Recognizing Actions using Embodiment & Empathy

by

Robert Louis McIntyre

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 23, 2014

Certified by . . . . . .
Patrick H. Winston
Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Recognizing Actions using Embodiment & Empathy

by

Robert Louis McIntyre

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Here I demonstrate the power of using embodied artificial intelligence to attack the *action recognition* problem, which is the challenge of recognizing actions performed by a creature given limited data about the creature's actions, such as a video recording. I solve this problem in the case of a worm-like creature performing actions such as curling and wiggling.

To attack the action recognition problem, I developed a computational model of empathy (`EMPATH`) which allows me to recognize actions using simple, embodied representations of actions (which require rich sensory data), even when that sensory data is not actually available. The missing sense data is "imagined" by the system by combining previous experiences gained from unsupervised free play. The worm is a five-segment creature equipped with touch, proprioception, and muscle tension senses. It recognizes actions using only proprioception data.

In order to build this empathic, action-recognizing system, I created a program called `CORTEX`, which is a complete platform for embodied AI research. It provides multiple senses for simulated creatures, including vision, touch, proprioception, muscle tension, and hearing. Each of these senses provides a wealth of parameters that are biologically inspired. `CORTEX` is able to simulate any number of creatures and senses, and provides facilities for easily modeling and creating new creatures. As a research platform it is more complete than any other system currently available.

Thesis Supervisor: Patrick H. Winston
Title: Ford Professor of Artificial Intelligence and Computer Science

Contents

1 Empathy & Embodiment: problem solving strategies

By the end of this thesis, you will have a novel approach to representing an recognizing physical actions using embodiment and empathy. You will also see one way to efficiently implement physical empathy for embodied creatures. Finally, you will become familiar with `CORTEX`, a system for designing and simulating creatures with rich senses, which I have designed as a library that you can use in your own research. Note that I *do not* process video directly — I start with knowledge of the positions of a creature's body parts and works from there.

This is the core vision of my thesis: That one of the important ways in which we understand others is by imagining ourselves in their position and emphatically feeling experiences relative to our own bodies. By understanding events in terms of our own previous corporeal experience, we greatly constrain the possibilities of what would otherwise be an unwieldy exponential search. This extra constraint can be the difference between easily understanding what is happening in a video and being completely lost in a sea of incomprehensible color and movement.

1.1 The problem: recognizing actions is hard!

Examine the following image. What is happening? As you, and indeed very young children, can easily determine, this is an image of drinking.

Nevertheless, it is beyond the state of the art for a computer vision program to describe what's happening in this image. Part of the problem is that many computer vision systems focus on pixel-level details or comparisons to example images (such as Ke, Sukthankar, and Hebert 2005), but the 3D world is so variable that it is hard to describe the world in terms of possible images.

In fact, the contents of a scene may have much less to do with pixel probabilities than with recognizing various affordances: things you can move, objects you can grasp, spaces that can be filled . For example, what processes might enable you to see the chair in figure 2?

Finally, how is it that you can easily tell the difference between how the girls *muscles* are working in figure 3?

Each of these examples tells us something about what might be going on in our minds as we easily solve these recognition problems:

- The hidden chair shows us that we are strongly triggered by cues relating to the position of human bodies, and that we can determine the overall physical configuration of a human body even if much of that body is occluded.

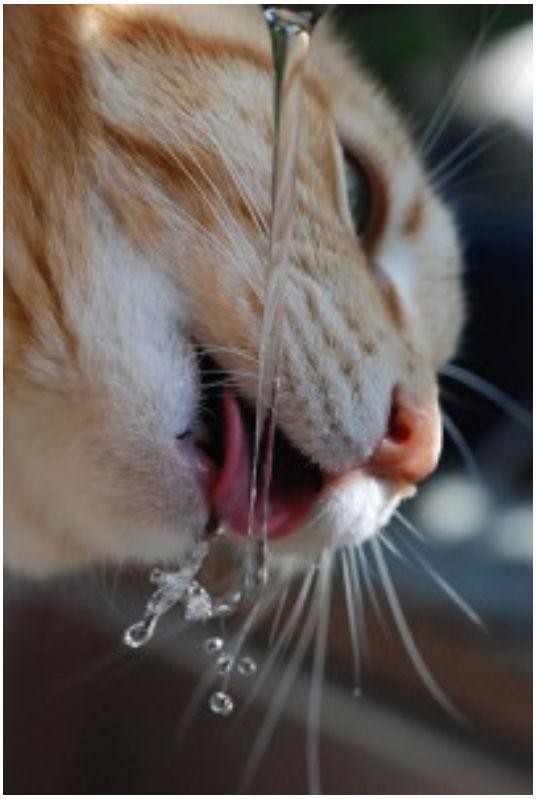

Figure 1: A cat drinking some water. Identifying this action is beyond the capabilities of existing computer vision systems.

- The picture of the girl pushing against the wall tells us that we have common sense knowledge about the kinetics of our own bodies. We know well how our muscles would have to work to maintain us in most positions, and we can easily project this self-knowledge to imagined positions triggered by images of the human body.
- The cat tells us that imagination of some kind plays an important role in understanding actions. The question is: Can we be more precise about what sort of imagination is required to understand these actions?

Figure 2: The chair in this image is quite obvious to humans, but it can't be found by any modern computer vision program.

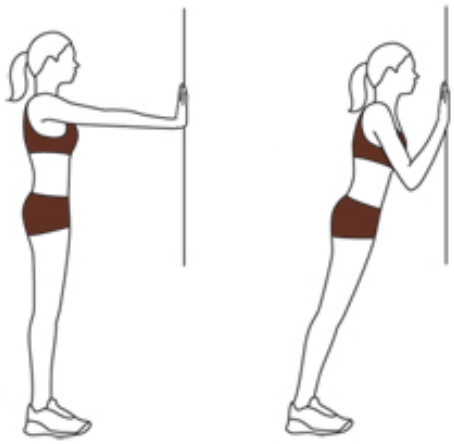

Figure 3: The mysterious "common sense" appears here as you are able to discern the difference in how the girl's arm muscles are activated between the two images.

1.2 A step forward: the sensorimotor-centered approach

In this thesis, I explore the idea that our knowledge of our own bodies, combined with our own rich senses, enables us to recognize the actions of others.

For example, I think humans are able to label the cat video as "drinking" because they imagine *themselves* as the cat, and imagine putting their face up against a stream of water and sticking out their tongue. In that imagined world, they can feel the cool water hitting their tongue, and feel the water entering their body, and are able to recognize that *feeling* as drinking. So, the label of the action is not really in the pixels of the image, but is found clearly in a simulation / recollection inspired by those pixels. An imaginative system, having been trained on drinking and non-drinking examples and learning that the most important component of drinking is the feeling of water sliding down one's throat, would analyze a video of a cat drinking in the following manner:

1. Create a physical model of the video by putting a "fuzzy" model of its own body in place of the cat. Possibly also create a simulation of the stream of water.

2. "Play out" this simulated scene and generate imagined sensory experience. This will include relevant muscle contractions, a close up view of the stream from the cat's perspective, and most importantly, the imagined feeling of water entering the mouth. The imagined sensory experience can come from a simulation of the event, but can also be pattern-matched from previous, similar embodied experience.

3. The action is now easily identified as drinking by the sense of taste alone. The other senses (such as the tongue moving in and out) help to give plausibility to the simulated action. Note that the sense of vision, while critical in creating the simulation, is not critical for identifying the action from the simulation.

For the chair examples, the process is even easier:

1. Align a model of your body to the person in the image.

2. Generate proprioceptive sensory data from this alignment.

3. Use the imagined proprioceptive data as a key to lookup related sensory experience associated with that particular proprioceptive feeling.

4. Retrieve the feeling of your bottom resting on a surface, your knees bent, and your leg muscles relaxed.

5. This sensory information is consistent with your `sitting?` sensory predicate, so you (and the entity in the image) must be sitting.

6. There must be a chair-like object since you are sitting.

Empathy offers yet another alternative to the age-old AI representation question: "What is a chair?" — A chair is the feeling of sitting!

One powerful advantage of empathic problem solving is that it factors the action recognition problem into two easier problems. To use empathy, you need an *aligner*, which takes the video and a model of your body, and aligns the model with the video. Then, you need a *recognizer*, which uses the aligned model to interpret the action. The power in this method lies in the fact that you describe all actions from a body-centered viewpoint. You are less tied to the particulars of any visual representation of the actions. If you teach the system what "running" is, and you have a good enough aligner, the system will from then on be able to recognize running from any point of view – even strange points of view like above or underneath the runner. This is in contrast to action recognition schemes that try to identify actions using a non-embodied approach. If these systems learn about running as viewed from the side, they will not automatically be able to recognize running from any other viewpoint.

Another powerful advantage is that using the language of multiple body-centered rich senses to describe body-centered actions offers a massive boost in descriptive capability. Consider how difficult it would be to compose a set of HOG (Histogram of Oriented Gradients) filters to describe the action of a simple worm-creature "curling" so that its head touches its tail, and then behold the simplicity of describing thus action in a language designed for the task (listing 1):

> Listing 1: Body-centered actions are best expressed in a body-centered language. This code detects when the worm has curled into a full circle. Imagine how you would replicate this functionality using low-level pixel features such as HOG filters!

```
(defn grand-circle?
  "Does the worm form a majestic circle (one end touching the other)?"
  [experiences]
  (and (curled? experiences)
       (let [worm-touch (:touch (peek experiences))
             tail-touch (worm-touch 0)
             head-touch (worm-touch 4)]
         (and (< 0.2 (contact worm-segment-bottom-tip tail-touch))
              (< 0.2 (contact worm-segment-top-tip    head-touch))))))
```

1.3 EMPATH recognizes actions using empathy

Exploring these ideas further demands a concrete implementation, so first, I built a system for constructing virtual creatures with physiologically plausible sensorimotor systems

and detailed environments. The result is CORTEX, which is described in section 2.

Next, I wrote routines which enabled a simple worm-like creature to infer the actions of a second worm-like creature, using only its own prior sensorimotor experiences and knowledge of the second worm's joint positions. This program, EMPATH, is described in section 3. It's main components are:

Embodied Action Definitions Many otherwise complicated actions are easily described in the language of a full suite of body-centered, rich senses and experiences. For example, drinking is the feeling of water sliding down your throat, and cooling your insides. It's often accompanied by bringing your hand close to your face, or bringing your face close to water. Sitting down is the feeling of bending your knees, activating your quadriceps, then feeling a surface with your bottom and relaxing your legs. These body-centered action descriptions can be either learned or hard coded.

Guided Play The creature moves around and experiences the world through its unique perspective. As the creature moves, it gathers experiences that satisfy the embodied action definitions.

Posture imitation When trying to interpret a video or image, the creature takes a model of itself and aligns it with whatever it sees. This alignment might even cross species, as when humans try to align themselves with things like ponies, dogs, or other humans with a different body type.

Empathy The alignment triggers associations with sensory data from prior experiences. For example, the alignment itself easily maps to proprioceptive data. Any sounds or obvious skin contact in the video can to a lesser extent trigger previous experience keyed to hearing or touch. Segments of previous experiences gained from play are stitched together to form a coherent and complete sensory portrait of the scene.

Recognition With the scene described in terms of remembered first person sensory events, the creature can now run its action-definition programs (such as the one in listing 1) on this synthesized sensory data, just as it would if it were actually experiencing the scene first-hand. If previous experience has been accurately retrieved, and if it is analogous enough to the scene, then the creature will correctly identify the action in the scene.

My program, EMPATH uses this empathic problem solving technique to interpret the actions of a simple, worm-like creature.

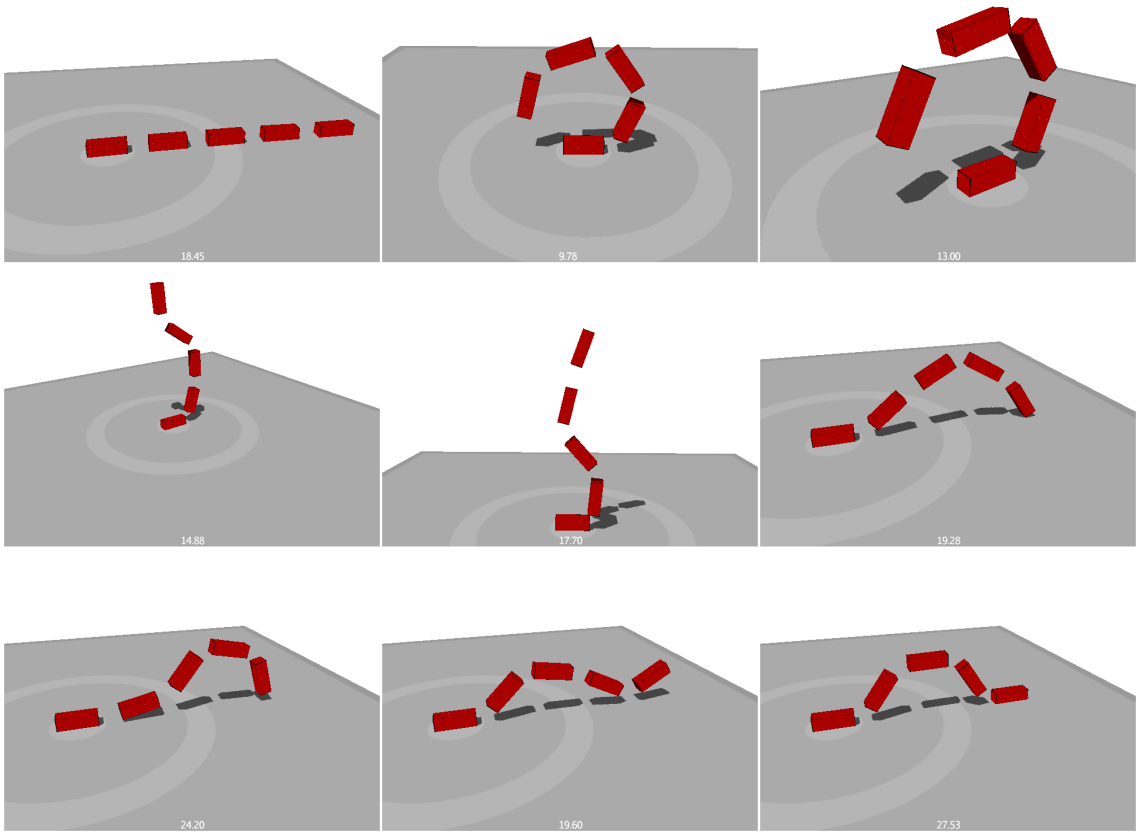

Figure 4: The worm performs many actions during free play such as curling, wiggling, and resting.

Main Results

- After one-shot supervised training, EMPATH was able to recognize a wide variety of static poses and dynamic actions—ranging from curling in a circle to wiggling with a particular frequency — with 95% accuracy.

- These results were completely independent of viewing angle because the underlying body-centered language fundamentally is independent; once an action is learned, it can be recognized equally well from any viewing angle.

- EMPATH is surprisingly short; the sensorimotor-centered language provided by CORTEX resulted in extremely economical recognition routines — about 500 lines in all — suggesting that such representations are very powerful, and often indispensable

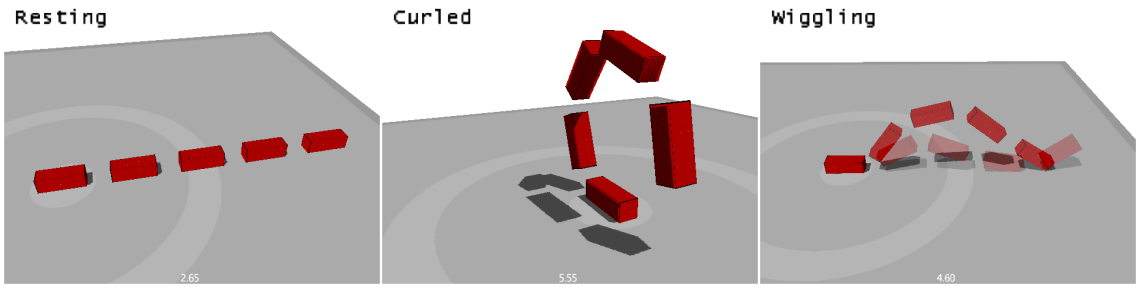

Figure 5: EMPATH recognized and classified each of these poses by inferring the complete sensory experience from proprioceptive data.

for the types of recognition tasks considered here.

- Although for expediency's sake, I relied on direct knowledge of joint positions in this proof of concept, it would be straightforward to extend EMPATH so that it (more realistically) infers joint positions from its visual data.

1.4 EMPATH is built on CORTEX, a creature builder.

I built CORTEX to be a general AI research platform for doing experiments involving multiple rich senses and a wide variety and number of creatures. I intend it to be useful as a library for many more projects than just this thesis. CORTEX was necessary to meet a need among AI researchers at CSAIL and beyond, which is that people often will invent wonderful ideas that are best expressed in the language of creatures and senses, but in order to explore those ideas they must first build a platform in which they can create simulated creatures with rich senses! There are many ideas that would be simple to execute (such as EMPATH or Larson's self-organizing maps (Larson 2003)), but attached to them is the multi-month effort to make a good creature simulator. Often, that initial investment of time proves to be too much, and the project must make do with a lesser environment or be abandoned entirely.

CORTEX is well suited as an environment for embodied AI research for three reasons:

- You can design new creatures using Blender (*Blender* 2013), a popular 3D modeling program. Each sense can be specified using special blender nodes with biologically inspired parameters. You need not write any code to create a creature, and can use a wide library of pre-existing blender models as a base for your own creatures.

- CORTEX implements a wide variety of senses: touch, proprioception, vision, hearing, and muscle tension. Complicated senses like touch and vision involve multiple

sensory elements embedded in a 2D surface. You have complete control over the distribution of these sensor elements through the use of simple png image files. CORTEX implements more comprehensive hearing than any other creature simulation system available.

- CORTEX supports any number of creatures and any number of senses. Time in CORTEX dilates so that the simulated creatures always perceive a perfectly smooth flow of time, regardless of the actual computational load.

CORTEX is built on top of jMonkeyEngine3 (*jMonkeyEngine3* 2013), which is a video game engine designed to create cross-platform 3D desktop games. CORTEX is mainly written in clojure, a dialect of LISP that runs on the java virtual machine (JVM). The API for creating and simulating creatures and senses is entirely expressed in clojure, though many senses are implemented at the layer of jMonkeyEngine or below. For example, for the sense of hearing I use a layer of clojure code on top of a layer of java JNI bindings that drive a layer of C++ code which implements a modified version of OpenAL to support multiple listeners. CORTEX is the only simulation environment that I know of that can support multiple entities that can each hear the world from their own perspective. Other senses also require a small layer of Java code. CORTEX also uses bullet, a physics simulator written in C.

Here are some things I anticipate that CORTEX might be used for:

- exploring new ideas about sensory integration
- distributed communication among swarm creatures
- self-learning using free exploration,
- evolutionary algorithms involving creature construction
- exploration of exotic senses and effectors that are not possible in the real world (such as telekinesis or a semantic sense)
- imagination using subworlds

During one test with CORTEX, I created 3,000 creatures each with their own independent senses and ran them all at only 1/80 real time. In another test, I created a detailed model of my own hand, equipped with a realistic distribution of touch (more sensitive at the fingertips), as well as eyes and ears, and it ran at around 1/4 real time.

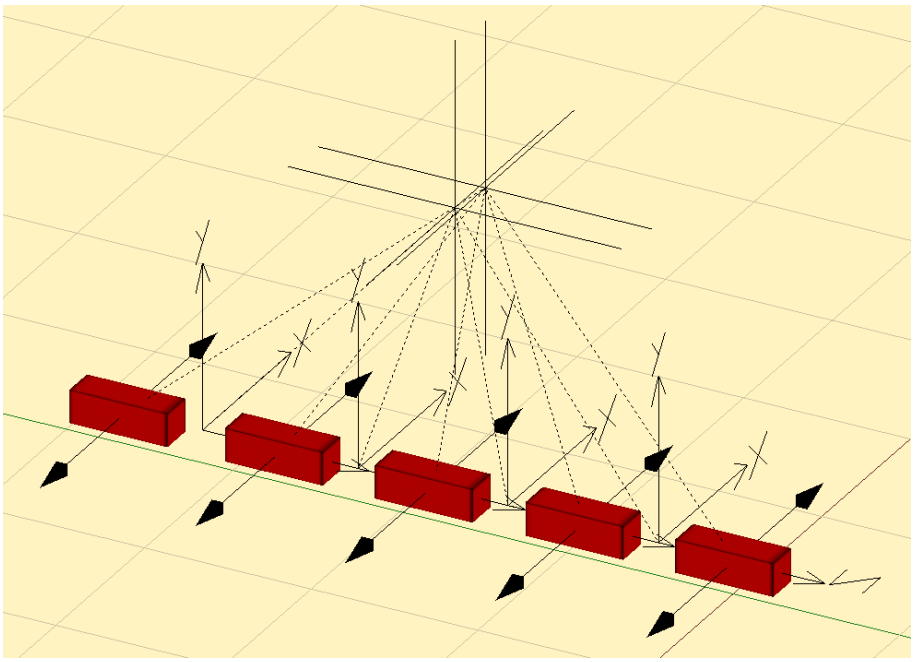

Figure 6: Here is the worm from figure 4 modeled in Blender, a free 3D-modeling program. Senses and joints are described using special nodes in Blender.

ture builder.

Figure 7: I modeled my own right hand in Blender and rigged it with all the senses that CORTEX supports. My simulated hand has a biologically inspired distribution of touch sensors. The senses are displayed on the right, and the simulation is displayed on the left. Notice that my hand is curling its fingers, that it can see its own finger from the eye in its palm, and that it can feel its own thumb touching its palm.

2 Designing CORTEX

In this section, I outline the design decisions that went into making CORTEX, along with some details about its implementation. (A practical guide to getting started with CORTEX, which skips over the history and implementation details presented here, is provided in an appendix at the end of this thesis.)

Throughout this project, I intended for CORTEX to be flexible and extensible enough to be useful for other researchers who want to test ideas of their own. To this end, wherever I have had to make architectural choices about CORTEX, I have chosen to give as much freedom to the user as possible, so that CORTEX may be used for things I have not foreseen.

2.1 Building in simulation versus reality

The most important architectural decision of all is the choice to use a computer-simulated environment in the first place! The world is a vast and rich place, and for now simulations are a very poor reflection of its complexity. It may be that there is a significant qualitative difference between dealing with senses in the real world and dealing with pale facsimiles of them in a simulation (Brooks 1991). What are the advantages and disadvantages of a simulation vs. reality?

