

1.1 SCRIPTING IN TRUESPACE.....	4
Script Commands and Objects.....	4
Scripting Possibilities.....	4
Getting Started with Scripting.....	5
Figure 1: Generating XML documentation.....	5
Introduction to the Script Editor.....	6
Figure 2: Accessing the Library Browser.....	6
Figure 3: 1D Script Editor.....	7
Figure 4: Script Editor toolbar.....	8
Adding Editing and Removing Attributes.....	8
Figure 5: Adding an attribute to our jScript command object.....	9
Figure 6: Editing an attribute.....	10
Figure 7: Reviewing our changes so far: Script Editor.....	11
Figure 8: Notice Link Editor.....	14
Figure 9: Editing the object's interface.....	15
Figure 10: Adding and editing elements.....	15
Figure 11: Editing elements.....	16
Figure 12: Editing a script object.....	17
Figure 13: Setting up a scenario.....	18
Video Tutorials: DevGuide.....	19
DevGuide1a.....	19
DevGuide1b.....	19
DevGuide1c.....	19
Tutorial: Modifying a Script.....	19
Figure 14: The ConeObj.....	19
Figure 15: The Workspace window shows the results.....	20
Figure 16: Inside the ConeObj.....	20
Figure 17: ConeMesh's attributes.....	21
Figure 18: The ConeMesh script in the Script Methods view.....	22
Figure 19: Changing just one value.....	25
Figure 20: Now our cone has become a top.....	25
1.2 SCRIPT COMMANDS.....	25
Script Commands in trueSpace.....	26
Creating a Script Command Object.....	26
Figure 21: Drag and drop jScript command.....	26
Figure 22: A JScript Command object.....	27
Figure 23: A JScript Command object with Exp aspect.....	27
Executing a Script Command.....	27
Figure 24: Setting up a JScript Command.....	27
Writing Script Commands.....	27
Figure 25: Calling a function.....	28
Forming complex activities from commands.....	29
One Shot activity.....	29
Figure 26: OneShot activity.....	29
Timer based loop.....	29

Figure 27: Timer Event sends constant stream.....	30
Infinity loop.....	30
Figure 28: Connecting commands.....	31
Figure 29: Activity-Base library.....	31
Tutorial: Writing a direction-control Script Command.....	32
Figure 30: Rename the jScript Command.....	32
Figure 31: The Add New Attribute dial.....	32
Figure 32: Adding code and editing the interface.....	33
Figure 33: Type toolbar.....	34
Figure 34: Random Direction connected to Compass.....	35
Tutorial: Creating Advanced Script Commands.....	35
Figure 35: Use Add Attribute button.....	36
Figure 36: The completed activity loop.....	37
Figure 37: Planet orbits in real-time.....	38
Tutorial: Script Commands in More Complex Objects.....	38
Figure 38: The Terrain System object panel.....	39
Figure 39: The terrain as shown in the Workspace view.....	40
Figure 40: Inside the Terrain System object.....	40
1.3 SCRIPT OBJECTS.....	43
Script Objects in trueSpace.....	43
Creating a Script Object.....	44
Figure 41: A JScript Script Object in the as it appears in the Link Editor.....	44
Inside the Script Object.....	44
Writing Script Object Scripts.....	44
OnComputeOutputs(params).....	45
Advanced Handlers.....	45
OnSetValue(params).....	45
OnDefaultValue(params).....	46
OnCreate(params).....	47
Other advanced handlers.....	48
OnInvalidate.....	48
OnCustomEvent.....	48
OnPostLoad.....	48
OnSharedSpace_NodeStatusChanged.....	48
OnSharedSpace_SpaceStatusChanged.....	48
Tutorial: Creating a Simple Math Object.....	48
Figure 42: The math object.....	49
Figure 43: The math object script generated.....	50
Figure 44: The final math object in the Link Editor.....	50
Tutorial: Examining a Simple Script Object.....	50
Figure 45: The output of ConeObj.....	51
Figure 46: ConeObj as seen in the Link Editor.....	51
Figure 47: The interior of ConeObj.....	51
Figure 48: Creating vertices and faces.....	54
Advanced implementation notes.....	54

1.4 NEW IN TRUESPACE 7.6.....	54
Set of array data objects.....	54
OnGetValues handler.....	55
Script caching.....	55
Persistent variables support.....	56
Global variables support.....	56
Script function sets.....	57
Nested script command calls.....	59
Figure 49: Mines game inner workings.....	59
Figure 50: Setting up the Reveal Helper.....	60
Other notable new and fixed.....	60

trueSpace Developer Guide

Scripting

1.1 Scripting in trueSpace

The trueSpace Script Editor provides a robust environment for creating procedural scripts to accomplish nearly any task. Scripts are interpreted, i.e. each command is executed by trueSpace as it is encountered, and can be edited while the program is running.

Through scripts, you can create, access, and interact with almost any trueSpace object from meshes to shaders. You can choose from several common scripting languages to create your script. trueSpace currently supports VBScript and Jscript, but new languages that are compatible with the Windows Script engine can easily be added.

There are two main methods of accessing the scripting capabilities of trueSpace. These are: Script Commands and Script Objects. Both are covered in this chapter.

Script Commands and Objects

Script Commands and Script Objects are very similar. You start by inserting either an Object or Command into the trueSpace Link Editor, create Input and Output attributes, and then write the code to act on these attributes. As noted previously you can create your script in languages including VBScript and JScript.

There are also important differences between Script Commands and Script Objects. You might think of Script Commands as ‘verbs’ or action items. They directly interact with other objects in your scene, requesting modifications or perhaps creating new objects, and they can act as direct participants in control-flow, within the Link Editor. Script Objects can be thought of as ‘nouns’. They are stand-alone objects that manipulate their own data and state and respond to requests from other objects in the Link Editor and the system itself.

Scripting Possibilities

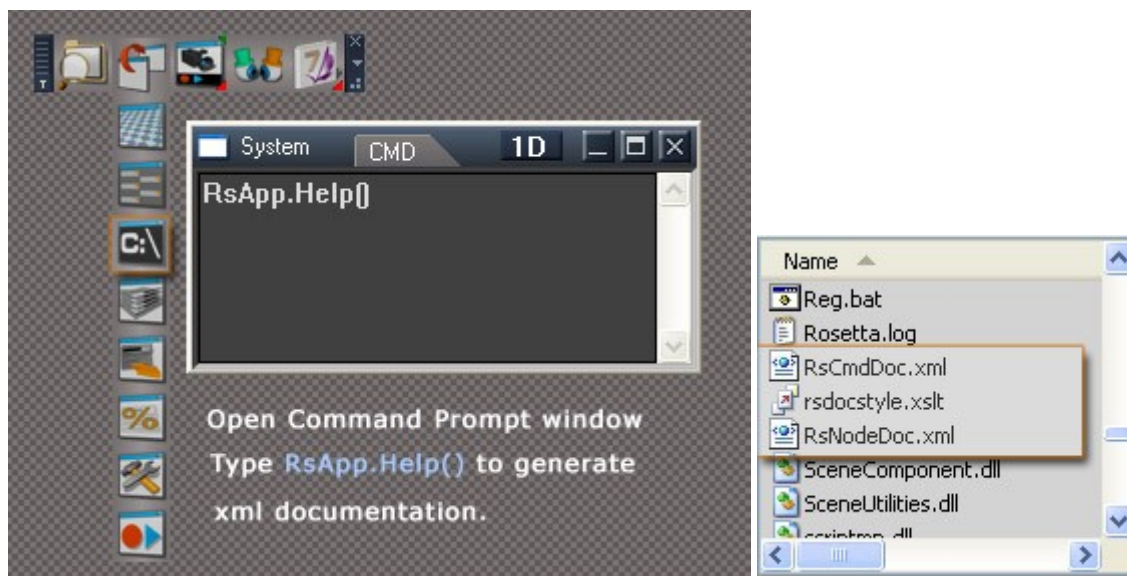
Scripts can help you accomplish a wide variety of tasks. You can write a script to create your own primitive objects (a geosphere, for instance) or to modify an existing object (perhaps processing the objects vertices algorithmically to change or morph its shape). You can use scripts to create behaviors for your objects (for example, causing a door to open when a certain condition is met; someone approaches the door) or to define how objects will interact (say, controlling a group of spheres to create a molecular simulation.)

