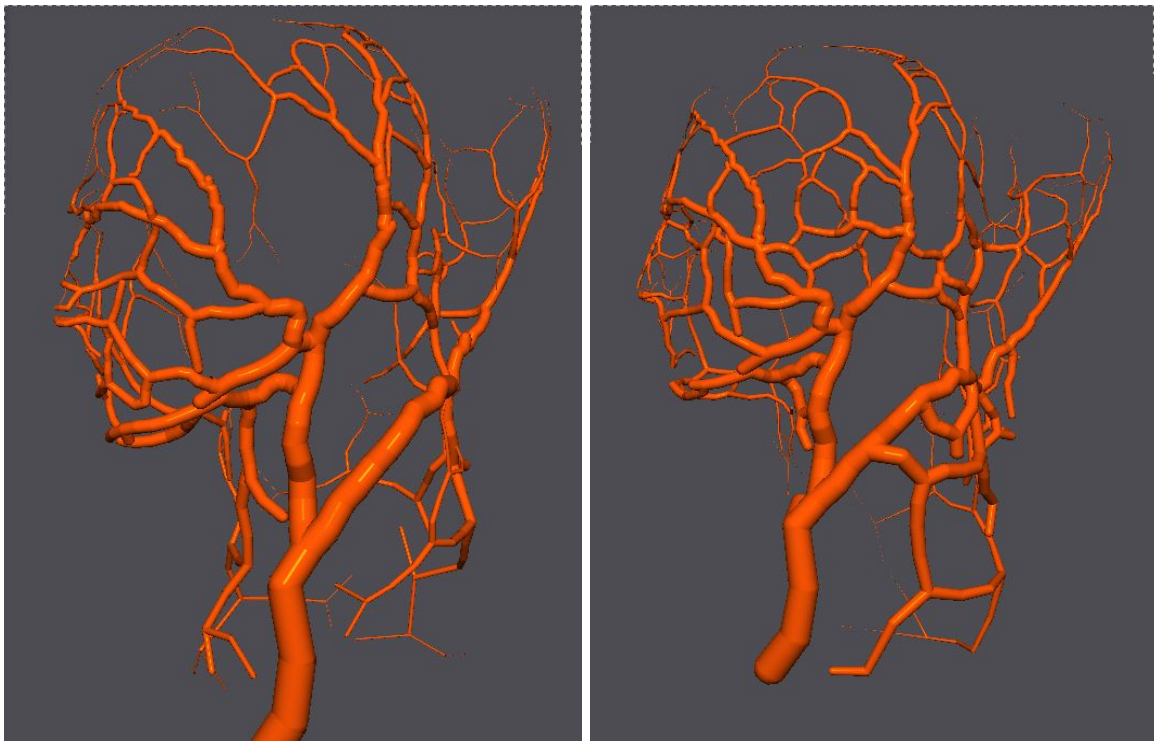
Digital 3D Geometry Processing

Course project : Geralt of Treevia

Justinien Bouron, Nicolas Casademont & Nicolas Roussel



Introduction

During the semester we learned state of the art algorithms and techniques from 3D Geometry that allowed us to manipulate and transform 3D meshes, using different metrics and operators.

This project aims to show what can be achieved with such tools and operators. Our goal was to produce a tree-like mesh in which the viewer can recognize the original mesh from a certain point of view.

This paper will explain each step of our implementations, the intermediate steps as well as the difficulties encountered during this project.

Preliminary steps

Inspirations

The first inspiration we had came from a korean artist named Sun-Hyuk Kim[1] who created interesting pieces of art in which roots of plants form diverse figures. You can find some of his art on the following webpage: https://www.saatchiart.com/sunhyuk

In our case making roots would not make much sense as roots grow downwards and the lamp points upward. So we changed the idea a bit and decided to make branches growing upwards from the base of the lamp. From an algorithmic point of view, this is essentially the same except that the direction of growth is inverted.

Researches

Most papers on procedural tree generation rely on the same core steps in their algorithm, where for a given branch they reach a statement similar to : “Randomly decide if you split your branch, randomly pick a direction to continue growing toward and randomly decide on the length of the growth”. Where the randomness is of course biased depending on the tree variety.

External tools

For any manual transformation of the mesh (vertex paint + mount, see later) Blender was the tool of choice.

Library-wise we decided to use GLM instead of Eigen for matrix transformations as we were already familiar with this library.

Finally Cura was used to check the printability of the final mesh.

Preprocessing

Remeshing

The base model of Geralt contains more than 760,000 vertices and 2,200,000 edges which is far too much for our limited computation power. On top of that the average edge length was very small (≈0.1) and the meshing was close to an average one, which is not optimal considering the bottom of the resulting mesh will be sparse.

Thus, remeshing is a necessary step.

We use the height-based remeshing algorithm we implemented during the labs, but slightly modified. Contrary to the lab, the result of the remeshing is so that the lower half of the mesh is sparse whereas the upper half is dense. For comparison this is equivalent to say that, on a real tree, branches that are close to the ground (trunk) are generally longer than the ones at the top, and there are few of them.

