SMILE: Simulator for Maryland Imitation Learning Environment

Technical Report CS-TR-5049

May 2016

Di-Wei Huang[1]*, Garrett E. Katz[1], Rodolphe J. Gentili[2], and James A. Reggia[1,3]

[1]*Department of Computer Science,* [2]*Department of Kinesiology,* [3]*UMIACS*
University of Maryland, College Park, MD 20742

Abstract

As robot imitation learning is beginning to replace conventional hand-coded approaches in programming robot behaviors, much work is focusing on learning from the actions of demonstrators. We hypothesize that in many situations, procedural tasks can be learned more effectively by observing object behaviors while completely ignoring the demonstrator's motions. To support studying this hypothesis and robot imitation learning in general, we built a software system named SMILE that is a simulated 3D environment. In this virtual environment, both a simulated robot and a user-controlled demonstrator can manipulate various objects on a tabletop. The demonstrator is not embodied in SMILE, and therefore a recorded demonstration appears as if the objects move on their own. In addition to recording demonstrations, SMILE also allows programing the simulated robot via Matlab scripts, as well as creating highly customizable objects for task scenarios via XML. This report describes the features and usages of SMILE.

Acknowledgements: This work was supported by ONR award N000141310597. We thank John Purtilo and Charmi Patel for assistance in the implementation of SMILE. This report is an update for CS-TR-5039 published in June 2014.

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

Contents

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 potential 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. However, there is currently no consensus regarding the most effective way to represent demonstrations for robot learning.

A popular method is to represent a demonstration as a sequence of human motion trajectories. This method typically involves a human physically demonstrating a task, during which the demonstrator motions are recorded. Although this method makes it intuitive for the human to demonstrate, processing and learning from demonstrator's trajectories is generally hard. For example, sensing or recording human motions usually requires special hardware, and the recorded trajectories are typically noisy and ambiguous. Complex processing may be needed to transform coordinates, to segment, and to classify the trajectories. Also, the robot may not be able to follow human trajectories due to distinct limb/body configurations/capabilities (e.g., sizes of limbs, ranges of joint rotations, etc.). More importantly, blindly reproducing human trajectories is unlikely to achieve the same effects in the task space as done by the human demonstrator, and inferring the effects achieved by the demonstrator's motion trajectories is difficult.

Since processing human motions is hard, we propose to consider an alternative where the demonstrator is made *invisible* to the robot during a demonstration, according to our *virtual demonstrator hypothesis* (Huang et al., 2015a,b). The rationale is that critical information about a procedural task is likely to lie 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. As such, learning can be focused entirely on the consequences of the demonstrator's manipulative movements in the task/object space.

Our software, SMILE (Simulator for Maryland Imitation Learning Environment), is built to support studies of robot imitation learning based on the virtual demonstrator hypothesis. SMILE provides a simulated 3D environment, in which a virtual (invisible) demonstrator, a robot, a tabletop, and a variety of task-related objects coexist. An example view of SMILE is shown in Figure 1. SMILE also incorporates simulated physics such as gravity, rigid body collisions, and center of mass representation. SMILE has recently been used successfully as the basis for developing a new approach to imitation learning that is based on cause-effect reasoning (Katz et al., 2016).

There are several advantages of using a simulated environment over the physical environment. A major advantage is that it is easy to manipulate the visual appearances of the environment, such as hiding the demonstrator from the robot and intentionally simplifying visual properties (e.g., lighting casts no shadows). The latter is to drive the efforts away from low-level image processing and towards developing a high-level cognitive framework that learns generalized task procedures. Also, demonstrations can be recorded in the simulated environment using commodity computer hardware without special equipment (e.g., motion trackers), and it can be easily operated by one person alone. Demonstrating in the simulated environment can reduce human risks and fatigue, and avoid extreme emotional responses (e.g., dangerous tasks). Additionally, object states such as locations can be tracked very precisely in the simulated environment, and they can be optionally furnished to a recorded demonstration to help learning. The simulated robot can be used to validate imitation learning ideas in an inexpensive way, without damaging a physical robot or even without owning a physical robot. Moreover, using the simulated environment enables demonstrating tasks in extreme scales that are otherwise too big or too small for humans to physically demonstrate (e.g., nanorobotics).

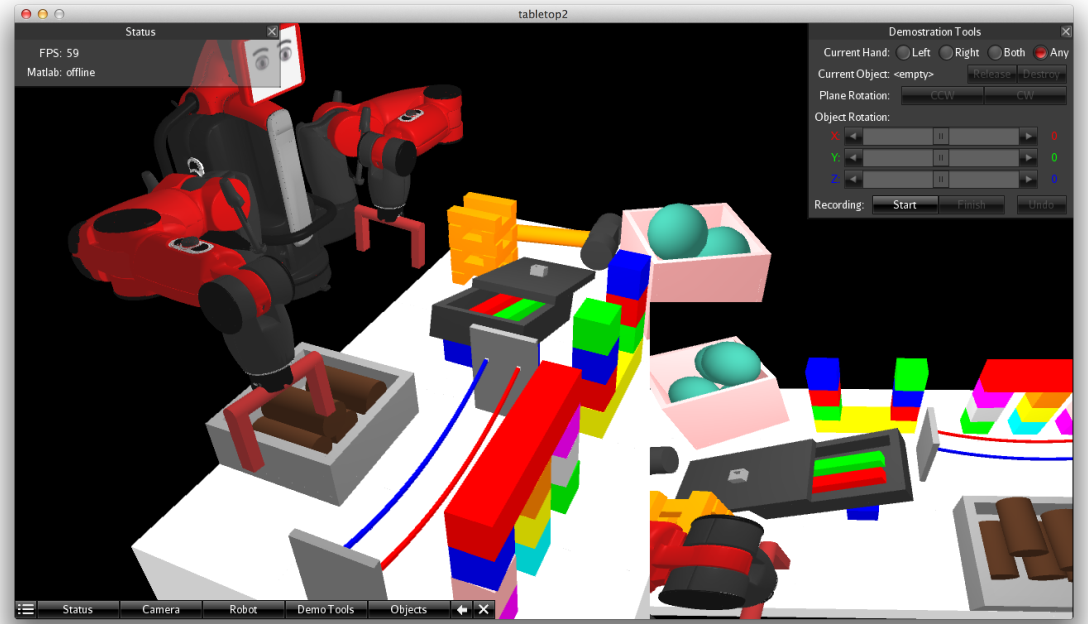


Figure 1: An example view of SMILE.

Figure 2 shows the three interfaces with which a user can interact with SMILE. First, the *demonstration interface* is provided for a human demonstrator to manipulate the objects through intuitive GUI controls using mouse inputs (Figure 2(a)). A demonstration containing multiple object manipulations can be recorded as a "video" (a sequence of images) with optional text description for training the robot. A demonstration 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 the robot learner can bypass difficult human trajectory processing. From the robot's perspective, the objects in the environment move on their own.

Second, the *robot interface* allows custom Matlab scripts to control the simulated robot in SMILE (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 (e.g., moving a robot arm to push an object away).

Third, the *object interface* initializes objects in SMILE using XML (Figure 2(c)). This interface accepts XML files specifying what objects (e.g., shape, color, etc.) to create in the simulated environment as well as their initial states (e.g., locations, orientations, etc.). This interface provides a way to generate a predefined set of objects that forms a task scenario for the demonstrator or

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

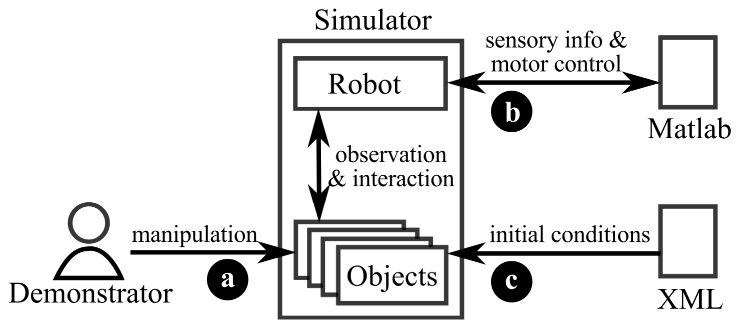


Figure 2: The three primary interfaces of SMILE: (a) the demonstration interface, (b) the robot interface, and (c) the object interface.

the robot to work on. An advantage for conducting controlled experiments is that, by loading the same XML file, this interface guarantees that objects' initial states are fixed exactly across different experiment groups/trials. This would be more difficult to achieve in the physical environment.

The three interfaces described above can be used together or separately for different purposes. A few possibilities are described as follows, although they are not intended to be exhaustive. First, without using the robot interface, one can use SMILE as a standalone demonstration recorder. The recorded demonstration can then be used to train either the simulated robot in SMILE or a physical robot. Second, one can use SMILE as a standalone robot simulator (i.e., leaving out demonstration interface). Controlling the simulated robot is relatively easy and without complicated setup compared with existing solutions such as ROS/Gazebo, and therefore it is well-suited for fast-prototyping robot controllers or for educational purposes. Third, one can use SMILE to study the scenario where a robot and a human are to cooperate to complete a task (human-robot teaming), since both the robot and the user can manipulate objects in SMILE. Finally, SMILE can support active learning where a robot queries user demonstration using the object interface. For example, when a robot (physical or simulated) does not know how to proceed during a task performance, it can reconstruct the state of the environment in SMILE using an XML structure and ask for human demonstration.

2 Getting started

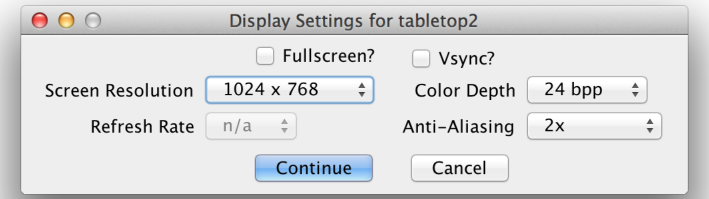
SMILE can be run on OS X, Windows, and Linux. Systems with dedicated graphics chips are preferred but not required. This section explains basic operations of SMILE.