In fact, there are so many possibilities that we can't even begin to name them all here. Think of it as a really large chest of tools and parts and start to imagine the things that you can build!

Getting Started with Scripting

In the following sections we will examine several scripting examples that you can use and modify to create your own scripts. Also, be sure to examine the many sample scripts that can be found in the trueSpace libraries. The best way to learn scripting is to explore and modify existing scripts.

To learn about native trueSpace objects, their attributes, and the methods that operate on them, trueSpace has a mechanism in place, that produces documentation, from the source code itself. By opening a Command Window and entering a command, the documentation is created in your default directory for trueSpace in the /ts folder. The image below consists of 2 screen-shots, left image showing open Command Prompt window and the command itself, with the right hand image showing partial windows explorer in the /ts folder. Highlighted area shows files created by the process. You can double click on the .xml files to open them in a web browser.



Generating XML documentation

Finally, to learn what commands and facilities, such as libraries and built-in objects, are available in your chosen scripting language please visit the sites listed below.

JScript, VBScript - <http://www.microsoft.com/scripting>

Also, if you have not already done so, you may want to read through Chapter 2.12, for an overview of the Script Editor interface and a brief look at how scripts work in trueSpace before moving on to the more in-depth discussion in the remainder of this chapter.

Introduction to the Script Editor

Although there are a number of excellent scripts for you to investigate and explore, we should first go over what is entailed in actual creation of a script, along with descriptions and explanations of some important items.

Drag and drop: by default the special library for scripts is not loaded into trueSpace. This is not a major problem as it is pretty easy to load. The image below shows the smaller toolbar with Library Browser icon highlighted. Click on the icon to start the Library Browser (it floats). Scroll down the list of libraries until you see the System grouping as shown below. Double-click the icon or right-click the icon for menu and select Open Library. Either will load the library to the top of the Library-stack area. You can see the System – Scripts library loaded at top of stack on the top right-hand side of the image. Notice that there are individual library items shown in the library. For purpose of this illustration, a jScript command object was dragged and dropped into the Link Editor.

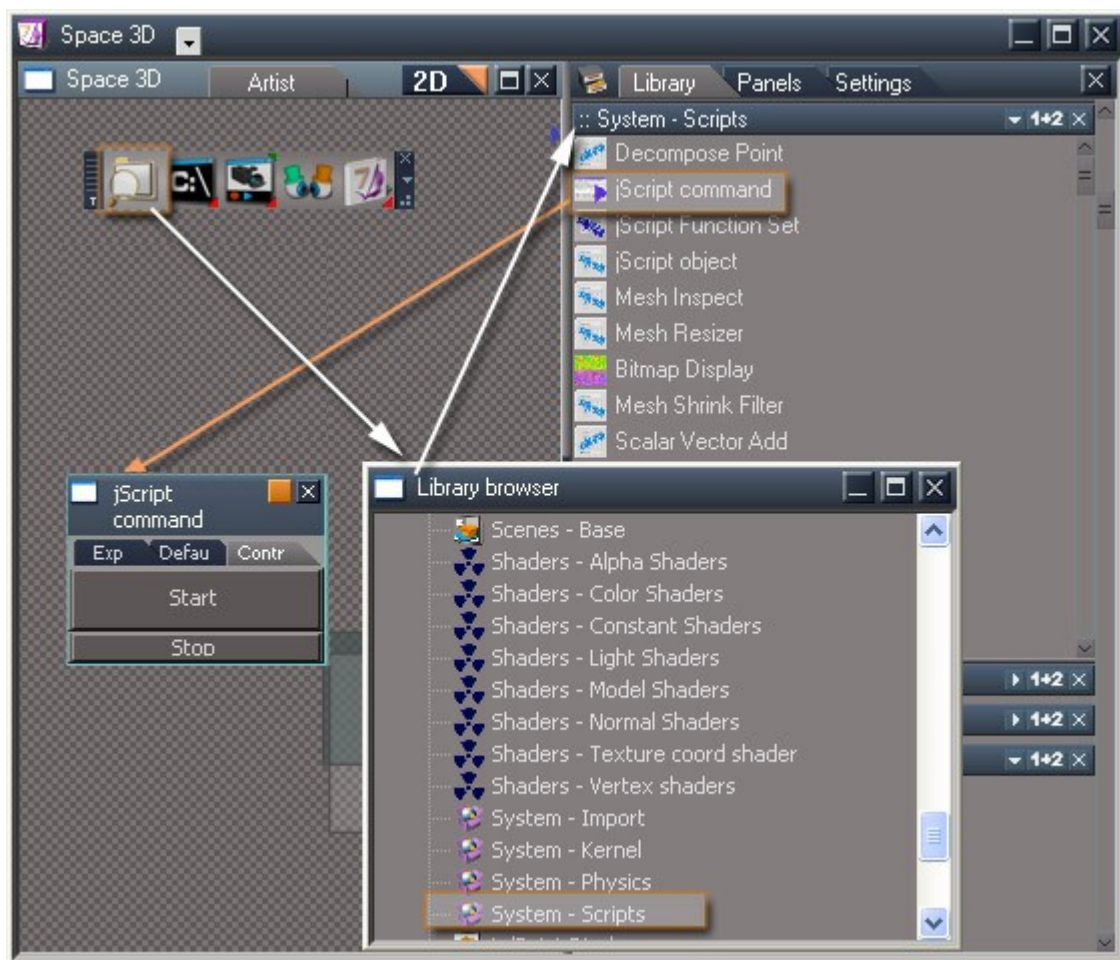
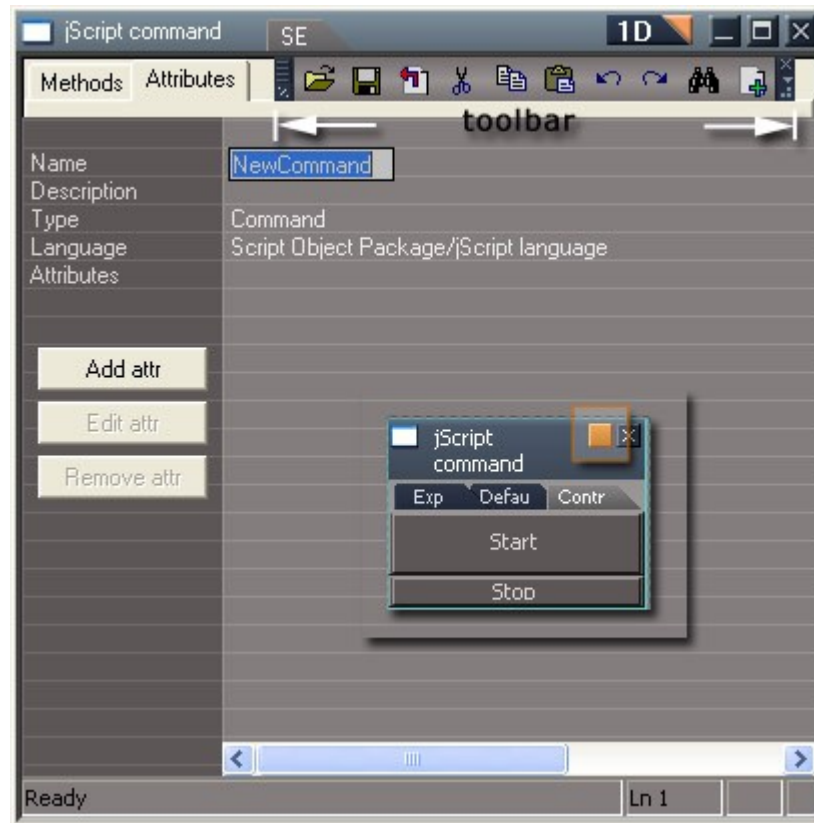