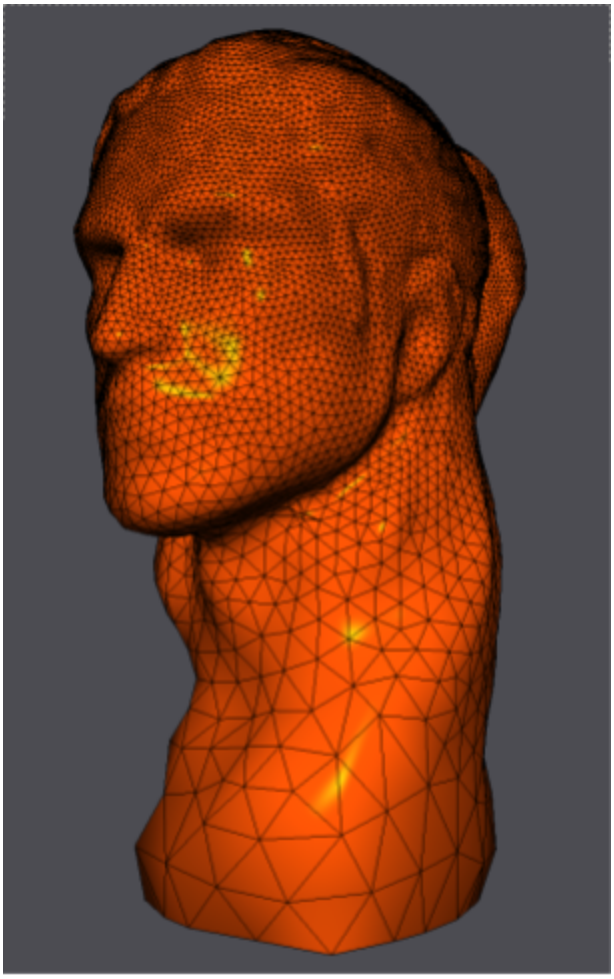


Figure 1 : Result of applying the inverted height-based remeshing operator on the original mesh.

The result generated by this modified version of the remeshing algorithm can be seen on figure 1.

This remeshing step outputs a mesh containing only around 16,000 vertices and 50’000 edges while keeping the main features of the face such as the jaw-line, the nose and the eyebrows.

Implementation steps

Our implementation schedule went through 5 phases :

1. 3D wireframe:
   Find a way to transform the mesh into it's "3D wireframe", where every edge is replaced by a cylinder and every vertex by a sphere. This is the most important step as it is at the core of our tree branches transformation.

2. Tree branches algorithm:
   Establishment of the algorithm which simulates tree branches growing. It is then applied to the mesh by considering edges as potential branch paths.

3. Lamp mount:
   Defining a strategy to mount our mesh onto the lamp without considerably affecting the mesh.

4. Make it pretty:
   Simulate the result of the 3 previous steps in a Blender/Maya scene to get a feel for the lighting produced by the lamp. From there, tweak the algorithm parameters to get a more aesthetically pleasing result if judged necessary.

5. Printability :
   Make sure that the model is printable using Cura. If the model is not printable add elements to make it more 'robust' and easier to print.

We will now go into details over several of our implementation steps.

1. 3D Wireframe

This constitutes the backbones of the tree branches algorithm. The idea is pretty straight forward, every vertex gets replaced by a sphere, and every edge by a cylinder. We do not have to worry about connecting them together as Cura takes care of this.

In the code this is implemented by the WireframeProcessing class.

This is where the first difficulty arose : the Surface Mesh library does not provide a simple way to add vertices and edges, it is not possible to insert an edge between two vertices. However we can add faces, which in turn take care of the edges and half-edges for us.

From this, the algorithm to insert a given mesh (sphere or cylinder) at a given position and with a given rotation and with a given scale is the following :

```
Function insert(target_mesh, mesh_to_insert, position, rotation, scale) :
        For each face F of mesh_to_insert:
                For each vertex V of F :
                        V' = Apply scale, rotation and translation to V
                        Add vertex V' to target_mesh
                End for
                Create face in target_mesh with the vertices added in the for loop above
        End for
End function
```

This algorithm is obviously simplified here for the sake of clarity. In practice we need to avoid adding some vertices multiple times. This algorithm is implemented in the wireframe_processing.cpp file, in the method insert_mesh.

Using the above algorithm, we can easily create a new mesh containing the 3D wireframe of the original mesh:

```
Function create_wireframe(original_mesh) :
        W = new empty mesh
        For each vertex V of original_mesh :
                insert(W, sphere, V.pos, V.rot, V.scale)
        End for
        For each edge E = (V1, V2) of original_mesh :
                P = center of the edge = (V1 - V2) / 2
                S = (cylinder_diameter, E.length, cylinder_diameter)
                R = Rotation to align the cylinder with the edge direction
                Insert (W, cylinder, P, R, S)
        Return W
End function
```

Results can be seen on the following figures :