2.1 Prerequisites

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

➢ **Java Runtime Environment (JRE)** is required to run SMILE. 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 SMILE, but is not required to run SMILE itself. Versions R2007b and greater are supported.


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

2.2 Files

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

- `SMILE.jar` – main program.
- `lib/` – directory containing libraries (`*.jar`) required for running SMILE. 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 SMILE. See Section 5.2.
- `tablesetup/` – directory where XML files for initializing objects in the environment should be stored.
- `tablesetup/schema/deep.xsd` and `tablesetup/schema/deep.xsd` – schema files required to parse XML documents. In most cases, no changes to these two files are necessary.

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

2.3 Starting and stopping

To start SMILE, run `SMILE.jar` by double clicking the file. Alternatively, use a command line interface and type: `java -jar SMILE.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.

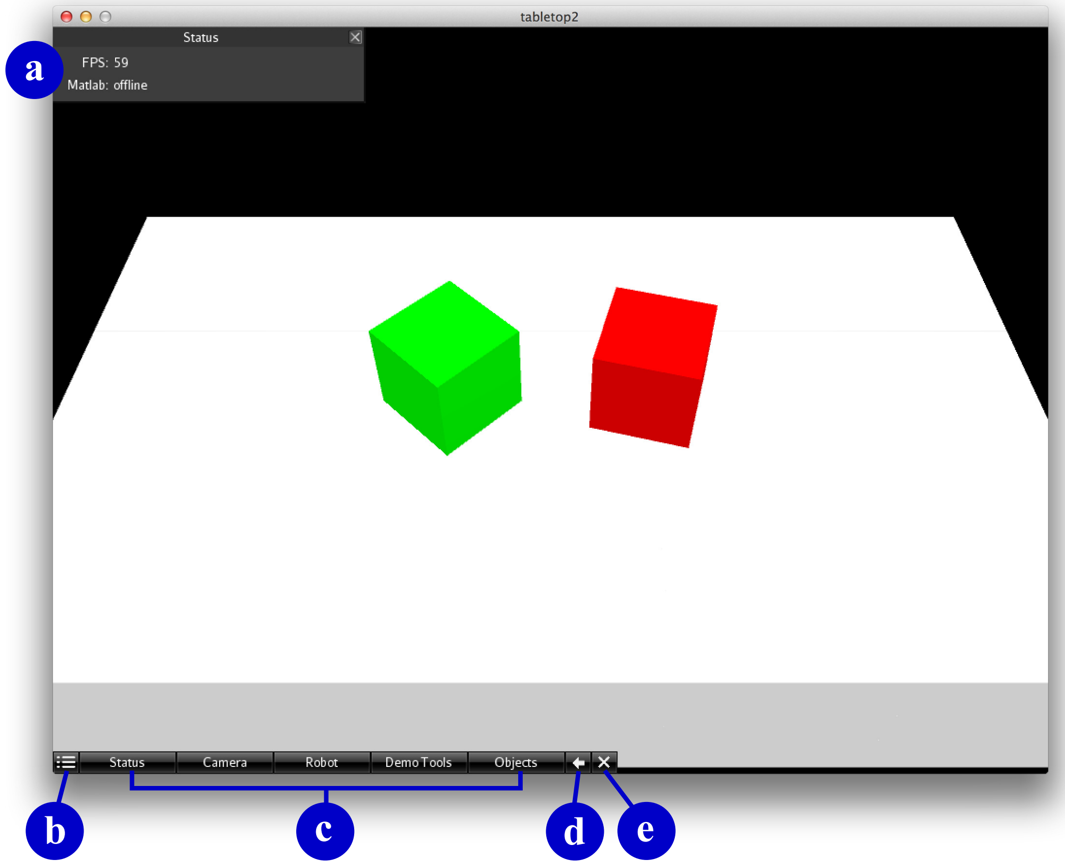


Figure 4: The main screen of SMILE gives a modifiable view of the simulated 3D world and overlaid GUI components. The GUI components include (a) a status window, (b) a button for showing all windows, (c) buttons for showing individual windows, (d) a button for aligning all visible windows, and (e) a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

➢ **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 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 SMILE to change any display settings.

To quit SMILE, press Esc anytime in the main window.

2.4 The main screen

Figure 4 shows the 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

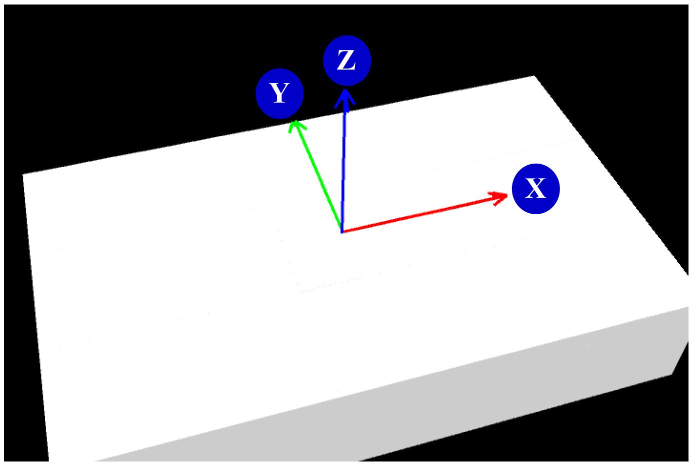


Figure 5: The coordinate system. The origin is located at the center of the tabletop. The XY plane is on the tabletop surface. The Z axis is perpendicular to the tabletop.

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 windows and place them at the left border of the main view, and (e) hide all windows.

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 SMILE is running, usually a large amount of CPU resources is consumed. Pausing SMILE helps reduce the amount taken by SMILE, especially when the main window loses focus. This is useful when the user needs to run another CPU-intensive job while retaining the state of SMILE.

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.

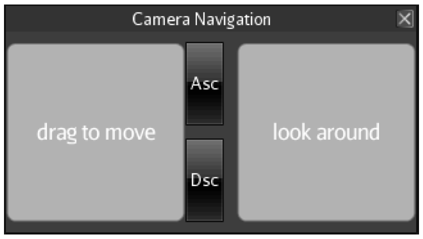


Figure 6: The Camera window that allows the user to navigate in the 3D simulated world.

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, a 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 captured by the robot head camera), as well as text descriptions. The recorded demonstration can be used for imitation learning. In the situation where the demonstrator makes an unwanted move, SMILE 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 using grasp, move, rotate, release actions, etc. A typical flow chart for one hand is shown in Figure 7. A manipulation starts at a free hand grasping an object, triggering a control, or pointing to an object (or a part of an object). When an object is grasped, it is temporarily unaffected by gravity, i.e., it is held by an invisible hand. The object then undergoes a series of movements and/or rotations. 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 gravity after it is released. Alternatively, the object can be destroyed (removed from the environment). 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, in which case a handoff is achieved). 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 (or simply press B or N).

4.1 Recording a demonstration

Using the Start and Finish buttons in the Demonstration Tools window (Figure 8(f)) to record a demonstration. Pressing the Start button will start saving demonstration actions to a subdirectory named `demo/`. The recorded demonstration is segmented in a way that every grasp and release action triggers creation of a new segment. Pressing the Finish button will stop recording, after which the user may want to rename the `demo/` subdirectory since pressing Start again will overwrite

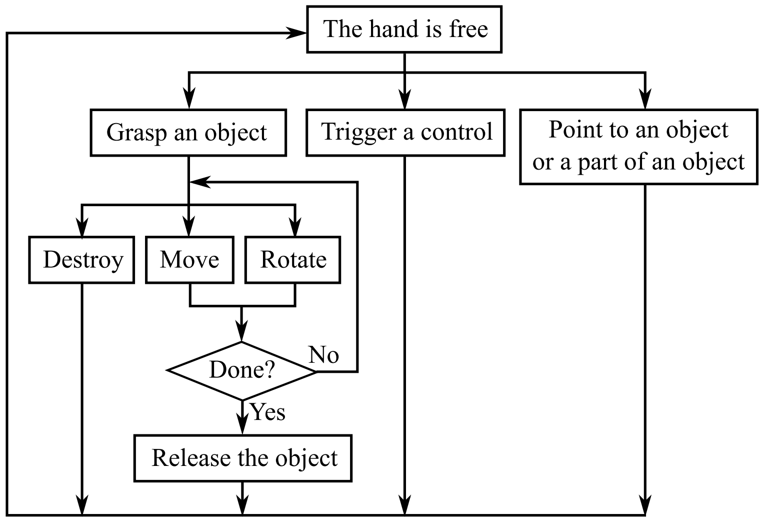


Figure 7: A typical demonstration flow. The demonstrator can grasp an object, move/rotate it until the object is at a desired location with a desired orientation, and release it. Repeat this process with different objects to complete the task being demonstrated.

the content of `demo/`. Anytime during a recording, the demonstrator can press the Undo button to discard the current 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 from the robot head camera, 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 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 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 describes what happens in the environment. The text file is in CSV format, where each line represents an event. The first value of each line indicates the frame number in which the event occurs. The second value indicates the type of the event, as follows:

➢ `create`. This event is generated whenever a new object is created in the environment. In this case, the third value of the line indicates the ID of the object being created. Object IDs can be specified using the object interface (via XML, see Section 6.1). User-specified IDs may be automatically appended a '#' character followed by a number to ensure their uniqueness. From the fourth value on is a list of key-value pairs indicating the properties of the object. For example, to indicate that the shape of the object is a cylinder, the fourth value can be a string `shape`, and the fifth value can be a string `cylinder`. Other properties of the object may appear after the shape property. The number of object properties is not limited. Object properties can be specified using the object interface (see Section 6.1 for XML element `<description/>`).

➢ `delete`. This event is generated whenever an object is removed from the environment. In this case, the third value indicates the ID of the object removed.

