Thomas Mcloughlin

Registration number 100203952

2021

GPU Accelerated Method for Constructing and Rendering Trees

Supervised by Dr. Stephen Laycock

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The inclusion of trees in virtual environments when trying to represent nature is essential and the branching structure of trees is vital is representing many other types of foliage. This project covers the research of related work involved with algorithmic generation of trees and the mathematical basis for the discussed approaches. The underlying principles relating to the algorithmic production of fractal patterns helpfully cooincides with the requirements for producing believable branching structures.
The project aim is to convert and extend the Lindenmayer-system based tree construction method presented by (Prusinkiewicz et al., 1996) to be used as an independent OpenGL module, the intuitive parametric L-system model was chosen because of its ability to produce diverse results with varying input. This module allows the addition of trees to a real-time environment with minimal user interaction, avoiding the difficulties and expenses of manually producing tree models.

Acknowledgements

I'd like to thank my supervisor, Stephen Laycock for his brilliant help and support throughout this project. Thank you to George Smith and Harry Tucker for their interest in my work and their support during the COVID-19 lockdowns.

Contents

List of Figures

1. Introduction

This section contains a contextual introduction for the project and a collection of related work found during the research phase of development.

1.1. Context

Creating and rendering realistic models of trees manually requires advanced expertise with modelling software packages. This limits the ability to produce convincing 3D scenes for small developers with restricted resources.

The purpose of including trees in a natural environment is to provide realism. Trees are common natural structures present in even simple environments throughout the history of computer graphics and have seen many iterations as technology has advanced allowing for more detailed and realistic results.

The aim of this project is to provide a method for creating and rendering trees to be used in a real-time graphics application. This method should be simple to use and implement into an existing OpenGL project.

1.2. Related Work

In this subsection various related works will be discussed with respect to how they contribute to the main knowledge required for this project, ordered chronologically. These areas of main knowledge are the branching structure, branch thickness and leaf placement of the constructed trees.

Aristid Lindenmayer proposed an early theory for the development of organisms using a cells current state and being combined with input that the cell receives from its neighbour (Lindenmayer, 1968a) (Lindenmayer, 1968b). Two new cells are produced from this development to replace the existing cell and the cycle repeats for the two new cells.

This process of generating new outputs recursively to produce larger structures became “Lindenmayer systems” or the abbreviated “L-systems” which became important tools in pattern generation for future applications. L-systems became a common approach for the generation of branching patterns in flora which is where the aim of this project is concerned.

Another early paper that is referenced by many later studies of the subject is by Honda (1971) where he presented one of the earliest algorithmic methods for creating a branching structure. This was done by starting with a parent branch which then bifurcates into two child branches and rotating them by given angles about the termination point of the parent branch, an example of which can be seen in Figure 1. By continuing this method he treated each child branch as another parent and bifuracted them to expand the branching structure which would be continued until a desired result is reached.

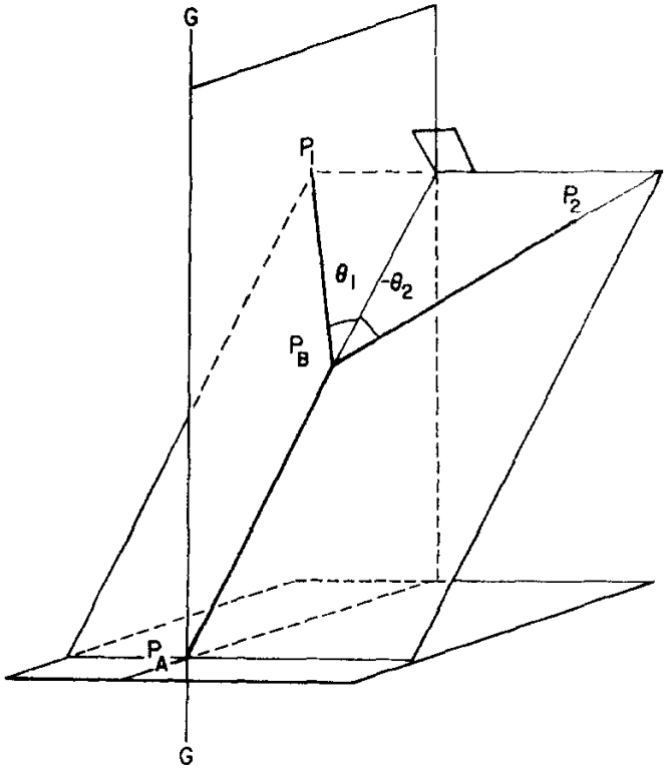


Figure 1.: Example diagram of parent branch P_AP_B bifurcating into child branches P_BP_1 with a rotation of θ_1 and P_BP_2 with a rotation of $-\theta_2$.

This paper does not include any information relating to branch thickness or leaf placement as its focus was only on the branching structure. However, this paper provides a groundwork for many later studies that will be discussed in this project.

Bloomenthal (1985) presents his own tree generation process, specifically for a maple tree. He wanted to move beyond simply the branching pattern of a tree which most

efforts before had focused on (Brooks et al., 1974; Marshall et al., 1980; Kawaguchi, 1982; Aono and Kunii, 1984; Smith, 1984). For the construction of the branches and trunk he uses splines to create a tree skeleton, the use of splines rather than straight lines is chosen to produce a more natural structure. He then uses generalised cylinders with varying radii across the splines to create thickness. He also describes the use of ramiforms, shown in Figure 2, to make branch bifurcations realisticly curve rather than have an acute separation.

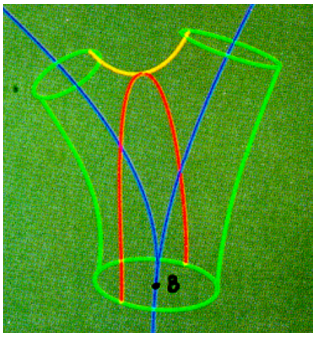


Figure 2.: A ramiform used to create a realistic bifurcation.

The method he describes for leaf placement uses polygonal structures to which he applies a leaf texture, these polygons are then brought together in clusters that he calls "configurations" and for each limb that does not exceed a given diameter and that has no outgoing limbs, a leaf configuration is chosen at random, scaled randomly and placed at the limb tip. The interior lines of each polygon correspond to hinge points of the leaves that could be manipulated to show leaf distortion from wind. This method provides a more realistic result than the use of simple primitives (Kawaguchi, 1982; Lintermann and Deussen, 1999; Candussi et al., 2005).

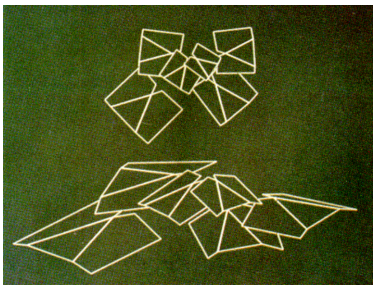


Figure 3.: A leaf "configuration": plan view (top) and perspective view (bottom).

Prusinkiewicz et al. (1996) presents a method of tree generation using an advanced L-system, developed from the work of Lindenmayer (1968a) and applying that system to the bifurcating parent and child branch trios displayed by Honda (1971). Their chosen method for representing their models is by using turtle graphics in a 3D space and the string generated from the L-system uses symbols that correspond to controlling the turtles orientation in space (Szilard and Quinton, 1979; Prusinkiewicz, 1986).

