs when L-Systems rules can be simply transformed with use of modules. These two areas were chosen as they were considered to be low-hanging fruit.

Our final area of improvement in the system, involves the representation of tree models. Storing complex models in memory can be significantly taxing on the system's resources, especially memory. We create a representation of L-System tree models that is not only easy to compute but saves memory while still being easy to render for modern graphics cards. This optimisation was chosen because it allows memory usage to be tailored to the available resources.

We place less emphasis on the rendering aspect of the system. Other systems, such as *SpeedTree*[1], provide better looking and more realistic tree models which can be rendered in real-time. Our aim is not to compete with these systems, but rather test our ideas concerning L-System optimisation and forest creation. The rendering component we provide is mainly for testing purposes. Commercial renderers, such as *Blender*[2], can produce high-quality renderings. Our system may later be modified to use such tools. In our system we apply optimisations that necessitate minor changes to the rendering procedure. While changes can be implemented in other software, we prefer to use our prototype renderer.

The task of generating forests in real-time is thus tackled on two fronts. The first deals with the acceleration of individual tree creation to decrease the amount of work to compute a single tree. Our final avenue is a new representation of trees in the forest so that parts of multiple trees can be created *at the same time* and stored only once. This combined with an adjustable caching scheme to tradeoff memory and computation time against tree part overuse.

3.2 System Overview

We divide the entire system into four major components: *Forest I/O*, *L-System Evaluation*, *L-System Optimisation* and *Forest Rendering* as shown in Figure 3.1.

The Forest I/O component contains all the procedures necessary for reading in forest and L-System description files. The L-Systems that are read in can be further optimised by one of the subcomponents of the L-System Optimiser.

[1] http://www.speedtree.com
[2] http://www.blender.org

L-Systems, optimised or unoptimised, are sent to the L-System Evaluation. This component has two fundamental processes: creation of the L-System strings and subsequent conversion to tree geometry. Each of these processes occurs once per tree in the forest. Finally, the Forest Renderer displays the resulting forest.

3.3 Forest I/O

Two subcomponents are described in their entirety: Forest Input and L-System Input. Both components read settings and data from files, however at different levels.

3.3.1 Forest Input

Each forest input file is has a small header with a version number and a list of external L-System files that are referenced. The system reads each of these files in a separate L-System input process. The remainder of the file describes each individual tree, listing the position, orientation, age and which L-System to use to generate the tree. This information is stored for later use. Other parts of the system, especially the rendering and tree generation sections, require such information to function.

3.3.2 L-System Input

The L-System configuration file represents metadata about each L-System. A field links to to the rule file. A rule file contains each rule in the L-System and is kept separate to make rule parsing easier. A grammar detailing the L-System file is provided in Appendix B. The L-System file contains an identifier so that it can be uniquely addressed by the system.

Finally, the L-System file lists the methods that may be used to optimise plant creation. Typically, a single L-System file corresponds to a particular species, but by using different L-System configurations we can easily test the effectiveness of different optimisation techniques.

3.4 L-System Evaluation

The main goal of the evaluation component is to produce strings from L-System rules using the *String Creation* component. String Creation is the application of applying the L-System's production rules. String Creation creates the string, a sequence of drawing commands, for each requested plant. The string that is created is used in the *Geometry Creation* component of *L-System Rendering* by converting these drawing commands into a triangle mesh to be used for online rendering.

3.4.1 String Creation

String Creation involves large amounts of string processing (match-replace rules). While the L-System optimisations attempt to reduce the extent of this processing, we still would like it to be as fast as possible.

We implement three techniques to speed up the process: *Brute Force Writer*, *Automaton Writer* and *Automaton Chain Writer*.

Brute Force Writer

The *Brute Force Writer* refers to the typical naïve method of creating strings. This method is implemented purely for comparison purposes and no special efforts have been made to optimise the technique further than what is required.

Automaton Writer

The *Automaton Writer* uses a preprocessing technique to optimise the match-replace process. We identify the problem of matching multiple rules as one of finding multiple patterns in a larger string. In a standard implementation, having twice as many rules leads to a doubling of execution time. Running time can also be dependent on the length of these rule: if the average length of the rules is increased, the running time increases accordingly.

We use an automaton approach to string matching. A preprocess creates an automaton from the rules using an algorithm called *Aho-Corasick*[2]. This automaton is modified to handle additional features required by L-Systems. The use of this automaton eliminates the execution time's dependence on the number of rules **and** the average rule length.

Automaton Chain Writer

There are further problems during string creation pertaining to resource management. String creation is an iterative process, the string that is output at generation N becomes the input for generation $N + 1$. This means that output from two steps has to be stored in memory.

We solve this problem by building upon our Automaton Writer with the *Automaton Chain Writer*. The Automaton Chain Writer transforms the problem of creating strings into a recursive approach. We create output for every generation in parallel, by attaching the automatons in a chain as shown 3.4.

The output from one generation is associated with the input to the next. When a generation requires input, it queries up the chain for more output. The benefit of this approach is that the buffer at each generation need only be of constant size, thus, limiting memory overhead.

3.5 L-System Optimisation

Our system approaches the problem of accelerating forest creation using two strategies. The first strategy is L-System Optimisation which seeks to automatically modify user-specified L-Systems to be more efficient. To this end, we implement four novel L-System optimisation techniques for forest generation. These are the *Implicit Function Optimisation*, *Orientation Optimisation*, *Growth Optimisation* and *Branch Optimisation*.

The first optimisation method attempts to address a problem in L-System String creation that, in some circumstances, requires forward knowledge. Our second optimisation helps improves on an issue in both String Creation and Geometry creation. The final two optimisations are directed at limiting the exponential nature of L-Systems.

3.5.1 Identity Function Optimisation

The *Identity Function Optimiser* is a component that adds extra rules to the L-System. Traditionally, L-System rules are simple match-replace methods. However, in the case where a rule fails to match it must be carried through to the next generation's string *unmodified*.

This optimisation inspects the existing rules in the L-System. The system makes a list of the possible states (in terms of possible strict predecessor and contexts) that L-System could exist in. This optimisation adds *identity rules*, that denote that a symbol will not change from generation to the next.

The benefit is that each possible state in the L-System will match exactly one rule, decreasing the complexity of the L-System String Creation process.

3.5.2 Orientation Optimisation

The *Orientation Optimisation* modifies existing rules in the L-System that contain orientation changes. Instances of these rules include when branching occurs and orientation symbols are used to move in the intended direction, and when the L-System draws a leaf and orientation symbols are used to trace the outline. Below shows a segment of an L-System string that contains many orientation changes (/ symbols).

$$[\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /\ '\ [\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /\ /\ /\ '\ [\ \&\ F\ L\ !\ A\ ]$$

The cost of using these symbols is incurred not only during string creation, but during the Geometry Creation process as well; each symbol requires conversion to a matrix and subsequent matrix multiplication. We optimise this by converting each symbol into its own matrix *before* any string creation. Additionally, consecutive orientation symbols are concatenated into a single matrix as a preprocess. Below shows the result of of using orientation optimisation to the rules. The large runs of orientation symbols have replaced by a rotateMatrix module.

$$[\ \&\ F\ L\ !\ A\ ]\ \text{rotateMatrix}(\ldots)\ '\ [\ \&\ F\ L\ !\ A\ ]\ \text{rotateMatrix}(\ldots)\ '\ [\ \&\ F\ L\ !\ A\ ]$$

This optimisation can shorten appropriate rules from many thousands of symbols to less than a dozen, while expensive matrix multiplications are moved to as early in the process as possible.

3.5.3 Growth Optimisation

A problem that occurs in L-System is that branches and trunks can potentially grow exponentially in length. This effect is necessary to make older branches grow faster and look larger than new branches.

Geometry created naïvely from this process exhibits too much detail in areas that do not require it, Figure 3.5. Second, final strings (and any intermediate strings) are also exponential in length and slowing down the entire process significantly. There is usually a one-to-one correspondence between the amount of geometry and the length of the final string.

The *Growth Optimisation* attempts to solve the problem by instead modelling the growth of certain rules. Rules that correspond to growth are indicated by: what type of growth they exhibit, which symbol is growing, and at which generation it first started growing. These three pieces of information allow us to reconstruct the corresponding parts of the final string without having to generate all intermediate strings. The second phase of the optimisation deals with the geometry aspect, limiting detail when little is required. The optimisation is equipped with heuristics to detect two types of growth patterns: exponential growth and binomial growth. In the case where these patterns are not detected, the optimisation has no effect. This optimisation is likely to be less effective for parametric L-Systems since the exponential behaviour has already been encapsulated in modules.

3.5.4 Branch Optimisation

The final optimisation, the *Branch Optimisation*, deals with the second form of exponential growth in L-Systems: branching. The number of branches that grow from a single tree can be exponential, each branch can spawn multiple sub-branches. Limiting the number of branches is useful in reducing computation times and memory overheads.

We use heuristics to determine if a branch can arise from a particular rule. Extra control symbols are added to indicate when branching occurs. The number of branches are limited by using *instancing*. This optimisation allows us to introduce two types of instancing to manage resources: *Static Instancing* and *Dynamic Instancing*.

Static Instancing uses the stochastic nature of L-Systems for decision-making. By altering the branching rules to incorporate an *instancing probability*, we can create L-Systems which use varying amounts of instanced geometry. If a right-hand side is chosen that indicates an instance should be placed rather than a new branch, we can accelerate the string creation process dramatically. In addition to being faster, the instancing approach is able to use less memory and allows geometry to be shared amongst groups of trees without having the entire tree instanced. This is our most significant contribution as it allows forests to be created to a specified memory limit.

Dynamic Instancing, on the other hand, is used more as a process to create new trees even when there is not enough memory to proceed. If insufficient resources are available to create a branch, the system will replace the current branch in progress with an instance.

Static Instancing is preventative in that it attempts to limit resource used by the system, while Dynamic Instancing is curative in that it attempts to create appropriate geometry while resources are scarce.

3.6 L-System Rendering

The renderer component displays the forest using an real-time rendering process. Figure 3.6 shows a rendering of a small forest created using our system. Many features are lacking from our renderer, including, lighting, heightmapped terrain and other advanced techniques. These features are not important for this project, but may be added by future developers. Our renderer component has two subcomponents: *Geometry Creation* and *Tree Rendering*.

3.6.1 Geometry Creation

Geometry Creation involves converting the final output of String Creation into renderable geometry. We have two components to facilitate this: *Instance Caching* and *Tree Generation*.

Instance Caching

Instance Caching stores metadata about each instance that is created as a result of the Branch optimisation, specifically, where geometry is located, its size and any transformations that need to be applied. This information is all that is required to render the instance.

The cache allows instances to be retrieved by their age and type. The type is a unique identifier determined during the L-System branch optimisation phase. Branches that have similar properties (geometry, textures, etc) are given the same identifier. Matching by both age and type ensures that appropriate instances are used when requested.

The component can be used in one of two modes: *Static Instancing* and *Dynamic Instancing*. Static Instancing refers to the instancing approach provided by the Branch Optimisation. This mode allows instancing to occur with a given probability. Dynamic Instancing is used when not enough resources are available to complete construction of the current tree. The state of tree creation is rolled back to a point where the tree can be completed using pre-built instances.

Tree Generation

Tree Generation is a high-level component that delegates the majority of its works to the Instance Cache. It makes decisions on whether to create new tree geometry or whether to build trees from instances in the cache. These decisions are based on the status of a fixed-size geometry buffer. If the buffer is empty, the decision is made to create more geometry. As the buffer fills up, however, the component increases the amount of instancing that is used. This is done by informing the instancing cache that a higher instancing probability should be used. Once the buffer is completely full, the component sets the instancing mode from Static Instancing to Dynamic Instancing. This component also performs the necessary linear transformations that are required to build trees from instances of trees and branches.

3.6.2 Tree Rendering

The aim of Tree Rendering is to display trees created from the Instance Cache efficiently. The trees must be correctly re-oriented for their final location before they can be rendered. For each instance, linear transformations are applied in order to attain correct orientation.

This component interfaces with the graphics API in order to achieve maximal performance. The implementation of this module is heavily dependent on our choice of API. We chose OpenGL for the implementation, however, there is no reason a different API could not be used instead.

3.7 Summary

In this chapter, we have discussed the design of our system. The system is divided up into four sections, which handle: *Forest I/O*, *L-System Evaluation*, *L-System Optimisation* and *Forest Rendering*. This chapter described the Forest I/O component in its entirety. Subsequent chapters describe each of the three remaining components in detail.

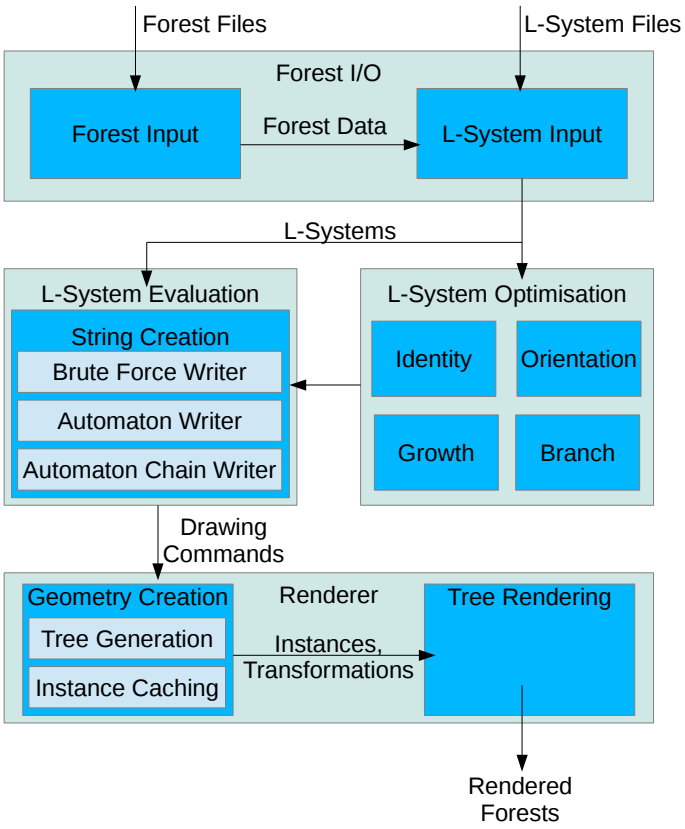


Figure 3.1: The above diagram shows a system breakdown with data flow. The system is loosely divided up into four subsystems: *Forest I/O*, *L-System Evaluation*, *L-System Optimisation* and the *Renderer*. Each of these components are described subsequently. The system reads in forest definition files and L-System files and feeds them into the L-System Evaluation system. When using L-System optimisations, the L-System Optimisation component sends optimised L-Systems to the L-System Evaluator. After evaluation, instructions for drawing trees are produced sent to the renderer which outputs rendered forests.

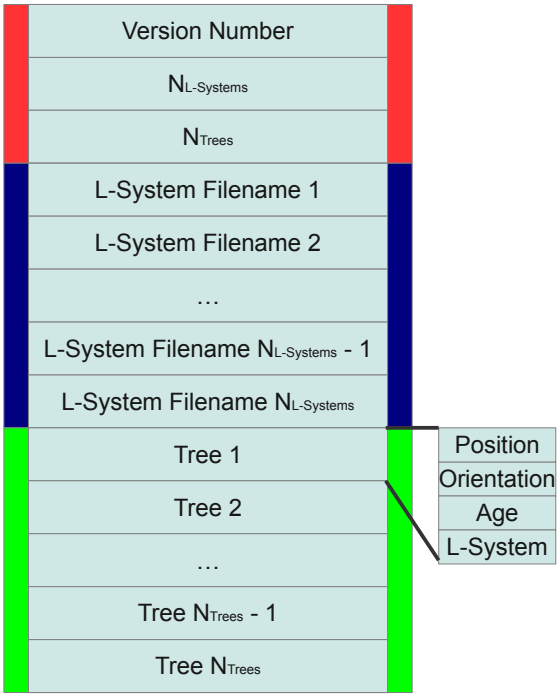


Figure 3.2: Forest file format. Red indicates file header information, dark blue represents links to the L-System filenames, and green is tree data.

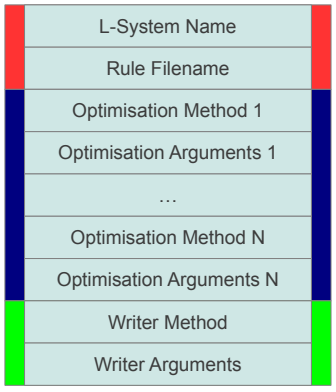


Figure 3.3: L-System file format. Red indicates information about the L-System such as rule file location and name, dark blue is optimisation flags, while the green block shows which writing method to use and with what arguments.

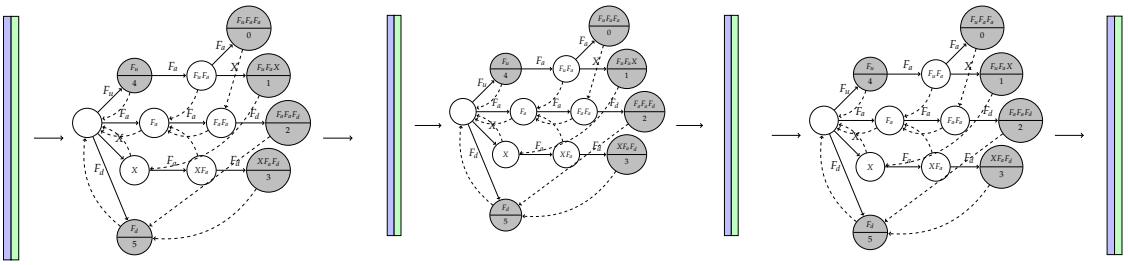


Figure 3.4: An automaton chain for three generations. Each of the automatons shown above represents a generation that we would like to generate. Buffers connect each of the automatons, enabling output from generation to be used as input for the next.

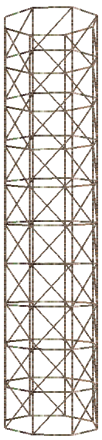


Figure 3.5: A section of an unoptimised L-System tree. The tree consists of many cylinders stacked up on top of each other. The cylinders do not curve and can thus be replaced by a single straight cylinder of equivalent dimensions.

Figure 3.6: A small forest of 100 trees created from the same L-System.

Chapter 4

L-System Evaluation

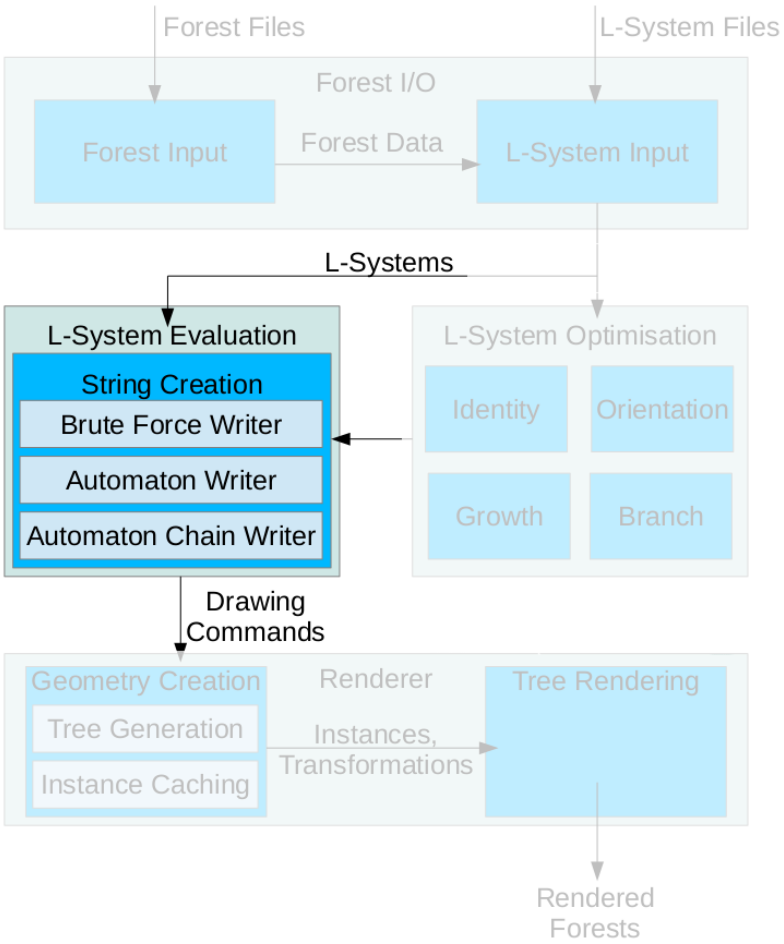


Figure 4.1: The above diagram shows the L-System Evaluation component. L-Systems from the Forest I/O and L-System Optimisation components are input to this stage. This component then creates a sequence of drawing commands in the form of L-System strings for use in the L-System Renderer.

L-System Evaluation, Figure 4.1, is the process of creating a final L-System string.

The final string represents a list of drawing instructions that can be transformed into tree geometry. We improve L-System Evaluation by focusing on two areas. First, we alter the representation of the rules in order to make common operations more efficient. Second, we investigate different algorithms for string creation. The most intensive part of string creation is the determination of which rules must be applied at each position. Our novel algorithms reduce several negative aspects of the determination leading to better performance. The results for the techniques are detailed and analysed in Chapter 7.

4.1 Rule Representation

Our system makes two changes over the naïve approach to the representation of rules. The first concerns the way in which right-hand sides are represented in the system. When using stochastic rules, a random number must be used to select one of several right-hand sides, which can be as slow as linear in the number of right-hand sides being considered. The new system requires an additional data structure to speed up the selection process. We discuss our approach in the following section, Section 4.1.1.

The second relates to argument representation in right-hand side modules. Arguments in right-hand side modules are expressions that may contain variables. These variables are aliases that refer to parameters in *already matched text*. Finding the correct value to substitute for the variable can be slow and may involve the use of several data structures. Instead of using strings to represent these aliases, the system replaces them with precomputed positional information. This allows aliases to be replaced efficiently by avoiding unnecessary work. This is discussed further in Section 4.1.2.

Both of these situations appear often in parametric and stochastic tree L-Systems and by reducing overhead we accelerate these cases.

4.1.1 RHS Selection Preprocessing

Right-hand side selection is the process of determining which right-hand side replaces the strict predecessor in the output. It occurs directly after the left-hand side of a rule has been matched.

Each right-hand side in a stochastic rule carries a probability that it will be chosen[1]. This probability is specified as a single floating-point decimal in the range [0, 1]. The sum of each right-hand side's probability in a given rule is one. The remainder of this section describes the roulette wheel selection algorithm, a common algorithm for selection, and our replacement algorithm for L-Systems.

Roulette Wheel Selection

Roulette Wheel Selection is an algorithm commonly used in Genetic Algorithms [5]. Given a list of events with associated probabilities, P_i, a roulette wheel is created whose circumference represents the range [0, 1). Each event is assigned a contiguous region of the perimeter which is proportional to its probability. For instance, an event whose

[1] Deterministic rules have an implicit probability of 1.0

probability is 50% will cover a region of the roulette which is roughly half of its circumference. Selecting an event is equivalent to selecting a point on the circumference and determining its corresponding region. This is shown in Figure 4.2.

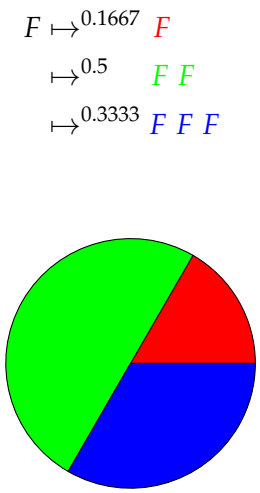


Figure 4.2: A ruleset with corresponding roulette wheel. Each right-hand side has an associated probability indicated by the superscript number. The three right-hand sides given correspond to red, green and blue areas of the roulette wheel, respectively. As with the probabilities, the areas are in the ratio $\frac{1}{6} : \frac{1}{2} : \frac{1}{3}$. The red, green and blue areas occupy the intervals $[0, \frac{1}{6})$, $[\frac{1}{6}, \frac{4}{6})$ and $[\frac{4}{6}, 1)$, respectively.

In the case of L-Systems, the events may be considered to be the right-hand sides of a particular rule. Determining exactly which right-hand side should be selected can be achieved in in linear time (to the total number of right-hand sides) by a simple method.

First, a random number in the range $[0, 1)$ is chosen, which represents a position on the circumference of the roulette wheel. A linear search is performed over the right-hand sides to determine the selection. The search stops when the right-hand of the roulette wheel containing the random position is found. Using a linear search for selection can be slow for rules with large numbers of right-hand sides, however, this selection algorithm can be accelerated by using a binary tree to store the intervals. This improves the search time from linear to logarithmic (assuming random number generation is constant time) at the cost of additional preprocessing

Array-based Selection

We present an algorithm that accelerates selection by discretising the positions around the roulette wheel. We do not claim this algorithm is novel, however, we are unable find any previously published work on the improvement. This optimisation makes it possible to select right-hand sides in constant time at the cost of higher memory usage. Usually the memory costs of the algorithm are within practical limits, however, in some cases it can quickly overwhelm the amount of memory on commodity hardware. We

also present an approximation scheme that handles the case when memory costs are too severe that reduce the amount required to manageable levels.

The first goal of the algorithm is the computation of quantised integral replacements for each right-hand side's real-valued probability. The probability with the largest number of digits after the decimal point is found. This length, T, is recorded and the following transformation is applied to each right-hand side's probability:

$$P_i = p_i \times 10^T$$

where P_i and p_i are the new and old probabilities of each right-hand side, respectively. This process increases the size of the roulette wheel to $[0, 10^T)$. The boundaries of each contiguous section of the roulette wheel now occur at integral points, thereby transforming the roulette wheel to a discrete space.

An array is used to store pointers to the right-hand sides associated with each circumference point in the new roulette wheel, Figure 4.3. Using this array, we reduce the process of selection to choosing a random element in the array which can be done in constant time (assuming random number generation is constant time). Algorithm 1 describes the array construction process given each P_i. The resulting array contains pointers to each rule. The occurrence of each pointer is in the same proportion as the probabilities.

Algorithm 1 Selection array building process.

```
function buildSelectionArray(P)
    Psum = 0
    selectionArray = []
    for i = 1 → length(P)
        for p = 1 → P[i]
            selectionArray.append(i)
    return selectionArray
end function
```

The discretisation process has some negative side effects. First off there is an additional preprocessing time required to create the array. Creation of the array requires $O(10^T)$ time to create. As the rules become more complicated, the number of significant digits for right-hand side probabilities are also likely to increase. Aside from the preprocessing time, there are significant memory requirements arising from the size of the array. As the number of digits used to represent the probability of right-hand sides increases, the size of the array increases exponentially.

To reduce these overheads, we apply an approximation algorithm to *downsample* the array. Downsampling an array involves shrinking each of the intervals present in the array by a factor of S. The size of each new interval is calculated as the size of the old interval divided by S which is then rounded off to the nearest integer. The resultant array is approximately $\frac{1}{S}$ of the original, however its distribution is slightly changed. Figure 4.4 shows the effects of discretisation.

Fortunately, there are certain safe factors which are guaranteed not to change the distribution of the array. Safe factors are numbers are factors in every interval's size. For instance, with intervals of size 12, 24, and 60, the safe factors are 1, 2, 3, 4, 6, and

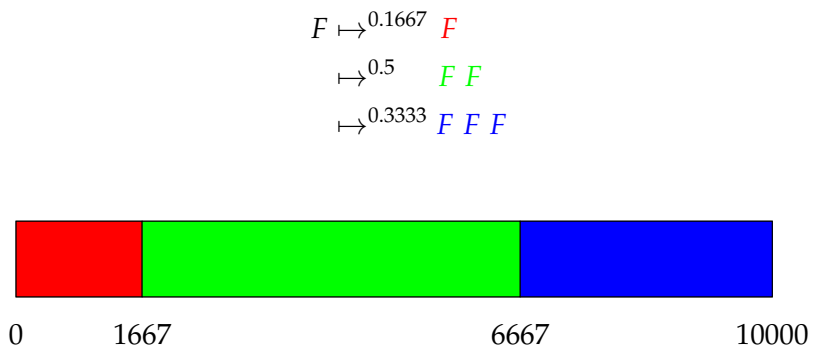


Figure 4.3: A ruleset with corresponding discretised roulette wheel. Each section represents the same interval that was presented in the original roulette wheel algorithm. With four significant digits for probabilities 1 and 3, this leads to a total array size of 10,000. The number of pointers to the red, green and blue right-hand sides are in the ratio 1667 : 5000 : 3333.

12. The maximal safe factor may be computed by finding the *greatest common divisor* of the sizes[27], in this case 12. The greatest common divisor is not guaranteed to be large and could, in fact, be 1.

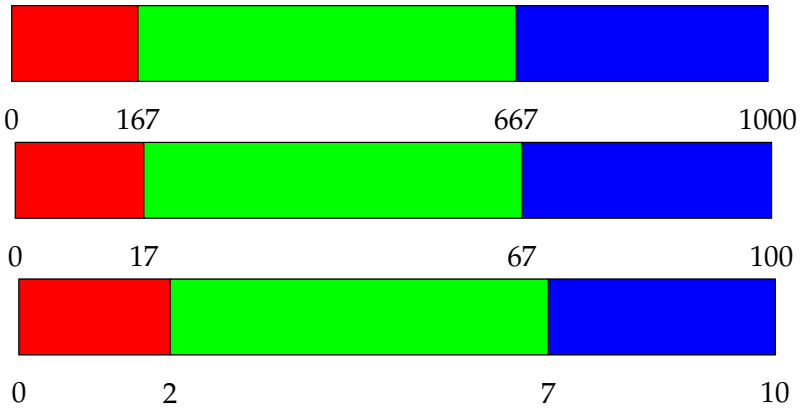


Figure 4.4: A discretised roulette wheel with shrinkage factors 10, 100 and 1000, respectively. The first two downsamplings lead to distribution errors which are less than 1%. Downsampling by 1000 times, in this case, leads to intervals which are approximately 10% off in terms of relative size.

4.1.2 Argument Preprocessing

After a left-hand side has been matched and a right-hand side selected, a process called *argument evaluation* is performed. Argument evaluation converts each of the argument expressions in a given right-hand sides into evaluated results.

The example rule that will be used throughout this section is shown in Table 4.1.

$$F(a,b)\ F < F >\ F(x,y,z) \mapsto F(a \times x + b \times y)\ F(0)\ F(z+1)$$

Table 4.1: The example L-System rule that we will use for this section.

$$\ldots\ F(3)\ F\ F\ F(2,0)\ F \underbrace{\quad F \quad}_{\text{symbol pointer}} F(6,0,10)\ F\ F\ F(3,2,7,-1)\ \ldots$$

Table 4.2: A selection of the current input string that will be used. The symbol pointer indicates that the current position of the symbol pointer.

