PHYSICALLY BASED TREE RENDERING

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

_____

Jackson Fields

Date: April 26, 2012

Approved:

_____

Professor Emmanuel Agu, Major Advisor

Abstract

This project produced and rendered physically based models of several tree species in real time. This was accomplished by procedurally generating a branch hierarchy and leaves within some bounding volumes according to rules that define a tree species. Multiple trees can be rendered at interactive frame rates greater than 45 frames per second. This enables the production of more realistic forests in games and visual applications procedurally and also frees up disk space for other resources because large model files of trees do not need to be stored on disk.

Acknowledgements

I would like to thank Professor Emmanuel Agu for advising this project, providing valuable advice and direction for the project, and meeting with me weekly for the duration of the project.

I would also like to thank WPI's Interactive Media and Game Development and Computer Science departments for providing the education and resources that provided a foundation to complete this project.

Table of Contents

List of Figures

Executive Summary

Trees are an essential element of a natural landscape. They have been present in games since the early days of game development, but these early trees are often cheap imitations, vague suggestions of trees, or blatantly reused models, distracting from an immersive experience.

This project focuses on procedurally creating realistic looking trees that are based on physical rules for particular tree species. This allows for a game to have an arbitrary number of convincing looking trees of any species without using excess disk space for a model. This also frees up an artist's time for tasks that might be more important to the game than the creation of trees.

To accomplish this, a tree generation program separates a tree into two fundamental components: a tree of branches and volumes of leaves. Rules that define the construction of these two components are used to create a convincing looking hierarchy of branches, then leaves are produced on these branches to fill out the leaf bounding volumes. By varying the tree rules parameters, different species of trees (either naturally-occurring or fictional) can be created.

1 - Introduction

Trees and foliage are elements that visually enhance natural landscapes. As such, inclusion in a game setting can provide added realism and might be expected in certain environments. For example, an outside scene with a barren terrain might seem out of place, but adding proper foliage could be make the scene appear more realistic.

A typical outdoor game scene will include a terrain with regions partitioned with different geographic features. For instance, there might be mountainous areas separated by valleys, large open plains, rivers, or even oceans. Each area might be textured differently, and could contain unique foliage that makes the area distinct. Some games even include large forests that simulate actual real-world forests to increase the player's immersion in the game.

There are many problems with generating organic elements in a game such as production time, rendering time, or the size of the final model file. The game development cycle operates under very tight deadlines to ensure a game is shipped on time, often for a holiday season. If a forest of unique trees is required in a game, the game's artists will be required to produce all of the tree models, as well as the rest of the game's art. Once all of these models are produced, they need to be rendered in the game, and saved on disk. The tree models need to be created at a low enough polygon count to be rendered at interactive frame rates, but a high enough detail to look good. Additionally, model formats store each vertex and edge in a file which might be very large depending on the complexity of the model. The final size of the game must be less than the size of the DVD or other storage medium on which the game will be shipped on, so including too many models might not be possible.

1.1 – The Goal of this MQP

This project focuses on producing realistic trees that can be rendered on-the-fly in real time without storing large model files. Trees can be procedurally generated or created at run time using some rules to define how they are to be constructed. This automates the entire tree generation process, so artists can focus on other aspects of a game. Additionally, the level of detail, or the number of polygons the tree requires, can be scaled down based on the generation rules, which can improve rendering speed. Since trees can be generated at run time using procedural methods, the tree does not need to be stored in a large model file, so disk space can be used for other features of the game.

2 - Background

There are many different ways to render trees. Some current techniques involve treating the branching structure of the tree as a self-similar fractal, generating the branches using input rules, and defining the leaves of a tree as texture lobes. These methods will be explained in the following sections.

2.1 – Trees in Games

Rendering trees in games can be a resource intensive process. Over the years, different methods for creating and rendering trees in games have been used. To illustrate this, consider the following brief and incomplete history of trees in games that I arbitrarily generated using games selected for their tree rendering techniques.

Nintendo's 1992 (IGN, 2012) game *A Link to the Past* is one of many examples of how trees can be depicted in a 2D game. While this project focuses on 3D tree generation, the 2D game was the precursor to 3D gaming, so it is important to show where trees in games originated. In Figure 1 you can see a few isometric tree sprites on the left and a few shrubbery sprites on the top right. These sprites add to the game environment and are level design features acting as obstacles that block player movement.

Figure 1 Screenshot from *A Link to the Past* showing tree and foliage sprites (IGN, 2007)

The same year (Giant Bomb, 2012), ID Software released the 3D game *Wolfenstein 3D*, which includes occasional potted plants scattered throughout the castle the game player is exploring (Figure 2). These plants are just billboards, or flat textures, which rotate along the up axis to always face the player. Rendering is still expensive at this time, so techniques like these add additional elements to the screen without putting too much strain on the computer.

Figure 2 Screenshot from *Wolfenstein 3D* showing billboard trees (Orr, 2011)

A few years later in 1998 (Zeldapedia, 2012), when gaming consoles and 3D graphics became slightly more mature, Nintendo released another Zelda game, *Ocarina of Time* for the Nintendo 64. Figure 3 shows the Deku Tree as you see it in the game. The tree has a very simple trunk geometry and a simple green volume for the leaves. It does effectively portray a tree, but there is limited detail in the construction of the tree, again largely due to limitations in how many triangles can be rendered on screen in real time.

Figure 3 Screenshot of *Ocarina of Time* showing the Deku Tree (Zeldapedia, 2012)

In 2002, Bethesda Game Studios released a large-scale role-playing game, *The Elder Scrolls III: Morrowind* (IGN 1, 2012). This game features large landscapes with different terrain and foliage. Figure 4 shows one scene from the game with trees somewhat resembling those one might find in the real world. These trees have branching trunks and some even have leaves. These leaves, however, are just large flat textures that grow out of the end of the trunk without being connected by any geometry. *Morrowind* was able to render many unique trees in the environment. This was probably accomplished by creating many different tree models and placing them individually in the world.

Also of note is the distinction between rock, road, and grass. Rather than having supplemental models for the grass which would be very expensive to render, the grass is just the same terrain texture but colored green, while the road is brown and the rocks are gray.

Figure 4 Screenshot of *TES III: Morrowind* showing multiple trees (Pullin, 2002)

One technique for drawing many trees in an environment is called instancing, and involves recycling the same tree model and rendering it many times to create a forest of the same tree. This concept was illustrated previously with the same billboarded tree sprite used many times in *Wolfenstein 3D* and is also shown using 3D models in Figure 5. The forest in this scene is made up of just one model rotated in various ways.

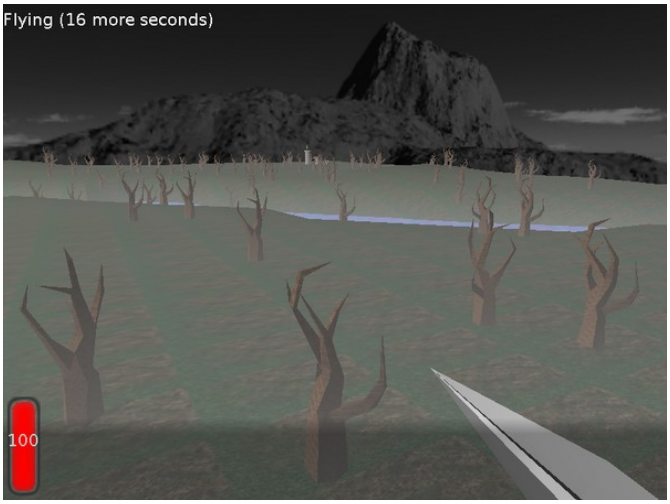

Figure 5 Screenshot of a VRML Engine showing model reuse (Michaelis, 2006)

While exaggerated in the previous example, reusing models is a viable way of rendering many trees in the same space. This is because models can be very large, so they take up space on disk and in memory. Especially when shipping a game on a DVD with finite space, any way to save space is beneficial. 2007's *Crysis* (vgReleases, 2012), includes very dense forests such as the one pictured in Figure 6. This is probably the most convincing forest out the all of these examples, and is definitely an immersive experience when run at interactive frame rates. A close look, however, reveals several cases where tree models are reused. Noticeably, the tree that branches into 2 segments at around a 30 degree angle appears to be drawn at least three times in this figure. While the forest is still impressive, it is made up of a few different tree models that are scattered around to give the illusion of uniqueness.

Figure 6 Screenshot of *Crysis* showing a forest with some model reuse (Harris, 2006)

In 2009, Valve released *Left 4 Dead 2* (IGN 2, 2012) which features very complex trees and convincing leaf structures as shown in Figure 7. These trees are large, unique, and include leaves that grow out of the branches just like real trees. Whether these trees are models that were made by an artist or produced using algorithms that will be discussed in this paper is unknown, but they are impressive enough to cause one reviewer to single them out as one of the most impressive aspects of the game (Walker, 2009). The trees do not play a part in the gameplay of *Left 4 Dead 2*, but the visual difference between these trees and trees featured in previous games is one feature that increases the immersion of the game. Additionally, both Figure 6 and Figure 7 depict geometry for grass which helps hide any seams between the base of the tree and the terrain.

Figure 7 Screenshot of *Left 4 Dead 2* showing a very detailed tree (Walker, 2009)

The same year (Minecraft Wiki, 2012) Minecraft is released with a very minimalistic art style shown in Figure 8. While this tree might not look as impressive as the previous example, Minecraft is a procedural world which means that all of the terrain, plant, and animal life is placed according to some rules rather than hardcoded. The trees, too, are procedurally generated using wood-textured blocks for the trunks and leaf-textured blocks for the leaves. This enables *Minecraft* to have an unlimited number of different looking trees without having to store any models on disk, thus creating a unique experience for each player.

