CoppeliaSim Tutorial

Author: Nick Short
Sponsor: Galois

Table of Contents

Introduction

This document serves as a basic tutorial for Coppelia Robotics' simulation suite, CoppeliaSim. This document was created as part of a Portland State University ECE/ME Capstone project sponsored by Galois. Dr. Marek Perkowski oversaw this project as the faculty advisor.

Scope

This tutorial will cover the basics of the simulation software, including an overview of the environment, menu options, and advanced dialogue boxes. Also included will be instructions on creating objects, attaching and configuring scripts, and hierarchical organization of objects.

Finally, there will be a section covering a selection of useful API functions built into the CoppeliaSim environment, as well as some of the pitfalls experienced during the creation of the Capstone project, as related to the simulation software.

Acknowledgements

I would like to recognize the other members of my Capstone team for shouldering the burden of basically every aspect of this project that did not revolve around CoppeliaSim. A hearty thanks to:

- Ben
- Jacob
- Victor
- Zeming
- Dylan

A special thanks to Jacob for his assistance with much of the control dynamics of the simulations.

I would also like to thank Dr. Marek Perkowski for advising on this project, as well as Dr. Mark Faust and Andrew Greenberg for their guidance during this project.

Finally, I would like to thank our sponsor's representative, Ethan for his supreme patience as this project metamorphized from what we thought it would be to what it became.

Environment

The following pages contain images and explanations of the various sections of the CoppeliaSim environment.

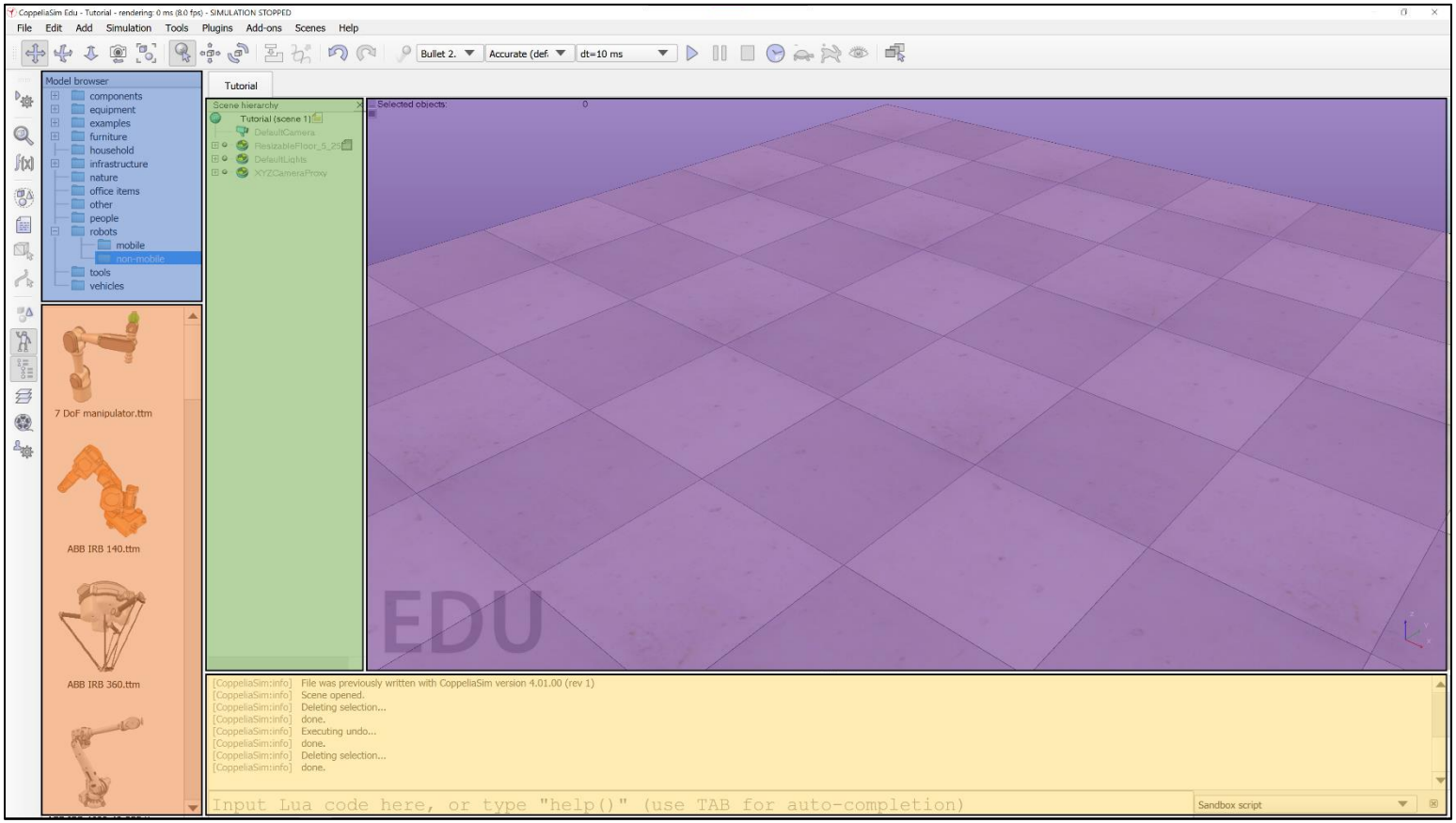

Figure 1 - CoppeliaSim environment

Figure 1 shows the CoppeliaSim environment, divided into functional sections. The blue section is the **Model browser** where the user can select from many different prefab models to insert into the simulation. The orange section is a visual list of the options available in the current selection of the **Model browser**. The green section is the **Scene hierarchy**—the list of all objects in the scene, respecting parent/child relationships. The purple section is the scene visualization. This is where the simulation plays out, and where the user can manually move objects around. The yellow section is the script output and (if enabled) the Lua Commander interface (Lua Commander will be discussed later). Information displayed using the Lua **print** function will be displayed here.

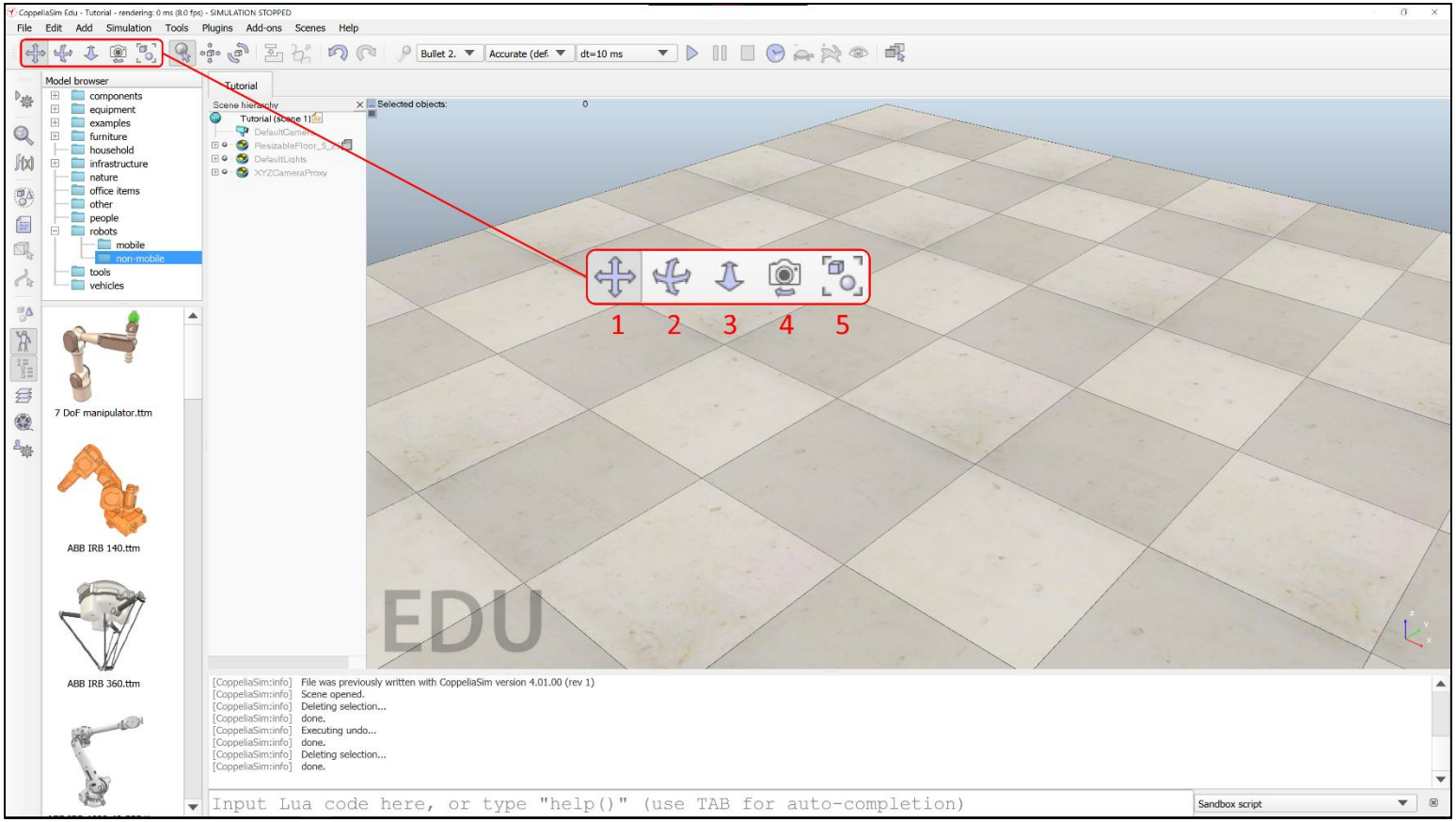

Figure 2 - Camera buttons

Toolbar Buttons – Camera

Figure 2 shows the camera view manipulation operations, as described below:

1. Translate – Move the camera view along the X-Y-Z axes.
2. Rotate – Rotate the camera view about the X-Y-Z axes.
3. Zoom – Zoom in and out. Function appears to be identical in orthogonal and perspective projection modes.
4. Camera Angle – In perspective projection mode, adjusts the projection angle of the camera. In orthogonal projection mode, acts as the zoom mode.
5. Fit-to-view – Fits the selected objects within a large portion of the scene view. If no items are selected, attempts to fit all objects within the view.

