SEMESTER PROJECT

Minor in Computational Neuroscience

LiveMesh

A TOOL FOR REAL-TIME RENDERING OF NEURONAL CELLS FROM MORPHOLOGIES

Student
Michaël DEFFERRARD

Professor
Felix SCHÜRMANN

Supervisors
Jafet VILLAFRANCA DIAZ
Stefan EILEMANN

Proposed by Blue Brain Project, EPFL

January 6, 2015

Contents

1 Introduction

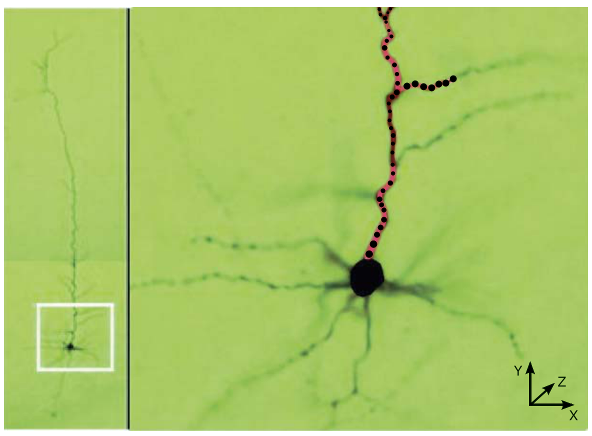
This report presents the work accomplished during an 8 ECTS semester project at EPFL Blue Brain Project (BBP). It is part of the minor in Computational Neuroscience curriculum. The project goal was to prove the feasibility of GPU-based tessellation to generate neuron membrane mesh representations from parametric descriptions of neurons. Several approaches have been considered and a prototype software has been developed.

We will present a simple and general algorithm which generates a neuronal membrane surface mesh from a set of sampled morphological points. This method produces a smooth, continuous and high-fidelity representation of neuron morphologies that can be used for scientific visualization. While existing techniques [Las+12], [Bri+13] target offline mesh generation, the proposed method was designed from the ground for real-time generation on highly parallel GPU hardware. Online rendering saves disk space as it is no more required to store mesh data. The reduced size of the parametric description, compared to mesh data, additionally saves memory bandwidth. Finally, on-the-fly geometry generation opens the door to dynamic level of detail, a technique which helps to reduce the triangle counts while maintaining the quality by adapting the mesh granularity.

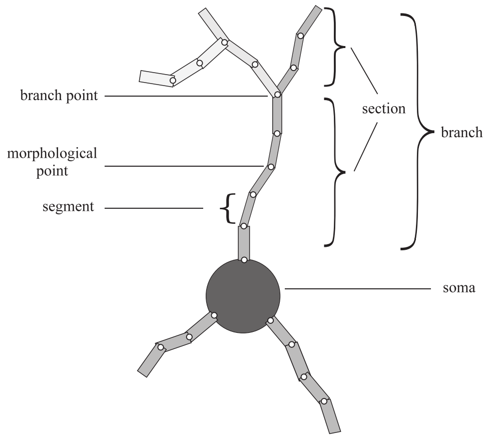
This project involved the study of state-of-the-art neuron membrane mesh generation, which includes the actual tool of the visualization team, NeuMesh. It was also necessary to review geometry processing techniques as well as the new functionalities provided by the OpenGL 4 pipeline in terms of vertex processing. Third-party libraries needed to be reviewed to make an informed choice. The API of the chosen ones (Qt, GLEW, vmmlib, Boost), especially vmmlib, was studied. Several toy applications have been incrementally developed to gain experience with libraries and GPU-based tessellation. The use of the visualization team existing software and infrastructure was required: BBPSDK for morphologies readout, Buildyard (based around cmake) as the build infrastructure, git and gerrit for code management and review, the vizcluster (vglconnect, vglrun, salloc, scontrol, ssh, Kerberos) for testing. The workings of Buildyard needed to be comprehended to stringify shaders as part of the build process.

2 Background

The raw data used by the proposed mesh generation process is obtained from biological experiments. A neuron is made visible under bright field microscopy by the diffusion of a chemical dye throughout it (see Fig. 1a). The neuron shape can then be digitally traced by means of a software application such as Neurolucida [GG90]. This process is carried out manually by a human operator which traces the fibers with mouse clicks specifying morphological points. We refer to this type of neuron reconstruction data as the *morphological point representation* or *morphological skeleton*.



