Neocortical Virtual Robot

Thomas J. Kelly Thomas J. Rushton
Yantao Shen Sergiu M. Dascalu Frederick C. Harris, Jr.

Computer Science and Engineering
University of Nevada, Reno
Reno, NV, USA
fred.harris@cse.unr.edu

Abstract

The NCS (NeoCortical Simulator) is a neural network simulator capable of simulating small brains in real time. The NeoCortival Virtual Robot framework allows researchers to build virtual worlds that NCS simulated brains are able to interact with. This is done by supplying scientists with a domain-specific language for interaction, as well as abstractions for environment creation.

Keywords: Computational Neuroscience, Interactive Visualization, Virtual Neurorobotics, Simulation.

1 Introduction

The brain is the most powerful computer in the world. For many years people have been attempting to figure out how it exactly it works. Recently, neurologists have worked on figuring out how the brain functions by observing brains with imaging technologies and electrical probes, yet these have their limits. Imaging cannot resolve features at the scale of the building blocks of the brain, neurons and synapses, while probes are limited by the mechanical difficulties of fitting more than a few wires into the brain. Kapoor, et al. demonstrate the latter case, where they present the implantation of six probes into the temporal lobe of a macaque as a major accomplishment [17]. Computational neuroscience, which studies the brain by simulating it, avoids these problems. If one has an accurate simulation of a brain, then it is possible to pause the simulation and observe every attribute of every cell in the simulated brain. In order to create such a simulation, one could develop an exact biophysical model of a neuron from first principles. However, simulating every chemical reaction has a downside because it is rather slow. Therefore, computational neuroscientists have developed approximations of neuron behavior that take considerably less time to run, such as the leaky integrate-and-fire [18] and Izhikevich [14] models.

Brains are extremely parallel in nature with every neuron acting independently. Thus, brain simulation is well suited for parallel computation using graphics processing units (GPUs). In recent years, GPUs have become flexible, allowing them to be programmed to perform more than just graphical tasks. This general-purpose computation on GPUs (GPGPU) allows us to work on diverse workloads, excelling at highly parallel ones. The NeoCortical Simulator (NCS) is able to simulate neuron models with GPGPU computing to allow for the simulation of a million neurons connected by 100 million synapses in real-time [13]. This ablility to simulate many neurons allows for the study of large scale neural behavior with great detail. It is even possible to simulate simple brains.

The rest of this paper presents the design and implementation of the Neocortical Virtual Robot. Section 2 gives a brief background on the field of Virtual Neurorobotics. An overview of the project is discussed in Section 3. The experimental components are described in Section 4, followed by the implementation of the system in Section 5. Finally, Section 6 presents a conclusion on the Neocortical Virtual Robot, as well as areas of potential future work.

2 Virtual Neurorobotics

In nature, there is no such thing as a brain without a body. Since animals generally learn by interacting with their environment, it has been suspected that the problem of creating artificial intelligence could be solved by giving an AI a body, allowing the AI to explore its environment. Instead of using an actual robot, using an avatar in a simulated environment allows more easily controlled experiments in environments that would be cost-prohibitive in the real world.

Goodman, *et al.* [10] further hypothesized that in

order for intelligence to occur, it is necessary for the intelligence to be motivated by emotional drives and learn by interacting with humans with a virtual body. In this framework, it is possible to simulate learning as it occurs in humans, in a social setting.

Previously, the Webots simulation environment built by Cyberbotics [8]. In one experiment[5], Bray, *et al.* simulated the mechanisms of trust in structures of the hypothalamus and amygdala to simulate how trust worked in a basic social setting. The robot would either wave its arm vertically or horizontally, then the experimenter would do the same. If the experimenter moved their arm in the same way, this would build trust, as illustrated in Figure 1.

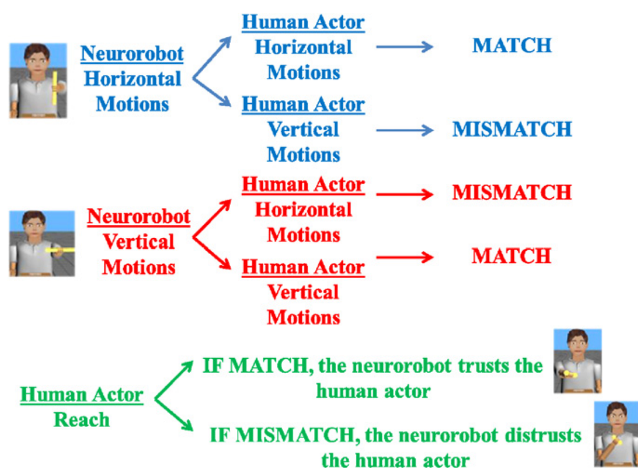

Figure 1: Diagram of "Real-time human-robot interaction underlying neurorobotic trust and intent recognition" [5]

