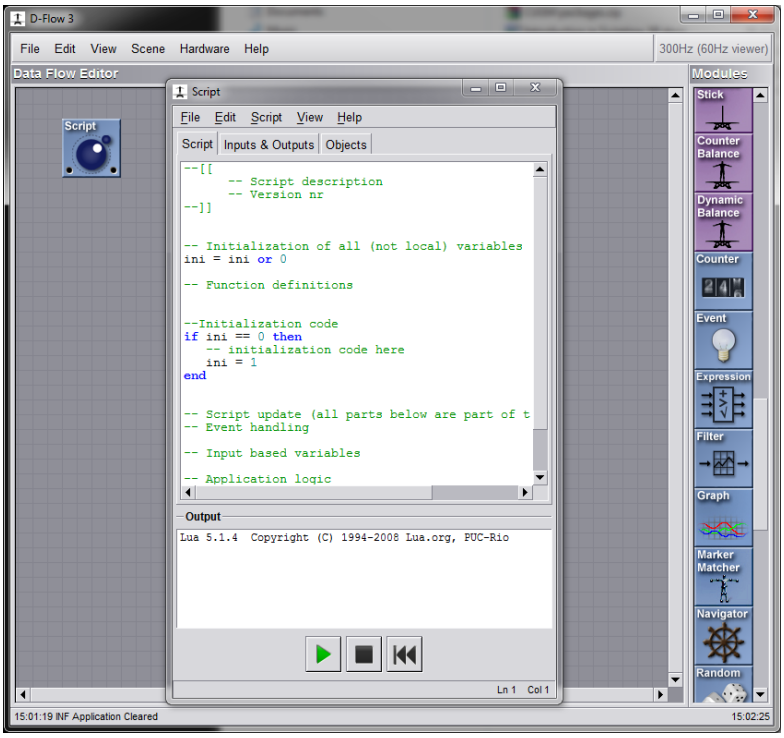
Motek
a DIH brand

Lua scripting in D-Flow

Training syllabus



1 Preface

Scripting in D-Flow, which was requested by many customers, largely extends the possibilities of creating more advance applications.

The ‘Lua scripting in D-Flow training syllabus’ was designed by Motek Medical BV to provide D-Flow users a brief look at some functionalities of the Script module. This syllabus is not designed to learn Lua, but to teach the user how to use the Script module. Some experience in line-based programming is highly recommended.

We hope you enjoy getting to know the Script module in D-Flow.

Regards,

Motek Medical BV

2 Contents

3 The Script module

The Script module is used to write scripts that are part of the D-Flow applications in the scripting language Lua. The Script module is used when complex logic or behavior of scene objects is required, which is hard to realize with the standard modules. A few examples when to use the Script module:

- When a lot of complex mathematical expressions is used for custom motion analysis algorithms
- To control complex behavior of a lot of objects
- For object animation based on events

Script module Icon

A second advantage of scripting is that one Script module can contain the same functionality as an entire group of other modules together. In a lot of cases it is possible to create the functionality with the standard modules, but instead it would be more efficient to create it within a single script. This saves space in the D-Flow editor and keeps the application clean and orderly. Another advantage is that some functionality is only available in the Script module and not in the other modules.

The scripting code is written in the 'Script'-tab. Input and output channels that are defined in the script are created in the 'Input & Outputs'-tab. Objects that are referred to inside the script, need to be added to the module (like is done with the collision module). These objects will appear inside the 'Objects'-tab.

Scripting functionality overview

- Get input values and set output values
- Modify scene object positions, rotation and scaling
- Adjust material settings
- Do collision detection
- Adjust camera and light settings
- React to module actions
- Schedule global events
- Create and delete scene objects

3.1 User interface

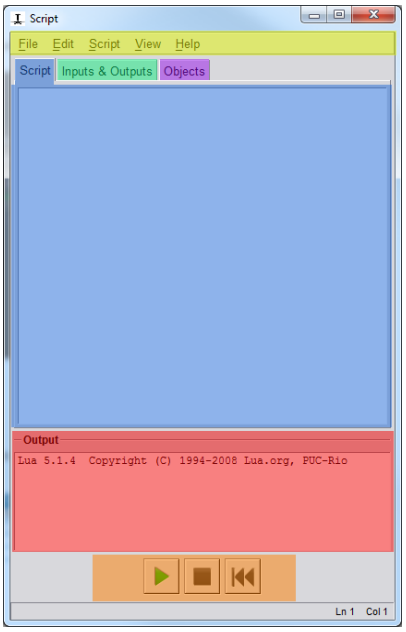
The user interface of the Script module is similar to other D-Flow modules. It contains 3 tabs, an output field, Play control buttons and a menu bar.

3.1.1 Menu bar (Yellow)

The menu bar provides several functionalities, such as Import, Export, Edit, Find & Replace, View and access to the Script function reference.

3.1.2 Script tab (Blue)

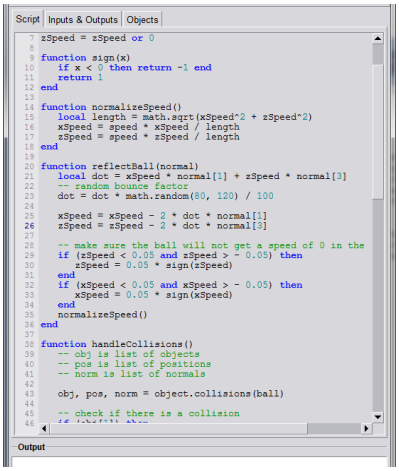
The script code is entered here in the text field. Sidebars will appear in case the script becomes longer or wider than the text field.



The Script module's user interface.

To help the programmer navigate through the code, the Script module uses:

- Syntax highlighting: display of code in different colors according to the category of terms
- Parenthesis matching: is a syntax highlighting feature that highlights matching sets of braces and brackets
- Line numbers can be enabled or disabled under View options
- Clear output: Delete the output printed in the Output field



Syntax highlighting and line numbers

3.1.3 Inputs & Outputs tab (Green)

This section is used to configure the input and output channels of the Script module. The module has six default channels, but these can be deleted or renamed. Additional channels can be added too. The script can refer to these input channels in order to retrieve data from these channels.

It is also possible to create in/outputs using the inputs/outputs library. For more information about the library, please refer to page - 10 - of this manual.

3.1.4 Objects tab (Purple)

This section shows all objects attached to the Script module. Collision detection is enabled on the Objects tab.

3.1.5 Output field (Red)

This section shows the print statements and errors from the script.

3.1.6 Play control section (Orange)

Controls the module actions: Play, Stop and Reset. These are used for quick testing of your script.

3.2 Module actions

The script has several module actions which it reacts to, either by a fixed reaction or a reaction specified in the script itself. The ‘Start’ and ‘Reset’ module action are explained below. The reaction of the ‘Stop’, ‘Stop + Reset’ and the ‘Run Once’ module actions are pretty straightforward. The reactions of the ‘Calibrate’, ‘Action’ and ‘Custom 1-6’ can be specified within the script.

3.2.1 Start

The script starts to run and keeps running until it is stopped. Once the script starts running the entire script is carried out each frame. Variables that aren’t declared locally are shared over frames, i.e. the assigned value stays assigned to that variable.

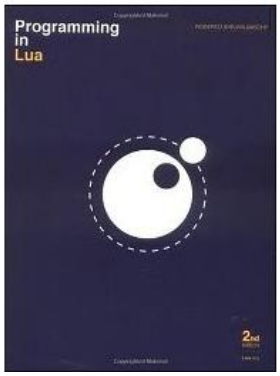
3.2.2 Reset

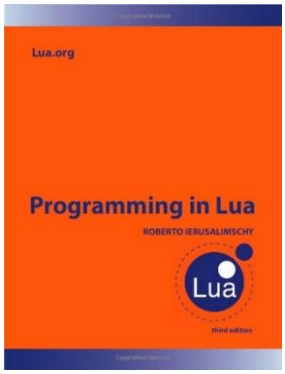
The ‘Reset’ module action brings the application back to the state before the script was carried out. For instance, all created objects are removed.

3.3 D-Flow related Lua basics

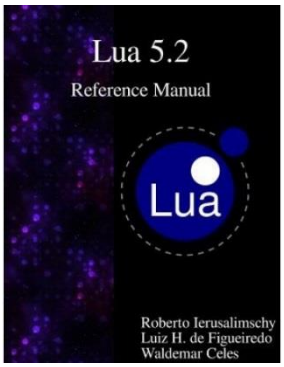
Lua is a powerful, fast, lightweight, embeddable scripting language created by the Pontifical Catholic University of Rio de Janeiro in Brazil. It is credited to be fast in processing and easy to learn.

Although this syllabus is not designed to learn Lua, it will cover some of the basic Lua language. This is needed to learn how to use the Script module in D-Flow. For more information on Lua please refer to www.Lua.org and the book "Programming in Lua" (2nd edition or 3rd edition) and for the reference manual please refer to "Lua 5.2 Reference Manual".