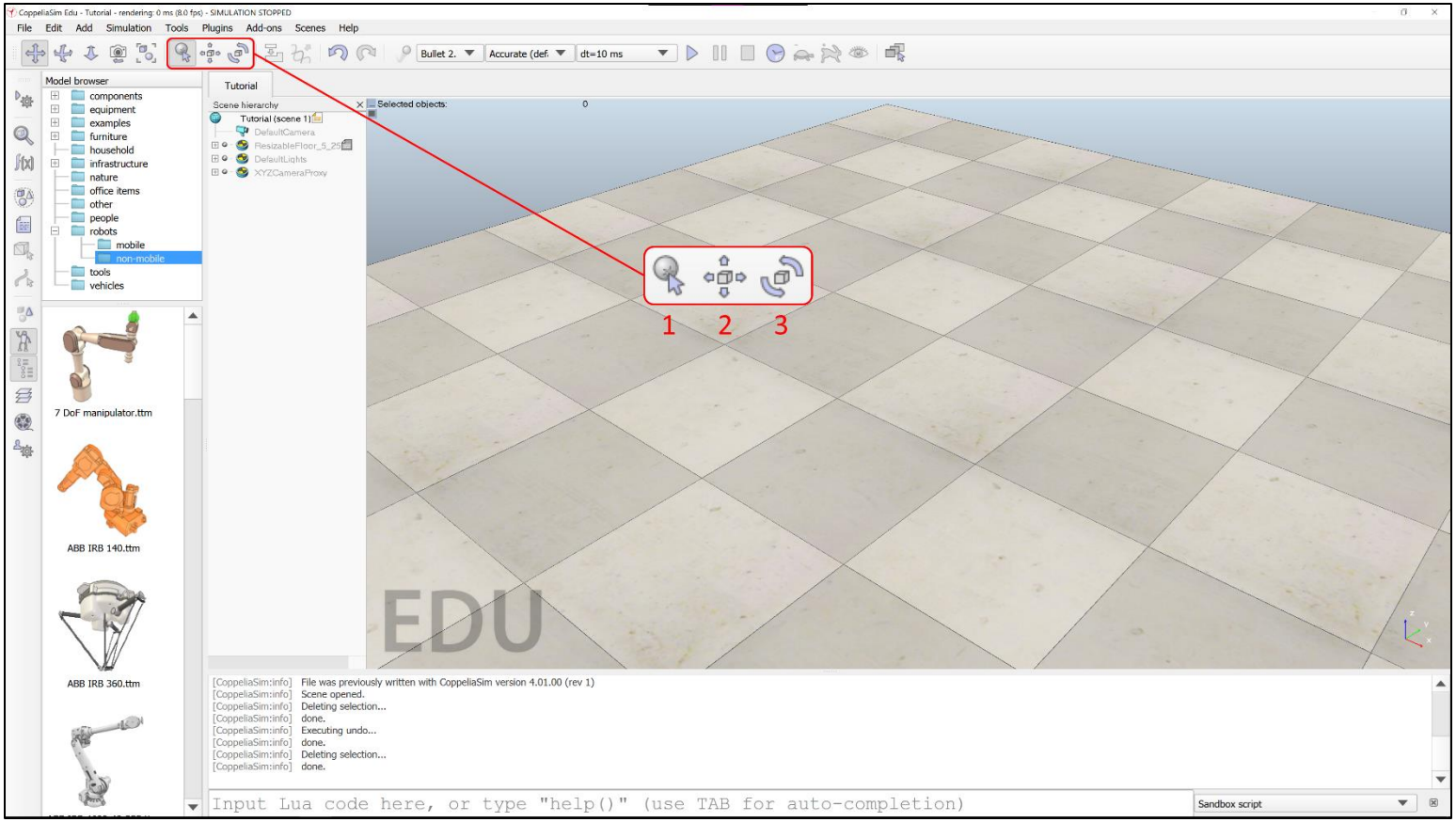

Figure 3 – Selection and object manipulation buttons

Toolbar Buttons – Selection and Object Manipulation

Figure 3 shows the buttons used to select an object and manipulate its position and orientation, as described below:

1. Object selection – Selects an object.
2. Object translation – Opens the object translation dialog box, allowing an object to be translated along one or more axes.
3. Object rotation – Opens the object rotation dialog box, allowing an object to be rotated around one or more axes.

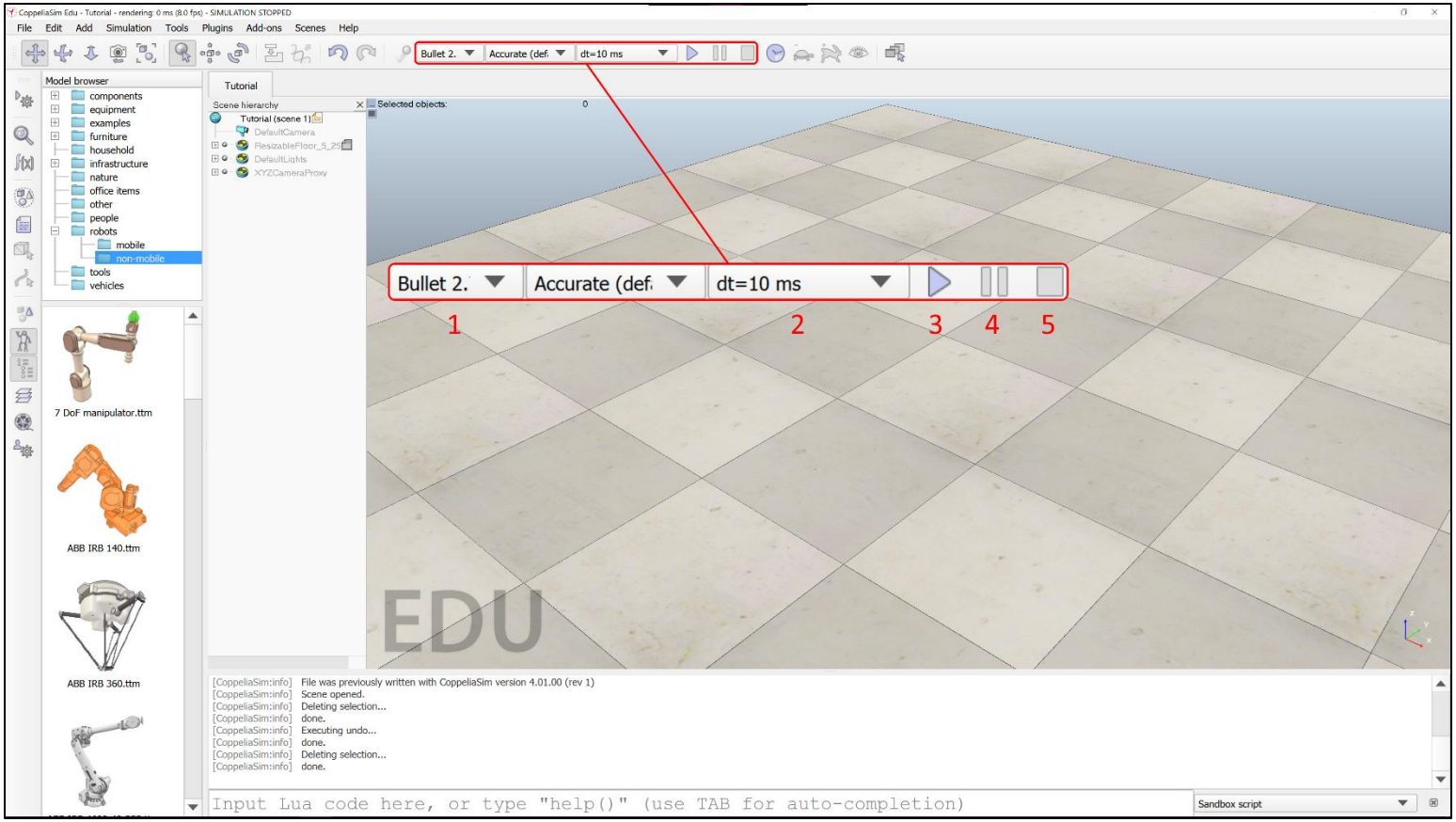

Figure 4 - Simulation buttons

Toolbar Buttons – Simulation

Figure 4 shows the buttons/drop downs used to change some parameters related to the simulation execution, as described below:

1. Physics engine – This drop down allows the user to select the desired physics engine. Available options are Bullet 2.78, Bullet 2.83, ODE, Vortex, and Newton.
2. Simulation time step – This drop down allows the user to select the amount of time that passes between simulation frames.
3. Start simulation – Starts the simulation.
4. Pause simulation – Pauses the simulation.
5. Stop simulation – Stops the simulation.

Menus

The following pages contain images and explanations of the various menus available in the CoppeliaSim environment.

