CONFORMING REALISTIC, PROCEDURAL TREE MODELS TO USER-DRAWN SHAPES

By

Makayla Moster

A paper submitted in partial fulfillment of the requirements to complete Honors in the Department of Computer Science.

Approved By:

Examining Committee:

Dr. Brittany Morago
Faculty Supervisor

Dr. Toni Pence

Dr. Russell Herman

Dr. Curry Guinn
Chair, Computer Science

Honors Council Representative

Director of the Honors Scholars College

University of North Carolina Wilmington

Wilmington, North Carolina

April, 2019

TABLE OF CONTENTS

ABSTRACT

Generating digital models of realistic trees is generally an arduous task and requires a specialized set of skills. While modeling realistic branching structures by hand can be a time-consuming and difficult chore, building tree models procedurally can quickly yield a wide variety of differing structures by closely adhering to patterns and rules observed in nature. There are two main components to this project. The first component involves generating realistic tree models by incorporating natural lighting, varying branch thickness, and natural randomness. The second component involves incorporating user input to constrain the tree structure to sketches of shapes. This paper describes such a system and shows how it can be used to build unique three-dimensional models of trees with user interaction.

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Generating digital models of realistic tree structures is generally an arduous task and requires a specialized set of skills. While modeling realistic branching structures by hand can be a time-consuming and difficult chore, building tree models procedurally can quickly yield a wide variety of differing structures. By adhering to the patterns and rules of nature, procedural modeling efficiently creates realistic models involving little to no user input.

Interactive modeling of tree structures has become popular in the recent years due to the quickness and ease of creating such structures. These structures can be incorporated into scenery for animations, virtual reality, video games, etc. There are many ways to generate procedural trees including using methods and technology such as particle systems, guiding vectors, sketches, silhouettes, images, virtual reality software, or combinations of these. In one example related to this project called TreeSketch, users sketch using a tablet and simultaneously a procedural modeled tree that resembles their brushstrokes is displayed on the screen [9].

The research within this paper focuses on the creation of three-dimensional tree structures that conform to two-dimensional user-drawn shapes in a realistic manner as shown in Figures 1 - 3. The base tree model was developed over a one-semester Directed Independent Study and then expanded upon as a one-year Honors Project. To create a more natural-looking tree, lighting and varying branch thicknesses are added to the base model. Then, user input is incorporated to constrain the tree structure into three-dimensional shapes from graphically drawn two-dimensional sketches.

The organization of this paper is as follows. Chapter 2 consists of background work and methods related to those detailed within this paper. Chapter 3 delves into the methodologies behind creating the base tree model, basic user interaction, and added realism. Chapter 4 details the techniques used to conform the tree structure

into shapes such as a cube, sphere, and user-drawn shapes. Chapter 5 concludes the paper and discusses potential future work.

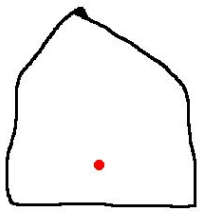


Figure 1: User-Drawn Shape.

Figure 2: Filled in Shape.

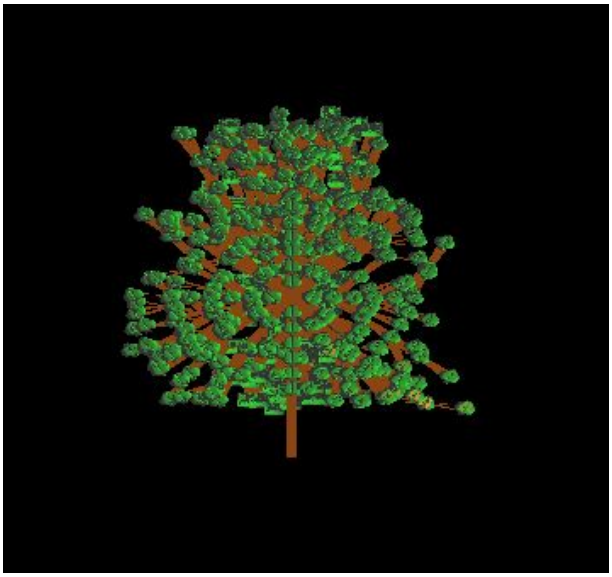


Figure 3: Tree Conformed to User-drawn Shape.

2 BACKGROUND

This project is related to prior work on procedural modeling of plants and trees, realistic tree modeling, modeling with particle flows, as well as user guided approaches, all of which are discussed within this section.

Procedural Modeling and L-systems. L-systems (Lindenmayer) were introduced by Aristid Lindenmayer in 1968 as a framework for studying the development of organisms [8]. For instance, simulating tree growth will ensure that relations amongst components are automatically maintained throughout the tree's structure [15]. L-systems closely mimic the growth and development of plants which can include decomposition rules to prune elderly branches [14]. There has also been research done that takes an inverse approach by taking a polygonal tree model as input and estimating the parameters using Monte Carlo Markov Chains to create a procedural model [21].

Realistic Tree Modeling. As already discussed, creating realistic tree models is an important aspect of procedural modeling. To make procedural modeled trees look more realistic, Weber and Penn created their tree structure out of cones [25]. Genetic algorithms can also be used in tandem with procedural modeling to control the look of a generated tree as discussed by Haubenwallner et al. [5]. To capture a more realistic tree model, procedural models have been created that take into account how real trees grow by removing 'dead' branches or those that no longer see the sunlight [13]. Neubert et al. pruned their models using the precision and recall algorithm [12].

There have also been image-based methods. Tan et al. used images of trees to create their tree structures [22]. Xie et al. used a library of three-dimensional living tree pieces to create software for users to design a large variety of tree models [26]. There have also been algorithms that determine how branches are spaced throughout

the tree's structure using space colonization algorithms [19].

Tree Modeling using Particle Flows. Xu and Mould use guiding vectors to determine branching patterns through a weighted graph [27]. Neubert, Franken, and Deussen take an input image of a tree and model the tree using direction fields and particle simulation [11]. Whereas, Kohek and Strnad create vast forests with varying level of detail by combining geometry-based and volumetric modeling techniques [7]. To create a more accurate animation of trees, Long and Jones place markers around trees to capture their movement and then reconstruct them by using particle flow [10].

User Interaction. Interactive tree modeling is becoming more popular due to its ability to create a wide range of trees with maximum control. There is software where users can use a tablet to draw the wanted tree structure and then use software to fill and 'grow' the tree [9]. Convolutional networks are also a popular choice for user-generated trees. Huang et al. created a convolutional network where the user inputs a two-dimensional drawing and the network produces a procedural model of a two-dimensional tree figure [6]. Sheeran et al. used virtual reality software to generate the tree structures and filled in the tree branches using a point cloud and edge generation methods [20]. Okabe et al. used two-dimensional sketches of trees to create a variety of three-dimensional tree models [16] [17]. Wither and Boudon created software for users to draw tree silhouettes and convert them into three-dimensional structures [24]. Anastacio et al. had a similar idea, they created a system for users to model plants and flowers from drawings of bounding lines [1].

3 TREE STRUCTURE GENERATION

This section details the background methods on how the basic tree and leaf structures were created. Once the basic tree is created, user interaction is added as well as updates to make the tree more realistic. All implementation was completed using C++ and the OpenGL shading language (GLSL).

3.1 Basic Trunk Methods

3.1.1 Tree Storage

The data structures that store the tree's main parts are arrays and stacks. The main arrays used are for branching points (`PT_ARRAY`) and for leaf points (`LEAF_ARRAY`). Two stacks are used to return to past branching points and headings within the pattern; the position stack and the heading stack.

