Microsoft Robotics Studio Walk-through

Installation requisites

This guide has been developed with ***Microsoft Robotics Studio (MSRS) 1.5 Refresh***. Everything is license-free and can be downloaded from the Microsoft website.

Microsoft Robotics Studio 1.5 can be downloaded from here:

- http://www.microsoft.com/downloads/details.aspx?FamilyId=73092FF6-E37B-45C6-8E5E-C23D5D632B1E&displaylang=en

The application is not 100% compatible with other versions, so it's recommended to use the same version. As well, you have to install this **update packages**:

- *Runtime and Tools Update for Microsoft Robotics Studio (1.5):* http://www.microsoft.com/downloads/details.aspx?FamilyId=2F747403-7D94-43F0-836B-84247FEE66C6&displaylang=en
- *Samples Update for Microsoft Robotics Studio (1.5):* http://www.microsoft.com/downloads/details.aspx?FamilyId=7EEB9B70-0E86-4E3E-92AF-6148BDA34B7C&displaylang=en

MSRS requires *Ageia Physics Engine* and *XNA* and *NET 3.0* runtimes, which are installed automatically. The *XNA* version including the package is 2.0, but the simulation environment could need the 1.0 version, so it's recommended to install it after the whole application. You can download it here:

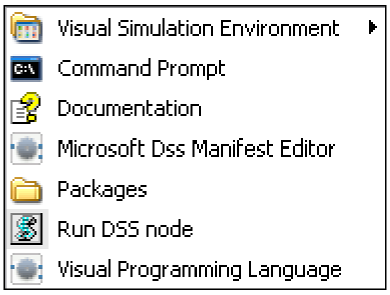
- http://www.microsoft.com/downloads/details.aspx?FamilyId=A7DA4763-6807-4BD5-8D18-18C60C437F93&displaylang=en

If you have any other problem please search the MSDN forums for help:

- http://forums.microsoft.com/qawizard/ask.aspx?siteid=1

System description

Microsoft Robotics Studio (MSRS) consists of a set of applications. When you installed MSRS, it should have created a folder in your Start Menu containing a shortcut for "Robotics Studio Command Prompt". Open one of these windows now.



Visual Simulation Environment
Command Prompt
Documentation
Microsoft Dss Manifest Editor
Packages
Run DSS node
Visual Programming Language

Figure 1. MSRS Program Contents

- **Visual Simulation Environmen**t: the simulation environment, it can be executed for a range of robots.
- **Command Prompt**: it opens a DOS Command Prompt. Many applications can be executed from this console, like starting a service.
- **Documentation:** the MSRS Official Documentation. This file contains a set of information such as user guides and tutorials of all the set of applications. Reading is highly recommended.
- **Microsoft Dss Manifest Editor**: it allows editing the manifests from a graphic user interface.
- **Packages**: all the update packages installed are here.
- **Run Dss Node**: it executes in the computer a DSS node as host.
- **Visual Programming Language**: it's the application we are interested in. It allows the user to program without a writing code.

Microsoft Robotics Studio Architecture

The primary task of robotics applications is to consume sensory input from a range of sources and orchestrate a set of actuators to respond to the sensory input in a manner that achieves the purpose of the application. As an example, the robotics application data-flow diagram shown below contains a simple bumper (sensor) that reports when it is hit, a message box (actuator) that controls the display, and an orchestrator that connects the pieces together.

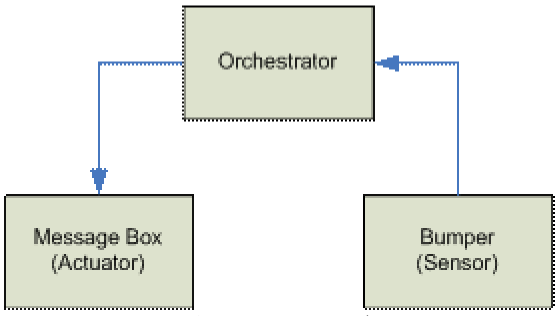


Figure 2. Example

The bases of the MSRS architecture are the runtimes: ***Concurrency and Coordination Runtime (CCR)*** and ***Decentralized Software Services (DSS)***.

CCR is a managed code library (DLL) accessible from any language targeting the *.NET 2.0. Common Language Runtime*. The *CCR* addresses the need of service-oriented applications to manage asynchronous operations, deal with concurrency, exploit parallel hardware and deal with partial failure. It enables you to design your application so that its software modules or components can be loosely coupled; that is, so that they can be developed independently, and makes minimal assumptions about their runtime environment and other components.

The *CCR* is appropriate for an application model that separates components into pieces that can interact only through messages. Components in this model need means to coordinate between messages, deal with complex failure scenarios, and effectively deal with asynchronous programming.

DSS provides a lightweight, service oriented application model that combines key aspects of traditional Web-based architecture (commonly known as *REST*) with pieces of Web Services architecture. The application model defined by DSS builds on the *REST* model by exposing services through their state and a uniform set of operations over that state but extends the application model provided by *HTTP* by adding structured data manipulation, event notification, and service composition.

The primary goal of *DSS* is to promote simplicity, interoperability, and loose coupling. This makes it particularly suited for creating applications as compositions of services regardless of whether these services are running within the same node or across the network. The result is a highly flexible yet simple platform for writing a broad set of applications. DSS uses HTTP and DSSP as the foundation for interacting with services. DSSP is a lightweight SOAP-based protocol that provides support for manipulation of structured state and for an event model driven by changes to the structured state. DSSP is used for manipulating and subscribing to services and hence compliments HTTP in providing a simple, state-driven application model.

The DSS runtime is built on top of CCR and does not rely on any other components in Microsoft Robotics Studio. It provides a hosting environment for managing services and a set of infrastructure services that can be used for service creation, discovery, logging, debugging, monitoring, and security.

A **service** is the basic building block for writing applications using Microsoft Robotics Studio and is a key component of the DSS application model. Services can be used to represent anything including but not limited to:

- Hardware components such as sensors and actuators
- Software components such as User Interface, storage, directory services, etc.
- Aggregations: Sensor fusion, mash-ups, etc

Services are executed within the context of a DSS Node. A DSS node is a hosting environment that provides support for services to be created and managed until they are deleted or the DSS node is stopped. Services are inherently network enabled and can communicate with each other in a uniform manner regardless of whether they are executed within the same DSS node or across the network.

The DSS service model has been designed to facilitate reuse of services by making them easy to use and compose with each other while enforcing a very loose coupling between them.

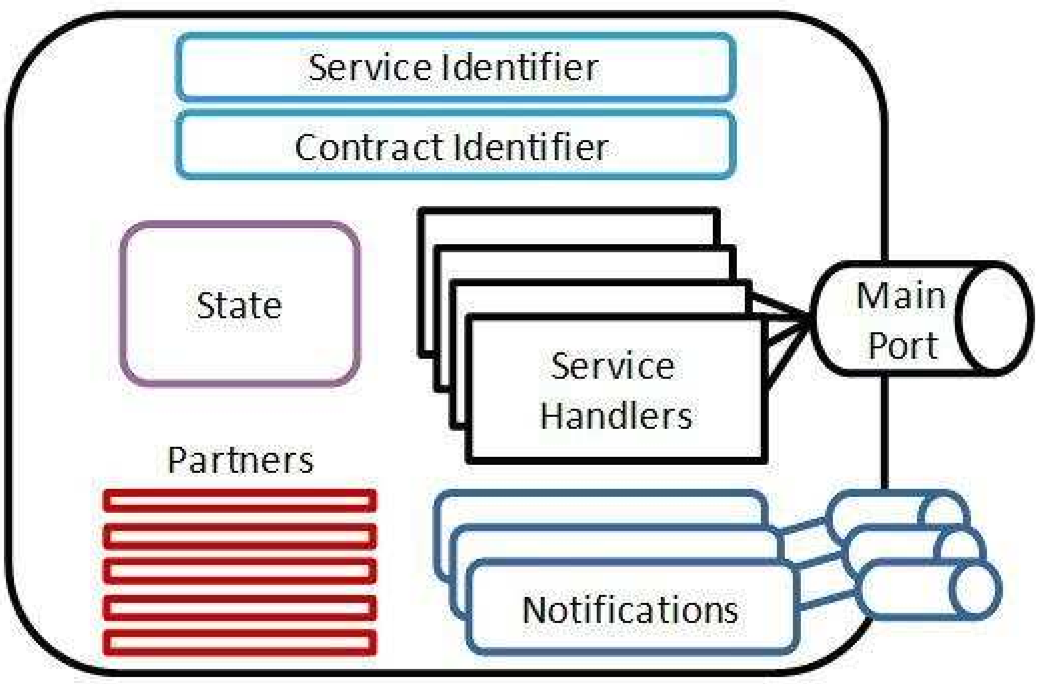


Figure 3. DSS Service Model

All DSS services consist of a common set of components:

- **Service Identifier:** The service identifier provides identity of a service instance and enables other services to communicate with that service as well as accessing DSS Services through a Web Browser.
- **Contract Identifier**: A contract is a condensed description of a service implementation that describes its behavior so that other services can compose and reuse services with a given contract. A contract identifier is a URI that uniquely identifies the contract of a service.
- **Service State**: The service state is a representation of a service at any given point in time. Any information that is to be retrieved, modified, or monitored as part of a DSS service must be expressed as part of the service state.
- **Service Partners**: A critical part of the *DSS* application model is to enable services to compose with each other to provide higher level functions. Partner services are other services that a service interacts with and possibly depends on in order to function properly. By declaring a set of other services as partners, a service can indicate to the runtime that it wants to be wired up with these services as part of the creation process of the service itself. A partner is declared using the *Partner* attribute.
- **Main Port**: The main port is a *CCR* port where messages from other services arrive (a.k.a. the operations port). A service can only talk to another service by

sending a message to its main port. The main port itself is a private member of the service class and is identified by the *ServicePort* attribute.

- **Service Handlers:** For each of the DSSP operations defined on the main port, service handlers need to be registered to handle incoming messages arriving on that port. Service handlers can be registered declaratively using the *ServiceHandler* attribute.

Visual Programming Language - Basics

VPL is targeted for beginning programmers with a basic understanding of concepts like variables and logic. However, *VPL* is not limited to novices.

A *Microsoft Visual Programming Language* data-flow consists of a connected sequence of activities represented as blocks with inputs and outputs which can be connected to other **activity blocks**.

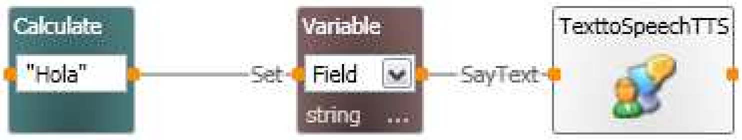


Figure 4. Block diagram with activities

Activities can represent pre-built services, data-flow control, functions, or other code modules. Activities can also include compositions of other of activities. This makes it possible to compose activities and reuse the composition as a building block. In this sense an application built in *VPL* is itself an activity.

Figure 5. Composed activity

Activity blocks typically include the activity's name and borders that represent its connection points. An activity block may also include graphics to illustrate the purpose of the activity as well as user interface elements that may enable the user to enter values, assignments, or transformations for data used in an activity.

An activity receives messages containing data through its input connection pins. An activity's input pins are connection points to its predefined internal functions known as actions or handlers (which can be either as functions provided by a service or nested data-flows). A data-flow is executed from **left to right**.

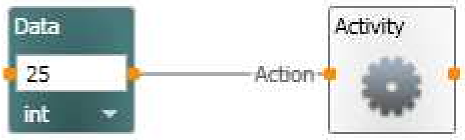


Figure 6. Activity receiving a message through its connection points

An activity may have multiple input connection pins and each with its own set of output connection pins. Output connection pins can be one of two kinds: a *result* output or *notification* output (sometimes also referred to as an event or publication output). Result outputs are displayed as rectangular connection pins while publication outputs have round connection pins.

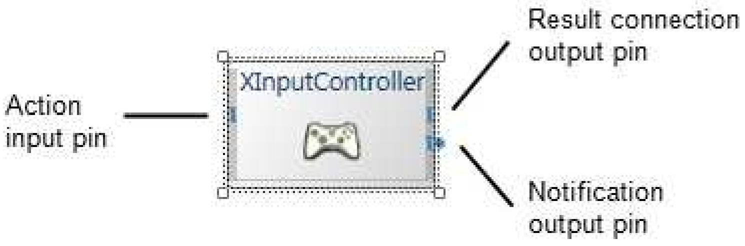


Figure 7. Connection pins

A response output pin is used in situations where an outgoing message (data) is sent as the result of a specific incoming action message. Notification pins can send information resulting from an incoming message, but more typically fire a message as a change their internal state.

Visual Programming Language – How to use

To connect two activity blocks, you drag from the connection pin of one activity to the pin of the other. If there are multiple ways (outputs and inputs) to connect the activities a Connections dialog box will appear so that you can define which of the inputs and outputs you want to connect.