Prusinkiewicz et al. produce a variation on a standard L-system known as a "deterministic" L-system or "D0L-system" which produces the same result every time. They demonstrate the application to turtle graphics with the well-known snowflake curve (Mandelbrot, 1982; Koch et al., 1906) shown in Figure 4.

The D0L-system uses an Axiom string and a production rule to recursively develop a longer and more complex string. Once a chosen result is reached, the string is traversed and each character or group of characters corresponds to the movement of the turtle. Fractals give a convenient illustration of the basic workings of an L-system (Prusinkiewicz, 1986; Prusinkiewicz and Lindenmayer, 2012). For example, the D0L-system below represents that of the snowflake fractal where $F(s)$ corresponds to a movement forward of distance s and $+(r)$ or $-(r)$ correspond to a rotation positively or negatively about the x axis by r degrees.

Axiom ω: $F(1)-(120)F(1)-(120)F(1)$

Production p_1: $F(s) \rightarrow F(s/3)+(60)F(s/3)-(120)F(s/3)+(60)F(s/3)$

The axiom draws an equilateral triangle with edges of unit length. Production p_1 replaces each line segment with a polygonal shape where the line segment has been given a convex triangular appendage. The Production p_1 can be visualised as so:

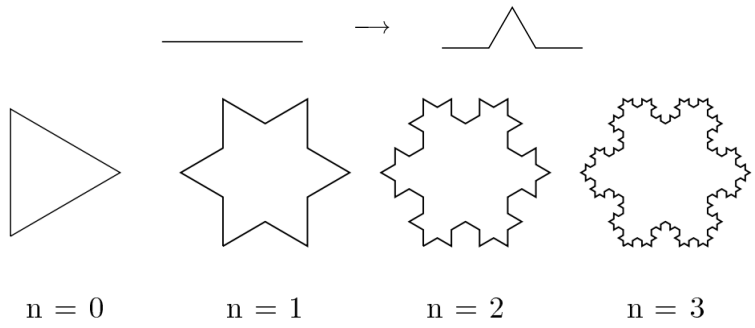


Figure 4.: The visualised production rule p_1, axiom and first 3 recursions defined by the snowflake D0L-system from (Prusinkiewicz et al., 1996) pg.12.

Prusinkiewicz et al. go on to present a complex D0L-system to be used for constructing tree structures by creating a Parametric L-system allowing for input to alter the axiom string producing different results along with using a conditional requirement to decide automatically when to stop recursing.

This L-system is defined below, in this context $A(s,w)$ refers to a line segment or "apex" of length s and width w. Production p_1 begins with the conditional statement to only continue if the length s of the current apex is greater than or equal to *min* which is a chosen minimum length. If the condition is met then $!(w)F(s)$ draws a line of length s and width w.

Lines a_1 and a_2 refer to the bifurcated child apices of the parent apex which are given a rotation of α_1 and α_2 radians about the x axis and a rotation of ϕ_1 and ϕ_2 radians about the y axis. The length of the child apices is calculated by multiplying the parent apex length s by the corresponding length degradation parameters r_1 and r_2 respectively. A degradation in width is calculated using q and e.

ω: $A(100, w_0)$

p_1: $A(s,w) : s >= min \rightarrow !(w)F(s)$

a_1 $[+(\alpha_1)/(\phi_1)A(s*r_1, w*q^e)]$

a_2 $[+(\alpha_2)/(\phi_2)A(s*r_2, w*(1-q)^e)]$

This L-system is used to produce various complex tree patterns, shown in section A of the Appendix, ranging from mathematical fractal patterns to some realistic looking structures.

Prusinkiewicz et al. do not directly propose a method of leaf placement as this paper focuses on tree structure and the manipulation of L-systems, they do however provide an example where terminating apices of an L-system could be replaced with a leaf shape to give a rudimentary leaf placement approach.

Weber and Penn (1995) produced a model for creating and rendering trees that emphasises overall geometric structure of the tree rather than strictly following botanical principles which they credit similarity to the aims of Reeves and Blau (1985). They admit that their model resembles that proposed by Oppenheimer (1986) which used fractals with parameters such as "branching angle" and "helical twist" to produce trees. However, Weber et al. propose that the use of fractal theory self similarity by Oppenheimer is not necessesary and restricts results to only a small number of basic trees.

Weber et al. used cylinders and cones as branch segments, each child branches base circle coinciding with the top circle of the parent branch. Transformation of the child

branches being controlled through the intentionally intuitively named parameters that can be edited by the user to produce a desired result.

Figure 5 shows some of the many parameters that can be used to affect tree generation, *0SegSplits* defines the number of splits at each branch termination, *0SplitAngle* controls the angle at each split, *0CurveRes* refers to the taper given to each branch to slowly reduce thickness of each branch, and *1Rotate* defines the *y* axis rotation of the branch. The number in front of each parameter refers to a branch, 0 being the root branch, the values of parent branch parameters are passed on to the child branches.

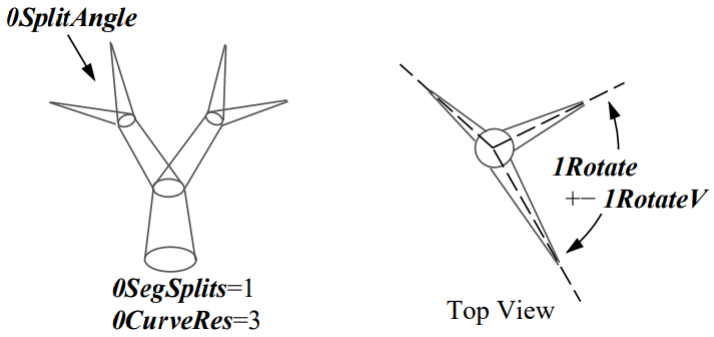


Figure 5.: Example of tree parameters.

Leaf placement is handled using a "Level" system that refers to the number of recursions from the root branch. If a parameter *Leaves* is non-zero then the final level of recursion is used to place leaves on the final set of child branches, using a set of leaf specific parameters to control the rotation of each leaf. The shape of the leaves is controlled using the parameter *LeafShape* allowing for different tree species to be represented by adding different leaf shapes. Weber and Penn (1995) served as a groundwork for many future approaches and advances in tree rendering (Remolar et al., 2004; Wesslén and Seipel, 2005). Weber himself went on to produce a simulation focused method of tree generation including some extra features such as branches avoiding obstacles (Weber, 2008).

Another branch structure method, that differs uniquely from methods mentioned above, is that of space colonisation (Prusinkiewicz et al., 2007). The steps of this method can be seen in Figure 6 and works by producing a three-dimensional envelope to be used for the tree crown containing a set of "attraction points" which are used as targets for branch growth. The tree skeleton interatively expands by producing new branches in the direction of the attraction points. Once an attraction point is reached by a branch, the

point is removed and the branch terminates. This process is continued until all attraction points are reached, when no new branches are within attraction radius of remaining points or when a user inputted number of iterations has been reached.

