An External Tabletop Environment for an Interactive Brain Model

Derek Monner, James Reggia
{dmonner, reggia}@cs.umd.edu

Dept. of Computer Science
A. V. Williams Bldg.
University of Maryland
College Park MD 20742

September 2007

CS-TR-4883 and UMIACS-TR-2007-41

As an important step towards creating a biologically-inspired model of language learning in the human brain, we have created an external environment in which to embody such a model. It is our hope that the learning in such a model will be enhanced and accelerated by the grounding effects that such an environment can provide. The environment can be easily configured to approximate common tabletop testing environments used in clinical studies of human aphasia patients, which will allow us to draw parallels between said studies and similar computational experiments to be conducted by artificially lesioning our language-learning brain model. This document describes the capabilities and programming interfaces of the external environment and how a computational agent might interact with it.

Acknowledgement: This work was supported by NIH Award NS35460.
J. Reggia is also with UMIACS and the Dept. of Neurology, UMB.

1 Introduction

As part of an ongoing effort to create a model of language learning in the brain, we have created an external environment for use with such a model. Our brain model will be embodied as an agent in this environment via a manipulator that is meant to represent an arm and hand. We desire our language-learning agent to be embodied in this environment because we want its use of language to be grounded by related visual stimuli. We also desire a testing environment similar to that used in the aphasia literature, so we can compare our model's results to those of human test subjects. For this reason, our environment resembles a tabletop with several different types of objects on it, each of which can be moved and manipulated by the agent.

2 Overview

The external environment is a 3-dimensional space, with real-valued axes in each direction. In the center of the space is a table, on which various objects, including blocks, boxes, pyramids, and more complex shapes, reside. Although the environment is represented internally as collections of state variables, we provide visualization tools, both for the benefit of the experimenters and as a potential means to interface with an agent having visual capacity. Figure 1 shows the default visualization of a moderately complex environment.

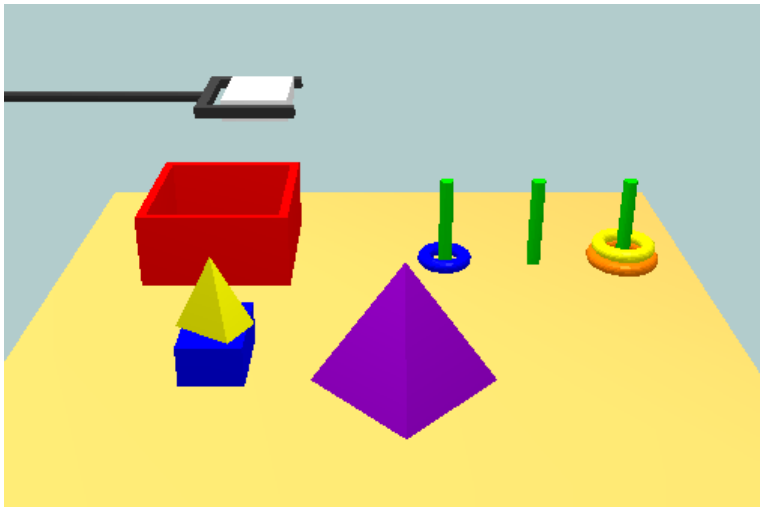

Figure 1: An environment shown with the default visualization. The tan-colored floor is the tabletop, with several inanimate objects on top of it. The only manipulator visible is a claw in the upper left corner of the image, where it is holding a white block above an open-topped red box.

Objects in the environment are divided into inanimate objects and manipulators, the latter of which are directly controlled by an outside observer or agent—such as an experimenter or a brain model—to change the state of the environment. Inanimate objects may be grasped and moved by the manipulators, but are otherwise part of the static background of the environment. Figure 2 shows a sequence of images of a manipulator interacting with the environment.

The environment may be simulated in real time or in discrete time steps. The simulator attempts to roughly approximate the physical processes that would be at work in a real tabletop environment.

