Honours Project Report

Procedural Tree Generation: Modelling Branching Structures as Subdivision Surfaces

Richard Pieterse

Supervised by:

A/Prof James Gain

	Category	Min	Max	Chosen
1	Requirement Analysis and Design	0	20	5
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	10
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	10
6	Aim Formulation and Background Work	10	15	10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total Marks		80		80

Department of Computer Science

University of Cape Town

2012

Abstract

The aim of this research project has been to improve the realism of models produced by an existing procedural tree generator. The models produced by this system are constructed as a set of intersecting generalized cylinders. Unfortunately, this approach fails to the capture smooth, natural blends where branches meet. It was postulated that these issues could be addressed by modelling the tree as a subdivision surface. To investigate this theory, a procedure was developed which first generates a parameterized mesh from an L-System graph and then converts it into subdivision surface. The most challenging aspect of the mesh generation process is modelling the points of furcation. This was solved by constructing a convex hull around ends of connected branches. This is a robust solution that stands up to any arbitrary branch configuration. This mesh is then converted into a subdivision surface through Loop subdivision. An experimental study was conducted to ascertain whether a subdivision surface is a more realistic representation of a tree than a set of generalized cylinders. The results of this study indicate that it is indeed are more realistic.

Acknowledgements

I would like to sincerely thank my supervisor, A/Prof James Gain, for his encouragement, guidance and invaluable insight. I would also like to thank my fellow project members Ryan Mazzolini and Donovan Foster for their hard work and support though both good times and bad. Finally I would like to thank our test participants for their assistance and our expert users for their wisdom and advice.

Table of Contents

Chapter 1 Introduction 1

1.1 System Overview 2

1.1.1 Mesh generator 4

1.1.2 Surface Subdivision 4

1.1.3 Texture Synthesiser 5

1.1.4 Leaf generator 5

1.2 Research Questions 5

1.3 Legal Acknowledgments 7

1.4 Thesis Outline 7

Chapter 2 Background of Subdivision Surfaces 9

2.1 Some Terminology 10

2.2 A word on NURBS 11

2.3 Significant subdivision schemes 11

2.3.1 Catmull-Clarke 12

2.3.2 Doo-Sabin 12

2.3.3 Loop 13

2.3.4 Butterfly 13

2.3.5 Extended Catmull-Clarke 13

2.3.6 Quasi-Interpolation 14

2.4 Discussion 15

Chapter 3 Related Work - Modelling Branching Structures 17

3.1 Parametric and Implicit Surfaces 18

3.2 Generating meshes for branching structures 19

Chapter 4 Design and Implementation 23

4.1 Graph Construction 25

4.1.1 Introduction 25

4.1.2 The Graph 25

4.1.3 Branch Trimming 26

4.1.4 Graph Simplification 29
4.1.5 LST file format 30
4.1.6 LST Parser and Graph construction 31
4.2 Mesh Generation 32
4.2.1 Introduction 32
4.2.2 Generating the branch segments 32
4.2.3 Joint Construction 35
4.2.4 Mesh Data Structure 43
4.2.5 Wavefront .OBJ model format 45
4.3 Parameterization 47
4.3.1 Introduction 47
4.3.2 Parameterization of the branch segments 48
4.3.3 Parameterization of the Joint Sections through Interpolation 49
4.3.4 Collapsing edges 50
4.3.5 Limitations 52
4.4 Subdivision Surfaces 53
4.4.1 Introduction 53
4.4.2 Loop Subdivision 54
4.4.3 Subdividing texture coordinates 55
4.5 Discussion 56
Chapter 5 Experimentation and Results 58
5.1 Initial Evaluation – Expert Users 58
5.1.1 Joint Mesh Construction Errors 59
5.1.2 Subdivision Surface Smoothness 60
5.1.3 Mesh Parameterization 60
5.2 Final Evaluation – Participant Study 61
5.2.1 Experimental Method 61
Experimental Procedure 66
5.2.2 Results 67
5.3 Performance Evaluation 71

5.4 System Limitations 73
5.4.1 Parameterization 73
5.4.2 Subdivision of Texture Coordinates 74
5.4.3 Graph Simplification 74
5.4.4 Bezier Curves 75
5.5 Example Models and Degenerate Cases 76
Chapter 6 Conclusion 82
6.1 Future Work 83
6.1.1 Avoiding simplification 83
6.1.2 Displacement maps 83
6.1.3 Bezier Curves 83
References 85
Appendix 90
A An overview of the TreeDraw system 90
Sketch Interface and Gesture Recognition 90
2D to 3D converter 90
L-System Generator and Compiler 91
Tree Model Generator 91
B Edge collapse algorithm 92
C Ethical Clearance 93
D Experiment Data 95
E Test Image Examples 98

List of Figures

Figure 1.1: a. Axiom (ώ) and production rules of an L-System. b. The production rules are iteratively applied to the axiom. With each iteration the string is rewritten, leading to greater complexity. c. An axial tree interpreted from a bracketed string representation [1]. 'F' indicates that a branch must be constructed, '+' and '-' indicate clockwise and anti-clockwise rotation respectively. The brackets denote depth. 2

Figure 1.2: Overview of the entire system. The arrows indicate the logical flow between components. Components in grey are part of the prior system, while coloured components are modules that have been developed by the members of this research project. The blue modules are the concern of this report. The mesh generator constructs a coarse polygon mesh from a graph produced by the L-System Compiler. This mesh is then submitted for subdivision, which produces a smooth, high resolution mesh. This mesh is rendered and displayed. 3

Figure 1.3: The TreeDraw Interface. To the left is the sketch interface and on right is the generated model. Users draw the structure of a tree that they would like to generate. When they are satisfied with the structure they then press the generate button and the sketch is submitted to the tree generation pipeline depicted in Figure 1.2. Once the model is generated it is presented for inspection in a 3D interface. 4

Figure 1.4: Artefacts present in the model produced by TreeDraw. Left: Branches do not blend naturally. Right: Gaps occur between the branches. 6

Figure 2.1: An example of the iterative process of subdivision. The model on the left is the control mesh. The model on the far right is the approximated limit surface [10] 9

Figure 2.2: The different surfaces generated by applying different schemes to a cube [7]. (Left) Quadrilateral mesh. (Right) 12

Figure 2.3: Smooth to infinitely sharp edges with Catmull-Clarke [10]. 15

Figure 2.4: Quasi-interpolation of a subdivision surface to fit the net of curves in green [13] 15

Figure 3.1: A generalized cylinder defined by the curve in red [14] 17

Figure 3.2: A model produced by TreeDraw, constructed from generalized cylinders 17

Figure 3.3: Three examples of BlobTrees [32]. These skeletal implicit surfaces consist of capsules melded together with a blend function. 18

Figure 3.4: Recursive tiling based on the SMART method. For every branch, a quadrilateral is identified and the branch is attached in place of it [37]. 20

Figure 3.5: Construction from contours. To left is the set of contour planes and to the right is the mesh that is triangulated to connect the contours [42]. 20

Figure 3.6. An example of how the Interim Core Scheme [21] projects branches onto an icosahedron and then connects them. Once the mesh is complete it is converted into Loop subdivision surface. 21

Figure 3.7: Skeleton to quad dominant mesh [43]. A graph is converted into a mesh by first creating polyhedrons at the joints and then quadrilating between them. 22

Figure 3.8: Joint constructions posed as a convex hull problem. The ends of the branches are projected onto the ends of a sphere and then a convex hull is formed from the projected vertices [25]. 22

Figure 4.1: The steps involved in constructing the model as well as the file outputs 23

Figure 4.2: An overview of the key steps involved in generating the control mesh. Figures a-c illustrate the construction of the graph, this is covered in section 4.2. Figures d-f illustrate the stages of the mesh generation procedure, which is described in section 4.3. 24

Figure 4.3: Half-sphere method of for offsets proposed by Hijazi et al. [25]..........................27

Figure 4.4: The five possible cases for calculating the minimum offset required to avoid intersection between two branches. 28

Figure 4.5: An example of a graph that has been simplified, and how the structure is undesirably affected 29

Figure 4.6: A sample LST file and the model that it represents 30

Figure 4.7: Steps involved in generating a single branch...32

Figure 4.8: Longer faces provide less control when subdivided. 33

Figure 4.9: Two faces constructed between the edge loops..34

Figure 4.10: The interim Core Scheme. The core refers to the icosahedron that can be seen in the centre of the joints. The branches are projected onto a sphere, and then a joint is triangulated using the vertices at the start of the branches, as well as select vertices from the icosahedron..35

Figure 4.11: An overview of the joint triangulation process. Every edge in every loop must form a triangle with a vertex in another loop. If a triangle is formed then the loop must be reorganized. Loops are processed in clockwise order. The dark lines indicate the edge loops and how they are updated during the triangulation process. Figure b shows an occurrence of a double edge (the same edge repeated e.g. {…AB, BA…}) which must be detected and removed. 36

Figure 4.12: two pairs of vertices become taboo with every triangle that is formed..........37

Figure 4.13: Reorganization of loops after every triangle is formed. The red and blue lines indicate the edge loops. As new triangles are formed, the edges loops are updates. A dotted line indicates a double edge that was detected and removed..................38

Figure 4.14: Left: Calculating the face normal. Right: A valid face and an invalid face based on the dot product rule...39

Figure 4.15: Figures a, b and c show how the loops are transformed to meet the criteria for joint triangulation. Figure d depicts how the triangulated joint. In figure e the vertices are transformed back to their original positions. 41

Figure 4.16: Convex Hull Construction with the Randomized Incremental Algorithm.........42

Figure 4.17: Constructing a convex hull for a set of vertex loops..42

Figure 4.18: An illustration of the connectivity between the faces, edges and vertices of the mesh data structure 44

Figure 4.19: Left: A model that has not been parameterized. Right: A parameterized model with a texture applied. 47

Figure 4.20: An example of a seamless texture that could be applied to the model. The white squares indicate the lower left corner of the texture 48

Figure 4.21: This is an illustration of how a texture is wrapped around a branch, as well as how the faces of the branch are unwrapped in texture space. For simplicity the faces are depicted as quadrilaterals. However, in the final implementation they are in fact tessellated into triangles...49

Figure 4.22: the various states of parameterization. Left: directly after the joint has been triangulated. Middle: After the edges of the joint have been collapsed many of the un-parameterized faces have been deleted. The final face is parameterized by interpolating the texture coordinates from the top left branch. 50

Figure 4.23: After collapsing the edges, the new topology will produce rounder curves on the limit surface when subdivided. The curves are illustrated in yellow.........................51

Figure 4.24: An example of a degenerate edge collapse. This occurred as a result of the large variation in the branch radii. 51

Figure 4.25: a: The structure of the model produced by the previous system. b: the control mesh produced by the mesh generation module. c: A high resolution mesh obtained through the repeated application of Loop subdivision to the control mesh................53

Figure 4.26: An illustration of the effect of linearly subdividing the texture coordinates with every subdivision step. 55

Figure 4.27: The steps involved in constructing the model as well as the file outputs.........56

Figure 4.28: Three models generated from the same LST file. The model at the top was created by TreeDraw's existing model generator. The bottom left models was created by the mesh generator described in this report. This coarse mesh is then converted into the subdivision surface seen in the bottom left 57

Figure 5.1: An example of an edge case where faces intersect..59

Figure 5.2: Ripples across the subdivision surface 60

Figure 5.3: Mesh that forms the joint had not yet been assigned texture coordinates. 60

Figure 5.4: The test interface developed to capture participant data 64

Figure 5.5: Box and Whisker plots for the three distributions obtained from the presenting silhouetted images to participants. 70

Figure 5.6: Box and Whisker plots for the three distributions obtained from the presenting shaded images to participants. 70

Figure 5.7: The performance of the joint construction algorithms in relation to the degree of branching 72

Figure 5.8: The performance of the Loop subdivision implementation in relation to the number of faces 72

Figure 5.9: Left: an example of seams introduces at the joint. Middle: Parameterized control mesh. Right: Distortion caused by subdivision near seams 73

Figure 5.10: (Left) Model produced by TreeDraw. (Right) The same model produced by Yggdrasil displaying structural change due to the graph simplification step...............74

Figure 5.11: (Left): A tree produced by the previous system modelled with Bezier curves. (Right)A tree produced by this project without Bezier curves. 75

Figure 5.12: The figures above illustrate how subdivision surfaces smooth out irregularities that occur in the control mesh 80

Figure 5.13: The images above depict a textured subdivision surface model. The model in the top image indicates that it is possible to generate curving branches despite the fact that Bezier Curves were not implemented. To achieve this, the curves must be manually drawn in the sketch interface. 81

Figure B.0.1: Collapsing 3 edges. First edge CB is collapsed, forming the merged vertex D with the combined weight of C and B. Next, Edge AD is collapsed forming merged vertex E whose weight is the combination of A B and C. 92

List of Tables

Table 1.1: The licence agreements of libraries and frameworks incorporated in the development of the system. 7

Table 2.1: A table comparing the key characteristics of the four outlined schemes. 16

Table 3.1: A useful comparison of the surface representations discussed [45]. With the exception of parameterization, subdivision surfaces combine the best properties of polygon meshes and implicit surfaces. 22

Table 4.1: The libraries and frameworks incorporated in the development of the system.25

Table 4.2: Operations that occur in the LST file format 31

Table 5.1: Top: Means and medians of the distributions. Bottom: Results of the t-tests performed on the various distributions 69

Table D.1: Motivations for data exclusion 95

Table D.2 Scores Assigned by Participants to the Shaded Renders 95

Table D.3 Scores Assigned by Participants to the Shaded Renders 96

Chapter 1

Introduction

Over the years, the rise in quality, affordability and accessibility of rendering technology has led to a significant growth in the demand for digital content. The manual creation of this content, such as modelling and texturing, can be a laborious task that requires both skill and experience. Procedural generation provides an alternative approach by automating the content creation process. This automation is achieved through an algorithmic combination of functions, rules and pseudo-random numbers. From these algorithms complex structures and systems can emerge. Rather than replacing the need for artists, procedural techniques speed up their workflow, making it easier for them to realize their intentions. They also ensure stylistic consistency across large volumes of content, which is otherwise difficult to achieve. Without the low cost and efficiency afforded by procedural generation, producing the vast amount of content expected of modern blockbusters and video games simply would not be feasible. In light of this, procedural generation has become a pressing area of research.

Tree models are particularly well suited to procedural generation. This is because their self-similar structure lends itself well to rule based construction. A prominent technique for procedurally modelling a tree is the parameterized L-System [1]. L-Systems are a formal grammar introduced by Aristid Lindenmayer in 1968 as a framework for modelling multi-cellular organisms. Over time L-systems have become a popular method of modelling plant growth [1].

An L-System grammar consists of an alphabet of characters, a set of production rules, and an initial string, called the axiom. The production rules are applied in parallel to the characters in the axiom, rewriting them with new substrings. From this simple process complex patterns emerge. In the context of procedural tree generation, the rewritten string produced by the L-system is interpreted geometrically. This is achieved by assigning operations to each of the characters in the string. These operations describe the branches of a tree through a sequence of translations and rotations. An illustration of an L-system appears in Figure 1.1.

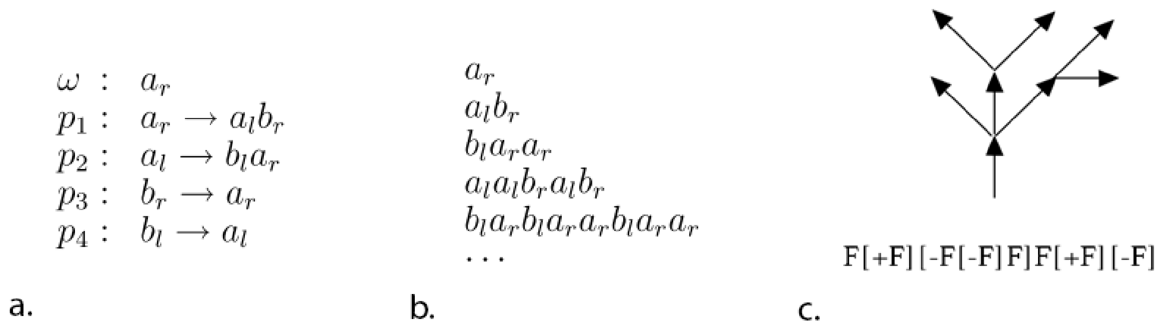


Figure 1.1: a. Axiom (ώ) and production rules of an L-System. b. The production rules are iteratively applied to the axiom. With each iteration the string is rewritten, leading to greater complexity. c. An axial tree interpreted from a bracketed string representation [1]. 'F' indicates that a branch must be constructed, '+' and '-' indicate clockwise and anti-clockwise rotation respectively. The brackets denote depth.

However, generating the structure of a tree is only half the challenge. For trees to be useful they must also look realistic and stand up to close inspection. A common approach to modelling a tree is to construct a separate mesh for each branch. The mesh of one branch is then intersected with another to create the appearance of connectivity. However, this method does not capture the smooth natural blend between branches that is observed in real trees. The aim of this research project is to address this blending problem by modelling the tree as a subdivision surface. A subdivision surface is formed when a coarse polygon mesh is recursively refined until it approximates a smooth surface. With each refinement step the faces are subdivided and vertex positions are smoothed. The end result of this process is a high-resolution polygon mesh, the surface of which appears smooth and continuous. In order for subdivision to be applied a manifold polygon mesh is required. In this context, manifold refers to a mesh where every edge has no more than two incident faces. Constructing such a mesh for an arbitrary branching structure is a non-trivial problem and the focus of this report.

1.1 System Overview

This research project builds on an existing procedural tree generation system developed during a previous research project [2, 3, 4]. This system, named TreeDraw, is a sketch-based procedural tree generator that uses parameterized L-Systems to generate trees that exhibit variation. The sketch interface allows a user to draw a two dimensional skeleton of a tree, from which an L-system is generated. A string is then derived from the L-System and geometrically interpreted as the branches of a tree. These branches are modelled in 3D as a set of generalized cylinders. A key feature of the system, which distinguishes it from other procedural tree generators, is the ability for the user to specify

variation during the sketching phase. This variation is encoded in the L-system and allows for multiple similar, yet individually unique, trees that appear to be variants of the same species. This allows a forest of unique trees to be generated from a single sketch. The system was designed to be modular, allowing the researchers to develop the different components concurrently. These components are illustrated in Figure 1.2. A more detailed overview of TreeDraw appears in Appendix A.

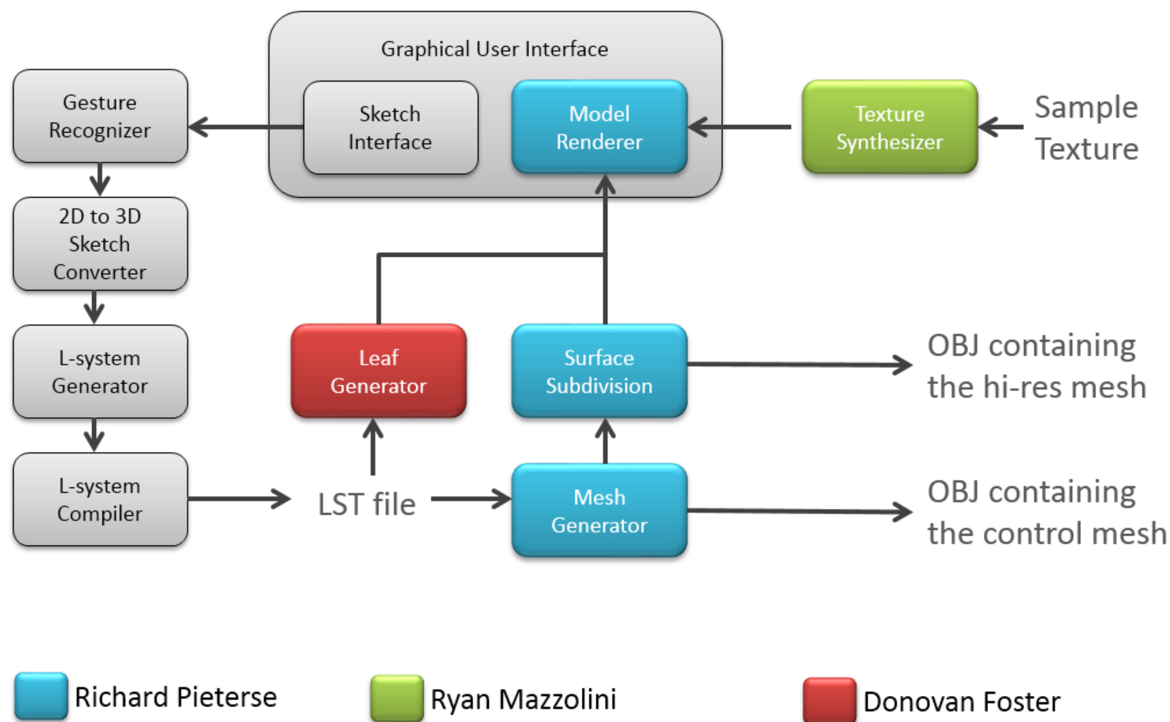


Figure 1.2: Overview of the entire system. The arrows indicate the logical flow between components. Components in grey are part of the prior system, while coloured components are modules that have been developed by the members of this research project. The blue modules are the concern of this report. The mesh generator constructs a coarse polygon mesh from a graph produced by the L-System Compiler. This mesh is then submitted for subdivision, which produces a smooth, high resolution mesh. This mesh is rendered and displayed.

The research presented in this paper is part of a three pronged attempt to improve the realism of the models produced by this system. While mesh generation and subdivision surfaces are the focus of this report, two other modules have been developed by fellow research group members. These are texture synthesis and procedural leaf generation. To distinguish between the extended system and the existing system (TreeDraw), the

extended system will be referred to as Yggdrasil for the duration of this report. Yggdrasil, named after the World Tree of Norse mythology, is essentially TreeDraw with the new modules integrated into the model generation pipeline. This makes it easier to draw comparisons between the systems in later chapters. The new modules introduced by this project are outlined below:

Figure 1.3: The TreeDraw Interface. To the left is the sketch interface and on right is the generated model. Users draw the structure of a tree that they would like to generate. When they are satisfied with the structure they then press the generate button and the sketch is submitted to the tree generation pipeline depicted in Figure 1.2. Once the model is generated it is presented for inspection in a 3D interface.

1.1.1 Mesh generator

This module constructs a manifold polygon mesh representing the entire tree using as input a graph produced by an L-system. The mesh is constructed entirely of triangles with the intention of being converted into a subdivision surface. It is parameterized so that a bark-like texture can be applied.

1.1.2 Surface Subdivision

The polygon mesh created by the mesh generator is a coarse representation of a tree and has an unnatural faceted surface. To improve this Loop subdivision is applied to the

mesh. The Loop subdivision scheme, proposed by Charles Loop [5], is tailored for the refinement of meshes consisting entirely of triangles. This makes it a very general solution. With each refinement step the mesh is tessellated and the vertex positions are smoothed. Ultimately, a smooth surface is produced that is continuous in appearance.

1.1.3 Texture Synthesiser

The simplest method of texturing a tree is to tile a seamless bark image along the length of each branch. This is the method used by TreeDraw and a similar method is implemented in the mesh generator. However, creating textures that are seamless is a difficult task. The purpose of the texture synthesizer is to produce a new yet similar seamless texture from a user provided sample. Texture synthesis affords the user the ability to create a tileable texture from an arbitrary source such as a photograph. It also increases the potential for variation in the final models as each model will have a similar but unique texture.

1.1.4 Leaf generator

The previous system concentrated entirely on generating the branching structure of the tree. This module greatly enhances the realism of the final model by placing leaves at the end of every branch. To create the leaves, users first draw an outline of a desired leaf in a sketch interface. The leaf is then generated through a process known as venation. This is involves iteratively placing nodes inside the sketched outline and growing veins towards them. The parameters that affect the vein growth and colour are selected randomly from user specified ranges. This allows multiple unique leaves to be generated from the same sketch.

1.2 Research Questions

The models produced by TreeDraw consist of a set of intersecting generalized cylinders. These generalized cylinders follow a Bezier curve that is aligned at its start with the branch from which it grows. A single seamless bark-like texture is tiled along the length of the branches. Although this representation provides a good approximation of the tree generated, it exhibits several artefacts which reduce the overall realism. The models appear particularly unnatural at the points where branches meet and gaps also occur at points where branches are not perfectly aligned.

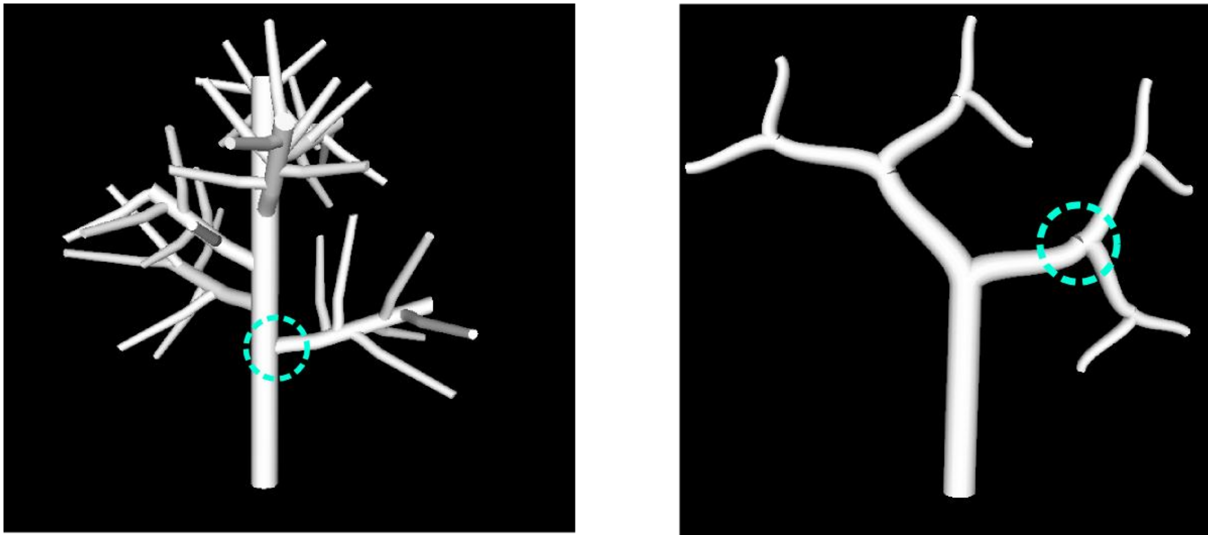


Figure 1.4: Artefacts present in the model produced by TreeDraw. Left: Branches do not blend naturally. Right: Gaps occur between the branches.

To address these issues we investigate whether subdivision surfaces can be used to more realistically model a tree. Besides subdivision surfaces, two other techniques for improving the realism of the models were investigated by fellow project members, as described earlier: texture synthesis and leaf generation through venation. From these investigations, three separate research questions arise. The first is the subject of this research report, while the last two of are the concern of group members [6, 7].

1. Can branching structure be more realistically modelled by subdivision surfaces?

2. Can texture synthesis be used to generate a realistic texture of bark that exhibits variation from a provided sample?

3. is the best method to create similar, realistic leaves from a user provided sketch, allowing for variation?

Realism is a difficult quality to quantify. To find out whether subdivision surfaces do in fact produce more realistic models than TreeDraw, an experimental study was performed with over thirty participants. In this study participants were presented with rendered images of branches from TreeDraw and Yggdrasil and asked to assign a realism score to each image. Following this they were presented with silhouetted images of models produced by both systems as well as segmented branch joints of real trees. Once again they were asked to assign a realism score to each. The range of the score was 0 to 100, where 0 corresponds to "not at all realistic" and 100 maps to "highly realistic". A detailed account of both the methodology and results can be found in chapter 5.

The performance of the system was also evaluated. In the previous research project a strict thirty second time limit was set within which the system had to interpret the user's sketch input, generate a complied L-system and produce the model. Although this is not a constraint placed on the model construction process described in this report, it is important to analyse its performance. The results of the performance evaluation can be found in chapter 5.

1.3 Legal Acknowledgments

The rapid development of this system owes credit to the following libraries. Their licenses allow them to be freely used for non-commercial purposes. This is a research project and no commercial release is intended.

Table 1.1: The licence agreements of libraries and frameworks incorporated in the development of the system.

LIBRARY	LICENCE	URL
QT	LPGL	http://qt.digia.com/
BOOST	Boost Software License	http://www.boost.org/
RAPIDXML	Boost Software License or MIT License (user decides)	http://rapidxml.sourceforge.net/
VMATH	BSD Licence	http://bartipan.net/vmath/

1.4 Thesis Outline

This report describes the research, development and evaluation of a mesh generation algorithm and an implementation of Loop subdivision. The research has been largely explorative. Although the field of subdivision surfaces is well established, the same cannot be said of generating meshes for branching structures. As such a large portion of time was spent researching and implementing different approaches to mesh generation. Two avenues of mesh generation were fully explored. These are presented in the design and implementation chapter.

The rest of this report is structured as follows. In chapter 2, the background behind subdivision surfaces is presented. Next, chapter 3 provides a brief overview of related

work pertaining to the algorithmic modelling of branching structures. In chapter 4 both the design and implementation are covered in detail. Following this, chapter 5 presents the methodology and results of expert user testing, experimental evaluation and performance tests. Chapter 6 concludes this report with a summary and a discussion of future work.

Chapter 2

Background of Subdivision Surfaces

In this chapter the background behind subdivision surfaces is presented. A subdivision surface is a polygon mesh that has been recursively refined until it approximates a smooth surface. With each refinement step the faces are subdivided and vertex positions are smoothed. It is important to understand the nuances of subdivision surfaces in order to appreciate their applicability to the modelling of trees. The chapter begins with a brief introduction to subdivision surfaces, after which, four of the most prominent subdivision surface schemes are outlined, namely: Catmull-Clarke, Doo-Sabin, Loop and Modified Butterfly. Two extensions to Catmull-Clarke are also discussed. The chapter concludes with a comparison of the schemes and a discussion of their applicability to procedurally generated trees.

In computer graphics subdivision surfaces are a hybrid of polygonal meshes and polynomial splines [8]. They are used to represent a continuous smooth surface derived from an arbitrary polygon mesh. The process begins with an input mesh consisting of vertices, edges and faces. This mesh, known as the control mesh, is then iteratively subdivided into smaller faces which conform to a piecewise linear approximation of the limit surface [9]. For rendering purposes, the mesh is usually only subdivided until each face is approximately the size of one screen pixel. Subdividing any further will have no perceivable effect on smoothness. As a technique, subdivision surfaces have become popular due to their efficiency, convenience and flexibility [10].

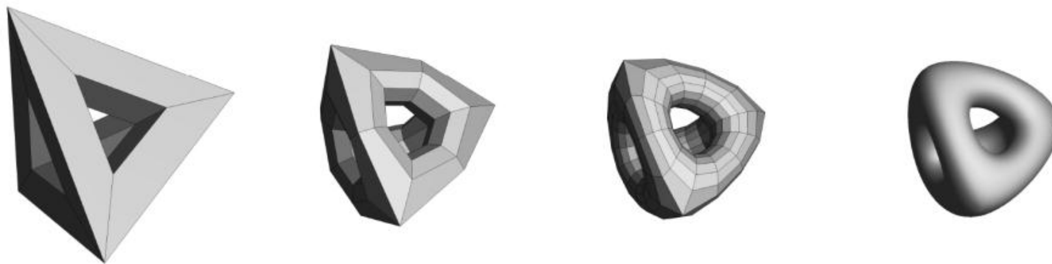


Figure 2.1: An example of the iterative process of subdivision. The model on the left is the control mesh. The model on the far right is the approximated limit surface [10]

2.1 Some Terminology

Over the last three decades a diverse assortment of subdivision schemes have been developed, each having their own distinct characteristics and ideal applications. The four most important characteristics are: face type, splitting preference, whether the scheme is approximating or interpolating, and surface continuity.

We begin with the face type. This refers to whether a scheme generates *triangle*, *quadrilateral* or *hexagonal* faces during each subdivision step [11]. This is an important consideration as a scheme that generates quadrilaterals, such as Catmull Clarke, will perform better on a mesh consisting entirely of quadrilaterals than one consisting of triangles or a mixture.

The second distinction lies in whether a scheme subdivides the mesh by *splitting the faces* or by *splitting the vertices*. In the former, the faces are simply divided into N new faces and the original vertices are retained. In the latter a new face is created in place of each previous vertex. Doo and Sabin [12] informally describe vertex splitting as 'chopping off the corners'. In the case of vertex splitting, the type of face produced depends on the valence of the vertex (the number of edges connected to it). For valences of 3, 4 or 6, vertex splitting will produce a triangle, a quadrilateral or a hexagon, respectively [11].

The third distinction is whether the subdivision scheme is *approximating* or *interpolating*. If the limit surface passes though the vertices of the control mesh then it is an interpolating scheme. If this is not the case, then the scheme is approximating. Approximating methods produce higher quality surfaces, but interpolating methods allow intuitive mesh manipulation since the control points lie on the surface [11]. Interpolation also has the added benefit of allowing algorithms to be simplified and performed in place [11]. It is worth noting that there is also an extension called quasi-interpolation [13], which attempts to combine the best of both approximation and interpolation, and is discussed in more detail in section 2.3.6.

The fourth and final distinction is in the *surface continuity* that a scheme provides. Continuity is often denoted as C^N. This means that all N derivatives of the surface are continuous. C^0 implies that all curves and surfaces are connected, an example of this is any genus-zero mesh. Almost all subdivision schemes produce C^1 continuity which indicates that there are no sharp seams. At best some schemes can boast C^2 continuity across the majority of the surface. C^1 denotes tangent plane continuity. A surface is C^2 continuous if the second derivative is continuous across all interior points [14]. C^2 continuity incorporates C^1 continuity as sub condition, and implies a greater smoothness. An ideal subdivision scheme is said to be one which produces a surface that is C^2

continuous everywhere [14]. Unfortunately such a scheme, which can be applied to any mesh, does not exist.

