University of Nevada, Reno

NEOCORTICAL VIRTUAL ROBOT: A FRAMEWORK TO ALLOW SIMULATED BRAINS TO INTERACT WITH A VIRTUAL REALITY ENVIRONMENT

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of Master of Science in

Computer Science and Engineering

by

Thomas J. Kelly

Dr. Frederick C. Harris, Jr. / Thesis Advisor

May 2015

UNIVERSITY OF NEVADA RENO **THE GRADUATE SCHOOL**

We recommend that the thesis prepared
under our supervision by

THOMAS J. KELLY

entitled

NEOCORTICAL VIRTUAL ROBOT: A FRAMEWORK TO ALLOW SIMULATED BRAINS TO INTERACT WITH A VIRTUAL REALITY ENVIRONMENT

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph.D. – Advisor

Sergiu M. Dascalu, Ph.D. – Committee Member

Yantao Shen, Ph. D. – Graduate School Representative

David Zeh, Ph.D. – Dean, Graduate School

May 2015

ABSTRACT

The NCS (NeoCortical Simulator) is a neural network simulator that can simulate small brains in real time. Since brains learn by interacting with the world, we created the NeoCortical Virtual Robot framework, which allows researchers to build virtual worlds for simulated brains to interact with, by supplying scientists with a domain-specific language for interaction and abstractions for environment creation.

ACKNOWLEDGEMENTS

I am grateful to Dr. Fred Harris for the support throughout the work and research of my thesis. Thank you Dr. Sergiu Dascalu and Dr. Yantao Shen for being on my committee. Special thanks are given to Christine Johnson for her help with the NCS web interface architecture. I'd like to thank Richard Kelley for his advice on robot arm actuation research. I'm also thankful for my parents' support throughout my educational career.

CONTENTS

LIST OF FIGURES

CHAPTER 1

MOTIVATION AND BACKGROUND

1.1 NeoCortical Simulator (NCS)

The brain is the most powerful computer in the world. People have been attempting to figure out how it works through many methods for hundreds of years. In recent years, neurologists have worked on figuring out how the brain works 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 [19]. 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 [21] and Izhikevich [17] models. Even so, a single CPU can only simulate a small number of neurons. Parallel computing, where multiple processors compute simultaneously, is often used in computational neuroscience to run multiple groups of neurons simultaneously.

Brains are extremely parallel in nature with every neuron acting independently.

Thus, brain simulation is well suited for parallel computation. Graphics processing units (GPUs) are built to perform extremely parallel computations, like computing millions of pixels to display on a screen simultaneously. In recent years, GPUs have become more 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. Since version 6, the NeoCortical Simulator (NCS) has simulated approximate neuron models with GPGPU computing to allow for the simulation of a million neurons connected by 100 million synapses in real-time [16]. By being able to simulate many neurons, it allows for the study of large scale neural behavior with great detail. It is even possible to simulate simple brains.

1.2 Virtual Neurorobotics

In nature, there is no such thing as a brain without a body. Since animals generally learn by using their bodies to interact with their environment, it has been suspected that the problem of creating artificial intelligence could be solved by giving an AI a body with which it can explore its environment. However, using an actual physical body requires setting up a controlled environment with a robot in order to get reproducible experiments. 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.

The search for artificial intelligence has also led to the study of natural intelligence, since the proven mechanisms of cognition in nature can provide inspiration for simulated cognition. By uniting a simulated brain with a virtual robot,

researchers can simulate the learning process as found in nature, allowing better insight into cognition.

Goodman, *et al.* 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 [12]. In this framework, it is possible to simulate learning as it occurs in humans, in a social setting. To this end, several experiments were conducted by the Brain Computation Lab at the University of Nevada, Reno.

Previously, the researchers of the UNR Brain Computation Lab used Webots, a robot simulation environment built by Cyberbotics [10]. In one such experiment [6], 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 1.1. In order for the robot to see the arm motion, the robot viewed the experimenter using a webcam. This proved that an artificial intelligence could trust in a manner similar to that of a human.

In another experiment [8], they simulated a social learning scenario where the experimenter held up a card and the robot would point to the right or to the left. The experimenter would then grade the robot vocally to teach it which direction to associate with which color, using emotional intonation to cue the robot as to its success. A happy intonation would induce a positive response and an angry intonation would induce a negative response. This process is shown in Figure 1.2. They were able to successfully teach the robot to select the correct side, modeling the neural process of learning.

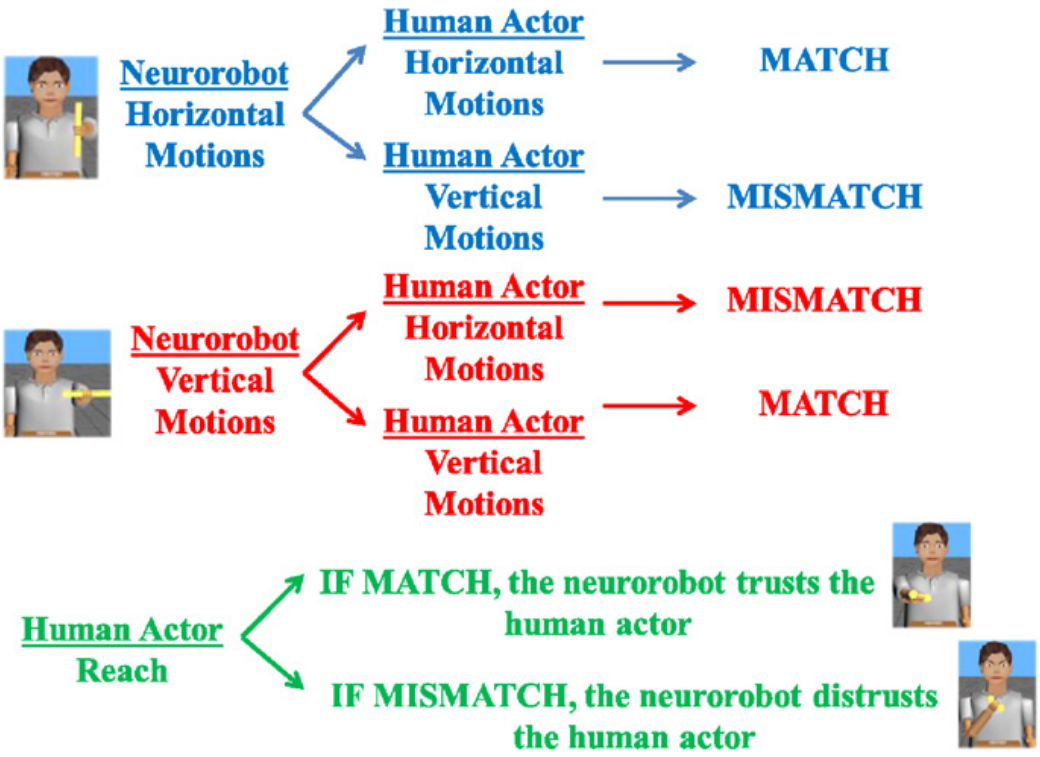

Figure 1.1: Diagram of "Real-time human-robot interaction underlying neurorobotic trust and intent recognition" [6]

Finally, this work was inspired by previous work on what would become "Goal-related navigation of a neuromorphic virtual robot" [7], which was built on work done in Bray's doctoral thesis [5], wherein she sliced and examined a mouse brain to build a detailed model of several regions of that brain, which was then used to run through a simulated maze. In a later paper [7], this was made visual, by using the mouse model to drive a virtual robot through a neighborhood.

In all of Bray's work, the only way to understand the internal mechanisms of the simulated brains was by looking at series of numbers of various parameters of the brain. These series were graphed, but richer visualizations could be achieved with a specially built tool that understood the structure of the simulated brain. Such a visualization came from the work of Jones, *et al.* [18], who created a tool

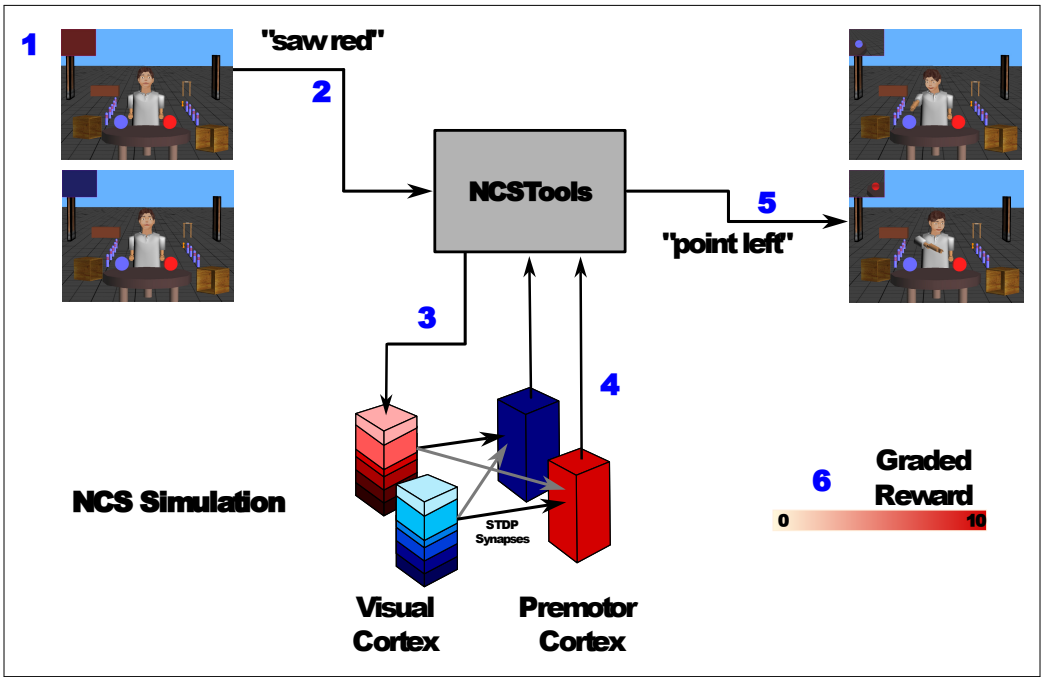

Figure 1.2: Diagram of "Reward-based learning for virtual neurorobotics through emotional speech processing" [8]

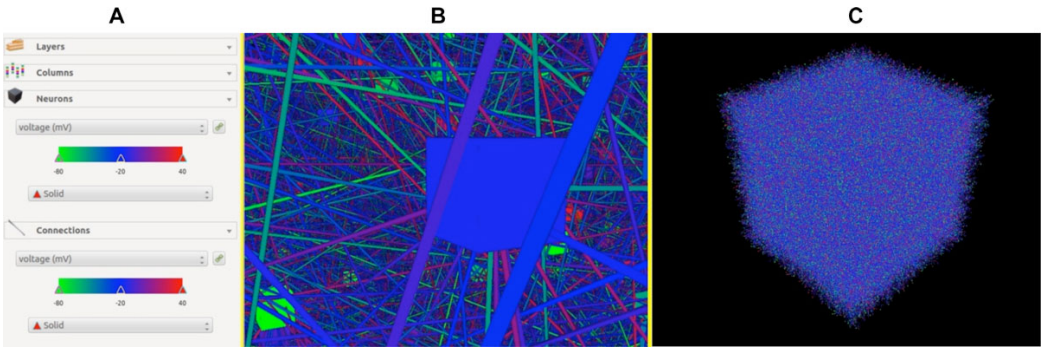

Figure 1.3: A 3D visualization of an NCS neural network [18]

to show the simulated neural network in 3D, making it easier for researchers see large scale behaviors in simulated brains, shown in Figure 1.3.