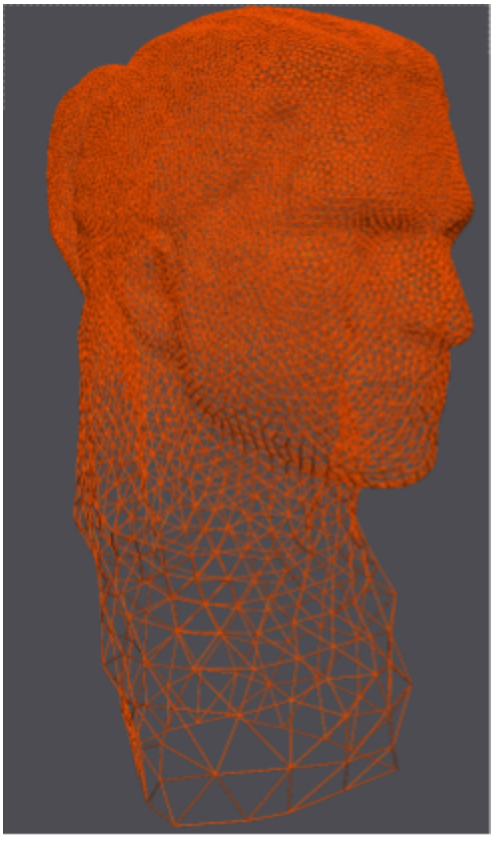


Figure 2 : Result of the wireframe on the remeshed mesh from figure 1

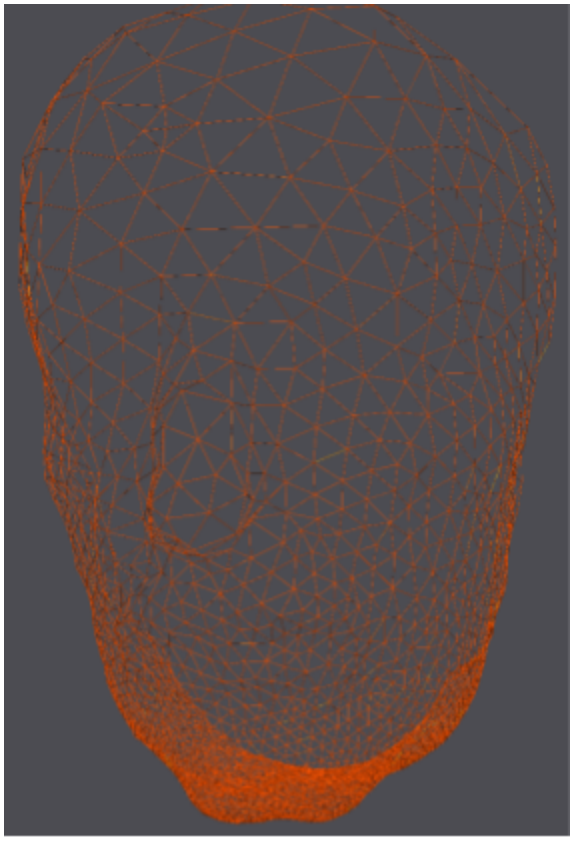


Figure 3 : Result of wireframe on Max-Plank.

2. Tree branches algorithm

Our tree branch algorithm contains two steps :

1. The user defines the trunk(s) of the tree using vertex painting (in red, green or blue) over the original mesh using Blender.
2. The algorithm from there generates branches coming out of the trunk(s) in a procedural manner.

We will now cover these two steps in details.

Defining the trunks

The very first implementation of our algorithm was completely procedural. There were two problems with this implementation. First the result looked chaotic, as branches would grow in any direction. The second problem was that the algorithm could potentially 'miss' the features of the face such as the jawline, nose, etc... These features are very important as they allow the viewer to recognize the original mesh, in our case, Geralt's face.

Figure 4 : Result of a procedural only implementation.

Figure 4 shows the result of such an implementation.

To overcome this we had to find a way to 'help' the algorithm keep these features. This help was naturally a user input.

The user is telling the algorithm where the important vertices of the model should be, so that he can insist on features by inserting them in the trunk(s). This is done using vertex

painting, the trunk(s) are defined by paths of vertices painted in red, green, or blue over the mesh. The maximum trunk number is set to three.

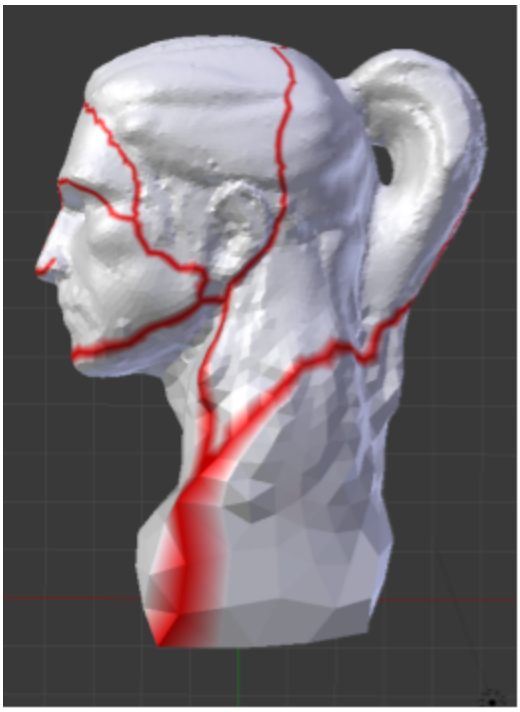


Figure 5 : The root used to get our final result, show in red vertex paint.

