The Maryland Virtual Demonstrator Environment for Robot Imitation Learning

Technical Report CS-TR-5039

June 2014

Di-Wei Huang[1]*, Garrett E. Katz[1], Rodolphe J. Gentili[2], and James A. Reggia[1]

[1]*Department of Computer Science,* [2]*Department of Kinesiology,*
University of Maryland, College Park, MD 20742

Abstract

Robot imitation learning, where a robot autonomously generates actions required to accomplish a task demonstrated by a human, has emerged as a potential replacement for a more conventional hand-coded approach to programming robots. Many past studies in imitation learning have human demonstrators perform tasks in the real world. However, this approach is generally expensive and requires high-quality image processing and complex human motion understanding. To address this issue, we developed a simulated environment for imitation learning, where visual properties of objects are simplified to lower the barriers of image processing. The user is provided with a graphical user interface (GUI) to demonstrate tasks by manipulating objects in the environment, from which a simulated robot in the same environment can learn. We hypothesize that in many situations, imitation learning can be significantly simplified while being more effective when based solely on objects being manipulated rather than the demonstrator's body and motions. For this reason, the demonstrator in the environment is not embodied, and a demonstration as seen by the robot consists of sequences of object movements. A programming interface in Matlab is provided for researchers and developers to write code that controls the robot's behaviors. An XML interface is also provided to generate objects that form task-specific scenarios. This report describes the features and usages of the software.

Acknowledgement: This work was supported by ONR award N000141310597.

*Email: `dwh@cs.umd.edu`

Contents

1 Introduction **2**

2 Getting started **4**
2.1 Prerequisites 4
2.2 Files 4
2.3 Start 5
2.4 The main screen 5

3 Camera navigation **7**

4 Demonstration interface **7**
4.1 Creating a demonstration video 8
4.2 Grasping and releasing 10
4.3 Moving and rotation 11

5 Robot control **11**
5.1 Manual control 14
5.2 Matlab scripting 14

6 Object generation **17**
6.1 Generating preset objects 18
6.2 Specifying objects using XML 18

7 Conclusion **24**

1 Introduction

Robot programming by demonstration, or imitation learning, refers to the notion that a robot can autonomously learn sequences of actions to accomplish a task that is demonstrated by a human. Imitation learning is widely considered as a possible replacement for the more conventional pre-programming approach, which is difficult and expensive, and does not generalize well to even moderately altered tasks or initial conditions. This software is built to support studies of robot programming by demonstration, by simulating an environment in which a virtual demonstrator, a robot, and task-related objects coexist. The simulated environment is intentionally simplified (e.g., lighting casts no shadows) to drive the efforts away from low-level image processing and towards developing a high-level cognitive framework that incorporates sensorimotor coordination, causal reasoning, mental simulation, and planning.

An example view of the simulator is shown in Figure 1, which presents a 3D virtual world containing a robot, a table, a variety of objects on the tabletop, and provides simulated physics such as gravity, rigid body collisions, and center of mass representation. Figure 2 shows the three interfaces with which a user can interact with the simulator. First, a human demonstrator can manipulate the objects through a graphical user interface (GUI) using mouse inputs (Figure 2(a)). A demonstration containing multiple object manipulations can be recorded as a "video" (a sequence of images) for training the robot. The demonstration video can be edited by undoing unwanted actions. The demonstrator is not embodied (i.e., invisible, hence the name "virtual demonstrator") in the simulated world, and therefore it is not necessary for the robot to capture and understand the demonstrator's motions, to solve the correspondence problem, or to transform between coordinate systems. Instead, learning can be focused entirely on the consequences of the demonstrator's manipulative movements in the task/object space. That is, from the robot's perspective, the objects in the environment move on their own. The hypothesis we base this on is that in many situations, critical information about a demonstration lies in the behaviors of objects being manipulated rather than those of the demonstrator, and therefore an imitation learner can be substantially simplified and simultaneously made more powerful by not seeing the demonstrator. This approach is in contrast to the more popular scenario where humans physically demonstrate tasks.

The second interface of the simulator is a Matlab scripting interface for controlling the simulated robot (Figure 2(b)). The robot is modeled after Baxter®, a bimanual robot produced by Rethink Robotics[†]. Each arm of the simulated robot has 7 joints and a gripper, which can interact with objects in the environment. A camera is mounted on the head of the robot that captures sequences of images about the environment, including the objects in the environment, the demonstrator's object manipulations, and the robot's own movements. These images, along with the robot's proprioceptive information such as joint angles, are sent as sensory inputs to the Matlab scripts that implement the robot's behaviors. Upon receiving these inputs, the Matlab scripts can optionally specify motor commands, such as rotating certain arm joints, to affect the state of the robot, which may in turn affect the states of the objects.

The last interface is an XML interface that specifies properties and initial states (e.g., shapes, colors, locations, orientations, etc.) of objects that exist in the simulated environment (Figure 2(c)). This interface provides a way to generate a predefined set of objects that forms a task scenario for the demonstrator or the robot to work on.

In general, the simulator serves two purposes:

- A testbed for prototyping robot imitation learning. The simulated 3D world provides a situated environment for experimenting with learning methods, without potential fatigue of

[†]`http://www.rethinkrobotics.com/products/baxter/`

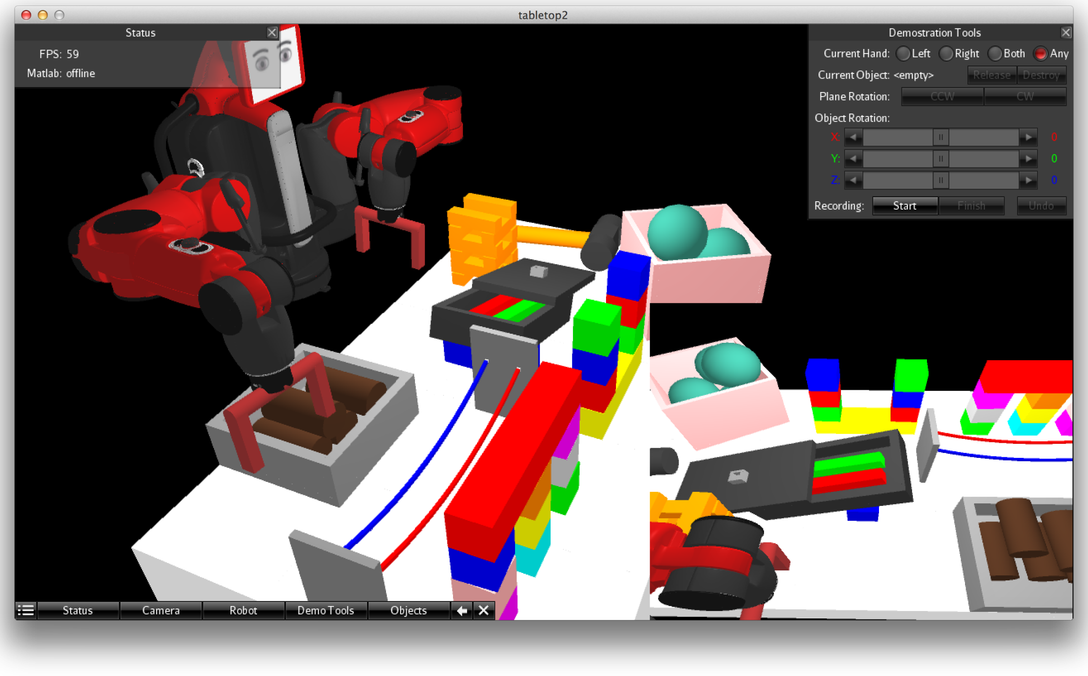


Figure 1: An example view of the simulator

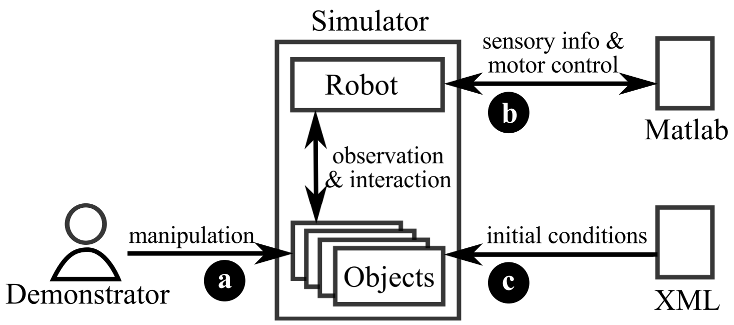


Figure 2: The three primary interfaces of the simulator: (a) the demonstration interface, (b) the Matlab programming interface, and (c) the XML object generation interface.

human demonstrators and without potentially causing damage to physical robots and other equipment. Additionally, training can be done more quickly compared with physical robots because mechanical latency of joint rotation can be reduced or eliminated in the simulation.