Many experiments from the psychological, neurological, and specifically brain aphasia literature

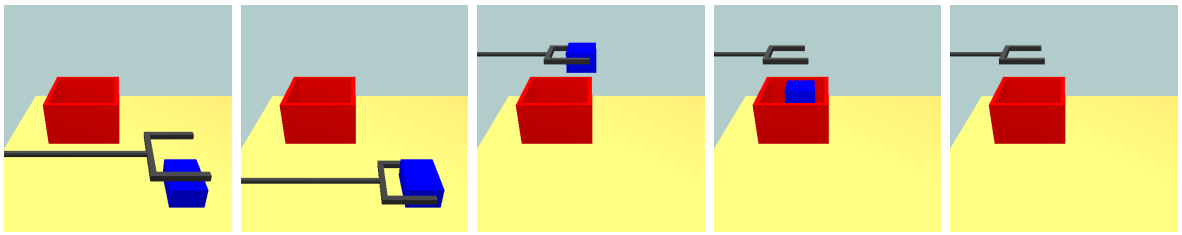

Figure 2: A manipulator picks up a blue block and drops in into a red box. The box is deep enough that the blue block disappears from sight.

find the subject and the experimenter at a table with a collection of simple objects, where the subject is asked to manipulate the objects or describe their configuration. This environment facilitates the duplication of this type of experiment in a simulated environment where a brain model plays the role of the subject, with the goal being to measure differences between the behavior of the model and the behavior of human subjects.

The external environment is implemented in Java and visualized via the Java3D library.

In the sections that follow the reader will find detailed descriptions of the environment and its representation (Section 3), the simulation process (Section 4), the provided visualization facilities (Section 5), thoughts on brain model integration with this environment (Section 6), and a discussion of planned improvements to this environment (Section 7).

3 Environment Representation

The tabletop environment is encapsulated by the `Environment` class. This class keeps state variables that define the dimensions and locations of the table and other objects, and the time that has passed during simulation. The `Environment` class also contains information about manipulators that can act on the environment. Each instance of the `Environment` class is created based on a Scene File provided at runtime by the user. Finally, the `Environment` class is responsible for simulating the motions of the objects in it as time passes. All of these responsibilities will be covered in detail in the subsections below.

3.1 Table

The table is represented as a flat surface whose x- and z-dimensions are specified at the time the `Environment` object is created. The origin of the environment—the point with coordinates $(0, 0, 0)$—is located on top of the table and at its center.

The table is, by default, the only inanimate object in the simulation that is unaffected by gravity (because the table represents the ground), and the only object that cannot be grasped by manipulators. The table is home to all the inanimate objects in the environment, which are described below.

3.2 Inanimate Objects

An inanimate object is any non-manipulator object in the environment. In other words, an inanimate object is one that cannot be affected directly by an agent. Generally, inanimate objects only move in one of two ways: being grasped and moved by a manipulator, or falling due to gravitational acceleration.

Several types of inanimate objects can appear in the scene, and each of these is a subclass of the abstract `EnvironmentShape` class. The `EnvironmentShape` derivatives, such as `Block`, `Pyramid`, and `Box`, each take several parameters to determine the size of the shape to create.

All the `EnvironmentShape` subclasses contain a string which identifies the object internally, and a color used to render the object. Each specific shape requires the user to provide further details about its size and properties. Each `EnvironmentShape` subclass is described in detail below.

After one of these shapes is instantiated, it is generally wrapped in an `EnvironmentObject`, which keeps track of position, orientation, velocity, and acceleration information about the shape relative to the rest of the environment. An `EnvironmentObject` is not limited to representing one `EnvironmentShape` in space, however: Each `EnvironmentObject` can hold a union of shapes (along with their relative positions) that will move and interact with the environment as a single unit. For example, if one wished to create a "house" shape, one could create an `EnvironmentObject` containing a squat `Block` underneath a small `Pyramid`. The resulting `EnvironmentObject` would be manipulated as a unit in the environment, instead of manipulating each piece individually.

