CoppeliaSim

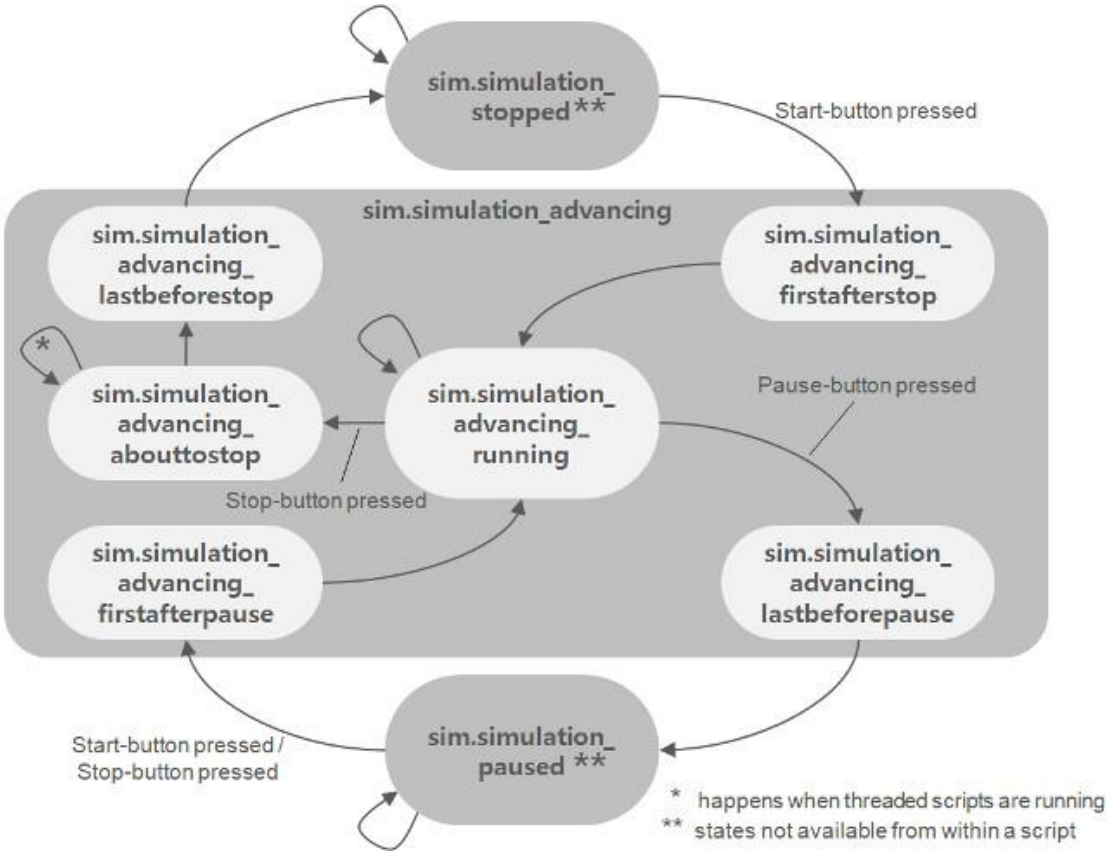
CoppeliaSim 用於快速算法開發、工廠自動化仿真、快速原型製作和驗證、機器人技術相關的教育、遠程監控、安全性雙重檢查以及數字孿生等等。

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：

[模擬開始/暫停模擬/停止模擬]

在內部之模擬器將使用其他中間狀態，以正確告知程式或程序接下來將發生的情況。 以下狀態圖說明了模擬器的內部狀態：



[模擬狀態圖]

程式和程序應根據當前系統呼叫功能以及可能的模擬狀態進行反應，以便正確運行。 最好的作法是將每個控制程式碼至少分為 4 個系統呼叫函數（例如，用於非線程子程式）：

初始化函數：sysCall_init：僅在程式初始化時才呼叫該函數。

激活函數：sysCall_actuation：應在發生激活時呼叫該函數。

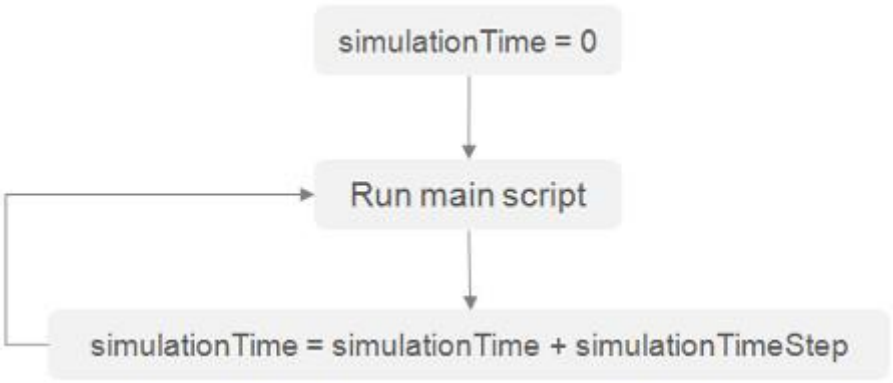
感測功能：sysCall_sensing：應在感測發生時呼叫此函數。

清理函數：sysCall_cleanup：該函數在程式未初始化之前被調呼叫（例如在仿真結束時或程式被銷毀時）。

有關如何安排典型程式的示例，請參閱主程式、子程式和自定義程式頁面。

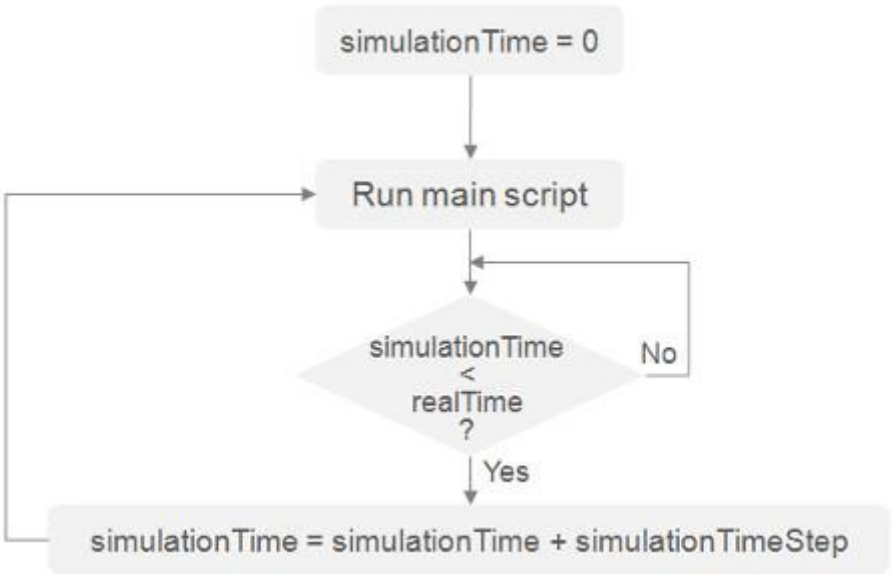
模擬循環

模擬器通過固定的時間步長推進模擬時間來進行操作。 下圖說明了主要的模擬循環：



[主模擬循環]

通過嘗試使模擬時間與即時保持同步來達到即時模擬：



[即時模擬循環]