- A user interface for robot programming. The intuitive GUI-based demonstration interface provides a convenient way for a human to specify a task to be accomplished by a robot, regardless of whether the robot is simulated or real. This approach requires no special equipments and removes the need to capture and parse complex human motions. Instead, task content is conveyed in terms of object movements. The GUI also provides a simple way to assign symbolic properties about demonstrations, such as specifying that a block being picked up must be red (not supported yet), which, in physical demonstrations, need to be communicated using a separate method such as natural languages. Further, the interface is readily extendible to extreme-sized robots where it is impractical for humans to physically demonstrate a task (e.g., nanorobotics and construction robots).

2 Getting started

The simulator can be run on OS X, Windows, and Linux. Systems with dedicated graphics chips are preferred but not required. This section explains basic operations of the simulator.

2.1 Prerequisites

Before using the simulator, check your system for the following dependencies:

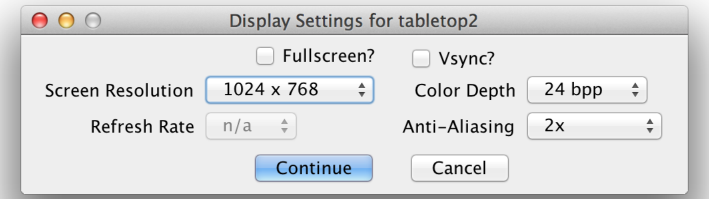
- **Java Runtime Environment (JRE)** is required to run the simulator. Version 1.7 is preferred, although older versions such as 1.6 or 1.5 may work too.
- **Matlab** is required to run robot control scripts in the simulator, but is not required to run the simulator itself. Versions R2007b and greater are supported.

2.2 Files

The files extracted from the distributed zip archive should be placed in a single directory. The essential files and their purposes are:

- `tabletop.jar` – main program.
- `lib/` – directory containing libraries (`*.jar`) required for running the simulator. Normally, it is not necessary to make any changes to this directory.
- `matlab/` – directory where Matlab scripts for robot control should be stored.
- `matlab/agentBehavior.m` – script specifying which Matlab scripts for robot control are to be run by the simulator. See Section 5.2.
- `tablesetup/` – directory where XML files for initializing objects in the environment should be stored.
- `tablesetup/tabletop.xsd` – schema file required to parse XML documents. Normally, it is not necessary to make any changes to this file.

The directories `matlab/` and `tablesetup/` also contain a few examples.


Display Settings for tabletop2

Fullscreen? [ ] Vsync? [ ]

Screen Resolution: 1024 x 768
Color Depth: 24 bpp
Refresh Rate: n/a
Anti-Aliasing: 2x

Figure 3: The display settings dialog

2.3 Start

To start the simulator, run `tabletop.jar` by double clicking the file. Alternatively, use a command line interface and type: `java -jar tabletop.jar` to run. The latter provides full debug messages in the console.

A display settings dialog will then appear (Figure 3). The options are explained below:

- **Screen Resolution** specifies the size of the main screen in pixels. Higher resolutions provide a finer view of the simulated world. A resolution of 1024×768 and higher is recommended.

- **Refresh Rate** sets the number of video frames to be rendered per second. A value of 60 frames per second (FPS), if available, is highly recommended.

- **Color Depth** sets the number of bits used to represent a color. Using the default value works fine in most cases.

- **Anti-Aliasing** sets the extent to which pixelation and jagged edges are eliminated. A value of 2x is recommended.

Pressing the Continue button will launch the simulator's main screen. In general, setting higher values for the above options yields better picture quality but takes more CPU/GPU resources. If a system cannot satisfy option values set by the user, a significant drop of frame rate will occur, e.g., dropping far below 60 FPS. The current frame rate is shown in the Status window (see Figure 4). If this happens, consider decreasing some option values, starting with screen resolution. Restart the simulator to change any display settings.

2.4 The main screen

Figure 4 shows the simulator's main screen, which consists of a simulated 3D world and several overlaid GUI components. The simulated world contains a white tabletop in front of a black background. Two blocks are initially placed on the tabletop in this example. This initial scenario is specified in `tablesetup/default.xml` (see Section 6). A window showing system status is initially located at the top-left corner (Figure 4(a)). All windows like this can be moved by dragging their title bar and hidden by clicking the × button at their top-right corner. An array of buttons at the bottom of the main view (Figure 4(b)–(e)) can be used to: (b) show all windows, (c) show a specific window, (d) align all visible window and place them at the left border of the main view, and (e) hide all windows.

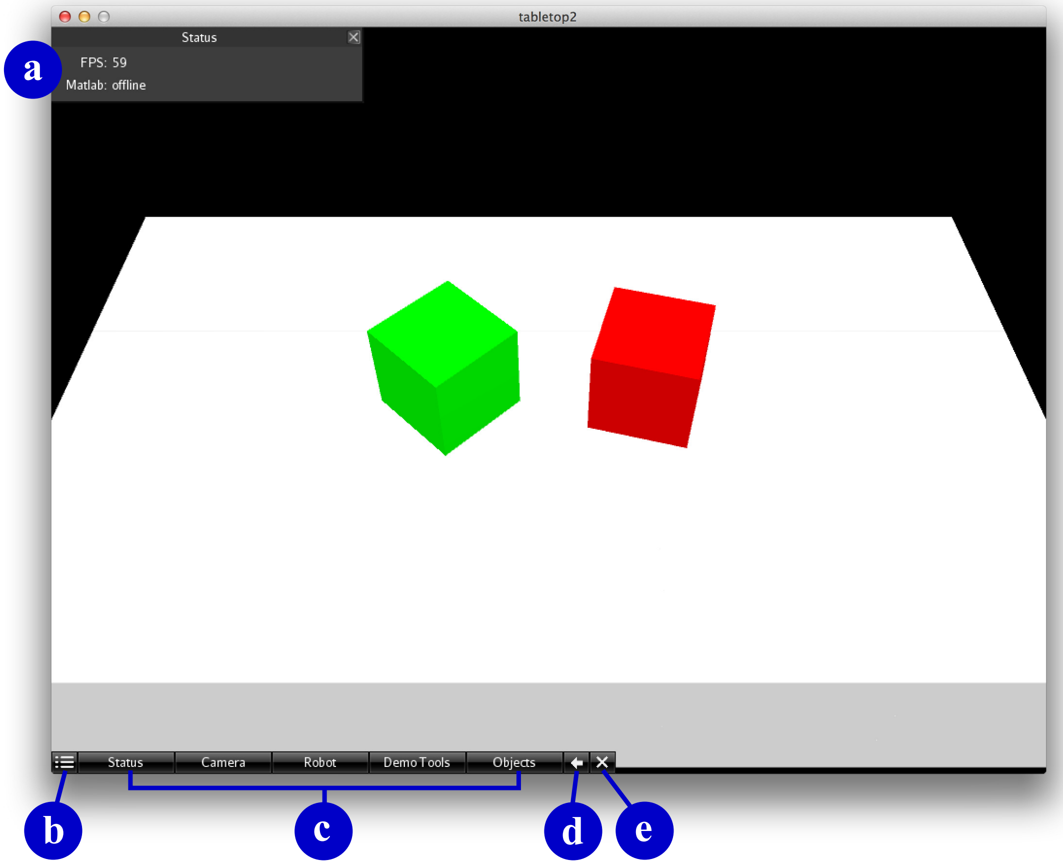


Figure 4: The main screen of the simulator gives a modifiable view of the simulated 3D world and overlaid GUI components. The GUI components include (a) a status window, (b) a button for showing all windows, (c) buttons for showing individual windows, (d) a button for aligning all visible windows, and (e) a button for hiding all windows.

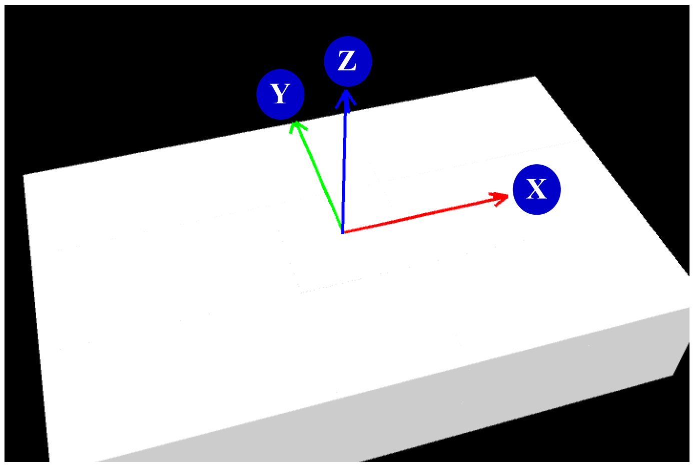


Figure 5: The coordinate system. The origin is located at the center of the tabletop. The XY plane is on the tabletop surface. The Z axis is perpendicular to the tabletop.