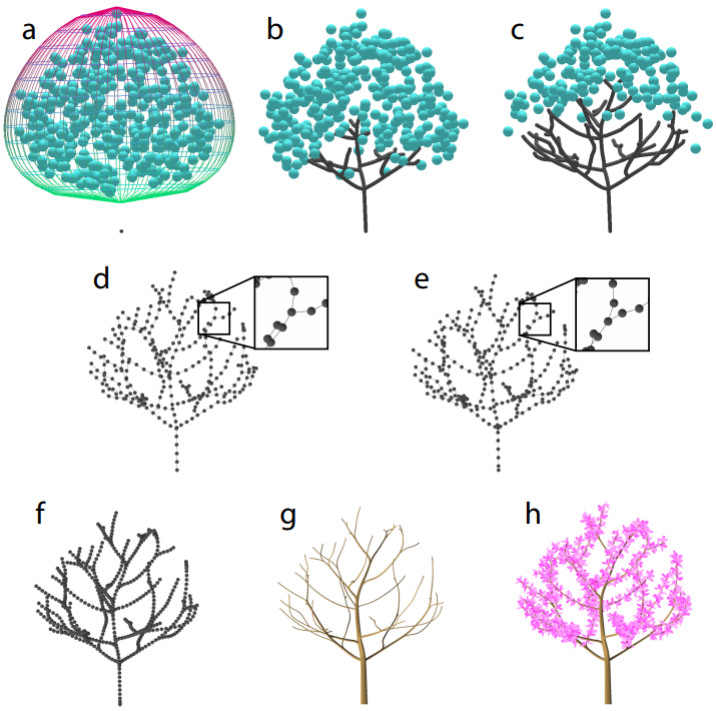


Figure 6.: Space colonisation steps.

Once a tree skeleton has been finalised they determine the width of the limbs using the pipe model (Shinozaki et al., 1964) and the tree geometry is modeled using generalised cylinders (Bloomenthal, 1985).

A leaf placement method is not provided as part of this paper due to it focusing solely on branch construction. However, they do suggest that the placement density of leaves or flowers could be controlled using absolute distance from the root position to give a realistic dispersion of leaves throughout the tree crown.

2. Design and Development

After going through the relevant related work, the chosen method for constructing the trees was using an L-system. This was chosen because a complex parameterised L-

system could be designed to produce various different types of trees while using the same logic pipeline. Therefore, the module could be relatively simple and contained, and still be quite flexible with its results.

Using an L-system meant that a basic branch object was needed to construct each section of the tree. The first approach investigated was the use of quadric cylinders, that could be used to give branches thickness. Quadric cylinders could also be defined with a taper allowing for a smooth reduction in thickness towards external branches. Figure 7 shows an early test bifurcation using quadric cylinders.

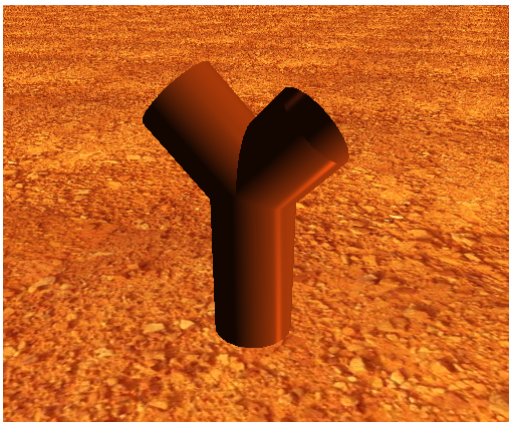


Figure 7.: 3 quadric cylinders positioned as a bifurcation.

When positioned correctly, the join between overlapping cylinders was acceptably smooth producing a believable bifurcation. However, this example was made by hard-coding transformation for each quadric that was applied separately with no direct relationship between the parent and child branches. The quadric class was also closed off from being modified, meaning that the construction method would be restricted to however the quadrics can be manipulated and I would not be able to make any tweaks to how the quadrics are constructed to assist with the L-system generation. Another foreseen difficulty was the issue of texturing, which would me more complex when using quadrics. Because of this the decision was made to instead create a cylinder class specifically for this purpose, therefore allowing better control of generating the models and allowing for tweaks to made when integrating them with the L-system.

The cylinder class was intially designed to produce a line rather than an entire cylinder so that testing could be carried out more quickly and tree skeletons would be easier to

analyse for mistakes in the L-system. Unfortunately the cylinder class did not advance further than using lines for the construction but this will be explained in the Evaluation.

These cylinder objects are constructed using vertex array objects and vertex buffer objects with the aim of being able to store the geometry data of a constructed tree as a vector of these objects and then simply interate through and load each cylinder object. The cylinder object is constructed using a glm::vec3 for the base and top centre point of the cylinder, and has a top and bottom radius that would be used to control tapering.

To test the use of this cylinder class and develop understanding of L-system production some basic L-systems were produced. These included a basic binary tree, and a barnsley fern L-system. The L-system for a binary tree is displayed below.

Binary Tree L-system:
ω: a
p_1: $a \rightarrow b[a]a$
p_2: $b \rightarrow bb$

These production rules are applied for a set number of recursions to control the size of the resulting tree. The first 3 recursions of this system, with an axiom of a, are:

1: b[a]a

2: bb[b[a]a]b[a]a

3: bbbb[bb[b[a]a]b[a]a]bb[b[a]a]b[a]a

Each character refers to a certain action when constructing the tree. In this case a means draw a branch segment terminating with two small lines to represent a leaf, b means draw a branch segment, [means push position and angle and rotate 45 degrees left about x and] means pop position and angle and rotate 45 degrees right about x. Six recursions using this system produces the binary tree shown in Figure 8, and you can see the steps used to construct the tree using this method in section C.

A somewhat more advanced L-system, used to test whether the line segments can produce a more natural shape, is an L-system designed to create an adapted version of the Barnsley fern fractal. The production rules for this L-system are shown below.

Barnsley Fern L-system:
ω: a
p_1: $a \rightarrow b + [[a] - a] - b[-ba] + a$
p_2: $b \rightarrow bb$

This L-system produces a natural looking branched fern pattern, shown in Figure 8. This proves that this method of line segments can be used for this projects aim.

Figure 8.: Binary Tree L-system result after six recursions (left) and Barnsley Fern L-system after five recursions (right).

Having confirmed the effectiveness of the cylinder class to produce lines correctly, the next step was to produce a more advanced L-system to create more details structures and make the transition to 3D.

The decision was made to adapt the parametric L-system described by Prusinkiewicz et al. (1996), discussed in the related work section, into OpenGL using the cylinder class to generate lines. This required developed understanding of L-systems and adaptation from the proposed turtle graphics approach, over to using OpenGL vertices.

This parametric L-system was described in the related work section but will be re-iterated with respect to it's adaptation in OpenGL.

ω: $A(100, w_0)$

p_1: $A(s,w) : s >= min \rightarrow !(w)F(s)$

a_1 $[+(\alpha_1)/(\phi_1)A(s*r_1, w*q^e)]$

a_2 $[+(\alpha_2)/(\phi_2)A(s*r_2, w*(1-q)^e)]$

The initial approach taken was to try and implement the classical L-system result of string generation and then producing results by iterating through the resulting string, such as the binary tree and barnsley fern described above. However, after an attempted

implementation of this approach, the decision was made to switch to an object oriented construction.

The L-system production rules takes an "Apex" defined as $A(s,w)$ and produces the two child apices a_1 and a_2 which are then also passed through the production rules to produce their own child apices. This proved clumsy when trying to use strings and therefore an Apex class was created to represent the relevant data of each Apex that could then be used to construct the branches.

