робосимулятор

V-REP

СГАУ кафедра АСЭУ

© <dponyatov@gmail.com>

9 января 2016 г.

Оглавление

Введение

[1]

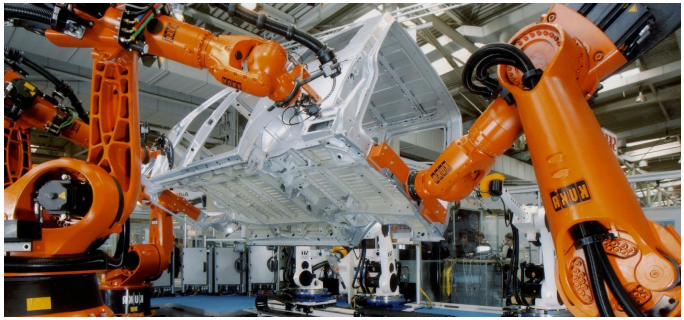
Программирование роботов — это интересно.

Многие наверное видели японских гуманоидных роботов, или французский учебный робот NAO, интересным выглядит проект обучаемого робота-манипулятор Baxter. Промышленные манипуляторы KUKA из Германии — это классика. Кто-то программирует системы конвейерной обработки (фильтрации, сортировки). Дельта роботы. Есть целый пласт — управление квадрокоптером/алгоритмы стабилизации. И конечно же простые трудяги на складе — Line Follower. *И самый доступный для самостоятельного изготовления, и в то же время полезный вариант — **универсальные роботележки** на колесном или гусеничном ходу, с размерами от настольного микробота размером с мышку, до танков и карьерных робоэлектровозов.*

[1] © [2]

NAO

Вахтёр



KUKA

Но всё это как правило — не дешевые игрушки, поэтому доступ к роботам есть в специализированных лабораториях или институтах/школах где получили финансирование и есть эти направления. Всем же остальным разработчикам (кому интересна робототехника) — остаётся завистливо смотреть.

Некоторое время назад я вышел на достаточно интересную систему — 3D робосимулятор **V-REP**, от швейцарской компании Coppelia Robotics.

К своему (приятному) удивлению я обнаружил, что эта система:

- имеет большой функционал[2]
- полностью open-source[3]
- кроссплатформенная — $\boxplus Windows$, mac, $Linux$[4]
- имеет API и библиотеки для работы с роботами через C/C^{++}, $Python$, Java, Lua, Matlab, Octave или Urbi
- бесплатная для некоммерческого использования

Все объекты, которые программируются в этой системе — "живут" в реальном с точки зрения физических законов мире — есть гравитация, можно захватывать предметы, столкновения, датчики расстояния, видео датчики и т.п.

---

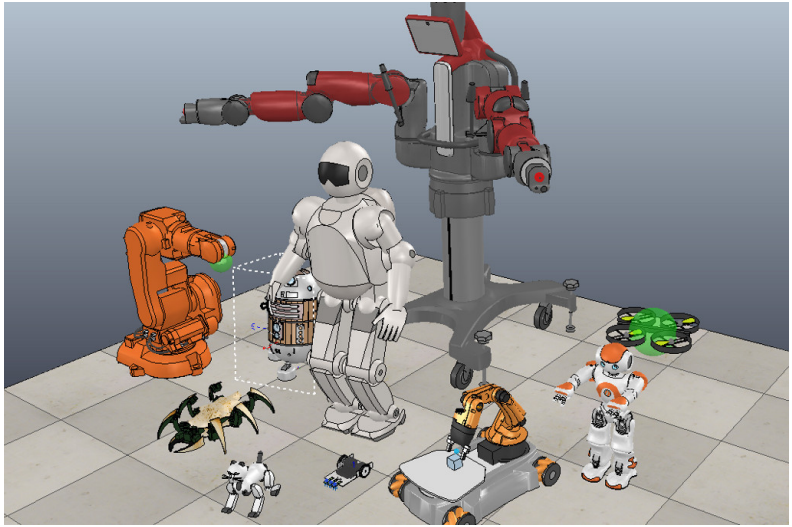
[2] система разрабатывается с марта 2010 года

[3] выложена в открытый доступ в 2013 году

[4] рализована на фреймворке Qt

Поработав некоторое время с этой системой, я решил рассказать про неё читателям хабра.

Да, и на картинке скриншот из **V-REP**, и модели роботов — которые вы можете программировать, и смотреть поведение, прямо на вашем компьютере.



Установка

V-REP скачайте и установите[5] версию Pro Edu

MinGW для сборки плагинов и интерпретатора bI нужен компилятор C^{++}. Сам **V-REP** собран с помощью GNU G++ (MinGW), поэтому написание плагинов с использованием Visual C не поддерживается.

Скачайте инсталлятор MinGW и установите пакеты **g++**, **flex** и **bison**.

[5] Next Next Next

mingw-get-setup.exe » Install » Directory » **C:/MinGW**

☒ ... also install support for the GUI | ☒ ... in the start menu | ☐ ... on the desktop

Continue » Continue

Basic Setup

☒ **mingw32-base** » ◁ » Mark for Installation отладчик **gdb** и утилита сборки проекта **make**.

☒ **mingw32-gcc-g++** компиляторы **gcc**[6] и **g++**[7], **binutils**[8]

All Packages » MSYS » MinGW Developer Toolkit

☒ **msys-bison.bin** генератор синтаксических анализаторов **bison**[9] 4.3

☒ **msys-flex.bin** генератор лексеров **flex**[10] 4.2

Installation » Apply Changes » Apply » Close

Для запуска утилит нужно добавить пути поиска в системную переменную PATH

⊞ » Компьютер » ▷ » Свойства » Дополнительные параметры » Переменные среды

Переменные среды пользователя...

```
PATH = C:\MinGW\msys\1.0\bin;C:\MinGW\bin;...
```

LLVM фреймворк для разработки компиляторов, трансляторов, статических анализаторов и оптимизаторов кода, а **clang** — компилятор Си/C^{++} на его основе. Если вы хотите ускорить интерпретатор вашего скриптового языка, добавив в него динамическую компиляция в машинный код, установка, сборка и установка LLVM подробно рассмотрена в [3].

---

[6] чистый Си

[7] C^{++}

[8] линкер, ассемблер i386, утилиты работы с объектными файлами .elf

[9] yacc

[10] lex

Обновление

MinGW

⊞ > Все программы > MinGW Installation Manager

Installation > Update Catalogue > Close

Installation > Mark All Upgrades

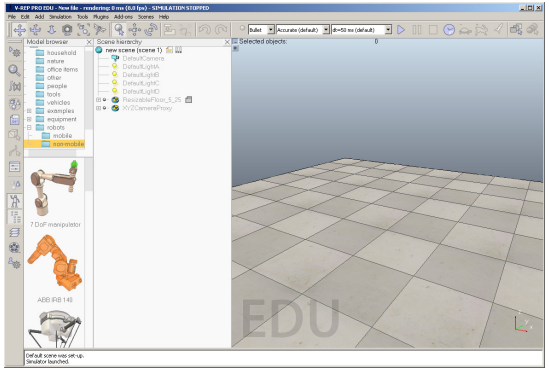
Installation > Apply Changes

Глава 1

Базовые уроки

1.1 Урок habr57 [2]

Старт

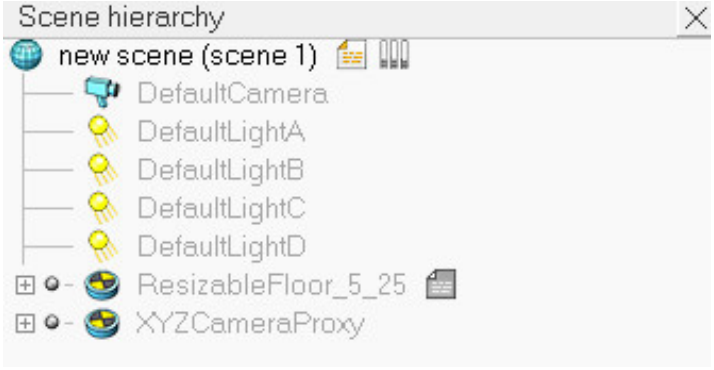
После установки, и старта мы увидим экран: 

Здесь мы видим следующие объекты:

- сцена — здесь и происходит всё действо, на данный момент она пуста[1]
- слева видим блок с библиотекой моделей — сверху папки, и под ней отображается содержимое выбранной папки[2]
- далее отображается иерархия мира

Иерархия включает в себя корневой объект (мир), в котором находятся все объекты.

В нашем примере это:



Видим источники света, видим объект для реализации пола[3], и группу для камер.

Есть главный объект-скрипт, контролирующий сцену и всех объектов на ней, и у каждого объекта может быть свой скрипт — внутренние скрипты реализованы на языке Lua.

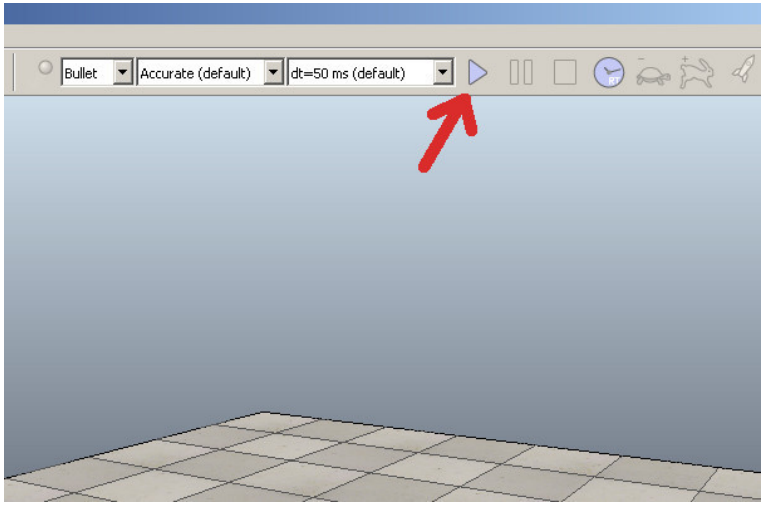
Вверху и слева мы видим toolbar — меню. Самой главной кнопкой является кнопка ▷ (**Start Simulation**) — после которой стартует симуляция сцены:

---

[1] есть только пол

[2] выбраны robots/non-mobile — то есть стационарные роботы—манипуляторы

[3] твердая поверхность с текстурой



Сценарий работы следующий:

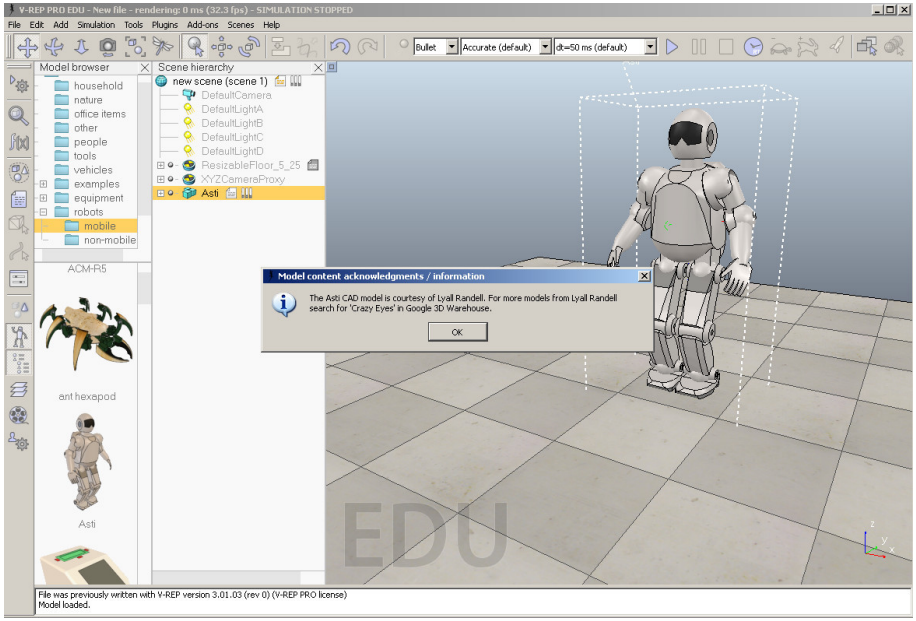
- мы перетаскиваем с помощью DragAndDrop объекты из библиотеки моделей.
- корректируем их местоположение
- настраиваем скрипты
- стартуем симулятор
- останавливаем симулятор

Попробуем что-нибудь на практике.

Быстрый старт

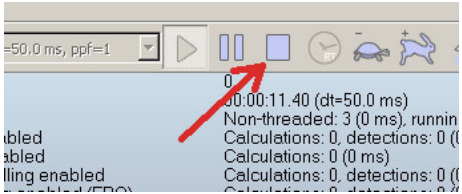
Попробуем оживить робота.

Для этого выбираем слева папку **robots/mobile** и в списке выбираем **Ansi**, захватываем, переносим на сцену и отпускаем, робот появляется на нашей сцене и появляется информация об авторе:



Теперь нажимаем на ▷ Start Simulation, и видим движение робота, и можем вручную управлять положением головы, рук[4], видео.

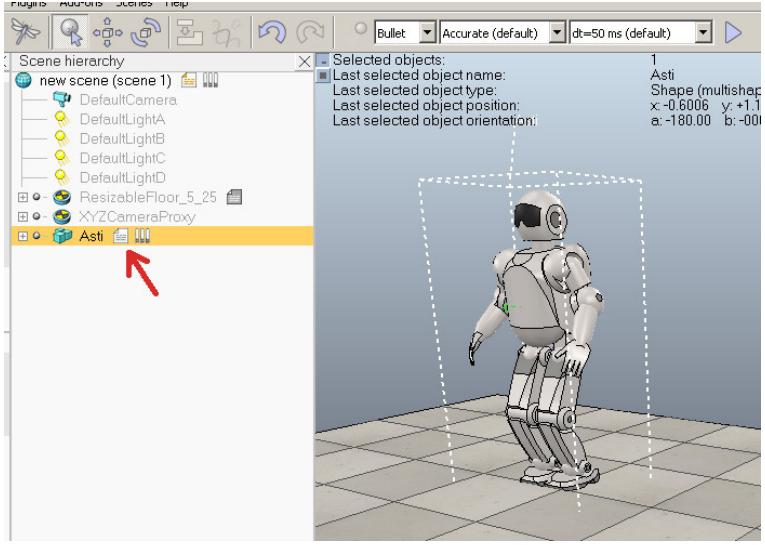
Далее останавливаем симуляцию:



[4] реализовано через панель Custom User Interface

Скрипт управления

Можем открыть и увидеть код, который[5] научил робота идти. Для этого на иерархии объектов, напротив модели **Asti**, дважды кликаем на иконке File:



Lua программа, которая осуществляет движение робота:

asti.lua

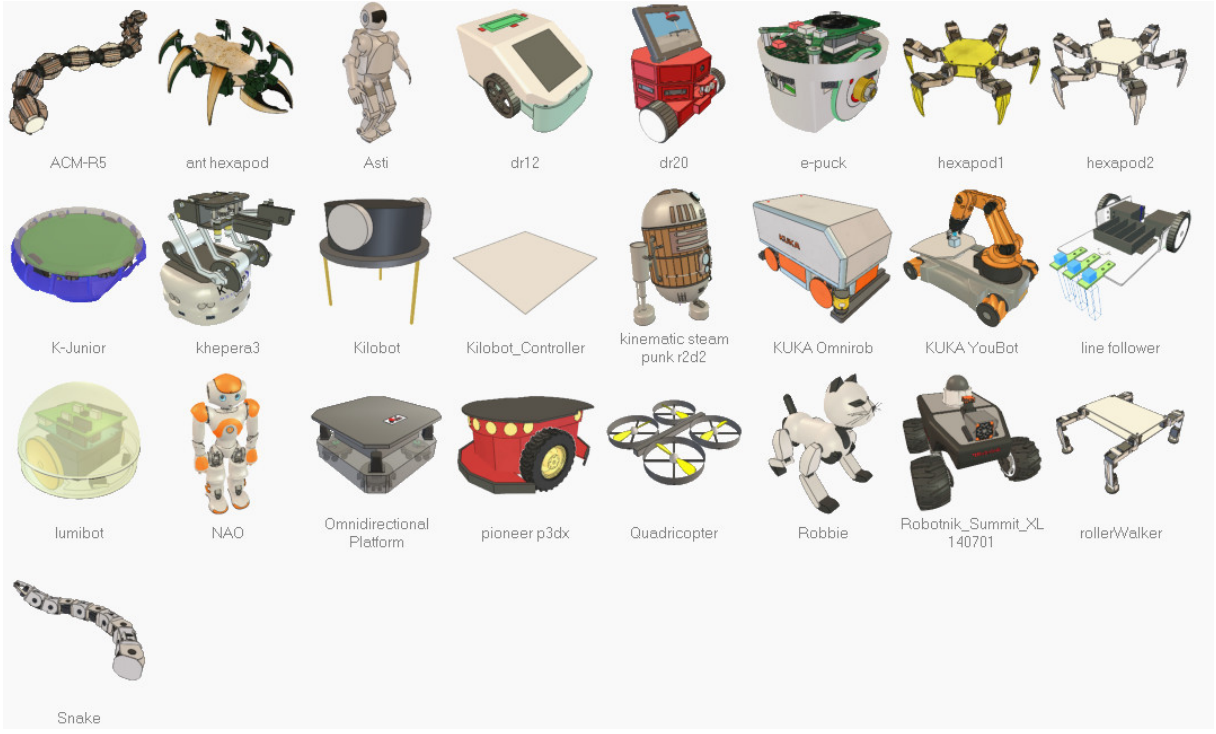
```
1 if (sim_call_type==sim_childscriptcall_initialization) then
2     asti=simGetObjectHandle("Asti")
3     lFoot=simGetObjectHandle("leftFootTarget")
4     rFoot=simGetObjectHandle("rightFootTarget")
5 ...
```

[5] управляет автономным передвижением робота

Другие модели

Вы можете удалить модель — для этого надо её выбрать на сцене или в дереве сцены, и нажать на Del. И можете попробовать посмотреть другие модели в работе, у некоторых есть скрипты для автономной работы.

