Synapse Modeling in Neuro-VISOR

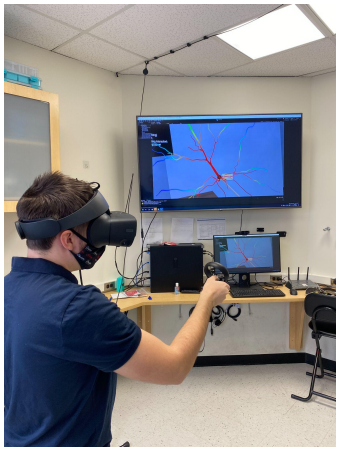
Michael Bennett

Fall 2021 Undergraduate Research Program

Center for Computational Mathematics and Modeling

Abstract

VISOR (Virtual Interactive Simulation of Reality) is a tool developed by Temple University's Center for Computational Mathematics and Modeling. The goal of this project is to produce efficient software immersed within virtual reality (VR) that can visualize real-time interactions with simulations.



The simulations are principled by mathematical equations. Unlike usual simulations, Neuro-VISOR can be visualized in real time. This means the user can interact with simulations and get instant feedback all while in VR.

1. Introduction

During my Fall 2021 research, the main goal for the semester was to implement a model to latch neurons together. These neurons should be able to interact with each other,that is, sending voltage by way of an electrical event called an action potential. This would be at the junction of two neurons, called a synapse. The approach to accomplish this goal was to create a synapse model that communicates voltage from the presynaptic to the postsynaptic neuron. The voltage would be copied from the presynaptic neuron and then applied to the vertex the postsynaptic neuron is located at. This rudimentary framework was simply for testing purposes before a proper synapse function could be implemented. This was a huge objective to tackle on all fronts since the present layout of the Neuro-VISOR program only allowed for one neuron simulation to be run at a time. Therefore, the biggest question that existed was how to utilize a synaptic event to communicate voltages from one

neuron to another?

2. Problem Description

An overarching goal was to create a framework that acted similar to the way voltage clamps work within the existing code. In the current software, clamps are a tool that allow users to select a region on the neuron and constantly apply voltage. The synaptic model would be referenced from this model. However, my synaptic model consists of two spheres in Unity. Thus, my first problem was… how do I create a synapse method or data structure in such a way that it acts as one? Additionally, the presynaptic and postsynaptic spheres would need to be attached to the neuron itself. This was a matter of visualization questions. Such as, the size of the spheres, an arrow pointing and distinguishing the presynaptic from the postsynaptic, and also a feature to enable the placement of synapses. Lastly, a fundamental problem to be solved was how the placement of the synapse would work. That being said, how would the user have the ability to ray-cast onto a given neuron and receive information back? The information received needed to be what node index the user ray-casted onto and also where that location was given in 3D space.

2.1 Visualization Problem and Solution Description

Synapses are gaps of extracellular space of about 20-40 nanometers wide. For size comparison, one inch is about 25.4 million nanometers long. In light of this fact, Neuro-VISOR is run in a 3-Dimensional virtual reality space, so there must be some compromises to allow for user friendliness. My visual model for the synapse consists of two spheres in Unity. One sphere is the presynaptic or sending neuron and the other is the postsynaptic or receiving neuron. Once the complete synapse is placed, an arrow is instantiated; it points from the presynaptic to the postsynaptic neuron. The arrow helps visually tell the two spheres apart. Then, Depending on where each synapse is placed, it is then scaled according to the size of the dendrite. This was extremely important since I did not want the size of the synapse to overpower the size of the soma. In fact, the soma should be about 100 times larger than the synapse. To scale the synapses properly, I first checked if the node I ray-casted onto was the soma. I accomplished this by

checking for a soma ID located in a node list by passing in our node index. If the node index does happen to be the soma's ID, I made sure that the synapse prefab was not larger than the soma. Additionally, I made it so the prefab does not transform its position inside the soma. This was done by using hit.point on the synapse prefab. Next, for placement on a regular node index I set the neuron as a parent of the transform. Mainly, this was done to achieve correct transformation at the local scale rather than the global scale. After achieving the correct position of the synapse, the scale must be at a reasonable size. To do this, I utilized a method called visual inflation which is a field that stores inflation data on the neuron. I then obtained a scaling value which contains the current node radius and also a height of the average dendrite radius. Once both float numbers were known I could scale our synapse by following this code,

```
float RadiusLength =
Math.max(radiusScale, heighScale) *
Visual Inflation;
```

RadiusLength is the new size of the synapse. Since the presynaptic and postsynaptic neurons are spheres, I had to scale them equally in all directions to maintain their spherical shape. At this point, a transformation localScale is called and set equal to a Vector3 which contains x, y, and z. Each axis was set equal to RadiusLength, which I solved for earlier.