(a)



(b)

Figure 1: Neuron reconstruction. (a) Bright field microscopy view of a neuron. (b) Schematic view of the neuronal anatomy. Both images from [Las+12].

A neuron is a cell composed by a soma, an axon and some dendrites. For its three-dimensional representation the soma is described as a sphere with a given radius. The axon and dendrites are described by a set of morphological points, comprised of a position in space and a radius. The conical frustum formed by two consecutive morphological points is called a segment. The set of segments between two branching points is called a section. A set of sections who represent a dendrite or an axon is called a branch (see Fig. 1b). Tracing the structures in order results in a tree which root node is the soma. Child sections are sections that branch off of the parent sections at the branch point. Each section has a parent. In the case of first-order branches, the parent is the soma.

3 Proposed method

We will first briefly present the new OpenGL 4 graphic pipeline that made our implementation possible. We will then present our algorithms to generate branches and somas surface meshes.

3.1 OpenGL 4 hardware tessellation

The idea of tessellation is to create smooth surfaces from a few control points, i.e. to represent higher level geometry than triangle meshes.

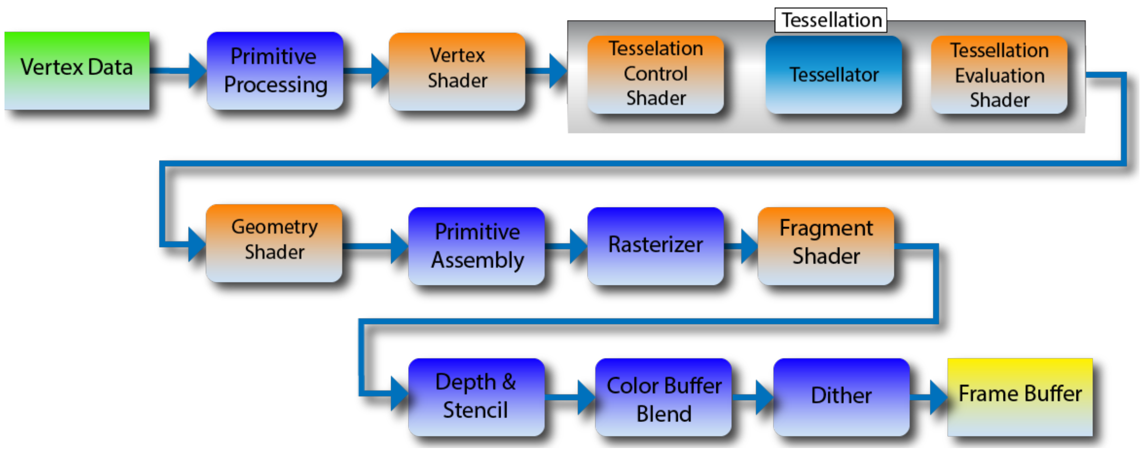


Figure 2: OpenGL 4 pipeline adds three stages for hardware tessellation[1].

Our method makes use of the three new stages introduced by OpenGL 4 capable hardware: the Tessellation Control Shader (TCS), the Tessellator and the Tessellation Evaluation Shader (TES) (see Fig. 2). The two shaders are programmable while the Tessellator is a configurable fixed functionality. These stages aim at vertex processing: they are used to generate vertices from a parametric description. The TCS takes as input what is called an abstract patch, composed by a configurable number of vertices. It instructs the Tessellator about the tessellation level to apply, i.e. the number of vertices to generate. The Tessellator generates the vertices which are arranged in a rectangular[2] grid. The TES is then responsible to displace the vertices to form the parametric surface.