Figure 5 shows the trunk we used to get our final result, we prefered using only one trunk because we think that the result is much cleaner than with multiple trunks. Besides, the fact that the viewer has to look at the right angle to recognize the mesh is a very interesting feature to us.

With this implementation, the algorithm is only growing the branches coming out of the trunk(s) while preserving the important features of the face. The result can be seen on figure 6.

Note that it is not mandatory to have multiple trunks, but at least one is required.

The next part will go into the detail of the branch growing part of our implementation.

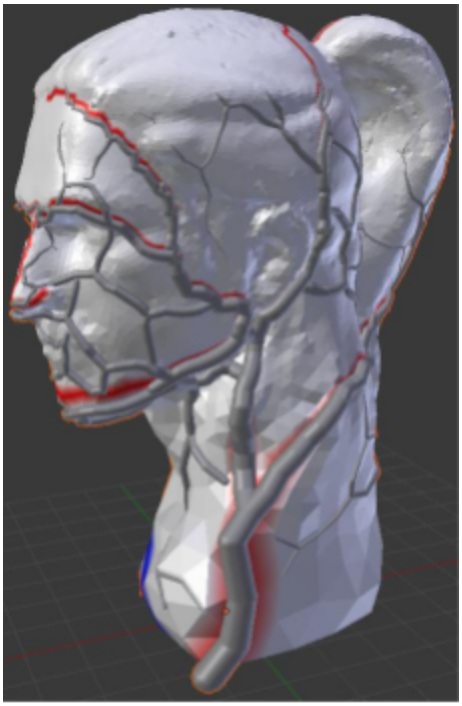


Figure 6 : The original mesh with a red trunk painted, overlayed by the result of the tree algorithm.

Growing the branches

This is the core part of our project. To keep things simple, we will describe the big idea of each step of the algorithm. The entire pseudocode is given in the annexe.

<u>The detailed steps :</u>

As input the algorithm receives the mesh in which the trunk(s) are painted. The algorithm is queue based (BFS) and runs until the queue is empty. Initially only the lowest vertex of the trunk(s) is(are)

pushed into the queue. At each iteration the algorithm executes the steps given below.

1. Pop vertex out of queue
2. Retrieve neighbors of the current vertex, and store them in a local neighbors set
3. If current vertex is in a trunk, remove any neighbors that are also in the same trunk from the neighbors set and add them to the queue
4. Remove any neighbor, from the set, which does not fulfill the condition on the branch angle described later on
5. Let N be the number of neighbours pushed to the queue in step 3. We have 3 cases:
    a. N = 0 : In this case we may split into two branches.
        i. If a split occurs choose the 2 neighbors that maximize the angle between them, and push them to the queue
        ii. Otherwise choose the neighbor that minimize the change of direction of the current branch
    b. N = 1 : In this case we may add a branch (split), or leave the tree unchanged
        i. If a split occurs we choose the neighbor that is as far as possible from the trunk (angle is maximized between them), and add it to the queue
        ii. Otherwise things are unchanged, no additional neighbors are pushed to the queue, the branch continues through the trunk
    c. N >= 2 : In this case splits are forbidden, no additional neighbors are pushed to the queue
6. For each neighbors pushed to the queue, compute their properties (scale, ...) and add it to the tree wireframe.
7. Go to the next iteration

The full pseudo code for the complete algorithm can be found in the annexe.

Let us go into details for some of the steps.

New properties :

Here is the list of tree wireframe related vertex properties :

- Absolute length of type float : This is the total length from a vertex to the lowest vertex of the trunk it is connected to. This property is mainly used as a termination condition (branches cannot grow indefinitely) and for the scaling of the wireframe : the larger this value is, the smaller the sphere becomes.
- Relative length of type float : This is the total length from a vertex to the last split of the same branch. This property is used on splitting condition.
- Trunk of type bool : This property is used to indicate whether a vertex is marked as part of a trunk.
- Trunk id of type int : As the model can contain up to three trunks we need a way to differentiate those. By default the red trunk as ID = 0, green ID = 1 and blue ID = 2. Note that vertices that are not marked as part of a trunk hold -1 as trunk index.

Neighbor filtering :

In step 4, we filter the neighbors on a certain angle condition. When selecting the next direction for a branch (or a split) we take one that does not 'go back', in other words we always take the next neighbor so that the angle between the new direction and the last one is larger than 90° and lower than 270° (dot product strictly positive). This is depicted in Fig. 7.

