IGR
Final semester project

Procedural geometry generation for plants and trees

Based on the paper:

Bosheng Li, Nikolas A. Schwarz, Wojtek Pałubicki, Sören Pirk, and Bedrich Benes. 2018. Interactive Invigoration: Volumetric Modeling of Trees with Strands. In . ACM, New York, NY, USA, 13 pages

PERONNET Antonin

1 Related work

The goal of this project is to generate the entire plant geometry (the mesh including colors) from no data.

In most plant mesh generation pipelines, there are 3 main steps:
- creating the skeleton (macro-level representation)
- computing the details (micro-level)
- generating the geometry

1.1 Skeleton generation

The first approach that was used to generate skeleton of trees was fractal modeling. The basic idea is that a branch contains other sub-branches, which look a lot like the main branch, but scaled down. The idea of L-systems[1] generalizes this idea. It consists of rules (similar to formal grammars) to generate segments recursively.

Nowadays, biological models of plants are used to simulate the growth. A very sophisticated one has been made to include wood fracture for example.[2]

The paper I will use depends on the work from an older paper, that simulate the growth of the roots of a tree[3]

1.2 Micro level

Genereating the micro-level gemoetry of a tree is very resource-intensive. To have a perfect simulation, one would need to simulate every cell, which can interact in complicated ways (such as the patterns in brak and dendrites).

As for many complex system, an succesful approach is particule-based[4]. Particles represent unitary volumes in the tree, in he same way as they can represent unit volumes of water in fluid simulations.

The main innovation of the paper is to use strands, sort of threads that interpolate between particles.

1.3 Geometry

Very often, the geometry part uses very well established techniques in the field of computer graphics. The most used techniques are:
- delaunay triangulation
- convex hull algorithms
- cubic splines
- mesh techniques such as smoothed normal calculation, mesh smoothing.

[1]P Prusinkiewicz. 1986. Graphical Applications of L-systems

[2]

- T. Hädrich, J. Scheffczyk, W. Palubicki, S. Pirk, and D. L. Michels. 2020. Interactive Wood Fracture. In Eurographics/ACM SIGGRAPH Symposium on Computer Animation Posters. The Eurographics Association.

[3]Li, J. Klein, D. L. Michels, B. Benes, S. Pirk, and W. Pałubicki. 2023. Rhizomorph: The Coordinated Function of Shoots and Roots. ACM Trans. Graph. 42, 4, Article 59 (jul 2023), 16 pages.

[4]X. Chen, B. Neubert, Y.-Q. Xu, O. Deussen, and S. B. Kang. 2008. Sketch-Based Tree Modeling Using Markov Random Field. ACM TOG 27, 5, Article 109 (Dec. 2008).

2 Implementation

From a very high level, the pipeline is the following:

$$\begin{tikzcd}
{} \arrow[d, "\text{tree caracteristics}"] \\
\boxed{\text{1) Plant Graph simulation (growing)}} \arrow[d, "\text{skeleton}"] \\
\boxed{\text{2) Particle flow and interpolation}} \arrow[d, "\text{strands}"] \\
\boxed{\text{3) Contours calculation}} \arrow[d, "\text{contours (branch sections)}"] \\
\boxed{\text{4) Meshing and branch fusion}} \arrow[d, "\text{3d mesh}"] \\
{}
\end{tikzcd}$$

I focused on the particle and meshing parts (2, 3, 4) and spend very little time on the skeleton calculation (1). Thus, I will present the steps 2), 3), 4) and 1) at the very end.

For this project, I used bevy, a rust cross-platfrom game design library. I decided to use it because of the following features:

- a `gizmos` tool that allow to draw 3d points and lines, vey useful for debuging
- a very good keyboard and mouse support
- a dedicated way to store assets, resources and properties
- the ability to compile to `wasm` and `webgl`

My work is available on github under the MIT license and can be tested on a web interface here: https://rambip.github.io/plant-mesh/

In the online demo, a tree is generated automatically every 5s. The user can visualize the different steps (see Section 4). I did not implement the ability to change the parameters, but it can easily be added[5]

[5]In the desktop version, the parameters are loaded from a config file.

2.1 Step 0: Defining the skeleton

The skeleton is a simple tree structure with some properties stored in each node:

```
pub struct PlantNode {
    children: Vec<PlantNode>,
    props: PlantNodeProps,
}

pub struct PlantNodeProps {
    // a 3d vector
    pub position: Vec3,
    pub radius: f32,
    // a normalized 3d vector
    pub orientation: Vec3,
}
```

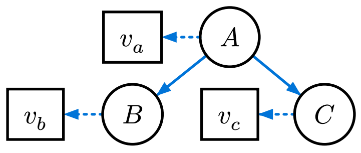
> **Note**
>
> An optimization I did later to make the code easier to understand is to represent the orientation each node by a quaternion. This way, the transformation to go from an absolute plane to a branch and back are just multiplication by the quaternion.

I decided to restrict the tree to a binary tree (each node has 0, 1 or 2 children) like in the paper. It makes the particle projection and the branch fusions a lot easier.

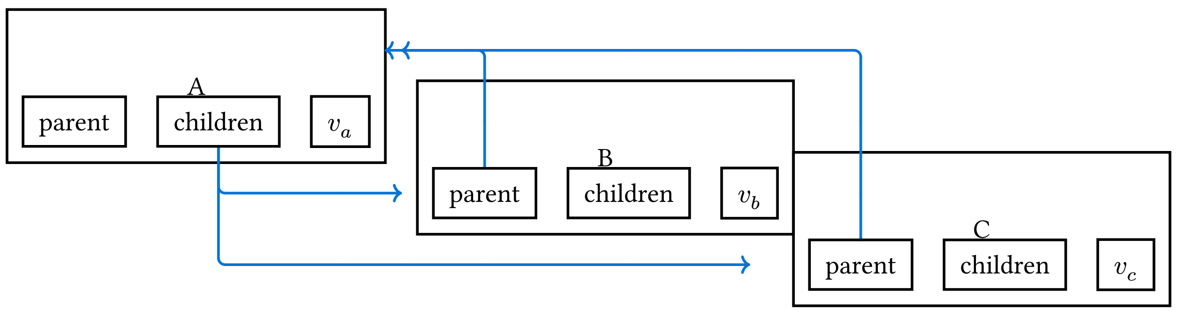
I took the convention that the first child is the **main** branch, and the other is the **secondary** branch (if it exists).

This pointer-based representation of the tree is ideal for the skeleton growing phase, because it is recursive in nature. For the latter stages of the pipeline, I decided to use an array based representation.

2.1.1 Pointer-based



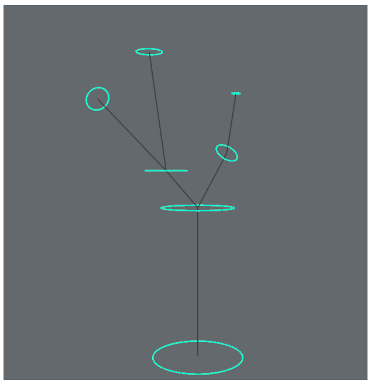
2.1.2 Array based



2.2 Step 1

As I said, I will detail the plant growing strategy later.