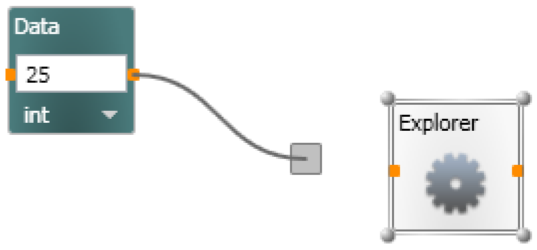
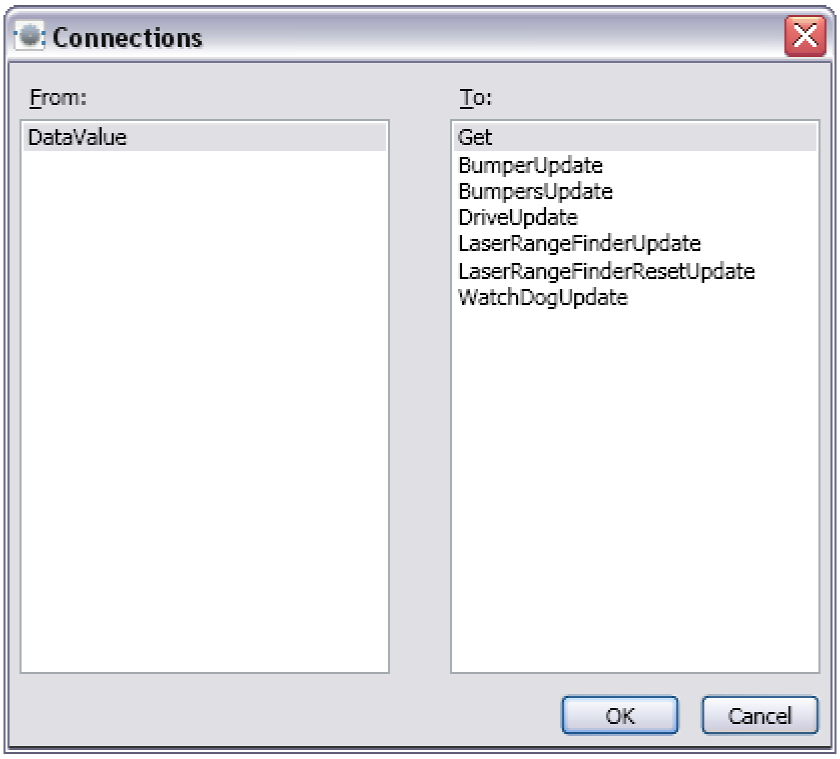


Figure 8. Connecting two blocks


Connections

From:

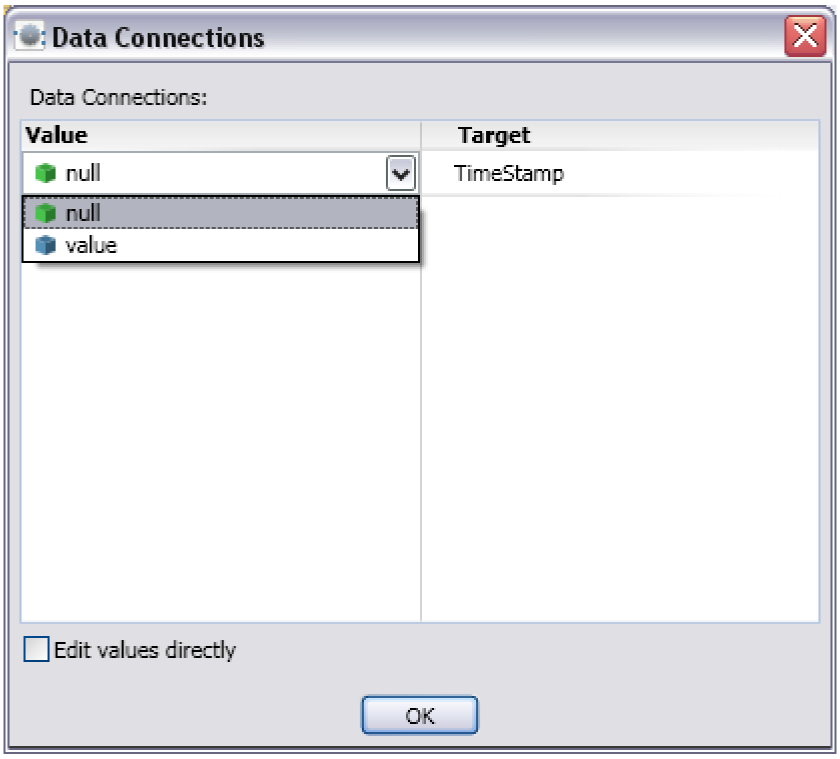
DataValue

To:

Get
BumperUpdate
BumpersUpdate
DriveUpdate
LaserRangeFinderUpdate
LaserRangeFinderResetUpdate
WatchDogUpdate

Figure 9. Possible inputs

Sometimes connections between activities also require matching up the data to be passed between the connections. When this occurs a Data Connections dialog appears and displays the message sender's data options on the left side and the receiver position on the right. Incoming data options can include the default value for its data type (0 for numeric types, false for Boolean types, and null for all other types), the value of the data, or sub-data (data member) values.


Data Connections

Data Connections:

Value	Target
null	TimeStamp

Figure 10. Matching data

If the data connection doesn't match you will be warned with ❗.

In MSRS you can create a diagram:

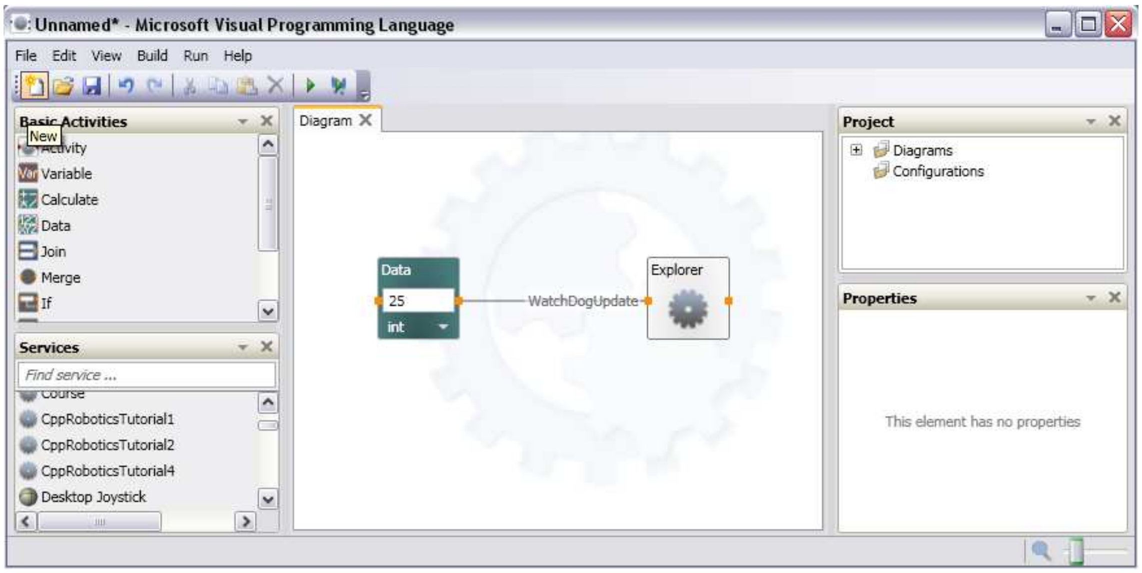
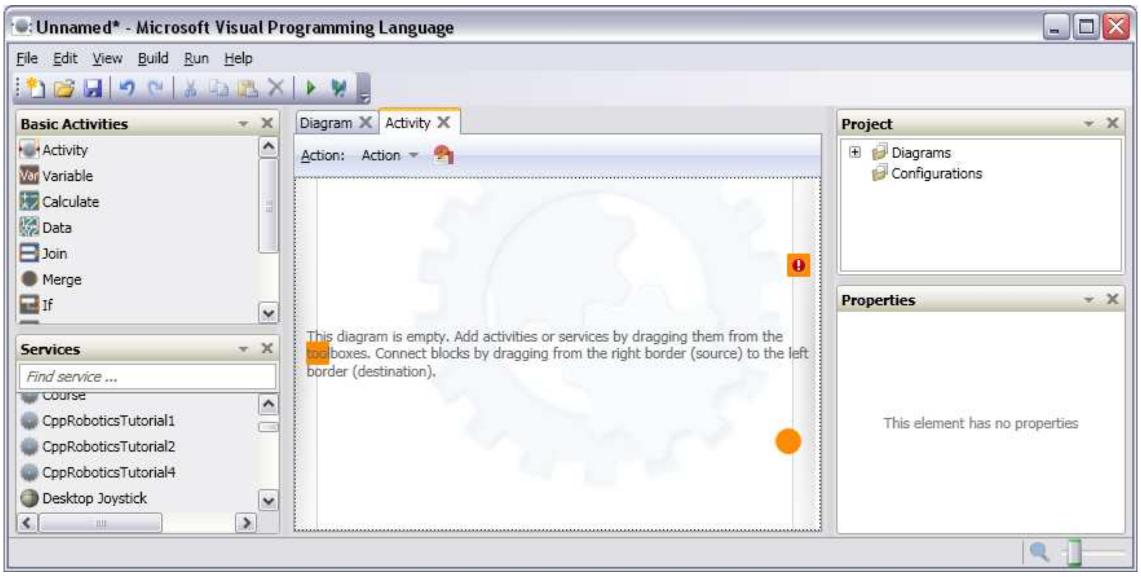


Figure 11. Creating a diagram

Or you can create an activity by dragging and dropping the basic activity "Activity" in the diagram (you will show another tab by clicking on the activity):



This diagram is empty. Add activities or services by dragging them from the toolboxes. Connect blocks by dragging from the right border (source) to the left border (destination).

Figure 12. Creating an activity

Note that the activity has input and output pins. You can edit them from *Edit* menu by clicking *Actions and Notifications*.

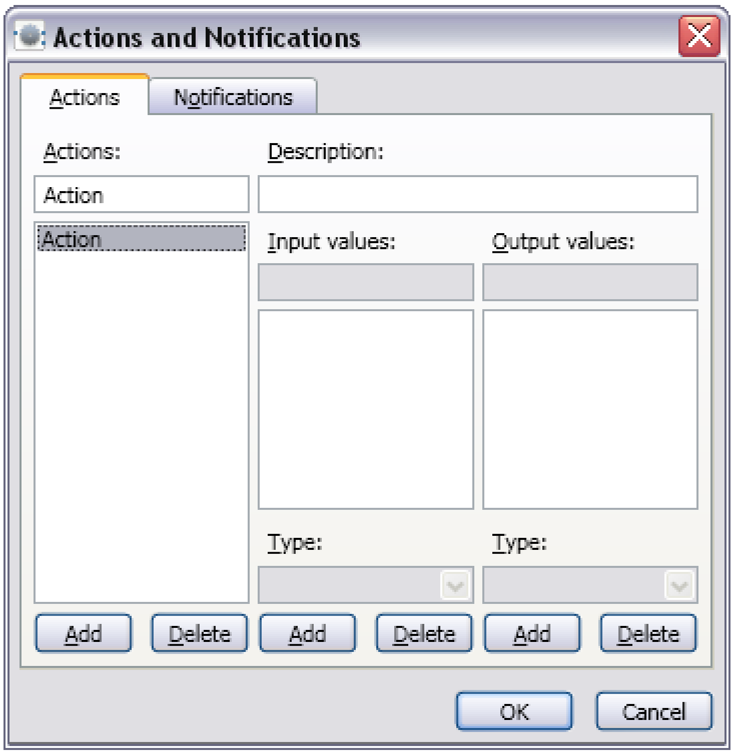


Figure 13. Actions and notifications

There are some toolboxes in a *VPL* window: *Basic Activities, Services, Project and Properties*. The toolboxes are movable and collapsible, so you can rearrange them within the main *VPL* window. In the *View* menu, you can hide or redisplay them.

To begin a dataflow diagram, drag and drop dataflow activity or service blocks from the toolboxes to the diagram area (tabbed page) and connect them. You can also add a new activity by double clicking its toolbox entry. Hovering over a toolbox entry with the pointer displays a tooltip that includes a description of how the activity may be used.

You can cut, copy, paste, or delete activities by selecting the block in the diagram using the commands on the Edit menu or the activities pop-up context menu.

At the left side you can see **Basic Activities toolbox**:

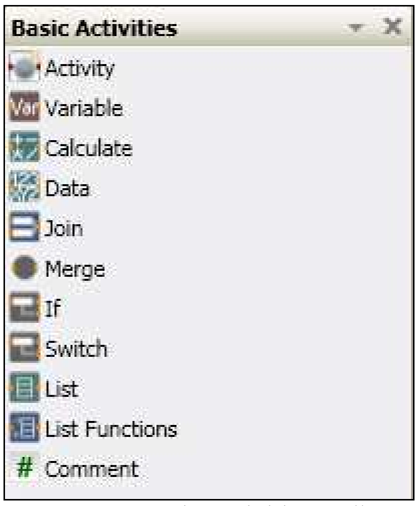


Figure 14. Basic Activities toolbox

These are the basic activities you can include in your diagram.