Programming in Lua (2nd and 3rd edition)



Lua 5.2 Reference Manual

3.3.1 Value types

Lua is a dynamically typed language. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type. The Script module supports most of the Lua value types: numbers, strings, Booleans, tables, functions and the value type nil. The types 'userdata' and 'thread' cannot be used directly in the Script module.
'Nil' is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value.
'Boolean' is the type of the values false and true. Both nil and false make a condition false; any other value makes it true.
The value type *'string'* is used to represent text in Lua and is assigned to the value by using double quotes, e.g. "This is a string".
The value type *'number'* will be assigned to variables with any numerical value, e.g. '3', '-8.17' or math.pi. Lua does not make a distinction between integers and floats. Expressions in Lua are very intuitive, as they evaluate to a number.

```
-- a comment
print("Hello, world!")

-- expressions are intuitive
a = 10 * 3 + 1
print(a)

b = 10 * 3 + 1.1
print(b)

print(math.pi)
```

```
Hello, world!
31
31.1
3.1415926535898
```

Print statements, strings and numbers in Lua

The type '*table*' implements associative arrays, that is, arrays that can be indexed not only with numbers, but with any Lua value except nil. Tables are the sole data structuring mechanism in Lua; they can be used to represent ordinary arrays, sequences, symbol tables, sets, records, graphs, trees, etc. The type table can be assigned to a value by using braces {}.

```
-- Assign object settings
objectSettings = {o, {1 , 1.2, -3}, "Cylinder", true}

-- or
pos = {1 , 1.2, -3}
objectSettings = {o, pos , "Cylinder", true}
```

An example of tables in Lua

'*Functions*' are first-class values in Lua. That means that functions can be stored in variables, passed as arguments to other functions, and returned as results.

```
-- Calculates the 3D distance between 2 points
function distance3d(point1, point2)
    -- (table, table -> number)
    local delta = {point1[1] - point2[1], point1[2] -point2[2],
                   point1[3] - point2[3]}

    return math.sqrt(delta[1] ^ 2 + delta[2] ^ 2 + delta[3] ^ 2)
end
```

An example of a function in Lua

3.3.2 Libraries

The Script module supports the use of several Lua function libraries. Next to some standard Lua libraries there are D-Flow specific libraries. All functions from these libraries are described in the script reference manual (Go to D-Flow Menu Bar/ Help/D-Flow 3 Help/Module Reference/Script/ Script function Reference).

The specific D-Flow functions are categorized in three groups, Global functions, input/output functions and DRS functions. With the use of the DRS functions it is possible to alter the properties of objects, cameras, lights, materials, 3DGraphs, haptics and physics. All node functions can be used on all nodes (objects/camera/light) attached to the Script module.

The Script module also supports the use of the following standard Lua libraries: *math*, *string*, *table*, *io* and *os*. Especially the *math* and *table* library are helpful when creating scripts in the D-Flow Script module. The math library comprises a standard set of mathematical functions, such as random, trigonometric and rounding functions. The table library comprises auxiliary functions to manipulate tables as arrays. One of its main roles is to give a reasonable meaning for the size of an array in Lua. It also provides functions to insert and remove elements from lists and to sort the elements of an array.

3.3.3 Control structures

Lua provides a small and conventional set of control structures, with *if* for conditional and *while*, *repeat*, and *for* for iteration. All control structures have an explicit terminator: *end* terminates the if, for and while structures; and until terminates the repeat structure.

The condition expression of a control structure may result in any value. Lua treats as true, all values different from false and nil, e.g. the number 0 is also evaluated to true.

```
for i = 1,8 do
      print(i)
end

local a = 0
while  a < 9 do
      print(a)
      a = a + 1
end

if 0 then
      print("text")
end
```

```
…
6
7
8
1
2
3
4
5
6
7
8
text
```

Examples of 'for', 'while' and 'if' control structures

3.4 D-Flow Script structure

To assist in getting a script structured, standardized and readable a specific structure is recommended. The script structure can be divided in five parts; a header, initialization of global variables, function definitions, initialization code and an update part.

The header contains a description of the script's functionalities and a version number. This will make the script share friendly. After the header, all global used variables are initiated by the standard Lua syntax "variable = variable or initialValue". This will give an overview of the variable names used. Hereafter the functions are defined as they must be declared before they are

used in the rest of the script. If an external functions file is used, it can be required in this part of the script structure.
The actual script is written after the definition of the functions used, starting with the initialization code. As the script runs each frame, an initialization needs to be set up which is only performed on the first run. The last part of the script is defined as the update. This part of the script has the handles which must be performed each D-Flow frame. As this is the most extensive part of the script we recommend splitting it in the following four parts: handling the module actions, getting the values for input based variables, the actual script logic and setting the values to the output channels.

```
--[[       -- Script description
           -- Version nr
--]]
-- Initialization of all (not local) variables
ini = ini or 0

-- Function definitions

-- Initialization code
if ini == 0 then
      -- initialization code here
      ini = 1
end

-- Script update (all parts below are part of the script
update)
-- Handling module actions

-- Input based variables

-- Application logic

-- Set Output
```

The scripting in D-Flow code structure template

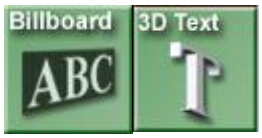
4 Starting tutorials

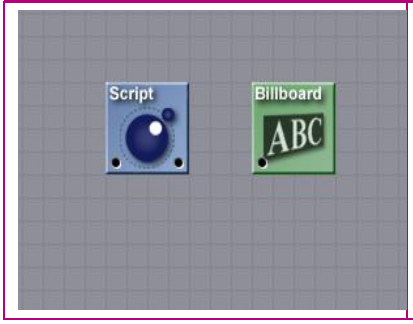
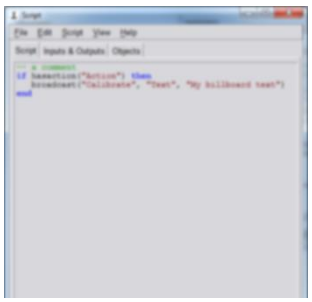
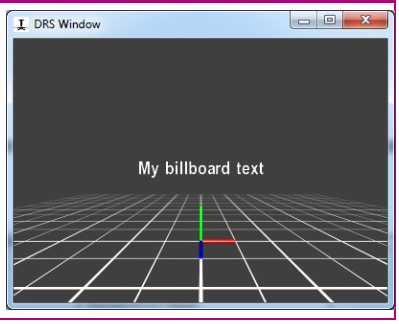
The following tutorials give a brief look into the possibilities of the Script module.

4.1 Using module actions to control a billboard text

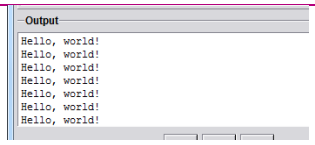
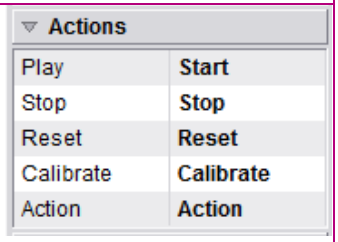
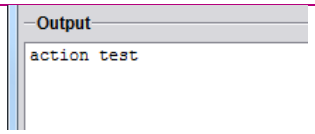
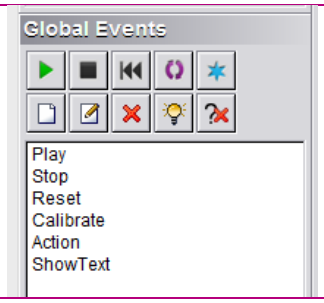
This example describes how to send text to a billboard using a Script module.

This is just a small piece of scripting code, which does not need the suggested structure in paragraph 3.4. However, when it is part of a larger script, element 1 of this tutorial should be positioned at the Script update part, to be more precise, in the part Handling module actions.

Using module actions to control a billboard text		
Time estimation: 20-30 min		
Goal	Getting familiar with the Lua scripting language and learning about module actions / event handling in the Script module.	
Topics covered in this tutorial	• Scripting using module actions • Broadcasting events with metadata • Setting up a billboard for “text from event”	
Modules used in this tutorial:		**Script** This module interacts with the Data flow using Lua scripting language.
		**Billboard or 3DText** This module is able to show text and numerals in the 3D-scenery.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Billboard, 3DText module	

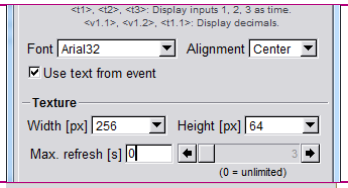
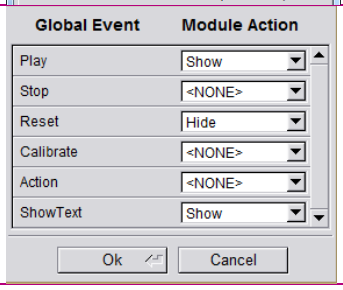
Preview of the result after completing this tutorial		
		

Element 1: Create the script	
Goal	In this element, we are going to create a Lua script which broadcasts an event on a certain module action.
How?	The Script module provides the possibility to create scripts. By line-programming a script, it is possible to send text to a billboard.

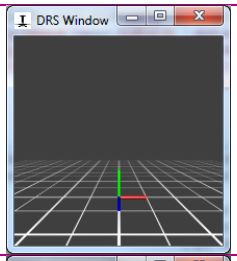
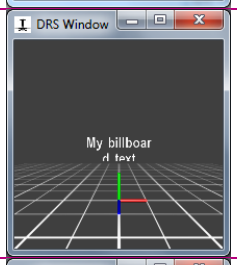
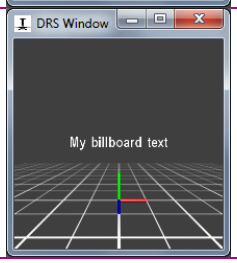
Procedure	Explanation	
Create a Script module and print to the output		
1. Get a Script module (blue) and open the user interface.	The Script module interacts with the Data flow using the Lua script language.	
2. Try to print to the output window. Type: `print("Hello,world!")` and run the script.	Additionally try other statements that print to the output window, such as info(), warning(), error(). **Note:** info/ warnings/ error messages will also appear as a pop-up in the bottom left corner of the screen.	
Create the Lua script		
3. Delete all previously created script.		
4. Create the following 'if-statement': `if hasaction("Action") then print("action test") end`	You do not have to setup the event mapping as the global event "Action" is already mapped to the module action "Action" `hasaction()` returns whether the passed module action was triggered during the D-Flow frame. So: If the script's module action is 'Action' then the text "action test" is printed in the output field.	
5. Try the if statement by running the script once.	This will show nothing, as the module action 'Action' is not triggered during the run.	
6. Test the script by starting it (Play) and clicking the 'Action' button (blue star) on the Runtime console or in the Global Events.	This will print the text 'action test' in the script's output field.	
7. Create a new global event called 'ShowText'.		
8. In the script replace `print("action")` by `broadcast("ShowText", "Text", "My billboard text")`	The function `broadcast (event [, meta data])`, broadcasts the global event, with attached meta data if provided. The meta data is passed by specifying the key, followed by the value. This is what happens:	

	If the script's module action is 'Action', then the global event 'ShowText' is broadcasted. This event is accompanied by the metadata of the type 'Text' with a value of 'My billboard text'. *See additional tutorial 5.1 Sending and using metadata for more information about the use of metadata.*	
9. Try to test the script	You will notice that this does *not* have any visual effect yet, as there is no module set to receive the text on 'ShowText'.	