The current state of the string is shown in Table 4.2. For the purposes of the example, the rule has already been matched at the current position. The argument evaluation process must evaluate each expression, $a \times x + b \times y$, 0, and $z + 1$. To fully evaluate an expression, the values of each variable, must be determined and substituted into each variable occurrence. A variable's value can be determined by looking at the match position in the current string. The left-hand side of the rule will *bind* values in a matched module to a unique variable name. For instance, the variable name a will be bound to value of the first argument in the first F, or 2. Similarly, b, x, y and z will be bound to the values 0, 6, 0 and 10, respectively. After binding, the variable occurrences in expressions can be *replaced* by their values.

We present two algorithms for performing argument evaluation. The first, Section 4.1.2, is a standard algorithm that performs binding and replacement every time an expression is evaluated. The second algorithm, Section 4.1.2, preprocesses L-System expressions so that the entire binding process can be ignored.

Before discussing the algorithms, however, we briefly describe two implementation details of the system when dealing with expressions. First, every expression is converted into *reverse-polish notation*[10] before any L-System evaluation has occurred. This conversion makes expression evaluation significantly easier. Second, two buffers are required during string creation per generation. One buffer stores a stream of string symbols, while the other stores a stream of parameter values. As the input string buffer is examined, two pointers track the current symbol and current parameter. An auxiliary lookup table stores the number of parameters from a given symbol id. With this mapping, one can move backwards and forwards through both buffers in lock step as shown in Figure 4.5. The main benefit of separating the symbol and parameter buffers is the ability to perform random access look-ups of symbols and parameters independently of one another.

For the purposes of these two algorithms, we use Figure 4.5 as the current state in the matching process and focus on the L-System rule discussed earlier as the rule that has already been matched.

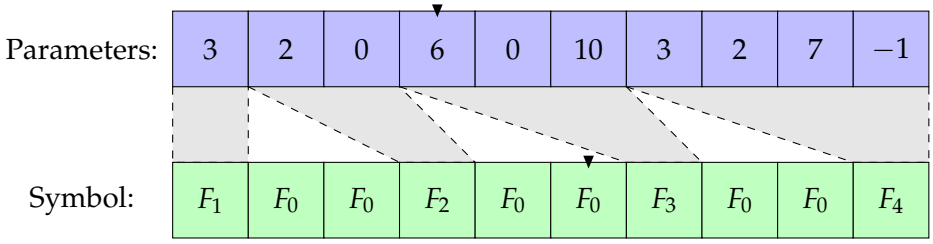


Figure 4.5: An example pair of symbol and parameter buffers. Subscripts refer to the number of parameters associated with a symbol. Shaded areas represent the mapping from symbol to its parameters. To advance by a single symbol, the symbol pointer is incremented one step and the parameter pointer is incremented by the current symbol's number of parameters. To retreat by a single symbol, the symbol pointer is decremented one step and the parameter pointer is decremented by the previous symbol's number of parameters.

Bind-and-Replace

Bind-and-Replace is a straight-forward method for replacing parameter names with their values. The algorithm operates in two phases: *binding* and *replacement* as above. During the binding phase, an empty hash table is populated to to convert from string names to values. First, the parameter position must be rewound so that it points to the very beginning of the left-context in the buffer. To perform binding, we loop through each symbol on the left-hand side in order. For each variable name in the current symbol, we update the hash table. The hash table key that is being updated corresponds to the variable name, while the associated hash table value is the value of the current parameter being pointed to. After each hash table update, the parameter pointer is advanced by a single step, ignoring the symbol pointer.

The process is shown in Algorithm 2 in the *buildValueHashTable* function which returns when every symbol on the left-hand side has been visited. The result is a hash table containing the bound variable values. In order not to affect other parts of the system, the symbol and parameter pointers are reset to their original positions. The following shows the hash table after binding has been performed for our example buffer in Figure 4.5.

$$
\begin{aligned}
H(\text{`}a') &\mapsto 2 \\
H(\text{`}b') &\mapsto 0 \\
H(\text{`}x') &\mapsto 6 \\
H(\text{`}y') &\mapsto 0 \\
H(\text{`}z') &\mapsto 10
\end{aligned}
$$

When an identifier is encountered during replacement, the hash table is queried for the name. If it exists, the identifier is replaced by the value in the hash table. If the identifier does not exist, however, it is assumed to be a callable function. If no matching function name exists, an syntax error is thrown. Once replacement has occurred, each expression is evaluated.

Algorithm 2 Variable name to value binding process.

```
function buildValueHashTable(lhs)
    savedSymbolPointer = symbolPointer
    savedParameterPointer = parameterPointer
    bindings = hashtable([])
    for s = 1 → length(lhs.leftContext)
        parameterPointer -= length(lhs.leftContext[s].parameters)
    for s = 1 → length(lhs.leftContext)
        for p = 1 → length(lhs.leftContext[s].parameters)
            key = lhs.leftContext[s].parameters[p].name
            value = parameterPointer.value
            parameterPointer += 1
            bindings[key] = value
    for p = 1 → length(lhs.strictPredecessor.parameters)
        key = lhs.strictPredecessor.parameters[p].name
        value = parameterPointer.value
        parameterPointer += 1
        bindings[key] = value
    for s = 1 → length(lhs.rightContext)
        for p = 1 → length(lhs.rightContext[s].parameters)
            key = lhs.rightContext[s].parameters[p].name
            value = parameterPointer.value
            parameterPointer += 1
            bindings[key] = value
    symbolPointer = savedSymbolPointer
    parameterPointer = savedParameterPointer
    return bindings
end function
```

The problem with this method is that all of the work must occur once per match; binding must occur every time in order to perform the other tasks. Each of these tasks may be slow, especially with very long rules. The binding process may, in fact, be done multiple times due to the effects of conditional guards, which can contain variables that need to be evaluated after binding has occurred. If the conditional guard fails, the system has performed an extra binding process without furthering the process. In this case, other rules which match the current position in the string must be tested. Table 4.3 shows an example of conditional guards causing additional bindings to occur. In the shown example, the left-hand side only contains five parameters, while in other cases many times that amount could be used.

$F(a,b)\ F < F >\ F(x,y,z) \mapsto F(a \times x + b \times y)\ F(-1)\ F(z+1)$	**if** a < b
$F(a,b)\ F < F >\ F(x,y,z) \mapsto F(a \times x + b \times y)\ F(1)\ F(z+1)$	**if** a > b
$F(a,b)\ F < F >\ F(x,y,z) \mapsto F(a \times x + b \times y)\ F(0)\ F(z+1)$	

Table 4.3: A set of rules with conditional guards. In the case of 'a = b', the binding process must occur once for each rule for a total of three times. In this example, a single bind involves inserting five variable-value pairs into the hash table.

The immediate consequence is that binding could be done multiple times per position in the string. The next method improves on this by moving much of the work to a preprocessing step.

Parameter Annotation

Parameter annotation removes the need for a run-time binding process by performing binding before L-System evaluation. The new scheme is motivated by the observation that when names are being bound to values, the *relative positions* of the values of a given rule are *constant*. For instance, given that we have matched the example rule, we know that the value for 'a' will always be two positions behind the current parameter buffer pointer, while the value for 'z' will always be two positions ahead.

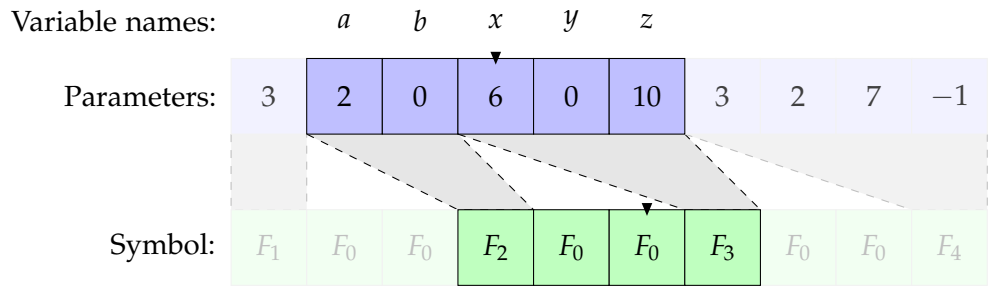


Figure 4.6: The relevant sections of both buffers for the argument evaluation process. The location of each bound variable is given by the variable name above the appropriate cell.

Algorithm 3 Variable name to relative position binding process.

```
function buildPositionHashTable(lhs)
    bindings = hashtable([])
    currentPosition = 0
    for s = 1 → length(lhs.leftContext)
        currentPosition -= length(lhs.leftContext[s].parameters)
    for s = 1 → length(lhs.leftContext)
        for p = 1 → length(lhs.leftContext[s].parameters)
            key = lhs.leftContext[s].parameters[p].name
            bindings[key] = currentPosition
            currentPosition += 1
    for p = 1 → length(lhs.strictPredecessor.parameters)
        key = lhs.strictPredecessor.parameters[p].name
        bindings[key] = currentPosition
        currentPosition += 1
    for s = 1 → length(lhs.rightContext)
        for p = 1 → length(lhs.rightContext[s].parameters)
            key = lhs.rightContext[s].parameters[p].name
            bindings[key] = currentPosition
            currentPosition += 1
    return bindings
end function
```

We can take advantage of this phenomenon by preprocessing each expression before L-System evaluation is performed. The preprocess examines each rule separately. Using a similar algorithm as before, shown in Algorithm 3, a hash table is set up to contain all the *relative positions* of each variable in a given rule. Figure 4.6 shows the relevant parts of both symbol and parameter buffers for this rule. The variable names are given above their appropriate position in the buffer. In the example above this creates a hash table whose contents are as follows:

$$\begin{aligned} H(\text{`}a') &\mapsto -2 \\ H(\text{`}b') &\mapsto -1 \\ H(\text{`}x') &\mapsto \quad 0 \\ H(\text{`}y') &\mapsto \quad 1 \\ H(\text{`}z') &\mapsto \quad 2 \end{aligned}$$

We now scan through the each expression on every right-hand side of the rule. When we encounter a variable, we make the following change to the expression. If the identifier exists in the hash table, we replace it with a built-in function called *get* which takes the relative position of the parameter in the parameter buffer as an argument. When the *get* function is encountered during run-time, it returns the value of the parameter located at the relative position to the parameter buffer pointer, or:

$$\text{get}(x) = \text{parameters}[\text{currentPosition} + x]$$

Since our system stores each buffer as a simple array, we are able to compute this result in constant time. The final expressions no longer references any variables, only their relative positions. This allows us to strip the variables names from the left-hand sides as they are no longer useful. Below is the results of transforming the example rule. We have left the variables names in the left-hand side for purpose of clarity,

$$F(a,b)\ F\ <F>\ F(x,y,z) \mapsto F(\overbrace{\text{get(-2)}}^{a} \times \overbrace{\text{get(0)}}^{x} + \overbrace{\text{get(-1)}}^{b} \times \overbrace{\text{get(1)}}^{y})\ F(0)\ F(\overbrace{\text{get(2)}}^{z})$$

The major benefit of this approach is this that we can perform parameter value lookups without the need of additional data structures or a pre-evaluation binding stage. The hash table is cleared after each expression has been examined as the data it contains is no longer necessary. The entire preprocess step occurs in proportional to the number of rules and their average length, $O(|R| \times R_{avg})$.

Summary

There are many possibilities for optimising the process further. For instance, the expressions present in L-Systems might not be the most optimal (x + x + x + x vs. 4x). Although individual speed-ups would be slight, expressions may be executed thousands if not hundreds of thousands of times. Optimising L-System expressions require a significant amount of work and may even require a large of amount computer architecture knowledge. For this reason it is left as future work.

4.2 String Creation

String creation is the process of transforming an input string to an output string. This involves applying the given L-System rules to a string up to G times, where G is the number of generations. Since the age of the tree is proportional to G and the size of the strings are often exponential functions of G, it is important that this process be as efficient as possible.

Many factors affect the time required to create a string. The number of rules and their average size both play a role but the biggest influence is the behaviour of the rules themselves. Techniques for reducing the (exponential) impact of L-System are discussed in Chapter 5. This section, however, describes various methods to increase performance of string creation without altering the rules themselves.

The first method that is discussed is the standard approach to L-System string creation. In this technique, which we call *Brute Force* (Section 4.2.1), each rule is tested at every position until a match is found. This algorithm has poor worst case behaviour so we introduce two replacement algorithms: *Automaton Writer* (Section 4.2.2) and *Automaton Chain Writer* (Section 4.2.3). Both methods rely on finite state automatons to reduce worst case performance, this accelerating the process.

Throughout this section we assume the existence of three functions: *evaluate*, *copyRHS* and *copySymbol*. The evaluate function merely returns the result of the argument expression. The function is called after a match has been found to determine the result of a conditional guard. If the result is false, the match is ignored and other rules are

tested. Although integral to string creation, the function is complicated and system specific so we leave its implementation to the reader. Our implementation uses a postfix expression evaluator and the insights discussed in the *Argument Preprocessing* section, Section 4.1.2.

The copyRHS function returns a sequence of the symbols from the given rule. First, a random right-hand side is selected using the array-based selection algorithm from Section 4.1.1. Second, all expressions that are present in the selected right-hand side are replaced by their results using the evaluate function. The resultant symbols and parameters are then returned by function. Finally, the copySymbol function is used to make a copy of the given symbol and its parameters. As a general rule for each symbol in the input string, copyRHS is used in the case of a match for a rule, while copySymbol is used otherwise.

4.2.1 Brute Force

Algorithm 4 Brute-force string creation algorithm.

```
function matchPosition(rule, inputString, position)
    if inputString[i] ≠ rule.strictPredecessor then return false
    for i = 1 → length(rule.leftContext)
        if rule.leftContext[i] ≠ inputString[position - i] then return false
    for i = 1 → length(rule.rightContext)
        if rule.rightContext[i] ≠ inputString[position + i] then return false
    return true
end function

function applyBruteforce(inputString, rules)
    for i = 1 → length(inputString)
        matched = false
        for j = 1 → length(rules)
            if matchPosition(rule[j], inputString, i) then
                if evaluate(rule[j].condition, inputString, i) then
                    output += copyRHS(rule[j])
                    matched = true
                    break
        if matched = false then output += copySymbol(inputString[i])
    return output
end function
```

The brute force algorithm is the standard technique for the L-System evaluation. Algorithm 4 shows the brute force method of evaluating a single generation of the L-System. The function *matchPosition* attempts to match a rule against a position in the string, returning true if it was successful and false otherwise. For a given rule, R, this function runs in time $O(|R_{LHS}|)$, where $|R_{LHS}|$ is the size of the left-hand side of R. This is as optimal because the function must examine each symbol in the left-context, strict predecessor and right-context exactly once in the case of a match. The logic of

the brute force approach is described in the *applyBruteforce* function which advances the input string forward by one generation. In the worst case it performs $O(|R||I|)$ calls to *matchPosition* where $|R|$ is the total number of rules and $|I|$ is the size of the input string.

The entire process runs in $O(R_{LHS_{avg}}|R||I| + |O|)$ where $R_{LHS_{avg}}$ is the average size of the left-hand side of a rule, $|R|$ is the number of rules, $|I|$ is the size of the input and $|O|$ is the size of the output. The majority of computation for a single generation is spent examining the input string. However, it is important to note that both $|I|$ and $|O|$ are dependent on the behaviour of the L-System since the output string is used as the input string in the subsequent generation. If the output size increases exponentially, the time taken to transform the next generation's input will be that much slower. On the other hand, the brute force approach requires no extra memory.

Our main concern with the brute force method is the factor of $R_{LHS_{avg}}|R|$ work performed per input symbol. The size of the input and output, themselves cannot be altered as they are decided by the L-System rules. The next method eliminates the majority of the overhead brought about by rule matching which is facilitated by processing the rules into a more efficient automaton structure.

4.2.2 Automaton Writer

The main motivation for this approach is to recognise that the rule matching part of string creation is a form of string searching. The process of string searching can be defined as finding pattern strings within a target string. There are two common approaches to efficient string searching algorithms: target preprocessing and pattern preprocessing.

Target preprocessing transforms the input string into a more optimal form for string searching. This is used when the target remains unchanged throughout and the patterns change regularly (or we do not know them in advance). There are several algorithms which fall under this category, for instance Suffix Trees[69] and Suffix Arrays[36]. In the context of L-Systems, the target corresponds to a single generation's input string and the patterns correspond to rule left-hand sides. While the target approach is applicable to our problem, this method is not desirable for two reasons. First, we do not know the input string in advance. Second, target preprocessing tends to be complicated and requires at least $O(|I|)$ time to perform. Since we do not know input strings in advance, this means that at least $O(I)$ time must be spent per generation to perform the preprocessing.

While target preprocessing transforms targets, pattern preprocessing, on the other hand, creates a data structure from the patterns which can be queried efficiently. This approach is well suited to circumstances where the target is preprocessing the target is unfavourable. This is most useful when the patterns do not change but the target does. This second approach is considerably more useful to our situation as the L-System rules do not change throughout the string creation process.

Boyer-Moore[8] and Knuth-Morris-Pratt [28] are two common algorithms which are used when only one pattern is involved. Our approach uses the Aho-Corasick[2] algorithm to transform the rules. Aho-Corasick is related to the Knuth-Morris-Pratt, however it allows multiple patterns, hence multiple rules, to be efficiently (and simultaneously) searched for. The following section explains the Aho-Corasick algorithm in

the original context of string matching.

Aho-Corasick

The Aho-Corasick algorithm creates the minimal finite automaton that matches a given set of patterns. The automaton is constructed in running time and space equivalent the size of the input patterns, or $O(\sum_{i=1}^{|P|} |P_i|)$. Once constructed, the automaton can be used to find all matches in the target in time $O(|T| + |Z|)$ where $|T|$ is the size of the target and $|Z|$ is the total number of matches. The number of of matches is bounded by $O(|T||P|)$ although in practice this is unlikely to occur.

The first stage of the Aho-Corasick algorithm is the construction of the pattern trie. A trie is a tree-like data structure. Each edge in a trie is labelled by a symbol and paths starting from the root represent items. A trie is an associative data structure that maps from strings (or string-like data in our case) to values. The key of a given value is the concatenation of the labels in the trie. Figure 4.7 shows an example trie containing English words.

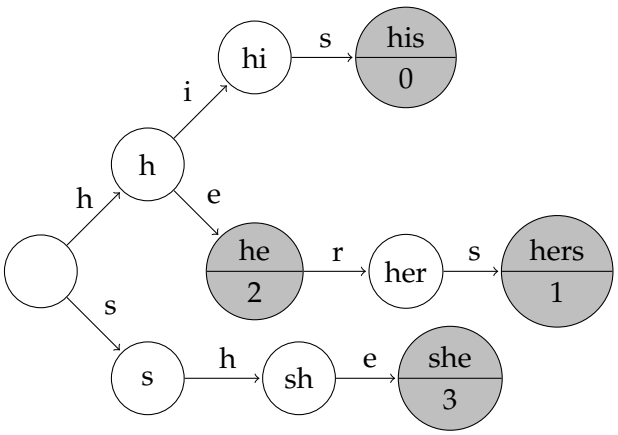


Figure 4.7: A graph representation of a Trie containing the words: his, hers, he, and she. Shaded vertices indicate that it is the end of some word and the number indicates which word. Vertex labels implicitly indicate the words or partial words contained in the trie. Edge labels indicate letters that, when added to their parent's label represent a new word.

Tries keep their data in sorted order and allow roughly constant time access to each element. They technically require linear time proportional to the number of symbols in the string for retrieval. Tries tend not to perform well with big datasets due to the large number of memory allocations and lack of spatial locality.

The traditional use of a trie is to efficiently determine if a specific string is in a set of strings. The algorithm to determine if a string exists may easily be extended to search through a string by running a separate search at each position. The possible matches from a given vertex are the nodes below the current node. When no further matches are available (the current trie node does not have an edge with the current symbol as its label), the algorithm restarts at the next position as demonstrated in Algorithm 5.

The Aho-Corasick algorithm upgrades the trie to a finite automaton by creating additional edges called *failure edges*. Failure edges allow one to run a single search from

Algorithm 5 Trie-based string searching algorithm.

```
function trieSearch(target, patternTrie)
    for i = 1 → length(target)
        node = patternTrie.root
        j = i
        for j = i → length(target)
            if not hasChild(node, target[j]) then break
            node = child(node, target[j])
            if hasOutput(node) then outputWords(node)
end function
```

the beginning of a string by determining which node is visited when no edge labels are matched. The Aho-Corasick automaton is shown in Figure 4.8.

The failure edge from a given node points to the node which is its *largest proper suffix*. A *proper suffix* of a node is a different node whose representative string is a *suffix* of the current node's string. A suffix of a string is a string obtained by deleting leading characters. The largest proper suffix is the node in the trie that can be obtained with by disregarding the fewest symbols (at least one) at the beginning of the string. The rationale for using the largest proper suffix is to reduce the criterion of the search without missing any future matches. The matching process is shown in Algorithm 6.

Algorithm 6 Aho-Corasick Automaton string searching algorithm.

```
function ACSearch(target, patternAutomaton)
    node = patternAutomaton.root
    for i = 1 → length(target)
        while not hasChild(node, target[i])
            node = failure(node)
        node = child(node, target[i])
        if hasOutput(node) then outputWords(node)
end function
```

It is worth noting that the above algorithms report matches in order of their end-points. While reporting matches out of order may be acceptable for string searching, it is problematic for string creation. We discuss how to remedy this in Section 4.2.2. While both the trie search in Algorithm 5 and the Aho-Corasick search in Algorithm 6 are applicable to the problem of rule matching, the latter is considerably faster since information about previous matches are exploited. Algorithm 6 runs in $O(|T| + |Z|)$ where $|T|$ is the size of the target and $|Z|$ is the total number of matches. For random targets and patterns, the number of matches is expected linear time. The number of matches is bounded by $O(|T|^2)$ in the worst case. The worst case occurs when patterns are suffixes of other patterns which match at every position of the target.

The construction process is described in Algorithm 7. The algorithm augments the pattern trie in place. The symbols argument is a list of all symbols that may be used in the target and patterns, also called the *alphabet*.

The algorithm can be broken down into two stages. The first makes the necessary

Algorithm 7 Aho-Corasick Automaton construction algorithm.

```
function ACConstruction(patternTrie, symbols)
    root = patternTrie.root
    queue = {}
    for i = 1 → length(symbols)
        if hasChild(root, symbols[i]) then
            setFailure(child(root, symbols[i]), root)
            enqueue(queue, child(root, symbols[i]))
        else
            addChild(root, symbols[i], root)
    while length(queue) > 0
        node = dequeue(queue)
        suffix = failure(node)
        for i = 1 → length(symbols)
            if not hasChild(node, symbols[i]) then continue
            next = child(node, symbols[i])
            enqueue(node, next)
            if hasChild(suffix, symbols[i]) then
                setFailure(next, child(suffix, symbols[i]))
                addOutput(next, failure(next).output)
            else
                setFailure(next, root)
end function
```

adjustments to the nodes of depth one or less while the second fixes the remainder of the trie. To avoid infinite loops, the root is not allowed to have a failure edge. This is rectified by creating the missing edges and redirecting back to the root. The failure edges for all nodes of depth one point to the root.

The remaining nodes are modified in a breadth-first ordering although any root-to-leaf ordering is valid. The failure edge of each node builds on the failure edge of its parent. If a node and its failure node both contain an edge with the same label, then the corresponding child's failure node is extended from its parent's failure node by the edge label. Children without matching labels have the root as their failure node as seen in Figure 4.8.

The running time of the building process is $O(|T||A|)$ where $|T|$ is the size of the pattern trie and $|A|$ is the number of symbols in the alphabet. The memory requirements for the automaton is dependent on the method used to store the edges. We use a simple perfect-hash table scheme that requires the most memory but makes matching the most efficient. This scheme results in a total size of $O(|T||A|)$. Since the running time and size are dependent on the size of the alphabet it is important that it is made as small as possible.

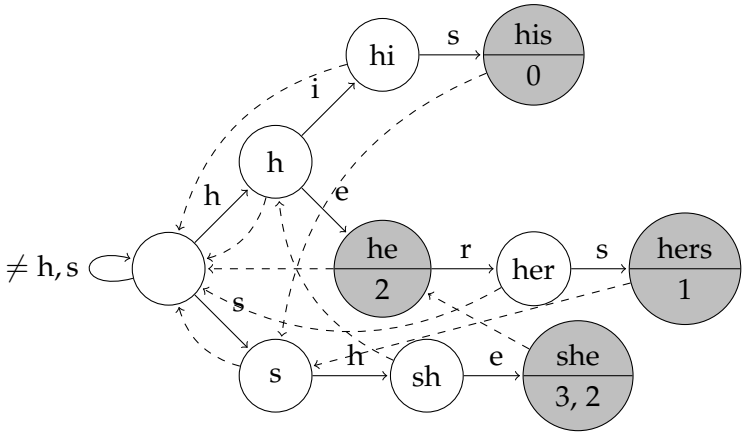


Figure 4.8: The Aho-Corasick automaton for the trie in figure 4.7. Dashed edges indicate failure edges. Shaded vertices indicate that it is the end of at least one word and the lower number indicates which words. The number of outputs for a given vertex may increase as seen in the 'she' vertex. Vertex labels implicitly indicate the words or partial words contained in the trie. Edge labels indicate letters that, when added to their parent's label, represent a new word.

L-System-Automaton Conversion

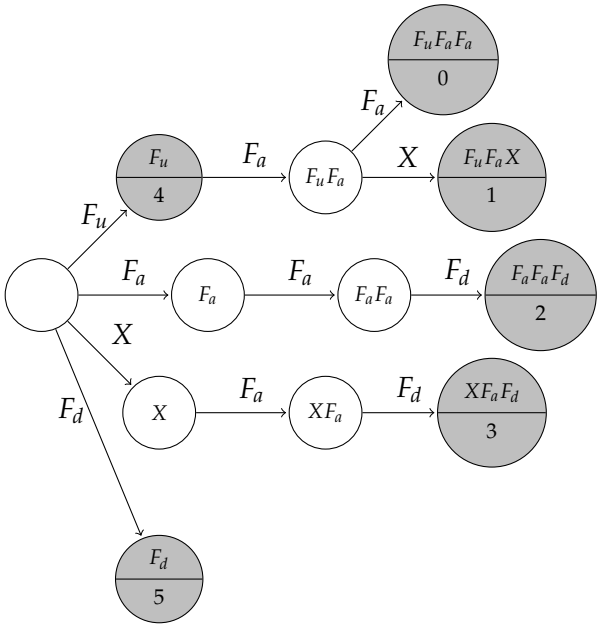
Before the algorithm can be used, the rules must first be transformed into patterns. The transformation process creates the patterns and associates them with each rule. The patterns are created by appending the left-context, strict predecessor and right-context of each rule in that order (the matching order). The following is the result of transforming the rules from Table 4.4.

$$\begin{aligned}
&\mapsto X\ F_u\ F_a\ X\\
F_u < F_a > F_a &\mapsto F_u\\
F_u < F_a > X &\mapsto F_d\ F_a\\
F_a < F_a > F_d &\mapsto F_d\\
X < F_a > F_d &\mapsto F_u\\
F_u &\mapsto F_a\\
F_a &\mapsto F_a
\end{aligned}$$

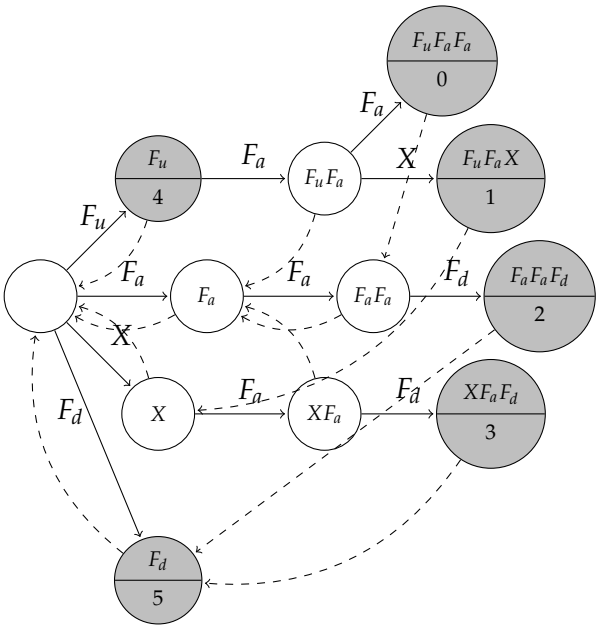
Table 4.4: An L-System demonstrating square-root growth from Algorithmic Beauty of Plants[54]. Square-root growth, while uncommon, is present in some natural objects.

$$\overbrace{F_uF_aF_a}^{Rule\ 0},\ \underbrace{F_uF_aX}_{Rule\ 1},\ \overbrace{F_aF_aF_d}^{Rule\ 2},\ \underbrace{XF_aF_d}_{Rule\ 3},\ \overbrace{F_u}^{Rule\ 4},\ \underbrace{F_d}_{Rule\ 5}$$

The remainder of the construction algorithm is unchanged. The following is the trie and automaton for Table 4.4.



Algorithm 8 shows the Aho-Corasick adapted for string creation. As stated before, the automaton only finds matches at their endpoints. This has several unexpected side effects. First, two rules of different sizes may be matched at the same point in the string. Secondly, rules can be matched out of order. On the other, L-System string creation requires that matches are performed in order. To accommodate this complication, a buffer of the same size as the input is required. The buffer is initialised to a special identifier, **infinity**, indicating that no matches have so far been discovered.



The buffer stores the results of each match at the corresponding strict predecessor. The corrected strict predecessor position is found by subtracting the size of the matched rule's right context from the current position. Once the entire input string has been scanned, the matched rules are evaluated in that order and stored in the output string. In order to evaluate expressions correctly, it is necessary to store an additional buffer, *savedParameterPositions* that caches the value of the parameter buffer pointer was at the start of a symbol. At the start of each iteration of the output loop, the parameter buffer pointer is set to the cached version.

The running time of this algorithm is $O(|I| + |Z| + |O|)$. That is, linear in the size of the text, number of input symbols, number of matches, and size of the output, respectively. In the case of L-Systems the worst case number of matches is $O(|T||R|)$ where $|T|$ is the size of the input and $|R|$ is the number of rules. For the worst case to occur, the patterns derived from the rules must be suffixes of other rules.

The algorithm above uses $O(N)$ memory to achieve a substantial speed up. In some cases, however, devoting additional memory to store the input, output and additional matching arrays is simply too much (when memory is expensive). The following algorithm reduces the memory overhead without sacrificing asymptotic performance.

4.2.3 Automaton Chain Writer

The previous method offers better performance over brute force, however, it comes at the cost of greater memory use. The Automaton Chain Writer remedies this by representing the entire process of constructing the final string from the axiom as a chain of automatons. Each automaton performs only a small part of the string creation process at a time. When output is ready at a stage, the next stage is notified. In this way, the final string can be read one symbol at a time.