The data defined in the Apex class includes:

- The *length* and *width* of the branch.
- The *rotateAlpha* and *rotatePhi* which refer to the rotation about the x and y axis respectively.
- Pointers to the parent Apex and child apices.
- The *localRoot* of the Apex which is a glm::vec3 used to represent the 3D coordinates of the base of the branch.
- The integer *level* that stores the number of recursions away and Apex is from the root Apex.
- A boolean *isRoot* to check whether an Apex is the root Apex or not.
- A rotation matrix used to rotate apices using *rotateAlpha* and *rotatePhi* with respect to the cumulative rotations of the parent Apex.

These apices are used to create the line segments required to generate the tree structure. The Line segments are stored in a vector and then, when loading the scene, iterated through to render the tree.

The parameters required for this L-system are passed into a function and then used to construct the Apex object necessary for the tree. These variables are:

- *rootPos*, a glm::vec3 representing the 3D coordinates where you want the tree to be rendered.
- *startBranchLength*, the starting length of branches.

- *minBranchLength*, the minimum branch length used to check the conditional statement in the L-system, if the current branch length is lower than the minimum then the system stops recursing.
- *startBranchWidth*, the starting branch width.
- *angleAlpha1* and *angleAlpha2* control rotations about the x-axis.
- *anglePhi1* and *anglePhi2* control rotations about the y-axis.
- *lengthDegrade1* and *lengthDegrade2* control the rate at which the length of branches degrades from parent to child.
- *count*, used to control the number of recursions directly if the minimum length approach is not suitable.

Alterations of these variables can be used to produce vastly different outcomes as shown by the results of Prusinkiewicz et al. (1996), which are included in section A.

To construct the apices vector, these variables are passed through to a recursive function that creates each Apex object, sets parent and child relationships, and handles length degredation. This function used to construct the apices vector is shown in section D. The initial implementation for constructing the apices vector used a simple loop, this made setting parent-child relationships unecessarilly obtuse however. Therefore the recursive method was developed to streamline the construction process and also improve readability of the code.

Having constructed the apices vector the last step was to iterate through each Apex and use their data to render a line. This is where most issues arose in development as multiple iterations were used for applying rotations before a working solution was found.

Prusinkiewicz et al. (1996) used a proprietary coordinate system to describe their transformations when using turtle graphics and those axes are shown in Figure 9. For reference the axes u, h, l correspond to x, y, z respectively.

The rotations discussed below only display rotations about the x and y axes because these are the only two rotations involved in this L-system.

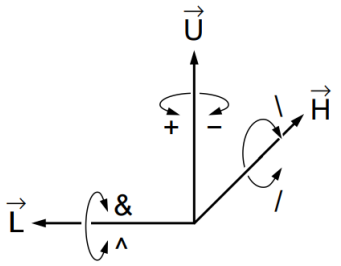


Figure 9.: Axes used to control the turtle in three dimensions.

Initially, rotations were applied by calculating the rotation of each element of the coordinate,

For the x-axis:

$x = x$

$y = y\cos(rotationAlpha) - z\sin(rotationAlpha)$

$z = y\sin(rotationAlpha) + z\cos(rotationAlpha)$

For the y-axis:

$x = x\cos(rotationPhi) + z\sin(rotationPhi)$

$y = y$

$z = z\cos(rotationPhi) - x\sin(rotationPhi)$

These were then simplified to complete all rotations with just one calculation for each element rather than having to run separate calculations for *rotationAlpha* and *rotationPhi*.

$x = length\cos(rotationPhi)\sin(rotationAlpha)$

$y = length\sin(rotationPhi)\sin(rotationAlpha)$

$z = length\cos(rotationAlpha)$

The final method that proved successful was the use of rotation matrices as part of the Apex objects with the glm::rotate function. When the parent of a child Apex is set in the apices vector construction process, the rotation matrix of the child is calculated using the rotation matrix of the parent so that the angle of the parent branch is taken into account.

The code below is a simplification of glm::rotate and assumes local variables of an Apex object, rotation values are converted to radians for use with glm::rotate and the last parameter refers to the axis of rotation shown x, y however, in code this would be a glm::vec3 with a 1 in the parameter corresponding to the chosen axis.

Alpha rotation:

$rotationMatrix = rotate(parentRotationMatrix, rotationAlpha, x)$

Phi rotation:

$rotationMatrix = rotate(rotationMatrixAlpha, rotationPhi, y)$

The rotation Matrix of an Apex object can then be used to rotate a point for creating a line segment along with the other Apex attributes. The algorithm below displays how the apices vector is used to create lines ready to be rendered into a scene.

Algorithm 1. constructLines(*apices*) **return** *lines*

Require: *apices*, vector of Apex objects
Ensure: *lines*, vector of line objects

```
1:  for all Apex in apices do
2:      top ← ApexRotationMatrix * (0, ApexLength, 0, 1)
3:      top ← (topX + ApexX, topY + ApexY, topZ + ApexZ)
4:      newLine ← generateLine(ApexLocalRoot, top)
5:      Push newLine onto lines
6:      if Apex has first child then
7:          Set Apex first child localRoot to top
8:      end if
9:      if Apex has second child then
10:         Set Apex second child localRoot to top
11:     end if
12: end for
13: return lines
```

Figure 10.: Construction of line segments ready for rendering.

The code pipeline has also been designed so that if new L-systems were needed then they could be implemented as part of a new *L-system* object. This was done to allow for future adopters of the module to expand the source code to meet their requirements.

3. Analysis of Results

I this section, three different trees will be analysed for their performance and then analysed when all three trees are loaded at once. These trees are results from the method desribed in the previous section, and are examples taken from Prusinkiewicz et al. (1996) to allow benchmarking in the evaluation section.

3.1. Short Tree

The first tree, shown in Figure 11, has a short, wide crown with an even spread of branches. Alterations to this structure would produce results close to that of an ash tree.

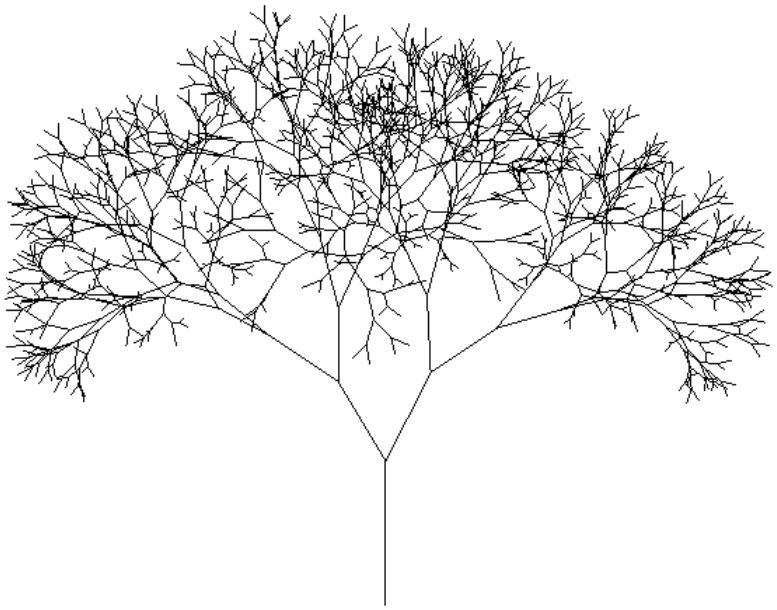


Figure 11.: Short Tree.