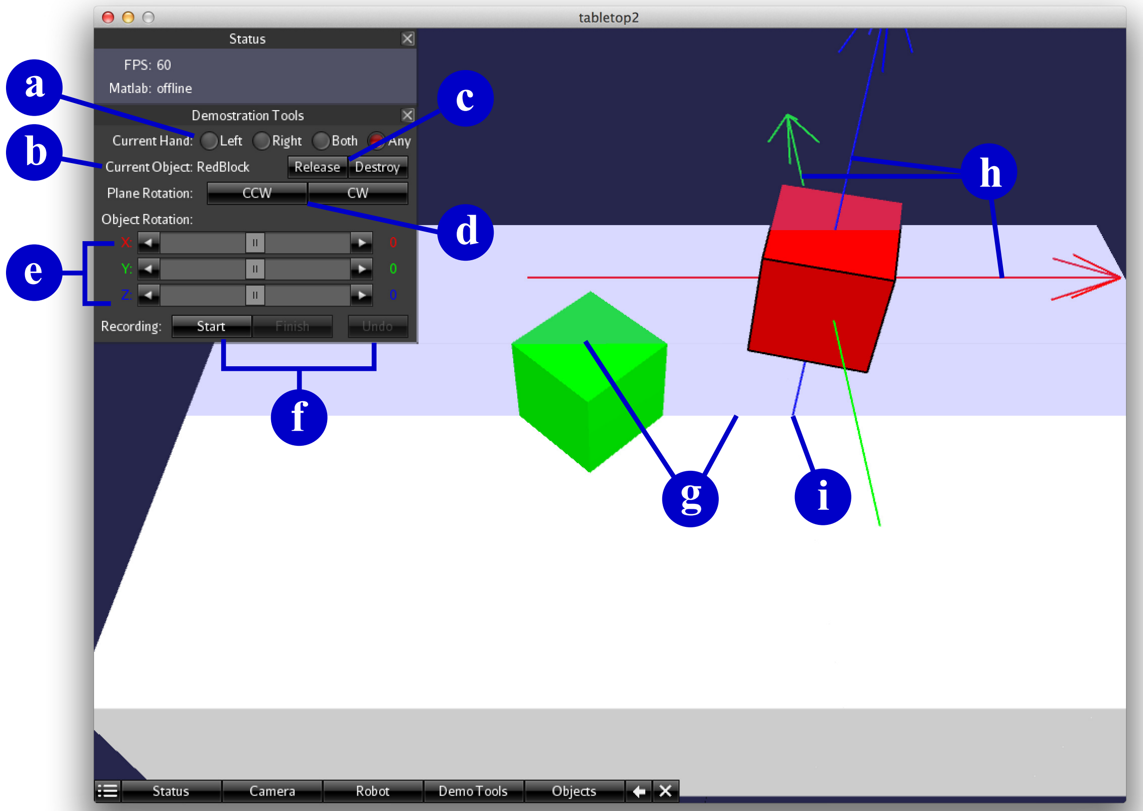


Figure 8: The demonstration interface. The red block (right) is currently grasped by the demonstrator and raised above the tabletop. (a) The radio button controlling which hand is currently in use. (b) The ID of the grasped object. (c) The buttons for releasing and removing a grasped object. (d) The 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 (virtual shadow).

➢ `move`. This event indicates the change in location or orientation of an object. In this case, 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. In this case, 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. In this case, the third value contains the name of the hand released.

➢ `destroy`. This event indicates that the demonstrator destroys a grasped object. In this case, the third value contains the name of the hand used to destroy the object. A corresponding `delete` event will also be generated.

➢ `initializeControl`. This event indicates that a control (e.g., a toggle switch) is initialized, often immediately following the `create` events. In this case, the third value indicates the ID of the control. Like object IDs, control IDs can be specified using the object interface and are guaranteed to be unique. The fourth value indicates the type of the control, such as `toggleSwitch` and `indicatorLights` (see Section 6.4). The fifth value indicates the ID of the object that this control is a part of. It is possible that a control is itself an object, in which case this value would be the same as the third value. The sixth value indicates the initial state of the control, represented as an integer or a string. For example, a toggle switch can be in state `0` or `1` depending on which side of the switch is pressed down. A string representation of control state can also be specified using the object interface (see Section 6.4 for element `<state/>` under `<indicatorLights/>`).

➢ `changeControlState`. This event indicates the change of a control's state. In this case, the third value indicates the ID of the control. The fourth value is a string or integer representing the state (see `initializeControl` event).

➢ `trigger`. This event indicates that the demonstrator triggers the state change of a control, such as toggling a switch control. In this case, the third value indicates the name of the hand being used. The fourth value indicates the control ID.

➢ `pointTo`. This event indicates that the demonstrator points to an object or a part of an object. In this case, the third value indicates the name of the hand being used. The fourth value indicates the ID of the object or the part of an object being pointed to. Like object IDs, the IDs of object parts that can be pointed to are guaranteed to be unique. The fifth value indicates the ID of the object containing the part being pointed to. If a whole object is pointed to, the fourth and fifth values will be identical. By default, all objects can be pointed to. To enable pointing to a part of an object, this part has to be explicitly marked as "pointable" using the object interface (see Section 6.2 for XML attribute `pointable`).

4.2 Grasping and releasing

To grasp an object, first select a hand using the radio buttons in the Demonstration Tools window (Figure 8(a)), and then simply click on the object. 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 Shift+/ rotate the plane in one direction or the other; holding Ctrl instantly rotates the plane 90 degrees.)

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).

4.4 Other actions

To virtually point at an object or a specific part of an object, right-click (secondary click) on it. This allows the demonstrator to provide a cue to direct the attention of the robot learner or to explain a demonstrated action in a non-verbal way. For example, the demonstrator can point to an indicator light before flipping a switch to turn it off. This signifies that the purpose of flipping the switch is to turn that light off. Without pointing to the indicator light, the purpose of flipping the switch is somewhat ambiguous. The pointing action can also be used to communicate where to hold an object, such as pointing to the handle of a drawer before pulling it open. Since pointing actions do not cause object states to change, they are not observable in the recorded visual images. However, `pointTo` events will be generated in the text description of the demonstration. All objects can be pointed to by default. Parts of an object can be pointed to if they are explicitly declared so via the object interface (see Section 6.2 for XML attribute `pointable`).

Right-clicking on a control object causes the control to be manually "triggered", meaning that the state of the control is changed. For example, one can right-click on a toggle switch to flip it to the other position. A triggered control may in turn trigger other controls in the environment,

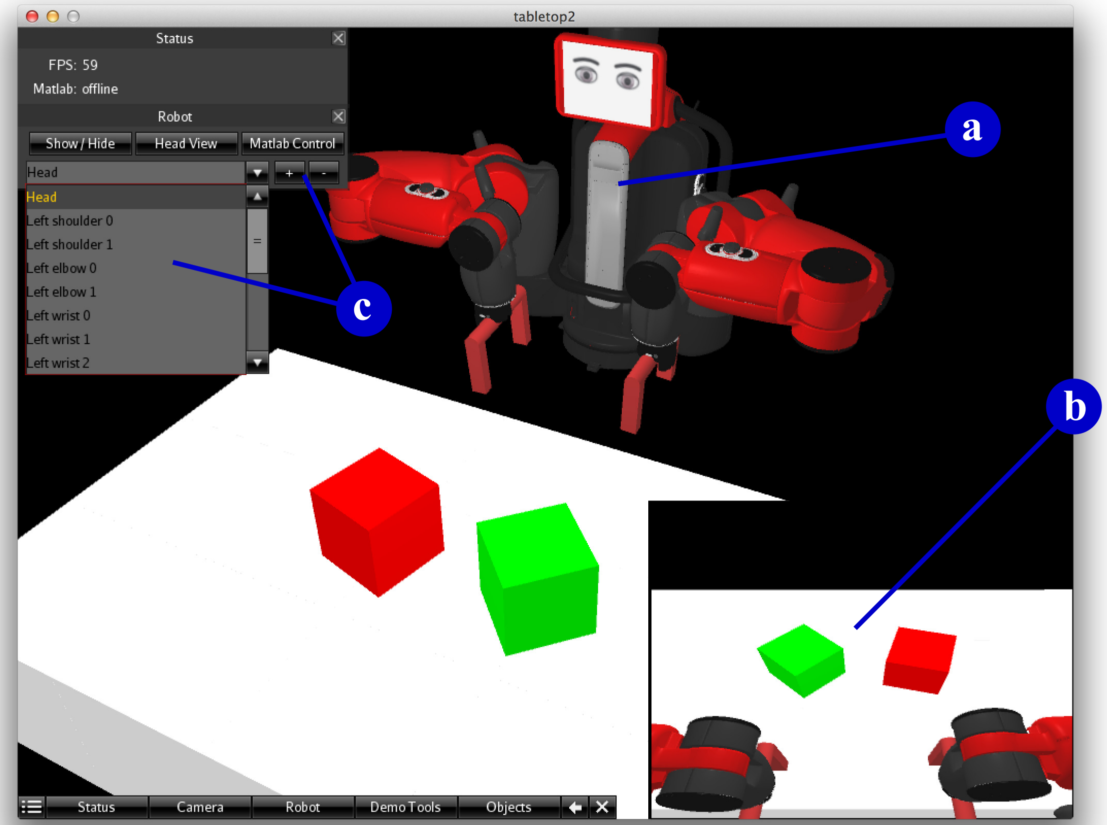


Figure 9: The robot in the simulated world. (a) The 3D model of the robot. (b) The robot's view of the simulated world. The images are captured by its head-mounted virtual camera. (c) GUI components for manually controlling the robot.

depending on how their relationship is defined using the object interface (see Section 6.4 for XML element `<downstream/>`). Unlike the pointing to action above, triggering a control is usually visible in recorded visual images, such as the change of switch toggle positions, the change of indicator light colors, etc. For each trigger action, a `trigger` event will be generated in the demonstration text descriptions. Additionally, one or more `changeControlState` events will be generated for those control whose states are directly or indirectly affected by this trigger action.