Мобильные роботы



Стационарные роботы-манипуляторы

Примеры сцен

Есть большое количество примеров сцен, которые поставляются сразу с программой. Для этого надо выбрать в меню File > Open scene... и там перейти в папку **C:/Program Files/V-REP3/V-REP_PRO_EDU/scenes**.

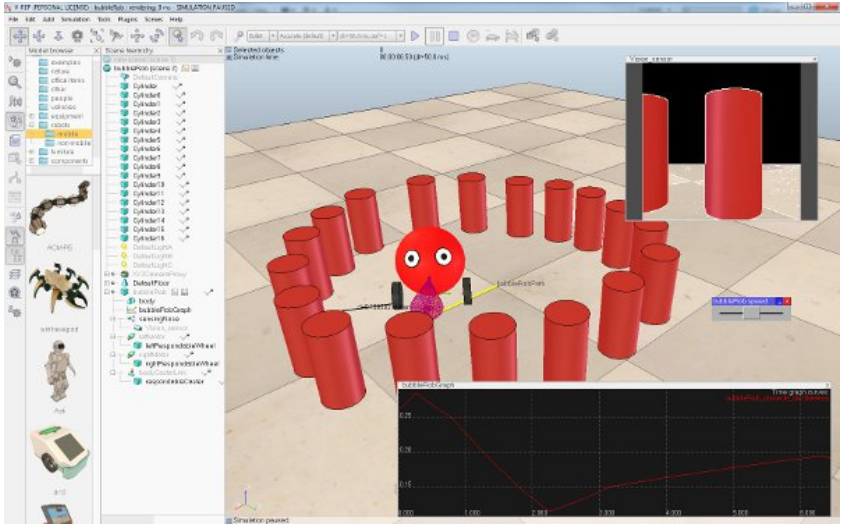
1.2 BubbleRob

[6]

This tutorial will try to introduce quite many **V-REP** functionalities while designing the simple mobile robot **BubbleRob**.

[6] © http://www.coppeliarobotics.com/helpFiles/en/bubbleRobTutorial.htm

The V-REP scene file related to this tutorial is located in V-REP's installation folder's **tutorials/BubbleRob** folder. Following figure illustrates the simulation scene that we will design:



Since this tutorial will fly over many different aspects, make sure to also have a look at the other tutorials, mainly the tutorial about building a simulation model 1.3. First of all, freshly start V-REP. The simulator displays a default scene. We will start with the body of **BubbleRob**.

We add a primitive sphere of diameter 0.2 to the scene with Menu bar > Add > Primitive shape > Sphere. We adjust the X-size item to 0.2, then click OK. The created sphere will appear in the visibility layer 1 by default, and be dynamic and respondable[7]. This means that BubbleRob's body will be falling and able to react to collisions with other respondable shapes[8]. We can see this is the shape dynamics properties: items Body is respondable and ☒ Body is dynamic are enabled. We start the simulation[9], and copy-and-paste the created sphere with Menu bar > Edit

[7] since we kept the item ☒ Create dynamic and respondable shape enabled

[8] i.e. simulated by the physics engine

[9] via the toolbar button, or by pressing Ctrl + Space in the scene window

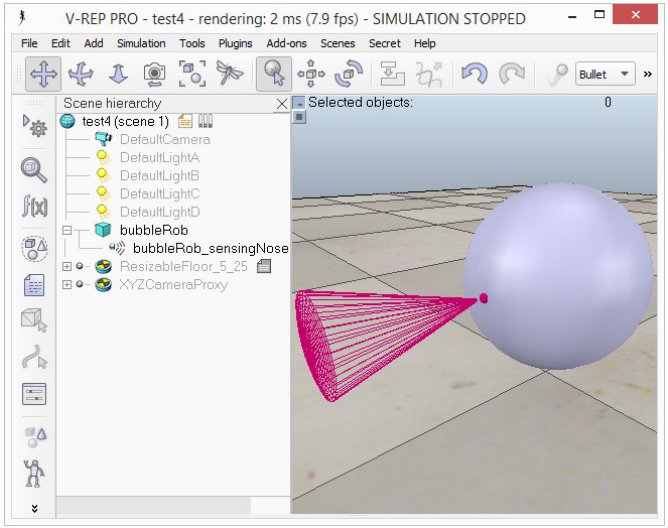
Copy selected objects then Edit Paste buffer, or with Ctrl + C then Ctrl + V: the two spheres will react to collision and roll away. We stop the simulation: the duplicated sphere will automatically be removed. This default behaviour can be modified in the simulation dialog.

We also want the BubbleRob's body to by usable by the other calculation modules (e.g. the minimum distance calculation module). For that reason, we enable Collidable, Measurable, Renderable and Detectable in the object common properties for that shape, if not already enabled. If we wanted, we could now also change the visual appearance of our sphere in the shape properties.

Now we open the position and translation dialog, select the sphere representing BubbleRob's body, and in the dialog's section Object/item translation & position scaling operations, we enter 0.02 for Along Z. We make sure that the Relative to-item is set to World. Then we click Translate selection. This translates all selected objects by 2 cm along the absolute Z-axis, and effectively lifted our sphere a little bit. In the scene hierarchy, we double-click the sphere's name, so that we can edit its name. We enter bubbleRob and press enter.

Next we will add a proximity sensor so that BubbleRob knows when it is approaching obstacles: we select Menu Add Proximity sensor Cone type. In the orientation and rotation dialog, in section Object/item rotation operations, we enter 90 for Around Y and for Around Z, then click Rotate selection. In the position and translation dialog, in section Object/item position, we enter 0.1 for X-coord. and 0.12 for Z-coord. The proximity sensor is now correctly positioned relative to BubbleRob's body. We double-click the proximity sensor's icon in the scene hierarchy to open its properties dialog. We click Show volume parameter to open the proximity sensor volume dialog. We adjust items Offset to 0.005, Angle to 30 and Range to 0.15. Then, in the proximity sensor properties, we click Show detection parameters. This opens the proximity sensor detection parameter dialog. We uncheck item Don't allow detections if distance smaller than then close that dialog again. In the scene hierarchy, we double-click the proximity sensor's name, so that we can edit its name. We enter bubbleRob_sensingNose and press enter.

We select bubbleRob_sensingNose, then control-select bubbleRob, then click Menu Edit Make last selected object paren
This attaches the sensor to the body of the robot. We could also have dragged bubbleRob_sensingNose onto bubbleRob in the scene hierarchy. This is what we now have:

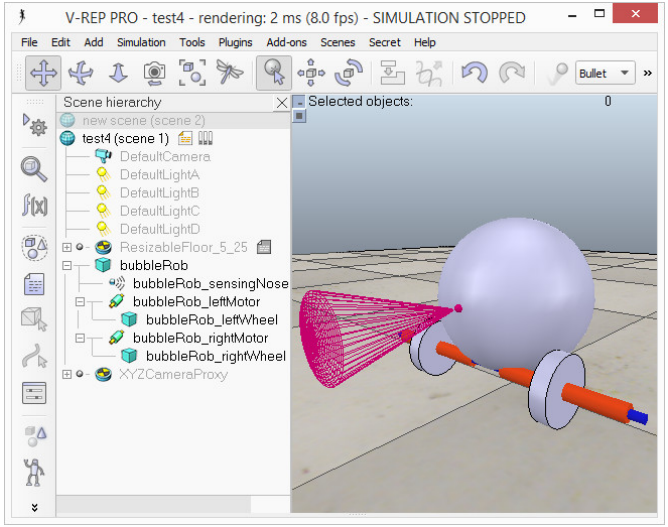


Proximity sensor attached to bubbleRob's body

Next we will take care of BubbleRob's wheels. We create a new scene with Menu › File › New scene. It is often very convenient to work across several scenes, in order to visualize and work only on specific elements. We add a pure primitive cylinder with dimensions (0.08,0.08,0.02). As for the body of BubbleRob, we enable Collidable, Measurable, Renderable and Detectable in the object common properties for that cylinder, if not already enabled. Then we set the cylinder's absolute position to (0.05,0.1,0.04) and its absolute orientation to (-90,0,0) (we can do this in the Coordinates and transformations dialog). We change the name to bubbleRob_leftWheel. We copy and paste the wheel, and set the absolute Y coordinate of the copy to -0.1. We rename the copy to bubbleRob_rightWheel. We select the two wheels, copy them, then switch back to scene 1, then paste the wheels.

We now need to add joints (or motors) for the wheels. We click Menu › Add › Joint › Revolute to add a revolute joint to the scene. Most of the time, when adding a new object to the scene, the object will appear at the origin of the world. We Keep the joint selected, then control-select bubbleRob_leftWheel. In the position and translation dialog, in section Object/item position, we click the Apply to selection button at the bottom of that section: this positioned the joint at

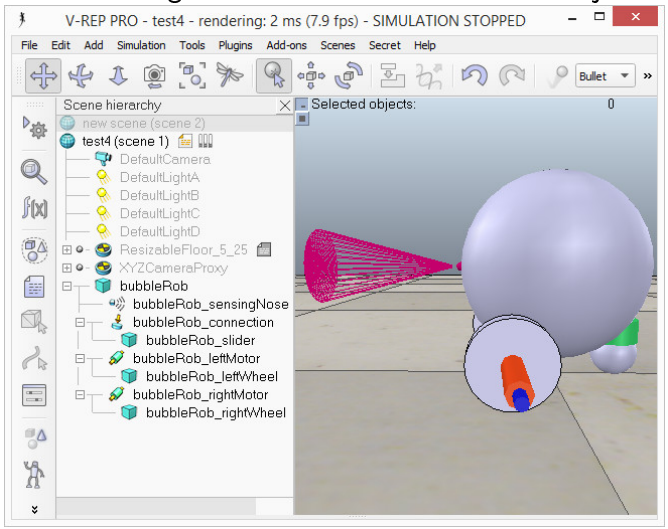
the center of the left wheel. Then, in the orientation and rotation dialog, in section Object/item orientation, we do the same: this oriented the joint in the same way as the left wheel. We rename the joint to bubbleRob_leftMotor. We now double-click the joint's icon in the scene hierarchy to open the joint properties dialog. Then we click Show dynamic parameters to open the joint dynamics properties dialog. We enable the motor, and check item Lock motor when target velocity is zero. We now repeat the same procedure for the right motor and rename it to bubbleRob_rightMotor. Now we attach the left wheel to the left motor, the right wheel to the right motor, then attach the two motors to bubbleRob. This is what we have:



Proximity sensor, motors and wheels

We run the simulation and notice that the robot is falling backwards. We are still missing a third contact point to the floor. We now add a small slider (or caster). In a new scene we and add a pure primitive sphere with diameter 0.05 and make the sphere Collidable, Measurable, Renderable and Detectable (if not already enabled), then rename it to bubbleRob_slider. We set the Material to noFrictionMaterial in the shape dynamics properties. To rigidly link the slider with the rest of the robot, we add a force sensor object with [Menu bar –> Add –> Force sensor]. We

rename it to bubbleRob_connection and shift it up by 0.05. We attach the slider to the force sensor, then copy both objects, switch back to scene 1 and paste them. We then shift the force sensor by -0.07 along the absolute X-axis, then attach it to the robot body. If we run the simulation now, we can notice that the slider is slightly moving in relation to the robot body: this is because both objects (i.e. bubbleRob_slider and bubbleRob) are colliding with each other. To avoid strange effects during dynamics simulation, we have to inform V-REP that both objects do not mutually collide, and we do this in following way: in the shape dynamics properties, for bubbleRob_slider we set the local respondable mask to 00001111, and for bubbleRob, we set the local respondable mask to 11110000. If we run the simulation again, we can notice that both objects do not interfere anymore. This is what we now have:



Proximity sensor, motors, wheels and slider

We run the simulation again and notice that BubbleRob slightly moves, even with locked motor. We also try to run the simulation with different physics engines: the result will be different. Stability of dynamic simulations is tightly linked to masses and inertias of the involved non-static shapes. For an explanation of this effect, make sure to carefully read this and that sections. We now try to correct for that undesired effect. We select the two wheels

and the slider, and in the shape dynamics dialog we click three times M=M*2 (for selection). The effect is that all selected shapes will have their masses multiplied by 8. We do the same with the inertias of the 3 selected shapes, then run the simulation again: stability has improved. In the joint dynamics dialog, we set the Target velocity to 50 for both motors. We run the simulation: BubbleRob now moves forward and eventually falls off the floor. We reset the Target velocity item to zero for both motors.

The object bubbleRob is at the base of all objects that will later form the BubbleRob model. We will define the model a little bit later. In the mean time, we want to define a collection of objects that represent BubbleRob. For that we define a collection object. We click [Menu bar –> Tools –> Collections] to open the collection dialog. Alternatively we can also open the dialog by clicking the appropriate toolbar button:

In the collection dialog, we click Add new collection. A new collection object appears in the list just below. For now the newly added collection is still empty (not defined). While the new collection item is selected in the list, select bubbleRob in the scene hierarchy, and then click Add in the collection dialog. Our collection is now defined as containing all objects of the hierarchy tree starting at the bubbleRob object (the collection's composition is displayed in the Composing elements and attributes section). To edit the collection name, we double-click it, and rename it to bubbleRob_collection. We close the collection dialog.

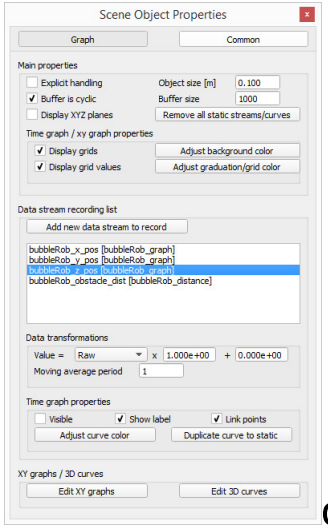
At this stage we want to be able to track the minimum distance between BubbleRob and any other object. For that, we open the distance dialog with [Menu bar –> Tools –> Calculation module properties]. Alternatively we can also open the calculation module properties dialog with the appropriate toolbar button:

In the distance dialog, we click Add new distance object and select a distance pair: [collection] bubbleRob_collection - all other measurable objects in the scene. This just added a distance object that will measure the smallest distance between collection bubbleRob_collection (i.e. any measurable object in that collection) and any other measurable

object in the scene. We rename the distance object to bubbleRob_distance with a double-click in its name. We close the distance dialog. When we now run the simulation, we won't see any difference, since the distance object will try to measure (and display) the smallest distance segment between BubbleRob and any other measurable object in the scene. The problem is that at this stage there is no other measurable object in the scene (the shape defining the floor has its measurable property turned off by default). At a later stage in this tutorial, we will add obstacles to our scene.

