Kathryn Williams
CSC 4357
May 8, 2014

Tree Generation

Introduction

My overall goal for the semester was to do a graphics related project involving trees. Originally I had planned to implement the paper *Plastic Trees: Interactive Self-Adapting Botanical Tree Models* by Pirk et al., which involved dynamically rearranging trees and having them adapt to the environment in real-time. This implementation required me to have a skeleton-based tree model as an input. I scoured the Internet for such models, and came across a bunch created by the program Xfrog. These tree models included a file used for input for the Xfrog program which outlined the structure of the tree. I set out to translate this file into a data structure I could use, however, with little documentation available, this task proved to be too difficult.

I then set out to create my own trees, which turned into the main focus and goal of my project. The background to my tree generation and its implementation and results are described in this paper.

Background

The tree generation algorithm that I implemented was based on the paper *Self-Organizing Tree Models for Image Synthesis* by Palubicki et al. The paper first describes the general structure of a tree and defines important terms, then outlines a very general algorithm for tree generation. It then proposes a few different options for implementing certain parts of the algorithm.

The basic structure of a tree is as follows: a tree is made up of multiple branches. Each branch is made up of multiple nodes. The branch segments between these nodes are called internodes. There are three key types of nodes. The first is a terminal bud, which only occurs at the end of a branch. When this bud produces a new shoot, it becomes an axillary bud. An axillary bud is a bud existing along the axis of a branch. When this bud produces a new shoot, it becomes a branch node. A branch node is simply a node existing in one branch that serves as the base of a separate branch.

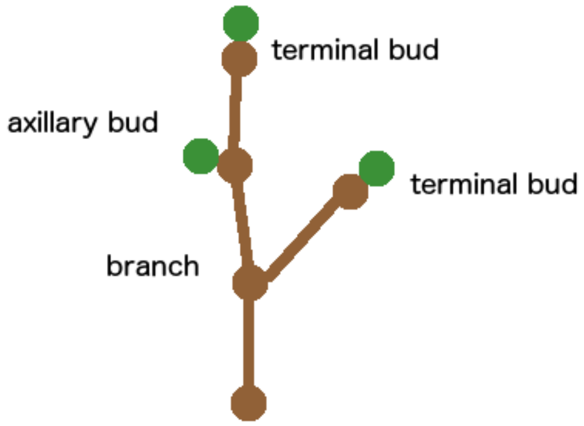

Fig. 1: Types of nodes

While the particular details can be quite complicated, the general tree generation algorithm is fairly straightforward. We first calculate the local environment surrounding each bud. Based on these calculations, we determine each bud's fate: whether it grows a shoot or not. For those buds that we determine should grow, we add those new shoots to the tree. Then we repeat the process, for however many steps specified.

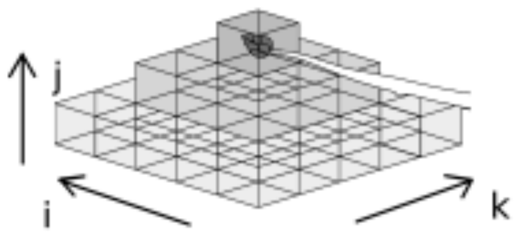

Fig. 2: Shadow propagation model

The paper suggests two different methods for calculating the local environment, one being a space-based calculation, and the other being a light-based calculation. I chose to implement the light-based calculation with the hopes that it would make the tree look more realistic. The light-based calculation keeps track of a grid of voxels within which the tree is contained. When a bud occupies a voxel (I, J, K), it propagates a pyramidal penumbra to all voxels (I ± p, J – q, K ± p) below, where q = 0, 1, …, q_{max} and p = 0, 1, …, q. Each affected voxel's shadow value is then increased by $\Delta s = ab^{-q}$, where a > 0 and b > 1 are user-defined parameters. The environmental input Q for each bud is based on the light in that bud's voxel and is calculated using the equation $Q = \max(C - s + a, 0)$, where C is a user-defined parameter representing full exposure.

The paper also suggests two different models for resource allocation, one being the extended Borchert-Honda model and the other being the priority model. I chose the latter because the paper mentions that it's generally faster. The priority model propagates the light information basipetally, starting from the ends of the newest branches and working toward the base of the oldest branch (the trunk). The base of each branch contains the sum of all environmental inputs for that branch. For each bud node, terminal or axillary, the environmental input is simply Q. For branch nodes, the environmental input is that of the base of the branch attached. In addition to storing the sum in the base of a branch, the average input over all child buds is also stored.

$$v_i = v\frac{Q_i w_i}{\sum_{j=1}^{N} Q_j w_j}, i = 1, 2, \dots, N$$

Eq. 1: determining node resources

$$w_i = \begin{cases} \frac{-(w_{min} - w_{max})i}{\kappa N} + w_{max}, & if\ i < \kappa N \\ w_{min}, & if\ i \geq \kappa N \end{cases}$$

Eq. 2: determining priority weight for each node

After all environmental inputs are calculated, we can distribute resources among the nodes. The base resource value, v_{base}, is directly proportional to the total environmental input value at the base of the trunk of the tree Q_{base}. We distribute these resources acropetally, starting from the base of the trunk of the tree, working towards the extremities. To determine how much resources are distributed to a node, we must first sort all the nodes on a branch by their average environmental input. Resources are then distributed based on a node i's sorted index, using Equation 1, where v is the branch's base resource value, N is the total number of nodes on this branch, and w is a weight determined by a piecewise function in Equation 2.