Finally, a grasped object can be destroyed by clicking the destroy button in the Demonstration Tools window. This will remove the object from the environment.

5 Robot interface

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

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	$-\pi/2$ – $\pi/2$

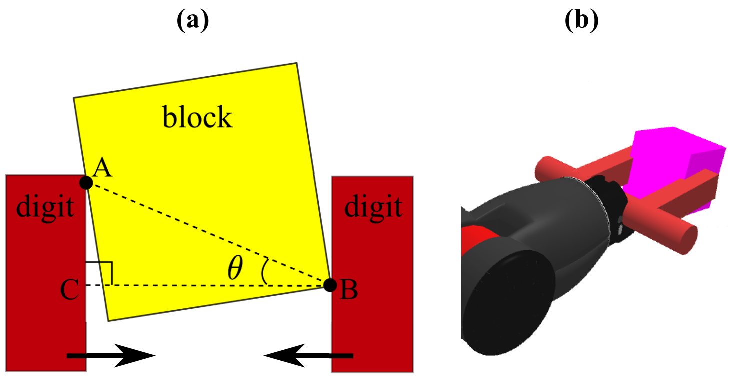


Figure 10: (a) The top view of a gripper attempting to grasp an object. A and B are the contact points between the object and the two digits of the gripper. $\overline{BC}$ is a line perpendicular to the inner surfaces of the gripper. If θ is sufficiently small, the block is considered being held by the gripper. (b) The 3D view of a gripper holding an object.

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: $\boxed{\text{R}}$ to show/hide the robot; $\boxed{1}$ to toggle the robot's view; $\boxed{2}$ 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: S0, S1, E0, E1, W0, W1, and W2[‡]. 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.

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

[‡]For more information, see `http://sdk.rethinkrobotics.com/wiki/Joint_Position_Example`
[§]`https://github.com/RethinkRobotics/`

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 10(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 Shift 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

SMILE 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 SMILE 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 SMILE before any calls to the callback script. The callback script is called repeatedly afterwards whenever SMILE is updating the main screen. If SMILE is running at 60 FPS, the callback script is also called 60 times per second. At each call to the callback script, SMILE supplies the callback script with the robot's sensory information (e.g., visual images and current joint angles). SMILE then retrieves 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 that SMILE 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. SMILE 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 SMILE. Clicking the Matlab Control button again will stop invoking the user's scripts. Whether the user's scripts are currently being called by SMILE is indicated in the Status window. (Keyboard shortcut: M to toggle the invocations of the user's Matlab scripts.)

SMILE communicates with the user's callback scripts through three global Matlab `struct` variables: `sensor`, `motor`, and `aux`. `sensor` is set by SMILE to pass sensory information to the user's scripts, `motor` can be set by the user's scripts to pass motor commands to SMILE, and `aux` is used to exchange auxiliary information between SMILE 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 SMILE (where `SMILE.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.
- ➢ `initJointAngles` is an optional 2-by-7 matrix to be set by the initialization scripts. It specifies the initial joint angles of the robot. The matrix indexing is identical `minJointAngles`.

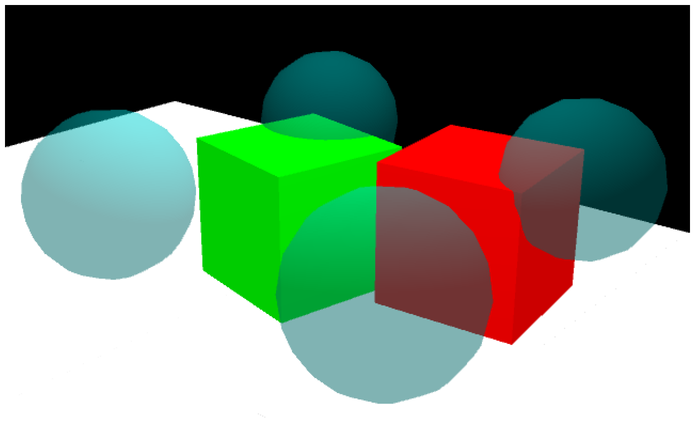


Figure 11: Visual markers are shown as transparent blue spheres. They do not interact with objects.

- ➢ `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 SMILE will draw a "marker" in the simulated world. Markers allow one to visually observe the exact locations of specified 3D coordinates. For example, in an arm control task, a marker can be placed at the target location of an arm reaching movement, so that one can visually observe how far apart the robot grippers are from the target. A marker is visualized as a transparent blue sphere (Figure 11). 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. SMILE stops invoking the user's scripts from this point on. All joint and gripper velocities are set to 0. All markers drawn are cleared.

Files `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 SMILE. 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 interface

This section describes how objects on the tabletop can be generated. In Figure 12, the Objects window contains two methods of generating objects: manually generating preset objects (the bottom row in the window) and loading an XML file (the top row in the window).

In the first method, several preset objects are available for generation in the bottom drop down box in the Objects window. Select one of the options and click the Execute button to generate the specified objects. Objects are generated in random colors and at random locations on the

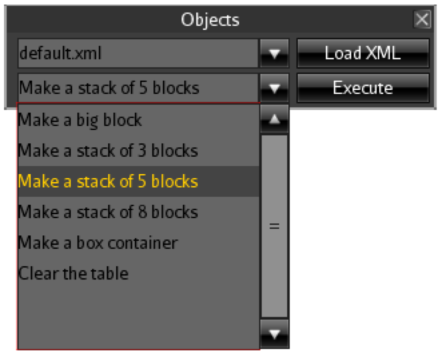


Figure 12: The Objects window for generating tabletop objects.

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 in the environment except those that are currently being grasped¶.)

In the second method, objects can be generated by loading custom XML files. The remainder of this section describes the XML format supported by SMILE. An XML file specifies a set of objects, along with their properties such as sizes, colors, locations, etc. Basic objects such as blocks and cylinders can be specified using simple XML elements such as `<block/>` and `<cylinder/>`. Custom 3D models can be specified using separate STL files, which can be generated by most CAD tools. An XML file can also define composite objects that are composed of multiple basic objects in a hierarchical fashion. Controls such as toggle switches and indicator lights, as well as their relationship, can also be specified using XML. The object interface also supports advanced features such as object templates and variable substitution, for improved flexibility and reusability in writing XML files.

SMILE always loads `tablesetup/default.xml` when it is launched. The user can load other 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.

A valid XML file has to have a `<tabletop/>` root element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
    ...
</tabletop>
```

where `xspan` and `yspan` specify the size of the table, and the ellipses indicate where a list of other XML elements should be written. XML elements directly under `<tabletop/>` either creates an object, imports another XML file, or defines a template of objects. The following subsections describe these XML elements.

6.1 Simple objects

Simple objects supported by SMILE include `<block/>`, `<cylinder/>`, `<sphere/>`, `<box/>`, and `<custom/>` (Figures 13, 14). All these elements have the following common attributes:

¶Hold down Shift at the same time to clear the table with some robot rage.

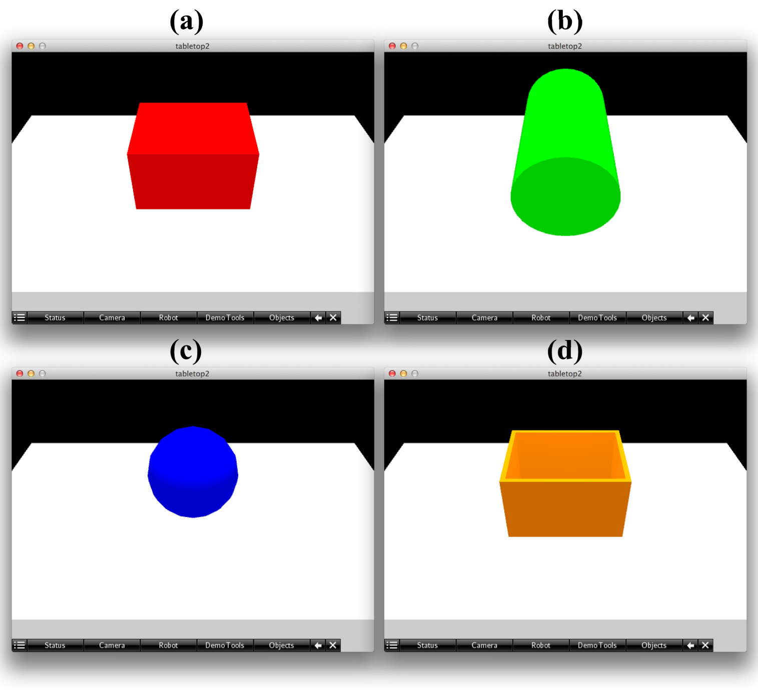


Figure 13: Simple objects that are located near the center of the tabletop in their default orientation: (a) a red block, (b) a green cylinder, (c) a blue sphere, and (d) an orange box.

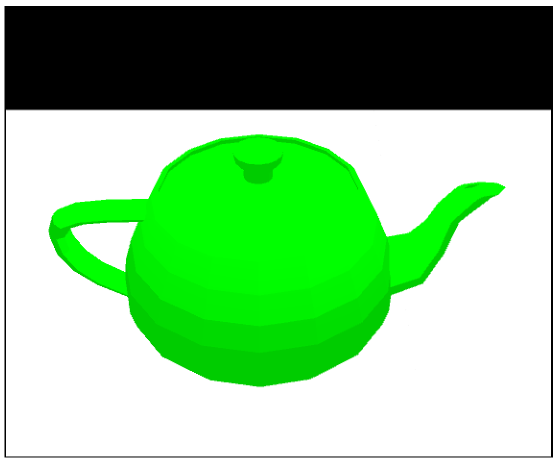


Figure 14: A custom geometry defined in a separate STL file and loaded into SMILE using a `<custom/>` element.

- ➢ `id` specifies the unique identifier of the object. This is for uniquely identifying the object. If the `id` value is not unique, SMILE will append a '#' character followed by an integer to enforce the uniqueness. **Default value:** an arbitrary 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)`. This center serves as the center of mass of the object. Numbers `x`, `y`, and `z` specifies the coordinates of a 3D location (see Figure 5 for how each axis is defined). Positioning an object should prevent the object from overlapping with other objects or with the table, or unexpected physics effects may occur. For example, since the center of a $1 \times 1 \times 1$ `<block/>` object lies at the centroid, placing it at `(0,0,0)` causes the bottom half of the block to intersect the table, which may result in the block 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 format `(x,y,z)` is the same as the `location` attribute, 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. The order of applying these angles are: X, Y, and then Z. 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`.

Currently, SMILE supports the following simple objects. While each example below creates only one object, an XML file can contain any number of objects.

- ➢ `<block/>` creates a solid cuboid, whose dimensions are specified by 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 13(a)).

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <block id="b1" location="(0,0,1.5)" rotation="(0,0,0)" color="red" mass="0.5"
    xspan="5" yspan="4" zspan="3" />
</tabletop>
```

  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. All attributes except `location` are optional. Their default values will be used if they are omitted.