The performance of this tree and others was measured using the built in diagnostics tools in the visual studio editor. An empty scene has been measured at a memory usage of 43MB so the results included are the total usage with the base 43MB subtracted.

The short tree has a consistent memory usage of 65.4MB, and subtracting the base usage of 43MB gives us a usage of 22.4MB. CPU utilization fluctuates between 6% and 9%, and when rotated the tree does not cause a reduction in frames per second.

3.2. Medium Tree

The next tree, shown in Figure 12, is a medium sized tree with some main spreading branches but has a lower length degredation than the short tree causing it to grow taller. This structure could be used to represent a maple tree.

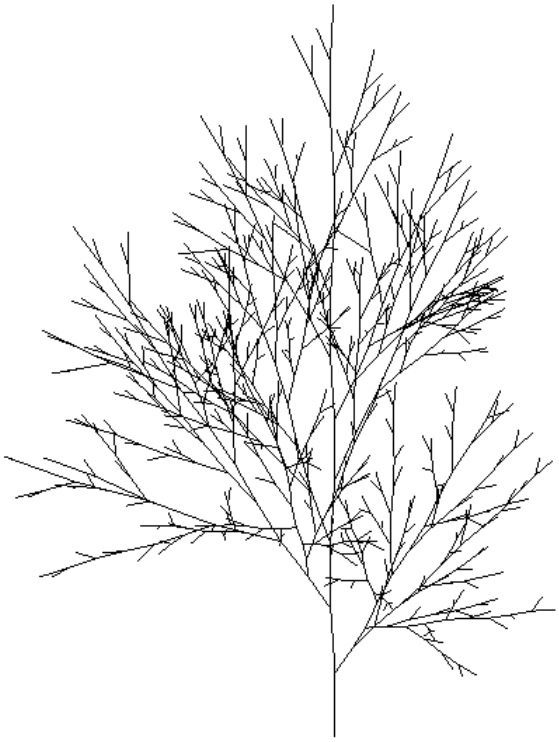


Figure 12.: Medium Tree.

The medium tree has a memory usage of 56.9MB, real usage 13.9MB. Fluctuates between 6% and 9% CPU utilization, and when rotated does not cause a reduction in frames per second.

This tree has a higher minimum branch length than the short tree, meaning that less branch segments are generated, which is why the memory usage is lower.

3.3. Tall Tree

The third tree, shown in Figure 13, is a tall tree with less widely spreading branches that have shorter child segments making the tree stand tall and compact with minimal overhang from the trunk. The structure below would be well suited for creating a birch tree.

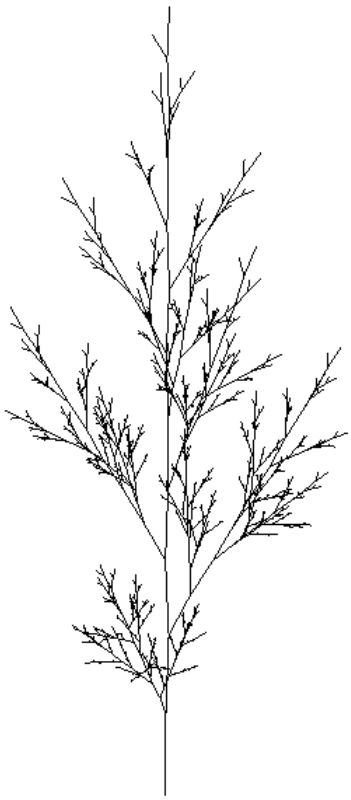


Figure 13.: Tall Tree.

The tall tree has a memory usage of 63.6MB, real usage 20.6MB. Fluctuates between 6% and 9% CPU utilization, and when rotated does not cause a reduction in frames per second.

This tree has a minimum branch length greater than the short tree but less than the medium tree, hence the memory usage being between that of the previous examples.

3.4. Multiple Trees

The final test was to take multiple tree models and load them into the same scene. This is important to test as an evironment containing trees is likely to contain more than one, such as a forest. Figure 14 shows a scene with all three of the previously analysed trees loaded in at once.

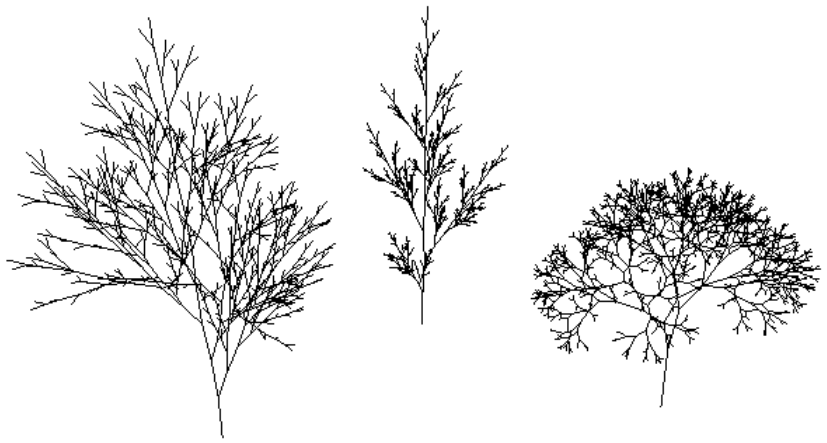


Figure 14.: Medium (left), Tall (centre) and Short (right) trees.

This scene requires more resources to load than a singular tree with a memory usage of 89MB. Worth noting is that with the 89MB measurement, subtracting the base 43MB gives us a usage of 46MB which does not cooincide with an addition of each trees separate memory usage. This is likely due to every tree being stored in a single vector.

This scene fluctuates CPU utilization between 6% and 9% for the first 5 seconds approximately before steadying at 6%. For the first few seconds this scene has some frame slowdowns and loads slower than the previous examples.

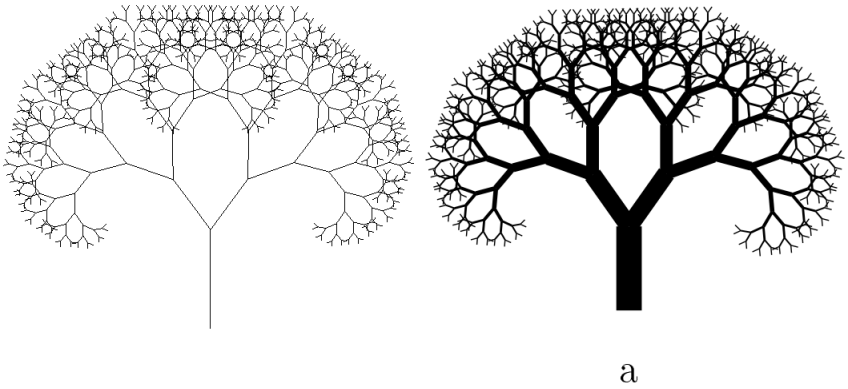
4. Evaluation and Discussion

In the first part of this section, benchmarking against the results of Prusinkiewicz et al. (1996) has been carried out to display the accuracy of adaptation for the parametric L-system. Following this will be a discussion about the issues of the project and how they would be remedied.

4.1. Benchmarking

This section will list multiple figures showing a comparison between the results of Prusinkiewicz et al. (1996) shown in section A, and results obtained from using the same input with the adapted L-system in OpenGL.