In our work on "Goal-related navigation of a neuromorphic virtual robot" [7], we had many difficulties with Webots. The software is intended to simulate the natural problems that robots have, like a tendency for the robot to not travel in a

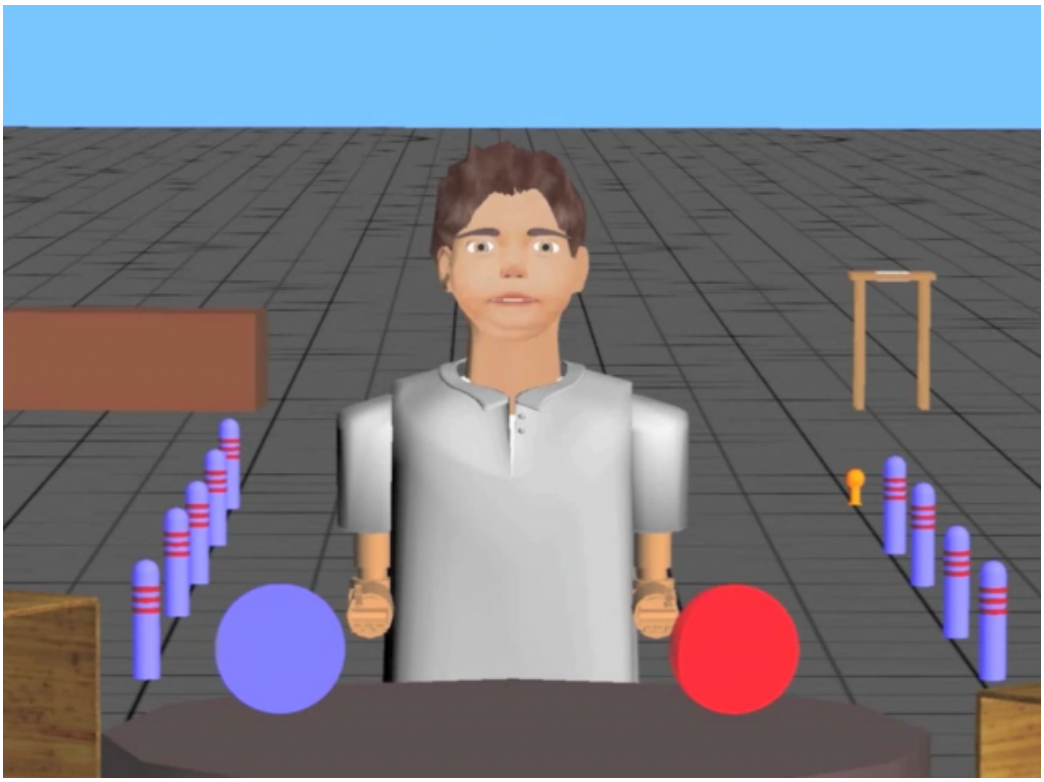

Figure 1.4: Webots childbot [8]

perfectly straight line, which meant that we had to expend time on compensating for these unnecessary simulated mechanical imperfections. Additionally, Webots is expensive software, in part because it supports these unneeded features. The software could only be installed on a single computer, which was inconvenient for development. Furthermore, the childbot model used fell squarely into the uncanny valley, as can be seen in Figure 1.4. Finally, it doesn't really fit into the new interface that we at the UNR Brain Computation Lab have been working on.

We at the Brain Computation Lab have been creating a browser-based interface to interact with NCS. 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

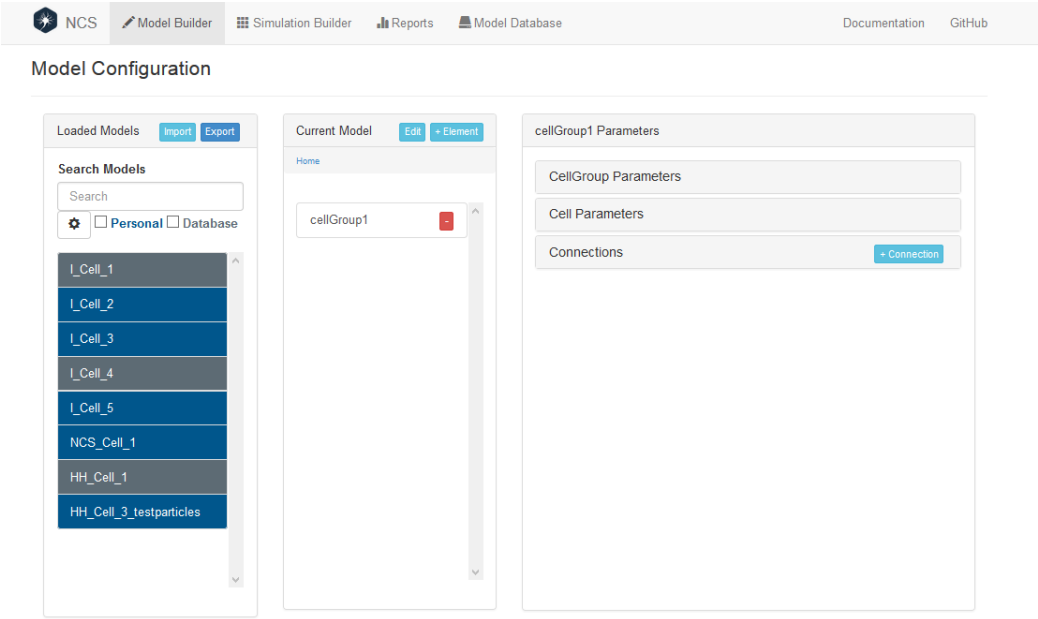

Figure 1.5: The NCB Model Builder interface

simulation input and output parameters [4]. While Bray used a domain-specific language to construct the brains in her work, the Model Builder makes model creation much easier with its intuitive visual interface, as shown in Figure 1.5.

1.3 The new webapp interface

In NCB, in order to add a neuron or synapse, you can simply find the appropriate menu option, instead of needing to learn a brain specification language. As can be seen in Figure 1.6, the Simulation Builder is used to set parameters for simulations and launch the simulations. These parameters include how long the simulation should run, inputs to the simulation, how output should be stored, as well as many others.

NeoCortical Repository and Reports (NCR) makes it easy to store and select

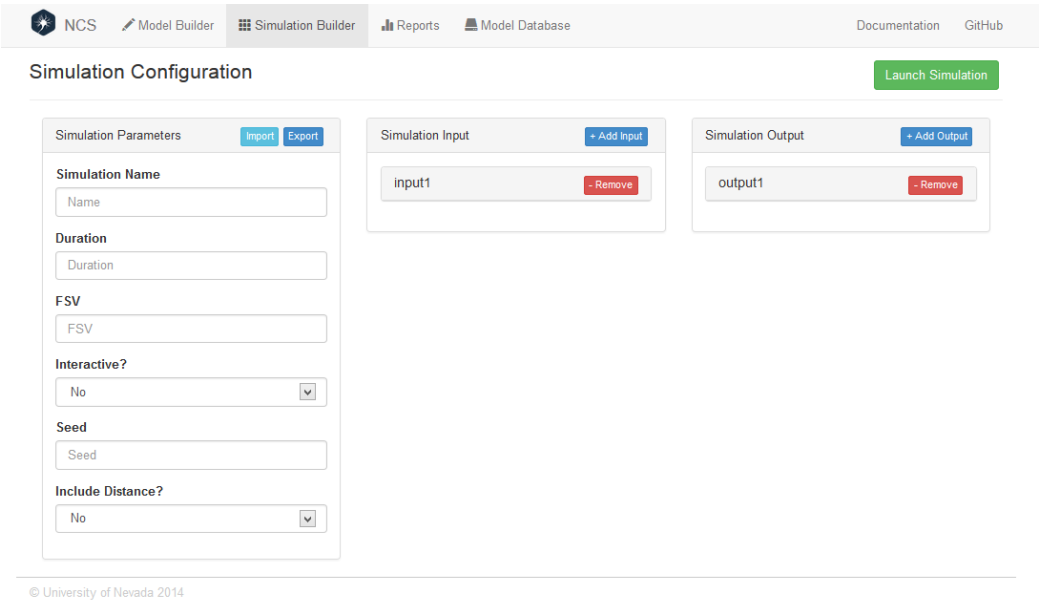

Figure 1.6: The NCB simulation builder interface

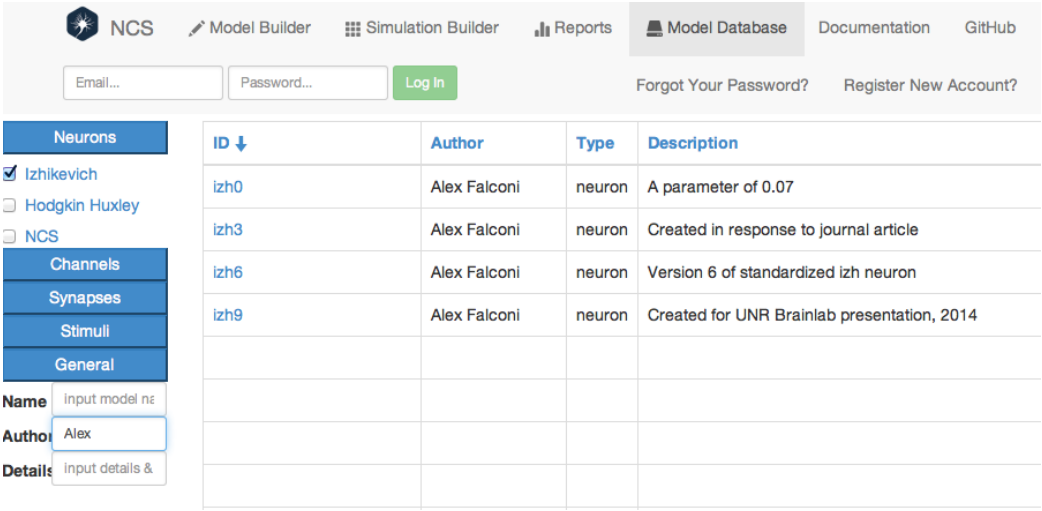

Figure 1.7: The NCR Repository Service

different brain models to use and visualize the activity of simulated neural networks [3]. The Repository Service, shown in Figure 1.7, can be used to save and load built brain models, allowing researchers to share data between projects. It is even possible to load multiple models, representing different portions of the