Simulation

The advantages of virtual reality are that when everything is a simulation, experiments in that simulation are absolutely reproducible. It's also easier to change the creature and environment to explore new situations and different sensory combinations.

If the world is to be simulated on a computer, then not only do you have to worry about whether the creature's senses are rich enough to learn from the world, but whether the world itself is rendered with enough detail and realism to give enough working material to the creature's senses. To name just a few difficulties facing modern physics simulators: destructibility of the environment, simulation of water/other fluids, large areas, nonrigid bodies, lots of objects, smoke. I don't know of any computer simulation that would allow a creature to take a rock and grind it into fine dust, then use that dust to make a clay sculpture, at least not without spending years calculating the interactions of every single small grain of dust. Maybe a simulated world with today's limitations doesn't provide enough richness for real intelligence to evolve.

Reality

The other approach for playing with senses is to hook your software up to real cameras, microphones, robots, etc., and let it loose in the real world. This has the advantage of eliminating concerns about simulating the world at the expense of increasing the complexity of implementing the senses. Instead of just grabbing the current rendered frame for processing, you have to use an actual camera with real lenses and interact with photons to get an image. It is much harder to change the creature, which is now partly a physical robot of some sort, since doing so involves changing things around in the real world instead of modifying lines of code. While the real world is very rich and definitely provides enough stimulation for intelligence to develop (as evidenced by our own existence), it is also uncontrollable in the sense that a particular situation cannot be recreated perfectly or saved for later use. It is harder to conduct Science because it is harder to repeat an experiment. The worst thing about using the real world instead of a simulation is the matter of time. Instead of simulated time you get the constant and unstoppable flow of real time. This severely limits the sorts of software you can use to program an AI, because all sense inputs must be handled in real time. Complicated ideas may have to be implemented in hardware or may simply be impossible given the current speed of our processors. Contrast this with a simulation, in which the flow of time in the simulated world can be slowed down to accommodate the limitations of the creature's programming. In terms of cost, doing everything in software is far cheaper than building custom real-time hardware. All you need is a laptop and some patience.

2.2 Simulated time enables rapid prototyping & simple programs

I envision CORTEX being used to support rapid prototyping and iteration of ideas. Even if I could put together a well constructed kit for creating robots, it would still not be enough because of the scourge of real-time processing. Anyone who wants to test their ideas in the real world must always worry about getting their algorithms to run fast enough to process information in real time. The need for real time processing only increases if multiple senses are involved. In the extreme case, even simple algorithms will have to be accelerated by ASIC chips or FPGAs, turning what would otherwise be a few lines of code and a 10x speed penalty into a multi-month ordeal. For this reason, CORTEX supports *time-dilation*, which scales back the framerate of the simulation in proportion to the amount of processing each frame. From the perspective of the creatures inside the simulation, time always appears to flow at a constant rate, regardless of how complicated the environment becomes or how many creatures are in the simulation. The cost is that CORTEX can sometimes run slower than real time. Time dilation works both ways, however — simulations of very simple creatures in CORTEX generally run at 40x real-time on

my machine!

2.3 All sense organs are two-dimensional surfaces

If CORTEX is to support a wide variety of senses, it would help to have a better understanding of what a sense actually is! While vision, touch, and hearing all seem like they are quite different things, I was surprised to learn during the course of this thesis that they (and all physical senses) can be expressed as exactly the same mathematical object!

Human beings are three-dimensional objects, and the nerves that transmit data from our various sense organs to our brain are essentially one-dimensional. This leaves up to two dimensions in which our sensory information may flow. For example, imagine your skin: it is a two-dimensional surface around a three-dimensional object (your body). It has discrete touch sensors embedded at various points, and the density of these sensors corresponds to the sensitivity of that region of skin. Each touch sensor connects to a nerve, all of which eventually are bundled together as they travel up the spinal cord to the brain. Intersect the spinal nerves with a guillotining plane and you will see all of the sensory data of the skin revealed in a roughly circular two-dimensional image which is the cross section of the spinal cord. Points on this image that are close together in this circle represent touch sensors that are *probably* close together on the skin, although there is of course some cutting and rearrangement that has to be done to transfer the complicated surface of the skin onto a two dimensional image.

Most human senses consist of many discrete sensors of various properties distributed along a surface at various densities. For skin, it is Pacinian corpuscles, Meissner's corpuscles, Merkel's disks, and Ruffini's endings (Bear et al. 2006), which detect pressure and vibration of various intensities. For ears, it is the stereocilia distributed along the basilar membrane inside the cochlea; each one is sensitive to a slightly different frequency of sound. For eyes, it is rods and cones distributed along the surface of the retina. In each case, we can describe the sense with a surface and a distribution of sensors along that surface.

In fact, almost every human sense can be effectively described in terms of a surface containing embedded sensors. If the sense had any more dimensions, then there wouldn't be enough room in the spinal cord to transmit the information!

Therefore, CORTEX must support the ability to create objects and then be able to "paint" points along their surfaces to describe each sense.

Fortunately this idea is already a well known computer graphics technique called *UV-mapping*. In UV-mapping, the three-dimensional surface of a model is cut and smooshed until it fits on a two-dimensional image. You paint whatever you want on that image, and when the three-dimensional shape is rendered in a game the smooshing and cutting

is reversed and the image appears on the three-dimensional object.

To make a sense, interpret the UV-image as describing the distribution of that senses sensors. To get different types of sensors, you can either use a different color for each type of sensor, or use multiple UV-maps, each labeled with that sensor type. I generally use a white pixel to mean the presence of a sensor and a black pixel to mean the absence of a sensor, and use one UV-map for each sensor-type within a given sense.

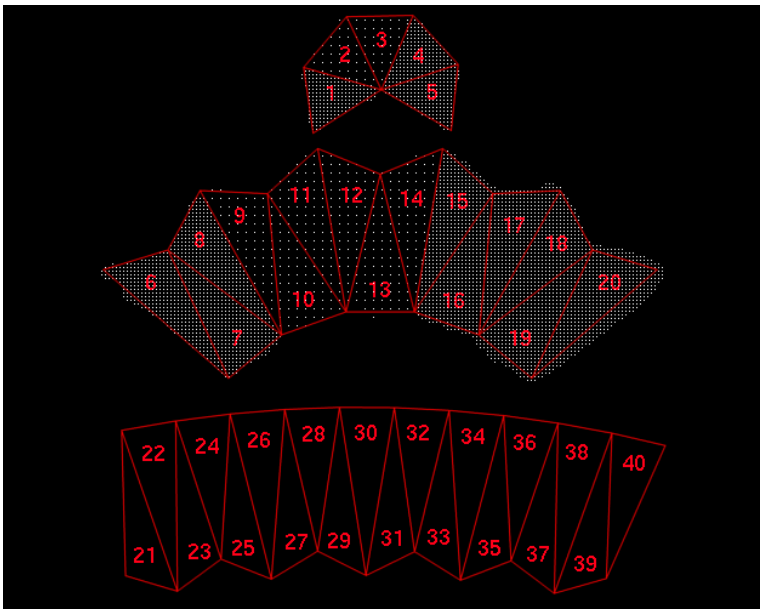

Figure 8: The UV-map for an elongated icososphere. The white dots each represent a touch sensor. They are dense in the regions that describe the tip of the finger, and less dense along the dorsal side of the finger opposite the tip.

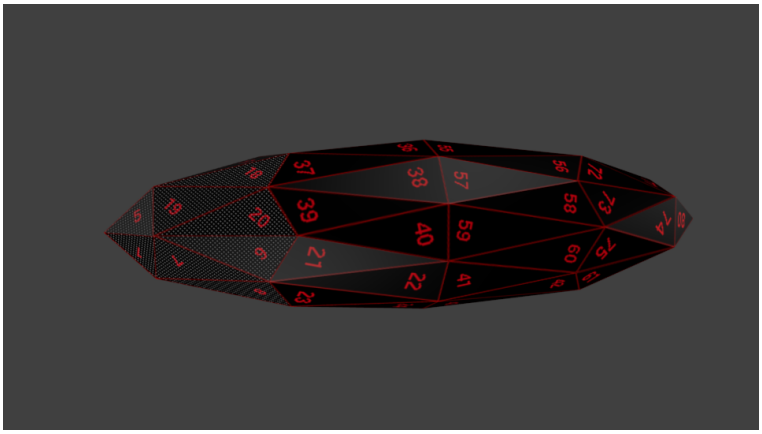

Figure 9: Ventral side of the UV-mapped finger. Notice the density of touch sensors at the tip.

2.4 Video game engines provide ready-made physics and shading

I did not need to write my own physics simulation code or shader to build CORTEX. Doing so would lead to a system that is impossible for anyone but myself to use anyway. Instead, I use a video game engine as a base and modify it to accommodate the additional needs of CORTEX. Video game engines are an ideal starting point to build CORTEX, because they are not far from being creature building systems themselves.

First off, general purpose video game engines come with a physics engine and lighting / sound system. The physics system provides tools that can be co-opted to serve as touch, proprioception, and muscles. Since some games support split screen views, a good video game engine will allow you to efficiently create multiple cameras in the simulated world that can be used as eyes. Video game systems offer integrated asset management for things like textures and creature models, providing an avenue for defining creatures. They also understand UV-mapping, since this technique is used to apply a texture to a model. Finally, because video game engines support a large number of developers, as long as CORTEX doesn't stray too far from the base system, other researchers can turn to this community for help when doing their research.

2.5 CORTEX is based on jMonkeyEngine3

While preparing to build CORTEX I studied several video game engines to see which would best serve as a base. The top contenders were:

Quake II/Jake2 The Quake II engine was designed by ID software in 1997. All the

source code was released by ID software into the Public Domain several years ago, and as a result it has been ported to many different languages. This engine was famous for its advanced use of realistic shading and it had decent and fast physics simulation. The main advantage of the Quake II engine is its simplicity, but I ultimately rejected it because the engine is too tied to the concept of a first-person shooter game. One of the problems I had was that there does not seem to be any easy way to attach multiple cameras to a single character. There are also several physics clipping issues that are corrected in a way that only applies to the main character and do not apply to arbitrary objects.

Source Engine The Source Engine evolved from the Quake II and Quake I engines and is used by Valve in the Half-Life series of games. The physics simulation in the Source Engine is quite accurate and probably the best out of all the engines I investigated. There is also an extensive community actively working with the engine. However, applications that use the Source Engine must be written in C++, the code is not open, it only runs on Windows, and the tools that come with the SDK to handle models and textures are complicated and awkward to use.

jMonkeyEngine3 jMonkeyEngine3 is a new library for creating games in Java. It uses OpenGL to render to the screen and uses screengraphs to avoid drawing things that do not appear on the screen. It has an active community and several games in the pipeline. The engine was not built to serve any particular game but is instead meant to be used for any 3D game.

I chose jMonkeyEngine3 because it had the most features out of all the free projects I looked at, and because I could then write my code in clojure, an implementation of LISP that runs on the JVM.

2.6 CORTEX uses Blender to create creature models

For the simple worm-like creatures I will use later on in this thesis, I could define a simple API in CORTEX that would allow one to create boxes, spheres, etc., and leave that API as the sole way to create creatures. However, for CORTEX to truly be useful for other projects, it needs a way to construct complicated creatures. If possible, it would be nice to leverage work that has already been done by the community of 3D modelers, or at least enable people who are talented at modeling but not programming to design CORTEX creatures.

Therefore I use Blender, a free 3D modeling program, as the main way to create creatures in CORTEX. However, the creatures modeled in Blender must also be simple to simulate in jMonkeyEngine3's game engine, and must also be easy to rig with CORTEX's senses. I accomplish this with extensive use of Blender's "empty nodes."

Empty nodes have no mass, physical presence, or appearance, but they can hold meta-data and have names. I use a tree structure of empty nodes to specify senses in the following manner:

- Create a single top-level empty node whose name is the name of the sense.
- Add empty nodes which each contain meta-data relevant to the sense, including a UV-map describing the number/distribution of sensors if applicable.
- Make each empty-node the child of the top-level node.

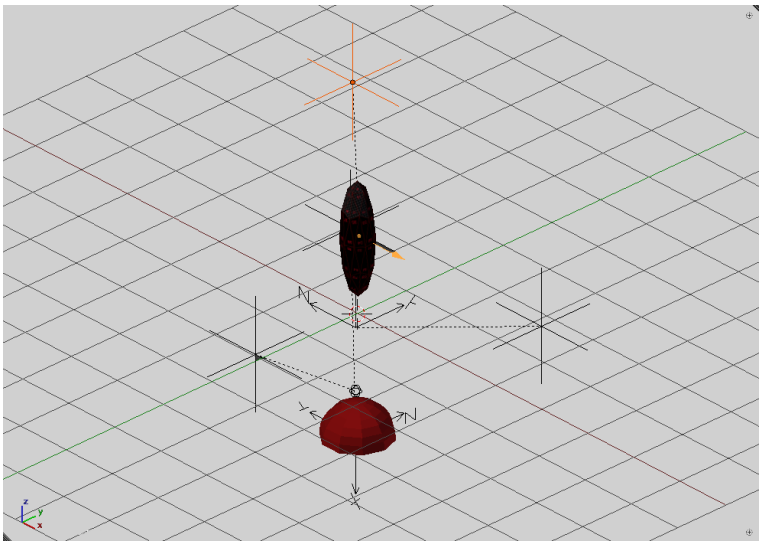

Figure 10: An example of annotating a creature model with empty nodes to describe the layout of senses. There are multiple empty nodes which each describe the position of muscles, ears, eyes, or joints.

2.7 Bodies are composed of segments connected by joints

Blender is a general purpose animation tool, which has been used in the past to create high quality movies such as Sintel (*Blender* 2013). Though Blender can model and render even complicated things like water, it is crucial to keep models that are meant to be simulated as creatures simple. `Bullet`, which `CORTEX` uses though jMonkeyEngine3, is a rigid-body physics system. This offers a compromise between the expressiveness of a game level and the speed at which it can be simulated, and it means that creatures should be naturally expressed as rigid components held together by joint constraints.

But humans are more like a squishy bag wrapped around some hard bones which define the overall shape. When we move, our skin bends and stretches to accommodate the new positions of our bones.

One way to make bodies composed of rigid pieces connected by joints *seem* more human-like is to use an *armature*, (or *rigging*) system, which defines a overall "body mesh" and defines how the mesh deforms as a function of the position of each "bone" which is a standard rigid body. This technique is used extensively to model humans and create realistic animations. It is not a good technique for physical simulation because it is a lie – the skin is not a physical part of the simulation and does not interact with any objects in the world or itself. Objects will pass right though the skin until they come in contact with the underlying bone, which is a physical object. Without simulating the skin, the sense of touch has little meaning, and the creature's own vision will lie to it about the true extent of its body. Simulating the skin as a physical object requires some way to continuously update the physical model of the skin along with the movement of the bones, which is unacceptably slow compared to rigid body simulation.

Therefore, instead of using the human-like "bony meatbag" approach, I decided to base my body plans on multiple solid objects that are connected by joints, inspired by the robot EVE from the movie WALL-E.

EVE's body is composed of several rigid components that are held together by invisible joint constraints. This is what I mean by *eve-like*. The main reason that I use eve-like bodies is for simulation efficiency, and so that there will be correspondence between the AI's senses and the physical presence of its body. Each individual section is simulated by a separate rigid body that corresponds exactly with its visual representation and does not change. Sections are connected by invisible joints that are well supported in jMonkeyEngine3. Bullet, the physics backend for jMonkeyEngine3, can efficiently simulate hundreds of rigid bodies connected by joints. Just because sections are rigid does not mean they have to stay as one piece forever; they can be dynamically replaced with multiple sections to simulate splitting in two. This could be used to simulate retractable claws or EVE's hands, which are able to coalesce into one object in the movie.

Solidifying/Connecting a body

CORTEX creates a creature in two steps: first, it traverses the nodes in the blender file and creates physical representations for any of them that have mass defined in their blender meta-data.

Listing 2: Program for iterating through the nodes in a blender file and generating physical jMonkeyEngine3 objects with mass and a matching physics shape.

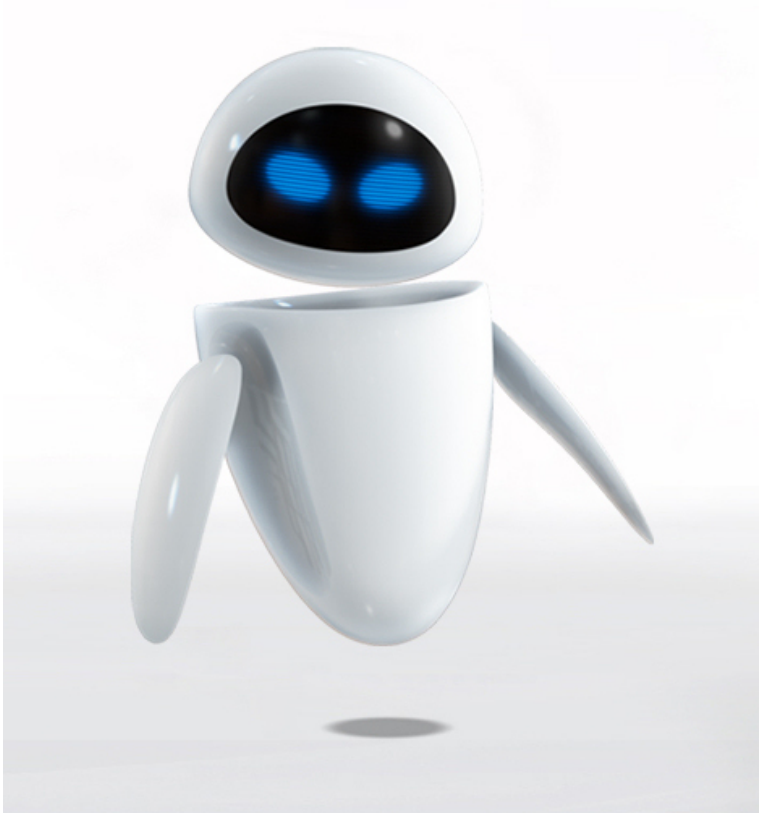

Figure 11: EVE from the movie WALL-E. This body plan turns out to be much better suited to my purposes than a more human-like one.

```
(defn physical!
  "Iterate through the nodes in creature and make them real physical
   objects in the simulation."
  [#^Node creature]
  (dorun
   (map
    (fn [geom]
      (let [physics-control
            (RigidBodyControl.
             (HullCollisionShape.
              (.getMesh geom))
             (if-let [mass (meta-data geom "mass")]
               (float mass) (float 1)))]
        (.addControl geom physics-control)))
    (filter #(isa? (class %) Geometry )
            (node-seq creature)))))
```

The next step to making a proper body is to connect those pieces together with joints. jMonkeyEngine has a large array of joints available via `bullet`, such as Point2Point, Cone, Hinge, and a generic Six Degree of Freedom joint, with or without spring restitution.

Joints are treated a lot like proper senses, in that there is a top-level empty node named "joints" whose children each represent a joint.

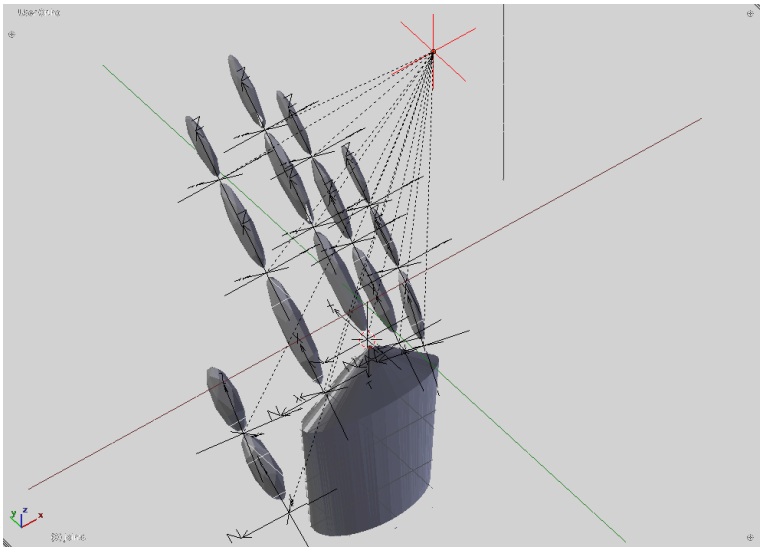

Figure 12: View of the hand model in Blender showing the main "joints" node (highlighted in yellow) and its children which each represent a joint in the hand. Each joint node has metadata specifying what sort of joint it is.

`CORTEX`'s procedure for binding the creature together with joints is as follows:

- Find the children of the "joints" node.
- Determine the two spatials the joint is meant to connect.
- Create the joint based on the meta-data of the empty node.

The higher order function `sense-nodes` from `cortex.sense` simplifies finding the joints based on their parent "joints" node.

Listing 3: Retrieving the children empty nodes from a single named empty node is a common pattern in `CORTEX` further instances of this technique for the senses will be omitted

```
(defn sense-nodes
  "For some senses there is a special empty blender node whose
   children are considered markers for an instance of that sense. This
   function generates functions to find those children, given the name
   of the special parent node."
  [parent-name]
  (fn [#^Node creature]
    (if-let [sense-node (.getChild creature parent-name)]
      (seq (.getChildren sense-node)) [])))

(def
  ^{:doc "Return the children of the creature's \"joints\" node."
    :arglists '([creature])}
  joints
  (sense-nodes "joints"))
```

To find a joint's targets, CORTEX creates a small cube, centered around the empty-node, and grows the cube exponentially until it intersects two physical objects. The objects are ordered according to the joint's rotation, with the first one being the object that has more negative coordinates in the joint's reference frame. Since the objects must be physical, the empty-node itself escapes detection. Because the objects must be physical, `joint-targets` must be called *after* `physical!` is called.

Listing 4: Program to find the targets of a joint node by exponentially growth of a search cube.

```
(defn joint-targets
  "Return the two closest two objects to the joint object, ordered
  from bottom to top according to the joint's rotation."
  [#^Node parts #^Node joint]
  (loop [radius (float 0.01)]
    (let [results (CollisionResults.)]
      (.collideWith
       parts
       (BoundingBox. (.getWorldTranslation joint)
                     radius radius radius) results)
      (let [targets
            (distinct
             (map  #(.getGeometry %) results))]
        (if (>= (count targets) 2)
          (sort-by
           #(let [joint-ref-frame-position
                  (jme-to-blender
                   (.mult
                    (.inverse (.getWorldRotation joint))
```

```
                (.subtract (.getWorldTranslation %)
                           (.getWorldTranslation joint))))]
           (.dot (Vector3f. 1 1 1) joint-ref-frame-position))
         (take 2 targets))
        (recur (float (* radius 2))))))))
```

Once CORTEX finds all joints and targets, it creates them using a dispatch on the meta-data of each joint node.