3.2.1 Block

The `Block` class describes a solid cuboid, or rectangular box. Figure 3 depicts an example of a block. At the time of creation, the x-, y-, and z-dimensions must be specified.

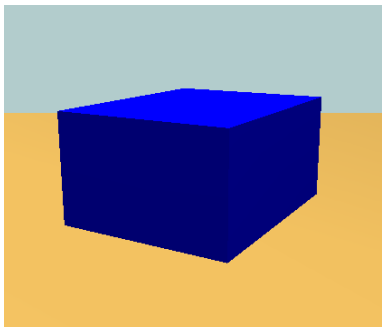

Figure 3: A blue `Block`

3.2.2 Pyramid

The `Pyramid` class describes a pyramid with a rectangular base where the apex is directly above the center of the base. Figure 4 depicts an example of a pyramid. At the time of creation, the x- and z-dimensions define the size of the base, and the y-dimension gives the perpendicular distance from the center of the base to the apex. By default the pyramid points, from base to apex, along the positive y-direction.

3.2.3 Box

The `Box` class describes a cuboid or rectangular box with an open top and hollow center. Figure 5 depicts an example of a box. When creating a box, the user can specify an outer dimension and an inner dimension, in which case the resultant box will have a square base and square sides of the outer dimension, and an open hole of the inner dimension on top (the positive y-facing side). For finer-grained control, the user could instead specify inner and outer dimensions for each of the

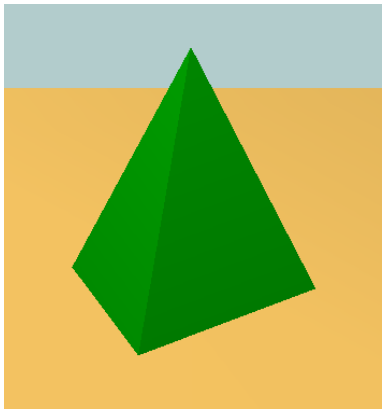

Figure 4: A green `Pyramid`

x, y, and z directions. In both cases, the thickness of the edges of the box is equal to the outer dimension minus the inner dimension.

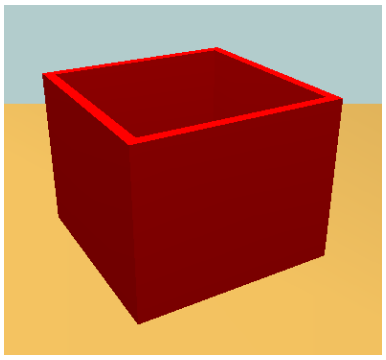

Figure 5: A red `Box`

3.2.4 Peg

The `Peg` class describes an approximation to a cylinder. Figure 6 depicts an example of a peg. The user generally provides a height and a radius, and the cylinder is constructed with the bases parallel to the xz-plane and extending in the y-direction. If an elliptical base is desired, separate x- and z-radii can be specified.

By default, the bases of circular cylinder are regular decagons (10 sides of equal length and angle), but the number of sides is customizable at the time of creation, allowing the creation of bases shaped like other regular polygons.

3.2.5 Disc

The `Disc` class is similar to a `Peg` with a cylindrical hole through the center. The class is named `Disc` because its most common usage is for creating objects that resemble compact discs (CDs). Figure 7 depicts an example of a disc. It is specified by an outer radius, inner radius, and height.

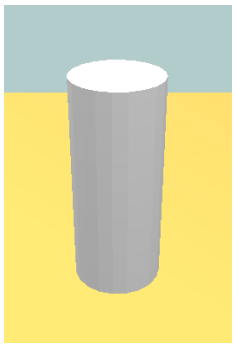

Figure 6: A white `Peg`

The outer radius and height work as in `Peg`, except that everything from the inner radius towards the center is not filled in.