3.1.2 Pattern Generation

The first step in creating a procedurally modeled tree is to generate a pattern to create the branching structure. This pattern consists of characters, where each character has a specific meaning drawing out the tree's structure. The characters used for the patterns in this paper include the letter F, left bracket, right bracket, plus, and minus. Table 1 below describes the meaning of each character.

F	Move forward a set distance
[	Save current position
]	Return to last saved position
+	Rotate the heading/direction to the left
−	Rotate the heading/direction to the right

Table 1: Character Descriptions.

Definition 1 (Axiom)**.** An axiom is a recognizable string that will be replaced using predefined rules and is the starting state for the pattern. [8]

Algorithm 1 Pattern Generation

Require: $pattern \leftarrow$ "F"

Require: $iterations \leftarrow$ number of times function is ran

Require: $newPattern \leftarrow$ empty string

```
for 0 to iterations do
    newPattern ← ""
    for idx ← 0 to len(pattern) do
        if pattern[idx] == F then
            newPattern += F[F][−F][+F]
        else
            newPattern += pattern[idx]
        end if
    end for
    pattern ← newPattern
end for
```

To interpret the pattern generated, a parser is created to determine the location of all of the points for the tree. The starting point for the parser is set, which in this example is Equation 1. The first character in this pattern is an F which leads to the second point created to complete the tree's trunk. Equations 1 and 2 are the starting state for drawing all of the trees for this project.

$$startPt = \begin{bmatrix} 0.0 \\ -0.25 \\ 0.0 \\ 1.0 \end{bmatrix}. \quad (1) \qquad currentHeading = \begin{bmatrix} 0.0 \\ 0.5 \\ 0.0 \\ 0.0 \end{bmatrix}. \quad (2)$$

Every time an F is encountered when parsing the pattern, a new line segment is added to the tree. To find the end point of this segment, the heading is added to the current position. After adding the vectors together, the new point is then saved in `PT_ARRAY`. The parser then continues to interpret the remaining characters in the pattern.

When [is interpreted, the parser saves the current position and heading. It does so by pushing the `currentPosition` onto the position stack, as well as pushing the `currentHeading` onto a separate heading stack. Saving this information allows the pattern to return to this point later when it has completed generating branches in the current section.

The parser interprets] as being the command to return to the last saved position. To complete the current branch, first, the parser saves `currentPosition` to `PT_ARRAY`, and then begins manipulating the stacks. The point on the top of the position stack is saved as `currentPosition`, and then is popped off. The same steps are undertaken for the heading stack, updating `currentHeading`.

The character − updates the direction of the current heading in the right direction. To update the heading direction, there is a 4×4 z-axis rotation matrix instantiated as shown in Equation (3).

$$rotZ(x) = \begin{bmatrix} \cos x & \sin x & 0 & 0 \\ -\sin x & \cos x & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

Using the notation from Equation 3, x is equal to the degree of rotation in radians. In this example, x is randomly generated within the range $0° \leq x \leq 65°$, and then converted to radians using the function $\frac{x \cdot \pi}{180}$. This value is added to the overall rotation value, then the variable x is updated within the rotation matrix,

and multiplied by `currentHeading` using a 4×4 matrix multiplication function to create a new heading vector. The magnitude of the new vector must be computed using:

$$\text{magnitude of heading} = \sqrt{x^2 + y^2 + z^2 + 0^2}. \tag{4}$$

The magnitude is then used to normalize the current vector. To normalize a vector, each value within the vector is divided by the magnitude. After normalization, the length of the heading vector will be equal to 1. Normalizing this vector allows the distance between each point to remain the same. After normalizing the vector, it is then saved as `currentHeading`.

The character + is used to update the heading direction negatively. The same steps are taken as were for the − character, but adding the new rotation value to the overall rotation value is changed to subtracting the new rotation value.

Once the parser has completed each character in the pattern, as shown in Figure 4, the points are arranged in the fashion shown in Figure 5.

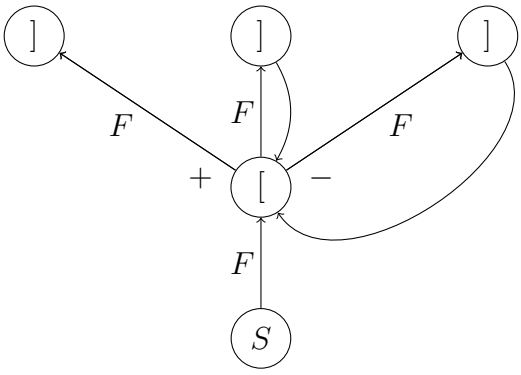


Figure 4: Pattern: $F[F][-F][+F]$.

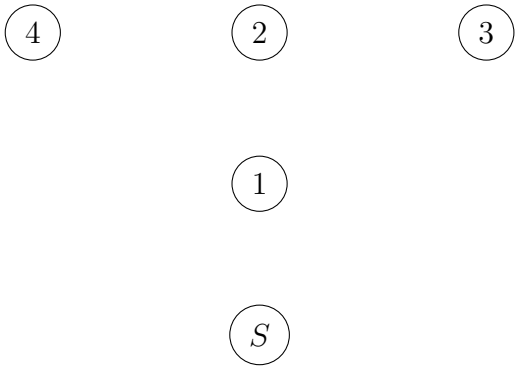


Figure 5: Arrangement of Points.

3.1.3 Graphics Pipeline

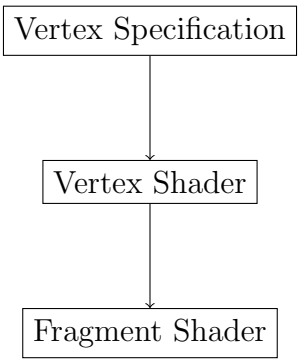


Figure 6: GLSL Graphics Pipeline.

The graphics pipeline, as shown above in Figure 6, is a sequence of steps required to transform raw vertex data into a 3D rendered scene. The first step is Vertex Specification, where the vertex array data is prepared [18]. This process is started by inputting the data to use into a vertex buffer object (VBO).

Definition 2 (Vertex Buffer Object)**.** An object that is used to store an array for OpenGL to use within a shader program.

The VBO will stores the `PT_ARRAY` that was created for the trunk and branches of the tree. A corresponding vertex array object (VAO) is also created.

Definition 3 (Vertex Array Object)**.** This object points to a VBO and informs the rendering window which VBO holds the data that is needed for a specific shader.

The VAO will point to the `PT_ARRAY` VBO. The next step is to create the vertex shader object as shown in Figure 6.

Definition 4 (Vertex Shader)**.** The vertex shader transforms vertex positions into the clipping space where everything is displayed. It is also used to transfer data to other shaders such as fragment or geometry shaders [4].

Each point in `PT_ARRAY` is passed to the vertex shader as a three dimensional vector, and as the position in the shader is being set, the vector updates from a three dimensional vector to a four dimensional vector to accurately represent a point. The next portion of the graphics pipeline is to create the fragment shader and color each point fragment.

Definition 5 (Fragment Shader)**.** The fragment shader sets the color, in RGB, for every fragment, or space in between vertices [4].

The purpose of the fragment shader is to set the color of the full surface to be rendered. This shader is passed individual fragments from the vertex shader. Colors are interpolated between vertices and lighting is applied to determine the final fragment colors and returned as output.

Once the shaders are linked to an executable shader program, it is time to draw the points to the window. The points are drawn to the window using `glDrawArrays` and `GL_POINTS`, which displays them from the `PT_ARRAY`.

