Pomegranate

Procedural 3D Tree Creation via User-Defined L-systems

Jeremy Berchtold
Department of Computer Science & Software Engineering
California Polytechnic State University, San Luis Obispo
March, 2021

© 2021 Jeremy Berchtold
Abstract

Pomegranate creates procedural 3D trees based on a user-specified template. The template supports randomness and allows users to generate an entire forest of unique trees from a single template. The output trees are a single closed mesh without intersecting geometry (with the exception of leaves). Additionally, the output contains a skeletal rig used for animating the trees. Pomegranate produces textured trees that can use either a realistic or stylized look, as well as supporting different mesh densities for games or film. Since this project uses a procedural workflow, artists can quickly create and make edits to their trees. This increase in work speed saves time for artists and lowers costs for studios. Pomegranate is a stand-alone command-line executable and was written in C++.

Introduction

Games, films, and advertisements all rely on 3D art. However, 3D art is time consuming and expensive to create. Artists spend many hours creating models. Furthermore, many of these models are discarded or have to be modified significantly in the revision process, costing even more time. This slow process hinders the artist's creativity and also costs more to the companies who hire them.

In these situations, a procedural workflow is helpful. Procedural workflows allow artists to create formulas and procedures for 3D art, much like programming. This allows artists to later modify models quickly and easily to their employer's liking. Additionally, it can allow artists to create variations of the same 3D asset faster and easier than starting from scratch. Procedural workflows are becoming increasingly popular, especially using programs like Houdini.

Creating formulas and procedures for buildings, machines, and other repetitive and mathematical objects is not too difficult. However, due to the complexity and variation of trees, it is difficult to create procedures to create them. As a result, artists will often non-procedurally model and texture trees which is a slow and expensive process.

This project is able to generate 3D trees procedurally. It outputs textured tree assets. Additionally, this project supports randomness so the user-defined procedure can be used to generate an entire forest of trees all stemming from a single procedure. Furthermore, the output tree meshes have manifold topology. A mesh with manifold topology is a mesh that can be unfolded into a single continuous surface\[21\]. This topology is important as non-manifold topology is more difficult to animate and is completely unusable for certain algorithms. Overall, this project provides a faster way for artists to create trees, allowing for rapid iteration of designs, as well as generating random variations from a single template.
Related Work

L-systems

L-systems are used throughout commercial solutions and papers to generate trees. An L-system is a set of replacement rules, each with a list of commands. From an initial list of commands, these replacement rules can be repeatedly applied to yield recursive plant-like structures\(^\text{[1]}\). The initial list of commands, rules, and other parameters are specified by the user in a configuration text file. The format of this file is documented more thoroughly in the appendix.

The commands of an L-system are used to generate a recursive structure. This is very beneficial since trees tend to have self-similar structures. The example below demonstrates the evaluation of a simple 2D L-system. The command \( F \) means go forward in the current direction and create a segment. The command \( \pm \) means rotate clockwise by the angle specified (this is extended into separate commands for pitch, yaw, and roll for 3D L-systems).

### Evaluation of a Simple 2D L-system

<table>
<thead>
<tr>
<th>Initial State: ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle: 45°</td>
</tr>
<tr>
<td>Rules:</td>
</tr>
<tr>
<td>( A = +FA )</td>
</tr>
</tbody>
</table>

\( Gen \ 0 \)

State: \( A \)

\( Gen \ 1 \)

State: \( +FA \)

\( Gen \ 2 \)

State: \( +F+FA \)

\( \ldots \)

This system is expanded to 3D space. Additionally, the commands "$[" and "]$" allow the user to push and pop the current position, rotation, and scale. This allows for L-systems to generate recursive branching structures.