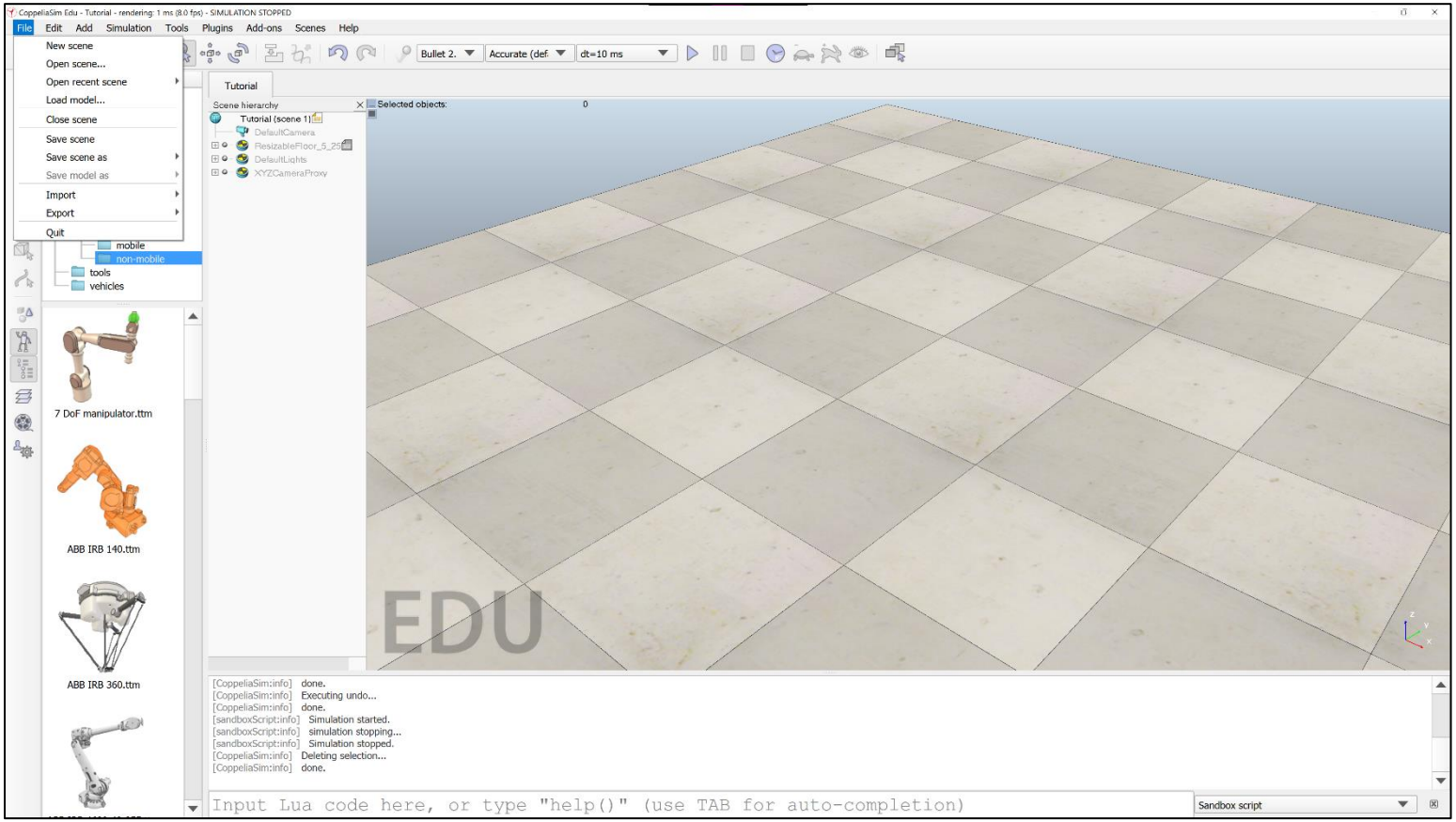

Figure 5 - File Menu

File

This is a standard file menu. It gives the user the option to create a new scene, save an open scene (or save a copy using “Save scene as”) and open a saved scene. The **Import**, **Export**, and **Load model…** options are also helpful, particularly **Import**. With the **Import** function, the user can add a custom complex shape (as a mesh) to a scene using a .dxf or .stl file, exported from the 3D modeling software of their choice. A list of recently opened scenes can also be found, allowing for quick access.

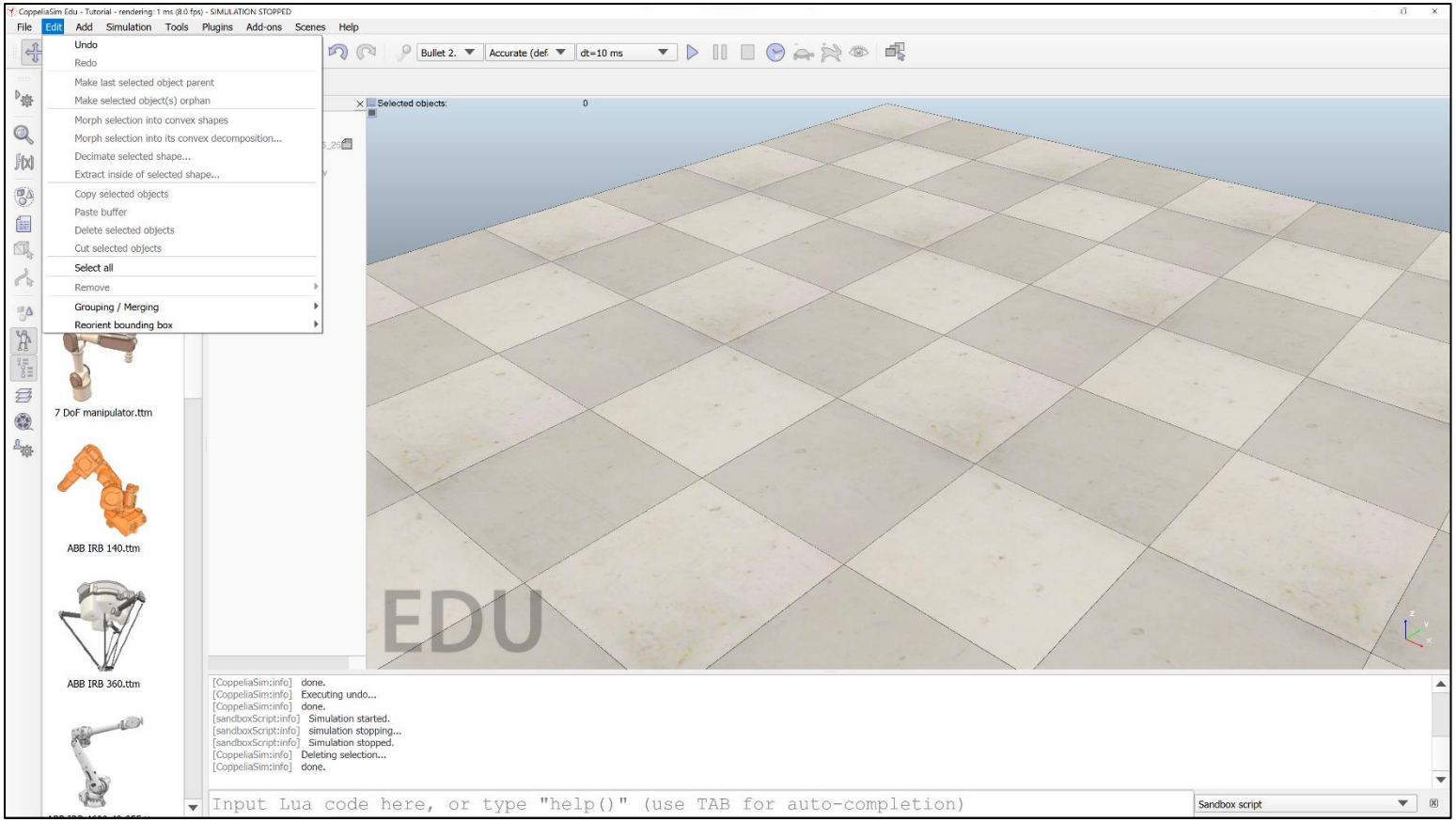

Figure 6 - Edit Menu

Edit

The edit menu has some standard edit menu options. **Undo**/**Redo**, **Select All**, and **Copy**/**Cut**/**Paste** have their normal uses. **Make last selected object parent**, when selecting multiple objects, makes the last object selected the parent object of the other selected objects. **Make selected object(s) orphan** causes all selected objects to have no child relationship with any other. **Grouping / Merging** allows the user to "combine" multiple objects into a single shape that can be translated, rotated, and otherwise utilized together. **Reorient bounding box** alters the rotation of the objects bounding box to be aligned with a different reference frame, usually the world reference frame.