Element 2: Setting up the billboard	
Goal	In this element, we are going to set up a billboard module to receive text from the Script module
How?	The billboard module provides the possibility to use the text sent as metadata with an event.

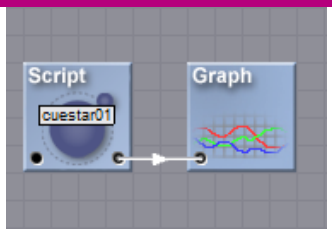
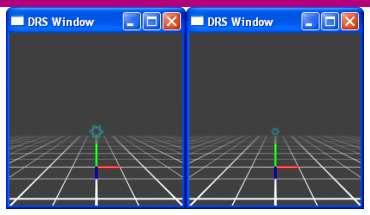
Procedure	**Explanation**	
Create a billboard module and adjust the GUI settings		
1. Drag a Billboard module from the modules onto the D-Flow editor.		
2. Open the Billboard's user interface.		
3. Set the billboard to your preferred settings.	**Tip**: type something in the text field to see how the text appears on the screen.	
4. Check "Use text from event" on the Text tab.	This setting enables the billboard to receive text (meta data) along with an event.	
5. Set up the event mapping: On the global events 'Play' and 'ShowText' the billboard should be set to 'Show'. On the event 'Reset' the Billboard should be set to 'Hide'.		

Element 3: Test the application	
Goal	In this element, we are going to test the application and if needed adjust the billboard settings
How?	Running the script.

Procedure	Explanation	
Test the application		
1. Run the application using the 'Play' button in de 'Global Events' section.	This runs the script. However, no billboard or text is displayed yet, because the 'Action' event is not triggered yet.	
2. Click the 'Action' button.	This shows the text entered in the script on the billboard. **Note**: If the text does not show, go to the 'Text' tab of the billboard and set the refresh rate to '0'.	
3. If needed, adjust the billboard settings (e.g. texture width).		
4. Go to the Script module and change the text `"My billboard text"` in something else.		
5. Run the application (Play) again and click the 'Action' button.	Your new text should appear on the Billboard.	

4.2 Create and use a simple function

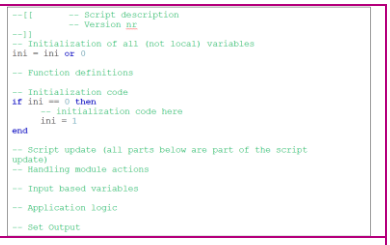
This example describes how you can create a simple path animation function and use it on an object. Try to place the pieces of code in the correct parts of the Script structure. Please refer to page - 11 - for an explanation about the script structure.

Using module actions to control a billboard text		
Time estimation: 20-30 min		
Goal	Getting familiar with the Lua scripting language and learning about functions and controlling an object using the Script module.	
Topics covered in this tutorial	• Scripting using function calls • Using outputs • Controlling an object using the Script module	
Modules used in this tutorial:		**Script** This module interacts with the Data flow using Lua scripting language.
		**Graph** This module is used for graphical data display.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Graph module	
Preview of the result after completing this tutorial		
		

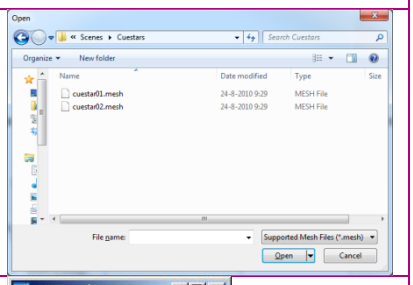
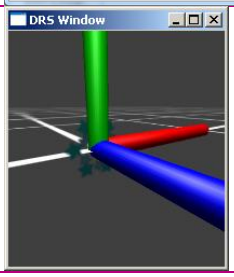
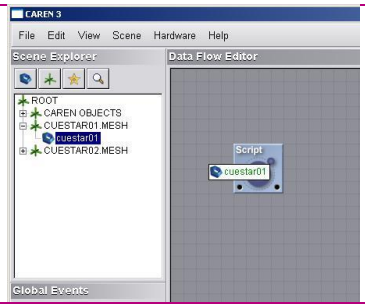
Element 1: Structure your script using comments	
Goal	Create a readable script in the Script module
How?	Use the Script structure template and the additional comments from paragraph 1.4 during the creation of each script code snippet/line

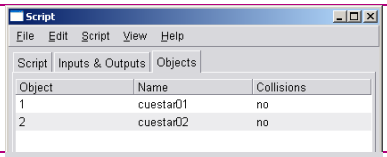
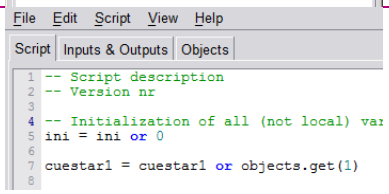
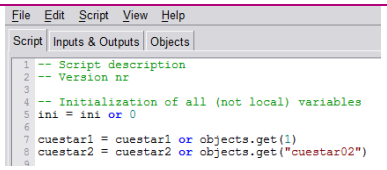
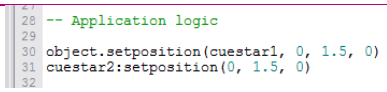
Procedure	**Explanation**	

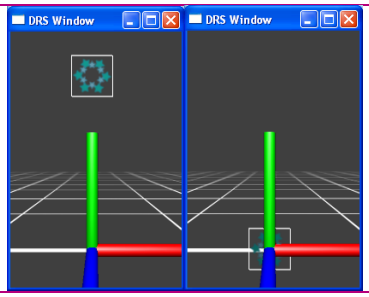
1. First define the header of your script:
 	
`-- Script description`
`-- Version nr`
	The header contains a description of the script's functionalities and a version number.	

2. Divide your script structure in the recommended parts (see 1.4 for more information)	The structure parts; • The header • Initialization of global variables • Function definitions • Initialization code • Update part	
3. Add additional comments explaining your code	e.g. add descriptions and examples to th functions.	

Element 2: Controlling an object using the Script module	
Goal	In this element, we are going to control the position of an object.
How?	The Script module provides the possibility to control the settings of an attached object.

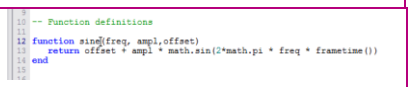
Procedure	**Explanation**	
Add objects to the Script module		
1. Go to 'Add scene' (Button or through scene in the menu).		
2. Select Files of type: ".mesh" and open the file "Cuestar01".	Now the mesh "Cuestar01" is added to the scene explorer.	
3. Perform the same steps for "Cuestar02".	Also "Cuestar02" is added to the scene explorer. Check the position of the cuestars. They are located in the origin of the scene.	
Using objects in the Script module		
4. Get a Script module		
5. Unfold the node in the scene explorer and drag the 2 cuestars, one by one, onto the Script module.		

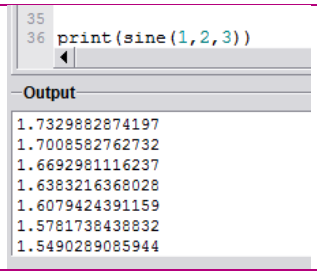
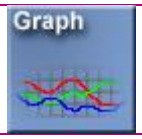
6. Check the objects tab of the Script module	The two objects should be visible on this tab.	
7. On the script tab: Assign a global variable containing the first object using the following script: `cuestar1 = cuestar1 or objects.get(1)`	Here the variable 'cuestar1' is defined. In this case we define it with the index '1'. This refers to the first object in the Objects-list.	
8. Assign another variable with the other object, using the objects name: `cuestar2 = cuestar2 or objects.get("cuestar02")`	Here the variable 'cuestar2' is defined. In this case we define it using the actual name of the object "cuestar02". **Note**: in case of referring to names you must use string notation (double quotes). Also the names are case sensitive. Both the index and the actual name could be used to define your object variable.	
9. Test the script to see if it generates errors by running it.	It is always good to test your scrip regularly to see if the script itself produces errors that stop the script.	
Setting the position of objects in the scripting module		
10. Initialize the position of the object using the following line code: `object.setposition(cuestar1, 0, 1.5, 0)`	Here we set the position of the object 'cuestar1'. We use the command '`object.setposition`'. Then we define the variable 'cuestar1', and then the x, y, and z coordinates. Notice that the object has moved in the DRS window.	
11. Initialize the position of the second object using the following line code: `cuestar2:setposition(0, 1.5, 0)`	Here we set the position of the object 'cuestar2'. Notice that we use another code here, but the actual result is the same. First the variable 'cuestar2', than the command 'setposition' and then the x,y,z, coordinates.	