Commercial Solutions

There are several other existing solutions for generating trees. These include SpeedTree\(^\text{[8]}\), SideFX Labs Quick Tree\(^\text{[7]}\), and using L-systems in Houdin\(^\text{[6]}\). However, these all suffer from the same issue. They don’t generate a single manifold mesh for the trees. Each branch is a disconnected piece of geometry with the individual branch meshes overlapping. This is sufficient for some use cases, however not generating a single manifold mesh can be very limiting. A single manifold mesh is required for any algorithm that requires consistent surface
parameterization, such as pathing along the surface. Additionally, skeletal animation performs significantly better with a single manifold mesh. Having the mesh in multiple pieces can negatively interfere with how well the mesh can support animation. The mesh could stretch or pull apart in unintended ways when animated.
Related Papers

In addition to the commercial software that exists, there are several related papers. In 2017, Charlie Hewitt published “Procedural generation of tree models for use in computer graphics” which contains techniques for generating 3D trees\(^9\). This paper produces very visually appealing results. Additionally, this paper does implement object-based pruning which ensures the tree will not grow into surrounding objects. Overall, this paper produces great trees that are useful for many purposes. However, it does not produce a single closed mesh. The disconnected pieces of the mesh intersect, which makes the resulting mesh not usable for some applications.

Another paper, “Modeling of Trees with Interactive L-System and 3D Gestures”, written by Onishi, et. al. covers an alternative technique for creating L-systems\(^{10}\). Instead of directly writing the L-system commands manually, they created software that allows artists to draw the shape of the tree and that produces L-system commands. This is very useful for artists who want to create trees quickly and it allows for a bit of a procedural approach as the tree is still based on an L-system. However, some use cases, such as generating an entire forest of unique trees, require the entire process to be fully automated and procedural.
Algorithm

Algorithm Overview

First, the algorithm reads in a user-specified text file containing the L-system that defines the formula for generating trees. This file is then parsed and converted into a set of parameters, a starting command list, and a list of replacement rules.

This list of rules is recursively applied for the user-specified number of generations to yield a single list of commands for generating the tree. The commands are then evaluated to give a data structure containing all segments of the tree in 3D space.

Each segment is then converted into polygons. Most segments are converted into a simple cylinder, however leaves are converted to a single quad plane, and segments with no children are converted to a cone. All geometry generated uses UVs and is assigned a material according to the rule that generated it. This is used for texturing the resulting asset.

These disconnected segments are then joined into manifold topology (with the exception of the leaves). The algorithm for branch topology will be explained in a later section and is the heart of this project. In short, the branches are joined first by bridging edge loops, then a manifold hull is generated to remove intersection geometry while still preserving edge flow as much as possible.

Lastly, extra leaves are added. If a segment has no children and the rule that generated it is tagged as leafable, leaves will be added according to a special leafable rule. This allows trees to always have leaves at their end caps, regardless of the number of generations.
L-system Evaluation

The first step of this algorithm is to read in the user-specified configuration file. This file contains all the information necessary to evaluate the L-system. Once the input has been parsed, the L-system commands are evaluated. The output of this step will be a set of segments. Each segment has a transformation as well as a length. This data is just an internal representation.

Additionally, this stage outputs a CSV file containing information necessary to create a skeletal rig. This rig is used to animate the tree in 3D animation software like Houdini. The rig can be used to create effects like wind, or to transform the branches of the tree however the user would like.

Instancing Segments as Polygons

The next step in the algorithm is to convert the L-system segments into cylinders. For each segment, the algorithm creates two circles, one at the start of the segment, and one at the end. The circles are created in the YZ-plane and then multiplied by the segment’s transformation matrix so they are translated and oriented correctly. The end circle is scaled to match the last child of this segment’s start (conventional place for the recursive rule) to allow a smoother taper between segments. If the segment has no children, the end circle is reduced to
a single point to ensure a closed mesh. Faces are then added to join the circles together to create a cylinder.

Now that the geometry is created, we can assign texture coordinates (UVs) to the vertices. The input textures must be tileable along both axes. The UV x-axis is used to wrap along the cylinder and goes perpendicular to the direction the segment is pointing. The UV y-axis goes along the direction of the segment. This UV coordinate system allows for tileable trunk, bark, or other textures to be applied to the procedural geometry.
After all segments are instanced, the result is something similar to this structure below.

