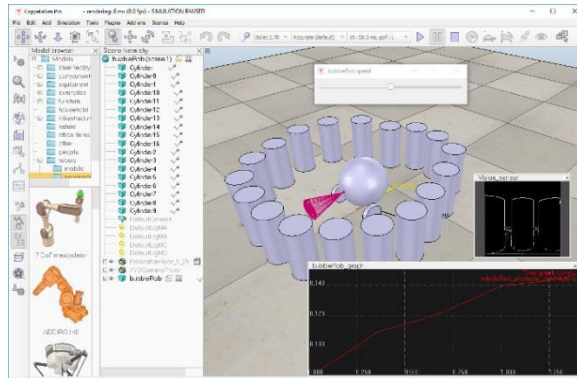


工作任務:您需要從 <http://www.coppeliarobotics.com/helpFiles/index.html> 了解什麼來

實現四輪機器人？

## 章節:BubbleRob tutorial



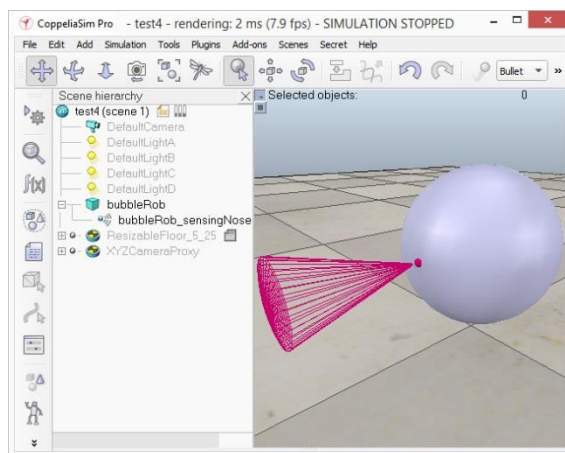
我們使用[菜單欄->添加->基本形狀->球體]將直徑為 0.2 的基本球體添加到場景中。我們將 X 尺寸項目調整為 0.2，然後點擊“確定”。默認情況下，創建的球體將顯示在可見性層 1 中，並且是動態且可響應的（因為我們已啟用“創建動態且可響應的形狀”項）。我們開始模擬，複製並貼上創建的球體（使用[菜單欄->編輯->複製所選對象]，然後[菜單欄->編輯->黏貼緩衝區]，這兩個球將對碰撞做出反應並滾動。

我們在“平移”選項卡上打開“位置”對話框，選擇表示 BubbleRob 身體的球體，並為“沿 Z”輸入 0.02。確保將“相對於”項設置為“世界”。然後我們點擊翻譯選擇。這會將所有選定對象沿絕對 Z 軸平移 2 cm，並有效地將我們的球體抬高了一點。在場景層次結構中，我們雙擊球體的名稱，以便我們可以編輯其名稱。我們輸入 bubbleRob，然後按 Enter。

接下來，我們將添加一個接近傳感器，以便 BubbleRob 知道它何時接近障礙物：我們選擇[菜單欄->添加->接近傳感器->圓錐類型]。在“方向”選項卡上的“方向”對話框中，我們為“周圍的 Y”和“周圍的 Z”輸入 90，然後點擊“旋轉選擇”。在位置對話框的“位

置”選項卡上，為 X 坐標輸入 0.1。 Z 坐標為 0.12。現在，接近傳感器已相對於 BubbleRob 的身體正確定位。我們在場景層次中點擊接近傳感器的圖標以打開其屬性對話框。我們點擊顯示體積參數以打開接近傳感器體積對話框。我們將偏移量調整為 0.005，角度調整為 30，範圍調整為 0.15。然後在接近傳感器屬性中，點擊“顯示檢測參數”。這將打開接近傳感器檢測參數對話框。如果距離小於則取消選中“不允許檢測”項，然後再次關閉該對話框。在場景層次結構中，我們點擊接近傳感器的名稱，以便我們可以編輯其名稱。我們輸入 bubbleRob\_sensingNose 並按返回鍵。

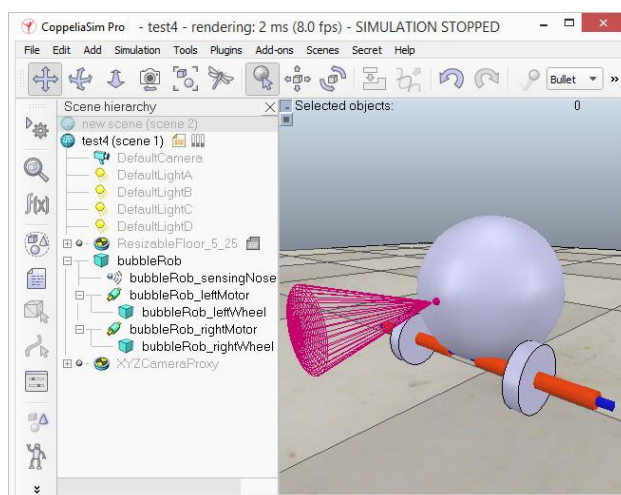
我們選擇 bubbleRob\_sensingNose，然後按住 Control 鍵選擇 bubbleRob，然後點擊[菜單欄->編輯->將上一個選定的對象設為父對象]。這會將傳感器連接到機器人的身體。我們還可以將 bubbleRob\_sensingNose 拖動到場景層次中的 bubbleRob 上。



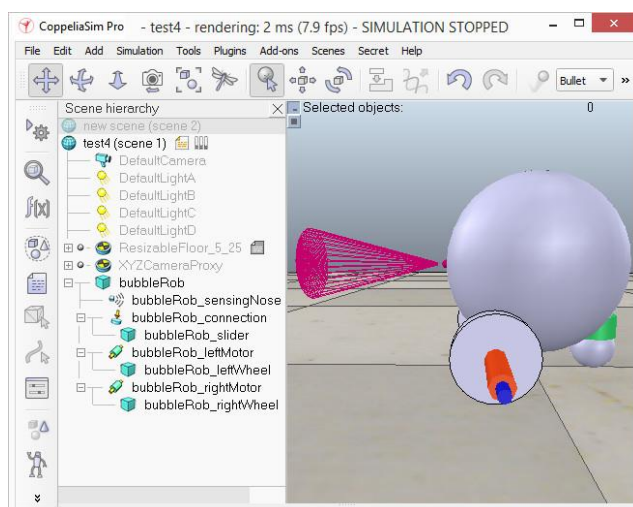
接下來，我們將製作 BubbleRob 的車輪。我們使用[菜單欄->文件->新場景]創建一個新場景。跨多個場景工作通常非常方便，以便可視化並僅對特定元素進行工作。我們添加一個尺寸為 ( 0.08,0.08,0.02 ) 的純原始圓柱體。對於 BubbleRob 的主體，如果尚未啟用，則在該圓柱的對象通用屬性中啟用 Collidable，Measurable，Renderable 和 Detectable。然

