Лабораторная работа №2

Основы программирования роботов в CoppeliaSim

Платформа CoppeliaSim поддерживает работу с разными языками программирования, в том числе и с наиболее распространенными, такими как C++ и Python, однако встроенным языком CoppeliaSim является Lua, и начать написание скриптов на этом языке можно сразу после запуска программы.

Отличительной особенностью Lua является простота: во многом синтаксис схож с популярным языком C, что значительно упрощает работу для тех, кто уже знаком с языком. Скрипты в CoppeliaSim открывают большие возможности для реализации управления как отдельными объектами сцены, так и платформой.

Платформа CoppeliaSim имеет ряд особенностей, которые предоставляют разработчику широкие возможности для создания симуляций. Как основной компонент CoppeliaSim можно выделить технологию встроенных скриптов, которые выполняют функции контроллеров в симуляциях. При этом, наличие возможности привязки отдельных скриптов к компонентам робота позволяет реализовать четкую иерархию, обеспечивая портативность и масштабируемость.

Любая симуляция в CoppeliaSim по умолчанию имеет основной скрипт, который крайне не рекомендуется менять, т.к. данный скрипт решает общие задачи обеспечения корректности данных при выполнении симуляции. Например, он вызывает разные подсистемы для моделирования кинематики и динамики механических элементов системы. Из основного скрипта также выполняется вызов дочерних скриптов каскадным способом, но эту функцию можно отключить.

Дочерние скрипты, в отличие от основного, прикрепляются к отдельным компонентам модели. Они являются неотъемлемой частью сценария симуляции и выполняются при каждой итерации моделирования, как и основной скрипт. Следует также помнить, что скрипты являются исполняемыми файлами и не требуют предварительной компиляции. Кроме того, дочерние скрипты могут быть потоковыми или непотоковыми. Разработчики CoppeliaSim рекомендуют по возможности использовать непотоковые скрипты, однако в некоторых задачах потоковые скрипты лучше решают поставленные задачи. Следует также заострить внимание на том, что симуляция – это итеративный процесс, т.е. перерасчет параметров моделируемой системы осуществляется через постоянный промежуток времени (шаг моделирования) итеративно. В CoppeliaSim по умолчанию используется 50 миллисекунд.

Потоковые скрипты – скрипты, выполнение которых будет продолжаться при следующей итерации. Такие скрипты выполняются один раз от начала до конца, но также можно отключить эту функцию, чтобы не было прерывания после первой итерации.

Непотоковые скрипты – скрипты, выполнение которых начинается с начала при каждой последующей итерации, при этом осуществляется полная

передача управления симуляцией в эти скрипты. Если по каким-то причинам скрипт не завершился, и управление не передалось обратно на основной скрипт, то симуляция прерывается. Такие скрипты вызываются основным скриптом 2 раза за каждый шаг симуляции: при активации и получении сенсорной информации.

Таким образом, для лучшего понимания процесса исполнения скрипта следует рассмотреть более подробно структуру не потокового скрипта. Такой скрипт всегда состоит из нескольких блоков (см. рис. 1).

```
1  -- DO NOT WRITE CODE OUTSIDE OF THE if-then-end SECTIONS BELOW!! (unle
2
3  if (sim_call_type==sim_childscriptcall_initialization) then
4
5      -- Put some initialization code here
6
7
8  end                       1. Блок инициализации
9
10
11 if (sim_call_type==sim_childscriptcall_actuation) then
12
13     -- Put your main ACTUATION code here
14
15     -- For example:
16     --
17     -- local position=simGetObjectPosition(handle,-1)
18     -- position[1]=position[1]+0.001
19     -- simSetObjectPosition(handle,-1,position)
20
21 end   2. Блок управляющего кода объектов
22
23
24 if (sim_call_type==sim_childscriptcall_sensing) then
25
26     -- Put your main SENSING code here
27
28 end
29     3. Блок управляющего кода сенсеров
30
31 if (sim_call_type==sim_childscriptcall_cleanup) then
32
33     -- Put some restoration code here
34     4. Блок завершения симуляции
35 end
```

Рисунок 1 – Окно редактирования непотокового скрипта в программе CoppeliaSim.

Блок инициализации. Содержимое данного блока выполняется только один раз при запуске симуляции. В данном блоке производятся такие операции, как объявление необходимых переменных и присвоение им исходных значений. Также в этой части скрипта задаются обработчики для управления объектами сцены.

