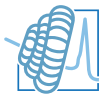

Fachgebiet für Bioanaloge
Informationsverarbeitung

Technichal University of Munich
Department of Electrical and Computer Engineering
Bio-inspired Information Processing

Projektpraktikum Bioanaloge Informationsverarbeitung

Simulating the Firing of Cochlear Neurons

Written by:
Karahan Yilmazer

Munich, 31. August 2022
SS 2022

1 Introduction

There are multiple ways of modeling the neuronal networks in the brain. One of the most prominent neuronal models is called the Hodgkin-Huxley Model (Hodgkin and Huxley, 1952).

Prior to this project, a human cochlea was reconstructed using a high-resolution micro-CT scan (Bai et al., 2019). Using this model, three dimensional coordinates of each scanned neuron was determined. Later, the Hodgkin-Huxley model was used to compute membrane potential values of each compartment, given a stimulation of a cochlear implant electrode.

The goal of the current project was to use all this data to simulate the firing of the cochlear neurons. For this purpose, the open-source 3D modeling software Blender (Blender Online Community, 2022) and its Python (Van Rossum and Drake Jr, 1995) scripting feature were used.

2 Data

The provided data consisted of:

- Neuronal fiber coordinates

  The model had 400 neuronal fibers. Each fiber were described by 200 to 500 coordinates corresponding to x, y, z values in the 3D space.

- Membrane voltage values

  For each fiber, 49 compartments were modeled. For each compartment, 1085 values were given which corresponded to different time steps.

- Compartment lengths

  Each compartment had a certain length. Compartments with different lengths corresponded to different parts of the neuron: soma, Node of Ranvier, internode, etc.

3 3D Model of the Fibers

3.1 Clearing the Scene

The first thing to do before starting with the simulation was clearing the default Blender workspace. As it can be seen in Figure 1, the default Blender window comes with a cube mesh, a light and a camera. Moreover, removing all of

these from the scene collection is not enough for removing them from the whole project. Although, leaving these in the background would not effect the performance of the program, it is good practice to start with a clean slate.

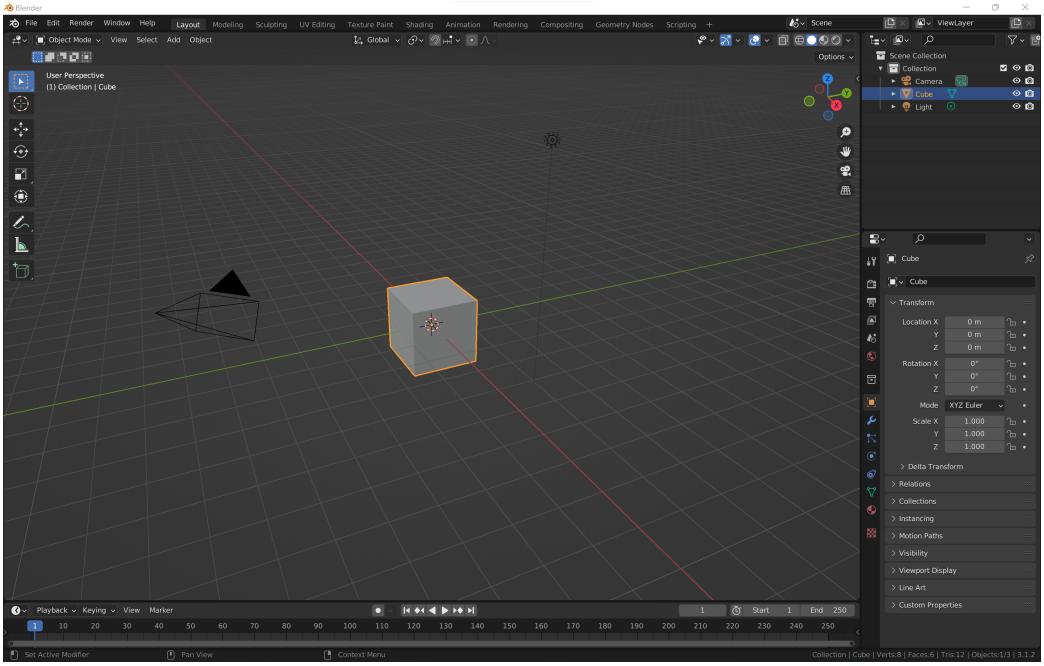

FIGURE 1 – The default Blender window. It comes with a mesh of a cube, a light and a camera. These have to be completely wiped for further processing

Clearing the scene could be easily done using the code from a public GitHub repository.

3.2 Reading in the Coordinates

As mentioned before, the coordinates for the fibers were provided. Fortunately, these could be read in to Blender pretty easily. Here, the built-in `from_pydata()` function came in handy. This function converts a list of 3D vertices into meshes.