For all the tests and illustrations you will see, I used a manually crafted tree with 7 nodes:



2.3 Step 2: Particle flow and interpolation

code: `src/meshing/particles.rs`

The pseudo-code for this step is the following:

```
compute_paticles(node):
    if node is a leaf:
        create a set of particles for this leaf
    else:
        for each child:
            compute_particles(child)
        project particles onto parent
        do particle based collision detection
        return the set of particles
```

As you see, they are 2 main parts in this algorithm:

- projecting the particles
- doing the particle simulation

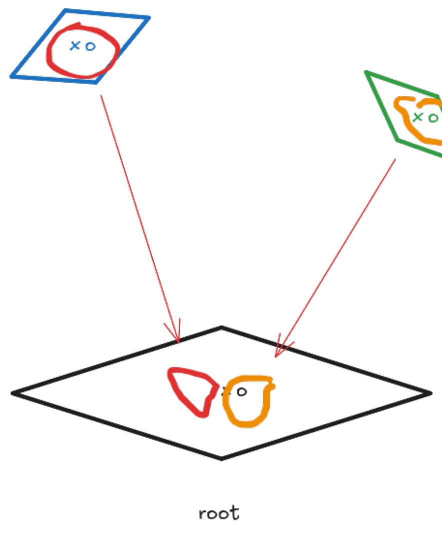
Note

The paper proposed to use arbitrary shaped for the section, I only implemented circles. It may improve slightly the diversity of possible trees, but most trees have circular branches.

2.3.1 Particle projection

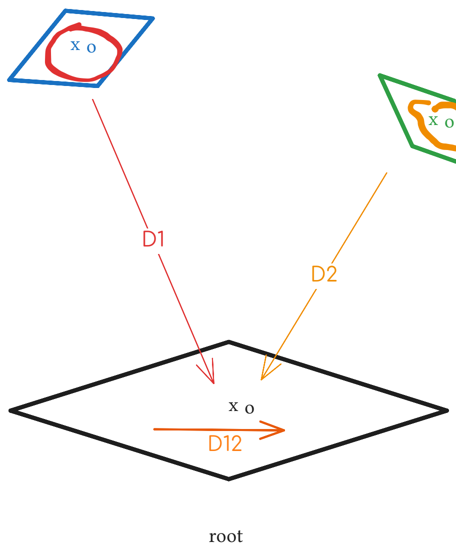
When there is only one child, the projection operation is pretty easy. For 2 children, it is more complicated. The paper did not describe how they defined the projection entirely, so I had to interpret what they did.

We want to compute the positions of the 2 clouds of points (orange and red) on the root parent plane:



The first step is to use the position of the parent and the 2 children to compute the important directions:

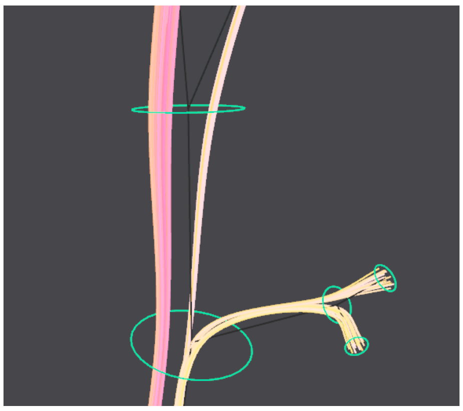
- from child1 to parent
- from child2 to parent
- from child1 to child2, in the parent plane



Then, the a point on the cloud of child1 will be projected on the plane parallel to D_1 and moved with an offset in the direction of -$D_{\{12\}}$. Similarly, a point in the cloud of child2 will be projected parralel to D_2 and offset in the direction D_{12}.

The offsets are calculated such that the two set of points are adjacent in the parent plane.

After projection and scaling, the particle trajectories look like:



In order to give the realistic result of trunks, we need to make the particles interact with each other.

2.3.2 Particle simulation

Again, the paper was not very precise regarding the type of particle simulation they used.

I tried to use an electrostatic-like repulsion force with an euler integratino scheme, with collisions on the border.

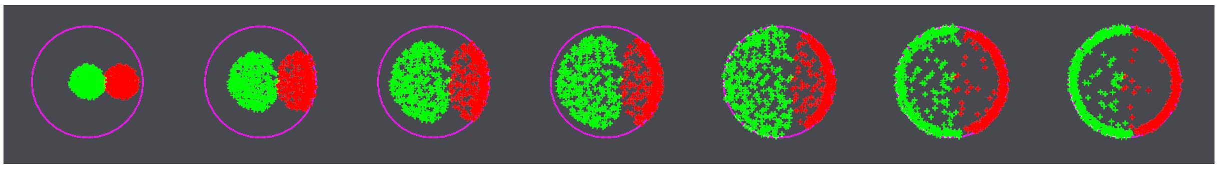
The force is defined as:

$$\overrightarrow{F_{ab}} = -\frac{k}{(AB)^2}\overrightarrow{u_{ab}}$$

My simulation has normalized parameters like:
- the repulsion force between particles
- the repulsion force with the wall
- the time step

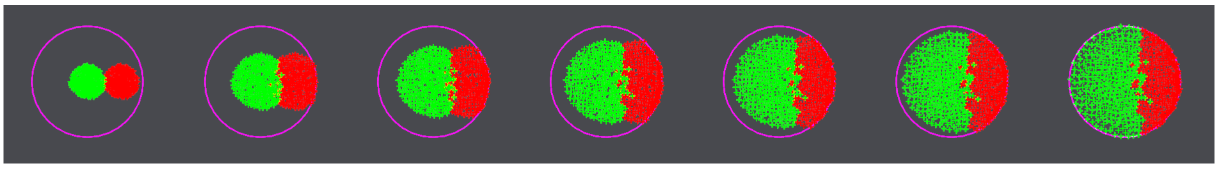
For numeric stability, I had to update the parameters depending on the radius and the number of particles. For example, the repulsion is proportionnal to the radius squared.

Here is the result on an example:



It works, but as you can see there is a problem: there are too much particles on the contours and not enough inside. This is due to the fact that the initial speed of the particles is too high.

To solve this problem, I introduced a maximum velocity. I also multiplied the repulsion constant by $\frac{1}{\sqrt{N}}$ where N is the number of particles. This way, the average speed is the same no matter the number of particles. After these adjustments, I got a more uniform particle density:



It made a huge difference for the phases after that:

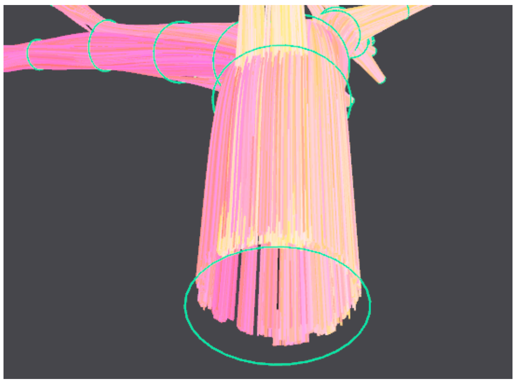


Figure 1: particles without max veolcity

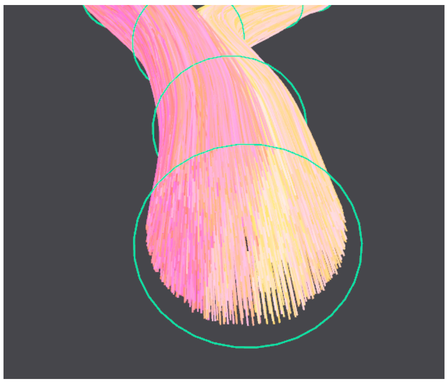


Figure 2: particles with max velocity

The particle simulation step is also the most resource-intensive part, because of the $O(n^2)$ complexity to compute the interactions between particles. (see Section 3.1). This is why I tried to optimize it, by constructing the neighbourgs less frequently.