Блок активации. Содержимое данного блока выполняется итеративно через равные промежутки времени (шаг моделирования). Часть скрипта, написанная в этом блоке, будет повторяться вплоть до остановки симуляции

или возникновения критической ошибки (при которой тоже симуляция будет остановлена). Здесь описывается основной алгоритм управления.

Блок управляющего кода сенсоров. Содержимое данного блока выполняется столько же раз, сколько и блок активации. Однако основной скрипт CoppeliaSim обращается к этому блоку только после завершения выполнения скрипта из блока активации. Данный блок разработан для получения данных из сенсоров.

Блок завершения симуляции. Данный блок позволяет выборочно стереть данные, полученные в ходе симуляции. Скрипт из этого блока начинает выполняться один раз перед завершением симуляции или удалением скрипта. Этот блок в большинстве случаев остается пустым.

Скрипты не потокового типа имеют более простую структуру. Весь скрипт может указываться в файле без разделения на блоки, однако ввиду итеративности моделируемых процессов почти всегда необходимо наличие основного цикла. На рис. 2 представлена классическая структура не потокового скрипта в программе CoppeliaSim, однако основной цикл там может и отсутствовать при решении задач определенного класса.

```
1  -- Put some initialization code here
2      1.Блок инициализации
3
4  -- Put your main loop here, e.g.:
5  while simGetSimulationState()~=sim_simulation_advancing_abouttostop do
6
7      2. Основной цикл управляющего кода
8
9  end
10
11
12 -- Put some clean-up code here
13     3. Блок завершения скрипта (очистка)
14
```

Рисунок 2 – Окно редактирования не потокового скрипта в программе CoppeliaSim.

Основные конструкции при написании скриптов

Комментарии на языке Lua должны начинаться с двойного дефиса (рис. 3, строка 4). Рекомендуется не удалять фрагменты скрипта во время разработки алгоритмов управления, а обозначать их как комментарий, чтобы потом не пришлось заново писать, если они будут нужны.

Как правило, ни один скрипт не обходится без использования переменных. В Lua допускается объявление новых переменных в любой момент времени, также при объявлении можно задать их начальное значение (например, рис. 3, строка 9). Также нет необходимости указывать тип переменной. Lua определит его в зависимости от значения, которое присваивается переменной в первый раз. Переменные с числовым значением воспринимаются как число с плавающей точкой. Также допускается использование переменных с символьными значениями и переменных логического типа (возможные значения: «истина» или «ложь»). Имеется и

функция уничтожения (освобождения) переменных. Все переменные в Lua по умолчанию являются глобальными, то есть доступны из любой области.

```
3  if (sim_call_type==sim_childscriptcall_initialization) then
4      -- Put some initialization code here
5      base=simGetObjectHandle('verh_dyn')
6      jointhandle=simGetObjectHandle('joint_1')
7      jointcurrentpos=simGetJointPosition(jointhandle)
8      simAddStatusbarMessage('position='..jointcurrentpos)
9      nextposition=1
```

Рисунок 3 – Пример скрипта инициализации.

Ветвление и циклы реализованы в Lua также, как и в большинстве C-подобных языков программирования с некоторыми оговорками.

Условный оператор «If» (Если) требует обязательного использования конструкции «then/end». При этом после ключевого слова «If» необходимо указать логическое выражение или переменную. После логического выражения следует ключевое слово «then», за которым начинается «тело» условия – часть скрипта, которая будет выполняться при истинности указанного в скрипте условия. В условном операторе имеется необязательная составляющая - дополнительное условие «иначе» (elseif), «тело» этого условия – часть скрипта, которая выполняется только если указанное в скрипте условие окажется ложным. На рис. 4 приведен пример использования условного оператора.

```
11 k=10
12 if k ~= 10 and k<0 then
13     print('negative')
14     simAddStatusbarMessage('negative')
15 else
16     print('positive')
17     simAddStatusbarMessage('positive and k='..k)
18 end
```

Рисунок 4 – Пример использования условного оператора и функций вывода информации в консольное окно и окно состояния.

Условный оператор также допускает комбинирование условий с помощью операторов «and» (логическое «и») и «or»(логическое «или»). Строки 13 и 16 на рис. 6 выводят ключевые слова базовыми функциями Lua в консольное окно, которое по умолчанию скрыто. Логичнее и удобнее выводить данные не в консольное окно, а в строку состояния CoppeliaSim, для этого необходимо воспользоваться регулярной функцией CoppeliaSim (пример приведен на рис. 4, строки 14 и 17). Также на рис. 4 в строке 17 выводится не только ключевое слово, но и значение переменной.