3.3 Dividing the Meshes Into Compartments

Once individual meshes were created for all 400 fibers, the next step was to calculate the compartment locations. For this purpose a small algorithm was

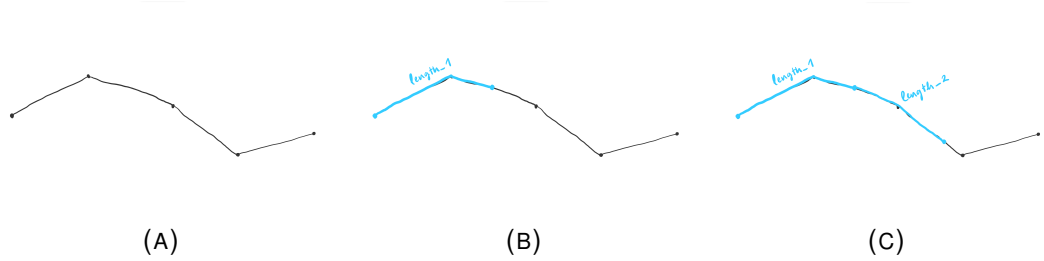

FIGURE 2 – Steps taken to compute compartment positions. (A) Each point corresponds to a coordinate used to create the mesh. (B) The first length value is used to calculate the point where the compartment would reach. The starting point is taken the first coordinate. (C) The next length value is used to calculate the reach of the next compartment. This time, the calculation starts from the point where the last compartment ended.

written which iteratively walks over the coordinates used to create the meshes. In the following, this algorithm will be discussed with the visual aid in Figure 1.

As it can be seen in Figure 2a, all the coordinates were connected by straight lines to create meshes for each fiber. Here, each point corresponded to a coordinate. In Figure 2b, the first length vector was used to calculate the end point of the first compartment. To achieve this, first the distance between the first two coordinates was calculated using the Pythagorean theorem. If the compartment length was smaller than this distance, then the compartment's end point should have lied between these two coordinates. In this case, the scalar value corresponding to the length of the compartment was multiplied with the unit vector originating from the first coordinate and pointing towards the second coordinate. The resulting vector was then added to the first coordinate to find the end point of the first compartment.

However, if the length of the compartment was larger than the distance between two neighboring coordinates, this distance was subtracted from the length, assuming that it was "walked". Then the next distance was calculated, i.e., the distance between the second and third coordinate. Again, if it was smaller than the length, the end point was calculated. Else, the distance was walked over again.

After the end point of the compartment was calculated, the starting point for the next iteration was chosen to be this end point. This way, all the compartments end up lying directly next to each other.

It is worth noting that, it was simpler to write this algorithm in 2D with known values or randomly generated simple values. Later, generalizing it from 2D to 3D was very straightforward.

At this point, the meshes were created and all the compartments' start

and end points were calculated. The initial approach was to separate the meshes at these points, as the points lied directly on top of the the meshes. However, doing this proved to be a challenging task. Furthermore, the pipeline of creating the meshes, calculating the points and then dividing the meshes did not seem very effective. So, an alternative approach was followed.

In the alternative approach, the first thing to do was to calculate the compartment positions once and store them in a pickle file. This way, the whole computation had to be done once which would save computation time. After storing these values, they could be used to create meshes for the compartments. This way, creating the meshes and dividing them into compartments boiled down to directly creating compartment meshes. As the calculation of compartment meshes used no approximations, even though all the sub-meshes were separate, each sub-mesh followed directly the previous one with no gaps in between.

3.4 Adding Modifiers to the Meshes

To later color the compartments based on membrane potential changes, materials had to be added to the meshes. However, at the current state, the meshes were simple lines with no depth. A possible approach would be to convert these meshes into tubes and then color their faces. Hoping for a simpler approach, the skin modifier was used. This modifier is used to quickly generate base meshes for sculpting, like for creating human-like models.

After adding the skin modifier, the results did not look very promising as it can be seen in the bottom subplot of Figure 3. One reason was that the skin modifier looked too rough and the second reason was that different compartments could be distinguished from each other.

To get a closer look to reality, both ends of each compartment were made slimmer. This way, the compartments could be distinguished from each other and the myelin sheaths could also be shown in the model. This version can be seen in the middle subplot of Figure 3. Lastly, to get a smoother look, the subdivision surface modifier was applied, which is used to split the faces of a mesh into smaller faces, giving it a smooth appearance. The results can be seen in the top subplot of Figure 3.