Figure 7: An orange `Disc`

By default, the discs are regular decagons (10 sides of equal length and angle), but, as with pegs, the number of sides is customizable at the time of creation.

Creating non-circular discs is not currently supported.

3.2.6 Ring

The `Ring` class describes an approximation to a torus. Figure 8 depicts an example of a ring. The user must provide an inner radius and outer radius. A circle (with diameter equal to the outer radius minus the inner radius) is then rotated about the y-axis around a circle of the inner radius. This leaves the empty inner circle of the inner radius, and creates a torus of the outer radius.

Note that the y-extent of the torus is equal to the diameter of the circle that was rotated to create the torus (i.e. two times the outer radius minus the inner radius).

Creating non-circular rings is not currently supported. Also, rings with non-circular cross-sections are not currently supported.

3.3 Manipulators

An agent interacts with the environment through manipulators. Manipulators have all the same properties as inanimate objects in the environment, and in fact the abstract `EnvironmentManipulator` class is a direct subclass of `EnvironmentObject`. Unlike inanimate objects, however, manipulators

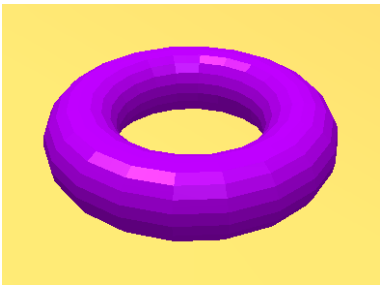

Figure 8: A purple `Ring`

are controlled directly by an outside agent. By default, manipulators are not affected by gravitational acceleration, as we assume that an agent can hold its manipulator aloft.

Manipulators may be acted upon in several ways. An agent can induce acceleration by application of a force, or set velocity directly. The agent may also translate or "teleport" a manipulator to another position instantaneously. And of course, manipulators can be rotated in three dimensions.

Manipulators have two other operations an agent can perform: grab and release. The exact definition of when and how a manipulator performs these operations is specific to the manipulator type (described below), but the basic idea is as follows. A grab operation associates some object in the environment with this manipulator, and causes said object to move in tandem with the manipulator (as would be expected once an object has been "grabbed"). Conversely, a release operation dissociates the grabbed object, returning it to the static background of the environment and freeing the manipulator to grab other objects.

3.3.1 Magnet Manipulator

The simple `MagnetManipulator` can be visualized as a magnet at the end of a long pole—think of the magnetic cranes used in scrap yards to move wrecked cars around. Figure 9 depicts a `MagnetManipulator`. This manipulator may grab an object that is sufficiently close to it and underneath its magnet.

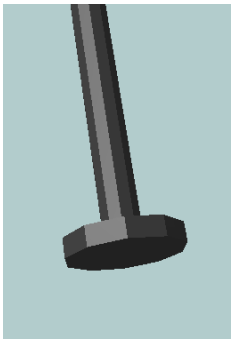

Figure 9: A `MagnetManipulator`

3.3.2 Claw Manipulator

This more advanced `ClawManipulator` is a basic implementation of the crane in the classical AI "Blocks World" problem. It consists of a two-pronged "claw" attached to a long pole. Figure 10 depicts a `ClawManipulator`. The `ClawManipulator` may grab any object that passes between the digits of its claw, at which point the claw closes to grasp the object. Upon release, the digits return to their original positions.

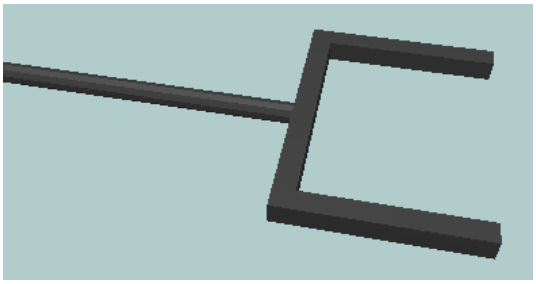

Figure 10: A `ClawManipulator`

This implementation imposes an obvious limitation: Objects larger than the distance between the claw's digits cannot be grasped. To remedy this, the size of the claw can be changed dynamically by calling the `setEnclosingSize()` method during simulation.