This work was inspired by previous work on what would become "Goal-related navigation of a neuromorphic virtual robot" [6], which was built off of Bray's doctoral dissertation [4], wherein she sliced and examined a mouse's brain to build a detailed model of several regions of that brain. This model was then able to be run through a simulated maze. In a later paper [6], this was made visual, by using the mouse model to drive a virtual robot through a neighborhood.

The issue with all of Bray's work is that the only way to view the internal mechanisms of the simulated brain is by looking at tables of numbers of various parameters of the brain. These tables are able to be graphed, but richer visualization techniques such as the tool created by Jones, *et al.*, show the simulated neural network in 3D, allowing for a deeper understanding of the simulated brain's structure.[16].

3 Project Overview

At the Brain Computation Lab we have been creating a browser-based interface to interact with NCS [13]. It consists of three components: the NeoCortical Builder, the NeoCortical Repository and Reports, and the NeoCortical Virtual Robot. The NeoCortical Builder (NCB) allows researchers to build brain models and connect simulation input and output parameters [3]. While Bray used a domain-specific language to construct the brains in her work, the Model Builder makes model creation much easier by supplying the user with an intuitive visual interface.

3.1 The webapp interface

In order to add a neuron or synapse in NCB, you can simply find the appropriate menu option, instead of needing to learn a brain specification language [15].

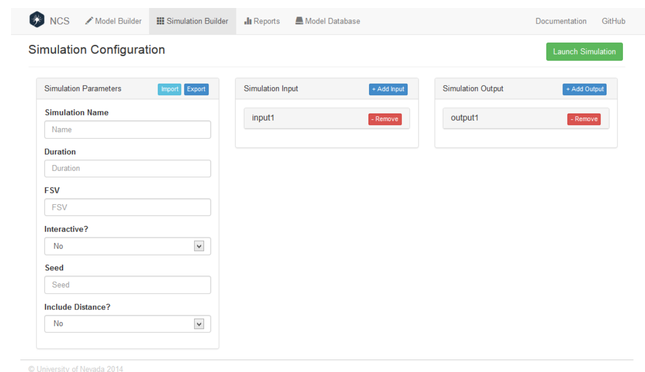

Figure 2: The NCB simulation builder interface

As can be seen in Figure 2, the Simulation Builder is used to set parameters for simulations and launch the simulations.

NeoCortical Repository and Reports (NCR) makes it easy to store and select different brain models to use and visualize the activity of simulated neural networks [2]. The Repository Service can be used to save and load brain models, allowing researchers to share data between projects. It is also possible to load multiple models, representing different portions of the brain, and connect them together, allowing for reuse of brain model components. A Reporting Interface is then used to display outputs from NCS in the form of graphs, allowing researchers the ability to quickly examine large amounts of data.

3.2 Virtual Robot

The NeoCortical Virtual Robot (NCVR) is a WebGL environment that integrates into our NCS web interface as its own tab. It gives neuroscience researchers a sandbox to create environments and scenarios where NCS-simulated brains can be observed in realistic circumstances. The environments are able to be built quite easily in SketchUp. The robot behavior can be scripted in JavaScript, as JavaScript is flexible enough that it can be adopted to a wide variety of experiments.

4 Components of an Experiment

4.1 Virtual Environment

SketchUp is a simple 3D modeling tool developed by Trimble Navigation [19]. It is optimized for architectural modeling and allows one to quickly build environments at human scale. SketchUp is natively compatible with Windows and Mac OS X.

By default, all objects composing the 3D environment have collision volumes generated using a point-based tessellation algorithm based on the work of Fei, *et al.* [9]. These collision volumes can be concave, convex, or even have holes. If one wants an object to be portable, which the robot can pick up and move around, then one simply puts the word “portable” into the name of the object. Portable objects’ bounding volumes are created using a single axis-aligned bounding box (AABB). These AABBs are able be rotated as they are stored as oriented bounding boxes (OBBs).