An abstract patch can be anything. A common choice to render smooth surfaces is to define them as meshes of bicubic Bézier patches. The geometry of a single

[1]Source: http://3dgep.com/introduction-to-opengl-and-glsl/
[2]Triangle or isoline tessellation are also possible.
[3]Source: http://prideout.net/blog/?p=49

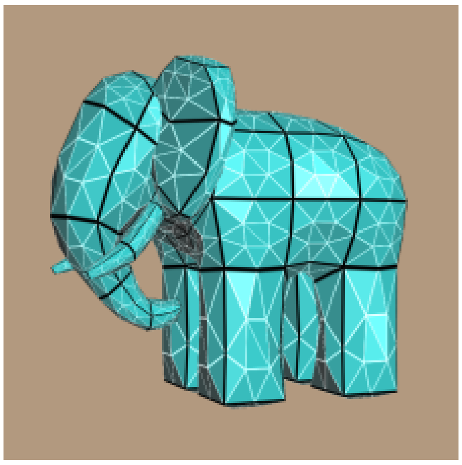


Figure 3: Catmull's Gumbo model defined by a mesh of bicubic Bézier patches[3].

bicubic patch is completely defined by a set of 16 control points which means that the TCS takes 16 vertices as inputs. The apparent smoothness of the surface then depends on the granularity of the generated vertices, i.e. the number of vertices per patch. Vertices are generated by the Tessellator while their number is controlled by the TCS. The TES then evaluates the position of the generated vertices using the mathematical definition of a Bézier surface (a sum of Bernstein polynomials) and the control points. Figure 3 shows an example of 12 vertices (14 triangles, depicted in white) generated per patch (depicted in black). Smoother or coarser approximations of the parametric surface can be generated by playing with the number of generated vertices. As the tessellation level can be provided by the TCS, it does not need to be constant for the whole seen. Far away patches could be coarser approximations than near ones[4].

3.2 Axons and dendrites

As described above, a segment is represented by a conical frustum and a section can be seen as a tube. The tube representation is ideally suited to form an abstract patch which control points are the morphological points. Figure 4a shows a section, composed by 8 segments, represented as a quad patch with an horizontal tessellation factor of 8 and a vertical tessellation factor of 6. The TES is responsible to displace the 63 generated vertices to bend this flat surface into a tube and position it in space. Each quadline is first transformed to a circle approximation in the xy-plane which diameter is defined by the morphological point (see Fig. 4b). The approximation quality depends upon the number of vertices, i.e. the vertical tessellation factor.

[4]Other parameters are also possible, like the curvature of the surface.

These cross-sections are then placed in space at the morphological point recorded position. To have them aligned and form a tube, we define a tube vector, which traces a path between the morphological points of a section. This tube vector is computed by the TCS by subtracting the positions of two consecutive cross-sections. Cross-sections are finally rotated to be perpendicular to the tube vector (see Fig. 5a). Figures 5b and 5c show segments rendered at different levels of details.

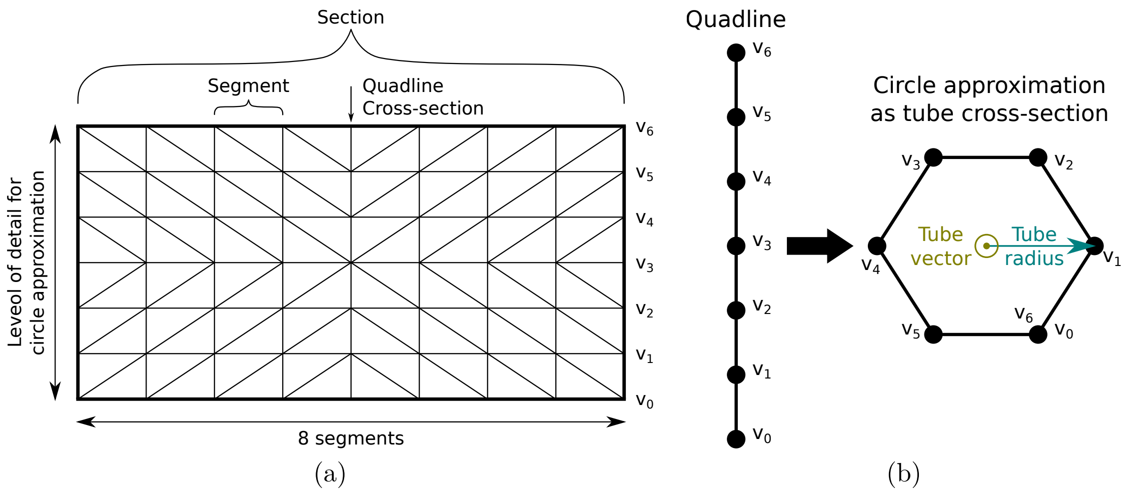


Figure 4: Sections as tubes. (a) Quad patch generated by the Tessellator. (b) Transformation of a quadline to a cross-section.