Figure 2: Accessing the Library Browser

Once we have access to Library Browser and the System-Scripts library, it is time to take a look at this scripting envelope. It can be called an envelope because it really has nothing in it yet. We know the envelope has a purpose, but is of little use when empty. Looking inside the jScript command object, we are first presented with the Script Editor (1D View). We are shown the jScript command's Attribute tab as

illustrated below. Note on the right-hand side of the title-bar, we see “1D” button with orange triangle button. That triangle button is your exit out of the Script Editor, back into the Link Editor.

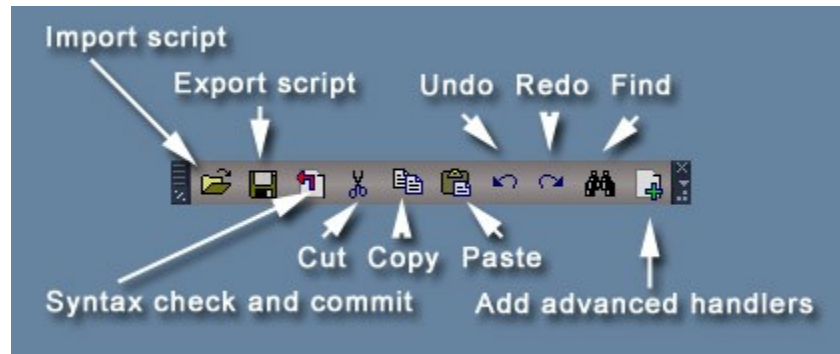


1D Script Editor Figure 3: Attributes tab Super-imposed jScript command for illustration

Note the super-imposed initial jScript command from the Link Editor to show you on its right-hand side is an orange rectangle button. This is your entrance into the Script Editor. It is vital that you be able to navigate objects in trueSpace. The orange buttons are your key to remember.

Once inside the Script Editor on the Attributes tab, we have some items to go over. Straight away we see the toolbar (it floats) at the top right hand side of the Script Editor, just under the title-bar area. This toolbar contains:

- **Import script:** Opens a dialog to allow you to import a script.
- **Export script:** Opens a dialog to allow you to export your script.
- **Syntax check and commit:** The script code is checked for potential problems and committed.
- **Cut:** Cut selected code from script.
- **Copy:** Copy selected script code.
- **Paste:** Paste script code from clipboard/memory to selected place in script.
- **Undo:** Undo last edit.
- **Redo:** Redo last edit.
- **Find:** Search in current code.
- **Add advanced handlers:** Advanced handlers are added to the bottom of any existing code.



Script Editor toolbar

The next vital elements on the Attributes tab of the Script Editor involve its description as follows:

- **Name:** notice in the earlier image shows the area to the right of the Name and ability to edit the name as desired. Hit return after naming your script with a descriptive type of name.
- **Description:** a few words about reason or purpose of the script can be placed here.
- **Type:** trueSpace fills this in for us.
- **Language:** trueSpace again fills this in for us depending on which language the object is based on.
- **Attributes:** signifies the beginning of listing for any attributes that have been added to the object.

What is an attribute? An attribute is a mechanism, which allows us to communicate with the outside world from within the script. The script will rely on the values of these attributes. In the code, these attributes may be referenced or changed by the code. By creating attributes, we empower the script object. As you progress through the next area of the manual, attributes and how they are used will be described in more detail.

Adding Editing and Removing Attributes

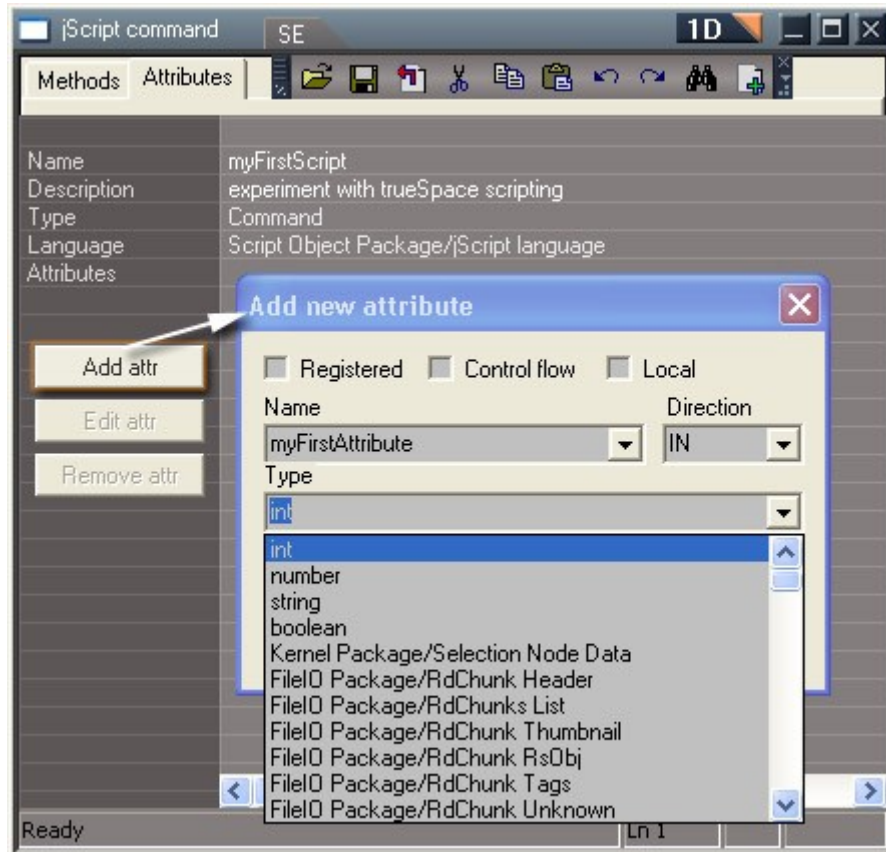
Clicking on the Add attr button brings up a dialog, which is used to add a new attribute to the script envelope. The image below depicts the Script Editor Attributes tab showing a Name, Description, Type and Language filled in. The Add attr button was pressed and the Add new attribute dialog has shown up. When we see the Add new attribute dialog, first on the panel we see:

- **Registered:**
- **Control Flow:**
- **Local:**

By default, none of these options are checked. As you progress in your scripting adventures, you may find a use for them. At this stage of the learning curve, they can be safely left at default.

The next item to cover is the Name of the Attribute. Names are important in the overall scheme of scripting. Using what is termed “descriptive naming”, you reflect on what you want the attribute to do within the script. You have a general idea of what your script will accomplish, so you begin to name attributes so they “remind” you of their intent. For our purposes, using myFirstAttribute is a fitting descriptive name.