2.2 Data Structure

How did I store multiple objects that all act as one? The synapse model as described before has a presynaptic and postsynaptic sphere that are actually two different game objects. When a user places the synapse, the presynaptic and postsynaptic neurons are connected by a list of synapses. The list stores the synapses in consecutive order (from presynaptic to postsynaptic) for each synapse. An example of when I used this method was during the placement of the arrow. For each synapse, the arrow is positioned at the presynaptic neuron. To do this, the method must access the list of synapses, which looks like SynapseList[currentSynapse]. When calling the list of synapses it was important to remember that SynapseList[currentSynapse] would be the

presynaptic and SynapseList[currentSynapse+1] would be the postsynaptic neuron. After knowing what synapse to create an arrow for, I then obtained the transform locations of both the presynaptic and postsynaptic neurons. Then, using the following code:

```
Vector3.Lerp(presynaptic.location,
postsynaptic.location, 0.5f);
```

the arrow is placed between the neurons. To continue, a high priority for me was to create a good data structure and methods for the synapse that others could easily use and could be further implemented into. Knowing this, I needed to create a synapse class that could initialize values and store them in each individual instance of the synapse. The class I used consisted of a constructor which defines values such as node index, voltage, and the object's prefab (i.e. what the object looks like). This allowed for easy manipulation and access to information. In summary, the whole data structure for how synapses are stored and accessed goes as follows.. First, when the user ray-casts for the first time the code will create the presynaptic neuron, define the values it holds such as voltage and node index, and add it to the synapse list. Once the user ray-casts again for the second time, the code follows the steps described above but instead adds the postsynaptic neuron and sets a variable "count" equal to zero. The "count" variable ensured that there was no possible way for a user to place a presynaptic sphere and another presynaptic sphere without first placing the postsynaptic sphere.

2.3 Ray-casting and Placement Implementation

I would like to expand more on the different ray-casting options for the placement of the synapse. Normally the simulation is run on a default ray-casting mode. This mode sends a normal ray-cast out which directly applies voltage to the neuron. The way I implemented a different ray-casting toggle followed similarly to the way clamps and plots work. The method goes as follows: the user triggers an event, such as pressing the button on the whiteboard to activate synapses, which enables a script with new ray-casting defaults. The script I created is called vertex snap and it creates a hit event listener which listens for user input when it is enabled. At this point the default ray-casting mode was disabled

and the synapse placement mode was enabled. I created it so when the user decides to switch back to normal ray-casting mode the vertex snap script first calls a disable method. This method will destroy the synapse ray-casting and return the hit events back to the original ray-casting mode. If the user decides they want to place either the presynaptic or postsynaptic neuron, the hit event is called and goes as follows:

```
hitEvent.onPress((hit) =>
presynapticPlacement(hit)).
```

The listener is waiting for a user to either left-click on the mouse or use the trigger on the oculus controller. This method also ties into how I made the option to delete synapses. The user must hold a ray-cast onto the selected presynaptic/postsynaptic neuron for X amount of frames. The frames can be defined within Unity. In short, the algorithm has a threshold integer that must be passed before the destroy methods are called. Moreover, the synapse ray-casting mode will call its own defined methods, but will still cast to one point on the neuron. That means I can gather information about the exact vertex the user has ray-casted onto. I made the script store this information under each instance of the synapse which is called the node index. The node index is where the synapse lies upon the neuron. In detail, the code to achieve this looked like,

```
 NodeIndex =
curSimulation.GetNearestPoint(hit)
```