Listing 5: Program to dispatch on blender metadata and create joints suitable for physical simulation.

```
(defmulti joint-dispatch
  "Translate blender pseudo-joints into real JME joints."
  (fn [constraints & _]
    (:type constraints)))

(defmethod joint-dispatch :point
  [constraints control-a control-b pivot-a pivot-b rotation]
  (doto (SixDofJoint. control-a control-b pivot-a pivot-b false)
    (.setLinearLowerLimit Vector3f/ZERO)
    (.setLinearUpperLimit Vector3f/ZERO)))

(defmethod joint-dispatch :hinge
  [constraints control-a control-b pivot-a pivot-b rotation]
  (let [axis (if-let [axis (:axis constraints)] axis Vector3f/UNIT_X)
        [limit-1 limit-2] (:limit constraints)
        hinge-axis (.mult rotation (blender-to-jme axis))]
    (doto (HingeJoint. control-a control-b pivot-a pivot-b
                       hinge-axis hinge-axis)
      (.setLimit limit-1 limit-2))))

(defmethod joint-dispatch :cone
  [constraints control-a control-b pivot-a pivot-b rotation]
  (let [limit-xz (:limit-xz constraints)
        limit-xy (:limit-xy constraints)
        twist    (:twist constraints)]
    (doto (ConeJoint. control-a control-b pivot-a pivot-b
                      rotation rotation)
      (.setLimit (float limit-xz) (float limit-xy)
                 (float twist)))))
```

All that is left for joints is to combine the above pieces into something that can operate on the collection of nodes that a blender file represents.

Listing 6: Program to completely create a joint given information from a blender file.

```
(defn connect
  "Create a joint between 'obj-a and 'obj-b at the location of
  'joint. The type of joint is determined by the metadata on 'joint.

   Here are some examples:
   {:type :point}
   {:type :hinge  :limit [0 (/ Math/PI 2)] :axis (Vector3f. 0 1 0)}
   (:axis defaults to (Vector3f. 1 0 0) if not provided for hinge joints)

   {:type :cone :limit-xz 0]
                :limit-xy 0]
                :twist 0]}   (use XZY rotation mode in blender!)"
  [#^Node obj-a #^Node obj-b #^Node joint]
  (let [control-a (.getControl obj-a RigidBodyControl)
        control-b (.getControl obj-b RigidBodyControl)
        joint-center (.getWorldTranslation joint)
        joint-rotation (.toRotationMatrix (.getWorldRotation joint))
        pivot-a (world-to-local obj-a joint-center)
        pivot-b (world-to-local obj-b joint-center)]
    (if-let
        [constraints (map-vals eval (read-string (meta-data joint "joint")))]
      ;; A side-effect of creating a joint registers
      ;; it with both physics objects which in turn
      ;; will register the joint with the physics system
      ;; when the simulation is started.
        (joint-dispatch constraints
                        control-a control-b
                        pivot-a pivot-b
                        joint-rotation))))
```

In general, whenever CORTEX exposes a sense (or in this case physicality), it provides a function of the type `sense!`, which takes in a collection of nodes and augments it to support that sense. The function returns any controls necessary to use that sense. In this case `body!` creates a physical body and returns no control functions.

Listing 7: Program to give joints to a creature.

```
(defn joints!
  "Connect the solid parts of the creature with physical joints. The
   joints are taken from the \"joints\" node in the creature."
  [#^Node creature]
  (dorun
   (map
```

```
    (fn [joint]
      (let [[obj-a obj-b] (joint-targets creature joint)]
        (connect obj-a obj-b joint)))
    (joints creature))))
(defn body!
  "Endow the creature with a physical body connected with joints.  The
   particulars of the joints and the masses of each body part are
   determined in blender."
  [#^Node creature]
  (physical! creature)
  (joints! creature))
```

All of the code you have just seen amounts to only 130 lines, yet because it builds on top of Blender and jMonkeyEngine3, those few lines pack quite a punch!

The hand from figure 12, which was modeled after my own right hand, can now be given joints and simulated as a creature.

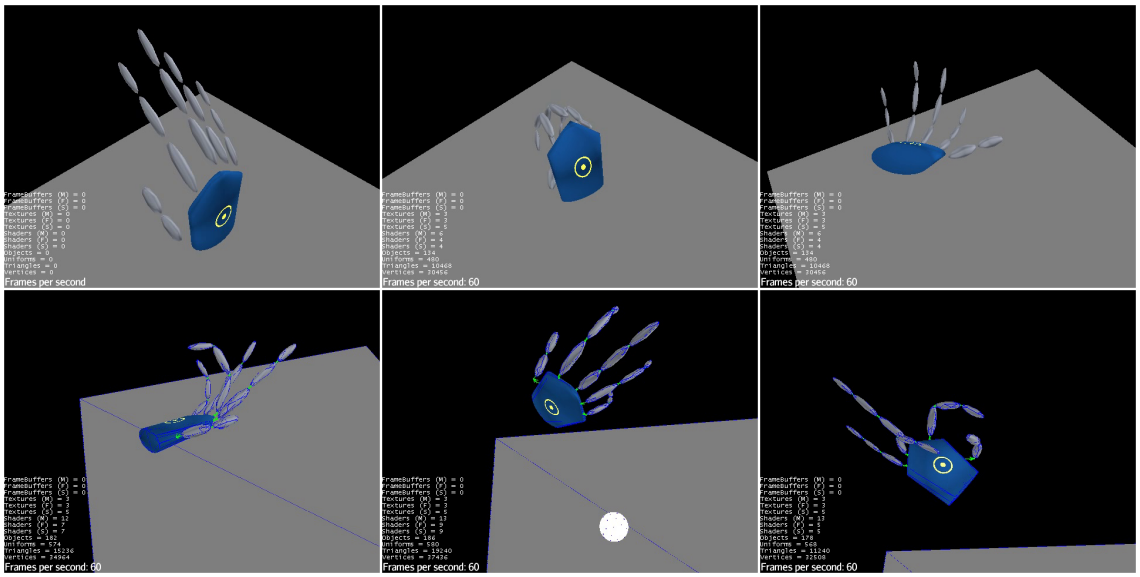

Figure 13: With the ability to create physical creatures from blender, CORTEX gets one step closer to becoming a full creature simulation environment.

2.8 Sight reuses standard video game components...

Vision is one of the most important senses for humans, so I need to build a simulated sense of vision for my AI. I will do this with simulated eyes. Each eye can be independently moved and should see its own version of the world depending on where it is.

Making these simulated eyes a reality is simple because jMonkeyEngine already contains extensive support for multiple views of the same 3D simulated world. The reason jMonkeyEngine has this support is because the support is necessary to create games with split-screen views. Multiple views are also used to create efficient pseudo-reflections by rendering the scene from a certain perspective and then projecting it back onto a surface in the 3D world.

Figure 14: jMonkeyEngine supports multiple views to enable split-screen games, like GoldenEye, which was one of the first games to use split-screen views.

A Brief Description of jMonkeyEngine's Rendering Pipeline

jMonkeyEngine allows you to create a `ViewPort`, which represents a view of the simulated world. You can create as many of these as you want. Every frame, the `RenderManager` iterates through each `ViewPort`, rendering the scene in the GPU. For each `ViewPort` there is a `FrameBuffer` which represents the rendered image in the GPU.

Each `ViewPort` can have any number of attached `SceneProcessor` objects, which are called every time a new frame is rendered. A `SceneProcessor` receives its `ViewPort's` `FrameBuffer` and can do whatever it wants to the data. Often this consists of invoking GPU specific operations on the rendered image. The `SceneProcessor` can also copy the GPU image data to RAM and process it with the CPU.

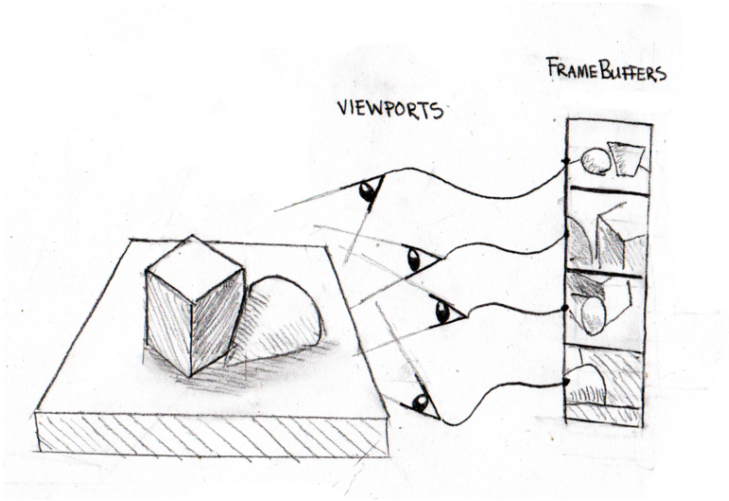

Figure 15: `ViewPorts` are cameras in the world. During each frame, the `RenderManager` records a snapshot of what each view is currently seeing; these snapshots are `FrameBuffer` objects.

Appropriating Views for Vision

Each eye in the simulated creature needs its own `ViewPort` so that it can see the world from its own perspective. To this `ViewPort`, I add a `SceneProcessor` that feeds the visual data to any arbitrary continuation function for further processing. That continuation function may perform both CPU and GPU operations on the data. To make this easy for the continuation function, the `SceneProcessor` maintains appropriately sized buffers in RAM to hold the data. It does not do any copying from the GPU to the CPU itself because it is a slow operation.

Listing 8: Function to make the rendered scene in jMonkeyEngine available for further processing.

```
(defn vision-pipeline
  "Create a SceneProcessor object which wraps a vision processing
  continuation function. The continuation is a function that takes
  [#^Renderer r #^FrameBuffer fb #^ByteBuffer b #^BufferedImage bi],
  each of which has already been appropriately sized."
  [continuation]
  (let [byte-buffer (atom nil)
        renderer (atom nil)
        image (atom nil)]
  (proxy [SceneProcessor] []
    (initialize
```

```
     [renderManager viewPort]
     (let [cam (.getCamera viewPort)
           width (.getWidth cam)
           height (.getHeight cam)]
       (reset! renderer (.getRenderer renderManager))
       (reset! byte-buffer
             (BufferUtils/createByteBuffer
              (* width height 4)))
        (reset! image (BufferedImage.
                       width height
                       BufferedImage/TYPE_4BYTE_ABGR))))
    (isInitialized [] (not (nil? @byte-buffer)))
    (reshape [_ _ _])
    (preFrame [_])
    (postQueue [_])
    (postFrame
     [#^FrameBuffer fb]
     (.clear @byte-buffer)
     (continuation @renderer fb @byte-buffer @image))
    (cleanup [])))
```

The continuation function given to `vision-pipeline` above will be given a `Renderer` and three containers for image data. The `FrameBuffer` references the GPU image data, but the pixel data can not be used directly on the CPU. The `ByteBuffer` and `BufferedImage` are initially "empty" but are sized to hold the data in the `FrameBuffer`. I call transferring the GPU image data to the CPU structures "mixing" the image data.

Optical sensor arrays are described with images and referenced with metadata

The vision pipeline described above handles the flow of rendered images. Now, `CORTEX` needs simulated eyes to serve as the source of these images.

An eye is described in blender in the same way as a joint. They are zero dimensional empty objects with no geometry whose local coordinate system determines the orientation of the resulting eye. All eyes are children of a parent node named "eyes" just as all joints have a parent named "joints". An eye binds to the nearest physical object with `bind-sense`.

Listing 9: Here, the camera is created based on metadata on the eye-node and attached to the nearest physical object with `bind-sense`

```
(defn add-eye!
  "Create a Camera centered on the current position of 'eye which
   follows the closest physical node in 'creature. The camera will
   point in the X direction and use the Z vector as up as determined
```

```
   by the rotation of these vectors in blender coordinate space. Use
   XZY rotation for the node in blender."
  [#^Node creature #^Spatial eye]
  (let [target (closest-node creature eye)
        [cam-width cam-height]
        ;;[640 480] ;; graphics card on laptop doesn't support
                    ;; arbitrary dimensions.
        (eye-dimensions eye)
        cam (Camera. cam-width cam-height)
        rot (.getWorldRotation eye)]
    (.setLocation cam (.getWorldTranslation eye))
    (.lookAtDirection
     cam                          ; this part is not a mistake and
     (.mult rot Vector3f/UNIT_X)  ; is consistent with using Z in
     (.mult rot Vector3f/UNIT_Y)) ; blender as the UP vector.
    (.setFrustumPerspective
     cam (float 45)
     (float (/ (.getWidth cam) (.getHeight cam)))
     (float 1)
     (float 1000))
    (bind-sense target cam) cam))
```

Simulated Retina

An eye is a surface (the retina) which contains many discrete sensors to detect light. These sensors can have different light-sensing properties. In humans, each discrete sensor is sensitive to red, blue, green, or gray. These different types of sensors can have different spatial distributions along the retina. In humans, there is a fovea in the center of the retina which has a very high density of color sensors, and a blind spot which has no sensors at all. Sensor density decreases in proportion to distance from the fovea.

I want to be able to model any retinal configuration, so my eye-nodes in blender contain metadata pointing to images that describe the precise position of the individual sensors using white pixels. The meta-data also describes the precise sensitivity to light that the sensors described in the image have. An eye can contain any number of these images. For example, the metadata for an eye might look like this:

```
{0xFF0000 "Models/test-creature/retina-small.png"}
```

Together, the number 0xFF0000 and the image above describe the placement of red-sensitive sensory elements.

Meta-data to very crudely approximate a human eye might be something like this:

```
(let [retinal-profile "Models/test-creature/retina-small.png"]
  {0xFF0000 retinal-profile
```

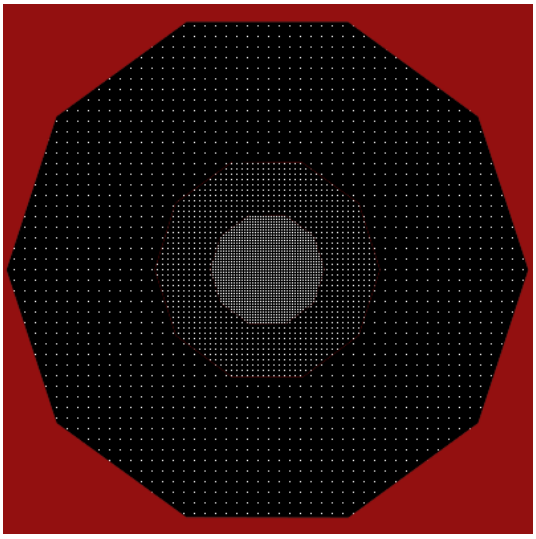

Figure 16: An example retinal profile image. White pixels are photo-sensitive elements. The distribution of white pixels is denser in the middle and falls off at the edges and is inspired by the human retina.

```
  0x00FF00 retinal-profile
  0x0000FF retinal-profile
  0xFFFFFF retinal-profile})
```

The numbers that serve as keys in the map determine a sensor's relative sensitivity to the channels red, green, and blue. These sensitivity values are packed into an integer in the order `|_|R|G|B|` in 8-bit fields. The RGB values of a pixel in the image are added together with these sensitivities as linear weights. Therefore, 0xFF0000 means sensitive to red only while 0xFFFFFF means sensitive to all colors equally (gray).

Listing 10: This is the core of vision in `CORTEX`. A given eye node is converted into a function that returns visual information from the simulation.

```
(defn vision-kernel
  "Returns a list of functions, each of which will return a color
   channel's worth of visual information when called inside a running
   simulation."
  [#^Node creature #^Spatial eye & {skip :skip :or {skip 0}}]
  (let [retinal-map (retina-sensor-profile eye)
        camera (add-eye! creature eye)
        vision-image
        (atom
         (BufferedImage. (.getWidth camera)
```

```
                    (.getHeight camera)
                    BufferedImage/TYPE_BYTE_BINARY))
      register-eye!
      (runonce
       (fn [world]
         (add-camera!
          world camera
          (let [counter  (atom 0)]
            (fn [r fb bb bi]
              (if (zero? (rem (swap! counter inc) (inc skip)))
                (reset! vision-image
                        (BufferedImage! r fb bb bi)))))))))]
   (vec
    (map
     (fn [[key image]]
       (let [whites (white-coordinates image)
             topology (vec (collapse whites))
             sensitivity (sensitivity-presets key key)]
         (attached-viewport.
          (fn [world]
            (register-eye! world)
            (vector
             topology
             (vec
              (for [[x y] whites]
                (pixel-sense
                 sensitivity
                 (.getRGB @vision-image x y))))))
          register-eye!)))
      retinal-map))))
```

Note that since each of the functions generated by `vision-kernel` shares the same `register-eye!` function, the eye will be registered only once the first time any of the functions from the list returned by `vision-kernel` is called. Each of the functions returned by `vision-kernel` also allows access to the `Viewport` through which it receives images.

All the hard work has been done; all that remains is to apply `vision-kernel` to each eye in the creature and gather the results into one list of functions.

Listing 11: With `vision!`, `CORTEX` is already a fine simulation environment for experimenting with different types of eyes.

```
(defn vision!
  "Returns a list of functions, each of which returns visual sensory
   data when called inside a running simulation."
```

```
[#^Node creature & {skip :skip :or {skip 0}}]
(reduce
 concat
 (for [eye (eyes creature)]
   (vision-kernel creature eye))))
```

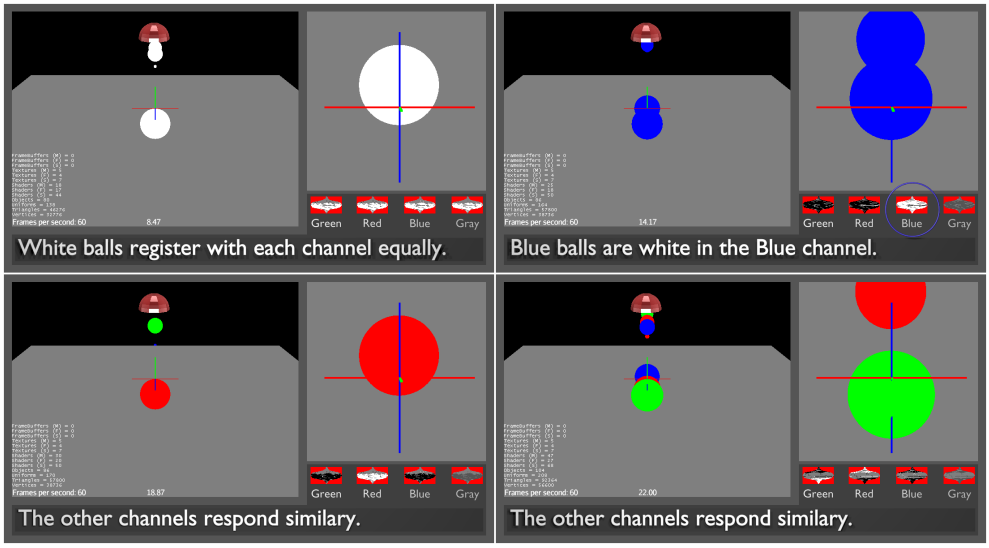

Figure 17: Simulated vision with a test creature and the human-like eye approximation. Notice how each channel of the eye responds differently to the differently colored balls.

The vision code is not much more complicated than the body code, and enables multiple further paths for simulated vision. For example, it is quite easy to create bifocal vision – you just make two eyes next to each other in blender! It is also possible to encode vision transforms in the retinal files. For example, the human like retina file in figure 35 approximates a log-polar transform.

This vision code has already been absorbed by the jMonkeyEngine community and is now (in modified form) part of a system for capturing in-game video to a file.

2.9 ...but hearing must be built from scratch

At the end of this section I will have simulated ears that work the same way as the simulated eyes in the last section. I will be able to place any number of ear-nodes in a blender file, and they will bind to the closest physical object and follow it as it moves around. Each ear will provide access to the sound data it picks up between every frame.

Hearing is one of the more difficult senses to simulate, because there is less support for obtaining the actual sound data that is processed by jMonkeyEngine3. There is no "split-screen" support for rendering sound from different points of view, and there is no way to directly access the rendered sound data.

CORTEX's hearing is unique because it does not have any limitations compared to other simulation environments. As far as I know, there is no other system that supports multiple listeners, and the sound demo at the end of this section is the first time it's been done in a video game environment.

Brief Description of jMonkeyEngine's Sound System

jMonkeyEngine's sound system works as follows:

- jMonkeyEngine uses the `AppSettings` for the particular application to determine what sort of `AudioRenderer` should be used.
- Although some support is provided for multiple AudioRendering backends, jMonkeyEngine at the time of this writing will either pick no `AudioRenderer` at all, or the `LwjglAudioRenderer`.
- jMonkeyEngine tries to figure out what sort of system you're running and extracts the appropriate native libraries.
- The `LwjglAudioRenderer` uses the LWJGL (LightWeight Java Game Library) bindings to interface with a C library called OpenAL
- OpenAL renders the 3D sound and feeds the rendered sound directly to any of various sound output devices with which it knows how to communicate.

A consequence of this is that there's no way to access the actual sound data produced by OpenAL. Even worse, OpenAL only supports one *listener* (it renders sound data from only one perspective), which normally isn't a problem for games, but becomes a problem when trying to make multiple AI creatures that can each hear the world from a different perspective.

To make many AI creatures in jMonkeyEngine that can each hear the world from their own perspective, or to make a single creature with many ears, it is necessary to go all the way back to OpenAL and implement support for simulated hearing there.

Extending OpenAl

Extending OpenAL to support multiple listeners requires 500 lines of C code and is too hairy to mention here. Instead, I will show a small amount of extension code and go over the high level strategy. Full source is of course available with the CORTEX distribution if you're interested.

OpenAL goes to great lengths to support many different systems, all with different sound capabilities and interfaces. It accomplishes this difficult task by providing code for many different sound backends in pseudo-objects called *Devices*. There's a device for the Linux Open Sound System and the Advanced Linux Sound Architecture, there's one for Direct Sound on Windows, and there's even one for Solaris. OpenAL solves the problem of platform independence by providing all these Devices.