Figure 4.9 shows how each element in the automaton chain is structured. The input buffers are logically connected to the output buffers of the previous generation, while

Algorithm 8 Aho-Corasick L-System string creation algorithm.

```
function applyAC(input, rules, patternAutomaton)
    node = patternAutomaton.root
    for i = 1 → length(input)
        savedParameterPositions[i] = parameterPointer
        while not hasChild(node, input[i])
            node = failure(node)
        node = child(node, input[j])
        for rule in node.output
            if evaluate(rule.condition, input, i) then
                position = i - length(rule.rightContext)
                if matches[position] > rule.index then matches[position] = rule.index
    for i = 1 → length(input)
        parameterPointer = savedParameterPositions[i]
        if matches[i] = infinity then
            output += copySymbol(input[i])
        else
            output += addToOutput(rules[matches[i]])
    return output
end function
```

the output buffers are logically connected to the input buffers of the next generation. From an implementation standpoint, connected buffers are represented by the same memory. The automaton (the same automaton as in the the previous method) is used to produce output symbols from input symbols, as shown in Figure 4.10.

Using this modification it is possible to generate strings recursively rather than the previous iterative methods. The recursive method allows one to 'lazily' create the final string at a given generation. Each automaton computes enough of its corresponding generation to output a single symbol of the final string. When insufficient information is available it queries up the chain until enough input is gathered.

At first glance it appears that this approach requires much more memory as more symbol and parameters buffers are needed. However, the approach allows significant memory reductions. In previous string generation methods, the sizes of the output buffers are related to the behaviour of the L-System. The output buffers needed to be large enough to hold all of the output. However, since each generation is computed in parallel, we need only store a small window of each previous generation to be able to compute the next symbols. The key observation is that input symbols from sufficiently far away in previous generation will never be used for matching again. Symbols and parameters past this limit can be safely discarded without affecting L-System behaviour. This means that each buffer can be *much* smaller than the previous techniques.

Instead of using the standard array data structure to store the buffers, a *circular buffer* is used to store the contents of each buffer without wasting too much space. A circular buffer is an array of fixed size. Accessing to an index is performed modulo the buffer's size, which creates circular access patterns.

The minimum size of the circular buffer can be determined by examining the rules.

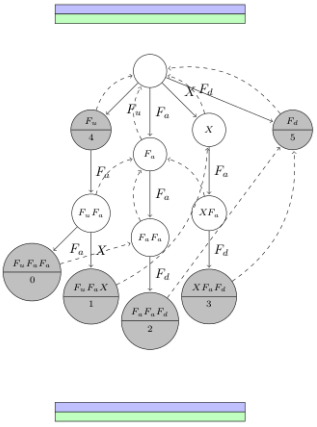


Figure 4.9: An element of the automaton chain. Each element contains two input buffers, an automaton and two output buffers.

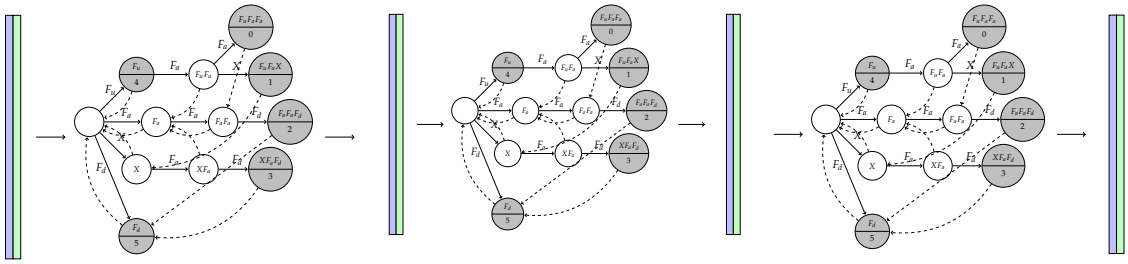


Figure 4.10: An automaton chain for three generations. Each of the automatons shown above represents a generation that we would like to generate. Buffers connect each of the automatons, enabling output from generation to be used as input for the next.

Since the same buffers are used for both input and output purposes, we must inspect both sides of each rule. The minimum size of each symbol buffer, S, is computed by the following expression:

$$S = \max(\max(1 + R_{i_{\text{right context}}}), \max(R_{i_{RHS}}))$$

where $R_{i_{\text{right context}}}$ is the size of the right context of the i-th rule and $R_{i_{RHS}}$ is the size of the largest right-hand side of the i-th rule. The first argument ensures that we have enough input buffer space to match the right context and strict predecessor of a rule, while the second argument ensures that enough output buffer space is available to fully write a given right-hand side. The size of the parameter buffers is computed as the maximum of the greatest number of parameters in a single module times the length of the symbol buffer. Although not exact, this simple calculation bounds the maximum number of parameters in the window to a small constant.

In other methods, the final string is fully stored for future interpretation. However, by performing string generation and interpretation at the same time it is possible to forgo this requirement.

The result of these modifications reduce the memory required for this approach significantly. Instead of requiring large arrays of possibly exponential size, several

buffers of only constant size are used. The number of buffers is related to the number of generations being simulated, however, this quantity is likely to be very small in comparison to the other quantities such as the string length. On the other hand, the additional complexity for the automaton chain representation and extra bookkeeping for lazy evaluation result in greatly increased run-time overhead.

Algorithm 9 Automaton Chain String creation algorithm.

```
function writeToOutput(generation)
    position = decidedPosition[generation]
    decidedPosition[generation] += 1
    ruleToApply = savedMatches[generation].valueAt(position)
    currentParameterPosition = parameterPointer
    parameterPointer = savedParameters[generation].valueAt(position)
    symbol = input[generation].valueAt(position)
    if symbol = -1 then
        addSymbol(-1, generation + 1)
        isFinished[generation + 1] = true
    else
        if ruleToApply = infinity then-
            output[generation + 1] += copySymbol(symbol)
        else
            output[generation + 1] += copyRHS(ruleToApply)
        end if
    end if
    parameterPointer = currentParameterPosition
end function
```

Algorithms 9, 10 and 11 show the main functionality required to generate strings using this method. This method is, by far, the most complex of the string creation methods. The complexity is exacerbated a similar problem as with the Automaton Writer method. Since we use the automaton to find matches, the matches that we do find are out of order with respect to the generation's input string. In the previous method, we solved the problem by simply writing all the matches to a buffer and computing output from them after the entire input string has been scanned. This is not possible for the Automaton Chain Writer, as each generation's output (and input) string is computed in parallel. One alternative is to have each generation fully compute the output string before proceeding. This is not a desirable solution as the method would devolve to the Automaton Writer and not give any memory reductions.

The main observation to our solution is the same as that which allows us to save memory. A set of L-System rules cannot be too far out of order. That is, a match can only be delayed by the length of the longest right-hand context. Our solution only outputs matches when it is certain that no further matching decisions can affect it. To do this, we keep several pointers to the input strings. First, we keep the amount of input that we have so far received from the previous automaton chain, *inputBufferLength*. Second, we keep a pointer to the amount of input data that we have fed into the automaton,

Algorithm 10 Automaton Chain String creation algorithm.

```
function readFromInput(generation)
    while true
        if autoPosition[generation] - readAhead < decidedPosition[generation]
            break
        if autoPosition[generation] ≠ inputBufferLength[generation]
            break
        position = autoPosition[generation]
        autoPosition[generation] += 1
        symbol = input[generation].valueAt(position)
        if symbol = -1
            break
        parameterPos = autoParameterPosition[generation]
        parameterSize = length(symbol.parameters.size)
        savedMatches[generation].setValueAt(infinity, position)
        savedParameters[generation].setValueAt(parameterPos, position)
        while not hasChild(node[generation], symbol)
            node[generation] = failure(node[generation])
        node[generation] = child(node, symbol)
        for rule in node[generation].output
            if not evaluate(rule.condition, input, i) then continue
            matchPosition = position - length(rule.rightContext)
            currentParameterPosition = parameterPointer
            parameterPointer = savedParameters[matchPosition]
            if savedMatches[position] > rule.index then matches[position] = rule.index
            parameterPointer = currentParameterPosition
        autoParameterPosition[generation] += parameterSize
    end while
end function
```

Algorithm 11 Automaton Chain String creation algorithm.

```
function READ(generation)
    canSafelyWrite = (autoPosition[generation] - readAhead >= decidedPosition[generation])
    canReadAbove = (not GenerationFinished[generation])
    hasWaitingInput = (decidedPosition[generation] ≠ inputBufferLength[generation])
    if canSafelyWrite
        writeToOutput(generation)
    else if not canReadAbove and hasWaitingInput
        readFromInput(generation)
        writeToOutput(generation)
    else if not canReadAbove and not hasWaitingInput
        addSymbol(-1, generation + 1)
        isFinished[generation + 1] = true
    else if canReadAbove
        safePosition = (autoPosition[generation] - readAhead)
        canReadInput = (safePosition < decidedPosition[generation])
        hasReadInput = (autoPosition[generation] ≠ inputBufferLength[generation])
        if hasReadInput and canReadInput
            readFromInput(generation)
            read(generation)
        else
            read(generation - 1)
            read(generation)
        end if
    end if
end function
```

autoPosition. Finally, we keep a pointer to the number of matches from the input string that we have produced output from, *decidedPosition*. There is a single instance of these circular buffers per layer in the automaton. By ensuring that, for a given generation, the *decidedPosition* and *autoPosition* are far enough apart, we can produce correct results.

Due to its complexity, we require three functions for this method: *writeToOutput*, *readFromInput* and *read*. Each of these methods takes a single parameter, the generation which we want to evaluate (corresponding to the level in the automaton).

The writeToOutput method writes a single *safe* match to the output buffers (the input buffer to the next element in the chain). readFromInput uses input contained in the input for the current automaton, represented by *node*. When a symbol and its parameters are required for the final string. The user calls the read function with the required generation. This method will return with fresh data available in its output buffer. The method essentially delegates work to the other functions by performing checks to see which ones are safe. If no input exists for this generation, the method queries up the chain by recursively calling the read function. In some instances, the function is required to restart as new input is available. In this case the function is called recursively as well.

Each of the methods return in constant time, assuming a constant number of matches per input symbol. The entire method has the same theoretical run-time complexity as the Automaton Writer, $O(|I| + |Z| + |O|)$. That is, linear in the size of the text, number of input symbols, number of matches, and size of the output, respectively. On the other hand, the method is not likely to be as fast as the Automaton Writer method due to the amount of recursion and other overheads that exists in order to keep the output correct.

4.3 Summary

In this chapter we have discussed various mechanism for accelerating L-System evaluation. In Section 4.1, methods for making the L-System rule representation more efficient were discussed. The problem was discussed from two different aspects: *RHS Selection* and *Argument Evaluation*. Two different mechanisms for RHS Selection were discussed. The first was a standard roulette wheel approach to selection, while the our method involved preprocessing an array to have the same probability distribution. The array could be used to select right-hand sides in constant time. In Argument Evaluation, a naïve Search-and-Replace algorithm was detailed. This method requires significant amount of processing to performed at run-time. We present a new technique for preprocessing the L-System rules so that no processing is required besides the evaluation of the L-System expressions.

Finally, in our last section we discussed methods for *String Creation*. Three techniques for String Creation were discussed. The Brute Force method and two novel methods labelled Automaton Writer and Automaton Chain Writer. The Brute Force method has a poor theoretical run-time complexity as the running time is dependent on both the number of rules and their average size. By recognising the problem as being string searching, we introduce an Automaton Writer method based on Aho-Corasick. This running time for this method does not rely on the number of rules or their average length giving it better overall complexity. A second method, Automaton Chain Writer, attempts to reduce the memory requirements of the string creation process. This

method has significantly more overhead, however, it is able to create the final string in a lazy manner thus reducing memory costs.

Testing of these techniques is left to Chapter 7. In the next chapter, *L-System Optimisation*, we discuss methods for altering the L-System rules to produce more efficient behaviour without affect the appearance of the output trees.

Chapter 5

L-System Optimisation

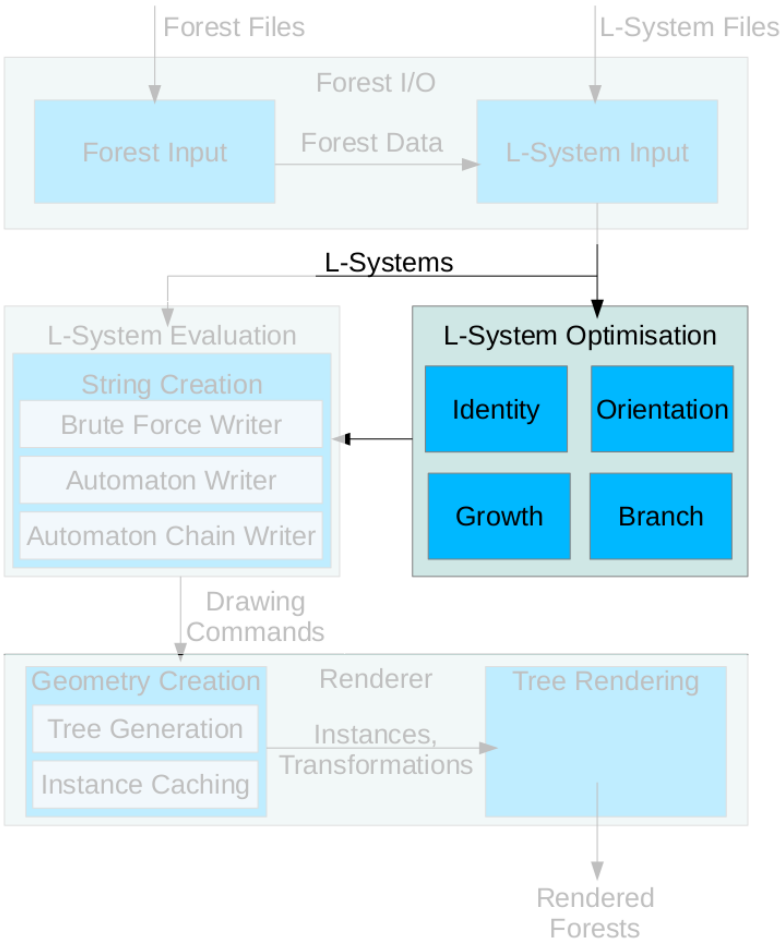


Figure 5.1: The above diagram shows the L-System Optimisation component. This component accepts input, in the form of unoptimised L-Systems, from the Forest I/O component. The optimised L-Systems are sent to L-System Evaluation component so that tree drawing commands can be created.

Our L-System Optimisation framework, shown in Figure 5.1, is designed to enable

easy testing of optimisation techniques. The component accepts unoptimised L-Systems which are transformed and sent to the L-System Evaluator. Most of the optimisations are independent of each other and may be performed in any order.

This chapter discusses four techniques for optimising L-Systems. These are, in increasing order of complexity: *Identity Rule Optimisation*, *Orientation Optimisation*, *Growth Optimisation* and *Branch Optimisation*. The identity rule optimisation adds extra rules to the L-System so that certain evaluation decisions can be made sooner. The orientation optimisation concatenates runs of orientation symbols in rules into a single module with the same effect. The growth optimisation attempts to model the growth of geometry creating rules more efficiently. Finally, the branch optimisation uses heuristics to detect rules which create branches so that they can be more efficiently handled by our system. Certain results of the L-System Optimisation may be dependent on which technique is chosen for String Creation. For this reason, testing and analysis are left to Chapter 7.

Each component is intended to limit a negative aspect of L-Systems, particularly the extreme computation times and memory requirements in certain cases. The downside of such optimisations is the preprocessing that is required, which in some cases necessitates complicated algorithms.

5.1 Identity Rule Optimiser

Identity rules are productions in an L-System that do not produce any new output, they simply carry the current symbol in a string unmodified to the next generation. An example of an identity function would be: $A \mapsto A$, which maps A with no parameters to itself. Symbols with differing numbers of parameters are treated as distinct in our system. For instance, the symbol A with one parameter is different from A with three parameters.

The task of carrying over unmatched symbols is traditionally performed by the rule matching algorithm. In some cases, the information required might not be known until much later when it is known that no other rules match. Only the naïve implementation, discussed in Section 4.2.1, does not suffer from this drawback. The *Identity Rule Optimisation* adds identity rules to the L-System, safely transferring the work to when it is required. The optimisation analyses the given rules to determine which identity rules are missing. The rationale behind this is to provide rule matching algorithms with extra information without increasing their complexity.

Table 5.1 shows a deterministic L-System for demonstration purposes. The L-System contains many symbols that will never be matched by existing rules. For instance, '&' does not occur in any rule's left-hand side and will remain unmatched regardless of the generation.

5.1.1 Symbol Detection

Symbols which do not occur as a strict predecessor in any rule are called *terminal symbols*. In context-free and non-parameterised L-Systems, terminal symbols are exactly those that require identity rules, since they are never matched. We use a two-pass algorithm to find these terminal symbols.

	$\mapsto A$	(axiom)
A	$\mapsto F(1)$ [/ & B]	
$F(t)$	$\mapsto F(2 \times t)$	
$F(t)$	$\mapsto f(2 \times t)$	
$f(t)$	$\mapsto f(2 \times t)$	**if** $t \leq 100$
$f(t)$	$\mapsto f(3 \times t)$	**if** $t > 100$
$/ < B$	$\mapsto A$	

Table 5.1: A contrived L-System designed to demonstrate identity optimisation.

The first pass over the rules collects all of the symbols in the ruleset. Only the axiom and the right-hand sides of rules are inspected. Each encountered symbol is inserted into a set if it is not already present. The second pass examines the left-hand sides of each rule. This time, however, instead of inserting symbols we delete them. The erased items represent all the non-terminal symbols. The set now contains only terminal symbols. In the case of context-free, non-parameterised L-Systems, these are all the symbols for which identity rules should be created.

Context-sensitive L-Systems require a modification to the second pass of the detection algorithm. The problem occurs when left-hand sides that contain a non-empty context is encountered such as the last rule in Table 5.1. During the rule matching process the strict predecessors could match but the contexts might not. In this circumstance the symbol should still be bought forward to the next generation. For this reason, the second pass is modified to exclude any context-sensitive rules from our search. Although the symbol is retained in the set at this point, it is possible for a later rule to erase the symbol.

It is possible, though unlikely, that the L-System will specify rules for every possible context for a strict predecessor. When this occurs, the symbol under consideration will always match at least one rule and an identity rule for the symbol will be unnecessary. Identifying this case is complicated which we consider too costly to be worthwhile catering for. In any event, including these extra identity rules should not affect the behaviour of an L-System.

When parameterised modules are involved, detection becomes more involved. The difficulty comes when rules have *conditional guards*. A conditional guard is an expression that must be true before the rule can be expanded. It is evaluated after the rule has been matched. The expression can reference parameter values in modules in the input. It is generally not possible to know the result of the conditional guards without running the L-System so our algorithm needs be as conservative as possible. In this case, we modify the second pass is modified to skip rules with conditions.

Although we use a conservative approach, it is possible to do better by examining the preconditions. It is possible that sets of preconditions could combine in a way that at least will be evaluated when rules match. An example of this is the second last and third last rules in Table 5.1. The conditional guards on these are opposite to each other; if the first condition is false, the second condition is true and vice versa. Other

L-Systems may contain other instances which are not as simple. Detecting these cases is complicated and requires logic solvers which require exponential time to run. Instead, our conservative approach will hit most cases and run in a fraction of the time.

With these two modifications are able to handle the context-sensitive, parameterised L-Systems. The running time of the symbol detection algorithm is dependent on set data structure that is being used. It requires $O(\sum |R_i|)$, (the sum of the lengths of each rule) set insertions and deletions to complete. In our implementation, a hash set (a variant of a hash table modified for set representation) is used for fast insertion and deletion in amortised constant time[58]. Results for this optimisation can be found throughout Chapter 7.

Algorithm 12 Non-terminal symbol detection.

```
function findTerminals(rules)
    terminals = set(uniqueSymbols(rules))
    for r = 1 → length(rules)
        rule = rules[r]
        if rule.hasCondition = true then
            continue
        if length(rule.leftContext) ≠ 0 or length(rule.rightContext) ≠ 0 then
            continue
        terminals.erase(rules[r].strictPredecessor)
    return terminals
end function
```

The process is described in Algorithm 12. The pseudocode assumes a function called uniqueSymbols, which simply returns the collection of unique symbols that are present in the rules. The next section will discuss the addition phase of this optimisation.

5.1.2 Identity Rule Addition

Once detection is complete, new rules must be added based on the contents of the set. For each symbol in the set, the following steps are applied. In the case of non-parameterised symbols a new rule is constructed containing the same symbol on the left- and right-hand sides. This is the identity rule for a non-parameterised symbol as shown in the previous section. Modules, on the other hand, require additional work for each parameter. *Identity functions* must be created for each parameter. While the symbol part of the identity rule performs the copying of the symbol to the next generation, the identity function copies the parameters from the old module over to the next. This is shown in Algorithm 13. The code assumes the existence of a createDummyVariable function which creates a simple dummy variable of the form t_i, to use in expressions.

The following is an example of an identity rule for a module with three parameters: $A(t_0, t_1, t_2) \mapsto A(t_0, t_1, t_2)$. A separate dummy variable must be used for each parameter, in this case t_0, t_1 and t_2. The dummy variables are used as arguments in the output module. The expression in each argument is the same as the the input parameter, in other words the identity function. This indicates that the output values should be identical to the input values when evaluated.

Algorithm 13 Identity rule addition.

```
function createIdentityRule(symbol, parameters)
    rule = Rule()
    rule.strictPredecessor = symbol
    for i = 1 → parameters
        rule.strictPredecessor.addParameter(createDummyVariable(i))
    rule.output = symbol
    for i = 1 → parameters
        rule.output.addArgument(createDummyVariable(i))
    return rule
end function
```

	$\mapsto A$	(axiom)
A	$\mapsto F(1)$ [/ & B]	
$F(t)$	$\mapsto F(2 \times t)$	
$F(t)$	$\mapsto f(2 \times t)$	
$f(t)$	$\mapsto f(2 \times t)$	**if** $t \leq 100$
$f(t)$	$\mapsto f(3 \times t)$	**if** $t > 100$
/ < B	$\mapsto A$	
[	$\mapsto$ [	
/	$\mapsto$ /	
&	$\mapsto$ &	
]	$\mapsto$]	
$f(t_0)$	$\mapsto f(t_0)$	
B	$\mapsto B$	

Table 5.2: L-System with identity rules added.

After the new rules have been added, the new ruleset is returned for use by the system. Table 5.2 shows the ruleset described in Table 5.1 after the identity. In the six tree L-Systems we tested, the number of rules tended to roughly double. On the other hand, the new rules are relatively simple compared to pre-existing rules. It is important to note that this optimisation changes the type of symbols in the grammar; after the optimisation has finished no non-terminal symbols will remain.

In our initial experiments, this optimisation was shown to be the fastest to apply but has the least impact. The speed is due to the process being only a simple loop over all the rules and a few insertions at the end of our system's 'rule vector.' One reason for the poor performance of this optimisation is that it does little to change the exponential behaviour inherent in tree L-Systems.

$$V = \begin{pmatrix} r_x & u_x & f_x \\ r_y & u_y & f_y \\ r_z & u_z & f_z \end{pmatrix}$$

Figure 5.2: The matrix stores the current right-, up- and forward-facing vectors as columns. These columns correspond to the relative x-, y- and z- axes, respectively.

5.2 Orientation Optimisation

The orientation optimisation improves both String Creation and the creation of tree geometry for tree L-Systems. Orientation changing symbols are crucial to tree L-Systems and occur frequently in rules. These symbols are responsible for creating the structure of a tree and its parts. Leaves, for instance, are specified using orientation symbols to describe shape. There are seven symbols which affect the current orientation in the L-System: $+$, $-$, &, ^, /, \, and | (explained in Section 2.1.1). These symbols change the orientation of the turtle a single axis at a time. This means that multiple symbols may be required to achieve the appropriate affect. Having large runs of orientation changing symbols is undesirable because it can slow down string creation and tree geometry creation.

Encountering an orientation symbol during the string creation phase, usually leads to the symbol being copied over to the next generation. If enough of these symbols exist, the overhead of copying these symbols can slow the system down significantly. When an orientation symbol is encountered during the interpretation phase several actions must be performed. The symbol is transformed into an appropriate matrix, which is then multiplied with the current *view transformation matrix*. The view transformation matrix stores the relative x-, y- and z-axes, as shown in Figure 5.2, and is required to grow tree parts in the correct directions.

In our examinations of tree L-Systems, orientation-changing symbols contribute the second largest proportion of symbols in the final string after geometry symbols. For this reason it is important to ensure that they are handled as efficiently as possible. Orientation optimisation modifies existing rules in the L-System that contain orientation changes. Examples of such rules include are when branching occurs and orientation symbols are used to build in the intended direction, and when the L-System draws a leaf and orientation symbols are used to trace the outline.

	$\mapsto A$
A	$\mapsto [\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /'\ [\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /\ /\ /'\ [\ \&\ F\ L\ !\ A\ ]$
F	$\mapsto S\ /\ /\ /\ /\ /\ F$
S	$\mapsto F\ L$
L	$\mapsto [\ ''\ ''\ \hat{}\ \hat{}\ \{\ -\ f\ +\ f\ +\ f\ -\ \vert\ -\ f\ +\ f\ +\ f\ \}\ ]$

Table 5.3: A bush L-System from The Algorithmic Beauty of Plants[54].

Orientation optimisation attempts to minimise costs by focusing on runs of consecu-

tive orientation symbols. Using a single rotation matrix, it is possible to represent every combination of orientation changes, thus using a single symbol place of several. Table 5.3 shows an example L-System from The Algorithmic Beauty of Plants[54]. Approximately 50% of the symbols in every rule, except the S rule, are orientation changing, many of them occurring in one of three runs.

This optimisation modifies existing rules in the L-System to make them more efficient, however, it is important that the modified rules retain their meaning. The algorithm must not change any symbol that is used in the matching process, doing so would lead to altered behaviour. We use the following algorithm to determine the set of modifiable symbols.

5.2.1 Valid Symbol Detection

First, a *valid* set is initialised to contain every possible orientation symbol. Since our system discriminates between orientation symbols with no parameters and orientation symbols with one parameter, there are a maximum of thirteen different entries. As with identity rule optimisation a hash set is used to store the valid set.

The algorithm, Algorithm 14, examines the left-hand side of each rule. If an orientation changing symbol is encountered it is removed from the valid set. Orienting symbols that occur on the left-hand side of rules mean that a decision will depend on the symbols presence. Optimising out the symbol could alter, or even break the L-System. After each left-hand side has been examined, the valid set contains those orientation symbols that can be further optimised.

Algorithm 14 Valid orientation symbol detection.

```
function findValidOrientationSymbols(rules)
    valid = set([\, /, +, -, &, ^, |, \#1, /#1, +#1, -#1, &#1, ^#1])
    for r = 1 → length(rules)
        rule = rules[r]
        if rule.strictPredecessor in valid then
            valid.erase(rule.strictPredecessor)
        for i = 1 → length(rule.leftContext)
            if rule.leftContext[i] in valid then
                valid.erase(rule.leftContext[i])
        for i = 1 → length(rule.rightContext)
            if rule.rightContext[i] in valid then
                valid.erase(rule.rightContext[i])
    return valid
end function
```

5.2.2 Elimination of Orientation Symbol Runs

With the set of eligible symbols identified, the optimisation can proceed to its final phase. Each right-hand side is now scanned for runs of orientation changing symbols. The runs do not have to be of the same symbol type, however they must be at least two symbols long. This limit is imposed to stop the needlessly performing the optimisation.

Secondly, leaving as many original symbols in place as possible makes the L-System more readable to its creator. The optimisation replaces the entire run with a symbol whose parameters represent a rotation matrix.

A single run is converted to a matrix by the following procedure. First, a 3x3 matrix, V, is initialised to the identity matrix. V represents a temporary view transformation matrix that can be used in the run. Since each symbol acts relative to the current orientation, V is necessary to calculate the correct axes. If V is the identity matrix then it indicates that there are no current effects.

The algorithm iterates through each symbol in the run. At each step, V is modified to incorporate the effects of the new symbol. Depending on the symbol, V is multiplied by one of seven matrices that take into account the current state of V. The matrices used are:

$$+_\theta = R(r,\theta) \qquad \&_\theta = R(u,\theta) \qquad /_\theta = R(f,\theta)$$

$$-_\theta = R(r,-\theta) \qquad \hat{}_\theta = R(u,-\theta) \qquad \backslash_\theta = R(f,-\theta)$$

$$| = R(u,180)$$

where

$$R(a,\theta) = \begin{pmatrix} \cos\theta + a_x^2(1-\cos\theta) & a_x a_y(1-\cos\theta) - a_z\sin\theta & a_x a_z(1-\cos\theta) + a_y\sin\theta \\ a_y a_x(1-\cos\theta) + a_z\sin\theta & \cos\theta + a_y^2(1-\cos\theta) & a_y a_z(1-\cos\theta) - a_x\sin\theta \\ a_z a_x(1-\cos\theta) - a_y\sin\theta & a_z a_y(1-\cos\theta) + a_x\sin\theta & \cos\theta + a_z^2(1-\cos\theta) \end{pmatrix}$$

and r, u and f correspond to the right-, up- and forward-facing axes for V, respectively.

$R(a,\theta)$ represents a procedure called *Rodrigues' rotation formula*[29] that transforms a rotation in *angle-axis form* to *rotation matrix form*. As each symbol is consumed V is modified to mimic its effects. The algorithm finishes with V capturing all the transformations in the run.

In order to convey the information stored in V, a new module called *rotateMatrix* is used. rotateMatrix takes exactly four arguments representing the elements of V. The mapping from V to these arguments is performed by computing the angle-axis form of the matrix. This is a more compact representation and does not require significant work to unpack. When the rotateMatrix module is encountered during the interpretation phase a special function is called. This function, not present in regular L-Systems, adds the required code to modify the system's current view transformation matrix based on the custom parameters. The process is shown in Algorithm 15. In this pseudocode, several functions are assumed to have been created: copyLHS, which copies the left-hand side from the argument, createRHS, which creates a blank RHS, addSymbol, which adds the argument symbol to the end of the RHS, and concatenate, which performs returns a rotateMatrix module representing the run in the arguments.

Algorithm 15 Elimination of Orientation Symbol Runs.

```
function eliminateRun(rule, valid)
    outputRule = Rule()
    outputRule.copyLHS(rule)
    for r = 1 → length(rule.rhses)
        rhs = rule.rhses[r]
        outputRHS = outputRule.createRHS(rhs)
        for s = 1 → length(rhs)
            addedNewSymbol = false
            if rhs[s] in valid then
                t = s + 1
                for t → length(rhs)
                    if rhs[t] not in valid then break
                if t > s + 1
                    outputRHS.addSymbol(concatenate(rule, s, t))
                    addedNewSymbol = true
            if not addedNewSymbol then
                    outputRHS.addSymbol(rhs[s])
    return outputRule
end function
```

Table 5.4 shows the ruleset from Table 5.1 after optimisation. Rules 1 and 2 have been reduced significantly, while rule 4 still has many orientations. The unoptimised version contains thirty orientation rules while the optimised version has just under half that at thirteen.