以下是一個非常精簡的主客戶端應用程序（為保持清晰，已省略了消息、插

件處理和其他詳細信息）：

```
void initializationCallback
{
    // do some initialization here
}

void loopCallback
{
    if
( (simGetSimulationState()&sim_simulation_advanc
ing)!=0 )
    {
        if
( (simGetRealTimeSimulation()!=1)||(simIsRealTim
eSimulationStepNeeded()==1) )
        {
            if
((simHandleMainScript()&sim_script_main_script_n
ot_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，即時模擬並不總是可能的。

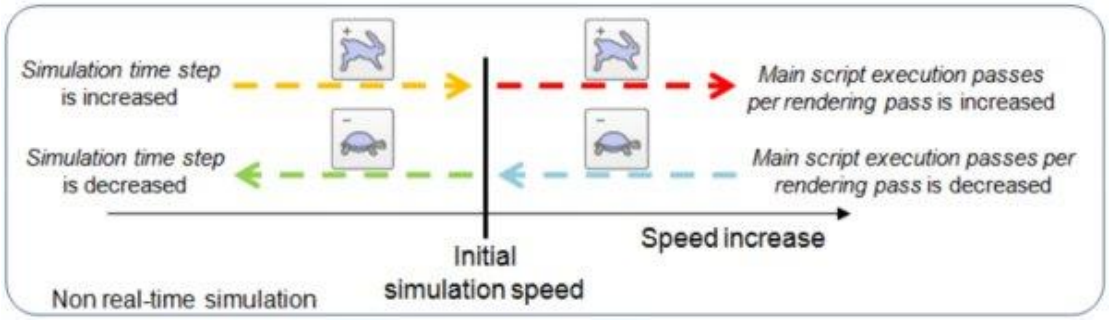
模擬速度

在非即時模擬中，模擬速度（感知速度）主要取決於兩個因素：模擬時間步

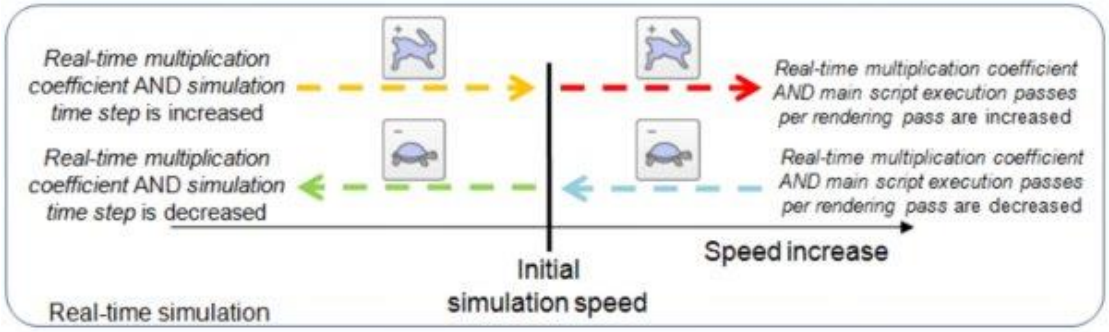
長和一個渲染通道的模擬通道數量（有關更多詳細信息，請參見模擬對話框）。 在即時模擬的情況下，模擬速度主要取決於即時乘法係數，而且在一定程度上取決於模擬時間步長（太短的模擬時間步長可能與即時時間不相容）。 由於計算機的計算能力有限，因此無法進行模擬。 在模擬過程中，可以使用以下工具欄按鈕來調整模擬速度：

[模擬速度調整工具欄按鈕]

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（但這可能因而導致機制中斷）。 以下兩個圖說明了模擬速度調整機制：



[非即時模擬的模擬速度調整機制]



[用於即時模擬的模擬速度調整機制]

默認情況下，每個模擬週期由以下順序操作組成：

- 執行主程式
- 渲染場景

線程渲染

渲染操作將增加模擬週期的持續時間，也降低了模擬速度。 可以定義每個場景渲染的主程式執行次數（請參閱後面的內容），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（這可能會限制即時性）。 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：

[線程渲染工具欄按鈕]

激活線程渲染模式後，模擬週期將僅包含在執行主程式中，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。 激活線程渲染後：

- 渲染將與模擬循環異步進行，並且可能會出現視覺故障
- 錄影將無法以固定速度運行（可能會跳過某些模擬步驟）
- 應用程序的穩定性可能會降低某些操作（例如消除對像等）需要等待渲染

線程完成工作才能執行，反之。 在那些情況下，循環可能比順序渲染模式花費更多的時間。

模擬對話框

可以通過[菜單欄->模擬->模擬設置]或點擊以下工具欄按鈕來呼叫模擬對話框：

[模擬工具欄按鈕]

[模擬設置對話框]

- **時間步驟：**模擬時間步驟。每次執行主程式時，模擬時間都會增加模擬時間步長。使用較長的時間步驟會產生快速但不准確/不穩定的模擬。另一方面，較短的時間步長通常會產生更精確的模擬，但是會花費更多時間。強

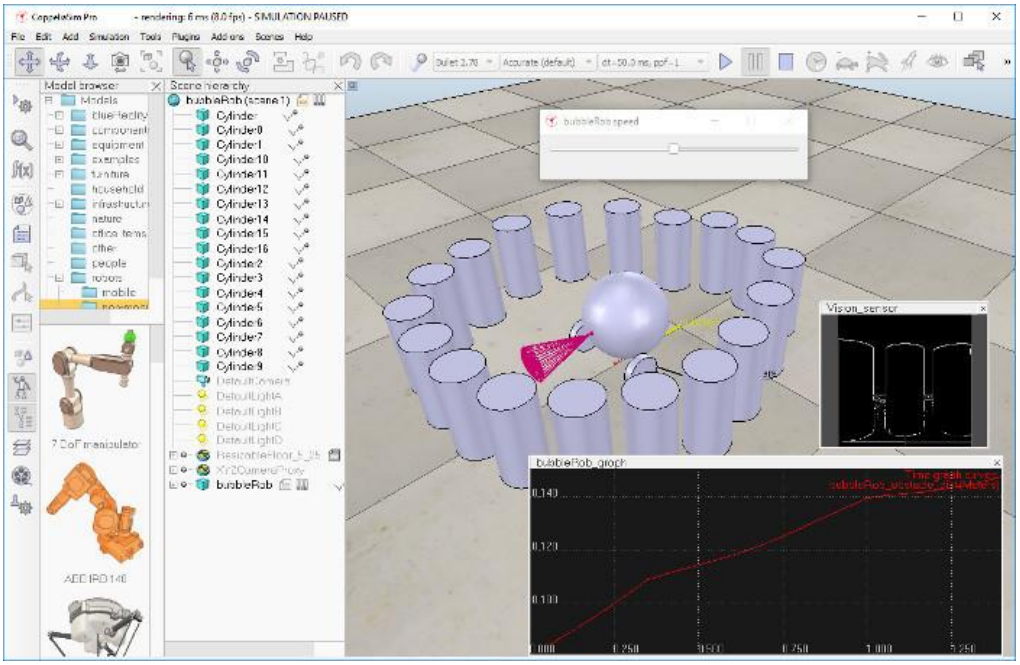
烈建議保留默認時間步長。

- **每幀數模擬遍數（ppf）：**一個渲染階段的模擬遍數。值為 10 表示刷新屏幕之前，主程式已執行 10 次（10 個模擬步驟）。如果您的顯示卡較慢，則可以選擇僅顯示兩幅中的一幅。
- **當模擬時間高於以下時間時暫停：**允許指定暫停模擬的模擬時間（例如，能夠在特定模擬時間分析某些結果）。
- **程式錯誤暫停：**如果啟用，則在程式錯誤發生時暫停模擬。