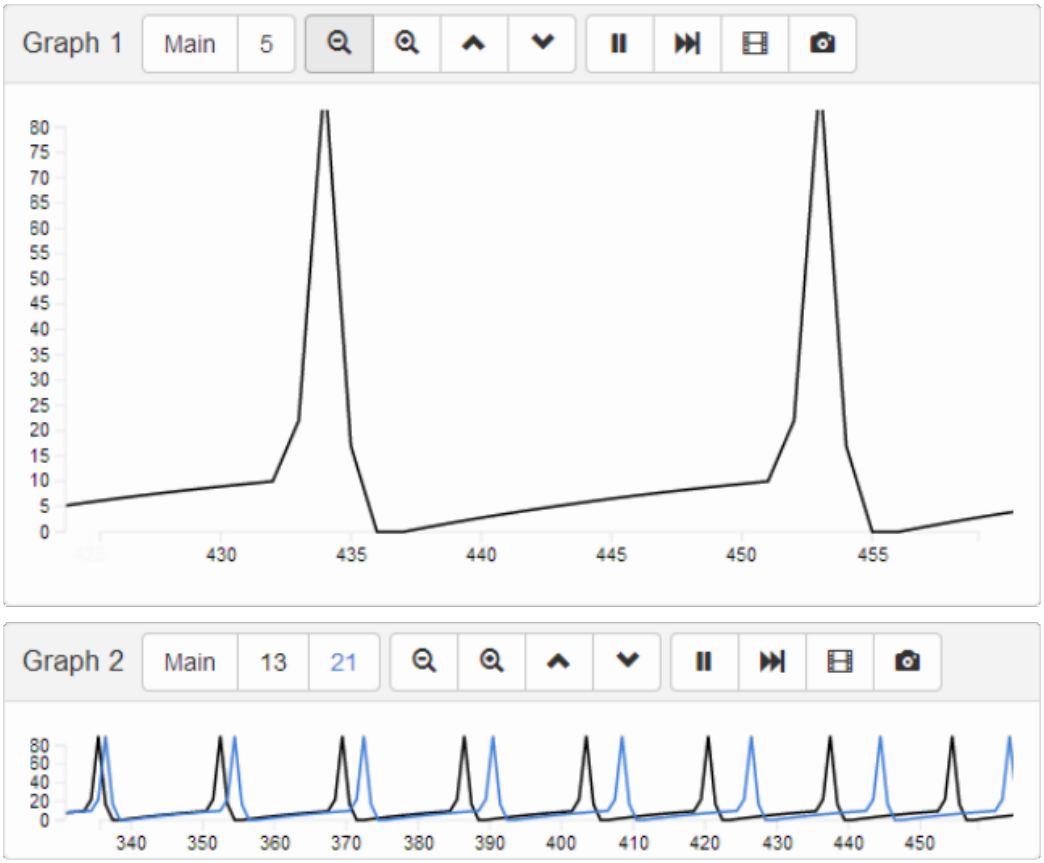

Figure 1.8: The NCR Reporting Interface

brain, and connect them together, allowing for reuse of brain model components. Figure 1.8 displays the Reporting Interface, which can graph outputs from NCS. This allows researchers to quickly examine large amounts of data, which can lead to discoveries that are not obvious when viewing a small amount of data at a time.

These two components are split into four tabs in the interface. Alongside these, we created the third component, the NeoCortical Virtual Robot,

1.4 Virtual Robot

The NeoCortical Virtual Robot (NCVR) is a WebGL environment that integrates into our new 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 require little expertise by allowing one to build the environment in SketchUp, which requires little training to use. Then, the robot behavior can be scripted in JavaScript. This scripting environment is flexible enough that it can be adopted to a wide variety of experiments.

We're going to walk through the a quick explanation of the what one needs to do make an environment with NCVR. Then, we will walk through a detailed example environment, similar to that seen in "Goal-related navigation of a neuromorphic virtual robot" [7]. This will be followed by a discussion of the internal architecture of NCVR. After that, we explain issues that were encountered throughout the development of NCVR. This leads to an explanation of potential areas of future work. Finally, there are appendices for reference by users of the scripting engine.

CHAPTER 2
NEOCORTICAL VIRTUAL ROBOT

2.1 Making an environment

SketchUp is an easy-to-use 3D modeling tool developed by Trimble Navigation [23]. It is optimized for architectural modeling and allows one to quickly build environments at human scale. There are two versions of SketchUp: the free version, SketchUp Make, works just fine for our purposes; and SketchUp Pro, which adds CAD software compatibility and features for presentation of environments built with the software, neither of which users of NCVR need. SketchUp is compatible with Windows and Mac OS X. It also works with Linux via the WINE compatibility layer. Albeit, one must supply the argument `/DisableRubyAPI`, disabling Ruby plug-ins in SketchUp.

NCVR has been optimized for ease of use, so users need not worry about placing collision volumes when creating environments. By default, all objects have collision volumes generated using the point-based tesselation algorithm detailed in Section 4.2, based on the work of Fei, *et al.* [11], so they can be concave, convex, or even have holes and the researcher need not worry about collision volumes. If one wants a portable object, 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). Even though the volume is created aligned to the axes, the box can still be rotated, as it is stored as an oriented bounding box (OBB).

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, such as Blender, Autodesk 3ds Max, or Autodesk Maya. Unlike, the KMZ files created by SketchUp, COLLADA files must be uploaded alongside any textures, meaning that all textures must be in the same directory as the COLLADA file so that they can all be selected using the file dialog used to set the environment for NCVR.

2.2 Writing a Controller Script

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. There are a finite number of states, specified in the script file. Every state is accompanied by a bit of code that is executed when the robot is in that 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.