Although the combination of the subdivision surface modifier and making the ends of compartments thinner resulted in a visually appealing model, there were some problems. First of all, the computation time got clearly longer. For each mesh, both modifiers had to be added and the skin modifier had to be made thinner at certain points. Secondly, even though the model looked better with the added modifiers, the fine details could not be

seen when the camera was zoomed out to focus on all 400 fibers. That is why, in the end only the skin modifier was added to later color for the animation. It is worth noting that the modifiers could be added with the built-in function bpy.ops.object.modifier_add().

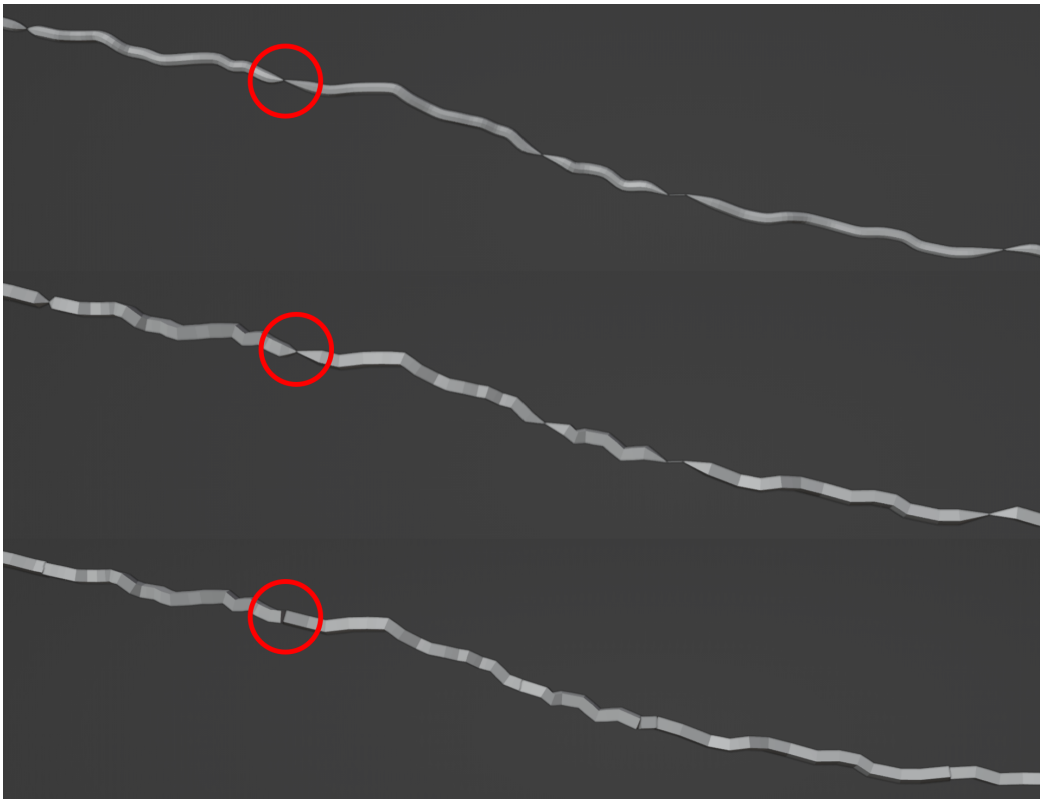

FIGURE 3 – Different levels of detail for modeling the fiber. The red circles show the meshes corresponding to nodes of Ranvier which were too small to be modeled by Blender. For a further discussion refer to Figure 4. (Bottom) Only the skin modifier was added to give depth to the line meshes. (Middle) The ends of the compartments were tapered down for easier separation. (Top) The subdivision surface modifier was applied to smoothen out the skin modifier. This results in a fine look. However, this results in a high computation time.

3.5 Dealing with Nodes of Ranvier

One other thing that was observed after adding the modifiers was that some compartments, like the nodes of Ranvier, were too small to be displayed. This could have been ignored as they were too small to be seen when zoomed out.

However, their lacking tempered with the voltage mapping in further steps. So, this issue had to be tackled.

An approach that could be followed to solve this issue could be to make the whole 3D model larger. However, a simpler approach was to add the nodes of Ranvier to their directly neighboring compartments. This way, the most compartments grew slightly larger but this was not apparent to the naked eye.

Another reason for this simplification was that the saltatory conduction could not be meaningfully visualized with such tiny compartments corresponding to the nodes of Ranvier. So the initial goal of simulating the saltatory conduction by coloring the faces of the nodes of Ranvier had to be dropped which meant that the visualization of the nodes of Ranvier was not a priority anymore.