- ➢ `<cylinder/>` creates a solid circular cylinder. The default orientation is that the axis of the cylinder is parallel to the Y axis. The center of the cylinder lies in the center of its axis. Use attribute `yspan` to specify the length of the cylinder and `radius` to specify the radius. **Default values:** `yspan="1" radius="0.5"`.

  Example: a green cylinder (Figure 13(b)).

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <cylinder id="c1" location="(0,0,2)" color="green" yspan="7" radius="2"/>
</tabletop>
```

➢ `<sphere/>` creates a solid ball. The radius can be specified by attribute `radius`. **Default value:** `radius="1"`.

Example: a blue sphere (Figure 13(c)).

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <sphere id="s1" location="(0,0,2)" color="blue" mass="0.5" radius="2"/>
</tabletop>
```

➢ `<box/>` creates a cuboid container with open top and hollow center. Attributes `xspan`, `yspan`, and `zspan` are used to specify the exterior lengths of the box along the three axes. Additionally, attribute `thickness` specifies the the thickness of the walls. The center of the box lies in the center of the whole cuboid (i.e., not at the center of the hollow space). **Default values:** `xspan="1" yspan="1" zspan="1" thickness="0.05"`.

Example: an orange box container (Figure 13(d)).

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <box id="x1" location="(0,0,1.5)" color="orange" xspan="5" yspan="4" zspan="3"
    thickness="0.2" />
</tabletop>
```

➢ `<custom/>` creates a custom object by referring to an external STL file. An STL geometry can be crafted using most CAD tools. The center of a custom object is implicitly defined in the STL file. In the case where the STL file contains more than one geometry, only the first geometry will be loaded. Attribute `file` (**required**) specifies the path of the STL file to be loaded, and attribute `scale` specifies the scaling factor to be applied to the loaded STL geometry (**default value:** `scale="1"`). The latter is for compensating the differences of distance units used in CAD tools and in SMILE.

Example: a green teapot (Figure 14).

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <custom id="teapot" location="(0,0,0)" rotation="(-90,0,0)" color="green"
    scale="0.1" file="tablesetup/stl/Teapot.stl"/>
</tabletop>
```

Note that the STL file path is relative to the root directory of SMILE.

All simple object elements can have zero or more `<description/>` elements as child elements. Each `<description/>` element has attributes `name` and `value` (**both required**). This element does not affect how objects are created/initialized by SMILE. Instead, it is intended to allow users to annotate/describe object properties, and its name and value will appear in the `create` event of a recorded demonstration (see Section 4.1). Note that SMILE always implicitly inserts to all object a description named `shape`, whose value is taken from the name of the simple object, such as `block` and `sphere`. In the case of a `<custom/>` element, the value of `shape` is taken from the filename loaded. For example, writing:

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <custom location="(0,0,0)" file="tablesetup/stl/Teapot.stl">
    <description name="style" value="modern"/>
  </custom>
</tabletop>
```

is equivalent to writing:

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <custom location="(0,0,0)" file="tablesetup/stl/Teapot.stl">
    <description name="shape" value="Teapot"/>
    <description name="style" value="modern"/>
  </custom>
</tabletop>
```

Users can override the SMILE-generated value of `shape` by explicitly defining a `<description/>` element of the same name.

6.2 Composite objects

Multiple simple objects can be hierarchically combined to form a single rigid composite object, using `<composite/>` elements. A `<composite/>` element can have one or more child elements that are either simple object elements or `<composite/>` elements. This forms a logical tree structure. Attributes `id`, `location` (**required**), and `rotation` can appear in a `<composite/>` element. Additionally, the top-level `<composite/>` element can have a `mass` attribute. Each descendent element of a `<composite/>` element can have an optional boolean attribute `pointable` (**default value:** `false`), which indicates whether a certain part of a composite object can be pointed to by the demonstrator.

In a `<composite/>` tree structure, while the `location` and `rotation` of the root element specifies the location and orientation of the entire object, those of other descendent elements are not absolute but are relative to their parent element. For example, the following XML does not place the block at location `(0,0,1)` but at `(2,2,1)` instead.

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <composite location="(2,2,0)" rotation="(0,0,0)" mass="10">
    <block xspan="2" yspan="2" zspan="2" location="(0,0,1)"/>
  </composite>
</tabletop>
```

This is because the block location is moved upwards relative to the center of the enclosing composite object, so that the center of the composite object is located at the bottom of the block. The center of the composite object is then moved to `(2,2,0)`, which results in the block center being moved to `(2,2,1)`. Note that the center of a composite object is also the object's center of mass. In this example, it is equivalent to having a block whose center of mass is at the center of its bottom surface. While the position and the location of the root `<composite>` object should not overlap with other objects or with the table, it is fine for its constituent shapes to overlap with one another. This is because all shapes under the root element are considered as parts of a whole object, and therefore they do not collide with one another.

The following example illustrates how multiple simple objects can be used to compose a hammer-like object (Figure 15). Notice how the handle is offset further from the center of the object than the

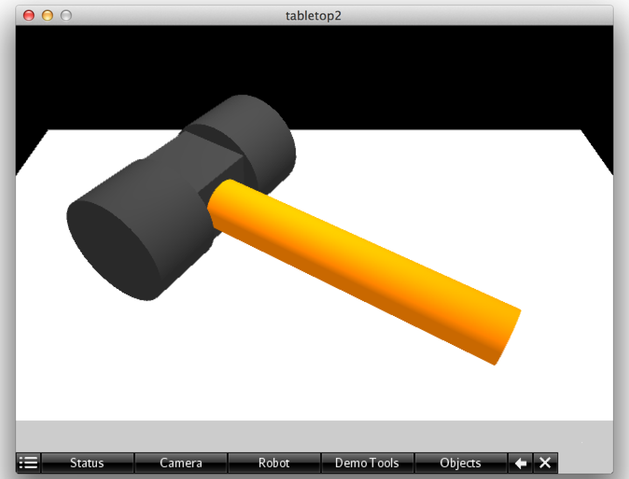


Figure 15: A composite object example. The hammer-like shape is composed of an orange handle and a dark gray head. The head is in turn composed of a cube and two cylinders.

head piece, such that the center of mass is near the center of the head piece. This object is treated as a single object, so that if one moves the handle, the other parts of the object move automatically with it.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <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>
</tabletop>
```

Similarly to simple object elements, each `<composite/>` element can contain zero or more `<description/>` element for annotating user-defined object properties, which will appear in the `create` event of a demonstration.

6.3 Physics constraints

It is possible to specify physics constraints between two objects, such that the two objects appear to be connected in a certain way. Currently, only slider joints are supported by SMILE. A slider joint creates a physics constraint analogous to the relation between a desk and a drawer. The drawer can slide away from the desk in one direction (1-DOF), and there exist a minimum and a maximum offset values between the two objects.

To create a slider joint between two object, use a `<sliderJoint/>` element. A `<sliderJoint/>` element should contain exactly two child elements that are each a whole object. The first child element will be referred to as `obj1` and the second `obj2` hereafter for convenience. A slider joint has two reference points. The first reference point is located at the origin `(0,0,0)`, where `obj1` is attached. The `location` and `rotation` of `obj1` are relative to this first reference point. The second reference point is located at `(-init,0,0)`, where `init` is an attribute specifying the initial offset between the two objects. The value of `init` must be in [`min`,`max`], where `min` and `max` are attributes specifying the minimum and maximum offset distances between the two object. Object `obj2` is attached to this second reference point. The `location` and `rotation` of `obj2` are relative to this reference point. So far, a sliding "rail" between `obj1` and `obj2` is created along the X axis. A `<sliderJoint/>` has its own `location` and `rotation` attributes too, which then transform the "rail" to desired location and orientation. **Default values:** `min=0, max=1, init=0`.

Physics properties of a slider joint can be specified using attributes `damping`, `restitution`, and `softness`. They each takes on value between 0 and 1 (inclusive). Damping specifies the drag of sliding movements (**Default:** `damping="0"`). Restitution specifies the bounciness when the limits of the allowed moving range are reached (**Default:** `restitution="0.7"`). Softness specifies the proportion of the allowed moving range that the objects can move freely. At the two ends of moving range that are outside of this proportion, the drag gradually increases to 1 (**Default:** `softness="1"`). Additionally, boolean attribute `collision` specifies whether `obj1` and `obj2` can collide with each other (**Default:** `collision="false"`).