What is displayed on the screen are points similar to Figure 5, and the next step is to connect the points into what looks like a tree trunk and branches. Simply changing `GL_POINTS` to `GL_LINE_STRIP` will create lines between each consecutive pair of points, as shown in Figure 7. Points S and 1 as well as 2 and 3 are connected, but it is necessary to connect points 1 and 2, 1 and 3, and 1 and 4 to create the correct structure.

To fix this, duplicate points need to be added to the `PT_ARRAY`. Each time F is parsed over, the current position is added to the array. Now, to fix the problem, each time a previous point is set as `currentPosition`, the revisited point needs to be added to the array again. So, each time] is parsed, the previous position is added to the array.

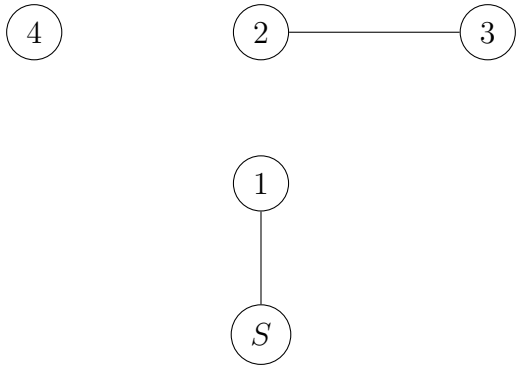


Figure 7: Incorrect Consecutive Points Tree Structure.

Making those adjustments will allow `GL_LINE_STRIP` to create fully-connected trees similar to Figures 8, 9, and 10 below. The `PT_ARRAY` now looks like {S, 1, 1, 2, 1, 3, 1, 4}, where each consecutive pair of points are connected by a line such as in Figure 8. Point 1 is added twice the first time it is parsed over to keep the pattern consistent. Without it, point 4 would not be connected to the rest of the tree.

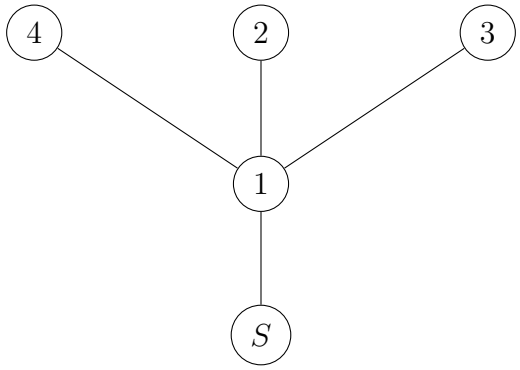


Figure 8: Correct Consecutive Points Tree Structure.

The next two figures are images of the trees that were created using these methods. Figure 9 is the tree after the third iteration of the pattern and Figure 10 is after the fourth iteration of the pattern.

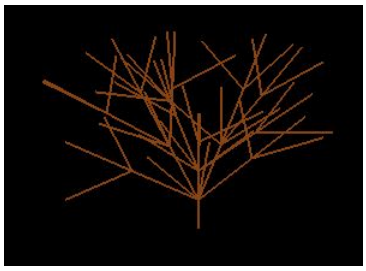


Figure 9: Tree with 3 Iterations.

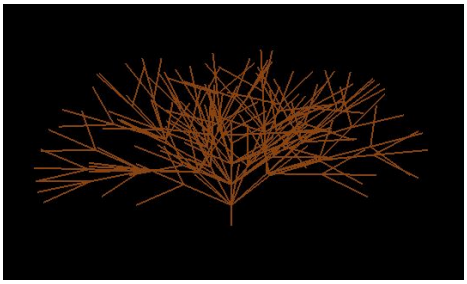


Figure 10: Tree with 4 Iterations.

3.2 Basic Leaf Methods

Next, to create a more realistic tree, leaves need to be added to the model. To create the leaves for the basic tree structure, the 3D modeling software Maya was used. Maya stores the color, vertex, face, and normal information within the exported OBJ file that will be used later [23]. Figure 11 is an image of the leaf model used for the basic tree structure.

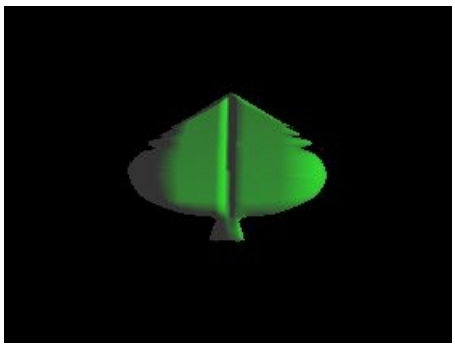


Figure 11: 3D Modeled Leaf.

After creating a leaf model, it now needs to be added to the basic tree model. The first step is to determine where the leaves are going to be placed on the branches. For this tree, the leaves are added to the end of the branches. For each point that is added to the `PT_ARRAY` by the F character, the same point will be added to a new array, called `LEAF_ARRAY`, for the purpose of placing leaves.

The second step is loading the leaf model into the program. The number of vertices and faces that the model has must be computed from the information within

the model, and then the number of face normals and vertex normals are computed to create arrays for each. Then, for each leaf in the tree, the vertices and vertex normals are added to their respective arrays to create all of the leaves needed on the tree. Once the locations of the leaves are known, they are then rotated, scaled, and translated to the correct size, direction, and location, respectively.

The last step is generating the vertex and fragment shaders to display the leaves on the screen. Similar to what was created for the branches as described in Section 3.1, there is a vertex shader that passes the position of each leaf to the rendering window as well as a fragment shader that determines the color and lighting for each leaf.

This leaf vertex shader includes everything found in the branch's vertex shader, except it is using the `LEAF_ARRAY` instead of the `PT_ARRAY`. The fragment shader, on the other hand, uses the Phong Model to illuminate the leaves. Before the Phong model is added in, the tree looks like those in Figures 12 and 13.

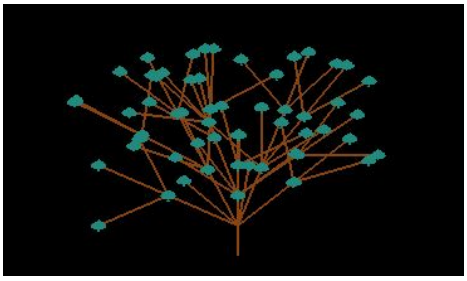


Figure 12: Tree with 3 Iterations Unlit.

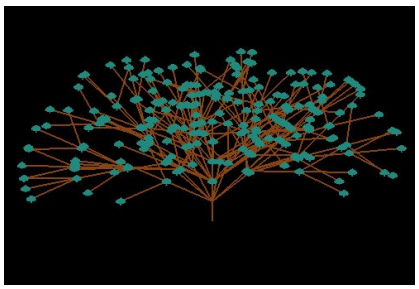


Figure 13: Tree with 4 Iterations Unlit.

Definition 6 (Phong Model)**.** The Phong Model is the sum of three types of light reflection, which includes diffuse lighting $\{I_d\}$, specular lighting $\{I_s\}$, and ambient lighting $\{I_a\}$. After calculating the three types of light reflection, they are then added together to create the final light intensity $\{I\}$, whose formula is $I = I_d + I_s + I_a$ [4].

Definition 7 (Diffuse Lighting)**.** The light that is scattered by the roughness of surfaces [4].

Definition 8 (Specular Lighting)**.** The highlights that reflect from a surface [4].

Definition 9 (Ambient Lighting)**.** The general lighting that is not direct [4].