Building the neighbourgs depending on some interaction radius every 5 steps worked quite well. To improve the performance further, I used a quadtree implementation for efficient retrieval. See Section 3.1 for more details.

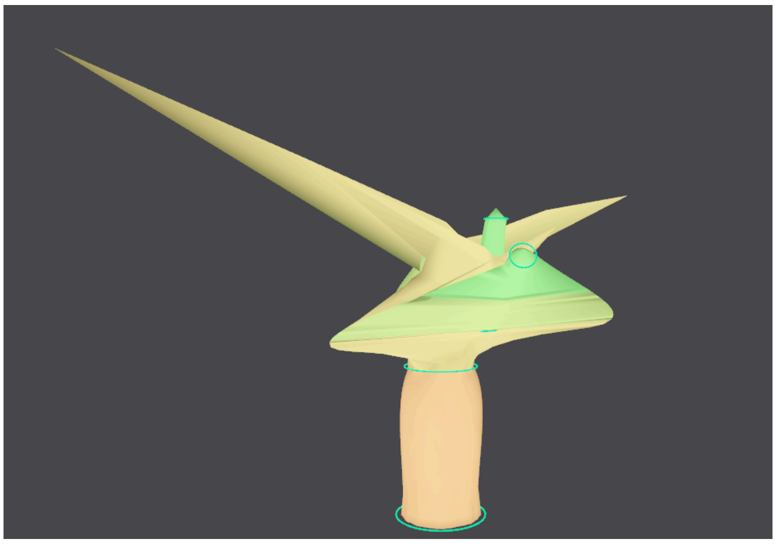


Figure 3: if dt is not small enough, it lead to numerical instability

2.3.3 Strands

Once we know the particle positions on each node, we can interpolate them using splines.

The paper proposed **B-splines**, but as this type of spline does not interploate the points,
I chose the Catmull-Rom spline.

A very frequent problem in this project is strands "mangling". After the projection operations and the simulation, strands can:

- go inside or outside the branch
- turn one around another

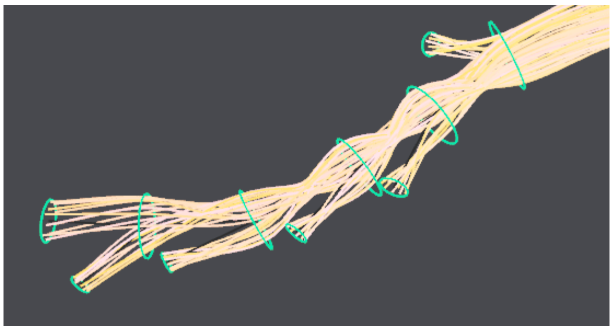


Figure 4: strands can zigzag

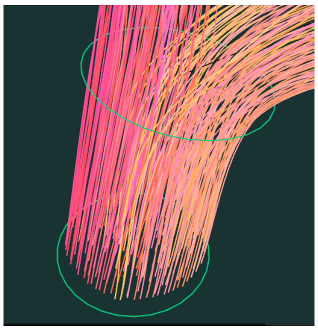


Figure 5: strands can mangle

This can cause a number of problems for the next steps, like contours calculation and triangulation. The paper proposed some techniques like swapping the points positions when it is minimizes the distance, but I did not implement it.

2.4 Step 3: Contours calculation

code: `src/meshing/mod.rs`

The pseudo-code for this part is the following:

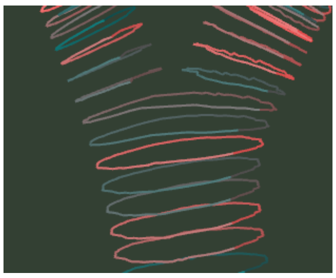
```
compute_contours(node):
    if node is a leaf:
        stop
    else if one children:
        for each position from parent to children:
            compute the positions of the point cloud
            select the boundary of the point cloud
        compute_contours(children)

    else if 2 children:
        compute the position when the branch splits
        for each position before the split:
            compute the positions of the point cloud
            select the boundary of the point cloud

        for each position between the split and first children:
            compute the positions of the point cloud
            select the boundary of the point cloud
        compute_contours(first children)

        for each position between the split and second children:
            compute the positions of the point cloud
            select the boundary of the point cloud
        compute_contours(second children)
```

Note that this algorithm is bottom-up, contrary to the previous one which is top-down.

2.4.1 Detecting if branch is spliting

To detect if the branch is splitting, I compute the minimum distance between the points in the first cloud and the points in the second cloud. If this distance is bigger than a treshold, that means the branch is splitting.[6]

Before the detected branch split, I use the union of the 2 points clouds to compute the contour. After the split, I compute the contours separately.



⚠ **Warning**

In some cases, the branch never splits. It is an edge case I did not imagine at first, but it can happen with lot's of branches. In this case, we stop at the shortest branch.

[6]I realized I can do it by dichotomy

2.4.2 Points selection

This was probably the hardest part of the entire project.

The paper proposed delaynay triangulation to compute the boundary of the points, but early on I thought this was not the best idea. Indeed:

- it is very costly, $O(n^2)$ in the worst case even more complex in 3d.
- it builds the entire triangulation of the section, but uses only the contour
- it does not give the contour directly, one must do a graph traversal afterwards.
- the paper removed the edges from the triangulation when they are longer than a treshold, but don't indicate how to set this treshold.

> **Note**
>
> To convert from 3d points (splines in space) to 2d, I considered using a minimization algorithm, to find the 2d plane that miminimize the error from the reality. After, I figured out I can simply project the points using the orientation vector of the nearest parent.

So my first intuition was to use a convex hull algorithm like Graham scan. It worked, but the issue is that it does not follow the contour when 2 branch merge (see below). I had to adapt the algorithm.

After a lot of work and debuging, I ended up with this algorithm:

```
select_contours $((P_i))$ :
    $O \leftarrow \frac{1}{n} \sum P_i$
    $i_0 \leftarrow \text{argmin}_i(e_y \cdot P_i)$
    $L \leftarrow [i_0]$
    $\sigma \leftarrow$ sort points depending on $P_i \mapsto P_{i_0} O P_i$
    for $i \in \sigma$ :
    | | add i to $L$
    | | remove the points in L depending on angle
    return L
```

To illustrate:

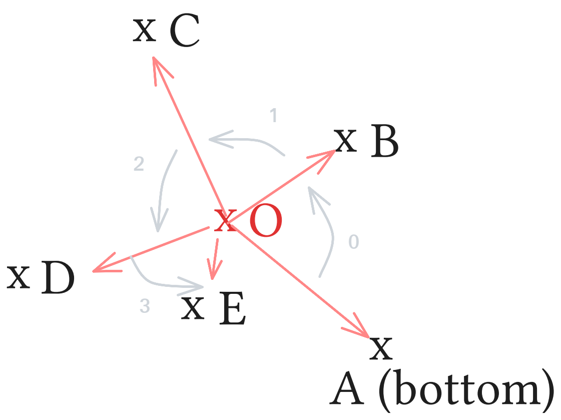


Figure 6: First, sort the points around the center

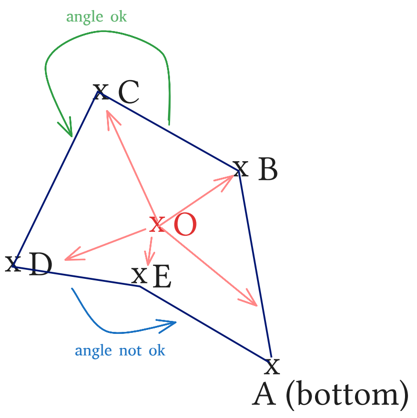
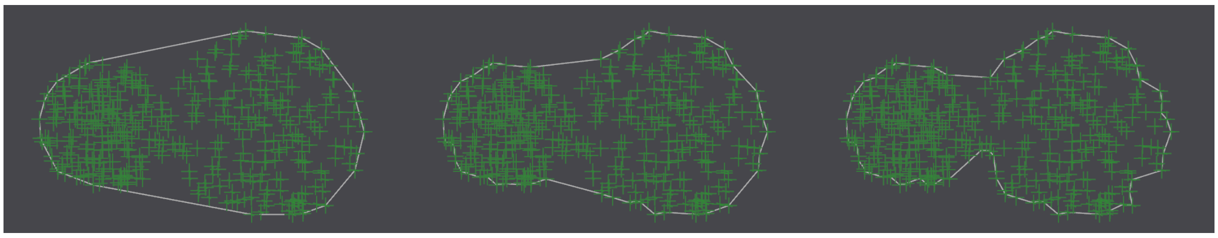


Figure 7: Each time one point is added, remove the previous one depending on angle

This algorithm is $O(n \log n)$ in the worst case, because of the sort.

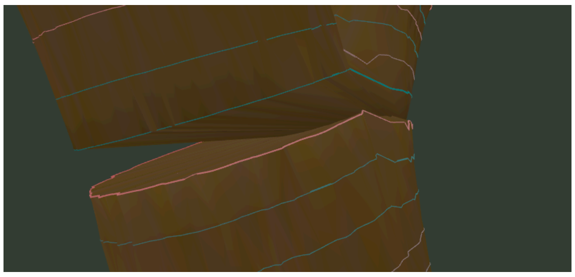
By varying the angle treshold, we can select different boundaries:



But setting the right treshold for the convex hull is not trivial.
- very high treshold: the contour does not follow the shape of the strands, it is especially problematic for meshing (see Section 2.5)
- very low treshold: too much noise and not enough regularity in the contour that is seleted.

⚠ **Warning**

When computing the mesh of a large number of points, I noticed some strange behaviours:



I was able to reproduce it and to understand where the issue was coming from:

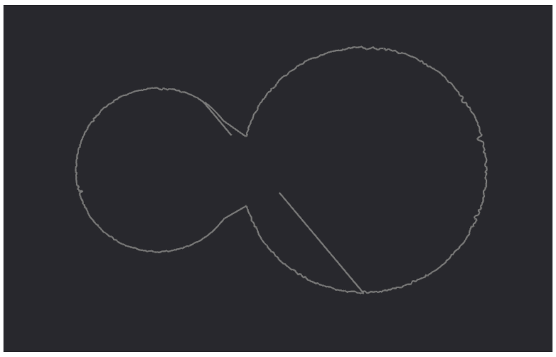


Figure 8: some points are added, but they are not on the boundary.

After checking my code again and again, I discovered that the issue comes from the way the angle is calculated in the math library:

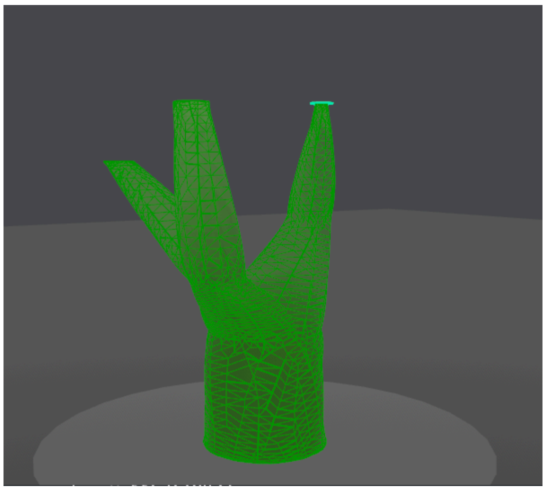
```rust
pub fn angle_to(self, rhs: Self) -> f32 {
    let angle = math::acos_approx(
        self.dot(rhs) / math::sqrt(self.length_squared() * rhs.length_squared())
    );
    angle * math::signum(self.perp_dot(rhs))
}
```

The floating points error caused the sign to flip when the points are almost colinear, and thus to be treated as almost 360deg.

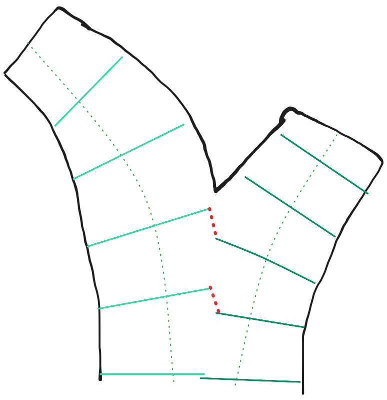
2.4.3 Parametrization

The paper did not mention how the step between 2 consecutive sections is calculated.

At the start, I used a fixed number of steps for each branch. This resulted in a non-uniform mesh:

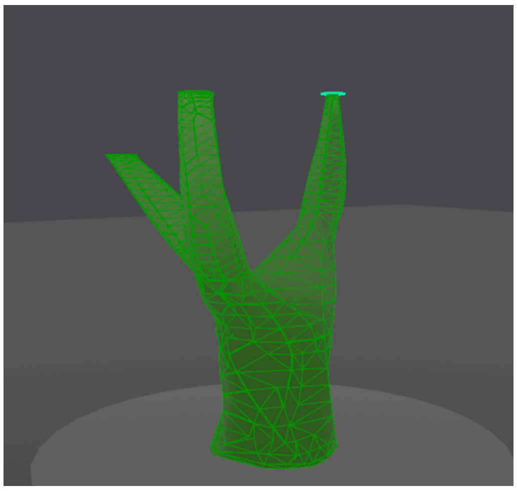


Other issue: When the two children branches are not the same length, it creates a gap between the 2 sides:



TO solve these issues, I changed the approach and used a step dz computed according to the radius of the branch. To get the spline parameter, I use $t = k\frac{dz}{L}$ with L the total branch length.

It was a lot better:



Unfrtunately, it did not solve all the problems. In particular, there was an irregularity between the last contour of the branch and the first of the next branch.

