Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator

Conference Paper · November 2010

DOI: 10.1007/978-3-642-17319-6_8 · Source: DBLP

CITATIONS
83

READS
1,485

4 authors:

Marc Freese
Tokyo Institute of Technology
20 PUBLICATIONS **414** CITATIONS
SEE PROFILE

Surya Singh
Army Institute of Technology
270 PUBLICATIONS **4,467** CITATIONS
SEE PROFILE

Fumio Ozaki
Shonan Institute of Technology
62 PUBLICATIONS **552** CITATIONS
SEE PROFILE

Nobuto Matsuhira
Shibaura Institute of Technology
168 PUBLICATIONS **654** CITATIONS
SEE PROFILE

Virtual Robot Experimentation Platform V-REP: A Versatile 3D Robot Simulator

Marc Freese*, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira

*K-Team Corporation, Y-Parc - Rue Galilée 9,
1400 Yverdon-les-Bains, Switzerland
mfreese@gmx.ch, spns@acfr.usyd.edu.au,
{fumio.ozaki,nobuto.matsuhira}@toshiba.co.jp
www.v-rep.eu, www.k-team.com

Abstract. From exploring planets to cleaning homes, the reach and versatility of robotics is vast. The integration of actuation, sensing and control makes robotics systems powerful, but complicates their simulation. This paper introduces a modular and decentralized architecture for robotics simulation. In contrast to centralized approaches, this balances functionality, provides more diversity, and simplifies connectivity between (independent) calculation modules. As the Virtual Robot Experimentation Platform (V-REP) demonstrates, this gives a small-footprint 3D robot simulator that concurrently simulates control, actuation, sensing and monitoring. Its distributed and modular approach are ideal for complex scenarios in which a diversity of sensors and actuators operate asynchronously with various rates and characteristics. This allows for versatile prototyping applications including systems verification, safety/remote monitoring, rapid algorithm development, and factory automation simulation.

Keywords: Robot Simulator, V-REP, Distributed Control

1 Introduction

The overall architecture and control methodology are crucial elements in robot simulators. A robust systems approach advocates for a versatile and fine-grained simulation strategy. By emphasizing modularity, scalability and expandability, simulations remain robust particularly when abstracting underlying robotic systems since system specificities cannot be foreseen.

The increased processing power of computers, the advent of several dynamics (physics) libraries, as well as faster 3D graphics hardware have drastically changed the landscape in the field of (3D) robotics simulation. While it is possible to cobble together these various elements almost trivially, good simulation requires careful architecture to yield both performance and accurate calculations (particularly with respect to dynamic conditions). For example, rather than using dynamics blindly for every aspect of a simulation, it is preferable to use it only when other methods (e.g. kinematics) fail. A modular architecture allows for the combination of various functionality to obtain the best possible synergy.

Compared to robots a few years ago, robots now employ highly complex control methods. Practically those are implemented in a distributed fashion in order to simplify development and the overall control task complexity. Just as distributed communication networks and protocols allow a *plug-and-play* behavior [3], a distributed control approach is needed for robotic simulation. In addition to robustness and parallelism (similar to the distributed hardware), it allows a robotic simulator to offer a *copy-and-paste* functionality not only for objects or models, but also for their associated control methods: robots or robot elements can be placed and combined in a scene without having to adjust any control code.

The paper is organized in three parts. The first part of this paper focuses on V-REP's architecture; based on various functionalities wrapped in *scene objects* or calculation modules, it allows the user to pick from them as requires, or as best for a given simulation. Moreover a special care has been given so as to develop those functionalities in a balanced manner, giving each one the same amount of attention as one would do in a real robotic system. The second part of this paper focuses on V-REP's control methodology; in V-REP, control is distributed and based on an unlimited number of scripts that are directly associated or attached to *scene objects*. Finally, in order to further clarify elements introduced in the first and second part of this paper, the third part describes a typical V-REP simulation set-up.