2.2 A word on NURBS

Prior to subdivision surfaces the dominant method for modelling smooth curved surfaces was Non-Uniform Rational B-Splines or NURBS. Unlike subdivision surfaces which can represent any arbitrary mesh topology, NURBS require that the topology be fundamentally equivalent to a sheet, a cylinder or a torus [15]. A good way to visualize topological equivalence is to consider the case of a coffee cup and a doughnut. It is possible to mould a donut into a coffee cup since both have exactly one hole punched through them (they are both genus one). As such, they are topologically equivalent. In contrast, it would not be possible to create a coffee cup from a sphere since the sphere does not have a hole in it to form the handle [10].

The implication of this limitation is that complex models, such as the human form, need to be stitched together from multiple NURBS surfaces. Often these stitched patchworks will exhibit rendering artefacts at their seams, especially upon deformation. Furthermore, in order to be stitched together the NURBS first need to be trimmed, a process which is both costly and prone to numerical error [15]. In his paper, De Rose [15] cites the significant labour taken to hide the seams of the NURBS surfaces in the face of the character *Woody* in the film *Toy Story*. Subdivision surfaces allow models to be constructed with the ease of polygon, while attaining the smoothness afforded by NURBS and other spline surfaces [16] . One benefit of using NURBS is that texture mapping is trivial due to the 2D parameterization intrinsic to them [17].

2.3 Significant subdivision schemes

In the three decades since the dawn of subdivision surfaces the field has become both vast and nuanced. For the purposes of this chapter only four of the most influential subdivision schemes are discussed. These are Catmull-Clarke, Doo-Sabin, Loop, and modified butterfly.

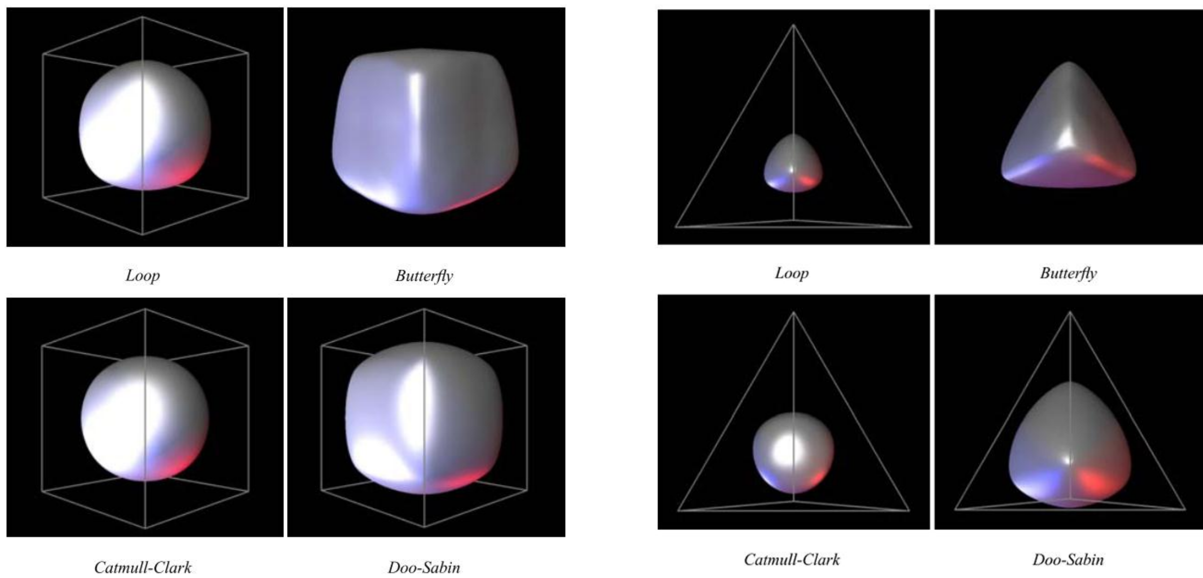


Figure 2.2: The different surfaces generated by applying different schemes to a cube [7]. (Left) Quadrilateral mesh. (Right)

2.3.1 Catmull-Clarke

The evolution of subdivision surfaces can be traced back to 1978 when Catmull and Clarke [9] first described a method for recursively generating both bi-quadratic and bi-cubic subdivision surfaces. The Catmull-Clark scheme is guaranteed to be C^2 continuous everywhere across the surface except at the extraordinary vertices where it is only C^1 continuous. An *extraordinary vertex* is defined as a vertex with an irregular valence. In these extraordinary cases the regular subdivision rules do not apply and a set of extraordinary rules must be defined. What is considered a regular valence varies between subdivision schemes and is often one of the primary motivations for a schemes existence. In the case of Catmull-Clarke an extraordinary vertex has a valence not equal to four. Similarly, an extraordinary face is one that is not made up of four edges. Although the surface displays singularities of C^1 continuity at extraordinary vertices, during subdivision the number of these vertices remains constant, while the rest of surface becomes increasingly C^2 continuous.

2.3.2 Doo-Sabin

Published in the same journal issue as Catmull-Clarke, Doo and Sabin [12] present alternative scheme that addresses the issue of extraordinary rules. Unlike Catmull-Clarke, which subdivides by splitting faces, Doo-Sabin is a vertex splitting scheme. As all vertices of any valence are treated equally, the only case where a special rule is required is at

mesh boundaries, where an edge has only one adjacent face. Doo [11] observes that this special rule can be avoided by extruding the boundary edge onto itself so that the boundary edge gets a second adjacent face with a surface area of zero. Alternative approaches to defining boundary rules have been described by Nasri [18]. The drawback of the Doo-Sabin method is that the subdivision surface is entirely C^1 continuous [12]. Although this is consistent, the surface produced by Catmull-Clarke is predominantly c^2 continuous, which is more desirable. Doo-Sabin is also an approximating scheme.

2.3.3 Loop

Later, in 1987, Loop [5] proposed a scheme similar to Catmull-Clarke in that it is both approximating and employs face-splitting to subdivide. The crucial distinction is that Loop is generalized to process regular triangular meshes. This is an important difference as the Catmull-Clarke method requires a control mesh consisting entirely of quads; a significant limitation since most meshes are in fact irregular, consisting of both of quadrilaterals and triangles. By including a pre-processing step that tessellates all quadrilaterals into triangles, Loop can effectively subdivide any arbitrary, possibly irregular, mesh. Like Catmull-Clarke, Loop also has the attribute of being C^2 continuous everywhere, except at extraordinary vertices where C^1 continuity applies [5]. An added benefit of Loop is that it supports valences of up to one hundred, even at the boundaries [11]. This makes it very flexible.

2.3.4 Butterfly

Later, in 1990, a scheme known as Butterfly was proposed by Dyn, Gregory and Levin [19]. It was also developed to process triangular faces, however, unlike Loop, which is an approximating scheme, Butterfly is an interpolating scheme. As a result Butterfly does not generate piecewise polynomial surfaces. In fact the scheme is not even guaranteed to be C^1 continuous. It does, however, benefit from being easier to implement. Later, In 1996 Zorin et al [1], proposed the Modified Butterfly scheme for handling extraordinary vertex cases. This modified method guarantees C^1 continuity across the entire surface of regular meshes. One weakness of interpolating schemes, like Modified Butterfly, is that unpredictable ripples and undulations are known to occur over the surface, especially near tight joint areas [11]. This it is particularly problematic for modelling branches

2.3.5 Extended Catmull-Clarke

Subdivision surfaces are effective, but they are also limited to representing smooth continuous surfaces. In the real world objects consist of curves, creases, sharp edges, and

everything in-between. Fortunately, in 1998 DeRose et al [15], who were at the time working at Pixar, published a paper on the subdivision techniques used in their short film, *Geri's Game*. The paper introduces the concept of infinitely sharp and semi-sharp creases. These creases can be used to represent sharp features such as fingernails and edges. To create semi-sharp creases they implement a hybrid subdivision procedure that uses a set of infinitely sharp rules for the first N iterations and then applies the regular smoothing rules to the limit [15]. The number of sharp rule iterations is determined by weighting the edges between 0 and 1. An edge with a weight of zero is perfectly smooth, while a weight of 1 will lead to an infinitely sharp crease. An example of these can be seen in Figure 3.

In the same paper De Rose et al [15] identified Catmull-Clarke as the most appropriate scheme for their purposes and showed that the propagation of UV coordinates through subdivision can be achieved implicitly. This propagation is achieved by subdividing the mesh in 5-space defined as (x, y, z, u, v) where u and v are the scalar texture coordinates for a given vertex [15]. UV coordinates are of great importance as they describe how textures map to a model's surface. De Rose et al [15] also show that textures deform more naturally over subdivided surfaces [8, 20] than over the control mesh. Unfortunately, no mention is made of how to handle texture seams.

2.3.6 Quasi-Interpolation

So far the various surface representations discussed all place a restriction on the kind of curve that can be interpolated. NURBS surfaces, as one would expect, are good for interpolating NURBS curves. Quadratic B-spline curves can be interpolated by Doo-Sabin and Catmull-Clarke produce cubic B-splines. Levin [13] addresses these restrictions by introducing the technique of *quasi-Interpolation* which is designed to interpolate nets of curves. This scheme extends Catmull-Clarke and can be categorized as a combined subdivision scheme. It begins by identifying the edges of the control mesh with segments on the curves. These edges are called c-edges and are comprised of c-vertices. The algorithm itself has 3 steps. First it performs standard Catmull-Clarke subdivision on all the vertices that are not c-vertices, called ordinary vertices. Next, the new c-vertices are calculated based on their associated curves. Finally, local corrections are performed on all the ordinary vertices that neighbour c-vertices. This three step process is iterated to the surface limit. Essentially, as the mesh is refined extra steps are taken to drag associated c-edges towards the specified parametric curves. This leads to the final subdivided mesh conforming to the net of curves. The scheme is constrained to no more than two curves intersecting at the same point. Levin [13] notes that creases can be achieved by not applying the local corrections at every step.

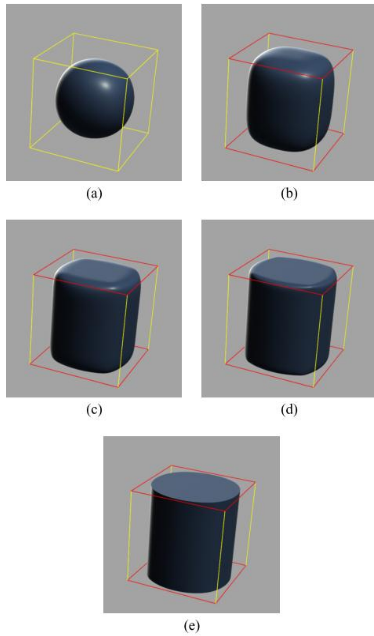


Figure 2.3: Smooth to infinitely sharp edges with Catmull-Clarke [10].

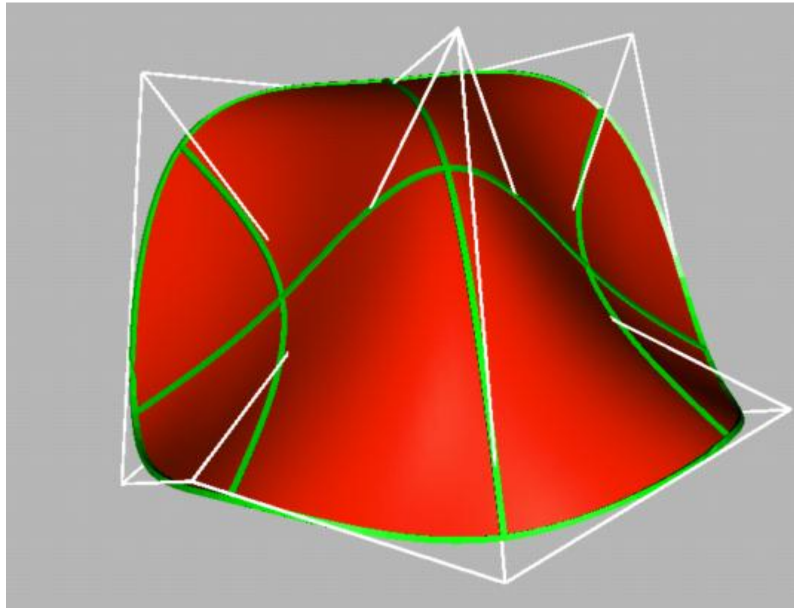


Figure 2.4: Quasi-interpolation of a subdivision surface to fit the net of curves in green [13]

2.4 Discussion

Subdivision surfaces have many advantages. They are a simple method that can describe complex surfaces with arbitrary topology. They guarantee at least C^1 continuity and texture coordinates can be propagated implicitly by performing subdivision in 5-space [15]. They certainly have applicability to the modelling of trees, however, the choice of scheme depends on the control mesh and the desired qualities of the final surface. The differences between the schemes presented is perhaps most apparent when examining the surfaces they produce rather than their characteristics. Figure 2.2 shows the different limit surfaces produced by the four schemes when applied to the same control mesh. A full comparison of scheme characteristics can be found in Table 2.1. From Figure 2.2 it is clear that Catmull-Clarke produces the most pleasing surface for a quadrilateral box. However, Catmull-Clarke does not perform as well on a tetrahedron comprised of triangle faces. From a continuity stand-point, Loop and Catmull-Clarke have the clear advantage over Doo-Sabin and Butterfly. This is because for regions away from the extraordinary vertices Loop and Catmull-Clarke generate C^2 continuous surfaces [11]. Interpolating schemes such as Modified Butterfly often exhibit undesirable ripples and undulations over the surface, especially near tight joint areas [11]. This is a

serious issue for tree meshes as the branch joints are of primary concern. That being said, a benefit of using an interpolation scheme is that it implicitly avoids branch shortening as the limit surface lies on the control mesh.

Table 2.1: A table comparing the key characteristics of the four outlined schemes.

Scheme	Approximating or interpolating	Face Type	Split	Curve generalization	Continuity	Year
Catmull-Clarke	Approximating	Quads	Face	Cubic B splines	C^2 except at extraordinary vertices	1978
Doo-Sabin	Approximating	Quads	Vertex	Quadratic Box splines	C^1	1978
Loop	Approximating	Triangles	Face	triangular Box splines	C^2 except at extraordinary vertices	1990
Modified Butterfly	Interpolating	Triangles	Face	n/a	C^1	1996

The use of subdivision surfaces to model the surfaces of trees is far from novel. Both loop and Catmull-Clarke have been popular choices [21, 14, 22]. Subdivision surfaces have also been used extensively to model blood vessels, which have a very similar branching structure to trees. When comparing subdivision schemes it is important to consider the characteristics of the mesh to which they will be applied. In the case of this project, the model consists of a single mesh consisting entirely of triangles. The branches cannot decrease in length during subdivision and the branch tips must remain sharp. The mesh has no holes, except at the tops of the branches, which are left open. As such the method must cater for boundary conditions. The primary motivation for applying subdivision to a tree mesh is to smooth and blend the joints where braches attach. The method of quasi-interpolation can be used to force the branches to conform to set of curves, affording more control over the limit surface. Subdivision surfaces can be extended for multi-resolution levels of detail. Using techniques like fitting [23] and displacement maps [24], extra geometric detail such as bark cracks and knots can be generated during subdivision. The use of such techniques greatly increases the potential fidelity of procedurally generated trees. Such functionality is available in most rendering packages. It is clear that surface subdivision is applicable to the task of modelling trees; however, before it can be applied a control mesh must be constructed. In the next chapter work relating to the generation such a mesh is reviewed.

Chapter 3

Related Work - Modelling Branching Structures

Branching structures appear frequently in nature. The vascular tree, for example, can branch into as many as seven arteries [25]. Some species of tree exhibit even higher degrees of branching. The ability to accurately model these branching structures is important in both the entertainment industry and for the visualization of scientific data. However, developing a robust and general solution to modelling branching structures is difficult. A commonly employed tool is the generalized cylinder [26, 3]. A generalized cylinder is a parametric surface defined by sweeping a planar cross section along a curve [14]. Generalized cylinders were used to great effect in modelling the trees generated by TreeDraw [3]. Unfortunately, they do have their limits. Most significantly, they do not accurately model the smooth blend between connected branches.

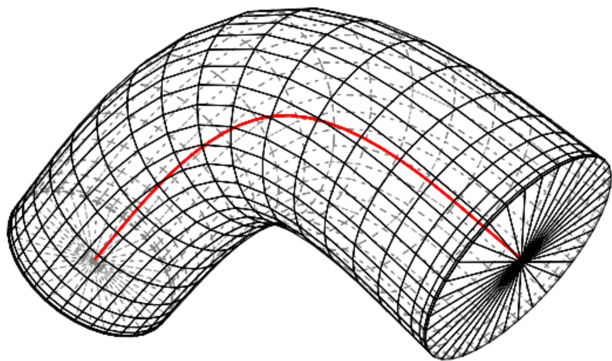


Figure 3.1: A generalized cylinder defined by the curve in red [14]

Figure 3.2: A model produced by TreeDraw, constructed from generalized cylinders

In the previous chapter, subdivision surfaces were introduced. For a subdivision surface to be created, a control mesh is first required. Creating such a mesh for a branching structure is non-trivial. In this chapter previous work that relates to the modelling of branching structures is presented. Historically, subdivision surfaces are not the only methods of modelling smooth furcating structures, and as such, alternative methods to subdivision surfaces are also discussed.

3.1 Parametric and Implicit Surfaces

One of the earliest attempts at modelling smooth bifurcations is due to Bloomenthal [27] who took a parametric approach. Branch limbs are still represented as generalized cylinders, however, the joint is modelled as a parameterized ramiform. This ramiform is a surface lofted from several splines that are defined by the bifurcation.

A pioneering force in procedural tree modelling; Bloomenthal [28] next introduced implicit surfaces as framework for representing branch junctions [29], which provides a more generalizable solution. Implicit surfaces is defined by a set of R^3points at which the value of an implicit function is equal to zero [29]. As an example [30], a unit sphere can be defined by the implicit function $f(x) = 1 - |x|$, for points $x \in R^3$, All points that lie on the surface of the sphere will evaluate to $f(x) = 0$. For points that lie inside the sphere the function will produce a positive value while points that lie outside will produce negative values. The most appropriate form of implicit surface for modelling trees is a skeletal implicit surface [28]. The function that defines the surface consists of a set of implicit functions and a set of primitives; usually spheres and cylinders. Branches are modelled by cylinders, while spheres are placed at the joints. A blend function is used to create a smooth transition between these primitives. A problem with using implicit surfaces is that bulging often occurs at the joints. Bloomenthal [31] addressed this problem with a variant of implicit surfaces known as convolution surfaces which reduce bulging [29]. Galbraith et al. [32] use convolution surfaces to great effect to model *BlobTrees,* which capture not only the smooth blends between branches but also the ridges and scars that are frequently occur in natural branch junctions. In another paper, Jin et al. [33] show that convolution surfaces can be used model general skeletal structures.

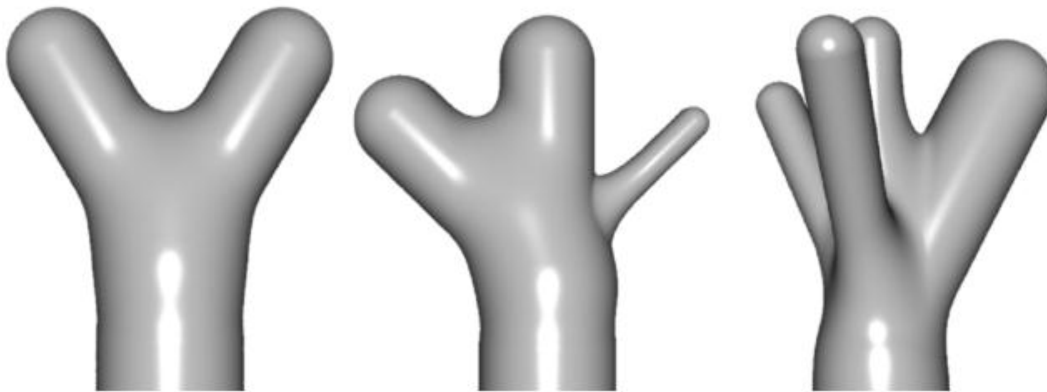


Figure 3.3: Three examples of BlobTrees [32]. These skeletal implicit surfaces consist of capsules melded together with a blend function.

Unfortunately, there are many technical difficulties associated with the use of implicit surfaces; two of the most significant are robustness and computational expense. The function that is used to blend between the primitives is often not robust, often requiring tuning on a per case basis [25]. The computational expense of implicit surfaces lies in their render complexity. They can either be ray traced or, alternatively, tessellated into a polygon mesh and rasterized. Implicit surfaces lend themselves to ray tracing since the ray intersection can be evaluated with the implicit function. Converting implicit surfaces to polygon meshes can be problematic. If the voxel grid used to sample the surfaces is too coarse then smaller branches maybe partially or entirely lost [14]. Despite these problems implicit surfaces have historically been a popular method of modelling procedurally generating trees. To their credit implicit surfaces lend themselves well to parameterizations for texture mapping [34].

3.2 Generating meshes for branching structures

As a result of the issues surrounding both parametric and implicit surfaces, subdivision surfaces have become a popular alternative. However, before a subdivision surface can be created, a coarse polygon control mesh must be constructed. The most difficult aspect of the mesh construction process is forming the joints where branches meet. Over the past decade several approaches have been proposed.

Felkel et al. [35] describe a method of constructing a mesh for blood vessels using recursive tiling. Their method is named SMART which is an acronym for Surface Models from by-Axis-and-Radius defined Tubes. SMART constructs a mesh consisting entirely of quadrilaterals. Branches are added one at a time. Every time a branch must be connected, a quadrilateral on the mesh surface is identified and replaced by the outgoing branch. Although the method generalizes to arbitrary branch configurations, only bifurcations and trifurcations are demonstrated, so the quality of surfaces with high order branching cannot be confirmed. Delingette [36] presents a similar solution, but instead of using a quadrilateral mesh, a simplex mesh is employed instead. A simplex mesh is a good choice when the mesh must undergo deformation. In their paper, Eidheim and Skjermo [37] point out that SMART sometimes produces meshes that intersect locally, requiring manual adjustment. In response to this, they propose a method, based on SMART which is fully automatic. In their method the Da Vinci rule [37, 38] is used to prevent self-intersection and ensure mesh consistency. Since the mesh consists of quads the Catmull-Clarke subdivision method may be applied to produce a smooth surface.

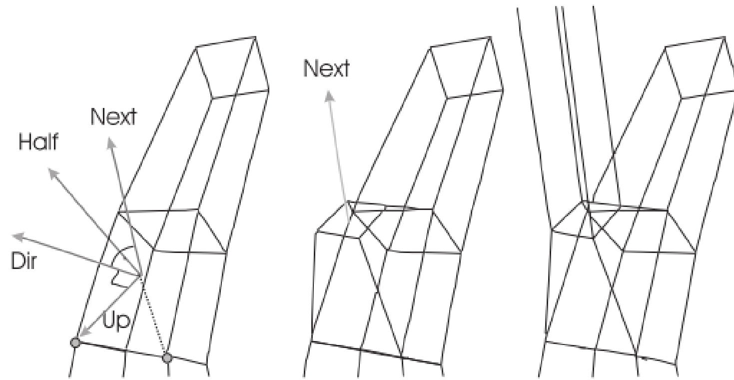


Figure 3.4: Recursive tiling based on the SMART method. For every branch, a quadrilateral is identified and the branch is attached in place of it [37].

An alternative approach is presented by Gabrielides et al. [39]. They describe a method of reconstructing branching surfaces from contours sampled at a regular interval. The motivation for such a method is to reconstruct 3D models of vessels and organs from stacks of segmented images obtained through Computed Tomography (CT) scans. Jha [40] presents a similar method of construction from contours that extends his previous work [41]. Burguet [42] also presents a method based on contours. Unfortunately, all these approaches to mesh construction from contours do not lend themselves to general branch construction. Cases where branch axes are parallel to the contour planes are problematic, since the contours planes split them down their length. Furthermore. to be applied in general the graph would first need to be sampled as a set of contours. These methods have also not been specifically demonstrated on high order branching structures.

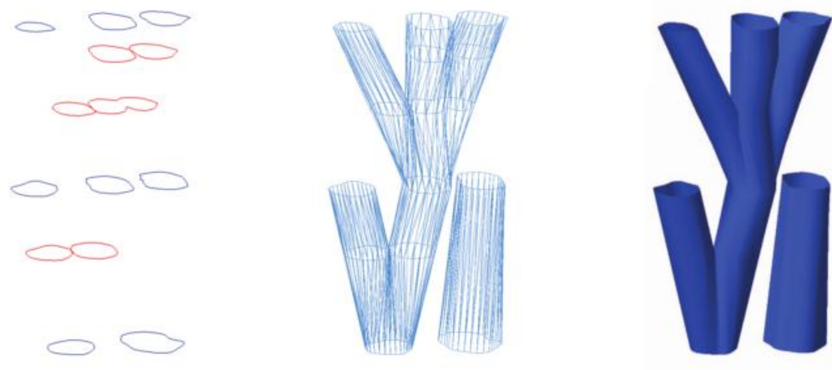


Figure 3.5: Construction from contours. To left is the set of contour planes and to the right is the mesh that is triangulated to connect the contours [42].

The research presented by MacMurchy [14] is similar to that conducted in this project. It also investigates the application of Loop subdivision to a single mesh constructed from an l-system skeleton. The fundamental distinction lies in the method employed in generating the joint sections of the mesh. MacMurchy's approach is a Junction Template method. A series of mesh templates are predefined for a set of potential branching patterns. Whenever one of these patterns occurs the appropriate template is selected and placed between the branches to form the joint. The benefit of this approach is that the templates are topologically well-formed with optimal parameterizations. The most significant shortcoming of this method is that it requires a template to be defined for every possible branch configuration and thus is not a general solution. In the paper only planar bifurcation and trifurcations are supported.

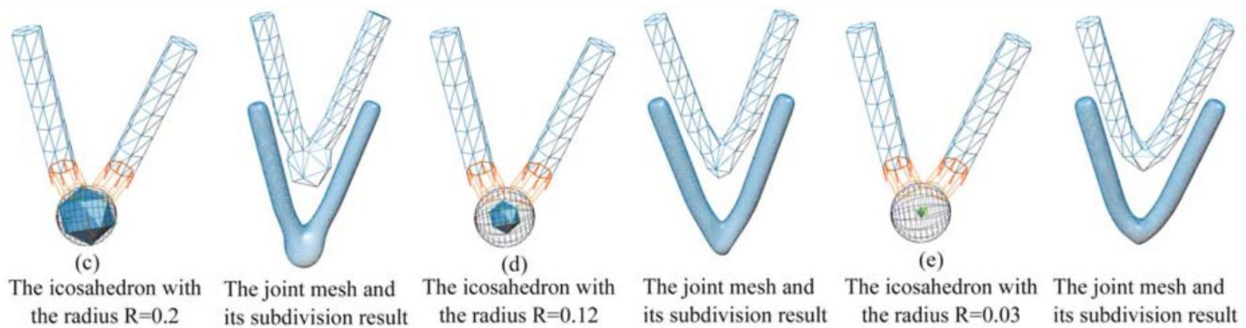


Figure 3.6. An example of how the Interim Core Scheme [21] projects branches onto an icosahedron and then connects them. Once the mesh is complete it is converted into Loop subdivision surface.

Ou and Bin [21] present a geometric approach to joint construction, which they named the Interim Core Scheme. In this scheme joint construction is treated as a triangulation problem. They make use of a convex polyhedron around which the joint is constructed. In the paper it the polyhedron is implemented as an icosahedron. To create the joint, first the ends of the branches are projected onto a virtual sphere around the icosahedron. Next, the joint mesh is triangulated from the ends of the branches, selectively using vertices from the icosahedron to smooth the transition. This approached seemed very viable and large amount of time was spent investigating and implementing it. Unfortunately, the placement and orientation are pivotal in creating a well formed joint (as is discussed in much more detail in the design and implementation chapter of this report). Hijazi [25] presents another geometric method, which is topologically driven and ensures robustness. The method reduces the problem of joint construction to a 3D convex hull problem. After realizing the issues surrounding the Interim Core Scheme, this approach was implemented instead.

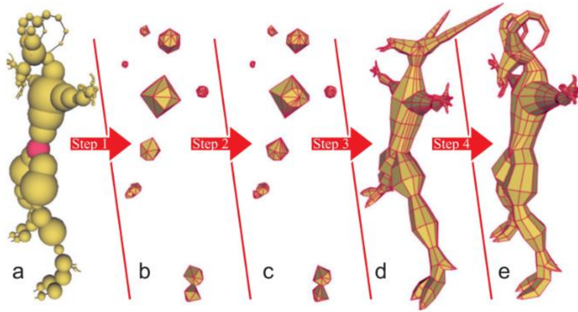


Figure 3.7: Skeleton to quad dominant mesh [43]. A graph is converted into a mesh by first creating polyhedrons at the joints and then quadrilating between them.

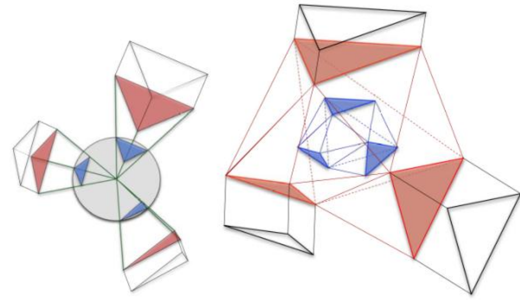


Figure 3.8: Joint constructions posed as a convex hull problem. The ends of the branches are projected onto the ends of a sphere and then a convex hull is formed from the projected vertices [25].

Most recently Bærentzen [43] presents the Skeleton to Quad-dominant Mesh (SQM) technique of constructing meshes from a graph. The method can be thought of as an inversion of the methods presented by Ou et al. [21] and Hijazi [25] who first construct the branch limbs and then triangulate joint between them. Instead SQM constructs a polyhedron at each joint quadrangulate between them to create the branches. The quad-dominance ensures that the mesh is well formed for Catmull-Clarke subdivision. The authors also mention that the method lends its self to Stripe Parameterization, which is a method directed at producing parameterizations for tube-like structures that appear seamless [44].

The properties of the surface representations discussed are compared below in Table 1. In the next chapter the design and implementation is presented; reference is made to the work of Ou and Bin [19] as well as Hijazi et al. [22].

Table 3.1: A useful comparison of the surface representations discussed [45]. With the exception of parameterization, subdivision surfaces combine the best properties of polygon meshes and implicit surfaces.

	Polygon Mesh	Implicit Surface	Parametric Surface	Subdivision Surface
Accurate	no	yes	yes	yes
Concise	no	yes	yes	yes
Intuitive Specification	no	no	yes	no
Arbitrary Topology	yes	no	no	yes
Guaranteed Continuity	no	yes	yes	yes
Parameterization	no	no	yes	no
Efficient Render	yes	no	yes	yes

Chapter 4

Design and Implementation

The goal of this research project is to develop a fully automatic procedure that converts skeletal representations of trees produced by an L-System into a single 3D mesh. Loop subdivision is then applied to the mesh to yield a smooth surface. In this chapter the design and implementation of this procedure are discussed in detail.

In context of the existing system (TreeDraw) this module is inserted towards the end of the model generation pipeline. It is placed immediately after the L-System compiler and before the model renderer. This is clearly illustrates in Figure 1.2. The module has been developed with a clean interface to afford painless integration with tree draw. The modified system, which is simply TreeDraw with this added component, will be referred to as Yggsrasil for the purposes of this report. This distinction makes drawing comparisons between the systems less convoluted.

The input to the module is an LST file, which is a standard produced by the L-System compiler. The artefacts produced are a low resolution parameterised control mesh and a smoother higher resolution mesh. The high resolution mesh is obtained by subdividing the faces and texture coordinates of the low resolution mesh using Loop subdivision. Both the low and high resolution meshes are passed on to the renderer. They can be optionally exported in Wavefront.OBJ format. This format was chosen due to its simplicity and universality. The stages of the procedure can be seen in Figure 4.1 below.

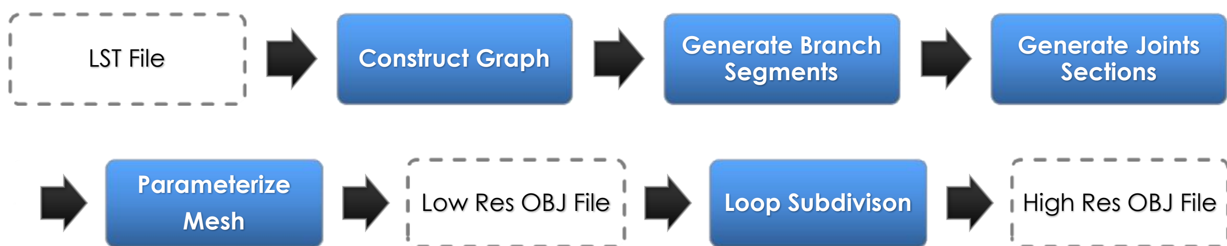


Figure 4.1: The steps involved in constructing the model as well as the file outputs

The procedure for generating the mesh is divided into four distinct stages. First, a directed acyclic graph representing the tree is constructed from an LST file. An LST file is simply a

textual description of a tree produced by an L-system. Next, the mesh is generated from the graph by constructing truncated cones for each branch and triangulating the joints between them. Following that, the mesh is parameterized to allow a bark-like texture to be applied. Finally, Loop subdivision is iteratively applied to the mesh to produce a smoother, more detailed mesh. Figure 4.1 illustrates the conceptual stages of constructing the graph and generating the control mesh.