The example below illustrates how a box with a sliding lid can be made (Figure 16):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <sliderJoint location="(0,-1,3)" min="0" max="5.8" init="4">
    <box id="box" xspan="6" yspan="4" zspan="3" thickness="0.2" location="(0,0,-1.5)"
      color="#00ffff" mass="10"/>
    <block id="lid" xspan="6" yspan="4" zspan="0.2" location="(0,0,0.1)" color="red"
      mass="1"/>
  </sliderJoint>
</tabletop>
```

Note that `obj1` (the box) attaches its top to the first reference point of the slider joint, and `obj2` (the lid) attaches its bottom to the second reference point of the slider joint. Therefore, to avoid intersecting the table, the Z coordinate of the slider joint must be at least the height of the box, which is 3.

In addition to slider joints, SMILE also supports the creation of a special object: a chain (Figure 17(a)). A chain is not a single rigid object, but a set of linearly connected links that mimics a string. The connections between links are special physics constraints that are different from slider joints described above. Currently, this kind of physics constraint is not available directly through XML yet. 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 Figure 17(b). 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`.

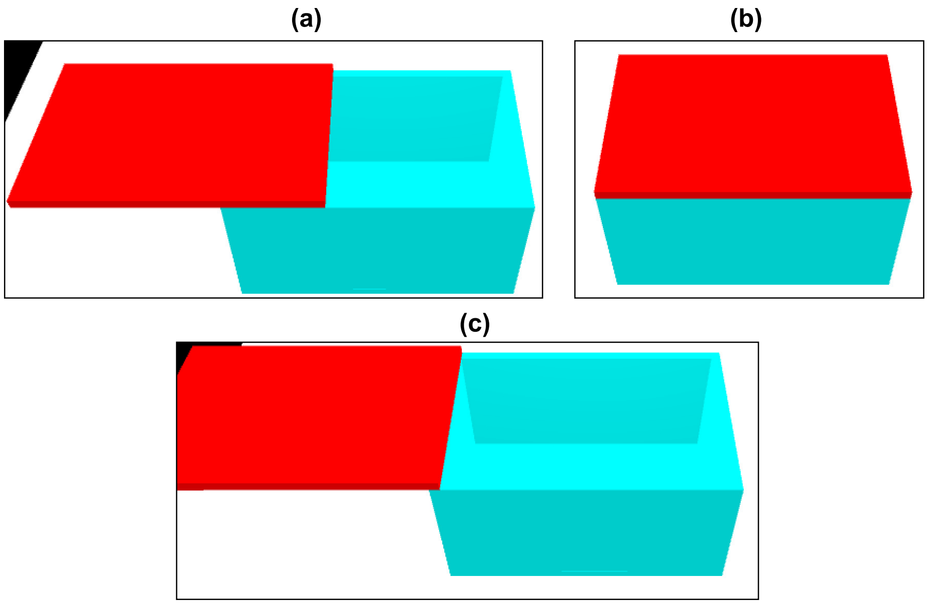


Figure 16: A slider joint connecting a box and a block (lid). (a) The lid is at its initial position. (b) The lid is slid to the minimum offset. (c) The lid is slid to the maximum offset.

- ➢ `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`.

Example: (Figure 17(a))

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

6.4 Controls

SMILE supports two types of controls, toggle switches and indicator lights. Each control can define a number of discrete states. Each state is represented by an integer starting from 0. A control can also define a set of downstream controls, so that whenever the current state of the control changes to a new value N, it triggers its downstream controls to also transition to state N. If state N is not a valid state for a downstream control, the state transition is ignored.

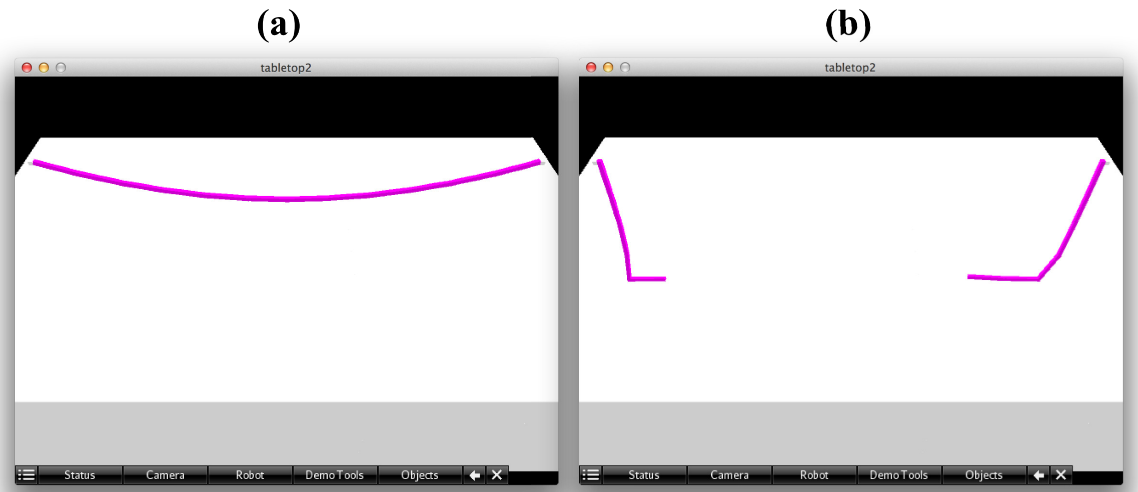


Figure 17: (a) A chain object with its both ends fixed above the tabletop. (b) After one of its links is removed, the remainder of the chain falls down to the tabletop.

A toggle switch can be created using a `<toggleSwitch/>` element. Whenever the demonstrator presses it using a right-click, its current state is increased by one. Once the current state value exceeds a predefined maximum value, it is reset to 0. The geometry of a toggle switch is illustrated in Figure 18. It is composed of two blocks that partially intersect each other with an angle. The attributes of `<toggleSwitch/>` are as follows:

- `id` specifies a unique ID of the control. It is important to ensure that the ID is unique if this control is to be triggered by other controls. **This attribute is required.**
- `location` and `rotation` control the position and orientation of the switch. **Default value:** `rotation="(0,0,0)"`; `location` **is required**.
- `xspan`, `yspan`, `zspan`, `angle` (in degrees) and `color` specify the appearances of the toggle switch, as illustrated in Figure 18. **Default values:** `xspan="0.6"`, `yspan="0.3"`, `zspan="0.1"`, `angle="6"`, `color="darkgray"`.
- `numStates` specifies the number of valid states that the switch can be in. A valid state is any integer in range [0, `numStates`). **Default value:** `2`.
- `initState` specifies the initial state of the toggle switch. **Default value:** `0`.
- `leftPressed` specifies if the left side of the switch is initially pressed down, as viewed from the angle illustrated in Figure 18. Which side of the switch is pressed down is not directly related to the current state of the switch, except that pressing the switch will cause the state to increment by one (modulo `numStates`). **Default value:** `true`.

Note that while a toggle switch can be in any state in [0, `numStates`), it's visual appearances can only be in one of the two states: left-side down and right-side down. A visual state is not directly related to the state of a toggle switch, only that whenever the visual state changes (i.e., when triggered by the demonstrator), the state is increased by one modulo `numStates`. In a recorded

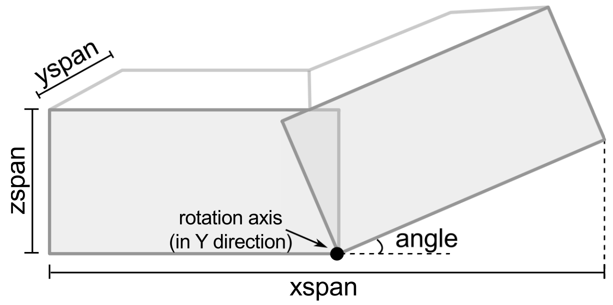


Figure 18: Geometry of a toggle switch.

demonstration, only the visual state is recorded in `initializeControl` and `changeControlState` events, where `0` represents left-side down and `1` represents right-side down (see Section 4.1). Further, a `<toggleSwitch/>` can have zero or more `<downstream/>` elements as children, which has a single attribute `id` referencing another control. Each `<downstream/>` element specifies a downstream control that can be controlled by this switch. Whenever the state of a switch changes to a new value, the states of all downstream controls are changed to this same value (where applicable).

A set of indicator lights can be created using an `<indicatorLights/>` element. The demonstrator cannot directly affect the states of indicator lights through right-clicking as with a toggle switch. Therefore, the state transition of a set of indicator lights can only be triggered by upstream controls. The geometry of a set of indicator lights is illustrated in Figure 19. The indicator lights are organized along the X axis with equal intervals, and each light is assigned an integer light ID starting from 0, from left to right. In each state, each light can either be turned off (made invisible) or displays a certain color, as defined by child elements `<state/>`. An `<indicatorLights/>` element has the following attributes:

➢ `id` specifies a unique ID of the control. It is important to ensure that the ID is unique if this control is to be triggered by other controls. **This attribute is required.**

➢ `location` and `rotation` control the position and orientation of the switch. **Default value:** `rotation="(0,0,0)"`; `location` **is required**.

➢ `numLights` specifies the number of lights that this control has. Each light is assigned an integer ID in [0, `numLights`) in an increasing order from left to right. **Default value:** `2`.

➢ `xspan, lightRadius,` and `lightZspan` specify the appearances of the set of indicator lights, as illustrated in Figure 18. **Default values:** `xspan="0.15", lightRadius="0.03", lightZspan="0.005"`.

➢ `initState` specifies the initial state of this control. The value should be a non-negative integer that is strictly less than the number of valid states of this control. The number of valid states is implicitly defined using child XML elements. **Default value:** `0`.

An `<indicatorLights/>` element can contain (as child elements) zero or more `<downstream/>` elements as in `<toggleSwitch/>`, followed by one or more `<state/>` elements. The number of contained `<state/>` elements determines the number of states that this control can be in. Each `<state/>` can have an optional attribute `descriptionName`, whose value is a string indicating the descriptive name of the state. This name will appear in demonstration text files under

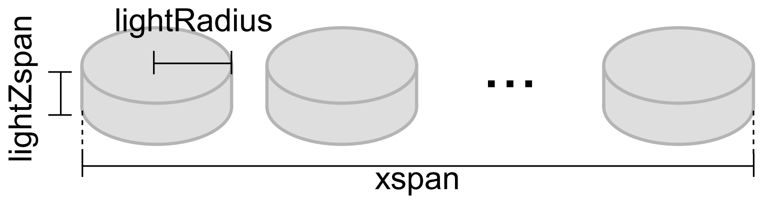


Figure 19: Geometry of a set of indicator lights.

`initializeControl` and `changeControlState` events (see Section 4.1). The first `<state/>` element represents state 0, the second represents state 1, and so on. Each `<state/>` element can contain zero or more `<light/>` elements. A `<light/>` has attributes `id` (**required**), which specifies the integer ID of a certain light, and `color` (**default value:** `red`), which specifies the color of the light. Each light can be in different color in different states. Lights that do not appear in a `<state/>` element is turned off (invisible) in that state.

The example below creates a toggle switch controlling two sets of indicator lights (Figure 20):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <composite location="(-2,0,1)">
    <block xspan="2" yspan="2" zspan="2" location="(0,0,0)"/>
      <toggleSwitch id="S1" xspan="1.7" yspan="1" zspan="0.5" location="(0,0,0.55)"
        angle="20" leftPressed="false" numStates="3">
        <downstream id="L1"/>
      </toggleSwitch>
  </composite>
  <composite location="(2,0,0.5)" rotation="(0,0,90)">
    <block xspan="5" yspan="2" location="(0,0,0)"/>
    <indicatorLights id="L1" xspan="4.5" location="(0,0,0.5)" numLights="3"
      lightRadius="0.7">
      <downstream id="L2"/>
      <state descriptionName="stop">
        <light id="0" color="red"/>
      </state>
      <state descriptionName="caution">
        <light id="1" color="yellow"/>
      </state>
      <state descriptionName="go">
        <light id="2" color="green"/>
      </state>
    </indicatorLights>
  </composite>
  <composite location="(5,0,0.5)" rotation="(90,0,0)">
    <block xspan="1" yspan="1" location="(0,0,0)"/>
    <indicatorLights id="L2" xspan="0.4" location="(0,0,0.5)" numLights="1"
      lightRadius="0.4">
      <state descriptionName="red">
        <light id="0" color="red"/>
      </state>
      <state descriptionName="dark">
        <light id="0" color="black"/>
      </state>
    </indicatorLights>
  </composite>
</tabletop>
```