Figure 8 Screenshot of *Minecraft* showing a simple procedural tree (Diving Follinica, 2012)

This paper will discuss how to procedurally generate an infinite number of different looking trees of different species based on physical rules that define that species. This is essentially employing techniques that generate the *Minecraft* trees to make trees that look like those in *Left 4 Dead 2* and forests like those in *Crysis* all without saving a single tree model to disk.

2.2 – Prior Research on Trees

2.2.1 - Fractal and Rule-Based Trees

A fractal is an image that contains a pattern that infinitely repeats itself over a series of iterations (Christersson, 2012). Popular fractals include the Mandelbrot set, the Sierpinski gasket, the Koch Snowflake, and Barnsley's Fern, which are all illustrated in Figure 9. The Fern, in particular, is one example showing that these mathematical models can be used to portray convincing plant geometry.

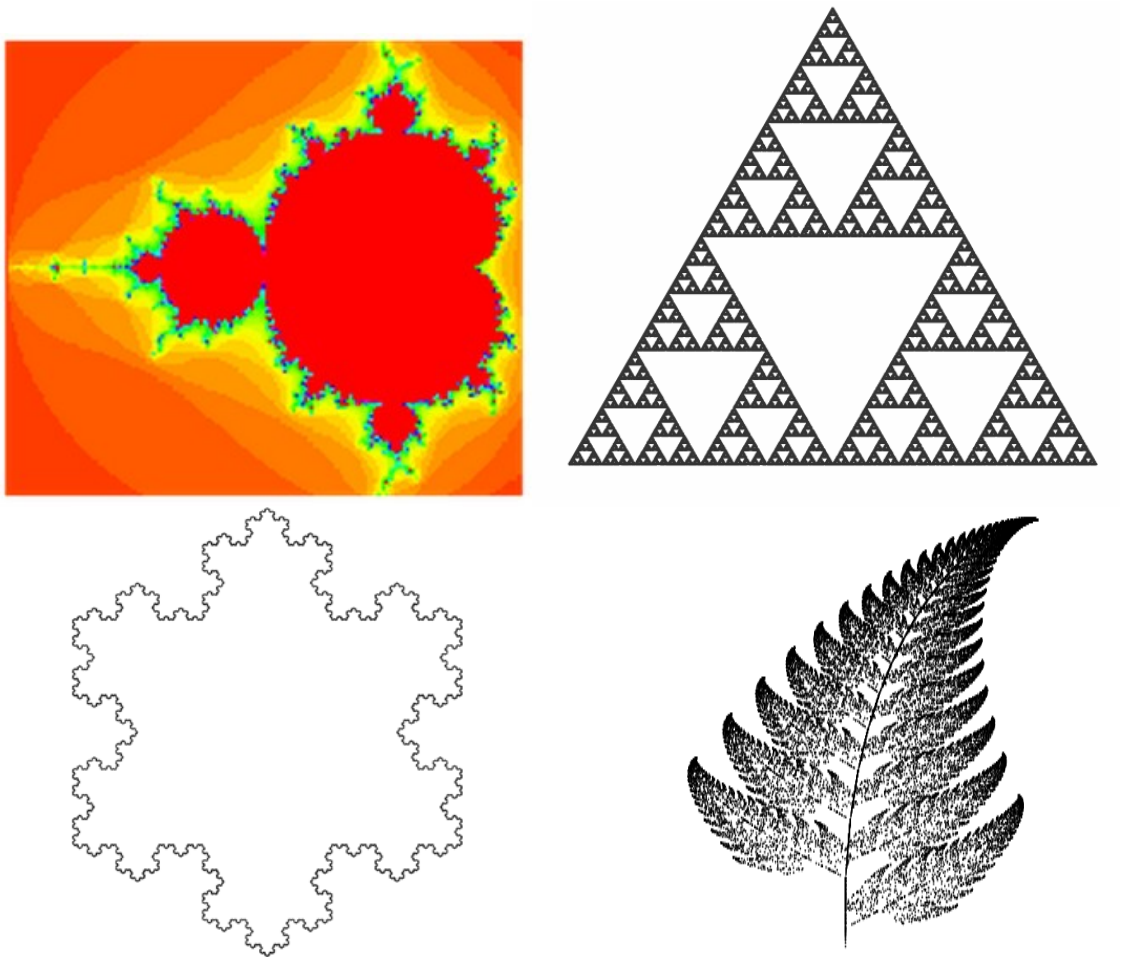

Figure 9 Fractals clockwise from top left: Mandelbrot Set, Sierpinski Gasket, Koch Snowflake, Barnsley's Fern (Weisstein, 2012)

One variation of fractals, Lindenmayer Systems or L-Systems, provides a set of simple drawing operations and recursion rules based on an input string to create fractals that might look like a plant. The pipeline takes a starting string and an expansion of each letter in that string into a new string. At each recursion, each letter in the input string will be replaced with its corresponding mapping. For example, take the following rules: "F" will draw a straight line, '-' will rotate 30 degrees left, '+' will rotate 30 degrees right, '[' pops the current state off the drawing stack, and

‘]’ pushes the current state onto the stack (Ochoa, 1998). Drawing the input string “F” mapped to F -> F[-F]F[+F][F] over 5 different iterations will result in a tree-like shape that looks like Figure 10.

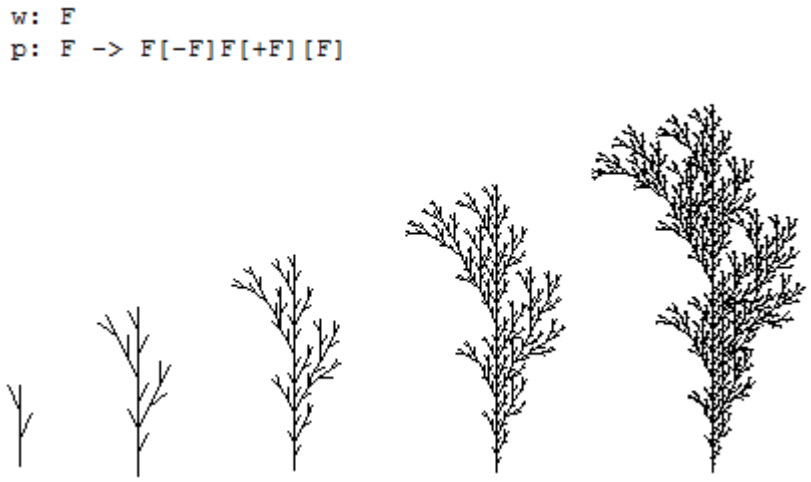

Figure 10 L-System mapping of F -> F[-F]F[+F][F] drawn with 5 different iteration levels (Ochoa, 1998)

These rules can be made more complex to cover many aspects of a tree. For instance, at a basic level, trees are composed of a trunk, branches, and leaves. Each section of a tree has rules that define parameters such as length, width, taper, angle, and iteration depths. Changing these rules for each section of a tree can create very unique branching structures. If these rules are created using real-world data, trees can be created that are physically accurate and resemble an actual specie.

One study compiled a list of 33 parameters that define a tree, then rendered various tree species based on differences in these parameters. These parameters include those mentioned above, as well as relevant parameters for each of three iterations defining a main branch, secondary branch, and tertiary branch (Weber, 1995). An example parameter list from this study for 4 tree species

can be found in Appendix 8.1. One example from this project includes the Black Tupelo shown in Figure 11. Here you can notice the distinct difference in angles between the primary branches off the main trunk and the secondary branches off of the primary ones. Additionally the taper value causes the width of the tree to converge at the top.

Figure 11 Generated model of a Black Tupelo with and without leaves. (Weber, et.al, 1995)

2.2.2 - Texture Lobes

Another technique to realistically model trees takes an opposite approach and starts with the leaves rather than the trunk (Livny, et. al, 2011). This approach starts with a laser scanned point cloud representation of a tree, reconstructs a branching structure within the bounds of the point cloud, then establishes convex geometric shapes that fill out the point cloud. These shapes, or

lobe-textures, represent the bounds of the leaves which will be connected to the nearest branch. This tree-generation process is illustrated in Figure 12 below.

Figure 12 Lobe-texture representation of a tree (Livny, et. Al, 2011)

The first block of the above image shows the original tree. The tree is then scanned and represented in the second image as a point cloud. The point cloud is decomposed into lobe-textures represented by the colored volumes in the third image and a tree branching structure is created to populate the tree. Finally, leaves are added from a set of leaf textures that connect from the branch to the exterior of the lobe geometry. By modifying the positioning, size, or density of the lobes, the leaf structure of the tree can be made to look significantly different.

Rather than providing many explicit rules for the tree composition, as with the previous example, the texture-lobe system primarily uses the point cloud and a set of leaf textures to accurately depict the species of a tree.

3 - Methodology

This project is based predominantly on Livny et. al's work on texture lobes, using an additional rule-based system adapted from Weber et. al's work to produce the tree's branches and ultimately creating procedural trees of any species using minimal disk space. This section will discuss the pipeline of the tree generation, the chosen rules for generating trees, and the application that allows for fast tree generation and user modifications. A block diagram is shown in Figure 13 below. The tree contains an optional point cloud and tree rules, which determine the final trunk and branch hierarchy, leaf placement, and leaf orientation.