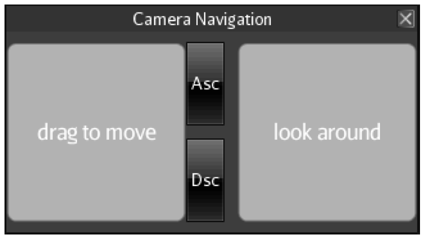


Figure 6: The Camera window that allows the user to navigate in the 3D simulated world.

A 3D cartesian coordinate system (X, Y, Z) is used in the simulated environment (Figure 5). The origin is located at the center of the tabletop surface. The XY plane is horizontal. The tabletop surface lies on the XY plane. From the initial view of the user, and from the robot's view as well, X increases from left to right, Y increases from near to far, and Z increases vertically from low to high. A unit of length in the simulated world corresponds to 10 cm in the real world, while a unit of mass corresponds to 1 kg.

Pressing Space anytime will pause the simulation. Pressing Space again will resume simulation. When the simulator is running, usually a large amount of CPU resources is consumed. Pausing the simulator helps reduce the amount taken by the simulator, especially when the simulator window loses focus. This is useful when the user needs to run another CPU-intensive job while retaining the state of the simulator.

Finally, Pressing Esc anytime will cause the simulator to quit.

3 Camera navigation

The camera navigation window (Figure 6) can be used to change the user's viewing position and direction in the simulated world. To locate the window, click the Camera button at the bottom of the screen.

Dragging the mouse cursor anywhere in the gray area on the left side of the window moves the user's viewing position. Dragging upward moves the viewing position forward, dragging downward moves it backwards, and dragging left or right moves it sideways. A longer dragging distance moves the viewing position at a higher speed. The Asc and Dsc buttons raise and lower the viewing position vertically. Similarly, to change the viewing direction, drag the mouse cursor anywhere in the gray area on the right side of the window. The viewing direction is then rotated in the direction dragged. The dragging distance determines the speed of rotation.

Keyboard shortcuts: W S A D move the viewing position forward, backward, left, and right, respectively; Q Z raise and lower the viewing position; ↑ ↓ ← → rotate the viewing direction.

4 Demonstration interface

With the interface described in this section, the user can demonstrate a task by manipulating objects on the tabletop in the simulated world. The user, or the demonstrator, is not embodied in the simulated world, and thus the result of a demonstration, as observed by the robot, is a sequence of object movements and rotations. A demonstration can be recorded and saved as a "video" (a sequence of images), which is captured by the robot's camera and can be used as the

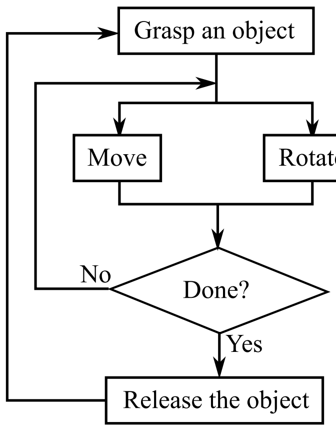


Figure 7: A typical demonstration flow. The demonstrator can grasp an object, move/rotate it until the object is at a desired location with a desired orientation, and release it. Repeat this process with different objects to complete the task being demonstrated.

basis of imitation learning. The video is segmented and annotated according to the demonstrator's actions. In the situation where the demonstrator makes an unwanted move, the simulator provides an "undo" function to restore a previous state.

The demonstrator is equipped with two manipulators, or "hands", which can simultaneously manipulate two objects. Each hand can manipulate an object by grasp, move, rotate, and release it. A typical flow chart for one hand is shown in Figure 7. A manipulation starts by grasping an object, from which point on the object is temporarily unaffected by the gravity, i.e., it is held by an invisible hand. The object then undergoes a series of moving and/or rotation. Finally, when the object has been moved to a desired destination and rotated to a desired orientation, it can then be released. The object resumes being affected by the gravity after it is released. The demonstrator can switch between the two hands while manipulating objects. When the demonstrator is operating one hand, the state of the other hand remains unchanged (unless the hand grasps the object held by the other hand). To manipulate two objects in parallel, the demonstrator needs to manually interleave the manipulative actions of the two hands.

Figure 8 shows the interface for demonstrations. The following subsections explain how demonstration recording and manipulations are done using the GUI. To generate randomly located objects for practicing demonstration, see Section 6.1 (or simply press B or N).

4.1 Creating a demonstration video

Using the Start and Finish buttons in the Demonstration Tools window (Figure 8(f)) to record a demonstration video. Pressing the Start button will start saving demonstration actions to a subdirectory named `demo/`. The video is segmented in a way that every grasp and release action triggers creation of a new segment. Pressing Finish button will stop recording, after which the user may want to rename the `demo/` subdirectory since pressing Start again will overwrite the content of `demo/`. Anytime during a recording, the demonstrator can press the Undo button to discard the current video segment. In this case, the state of the environment will be restored to the previous state accordingly.

The recording captures 4 images, or frames, per second, where each image is given a consecutive frame number starting from 0. Note that if the states (e.g., location) of objects in the environment

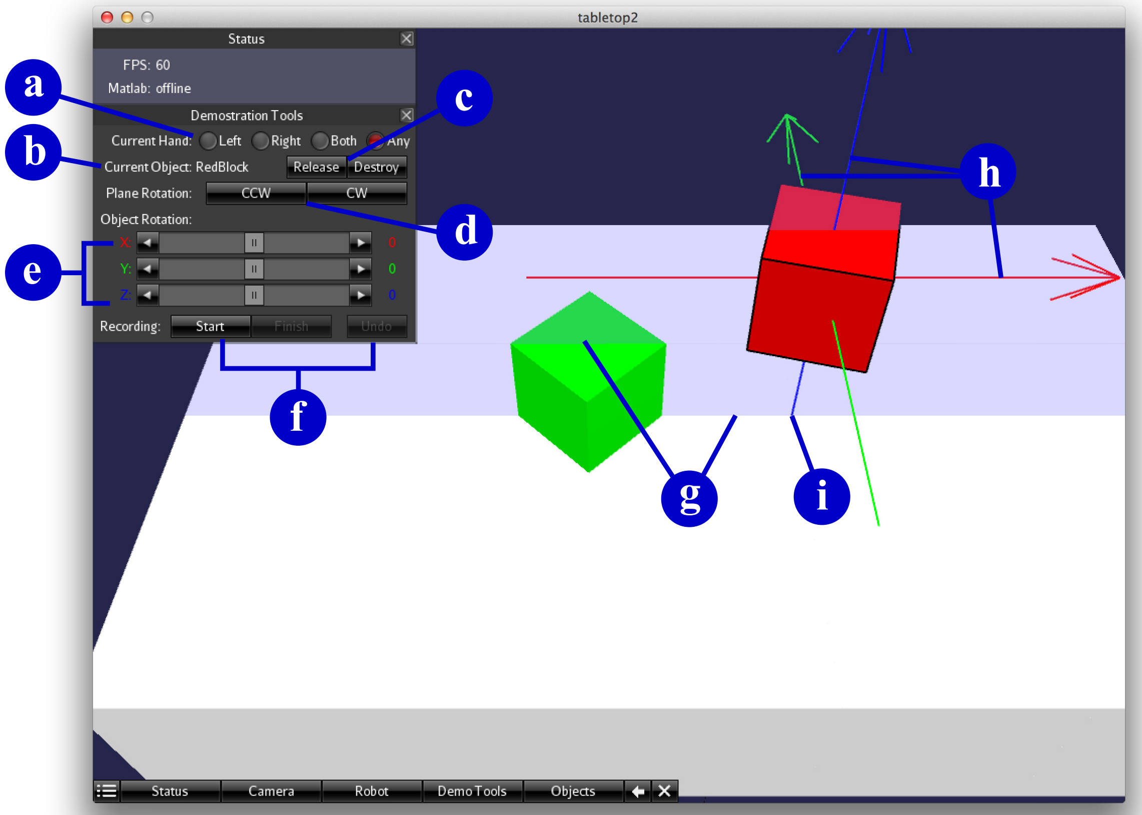


Figure 8: The demonstration interface. The red block (right) is currently grasped by the demonstrator and raised above the tabletop. (a) The radio button controlling which hand is currently in use. (b) The name of the grasped object. (c) The buttons for releasing and removing a grasped object. (d) The buttons for rotating the vertical plane (shown in transparent blue color) on which the movement of the grasped object is restricted. (e) The sliders for rotating the object around three orthogonal axes. (f) The buttons controlling the recording of demonstration videos. (g) The intersections between the vertical plane restricting object movements and the tabletop. The intersections visualize the horizontal path along which the object is moving during the current operation. (h) The three rotation axes for the grasped object. They are colored in accordance to the sliders in (e). (i) The vertical projection cast from the center of the grasped object to the tabletop. It visualizes the current XY location of the grasped object.

