Charlie Hewitt

Procedural generation of tree models for use in computer graphics

Computer Science Tripos – Part II

Trinity Hall

19th May 2017

"Mathematical analysis and computer modelling are revealing to us that the shapes and processes we encounter in nature – the way that plants grow, the way that mountains erode or rivers flow, the way that snowflakes or islands achieve their shapes, the way that light plays on a surface, the way the milk folds and spins into your coffee as you stir it, the way that laughter sweeps through a crowd of people – all these things in their seemingly magical complexity can be described by the interaction of mathematical processes that are, if anything, even more magical in their simplicity."

— Douglas Adams, *Dirk Gently's Holistic Detective Agency*

Proforma

Name:	Charlie Hewitt
College:	Trinity Hall
Project Title:	Procedural generation of tree models for use in computer graphics
Examination:	Computer Science Tripos - Part II, July 2016
Word Count:	11892
Project Originator:	Charlie Hewitt
Supervisor:	György Dénes

Original Aims of the Project

The aim of the project is to build on the principles set out by Lindenmayer in order to procedurally generate realistic 3D models of trees, incorporating modern graphics techniques to enhance the output of conventional Lindenmayer systems. In addition to this more conventional approach, I hope to develop a similar tool using Weber and Penn's parametric methodology. This will enable me to compare and contrast these two approaches and assess the merit of each.

Work Completed

I have completed both generation tools, as well as analysis tools and example L-system grammars and parameter lists which have enabled me to perform qualitative and quantitative comparisons of the two systems. In addition, I have completed some extension material aimed at simplifying the tree design process for the parametric system.

Special Difficulties

None

Declaration

I, Charlie Hewitt of Trinity Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date: 19th May 2017

Contents

List of Figures

Acknowledgements

The refinement of the focus of this project was aided significantly be the contributions of Erroll Wood and Alex Benton. Graham Hewitt, Andrew Halliday and Simon Moore were of great help in the editing of the dissertation itself and Rafał Mantiuk provided useful insight as to the evaluation of the results. Thanks is also due, of course, to György Dénes for his continued help and support throughout the project.

Chapter 1

Introduction

1.1 Motivation

In this project I intend to develop a system for the procedural generation of highly detailed 3D models of trees and similar plants for use in computer generated imagery (CGI).

In recent years CGI has become increasingly prevalent in the entertainment industry; improvements in technology have enabled increasingly large scale and complex scenes to be constructed. Consequently there has been a greater desire to provide fast, effective ways of producing assets which can be used in these scenes. With the significant presence of plants in the real world, it is not surprising that there is a high demand for botanical assets. Therefore, finding a simple a way to generate large numbers of realistic plant models would be of great benefit to artists working on film and game environments.

My aim is to implement a system which can produce models of a wide variety of tree species in a pseudo-random manner. I have chosen to create two systems employing alternative techniques with the intention of comparing these approaches and the results achieved. I will be implementing systems based on Aristid Lindenmayer's fractal based method [PL90], and Weber and Penn's parametric approach [WP95], both described in detail below. I also hope to improve the design process of a tree model for the artist by utilising techniques such as dimensionality reduction and genetic algorithms to simplify/automate some aspects of the task.

1.2 Related Work

There is a reasonable volume of existing work in this field. The most successful commercial offering being SpeedTree [Inc], a software package designed specifically for generating plant models for use in games and film. There has also been activity in the research community and a number of completely different approaches to tackle the problem have been suggested.

1.2.1 Lindenmayer-Systems

One technique is the use of Lindenmayer-systems (L-systems), proposed by Przemyslaw Prusinkiewicz and Aristid Lindenmayer in their iconic book of 1990, *The Algorithmic Beauty of Plants* [PL90] and first described in the latter's paper of 1968 [Lin68]. L-systems build on the well recognised self-similarity of natural structures, as popularised by Mandelbrot [Man82], using a fractal style approach to the modelling process. Smith employed this technique with impressive results at Lucasfilm [Smi84], though it has fallen out of favour in recent years.

I have chosen to implement a tool based on L-systems, with the aim of incorporating modern techniques not available at the time they were most popular. The principle aspects of L-systems are described in more detail in §2.3.1.

1.2.2 Parametric Approach

Honda was the first to propose representation of trees through the use of a number of simple parameters [Hon71]. This approach was adopted by Aono and Kunii for the purpose of generating 3D models [AK84]. Oppenheimer also began to incorporate this method [Opp86], though both still made use of fractal based structures resulting in somewhat unrealistic models.

Weber and Penn later introduced a much more comprehensive parametric description for tree models with impressive results [WP95]. Their system recursively constructs a 3D model based on a list of numeric input parameters. Existing implementations such as *Arbaro* [Die15], a standalone Java app which outputs to standard 3D file formats, and *Sapling Tree Add-on* [Hal], a plugin for *Blender*, have been quite successful.

I have chosen to implement a system based on Weber and Penn's approach given its relative ease of use and impressive visual results. An overview of Weber and Penn's model is given in §2.3.2 and all parameters are outlined in Appendix C.

1.2.3 Modular Approach

Another more recent approach is to construct a tree model from a number of smaller blocks, often manually modelled or 3D scanned from real trees. The blocks are arranged and merged in a manner such that each join is contiguous, and the resulting complete tree model is therefore visually pleasing. This allows for very highly detailed and visually interesting model with many complex elements, but limits the scope of generation to those tree sections that have been obtained manually.

Impressive results have been achieved using this method [Xie+15], and *Modular Tree Add-on* for *Blender* [MD] uses many aspects of this approach, but I have decided not to focus on this method as it requires a significant library of modelled/scanned tree parts which I am not able to easily obtain.

1.2.4 Space Colonisation

A completely different system was proposed more recently by Runions et al., this involved the modelling of trees using a space colonisation algorithm [RLP07]. The algorithm uses a botanically informed growth process to colonise an enclosing envelope, and was greatly improved by Pałubicki et al. [Pał+09].

There have since been a number of successful implementations, such as *TreeSketch* [Lon+12] and *SpaceTree Add-on* for *Blender* [And]. The results produced using this method are certainly comparable in realism to those of Weber and Penn's approach, but due to time limitation I have decided not to tackle space colonisation in this project.

1.2.5 Botanical Simulation

A somewhat less prominent method is to attempt to accurately model the growth process of the tree in order to create a 3D model. De Reffye et al. achieved relative success with this method [Ref+88], though there has been little work in this direction recently. This is likely due to the computational complexity of modelling the growth process, when comparable visual results can be achieved with much simpler models such as those proposed above. Consequently I have decided not to explore this approach.

Chapter 2

Preparation

2.1 Requirements Analysis

2.1.1 Use Cases

There is currently a great need in the visual effects industry to create virtual representations of the real world, including natural objects such as trees, for use in films, games and static renders. At present these tree models must be created manually by an artist, or generated using commercial systems very similar to those which I intend to develop. Bespoke modelling by an artist is slow and expensive due to the highly skilled nature of the work. My proposed system will allow generation of highly detailed tree models in a much shorter time, without requiring any skill beyond basic use of the *Blender* application [Foua].

For a tree that features prominently in a scene, an artist may wish to augment the generated model with bespoke elements or more detail, but the vast majority of the modelling work has been automated, reducing the overall cost drastically. In many scenarios the model will be suitable as generated; one particular scenario where there is a clear benefit is in the generation of large groups of trees in the form of woodlands or forests. Previously every tree would need to be modelled separately, or identical trees repeated, with my method a huge number of similar but unique trees can be generated very easily giving a much improved visual result.

2.1.2 Goals

More in depth research of the field has enabled me to refine and better specify my success criteria, breaking them down into subgoals that can be more measurably attained. These are as follows:

- Develop working tree generation system based on the L-systems approach.
  - Implement stochastic, parametric L-systems representation in Python.

  - Devise parsing system to convert L-system output into basic geometry within *Blender*.
  - Enhance visual appearance of resultant model through refinement of parsing system.

- Develop working tree generation system based on a purely parametric approach.
  - Implement the core elements of the system outlined in Weber and Penn's paper using Python.
  - Modify any aspects of the model which require it to facilitate incorporation into a *Blender* plugin.
  - Enhance any features of the model which I find to be lacking.

- Generate a series of test models which can be used for performance comparisons of the two systems.
  - A limited set of parameter lists are provided in Weber and Penn's paper, and others with existing implementations, these can be transcribed for use with my system. A number of further parameter lists will have to be designed from scratch.
  - Very basic L-systems are described in *The Algorithmic Beauty of Plants* which may provide a basis for my own. New bespoke L-systems will have to be devised to represent a wide enough range of tree types.

The goals described above are now taken to include provision for leaf modelling which I originally viewed to be an extension goal. I have also refined my other extensions:

- Investigate the use of dimensionality reduction for parameter lists with the aim of enabling an artist to design trees for the parametric system more easily.
- Investigate automation of parameter list design using a genetic algorithm given an image input. The scope of this would likely be quite restricted given the time scale, though will hopefully serve as a proof of concept for automated parameterisation of trees.

Ultimately I would like to assess which of the two systems I develop, L-system based or parametric, provides the best overall performance. This will be judged primarily on visual result, though generation time and complexity of use are also considerations.

2.1.3 Plan

The project is broken into three main implementation stages; for each I intend to employ a spiral model of software development. First working on basic prototypes of the systems, covering the most complex aspects of the implementation. Then iteratively improving these until the system fully satisfies my implementation goals.

L-Systems Implementation

- Develop versatile Python implementation of stochastic, parametric L-systems.
- Implement 2D parsing system for basic L-system definitions.
- Extend parsing system to 3D, first using just line segments then incorporating Bézier curves.
- Augment parsing system with required features to encapsulate fully detailed tree models.
- Devise series of L-system definitions to use as input to the parsing system which describe a number of different tree types.

Parametric Implementation

- Translate general formulae presented in paper to Python.
- Basic implementation of parametric system using line segments to represent branches.
- Extend system to utilise Bézier curves to model branches, including bevelling of branches.
- Further refine system, implementing more detailed aspects of the paper, and introduce any modifications as necessary.
- Devise a number of parameter lists, to use as input to the system, which describe a number of different tree types.

Extensions/Evaluation

- Carry out any final refinements to both systems.
- Develop any profiling tools required to evaluate the two systems.
- Investigate extension goals, with potential for implementing genetic algorithm to devise parameter lists from image input.

2.2 Starting Point

To implement the system I chose to utilise *Blender*'s built in Python scripting functionality. *Blender* provides a full Python 3 interpreter within the application which can be used to manipulate the modelling environment through calls to a number of libraries which are also provided by *Blender*.

Specifically, I made use of *Blender*'s Bézier curve modelling tools which require all control points of the curve to be provided, as well as the bevel radii of these points, and produce a 3D mesh of controllable resolution along the resulting curve—see Figure 2.1. I also utilised the `mathutils` module for its implementation of basic data structures such as

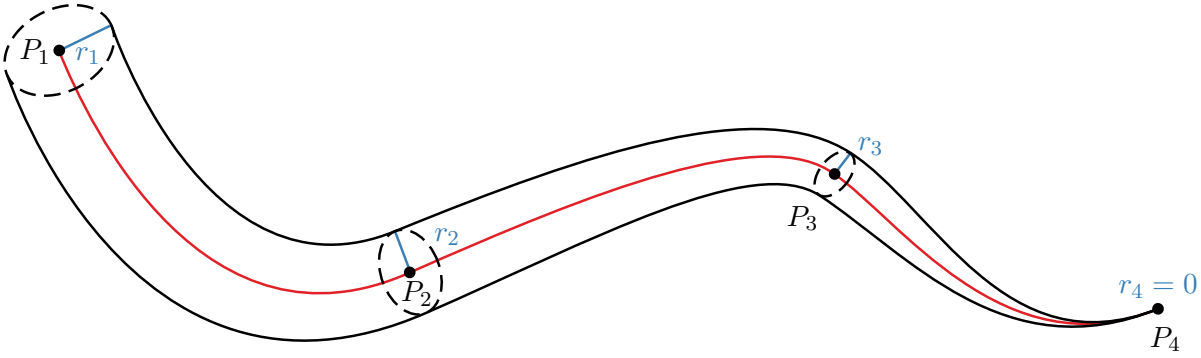


Figure 2.1: Bézier spline (red) and associated bevelled solid (black). Cross sections of radius r_1, r_2, r_3 and r_4 (blue) shown at Bézier points P_1, P_2, P_3 and P_4 respectively—intermediate control points are omitted.

vectors and matrices, and their operators, in order to integrate seamlessly with the other *Blender* libraries required. Standard Python modules such as `random` and `math` were also used extensively.

2.3 Preparative Work

I had not used Python prior to the project so some initial work was required to familiarise myself with the syntax and general programming conventions of the language. This did not require a great deal of effort given Python's relative syntactic simplicity, and the familiarity of many general concepts covered by the Computer Science Tripos which are present in the language.

Given that the primary objective of the project is to create realistic models of trees, I took care to familiarise myself with some of the botany of trees. For this purpose *Cassell's Trees of Britain and Northern Europe* [MW03] proved an invaluable resource.

Effort was also required to understand the principles laid out in both Lindenmayer's *Algorithmic Beauty of Plants*, and Weber and Penn's 1995 paper. The necessary theoretical knowledge for each system is outlined below.

2.3.1 L-Systems

The formalism of L-systems was originally introduced to describe the structure of plants in their most basic form, and resembles context free grammars. An axiom string, s, is defined along with a set of production rules, p_i, mapping an input character to a string of output characters. The system is then iterated a number of times, starting with the axiom. The characters making up the active string at each stage are replaced with the output characters of the applicable production rule, if there is none then the symbol remains unchanged.

The following is a basic example of an L-system definition:

$$\begin{aligned} s &: a \\ p_1 &: a \mapsto bab \\ p_2 &: b \mapsto c \end{aligned}$$

Iterating the system we obtain the following output:

$$\begin{aligned} &a \\ &bab \\ &cbabc \\ &ccbabcc \\ &... \end{aligned}$$

Clearly this alone is not sufficient to model complex 3D plant structures. Lindenmayer therefore extended the basic definition via the introduction of stochasticity; a production rule can now be defined as a series of constituent mappings each with an associated probability:

$$\begin{aligned} s &: a \\ p_1 &: a \overset{0.25}{\longmapsto} ab \\ &\quad\ \overset{0.25}{\longmapsto} ba \\ &\quad\ \overset{0.50}{\longmapsto} bab \\ p_2 &: b \longmapsto c \end{aligned}$$

This allows for far more natural structures as a degree of random variation can be introduced across the system as a whole. We assume any production with a single constituent mapping introduces no stochasticity, i.e., the probability for that mapping is 1. The probabilities for the constituent mappings of any single production rule must of course sum to 1.

Another way in which simple L-systems can be extended is through the introduction of parameters. The parameters of a symbol can then be altered within a production rule, or used as a condition on the production itself. This allows us to propagate useful information through the system, which is critical when producing more complex structures:

$$\begin{aligned} s &: A(1) \\ p_1 &: A(x) : * \quad\ \mapsto A(x*2)B(0) \\ p_2 &: B(x) : x \leq 3 \mapsto B(x+1) \\ p_3 &: B(x) : x > 4 \mapsto B(x-1) \end{aligned}$$

Where $*$ is used to indicate the lack of a condition, i.e., the production always applies.

These two augmentations can be combined to create a stochastic, parametric L-system. Another possible extension of the formalism is context sensitivity, though in practice stochastic parametricity is generally adequate to describe sufficiently complex structures.