2.2.1 Syntax

The controller script file consists of one or more top-level functions, as seen in Figure 2.1. The initial state is the first function in the script file, which is `startState` in the example. The name is unimportant, as long as the function is first. The contents of each of the functions is executed on each time step. In Figure 2.1, the given script waits for 300 timesteps, which is typically between 5 and 10 seconds,

```
1  var count;
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 2.1: An example of controller script syntax

depending on the speed of the computer, then makes the robot walk forward at 1 meter per second.

JavaScript's syntax will be relatively familiar to anyone who has used a C-based language, but the actual semantics of JavaScript are unlike most other languages. In addition to our states, which are also functions, we declare and initialize a variable `count`, which we use to track the number of timesteps that have passed. We have attempted to make scripting easy by making it so that users do not have to worry about more advanced features of JavaScript, like variable hoisting, context scope, and prototypical inheritance, but a sufficiently complex script may need these.

2.2.2 API

All of the sensors are accessible through a global `sensors` object, which is explained in in Appendix A. This object 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 2.1, 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` when that is done. Appendix B has a complete list of the available actions.

CHAPTER 3
A DETAILED EXAMPLE

3.1 Building an environment

Here we discuss how to make an environment where the robot can navigate a maze, pick up an object, and move it somewhere else.

When you start up SketchUp [23], you should see a screen like Figure 3.1. Open the Template tab, select "Simple Template - Meters", and click the "Start using SketchUp" button. This should bring up the modeling inteface, shown in Figure 3.2. First, delete the cardboard cutout person by clicking on it and pressing Delete. We're going to start by making a grassy ground plane. Click on the Shape→Rectangle button on the toolbar, as shown in Figure 3.3. Drag a rectangle from a point behind the origin to a point in front of the origin, making a horizontal plane centered on the origin, like that seen in Figure 3.4. Click on the scale button, shown in Figure 3.5. Eight anchor points, like those seen in Figure 3.6, should appear on the plane. Zoom out by holding ctrl and scrolling out. This gives us room to enlarge the rectangle. While holding the ctrl key, click and drag a corner anchor point to make the plane bigger, as shown in Figure 3.7. You may want to repeat this process of zooming out and scaling a couple of times to make the plane large enough that the horizon appears to be nearly flat from the robot's point of view in the center. Click the Paint Bucket icon, shown in Figure 3.8. This should bring up the Materials window, shown in Figure 3.9. Open the Vegetation folder and select a grass texture in that window. Then, click on the ground plane with the Paint Bucket. SketchUp may not reflect these changes immediately, but zooming in or out is a fast way to force it to update.

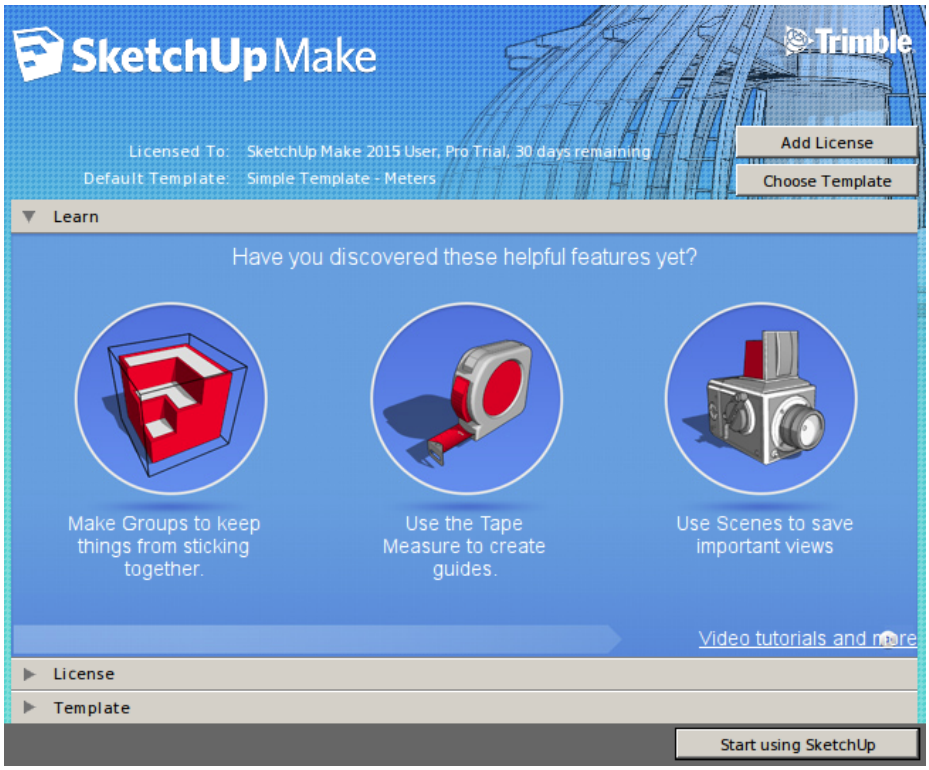

Figure 3.1: SketchUp after starting up

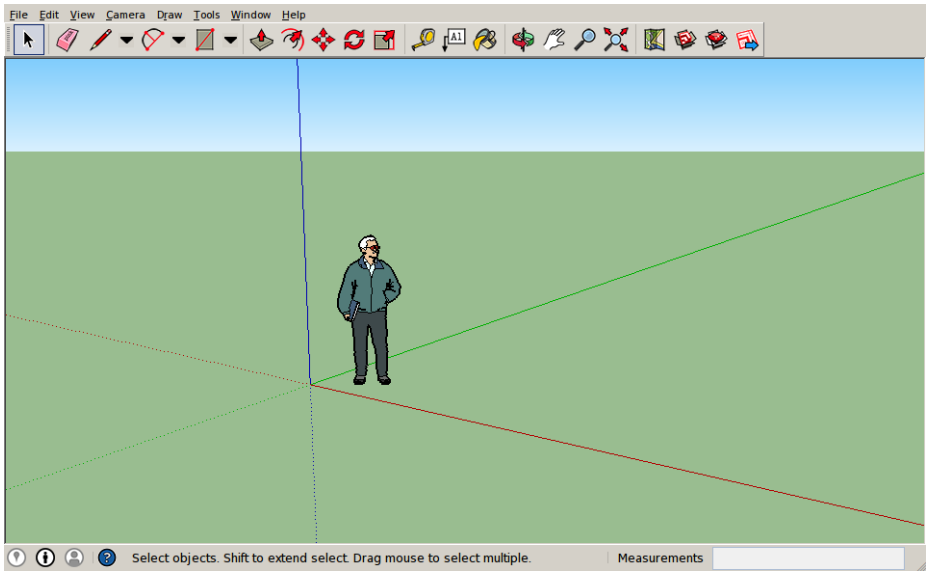

Figure 3.2: SketchUp modeling interface

Figure 3.3: The rectangle tool

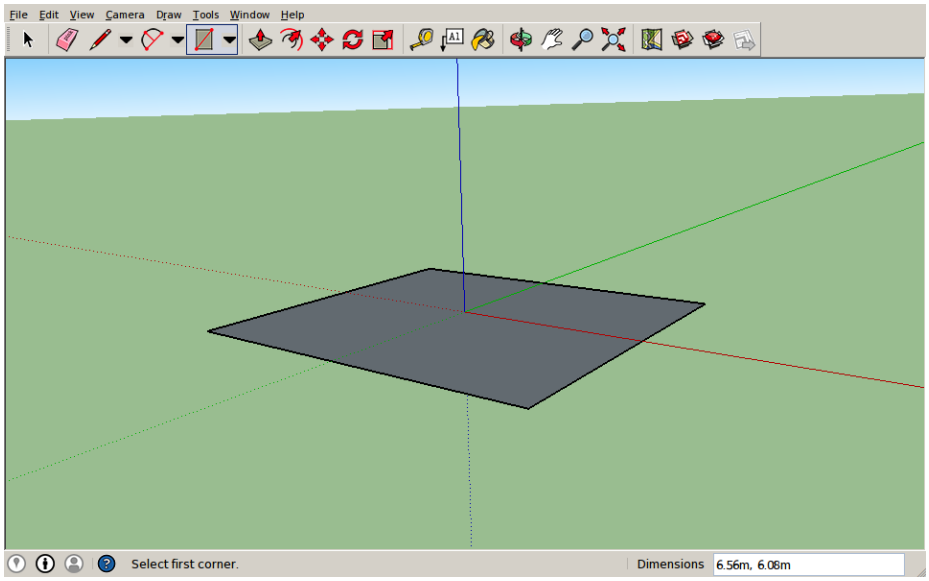

Figure 3.4: A rectangle around the origin in SketchUp

Next, we're going to make a road maze, like the design shown in Figure 3.10. To make it easier to get the dimensions correct, in the Window→Model Info window, set "Enable length snapping" to 1 m. Select the Line tool, which is shown in Figure 3.11. Click on the ground plane at one of the vertices of the planned design. Click at each of the vertices of the design in order. In order to maintain right angles, SketchUp will lock lines to be aligned with the axes and color them like the parallel axis. For alignment, SketchUp will sometimes show a dotted axis-aligned line and lock to it. Once the lines are drawn, use the Paint Bucket tool to fill in the road lines with an asphalt texture. The result should look like Figure 3.12.

