打开
to start V-REP in headless mode (i.e. without any GUI), load the scene *myScene.ttt*, run the simulation for 5 seconds, then stop the simulation and automatically leave V-REP again
./vrep.sh -h -s5000 -q myScene.ttt

导入/导出图形
V-REP 使用三角形面片来描述和显示图片，因此只支持此类格式的模型导入([Menu bar --> File --> Import --> Mesh...])，支持的格式有：OBJ, DXF, 3DS, STL, COLLADA。此外还可以通过 URDF 插件导入([Menu bar --> Plugins --> URDF import...])UEDF 模型。导入的装配图分为零件图：([Menu bar --> Edit --> Grouping/Merging --> Divide selected shapes])
选中建好的模型后可以导出选中的模型([Menu bar --> File --> Export --> Selected shapes...])，VREP 支持的导出模型有：DXF, OBJ, STL, COLLADA
Importing and exporting shapes

仿真
仿真过程设立了很多中间状态来告知脚本文件，可以通过 simGetSimulationState()返回。
仿真进程按照固定时间步长来执行。需要注意的是，仿真时间与真实时间很可能不一致，如果没有使能实时模式，仿真器会尽可能快地运行。
默认情况下，每个仿真循环会执行两个操作：执行主脚本，渲染场景。仿真速度主要取决于仿真时间步长(Time Step)和渲染一步需要的仿真次数(Simulation passes per frame, ppf)，两者可以在仿真对话框中设置。在仿真过程中，可以通过工具栏上的仿真速度调节按钮控制仿真速度


仿真对话框


仿真速度调节按钮


多线程渲染按钮

增大仿真时间步长虽然会加快仿真速度，但是会使仿真精度变差，甚至会变得不稳定。因此通常通过增大 ppf 的值来提高仿真速度。有时，为了以最大速度运行仿真，在仿真循环中只包括主脚本的执行，图形的渲染工作将会在新的线程中进行，原先的 simulation cycle 将只执行 main script，因此计算速度会大大加快。但这也会带来许多问题，比如渲染和仿真的不同步可能会导致视觉差错的产生。

主脚本和子脚本(main script and the child scripts)
在仿真中，仿真脚本（包括主脚本和子脚本）起着关键的作用，主脚本包括仿真中的循环代码，子脚本通常是机器人、传感器或驱动器的控制代码。默认情况下，每个场景都有一个主脚本来处理所有的功能，场景中的每个对象(object)都可以关联一个子脚本来处理仿真中某一部分，最常见的用法是控制模型。子脚本会随这关联对象的复制一起复制，而主脚本不会–在一个场景中，主脚本有且必须只有一个。

主脚本（main script)
VREP 中的每一个场景都默认带一个主脚本（main script）文件，它包含了基本的控制代码，用于控制仿真进程。
main script 中的代码按照功能可分为 *Initialization part, regular part, Clean-up part*，如下是一个典型的主脚本：

```
-- Initialization part:
if (sim_call_type==sim_mainscriptcall_initialization) then
    simHandleSimulationStart()
    simOpenModule(sim_handle_all)
    simHandleGraph(sim_handle_all_except_explicit,0)
end
-- regular part:
if (sim_call_type==sim_mainscriptcall_regular) then
    -- Actuation part:
    simResumeThreads(sim_scriptthreadresume_default)
    simResumeThreads(sim_scriptthreadresume_actuation_first)
    simLaunchThreadedChildScripts()
    simHandleChildScripts(sim_childscriptcall_actuation)
    simResumeThreads(sim_scriptthreadresume_actuation_last)
    simHandleCustomizationScripts(sim_customizationscriptcall_simulationactuation)
    simHandleModule(sim_handle_all,false)
    simResumeThreads(2)
    simHandleMechanism(sim_handle_all_except_explicit)
    simHandleIkGroup(sim_handle_all_except_explicit)
    simHandleDynamics(simGetSimulationTimeStep())
    simHandleMill(sim_handle_all_except_explicit)
    -- Sensing part:
    simHandleSensingStart()
    simHandleCollision(sim_handle_all_except_explicit)
```

```
    simHandleDistance(sim_handle_all_except_explicit)
    simHandleProximitySensor(sim_handle_all_except_explicit)
    simHandleVisionSensor(sim_handle_all_except_explicit)
    simResumeThreads(sim_scriptthreadresume_sensing_first)
    simHandleChildScripts(sim_childscriptcall_sensing)
    simResumeThreads(sim_scriptthreadresume_sensing_last)
    simHandleCustomizationScripts(sim_customizationscriptcall_simulationsensing)
    simHandleModule(sim_handle_all,true)
    simResumeThreads(sim_scriptthreadresume_allnotyetresumed)
    simHandleGraph(sim_handle_all_except_explicit,simGetSimulationTime()+simGetSimulationTimeStep())
end
-- Clean-up part:
if (sim_call_type==sim_mainscriptcall_cleanup) then
    simResetMilling(sim_handle_all)
    simResetMill(sim_handle_all_except_explicit)
    simResetCollision(sim_handle_all_except_explicit)
    simResetDistance(sim_handle_all_except_explicit)
    simResetProximitySensor(sim_handle_all_except_explicit)
    simResetVisionSensor(sim_handle_all_except_explicit)
    simCloseModule(sim_handle_all)
end
```