have not changed since the last frame, the image file of this frame is not generated (to save storage space), although the frame number keeps increasing to reflect the passing of time. The image files are grouped and placed in subdirectories corresponding to their video segments, which are again numbered consecutively starting from 0. For example, the images of the first segment are stored in `demo/0/`. Along with the image files, an accompanying text file is also created for each segment that symbolically annotates the events occur in the environment. The text file is formatted in comma-separated values (CSV), where each line represents an event. The first value of each line indicates the frame number in which the event occurs, and the second value indicates the type of the event, which can be one of the following:

- `create`. This event is generated whenever a new object first appears in the video. The third value of the line indicates the ID of the object being created. Object IDs are unique. From the fourth value on is a list of key-value pairs indicating the properties of the object, such as dimensions, color, and mass.
- `delete`. This event is generated whenever an object is removed from the environment. The third value indicates the ID of the object removed.
- `move`. This event indicates the change in location or orientation of an object. The third value indicates the ID of the object. The next three values indicate the X, Y, and Z coordinates of the object's new location, followed by three values indicating the object's rotation along the X, Y, and Z axes (in that order).
- `grasp`. This event indicates that the demonstrator grasps an object. The third value contains the name of the hand being used: `LeftHand`, `RightHand`, `BothHands`, or `AnyHand`. The fourth value contains the ID of the object grasped.
- `release`. This event indicates that the demonstrator releases a grasped object. The third value contains the name of the hand released.
- `destroy`. This event indicates that the demonstrator destroys a grasped object. The third value contains the name of the hand released. A `delete` event will also be generated.

4.2 Grasping and releasing

First, select a hand using the radio buttons in the Demonstration Tools window (Figure 8(a)). Grasp an object using the selected hand simply by clicking the object with the mouse cursor. The name of the grasped object will appear (Figure 8(b)), and the object itself will be highlighted by black outlines, a plane to guide movements (Figure 8(g)), and axes to guide rotations (Figure 8(h)). In the situation depicted in Figure 8, the red block (the one on the right) is being grasped. The demonstrator may then move or rotate the grasped object (explained in the next subsection), or switch to using the other hand (by clicking the corresponding radio buttons) to perform other manipulations. To release an object being grasped, first switch to the hand that is holding the object. The object will be highlighted only if the demonstrator switches to the hand that is holding the object. The demonstrator can either click the Release button (Figure 8(c)) or simply click anywhere other than the grasped object. The demonstrator can also click the Destroy button to remove the grasped object from the environment.

4.3 Moving and rotation

To move a grasped object, simply drag it with the mouse cursor. The movement is restricted on a plane perpendicular to the tabletop. The plane is visualized as a transparent blue sheet. Notice where the plane intersects the tabletop and other objects (Figure 8(g)). The intersection indicates a potential "fly by" path for the object being dragged by the demonstrator. Locations along the path are possible destinations where the demonstrator can place the object. For example, in Figure 8, since the plane for moving the red block (the one on the right) intersects the green block (the one on the left), it is possible to place the red block on top of the green block. The plane can be rotated around a vertical axis using the CCW (for counterclockwise rotation) and CW (for clockwise rotation) buttons in Figure 8(d). It is advisable to rotate the plane until a desired destination intersects the plane before dragging the object. (Keyboard shortcuts: `/` and `⇧`+`/` rotate the plane in one direction or the other.)

To rotate the grasped object, first notice that there are three orthogonal axes colored in red, green, and blue in Figure 8(h). They are the axes around which the object can be rotated. Use the sliders in Figure 8(e), which are colored in accordance with the axes, to rotate the object around the corresponding axes. The angles of rotations are shown on the right side of the sliders. Drag the knob on a slider to arbitrarily adjust the rotational angle. Alternatively, click the button at either end of a slider to increase or decrease the angle by 1 degree, or click on the slider track to increase or decrease the angle by 45 degrees. The range of rotation is between -180 and 180 degrees.

The blue axis (the vertical one) also serves as a "virtual shadow" of the selected object (Figure 8(i)), which is a vertical projection of the object's 3D location. It indicates the horizontal location of the object's center when projected vertically onto the tabletop (or other objects).

5 Robot control

This section describes the simulated robot and how its states can be affected/controlled either manually or using Matlab scripts. This interface, corresponding to Figure 2(b), is intended to be used by software developers to implement the robot's cognitive controller.

In the Robot window (Figure 9; press the Robot button at the bottom of the screen to show this window), the Show/Hide button toggles the visibility of the robot (Figure 9(a)). It is recommended to hide the robot temporarily while a demonstration is in progress, because the robot may block the demonstrator's view, and because rendering the robot serves no purposes during demonstrations. The robot can still observe the demonstration when invisible. On the other hand, whenever the robot needs to interact with the environment, it has to be visible. The Head View button in the Robot window toggles the display of the robot's vision. The robot "sees" the environment through its head-mounted camera. The environment as seen by the robot is displayed in a picture-in-picture style at the lower right corner (Figure 9(b)). (Keyboard shortcuts: `R` to show/hide the robot; `1` to toggle the robot's view; `2` to save the robot's view as an image file.)

The simulated robot is modeled after Baxter®, a commercial bimanual robot. Each arm consists of 7 joints (see Figure 10). Joints S0 and S1 are located at the "shoulder", joints E0 and E1 are at the "elbow", and joints W0, W1, and W2 are at the "wrist". Additionally, a head joint H0 allows the robot's head, together with the virtual camera mounted on it, to pan horizontally. The robot's field of view (Figure 9(b)) is updated according to H0. The 3D model and the arm configurations of the robot are extracted from data released by the manufacturer[‡]. The ranges of the joint angles are listed in Table 1 for reference.

[‡]`https://github.com/RethinkRobotics/`

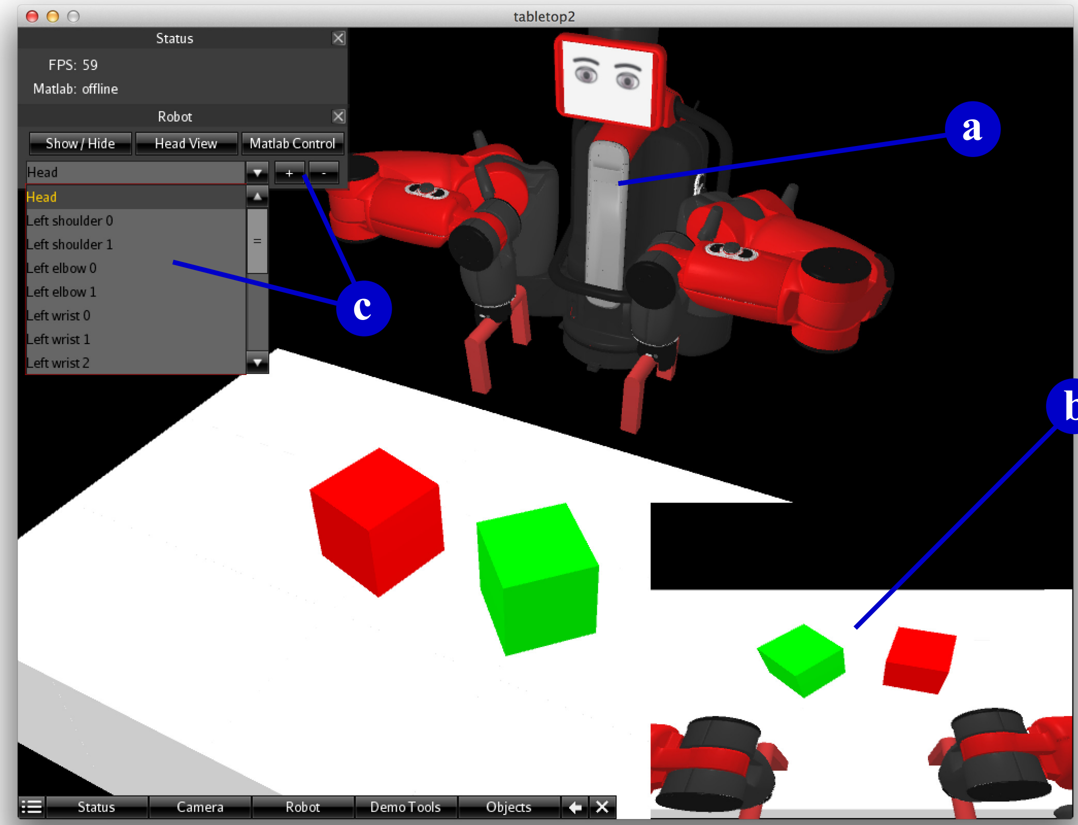


Figure 9: The robot in the simulated world. (a) The 3D model of the robot. (b) The robot's view of the simulated world. The images are captured by its head-mounted virtual camera. (c) GUI components for manually controlling the robot.