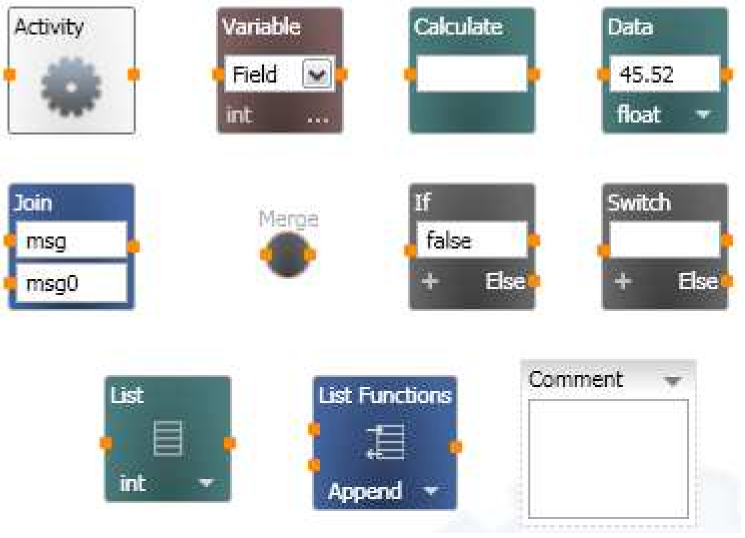


Figure 15. Basic Activities

These activities are designed to be used like in a code programming language:

- **Activity**: this activity block is used to enable you to create your own activities
- **Variable**: enables you to create a variable and set or get its value (or from the *Edit* menu selecting *Variables*)
- **Calculate**: performs simple arithmetic or logical operations on the expression entered (can include numeric values, the value of the message, its data members, or predetermined values)
- **Data**: used to supply a simple data value to another activity or service directly, without using variables
- **Join**: combines the flow of two (or more) data-flow. All messages must be received on all incoming connections before the activity passes on the data. It operates like an AND.
- **Merge**: simply merges the flow of two (or more) data-flows together without conditional dependency. It operates like an OR.
- **If**: provides a choice of outputs to forward the incoming message based on a condition you enter. It’s like classical *If* in code programming languages.
- **Switch**: can be used to route messages based on the whether the incoming message matches the expression entered into the text box. Like the classical *Switch*.
- **List**: creates an empty list of data items.
- **List Functions**: enables you to modify an existing list.
- **Comment**: enables you to add a block of text to a diagram. Like comments in code programming language.

You can use activities by dragging and dropping in the diagram.

At the right side of the window you can see the **Properties window**. The Properties window displays the properties for the current selected item. This allows you to adjust the settings exposed by that service or activity.

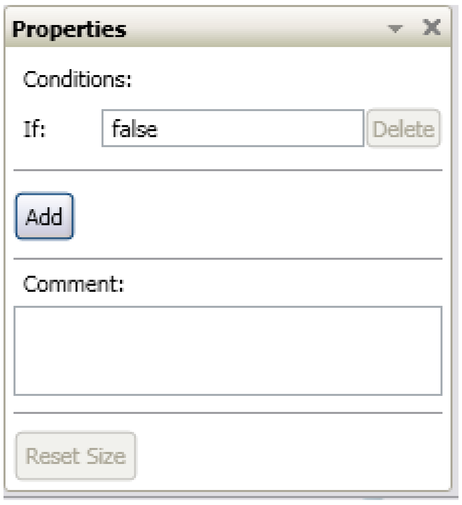


Figure 16. Properties of *If* activity

At the left side, there is the **Service Toolbox**:

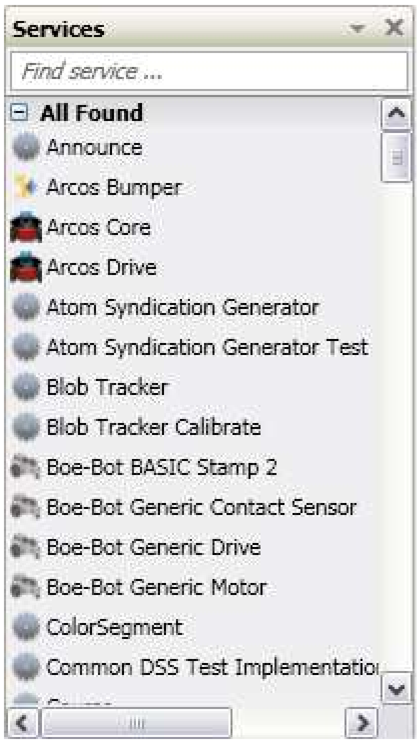


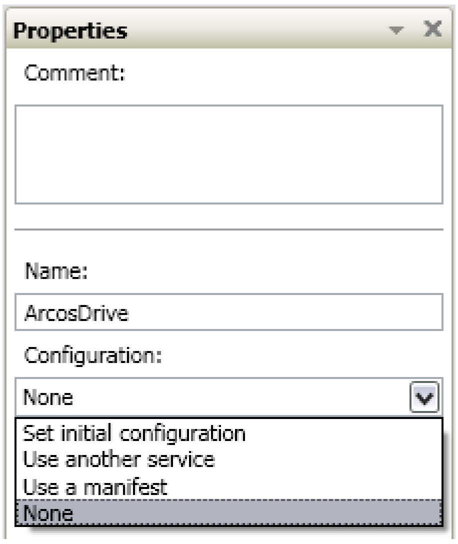
Figure 17. Service toolbox

The Services toolbox displays services that are compatible with *VPL*.

When you drag a service from the toolbox that already exists in your diagram, you are asked whether you want to create a new instance of the service or you want to create a reference to the service already in your diagram. For example you might use the same service for a set of sensors or motors, but with different configuration settings.

There is a search filter. If you write a word, the services displayed in the toolbox are the services that contain this word. You can save the search filter by clicking the + button.

Services initial state settings need to be configured:


Properties

Comment:

Name:

ArcosDrive

Configuration:

None

- Set initial configuration
- Use another service
- Use a manifest
- None

Figure 18. Starting Configuration

When you select the service, the properties window displays the initial settings that you can choose. Depending on the service, you may have one of four options to set the initial configuration of a service.

- **Set initial configuration**: Here you set the initial state values for the service (instance) and can also select what partner services (services that provide support for the service) the service uses.
- **Use another service**: You can select a specific service to use. This enables you to use generic services in your diagram and simply change the actual service used without changing your diagram.
- **Use a manifest**: This option enables you to select an existing manifest file to start the service. A manifest is a special file that describes the services to be started and their configuration. Use the Import Manifest command to display a list of existing manifests which can select. If you want to use a robot configuration, you have to choose his manifest (even simulated or real robot)
- **None**: This option indicates you want the DSS runtime to find or create an appropriate service.

In the right side there is a window called **Project**. This window shows the diagrams and configuration files (for services) included in your project. You can add or delete diagrams for this project from here.

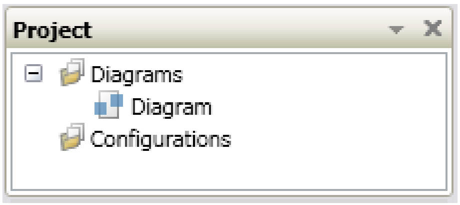
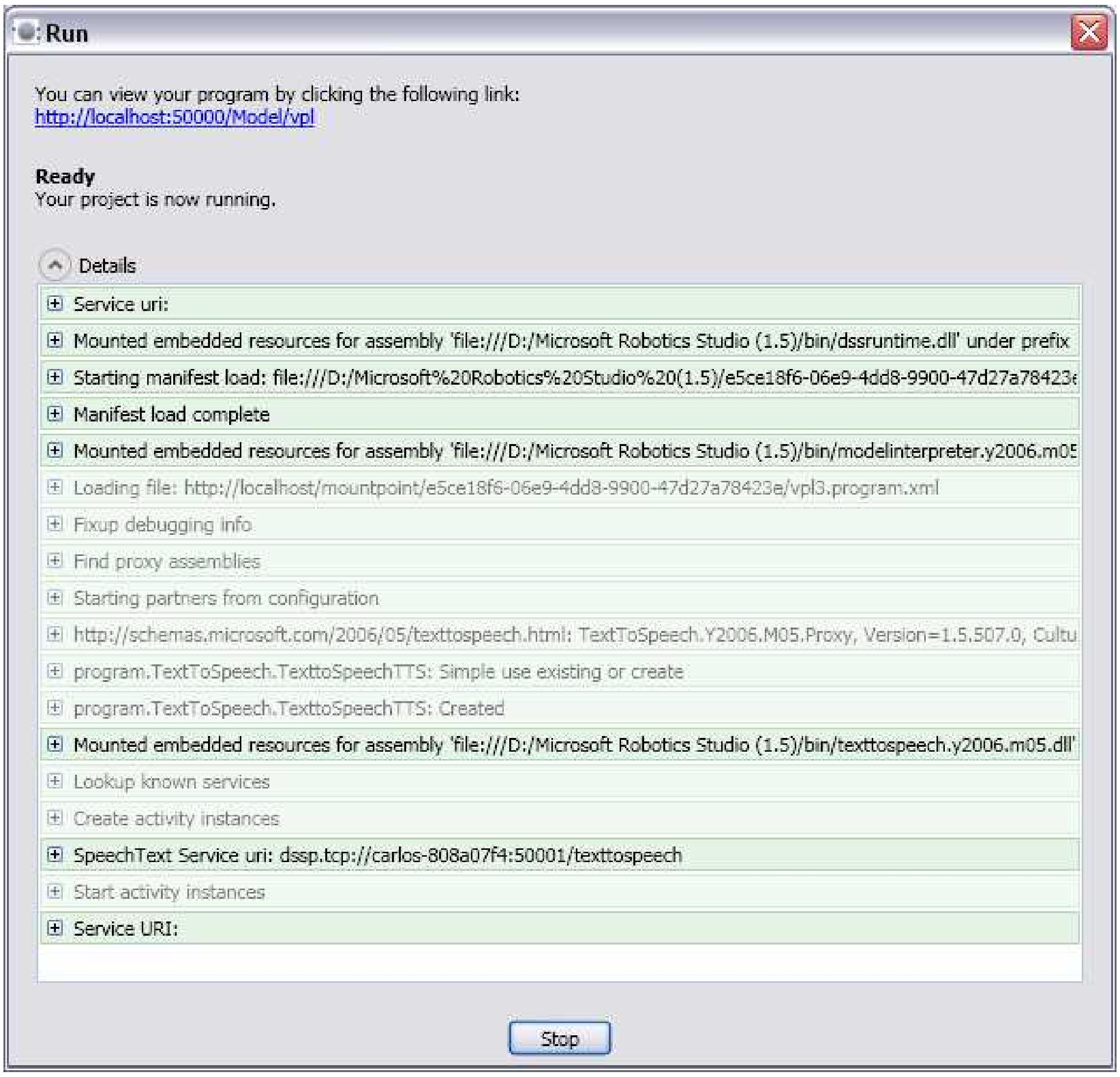


Figure 19. Project window

You can use *Zoom* on your diagram, it is located at the bottom of the window on the right side.

Visual Programming Language – Run

To run your *VPL* project, select *Start* from the *Run* menu. This will start your application. A dialog box appears with a hyperlink to a debug trace as well as a Stop button to stop your project and a set of details:



You can view your program by clicking the following link:
http://localhost:50000/Model/vpl

Ready
Your project is now running.

```
Details
⊞ Service uri:
⊞ Mounted embedded resources for assembly 'file:///D:/Microsoft Robotics Studio (1.5)/bin/dssruntime.dll' under prefix
⊞ Starting manifest load: file:///D:/Microsoft%20Robotics%20Studio%20(1.5)/e5ce18f6-06e9-4dd8-9900-47d27a78423
⊞ Manifest load complete
⊞ Mounted embedded resources for assembly 'file:///D:/Microsoft Robotics Studio (1.5)/bin/modelinterpreter.y2006.m0
⊞ Loading file: http://localhost/mountpoint/e5ce18f6-06e9-4dd8-9900-47d27a78423e/vpl3.program.xml
⊞ Fixup debugging info
⊞ Find proxy assemblies
⊞ Starting partners from configuration
⊞ http://schemas.microsoft.com/2006/05/texttospeech.html: TextToSpeech.Y2006.M05.Proxy, Version=1.5.507.0, Cultu
⊞ program.TextToSpeech.TexttoSpeechTTS: Simple use existing or create
⊞ program.TextToSpeech.TexttoSpeechTTS: Created
⊞ Mounted embedded resources for assembly 'file:///D:/Microsoft Robotics Studio (1.5)/bin/texttospeech.y2006.m05.dll'
⊞ Lookup known services
⊞ Create activity instances
⊞ SpeechText Service uri: dssp.tcp://carlos-808a07f4:50001/texttospeech
⊞ Start activity instances
⊞ Service URI:
```

Figure 20. Running the project

You can use the *Debug Start* command to start your project and stop at its first block, and enable you to step through the program. You can hide the Details by clicking the up arrow.