Manifold Branch Topology

The goal of this section of the algorithm is to join the disconnected segments with manifold topology. The following steps in this section are applied to each branch point, in other words, where a parent segment should be connected to more than one child. The case of a parent with a single child is trivial and is handled with a single edge loop bridge between the parent and child.
The disconnected segments are joined by bridging their edge loops. For the parent the end loop is used. For the children the start loops are used. Any edge that is used to connect vertices of separate loops will be referred to as a cross edge. The bridge edge loop algorithm calculates the best fitting connections between corresponding vertices by minimizing the sum of the length of all cross edges. This ensures there is no twisting that could cause self-intersecting geometry.
From here, the program calculates the intersection points between all cross edges and faces of the other edge loop bridges. The intersection points are stored in a list. Each intersection point stores the position of the intersection, a reference to the cross edge, a reference to the face the edge intersected with, and parametric position $t$ that the intersection occurred at along the edge. The $t$-value goes from 0.0 to 1.0, with a value of 0.0 being at the child segment's loop, and a value of 1.0 being at the parent segment's loop. After all intersections have been calculated the intersections are filtered. For each edge, only the intersection with the smallest $t$-value is kept. This has the effect of only keeping the outermost intersections and ignoring internal intersections. This is beneficial since the goal is to create a manifold hull of these intersecting edge loop bridges.
Next, each cross edge with an intersection is split at the intersection point. A new vertex is created at the intersection point. This new vertex replaces the vertex on the parent edge loop for this edge as well as the faces adjoining this edge. This process is applied to every edge, yielding branch topology that does not have edge-face intersections, but now has holes.

The newly created vertices from the edge splits are now used to join cross edges to the faces they intersected with. To do this the intersection points are grouped according to the face they intersect with.

For each face, the newly created vertex of each cross edge is added to a list. The vertices in this list will be merged with F2 or point F3 (shown below). These points are determined by keeping track of vertex ordering when creating the faces of the edge loop bridges. In this implementation, the first and second vertices in the face always correspond to vertices created from an edge split. Therefore, this vertex will be close to the intersecting cross edge vertices. The newly created vertices on the cross edges are then merged into F2 or F3, whichever is closer to the vertex to be merged.
This process is then applied to all edge loop bridge faces with intersections. This method preserves quads and edge flow as much as possible. In many cases only a few triangles are created. Triangles only occur when a face has 3 or more intersections, or when 2 intersections both are closer to the same vertex (F2 or F3).

This method can occasionally produce holes at the parent edge loop resulting from the step when all edges are split. This is fixed by finding holes once the branch topology is finished. The holes are found by finding all open edges, edges with only one face attached and then grouping them by determining the connected components of the open edge graph. These vertices of these groups of open edges are then merged together until there are no more open edges.

Additionally, this method for merging vertices preserves proper UV coordinates. The cross edges in the original bridges have the same UVs as a cylindrical segment ($UV_x$ wraps around and $UV_y$ goes along the length of the cylinder). The edge splitting adjusts UVs proportionally to the t-value of the intersection. The merging operation preserves UVs by allowing individual vertices to have different UVs per face. When vertices are merged, a UV override is added to one of the faces, ensuring both vertices’ original UV coordinates are preserved.
Final Branch Topology (Exterior View)

Final Branch Topology (Interior View)

Final Branch Topology with UVs
Adding Leaves

Simple Leaves

Leaves are implemented as planar quads. They have UVs that map over the entire texture space (0-1 along x, and 0-1 along y). Leaves allow for creating fuller trees. Leaf textures can be a single leaf or include multiple leaves and even small branches in the texture. Leaves are created by simply tagging a rule with [leaf] in the user-specified configuration file.

Leaf Texture with Alpha Channel [16]                        Single Planar Quad

Leaf with Texture

Leafable Rule

Leaves are useful, however adding commands to generate leaves at the end of all rules can be time-consuming and make the overall command list more cluttered. The solution to this is to tag certain rules as leafable by using [leafable] in the user-specified configuration file. If a segment has zero non-leaf children and the rule that generated it is tagged leafable, extra leaves will be added. The leaf pattern is defined in a leafable rule. This allows leaves to easily
be added to the end of all branches, creating a fuller, more realistic tree without much additional effort.