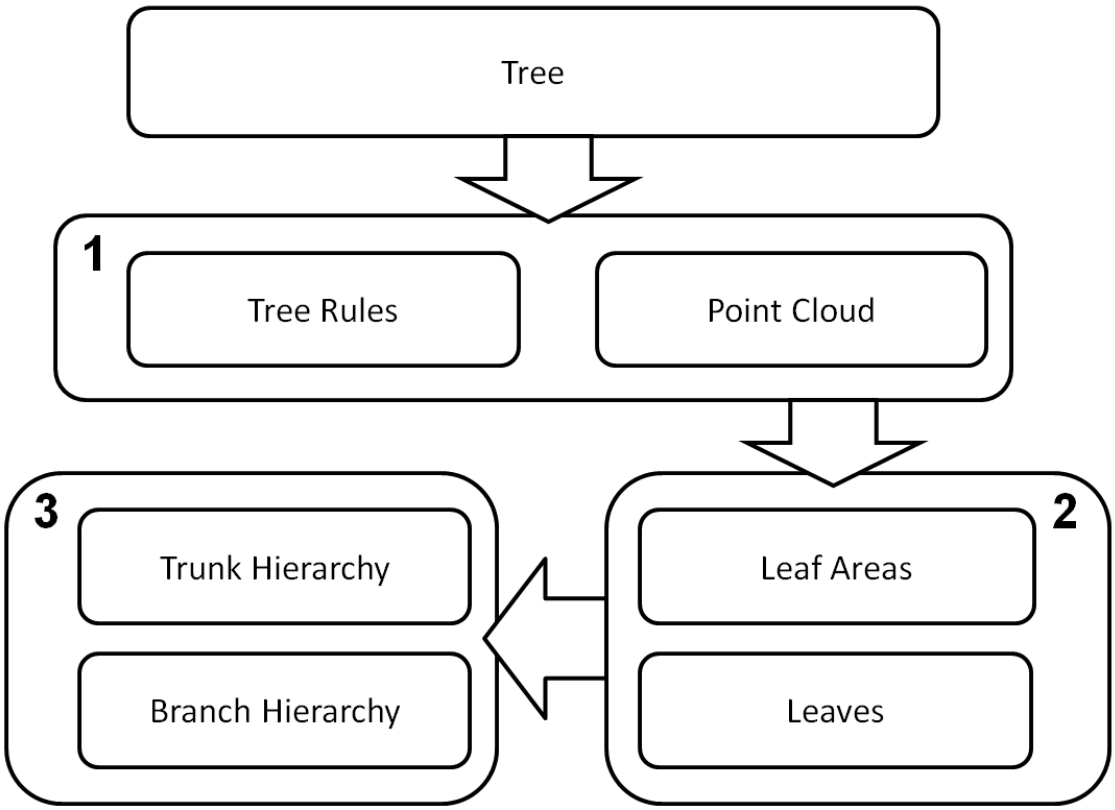

Figure 13 - Block Diagram of Tree Generation Pipeline

3.1 - Tree Construction Pipeline

Our pipeline is a six-stage process involving creating the initial trunk, creating the leaf lobes, connecting the trunk to the leaf lobes, creating branches inside the leaf lobes, adding leaves to fill out the leaf lobes, and finally creating the geometry of the branches. This pipeline fits into three main groups: construction parameters shown in group 1 within Figure 13, leaf placement in group 2, and branch hierarchies in group 3.

The construction parameters for the tree include the tree rules, discussed in more depth in the next section and an optional point cloud to define the bounds the tree will be created within. The tree rules define the placement of the different components of the tree. By adjusting these rules, different species of trees can be procedurally generated. The point cloud defines where the trunk and leaves will be placed. If the point cloud is not provided, trees can be generated using just the tree rules.

Leaf placement involves creating one or more bounding volumes that contains the tree's leaves. The positioning of these volumes can either be determined based on an input point cloud or can be placed arbitrarily to produce trees that look different. The rules from the construction parameters are used to determine position and orientation of each leaf when generating the individual leaves within these bounding volumes.

Branch hierarchies include both the trunk hierarchy and the branch hierarchy which are representations of where the tree's branches are placed. The trunk hierarchy is the structure of the main trunk of the tree. The branch hierarchy includes the branches that stem out of the trunk. Both hierarchies are created based on the construction parameters from the first step and are influenced by the locations of the leaf volumes. The trunk hierarchy includes a trunk for each of the leaf volumes and the branches are created within the leaf volume to ensure that leaves can be

generated within the volume. Our implementation of this pipeline is discussed later in this chapter.

3.2 - Tree Generation Rules

The generation of the tree in the previously mentioned pipeline is entirely driven by input variables that define the structure of different species of trees. The system has a TreeRules interface with the methods shown in Figure 14. This interface can be extended and populated with variables that will make convincing looking tree species. Some example data can be found in Appendix 8.2.

We now describe some of the variables that are used to specify the look of a tree based on its specie.

Figure 14 - Interface definition for TreeRules interface.

3.2.1 - TrunkAngle

If the tree has multiple trunk iterations, each iteration will have a certain number of branches coming off it. These branches will be generated with an ending position such that the angle between the child branches will match the TrunkAngle field in degrees. For example, if the trunk angle is 45, and there are two branches being created off a trunk split, the parent trunk will end in two child trunk branches with an interior angle of 45 degrees.

3.2.2 - BranchAngle

This field indicates the angle between the trunk and the first branch in the branch hierarchy that is attached to the trunk. For example, if the branch angle is 90 degrees, the first branch iteration for each branch on the trunk will be perpendicular to the parent trunk.

3.2.3 - SubsequentAngle

The subsequent angle field is similar to the trunk angle field, but creates the angle between branches on a branch iteration rather than trunks on a trunk iteration.

3.2.4 - NumTrunkIterations

This field sets the number of times the trunk iterates. Each iteration creates a new trunk segment split with a number of new tree branches created at the parent's end point.

3.2.5 - BranchIterations

This field sets the number of times each branch iterates. Just like the trunk iterations, each branch iteration creates a number of new tree branches starting at the parent's end point.

3.2.6 - LeafTaper

This field sets the value of the leaf taper to have a cone shaped lobe (if 1), fills out the lobe area (if 0), or something in between. This is accomplished by shifting the location of the branch control points within the leaf lobe. For example, if the leaf taper value is 1, the control points on the top of the lobe will be translated to the center of the sphere while maintaining their vertical

component, while the points at the midpoint of the sphere will maintain their positions, and the points on the bottom of the sphere will be translated outward to fit the line between the top points and the midpoints.

3.2.7 - NumTrunkBranchesOffSplit

This field sets the number of branches that will be created off a trunk each time the trunk iterates. The angle between opposite trunks is set earlier in the trunk angle field, and the rotational angle is automatically based on the number of branches to keep an even distribution around the parent. For example, if there are 2 branches, the interior angle is set by the trunk angle field, and the rotational angle will be 180 degrees so that the branches are on opposite sides of the parent. If there were three branches, the rotation would be partitioned in thirds, or 120 degrees.

3.2.8 - NumBranchesOffSplit

This field is the same as the number of trunk off a split, except for branches.

3.2.9 - NumSegmentsInBranch

This field sets how many times to split each branch in the tree to create a more organic looking branch. If the value is 1, the branch will be rendered as-is, with a single straight line from the starting point to the ending point. However, if the value is greater than 1, the original branch will be divided into this number of equal lengths with a slight offset set by the following parameter.

3.2.10 - SegmentOffset

This field sets a vector for how much to offset each split in a branch. The end point of each segment will be the parent segment's end point plus the parent segment's direction times the segment length plus this offset.

3.2.11 - LengthReductionFactor

This field sets the percentage a child branch's length will be over its parent. At each iteration, the branches will get this percent shorter.

3.2.12 - BranchStartWidth

This field sets how wide each branch will start at. Typically a branch will be narrower than the trunk.

3.2.13 - TrunkStartWidth

This field sets the starting width of a trunk. This is how wide the tree will be at the base.

3.2.14 - TwigStartWidth

The final iteration of a branch is the connection between the branch and the leaf, in this case called a twig. This field sets the starting width of the twigs in the tree.

3.2.15 - MinBranchWidth

Each trunk, branch, and twig is tapered to cause the ending width to be less than the starting width. This field sets the minimum width a branch can be to ensure that there is no overcompensation from the tapering.

3.2.16 - BranchTaperPercent

This field sets the percentage a branch is tapered from the start to the end. The ending width of the branch will be the starting width multiplied by this percentage.

3.2.17 - TrunkTaperPercent

This field is the same as the branch taper percentage, but only affects the tree trunks.

3.2.18 - NumBranchControlPointsOnLobe

This field sets the number of control points to place on the surface of each lobe. These control points determine where branches will grow toward, so the number of control points determines branch density inside a lobe.

3.2.19 - NumLeafPointsOnLobe

This field sets the number of leaf control points to place on the surface of a lobe. These are only used internally during the CreateLeaves method, and might not be applicable for each tree species. For example, the weeping cherry tree has very characteristic leaves that fall away from the top of the leaf lobe along the surface of the lobe. The leaf control points indicate where the leaves will be grown toward, rather than where branches are grown.

3.2.20 - MaxTrunkHeightPercent

This field sets the maximum percentage a trunk can pass through a leaf lobe. For example, if this field is 0, when a trunk is attached to a lobe, the end point will be on the surface of the lobe closest to the trunk. If the field is 1, the trunk will pass through the entire leaf lobe and end on the opposite surface.