You can also run your *VPL* application independently from the *VPL* development environment. To do this use the *Compile As a Service* command from the Build menu. You can run the service created from the ***Command Prompt*** or clicking ***Run DSS Node*** on the *Start Menu MSRS* group.

Visual Programming Language – Simulator

If the project contains simulation, when you run it the Visual Simulation Environment will be displayed:

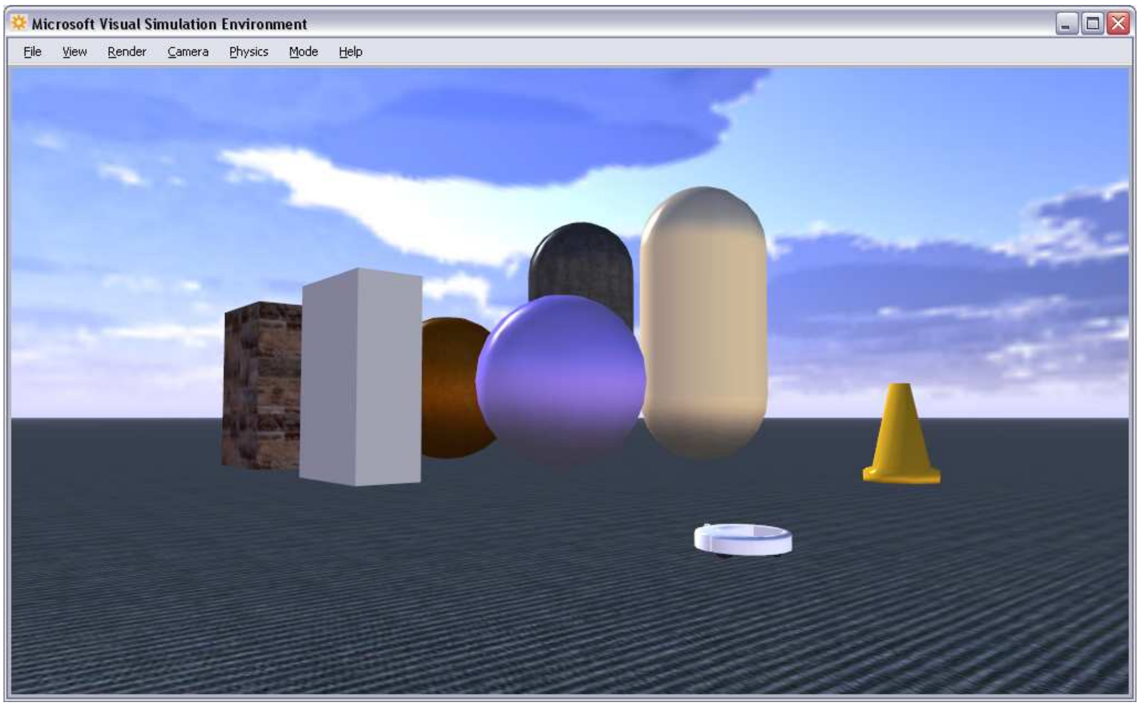

Microsoft Visual Simulation Environment
File View Render Camera Physics Mode Help

Figure 21. Visual Simulation Environment with iRobot Create

You can move the camera with the keyboard and the mouse by pushing the keys in the keyboard: *w,a,s,d,q,e* and clicking and moving the mouse.

You can configure a wide variety of settings, like the speed of the camera, the render quality or even see from a robot cam!

Walk-through

Introduction

This guide is divided in two blocks:

- Basic programming
- Robot programming

In basic programming you will learn how to program very simple routines with Visual Programming Language, like in C language or other programming languages.

In this tutorial you will learn how to use most of the basic activities to program an application and some services. The application will be a desktop joystick with buttons (0-9) and you will assign to some of them a function.

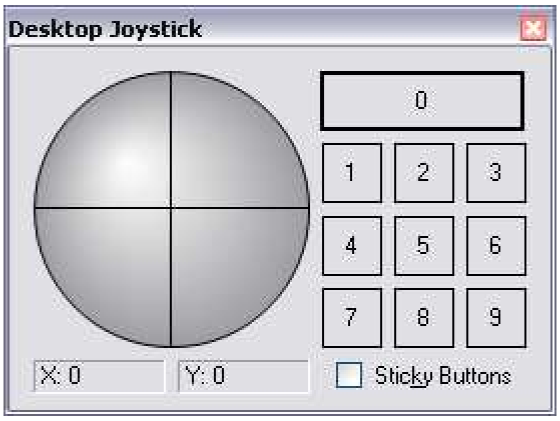


Figure 22. Desktop Joystick

The functions associated with the buttons are:

- A counter:
    - 0: reset the counter
    - 1: increment counter in one
    - 2: show a simple dialog with the counter value
- A simple dialog showing if some buttons are pressed or not pressed
- The computer will speak (text to speech service)

This tutorial introduces the user to the environment with many features but only in one practice.

In robot programming you will learn how to program and control a robot and simulate it. It's divided in three practices:

- Controlling a robot
- Moving a robot
- Sensing and acting

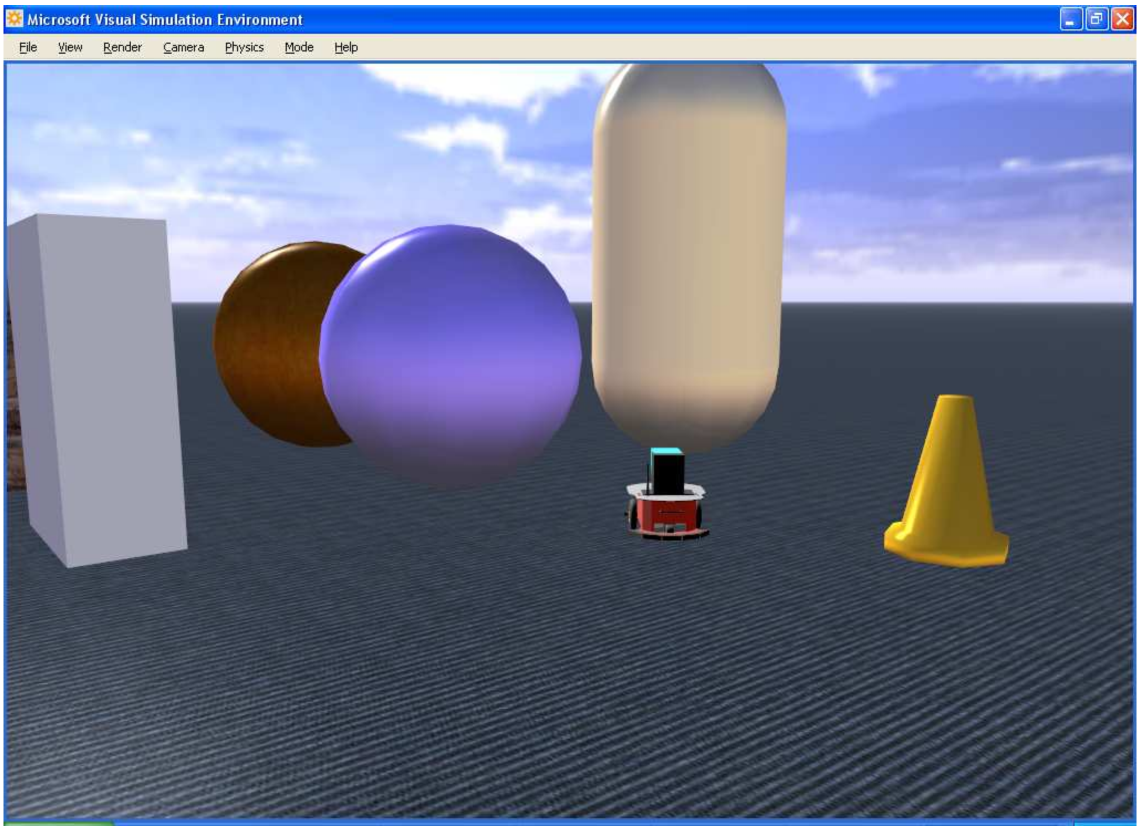

Microsoft Visual Simulation Environment
File View Render Camera Physics Mode Help

Figure 23. Robot simulation environment

In these tutorials you will learn how to control a robot with a desktop joystick (quite similar to the basic programming application), how to program a set of movements and how to access to its sensors and actuators to get data from them or to send orders.

Basic programming

To begin, create a new project:

1. File – New

An empty page will appear. Now you have to search the **Desktop Joystick** in the **Services list**. You have to install the samples update package to have the Desktop Joystick service.

1. Go to **services** and write in the box: **<u>Desktop Joystick</u>**
2. Drag to the diagram the **Desktop Joystick** service

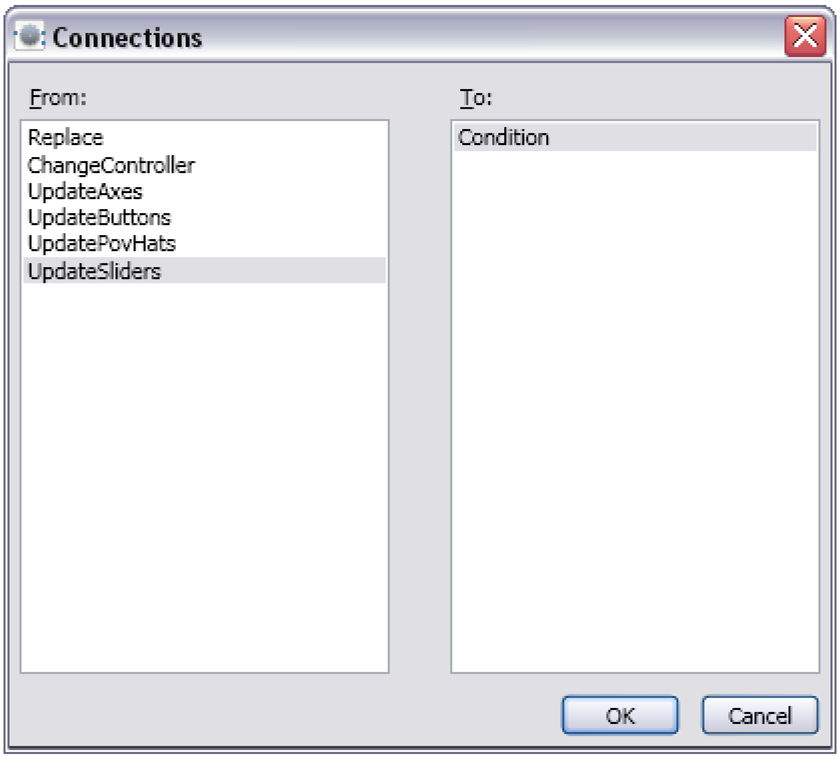
Figure 24. Desktop Joystick icon

Now drag an **If activity**:

1. Go to **Basic Activities** and drag the **If** in the diagram

You have to connect the notification pin (the round one) from the **Desktop Joystick** to the **If** input pin. **Notification output** will send a message every time an event occurs with the joystick, we will use it when you push a button.

When you connect the pins a window will appear:



From:
- Replace
- ChangeController
- UpdateAxes
- UpdateButtons
- UpdatePovHats
- UpdateSliders

To:
- Condition

Figure 25. Connections

In the **From** side you have to select what will be sent by the output pin of the **Desktop Joystick**. In this case we are interested in the buttons, so select **Update Buttons**. The **If** input doesn't have options, only **Condition**. You should have a simple diagram like this:

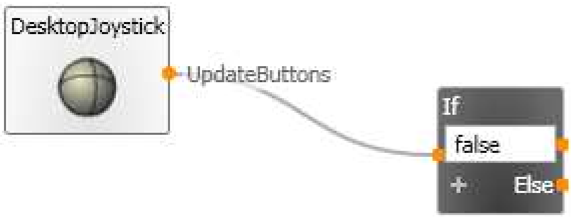


Figure 26. Diagram

If you write in the **If** condition you will see a set of options as a condition. The only **Desktop Joystick** options are **Pressed** and **TimeStamp**.

We are interested in **Pressed**. It is a list (in fact, it works like an array in classical programming language) of Booleans that are associated to all the buttons. For example, to access button "5", you have to write Pressed[5].

We will add conditions to **If**:

1. Select the **If**

2. In the right side, in **Properties** you will see a condition.
3. Write: **<u>Pressed[0] = true</u>**
4. Now add more conditions by pressing **Add** button:
    - <u>Pressed[1] = true</u>
    - <u>Pressed[2] = true</u>
    - <u>Pressed[8] = true</u>
    - <u>Pressed[9] = true</u>

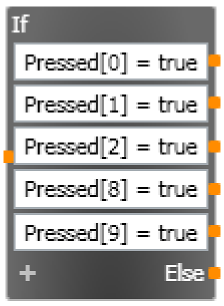


Figure 27. If activity

We can't use a switch because **Pressed** is not a variable, it is a list. Note that you can change the zoom of the view of the diagram in the right bottom of the window.

You will create the counter. The button 0 will reset the counter (pushing the 0 value to the variable), the button 1 will adds 1 (counter+1) and the button 2 will show a dialog with the counter value.