Next to the Name, on the right, we have the Direction this particular attribute will take. There are only two directions an attribute may have, IN or OUT. If the attribute is set to have a direction “IN”, then the attribute is considered as an Input Connector. If the attribute is set to have a direction “OUT”, then it is considered as an Output connector. You could look at attributes as being either Input: raw materials or alternately Output: finished product. The script will use the Input attributes, along with actual script code, to calculate Output attributes.

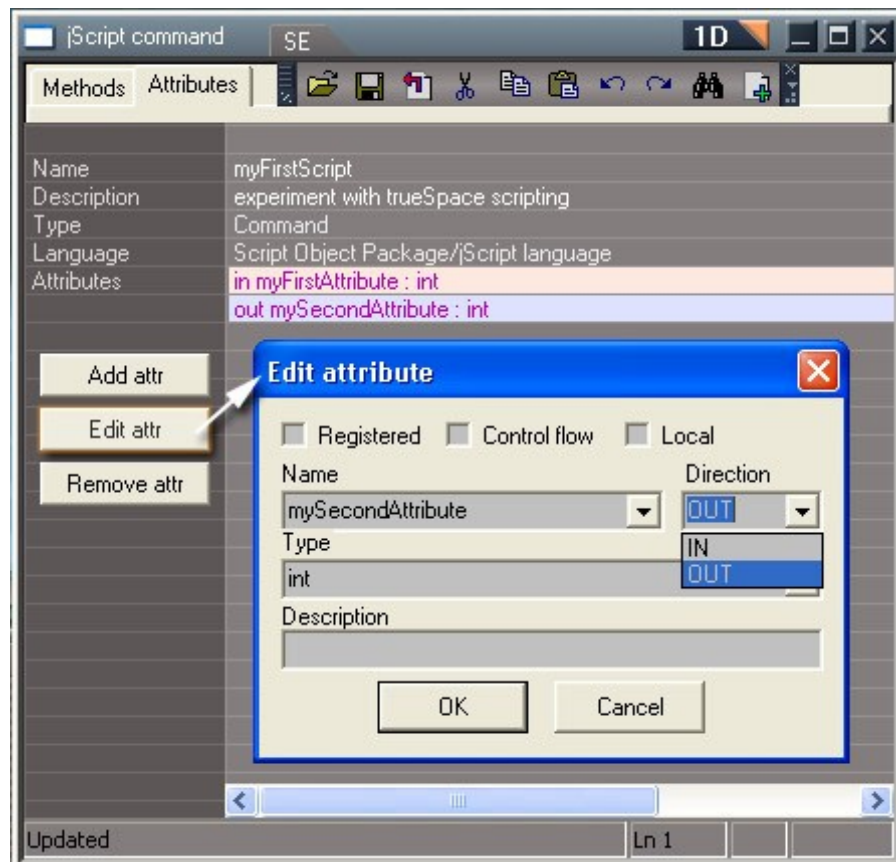


Adding an attribute to our jScript command object

Once you have decided on the direction your attribute will take, you must decide on the Type of attribute you wish to use. trueSpace has many different types of attributes to cover many different areas. The most simple of which are:

- **Int:** an integer has no decimal place(s). It only uses whole numbers like 1, 10, 36, 102458 etc.
- **Number:** now we can use decimal places. Uses numbers like 1.23, 16.0, 0.23459 etc.
- **String:** a non-number. Treated like text as in String-of-text: “My trueSpace Rocks!” etc.
- **Boolean:** True or False.

For purposes of this illustration, both our Input and Output attributes are set as Type: int (Integer).

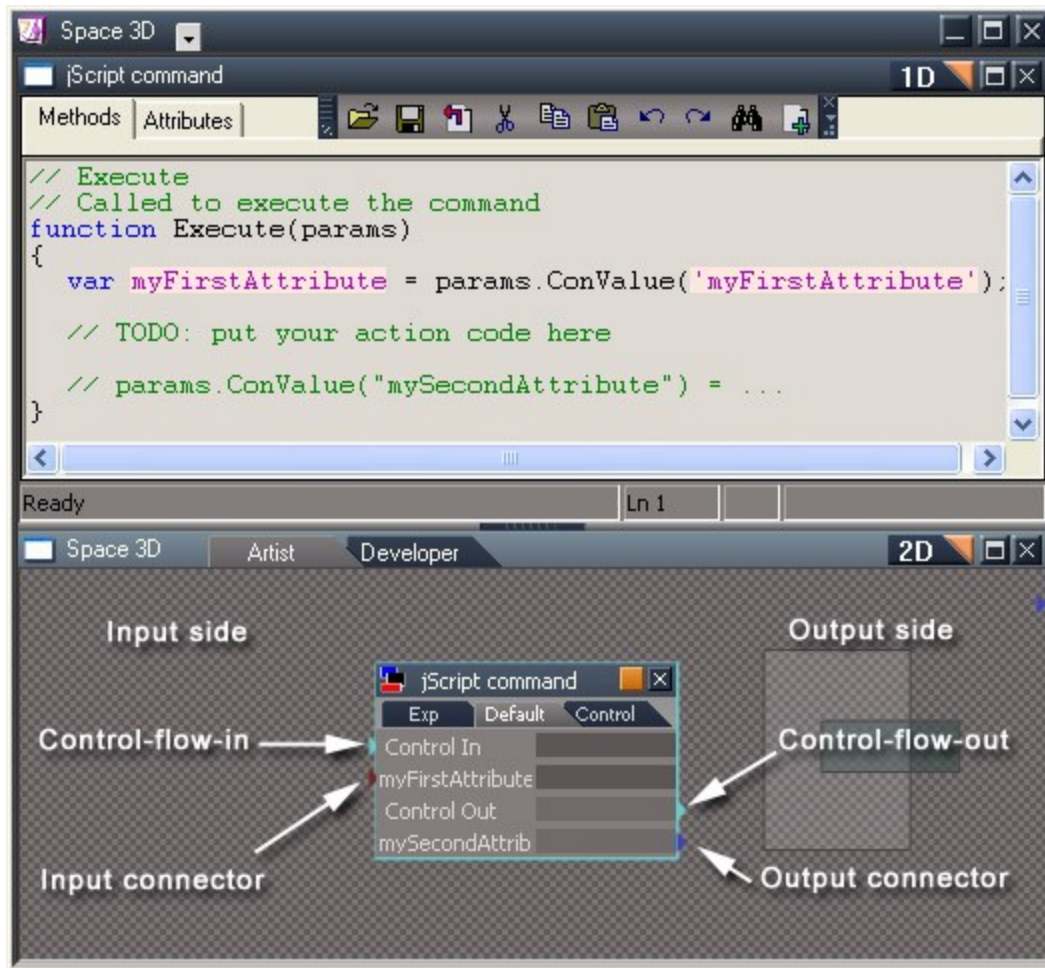


Editing an attribute

Here in this second image above, the Script Editor now shows the two attributes we created, listed to the right of Attributes area. Notice how our Input connector is red in color? This indicates that it is an Input attribute. You also notice that the Output attribute is colored blue. Both these colors are important to notice. As we will see in actual script code, there will be red and blue highlighted text/words, to indicate Input or Output. These colors become helpful over time, as you will discover.

Once you have set the Type of attribute, we have a Description area, which we can use to describe, if desired, the purpose of the attribute. If filled in, the description will show up to the right of the attribute area, on the same line as the attribute. This area is useful to remind you what the attribute is being created for. It has no bearing on the script or how it functions. It is purely a user-friendly area for you to utilize.