The output of the iterated L-system at this point, however, is just a string—possibly with associated parameter values. This output string must therefore be parsed in order to convert the symbols into some geometry which will make up the final tree model.

2.3.2 Weber and Penn

The approach of Weber and Penn considers only numerical parameters as input. The tree is considered as a number of distinct levels of recursion; the first being the trunk, the second the primary level of monopodial branches[1], the third the child branches of these and so on. They include a number of parameters to describe the tree model as a whole such as the global scale, the number of levels of branching and the flare at the base of the trunk. In addition to these are a series of level specific parameters, defined independently for each recursive depth—suffixed **[n]** to indicate the parameter value at the nth level of recursive branching. Many of these parameters are specified relative to the parent branch, such as the length, while others are independent, such as the curve of the branch and the angle it makes from its parent.

Dichotomous branching[2] is also incorporated in the model, with stems able to be 'cloned' at a random point along their length. The two stems then continue to grow in identical condition but using different random seeds. The overall shape of the tree can also be controlled via modification of the first level branch lengths. This is achieved using an overall shape parameter which takes discrete values based on predefined common tree shapes[3]. In addition the shape can be modified by defining an enclosing envelope and then iteratively shortening branches until they fit within this envelope, similar to the act of pruning.

Leaves are included in the model, with their positioning and orientation determined in the same manner as for stems. There is, however, a further modification to the leaves' orientations—they are reoriented, depending on the fractional value of a parameter, to face outwards and upwards from the centre of the tree—to emulate the effect that light

[1]Subordinate branches when central parent branch continues to grow unaltered, see Figure 2.2.

[2]Branching as a result of equal division of the terminal bud, see Figure 2.2.

[3]Conical, spherical, hemispherical, cylindrical, tapered cylindrical, flame, inverse conical and tend flame—see Figure 2.4.

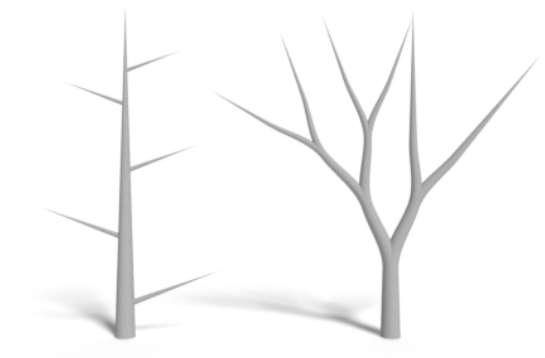


Figure 2.2: Monopodial (left) and dichotomous (right) branching.

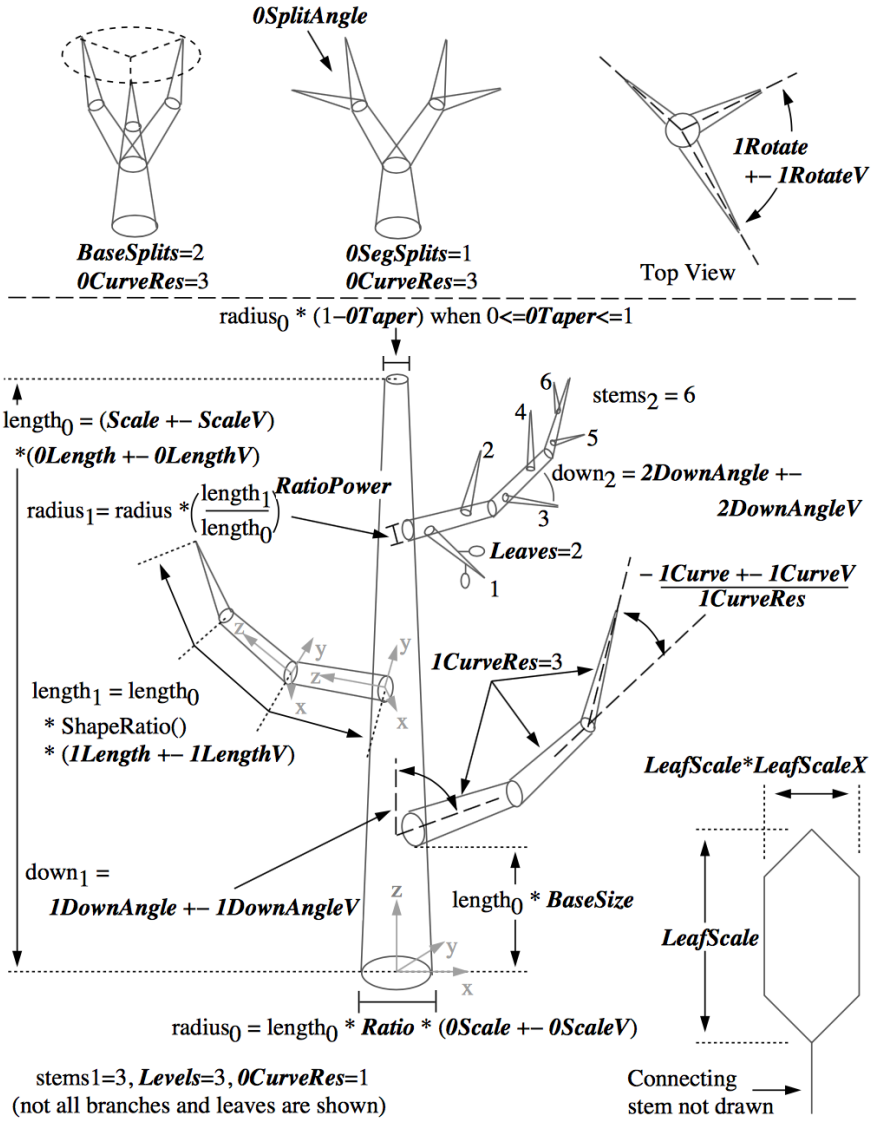


Figure 2.3: Tree diagram from *Creation and Rendering of Realistic Trees*, Jason Weber and Joseph Penn. ©1995 ACM, Inc.

has on the growth of the tree.

There are many more parameters than described above which influence the resulting 3D model, Figure 2.3 taken from the original paper illustrates some of these, for a complete description of all original parameters consult [WP95]. Brief descriptions of all the parameters in my own implementation of the system, broadly similar to the originals, are given in Appendix C.

The geometry of the model is built up recursively; each stem created as a series of segments with all levels of branching for each segment are generated before moving on to the next. This is necessary because many aspects of each stem are calculated based on the attributes of the parent, for example the length of child stems is proportional to the length of the parent.

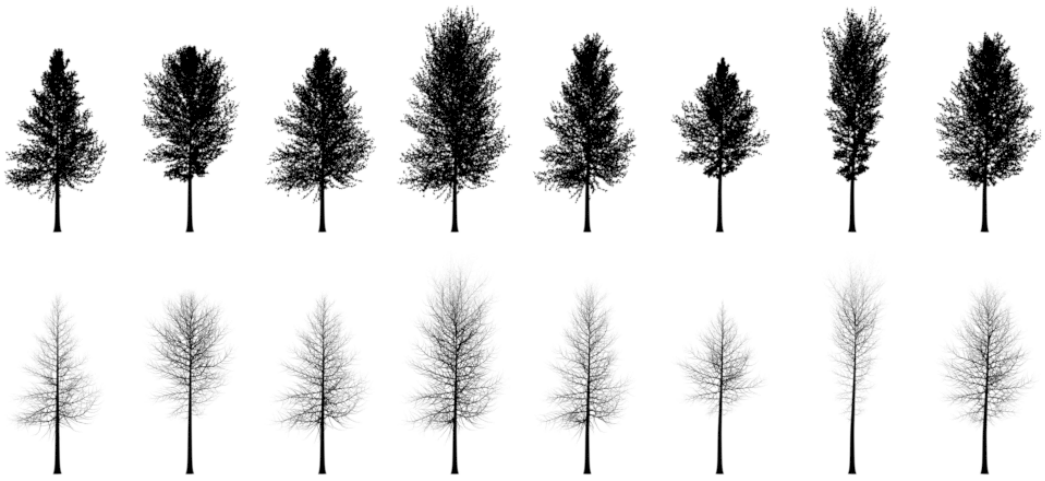


Figure 2.4: Quaking aspen generated using my parametric tool for each common tree shape, with and without leaves.

2.4 Summary

My two tree generation tools will be developed using a spiral model of development. They will be written in Python for use by artists within the *Blender* application as a plugin.

L-systems provide a simple formalism, similar to context free grammars, to help describe the structure of a tree. The implementation of these systems is described below in §3.1, in addition to a full explanation of the parsing process to generate 3D models from the resulting symbol string.

The parametric model of Weber and Penn is a more bespoke solution to the problem, further details on various aspects of the model and their implementation as well as numerous additions and alterations are described in §3.2.

Chapter 3

Implementation

As described in §2.1.3, there are three key phases in the implementation of my project. Firstly, the L-system based tool including both an abstract implementation of L-systems and a parsing system to generate 3D models from the resulting symbolic representation. Secondly, the parametric tool based on the model proposed by Weber and Penn [WP95] with a number of modifications to improve realism. Finally, the exploration of extension goals including dimensionality reduction and use of a genetic algorithm to simplify parameter list design.

3.1 L-Systems Approach

My L-systems tool consists of two primary phases; generation and parsing. In the first phase an L-system grammar is iterated a number of times to produce an output string consisting of L-system symbols and associated parameters. This string is then parsed, with all symbols iterated over and actions taken based on their interpretation to construct the output model.

In order to ensure the system is robust, my L-system implementation is as general purpose as possible and the generation and parsing phases are completely independent. So in theory an alternative parser could be developed which produces models in an entirely different format to those which my parser currently does.

3.1.1 L-System Implementation

My implementation of L-systems in Python includes an `LSystem` class which contains the current system state as a list of `LSymbol`s. Each `LSymbol` is defined by a key character and an optional parameter dictionary. The `LSystem` also stores a dictionary of production rules, which map `LSymbol` key characters to a function which represents the production rule associated with that symbol, as well as a number of universal parameters for the tree such as leaf shape and tropism.

The `LSystem` is initialised with a production dictionary and an axiom, which is a list of `LSymbol`s. Every time the `LSystem` is iterated, the list of `LSymbol`s currently stored (initially the axiom) is walked over and the the production rule which corresponds to the key of the current `LSymbol` is looked up and executed, replacing the current symbol with

the result of executing that production rule. This may be a single `LSymbol` or a list of many `LSymbols`, if no production rule exists then the `LSymbol` is left unchanged. As only a single symbol is considered at a time, this implementation supports only context free L-Systems, this greatly simplifies the implementation while retaining more than adequate expressivity.

Production rules are defined as Python functions. This means that arbitrary code can be executed in order to determine the output set of `LSymbols` the rule generates. Stochasticity can therefore be introduced to the system through use of the `random` library within these production rules. Productions rules are given the current symbol as input, so the parameter dictionary of the symbol is also accessible. Parameter values can therefore be modified and propagated through the system as well as used as conditionals within the production rule.

This means that all aspects of the stochastic, parametric L-systems described in §2.3.1 can be easily translated. For example, the L-system with formal definition

$$\begin{aligned} s &: A(1) \\ p_1 &: A(x) : * \quad \longmapsto A(x*2)B(0) \\ p_2 &: B(y) : y \leq 3 \xmapsto{0.3} B(y-1) \\ &\qquad\qquad\quad \xmapsto{0.7} B(y+3) \\ p_3 &: B(y) : y > 4 \longmapsto B(y)A(y) \end{aligned}$$

becomes in Python:

```
def a_production(sym):
    return [LSymbol("A", {"x": sym.parameters["x"] * 2}), LSymbol("B", {"y": 0})]

def b_production(sym):
    res = [sym]
    if sym.parameters["y"] <= 3:
        if random() <= 0.3:
            res = [LSymbol("B", {"y": sym.parameters["y"] - 1})]
        else:
            res = [LSymbol("B", {"y": sym.parameters["y"] + 3})]
    elif sym.parameters["y"] > 4:
        res = [LSymbol("B", {"y": sym.parameters["y"]}),
               LSymbol("A", {"x": sym.parameters["y"]})]
    return res

LSystem(axiom=[LSymbol("A", {"x": 1})], rules={"A": a_production, "B": b_production})
```

As shown here, multiple productions for a single symbol are collapsed into a single Python function definition which encapsulates all rules, conditioned as specified in the formal definition.

3.1.2 Parsing

The 3D model is constructed from the symbols and associated parameters generated by the first phase of the process using a parser. Predefined geometric interpretations for various symbols are described using a turtle; these descriptions enable the final output model to be constructed by the parser as a series of Bézier curves, with a polygonal mesh used for leaves.

Turtle Graphics

Turtle graphics in its simplest form describes a 2D turtle which stores its position and direction as well as some state to describe its 'pen'. The turtle can then be controlled using these attributes and instructed to walk forward, drawing a line. I have used an extension of this principle similar to that described in Chapter 2 of *The Algorithmic Beauty of Plants* to enable use of a turtle in 3D.

The turtle's orientation is represented using two vectors; the `forward` vector determines the direction in which the turtle faces, and the `right` vector specifies its roll about the `forward` vector. These, along with the `position` vector, fully specify the turtle's geometric state. This can then be modified using pitch, turn and roll operations and the standard walk forward command. Instead of a 'pen', only the current radius of the branch being drawn is stored as an attribute of the turtle, which can also be set dynamically.

Symbol Interpretation

The following list defines all `LSymbol`s which have predefined parsing interpretations and their associated actions, all bold strings represent parameter names of the `LSymbol` in question.

`!`	Set turtle width to **w**.
`F`	Move turtle forward by **l** and draw branch of this length, if **leaves** is non-zero then distribute that number of leaves along the branch according to **leaf_r_ang** and **leaf_d_ang**.
`A` or `%`	Close end of branch, i.e., taper to 0 radius.
`+`	Turn turtle left by **a**.
`-`	Turn turtle right by **a**.
`&`	Pitch turtle down by **a**.
∧	Pitch turtle up by **a**.
`/`	Roll turtle right by **a**.
\	Roll turtle left by **a**.
`L`	Create leaf according to **d_ang** and **r_ang**.
`[`	Start branch.
`]`	End branch.
`$`	Reset turtle to vertical.

Most of these definitions are very similar to those given in *The Algorithmic Beauty of Plants*, though some slight alterations have been made.

Model Construction

Interpreting the output of the L-system using a turtle and the symbol interpretations outlined above provides a geometric representation from the input L-system grammar. A 3D model of the tree is then constructed from the basic position, orientation and thickness information the turtle encapsulates. The action associated with the `F` symbol is key to this, as it describes drawing a branch.

The turtle's motion as described above only allows for straight branch sections as the turtle can only move forward in the direction it is currently facing. To achieve the desired curved branches from the turtle geometry obtained during the parsing procedure the current branch's Bézier spline is extended by one point for each `F` encountered. The position of the new point is given by the turtle's position after moving forward by the amount specified by the **l** parameter of the `F` symbol. The first point in each spline is generated either when the trunk is first constructed, or on encountering a `[`, so there is always a Bézier point prior to the one added when encountering a `F`.

When each new Bézier point is added to the spline the control points of the previous point, the start point of the segment being constructed, are also updated. The difference vector for the two control points nearest the start point (i.e., the tangent to the curve at that point) is aligned with the direction of the segment being built and scaled to be proportional to the length of the segment. The resulting branch therefore forms a smooth curve. Some random variation in the segment direction can be introduced to create a more natural winding appearance for the branch. The radius of the branch at the start point is also updated to a linear interpolation of the radius at the point prior to the start point and the turtle's current radius value to ensure a smoothly tapering branch.