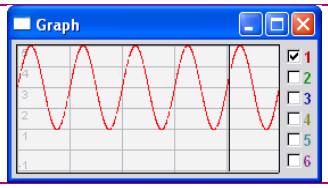
12. Notice that 'Stop' followed by 'Reset', resets the objects to the original state.		

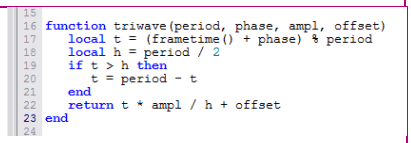
Element 3: Creating a Lua function	
Goal	In this element, we are going to create a Lua function which we are going to use as path animation (scale) on the cuestar objects.
How?	Using a self-created function call.

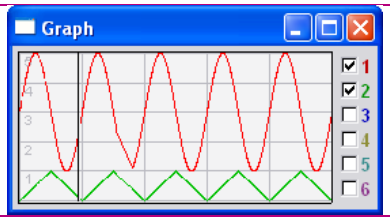
Procedure	**Explanation**	
Create a simple Lua function		

1. Add the following code in the Script module ('Function definitions'):
 	
`function sine(freq, ampl, offset)`
`  return offset + ampl * math.sin (2 * math.pi * freq * frametime())`
`end`
	This will create a sine function with input values for frequency, amplitude and offset.
	
	Inside a function you can fill variables or you can return a value.
		

2. Test your function by printing the function: `print(sine(1, 2, 3))`	Notice that the output field is filled with numbers.	
3. Delete the line: `print(sine(1, 2, 3))` again from the script.	Actually we do not need this to be printed in the output field, this was just a test.	
Test your function		
4. Get a Graph module.	This module is used for graphical data display.	
5. Connect the output of the Script module to the input of the Graph module.		

6. Delete every wire in the connection editor, except the first two.		
7. Use the created function as output for the Script module: `outputs.set(1, sine(1, 2, 3))`	`outputs.set(1, sine(1, 2, 3))` tells the module to output the sine wave on output channel '1' with a frequency of '1', an amplitude of '2', and an offset of '3'.	
8. Click play in the Global events and check the Graph module		
More complex functions		

9. Create the following line code in the Script module:
 	
`function triwave(period, phase, ampl, offset)`
`  local t = (frametime() + phase) % period`
`  local h = period / 2`
`  if t > h then`
`  t = period - t`
`  end`
`  return t * ampl / h + offset`
`end`
	Inside loop, if statement, functions, etc. you can use local variables.
	
	This code creates a triwave. In the end, you are interested in the returned value of 't * ampl / h + offset'
	Variables 't' and 'h' are local variables and are defined in the script lines.
		

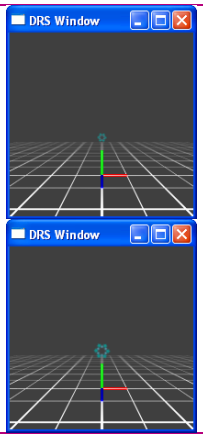
10. Try to write a code that will output the triwave on channel 2, using the following parameters: Period =1 Phase = 1 Ampl = 1 Offset = 0		
11. Test the function using the Graph module.	The result of your code written in step 9 is visualized with the green line.	

Element 4: Using the created function as path animation on the objects	
Goal	In this element, we are going to use the created function as path animation to set the scaling of the cuestars.
How?	Setting the scale of the cuestars with the triwave function.

Procedure	**Explanation**

Set the scale of the cuestars dynamically		

1. Create the following line code in the Script module:
 	
`s = triwave(1,0,1,1)`
`object.setscaling(cuestar1, s, s, 1)`
`object.setscaling(cuestar2, s, s, 1)`
		`-- Application logic`
		
		`object.setposition(cuestar1, 0, 1.5, 0)`
		`cuestar2:setposition(0, 1.5, 0)`
		
		`s = triwave(1, 0, 1, 1)`
		`object.setscaling(cuestar1, s, s, 1)`
		`object.setscaling(cuestar2, s, s, 1)`

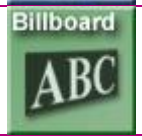
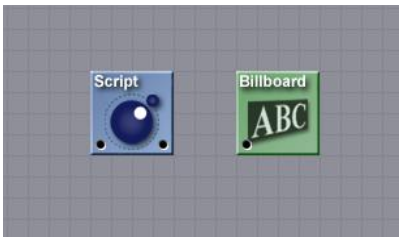
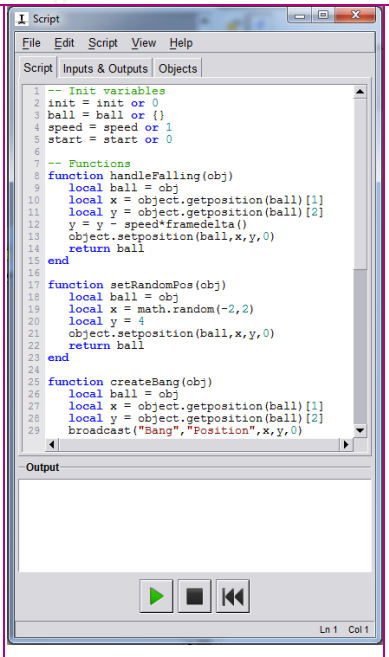
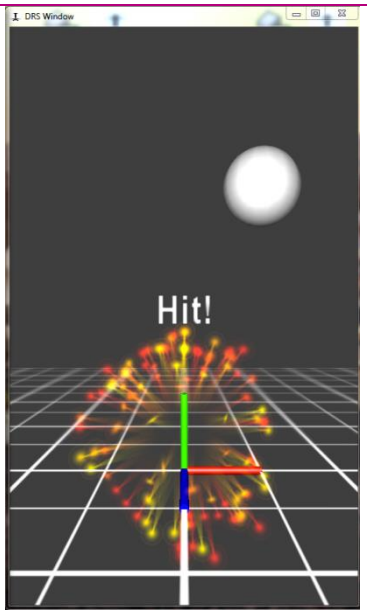
2. Run the script and check the results.	You will notice that the stars start to grow and shrink. This is because the cuestar objects are scaled on x, y, z, with respectively 's', 's' and '1'. 's' in this case is a triwave with a period of 1, phase of 0, amplitude of 1, and an offset of 1.	
Use simple math to rotate the cues		

3. Create the following line code in the Script module:
 	
`r = 100 * framedelta()`
`cuestar1:rotate(0, 0, r)`
`cuestar2:rotate(0, 0, -r)`		

4. Run the script and check the results.	You will notice that the stars start to rotate. This is because the cuestar objects are rotated on x, y, z with respectively '0', '0', 'r'. 'r' is in this case defined as '100 * framedelta()' framedelta() is a global function created for d-flow, which returns the delta time per frame.	

5 Additional tutorials

5.1 Sending and using metadata

Using module actions to control a billboard text		
Time estimation: 20-30 min		
Goal	This tutorial explains the use of metadata with functions action() and broadcast().	
Modules used in this tutorial:		**Script** This module interacts with the Data flow using Lua scripting language
		**Billboard** This module is able to show text and numerals in the 3D-scenery.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Billboard module	
Preview of the result after completing this tutorial		
		

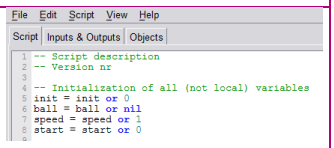
The Script module can take over the transfer of information about positions and texts with broadcasted events. This information can subsequently be interpreted by other modules like the Billboard or the Particle Module. This way, for example, only one billboard is needed to display several different texts, or a Particle (Particle module) can be sent to a certain place based on an event. Both examples are explained in this tutorial.

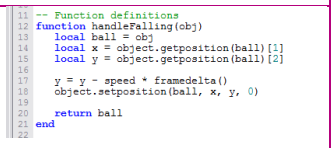
First some basics: broadcast ("Name") is a function that broadcasts a global event "Name". It is possible to send additional data (metadata) with this event broadcast. This metadata is sent by specifying a key type (e.g. "Text") followed by (a) value(s).
Example: broadcast ("Action","Position", 1.0, 2.0, 3.0, "Text", "Hit!"). This broadcasts the global event "Action" with positional metadata (x = 1.0, y = 2.0, z = 3.0) and textual metadata ("Hit!").

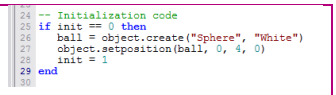
Note: this tutorial also uses hasaction(), in case this function is not familiar please refer to the tutorial "5.2 Different actions" for more information.

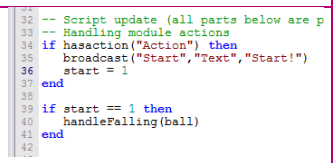
Element 1: Creating the application	
Goal	This tutorial explains the use of metadata with functions action() and broadcast().
How?	Create an application where a ball drops from a certain height and create a billboard to display the start and the moment the ball hits the ground.

Procedure	**Explanation**	
1. Go to global events and create two custom events called; "Start" and "Hit"		
2. Get a Script module & open the Script module's user interface		

3. Under `'Initialization of all variables'`, enter the following code:
 	
`init = init or 0`
`ball = ball or nil`
`speed = speed or 1`
`start = start or 0`
	`'Initialization of all variables'` is the place where all global variables are created before they are used or filled. So this is the beginning of your script. This is not mandatory in Lua, but it will give an overview of the used variable names.	