Wrapper libraries such as LWJGL are free to examine the system on which they are running and then select an appropriate device for that system.

There are also a few "special" devices that don't interface with any particular system. These include the Null Device, which doesn't do anything, and the Wave Device, which writes whatever sound it receives to a file, if everything has been set up correctly when configuring OpenAL.

Actual mixing (Doppler shift and distance.environment-based attenuation) of the sound data happens in the Devices, and they are the only point in the sound rendering process where this data is available.

Therefore, in order to support multiple listeners, and get the sound data in a form that the AIs can use, it is necessary to create a new Device which supports this feature.

Adding a device to OpenAL is rather tricky – there are five separate files in the OpenAL source tree that must be modified to do so. I named my device the "Multiple Audio Send" Device, or Send Device for short, since it sends audio data back to the calling application like an Aux-Send cable on a mixing board.

The main idea behind the Send device is to take advantage of the fact that LWJGL only manages one *context* when using OpenAL. A *context* is like a container that holds samples and keeps track of where the listener is. In order to support multiple listeners, the Send device identifies the LWJGL context as the master context, and creates any number of slave contexts to represent additional listeners. Every time the device renders sound, it synchronizes every source from the master LWJGL context to the slave contexts. Then, it renders each context separately, using a different listener for each one. The rendered sound is made available via JNI to jMonkeyEngine.

Switching between contexts is not the normal operation of a Device, and one of the problems with doing so is that a Device normally keeps around a few pieces of state such as the ClickRemoval array above which will become corrupted if the contexts are not rendered in parallel. The solution is to create a copy of this normally global device

state for each context, and copy it back and forth into and out of the actual device state whenever a context is rendered.

The core of the Send device is the `syncSources` function, which does the job of copying all relevant data from one context to another.

Listing 12: Program for extending `OpenAL` to support multiple listeners via context copying/switching.

```
void syncSources(ALsource *masterSource, ALsource *slaveSource,
                 ALCcontext *masterCtx, ALCcontext *slaveCtx){
  ALuint master = masterSource->source;
  ALuint slave = slaveSource->source;
  ALCcontext *current = alcGetCurrentContext();

  syncSourcef(master,slave,masterCtx,slaveCtx,AL_PITCH);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_GAIN);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_MAX_DISTANCE);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_ROLLOFF_FACTOR);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_REFERENCE_DISTANCE);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_MIN_GAIN);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_MAX_GAIN);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_CONE_OUTER_GAIN);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_CONE_INNER_ANGLE);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_CONE_OUTER_ANGLE);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_SEC_OFFSET);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_SAMPLE_OFFSET);
  syncSourcef(master,slave,masterCtx,slaveCtx,AL_BYTE_OFFSET);

  syncSource3f(master,slave,masterCtx,slaveCtx,AL_POSITION);
  syncSource3f(master,slave,masterCtx,slaveCtx,AL_VELOCITY);
  syncSource3f(master,slave,masterCtx,slaveCtx,AL_DIRECTION);

  syncSourcei(master,slave,masterCtx,slaveCtx,AL_SOURCE_RELATIVE);
  syncSourcei(master,slave,masterCtx,slaveCtx,AL_LOOPING);

  alcMakeContextCurrent(masterCtx);
  ALint source_type;
  alGetSourcei(master, AL_SOURCE_TYPE, &source_type);

  // Only static sources are currently synchronized!
  if (AL_STATIC == source_type){
    ALint master_buffer;
    ALint slave_buffer;
    alGetSourcei(master, AL_BUFFER, &master_buffer);
    alcMakeContextCurrent(slaveCtx);
    alGetSourcei(slave, AL_BUFFER, &slave_buffer);
```

```
    if (master_buffer != slave_buffer){
      alSourcei(slave, AL_BUFFER, master_buffer);
    }
  }

  // Synchronize the state of the two sources.
  alcMakeContextCurrent(masterCtx);
  ALint masterState;
  ALint slaveState;

  alGetSourcei(master, AL_SOURCE_STATE, &masterState);
  alcMakeContextCurrent(slaveCtx);
  alGetSourcei(slave, AL_SOURCE_STATE, &slaveState);

  if (masterState != slaveState){
    switch (masterState){
    case AL_INITIAL : alSourceRewind(slave); break;
    case AL_PLAYING : alSourcePlay(slave);   break;
    case AL_PAUSED  : alSourcePause(slave);  break;
    case AL_STOPPED : alSourceStop(slave);   break;
    }
  }
  // Restore whatever context was previously active.
  alcMakeContextCurrent(current);
}
```

With this special context-switching device, and some ugly JNI bindings that are not worth mentioning, CORTEX gains the ability to access multiple sound streams from OpenAL.

Listing 13: Program to create an ear from a blender empty node. The ear follows around the nearest physical object and passes all sensory data to a continuation function.

```
(defn add-ear!
  "Create a Listener centered on the current position of 'ear
   which follows the closest physical node in 'creature and
   sends sound data to 'continuation."
  [#^Application world #^Node creature #^Spatial ear continuation]
  (let [target (closest-node creature ear)
        lis (Listener.)
        audio-renderer (.getAudioRenderer world)
        sp (hearing-pipeline continuation)]
    (.setLocation lis (.getWorldTranslation ear))
    (.setRotation lis (.getWorldRotation ear))
    (bind-sense target lis)
    (update-listener-velocity! target lis)
```

```
    (.addListener audio-renderer lis)
    (.registerSoundProcessor audio-renderer lis sp)))
```

The Send device, unlike most of the other devices in OpenAL, does not render sound unless asked. This enables the system to slow down or speed up depending on the needs of the AIs who are using it to listen. If the device tried to render samples in real-time, a complicated AI whose mind takes 100 seconds of computer time to simulate 1 second of AI-time would miss almost all of the sound in its environment!

Listing 14: Program to enable arbitrary hearing in CORTEX

```
(defn hearing-kernel
  "Returns a function which returns auditory sensory data when called
   inside a running simulation."
  [#^Node creature #^Spatial ear]
  (let [hearing-data (atom [])
        register-listener!
        (runonce
         (fn [#^Application world]
           (add-ear!
            world creature ear
            (comp #(reset! hearing-data %)
                  byteBuffer->pulse-vector))))]
    (fn [#^Application world]
      (register-listener! world)
      (let [data @hearing-data
            topology
            (vec (map #(vector % 0) (range 0 (count data))))]
        [topology data]))))

(defn hearing!
  "Endow the creature in a particular world with the sense of
   hearing. Will return a sequence of functions, one for each ear,
   which when called will return the auditory data from that ear."
  [#^Node creature]
  (for [ear (ears creature)]
    (hearing-kernel creature ear)))
```

Armed with these functions, CORTEX is able to test possibly the first ever instance of multiple listeners in a video game engine based simulation!

Listing 15: Here a simple creature responds to sound by changing its color from gray to green when the total volume goes over a threshold.

```
/**
 * Respond to sound!  This is the brain of an AI entity that
```

```
 * hears its surroundings and reacts to them.
 */
public void process(ByteBuffer audioSamples,
                    int numSamples, AudioFormat format) {
    audioSamples.clear();
    byte[] data = new byte[numSamples];
    float[] out = new float[numSamples];
    audioSamples.get(data);
    FloatSampleTools.
        byte2floatInterleaved
        (data, 0, out, 0, numSamples/format.getFrameSize(), format);

    float max = Float.NEGATIVE_INFINITY;
    for (float f : out){if (f > max) max = f;}
    audioSamples.clear();

    if (max > 0.1){
        entity.getMaterial().setColor("Color", ColorRGBA.Green);
    }
    else {
        entity.getMaterial().setColor("Color", ColorRGBA.Gray);
    }
```

This system of hearing has also been co-opted by the jMonkeyEngine3 community and is used to record audio for demo videos.

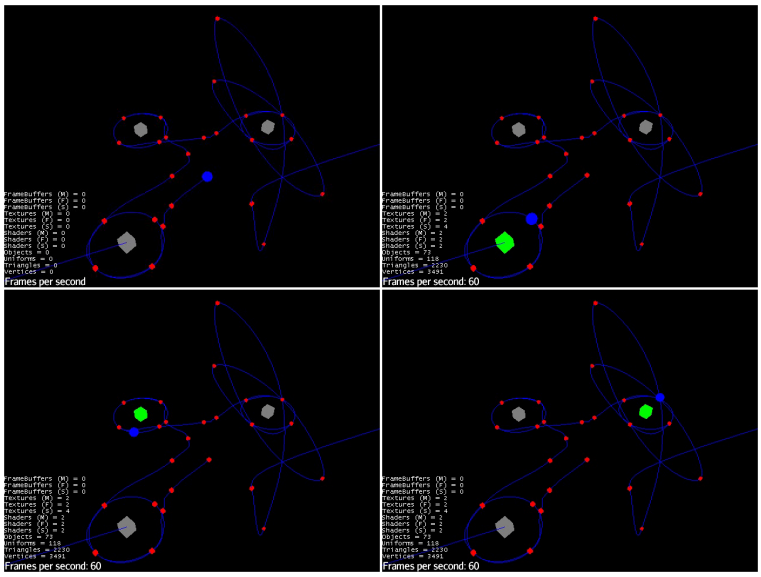

Figure 18: First ever simulation of multiple listeners in CORTEX. Each cube is a creature which processes sound data with the `process` function from listing 15. the ball is constantly emitting a pure tone of constant volume. As it approaches the cubes, they each change color in response to the sound.

2.10 Hundreds of hair-like elements provide a sense of touch

Touch is critical to navigation and spatial reasoning and as such I need a simulated version of it to give to my AI creatures.

Human skin has a wide array of touch sensors, each of which specialize in detecting different vibrational modes and pressures. These sensors can integrate a vast expanse of skin (i.e. your entire palm), or a tiny patch of skin at the tip of your finger. The hairs of the skin help detect objects before they even come into contact with the skin proper.

However, touch in my simulated world can not exactly correspond to human touch because my creatures are made out of completely rigid segments that don't deform like human skin.

Instead of measuring deformation or vibration, I surround each rigid part with a plenitude of hair-like objects (*feelers*) which do not interact with the physical world. Physical objects can pass through them with no effect. The feelers are able to tell when other objects pass through them, and they constantly report how much of their extent is covered. So even though the creature's body parts do not deform, the feelers create a margin around those body parts which achieves a sense of touch which is a hybrid between a human's sense of deformation and sense from hairs.

Implementing touch in jMonkeyEngine follows a different technical route than vision and hearing. Those two senses piggybacked off jMonkeyEngine's 3D audio and video rendering subsystems. To simulate touch, I use jMonkeyEngine's physics system to execute many small collision detections, one for each feeler. The placement of the feelers is determined by a UV-mapped image which shows where each feeler should be on the 3D surface of the body.

Defining Touch Meta-Data in Blender

Each geometry can have a single UV map which describes the position of the feelers which will constitute its sense of touch. This image path is stored under the "touch" key. The image itself is black and white, with black meaning a feeler length of 0 (no feeler is present) and white meaning a feeler length of `scale`, which is a float stored under the key "scale".

Listing 16: Touch does not use empty nodes, to store metadata, because the meta-data of each solid part of a creature's body is sufficient.

```
(defn tactile-sensor-profile
  "Return the touch-sensor distribution image in BufferedImage format,
   or nil if it does not exist."
  [#^Geometry obj]
  (if-let [image-path (meta-data obj "touch")]
    (load-image image-path)))

(defn tactile-scale
  "Return the length of each feeler. Default scale is 0.01
  jMonkeyEngine units."
  [#^Geometry obj]
  (if-let [scale (meta-data obj "scale")]
    scale 0.1))
```

Here is an example of a UV-map which specifies the position of touch sensors along the surface of the upper segment of a fingertip.

Implementation Summary

To simulate touch there are three conceptual steps. For each solid object in the creature, you first have to get UV image and scale parameter which define the position and length of the feelers. Then, you use the triangles which comprise the mesh and the UV data stored in the mesh to determine the world-space position and orientation of each feeler. Then once every frame, update these positions and orientations to match the current

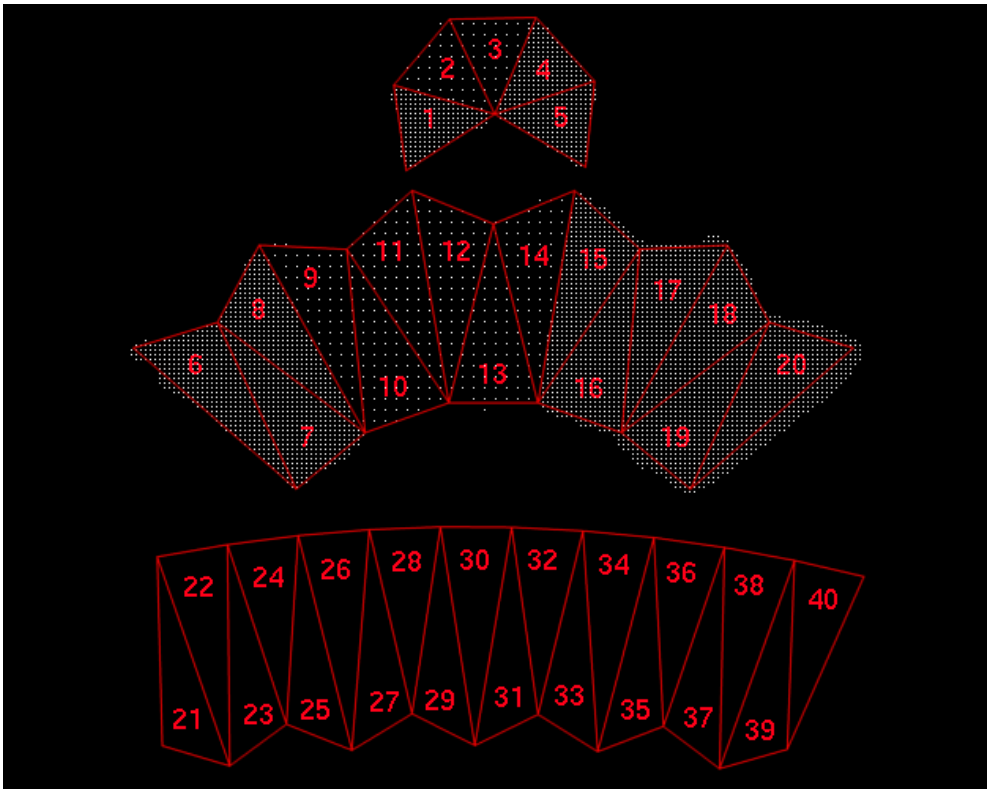

Figure 19: This is the tactile-sensor-profile for the upper segment of a fingertip. It defines regions of high touch sensitivity (where there are many white pixels) and regions of low sensitivity (where white pixels are sparse).

position and orientation of the object, and use physics collision detection to gather tactile data.

Extracting the meta-data has already been described. The third step, physics collision detection, is handled in `touch-kernel`. Translating the positions and orientations of the feelers from the UV-map to world-space is itself a three-step process.

- Find the triangles which make up the mesh in pixel-space and in world-space. (`triangles`, `pixel-triangles`).
- Find the coordinates of each feeler in world-space. These are the origins of the feelers. (`feeler-origins`).
- Calculate the normals of the triangles in world space, and add them to each of the origins of the feelers. These are the normalized coordinates of the tips of the feelers.

(feeler-tips).

Triangle Math

The rigid objects which make up a creature have an underlying Geometry, which is a Mesh plus a Material and other important data involved with displaying the object.

A Mesh is composed of Triangles, and each Triangle has three vertices which have coordinates in world space and UV space.

Here, triangles gets all the world-space triangles which comprise a mesh, while pixel-triangles gets those same triangles expressed in pixel coordinates (which are UV coordinates scaled to fit the height and width of the UV image).

Listing 17: Programs to extract triangles from a geometry and get their vertices in both world and UV-coordinates.

```
(defn triangle
  "Get the triangle specified by triangle-index from the mesh."
  [#^Geometry geo triangle-index]
  (triangle-seq
   (let [scratch (Triangle.)]
     (.getTriangle (.getMesh geo) triangle-index scratch) scratch)))

(defn triangles
  "Return a sequence of all the Triangles which comprise a given
   Geometry."
  [#^Geometry geo]
  (map (partial triangle geo) (range (.getTriangleCount (.getMesh geo)))))

(defn triangle-vertex-indices
  "Get the triangle vertex indices of a given triangle from a given
   mesh."
  [#^Mesh mesh triangle-index]
  (let [indices (int-array 3)]
    (.getTriangle mesh triangle-index indices)
    (vec indices)))

    (defn vertex-UV-coord
  "Get the UV-coordinates of the vertex named by vertex-index"
  [#^Mesh mesh vertex-index]
  (let [UV-buffer
        (.getData
         (.getBuffer
          mesh
          VertexBuffer$Type/TexCoord))]
```

```
    [(.get UV-buffer (* vertex-index 2))
     (.get UV-buffer (+ 1 (* vertex-index 2)))]))

(defn pixel-triangle [#^Geometry geo image index]
  (let [mesh (.getMesh geo)
        width (.getWidth image)
        height (.getHeight image)]
    (vec (map (fn [[u v]] (vector (* width u) (* height v)))
              (map (partial vertex-UV-coord mesh)
                   (triangle-vertex-indices mesh index))))))

(defn pixel-triangles
  "The pixel-space triangles of the Geometry, in the same order as
   (triangles geo)"
  [#^Geometry geo image]
  (let [height (.getHeight image)
        width (.getWidth image)]
    (map (partial pixel-triangle geo image)
         (range (.getTriangleCount (.getMesh geo))))))
```

The Affine Transform from one Triangle to Another

`pixel-triangles` gives us the mesh triangles expressed in pixel coordinates and `triangles` gives us the mesh triangles expressed in world coordinates. The tactile-sensor-profile gives the position of each feeler in pixel-space. In order to convert pixel-space coordinates into world-space coordinates we need something that takes coordinates on the surface of one triangle and gives the corresponding coordinates on the surface of another triangle.

Triangles are affine, which means any triangle can be transformed into any other by a combination of translation, scaling, and rotation. The affine transformation from one triangle to another is readily computable if the triangle is expressed in terms of a $4x4$ matrix.

$$\begin{bmatrix} x_1 & x_2 & x_3 & n_x \\ y_1 & y_2 & y_3 & n_y \\ z_1 & z_2 & z_3 & n_z \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Here, the first three columns of the matrix are the vertices of the triangle. The last column is the right-handed unit normal of the triangle.

With two triangles T_1 and T_2 each expressed as a matrix like above, the affine transform from T_1 to T_2 is $T_2 T_1^{-1}$.

The clojure code below recapitulates the formulas above, using jMonkeyEngine's `Matrix4f` objects, which can describe any affine transformation.

Listing 18: Program to interpret triangles as affine transforms.

```
(defn triangle->matrix4f
  "Converts the triangle into a 4x4 matrix: The first three columns
   contain the vertices of the triangle; the last contains the unit
   normal of the triangle. The bottom row is filled with 1s."
  [#^Triangle t]
  (let [mat (Matrix4f.)
        [vert-1 vert-2 vert-3]
        (mapv #(.get t %) (range 3))
        unit-normal (do (.calculateNormal t)(.getNormal t))
        vertices [vert-1 vert-2 vert-3 unit-normal]]
    (dorun
     (for [row (range 4) col (range 3)]
       (do
         (.set mat col row (.get (vertices row) col))
         (.set mat 3 row 1)))) mat))

(defn triangles->affine-transform
  "Returns the affine transformation that converts each vertex in the
   first triangle into the corresponding vertex in the second
   triangle."
  [#^Triangle tri-1 #^Triangle tri-2]
  (.mult
   (triangle->matrix4f tri-2)
   (.invert (triangle->matrix4f tri-1))))
```

Triangle Boundaries

For efficiency's sake I will divide the tactile-profile image into small squares which inscribe each pixel-triangle, then extract the points which lie inside the triangle and map them to 3D-space using `triangle-transform` above. To do this I need a function, `convex-bounds` which finds the smallest box which inscribes a 2D triangle.

`inside-triangle?` determines whether a point is inside a triangle in 2D pixel-space.

Listing 19: Program to efficiently determine point inclusion in a triangle.

```
(defn convex-bounds
  "Returns the smallest square containing the given vertices, as a
   vector of integers [left top width height]."
  [verts]
  (let [xs (map first verts)
```

```
        ys (map second verts)
        x0 (Math/floor (apply min xs))
        y0 (Math/floor (apply min ys))
        x1 (Math/ceil (apply max xs))
        y1 (Math/ceil (apply max ys))]
    [x0 y0 (- x1 x0) (- y1 y0)]))

(defn same-side?
  "Given the points p1 and p2 and the reference point ref, is point p
  on the same side of the line that goes through p1 and p2 as ref is?"
  [p1 p2 ref p]
  (<=
   0
   (.dot
    (.cross (.subtract p2 p1) (.subtract p p1))
    (.cross (.subtract p2 p1) (.subtract ref p1)))))

(defn inside-triangle?
  "Is the point inside the triangle?"
  {:author "Dylan Holmes"}
  [#^Triangle tri #^Vector3f p]
  (let [[vert-1 vert-2 vert-3] [(.get1 tri) (.get2 tri) (.get3 tri)]]
    (and
     (same-side? vert-1 vert-2 vert-3 p)
     (same-side? vert-2 vert-3 vert-1 p)
     (same-side? vert-3 vert-1 vert-2 p))))
```

Feeler Coordinates

The triangle-related functions above make short work of calculating the positions and orientations of each feeler in world-space.