子脚本（child script）

non-threaded child script icon (left), threaded child script icon (right)

1. 非线程子脚本(non-threaded child script): 每次被调用时，执行任务并返回控制指令，如果控制指令未返回，则整个仿真等待直到返回指令。如果场景中有多个 Non-threaded 子脚本，则它们会按照父子关系链的顺序逐级向下执行，即会先从模型的根节点（或没有父节点的物体）开始，逐级向下，直到叶节点（或没有子节点的物体）结束。一个非线程脚本包括四个部分：the initialization part, the actuation part, the sensing part, the restoration part

例子：门前、门后有两个接近传感器，当感测到人时，门自动打开。

```
--the initialization par
if (sim_call_type==sim_childscriptcall_initialization) then
    sensorHandleFront=simGetObjectHandle("DoorSensorFront")
    sensorHandleBack=simGetObjectHandle("DoorSensorBack")
    motorHandle=simGetObjectHandle("DoorMotor")
end
-- the actuation part
if (sim_call_type==sim_childscriptcall_actuation) then
    resF=simReadProximitySensor(sensorHandleFront)
    resB=simReadProximitySensor(sensorHandleBack)
    if ((resF>0)or(resB>0)) then
        simSetJointTargetVelocity(motorHandle,-0.2)
    else
        simSetJointTargetVelocity(motorHandle,0.2)
    end
end
-- the sensing part
if (sim_call_type==sim_childscriptcall_sensing) then

end
-- the restoration part
if (sim_call_type==sim_childscriptcall_cleanup) then
    -- Put some restoration code here
end
```

2. 线程子脚本(threaded child script): 每次调用时开启线程执行程序。与非线程脚本相比，如果编程不当，会耗费更多的处理时间，并且对停止仿真命令的响应慢点。一个线程脚本包括三个部分：the initialization part, the regular part, the restoration part

对于同一任务，非线程子脚本的程序：

```
threadFunction=function()
    while simGetSimulationState()~=sim_simulation_advancing_abouttostop do
     -- simSetThreadAutomaticSwitch(false) -- disable automatic thread switches
        resF=simReadProximitySensor(sensorHandleFront)
        resB=simReadProximitySensor(sensorHandleBack)
        if ((resF>0)or(resB>0)) then
            simSetJointTargetVelocity(motorHandle,-0.2)
        else
            simSetJointTargetVelocity(motorHandle,0.2)
        end
```

```
        -- this loop wastes precious computation time since we should only read new
        -- values when the simulation time has changed (i.e. in next simulation step).
    end
end
--the initialization par
sensorHandleFront=simGetObjectHandle("DoorSensorFront")
sensorHandleBack=simGetObjectHandle("DoorSensorBack")
motorHandle=simGetObjectHandle("DoorMotor")
-- the regular part
res,err=pcall(threadFunction)
if not res then
        simAddStatusbarMessage('Lua runtime error: '..err)
end
-- the restoration part
```

默认情况下（可以通 simSetThreadSwitchTiming 命令改变），VREP 使用线程来模仿协程，每个线程运行 1-2ms 然后转向其他线程。线程的转换是自动完成的，使用 simSwitchThread 命令可以实现同步的效果、减小资源浪费。在上边程序对应注释处添加：

```
simSwitchThread() -- Switch to another thread now!
       -- from now on, above loop is executed once every time the main script is about to execute.
       -- this way you do not waste precious computation time and run synchronously.
```