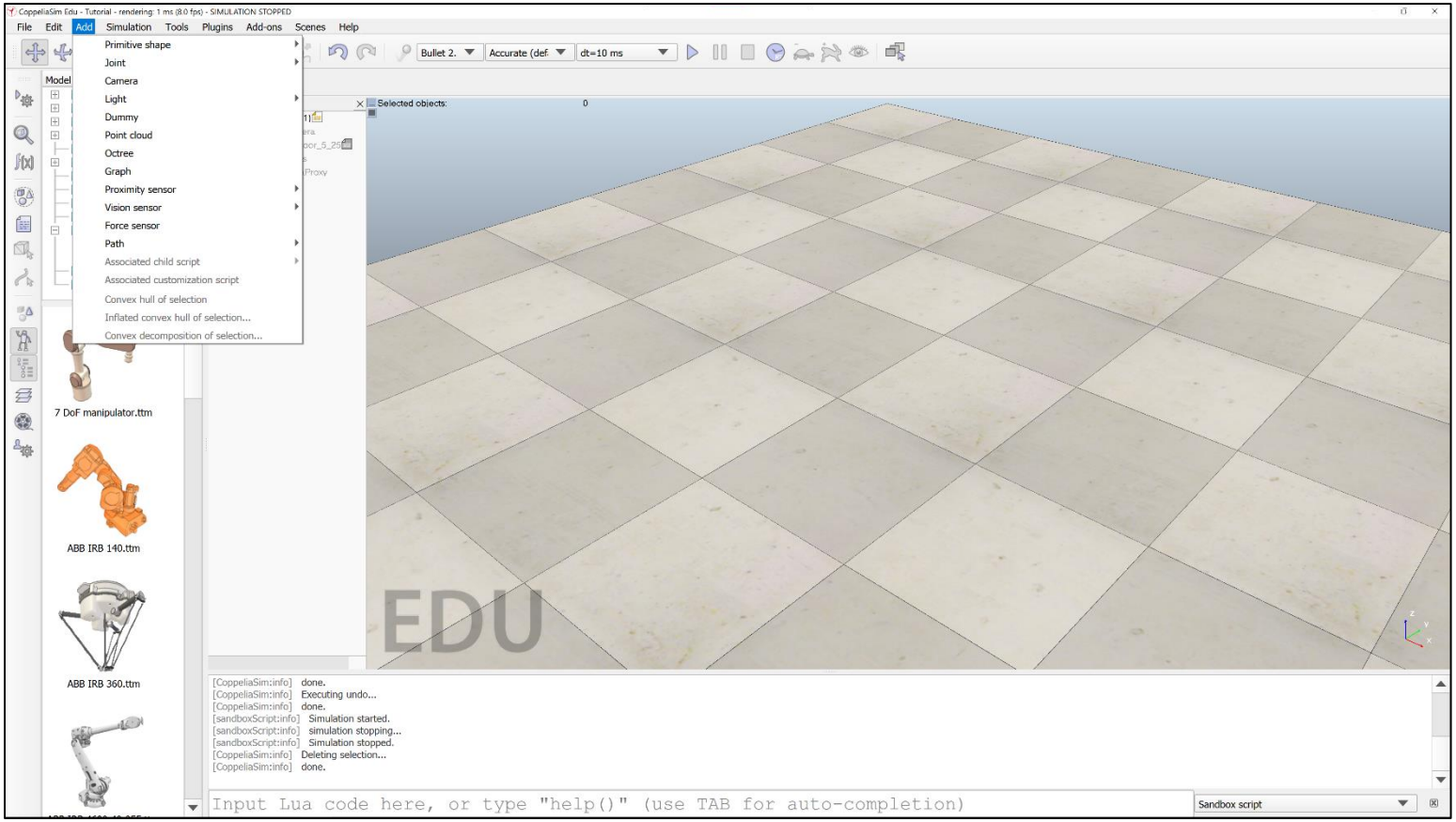

Figure 7 - Add Menu

Add

As the name suggests, the Add menu allows the user to add entities to the scene. Covered in this tutorial will be **Primitive Shape**, **Joint**, and **Associated child script**. For information on the addition entities that CoppeliaSim provides, please refer to the CoppeliaSim documentation, a link to which can be found at the end of this document.

Primitive shapes available include: Planes, Discs, Cuboids, Spheres, and Cylinders. Available joints include: Revolute, Prismatic, and Spherical.

Child scripts can be either threaded, or non-threaded.

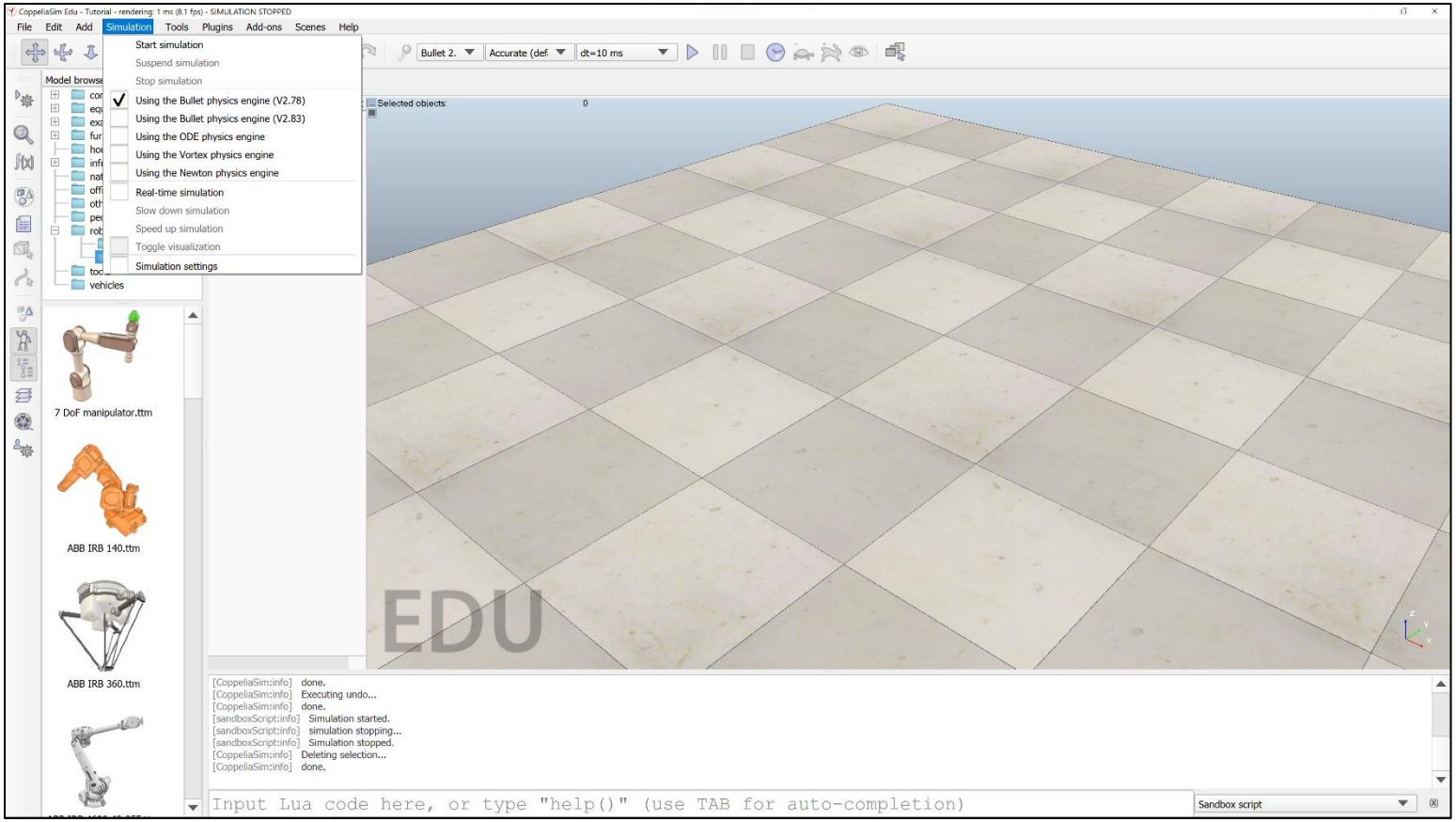

Figure 8 - Simulation Menu

Simulation

The simulation menu allows the user to start, pause, or stop the simulation, as well as select the desired physics engine. Also available are options to run a **Real-time simulation**, **Speed up** and **Slow down** the simulations, and open the **Simulation settings**.

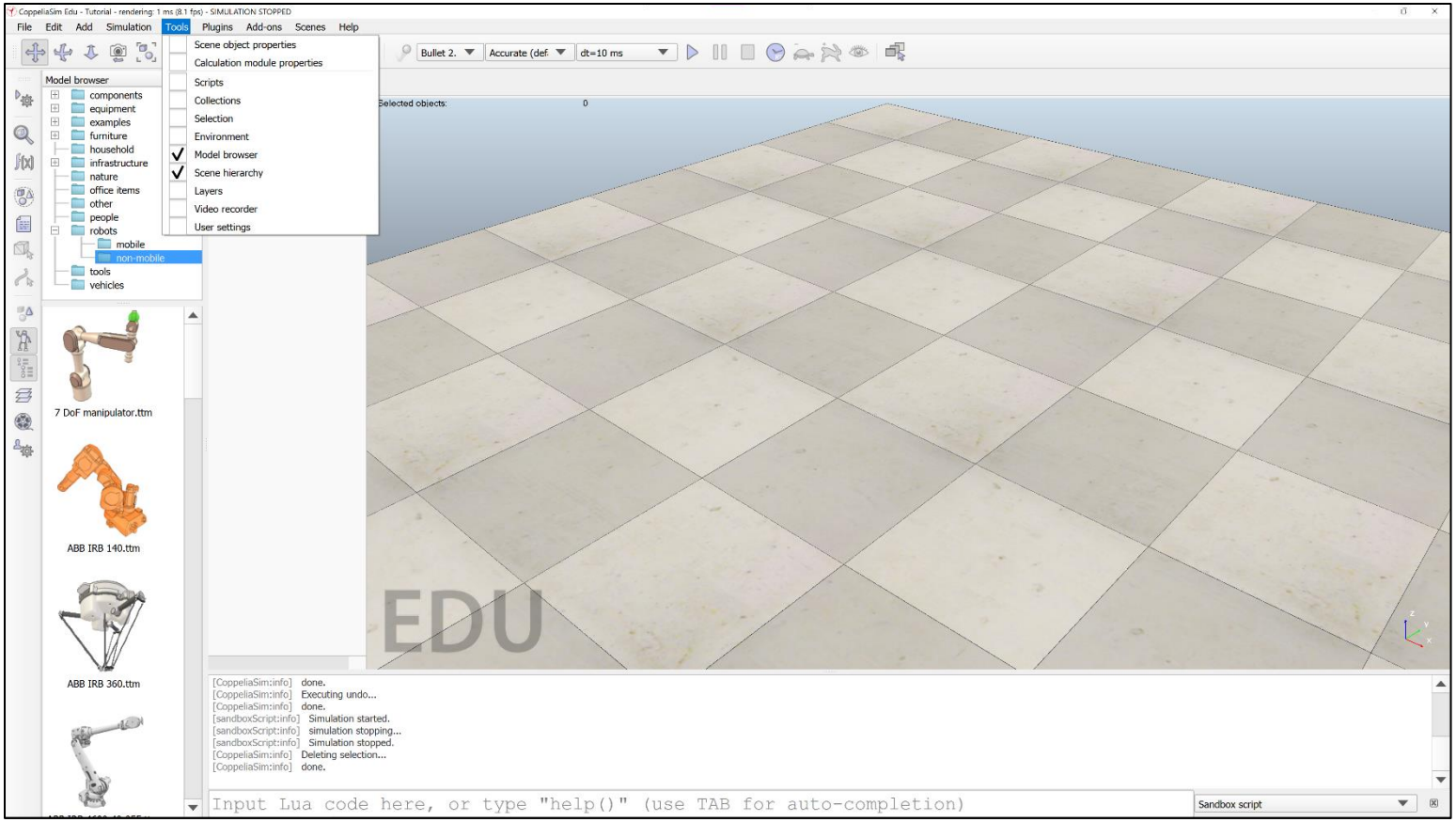

Figure 9 - Tools Menu

Tools

This menu gives the user the option of opening several different side-panes and advanced dialog boxes. Of note are the **Model browser** and **Scene hierarchy**, which are enabled by default in a new scene. Discussed later in this tutorial will be the **Scene object properties**, **Scripts**, and **User settings** advanced dialog boxes.