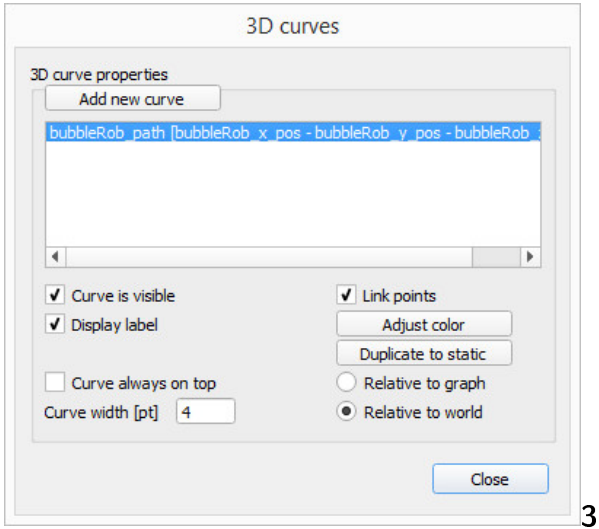
Next we are going to add a graph object to BubbleRob in order to display above smallest distance, but also BubbleRob's trajectory over time. We click [Menu bar –> Add –> Graph] and rename it to bubbleRob_graph. We attach the graph to bubbleRob, and set the graph's absolute coordinates to (0,0,0.005). Now we open the graph properties dialog by double-clicking its icon in the scene hierarchy. We uncheck Display XYZ-planes, then click Add new data stream to record and select Object: absolute x-position for the Data stream type, and bubbleRob_graph for the Object / item to record. An item has appeared in the Data stream recording list. That item is a data stream of bubbleRob_graph's absolute x-coordinate (i.e. the bubbleRobGraph's object absolute x position will be recorded). Now we also want to record the y and z positions: we add those data streams in a similar way as above. We now have 3 data streams that represent BubbleRob's x-, y- and z-trajectories. We are going to add one more data stream so that we are able to track the minimum distance between our robot and its environment: we click Add new data stream to record and select Distance: segment length for the Data stream type, and bubbleRob_distance for the Object / item to record. In the Data stream recording list, we now rename Data to bubbleRob_x_pos, Data0 to bubbleRob_y_pos, Data1 to bubbleRob_z_pos, and Data2 to bubbleRob_obstacle_dist.

We select bubbleRob_x_pos in the Data Stream recording list and in the Time graph properties section, uncheck Visible. We do the same for bubbleRob_y_pos and bubbleRob_z_pos. By doing so, only the bubbleRob_obstacle_dist data stream will be visible in a time graph. Following is what we should have:



Graph properties

Next we will set-up a 3D curve that displays BubbleRob's trajectory: we click Edit 3D curves to open the XY graph and 3D curve dialog, then click Add new curve. In the dialog that pops open, we select bubbleRob_x_pos for the X-value item, bubbleRob_y_pos for the Y-value item and bubbleRob_z_pos for the Z-value item. We rename the newly added curve from Curve to bubbleRob_path. Finally, we check the Relative to world item and set Curve width to 4:

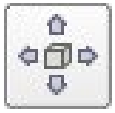


3D curve properties

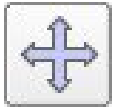
We close all dialogs related to graphs. Now we set one motor target velocity to 50, run the simulation, and will see BubbleRob's trajectory displayed in the scene. We then stop the simulation and reset the motor target velocity to zero.

Next, we will add a visualization window for the minimum distance data stream with [Menu bar menu –> Add –> Floating view]. We select bubbleRob_graph, then in the floating view, right-click [Popup menu –> View –> Associate view with selected graph]. Running the simulation will not yet display anything in the graph window, since there are still no objects (i.e. obstacles) to measure against. Let's now add some obstacles!

We add a pure primitive cylinder with following dimensions: (0.1, 0.1, 0.2). We want this cylinder to be static (i.e. not influenced by gravity or collisions) but still exerting some collision responses on non-static respondable shapes. For this, we disable Body is dynamic in the shape dynamics properties. We also want our cylinder to be Collidable, Measurable, Renderable and Detectable. We do this in the object common properties. Now, while the cylinder is still selected, we click the object translation toolbar button:

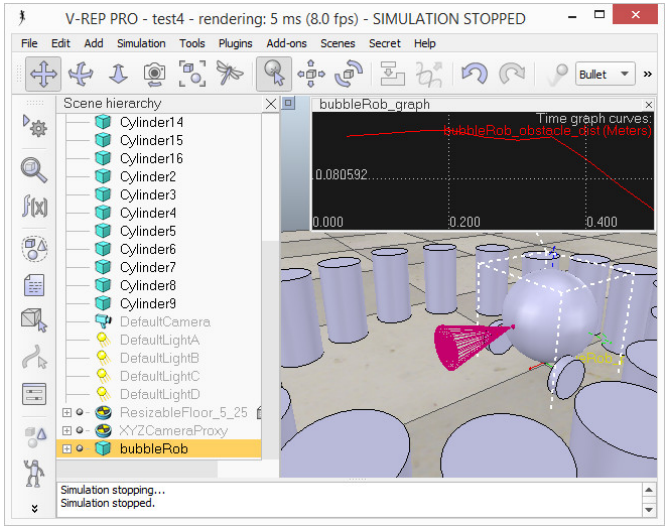


Now we can drag any point in the scene: the cylinder will follow the movement while always being constrained to keep the same Z-coordinate. We copy and paste the cylinder a few times, and move them to positions around BubbleRob (it is most convenient to perform that while looking at the scene from the top). During object shifting, holding down the shift key allows to perform smaller shift steps. Holding down the ctrl key allows to move in an orthogonal direction to the regular direction(s). When done, select the camera pan toolbar button again:



We set a target velocity of 50 for the left motor and run the simulation: the graph view now displays the distance to the closest obstacle and the distance segment is visible in the scene too. We stop the simulation and reset the target velocity to zero.

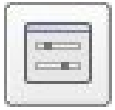
We now need to finish BubbleRob as a model definition. We select the model base (i.e. object bubbleRob) then check items Object is model base and Object/model can transfer or accept DNA in the object common properties: there is now a stippled bounding box that encompasses all objects in the model hierarchy. We select the two joints, the proximity sensor and the graph, then enable item Don't show as inside model selection and click Apply to selection, in the same dialog: the model bounding box now ignores the two joints and the proximity sensor. Still in the same dialog, we disable camera visibility layer 2, and enable camera visibility layer 10 for the two joints and the force sensor: this effectively hides the two joints and the force sensor, since layers 9-16 are disabled by default. At any time we can modify the visibility layers for the whole scene. To finish the model definition, we select the vision sensor, the two wheels, the slider, and the graph, then enable item Select base of model instead: if we now try to select an object in our model in the scene, the whole model will be selected instead, which is a convenient way to handle and manipulate the whole model as a single object. Additionally, this protects the model against inadvertant modification. Individual objects in the model can still be selected in the scene by click-selecting them with control-shift, or normally selecting them in the scene hierarchy. We finally collapse the model tree in the scene hierarchy. This is what we have:



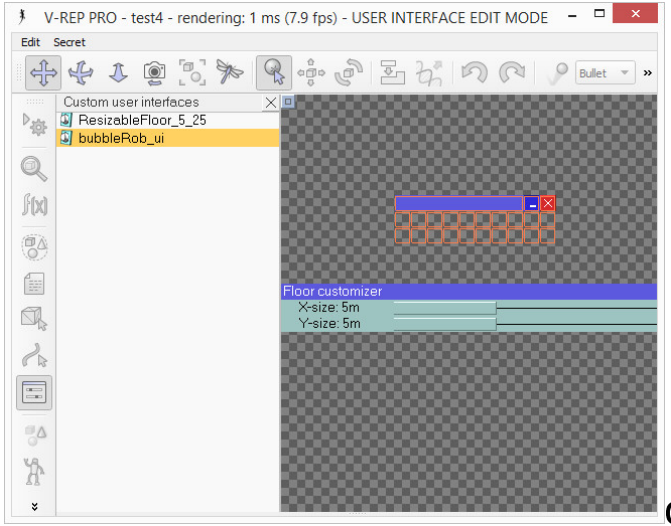
BubbleRob model definition

Next we will add a vision sensor, at the same position and orientation as BubbleRob's proximity sensor. We open the model hierarchy again, then click [Menu bar –> Add –> Vision sensor –> Perspective type], then attach the vision sensor to the proximity sensor, and set the local position and orientation of the vision sensor to (0,0,0). We also make sure the vision sensor is not not visible, not part of the model bounding box, and that if clicked, the model will be selected instead. In order to customize the vision sensor, we open its properties dialog. We set the Far clipping plane item to 1, and the Resolution x and Resolution y items to 256 and 256. We then open the vision sensor filter dialog by clicking Show filter dialog. We select the filter component Edge detection on work image and click Add filter. We position the newly added filter in second position (one position up, using the up button). We double-click the newly added filter component and adjust its Threshold item to 0.2, then click OK. We add a floating view to the scene, and over the newly added floating view, right-click [Popup menu –> View –> Associate view with selected vision sensor] (we make sure the vision sensor is selected during that process). To be able to see the vision sensor's image, we start the simulation, then stop it again.

Next, we want to be able to modulate BubbleRob's velocity with a custom user interface. To open the custom user interface dialog we click the appropriate toolbar button:



We click Add new user interface and enter 2 for the Client y-size item then click OK. A new custom user interface was added to the scene (in the middle of the page). We rename it to bubbleRob_ui:



Custom user interface editor

Other custom user interfaces might be visible. We now shift-select all free cells of the custom user interface, and click Insert merged button: a large button was placed over the selected cells. Notice the Button handle item when the new button is selected: 3. This number can be used at a later stage to programmatically access that button. If we wanted, we could change that handle. We then select Slider as type. The button changed to a slider. We deselect all cells, then select bubbleRob for item UI is associated with. This is important so that the custom user interface is also automatically copied (and saved) if bubbleRob is copied (or saved as a model). Finally, we enable item UI is visible

only during simulation. We leave the custom user interface edit mode by closing its dialog.

The last thing that we need for our scene is a small child script that will control BubbleRob's behavior. We select bubbleRob and click [Menu bar –> Add –> Associated child script –> Non threaded]. This just added a non-threaded child script to the scene, and associated it with bubbleRob. We can also add, remove or modify scripts via the script dialog which can be opened with [Menu bar –> Tools –> Scripts] or through the appropriate toolbar button:

We double-click the little script icon that appeared next to bubbleRob's name in the scene hierarchy: this opens the child script that we just added. We copy and paste following code into the script editor, then close it:

bubble.lua

```
1  if (sim_call_type==sim_childscriptcall_initialization) then
2      -- This is executed exactly once, the first time this script is executed
3      bubbleRobBase=simGetObjectAssociatedWithScript(sim_handle_self) -- this is bubbleRob's
4      -- following is the handle of bubbleRob's associated UI (user interface):
5      ctrl=simGetUIHandle("bubbleRob_ui")
6      -- Set the title of the user interface:
7      simSetUIButtonLabel(ctrl,0,simGetObjectName(bubbleRobBase).." speed")
8      leftMotor=simGetObjectHandle("bubbleRob_leftMotor") -- Handle of the left motor
9      rightMotor=simGetObjectHandle("bubbleRob_rightMotor") -- Handle of the right motor
10     noseSensor=simGetObjectHandle("bubbleRob_sensingNose") -- Handle of the proximity sens
11     minMaxSpeed={50*math.pi/180,300*math.pi/180} -- Min and max speeds for each motor
12     backUntilTime=-1 -- Tells whether bubbleRob is in forward or backward mode
13 end
14
15 if (sim_call_type==sim_childscriptcall_actuation) then
16     -- Retrieve the desired speed from the user interface:
```

```
17      speed=minMaxSpeed[1]+(minMaxSpeed[2]-minMaxSpeed[1])*simGetUISlider(ctrl,3)/1000
18
19      result=simReadProximitySensor(noseSensor) -- Read the proximity sensor
20      -- If we detected something, we set the backward mode:
21      if (result>0) then backUntilTime=simGetSimulationTime()+4 end
22
23      if (backUntilTime<simGetSimulationTime()) then
24          -- When in forward mode, we simply move forward at the desired speed
25          simSetJointTargetVelocity(leftMotor,speed)
26          simSetJointTargetVelocity(rightMotor,speed)
27      else
28          -- When in backward mode, we simply backup in a curve at reduced speed
29          simSetJointTargetVelocity(leftMotor,-speed/2)
30          simSetJointTargetVelocity(rightMotor,-speed/8)
31      end
32  end
33
34  if (sim_call_type==sim_childscriptcall_sensing) then
35
36  end
37
38  if (sim_call_type==sim_childscriptcall_cleanup) then
39
40  end
```

We run the simulation. BubbleRob now moves forward while trying to avoid obstacles (in a very basic fashion). While the simulation is still running, change BubbleRob's velocity, and copy/paste it a few times. Also try to scale a few of them while the simulation is still running. Be aware that the minimum distance calculation functionality might be heavily slowing down the simulation, depending on the environment. You can turn that functionality on and off in

the distance dialog, by checking / unchecking the Enable all distance calculations item.

Using a script to control a robot or model is only one way of doing. V-REP offers many different ways (also combined), have a look at the external controller tutorial.

1.3 Building a clean model

10

This tutorial will guide you step-by-step into building a clean simulation model, of a robot, or any other item. This is a very important topic, maybe the most important aspect, in order to have a nice looking, fast displaying, fast simulating and stable simulation model.

To illustrate the model building process, we will be building following manipulator:

Model of robotic manipulator

[10] © http://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm

Building the visible shapes

When building a new model, first, we handle only the visual aspect of it: the dynamic aspect[11], joints, sensors, etc. will be handled at a later stage.

We could now directly create primitive shapes in **V-REP** with Menu » Add » Primitive shape » ... . When doing this, we have the option to create pure shapes, or regular shapes. Pure shape will be optimized for dynamic interaction, and also directly be dynamically enabled[12]. Primitive shapes will be simple meshes, which might not contain enough details or geometric accuracy for our application. Our other option in that case would be to import a mesh from an external application.

When importing CAD data from an external application, the most important is to make sure that the CAD model is not too heavy, i.e. doesn't contain too many triangles. This requirement is important since a heavy model will be slow in display, and also slow down various calculation modules that might be used at a later stage[13]. Following example is typically a no-go[14]:

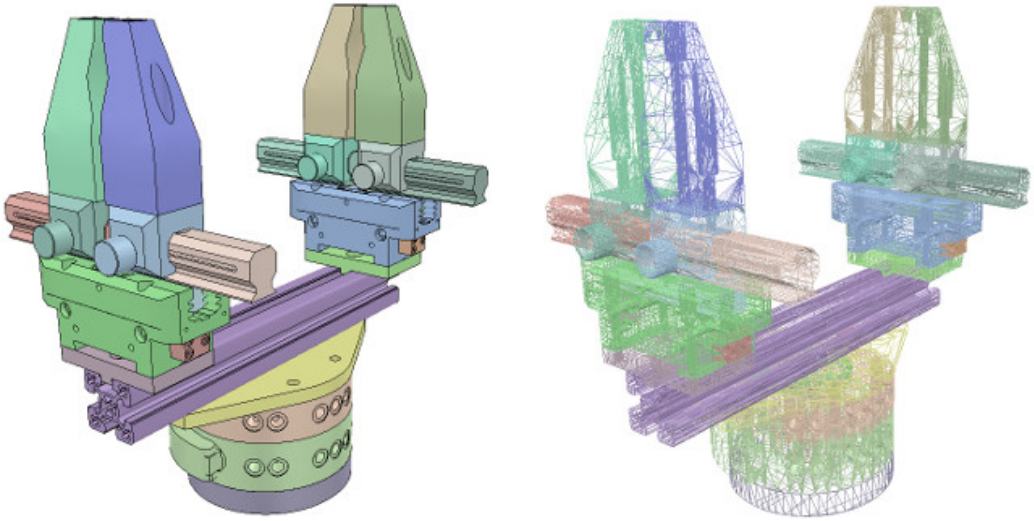
---

[11] its undelying even more simplified/optimized model

[12] i.e. fall, collide, but this can be disabled at a later stage

[13] e.g. minimum distance calculation, or dynamics

[14] even if, as we will see later, there are means to simplify the data within **V-REP**



Complex CAD data (in solid and wireframe)

Above CAD data is very heavy: it contains many triangles[15], which would be ok if we would just use a single instance of it in an empty scene. But most of the time you will want to simulate several instances of a same robot, attach various types of grippers, and maybe have those robots interact with other robots, devices, or the environment. In that case, a simulation scene can quickly become too slow. Generally, we recommend to model a robot with no more than a total of 20'000 triangles, but most of the time 5'000..10'000 triangles would just do fine as well. Remember: less is better, in almost every aspect.