Как и в случае с условными операторами, система управления редко обходится без использования хотя бы одного цикла. Циклы в Lua задаются с помощью ключевых слов, обозначающих тип цикла совместно с ключевыми

словами «do» и «end». При этом порядок такой: сначала указывается тип, затем условие выполнения цикла, далее слово «do», далее следует «тело цикла» - фрагмент скрипта, который будет выполняться циклично, и заканчивается цикл словом «end». Наиболее распространенными являются циклы типа «while» и«for». На рис. 5 приведен пример самого простого цикла while, который увеличивает на единицу значение переменной «k», пока условие «k<50» не станет ложным.

```
10  k=1
11  while k < 50 do
12  k = k + 1
13  end
```

Рисунок 5 – Пример использования цикла «while»

Циклы в Lua, как и условное ветвление, должны завершаться ключевым словом «end». Особое внимание следует уделить условию, которое используется в цикле. Особенно это важно, когда используется значение переменной вместо логического условия. У неопределенных переменных значение по умолчанию равно «nil». При этом в условии цикла только переменные со значением «nil» и «false» (логический тип переменной «ложь») возвращают false, в то время как значение переменной «0» и ’ ’ возвращают true.

Второй тип циклов, который наиболее часто применяется, – «for». Данный тип хорошо подходит для задач, когда необходимо выполнить цикл со счетчиком, но следует упомянуть, что в большинстве случаев «for» можно заменить на цикл «while», задействовав несколько дополнительных переменных. Пример использования цикла «for» приведен на рис. 6, данный фрагмент скрипта выполняет 100 итераций начиная от 1 (включая 1 и 100). При этом можно поменять условия местами, вместо 1 поставить 100 и 100 вместо 1, тогда цикл будет выполняться с изменением значения переменной «i» от 100 до 1. Также в цикле «for» имеется необязательный параметр шага, по умолчанию шаг равен единице, и нет необходимости его указывать. Однако, для случаев, когда требуется использование шага со значением отличным от 1, то нужно указать его через запятую после конечного значения (если необходим шаг 2, то для примера на рис. 8 это будет условие «i = 1, 100, 2»).

```
10      Sum = 0
11      for i = 1, 100 do -- 100 iterations from 1.
12       Sum = Sum + i
13      end
```

Рисунок 6 – Пример использования цикла «for».

Таблицы в Lua являются единственным структурным элементом, они сочетают в себе свойства массива, хэш-таблицы («ключ» - «значение»),

структуры и объекта. Чаще всего таблицы используются в качестве словарей, а ключ при этом по умолчанию имеет строковый (символьный) тип. Пример приведен на рис. 7. Строка 11 объявляет переменную типа «таблица» и задает 2 ключа и соответствующие им значения. Доступ к значениям можно получить указанием названия таблицы и ключа через точку (пример на рис. 7).

```
11    t = {key1 = 'value1', key2 = false}
12    print(t.key1) -- 'value1'
13    t.newKey = {} --add new key
14    t.key2 = nil -- delete key2
```

Рисунок 7 – Пример использования таблиц на языке Lua

Как видно из примера, допускается использование в качестве ключа не только строк, но также и всех остальных типов переменных, которые доступны в Lua.

Другой и крайне важной составляющей языка программирования являются функции. Функции можно создавать свои собственные, также можно использовать уже готовые, которые можно найти в справочнике по Lua. Пример заданной разработчиком функции приведен на рис. 8. Как видно из примера, функция может принимать несколько значений. Также функции могут возвращать несколько значений.

```
6  function bar(a, b, c)
7      sum=a+b+c
8      r=a-b-c
9    return sum,r
10 end
11 k,p=bar(2,2,3)
12 simAddStatusbarMessage('k='..k)
13 simAddStatusbarMessage('p='..p)
```

Рисунок 8 – Пример использования пользовательской функции на Lua.

Особое внимание стоит уделить регулярным функциям CoppeliaSim. Эти функции уже интегрированы в Lua при написании скриптов CoppeliaSim и начинаются на «sim». Именно эти функции и используются для взаимодействия с симуляцией: управления, считывания данных, отладки и решения многих других задач. Полный список функций с подробным описанием принимаемых ими переменных и возвращаемых данных доступен в официальной документации CoppeliaSim (https://www.coppeliarobotics.com/helpFiles/en/apiFunctions.htm).