3.2.21 - CreateBranchControlPoints

This field creates the number of control points corresponding to the number of branch control points (set previously) on each of the inputted LeafLobes. This controls the distribution of branches on a lobe. The control points can either be uniformly distributed along the surface or distributed using some other algorithm. These control points are altered by the leaf taper field which translates these points to achieve a desired ending location for the tree's branches.

3.2.22 - CreateLeafControlPoints

This field creates the number of control points corresponding to the number of leaf control points (set previously) on each of the inputted LeafLobes. This control the distribution and density of leaves within each lobe. This can default to a uniform distribution or be some other algorithm.

3.2.23 - CreateLeaves

This function takes a tree branch input and creates leaves on that branch. This function is responsible for deciding the orientation and size of the leaf.

3.2.24 - LeafTexture

This function sets the leaf texture that will be drawn for this tree species.

3.2.25 - Tessellation

This function determines the level of detail the trunk geometry will be rendered at. Each tree branch is a modified cylinder with a tapered end. The cylinder is rendered with a variable number of sides that corresponds to this field. For example, if this field is set to 4, the cylinder will be a rectangular prism. As this number increases, the geometry more closely resembles a cylinder.

A user might want to extend the application to support different tree species. This is simply done by creating a new class deriving from the TreeRules interface that contains implementations for all of the above mentioned functions.

3.3 - Tree Generation Application

To facilitate easier and interactive creation of trees, a Windows application was developed to guide users through the tree generation pipeline. This application is shown in Figure 15. The toolbar on the top contains buttons for creating different aspects of the tree, selecting the tree

type, and creating the final render of the tree. The main blue window is the drawing area where the tree is rendered. A user can interact with this window as well as move around by holding the appropriate mouse buttons and using the W, A, S, and D buttons on the keyboard.

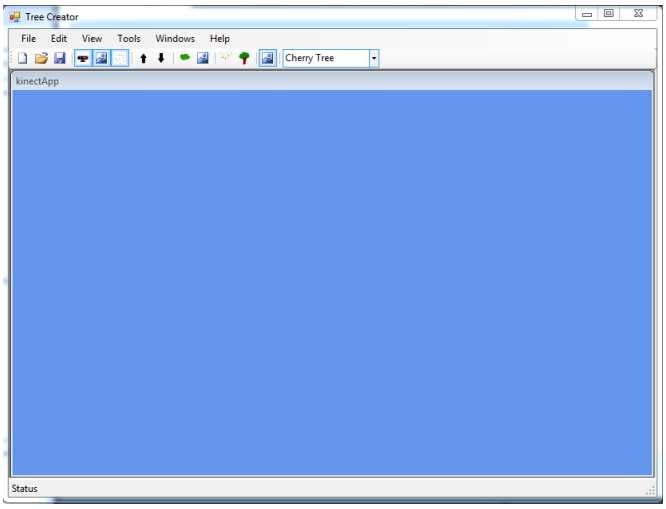

Figure 15 Tree Creator Application View

A tree can be created in two ways: automatic and manual generation. Both processes involve different construction of the trunk hierarchy and leaf lobe placement, but the subsequent creation of branches and leaves is the same. Manual generation allows for the user to create specific starting trunks and leaf lobe locations, while automatic generation will place the trunk and leaf lobes in algorithmically determined locations. The application is implemented mainly as a Tree class that contains all the logic for creating the tree, which will be discussed in this section. This class contains vertex buffers (or fields for classes with vertex buffers) for all the geometry that needs to be drawn for the tree, which includes a tree hierarchy of lines, leaf lobes, a point cloud, geometry of the branches, and the leaves. This class also contains logic for running through the tree creation pipeline. See the UML class diagram in Figure 16 for a high-level overview of the structure of this application.

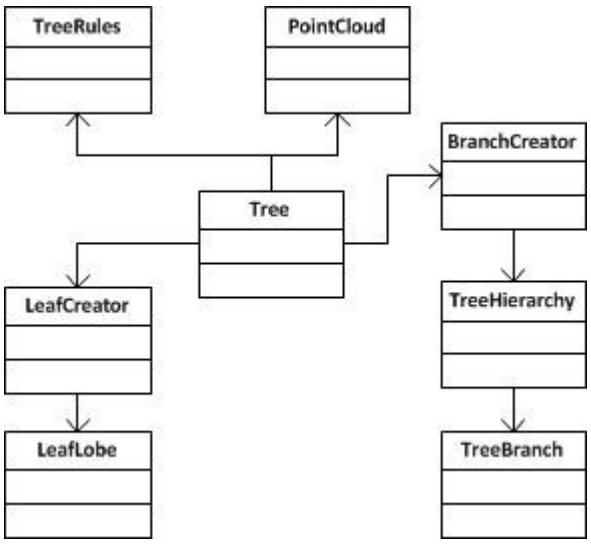

Figure 16 - High level UML Class Diagram of tree generation application

The core components used to create the tree are a TreeBranch class and a LeafLobe class. Tree Branches are used to create the tree's trunk and branch hierarchies from section 3 in Figure 13. Leaf lobes, shown in section 2 in Figure 13, are used to define the leaf construction of the tree including leaf areas, or bounding volumes where leaves will be rendered. TreeBranches represent the branches of the tree and contain a starting position, an ending position, a parent TreeBranch, and a list of child TreeBranches. When creating a TreeBranch, the start and end might be explicitly set or the start, length, and a rotation angle can be set which will calculate the end position. LeafLobes are simple bounding volumes that represent where the tree will produce branches.

To set up the scene for tree creation, a trunk and leaf lobes are created in world space. If the user is creating the tree manually, they have the option of drawing out an initial trunk hierarchy and placing the lobes. Alternatively, the application can automatically create an initial trunk with angles and branching structures based on specified input rules and will create the tree using a single leaf lobe. Once the initial trunks and leaf lobes are in the scene, if there are LeafLobes

that do not have branches contained inside them, the nearest branch to that leaf lobe will be connected to the lobe by adding a child branch off the closest branch. This child branch will start at the end of the existing trunk, will be drawn in the direction of this start position to the leaf lobe's center, and will end at the intersection of this vector and the outer bound of the lobe.

The user has the option of using a point cloud generated from a Microsoft Kinect Sensor to assist in placing initial trunk tree branches and leaf lobes. The Kinect generates a point cloud of a scene, which can be rendered in the tree application. When generating the tree branches or leaf lobes, the positions are clamped to existing point cloud positions, and can later be translated to a desired location by the user.

When a LeafLobe is created, a set of control points is randomly distributed on the surface of the lobe as shown in Figure 17 below. Each branch will generate a branching structure within the leaf lobe according to some input angle, and will use these control points as a desired ending location. So, generating the branching structure works backwards by starting at the control point on the surface of the lobe and ending at some point on the parent branch that achieves the desired angle as shown in Figure 18. This ensures that branches will cover enough of the lobe to achieve proper leaf placement. Increasing or decreasing the number of control points on a leaf lobe will accordingly change the number of branches in that lobe. Additionally, moving these control points alters where the branch ends, so an input taper value is used to move the control points into or away from the sphere's center to enable creation of cone shaped leaf volumes.

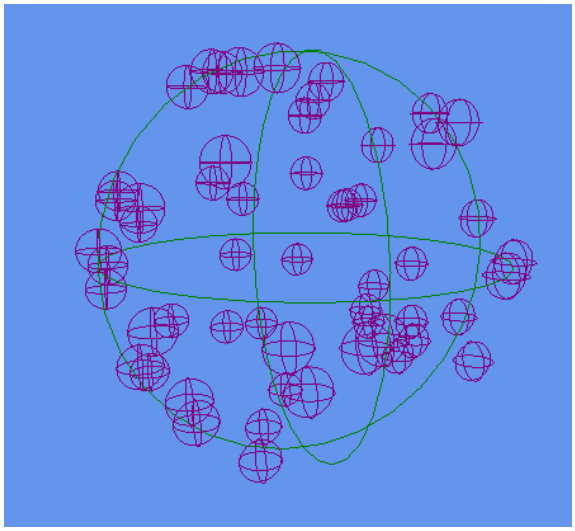

Figure 17 - Control points (in purple) defining branch growth locations are distributed along the surface of leaf lobes (in green)

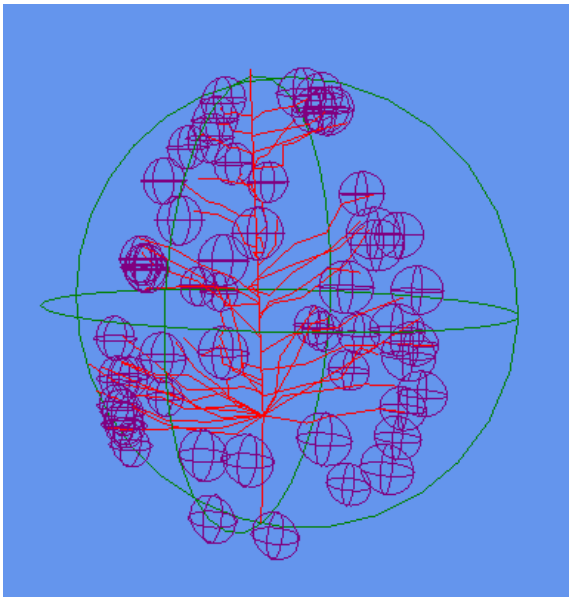

Figure 18 - Branches are grown toward branch control points at a specified input angle. This figure also illustrates branch tapering which involves converging these control points inward.