Although our method is very effective at reducing the number of orientation symbols, it is not the only method that can be used. Simply by using parameterising the orientation symbols from Table 5.3, one can reduce the number of symbols required dramatically. Table 5.5 shows the reduction that can be achieved by parameterising orientation symbols, however, there are still some situations where it is not as powerful as our optimisation. This is because parameterisation can only be used to concatenate orientation symbols of the same type. For instance, our optimisation is able to combine the symbols '$-\mid-$' into a single rotateMatrix symbol.

Initial experiments show that this optimisation, while fast to compute, also has limited success in reducing the running time of the L-System. The majority of running time for an L-System occur in branch creation, which adds 'recursive' work to the tree, and creation of geometry and geometry symbols, which increase the amount of symbols required to represent each branch. Both of these circumstances are not affected by this optimisation, however, the next optimisation will improve these cases. The results for the orientation optimisation can be found throughout 7.

5.3 Growth Optimisation

A problem inherent to L-Systems is the possibility for rules to exhibit exponential behaviour. A major contributor to exponential behaviour is the *growth rules* which

$$
\begin{array}{ll}
 & \mapsto A \\
A & \mapsto [\ \&\ F\ L\ !\ A\ ]\ \text{rotateMatrix}(\ldots)\ '\ [\ \&\ F\ L\ !\ A\ ]\ \text{rotateMatrix}(\ldots)\ '\ [\ \&\ F\ L\ !\ A\ ] \\
F & \mapsto S\ \text{rotateMatrix}(\ldots)\ F \\
S & \mapsto F\ L \\
L & \mapsto [\ '\ '\ '\ \text{rotateMatrix}(\ldots)\ \{\ -\ f\ +\ f\ +\ f\ \text{rotateMatrix}(\ldots)\ f\ +\ f\ +\ f\ \}\ ]
\end{array}
$$

Table 5.4: The bush L-System with orientation symbols optimised. The parameters to rotateMatrix have been omitted. Using this optimisation only 13 orientation symbols and modules remain.

$$
\begin{array}{ll}
 & \mapsto A \\
A & \mapsto [\ \&\ F\ L\ !\ A\ ]\ /(112.5)\ '\ [\ \&\ F\ L\ !\ A\ ]\ /(157.5)\ '\ [\ \&\ F\ L\ !\ A\ ] \\
F & \mapsto S\ /(112.5)\ F \\
S & \mapsto F\ L \\
L & \mapsto [\ '\ '\ '\ \hat{}\ (45)\ \{\ -\ f\ +\ f\ +\ f\ -\ |\ -\ f\ +\ f\ +\ f\ \}\ ]
\end{array}
$$

Table 5.5: The bush L-System with orientation symbols optimised. This version has been optimised by using additional parameters to combine multiple orientation symbols of the same type into a single symbol. In this example, we assume that the default angle for parameterless symbols is 22.5 degrees. Only 15 orientation symbols and modules remain after optimisation.

lengthen parts of the tree so that older branches are longer than newer branches. Salomaa and Rozenberg[60] observe that growth of L-System rules ranges from polynomial to exponential. Branches in trees tend to grow slowly at first but much faster as they age. This is often modelled as a form of exponential growth. It is this behaviour that growth optimisation seeks to address.

Rules that exhibit such growth affect performance of the system in two ways. First, the size of the tree is directly proportional to the length of the intermediate and final strings. As the tree grows, these intermediate strings become longer, resulting in increased memory usage and computation time.

Second, the number of geometric primitives required to render the tree is roughly equivalent to the number of geometry symbols in the string. The net result is that a tree can consist of many thousands of geometric primitives even if they contribute little to perceivable tree detail.

An example of these effects can be seen in Tables 5.6 and 5.7. As the generation increases, the number of geometry symbols, F and L, increases dramatically. Much of the geometry for this model is concentrated in long, straight branches which leads to slower rendering and increased memory usage. Instead of using many small cylinders, it is much more efficient to use a few longer ones with the caveat of avoiding slivers. The goal of this optimisation is to recognise instances of problematic growth and perform

$$
\begin{array}{ll}
 & \mapsto A \\
A & \mapsto [\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /'\ [\ \&\ F\ L\ !\ A\ ]\ /\ /\ /\ /\ /\ /\ /'\ [\ \&\ F\ L\ !\ A\ ] \\
F & \mapsto F\ F \\
L & \mapsto [\ '''\ \hat{}\ \hat{}\ \{\ -\ f\ +\ f\ +\ f\ -\ |\ -f\ +\ f\ +\ f\ \}\ ]
\end{array}
$$

Table 5.6: A leafy bush L-System from The Algorithmic Beauty of Plants [54]. A visualisation of this L-System can be seen in

Generation	Fs	Leaves	Triangles
5	633	120	10,488
6	1,995	363	33,009
7	6,177	1,092	102,108
8	18,915	3,279	312,477

Table 5.7: Complexity of the output string from the L-System described in 5.6. F refers to the number of cylinder symbols in the final string.

the replacement at the L-System rules level.

There has been a significant amount of previous work in the area of the growth of rules, although much of it is too specific to be used in our system. The majority of work is concerned with the total length of rules, rather than the number of geometry symbols that are created. It has been proven that the growth functions achievable by deterministic OL-System grammars are combinations of polynomial and exponential functions[60]. This has important implications in that it may not be possible to create a standard OL-System with a desired growth rate. On the other hand, it may be possible to create an IL-Systems with the appropriate behaviour, however it is undecidable whether a desired growth is representable[65]. Prusinkiewicz and Lindenmayer[54] present a matrix formula for determining the growth of any deterministic OL-System.

We focus our research on two forms of problematic growth that are common in tree L-Systems. These cases are the *constant-base exponential* case and the *binomial* case and are shown in Table 5.8 and Table 5.9.

5.3.1 Constant-base Exponential Case

The constant-base exponential is the simplest of the two cases to recognise and optimise. The algorithm performs multiple passes. First, each rule is matched against the following template: $X \mapsto X\ \ldots\ X$ where X is some non-terminal symbol. If no matching rule is found, the algorithm exits, reporting no match for the constant-base exponential case. Alternatively, if a match is found then we record X, the type of the symbol, and the number of X's which corresponds to the *base* of the exponential. The most common, and costly, symbol found in growth rules is F, but the algorithm is not limited to this case.

Since the growth rules will be modified it is important that no other rules are affected. A second pass is made over each rule's left-hand side to ensure that no other

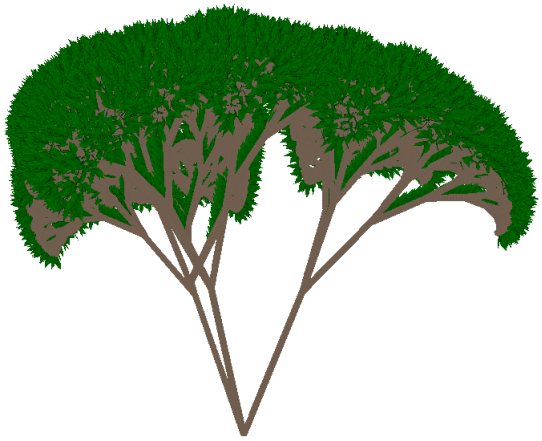


Figure 5.3: A visualisation of the tree from the L-System in 5.6. The plant has a bushy appearance due the many leaves present.

$$F \mapsto F\ F$$

Table 5.8: A rule show the constant-base exponential case. The base of the exponential is the number of 'F' symbols on the right-hand side. In this case, the base is two which represents the number of 'F' symbols doubling from one generation to the next.

$$F \mapsto^{a} F$$
$$F \mapsto^{b} F\ F$$

Table 5.9: A stochastic rule demonstrating the binomial case. The superscript a and b ($a + b = 100\%$) represent the probabilities that each rule will be expanded. The distribution of growth of 'F' symbols from one generation to the next is a binomial distribution.

Xs are found. The presence of other X symbols indicates that the L-System makes decisions based on its current growth and the optimisation is not viable in such situations. Algorithm 16 shows the first pass of the optimisation. If the heuristic detects the exponential case, it returns the base involved. Otherwise, it returns -1 indicating a failure.

The final stage is to modify the growth rule to be more efficient. To do this, we use a module that indicates that exponential growth has taken place. The growth rule is replaced by the following parameterised rule:

$$X \mapsto \text{exp("X", base, getGeneration())}$$

exp is a custom module that represents the large sequence of consecutive symbols formed by an exponential growth rule. The first parameter is the symbol that is growing, X, the second is the base of the exponential and the third refers to the generation at which the growth rule first started. *getGeneration* is a custom function that returns the

Algorithm 16 Constant-base Exponential Case

```
function detectExponential(rule)
    if length(rule.rhses) ≠ 1 then return -1
    for i = 1 → length(rule.rhses[0])
        if rule.rhses[0][i] ≠ rule.strictPredecessor then return -1
    return length(rule.rhses[0])
end function
```

generation at which the rule was instantiated. Using this information, it is possible to replicate an entire sequence after several iterations, while only using a single module.

5.3.2 Binomial Case

The binomial case arises when stochastic rules are mixed with growth rules. Such rules have the following form:

$$\begin{aligned} X &\mapsto^{a} X \\ &\mapsto^{b} X\ X \end{aligned}$$

Algorithm 17 Binomial Case

```
function detectBinomial(rule)
    if length(rule.rhses) ≠ 2 then return -1
    for i = 1 → length(rule.rhses[0])
        if rule.rhses[0][i] ≠ rule.strictPredecessor then return -1
    for i = 1 → length(rule.rhses[1])
        if rule.rhses[1][i] ≠ rule.strictPredecessor then return -1
    a = 0, b = 0
    if length(rule.rhses[0]) = 1 and length(rule.rhses[1]) = 2 then
        a = rule.rhses[0].probability
        b = rule.rhses[1].probability
    else
        b = rule.rhses[0].probability
        a = rule.rhses[1].probability
    end if
    return b/(a + b)
end function
```

The binomial case is so named from the way it grows. Each item in the growth sequence has a probability, b of producing two items and a probability, a, of remaining unchanged. This results in a more complicated reconstruction procedure, as probabilities must be accounted for in the final interpretation.

While X is allowed to vary, the size of both right-hand sides is fixed. The two right-hand sides must have length one and two, respectively. At each generation, the output

of symbol lengths obeys a binomial distribution. The binomial distribution's associated probability, P, is calculated from the relative probabilities of the right-hand sides, a and b as follows:

$$P = \frac{b}{a+b}$$

As with the exponential case, the left-hand sides of each rule must be checked to guarantee that X is not present. Pseudocode to detect the binomial case is given in Algorithm 17.For the binomial case, we use the following replacement rule:

$$X \mapsto \text{binomial("X", probability, getGeneration())}$$

Like exp, *binomial* is parameterised to make it possible to recover the entire sequence. The first parameter indicates the symbol that is being grown and the final parameter, as before, is the starting generation. The second parameter, however, is the binomial distribution's probability, P, calculated above.

5.3.3 Interpretation

In both cases, the sequence is reconstructed during final string interpretation. In the case of the exponential case, the number of symbols is:

$$L_{exponential} = B^{(T-G)}$$

where B is the base, T is the total number of generations, and G is the initial generation.

An iterative approach is used to compute length in the binomial case. At each generation the length must be known in order to calculate subsequent generations. The following recurrence is used:

$$\begin{aligned} L_{binomial}(0) &= 1 \\ L_{binomial}(N) &= L_{binomial}(N-1) + bin(L_{binomial}(N-1, P) \end{aligned}$$

where P is the distribution probability and *bin* is a mathematical function that returns a sample from the binomial distribution. Writing an efficient function to sample from a binomial distribution is a non-trivial task, but, fortunately, many efficient implementations are available[7]. Our system uses a fast implementation from the C++11 random number library, which operates in constant time. The desired sequence length is computed as $L_{binomial}(T-G)$.

5.3.4 Further Optimisation

This optimisation provides more benefits from using modules to convey growth than simply shortening each string. Depending on the symbol being grown, it may be unnecessary to reconstruct the entire sequence. An example of this is the F symbol, which can be made to take a single parameter indicating the length of the cylinder to construct. Instead of using many separate Fs during interpretation, our system uses a single F

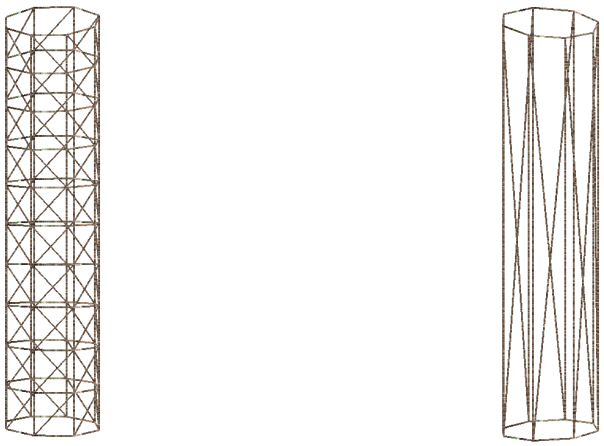


Figure 5.4: Two trees segments that have been created without and with growth optimisation respectively. The main trunk area can be seen to use considerably fewer cylinders, hence, triangles. The effect is less visible as the bush grows, which is due to the branches becoming smaller and requiring fewer cylinders to represent them naïvely.

with a larger than regular length. Building very tall (in comparison to the base) cylinders has the problem of creating 'sliver' (small, but tall) triangles. We deal with this by potentially cutting up the large cylinder into the smaller ones with a lower height to base ratio. Not only does this optimisation significantly reduce work during interpretation but it reduces the number of polygons required to represent the tree, thus making rendering more efficient. Figure 5.4 makes this clearer with a side-by-side comparison.

While it is possible to replicate the same effect for some other standard symbols, *F* is the most useful to optimise. For the symbols that cannot be optimised in this manner, the interpretation is looped by the number of symbols occurring in the sequence.

Generation	Fs	exps	Leaves	Triangles	Previous Triangles
5	243	120	120	6,168	10,488
6	729	363	363	18,561	33,009
7	2,187	1,092	1,092	55,740	102,108
8	6,561	3,279	3,279	167,277	312,477

Table 5.10: Complexity of the output string from the L-System described in 5.6 after optimisation. Here Fs and exps refer to their number of occurrences in the final string. The number of leaves remains unchanged by the optimisation.

Table 5.10 shows the results of applying the growth optimisation of F symbols to the L-System in Table 5.6. The results show that the amount of geometry required to represent the tree has dropped by approximately half. This improvement is made less dramatic by the geometry required to represent leaves which this algorithm does not influence.

5.4 Branch Optimisation

Our final optimisation deals with a second form of exponential growth in L-Systems: *branching*. Branching is a crucial aspect of tree growth, a tree without branches would look bare. While branches add to the realism of generated trees, they are responsible for a significant amount of geometry. This problem is exacerbated when creating large numbers of trees. Our system attempts to solve this problem through the application of *instancing*.

The technique of instancing refers to replacing geometry, or parts of geometry, with *instances* or clones. A clone is a copy of an already created object. Instancing is commonly used to reduce memory overheads, however, it is not without side effects. Some users may notice when too many instances are close together, which leads to a loss of believability and hence immersion. The amount of instancing is a fine balance between visual quality and resource constraints.

The branch optimisation component uses instancing to reduce the number of tree branches that are created. In the context of tree L-Systems, branches represent additional *recursive* work that must be performed. With the help of instancing, we essentially memoise the results of this recursive work so it can be used later. A static *instancing probability*, P, is used to control how often instances are used and is a percentage represented by an integral number between 0 and 100, where 0 indicates no instancing and 100 indicates full instancing.

Branch optimisation identifies segments of rules that branch and annotates them with extra instancing control symbols. First, each rule must be examined to see if it branches. Parts of these rules are identified as starting points for branches and control symbols inserted. Stochastic rules are then used to determine the choices of when to instance by embedding into the probabilities of the right-hand sides of the appropriate rules.

5.4.1 Rule Detection

Each rule is examined to determine if it contributes branches to the tree. It is difficult to accurately determine exactly which rules branch so a heuristic is used instead. Bracket symbols, [and], are a common indicator of branching and are used to separate out state changes.

When the left bracket is encountered, state information, such as position and orientation, is saved and restored when the right bracket is met. While this behaviour is useful in branching it is also applied to create leaves, as in Table 5.11, and other non-branching phenomena.

$$L \mapsto [\ \hat{}\ \hat{}\ -f\ +\ f\ +\ f\ -\ |\ -\ f\ +\ f\ +\ f\ ]$$

Table 5.11: A rule from an L-System used to draw a leaf. Brackets are employed but no branching occurs.

The heuristic uses brackets as an indicator of branching, but in order to filter out erroneous cases a further restriction is applied: the brackets must contain at least one

Figure 5.5: A leaf drawn from the L-System in 5.11.

non-terminal symbol. Non-terminal symbols are simply the strict predecessors on the left-hand side of each rule. While this method may still incorrectly identify rules as branching, it is significantly more accurate than just using brackets alone.

$$
\begin{aligned}
&X \mapsto [\ L\ ] \\
&L \mapsto \hat{}\ \hat{}\ -f + f + f - |\ - f + f + f
\end{aligned}
$$

Table 5.12: A modified version of the L-System in Table 5.11. The second rule is incorrectly identified as a branching rule by our heuristic.

Segments of the rules are identified as start and end points for the branch. The brackets and the symbols between them are tagged with an identifier. Branches represented by identical symbols share an identifier, the assumption being that the resultant geometry is the same. The identifiers are global in that they may be shared across different rules. Algorithm 18 shows the detection process. In the pseudocode, seenBranch and branchGUID return information about the branch currently in question. seenBranch returns true if it is identical to a previously seen branch and false otherwise. branchGUID returns the unique identifier of a previously seen branch. The function addBranch adds a branch to the global list of previously seen branches and returns its new unique identifier. createBranch creates a structure that packages the unique identifier and the branch symbol information for use later in the program.

$$
\begin{aligned}
&\mapsto F\ A \\
A \quad &\mapsto \overbrace{[\ \&\ F\ L\ !\ A\ ]}^{Branch1}\ /\ /\ /\ /\ /\ '\ \overbrace{[\ \hat{}\ F\ L\ !\ A\ ]}^{Branch2}\ /\ /\ /\ /\ /\ /\ /\ '\ \overbrace{[\ \&\ F\ L\ !\ A\ ]}^{Branch1} \\
F \quad &\mapsto F\ F \\
L \quad &\mapsto [\ '\ '\ '\ \hat{}\ \hat{}\ \{\ -f + f + f - |\ -f + f + f\ \}\ ]
\end{aligned}
$$

Table 5.13: A leafy tree L-System from The Algorithmic Beauty of Plants[54].

5.4.2 Rule Modification

The rule modification process is again complicated by stochastic rules, and so we first describe the modification with deterministic L-Systems. Each rule's right-hand side is

Algorithm 18 Branching RHS Detection.

```
function detectBranchingRHS(rhs, nonTerminals)
    output = []
    for s = 1 → length(rhs)
        if rhs[s] = [ then
            t = s + 1
            for t → length(rhs)
                if rhs[t] = ] then break
            if t = length(rhs) then continue
            for u = s + 1 → t
                if rhs[u] in nonTerminals then continue s
            guid = -1
            if seenBranch(s, t) then
                guid = branchGUID(s, t)
            else seenBranch(s, t) then
                guid = addBranch(s, t)
            output += [branch = createBranch(guid, s, t)]
    return output
end function
```

replaced by several right-hand sides (meaning that the resulting rule becomes stochastic) depending on the number of branches that occur. If B branches are present on the right-hand side, 2^B right-hand sides are created, representing the possibilities of either instancing each branch or not.

If a branch is to be instanced, the relevant symbols are replaced by a *getInstance* module. If a branch is not instanced, other control modules, *startInstance* and *stopInstance* are inserted instead. These two modules demarcate segments of a string that correspond to branch information. Each of these modules take exactly two parameters: an identifier and an age. The identifier is the same as the one associated with the branch in the rule detection phase. The age, as with the growth optimisation, is determined by the *getGeneration* function.

Each right-hand side is given a probability, p, based on the instancing probability and the number of branches being instanced, which is calculated as follows:

$$p(I) = P^I \times (1 - P)^{B-I}$$

where P is the probability of replacing a branch with an instance, B is the total number of branches and I is an index variable. For each right-hand side, the index variable is the number of times that the decision is made to instance a particular branch.

A simple binary number counting algorithm[41] is used to enumerate these rules that is both efficient and easy to implement. The algorithm is only suitable if the number of branches in a rule is less than the machine's word size. This is a reasonable assumption, even for older 32-bit machines as 32 branches would correspond to $4,294,967,296$ different right-hand sides, an amount large enough to overflow most computer's memory.

Table 5.14 shows the optimisation's effect on the 'A' rule from Table 5.13. The disadvantages of this optimisation are evident: the number of right-hand sides has greatly increased and each is significantly less humanly readable.

Stochastic L-Systems add complexity that the several right-hand sides may create branches. The above algorithm is performed on each original right-hand that contains branching segments. The relative probabilities of each group of newly created right-hand sides must reflect the original distribution. To enforce this, the equation for p is modified:

$$p(I) = P_{original} \times P^I \times (1 - P)^{B-I}$$

where $P_{original}$ is the probability of the originating rule. Multiplying by the original probability ensures that the probabilities are in the correct distribution.

The amount of time required to apply this optimisation to a set of rules depends on the number of branches in each right-hand side, B_i, and the length of each right-hand side, L_i. The total computation and memory cost is bounded by $O(\sum 2^{B_i} L_i)$. The main culprit for the inefficiency of this algorithm is the number of right-hand sides that are created. In our investigations, B_i is rarely larger three and is thus not currently a problem. It may be possible to achieve the same effects using fewer right-hand sides, however, this is left as future research.

5.4.3 Interpretation

The *getInstance*, *startInstance* and *stopInstance* modules are used during interpretation to communicate with the higher levels of the system. getInstance indicates that the system should record the current position and orientation as places where an instance should be used to complete the tree. startInstance notifies the system that all the geometry created between the current symbol and the corresponding stopInstance symbol all form a coherent instance that can be used as parts of other trees.

The system uses these control symbols during interpretation for two types of instancing: *Static* and *Dynamic Instancing*. Static instancing more closely follows the stochastic nature of L-Systems in deciding when to instance. If the L-System randomly (via rules) decides to apply an instance, the system will follow accordingly.

Dynamic Instancing is performed as a 'last resort' in the case that sufficient system resources are no longer available to create a tree during run-time. This arises when the L-System rules decides not to instance, but the creation process fails midway through creating geometry for the tree. In this case, the state of the system is *rolled back* to the previously seen instance control symbols.

While both of these approaches are discussed in more detail in Chapter 6, it is important to note that these new modules serve a dual purpose. However, we only discuss the L-System interpretation aspect of the symbols here.

The branching optimisation is unique among the optimisations developed here in that as more trees are made, more instances are collected and can be used for other trees. This makes the optimisation amenable for forest generation as memory constraints for the entire forest can be lowered considerably by maintaining instances in a forest-wide data structure. As more trees are created it collects more instances which can later be accessed. Details of this data structure, and other technical aspects of the instancing process, also discussed in Chapter 6.

The effectiveness of the branching optimisation depends on many factors. The most important being the instancing probability. An instancing probability set too high and repetition will become evident, while an instancing probability that is too low results in the amount of memory required for a forest becoming too large. In our system, we use an instancing probability that varies as more trees are created. This functionality is tied heavily to the rendering aspect of the system and will be discussed further in Chapter 6.

5.5 Summary

In this chapter we have discussed four optimisations techniques that can be applied to L-System strings. In order of complexity these are the *Identity Rule Optimisation*, *Orientation Optimisation*, *Growth Optimisation* and *Branch Optimisation*. In micro-benchmarks we have determined that this is also their relative order of effectiveness, from least to most effective, although exact results depend on the L-System used.

The identity rule and orientation optimisations modify the rules to improve inefficiencies in the L-System representation. The growth optimisation and branch optimisation are considerably more complex but allow the system to efficiently deal with two sources of exponential behaviour common in L-Systems. It is possible that some of these optimisations, especially the identity function optimisation, are dependent on the exact string creation algorithm used. For this reason, testing of these optimisation techniques is left to Chapter 7 of this thesis.

The next chapter, *L-System Renderer*, discusses the internals of the renderer used. In this chapter, we describe the processes of static and dynamic instance and how they are used to limit the memory required for L-System rendering.

$A \mapsto^{p(3)}$ getInstance(1, getGeneration()) / / / / / ′ getInstance(2, getGeneration()) / / / / / / / ′ getInstance(1, getGeneration())

$\mapsto^{p(2)}$ getInstance(1, getGeneration()) / / / / / ′ getInstance(2, getGeneration()) / / / / / / / ′ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration())

$\mapsto^{p(2)}$ getInstance(1, getGeneration()) / / / / / ′ startInstance(2, getGeneration()) [^ F L ! A] stopInstance(2, getGeneration()) / / / / / / / ′ getInstance(1, getGeneration())

$\mapsto^{p(1)}$ getInstance(1, getGeneration()) / / / / / ′ startInstance(2, getGeneration()) [^ F L ! A] stopInstance(2, getGeneration()) / / / / / / / ′ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration())

$\mapsto^{p(2)}$ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration()) / / / / / ′ getInstance(2, getGeneration()) / / / / / / / ′ getInstance(1, getGeneration())

$\mapsto^{p(1)}$ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration()) / / / / / ′ getInstance(2, getGeneration()) / / / / / / / ′ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration())

$\mapsto^{p(1)}$ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration()) / / / / / ′ startInstance(2, getGeneration()) [^ F L ! A] stopInstance(2, getGeneration()) / / / / / / / ′ getInstance(1, getGeneration())

$\mapsto^{p(0)}$ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration()) / / / / / ′ startInstance(2, getGeneration()) [^ F L ! A] stopInstance(2, getGeneration()) / / / / / / / ′ startInstance(1, getGeneration()) [& F L ! A] stopInstance(1, getGeneration())

Table 5.14: The effects of the optimisation on Table 5.13.

Chapter 6

L-System Renderer

The renderer, Figure 6.1, is split into two sections: *Geometry Creation* and *Tree Rendering*. Geometry creation handles the commands from the interpretation phase of the L-System and stores the result to various geometry buffers, while the tree rendering phase draws the contents of these buffers to the screen. The renderer uses the geometry creation module in order to translate drawing commands from the L-System Evaluator to renderable geometry. Modifications are made to the standard rendering process in order to facilitate our special instance rendering. These changes are outlined in the Instance Cache section.

6.1 Geometry Creation

Trees are created in the system in two distinct phases: *Hero Creation* and *Tree Placement*. Hero Creation runs the L-Systems in order to create geometry. In this case, a 'hero' refers to geometry that is created for use as instance geometry. In other words, they serve as the template geometry for tree and branch instances. Although each hero created is a bonafide tree in its own right, they are not used directly for rendering purposes. Instead, they are treated as templates for other trees. The actual creation of trees for rendering is left to the Tree Placement phase. Tree Placement constructs new renderable trees from these templates by cutting and joining parts of the hero trees together and calculating the necessary transformations to and place the trees on the terrain. To facilitate these processes, the *Instance Cache* and *Tree Generator* components are used.

The Instance Cache stores and retrieves tree data geometry based on its metadata, which identifies tree instances by species, age and other criteria. This cache is necessary for the *Branching Optimisation* when the *getInstance*, *startInstance* and *stopInstance* symbols are encountered during interpretation. The Instance Cache also controls both the rate at which instancing is performed and when Static or Dynamic Instancing is applied.

The Tree Generator, on the other hand, contains the logic necessary to construct trees from the Instance Cache data itself.

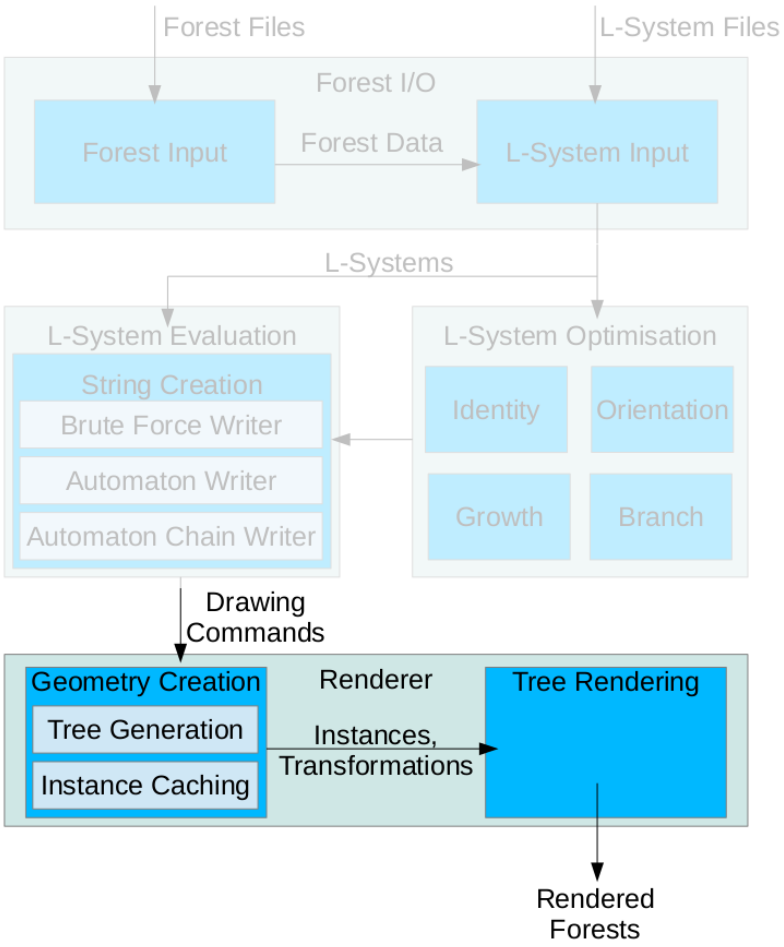


Figure 6.1: The above diagram shows the rendering component. Drawing commands in the form of L-System strings are sent to the Renderer from the L-System Evaluator. The output of this stage is rasterised geometry in the form of rendered forests.

6.1.1 Instance Cache

The end result of the interpretation phase is geometry. This geometry is separated into two groups for the purposes of rendering: vertex data, stored in vertex buffers, and index data, stored in index buffers[72]. Vertex information represents the trunks, branches and leaves of the plants. It is common for vertices to be inefficiently duplicated across different trees and even within the same tree. To save space index buffers are used. Index buffers store indices, essentially pointers to vertices. Instead of duplicating many vertices (approximately 50 bytes per vertex), one only needs to duplicate the indices (4 bytes per index).