1. Drag a **Data** activity (don't change the value, we want a 0 integer)
2. Drag another **Data** activity, change the data type to **string** and write: <u>Counter reset</u>
3. Drag **Variable** activity and create an **integer** variable named "Counter" (you will access a window from where to define variables by clicking "…")
4. Add a **Simple Dialog** service

You already have all the necessary to make running a button. Connect the boxes:
1. Connect *"If pressed[0]"* to the first **Data** box (0 int)
2. Connect this **Data** box to the another one and the **Variable** box
    - Select **SetVariable**
3. Connect the **Data** box with the text "Counter reset" to the **Simple Dialog** service
    - Select **AlertDialog**
    - Select "**value**" in the next window

If you Run the program now the button 0 will have the desired behavior.

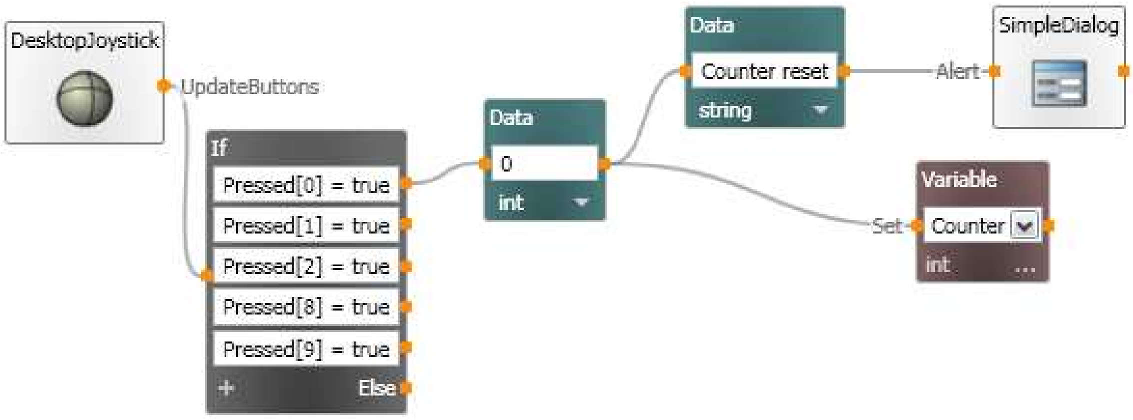


Figure 28. The diagram should look like this

The next button is *"If pressed[1]"*:

1. Copy and paste the **Variable** you used (note that it must contain the variable *Counter*) or drag a new **Variable** box and select *Counter*.
2. Connect the output pin of the **If box** (*If pressed[1]*) to the input pin of the **Variable**
   - Select **GetValue**.
3. Drag a **Calculate Activity** and connect it with the **Variable**.
4. Write in the **Calculate**: <u>Counter + 1</u>
5. Now place a **Merge Activity** on top of the line of the variable from the *If pressed[0]* sequence. It will split the line.
6. Connect the **Calculate** box with the **Merge** you have placed in step 5

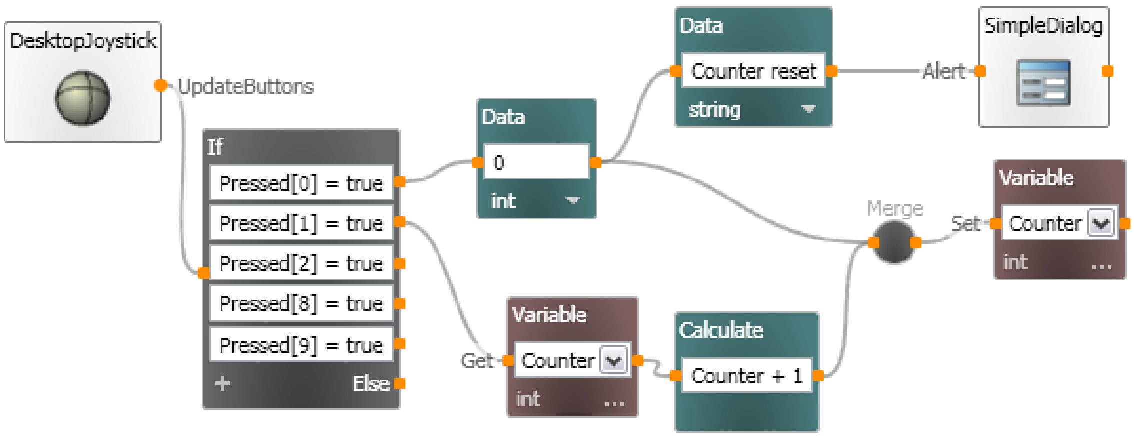
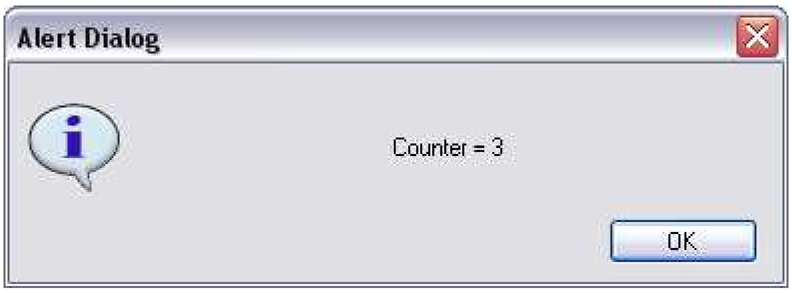


Figure 29. Diagram with 2 buttons

By pushing "2" this dialog will be displayed:



Counter = 3

Figure 30. Counter value

Note that you can use zoom or scroll left or right. The next button is *"If pressed[2]":*

1. Copy and paste the **Variable** you used (note that it has to content the variable *Counter*) or drag a new **Variable** box and select *Counter*.
2. Connect the **If** box output for button 2 with the **Variable**
   - Select **GetValue**.
3. Drag a **Calculate Activity** and connect it with the **Variable**.
4. Write in the **Calculate** the whole line(don't copy it, it won't work)**:** "Counter = " + Counter *(it works quite similar with printf in C)*
5. Now place a **Merge Activity** on top of the line that is connected to the **Simple Dialog** activity.
6. Connect the **Calculate** with that **Merge** (which is connected to **Simple Dialog**)
7. **Run**

When you run an application you have to save it. Save whenever you want, with the name you wish. The counter application is now **complete**:

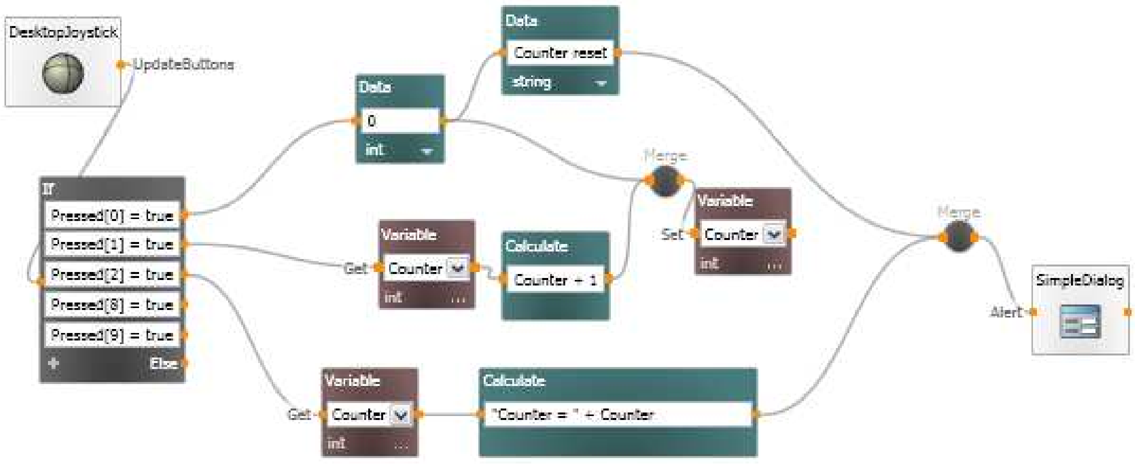


Figure 31. Counter application

It will count the number of times you pushed 1. You can reset this value by clicking 0 and show it by clicking 2.

Now you will add some new features to the application (button 8):

1. Drag two **Calculate** activities to the diagram
2. Connect the *If pressed[8]* to both
3. Write in the boxes the whole line(don't copy it, it won't work):
   - "Button 8 state: " + Pressed[8] + "\n"
   - "Button 9 state: " + Pressed[9] + "\n"
4. Drag a **Join** activity and connect each button with each message.
5. Place a new **Calculate** activity and write: msg + msg0
6. Connect the **Join** output with this **Calculate** input
7. Connect the **Calculate** with the **Merge** connected to the **Simple Dialog** you have created before (in counter application).

The last button that you will use (9):

1. Add a **Data** activity to the diagram and connect button 9 to it.
2. Select **string** and write: You have pushed button 9
3. Drag a **TextToSpeech** service. Search for it in the Service Toolbox "Find service" box. This service is used to talk by the computer speakers.

4. Connect the **Data** activity to the **service**:
    - Select **SayText** in the first formulary
    - Select **Value** in the last one

Now you can **Run** it. If you get a message asking whether to unblock the application or not, select Unblock. The diagram should look like this:

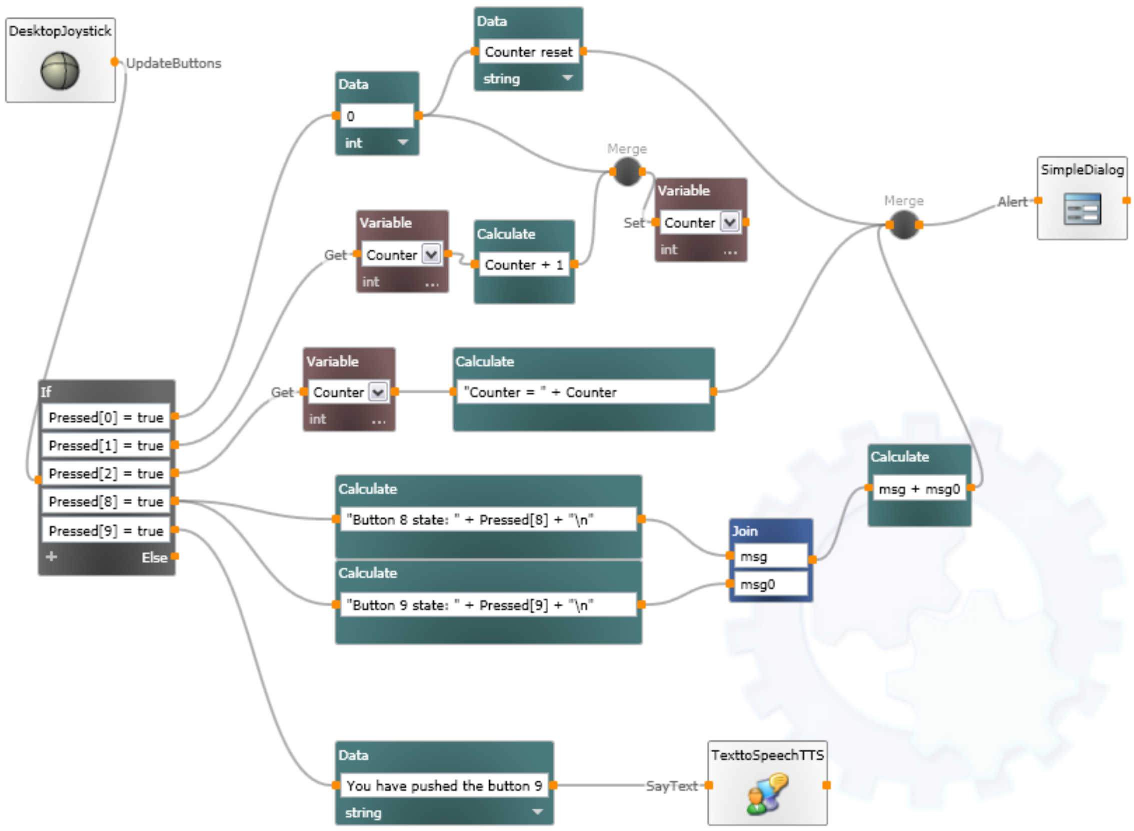


Figure 32. Basic Programming Diagram

You have finished **Basic Programming Tutorial**! The functions associated with the buttons are:

- [0-2] Counter application:
    - 0: resets the counter
    - 1: increments the counter in one
    - 2: shows a simple dialog with the counter value
- [8] A simple dialog that shows if buttons 8 and 9 are pressed or not
- [9] The computer will speak (text to speech service)

Robot programming

This tutorial is designed to be executed in several steps. First, you will create a diagram with a robot and you will control it. Secondly, you will create a robot that moves on its own and finally you will create a robot with sensors.

Controlling a robot

1. File-New
2. Add the service: **DirectionDialog**

Direction Dialog is a virtual controller (similar to Desktop Joystick) with only four directions and a *Stop* button.