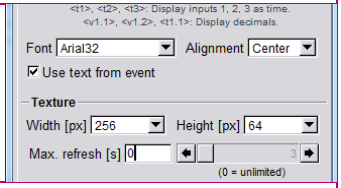
4. Under `'Function definitions'`, enter the following code:
 	
`function handleFalling(obj)`
`  local ball = obj`
`  local x = object.getposition(ball)[1]`
`  local y = object.getposition(ball)[2]`
 	
`  y = y - speed * framedelta()`
`  object.setposition(ball, x, y, 0)`
 	
`  return ball`
`end`
	This function will handle the dropping of the ball that we are going to create next. `'Function definitions'` is the place for all the custom function definitions.	

5. Under 'Initialization code', enter the following code:
`if init == 0 then`
`ball = object.create("Sphere", "White")`
`object.setposition(ball, 0, 4, 0)`
`init = 1`
`end`
	The 'Initialization code' of the script, in which the scene is set up and the script is made ready to start, should be written below the functions, because often functions are used during initialization. In this case we create a white sphere with position y = 4.	

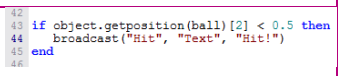
6. Finally under 'Script Update', enter the following code:
`if hasaction("Action") then`
`broadcast("Start","Text", "Start!")`
`start = 1`
`end`
`if start == 1 then`
`handleFalling(ball)`
`end`
	'Script Update' is the part that is used to: handle application logic, update the scene and update output values.	

7. Start the application (Play)	Nothing should happen so far.	
8. Click the Action button in the global events window.	Notice the ball slowly dropping.	
The first part of the update checks whether the event "Action" has occurred during the current frame. If so, the event "Start" is broadcast with the metadata "Start!" (in the form of text). The value for the variable "start" is set to 1. The second part of the update checks whether the variable "start" is 1. When the value is '1', the ball will fall.		

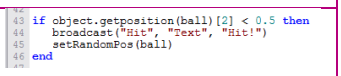
Element 2: Creating billboard feedback	
Goal	To display the billboard texts for the start when it is at y = 0.5. This can be done with and without the use of metadata.
How?	1. Without the use of metadata, two billboard modules would be required. Input the text in the billboards, set their event-mapping to "Start" and "End" and make the script broadcast these events based on the height of the ball. 2. With the use metadata, only one billboard module is needed. The script will broadcast the events based on the height of the ball. The events are accompanied by metadata in the form of text that is displayed by the receiving billboard.

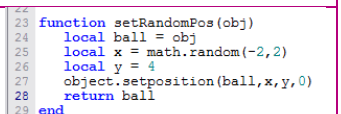
Create 1 billboard module with the use of metadata		
1. Get a Billboard module	Open the Billboard user interface and set the billboard to your preferred settings.	
2. Check 'Use text from event' in the 'Text' tab.	This setting enables the billboard to receive text along with an event.	
3. Set up the event mapping: On the global events both 'Start' and 'Hit' the billboard should 'Show'.	.	

4. Go to the appearance tab and set Auto-hide [s] to 1 second and close the Billboard module.		
5. Start the entire application.	Nothing happens.	
6. Click the 'Action' button in the global events window.	Notice the ball drops and the text "Start!" appears.	

7. Open the Script module and enter the following in the `'Update'` part of the script:
 	
`if object.getposition(ball)[2] < 0.5 then`
`  broadcast("Hit","Text","Hit!")`
`end`
	This tells the script to broadcast the event "Hit" when the ball is lower than y = 0.5. The metadata "Hit!" will be passed along, as key type text.	

8. Reset the application and press 'Play".		
9. Click the Action button in the global events window.	Notice first the text "Start!" is shown and when the ball is below 0.5 the text "Hit!" is shown.	
Explode the ball and make it reappear at a random position at the top of the screen		
10. Go to Script module.	There are two options to destroy the ball. 1. Hide the ball and create a new ball at the original position. 2. Give the already existing ball a new, random, position. Option 2 is explained in this tutorial.	

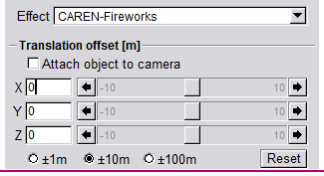
11. Go to the `'Update'` part and change the if-statement for checking the height of the ball into:
 	
`if object.getposition(ball)[2] < 0.5 then broadcast("Hit", "Text", "Hit!")`
`  setRandomPos(ball)`
`end`
		

12. Add the following code to the `'Function definitions'` part:
 	
`function setRandomPos(obj)`
`  local ball = obj`
`  local x = math.random(-2,2)`
`  local y = 4`
`  object.setposition(ball,x,y,0)`
`  return ball`
`end`
	This sets the ball at a random x-position between -2 and 2 at a whole number {-2, -1, 0, 1, 2}. For more information about Lua's random functionality, please refer to the tutorial 5.4 ' Random with Lua'.	

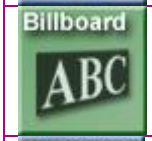
13. Reset and play the application	Notice the ball is sent back to the top as soon as it hits the ground	
Explosion of the ball: use the partial module		

14. Go to the Script module and change the `'Script update'`, add the following to the body of the if-statement to check the height of the ball: `createBang(ball)`	Make sure this function is inserted above the setRandomPos() function because this will influence the position of the fireworks .	

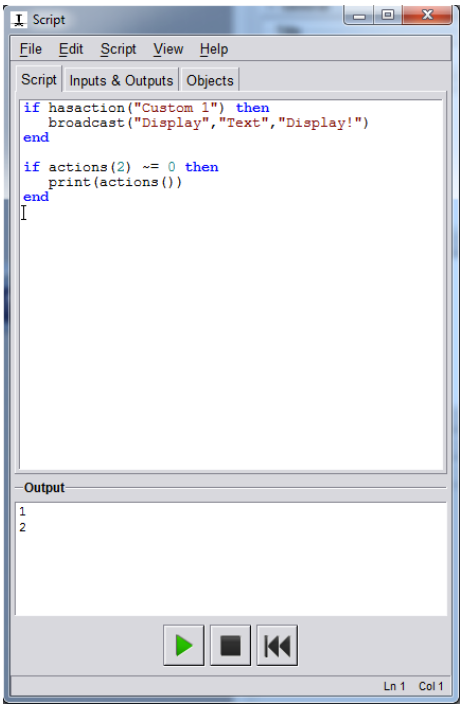
15. Go to `'Function definitions'` part and add the following code:
 	
`function createBang(obj)`
`  local ball = obj`
`  local x = object.getposition(ball)[1]`
`  local y = object.getposition(ball)[2]`
`  broadcast("Bang", "Position", x,y,0)`
`end`
	This will broadcast the global event 'Bang' with the positional metadata x, y and z (0)	`function createBang(obj)`
		`  local ball = obj`
		`  local x = object.getposition(ball)[1]`
		`  local y = object.getposition(ball)[2]`
		`  broadcast("Bang","Position",x,y,0)`
		`end`

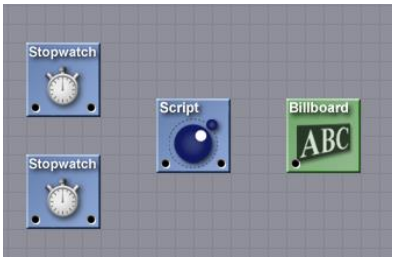
16. Go to global events and click "Create new event"	Enter "Bang" and Click OK.	
17. Create a Particle module	Go to the event mapping of the particle module and set the global events 'Play' to 'None' and 'Bang' to 'Play'.	
18. Open the Particle module and select the effect 'Fireworks'. Close the Particle module.		
19. Reset and play the application.		.
20. Click Action	Notice the Ball starts to drop, creates fireworks when it reaches the bottom, and is moved back to the top.	
Tip! There are also other ways to use metadata, for instance when using the Pointer module.		

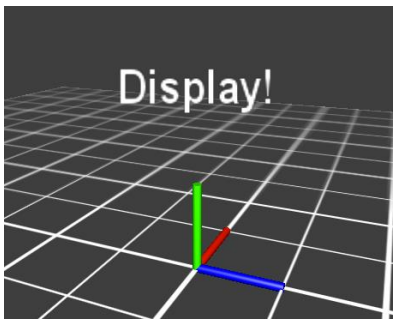
5.2 Different actions

Using different module actions in the Script module		
Time estimation: 20-30 min		
Goal	In this tutorial, the use of the different functions as action(), actions() and hasaction() is explained.	
Modules used in this tutorial:		**Script** This module interacts with the data flow using Lua scripting language
		**Billboard** This module is able to show text and numerals in the 3D-scenery.
		**Stopwatch** This module displays a time counter on a stopwatch and is able to output the time as a value.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Stopwatch module	

Preview of the result after completing this tutorial







D-Flow contains several global events by default like; 'Calibrate', 'Reset' or 'Action'. It is also possible to create new custom global events by clicking the 'Create new event'-button in the 'Global Events' section. These global events are broadcast by the following modules: Collision, Controller, Event, Pointer3D, Random, Stopwatch, or by the Script module itself. By using event mapping these global events can be linked to the module actions of the Script module.

Events and actions can also carry so called metadata, which can be positional or textual information that comes with the text and can be used by other modules like the Billboard. For more information about metadata please refer to the tutorial "5.1 Sending and using metadata".