Figure 3.5: The scale tool

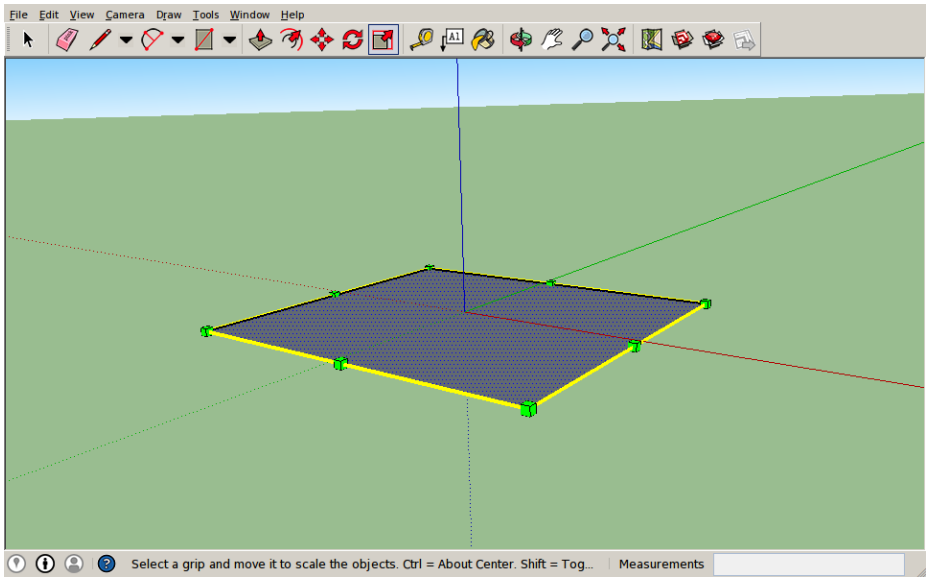

Figure 3.6: Scaling in SketchUp

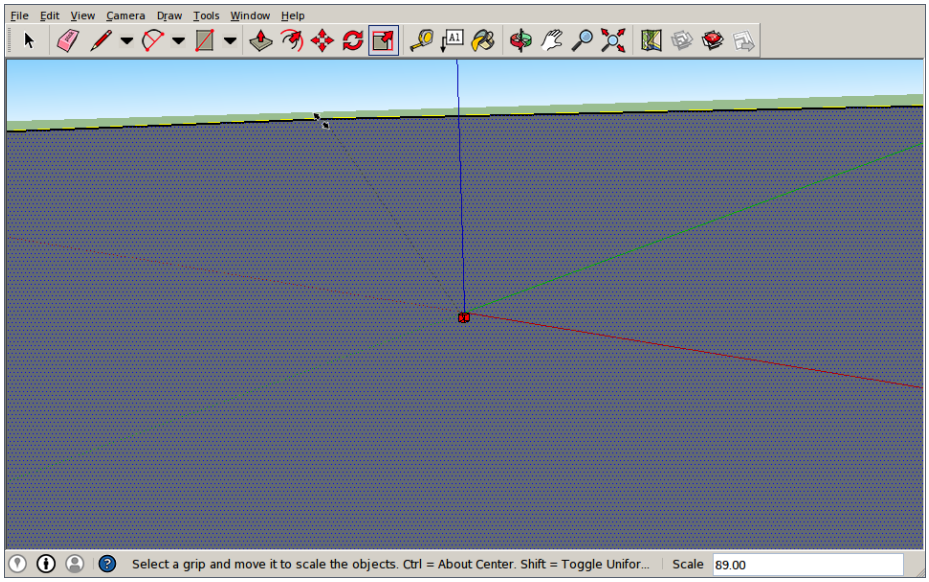

Figure 3.7: The results of scaling in SketchUp

Figure 3.8: The paint bucket tool

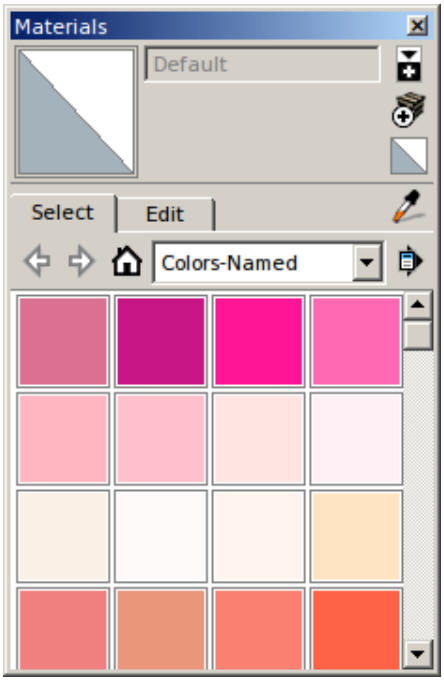

Figure 3.9: SketchUp's Materials window

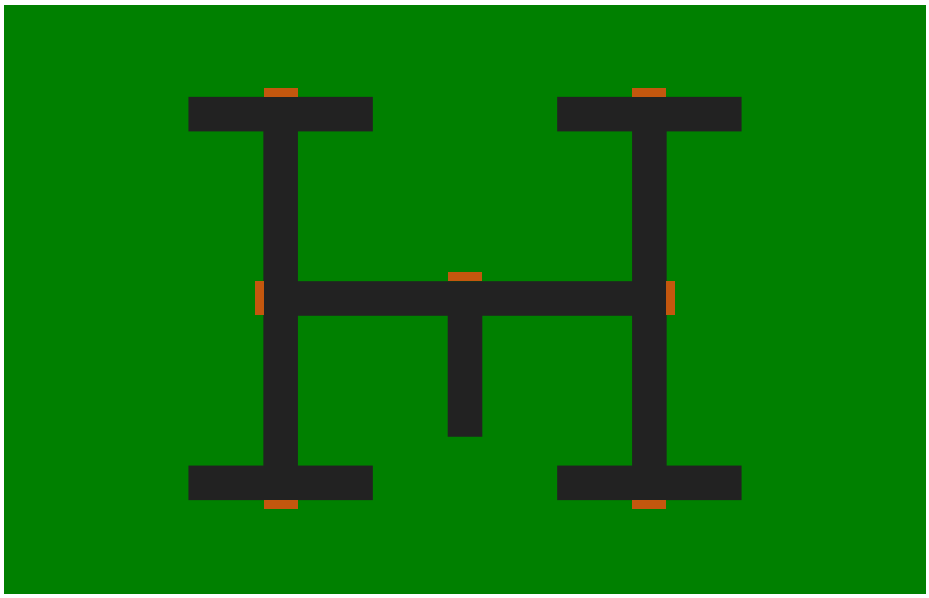

Figure 3.10: Road maze design

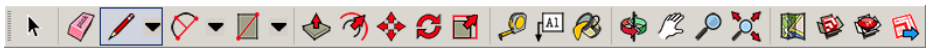

Figure 3.11: The line tool

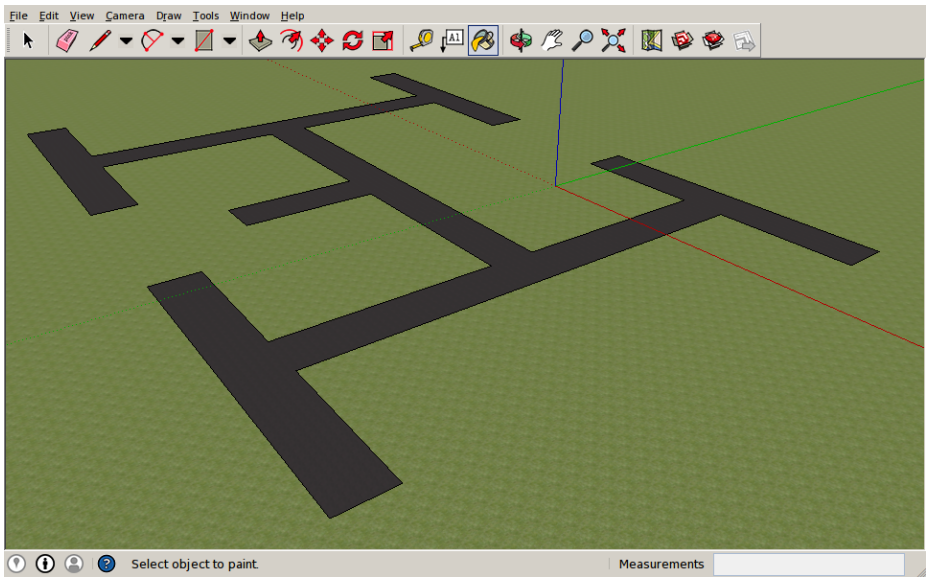

Figure 3.12: The completed roads

Figure 3.13: The Push/Pull tool

While it is possible to make the robot recognize that it has reached an intersection with computer vision, for this demonstration, we're going to instead use walls to stop the robot at each intersection. These walls are shown in red in Figure 3.10. Using the Rectangle tool, draw a rectangle at each T-intersection. Then fill each rectangle with a brick pattern. In order to turn the brick ground into brick walls, use the Push/Pull tool, shown in Figure 3.13. With that tool, it is possible to drag each brick region up, making a wall. The result should look like Figure 3.14.

In order to put a target table, open up the 3D Warehouse website [22] and search for "table." Find a reasonably high table, so the robot can reach the surface from a standing position. Download the table model and load it into SketchUp by going to File→Import and selecting the downloaded model.

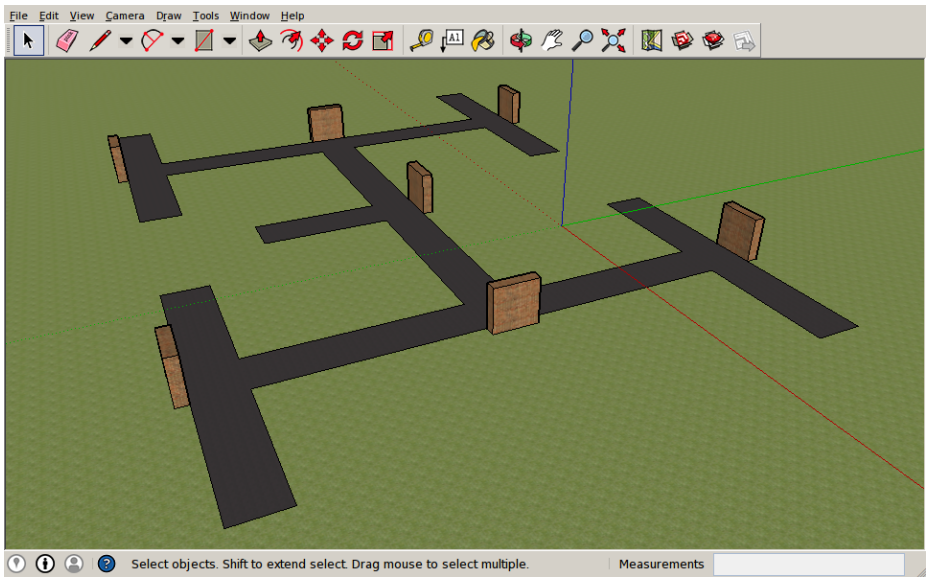

Figure 3.14: After setting up walls

Figure 3.15: The Move tool

Position the table at the end of one of the paths using the Move tool, shown in Figure 3.15, resulting in something like Figure 3.16. Download and import a small object that fits on the table. For this example, we used a bottle. Place it on the table, near the side closest to the path, as seen in Figure 3.17. With the Select tool, shown in Figure 3.18, right click on the small object and open the Entity Info menu option. Change the name field to "portable_thing". The important part is that the name includes the word "portable". This allows the object to be picked up by the robot. Other objects, like the table and the ground plane, act as static scenery. You may add other scenery for fun.

Finally, we need to configure the location of our robot. We start by selecting Tools→3D Text. SketchUp will ask for some text. Put the word "Robot". Place the text at the robot's intended location. The robot will end up facing toward the

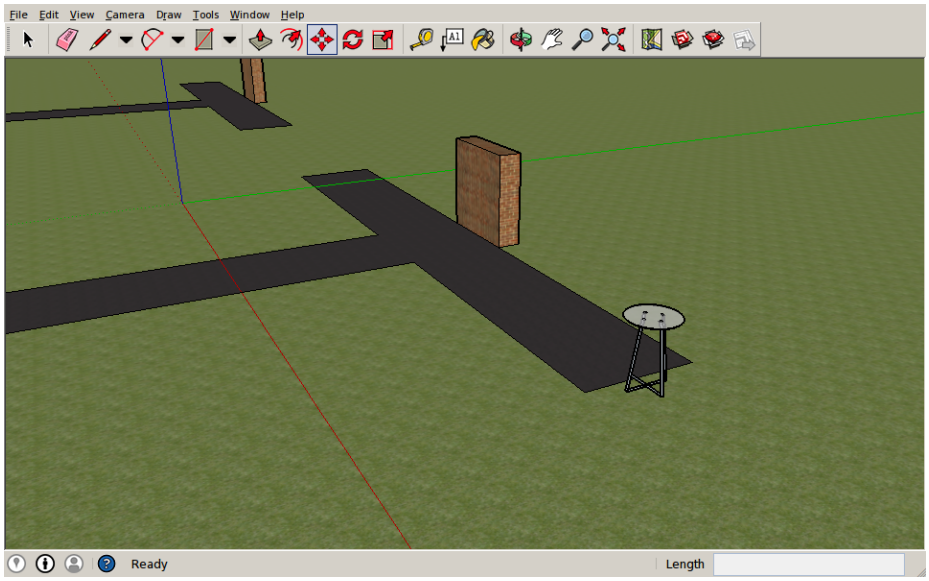

Figure 3.16: How to place the table

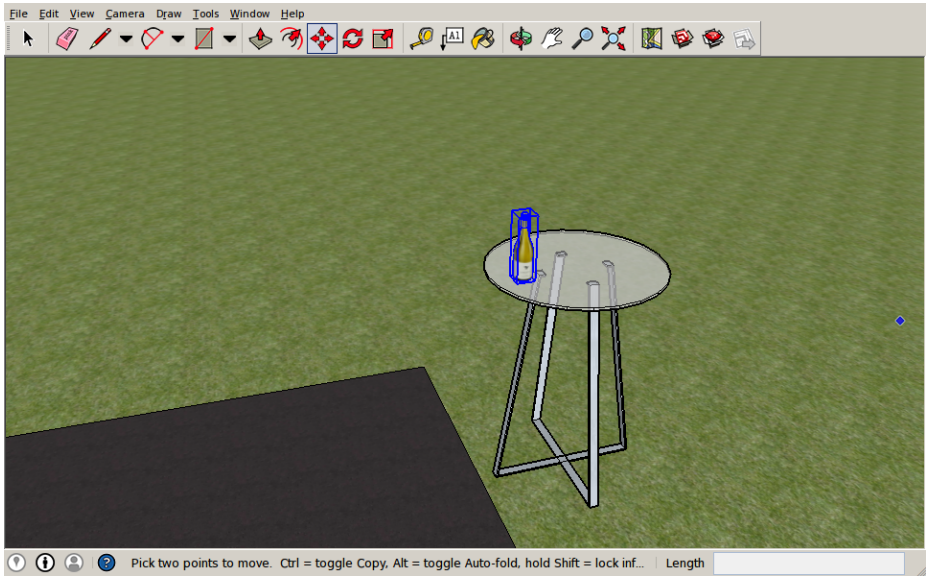

Figure 3.17: How to place the object

Figure 3.18: The Select tool

Figure 3.19: The rotate tool

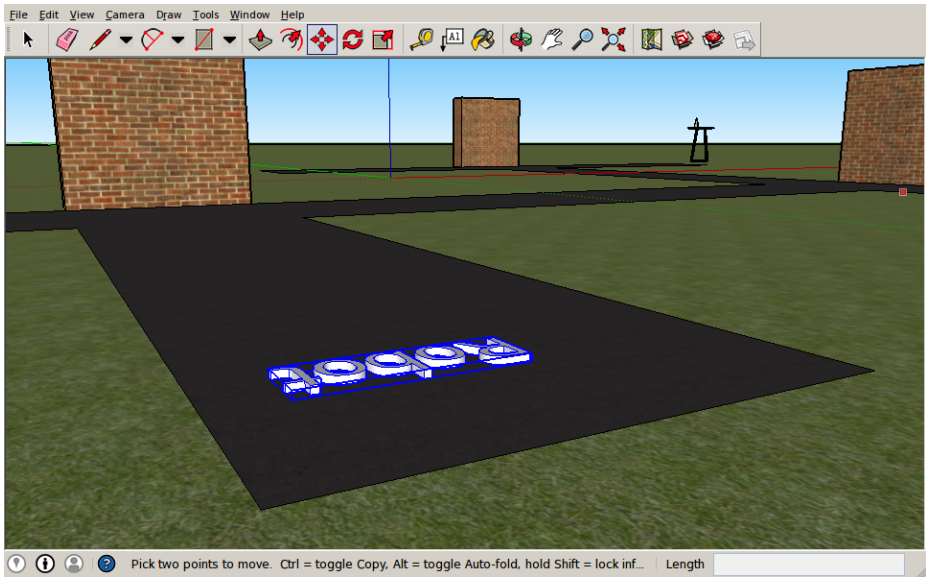

Figure 3.20: The robot placeholder

bottom of the text, positioned at the lower-left corner of the letter "R." The reason that we use 3D text for the robot placeholder is that it is the easiest way to create a model that is named "Robot", while making its orientation clear. Any submodel could be used, but the name has to be exactly the word "Robot", with that capitalization and punctuation. If the default orientation is unwanted, the robot can be reoriented with the Rotate tool, shown in Figure 3.19. Click on a horizontal face of the word "Robot," and then click elsewhere on the same face. Move the mouse to rotate until the placeholder is facing the correct direction. Then, click to fix the orientation. Moving the word around with the Move tool may also be necessary. Our final result can be seen in Figure 3.20. In order to export it, go to File→Export→3D Model. Select Google Earth file (KMZ) for the file type and save it.

3.2 Controller Script

This script will make the robot walk through the maze and pick up the small object we placed on the table.