All vertices for all template trees are stored in a massive global vertex buffer. The size of the vertex buffer is fixed at the start of the program to either 32MB (approximately 700,000 unique vertices) or 128MB (approximately 2,800,000 unique vertices) depending on the memory constraints being simulated. These sizes were chosen to represent different scenarios that our system could be used in. On modern video hardware we are often limited to around 1GB of video memory. Modern games (or other

graphically intensive simulations) are required to fit all of a scenes resources in this space in order to achieve best performance. These resources can include geometry data (positions, colours, normals, etc) textures and animation. In modern games, the majority of space is used for anti-aliased and anisotropically filtered textures[20]. The next largest resource is geometry data. In our thesis, we generate forests under normal circumstances (128MB cache size) and severely restricted circumstances (32MB cache size).

The indices for the template trees are stored in specially annotated index buffers. The cumulative size of all index buffers is effectively limited to roughly 20% the size of vertex buffer. To simplify the rendering stage there is a unique index buffer associated with each material used in the L-System. A material[23] is a collection of graphical properties for geometry, for instance which textures, colours and shaders to use. The leaves, for example, have a different material to bark and different types of bark all have different materials.

Aside from storing different data there is an important distinction between vertex buffers and index buffers. Vertex buffers are unordered, while index buffers are ordered. The vertices for a single tree may be scattered throughout the vertex buffer; vertices for a tree are not re-inserted if they are already present. Index buffers, in contrast, retain their data in the order in which they were created during the interpretation phase.

This means that indices for a particular branch of the tree will occupy a contiguous range of the index buffer and, by using the Branch Optimisation symbols, we can tag these ranges with appropriate data. *startInstance* and *stopInstance* demarcate the begin and end of a range, respectively, while *getInstance* indicates an incomplete section of the tree where a pre-existing range of the index buffer should be rendered.

Each index buffer range also stores metadata about the range. The age, species type and branch identification information is stored in hash tables to allow for easy and efficient access later. The hash tables, themselves, contain vectors of pointers to these ranges, which allows for efficient random selection. The Instance Cache can consequently retrieve a random index buffer range for any set of criteria of age and species.

Figure 6.2 shows the annotated information for the index buffers. In this case the buffer contains only a single tree. The green segments indicate an entire tree. Red and blue bars are first-order branches and second-order branches, respectively.

In addition to storing the range, the transformation of geometry is retained. Each transformation is a matrix that represents the effective spatial orientation of the geometry in the index buffer. This information is necessary to correctly place the branch on a renderable tree.

Static and Dynamic Instancing

The Instance Cache can be set in one of two instancing modes: Static and Dynamic Instancing. Static Instancing refers to the instancing approach provided by the Branch Optimisation described in Section 5.4. Using Static Instancing, memory can be conserved by sharing pre-generated tree geometry amongst several trees. Dynamic Instancing allows the system to finish generating the current tree when the necessary resources (for instance, memory) are unavailable. Dynamic Instancing rewinds the current state of interpretation until a pre-existing instance can be used to complete the entire tree.

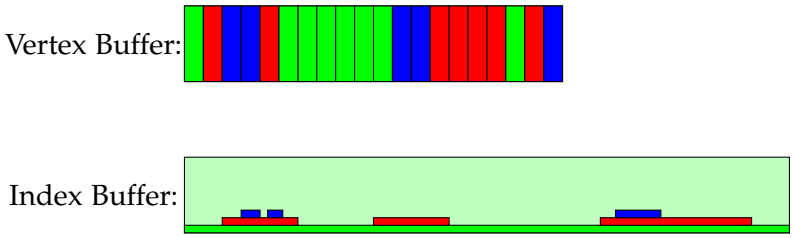


Figure 6.2: This is a pictorial representation of the Vertex Buffer and Index Buffer containing a single tree. Each cell in the Vertex Buffer represents a different vertex in the tree. These vertices are colour coded based on which level of the tree they come from. Green represents a vertex form the main trunk (level 0). Red indicates a vertex that belongs to a branch off the main trunk (level 1), and blue is a vertex that is from a subsequent branch (level 2). Although the vertex buffer only shows three levels, a larger tree would contain more levels. Each of these levels is annotated in the Index Buffer as well. In the case of the Index Buffer, the levels include each subsequent level as well (green contains level 0, level 1, and level 2, etc). Notice that the Vertex Buffer vertices appear out of order with respect to the Index Buffer. This is because the Index Buffer indices follow the order in which the tree is made, while vertices need not.

Static Instancing and Dynamic Instancing both provide solutions to memory issues that are encountered when creating large amounts of geometry for forests. In terms of exceeding memory bounds they can be viewed as the prevention and cure, respectively. Static Instancing prevents an out-of-memory situation by using progressively more and more precomputed geometry, while Dynamic Instancing cures an out-of-memory situation by re-interpreting previous symbols to incorporate more instancing.

By default, Static Instancing as provided by the Branching Optimisation is employed. The process switches immediately to Dynamic Instancing when there is not enough memory available in the vertex buffer to service the interpretation of the current L-System.

In Static Instancing, rules that branch are altered to reflect that, with a predefined probability, they will use pre-constructed branch instances from the Instance Cache instead of producing new geometry and thus saving memory. The probability of choosing a branch is called the *instancing probability* and can be specified as a parameter to the Branch Optimisation process, however these optimisations are typically performed once-off and in our implementation must occur before any Tree Heroes have been created[1]. This poses a challenge during Hero Creation where different instancing probabilities must be selected at different times.

Our solution to this problem is to have the Instance Cache precompute 101 optimised L-System rulesets, one for each integral percentage from 0% to 100%. This allows one to select any integral percentage without having to perform the optimisation during Hero Creation simply by selecting different rulesets from the array. Although performing all the optimisations and storing all the rulesets may seem wasteful, the

[1]Our implementation of the Instance Cache requires prior knowledge of the instance identifiers before use. Our design uses a fixed-size array, however it could easily be made to use a vector or other growable data structure.

optimisations are quick to evaluate and only consume approximately 1MB of memory per species, which is insignificant compared to the geometry memory requirements.

During Hero Creation, as the buffer fills up, it is important to alter the instancing probability upwards so that more trees may be added to the forest. It must not be so low that it limits the number of hero trees that are created, and it must not be so high that the hero trees exhibit less variation because they reuse branches too extensively. The Instance Cache balances these concerns by varying the probability linearly with respect to the amount of geometry in the cache. The probability can be calculated as:

$$P_{instancing} = \left\lceil 100 \times \frac{V_{current}}{V_{max}} \right\rceil$$

where $P_{instancing}$ is the instancing probability, $V_{current}$ is the current number of unique vertices in the vertex buffer and V_{max} is the maximum number of unique vertices in the vertex buffer.

While Static Instancing is designed to save memory by sharing geometry across multiple trees, there will inevitably be a situation when the geometry buffers become full and no more data can be added. Dynamic Instancing is used in such circumstances. Dynamic Instancing ignores any further requests from the interpretation stage to create more geometry. Instead, the interpretation is advanced to the completion of the current branch. This branch is then treated as if it were a *getInstance* symbol indicating that a pre-existing branch should be placed in the position.

All of the current branch's indices for that exist in the index buffers are deleted, but vertices are not deleted as they may be used in other trees. Finally, any subsequent *startInstance* symbols are replaced by *getInstance* symbols and the symbols in between it and the corresponding *stopInstance* are deleted. This results in a tree which contains a mixture of new geometry and missing entry points for pre-existing branches to be placed.

6.1.2 Tree Generator

The *Tree Generator* is a high-level component that utilises the Instance Cache in an appropriate manner to create renderable trees out of hero trees for Tree Placement. A renderable tree is represented by a collection of pointers to index buffer ranges that are stored in the Instance Cache.

Creating a tree from the instance cache is a recursive algorithm. The algorithm takes the species of the desired tree, its generation and a placement matrix describing the desired position and orientation, as input. Given this information, an instance is selected from the instance cache to serve as the base of the tree.

Each instance is not limited to representing a single generation. Instead, an instance could represent the entire tree, only a single generation or, more likely, several generations with exit points to fill with sub-branches. The exit points describe not only the desired position and orientation relative to the start of the instance but the desired age and branch identification of the instance that should fill the gap. Figures 6.3 shows the an example instance cache for a single tree.

The algorithm, Figure 6.4, recurses for each exit point that the instance requires. The age and branch information are used as parameters in order to constrain the subsequent

search of the instance cache. The instances are re-oriented by computing a transformation matrix. Given the desired orientation matrix, D and the orientation matrix of the instance within its hero tree, we calculate the transformation reorient an instance, T, as:

$$T = D \times M^{-1}$$

In other words, the transformation is calculated as the inverse of the instance's geometry transformation, multiplied by the desired transformation parameter. The initial desired orientation is passed as a parameter to the recursive function so it must be updated being passed onto the next function. In order to update the orientation we use the following formula:

$$D_{new} = D_{old} \times E$$

where D_{new} is the new orientation, D_{old} is D from above, and E is the orientation of the next exit point in relation to its hero tree.

Algorithm 19 Selection array building process.

```
function CREATETREE(outputTreeObject, desiredOrientation, treeGenerations, instanceCache)
    instance = instanceCache.getInstance(treeGenerations)
    transformation = desiredOrientation × instance.heroOrientation.getInverse()
    outputTreeObject.addTreeInstance(instance.indexBuffers, transformation)
    for i = 1 → length(instance.exitPoints)
        newGenerations = instance.exitPoints[i].exitGeneration
        newOrientation = desiredOrientation × instance.exitPoints[i].exitOrientation
        createTree(outputTreeObject, newOrientation, newGenerations, instanceCache)
end function
```

Algorithm 19 shows the tree creation process described above. The recursive function, *createTree*, takes several input arguments that describe the instance we should find. In this algorithm we only search the instance cache by generation, but in practice we use other criteria as well. The desired orientation is used to compute the transformation we need to correctly draw the index buffer. The *outputTreeObject* contains a list of index buffers that we should draw and their correcting transformations. The *addTreeInstance* method simply appends index buffers and transformations to this list. The recursive function terminates when all exit points have been filled.

Rather than applying the transformations to the geometry data, the transformation is instead stored so that the renderer can perform the transformation on the fly. This allows the geometry data to be efficiently reused across multiple trees and branches. The index buffer range and the required transformation is saved to the tree object for use with the renderer. The end result of this process is a collection of pointers to index buffer ranges and transformations to apply to correctly render the tree at a particular position.

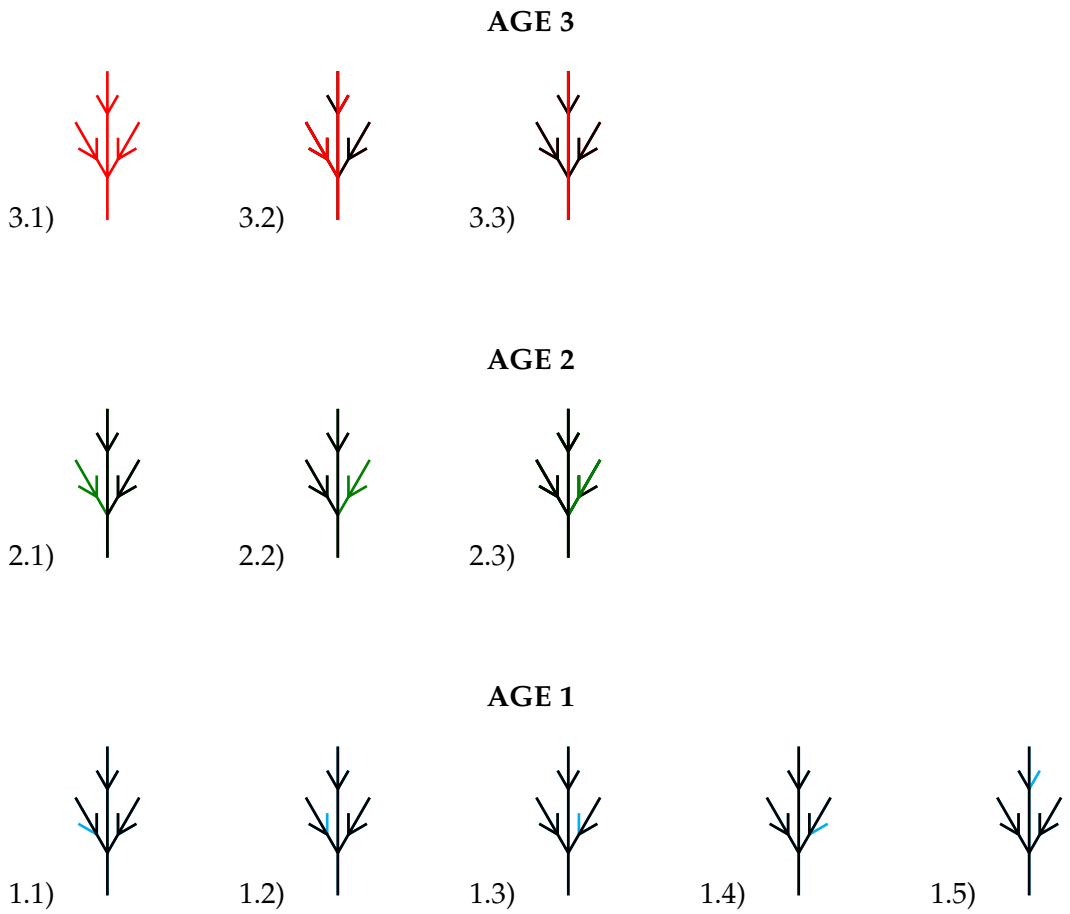


Figure 6.3: A representation of the geometry in the Instance Cache after Hero Creation, organised by age. For each tree shown above, Red(age 3), green(age 2), and blue(age 1) lines indicate the actual geometry and the location and orientation within the tree. Black areas indicate the remaining (unselected) sections of the Hero Tree from which the instance made. For the purposes of this example all Hero Trees are the same but in practice they vary.

Figure 6.4: This figure shows a renderable tree being constructed from the instance cache in figure 6.3. Each tree from left to right shows the tree at a various stages of construction. The first tree shows the selected base for the tree, entry 3.2, with two missing gaps for branches of age 1 and 2. Entry number 2.3 is chosen to fill the gap of age 2, which subsequently has a gap for an entry of age 1. Finally, the remaining gaps of age 1 are filled with entries 1.3 and 1.4, respectively. Before each entry is added, the a transformation is created to map from the entry geometry modelview coordinates to the worldspace coordinates of the new tree.

6.2 Tree Rendering

Tree Rendering is performed via OpenGL using the vertex and index buffers. To draw a single tree we first bind (send to the graphics card) all vertex and index buffers. The tree object, consisting of pointers to index buffers and index buffer ranges, is examined on a per material basis. We iterate over each material that is used in at least one L-System and render the index buffer ranges for the appropriate index buffers. Our implementation uses bulk draw calls such as *glDrawElements* that provide an efficient method for drawing large amounts of precomputed tree geometry. Finally, the buffers are unbound leaving the graphics card free to remove our vertex buffer from video memory and use it to render other objects such as terrain or character animation.

Rendering an entire forest is not vastly different. We keep the bind and unbind calls at the beginning and end of the entire process instead of per-tree, for efficiency reasons. The second change is that we draw all trees in parallel by drawing one material across all trees. This means that before we start drawing the leaves for a particular tree, the bark of all trees must have completely finished. This has no bearing on what the user sees and is only implemented to avoid expensive OpenGL draw calls such as extraneous glDrawElements calls or binding and unbind of textures and geometry.

6.3 Summary

In this chapter we have discussed how the rendering component communicates with the interpretation stage of the L-Systems to produce geometry. The resulting geometry is stored and annotated within the Instance Cache, which also controls the type of instancing and its probability.

The Tree Generator interacts with the Instance Cache when constructing renderable trees by randomly choosing instances that fulfil certain criteria like age and species. Finally, trees created by the Tree Generator can be efficiently rendered at any time via an OpenGL framework, using the vertex buffers and index buffers referenced through the Instance Cache.

Chapter 7

Testing

This chapter presents and analyses the results of each of the L-System acceleration methods. Testing was done on an Intel Core i5 2.80GHz quad-core machine with 8GB of RAM with an Nvidia 580GTX graphics card. To avoid giving preferential treatment, for instance if a background process instantiates a large block of memory during a test, we have limited the system to using exactly 4GB of RAM.

The experiments test the *Rule Representation* (Section 4.1), *Argument Replacement* (Section 4.1.2), *String Creation* (Section 4.2) and *L-System Optimisations* (Chapter 5) on both individual trees and forests of different sizes. A total of four experiments were performed to test each method. These experiments are detailed below.

7.1 Experiment Design

Our first two sets of experiments test the effectiveness of our modifications to the naïve Rule Representation and Argument Replacement methods. In these tests we determine the amount of memory, preprocessing time and speed up that is offered over the traditional methods. While the input we use for these tests do not correspond to (tree) L-Systems, they are nonetheless important in gauging the algorithm's impact. We test the second set of algorithms, String Creation and L-System Optimisations, using tree L-Systems as input. We split testing of these two optimisations into two parts concerning performance for a single tree and performance for forests of various sizes. For the purposes of these tests, we ignore the visual realism of the trees (and forest) in question as it is not the main focus of our work, however it is mentioned briefly in Section 7.6.

7.1.1 Experiment: Rule Representation

The representation of rules is important for the process of right-hand side selection. The selection process occurs once for each input symbol, a figure that can be very large. In mini-experiments, we determined that, for rules with many right-hand sides, the overall impact with using a naïve approach was on the order of several seconds. We test the two original methods, *Linear Roulette Wheel Selection* and *Binary Search Tree Roulette Wheel Selection*, our *Array Selection* technique for RHS selection. The results are evaluated in terms of their selection time, start-up time and memory requirements for different RHS sizes. L-Systems of sufficiently large RHS are generated for this experiment as tree L-

Systems with appropriate RHS sizes are difficult, if not impossible, to find. The test data does not correspond to regular tree L-Systems as we find it more convenient to generate the appropriate L-Systems for this experiment. The main purpose of this experiment is to determine the efficiency of our rule representation as the number of right-hand sides increase, which is independent of whether or not a tree L-System is used. It is important to test rules with very large numbers of right-hand sides as they are not impossible in our system. Using the *Branch Optimisation*, for instance, on a single rule that contains ten branches will create a rule with 1024 right-hand sides.

7.1.2 Experiment: Argument Replacement

This experiment tests the Argument Replacement methods *Search-and-Replace* against *Parameter Annotation*. Both methods have a trivial start-up time which we exclude from the results. We test their performance as the number of evaluated arguments increases. L-Systems with random numbers of differently sized modules are generated for use as input data since we are only concerned with running time versus number of arguments.

7.1.3 Experiment: Single Tree L-System Optimisation

We determine the efficacy of the String Creation algorithms and the L-System Optimisations on a selection of six tree L-Systems (of which more detail is provided in Section 7.1.5). We also measure the performance of the String Creation algorithms based on the average memory used and running time metrics.

In addition to the metrics used in String Creation, we intend to measure the increase in graphical performance. Measuring raw FPS (frames per second) in this case is a poor determinant of performance because, in terms of workloads, single trees do not stress the card enough. Most of time recorded is due to the overhead of initialising the render process on the GPU. For this reason, we use number of polygons as our performance metric instead. This number does not vary from run to run and is not subject to overhead. All L-System Optimisations are tested except for the Branching Optimisation, which is reserved for rendering entire forests.

7.1.4 Experiment: Forest L-System Optimisation

This experiment determines the results of using the Branching Optimisation when creating forests of various sizes. Unlike the previous experiments, we fix many of the parameters, the String Creation method and the other L-System Optimisations to their optimal values.

For this experiment, we maintain geometry cache of fixed size as detailed in detailed in Chapter 6. The cache is filled with the initial hero trees of the different species. As the cache fills up, the instancing parameter for the Branching Optimisation is increased. After each hero tree is added to the cache, the value is adjusted to the percentage of cache occupancy. Once the cache has been completely filled the trees are placed on the terrain. These trees are stitched together from different partial instances that were created from Branching Optimisation. The net result is a 'frankenstein' forest that does not contain any singularly instanced tree but rather a collection of trees that share branches from different hero trees.

We determine the total amount of time to create the forest, memory required to represent the forest geometry and other data structures, and frame rate in terms of both the size of the geometry cache and the size of the forest. This data is compared to the results of the same process when no instancing or Branch Optimisation is used.

7.1.5 Tree L-System Test Cases

Six L-Systems were used as input for the Single Tree L-System and Forest L-System experiments. These L-Systems are based on L-Systems found in the *Algorithmic Beauty of Plants*[54]. The rules for these L-Systems can be found in Appendix C, but we briefly describe each of them here.

1. This L-System, shown in Figure 7.1a, is a tree based on a bush design found in Algorithmic Beauty of Plants. The L-System itself contains three simple rules for creating branches, growing branches and drawing leaves, respectively. The branches are created by splitting the main branch into three and are lengthened in an exponential fashion.

2. A fern-like tree (Figure 7.1b) consisting of only two rules. The branching and growth is contained in a single rule and a second rule provides the mechanism for drawing leaves. Most symbols in this L-System are the result of branching symbols rather than branch growth.

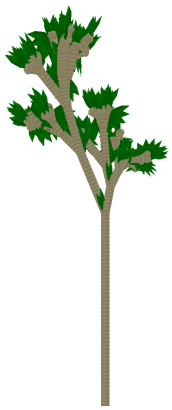
3. A complicated stalk-like tree with offshoot branches coming off the main stem. Each consecutive offshoot is placed further up the tree and at a 137.5 degree angle to the previous branch. This angle is common in natural leaf and branch arrangements. The branches grow in a binomial fashion. This tree is shown in Figure 7.1c.

4. This L-System, shown in Figure 7.1d, is an extended version of our first L-System, and contains additional rules with complicated LHSes in an attempt to increase the amount of foliage created, while decreasing the amount of branch geometry. The growth rule has also been changed to have slightly different growth characteristics. Instead of growing straight, trunks and branches following a complicated oak-like path.

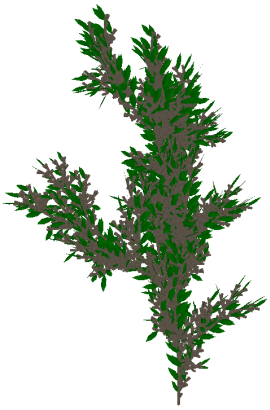
5. A deterministic, non-realistic tree L-System that contains only two rules for branching and exponential growth, respectively. The generated trees resemble binary trees with an attached trunk, Figure 7.1e.

6. This L-System, Figure 7.1f, is a stochastic version of L-System 1 with a different growth function. Instead of growing exponentially or in a binomial fashion, the L-System contains a set of rules that ensure that growths occurs proportional to $O(\sqrt{N})$.

We determine the adequacy of the L-Systems used here in terms of the criteria put forward by Aono and Kunii[4]. Specifically, these are phyllotaxis (the arrange of the leaves or branches around the main stem), apical dominance (main trunk should be visible) and shape (appearance should be natural). The rulesets that we have chosen



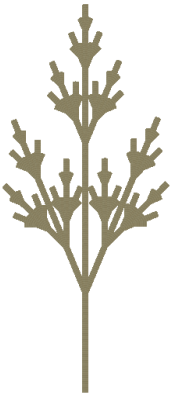
(a) *ABOP Tree*



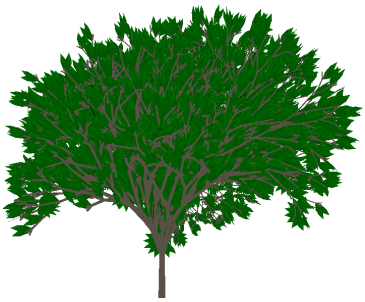
(b) *Fern*

(c) *Stalk Tree*

(d) *Extended ABOP Tree*



(e) *Bifurcation*



(f) *Sqrt-Growth*

for our test cases, except for test case 5, each show different aspects of these features. Finally, each of these rulesets have differing complexities. There are two L-Systems which have very simple rulesets, test case 2, which contains only one rule for branching, and test case 5, which contains one rules for branching and one rule for growth. Test case 4 was an attempt to introduce more complexity into an existing ruleset, test case 1, without adding bogus (non-functional or non-useful) rules to the L-System. A rule in the L-System attempts to detect when too much branch geometry is present and removes, placing more leaves in that position. Finally, there is a mixture of rules with conditional guards and expressions. The most complicated ruleset is test case 6, with non-trivial rules to achieve $O(\sqrt{N})$ growth. The average size of the tree L-System is about five rules, which is not unusual for tree (stochastic, parameterised and context-sensitive) L-Systems.

7.2 Rule Representation

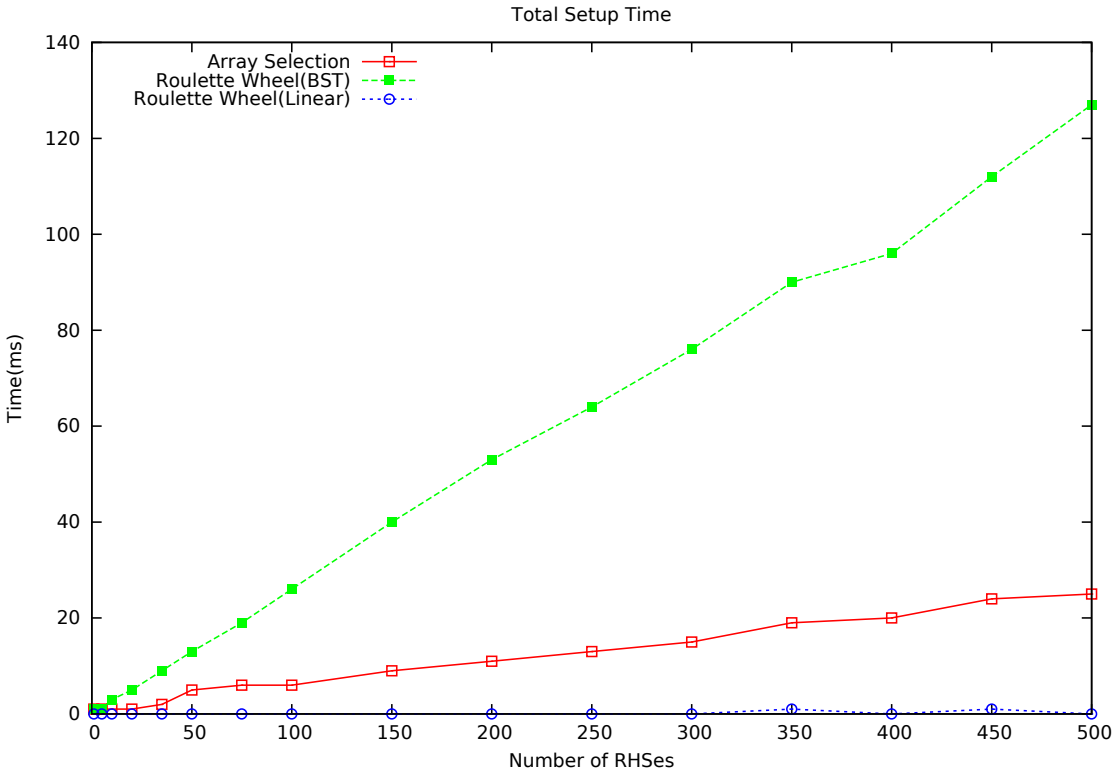


Figure 7.1: Time required to create the data structures to perform each of the selection methods, as RHS size increases. Red represents the Array Selection method while green and blue represent the Roulette Wheel Selection method with and without using a tree respectively. In terms of setup time, our Array Selection spends less time constructing the data structures than the Binary Search Tree Roulette Wheel Selection method, but more than Linear Roulette Wheel Selection.

This section shows the results of the different rule representation schemes. These

techniques structure the right-hand sides of rules such that random selection of right-hand sides is efficient. Figure 7.1 shows total setup time as RHS size increases. We run the experiments up to 500 to clearly RHSes to show a clear separation of the tested techniques. Typically, there are not more than 30 RHSes in a tree L-System. This is the time required to create data structures to accelerate the selection process. Since Linear Roulette Wheel Selection does not require any data structures its setup time is always zero. In several places in the graph, the Linear Roulette Wheel Selection deviates from the x-axis but this just noise from the time measuring process.

Binary Search Tree Roulette Wheel Selection requires the construction of a decision tree. The construction process is non-trivial and the resulting structure contains many fields, such as the probability of choosing the left child over the right child, designed to facilitate the selection process. Array Selection, on the other hand, only constructs an array whose contents are easily computed from the rules.

Although both Binary Search Tree Roulette Wheel Selection and Array Selection require $O(N)$ start up time, the constant factor of the Array Selection is significantly lower, approximately one fifth that of Binary Search Tree Roulette Wheel Selection.

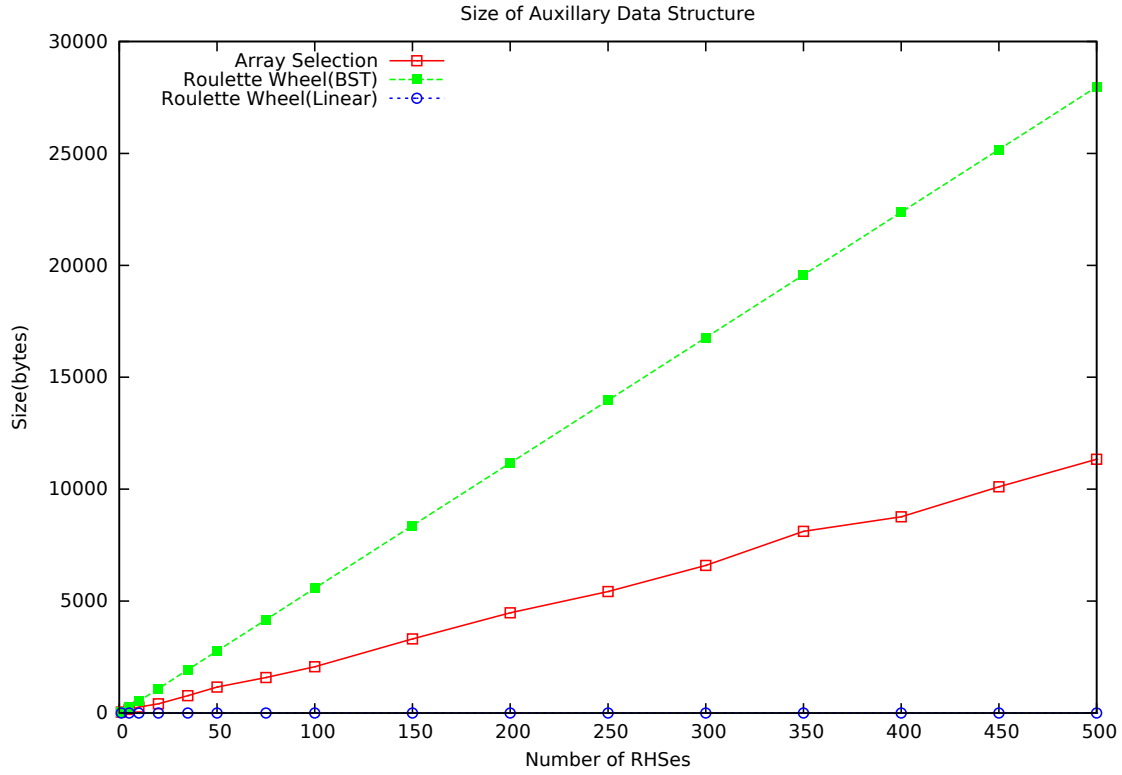


Figure 7.2: Space required to represent additional data structures. Red represents the Array Selection method while green and blue represent the Roulette Wheel Selection method with and without using a tree, respectively. As with the setup time, our Array Selection uses less memory than the Binary Search Tree Roulette Wheel Selection method, but more than Linear Roulette Wheel Selection.