Listing 20: Program to get the coordinates of "feelers " in both world and UV-coordinates.

```
(defn feeler-pixel-coords
 "Returns the coordinates of the feelers in pixel space in lists, one
  list for each triangle, ordered in the same way as (triangles) and
  (pixel-triangles)."
 [#^Geometry geo image]
 (map
  (fn [pixel-triangle]
    (filter
     (fn [coord]
       (inside-triangle? (->triangle pixel-triangle)
                         (->vector3f coord)))
```

```
       (white-coordinates image (convex-bounds pixel-triangle))))
  (pixel-triangles geo image)))

(defn feeler-world-coords
 "Returns the coordinates of the feelers in world space in lists, one
  list for each triangle, ordered in the same way as (triangles) and
  (pixel-triangles)."
 [#^Geometry geo image]
 (let [transforms
       (map #(triangles->affine-transform
              (->triangle %1) (->triangle %2))
            (pixel-triangles geo image)
            (triangles geo))]
   (map (fn [transform coords]
          (map #(.mult transform (->vector3f %)) coords))
        transforms (feeler-pixel-coords geo image))))
```

Listing 21: Program to get the position of the base and tip of each "feeler"

```
(defn feeler-origins
  "The world space coordinates of the root of each feeler."
  [#^Geometry geo image]
   (reduce concat (feeler-world-coords geo image)))

(defn feeler-tips
  "The world space coordinates of the tip of each feeler."
  [#^Geometry geo image]
  (let [world-coords (feeler-world-coords geo image)
        normals
        (map
         (fn [triangle]
           (.calculateNormal triangle)
           (.clone (.getNormal triangle)))
         (map ->triangle (triangles geo)))]

    (mapcat (fn [origins normal]
              (map #(.add % normal) origins))
            world-coords normals)))

(defn touch-topology
  [#^Geometry geo image]
  (collapse (reduce concat (feeler-pixel-coords geo image))))
```

Simulated Touch

Now that the functions to construct feelers are complete, `touch-kernel` generates functions to be called from within a simulation that perform the necessary physics collisions to collect tactile data, and `touch!` recursively applies it to every node in the creature.

Listing 22: Efficient program to transform a ray from one position to another.

```
(defn set-ray [#^Ray ray #^Matrix4f transform
               #^Vector3f origin #^Vector3f tip]
  ;; Doing everything locally reduces garbage collection by enough to
  ;; be worth it.
  (.mult transform origin (.getOrigin ray))
  (.mult transform tip (.getDirection ray))
  (.subtractLocal (.getDirection ray) (.getOrigin ray))
  (.normalizeLocal (.getDirection ray)))
```

Listing 23: This is the core of touch in `CORTEX` each feeler follows the object it is bound to, reporting any collisions that may happen.

```
(defn touch-kernel
  "Constructs a function which will return tactile sensory data from
   'geo when called from inside a running simulation"
  [#^Geometry geo]
  (if-let
      [profile (tactile-sensor-profile geo)]
    (let [ray-reference-origins (feeler-origins geo profile)
          ray-reference-tips (feeler-tips geo profile)
          ray-length (tactile-scale geo)
          current-rays (map (fn [_] (Ray.)) ray-reference-origins)
          topology (touch-topology geo profile)
          correction (float (* ray-length -0.2))]
      ;; slight tolerance for very close collisions.
      (dorun
       (map (fn [origin tip]
              (.addLocal origin (.mult (.subtract tip origin)
                                       correction)))
            ray-reference-origins ray-reference-tips))
      (dorun (map #(.setLimit % ray-length) current-rays))
      (fn [node]
        (let [transform (.getWorldMatrix geo)]
          (dorun
           (map (fn [ray ref-origin ref-tip]
                  (set-ray ray transform ref-origin ref-tip))
                current-rays ray-reference-origins
```

```
            ray-reference-tips))
          (vector
           topology
           (vec
            (for [ray current-rays]
              (do
                (let [results (CollisionResults.)]
                  (.collideWith node ray results)
                  (let [touch-objects
                        (filter #(not (= geo (.getGeometry %)))
                                results)
                        limit (.getLimit ray)]
                    [(if (empty? touch-objects)
                       limit
                       (let [response
                             (apply min (map #(.getDistance %)
                                             touch-objects))]
                         (FastMath/clamp
                          (float
                           (if (> response limit) (float 0.0)
                               (+ response correction)))
                          (float 0.0)
                          limit)))
                     limit])))))))))
```

Armed with the `touch!` function, `CORTEX` becomes capable of giving creatures a sense of touch. A simple test is to create a cube that is outfitted with a uniform distribution of touch sensors. It can feel the ground and any balls that it touches.

Listing 24: `CORTEX` interface for creating touch in a simulated creature.

```
(defn touch!
  "Endow the creature with the sense of touch. Returns a sequence of
   functions, one for each body part with a tactile-sensor-profile,
   each of which when called returns sensory data for that body part."
  [#^Node creature]
  (filter
   (comp not nil?)
   (map touch-kernel
        (filter #(isa? (class %) Geometry)
                (node-seq creature)))))
```

The tactile-sensor-profile image for the touch cube is a simple cross with a uniform distribution of touch sensors:

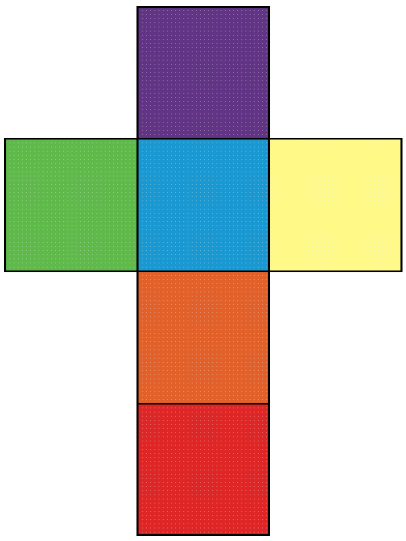

Figure 20: The touch profile for the touch-cube. Each pure white pixel defines a touch sensitive feeler.

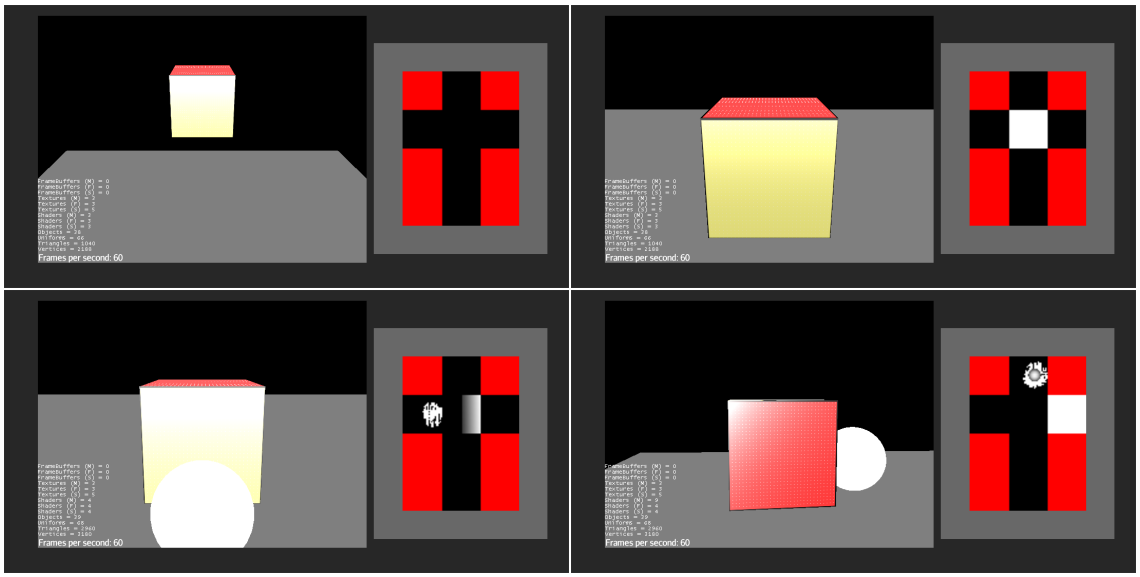

Figure 21: The touch cube reacts to cannonballs. The black, red, and white cross on the right is a visual display of the creature's touch. White means that it is feeling something strongly, black is not feeling anything, and gray is in-between. The cube can feel both the floor and the ball. Notice that when the ball causes the cube to tip, that the bottom face can still feel part of the ground.

2.11 Proprioception provides knowledge of your own body's position

Close your eyes, and touch your nose with your right index finger. How did you do it? You could not see your hand, and neither your hand nor your nose could use the sense of touch to guide the path of your hand. There are no sound cues, and Taste and Smell certainly don't provide any help. You know where your hand is without your other senses because of Proprioception.

Humans can sometimes loose this sense through viral infections or damage to the spinal cord or brain, and when they do, they loose the ability to control their own bodies without looking directly at the parts they want to move. In The Man Who Mistook His Wife for a Hat (Sacks 1998), a woman named Christina looses this sense and has to learn how to move by carefully watching her arms and legs. She describes proprioception as the "eyes of the body, the way the body sees itself".

Proprioception in humans is mediated by joint capsules, muscle spindles, and the Golgi tendon organs. These measure the relative positions of each body part by monitoring muscle strain and length.

It's clear that this is a vital sense for fluid, graceful movement. It's also particularly easy to implement in jMonkeyEngine.

My simulated proprioception calculates the relative angles of each joint from the rest position defined in the blender file. This simulates the muscle-spindles and joint capsules. I will deal with Golgi tendon organs, which calculate muscle strain, in the next section.

Helper functions

`absolute-angle` calculates the angle between two vectors, relative to a third axis vector. This angle is the number of radians you have to move counterclockwise around the axis vector to get from the first to the second vector. It is not commutative like a normal dot-product angle is.

The purpose of these functions is to build a system of angle measurement that is biologically plausible.

Listing 25: Program to measure angles along a vector

```
(defn right-handed?
  "true iff the three vectors form a right handed coordinate
   system. The three vectors do not have to be normalized or
   orthogonal."
  [vec1 vec2 vec3]
  (pos? (.dot (.cross vec1 vec2) vec3)))
```

```
(defn absolute-angle
  "The angle between 'vec1 and 'vec2 around 'axis. In the range
   [0 (* 2 Math/PI)]."
  [vec1 vec2 axis]
  (let [angle (.angleBetween vec1 vec2)]
    (if (right-handed? vec1 vec2 axis)
      angle (- (* 2 Math/PI) angle))))
```

Proprioception Kernel

Given a joint, `proprioception-kernel` produces a function that calculates the Euler angles between the objects the joint connects. The only tricky part here is making the angles relative to the joint's initial "straightness".

> Listing 26: Program to return biologically reasonable proprioceptive data for each joint.

```
(defn proprioception-kernel
  "Returns a function which returns proprioceptive sensory data when
  called inside a running simulation."
  [#^Node parts #^Node joint]
  (let [[obj-a obj-b] (joint-targets parts joint)
        joint-rot (.getWorldRotation joint)
        x0 (.mult joint-rot Vector3f/UNIT_X)
        y0 (.mult joint-rot Vector3f/UNIT_Y)
        z0 (.mult joint-rot Vector3f/UNIT_Z)]
    (fn []
      (let [rot-a (.clone (.getWorldRotation obj-a))
            rot-b (.clone (.getWorldRotation obj-b))
            x (.mult rot-a x0)
            y (.mult rot-a y0)
            z (.mult rot-a z0)

            X (.mult rot-b x0)
            Y (.mult rot-b y0)
            Z (.mult rot-b z0)
            heading  (Math/atan2 (.dot X z) (.dot X x))
            pitch  (Math/atan2 (.dot X y) (.dot X x))

            ;; rotate x-vector back to origin
            reverse
            (doto (Quaternion.)
              (.fromAngleAxis
               (.angleBetween X x)
               (let [cross (.normalize (.cross X x))]
```

```
                (if (= 0 (.length cross)) y cross))))
            roll (absolute-angle (.mult reverse Y) y x)]
        [heading pitch roll])))))

(defn proprioception!
  "Endow the creature with the sense of proprioception. Returns a
   sequence of functions, one for each child of the \"joints\" node in
   the creature, which each report proprioceptive information about
   that joint."
  [#^Node creature]
  ;; extract the body's joints
  (let [senses (map (partial proprioception-kernel creature)
                    (joints creature))]
    (fn []
      (map #(%) senses))))
```

`proprioception!` maps `proprioception-kernel` across all the joints of the creature. It uses the same list of joints that `joints` uses. Proprioception is the easiest sense to implement in `CORTEX`, and it will play a crucial role when efficiently implementing empathy.

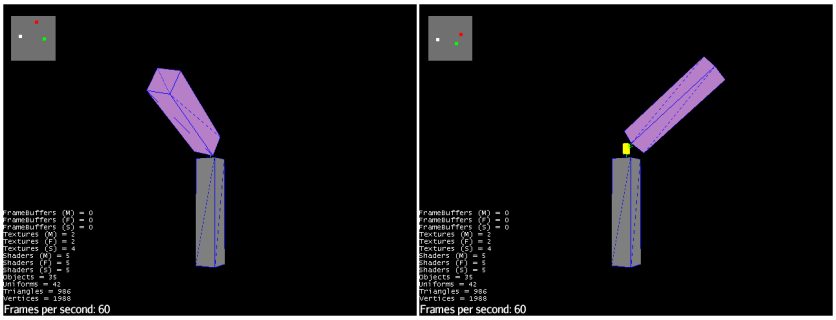

Figure 22: In the upper right corner, the three proprioceptive angle measurements are displayed. Red is yaw, Green is pitch, and White is roll.

2.12 Muscles contain both sensors and effectors

Surprisingly enough, terrestrial creatures only move by using torque applied about their joints. There's not a single straight line of force in the human body at all! (A straight line of force would correspond to some sort of jet or rocket propulsion.)

In humans, muscles are composed of muscle fibers which can contract to exert force. The muscle fibers which compose a muscle are partitioned into discrete groups which

are each controlled by a single alpha motor neuron. A single alpha motor neuron might control as little as three or as many as one thousand muscle fibers. When the alpha motor neuron is engaged by the spinal cord, it activates all of the muscle fibers to which it is attached. The spinal cord generally engages the alpha motor neurons which control few muscle fibers before the motor neurons which control many muscle fibers. This recruitment strategy allows for precise movements at low strength. The collection of all motor neurons that control a muscle is called the motor pool. The brain essentially says "activate 30% of the motor pool" and the spinal cord recruits motor neurons until 30% are activated. Since the distribution of power among motor neurons is unequal and recruitment goes from weakest to strongest, the first 30% of the motor pool might be 5% of the strength of the muscle.

My simulated muscles follow a similar design: Each muscle is defined by a 1-D array of numbers (the "motor pool"). Each entry in the array represents a motor neuron which controls a number of muscle fibers equal to the value of the entry. Each muscle has a scalar strength factor which determines the total force the muscle can exert when all motor neurons are activated. The effector function for a muscle takes a number to index into the motor pool, and then "activates" all the motor neurons whose index is lower or equal to the number. Each motor-neuron will apply force in proportion to its value in the array. Lower values cause less force. The lower values can be put at the "beginning" of the 1-D array to simulate the layout of actual human muscles, which are capable of more precise movements when exerting less force. Or, the motor pool can simulate more exotic recruitment strategies which do not correspond to human muscles.

This 1D array is defined in an image file for ease of creation/visualization. Here is an example muscle profile image.

Figure 23: A muscle profile image that describes the strengths of each motor neuron in a muscle. White is weakest and dark red is strongest. This particular pattern has weaker motor neurons at the beginning, just like human muscle.

Muscle meta-data

Listing 27: Program to deal with loading muscle data from a blender file's metadata.

```
(defn muscle-profile-image
  "Get the muscle-profile image from the node's blender meta-data."
  [#^Node muscle]
  (if-let [image (meta-data muscle "muscle")]
```

```
    (load-image image)))

(defn muscle-strength
  "Return the strength of this muscle, or 1 if it is not defined."
  [#^Node muscle]
  (if-let [strength (meta-data muscle "strength")]
    strength 1))

(defn motor-pool
  "Return a vector where each entry is the strength of the \"motor
   neuron\" at that part in the muscle."
  [#^Node muscle]
  (let [profile (muscle-profile-image muscle)]
    (vec
     (let [width (.getWidth profile)]
       (for [x (range width)]
       (- 255
          (bit-and
           0x0000FF
           (.getRGB profile x 0))))))))
```

Of note here is `motor-pool` which interprets the muscle-profile image in a way that allows me to use gradients between white and red, instead of shades of gray as I've been using for all the other senses. This is purely an aesthetic touch.

Creating muscles

Listing 28: This is the core movement function in `CORTEX`, which implements muscles that report on their activation.

```
(defn movement-kernel
  "Returns a function which when called with a integer value inside a
   running simulation will cause movement in the creature according
   to the muscle's position and strength profile. Each function
   returns the amount of force applied / max force."
  [#^Node creature #^Node muscle]
  (let [target (closest-node creature muscle)
        axis
        (.mult (.getWorldRotation muscle) Vector3f/UNIT_Y)
        strength (muscle-strength muscle)

        pool (motor-pool muscle)
        pool-integral (reductions + pool)
        forces
        (vec (map  #(float (* strength (/ % (last pool-integral))))
```

```
            pool-integral))
        control (.getControl target RigidBodyControl)]
    ;;(println-repl (.getName target) axis)
    (fn [n]
      (let [pool-index (max 0 (min n (dec (count pool))))
            force (forces pool-index)]
        (.applyTorque control (.mult axis force))
        (float (/ force strength))))))

(defn movement!
  "Endow the creature with the power of movement. Returns a sequence
   of functions, each of which accept an integer value and will
   activate their corresponding muscle."
  [#^Node creature]
    (for [muscle (muscles creature)]
      (movement-kernel creature muscle)))
```

`movement-kernel` creates a function that controls the movement of the nearest physical node to the muscle node. The muscle exerts a rotational force dependent on it's orientation to the object in the blender file. The function returned by `movement-kernel` is also a sense function: it returns the percent of the total muscle strength that is currently being employed. This is analogous to muscle tension in humans and completes the sense of proprioception begun in the last section.

2.13 CORTEX brings complex creatures to life!

The ultimate test of CORTEX is to create a creature with the full gamut of senses and put it though its paces.

With all senses enabled, my right hand model looks like an intricate marionette hand with several strings for each finger:

With the hand fully rigged with senses, I can run it though a test that will test everything.

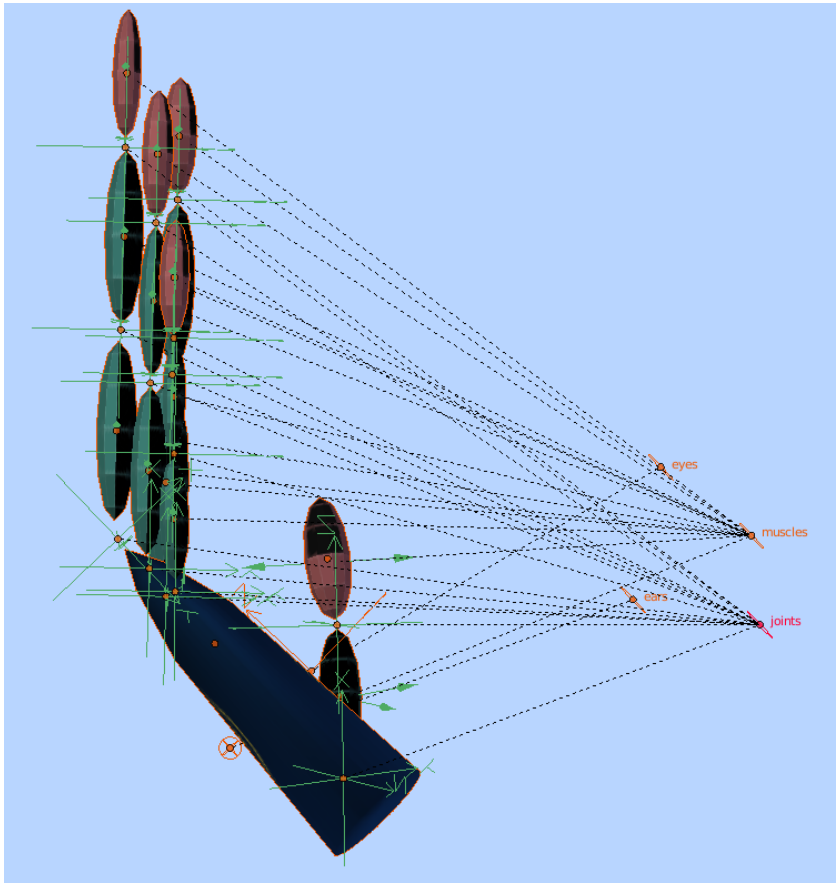

Figure 24: View of the hand model with all sense nodes. You can see the joint, muscle, ear, and eye nodes here.

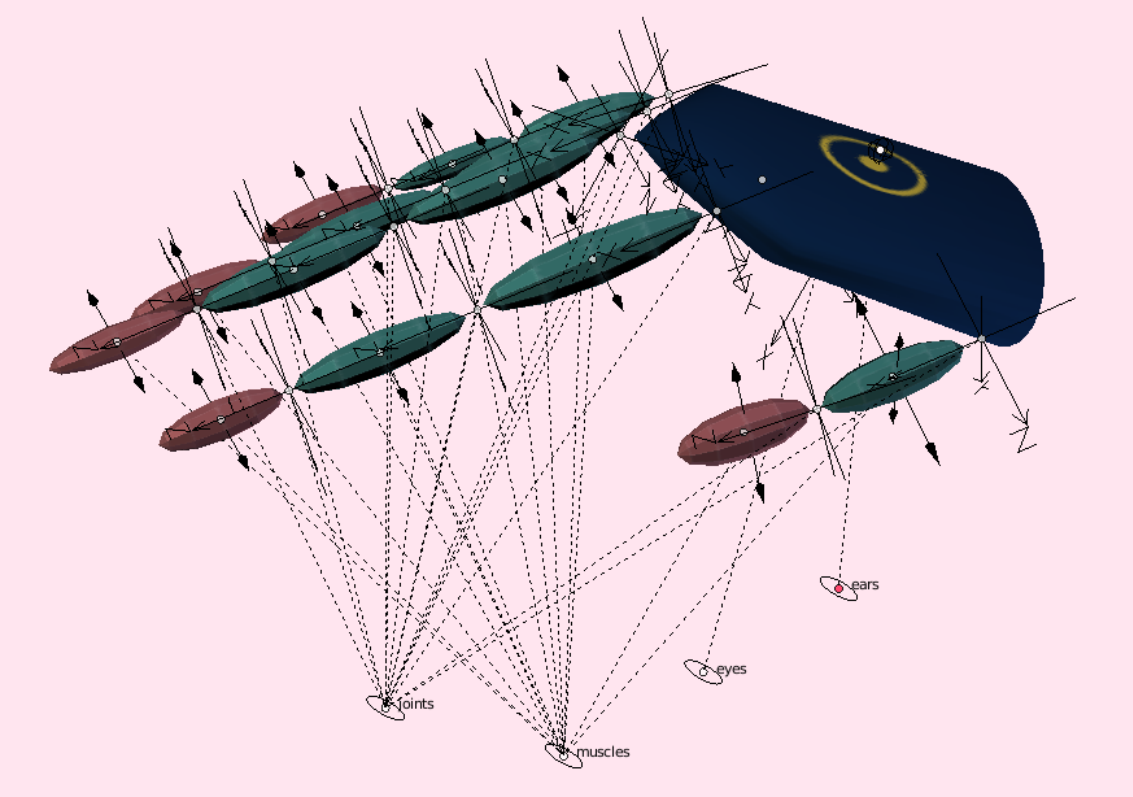

Figure 25: An alternate view of the hand.