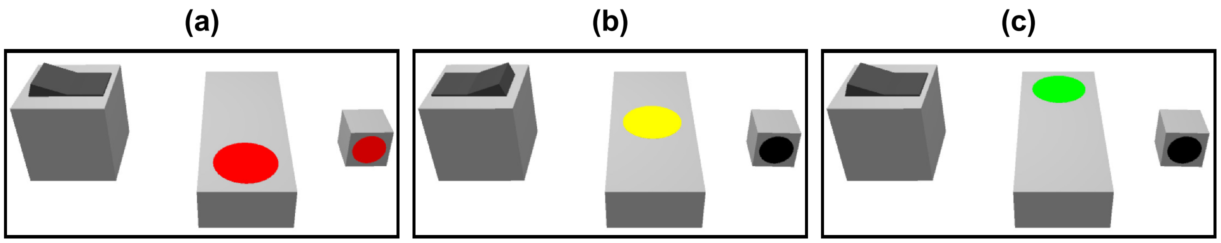


Figure 20: An example containing three controls (from left to right): a switch (S1), a set of indicator lights (L1), and another indicator light control (L2). The numbers of states for S1, L1, and L2 are 3, 3, 2, respectively. S1 has a single downstream control L1, and L1 has a single downstream control L2. (a) Initially, all controls are in state 0. (b) After the demonstrator pressed S1, all controls are in state 1. (c) After the demonstrator pressed S1 again, both S1 and L1 are in state 2, while L2 remains in state 1. If the demonstrator press S1 again, all controls will go to state 0 as in (a), although the left side instead of the right side of S1 will be down.

In this example, a toggle switch (ID=S1) is created within the first composite object. S1 can be in one of three possible states (`numStates="3")` and has a downstream L1. The initial state of S1 is `0` by default. The second composite contains a indicator light control (ID=L1). L1 contains three lights (`numLights="3"`) and can be in one of three states, which are defined by its three child `<state/>` elements. In state `0`, only the first light is turned on, and the color is red. Similarly, in state `1`, only the second yellow light is turned on, while in state `2`, only the third green light is turned on. L1 also has a downstream L2. In the third composite object, another indicator light control is defined (ID=L2). This control has one light and two possible states. When the demonstrator presses S1, the state of S1 becomes `1`. This causes its downstream L1 to go to state `1` as well, which results in a single yellow light being turned on. L1 also passes its state to its downstream L2, resulting in the light in L2 becomes black. Similarly, if the demonstrator presses S1 again, both S1 and L1 will go to state `2`, and the light appearances of L1 will change accordingly. However, since L2 has only two states, it cannot transition to state `2`. Therefore, it will remain in state `1`.

In addition to the two types of controls described above, users can also define custom controls using `<customControl/>` elements. Users can specify the control appearances in each state, as well as state transitions of the custom controls. This grants significant freedom in creating controls. A `<customControl/>` element has the following attributes:

- `id` specifies a unique ID of the control. It is important to ensure that the ID is unique if this control is to be triggered by other controls. **This attribute is required.**
- `location` and `rotation` control the position and orientation of the control. **Default value:** `rotation="(0,0,0)"`; `location` **is required**.
- `initState` specifies the initial state of this control. The value should be a non-negative integer that is strictly less than the number of valid states of this control. The number of valid states is the number of `<state/>` child elements that this `<customControl/>` has. **Default value:** `0`.
- `name` specifies the name of the control. This value is used in `initializeControl` events in recorded demonstrations (see Section 4.1). **Default value:** `customControl`.

A `<customControl/>` element can contain (as child elements) zero or more `<downstream/>` elements, followed by one or more `<state/>` elements. The `<downstream/>` elements list a set of downstream controls such that, when the state of this custom control changes to a new value N, those downstream controls will also transition to this same state N. The number of `<state/>` elements following `<downstream/>` determines the number of states that this control can be in. Each `<state/>` can have an optional attribute `descriptionName`, whose value is a string indicating the descriptive name of the state. This name will appear in recorded demonstration text files under `initializeControl` and `changeControlState` events (see Section 4.1). The first `<state/>` element represents state 0, the second represents state 1, and so on. Each `<state/>` element can contain zero or more simple and composite objects. These objects specify the appearances of the control when the control is in the corresponding state. Each of these simple or composite object elements can have an optional attribute `nextStateWhenTriggered` specifying a numerical state value. This value indicates that, when the corresponding part of the control is triggered by the demonstrator (via right-clicking), the control will go to the indicated state. Figure 21 shows an example of creating three different custom controls. The XML for creating the sphere control on the left-side of the figure is shown below as an example (see `tablesetup/custom-control.xml` for complete XML):

```
  <customControl id="s1" location="(-3,0,0)">
    <downstream id="c1"/>
    <state>
      <sphere radius="0.5" location="(0,0,0.5)" color="red"
        nextStateWhenTriggered="1"/>
    </state>
    <state>
      <sphere radius="0.8" location="(0,0,0.8)" color="orange"
        nextStateWhenTriggered="2"/>
    </state>
    <state>
      <sphere radius="1.1" location="(0,0,1.1)" color="green"
        nextStateWhenTriggered="3"/>
    </state>
    <state>
      <sphere radius="1.4" location="(0,0,1.4)" color="magenta"
        nextStateWhenTriggered="0"/>
    </state>
  </customControl>
```

While in this example, each `<state/>` contains only a `<sphere/>`, it can certainly contain multiple object elements, and each object element can have different `nextStateWhenTriggered` values.