According to the OpenGL specification, each abstract patch is defined by a configurable but fixed number of vertices which cannot exceed 32[5]. The implication to our implementation is that a section has to be composed by a configurable yet fixed number of segments. Of course this is not the case with real morphologies. We thus have to break down longer sections in multiple ones, introducing sections that have one child. The last cross-section of shorter sections is replicated, as if there were multiple morphological points at the same location.

The branching between a parent section and one or more child sections is handled as if each child section was the sole continuation of its parent section. Look at figure 5a: if one of the two children were not present, it would have been as if the remaining child and the parent formed one section. This behavior is simple and produces good results. It does however not produce watertight meshes as part of the child section surface is inside the other child section. This would be a problem for translucent rendering or volume rendering. It is however possible to fix it as the branching is entirely defined by a limited number of morphological points: the branching point plus one point per intersecting section. The TES could displace the vertices of the

[5]The maximum of 32 can be circumvented by passing various vertex buffers or by concatenating vertex data.

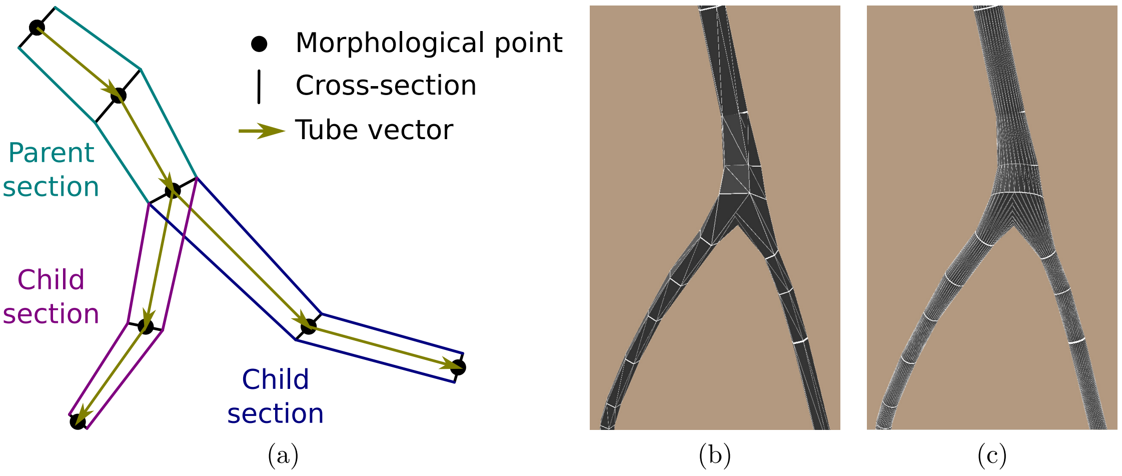


Figure 5: Cross-sections and branching. (a) Cross-sections alignment and sections branching. (b) Real example with a tessellation factor of 5. (c) Tessellation factor of 32.

first and last cross-sections of a section along the tube direction. Another degree of freedom is the rotation of the cross-section around the tube vector. The TES would use these two degrees of freedom to position the vertices such that each intersecting section takes the responsibility of some part of the branching. That would create clean boundaries between involved patches resulting in watertight meshes.

3.3 Somas

The same principle is used to render the soma and the beginning of the first-order sections. Each first-order section is defined as a patch, which is again a parametrized tube. The difference is that cross-sections are not at morphological points but are disposed between the soma and the first morphological point. The tube diameter is a function of the distance to the soma center. The radius is defined by the circle equation (with the soma radius) around distance zero. The radius is constant near the first morphological point of the section. In-between there is a configurable distance where the radius is defined by a cubic spline interpolation, which is C^1 continuous. This interpolation provides a smooth transition between the soma and the first-order section (see Fig. 6). Figures 7a and 7b show the effect of the interpolation starting point on the transition.