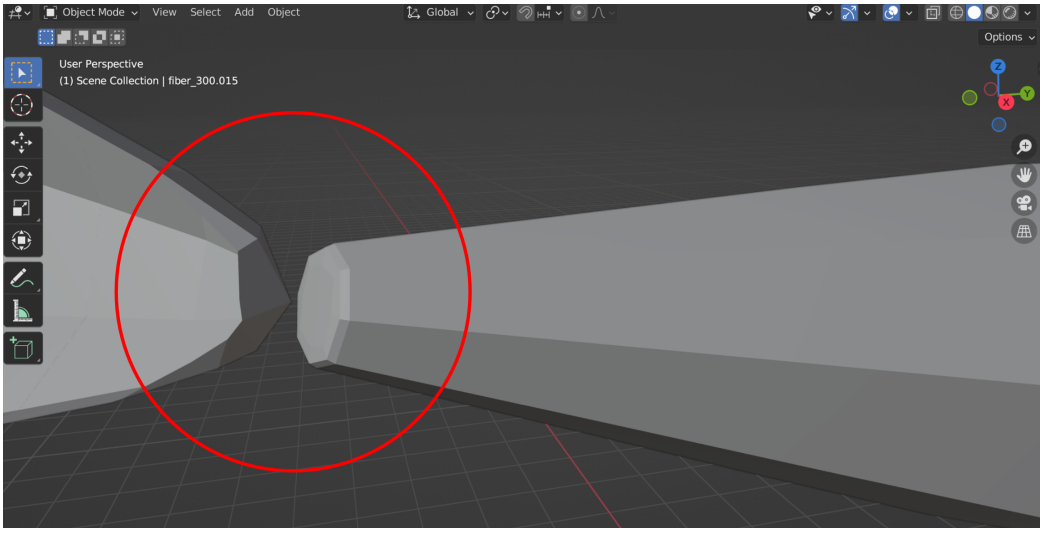

FIGURE 4 – A zoom-in onto the node of Ranvier which could not be modeled by Blender due to its small length. This problem was solved by removing the nodes of Ranvier from the 3D model as they were too small to be seen anyway.

Removing the nodes of Ranvier from the 3D model also required removing their corresponding voltage values from the voltage values array, so that the mapping stayed correct.

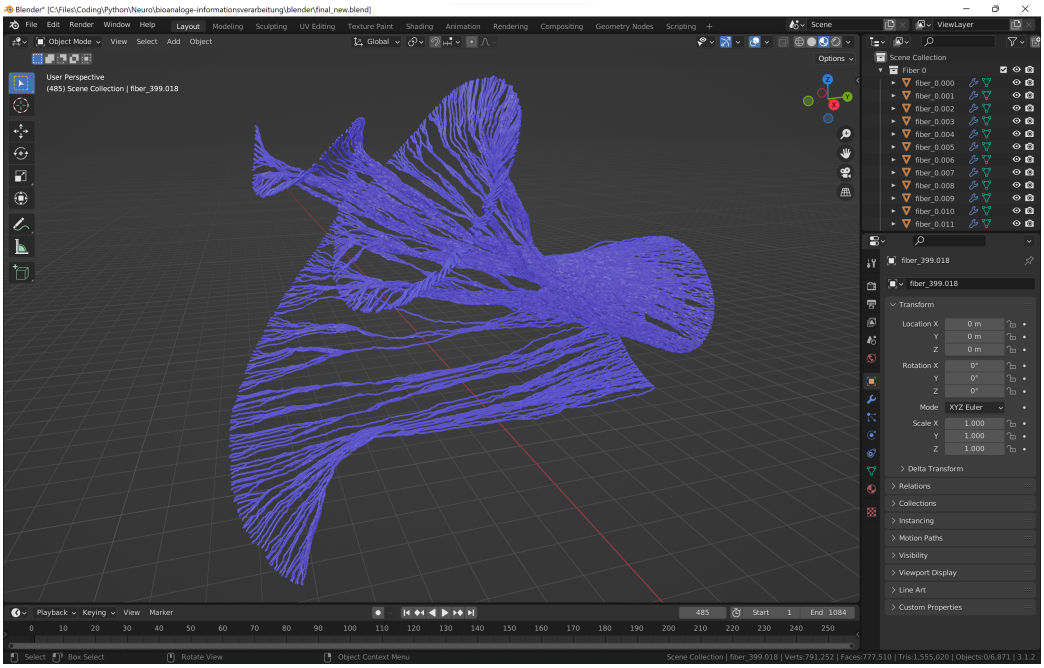

FIGURE 5 – The resulting 3D model of the cochlea once all the changes have been applied. The fine details cannot be seen from this distance, justifying the simplifications.

4 Animating the Action Potentials

4.1 Finding the Action Potentials

For the animation of the membrane voltage changes, a coloring method was used. Although the initial thought was to map small voltage ranges to different hues of a color, this proved to be computationally too expensive. So, only the action potentials were visualized.