What makes above model so heavy? First, models that contain holes and small details will require much more triangular faces for a correct representation. So, if possible, try to remove all the holes, screws, the inside of objects, etc. from your original model data. If you have the original model data represented as parametric surfaces/objects, then it is most of the time a simple matter of selecting the items and deleting them[16]. The second important step

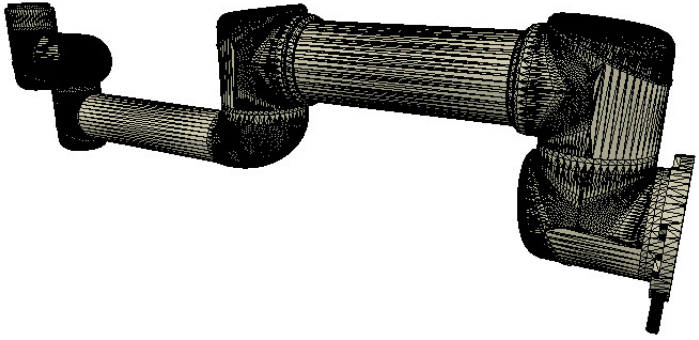
[15] more than 47'000

[16] e.g. in SolidWorks

is to export the original data with a limited precision: most CAD applications let you specify the level of details of exported meshes. It might also be important to export the objects in several steps, when the drawing consists of large and small objects; this is to avoid having large objects too precisely defined[17] and small objects too roughly defined[18]: simply export large objects first[19], then small objects[20].

V-REP supports currently following CAD data formats: **OBJ**, **STL**, **DXF**, **3DS**[21], and **Collada**. **URDF** is also supported, but not mentionned here since it is not a pure mesh-based file format.

Now suppose that we have applied all possible simplifications as described in previous section. We might still end-up with a too heavy mesh after import:



Imported CAD data

You can notice that the whole robot was imported as a single mesh. We will see later how to divide it appropriately. Notice also the wrong orientation of the imported mesh: best is to keep the orientation as it is, until the whole model was built, since, if at a later stage we want to import other items that are related to that same robot, they will automatically have the correct position/orientation relative to the original mesh.

At this stage, we have several functions at our disposal, to simplify the mesh:

[17] too many triangles
[18] too few triangles
[19] by adjusting the desired precision settings
[20] by adjusting up precision settings
[21] ⊞*Windows* only

Automatic mesh division allows to generate a new shape for all elements that are not linked together via a common edge. This does not always work for the selected mesh, but is always worth a try, since working on mesh elements gives us more control than if we had to work on all elements at the same time. The function can be accessed with Menu » Edit » Grouping/Merging » Divide selected shapes. Sometimes, a mesh will be divided more than expected. In that case, simply merge elements that logically belong together[22] back into one single shape Menu » Edit » Grouping/Merging » Merge selected shapes.

Extract the convex hull allows to simplify the mesh by transforming it into a convex hull. The function can be accessed with Menu » Edit » Morph selection into convex shapes.

Decimate the mesh allows to reduce the number of triangles contained in the mesh. The function can be accessed with Menu » Edit » Decimate selected shape....

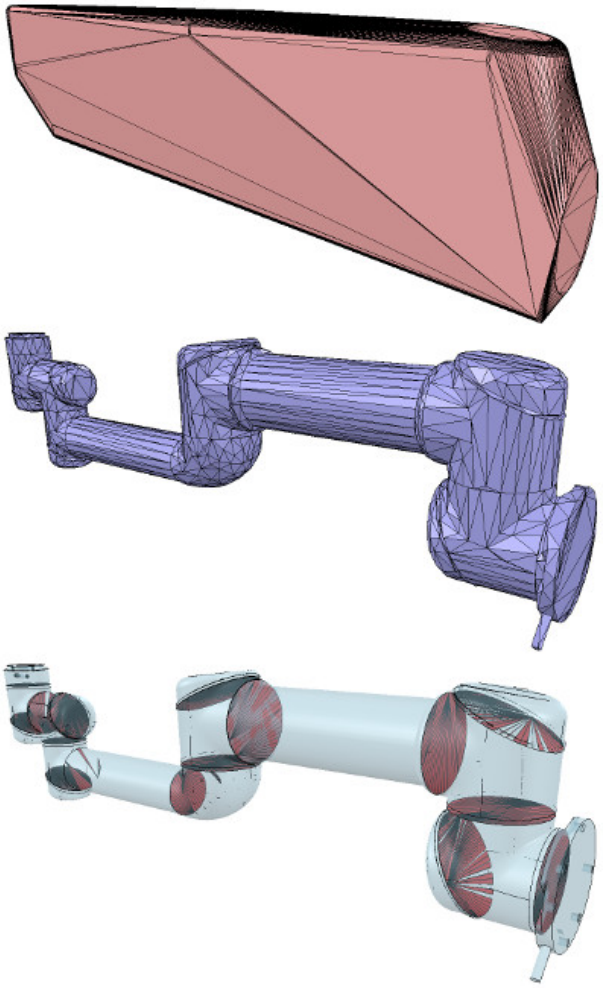
Remove the inside of the mesh allows to simplify the mesh by removing its inside. This function is based on vision sensors and might give more or less satisfying results depending on the selected settings. The function can be accessed with Menu » Edit » Extract inside of selected shape.

There is no predefined order in which above functions can/should be applied[23], it heavily depends on the geometry of the mesh we are trying to simplify. Following image illustrates above functions applied to the imported mesh[24]:

[22] i.e. that will have the same visual attributes and that are part of the same link

[23] except for the first item in the list, which should always be tried first

[24] let's suppose the first item in the list didn't work for us



Convex hull, decimated mesh, and extracted inside

Notice how the convex hull doesn't help us at this stage. We decide to use the mesh decimation function first, and run the function twice in order to divide the number of triangles by a total of 50. Once that is done, we extract the inside of the simplified shape and discard it. We end-up with a mesh containing a total of 2'660 triangles[25]. The number of triangles/vertices a shape contains can be seen in the shape geometry dialog. 2'660 triangles are extremely few triangles for a whole robot model, and the visual appearance might suffer a little bit from it.

At this stage we can start to divide the robot into separate links[26]. You can do this in two different ways:

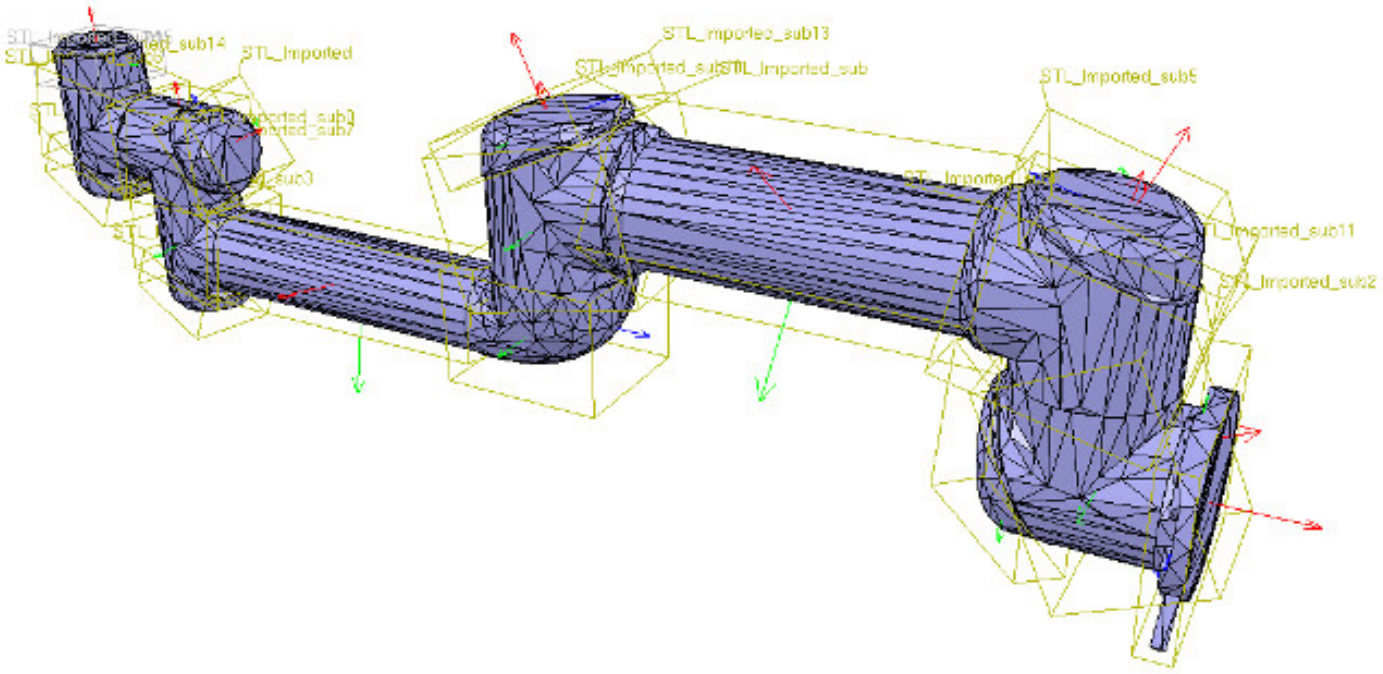
Automatic mesh division this function, which was already described in previous section, will inspect the shape and generate a new shape for all elements that are not linked together via a common edge. This does not always work, but is always worth a try. The function can be accessed with Menu > Edit > Grouping/merging > Divide selected shapes.

Manual mesh division via the the triangle edit mode, you can manually select the triangles than logically belong together, then click Extract shape. This will generate a new shape in the scene. Delete the selected triangles after that operation.

In the case of our mesh, method 1 worked fine:

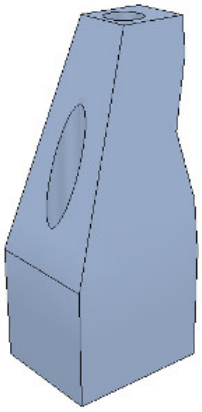
[25] the original imported mesh contained more than 136'000 triangles!

[26] remember, we currently have only a single shape for the whole robot



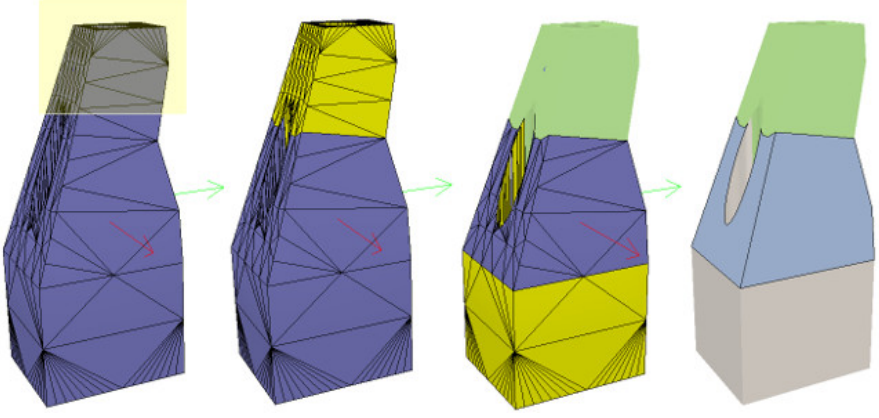
Divided mesh

Now, we could further refine/simplify individual shapes. Sometimes also, a shape might look better if its convex hull is used instead. Othertimes, you will have to use several of above's described techniques iteratively, in order to obtain the desired result. Take for instance following mesh:



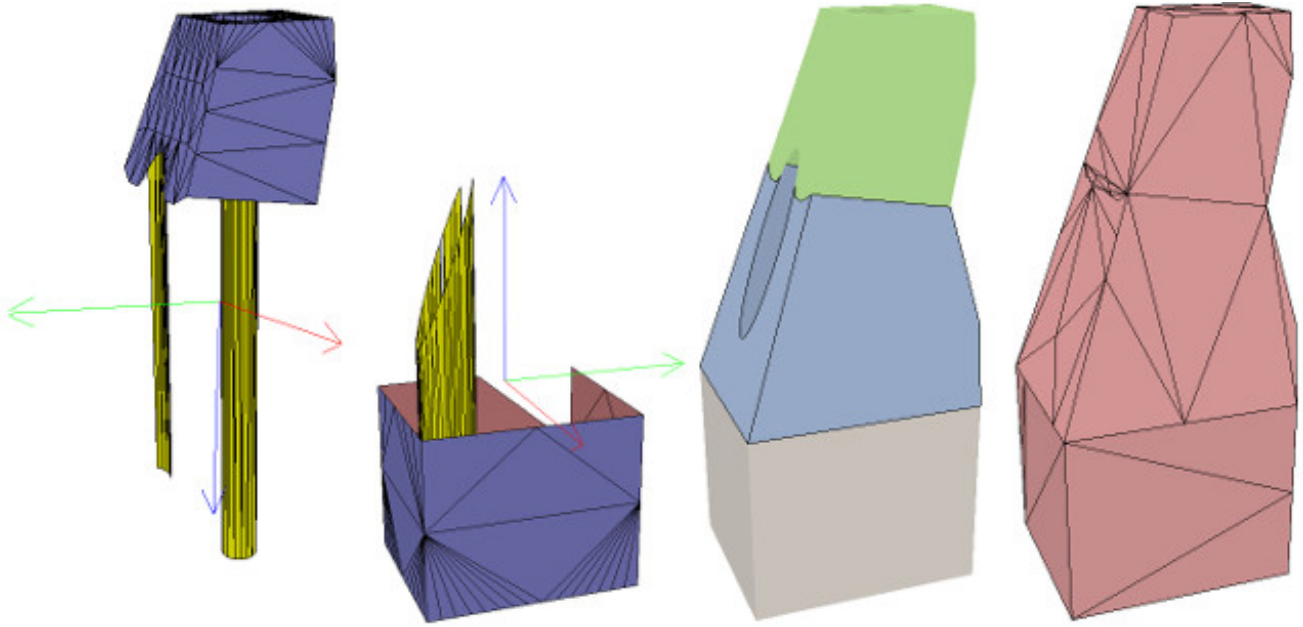
Imported mesh

The problem with above's shape is that we cannot simplify it nicely, because of the holes it contains. So we have to go the more complicated way via the shape edit mode, where we can extract individual elements that logically belong to the same convex sub-entity. This process can take several iterations: we first extract 3 approximate convex elements. For now, we ignore the triangles that are part of the two holes. While editing a shape in the shape edit mode, it can be convenient to switch the visibility layers, in order to see what is covered by other scene items.



Step 1

We end up with a toal of three shapes, but two of them will need further improvement. Now we can erase the triangles that are part of the holes. Finally, we extract the convex hull individually for the 3 shapes, then merge them back together with Menu > Edit > Grouping/Merging > merge selected shapes:

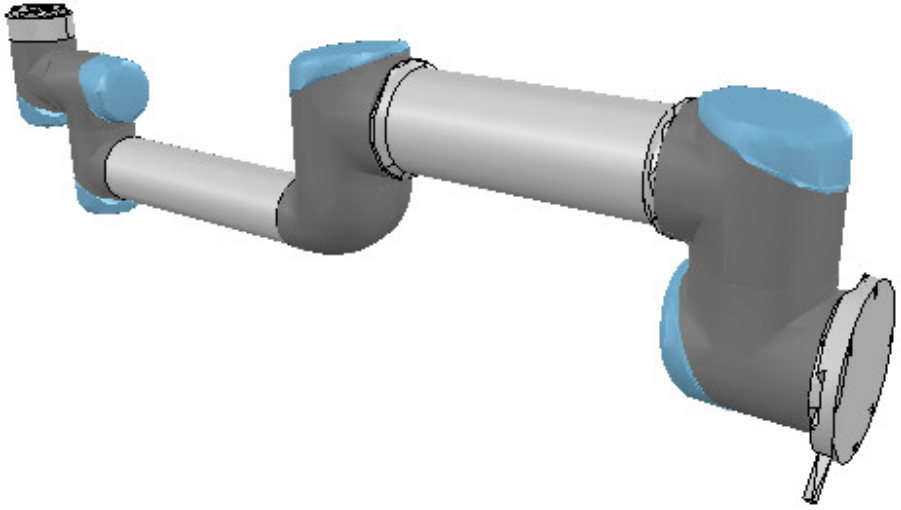


Step 2

In **V-REP**, we can enable/disable edge display for each shape. We can also specify an angle that will be taken into account for the edge display. A similar parameter is the shading angle, that dictates how facetted the shape will display. Those parameters, and a few others such as the shape color, can be adjusted in the shape properties. Remember that shapes come in various flavours. In this tutorial we have only dealt with simple shapes up to now: a simple shape has a single set of visual attributes[27]. If you merge two shapes, then the result will be a simple shape. You can also group shapes, in which case, each shape will retain its visual attributes.

[27] i.e. one color, one shading angle, etc.