Рассмотрим простейший пример программирования движения мобильного робота на языке Lua. Установим на сцену робот Pioneer 3dx, как это было описано в лабораторной работе №1. Для того, чтобы посмотреть все скрипты модели, нажмем кнопку Scripts боковой панели. В открывшемся

окне менеджера скриптов (рис. 9) мы увидим, что у модели имеется основной скрипт и дочерний непотоковый скрипт (он был приведен в лабораторной работе №1).

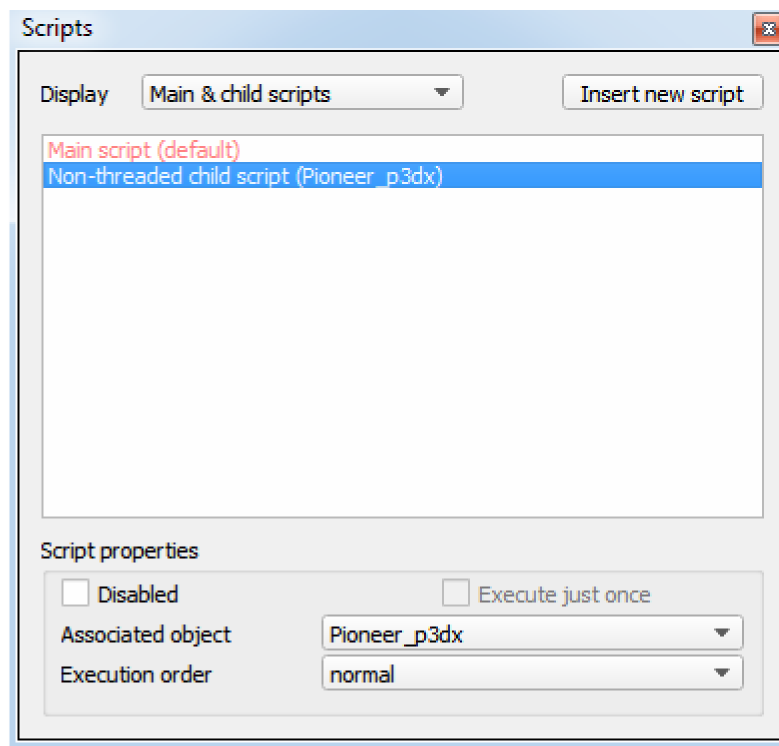

Рисунок 9 – Окно менеджера скриптов.

Отключим дочерний скрипт, поставив ему галочку в окне Disabled и указав на отсутствие ассоциируемых с ним объектов (рис. 10). Запустив симуляцию мы увидим, что робот остается неподвижным.

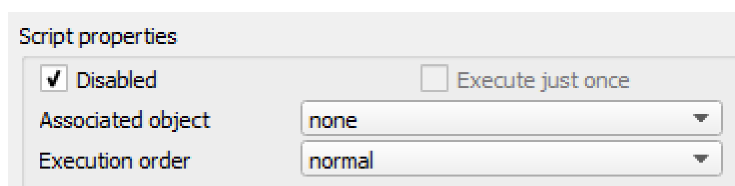

Рисунок 10 – Отключение скрипта.

Добавим новый дочерний непотоковый скрипт, который запрограммирует перемещение робота вперед по прямой на 1 м (рис. 11). В качестве ассоциируемого с ним объекта укажем Pioneer_p3dx.