The leaf model created was originally green. Instead of coloring the leaf with a flat color, the Phong model is applied to add more detail and make the tree look more realistic. The first step is to determine where the light source will be relative to the tree model. In this example, the light source is on the right-hand side at the location shown in Equation (5) as well as the circle shown in Figure 14.

$$\text{Light location} = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.0 \end{bmatrix}. \tag{5}$$

Figure 14: Location of the Light Source in Relation to the Tree Window.

The light source also has attributes such as the color of the diffuse, specular, and ambient lighting, or L_d, L_s, and L_a respectively. For this model, the diffuse lighting component will be a dull white, the specular light will be white, and the ambient light will be gray.

The next few properties that need to be determined are the amount of light that the leaf's surface is going to be reflecting from each component. For the specular and ambient components, or K_s and K_a respectively, all of the light is going to be

reflected from the surface. However, for the diffuse component, K_d, the only light that needs to be reflected is the green channel to have green leaves. For fall foliage, a combination of red and green values are reflected as shown in Figures 15 and 16.

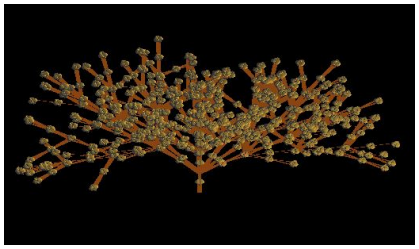


Figure 15: Tree with Orange Leaves.

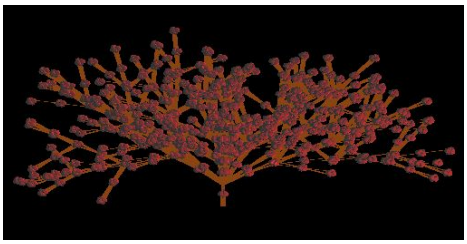


Figure 16: Tree with Red Leaves.

To create the ambient lighting component, the color and reflecting amount need to be multiplied together, $I_a = L_a \cdot K_a$.

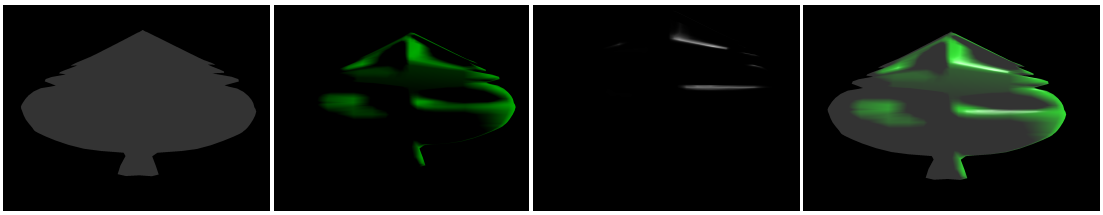
The diffuse component is at its brightest when the surface is directly facing the light source and at its darkest when the surface is perpendicular to the light source. The normal is the current facing direction of the surface, so to get the brightness factor, first the light's location and the surface normal need to be subtracted and normalized to get the distance vector. Then, the dot product, or DOT_PROD, is computed from the surface normal and the distance vector. Finally, the color, reflecting amount, and dot product are multiplied together to create the diffuse lighting component, $I_d = L_d \cdot K_d \cdot DOT\_PROD$.

The last component, specular, is light that is reflected from around the surface normal. Getting the specular component is almost the same as determining the diffuse component, but the viewing angle is factored in. To get the viewing angle, the distance vector needs to be subtracted from the camera's position and then normalized. The DOT_PROD_SPEC is computed from the viewing angle and the area where the light will be reflected from. The dot product is raised to the power of 100, or SPEC_POW, and finally, the specular component can be determined by $I_s = L_s \cdot K_d \cdot SPEC\_POW$.

With all of these components generated, the leaves will no longer look dull and flat once they are added together. The final lighting of the tree will come from the vector

$$\begin{bmatrix} I_a + I_d + I_s \\ 1.0 \end{bmatrix},$$

which will make the tree look more realistic to the viewer. Figure 17 depicts the Phong model in each of its different stages. Figure 17a is the ambient lighting, Figure 17b is the diffuse lighting, Figure 17c is the specular lighting, and Figure 17d is all of the components (Figures 17a - 17c) added together. Figures 18 and 19 are images of the example tree after the lighting model was incorporated.



(a) Ambient (b) Diffuse (c) Specular (d) Phong Model

Figure 17: Phong Lighting Model Components.

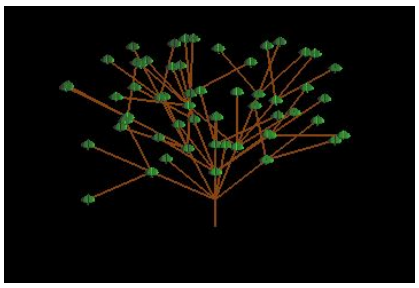


Figure 18: Tree with 3 Iterations Lit.

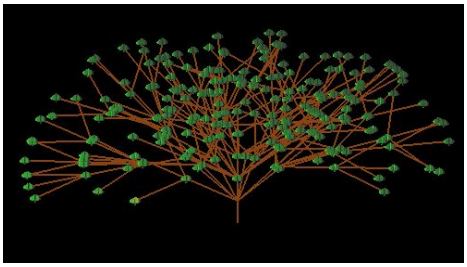


Figure 19: Tree with 4 Iterations Lit.

3.3 User Interaction

User interaction was added to the tree model so users could rotate and translate the tree as well as for debugging purposes. First, all of the matrices that will be used to translate and rotate the tree must be initialized. These matrices include the

translation, x-axis rotation, y-axis rotation, and z-axis rotation matrices as shown in Equations (6) - (9), respectively where x, y, z denote translation values or rotation angles.

$$\text{Translation} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6}$$

$$rotX(x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos x & -\sin x & 0 \\ 0 & \sin x & \cos x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

$$rotY(x) = \begin{bmatrix} \cos x & 0 & \sin x & 0 \\ 0 & 1 & 0 & 0 \\ -\sin x & 0 & \cos x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{8}$$

$$rotZ(x) = \begin{bmatrix} \cos x & -\sin x & 0 & 0 \\ \sin x & \cos x & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{9}$$

For the program to be responsive, key listeners are added for each movement and its inverse. The key listeners will be waiting for a specific key on the keyboard to be pressed, and when it is pressed the variable corresponding to a specific matrix is then increased or decreased. While the tree is being displayed, when a key is pressed, the variables within the matrix are updated, and the matrices are then multiplied

together to rotate the tree.

3.4 Creation of a More Realistic Tree

To create a more realistic tree, additions were made to vary the thickness of the trunk and branches, generate more leaves on branches, update lighting, and add depth to make the tree three dimensional.

3.4.1 Thicker Trunk and Branches

To create a thicker trunk and branches, a geometry shader is used to transform the lines into triangles. This shader is placed in between the vertex and fragment shaders within the graphics pipeline as shown in Figure 20 below.

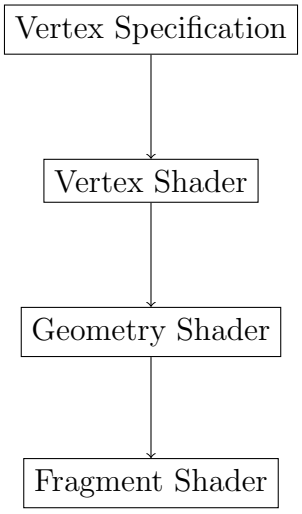


Figure 20: Updated GLSL Graphics Pipeline.