Branching

The `[` and `]` symbols also present an interesting challenge. It is necessary to keep track of the series of turtle states in order to backtrack on completion of the new branch and we also want to minimise the number of Bézier splines required to construct the tree (rather than simply starting a new spline after backtracking). A stack is used to achieve this.

Whenever a `[` is encountered a spline is created for the new branch, which is then constructed as normal. When the `[` is encountered a copy of the current turtle is also placed onto the stack, along with the previously active spline. Upon encountering the matching `]`, the current turtle and spline are discarded and the previous turtle and spline restored by popping from the stack. Construction of the parent branch can then continue on its original spline, restarting from the position at which the child was produced.

An edge case arises if a branch, or series of nested branches, is encountered which contain no actual branch segments (`F` symbols). In this case the branch has no visible geometric interpretation so can be ignored.

A flag is therefore used to track the validity of each branch, initialised to `FALSE`. Upon encountering an `F` during traversal of the branch, the flag is set to `TRUE`. If the end of the branch, `]`, is encountered while the branch is still indicated as being invalid then no `F` is present within the branch so its spline can be safely removed. This flag must also be placed on the stack each time a `[` is encountered and restored with the matching `]`.

If the stack is ever empty then an unmatched end branch symbol, `]`, must have been encountered, as the trunk (the spline created upon commencing parsing of the system) should always remain on the stack. An exception is therefore raised as the input string to the parser, and underlying L-system definition, must be invalid. Similarly, if after parsing all symbols the stack contains more than just the trunk then an unmatched start branch symbol, `[`, must have been encountered so again an exception is raised.

Tropism

In order to enhance the realism of the model, I also included the concept of a tropism vector, similar to that described by Lindenmayer in Chapter 2 of *The Algorithmic Beauty of Plants* [PL90]. At the start of each branch segment, the current direction of the branch, $\vec{D}$, is rotated towards the 3D tropism vector, $\vec{T}$, specified as a parameter of the `LSystem`. This is achieved by rotating $\vec{D}$ about the axis $\vec{D} \times \vec{T}$ by an angle, θ, proportional to $|\vec{D} \times \vec{T}|$, as demonstrated in Figure 3.1

This causes the tree to appear to grow as if acted on by a force in the direction of the tropism vector, with the severity of this effect proportional to the length of the tropism vector. We can therefore model trees which are affected visibly by gravity (downwards), light (upwards) or a prevailing wind (lateral). The results of using this effect for the parametric system can be seen in Figure 3.6.

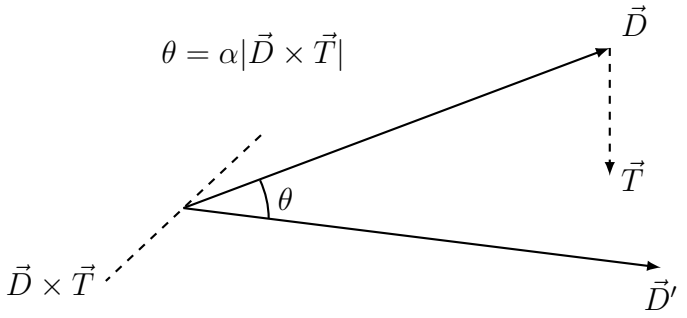


Figure 3.1: Application of tropism $\vec{T}$ to branch of initial direction $\vec{D}$ with resultant direction $\vec{D}'$.

Leaves

For the modelling of leaves, Lindenmayer focuses primarily on smaller plants exhibiting complex flowers and leaf shapes. He therefore suggests the use of symbols within the grammar to generate custom polygons allowing leaves and flowers to be specified manually within the L-system definition. This seems cumbersome for trees given that we are likely to reuse the same leaf shapes at many points in the grammar and for many different tree types. I have instead opted to use the same method as employed in the parametric system, fully described in §3.2.2. Parameters are instead stored as attributes of the `LSystem` and two methods of generating leaves are incorporated within the interpretation of L-systems.

A single leaf can be generated using the `L` symbol, with required parameters **d_ang** and **r_ang** used to calculate its orientation—specifying the angle away from and around the stem respectively. Alternatively, leaves can be generated along a stem using the **leaves** parameter of the `F` symbol. This indicates that the specified number of leaves will be distributed uniformly along the stem with **leaf_r_ang** and **leaf_d_ang** used to calculate their orientations. The current rotation angle of the previous leaf around the stem is stored as leaves are constructed along the stem, and incremented by **leaf_r_ang** with each leaf, this allows leaves to be spread uniformly around the stem as well as along it.

As the output of the generation phase is parsed, each leaf is added to an array as a position and orientation pair. After initial parsing is complete this array is iterated over and the leaf mesh generated in an identical manner to that described in §3.2.2.

3.1.3 L-System Design

The fixed generation and parsing systems described above are in themselves fairly simple and small in terms of volume of code. A large portion of the complexity of modelling the tree is introduced within the definition of the L-system grammar, that is the axiom and production rules, which is given as input to the system.

The design of the grammar can be intrinsically difficult as the geometric output we desire must be encoded as a series of production rules which will be iterated. Code for such grammars can become fairly large and complex, some examples are provided in Appendix B.

3.2 Parametric Approach

My parametric tool uses a more integrated approach; a parameter list, in the form of a Python dictionary, is given as input and the output model is constructed directly. Any values missing from the input parameters are given default values (those of the quaking aspen).

The branches are first constructed as a series of Bézier curves and then the leaves as a polygonal mesh, using locations determined during branch generation. A significant portion of the work was in translating the many definitions within Weber and Penn's paper into Python for use within the *Blender* plugin. Most effort, though, was focussed on the generation of the output model from the values that these translated formulae gave about the geometry of the tree. Some of the particular difficulties faced during implementation of the tool are described below.

3.2.1 Branches

The paper suggests the use of transformation matrices to store orientation data at different points in the tree as it is constructed. I have instead opted to re-use the turtle developed for use with the L-system approach above as it provides a more intuitive representation to work with while developing the system.

Originally the models were constructed as polygonal meshes directly using a series of tapered cylindrical segments for each branch, as seen in Figure 2.3. I have instead chosen to use Bézier curves to model branches, these allow for more smoothly curving representations and are very simple to specify.*Blender* also provides the ability to dynamically alter the resolution of the Bézier curve after generation. This provides a major benefit in terms of adaptive levels of detail, as previously the entire mesh would need to be regenerated for each resolution.

Without With

Figure 3.2: Effect of **radius_mod[n]** parameter on weeping willow.

Branch Radius

The uniformly tapering cylinders of the original model can be replicated easily using *Blender*'s Bézier curve bevelling functionality (as in Figure 2.1). The model also includes provision for flaring of the trunk near the base and non-uniform and periodic tapering along the entire length of stems.

In these cases the ordinary behaviour—assigning radius at each of the Bézier points in the branch, typically between 5 and 15, specified by **curve_res[n]**— is not sufficient, a higher resolution of radius control along the length of the branch is required. This is achieved by evaluating the curve for each segment at a number of points spaced evenly along its length and taking these as the new Bézier points, with their control points determined by evaluating the tangent to the original curve at each point. The size of the vectors to each new control point are rescaled in order to preserve the approximate shape of the curve.

By carrying out this increase in resolution a much higher degree of control is gained over the radius of extrusion along the curve, without noticeably affecting the path of the curve itself. This operation is fairly expensive so is only performed for the trunk (as flaring is necessary) and for branches which have a value of **taper[n]** which indicates periodic tapering (see Appendix C).

Weber and Penn's model does not allow for extraordinary changes in radius between levels of branching, this was noticeably a problem for the weeping willow (Figure 3.2). I have therefore introduced the **radius_mod[n]** parameter which the original branch radius value is multiplied by. For the weeping willow **radius_mod**[**2**] = 0.1 is used so that the secondary branches are significantly thinner and consequently more realistic.

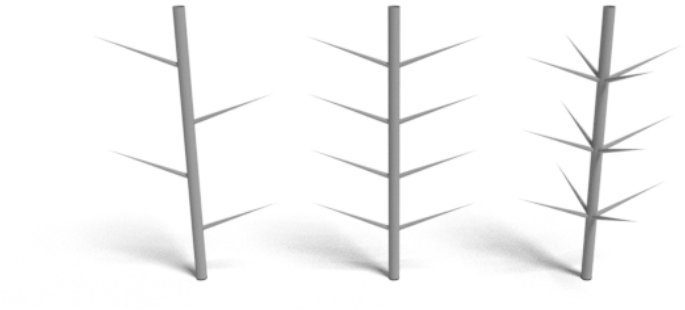


Figure 3.3: Alternate, opposite and whorled branching.

Branch Distribution

One major shortcoming of Weber and Penn's model is the lack of any clear consideration for how the child branches of a stem are distributed along its length. Looking at trees in a botanical context, we observe three types of branch distribution along a stem: alternate, opposite and whorled [Ken], as shown in Figure 3.3. Alternate is by far the most commonly observed type of branching in trees; opposite monopodial branching is quite rare, though perhaps useful for leaf distribution in some cases, and whorled branching is seen almost exclusively in pine trees.

I have chosen to build the tree geometry using a recursive procedure, each stem is built in a series of segments with all child stems for each segment built before moving on to the next segment in the parent branch. This means that it is not necessary to keep track of all stems and their parameters at a specific level in order to construct each level iteratively. Instead, a recursive call is made for each child branch, stopping when the maximum number of branching levels is reached.

The number of child branches per segment is calculated from the total number for the branch as defined in [WP95, §4.3] using a method similar to Floyd-Steinberg error diffusion [FS76]. The fractional offset for each child along the segment is then determined based on the branching pattern (alternate, opposite or whorled) defined by **branch_dist[n]**[1]. Each child is then positioned by evaluating the Bézier curve between the start and end points of the segment at the fractional offset for that child.

The child branch's `forward` vector is determined by calculating the tangent to the curve and then modifying this according to **down_angle[n]**, **rotate[n]** and the current branching pattern. The `right` vector of the turtle must also be maintained, this is calculated based on the turtle orientation at the end of the segment and the child's `forward` vector.

$$right_{child} = (forward_{seg\_end} \times right_{seg\_end}) \times forward_{child}$$

This approximation is valid in general because the branches typically lie within the plane normal to the `right` vector of the turtle at the end of the segment. If the branch does not lie within this plane, due to a non-zero **bend_v[n]** value, then this method still gives a visually pleasing result while remaining easy to compute.

[1] A detailed explanation of **branch_dist[n]** is given in Appendix C.

In the case of helices the approximation is not valid, so to preserve a correct orientation along the curve the child turtle's `right` vector is instead taken as the cross product of the tangent evaluated a short distance further along the curve with the child turtle's `forward` vector (the tangent at the original location). This ignores the roll of the turtle about the branch at the start of the segment, but provides a consistent `right` vector along the helix, so gives a good visual result.

The above method will, however, position the start of the child branch at the centre of the cross section of the parent stem which can give poor visual results for children of stems with a large radius. I therefore chose to shift the start position of the branches toward the circumference of the parent. This is achieved by maintaining a turtle which is positioned at the original start position of the child and is directed normal to the parent curve, at the same rotation about the parent curve as the child's direction. Then by moving the turtle forward by an amount equal to the radius of the parent minus the radius of the child, a new start position for the child is determined which is as close as safely possible to the parent's circumference.

Helical Branches

Weber and Penn's parameterisation includes an option for stems which form a helical shape, though the paper qualifies this quite poorly, stating only that "a special mode is used when **curve_v[n]** is negative. In that case, the stem is formed as a helix. The declination angle is specified by the magnitude of **curve_v[n]**" [WP95, §4.1]. No currently available implementations of the system include this functionality as it presents a significant implementation challenge.

Firstly, a method to model helices using Bézier curves must be devised. Riškus presents a method to fit a Bézier curve to a circle of radius r, drawing an arc on the circumference through angle 2α [Riš06]. The curve is then defined by the points (in 2D)

$$
\begin{aligned}
b0 &= (r\cos\alpha, -r\sin\alpha) \\
b1 &= \left(\frac{4r - r\cos\alpha}{3}, -\frac{(r - r\cos\alpha)(3r - r\cos\alpha)}{3r\sin\alpha}\right) \\
b2 &= \left(\frac{4r - r\cos\alpha}{3}, \frac{(r - r\cos\alpha)(3r - r\cos\alpha)}{3r\sin\alpha}\right) \\
b3 &= (r\cos\alpha, r\sin\alpha)
\end{aligned}
$$

Where $b0$ and $b3$ are the endpoints of the curve, and $b1$ and $b2$ the control points.

To form a helix the z coordinates of these points must also be determined. The definition of a helix parameterised by t states that $z = \frac{pt}{2\pi}$ for any point on the helix, where p is the pitch of the helix. An arc through angle 2α will therefore rise by $\frac{\alpha p}{\pi}$. So the z coordinates of the endpoints are given by:

$$
\begin{aligned}
b0_z &= -\frac{\alpha p}{2\pi} \\
b3_z &= \frac{\alpha p}{2\pi}
\end{aligned}
$$

The z coordinates of the control points, however, are considerably more complex. Juhasz proposes a method where these z coordinates are constrained such that the osculating plane of the curve at the end of the initial segment is common with the osculating plane at the start of the next segment [Juh95]. This retains G2 continuity and gives the following:

$$b1_z = -\frac{(r - r\cos\alpha)(3r - r\cos\alpha)\alpha p}{r\sin\alpha(4r - r\cos\alpha)\tan\alpha}$$
$$b2_z = \frac{(r - r\cos\alpha)(3r - r\cos\alpha)\alpha p}{r\sin\alpha(4r - r\cos\alpha)\tan\alpha}$$

For this application, I found that a value of $\frac{\pi}{2}$ for α was sufficient to give an adequately high resolution Bézier curve, and simplifies the calculation significantly, resulting in the Bézier points:

$$b0 = \left(0, -r, -\frac{p}{4}\right)$$
$$b1 = \left(\frac{4r}{3}, -r, 0\right)$$
$$b2 = \left(\frac{4r}{3}, r, 0\right)$$
$$b3 = \left(0, r, \frac{p}{4}\right)$$

The input parameters r, the radius of the helix, and p, the pitch must also be calculated. The paper states that the declination of the helix is $|\mathbf{cure\_v}[\mathbf{n}]|$, so $\theta = 90 - |\mathbf{cure\_v}[\mathbf{n}]|$ is taken as the angle (in degrees) the tangent to the helix makes with the plane whose normal is the axis of the helix. It follows that

$$\tan\theta = \frac{b1_z - b0_z}{b1_x - b0_x}$$

as $b1_y = b0_y$. Which gives

$$r = \frac{3p}{16\tan\theta}$$

The length of the branch $L = \mathbf{length}[\mathbf{n}]$ is defined within the input parameters, as is the resolution of the curve $m = \mathbf{curve\_res}[\mathbf{n}]$. Interpreting m as the number of rotations through α the helix will make along its length, and L as the total height of the helix gives

$$L = \frac{\alpha p m}{\pi}$$

and with $\alpha = \frac{\pi}{2}$,

$$p = \frac{2L}{m}$$

This provides all the required information to construct the points describing one half spiral of the helix.

To construct the entire branch from this, firstly, the base spiral points are reoriented so that the axis of the helix and the branch direction are collinear, then the helix is rotated around this axis by a random angle to introduce some variation between branches. The

Figure 3.4: Black Oak.