Table 1: The ranges of the robot's joint angles (in radians).

Joint	Angle range	Joint	Angle range
S0	-1.7 – 1.7	W0	-3.059 – 3.059
S1	-2.147 – 1.047	W1	$-\pi/2$ – 2.094
E0	-3.054 – 3.054	W2	-3.059 – 3.059
E1	-0.050 – 2.618	H0	$-\pi/2$ – $\pi/2$

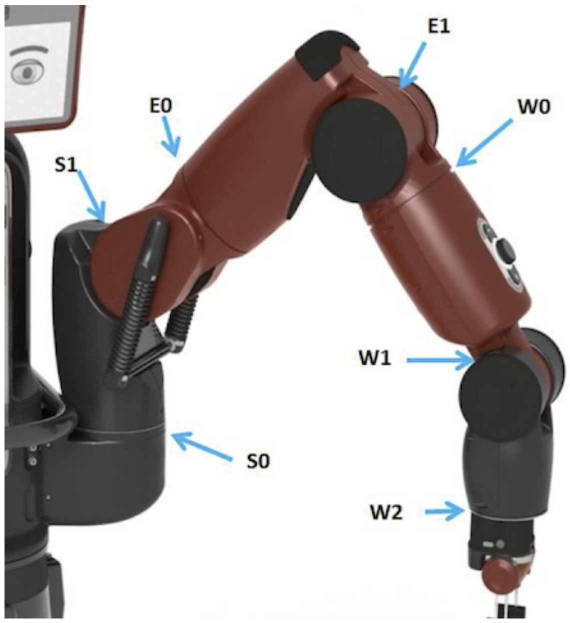


Figure 10: The locations of arm joints for Baxter®.

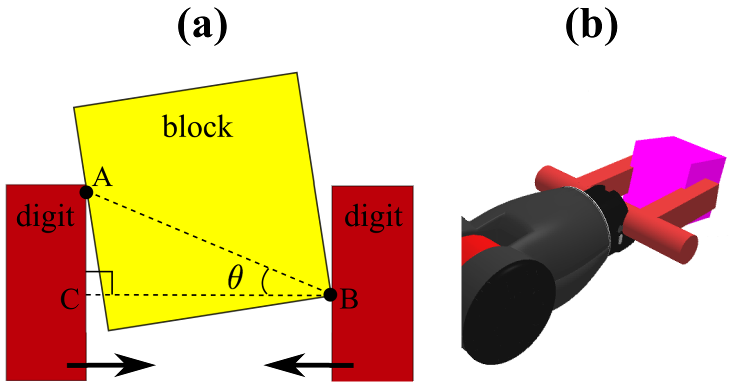


Figure 11: (a) The top view of a gripper attempting to grasp an object. A and B are the contact points between the object and the two digits of the gripper. $\overline{BC}$ is a line perpendicular to the inner surfaces of the gripper. If θ is sufficiently small, the block is considered being held by the gripper. (b) The 3D view of a gripper holding an object.

A gripper is attached at the end of each arm. The distance between the two digits of a gripper can be widened or narrowed, so the gripper can hold or release objects. The maximum inner distance between the two digits is 2 units (corresponding to 20 cm). When both of the digits are in contact with the same object while the gripper is closing, a test depicted in Figure 11(a) is performed to determine if the grip is successful. A and B are the contact points, and θ is the angle between $\overline{AB}$ and $\overline{BC}$, which is a line perpendicular to the contact surfaces on the two digits. The grip is successful if $\theta < \pi/16$. If the grip is successful, the object is considered being held by the gripper. When a gripper opens, all objects being held by it are released.

In the following subsections, two methods of controlling the robot are discussed. The manual control method serves mainly debugging purposes, which allows developers of the robot controller to directly alter joint angles of the robot through GUI. The Matlab scripting method provides a programmatic way of implementing the robot's behaviors.

5.1 Manual control

To manually control the robot, simply use the drop down box and the +/- buttons (Figure 9(c)). The drop down box contains all joint angles, including 7 joints on each arm, the head joint, and both grippers. Select a joint and then click the + or - button to rotate the joint in one direction or the other.

Keyboard shortcuts: use T, Y, U, I, O, P, [(of the same row on a common keyboard) to rotate the W2, W1, W0, E1, E0, S1, and S0 joints of the right arm; similarly, use G, H, J, K, L, ;, ' to rotate the respective joints of the left arm; use] to rotate the head joint H0. Hold ⇧ when pressing the above keys to rotate the corresponding joints in the other direction. Use =/- to open/close the right gripper, and 0/9 to open/close the left gripper.

5.2 Matlab scripting

The simulator provides an interface to allow the user's Matlab scripts to control the simulated robot. The "user" in this context refers to the software developers of the robot's cognitive controller. Through this interface, the robot's behaviors, including its imitation learning mechanism and the way it reacts to environmental stimuli, can all be implemented in Matlab, which is known to be fast in prototyping. This subsection describes how the user's Matlab scripts can interact with the simulator by receiving sensory inputs and generating motor outputs in real time.

The user needs to provide two scripts: an initialization script and a callback script, both placed in the subdirectory `matlab/`. The initialization script is called once by the simulator before any calls to the callback script. The callback script is called repeatedly afterwards whenever the simulator is updating the main screen. If the simulator is running at 60 FPS, the callback script is also called 60 times per second. At each call to the callback script, the simulator supplies the callback script with the robot's sensory information (e.g., visual images and current joint angles). The simulator then retrieve motor commands (e.g., joint velocities) that the callback script may issue and executes them on the robot. Note that the calls to the user's scripts are *blocking*, meaning the simulator will always wait for the user's scripts to complete before it processes the next update of the main screen. Therefore, it is important to keep the user's scripts fast, or the frame rate may drop significantly. When the frame rate drops below ∼12 FPS, obvious visual latencies and strange physics effects may occur.

To specify the filenames of the user's scripts, edit `matlab/agentBehavior.m`, whose content is shown below, and replace `__init__` and `__callback__` with the filenames of the user's initialization and callback script.

```
function [init, callback] = agentBehavior()
init = @__init__; % fill in the name of the initialization script
callback = @__callback__; % fill in the name of the callback script
end
```

In the Robot window (Figure 9), clicking the Matlab Control button will start invoking the user's initialization and callback scripts. The simulator does so by automatically launching the Matlab environment (if Matlab is installed) before calling the user's scripts. Therefore, it is not necessary to manually launch the Matlab environment before running the simulator. Clicking the Matlab Control button again will stop invoking the user's scripts. Whether the user's scripts are currently being called by the simulator is indicated in the Status window. (Keyboard shortcut: M to toggle the invocations of the user's Matlab scripts.)

The simulator communicates with the user's callback scripts through three global Matlab `struct` variables: `sensor`, `motor`, and `aux`. `sensor` is set by the simulator to pass sensory information to the user's scripts, `motor` can be set by the user's scripts to pass motor commands to the simulator, and `aux` is used to exchange auxiliary information between the simulator and the user's scripts.

A sample of the `sensor` variable is printed below:

```
sensor =
       timeElapsed: 0.0283
       jointAngles: [2x7 double]
         endEffPos: [2x3 double]
    gripperOpening: [2 2]
         rgbVision: [150x200x3 double]
```

The `sensor` variable includes the following fields:

- `timeElapsed` is the time elapsed (in seconds) since the last time the callback script is invoked.
- `jointAngles` is a 2-by-7 matrix containing the current joint angles of all arm joints. The first row corresponds to the left arm and the second to the right arm. The columns are indexed in the order of: S0, S1, E0, E1, W0, W1, W2.
- `endEffPos` is a 2-by-3 matrix containing the (X, Y, Z) coordinates of two end effector positions. An end effector is located at the center of a gripper. The first row corresponds to the left gripper and the second to the right. Columns are indexed in the order of: X, Y, Z.
- `gripperOpening` contains the current widths of the grippers. The width is measured as the inner distance between the two digits of each gripper. The first element corresponds to the left gripper and the second to the right gripper.
- `rgbVision` contains the visual image captured by the head-mounted virtual camera of the robot. The image size is 150 (height) by 200 (width) pixels. The first two indices of the field refer to pixel locations. The third index refers to the red, green, and blue intensities of each pixel. Note that `rgbVision` may not be available in every invocation of the callback script due to performance considerations. Therefore, it is important to test the existence of this field before accessing it, such as:

```
if any(strcmp('rgbVision', fieldnames(sensor)))
    image(sensor.rgbVision); % draw the image whenever it is available
end
```

A sample of the `motor` variable is printed below:

```
motor =
      jointVelocities: [2x7 double]
    gripperVelocities: [2x1 double]
```

The `motor` variable includes the following fields, which are set by the user's scripts:

- `jointVelocities` is a 2-by-7 matrix setting the intended rotational velocities for the arm joints (radian per second). The first row corresponds to the left arm and the second to the right. The columns are indexed in the order of: S0, S1, E0, E1, W0, W1, W2.
- `gripperVelocities` sets the intended opening/closing velocities for the grippers. Positive values open the grippers and negative values close them. The first element corresponds to the left gripper and the second to the right.
- `jointAngles` (not shown in the above sample) is an optional 2-by-7 matrix that sets each joint instantly to a specific angle. If this field exists, the `jointVelocities` field will have no effects. The indexing of the matrix is identical to `jointVelocities`.