後，將圓柱的絕對位置設置為 ( 0.05,0.1,0.04 )，並將其絕對方向設置為 ( -90,0,0 )。我們將名稱更改為 bubbleRob\_leftWheel。我們複製並貼上滾輪，然後將複製的絕對 Y 坐標設置為-0.1。我們將副本重命名為 bubbleRob\_rightWheel。我們選擇兩個輪子，複製它們，然後切換回場景 1，然後貼上輪子。

現在，我們需要為車輪添加接頭（或電動機）。我們單擊[菜單欄->添加->關節->旋轉]將旋轉關節添加到場景。在大多數情況下，將新對象添加到場景時，該對象將出現在世界的起源處。我們保持關節處於選中狀態，然後控制選擇 bubbleRob\_leftWheel。在位置對話框的“位置”選項卡上，我們單擊“應用於”選擇按鈕：這將關節定位在左輪的中心。然後，在“方向”對話框中的“方向”選項卡上，執行相同的操作：這將關節與左輪定向的方向相同。我們將關節重命名為 bubbleRob\_leftMotor。現在，我們在場景層次中雙擊關節的圖標以打開關節屬性對話框。然後，單擊“顯示動態參數”以打開關節動力學屬性對話框。我們啟用電動機，然後選中目標速度為零時鎖定電動機。現在，我們對右馬達重複相同的過程，並將其重命名為 bubbleRob\_rightMotor。現在，我們將左輪連接到左馬達，將右輪連接到右馬達，然後將兩個馬達連接到 bubbleRob。



我們運行模擬，並注意到機器人向後倒下。我們仍然缺少與地板的第三個聯繫點。現在，我們添加一個小的滑塊（或腳輪）。在一個新場景中，我們添加一個直徑為 0.05 的純原始球體，並使該球體可碰撞，可測量，可渲染和可檢測（如果尚未啟用），然後將其重命名為 bubbleRob\_slider。我們在形狀動力學屬性中將 Material 設置為 noFrictionMaterial。為了將滑塊與機器人的其餘部分牢固地鏈接在一起，我們使用[菜單欄->添加->力傳感器]添加了力傳感器對象。我們將其重命名為 bubbleRob\_connection 並將其上移 0.05。我們將滑塊連接到力傳感器，然後復制兩個對象，切換回場景 1 並貼上它們。然後，我們將力傳感器沿絕對 X 軸移動-0.07，然後將其安裝到機器人主體上。如果現在運行仿真，我們會注意到滑塊相對於機器人主體略微移動，這是因為兩個對象彼此碰撞。為了避免在動力學模擬過程中產生奇怪的影響，我們必須通知 CoppeliaSim 兩個對象不會相互碰撞，我們可以通過以下方式進行此操作，在形狀動力學屬性中，對於 bubbleRob\_slider，我們將本地可響應蒙版設置為 00001111，對於 bubbleRob，我們將本地可響應掩碼設置為 11110000。再次運行仿真，我們會注意到兩個對象不再相互干擾。



我們再次運行仿真，發現即使在電機鎖定的情況下，BubbleRob 也會輕微移動。我們還嘗試使用不同的物理引擎運行仿真，結果將有所不同。動態仿真的穩定性與所涉及的非靜態形狀的質量和慣性緊密相關。現在，我們嘗試糾正這種不良影響。我們選擇兩個輪子和滑塊，然後在“形狀動力學”對話框中點擊 3 次  $M = M * 2$ （用於選擇）。效果是所有選定形狀的質量都將乘以 8。我們對 3 個選定形狀的慣性進行相同的操作，然後再次運行仿真：穩定性得到了改善。在關節動力學對話框中，我們將兩個電機的目標速度都設置為 50。我們運行模擬，BubbleRob 現在向前移動並最終掉落在地板上。我們將兩個電機的目標速度項都重置為零。

對象 bubbleRob 是所有對象的基礎，所有對象隨後將形成 BubbleRob 模型。我們將在稍後定義模型。同時，我們要定義代表 BubbleRob 的對象的集合。為此，我們定義了一個收集對象。我們單擊[菜單欄->工具->集合]以打開集合對話框。



在集合對話框中，點擊添加新集合。一個新的集合對象出現在下面的列表中。目前，新添加的集合仍為空（未定義）。在列表中選擇新的收藏項時，在場景層次中選擇 bubbleRob，然後在收藏對話框中點擊“添加”。現在，我們的集合被定義為包含層次結構樹的所有對象（從 bubbleRob 對象開始）（集合的組成顯示在“組成元素和屬性”部分中）。要編輯集合名稱，請點擊它，然後將其重命名為 bubbleRob\_collection。

在此階段，我們希望能夠跟踪 BubbleRob 與任何其他對象之間的最小距離。為此，我們使用[菜單欄->工具->計算模塊屬性]打開距離對話框。

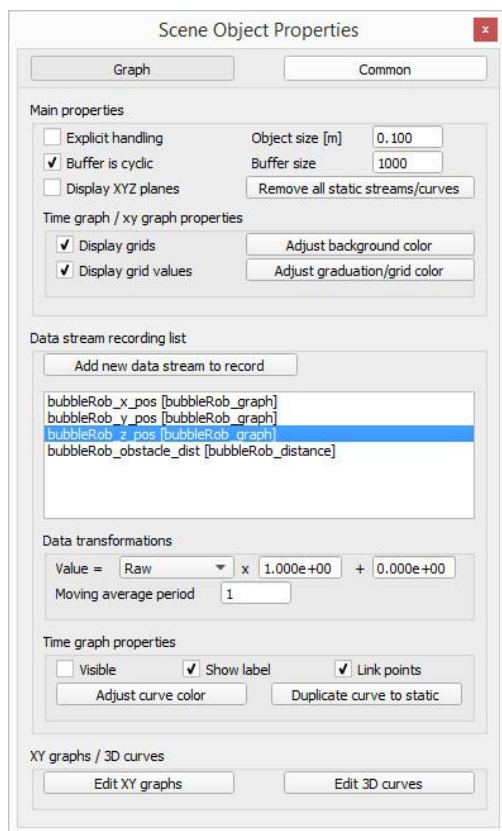


在距離對話框中，點擊“添加新距離對象”並選擇一個距離對 bubbleRob\_collection-場景中所有其他可測量對象。這只是添加了一個距離對象，該距離對象將測量集合 bubbleRob\_collection（即該集中的任何可測量對象）與場景中任何其他可測量對象之間的最小距離。我們通過點擊其名稱將距離對象重命名為 bubbleRob\_distance。我們關閉距離對話框。現在，當我們運行模擬時，我們不會看到任何區別，因為距離對象將嘗試測量（並顯示）BubbleRob 與場景中任何其他可測量對象之間的最小距離段。