如上面的代码中没有调用 simSwitchThread()与 main script 进行同步，则线程中的 while 循环会以最大速度一直执行。即如果 main script 以默认速度 50ms 执行一次，而 threaded child script 中的 while 循环可能不到 1ms 就执行了一次，这样还没等场景中其它物体发生改变（比如人还没有走近），就已经查询传感器状态和设置关节速度很多次，导致资源浪费。

Child Scripts

定制脚本（Customization script）

customization script icon

与子脚本同样，定制脚本与具体的对象关联，选中对象直接右击添加或通过主菜单添加[menu bar --> Add --> Associated customization script]. 定制脚本的主要特征是：

1. 非线程运行，与非线程子脚本同样，执行任务结束返回控制代码，控制代码未返回则等待任务完成
2. 不管仿真是否运行，都在执行。因此一个典型的应用是通过滑条来重新设置机器人各个连杆的长度，如场景中的默认对象 *ResizableFloor_5_25*
3. 与场景中的对象关联，这种脚本关联框架组成了 VREP 分布式控制框架的基础，与子脚本相同，具有对应场景对象复制时自动复制的灵活特点

VREP 中的通信方式

Signals：全局变量，支持整型、浮点型和字符串数据类型

Tubes： 类似与管道的双向通信，由于数据只在所选的 Tube 中进行通信，可避免混乱，常用于两实体间的数据通信。相关通信函数有：

```
-- Tubes opened via a script will automatically close upon simulation end.
number tubeHandle = simTubeOpen(number dataHeader, string dataName, number readBufferSize)
-- Sends a data packet into a communication tube previously opened with simTubeOpen
number result = simTubeWrite(number tubeHandle, string data)
-- Receives a data packet from a communication tube previously opened with simTubeOpen.
string data = simTubeRead(number tubeHandle, boolean blockingOperation=false)
```

Wireless communication simulation： vrep 的无线通信非常灵活，传输的数据可以以特定的方向和距离传播，只有在对应位置，接收者才能接收到数据。可通过 Menu bar --> Tools --> Environment --> Visualize wireless emissions /Visualize wireless receptions 实现可视化。

Persistent data blocks：持久化的全局缓存变量，可以在打开的所有场景中共享。persist 直到仿真结束,也可以在文件上 presist，在下次仿真时自动加载。

Child scripts：子脚本不仅可以读写自有的参数，还可以处理其他脚本的仿真参数。

Serial port communication：串口通信

LuaSocket：网络通信

Means of communication in and around V-REP

以编程方式访问对象

在 VREP 中编程时通常需要访问不同的对象，比如 scene objects, IK groups, distance objects 等等，可以通过句柄索引函数来实现。

1. 由未关联代码访问（Access from *unassociated* code ）

未关联代码是没有附加到任何场景对象的代码，包括 plugins, adds-on, remote API, ROS nodes, the main script 和 contact callback script

```
// e.g. inside of a c/c++ plugin:
int cuboid1Handle=simGetObjectHandle("Cuboid1"); // handle of object "Cuboid1"
int cuboid2Handle=simGetObjectHandle("Cuboid2"); // handle of object "Cuboid2"
int cuboid1Hash0Handle=simGetObjectHandle("Cuboid1#0"); // handle of object "Cuboid1#0"
int ikGroupHash42Handle=simGetIkGroupHandle("ikGroup#42"); // handle of ik group "ikGroup#42"
# e.g. inside of a Python remote API client:
opMode=vrep.simx_opmode_blocking
res,cuboid1Handle=vrep.simxGetObjectHandle(clientId,"Cuboid1",opMode) # handle of object "Cuboid1"
res,cuboid2Handle=vrep.simxGetObjectHandle(clientId,"Cuboid2",opMode) # handle of object "Cuboid2"
res,cuboid1Hash0Handle=vrep.simxGetObjectHandle(clientId,"Cuboid1#0",opMode) # handle of object
"Cuboid1#0"
```

2. 由关联代码访问（Access from *associated* code ）

未关联代码是没有附加到场景对象的代码，包括 child script, customization scripts 和 joint control callback scripts