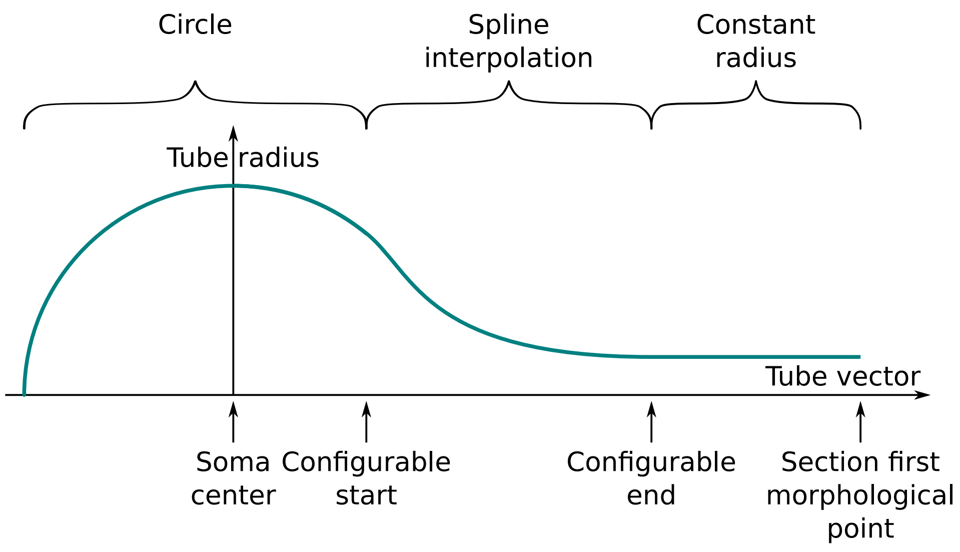
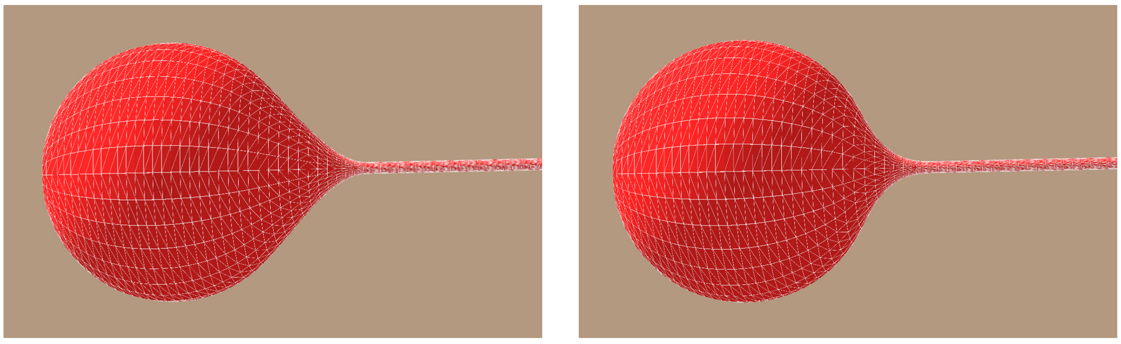


Figure 6: Radius evolution of first-order sections to create the soma.

The actual implementation suffers from the same kind of problem as branchings: surface meshes of multiple first-order branches will overlap at the soma. It can again be fixed by creating boundaries between branches, i.e. each branch takes the responsibility of some part of the soma.