To find the firing compartments and the frames where they fire a simple threshold model was used. First, the maximum voltage value out of all voltage values was computed. This value was the reference for all other membrane voltages. Then, a threshold was selected, which was 55% in the final setup. So, any compartment reaching 55% of the maximum voltage value was assumed to be firing an action potential.

Once the threshold was set, all the compartments were iterated over for all frames. Every time a compartment was above the threshold, its fiber, mesh index and frame was stored in a Python dictionary. Once all the values were iterated over, the same procedure was repeated, this time with all the compartments that fired before and then dropped below the threshold again. These

results were stored in a second dictionary.

This way, the dictionaries could be used to animate the action potential propagation through the fibers.

Refractory periods were not taken into account while computing the dictionaries, as they were taken into account already during the data generation part.

It is worth noting that, lowering the firing threshold increases the number of firing compartments which also changes the animation in further steps.

4.2 Adding Keyframes

For animating any value, a so-called keyframe was used. A keyframe was added to each frame, where a state was reached which was wished to be explicitly displayed. Later, Blender connected one keyframe to the next one to allow for a smooth transition.

For coloring the compartments, the previously created dictionaries were used. A keyframe was added when the membrane voltage of a compartment exceeded a certain threshold, indicating the firing of an action potential. Another keyframe was added when the voltage dropped below this threshold again. Assuming that the resting potential was mapped to the color blue, whereas an action potential was mapped to red, at the first keyframe, the compartment should be completely red and at the second keyframe, it should be blue. In the frames between these two keyframes, the compartment gradually changed colors from red to blue.

However, this was not enough for the whole animation, as another keyframe had to be added a few frames before the action potential keyframe. Otherwise, Blender implicitly assumes the keyframe before the action potential keyframe to be the first ever frame of the animation. Then, up until the frame of the firing was reached, the compartment slowly changed colors from blue to red, which sometimes lasted more than 5 seconds. That is where the prior keyframe came into play. With this keyframe, it was signaled to Blender, that the change from blue to red should happen in as many frames as there were between the action potential keyframe and the helper keyframe before that. The number of frames between these two keyframes defined how fast a compartment reaches the color red. For this application this number was chosen to be 10.

Although this method produced working animations, the performance could be further increased. With the explained setup, a keyframe was added to each frame a compartment was above the threshold. However, it was enough to add only one keyframe to the frame where the threshold was exceeded, as a smooth transition from this keyframe to the next could be handled by Blender,

as mentioned before. With this simplification, a lot of unnecessary keyframes were removed, making the program run faster.

Figure 6 shows the final setup of the keyframes.

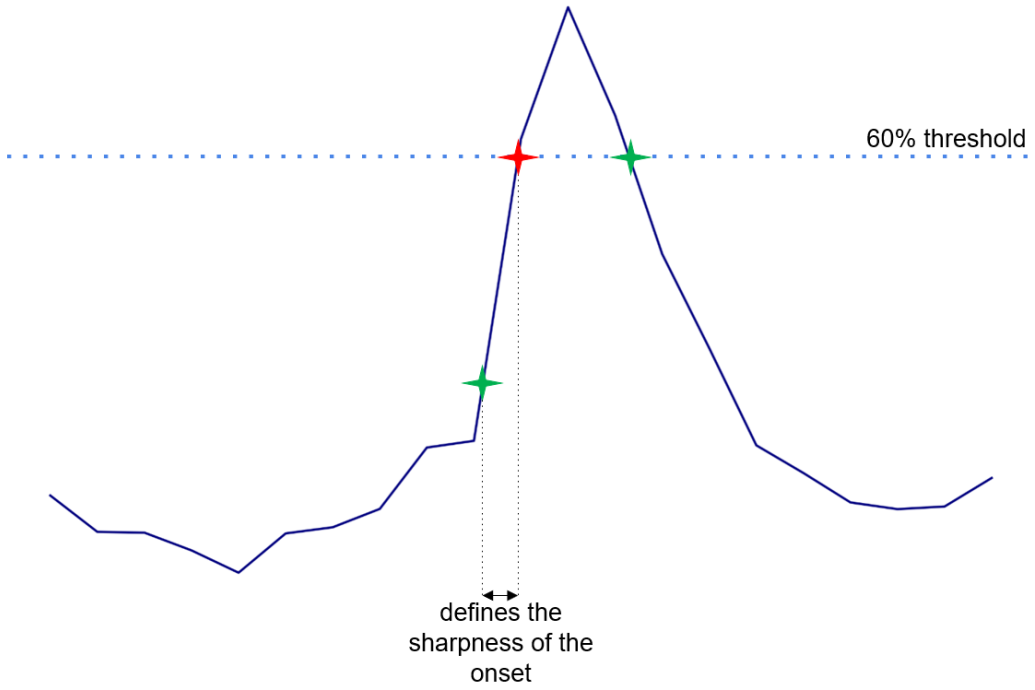

FIGURE 6 – The voltage changes of a compartments going through an action potential. On the x-axis is the time and on the y-axis is the voltage value. Once the action potential detection threshold is exceeded, a keyframe is added to the current time step. Another keyframe is added a fixed number of time steps before this keyframe. The difference between these two keyframes defines the sharpness of the action potential's onset. Lastly, another keyframe is added once the voltage value drops below the detection threshold. The green keyframes correspond to the color being set to the resting potential, whereas the red keyframe corresponds to the color being set to the firing potential value.