- **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隱藏，只有鼠標左鍵處於活動狀態才會顯示。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，並在模擬過程中通過布爾參數 *sim_booparam_fullscreen* 進行切換。 Unler Linux 和 MacOS 可能僅部分支援全屏模式，並且在某些系統上切換回普通模式可能會失敗。
- **即時模擬，倍增係數：**如果啟用，則模擬時間將嘗試即時跟隨。 X 的乘數將使模擬運行比即時快 X 倍。
- **在落後時嘗試趕上：**在即時模擬過程中，模擬時間可能無法實時跟踪（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（例如，當計算負載再次減少時），從而明顯加快速度。

- **將場景重置為初始狀態：**如果啟用，所有對像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。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。此項目將忽略一些次要設置。
- **刪除新對象：**如果啟用，在模擬運行期間添加的場景對象將在模擬結束時被刪除。

BubbleRob 教程

本教程將在設計簡單的移動機器人 BubbleRob 時嘗試介紹很多 CoppeliaSim 功能。 與本教程相關的 CoppeliaSim 場景文件位於 CoppeliaSim 的安裝文件夾的 tutorials / BubbleRob 文件夾中。 下圖說明了設計的模擬場景：



由於本教程將跨越許多不同的方面，因此也需學習其他教程，主要是有關構建模擬模型的教程。 首先，重新啟動 CoppeliaSim， 模擬器顯示默認場景，將從 BubbleRob 的主體開始。

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。將 **X 尺寸**項目調整為 0.2，然後點擊**確定**。在默認情況下，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（因為已啟用**創建動態且可響應的形狀**）。這代表 BubbleRob 的主體會掉落並且能夠對與其他可響應形狀的碰撞做出反應（即由物理引擎模擬）。可以看到這是形狀動力學屬性：啟用了**主體可響應**和**主體是動態**項目。開始模擬（通過工具欄按鈕，或在場景窗口中按<control-space>），然後複製並貼上創建的球體（使用[菜單欄->編輯->複製所選物體]，然後[菜單欄->編輯->粘貼緩衝區]，或者先按<control-c>，再按<control-v>）：這兩個球將對碰撞做出反應並滾動。停止模擬：重複的球體將自動刪除，可以在模擬對話框中修改此默認行為。

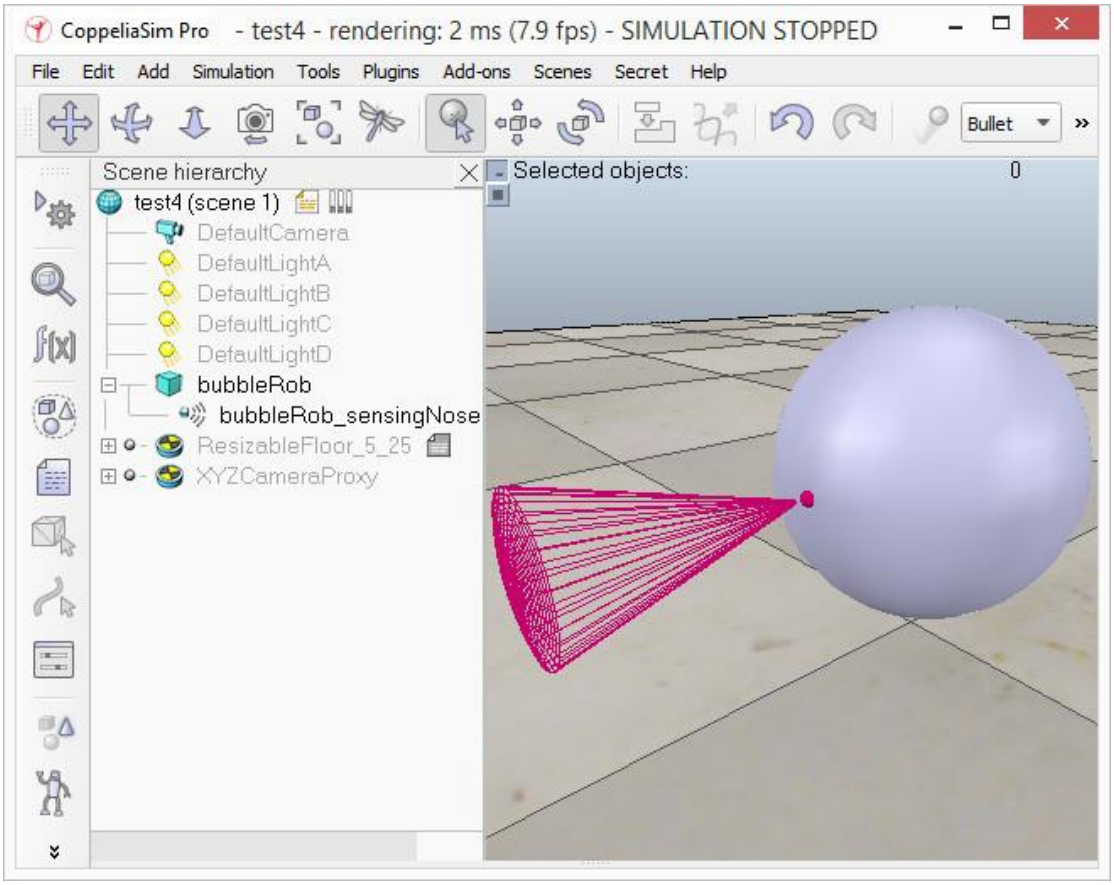
我們還希望 BubbleRob 的主體可以被其他計算模塊（例如最小距離計算模塊）使用。 因此，如果尚未啟用，則在該形狀的物體公共屬性中啟用**可碰撞**、**可測量**、**可渲染**、**可檢測**。 如果需要，還可以在形狀屬性中更改球體的視覺外觀。

現在，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 **Z 軸**輸入 0.02，確保將**相對於**項設置為 **World**，然後點擊翻譯選

擇， 這會將所有選定物體沿絕對 Z 軸平移 2 cm，並有效地將球體抬高了一點。 在場景層次結構中，點擊球體的名稱，以便可以編輯其名稱，接著輸入 bubbleRob，然後按 Enter。

接下來，將添加一個感測器，以便 BubbleRob 知道它何時接近障礙物：選擇[菜單欄->添加->接近感測器->圓錐類型]。在**方向**選項卡上的方向對話框中，為 **Y 軸範圍**和 **Z 軸範圍**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，接近感測器已相對於 BubbleRob 的主體正確定位。在場景層次中點擊感測器的圖標以打開其屬性對話框，接著點擊顯示體積參數以打開感測器體積對話框，然後將偏移量調整為 0.005，角度調整為 30，範圍調整為 0.15。然後在感測器屬性中，點擊**顯示檢測參數**，這將打開感測器檢測參數對話框。如果距離小於則取消選中**不允許檢測**，然後再次關閉該對話框。在場景層次結構中，點擊接近感測器的名稱，以便可以編輯其名稱，輸入 bubbleRob_sensingNose 並按返回鍵。