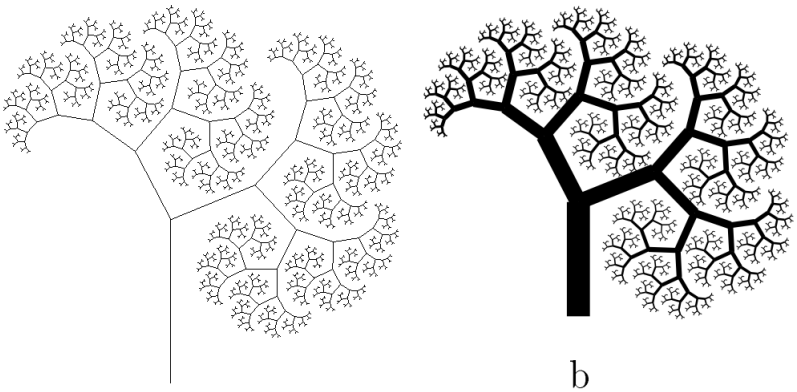
The inputs used for each tree can be seen in section B.



a

Figure 15.: Tree A Comparison: OpenGL (left), Prusinkiewicz et al. (right)

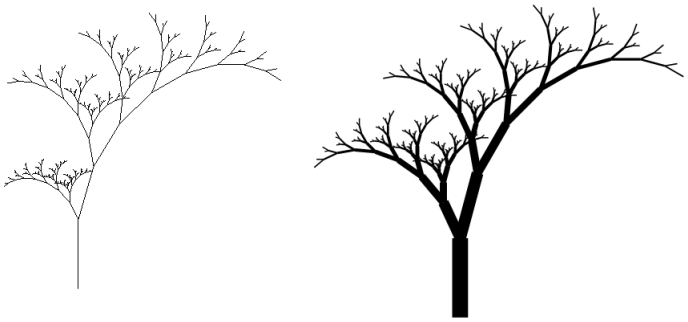
Similarly to the intial testing of string based L-systems, the binary trees from 15 are the most basic trees produced for testing branch bifurcation and *x* rotations.



b

Figure 16.: Tree B Comparison: OpenGL (left), Prusinkiewicz et al. (right)

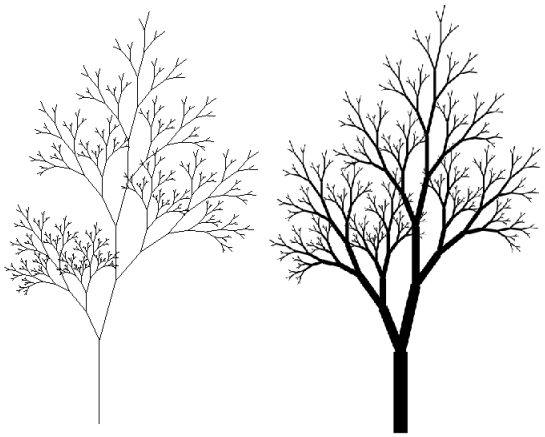
A classical use of L-system productions to create a common fractal.



c

Figure 17.: Tree C Comparison: OpenGL (left), Prusinkiewicz et al. (right)

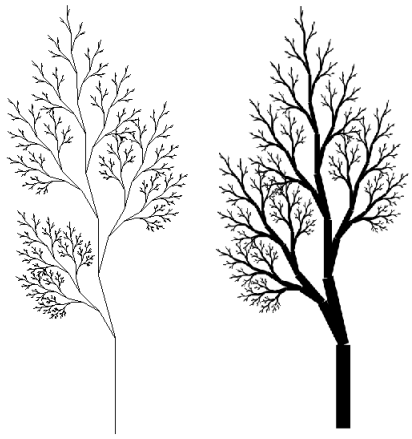
Figure 17 shows the most simple tree to include a *y* rotation of 180 degrees to flip bifurcations, producing a fan like pattern.



d

Figure 18.: Tree D Comparison: OpenGL (left), Prusinkiewicz et al. (right)

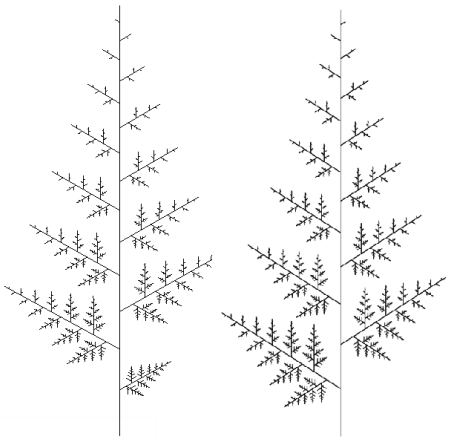
Tree D was the most simple tree to start including more believably natural structures with nicely sweeping branches that shorten into small twigs.



e

Figure 19.: Tree E Comparison: OpenGL (left), Prusinkiewicz et al. (right)

Tree E shows the fractal capabilities of the L-system once again while also including some of the natural structuring seen in Tree D.



f

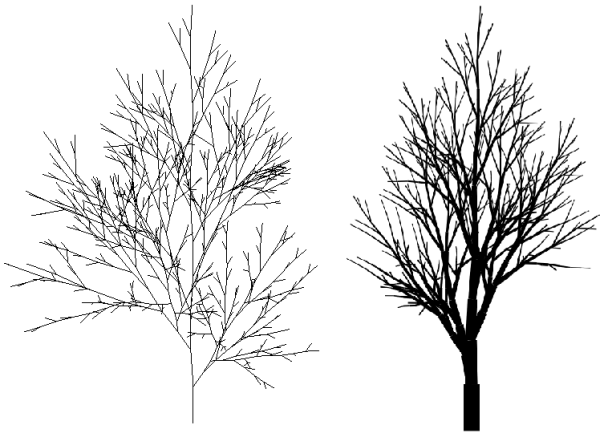
Figure 20.: Tree F Comparison: OpenGL (left), Prusinkiewicz et al. (right)

Tree F is very different from the other trees presented and is another display of fractal construction like Tree B.

Tree G, H and I are the trees that make use of *y* rotations to create a 3D structure that looks as close to a natural structure as is possible with this system.

g

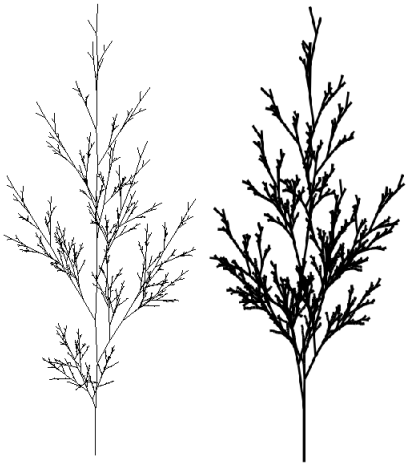
Figure 21.: Tree G Comparison: OpenGL (left), Prusinkiewicz et al. (right)



h

Figure 22.: Tree H Comparison: OpenGL (left), Prusinkiewicz et al. (right)

These trees show the full potential of the system and the structures it can produce. Overall the benchmarks against the results of Prusinkiewicz et al. (1996), have been successful in reproducing the presented examples. However, due to some issues with development of this project, some features are absent from the final system. Most apparent of these when looking at the benchmarks is that the OpenGL system does not



i

Figure 23.: Tree I Comparison: OpenGL (left), Prusinkiewicz et al. (right)

take into account width of branches. This, and other problems, will be explained in the next section.