2 V-REP's Architecture

V-REP is designed around a versatile architecture. There is no *main* or *central* function in V-REP. Rather, V-REP possesses various relatively independent functions, that can be enabled or disabled as required.

Imagine a simulation scenario where an industrial robot has to pick-up boxes and move them to another location; V-REP computes the dynamics for grasping and holding the boxes and performs a kinematic simulation for the other parts of the cycle when dynamic effects are negligible. This approach makes it possible to calculate the industrial robot's movement quickly and precisely, which would not be the case had it been simulated entirely using complex dynamics libraries. This type of hybrid simulation is justified in this situation, if the robot is stiff and fixed and not otherwise influenced by its environment.

In addition to adaptively enabling various of its functionalities in a selective manner, V-REP can also use them in a symbiotic manner, having one cooperate with another. In the case of a humanoid robot, for example, V-REP handles leg movements by (a) first calculating inverse kinematics for each leg (i.e., from a desired foot position and orientation, all leg joint positions are calculated); and then (b) assigns the calculated joint positions to be used as target joint positions by the dynamics module. This allows specifying the humanoid motion in a very versatile way, since each foot would simply have to be assigned to follow a 6-dimensional path: the rest of calculations are automatically taken care of.

2.1 *Scene Objects*

A V-REP simulation scene contains several *scene objects* or elemental objects that are assembled in a tree-like hierarchy. The following *scene objects* are supported in V-REP:

- **Joints**: *joints* are kinematic lower pairs for that link two or more *scene objects* together with one to three degrees of freedom (e.g., prismatic, revolute, screw-like, etc.).
- **Paths**: *paths* allow complex movement definitions in space (succession of freely combinable translations, rotations and/or pauses), and are used for guiding a welding robot's torch along a predefined trajectory, or for allowing conveyor belt movements for example. Children of a path object can be constrained to move along the path trajectory at a given velocity.
- **Shapes**: *shapes* are triangular meshes, used for rigid body visualization. Other *scene objects* or calculation modules rely on *shapes* for their calculations (collision detection, minimum distance calculation, etc.).
- **Cameras and lights**: *cameras* and *lights* are used for scene visualization purposes mainly, but can also have effects on other *scene objects* (e.g. *lights* directly influence *rendering sensors*).
- **Dummies**: *dummies* are "points with orientation," or reference frames, that can be used for various tasks, and are mainly used in conjunction with other *scene objects*, and as such can be seen as "helpers."
- **Proximity sensors**: The *proximity sensors* objects perform an exact minimum distance calculation within a given detection volume [4] (see Fig. 1) as opposed to simply performing collision detection between some selected *sensing rays* and the environment; hence allowing for reflectance effects due to sensor/surface angles.
- **Rendering sensors**: *rendering sensors* in V-REP are camera-like sensors, allowing to extract complex image information from a simulation scene (colors, object sizes, depth maps, etc.) (see Fig. 1). The built-in filtering and image processing enable the composition of as blocks of filter elements (with additional filter elements via plugins). *Rendering sensors* make use of hardware acceleration for the raw image acquisition (OpenGL).
- **Force sensors**: *force sensors* are rigid links between *shapes*, that can record applied forces and torques, and that can conditionally saturate.
- **Mills**: *mills* are customizable convex volumes that can be used to simulate surface cutting operations on *shapes* (e.g., milling, laser cutting, etc.).
- **Graphs**: *graphs* are *scene objects* that can record a large variety of one dimensional data streams. Data streams can be displayed directly (time graph of a given data type), or combined with each other to display X/Y graphs, or 3D curves.