Once you have completed naming, describing and adding attributes to your jScript command object, it is time to examine the changes, which have taken place as a result. The image below, illustrates how the object now looks in the Link Editor (bottom window) as well as how the actual code looks when we change tabs from Attributes to Methods in our Script Editor. You will notice that we already have the “skeleton” of the code created for us by trueSpace, based on the information we entered over on the Attributes tab of the Script Editor. More on this initial code will be discussed later.



Reviewing our changes so far: Script Editor in top window, Link Editor in bottom window

The bottom window in the image above illustrates the Link Editor and changes that took place to the exterior of our jScript command object. Because this is a command type of script object, trueSpace automatically creates the Control In and the Control Out connectors for us. The arrows point to the exterior connectors. They are an arrowhead/triangle shaped and colored indicator on either side of the object. The way the indicator points will also tell us if it represents an Input or an Output connector. The color tells us as well. Remember we discussed red or blue highlighted text over on the Attributes tab? This is the same as direction. Notice the control-flow connectors are both green in color. Green means control-flow in trueSpace. Make sure of the direction for the control-flow connectors and you will be safe.

The illustration above shows by default, how the exterior connection indicators will be located. It is possible that these connectors may show up on top or bottom of the object. When minimized, these connectors will gather around the minimized object any way they can.

Now that we have reviewed the changes that took place after our work in the Attributes tab area of the Script Editor, it is time to review the code that makes up the script command object. So before we have typed a single character ourselves, trueSpace has populated the Methods tab of our Script Editor with the following code:

```

// Execute
// Called to execute the command
function Execute(params)
{
    var myFirstAttribute = params.ConValue('myFirstAttribute');

    // TODO: put your action code here

    // params.ConValue("mySecondAttribute") = ...
}

```

Because this code is using JavaScript as its language, we first see two lines of comment, indicated by the double forward-slashes. There are two other lines of comments further down in the script code. Notice these lines are green? Green in your script code represents “comment” lines. These lines are not of any importance to the script; rather the author of the script, to describe what the code means or perhaps its purpose, uses them. You can also leave reminders to yourself describing what needs to be done:

```

// Remember: pick up some milk and bread on way home from work.

```

The very first line of “action” follows these first two comment lines, which describe to us that the Execute function is called to execute the “command” object. This simply means that the code is executed when the script is “run”.

```

function Execute(params)

```

Because this is called the Methods tab, we use methods to accomplish our goals. Functions are considered methods. They perform a task for us. In this first script example, we will have the Execute function do a very simple task for us. We will have the script place the same value that is found in the Input attribute `myFirstAttribute`, into the Output attribute `mySecondAttribute`. In order to accomplish this, we first observe that the function itself is “opened” using a curly brace character (see it one line below the function `Execute(params)` line of code. The last line in the entire script is a close curly brace character. This defines the function. They open then close or end the function. They help you determine where a function starts and ends so keep them in mind.

Just below our opening brace for the function we have a line of code that performs a certain basic task for us. This line of code creates that exterior, Input-connector, arrowhead/triangle-indicator for us and places this line of code into the function for us:

```

var myFirstAttribute = params.ConValue('myFirstAttribute');

```

In scripting, variables are components of script that are subject to change. These components must be “declared” in order for our scripts to run properly. The “var” at the beginning of this line indicates just that. We have a variable named `myFirstAttribute`. It is equal to the exterior Input attribute connector named `myFirstAttribute`. This line is a virtual connection between that exterior connector and this interior attribute.

Pretty slick right? We did not have to type any of this, however it is important for you to understand how the mechanism works in trueSpace. You may decide at some later point in time to add more Input

attributes. The problem is that trueSpace will only do this for us once. As soon as we finished our Attributes tab work. Once we change to the Methods tab and edit the code ... this feature is no longer available for this particular script command or simple script, depending on what you are creating. You can safely switch from the Methods tab to the Attributes tab, as you are Adding (or editing) attributes. The minute you about editing your code, you have committed yourself to end of this feature. But if you are observant, you can easily make your own lines of code to accomplish this connection between any new Input connectors and interior variables by using existing lines of code, as your guide.

The very next line of trueSpace generated code, is another comment line. It tells us to place code in this area of the script as desired. This is where we would add any tasks we may wish to do in our script.

The last commented line of code is a bit special. trueSpace does not wish the code to execute until the author (you) have had a chance to edit the script to do as you require. It does however add this special line that can be activated by removing the double forward-slashes from the beginning of the line.

```
params.ConValue( "mySecondAttribute" ) = ...
```

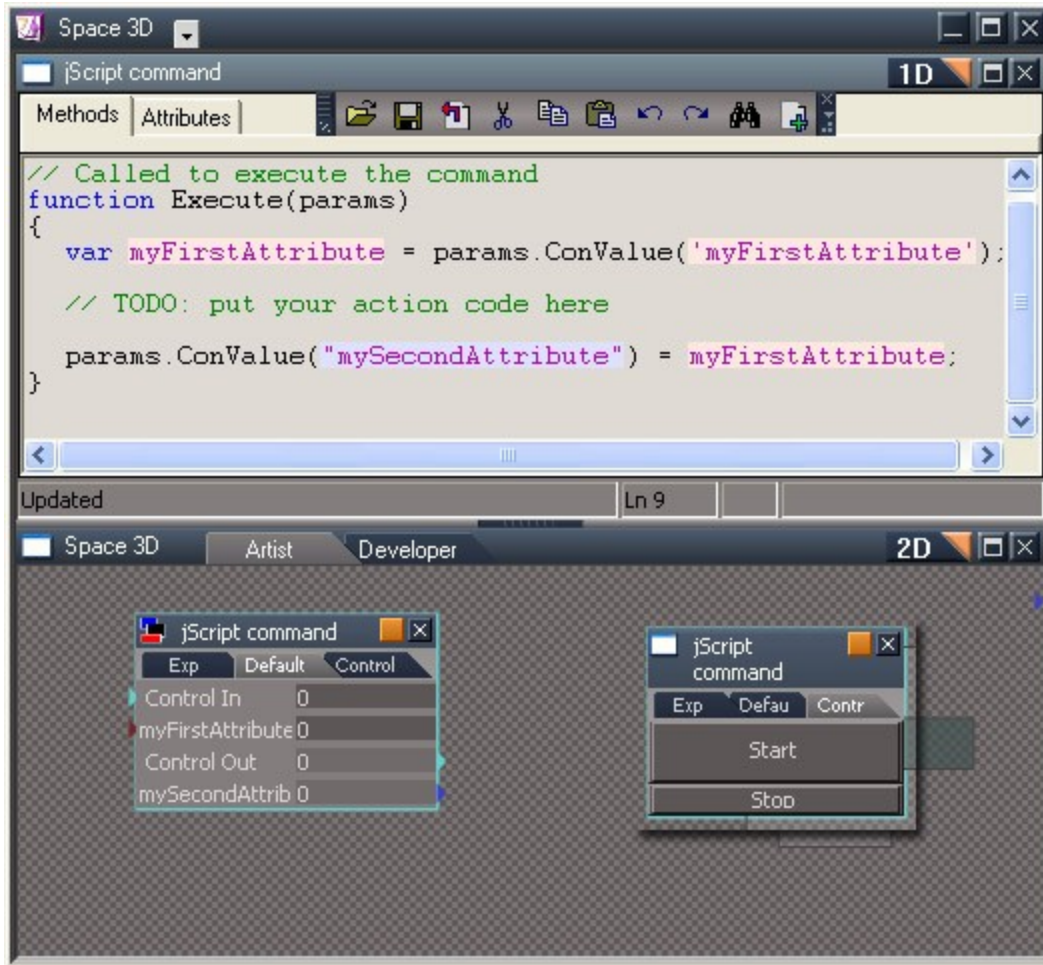
Notice in your script editor how this line of code has blue highlighting? This indicates the Output connector is being connected to an interior (incomplete ...) value/attribute.