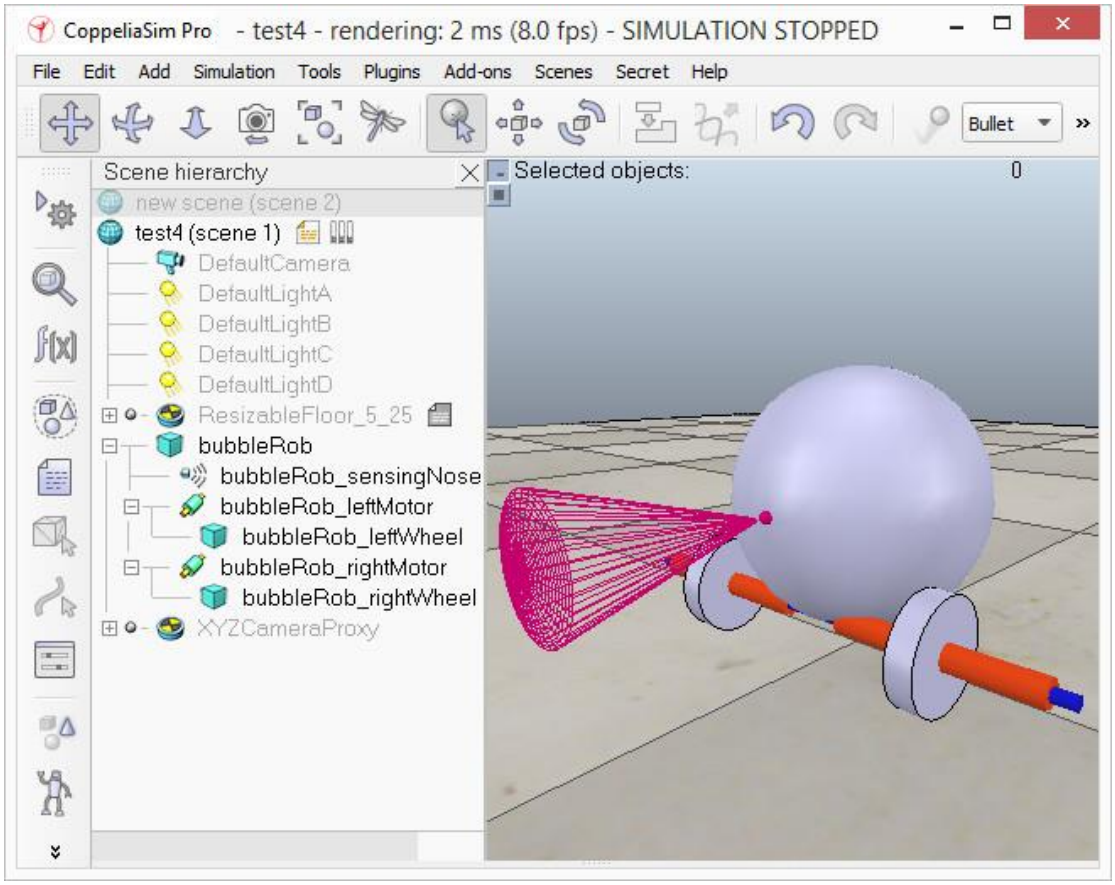
選擇 bubbleRob_sensingNose，然後按住 Control 鍵選擇 bubbleRob，然後點擊[菜單欄->編輯->將上一個選定的物體設為父物體]。這會將感測器連接到機器人的身體，還可以將 bubbleRob_sensingNose 拖動到場景層次中的 bubbleRob 上。如圖：



[感測器連接到 bubbleRob 的主體]

接下來，將創建 BubbleRob 的車輪，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。 添加一個尺寸為（0.08,0.08,0.02）的圓柱體。 對於 BubbleRob 的主體，如果尚未啟用，則在該圓柱的物體通用屬性中啟用**可碰撞**、**可測量**、**可渲染**、**可檢測**。 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），接著將名稱更改為 bubbleRob_leftWheel，然後複製並貼上車輪，並將複製的絕對 Y 坐標設置為-0.1。 我們將副本重命名為 bubbleRob_rightWheel，選擇兩個輪子並複製，然後切換回場景 1，然後貼上車輪。

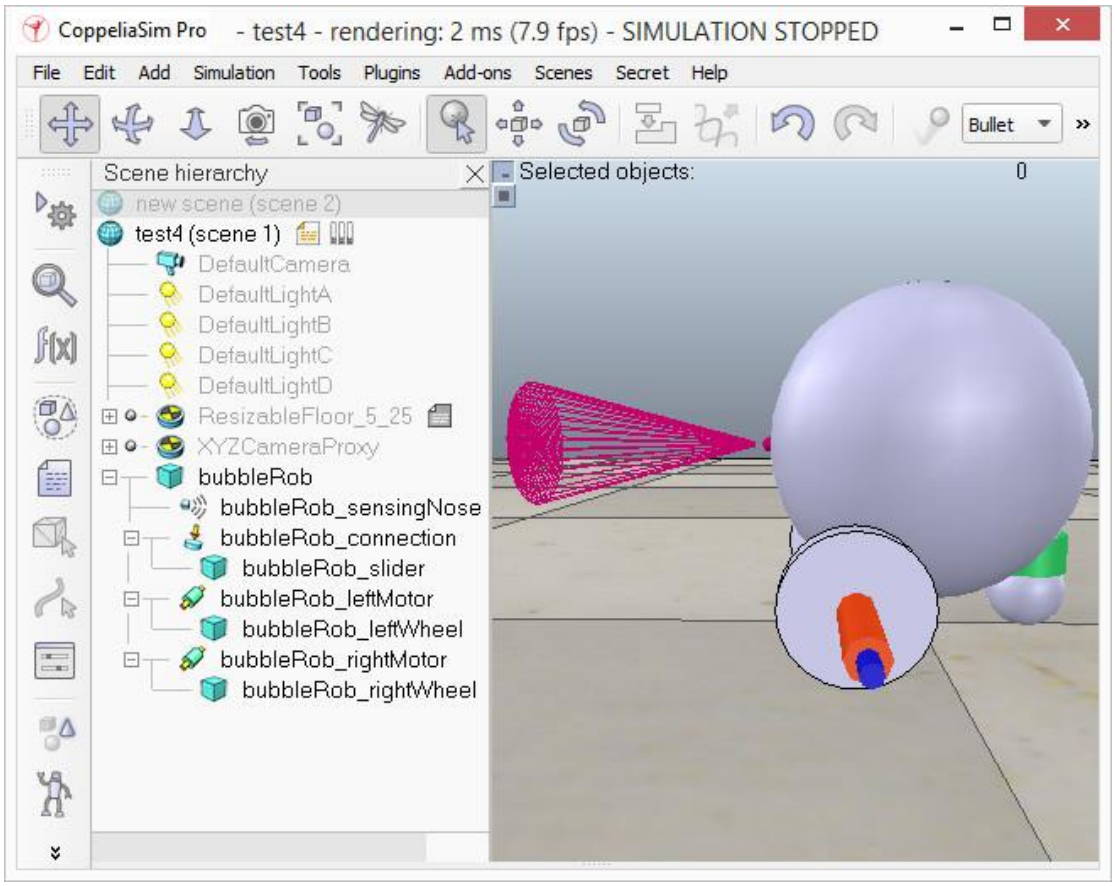
現在，需要為車輪添加連桿（或發動器）。我們點擊[菜單欄->添加->連桿->旋轉]將旋轉連桿添加到場景。在大多數情況下，將新物體添加到場景時，該物體將出現在 Wrold 的起源處。保持連桿處於選中狀態，然後控制選擇 bubbleRob_leftWheel，在位置對話框的**位置**選項卡上，點擊**應用於**選擇按鈕：這將連桿定位在左車輪的中心。然後在方向對話框中的**方向**選項卡上，執行相同的操作：這將連桿與左輪定向的方向相同。我們將連桿重命名為 bubbleRob_leftMotor。在場景層次中點擊連桿的圖標以打開連桿屬性對話框，然後點擊**顯示動態參數**以打開連桿動力學屬性對話框。接著**啟用發動器**，選中**目標速度為零時鎖定發動器**，然後對右馬達重複相同的過程，並將其重命名為 bubbleRob_rightMotor。接下來將左車輪連接到左馬達，將右車輪連接到右馬達，然後將兩個馬達連接到 bubbleRob。如圖：



[感測器器與發動器、車輪]

運行模擬，並注意到機器人向後倒下，因為仍然缺少與地板的第三個聯繫點。接下來需添加一個小的滑塊（或腳輪）。在一個新場景中，添加一個直徑為 0.05 的球體，並使該球體**可碰撞**、**可測量**、**可渲染**、**可檢測**（如果尚未啟用），然後將其重命名為 bubbleRob_slider。在形狀動力學屬性中將**材料**設置為**無摩擦材料**。為了將滑塊與機器人的其餘部分牢固地鏈接在一起，需使用[菜單欄->添加->力感測器]添加了力感測器。將其重命名為 bubbleRob_connection 並將其上移 0.05。接著將滑塊連接到力感測器，然後複製兩個力感測器，切換回場景 1 並貼上。然後將力感測器沿絕對 X 軸移動-0.07，並將其安裝到機器人主體上。如果現在運行模擬，要注意到滑塊相對於

機器人主體略微移動：這是因為兩個物體（即 bubbleRob_slider 和 bubbleRob）彼此碰撞。為了避免在動力學模擬過程中產生奇怪的影響，必須設定 CoppeliaSim 兩個對像不會相互碰撞，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，將本地可響應蒙版設置為 00001111；對於 bubbleRob，將本地可響應掩碼設置為 11110000。如果再次運行模擬，會注意到兩個物體不再相互干擾。如圖：



[感測器、發動器、車輪、滑塊]

再次運行模擬，發現即使在發動器鎖定的情況下，BubbleRob 也會輕微移動，接著嘗試使用不同的物理引擎運行模擬：結果將有所不同，動態模擬的穩定性與所涉及的非靜態形狀的質量和慣性緊密相關。有關此效果的說明，務必