A sample of the `aux` variable is printed below:

```
aux =
               path: 'matlab/'
           numLimbs: 2
          numJoints: 7
     minJointAngles: [2x7 double]
     maxJointAngles: [2x7 double]
    initJointAngles: [2x7 double]
        drawMarkers: [4x3 double]
```

Each field of `aux` is explained below:

- `path` contains a string specifying the path that contains the users' Matlab scripts. Currently this field is a constant string "`matlab/`".

  The default working directory of the user's scripts is at the root directory of the simulator (where `tabletop.jar` is at). It is suggested that the user's scripts access files contained in the `matlab/` subdirectory only. To do so, simply prepend this field to any filenames appearing in the user's scripts. The user's scripts should not modify the value of this field.
- `numLimbs` contains the number of limbs the robot has. A bimanual robot has two arms, and hence the value 2. The user's scripts should not modify the value of this field.
- `numJoints` contains the number of joints per limb. The simulated robot current has 7 joints per arm. The user's scripts should not modify the value of this field.
- `minJointAngles` is a 2-by-7 matrix containing the lower bounds of the joint angles. The first row corresponds to the left arm and the second to the right arm. The columns are indexed in the order of: S0, S1, E0, E1, W0, W1, W2. The user's scripts should not modify the values of this field.
- `maxJointAngles` is a 2-by-7 matrix containing the upper bounds of the joint angles. The matrix indexing is identical to `minJointAngles`. The user's scripts should not modify the values of this field.

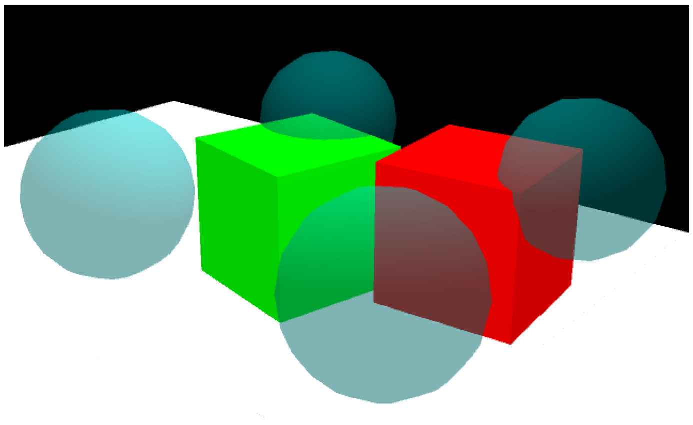


Figure 12: Visual markers are shown as transparent blue spheres. They do not interact with objects.

- `initJointAngles` is an optional 2-by-7 matrix to be set by the user's initialization scripts. It specifies the initial joint angles of the robot. The matrix indexing is identical `minJointAngles`.
- `drawMarkers` is an optional N-by-3 matrix to be set by the user's initialization scripts, to visualize N spatial locations in the simulated world. Each row specifies an (X, Y, Z) coordinates where the simulator will draw a "marker" in the simulated world. A marker is visualized as a transparent blue sphere (Figure 12). The markers serve visualization purposes only and do not interact with objects.
- `exit` (not shown in the above sample) is an optional field that, once set by the user's scripts, disables the Matlab interface. The simulator stops invoking the user's scripts from this point on. All joint and gripper velocities are set to 0. All markers drawn are cleared.

`matlab/exampleInit.m` and `matlab/exampleCallback.m` provide a simple example for writing initialization and callback scripts. They rotate all arm joints simultaneously at a constant speed. This example shows the content of `sensor`, `motor`, and `aux`, and how they are used to communicate with the simulator. To run the example, first modify `matlab/agentBehavior.m` to specify the name of the example, as follows:

```
function [init, callback] = agentBehavior()
init = @exampleInit;
callback = @exampleCallback;
end
```

and then click the Matlab Control button in the Robot window to start.

6 Object generation

This section describes how objects on the tabletop are generated. In Figure 13, the Objects window contains two ways of generating objects: by using an XML file (the top row in the window), or by manually generating preset objects (the bottom row in the window).

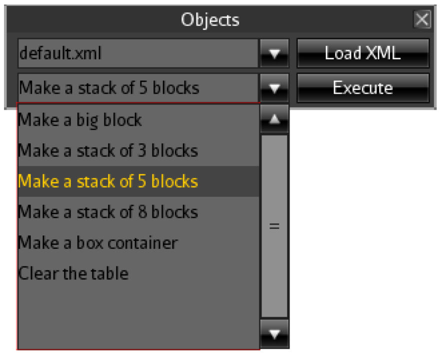


Figure 13: The Objects window for generating tabletop objects.

6.1 Generating preset objects

Preset objects are available for generation in the second drop down box in the Objects window (Figure 13). Select one of the options and click the Execute button to generate the specified objects. Preset objects are generated in random colors and at random locations on the tabletop. A "Clear the table" option is also provided to remove all objects from the simulated world. (Keyboard shortcuts: N to generate a stack of 5 blocks; B to generate a larger block; C to clear all objects.)

6.2 Specifying objects using XML

Objects can also be generated by loading XML files provided by a user. An XML file specifies a set of objects, along with their properties such as sizes, colors, locations, etc. The XML file can also define composite objects, which are composed of multiple basic objects. In general, using XML provides more flexibility in specifying objects for complex task scenarios than merely generating preset objects randomly.

The simulator always loads `tablesetup/default.xml` when it is launched. The user can load custom XML files in the top row of the Objects window. All XML files stored in the directory `tablesetup/` are listed in the drop down box. Selecting a filename in the drop down box and clicking the Load XML button will load the XML file.

The root element of an XML document has to be a `<tabletop>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://synapse.cs.umd.edu/tabletop-xml␣tabletop.xsd␣"
        xspan="20" yspan="12">
    ...
</tabletop>
```

where `xspan` and `yspan` specify the size of the table. The `<tabletop>` element can contain a list of other elements (in the place where the ellipsis above is located), each of which describes an object. Different types of objects are described using different element names (listed below), such as `<block>` and `<sphere>`. The properties of each object (e.g., color) can be specified by assigning attribute values. For example, a $2 \times 2 \times 2$ green cube, whose center is located at the coordinates (-2, 0, 1) with a 45-degree rotation around the Z axis, can be described by the following XML element:

```
    <block id="GreenBlock" xspan="2" yspan="2" zspan="2" location="(-2,0,1)"
        rotation="(0,0,45)" color="green" />
```

All simple objects, including `<block>`, `<cylinder>`, `<sphere>`, and `<box>`, share the following common attributes:

- `id` specifies the unique identifier of the object. When an object is grasped in a demonstration, the name of the object will appear in the Demonstration Tools window. If the `id` value specified by in the XML file is not unique, the simulator will append strings or generate a new string to enforce the uniqueness of the identifiers. **Default value:** a random unique string.

- `location` specifies where the *center* of the object is to be placed in the simulated world, in the format of `(x,y,z)`. `x`, `y`, and `z` are numbers specifying the coordinates of a 3D location. Positioning an object should prevent the object from overlapping with other objects or with the table, or unexpected physics effects may occur. For example, placing a unit cube at (0, 0, 0) causes the bottom half of the cube to intersect the table, which may result in the cube sinking into the table. Instead, place it at e.g., (0, 0, 0.5). **This attribute is required.**

- `rotation` specifies the orientation of the object. The same format as `location`, `(x,y,z)`, is used, where `x`, `y`, and `z` are the angles (in degrees) that this object is to be rotated around the X, Y, and Z axis, respectively. For the same reason explained in `location`, orienting an object should also prevent the object from overlapping with other objects and with the table. **Default value:** (0, 0, 0).

- `mass` specifies the weight of the object in kilograms. **Default value:** 1.

- `color` specifies the color of the object. Predefined colors can be assigned using the following values: black, blue, brown, cyan, darkgray, gray, green, lightgray, magenta, orange, pink, red, white, and yellow. User-defined colors can be specified using the format `#rrggbb`, where each letter is a hexadecimal digit, and `r`, `g`, `b` corresponds to the red, green, and blue components of the color. For example, `#00A57F` defines a bluish green color. **Default value:** gray.

Each type of simple objects is described below:

- `<block>` describes a solid cuboid, whose dimensions are specified by the attributes: `xspan`, `yspan`, and `zspan`. Their values specified the length of the block along the X, Y, and Z axes. **Default values:** `xspan="1" yspan="1" zspan="1"`.

  Example: a red block at the center of the table (Figure 14(a)).