The graph in Figure 7.2 shows the size in bytes of the data structures as RHS size increases. The size of the Binary Search Tree Roulette Wheel Selection can be calculated

as an exact multiple, sixty-four bytes[1], per element of the RHS size. Array Selection does not have such a direct relationship. Its size is dependent on the relative probabilities of each element in the RHS.

However, Array Selection has a much lower memory footprint than the Binary Search Tree Roulette Wheel method, due to its simplicity. It is worth noting that the probabilities that occur in L-Systems tend to be very simple; selection often involves probabilities such as 33%, 50% and 66%. These probabilities are handled using very little memory in Array Selection. Less common probabilities, such as 12.34%, are not likely to be handled as well by Array Selection. One reason for the common probabilities is that L-Systems are usually made by hand and people tend to favour simpler probabilities.

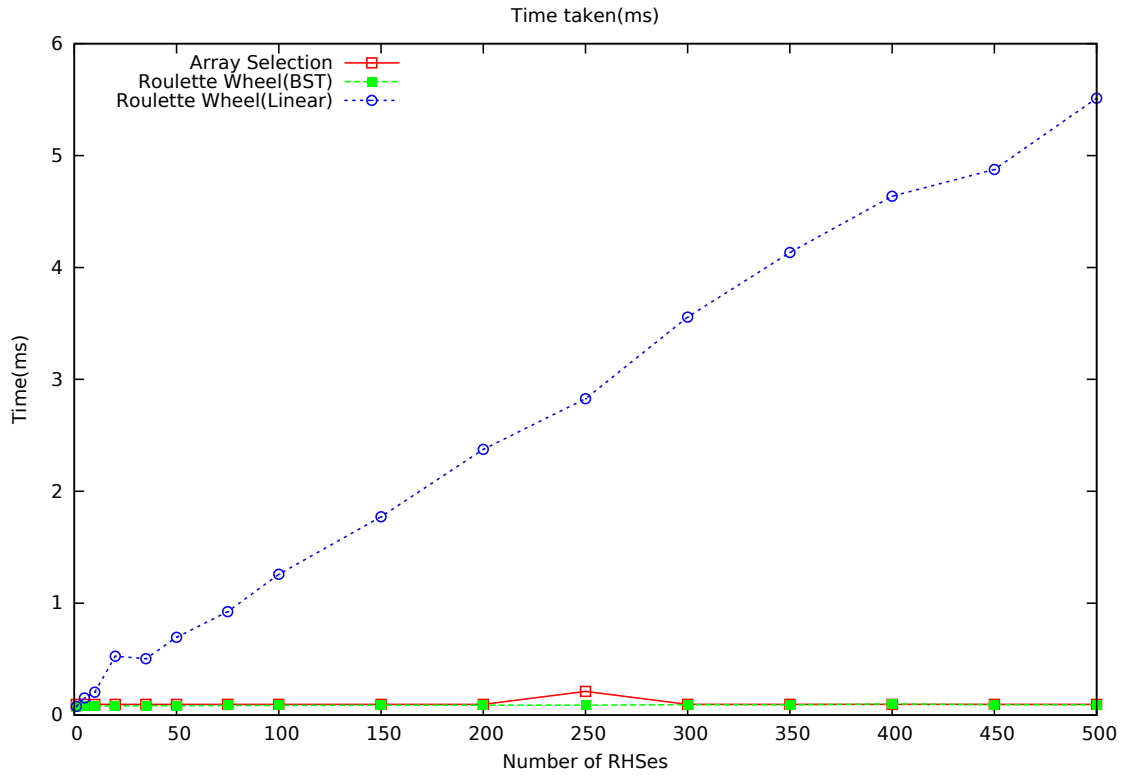


Figure 7.3: Total running time of the selection process as the size of the RHS increases. Red represents the Array Selection method while green and blue represent the Roulette Wheel Selection method with and without using a tree, respectively. The running time of our Array Selection method is roughly equivalent to the Binary Search Tree Roulette Wheel Selection method and considerably better than the non-tree version.

Finally, the running time of competing selection processes is shown in Figure 7.3. A single data-point represents the time required to perform 1,000 random selections. Each technique was given the same set of random selections to perform.

The graph shows that, although it has no setup time or additional memory usage, Linear Roulette Wheel Selection performs poorly. As the name implies, the method

[1]On a 64-bit machine.

takes roughly linear time to perform selection. If all RHSes are small, about one or two elements, Linear Roulette Wheel Selection is approximately the same speed as other methods.

Binary Search Tree Roulette Wheel Selection and Array Selection have roughly equivalent running times for almost all RHS sizes. The graph shows discrepancies at an RHS size of 250, but we attribute this to noise in the data collection process. The individual selection times for these techniques was within 10 microseconds of each other.

At RHS sizes of about 300 the Array Selection methods overtakes the Binary Search Tree Roulette Wheel Selection method. At 300 Array Selection is faster by approximately one percent. Array Selection has a theoretical performance of constant time, while Binary Search Tree Roulette Wheel Selection runs in logarithmic time. For this reason it is expected that for larger RHSes, Array Selection will still run faster, however, noticeable differences might only be present at unreasonably large RHS sizes. In our system, most rules have only one or two right-hand sides but certain rules, especially those with respect to growth or branching, have significantly more than that. For instance, in one of our L-System test cases there is a branching rule with seven right-hand sides: three to growth 1 branch offshoot, three to grow 2 branch offshoots at a single point, and one to grow 3 branch offshoots at 120° angles. Applying the Branch Optimisation to this rule yields a single rule with 23 right-hand sides which, although large for manually created L-Systems, is still quite small.

7.3 Argument Replacement

In the following section we provide the results of the two competing argument selection schemes: the traditional Search and Replace algorithm and the Parameter Annotation method. Figure 7.4 shows the running time of both methods as the total size of a module expression increases. The length of a module expression is measured in number of variable references in all modules. The graph shows that the running times increase linearly with the number of variables. The running time measures both the time to evaluate the expressions and compute their results. This explains why the running time of the Parameter Annotation method grows linearly instead of the predicted constant time.

The graph shows that Parameter Annotation is generally faster than the naïve Search and Replace method even for L-Systems with small module expression length. While the Parameter Annotation method is faster, it needs a special 'get' function. Optimal implementation of this function requires that the system has random-access to the parameters. Since Search and Replace does not have this requirement, the system designer is free to use linked-lists and trees for the parameters.

7.4 Single Tree L-System Optimisation

The results for this section are split up into four parts. First, we present the results for each of the String Creation algorithms (Section 4.2) without any L-System Optimisations added. This is followed by the running time and space requirements of the L-System Rule Optimisations(Chapter 5) on the L-Systems themselves. The results of

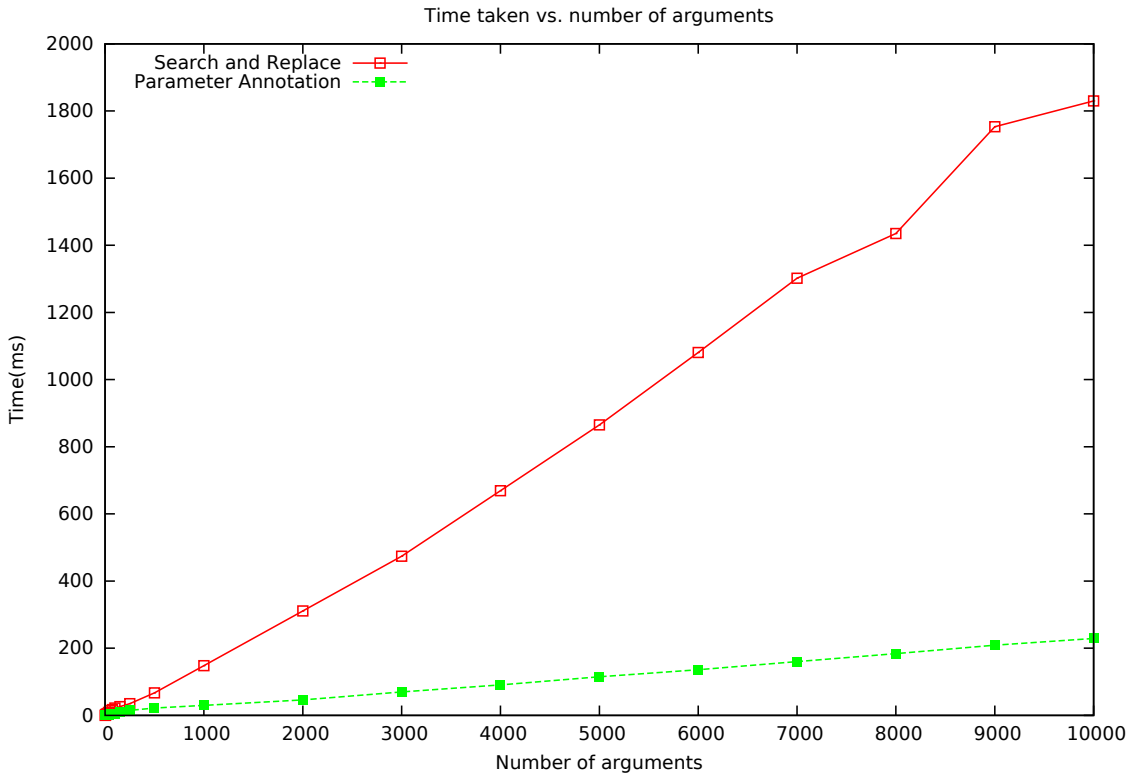


Figure 7.4: Time taken to evaluate a module expression as a function of its length. The steep red line indicates the running time of the Search and Replace method and, while the relatively flat green line shows the running time of the Parameter Annotation method.

the L-System improvements to the string creation process follow, and finally, we discuss the results of applying the L-System Optimisations on improving the time requirements to entire trees and its affect on final polygon count. In each test, we use both big and small trees created from each L-System discussed earlier as small cases may give different results to large cases.

7.4.1 String Creation Algorithms

The String Creation process is one of the largest components in the creation of trees. It not only requires significant time, but also creates a lot of intermediate data which eventually becomes the tree itself. We measure the performance of our own String Creation algorithms in terms of both speed and memory compared to the brute force implementation.

Table 7.1 shows the comparative results of the our novel Automaton and Automaton Chain algorithms against the naïve string creation algorithm. The Automaton algorithm shows the most overall improvement in time over Brute force, on average about 25%. The large *Sqrt-Growth* test case shows the best improvement of 40%. The main reasons for this improvement is the complexity of the contexts of the rules and the number of rules present. The worst test case for the Automaton method is the *Bifurcation* tree

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	207(100%)	170(82%)	290(140%)
Fern(small)	786(100%)	642(81%)	1000(127%)
Stalk Tree(small)	109(100%)	87(79%)	181(165%)
Extended ABOP Tree(small)	179(100%)	135(75%)	284(158%)
Bifurcation(small)	17(100%)	16(97%)	68(398%)
Sqrt-Growth(small)	4344(100%)	2985(68%)	4640(106%)
ABOP Tree(large)	3247(100%)	2545(78%)	4655(143%)
Fern(large)	31978(100%)	25893(80%)	36469(114%)
Stalk Tree(large)	214632(100%)	153809(71%)	238845(111%)
Extended ABOP Tree(large)	9728(100%)	7243(74%)	13065(134%)
Bifurcation(large)	188(100%)	172(91%)	344(183%)
Sqrt-Growth(large)	72438(100%)	43748(60%)	71235(98%)

Table 7.1: The total time required, in milliseconds, to perform string creation using each of the discussed string creation methods. The first number in each cell indicates the average run-time to create the strings, while the number in brackets is the percentage run-time when compared to the Brute Force implementation. A number less than 100% indicates that the method is faster than Brute Force, while a larger number indicates a slower implementation. The percentages are calculated from the full figure in microseconds rather than the milliseconds presented here.

which, in both the large and small cases, improves by less than 10%. This is due to simplicity of this test case (only two rules and no contexts present).

The worst performing approach is the Automaton Chain algorithm. This algorithm performs, on average, 25% worse when compared to Brute Force, and, in some cases, almost four times worse. The Automaton tends to better on the larger test cases than the smaller test cases. This is mainly due to the complexity of the algorithm, which contains greater overhead (mainly recursive and data structure overheads) due to its lazy nature. The algorithm fares poorly in *Bifurcation* due to the Brute Force algorithm being particularly efficient at processing the test case.

Table 7.2 shows the average memory requirements during the string creation process. The Automaton Chain algorithm outperforms the other methods in almost every test case. The lazy evaluation allows it to function without storing more than a few small portions of the string. Both the Brute Force algorithm and the Automaton algorithm require that up to two generations of the string be kept in memory, which dominates the memory requirements. The Automaton Chain algorithm performs less well on the small test cases. This is because less memory is required to represent them with the Brute Force and Automaton methods, while the majority of the Automaton Chain's memory requirements come from overhead which are less dependent on the size of the tree being generated.

The Automaton algorithm performs the worst in terms of memory requirements because it not only needs to keep string generations but also an automaton that represents the rules. For this reason, it can never be as memory efficient as Brute Force. On the other hand, the size of the automaton is relatively small, only a hundred bytes big on average, so the lost efficiency is not large.

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	26(100%)	26(100%)	10(37%)
Fern(small)	144(100%)	144(100%)	5(3%)
Stalk Tree(small)	9(100%)	10(100%)	10(100%)
Extended ABOP Tree(small)	19(100%)	19(100%)	5(30%)
Bifurcation(small)	<1(100%)	<1(101%)	1(188%)
Sqrt-Growth(small)	911(100%)	911(100%)	17(1%)
ABOP Tree(large)	499(100%)	499(100%)	15(3%)
Fern(large)	5215(100%)	5215(100%)	7(<1%)
Stalk Tree(large)	17438(100%)	17438(100%)	24(<1%)
Extended ABOP Tree(large)	914(100%)	914(100%)	11(1%)
Bifurcation(large)	22(100%)	22(100%)	2(10%)
Sqrt-Growth(large)	15040(99%)	15040(100%)	22(<1%)

Table 7.2: The total space required, in kilobytes, to perform string creation using each of the discussed string creation methods. The first number in each cell indicates the average memory required to create the strings, while the number in brackets is the percentage memory when compared to the Brute Force implementation.

7.4.2 Effects of Optimisations on L-System Rules

	Identity Function	Growth	Orientation	All
ABOP Tree	353	222	329	1018
Fern	313	129	273	758
Stalk Tree	428	250	366	1126
Extended ABOP Tree	476	228	526	1144
Bifurcation	149	96	80	331
Sqrt-Growth	475	232	374	1143

Table 7.3: A table of the average time(in microseconds) required to perform the optimisations for L-System.

Table 7.3 and Table 7.4 shows, for rulesets, the time required to perform each L-System optimisation and the subsequent percentage size increases, respectively. As can be seen, the optimisations do not require much more than a few milliseconds. The most time-intensive optimisation to perform is the Orientation optimisation due to the number of computation required to compress the L-System rules. The most space-intensive optimisation is the Identity optimisation due to increase in the number of rules. The *Fern* and *Bifurcations* cases are the quickest to perform L-System Optimisations on. We attribute this to the fact that both of these L-Systems only have two rules in them, while the rest have at least five.

The Growth optimisation is the quickest to perform. This optimisation spends the majority of its time, applying the heuristics to the L-System to determine the growth rules. If none of the heuristics match, for instance *Fern*, *Extended ABOP Tree*, and *Sqrt-Growth*, the optimisation does not perform any work. The L-Systems which are not altered by Growth can also be seen in Table 7.4 where there is no change in overall size.

	None	Identity Function	Growth	Orientation	All
ABOP Tree	628(100%)	1448(230%)	734(116%)	764(121%)	2030(323%)
Fern	376(100%)	956(254%)	376(100%)	488(129%)	1232(327%)
Stalk Tree	752(100%)	1740(231%)	858(114%)	888(118%)	2322(308%)
Extended ABOP Tree	776(100%)	1792(230%)	776(100%)	1154(148%)	2576(331%)
Bifurcation	68(100%)	216(317%)	116(170%)	68(100%)	374(550%)
Sqrt-Growth	748(100%)	1736(232%)	748(100%)	884(118%)	2048(273%)

Table 7.4: A table of the percentage size(in bytes) increase of the rules when the optimisations are performed. The size of a rule is the sum of the number of symbols and parameters across each rule. If conditional guards or module expressions exist then they are taken into account as well. The figure in brackets is the percentage memory required when compared to performing no optimisations.

Finally, the time to perform all of these optimisations consecutively is around one to two milliseconds. This is important because in the forest generation requires that L-Systems optimisations are performed many times. For example, in our system, we keep one hundred copies of L-System rules for each instancing probability which means we apply the above optimisations one hundred times per L-System.

Although the running time is dependent on the order that the optimisations are performed, the maximum difference was calculated at around two percent. One thing that can be seen from the table is that the running time of each optimisation does not sum to the running time of performing all of them consecutively. The reason for this is that the running time of subsequent optimisations will increase as a consequence of the added rules.

The biggest size increase is seen in *Bifurcation* with the Identity optimisation which is roughly three times of the unoptimised size. This is due to the relatively small number of rules present in the L-System (two). The number of rules more than triples, in this case, after the Identity optimisation. The lowest size increase is seen in *Fern*, *Extended ABOP Tree*, and *Sqrt-Growth* for Growth optimisation. *Fern* does not contain any rules for growth and instead relies on continual branching to add more geometry. *Extended ABOP Tree* and *Sqrt-Growth* contain growth rules that do not match any of the growth heuristics used. *Extended ABOP Tree* grows in a pseudo-exponential fashion (exponential but with some added delay that causes it not to match the heuristics). *Sqrt-Growth* contains several complicated rules that exhibit growth equivalent to $O(\sqrt{N})$, which is, again, not detected by the heuristic.

The overall increase in size is between 250% and 550%. Although this may be viewed as a drastic relative increase, the average absolute size of an L-System ruleset in memory is still only a few kilobytes. The largest absolute size of an optimised L-System occurred for *Extended ABOP Tree*, which was approximately two and a half kilobytes (2,576 bytes in Table 7.4). However, this is still dwarfed by the size of the trees' geometry and textures which are on the order of several megabytes in size.

7.4.3 Effects of Optimisations on String Creation

This section details the effects of the optimisations on the String creation process. We measure both the running time and memory required to perform string creation, how-

ever, we do not present the memory requirements results here. We found that most optimisations have a negative impact on the memory required for string creation, however, this has little effects in terms of forest generation. In our system, each tree is created sequentially. The memory that is used in the string creation for one tree is ephemeral in that it is recycled so it can be used for the next tree. On the other hand, the memory that is used for interpretation is kept, in the form of geometry and instance cache entries, for the entire execution of the program. For this reason, we focus only each optimisation's impact on string creation running time. We have, however, included optimisation effects on memory in Appendix D. It is possible that optimisations affect all of the three string creation methods differently, so we include entries for each string creation algorithm.

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	207(100%)	171(100%)	302(103%)
Fern(small)	786(99%)	648(100%)	1011(101%)
Stalk Tree(small)	107(97%)	80(92%)	168(93%)
Extended ABOP Tree(small)	325(181%)	137(101%)	298(104%)
Bifurcation(small)	15(90%)	16(100%)	67(98%)
Sqrt-Growth(small)	4991(114%)	2559(85%)	4944(106%)
ABOP Tree(large)	3260(100%)	2623(103%)	4641(99%)
Fern(large)	32160(100%)	25915(100%)	36549(100%)
Stalk Tree(large)	215362(100%)	152930(99%)	241169(100%)
Extended ABOP Tree(large)	10007(102%)	7302(100%)	13008(99%)
Bifurcation(large)	199(106%)	171(99%)	350(101%)
Sqrt-Growth(large)	72082(99%)	43779(100%)	72845(102%)

Table 7.5: The change in running time as a result of using only the Identity optimisation. The number outside the brackets indicates the absolute running time in microseconds. The number inside the parentheses indicates the change relative to having no optimisations performed.

Table 7.5 shows the difference in running time with the Identity Function optimisation turned on. As expected, the optimisation performs worst for Brute Force. This is because the Brute Force algorithm is sensitive to the number of rules in the L-System, which this optimisation greatly increases. Ignoring the small *Bifurcation* test case (because of its already low creation time), the optimisation only improves the *Sqrt-Growth* test, albeit by one percent. This case is already a poor performer for the Brute Force algorithm, with its many rules and complex left and right contexts. The other large test cases are not affected or have only minor performance losses.

The Automaton and Automaton Chain algorithms are not particularly influenced by the optimisation. Although the optimisation introduces many more rules, these string creation algorithms are not sensitive to the number of rules. The optimisation does not provide sufficient benefit, however, as the majority of large test cases improve by less than one percent. On the other hand, the small test cases show more improvement with the *Stalk Tree* improving by 7% to 8%.

Table 7.6 shows the difference in running time with the Growth optimisation turned on. It decreases performance almost universally. The optimisation appears to perform

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	215(103%)	185(108%)	313(107%)
Fern(small)	766(100%)	639(100%)	1017(101%)
Stalk Tree(small)	155(141%)	98(112%)	196(108%)
Extended ABOP Tree(small)	181(101%)	173(127%)	280(98%)
Bifurcation(small)	21(122%)	21(128%)	68(99%)
Sqrt-Growth(small)	4182(100%)	2792(93%)	4648(100%)
ABOP Tree(large)	3431(105%)	2649(104%)	4917(105%)
Fern(large)	32012(100%)	25912(100%)	36505(100%)
Stalk Tree(large)	229803(107%)	163813(106%)	251566(105%)
Extended ABOP Tree(large)	9815(100%)	7280(100%)	13104(100%)
Bifurcation(large)	248(132%)	214(124%)	412(119%)
Sqrt-Growth(large)	72738(100%)	44080(100%)	71448(100%)

Table 7.6: The change in running time as a result of using only the Growth optimisation. The number outside the brackets indicates the absolute running time in microseconds. The number inside the parentheses indicates the change relative to having no optimisations performed.

better with the Automaton Chain algorithm, however, even optimised, it is still significantly slower than the others. As stated before, the optimisation has no affect on the *Fern*, *Extended ABOP Tree*, and *Sqrt-Growth* test cases because the heuristics do not detect these cases. It does, however, show a large relative difference in *Extended ABOP Tree*(small), however, the absolute difference is less than 40 microseconds. we attribute this slowdown to the following two phenomena.

First, the optimisation introduces new modules into the L-System which add complexity. These modules require that their parameters be copied (the modules are not matched by any rule) making them more expensive than a single 'F'(tree cylinder) symbols. The module contains three parameters so our estimate for the best equivalent amount of processing is approximately four 'F' symbols. This is further compounded by the second observation. As the tree grows, the exponentially branching nature of the tree creates more branches which are shorter than the previous. This means that this optimisation will have a diminishing effect on string creation as the tree grows and can even degrade performance as complicated modules replace single F symbols. One way to test this hypothesis is to encode a cut-off depth that stops the optimisation from firing when it is too far away from the root. We leave the testing of this hypothesis as future work.

The results of using the Orientation optimisation are shown in Table 7.7. Like the Growth optimisation, it decreases performance almost across the board. The best performing cases is *Bifurcation*, however, upon closer inspection of its L-System we determine that it is not effected by the algorithm. Since the tree L-System draws a 2D shape, it never needs more than a single orientation changing symbol to get to the correct angle. The delta that is present in the *Bifurcation* optimisation is, therefore, noise, owing to its small size. The worst performing test case is the *Fern* test case which deteriorates by 15% to 22%. The *Fern* case uses small runs (approximately three symbols in size) of orientation changing symbols.

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	222(107%)	186(109%)	302(104%)
Fern(small)	963(122%)	749(116%)	1155(115%)
Stalk Tree(small)	119(108%)	99(114%)	180(99%)
Extended ABOP Tree(small)	224(125%)	158(116%)	310(109%)
Bifurcation(small)	15(88%)	17(101%)	69(101%)
Sqrt-Growth(small)	4647(106%)	3367(112%)	4936(106%)
ABOP Tree(large)	3453(106%)	2746(107%)	5114(109%)
Fern(large)	41173(128%)	33535(129%)	43372(118%)
Stalk Tree(large)	234655(109%)	170728(110%)	258617(108%)
Extended ABOP Tree(large)	10130(104%)	7673(105%)	13208(101%)
Bifurcation(large)	198(105%)	169(97%)	419(121%)
Sqrt-Growth(large)	76001(104%)	47349(108%)	75306(105%)

Table 7.7: The change in running time as a result of using only the Orientation optimisation. The number outside the brackets indicates the absolute running time in microseconds. The number inside the parentheses indicates the change relative to having no optimisations performed.

This could be due to poor fine-tuning of the optimisation; in our micro-benchmarks we determined that our *rotate* module was equivalent to a single orientation symbol in overhead. Another reason for the slowdown could be that copying of parameters, from one generation to the next, could be slower than the same task for symbols. This would lead to a problem where the optimisation would be faster at lower generations because less parameter copies would be performed. As before, more benchmarking is required to determine exactly where the problem occurs, however we leave this as future work.

7.4.4 Effects of Optimisations on Tree Interpretation

The following tables represent the effects of the L-System optimisations on the creation of individual trees. The Branching Optimisation is excluded from this section as it is more relevant to creation of forests rather than single trees.

Table 7.8 shows the performance improvement during L-System interpretation using different optimisations. These optimisations can only affect processing by altering the input string. Most optimisations have a very small effect on the time required to create tree geometry. The Identity optimisation, for instance, does not affect any aspect of the process as the input to interpretation remains the same. As can be seen from the table, most of the figures for the Identity optimisation are close to 100% with small absolute variations attributable to noise in the data collection process.

The Growth and Orientation optimisations, however, do have an effect on the creation time. As before the Growth optimisation does not affect the *Fern*, *Extended ABOP Tree*, and *Sqrt-Growth* test cases. The biggest change in time, 30% to 40%, occurs in the *Bifurcation* cases where the exponential growth rule is optimised to be performed in constant time and little else exists in the L-System. The remaining cases, *ABOP Tree* and *Stalk Tree*, are affected by the Growth optimisation but to a lesser extent, roughly 10% to 20%. This is due to rules for drawing foliage that are not present in the *Bifurcation*

	None	Identity Function	Growth	Orientation	All
ABOP Tree(small)	4(100%)	4(102%)	3(87%)	4(98%)	3(83%)
Fern(small)	54(100%)	54(99%)	54(100%)	54(100%)	54(100%)
Stalk Tree(small)	2(100%)	2(103%)	1(85%)	1(86%)	1(88%)
Extended ABOP Tree(small)	3(100%)	4(102%)	3(98%)	3(101%)	4(102%)
Bifurcation(small)	1(100%)	1(102%)	0(72%)	1(96%)	0(70%)
Sqrt-Growth(small)	190(100%)	190(99%)	196(103%)	186(97%)	192(100%)
ABOP Tree(large)	84(100%)	85(101%)	68(80%)	83(99%)	66(78%)
Fern(large)	2478(100%)	2475(99%)	2478(100%)	2454(99%)	2457(99%)
Stalk Tree(large)	3341(100%)	3368(100%)	2862(85%)	3289(98%)	2798(83%)
Extended ABOP Tree(large)	176(100%)	175(99%)	177(100%)	169(96%)	168(95%)
Bifurcation(large)	41(100%)	42(102%)	25(60%)	42(101%)	25(60%)
Sqrt-Growth(large)	3528(100%)	3542(100%)	3530(100%)	3501(99%)	3487(98%)

Table 7.8: The change in the total time required, measured in milliseconds, to perform interpretation using different optimisations. The first number in each cell shows the absolute time, while the bracketed number shows the relative change when compared to using no optimisations.

test case. The Orientation optimisation has a small impact on performance, approximately 4% in the best case (large case of *Extended ABOP Tree*). We exclude the small *Stalk Tree* test case as the absolute difference in time taken is negligible. The reason for the poor performance of the Orientation optimisation is that the majority of input to the interpretation process does not affect orientation. On the other hand, the Growth optimisation performs much better as it transforms a large portion of the input string from running in exponential time to constant time.

	None	Identity Function	Growth	Orientation	All
ABOP Tree(small)	1884(100%)	1884(100%)	1366(72%)	1884(100%)	1364(72%)
Fern(small)	22456(100%)	22456(100%)	22456(100%)	22456(100%)	22456(100%)
Stalk Tree(small)	628(100%)	628(100%)	564(89%)	628(100%)	564(89%)
Extended ABOP Tree(small)	1758(100%)	1758(100%)	1758(100%)	1758(100%)	1758(100%)
Bifurcation(small)	608(100%)	608(100%)	416(68%)	608(100%)	416(68%)
Sqrt-Growth(small)	53995(100%)	53995(100%)	53995(100%)	53995(100%)	53995(100%)
ABOP Tree(large)	32933(100%)	32933(100%)	22472(68%)	32933(100%)	22460(68%)
Fern(large)	808696(100%)	808696(100%)	808696(100%)	808696(100%)	808696(100%)
Stalk Tree(large)	862340(100%)	862340(100%)	667319(77%)	862340(100%)	667319(77%)
Extended ABOP Tree(large)	68374(100%)	68374(100%)	68374(100%)	68374(100%)	68374(100%)
Bifurcation(large)	21280(100%)	21280(100%)	11952(56%)	21280(100%)	11952(56%)
Sqrt-Growth(large)	871592(100%)	871592(100%)	871592(100%)	871592(100%)	871592(100%)

Table 7.9: The number of output polygons from the interpretation phase while using different optimisations. The percent change is recorded in brackets. Most columns show no change as they have no effect on polygon creating symbols in the L-System.

Table 7.9 shows the number of polygons that were created after each optimisation. This table is almost equivalent to the total amount of memory required during interpretation. The only difference is a relatively small amount of memory used per test case that includes the data structure that points to the various index buffers so that the tree can be rendered later. For this reason, we report only the number of total polygons which has a more direct application to rendering performance. As discussed before, we do not report frame rate figures as they do not show any discrepancy between the optimisations.

The Growth optimisation performs best on the large cases of *Bifurcation* (56%), *ABOP Tree* (68%) and *Stalk Tree* (77%). The small cases also the same relative improvements but to a lesser degree. The other cases are not affected at all by the Growth optimisation and, thus, do not present any changes. As stated before, the *Bifurcation* tree does not have any rules for drawing leaves which dilutes the effect of the optimisation.