仔細閱讀本節和該節。現在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，對 3 個選定形狀的慣性進行相同的操作，然後再次運行模擬：穩定性得到了改善。在連桿動力學對話框中，將兩個發動器的**目標速度**都設置為 50。運行模擬：BubbleRob 現在向前移動並最終掉落在地板上，機著將兩個電機的**目標速度**都重置為零。

對象 bubbleRob 是所有物體的基礎，所有物體隨後將形成 BubbleRob 模型。我們將在稍後定義模型。同時要定義代表 BubbleRob 之物體的集合。為此，定義了一個集合物體，點擊[菜單欄->工具->集合]以打開集合對話框。或者可以通過點擊相應的工具欄按鈕來打開對話框：

在集合對話框中，點擊**添加新集合**，一個新的集合物體出現在下面的列表中，但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**。 現在的集合被定義為包含層次結構樹的所有物體（從 bubbleRob 物體開始）（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，點擊然後將其重命名為 bubbleRob_collection，接著關閉收集對話框。

在此階段，我們希望能夠跟踪 BubbleRob 與任何其他物體之間的最小距離。 為此使用[菜單欄->工具->計算模塊屬性]打開距離對話框，或者可以使用

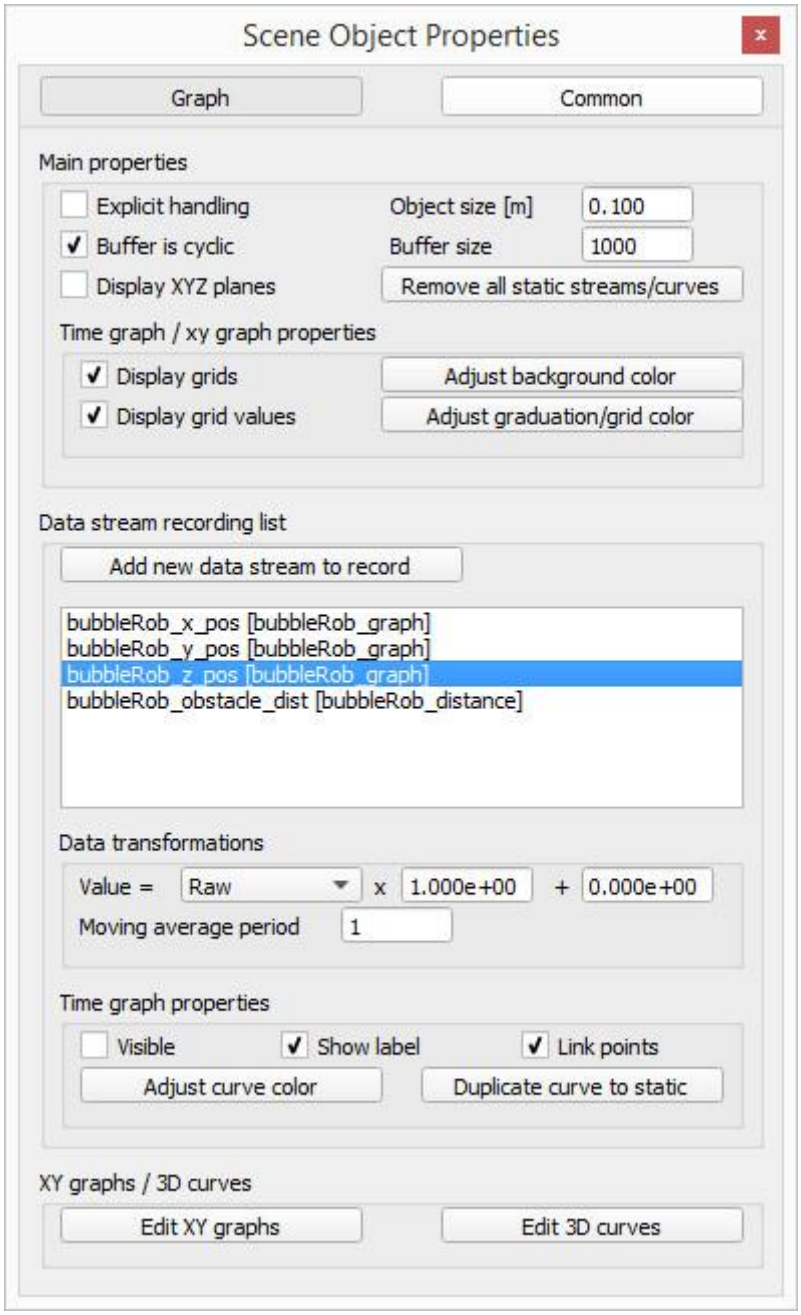
相應的工具欄按鈕打開計算模塊屬性對話框：

在距離對話框中，點擊**添加新的距離對象**並選擇一個距離對：[collection] 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（即該集合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。透過點擊其名稱將距離對象重命名為 bubbleRob_distance 並關閉距離對話框。接著運行模擬時，將不會看到任何區別，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。問題在於此階段的場景中沒有其他可測量的對象（定義地板的形狀默認情況下已禁用其可測量的屬性），在本教程的後續階段將為場景添加障礙。

接下來將向 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。點擊[菜單欄->添加->圖]，並將其重命名為 bubbleRob_graph，然後將圖形附加到 bubbleRob，並將圖形的絕對坐標設置為（0,0,0.005）。接著透過在場景層次結構中點擊其圖標來打開圖形屬性對話框，取消選中**顯示 XYZ 平面**，然後點擊**添加新數據以進行記錄**並**選擇對象：數據類型的絕對 x 位置**，接著選擇 **bubbleRob_graph** 作為要記錄的對象/項目，數據記錄列表中出現了一個項目，該項目是 bubbleRob_graph 的絕對 x 坐標的數據（即將記錄 bubbleRobGraph 的對象的絕對 x 位置）。 現在

我們還想記錄 y 和 z 位置：以與上述類似的方式添加這些數據。有 3 個數據，分別表示 BubbleRob 的 x、y 和 z 軌跡，接著將再添加一個數據，以便能夠跟踪機器人與其環境之間的最小距離：點擊**添加新數據以進行記錄**，然後選擇**距離：數據類型的選擇距離：段長**和 **Rob_distance** 作為要**記錄的對象/項目**。在**數據記錄列表**中，我們現在將 Data 重命名為 bubbleRob_x_pos，並將 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。

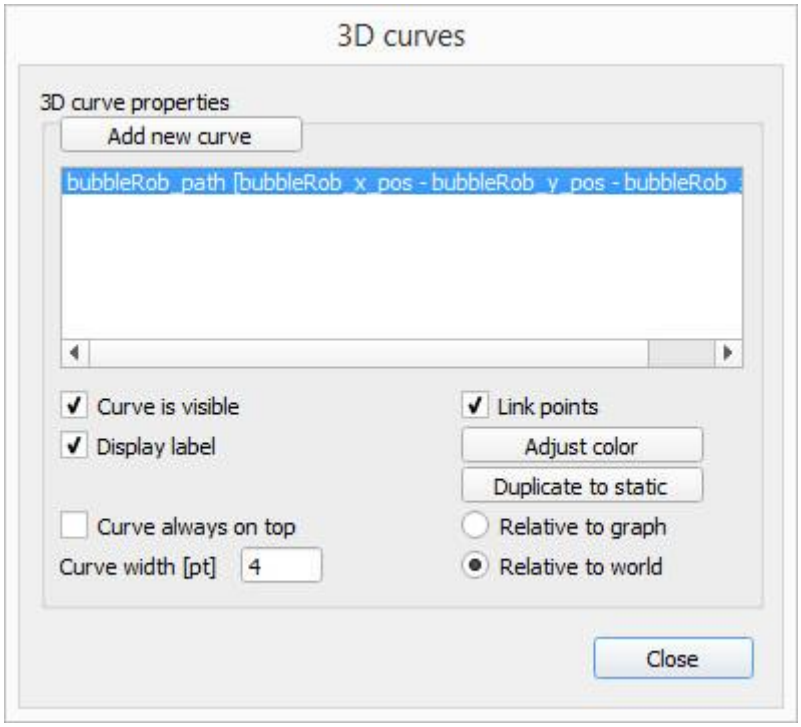
在**數據**記錄列表中和**時間圖屬性**部分中選擇 bubbleRob_x_pos，並取消選中**可見**。接著對 bubbleRob_y_pos 和 bubbleRob_z_pos 都執行相同的操作。這樣在時間圖中只能看到 bubbleRob_obstacle_dist 數據。如圖:



[圖形屬性]

接下來將建立一個顯示 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，為 X 項目選擇 bubbleRob_x_pos、為 Y 項目選擇 bubbleRob_y_pos、為 Z 項目選擇 bubbleRob_z_pos。接著將新添加的曲線從 Curve 重命名為

bubbleRob_path。 最後檢查**相對於 World** 項目並將**曲線寬度**設置為 4：



[3D 曲線屬性]

關閉與圖有關的所有對話框，然後將一個發動器目標速度設置為 50，運行模擬，然後將看到 BubbleRob 的軌跡顯示在場景中，接著停止模擬並將發動器目標速度重置為零。

添加具有以下尺寸的圓柱體：(0.1，0.1，0.2)。 我們希望此圓柱體是靜態的（即不受重力或碰撞的影響），但仍會對非靜態的可響應形狀施加一些碰撞響應。 為此，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**、**可測量**、**可渲染**、**可檢測**，在物體的公共屬性中執行此操作。 在選擇圓柱體的情況下，點擊物體平移工具欄按鈕：

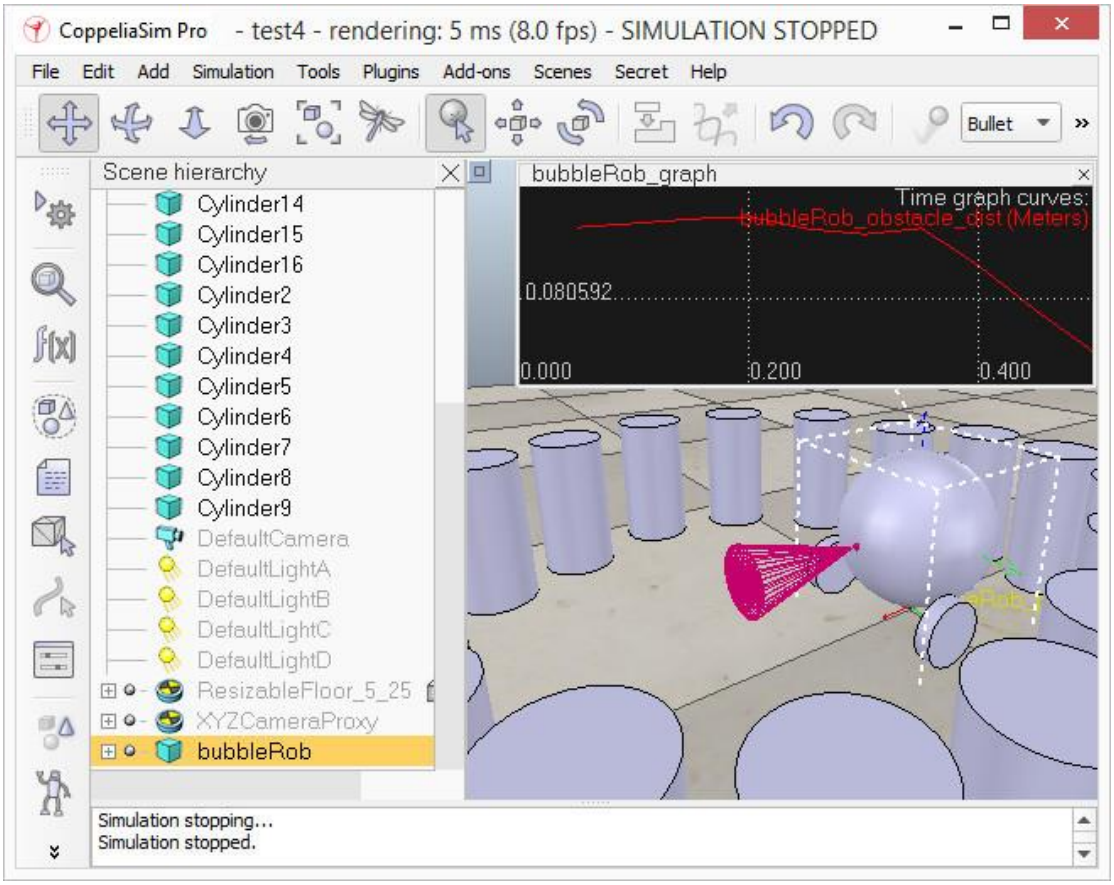
現在可以拖動場景中的任何點：圓柱體將跟隨運動，同時始終受約束以保持

相同的 Z 坐標。 接著複製並貼上圓柱數次，然後將圓柱移動到 BubbleRob 周圍的位置（從頂部查看場景時執行該操作最方便）。 在物體移動期間，按住 Shift 鍵可以執行較小的移動。 按住 ctrl 鍵可以在與常規方向正交的方向上移動。 完成後，再次選擇平移工具欄按鈕：

將左馬達的目標速度設置為 50 並運行模擬：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。接著停止模擬並將目標速度重置為零。

接著需要完成 BubbleRob 作為模型定義，選擇模型基礎（即物體 bubbleRob），然後選擇**物體是模型基礎**，接著選擇**物體/模型可以轉移或接受對象共同屬性中的 DNA**：有一個點畫的邊界框包含模型層次結構中的所有物體。選擇兩個連桿，即感測器和圖形，然後啟用項目**不顯示為內部模型選擇**，接著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連桿和感測器。仍在同一對話框中，禁用攝像機可見性層 2，並為兩個桿測和力感測器啟用攝像機可見性層 10：這有效地隱藏了兩個連桿和力感測器，因為默認情況下禁用第 9-16 層，然而可以隨時修改整個場景的可見性層。接著要完成模型定義，選擇視覺感測器、兩個輪子、滑塊和圖形，然後啟用**選擇模型基礎選項**：如果嘗試在場景中選擇模型中的物體，則整個模型將被選擇，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 的同時點擊選擇物體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物體。最後將模型樹折疊到場景層次中。如圖：



[BubbleRob 模型定義]

接下來將在與 BubbleRob 感測器相同的位置和方向上添加視覺感測器，再次打開模型層次結構，然後點擊[菜單欄->添加->視覺感測器->透視類型]，然後將視覺感測器連接到感測器，並將視覺感測器的本地位置和方向設置為（0,0,0），接著還需確保視覺感測器不可見，不是模型邊界框的一部分，並且如果點擊該模型，則會選擇該模型。為了自定義視覺感測器，需打開其屬性對話框。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，右鍵點擊[彈出菜單->視圖->將視圖與選定的視覺感測器關聯]（需確保在該過程中選擇了視覺感測器）。

通過點擊[菜單欄->添加->關聯的子程式->非線程]，將非線程子程式附加到視覺感測器，接著點擊場景層次結構中視覺感測器旁邊出現的小圖標：這將打開剛剛添加的子程式。將以下程式碼複製並貼程式編輯器中，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
```