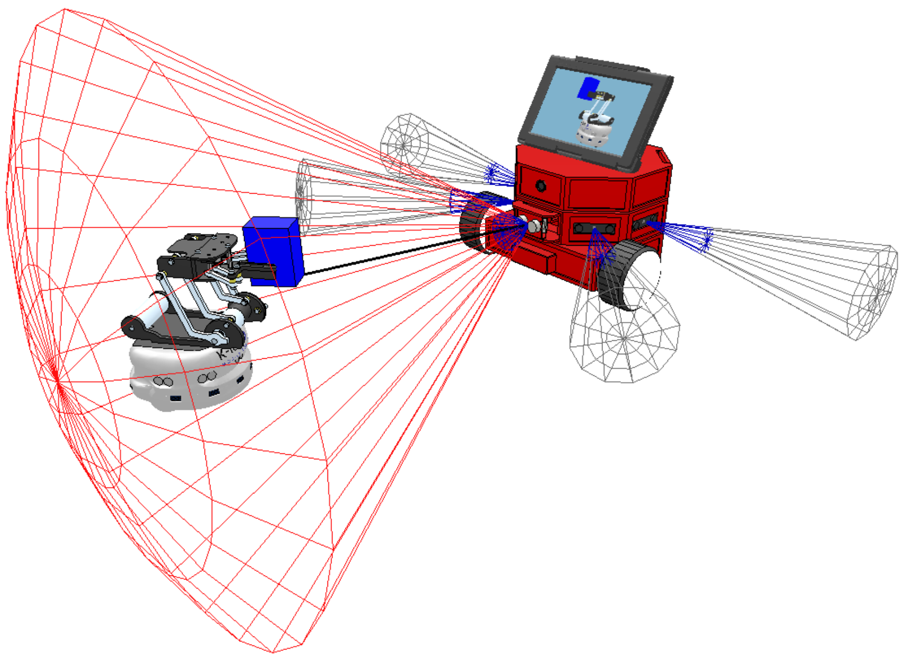

Fig. 1. Mobile robot (right) equipped with 5 *proximity sensors* and one *rendering sensor*. The *rendering sensor* is not used for detection is this case, but for texture generation for the LCD panel. (The left robot model is courtesy of K-Team Corp., the right robot model is courtesy of Cubictek Corp. and NT Research Corp).

2.2 Calculation Modules

Scene objects are rarely used on their own, they rather operate on (or in conjunction with) other *scene objects* (e.g. a *proximity sensor* will detect *shapes* or *dummies* that intersect with its detection volume). In addition, V-REP has several calculation modules that can directly operate on one or several *scene objects*. Following are V-REP's main calculation modules:

– **Forward and inverse kinematics module**: allows kinematics calculations for any type of mechanism (branched, closed, redundant, containing nested loops, etc.). The module is based on calculation of the damped least squares pseudoinverse [7]. It supports conditional solving, damped and weighted resolution, and obstacle avoidance based contraints.
– **Dynamics or physics module**: allows handling rigid body dynamics calculation and interaction (collision response, grasping, etc.) via the Bullet Physics Library [1].
– **Path planning module**: allows holonomic path planning tasks and nonholonomic path planning tasks (for car-like vehicles) via an approach derived from the *Rapidly-exploring Random Tree (RRT)* algorithm[6].

- **Collision detection module**: allows fast interference checking between any *shape* or collection of *shapes*. Optionally, the collision contour can also be calculated. The module uses data structures based on a binary tree of Oriented Bounding Boxes [5] for accelerations. Additional optimization is achieved with a temporal coherency caching technique.
- **Minimum distance calculation module**: allows fast minimum distance calculations between any *shape* (convex, concave, open, closed, etc.) or collection of *shapes*. The module uses the same data structures as the collision detection module. Additional optimization is also achieved with a temporal coherency caching technique.

Except for the dynamics or physics modules that directly operate on all dynamically enabled *scene objects*, other calculation modules require the definition of a calculation task or calculation object, that specifies on which *scene objects* the module should operate and how. If for example the user wishes to have the minimum distance between ***shape*** **A** and ***shape*** **B** automatically calculated and maybe also recorded, then a minimum distance object has to be defined, having as parameters ***shape*** **A** and ***shape*** **B**. Fig. 2 shows V-REP's typical simulation loop, including main *scene objects* and calculation modules.