3.4 Scene Files

Scene Files can be passed as an argument when creating an `Environment` object, and they describe the environment's initial state. A Scene File lists each object present in the environment and its properties, including name, color, dimensions, initial position, and orientation.

Improvements are planned to the Scene File language that allow it to describe the dimensions of the environment and the initial state of manipulators in that environment. This will allow a Scene File to be a complete description of the initial state of the environment.

4 Simulation

Time progresses in the `Environment` only when its `simulate()` method is invoked. As a parameter, this method takes an amount of time to simulate, and assumes that no forces are changed during the simulation interval, resulting in constant accelerations (except in cases of collision), but changing velocities and positions throughout the interval.

The `simulate()` method is programmed such that, starting from the same initial state, any two sequences of calls to `simulate()` will produce the same resultant state if the sum of the parameters in one sequence is equal the other sequence's sum. For example, if we start from some initial state, the following sequences of calls are all equivalent, assuming no forces are applied in between:

- `simulate(0.5)`, `simulate(0.5)`
- `simulate(0.2)`, `simulate(0.7)`, `simulate(0.1)`
- `simulate(1.0)`

The simulation performs a very rough approximation of the physical processes involved in the movement of the objects through the environment. The next few paragraphs describe a few important ways that the behavior of the simulation differs from what would be expected in a physical environment.

Each object other than the table is subject to a force of gravity until such time as it collides with an object below it. If the landed-upon object is flat (i.e. the surface normal is parallel to gravity), then the first object ceases to fall. An example of where this behavior is odd occurs if a large block falls onto a much smaller block, where the smaller block is supporting not the center of the large block, but a corner of the larger block. In such a case, the larger block would be considered supported, and gravitational effects would cease. Similarly, if an inverted pyramid hits the table apex-first, it will remain in that precariously-balanced position. However, if an object lands on a smooth gradient, such as the face of a pyramid, it will progress down the gradient under the influence of gravity as would be expected.

Air resistance is simulated by degrading velocities by a constant factor. The consequence of this is that a constantly-accelerating object travelling through the air will reach a terminal velocity, thought this velocity does not depend on the object's surface area with respect to the direction of travel. Objects do not exhibit friction with respect to each other.

Collision detection does what one would expect—disallows objects from occupying the same physical space. However, collision behavior is odd in that the collisions do not conserve energy. Any two objects involved in a collision have their velocities set to zero. For example, if an experimenter rams a manipulator at full speed into a stationary block, the block will not budge, and the velocity of the manipulator will go directly to zero.

These anomalies exist for reasons of computational tractability, and will be eliminated if they are found to detract from experiments.

5 Visualization

Once an environment has been created, it is often useful to visualize the environment on a display. This has obvious advantages for the experimenters, as it is easier for humans to assess the state of the environment visually than by reading state variables directly. Figure 11 shows a visualization of an environment depicting the game Towers of Hanoi.

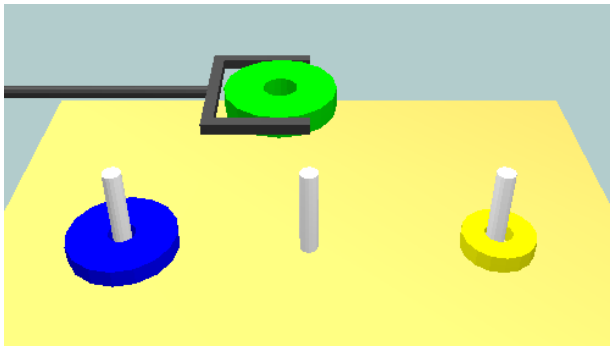

Figure 11: Experimenters can observe an agent playing the Towers of Hanoi game in this visualized environment.