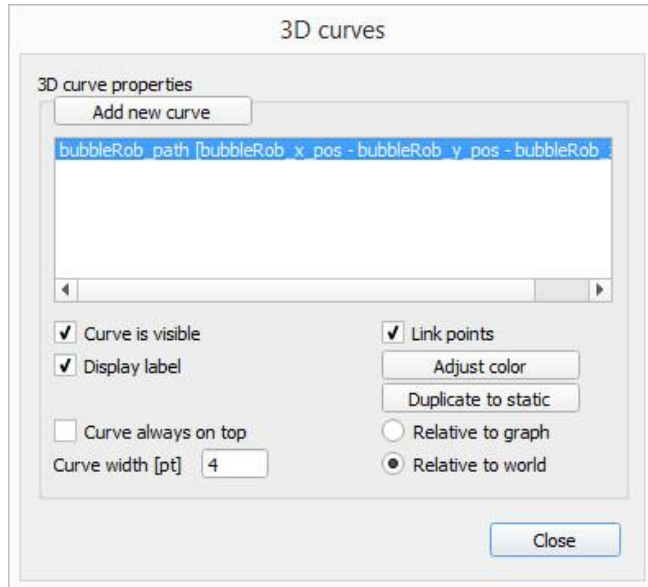
接下來，我們將向 BubbleRob 添加一個圖形對象，以顯示最小距離以上的距離，同時還顯示 BubbleRob 隨時間的軌跡。我們點擊[菜單欄->添加->圖]，並將其重命名為 bubbleRob\_graph。我們將圖形附加到 bubbleRob，並將圖形的絕對坐標設置為（0,0,0.005）。現在，通過在場景層次結構中雙擊其圖標來打開圖形屬性對話框。我們取消選中“顯示 XYZ 平面”，然後點擊“添加新數據流以進行記錄”，然後選擇“對象：數據流類型的絕對 x 位置”，並選擇“bubbleRob\_graph”作為要記錄的對象/項目。數據流記錄列表出現了一個項目。該項目是 bubbleRob\_graph 的絕對 x 坐標的數據流。現在，我們還想記錄 y 和 z 位置：我們以與上述類似的方式添加這些數據流。現在，我們有 3 個數據流，分別表示 BubbleRob 的 x, y 和 z 軌跡。我們將再添加一個數據流，以便能夠跟踪機器人與其環境之間的最小距離，點擊添加新數據流以進行記錄，然後選擇“距離，數據流類型的段長度”和“氣泡 Rob\_distance”作為要記錄的對象/項目。在數據流記錄列表



中,我們現在將 Data 重命名為 bubbleRob\_x\_pos,將 Data0 重命名為 bubbleRob\_y\_pos ,  
將 Data1 重命名為 bubbleRob\_z\_pos , 將 Data2 重命名為 bubbleRob\_obstacle\_dist。  
我們在“數據流”記錄列表中和“時間圖屬性”部分中選擇 bubbleRob\_x\_pos ,取消選中  
“可見”。我們對 bubbleRob\_y\_pos 和 bubbleRob\_z\_pos 都執行相同的操作。這樣,在  
時間圖中只能看到 bubbleRob\_obstacle\_dist 數據流。



接下來,我們將建立一個顯示 BubbleRob 軌跡的 3D 曲線,點擊“編輯 3D 曲線”以打開  
XY 圖形和 3D 曲線對話框,然後點擊“添加新曲線”。在彈出的對話框中,我們為 X 值項  
目選擇 bubbleRob\_x\_pos,為 Y 值項目選擇 bubbleRob\_y\_pos,為 Z 值項目選擇  
bubbleRob\_z\_pos。我們將新添加的曲線從 Curve 重命名為 bubbleRob\_path。最後,我  
們檢查“相對於世界”項目並將“曲線寬度”設置為 4。



我們關閉與圖有關的所有對話框。現在我們將一個電機目標速度設置為 50，運行模擬，然後將看到 BubbleRob 的軌跡顯示在場景中。然後，我們停止仿真並將電動機目標速度重置為零。

我們添加具有以下尺寸的純原始圓柱體（0.1，0.1，0.2）。我們希望此圓柱體是靜態的，但仍會對非靜態的可響應形狀施加一些碰撞響應。為此，我們在形狀動力學屬性中禁用“主體是動態的”。我們還希望圓柱體是可碰撞的，可測量的，可渲染的和可檢測的。我們在對象的公共屬性中執行此操作。現在，在仍然選擇圓柱體的情況下，我們點擊對象平移工具欄按鈕。



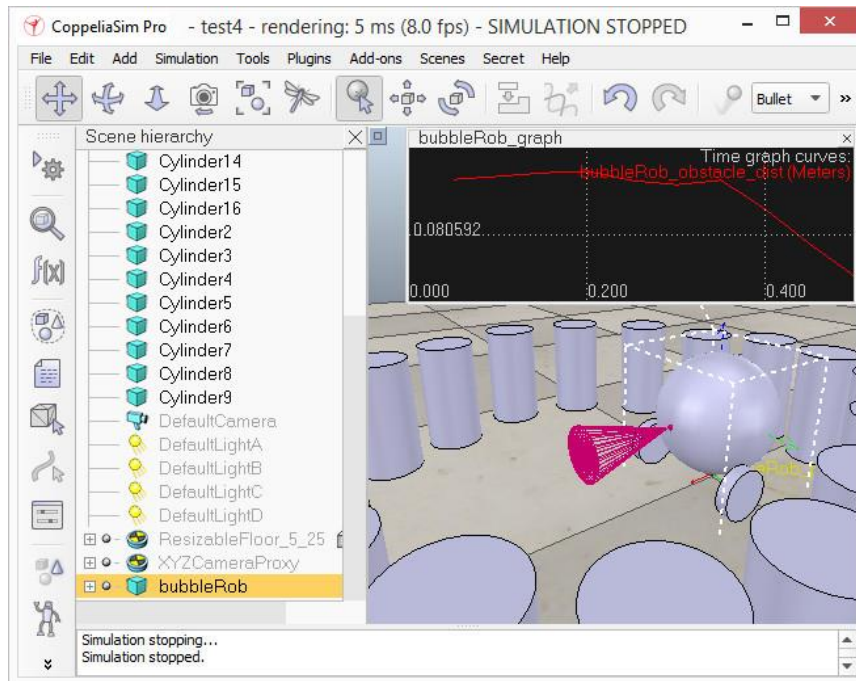
現在我們可以拖動場景中的任何點，圓柱體將跟隨運動。同時始終受約束以保持相同的 Z 坐標。我們複製並貼上圓柱幾次，然後將它們移動到 BubbleRob 周圍的位置。在對象移動期間，按住 Shift 鍵可以執行較小的移動步驟。按住 ctrl 鍵可以在與常規方向正交的方向上移動。完成後，再次選擇相機平移工具欄按鈕。