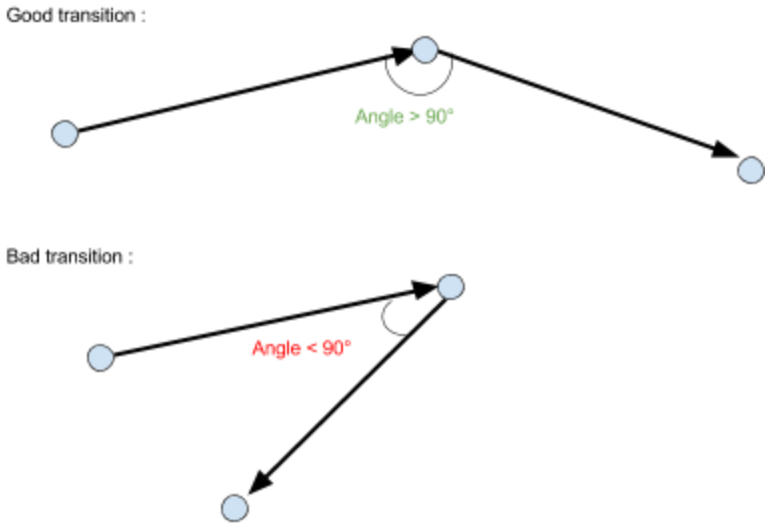


Figure 7: Example of a good and a bad transition, determined by the angle between the previous and the new branch.

Splitting :

Splitting at a given vertex is decided randomly. There are two conditions that must hold in order to split :

1. The number of branches already coming out this vertex must be strictly lower than three. Because of this we have cases depending on the number of trunk neighbors as explained in step 5.
2. The vertex must have a relative length over a certain threshold. The reason behind this is to avoid having splits at every vertices.

If those two conditions holds, then we randomly decide if the split occurs or not.

The random variable that dictates a split is nonlinear and depends on the absolute length of the vertex. It works as the following :

```
Function split(v):
        P = uniform random sample ∈ (0, 1)
        Return (max_branch_length * P < v.absolute_length)
End function
```

The intuition is that splits should occur more often at the top of the tree than near the bottom.

If a split occurs we pick the two neighbors that have the maximum angle between them. Firstly, because it avoids having branches that are really close to each others. Secondly, because the tree will cover more surface over the mesh. This is depicted in Fig.8.

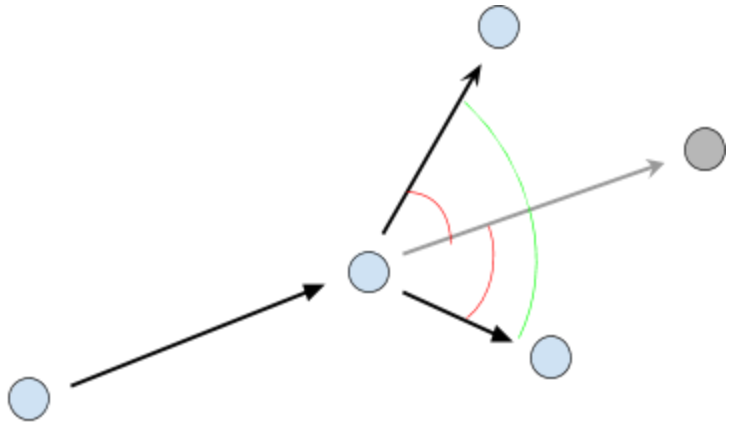


Figure 8: Example of the decision process on a split into two branches. The neighbors having the maximal angle between them are kept, while the others are discarded.

If no split occurs, we take the neighbor that minimizes the change of direction in the branch, as below :

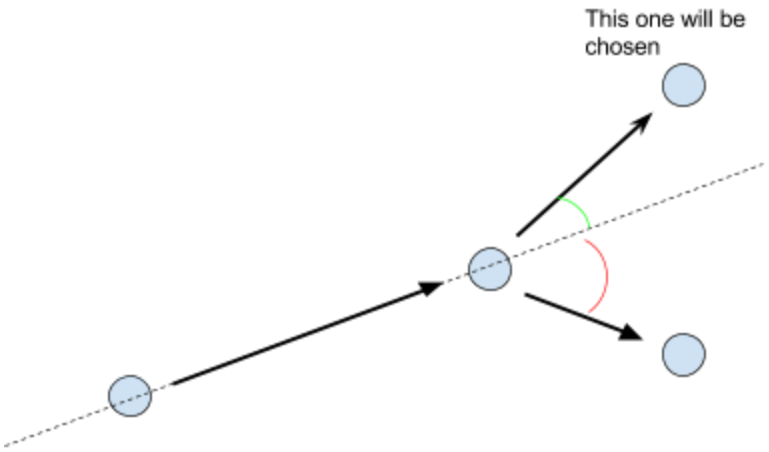


Figure 9 : Example of the decision process when choosing the next neighbor without split.