為了能夠看到視覺感測器的圖像，開始模擬，然後再次停止。

場景所需的最後一件事是一個小的子程式，它將控制 BubbleRob 的動作，選擇 bubbleRob 並點擊[菜單欄->添加->關聯的子程式->非線程]。 點擊場景層次結構中 bubbleRob 名稱旁邊顯示的程式圖標，然後將以下程式碼複製並貼到程式編輯器中，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_leftMot
or") -- Handle of the left motor

rightMotor=sim.getObjectHandle("bubbleRob_rightM
otor") -- Handle of the right motor

noseSensor=sim.getObjectHandle("bubbleRob_sensin
gNose") -- 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.getObjectName(bubbleRobBase)..'
speed" closeable="false" resizeable="false"
activate="false">'..[[
        <hslider minimum="0" maximum="100"
onchange="speedChange_callback" id="1"/>
        <label text="" style="* {margin-left:
300px;}"/>
        </ui>
        ]]
    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
```

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，然後將其複制/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

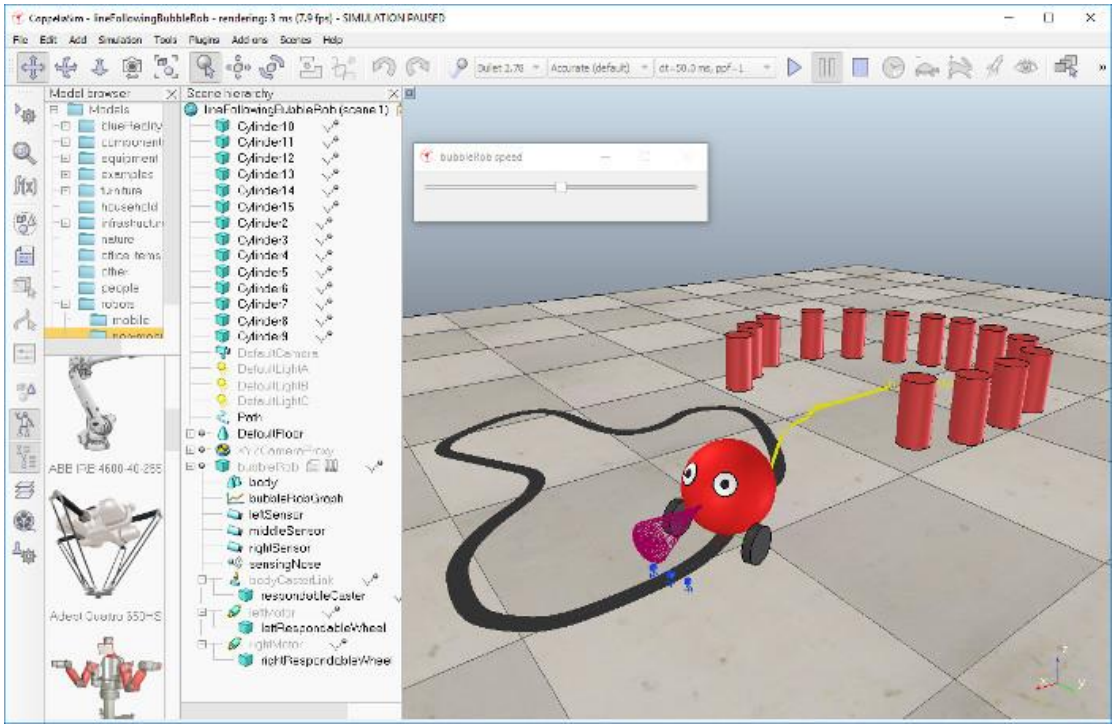
同，最小距離計算功能可能會嚴重降低模擬速度。可以通過選擇/取消選擇**啟用所有距離計算**來在距離對話框中打開和關閉該功能。

使用程式控制機器人或模型只是一種方法。 CoppeliaSim 提供了許多不同的方法（也可以結合使用），請參閱外部控制器教程。

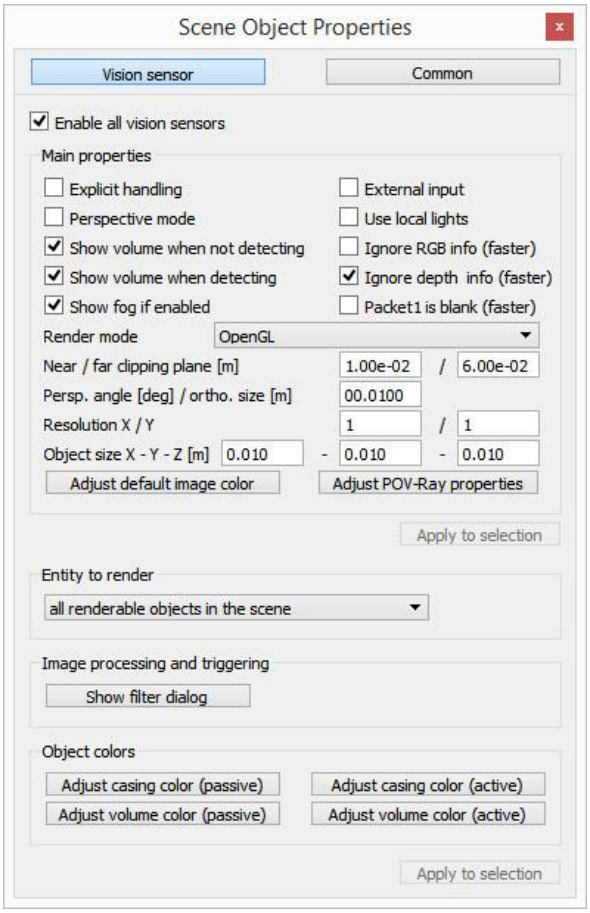
BubbleRob 跟隨路徑

在本教程中，旨在擴展 BubbleRob 的功能，以使 BubbleRob 遵循地面上的規則。確保您已經閱讀並理解了第一個 BubbleRob 教程，本教程由 Eric Rohmer 提供。

在 CoppeliaSim 的安裝文件夾中的 tutorials / BubbleRob 中加載第一個 BubbleRob 教程的場景。與本教程相關的場景文件位於 tutorials / LineFollowingBubbleRob 中。 下圖說明了將設計的模擬場景：



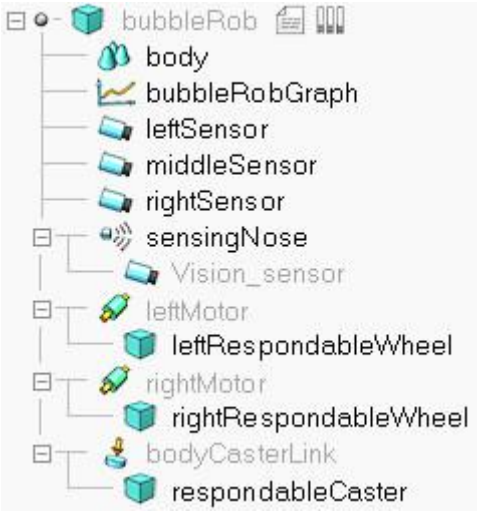
首先創建 3 個視覺感測器中的第一個，並將其附加到 bubbleRob 物體選擇[菜單欄->添加->視覺感測器->正交類型]。 通過點擊場景層次中新創建的視覺感測器圖標來編輯其屬性，並更改參數以反映以下對話框：



視覺感測器必須面向地面，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]。

有幾種可能性可以讀取視覺感測器，由於視覺感測器只有一個像素，並且操作簡單，因此只需查詢視覺感測器讀取的圖像的平均強度值即可。對於更複雜的情況，可以設置視覺回調函數。接著複製並貼上視覺感測器兩次，並將其名稱調整為 leftSensor、middleSensor、rightSensor。將 bubbleRob 設置為其父級（即將其附加到 bubbleRob 物體）。 現在感測器在場景層次中應如下

所示：



需正確放置傳感，因此使用位置對話框，選擇**位置**選項並設置以下絕對坐標：

- 左感測器：[0.2; 0.042; 0.018]
- 中間感測器：[0.2; 0; 0.018]
- 右感測器：[0.2; -0.042; 0.018]

接著修改環境，移除 BubbleRob 前面的幾個圓柱體。接下來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，然後點擊[菜單欄->添加->路徑->圓圈類型]。使用鼠標啟用物體移動，可以通過兩種方式調整路徑的形狀：

- 選擇路徑（並且只有路徑）後，按住 Ctrl 並點擊其控制點之一，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，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（隨時可以在以後的階段對其進行修改），選擇它

並在路徑屬性中取消選擇**顯示點的方向、顯示路徑線**和**顯示路徑上的當前位置**，然後點擊顯示路徑整形對話框，這將打開路徑整形對話框，接著點擊啟用路徑整形，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，接著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，而且還會影響視覺感測器所看到的。 為了避免與 z 值有關的問題，只需將路徑對象的位置向上移動 0.5mm 即可。

最後一步是調整 BubbleRob 的控制器，使其也將遵循黑色路徑，打開附加到 bubbleRob 的子程式，並將其替換為以下程式碼：

```
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)
    leftMotor=sim.getObjectHandle("leftMotor")
    rightMotor=sim.getObjectHandle("rightMotor")
    noseSensor=sim.getObjectHandle("sensingNose")
    minMaxSpeed={50*math.pi/180,300*math.pi/180}
    backUntilTime=-1 -- Tells whether bubbleRob
is in forward or backward mode
    floorSensorHandles={-1,-1,-1}
```

```
floorSensorHandles[1]=sim.getObjectHandle("leftS
ensor")

floorSensorHandles[2]=sim.getObjectHandle("middl
eSensor")

floorSensorHandles[3]=sim.getObjectHandle("right
Sensor")
    -- Create the custom UI:
        xml = '<ui
title="'..sim.getObjectName(bubbleRobBase)..'
speed" closeable="false" resizeable="false"
activate="false">'..[[
        <hslider minimum="0" maximum="100"
onchange="speedChange_callback" id="1"/>
        <label text="" style="* {margin-left:
300px;}"/>
        </ui>
        ]]
    ui=simUI.reate(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)
    if (result>0) then
backUntilTime=sim.getSimulationTime()+4 end

    -- read the line detection sensors:
    sensorReading={false,false,false}
    for i=1,3,1 do

result,data=sim.readVisionSensor(floorSensorHand
les[i])
        if (result>=0) then
```

```
          sensorReading[i]=(data[11]<0.3) --
data[11] is the average of intensity of the
image
       end
       print(sensorReading[i])
    end