  ```
  <block id="block1" location="(0,0,1.5)" color="red" mass="0.5"
      xspan="5" yspan="4" zspan="3" />
  ```

  Note that the Z coordinate of the location is set to 1.5 so that the bottom half of the block does not overlap with the table.

- `<cylinder>` describes a solid circular cylinder. Use the `yspan` attribute to specify the length of the cylinder, and the `radius` attribute to specify the radius. **Default values:** `yspan="1" radius="0.5"`.

  Example: a green cylinder (Figure 14(b)).

```
<cylinder id="c1" location="(0,0,2)" color="green" mass="0.5"
    yspan="7" radius="2"/>
```

- `<sphere>` describes a solid ball. The radius can be specified by the `radius` attribute. **Default value:** 1.

  Example: a blue sphere (Figure 14(c)).

  ```
  <sphere id="s1" location="(0,0,2)" color="blue" mass="0.5" radius="2"/>
  ```

- `<box>` describes a cuboid container with open top and hollow center. The attributes `xspan`, `yspan`, and `zspan` (see `<block>`) are used to specify the exterior dimensions of the box. Additionally, the `thickness` attribute specifies the the thickness of the walls. **Default values:** `xspan="1" yspan="1" zspan="1" thickness="0.05"`.

  Example: an orange box container (Figure 14(d)).

  ```
  <box id="box1" location="(0,0,1.5)" color="orange" mass="1"
      xspan="5" yspan="4" zspan="3" thickness="0.2" />
  ```

In addition to the simple objects described above, the following complex objects are also supported:

- `<composite>` describes an user-defined rigid object composed of multiple simple objects. The constituent objects are described by elements listed in the content of the `<composite>` element. A `<composite>` element can also contain other `<composite>` elements, forming a tree structure. The top-level `<composite>` describes the whole rigid object. The supported attributes for the top-level `<composite>` are : `id`, `location`, `rotation`, and `mass`. The `mass` attribute can be specified only at the top-level, but not in any constituent objects.

  The `location` and `rotation` attributes of constituent objects are not absolute but are relative to the center of the containing `<composite>` object. For example, specifying `location="(1,1,1)"` in a constituent object does not mean the absolute coordinates (1, 1, 1) in the simulated world, but an offset of (1, 1, 1) away from the center of the containing object. The center of a `<composite>` object is treated as the center of mass. While the position and the location of the top-level `<composite>` object should not overlap with other objects or with the table, it is fine for its constituent objects to overlap with each other, because they are all considered as the same rigid object.

  Example: a hammer (Figure 15).

  ```
  <composite location="(0,0,1)" rotation="(0,0,-30)" mass="10" id="hammer">
      <composite id="head" location="(-0.2,0,0)">
          <block color="darkgray" id="head0" location="(0,0,0)"/>
          <cylinder color="darkgray" id="head1" radius="0.75" yspan="0.8"
                  location="(0,0.9,0)"/>
          <cylinder color="darkgray" id="head2" radius="0.75" yspan="0.8"
                  location="(0,-0.9,0)"/>
      </composite>
      <cylinder id="handle" color="orange" radius="0.35" yspan="5"
              rotation="(0,0,90)" location="(2.8,0,0)"/>
  </composite>
  ```

  Notice how the handle is offset further from the center of the object than the head piece, such that the center of mass is near the center of the head piece.

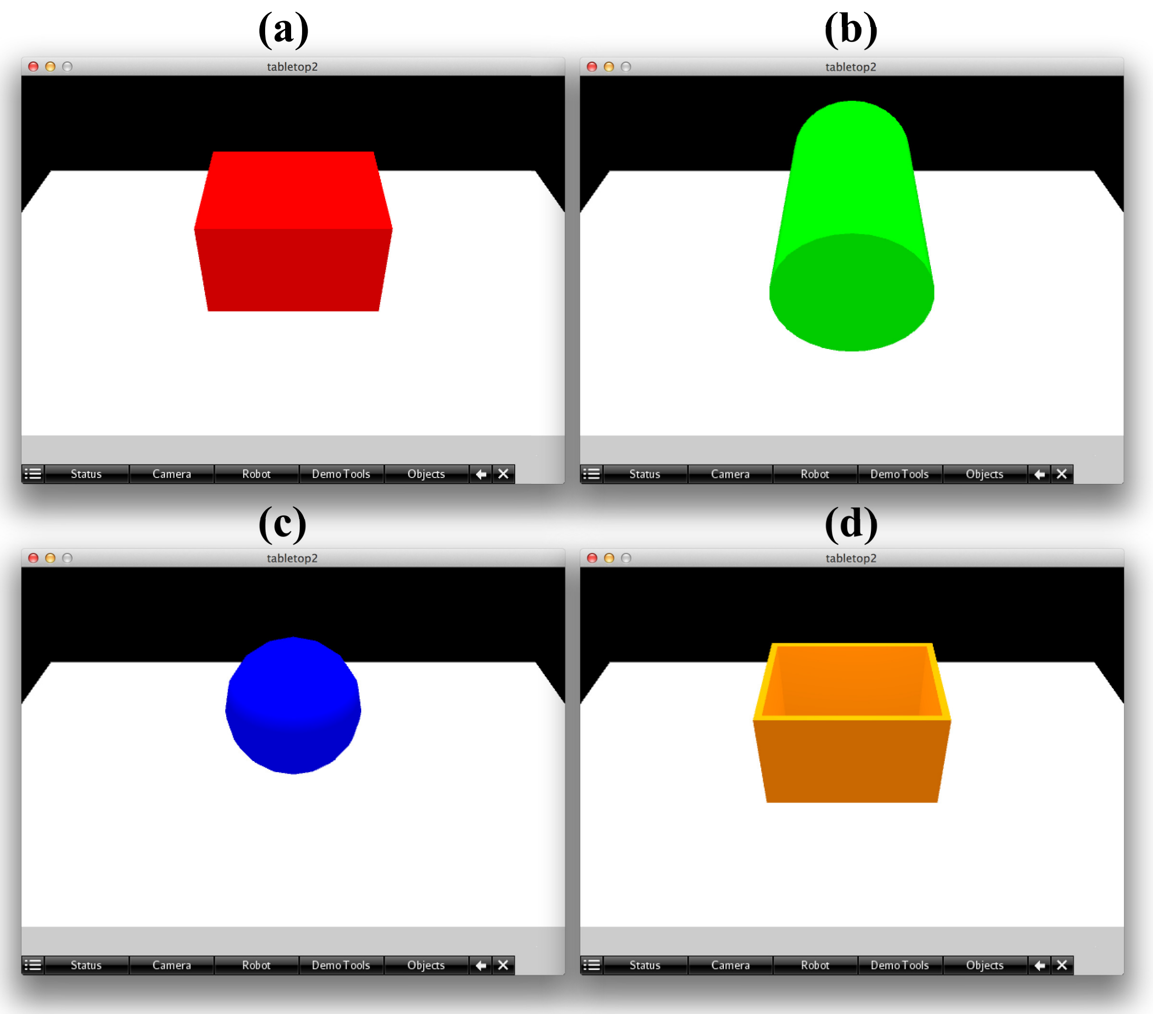


Figure 14: Simple objects that are located near the center of the tabletop in their default orientation: (a) a red block, (b) a green cylinder, (c) a blue sphere, and (d) an orange box.

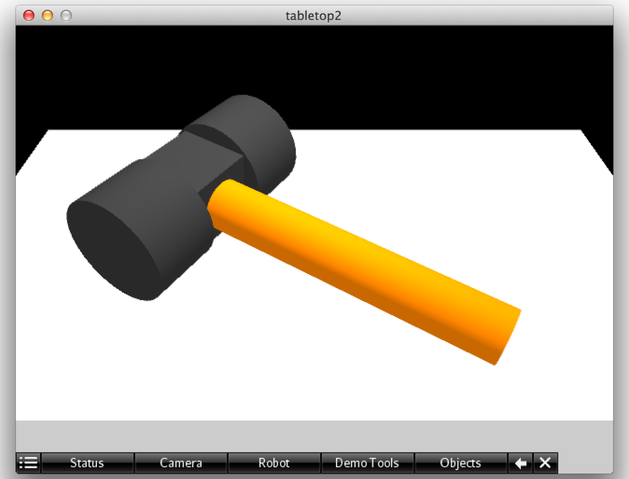


Figure 15: A composite object example. The hammer-like shape is composed of an orange handle and a dark gray head. The head is in turn composed of a cube and two cylinders.

- `<chain>` describes a set of linearly connected links that mimics a string. Links are narrow-shaped blocks. A chain is not rigid as a whole. Instead, the physics of each link are simulated independently, except that physics constraints are posed to "pull" connected links together. The two ends of a chain are fixed at user-specified locations. A chain can be disconnected (i.e., broken into two pieces) by removing one of its links, using the the Destroy button in the Demonstration Tools window. The attributes of a chain are:
  - `id` specifies the identifier of the chain. This identifier will be used as a prefix to generate identifiers for the links that compose this chain. **Default value:** an auto-generated unique string.
  - `color` specifies the color of the chain. All links in a chain share the same color. **Default value:** gray.
  - `start` and `end` specify the fixed 3D locations of the two ends of the chain, in the format of `(x,y,z)`. **These attributes are required.**
  - `linkXspan`, `linkYspan`, and `linkZspan` specify the dimensions of a link. A link is simply a narrow `<block>` object with its longest side lying on the Y axis. **Default values:** `linkXspan="0.1" linkYspan="1" linkZspan="0.1"`
  - `linkCount` specifies the number of links composing the chain. The value must be large enough such that `linkCount` × `linkYspan` is enough to cover the distance between `start` and `end`. Otherwise, the chain may not be generated. **Default value:** 0.
  - `linkPadding` specifies the added length at both ends of each link (in the same direction as `linkYspan`). The paddings aim to visually cover gaps between links, so as to make the appearance of the chain resemble a string. The paddings are for visual purposes only. They do not interact with other objects. **Default value:** 0.05.
  - `linkMass` specifies the weight of each link. **Default value:** 1.

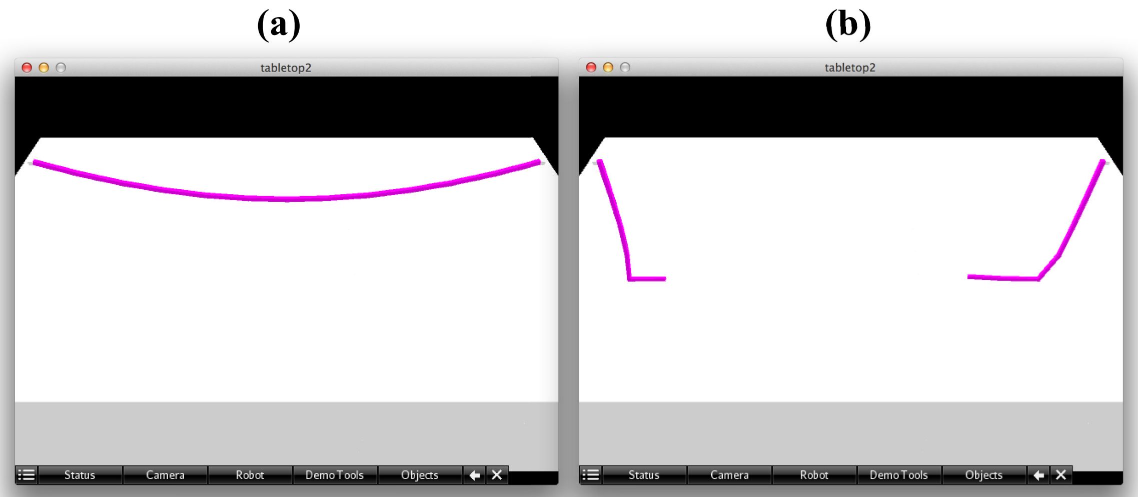


Figure 16: (a) A chain object with its both ends fixed above the tabletop. (b) After one of its links is removed, the remainder of the chain falls down to the tabletop.

Example: (Figure 16(a))