Alongside the environment model, we provide simple visualization facilities, rendered via the Java3D libraries. While at present these are of primary importance to the experimenter, planned

extensions will provide useful images to vision-equipped agents designed to operate in the environment.

One starts a visualization by instantiating a `Visualization` object, passing the `Environment` as a parameter. In the default visualization, the positive x-axis points to the right and parallel to the table, the positive y-axis points up and is normal to the table, and the positive z-axis points towards the screen, parallel to the table. Figure 12 shows the axis orientations visually. The default visualization simulates the environment in real time and provides a simple keyboard interface for adjusting the view. The details of the implementation are provided below.

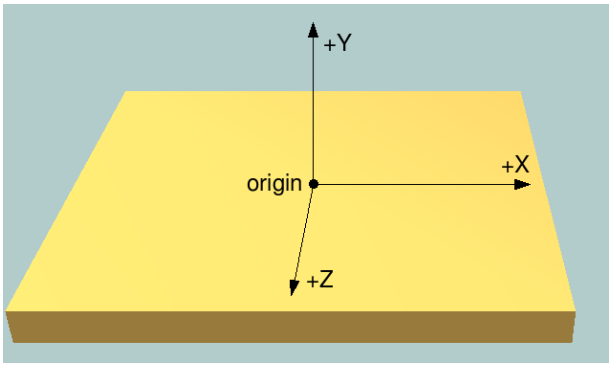

Figure 12: The orientations of the coordinate axes with respect to the table in the default visualization

5.1 Real-time Simulation

The `RealtimePhysicsBehavior` class provides a visualization with an environment that is simulated in real time. It measures the time that elapses between drawn frames (down to tens of milliseconds), and then calls the environment's `simulate()` method for that amount of time. This provides a smooth visual interface that effectively updates the environment in real time.

Unfortunately, as environment complexity goes up, so do the computational resources required per call to `simulate()`. As such, if using an environment which is more complex than the hardware can support, an experimenter may experience low frame rates, causing him or her to miss critical features of the simulation. To combat this problem, an alternative class, `StepPhysicsBehavior`, can replace `RealtimePhysicsBehavior` in situations as described above. `StepPhysicsBehavior` calls `simulate()` with a fixed time value (10 milliseconds, by default) before rendering each frame. As a result, if the frame rate slows down, so too does simulated time, allowing experimenters to see every possible detail.

5.2 Keyboard Interface

The `CameraControlBehavior` class provides a keyboard interface that allows an experimenter to control the position of the camera during a visualization. The camera operates by centering on a "point of focus" in the environment, and rotating or zooming with respect to that point. The default point of focus is $(0, 3, 0)$, and the default camera position is 18 units away from the point of focus in the xz-plane, inclined 30° from the table. Table 1 lists the key bindings for controlling the camera.

Key Combination	Effect
Left Arrow	Rotate the camera to the left about the point of focus
Right Arrow	Rotate the camera to the right about the point of focus
Up Arrow	Rotate the camera up about the point of focus
Down Arrow	Rotate the camera down about the point of focus
Page Up	Zoom in on the point of focus
Page Down	Zoom out
Shift + Left Arrow	Shift the point of focus in the negative x-direction
Shift + Right Arrow	Shift the point of focus in the positive x-direction
Shift + Up Arrow	Shift the point of focus in the negative z-direction
Shift + Down Arrow	Shift the point of focus in the positive z-direction
Shift + Page Up	Shift the point of focus in the positive y-direction
Shift + Page Down	Shift the point of focus in the negative y-direction
Home	Return to the default camera position and focus point

Table 1: Default key bindings for camera control

The keyboard can simultaneously be used to control a manipulator in the environment through use of the `ManipulatorControlBehavior` class. Table 2 lists the key bindings for manipulator control.