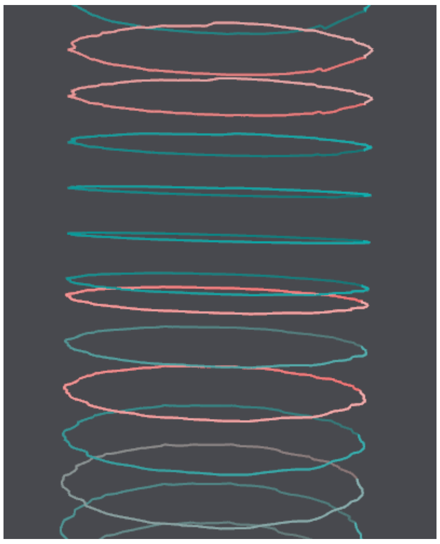
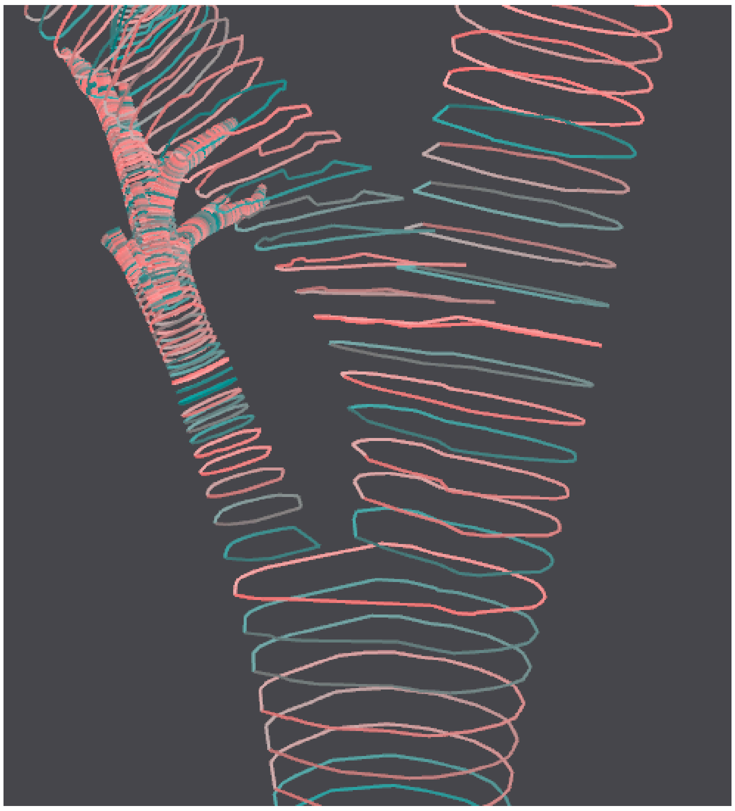


Figure 9: the distance between contours is too small

The solution was to pass the offset from the end of the branch in the recursive call to the function.

After all of this, I got a good parametrization:



Note

I could have used a parametrization of the spline directly, but it would have been unpractical to compute the branch joins.

2.5 Step 4: Meshing and branch fusion

The final step is to comupte the mesh (vertices, colors, normals and triangles) of the entire tree

The basic idea is to take 2 contours, and join them together using triangles on the lateral surface. Again, it was not entirely clear from the paper so I had to adapt.

2.5.1 Joining contours

I looked at the problem as a minimazation problem.

Given two lists of points P_i and Q_i, you want to interlace them in the way that minimizes the sum of the perimeters of the triangles.

The perimeter of the triangle at one moment of the algotihm is one of 2 possibilities:

- $P_iP_{i+1} + P_{i+1}Q_j + Q_jP_i$
- $P_iQ_j + Q_jQ_{j+1} + Q_{j+1}P_i$

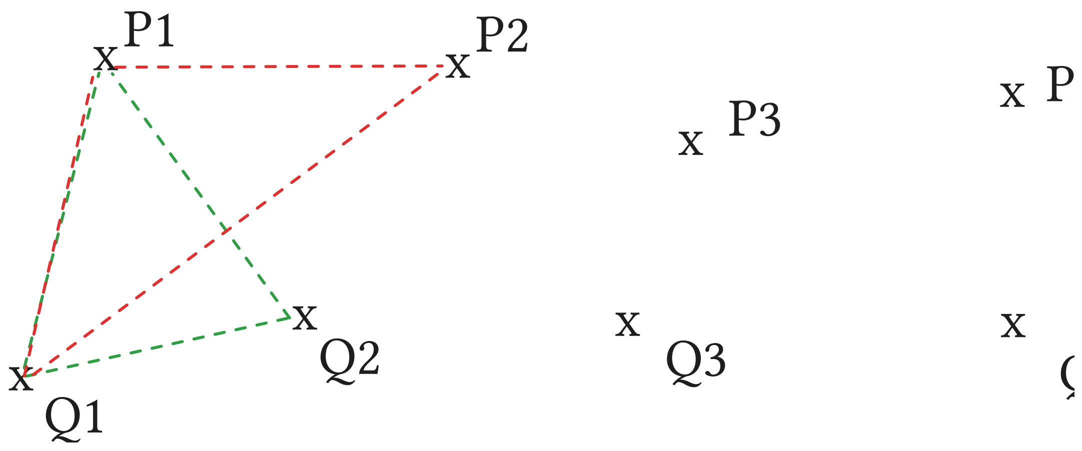


Figure 10: the 2 possible choices of triangles

Since the terms P_iP_{i+1} and Q_jQ_{j+1} will be added either way, we just have to minimize the sum of P_iQ_j.

This leads to the following greedy algorithm:

```
join_contours(P, Q):
    (i, j) = (0, 0)
    if d(P[i], Q[j+1]) < d(P[i+1], Q[j]):
        add the triangle P[i],Q[j],Q[j+1]
        i += 1
    else:
        add the triangle P[i],P[i+1],Q[j]
        j += 1
```

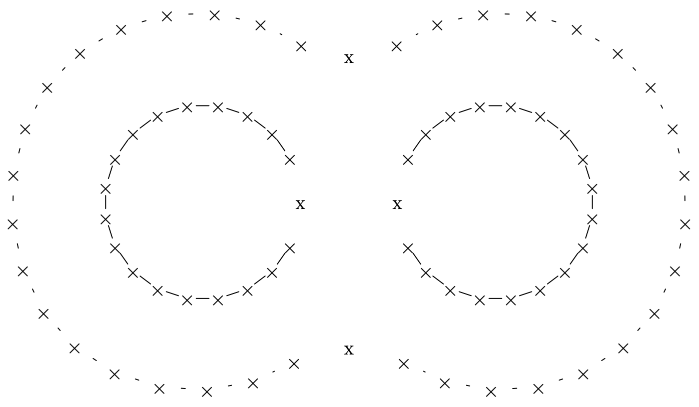
Note

At the end, P_n and P_0, Q_n and Q_0 must be linked together to close the lateral surface.

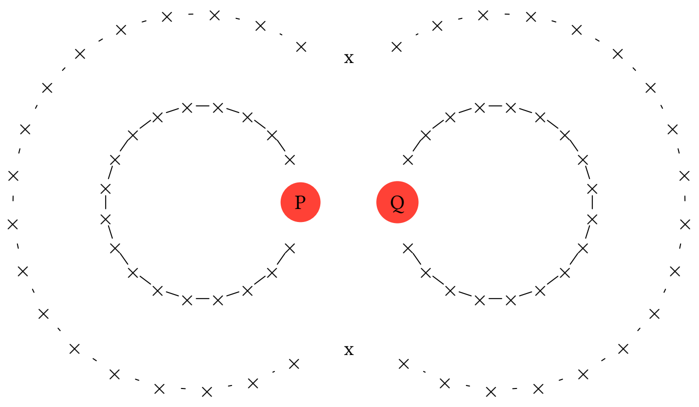
2.5.2 Branch fusion

There is a very particular case of meshing when 2 branch split: there is one contour above and 2 contours over.