The specific file type accepted by NCVR from SketchUp is the Google Earth KMZ format. This file type is convenient in that it includes textures in the file. The open COLLADA model format is also supported, allowing researchers to build environments with other 3D modeling tools. However, unlike the KMZ files created by SketchUp, COLLADA files must be uploaded alongside any desired textures.

4.2 Controller Scripts

A controller script is a single JavaScript file that can be uploaded to the server. At its core, the controller script acts as a finite state machine and code is executed for the robot depending on the current state. This execution occurs right after each frame is rendered and all sensory data has been calculated. In each state’s code, one may do arbitrary calculations based on sensory data, instruct the robot to perform an action, and change the state for the next time step.

```
1  var count = 0;
2
3  function startState() {
4    // After 300 timesteps, change to
         secondState
5    if (count > 300) {
6      next('secondState');
7    }
8    // Count the timesteps
9    count += 1;
10 }
11
12 function secondState() {
13   // Start walking at 1 m/s
14   setSpeed(1);
15 }
```

Figure 3: An example of controller script syntax

Any controller script file consists of one or more top-level functions, as seen in Figure 3. The initial state is the first function in the script file, which is `startState` in the example. In Figure 3, the given script waits for 300 timesteps, which is typically between 5 and 10 seconds, then makes the robot walk forward at 1 meter per second.

API:

All of the sensors are accessible through a global `sensors` object, which has many properties corresponding to all sorts of stimuli that the robot can receive. In addition to stimuli, the sensors object also has an array of outputs from NCS in the property `brainOutputs`. Sensor data that are difficult to directly parse, like images from the camera, have functions that allow users to work with the data in a more intuitive fashion, like letting one check the hue of given location in the camera’s view.

There are two kinds of actions that the script can tell the robot to do: instant actions and extended actions. Instant actions, like changing walking speed with `setSpeed`, as seen in Figure 3, trigger and complete immediately. Extended actions, like going forward a certain distance with `goForward`, take a while. In order to make it easier to reason about extended actions, we added the ability to switch states after the completion of an extended action. Calls to perform extended actions return `ExtendedAction` objects, which have several methods that allow users to set their properties using the builder pattern. For example, `goForward( 5 ).over( 3 ).then( 'secondState' )` will go forward 5 meters over 3 seconds, then change the state to `secondState`.

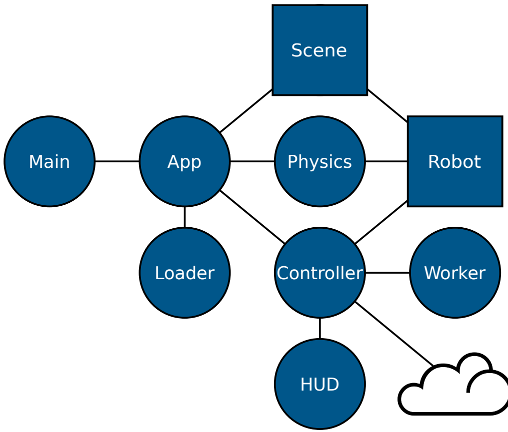

Figure 4: How the various modules are connected. The cloud represents the connection to the simulation, outside of the NCVR application.

5 Implementation

The internals of NCVR are divided into several modules, connected as seen in Figure 4. The main module sets up the initial configuration UI, connecting NCVR to the rest of the web interface. From this UI, a user has the ability to start a simulation and load a level and a script. The level, script, and simulation URL are sent to the app module, which then starts the rest of the modules. The level data is formatted into a scene by the loader module, which renders with the Three.js 3D library [7]. The level data is also sent to the physics engine, which builds a representation of the physical properties of the level. The app places the robot object into the scene. Once the level and robot have been created, the app sends the script and simulation URL to the controller module. The controller module relays the script to the worker module, which executes it in an isolated process. At this point, everything has been loaded, so the app module tells the scene to render itself and the physics module to simulate physics. After performing the calculations of a single frame, the data goes to the app, where it is stored until it is retrieved by the controller. The controller connects to the simulation at the URL it has received, which means that the daemon will send brain outputs to the controller regularly. The controller also gathers data from the robot and the app. The combined brain outputs and robot sensor data are sent to the worker, which executes one step of the script on the data, after which it may respond with an action sent to the controller. The controller will send this action to the robot, which will change the physical state of the robot and how the robot is rendered in the scene.