我們將左馬達的目標速度設置為 50 並運行模擬。現在，圖形視圖顯示了到最近障礙物的距離，並且該距離段在場景中也可見。我們停止模擬並將目標速度重置為零。

現在，我們需要完成 BubbleRob 作為模型定義。我們選擇模型基礎（即對象 bubbleRob），然後選中“對象是模型基礎”，然後選擇“對象/模型可以轉移或接受對象共同屬性中的 DNA”，現在有一個點畫的邊界框包含模型層次結構中的所有對象。我們選擇兩個關節，即接近傳感器和圖形，然後啟用項目“不顯示為內部模型選擇”，然後在同一對話框中點擊“應用於選擇”，模型邊界框現在將忽略兩個關節和接近傳感器。仍在同一對話框中，我們禁用攝像機可見性層，並為兩個關節和力傳感器啟用攝像機可見性層，這有效地隱藏了兩個關節和力傳感器。我們可以隨時修改整個場景的可見性層。要完成模型定義，我們選擇視覺傳感器，兩個輪子，滑塊和圖形，然後啟用“選擇模型基礎”選項：如果現在嘗試在場景中選擇模型中的對象，則整個模型而是選擇，這是一種將單個模型處理和操縱整個模型的便捷方法。此外，這可以防止模型受到意外修改。仍然可以通過在按住 Shift 的同時點擊選擇對象或在場景層次結構中正常選擇它們，來在場景中選擇模型中的單個對象。最後，我們將模型樹折疊到場景層次中。



接下來，我們將在與 BubbleRob 接近傳感器相同的位置和方向上添加視覺傳感器。我們再次打開模型層次結構，然後點擊[菜單欄->添加->視覺傳感器->透視類型]，然後將視覺傳感器連接到接近傳感器，並將視覺傳感器的本地位置和方向設置為 ( 0,0,0 )。我們還確保視覺傳感器不可見，不是模型邊界框的一部分，並且點擊該模型，則會選擇模型。為了自定義視覺傳感器，我們打開其屬性對話框。將“遠裁剪平面”項設置為 1，將“分辨率 x”和“分辨率 y”項設置為 256 和 256。向場景中添加一個浮動視圖，並在新添加的浮動視圖上，點擊[彈出菜單->視圖->將視圖與選定的視覺傳感器關聯]。

通過點擊[菜單欄->添加->關聯的子腳本->非線程]，將非線程子腳本附加到視覺傳感器。

我們點擊場景層次結構中視覺傳感器旁邊出現的小圖標，這將打開我們剛剛添加的子腳本。

我們將以下代碼複製並粘貼到腳本編輯器中，然後將其關閉。

```

function sysCall_vision(inData)
    simVision.sensorImgToWorkImg(inData.handle) -- copy the vision sensor image to the work image
    simVision.edgeDetectionOnWorkImg(inData.handle,0.2) -- perform edge detection on the work image
    simVision.workImgToSensorImg(inData.handle) -- copy the work image to the vision sensor image buffer
end

function sysCall_init()
end

```

為了能夠看到視覺傳感器的圖像，我們開始模擬，然後再次停止。

我們場景所需的最後一件事是一個小的子腳本，它將控制 BubbleRob 的行為。我們選擇

bubbleRob 並點擊[菜單欄->添加->關聯的子腳本->非線程]。我們點擊場景層次結構中

bubbleRob 名稱旁邊顯示的腳本圖標，然後將以下代碼複製並粘貼到腳本編輯器中，然後

將其關閉。

```

function speedChange_callback(ui,id,newVal)
    speed=minMaxSpeed[1]+(minMaxSpeed[2]-minMaxSpeed[1])*newVal/100
end

function sysCall_init()
    -- This is executed exactly once, the first time this script is executed
    bubbleRobBase=sim.getObjectAssociatedWithScript(sim.handle_self) -- this is bubbleRob's handle
    leftMotor=sim.getObjectHandle("bubbleRob_leftMotor") -- Handle of the left motor
    rightMotor=sim.getObjectHandle("bubbleRob_rightMotor") -- Handle of the right motor
    noseSensor=sim.getObjectHandle("bubbleRob_sensingNose") -- Handle of the proximity sensor
    minMaxSpeed=(50*math.pi/180,300*math.pi/180) -- Min and max speeds for each motor
    backUntilTime=-1 -- Tells whether bubbleRob is in forward or backward mode
    -- Create the custom UI:
    xml = '<ui title=" '..sim.getObjectHandle(bubbleRobBase).. ' speed' closeable="false" resizable="false" activate="false">'....[
        <slider minimum="0" maximum="100" onchange="speedChange_callback" id="1"/>
        <label text="" style="margin-left: 300px;"/>
    ]>
    ui=simUI.create(xml)
    speed=(minMaxSpeed[1]+minMaxSpeed[2])*0.5
    simUI.setSliderValue(ui,1,100*(speed-minMaxSpeed[1])/(minMaxSpeed[2]-minMaxSpeed[1]))
end

function sysCall_actuation()
    result=sim.readProximitySensor(noseSensor) -- Read the proximity sensor
    -- If we detected something, we set the backward mode:
    if (result>0) then backUntilTime=sim.getSimulationTime()+4 end

    if (backUntilTime<sim.getSimulationTime()) then
        -- When in forward mode, we simply move forward at the desired speed
        sim.setJointTargetVelocity(leftMotor,speed)
        sim.setJointTargetVelocity(rightMotor,speed)
    else
        -- When in backward mode, we simply backup in a curve at reduced speed
        sim.setJointTargetVelocity(leftMotor,-speed/2)
        sim.setJointTargetVelocity(rightMotor,-speed/8)
    end
end

function sysCall_cleanup()
    simUI.destroy(ui)
end