Creating trees from L-Systems is memory intensive, so it is reasonable to assume that some of the savings in performance comes from using less memory. Comparing the performance increases from Table 7.8 to memory decreases in Table 7.9 we can see comparable numbers. In the three test cases that the Growth Optimisation is useful there is roughly a ten percent difference in performance over and above that of using less memory. We attribute this remaining performance increase to the fact that fewer symbols need processing to facilitate the growth functions.

7.5 Forest L-System Optimisation

The following tables show the results when running the system to create forests, which is the main purpose of this research. We use the six L-Systems discussed earlier as prototype trees. To decrease the size of the results, we have focussed on the most optimal settings determined from the previous sections. The String Creation algorithm used is the Automaton algorithm due to its speed advantage over the naïve and Automaton Chain algorithms.

In addition, the following two L-System optimisations were used: Growth optimisation and Branch optimisation. The Identity Function and Orientation optimisations were excluded from this test as they did not exhibit a significant beneficial effect on the string creation or interpretation time as a whole. The Branch optimisation is performed many times per tree L-System, as explained in 6.1.1, to give L-Systems with precomputed instancing probabilities.

We discuss the results of the forest instancing in terms of three scenarios. The first scenario pertains to the use of a small 32MB geometry cache for the Branch Optimisation, while the second has a much larger 128MB geometry cache. The third scenario is the result without any Branch optimisation in place. The size of forests we create range from 5 trees to 100,000 trees. For the sake of visual clarity we split this range up into small (5 to 10,000) and large (10,000 to 100,000) cases. While it is true that a user would have a difficult time distinguishing between 100,000 trees in a single scene, the same is not necessarily true for the forest as a whole as the viewer may pick up on slight differences between forests. We decided to base the 100,000 on the total number of trees a forest, rather than a single scene.

Finally, we measure the time for the creation of the forest as a whole instead of reporting the times for string creation and interpretation separately. The reason for this is that the use of instances can dramatically improve both string creation and interpretation times, depending on the instancing percentage used in the Branch Optimisation process. A more sensible metric is to use the creation time of the forest as a whole which is less sensitive to the amount of instancing used for a single L-System. We also present the memory requirements for the forest and the average frame rate achieved.

Figures 7.5 and 7.7 show the creation time in milliseconds for all three scenarios when creating small forests, while Figure 7.6 shows the first two scenarios when cre-

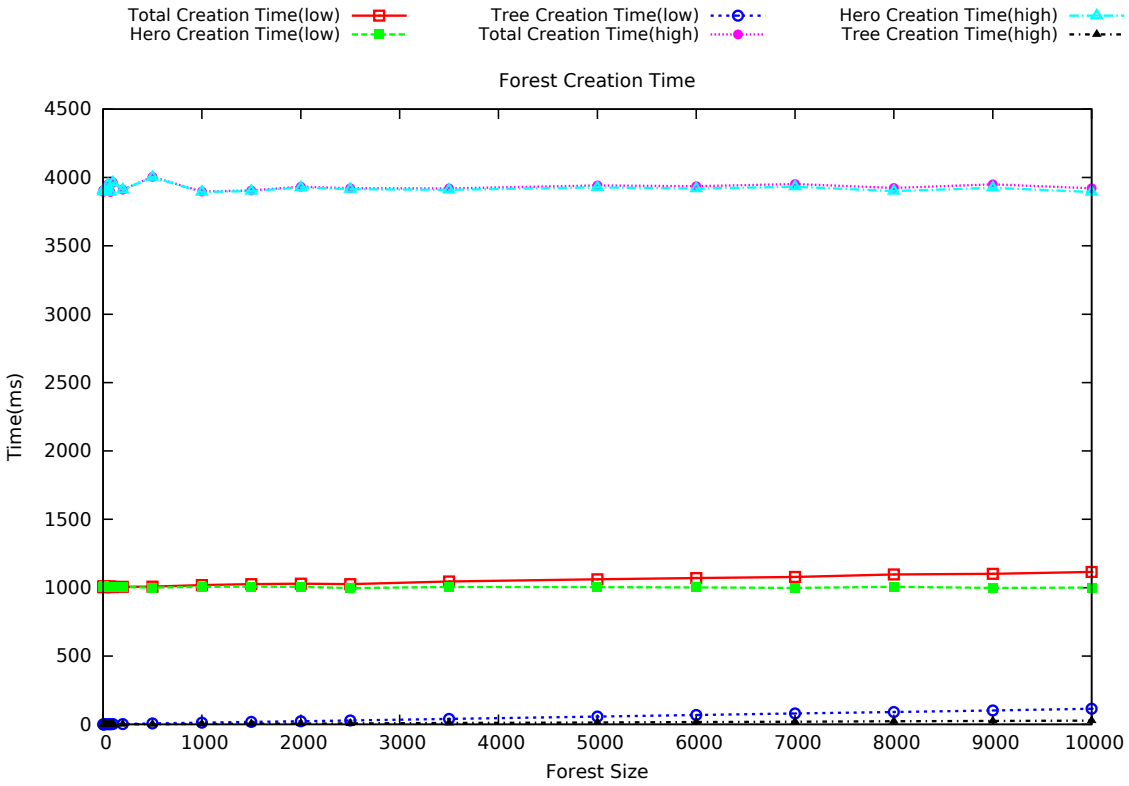


Figure 7.5: This graph shows the time(in milliseconds) required to the construct small forest sizes, as a function of the size of the forest. Red and pink lines show the total running time of the creation process for 32MB and 128MB, respectively. The red line is the sum of green line (hero creation phase) and dark blue line (instance tree creation phase) for the 32MB case. The pink line is the sum of the cyan line (hero creation phase) and yellow (tree creation phase) for the 128MB case.

ating large forests. Due to the high amount of memory required by the uninstanced approach, we are unable to obtain large forest creation time for the third scenario as our system runs out of memory on the machine.

The small and large forest graphs, Figures 7.5 and 7.6, are broken up into the setup costs (hero creation) and the costs to place the trees (tree creation). In both 32MB and 128MB scenarios, the majority of time is attributed to the once-off cost of computing the hero trees to fill the instance cache. At the medium sizes (around $10,000$ trees), approximately 90% of the time, on average, is spent computing the instance cache for the 32MB scenario, while this is approximately 95% for the 128MB scenario. We also notice that the hero creation phases for both scenarios correspond roughly linearly; the 128MB version is about four times slower than 32MB version for hero creation. We attribute this phenomenon to fact that roughly four times as much must be done to fill the instance cache. Finally, the graphs also show that the hero generation costs for both cache sizes are not dependent on the size of the forest, as expected. For the 32MB cache the running time of the hero creation is centred around 1 second, while the 128MB cache the average hero creation time is around 3.8 seconds.

The tree placement phases shows a linear increase in running time as the number

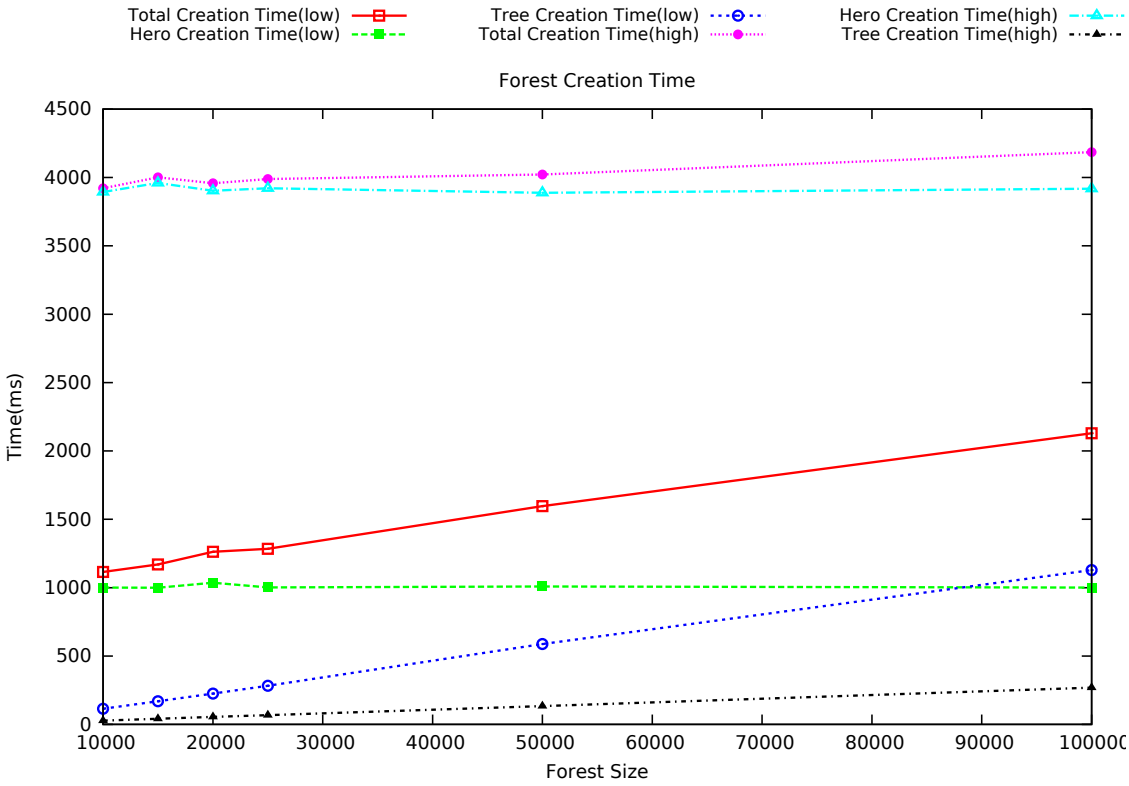


Figure 7.6: The time(in milliseconds) required to the construct large forest sizes, as a function of the size of the forest. Red and pink lines show the total running time of the creation process for 32MB and 128MB, respectively. The red line is the sum of green line (hero creation phase) and dark blue line (instance tree creation phase) for the 32MB case. The pink line is the sum of the cyan line (hero creation phase) and yellow (tree creation phase) for the 128MB case.

of trees increases. The curve for the tree placement for the smaller cache size has a much steeper slope than that of the larger cache. This is an artifact of the way that the tree placement algorithm works. The amount of work performed to place a single tree is related to the level of instancing; more instancing means more work to construct a single tree instance. An instance may contain many 'exit points' where other (smaller) trees must be placed.

These smaller branches may, themselves, contain exits which must be filled. The number of these exits is related to the instancing parameter in the Branch Optimisation; the higher the instancing, the higher the chance that an exit will need to be filled recursively. For smaller cache sizes, the instancing parameter for the Branch Optimisation increases much more rapidly, increasing the number of instances with exit and increasing the placement time. On the other hand, considerably less work must be done for larger cache sizes to build up the tree. As a result, the 32MB cache can compute trees at a rate of 100 trees per millisecond (0.1ms/tree), while the 128MB cache version can output 200 trees per millisecond (0.005ms/tree). This compares favourably to the texture lobe approach of Livny et al[35]. Their approach requires approximately 10ms per tree to construct, depending on the species.

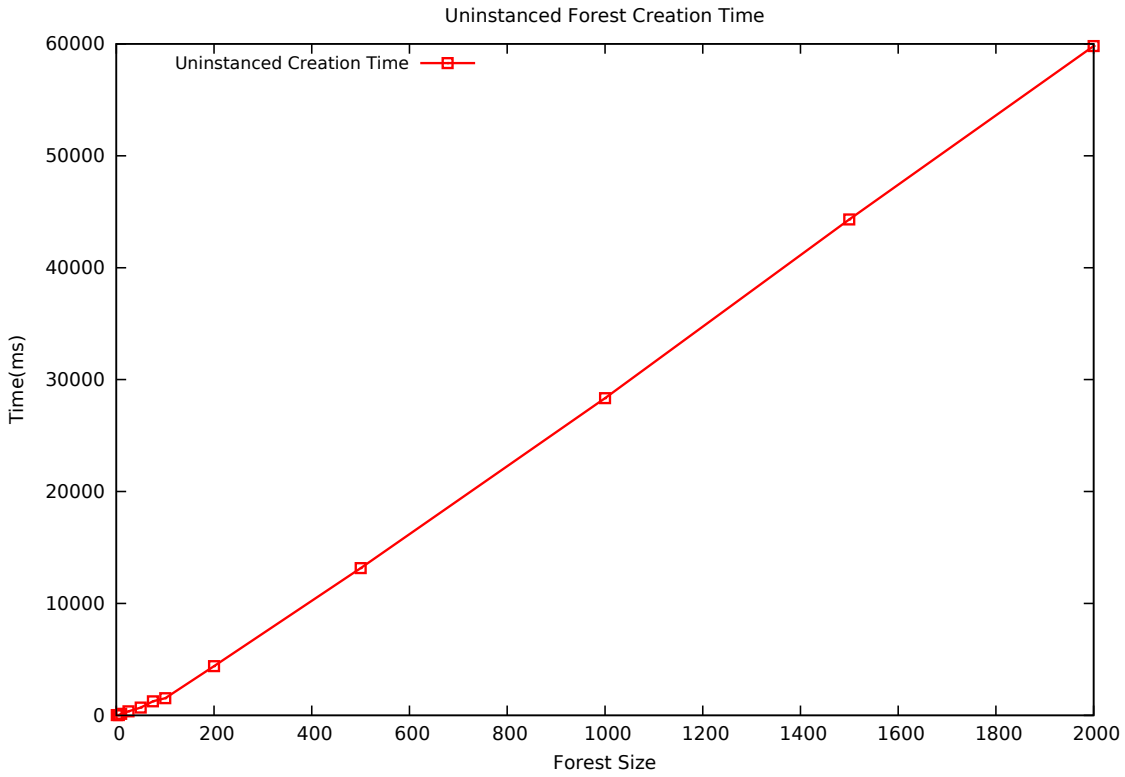


Figure 7.7: The graph shows the time (in milliseconds) required to construct a forest in our system without any instancing. Note that the axes of this graph differ from Figures 7.5 and 7.6. The program ran out of the memory on the machine, thus, we are only able to provide data up to 2,000 trees.

An interesting observation from the large forest graph, Figure 7.6, is that, at approximately 88,000 trees, the time required to create all the tree instances exceeds the time necessary to fill the 32MB cache, while instance creation for the 128MB version is still only 10% of the entire process. As a result, the total time graph for the 32MB cache version grows at a much faster rate than the 128MB version. It is possible that, at some stage, these two lines could cross over and the 128MB version become faster overall. More research is required, however, to determine if and when this occurs.

Comparing the total creation time of both cache sizes to the creation time for the uninstanced scenario shows a considerable speed improvement as forest size increases. The graph shows a total forest creation time that increases linearly at a rate of approximately 30ms per tree[1]. The uninstanced approach works better at lower forest sizes than both small and large caches. The small cache scenario becomes faster than the uninstanced case for a forest of size fifteen, while the larger cache is faster for a forest of around fifty trees.

Figures 7.8 and 7.9 show graphs of the memory required to represent a forest as its size increases in all three scenarios. The graph shows that, without any Branch

[1]Note that this time is not directly comparable to the to the results from tables prior to this section (especially Tables 7.1 and 7.8) as the forest contains trees from all species at different sizes, not only the small and large cases.

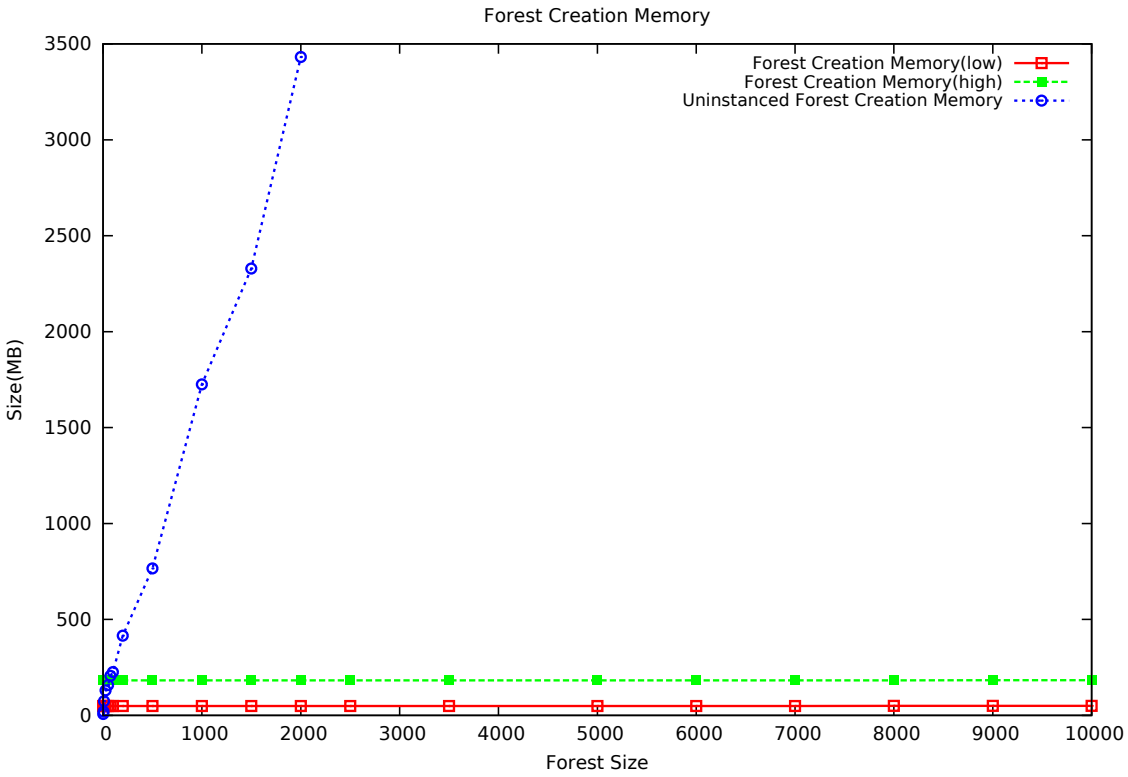


Figure 7.8: The amount of memory required to represent a small forest as a function of the number of trees in all three scenarios. No data is available for the uninstanced approach after 2,000 trees due to memory constraints.

Optimisation enabled, the amount of memory required grows linearly with the number of trees. At approximately 750 trees the memory requirements exceed 1GB, more than the amount of memory available on most modern graphics cards. For forests of 2,000 trees and greater the uninstanced approach consumes over 3.5GB which is more than many modern computers and can handle.

Although not using the Branch Optimisation works better for memory at lower forest sizes, the other scenarios quickly outperform it for sufficiently large forests. In terms of memory efficiency, the 32MB and 128MB experiments overtake the uninstanced version at sixteen and fifty trees respectively. One can also see that memory usage for the cache versions increase slowly as the size of the forest grows. This is due to the various structures that must be maintained, on a per-tree basis, to be able render the individual trees. The largest such structure that is maintained per tree is a list of index buffer pointers which must be used in order to render the tree. Stored with each pointer, there are two integers denoting the start and end positions that must be rendered. Although these structures are not usually more than a few kilobytes in size, with many thousands of trees their effects can build up.

The net result is that, excluding the cache, the average tree was represented in approximately 600 bytes (being just list of indices). However, owing to random selection, one could see a tree being represented in as much ten times that amount when many tree instances are used to construct a tree. This compares favourably when compared

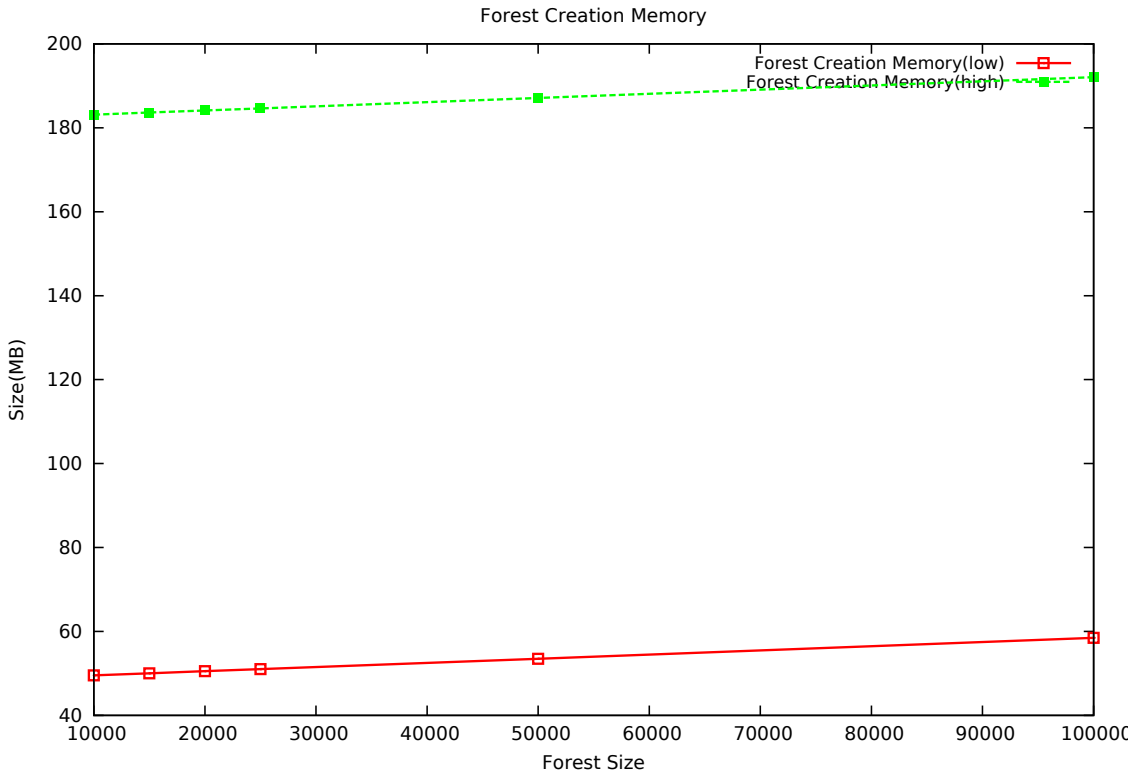


Figure 7.9: The amount of memory required to represent a large forest as a function of the number of trees in the 32MB and 128MB scenarios. No data is available for the uninstanced approach due to memory constraints.

to Livny et al's texture lobe approach[35] which is approximately 20 kilobytes on average (depending on the species). It should be noted, however, that the texture lobe approach can produce much higher fidelity trees than our current system as they are not constrained to using only L-Systems. Secondly, some of their storage is devoted to each tree's lobe textures which are a non-negligible amount.

The Figure 7.10 visualises the frames per second of the rendering as a function of forest size. The worst performer is the 32MB cache size version as rendering a tree could require a significant amount of CPU work, one draw call for each 'frankenstein' instance in the tree. Since the small cache size corresponds to higher average instancing probability per hero tree (hero trees created with low instancing will fill a larger proportion of the smaller cache, leading to a higher instancing probability for subsequent hero trees), we deduce that this is the reason for the roughly 15% difference between frame rates. At approximately $3,500$ trees the three methods become indistinguishable in terms of frame rate. The graph shows a roughly hyperbolic shape, which corresponds to the time per frame (inverse of frames per second) growing roughly linearly as forest size increases. $2,500$ is the point where the schemes stop rendering at interactive standards, achieving less than 24 frames per second.

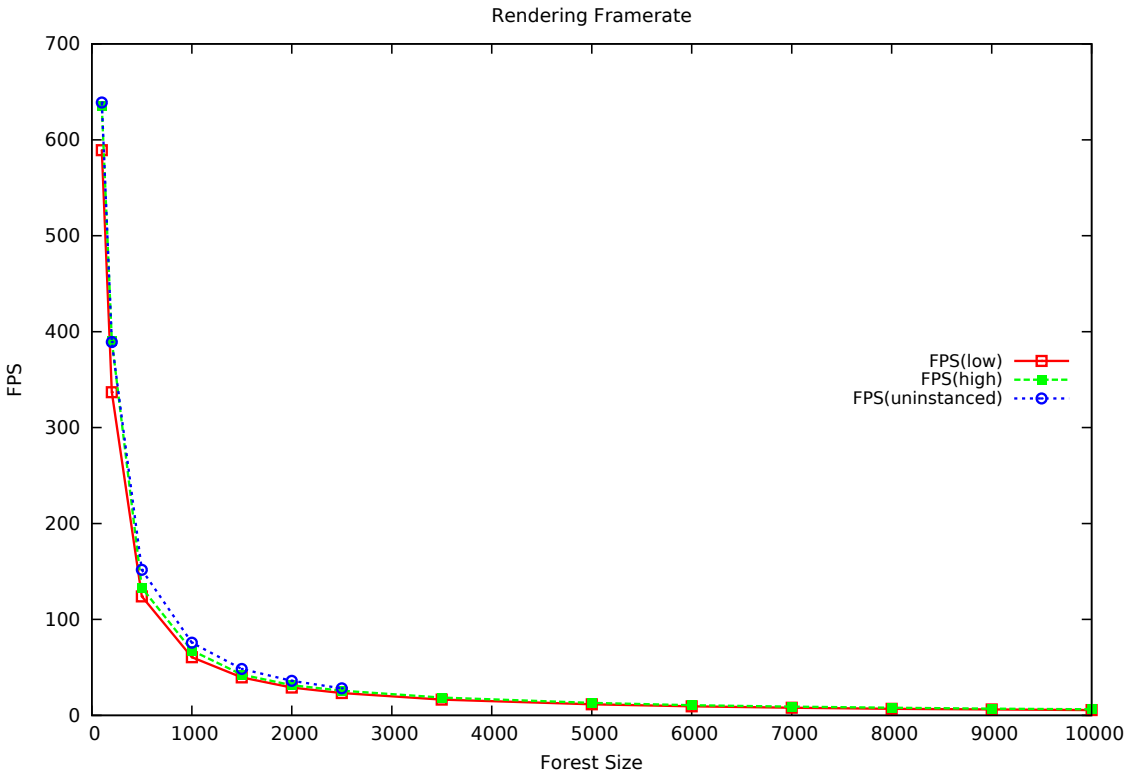


Figure 7.10: The above shows a plot of the frames per second for all three scenarios. Forests less than 25 trees and larger than 10,000 trees are not shown on this graph. Forests of size less than 25 render at over 1,000 frames per second, while forests of size greater than 10,000 render at approximately 1 frame per second. The 32MB cache size rendering is shown in red, 128MB in green and the uninstanced approach in blue.

7.6 Visual Artifacts of Instancing

Using instancing for procedural forest generation is not without its drawbacks. A major criticism that can be levelled at it is a reduction in the visual quality of the output. We attempt to reduce the effect on appearance by letting the non-deterministic nature of the L-Systems decided where instances should be placed. This can still lead to problems, however, in that the L-System could randomly decide two trees that are near to each other should use the same instances, or, even worse, be constructed entirely from the same instance. An example of the latter is shown in Figure 7.11.

Identical trees are unavoidable by the random nature of L-Systems, however, certain steps can be taken to mitigate the situation. The first is to use L-Systems which are very non-deterministic, in that they are able to produce a large number of varied trees. The second approach is to detect when we are about to place an offending instance, one which is too close to another instance of the same type, and replace it with a different one. This second approach is made more difficult by the random nature of L-Systems. The L-System could, for instance, decide to add multiple instances to the cache which all represent the same geometry. In this case we would not even be aware that we are using the same geometry when using different instances.

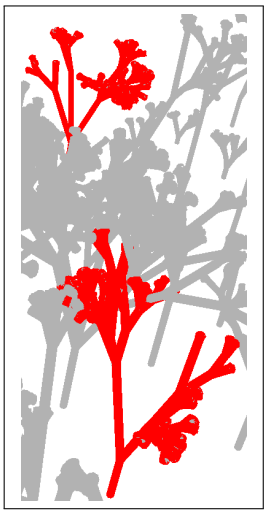


Figure 7.11: The trees highlighted in red are created from the same instance and used to represent the entire tree. The geometry of these trees is identical, however, a change in rotation of the instances causes them to look dissimilar.

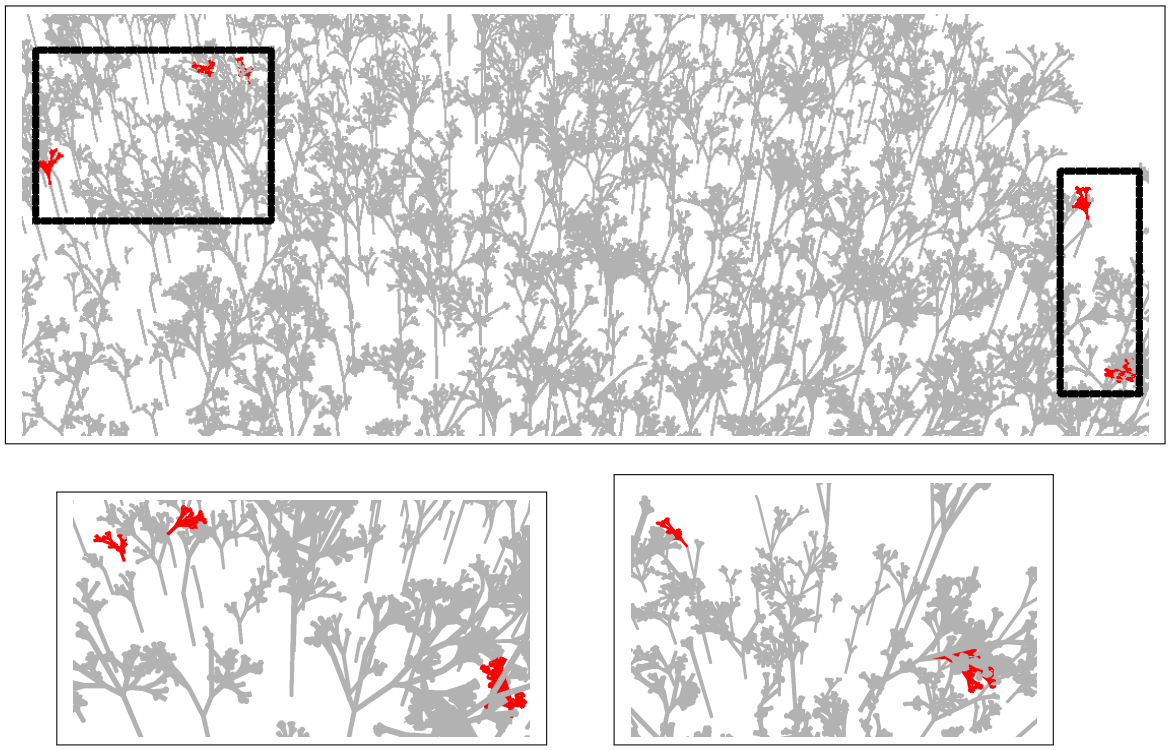
In order to correct this behaviour we would need to add mechanisms to the instance cache to detect when duplicate geometry is added and ignore it. This too is not an easy task because it would require us to perform complicated matching on the geometry and would surely slow the system down measurably. A simplifying assumption is that the same substring (the part of the string that represents the branch) represents the same geometry. Comparing strings instead of geometry is significantly easier (and more efficient). In either case, we defer a solution to future work.