Lastly, it can be noted that the interpolation from the color blue to red and then back again to blue can be done automatically by Blender. However, this is mostly done through a simple shading. If, more details are desired to be shown, a so-called color ramp node can be used in the material properties. This node helps define how this interpolation should look like and can add middle colors, like cyan, green and yellow, to the color transitions. With this node, the animations looked subjectively better.

5 Rendering

After everything was done, the only steps remaining were to set up the camera and the lights and to render the video. For viewing the model from different angles, a rotating camera was set up. Furthermore, two lights were used for making the colors look more vibrant. A screenshot of the final setup in Blender can be seen in Figure 7.

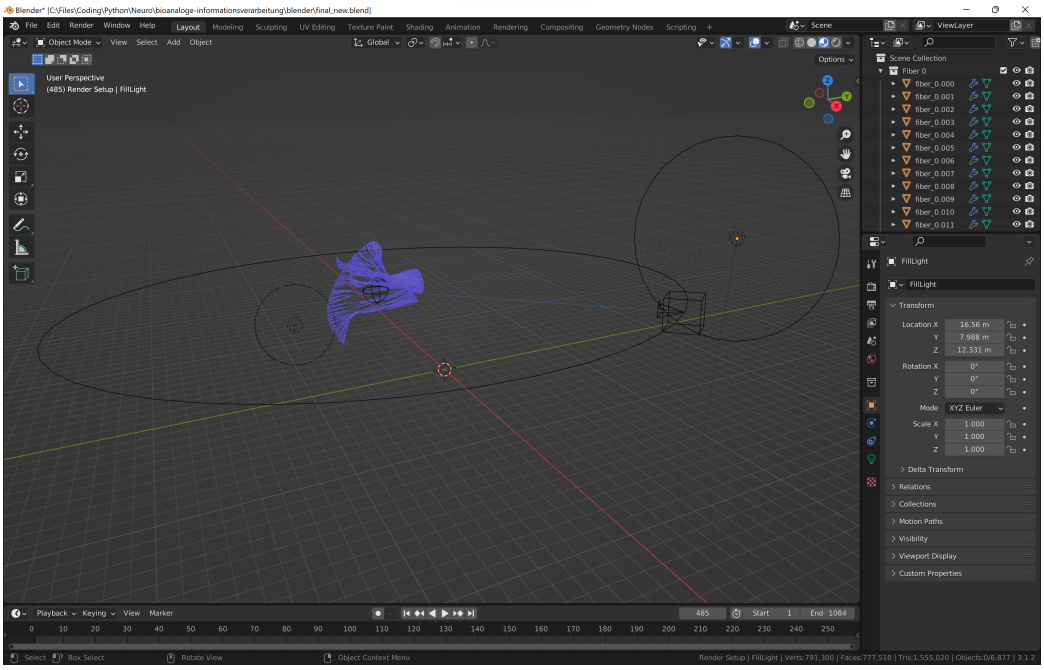

FIGURE 7 – The final scene setup consisting of the model of the cochlear neurons, a rotating camera and two lights.

The rendering was done frame by frame, meaning that the animation was not rendered to output a video, but rather an image for each frame. This way, if the rendering stopped or crashed abruptly, the program could be restarted and rendering could go on from the last frame that was produced. This added an extra level of complexity where all the images had to be stitched together to a video. However, Blender also had simple video editing functionalities to allow this.

The first render was done on all 1084 frames. However, later it was observed that not all frames were relevant for the animation. Figure 8 clearly shows this. In this figure, each yellow circle corresponds to a keyframe that was added for the animation. As, it can be seen, nearly all the keyframes were in the range of frames 100 to 700. This meant that rendering only this range

would suffice to fully capture the animation, given that the cochlear implant electrode activation was not to be animated. This simplification sped up the overall rendering process.

FIGURE 8 – The animation tab of Blender showing the frames where a keyframe was set, where each yellow circle corresponds to a keyframe. The top red row shows all the keyframes in the animation and that most of them lie between the frames 100 to 700. This means that rendering this interval was enough to capture the whole action.