```

我們運行模擬 BubbleRob 現在在嘗試避開障礙物的同時向前移動(以非常基本的方式)。

在模擬仍在運行時，更改 BubbleRob 的速度，然後將其複製/貼上幾次。在模擬仍在運行

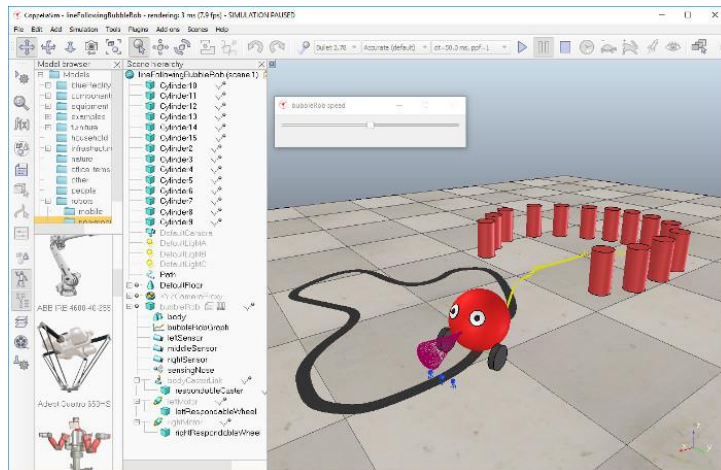
時，也嘗試擴展其中的一些。請注意，根據環境的不同，最小距離計算功能可能會嚴重降低

仿真速度。您可以通過選中/取消選中“啟用所有距離計算”項來在“距離”對話框中打開

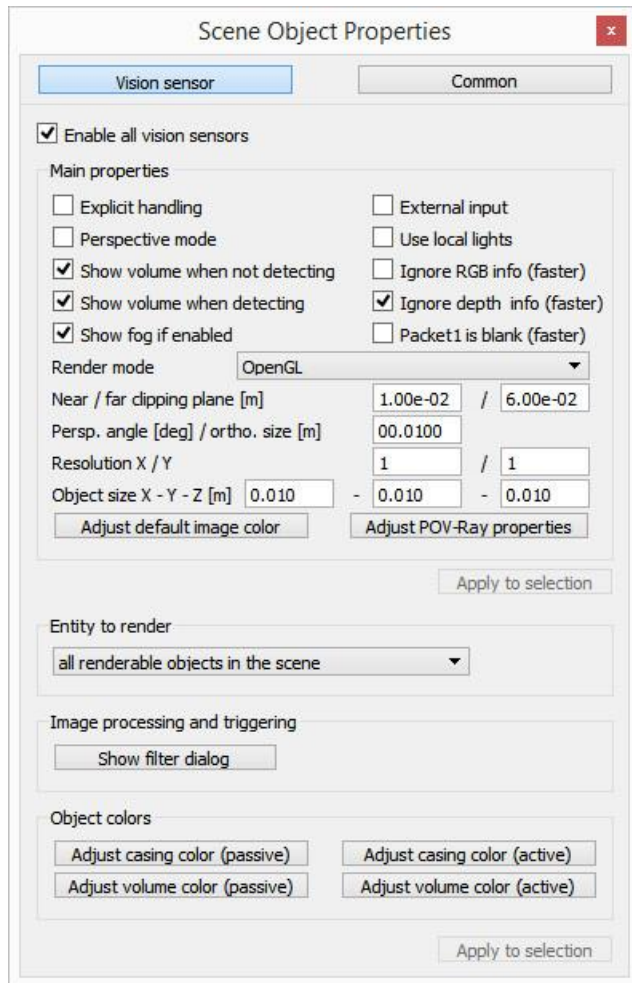
和關閉該功能。

## 章節:Line following BubbleRob tutorial

在 CoppeliaSim 的安裝文件夾中的 tutorials / BubbleRob 中加載第一個 BubbleRob 教程的場景。

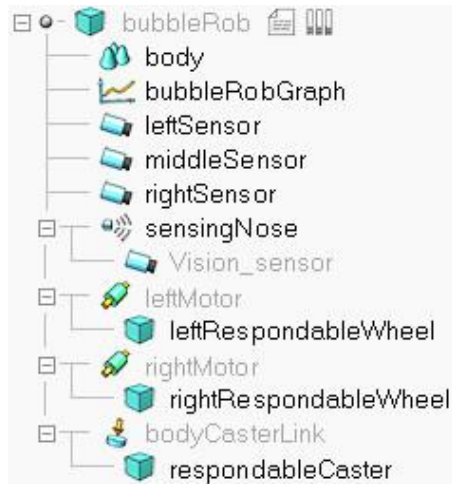


我們首先創建 3 個視覺傳感器中的第一個，並將其附加到 bubbleRob 對象。選擇[菜單欄->添加->視覺傳感器->正交類型]。通過雙擊場景層次中新創建的視覺傳感器圖標來編輯其屬性，並更改參數以反映。



視覺傳感器必須面向地面，因此選擇它，然後在“方向”對話框中的“方向”選項卡上，將“Alpha-Beta-Gamma”項設置為[180; 0; 0]。

我們有幾種可能性可以讀取視覺傳感器。由於我們的視覺傳感器只有一個像素，並且操作簡單，因此我們只需查詢視覺傳感器讀取的圖像的平均強度值即可。對於更複雜的情況，我們可以設置視覺回調函數。現在，複製並粘貼視覺傳感器兩次，並將其名稱調整為 leftSensor，middleSensor 和 rightSensor。



讓我們正確放置傳感器。為此，使用位置對話框，在位置選項卡上，並設置以下絕對坐標：

- 左傳感器：[0.2; 0.042; 0.018]
- 中間傳感器：[0.2; 0; 0.018]
- 右傳感器：[0.2; -0.042; 0.018]

現在讓我們修改環境。我們可以移去 BubbleRob 前面的幾個圓柱體。接下來，我們將構建機器人將嘗試遵循的路徑。從現在開始最好切換到頂視圖：通過頁面選擇器工具欄按鈕選擇頁面 4。然後單擊[菜單欄->添加->路徑->圓圈類型]。使用鼠標啟用對象移動。