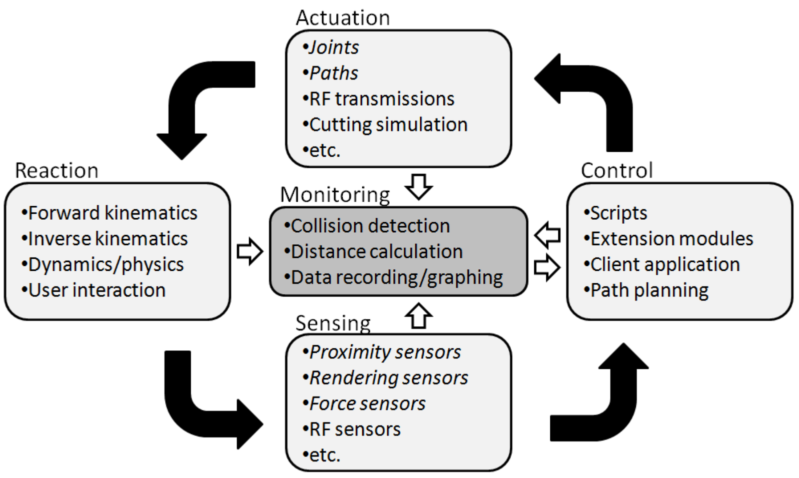

Fig. 2. Simulation loop in V-REP.

2.3 Scalability

Destruction of one or several *scene objects* can involve automatic destruction of an associated calculation object. In a similar way, duplication of one or several *scene objects* can involve automatic duplication of associated calculation objects. This also includes automatic duplication of associated control scripts (see next section). The result of this is that duplicated *scene objects* will automatically be fully functional, allowing a flexible *plug-and-play* like behaviour.

3 V-REP's Control Methodology

V-REP offers various means for controlling simulations or even to customizing the simulator itself (see Fig. 3). V-REP is wrapped in a function library, and requires a client application to run. The V-REP default client application is quite simple and takes care of loading extension modules , registers event callbacks (or message callbacks), relays them to the loaded extension modules, initializes the simulator, and handles the application and simulation loop. In addition, custom simulation functions can be added via:

- **Scripts in the *Lua* language**. *Lua* [2] is a lightweight extension programming language designed to support procedural programming. The *Lua* script interpreter is embedded in V-REP, and extended with several hundreds of V-REP specific commands. Scripts in V-REP are the main control mechanism for a simulation.
- **Extension modules to V-REP (plugins)**. Extension modules allow for registering and handling custom commands. A high-level script command (e.g., *robotMoveAndAvoidObstacles(duration)*) can be then an extension module can handle this high-level command by executing the corresponding logic and low-level API function calls in a fast and hidden fashion.

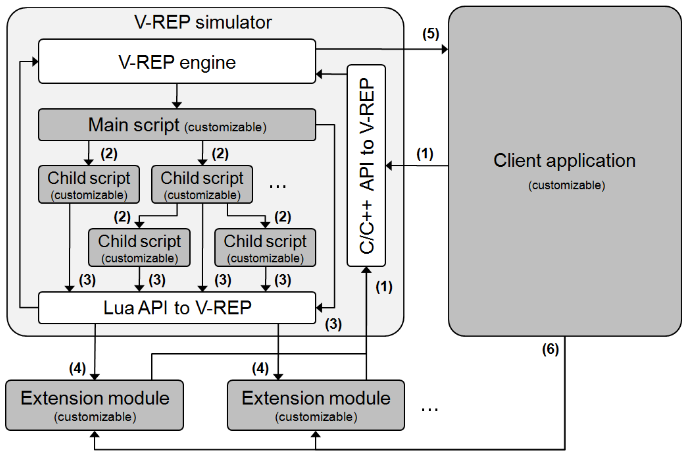
V-REP simulator
V-REP engine
Main script (customizable)
Child script (customizable)
Child script (customizable)
Child script (customizable)
Child script (customizable)
...
Lua API to V-REP
C/C++ API to V-REP
Client application (customizable)
Extension module (customizable)
Extension module (customizable)
...
(1)
(2)
(3)
(4)
(5)
(6)