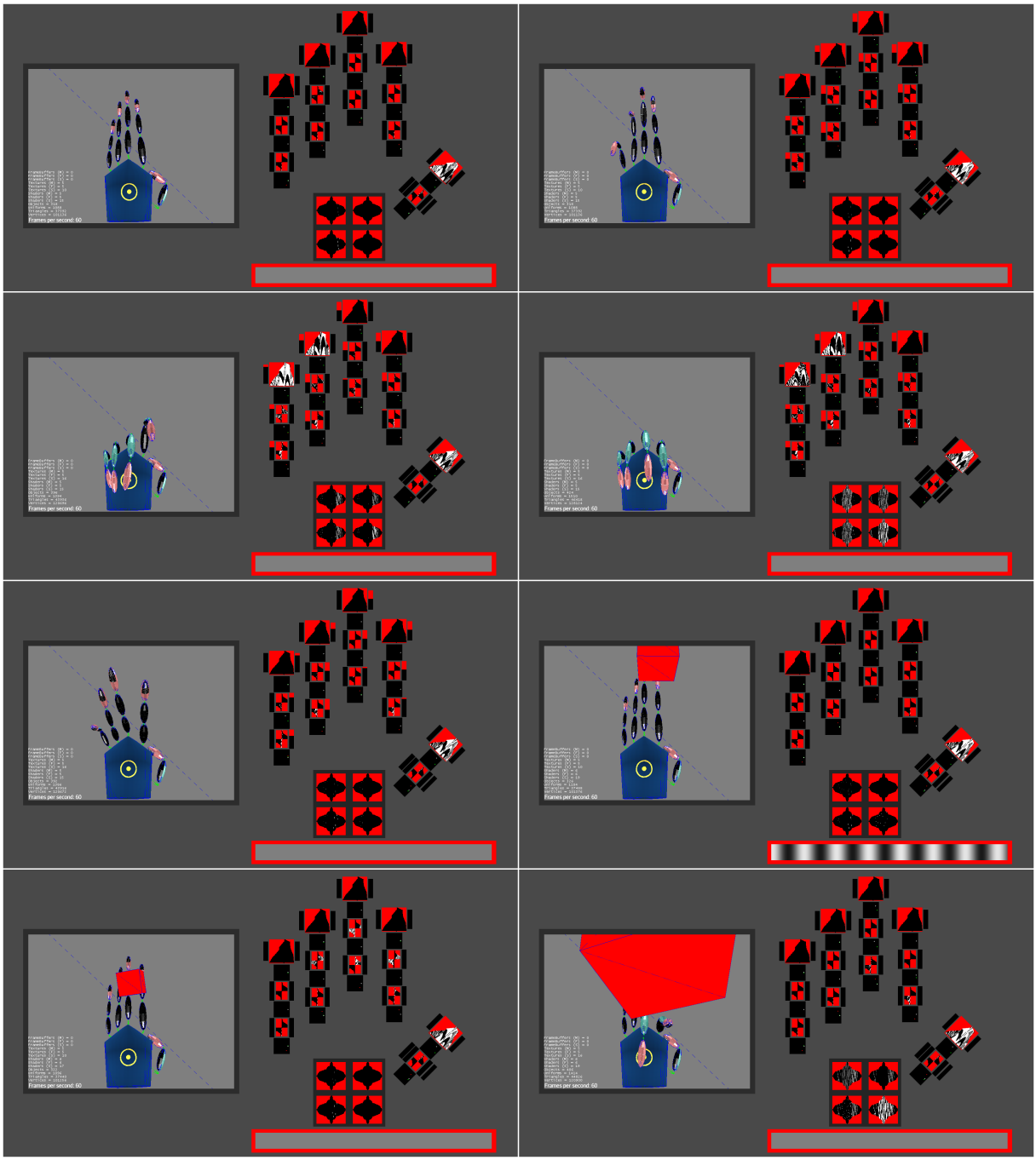

Figure 26: A full test of the hand with all senses. Note especially the interactions the hand has with itself: it feels its own palm and fingers, and when it curls its fingers, it sees them with its eye (which is located in the center of the palm. The red block appears with a pure tone sound. The hand then uses its muscles to launch the cube!

2.14 CORTEX enables many possibilities for further research

Often times, the hardest part of building a system involving creatures is dealing with physics and graphics. CORTEX removes much of this initial difficulty and leaves researchers free to directly pursue their ideas. I hope that even undergrads with a passing curiosity about simulated touch or creature evolution will be able to use cortex for experimentation. CORTEX is a completely simulated world, and far from being a disadvantage, its simulated nature enables you to create senses and creatures that would be impossible to make in the real world.

While not by any means a complete list, here are some paths CORTEX is well suited to help you explore:

Empathy my empathy program leaves many areas for improvement, among which are using vision to infer proprioception and looking up sensory experience with imagined vision, touch, and sound.

Evolution Karl Sims created a rich environment for simulating the evolution of creatures on a Connection Machine (Sims 1994). Today, this can be redone and expanded with CORTEX on an ordinary computer.

Exotic senses Cortex enables many fascinating senses that are not possible to build in the real world. For example, telekinesis is an interesting avenue to explore. You can also make a "semantic" sense which looks up metadata tags on objects in the environment the metadata tags might contain other sensory information.

Imagination via subworlds this would involve a creature with an effector which creates an entire new sub-simulation where the creature has direct control over placement/creation of objects via simulated telekinesis. The creature observes this subworld through its normal senses and uses its observations to make predictions about its top level world.

Simulated prescience step the simulation forward a few ticks, gather sensory data, then supply this data for the creature as one of its actual senses. The cost of prescience is slowing the simulation down by a factor proportional to however far you want the entities to see into the future. What happens when two evolved creatures that can each see into the future fight each other?

Swarm creatures Program a group of creatures that cooperate with each other. Because the creatures would be simulated, you could investigate computationally complex rules of behavior which still, from the group's point of view, would happen in

real time. Interactions could be as simple as cellular organisms communicating via flashing lights, or as complex as humanoids completing social tasks, etc.

HACKER for writing muscle-control programs Presented with a low-level muscle control / sense API, generate higher level programs for accomplishing various stated goals. Example goals might be "extend all your fingers" or "move your hand into the area with blue light" or "decrease the angle of this joint". It would be like Sussman's HACKER, except it would operate with much more data in a more realistic world. Start off with "calisthenics" to develop subroutines over the motor control API. The low level programming code might be a turning machine that could develop programs to iterate over a "tape" where each entry in the tape could control recruitment of the fibers in a muscle.

Sense fusion There is much work to be done on sense integration – building up a coherent picture of the world and the things in it. With CORTEX as a base, you can explore concepts like self-organizing maps or cross modal clustering in ways that have never before been tried.

Inverse kinematics experiments in sense guided motor control are easy given CORTEX's support – you can get right to the hard control problems without worrying about physics or senses.

3 EMPATH: action recognition in a simulated worm

Here I develop a computational model of empathy, using CORTEX as a base. Empathy in this context is the ability to observe another creature and infer what sorts of sensations that creature is feeling. My empathy algorithm involves multiple phases. First is free-play, where the creature moves around and gains sensory experience. From this experience I construct a representation of the creature's sensory state space, which I call Φ-space. Using Φ-space, I construct an efficient function which takes the limited data that comes from observing another creature and enriches it with a full compliment of imagined sensory data. I can then use the imagined sensory data to recognize what the observed creature is doing and feeling, using straightforward embodied action predicates. This is all demonstrated with using a simple worm-like creature, and recognizing worm-actions based on limited data.

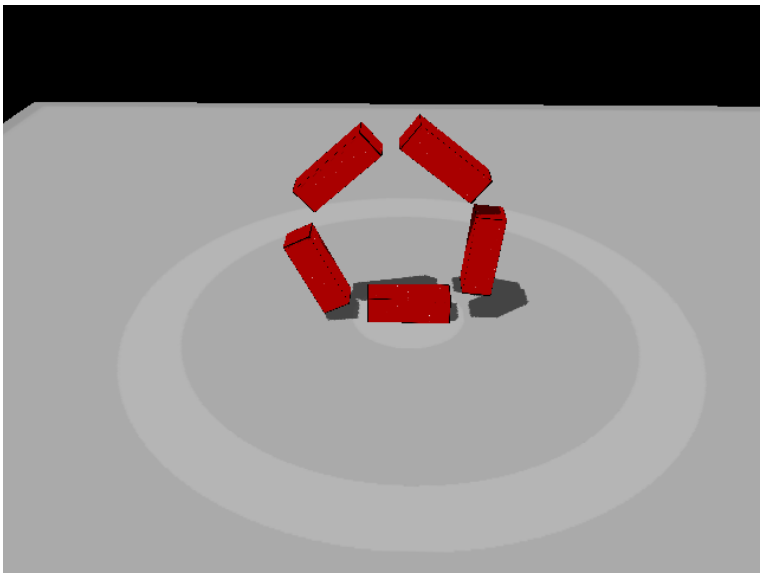

Figure 27: Here is the worm with which we will be working. It is composed of 5 segments. Each segment has a pair of extensor and flexor muscles. Each of the worm's four joints is a hinge joint which allows about 30 degrees of rotation to either side. Each segment of the worm is touch-capable and has a uniform distribution of touch sensors on each of its faces. Each joint has a proprioceptive sense to detect relative positions. The worm segments are all the same except for the first one, which has a much higher weight than the others to allow for easy manual motor control.

Listing 29: Program for reading a worm from a blender file and outfitting it with the senses of proprioception, touch, and the ability to move, as specified in the blender file.

```
(defn worm []
  (let [model (load-blender-model "Models/worm/worm.blend")]
    {:body (doto model (body!))
     :touch (touch! model)
     :proprioception (proprioception! model)
     :muscles (movement! model)}))
```

3.1 Embodiment factors action recognition into manageable parts

Using empathy, I divide the problem of action recognition into a recognition process expressed in the language of a full compliment of senses, and an imaginative process that generates full sensory data from partial sensory data. Splitting the action recognition problem in this manner greatly reduces the total amount of work to recognize actions: The imaginative process is mostly just matching previous experience, and the recognition process gets to use all the senses to directly describe any action.

3.2 Action recognition is easy with a full gamut of senses

Embodied representations using multiple senses such as touch, proprioception, and muscle tension turns out be exceedingly efficient at describing body-centered actions. It is the right language for the job. For example, it takes only around 5 lines of LISP code to describe the action of curling using embodied primitives. It takes about 10 lines to describe the seemingly complicated action of wiggling.

The following action predicates each take a stream of sensory experience, observe however much of it they desire, and decide whether the worm is doing the action they describe. `curled?` relies on proprioception, `resting?` relies on touch, `wiggling?` relies on a Fourier analysis of muscle contraction, and `grand-circle?` relies on touch and reuses `curled?` in its definition, showing how embodied predicates can be composed.

Listing 30: Program for detecting whether the worm is curled. This is the simplest action predicate, because it only uses the last frame of sensory experience, and only uses proprioceptive data. Even this simple predicate, however, is automatically frame independent and ignores vermopomorphic[1] differences such as worm textures and colors.

```
(defn curled?
  "Is the worm curled up?"
```

```
  [experiences]
  (every?
   (fn [[_ _ bend]]
     (> (Math/sin bend) 0.64))
   (:proprioception (peek experiences))))
```

Listing 31: Program for summarizing the touch information in a patch of skin.

```
(defn contact
  "Determine how much contact a particular worm segment has with
   other objects. Returns a value between 0 and 1, where 1 is full
   contact and 0 is no contact."
  [touch-region [coords contact :as touch]]
  (-> (zipmap coords contact)
      (select-keys touch-region)
      (vals)
      (#(map first %))
      (average)
      (* 10)
      (- 1)
      (Math/abs)))
```

Listing 32: Program for detecting whether the worm is at rest. This program uses a summary of the tactile information from the underbelly of the worm, and is only true if every segment is touching the floor. Note that this function contains no references to proprioception at all.

```
(def worm-segment-bottom (rect-region [8 15] [14 22]))

(defn resting?
  "Is the worm resting on the ground?"
  [experiences]
  (every?
   (fn [touch-data]
     (< 0.9 (contact worm-segment-bottom touch-data)))
   (:touch (peek experiences))))
```

Listing 33: Program for detecting whether the worm is curled up into a full circle. Here the embodied approach begins to shine, as I am able to both use a previous action predicate (`curled?`) as well as the direct tactile experience of the head and tail.

[1]Like *anthropomorphic* except for worms instead of humans.

```
(def worm-segment-bottom-tip (rect-region [15 15] [22 22]))

(def worm-segment-top-tip (rect-region [0 15] [7 22]))

(defn grand-circle?
  "Does the worm form a majestic circle (one end touching the other)?"
  [experiences]
  (and (curled? experiences)
       (let [worm-touch (:touch (peek experiences))
             tail-touch (worm-touch 0)
             head-touch (worm-touch 4)]
         (and (< 0.55 (contact worm-segment-bottom-tip tail-touch))
              (< 0.55 (contact worm-segment-top-tip    head-touch))))))
```

Listing 34: Program for detecting whether the worm has been wiggling for the last few frames. It uses a Fourier analysis of the muscle contractions of the worm's tail to determine wiggling. This is significant because there is no particular frame that clearly indicates that the worm is wiggling — only when multiple frames are analyzed together is the wiggling revealed. Defining wiggling this way also gives the worm an opportunity to learn and recognize "frustrated wiggling", where the worm tries to wiggle but can't. Frustrated wiggling is very visually different from actual wiggling, but this definition gives it to us for free.

```
(defn fft [nums]
  (map
   #(.getReal %)
   (.transform
    (FastFourierTransformer. DftNormalization/STANDARD)
    (double-array nums) TransformType/FORWARD)))

(def indexed (partial map-indexed vector))

(defn max-indexed [s]
  (first (sort-by (comp - second) (indexed s))))

(defn wiggling?
  "Is the worm wiggling?"
  [experiences]
  (let [analysis-interval 0x40]
    (when (> (count experiences) analysis-interval)
      (let [a-flex 3
            a-ex   2
            muscle-activity
            (map :muscle (vector:last-n experiences analysis-interval))
            base-activity
```

```
     (map #(- (% a-flex) (% a-ex)) muscle-activity)]
(= 2
   (first
    (max-indexed
     (map #(Math/abs %)
          (take 20 (fft base-activity))))))))))
```

With these action predicates, I can now recognize the actions of the worm while it is moving under my control and I have access to all the worm's senses.

Listing 35: Use the action predicates defined earlier to report on what the worm is doing while in simulation.

```
(defn debug-experience
  [experiences text]
  (cond
   (grand-circle? experiences) (.setText text "Grand Circle")
   (curled? experiences)       (.setText text "Curled")
   (wiggling? experiences)     (.setText text "Wiggling")
   (resting? experiences)      (.setText text "Resting")))
```

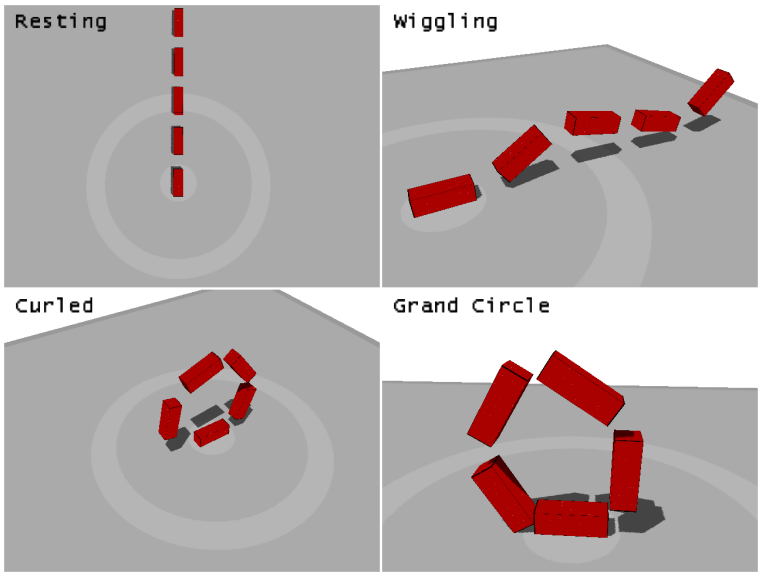

Figure 28: Using `debug-experience`, the body-centered predicates work together to classify the behavior of the worm. the predicates are operating with access to the worm's full sensory data.

These action predicates satisfy the recognition requirement of an empathic recognition system. There is power in the simplicity of the action predicates. They describe their actions without getting confused in visual details of the worm. Each one is independent of position and rotation, but more than that, they are each independent of irrelevant visual details of the worm and the environment. They will work regardless of whether the worm is a different color or heavily textured, or if the environment has strange lighting.

Consider how the human act of jumping might be described with body-centered action predicates: You might specify that jumping is mainly the feeling of your knees bending, your thigh muscles contracting, and your inner ear experiencing a certain sort of back and forth acceleration. This representation is a very concrete description of jumping, couched in terms of muscles and senses, but it also has the ability to describe almost all kinds of jumping, a generality that you might think could only be achieved by a very abstract description. The body centered jumping predicate does not have terms that consider the color of a person's skin or whether they are male or female, instead it gets right to the meat of what jumping actually *is*.

Of course, the action predicates are not directly applicable to video data, which lacks the advanced sensory information which they require!

The trick now is to make the action predicates work even when the sensory data on which they depend is absent. If I can do that, then I will have gained much.

3.3 Φ-space describes the worm's experiences

As a first step towards building empathy, I need to gather all of the worm's experiences during free play. I use a simple vector to store all the experiences.

Each element of the experience vector exists in the vast space of all possible worm-experiences. Most of this vast space is actually unreachable due to physical constraints of the worm's body. For example, the worm's segments are connected by hinge joints that put a practical limit on the worm's range of motions without limiting its degrees of freedom. Some groupings of senses are impossible; the worm can not be bent into a circle so that its ends are touching and at the same time not also experience the sensation of touching itself.

As the worm moves around during free play and its experience vector grows larger, the vector begins to define a subspace which is all the sensations the worm can practically experience during normal operation. I call this subspace Φ-space, short for physical-space. The experience vector defines a path through Φ-space. This path has interesting properties that all derive from physical embodiment. The proprioceptive components of the path vary smoothly, because in order for the worm to move from one position to another, it must pass through the intermediate positions. The path invariably forms

loops as common actions are repeated. Finally and most importantly, proprioception alone actually gives very strong inference about the other senses. For example, when the worm is proprioceptively flat over several frames, you can infer that it is touching the ground and that its muscles are not active, because if the muscles were active, the worm would be moving and would not remain perfectly flat. In order to stay flat, the worm has to be touching the ground, or it would again be moving out of the flat position due to gravity. If the worm is positioned in such a way that it interacts with itself, then it is very likely to be feeling the same tactile feelings as the last time it was in that position, because it has the same body as then. As you observe multiple frames of proprioceptive data, you can become increasingly confident about the exact activations of the worm's muscles, because it generally takes a unique combination of muscle contractions to transform the worm's body along a specific path through Φ-space.

The worm's total life experience is a long looping path through Φ-space. I will now introduce simple way of taking that experience path and building a function that can infer complete sensory experience given only a stream of proprioceptive data. This *empathy* function will provide a bridge to use the body centered action predicates on video-like streams of information.

3.4 Empathy is the process of building paths in Φ-space

Here is the core of a basic empathy algorithm, starting with an experience vector:

An *experience-index* is an index into the grand experience vector that defines the worm's life. It is a time-stamp for each set of sensations the worm has experienced.

First, group the experience-indices into bins according to the similarity of their proprioceptive data. I organize my bins into a 3 level hierarchy. The smallest bins have an approximate size of 0.001 radians in all proprioceptive dimensions. Each higher level is 10x bigger than the level below it.

The bins serve as a hashing function for proprioceptive data. Given a single piece of proprioceptive experience, the bins allow us to rapidly find all other similar experience-indices of past experience that had a very similar proprioceptive configuration. When looking up a proprioceptive experience, if the smallest bin does not match any previous experience, then successively larger bins are used until a match is found or we reach the largest bin.

Given a sequence of proprioceptive input, I use the bins to generate a set of similar experiences for each input using the tiered proprioceptive bins.

Finally, to infer sensory data, I select the longest consecutive chain of experiences that threads through the sets of similar experiences, starting with the current moment as a root and going backwards. Consecutive experience means that the experiences appear

next to each other in the experience vector.

A stream of proprioceptive input might be:

```
[ flat, flat, flat, flat, flat, flat, lift-head ]
```

The worm's previous experience of lying on the ground and lifting its head generates possible interpretations for each frame (the numbers are experience-indices):

```
[ flat, flat, flat, flat, flat, flat, flat, lift-head ]
   1     1     1     1     1     1     1     4
   2     2     2     2     2     2     2
   3     3     3     3     3     3     3
   7     7     7     7     7     7     7
   8     8     8     8     8     8     8
   9     9     9     9     9     9     9
```

These interpretations suggest a new path through phi space:

```
[ flat, flat, flat, flat, flat, flat, flat, lift-head ]
   6     7     8     9     1     2     3     4
```

The new path through Φ-space is synthesized from two actual paths that the creature has experienced: the "1-2-3-4" chain and the "6-7-8-9" chain. The "1-2-3-4" chain is necessary because it ends with the worm lifting its head. It originated from a short training session where the worm rested on the floor for a brief while and then raised its head. The "6-7-8-9" chain is part of a longer chain of inactivity where the worm simply rested on the floor without moving. It is preferred over a "1-2-3" chain (which also describes inactivity) because it is longer. The main ideas again:

- Imagined Φ-space paths are synthesized by looping and mixing previous experiences.
- Longer experience paths (less edits) are preferred.
- The present is more important than the past — more recent events take precedence in interpretation.

This algorithm has three advantages:

1. It's simple

2. It's very fast – retrieving possible interpretations takes constant time. Tracing through chains of interpretations takes time proportional to the average number of experiences in a proprioceptive bin. Redundant experiences in Φ-space can be merged to save computation.