In next step, we can merge elements that logically belong together[28]. Then we change the visual attributes of the various elements. The easiest ist to adjust a few shapes that have different colors and visual attributes, and if we name the color with a specific string, we can later easily programmatically change that color, also if the shape is part of a compound shape. Then, we select all the shapes that have the same visual attributes, then control-select the shape that was already adjusted, then click Apply to selection, once for the Colors, once for the other properties, in the shape properties: this transfers all visual attributes to the selected shapes[29]. We end up with 17 individual shapes:



Adjusted visual attributes

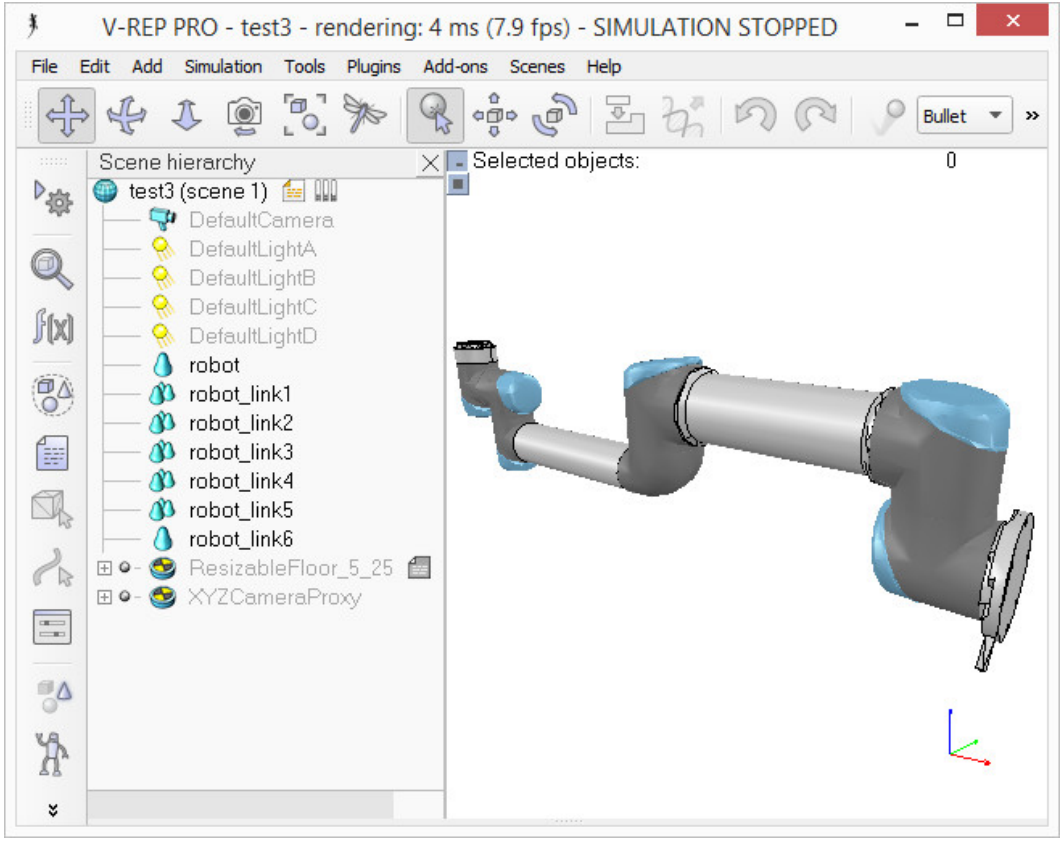
Now we can group the shapes that are part of the same link with Menu » Edit » Grouping/merging » Group selected shapes. We end up with 7 shapes: the base of the robot[30], and 6 mobile links. It is also important to correctly name your objects: you we do this with a double-click on the object name in the scene hierarchy. The base should always be the robot or model name, and the other objects should always contain the base object name, like: `robot` (base),

[28] if they are part of the same rigid element, and if they have the same visual attributes

[29] including the color name if you provided one

[30] or base of the robot's hierarchy tree

`robot_link1`, `robot_proximitySensor`, etc. By defaut, shapes will be assigned to visibility layer 1, but can be changed in the object common properties. By default, only visibility layers 1-8 are activated for the scene. We now have following (the model `ResizableFloor_5_25` was temporarily made invisible in the model properties dialog):



Individual elements compositn the robot

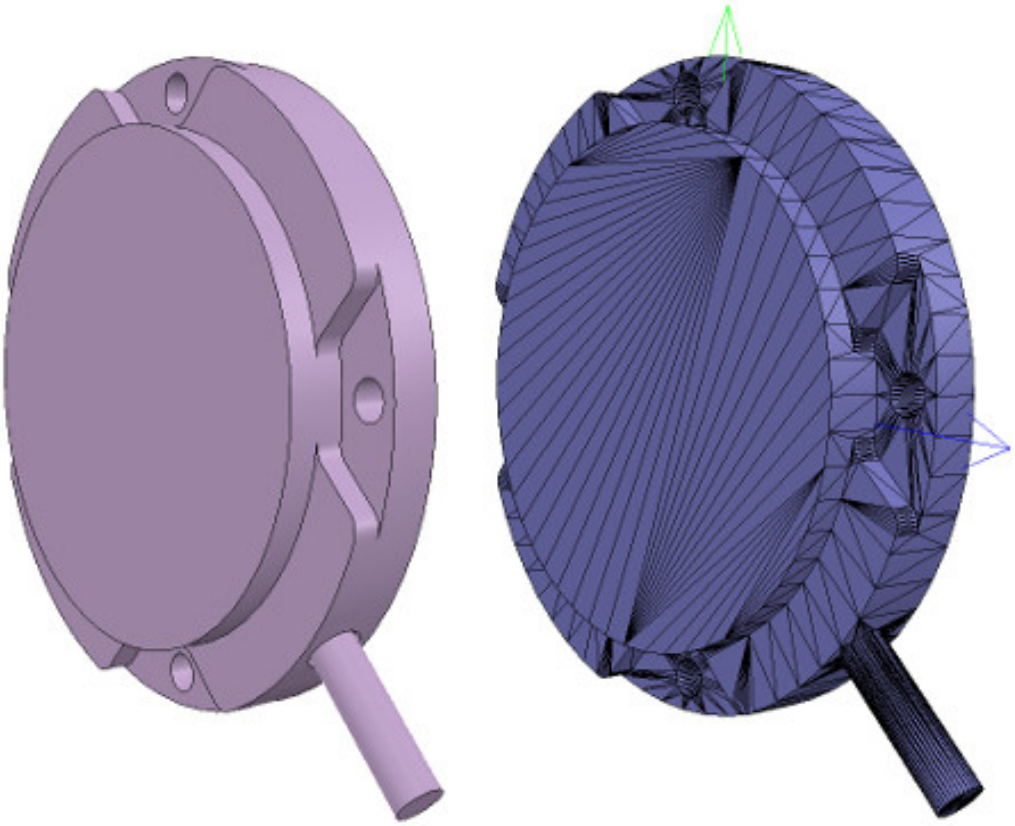
When a shape is created or modified, V-REP will automatically set its reference frame position and orientation.

A shape's reference frame will always be positioned at the shape's geometric center. The frame orientation will be selected so that the shape's bounding box remains as small as possible. This does not always look nice, but we can always reorient a shape's reference frame at any time. We now reorient the reference frames of all our created shapes with Menu > Edit > Reorient bounding box > with reference frame of world. You have more options to reorient a reference frame in the shape geometry dialog.

Building the joints

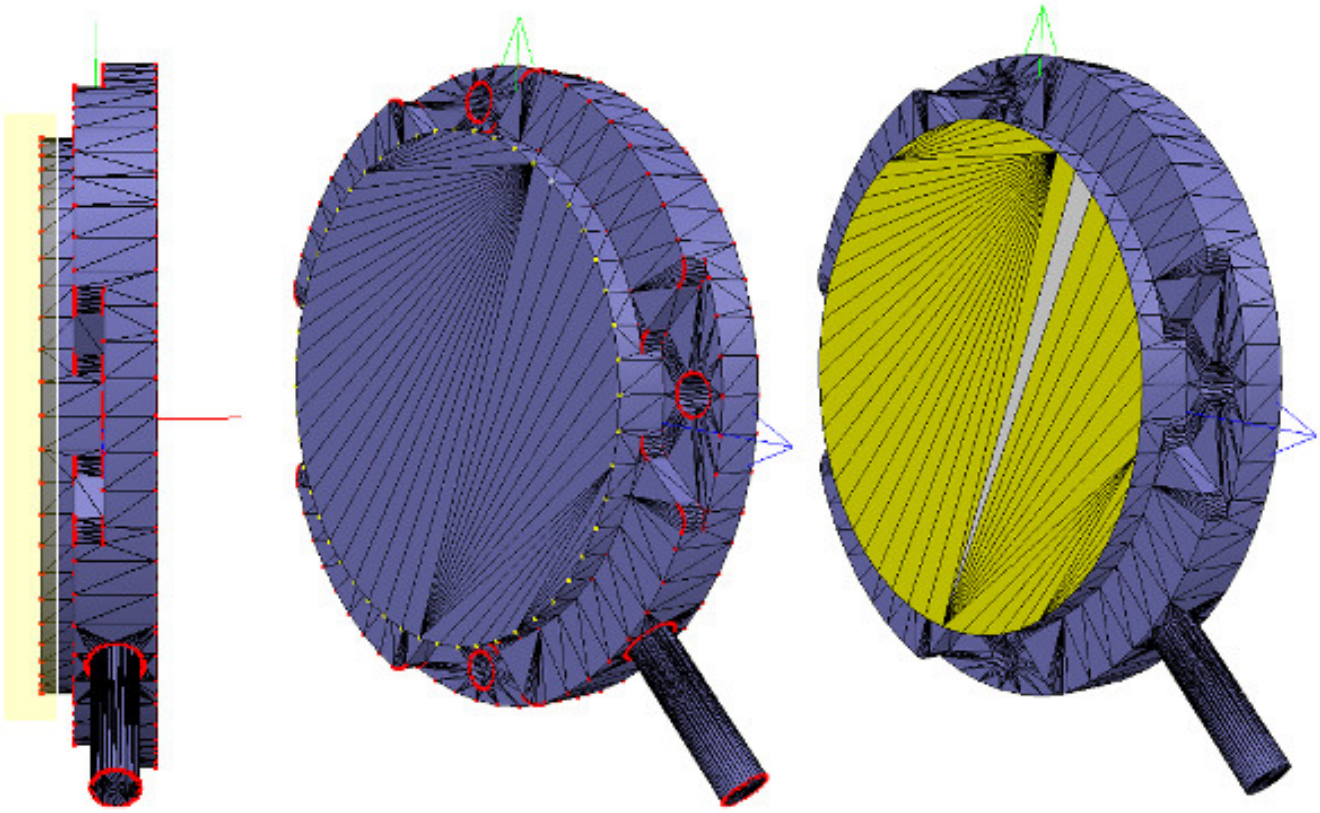
Now we will take care of the joints/motors. Most of the time, we know the exact position and orientation of each of the joints. In that case, we simply add the joints with Menu > Add > Joints > ..., then we can change their position and orientation with the coordinate and transformation dialogs. In other situations, we only have the Denavit-Hartenberg[31] parameters. In that case, we can build our joints via the tool model located in **Models/tools/Denavit-Hartenberg joint creator.ttm**, in the model browser. Othertimes, we have no information about the joint locations and orientations. Then, we need to extract them from the imported mesh. Let's suppose this is our case. Instead of working on the modified, more approximate mesh, we open a new scene, and import the original CAD data again. Most of the time, we can extract meshes or primitive shapes from the original mesh. The first step is to subdivide the original mesh. If that does not work, we do it via the triangle edit mode. Let's suppose that we could divide the original mesh. We now have smaller objects that we can inspect. We are looking for revolute shapes, that could be used as reference to create joints at their locations, with the same orientation. First, remove all objects that are not needed. It is sometimes also useful to work across several opened scenes, for easier visualization/manipulation. In our case, we focus first on the base of the robot: it contains a cylinder that has the correct position for the first joint. In the triangle edit mode, we have:

[31] i.e. D-H



Robot base: normal and triangle edit mode visualization

We change the camera view via the page selector toolbar button, in order to look at the object from the side. The fit-to-view toolbar button can come in handy to correctly frame the object in edition. Then we switch to the vertex edit mode and select all vertices that belong to the upper disc. Remember that by switching some layers on/off, we can hide other objects in the scene. Then we switch back to the triangle edit mode:

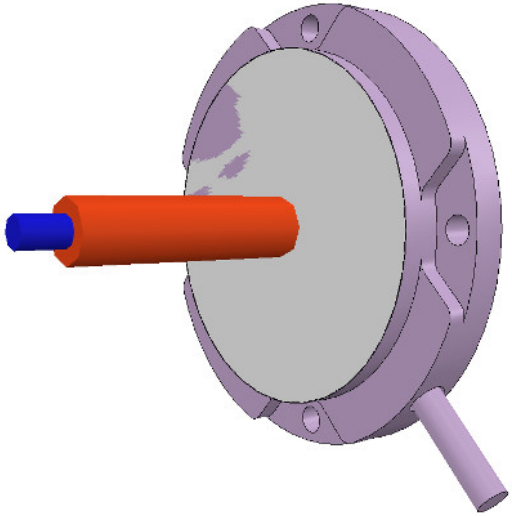


Selected upper disc, vertex edit mode (1 & 2), triangle edit mode (3)

Now we click Extract cylinder[32], this just created a cylinder shape in the scene, based on the selected triangles. We leave the edit mode and discard the changes. Now we add a revolute joint with Menu > Add > Joint > Revolute, keep it

[32] Extract shape would also work in that case

selected, then Ctrl-◁ select the extracted cylinder shape. In the coordinate and transformation dialogs we click the Position/Translation button, then, in the Object/item position section, we click Apply to selection (the one under the edit boxes): this basically copies the x/y/z position of the cylinder to the joint. Both positions are now identical. We click the Orientation/Rotation button, then in the Object/item orientation section, we click Apply to selection: the orientation of our selected objects is now also the same. Sometimes, we will need to additionally rotate the joint about 90/180 degrees around its own reference frame in order to obtain the correct orientation or rotation direction. We could do that in the Object/item rotation opertations section if needed (in that case, do not forget to click the Own frame button). In a similar way we could also shift the joint along its axis, or even do more complex operations. This is what we have:

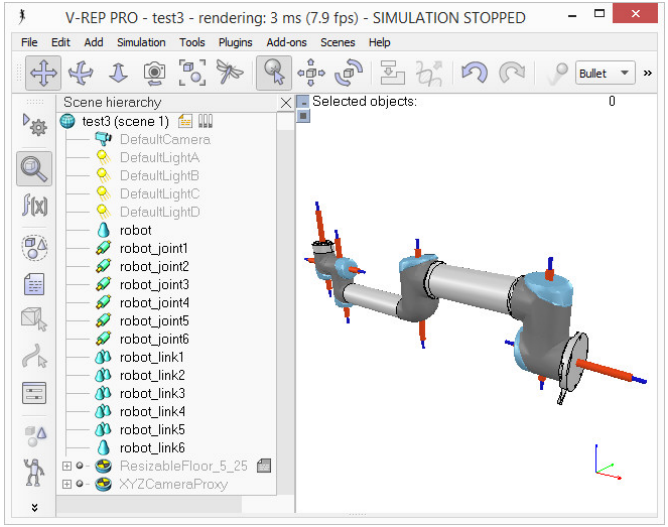


Joint in correct location, with the correct orientation

Now we copy the joint back into our original scene, and save it[33]. We repeat above procedure for all the joints in our robot, then rename them. We also make all joints a little bit longer in the joint properties, in order to see them all. By defaut, joints will be assigned to visibility layer 2, but can be changed in the object common properties. We

[33] do not forget to save your work on a regular basis! The undo/redo function is useful, but doesn't protect you against other mishaps

assign now all joints to visibility layer 10, then temporarily enable visibility layer 10 for the scene to also visualize those joints (by default, only visibility layers 1-8 are activated for the scene). This is what we have[34]:



Joints in correct configuration

At this point, we could start to build the model hierarchy and finish the model definition. But if we want opur robot to be dynamically enabled, then there is an additional intermediate step:

Building the dynamic shapes

If we want our robot to be dynamically enabled, i.e. react to collisions, fall, etc., then we need to create/configure the shapes appropriately: a shape can be:

dynamic or static a dynamic (or non-static) shape will fall and be influences by external forces/torques. A static (or non-dynamic) shape on the other hand, will stay in place, or follow the movement of its parent in the scene

[34] the model **ResizableFloor_5_25** was temporarily made invisible in the model properties dialog

hierarchy.

respondable or non-respondable a respondable shape will cause a collision reaction with other respondable shapes. They (and/or) their collider, will be influenced in their movement if they are dynamic. On the other hand, non-respondable shapes will not compute a collision response if they collide with other shapes.