```
1  // Start by walking forward
2  function startWalking() {
3      // At 1.7m/s
4      setSpeed(1.7);
5      next(walkToWall);
6  }
```

Figure 3.21: The starting state

The starting state, `startWalking` (Figure 3.21), just sets the speed and moves on to the `walkToWall` state.

```
1  // After the robot starts walking,
2  function walkToWall() {
3      // If it has run into the sign,
4      if (sensors.collision.front) {
5          // Stop
6          setSpeed(0);
7          // Back up 1m
8          goBackward(1.0)
9          // At 1.7m/s
10             .at(1.7)
11         // Then it waits for the turn signal
12             .then(waitForAction);
13         next(doNothing);
14     }
15 }
16
17 // A dummy state for when it needs to do nothing.
18 function doNothing() {}
```

Figure 3.22: The walking state and the dummy state

In `walkToWall` (Figure 3.22), on every time step, the script polls for if the robot has run into a wall. If it does run into something, the robot stops and backs up

1 meter. The command to go backward, `goBackward`, is an `ExtendedAction`, which allows it to change to a given state upon completion of the action. However, in order to prevent the robot from going backward twice, the script immediately switches to a dummy state, `doNothing`, which, as can be expected, does nothing.

```
function waitForAction() {
    // If the first output is positive,
    if (sensors.brainOutputs[0] > 0) {
        // Turn left 90 degrees and start walking again
        turnLeft(90)
            .then(startWalking);
        // This prevents it from turning multiple times.
        next(doNothing);
    } else if (sensors.brainOutputs[1] > 0) {
        // Similarly for turning right on the second input.
        turnRight(90)
            .then(startWalking);
        next(doNothing);
    } else if (sensors.brainOutputs[2] > 0) {
        // The third input triggers picking up the object
        goForward(1.0)
            .then(pickItUp);
        next(doNothing);
    }
}
```

Figure 3.23: The state where the robot waits for brain input

After walking backward, the state is set to `waitForAction` (Figure 3.23), which polls the `brainOutput` sensor, so that when the first `brainOutput` signal goes high, the robot turns left and when the second `brainOutput` goes high, it turns right. If the third `brainOutput` goes high, that's a signal to walk up to the table and pick up the object, in state `pickItUp` (Figure 3.24).