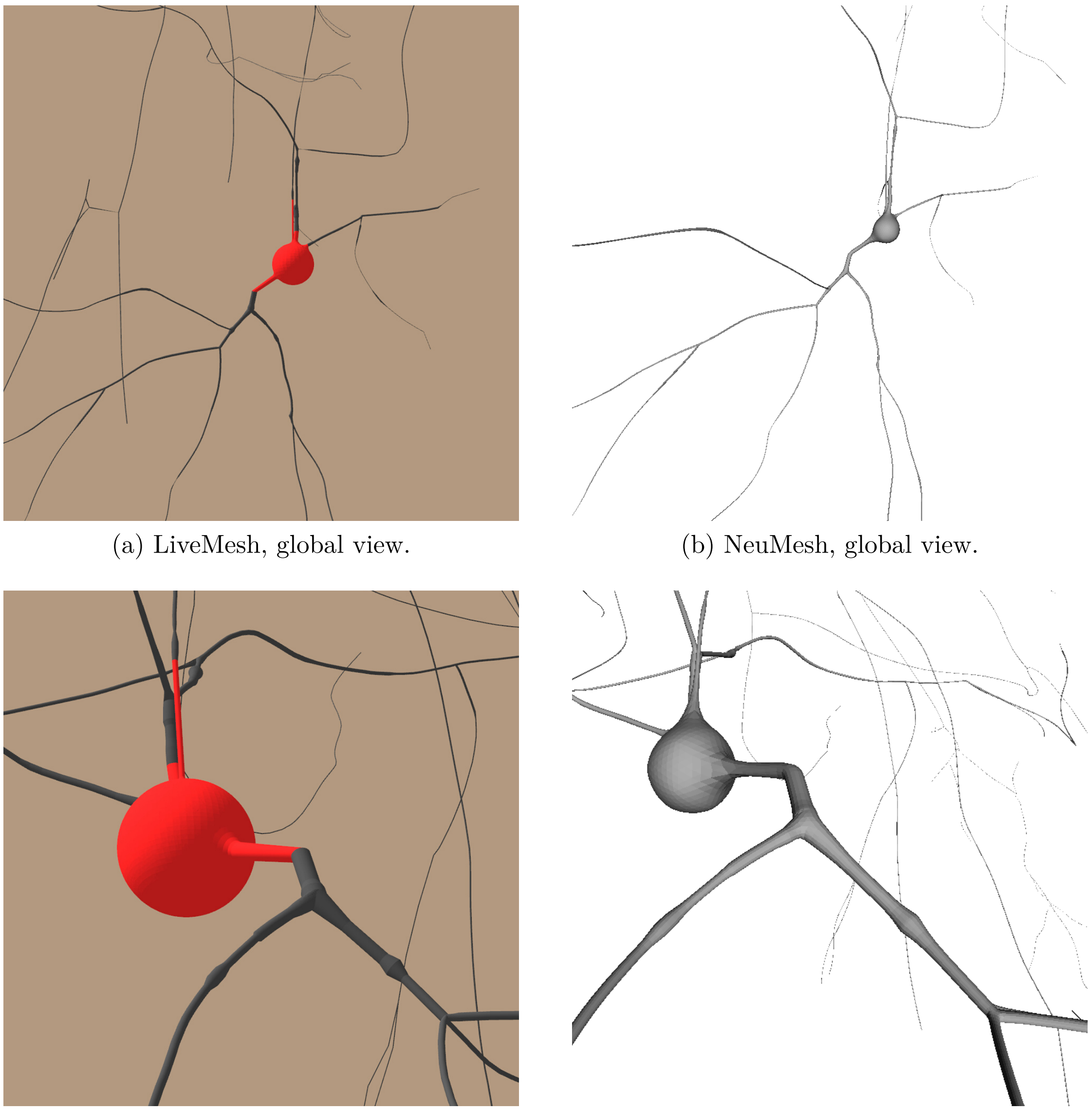


(a) (b)

Figure 7: Junction of first-order section and soma. (a) Interpolation starts at 70%, ends at 150% of soma radius. (b) Interpolation starts at 90%, ends at 150% of soma radius.

4 Results

4.1 Visualization of a single cell



(a) LiveMesh, global view.

(b) NeuMesh, global view.

(c) LiveMesh, detail view.

(d) NeuMesh, detail view.

Figure 8: Comparison of LiveMesh and Neumesh on a single cell.

Figures 8a and 8b show a global view of the *C040426* morphology rendered by LiveMesh and NeuMesh, respectively. While the camera position is not exactly the same and there is a lot of missing small ramifications, it is evident that the two softwares render the same thing. A more detailed view (see Fig. 8c and 8d) shows that NeuMesh performs some smoothing while LiveMesh renders the raw information provided by the morphology, without any prior about the smoothness of a neuron shape.

4.2 Space and bandwidth savings

The real-time rendering capability of the proposed solution saves storage space by eliminating the need to store the generated mesh, which is an intermediate results. The actual solution, NeuMesh, generates the mesh offline and stores it. For reference, the size of the mesh generated by the *C040426* morphology in the Polygon File Format[6] is of nine megabytes. This mesh is then loaded by a rendering software like RTNeuron. The LiveMesh library would instead load the morphology, which is stored anyway, and generate the mesh online.