After the branch structure is created for the trunk and branches, leaves are attached to the branches according to the TreeRules class. The orientation and texture of the leaves of a tree species are very dependent on the type of tree, so this function is left for each tree species to determine proper leaf placement. For example, a weeping cherry tree has leaves that fall outward along the surface of the leaf lobe while an oak tree has leaves that remain closer to their parent branch. Adding leaves typically adds branches to the tree hierarchy.

Finally, the tree is finished by generating geometry for all of the branches. Each branch is given a type for whether it is a trunk, a branch, or a twig connecting a branch to a leaf. Depending on the type of branch the created geometry will have a corresponding starting width and taper percentage that is read from the specie's tree rules class. The geometry is rendered as a textured cylinder with the ending tapered.

A close up of the application's tool bar can be seen in Figure 19 below. The first three icons represent new (A), open (B), and save (C) which work as expected. Saving a tree will save to a proprietary .tree format, developed for this application and described in Appendix 8.3, which includes a tree type, a list of branches, and a list of leaf lobes. The branches are saved as an id, a parent id, a start position, and an end position. Leaf lobes are saved as a position and a radius. Only the trunk branches and leaf lobes are saved to reduce the file size. The tree is re-created from this data when it is loaded.

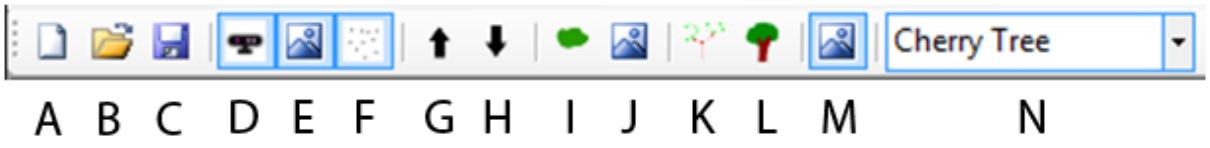

Figure 19 Close up view of the Tree Creator Tool Bar

The next five icons, D through H, represent actions dealing with the Kinect. This application supports rendering a real-time point cloud from a connected Kinect device to create a tree using that point cloud as input. Button D, with the image of a Kinect Sensor, turns the Kinect on or off. Button E toggles the distance the sensor is reading (if the connected Kinect is a Kinect for Windows sensor that supports this toggle.) This increases precision for trees that are being scanned closer to the sensor. Next, button F is a toggle to render the point cloud on screen. Finally, Buttons G and H raise or lower the angle of the sensor.

The next three buttons, I through J, deal with manual creation of trees. Button I enables the user to place leaf lobes. If selected, the user can click on the screen to place a lobe. Each lobe can be selected by clicking on it again. The radius can be increased or decreased with + or - and the position can be changed by pressing W when selected and moving the on-screen movement widget. Button J allows the user to place trunk branches on the screen. When selected, a user can click and drag to place a TreeBranch. When the user releases the Left Mouse Button, the end position of the TreeBranch is set and the TreeBranch is added to the internal tree hierarchy. Next are two buttons, K and L, to create the Tree. Button K concerns manual creation of trees. If there are TreeBranches and LeafLobes on the screen and Button K is pressed, a tree will be created with those inputs. Otherwise if the Button L is pressed, a Tree will be created in the bounds of a single LeafLobe of radius 1 with the current selected TreeRules from the drop down list (N) on the far right. Finally there is a toggle button M to show or hide leaves.

Additionally, there are two supplemental windows that help with the tree creation. A "Bound Window" and a "Branch View Window" are accessible through the Windows menu. The Bound Window controls the minimum and maximum values of x, y, and z to render the point cloud from the Kinect. This is useful to reduce any noise from irrelevant background elements in the scene that is being scanned. The Branch View Window shows a hierarchy of all of the branches that are currently in the scene. A branch can be selected in this window and can be moved on screen with the W button or deleted with the Delete button just as the Leaf Lobes can be.

Figure 20 below shows a sample usage of the Tree Creator application with the Branch View Window. There is a tree hierarchy with a parent TreeBranch "0" with two TreeBranch children "1" and "2". At the ends of both children TreeBranches are two LeafLobes of radius 1. The type of tree is not selected, so the system will default to the first TreeRule that was registered in the main application if generation is attempted without selecting a tree species. Notice also that the selected LeafLobe (colored yellow) is being translated with the translation widget drawn at its center.

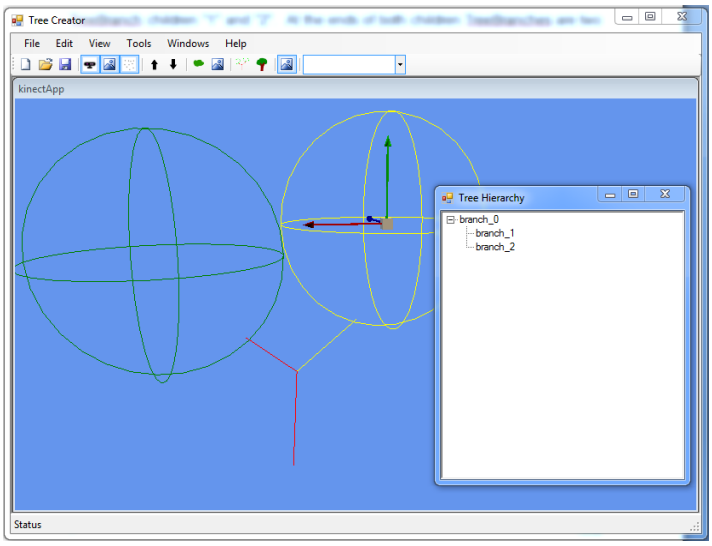

Figure 20 Sample usage of the Tree Creator Application

4 - Results

This section contains some sample screen shots and performance metrics of generated trees. The rules for each of the tree types can be found in Appendix 8.2. Three different tree types were rendered, Black Oak trees, Balsam Fir trees, and Weeping Cherry trees. Each rendered tree is described along with performance tables with Frames per Second (FPS) charts and tables for single trees as well as forests of trees.

4.1 - Tree Samples

Tree Samples were generated at different complexities to monitor the frame rate of the system when drawing a variable number of trees. Increasing numbers of trees were rendered, starting with 1 tree and ending with 20. Additionally different parts of the tree were generated independently including just the trunk, and the trunk, and the trunk with branches. Finally, the trees are rendered using high level of geometry detail and low level of detail as measured by the tessellation rule. This rule defines how many subdivisions the cylinder geometry of the tree branch will be rendered at. For example, if the tessellation is 4, the cylinders for the branches are rendered as rectangular prisms while larger tessellation values appear more as cylinders.

4.1.1 – Black Oak

Figure 21 Black Oak (Left: t = 36; Right: t = 5) Notice the angled geometry at the base of the right render

In the above renders of the Black Oak tree, the rules from Appendix 8.2.1 are apparent. Noticeably, the branches have an iteration depth of 3, each iteration creates 2 children, and each child is rotated 45 degrees apart. Additionally, the branches have 25 segments creating branches that are not completely straight. Also visible, the trunk is set to grow halfway through the leaf volume.

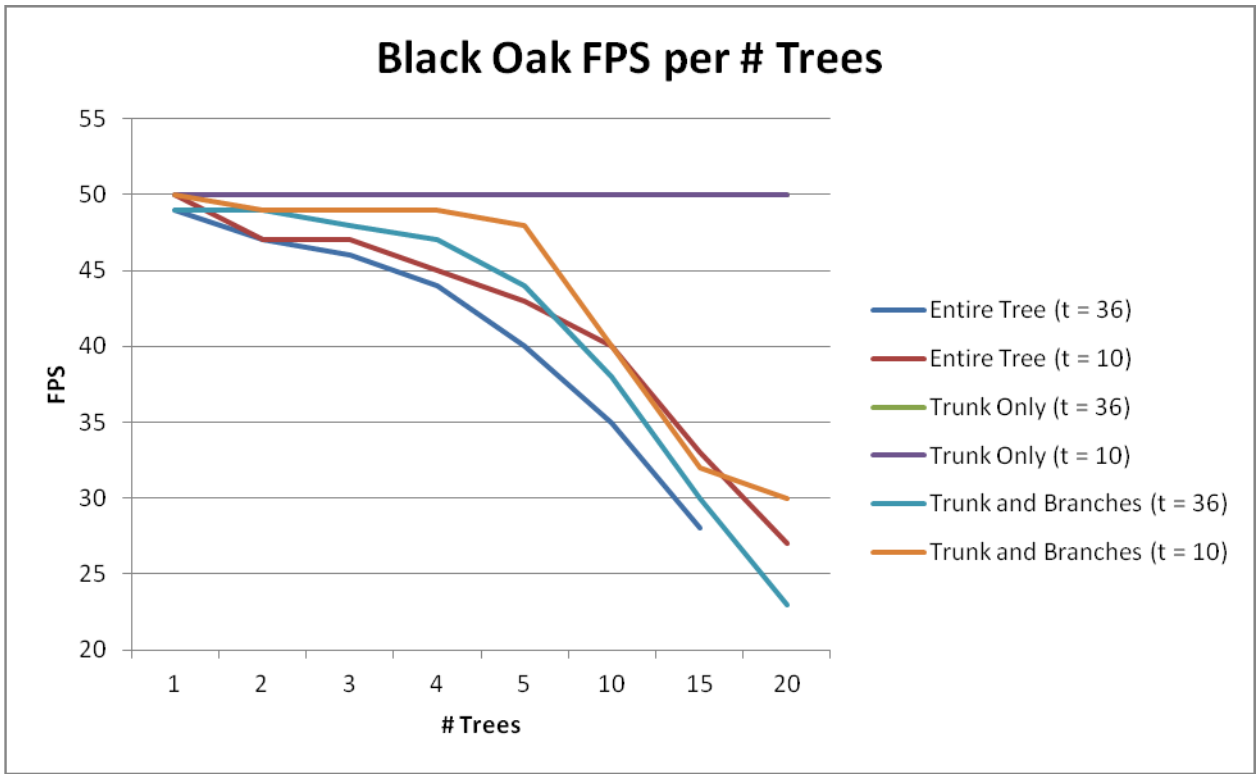

Figure 22 - Frame rates for increasing # of Black Oak trees at different levels of detail

t = 36	(t = Tesselation)		
# trees	Entire Tree (t = 36)	Trunk Only (t = 36)	Trunk and Branches (t = 36)
1	49	50	49
2	47	50	49
3	46	50	48
4	44	50	47
5	40	50	44
10	35	50	38
15	28	50	30
20		50	23

t = 10	(t = Tesselation)		
# trees	Entire Tree (t = 10)	Trunk Only (t = 10)	Trunk and Branches (t = 10)
1	50	50	50
2	47	50	49
3	47	50	49
4	45	50	49
5	43	50	48
10	40	50	40
15	33	50	32
20	27	50	30