Definition 10 (Geometry Shader)**.** A geometry shader is an optional shader that is placed between the vertex and fragment shaders. It takes in a set of vertices of a single primitive (e.g., a triangle or point) as input and transforms the primitives into a different primitive using more vertices than were initially given [3].

Since what is being displayed on the screen are strips of lines, the input to the

geometry shader will be lines and the output will be strips of triangles. Four vertices, or two triangles, are going to be generated around each point as shown in Figure 21. The four vertices are generated by taking the original point and adding an offset value such as in Algorithm 2.

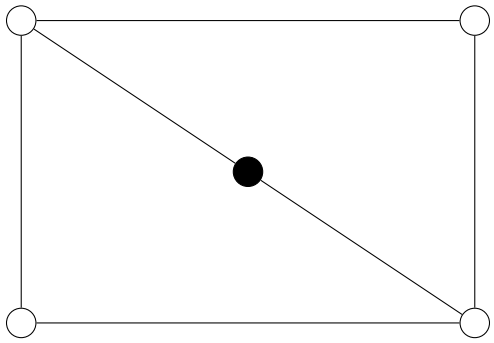


Figure 21: Triangle Strip Structured around Singular Point.

Algorithm 2 Adding Offset Values to Geometry

Require: $num \leftarrow$ a value to be added or subtracted
$bottomRightPoint \leftarrow currentPosition + [num, -num, 0, 0]$
$bottomLeftPoint \leftarrow currentPosition + [-num, -num, 0, 0]$
$topRightPoint \leftarrow currentPosition + [num, num, 0, 0]$
$topLeftPoint \leftarrow currentPosition + [-num, num, 0, 0]$

Figure 22 depicts the tree structure before the geometry shader transforms the lines into quads. Figure 23 depicts the tree after the quads are generated.

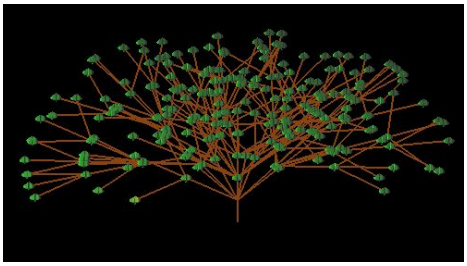


Figure 22: Before Geometry Shader.

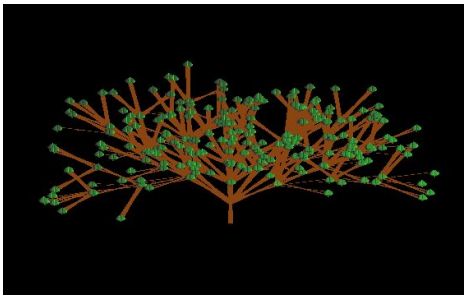


Figure 23: After Geometry Shader.

3.4.2 Generating More Realistic Leaves

Currently, the leaves are located at the end of the tree branches and are all uniformly rotated as shown in Figure 23. To generate a more realistic tree, more leaves are added as well as rotated to random angles.

The first step is adding more leaves to the branches. When the character of the pattern is F, more points for leaves are generated by taking the midpoint of the current and previous leaf positions, as in Algorithm 3. The midpoint is then added to the `LEAF_ARRAY`, and the tree generated has twice the number of leaves than the previous ones. Figure 24 displays the fuller, more realistic tree that was generated.

Algorithm 3 Calculate the Midpoint

$midpointX \leftarrow (lastPositionX + currentPositionX)/2$

$midpointY \leftarrow (lastPositionY + currentPositionY)/2$

$midpointZ \leftarrow (lastPositionZ + currentPositionZ)/2$

$midpoint \leftarrow [midpointX, midpointY, midpointZ, 0]$

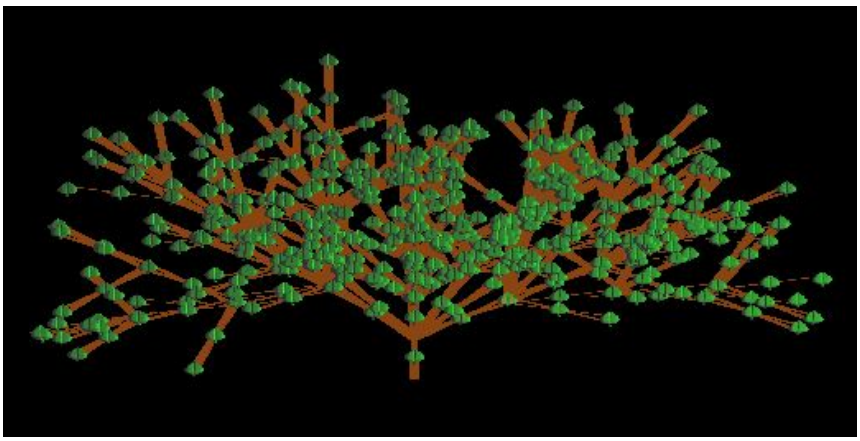


Figure 24: Tree After Adding Leaves.

The next step is to rotate the leaves to look more natural. This is done during the step where all of the leaves are translated, rotated, and scaled. First, two random numbers are generated to rotate the leaves by, one positive and one negative. These

numbers are then converted to radians, and used to rotate the leaf as shown in Algorithm 4.

Algorithm 4 Rotating Leaves

Require: $Z\_Mat\_X \leftarrow$ Z rotational matrix's x value as shown in Figure 3

Require: $posNum \leftarrow$ positive random number

Require: $negNum \leftarrow$ negative random number

Require: $X\_Coord \leftarrow$ value of leaf's X coordinate

if $X\_Coord > 0$ **then**

$Z\_Mat\_X \leftarrow posNum$

else if $X\_Coord == 0$ **then**

$Z\_Mat\_X \leftarrow 0$

else

$Z\_Mat\_X \leftarrow negNum$

end if

Once the z rotational matrix is updated, and the position of the leaf is multiplied by all of the translation, rotation, and scaling matrices, the tree generated will look like Figure 25.

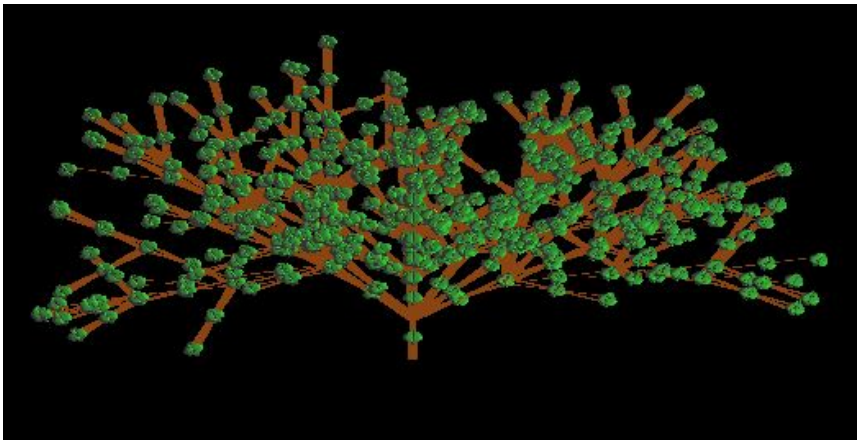


Figure 25: Tree After Leaf Rotation.

3.4.3 Adding Depth

The next step is to add depth to the tree. So far, the tree has only spanned two dimensions. When the user rotated the tree, at some point during the rotation, the tree became flat as depicted in Figures 26 and 27 below. Adding a third dimension to the model will allow the user to view the tree at any angle, like a living tree.