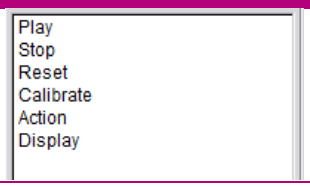
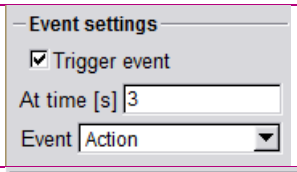
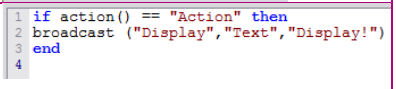
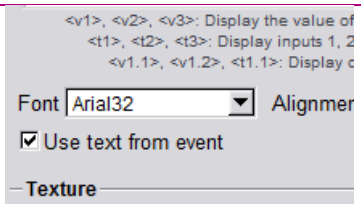
The Script module uses three different functions that handle actions:

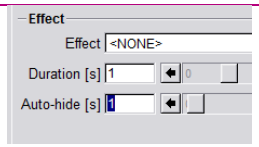
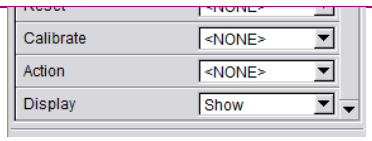
action(i)
Returns the name of the action in the action list at position "i". If no argument is given, the function will return the first action from the action list.

actions()
Returns the number of actions in the action list, for the current frame.

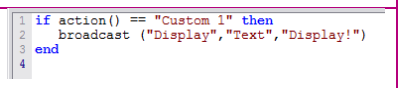
hasaction("action name")
Returns whether the Script module action "action name" has occurred in this frame (true or false).

Element 1:	**The use of different actions**
Goal	To learn how use a global event to execute a section of a script, based on the function action().
How?	By displaying a text on a billboard when a certain module action in the Script module is triggered.

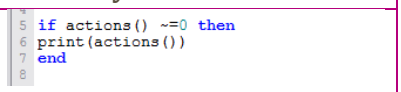
Procedure	**Explanation**	
1. Go to the 'Global Events' section and create a new event called 'Display'.		
2. Create a Stopwatch module.		
3. Open the Stopwatch module. Enable 'Trigger event', at time '3' seconds, and Event 'Action'.	Once this stopwatch is started, it will broadcast the 'Action' event after 3 seconds.	
4. Create a Script module and add the following code: `if action() == "Action" then broadcast ("Display","Text","Display!") end`	This line states that when the first module action in the action list is 'Action' then the global event 'Display' will be triggered, accompanied by the text: "Display!".	
5. Create a Billboard module. Check the 'Use text from event'-box.		

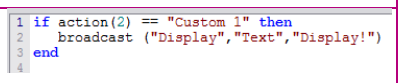
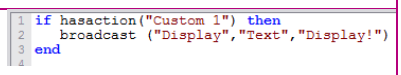
6. On the appearance tab, set 'Auto-hide' after 1 second.		
7. Go to event mapping of the Billboard module. Set the global event 'Display' to the module action 'Show' and click 'OK'.		
8. Go to the 'Global Events' section and press the 'Play' button.	This will start the Stopwatch and the Script. After 3 seconds the Stopwatch will broadcast 'Action', thereby triggering the Script to broadcast the global event 'Display' and this is received by the Billboard, which displays the text 'Display'.	

Element 2: Double events	
Goal	To learn how to handle multiple module actions with the script module, based on the actions() and has action() functions.
How?	By triggering the Script module with two module actions that are send simultaneously.

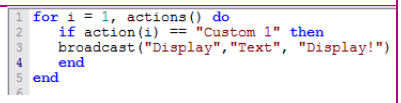
Procedure	**Explanation**	
1. Go to global events and create a new event called: 'Extra'.		
2. Copy the existing Stopwatch module and open it. Set trigger event to 'Extra' and close the Stopwatch.		
3. Go to Event Mapping of the Script module. Set the global event 'Extra' to: 'Custom 1' and click 'OK'.		
4. Open the Script module and change `if action() == "Action"` into: `if action() == "Custom 1"`		
5. Go to the 'Global Events' section press 'Reset' and 'Play'.	Notice that after 3 seconds no Billboard appears. This is because both events "Action" and "Extra" are triggered simultaneously by the two Stopwatch modules. This means that both module actions 'Action' and 'Custom 1' are in the action-list. Since action()	

	only checks the first entry in the action-list, which in this case is 'Action' and not 'Custom 1', the if statement is false and therefore the global event 'Display' is not broadcast.	
Using actions(): to check whether both events are triggered correctly		

6. Open the Script module and add the following:
 	
`if actions() ~=0 then`
`print(actions())`
`end`
	This script returns the number of actions in the action list, but only when the number of actions in the action-list is not equal to zero. (~= is *'unequal to'*)	

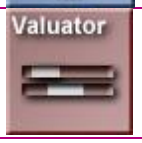
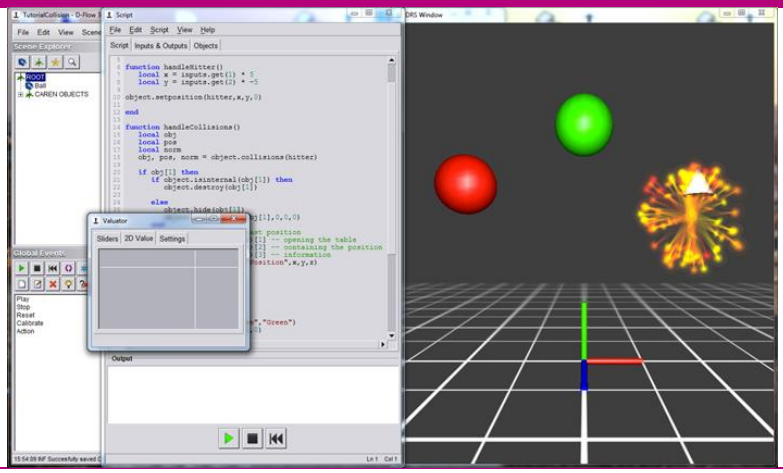
7. Go to global events and press 'Reset' and 'Play'.	Notice that the 'Output' window in the Script module first displays a '1', which represents the 'Play' event and after 3 seconds it displays a '2', which are the events for both Stopwatch modules. This checks whether the events were broadcasted correctly.	
Using action(i)		
8. Open the Script module and change `if action() == "Custom 1"` into: `if action(2) == "Custom 1"`		
9. Go to the 'Global Events' section and press 'Reset' and 'Play'.	Notice the text is displayed now because action(2) checks the second entry of the action-list and not the first one. This is however a temporary fix, because it means that each time the Script module receives multiple events this number "i" needs to be correct, and with larger applications it is not always the same.	
Using hasaction("Name")		
10. Open the Script module and change `if action(2) == "Custom 1"` into: `if hasaction("Custom 1")`		
11. Go to the 'Global Events' section and press 'Reset' and 'Play'.	The text is displayed. hasaction() checks whether the event is somewhere in the action-list, indifferent of its location. This means that for	

	larger applications hasaction() often is a safer choice, since it does not depend on the amount of actions that are received.	
Using a loop for actions		

12. To create a loop for all actions change:
 	
`if hasaction("Custom 1") then broadcast ("Display","Text","Display!") end`
into:
 	
`for i = 1, actions() do if action(i) == "Custom 1" then broadcast("Display","Text", "Display!") end end`
	This for-loop checks for each entry in the action-list (from 1 to actions()), whether that entry is "Custom 1". If not, it goes to the next entry. If true, then it will broadcast.	

13. Go to the 'Global Events' and press 'Reset' and 'Play'.		

5.3 Collision Detection

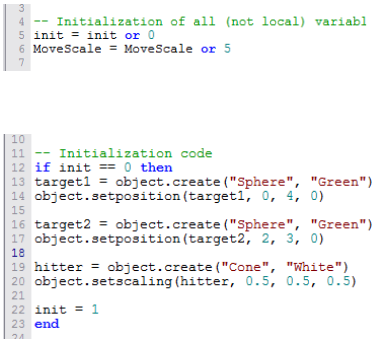
Use collision detection of objects		
Time estimation: 20-30 min		
Goal	In this tutorial is explained how to use collision detection of objects created inside and outside the Script module	
Modules used in this tutorial:		**Script** This module interacts with the Data flow using Lua scripting language
		**Valuator** This module has a number of sliders with a predetermined range which can be changed during runtime. This module is helpful to simulate output in the testing phase of application creation.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Valuator module	
Prieview of the result after completing this tutorial		
		

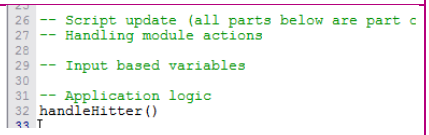
The Script module is able to detect collisions between objects and obtain data from it. Collisions are both possible between internal objects as well as between internal and external objects. Internal objects are objects created within the Script module. External objects are objects add to the scene explorer and dragged onto the Script module.
Additionally the use of the data that is produced by each collision is explained. With each collision three types of data are generated: an objects-list (the objects involved in the collision), a position-list (the positions of the colliding objects), and a normal-list (a list of all collision normals*).

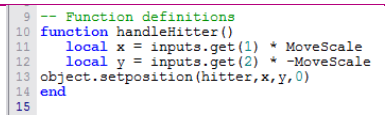
*Normal: a perpendicular line to the impact of both objects that can be used to i.e. determine the angle of impact and angle of reflection.

Element 1: Creating the application	
Goal	In this tutorial is explained how to use collision detection of objects created inside and outside the Script module
How?	This tutorial shows how to handle collisions of both internal and external objects.

Procedure	Explanation	

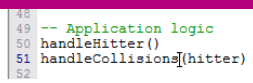
1. Get a Script module and enter the following into the script:
 	
`-- Initialization of all (not local) variables`
`init = init or 0`
`MoveScale = MoveScale or 5`
 	
`-- Initialization code`
`if init == 0 then`
`target1 = object.create("Sphere", "Green")`
`object.setposition(target1, 0, 4, 0)`
 	
`target2 = object.create("Sphere", "Green")`
`object.setposition(target2, 2, 3, 0)`
 	
`hitter = object.create("Cone", "White")`
`object.setscaling(hitter, 0.5, 0.5, 0.5)`
 	
`init = 1`
`end`
	The green headers display a basic setup in which Scripts can be created to ensure a better overview when creating larger scripts.
	