We do however have to fill in the right-hand side of the equal sign above, in order for the script to perform an update of the Output attribute `mySecondAttribute`. trueSpace indicates this by the ... three period characters. This is what was meant earlier when it was mentioned that the line of code is incomplete. For our illustration purposes, we will simple make the Output connector equal to the Input connector by adding the following to the line of code so it now looks like this:

```
params.ConValue( "mySecondAttribute" ) = myFirstAttribute;
```

The line of code now simply says that the Output connector named `mySecondAttribute` is equal to the variable `myFirstAttribute`, which is highlighted red to indicate to us it is a known Input connector. With jScript, the semi-colon “;” indicates the end of a line in code. Once this is added to end of the line, we have a finished script object. It does not do much, but it is performing a task and therefore considered a valid script object.

The illustration below shows the bottom window Link Editor, with the jScript command object on left in its default aspect/state and super-imposed on the right side, the same jScript command object is shown in its Control aspect/state. This is an important state for a “command” type of script object. It shows 2 buttons. One will start the execution of the script and the other will stop the execution of the script (if required). Some command scripts types are intended to run through once and that would end their “life”. Other command scripts types can run in a loop, which means they would not “stop” unless the stop button was used. Either way, these buttons are available on the command script objects to server this purpose.



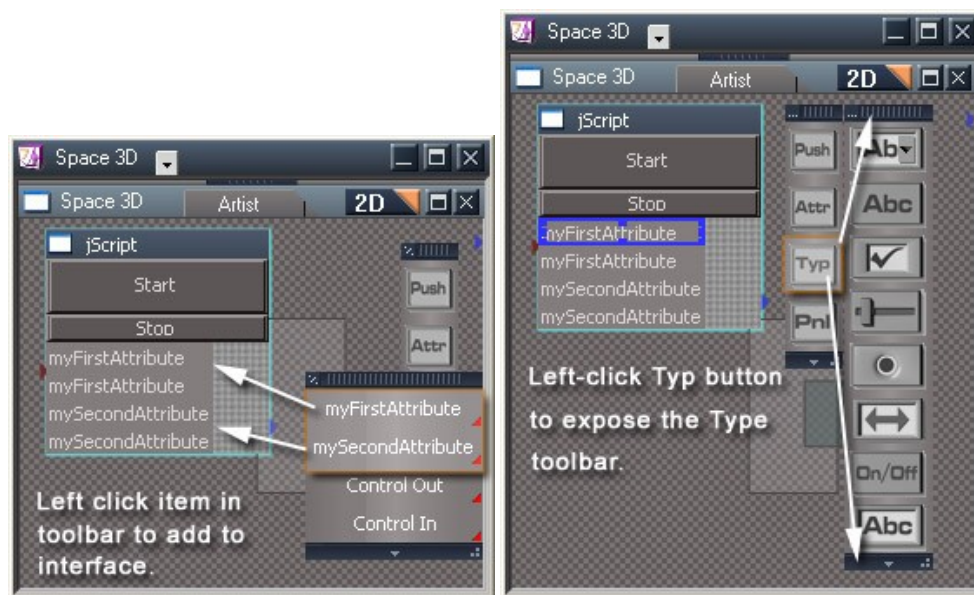
Notice Link Editor right-hand side, super-imposed jScript command showing Control aspect

It is possible to change the way an aspect looks. The control aspect of our script command object has only two buttons showing. We can edit this “interface” and make it more user- friendly. Each object within the Link Editor has this capability, as illustrated below, right-click on the object’s title-bar to reveal the first toolbar related to panel/interface editing. This toolbar has four buttons, which will call up additional toolbars or edit panels for the object. At this point in our illustration, we wish to access the Attributes toolbar so left-click on the Attr button on the toolbar, to bring up the Attributes toolbar, shown in the image on right-hand side below.



Editing the object's interface in the Link Editor

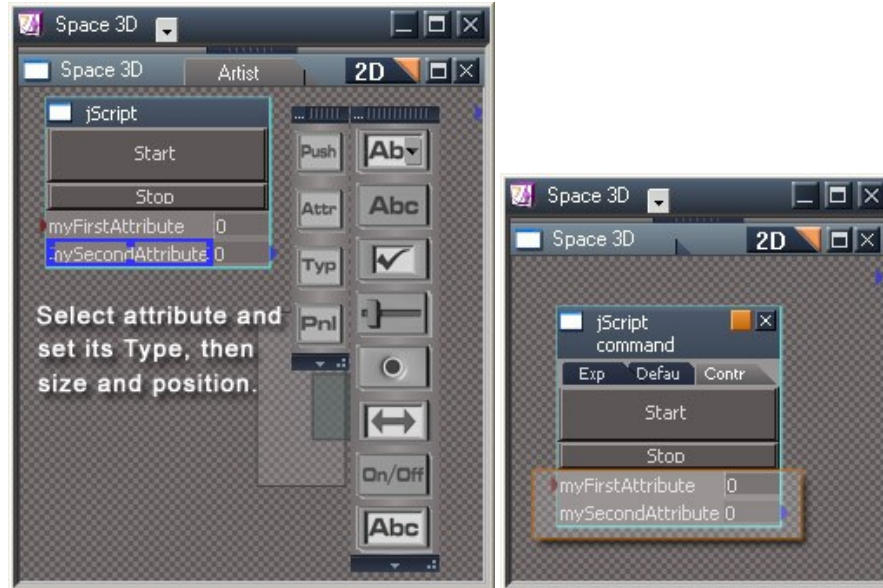
You see from the second toolbar in the image on the right, our attributes are listed along with Control Out and Control In. This toolbar will list all the attributes, which you set up on the Attributes tab of the Script Editor. Now on this second toolbar, we click twice on the top two attributes, myFirstAttribute and mySecondAttribute. As we do this the panel changes to reflect our additions to the interface as shown below.



Adding and editing elements to the interface

We added two of each attribute so one can be used as a text description (which we see already), while the second one can be set to show the value. We do this by selecting the Edit Control (bottom button on the Type toolbar; ABC) from the Type toolbar. By left clicking on an element in the panel/interface, it becomes selected and will highlight blue as shown above. This indicates that element only has focus at the moment

for editing. By selecting one of each named elements and changing its type to the Edit Control, we can then move and align the elements as shown below.



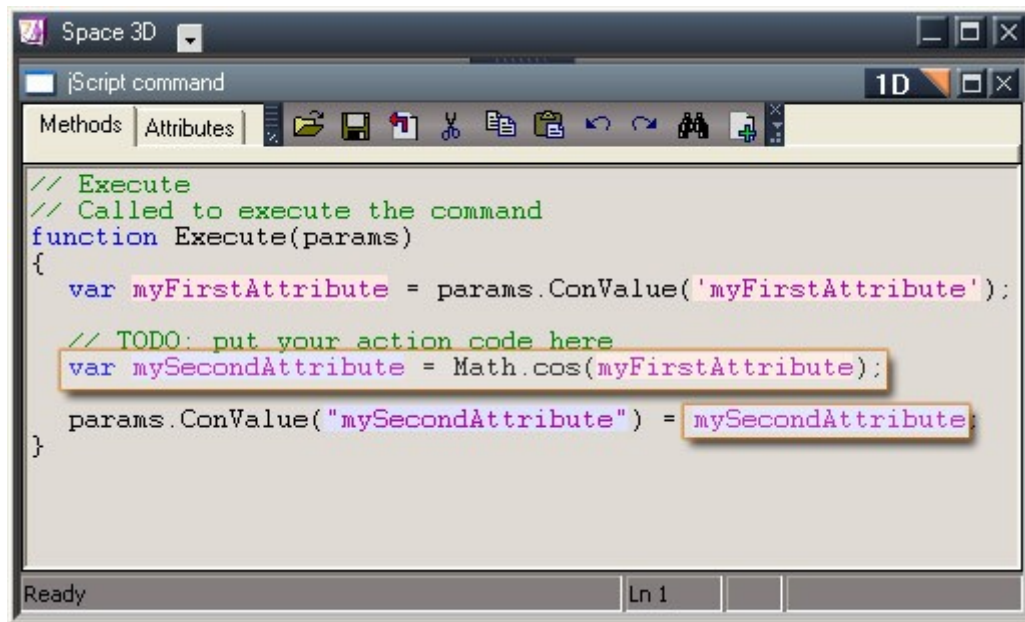
Editing elements on left and final interface on right