Key Combination	Effect
Control + Left Arrow	Move the manipulator in the negative x-direction
Control + Right Arrow	Move the manipulator in the positive x-direction
Control + Up Arrow	Move the manipulator in the negative z-direction
Control + Down Arrow	Move the manipulator in the positive z-direction
Control + Page Up	Move the manipulator in the positive y-direction
Control + Page Down	Move the manipulator in the negative y-direction
Plus	For `ClawManipulator`s, increases the size of the claw.
Minus	For `ClawManipulator`s, decreases the size of the claw.
Spacebar	If not currently grasping an object, grab the object within reach of the manipulator; otherwise, release any held object

Table 2: Default key bindings for the manipulator

6 Model Integration

Although no interactive brain model has yet been completed for use with this external environment, significant consideration has been given to the environment's use with such a brain model, which we shall henceforth refer to as an agent.

In the simplest case, an agent would be associated with a single manipulator. The agent would have full access to the state-variable description of the environment, and would act upon the environment by applying forces to the manipulator (or simply by teleporting the manipulator), grabbing and releasing objects, and so forth. The agent would also have the sole power to call the environment's `simulate()` method after its actions, enabling it to learn from the results.

A more complicated setup could involve an agent using several manipulators, or multiple agents in the same environment (in either cooperative, competitive, or mentor/mentee relationships), or an agent's manipulator cooperating with or learning from one controlled by an experimenter. Also, rather than access the environment's state-variable representation directly, an agent may be forced to rely solely upon a visualization of the environment. Variations of these setups will be explored in future research.

6.1 Controlling a Manipulator via Software

A software agent can interact with an `EnvironmentManipulator` by calling various methods that change the state of the manipulator. These methods are described in detail below.

6.1.1 Controlling Acceleration

Certainly the most physically realistic way to interact with a manipulator is to simulate applying forces directly to it. The `applyForce(Vector3f)` method takes in a 3-vector representing the direction and magnitude of a force to be applied uniformly to the manipulator, causing its acceleration to increase, along the direction the vector points and in proportion to its magnitude. The force is applied continuously, and causes changes to the manipulator's velocity during subsequent calls to the `Environment`'s `simulate()` method. Acceleration continues until the manipulator's `resetAcceleration()` method is called, which causes all acceleration to cease, or until a collision occurs.

An agent can see the manipulator's current acceleration with `getAcceleration(Vector3f)`, which copies the current acceleration into the argument vector. The `setAcceleration(Vector3f)` method discards the manipulator's current acceleration in favor of the new one passed in as an argument. It can be used instead of the `applyForce(Vector3f)` method if dealing with only one force at a time is favorable.

Generally only calls to these methods will change the acceleration of a manipulator. The only exception is when the manipulator collides with other objects in the environment, which will cause acceleration (and velocity) to change or cease.

The above methods are sufficient for manipulator movement. However, it may not always be convenient to move a manipulator in this manner, so the following alternative control mechanisms are provided.

6.1.2 Controlling Velocity Directly

A less realistic but often far easier way to control a manipulator is to give it instantaneous changes in velocity. An agent can get a copy of the current velocity with the `getVelocity(Vector3f)` method. The `addVelocity(Vector3f)` method adds the given velocity vector to the manipulator's current velocity. Similarly, `setVelocity(Vector3f)` replaces the current velocity with the argument. The `resetVelocity()` method sets an object's velocity to zero, but if acceleration remains nonzero the object will begin to move again in subsequent `simulate()` calls made on the `Environment`.

Velocity changes take effect immediately when `simulate()` is next called, causing the manipulator's position to change. `simulate()` is also charged with updating velocities based on the acceleration experienced during the amount of time simulated. Besides the method calls listed above and updates due to acceleration, the only other way a velocity will change is when the manipulator collides with another object.