An additional benefit of GPU-based tessellation is that it saves memory bandwidth. By generating the geometry directly on the GPU, we do not have to transfer it. Sole the control points, i.e. the morphological data, is copied from main memory to GPU memory.

The vertex buffer is filled by three floating point values (radius, x, y and z positions) per morphological point. There is as much vertices as there is segments in the morphology. An index buffer is then used to point to the vertices. As the number of segments per section is actually set to 32, there is 32 indices per section. As discussed before, some vertices are used more than once for branching or sections breakdown. The use of an index buffer thus saves further memory and bandwidth.

The *C040426* morphology consists of 271 sections and 6926 segments. The current implementation fixes the number of segments per section at 32. Some sections are thus broken in smaller pieces which results in 387 sections of 32 segments. Remember that smaller sections are padded to 32. The size of the vertex buffer is 6926 segments times 4 values per segment times 4 bytes per floating point value, resulting in roughly 108 kB. The size of the index buffer used to draw the branches is 387 sections times 32 segments times 4 bytes per unsigned int, resulting in roughly 48 kB. Adding some bytes for the index buffer used to draw the soma, this gives us 160400 bytes to transfer from main memory to GPU memory to render the whole morphology. Discounting the soma, the Tessellator would generate from $387 \cdot 32 \cdot 3 = 37152$ (tessellation factor of 3) to $387 \cdot 32 \cdot 64 = 792576$ (tessellation factor of 64) vertices. In comparison NeuMesh generates 240062 vertices which positions, requiring at least 3 MB, have to be copied to GPU memory.

[6] Also known as the Stanford Triangle Format.

5 Discussion

We first discuss an alternative which was envisaged before the current solution. We then present some limitations of the current implementation and some future directions.

5.1 Alternative

We first thought to use the same principle as presented in [Las+12], i.e. to apply a subdivision surface algorithm to smooth a coarse approximation of the branches. According to this paper, subdivision is by far the slower step of the overall process of mesh generation from neuron morphologies. As this is easily parallelizable, we wanted to implement it on the GPU to allow real-time rendering without the need of storing the entire mesh. See [Sch+14] for a discussion of the algorithms. As it is possible to implement some subdivision surface algorithm using hardware tessellation, it is not trivial. This approach would still need to generate the coarse approximation first. Trying to do so on the GPU would cause additional difficulties as it would have to be done before the rendering pipeline, possibly with some compute shader. This alternative was thus abandoned in favor of the presented approach.

5.2 Limitations

As discussed earlier, the actual implementation produces meshes which are not watertight due to overlap and inner surfaces. It can be overcome by displacing the vertices but should be done once we are satisfied with the soma and branching algorithms, which are actually quite basic. Although the principle stays the same, dealing with the watertight issue before settling on those algorithms would be a waste of resource.

5.3 Future directions

A great visual improvement would be to introduce some additional cross-sections between morphological points. The position and radius of these additional cross-sections would be computed by a cubic spline interpolation. Their concentration would be dynamically chosen according to the curvature of the path, i.e. dynamic re-sampling. This would smooth the geometry and the neuron would look more natural. See figure 9 for a visual assessment. If we don't fully trust the morphology, because of some noise in the measurements, it would be appropriate to opt for an approximation scheme instead. The penalized spline regression would allow for a balance between the data term, i.e. the conformance to the measurements, and the prior, a smoothness measure.

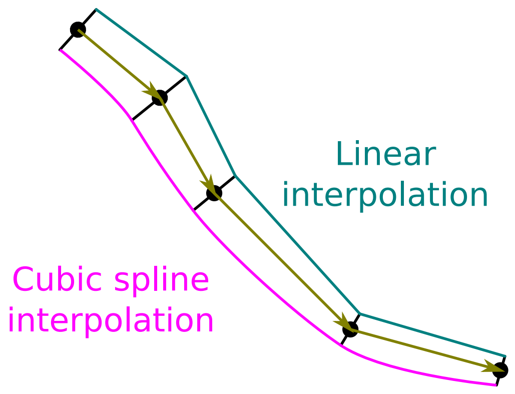


Figure 9: Linear versus cubic spline interpolations.