Figure 23 - Frame rates for increasing # of Black Oak trees at different levels of detail

The graph in

t = 36	(t = Tesselation)		
# trees	Entire Tree (t = 36)	Trunk Only (t = 36)	Trunk and Branches (t = 36)
1	49	50	49
2	47	50	49
3	46	50	48
4	44	50	47
5	40	50	44
10	35	50	38
15	28	50	30
20		50	23
t = 10	(t = Tesselation)		
# trees	Entire Tree (t = 10)	Trunk Only (t = 10)	Trunk and Branches (t = 10)
1	50	50	50
2	47	50	49
3	47	50	49
4	45	50	49
5	43	50	48
10	40	50	40
15	33	50	32
20	27	50	30

Figure 23 shows that frame rate decreases as the number of trees increases because of the complexity of the branches and the leaves. When only the trunks are drawn, the frame rate is constant, so the decrease in frame rate between 5 and 20 trees is due to rendering the branches and leaves. This graph also shows that to keep the frame rate above 45 FPS, 4 trees can be drawn at a low tessellation, or 3 trees can be drawn at a high resolution. To maintain 30 FPS, 15 trees can be drawn at a low resolution, or 10 trees can be drawn at a high resolution.

4.1.2 – Balsam Fir

Figure 24 - Balsam Fir (Left: t = 36; Right: t = 5)

In the above renders of the Balsam Fir tree, the rules from Appendix 8.2.2 are apparent. The Trunk angle is 0, so the trunk grows completely vertical; the branch angle is 20, so the first branches are created 20 degrees up from perpendicular with the parent trunk. There are not branch iterations, so there is only a singular branch from the trunk to the leaf lobe. The trunk is created to pass through the entire length of the leaf lobe. Also, the leaf taper is 100%, so the leaf lobe becomes a cone shaped volume.

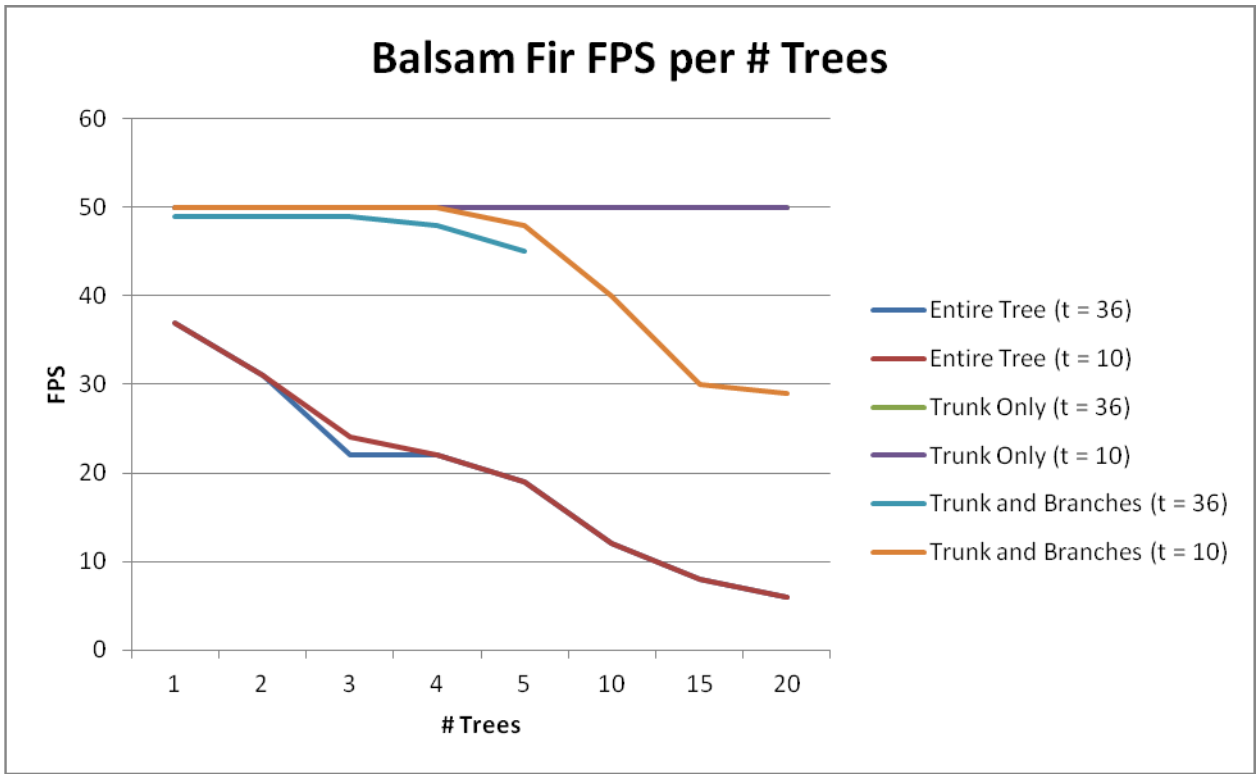

Figure 25 - Frame rates for increasing # of Balsam Fir trees at different levels of detail

t = 36	(t = Tessellation)		
# trees	Entire Tree (t = 36)	Trunk Only (t = 36)	Trunk and Branches (t = 36)
1	37	50	49
2	31	50	49
3	22	50	49
4	22	50	48
5	19	50	45
10	12	50	
15	8	50	
20	6	50	

t = 10	(t = Tessellation)		
# trees	Entire Tree (t = 10)	Trunk Only (t = 10)	Trunk and Branches (t = 10)
1	37	50	50
2	31	50	50
3	24	50	50
4	22	50	50
5	19	50	48
10	12	50	40
15	8	50	30
20	6	50	29

Figure 26 - Frame rates for increasing # of Balsam Fir trees at different levels of detail

The data from Figure 25 and Figure 26 support the trends seen in the Black Oak. However, there is a much greater performance loss when rendering this tree type at both high and low resolutions. This implies that the leaves are the factor most contributing to the lower FPS. Only two trees can be rendered at either a high or low tessellation value to maintain an FPS greater than 30.

4.1.3 – Weeping Cherry Tree

Figure 27 Weeping Cherry Tree (Left: t = 36; Right: t = 5)

In the above renders of the Weeping Cherry tree, the rules from Appendix 8.2.3 are apparent. The branch angle is 45 which can be seen on the twigs connecting to the cherry blossoms. The Trunk Angle is 3 which can be seen in the slight rotation of the 2 trunk nodes. The trunk is created to pass through the entire length of the leaf lobe. Finally, there are between 1 and 3 branches off each trunk, which are created at the top of the trunk.