3. It protects from wrong interpretations of transient ambiguous proprioceptive data. For example, if the worm is flat for just an instant, this flatness will not be interpreted as implying that the worm has its muscles relaxed, since the flatness is part of a longer chain which includes a distinct pattern of muscle activation. Markov chains or other memoryless statistical models that operate on individual frames may very well make this mistake.

Listing 36: Program to convert an experience vector into a proprioceptively binned lookup function.

```
(defn bin [digits]
  (fn [angles]
    (->> angles
         (flatten)
         (map (juxt #(Math/sin %) #(Math/cos %)))
         (flatten)
         (mapv #(Math/round (* % (Math/pow 10 (dec digits))))))))

(defn gen-phi-scan
  "Nearest-neighbors with binning. Only returns a result if
   the proprioceptive data is within 10% of a previously recorded
   result in all dimensions."
  [phi-space]
  (let [bin-keys (map bin [3 2 1])
        bin-maps
        (map (fn [bin-key]
               (group-by
                (comp bin-key :proprioception phi-space)
                (range (count phi-space)))) bin-keys)
        lookups (map (fn [bin-key bin-map]
                       (fn [proprio] (bin-map (bin-key proprio))))
                     bin-keys bin-maps)]
    (fn lookup [proprio-data]
      (set (some #(% proprio-data) lookups)))))
```

`longest-thread` infers sensory data by stitching together pieces from previous experience. It prefers longer chains of previous experience to shorter ones. For example, during training the worm might rest on the ground for one second before it performs its exercises. If during recognition the worm rests on the ground for five seconds,

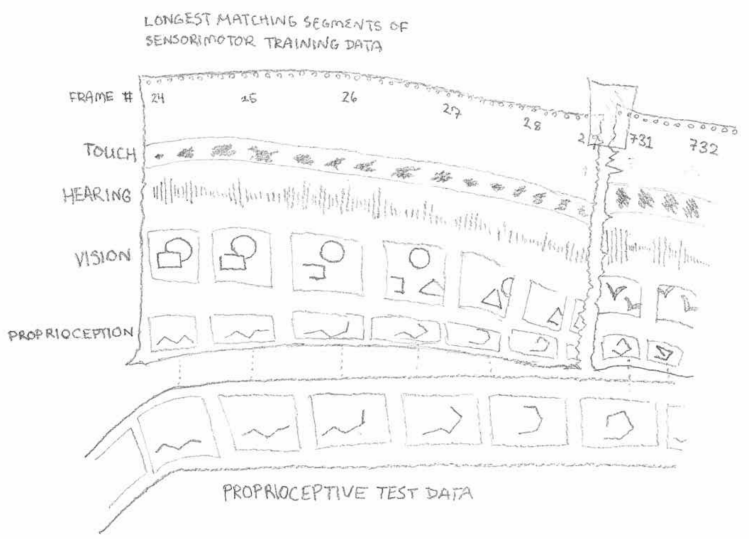

Figure 29: `longest-thread` finds the longest path of consecutive past experiences to explain proprioceptive worm data from previous data. Here, the film strip represents the creature's previous experience. Sort sequences of memories are spliced together to match the proprioceptive data. Their carry the other senses along with them.

`longest-thread` will accommodate this five second rest period by looping the one second rest chain five times.

`longest-thread` takes time proportional to the average number of entries in a proprioceptive bin, because for each element in the starting bin it performs a series of set lookups in the preceding bins. If the total history is limited, then this takes time proportional to a only a constant multiple of the number of entries in the starting bin. This analysis also applies, even if the action requires multiple longest chains – it's still the average number of entries in a proprioceptive bin times the desired chain length. Because `longest-thread` is so efficient and simple, I can interpret worm-actions in real time.

Listing 37: Program to calculate empathy by tracing though Φ-space and finding the longest (ie. most coherent) interpretation of the data.

```
(defn longest-thread
  "Find the longest thread from phi-index-sets. The index sets should
   be ordered from most recent to least recent."
  [phi-index-sets]
  (loop [result '()
         [thread-bases & remaining :as phi-index-sets] phi-index-sets]
```

```
    (if (empty? phi-index-sets)
      (vec result)
      (let [threads
            (for [thread-base thread-bases]
              (loop [thread (list thread-base)
                     remaining remaining]
                (let [next-index (dec (first thread))]
                  (cond (empty? remaining) thread
                        (contains? (first remaining) next-index)
                        (recur
                         (cons next-index thread) (rest remaining))
                        :else thread))))
            longest-thread
            (reduce (fn [thread-a thread-b]
                      (if (> (count thread-a) (count thread-b))
                        thread-a thread-b))
                    '(nil)
                    threads)]
        (recur (concat longest-thread result)
               (drop (count longest-thread) phi-index-sets))))))
```

There is one final piece, which is to replace missing sensory data with a best-guess estimate. While I could fill in missing data by using a gradient over the closest known sensory data points, averages can be misleading. It is certainly possible to create an impossible sensory state by averaging two possible sensory states. For example, consider moving your hand in an arc over your head. If for some reason you only have the initial and final positions of this movement in your Φ-space, averaging them together will produce the proprioceptive sensation of having your hand *inside* your head, which is physically impossible to ever experience (barring motor adaption illusions). Therefore I simply replicate the most recent sensory experience to fill in the gaps.

Listing 38: Fill in blanks in sensory experience by replicating the most recent experience.

```
(defn infer-nils
  "Replace nils with the next available non-nil element in the
   sequence, or barring that, 0."
  [s]
  (loop [i (dec (count s))
         v (transient s)]
    (if (zero? i) (persistent! v)
        (if-let [cur (v i)]
          (if (get v (dec i) 0)
            (recur (dec i) v)
```

```
          (recur (dec i) (assoc! v (dec i) cur)))
        (recur i (assoc! v i 0))))))
```

3.5 EMPATH recognizes actions efficiently

To use EMPATH with the worm, I first need to gather a set of experiences from the worm that includes the actions I want to recognize. The `generate-phi-space` program (listing 39 runs the worm through a series of exercises and gathers those experiences into a vector. The `do-all-the-things` program is a routine expressed in a simple muscle contraction script language for automated worm control. It causes the worm to rest, curl, and wiggle over about 700 frames (approx. 11 seconds).

Listing 39: Program to gather the worm's experiences into a vector for further processing. The `motor-control-program` line uses a motor control script that causes the worm to execute a series of "exercises" that include all the action predicates.

```
(def do-all-the-things
  (concat
   curl-script
   [[300 :d-ex 40]
    [320 :d-ex 0]]
   (shift-script 280 (take 16 wiggle-script))))

(defn generate-phi-space []
  (let [experiences (atom [])]
    (run-world
     (apply-map
      worm-world
      (merge
       (worm-world-defaults)
       {:end-frame 700
        :motor-control
        (motor-control-program worm-muscle-labels do-all-the-things)
        :experiences experiences})))
    @experiences))
```

Listing 40: Use `longest-thread` and a Φ-space generated from a short exercise routine to interpret actions during free play.

```
(defn init []
  (def phi-space (generate-phi-space))
  (def phi-scan (gen-phi-scan phi-space)))
```

```
(defn empathy-demonstration []
  (let [proprio (atom ())]
    (fn
      [experiences text]
      (let [phi-indices (phi-scan (:proprioception (peek experiences)))]
        (swap! proprio (partial cons phi-indices))
        (let [exp-thread (longest-thread (take 300 @proprio))
              empathy (mapv phi-space (infer-nils exp-thread))]
          (println-repl (vector:last-n exp-thread 22))
          (cond
           (grand-circle? empathy) (.setText text "Grand Circle")
           (curled? empathy)       (.setText text "Curled")
           (wiggling? empathy)     (.setText text "Wiggling")
           (resting? empathy)      (.setText text "Resting")
           :else                   (.setText text "Unknown")))))))

(defn empathy-experiment [record]
  (.start (worm-world :experience-watch (debug-experience-phi)
                      :record record :worm worm*)))
```

These programs create a test for the empathy system. First, the worm's Φ-space is generated from a simple motor script. Then the worm is re-created in an environment almost exactly identical to the testing environment for the action-predicates, with one major difference : the only sensory information available to the system is proprioception. From just the proprioception data and Φ-space, `longest-thread` synthesizes a complete record the last 300 sensory experiences of the worm. These synthesized experiences are fed directly into the action predicates `grand-circle?`, `curled?`, `wiggling?`, and `resting?` from before and their output is printed to the screen at each frame.

The result of running `empathy-experiment` is that the system is generally able to interpret worm actions using the action-predicates on simulated sensory data just as well as with actual data. Figure 30 was generated using `empathy-experiment`:

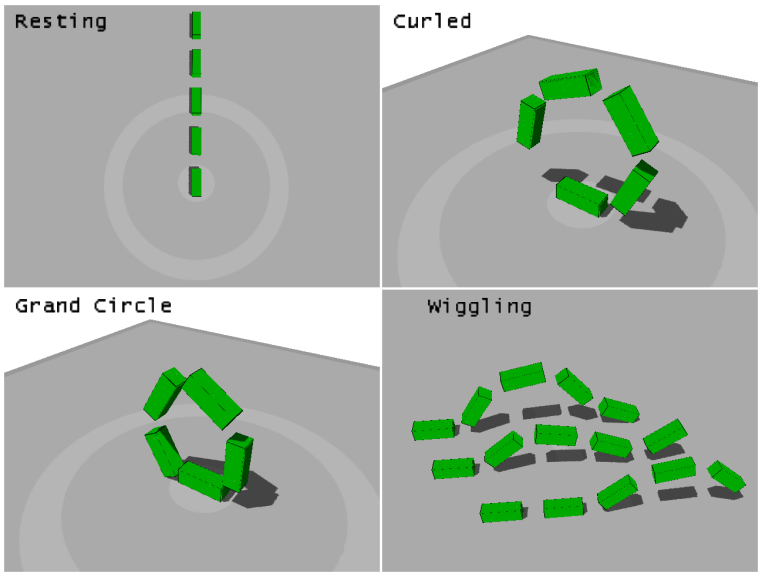

Figure 30: From only proprioceptive data, EMPATH was able to infer the complete sensory experience and classify four poses (The last panel shows a composite image of *wiggling*, a dynamic pose.)

One way to measure the performance of EMPATH is to compare the suitability of the imagined sense experience to trigger the same action predicates as the real sensory experience.

Listing 41: Determine how closely empathy approximates actual sensory data.

```
(def worm-action-label
  (juxt grand-circle? curled? wiggling?))

(defn compare-empathy-with-baseline [matches]
  (let [proprio (atom ())]
    (fn
      [experiences text]
      (let [phi-indices (phi-scan (:proprioception (peek experiences)))]
        (swap! proprio (partial cons phi-indices))
        (let [exp-thread (longest-thread (take 300 @proprio))
              empathy (mapv phi-space (infer-nils exp-thread))
              experience-matches-empathy
              (= (worm-action-label experiences)
                 (worm-action-label empathy))]
          (println-repl experience-matches-empathy)
          (swap! matches #(conj % experience-matches-empathy)))))))
```

```
(defn accuracy [v]
  (float (/ (count (filter true? v)) (count v))))

(defn test-empathy-accuracy []
  (let [res (atom [])]
    (run-world
     (worm-world :experience-watch
                 (compare-empathy-with-baseline res)
                 :worm worm*))
    (accuracy @res)))
```

Running `test-empathy-accuracy` using the very short exercise program defined in listing 39, and then doing a similar pattern of activity manually yields an accuracy of around 73%. This is based on very limited worm experience. By training the worm for longer, the accuracy dramatically improves.

Listing 42: Program to generate Φ-space using manual training.

```
(defn init-interactive []
  (def phi-space
    (let [experiences (atom [])]
      (run-world
       (apply-map
        worm-world
        (merge
         (worm-world-defaults)
         {:experiences experiences})))
      @experiences))
  (def phi-scan (gen-phi-scan phi-space)))
```

After about 1 minute of manual training, I was able to achieve 95% accuracy on manual testing of the worm using `init-interactive` and `test-empathy-accuracy`. The majority of errors are near the boundaries of transitioning from one type of action to another. During these transitions the exact label for the action is more open to interpretation, and disagreement between empathy and experience is essentially irrelevant at this point, giving a practical identification accuracy of even higher than 95%. When I watch this system myself, I generally see no errors in action identification.

3.6 Digression: Learning touch sensor layout through free play

In the previous section I showed how to compute actions in terms of body-centered predicates, but some of those predicates relied on the average touch activation of pre-defined regions of the worm's skin. What if, instead of receiving touch pre-grouped into the six

faces of each worm segment, the true topology of the worm's skin was unknown? This is more similar to how a nerve fiber bundle might be arranged inside an animal. While two fibers that are close in a nerve bundle *might* correspond to two touch sensors that are close together on the skin, the process of taking a complicated surface and forcing it into essentially a circle requires that some regions of skin that are close together in the animal end up far apart in the nerve bundle.

In this section I show how to automatically learn the skin-topology of a worm segment by free exploration. As the worm rolls around on the floor, large sections of its surface get activated. If the worm has stopped moving, then whatever region of skin that is touching the floor is probably an important region, and should be recorded.

Listing 43: Program to detect whether the worm is in a resting state with one face touching the floor.

```
(def full-contact [(float 0.0) (float 0.1)])

(defn pure-touch?
  "This is worm specific code to determine if a large region of touch
   sensors is either all on or all off."
  [[coords touch :as touch-data]]
  (= (set (map first touch)) (set full-contact)))
```

After collecting these important regions, there will many nearly similar touch regions. While for some purposes the subtle differences between these regions will be important, for my purposes I collapse them into mostly non-overlapping sets using `remove-similar` in listing 44

Listing 44: Program to take a list of sets of points and "collapse them" so that the remaining sets in the list are significantly different from each other. Prefer smaller sets to larger ones.

```
(defn remove-similar
  [coll]
  (loop [result () coll (sort-by (comp - count) coll)]
    (if (empty? coll) result
        (let  [[x & xs] coll
               c (count x)]
          (if (some
               (fn [other-set]
                 (let [oc (count other-set)]
                   (< (- (count (union other-set x)) c) (* oc 0.1))))
               xs)
            (recur result xs)
            (recur (cons x result) xs))))))
```

Actually running this simulation is easy given CORTEX's facilities.

Listing 45: Collect experiences while the worm moves around. Filter the touch sensations by stable ones, collapse similar ones together, and report the regions learned.

```
(defn learn-touch-regions []
  (let [experiences (atom [])
        world (apply-map
               worm-world
               (assoc (worm-segment-defaults)
                 :experiences experiences))]
    (run-world world)
    (->>
     @experiences
     (drop 175)
     ;; access the single segment's touch data
     (map (comp first :touch))
     ;; only deal with "pure" touch data to determine surfaces
     (filter pure-touch?)
     ;; associate coordinates with touch values
     (map (partial apply zipmap))
     ;; select those regions where contact is being made
     (map (partial group-by second))
     (map #(get % full-contact))
     (map (partial map first))
     ;; remove redundant/subset regions
     (map set)
     remove-similar)))

(defn learn-and-view-touch-regions []
  (map view-touch-region
       (learn-touch-regions)))
```

The only thing remaining to define is the particular motion the worm must take. I accomplish this with a simple motor control program.

Listing 46: Motor control program for making the worm roll on the ground. This could also be replaced with random motion.

```
(defn touch-kinesthetics []
  [[170 :lift-1 40]
   [190 :lift-1 19]
   [206 :lift-1  0]

   [400 :lift-2 40]
```

```
[410 :lift-2  0]

[570 :lift-2 40]
[590 :lift-2 21]
[606 :lift-2  0]

[800 :lift-1 30]
[809 :lift-1 0]

[900 :roll-2 40]
[905 :roll-2 20]
[910 :roll-2  0]

[1000 :roll-2 40]
[1005 :roll-2 20]
[1010 :roll-2  0]

[1100 :roll-2 40]
[1105 :roll-2 20]
[1110 :roll-2  0]
])
```

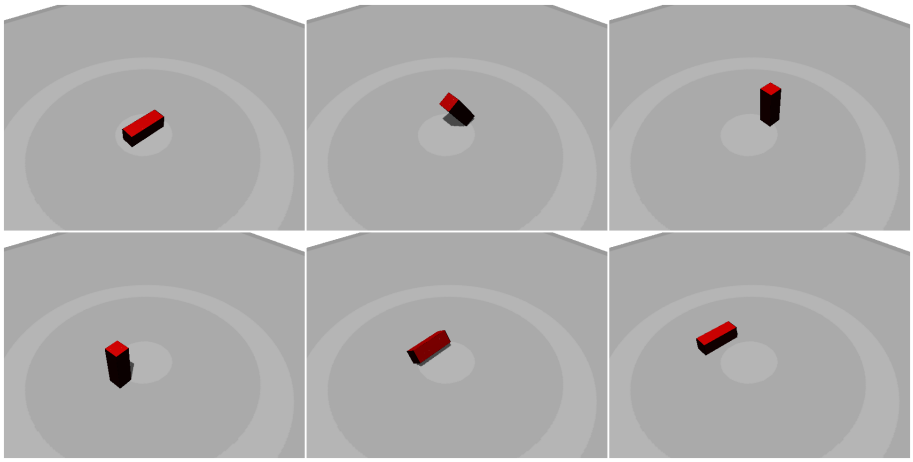

Figure 31: The small worm rolls around on the floor, driven by the motor control program in listing 31.

While simple, `learn-touch-regions` exploits regularities in both the worm's physiology and the worm's environment to correctly deduce that the worm has six sides. Note that `learn-touch-regions` would work just as well even if the worm's touch sense data

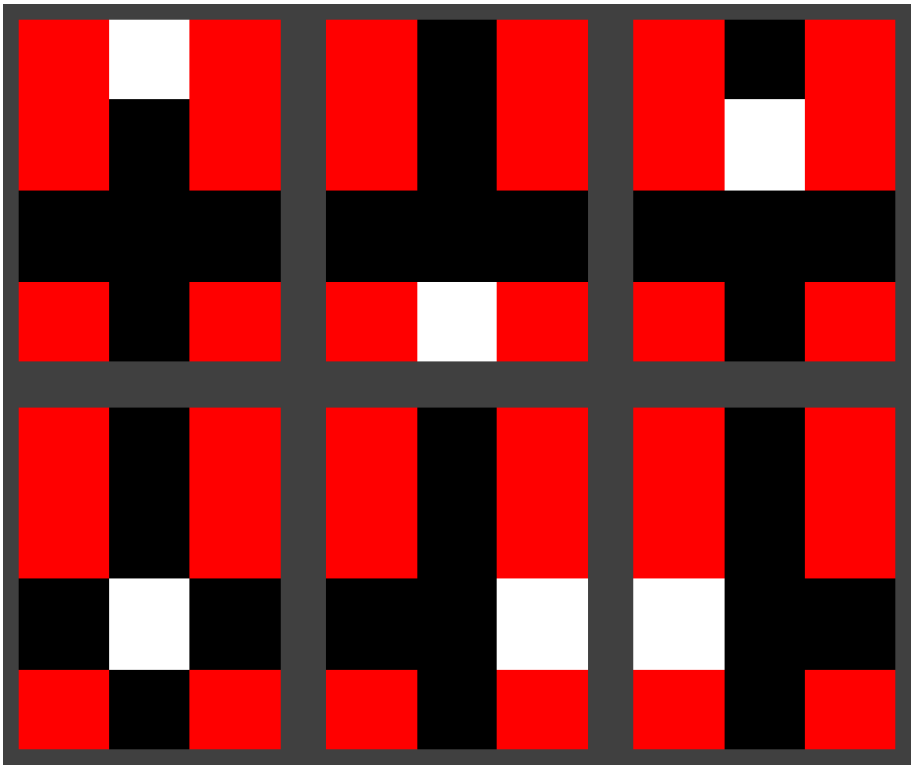

Figure 32: After completing its adventures, the worm now knows how its touch sensors are arranged along its skin. Each of these six rectangles are touch sensory patterns that were deemed important by `learn-touch-regions`. Each white square in the rectangles above is a cluster of "related" touch nodes as determined by the system. The worm has correctly discovered that it has six faces, and has partitioned its sensory map into these six faces.

were completely scrambled. The cross shape is just for convenience. This example justifies the use of pre-defined touch regions in EMPATH.

3.7 Recognizing an object using embodied representation

At the beginning of the thesis, I suggested that we might recognize the chair in Figure 2 by imagining ourselves in the position of the man and realizing that he must be sitting on something in order to maintain that position. Here, I present a brief elaboration on how to this might be done.

First, I need the feeling of leaning or resting *on* some other object that is not the floor. This feeling is easy to describe using an embodied representation.

Listing 47: Program describing the sense of leaning or resting on something. This involves a relaxed posture, the feeling of touching something, and a period of stability where the worm does not move.

```
(defn draped?
  "Is the worm:
    -- not flat (the floor is not a 'chair')
    -- supported (not using its muscles to hold its position)
    -- stable (not changing its position)
    -- touching something (must register contact)"
  [experiences]
  (let [b2-hash (bin 2)
        touch (:touch (peek experiences))
        total-contact
        (reduce
         +
         (map #(contact all-touch-coordinates %)
              (rest touch)))]
    (println total-contact)
    (and (not (resting? experiences))
         (every?
          zero?
          (-> experiences
              (vector:last-n 25)
              (#(map :muscle %))
              (flatten)))
         (-> experiences
             (vector:last-n 20)
             (#(map (comp b2-hash flatten :proprioception) %))
             (set)
             (count) (= 1))
         (< 0.03 total-contact))))
```

Though this is a simple example, using the `draped?` predicate to detect the cube has interesting advantages. The `draped?` predicate describes the cube not in terms of properties that the cube has, but instead in terms of how the worm interacts with it

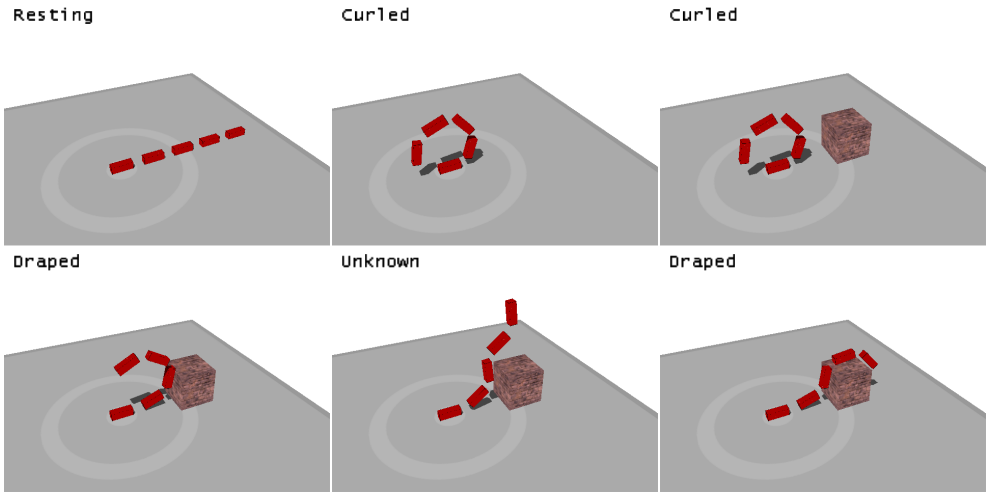

Figure 33: The `draped?` predicate detects the presence of the cube whenever the worm interacts with it. The details of the cube are irrelevant; only the way it influences the worm's body matters.

physically. This means that the cube can still be detected even if it is not visible, as long as its influence on the worm's body is visible.