Figure 5: The Carl model from Mixamo, Inc. [1]

5.1 Collision detection

This module holds the oriented bounding box (`OBB`) object factories, methods, and utility functions. The factories allow for the creation of OBBs fit to vertices (`OBB.fromPoints`), from an axis-aligned bounding box (`OBB.fromBox3`), and from box parameters (`new OBB`). `OBB.fromPoints` uses a covariance method of fitting OBBs. It uses all of the vertices in an object, rather than weighted triangles of the convex hull, as Gottschalk recommends [11].

There are also several methods for `OBB` objects. `OBB.calcVertices` simply calculates the vertices of an `OBB`. `OBB.makeMatrix` calculates a matrix that transforms a unit cube centered at the origin into the shape and location of the bounding box represented by the `OBB` instance the method is called on. `OBB.testGround`, `OBB.testOBB`, and `OBB.testTri` are all functions that test whether an `OBB` is penetrating the ground, another `OBB`, or a triangle. All of them return an object containing the properties `contactNormal` and `penetration`, which are then used in collision resolution.

5.2 Robot

This object wraps the behavior of the robot, abstracting the process of moving the robot around. That way, users don't have to worry about, for example, animating each leg as the robot walks or the kinematics of arm motion. The program is using the Mixamo Corporation's [1] Carl model for our robot, shown in Figure 5.

In order for a hand to grasp an object, the hand must first move through space. It is straightforward to find the location of a hand from the angles of the joints in the arm using basic trigonometry, but the reverse is not true. The assumption is that an arm has seven degrees of freedom: three at the shoulder, two at the elbow, and two at the wrist. However, the pose of the hand can

be completely specified using six degrees of freedom: three for the position and three for the orientation. Due to this difference in the number of variables, there are potentially an infinite number of solutions to the question of which arm configurations yield a given hand pose. Furthermore, exact solutions of the problem are slow to run and complex to implement, so we needed an approximate solution.

It starts with the fact that the joints need to be limited. Not only can this prevent impossible arm positions, like bending the elbow backwards, it can also prevent the arms from penetrating other parts of the arms or the body of the robot. While this does restrain the robot, it prevents the need to do costly collision detection. This restriction keeps the configuration vector, a 7-dimensional real vector, in a 7-cuboid. These limits are hard-coded to work with the Carl model.

So, the problem boils down to finding a path in 7-space from the current configuration of the arm to a configuration that is as close as possible to the desired pose of the arm. We used an algorithm based upon Weghe's JT-RRT algorithm [20] with some modifications. The algorithm runs as the arm is moving and we made compromises to make the algorithm run faster. The largest difference is that there is no connectivity graph of valid positions. There is actually no method used to take into account collisions, which seems like it would be suboptimal, but appears to have worked relatively well in practice.

5.3 Motion planning

The motion planning algorithm described in the last section requires a few primitives: hand pose distance, stepping the arm configuration toward the target using the Jacobian, calculating an internal representation of the arm configuration, and converting such internal representations into a form usable by the 3D engine. This module puts all of the details of the the internal representation of the arm joints into one place. This is because all of these functions are generated from the equations of arm motion using Sympy, instead of being written by a human.

The hand location is calculated from the product of eight matrices: one for the transform from the torso to the shoulder, three for the shoulder joint, two for the elbow, and two more for the wrist. The last seven are parameterized by the configuration space parameters. This product, which we will call $R_{0,7}\left(\vec{\theta}\right)$, which includes matrices 0 to 7, transforms us from a space centered at the torso of the model to one centered at the hand, oriented to the hand. $\vec{\theta}$ represents the configuration vector, which is just a series of angles in radians. Since each joint has a matrix that is the product of these matrices, the shoulder matrix is $R_{0,3}\left(\vec{\theta}\right)$, the elbow matrix is $R_{4,5}\left(\vec{\theta}\right)$, and the wrist matrix is $R_{6,7}\left(\vec{\theta}\right)$.

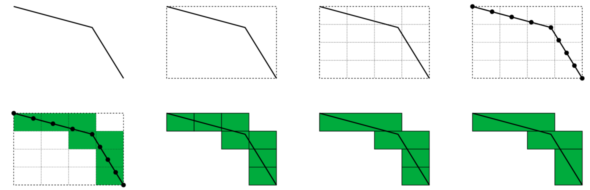

Figure 6: The steps used to build bounding boxes for static objects. This only uses two dimensions, while the actual algorithm works in three dimensions

5.4 Physics

The physics module is primarily responsible for collision resolution and bounding volume generation. Due to how our collision detection algorithm works, it does not know where two collision volumes are colliding. However, it does know the axis of collision and the penetration depth. This module also accounts for the fact that objects fall by giving the ground object its own collision volume covering the space at and below the ground itself.