	The "initialization" part creates the objects when the script runs for the first time.
		

2. Create a Valuator module.		
3. Connect the Valuator to the Script.	Parameter 1 of the Valuator module is automatically linked to input 1 of the Script module.	
4. Open the Script module and add the following code to 'Update script' part: `handleHitter()`		

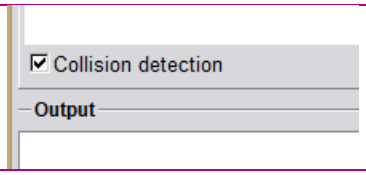
5. Add the following code to the `'Function definitions'` part:
 	
`function handleHitter()`
`  local x = inputs.get(1) * MoveScale`
`  local y = inputs.get(2) * -MoveScale`
`object.setposition(hitter,x,y,0)`
`end`
	The data from the valuator is connected to the white cone. For more information about the use of inputs, please refer to the 'Script function reference' (Script module > Help > Function Reference).	

6. Go to 'Global Events' section and press 'Play'.		
7. Open the Valuator module and go to the 2D Value tab.	Notice that you can move the cone	

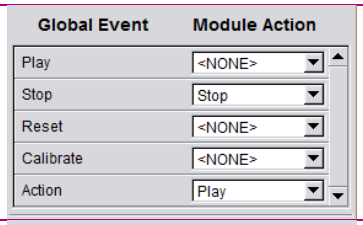
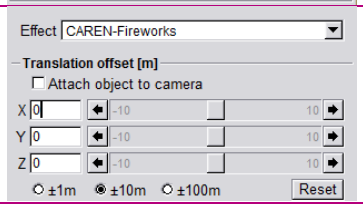
Element 2: Detect internal collisions	
Goal	To detect a collision between two objects and destroy the target.
How?	Destroy a target is only possible for internal objects, i.e. objects created within the Script module.

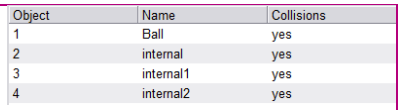
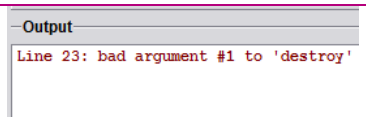
Procedure	**Explanation**	
1. Add the following code to "Script update" part of the script: `handleCollisions(hitter)`		

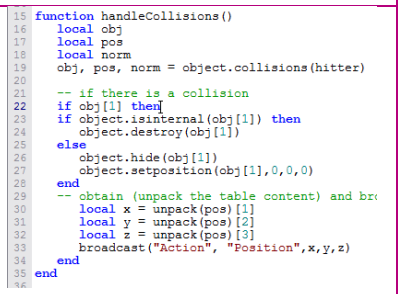
2. Add the following code to the `'Function definitions'` part:
 	
`function handleCollisions()`
`local obj`
`local pos`
`local norm`
`obj, pos, norm = object.collisions(hitter)`
`-- if there is a collision`
`if obj[1] then`
`object.destroy(obj[1])`
`end`
`end`
	The data (objects list, position list and normal list) obtained at the moment of a collision is stored in the variables 'obj', 'pos', and 'norm'.
	
	The if-statement looks whether obj[1] exists, i.e. is not nil (if it is not nil, there has been a collision). If yes, then the first object in the collision-list (the target), is destroyed.	

3. Go to the Objects-tab of the Script module and check the 'collision detection'-box.		
4. Go to 'Global Events' section and press 'Play'.		
5. Go to the Valuator module. Move the white cone over the green balls and notice the green balls disappear upon collision.		
Using data from the collision		
6. In the "function handle Collision()change:	Fireworks should start when the object disappears.	

`--if there is a collision`
`if obj[1] then`
`object.destroy(obj[1])`
 	
`-- obtain (unpack the table content) and broadcast position`
`local x = unpack(pos)[1]`
`local y = unpack(pos)[2]`
`local z = unpack(pos)[3]`
`broadcast("Action", "Position",x,y,z)`
`end`
	Therefore the positional information of the destroyed object needs to be obtained and send that to a Particle module, in order to show the fireworks at the right location.	

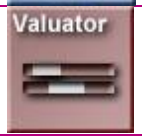
The "pos" is a table containing positional data of all collisions and there can be more at the same time. In this case there is only one, so unpacking "pos" gives a table with three values, of which the first one is set to 'x'. The information is broadcasted as metadata with the global event "Action". For more information on metadata go to the tutorial 3.1 "Scripting: Sending and using metadata".		
7. Create a Particle module.		
8. Open Event mapping and set the global events 'Play' to 'None' and 'Action' to 'Play'.		
9. Open the Particle module and set the effect to 'Firework'.		
10. Go to the 'Global Events' section and press 'Play'.	Move the white cone and notice the fireworks when the balls are hit.	
Detecting collisions with external objects		
11. Go to Scene > Add object > Name: Ball > Sphere > OK.		
12. Drag the Ball from the Scene Explorer onto the Script module.	For collision detection with external objects, you must manually add these external objects to the Script module. Some object functions, like 'object.destroy()' are not applicable on external objects, because the object is not handled in this script.	
13. Reset and Play the Script module.		

14. Go to the Objects-tab of the Script module and notice it looks like this.	The 'Name' column displays the name of the external objects. Internal objects are not named, for they are handled inside the Script.	
15. Now add the Ball into the script so it can be handled. Add to the 'Initialization' part: `ball = objects.get(1)`	objects.get() gets the first object from the list.	
16. Reset and Play the application.		
17. Open the Valuator and try to hit the objects.	Notice that the application stops after hitting the red ball. This is because the script tries to destroy the object involved in the collision, but has no authority to do so. To solve this problem, change the if-statement.	

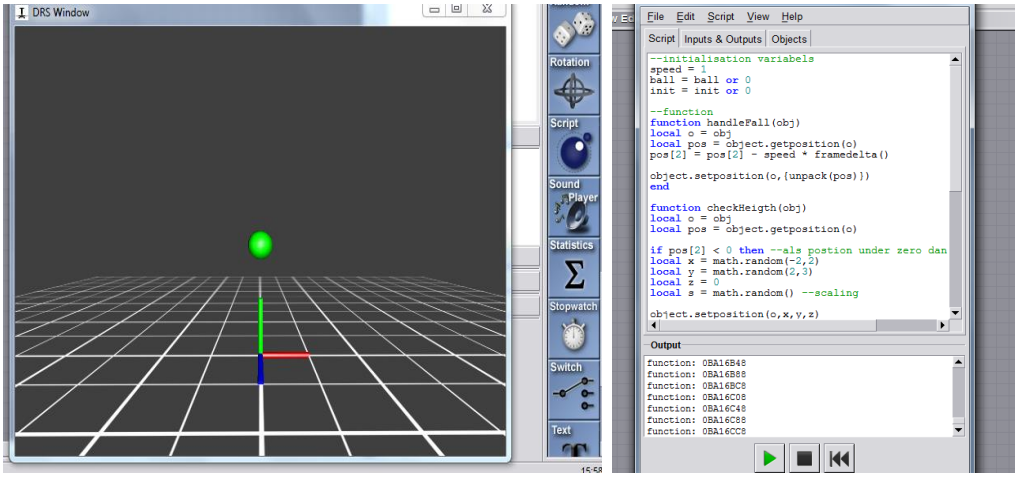
18. Change the if-statement in the function `handleCollisions():`
 	
`-- if there is a collision`
`if obj[1] then`
`  if object.isinternal(obj[1]) then`
`  object.destroy(obj[1])`
`  else`
`  object.hide(obj[1])`
`  object.setposition(obj[1],0,0,0)`
`  end`
`end`
	The function object.isinternal checks whether the colliding object is internal. If not, the object is hidden and send towards the origin (0,0,0). However, this does not remove the object, so when the white cone is now moved towards the origin, collisions can still be made although the object is hidden.
	This method of hiding external scene objects might be useful when in loading and initializing scenes.
		

Assignment:
Can you think of a solution to avoid collisions with the hidden ball object?
Another if statement in the 'handleCollisions()' function maybe?

5.4 Random with Lua

Using random functions in Lua		
Time estimation: 20-30 min		
Goal	This tutorial shows examples of how to use random() functions in Lua.	
Modules used in this tutorial:		**Script** This module interacts with the data flow using Lua scripting language.
		**Valuator** This module has a number of sliders with a predetermined range which can be changed during runtime. This module is helpful to simulate output in the testing phase of application creation.
References:	• D-Flow 3 Help/Module Reference/Script/ Script function reference • Programming in Lua • D-Flow 3 Help/ module references/Valuator module	

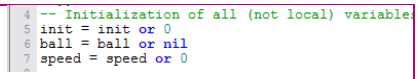
Preview of the result after completing this tutorial

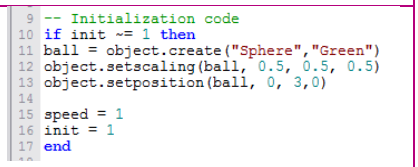


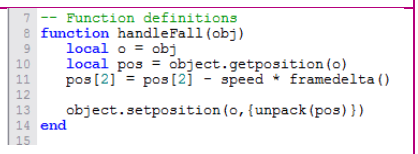
Lua's own function library contains the function math.random(n,m). This function returns a random value between 'n' and 'm'. When no values are inputted and only math.random() is given, the function returns random numbers between 0 and 1, with +10 decimals. When only one number is inputted, e.g. math.random(6), then the function returns numbers between 0 and 6, without any decimals. The same goes for when two numbers are inputted, e.g. math.random(80,90), then the function returns numbers between 80 and 90, without any decimals.