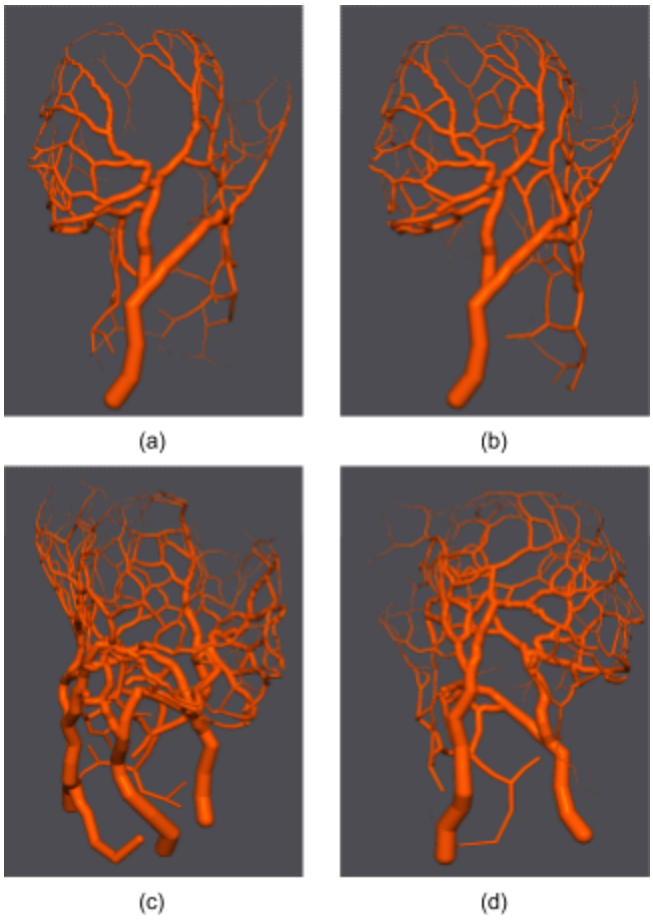
Results

WIth this algorithm we achieve interesting results.

On the right hand side all images share the same (default) parameters for the branches generation.

A and b have both one trunk, and are two possible outcomes.

C shows an example with three trunks, D with two.



3. Lamp Mount

The tool of choice for making the lamp mount was Blender. We create a small plate which simply holds the trunk. Additionally, we removed all branches which went below this plate.

We rendered the scene with a light source in the middle of the mesh to get a feeling on how it would interact with light once printed, the results can be seen in figures 10 and 11. Of course these results are simulations and must be taken with a grain of salt as, in real life, light may not interact like pictured in those figures.

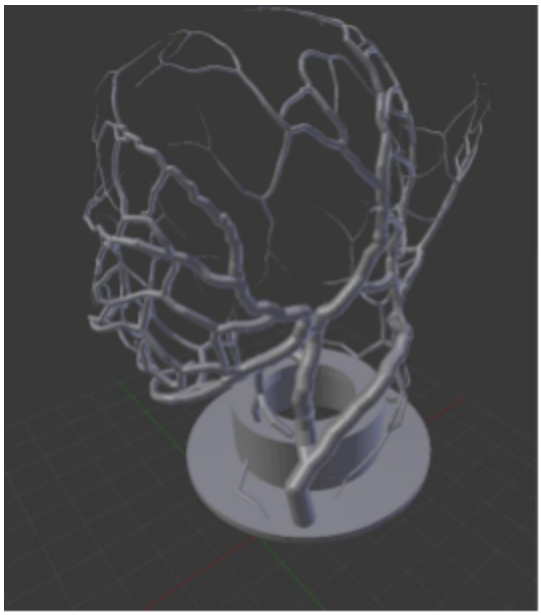


Figure 10 : The mount we created which holds our mesh.