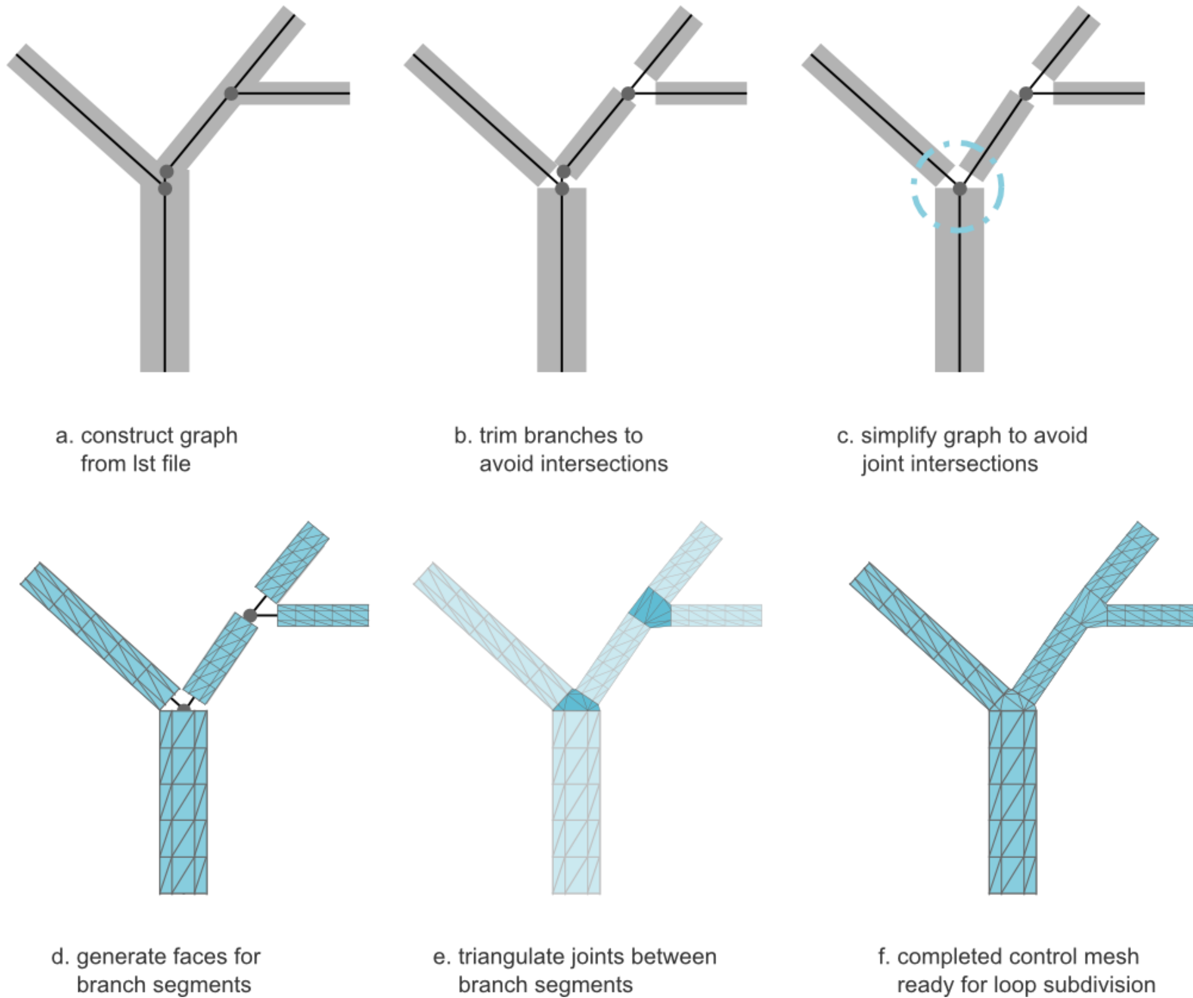


Figure 4.2: An overview of the key steps involved in generating the control mesh. Figures a-c illustrate the construction of the graph, this is covered in section 4.2. Figures d-f illustrate the stages of the mesh generation procedure, which is described in section 4.3.

TreeDraw was developed in C++, and makes extensive use of the Nokia QT framework, as well as Boost which contains an extensive supporting library of C++ algorithms. The GUI uses the widgets provided by QT. For consistency this module was built using these same tools, and introduces no other external dependencies. Both LST and OBJ file formats used as inputs and outputs are discussed in detail later in the chapter.

Table 4.1: The libraries and frameworks incorporated in the development of the system.

LIBRARY	URL
QT	http://qt.digia.com/
BOOST	http://www.boost.org/
RAPIDXML	http://rapidxml.sourceforge.net/
VMATH	http://bartipan.net/vmath/

4.1 Graph Construction

4.1.1 Introduction

The first step on the road to generating a full mesh is constructing the graph that represents the tree. For this purpose a directed acyclic graph is implemented, where each node represents a straight, uninterrupted branch. This graph is constructed from an LST file, which is produced by the L-System compiler. The LST file describes the complete skeletal structure of the tree for which the mesh will be generated. Before the graph can be passed on for mesh generation, the branches must to be trimmed so that none of them overlap. This step is shown in Figure 4.2 b. The graph must also be simplified in order to remove joints that would overlap with one another once the mesh was created. Such an overlap would involve intersecting polygons which appear unnatural, especially after subdivision. The graph construction process is detailed below.

4.1.2 The Graph

Unsurprisingly the type of graph needed is a tree. More specifically it is a directed acyclic graph, where any given vertex can have no more than one parent but any number of children. Since each vertex of the graph represents a portion of a branch, the term *branch* will refer to a vertex in the graph for the duration of this chapter.

Every branch is defined in three dimensional world-space by its starting position, rotation, start radius, end radius, and length. The rotation is relative to the unit vector in the upward direction. Every branch contains a pointer to its parent branch, as well as a list of pointers to zero or more children. The start position of a branch is the same as the end position of its parent. In contrast, there is no constraint placed on the relationship between the start and end radii. A child may potentially have a start radius that is larger than its parent's end radius, as unnatural as it may appear. Similarly, it is also possible for the branch's start radius to be smaller than its own end radius. As the mesh generation algorithm is intentionally kept as general as possible, no attempt is made to rectify these cases. Another issue present in the trees generated by TreeDraw is that completely unrelated limbs may intersect. This is most apparent in dense branching patterns. Although this appears unnatural, it's a procedural generation issue and should be accounted for in one of the earlier modules in the pipeline. Furthermore, modifications to avoid such intersections may cause the structure of the tree to become incompatible with the distributed leaves which are generated concurrently by the leaf generation module.

4.1.3 Branch Trimming

To generate a well formed mesh with no intersecting faces, it is important that none of the branches meeting at a joint overlap with one another. For a given joint, all branches initially overlap since they all share the same starting point. These overlaps are not to be confused with the overlapping limbs described in the previous section. To resolve these overlaps the starts and ends of branches must be translated along their axes by a calculated offset.

Initially, a method proposed by Hijazi et al. [25] was implemented for calculating these offsets. This method calculates the offsets using a series sphere intersection test, where the end of every branch is represented as a half-sphere. The idea is to keep shifting the spheres down the axes of the branches until none of them intersect. The distance each sphere has to be shifted is the offset for that branch. This can be seen in Figure 4.3.

Although effective, this method is both inefficient and inaccurate. Instead, a geometric approach was implemented. To find the offsets, the point at which every pair of branches intersect must be found. The offset for a branch is then the distance of its furthest intersection point with another branch. There are five possible cases for calculating the intersection between two branches. These are based on the dot product between the two branches. In this case the dot product is taken between the unit vectors that define the axes of the branches.

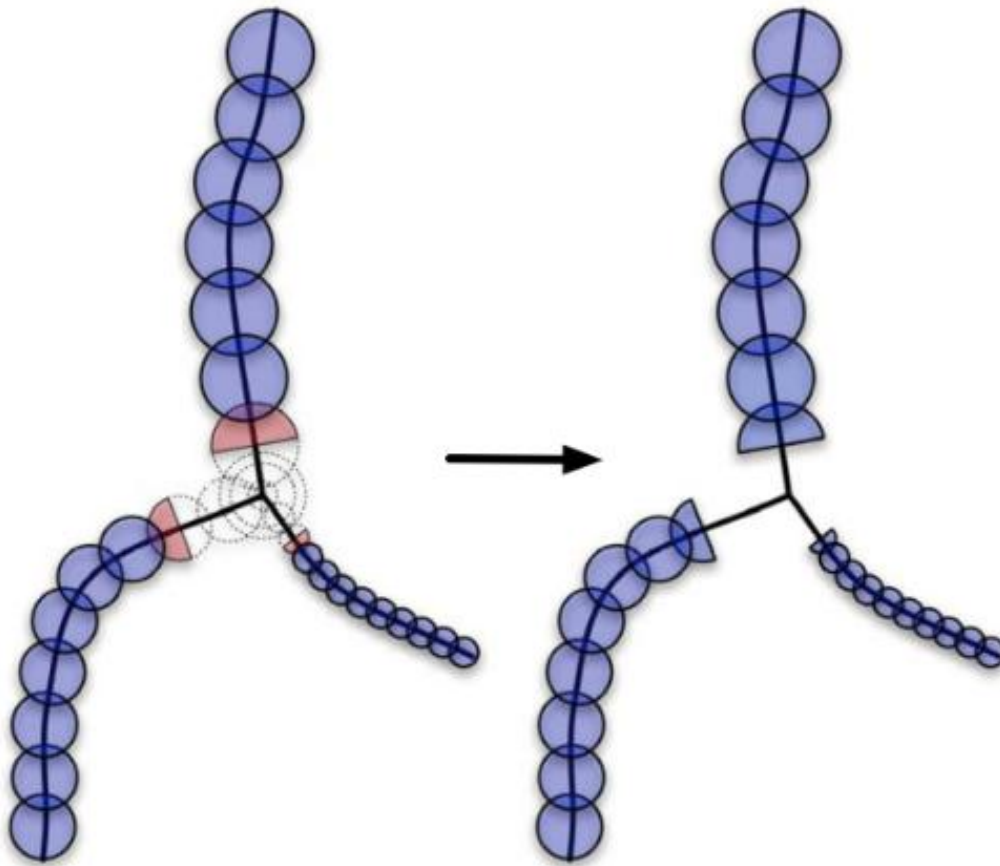


Figure 4.3: Half-sphere method of for offsets proposed by Hijazi et al. [25]

The dot product of vectors A and B can be interpreted as the length of A after it has been projected onto B. The dot product between two unit vectors has a range between -1 and 1. When the vectors form an acute angle, the dot product between them lies between 0 and 1, when they are at an obtuse angle the dot product lies between -1 and 0. The five potential cases are described below and illustrated in figure 3. The first three are trivial, while last two require some basic trigonometry. Although this method was developed independently, a similar method of branch trimming is presented by MacMurchy [14].

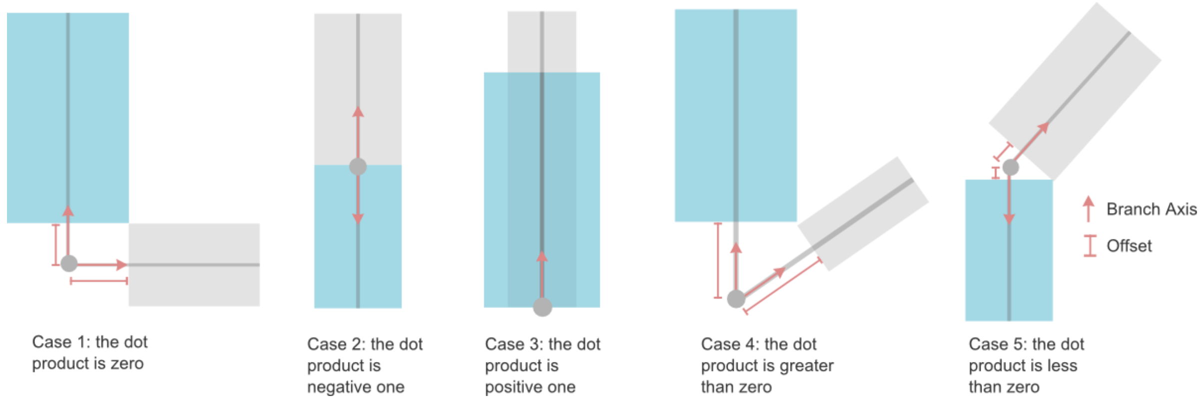


Figure 4.4: The five possible cases for calculating the minimum offset required to avoid intersection between two branches.

1 ***The dot product is 0:*** This implies that the branches are orthogonal to one another. In this case offset of A is simply equal the radius of B.

2 ***The dot product is -1:*** This implies that the Branch B is a perfect continuation of branch A, and as such the offset between them is zero.

3 ***The dot product is 1:*** This implies that the Branch B is perfectly aligned with branch A, and as such no solution exists in which the branches do not intersect. In this case the offset is simply assumed to be zero and the joint generation process is marked as failing. A graph where this case occurs is considered bad input.

4 ***The dot product lies between 0 and 1:*** The branches intersect at an acute angle to one another. In this case $offsetA$ is calculated as:

$$offsetA = \frac{RadiusA}{\tan\alpha} + \frac{RadiusB}{\sin\alpha},$$

$$\text{where } \alpha \text{ is the angle between the branch normals}$$

5 ***The dot product lies between 0 and -1.*** The branches intersect at an obtuse angle to one another. In this case $OffsetA$ is calculated as:

$$offsetA = \frac{RadiusA}{\tan(\pi - \alpha)},$$

$$\text{where } \alpha \text{ is the angle between the branch normals}$$

In both case 4 and case 5, $OffsetB$ can be calculated using Pythagoras' Theorem:

$$offsetB = \sqrt{RadiusA^2 + RadiusB^2 + OffsetA^2}$$

4.1.4 Graph Simplification

During the trimming phase it may happen that the offsets from the start and end of a branch cross over. This occurs when two joints are too close together and implies that there insufficient space to construct both. If this issue is not resolved it will lead to an intersection of the polygons that form the joint, and potentially polygons that are entirely occluded by the two joints. To account for these situations it is necessary to collapse the branch at which the crossover occurs. Collapsing a branch refers to deleting the branch and adding its children to the parent's list of children. As such the start of every child is shifted to the end position of their new parent. An example of this can be seen in Figure 4.5. The end points of the branches are never altered and in this way the branch tips of the tree remain fixed. This usually leads to branches becoming longer. The motivation for keeping the tips in the same position is that the leaf generation module [Donovan] uses the same LST file to distribute the leaves. If the positions of the tips are changed then the leaf placement will be inconsistent with the mesh. Since two joints have now merged it is necessary to perform the trimming step once again. This is because the minimum offsets needed to avoid intersection may have increased.

This method of simplification is applied to all of the branches in the graph in a 'depth first' fashion. The reason for this is that intersections that occur deeper in the graph may be resolved from addressing shallower intersections, and avoid unnecessary branch collapse operations. After the collapse has taken place the trimming procedure is applied once again to all the branches that were affected. Trimming updates the offsets so branches that were previously acceptable may now require collapse. The algorithm cycles between trimming and collapsing branches until no more collapses occur, at which point the graph is considered stable. The number of iterations depends on the topology of the graph. An extra iteration is required when a collapse causes another intersection to form.

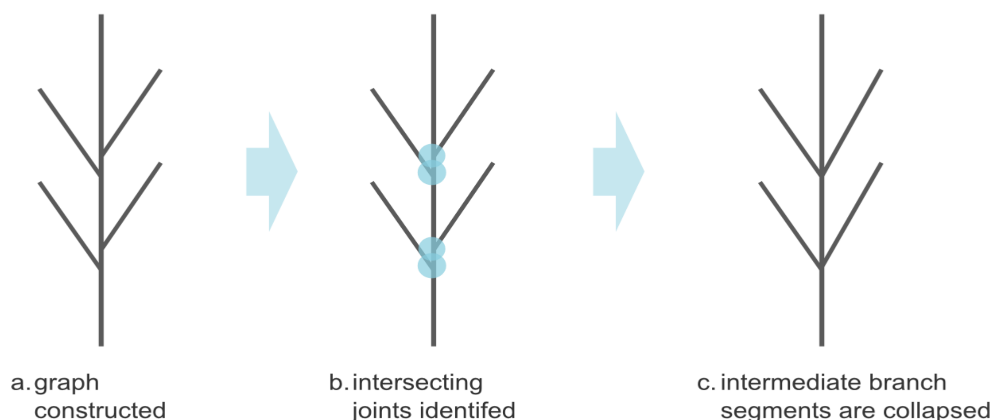
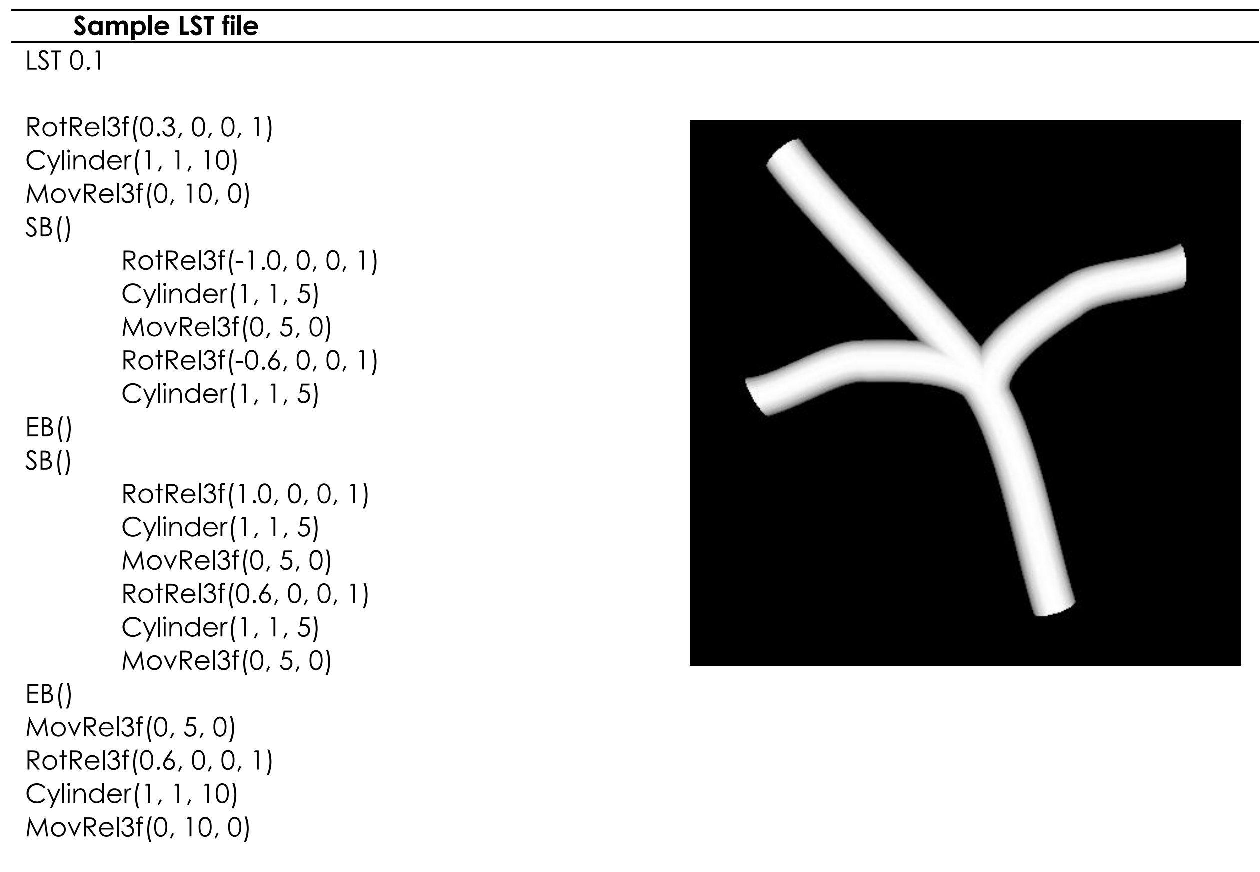


Figure 4.5: An example of a graph that has been simplified, and how the structure is undesirably affected

Although this method is effective for simple intersections, the graph can potentially lose key structural features in degenerate cases. A case that produces particularly undesirable results is depicted in Figure 4.5. In this case, a characteristic branching structure, known as alternate branching is lost. The previous system classified all trees as having either opposite or alternate structures. Losing such structure essentially changes the species of the tree.

4.1.5 LST file format

As previously mentioned in the design chapter, the input to this module is an LST file. This is a textual description of the tree produced by the L-system compiler. The LST file encodes a depth first walk of the tree through a series of pushes and pops on a state stack. The set of operations used is a subset of those defined for the lpfg plant modelling system [3]. A simple example of a LST file is depicted in Figure 4.6. The LST format begins with a single line containing the word "LST" followed by the format version number. The remainder of the file lists the operations and their parameters. Each operation is listed on a new line. Empty lines are ignored and have no significance. The operations that occur correspond approximately to the OpenGL render context where state matrices containing 3D transformations are pushed and popped off of a stack. These operations are described in Table 1:



Sample LST file

```
LST 0.1

RotRel3f(0.3, 0, 0, 1)
Cylinder(1, 1, 10)
MovRel3f(0, 10, 0)
SB()
        RotRel3f(-1.0, 0, 0, 1)
        Cylinder(1, 1, 5)
        MovRel3f(0, 5, 0)
        RotRel3f(-0.6, 0, 0, 1)
        Cylinder(1, 1, 5)
EB()
SB()
        RotRel3f(1.0, 0, 0, 1)
        Cylinder(1, 1, 5)
        MovRel3f(0, 5, 0)
        RotRel3f(0.6, 0, 0, 1)
        Cylinder(1, 1, 5)
        MovRel3f(0, 5, 0)
EB()
MovRel3f(0, 5, 0)
RotRel3f(0.6, 0, 0, 1)
Cylinder(1, 1, 10)
MovRel3f(0, 10, 0)
```

Figure 4.6: A sample LST file and the model that it represents

Table 4.2: Operations that occur in the LST file format

Operation	Action
SB()	Push the current location onto the stack
EB():	Set the current location to the location on the top of the stack and then pop that location off the stack.
MovRel3f(x, y, z)	Translate the current position by a 3D vector with components x, y, z.
RotRel3f(angle, x, y, z)	Rotate the current orientation around the axis (x, y, z) provided by the stated angle.
Cylinder(startRadius, endRadius, length)	Create branch (truncated cone) beginning at current position and continuing along the vector defined by the unit up vector scaled by the length and rotated by the current rotation.

4.1.6 LST Parser and Graph construction

The structure encoded in the LST file is most easily extracted using three stacks containing positions, orientations, and branches that have already been constructed. A 3D vector is used to store position, a standard 3x3 affine rotation matrix represents the rotation, and the branches are stored as a C++ *struct*. Initially the current position is set as the origin, the orientation is an identity matrix and the current branch is set to null.

The parser processes the operations from top to bottom. Whenever the cylinder() operation is encountered a new branch must be constructed. The branch is defined by its start radius, end radius and length. Its starting position and orientation are the values at the top of the relevant stacks. Its parent is set to the branch at the top of the branch stack. The RotRel3f() operation describes an axis-angle rotation that must be composed with the current rotation. The MoveRel3F() operation provides a 3D vector by which the current position must be translated. Whenever the SB() command is encountered the current position, rotation and branch are pushed onto their respective stacks. EB() indicates the opposite, the top values of the stacks are popped off and stored as the current position, rotation and branch.

As the graph is constructed the branches are stored in a C++ *std::vector* in depth first order. This is convenient as all operations that are applied to the graph, such as simplification, take place in this order. The final output of the parser is a directed acyclic graph, an example of which can be seen in more detail in Figure 4.2 c.

4.2 Mesh Generation

4.2.1 Introduction

In this section the algorithms that convert the graph into a mesh are presented. This process has two distinct stages. First, all the branch segments between the joints are created. Once this has taken place the joint sections of the model are generated by triangulating between the end vertices of the branch sections. An illustration of this can be seen in in Figure 4.2 (d-f). The mesh generated by this process is continuous except at the base and branch tips, which are left open.

4.2.2 Generating the branch segments

The first step towards generating a complete mesh is to construct the branch sections that lie between the joints. Each branch is modelled as a truncated cone; this can be thought of as a cylinder whose end radius is shorter than is start radius. The process of constructing a branch involves three steps. First, the required number of segments is calculated. Next, all the vertices are created as sets of ordered loops equal in total to the number of segments plus one. Finally, faces are generated between the corresponding vertices of the loops. These three steps are illustrated in the Figure 4.7 below.

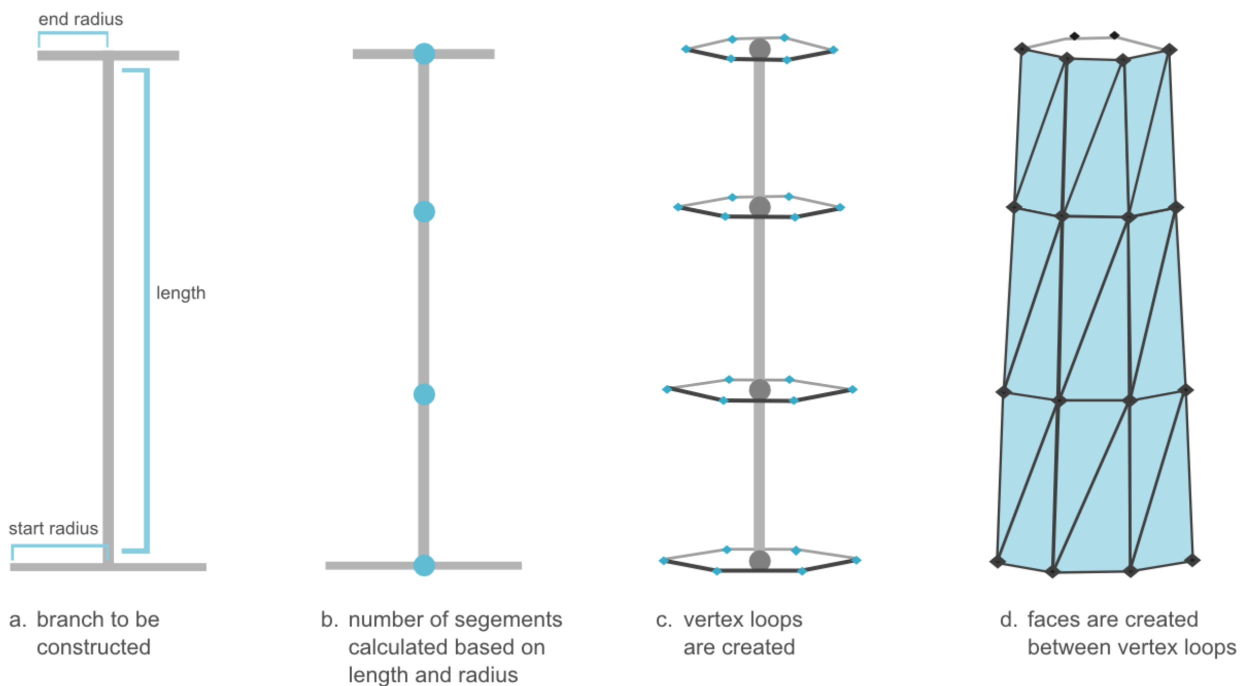


Figure 4.7: Steps involved in generating a single branch.

The number of vertices used to construct a loop can be any integer larger than three, so long as all loops in a branch are created with the same integer. The number of vertices in the figures varies between four and eight; however the loops in the final implementation are constructed with only four vertices. This number is selected with two considerations in mind. First, it is desirable to keep the mesh as coarse as possible to improve performance and save memory, extra resolution can always be obtained through subdivision. Second, the subdivision process tends to shrink the surface away from the control mesh. This is because an approximating subdivision scheme was implemented rather than an interpolating one. With a coarser control mesh, more shrinkage occurs. It was decided by inspecting the final surfaces that four vertices per loop provided the best trade-off between coarseness and shrinkage.

Calculating the Number of Segments

Each branch could be modelled by one long segment using only two vertex loops, one at the start and one at the end. At face value this approach seems favourable as it leads to a lower polygon count. However, the long thin faces that are produced do not provide enough control for subdivision. This is because during the subdivision process the original vertices are pulled towards their neighbouring vertices. The further away a neighbouring vertex, the greater its effect on the vertex being pulled. This effect is illustrated in Figure 4.8. In the context of the final subdivided model, this leads to an exaggerated blend between the branches. To account for this, the branch section is tessellated based on the relationship between its length and average circumference. As a result, longer branches with a lower circumference are more finely tessellated than shorter branches with larger circumferences. The following formula is used to calculate the optimal number of segments.

$$AverageRadius = \frac{StartRadius + EndRadius}{2}$$

$$N = \left\lceil \frac{BranchLength}{AverageRadius * \pi} \right\rceil$$

Figure 4.8: Longer faces provide less control when subdivided.

Constructing the Vertex Loops

Now that the number of segments, *N*, is known, *N* + 1 vertex loops are constructed. A vertex loop is a set of vertices that lie in the same plane. This plane is always orthogonal to the branch axis. The branch axis is simply the vector that runs from the start of the branch to the end. The vertices in a loop are connected sequentially. All vertices in a loop are also placed equidistant from the branch axis. The loops are constructed by linearly interpolating between the start and end position of the branch. The start and end radii are also interpolated. As the loops are created they are placed in a *loop list,* which is ordered from the start of the branch to the end.

Constructing the Faces

Once all the vertex loops are initialized the faces can finally be constructed. These faces can be either quadrilaterals or triangles. Triangles were chosen since the mesh is generated with the intention of applying Loop subdivision. The algorithm for constructing the faces between the loops is outlined below in Algorithm 1. This approach produces vertex valances of six which is the regular valence for loop subdivision. Regular and extraordinary vertices are discussed in the background chapter on subdivision surfaces. A subdivision scheme produces its highest level surface continuity at regular vertices, in the case of Loop subdivision this is C^2 continuity. Before exiting the routine, the first and last vertex loops must be stored in the branch node as they will be used to construct the joint.

Algorithm 1: Faces

```
Input: loop_list
for i ← 0 to n -1
    loopA ← loop_list[i]
    loopB ← loop_list[i+1]
    len ← number_of_sides
    for j ←  0 to len - 1
            Vertex A ← loopA[j]
            Vertex B ← loopA[(j+1)%len]
            Vertex C ← loopB[j]
            Vertex D ← loopB[(j+1)%len]


            Create Face ABC
            Create Face BDC
    end for
end for
```

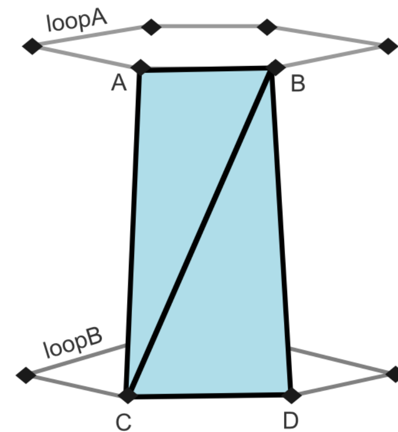


Figure 4.9: Two faces constructed between the edge loops

4.2.3 Joint Construction

Joint construction refers to generating the sections of the mesh which connect the branches. This step can be seen in Figure 4.2e. Most of the research performed during this project was directed towards developing a robust joint construction algorithm. Although many papers exist on constructing such joints, it is difficult to conclude from the results provided which solution is optimal. Another factor is how concisely the algorithms are presented. Many methods researched are outlined in chapter 3. In the end two methods were implemented.

The first was the Interim Cores Scheme proposed by Ou and Bin [21]. The results presented in their paper are appealing and the algorithm is described explicitly. It is a complex algorithm to implement and consumed the bulk of development time. Only after it was completed was it realized the algorithm is essentially a solution to 3D convex hull construction, only it approached as a special case of point cloud triangulation. The rules presented for triangulation produce reasonable results, however, cannot guarantee robustness for every possible edge case. Later, a more recent paper was discovered, in which joint construction was in fact solved using convex hulls. This lead to the previous algorithm being discarded, and a convex hull algorithm to be implemented in its stead. The end result is a simpler and more robust final solution. Due to significant amount of development time dedicated to the Interim Core Scheme, the algorithm is still included in this chapter. For interest's sake, the performance of both algorithms are analysed in chapter 5.

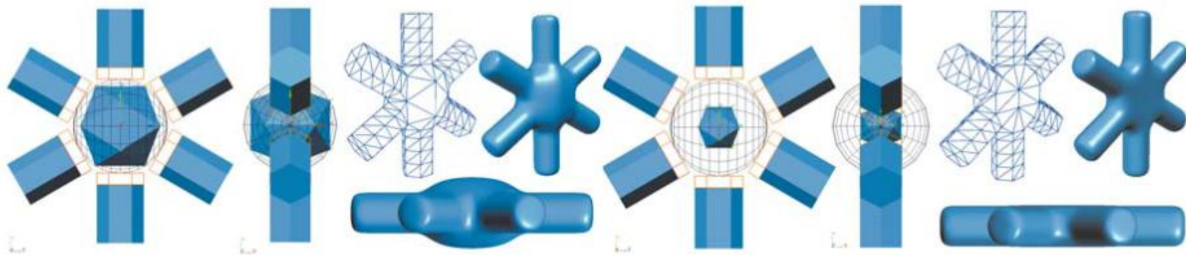


Figure 4.10: The interim Core Scheme. The core refers to the icosahedron that can be seen in the centre of the joints. The branches are projected onto a sphere, and then a joint is triangulated using the vertices at the start of the branches, as well as select vertices from the icosahedron.

4.2.3.1 The Interim Core Scheme (ICS)

Initially the entire ICS algorithm was implemented as described by Bin and Ou [21]. The core alluded to in the title is an icosahedron onto which the ends of the branches are projected. The joint mesh is the triangulated between the vertices at the start of each branch. A few of the vertices in the icosahedron are also selected to provide extra intermediate structure. Unfortunately the scheme was never designed to be fully automatic. If the icosahedron is not optimally oriented by hand then the results produced are often undesirable.

Furthermore, in many cases the structure added by the core is redundant. This can be seen in the mesh on the far right of Figure 4.10f; the two extra vertices in the middle of the joint do not add any useful structure. What is worse is that if icosahedron was orientated differently it could add undesirable structure, creating a bulge or an indentation. Due to these issues the actual core was discarded, though the triangulation rules presented by Ou and Bin [21] are still implemented.