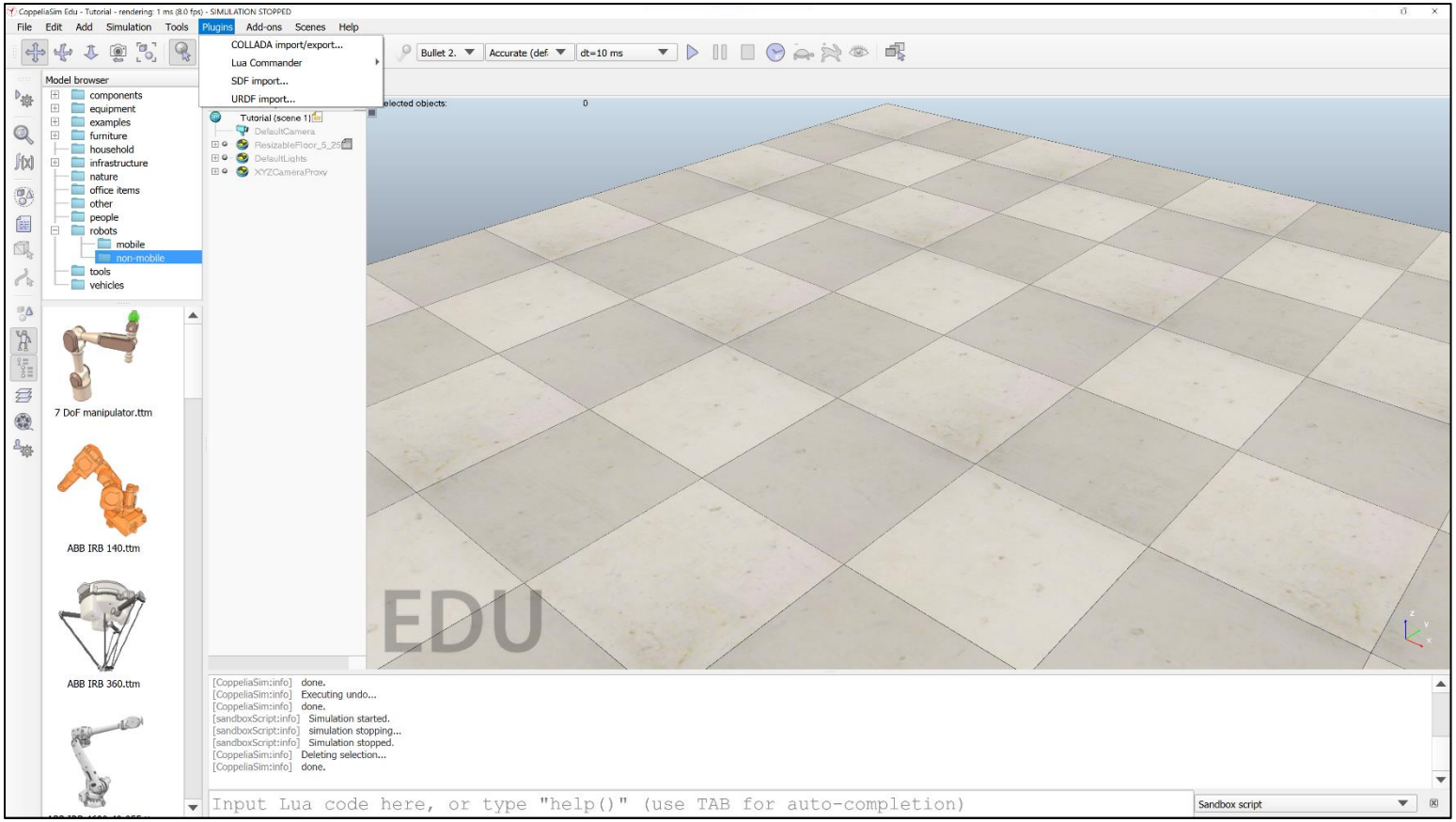

Figure 10 - Plugins Menu

Plugins

The only option in this menu addressed in this tutorial is **Lua Commander**. Entering the Lua Commander sub-menu and selecting **Enable** will allow the user to "inject" Lua commands—including functions included in the CoppeliaSim API—into the simulation during runtime. This feature allows the user to change many simulation runtime parameters, as well as user-defined **signals** (global variables). These commands are executed right after the last line executed in whichever script is being executed when the command is given to Lua Commander.

The utility of this plugin as a way to monitor signals in state-based simulations will be discussed in a later section.

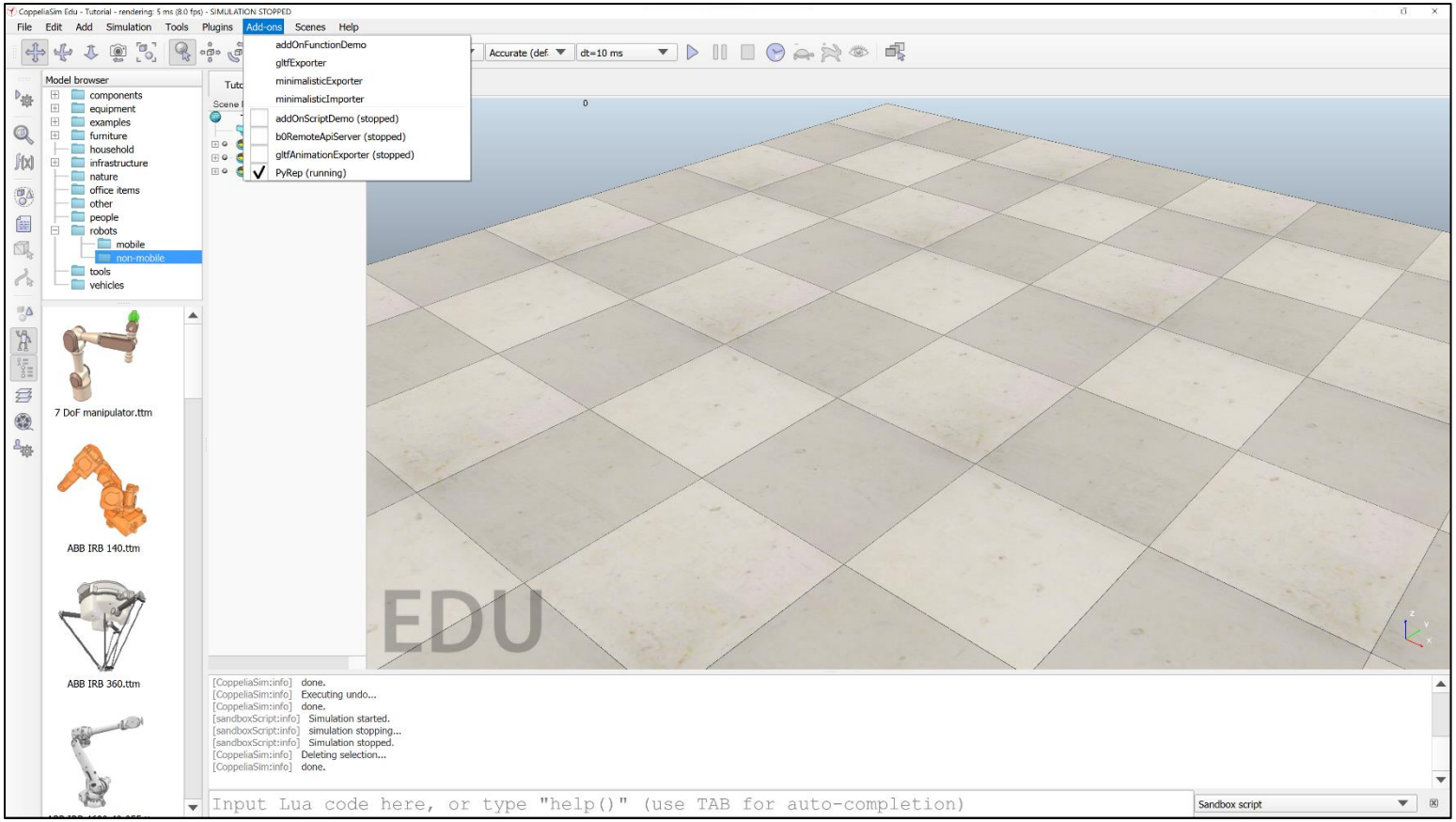

Figure 11 - Add-ons Menu

Add-ons

The **PyRep** add-on allows the user to control CoppeliaSim directly with Python. The creators of PyRep modified the open-source version of CoppeliaSim to eliminate the communication delays associated with remote API access (notably, the socket and interthread communication delays). This results in an approximate four orders of magnitude speed increase over the standard remote API access.