Above two points are illustrated here. Respondable shapes should be as simple as possible, in order to allow for a fast and stable simulation. A physics engine will be able to simulate following 5 types of shapes with various degrees of speed and stability:

Pure shapes a pure shape will be stable and handled very efficiently by the physics engine. The draw-back is that pure shapes are limited in geometry: mostly cuboids, cylinders and spheres. If possible, use those for items that are in contact with other items for a longer time[35]. Pure shapes can be created with Menu » Add » Primitive shape.

Pure compound shapes a pure compound shape is a grouping of several pure shapes. It performs almost as well as pure shapes and shares similar properties. Pure compound shapes can be generated by grouping several pure shapes Menu » Edit » Grouping/Merging » Group selected shapes.

Convex shapes a convex shape will be a little bit less stable and take a little bit more computation time when handled by the physics engine. It allows for a more general geometry[36] than pure shapes. If possible, use convex shapes for items that are sporadically in contact with other items[37]. Convex shapes can be generated with Menu » Add » Convex hull of selection or with Menu » Edit » Morph selection into convex shapes.

Compound convex shapes, or convex decomposed shapes a convex decomposed shape is a grouping of several convex shapes. It performs almost as well as convex shapes and shares similar properties. Convex decomposed shapes can be generated by grouping several convex shapes Menu » Edit » Grouping/Merging » Group selected shapes, with Menu » Add » Convex decomposition of selection..., or with Menu » Edit » Morph selection into its convex decomposition..

[35] e.g. the feet of a humanoid robot, the base of a serial manipulator, the fingers of a gripper, etc.

[36] only requirement: it need to be convex

[37] e.g. the various links of a robot

Random shapes a random shape is a shape that is not convex nor pure. It generally has poor performance[38]. Avoid using random shapes as much as possible.

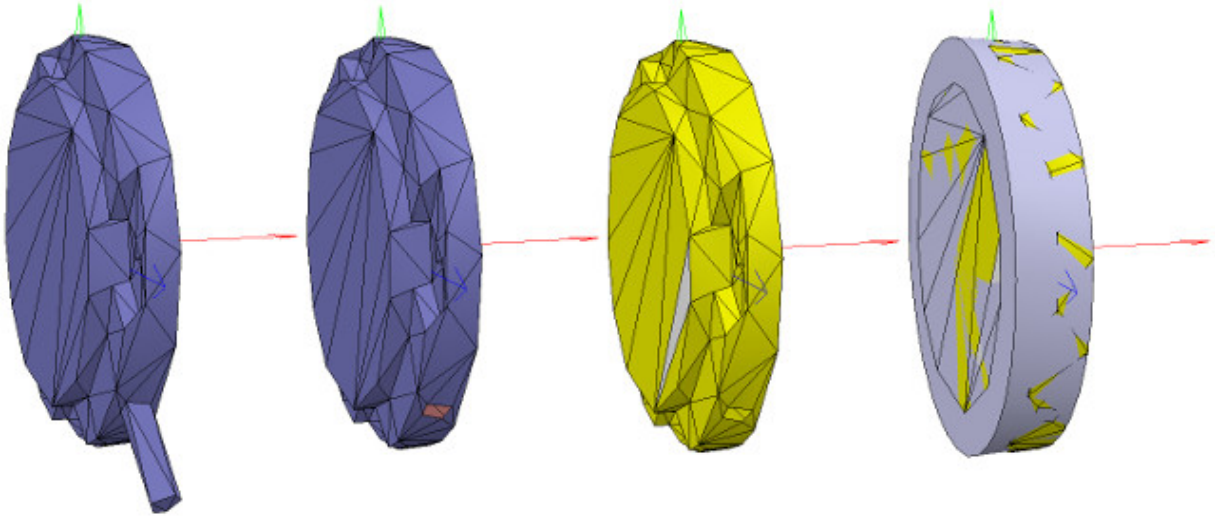
So the order of preference would be: pure shapes, pure compound shapes, convex shapes, compound convex shapes, and finally random shapes. Make sure to also read this page. In case of the robot we want to build, we will make the base of the robot as a pure cylinder, and the other links as convex or convex decomposed shapes.

We could use the dynamically enabled shapes also as the visible parts of the robot, but that would probably not look good enough. So instead, we will build for each visible shape we have created in the first part of the tutorial a dynamically enabled counterpart, which we will keep hidden: the hidden part will represent the dynamic model and be exclusively used by the physics engine, while the visible part will be used for visualization, but also for minimum distance calculations, proximity sensor detections, etc.

We select object robot, copy-and-paste it into a new scene[39] and start the triangle edit mode. If object robot was a compound shape, we would first have had to ungroup it Menu » Edit » Grouping/Merging » Ungroup selected shapes then merge the individual shapes Menu » Edit » Grouping/Merging » Merge selected shapes before being able to start the triangle edit mode. Now we select the few triangles that represent the power cable, and erase them. Then we select all triangles in that shape, and click Extract cylinder. We can now leave the edit mode and we have our base object represented as a pure cylinder:

[38] calculation speed and stability

[39] in order to keep the original model intact

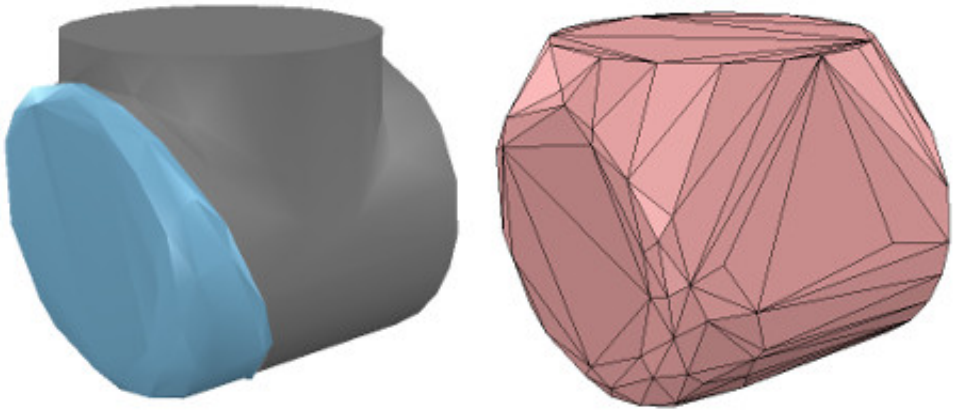


Pure cylinder generation procedure, in the triangle edit mode

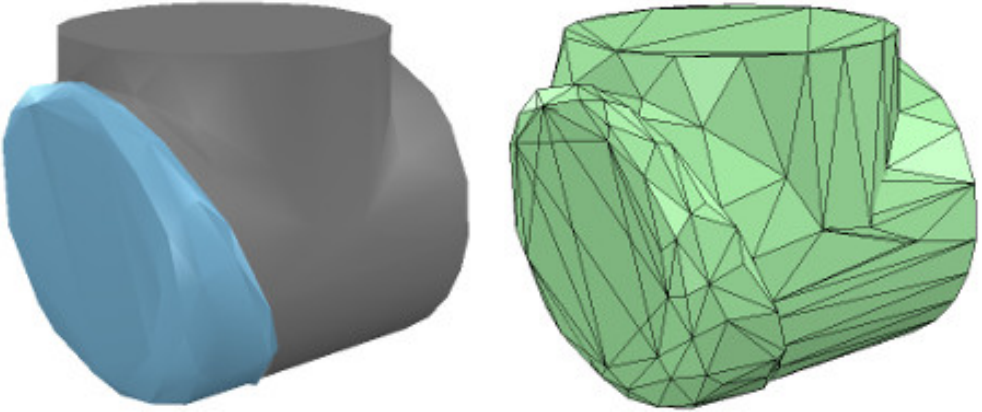
We rename the new shape[40] as `robot_dyn`, assign it to visibility layer 9, then copy it to the original scene. The rest of the links will be modelled as convex shapes, or compound convex shapes. We now select the first mobile link[41] and generate a convex shape from it with Menu » Add » Convex hull of selection. We rename it to `robot_link_dyn1` and assign it to visibility layer 9. When extracting the convex hull doesn't retain enough details of the original shape, then you could still manually extract several convex hulls from its composing elements, then group all the convex hulls with Menu » Edit » Grouping/Merging » Group selected shapes. If that appears to be problematic or time consuming, then you can automatically extract a convex decomposed shape with Menu » Add » Convex decomposition of selection...:

[40] with a double-click on its name in the scene hierarchy

[41] i.e. object **robot_link1**



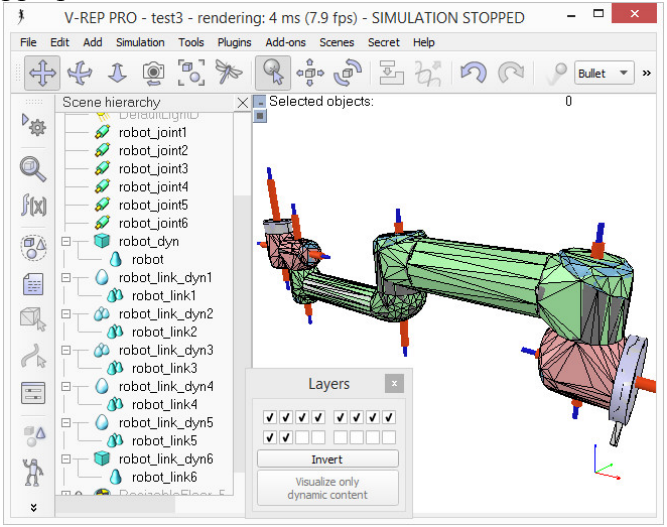
Original shape, and convex shape pendant



Original shape, and convex decomposed shape pendant

We now repeat the same procedure for all remaining robot links. Once that is done, we attach each visible shape to its corresponding invisible dynamic pendant. We do this by selecting first the visible shape, then via Ctrl + ◁ selecting its dynamic pendant then Menu » Edit » Make last selected object parent. The same result can be achieved by

dragging the visible shape onto its dynamic pendant in the scene hierarchy:



Visible shapes attached to their dynamic pendants

We still need to take care of a few things: first, since we want the dynamic shapes only visible to the physics engine, but not to the other calculation modules, we uncheck all object special properties for the dynamic shapes, in the object common properties.

Then, we still have to configure the dynamic shapes as dynamic and respondable. We do this in the shape dynamics properties. Select first the base dynamic shape[42], then check the ☒ Body is respondable item. Enable the first 4 Local respondable mask flags, and disable the last 4 Local respondable mask flags: it is important for consecutive respondable links not to collide with each other. For the first mobile dynamic link in our robot[43], we also enable the Body is respondable item, but this time we disable the first 4 Local respondable mask flags, and enable the last 4 Local respondable mask flags. We repeat the above procedure with all other dynamic links, while always alternating

[42] i.e. **robot_dyn**

[43] i.e. **robot_link_dyn1**

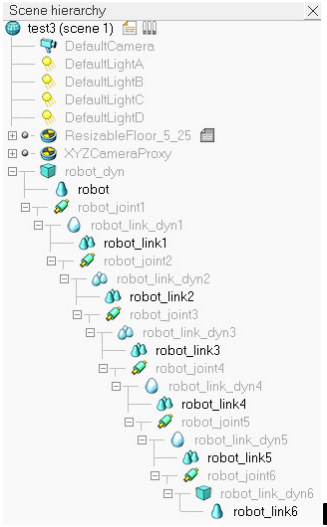
the Local respondable mask flags: once the model will be defined, consecutive dynamic shapes of the robot will not generate any collision response when interacting with each other. Try to always end up with a construction where the dynamic base of the robot, and the dynamic last link of the robot have only the first 4 Local respondable mask flags enabled, so that we can attach the robot to a mobile platform, or attach a gripper to the last dynamic link of the robot without dynamic collision interferences.

Finally, we still need to tag our dynamic shapes as Body is dynamic. We do this also in the shape dynamics properties. We can then enter the mass and inertia tensor properties manually, or have those values automatically computed[44] by clicking Compute mass & inertia properties for selected convex shapes. Remember also this and that dynamic design considerations. This dynamic base of the robot is a special case: most of the time we want the base of the robot[45] to be non-dynamic[46], otherwise, if used alone, the robot might fall during movement. But as soon as we attach the base of the robot to a mobile platform, we want the base to become dynamic[47]. We do this by enabling the Set to dynamic if gets parent item, then disabling the Body is dynamic item. Now run the simulation: all dynamic shapes, except for the base of the robot, should fall. That attached visual shapes will follow their dynamic pendants.

Model definition

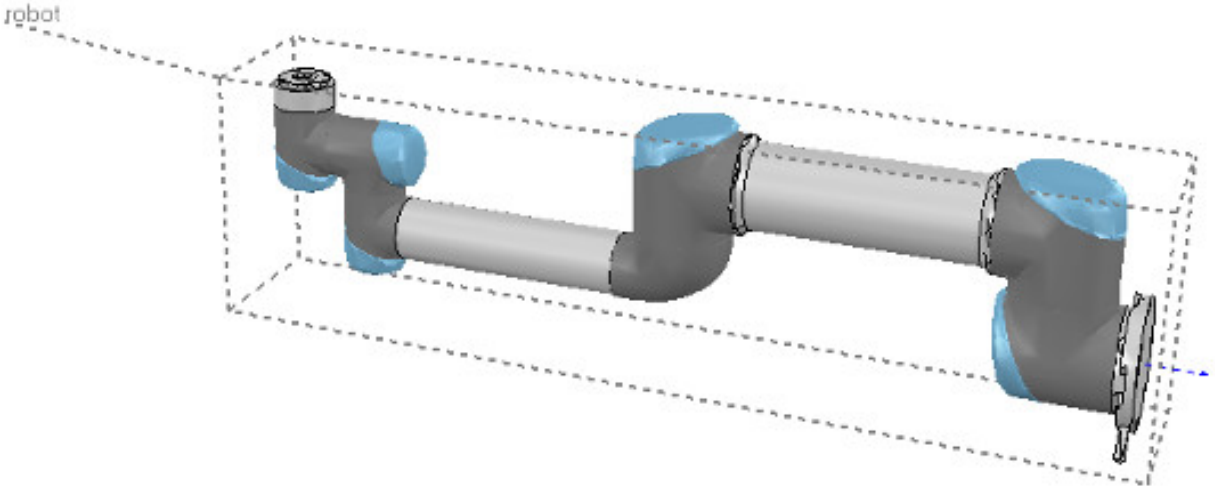
Now we are ready to define our model. We start by building the model herarchy: we attach the last dynamic robot link (robot_link_dyn6) to its corresponding joint (robot_joint6) by selecting robot_link_dyn6, then control-selecting robot_joint6, then Menu » Edit » Make last selected object parent. We could also have done this step by simply dragging object robot_link_dyn6 onto robot_link6 in the scene hierarchy. We go on by now attaching robot_joint6 to robot_link_dyn5, and so on, until arrived at the base of the robot. We now have following scene hierarchy:

[44] recommended
[45] i.e. robot_dyn
[46] i.e. static
[47] i.e. non-static



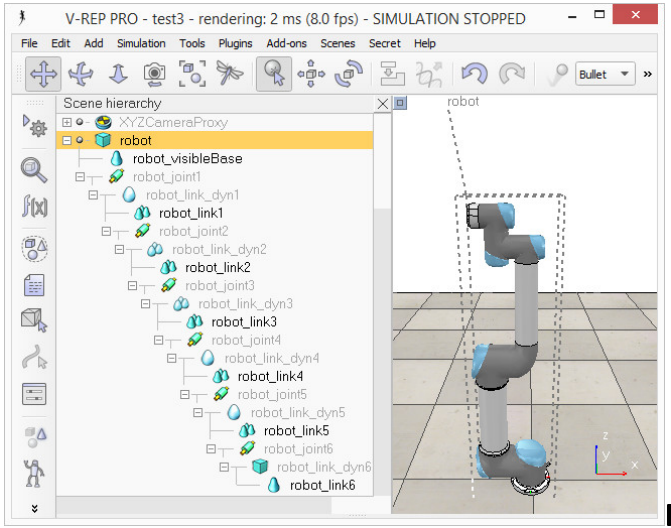
Robot model hierarchy

It is nice and more logical to have a simple name for the model base, since the model base will also represent the model itself. So we rename robot to robot_visibleBase, and robot_dyn to robot. Now we select the base of the hierarchy tree (i.e. object robot) and in the object common properties we enable Object is model base. We also enable Object/model can transfer or accept DNA. A model bounding box appeared, encompassing the whole robot. The bounding box however appears to be too large: this is because the bounding box also encompasses the invisible items, such as the joints. We now exclude the joints from the model bounding box by enabling the Don't show as inside model selection item for all joints. We could do the same procedure for all invisible items in our model. This is also a useful option in order to also exclude large sensors or other items from the model bounding box. We now have following situation:



Robot model bounding box

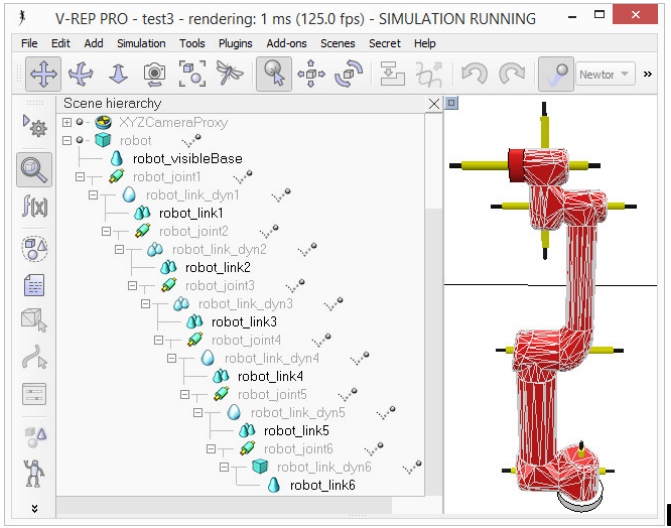
We now protect our model from accidental modification. We select all visible objects in the robot, then enable Select base of model instead: if we now click a visible link in the scene, the base of the robot will be selected instead. This allows us to manipulate the model as if it was a single object. We can still select visible objects in the robot via control-shift-clicking in the scene, or by selecting the object in the scene hierarchy. We now put the robot into a correct default position/orientation. First, we save current scene as a reference (e.g. if at a later stage we need to import CAD data that have the same orientation at the curent robot). Then we select the model and modify its position/orientation appropriately. It is considered good practice to position the model (i.e. its base object) at X=0 and Y=0.