```
function pickItUp() {
    reachForAndGrabWithRightHand("portable_thing")
        .then(turnAround);
    next(doNothing);
}
```

Figure 3.24: The state for picking up an object

```
function turnAround() {
    turnRight(180);
    next(doNothing);
}
```

Figure 3.25: The final turning around state

After that has been picked up, the robot turns around 180 degrees in state `turnAround` (Figure 3.25), taking the object with it. With this script, the simulated brain can now perform several actions to interact with its environment. Save this file as a JavaScript file.

3.3 Running the Environment

In order to run the virtual robot, the NCS web interface and the mock NCS daemon need to be setup and running. Each of these include their own installation directions. Once they are both running, go to the NCS web interface and find the Virtual Robot tab.

The Virtual Robot tab, as shown in Figure 3.26, has several parameters. The first section shows the simulation parameters. In the future, these parameters should be handled in other tabs, but for now, since the NCS daemon, the entity that allows

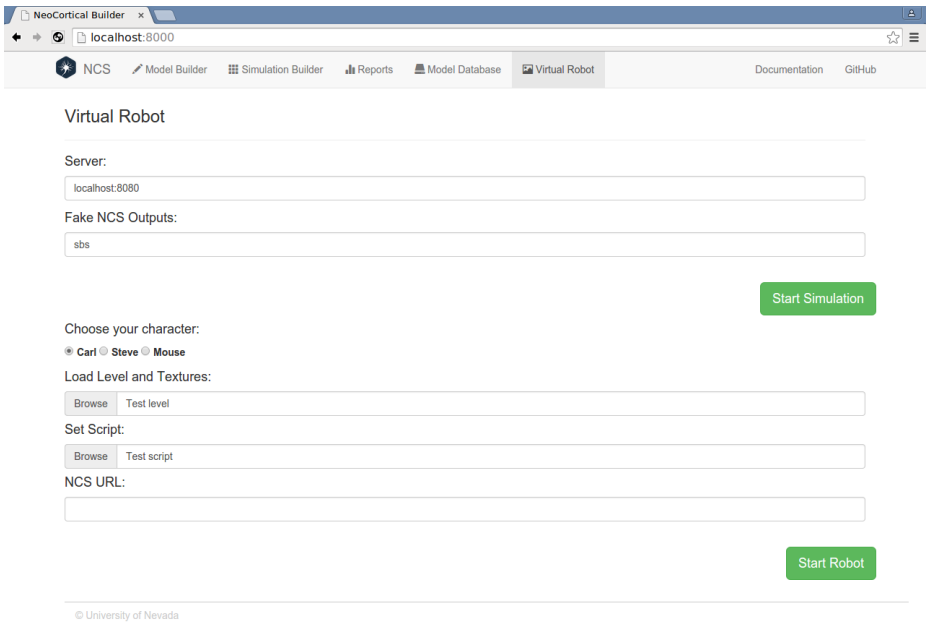

Figure 3.26: Virtual Robot tab

communication with NCS, does not exist yet, the setup for a mock NCS daemon is available.

In this section there is the NCS daemon's location, which is a domain and a port number, such as localhost:8080. Then, there are the NCS outputs. For the mock daemon, this should be a series of lower case S's and B's, for example "sbs". The S's stand for "slider" and the B's stand for "button." So, the string "sbs" would yield a mock daemon UI like that seen in Figure 3.27, with a slider, followed by a button, and then another slider. For this demo, we are going to use three buttons, so the outputs should be set to "bbb". With those two parameters filled in, a simulation can be started with the mock daemon, giving you something like Figure 3.28.

The next parameters are for the Virtual Robot environment. First, a character can be selected. At this point, the options are: Carl, a realistic man; Steve, a pixelated man; and Mouse, a white computer mouse. For this, use Carl. The next

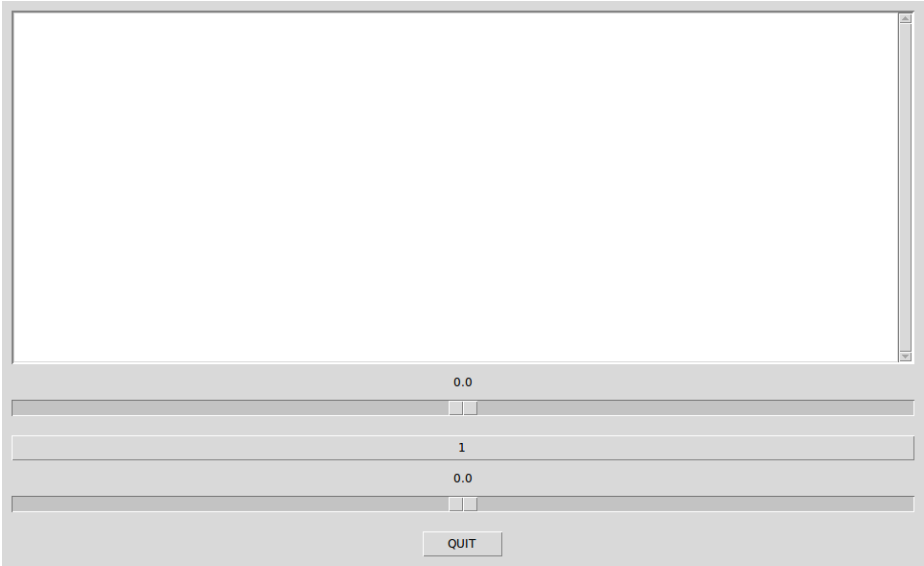

Figure 3.27: The mock daemon user interface

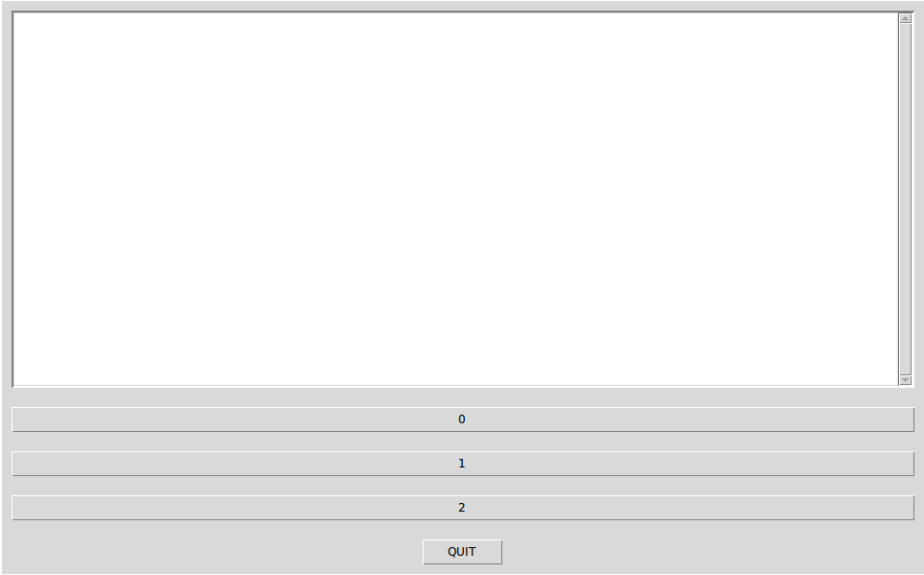

Figure 3.28: The user interface for the example

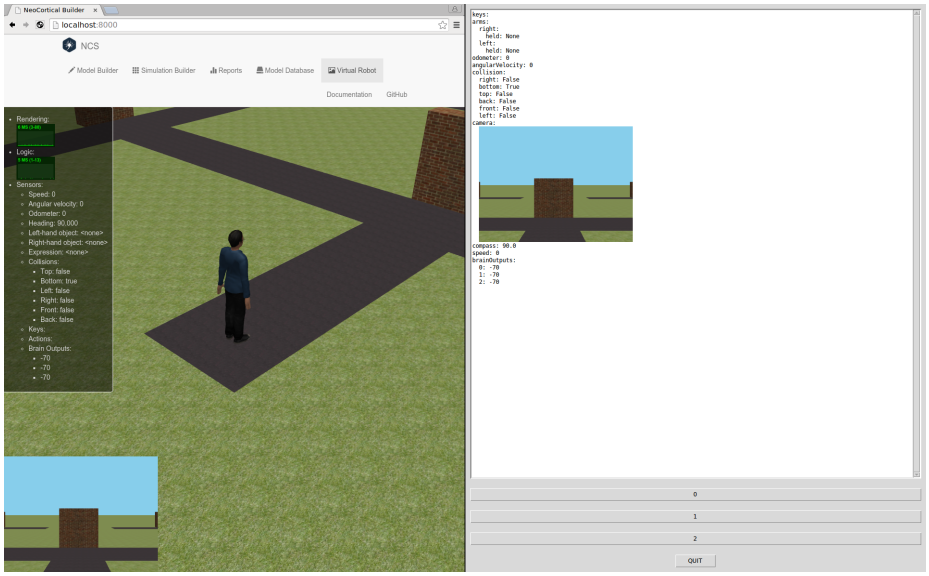

Figure 3.29: The running simulation

option is the level. Select the KMZ file produced in Section 3.1. Next, set the script from Section 3.2. The NCS daemon URL should have been filled out when you started the "simulation." With everything filled out, you can start the robot. The result should look like Figure 3.29.

Once its started, press the enter key to start the script. The robot will start walking until it hits something. To make it turn left or right, click on the 0 or 1 buttons, respectively, in the mock daemon. When it reaches the small object to be picked up, press the 2 button and the robot will pick it up.

CHAPTER 4
IMPLEMENTATION

4.1 Libraries and structure

There are several libraries that are used. Three.js is used as a 3D engine. Underscore.js has many useful utility functions. Numeric.js is needed for the linear algebra used in physics and motion planning. JSZip is used to extract data from KMZ files. JQuery is used to integrate NCVR into the rest of the NCS webapp.

The internals of NCVR are divided into several modules, connected as seen in Figure 4.1. The main module sets up the initial configuration UI, connecting NCVR to the rest of the web interface. From this UI, you can 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. It sends the level data to the loader and the loader sends parsed level data back. This level data is formatted into a scene, which renders with the Three.js 3D library [9]. It also sends the level data to the physics engine, which builds a representation of the physical properties of the level. The app also 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 sends 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

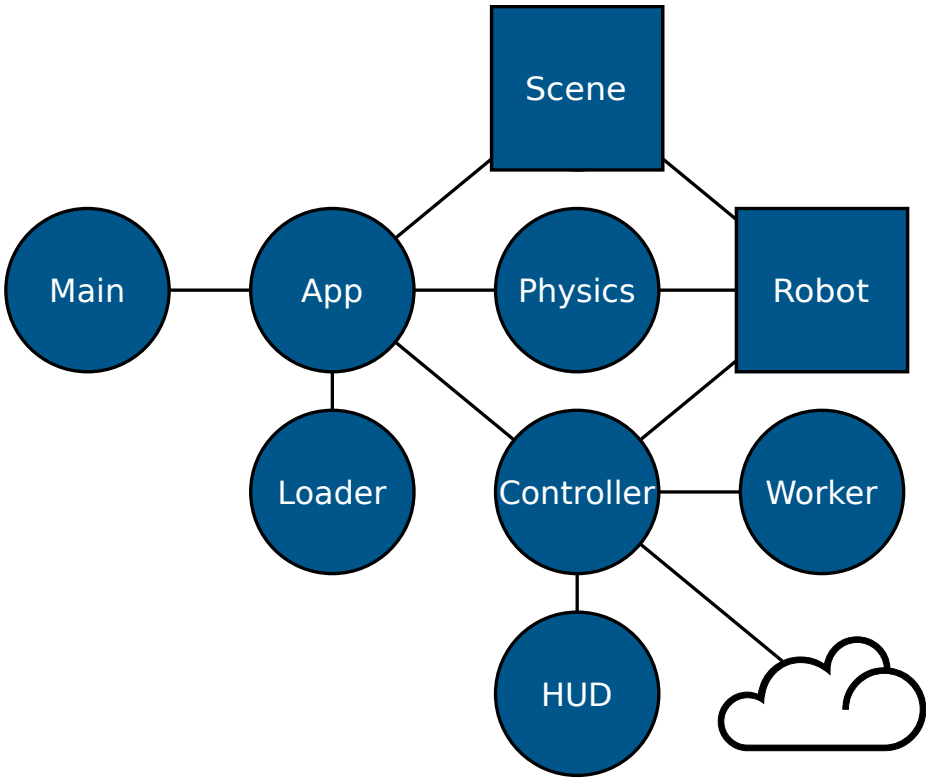

Figure 4.1: How the various modules are connected. The cloud represents the connection to the simulation, outside of the NCVR application.

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. For debugging purposes the controller also sends all of the sensor data and currently running actions to the HUD.

4.2 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`), that is, a center, three bases, the half extents, and an optional parent object. `OBB.fromPoints` uses a covariance method of fitting OBBs. The method used is relatively naive, as it uses all of the vertices in an object, rather than weighted triangles of the convex hull, as Gottschalk recommends [13]. This was done because the algorithms for convex hulls are rather difficult to implement and no implementations in JavaScript were found. In practice, the technique used was found to produce satisfactory bounding volumes for the purposes of this application.

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`. These are used in collision resolution. Note that none of them return a point of intersection, which makes the calculation of torques impossible.

For our utility functions, there are `calcBoundingSphereFromOBBs` and `calcOBBsVertices`. The former calculates a bounding sphere for an array of `OBB`s and the latter calculates all of the vertices of an array of `OBB`s. These are both used to

Figure 4.2: The Carl model from Mixamo, Inc. [2]

make bounding spheres, which are rather fast for coarse phase collision detection.

4.3 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 [2] Carl model for our robot, shown in Figure 4.2.

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 [24] 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.

The algorithm is always keeping track of a "best configuration." The initial best configuration is the current location of the arm. Every time the arm needs to be stepped toward a target, it spends 10 ms to find a better configuration. For these 10 ms, the following loop is repeated.

The first step of the loop is based directly on the work of Weghe [24], wherein 90% of the time, the program steps the best known arm configuration toward the target a short distance using the transpose of the Jacobian. The calculation of this is detailed in the Section 4.4.

The other 10% of the time, it searches for a better location "randomly." This prevents it from getting stuck in a local minimum, which tends to be where joints hit their limits without getting close enough to the object. We have found through informal experimentation that 10% appears to yield the fastest results. Unlike Weghe, we do not actually use a random configuration, but a configuration from the deterministic Halton sequence [14], which has guaranteed dispersal, covering the space more quickly and efficiently than a random sequence of points. This is in contrast to JavaScript's built-in random number generator, which is generally implemented as a weak random number generator and may be unevenly distributed in seven dimensions depending on the browser. For the Halton sequence, to fulfill our need for the seven mutally prime numbers required for the algorithm, the first seven primes are used: 2, 3, 5, 7, 11, 13, and 17.

In either case, there is a new candidate to test for our best target configuration. After ascertaining that the new configuration conforms to our joint bounds, it is checked for if the new configuration is better than the best seen so far. If it is, it becomes our new best configuration.

After spending 10 ms searching for a better configuration, it actually moves the arm toward the best configuration, by moving the joints a fixed distance through configuration space toward this target.

This entire procedure is done every frame until the hand reaches the best con-

figuration. It reports that the motion failed or succeeded, depending on the distance of the hand from the target pose.

4.4 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 $R_{6,7}\left(\vec{\theta}\right)$.

With all of our matrices, the program can calculate the arm configuration from the raw matrices of the arm provided by the 3D engine. This is done by using Sympy to solve for the equations. That is, if the 3D engine is using an arbitrary matrix M for the loaded shoulder, we can solve $R_{0,3}\left(\vec{\theta}\right) = M$ for $\vec{\theta}_0$, $\vec{\theta}_1$, and $\vec{\theta}_2$,

which parameterize the angles of the shoulder joints. The result of this is compiled into the `calcConfig` function. For consistency, it also calculates the inverse function from the exact same equations for the `configToMatrices` function.

In order to find if it is close to our target configuration and to define what we mean by moving the hand closer to its target, it needs to have a measurement of how far the hand is from a given pose. A pose is composed of two parts, the position and the orientation. These have different, incompatible units, meters and radians. Instead of measuring the distance in these terms, we instead pick two points on the hand and measure the distance between them and where they should be. To simplify the math, we assume that these two points are combined into a 6-dimensional vector, $\vec{P}$, in what we will call pose space. Then, the Euclidean distance can be taken between the two points. This distance is used to make the `calcDist` function of this module.

All of these are combined to calculate the Jacobian matrix of the hand, $\frac{\partial \vec{P}}{\partial \vec{\theta}}$. The Jacobian is the higher dimensional analog of the derivative. This matrix can tell us the velocity with which the hand will move given a change in any of the configuration parameters. By inverting the Jacobian, we can do the opposite, telling us what changes in the configuration parameters are needed to move the hand with a certain velocity. The problem is that calculating this inverse is not actually possible, as the Jacobian is not a square matrix. Fortunately, it has been found that the transpose of the Jacobian produces satisfactory results over short distances by Weghe [24]. Therefore, we can take a short vector in pose space that points to our target pose, multiply it by our Jacobian transpose, and get a vector in the configuration space that moves the arm such that the hand is closer to our target. The `stepConfig` function does this all for us.

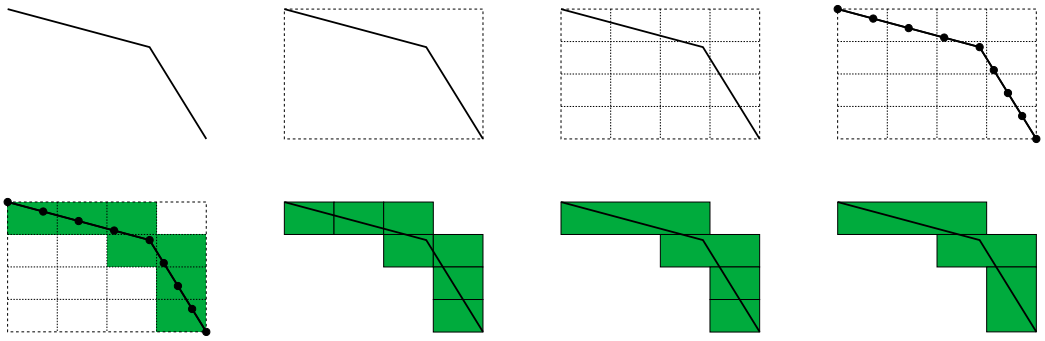

Figure 4.3: The steps used to build bounding boxes for static objects. This only uses two dimensions, while the actual algorithm works in three dimensions

4.5 Physics

The physics module is primarily responsible for collision resolution and bounding volume generation. Collision resolution is rather simple. 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. So, it translates whichever object is most portable by the distance of the penetration depth away from the other object, along the collision axis. This module also manages the fact that stuff falls. The ground has its own collision volume, which is a horizontal half space covering the space at and below the height of the lowest vertex in the environment.

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-tesselated voxelization [11]. The following steps are illustrated in Figure 4.3. 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. However, this yields many boxes, which slows down collision detection. So, we invented an algorithm to reduce the number of boxes, which sweeps across each axis, merging consecutive boxes if their dimensions match.

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`, which is detailed in Section 4.2, creating well fit boundaries around the arms.

4.6 Mock NCS Daemon

While this is not actually part of the NeoCortical Virtual Robot, it is necessary at this time. Where the real NCS daemon would create and run a brain simulation, this instead creates a graphical user interface that enables a human to view sensor data from the virtual robot and respond to stimuli, as if the human were a simulated brain. Since this is intended to be a temporary replacement for the real NCS daemon, the interface is rough. As explained in Section 3.3, this accepts a simple string indicating the number and types of input are available to a user. The only methods available are sliders and buttons, which can be denoted with the letters “s” and “b”. So, if a POST request is made to the server at the URL

`/simulations/` with the body "sbs", then an interface with a slider, a button, and another slider would be created in that order. This would be made accessible as a WebSocket as a number under that URL, like `/simulations/3`.

4.7 Other Modules

4.7.1 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.

4.7.2 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.

4.7.3 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.

4.7.4 Worker

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

4.7.5 Heads-Up Display

In Figure 4.4 is a box on the side that shows various sensors that the robot has access to. Its visibility can be toggled with a click.

Figure 4.4: Heads-up display

4.7.6 Utilities

These are a few utility functions. They unify how to do things between browsers and polyfill a few functions that should be included.

CHAPTER 5

ISSUES

5.1 JavaScript

JavaScript is not typically used to develop 3D video games, and, as such, many tools typically used for 3D simulations, such as collision volume generation libraries, do not exist. Since we want the skill required to be low, we needed to prevent a need for knowledge about collision volumes. Unfortunately, that means that the program needs to support colliding with models that are problematic, where, for example, the model has no thickness, thus violating many assumptions about how collisions work. In C and C++, many libraries are available to turn problematic models into workable collision volumes, but, since the algorithms involved are rather complex, porting them would be a difficult task.

Three.js has imperfect support for major model formats, making it difficult to get working articulated models. It's a better idea to convert the model into one of Three.js' internal formats using a plug-in with modeling software. It seems that Maya works best for articulated models, but we could not create articulated models that were also capable of facial expressions.

5.2 Physics

When starting this project, we evaluated a few options for 3D engines, such as Physijs [20], Cannon.js [15], and Ammo.js [25]. Since, at the time, it appeared that none of them supported arbitrary shapes with holes and we could increase the

running speed of NCVR by avoiding the use of unneeded features, we decided to create our own physics engine. However, as work progressed, it became clear that there were few features that could omitted and the missing support for arbitrary shapes could have been added to one of the existing solutions in less time than it took to create a physics engine. Throughout this process, the difficulty of getting our physics engine to run quickly enough made it clear that other physics engines, having already been optimized extensively, would be able to run faster, while maintaining physical accuracy.

5.3 Motion planning

The motion planning algorithm takes a long time if the object to be picked up is near the edge of reach or is out of reach. Motion ends up being jerky, changing direction when the algorithm finds a shorter path to grasp the object. The algorithm performs badly on long objects, as the grasping radius is a sphere, which is far from the actual object perpendicular to the longest axis.

CHAPTER 6
CONCLUSIONS 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 4.5. 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 [12]. 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 4.3.

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 probably require entire programs, which would need to be written is some programming language. In order to prevent neglectful programs from causing issues, this programming language should be amenable to sandboxing. Mainstream options for such languages are JavaScript, Lua, and Java.

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 require refactoring a large portion of NCVR's code.

The addition of sensors to allow for interfacing with the real world, through a webcam or microphone would be needed to reproduce those experiments. In addition to using sound in the real world, having sound in the virtual environment would make the environment more realistic.

The physics engine should be replaced with one that has been better tested and faster, such as Cannon.js [15]. 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.

The Carl model used is not capable of multiple expressions, which was originally a goal, based on Bray's previous experiments [6, 8]. We believe that this was caused by the exporting process, using the Autodesk Maya to Three.js export

plugin [9]. Specifically, since Three.js does not support more than four bones influencing any vertex at the same time, in order to export the model, it was required to reduce the number of influences. This was done automatically with Maya, but the results were wrong. For example, a bone that modifies eyebrow position also bends the glasses. It may be possible to reduce the number of influences better by using different settings; we had difficulty reinstalling Maya so that we could try again.

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

The scripting engine is somewhat awkward to use, since program state needs to be explicitly managed. The simplest abstraction for this is coroutines. The latest version of JavaScript (ES6) has generators, which provide a nice abstraction for managing state and temporarily giving up control. These features have not been implemented yet, but Babel [1] can compile such code. With these features one could reduce the script detailed in Section 3.2 from 37 lines of code to the 20 lines shown in Figure 6.1.

A level designer built into NCVB would make it easier to make experiments, as it would not require setting up separate software.