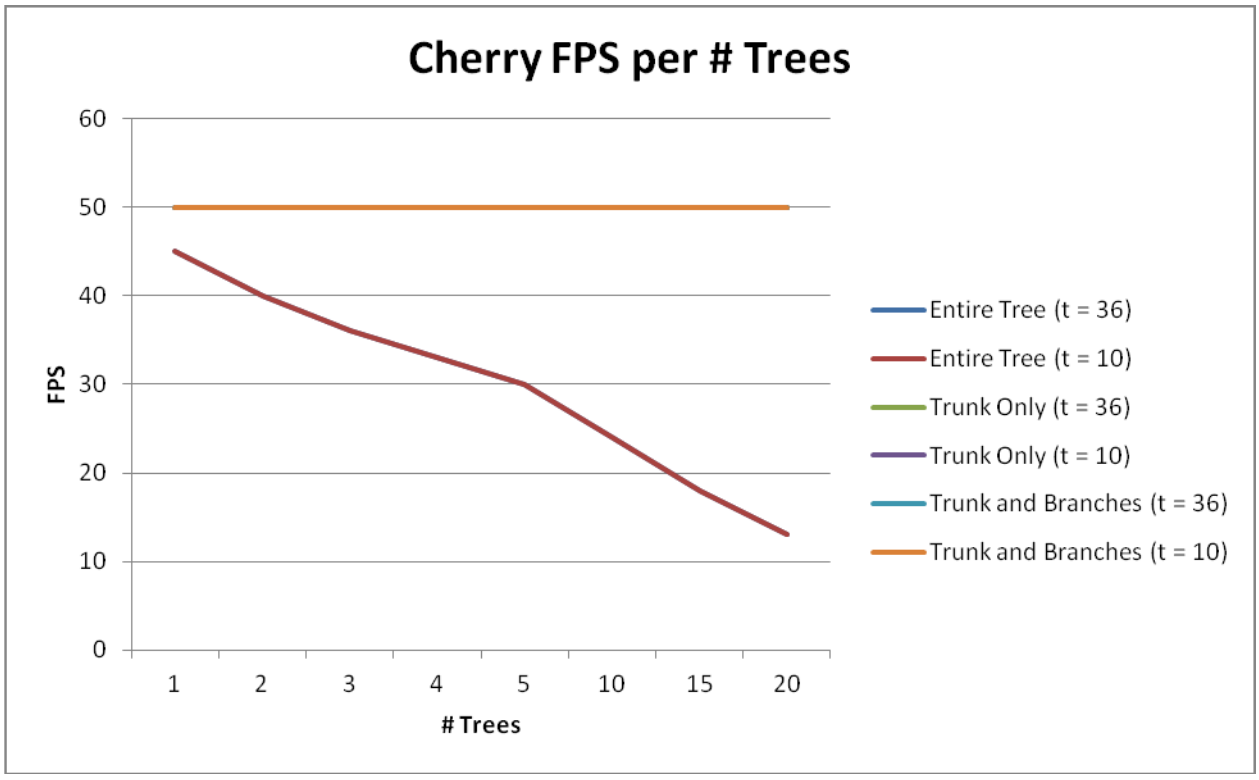

Figure 28 - Frame rates for increasing # of Weeping Cherry trees at different levels of detail

t = 36	(t = Tessellation)		
# trees	Entire Tree (t = 36)	Trunk Only (t = 36)	Trunk and Branches (t = 36)
1	45	50	50
2	40	50	50
3	36	50	50
4	33	50	50
5	30	50	50
10	24	50	50
15	18	50	50
20	13	50	50

t = 10	(t = Tessellation)		
# trees	Entire Tree (t = 10)	Trunk Only (t = 10)	Trunk and Branches (t = 10)
1	45	50	50
2	40	50	50
3	36	50	50
4	33	50	50
5	30	50	50
10	24	50	50
15	18	50	50
20	13	50	50

Figure 29 - Frame rates for increasing # of Weeping Cherry trees at different levels of detail

The Weeping Cherry maintains a similar FPS profile between the high and low tessellation renders as seen in Figure 28 and Figure 29. The tessellation affects the number of triangles that need to be rendered for the branches. Because there is no frame rate difference between the two species in this case, the frame rate decrease is probably mostly caused by the leaves. This could be explained by this tree type having fewer branches than the previous trees, yet a large number of leaves.

5 - Analysis

5.1 - Functionality

This project created a system to create a wide range of physically based species. Based on input rules, unique looking trees of the same species can be generated to create forests of trees with true variation. This can all be done in real time on a GTX 560 graphics card.

This is the first step in getting unique forests into a game setting. By including the tree generation code in a game project, these trees can be generated in a game setting without having to store any large model files.

5.2 - Performance

Frame rate is an important measure of performance in a game setting with a target of 30 to 60 frames per second being desired to maintain an interactive experience. To measure the performance of this system, an increasing number of trees can be generated and the frame rate can be monitored at each level. Additionally, each stage in the tree generation process will have varying degrees of complexity, so some stages might cause more of a performance hit than others. A tree can be generated in sections and the frame rate of a forest of trees (with just that section) can be measured.

The frame rates collected in the results section are generated by rendering a specific number of trees on top of each other. These trees are rendered at full detail and all of them are visible to the camera. Samples were taken for a high tessellation value and a medium level of tessellation.

Based on this performance analysis, it appears that adding leaves to the trees causes the majority of slowdowns. However, there are some noticeable cases when drawing many fir trees and oak trees with no leaves will also cause significant slow-downs. In the case of the fir tree, the system crashes when attempting to draw 10 or more trees because it is drawing too many triangles. This number was able to be drawn with leaves however, probably due to occlusion from the many leaves blocking the branch geometry from being drawn. When rendering the same number of black oak trees, the frame rate decreases significantly because there is minimal occlusion between the multiple trees. The cherry tree, on the other hand, has much more occlusion between different trees and consequently less geometry is drawn, so the frame rate stays constant.

Lowering the tessellation level greatly improves the frame rate of some of the trees because it has the effect of reducing the number of triangles that are rendered. However, in the case of the Balsam Fir tree and the Cherry tree, the tessellation has little effect. This is probably because these two trees have a greater number of leaves being rendered and fewer branches.

6 - Conclusions

In conclusion, this system is able to create physically based models of any tree species. A multi-step tree pipeline uses a set of input rules to generate a trunk, a branching tree hierarchy off of that trunk, and leaves that fill out leaf bounding volumes.

This system enables procedural creation of unique looking trees that can be applied to a game setting. This means that entire forests of realistic looking trees could be rendered in a game without repeating models or even saving any models on disk.

These trees can easily be integrated into a game by adding the tree creation code to an existing game project, creating an instance of a Tree, then drawing it during the game's draw call. The Tree class has its own vertex buffers for every entity that needs to be drawn.

6.1 - Future Work

Future work can be done in optimizing the pipeline for drawing many trees at once. The above performance analysis is an upper bound because each tree was drawn at full resolution, and each tree was visible to the camera. One way to increase performance would be a level of detail system. When adding this tree renderer to a game, it is unnecessary to render each tree at full resolution, especially when some trees might be far away from the camera. Creating a system where the number of branches and leaves are reduced when the tree is far enough away from the camera would solve this problem and reduce the load on the GPU.

Future work can also be done on better automating the generation of each tree type. There is support for Kinect integration, but the functionality there is limited to tracing a point cloud. More research can go into decomposing the visible branches and placing LeafLobes to match the leaf areas of a point cloud. This was explored early in the development, when scanned trees had

a single vertical trunk. These trunks were easy to isolate by taking horizontal slices of the point cloud and mapping the x and z values until the width increased to some width significantly larger than the average width of the slices below. However, when scanning trees that have multiple visible trunks or trunks that extend horizontally, this algorithm falters.

7 - References

(Diving Follinica, 2012). Avatar Tree Minecraft *Diving Follonica*. Retrieved 4-1-2012, from: http://www.divingfollonica.com/avatar-tree-minecraft&page=3

(Christersson, 2012). Christersson, Malin. Self-Similar Pattern. Retrieved 4-18-2012, from: http://www.malinc.se/math/fractals/mainen.php

(VGReleases, 2012). Crysis Release Dates - PC *VGReleases*. Retrieved 4-1-2012, from: http://www.vgreleases.com/PC/ReleaseDate-22507.aspx

(IGN 1, 2012). The Elder Scrolls III: Morrowind *IGN.* Retrieved 4-1-2012, from: http://pc.ign.com/objects/014/014332.html

(Zeldapedia, 2012). Great Deku Tree's Meadow *Zeldapedia*. Retrieved 4-1-2012, from: http://zelda.wikia.com/wiki/Great_Deku_Tree's_Meadow

(Harris, 2006). Harris, Will (4-20-2006) Crysis: New Screenshots and Preview. *BitGamer*. Retrieved 4-1-2012, from: http://www.bit-tech.net/gaming/pc/2006/04/20/crysis_new_screenshots/3

(Kamburelis, 2006). Kamburelis, Michalis (2006) Castle Game Engine Documentation. Retrieved 4-1-2012, from: http://castle-engine.sourceforge.net/vrml_engine_doc/output/xsl/html-nochunks/vrml_engine.html

(IGN 2, 2012). Left 4 Dead 2 *IGN*. Retrieved 4-1-2012, from: http://pc.ign.com/objects/143/14352245.html

(IGN, 2007). The Legend of Zelda: A Link to the Past (2007) *IGN*. Retrieved 4-1-2012, from: http://top100.ign.com/2007/ign_top_game_8.html

(Livny, et.al, 2011). Livny, Yotam. Pirk, Soeren. Cheng, Zhanglin. Yan, Feilong. Deussen, Oliver. Cohen-Or, Daniel. Chen, Baoquan. (2011) Texture-Lobes for Tree Modeling *ACM Transactions on Graphics,* from https://kops.ub.uni-konstanz.de/xmlui/bitstream/handle/urn:nbn:de:bsz:352-177194/Deussen%20etal.pdf?sequence=3

(Minecraft Wiki, 2012) . Minecraft Wiki. Retrieved 4-1-2012, from: http://www.minecraftwiki.net/wiki/Minecraft_Wiki

(Ochoa, 1998). Ochoa, Gabriela (12-02-1998) An Introduction to Lindenmayer Systems *University of Sussex* Retrieved 4-2-2012, from http://www.biologie.uni-hamburg.de/b-online/e28_3/lsys.html

(Orr, 2011). Orr, Lucy (8-17-2011) id Software Wolfenstein 3D *reghardware*. Retrieved 4-1-2012, from http://www.reghardware.com/2011/08/17/antique_code_show_wolfenstein_3d/

(Pullin, 2002). Pullin, Keith (4-26-2002) Elder Scrolls III: Morrowind *Computer And Video Games*. Retrieved 4-1-2012, from http://www.computerandvideogames.com/27403/previews/elder-scrolls-iii-morrowind/

(Walker, 2009). Walker, John (11-17-2009) RPS Left 4 Dead 2 Review *Rock, Paper Shotgun.* Retrieved 4-1-2012, from: http://www.rockpapershotgun.com/2009/11/17/rps-left-4-dead-2-review/