VREP 中的场景对象可以复制，场景对象复制后，复制的对象会在原对象名称的上自动添加后缀，如下所示

Accessing objects programmatically

BubbleRob tutorial 学习笔记

在一个物体内部为避免各对象间 collision 可以设置不同的掩码，在 shape dynamics properties[Scene Object Properties→ show dynamic properties dialog])中设置 bubbleRob_slider 的 local **respondable mask** 为 00001111，bubbleRob 的 local respondable mask 为 11110000

VREP 中可以将场景里的物体组成一个集合(collection)，后续的一些操作，比如碰撞检测时就可以选取定义好的集合。将物体添加进集合有多种方式.**Collection**([Menu bar --> Tools --> Collections]或直接点击左侧图标)定义了对象的 Collection，在本例中将机器人 BubbleRob 涉及的对象建一个 collection，命名为 bubbleRob_collection，然后打开 Calculation([Menu bar --> Tools --> Calculation module properties]或直接点击左侧图标)在 distance 对话框添加新的 **distance** object，选择距离对为 bubbleRob_collection 和 any other measurable object in the scene，重命名为 bubbleRob_distance。bubbleRob_distance 用来测量障碍物的最近距离，仿真开始后实时计算，并会显示出来，这个过程计算量比较大，如要关闭，可以在 distance 对话框中关闭 Enable distance calculations。

Collection

Calculation module properties

在添加 Cylinder 型障碍物后，需要使其保持静态（即不能被其他物体 Collision），但仍然可以 Collision 其它对象， 可以在 shape dynamics properties 勾掉 **Body is dynamic**。在 Object common properties 中勾选 Collidable, Measurable, Renderable（可以被 vision sensors 感知） and Detectable(可以被 proximity sensors 感知）以使其可碰撞、可测、可感知。

利用 **graph**([Menu bar --> Add --> Graph])可以显示轨迹、数据。在 Scene Object Properties 中 Data stream recording list 添加新的数据流后在仿真的时候会以在 t,x 坐标轴中显示数据的变化（可以选中数据流勾掉下方的 visible 来不显示），如在坐标轴中添加 Object: absolute x-position/bubbleRob 来记录 bubbleRobx 轴坐标轴的变化，由于之前定义了 bubbleRob_distance 因此还可在坐标轴可以添加 Distance: segment length /bubbleRob_distance 对来显示障碍物距离。数据流还有一个作用是作为轨迹曲线的坐标值，点击 Edit 3D curves 可以在添加的 graph 上绘制轨迹，依次将对用的数据流赋给 x,y,z 坐标值。

要将 BubbleRob 定义为 **model**，在 base model 即 bubbleRob 的 object common properties 中使能 Object is model base 和 Object/model can transfer or accept DNA（选中该项当 object/model 复制时，它们拥有共同的 identifier，选中修改后的任一 object/model 然后点击 DNA transfer toolbar 其他模型都会修改）。在 Scene Hierarchy 中选中 BubbleRob 下的两个关节、Graph、Proximity sensor 使能 Don't show as inside model selection，点击 Apply to selection 将这些对象不包括在模型中。

可以保存选中的模型[Menu bar --> File --> Save model as...]

可以双击 Scene Hierarchy 中的 model base 旁的 Model tag 来修改模型属性。

添加 **vision sensor** 在 Scene Object Properties 中可以设置相关参数，选择 show filter dialogue 可以选择滤波算法来处理得到的图像。设置完成后为了显示得到的图像，在 scene 中右击选择 Add--> floating view，出现悬浮框，选择需要显示的 Vision sensor 在悬浮框中右击 view Associate-->view with selected sensor.

Vision sensor 和 Camera 的几点不同:

vision sensor 有固定的分辨率，而 Camera 自动调节分辨率;

vision sensor 的图像内容可以通过 API 读取，可以进行滤波等图像处理;

vision sensor 的内存占用更多、处理速度更慢;

vision sensor 只能显示 renderable 的对象，而 camera 可以显示所有对象。

对象的 object common properties 有一项 camera visibility layer，如果选为 9-16 默认情况是对象不可见，在 scene hierarchy 中对象颜色也会变灰。

Line following BubbleRob tutorial 学习笔记
可以用 **page** selector toolbar button 选择不同的视角，有 8 个预定义的视角可以选择。这里选择 page 4 俯视图来添加路线。[Menu bar --> Add --> Path --> Circle type]添加圆，在 Scene hierarchy 选中 Path，点击 Path edit mode toolbar button 进入 path edit model 后选择点然后通过 Object/Item shift 拖拽，形成想要的环形路径。