The visual artifacts from instancing are not necessarily a problem in our system. Using informal tests (a user-guided fly-through of the forest), we determined that most users do not notice that instances exist. Identical trees that are nearby often go unnoticed if oriented at a random angle and branches that are identical can appear at different levels of the tree which masks their similarities. Figure 7.12 shows an example forest scene where a single instance has been selected and highlighted in red. The zoomed in areas shows show the instance used as both an entire tree (leftmost instance of top-left zoomed area) and as branches to other trees.

7.7 Summary

In this chapter we presented the results of our system. We showed the various methods for rule representation when dealing with stochastic L-Systems. Our *Array Selection* method performed at roughly the same speed as the next best while using less than half the memory. Our *Parameter Annotation* method for encoding the arguments directly into L-System expressions is significantly faster than the traditional method because it does not require any binding stage to occur. In this regard, we achieved an almost 90% speed up in the time required to evaluate an L-System expression in terms of the number of L-System arguments.

The next section showed the results of our string creation algorithms. The *Automaton*



(a) The top-left zoomed in region from the forest overview. Note that the orientation is different to make it clearer to determine the instance position and orientation in the larger trees.

(b) The bottom-right zoomed in region from the forest overview. Note that the orientation is different to make it clearer to determine the instance position and orientation in the larger trees.

Figure 7.12: This figure shows an area of a forest generated with our system. To show the effects of our instancing approach, a particular instance hero has been highlighted. The selected instance is shown in red, while unselected instances are shown in grey.

method outperformed the *Brute Force* method by 25%, on average, in terms of speed at the cost of a small amount of memory. On the other hand, the *Automaton Chain* method was able to get over 99% decrease in total memory at the cost of more time.

The L-System Optimisations that we tested had mixed success. The *Identity Function* optimisation and *Orientation* optimisation failed to provide an adequate decrease in string creation and interpretation time. The *Growth* optimisation, on the other hand, was successful in improving the interpretation and decreasing the number of polygons required to represent a single tree for certain test cases.

The main focus of our research, however, was the forest generation aspect of the system. Using the *Branch* optimisation, the system is able to effectively make very large forests without using enormous amounts of memory. The traditional, uninstanced method was only able to create $3,500$ trees in 60 seconds before exhausting memory. Using the *Branch* optimisation as our system does, however, we were able to create over $100,000$ trees in less than 4 seconds while using less than 1GB of memory. The majority of this time, 90% and above, is due to a once-off setup cost, after which new trees can be created quickly. At present our system can only render up to $3,500$ trees

simultaneously and remain at real-time speeds, however, we do little in the way of accelerating the rendering process. A drawback of our *Branch* optimisation approach is the way instancing is performed. Instances are selected at random from the instance cache, which means two instances could occur in close proximity. Overall, we conclude that the forest generation aspect of the research is successful in generating large forests with a lower memory footprint.

Chapter 8

Conclusion

The goal of our project was to accelerate large procedural forest generation. In this regard, we have successfully implemented a system for the creation of large forests from L-Systems. Our system demonstrates the utility of the optimisation algorithms for accelerating tree modelling.

The optimisations that we developed can be split into two categories: *String Creation* and *L-System Optimisation*. The String Creation algorithms are essential to producing a list of drawing instructions to create a tree and our novel methods are able to accelerate the process noticeably. The following techniques were tested:

- Brute Force: The naïve method when execution time is dependent on the number of rules, the size of each rule and the size of the input and output. The algorithm performs reasonably in some circumstances, however, as the number of rules (and their complexity) starts to increase, this algorithm's performance deteriorates quickly.

- Automaton: Recognising the process as being related to string processing, we create an automaton for the rules based on the Aho-Corasick algorithm[2]. This new algorithm improves the run-time complexity; run-time is now only dependent on the input and output sizes. In practical terms, this algorithms performs about 25% better than the naïve method.

- Automaton Chain: This algorithm builds on the efficient nature of the Automaton to create a lazy version of the String Creation process. This version computes output only as required. This decreases performance slightly, but uses almost no memory except for a few small buffers. In some of our test cases we achieve memory costs one hundredth that of the Brute Force method, while only running approximately 30% slower. The Automaton Chain has the same run-time complexity as the Automaton method but runs almost twice as slow. This makes it suitable for instances where memory is limited but computing power is abundant. An example of such a scenario is a embedded device (or mobile phone for instance) where we are limited by memory.

The L-System Optimisation operate on the list of rules that make up L-Systems. These optimisation attempt to address some of the undesirable behaviour that is present in many L-Systems (such as those with branching or exponential growth rules) without

affecting the variety and appearance of trees that can be created. A total of four L-System Optimisations were tested:

- Identity Function Optimisation: This optimisation adds rules to the L-System that help most String Creation algorithms make decisions earlier. Unfortunately, this optimisation only improves performance for the Automaton and Automaton Chain by less than 1%, which is not significant. On the other hand, it results in significantly worse performance for Brute Force.

- Orientation Optimisation: Often in L-Systems there can be a large group of symbols whose net result is to change the current orientation of a branch. This is not favourable as these symbols often remain identical but still require re-examination. This optimisation concatenates groups of symbols into a single symbol with the same net orientation. This algorithm does not improve the running time at all, which we attribute to the optimisation being difficult to correctly tune. On average, this optimisation results in a 10% slowdown for string creation but just under a 1% speedup for interpretation.

- Growth Optimisation: The Growth Optimisation attempts to address a common source of poor performance in L-System: branch growth. As the tree is constructed branches lengthen, often exponentially. This results in strings (the drawing instructions) that are extremely large consisting of many small segments of branch each treated as an individual symbol. This optimisation attempts to recognise the growth rules and represented it functionally in the L-System rather than with separate symbols. If the heuristics are able to model the growth rules, the optimisation performs significantly better. In addition to decreasing string creation time, this optimisation can improve rendering performance as straight branch segments are automatically combined into a single, longer branch. Although the optimisation slows down string creation by 5% (in a small case it slowed down by 24%), it accelerates interpretation by up to 40% and can reduce the number of polygons by almost half.

- Branch Optimisation: This optimisation addresses the other performance sink in the L-System rules, that of branching. Tree L-Systems often contain rules with a fixed number of sub-branches per branch. As the tree grows, the number of branches may increase exponentially. In order to address this problem a form of instancing is applied. Rules that branch are tagged with additional metadata signifying where they are in the tree. The branching rules are updated such that, with a certain probability, the branches are either written to or read from a 'branch' cache. Branches that are read from the cache require significantly less processing thereby speeding up the tree creation process. Another benefit of this optimisation is that tree branches may be shared across multiple trees making the algorithm suitable for improving forest creation and rendering performance. Depending on the size of the forest, this algorithm improves creation time anywhere from about 60 seconds to create $2,000$ trees to roughly 4 seconds to create $100,000$.

Each of these String Creation algorithms and L-System Optimisations were tested on six separate L-Systems with different characteristics as well as forests of different sizes.

The algorithms presented were able to achieve a significant improvement in speed and memory usage when creating very large forests. At the same time the system is able to maintain real-time frame rate for forests below $2,500$ trees. As the number of trees in the forest increases, the performance decreases. This is expected, however, due to the sheer volume of geometry being rendered. The acceleration of procedural generation for forests, itself, has many uses. It opens the way for compressing forests used for games down to a few random seeds and a list of tree positions. The game could then generate the forest on the fly as the user enters and leaves the area. The 1 to 2 second waiting time for the geometry cache to fill could be masked as a loading screen or hidden in a more subtle manner.

8.1 Future Work

There are several aspects of the algorithms presented in this thesis that can be improved. The version of L-Systems that we used in the system, although sufficient for most tree L-Systems, can be improved by fully catering for 'Bracketed' L-Systems. Brackets that represent a branch, and their contents, should be ignored for the purposes of context matching. In our system, however, these symbols are matched leading to different results. This can be easily accommodated via our Automaton version by replacing a single automaton with a stack of automatons. When encountering a opening bracket ([) symbol in the string creation, we simply push a new automaton onto the stack, while popping that automaton off at the matching close bracket (]) symbol.

At present, the system cannot handle the full gamut of L-Systems that are used today. One of the more popular forms of L-System today is the Open L-System. Open L-Systems interleave string creation and interpretation, allowing the string creation phase to make choices based on their current interpretation, which can significantly slow down the entire process. The speed of these L-Systems depends on the exact interpretation queries being performed. For instance, just perform collision detection might seem costly, however with appropriate use of data structures, the cost can be reduced. On the other hand, a query might require a full simulation of the light shining through the canopy to determine whether the plant will grow, which will add significant overhead. In the future, we plan to expand the framework so that Open L-Systems are available for use without sacrificing the system's current performance levels.

The L-Systems are not only useful for generating trees, but also buildings and cities. In these circumstances, the number of rules, and their complexity, increase dramatically. Although not trivial, applying the new String Creation algorithms may accelerate the slow process of city creation. The L-System Optimisations presented here were specifically designed for trees so more generalised techniques would need to be developed. Constructing adequate cities with L-Systems necessitates Open L-Systems. A future area for framework is the construction of, not just forests, but other large scale environments which may be procedurally generated. One could also envisage a system that allows one to generate entire planets with cities, forests, lakes, oceans and other terrains with the appropriate tools.

In terms of our other improvements, there are also more L-System Optimisations that could be applied to tree generation. An optimisation that we considered but did not implement due to time limitations is to recognise static parts of the string and skip over

them to avoid needless processing. The main application for this optimisation would be for leaves. Each leaf is represented by approximately twenty terminal symbols. In our L-Systems we have trees that can have thousands of leaves, thus avoiding processing the symbols in each leaf could result in a significant performance increase. This could also be used to improve the running time of context-sensitive L-Systems, which are important for modelling communication (such as bud growth) within tree L-Systems.

The Growth Optimisation relies on heuristics to match a limited set of growth functions which could be expanded to many more functions. In our test suite, there were two L-Systems which did not match any of the growth patterns, test case 4 and test case 6. Although it is not always possible to compute a formula for growth of a particular L-System, there are many detectable cases which this optimisation could be used for. Two examples of such would be polynomial growth and rules which grow as the square-root of N. The latter case is evident in the last L-System test case.

The Branch Optimisation has several shortcomings. It currently assumes that branches from certain generations of an L-System of different L-System hero trees may be interchanged freely. This is not always the case as certain L-Systems use parameters to decrease the chance of tree branching as the depth of the tree increases. This probability might not decrease at the same rate throughout the tree, for instance when the probability decreases by a random amount or factor. Future work for the framework could involve incorporating these rules when matching branches instances to combine.

One final area of potential improvement is the renderer. While the renderer allows complex shaders to be used based on an L-System's requirements, it is still fairly basic. The system does not perform much in the way of acceleration. For instance, every tree is drawn at each frame regardless of whether it is visible. An initial prototype used the OGRE framework[1] for rendering, however, it soon became clear that extra capabilities were needed. For instance, OGRE does not allow sharing of index buffers across multiple objects in the scene, which is crucial to our system's performance, Other rendering engines may not exhibit such limitations which represents an avenue for improving our current system.

The system, as a whole, also has several directions in which it could take. When the project was started it was intended to be used for real-time purposes such as games and simulations. However, the system can be used for various off-line problems such as designing of large persistent environments. The system, although relying on randomness, is completely determinisitic once the seeds are known. One could extend the system to create these large deterministic environments for purposes such as for environment creation in a 3D modelling package. One could even adapt the system to allow creation of static assets for games such as skyboxes and backgrounds, by simply capturing the rendered output to several textures.

[1] http://www.ogre3d.org

Bibliography

[1] H. Abelson and A.A. DiSessa. *Turtle geometry: The computer as a medium for exploring mathematics*. The MIT Press, 1986.

[2] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] A.V. Aho, J.D. Ullman, and S. Biswas. *Principles of compiler design*, volume 21. Addison-Wesley, 1977.

[4] M. Aono and T.L. Kunii. Botanical tree image generation. *Computer Graphics and Applications, IEEE*, 4(5):10–34, 1984.

[5] T. Bäck. Evolutionary algorithms in theory and practice, 1996.

[6] Michele Bahr and David Patterson. Anabaena filaments with heterocysts and cells of variable size. differential interference contrast., January 2013.

[7] P. Becker et al. Working draft, standard for programming language c++. *ISO/IEC, Tech. Rep N*, 2798:904–949, 2010.

[8] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[9] J. Brooks, R.S. Murarka, D. Onuoha, F.H. Rahn, and H.A. Steinberg. An extension of the combinatorial geometry technique for modeling vegetation and terrain features. Technical report, DTIC Document, 1974.

[10] A.W. Burks, D.W. Warren, and J.B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical tables and other aids to computation*, pages 53–57, 1954.

[11] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[12] K. Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.

[13] R.L. Cook and T. DeRose. Wavelet noise. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 803–811. ACM, 2005.

[14] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 421–430, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[15] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech. Plant models faithful to botanical structure and development. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 151–158. ACM, 1988.

[16] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *Proceedings of the conference on Visualization'02*, page 226. IEEE Computer Society, 2002.

[17] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 275–286. ACM, 1998.

[18] Herbert Edelsbrunner and Ernst P Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics (TOG)*, 13(1):43–72, 1994.

[19] P. Eichhorst and W.J. Savitch. Growth functions of stochastic lindenmayer systems*. *Information and Control*, 45(3):217–228, 1980.

[20] C. Everitt. Anisotropic texture filtering in opengl. *NVIDIA White Paper*, 6, 2000.

[21] H. Hamano et al. Preliminary study on the shape of trees. In *Proc. Autumnal Conf. Japanese Gardening Society*, pages 38–39, 1982.

[22] G.S. Hornby and J.B. Pollack. Evolving l-systems to generate virtual creatures. *Computers & Graphics*, 25(6):1041–1048, 2001.

[23] N. Jackie. Opengl programming guide (red book).

[24] C. Jacob. Modeling growth with l-systems & mathematica. *Mathematica in Education and Research*, 4(3):12–19, 1995.

[25] Jovan Janari. personal communication, 2012.

[26] Y. Kawaguchi. A morphological study of the form of nature. In *ACM SIGGRAPH Computer Graphics*, volume 16, pages 223–232. ACM, 1982.

[27] D.E. Knuth. Seminumerical algorithms, volume 2 of the art of computer programming, 1997.

[28] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[29] D. Koks. *Explorations in Mathematical Physics: The Concepts Behind an Elegant Language*. Springer, 2006.

[30] Brendan Lane, Przemyslaw Prusinkiewicz, et al. Generating spatial distributions for multilevel models of plant communities. In *Graphics Interface*, pages 69–80. Citeseer, 2002.

[31] M. Larive and V. Gaildrat. Wall grammar for building generation. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, page 437. ACM, 2006.

[32] A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[33] A. Lindenmayer and G. Rozenberg. Developmental systems and languages. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 214–221. ACM, 1972.

[34] A. Lindenmayer and G. Rozenberg. *Automata, languages, development*. North-Holland, 1976.

[35] Yotam Livnyl, Soeren Pirk, Zhanglin Chengl, Feilong Yanl, Oliver Deussen, Daniel Cohen-Or, and Baoquan Chenl. Texture-lobes for tree modelling. 2011.

[36] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[37] B.B. Mandelbrot. *The fractal geometry of nature*. Wh Freeman, 1983.

[38] B.B. Mandelbrot and M. Aizenman. Fractals: form, chance, and dimension. *Physics Today*, 32:65, 1979.

[39] B.B. Mandelbrot and J.W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM review*, 10(4):422–437, 1968.

[40] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410. ACM, 1996.

[41] Bruce Merry. A bit of fun: fun with bits, 2009. http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation.

[42] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. *Procedural modeling of buildings*, volume 25. ACM, 2006.

[43] P. Müller, G. Zeng, P. Wonka, and L. Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, 26(3):85, 2007.

[44] F.K. Musgrave. Methods for realistic landscape imaging. *Yale University, New Haven, CT*, 1993.

[45] B. Neubert, T. Franken, and O. Deussen. Approximate image-based tree-modeling using particle flows. *ACM Transactions on Graphics (TOG)*, 26(3):88, 2007.

[46] J. Olsen. Realtime procedural terrain generation. *Department of Mathematics And Computer Science (IMADA) University of Southern Denmark*, 2004.

[47] P.E. Oppenheimer. Real time design and animation of fractal plants and trees. In *ACM SiGGRAPH Computer Graphics*, volume 20, pages 55–64. ACM, 1986.

[48] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. In *ACM Transactions on Graphics (TOG)*, volume 28, page 58. ACM, 2009.

[49] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[50] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–12. CSIRO Publishing, 1996.

[51] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Mech. Visual models of plant development. *Handbook of formal languages*, 3:535–597, 1996.

[52] P. Prusinkiewicz, M.S. Hammel, and E. Mjolsness. Animation of plant development. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 351–360. ACM, 1993.

[53] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 351–358. ACM, 1994.

[54] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants (the virtual laboratory). 1991.

[55] P. Prusinkiewicz and University of Regina. Dept. of Computer Science. *Graphical applications of L-systems*. Department of Computer Science, University of Regina, 1985.

[56] P. Prusinkiewicz, M. Shirmohammadi, and F. Samavati. L-systems in geometric modeling. *arXiv preprint arXiv:1008.1664*, 2010.

[57] W.T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *ACM Siggraph Computer Graphics*, volume 19, pages 313–322. ACM, 1985.

[58] R. Robson. *Using the STL: the C++ standard template library*. Springer-Verlag New York Inc, 2000.

[59] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. In *NPH*, pages 63–70, 2007.

[60] A. Salomaa and G. Rozenberg. Handbook of formal languages. *Vol 3: Beyond*, pages 33–36, 1997.

[61] I. Shlyakhter, M. Rozenoer, J. Dorsey, and S. Teller. Reconstructing 3d tree models from instrumented photographs. *Computer Graphics and Applications, IEEE*, 21(3):53–61, 2001.

[62] G. Stiny. Introduction to shape and shape grammars. *Environment and planning B*, 7(3):343–351, 1980.

[63] D.G. Tarboton, R.L. Bras, and I. Rodriguez-Iturbe. The fractal nature of river networks. *Water Resources Research*, 24(8):1317–1322, 1988.

[64] M. Teichmann and S. Teller. Assisted articulation of closed polygonal models. In *Proc. 9th Eurographics Workshop on Animation and Simulation*, volume 387. Citeseer, 1998.

[65] P. Vitányi. Growth of strings in context dependent lindenmayer systems. *L systems*, pages 104–126, 1974.

[66] A. Walker. Adult languages of l systems and the chomsky hierarchy. *L Systems*, pages 201–215, 1974.

[67] J. Weber and J. Penn. Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128. ACM, 1995.

[68] L.Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.

[69] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[70] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 91–100, New York, NY, USA, 1994. ACM.

[71] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. *Instant architecture*, volume 22. ACM, 2003.

[72] R.S. Wright, N. Haemel, G. Sellers, and B. Lipchak. *OpenGL SuperBible: comprehensive tutorial and reference*. Addison-Wesley Professional, 2010.

Appendix A

Licensing

Although most of our system is written as a stand-alone application with little use of external libraries, we do require the SDL and OpenGL libraries for tasks that do not relate to L-Systems.

SDL is a cross-platform library for interacting with low-level OS features such as reading from input devices and window management. We use SDL version 1.2 which is licensed under GNU LGPL version 2. The text for this license is available at *http://www.gnu.org/licenses/gpl-2.0.html*. This is a non-restrictive license that, provided that no source code is copied and it is used as a shared library only. SDL can be found at: *http://www.libsdl.org*.

OpenGL is a graphics API that allows access to accelerated hardware. It is cross-platform as opposed to other popular alternatives such as the Direct3D API. No additional licensing is required to use the API in a project.

The license that we have chosen for our project is the MIT license. This license can be found at *http://opensource.org/licenses/MIT*. This license allows free distribution of our work, but prohibits other programmers from using our code in a project released under any other license. This also prohibits other people from using our work in a commerical project.

Appendix B

L-System Ruleset Grammar

The following is a grammar which describes the expected format of an L-System rule file by our system. Square brackets indicate that its contents are optional. Double-quoted text represents terminal symbols used within the grammar.

$$
\begin{aligned}
\mathit{string} &\Rightarrow \mathit{tokenName}[\text{“}(\text{”}\,\mathit{parameterList}\text{“})\text{”}] \\
\mathit{parameter} &\Rightarrow \mathit{expression} \\
\mathit{parameterList} &\Rightarrow \mathit{parameter}[\text{“,”}\,\mathit{parameterList}] \\
\mathit{leftContext} &\Rightarrow [\mathit{string}\text{“}\,\text{<”}] \\
\mathit{rightContext} &\Rightarrow [\text{“}\,\text{>”}\,\mathit{string}] \\
\mathit{conditional} &\Rightarrow [\text{“}\,\text{:”}\,\mathit{parameterList}] \\
\mathit{strictPredecessor} &\Rightarrow [\mathit{token}] \\
\mathit{LHS} &\Rightarrow \mathit{leftContext}\ \mathit{strictPredecessor}\ \mathit{rightContext}\ \mathit{conditional} \\
\mathit{RHS} &\Rightarrow [\mathit{string}] \\
\mathit{RHSList} &\Rightarrow \text{“}\,\text{→”}\,\mathit{RHS}[\text{“}\,\text{→”}\,\mathit{RHSList}] \\
\mathit{rule} &\Rightarrow \mathit{LHS}\ [\ \mathit{probability}\ ]\ \text{“}\,\text{→”}\,\mathit{string}\ \mathit{RHSList}
\end{aligned}
$$

Appendix C

Test Case L-Systems

The L-Systems that we use in the system are based off ones found in *Algorithmic Beauty of Plants*[54]. While the meaning of many of the symbols present in these L-Systems can found in this book, we have added several modules to the system to simplify some graphical tasks. Specifically, we have added the *SetMaterial* module which easily allows an L-System to switch to a different material, for example from bark to trees. We have also added a texture coordinate module, *TC*, so that vertices created with the f symbol can be textured simply. Like the position of the turtle, the current material and texture coordinates are reverted back to their previous states when the] symbol is encountered.

C.1 Test Case 1

$$
\begin{aligned}
&\mapsto \text{setMaterial(“Bark”) } F\ A \\
A\ &\mapsto [\&F\&L\hat{}L!A]\ /////\text{‘}\ [\&F\&L\hat{}L!A\hat{}L\hat{}L\hat{}L\hat{}L\hat{}L\hat{}L\hat{}L]\ ///////\text{‘}\ [\&FL\&L\hat{}!A] \\
A\ &\mapsto [\&FL\&L\hat{}!A]/////\text{‘}[\&F\&L\hat{}L!A] \\
A\ &\mapsto [FL\&L\hat{}!A] \\
F\ &\mapsto^{0.45} F \\
F\ &\mapsto^{0.55} FF \\
L\ &\mapsto [\text{setMaterial(“Leaves”)}\hat{}\hat{}\{\&\text{TC}(0.62,0.33)f\hat{}\text{TC}(0.34,\ 0.28)f\hat{}\text{TC}(0.07,\ 0.42)f\&| \\
&\quad \&\text{TC}(0.30,\ 0.70)f\hat{}\text{TC}(0.61,\ 0.72)f\hat{}\text{TC}(0.81,\ 0.53)f\}]
\end{aligned}
$$

C.2 Test Case 2

$$
\begin{aligned}
&\mapsto \text{setMaterial(“Bark3”) } F \\
F \;&\mapsto \backslash FL[L - \&\&\ FL][L\ + \&FL]||F[L\&/FL][L + LFL] \\
L \;&\mapsto [\text{setMaterial(“Leaves”) } \hat{}\hat{}\{\&\text{TC}(0.62,0.33)f\,\hat{}\,\text{TC}(0.34,\ 0.28)f\,\hat{}\,\text{TC}(0.07,\ 0.42)f\&| \\
&\quad \&\text{TC}(0.30,\ 0.70)f\,\hat{}\,\text{TC}(0.61,\ 0.72)f\,\hat{}\,\text{TC}(0.81,\ 0.53)f\}]
\end{aligned}
$$

C.3 Test Case 3

$$
\begin{aligned}
&\mapsto \text{setMaterial(“Bark2”) } F(0) \\
F(t) \;&\mapsto F(10)/(137.5) + \text{TrunkOffshoot} - F(0) \textbf{ if } t\ =\ 0 \\
F \;&\mapsto^{0.7} F \\
F \;&\mapsto^{0.3} FF \\
A \;&\mapsto [\&F\&L\,\hat{}\,L!A]\ /////\text{`}\ [\&F\&L\,\hat{}\,L!A\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L]\ ///////\text{`}\ [\&FL\&L\hat{}!A] \\
A \;&\mapsto [\&FL\&L\hat{}!A]/////\text{`}[\&F\&L\,\hat{}\,L!A] \\
A \;&\mapsto [FL\&L\hat{}!A]
\end{aligned}
$$

C.4 Test Case 4

$$
\begin{aligned}
& \mapsto \text{setMaterial(“Bark”) } F\ A \\
A \quad & \mapsto [\&FL!A]/////\text{`}[\&FL!A]///////\text{`}[\&FL!A] \\
A \quad & \mapsto [\&FL!A]/////\text{`}[\&FL!A] \\
A \quad & \mapsto [FL!A] \\
A \quad & \mapsto FA \\
FFFFFFFFFF < \quad F \quad & \mapsto L///L///L+++L+++L+++L\&L\&L\&L\backslash\backslash\backslash\backslash\backslash\backslash\backslash \\
F \quad & \mapsto F \\
F \quad & \mapsto SF \\
F \quad & \mapsto S/\&//-//\hat{}//+F \\
S \quad & \mapsto FL/\text{dummy}/\text{dummy}\backslash\backslash \\
L \quad & \mapsto [\text{setMaterial(“Leaves”)}\hat{}\hat{}\{\&\text{TC}(0.62,0.33)f\hat{}\,\text{TC}(0.34,\ 0.28)f\hat{}\,\text{TC}(0.07,\ 0.42)f\&| \\
& \quad \&\text{TC}(0.30,\ 0.70)f\hat{}\,\text{TC}(0.61,\ 0.72)f\hat{}\,\text{TC}(0.81,\ 0.53)f\}]
\end{aligned}
$$

C.5 Test Case 5

$$
\begin{aligned}
& \mapsto \text{setMaterial(“Bark”) } X \\
X \mapsto & F[+X][-X]FX \\
F \mapsto & FF
\end{aligned}
$$

C.6 Test Case 6

$$
\begin{array}{rl}
 & \mapsto \text{setMaterial(“Bark3”) } XF_uF_aXG \\
F_u <F_a> F_a & \mapsto F_u \\
F_u <F_a> X & \mapsto F_dF_a \\
F_a <F_a> F_d & \mapsto F_d \\
X <F_a> F_d & \mapsto F_u \\
F_u & \mapsto F_a \\
F_d & \mapsto F_a \\
G & \mapsto AG \\
A & \mapsto [\&F\&L\,\hat{}\,L!A]/////\text{`}[\&F\&L\,\hat{}\,L!A\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L\,\hat{}\,L]////\text{`}[\&FL\&L\hat{}!A] \\
A & \mapsto [\&FL\&L\hat{}!A]/////\text{`}[\&F\&L\,\hat{}\,L!A] \\
A & \mapsto [FL\&L\hat{}!A] \\
F & \mapsto XF_uF_aX \\
L & \mapsto [\text{setMaterial(“Leaves”)}\,\hat{}\,\hat{}\{\&\text{TC}(0.62,0.33)f\,\hat{}\,\text{TC}(0.34,\,0.28)f\,\hat{}\,\text{TC}(0.07,\,0.42)f\&| \\
 & \quad \&\text{TC}(0.30,\,0.70)f\,\hat{}\,\text{TC}(0.61,\,0.72)f\,\hat{}\,\text{TC}(0.81,\,0.53)f\}]
\end{array}
$$

Appendix D

Effects of L-System Optimisation on String Creation

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	29(100%)	29(100%)	10(100%)
Fern(small)	144(100%)	144(100%)	5(100%)
Stalk Tree(small)	13(100%)	13(100%)	10(100%)
Extended ABOP Tree(small)	21(100%)	21(100%)	5(100%)
Bifurcation(small)	0(100%)	0(100%)	1(100%)
Sqrt-Growth(small)	473(100%)	473(100%)	17(100%)
ABOP Tree(large)	526(100%)	527(100%)	15(100%)
Fern(large)	5215(100%)	5215(100%)	7(100%)
Stalk Tree(large)	26824(100%)	26824(100%)	24(100%)
Extended ABOP Tree(large)	1138(100%)	1138(100%)	11(100%)
Bifurcation(large)	22(100%)	22(100%)	2(100%)
Sqrt-Growth(large)	7569(100%)	7569(100%)	22(100%)

Table D.1: The change in memory required as a result of using only the Identity optimisation. The number outside the brackets indicates the absolute memory required in kilobytes. The number inside the parentheses indicates the change relative to having no optimisations performed.

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	33(113%)	33(113%)	10(100%)
Fern(small)	144(100%)	144(100%)	5(100%)
Stalk Tree(small)	15(110%)	15(110%)	10(100%)
Extended ABOP Tree(small)	21(100%)	21(100%)	5(100%)
Bifurcation(small)	1(170%)	1(169%)	1(100%)
Sqrt-Growth(small)	473(100%)	473(100%)	17(100%)
ABOP Tree(large)	582(110%)	582(110%)	15(100%)
Fern(large)	5215(100%)	5215(100%)	7(100%)
Stalk Tree(large)	29332(109%)	29332(109%)	24(100%)
Extended ABOP Tree(large)	1138(100%)	1138(100%)	11(100%)
Bifurcation(large)	37(171%)	37(171%)	2(100%)
Sqrt-Growth(large)	7569(100%)	7569(100%)	22(100%)

Table D.2: The change in memory required as a result of using only the Growth optimisation. The number outside the brackets indicates the absolute memory required in kilobytes. The number inside the parentheses indicates the change relative to having no optimisations performed.

	Brute Force	Automaton	Automaton Chain
ABOP Tree(small)	36(121%)	36(121%)	8(86%)
Fern(small)	190(132%)	190(132%)	4(92%)
Stalk Tree(small)	16(121%)	16(121%)	8(86%)
Extended ABOP Tree(small)	27(124%)	27(124%)	4(77%)
Bifurcation(small)	0(100%)	0(100%)	1(100%)
Sqrt-Growth(small)	575(121%)	575(121%)	14(86%)
ABOP Tree(large)	642(121%)	642(121%)	13(86%)
Fern(large)	6888(132%)	6888(132%)	6(92%)
Stalk Tree(large)	32720(121%)	32720(121%)	21(86%)
Extended ABOP Tree(large)	1394(122%)	1394(122%)	8(77%)
Bifurcation(large)	22(100%)	22(100%)	2(100%)
Sqrt-Growth(large)	9210(121%)	9210(121%)	19(86%)

Table D.3: The change in memory required as a result of using only the Orientation optimisation. The number outside the brackets indicates the absolute memory required in kilobytes. The number inside the parentheses indicates the change relative to having no optimisations performed.