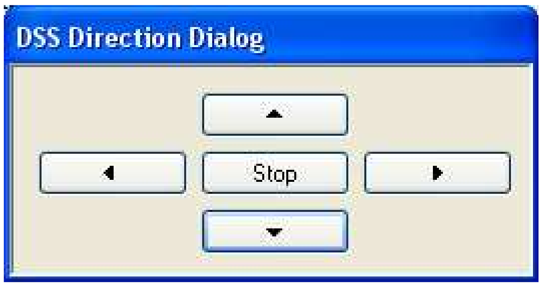


Figure 33. Direction Dialog

With that dialog you will be able to control a robot to go forwards, backwards or stop it. Next:

1. Drag a **Calculate** activity
2. Connect the **notification** pin of **DirectionDialog** to the **Calculate** input
    a. Select **ButtonPress**
3. In the **Calculate** box write <u>Name</u>

Name is the variable containing the name of the button pressed. The names are:
1. Forwards
2. Backwards
3. Stop
4. Left
5. Right

We will use a **Switch**:
1. Add **Switch** activity
2. Add the **cases**:
    a. <u>“Forwards”</u>
    b. <u>“Backwards”</u>
    c. <u>“Stop”</u>
3. Connect **Calculate** box with the switch
4. Create 3 **Data** activities
5. Write in them:
    a. <u>0.5</u>
    b. <u>-0.5</u>

    c. 0
6. Connect the 3 cases with the 3 diagrams
    a. "Forwards"_ 0.5
    b. "Backwards" _ -0.5
    c. "Stop" _ 0

These are the values for the robot movement. Now you will add a Generic Differential Drive service; it is called generic because it supports common operations available on most differential drives, but is not associated by default with a specific drive system. One of the benefits of Robotics Studio is that you can write general programs which will run on a variety of robots.

1. Add a **Merge** activity
2. Connect the 3 **Data** boxes to it
3. Add **Generic Differential Drive** service
4. Connect the merge to it
    a. Select **SetDriveSpeed**
    b. Select **Value** in both cases

The diagram should be like this:

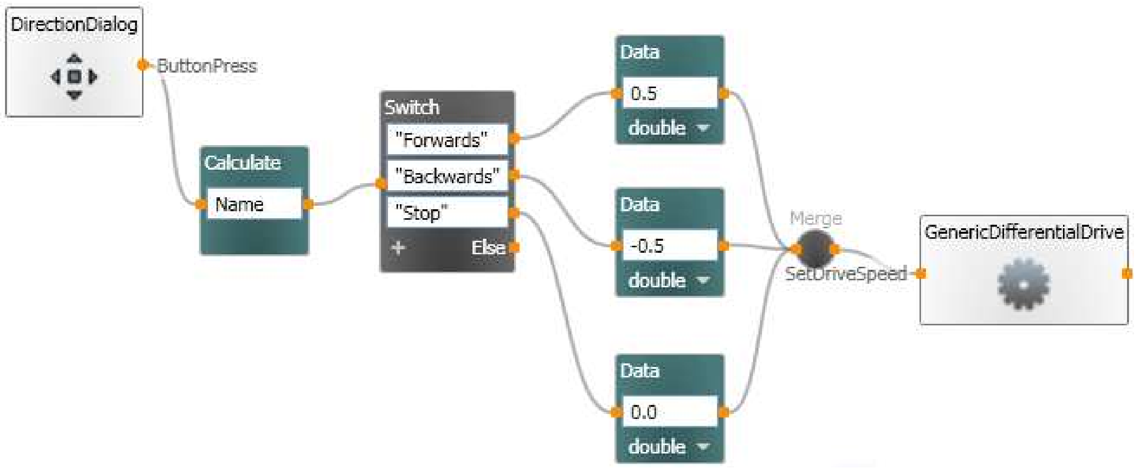


Figure 34. Control a robot diagram

Generic Differential Drive is used to control robots that commonly have wheels. You can control left wheels and right wheels separately. However, in this case, by introducing the same value for both wheels the robot will move in one direction. You must configure the drive:

1. Select **Generic Differential Drive**
2. In the properties toolbox go to **Configuration** and select **Use a manifest**
3. Click on **Import**
4. Select **MobileRobots.P3DX.Simulation.manifest.xml**

To run in **simulation environment** you must select a manifest that could be simulated. This robot has a camera that can be used in the simulation environment, so you will select it instead of another one. Usually the manifests you can simulate have the word "Simulation" on their names. Let's see it:

1. Click on **Run** or press **F5** (if you have some problem please refer to **Installation requisites)**.

Now you should be able to see that:

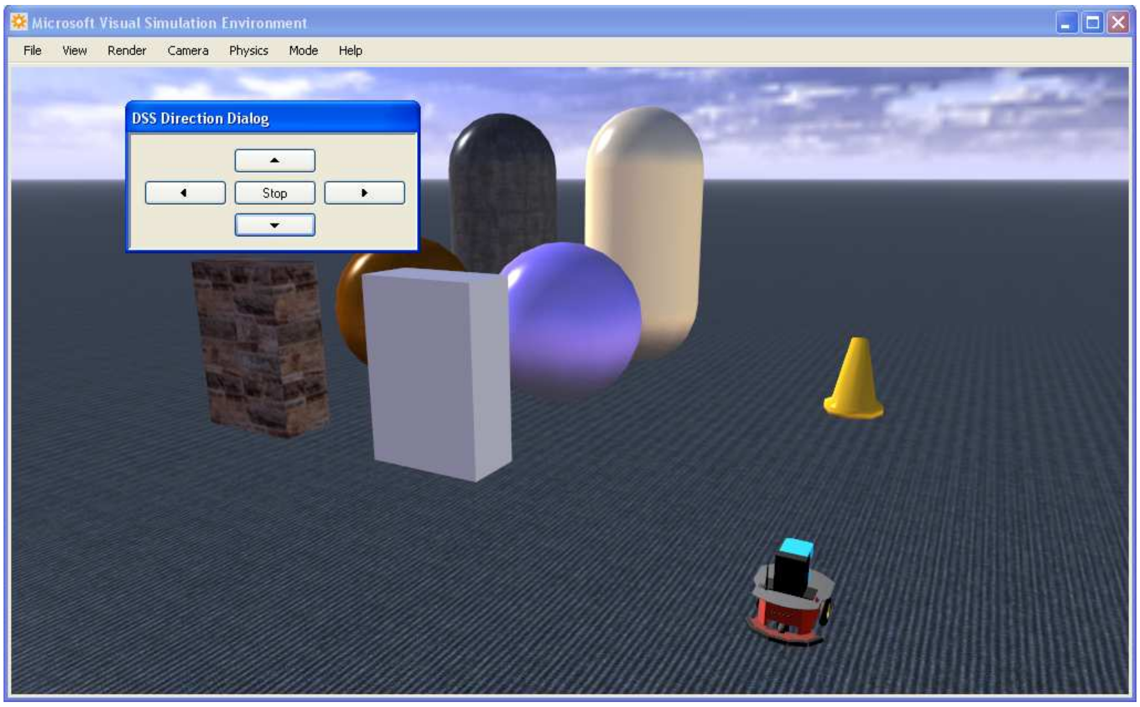


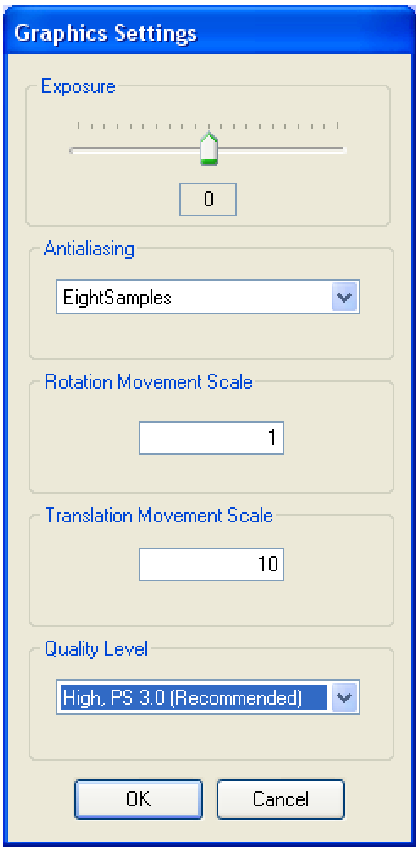
Figure 35. Simulation environment

If you click the buttons *forwards* or *backwards* the robot will move in that direction and if you click the *Stop* button it will stop its movement.

You can move around the simulation world with the mouse and the keyboard. Keyboard: arrows or: a,s,d,q,w,e.

The simulation environment has multiple options. Let's configure some of them:

1. Click on **Render** menu
2. Click on **Graphics Settings**


Graphics Settings

Exposure
0

Antialiasing
EightSamples

Rotation Movement Scale
1

Translation Movement Scale
10

Quality Level
High, PS 3.0 (Recommended)

OK
Cancel

Figure 36. Graphics Settings

In this menu you can configure some graphic properties related with graphics' quality. Configure **Antialising** and **Quality Level** as your computer permits. Rotation and Translation are settings of movement. Try some values and choose the most comfortable for you.

In the **Camera** menu you can select **MainCamera** or **RoboCam**. If you select the latter you will be able to see what the robot is seeing through its integrated cam.

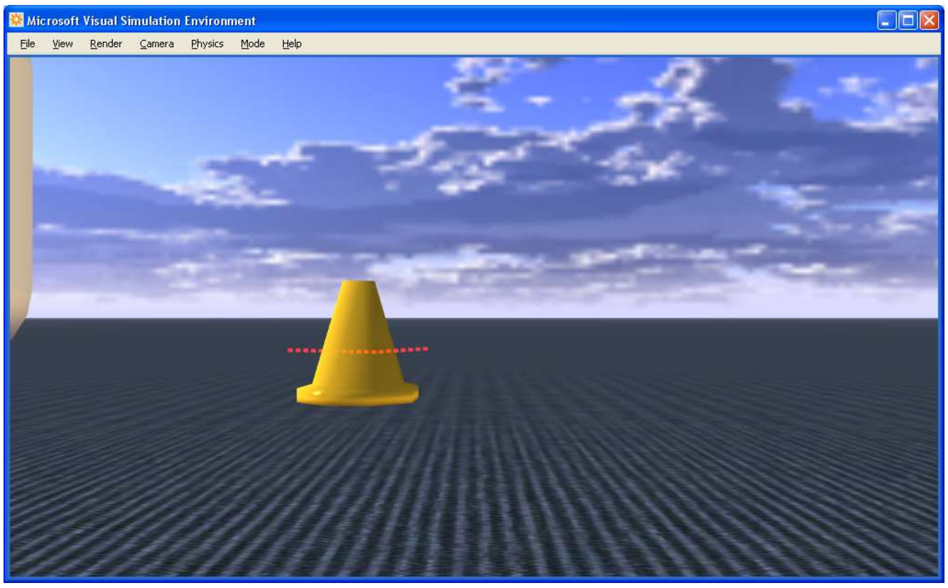


Figure 37. RoboCam

Phase one (controlling a robot) is complete. Now you will make a robot move on its own.

Moving a robot

In this practice you will create a robot that moves on its own. It will have some movements and a timer. The timer will control the time that every movement is executed. When a movement finishes it starts next one.

Start a new project.

1. File -> New
2. Drag two **Data** activities
3. Drag a **Variable** activity
4. Drag a **Timer** service

Firstly, you will set a starting value to Timer and variable. In this tutorial you will use a Timer to control the robot movement. The timer value is set in milliseconds (data integer), and it must be initialized.

1. Set a **Data** activity to 1500 integer and let the another to be a 0 integer
2. Create an **integer** variable and name it Case
3. Connect the **Data** activity (1500) to the **Timer**
   a. Select **SetTimer**
4. Connect the another **Data** activity to the **Variable**

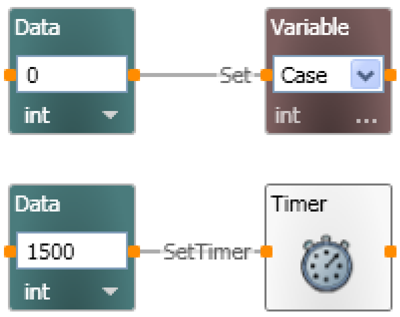


Figure 38. Initializing values

Now you will create the application.