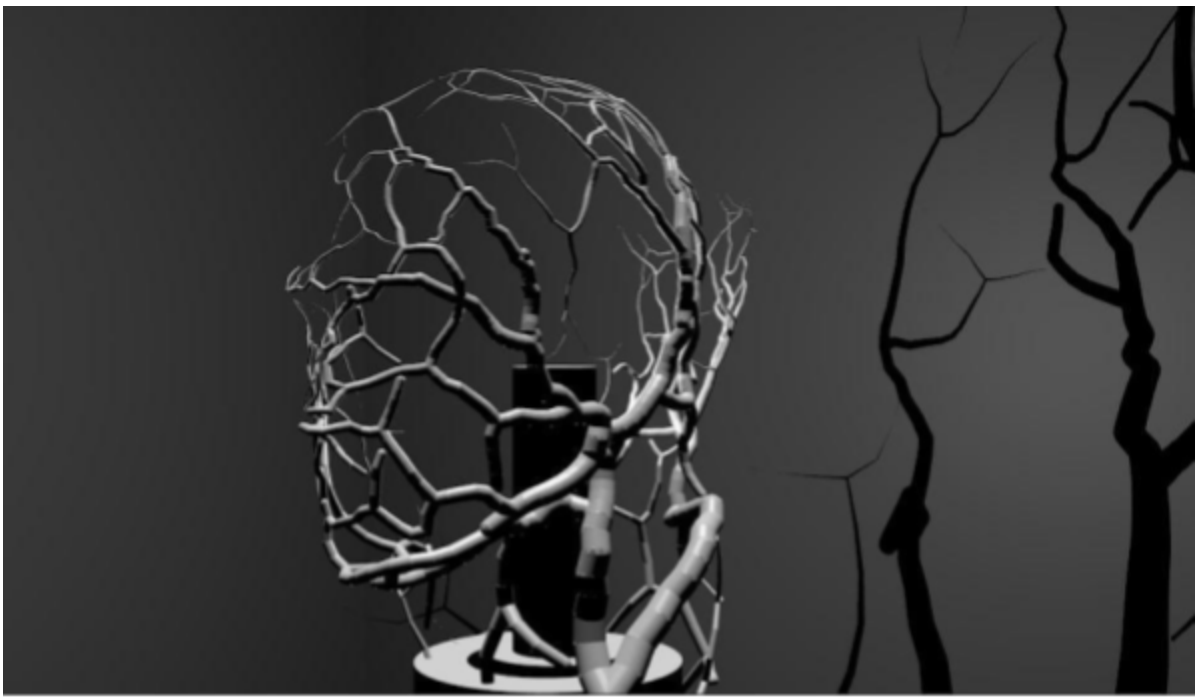


Figure 11: Simulation of the lighting effects when a light source is placed in the middle of the mesh.

4. Printability

One of our main concern was the printability of the final mesh, as it contains a lot of small branches near the top. Fortunately by making the mesh bigger we are able to print almost the entire mesh. Some branches at the top may not be printable (Fig. 12), but as long as we are able to recognize Geralt (Fig. 13), it is okay.

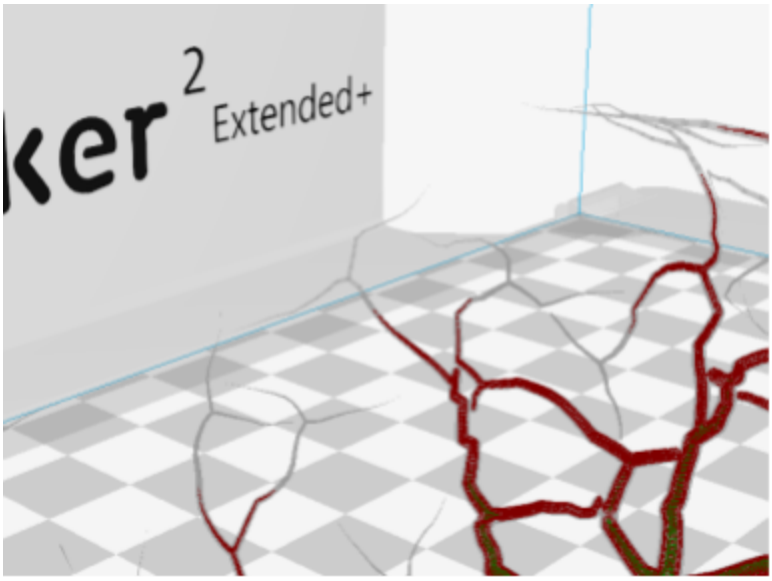


Figure 12: Non printability at the tip of the branches.

Figure 13 : The whole mesh on Cura, using layer view mode

Although the printing process might be hard due to the complexity of the mesh, and the additional supports required, it is still possible as seen on the Cura software. (Fig. 14).

Figure 14 : Supports needed for the 3D printing process.

The dimensions of our model is around 9 x 10 x 6 centimeters, and the estimated printing time is around 11 hours.

Conclusion

This project was really interesting as we made use at the same time of the course algorithms (height remeshing for example) but also created our own for the tree growth which after some trial and error gave really pretty results that effectively looked like trees.

For such a task, procedural generation is a strength but also a weakness. While we can obtain good looking results from the random generation, some others yield lackluster meshes. A future improvement could be to avoid such meshes, by defining a set of rules that must held to be considered good looking.

As future improvement we thought of adding some realistic textures to the mesh and then extrude them with blender so that they are visible on the 3D printed model. Sadly we didn't have time to finish this part and some complications could appear while trying to print the model with holes in thin branches.

Given the way our code functions, it is possible to plug in any other mesh. The only step which would absolutely be needed is the trunk drawing. This makes our overall procedure very generic, and it would be interesting to see the behaviour on other meshes.

The final results of our project can be viewed in the video.avi file we provided with the submission. The video shows the GUI and a small simulation of the lighting.

Instructions

Building the code

These instructions concern GNU/Linux or MacOS users.

Extract the zip file in any directory and execute the following commands :

```
$ cd [project folder]
$ mkdir build && cd build
$ cmake .. && make && ./tree_mesh/tree_mesh
```