Figure 9 shows some statistics from three different renders. As discussed, the first render used all 1084 frames, however the action potential detection threshold was way too high to a proper animation. With this threshold, only a few compartments fired, making the rendering process very simple, as not much had changed from frame to frame.

In the following renders, the simplification of rendering only the 100-700 frames was utilized to lower the rendering time. The runtime for these cases were higher than the first case because the threshold was selected to be lower, causing more compartments to fire which had to be animated. However, it can be seen that, even with the final settings of the third render, the whole runtime was about two hours, which can be acceptable taking the sheer number of objects and animations into account.

After all these considerations, a screenshot from the final animation can be seen in Figure 10. The video lasted half a minute with a simple to follow action potential propagation. At this frame, the propagation of action potentials both to the apex and the base of the cochlea can be seen, as none of the compartments were in the refractory period. It can also be seen that the electrode stimulation caused a wide area to be activated.

The mainly dominant blue color corresponded to the resting potential, whereas the red color was used for indicating the exceeding of the action potential detection threshold. As mentioned before, the colors in between, like cyan, green and yellow, are simple interpolations using the color ramp function. These colors indicate at which point of the action potential cycle the compartments are.

Frames	Color Ramp	Threshold	Runtime	frame/s
1084	No	0.85	1:30 h	5
600	No	0.70	1:58 h	10
600	Yes	0.60	2:02 h	10,5

FIGURE 9 – A table showing the rendering times for different settings. Even though more frames were rendered, the first render took the least time, as the threshold was too high and not a lot of compartments fired. Only rendering the 600 frames where real firing happened resulted in a longer rendering time, as the firing detection threshold was decreased and thus more frames had to be animated.

6 General Tips

Throughout the project, there were some points that were very helpful, which can be considered in future 3D modeling projects.

It was already mentioned that Blender had a Python integration. However, it is worth emphasizing that this can be made even simpler by activating the Python tooltips. For this Edit –> Preferences –> Interface –> Python Tooltips has to be toggled on. Then, hovering over any button will also show what the Python command for this button is. Another way of directly seeing the Python commands for the actions taken is to open a window with the "Info" function. Lastly, Window –> Toggle System Console can be useful to see the output of the code. This is especially helpful when there are iterations. By using, for example, the tqdm library (*Tqdm/Tqdm* 2022), the progress of the iterations can be seen. This way the user can make sure that Blender did not crash, but in fact still running.

At certain points, it became logical to save the Blender workspaces. For example, after reading in the coordinates, importing the 3D models, adding materials to the fibers or adding animations to the fibers separate saves were made. This way some these processes that take a long time could be cached. If a change was necessary at a certain point, then the latest relevant Blender workspace could be opened, without having to rerun the whole steps again.

Another way of caching was with storing some values that had to be calculated every time a script was run. For example, the starting and end points of each compartment had to be calculated every time the 3D model had to be created from scratch. Or some voltage values that corresponded to the nodes of Ranvier had to be discarded so that the voltage mapping was correct. How-

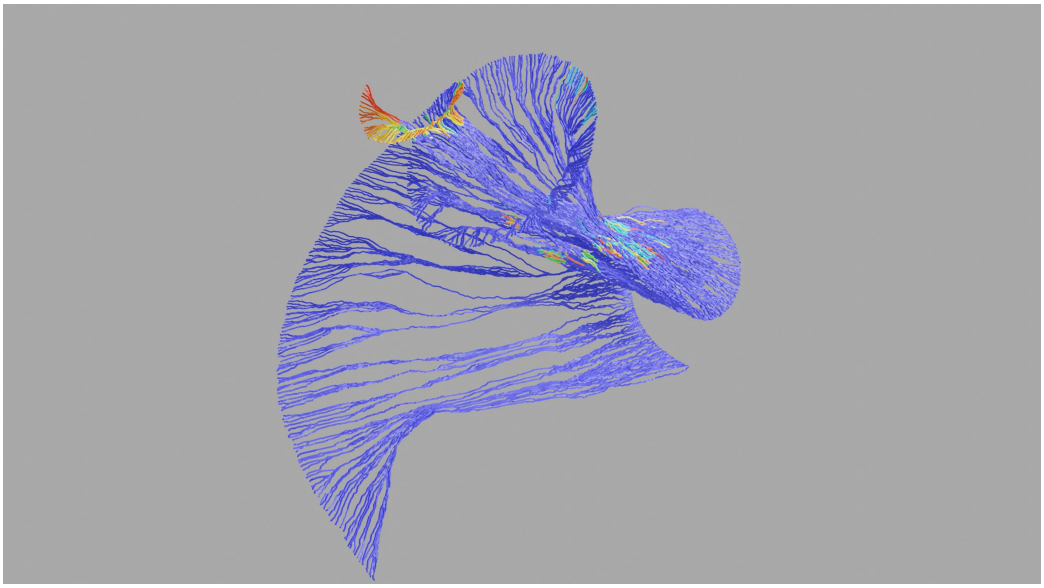

FIGURE 10 – A frame out of the final animation. It can be seen that the action potentials propagated both to the apex and the base, as none of the neurons are in the refractory period. The color ramp adds color gradients, making it easier to distinguish at which point of the action potential cycle the compartment is.

ever, saving these oftenly used values into pickle files helped with simplifying and speeding up the code.