After we distribute resources, the only step remaining is to grow new shoots. If a bud node has a resource value less than 1, then it does not grow a new shoot. Otherwise, the bud will grow a new shoot. The number of internodes n of a shoot stemming from node i is the value of floor(v_i). The length l of these internodes is defined as $l = v/n$.

The direction of the new shoots is a normalized combination of three vectors: the default direction, the optimal direction, and the tropism direction. If the bud is a terminal bud, the default direction of the internode is the same as the previous internode. If the bud is an axial bud, the default direction of the internode is the direction of the previous internode, with a 50% chance of adding 45° and a 50% chance of subtracting 45°. To find the optimal direction we look at the neighboring voxels within a cone with a 45° angle extending from the node in the default direction of the node. Of these neighboring voxel, we select the voxel with the minimum shadow value, and define the optimal direction as the direction from the current node to the center of that voxel. The tropism vector is a user defined parameter that takes into account the tendency to grow towards light, gravity, etc. These vectors are summed, with a user-defined weight assigned to each, and then normalized to produce the final direction of a new shoot.

Implementation

My implementation is divided into two key files. The first is TreeGenerator.cpp, which contains the code for generating a tree structure. The second is TreeRenderer.cpp, which contains the code for rendering a tree structure produced by the generation process. The main.cpp file is the entry point for the program and implements some simple GLUT functions to reshape, draw, and interact with the produced tree model.

The most important function in the TreeGenerator class is the grow_tree function, which accepts as an argument the number of steps to take in the generation process. The resulting tree is stored in a vector of branch structures called branches, which is a publicly accessible variable. There were a number of parameters I needed to define when implementing the algorithm mentioned in the paper. The size of the shadow voxel grid is 200x200x200. The depth of shadow propagation q_{max} is defined as 4. The value of full exposure C is set at 1. Two variables involved in calculating shadow values are defined as a = 0.2 and b = 2. For determining node resource weights, I used values suggested in the paper: w_{max} = 1.0, w_{min} = 0.006, and kappa = 0.5.

To render the tree, I use a separate vertex buffer object for each branch structure in the tree, with the original intent of incorporating this into the interactive plastic trees project. When looking at a branch structure, I give each branch node a radius of 0.5. I calculate a ring of 5 points around the circumference of each branch node circle. Using triangle strips, I create cylinders between the branch nodes. When rendering these triangle strips, I color them brown to look like trees. In the current release, no normal or texture vertices are calculated, although I would like to do that for a future release.

A Makefile is included with the program, so compiling the programing is straightforward using *make*. To run the program, simply run *./trees <number of steps>*. If the number of steps is left out, it defaults to three steps.

Results

The implementation based on this tree generation method produced some fairly realistic trees. There's certainly room for improvement, but the tree structure is evident when looking at the results. For my implementation, only a certain number of steps can be taken without running into boundary issues, however it doesn't take too many steps to produce something that looks like a tree. I recommend 5-15 steps. It doesn't take long at all to process and render.

There's an odd tendency for a tree to start growing towards the right or left rather than straight upward. This may have something to do with the selection of minimum voxels: when there are multiple voxels with the same minimum value, it selects the first, which may be far from the default direction.

Another odd thing is how the tree stem grows really tall, but branches seem to not stretch out as much. If I implement branch shedding for branches that receive too few resources, this may help with distributing resources among branches still growing, while also giving the tree a more typical tree structure.

The most influential parameter on the final tree structure seemed to the weights assigned to the different directions. To demonstrate the different directions and their effect on the tree shape, shown below are several outputs of the program using different weights for different directions. Also illustrated are some of the issues I just described involving the final structure.