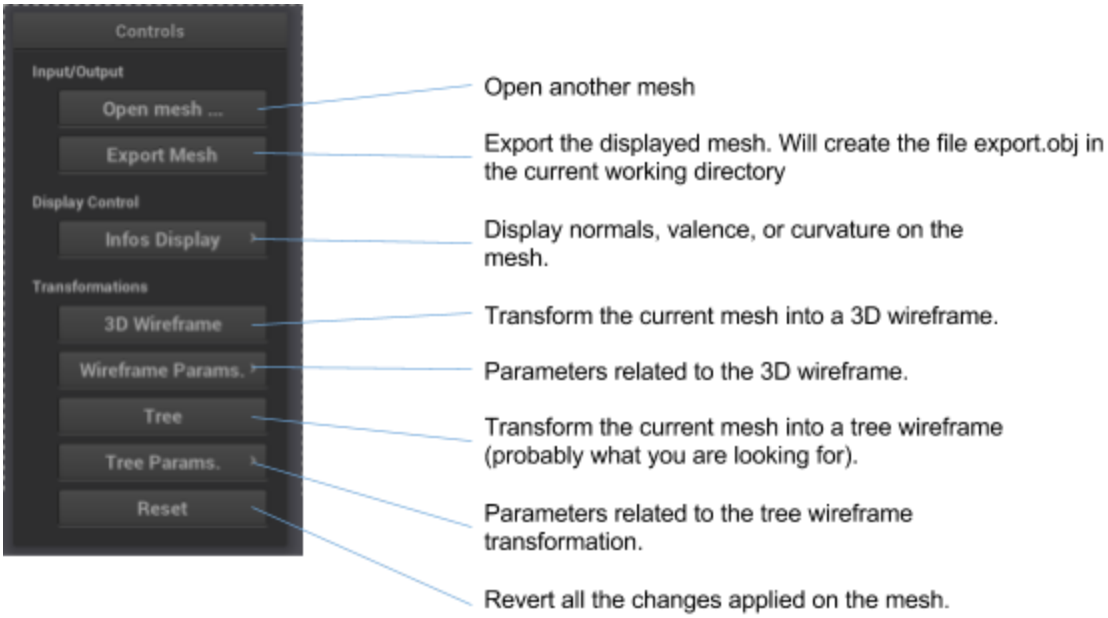
You will need the following libraries to successfully compile the project :

- libsurface_mesh
- openGL
- Eigen

GLM is provided with the source code.

GUI

You can see the instructions on how to use the GUI in the image below :



The GUI was made in a way that allows you to play with all the parameters of the 3D Wireframe and the Tree algorithms, so that you can see how the algorithms behave when changing them.

Here are all the tweakable parameters provided with a short explanation :

- 3D Wireframe
    - Spheres diameter : Diameter of the spheres that will replace vertices in the wireframe
    - Cylinder diameter : Diameter of the cylinders that will replaces edges in the wireframe
- Tree Transformation
    - Sphere base diameter : Diameter of the spheres of the wireframe as seen at the bottom of the trunk
    - Cylinder base diameter : Diameter of the cylinders of the wireframe as seen at the bottom of the trunk
    - Branch max length : Maximum (absolute) length that a branch can be
    - Trunk scale multiplier : Scale multiplier only applied on the vertices marked as trunk
    - Min. dot between branches : This is the condition on the angle presented in the neighbor filtering part above.
    - Min. relative length before split : The minimum relative length needed before a branch is allowed to split.
    - Red, Green, and Blue trunk checkboxes : Allows the user to select which trunk(s) to use during the transformation. Make sure that at least one of them is checked.

Beware that your model needs a trunk painted using vertex paint on it (it is already the case for the default one), otherwise the tree algorithm won't work (3D Wireframe is fine though).

Annexe

Pseudo code for the full tree transformation :

Legend : Trunk growth Neighbor filtering Split Recursion

```
Function create_tree(mesh) :
    S = lowest vertex of each trunk
    Q = empty queue
    For each s in S :
        push (s, s) into the queue
    End for
    create_tree_inner(Q)
End function

Function create_tree_inner(Q) :
    While Q is not empty :
        current_vertex, last_vertex = Q.pop()
        if current_vertex.abs_length > max_length :
            goto next iteration
        end if
        next = Empty set
        N = neighbors(current_vertex)
        For each n of N such that n is in a trunk:
            if current_vertex is in a trunk, and the trunk is the same trunk as n :
                add n to next
            else
                remove n from N
            end if
        end for
        For each n from N:
            current_branch_direction = current_vertex.pos - last_vertex.pos
            new_branch_direction = n.pos - current_vertex.pos
            if not dot(current_branch_direction, new_branch_direction) >
min_dot_between_branches):
                remove n from N
            end if
        end for

        if N.size == 0:
            go to next iteration.
        end if

        if next.size < 2:
```

```
                split_count = 1 - next.size
                if current_vertex.rel_length >= min_rel_len_before_split and
split(current_vertex) gives true:
                        split_count += 1
                end if
                if next.size <= split_count:
                        split_count = min(split_count, N.size)
                        if split_count = 1:
                                if next.size = 0:
                                        add the neighbor that would minimize the change
of direction of the branch if followed
                                else :
                                        take the branch that would have the greatest angle
compared to the neighbor already present in next and add it to next
                                endif
                        else :
                                take the two neighbors that have the maximum angle
between them and add them to next.
                        endif
                endif
        endif

        split_occured = next.size > 1

        For each vertex n of next:
                e = edge between current_vertex and n
                n.abs_length = current_vertex.abs_length + e.length
                if split_occured :
                        n.rel_length = e.length
                else
                        n.rel_length = current_vertex.rel_length + e.length
                endif
                add e to the wireframe with a scale going from the size of the current
vertex to the size of n along the axis of the cylinder.
                add current_vertex to the wireframe with scale sphere_diameter *
scale_factor(current_vertex.abs_length)
                push (current_vertex, n) to the queue
        end for
    End While
End function
```

References

[1] Sun-Hyuk Kim https://www.saatchiart.com/sunhyuk