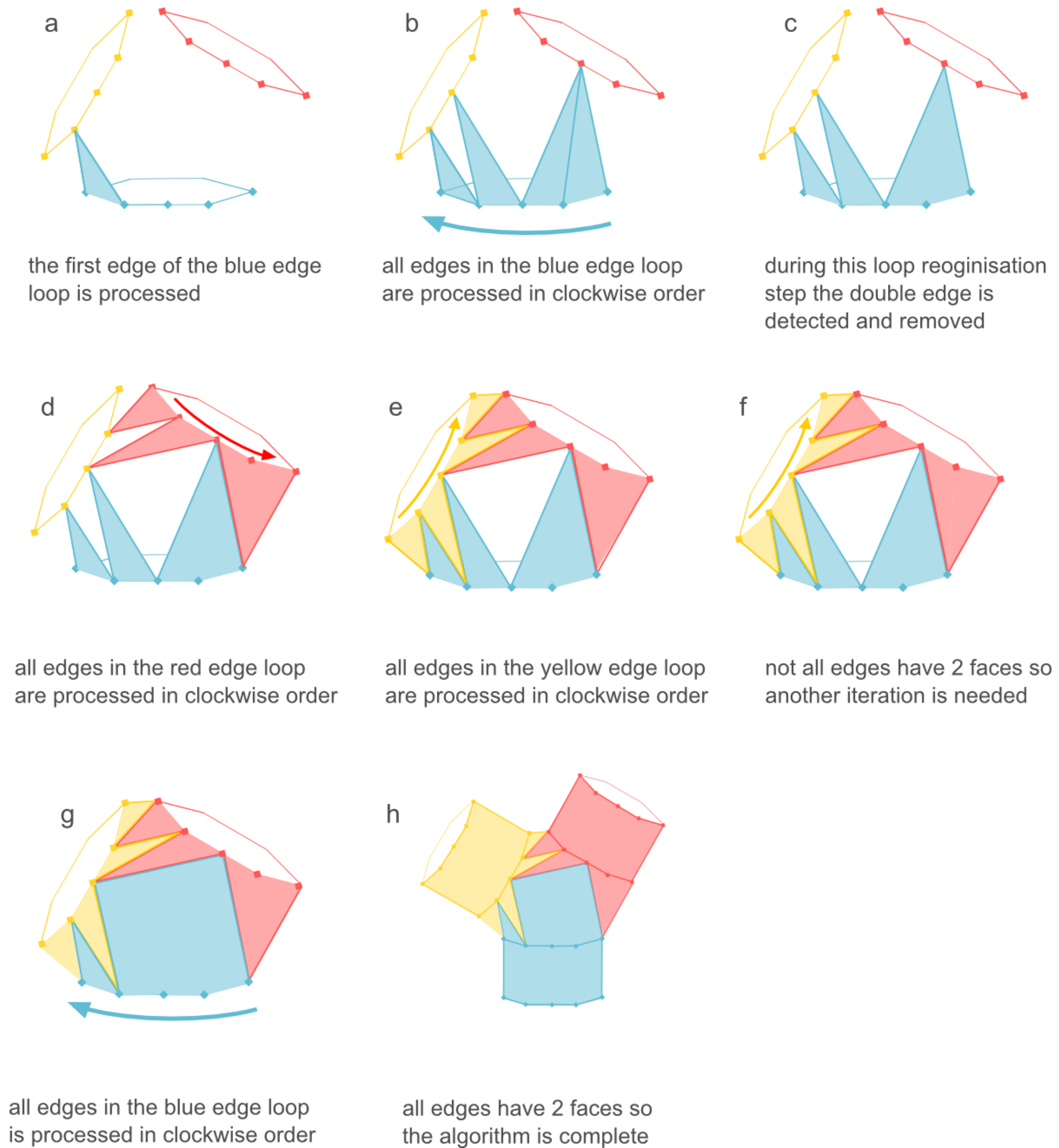


Figure 4.11: An overview of the joint triangulation process. Every edge in every loop must form a triangle with a vertex in another loop. If a triangle is formed then the loop must be reorganized. Loops are processed in clockwise order. The dark lines indicate the edge loops and how they are updated during the triangulation process. Figure b shows an occurrence of a double edge (the same edge repeated e.g. {…AB, BA…}) which must be detected and removed.

The input to the algorithm is a list of vertex loops. These are the vertex loops that define start of each branch. From these vertex loops, a list of corresponding edge loops is created. Each edge is made of two sequential vertices from a vertex loop. All of the edges in these loops have only one adjacent face and are ordered by their connectivity.

The algorithm processes each edge one at a time, loop by loop. For every edge the algorithm selects the best vertex with which to form triangular polygon. The set of vertices with which an edge can form a valid face are those that are visible to both of the vertices that compose the edge. Two vertices are said to be visible to one another if they can form a valid edge. A valid edge is one that will lead to a convex joint with no intersecting polygons or holes. Every vertex has a corresponding list of visible vertices. A notion closely related to the concept of visible vertices is what Ou and Bin describe as *taboo* vertices. Two vertices become taboo to one another if the edge formed between them intersects a face. When two vertices become taboo they are removed from each other's visible vertex list. Every time a new face is added two pairs of vertices become taboo. This is illustrated in Figure 4.12. Initially the visible list of a vertex contains every other vertex. The first vertices to become taboo are the ones that share the same initial edge loop, as these can never form valid triangles. As the algorithm progresses more and more vertices become taboo, and the visible lists become gradually smaller.

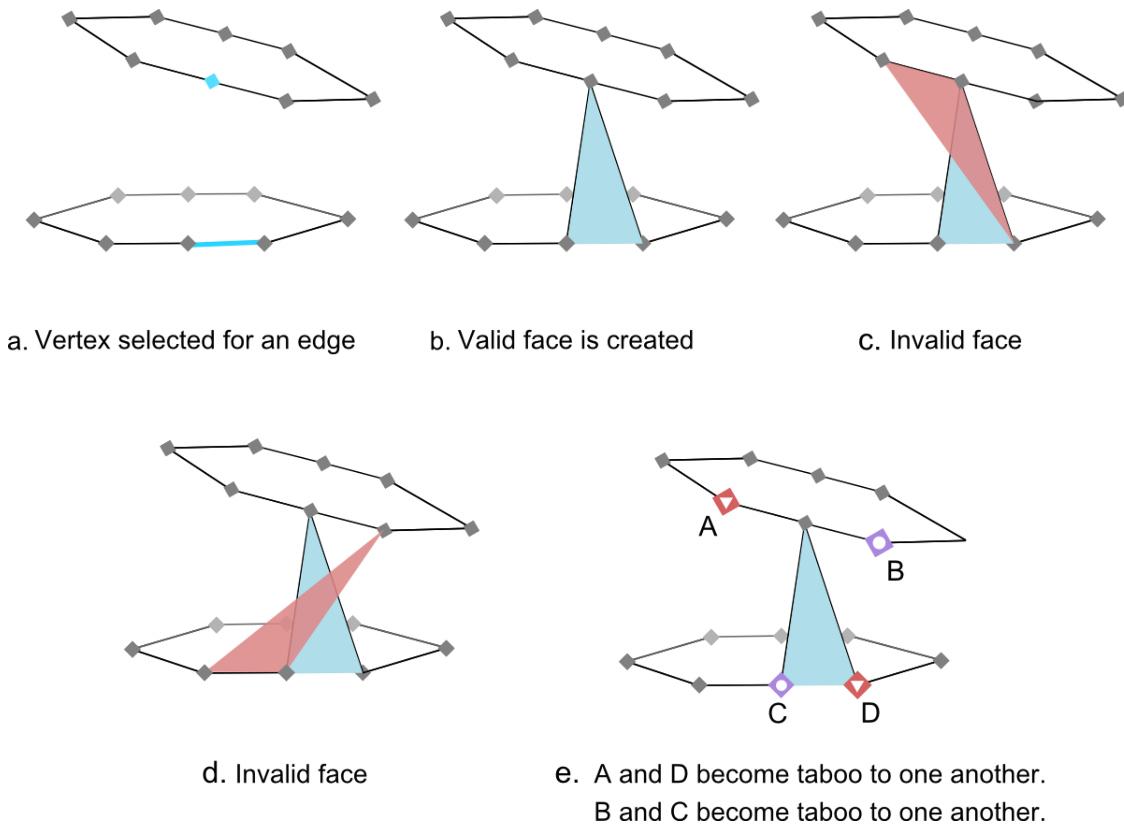


Figure 4.12: two pairs of vertices become taboo with every triangle that is formed.

In order to work out which vertices must become taboo when a face is formed, the edge loops must always be ordered clockwise and only contain open edge. To enforce this, every time a face is added the loop must be reorganised. Sometimes a double edge will form in the loop, these must be removed. As an example consider the following loop: {…AB, BG, GB, BC…}. I this loop edged BG and GB are actually the same edge. Consequently, they form a double edge. To remedy this, the loop must be reorganised to be {… AB, BC …}. Figure 4.13 illustrates how the loops are reorganised after every face is created.

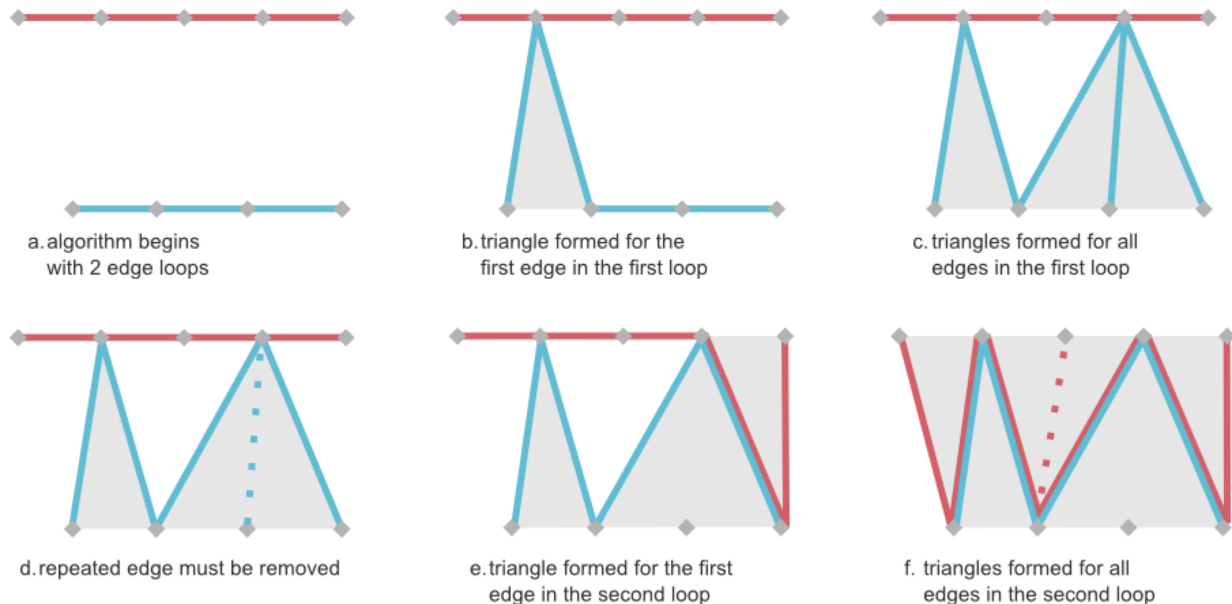


Figure 4.13: Reorganization of loops after every triangle is formed. The red and blue lines indicate the edge loops. As new triangles are formed, the edges loops are updates. A dotted line indicates a double edge that was detected and removed.

Once all the loops have been processed, the algorithm checks if the joint is complete. Completion occurs when every edge in the joint mesh is incident on two faces. An edge with only one face implies that there is still a hole in the joint. To check for completion every edge in the joint mesh is inspected. If the joint is found to be incomplete then a new iteration of the algorithm begins, this time with the reorganised loops. An overview of this algorithm appears in Algorithm 2.

The set of vertices with which a face can be formed are the vertices that are visible to both of the edge's vertices (the intersection of their visible lists). The best vertex selected from this list must meet two criteria.

1) ***Largest Opposite Angle:*** The primary rule for selecting a vertex to pair with an edge is a simple geometric principle. From the set of shared visible vertices, always choose the vertex that forms the largest opposite angle with the edge. This produces good results, however there are occasions when this is not the best choice.

2) ***Dot product test to avoid inward facing triangles:*** Although the principle of selecting the largest opposite angle is good heuristic, situations will arise where the chosen vertex that meets this criteria undesirable. This occurs because the principle of selecting the largest angle does not take any topological information into account. To avoid forming faces that are concave with respect to the joint centre, and therefore in contravention of a convex hull, a dot product test must be performed between the centre normal and the normal of the potential face. The centre of the joint is defined as the midpoint of all the vertices. This is described in more detail in a subsequent section. The centre normal is defined as the unit vector from the vertex being examined, to the centre. The face normal is calculated by normalizing the cross product between the potential edges. It is important that the edges are crossed in the correct order to ensure that the normal faces outward. If the dot product is less than or equal to zero then the face formed will not meet the convex requirements

$$CenterNormal \ = \frac{geometricCenter \ - \ vertex}{\|geometricCenter \ - \ vertex\|}$$

$$FaceNormal \ = \frac{EA \ \times \ EB}{\|EA \ \times \ EB\|}$$

$$dot = \ CenterNormal \odot FaceNormal$$

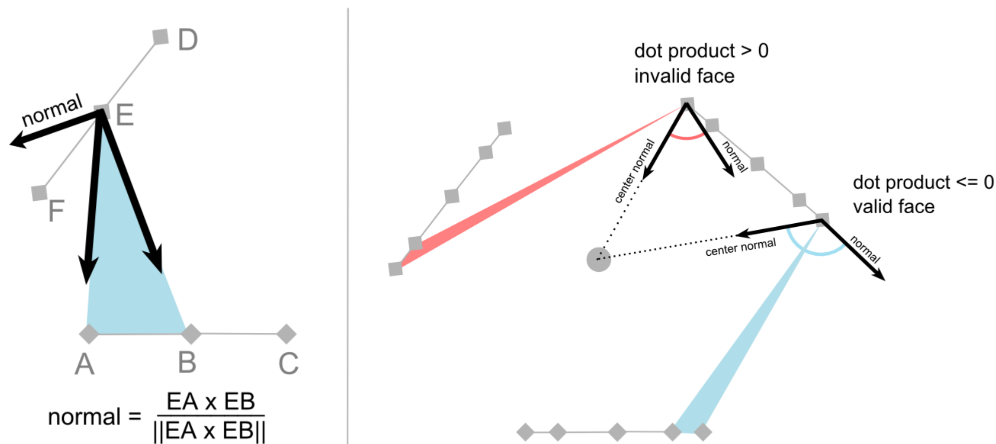


Figure 4.14: Left: Calculating the face normal. Right: A valid face and an invalid face based on the dot product rule.

Preparing the Edge Loops

As mentioned earlier, the algorithm requires a list of edge loops representing the incoming branches. These edge loops are derived from the vertex loops created during the construction of the branches. For the algorithm to be successful the edge loops that are

submitted for triangulation must meet the following criteria: The edges of a given loop must be ordered clockwise and lie on the same plane. All loops must be equidistant and tangential to the joint centre. No loop may contain any double edges. Although this final rule is met initially, double edges will occur as more faces are created. Consider the loop {... AB, BF, FB, BC ...}, edges BF and FB are the same edge. This must be detected and the loop must be reorganised to become {...AB, BC...}. An example of a double edge and it removal is illustrated in figure 19(c-e).

The joint is effectively constructed as a convex hull, however, sometime the loops that are submitted are not suitable for convex hull construction. This often occurs in complex bifurcations, where the offsets calculated in the graph construction stage exhibit great variation. To account for this, the loops must be projected onto a sphere and orientated tangentially. The radius of this sphere is defined as the largest offset of any of the branches. The sphere centre is the midpoint of all the loop centres, and a loop centre is the midpoint of the loop vertices. The joint can now be constructed. Once complete all vertices are translated back to their original positions. This process is illustrated below in figure Figure 4.15

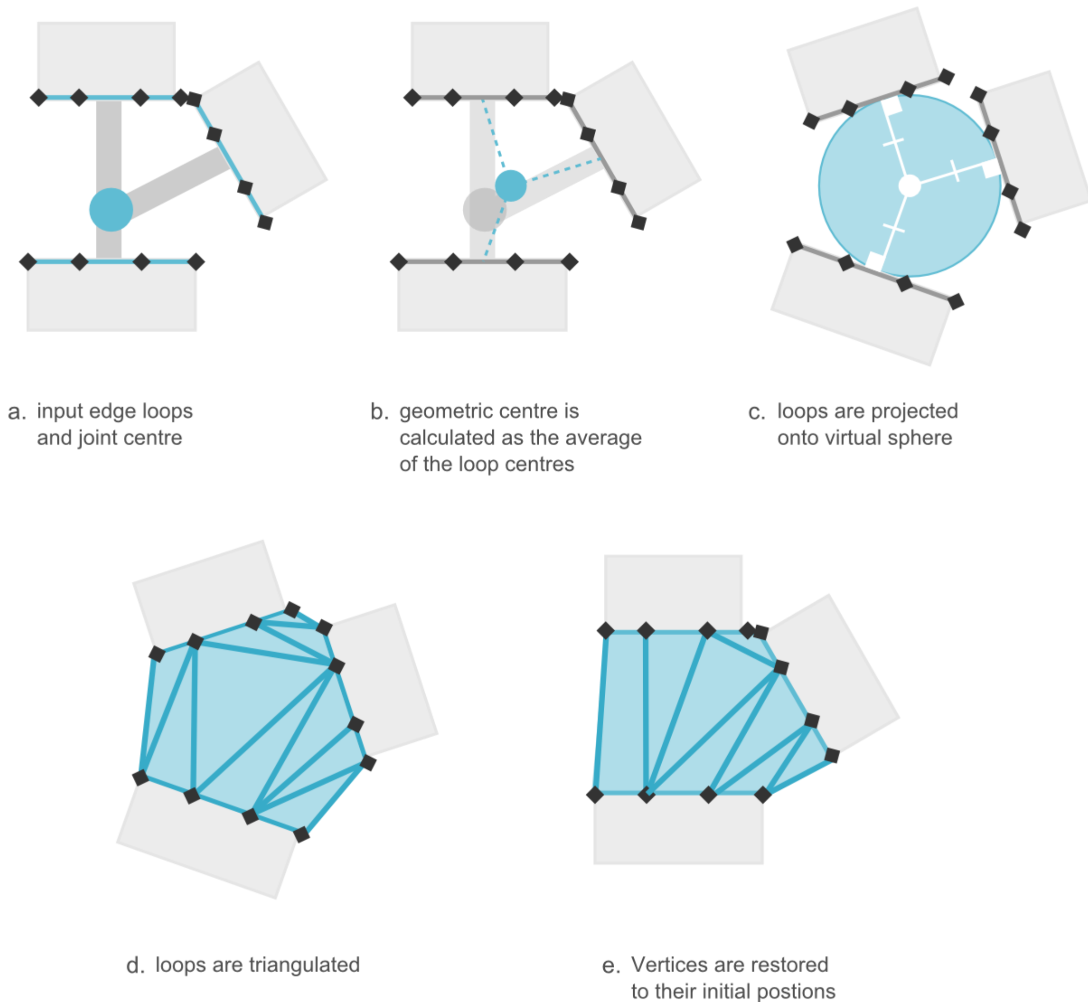


Figure 4.15: Figures a, b and c show how the loops are transformed to meet the criteria for joint triangulation. Figure d depicts how the triangulated joint. In figure e the vertices are transformed back to their original positions.

Algorithm 2: Joint construction

```
Input: edge_loops
complete ← false
while not complete:
    for every loop in edge_loops:
        for every edge in loop:
            if  edge has only one face:
                find a vertex from another to form a face with the  edge
                create new face from the edge and vertex
                reorganise loop
            end if
        end for
    end for

    complete ← true
    for every loop:
        for every edge:
            if  edge has only one face:
                complete ←  false
            end if
        end for
    end for
end while
```

4.3.3.2 Convex Hull Construction - Randomized Incremental Algorithm

During the implementation of the Interim Core Scheme it was discovered that the problem of triangulating the joints was, in fact, a convex hull problem in 3D. This was confirmed by Hijazi et al. [25] who formally pose joint construction as a hull construction problem. Not only is this approach significantly less complex, it is also highly robust. This is in contrast to the Interim Core Scheme, which cannot guarantee an absence of edge cases where the triangulation rules will fail.

The convex hull of a set of points is the smallest convex set that contains all of the points [46]. The algorithm implemented is the Random Incremental Algorithm first described by K.L. Clarkson and P.W. Shor in 1989 [47]. This algorithm was chosen for its simplicity as little

development time remained. A more efficient method, such as Quickhull would have been preferable.

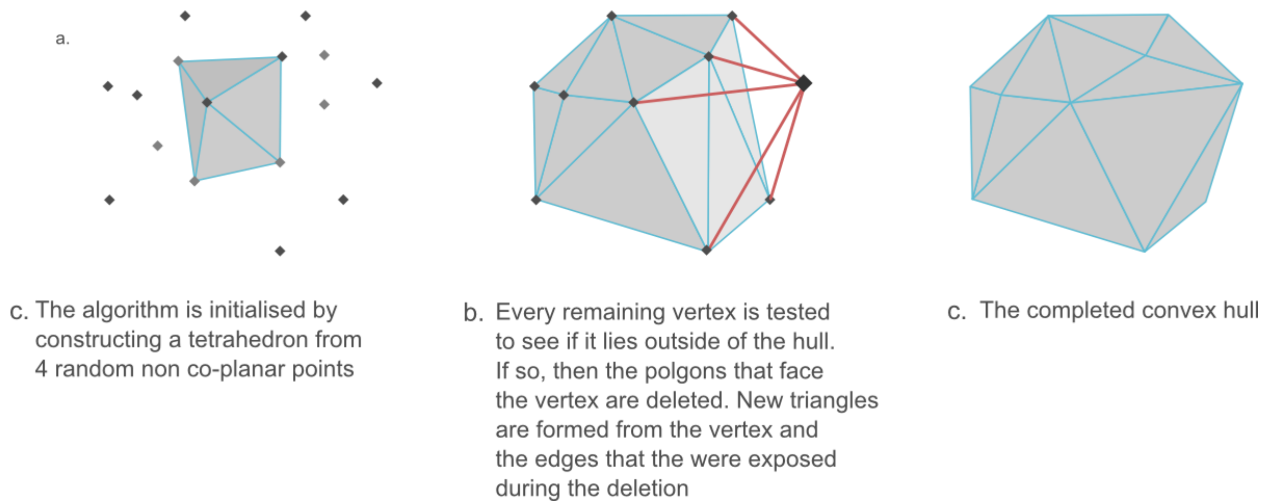


Figure 4.16: Convex Hull Construction with the Randomized Incremental Algorithm

The algorithm begins by removing four random non-co-planar vertices from the set. From these four vertices a tetrahedron is constructed. As the algorithm progresses, faces are deleted from and added to the tetrahedron, expanding it to a polyhedron that will eventually form the final convex hull. To achieve this, every remaining vertex is examined to see if it lies within the current hull. If it does then it is discarded. If it does not then the hull must be altered to include it. For this to occur, the triangles that face the vertex must be deleted. These triangles are found by examining the dot product between the face normal and the vector from the triangles midpoint to the vertex under consideration. If the dot product is greater than zero then the triangle faces toward the vertex and must be deleted. After deleting these triangles, several edges are exposed, now having only one incident face. New triangles are formed from the edges and the vertex being incorporated. The hull now includes the vertex that previously lay outside. This is repeated for every remaining vertex.

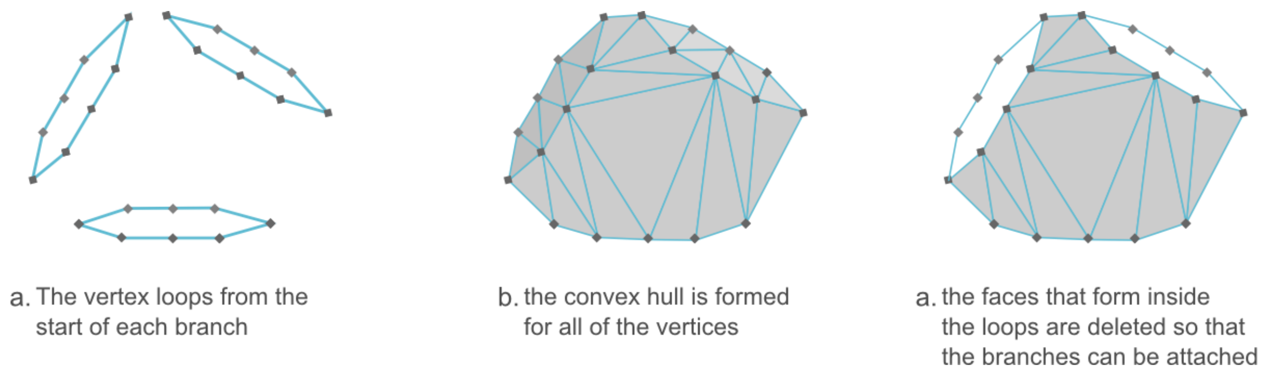


Figure 4.17: Constructing a convex hull for a set of vertex loops.

In the context of joint construction the set of points for which the hull must be constructed are the vertices at the start of each branch. As with the Interim Core Scheme implementation, these vertices are submitted as a set of vertex loops. Once the hull has been constructed the faces that form inside the loops are deleted so that the branches can be attached, The Randomized Incremental Algorithm is outlined below in Algorithm 3

Algorithm 3: Randomized Incremental Algorithm

Input: *vertex_list*

select 4 no co-planar vertices and construnct a *tetrahedron*
remove the 4 vertices from *vertex_list*
convex hull ← *tetrahedron*
for each *vertex* in *vertex_list*:
    for each *face* in *convex_hull*:
            visiblityVector ← *vertex* – *face_midpoint*
            dotvalue ← DotProduct of *visiblityVector* with *face_normal*
            if *dotvalue > 0*:
                delete *face*
            end if
        end for

    for each *edge* in *convex_hull*:
            if *edge* has only one incident face:
                create new face defined by the *vertex* and the 2 *edge* vertices
                add the face to the *convex_hull*
            end if
        end for
end for

4.2.4 Mesh Data Structure

For the mesh to be generated there needs to be a corresponding data structure. In its most elemental form a mesh can be represented as a list of vertices and a list of faces indexed to the vertices that define them. In order to render the mesh with shading, every vertex must have a corresponding normal, and if a texture is to be applied to the mesh, then a two dimensional texture coordinate is required for every face point. Both triangle and quadrilateral faces are supported in our implementation.

For the purposes of simplicity the mesh is represented in an object-oriented fashion, where every vertex stores its position and normal, and every face stores a list of pointers to the

vertices that define it. Each face must also store a list of texture coordinates which are ordered relative the vertex list. Texture coordinates are stored in the face rather than the vertices due to the occurrence of seams. A seam occurs when faces share vertices, but reference entirely different areas of the 2D texture space. The position and normal of a vertex are both stored as a 3D vector, while a 2D vector is used to store a texture coordinate. A vertex's normal is calculated as the average of its surrounding face normals. For a given face consisting of vertices A, B and C; its face normal is calculated as the normalized cross product between the edge BA and the edge BC. The vertex normals are used by OpenGL to perform smooth shading when rendering the mesh.

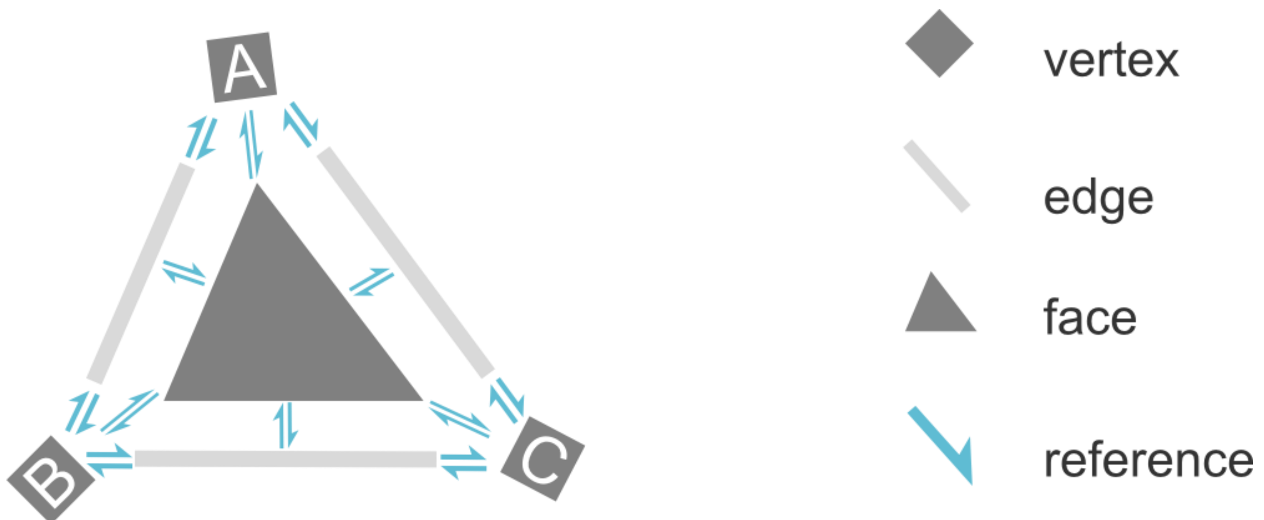


Figure 4.18: An illustration of the connectivity between the faces, edges and vertices of the mesh data structure

To simplify the implementation of surface subdivision, a C++ *struct* is used to represent edges in the mesh. When considering a mesh, or any other data structure, there is always a trade-off between performance and memory use. Mesh representations such as the Split-Edge data structure and Winged-Edge data structure are good choices for reducing the memory footprint as they store the minimum number of pointers needed to reach any component of the mesh from any other component. The drawback of these structures, however, is that they greatly increase the implementation complexity. They also have a performance overhead because, on average, more pointer traversals are necessary to access desired components. For the purposes of this implementation, a well-connected graph was used. Faces index their edges and vertices, edges index their vertices and faces, and vertices index their faces and edges. This is illustrated in Figure 4.18 below.

Dynamically updating this structure is a complex process and can make the simplest of algorithms cumbersome. As such the data structure is completely rebuilt after ever significant process. For example, an iteration of Loop subdivision splits all the edges and faces, so the data structure must be rebuilt before the next iteration. Although this is a costly

overhead, it is acceptable since performance is not the main concern of this research project.

Sample OBJ file: A quadrilateral formed from two triangles

```
g plane
v 1.118196 0.439762 -0.508643
v 0.914742 0.439762 1.491357
v -1.085258 0.439762 1.491357
v -1.082957 0.439762 -0.508643
# 8 vertices

vn 0.0000 1.0000 -0.0000
# 1 normal

vt 0.000575 1.000000
vt 1.000000 1.000000
vt 0.997600 0.002400
vt 0.004688 0.005600
vt 0.003200 0.996000
vt 0.999425 0.000000
# 6 texture coordinates

f 4/1 3/2 2/3
f 1/4 4/5 2/6
# 2 face
```

4.2.5 Wavefront .OBJ model format

The format used for exporting the final models is the Wavefront OBJ file format, which is a well-documented, open standard for representing geometry. It was chosen for its simplicity and ubiquity. The format can store multiple meshes per file, each of which is represented by plain text lists of vertices, normals, texture coordinates and faces. The format begins by listing all of the vertices in the mesh. These are indicated by the character 'v' followed by three floating point values that correspond to (x, y, z). The vertex normals, denoted as 'vn' are listed similarly. Next, the texture coordinates are listed as 'vt', with their values corresponding to (u, v, w). Finally the faces are defined. Indicated by the letter 'f', they are followed by one or more face points. A face point consists of v/vt/vn triplet, which index into the vertex, texture and normal lists respectively. A pitfall to be aware of is that the indexing starts at one, rather than zero.

Algorithm 4: Overview of the Mesh Generation Algorithm

```
Input:  rootBranch

construct branch faces for rootBranch
branchStack ←{}
branchStack.push(rootBranch)
currentBranch  ←null
while branchStack not empty
    currentBranch ← branchStack.pop()
    for each child in currentBranch
        construct branch section child child
        branchStack.push(child)
    end for
    if branch has children
        prepare the edge loops
        construct joint between branch and children
        re-triangulate the joint
        set the loop vertices back to their original positions
        perform edge collapses on the joint
    end if
end while
```

4.3 Parameterization

4.3.1 Introduction

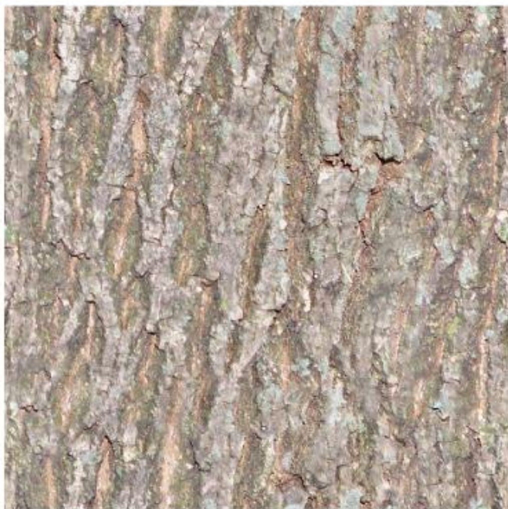
Parameterization is an important aspect of any mesh generation system as it allows a texture to be mapped to the surface. It is often also used for bump, normal and displacement mapping. Mesh parameterization refers to a piecewise linear mapping between the faces of the mesh and a simpler domain, usually a 2D plane. Texture mapping associates every point on the mesh's surface with a corresponding point in a texture. Regardless of the image dimensions the mapping domain always lies between [0, 0] and [1, 1] where [0, 0] is the lower left corner of the texture. The components of the two dimensional orthogonal basis used to describe texture space are U and V. These correspond to the X and Y axes respectively, because of this parameterization sometimes referred to as UV mapping.

Figure 4.19: Left: A model that has not been parameterized. Right: A parameterized model with a texture applied.