選擇路徑（並且只有路徑）後，按住 Ctrl 並單擊其控制點之一。然後可以將它們拖動到正確的位置。

選擇路徑後，進入路徑編輯模式。在那裡，您可以靈活地調整各個路徑控制點。

一旦對路徑的幾何形狀滿意（您隨時可以在以後的階段對其進行修改），請選擇它，然後在路徑屬性中取消選中“顯示點的方向”，“顯示路徑線”和“顯示路

徑上的當前位置”。然後單擊顯示路徑整形對話框。這將打開路徑整形對話框。

單擊啟用路徑整形，將類型設置為水平線段，並將縮放因子設置為 4.0。最後將顏色調整為黑色。我們必須對路徑進行最後一個重要的調整：當前，路徑的 z 位置與地板的 z 位置重合。結果是有時我們會看到路徑，有時會看到地板（這種效果在 openGL 行話中被稱為 “z-fighting”）。這不僅影響我們所看到的，而且還會影響視覺傳感器所看到的。為了避免與 z 戰鬥有關的問題，只需將路徑對象的位置向上移動 0.5 毫米即可。

最後一步是調整 BubbleRob 的控制器，使其也將遵循黑色路徑。

## 章節:External controller tutorial

在 CoppeliaSim 中，有幾種方法可以控制機器人或仿真：

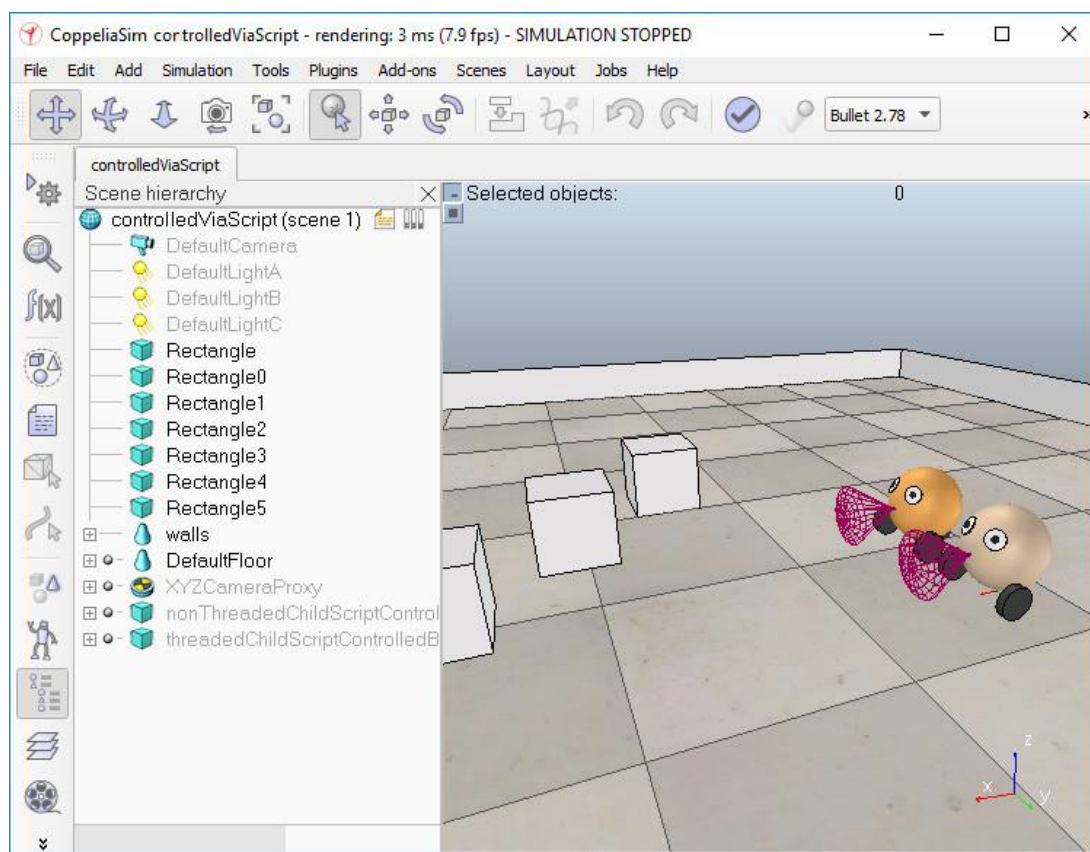
- 方法一是編寫一個子腳本來處理給定機器人或模型的行為。這是最方便的方法，因為子腳本直接附加到場景對象，它們將與相關的場景對象一起復制，它們不需要使用外部工具進行任何編譯，它們可以在線程或非線程模式下運行，它們可以通過自定義 Lua 函數或 Lua 擴展庫進行擴展。使用子腳本的另一個主要優點是：與本節中提到的後 3 種方法（即使用常規 API）一樣，沒有通信延遲，並且子腳本是應用程序主線程的一部分（固有的同步操作）。
- 方法二是編寫插件。插件機制允許使用回調機制，自定義 Lua 函數註冊，當然還可以訪問外部函數庫。插件通常與子腳本結合使用（例如，



插件註冊自定義的 Lua 函數，當從子腳本中調用時，該 Lua 函數將回調特定的插件函數）。使用插件的主要優勢還在於，與本節中提到的後 3 種方法（即使用常規 API）一樣，沒有通信延遲，並且插件是應用程序主線程的一部分（固有的同步操作）。插件的缺點是：它們的編程更加複雜，並且也需要使用外部編譯。

- 方法三是編寫依賴於遠程 API 的外部客戶端應用程序。如果您需要從外部應用程序，機器人或另一台計算機運行控制代碼，這是一種非常便捷的方法。這也使您可以使用與運行真實機器人完全相同的代碼來控制仿真或模型（例如虛擬機器人）。
- 方法四是通過 ROS 節點。ROS 與遠程 API 相似，是使多個分佈式進程相互通信的便捷方法。儘管遠程 API 非常輕巧且快速，但它僅允許與 CoppeliaSim 通信。另一方面，ROS 允許幾乎將任意數量的進程相互連接，並且提供了大量兼容的庫。但是，它比遠程 API 重並且更複雜。
- 方法五是通過 BlueZero (BØ) 節點。與 ROS 類似，BlueZero 是使多個分佈式進程相互通信的一種便捷方法，並且是一種輕量級的跨平台解決方案。
- 方法六是編寫一個外部應用程序，該應用程序通過各種方式（例如管道，套接字，串行端口等）與 CoppeliaSim 插件或 CoppeliaSim 腳