Page selector toolbar button

Path edit mode toolbar button

打开在 hierarchy 中双击 **path** object 打开属性界面，点击 Show path shaping dialog 后，选择 Type 为 horizontal segment(轮廓样式），scaling factor 为 4，调节颜色为黑。最后为了避免 Z-fighting(主要是指当两个面共面时，二者的深度值一样，深度缓冲就不能清楚的将它们两者分离开来，位于后面的图元上的一些像素就会被渲染到前面的图元上，最终导致图象在帧与帧之间产生微弱的闪光)将 path object 的位置向上移动路径 0.5mm。
最后程序部分也很有意思，可以看看。
Inverse kinematics tutorial 学习笔记
在本教程中，整个过程为：1.其它三维绘图软件建立的机械臂 stl 文件导入到 vrep 中并 divide 为各个部件、2.添加运动关节、3.group 部件为连杆、4.添加 dummy、5.定义 IKgroup 进行逆运动学求解、6.添加 collision 进行碰撞检测。
1. 选中导入的机械臂，Menu bar --> Edit --> Grouping/Merging --> **Divide** selected shapes
2. 选中所有导入的对象[Menu bar --> Edit -->Reorient Bounding box--> with reference frame of world]来使得相对于世界坐标系。添加关节[Menu bar --> Add --> Joint --> Revolute] 然后选中与每个关节对应的机械臂对象，点击 Item/Object shift 选择 apply to selection 使得关节移到对应的位置，再通过 Item/Object Rotate 调整关节转向。
3. 将各 divide 的部件通过[Menu bar --> Edit --> Grouping/Merging --> **Group** selected shapes]group 为 1-7 的连杆。然后按照基座（连杆 0）--> 关节 1--> 连杆 1 -->关节 2--> 连杆 2--> … -->关节 7--> 连杆 7（末端执行器）在 Scene hierarchy 中建立层级结构。为了便于操作，将机械臂定义为 model，link0 为 Object is model base。
4. 为了组成运动链，需要添加两个 **dummy**[Menu bar --> Add --> Dummy]对象，一个为 tip，一个为 target。在 Scene hierarchy 中将 tip 放到 link7 下，并将它们位置对齐。打开 tip 的属性，选择 linked dummy 为 target，link type 为 IK,tip-target
5. [Menu bar --> Tools --> Calculation module properties]或直接点击左侧图标)在 Inverse Kinematics 对话框添加新的 **IK group**，由于该机械臂是七自由度的使能下边的 Mechanism is redundant 选项。然后 Edit IK elements，打开新的对话框，下拉菜单选择 tip 然后 Add，添加后选中，选择 Base 为 link0。这时候就可以进行仿真了，仿真时移动 target 的位置，机械臂会进行跟随。为了方便拖动，可以添加一个尺寸为 0.05 的 Sphere 对象， 在 Scene hierarchy 中将 target 放到 Sphere 下，并将它们位置对齐，该对象更像是一个 UI 元素，因此勾掉其所有动态性能。
6. 添加 Collection[Menu bar --> Tools --> Collections]，选择 link0，然后点击 Add new collection，然后定义内容，点击 Add，会添加选中的对象，可以点击 Visualize selected collection 来观察，无误后将 collection 重命名为"redundantRob"。建立 Collection 后打开 Calculation 里的 **Collision** 对话框，点击 Add new collision object 选择 pairs 为"[Collection] redundantRob" - "all other collidable objects in the scene"。现在进行仿真时如果碰到障碍物，颜色会变红表示发生碰撞。

External controller tutorial 学习笔记
在 V-REP 中有多种方式可以控制机器人：

child script	no communication lag as with the last 3 methods mentioned 只能用 Lua 编程，除了 Lua extension Lib 外无法使用其他第三方库
plugin	often used in conjunction with child scripts no communication lag as with the last 3 methods mentioned more complicated to program, need to be compiled with an external
remote API	often used to run the control code from an external application open source, several supported languages
ROS node	interface with ROS, similar to remote API but more complicated
others	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.

还有其它两种方法：customization scripts, add-ons, 但一般不推荐用来控制，而是在仿真不运行时用于处理功能。

Plugins

在 Linux 上 plugin 的命令 libv_repExt*.so，如在该例中为 v_repExtBubbleRob。 V-REP 在启动时会自动加载安装包下的所有插件。