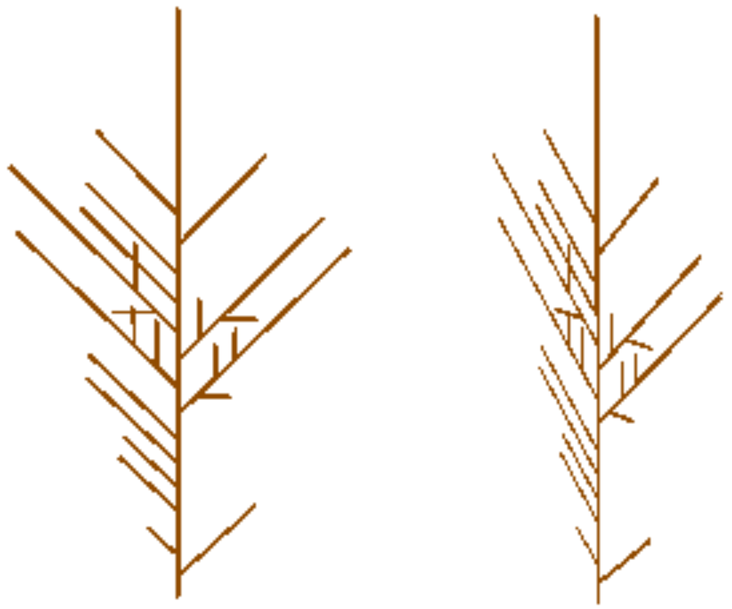

Fig. 3: A 5-step tree with all the weight on default direction and no weight on the other directions. Note that it only exists in 2D space, because the direction of new shoots is decided to be 45° right or 45° left of a node's current direction

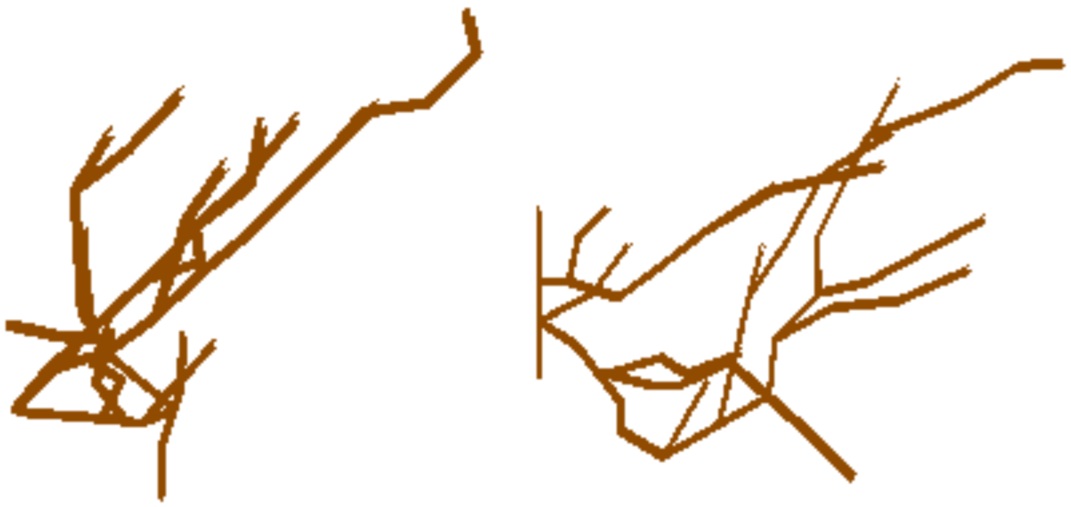
Fig 4: A 5-step tree with all the weight on the optimal direction and no weight on the other directions. The issue with leaning left/right is clearly visible in this model.

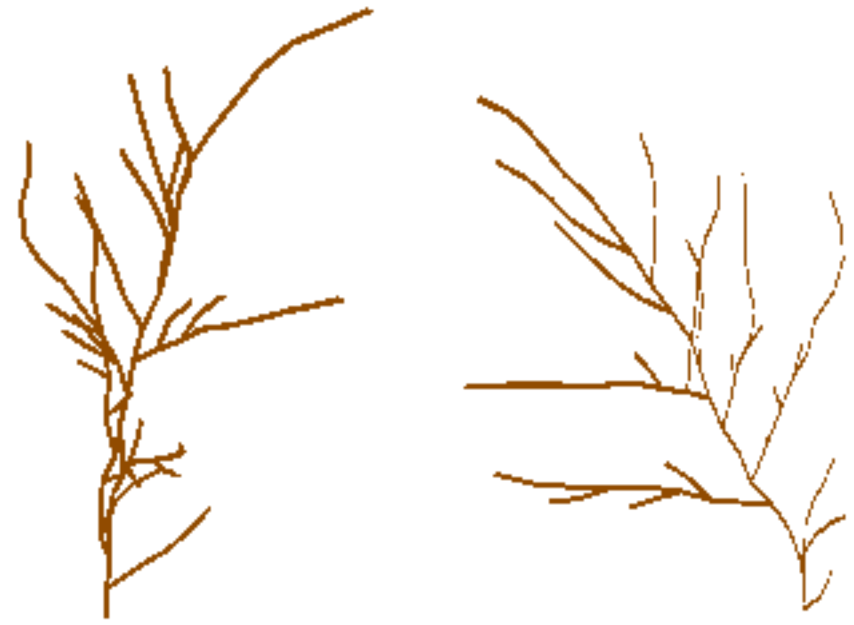
Fig. 5: A 5-step tree assigning a weight of 0.5 to the optimal direction, 1.0 to the default direction, and no weight on the tropism direction. Looks more like a tree, but still heavily leaning right or left.