To add depth, the tree needs to be growing three dimensionally. This means that the z coordinate in the position vector needs to be updated. Every time a new branch is created, a random number generator is used to determine the new z component of the current branch heading. This is executed before the `currentHeading` is added to the `currentPosition`.

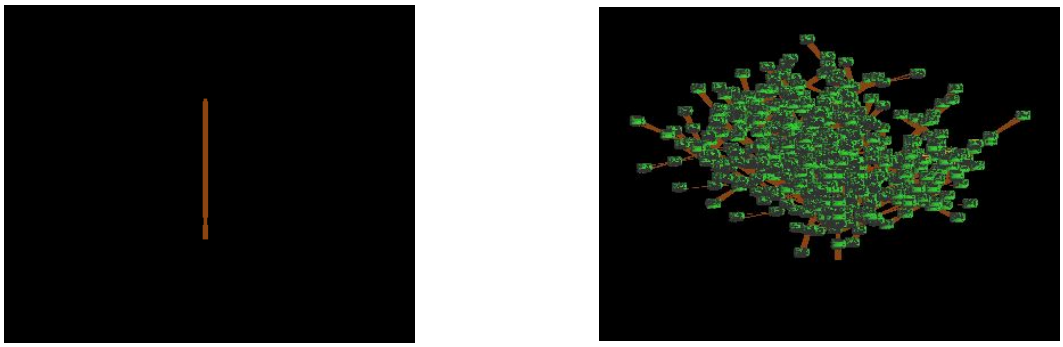


Figure 26: Tree Side-view Before and After Adding Depth.

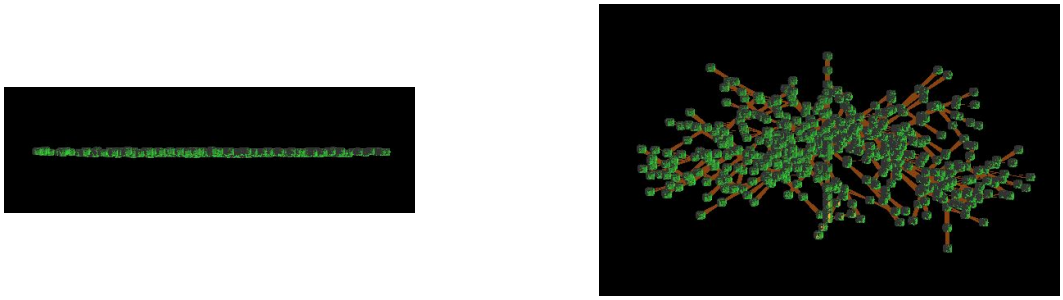


Figure 27: Tree Bottom-Up View Before and After Adding Depth.

4 SHAPE DEFINITION

This section details the main portion of this project which includes conforming the tree to three-dimensional shapes such as cubes, spheres, and user-drawn shapes. The CImg library was incorporated to complete this portion of the project.

4.1 Containing Shapes

To begin the user-created tree portion of this project, there needs to be the ability to constrain the branches of the tree structure within certain parameters. This subsection explains the methods behind 'growing' the tree into certain three-dimensional shapes starting with simpler geometry, such as cubes and spheres, and ending with more complex, user-defined shapes.

4.1.1 Cube

The first step is to start with a cube's two-dimensional equivalent, a square. To do this, the size of the cube needs to be defined (`SIZE`). In this project, the `SIZE` is defined as a float smaller than 1 but larger than 0.1. During the pattern parsing stage, as defined in Section 3.1.2, a flag is created when the current position is larger than `SIZE` or smaller than `-SIZE`. This flag is tripped before the current position is added to the `PT_ARRAY`.

Once the flag is tripped, the current position is saved, as well as the tree's origin. Within this paper, the origin of the tree is shown in Equation (10).

$$origin = \begin{bmatrix} 0.0 \\ 0.85 \\ 0.0 \\ 0.0 \end{bmatrix}. \tag{10}$$

After these things are saved, the current position is then manipulated. If the current position is greater than `SIZE`, then the origin (scaled by a factor of 0.3 to reduce the branch length) is subtracted until the current position is within the `SIZE` parameter. Then, the origin (scaled by a factor of 0.01 to increase the branch length by a small factor) is added to the current position until it reaches the square's boundary.

When the current position is smaller than `-SIZE`, then the same steps are taken, just inverted. Instead of subtracting the origin, it is added, and when it is added, the origin is subtracted as shown in Algorithm 5. As for a cube, the same steps are taken, but the z-coordinate is taken into consideration instead of only the x and y coordinates.

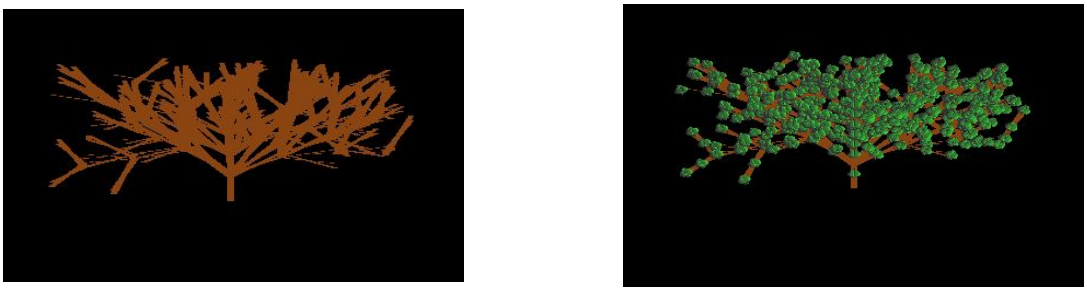


Figure 28: Tree Constrained in a 0.5 Cube.

Algorithm 5 Cube Constrain

Require: $lastPos \leftarrow$ Save current position

Require: Save origin point

Require: $size \leftarrow$ size of the cube to be created

  if current position $>$ $size$ **then**

    while current position $>$ $size$ **do**

      current position $\leftarrow$ current position $-$ origin

    end while

    while current position $<$ $size$ **do**

      current position $\leftarrow$ current position $+$ origin

    end while

  else

    while current position $<$ $-size$ **do**

      current position $\leftarrow$ current position $+$ origin

    end while

    while current position $>$ $-size$ **do**

      current position $\leftarrow$ current position $-$ origin

    end while

  end if

Require: current position $\leftarrow$ $lastPos$

4.1.2 Sphere

To grow the tree into a sphere, instead of a `SIZE` parameter, there is a `RADIUS` parameter. The radius determines how large of a circle or sphere will be used to shape the tree.

`RADIUS` is defined as a float between 0.1 and 1.0. The general equation of a sphere is used to determine whether the current position is outside of the sphere, as shown

in Equation (11). The same flag that was created for the cube constraining portion is tripped when the current position is outside of the `RADIUS`. Once this happens, the current position is saved and the origin is determined.

$$X^2 + Y^2 + Z^2 = R^2. \tag{11}$$

After the flag is tripped, a new position is created (`newPt`) to retract or expand the branches back within the sphere's radius. The `newPt` is created by taking the `currentPosition` and subtracting the `origin`.

Using Equation (11) within Algorithm 6, we can determine whether the branch is outside the circle using the current position of its endpoint. If $X^2 + Y^2 + Z^2$ is greater than R^2, then the endpoint is outside of the sphere. While `currentPosition` is outside of the circle, the new position is scaled and subtracted from it until it is within the radius.