Figure 3.5: Tree with helical trunk.

base helix points are then translated so that $b0$ is located at the start position of the branch, and all other points retain their positions relative to $b0$. Looping for n iterations, the rest of the branch is constructed by rotating the base helix points by $\frac{\pi}{2}$ around the axis of the helix (i.e., the branch direction) with each iteration and translating them such that the start point of the current segment is equal to the endpoint of the last.

The calculations to determine the characteristics of the helix are therefore only carried out once, with the points then just being rotated and translated with each iteration, making this quite an efficient method of constructing such branches. The radius of the resulting Bézier curve is tapered using the standard method resulting in an effective approximation to a spiralling branch. This method also enables the resolution of the Bézier curve to be increased to allow for flaring and periodic tapering, as described above, using the same method as non-helical, branches. Child branches can also be distributed along the stem in the same manner as for ordinary branches.

This sort of branch is typically used for small twigs at the third or fourth level of branching such as in the black oak (Figure 3.4). Though can also be used to model trees which have been trained by humans for aesthetic effect, such as in Figure 3.5.

Tropism

Weber and Penn include an **attraction_up** parameter which applies to branches in the second and deeper levels of the tree. This affects their growth direction either upward towards light, or downwards due to gravity. While this is a good approximation in many cases, it ignores lateral effects on the tree—typically due to a prevailing wind. I therefore make use of the more general tropism vector described above in §3.1.2, specified using the **tropism** parameter.

As with the original **attraction_up** parameter, the vertical component of the tropism vector is only applied to secondary or deeper level branches as excessive effects on the trunk and primary branches result in degraded appearance otherwise. Lateral tropism is applied to all levels of the tree, these effects can be seen in Figure 3.6.

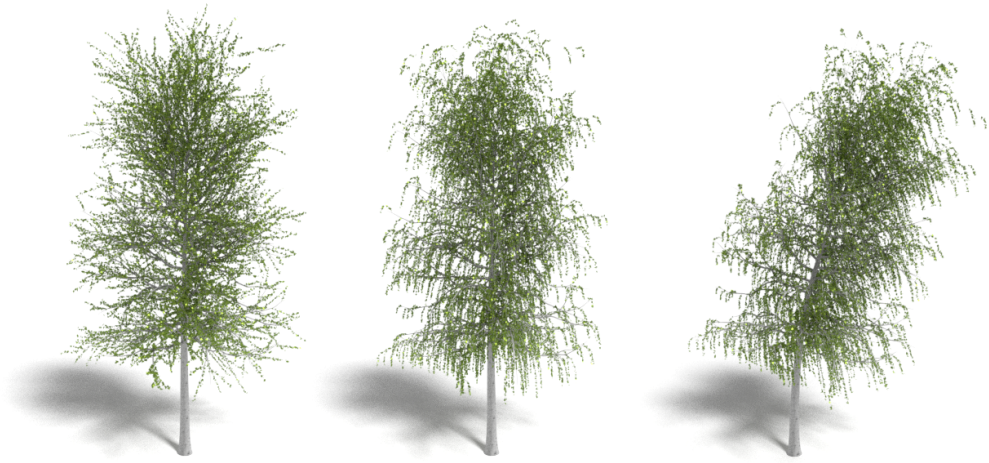


Figure 3.6: Effect of **tropism** parameter on the silver birch.
No tropism, vertical tropism and vertical and lateral tropism.

Pruning

Pruning is introduced by Weber and Penn to control the shape of particularly unruly trees for which the standard parameters alone cannot adequately describe their overall shape [WP95, §4.6], one clear example is the weeping willow.

First a test pass is performed in which the basic branch geometry is generated but not added to the final model, and with no recursive calls for child branches. If at all points along its length the branch remains within the envelope then the branch is simply added to the model, otherwise it is retested with a reduced total length. This process is repeated until the branch is contained within the pruning envelope or it is less than 15% of its original length, at which point the branch is discarded entirely. All random aspects of the branch must be identical with each iteration for this method to be valid. Fortunately Python's `random` module provides the methods `getstate` and `setstate` which are used to ensure that the pseudo random number generator starts with the same state for each iteration and therefore produces valid pruning results.

Figure 3.7: Effect of the **prune_ratio** parameter for values 0, 0.5 and 1.

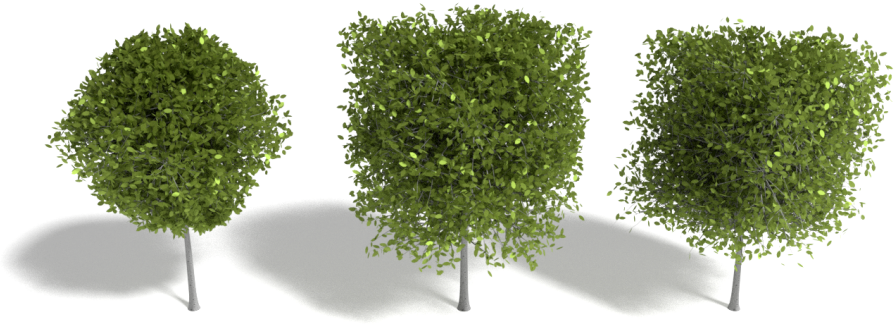


Figure 3.8: Spherically, cylindrically and cubically pruned trees.

Figure 3.9: Tree pruned based on contextual objects.

The severity of pruning is controlled by the **prune_ratio** parameter, as is visualised in Figure 3.7. This is applied by finding the fitting length for the branch, then interpolating between this and the original length according to the value of **prune_ratio**.

The model presented for the pruning envelope is that of a solid of revolution around the z-axis. This is generally very good for modelling the shapes of trees, though in some cases we desire a more arbitrary shape. The above method simply requires a point test to determine if the segmentation points of a branch are within the envelope. The point test can, therefore, be arbitrarily defined to generate a wide variety of rotationally non-uniform shapes, such as the cube shaped tree in Figure 3.8. This can be extended to account for other objects in the environment which real trees would grow around, such as buildings and walls. A basic example is given in Figure 3.9.

Figure 3.10: Bamboo.

Multiple Trunks

In order to extend the parameterisation to allow for modelling of trees and plants with multiple trunks from ground level, I added the **floor_splits** parameter. An example usage of this parameter is the bamboo in Figure 3.10.

The start point of each trunk is distributed within a circle of radius proportional to the radius of each stem using Poisson disc sampling with a minimum separation distance also proportional to the radius of each stem. The start orientation of the trunk is then modified such that it faces outwards from the centre of the circle allowing branches to curve primarily outwards, as seen in the bamboo. Each trunk is generated using the standard procedure as if it were an isolated tree.

If we wished to automate generation of a large collection of trees, such as those in a forest, then a similar method could be used, distributing trunks over a larger area. In this case it would likely be beneficial to use a faster method of Poisson disc sampling such as that proposed by Bridson [Bri07].

3.2.2 Leaves

Leaves are treated in a very similar way to branches, utilising the values of **down_angle[n]**, **down_angle_v[n]**, **rotate[n]** and **rotate_v[n]** in exactly the same way. The number of leaves on a branch is also calculated in a very similar manner to child branches, though is instead based on **leaf_blos_num**. Leaves are added only to the final, deepest, level of branches; so for a tree with **levels** $= 3$ the leaves would use the parameters for $n = 4$. There are a number of predefined leaf shapes[2], the shape used for a tree is chosen using the parameter **leaf_shape** and can be scaled in totality or just in the x-direction.

To construct the output leaf mesh, an array of leaf positions and orientations is maintained. This is populated as the branches are constructed recursively; when the final level of recursion is reached the position and orientation of the leaf, determined as they would be for child branches, are appended to the array rather than constructing a further level

[2]Ovate, linear, cordate, maple, palmate, spiky oak, rounded oak, elliptic, rectangle and triangle.

Figure 3.11: Apple.

Figure 3.12: Hill Cherry.

of branching. Then once the branch construction is complete, this array of leaf locations is iterated over and the base leaf polygons defined for the current leaf shape transformed to the relevant location and orientation. The transformed polygons are then added to a single mesh which will ultimately contain all leaves for the tree.

When the base polygon is transformed an additional re-orientation is also performed, with its effect fractionally controlled by **leaf_bend**. This rotates the leaf such that it faces upwards and outwards to simulate the effect of sunlight on the growth pattern of the tree [WP95, §4.9].

Blossom

One common element in the growth cycle of many trees, which Weber and Penn's original model ignores entirely, is blossom. Their model can very easily be expanded to accommodate this; I have introduced the **blossom_rate** parameter to describe the percentage of the total number of leaves and blossom on the tree which are blossom. The **blossom_shape** and **blossom_scale** parameters are also introduced to specify the geometry of the blossom themselves, again using a set a predefined shapes[3].

While iterating over the list of leaf positions as described above, a random choice is made for each leaf (based on the value of **blossom_rate**) as to whether it will be a leaf or blossom. The rest of the procedure is identical except that the base polygon used is for the selected blossom shape rather than the leaf, and is added to a separate blossom mesh.

The results of this simple modification are quite effective, as can be seen in the apple (Figure 3.11, **blossom_rate** = 0.35) and hill cherry (Figure 3.12, **blossom_rate** = 1).

[3]Cherry, orange and magnolia.

3.3 Extensions

Given that the parameter list for my implementation of Weber and Penn's method is quite extensive and complex, I hoped to simplify the process of tree design in order that artists could more easily utilise the system. I had two ideas as to how this may be easily achieved. Firstly, reducing the number of input parameters required through dimensionality reduction, and secondly, automated determination of parameters given a goal in image form.

3.3.1 Dimensionality Reduction

To perform dimensionality reduction on the parameter list I opted to use the principle component analysis (PCA, also known as KLT or the Hotelling transform) functionality of `scikit-learn`. This uses singular value decomposition (SVD) to project the data to a lower dimensional space. The aim being to map the standard parameter space down to just a few meta-parameters which the artist can then easily control.

The most intuitive mapping would probably be to a two-dimensional space with labeled points marking specific tree breeds. The artist can then choose a point to represent the blend of trees they want to produce. Alternatively a number of specific meta-parameters may be identified, though it is hard to devise a suitable number and map these to meaningful behaviours via the full parameter list. The two-dimensional results are somewhat successful, though there are a number of issues with the method as further explored in §4.4.1.

3.3.2 Genetic Algorithm

To devise parameter lists automatically I chose to use a genetic algorithm. The algorithm starts with a randomised parameter list and generates a population by copying this and permuting a randomly selected parameter by a random amount proportional to its current value. The permutation process also has a small chance of flipping the sign of the parameter value to incorporate the negative flags used for some parameters, these values would not be reached by gradual incrementation alone as the visual result going from positive to negative is non-monotonic.

Blender's command line interface allows for the generation script to be executed and a black and white render of the resulting model produced automatically. This is then compared to the goal image using the `misc.imread` functionality of the `scipy` library to obtain a simple pixel difference value between the images. This is used to rank the members of the current generation by their fitness (inverse of pixel difference). This is a fairly crude fitness function, but does provide meaningful results. The generation and rendering of each member of the population are preformed in parallel using the `subprocess` module.

The two fittest parameter lists from the current generation are taken and merged together to form the basis of the next generation. Using the same permutation procedure as above, a new population for this generation is generated and evaluated using the fitness

function. This process is repeated for a very large number of generations, hopefully resulting in a parameter list giving a very high fitness value.

Initially I chose to use an elitist approach where the current best parameter lists were only taken if the error was an improvement on the previous generation, though I found better results were achieved by always taking the current best, even if they were worse than the last generation, so as to avoid getting stuck in local maxima. During the execution of the genetic algorithm the seed from which the tree is generated was also fixed so that the only change in output image between generations is due to the altered parameter, and not random variation caused by changing the seed.

I decided to limit the goal imagery to black and white renders of trees generated using my system with known parameters in order to simplify the implementation due to the time constraints of the project. Though the principle could certainly be expanded to higher information imagery or even evaluation of fitness of the model geometry directly provided an adequate fitness function could be developed.

The results of the algorithm can be seen in §4.4.2.

3.4 Summary

As detailed above, I have completed a general purpose L-system implementation in python, as well as a parser to translate the symbol lists generated by the iterated L-systems into 3D models within the *Blender* application.

In addition, I have implemented a tool based on Weber and Penn's parametric model for tree generation within *Blender* and explored the effects of dimensionality reduction on the parameter lists used as input for this system. Finally, I have developed a genetic algorithm which attempts to automatically devise an input parameter list which results in a tree similar to one pictured in a given 2D goal image.

Chapter 4

Evaluation

There are three critical aspects of evaluation for the two tree generation systems which I have developed. Most importantly is the visual result, this may be in 3D when used in a game or film, or in 2D if rendered to a static image. Secondly is the performance of the system; it must be reasonable for the system to be used as part of the conventional work flow for someone producing content for the forms of media discussed. A sensible benchmark is the time taken for an experienced artist to model a tree by hand, which is in the order of minutes to hours[1]. Finally is the usability of the system; it must be accessible to users who may have no, or limited knowledge of programming concepts but are competent artists or content creators.

4.1 Visual Result

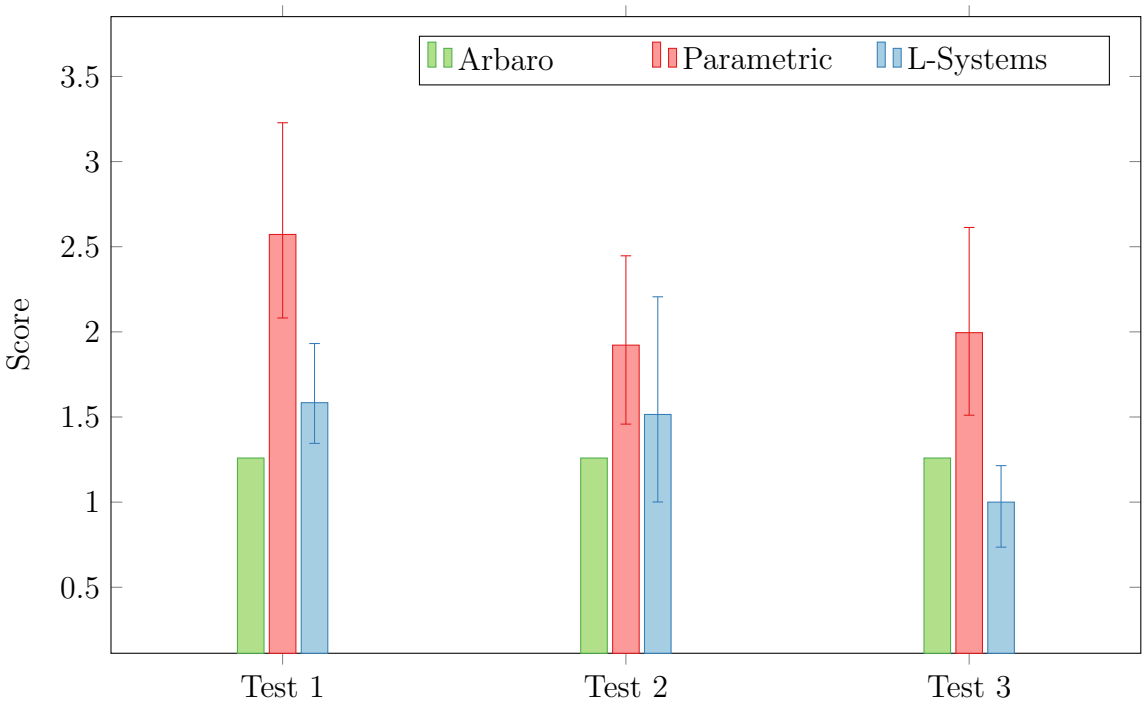
For evaluation of the visual results of the systems, I chose to limit my investigation to 2D renders of the resulting trees (similar to those in Appendix A). The conclusions should still be somewhat representative for 3D, and it is a great deal more difficult to accurately assess the measures I wished to effectively in 3D—either a video of the tree model or some interactive game-like environment would need to be produced.