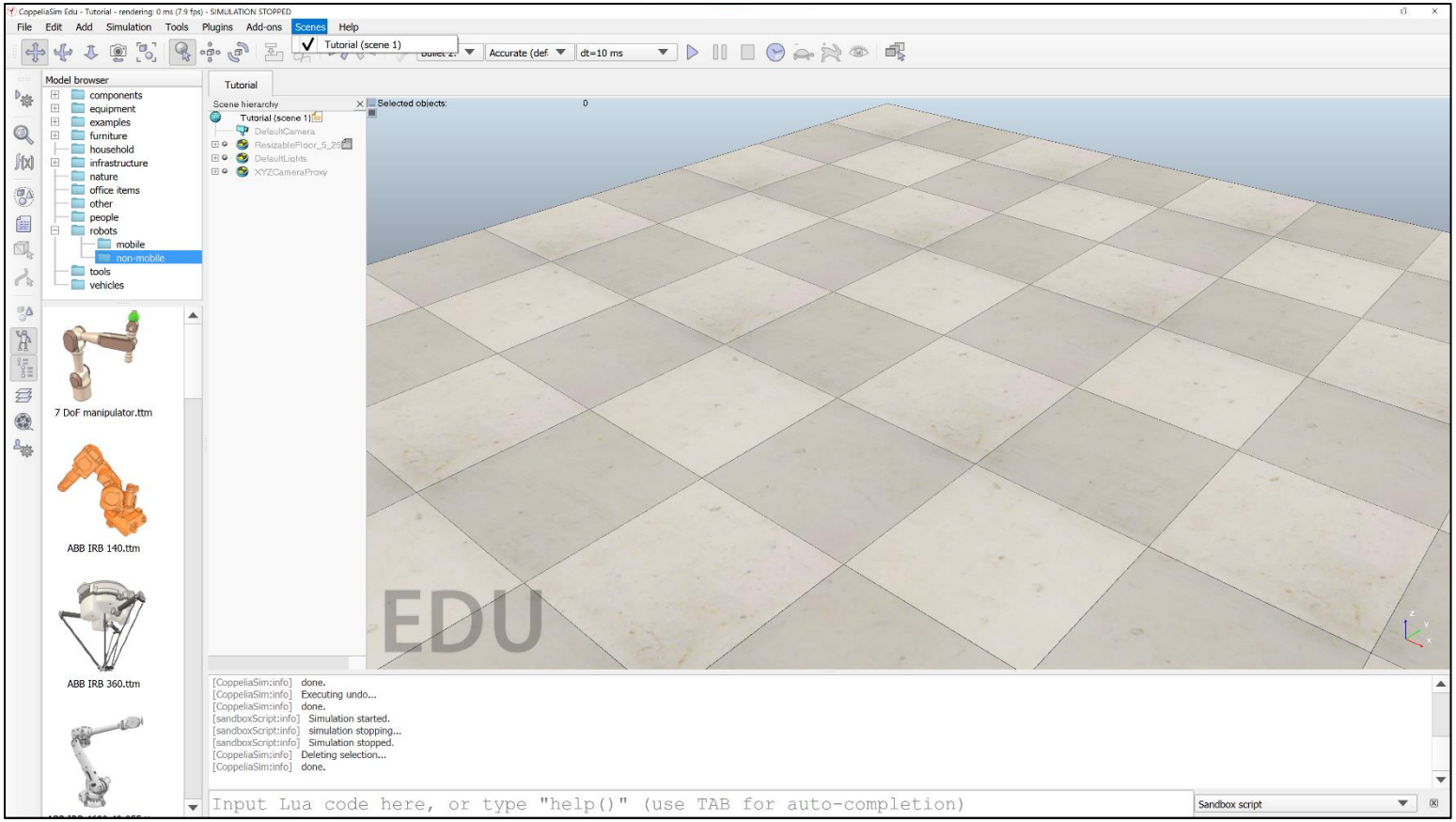

Figure 12 - Scenes Menu

Scenes

This menu is equivalent to a standard “Windows” menu, allowing the user to switch between active and inactive (but loaded) simulations.

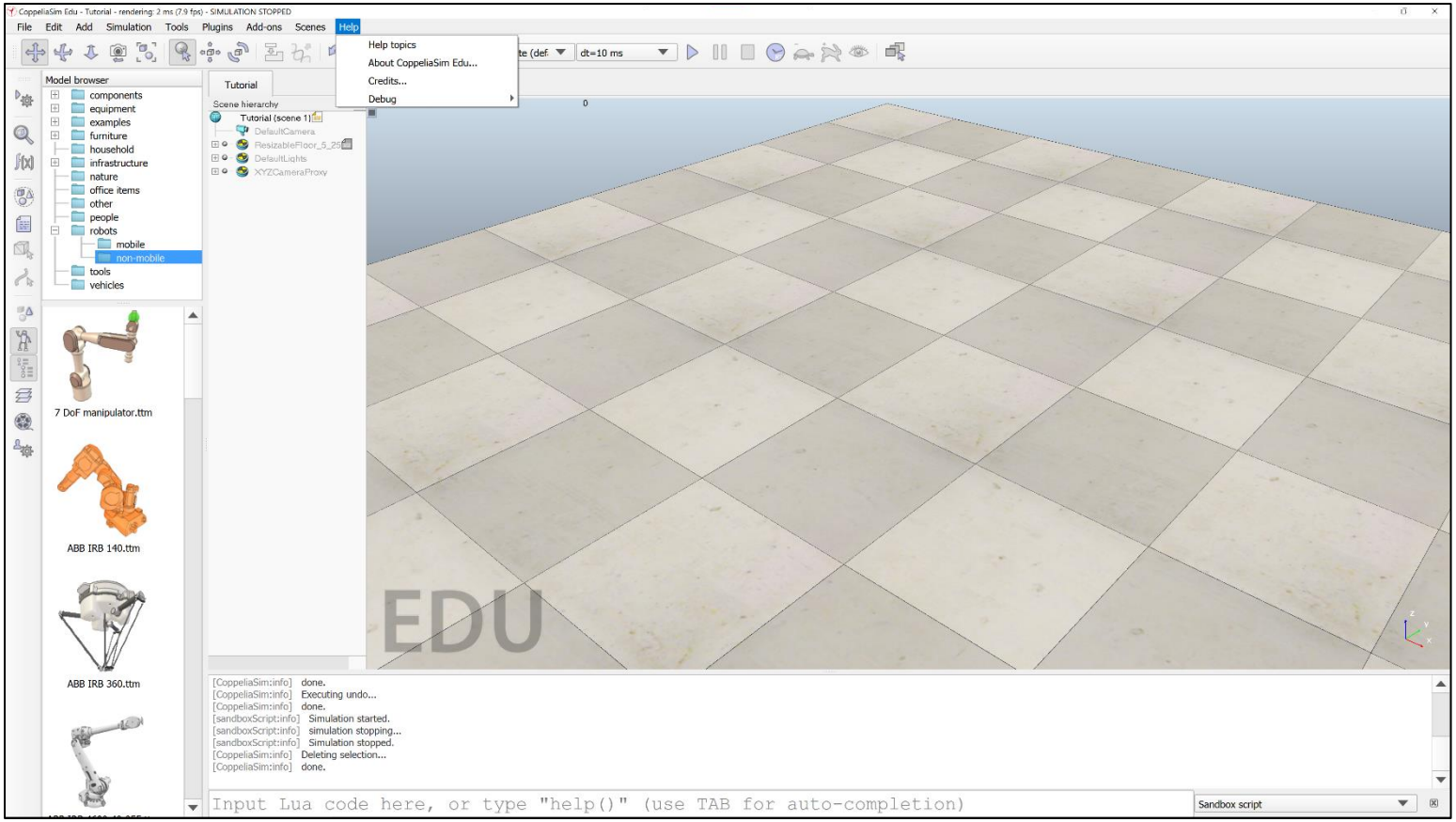

Figure 13 - Help Menu

Help

The **Help options** selection in the Help menu directs the user to the CoppeliaSim User Manual website. This manual contains explanations of the features and functions of CoppeliaSim, basic and advanced tutorials, and the complete CoppeliaSim API (including the remote API), among others.

First Steps

In this section, we will cover the basics of adding a primitive shape to a scene and looking at the **Geometry** and **Dynamic Properties** dialog boxes.

Adding a Shape to a Scene

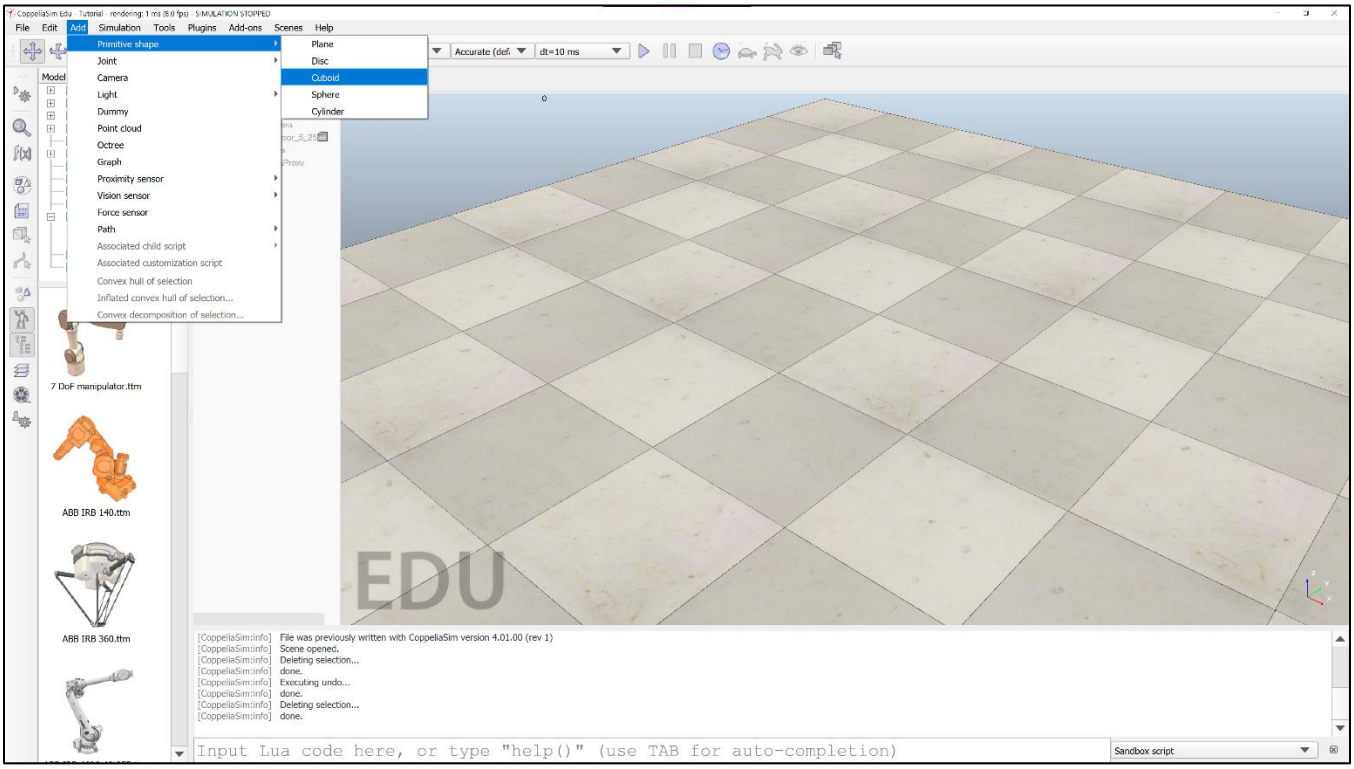

Figure 14 - Adding a shape to a scene

As shown in Figure 14, clicking through **Add->Primitive shape->Cuboid** will allow us to add a **Cuboid** shape to the scene. Upon clicking **Cuboid**, you will see the following dialog box, shown in Figure 15.