6.5 Importing other XML files

An XML file can import other XML files using `<include/>` elements that are direct children of the root `<tabletop/>` element. An `<include/>` element has a single attribute `file`, which specifies the path of the target XML to be imported. The path is relative to the root directory of SMILE. Upon seeing this element, SMILE substitutes it with the content of the target XML file. If the target XML file contains other `<include/>` elements, they are processed recursively. Each XML file is loaded only once, meaning that SMILE ignores those `<include/>` elements whose target file has already been loaded. The following is an example of importing an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <include file="tablesetup/twoblocks.xml"/>
  <block id="BlueBlock" location="(0,-3,0.5)" color="blue" />
</tabletop>
```

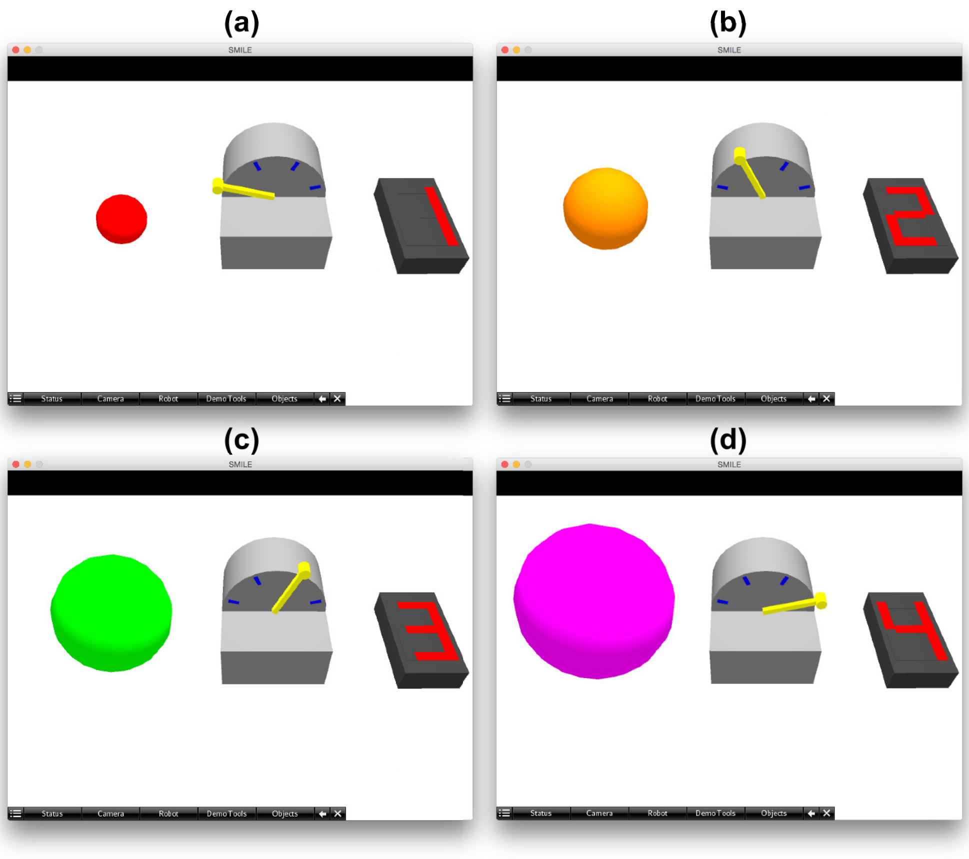


Figure 21: A custom control example. Three custom controls are created on the tabletop. On the left side is a sphere that changes its color and size in different states. A state transition can be triggered by right-clicking on the sphere. In the middle is a dial whose lever can be in one of four positions as indicated by blue ticks. A state transition can be triggered by right-clicking on the blue ticks. On the right side is a seven-segment display. The dial is a downstream control of the sphere, and the display is downstream control of the dial. (a)–(d) shows the controls in states 0–3, respectively, where state transitions are triggered by right-clicking on the sphere.

where the content of `tablesetup/twoblocks.xml` is:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <block id="RedBlock" location="(2,0,0.5)" color="red"/>
  <block id="GreenBlock" location="(-2,0,0.5)" color="green"/>
</tabletop>
```

This is equivalent to the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <block id="RedBlock" location="(2,0,0.5)" color="red"/>
  <block id="GreenBlock" location="(-2,0,0.5)" color="green"/>
  <block id="BlueBlock" location="(0,-3,0.5)" color="blue" />
</tabletop>
```

6.6 Object templates and instances

For better XML code modularity and reusability, SMILE allows XML files to define object templates that can be used to generate object instances. When generating object instances, different user-defined parameter values can be passed to the same template to create objects of different attributes. Internally, SMILE performs variable substitutions and basic arithmetic.

To define an object template, elements `<def/>` can be used directly under the root element `<tabletop/>`. Each `<def/>` has a single attribute `name` specifying the name of the template. The content of the template is specified as child elements of `<def/>`. A copy of all these child elements will replace each `<instance/>` element that refers to this template, after variable substitutions where appropriate. XML elements for simple objects, composite objects, slider joints, controls, etc., may be used as content of templates. The attribute values of these content elements may contain variables, which are surrounded by dollar signs, e.g., `xspan="$width$"`, where `width` is the name of the variable. Variable values are determined by (passed in from) each `<instance/>` element where an object instance is to be created. It is also possible to use multiple variables combined with string literals in the same attribute value, such as `location="($x$,$y$,1)"`, where `x` and `y` are variables. Basic arithmetic is also supported within each dollar-sign surrounded area, assuming each variable value occurring in this area can be successfully converted to a float value in SMILE, e.g., `radius="$width/2-(margin*2+0.3)$"`, where `width` and `margin` are two variables convertible to float values. Although the values of variables are determined by individual object instances, a template can also assign default values to variables in case that some variable values are left unassigned by the instances. To do so, zero or more `<var/>` elements can be added as direct child elements of `<def/>`, preferably before template content. Each `<var/>` element has attributes `name` and `value`, which are the name and default value of a variable. Additionally, a boolean attribute `derived` can also be specified in a `<var/>` element (**default value:** `false`). When assigned `true`, this attribute tells SMILE that this variable's value should be derived from other non-derived variable values. That is, variable substitutions will be performed on this `<var/>` element's `value` attribute right before template content is processed. Finally, the following example defines a template "cube":

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <def name="cube">
    <var name="size" value="1"/>
    <var name="x" value="0"/>
    <var name="y" value="0"/>
    <var name="annotation" value="a␣cube␣of␣size␣$size$" derived="true"/>
    <block id="$id$" xspan="$size$" yspan="$size$" zspan="$size$"
      location="($x$,$y$,$size/2$)">
      <description name="annotation" value="$annotation$"/>
    </block>
  </def>
</tabletop>
```

A cube defined here is a block of the same width, heigh, and depth, which are determined by variable `size`. Although this example contains only one content element `<block/>`, it is certainly possible to contain multiple more complicated XML structures. The location of a cube is determined by two variables `x` and `y`, while the Z coordinate, a half of `size`, will place the cube right on top of the table. The default values are: `size=1, x=0, y=0`. Note that variable `id` has no default value, and thus this value must come from each instance of cube. If the value of `id` is not assigned by an instance, SMILE will not perform variable substitution for this attribute (i.e., will leave it as it is), which will cause an error because a dollar sign is not a valid character for object IDs.

Variable annotation is marked as a derived variable, which means that its value is derived from other non-derived variable values, i.e., size in this case.

To create an instance using a template, element <instance/> can be used anywhere simple object elements are allowed. SMILE searches and replaces all occurrences of <instance/> elements with template content defined by <def/> elements. An <instance/> element has a single attribute def that specifies the name of a template. To pass variable values to a template, zero or more <var/> elements can be used as child elements of <instance/>. Each <var/> element has attributes name, value, and derived as described earlier. Note that variable values given by an instance override those given by a corresponding template. That is, when both the instance and the template assign different values to a variable of the same name, only the value assigned by the instance will be used. The following example instantiates a cube whose template is given in the previous example (assuming the template filename is def-cube.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <include file="tablesetup/def-cube.xml"/>
  <instance def="cube">
    <var name="size" value="5"/>
    <var name="id" value="c0"/>
    <var name="y" value="-1"/>
  </instance>
</tabletop>
```

The XML structure after all variable substitutions is automatically stored in debug.xml for debugging purposes. The content of debug.xml in this case is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <block id="c0" location="(0,-1,2.5)" xspan="5" yspan="5" zspan="5">
    <description name="annotation" value="a␣cube␣of␣size␣5"/>
  </block>
</tabletop>
```

Since the value of variable x is not assigned by the instance, the default value in the template x=0 is used.

When creating an instance using a template that contains nested instances of other templates, variable values in the enclosing instance are inherited by the enclosed instances. For variables of the same names, the values that are "the most local" to an instance are used. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <def name="block">
    <block id="$id$" location="($x$,$y$,$z$)"/>
  </def>
  <def name="pileOfTwo">
    <instance def="block">
      <var name="z" value="0.5"/>
      <var name="id" value="$id$Bottom"/>
    </instance>
    <instance def="block">
      <var name="z" value="1.5"/>
      <var name="id" value="$id$Top"/>
    </instance>
  </def>
  <instance def="pileOfTwo">
    <var name="x" value="3"/>
    <var name="y" value="2"/>
    <var name="z" value="10"/>
    <var name="id" value="pile"/>
  </instance>
</tabletop>
```

In this example, the variable values for the outer instance (of template `pileOfTwo`) are: `x=3, y=2, z=10, id=pile`. In either inner instance, values of `x=3` and `y=2` are inherited from the outer instance, and the value of `z` is locally assigned (`0.5` or `1.5`). The value of `id` in either inner instance is derived from the that of the outer instance, so that the value becomes `pileBottom` or `pileTop`. Note that attribute `derived="true"` is not needed for variable `id` because its value is derived from that of the outer instance, not from other variable values in the same instance. The resulting XML for the above example is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tabletop xmlns="http://synapse.cs.umd.edu/tabletop-xml" xspan="20" yspan="12">
  <block id="pileBottom" location="(3,2,0.5)"/>
  <block id="pileTop" location="(3,2,1.5)"/>
</tabletop>
```

7 Conclusion

This report has described the features and usages of SMILE, a software-simulated virtual environment for supporting robot imitation learning. In the virtual environment, a user can demonstrate procedural tasks, through intuitive GUI controls and mouse inputs, using various simulated objects on a tabletop. User-performed procedures can be recorded as demonstrations for training a robot agent. Since the demonstrator in SMILE is purposely made invisible to the robot, a demonstration appears as if objects move on their own. SMILE has three main interfaces, through which a user can affect the virtual environment. First, the demonstration interface consists of a set of overlaid GUI controls and mouse inputs that can be used to manipulate simulated objects and record demonstrations. Second, the robot interface allows a user to control the simulated robot in SMILE, either manually or through programming Matlab scripts. Third, the object interface allows a user to define and specify what objects are to be created in the environment, using XML structures.

In the future, we plan to continue adding more types of objects to SMILE, including tool objects that can be used to affect the state of other objects, such as scissors and screw drivers. This can significantly increase the types of actions that a demonstrator can do to objects, which is currently limited in moving objects and triggering controls.

Bibliography

Huang, D.-W., Katz, G., Langsfeld, J., Gentili, R., and Reggia, J. (2015a). A virtual demonstrator environment for robot imitation learning. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA).*

Huang, D.-W., Katz, G., Langsfeld, J., Oh, H., Gentili, R., and Reggia, J. (2015b). An object-centric paradigm for robot programming by demonstration. In Schmorrow, D. and Fidopiastis, C., editors, *Foundations of Augmented Cognition*, pages 745–756. Springer.

Katz, G., Huang, D.-W., Gentili, R., and Reggia, J. (2016). Imitation learning as cause-effect reasoning. In *Conference on Artificial General Intelligence (AGI).*