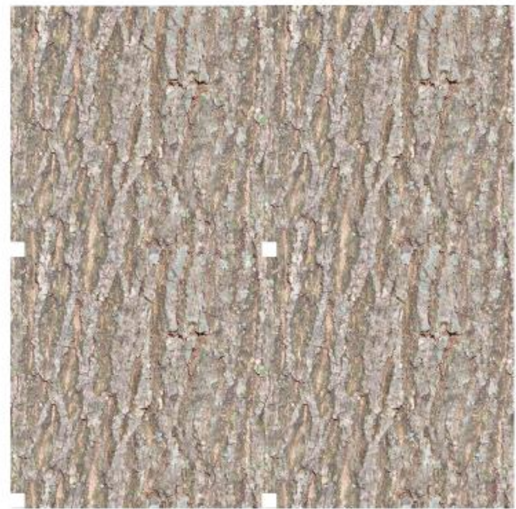
The automatic parameterization of a general mesh is a challenging problem that easily warrants a research project of its own. CGAL (http://www.cgal.org/), an open source library of common graphics operations, was initially investigated as a potential solution to the parameterization of the control mesh. Unfortunately, bark textures have an added constraint of directionality, necessary to produce the vertical striations present in nearly all bark. Since the solutions provided by CGAL are general, they do not take this constraint into account. If CGAL were used to derive a parameterization then the striations would wrap around the branches in arbitrary, unpredictable ways.

Instead, a method of parameterization is integrated directly into the mesh generation process. This allows the topological structure of the tree to be considered, so that the striations in the bark texture are mapped along the length of a branch. In accordance with the stages of the mesh generation process, first the branch sections are parameterized, then the joints sections. The intention is to produce as few seams as possible, minimise stretching and map the striations along the length of the bark.

The texture that is applied to the model is a square seamless sample of bark. The term *seamless* implies that should two of the samples be placed adjacent to one another, then the boundary where one sample ends and the next begins will be imperceptible. An example of such a texture is displayed in Figure 4.20. The texture synthesis module, developed by Ryan Mazzolini [7], is responsible for generating seamless textures for the system. Given a small sample of bark, such as a photograph, the module generates a seamless (usually a larger) texture that is recognisably similar to the sample. The texture synthesis technique employed is often referred to as discrete optimisation [48]



a. seamless bark texture



b. 2x2 tiling of the texture

Figure 4.20: An example of a seamless texture that could be applied to the model. The white squares indicate the lower left corner of the texture

4.3.2 Parameterization of the branch segments

The first faces of the mesh to be parameterised are the branch sections. Since a branch is modelled as truncated cone, which is topologically equivalent to a cylinder, texture coordinates are assigned branches using a straightforward cylindrical parameterisation. This is achieved by creating loops of texture coordinates along the corresponding vertex loops. The U component is calculated based on a vertex' loop index. The value of V is interpolated

along the length of the branch, repeating with every 2 segments. This causes the texture to be tiled along the length of the branch. When the faces are created they are assigned the texture coordinates corresponding to the vertices that define them. This is illustrated in figure 17.

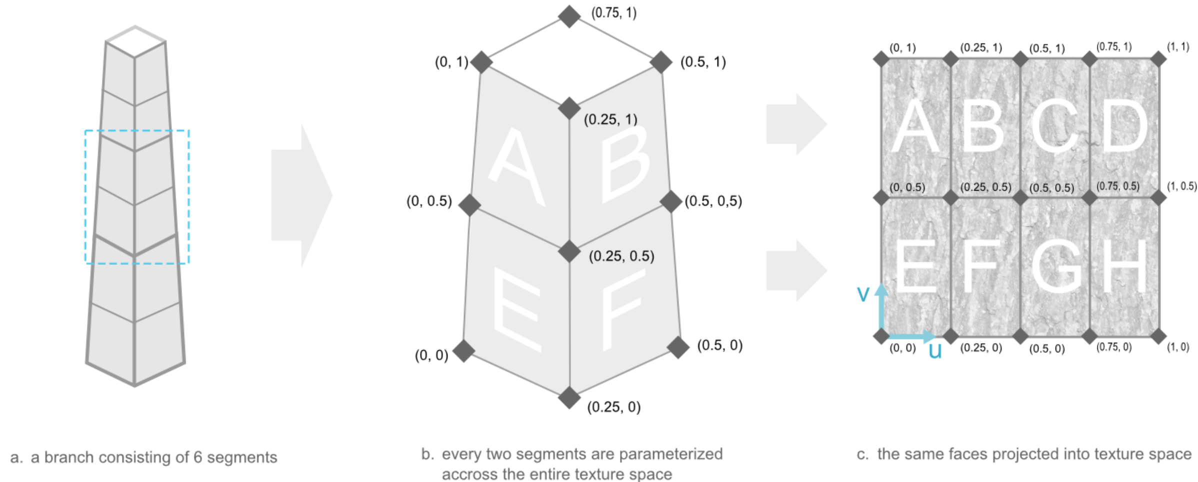


Figure 4.21: This is an illustration of how a texture is wrapped around a branch, as well as how the faces of the branch are unwrapped in texture space. For simplicity the faces are depicted as quadrilaterals. However, in the final implementation they are in fact tessellated into triangles.

It was explained in section 3.2.1 that the branches are tessellated into a number of shorter segments based on the following formula:

$$AverageRadius = \frac{StartRadius + EndRadius}{2}$$

$$N = \left\lceil \frac{BranchLength}{AverageRadius * \pi} \right\rceil$$

Since N is derived by dividing the branch length by half the circumference, every two segments unwrap approximately to a square. The dimensions are usually not uniform (ie not precisely a square) due to the rounding of *N*). Using the average radius means that some squashing will occur near the fatter starting segments and some stretching will occur near the thinner ending segments. However, these artefacts are hardly noticeable.

4.3.3 Parameterization of the Joint Sections through Interpolation

With the branch sections parameterized, all that remains is to parameterize the faces that form the joints. This, however, is a difficult problem to solve. Seams will inevitably be introduced, since the surface of a joint is not topologically equivalent to a planar texture

domain. To parameterize these faces, the textures coordinates of faces that have already been parameterized are interpolated across adjacent faces that have not yet been parameterized. This method keeps stretching to a minimum, and preserves the directions of bark striations.

The algorithm for interpolating the coordinates is as follows: Coordinates are assigned on a per triangle basis. Every triangle that has not yet been parameterized selects the adjacent triangle with the longest shared edge. If the selected adjacent triangle is not itself yet parameterized, then the second longest edge is examined and so on. If the selected triangle has been parameterized then the texture coordinates are interpolated across the vertices of the un-parameterized triangle. An example of this can be seen in Figure 4.22c. The coordinates that are interpolated always lie outside the texture space. To account for this wrapping conditions are applied to the coordinates, such that if U or V are greater than 1 or less than 1, then 1 is subtracted or add added respectively. This is not noticeable as the bark texture used is seamless.

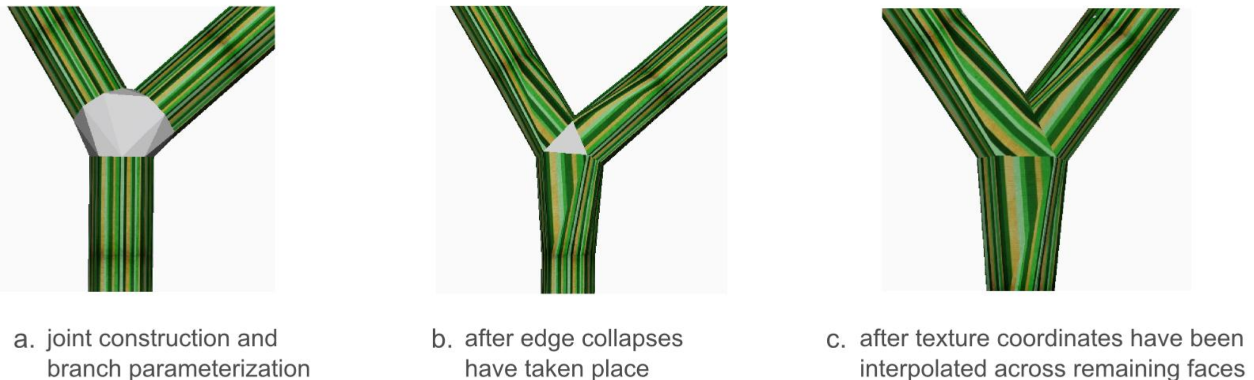


Figure 4.22: the various states of parameterization. Left: directly after the joint has been triangulated. Middle: After the edges of the joint have been collapsed many of the un-parameterized faces have been deleted. The final face is parameterized by interpolating the texture coordinates from the top left branch.

4.3.4 Collapsing edges

Unfortunately seams must be introduced somewhere. This method of parameterisation leads to multiple zigzagging seams occurring at the joints. In an attempt to reduce the number of seams a post process step was implemented that collapses the edges of the joint mesh. An edge collapse refers to merging the two vertices that make up an edge into a new vertex, which is a weighted average of the vertices being merged. Every time an edge collapse occurs, the faces on either side of the edge are also implicitly collapsed. Collapsing edges reduces the number of triangles that need to be parameterized through interpolation. Figure 4.22 shows what the joint looks like before and after performing edge collapses. In special cases all of the edges in the joint can be collapsed leaving no triangles

left to be parameterized through interpolation. A case where this occurs is planar bifurcations. Most of the time, however, there are a few isolated faces which must still be parameterized using the interpolation method described above. A second benefit of performing edge collapses is that if yields rounder curves between branches on the limit surface, this is illustrated in Figure 4.23.

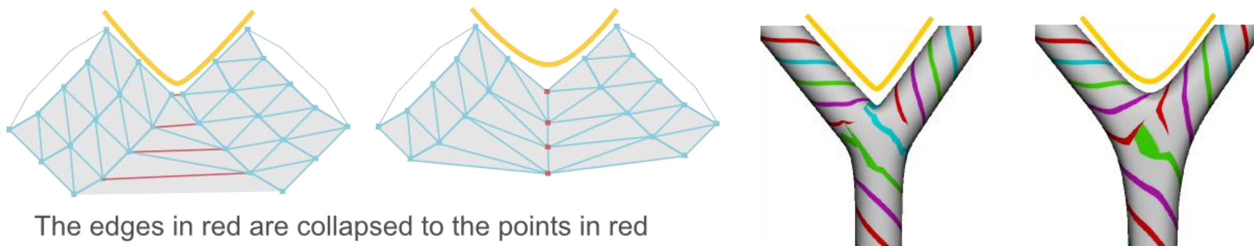


Figure 4.23: After collapsing the edges, the new topology will produce rounder curves on the limit surface when subdivided. The curves are illustrated in yellow.

The heuristic for collapsing edges is simple: Always select the shortest edge. To assist in this process the edges are stored in a priority queue based on their length. An edge cannot be collapsed if its vertices share the same initial boundary loop. This ensures that the joint maintains its overall structure since none of the initial edge loops that belong to the incoming branches are lost.

Unfortunately using the edges length as the heuristic turned out to be naïve. Although the method produced pleasing results in most cases, edge cases were encountered where the method flat out failed. Such a case is demonstrated in the figure below. In favour of robustness the edge collapse post process was left out of the final system. Despite this the method was an interesting avenue to exploration and a detailed explanation of the edge collapse algorithm is found in appendix B.

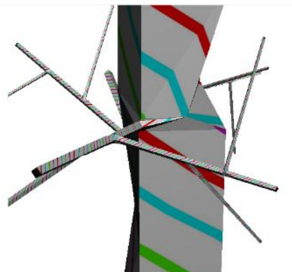


Figure 4.24: An example of a degenerate edge collapse. This occurred as a result of the large variation in the branch radii.

4.3.5 Limitations

Despite the simplicity of this parameterization method, the results produced are reasonable. Nonetheless, there are shortcomings that must be discussed. The first of these is that the parameterization does not produce a one-to-one mapping. This means that although every point on the surface of the mesh maps to exactly one location in the texture, the reverse is not true; every location on the texture maps to multiple different locations on the mesh surface. This is both good and bad, depending on the context. Very often it is desirable to reuse the same texture space in order to save texture memory. Besides memory efficiency, reusing texture space means that more texture detail can be seen on any given part of the mesh. The downside is that reusing the same texture space introduces repetition if the bark texture has any distinct features. Furthermore, an artist would be prevented from loading the model into a 3D package to detail to specifics area of the mesh. However, above all these limitations is the fact that the mesh is not appropriate for texture synthesis over surfaces. This approach has the potential to produce good results by hiding the seams. However, our parameterisation method could easily be extended to a one-to-one mapping by creating a bounding box around every chart and organizing them so that none overlap.

An alternative approach to texture synthesis is to line the seams up so that the mesh appears seamless. This is no easy task but recently Kalberer et al [44] proposed an extension to the Quad Cover algorithm [49] that parameterizes tube-like surfaces with a globally consistent stripe texture. Although this method still introduces seams similar to those in the present system, the coordinates are assigned such that the textures are hiding the appetency of the seams. The results produced are quite promising. To achieve this, the principal curvature frame field of the tube-like structure is used to drive the parameterization process.

4.4 Subdivision Surfaces

4.4.1 Introduction

By this point the control mesh is fully constructed and parameterized. The final step is to apply surface subdivision to produce a smooth high resolution mesh. The general theory and background behind subdivision surfaces is covered in chapter 2. In this section an implementation of the Loop Subdivision scheme is presented. The foremost motivation for our use of subdivision is to smooth the surface where branches meet. This is what most significantly distinguishes the models produced by Yggdrasil from those produced by TreeDraw. In prior work, the models were constructed as a set of intersecting generalized cylinders defined along a Bezier curve. Although this Bezier curve produced a smooth blend between child and parent, it does not blend child-child intersections. In contrast, through the iterative application of loop subdivision to a generated control mesh, all branches are blended smoothly. A comparison of the models appears in Figure 4.25.

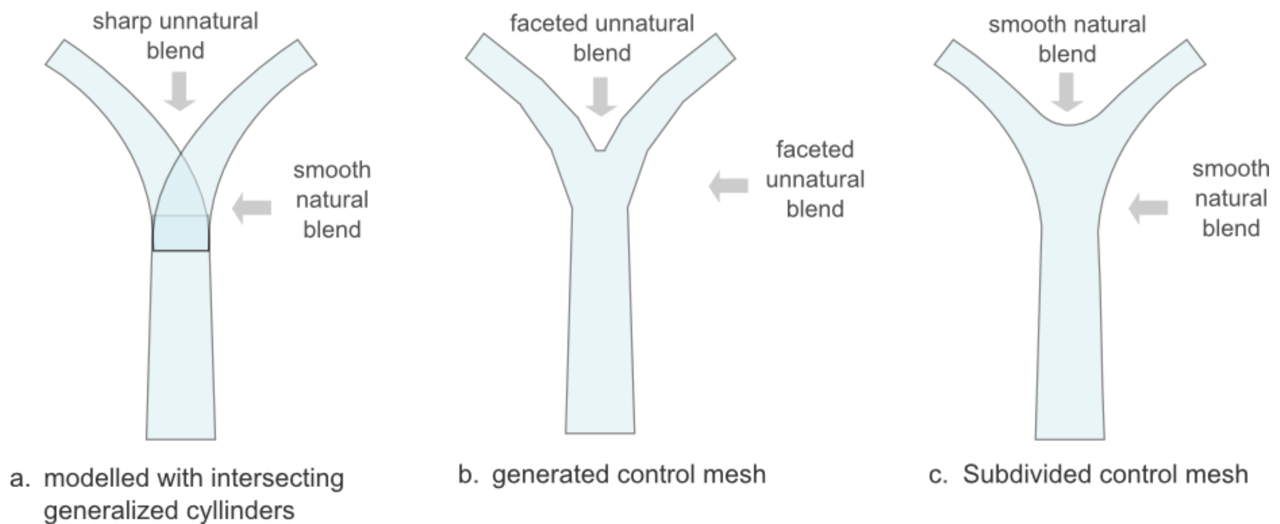


Figure 4.25: a: The structure of the model produced by the previous system. b: the control mesh produced by the mesh generation module. c: A high resolution mesh obtained through the repeated application of Loop subdivision to the control mesh.

This subdivision component was designed modularly and has no direct dependencies on the mesh generation component, other than requiring the generated control mesh as input. There is nothing unusual about the control mesh; any arbitrary OBJ model can be loaded subdivided, provided it consists entirely of triangles. There are two requirements of the subdivided mesh: Firstly, the base and branch tips must remain fixed; secondly, since the mesh is parameterized the texture coordinates must propagated down to the subdivided

mesh. The texture should not be shifted, stretched or otherwise deformed beyond reason by the subdivision process.

4.4.2 Loop Subdivision

As discussed in the background chapter on subdivision surfaces, Loop is designed specifically for meshes consisting entirely of triangles. The mesh data structure is constructed to be as convenient as possible. For this purpose, each face contains a list of vertices {A, B, C} as well as a list of edges {AB, BC, CA}. Each edge points to two faces. Both vertices and edges also point back to the faces the constitute them. There are three steps in the algorithm. First, a new *edge point* is constructed for every edge. The edge point is a weighted average of the edge's end points and the two far points that form the triangles incident on the edge.

$$edgePoint = \frac{3}{8}(endPointA + endPointB) + \frac{1}{8}(facePointA + facePointB)$$

The set of new vertices are referred to as odd vertices while the original vertices are referred to as even. Since the control mesh generated is open at its base and branch tips. It is necessary to modify this rule for boundary cases. A boundary is an edge with only one incident face. If the rule were simply used as it stands then the boundaries would pull away from their initial positions. This would significantly alter the structure of the tree. To account for this, boundary edge points at boundaries are simply the average of the end points, the opposite face point is not considered.

$$boundaryEdgePoint = \frac{1}{2}(endPointA + endPointB)$$

Next, the positions of the original vertices in the control mesh are updated. The new location is based on the old position and the surrounding original vertices.

$$where\ n\ is\ the\ number\ of\ adjacent\ vertices, and\ k\ is\ the$$

$$weighting\ assigned\ to\ surrounding\ vertices:$$

$$k = \begin{cases} n = 3: & \frac{3}{16} \\ n > 3: & \frac{1}{n}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\sin\left(\frac{2\pi}{n}\right)\right)^2\right) \end{cases}$$

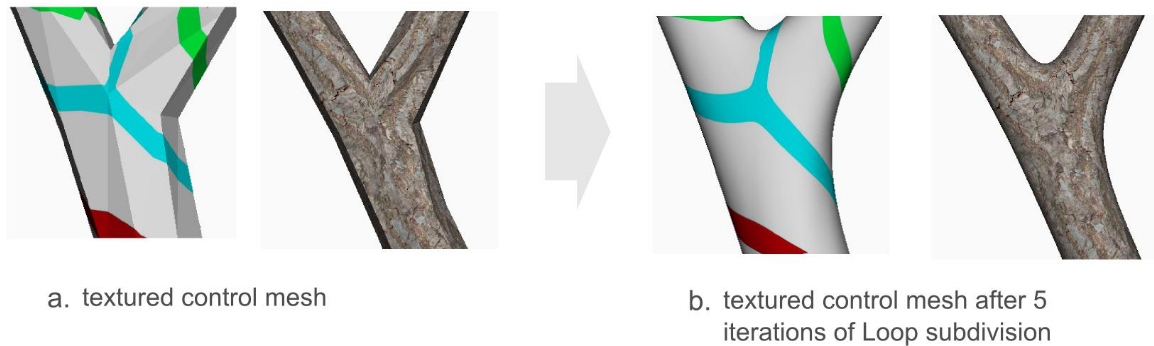
$$where\ \beta\ is\ the\ set\ of\ adjacent\ vertices:$$

$$newPosition = (1 - n \times k) \times \mathrm{oldPosition} \quad + \mathrm{k} \times \sum_{\mathrm{i}=0}^{\mathrm{n}} \beta_{\mathrm{i}}$$

Every triangle is subdivided into four new triangles. One triangle is formed in the centre and is defined by the three edge points. The other three triangles are each constructed from one of the original vertices and two of the new edge points. The fractional values ensure that the new point is normalized. Although not implemented in our system, the efficiency of the algorithm can be improved by constructing a matrix and performing a linear transformation on the surrounding vertices.

4.4.3 Subdividing texture coordinates

Subdividing texture coordinates is more challenging than subdividing faces. This is due to the presence of seams. A seam occurs when a vertex is be associated with multiple texture coordinates, depending on which face is being considered. If this was not the case and every vertex was simply associated with one texture coordinate then subdividing the coordinates becomes trivial. The subdivision rules are simply applied in 5-space {x, y, z, u, v} where u and v are texture coordinates. Since this is not the case and subdivision at seams is complex, it was decided that texture coordinates would simply be subdivided linearly. With this approach the texture coordinate of a new edge point is set as the midpoint of the texture coordinates associated with the edge. In general such a linear approach produces poor results with visible discontinuities across the subdivided surface. However, due to the inherent structural randomness of bark much of these artefacts are hidden. Figure 19 contains two examples of surfaces with subdivided coordinates.



a. textured control mesh

b. textured control mesh after 5 iterations of Loop subdivision

Figure 4.26: An illustration of the effect of linearly subdividing the texture coordinates with every subdivision step.

4.5 Discussion

In this chapter a procedure for generating a mesh from a graph was presented. Although the graph implemented was directed and acyclic, this method can be generalized to a graph of arbitrary topology. During this construction of this graph, joints that overlap with one another are identified and merged. This is one of the algorithm's shortcomings as it changes the structure of the final tree. Once the graph is complete, mesh generation begins by constructing all the branch segments between the joints. Each branch is parameterized such that a bark-like texture is tiled along its length. The reason these branch segments are created first is that the vertex loops that define their ends are needed as input for the joint construction routine. The joint is created by constructing convex hulls around the input vertex loops, and then removing the faces that form within the loops. This was the second joint construction algorithm implemented. The first approach, known as the interim Core Scheme, took up a great deal of development time, but was abandoned as it could not guarantee robustness.

To complete the control mesh, all that remains is to parameterize the faces that form the joint. This is achieved by interpolating the texture coordinates that were assigned to the branch faces across the faces in the joint. This produces reasonable results, however seams are introduced. The control mesh is now finished, but is a coarse representation of the tree. In order to create a smooth and natural surface, Loop Subdivision is applied. On average only the iteration of subdivision on are necessary to achieve reasonable smoothness, however this is entirely dependent on the distance between the camera and the model.

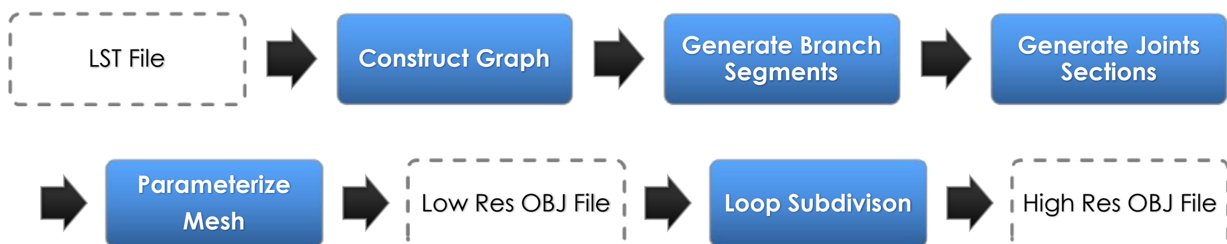


Figure 4.27: The steps involved in constructing the model as well as the file outputs

To be of use, this procedure was interpreted into TreeDraw's procedural generation pipeline. Rather than replacing the generalized cylinder models produced by TreeDraw, the user is able to choose which model they would like to generate. The motivation for this is that the branches in the generalized cylinder models have the benefit of being converted into Bezier curves. By comparison the branches of our models often look unnaturally straight. To compensate for this, users can approximate curves as a sequence of short line segments when sketching the tree. These will become a smooth curve when subdivided.

Both models can be exported as an OBJ file. Exporting subdivided models is possible, though due to their size, it is advisable to simply export the control mesh. The model size approximately quadruples with every iteration of subdivision. The control mesh can be loaded and converted into a subdivision surface in most 3D modelling and rendering packages, however if they don't support Loop then the surfaces produced will likely be inferior.

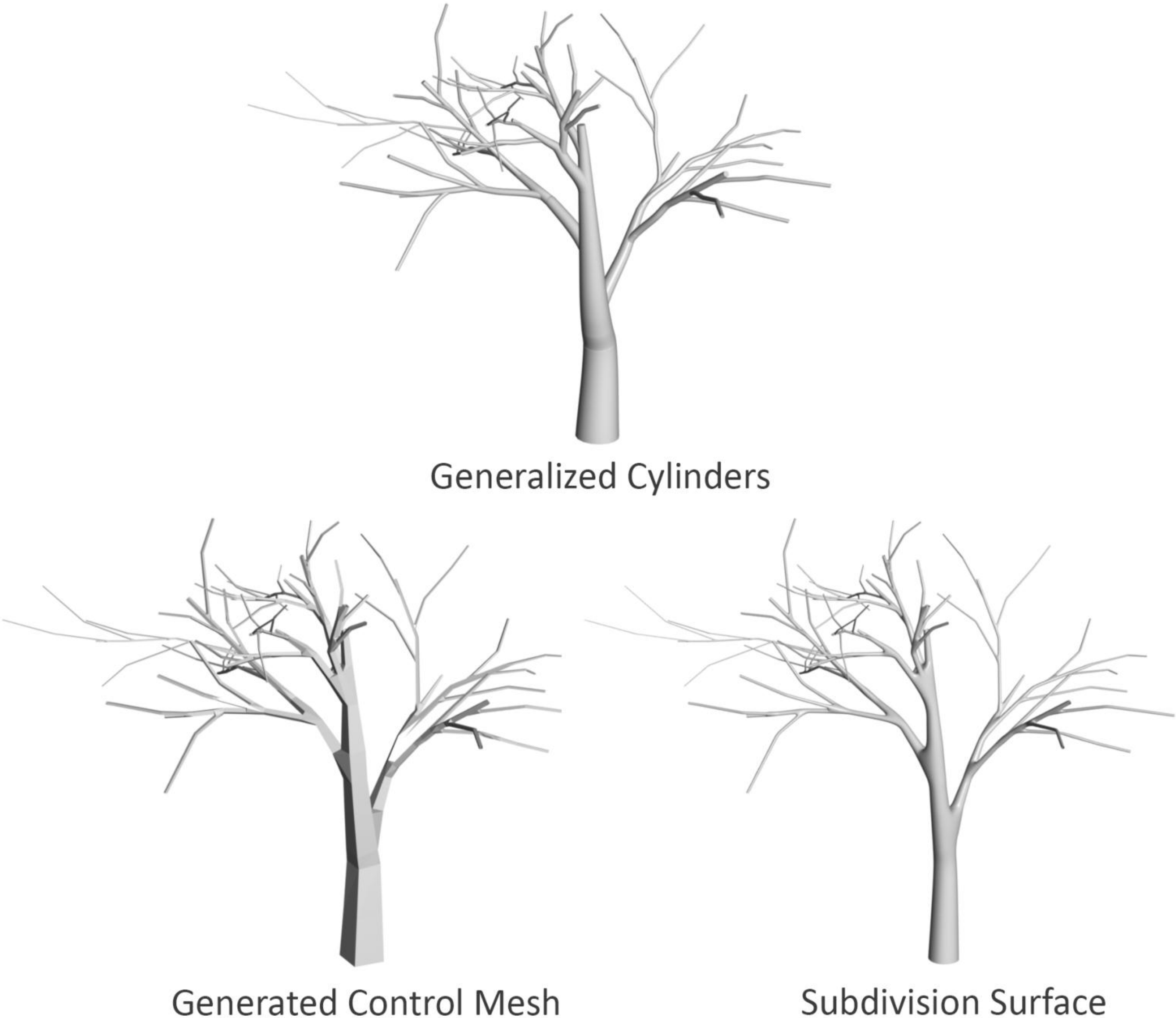


Figure 4.28: Three models generated from the same LST file. The model at the top was created by TreeDraw's existing model generator. The bottom left models was created by the mesh generator described in this report. This coarse mesh is then converted into the subdivision surface seen in the bottom left

Chapter 5

Experimentation and Results

This chapter details three separate evaluations that were performed, as well as the results obtained from them. The purpose of these evaluations was to answer the main research question and determine the overall success of the project. The three evaluations were:

1. An initial evaluation performed midway through development
2. A final evaluation directed at answering the primary research question
3. Performance testing to analyse the scalability of the solution.

5.1 Initial Evaluation – Expert Users

The initial evaluation took the form of an expert user test. The goal of this evaluation was to receive feedback on the algorithms used and quality of the models produced. In total five users took part; two lecturers from the computer sciences department and three computer science masters student. All of the users had a wealth of experience and provided unique insights into the field of computer graphics. The solution was demonstrated to each user on a one-on-one basis over the course of a week. This allowed feedback from earlier users to be incorporated before demonstrating to later users. As a result later users identified issues that were not perceptible or did not exist earlier.

It was decided that the best approach to initial development would be to create a complete system but with minimal required functionality for individual stages. Once the core functionality was in place, further implementation iterations were undertaken. An informal evaluation was performed midway through the development cycle in order to identify focus areas for remaining development. At this point only the Interim Core Scheme, branch parameterization and loop subdivision had been implemented, and the solution had not yet been integrated into TreeDraw. Instead, a separate GUI was created. This interface allowed users to load an LST file, perform the mesh construction algorithm and apply multiple iterations of surface subdivision. Two slider controls were exposed that

allowed users to control the progress of the algorithms. The first slider controlled the progress of the joint construction algorithm which could be halted at any point by moving the slider back and forth. The second slider set the number of iterations of loop subdivision applied to the model. An interactive 3D OpenGL viewport was implemented in which the model could be rotated and inspected. The most significant issues identified by users fall into three distinct categories. These are construction errors in the joint mesh, surface smoothness, and mesh parameterization issues.

5.1.1 Joint Mesh Construction Errors

Allowing users to load up the various LST files and inspect the constructed joints at their leisure exposed a lot edge of cases in the construction algorithm that had previously gone unnoticed. These were occurrences where the joints of the mesh were poorly constructed. Issues exhibited include intersecting polygons, polygons that pass through the joint centre, and holes in the mesh surface. One of the initial steps in the joint construction algorithm involves projecting the ends of the branches onto a sphere. Upon investigation, it was determined that the edge cases were the result of defining the joint centre as the end of the parent branch. In cases where all branches lie on the same side of the centre, the triangulation rules fail. To address this, the joint centre is geometrically calculated as the midpoint of the all the branch starts. This distributes the projected branches more evenly around the sphere, and leads to improved robustness of the ICS algorithm.

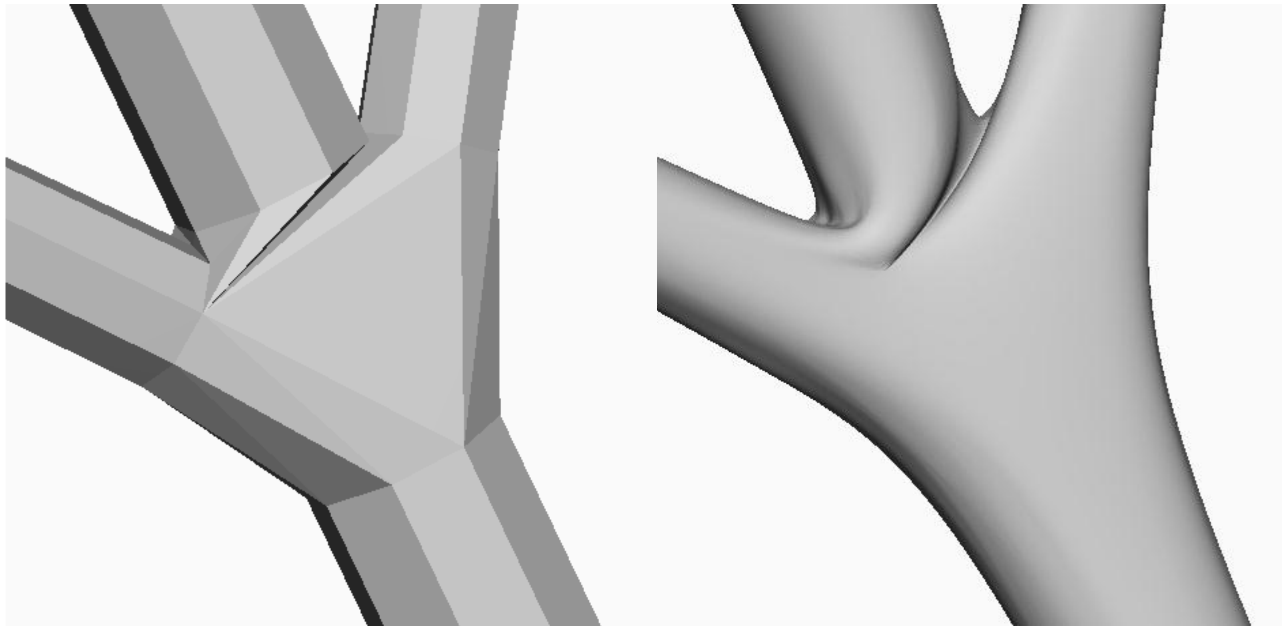


Figure 5.1: An example of an edge case where faces intersect.

5.1.2 Subdivision Surface Smoothness

The purpose of a subdivision surface is to model a smoothly curving surface; however, the surfaces were exhibiting ripples and undulations near the joints. One user highlighted that although these were a theoretical issue, they actually produced a more tree-like surface. Nevertheless the source of the undulations was investigated and it was found that they were the result of under-tessellating the branch segments between the faces. This is because the Loop subdivision scheme first inserts new vertices at approximately the midpoint of every edge and then smooths the positions of the original vertices by pulling them towards their immediate neighbours. The further a neighbouring vertex is, the stronger its pull. Since the branches were under tessellated, the vertices were far apart. This resulted in a large pull leading to a significant difference between the new midpoint and the smoothed position of the original vertex. These differences lead to a ripple across the surface. To resolve this, the branch segments were more finely tessellated based on their length and radius. This results in shorter neighbour distances and a much smoother and more predictable limit surface.