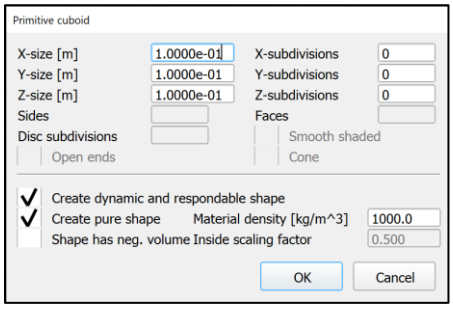

Figure 15 - Add primitive shape dialog box (cuboid)

In this dialog box, the dimensions and density of the new primitive shape can be configured. The shape can also be configured as a dynamic and respondable shape, allowing it to be affected by the physics engine, and to collide with other objects.

Modifying a Shape's Geometry and Dynamic Properties

A shape's geometry and dynamic properties can be modified post creation, through the appropriate dialog box. First, double-click on the shape's symbol (the blue cube, in this case) in the hierarchical tree, as shown in Figure 16.

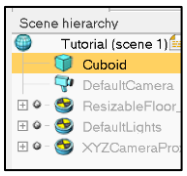

Figure 16 - Opening shape properties

After double-clicking on the shape's symbol, the dialog box in Figure 17 will be shown.

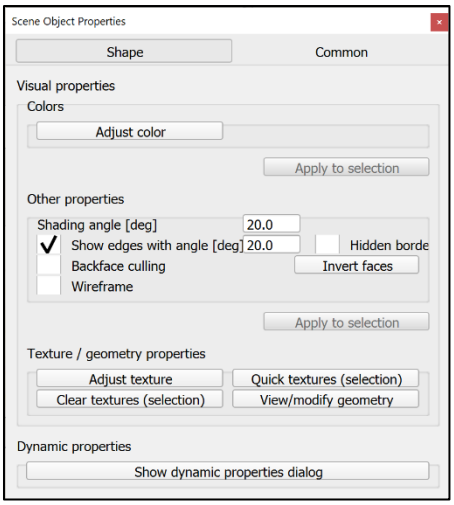

Figure 17 - Object properties

From this dialog box, the user has access to the geometry and dynamic properties dialog boxes.

Figure 18 shows the **Geometry associated with 'shape'** dialog box (here, for a cuboid).

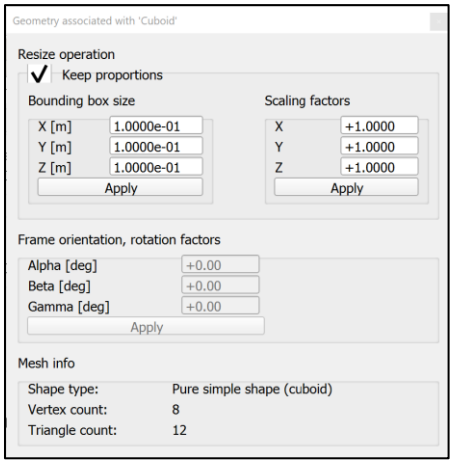

Figure 18 - Object geometry dialog box

This dialog allows the user to adjust the dimensions of a shape, either by setting a dimension to an absolute size, or by scaling the dimension by a factor. By default, the **Keep proportions** option is selected and the other dimensions will be scaled by a factor similar to how the altered dimension is scaled.

Figure 19 shows the **Rigid Body Dynamic Properties** dialog box. It is recommended that the user only alter whether or not the object is respondable or dynamic, as changing the settings below the **Body is dynamic** checkbox can drastically affect the simulation (Principle moments of inertia and position/orientation of the inertial frame are all calculated at shape creation, using the shape's dimensions and mass. Mismatches between mass and inertial moments can cause odd things during simulaion.)

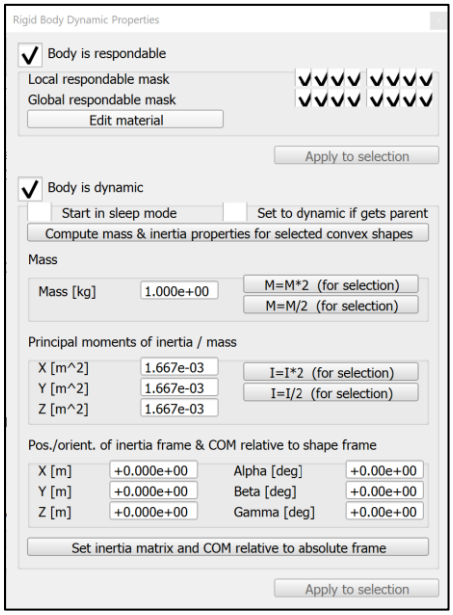

Figure 19 - Shape dynamic properties dialog box

Manipulating a Shape's Position and Orientation

The position and orientation of a shape can be adjusted precisely with a few different methods. Figures 20 through 22 show the various options available for shape translation.

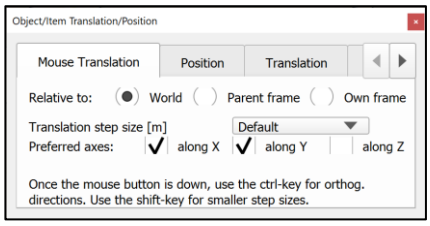

Figure 20 - Mouse translation

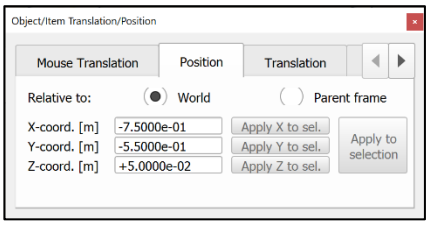

Figure 21 – Specific position

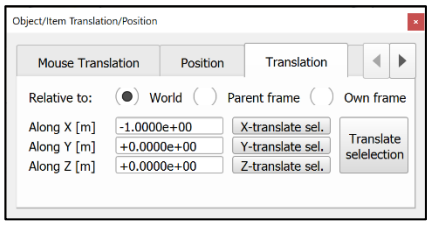

Figure 22 - Exact translation along axes

Mouse translation allows the user to drag a shape around the scene. The user can "lock" axes by deselecting them. With specific position, the user can specify the exact location of a shape in X-Y-Z, relative to the world, or the parent frame. This is useful for centering a child object on a parent object in one or more axes (to center on a parent, the coordinate would just be set to 0). Exact translation allows the user to "slide" a shape along one or more axes by a specific amount.

Figures 23 through 26 show the various options for shape rotation.

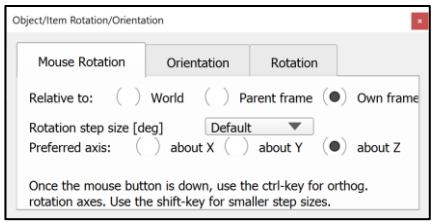

Figure 23 - Mouse rotation

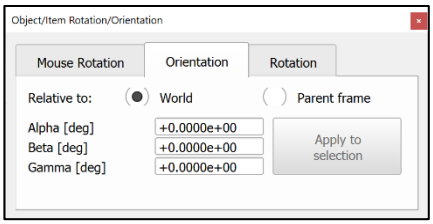

Figure 24 – Specific rotation

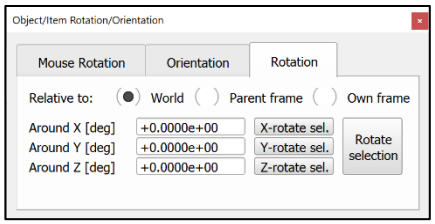

Figure 25 - Exact rotation about axes

Mouse rotation allows the user to freely rotate a shape about an axis using the mouse. Specific rotation allows the user to rotate a shape to a specific angle, relative to the world, or the parent frame. This is useful for equalizing the rotation between parent-child objects. Exact rotation about an axis allows the user to rotate a shape about one or more axes by a specific amount.

Joints

In CoppeliaSim, joints are what hold objects together. There are three types of joints: **revolute**, **prismatic**, and **spherical**.

- Revolute – These joints allow one shape to rotate with respect to another shape.
- Prismatic – These joints allow one shape to extend or retract from another shape. A linear actuator is an example of this.
- Spherical – These joints allow one shape to rotate in three dimensions about another shape.

Joints have three modes of operation: **passive**, **dependent**, and **torque/force**. The passive mode is the only mode used in this basic tutorial. In passive mode, a joint connects two objects together and the joint position can be controlled through the API.

Scripts

Scripts are an integral part of CoppeliaSim simulations. They control pretty much everything aside from the application of the physics engine. Scripts are written in Lua, using CoppeliaSim's API functions, which all follow the format **sim.someFunctionInCoppeliaSIM(arg1, arg2, …)**. In addition to CoppeliaSim's API functions, all base Lua functions (for Lua version 5.1) are available, as well as some extended libraries, like **string** and **math**.

Scripts in CoppeliaSim are attached to shapes or other objects and have a configurable execution order, largely based upon the parent-child hierarchy of the scene. Scripts are not accessible to external editors, as they are stored in the simulation **.ttt** file. Scripts are not always required to be attached to shapes or other objects, though. Through the Lua function **require(path)**, the user can execute external scripts within an integrated script.