本進行通信。選擇編程語言（可以是任何一種語言）和靈活性是兩個主要優點。同樣，控制代碼也可以在機器人或其他計算機上運行。



## 章節:Simulation dialog

可以通過[菜單欄->模擬->模擬設置]或點擊以下工具欄按鈕來訪問模擬對話框





- **時間**: 模擬時間。每次執行主腳本時, 仿真時間都會增加仿真時間步長。  
  
使用較大的時間步會導致快速但不準確/不穩定的仿真。另一方面, 較小的時間步長 (通常) 會導致更精確的仿真, 但是會花費更多時間。強烈建議保留默認時間步長。
- **每幀模擬遍數**: 一個渲染遍的模擬遍數。值為 10 表示刷新屏幕之前, 主腳本已執行 10 次。如果您的圖形卡較慢, 則可以選擇僅顯示兩幅中的一幅。
- **當仿真時間高於以下時間時暫停**: 允許指定暫停仿真的仿真時間。
- **腳本錯誤暫停**: 如果啟用, 則在腳本錯誤發生時暫停仿真。
- **模擬開始時全屏**: 如果啟用, 則模擬以全屏模式開始。請注意, 在全屏模式下, 對話框和消息將不會出現或不可見, 只有鼠標左鍵處於活動狀態。因此, 僅在正確配置場景並最終確定場景後才建議使用該模式。可以使用 esc 鍵保留全屏模式, 並在仿真過程中通過布爾參數

sim\_booparam\_fullscreen 進行切換。 Under Linux 和 MacOS 可能僅部分支持全屏模式,並且在某些系統上切換回普通模式可能會失敗。

- **實時仿真，倍增係數**：如果選擇，則仿真時間將嘗試跟隨實時。 X 的乘數將使仿真運行比實時快 X 倍。
- **在落後時嘗試趕上**：在實時仿真過程中,仿真時間可能無法實時跟蹤(例如,由於某些瞬間繁重的計算)。在這種情況下,如果選中此復選框,則模擬時間將嘗試趕上損失的時間,從而明顯加快速度。
- **將場景重置為初始狀態**：選中後,所有對象都將重置為其初始狀態：包括對象的局部位置,局部方向及其父對象(只要未進行其他修改(例如,縮放)),以及路徑的固有位置,浮動視圖的位置和大小等。這意味著除非進行了重大更改(形狀縮放,對象移除等),否則下一次模擬運行將以與上一次相同的方式執行。
- **刪除新對象**：選中後,在仿真運行期間添加的場景對象將在仿真結束時被刪除。