```
setSpeed(1.7);
while (!sensors.collision.front) {
    yield;
}
setSpeed(0);
yield goBackward(1.0).at(1.7);
while (true) {
    if (sensors.brainOutputs[0] > 0) {
        yield turnLeft(90);
        break;
    } else if (sensors.brainOutputs[1] > 0) {
        yield turnRight(90);
        break;
    } else if (sensors.brainOutputs[2] > 0) {
        yield goForward(1.0);
        break;
    }
}
yield reachForAndGrabWithRightHand("portable_thing");
yield turnRight(180);
```

Figure 6.1: What scripting could look like with ES6

APPENDIX A

SENSORS

All sensors are accessible as properties of the `sensor` object, like `sensor.speed`.

Name	Type	Description
`speed`	number	speed in m/s
`angularVelocity`	number	angular velocity in radians per second
`odometer`	number	distance travelled in meters
`compass`	number	current angle in degrees as a header
`arms.left.held`	string	the name of the object in the left hand
`arms.right.held`	string	the name of the object in the right hand
`expression`	string	the name of the current expression
`camera.data`	`Int8Buffer`	an array of raw RGBA pixel values for the first-person view from the robot
`camera.height`	number	the height, in pixels, of the robot camera view
`camera.width`	number	the width, in pixels, of the robot camera view
`collision.top`	boolean	if there is a collision with the top of the robot
`collision.bottom`	boolean	if there is a collision with the bottom of the robot
`collision.left`	boolean	if there is a collision with the left of the robot
`collision.right`	boolean	if there is a collision with the right of the robot

Name	Type	Description
`collision.front`	boolean	if there is a collision with the front of the robot
`collision.back`	boolean	if there is a collision with the back of the robot
`keys`	object	an object whose properties are the keys pressed on the keyboard
`brainOutputs`	array	an array of outputs from NCS

APPENDIX B

SCRIPTING FUNCTIONS

B.1 The `ExtendedAction` class

The `ExtendedAction` class uses the builder pattern to set several optional parameters for ExtendedActions. This means that `ExtendedAction` method calls can be chained.

at `at(speed)`

Sets a speed for the action to performed at, which is going to either be in meters per second or radians per second, depending on the action. This option is mutually exclusive with `over`. Returns this object instance.

---

over `over(seconds)`

Sets a time period for the action to take place over in seconds, as a number. This option is mutually exclusive with `at`. Returns this object instance.

---

then `then(newState)`

Sets the state to switch to on the completion of this action. `newState` can be a string for the name of the state, or a function. Returns this object instance.

B.2 Actions

getPixel `getPixel(x, y)`

	Gets the pixel value at `x` and `y`, where `x` and `y` are in $[-1, 1]$ on a Euclidean plane. Returns a color, which has the properties: `red`, `green`, `blue`, `hue`, `saturation`, and `value`.
goBackward	`goBackward(`*`distance`*`)` Goes backward by the distance specified, in meters. Returns an `ExtendedAction`.
goForward	`goForward(`*`distance`*`)` Goes forward by the distance specified, in meters. Returns an `ExtendedAction`.
grabWithLeftArm	`grabWithLeftArm()` Grabs an object with the left hand if there is one near enough to grab.
grabWithRightArm	`grabWithRightArm()` Grabs an object with the right hand if there is one near enough to grab.
reachForAndGrab-WithRightHand	`reachForAndGrabWithRightHand(`*`objName`*`)` Reaches for and grabs an object with the right hand.
reachForAndGrab-WithLeftHand	`reachForAndGrabWithLeftHand(`*`objName`*`)` Reaches for and grabs an object with the left hand.
log	`log(`*`error`*`)` Logs a message to the JavaScript error console.
next	`next(`*`newState`*`)` Sets the state. `newState` should be the name of another state specified in the script.

pointLeftArm	`pointLeftArm(`*`x, y, z`*`)` Points the left arm parallel to the direction of the given vector, relative to the shoulder. Returns an `ExtendedAction`.
pointLeftArmAt	`pointLeftArmAt(`*`objName`*`)` Points the left arm at the given object, specified by its name in a string. Returns an `ExtendedAction`.
pointRightArm	`pointRightArm(`*`x, y, z`*`)` Points the right arm parallel to the direction of the given vector, relative to the shoulder. Returns an `ExtendedAction`.
pointRightArmAt	`pointRightArmAt(`*`objName`*`)` Points the right arm at the given object, specified by its name in a string. Returns an `ExtendedAction`.
setSpeed	`setSpeed(`*`speed`*`)` Set the walking speed in m/s. Can be negative to walk backwards.
startTurningLeft	`startTurningLeft(`*`speed`*`)` Set the turning speed in radians per second to the left.
startTurningRight	`startTurningRight(`*`speed`*`)` Set the turning speed in radians per second to the right.
turnRight	`turnRight(`*`degrees`*`)` Turns right by the given number of degrees. Returns an `ExtendedAction`.

turnLeft	`turnLeft(`*degrees*`)` Turns left by the given number of degrees. Returns an `ExtendedAction`.
turnTowards	`turnTowards(`*objName*`)` Turn towards the named object. Returns an `ExtendedAction`.

BIBLIOGRAPHY

[1] Babel: The compiler for writing next generation javascript. [Online]. Available: https://babeljs.io/

[2] (2015) Mixamo, inc. Accessed: 2015-05-14. [Online]. Available: https://www.mixamo.com/

[3] E. O. Almachar, A. M. Falconi, K. A. Gilgen, N. M. Jordan, D. Tanna, R. V. Hoang, S. M. Dascalu, L. C. J. Bray, and F. 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, Oct. 2014.

[4] J. Berlinski, M. D. Chavez, C. Rowe, N. M. Jordan, D. Tanna, R. V. Hoang, S. M. Dascalu, L. C. J. Bray, and F. C. Harris Jr., "Neocortical builder: A web based front end for ncs," in *Proc. Computer Applications in Industry and Engineering*, New Orleans, Louisiana, Oct. 2014.

[5] L. C. J. Bray, "A circuit-level model of hippocampal, entorhinal and prefrontal dynamics underlying rodent maze navigational learning," Ph.D. dissertation, University of Nevada, Reno, 2010.

[6] L. C. J. Bray, S. R. Anumandla, C. M. Thibeault, R. V. Hoang, P. H. Goodman, S. M. Dascalu, B. D. Bryant, and F. C. Harris Jr, "Real-time human–robot interaction underlying neurorobotic trust and intent recognition," *Neural Networks*, vol. 32, pp. 130–137, 2012.

[7] L. C. J. Bray, E. R. Barker, G. B. Ferneyhough, R. V. Hoang, B. D. Bryant, S. M. Dascalu, and F. C. Harris Jr, "Goal-related navigation of a neuromorphic virtual robot," *BMC Neuroscience*, vol. 13, no. Suppl 1, p. O3, 2012. [Online]. Available: http://www.biomedcentral.com/1471-2202/13/S1/O3

[8] L. C. J. Bray, G. B. Ferneyhough, E. R. Barker, C. M. Thibeault, and F. C. Harris Jr, "Reward-based learning for virtual neurorobotics through emotional speech processing," *Frontiers in neurorobotics*, vol. 7, 2013.

[9] R. Cabello, "Three.js," https://github.com/mrdoob/three.js, 2015, accessed: 2015-05-14.

[10] Cyberbotics, "Webots: Robot simulator," accessed: 2015-05-14. [Online]. Available: http://www.cyberbotics.com

[11] Y. Fei, B. Wang, and J. Chen, "Point-tessellated voxelization," in *Proceedings of Graphics Interface 2012*, ser. GI '12. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2012, pp. 9–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=2305276.2305280

[12] P. H. Goodman, Q. Zou, and S.-M. Dascalu, "Framework and implications of virtual neurorobotics," *Frontiers in neuroscience*, vol. 2, no. 1, p. 123, 2008.

[13] S. Gottschalk, "Collision queries using oriented bounding boxes," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2000.

[14] J. H. Halton, "Algorithm 247: Radical-inverse quasi-random point sequence," *Commun. ACM*, vol. 7, no. 12, pp. 701–702, Dec. 1964. [Online]. Available: http://doi.acm.org/10.1145/355588.365104

[15] S. Hedman, "Cannon.js," 2015, accessed: 2015-05-14. [Online]. Available: http://cannonjs.org/

[16] R. V. Hoang, D. Tanna, L. C. J. Bray, S. M. Dascalu, and F. C. Harris Jr, "A novel cpu/gpu simulation environment for large-scale biologically realistic neural modeling," *Frontiers in neuroinformatics*, vol. 7, 2013.

[17] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[18] A. Jones, J. Cardoza, D. Liu, L. Jayet Bray, S. Dascalu, S. Louis, and F. Harris, "A novel 3d visualization tool for large-scale neural networks," *BMC Neuroscience*, vol. 14, no. Suppl 1, p. P158, 2013. [Online]. Available: http://www.biomedcentral.com/1471-2202/14/S1/P158

[19] V. Kapoor, E. Krampe, A. Klug, N. K. Logothetis, and T. I. Panagiotaropoulos, "Development of tube tetrodes and a multi-tetrode drive for deep structure electrophysiological recordings in the macaque brain," *Journal of Neuroscience Methods*, vol. 216, no. 1, pp. 43 – 48, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0165027013001222

[20] C. Prall, "Physijs," https://github.com/chandlerprall/Physijs, 2015, accessed: 2015-05-14.

[21] T. Trappenberg, *Fundamentals of Computational Neuroscience*, ser. Fundamentals of Computational Neuroscience. OUP Oxford, 2010. [Online]. Available: http://books.google.com/books?id=1xSLktiRAX4C

[22] Trimble Navigation. (2015) The 3d warehouse website. Accessed: 2015-05-14. [Online]. Available: https://3dwarehouse.sketchup.com/

[23] ——. (2015) Sketchup. Accessed: 2015-05-14. [Online]. Available: http://www.sketchup.com/

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

[25] A. Zakai, "ammo.js," https://github.com/kripken/ammo.js/, 2015.