Collision detection is done by using OBBs as physical proxies, as detecting collisions between boxes is simpler than on more complex shapes. Collision volume generation for static objects is done with point-tessellated voxelization [9]. The following steps are illustrated in Figure 6. For every object, it divides a box fit to the model into smaller boxes with less than 10 cm to a side. It then subdivides the constituent triangles into smaller triangles until they are smaller than 10 cm in every dimension, guaranteeing that at least one vertex will be in every box that intersects with the original triangle. It then iterates over the vertices of the triangles, marking the box that the vertex is in as occupied and merges consecutive boxes if they have the same dimensions.

Portable objects simply have a single axis-aligned bounding box (AABB) drawn around them. This AABB can then rotate when the object is moved. For the robot, the central mass, consisting of the legs, torso and head, uses an AABB covering the vertices of these body parts. The arm collision volumes are made by finding the vertices of the robot model that are influenced by each arm bone and building `OBB`s with `OBB.fromPoints`, creating well fit boundaries around the arms.

5.5 Other Modules

Main:

This module loads up the other modules and integrates NCVR into the webapp, taking parameters from the initial configuration screen and starting the simulation.

App:

This module actually starts all of the other components, setting up the 3D engine, loading the environment, and starting the event loop. When loading the environment, it initializes the robot, the camera, tell the physics engine about all of the objects, as well as if they are portable or dynamic. It also fixes textures, so that it can work with models that supply incorrectly sized textures. When running the event loop, it needs to calculate the physics and logic and render the environment on every frame. It also copies the robot's front-facing camera to a buffer, so it can be used by the controller for the camera sensor.

Controller:

This starts up a Web Worker, which runs in another process, sandboxing the controller script given by a researcher. On every frame, it calls the controller to package up sensor data to send to the worker, as well as the heads-up display. Eventually, the worker will respond with commands. It translates these commands into actions. Some actions run instantly, while those that are not instant are queued and, on every frame, a single step of that action is run. If an extended action had data on something that should be run upon completion, it sends that back to the worker.

This module also handles communication with the NCS daemon, sending sensor data to the server and receiving brain outputs from the server, over a WebSocket. These are relayed to the worker and the heads-up display.

Worker:

This provides a sandboxed environment for the controller scripts to run in. Every time sensor data is received, it is unpacked and a single step in the state machine is run. This also provides the API that the script uses to communicate actions to the controller.

6 Conclusion and Future Work

6.1 Conclusion

In the course of creating this, several problems needed to be solved. These can be divided into creating an interactive environment and making the robot interact with it. The primary contribution for the former is the method used to turn what amounts to a 3D drawing of an environment into a world for the robot to explore, while still being able to run in real time, detailed in Section 5.4. This real time aspect is crucial to enabling experimenters to socially interact with the robot, as Goodman recommends for allowing robots to learn like people do [10]. To let the robot interact with the environment, we created a domain-specific language (DSL) for researchers to specify robot behavior, so we can abstract away robot hand-eye coordination. Of the behaviors we've abstracted away, the most difficult was also one of the most basic ways we have for affecting our environment, the action of picking up an object, which we detailed in Section 5.2.

All of this was built in JavaScript to make it fit into the web interface. This is an accomplishment in itself because existing code for 3D game development and robot control are sparse, which forced us to construct solutions to problems that have been solved elsewhere. JavaScript did make it easier to develop the DSL, since JavaScript allows for runtime evaluation of code. By using JavaScript syntax, researchers get the full power of a general-purpose programming language.

6.2 Future Work

There is an obvious future work potential on integrating with NCS. The sensors, as emitted by NCVR, will need to be processed before becoming usable by the brain simulation. Given the probable complexity of processing, which may use several computer vision techniques with the robot's camera, sensor processing will potentially require entire programs.

We would like to have multiple robots. NCVR assumes that there is only one robot and only one brain simulation, but the ability to model several robots with different brains would allow for a wide variety of new experimentation possibilities.

The physics engine should be replaced with one that has been better tested and faster, such as Cannon.js [12]. One obstruction to this is that the current physics engine uses positions directly from the models used by the graphics engine. These need to be separated before a new physics engine is used.