Once `currentPosition` is within the radius and the z-coordinate is greater than 0, then `newPt` is added to `currentPosition` until the branch reaches the radius of the sphere. Then, `currentPosition` is added to the `PT_ARRAY` and `currentPosition` is added to the `LEAF_ARRAY`. Figure 29 below depicts a tree whose radius is 0.5.

Figure 29: Tree Constrained in a 0.5 Sphere.

Algorithm 6 Sphere Constrain

Require: $lastPos \leftarrow$ Save current position
Require: Save origin point
Require: $radius \leftarrow$ radius of the sphere to be created
Require: $newPt \leftarrow$ current point - origin point
  if current position$^2 > radius^2$ **then**
    while current position$^2 > radius^2$ **do**
      current position $\leftarrow$ current position $- newPt$
    end while
    while current position$^2 \leq radius^2$ **do**
      current position $\leftarrow$ current position + origin
    end while
  end if
Require: current position $\leftarrow lastPos$

4.1.3 User-drawn Shapes

To fully customize the shape of the tree, the user also has the option of drawing a silhouette shape. A window is created for the user to draw a closed, convex shape around an origin point using the mouse as shown in Figure 30 below.

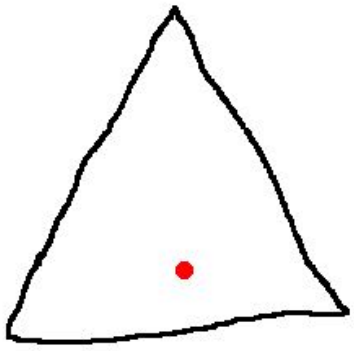


Figure 30: User Drawn Shape.

Next, we need to determine what is inside the shape and outside of the shape. For this step, the scanline algorithm as shown in Algorithm 7 is implemented. The output of the scanline algorithm is used to determine which branches should be pruned.

Algorithm 7 Scanline Algorithm

```
for pixel in image do
  if inside shape then
    make pixel black
  else
    make pixel white
  end if
end for
```

Once the algorithm is run, the user-drawn shape is then filled in as depicted in Figure 31.

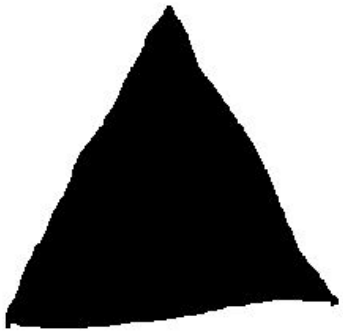


Figure 31: Filled Shape.

To make the tree conform to the shape, we now need to determine whether the points of each branch are inside or outside of the shape. Since the size of the image is larger than the resolution of the space containing the tree's vertex data, the image needs to be scaled to the tree's coordinate plane to accurately prune the branches. For this project, the image size is 640 by 320 pixels and the tree is on a -1 to 1 coordinate plane.

$$x = 320x + 320. \tag{12}$$

$$y = -240y + 240. \tag{13}$$

To determine whether the branching point is within the shape, Equations (12) and (13) are used to convert from the image to the tree plane. This conversion gives the ability to check whether the branching point on the tree plane is a black or white pixel within the image and prune the branches as shown in Algorithm 8.

Algorithm 8 Pruning Algorithm

```
img ← image plane
branchX ← the X position of the branch
branchY ← the Y position of the branch
xPosition ← (320 · branchX) + 320
yPosition ← (−240 · branchY) + 240
for branching point do
  if img[xPosition][yPosition] == black then
    save branching point
  else
    remove branching point
  end if
end for
```

The tree shown below in Figure 32 has been conformed to the shape as previously displayed in Figure 31.

Figure 32: Shape Conformed Tree.

4.1.4 Refining Tree Conforming

The final stage of this project is to refine how the final user-drawn tree looks. Since the leaves in Figure 32 are in a uniform circular pattern, random z-coordinate values were incorporated into the two-dimensional tree to vary the leaf placement as shown in Figure 33.

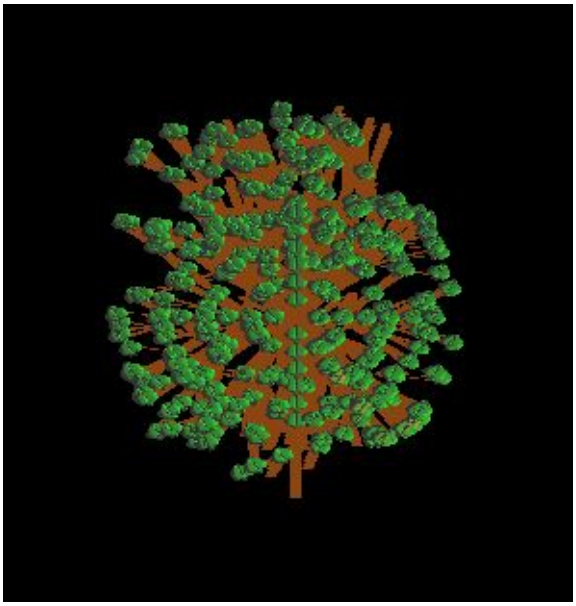


Figure 33: Minor Leaf Variation.

As compared to Figure 32, the leaves around the edge of the tree are less uniform within Figure 33. This makes the tree look more realistic, which is the goal of this stage. The next step is to make the tree three-dimensional again by incorporating a duplicate tree model that is rotated around the y-axis by 90°.

The first step in rotating a duplicate tree model is to create another set of branching points that are rotated. This is done by duplicating the `PT_ARRAY` and then multiplying each point in the array by the y-axis rotation matrix as shown in Equation (8).

Then, once all of the points are rotated, another shader program that consists of a vertex, geometry, and fragment shader needs to be created. This is done by linking

the shader program to the vertex, geometry, and fragment shaders from Section 3.1 and using the new rotated point array (`PT_ARRAY_ROT`) instead of `PT_ARRAY`. The tree with another leafless cross-section is displayed in Figure 34.

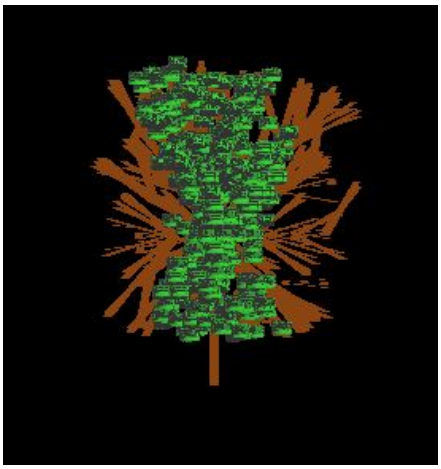


Figure 34: Branch Cross-Sections.

Figure 35: Leaf Cross-Sections.

Next, another set of leaves is rotated to match the new branch cross-section. This is accomplished by creating another shader program that consists of the vertex and fragment shaders from the first set of leaves as described in Section 3.2. Then, the leaves themselves need to be rotated to match the branches. If the user is drawing a shape, then a new matrix will be created to rotate each leaf 90°by multiplying each point by the y-rotation matrix. This new matrix is then attached to the new leaf shader program and creates the new leaves. Figure 35 is now a fully three-dimensional tree that conforms to a user-specified shape. To ensure that the branches of the rotated tree are within the boundaries of the shape, the z coordinate of the rotated tree is compared to the x coordinate of the original tree. If the z coordinate is larger than the x coordinate, then the point is pruned from the tree.

To make the tree conform to the shape over 360°, another tree model was rotated 90°and added into the scene using the existing shader programs. Figures 36 and 37 are the shape that was drawn to create the tree in Figures 38 and 39. Figure 38

depicts the tree before rotation and Figure 39 shows the tree after rotating 90°.