“curSimulation” is a reference to the current active simulation, whereas GetNearestPoint(hit) is a method that returns the node index that was raycasted onto. Furthermore, now that I stored the node index for each individual presynaptic and postsynaptic neuron, I could then apply voltage. The algorithm that I wrote can be summarized in a few steps. First, it gathers a list of all node index voltages and sees if the presynaptic node index is contained in that list.

```
Double[] curVoltage = Get1DValues();

Synapses[i].voltage =
curVoltage[synapses[i].nodeIndex];
```

Next, the algorithm separates the presynaptic and postsynaptic neurons into their own respective list.

```
for(int i = 0; i < synapses.count;

i++){
```

```
        if(i % 2 != 0){
           Synapse curPreSynaptic =
           synapses[i - 1];
           Synapse curPostSynapse =
           synapses[i];
       postSynapse.Add(curPostSynapse);
       Double[] curVoltage =
curPreSynaptic.attachedSim.Get1DValues
();
curPreSynaptic.voltage =
curVoltage[curPreSynaptic.nodeIndex];
preSynapse.add(curPreSynapse);
        }
}
```

After this is known, the algorithm makes a tuple with the synaptic information.

```
for(int i = 0; i < postSynapse.Count;
i++){
        new1DValues[i] = new Tuple<int,
double>
        (postSynapse[i].nodeIndex,
preSynapse[i].voltage);
}
Set1DValues(new1DValues);
```

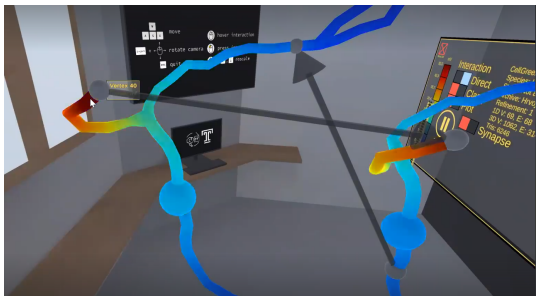
Ultimately, when the voltage was attained at the presynaptic neuron it simply copied over to the postsynaptic neuron. In the future, I would like to implement an accurate mathematical model that resembles real synaptic function more closely.

2.4 Merging Branches Together

One last topic I would like to discuss is my current undertaking to conclude the semester. This involves the merging of all the different branches. It is vital that these branches are merged together in a way that the code does not conflict. This was especially important when merging my synapse branch with the development branch since everything needs to work in unison. I can choose to merge using github directly or git. Most times this can be done automatically, however, there are some instances where the code will conflict and I will have to manually fix it.

3. Results

The results shown in this section are the complete works of my synapse model. Each problem presented has been remedied with a solution. In the final works we can see:



- Multiple neurons working simultaneously and communicating to one another via a synapse.
- An established synapse data structure that sends correct information to the correct neuron.
- A working visual model with a disguising arrow that points the presynaptic to the postsynaptic neuron.

4. Conclusion/Discussion

My semester's research has been an incredible, rewarding, and very challenging experience. I learned a great deal of Neuroscience, strengthened my coding, and was able to further gain knowledge in Unity and Virtual Reality. The work I accomplished ranged from creating an emulator script that utilized the desktops keyboard and mouse layout of Neuro-VISOR, to modeling synaptic functions. I made the synapse framework as modular as possible so future researchers can easily build upon it. In the future, I would like to continue my research into synapse modeling and high performance computations within Unity. I also would like to add a highlight feature which is similar to clamps for user friendliness and easier understanding. In addition, investigation into a more efficient and neater data structure to hold all the synapses would be greatly beneficial.

5. Acknowledgements and Suggestions

Thank you to everyone who has contributed to my research experience and gave me the opportunity to further myself in my academic career. Seeing that this is an ongoing project, I would recommend a few tips to new researchers in my position. First and foremost, I recommend looking at the preexisting code that has been already made. A good metaphor to describe what I mean by this is, do not be afraid to stand on the shoulders of giants! You do not need to make everything from the ground up when you have access to similar methods that can be slightly altered. To continue, remember to document everything you write and keep your code clean. As mentioned above, you will need to learn to work with others by writing code that other people can comprehend. Most importantly, I recommend not stressing out! Asking questions is crucial in a lab setting, especially on such a mind-provoking project. Rather than wasting time being confused, discuss the topic with your mentors and peers and take every task step by step. Enjoy the research process!