Improving grasping to actually grasp objects, would make picking up objects look much more realistic. Making sure that the hand actually touched the object would also partially solve this problem.

Acknowledgements

This material is based in part upon work supported by: The National Science Foundation under grant

number(s) IIA-1329469. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mixamo, inc., 2015. Accessed: 2015-05-14.

[2] Edson O. Almachar, Alexander M. Falconi, Katie A. Gilgen, Nathan M. Jordan, Devyani Tanna, Roger a novel V. Hoang, Sergiu M. Dascalu, Laurence C Jayet Bray, and Frederick C. Harris Jr. Design and implementation of a repository service and reporting interface for the ncs. In *Proc. Software Engineering and Data Engineering*, New Orleans, Louisiana, October 2014.

[3] Jakub Berlinski, Marlon D. Chavez, Cameron Rowe, Nathan M. Jordan, Devyani Tanna, Roger V. Hoang, Sergiu M. Dascalu, Laurence C Jayet Bray, and Frederick C. Harris Jr. Neocortical builder: A web based front end for ncs. In *Proc. Computer Applications in Industry and Engineering*, New Orleans, Louisiana, October 2014.

[4] Laurence C Jayet Bray. *A Circuit-Level Model of Hippocampal, Entorhinal and Prefrontal Dynamics Underlying Rodent Maze Navigational Learning.* PhD thesis, University of Nevada, Reno, 2010.

[5] Laurence C Jayet Bray, Sridhar R Anumandla, Corey M Thibeault, Roger V Hoang, Philip H Goodman, Sergiu M Dascalu, Bobby D Bryant, and Frederick C Harris Jr. Real-time human–robot interaction underlying neurorobotic trust and intent recognition. *Neural Networks*, 32:130–137, 2012.

[6] Laurence C Jayet Bray, Emily R Barker, Gareth B Ferneyhough, Roger V Hoang, Bobby D Bryant, Sergiu M Dascalu, and Frederick C Harris Jr. Goal-related navigation of a neuromorphic virtual robot. *BMC Neuroscience*, 13(Suppl 1):O3, 2012.

[7] Ricardo Cabello. Three.js. `https://github.com/mrdoob/three.js`, 2015. Accessed: 2015-05-14.

[8] Cyberbotics. Webots: Robot simulator. Accessed: 2015-05-14.

[9] Yun Fei, Bin Wang, and Jiating Chen. Point-tessellated voxelization. In *Proceedings of Graphics Interface 2012*, GI '12, pages 9–18, Toronto, Ont., Canada, Canada, 2012. Canadian Information Processing Society.

[10] Philip H Goodman, Quan Zou, and Sergiu-Mihai Dascalu. Framework and implications of virtual neurorobotics. *Frontiers in neuroscience*, 2(1):123, 2008.

[11] Stefan Gottschalk. *Collision queries using oriented bounding boxes.* PhD thesis, The University of North Carolina at Chapel Hill, 2000.

[12] Stephen Hedman. Cannon.js, 2015. Accessed: 2015-05-14.

[13] Roger V Hoang, Devyani Tanna, Laurence C Jayet Bray, Sergiu M Dascalu, and Frederick C Harris Jr. A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling. *Frontiers in neuroinformatics*, 7, 2013.

[14] Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[15] Christine Johnson. A centralized service for accessing the ncs brain simulator through a web interface. Master's thesis, University of Nevada, Reno.

[16] Alexander Jones, Justin Cardoza, Denver Liu, Laurence Jayet Bray, Sergiu Dascalu, Sushil Louis, and Frederick Harris. A novel 3d visualization tool for large-scale neural networks. *BMC Neuroscience*, 14(Suppl 1):P158, 2013.

[17] Vishal Kapoor, Eduard Krampe, Axel Klug, Nikos K. Logothetis, and Theofanis I. Panagiotaropoulos. Development of tube tetrodes and a multi-tetrode drive for deep structure electrophysiological recordings in the macaque brain. *Journal of Neuroscience Methods*, 216(1):43 – 48, 2013.

[18] T. Trappenberg. *Fundamentals of Computational Neuroscience.* Fundamentals of Computational Neuroscience. OUP Oxford, 2010.

[19] Trimble Navigation. Sketchup, 2015. Accessed: 2015-05-14.

[20] M. Vande Weghe, Dave Ferguson, and S.S. Srinivasa. Randomized path planning for redundant manipulators without inverse kinematics. In *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*, pages 477–482, Nov 2007.