(Weber, et. al, 1995). Weber, Jason and Penn, Joseph (1995) Creation and rendering of realistic trees. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (SIGGRAPH '95), 119-128. http://doi.acm.org/10.1145/218380.218427

(Weisstein, 2012). Weisstein, Eric W. Fractal. Retrieved 4-2-2012, from *MathWorld – A Wolfram Web Resource* http://mathworld.wolfram.com/Fractal.html

(Giant Bomb, 2012). Wolfenstein 3-D *Giant Bomb.* Retrieved 4-1-2012, from http://www.giantbomb.com/wolfenstein-3-d/61-7694/

8 - Appendices

8.1 – Parameter List from (Weber, et. al, 1995)

Parameter	Description	Quaking Aspen	Black Tupelo	Weeping Willow	CA Black Oak
Shape	general tree shape id	7	4	3	2
BaseSize	fractional branchless area at tree base	0.4	0.2	0.05	0.05
Scale,ScaleV,ZScale,ZScaleV	size and scaling of tree	13, 3, 1, 0	23, 5, 1, 0	15, 5, 1, 0	10, 10, 1, 0
Levels	levels of recursion	3	4	4	3
Ratio,RatioPower	radius/length ratio, reduction	0.015, 1.2	0.015, 1.3	0.03, 2	0.018, 1.3
Lobes,LobeDepth	sinusoidal cross-section variation	5, 0.07	3, 0.1	9, 0.03	5, 0.1
Flare	exponential expansion at base of tree	0.6	1	0.75	1.2
0Scale,0ScaleV	extra trunk scaling	1, 0	1, 0	1, 0	1, 0
0Length,0LengthV, 0Taper	fractional trunk, cross-section scaling	1, 0, 1	1, 0, 1.1	0.8, 0, 1	1, 0, 0.95
0BaseSplits	stem splits at base of trunk	0	0	2	2
0SegSplits,0SplitAngle,0SplitAngleV	stems splits & angle per segment	0, 0, 0	0, 0, 0	0.1, 3, 0	0.4, 10, 0
0CurveRes,0Curve,0CurveBack,0CurveV	curvature resolution and angles	3, 0, 0, 20	10, 0, 0, 40	8, 0, 20, 120	8, 0, 0, 90
1DownAngle,1DownAngleV	main branch: angle from trunk	60, -50	60, -40	20, 10	30, -30
1Rotate,1RotateV,1Branches	spiraling angle, # of branches	140, 0, 50	140, 0, 50	-120, 30, 25	80, 0, 40
1Length,1LengthV,1Taper	relative length, cross-section scaling	0.3, 0, 1	0.3, 0.05, 1	0.5, 0.1, 1	0.8, 0.1, 1
1SegSplits,1SplitAngle,1SplitAngleV	stem splits per segment	0, 0, 0	0, 0, 0	0.2, 30, 10	0.2, 10, 10
1CurveRes,1Curve,1CurveBack,1CurveV	curvature resolution and angles	5, -40, 0, 50	10, 0, 0, 90	16, 40, 80, 90	10, 40, -70, 150
2DownAngle,2DownAngleV	secondary branch: angle from parent	45, 10	30, 10	30, 10	45, 10
2Rotate,2RotateV,2Branches	spiraling angle, # of branches	140, 0, 30	140, 0, 25	-120, 30, 10	140, 0, 120
2Length,2LengthV, 2Taper	relative length, cross-section scaling	0.6, 0, 1	0.6, 0.1, 1	1.5, 0, 1	0.2, 0.05, 1
2SegSplits,2SplitAngle,2SplitAngleV	stem splits per segment	0, 0, 0	0, 0, 0	0.2, 45, 20	0.1, 10, 10
2CurveRes,Curve,2CurveBack,2CurveV	curvature resolution and angles	3, -40, 0, 75	10, -10, 0, 150	12, 0, 0, 0	3, 0, 0, -30
3DownAngle,3DownAangleV	tertiary branch: angle from parent	45, 10	45, 10	20, 10	45, 10
3Rotate,3RotateV,3Branches	spiraling angle, # of branches	77, 0, 10	140, 0, 12	140, 0, 300	140, 0, 0
3Length,3LengthV, 3Taper	relative length, cross-section scaling	0, 0, 1	0.4, 0, 1	0.1, 0, 1	0.4, 0, 1
3SegSplits,3SplitAngle,3SplitAngleV	stem splits per segment	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
3CurveRes,3Curve,3CurveBack,3CurveV	curvature resolution and angles	1, 0, 0, 0	1, 0, 0, 0	1, 0, 0, 0	1, 0, 0, 0
Leaves,LeafShape	number of leaves per parent, shape id	25, 0	6, 0	15, 0	25, 0
LeafScale,LeafScaleX	leaf length, relative x scale	0.17, 1	0.3, 0.5	0.12, 0.2	0.12, 0.66
AttractionUp	upward growth tendency	0.5	0.5,	-3	0.8
PruneRatio	fractional effect of pruning	0	0	1	0
PruneWidth,PruneWidthPeak	width, position of envelope peak	0.5, 0.5	0.5, 0.5	0.4, 0.6	0.5, 0.5
PrunePowerLow,PrunePowerHigh	curvature of envelope	0.5, 0.5	0.5, 0.5	0.001, 0.5	0.5, 0.5

8.2 - Tree Parameters

8.2.1 Black Oak

TrunkAngle (degrees)	**10**
BranchAngle (degrees)	**45**
SubsequentAngle (degrees)	**15**
NumTrunkIterations	**1**
BranchIterations	**3**
LeafTaper (percentage: 0-1)	**0**
NumTrunkBranchesOffSplit	**1**
NumBranchesOffSplit	**2**
NumSegmentsInBranch	**25**
SegmentOffset	**0.4, 0.4, 0.4**
LengthReductionFactor (percentage: 0-1)	**0.9**
BranchStartWidth	**0.1**
TrunkStartWidth	**0.2**
TwigStartWidth	**0.005**
MinBranchWidth	**0.005**
BranchTaperPercent (percentage: 0-1)	**0.05**
TrunkTaperPercent (percentage: 0-1)	**0.4**
NumBranchControlPointsOnLobe	**15**
NumLeafPointsOnLobe	**30**
MaxTrunkHeightPercent (percentage: 0-1)	**0.5**

8.2.2 Balsam Fir

TrunkAngle (degrees)	**0**
BranchAngle (degrees)	**20**
SubsequentAngle (degrees)	**15**
NumTrunkIterations	**1**
BranchIterations	**1**
LeafTaper (percentage: 0-1)	**1**
NumTrunkBranchesOffSplit	**1**
NumBranchesOffSplit	**1**
NumSegmentsInBranch	**25**
SegmentOffset	**0.4, 0.4, 0.4**
LengthReductionFactor (percentage: 0-1)	**0.9**
BranchStartWidth	**0.1**
TrunkStartWidth	**0.2**
TwigStartWidth	**0.005**
MinBranchWidth	**0.005**
BranchTaperPercent (percentage: 0-1)	**0.05**
TrunkTaperPercent (percentage: 0-1)	**0.4**
NumBranchControlPointsOnLobe	**55**
NumLeafPointsOnLobe	**30**
MaxTrunkHeightPercent (percentage: 0-1)	**1**

8.2.3 Weeping Cherry Tree

TrunkAngle (degrees)	**3**
BranchAngle (degrees)	**45**
SubsequentAngle (degrees)	**15**
NumTrunkIterations	**2**
BranchIterations	**1**
LeafTaper (percentage: 0-1)	**0**
NumTrunkBranchesOffSplit	**1**
NumBranchesOffSplit	**2 +- 1**
NumSegmentsInBranch	**4**
SegmentOffset	**0.4, 0.4, 0.4**
LengthReductionFactor (percentage: 0-1)	**0.2**
BranchStartWidth	**0.1**
TrunkStartWidth	**0.25**
TwigStartWidth	**0.005**
MinBranchWidth	**0.005**
BranchTaperPercent (percentage: 0-1)	**0.2**
TrunkTaperPercent (percentage: 0-1)	**0.3**
NumBranchControlPointsOnLobe	**6**
NumLeafPointsOnLobe	**50**
MaxTrunkHeightPercent (percentage: 0-1)	**1**

8.3 – Tree File Format

The tree file format contains the following data:

[Tree Type]
[Random number generator seed]
#BRANCHES
[Branch id] [parent id (-1 = no parent)] [startX] [startY] [startZ] [endX] [endY] [endZ]
#LEAF LOBES
[x] [y] [z] [radius]

Where Tree Type is a string representing what tree rules the tree will be generated using; Random number generator seed is an integer that will seed the application's random number generator; startX, startY, and startZ are the x, y, and z components of the branch's starting vector; endX, endY, and endZ are the components of the branch's ending vector; x, y, and z are the components of the lobe's center vector; and radius is the radius of the lobe.

For example:

Cherry Tree
100
#BRANCHES
0 -1 1.110022 -1.582516 4.728205 1.14952 -0.8700994 4.896452 1
1 0 1.14952 -0.8700994 4.896452 1.800362 -0.3780254 4.764441 1
2 0 1.14952 -0.8700994 4.896452 0.7252637 -0.7159721 4.99926 1
3 2 0.7252637 -0.7159721 4.99926 0.5519101 0.02624928 5.070594 1
#LEAF LOBES
2.307282 0.5082518 4.534246 1
0.4439276 1.201943 4.939856 1