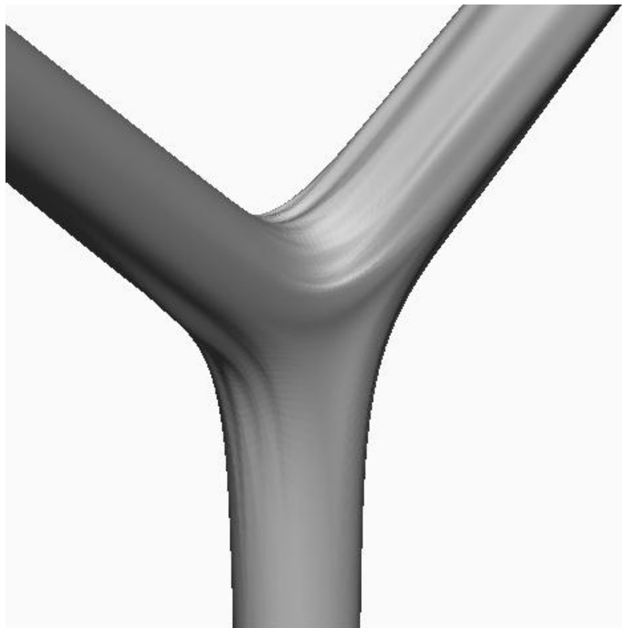


Figure 5.2: Ripples across the subdivision surface

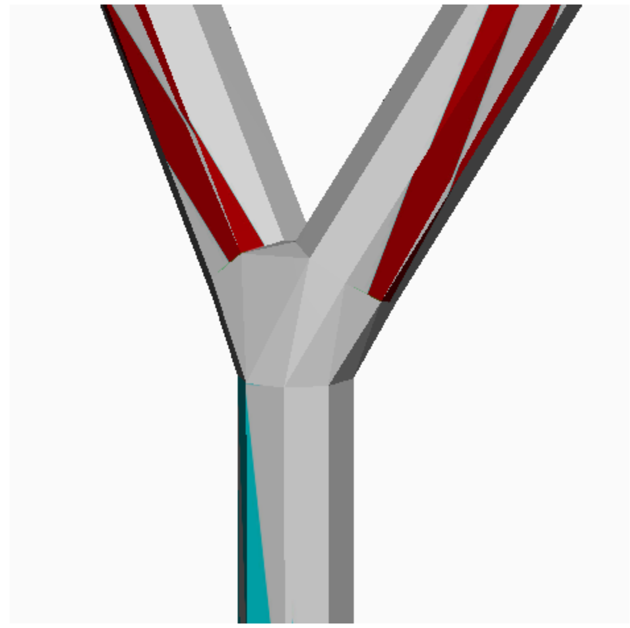


Figure 5.3: Mesh that forms the joint had not yet been assigned texture coordinates.

5.1.3 Mesh Parameterization

Two issues regarding parameterizations were identified. The first was that the polygons in the joint did not yet have texture coordinates assigned to them, and as such were not yet textured. The reason for this was that the parameterization of the joint mesh had not yet been implemented, and was thus was not a significant case for concern. The second, more

problematic, issue raised was that the texture coordinates were not being smoothly subdivided with the mesh; instead they were simply being linearly interpolated. This was because subdividing texture coordinates is inherently difficult due to the unavoidable presence of seams. Seams occur when one vertex maps to multiple texture coordinates. Linearly interpolating texture coordinates leads to distortions in the texture mapping. Although this is problematic, one user rightly pointed out that this warping is hidden by the noise present in bark patterns. As such improving, the subdivision of coordinates was assigned a low priority.

5.2 Final Evaluation – Participant Study

The intended outcome of the final evaluation was to establish a quantitative measure of the realism for models produced by both the previous system and by this project, and answer the following research question:

> Can the realism of the branching structure be improved through the application of subdivision surfaces?

5.2.1 Experimental Method

To experimentally evaluate this question, a participant study was undertaken; specifically a single factor within-subjects design (repeated measures). Participants were asked to separately score the realism of the models produced by TreeDraw and those constructed using subdivision surfaces (Yggdrasil). T-tests were applied to the score distributions to discover how the models compared statistically. The null hypothesis postulated is:

> H_0: The score distribution for models generated by Yggdrasil is the same as the score distribution for models generated by TreeDraw.

> H_1: The score distribution for models generated by Yggdrasil is higher than the score distribution for models generated by TreeDraw.

The models themselves were not presented to participants; instead shaded renders of the models were displayed. The alternative to presenting images would have been to display the models in a 3D interface and allow users to rotate and examine the models at their leisure. The reason this route was not taken was that the TreeDraw models approximate the skeleton of the tree using Bezier curves, while the models produced by Yggdrasil interpret the skeleton directly. The use of Bezier curves arguably creates trees which curve more realistically. Since the research question pertains specifically to modelling the joints as subdivision surfaces, it was decided that the Bezier curves would confound the realism scores. As such renders which focus on the joints were used instead. A consequence of this

is that the results obtained do not indicate which method produces a more realistic model as a whole.

In attempt to compare the realism of models produced by Yggdrasil to real trees, a second set of silhouetted images were presented to participants. These silhouettes were created for the models by rendering them without shading. Creating silhouettes of real trees proved more arduous. Photographs of trees needed to be manually segmented and masked. The reasoning behind using silhouettes is that many of the confounding variables are removed, leaving only the structure of the branch. From this a second null hypothesis was postulated:

> H_0: The score distribution for silhouettes of models generated by Yggdrasil is the same as the score distribution for silhouettes of real trees.
>
> H_1: The score distribution for silhouettes of models generated Yggdrasil is lower than the score distribution for silhouettes of real trees.

To reject both of these hypotheses a contradicting result must be found that lies within the 5 percent range of significance ($p = 0.05$) using a two-tailed paired t-test. The dependent variable in the study is the realism score that participants assign to an image. The independent variable is the source of the image: either a model produced by TreeDraw, a model produced by Yggdrasil, or a photograph of a tree.

Participants were first presented with the set of shaded renders, then the set of silhouettes. In both cases the images were displayed one at a time. For each image, participants were asked to assign a score based on how realistic the branch depicted in the image was. The scale used for scoring was a continuous analogue scale between 0 and 100, where 0 represents “not at all realistic” and 100 represents “Highly realistic”.

The benefit of using a repeated measures design is that each participant is used as their own control. This is important as realism is a highly subjective quality. It makes no difference whether a participant scores all images between 5 and 15, 20 and 80, or 60 and 100; one image is scored higher than another. A repeated measure design also does not require as many participants to attain statistical significance compared to a between-groups design. The drawback of the repeated measures design is carryover effects. Fatigue, boredom and practice effects all serve to reduce the internal validity of the experiment. These issues were addressed through partial counterbalancing by randomizing the order of the images in each set. However, the algorithm used for randomization did not guarantee that any given permutation would only be shown to one user. Furthermore, the order that the sets were presented to participants was fixed. The order of the sets was always: bark, shaded renders, silhouettes. This leads to potential carryover effects occurring between the shaded renders and the silhouettes.

Testing Interface

The research question pertaining to subdivision surfaces was not the only question evaluated during this experiment. The realism of the textures produced by the texture synthesis module was also gauged. This was accomplished through a similar methodology: participants were presented with square samples of synthesized bark textures, as well as square samples of real bark taken from photographs, and asked to score both on their realism. This led to a total of three image sets presented to participants.

1) Shaded Renders of:
    - Models produced by TreeDraw
    - Models produced by Yggdrasil
2) Silhouettes of:
    - Models produced by TreeDraw
    - Models produced by Yggdrasil
    - Segmented images of real tress
3) Bark textures:
    - Samples obtained from photographs
    - Samples Synthesized from photograph samples

To display these images and capture the assigned scores, a graphical user interface was developed. The interface was designed collaboratively to meet the testing needs of both research questions and was implemented by a fellow project member. Due to its robust image loading functionality, QT was the chosen framework for developing the interface. The data collected was saved anonymously in comma separated format (.CSV) so that it could be easily imported into a spread sheet application and analysed. The testing interface consisted of three sections: image rating, commenting on images that received low ratings and, finally, a demographic information entry form.

The image sets were presented one after another in a fixed order, and the order of the images in the each set was randomized. The algorithm used for randomizing the order is a simple one. For every image in the set, pseudo-randomly select another image and swap them. The choice to display the images sets in a fixed order was a concession that resulted from implementation time constraints; the initial experimental design called for randomization of the set order. It is likely that presenting the shaded images first affected the scores assigned to the silhouetted images later. When it became apparent that the set order would be fixed, it was decided that it would be better to display the shaded images first, thus providing context for the silhouettes. Without this context participants were more likely to misunderstand what the silhouettes represented. Since the no conclusions are drawn by comparing the distributions from the shaded images to those of the silhouetted images, the carry over effects are negligible.

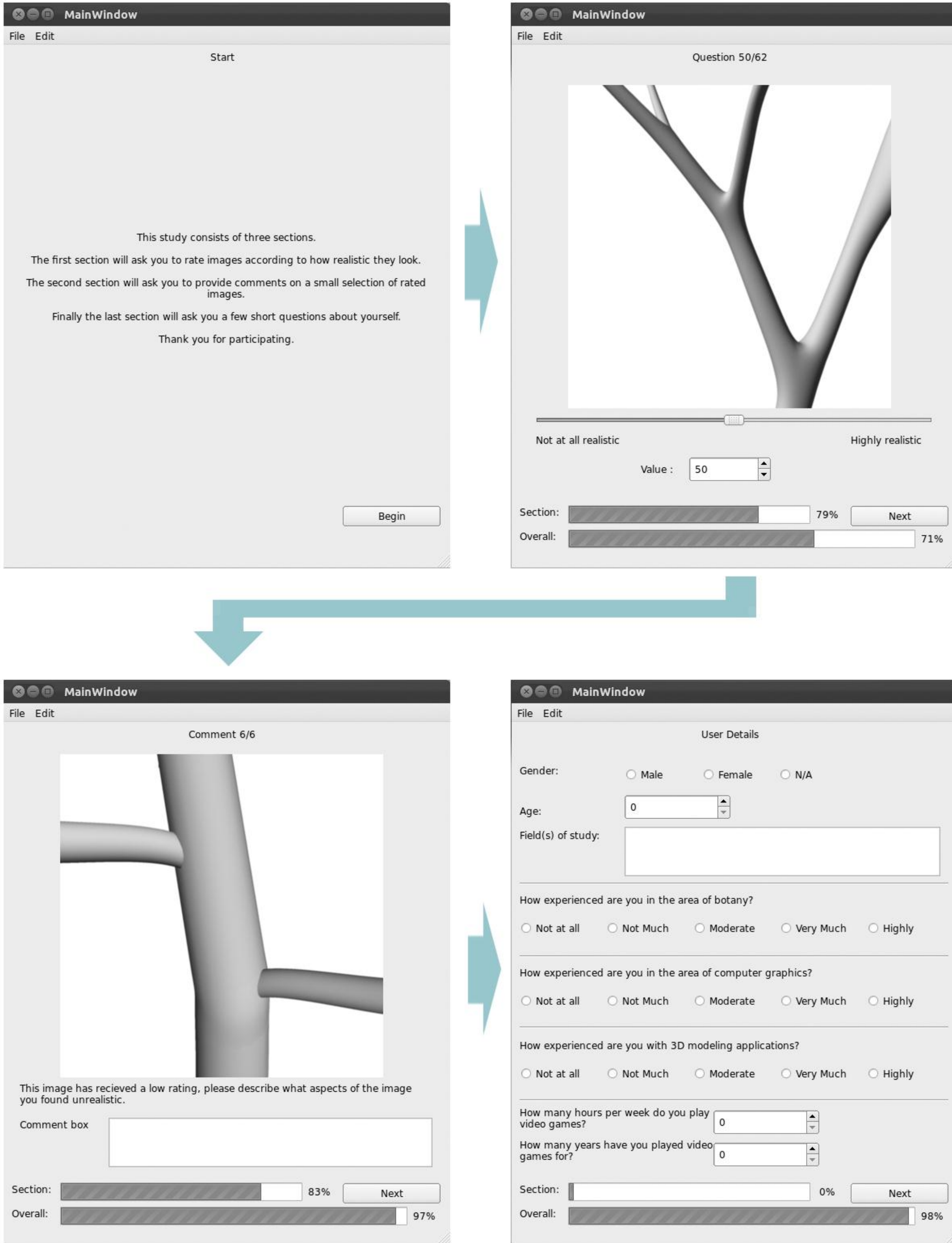


Figure 5.4: The test interface developed to capture participant data

To avoid scaling artefacts, all images were created and presented at a fixed resolution of 400 by 400 pixels which captured adequate detail. The models were rendered at this resolution with 16x anti-aliasing. Once a user had assigned a score to every image, they were asked to write a short comment on the two images from each section that received the lowest scores. More specifically, users were asked to comment on why they found the branches depicted in the images so unrealistic. These comments revealed qualitative insights into the score distributions as well as flaws in the experiment methodology. These are discussed in section 5.2.3.

To find out more about our test participants the final section of the interface consisted of a short survey requesting some demographic information. This data was not captured with the intention of forming correlations, but rather finding out if the test sample was a fair cross section of the population. The data captured pertained to gender, age, field of study, experience in botany, experience in 3D graphics, and finally video gaming experience. Experience in botany and graphics were measured on a five point Likert scale. The intervals used were “Not at all”, “Not Much”, “Moderate”, “Very much”, and “High”. However, it was felt that gaming experience would be more accurately measured in years.

Image Preparation

The preparation of the images presented to participants is an important aspect of the methodology. This is due to the fact that all images were prepared by the experimenter. This creates the potential for experimenter bias. As mentioned, the images can be classified into two sets: Shaded and Silhouetted. There were three sources of silhouettes. These were renders of models from TreeDraw, renders of models from Yggdrasil and segmented photographs of real trees. The reasoning behind asking users to score the realism of silhouettes rather than regular images is that it allows the models to be compared to real trees purely on the basis of their branch structure. Other confounding variables, such as bark, are removed. The silhouettes were created for the models by ray tracing them without shading in Softimage. All silhouettes were black against a white background. Constructing silhouettes from real images of trees proved more challenging. Images were sourced from photographs of trees taken for the purposes of this experiment. The trees chosen for segmentation exhibited smooth curves at the joints similar to those produced by subdivision surfaces. Segmenting the images algorithmically would have been the most time efficient and unbiased way of extracting the branches. Unfortunately this was not possible due to the high level of noise present in the image background. Instead the images were segmented by hand in Photoshop. During this process the rough texture of the bark was removed to produce smooth silhouettes similar to those from the renders. This loss in texture was deemed necessary since the focus of the experiment was on structure. After segmentation a threshold was applied to the image to produce a silhouette. The potential for experimenter bias in both the photograph selection and segmentation reduces the internal validity of the experiment.

Besides the silhouettes, shaded renders were also produced for both the previous models and the new models. These were also ray traced in Softimage. However, this time using a diffuse shader and a single directional light. The models produced by the L-System compiler vary considerably in realism. For both Yggdrasil and TreeDraw, the models that appeared most realistic to the experimenter were chosen. The angle at which the models were framed for rendering varied from model to model; however, the shot was always focused on a characteristic feature. Examples of images appear in Appendix E.

Experimental Procedure

In order to conduct the experiment ethical clearance was obtained from the Science Faculty Ethics in Research Committee. The certificate of clearance appears in appendix C. No prior knowledge, experience or skills were required of the participants who took part in the study. The interface was kept as simple as possible so that minimal computer literacy was necessary. Participants were also encouraged to ask questions if they were unsure of the test interface at any point. The experiment population was drawn from the University of Cape Town (UCT) student body. Convenience sampling was employed and participants were recruited through posters placed around the upper campus of UCT a week before the study began. These posters advertised the experiment, but did not provide many details, only that it was in relation to procedural tree generation and that participants would be paid R30. To appeal to as diverse a range of students, posters were placed on the notice boards of the Science, Humanities, Commerce, and Engineering faculties. There were no issues filling all of the testing slots. Participants were accepted on a first-come-first-served fashion, which avoided selection bias. A pilot study was run with three participants to rehearse the testing procedure, check the stability of the test interface and ensure the integrity of the data captured. Two issues arose during the pilot. The interface crashed once and it was discovered that only the first word of every comment was being saved. These issues were quickly identified and corrected. No other issues occurred during subsequent tests. The data obtained during pilot tests was excluded from the final statistical analysis.

The venue selected for testing was the Macintosh Lab, which resides on the third floor of the UCT Computer Science building. Participants were guided to the venue through a series of signs. There were no reports of participants struggling to locate the venue. This venue provided a quiet, insulated environment that could be easily regulated. Air conditioning ensured a regular room temperature. Three identical work stations were set up allowing three participants to be tested per time slot. All work stations were employed, running version 11.10 of the Ubuntu operating system. Additionally all workstations had identical monitors. This ensured a consistent presentation quality for the images. Unfortunately, the workstations were not isolated from one another. Participants could easily observe their neighbour's screens should they so desire. This introduced a minor confounding variable in that participant's scores may be biased by seeing the scores assigned by their neighbours.

Although the testing time slots were scheduled for 40 minute the average time taken for the test to be completed was 15 minutes. These slots were evenly distributed over the course of a week. While it is likely that many participants may not have been first language English speakers, the language used was kept simple and questions were encouraged. To maintain consistency across all of the test sessions and between test invigilators, a script was drawn up and used as a guideline. Each session began with an experimenter briefing the participant. During this briefing the term procedural generation was defined and participants were told that the images they would be shown would be of either bark or models. No indication was made as to the source of the models. Participants were then assured that all data was to be captured anonymously and told that it would be included in this report. Before beginning the test two forms of informed consent were signed both by the researcher and the participant. One was given to the participant and the other was kept by the researcher. Upon test completion participants were debriefed and asked if they had any further questions. Finally, they were paid, for which their name, student number and signature was required as proof of payment.

5.2.2 Results

Over the course of a week, a total of 39 participants took part in the evaluation. This included three pilot participants whose data was excluded from the final analysis. Of the 36 participants whose data was included, one participant did not fill out the demographic form. The gender ratio of the 35 that did is rather skewed, 9 were male and 26 were female. The ages of the participants ranged from 18 to 23. The mean age was 20.5 years, with a standard deviation of 1.9 years. As hoped, participants came from a diverse range of studies, broken down into faculties there were 14 from Science, 7 from Humanities, 5 from commerce, 3 from Law, and 4 from Engineering. Since both shaded and silhouetted images were shown to participants, two separate sets of data were acquired.

Accounting for Anomalies

To achieve normality in the distributions, anomalous results were removed from both the shaded results and the silhouetted results. Two anomalous scoring patterns were identified. In the first pattern, participants assigned a score of 50 to over a third of all shaded renders. A potential explanation for this pattern can be found in the test interface design. To allow participants to easily assign the scores, a slider control was implemented. Whenever the participant were presented with new image, the slider was set to a default position of 50. The reasoning behind this was that it would avoid biasing the results in either direction. The slider in question can be seen in Figure 5.4. An alternative explanation for the occurrence of this pattern is that the participants misunderstood the experiment or the scoring scale. A possible third explanation is a malfunction in the testing interface. To account for this anomaly two sets of results were excluded from the shaded image analysis, and one participants results were excluded from the silhouetted image analysis.

The second anomaly that occurred was participants who assigned a score of 0 to multiple images. One participant went so far as to assign a total of 8 out of 20 shaded images a score of 0. This is unusual because 0 was defined as “not at all realistic”. Theoretically none of the branches presented should receive a score of zero as they all have some tree-like qualities. This imposes a lower bound on the data captured that lead to a skewed distribution. These anomalous scoring patterns indicate that these participants did not fully understand the scale, and that experimenters should have been more explicit in their explanation. To account for this three participants, who assigned a score of zero to more than three images, were excluded from the shaded image analysis, and one from the silhouetted image analysis.

The data of one more participant was rejected from the shaded image results based on their comments. This participant assigned a score of 0 to the first two images presented, and then commented that the appearance of the bark was too plastic. Since all of the models were rendered without texture, the participant clearly misunderstood the aspects that they were supposed to be considering.

In total of the data six participants was excluded from the shaded image analysis and two from the silhouetted image analysis. This left 30 data sets for the shaded images and 34 data sets for the silhouetted images. All distributions were tested for normality by constructing normal quartile plots and performing Shapiro-Wilks [50] normality tests. The distributions were all found to be normal at a significance level of p = 0.01.

Analysis of Results

In the experimental method, two null hypotheses were postulated.

1) H_0: The score distribution for models generated by Yggdrasil is the same as the score distribution for models generated by TreeDraw.

2) H_0: The score distribution for silhouettes of models generated by Yggdrasil is the same as the score distribution for silhouettes of real trees.

An ideal result would be to reject the first null hypothesis and accept the second null hypothesis. On rejecting the first hypothesis we hope to find that the distribution for Yggdrasil’s models is significantly higher than the distribution for TreeDraw models. This would imply that subdivision surfaces are a more realistic method of modelling branching than generalized cylinders.

To reject the first hypothesis the score distribution of TreeDraw and Yggdrasil are compared for both the shaded and silhouetted images. On performing a two tailed paired t-tests for the shaded and silhouetted results we find that in both cases we can reject this hypothesis at the p = 0.02 significance level. The exact p-values of the t-tests are available in Table 5.1. By examining the mean it is possibly to infer that on average, participants found the models

produced by Yggdrasil more realistic than the models produced by TreeDraw. These means can also be found in Table 5.1.

Table 5.1: Top: Means and medians of the distributions. Bottom: Results of the t-tests performed on the various distributions

Means and Medians of the Score Distributions				
	Shaded		Silhouettes	
Distribution	Mean	Median	Mean	Median
TreeDraw	49.0	49.65	45.21	46.81
Yggdrasil	55.6	55.8	48.34	48.25
Real	N/A		62.41	64.44
Two Tailed Pair T-Test Results				
	Shaded	Silhouettes		
Distribution One	Yggdrasil	Yggdrasil	Yggdrasil	TreeDraw
Distribution Two	TreeDraw	TreeDraw	Photographs	Photographs
p-value	0.000158	0.017226	7.15294E-09	3.32342E-10

For the second hypothesis only the silhouetted results are analysed as the shaded renders did not include any real trees. On performing a two tailed paired t-test on score distribution of Yggdrasil, and the score distribution of segmented photographs, it was found that the null hypothesis must be rejected at a 0.01 level of significance. The means for the distributions indicate that the silhouettes of model produced by Yggdrasil are not as realistic as silhouettes of real trees.

The results show that participants found the silhouettes and renders of models produced by Yggdrasil more realistic than those of TreeDraw, however, less realistic than photographs of real trees. This was the anticipated result and is confirmed by the box and whisker plots below.

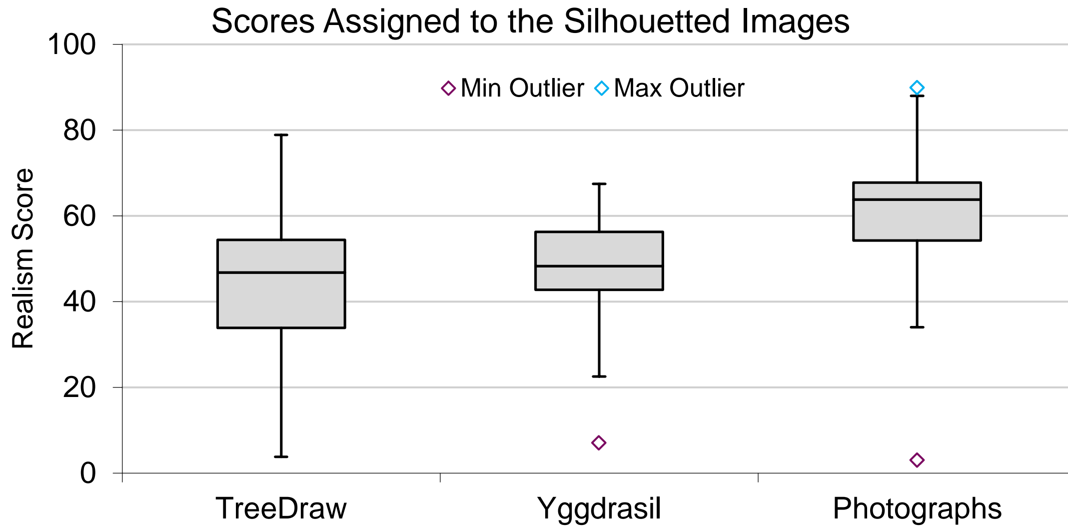


Figure 5.5: Box and Whisker plots for the three distributions obtained from the presenting silhouetted images to participants.

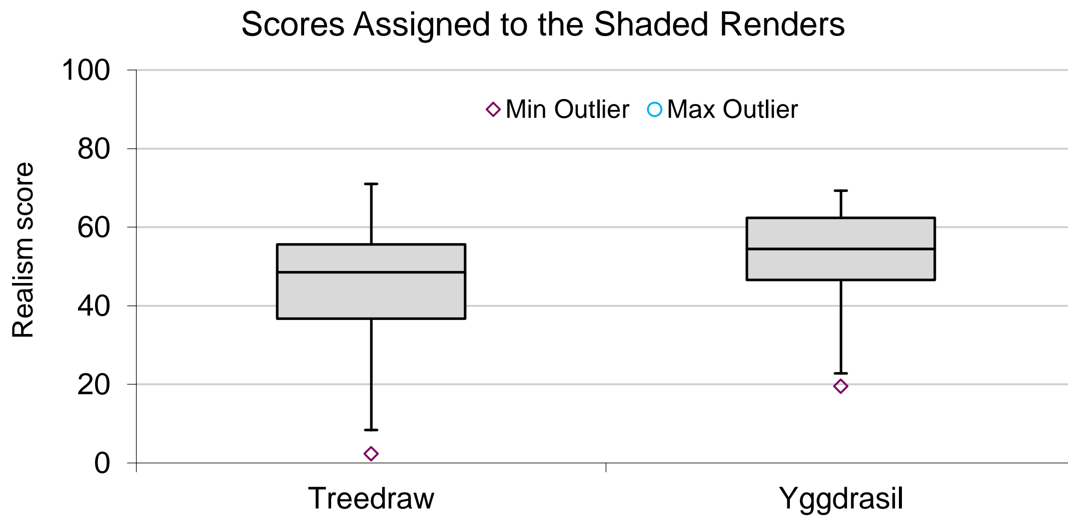


Figure 5.6: Box and Whisker plots for the three distributions obtained from the presenting shaded images to participants.

Qualitative Analysis

Besides the quantitative scores captured for each image, participants were also asked to write a short comment for the two images in each section that received the lowest rating. This revealed key insights into the basis on which participants were assigning scores. Many of the issue raised were to do with the construction of the model. Some commented that the

models were too perfect or that the branches were too straight. In regard to the joints many participants took issue with the abrupt blend in the models produced by TreeDraw, commenting that it looked as though the branches had been drilled into one another. Issues were also raised for models from Yggdrasil, a few comments referring to the joints as being too smooth or flat. In general, participants were bothered that the trunk was the same width as the branches. This was an error on the part of the experimenter; more time should have been allocated to constructing more realistic models with the TreeDraw sketch interface.

5.3 Performance Evaluation

To evaluate the scalability of the joint construction algorithms and Loop Subdivision, performance testing was conducted. The tests were executed on a Lenovo Y570 laptop running version 12.04 of the Ubuntu operating. The laptop consisted of an Intel Core i7 CPU 2.20 GHz with 8 logical cores and 8 gigabytes of DDR3 ram. The time taken for a function to complete was measured as the differences between the clock time at the start of the function and the clock time at the end of the function. The clock time was measured at millisecond granularity using the *gettimeofday()* function provided by "*system/time.h*" library.

The performance of the mesh generation process is directly related the complexity of the submitted graph. Two main factors influence the graph complexity. These are the number of branches and the degree of branching at the joints. Since the branch segments between joints are constructed independently of one another, an increase in the number of branches leads to a linear decrease in performance. A more interesting performance relationship exists based on the degree of branching. To understand this relationship better the execution time of the joint construction function was sampled for various branching degrees.

Although the Interim Core Scheme method was abandoned in favour of a convex hull method, its performance was nevertheless measured for comparison. The independent variable is time in milliseconds and the dependent variable is the number of branches for which a joint must be constructed. This number ranged between 2 and 200. To account for random fluctuations caused by the operating system the function was executed for the entire input range 50 consecutive times. At the end of the test every value of N had 50 associated time measurements, which were then averaged. The plotted measurements can be seen below in Figure 5.7. Both methods display a polynomial growth of order N^2, however, it is clear from the data that the convex hull method exhibits superior scaling.

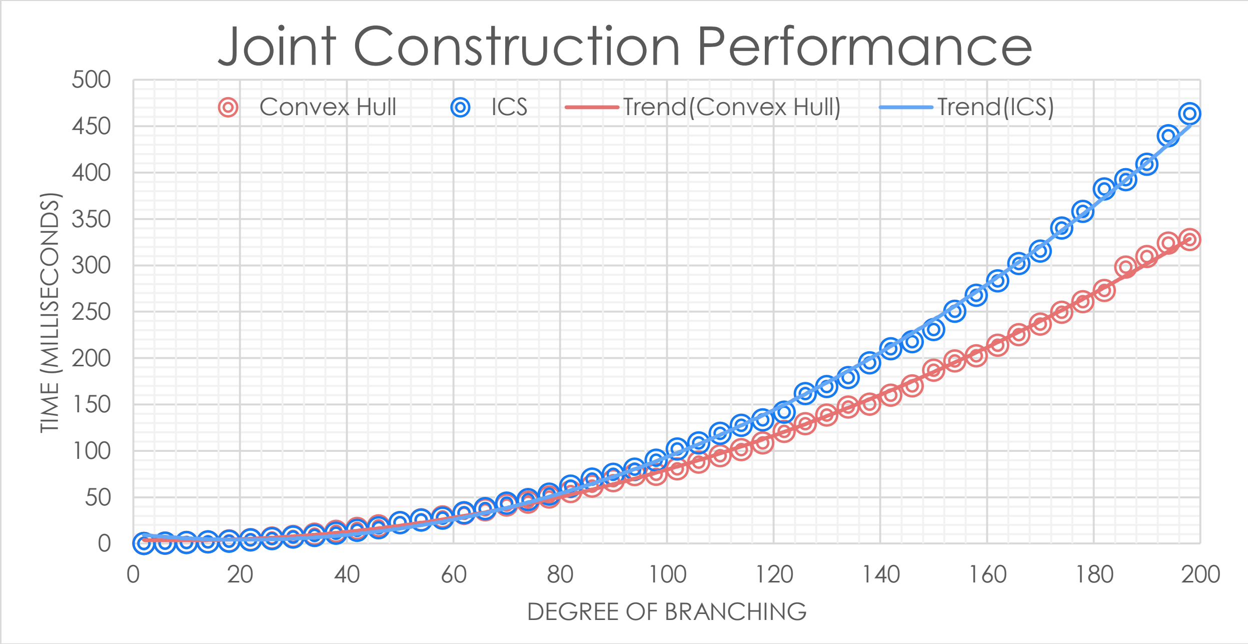


Figure 5.7: The performance of the joint construction algorithms in relation to the degree of branching

As a point of interest the performance of the loop subdivision implementation was also measured. Once again the dependent variable is time in milliseconds; however, the independent variable is now the number faces in the control mesh. The relationship, illustrated in figure 3, was found to be linear. This is to be expected as the algorithm subdivides the mesh on a per face basis.

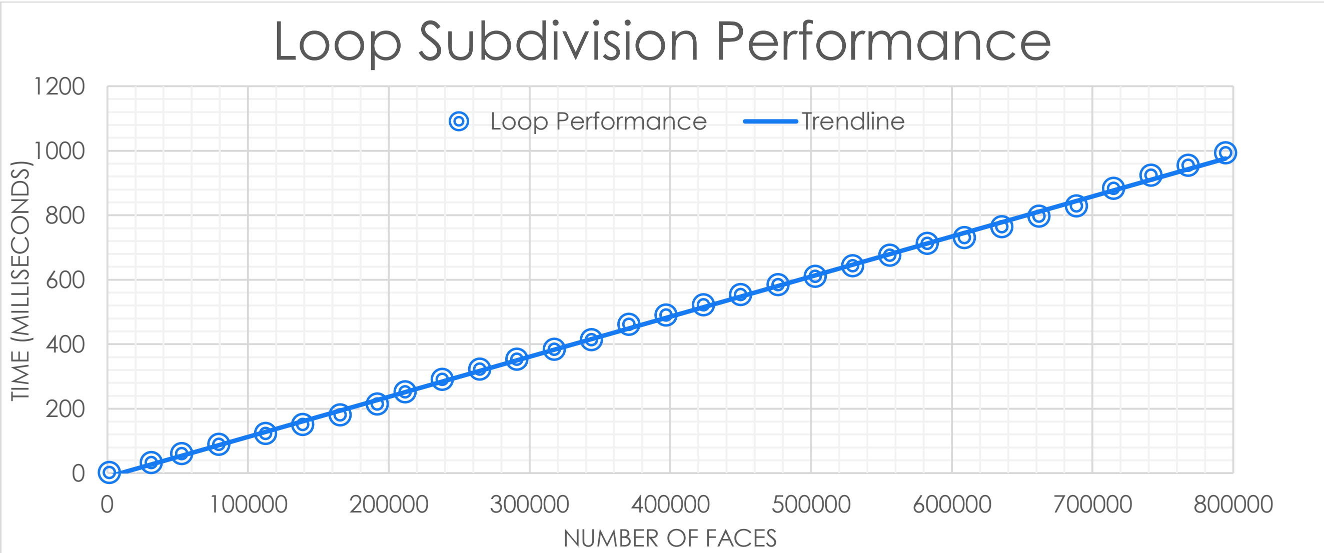


Figure 5.8: The performance of the Loop subdivision implementation in relation to the number of faces

The processes of mesh generation and subdivision both lend themselves particularly well to parallelization. In the case of mesh generation, all of the branch sections can be constructed in parallel, after which the joints can all be created in parallel. Since Subdivision is a local operation subdividing one face at a time it can also be trivially parallelized. Besides parallelization, other optimizations exist. For a start a more efficient convex hull algorithm could be implemented, such as Quickhull, which has a best case complexity of O(n*log(n)). To reduce the procedures memory a more weakly connected mesh data structure could be implemented. This comes with a performance penalty as more pointer traversals are needed to access the components. It also significantly increases implementation complexity.