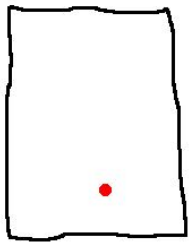


Figure 36: User-drawn Shape.

Figure 37: Filled Shape.

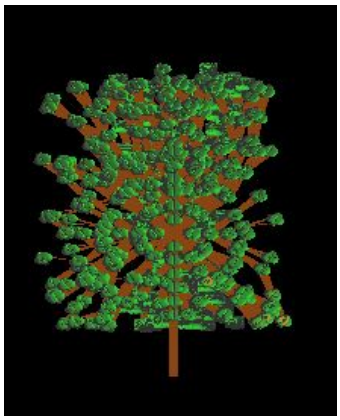


Figure 38: Front of Tree.

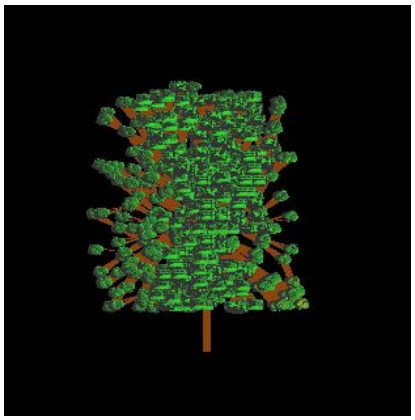


Figure 39: Side of Tree.

5 CONCLUSION

Procedural modeling of trees can quickly yield a wide variety of differing structures by closely adhering to patterns and rules observed in nature. This paper describes a system that involves generating unique, realistic tree models with simple user specifications such as sketches of shapes.

The first component of this project was to create a realistic tree structure using GLSL by incorporating natural lighting, varying branch thickness, and natural randomness in the model. This was done by integrating a geometry shader into the graphics pipeline that generates quads around each branching point to make the branches thicker. The Phong lighting model was added to the leaves and branching structure for the realistic lighting portion of this project. Finally, randomness was included when rotating the branches for added naturalness.

The second component includes incorporating user input to constrain the tree structure to drawn shapes. The CImg library is included to generate a canvas for the user to sketch their shape. Once the user sketches their shape, the scanline algorithm is used to determine the inside and outside of the shape. Then the tree structure is constrained to the inside of the shape by determining whether the branching point is inside or outside of the shape boundaries. The tree is made three-dimensional by incorporating another tree rotated along the y-axis by 90°. It is then further refined by ensuring that the rotated tree is within the shape's boundaries as well.

5.1 Future Work

This project can be expanded by generating a forest of user-constrained trees rather than a singular tree structure. Furthermore, creating distinct species of trees for the user to choose from with different leaf models and branching patterns will be an important improvement to this project.

REFERENCES

[1] F. Anastacio, M. Sousa, F. Samavati, & J. Jorge, Modeling Plant Structures Using Concept Sketches, *NPAR*, 2006.

[2] H. Dale, A. Runions, D. Hobill, & P. Prusinkiewicz, Modelling biomechanics of bark patterning in grasstrees, *Annals of Botany*, 114(4):629-641, 2014.

[3] J. de Vries, Geometry Shader, *Learn OpenGL, Extensive Tutorial Resource for Learning Modern OpenGL*, June 2014, learnopengl.com/Advanced-OpenGL/Geometry-Shader.

[4] A. Gerdelan, *Anton's OpenGL 4 Tutorials*, 2014.

[5] K. Haubenwallner, H. P. Seidel, & M. Steinberger, ShapeGenetics: Using Genetic Algorithms for Procedural Modeling, *Computer Graphics Forum*, 36(2), 2017.

[6] H. Huang, E. Kalogerakis, E. Yumer, & R. Mech, Shape Synthesis Sketches via Procedural Models and Convolutional Networks, *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[7] Š. Kohek & D. Strnad, Interactive large-scale procedural forest construction and visualization based on particle flow simulation, *Computer Graphics Forum*, 2018.

[8] A. Lindenmayer & P. Prusinkiewicz, *The Algorithmic Beauty of Plants*, Springer-Verlag, New York, NY, 1996.

[9] S. Longay, A. Runions, F. Boudon, & P. Prusinkiewicz, TreeSketch: Interactive Procedural Modeling of Trees on a Tablet, *Eurographics Symposium on Sketch-based Interfaces and Modeling*, 2012.

[10] J. Long & M. Jones, Reconstructing 3D tree models using motion capture and particle flow, *International Journal of Computer Games Technology*, 2013.

[11] B. Neubert, T. Franken, & O. Deussen, Approximate image-based tree-modeling using particle flows, *ACM SIGGRAPH*, 2007.

[12] B.Neubert, S. Pirk, O. Deussen, & C. Dachsbacher, Improved Model- and View-Dependent Pruning of Large Botanical Scenes, *Eurographics Symposium on Geometry Processing*, 2011.

[13] S. Pirk, T. Niese, O. Deussen, & B. Neubert, Capturing and Animating the Morphogenesis of Polygonal Tree Models, *ACM Transactions on Graphics*, 31(6):169, 2012.

[14] P. Prusinkiewicz, M. Cieslak, P. Ferraro, & J. Hanan, Modeling plant development with L-systems, *Mathematical Modelling in Plant Biology*, Springer, 2018.

[15] P. Prusinkiewicz, M. Shirmohammadi, & F. Samavati, L-systems in Geometric Modeling, 2010.

[16] M. Okabe, S. Owada, & T. Igarashi, Interactive Design of Botanical Trees using Freehand Sketches & Example-based Editing, *Eurographics*, 24(3), 2005.

[17] M. Okabe & T. Igarashi, 3D Modeling of Trees from Freehand Sketches, *SIGGRAPH*, 2003.

[18] OpenGL Wiki Contributors, Rendering Pipeline Overview, *OpenGL Wiki*, February 2017, https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[19] A. Runions, B. Lane, & P. Prusinkiewicz, Modeling Trees with a Space Colonization Algorithm, *Eurographics Workshop on Natural Phenomena*, 2007.

[20] B. Sheeran, K. Endo, P. Ricaud, & V. Ramanujan, willowGAN: Neurally-aided 3D VR sketch-to-object tree modeling.

[21] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Mech, O. Deussen, & B. Benes, Inverse Procedural Modeling of Trees, *Computer Graphics Forum*, 33(6), 2014.

[22] P. Tan, G. Zeng, J. Wang, S. Kang, & L. Quan, Image-based Tree Modeling, *ACM Transactions on Graphics*, 26(99), 2008.

[23] Wavefront OBJ File Format Summary, *Encyclopedia of Graphics File Formats*, O'Reilly, March 2019, www.fileformat.info/format/wavefrontobj/egff.htm.

[24] J. Wither, F. Boudon, M.P. Cani, & C. Godin, Structure from Silhouettes: A New Paradigm for Fast Sketch-based Design of Trees, *Computer Graphics Forum*, 28(2), 2009.

[25] J. Webber & J. Penn, Creation and Rendering of Realistic Trees, *Proceedings of the* 22^{nd} *Annual Conference on Computer Graphics and Interactive Techniques*, pp. 119-128, 1995.

[26] K. Xie, F. Yan, A. Sharf, O. Deussen, B. Chen, & H. Huang, Tree Modeling with Real Tree-Parts Examples, *IEEE Transactions on Visualization and Computer Graphics*, 2016.

[27] L. Xu & D. Mould, Procedural Tree Modeling with Guiding Vectors, *Computer Graphics Forum*, pp. 47-56, 2015.