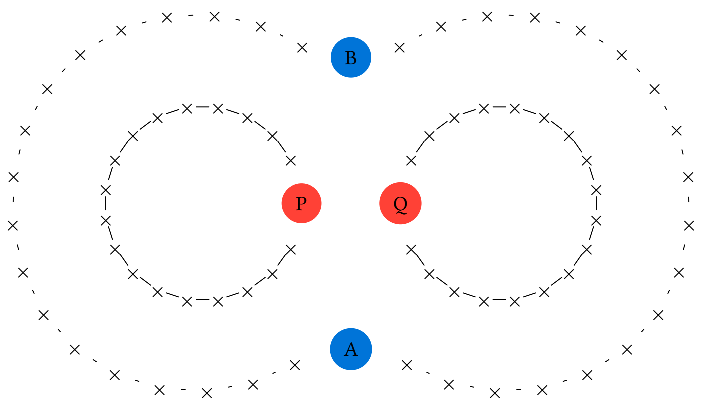
The situation can be represented by:



In this situation, we want to mesh everything such that the result has no hole. In order to create the mesh, I start by finding the 2 nearest points from the 2 branches:



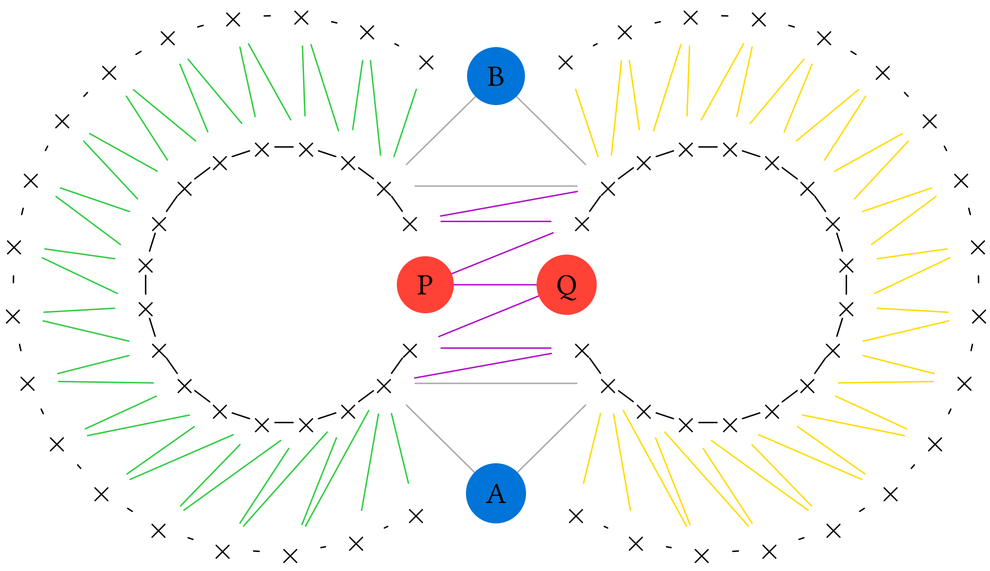
And then I find the points on the above contour that are the nearest from them:



Note

To be sure the 2 selected points A and B are not on the same side, I compute the determinant $\det(PQ, QA)$ and det (PQ, QB) . They must be opposite signs.

Once I found all these special points, I can use the previous algorithm to create 3 strips. There are 2 special triangles, represented in gray on the figure.

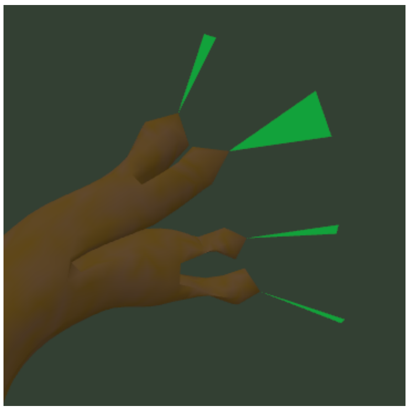


2.5.3 Normals

The first version of my algorithm computed the normals section by section, computing the bisector for each segment and interpolating. The result was very unstable and did not work when the branch changed direction abruptly. To solve this issue I computed the normals when my mesh was completely generated, using a pre-existing algorithm.

2.5.4 Adding leaves

For the visual aspect, I added leaves by creating random triangles at the end of each trunk.



2.6 Skeleton generation

For the skeleton generation, I used a simpler version than what has done the paper. I adapted the ideas from another paper that used L-systems to generate the skeleton.

I create the tree from the root, and add 1 or 2 children with random rotations and random branch lengths. Some details:

- the probablity of having 2 childrens increase with depth
- I make sure that the angle between 2 children is not too small, because in reality 2 children branches interact one with another
- the radiuses of the 2 children don't have the same probability distribution.
- I stop when the branch radius is smaller than a treshold.

As I said, I chose to focus on the mesh generation and performance, and not the skeleton generation part.

3 Analysis

3.1 Performance

With the quadtree implementation, my algorithm took around 15s to generate 200_000 particles and 16_000 strands. This is slower than the paper. For reference:[7]

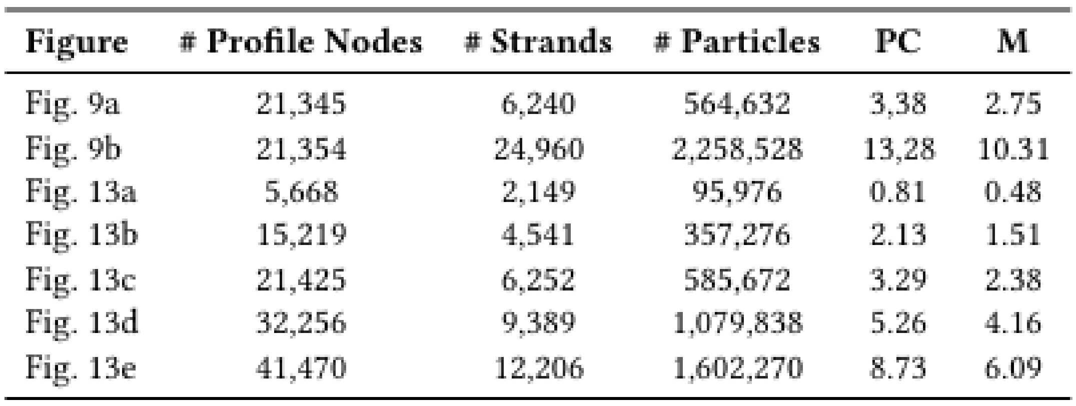


Figure	# Profile Nodes	# Strands	# Particles	PC	M
Fig. 9a	21,345	6,240	564,632	3,38	2.75
Fig. 9b	21,354	24,960	2,258,528	13,28	10.31
Fig. 13a	5,668	2,149	95,976	0.81	0.48
Fig. 13b	15,219	4,541	357,276	2.13	1.51
Fig. 13c	21,425	6,252	585,672	3.29	2.38
Fig. 13d	32,256	9,389	1,079,838	5.26	4.16
Fig. 13e	41,470	12,206	1,602,270	8.73	6.09

I think my particle simulation code is slower, but my mesh generation code is faster. Also, I don't know what machine they used.

Iw would be very valuable to benchmark each phase to know what is the bottleneck, but I did not have time to do it.

3.2 Limitations and improvements

My implementation suffer from some edge cases, and some of them can make the app panic.