1. Copy and paste the **Timer** (it must be the same timer, another instance, not a new one)
2. Copy and paste the **variable Case**
3. Connect them from the notification pin to the input pin
   a. Set **TimerComplete**
   b. Set **GetValue**
4. Drag a **Calculate** activity and connect the **variable** to it
5. Write in the calculate box: Case (the name of the variable)
6. Drag a **Switch** with the values 0,1 and 2
7. Connect the **Calculate** box to it

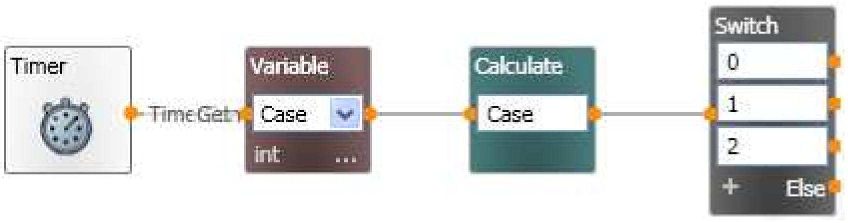


Figure 39. The resulting diagram

When the timer is complete the next iteration starts. The switch has 4 cases:

1. Move forwards
2. Move backwards
3. Move turning left or right
4. Restart the loop

Every new iteration will reset the timer and will add 1 to the case variable. You will program this now, and then all the cases must be connected to this figure:

1. Drag two **Variables** (<u>Case</u> variable, you can copy the instance)
2. Drag a **Calculate** activity
3. Connect the first **Case** to the **Calculate**
4. Write: <u>Case + 1</u>
5. Connect the **Calculate** to the another **Variable**
    a. Select **SetValue**
6. Create a **Data** integer with the value: <u>2000</u>
7. Create another instance of the same **Timer** you have created before
8. Connect the **Data** activity to it
    a. Select **SetTimer**
9. Drag a **Merge** activity
10. Connect **it** to the two flows
    a. Connect it to the first **Variable**
        i. Select **GetValue**
    b. Connect the merge to the **Data** (2000)

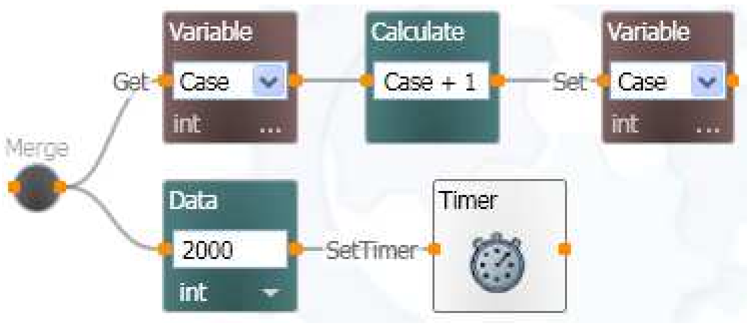


Figure 40. New iteration requisites

The three cases of the switch will be connected to this figure to make the application advance. The "Else" case must reset the application (calculate=0).

1. Copy the entire figure (Figure 40)
2. Connect **Case "else"** to the Merge activity
3. Change the **Calculate** activity with a **Data** activity with the value <u>0 integer</u>

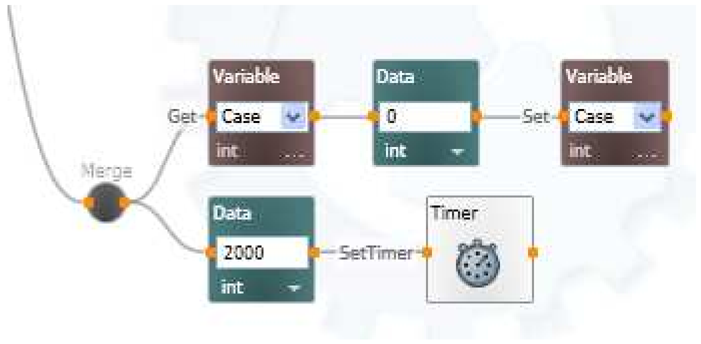


Figure 41. “Else” diagram

Now you will program the robot movement:

1. Drag a **Data** activity: set as **Double** with value 1.0
2. Drag a **Generic Differential Drive**
    a. Configuration: **Use a Manifest**
    b. Select: **MobileRobots.P3DX.Simulation.manifest.xml**
3. Connect the switch **case 0** to the **Data** activity
4. Connect the **Data** activity to the **Generic Differential Drive**
    a. Select **SetDrivePower**
5. Connect the switch **case 0** to the **Merge** you have created before

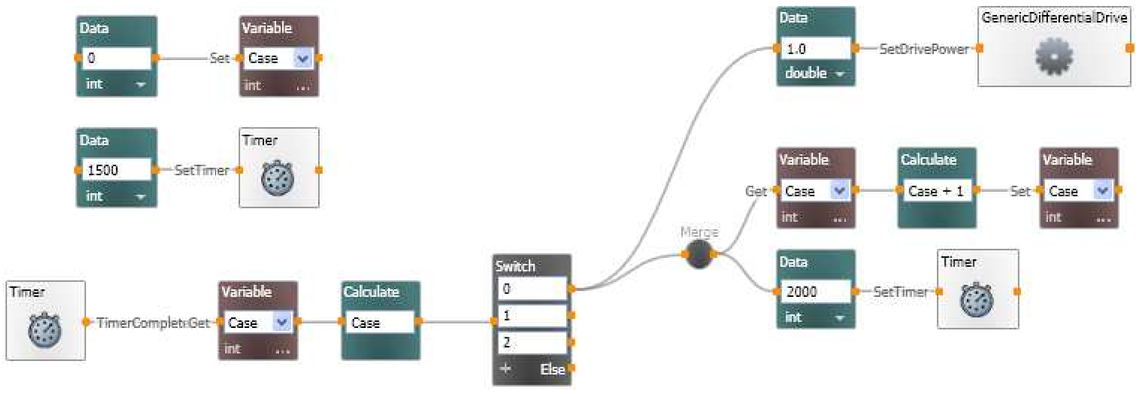


Figure 42. Resulting diagram

This is the diagram with the first case working. The second case is very easy:

1. Connect **Case 1** to the **Merge**
2. Drag a **Data** with value -1.0
3. Copy the **GenericDifferentialDrive** service (a new instance of the same one) and connect the **Data** box to it
    a. Select **SetDrivePower**
4. Connect **Case 1** to this **Data** box

Note: you can copy the **Case 1** diagram and change only the attributes.

You will now add the **Case 2** control program. You will use a new service: **TurningRadiusToWheelPowers**. You will use this service with the Generic Differential Drive. This service is used to move the robot in circles. You can select the side you want to move and the radius.

1. Connect **Case 2** to the **Merge (**controlling loops)
2. Drag two **Data** activities with the values:
    a. 6.0 Double

    b. False Bool
3. Connect **Case 2** to both
4. Drag a **Join** activity and name its messages as:
    a. TurningRadius
    b. TurnRight
5. Connect them:
    a. 6.0 Double - TurningRadius
    b. False Bool - TurnRight
6. Drag a **TurningRadiusToWheelPowers** service
7. Connect the **Join** output to it
    a. Select **CalculateWheelPowers**
    b. Select values with the same name
8. Create another instance of the **GenericDifferentialDrive** (by dragging the service or by copying another one)
9. Connect **TurningRadiusToWheelPowers** output to it
    a. Select **CalculateWheelPowers – Success**
    b. Select **SetDrivePower**
    c. Select **Left** for **LeftWheelPower** and do the same for the Right one
10. **Run**!

You can see the robot moving on its own and doing the same movement all the time. The diagram should be like this:

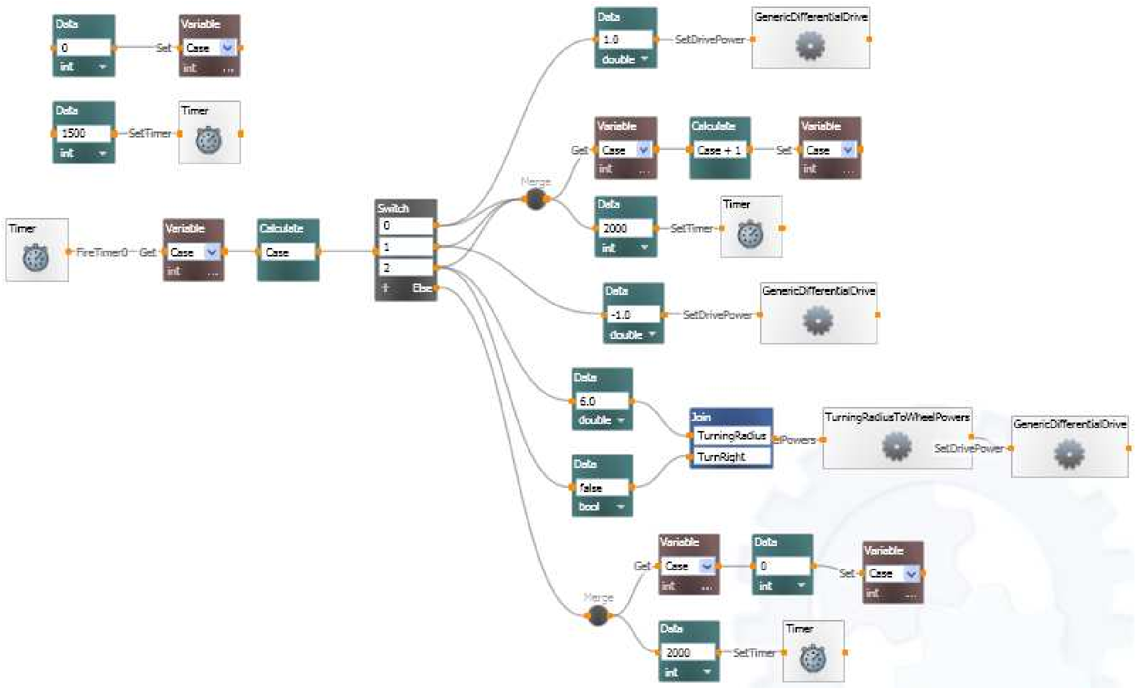


Figure 43. Final diagram of second robot

The robot will crash with the objects and will throw them to the ground.

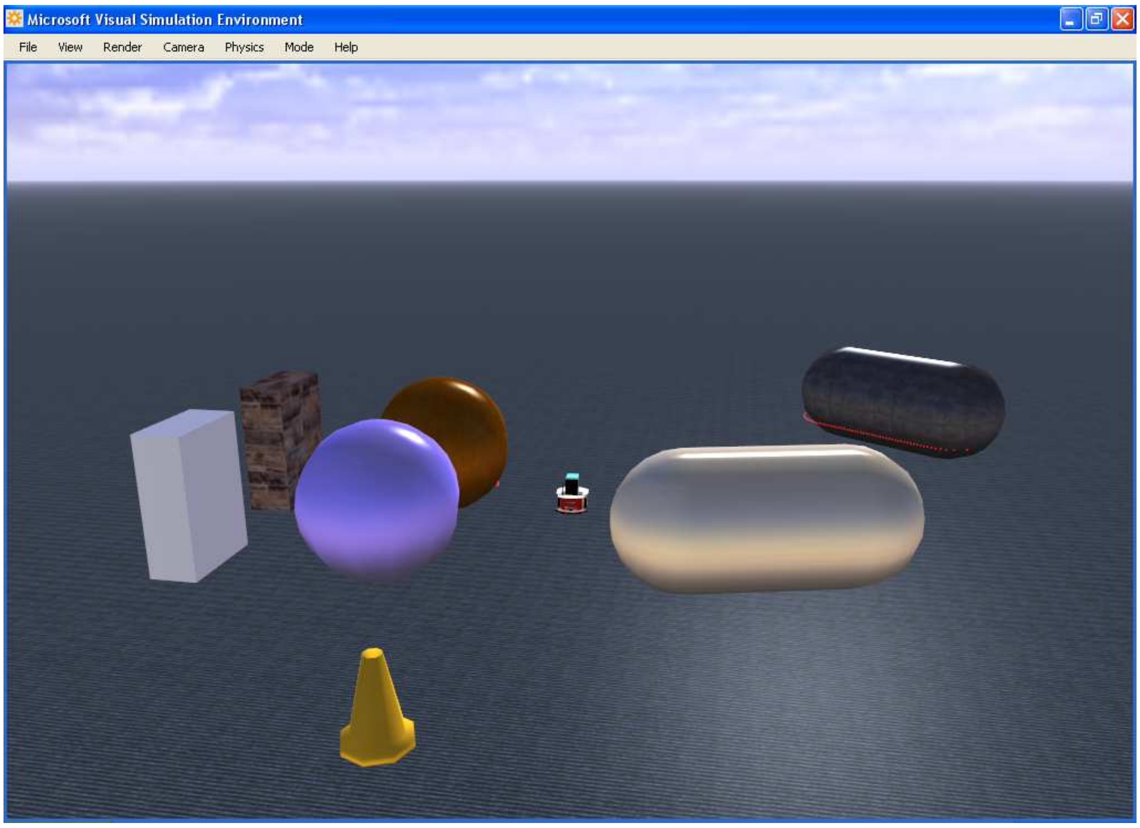


Figure 44. Robot moving itself and throwing objects

Sensing and acting

In this last practice you will interact with an *iRobot Create* and his sensors and actuators.

Figure 45. iRobot Create

It has sensors to detect walls and objects and a “music” player with some “songs”. These songs are beep sounds of the computer. This is a very simple tutorial to know how to interact with the robot.

You will create two flows: one to move forwards and one that works like an interruption (active when sensor is updated) that makes the robot turn.