It will take a little getting used to how to resize and move around the elements in the interface; however with a little practice it becomes easy enough. When you are finished editing the interface, right-click the title bar of the object once more and select “Replace” from the menu. On the right-side image above, our finished interface is shown with highlighted area we just added.

If you have followed along and created your own script command object in the Link Editor, now is time to test it out. Enter an integer in the myFirstAttributes Edit Control (it shows 0 above). Once you enter the integer value, left-click the start button. The start button activates the script. All that should happen is the value for mySecondAttribute now matches the value in myFirstAttribute area. If this does indeed happen for your script command, you have successfully completed your first script in trueSpace. Congratulations!

Admittedly the script does very little except make the Output connector equal the Input connector. How about if we edit the script just a little and do something useful with it afterwards?

We begin by entering the jScript command object and enter the Script Editor’s Methods tab. In the illustration below, the highlighted areas were changed. Lets take a closer look at these changes so you understand what happens next.

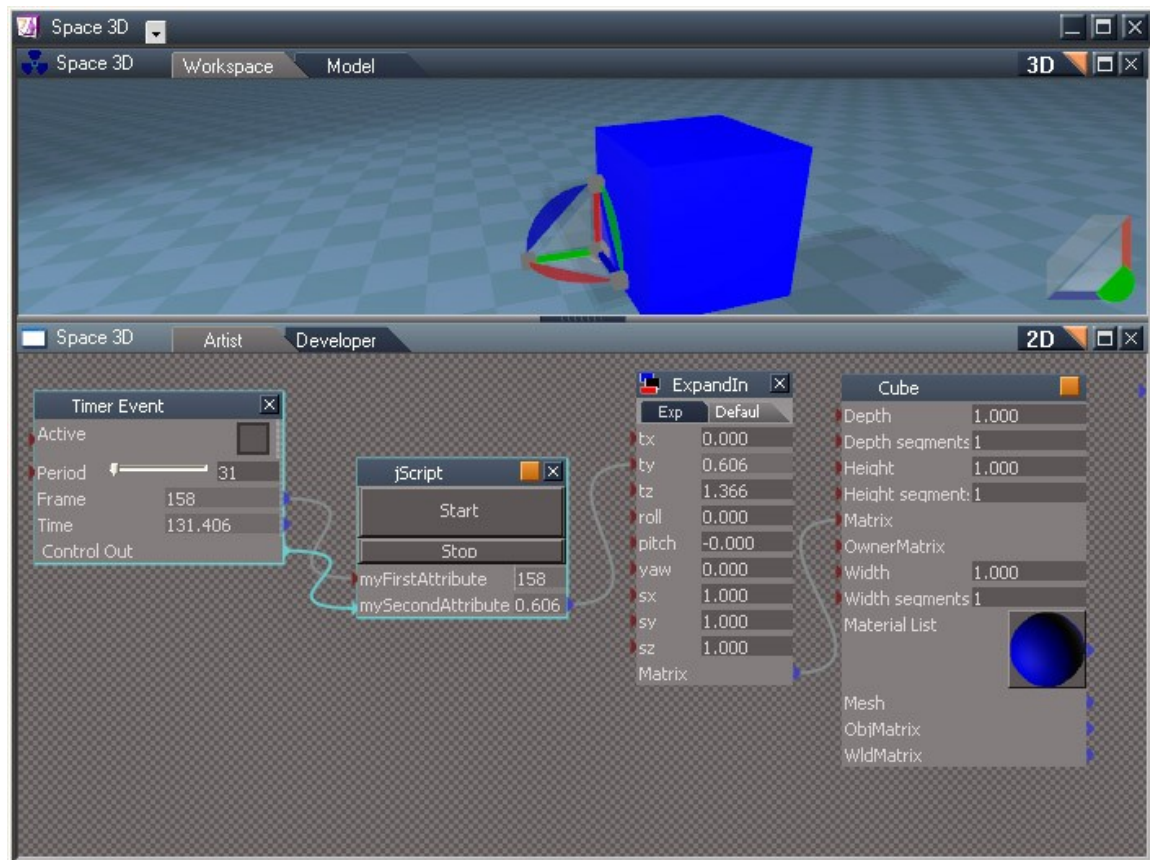


Editing a script object in trueSpace's Script Editor

The first change made is this line:

```
var mySecondAttribute = Math.cos(myFirstAttribute);
```

We declare a new variable using “var” and that variable is named `mySecondAttribute`. This new variable is equal to a math function, which pre-exists for both jScript and VBScript. `Math.cos` does a mathematical calculation on the value shown inside the () characters. In this case we use the value of `myFirstAttribute`. The final edit made was switching the right side of the equal sign in that last line of code to equal our new variable; `mySecondAttribute`. Now the numbers will no longer be the same as each other. One will represent the cosine value of the other. You should be able to find these types of functions with online searches using your favorite Search Engine. Try using jScript math functions ... or VBScript math functions ... as search criteria. You will receive a large number of hits, so spend a little time and find a page who's author is easy to understand and bookmark that site for future reference. When I was researching cosine, I found <http://mathworld.wolfram.com/Cosine.html> as an excellent site overall. I soon discovered it had more than just simple sine and cosine descriptions. The value of cosine is never greater than 1.000 nor less than -1.000.



Setting up a scenario to try the script changes.

Here is how we set this up. From the System – Kernel library (you will need to use your Library Browser to access this library) find and drag the Timer Event object into the Link Editor to the left of the jScript command object as illustrated above.

Create a simple cube object (paint it blue if you like) and look at the Expanded aspect of the Cube object. You see it has a Matrix Input connector? Right click on the Matrix Input connector and select “Expand” from the menu that appears. When you do, the object between the jScript command and the Cube appears. This represents the Cube object’s location, rotation and scale values. The object is called ExpandInMatrix.

Now if you change the jScript command object to see its Expanded aspect, you will be able to connect the Control Out from the Timer Event object to the Control In on the jScript object. It is pretty easy to do, just left-click-hold and drag the little green arrowhead from the Timer Event (you see a green wire appear as you drag towards the jScript command object) over to the Control In connector of the jScript command object. When you see the jScript command object’s Control In connector light up and highlight, let go of mouse button and you should see the green wire connecting the two objects. Do the same thing between the Timer Event’s Frame Output connector and the jScript command object’s myFirstAttribute Input connector. The last connection you will make is between the jScript command object’s mySecondAttribute Output connector and the ExpandInMatrix object’s ty Input connector. You can see these three connections in the image above.

To test, check the Timer Event's Active checkbox and watch what happens to the Cube object in 3D View. When you are finished reviewing how the cube moves along its y-axis, uncheck the Timer Event's Active checkbox to stop the scenario. You can see how using `Math.cos()` in our script, allows us to move this cube by a small number along the y-axis. As mentioned earlier, between -1.000 and +1.000.