    -- compute left and right velocities to
follow the detected line:
    rightV=speed
    leftV=speed
    if sensorReading[1] then
       leftV=0.03*speed
    end
    if sensorReading[3] then
       rightV=0.03*speed
    end
    if sensorReading[1] and sensorReading[3] then
       backUntilTime=sim.getSimulationTime()+2
    end

    if (backUntilTime<sim.getSimulationTime())
then
       -- When in forward mode, we simply move
forward at the desired speed

sim.setJointTargetVelocity(leftMotor,leftV)

sim.setJointTargetVelocity(rightMotor,rightV)
    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
```

可以通過視覺感測器輕鬆調試生產線：選擇一個，然後在場景視圖中選擇[右鍵->添加->浮動視圖]，然後在新添加的浮動視圖中選擇[右鍵->視圖- ->將視圖與選定的視覺感測器關聯]。

最後刪除在第一個 BubbleRob 教程中添加的輔助項：刪除圖像處理視覺感測器、其關聯的浮動視圖，該浮動視圖表示障礙物的清除。 透過距離對話框也刪除距離計算對象。

外部控制器教程

在 CoppeliaSim 中，有幾種方法可以控制機器人或模擬機器人：

- 最方便的方法是編寫一個子程式來處理給定機器人或模型的行為，這是最方便的方法，因為子程式直接附加到場景對象，將與相關的場景物體一起複製，不需要使用外部工具進行任何編譯，且可以在線程或非線程模式下運行，也可以鬥過自定義 Lua 函數或 Lua 擴展庫進行擴展。使用子程式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，沒有連接延遲，並且子程式是應用程序主線程的一部分（固有的同步操作）。但是編寫程式有幾個缺點：無法選擇編程語言，不能擁有最快的程式碼，並且除了 Lua 擴展庫之外，無法直接訪問外部函數庫。

- 可以控制機器人或模擬的另一種方法是編寫插件。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。插件通常與子程式結合使用（例如，插件註冊自定義的 Lua 函數，當從子程式中調用時，該 Lua 函數將回調特定的插件函數）。使用插件的主要優勢在於與本節中提到的後 3 種方法（即使用常規 API）一樣，沒有連接延遲，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：編程更加複雜，並且也需要使用外部編譯。另請參閱插件教程。
- 控制機器人或模擬的第三種方法是編寫依賴於遠程 API 的外部客戶端應用程序。如果需要從外部應用程序，機器人或另一台計算機運行控制程式碼，這是一種非常便捷的方法，也可以使用與運行真實機器人完全相同的程式碼來控制或模擬模型（例如虛擬機器人）。遠程 API 有兩個版本：基於 B0 的遠程 API 和舊版遠程 API。
- 控制機器人或模擬機器人的第五種方法是通過 ROS 節點。 ROS 與遠程 API 相似，是使多個分佈式進程相互連接的便捷方法。儘管遠程 API 非常輕巧且快速，但僅允許與 CoppeliaSim 連接。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。有關詳細信息，請參閱 ROS 接口。
- 控制機器人或模擬機器人的第六種方法是通過 BlueZero（BØ）節點。與 ROS 類似，BlueZero 是使多個分佈式進程相互連接的一種便捷方法，並

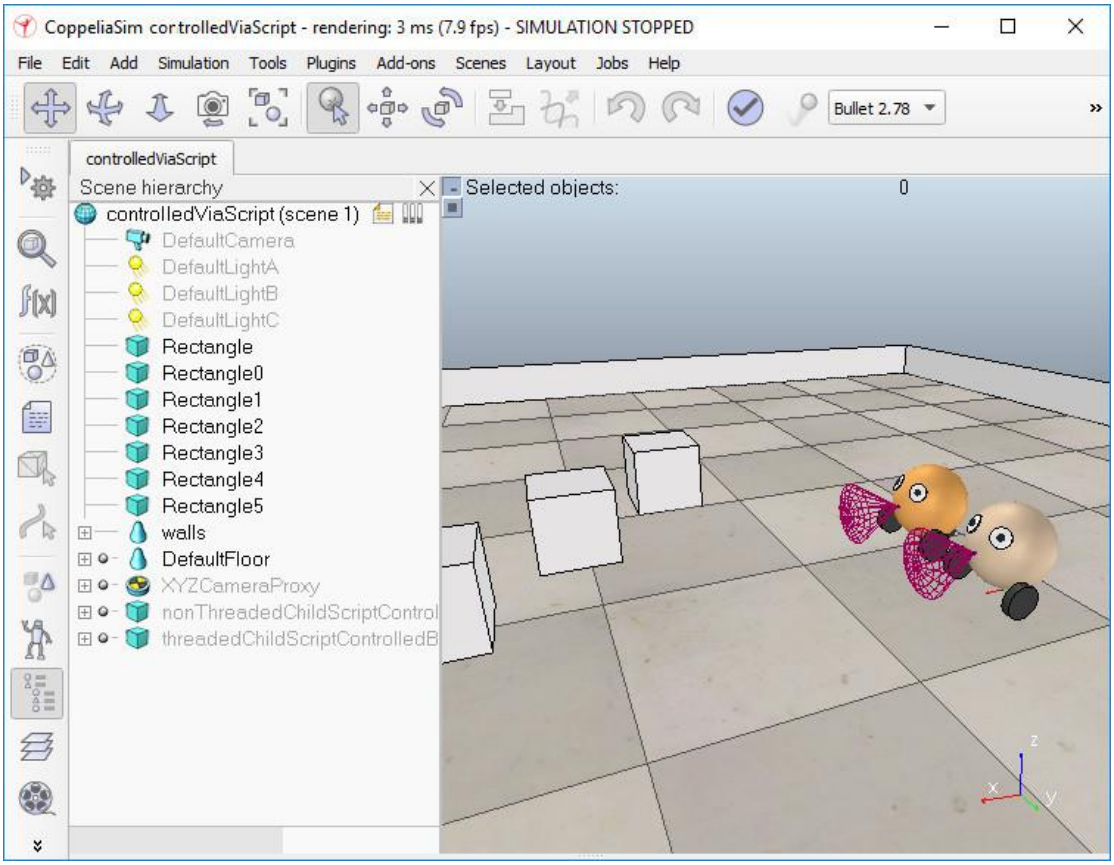
且是一種輕量級的跨平台解決方案。有關詳細信息，請參考 BlueZero 界面。

- 控制機器人或模擬機器人的第七種方法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 程式碼進行連接。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，控制程式碼也可以在機器人或其他計算機上運行。但是與使用遠程 API 的方法相比，這種控制或模擬模型的方法更加乏味。

有 8 個與本教程相關的場景文件：

- scenes / controlTypeExamples /受控 ViaScript：一個機器人是通過非線程子程式控制的，另一個是通過線程子程式控制的。
- scenes / controlTypeExamples /受控 ViaPlugin：機器人是通過插件控制的。
- scenes / controlTypeExamples / controlViaB0RemoteApi：通過基於 B0 的遠程 API 來控制機器人。
- scenes / controlTypeExamples /受控 ViaLegacyRemoteApi：通過舊版遠程 API 控制機器人。
- scenes / controlTypeExamples / controlViaB0：通過 BlueZero 界面控制機器人。

- scenes / controlTypeExamples /受控 ViaRos：通過 ROS 接口控制機器人。
- scenes / controlTypeExamples / controlViaRos2：通過 ROS2 接口控制機器人。
- scenes / controlTypeExamples /受控 ViaTcp：通過 LuaSocket 和 TCP 控制機器人。



在所有 8 種情況下，都使用子程式，主要是為了與外界建立鏈接（例如，啟動正確的客戶端應用程序，並將正確的對象句柄傳遞給它）。有兩種其他方法可以控制機器人、模擬或模擬器本身：使用自定義程式或附加組件。但是不建議將它們用於控制，而應在不運行模擬時將其用於處理功能。

例如，連接到場景控制的 ViaB0RemoteApi.ttt 中的機器人的子程式具有以下主要任務：

- 使用某些對象句柄作為參數啟動控制器應用程序（bubbleRobClient_b0RemoteApi）。基於對象 B0 的遠程 API 的服務器功能由對象 b0RemoteApiServer 提供。

作為另一個示例，連接到場景控制的 ViaRos.ttt 中的機器人的子程式具有以下主要任務：

- 檢查是否已加載 CoppeliaSim 的 ROS 接口
- 使用某些主題名稱或對象句柄作為參數啟動控制器應用程序（rosBubbleRob）

然而，作為另一個示例，連接到場景控制的 ViaTcp.ttt 中的機器人的子程式具有以下主要任務：

- 搜索空閒的套接字連接端口。
- 使用所選的連接端口作為參數啟動控制器應用程序（bubbleRobServer）。
- 本地連接到控制器應用程序。
- 在每次模擬過程中，將感測器值發送到控制器，並從控制器讀取所需的發動值。

- 在每次模擬過程中，將所需的發動值應用於機器人的連桿。

運行模擬，然後複製並貼上機器人：將看到重複的機器人將直接運行，因為附加的子程式負責啟動各自外部應用程序的新實例，或調用適當的插件函數。