Fig. 3. Control architecture in V-REP. Greyed areas can be customized by the user. (1) C/C++ API calls to V-REP from the client application, or from extension modules. (2) Script handling calls. Typically *simHandleChildScript(sim_handle_all)*. Executes all first encountered child scripts in the current hierarchy. (3) Lua API calls to V-REP from scripts. (4) Callback calls to extension modules. Originate when a script calls a custom function, previously registered by an extension module. (5) Event callback calls to the client application. (6) Relayed event calls to extension modules.

3.1 Script Calling Methodology

A simulation is handled when the client application calls a *main script*, which in turn can call *child scripts*.

Each simulation scene has exactly one *main script* that handles all default behaviour of a simulation, allowing simple simulations to run without even writing a single line of code. The *main script* is called at every simulation pass and is non-threaded.

Child scripts on the other hand are not limited in number, and are associated with (or attached to) *scene objects*. As such, they are automatically duplicated if the associated *scene object* is duplicated. In addition to that, duplicated scripts do not need any code adjustment and will automatically fetch correct object handles when accessing them. *Child scripts* can be non-threaded or threaded (i.e. launch a new thread).

The default *child script* calling methodology is hierarchial; each script is in charge of calling all first encountered *child scripts* in the current hierarchy (since *scene objects* are built in a tree-like hierarchy, scripts automatically inherit the same hierarchy). This is achieved with a single function call: *simHandleChildScript(sim_handle_all)*.

Taking the example of Fig. 4, when the *main script* calls *simHandleChildScript(sim_handle_all)*, then *child scripts* associated with objects 3, 4 and 7 will be executed. Only when the *child script* associated with object 3 in its turn calls *simHandleChildScript(sim_handle_all)*, will *child script* associated with object 6 also be executed.

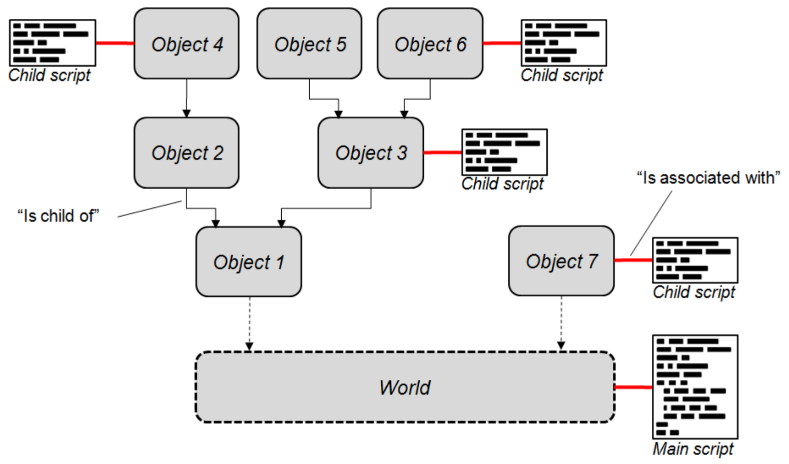

Fig. 4. *Main script* and *child scripts* in a scene.

The default *main script* will always call *simHandleChildScript(sim_handle_all)*, and *child scripts* should do the same.