Robot model in default configuration

We now run the simulation: the robot will collapse, since the joints are not controlled by default. When we added the joints in the previous stage, we created joints in force/torque mode, but their motor or controller was disabled (by default). We can now adjust our joints to our requirements. In our case, we want a simple PID controller for each one of them. In the joint dynamic properties, we click Motor enabled and adjust the maximum torque. We then click Control loop enabled and select Position control (PID). We now run the simulation again: the robot should hold its position. Try to switch the current physics engine to see if the behaviour is consistent across all supported physics engines. You can do this via the appropriate toolbar button, or in the general dynamics properties.

During simulation, we now verify the scene dynamic content via the Dynamic content visualization & verification toolbar button. Now, only items that are taken into account by the physics engine will be display, and the display is color-coded. It is very important to always do this, and specially when your dynamic model doesn't behave as expected, in order to quickly debug the model. Similarly, always look at the scene hierarchy during simulation: dynamically enabled objects should display a ball-bounding icon on the right-hand side of their name.



Dynamic content visualization & verification

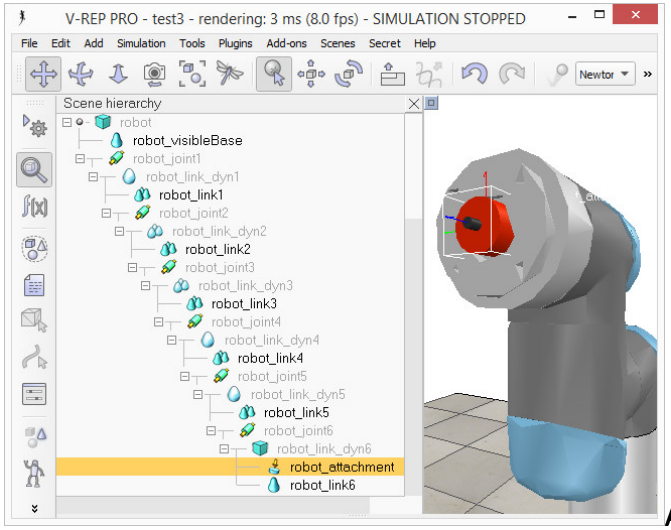
Finally, we need to prepare the robot so that we can easily attach a gripper to it, or easily attach the robot to a mobile platform (for instance). Two dynamically enabled shapes can be rigidly attached to each other in two different ways:

by grouping them select the shapes, then Menu » Edit » Grouping/Merging » Group selected shapes.

by attaching them via a force/torque sensor a force torque sensor can also act as a rigid link between two separate dynamically enabled shapes.

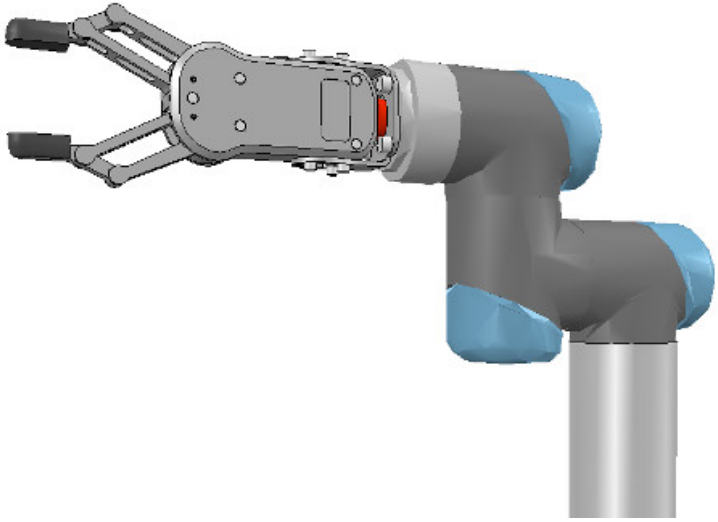
In our case, only option 2 is of interest. We create a force/torque sensor with Menu » Add » Force sensor, then move it to the tip of the robot, then attach it to object robot_link_dyn6. We change its size and visual appearance appropriately[48]. We also change its name to robot_attachment:

[48] a red force/torque sensor is often perceived as an optional attachment point, check the various robot models available



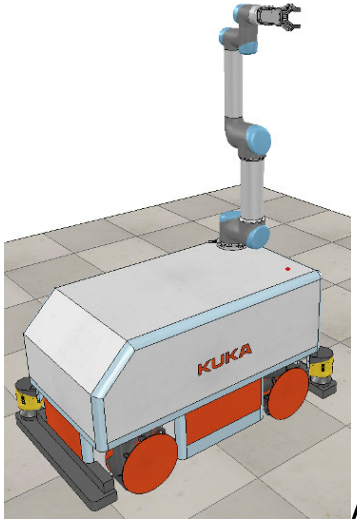
Attachment force/torque sensor

Now we drag a gripper model into the scene, keep it selected, then Ctrl+◁ the attachment force sensor, then click the Assembling/disassembling toolbar button. The gripper goes into place:



Attached gripper

The gripper knew how to attach itself because it was appropriately configured during its model definition. We now also need to properly configure the robot model, so that it will know how to attach itself to a mobile base for instance. We select the robot model, then click Assembling in the object common properties. Disable Object can have the 'parent' role, then click Set matrix. This will memorize the current base object's local transformation matrix, and use it to position/orient itself relative to the mobile robot's attachment point. To verify that we did things right, we drag the model Models/robots/mobile/KUKA Omnirob.ttm into the scene. Then we select our robot model, then control-click one of the attachment points on the mobile platform, then click the Assembling/disassembling toolbar button. Our robot should correctly place itself on top of the mobile robot:



Attached robot

Now we could add additional items to our robot, such as sensors for instance. At some point we might also want to attach embedded scripts to our model, in order to control its behaviour or configure it for various purposes. In that case, make sure to understand how object handles are accessed from embedded scripts. We can also control/access/interface our model from a plugin, from a remote API client, from a ROS node, or from an add-on.

Now we make sure we have reverted the changes done during robot and gripper attachment, we collapse the hierarchy tree of our robot model, select the base of our model, then save it with Menu > File > Save model as... . If we saved it in the model folder, then the model will be available in the model brower.

Глава 2

Внешние контроллеры

[1]

There are several ways one can control a robot or simulation in **V-REP**:

- The most convenient way is to write a child script that will handle the behaviour of a given robot or model. It is the most convenient way, because child scripts are directly attached to scene objects, they will be duplicated together with their associated scene objects, they do not need any compilation with an external tool, they can run in threaded or non-threaded mode, they can be extended via custom Lua function or via a Lua extension library. Another major advantage in using child scripts: there is no communication lag as with the last 3 methods mentioned in this section (i.e. the regular API is used), and child scripts are part of the application main thread (inherent synchronous operation). There are several drawback to writing scripts however: you don't have the choice of the programming language, you can't have the fastest code, and you can't directly access external function libraries, except the Lua extension libraries.

[1] © http://www.coppeliarobotics.com/helpFiles/en/externalControllerTutorial.htm

- Another way one can control a robot or a simulation is by writing a plugin. The plugin mechanism allows for callback mechanisms, custom Lua function registration, and of course access to external function libraries. A plugin is often used in conjunction with child scripts (e.g. the plugin registers custom Lua functions, that, when called from a child script, will call back a specific plugin function). A major advantage in using plugins is also that there is no communication lag as with the last 3 methods mentioned in this section (i.e. the regular API is used), and that a plugin is part of the application main thread (inherent synchronous operation). The drawbacks with plugins are: they are more complicated to program, and they need to be compiled with an external too. Refer also to the plugin tutorial.

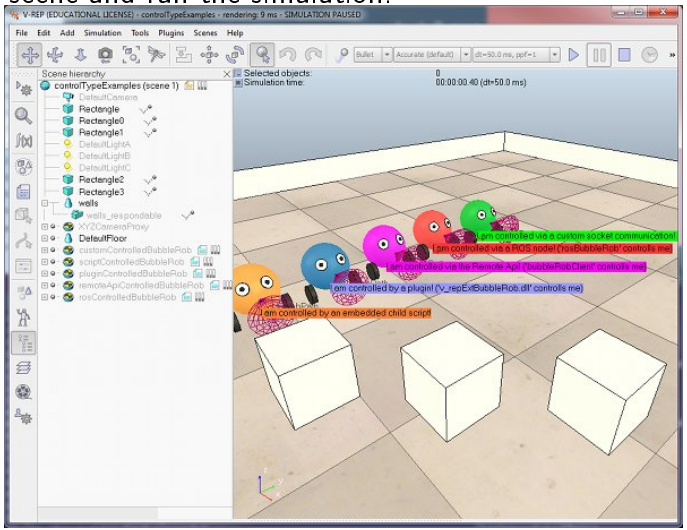
- A third way one can control a robot or a simulation is by writing an external client application that relies on the remote API. This is a very convenient and easy way, if you need to run the control code from an external application, from a robot or from another computer. This also allows you to control a simulation or a model (e.g. a virtual robot) with the exact same code as the one that runs the real robot. The remote API functionality relies on the remote API plugin (on the server side), and the remote API code on the client side. Both programs/projects are open source (i.e. can be easily extended or translated for support of other languages) and can be found in the 'programming' directory of V-REP's installation. Currently there are seven supported languages: C/C++, Python, Java, Matlab, Octave, Lua and Urbi.

- A forth way to control a robot or a simulation is via a ROS node. In a similar way as the remote API, ROS is a convenient way to have several distributed processes communicate with each other. While the remote API is very lightweight and fast, it allows only communication with V-REP. ROS on the other hand allows connecting virtually any number of processes with each other, and a large amount of compatible libraries are available. It is however heavier and more complicated than the remote API. Refer to the ROS interface for details.

- A fifth way to control a robot or a simulation is by writing an external application that communicates via various means (e.g. pipes, sockets, serial port, etc.) with a V-REP plugin or V-REP script. Two major advantages are the choice of programming language, which can be just any language, and the flexibility. Here also, the control

code can run on a robot, or a different computer. This way of controlling a simulation or a model is however more tedious that the method with the remote API.

The V-REP scene file related to this tutorial is "controlTypeExamples.ttt"located in V-REP's "scenes"folder. Open the scene and run the simulation:



The simulation illustrates the 5 control types previously mentioned: the orange robot is controlled by an embedded child script, the blue robot is controlled by a plugin, the purple robot is controlled via the remote API and an external client application, the red robot is controlled via an external ROS application, and the green robot is controlled by an external server application via a custom socket communication. In all 5 cases, child scripts are used, mainly to make the link with the outside world (e.g. launch the correct client application, and pass the correct object handles to it). There are two other ways one can control a robot, a simulation, or the simulator itself: by using customization scripts, or add-ons. They are however not recommended for control and should be rather used to handle functionality while simulation is not running.

As an example, the child script linked to the purple robot has following main tasks:

- Search for a free socket connection port
- Start a remote API server service on the above port
- Launch the controller application ("bubbleRobClient") with the chosen connection port and some object handles as argument

As another example, the child script linked to the red robot has following main tasks:

- Check if the ROS plugin for V-REP was loaded
- Launch the controller application ("rosBubbleRob") with some object handles as arguments

Yet, as another example, the child script linked to the green robot has following main tasks:

- Search for a free socket connection port
- Launch the controller application ("bubbleRobServer") with the chosen connection port as argument
- Locally connect to the controller application
- At each simulation pass, send the sensor values to the controller, and read the desired motor values from the controller
- At each simulation pass, apply the desired motor values to the robot's joints

The project files for the "bubbleRobClient "rosBubbleRob and "bubbleRobServer"applications are located in V-REP's installation folder "programming".

Run the simulation, and copy-and-paste all robots: you will see that the duplicated robots will directly be operational, since their attached child scripts are in charge of launching new instances of "bubbleRobClient "rosBubbleRob calling the appropriate plugin Lua extension function.

Finally, notice that the green robot is controlled via "bubbleRobServer but that its child script acts as client that communicates with the server. You could also imagine having the client part not inside of a child script, but inside of a plugin. Such an example is illustrated in the robot language interpreter integration tutorial 4.1.

Глава 3

Расширение платформы

3.1 Плагины

[1]

A plugin is a shared library (e.g. a dll) that is automatically loaded by V-REP's main client application at program start-up (in a future release, it will also be possible to load/unload plugins on-the-fly). It allows V-REP's functionality to be extended by user-written functions (in a similar way as with add-ons). The language can be any language able to generate a shared library and able to call exported C-functions (e.g. in the case of Java, refer to GCJ and IKVM). A plugin can also be used as a wrapper for running code written in other languages or even written for other microcontrollers (e.g. a plugin was written that handles and executes code for Atmel microcontrollers).

Plugins are usually used to customize the simulator and/or a particular simulation. Often, plugins are only used to provide a simulation with custom Lua commands, and so are used in conjunction with scripts. Other times, plugins are used to provide V-REP with a special functionality requiring either fast calculation capability (scripts are most of

[1] © http://www.coppeliarobotics.com/helpFiles/en/plugins.htm

the times slower than compiled languages) or an interface to a hardware device (e.g. a real robot).

Each plugin is required to have following 3 entry point procedures:

```
extern "C" __declspec(dllexport)
    unsigned char v_repStart(void* reserved,int reservedInt);
extern "C" __declspec(dllexport)
    void v_repEnd();
extern "C" __declspec(dllexport)
    void* v_repMessage(int message,int* auxiliaryData,void* customData,int* replyData);
```

If one procedure is missing then the plugin will be unloaded and won't be operational. Refer to the console window at start-up for the loading status of plugins. Following briefly describes above three entry point purpose:

v_repStart

This procedure will be called one time just after the main client application loaded the plugin. The procedure should:

- check whether the version of V-REP is same or higher than the one that was used to develop the plugin (just make sure all commands you use in the plugin are supported!).
- allocate memory, and prepare GUI related initialization work (if required).
- register custom Lua functions (if required).
- register custom Lua variables (if required) .
- return the version number of this plugin if initialization was successful, otherwise 0. If 0 is returned, the plugin is unloaded and won't be operational.

v_repEnd

This procedure will be called one time just before the simulation loop exits. The procedure should release all resources reserved since `v_repStart` was called.

v_repMessage

This procedure will be called very often while the simulator is running. The procedure is in charge of monitoring messages of interest and reacting to them. It is important to react to following events (best by intercepting the `sim_message_eventcallback_instancepass` message) depending on your plugin's task:

- When objects were created, destroyed, scaled, or when models are loaded: make sure you reflect the change in the plugin (i.e. synchronize the plugin with the scene content)
- When scenes were loaded or the undo/redo functionality called: make sure you erase and reconstruct all plugin objects that are linked to the scene content
- When the scene was switched: make sure you erase and reconstruct all plugin objects that are linked to the scene content. In addition to this, remember that a scene switch will discard handles of following items:
  - communication tubes
  - signals
  - banners
  - drawing objects
  - etc.
- When the simulator is in an edit mode: make sure you disable any "special functionality"provided by the plugin, until the edit mode was ended. In particular, make sure you do not programmatically select scene objects.
- When a simulation was launched: make sure you initialize your plugin elements if needed
- When a simulation ended: make sure you release any memory and plugin elements that are only required during simulation
- When the object selection state was changed, or a dialog refresh message was sent: make sure you actualize the dialogs that the plugin displays

Refer to the messages of type `sim_message_eventcallback_` for more details. When writing plugins several additional points have to be observed or taken into account:

Recommended topics

The main client application

Robot language interpreter plugin tutorial

Глава 4

Создание собственного скриптового языка

В этом разделе мы рассмотрим методику создания плагина, добавляющего в V-REP ваш собственный язык программирования.