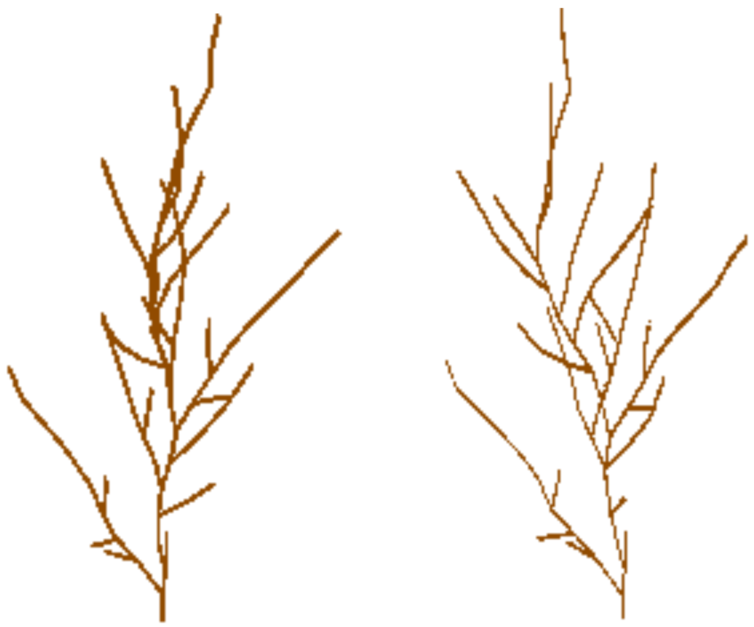

Fig. 6: A 5-step tree assigning a weight of 0.5 to the optimal direction, 1.0 to the default direction, and 0.3 to the tropism direction. The tropism direction is straight up (<0, 1, 0>). This produces much more realistic trees.

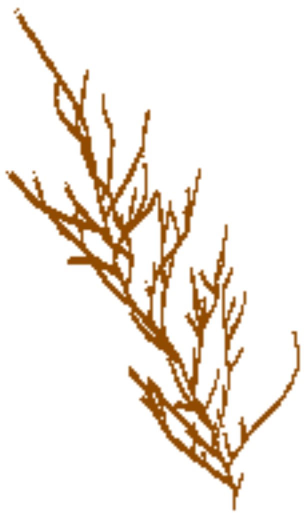

Fig. 7: A 7-step tree with the same parameters used for Fig. 6. As it grows more, the leaning becomes more prominent again.

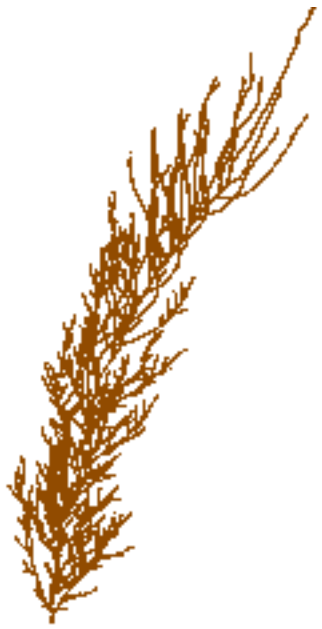
Fig. 8: A 10-step tree: Leaning is also prominent here, as well as the issue of the stem growing very large while branches remain relatively short.

Future Work

Much can be done to improve the results of this implementation, largely involving finer controls over certain parts of the implementation. For example, in my implementation, when sorting the nodes by average light value, I always give highest priority to the terminal nodes by setting their sorted index to 1. The paper recommends doing this towards the beginning of generation, but then ceasing to bias the terminal node after a while.

Some parts of the paper were left unimplemented, but would, if implemented, result in a better-looking tree. The two parts I didn't include were calculating the branch width, and shedding branches. Branch width would obviously give the tree a more appealing tree-like look, and shedding branches would certainly help give it a more realistic structure.

Right now a tree structure is generated and rendered, which looks nice; however this is not necessarily useful yet. To make this a more useful program, we could give an option to output data to a file for the user to use later. A .obj file would be useful for those just wanting to use the tree model generated. Another file containing information about the tree structure would also be really useful for those desiring to analyze tree structures.

The cherry on top would be to implement a GUI for tree generation. There are a lot of user-defined parameters that can change the outcome of the tree structure, and it would certainly be convenient to have this in an easy-to-use graphical user interface rather than having to modify the source files to change all the variables. This could also be a very practical way of packaging this for use by anyone – artists or researchers.