3.2 Non-threaded *child script* example

Following code illustrates an empty non-threaded *child script* in V-REP:

```
if (simGetScriptExecutionCount()==0) then
   -- Initialization code comes here
end
simHandleChildScript(sim_handle_all) -- Handles child scripts
-- Main code comes here
if (simGetSimulationState()==sim_simulation_advancing_lastbeforestop) then
   -- Restoration code comes here
end
```

Non-threaded *child scripts* are "pass-through", which means that at each simulation pass they will execute, and directly return control to the caller. The caller can provide input parameters (input arguments). When a *child script* is called explicitely (i.e. *simHandleChildScript("childScriptID")* instead of *simHandleChildScript(sim_handle_all)*), then it can also return output parameters (return values).

3.3 Threaded *child script* example

Threaded *child scripts* require a slightly differentiated handling, since other *child scripts* built on top of them should also be guaranteed to be executed at each simulation pass. Following code illustrates an empty, threaded *child script* in V-REP:

```
simDelegateChildScriptExecution() -- Delegates child script execution
simSetThreadSwitchTiming(100) -- optional
-- Initialization code comes here
while (simGetSimulationState()~=sim_simulation_advancing_abouttostop) do
   -- Main code comes here
   simSwitchThread() -- optional
end
-- Restoration code comes here
```

As can be seen from above code, threaded *child scripts* should delegate their *child script* execution to the *main script* to make sure they will be called at each simulation pass. The code also shows a particularity of V-REP's threads: V-REP doesn't use *regular* threads, but rather coroutines. The advantage of this is a greater flexibility in thread execution timing, with possibility of synchronization with the *main script*. Indeed, *simSetThreadSwitchTiming* sets the time after which the thread should automatically switch to another thread. The switching can also be explicitely performed with *simSwitchThread*. Control is given back to threads each time the *main script* is about to execute.

3.4 Scalability

This distributive, hierachial script execution mechanism makes the handling of newly added (e.g. copy/pasted) *scene objects* or models very easy, since associated *child scripts* will automatically be executed, without having to adjust or modify any code. Additionally, added *child scripts* will not be executed in a random order, but according to their position within the scene hierarchy.

Extension modules to V-REP seamlessly integrate into this distributive control approach: extending V-REP with a specific robot language becomes as easy as wrapping the robot language interpreter into a V-REP extension module. A similar approach can be taken to embed emulators (e.g. microcontroller emulators) into V-REP, in order to control a simulation natively for instance.

Finally, V-REP offers sofisticated messaging mechanisms. In particular for inter-script communications; messages can be global (i.e. can be received by all *child scripts*), local (i.e. can be received only by *child scripts* in the current hierarchy), or direct (i.e. can only be received by a single specific *child script*).

4 Example Simulation Set-up

Following example scene in V-REP (see Fig. 5) clarifies several previously mentioned aspects of the simulator.

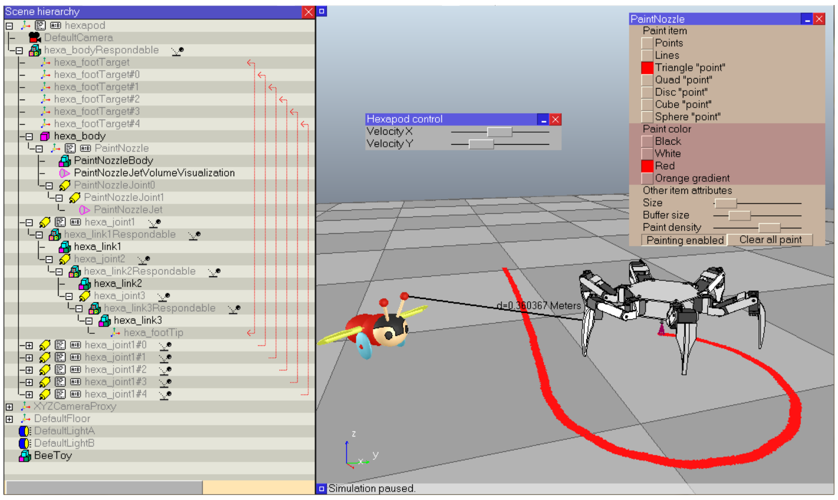

Fig. 5. V-REP example scene. The left part shows the scene hierarchy, the right part shows the scene 3D content.