4.2. Discussion of Issues

There are some features, researched and discussed as part of the literature review, that have not been included in the system due to time constraints. This is because of an unseen issue with *y* rotations that blocked progress for a significant amount of time.

The first feature not included that was planned is adding thickness to branches. The framework for adding thickness has been designed into the existing system but not implemented. This would have been an extension of the lines used for tree construction using cylinders, inspired by the branch implementation of Weber and Penn (1995). This would require additions to the *constructGeometry()* function of the cylinder class. Currently the function produces three points, one at the top of the line and two at the bottom.

A cylinder implementation would take the top and bottom points of the line and use them as a centre point for a circle. Using the top and bottom radii variables as an offset from their respective centres, multiple points would be generated around the radius of the points to create the top and bottom pseudo-circular faces. A number of segments would need to be chosen to give the desired smoothness, this could be factored into the L-system input to give more control to the user. Figure 24 shows the proposed geometry.

Another issue not addressed by the project is that of texturing the tree branches. This would require further research into how to apply textures to proprietary geometry. Setting texture indices would be included as part of the *createGeometry()* function. Texturing may require more advanced geometry including vertical segmentation of the cylinders to allow the use of square textures without stretching.

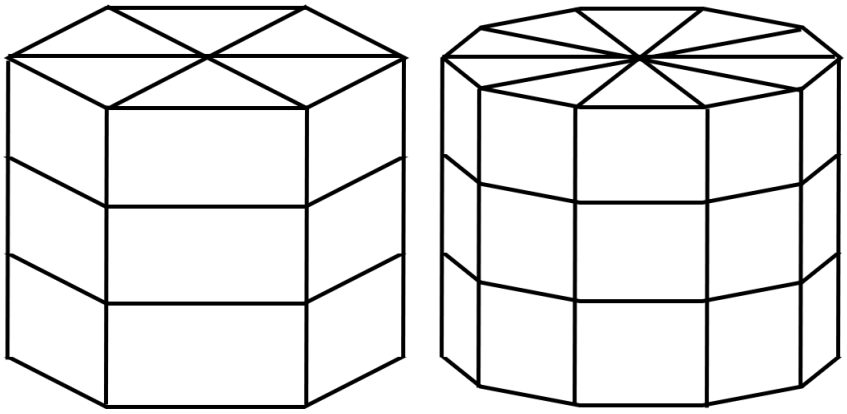


Figure 24.: Example of the proposed cylinder generation method with six and ten segments.

Leaf placement was also researched and discussed as part of this project, but was also dropped due to time constraints. The method that would be used for leaf placement would use a combination of approaches proposed by Weber and Penn (1995) and Bloomenthal (1985). The *level* attribute of an Apex object, which tracks how far from the root an Apex is, would be used to decide which branches to place leaves on. The furthest levels of the tree would be populated with leaves to emulate how real trees have more leaves on external branches further from the root. A choice of what level to start placing leaves at could be included in the L-system inputs to allow users to control the density of leaves.

The specific method of leaf placement would be more complicated. Again following the approaches of Bloomenthal and Weber et al., leaves would be rendered as prefabs with a chosen texture. The difficulty here would come from producing a method to place the leaf models on the surface of the branch correctly. A possible method for choosing placement on a branch would be to choose points on the surface of a cylinder, such as some of the vertical segment vertices, and to render the leaf with its base at the chosen

point. The leaf would then need to be rotated to face upwards and make sure that the leaf is rotated so it does not clip with the branch it is on.

Another issue not remedied in this system is the problem of branches overlapping. This would be solved as part of the cylinder generation process. Each cylinder would have to be checked by calculating the local space taken up by the cylinder. Using the top and bottom termination points along with the assigned widths of the cylinder you would then have to check if any other cylinders local space overlaps with the local space of the current cylinder. If an overlap is found then pruning needs to be done to remove one of the overlapping branches. The branch that will be pruned will be chosen based on its *level*, the higher level branch will be pruned because it will be further from the root and will likely have less child branches that also need to be pruned. The pruned branch and all of its children must then be removed from the cylinder vector so that they will not be loaded. The parent of the pruned branch must then have its top width reduced to 0 so that it properly terminates and does not leave a flat termination point.

The system should also make better utilization of hardware to increase performance. Currently when trying to load multiple trees at once there is a noticeable slow down in the program. This slow down will not be adequate for creating more complicated environments with many more trees.

5. Conclusion

References

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

Bloomenthal, J. (1985). Modeling the mighty maple. *ACM SIGGRAPH Computer Graphics*, 19(3):305–311.

Brooks, J., Murarka, R. S., Onuoha, D., Rahn, F. H., and Steinberg, H. A. (1974). An extension of the combinatorial geometry technique for modeling vegetation and terrain features.

Candussi, A., Candussi, N., and Höllerer, T. (2005). Rendering realistic trees and forests in real time. In *Eurographics (Short Presentations)*, pages 73–76. Citeseer.

Honda, H. (1971). Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of theoretical biology*, 31(2):331–338.

Kawaguchi, Y. (1982). A morphological study of the form of nature. *SIGGRAPH Comput. Graph.*, 16(3):223–232.

Koch, H. et al. (1906). Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta mathematica*, 30:145–174.

Lindenmayer, A. (1968a). Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299.

Lindenmayer, A. (1968b). Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of theoretical biology*, 18(3):300–315.

Lintermann, B. and Deussen, O. (1999). Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):56–65.

Mandelbrot, B. B. (1982). *The fractal geometry of nature*, volume 1. WH freeman New York.

Marshall, R., Wilson, R., and Carlson, W. (1980). Procedure models for generating three-dimensional terrain. *SIGGRAPH Comput. Graph.*, 14(3):154–162.

Oppenheimer, P. E. (1986). Real time design and animation of fractal plants and trees. *ACM SiGGRAPH Computer Graphics*, 20(4):55–64.

Prusinkiewicz, P. (1986). Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253.

Prusinkiewicz, P., Hammel, M., Hanan, J., and Mech, R. (1996). L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer.

Prusinkiewicz, P., Lane, B., and Runions, A. (2007). Modeling trees with a space colonization algorithm. *NPH*, 7:63–70.

Prusinkiewicz, P. and Lindenmayer, A. (2012). *The algorithmic beauty of plants*. Springer Science & Business Media.

Reeves, W. T. and Blau, R. (1985). Approximate and probabilistic algorithms for shading and rendering structured particle systems. *ACM siggraph computer graphics*, 19(3):313–322.

Remolar, I., Rebollo, C., Chover, M., and Ribelles, J. (2004). Real-time tree rendering. In Bubak, M., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *Computational Science - ICCS 2004*, pages 173–180, Berlin, Heidelberg. Springer Berlin Heidelberg.

Shinozaki, K., Yoda, K., Hozumi, K., and Kira, T. (1964). A quantitative analysis of plant form-the pipe model theory: I. basic analyses. *Japanese Journal of ecology*, 14(3):97–105.

Smith, A. R. (1984). Plants, fractals, and formal languages. *SIGGRAPH Comput. Graph.*, 18(3):1–10.

Szilard, A. L. and Quinton, R. (1979). An interpretation for dol systems by computer graphics. *The Science Terrapin*, 4(2):8–13.

Weber, J. and Penn, J. (1995). Creation and rendering of realistic trees. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 119–128, New York, NY, USA. Association for Computing Machinery.