In order to determine if my implementations are successful I decided to compare them to an existing implementation of Weber and Penn's model which, for the purposes of evaluation, I assume to be a successful implementation. This existing system is a Java app called *Arbaro* [Die15] which can generate `.OBJ` files which can be imported into *Blender* and rendered using the exact same method as my own models. In order to asses the relative quality of the rendered models from each system I chose to use a pairwise comparison method similar to that described by Silverstein and Farrell [SF01], this was achieved using Rafał Mantiuk's `pwcmp` library [Man17].

I therefore selected a number of models from each system which were then rendered in an identical way; each survey consisted of a number of pairs of these renders. Each pair used trees of the same type (aspen with aspen, willow with willow etc.) generated using different systems. The range of different comparisons were distributed evenly across

[1] A small group of artists were surveyed, they estimated that to produce models of comparable quality to the parametric tool would take between 30 minutes and 2 hours depending on the type of tree.

Figure 4.1: Scores derived from survey of visual results for 3 different test designs.



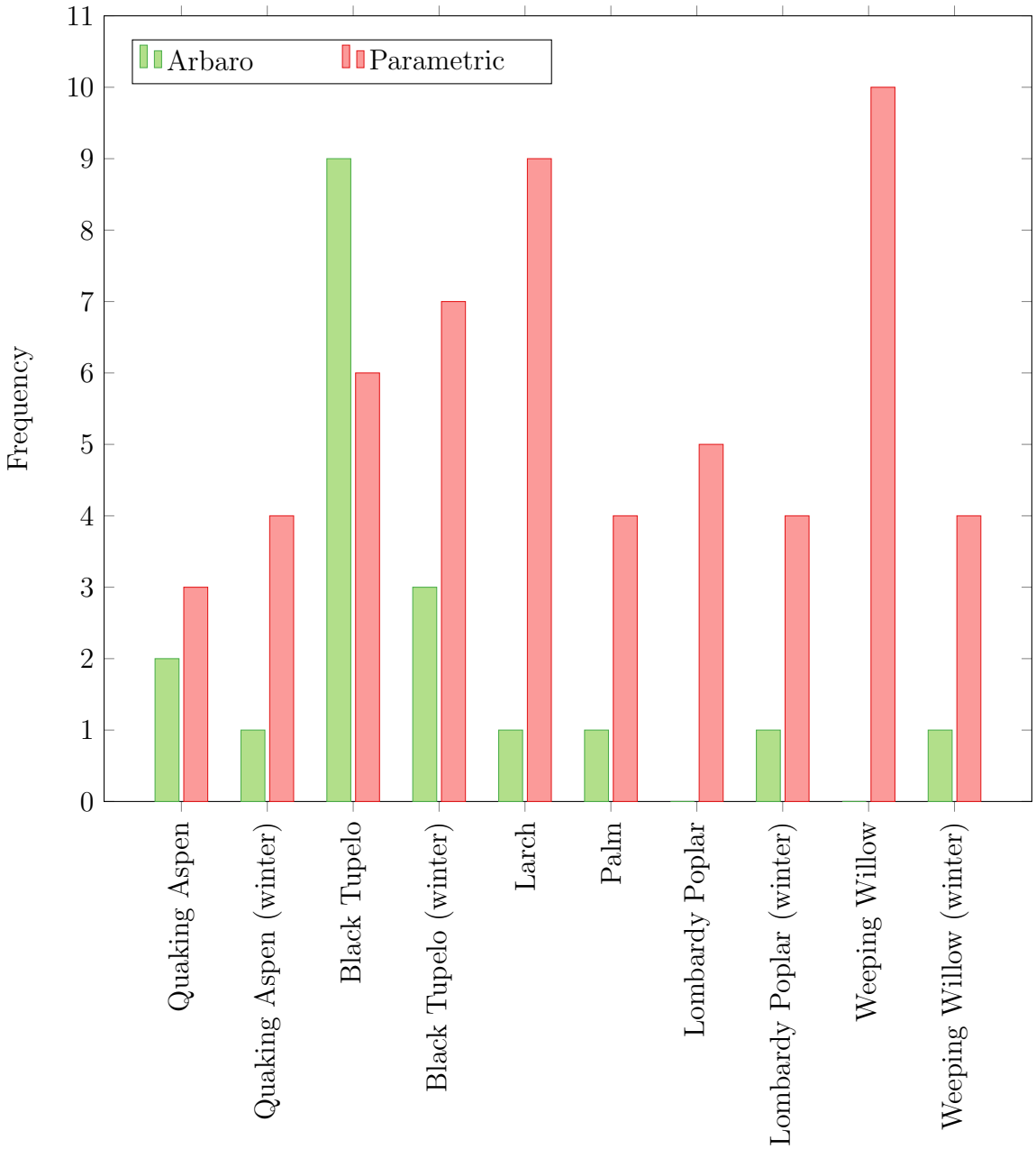
three distinct survey designs, or tests. Participants had to select which of each pair they deemed to be the most realistic. A group of fifteen participants were surveyed, each given fifteen comparisons, the results were then tabulated and final scores calculated for each system.

This method gives an average realism score across all tests of $1.313^{+0.600}_{-0.480}$ for the parametric system and $0.325^{+0.418}_{-0.339}$ for the L-system based tool, both relative to a baseline of 0 for *Arbaro*. This indicates that the parametric system performs better than *Arbaro* in a significant way and is therefore a successful implementation. The L-systems tool, however, does not outperform *Arbaro* significantly given the large relative negative uncertainty of the score. It therefore only produces trees of comparable, not definitively higher, quality to *Arbaro*; this is arguably still a useful implementation as it approximately equals the currently available tool.

As visible in Figure 4.1, the above conclusions hold on a test by test basis. The parametric tool outperforms *Arbaro* significantly for all tests, while the L-systems tool outperforms *Arbaro* in Test 1, is not conclusively better or worse in Test 2, and is significantly worse in Test 3. This demonstrates that the parametric system provides the best visual results across a range of tree types spread across the different survey designs.

I do not have a sufficient number of samples to carry out the above analysis for each individual tree type reliably, though looking at the raw comparison data (Figure 4.2) can provide some insights. As expected we see that in the majority of cases my parametric tool is preferred to *Arbaro*, with it chosen by all participants for the Lombardy poplar and weeping willow. Though there are still cases where *Arbaro* does produce models of greater realism, such as the black tupelo.

Figure 4.2: Raw comparison data by tree type for parametric system against *Arbaro*.

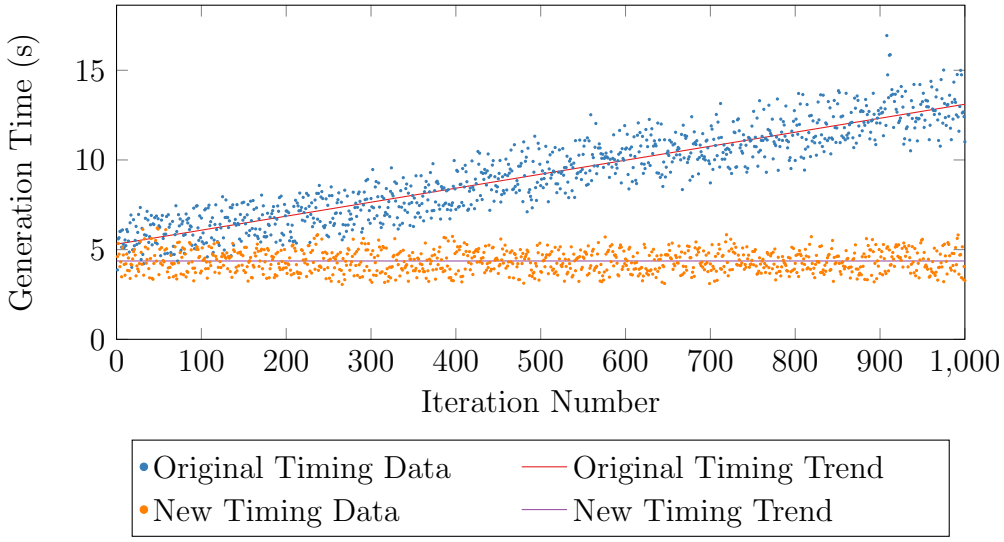


4.2 Performance

4.2.1 Data Collection

In order to gather data on the performance I included in both tools provision to output times taken to complete various aspects of the generation process[2], as well as various complexity measures for the resulting models. I then created a short script that would enable a large number of executions of the generation process to be performed automatically with the results written to a file. Initially, I intended to run the entire timing process within the *Blender* application. After some testing I noticed a systematic error present in the results obtained; over time the generation times were increasing dramatically as can be seen in Figure 4.3.

Figure 4.3: Timing data for generation of 1000 quaking aspen models using the parametric system with two alternative timing mechanisms.



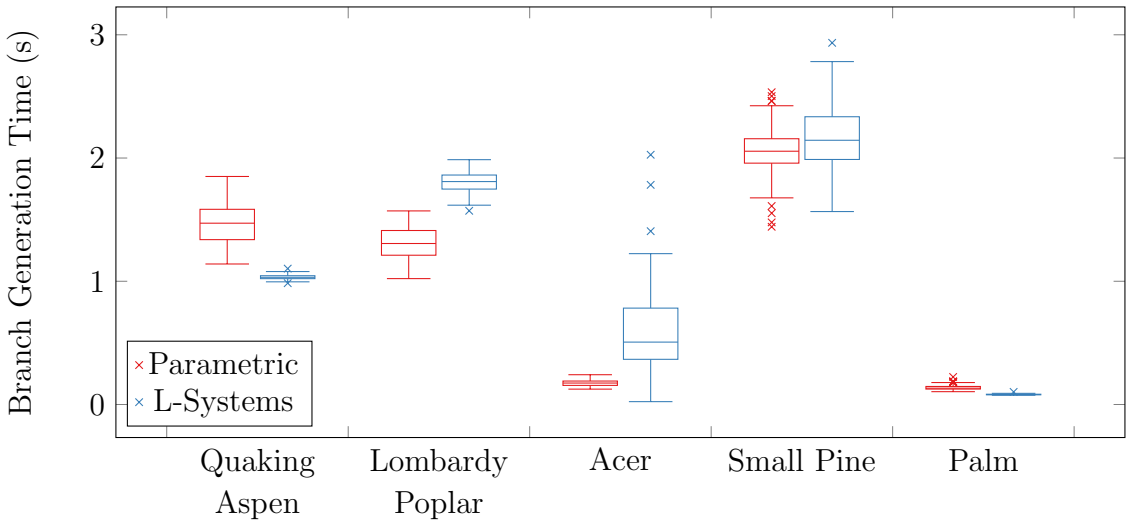
After some research, it seems that this was due to *Blender* caching the geometry of the resulting models, even if the model was deleted after each iteration, resulting in increasing memory usage and slowdown. It is not possible to avoid this issue working entirely within *Blender*, so I instead opted to run the timing script externally and invoke the generation script via *Blender*'s command line interface. The output of the process was then parsed and relevant information extracted and written to file. This added some overhead to the timing procedure itself as *Blender* had to be initialised for each execution, but eliminated the systematic error as shown in Figure 4.3.

4.2.2 Generation Time

Both the parametric and L-system based approaches generate models in a time in the order of seconds, with a maximum of around 400 seconds for weeping willow. All these times are significantly shorter than it would take a skilled artist to create a model of

[2]Measured using 2013 MacBook Pro: 2.8 GHz Intel Core i7 - 16 GB 1600 MHz DDR3

Figure 4.4: Generation time for 100 instances of 5 trees types generated using each system.



similar quality by hand. In this respect the performance of both systems is certainly adequate to qualify as a successful implementation.

It is difficult to make any reasonable conclusions as to the relative speeds of the two systems. In general the time taken to generate the same type of tree using both systems is fairly similar, as shown in Figure 4.4. There does not seem to be any clear trend for one system to be quicker than the other; in some cases such as the Lombardy poplar the parametric system is faster, while in others such as the quaking aspen the L-system is quicker.

The spread of speeds for the L-systems tool for each tree also seem to vary by a much larger amount than the parametric tool. With the quaking aspen having a very small spread, while the acer and pine have much higher spreads. The L-systems tool's performance seems to vary wildly, a result that makes sense given the arbitrary nature of the input L-system grammars. The performance of the parametric tool, however, seems to be a deal more predictable.

As shown in Figure 4.5A, branch generation time grows exponentially with branch complexity[3], bounded by $O(n^3)$ and $\Omega(n^2)$ - the dashed line plotted on Figure 4.5A is raised to a power of 2.7. The constant of proportionality, however, is extremely small giving us reasonable performance, even for very complex trees such as the weeping willow with a branch complexity of ~ 160000.

Again, the results for the L-systems tool seem to vary drastically based on the tree type that is being generated. As can be seen in Figure 4.6, the Acer seems to follow an approximately $O(n^2)$ complexity, while the Lombardy poplar and small pine are clearly linear in branch complexity. Others like the palm and quaking aspen are very tightly clustered with no clear trend. This makes it very difficult to predict the performance characteristics of the system in general given that it is largely dictated by the input L-system grammar which is entirely arbitrary.

[3]The number of Bézier points in the tree, proportional to $\sum_n$ **branches**[**n**] $*$ **curve_res**[**n**]

Figure 4.5: Timing results for two generation phases of parametric tool.