6.1.3 Teleportation

The least realistic way to move the manipulator is via teleportation. The `translate(Vector3f)` method changes the manipulator's relative position according to the direction and magnitude of the input vector. The `translateTo(Vector3f)` treats the argument as a point in space, and moves the manipulator so its center is at that point.

These methods act immediately; they do not wait for the next call to `simulate()`. As a result, they also do not have access to the normal collision detection routines, so use them with caution, as a wrong move can situate the manipulator inside another object. As such, these methods are most often useful for manipulator placement before the beginning of a simulation.

6.1.4 Rotation

The following methods are used to change the orientation of the manipulator in space. The `rotate(AxisAngle4f)` and `rotate(Matrix4f)` rotate the manipulator from its current orientation based on the argument. An `AxisAngle4f` is a 4-tuple, the first 3 items of which specify the axis of rotation, with the fourth item giving the rotation angle in radians. A `Matrix4f` is a standard rotation matrix.

Similarly, `rotateTo(AxisAngle4f)` and `rotateTo(Matrix4f)` discard the current orientation of the object, returning it to its default orientation before applying the rotation specified in the argument.

These rotation methods act immediately, independent of calls to `simulate()`, and so, like the teleportation methods, do not take advantage of collision detection routines. This makes them most suitable for placement of objects before starting a simulation. This behavior may be changed in a future release to allow for collision-aware, non-instantaneous rotation of objects.

6.1.5 Grab and Release

A software agent can control which objects the manipulator interacts with via the `grab()` and `release()` methods. `grab()` instructs the manipulator to attempt to grasp anything within range (which varies by manipulator); it will return `true` if another object was able to be grasped and `false` otherwise. `release()` does the opposite, causing the manipulator to drop any held object; it returns `true` if there was a held object that was released, and `false` otherwise.

Both the `grab()` and `release()` methods act immediately; they do not wait for the next call to `simulate()`. However, unlike other instantaneous methods, they do take collisions into account.

For the benefit of forgetful agents, the `isGrabbing()` method will return `true` if and only if an object is being grasped.

The `ClawManipulator` has two additional important methods. `getEnclosingSize()` returns the amount of space between the digits of the claw. `setEnclosingSize(float)` resizes the claw in order to accommodate space equal to the argument between the digits (or returns `false` if the resize is impossible because of the proximity of other objects). These methods are useful mainly for allowing the claw to grasp objects larger than its default size.

6.2 Running the Simulator

In order to apply many of the changes an agent can make to the state of a manipulator, it must call the `Environment`'s `simulate()` function or its variant, `simulate(float)`. The former simulates one hundredth of a second, and the latter simulates the number of seconds provided by the argument.

In general, it is computationally most efficient to simulate the largest amount of time possible in which no manual changes will be made to the manipulator via method calls.

An agent wanting to simulate the environment has several options, the simplest of which is manually calling `simulate(float)`, with an appropriately-sized time interval, each time the agent acts on the manipulator. Such a strategy involves the least amount of computation but provides the lowest accuracy visualization for the agent or the experimenter. If more impressive graphics are desired, the agent can employ the previously described `StepPhysicsBehavior` or `RealtimePhysicsBehavior` classes, which call `simulate()` automatically in step with some visualization.

7 Future Additions

Planned improvements to the external environment model include:

- Introducing a wider selection of primitive shapes that are available to add to the environment.
- Introducing tools—objects that have special effects or that cause environmental changes when used in a certain way.
- Introducing the notion of weight to objects, and weight limits for what manipulators can successfully pick up.
- Expanding the Scene File implementation to specify the dimensions of the environment, as well as locations and orientations of manipulators.
- Providing a visualization facility for rendering a single frame of the environment via a method call. This would be useful for when a single image needs to be used as input to a neural or symbolic portion of a brain model.
- Extending the physics modeled during simulation as necessary to provide desired effects in the environment.