The user can define their own functions to perform often repeated actions, or even to expand upon existing CoppeliaSim API functions (often referred to as 'wrapping'). All user-defined functions must be called within one of the built-in state functions, or they will not execute.

Script Types

There are two types of scripts: child and customization. Additionally, child scripts can either threaded or non-threaded.

- Child – These scripts are executed each simulation pass (so, only when the simulation is running), in parts (aside from the **sysCall_init()** function, which is called at the beginning of the simulation, and only once).
  - Threaded – Threaded scripts are not covered in this tutorial. They are useful for some things, but carry a hefty thread-switching overhead which can quickly make simulations choppy and unresponsive.
  - Non-threaded – Non-threaded scripts must be non-blocking, meaning that they should always return control to the system (e.g. no wait loops).
- Customization – Customization scripts run all the time, whether a simulation is started or stopped. These scripts are useful, especially when combined with a remote API connection. For example, a GUI could be designed in Python to allow the live scaling of an object's bounding box.

There are four main built-in state functions for child scripts: **sysCall_init()**, **sysCall_actuation()**, **sysCall_sensing()**, and **sysCall_cleanup()**. All **sysCall_init()** functions are executed at the start of a simulation. All **sysCall_actuation()** functions are executed during the actuation stage of a simulation frame. Likewise, all **sysCall_sensing()** functions are executed during the sensing stage of a simulation frame. The **sysCall_cleanup()** functions are called at the end of a simulation.

Customization scripts have three built-in state functions (in addition to **init** and **cleanup**, which function the same as in child scripts): **sysCall_nonSimulation()**, **sysCall_beforeSimulation()**, and **sysCall_afterSimulation()**. **sysCall_nonSimulation()** runs constantly when the simulation is not running. **sysCall_beforeSimulation()** runs just before a simulation starts, and **sysCall_afterSimulation()** runs just before a simulation ends.

Adding a Script to a Shape

Figure 26 shows how to add a script to a selected shape or object using the **Add** menu: **Add->Associated child script->Non threaded**. This option is only highlighted when a shape or object is selected.

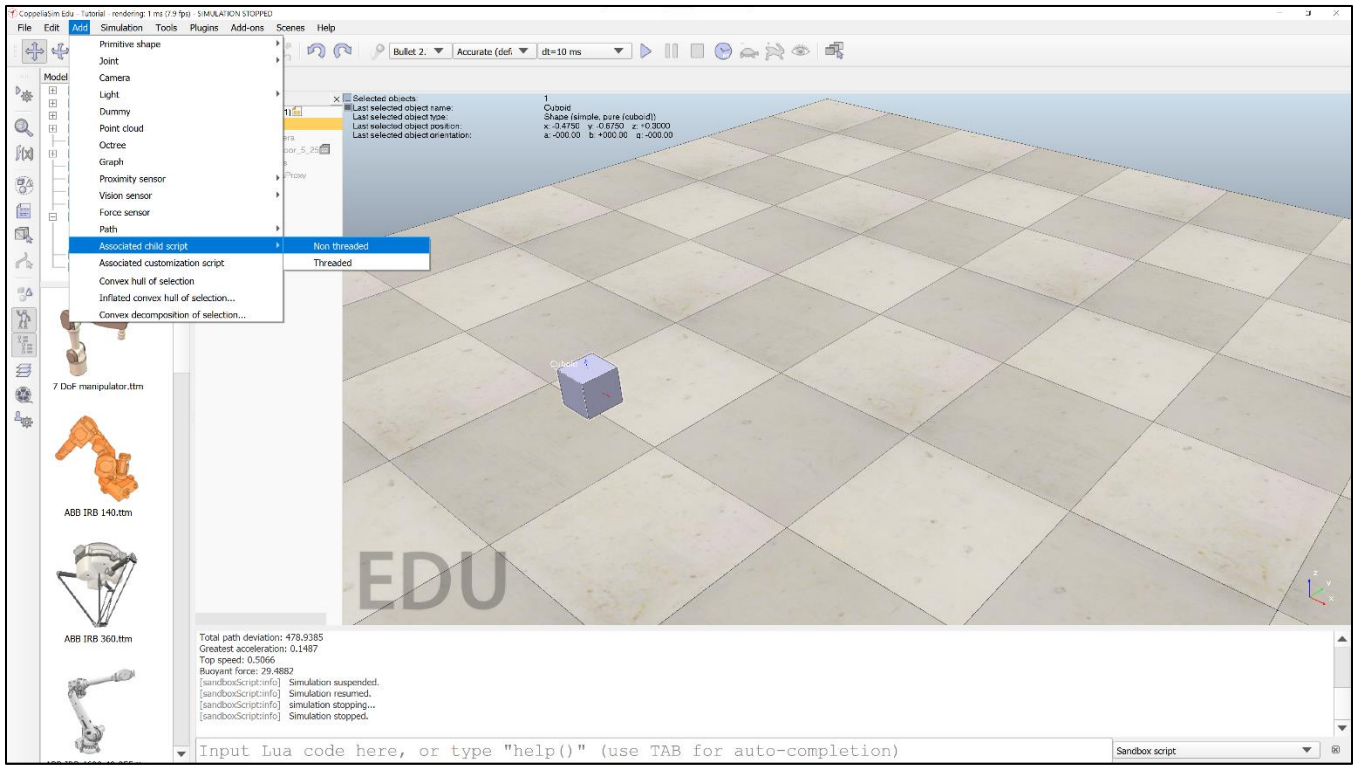

Figure 26 - Adding a script to a shape

API

This section will cover some common CoppeliaSim API calls and will address some caveats associated with them (as necessary). There will be a couple of suggestions for custom functions that implement a functionality similar to an API call, where the API call returns an unintuitive, but similar value.

Common API Calls

The following is a list of common API calls (not included, but required is a **sim.** prefix):

- **getObjectHandle(string objectName)** – This API call returns the integer handle of the object named objectName.
- **getObjectOrientation(int objectID, int relativeTo)** – This API call returns a three member table containing the rotation (in radians) matrix of the object with handle objectID, relative to the object with handle relativeTo.
  - If -1 is passed as relativeTo, the orientation is returned relative to the world.
  - The angles returned are the Euler angles, different than what one would commonly consider a rotation angle.
- **getObjectPosition(int objectID, int relativeTo)** – This API call returns a three member table containing the position (in meters) matrix of the object with handle objectID, relative to the object with handle relativeTo.
  - If -1 is passed as relativeTo, the position is returned relative to the world.
- **getObjectVelocity(int objectID)** – This API call returns two three member tables.
  - The first table contains the objects linear velocities (units of m/s) in x, y, and z.
  - The second table contains the objects angular velocities (units of rad/s) in x, y, and z.
  - The radians in the angular velocity table are Euler angles.
- **alphaBetaGammaToYawPitchRoll(float alpha, float beta, float gamma)** – This API function converts alpha-beta-gamma angles to yaw-pitch-roll angles.
  - It returns three separate float values—yaw first, then pitch, and, finally, roll.
- **checkDistance(int objectID1, int objectID2, float threshold)** – This API call returns the minimum distance between the bounding boxes of the two objects (or collections of objects).
  - objectID1 and objectID2 can be the integer handles for individual objects, object collections, or any combination of those. The threshold determines whether the distance is returned. If the distance is greater than the threshold, the distance is not returned. If threshold is ≤0, then no threshold is used.
  - The call returns three values.
    1. The success of the call (0 if failed, 1 if no threshold or distance is smaller than threshold).
    2. A seven-member table of floats, where members 1-6 represent the distance segment. The last member is the minimum distance.
    3. A tuple containing the integer handles of the two objects with the minimum distance.
  - See caveats.
- **addForceAndTorque(int objectID, table3 forces, table3 torques)** – This API call is the main interaction with the physics engine. It applies a 3-dimensional force vector and torques about each axis to the object with handle objectID.
- **setIntegerSignal(string signalName, int value)** – This API call sets an integer signal with name signalName to a specific value.

- **getIntegerSignal(string signalName)** – This API call retrieves the value of the integer signal with name signalName

Note: There are getXSignal and setXSignal API calls for many of the variable types in Lua. See the CoppeliaSim API reference for a complete list.

Caveats

The use of some functions in CoppeliaSim may not be intuitive, or the functions may return values that may differ from what you expect. The best example of this is:

sim.checkDistance(int objectID1, int objectID2, float threshold)

If one has a habit of forgetting that CoppeliaSim is a *physics-based* robotics simulator, one might think that this API call returns just the distance between the two objects center points. One might think, though, that collision detection is a common need in physics simulation. Then, one might remember that this API call returns the minimum distance between objects.

One might also forget that this function returns multiple values (counter-intuitive if one is just expecting a distance). This memory lapse would result in all distance measurements being equal to one, which is obviously broken, but does not error.

In Lua, when functions return multiple values, assignment is like Python, except that "uncaught" return values are discarded in Lua, rather than converted into a tuple, as in Python. As an example, the proper use of the above API call is:

success, distances, objectPair = sim.checkDistance(int objectID1, int objectID2, -1)

The value of **success** would be 1, **distances** would be a table[7], and **objectPair** would be a table[2]. If, instead, the following command was executed:

success = sim.checkDistance(int objectID1, int objectID2, -1)

The value of **success** would be 1. In Python, the value of **success** would be:

(1, [val1, val2, … , val7], [handle1, handle2])

The other caveat that is important to mention is that API calls that return angles return the Euler angles (alpha, beta, gamma), not the angles that most people are used to (yaw, pitch, and roll). The only exception to this is the API call that converts from alpha/beta/gamma to yaw/pitch/roll values.

Manual calculations on Euler angles will not work as expected, so they should be converted before use.

Additional Resources

CoppeliaSim Online Reference Manual – https://www.coppeliarobotics.com/helpFiles/

PyRep Github Repository – https://github.com/stepjam/PyRep