Results

Renders

The following images were rendered in Blender with the Cycles and Eevee engines for the realistic and stylized renders, respectively. All trees were generated by this project directly. Other 3D assets in the scenes were created manually. Textures from a variety of sources were used. See texture citations below each render for more information.
Realistic Forest Scene at a Lake. Textures [16, 19]

Realistic Forest Scene at a Lake (Alternate View). Textures [16, 19]
Stylized Tree Render. Bark texture [20]. Leaf texture was made by myself.

Stylized Tree Full Scene Render. Textures [15, 18, 20]. Leaf texture was made by myself.
The following images show an example rig outputted from this project. The rig allows the user to animate parts of the tree. This can be useful to add effects such as wind. The first image shows the tree before it has undergone and deformations. The last two images show the tree after it has been deformed, with each image showing a different deformation.
The biggest takeaway from this project was the visualization techniques I learned while debugging my code. If I ran into bugs, I couldn’t simply read through thousands of matrices to spot the bug. Instead, I generated relevant output data into a CSV table and visualized it within Houdini with a custom visualization setup.

For example, I had a bug with my transformation matrices of the individual segments. I tried manually looking through the matrices as well as running debugging code to perform checks on each matrix (i.e. are basis vectors orthogonal). However, these did not give me enough information. Instead I wrote each 4x4 matrix into a CSV file, with each row containing a single matrix. In Houdini, I imported the table. Then I wrote a visualization script to extract the basis vectors from the matrices and render red, green, and blue lines for the x-, y-, and z-axes, respectively. I then translated each set of axes to the position specified by the matrix’s transform. The following is a screenshot of that visualization. Using this visualization while I debugged my code was enough to find out the issue and fix it.

Reflections

The biggest takeaway from this project was the visualization techniques I learned while debugging my code. If I ran into bugs, I couldn’t simply read through thousands of matrices to spot the bug. Instead, I generated relevant output data into a CSV table and visualized it within Houdini with a custom visualization setup.

For example, I had a bug with my transformation matrices of the individual segments. I tried manually looking through the matrices as well as running debugging code to perform checks on each matrix (i.e. are basis vectors orthogonal). However, these did not give me enough information. Instead I wrote each 4x4 matrix into a CSV file, with each row containing a single matrix. In Houdini, I imported the table. Then I wrote a visualization script to extract the basis vectors from the matrices and render red, green, and blue lines for the x-, y-, and z-axes, respectively. I then translated each set of axes to the position specified by the matrix’s transform. The following is a screenshot of that visualization. Using this visualization while I debugged my code was enough to find out the issue and fix it.
This experience with debugging large amounts of data will be helpful regardless of the career area I choose within the field of Computer Science. Being able to create visualizations is especially useful for graphics programming (whether debugging or the final result), but it is also helpful for many other fields. Machine learning often uses large amounts of data so it can also benefit from visualizations. Additionally, almost any field within Computer Science can benefit from it as creating meaningful visualizations from code can help others, whether a manager, a client, or a team member, understand your work better.

Future Work

Future Improvements

The first improvement is blending between parent and child branches so the branches can curve out instead of going straight out at a sharp angle. This would allow a more smooth transition between segments that looks more natural. This could be implemented by adding more rings to the edge loop bridges. This is non-trivial because the branch topology code would need to be significantly modified to support multiple faces per cross edge.

The second improvement is adding procedurally generated skinning weights for the tree’s skeletal rig. The automatic skinning in 3D software is fairly good, however, it sometimes will attach weights across branches if they are too close together. Since this project is generating the geometry for each segment, it could assign rigging weights to the bones and joints accordingly. This would allow for more accurate results when animating.
The last improvement is to prevent branches from intersecting each other. This is a difficult problem to solve. A trivial solution could likely be implemented by simply rotating a branch away if it would intersect another. However, this could lead to unnatural looking trees and weird behavior that the user does not expect. A more advanced solution would be to implement small IK rigs and adjust multiple segments at a time when moving branches to prevent intersections. This could provide a smoother tree while still preventing self-intersections.