Modularizing code by dividing big sections into separate scripts also helped with having a flow. If the whole scripts were put into one long script, the runtime would be too long, and the user might be tended to close the program thinking that Blender crashed, as it tends to be unresponsive while Python programs are running. By separating chunks of processing into separate scripts, the state of the workspace could be saved into different .blend files to continue from again if anything crashes.

While running multiple iterations, the iterations tend to get longer through time. This can also happen for rendering in some cases. A quick fix can be to not run the whole iteration at once, but to do it in small chunks. This comes with less automation, but at least the whole processing takes less time.

Lastly, it was very helpful to test new changes on single fibers. This way, it could be seen how changes in the code reflected to the 3D model and animation. It was also much quicker to perform any processing on a single fiber. So, prototyping could be done rapidly and before committing to a change and waiting for an hour for the whole script to run for all 400 fibers, one could make sure that it was the right change to apply.

7 Future Improvements

As there is no "right way" of accomplishing the task of animating the firing of the cochlear neurons, there will always be points one can tweak to optimize the animation. For the current state of the animation, a few of these points can be listed.

First of all, the STL files of the cochlea and the cochlear implant can be used for the animation. It would be helpful to show which cochlear implant electrode gets activated. This way it can be more easily seen that the neurons in the proximity of this electrode get stimulated first. For this animation, it would make sense to make the 3D models out of transparent material to have the focus on the action potential propagation.

The STL files are not the only place where the material properties can be tweaked. In the final animation, the compartments in the front block off the compartments in the back. This could be solved by either decreasing the alpha value or adding some transparency to the materials.

The voltage-color mapping can also be improved. Right now, the colors do not fully represent the actual membrane voltages. The way the colors were computed was more of a rough estimate at which point of the action potential cycle the compartment was at.

Furthermore, the action potential detection can be further improved. Right now, a global threshold is being used. However, each fiber's compartments have different resting potentials. Taking this variability into account and normalizing the voltage values fiber by fiber would help with detecting action potentials more robustly.

Although it was not thoroughly tested, the rendering properties can be tweaked to allow for a faster or a better looking render. This can be utilized for faster prototyping or visually appealing animations.

The animations can also be improved by changing the camera focus throughout the animation. For example, the camera could follow the action potentials or areas with the most activation. Another approach could be to look at certain cross-sections, fibers, or compartments of interest. Rendering the animation with cameras from different angles and putting them all into a video collage might also be useful. But all of these suggestions come with an increased rendering time. So, a meaningful trade-off has to be found.

Another improvement could be to put the whole model into perspective by using a human or human head model. With this approach, all the components of a cochlear implant can be visualized and its activation can be put into context by showing how it affects the neurons. This could prove to be a helpful animation for cochlear implant users.

8 Conclusion

In this project, the activation of the cochlear neurons upon a stimulation through a cochlear implant was animated using the 3D modeling software Blender. For this purpose, the membrane voltage values that were computed using the Hodgkin-Huxley model were used to find action potential firings. These were then animated using color changes of the compartments.

The final animation could clearly show the propagation of action potentials through the cochlea. With a lot of performance tweaks, the whole animation could be rendered within a reasonable time of two hours. The modularized nature of the code made it possible to swap the data with another simulated data to update the animation for different kinds of experiments.

Code Availability

The code for this project can be found under:
https://github.com/karahanyilmazer/Bioanaloge-Informationsverarbeitung.

References

Bai, Siwei, Jörg Encke, Miguel Obando-Leitón, Robin Weiß, Friederike Schäfer, Jakob Eberharter, Frank Böhnke, and Werner Hemmert (2019). "Electrical Stimulation in the Human Cochlea: A Computational Study Based on High-Resolution Micro-CT Scans". In: *Frontiers in Neuroscience* 13.

Blender Online Community (2022). *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam.

Hodgkin, A. L. and A. F. Huxley (1952). "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve". In: *The Journal of Physiology* 117.4, pp. 500–544. pmid: 12991237.

Tqdm/Tqdm (2022). tqdm developers.

Van Rossum, Guido and Fred L Drake Jr (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.