1. File->New
2. Create a **Data** <u>integer</u> with value <u>2000</u> and a **Timer**

3. Connect them

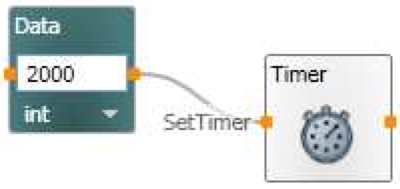


Figure 46. Initialization of the variable

This is the initialization of the timer. You will create the robot behavior now. First you will program the forwards movement:

1. Copy an instance of the **Timer**
2. Drag a **Data** <u>double</u> with the value <u>1.0</u> (you can choose higher speeds, but they may cause the robot to overturn)
3. Connect the **Timer** output notification pin with the **Data** input pin
    a. Select **TimerComplete**
4. Drag a **GenericDifferentialDrive** service
5. Connect the **Data** box
    a. Select **SetDrivePower**
    b. Select **Value** for the left and right wheels
6. Drag a **Data** <u>integer</u> with the value <u>2000</u>
7. Place a **Merge** activity (that you will use later) and connect it to the **Data** box
8. Connect the **GenericDifferentialDrive** to this **Merge** box
    a. Select **SetDrivePower – Success**
9. Copy and paste another instance of the **Timer**
10. Connect the **Data** box to it
    a. Select **SetTimer**
    b. Select **Value**

Your diagram should be like this one:

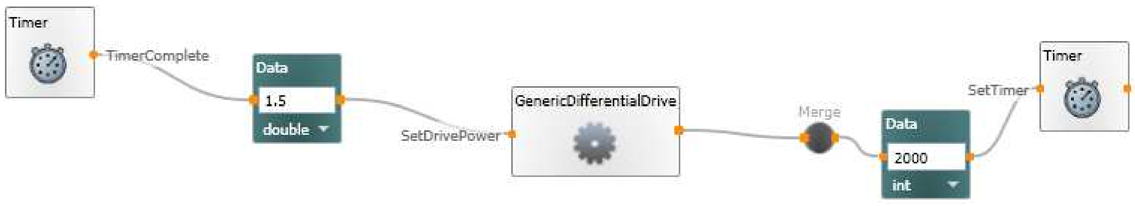


Figure 47. Diagram

Now you will create the interruption diagram:

1. Drag an **iRobotCreateLiteSimulation** service
    a. Set Configuration: **use a manifest**
    b. Select **Import**
    c. Select **testwallsensorsim.Manifest.xml**
2. In the **GenericDifferentialDrive** created before
    a. Set Configuration: **use a manifest**
    b. Select **SimulatedGenericDifferentialDrive in testwallsensorsim.manifest.xml**

You will use a special scenario for the iRobot Create and you are selecting it as a differential drive.

3. Drag a **Merge** activity
4. Connect the notification pin of the **iRobotCreateLiteSimulation** service to the input pin of the **Merge** activity
    a. Select **UpdateBumpsCliffsAndWalls**
5. Create a **Data** integer with the value 1
6. Connect the **Merge** to it
7. Copy and paste an instance of the **iRobotCreateLiteSimulation**
8. Connect the **Data** to it
    a. Select **PlaySong**
    b. Select **Value**

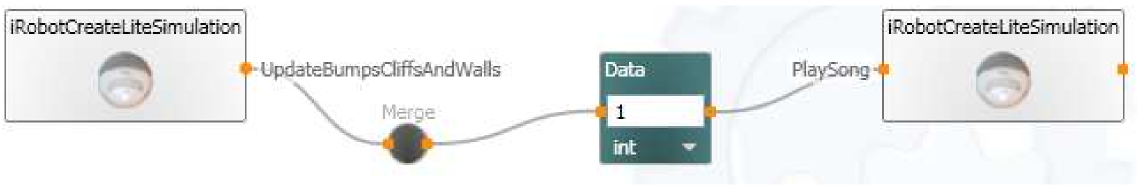

iRobotCreateLiteSimulation
UpdateBumpsCliffsAndWalls
Merge
Data
1
int
PlaySong
iRobotCreateLiteSimulation

Figure 48. Playing song

This is part of the interruption. When a sensor is updated the robot will turn. When the flow is turning the iRobot will play a song (computer ”beeps”).

Let’s configure the robot to turn:
1. Create two **Data** double activities with the values: -0.2 and 0.2 respectively
2. Connect the last **Merge** (Figure 48) to both of them
3. Create a **Join** activity with the values: **Left** and **Right** respectively
4. Connect the **Data** boxes to them
    a. -0.2 **Left**
    b. 0.2 **Right**
5. Copy and paste an instance of the **GenericDifferentialDrive**
6. Connect the **Join** activity to the input pin of the **GenericDifferentialDrive**
    a. Select **SetDrivePower**
    b. Select **Left** for **LeftWheelPower** and do the same for the right wheel
7. Connect this **GenericDifferentialDrive** to the Merge you have created before in the forward diagram (Figure 47)
    a. Select **SetDrivePower - Success**

It will override the timer to make a turning of the same time every forward flow. The resulting final diagram should be like this one:

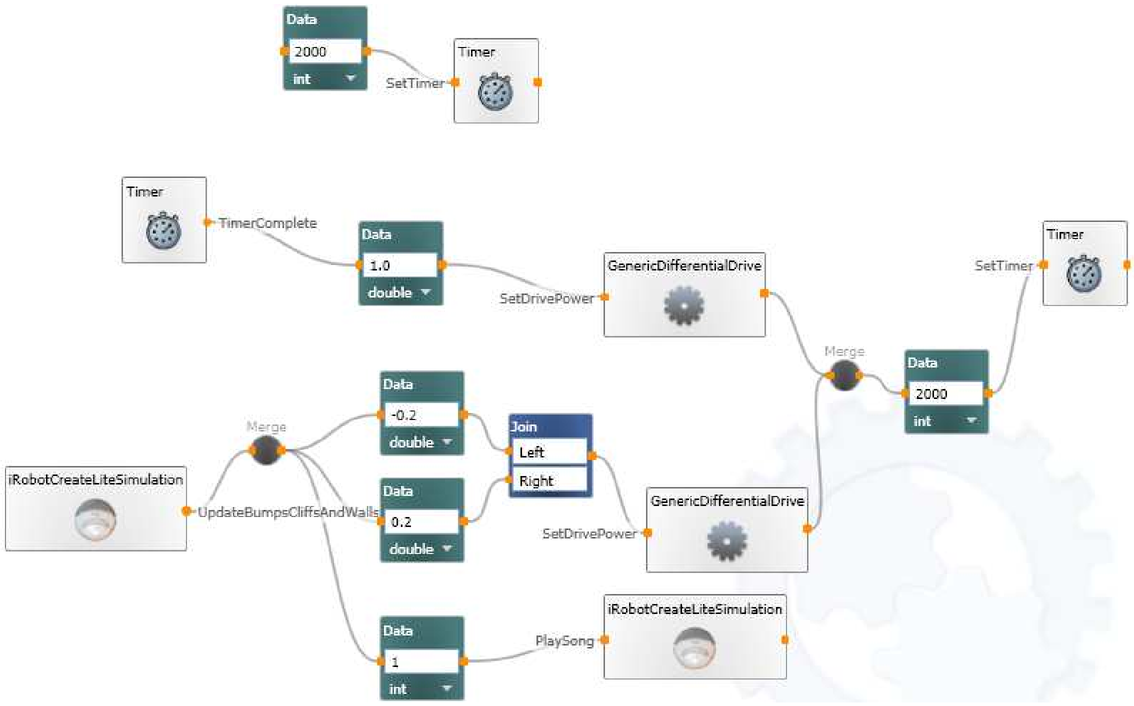


Figure 49. Final diagram

You will see an environment with walls and objects. When the robot crashes with any of them it will turn left and keep moving. Note that crashing with objects doesn't work in 100% of the cases because the robot moves them.

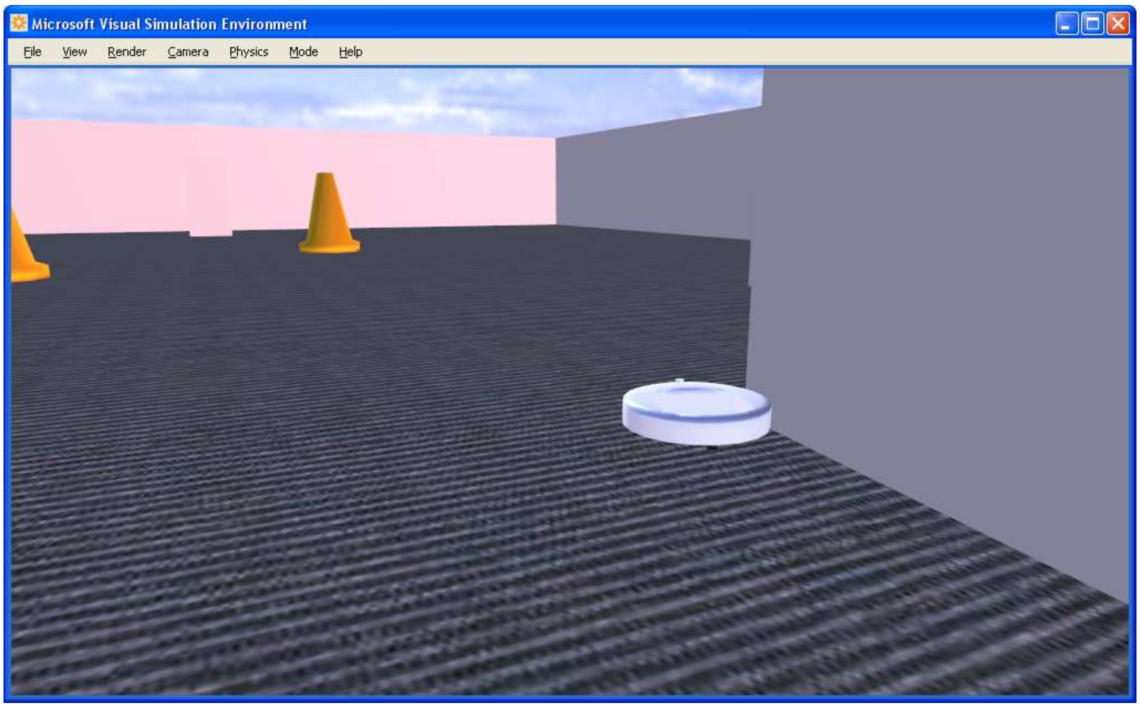


Figure 50. Robot turning after a collision

The robot created will move over the entire scenario.

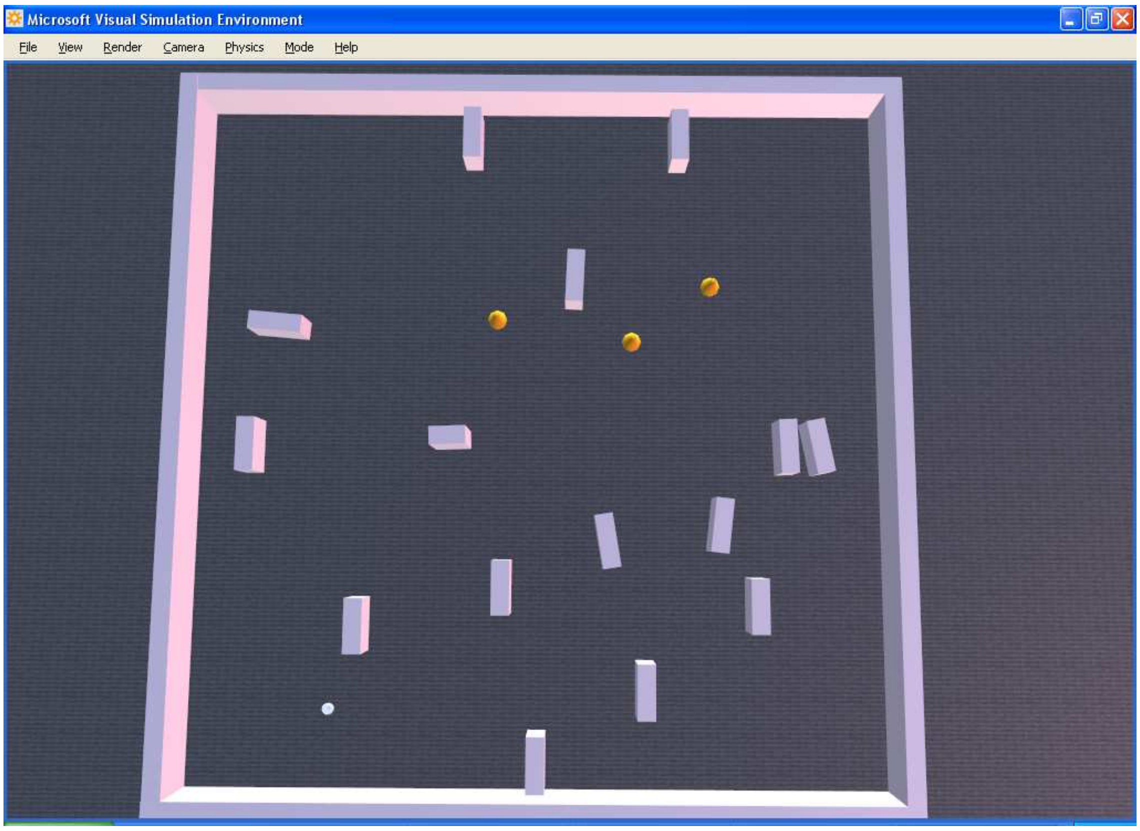


Figure 51. Entire environment