```
<chain color="magenta" start="(6,-2,4)" end="(-6,-2,4)" linkCount="12"/>
```

- `<lidbox>` describes a box container with a lid that slides open in the negative X direction (Figure 17(a)). A small box-shaped handle is located at the center of the lid. The physics of the box container and the lid are simulated independently, except that a physics constraint is applied to both of the objects to realize the sliding relationship. Attributes supported by a `<box>` are also supported here, including `id`, `location`, `rotation`, `color`, `mass`, `xspan`, `yspan`, `zspan`, and `thickness`. The attributes `xspan`, `yspan`, and `zspan` specify the exterior dimensions. Additionally, the following attributes are used to describe the lid handle:
  - `handleColor` species the color of the handle. **Default value:** gray.
  - `handleXspan`, `handleYspan`, and `handleXspan` species the sizes of the handle. **Default value:** 0.5.

  Example: (Figure 17(a))

```
<lidbox xspan="6" yspan="5" zspan="5" thickness="0.6" location="(0,0,2.5)"
    rotation="(0,0,0)" color="#3333bb" mass="10"
    handleXspan="1" handleYspan="2" handleZspan="0.6" handleColor="#33bb33"/>
```

  To achieve the same orientation as shown in Figure 17(b), modify the above example to use the following attribute values:

```
location="(0,0,3)" rotation="(0,90,-90)"
```

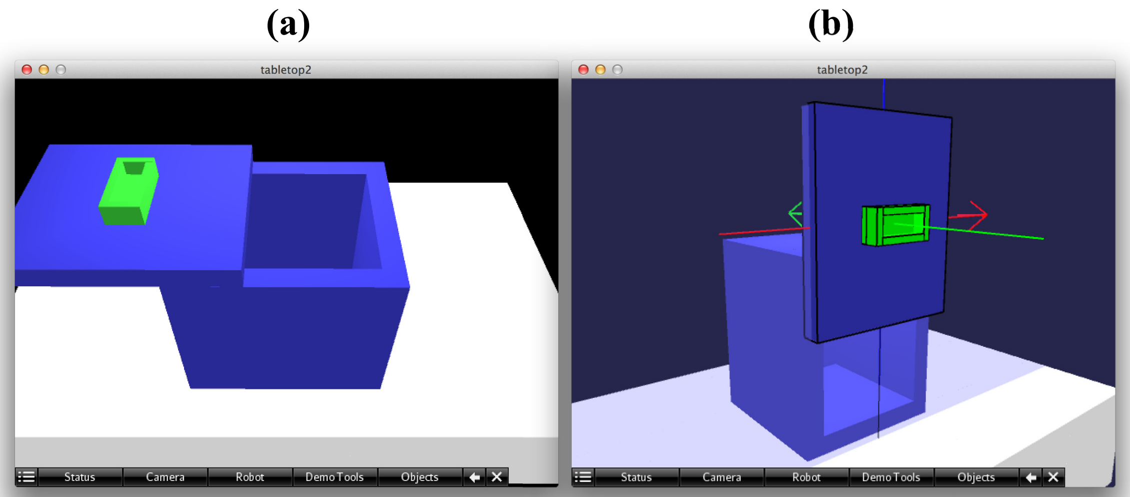


Figure 17: (a) A box container with a sliding lid that slides open in the negative X direction. (b) The same box oriented vertically such that the sliding lid acts as a trapdoor.

7 Conclusion

This report has described the features of software that simulates a virtual environment including a robot, a tabletop, and movable objects along with their rigid-body physics. The primary ways of interacting with the software are through the three interfaces it provides. First, the demonstration interface provides an intuitive GUI for a human to demonstrate a task in the virtual environment using mouse inputs. We introduced the notion of virtual demonstrator in the sense that the robot cannot see the demonstrator's body, but only the movement of the objects that are manipulated by the demonstrator. Demonstrations can be recorded and saved as videos, which are to be used as a basis for imitation learning. Second, the Matlab programming interface for robot control allows software developers to write code that controls a robot's behaviors, including its learning methods and the ways it responds to the environmental stimuli in real time. Finally, the XML interface for object generation provides a way to define and initialize objects in the environment. These objects can then be manipulated by the demonstrator and/or the robot.

In the future, we plan to add more object types to the simulator, including functional objects such as scissors that can be used to cut a wire, and buttons that, when pushed, trigger predefined changes in the environment. The simulation of the robot can be improved by including finer physics details such as torques. We will also work on porting the Matlab interface that this simulator supports to a real Baxter®, such that Matlab scripts that work in the simulator also work in a real robot.