5.4 System Limitations

Although the procedure presented in this report provides a robust solution to the problem of generating a mesh from a graph, it does have several shortcomings. Some of the issues, such as parameterization and Bezier curves can be resolved with more implementation time, while other issues such as graph simplification are inherent in the approach to joint construction.

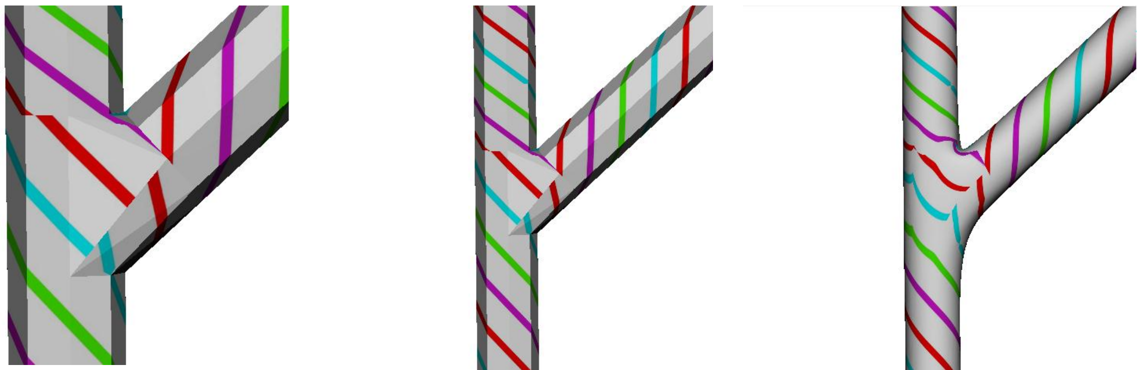


Figure 5.9: Left: an example of seams introduces at the joint. Middle: Parameterized control mesh. Right: Distortion caused by subdivision near seams

5.4.1 Parameterization

Parameterization in general is a difficult problem. Many modelling packages require the user to manually assign texture coordinates. Spheres in particular prove troublesome as there is no way to parameterize them without introducing stretching. The parameterizations produced by this system are reasonable and take into account striation direction, however, the seams introduced at the joints cause discontinuities that do not stand up to close inspection. A potential solution to creating a parameterization that appears seamless is Stripe Parameterization [44]. A more common approach, however, is to perform texture

synthesis across the surface of the mesh. The only prerequisite of this method is that the mesh has a one-to-one parameterization that minimizes stretching. There are many techniques to produce such a mapping over the generated meshes [51].

5.4.2 Subdivision of Texture Coordinates

As the mesh is subdivided, new vertices are created and old vertices are moved. To prevent the texture from stretching, shifting or distorting it is necessary to also move the texture coordinates associated with these vertices. The easiest way of doing this is to treat the vertices as existing in 5 dimensions, (x, y, z, u, v), where x, y and z refer to the vertices position in Euclidean space and u and v refer to the as associated texture coordinate in texture space. In this case, the subdivision rules can simply occur in 5-space. Unfortunately, due the presence of seams, not every vertex is associated with exactly one texture coordinate. As a result of this, when a new edge vertex is created its textured coordinate is linearly interpolated between the texture coordinates at either end of the edge. This leads to distortion occurring near the joints, as seen in Figure 5.9.

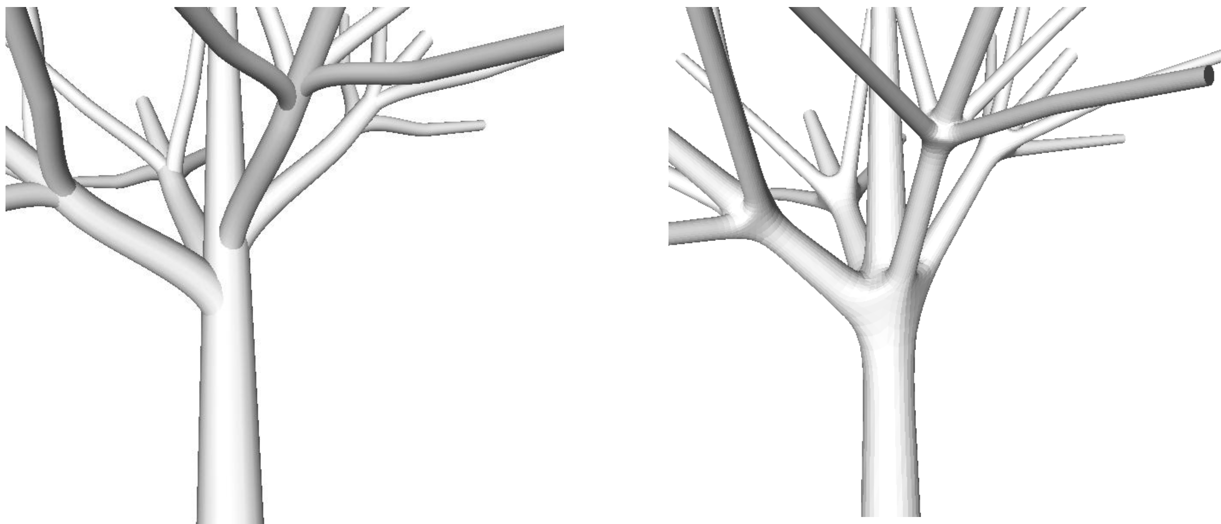


Figure 5.10: (Left) Model produced by TreeDraw. (Right) The same model produced by Yggdrasil displaying structural change due to the graph simplification step.

5.4.3 Graph Simplification

As described in the design chapter, a requirement of the joint construction algorithm is that no joints intersect. Unfortunately no such constraint is placed on the graphs produced by the L-system; because of this, an intermediate simplification step is needed to remove these intersections. These simplifications introduce structural changes in the graphs. Often the changes are negligible; however, there is the potential for degenerate cases. An example of a significant structural change is illustrated in the Figure 5.10 below. In this case an alternating branching pattern is simplified into an opposite branching pattern. This is an

inherent weakness of the joint-centric approach. A potential solution to generating a mesh for an arbitrary graph is constructive solid geometry (CSG), where two intersecting meshes can be joined with a union operation. However, no literature was encountered that could confirm this.

5.4.4 Bezier Curves

This research project is focused on whether subdivision surfaces can be used to more realistically model the branching points of a tree. The findings of the participant study indicate that the realism, in fact, improved. However, due to the nature of the experiment this does not imply that the overall model is more realistic. In the previous system, the graph produced is not directly interpreted; instead it was converted to a set of Bezier curves. These curves gave the braches a natural look. Due to time constraints, Bezier curves were not implemented in this project. This arguably causes a significant loss in realism.

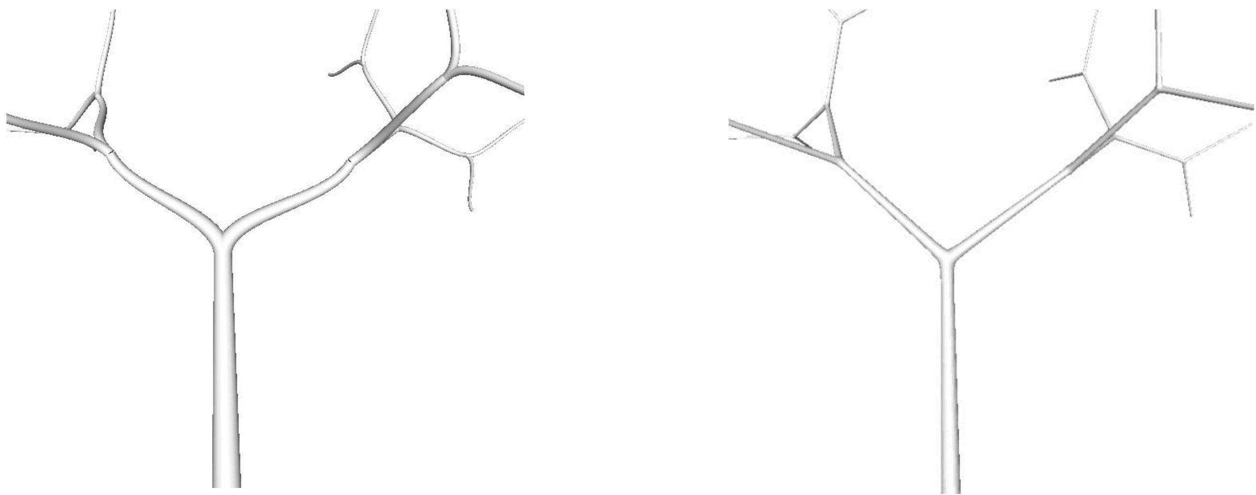
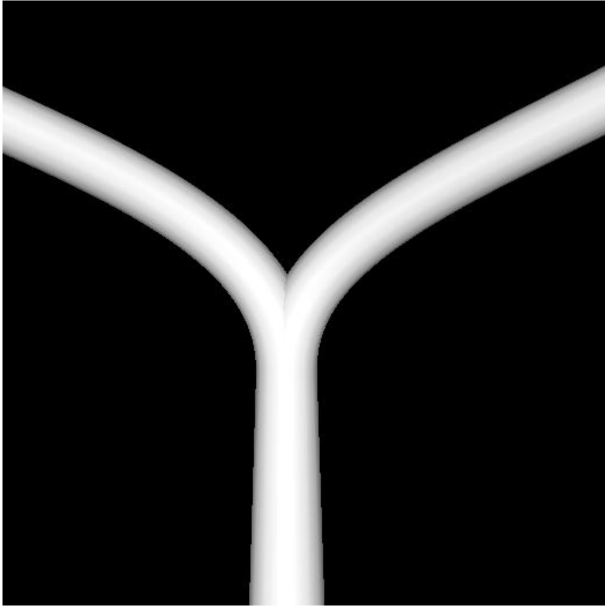


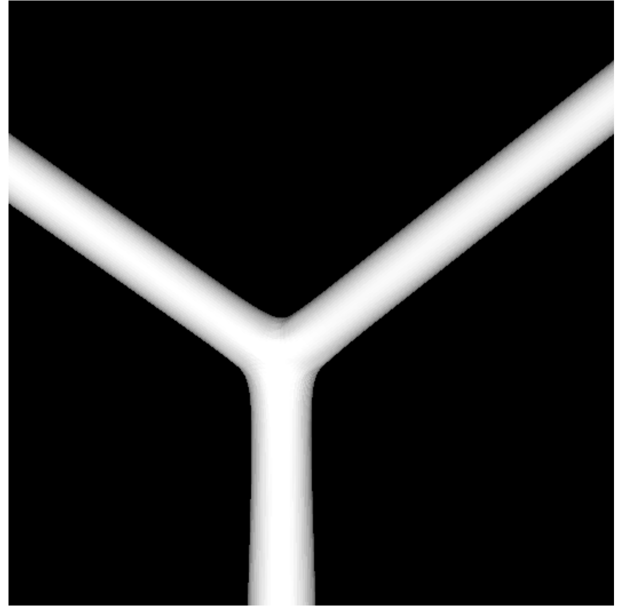
Figure 5.11: (Left): A tree produced by the previous system modelled with Bezier curves. (Right)A tree produced by this project without Bezier curves.

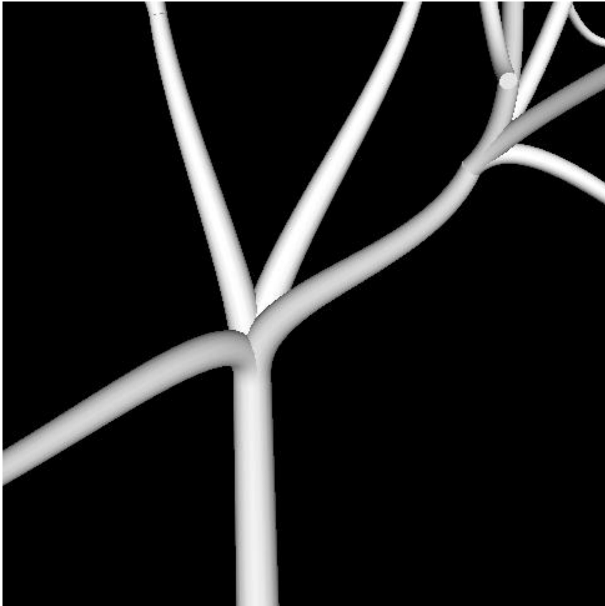
5.5 Example Models and Degenerate Cases

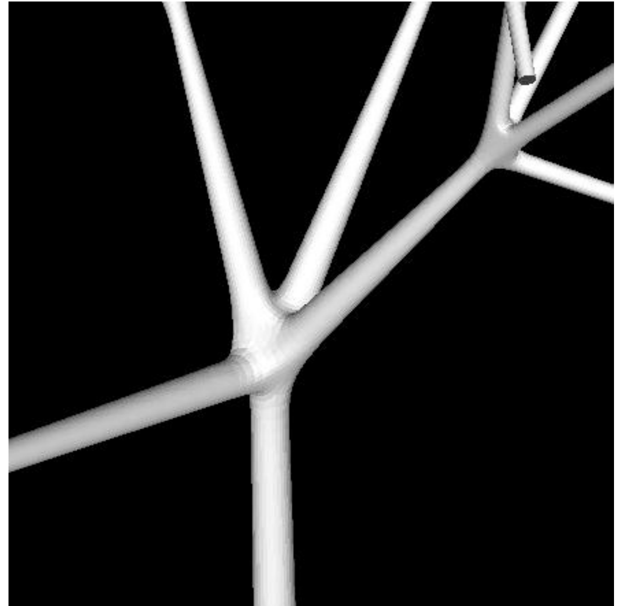
Generilzed Cyilnder Models (TreeDraw)

Subdivision Surfaces (Yggdrasil)



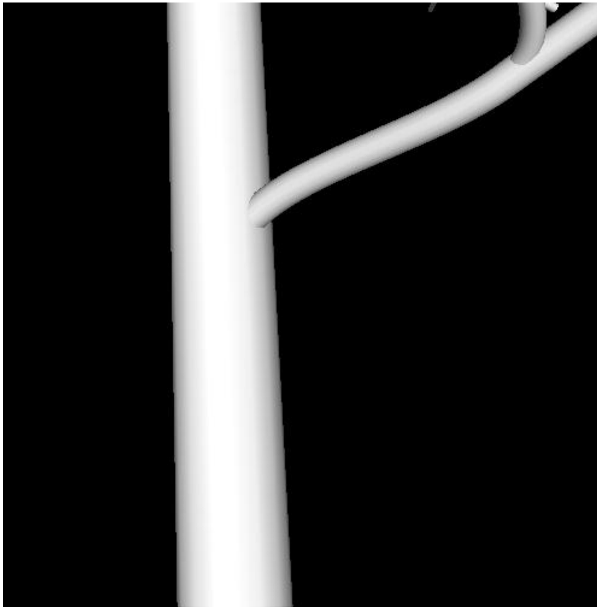


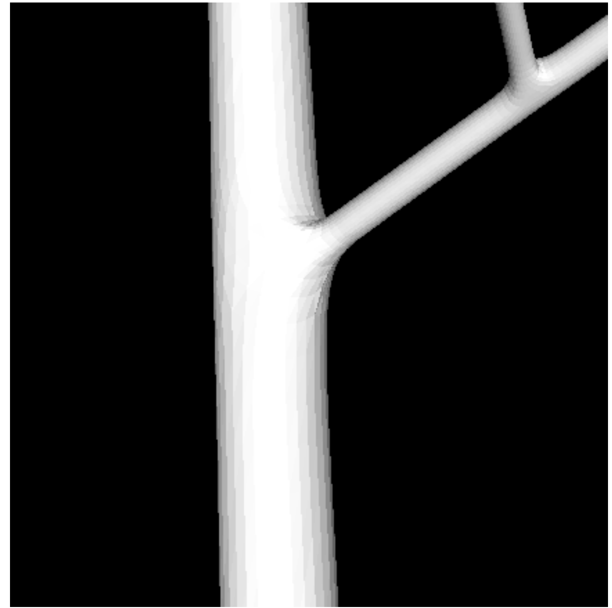




Generilzed Cyilnder Models (TreeDraw)

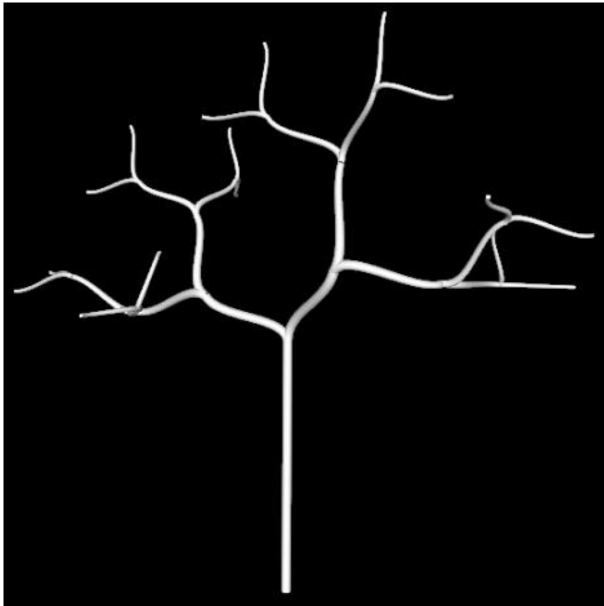
Subdivision Surfaces (Yggdrasil)

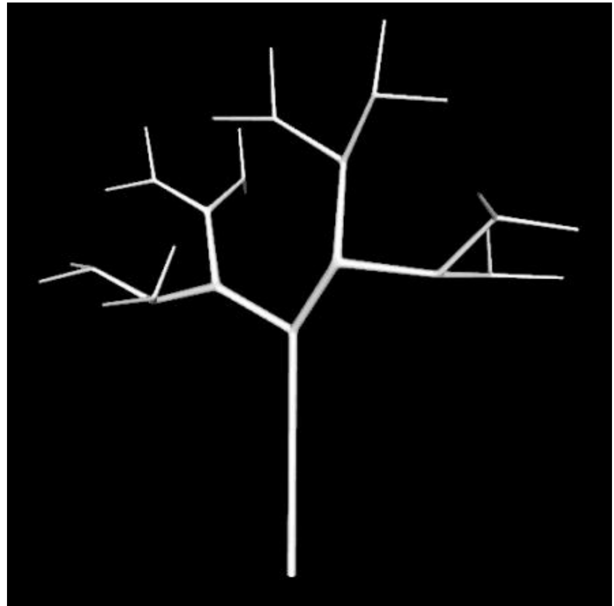


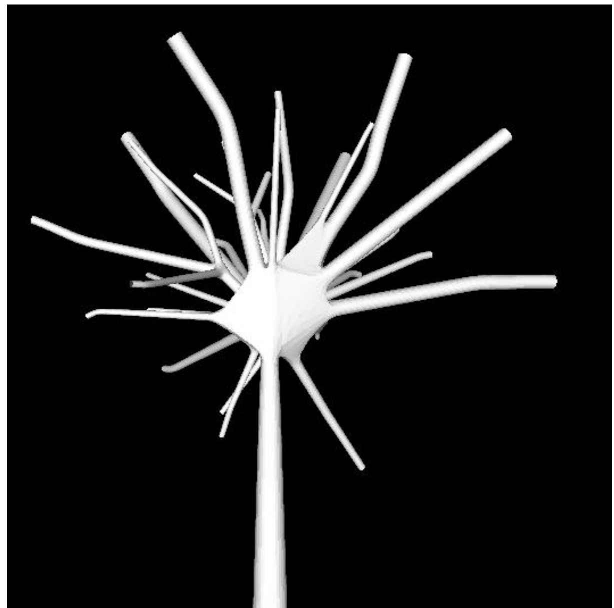


Generilzed Cyilnder Models (TreeDraw)

Subdivision Surfaces (Yggdrasil)

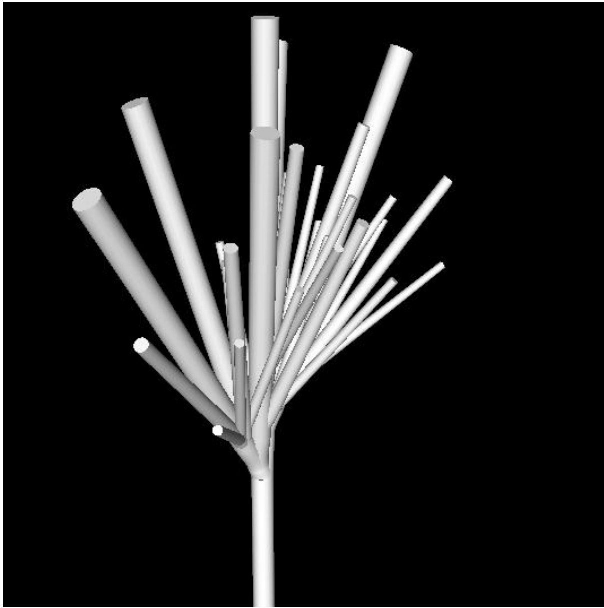


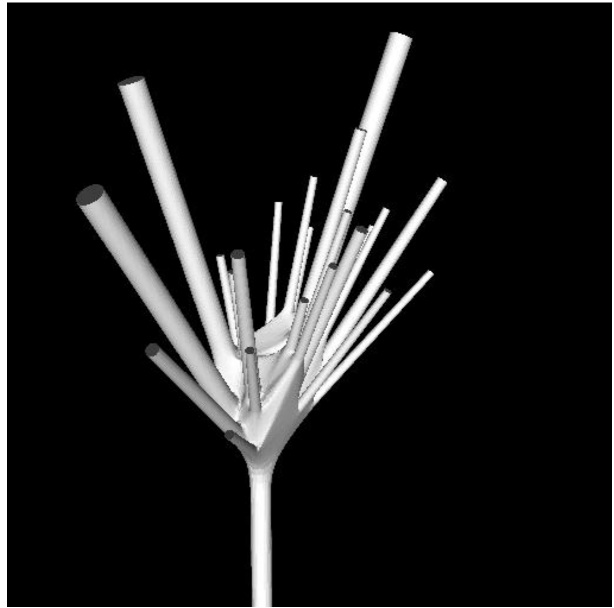


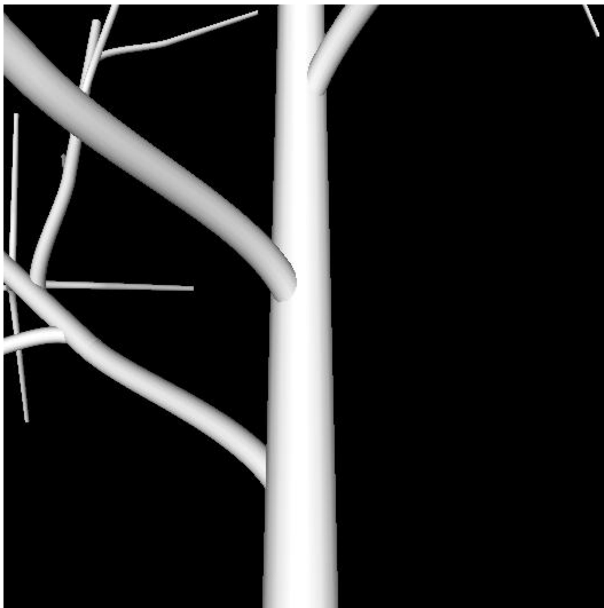


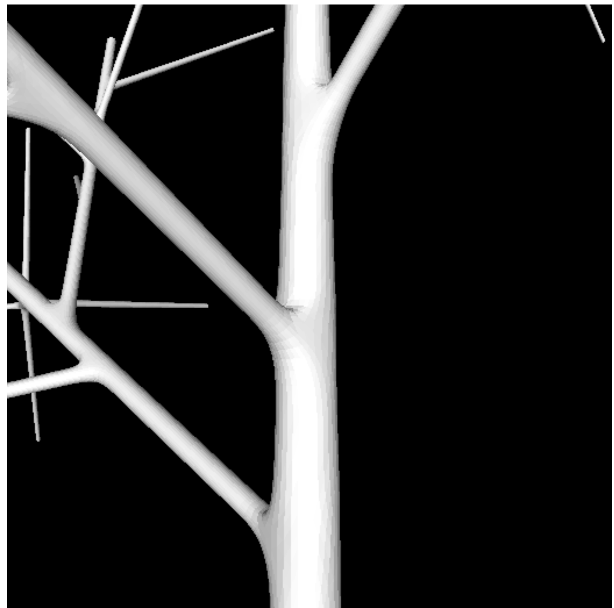
Generilzed Cyilnder Models (TreeDraw)

Subdivision Surfaces (Yggdrasil)









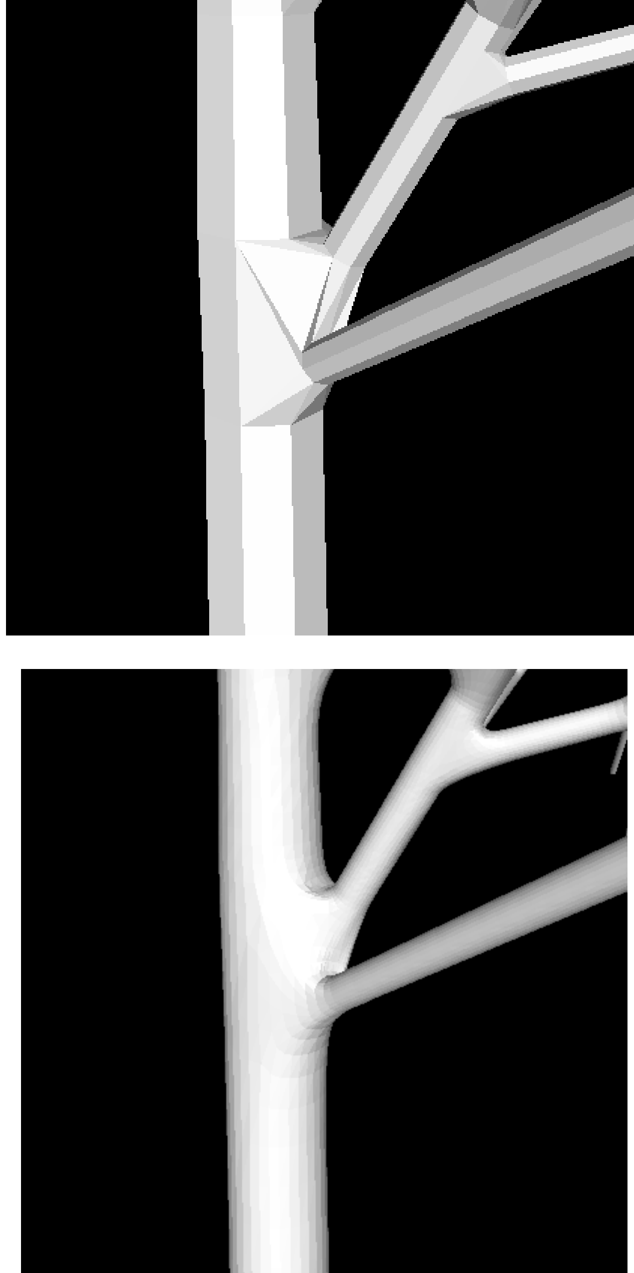


Figure 5.12: The figures above illustrate how subdivision surfaces smooth out irregularities that occur in the control mesh

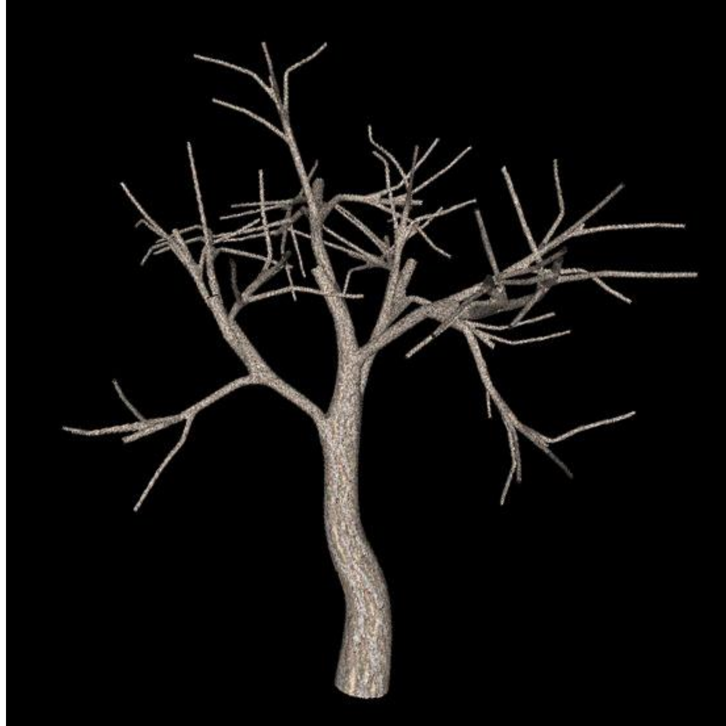


Figure 5.13: The images above depict a textured subdivision surface model. The model in the top image indicates that it is possible to generate curving branches despite the fact that Bezier Curves were not implemented. To achieve this, the curves must be manually drawn in the sketch interface.

Chapter 6

Conclusion

Procedural content is crucial to the development of computer generated productions such as films and video games. The procedurally generation of trees is particularly useful for creation of large outdoor render environments. A common approach to modelling a tree is to construct each branch individually, and then overlap them to create the illusion of connectivity; however, this fails to capture the smooth natural blends at points of furcation.

It was postulated that this issue could be resolved by modelling the tree as a subdivision surface. To verify this theory, a procedure was developed which generates a single manifold mesh from an L-System graph. This mesh is then converted into a subdivision surface though Loop subdivision. This procedure was then integrated into an existing procedural tree generator, which originally modelled each limb of the tree individually as a generalized cylinder. To find out whether the subdivision surface models were more realistic than those constructed from generalized cylinders, an experimental study was conducted. In this study participants were asked to rate the realism of the two model types. The results of the experiment indicate that the participants found the subdivision surfaces more realistic, confirming the hypothesis.

From our experience it is clear that subdivision surfaces are a powerful tool for the modelling of trees. However, the surfaces produced are only as good as their underlying control mesh. Although subdivision surfaces produce more realistic models, the models generated procedurally by the system are often far from realistic. This in part due to the L-System compiler, which occasionally produces an awkward skeleton, however, it is also a consequence of shortcomings in the mesh generation algorithm. A great deal of trial and error was required in designing this algorithm. Constructing the mesh at the branching points proved particularly difficult. In the end, this was solved by constructing a convex hull around the ends of the connected branches. Although this method produces pleasing results for low branching degrees, for high branching degrees (greater than six) the results are often quite unnatural. This is depicted in graphic detail by the degenerate cases presented in the previous section. Another caveat to the method is that the graph has to

be simplified before the mesh can be generated. This is an inherent requirement of the mesh generation algorithm and to overcome it a different avenue of mesh generation must be pursued. Despite these shortcomings, the models produced are reasonable. Furthermore, the development of this procedure has ruled out many approaches that do not work and marks another step towards the procedural generation of more realistic trees.

6.1 Future Work

6.1.1 Avoiding simplification

The method of mesh construction presented requires that no joints overlap. To ensure this condition a graph simplification step is implemented which merges overlapping joints. This approach leads to undesirable structural changes. A better approach to mesh generation would be to add very branch to the model one at a time in an arbitrary order, and avoid structural changes entirely. A field of research that could potentially handle such an approach is constructive solid geometry (CSG). CSG allows operations from set theory to be applied to arbitrary meshes. The branches could then be connected through a series of union operations. A drawback CSG, is its tendency to create poor topology at the points where the union occurred. A solution to this is to perform re-meshing after all of branches have been joined. However, no literature was encountered that could confirm this as viable approach.

6.1.2 Displacement maps

Displacement maps add extra detail to a model by offsetting the positions of the actual geometry by a distance stored in a height map. This is in contrast to bump and normal maps which simply create the illusion of detail perturbing the surface normals during the lighting calculations. For displacement maps to be truly effective, a highly tessellated mesh is required. As such subdivision surfaces and displacement maps often go hand in hand. With every subdivision of the mesh, the vertices are updated to better approximate the detail in the displacement map. The finer the subdivision, the better the approximation. A displacement map could be used to great effect in capturing the rough texture and cracks in bark, and unlike with bump and normal maps, this detail will be present in the silhouettes of the model.

6.1.3 Bezier Curves

The input to the mesh generation procedure is a graph representing a tree. The original model generator first converts this graph into a set of Bezier curves. This results in more natural looking limbs. Due to development time constraints, Bezier curves were not included

in our procedure, instead the graph is interpreted directly. This leads to unnaturally straight branches. Although the branches of these models blend naturally, from a distance, the models look arguably less realistic as a whole. By Incorporating Bezier curved our models would surpass the previous models in realism. Since the subdivision surfaces already approximate B-splines all that is required is to convert the branches in the graph into coarse approximations of a curve before submitting t to the mesh generator.

References

[1] P. Prusinksiewcz and A. Lindenmayer, The algorithmic beauty of plants, Springer-Verlag, 1990.

[2] M. Black, “A sketch-based interface for realistic tree generation with variation,” University of Cape Town, 2011.

[3] M. Danoher, “Honours Project Report: A sketch-based interface for tree modelling : 3D tree sketch interpretation and model generation,” University of Cape Town, 2011.

[4] N. Goldberg, “Honours Project Report: A sketch-based interface for tree modelling : L-system parameterisation , generation , and compilation,” University of Cape Town, 2011.

[5] C. Loop, “Smooth Subdivision Surfaces Based on Triangles. Masters Thesis,” University of Utah, 1987.

[6] D. Foster, “Procedural leaf generation using biologically-motivated algorithms. Honours Thesis,” University of Cape Town, 2012.