Weber, J. P. (2008). Fast simulation of realistic trees. *IEEE Computer Graphics and Applications*, 28(3):67–75.

Wesslén, D. and Seipel, S. (2005). Real-time visualization of animated trees. *The Visual Computer*, 21(6):397–405.

A. Parameterised L-system Trees

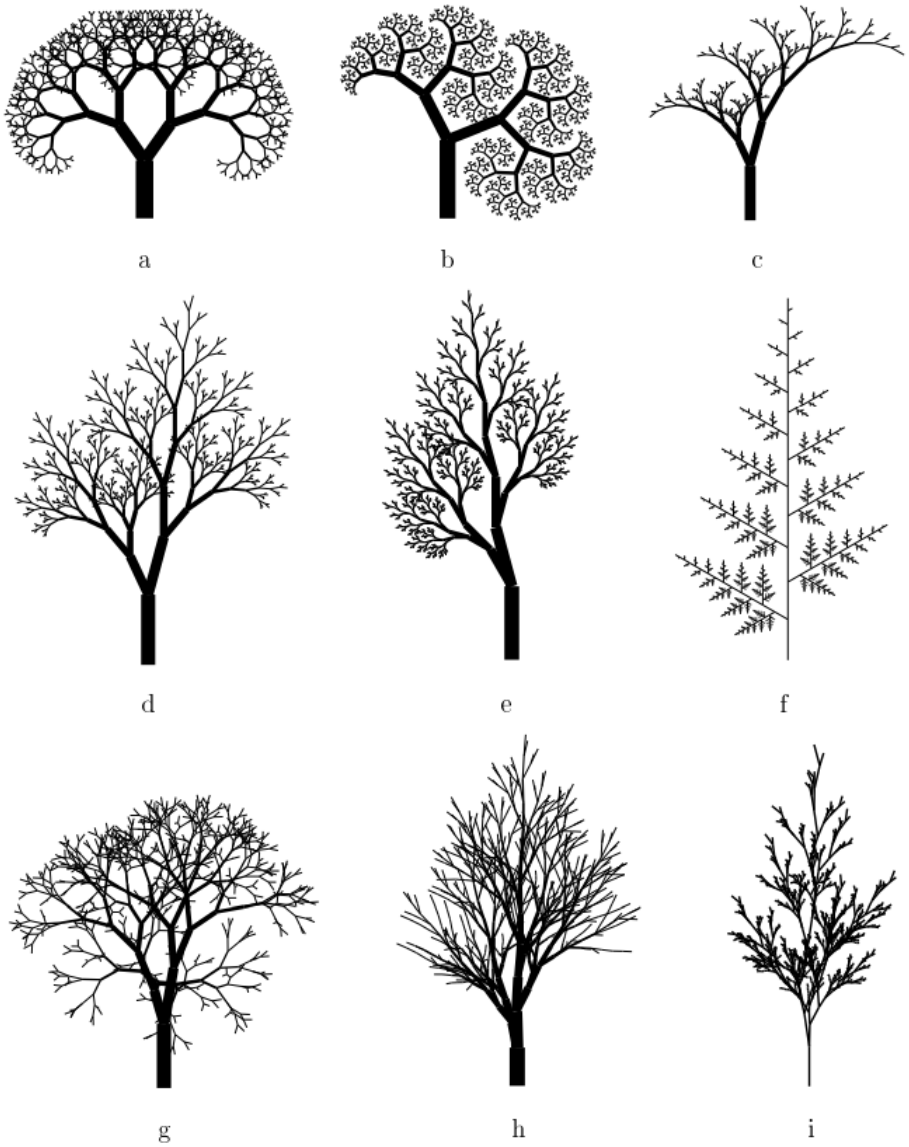


Figure 25.: Tree structures produced using the proposed L-system described by Prusinkiewicz et al. (1996)

B. L-system Input Table

Tree	*min*	a_1	a_2	p_1	p_2	r_1	r_2	*count*
a	0.0	35	-35	0	0	0.75	0.77	10
b	0.017	27	-68	0	0	0.65	0.71	12
c	0.005	25	-15	180	0	0.50	0.85	9
d	0.0	25	-15	180	180	0.60	0.85	10
e	0.001	30	15	0	180	0.58	0.83	11
f	0.005	0	60	180	0	0.92	0.37	15
g	0.0	30	-30	137	137	0.80	0.80	10
h	0.25	5	-30	-90	90	0.95	0.75	12
i	0.05	-5	30	137	137	0.55	0.95	12

Figure 26.: Table of inputs that will produce the example trees shown in the benchmarking section. The *min* input assumes a *startBranchLength* of 1.0.

C. Binary Tree L-system Algorithm

Algorithm 2. generateBinaryTree(*axiom, numRecursions*) **return** *branchList*

Require: *axiom*, the initial string for the L-system
Require: *numRecursions*, number of times to run production rules
Ensure: *branchList*, list of branch objects that can be rendered

```
1:  systemString ← getLsystemString(axiom, numRecursions)
2:  currentPosition ← bottom of tree
3:  rotationX ← 0
4:  positionList, angleList
5:  for all character in systemString do
6:      if character = a then
7:          newBranch ← short terminating branch
8:          rotate newBranch by rotationX about x-axis
9:          translate bottom of newBranch to currentPos
10:         push newBranch onto branchList
11:     else if character = b then
12:         newBranch ← long branch
13:         rotate newBranch by rotationX about x-axis
14:         translate bottom of newBranch to currentPos
15:         push newBranch onto branchList
16:         currentPos ← top of newBranch
17:     else if character = [ then
18:         push currentPos onto positionList
19:         push rotationX onto angleList
20:         rotationX ← rotationX + 45
21:     else if character = ] then
22:         pop positionList to currentPos
23:         pop angleList to rotationX
24:         rotationX ← rotationX − 45
25:     end if
26: end for
27: return branchList
```

Figure 27.: Construction method with Binary Tree L-system

D. Construct Apices Vector

Algorithm 3. constructTree(*root, apices, a1, a2, p1, p2, l1, l2, r1, r2, min, w1, w2, count*) **return** *branchList*

Require: *root*, parent Apex of this recursion
Require: *apices*, current vector of all generated apices
Require: Generation variables, explained in the caption below
Ensure: *apices*, vector of Apex objects

1: Create new Apex object *apex1* with variables *a1, p1, l1*r1, w1*
2: Create new Apex object *apex2* with variables *a2, p2, l2*r2, w2*
3: Set children of *root* to *apex1* and *apex2*
4: Add *root* to *apices*
5: $count \leftarrow count - 1$
6: **if** $count > 0$ **then**
7: **if** $l1 > min$ **then**
8: *apices* ← constructTree(*apex1, apices, a1, a2, p1, p2, l1*r1, l1*r1, r1 r2, min, w1, w2, count*)
9: **end if**
10: **if** $l2 > min$ **then**
11: *apices* ← constructTree(*apex2, apices, a1, a2, p1, p2, l2*r2, l2*r2, r1 r2, min, w1, w2, count*)
12: **end if**
13: **end if**
14: **return** *apices*

Figure 28.: Construction of apices vector: *a1*, *a2*, *p1* and *p2* are rotations, *l1* and *l2* are lengths, *r1* and *r2* are length degrade values, *w1* and *w2* are widths.