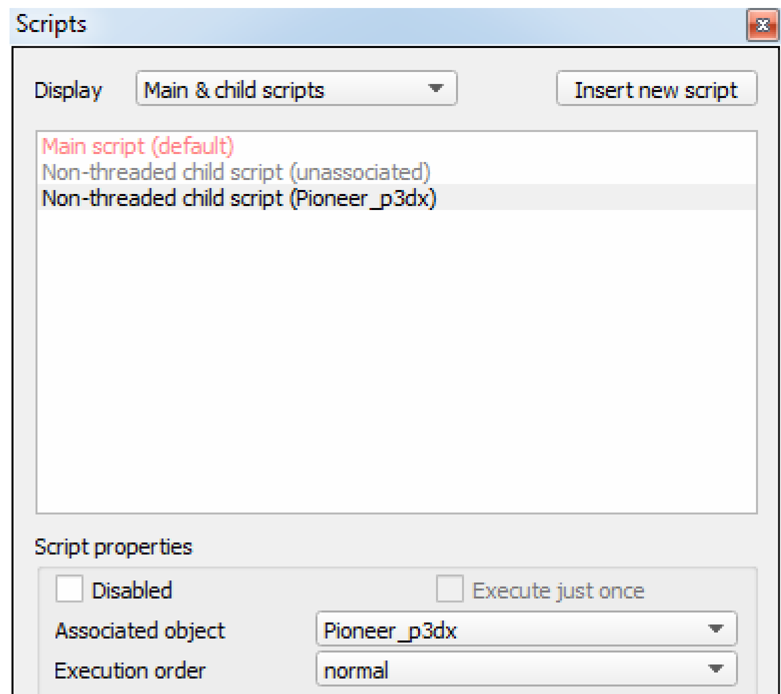

Рисунок 11 – Добавление нового скрипта.

Необходимое перемещение робота зададим через скорость вращения колес v и время t (размерность физических величин в CoppeliaSim метр, килограмм, секунда, радиан). В результате скрипт будет иметь вид:

```
1  motorLeft=simGetObjectHandle("Pioneer_p3dx_leftMotor")
2  motorRight=simGetObjectHandle("Pioneer_p3dx_rightMotor")
3  t=simGetSimulationTime("Pioneer_p3dx")
4  if t<100 then
5  vLeft=0.1
6  vRight=0.1
7  else
8  vLeft=0
9  vRight=0
10 end
11 simSetJointTargetVelocity(motorLeft,vLeft)
12 simSetJointTargetVelocity(motorRight,vRight)
```

Запустив симуляцию мы увидим, что робот проедет по прямой 2 клетки сцены (1 м) и остановится.

Поворот двухколесного робота обеспечивается за счет разности скоростей вращения левого и правого колеса.

Задание на лабораторную работу:

Написать скрипты для движения робота Pioneer 3dx по траекториям, показанным на рисунке 12.

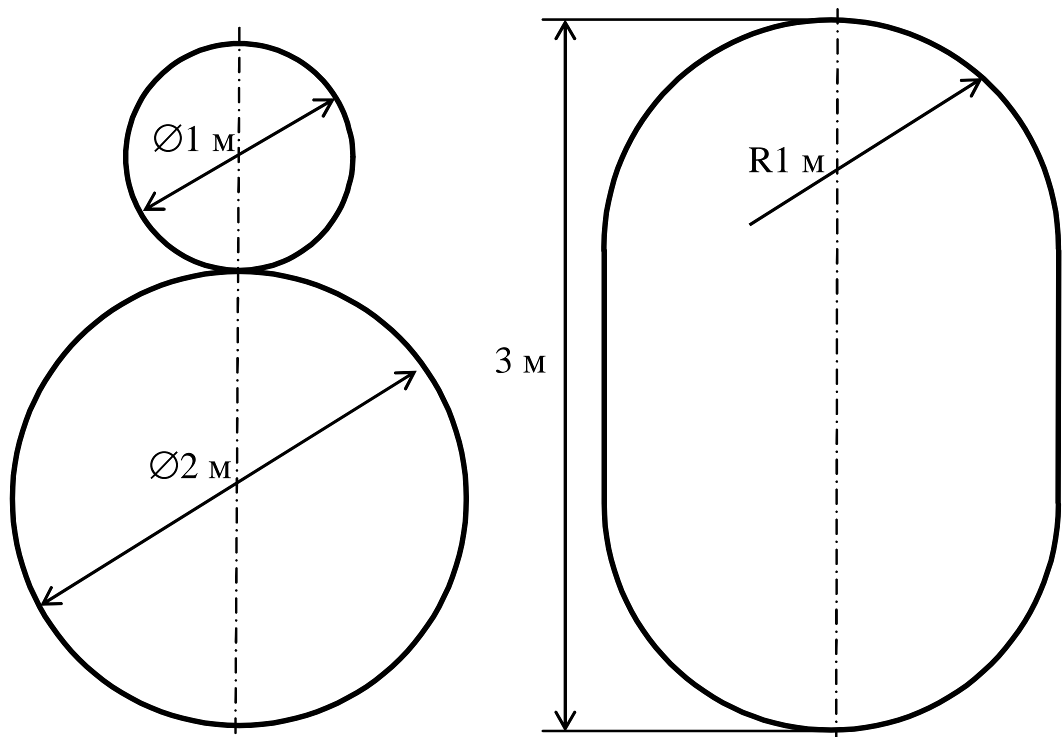

Рисунок 12 – Требуемые траектории движения робота.