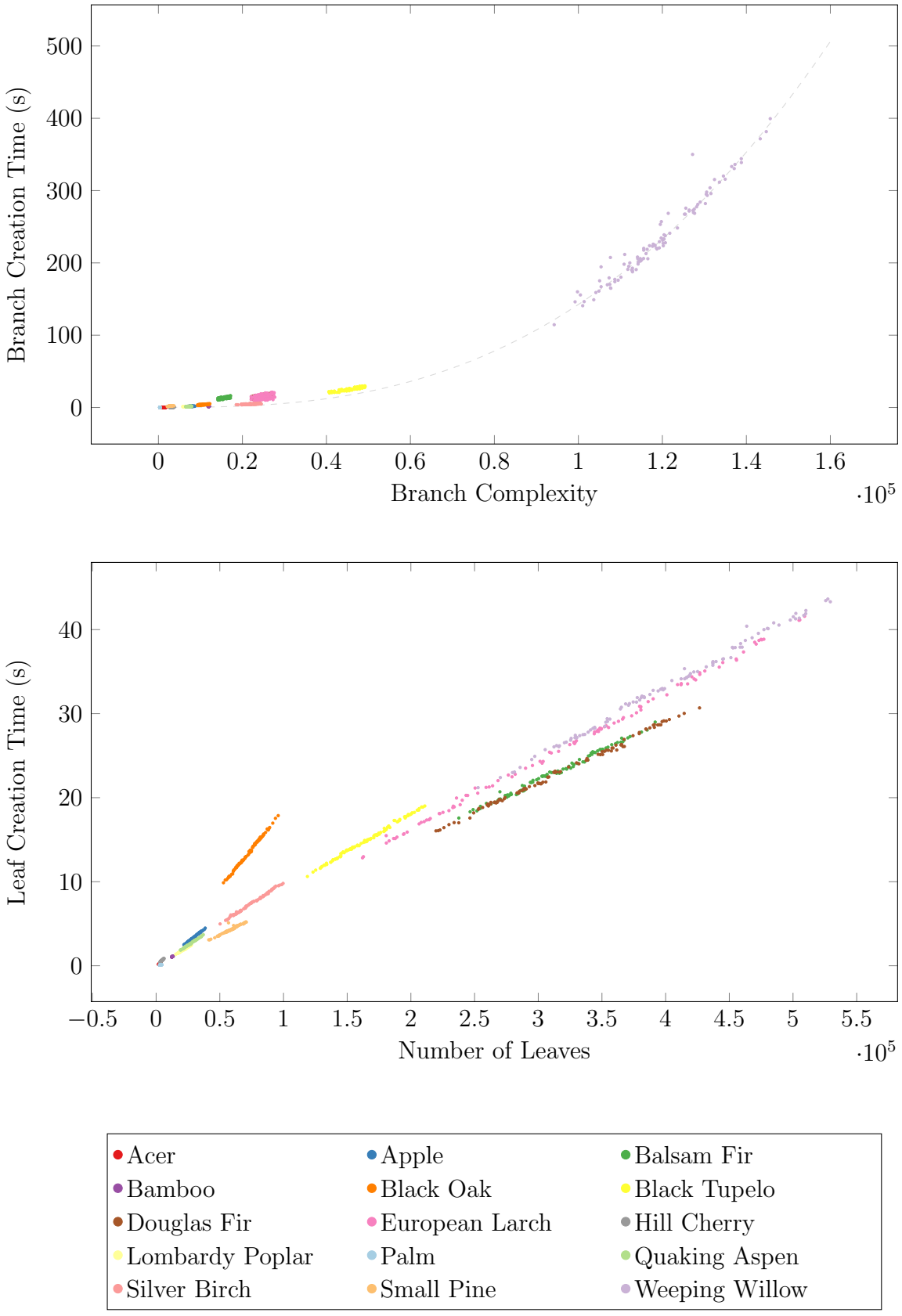
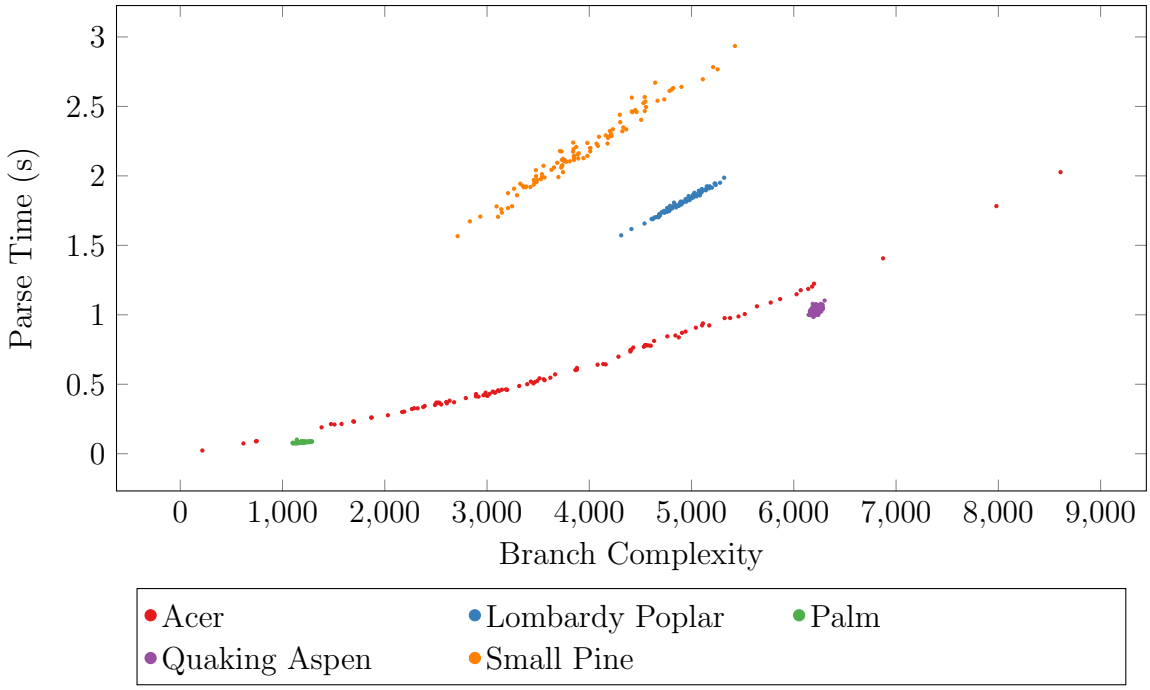


Figure 4.6: Timing results for parsing phase of L-systems tool.



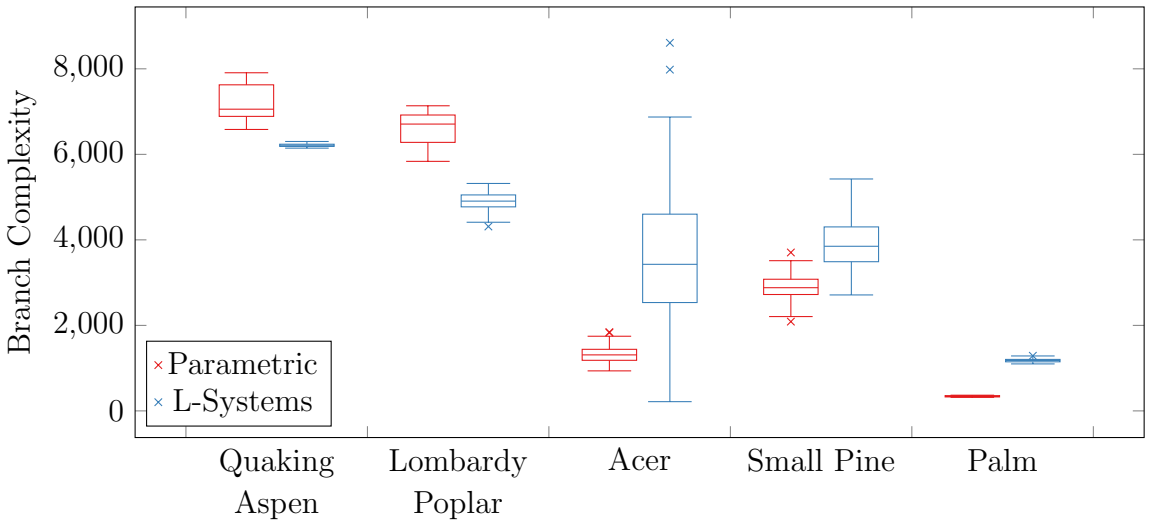
Leaf generation time is $O(n)$ in the number of leaves on the tree, controlled by **leaf_blos_num**. The constant of proportionality is defined by the number of vertices in the leaf and therefore controlled by **leaf_shape**. For leaves with a high number of vertices, such as the oak leaf, the constant is significantly higher as can be seen for the black oak in Figure 4.5B. By contrast leaves with fewer vertices, such as pine leaves, result in a much lower constant as seen for the Douglas and balsam firs. The leaf time characteristics for both systems are near identical as the code is largely common between systems.

It is possible to drastically improve the speed of generation by exploiting parallelism. *Blender* does not allow for multi-processing within the application, but as the output of the system is just a series of Bézier points and polygons we could easily abstract away from *Blender*. The libraries *Blender* provides can also be used outside of the application, further facilitating this option. Issues may be introduced where determinism is required, such as pruning, as multi-processing would potentially disrupt this. Limiting the level of parallelism to ensure safety in these cases would certainly be feasible, and in cases where pruning is not required there should be no issue.

4.2.3 Model Complexity

Another aspect of the performance of the system is the complexity of the resulting models. Ideally we want to produce the least complex model possible for a good visual result. Again it is not clear that either the parametric or L-system based tool is best in this respect from Figure 4.7. As before, the behaviour of the parametric system seems to be more predictable, with variance roughly proportional to the magnitude of complexity. In contrast to the the L-system which varies drastically; from complex models like the quaking aspen with very low variance to models with a extremely large range of complexities

Figure 4.7: Branch complexity for 100 instances of 5 trees types generated using each system.



like the acer.

It is unlikely that either automated system outperforms a skilled human artist in producing visually pleasing models of minimal complexity. A human can optimise specific aspects of the model based on how they perceive it in order to eliminate unnecessary vertices, a task which a computer will struggle greatly to emulate. However, given the much quicker generation process this tradeoff may be acceptable; a human can always manually optimise the generated model post construction. Making use of the Bézier curve functionality in *Blender* also allows for dynamic alteration of the resolution of the resulting polygon mesh after the model has been generated.

4.3 Usability

Usability is also a key aspect of each system. The parametric system has a large number of input parameters, but they are quite intuitive and one can quite easily visualise the effect of changing a numeric parameter on the geometry of the model. With a small amount of use the design process can become fairly quick; I have found that at the end of the project I can effectively design a new tree type in just a few minutes. By designing each level distinctly, starting with the trunk, and generating only up to the level being designed we can also obtain rapid visual feedback for parameter changes.

Currently the input must be defined as text, but it is feasible to develop a user interface for this system with sliders and selection boxes to further simplify the design process for artists. Other methods of simplifying the design process are also explored in §4.4.

L-system design is a great deal more difficult; it is very challenging to visualise the output geometry of a given system. We first have to determine the result of iterating the production rules and then how this will be geometrically interpreted. As a result, developing grammars for new tree types can take far longer than designing for the parametric tool. There is also the issue that the grammar must be specified directly in Python, re-

quiring some programming knowledge from the user. This is not feasible given that the intended users are primarily artists and content creators, not programmers. Alternatively a parsing system to translate formal definitions (as used in §2.3.1) to Python could be developed, though this would further slow the generation process down and still require the theoretical knowledge to design a formally defined L-system.

There is some consideration necessary as to the expressivity of each system. The parametric system is comprehensive enough to represent a huge variety of tree types, as can be seen in Appendix A.1, though is still limited to the fixed parameters provided. The L-system approach, by contrast, can represent entirely arbitrary structures. It would even be possible to entirely encapsulate the parametric system within an L-system grammar, though this would be a great deal less efficient. While this enhanced expressivity can be useful, there is little need for it given the quite restricted use cases of the tool, particularly with the associated sacrifice in general usability.

4.4 Extensions

4.4.1 Dimensionality Reduction

The process described in §3.3.1 proved somewhat successful at producing appropriate output parameters when mapping down to two dimensions, though had some clear shortcomings. Mapping to a number of dimensions greater than two proved difficult as the resulting dimensions could not be assigned any logical interpretation which an artist could use to visualise the resulting tree.

As Figure 4.8 demonstrates, we can quite successfully generate parameter lists which merge the appearance of two other trees by extracting the output parameters for a coordinate half way between these two trees in the lower dimensional space. The number of input trees parameters from which the data is being fit (fifteen) is perhaps not sufficient to provide full enough coverage for this to be a useful tool at present, though does serve as a proof of concept that the principle of dimensionality reduction may result in a useful simplification of the parameter selection process.

Figure 4.8: Merging (centre) of quaking aspen (left) and silver birch (right) produced using dimensionality reduction.

The key issue with this method is the non-monotonicity of parameters, particularly those using negative flags to specify unusual behaviour, meaning we do not have a direct mapping between parameter value and visual perception. For any parameters (e.g., **length[n]**, **leaf_blos_num**) for which this is not the case, and the reduction works very well. But if we take a merging of, for example, the palm tree and another tree then the taper of the branch will likely appear incorrect due to the non-uniform way in which the **taper** parameter is defined.

4.4.2 Genetic Algorithm

Similarly, the genetic algorithm described in §3.3.2 has provided some fairly successful results, although in a quite restricted domain—certainly it can be considered as a proof of concept. Over the course of a few hours runtime a fitness of around 95-97% based on a measure of pixel accuracy was obtained, see Figure 4.9. Perceptually the result is not quite as impressive as these fitness values suggest, but is somewhat successful nonetheless, as shown in Figure 4.10.

Some aspects of the trees are matched quite accurately; the trunk of the quaking aspen is almost exact and the overall shapes of the trees, primarily a result of branch length parameters, are very accurate. Leaves seem to pose a particular problem due to the high fidelity required in assessing their fitness. This is a very restricted application of the genetic algorithm, working only in two dimensions and two colours, but these results demonstrate that the concept is certainly a promising one. The runtime is also an issue, though optimisation is certainly possible, a major issue here was the necessity to render the output for every member of every generation which would be unnecessary if the fitness was evaluated using the geometry of the model directly.

Stava et al. demonstrate a similar technique making use of Monte Carlo Markov Chains to determine the optimal set of parameters [Sta+14]. They use LiDAR scanning to capture real trees which are used as goals, with fitness evaluated based on the 3D structure of the tree. Their approach generates quite realistic results, given sufficient input data, in times typically under an hour.

Figure 4.9: Progress of genetic algorithm in devising parameters for two goal tree types.

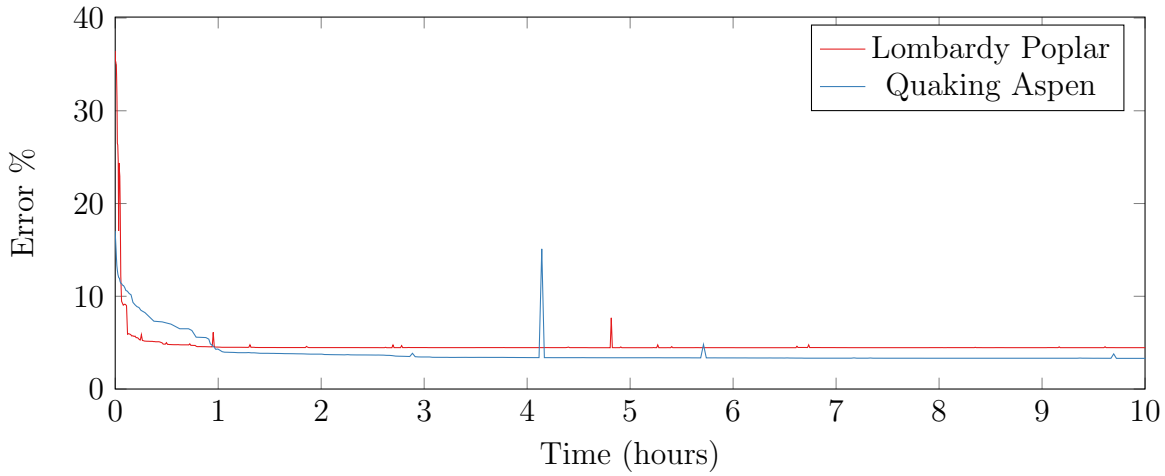


Figure 4.10: Start, goal and output models of genetic algorithm.

Render of tree made using start parameters of genetic algorithm.

Quaking aspen goal image.

Output of genetic algorithm.

Lombardy poplar goal image.

Output of genetic algorithm.

4.5 Summary

From the above we observe that, in terms of visual quality, the parametric system outperforms *Arbaro* in almost all cases and perform significantly better than L-systems based tool in most cases. The performance of the parametric tool is also more reliable than that of the L-systems tool, as is the complexity of the models it produces. Finally, the parametric tool offers significant scope for improvement in terms of usability; a proof-of-concept result was obtained for automatic parameter list design using a genetic algorithm, and PCA provided some interesting results in regard to dimensionality reduction.