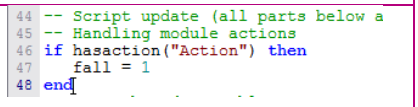
Element 1: Using random functions in Lua	
Goal	This tutorial shows examples of how to use random() functions in Lua.
How?	To use the random function in general with `math.random` and how it can be used to set the position of objects.

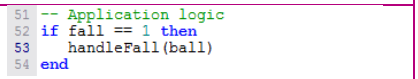
Procedure	**Explanation**

1. Create a Script module. Under 'Initialization of all variables', enter the following code:
 	
`init = init or 0`
`ball = ball or nil`
`speed = speed or 1`
		

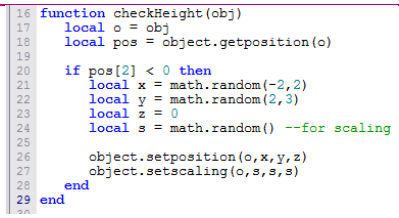
2. Under 'Initialization code' enter the following code:
 	
`if init ~= 1 then`
`ball = object.create("Sphere","Green")`
`object.setscaling(ball, 0.5, 0.5, 0.5)`
`object.setposition(ball, 0, 3,0)`
`speed = 1`
`init = 1`
`end`
	A spherical object is created with scaling '0.5' and y at '3'.	

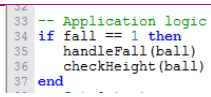
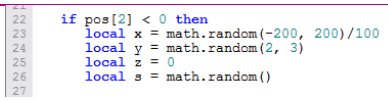
3. Under 'Function definitions', enter the following code:
 	
`function handleFall(obj)`
`local o = obj`
`local pos = object.getposition(o)`
`pos[2] = pos[2] - speed * framedelta()`
 	
`object.setposition(o,{unpack(pos)})`
`end`
	This function will handle the falling of the object.
	`object.getposition(o)` returns a table with three values {x, y, z}.
	`pos[2]` calls the second value of the table, in other words, the y-position.
		

4. Under 'Script update' enter the following code:
 	
`if hasaction("Action") then`
`fall = 1`
`end`
		

5. Under 'Script update' enter the following code:
 	
`if fall == 1 then`
`handleFall(ball)`
`end`
		

6. Go to the 'Global Events' section and press 'Reset' and 'Play'. Now press 'Action'.	Notice that the ball starts to drop, and keeps going.	
7. Stop and Reset the application.		
8. In the script, add the following function:	When the ball gets below y = 0 it is returned to the top	

`function checkHeight(obj)`
`  local o = obj`
`  local pos = object.getposition(o)`
 	
`  if pos[2] < 0 then`
`  local x = math.random(-2,2)`
`  local y = math.random(2,3)`
`  local z = 0`
`  local s = math.random()`
 	
`  object.setposition(o,x,y,z)`
`  object.setscaling(o,s,s,s)`
`  end`
`end`
	with a random position and scaling.
	The new 'x' will be one of the following numbers: {-2, -1, 0, 1, 2}.
	The new 'y' will be either 2 or 3.
	The new scaling will be any number between 0 and 1, with 10 decimals.
		

9. Add the following to the `if fall == 1` statement: `checkHeight(ball)`	This call is needed to actually execute the 'checkHeight()' function.	
10. Change `local x = math.random(-2,2)` in to: `local x = math.random(-200, 200) /100`	The x position of the green ball is desired to be between -2 and 2, with 2 decimals. In this random function, the x-value will return random numbers between -200 and 200. Afterwards they are divided by 100. This means that a random number, like 165, will become 1.65.	
11. Go to the 'Global Events' section and press 'Play' and 'Action'.	Notice that the ball starts to drop, but restarts at a new location after it reaches the grid. The ball also changes size every time it drops.	

6 D-Flow Script function reference

For the most up-to-date function reference please refer to the Script function reference section of the D-Flow reference manual. This Script function reference is updated till D-Flow version 3.16.0. This document describes Lua functions provided by the Script module.
For the standard Lua functions, consult the Lua reference manual (www.Lua.org/manual/5.2).

7 Appendix: Lua Cheat sheet

Lua Core Language

Reserved words

and break do else elseif end false for function if in local nil not or repeat return then true until while

Other reserved strings

+ - * / % ^
 # == ~= <= >= <
> = ( ) { } [ ] ; :
, ...

Identifiers

Any string of letters, digits and underscores not starting with a digit and not a reserved word. Identifiers starting with underscore and uppercase letter are reserved.

Comments

-- Comment to end of line.
--[[...]] Multi-line comment (commonly --[[to --]])
#! At start of first line for Linux executable.

Strings and escape sequences

' ' " " [[ ]] [=[ ]=]
string delimiters; **[[ ]]** can be multi-line, escape sequences ignored. If **[=[ ]=]** number of =’s must balance.
\a - bell **\b** - backspace **\f** - form feed
\n - newline **\r** - return **\t** - tab
\v - vert. tab **** - backslash **\"** - double quote
\' - single quote **\[** - square bracket **\]** - square bracket
\ddd (character represented decimal number).

Types

Type belongs to the value, NOT the variable:

boolean nil and false count as false, all other true including 0 and null string. Use type(x) to discover type of x.
number 64 bit IEEE floating point
string Can include zero, internally hashed.
table Index by numbers, strings
function Can return multiple values
thread A cooperative coroutine.
userdata C pointer to a C object. Can be assigned a metatable to allow use like a table or function
nil A special value meaning “nothing”.

Operators in precedence order

^ (right-associative, math lib required)
not # (length) –(unary negative)(unary positive illegal)
***** **/** **%**
+ **–**
.. (string concatenation, right-associative)
< **>** **<=** **>=** **~=** **==**
and (stops on false or nil, returns last evaluated value)
or (stops on true (not false or nil), returns last evaluated value)

The basic library

require (module)
Tries to load a module (.Lua or .dll) from the application folder or scripts folder.

Information and conversion

type (x)
Returns type of x as string e.g. "nil", "string", “number”.
tostring (x)
Converts x to a string, using table's metatable's __tostring if available.
tonumber (x [, b])
Converts string x representing a number in base b [2..36, default: 10] to a number, or nil if invalid; for base 10 accepts full format (e.g. "1.5e6").
unpack (t)
Returns t [1]..t [n] as separate values, where n = #t.

Iterators

ipairs (t)
Returns an iterator getting index, value pairs of array t in numeric order.
pairs (t)
Returns an iterator getting key, value pairs of table t in no particular order.
next (t [, index])
Returns next index-value pair (nil when finished) from index (default nil, i.e. beginning) of table t.

The Math Library

The Table Library

Basic operations

math.abs (x) Returns the absolute value of x.

math.fmod (x, y) Returns the remainder of x / y as a rounded down integer, for y ~= 0.

math.floor (x) Returns x rounded down to integer.

math.ceil (x) Returns x rounded up to the nearest integer.

math.min(args) Returns minimum value from args

math.max(args) Returns maximum value from args.

math.huge Returns largest represented number

math.modf (x) Returns integer AND fractional parts of x

Exponential and logarithmic

math.sqrt (x) Returns square root of x, for x >= 0.

math.pow (x, y) Returns x raised to the power of y, i.e. x^y; if x < 0, y must be integer.

math.exp (x) Returns e to the power of x, i.e. e^x.

math.log (x) Returns natural logarithm of x, for x >= 0.

math.log10 (x) Returns base-10 log of x, for x >= 0.

math.frexp (x) If $x = m2^e$, returns m (0, 0.5-1) and Integer e

math.ldexp (x, y) Returns $x2^y$ with y an integer.

Trigonometrical

math.deg (a) Converts angle a from radians to degrees.

math.rad (a) Converts angle a from degrees to radians.

math.pi Constant containing the value of Pi.

math.sin (a) Sine of angle a in radians.

math.cos (a) Cosine of angle a in radians.

math.tan (a) Tangent of angle a in radians.

math.asin (x) Arc sine of x in radians, for x in [-1, 1].

math.acos (x) Arc cosine of x in radians, for x in [-1, 1].

math.atan (x) Arc tangent of x in radians.

Pseudo-random numbers

math.random ([n [, m]) Pseudo-random number in range [0, 1], [1, n] or [n, m].

math.randomseed (n) Sets a seed n for random sequence.

Tables as arrays (lists)

table.insert (table, [i,] v)
Inserts v at numerical index i [default: after the end] in table, increments table size.

table.remove (table [, i])
Removes element at numerical index i [default: last element] from table, decrements table size, returns removed element.

table.maxn (table)
Returns largest positive numeric index of table. Slow.

table.sort (table [, cf])
Sorts (in-place) elements from table[1] to table[#t], using compare function cf (e1, e2) [default: '<']. May swap equals.

table.concat (table [, string [, i [, j]]])
Returns a single string made by concatenating table elements table[i] to table[j] (default: i =1, j = table length)separated by string (default = nil). Returns empty string if no given elements or i > j

Iterating on table contents

Use the **pairs** or **ipairs** iterators in a for-loop. Example:

for k, v **in pairs**(table) **do print** (k, v) **end**

will print the **key** (k) and **value** (v) of the entire table's content.

The String Library

Basic operations

String indices start from 1. Negative indices from end of string so -1 is last element of string. String element values 0-255.

string.len (string)
Returns length of string, including embedded zeroes.

string.sub (string, i [, j])
Returns substring of string from position i to j [default: -1 which is to end].

string.rep (string, n)
Returns a string of n concatenated copies of string.

string.upper (string)
Returns a copy of string converted to uppercase.

string.lower (string)
Returns a copy of string converted to lowercase.

string.reverse (string)
Returns a string that is the reverse of string.