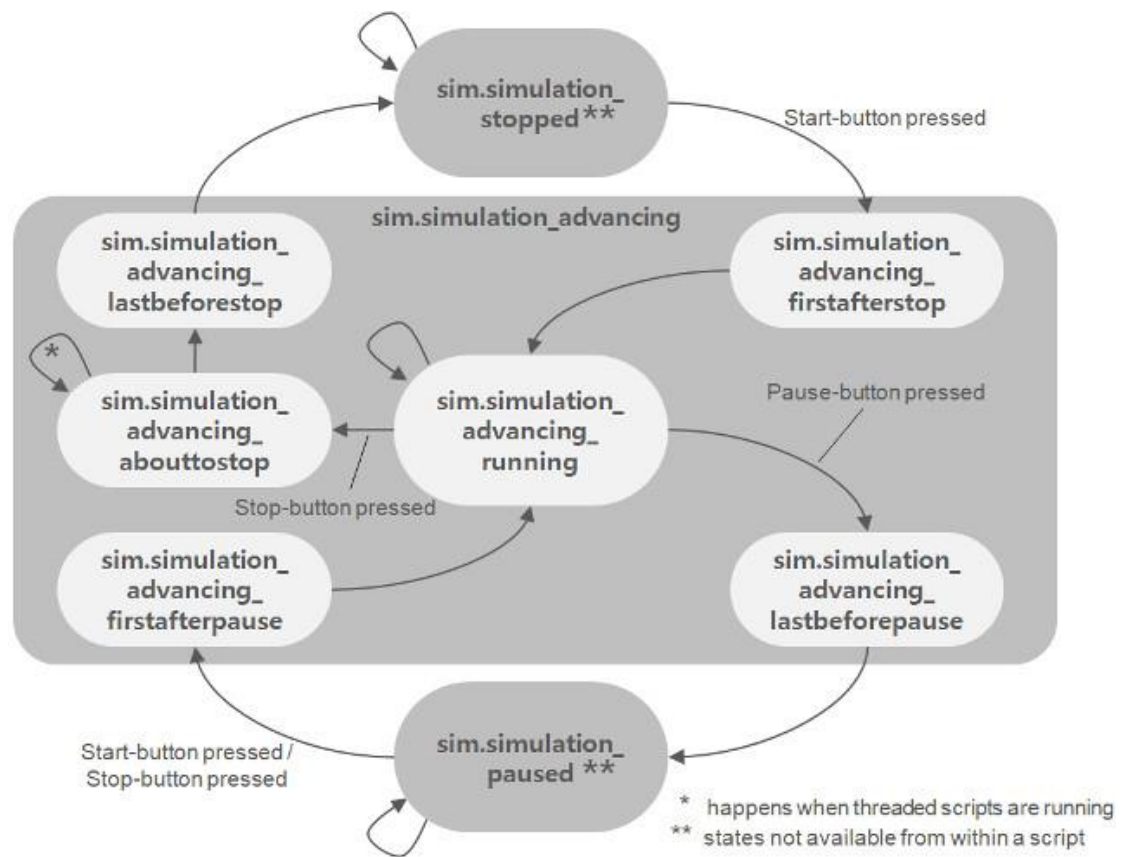
## 章節:Simulation

### 模擬

使用[菜單欄->模擬->開始/暫停/停止模擬]或通過相關的工具欄按鈕來啟動,暫停和停止 CoppeliaSim 中的模擬：



在內部，模擬器將使用其他中間狀態，以正確告知腳本或程序接下來將發生的情況。

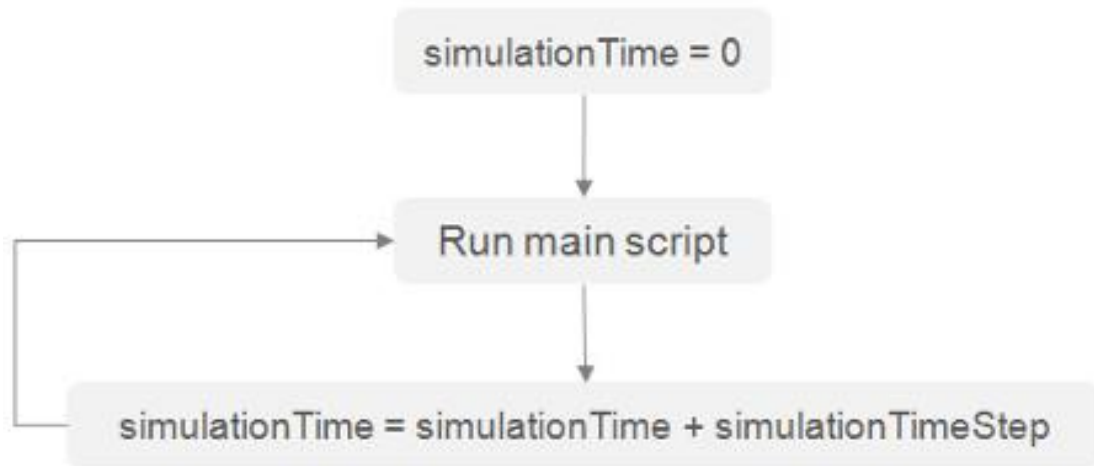


腳本和程序應始終根據當前系統調用功能以及可能的模擬狀態進行反應，以便正確運行。優良作法是將每個控制代碼至少分為 4 個系統調用函數

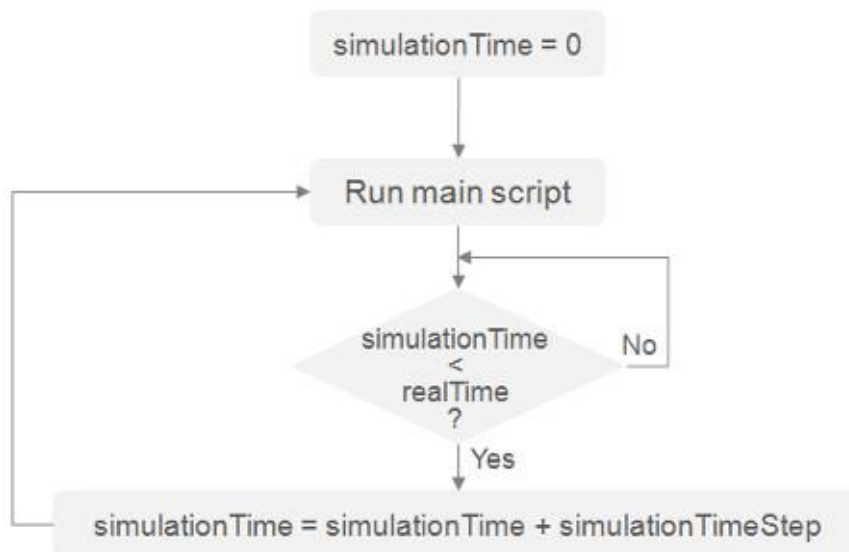
- **初始化函數**：僅在腳本初始化時調用該函數
- **激活函數**：應在發生激活時調用該函數。
- **傳感功能**：應在傳感發生時調用此函數。
- **清理函數**：該函數在腳本未初始化之前被調用。

## 仿真循環

模擬器通過以恆定的時間步長推進模擬時間來進行操作。



通過嘗試使仿真時間與實時保持同步來支持實時仿真：



以下是一個非常簡化的主客戶端應用程序：

```

void initializationCallback
{
    // do some initialization here
}

void loopCallback
{
    if ( (simGetSimulationState() &sim_simulation_advancing) !=0 )
    {
        if ( (simGetRealTimeSimulation() !=1) || (simIsRealTimeSimulationStepNeeded() ==1) )
        {
            if ( (simHandleMainScript() &sim_script_main_script_not_called) ==0)
                simAdvanceSimulationByOneStep();
        }
    }
}

void deinitializationCallback
{
    // do some clean-up here
}

```

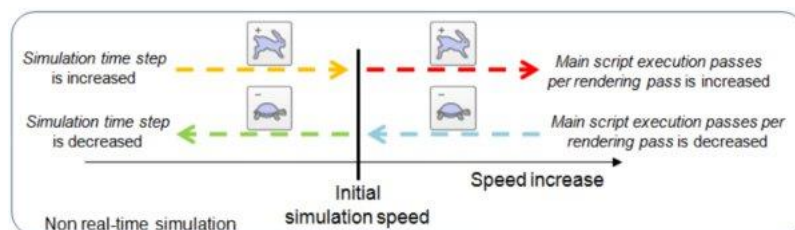
取決於模擬的複雜性，計算機的性能和模擬設置，實時模擬可能並不總是可能的。

## 仿真速度

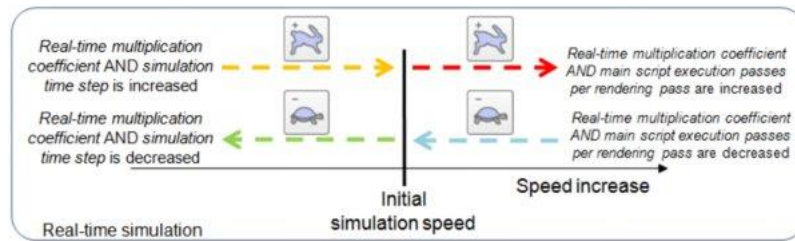
在非實時仿真中，仿真速度主要取決於兩個因素，仿真時間長和一個渲染通道的仿真通道數量。在實時仿真的情況下，仿真速度主要取決於實時乘法係數，而且在一定程度上取決於仿真時間步長。由於計算機的計算能力有限，因此無法進行仿真。在模擬過程中，可以使用以下工具欄按鈕來調整模擬速度：



以某種方式調整模擬速度，以使初始模擬時間步長永遠不會增加。以下兩個圖說明了仿真速度調整機制：







默認情況下，每個模擬週期由以下順序操作組成：

- 執行主腳本
- 渲染場景

## 螺紋渲染

渲染操作將始終增加仿真週期的持續時間，從而也降低了仿真速度。可以定義每個場景渲染的主腳本執行次數，但這在某些情況下還不夠，因為渲染仍然會減慢每個第  $x$  個模擬週期的時間。在這種情況下，可以通過用戶設置或以下工具欄按鈕激活線程渲染模式：



激活線程渲染模式後，模擬週期將僅包含在執行主腳本中，因此模擬將以最大速度運行。渲染將通過不同的線程進行，並且不會減慢模擬任務的速度。然而，必須考慮缺點。

- 渲染將與模擬循環異步進行，並且可能會出現視覺故障。
- 錄像機將無法以恆定速度運行。
- 应用程序的穩定性可能會降低。

- 某些操作需要等待渲染線程完成工作才能執行，反之亦然。在那些情況下，循環可能比順序渲染模式花費更多的時間。