Project Expansion

A larger addition would be to expand this project to not only trees, but characters as well. Characters are significantly more difficult than trees because of several reasons. First, characters have a wide variety of shapes, a simple cylindrical segment will not work. Second, joining these shapes requires more advanced topology manipulation. Third, animation-ready topology is dependent on how the character will move, which is not solely dictated by the shape.

A solution to this would be to create a hybrid procedural/traditional system. Individual pieces like arms, heads, and torsos would be created using traditional methods (sculpting, re-topology, etc.). These individual pieces would then be instanced and placed in 3D space. After the artist is happy with the placement of the individual pieces, the procedural topology merge would take place. This would result in a hybrid approach allowing artists to quickly create new characters from a base asset library of pieces.
References

L-system Tutorials

Related Works

Libraries
Parser generator used for parsing the user-specified configuration file.
Command-line argument parser.
Basic vector and matrix math library.

Mesh editing library.

**Note:** OpenMesh was not used for the final project. The restrictive topology and winding-order rules in OpenMesh made topology manipulation unnecessarily difficult. OpenMesh was only used for the initial prototyping stages. A small custom mesh editing library was used for the final project.

**Final Render Resources**


[16] CC0 Textures - Free Public Domain PBR Materials, cc0textures.com/.


**Other**

Appendix

Configuration File Format

generations=INT,
angle=FLOAT,
ringvertexcount=INT,
ringsperworlddist=FLOAT,
minringspersegment=INT,
init=COMMANDS,
leafable=COMMANDS,
rules:
RULE

Key

- Required
- Optional

Rule

<table>
<thead>
<tr>
<th>name</th>
<th>isleaf</th>
<th>isleafable</th>
<th>commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>[leaf]</td>
<td>[leafable]</td>
<td>COMMANDS</td>
</tr>
</tbody>
</table>
Table of L-system Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Command</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>-(angle)?</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-(15)</td>
<td>-(0 dev 30)</td>
</tr>
<tr>
<td>Right</td>
<td>+(angle)?</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+(15)</td>
<td>+(0 dev 30)</td>
</tr>
<tr>
<td>Roll Clockwise</td>
<td>/(angle?)</td>
<td>/(15)</td>
</tr>
<tr>
<td></td>
<td>/(0 dev 30)</td>
<td></td>
</tr>
<tr>
<td>Roll Counterclockwise</td>
<td>/(angle?)</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>/(15)</td>
<td>/(0 dev 30)</td>
</tr>
<tr>
<td>Pitch Up</td>
<td>&amp;(angle?)</td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td>&amp;(15)</td>
<td>&amp;(0 dev 30)</td>
</tr>
<tr>
<td>Pitch Down</td>
<td>^(angle?)</td>
<td>^</td>
</tr>
<tr>
<td></td>
<td>^(15)</td>
<td>^(0 dev 30)</td>
</tr>
<tr>
<td>Scale Length</td>
<td>&quot;(scale)</td>
<td>&quot;(0.5)</td>
</tr>
<tr>
<td></td>
<td>&quot;(2.0 dev 0.5)</td>
<td></td>
</tr>
<tr>
<td>Scale Uniform</td>
<td>@(scale)</td>
<td>@(0.5)</td>
</tr>
<tr>
<td></td>
<td>@(2.0 dev 0.5)</td>
<td></td>
</tr>
<tr>
<td>Scale Radius</td>
<td>`(scale)</td>
<td>`(.5)</td>
</tr>
<tr>
<td></td>
<td>`(2.0 dev 0.5)</td>
<td></td>
</tr>
<tr>
<td>Push</td>
<td>[(branchChance?)</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>[(0.5)</td>
<td>[(0.75)</td>
</tr>
<tr>
<td>Pop</td>
<td>]</td>
<td>]</td>
</tr>
<tr>
<td>Forward</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Move Forward (no segment)</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>Rule</td>
<td>ID</td>
<td>Trunk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leaf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Branch</td>
</tr>
</tbody>
</table>

Notes:

- ? denotes the parameter to the command is optional
- 15 dev 30 is a random value. Each time this command is evaluated it will produce a different value sampled from a normal distribution with mean 15, and standard deviation 30.