Chapter 5

Conclusion

5.1 Summary

In this project I have fulfilled all success criteria. I have implemented tree model generation systems using two alternative approaches and devised a set of input parameter lists and L-system definitions to generate a wide variety of trees (see Appendix A). I have also explored both my extension goals. I demonstrated a moderately successful application of dimensionality reduction to the input used by the parametric system. I also implemented a genetic algorithm to automatically devise parameter lists given a goal image which showed significant promise.

The parametric system was the more successful of the two. Generation times were faster than an artist, realism better than existing alternatives and usability was good, with the potential to be improved via my extensions or the implementation of a full user interface. The L-system based tool was also successful, with similar generation times and visual results comparable to existing alternatives, though the poor usability of the system let it down as a viable tool for artists.

I have enjoyed the project greatly and learned a great deal in its completion; learning Python as part of the project has been good fun and will likely prove very useful in the future. Interpreting and translating research papers was also an interesting challenge and something which I had no previous experience. The project has also given me an experience of academic writing and what it might be like to continue within academia and to do independent research, this is very valuable at a time where I have a choice to pursue further study or go into industry.

If starting the project again, I would like to shift focus from the L-systems approach, which is fundamentally limited by its lack of usability, to better the explore space colonisation and modular approaches outlined in §1.2. I think that these have greater scope for development than L-systems, in particular a hybrid of these systems incorporating some aspects of Weber and Penn's model might produce the best possible results.

5.2 Future Work

Going forward I would like to further develop the parametric system. Implementation of a user interface within the *Blender* plugin would enable content creators to use the plugin in real world projects. I would like to complete this in the near future and release the finished plugin for free use by the *Blender* community.

There are also potential extensions to the generation system itself. In their paper Weber and Penn also outline a methodology for animation of the tree structure to replicate the effects of wind, this could be implemented using *Blender*'s armature features and would greatly improve the visual results for animated content. An improvement of larger scope would be to implement my own application to perform the modelling of the branch structure from the generated Bézier points to enable further improvements in realism such as non-uniform branch cross sections.

Bibliography

[AK84] Masaki Aono and Tosiyasu Kunii. "Botanical Tree Image Generation". In: *IEEE Comput. Graph. Appl.* 4.5 (May 1984), pp. 10–34. ISSN: 0272-1716. DOI: 10.1109/MCG.1984.276141. URL: http://dx.doi.org/10.1109/MCG.1984.276141.

[And] Michel J. Anders. *Space Tree Addon*. URL: https://github.com/varkenvarken/spacetree.

[Bri07] Robert Bridson. "Fast Poisson Disk Sampling in Arbitrary Dimensions". In: *SIGGRAPH '07*. 22. New York: ACM Inc., Aug. 2007. URL: https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf.

[Cal] University of Calgary. *Department of Algorithmic Botany - Publications*. URL: http://algorithmicbotany.org/papers/.

[Die15] Wolfram Diestel. *Arbaro*. 2015. URL: http://arbaro.sourceforge.net.

[Foua] Blender Foundation. *Blender*. URL: http://www.blender.org/.

[Foub] Blender Foundation. *Blender Python API*. URL: https://docs.blender.org/api/current/.

[Fouc] Blender Foundation. *Cycles Rendering Engine*. URL: http://www.blender.org/features/cycles/.

[FS76] R. Floyd and L. Steinburg. "An Adaptive Algorithm for Spatial Grey Scale". In: *Proceedings of the Society of Information Display*. Vol. 17. 1976, pp. 75–77.

[Hal] Andrew Hale. *Sapling Tree Addon*. URL: http://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Curve/Sapling_Tree.

[Hon71] Hisao Honda. "Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body." In: *Journal of Theoretical Biology* 31 (1971), pp. 331–338.

[Inc] Interactive Data Visualization Inc. *SpeedTree*. URL: http://www.speedtree.com.

[Juh95] Imre Juhász. "Approximating the helix with rational cubic Bézier curves". In: *Computer-Aided Design* 27.8 (Aug. 1995), pp. 587–593.

[Ken] University of Kentucky. *Introduction to Plant Identification*. URL: http://dept.ca.uky.edu/PLS220/Stemmorphbranching.pdf.

[Lin68] Aristid Lindenmayer. "Mathematical Models for Cellular Interactions in Development". In: *Journal of Theoretical Biology* 8.13 (Mar. 1968), pp. 300–315. URL: http://www.sciencedirect.com/science/article/pii/0022519368900805.

[Lon+12] Steven Longay et al. *TreeSketch*. 2012. URL: http://algorithmicbotany.org/papers/TreeSketch.SBM2012.large.pdf.

[Man17] Rafał Mantiuk. *pwcmp*. 2017. URL: https://github.com/mantiuk/pwcmp.

[Man82] Benoît Mandelbrot. *The fractal geometry of nature*. W. H. Freeman and Co., 1982.

[MD] Herpin Maxime and Jake Dube. *Modular Tree Addon*. URL: https://github.com/MaximeHerpin/modular_tree.

[MW03] David More and John White. *Cassell's Trees of Britain and Northern Europe*. London: Cassell, 2003.

[Nys10] Jørgen Nystad. "Parametric Generation of Polygonal Tree Models for Rendering on Tessellation-Enabled Hardware". MA thesis. Norwegian University of Science and Technology, June 2010. URL: http://www.diva-portal.org/smash/get/diva2:402916/FULLTEXT01.pdf.

[Opp86] Peter E. Oppenheimer. "Real Time Design and Animation of Fractal Plants and Trees". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 55–64. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15892. URL: http://doi.acm.org/10.1145/15922.15892.

[Pał+09] Wojciech Pałubicki et al. "Self-organizing tree models for image synthesis". In: *SIGGRAPH '09*. 58. New York: ACM Inc., 2009. URL: http://algorithmicbotany.org/papers/selforg.sig2009.small.pdf.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. New York: Springer-Verlag, 1990. URL: http://algorithmicbotany.org/papers/abop/abop.pdf.

[Ref+88] Phillippe de Reffye et al. "Plant Models Faithful to Botanical Structure and Development". In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: ACM, 1988, pp. 151–158. ISBN: 0-89791-275-6. DOI: 10.1145/54852.378505. URL: http://doi.acm.org/10.1145/54852.378505.

[Riš06] Aleksas Riškus. *Approximation of a cubic Bézier curve by circular arcs and vice versa*. 2006. URL: https://people-mozilla.org/~jmuizelaar/Riskus354.pdf.

[RLP07] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. "Modeling Trees with a Space Colonization Algorithm". In: *Proceedings of the Third Eurographics Conference on Natural Phenomena*. NPH'07. Prague, Czech Republic: Eurographics Association, 2007, pp. 63–70. ISBN: 978-3-905673-49-4. DOI: 10.2312/NPH/NPH07/063-070. URL: http://algorithmicbotany.org/papers/colonization.egwnp2007.pdf.

[SF01] D. Silverstein and J. Farrell. "Efficient method for paired comparison". In: *Journal of Electron Imaging* 10.2 (2001), pp. 394–398.

[Skj09] Jo Skjermo. "Generation, Rendering and Animation of Polygon Tree Models". PhD thesis. Norwegian University of Science and Technology, Apr. 2009. URL: http://www.idi.ntnu.no/research/doctor_theses/joskj.pdf.

[Smi84] Alvy Ray Smith. *Plants, fractals and formal languages*. ACM, 1984. URL: http://www.alvyray.com/Papers/CG/PlantsFractalsandFormalLanguages.pdf.

[Sta+14] O. Stava et al. "Inverse Procedural Modelling of Trees". In: *Computer Graphics Forum* (2014). ISSN: 1467-8659. DOI: 10.1111/cgf.12282. URL: http://hpcg.purdue.edu/papers/Stava14CGF.pdf.

[Trv] Tom Trval. *VegGen Addon*. URL: http://eiler2.wixsite.com/treegen.

[WP95] Jason Weber and Joseph Penn. "Creation and Rendering of Realistic Trees". In: *SIGGRAPH '95*. ACM Inc., Sept. 1995, pp. 119–128. URL: http://www.cs.duke.edu/courses/cps124/spring08/assign/07_papers/p119-weber.pdf.

[Xie+15] Ke Xie et al. "Tree Modeling with Real Tree-Parts Examples". In: *IEEE Transactions on Visualization and Computer Graphics* 22.12 (Dec. 2015), pp. 2608–2618. URL: https://www.cs.bgu.ac.il/~asharf/tree.pdf.

Appendix A

Renders of Resulting Tree Models

(Tree renders not to scale)

A.1 Parametric

Figure A.1: Quaking Aspen (autumn)

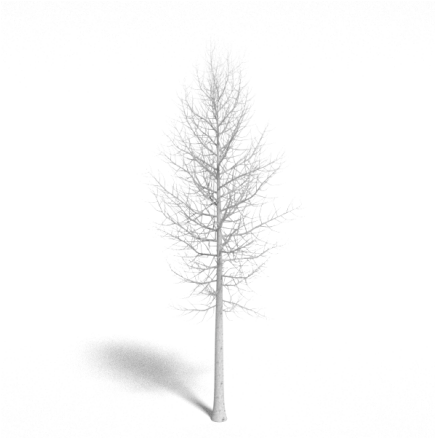


Figure A.2: Quaking Aspen (winter)

Figure A.3: Weeping Willow (summer)

Figure A.4: Weeping Willow (winter)

Figure A.5: Bamboo

Figure A.6: Palm

Figure A.7: Lombardy Poplar (summer)

Figure A.8: Lombardy Poplar (winter)

Figure A.9: Small Pine

Figure A.10: Acer

Figure A.11: Silver Birch

Figure A.12: Douglas Fir

Figure A.13: Apple (summer)

Figure A.14: Apple (spring)

Figure A.15: Black Tupelo (autumn)

Figure A.16: Black Tupelo (winter)

Figure A.17: Black Oak (summer)

Figure A.18: Black Oak (winter)

Figure A.19: European Larch

Figure A.20: Hill Cherry (spring)

Figure A.21: Balsam Fir

A.2 L-Systems

Figure A.22: Quaking Aspen (autumn)

Figure A.23: Quaking Aspen (winter)

Figure A.24: Lombardy Poplar (summer)

Figure A.25: Lombardy Poplar (winter)

Figure A.26: Small Pine

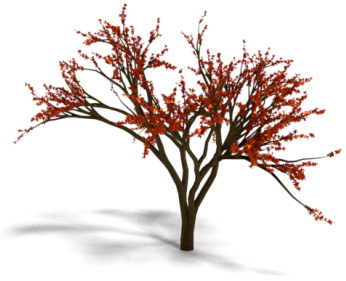


Figure A.27: Acer

Figure A.28: Palm

Appendix B

L-System Grammars

B.1 Palm

Used for Figure A.28.

```
__d_t__ = 4
__t_max__ = 350
__p_max__ = 0.93

def q_prod(sym):
    """Production rule for Q"""
    prop_off = sym.parameters["t"] / __t_max__
    if prop_off < 1:
        res = [LSymbol("!", {"w": 0.85 + 0.15 * sin(sym.parameters["t"])}),
               LSymbol("^", {"a": random() - 0.65})]
        if prop_off > __p_max__:
            d_ang = 1 / (1 - __p_max__) * (1 - prop_off) * 110 + 15
            res.extend([LSymbol("!", {"w": 0.1})])
            for ind in range(int(random() * 2 + 5)):
                r_ang = sym.parameters["t"] * 10 + ind * (random() * 50 + 40)
                e_d_ang = d_ang * (random() * 0.4 + 0.8)
                res.extend([LSymbol("/", {"a": r_ang}),
                            LSymbol("&", {"a": e_d_ang}),
                            LSymbol("["),
                            LSymbol("A"),
                            LSymbol("]"),
                            LSymbol("^", {"a": e_d_ang}),
                            LSymbol("\\", {"a": r_ang})],)
            res.append(LSymbol("F", {"l": 0.05}))
        else:
            res.append(LSymbol("F", {"l": 0.15}))
        res.append(LSymbol("Q", {"t": sym.parameters["t"] + __d_t__}))
    else:
        res = [LSymbol("!", {"w": 0}),
               LSymbol("F", {"l": 0.15})]
    return res

def a_prod(_):
    """Production rule for A"""
    res = []
    num = int(random() * 5 + 30)
    for ind in range(num):
        d_ang = (num - 1 - ind) * (80 / num)
        res.extend([LSymbol("!", {"w": 0.1 - ind * 0.1 / 15}),
                    LSymbol("F", {"l": 0.1}),
                    LSymbol("L", {"r_ang": 50 * (random() * 0.4 + 0.8),
                                  "d_ang": d_ang * (random() * 0.4 + 0.8)}),
                    LSymbol("L", {"r_ang": -50 * (random() * 0.4 + 0.8),
```

```
                                    "d_ang": d_ang * (random() * 0.4 + 0.8)}),
                    LSymbol("&", {"a": 1})])
    return res

def system():
    """initialize and iterate the system as appropriate"""
    l_sys = LSystem(axiom=[LSymbol("!", {"w": 0.2}),
                           LSymbol("/", {"a": random() * 360}),
                           LSymbol("Q", {"t": 0})],
                    rules={"Q": q_prod, "A": a_prod},
                    tropism=Vector([0, 0, -1]),
                    thickness=0.2,
                    bendiness=0,
                    leaf_shape=10,
                    leaf_scale=1,
                    leaf_scale_x=0.1,
                    leaf_bend=0)
    l_sys.iterate_n(100)
    return l_sys
```

B.2 Lombardy Poplar

Used for Figures A.24 and A.25.

```
__iterations__ = 10.0
__base_width__ = 0.7

def q_prod(sym):
    """Production rule for Q"""
    ret = []
    prev_ang = 0
    for _ in range(int(random() * 2 + 3)):
        ang = random() * 10 + 30
        ret.extend([LSymbol("/", {"a": prev_ang + 75 + random() * 10}),
                    LSymbol("&", {"a": ang}),
                    LSymbol("!", {"w": sym.parameters["w"] * 0.2}),
                    LSymbol("["),
                    LSymbol("A", {"w": sym.parameters["w"] * 0.3,
                                   "l": 1.5 * sqrt(sym.parameters["w"]) * (random() * 0.2
                                       + 0.9)}),
                    LSymbol("]"),
                    LSymbol("!", {"w": sym.parameters["w"]}),
                    LSymbol("^", {"a": ang}),
                    LSymbol("F", {"l": sym.parameters["l"]})])
    ret.append(LSymbol("Q", {"w": max(0, sym.parameters["w"] - __base_width__ / 14),
                             "l": sym.parameters["l"]}))
    return ret

def a_prod(sym):
    """Production rule for A"""
    ret = []
    n = int(random() * 5 + 22.5)
    w_d = sym.parameters["w"] / (n - 1)
    prev_rot = 0
    for ind in range(n):
        wid = sym.parameters["w"] - ind * w_d
        ang = random() * 10 + 25
        ret.extend([LSymbol("!", {"w": wid}),
                    LSymbol("F", {"l": sym.parameters["l"] / 3}),
                    LSymbol("/", {"a": prev_rot + 140}),
                    LSymbol("&", {"a": ang}),
                    LSymbol("!", {"w": wid * 0.3}),
                    LSymbol("["),
                    LSymbol("F", {"l": 0.75 * sqrt(n - ind) * sym.parameters["l"] / 3,
```

```
                                    "leaves": 25,
                                    "leaf_d_ang": 40,
                                    "leaf_r_ang": 140}),
                    LSymbol("^", {"a": 20}),
                    LSymbol("F", {"l": 0.75 * sqrt(n - ind) * sym.parameters["l"] / 3,
                                    "leaves": 25,
                                    "leaf_d_ang": 40,
                                    "leaf_r_ang": 140}),
                    LSymbol("%"),
                    LSymbol("]"),
                    LSymbol("!", {"w": wid}),
                    LSymbol("^", {"a": ang}),
                    LSymbol("\\", {"a": prev_rot + 140}),
                    LSymbol("^", {"a": 1.2})])
        prev_rot += 140
    return ret

def system():
    """initialize and iterate the system as appropriate"""
    l_sys = LSystem(axiom=[LSymbol("!", {"w": __base_width__}),
                           LSymbol("/", {"a": 45}),
                           LSymbol("Q", {"w": __base_width__, "l": 0.5})],
                    rules={"Q": q_prod, "A": a_prod},
                    tropism=Vector([0, 0, 0]),
                    thickness=0.5,
                    bendiness=0,
                    leaf_shape=0,
                    leaf_scale=0.3,
                    leaf_bend=0.7)
    l_sys.iterate_n(15)
    return l_sys
```