- when there are not enough points to merge
- when two children branches do not split (solved)
- when the branches split just before or just after a node in the tree

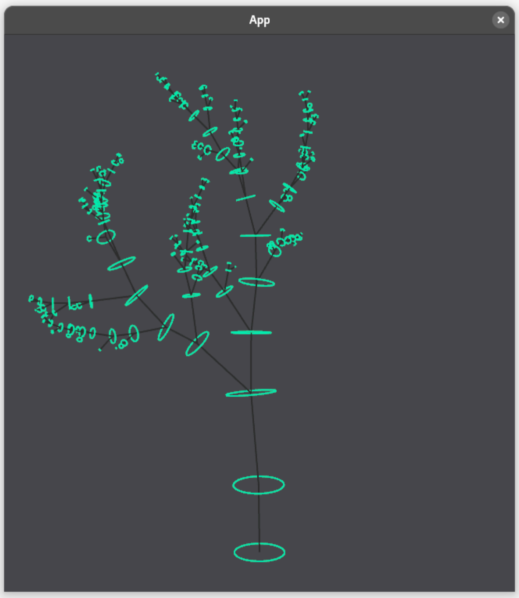
The most useful improvements in the current state of the project are:

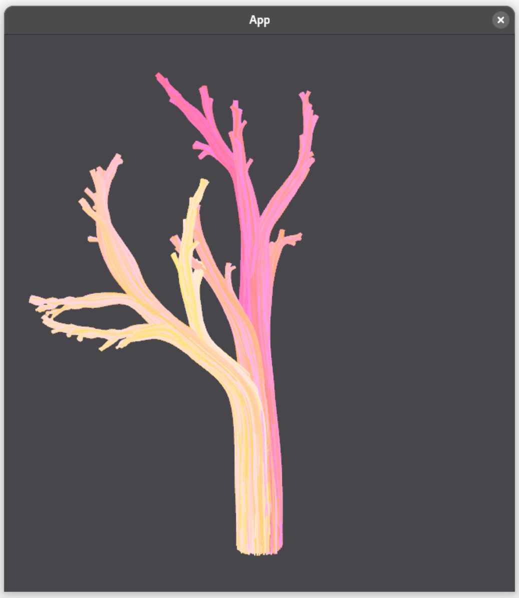
- using a biology-based model to generate the skeleton
- being able to specify custom branch profiles, not just circles.
- benchmarking each phase to know which phase should be optimized
- testing other data-structures to improve the performance of the simulation.
- using paralelization

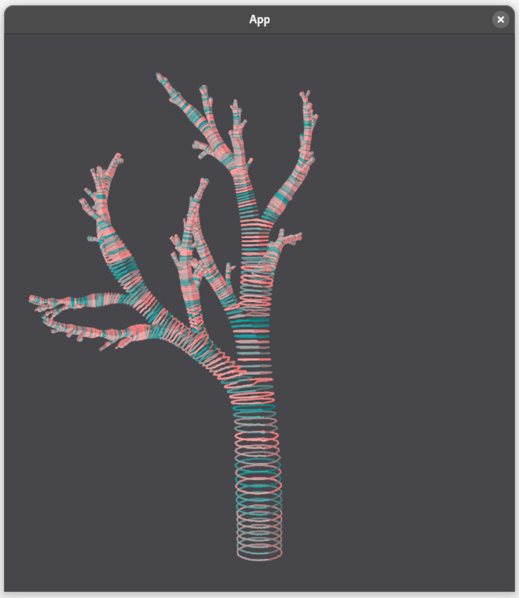
[7]PC is time for particle simulation in seconds, M is time for mesh generation in second

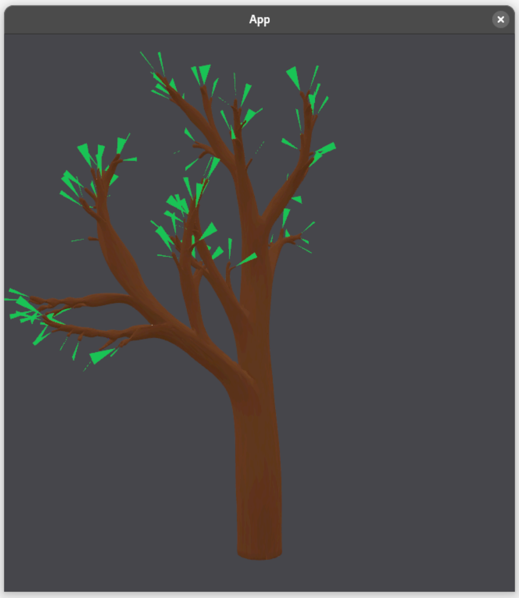
4 Gallery

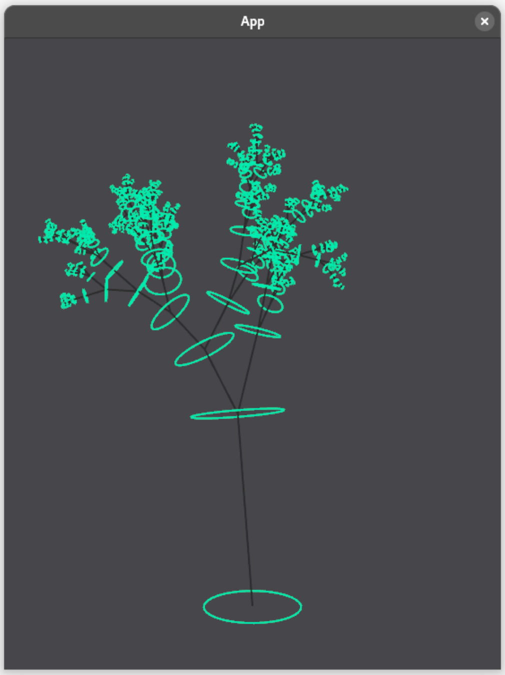
I spent some time adding visualization code for each phase of the generation. They can be enabled or disabled in the web interface also.