The scene contains a hexapod walking robot (defined by the hierarchy tree starting at the *scene object* "hexapod"), and a little toy *scene object* ("BeeToy"). The hexapod itself is set-up with 6 identical legs rotated by 60 degrees relative to each others. For each leg, an inverse kinematics object was defined that resolves the correct leg joint positions for a given desired foot position; inverse kinematics constraints are indicated in the scene hierarchy with red arrows (e.g. "hexa_footTarget" is the desired foot position for "hexa_footTip"). The joint positions, calculated by the forward and inverse kinematics module, are then applied as target positions by the dynamics or physics module. While a *child script* associated with "hexapod" is in charge of generating a foot motion sequence, each leg has an own *child script* that will apply the generated foot motion sequence with a different delay. Except for that delay, the 6 leg *child scripts* are identical and the first one is reproduced here:

```
baseHandle=... -- Handle of the base object ("hexapod")
if (simGetScriptExecutionCount()==0) then
   -- Following is the movement delay for this leg:
   modulePos=0
   -- Retrieve various handles and prepare initial values:
   tip=simGetObjectHandle('hexa_footTip')
   target=simGetObjectHandle('hexa_footTarget')
   j1=simGetObjectHandle('hexa_joint1')
   j2=simGetObjectHandle('hexa_joint2')
   j3=simGetObjectHandle('hexa_joint3')
   simSetJointPosition(j1,0)
   simSetJointPosition(j2,-30*math.pi/180)
   simSetJointPosition(j3,120*math.pi/180)
   footOriginalPos=simGetObjectPosition(tip,baseHandle)
   isf=simGetObjectSizeFactor(baseHandle)
end
-- Read the movement data:
data=simReceiveData(0,'HEXA_x')
xMovementTable=simUnpackFloats(data)
data=simReceiveData(0,'HEXA_y')
yMovementTable=simUnpackFloats(data)
data=simReceiveData(0,'HEXA_z')
zMovementTable=simUnpackFloats(data)
data=simReceiveData(0,'HEXA_imd')
interModuleDelay=simUnpackInts(data)[1]
-- Make sure that scaling during simulation will work flawlessly:
sf=simGetObjectSizeFactor(baseHandle)
af=sf/isf
-- Apply the movement data (with the appropriate delay):
targetNewPos={
   footOriginalPos[1]*af+xMovementTable[1+modulePos*interModuleDelay]*sf,
   footOriginalPos[2]*af+yMovementTable[1+modulePos*interModuleDelay]*sf,
   footOriginalPos[3]*af+zMovementTable[1+modulePos*interModuleDelay]*sf}
-- The IK module will automatically have the foot tip follow the "target",
-- so we just need to set the position of the "target" object:
simSetObjectPosition(target,baseHandle,targetNewPos)
-- Make sure that any attached child script will also be executed:
simHandleChildScript(sim_handle_all_except_explicit)
```

A minimum distance object between all *scene objects* composing the hexapod and all other *scene objects* was defined so that the hexapod clearance can be tracked. A paint nozzle model was attached to the hexapod and allows marking the floor with color. The paint nozzle model operates independently from the hexapod robot and is handled by its own *child script*. The *scene objects* "hexapod" and "PaintNozzle" are associated with 2 custom dialogs that allow users to interact with a script's behaviour. This represents another powerful feature in V-REP: an unlimited number of custom dialogs can be defined and associated with *scene objects*. They are destroyed or duplicated in a similar way as calculation objects (refer to section 2.3).

5 Conclusion

V-REP demonstrates a modular simulation architecture combined with a distributed control mechanism. This results in a versatile and scalable framework that fits the simulation needs of complex robotic systems with several forms of asynchronous interaction.