This system will also see the virtual cube created by a "mimeworm", which uses its muscles in a very controlled way to mimic the appearance of leaning on a cube. The system will anticipate that there is an actual invisible cube that provides support!

This makes me wonder about the psychology of actual mimes. Suppose for a moment that people have something analogous to Φ-space and that one of the ways that they find objects in a scene is by their relation to other people's bodies. Suppose that a person watches a person miming an invisible wall. For a person with no experience with miming, their Φ-space will only have entries that describe the scene with the sensation of their hands touching a wall. This sensation of touch will create a strong impression of a wall, even though the wall would have to be invisible. A person with experience in miming however, will have entries in their Φ-space that describe the wall-miming position without a sense of touch. It will not seem to such as person that an invisible wall is present, but merely that the mime is holding out their hands in a special way. Thus, the theory that humans use something like Φ-space weakly predicts that learning how to mime should break the power of miming illusions. Most optical illusions still work no matter how much you know about them, so this proposal would be quite interesting to test, as it predicts a non-standard result!

Figure 34: Can you see the thing that this person is leaning on? What properties does it have, other than how it makes the man's elbow and shoulder feel? I wonder if people who can actually maintain this pose easily still see the support?

4 Contributions

The big idea behind this thesis is a new way to represent and recognize physical actions, which I call *empathic representation*. Actions are represented as predicates which have access to the totality of a creature's sensory abilities. To recognize the physical actions of another creature similar to yourself, you imagine what they would feel by examining the position of their body and relating it to your own previous experience.

Empathic representation of physical actions is robust and general. Because the representation is body-centered, it avoids baking in a particular viewpoint like you might get from learning from example videos. Because empathic representation relies on all of a creature's senses, it can describe exactly what an action *feels like* without getting caught up in irrelevant details such as visual appearance. I think it is important that a correct description of jumping (for example) should not include irrelevant details such as the color of a person's clothes or skin; empathic representation can get right to the heart of what jumping is by describing it in terms of touch, muscle contractions, and a brief feeling of weightlessness. Empathic representation is very low-level in that it describes actions using concrete sensory data with little abstraction, but it has the generality of much more abstract representations!

Another important contribution of this thesis is the development of the `CORTEX` system, a complete environment for creating simulated creatures. You have seen how to implement five senses: touch, proprioception, hearing, vision, and muscle tension. You have seen how to create new creatures using blender, a 3D modeling tool.

As a minor digression, you also saw how I used `CORTEX` to enable a tiny worm to discover the topology of its skin simply by rolling on the ground. You also saw how to detect objects using only embodied predicates.

In conclusion, for this thesis I:

- Developed the idea of embodied representation, which describes actions that a creature can do in terms of first-person sensory data.
- Developed a method of empathic action recognition which uses previous embodied experience and embodied representation of actions to greatly constrain the possible interpretations of an action.
- Created `EMPATH`, a program which uses empathic action recognition to recognize physical actions in a simple model involving segmented worm-like creatures.
- Created `CORTEX`, a comprehensive platform for embodied AI experiments. It is the base on which `EMPATH` is built.

1 Appendix: CORTEX User Guide

Those who write a thesis should endeavor to make their code not only accessible, but actually usable, as a way to pay back the community that made the thesis possible in the first place. This thesis would not be possible without Free Software such as jMonkeyEngine3, Blender, clojure, emacs, ffmpeg, and many other tools. That is why I have included this user guide, in the hope that someone else might find CORTEX useful.

1.1 Obtaining CORTEX

You can get cortex from its mercurial repository at http://hg.bortreb.com/cortex. You may also download CORTEX releases at http://aurellem.org/cortex/releases/. As a condition of making this thesis, I have also provided Professor Winston the CORTEX source, and he knows how to run the demos and get started. You may also email me at cortex@aurellem.org and I may help where I can.

1.2 Running CORTEX

CORTEX comes with README and INSTALL files that will guide you through installation and running the test suite. In particular you should look at test cortex.test which contains test suites that run through all senses and multiple creatures.

1.3 Creating creatures

Creatures are created using *Blender*, a free 3D modeling program. You will need Blender version 2.6 when using the CORTEX included in this thesis. You create a CORTEX creature in a similar manner to modeling anything in Blender, except that you also create several trees of empty nodes which define the creature's senses.

Mass

To give an object mass in CORTEX, add a "mass" metadata label to the object with the mass in jMonkeyEngine units. Note that setting the mass to 0 causes the object to be immovable.

Joints

Joints are created by creating an empty node named joints and then creating any number of empty child nodes to represent your creature's joints. The joint will automatically

connect the closest two physical objects. It will help to set the empty node's display mode to "Arrows" so that you can clearly see the direction of the axes.

Joint nodes should have the following metadata under the "joint" label:

```
;; ONE of the following, under the label "joint":
{:type :point}

;; OR

{:type :hinge
 :limit [<limit-low> <limit-high>]
 :axis (Vector3f. <x> <y> <z>)}
;;(:axis defaults to (Vector3f. 1 0 0) if not provided for hinge joints)

;; OR

{:type :cone
 :limit-xz <lim-xz>
 :limit-xy <lim-xy>
 :twist    <lim-twist>}   ;(use XZY rotation mode in blender!)
```

Eyes

Eyes are created by creating an empty node named `eyes` and then creating any number of empty child nodes to represent your creature's eyes.

Eye nodes should have the following metadata under the "eye" label:

```
{:red    <red-retina-definition>
 :blue   <blue-retina-definition>
 :green  <green-retina-definition>
 :all    <all-retina-definition>
 (<0xrrggbb> <custom-retina-image>)...
}
```

Any of the color channels may be omitted. You may also include your own color selectors, and in fact :red is equivalent to 0xFF0000 and so forth. The eye will be placed at the same position as the empty node and will bind to the neatest physical object. The eye will point outward from the X-axis of the node, and "up" will be in the direction of the X-axis of the node. It will help to set the empty node's display mode to "Arrows" so that you can clearly see the direction of the axes.

Each retina file should contain white pixels wherever you want to be sensitive to your chosen color. If you want the entire field of view, specify :all of 0xFFFFFF and a retinal map that is entirely white.

Here is a sample retinal map:

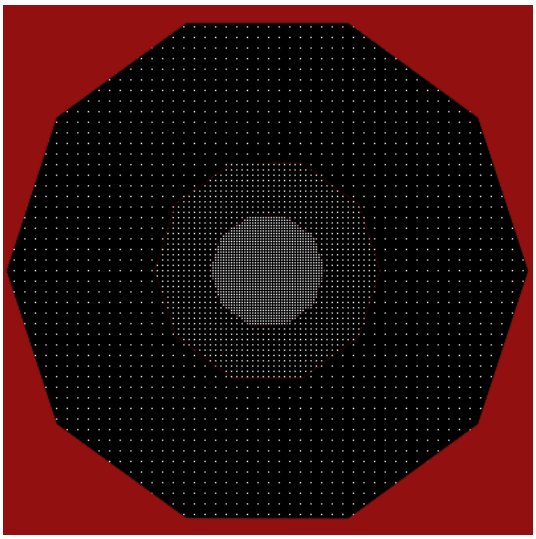

Figure 35: An example retinal profile image. White pixels are photo-sensitive elements. The distribution of white pixels is denser in the middle and falls off at the edges and is inspired by the human retina.

Hearing

Ears are created by creating an empty node named `ears` and then creating any number of empty child nodes to represent your creature's ears.

Ear nodes do not require any metadata.

The ear will bind to and follow the closest physical node.

Touch

Touch is handled similarly to mass. To make a particular object touch sensitive, add metadata of the following form under the object's "touch" metadata field:

```
<touch-UV-map-file-name>
```

You may also include an optional "scale" metadata number to specify the length of the touch feelers. The default is 0.1, and this is generally sufficient.

The touch UV should contain white pixels for each touch sensor.

Here is an example touch-uv map that approximates a human finger, and its corresponding model.

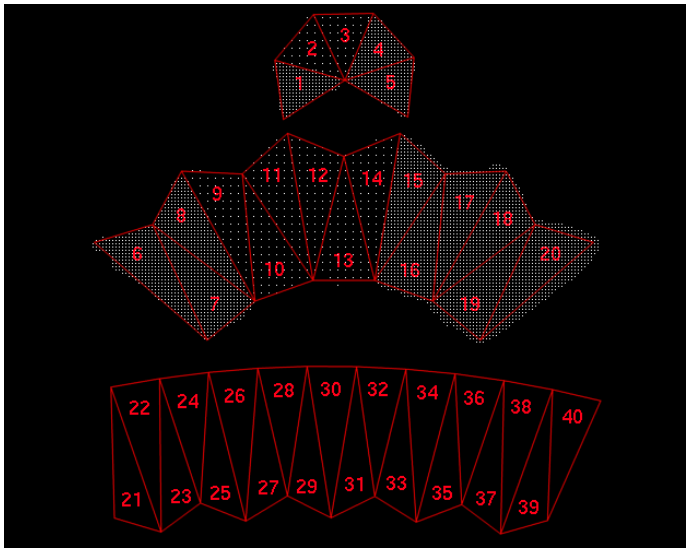

Figure 36: This is the tactile-sensor-profile for the upper segment of a fingertip. It defines regions of high touch sensitivity (where there are many white pixels) and regions of low sensitivity (where white pixels are sparse).

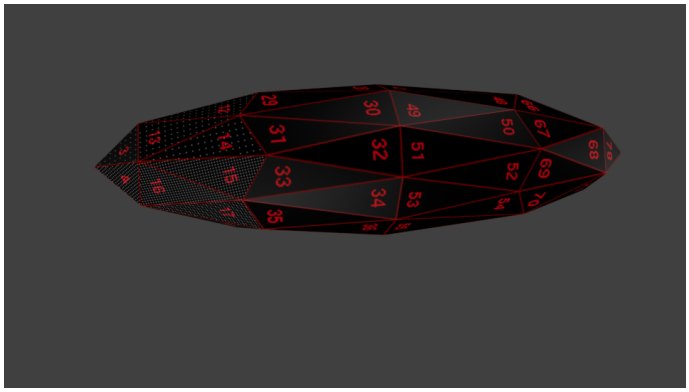

Figure 37: The fingertip UV-image form above applied to a simple model of a fingertip.

Proprioception

Proprioception is tied to each joint node – nothing special must be done in a blender model to enable proprioception other than creating joint nodes.

Muscles

Muscles are created by creating an empty node named `muscles` and then creating any number of empty child nodes to represent your creature's muscles.

Muscle nodes should have the following metadata under the "muscle" label:

```
<muscle-profile-file-name>
```

Muscles should also have a "strength" metadata entry describing the muscle's total strength at full activation.

Muscle profiles are simple images that contain the relative amount of muscle power in each simulated alpha motor neuron. The width of the image is the total size of the motor pool, and the redness of each neuron is the relative power of that motor pool.

While the profile image can have any dimensions, only the first line of pixels is used to define the muscle. Here is a sample muscle profile image that defines a human-like muscle.

Figure 38: A muscle profile image that describes the strengths of each motor neuron in a muscle. White is weakest and dark red is strongest. This particular pattern has weaker motor neurons at the beginning, just like human muscle.

Muscles twist the nearest physical object about the muscle node's Z-axis. I recommend using the "Single Arrow" display mode for muscles and using the right hand rule to determine which way the muscle will twist. To make a segment that can twist in multiple directions, create multiple, differently aligned muscles.

1.4 CORTEX API

These are the some functions exposed by `CORTEX` for creating worlds and simulating creatures. These are in addition to jMonkeyEngine3's extensive library, which is documented elsewhere.

Simulation

`(world root-node key-map setup-fn update-fn)` create a simulation.

root-node a `com.jme3.scene.Node` object which contains all of the objects that should be in the simulation.

key-map a map from strings describing keys to functions that should be executed whenever that key is pressed. the functions should take a SimpleApplication object and a boolean value. The SimpleApplication is the current simulation that is running, and the boolean is true if the key is being pressed, and false if it is being released. As an example,

```
{"key-j" (fn [game value] (if value (println "key j pressed")))}
```

is a valid key-map which will cause the simulation to print a message whenever the 'j' key on the keyboard is pressed.

setup-fn a function that takes a `SimpleApplication` object. It is called once when initializing the simulation. Use it to create things like lights, change the gravity, initialize debug nodes, etc.

update-fn this function takes a `SimpleApplication` object and a float and is called every frame of the simulation. The float tells how many seconds is has been since the last frame was rendered, according to whatever clock jme is currently using. The default is to use IsoTimer which will result in this value always being the same.

`(position-camera world position rotation)` set the position of the simulation's main camera.

`(enable-debug world)` turn on debug wireframes for each simulated object.

`(set-gravity world gravity)` set the gravity of a running simulation.

`(box length width height & {options})` create a box in the simulation. Options is a hash map specifying texture, mass, etc. Possible options are `:name`, `:color`, `:mass`, `:friction`, `:texture`, `:material`, `:position`, `:rotation`, `:shape`, and `:physical?`.

`(sphere radius & {options})` create a sphere in the simulation. Options are the same as in box.

`(load-blender-model file-name)` create a node structure representing the model described in a blender file.

`(light-up-everything world)` distribute a standard compliment of lights throughout the simulation. Should be adequate for most purposes.

`(node-seq node)` return a recursive list of the node's children.

(nodify name children) construct a node given a node-name and desired children.

(add-element world element) add an object to a running world simulation.

(set-accuracy world accuracy) change the accuracy of the world's physics simulator.

(asset-manager) get an *AssetManager*, a jMonkeyEngine construct that is useful for loading textures and is required for smooth interaction with jMonkeyEngine library functions.

(load-bullet) unpack native libraries and initialize the bullet physics subsystem. This function is required before other world building functions are called.

Creature Manipulation / Import

(body! creature) give the creature a physical body.

(vision! creature) give the creature a sense of vision. Returns a list of functions which will each, when called during a simulation, return the vision data for the channel of one of the eyes. The functions are ordered depending on the alphabetical order of the names of the eye nodes in the blender file. The data returned by the functions is a vector containing the eye's *topology*, a vector of coordinates, and the eye's *data*, a vector of RGB values filtered by the eye's sensitivity.

(hearing! creature) give the creature a sense of hearing. Returns a list of functions, one for each ear, that when called will return a frame's worth of hearing data for that ear. The functions are ordered depending on the alphabetical order of the names of the ear nodes in the blender file. The data returned by the functions is an array of PCM (pulse code modulated) wav data.

(touch! creature) give the creature a sense of touch. Returns a single function that must be called with the *root node* of the world, and which will return a vector of *touch-data* one entry for each touch sensitive component, each entry of which contains a *topology* that specifies the distribution of touch sensors, and the *data*, which is a vector of `[activation, length]` pairs for each touch hair.

(proprioception! creature) give the creature the sense of proprioception. Returns a list of functions, one for each joint, that when called during a running simulation will report the `[heading, pitch, roll]` of the joint.

(movement! creature) give the creature the power of movement. Creates a list of functions, one for each muscle, that when called with an integer, will set the recruitment of that muscle to that integer, and will report the current power being exerted by the muscle. Order of muscles is determined by the alphabetical sort order of the names of the muscle nodes.

Visualization/Debug

(view-vision) create a function that when called with a list of visual data returned from the functions made by `vision!`, will display that visual data on the screen.

(view-hearing) same as `view-vision` but for hearing.

(view-touch) same as `view-vision` but for touch.

(view-proprioception) same as `view-vision` but for proprioception.

(view-movement) same as `view-vision` but for muscles.

(view anything) `view` is a polymorphic function that allows you to inspect almost anything you could reasonably expect to be able to "see" in `CORTEX`.

(text anything) `text` is a polymorphic function that allows you to convert practically anything into a text string.

(println-repl anything) print messages to clojure's repl instead of the simulation's terminal window.

(mega-import-jme3) for experimenting at the REPL. This function will import all jMonkeyEngine3 classes for immediate use.

(display-dilated-time world timer) Shows the time as it is flowing in the simulation on a HUD display.

TODO – add a paper about detecting biological motion from only a few dots.

Bibliography

Bear et al. (2006). *Neuroscience: Exploring the Brain*. 3rd Edition. Lippincott Williams & Wilkins. ISBN: 9780781760034.

This is the introductory textbook to 9.01. It provides a good introduction to all major human senses.

Blender (2013). http://www.blender.org/.

All complicated creatures in CORTEX are described using Blender's extensive 3D modeling capabilities. Blender is a very sophisticated 3D modeling environment and has been used to create a short movie called Sintel http://www.sintel.org/.

Brooks, Rodney A. (1991). "Intelligence Without Representation". In: *Artificial Intelligence* 47.1-3. Available at : http://people.csail.mit.edu/brooks/papers/representation.pdf, pp. 139–159.

Presents an argument that simulation will not be enough to develop artificial intelligence, and that we must rely on the real world and robots if we are to build truly robust systems. While CORTEX embraces simulation because of Time, this paper remains a compelling argument for why the entire enterprise might not even be a good idea.

Coen, Michael Harlan (2006). "Multimodal dynamics : self-supervised learning in perceptual and motor systems". Available at: http://hdl.handle.net/1721.1/34022. PhD thesis. MIT.

This thesis shows how to use multiple senses to mutually bootstrap off of each other and achieve clustering results that no sense could be able to achieve alone. Cross-modal clustering becomes more powerful the more senses it has, and is ideal to implement in an environment such as CORTEX's.

Harvey, Christopher D. et al. "Intracellular dynamics of hippocampal place cells during

virtual navigation". In: *Nature* 461. Available at : http://papers.cnl.salk.edu/PDFs/IntracelllularDynamicsofVirtualPlaceCells2011-4178.pdf, pp. 941–946.

Researchers at Princeton created a special Quake II level that simulated a maze, and added an interface where a mouse could run on top of a ball in various directions to move the character in the simulated maze. They measured hippocampal activity during this exercise to try and tease out the method in which spatial data was stored in that area of the brain. I find this promising because it shows that simulated worlds are still clear enough for a simple rat to navigate — they don't just have meaning from our own highly advanced imaginations. I want to see if a rat can reasonably grow up if it lives its entire live hooked up to the game!

jMonkeyEngine3 (2013). http://hub.jmonkeyengine.org/.

This is the video game engine on which CORTEX is based.

Ke, Yan, R. Sukthankar, and M. Hebert (2005). "Efficient visual event detection using volumetric features". In: *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference*. Vol. 1. http://www.intel-research.net/Publications/Pittsburgh/092620050705_320.pdf, 166–173 Vol. 1.

This is an example of using frame-dependent methods to detect actions in video. I consider this to be the wrong language for describing actions, because it has no way to completely describe even a simple action like "curling" form all points of view.

Larson, Stephen David (2003). "Intrinsic representation : bootstrapping symbols from experience". Available at: http://hdl.handle.net/1721.1/28462. MA thesis. MIT.

This is an example of a thesis that I think could be improved with CORTEX. Larson uses a simple blocks world simulator to explore using self-organizing maps to bootstrap symbols just from exploration with a simulated arm and colored blocks.

Sacks, Oliver (1998). *The Man Who Mistook His Wife For A Hat: And Other Clinical Tales*. Simon and Schuster. ISBN: 9780330700580.

This book describes exotic cases where the human mind goes wrong. The section on proprioception is particularly relevant to this thesis, and one of the best explanations of how important proprioception

is, though the eyes of someone who has lost the sense.

Sims, Karl (1994). "Evolving Virtual Creatures". In: *Computer Graphics (Siggraph '94 Proceedings)*. Available as: http://www.karlsims.com/papers/siggraph94.pdf, pp. 15–22.

Karl Sims uses a simulated virtual environment similar to CORTEX to study the evolution of a set of creatures as they develop to perform various tasks such as swimming or competing for a ball. His code only ran on the Connection Machine (CM-5), which sadly doesn't exist anymore. CORTEX presents an opportunity to continue this line of research.

Sussman, Gerald J. (1973). "A Computational Model of Skill Acquisition". Available at: http://hdl.handle.net/1721.1/6894. PhD thesis. MIT.

Sussman creates a program called HACKER, which operates in a blocks world environment and learns to debug programs to build things with blocks and control its own body. This sort of approach to problem solving is begging to be implemented in CORTEX's rich world. Will program debugging still work well with many more senses and a more complicated environment?

Turing, Alan M. (1950). "Computing machinery and intelligence". In: *Mind*. Available as: http://www.csee.umbc.edu/courses/471/papers/turing.pdf, pp. 433–460.

The original paper that inspired the Turing test. It's important because in it Turing states that we don't have to care about the "hand" part of "mind and hand", using the example of Helen Keller as motivation. I think that this is a mistake, and that embodiment is critical to intelligence.

Winston, Patrick Henry (2011). "The Strong Story Hypothesis and the Directed Perception Hypothesis". In: *Technical Report FS-11-01, Papers from the AAAI Fall Symposium*. Ed. by Pat Langley. Available as: http://hdl.handle.net/1721.1/67693. Menlo Park, CA: AAAI Press, pp. 345–352.

Discusses an idea called the *directed perception hypothesis*, which argues that much of our intelligence resides in our senses themselves, and our ability to direct their resources on imagined problems. This has had the greatest influence on CORTEX.

— (2012). "The Next 50 Years: a Personal View". In: *Biologically Inspired Cognitive Architectures* 1. Available as : http://groups.csail.mit.edu/genesis/papers/

2012bica-phw, pp. 92–99.

Great summary of historical attempts at AI, and more thoughts on how directed perception and mimicry as in EMPATH might play an important role in intelligence.