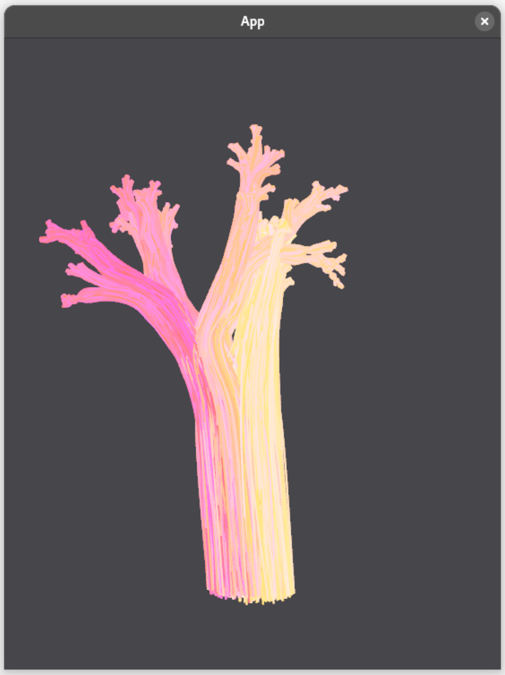


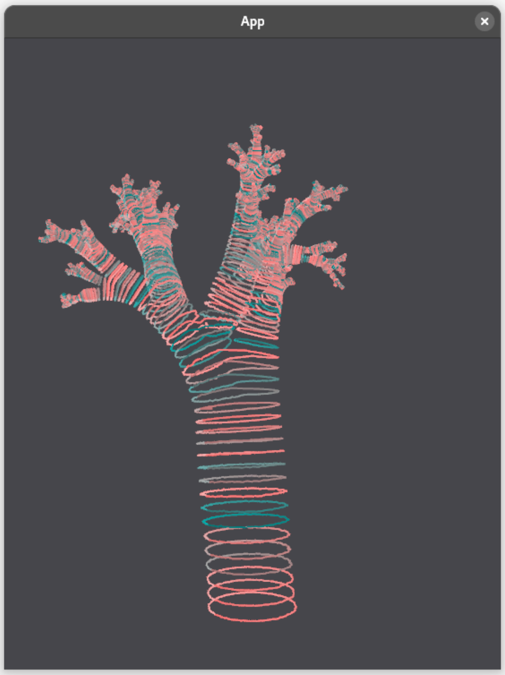


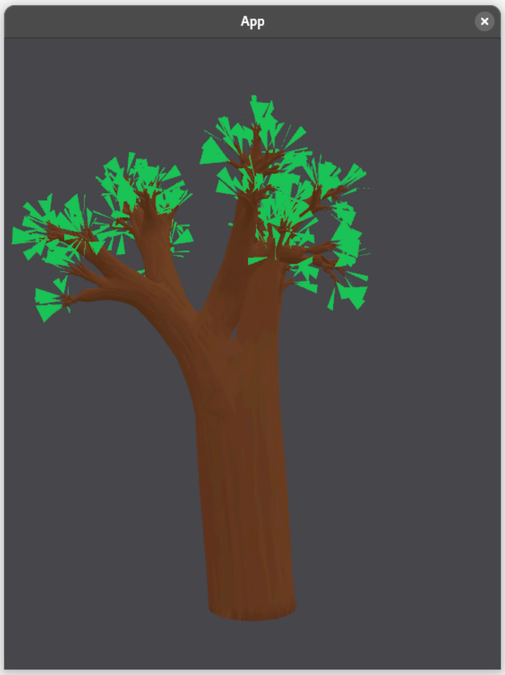


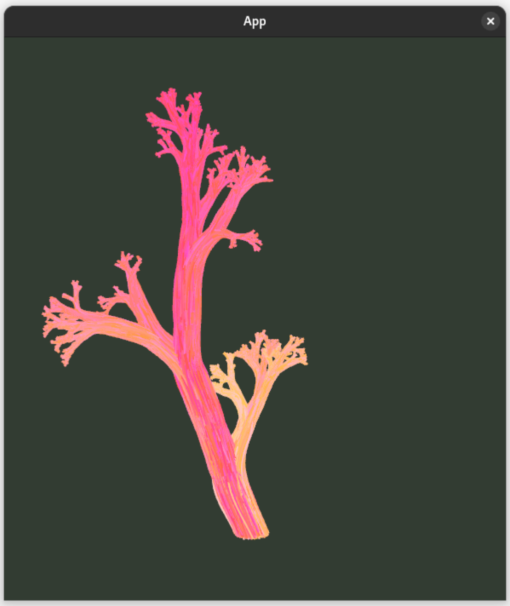
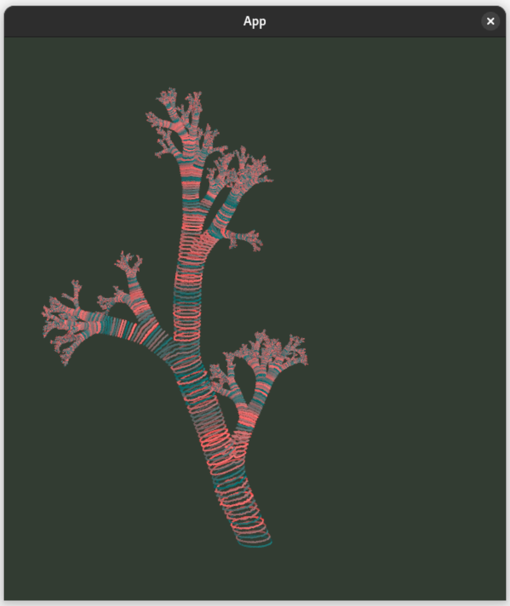



App

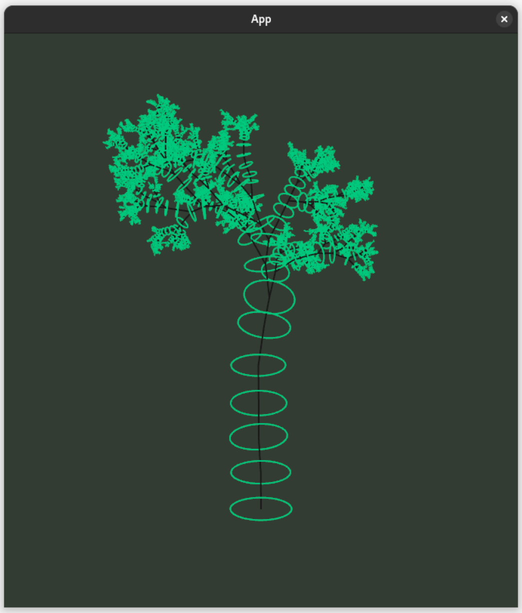

App

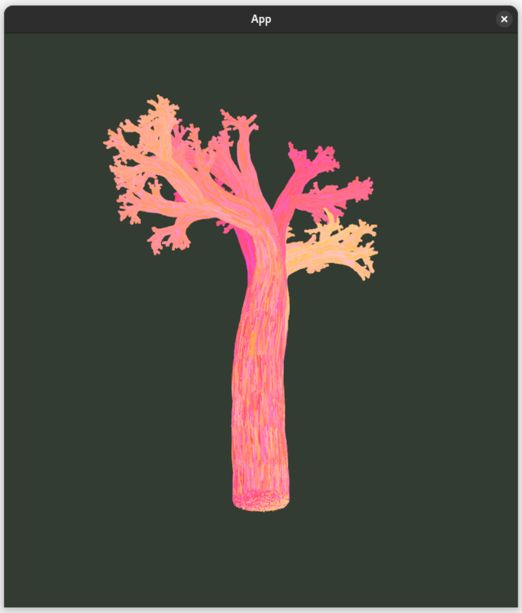

App

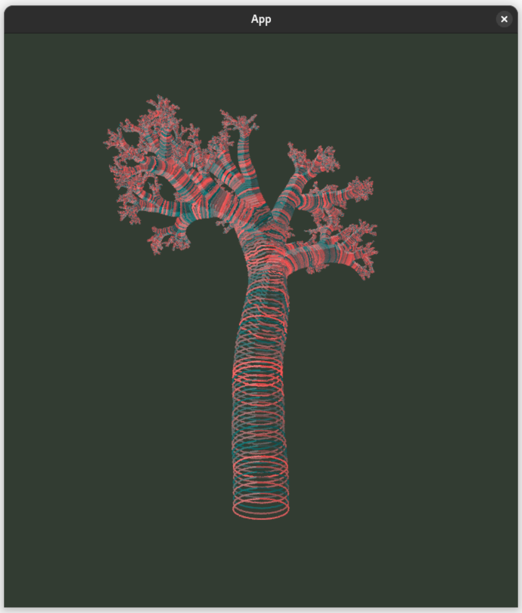

App

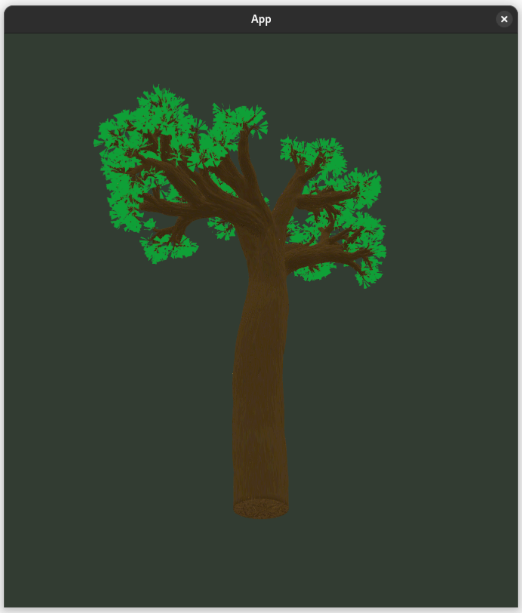
App

App

App

App


App


App


App


App