Going further with visual improvement we could try to make the soma looks more real. The actual sphere approximation is somewhat artificial. Neuronize uses mass-spring systems to deform the soma in the direction of the branches [Bri+13]. A simpler approach would be to use displacement mapping to introduce some noise on the surface.

Yet another improvement could be made on the branching. This would require to look at real neurons as the morphology itself says nothing about how branching is performed. All we know is that some section is a child of another one. Maybe we should have different algorithms and choose one according to the branching type. The type could depend on the number of children, the angle between them, if there is evident main and second children etc. NeuMesh always defines a main child which is the continuation of the parent. The secondary children keep their diameters and are inserted by mean of a hole in the knot. Their could be other approaches too.

Dynamic level of detail is an attractive feature to reduce the complexity of the scene. It works by reducing the tessellation factor where details are less needed. A trivial heuristic is to use the distance from the camera: farther away the patch is from the camera, the less vertices are needed to render it visually accurately. Another heuristic would be to keep constant screen-space projected triangle sizes. Yet another heuristic would be to increase the tessellation on the silhouette so it looks smoother. An advanced technique for pixel accurate rendering is to use a bounding structure called a slefe; an acronym meaning Subdividable Linear Efficient Function Enclosures [LP01]. The concept of pixel accurate rendering is used to determine a tessellation factor that guarantees that a tessellation of a surface differs from the true surface, when measured in screen-space pixel coordinates, by at most half a pixel.

Some additional features would still be needed for a production-quality library. First we will have to take spines into account. Another great feature would be to export and save the generated mesh for further processing. This could be done by storing the generated vertices position to a buffer and copying it back to main memory.

The final step, once we are satisfied with the quality of the generated mesh, will be to factor out the core algorithms as a library and integrate it in the BBP rendering infrastructure.

6 Conclusion

While the software is still a prototype and there is a lot of work to be done, we have demonstrated the feasibility of highly parallel mesh generation for neuronal visualization designed to run on GPU hardware. The presented approach is able to turn neuron morphologies into surface meshes in real-time and enables real-time rendering of large sets of neurons. The advantages of the proposed solution over the current solution, NeuMesh, is threefold: real-time online rendering, no storage of the mesh, lower memory bandwidth. The implementation, both the core library and the demonstration application, is also very concise: around 1000 lines of C++ and 300 lines of GLSL. The code is available at `https://bbpcode.epfl.ch/code/#/admin/projects/viz/livemesh`.

The actual implementation produces membrane meshes which accurately represent the neuron morphologies. It is the simplest algorithm based solely on the morphology. It does not use any prior knowledge about the shape of neurons which results in neurons who look artificial due to sudden changes of branch diameter or direction. For higher fidelity and nicer looking we need to integrate a biological model, i.e. to add some prior knowledge in addition to the morphology. The introduction of additional cross-sections between morphological points will make the neurons smoother and more realistic. Some prior knowledge about the shape of somas and branchings would also make them more realistic. Note that we already use some prior knowledge to smoothly connect first-order sections to somas. We could then compare with real neurons to assess the accuracy of our reconstruction and learn new priors. An accurate representation may enable the use of the mesh for other purposes than visualization, such as functional modeling or simulation.

References

[Bri+13] Juan P Brito et al. “Neuronize: a tool for building realistic neuronal cell morphologies”. In: *Frontiers in neuroanatomy* 7 (2013).

[GG90] Jacob R Glaser and Edmund M Glaser. “Neuron imaging with Neurolucida—a PC-based system for image combining microscopy”. In: *Computerized Medical Imaging and Graphics* 14.5 (1990), pp. 307–317.

[Las+12] Sébastien Lasserre et al. “A neuron membrane mesh representation for visualization of electrophysiological simulations”. In: *Visualization and Computer Graphics, IEEE Transactions on* 18.2 (2012), pp. 214–227.

[LP01] David Lutterkort and Jörg Peters. “Optimized refinable enclosures of multivariate polynomial pieces”. In: *Computer Aided Geometric Design* 18.9 (2001), pp. 851–863.

[Sch+14] H Schäfer et al. “State of the Art Report on Real-time Rendering with Hardware Tessellation”. In: *Eurographics 2014-State of the Art Reports.* The Eurographics Association. 2014, pp. 93–117.