Техника программирования синтаксических анализаторов будет особенно полезна, если вам понадобиться читать данные из файлов текстовых форматов, например если нужно будет реализовать в V-REP загрузку файлов моделей, в не поддерживаемых форматах САПР, или реализовать один из ISO-языков промышленных PLC контроллеров (LD/FBD/SFC/ST/IL/CFC)

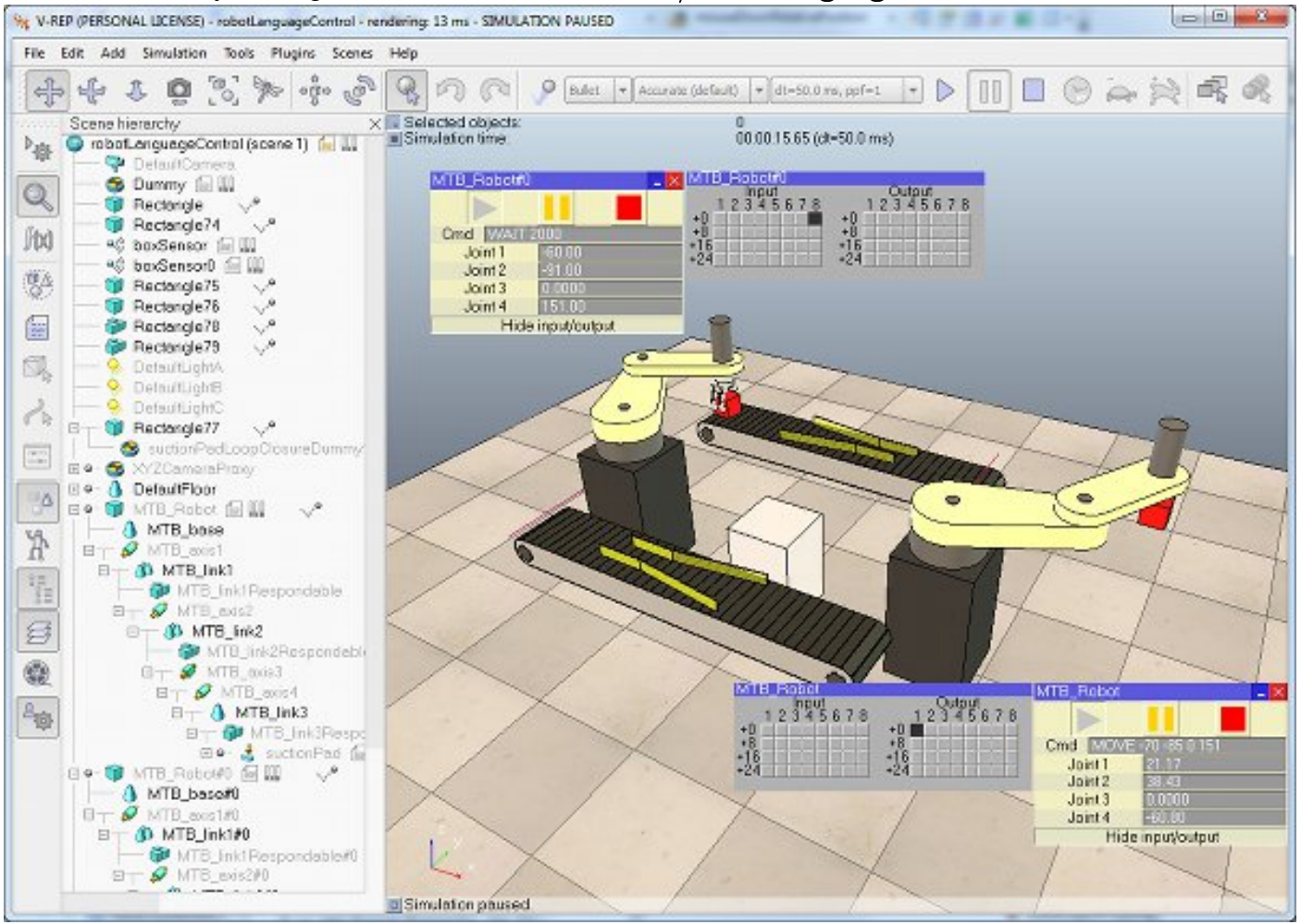
Подробно тема написания трансляторов и компиляторов с использованием фреймворка LLVM рассмотрена в отдельном учебнике [3], а полнофункциональная система разработки и преобразования программ на языке bI описана в мануале на английском [4].

4.1 Robot language interpreter integration

This tutorial will try to explain how to integrate or embed a robot language interpreter into V-REP. The procedure is very similar in case you want to integrate an emulator (e.g. a specific microcontroller emulator) into V-REP. Extending V-REP's functionality requires most of the time the development of a plugin. Make sure you have read the tutorial on plugins 2.2, and the tutorial on external controllers 2 before proceeding with this tutorial.

The V-REP scene file related to this tutorial is located in V-REP's installation folder **scenes/robotLanguageControl** The plugin project files, and the server application project files of this tutorial are located in V-REP's installation folder **programming/v_repExtMtb** and **programming/mtbServer**. The MTB plugin and MTB server should already be compiled in the installation directory.

First, let's start by loading the related scene file **scenes/robotLanguageControl.ttt**:



The MTB robot is an imaginary robot (MTB stands for Machine Type B), that will be controlled with an imaginary robot language.

As previously stated, the used robot language is imaginary and very very simple. Following commands are supported (one command per line, input is case-sensitive):

langtut.imag

```
"REM" starts a comment line
"SETLINVEL v": sets the prismatic joint velocity for next movements (v is in m/s)
"SETROTVEL v": sets the revolute joint velocity for next movements (v is in degrees/s)
"MOVE p1 p2 p3 p4": moves to joint positions (p1;p2;p3;p4) (in degrees except for p3 in m
"WAIT x": waits x milliseconds
"SETBIT y": sets the bit at position y (1-32) in the robot output port
"CLEARBIT y": clears the bit at position y (1-32) in the robot output port
"IFBITGOTO y label": if bit at position y (1-32) in the robot input port is set, jump to
"IFNBITGOTO y label": if bit at position y (1-32) in the robot input port is not set, jump
"GOTO label": jumps to "label"
```

Any word different from "REM "SETLINVEL "SETROTVEL "MOVE "WAIT "SETBIT "CLEARBIT "IFBITGOTO "IF "GOTO"is considered to be a label. Now run the simulation. If the related plugin was not found, following message displays (the display of the message is handled in the child scripts attached to objects MTB_Robot and MTB_Robot#0):

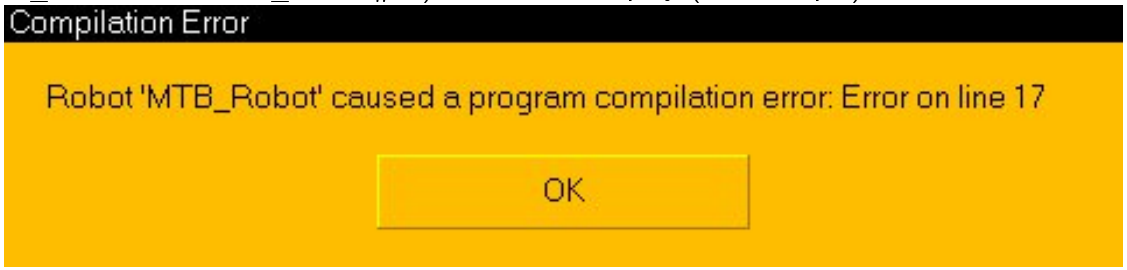
If the related plugin was found, then the the MTB plugin will launch a server application (i.e. mtbServer) that basically represents the robot language interpreter and controller. There is no direct need for a server application, the mtbServer functionality could also be directly running inside of the MTB plugin. The main advantages of using that functionality inside of a server application are:

- The MTB plugin can act as intermediate for as many different languages as needed, also those that haven't been developed yet: the MTB plugin will simply launch the appropriate server depending on the used robot/language.
- If the robot language interpreter / controller crashes, **V-REP** won't crash, since the two are distinct and separate processes.

Currently, the MTB server is in charge of two main tasks:

- receive the program code (i.e. a buffer) from the MTB plugin, compile it, and initialize the robot controller.
- apply input signals, step through the program code (the step duration can be different from step to step), and return output signals and joint angles.

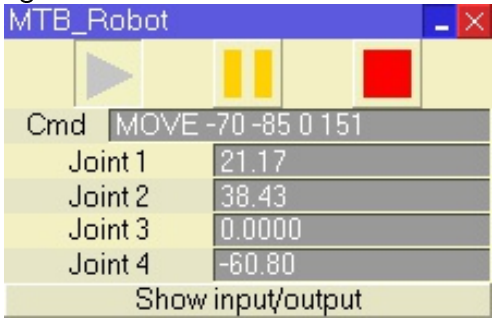
If the MTB server detects an error during compilation of the program code, it will return an error message to the plugin, that will hand it over to the calling child script (i.e. in our case, the child scripts attached to objects MTB_Robot and MTB_Robot#0.), which will display (for example):



If the compilation was successful, then the robots start executing their respective program. The simulation is a maximum speed simulation, but can be switched to real-time simulation by toggling the related toolbar button:

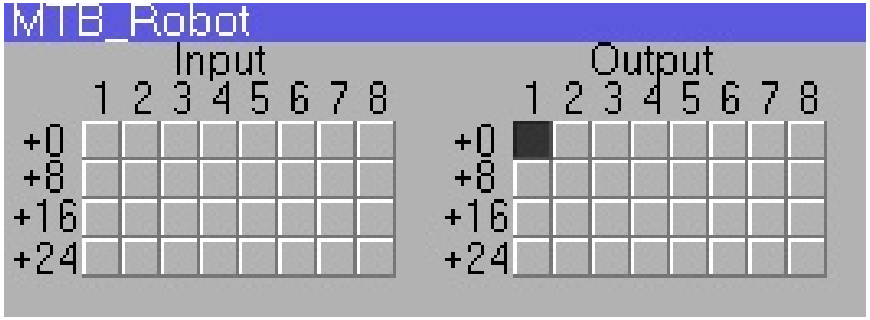
The execution speed can be even more accelerated by pressing the appropriate toolbar button several times:

Each MTB robot program can be individually paused, stopped or restarted at any time via their displayed custom dialog, which are custom user interfaces:



Above custom user interface is the user-interface of the MTB robot and can be fully customized. Should the MTB robot be copied, then its custom user interface will also be copied. Next to being able to controlling the program execution state, the custom user interface also displays current program line (Cmd) and the MTB's current joint values. The button located at the bottom of the custom user interface[1] allows toggling the input/output dialog:

[1] show input/output



The above custom user interface allows the user to change the robot's input port bits, and to read the robot's output port bits. Input and output ports can be read and respectively written by the robot language program. Input and output ports can also be written and read by external devices[2] by using appropriate function calls [3].

There are two child scripts attached to the MTB_Robot and MTB_Robot#0 objects. They are in charge of handling the custom dialogs[4] and communicating with the MTB plugin. Most code in the child scripts could be handled by the plugin too. Open the child script attached to one of the two MTB robot[5]. At the top of the script, you will see the robot language code.

Try to modify an MTB robot's program to have it perform a different movement sequence. Experiment a little bit.

The MTB robots are handled in following way:

- the actual robot language program is compiled and executed by the "mtbServer" application. That application also holds the MTB robot's state variables. For each MTB robot in the simulation scene, there will be an instance of the mtbServer application launched by the v_repExtMtb plugin.
- the v_repExtMtb plugin is in charge of providing custom Lua functions, and also launches the mtbServer application when needed, and communicates with it via socket communication.

[2] e.g. the robot's gripper or suction pad

[3] see further below

[4] custom user interfaces

[5] e.g. with a double-click on the script icon next to the robot model in the scene hierarchy

- the child scripts attached to MTB_Robot and MTB_Robot#0 check whether the v_repExtMtb plugin is loaded, update the custom user interfaces for each robot, and handle the communication with the plugin.

The MTB robot and its simple robot language is a simple prototype meant to demonstrate how to integrate a robot language interpreter into V-REP. It is very easy to extend current functionality for much more complex robots or robot languages. All what is needed is:

- Building the model of the robot. This includes importing CAD data, adding joints, etc. This step can be entirely done in V-REP.
- Writing a plugin to handle the new robot natively, i.e. to handle the new robot by interpreting its own robot language. Any language capable of accessing C-API functions and capable of being wrapped in a dll can be used to create the plugin (but c/c++ is preferred). The robot language interpreter could be directly embedded in the plugin, or launched as an external application (mtbServer) as is done in this tutorial.
- Writing a small child script responsible for handling custom dialogs and linking the robot with the plugin. This step can be entirely done in V-REP.

Now let's have a look at the MTB's plugin project. There is one prerequisites to embedding a robot language interpreter (or other emulator) into V-REP:

- The robot language interpreter should be able to be executed several times in parallel. This means that several interpreter instances should be supported, in order to support several identical, in-parallel operating robots. This can be handled the easiest by launching a new interpreter for each new robot, as is done in this tutorial.

When writing any plugin, make sure that the plugin accesses V-REP's regular API only from the main thread (or from a thread created by V-REP)! The plugin can launch new threads, but in that case those new threads should not be used to access V-REP (they can however be used to communicate with a server application, to communicate with some hardware, to execute background calculations, etc.).

Now let's have a look at the child script that is attached to the MTB robot. The code might seem quite long or complicated. However most functionality handled in the child script could also be directly handled in the plugin, making the child script much smaller/cleaner. The advantage in handling most functionality in the child script is that modifications can be performed without having to recompile the plugin!

Following is the MTB robot's child script main functionality:

- Checking whether the plugin was loaded. If not, an error message is output.
- Communicating with the plugin. This means that information is sent to and received from the MTB plugin with custom Lua functions.
- Applying the newly calculated joint values to the MTB robot model. This could also be handled in the MTB's plugin.
- Reacting to events on the custom dialogs (custom user interfaces), like button presses. This could also be handled in the MTB's plugin.
- Updating the state of the custom dialogs (custom user interfaces), by changing their visual appearance (e.g. displaying the current joint positions, etc.). This could also be handled in the MTB's plugin.

Following 3 custom Lua functions are of main interest (others are exported by the plugin):

```
number mtbServerHandle,string message=simExtMtb_startServer(string mtbServerExecutable,
    number portNumber,charBuffer program,table_4 jointPositions, table_2 velocities)

number result,string message=simExtMtb_step(number mtbServerHandle,number timeStep)

table_4 jointValues=simExtMtb_getJoints(number mtbServerHandle)
```

simExtMtb_startServer launches the server application (e.g. mtbServer) on the specified port, connects to it, and sends it the robot language code, the initial joint positions, and the initial velocities. In return, the function returns a server handle (if successful), and a message (usually a compilation error message).

simExtMtb_step steps through the robot language program with the specified timeStep, and returns a result value and a message (usually the code being currently executed).

simExtMtb_getJoints retrieves the current joint positions. The joint positions are automatically updated when simExtMtb_step is called.

You could also imagine slightly modifying the step function, and add one additional function, in order to be able to handle intermediate events triggered by the robot language program execution. In that case, each simulation step

would have to execute following script code (in a non-threaded child script):

```
local dt=simGetSimulationTimeStep()
while (dt>0) do
    result,dt,cmdMessage=simExtMtb_step(mtbServerHandle,dt)
        -- where the returned dt is the remaining dt
    local event=simExtMtb_getEvent()
    while event~=-1 do
        -- handle events here
        event=simExtMtb_getEvent()
    end
end
```

4.2 Лексер

4.3 Парсер (синтаксический анализатор)

4.4 Ядро скриптового языка на C^{++}

Литература

[1] v-rep: virtual robot experimentation platform

[2] Habr Программируем роботов — бесплатный робосимулятор **V-REP**. Первые шаги

[3] github: `https://github.com/ponyatov/lexman`
обработка текстовых форматов данных и реализация компьютерных языков

[4] github: `https://github.com/ponyatov/Y/tree/dev`
Язык bI