[7] R. Mazzolini, “Using Texture Synthesis to Generate Bark Textures with Variation. Honours Thesis,” University of Cape Town, 2012.

[8] G. Arden, “Approximation Properties of Subdivision Surfaces. PhD thesis,” University of Washington., 2001.

[9] E. Catmull and J. Clark, “Recursively Generated B-Spline Surfaces on Arbitrary Topologically Generated Meshes,” *Computer-Aided Design,* vol. 10, no. 6, pp. 350-355, 1978.

[10] J. Hubbard and B. West, “Differential Equations: A Dynamical Systems Approach. Part II: Higher-Dimensional Systems,” *Texts in Applied Mathematics,* p. 204, 1995.

[11] C. Notes and D. Zorin, “Subdivision for Modeling and Animation,” in *IGGRAPH 98 Course Notes*, 1998.

[12] D. Doo and M. Sabin, “Behaviour of recursive division surfaces near extraordinary points,” *Computer Aided Design,* vol. 10, no. 6, p. 356–360, 1978.

[13] A. Levin, “Interpolating nets of curves by smooth subdivision surfaces,” in *SGGRAPH 99 Proc of Computer graphics and interactive techniques*, 1999.

[14] P. MacMurchy, “The use of subdivision surfaces in the modelling of plants. Dissertation,” University of Calgary, 2004.

[15] T. DeRose, K. Michael and T. Truong, “Subdivision Surfaces in Character Animation,” in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, New York, 1998.

[16] D. Piponi and G. Borshukov , “Seamless texture mapping of subdivision surfaces by model pelting and texture blending,” in *SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, New York, 2000.

[17] B. Burley and D. Lacewell, “Ptex: Per-Face Texture Mapping for Production Rendering,” *Computer Graphics Forum,* vol. 27, no. 4, pp. 1155-1164, 2008.

[18] A. H. Nasri, “Polyhedral subdivision methods for free-form surfaces,” *CM Transactions on Graphics,* vol. 6, no. 1, pp. 29-73, 1987.

[19] N. Dyn, J. A. Gregory and D. Levine, “A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control,” *ACM Transactions on Graphics,* vol. 9, no. 2, pp. 160 - 169, 1990.

[20] P. E. Oppenheimer, “Real Time Design and Animation of Fractal Plants and Trees,” *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques,* vol. 20, no. 4, pp. 55-64, 1986.

[21] S. Ou and H. Bin, “Subdivision method to create furcating object with multibranches,” *he Visual Computer: International Journal of Computer Graphics,* vol. 21, no. 3, pp. 170-187, 2005.

[22] S. Maierhofer , “Rule-Based Mesh Growing and Generalized Subdivision Meshes. PhD thesis,” Technische Universitaet Wien, 2002.

[23] N. Litke, A. Levin and P. Schröder , “Fitting subdivision surfaces,” in *12th IEEE Visualization*, 2001.

[24] H. Moreton and A. Lee, “Displaced subdivision surfaces,” in *Computer Graphics Proceedings, Annual Conference Series*, 2000.

[25] Y. Hijazi, D. Bechmann, D. Cazier, C. Kern and S. Thery, “Fully-automatic branching reconstruction algorithm: application to vascular trees,” in *Shape Modeling International Conference*, Strasbourg, France, 2010.

[26] X. Tian, G. Han, M. Chen and Z. Situ, “Skeleton-based surface reconstruction for visualizing Plant Roots,” in *International Conference of Artificial Reality and Telexistence.*, 2006.

[27] J. Bloomenthal, “Modeling the mighty maple,” in *In Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, New York, 1985.

[28] J. Bloomenthal, “Skeletal Design of Natural Forms. Ph.D. dissertation,” in *University of Calgary*, 1995.

[29] F. Boudon, A. Meyer and C. Godin, “Survey on computer representations of trees for realistic and efficient rendering,” LIRIS, Université Claude Bernard Lyon 1, 2006.

[30] G. Turk and J. O’Brien, “Modelling with Implicit Surfaces that Interpolate,” *ACM Transactions on Graphics,* vol. 21, no. 4, pp. 855 - 873, 2002.

[31] J. Bloomenthal, “Bulge elimination in implicit surface blends,” in *Implicit '95--The First Eurographics Workshop on Implicit Surfaces*, 1995.

[32] C. Galbraith, P. MacMurchy and B. Wyvill, “BlobTree Trees,” in *CGI '04 Proceedings of the Computer Graphics International*, Washington, 2004.

[33] X. Jin, J. Fen, J. Feng and Q. Peng, “Convolution surfaces for line skeletons with polynomial weight distributions,” *Journal of Graphics Tools,* vol. 6, no. 3, pp. 17-28 , 2001.

[34] J. Hart and B. Baker, “Skjermo, J., & Eidheim, O. C. (2005). Polygon Mesh Generation of Branching Structures, 750–759.,” in *Proceeding of Implicit Surfaces 96*, 1996.

[35] P. Felkel, A. L. Fuhrmann and R. Wegenkittl , “SMART–Surface Models from by-Axis-and-Radius-defined Tubes,” VRVis Research Center, Vienna, Austria, 2001.

[36] H. Delingette, “General Object Reconstruction Based on Simplex Meshes,” *International*

Journal of Computer Vision, vol. 32, no. 2, pp. 111 - 146 , 1999.

[37] O. Eidheim and J. Skjermo, "Polygon Mesh Generation of Branching Structures," in *SCIA'05 Proceedings of the 14th Scandinavian conference on Image Analysis*, Berlin, 2005.

[38] J. Richter, The notebooks of Leonardo da Vinc Vol. 1, Courier Dover Publications, 1970.

[39] N. C. Gabrielides, A. I. Ginnis and P. D. Kaklis, "Constructing Smooth Branching Surfaces from Cross Sections," in *ACM Symposium on Solid and Physical Modeling*, Cardiff, Wales, 2006.

[40] K. Jha, "Reconstruction of Branching Surface and Its Smoothness by Reversible Catmull-Clark Subdivision," in *ICCS 2009 Proceedings of the 9th International Conference on Computational Science*, Berlin, 2009.

[41] K. Jha, "Construction of branching surface from 2-D contours," in *International Journal of CAD/CAD*, Berlin, 2008.

[42] J. Burguet, "Reconstructing the three-dimensional surface of a branching and merging biological structure from a stack of coplanar contours," in *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium*, Jouy-en-Josas, France, 2011.

[43] J. Bærentzen, "Converting skeletal structures to quad dominant meshes," in *Shape Modeling International Conference*, 2012.

[44] F. Kalberer, M. Nieser and K. Polthier, "Stripe Parameterization of Tubular Surfaces," in *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*, Springer Berlin Heidelberg, 2011, pp. 13-26.

[45] F. Pellacini, "Subdivision Surfaces: Course Lecture Slides," http://www.cs.dartmouth.edu/~cs77/lectures/10_SubdivisionSurfaces.pdf., Accessed 8th october 2012..

[46] C. Bradford Barber, D. Dobkin and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software,* vol. 22, no. 4, pp. 469-483 , 1996 .

[47] K. L. Clarkson and P. W. Shor, "Applications of random sampling in computational geometry, II," in *SCG '88 Proceedings of the fourth annual symposium on Computational geometry*, New York, 1988.

[48] J. Han, K. Zhou, M. Gong, H. Bao, X. Zhang and B. Guo, "Fast example-based surface texture synthesis via discrete optimization," *The Visual Computer,* vol. 22, no. 9, pp. 918-925, 2006.

[49] F. Kälberer, M. Nieser and K. Polthier, "QuadCover - Surface Parameterization using Branched Coverings," *Computer Graphics Forum,* vol. 26, no. 3, pp. 375-284, 2007.

[50] S. S. Shapiro and M. B. Wilk, "An Analysis of Variance Test for Normality," *Biometrika,* vol. 52, no. 3/4, pp. 591-611, 1965.

[51] A. Sheffer, E. Praun and K. Rose, "Mesh Parameterization Methods and their Applications," *Foundations and Trends® in Computer Graphics and Vision,* vol. 2, no. 2, pp. 105 - 171, 2006 .

[52] M. Okabe, S. Owada and T. Igarashi, "Interactive Design of Botanical Trees using Freehand Sketches and Example-based Editing," *Computer Graphics Forum,* vol. 24, no. 3, pp. 487-496, 2005.

[53] J. Schweitzer, "Analysis and application of subdivision surfaces. PhD thesis," University of Washington, 1996.

Appendix

A An overview of the TreeDraw system

The mesh generating procedure described in this report integrates into an existing procedural tree generator called TreeDraw. TreeDraw is the end result of a previous research project conducted Danoher [3], Black [2]and Goldberg [4]. The system is designed with a sketch based interface that allows users to draw the skeleton of a tree from which an L-system is created. This L-system then generates the structure of a 3D tree which is modelled as a set of generalized cylinders. An important aspect of TreeDraw is its ability to encode variation specified in the sketch into the L-System. This allows multiple similar though distinct trees to be generated from a single sketch. The system was developed as several discrete components. These are outlined below.

Sketch Interface and Gesture Recognition

The sketch interface forms the front end of the system. Users sketch the general structure of a tree by drawing one branch at a time using the mouse. Each branch has several parameters including start radius, end radius and angle to the parent. Additionally, the user can specify a degree of variation for each of these parameters. Besides the trunk, all branches drawn must be parented to a previous branch. Branches drawn further away will snap to the closest branch. When the user is satisfied with their sketch, they submit it for generation. The trees sketched by the user are interpreted as axial trees [1, 2]. The parameters are extracted from the sketch and saved as an XML file. This XML file is submitted to the next component in the model generation pipeline, the 2D to 3D converter.

2D to 3D converter

Once the user has submitted the sketch, the first step is to transform it into a three dimensional representation. This is achieved by shifting the branches along the length of their parent and then rotating them about their parent. This spreads the child branches uniformly. The goal of this is to maximize the distance between branches. This models the botanical principle that states that trees grow in such a way that the distance between

branches is maximized [52]. The transformed sketch is once again saved as an XML file and submitted to the L-system generator.

L-System Generator and Compiler

The L-System generator parses the XML input from the 2D to 3D converter, normalizes it, and constructs a parameterized L-System [1]. This normalization is necessary to accommodate the idiosyncrasies in the way a user may draw a tree [4]. The production rules of the L-system are a predetermined set which are applied based on branching patterns in the input. The L-System generated is saved as an LPFG file which is in turn parsed by the L-System Compiler. The compiler interprets the L- system and constructs a C++ file. When the C++ file is compiled and executed a unique string representing a tree is produced. From this single L-system multiple tree models can be derived. Although the models produced appear similar to the users sketch, each one exhibits variation.

Tree Model Generator

The final stage in the original pipeline is creating the polygon model that will be displayed to the user. The model generator executes the compiled L-system to generate a string representing a tree. Every time the L-system is executed a new variation is produced. From this string a model representing the tree is constructed. The tree is modelled as a set, of intersecting generalized cylinders that follow Bezier curves. These cylinders are tessellated to capture smooth curves. The cylinders are parameterized so that a bark texture can be mapped to the surfaces. Finally, the model is submitted to an Opengl rendering environment which displays the final 3D model.

B Edge collapse algorithm

The heuristic for collapsing edges is simple: Always choose the shortest edge. To assist in this process the edges are stored in a priority queue based on their length. The algorithm is complete when the queue is empty. An edge cannot be collapsed if its vertices share the same initial boundary loop. This ensures that the joint maintains it overall structure since none of the edges that form the base of a branch are lost.

Every iteration begins by popping the first edge off the queue and examining it. Each vertex contains a list of boundary loops that it is part of. If neither of the edge's vertices share any boundary loops, then the edge is collapsed by merging the vertices. Collapsing an edge also implicitly collapses the faces adjacent to it, so they also need to be removed. All of the edges and faces that indexed the old vertices are updated to point at the new merged vertex. For every triangle that is collapsed there are now two edges that index the same vertices but different adjacent faces. These must be merged into one edge that points to both adjacent faces. The priority queue must be updated since the edge lengths have changed.

Vertices are merged by calculating a weighted average of their positions and appending their loop lists. The weight of a vertex is the number of merged vertices that it represents. All vertices start with a weight of 1. When two vertices with weights of 1 are merged the new vertex, A, will have a weight of 2. When that A is merged with vertex C of weight 1, the new vertex will have a weight of 3 and its position will be 2/3 A's position and 1/3 third C's position. This means you can merge three vertices in any order and always arrive at the same position. The collapse of three edges is illustrated in Figure B.1.

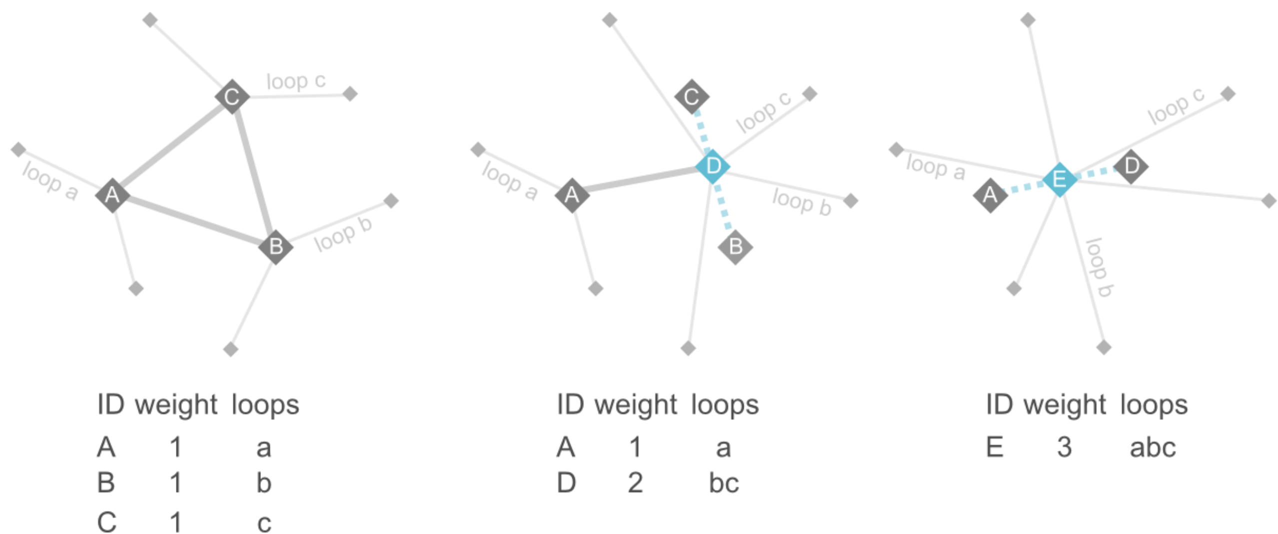


Figure B.1: Collapsing 3 edges. First edge CB is collapsed, forming the merged vertex D with the combined weight of C and B. Next, Edge AD is collapsed forming merged vertex E whose weight is the combination of A B and C.

C Ethical Clearance

	RESEARCH ACCESS TO STUDENTS	DSA 100

NOTES

1. This form must be FULLY completed by applicants that want to access UCT students for the purpose of research.
2. Return the completed application form together with your research proposal to: Moonira.Khan@uct.ac.za; or deliver to: Attention: Executive Director, Department of Student Affairs, North Lane, Steve Biko Students' Union, Room 7.22, Upper Campus, UCT.
3. The turnaround time for a reply is approximately 10 working days.
4. NB: It is the responsibility of the researcher/s to apply for and to obtain ethical clearance and access to staff and/or students, respectively to the (a) Faculty's 'Ethics in Research Committee' (EiRC) for ethics approval, and (b) Executive Director, HR for approval to access staff for research purposes and the (c) Executive Director, Student Affairs for approval to access students for research purposes.
5. For noting, a requirement of UCT (according to Senate policy) is that items (1) and (4) apply even if prior clearance has been obtained by the researcher/s from any other institution.

SECTION A: RESEARCH APPLICANT/S DETAILS

Position	Staff / Student No	Title and Name	Contact Details (Email / Cell / land line)
A.1 Student Number	FSTDON001	Mr. Donovan Foster	FSTDON001@myuct.ac.za Cell: 072 934 5664
A.2 Academic / PASS Staff No.			
A.3 Visiting Researcher ID No.			
A.4 University at which a student or employee	UCT	Address if *not* UCT:	
A.5 Faculty/ Department/School	Department of Computer Science		
A.6 APPLICANTS DETAILS If different from above	Title and Name	Tel.	Email

SECTION B: RESEARCHER/S SUPERVISOR/S DETAILS

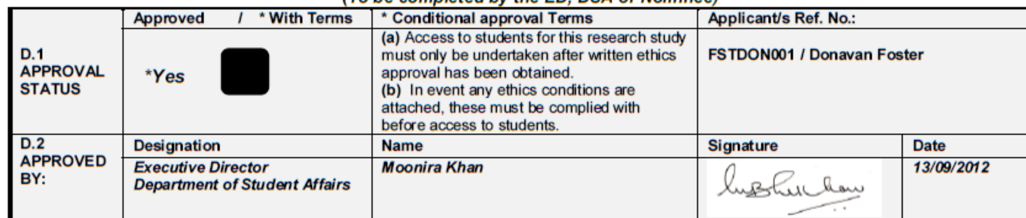
Position	Title and Name	Tel.	Email
B.1 Supervisor	A/Prof. James Gain	021 650 4058	jgain@cs.uct.ac.za
B.2 Co-Supervisor/s (a)			

SECTION C: APPLICANT'S RESEARCH STUDY FIELD AND APPROVAL STATUS

C.1 Degree (if a student)	BSc (Honours)
C.2 Research Project Title	Procedural Tree Generation Improvements
C.3 Research Proposal	Attached: Yes ■ No ☐
C.4 Target population	UCT registered students
C.5 Lead Researcher details	If different from applicant:
C6. Will use research assistant/s	Yes ☐ No ■
C.7 Research Methodology and Informed consent:	Research methodology: Scientific Method Informed consent will be obtained for participants. Participation is voluntary. Confidentiality, anonymity and privacy of participants, is assured by the researcher.
C.8 Ethics clearance status from UCT's Ethics in Research Committee (EiRC)	Approved by the EiRC: Yes ☐ No ☐ Awaiting response : ■ If yes, attach copy and state the date and ref. no of EiRC approval: Date of application if awaiting response: 07-09-2012

SECTION D: APPLICANT/S APPROVAL STATUS FOR ACCESS TO STUDENTS FOR RESEARCH PURPOSE

(To be completed by the ED, DSA or Nominee)



	Approved / * With Terms	* Conditional approval Terms	Applicant/s Ref. No.:	
D.1 APPROVAL STATUS	*Yes ■	(a) Access to students for this research study must only be undertaken after written ethics approval has been obtained. (b) In event any ethics conditions are attached, these must be complied with before access to students.	FSTDON001 / Donavan Foster	
D.2 APPROVED BY:	Designation	Name	Signature	Date
	Executive Director Department of Student Affairs	*Moonira Khan*	[signature]	*13/09/2012*

Department of Environmental and Geographical Science
University of Cape Town
RONDEBOSCH 7701
South Africa

e-mail: Michael.meadows@uct.ac.za
phone : + 27 21 650 2873
fax : +27 21 650 3791

27th September 2012

Mr Donavon Foster
Department of Computer Science
FSTDON001@myuct.ac.za

Dear Mr Foster

Procedural Tree Generation Improvements

I am pleased to inform you that, having scrutinized the details of your above-named applications for research ethics clearance, the Faculty of Science Research Ethics Committee has approved your proposal in terms of its attention to ethical principles. You must have formal agreement from the Department of Student Affairs in order to approach UCT students with the survey.

Your approval code for the project is: SFREC 42_2012

I wish you success in the work involved.

Yours sincerely

M Meadows

Michael E Meadows
Professor and Head of Department
Chair: Science Faculty Ethics in Research Committee

D Experiment Data

Table D.1: Motivations for data exclusion

Anomalous Data Excluded from Analysis	
	Participant assigned a score of 50 to more than a third of the images
	Participant assigned a score of 0 to more than a third of the images
	Comments indicate that the participant misunderstood the experiment

Table D.2 Scores Assigned by Participants to the Shaded Renders

Yggdrasil																		
Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
/a_ygg.png	70	40	88	19	26	59	50	40	31	50	32	29	33	17	0	50	50	23
/d_ygg.png	60	45	16	24	41	50	60	30	67	60	30	6	58	26	40	45	45	61
/e_ygg.png	70	50	70	69	68	34	74	50	56	65	60	34	43	30	30	53	50	58
/f_ygg.png	70	60	88	52	54	50	85	67	50	66	50	41	36	27	80	55	80	46
/h_ygg.png	70	50	80	46	60	30	64	50	40	44	14	29	35	18	30	56	45	30
/k_ygg.png	70	50	0	6	29	31	50	25	50	51	50	3	34	28	45	50	75	16
/n_ygg.PNG	60	60	62	25	31	50	56	50	30	41	60	9	39	25	50	45	45	23
Mean	67.5	50.6	59.1	35.9	43.6	44.3	61.1	44.3	46.8	54.0	39.0	19.5	39.6	23.9	36.9	51.3	56.3	35.9
Participants	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
/a_ygg.png	47	50	60	60	30	50	45	56	54	60	50	46	0	60	70	47	52	34
/d_ygg.png	55	37	55	30	53	70	56	63	87	40	50	64	0	40	60	56	64	37
/e_ygg.png	53	50	45	50	52	75	42	67	70	60	50	62	0	65	50	57	79	72
/f_ygg.png	52	55	50	65	42	63	58	62	62	60	60	65	25	70	50	47	39	44
/h_ygg.png	48	49	45	40	50	50	37	72	87	30	70	50	8	55	60	45	47	41
/k_ygg.png	59	50	45	35	55	54	65	38	76	70	70	64	9	50	40	27	33	39
/n_ygg.PNG	34	48	40	45	47	50	37	45	48	70	40	47	0	70	60	20	28	39
Mean	49.5	48.1	48.0	44.4	48.0	58.9	48.4	58.5	66.8	56.3	57.5	54.9	7.0	58.1	53.8	43.1	49.6	41.6
TreeDraw																		
Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
/a_td.png	60	70	68	17	51	44	60	36	20	55	37	38	33	25	50	50	61	25
/b_td.png	60	40	74	23	40	39	60	50	31	70	30	19	38	34	45	50	80	7
/c_td.png	70	50	81	32	58	14	61	35	30	67	15	21	36	3	40	50	55	31
/e_td.png	80	50	73	23	40	37	50	85	32	70	50	27	36	71	30	50	95	63
/k_td.png	60	40	5	19	11	20	50	55	30	70	10	18	21	42	50	45	45	24
/l_td.png	40	65	5	3	0	22	50	22	20	40	20	4	34	31	20	55	12	20
/r_td.png	40	50	6	15	43	19	80	40	30	55	30	6	35	14	35	45	70	25
/td_a.jpg	40	55	16	2	44	41	60	34	20	40	53	5	34	25	30	47	60	23
/td_b.jpg	50	50	70	2	7	50	49	15	55	80	10	13	37	30	70	50	60	0
/td_c.jpg	70	65	100	65	47	40	31	85	30	69	20	16	37	60	37	56	45	27
/td_d.png	70	40	98	24	34	7	70	30	40	85	10	13	74	27	70	55	60	40
Mean	58.2	52.3	54.2	20.5	34.1	30.3	56.5	44.3	30.7	63.7	25.9	16.4	37.7	32.9	43.4	50.3	58.5	25.9
Participants	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
/a_td.png	37	50	45	55	40	59	42	74	88	40	60	69	6	60	50	31	40	45
/b_td.png	48	48	45	45	58	60	36	72	77	70	50	56	0	40	50	39	55	37
/c_td.png	58	50	55	45	50	68	70	88	95	30	70	48	12	40	40	43	66	35
/e_td.png	38	60	72	47	45	55	42	61	94	40	70	61	0	40	50	44	44	29
/k_td.png	47	49	45	40	47	48	40	39	60	30	40	50	15	60	30	31	41	30
/l_td.png	39	39	40	20	52	69	43	55	56	30	50	44	3	40	40	18	21	29
/r_td.png	56	50	45	60	50	50	65	43	81	30	40	44	4	42	50	29	31	40
/td_a.jpg	21	49	53	45	50	47	51	59	52	60	50	47	0	30	40	33	47	16
/td_b.jpg	55	48	45	35	61	73	39	43	65	70	70	46	0	40	60	45	22	24
/td_c.jpg	56	50	47	50	50	58	54	45	100	40	50	67	2	75	50	45	67	38
/td_d.png	63	44	45	70	55	80	73	34	100	60	80	73	0	30	60	33	62	43
Mean	47.1	48.8	48.8	46.5	50.7	60.6	50.5	55.7	78.9	45.5	57.3	55.0	3.8	45.2	47.3	35.5	45.1	33.3
Real Trees																		
Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
/IMG_1295.JPG	60	40	92	58	84	29	80	60	61	40	60	22	50	53	20	50	90	64

/IMG_1298.JPG	70	65	100	83	86	28	78	65	80	54	40	31	44	40	50	50	91	22
/IMG_1299.JPG	60	60	78	52	66	12	71	73	50	80	60	37	63	50	50	50	60	73
/IMG_1305.JPG	70	60	62	48	64	50	73	60	50	60	50	17	70	70	10	40	98	80
/IMG_1329.JPG	70	70	92	86	62	56	88	76	80	83	50	37	64	61	80	55	65	76
/IMG_1333.JPG	60	63	71	66	53	42	66	93	80	75	50	7	70	25	35	50	81	64
/IMG_1343.JPG	40	70	29	58	59	50	92	50	60	70	33	17	80	65	60	50	95	80
/IMG_1346.JPG	60	70	100	78	61	0	61	100	50	80	83	23	81	70	70	50	78	66
/IMG_1351.JPG	80	60	100	62	54	32	88	78	80	84	60	33	67	35	70	59	91	74
Mean	63.3	62.0	80.4	65.7	65.4	33.2	77.4	72.8	65.7	69.6	54.0	24.9	65.4	52.1	49.4	50.4	83.2	66.6
Participants	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
/IMG_1295.JPG	61	70	70	75	56	49	46	73	83	40	70	68	0	60	60	55	52	23
/IMG_1298.JPG	66	63	47	55	50	57	72	75	64	60	90	70	4	75	40	43	55	26
/IMG_1299.JPG	65	58	47	65	50	65	65	64	94	80	50	59	2	70	50	61	70	36
/IMG_1305.JPG	52	67	84	70	50	70	52	67	96	60	80	50	0	50	40	53	66	15
/IMG_1329.JPG	71	68	53	50	48	80	73	75	100	80	90	75	3	75	40	59	74	36
/IMG_1333.JPG	52	57	47	60	60	59	65	80	90	30	80	73	10	60	40	57	74	51
/IMG_1343.JPG	58	68	90	40	53	72	73	77	96	70	70	66	0	60	40	65	52	17
/IMG_1346.JPG	64	44	50	40	60	75	70	71	100	70	100	86	5	75	60	75	74	67
/IMG_1351.JPG	61	68	50	75	62	60	62	80	86	50	90	77	3	80	40	60	65	51
Mean	61.1	62.6	59.8	58.9	54.3	65.2	64.2	73.6	89.9	60.0	80.0	69.3	3.0	67.2	45.6	58.7	64.7	35.8

Table D.3 Scores Assigned by Participants to the Shaded Renders

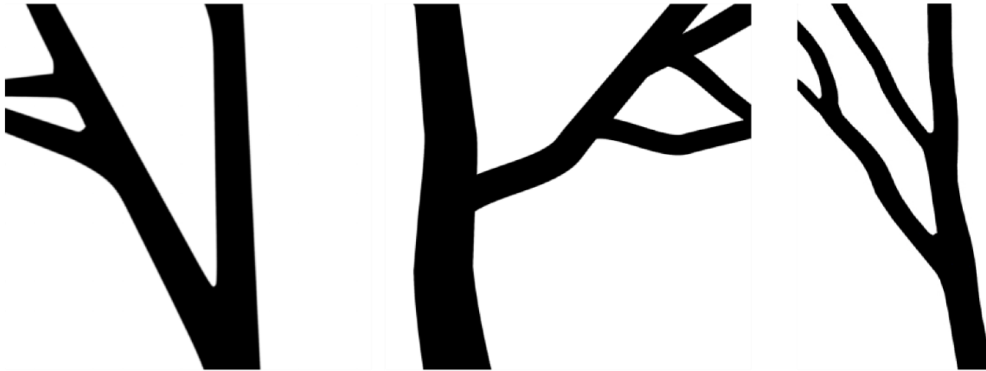
Yggdrasil																		
Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
/a_ygg_shaded.png	80	50	18	40	54	31	82	76	20	30	50	38	60	64	40	50	90	24
/b_ygg_shaded.png	60	60	3	50	76	20	55	55	50	70	3	31	72	33	30	58	65	36
/d_ygg_shaded.png	60	60	85	33	70	27	80	40	40	75	23	15	61	17	50	59	90	60
/e_ygg_shaded.png	70	60	97	69	47	4	69	73	50	55	21	47	80	29	30	50	70	38
/f_ygg_shaded.png	90	70	95	78	42	24	76	76	70	80	11	44	80	32	50	46	60	28
/h_ygg_shaded.png	80	50	82	52	44	0	70	80	60	53	17	52	70	72	0	50	55	36
/k_ygg_shaded.png	40	40	21	11	22	18	50	40	30	21	0	15	53	31	0	55	50	14
/n_ygg_shaded.png	80	60	1	54	48	14	60	87	50	44	24	39	70	32	20	55	60	41
/p_ygg_shaded.png	40	40	82	2	35	27	50	30	50	0	0	8	65	14	0	59	60	17
/q_ygg_shaded.png	90	60	70	75	66	30	60	85	70	70	56	54	82	100	70	63	55	82
Mean:	69	55	55	46	50	20	65	64	49	50	21	34	69	42	29	55	66	38
Participants	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
/a_ygg_shaded.png	48	48	40	75	49	46	47	67	48	50	70	50	0	70	50	49	25	0
/b_ygg_shaded.png	55	46	50	35	50	50	60	61	59	70	65	50	29	55	50	48	63	10
/d_ygg_shaded.png	56	48	58	50	53	71	67	70	47	40	50	55	41	70	60	40	66	0
/e_ygg_shaded.png	47	47	55	55	79	58	60	64	59	70	80	49	50	70	70	57	58	12
/f_ygg_shaded.png	55	48	50	55	45	63	58	84	58	80	70	74	50	71	80	74	66	50
/h_ygg_shaded.png	56	47	50	70	66	60	60	66	60	50	60	64	50	65	80	55	55	26
/k_ygg_shaded.png	33	36	44	40	48	50	60	47	39	40	60	48	13	55	50	19	32	13
/n_ygg_shaded.png	44	45	50	55	74	52	57	62	66	70	60	75	50	70	70	59	63	50
/p_ygg_shaded.png	44	48	50	30	81	50	50	61	26	40	40	46	50	40	60	26	32	21
/q_ygg_shaded.png	73	53	55	80	75	75	66	72	100	80	80	63	50	80	70	60	82	20
Mean:	51	47	50	55	62	58	59	65	56	59	64	57	38	65	64	49	54	20
TreeDraw																		
Participants	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18
/a_td_shaded.png	70	50	86	53	45	12	66	50	60	62	26	27	80	15	20	45	33	58
/b_td_shaded.png	40	50	100	39	44	10	54	94	50	31	12	8	70	40	40	50	58	24
/c_td_shaded.png	60	40	72	43	64	1	50	50	50	45	0	28	50	15	80	50	40	29
/e_td_shaded.png	80	60	66	61	55	0	59	77	60	20	50	48	60	72	10	45	60	55
/j_td_shaded.png	80	40	83	20	71	0	50	60	0	51	0	8	42	25	60	50	45	21
/k_td_shaded.png	70	60	53	44	45	0	50	68	50	69	7	42	53	20	10	60	45	38
/l_td_shaded.png	50	40	76	7	21	0	50	35	40	0	0	11	52	5	0	40	40	19
/q_td_shaded.png	70	30	15	42	52	0	23	9	0	60	0	2	55	18	0	60	20	22
/r__td_shaded.png	50	30	2	38	43	0	57	37	10	75	5	17	41	15	0	50	30	28

/td_d_shaded.png	70	50	93	35	42	0	40	87	0	30	0	17	46	50	60	50	40	29
Mean:	64	45	65	38	48	2.3	50	57	32	44	10	21	55	28	28	50	41	32

Participants	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
/a_td_shaded.png	74	50	54	45	66	52	56	66	74	60	60	63	50	50	70	48	63	40
/b_td_shaded.png	52	46	50	45	71	52	54	63	25	60	60	55	50	68	80	48	29	40
/c_td_shaded.png	92	48	55	55	87	69	55	70	34	60	60	69	50	55	70	16	69	4
/e_td_shaded.png	57	49	45	70	64	47	60	66	63	60	60	70	50	80	80	52	54	33
/j_td_shaded.png	33	48	39	20	50	50	71	47	36	60	50	46	50	55	60	12	68	2
/k_td_shaded.png	58	50	45	40	65	50	55	62	69	70	60	60	50	65	80	48	62	23
/l_td_shaded.png	48	36	45	40	62	60	46	44	47	60	30	50	35	20	40	8	9	6
/q_td_shaded.png	52	45	65	40	86	52	68	52	32	40	60	46	50	30	80	7	43	20
/r__td_shaded.png	66	62	47	35	79	60	55	57	26	40	50	50	50	44	80	40	65	40
/td_d_shaded.png	42	52	40	25	65	60	68	64	40	50	50	46	34	55	70	21	32	5
Mean:	57	49	49	42	70	55	59	59	45	56	54	56	47	52	71	30	49	21

E Test Image Examples

a. a render of a model produced by this project
b. a render of a model produced by the previous system



a. a silhouette of a model produced by this project
b. a silhouette of a model produced by the previous project
c. a silhouette of a real tree

a. an example of a silhouette cerated from for a real tree

Figure E.1: Some of the images presented to participants during the experimental study