B.3 Quaking Aspen

Used for Figures A.22 and A.23.

```
__base_width__ = 0.3
__base_length__ = 4

def q_prod(sym):
    """Production rule for Q"""
    ret = []
    prev_ang = 0
    n = int(random() * 2 + 7)
    for ind in range(8):
        offset = 1 - (__base_width__ - sym.parameters["w"]) / __base_width__
        offset += ind / 8 / 12
        dang = 30 + 85 * offset
        if offset <= 0.7:
            b_len = 0.4 + 0.6 * offset / 0.7
        else:
            b_len = 0.4 + 0.6 * (1.0 - offset) / 0.3
        ret.extend([LSymbol("/", {"a": prev_ang + 75 + random() * 10}),
                    LSymbol("&", {"a": dang}),
                    LSymbol("!", {"w": sym.parameters["w"] * 0.08 * b_len}),
                    LSymbol("["),
                    LSymbol("F", {"l": sym.parameters["w"] / 2}),
                    LSymbol("A", {"w": 0.08 * b_len,
                                  "l": 0.6 * b_len}),
                    LSymbol("]"),
                    LSymbol("!", {"w": sym.parameters["w"]}),
                    LSymbol("^", {"a": dang}),
                    LSymbol("F", {"l": sym.parameters["l"]})])
    ret.append(LSymbol("Q", {"w": max(0, sym.parameters["w"] - __base_width__ / 11),
```

```
                                "l": sym.parameters["l"]}))
    return ret

def a_prod(sym):
    """Production rule for A"""
    ret = []
    w_d = sym.parameters["w"] / 14
    prev_rot = 0
    n = int(random() * 3 + 15.5)
    for ind in range(n):
        wid = sym.parameters["w"] - ind * w_d
        l_count = int((sqrt(n - ind) + 2) * 4 * sym.parameters["l"])
        ret.extend([LSymbol("!", {"w": wid}),
                    LSymbol("F", {"l": sym.parameters["l"] / 3}),
                    LSymbol("/", {"a": prev_rot + 140}),
                    LSymbol("&", {"a": 60}),
                    LSymbol("!", {"w": wid * 0.4}),
                    LSymbol("["),
                    LSymbol("F", {"l": sqrt(n - ind) * sym.parameters["l"] / 3,
                                  "leaves": l_count,
                                  "leaf_d_ang": 40,
                                  "leaf_r_ang": 140}),
                    LSymbol("^", {"a": random() * 30 + 30}),
                    LSymbol("F", {"l": sqrt(n - ind) * sym.parameters["l"] / 4,
                                  "leaves": l_count,
                                  "leaf_d_ang": 40,
                                  "leaf_r_ang": 140}),
                    LSymbol("%"),
                    LSymbol("]"),
                    LSymbol("!", {"w": wid}),
                    LSymbol("^", {"a": 60}),
                    LSymbol("\\", {"a": prev_rot + 140}),
                    LSymbol("+", {"a": -5 + random() * 10}),
                    LSymbol("^", {"a": -7.5 + random() * 15})])
        prev_rot += 140
    ret.append(LSymbol("F", {"l": sym.parameters["l"] / 2}))
    return ret

def system():
    """initialize and iterate the system as appropriate"""
    axiom = []
    con = int(__base_length__ / 0.1)
    s = random() * 0.2 + 0.9
    for ind in range(con):
        axiom.append(LSymbol("!", {"w": s * (__base_width__ + ((con - ind) / con) ** 6 *
            0.2)}))
        axiom.append(LSymbol("F", {"l": s * 0.1}))
    axiom.append(LSymbol("Q", {"w": s * __base_width__, "l": s * 0.1}))
    l_sys = LSystem(axiom=axiom,
                    rules={"Q": q_prod, "A": a_prod},
                    tropism=Vector([0, 0, 0.2]),
                    thickness=0.5,
                    bendiness=0,
                    leaf_shape=3,
                    leaf_scale=0.17,
                    leaf_bend=0.2)
    l_sys.iterate_n(12)
    return l_sys
```

Appendix C

Parameter Definitions

There follows a complete list of parameters for the system, with valid values and brief descriptions. [**n**] after a parameter name indicates that it has a distinct value at each level of branching.

shape

Integer 0–8, controls shape of the tree by altering the first level branch length. Predefined options conical, spherical, hemispherical, cylindrical, tapered cylindrical, flame, inverse conical, tend flame and custom respectively. Custom uses the envelope defined by the **prune_*** parameters to control the tree shape directly rather than through pruning.

g_scale

Float > 0, scale of the entire tree.

g_scale_v

Float, maximum variation in **g_scale**.

levels

Integer > 0, number of levels of branching, typically 3 or 4.

ratio

Float > 0, ratio of the stem length to radius.

ratio_power

Float, how drastically the branch radius is reduced between levels.

flare

Float, by how much the radius of the base of the trunk increases.

floor_splits

Integer ≥ 0, number of stems of the tree coming from the floor. See bamboo, Figure A.5.

base_splits

Integer, number of splits at base height on trunk, if negative then the number of splits will be randomly chosen up to a maximum of |**base_splits**|.

base_size[n]

Float ≥ 0, proportion of branch on which no child branches/leaves are spawned.

down_angle[n]

Float, controls the angle of the direction of a child branch (at level n) away from that of its parent (at level $n-1$).

down_angle_v[n]

Float, maximum variation in down angle **down_angle[n]**, if < 0 then the value of **down_angle_v[n]** is distributed along the parent stem.

rotate[n]

Float, angle about the parent branch (at level $n-1$) between each child branch. If < 0 then child branches are directed **rotate[n]** degrees away from the downward direction in their parent local basis. For fanned branches, the fan will spread a total

angle of **rotate**[**n**] and for whorled branches, each whorl will rotate by **rotate**[**n**].

rotate_v[n]

Float, maximum variation in **rotate**[**n**]. For fanned and whorled branches, each branch will vary in angle by **rotate_v**[**n**].

branches[n]

Integer > 0, the maximum number of child branches at level n on each parent stem.

length[n]

Float > 0, the length of branches at level n as a fraction of their parent branch's length.

length_v[n]

Float, maximum variation in **length**[**n**].

taper[n]

Float 0–3, controls the tapering of the radius of each branch along its length. If < 1 then the branch tapers to **taper**[**n**] of its base radius at its end, so a value 1 results in conical tapering. At **taper**[**n**] $= 2$ the radius remains uniform until the end of the stem where the branch is rounded off in a hemisphere, fractional values between 1 and 2 interpolate between conical tapering and this rounded end. Values > 2 result in periodic tapering with a maximum variation in radius equal to (**taper**[**n**] $- 2$) of the base radius - so a value of 3 results in a series of adjacent spheres.

seg_splits[n]

Float 0–2, maximum number of dichotomous branches at each segment of a branch, fractional values are distributed along the stem using a method similar to Floyd-Steinberg error diffusion.

split_angle[n]

Float, angle between dichotomous branches.

split_angle_v[n]

Float, maximum variation in **split_angle**[**n**].

curve_res[n]

Integer > 0, number of segments in each branch.

curve[n]

Float, angle by which the direction of the stem will change from start to end, rotating about the stem's local x-axis.

curve_v[n]

Float, maximum variation in **curve**[**n**]. Applied randomly at each segment.

curve_back[n]

Float, angle in the opposite direction to **curve**[**n**] that the stem will curve back from half way along, creating S shaped branches.

bend_v[n]

Float, maximum angle by which the direction of the stem may change from start to end, rotating about the stem's local y-axis. Applied randomly at each segment.

branch_dist[n]

Float ≥ 0, controls the distribution of branches along their parent stem. 0 indicates fully alternate branching, interpolating to fully opposite branching at 1. Values > 1 indicate whorled branching with $n + 1$ branches in each whorl. Fractional values result in a rounded integer number of branches in each whorl, with the difference propagated using a method similar to Floyd-Steinberg error diffusion.

radius_mod[n]

Float ≥ 0, modifies the base radius of branches, only for use in special cases such as the weeping willow (Figure A.4) where the standard radius model is not sufficient.

leaf_blos_num

Integer ≥ 0, number of leaves or blossom on each of the deepest level of branches.

leaf_shape

Integer 1–10, predefined leaf shapes corresponding to ovate, linear, cordate, maple, palmate, spiky oak, rounded oak, elliptic, rectangle and triangle respectively.

leaf_scale

Float > 0, scale of leaves

leaf_scale_x

Float > 0, x direction scale of leaves.

leaf_bend

Float 0–1, fractional amount by which leaves are reoriented to face the light (upwards and outwards).

blossom_shape

Integer 1–3, predefined blossom shapes corresponding to cherry, orange and magnolia respectively.

blossom_scale

Float > 0, scale of blossom.

blossom_rate

Float 0–1, fractional rate at which blossom occur relative to leaves.

tropism

Float Vector 3D, influence upon the growth direction of the tree in the x, y and z directions, the z element only applies to branches in the second level and above ($n \geq 2$).

prune_ratio

Float 0–1, fractional amount by which the effect of pruning is applied.

prune_width

Float > 0, width of the pruning envelope as a fraction of its height (the maximum height of the tree).

prune_width_peak

Float ≥ 0, the fractional distance from the bottom of the pruning up at which the peak width occurs.

prune_power_low

Float, the curvature of the lower section of the pruning envelope. < 1 results in a convex shape, > 1 in concave.

prune_power_high

Float, the curvature of the upper section of the pruning envelope. < 1 results in a convex shape, > 1 in concave.

Appendix D

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Procedural generation of tree models for use in computer graphics

Charlie Hewitt, Trinity Hall
20 October 2016

Project Supervisor: György Dénes
Director of Studies: Prof S. Moore
Project Overseers: Dr S. B. Holden & Dr S. H. Teufel

Introduction

Computer graphics is becoming increasingly prominent within the entertainment industry, and so there are increasing demands to provide effective ways of producing visual assets which can be used in films and games created using CGI. The field of algorithmic botany is well established, with Lindenmayer's definitive book on the subject *The Algorithmic Beauty of Plants* first published in 1990 [PL90]. In addition to Lindenmayer's fractal based methodology, a parametric approach to tree generation is also possible, as described by Weber and Penn [WP95] in 1995.

In this project I intend to build on the principle set out by Lindenmayer in order to procedurally generate realistic 3D models of trees, incorporating modern graphics techniques to enhance the output of conventional Lindenmayer systems (L-systems). In addition to this more conventional approach, I hope to develop a similar tool using Weber and Penn's parametric approach. This will enable me to compare and contrast these two approaches and assess the merit of each.

Starting point

There is a reasonable volume of existing literature in the field of procedural vegetation modelling, with extensive work from the Department of Algorithmic Botany at the University of Calgary [Cal], stemming from Lindenmayer's 1990 paper [PL90]. Mostly these focus on the development of L-system based approaches, though others have proposed entirely different methods such as the parametric system of Weber and Penn [WP95], with similar work from Nystad [Nys10] and Skjermo [Skj09].

There are a number of existing tree generation plugins for Blender which may provide useful reference and opportunity for comparison, these include *SaplingTree* [Hal] and *VegGen* [Trv]. There also exist some less directly related applications such as *TreeSketch* [Lon+12] and *SpeedTree* [Inc].

I hope to utilise Blender's python scripting functionality to create the model generation tools. This provides access to a suite of standard modelling tools, enabling simple generation of polygon meshes and Bézier/NURBS curves [Foua] as well as a UI framework which can be used to control parameters used by the program.

I will not tackle rendering as part of this project, this will be performed entirely using Blender's built in Cycles rendering engine [Fouc]. This should enable me to quickly generate high quality renders of the models produced which can then be used for visual evaluation.

Resources required

For this project I shall mainly use my own dual-core computer running MacOS Sierra. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. All code will be backed up to a github repository as well as to an external HDD on a regular basis. Project files will also be hosted in Google Drive which provides a further backup as well as version control.

I will be using the open source software package Blender for modelling and rendering [Foua].

Work to be done

The project breaks down into the following sections:

- Understanding and creating a number of applicable L-systems which generate structures resembling a variety of tree types. These basic L-system will need to be transformed into a 3D model, most likely through the use of a 'turtle' controlled by special symbols in the L-system grammar. Initially the tree structure will simply be generated as a series of cylinders.

- Extension of system to use Bézier or NURBS curves to model branches rather than basic shapes. Effort will need to be taken to ensure continuity at branching points, stochastic methods will also be utilised to introduce an element of randomness to the generation process.

- Understanding and creating a parametric generation system similar to that outlined by Weber and Penn [WP95]. This system will likely take a more direct approach to geometry creation rather than using a turtle, and again will incorporate an element of randomness.
- Creation of a series of test models and subsequent comparison of the generation times, model complexity and visual results of the two systems. Additional performance comparisons can be made by varying L-systems and parameters to each system, for example the rate at which generation time increases relative to the number of iterations of the L-system. Visual results will be assessed through a survey based on rendered images of the test models.

Success citeria

- Develop working tree generation system based on the L-systems approach.
- Develop working tree generation system based on a purely parametric approach.
- Generate a series of test models which can be used for performance comparisons of the two systems

Possible extensions

- Improve rendering quality using textures including normal/bump mapping and possibly displacement mapping.
- Leaves and roots are somewhat neglected in the above outline, some effort could be put towards an improved generation system for these aspects of the tree.
- There will likely be a number of user inputs to control the parameters of the generation, some work may be undertaken to streamline these inputs and create the optimal control arrangement for artists to design trees, balancing controllability with usability.

Timetable

Planned starting date is 21/10/2016.

- **Michaelmas weeks 3–4**
  Research L-systems and other prior work in the area to gain a full understanding of the theory underpinning the project. Familiarise myself with Blender's python scripting functionality.
- **Michaelmas weeks 5–6**
  Implementation of basic L-system generator in python and basic translation to 3D models to assess visual output of L-systems.

- **Michaelmas weeks 7–8**
  Extension of L-system generator to incorporate parametric and stochastic elements, extend modelling system to reflect this.

- **Michaelmas vacation**
  Implementation of parametric generation system. Create series of L-system grammars and parametric configurations to use as a test set.

- **Lent weeks 0–2**
  Write progress report. Refine generation systems, potential to work on some extensions.

- **Lent weeks 3–5**
  Start measurement of key metrics using the test set.

- **Lent weeks 6–8**
  Final refinements and measurements to system and start main dissertation write up, begin to consider evaluation.

- **Easter vacation:**
  Finish main dissertation write up and continue work on evaluation.

- **Easter term 0–2:**
  Finalise evaluation and complete dissertation.

- **Easter term 3:**
  Proof reading and then an early submission so as to concentrate on examination revision.