V-REP provides a balanced functionality through a multitude of additional calculation modules, offering a real advantage in terms of simulation fidelity and simulation completeness (see Fig. 6) .

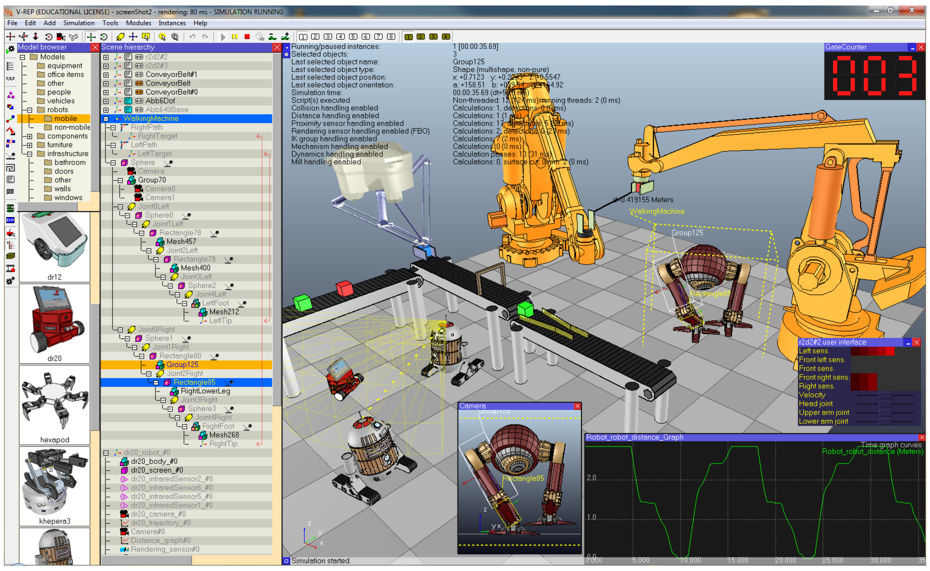

Fig. 6. Screen shot of V-REP's application main window (robot models are courtesy of Lyall Randell, Cubictek Corp., NT Research Corp., and ABB Corp. there is no link of any kind between V-REP and ABB Corp.)

One central aspect further enabling V-REP's versatility and scalability is its distributed control approach; it is possible to test configurations with 2, 3, or up to several hundreds of (identical or different) robots (e.g. in a swarm configuration), by simple drag-and-drop or copy-and-paste actions. There is no need for code adjustment, not even a supervisor program is required. V-REP's versatility and scalability allows for this *plug-and-play* like behavior.

Finally, V-REP's expandability through extension modules (plugins), gives the user an easy and fast way to customize a simulation, or the simulator itself.

References

1. Bullet physics library, `http://www.bulletphysics.org`
2. Lua, `http://www.lua.org`
3. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., Yoon, W.: RT-component object model in RT-middleware - distributed component middleware for RT (robot technology). In: 2005 IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA2005). pp. 457–462. Espoo, Finland (June 2005)
4. Freese, M., Ozaki, F., Matsuhira, N.: Collision detection, distance calculation and proximity sensor simulation using oriented bounding box trees. In: 4th International Conference on Advanced Mechatronics. pp. 13–18. Asahikawa, Japan (October 2004)
5. Gottschalk, S., Lin, M.C., Manocha, D.: OBB-tree : a hierarchical structure for rapid interference detection. In: ACM SIGGRAPH. pp. 171–180. New Orleans, USA (October 1996)
6. Kuffner Jr., J.J.: RRT-connect: an efficient approach to single-query path planning. In: IEEE International Conference on Robotics and Automation. pp. 995–1001. San Fransisco, USA (April 2000)
7. Wampler, C.W.: Manipulator inverse kinematic solutions based on vector formulations and damped least squares methods. IEEE Trans. Syst., Man, Cybern. 16(1), 93–101 (1986)