Video Tutorials: Developer's Guide

-  DevGuide1a
-  DevGuide1b
-  DevGuide1c

Tutorial: Modifying a Script

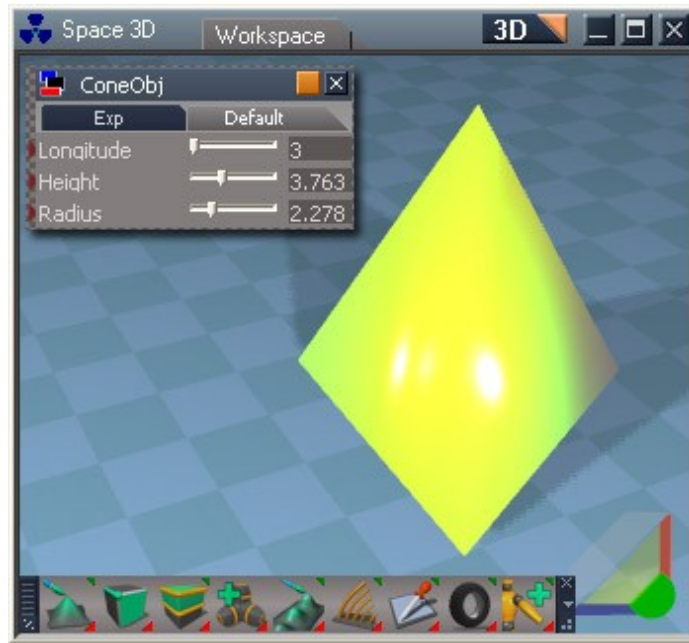
As a quick introduction to working with scripts in trueSpace, follow along with the example below to change the way the cone in the ConeObj object is created. In this example we will change just one line of script to dramatically alter the way the final cone geometry is laid out.

To get started open the Object – Script Objects library. Locate the ConeObj object and drag it into the Link Editor. You should see the object's panel, complete with three sliders for controlling the construction of the cone. In addition, we also see three numerical values representing the values of the sliders.



The ConeObj object as it appears in the Link Editor

The Workspace window will show a greenish cone. Try moving the sliders on the ConeObj's panel. You should see the shape of the cone change as you move them back and forth. These sliders are hooked directly to attributes that the script is using to define how the cone mesh itself is constructed.



The Workspace window shows the results of the script as you change the sliders

When you are done experimenting with the sliders enter the ConeObj object by clicking on the orange enter triangle on the upper-right side of the object's panel. This will take you inside of the object, the contents of which are shown in the image below.



Inside the ConeObj we find a Script Object called ConeMesh

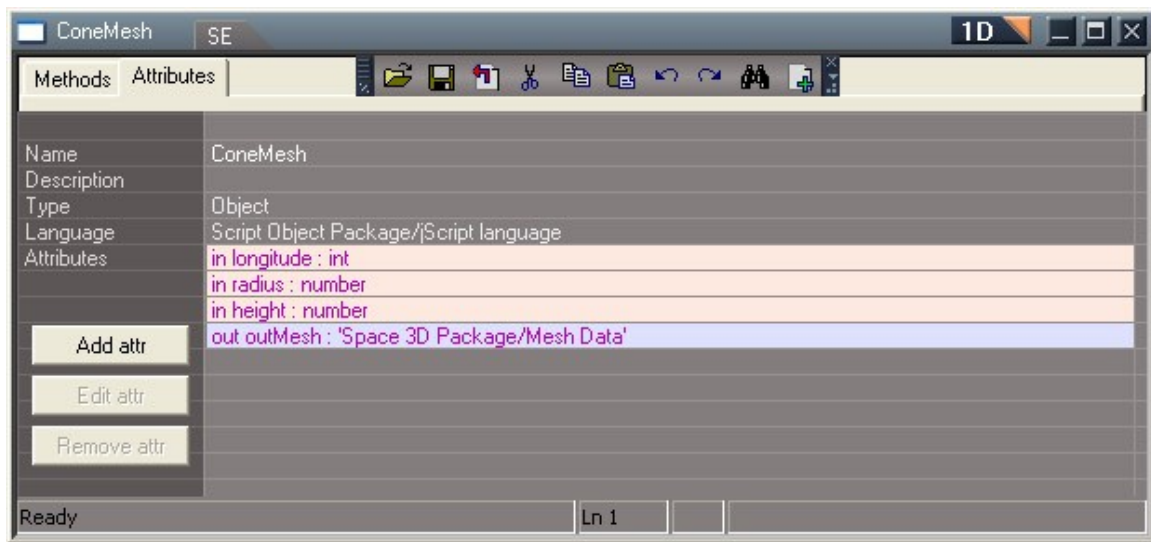
The object contains some additional objects that control how the object itself is displayed:

- Gooch2 material object: defines the material of the object.
- Transform object: defines where object is located as well as the size.
- Shape object: defines the shape of the object.

The most interesting object here is a Script Object called ConeMesh. You can see that this object has three Input attributes: height, longitude, and radius and that these are exported, in fact these are the parameters that are being changed when you move the sliders on the main panel.

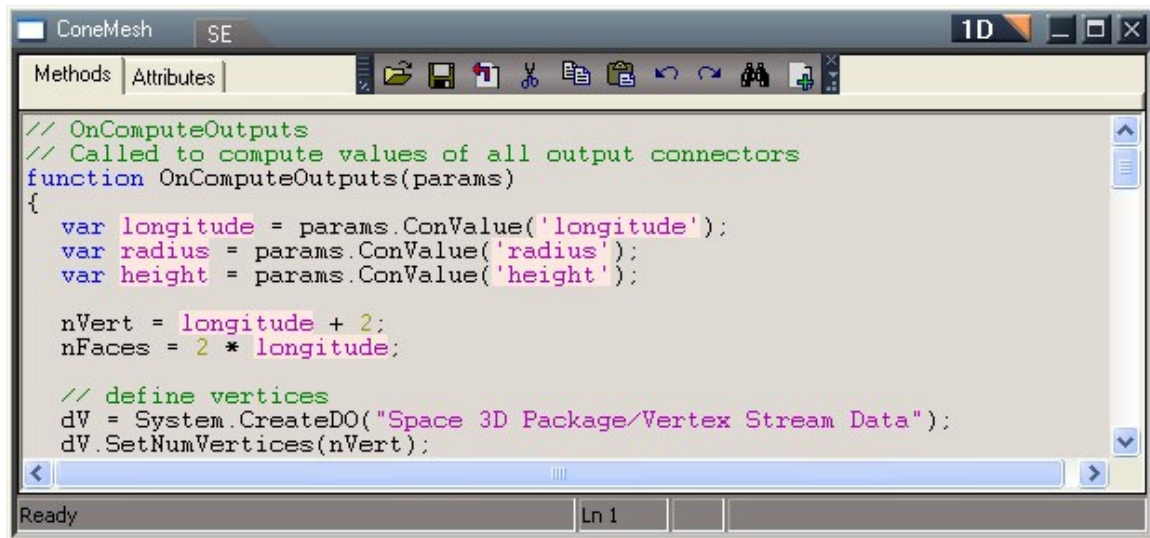
ConeMesh also has an Output attribute called outMesh which is connected to the Shape object. outMesh is where the script inside ConeMesh deposits its finished product. The mesh data is then passed on to a Shape object allowing the cone to be rendered by trueSpace.

Enter ConeMesh by clicking on the orange enter. This will open the Script Editor. Now, click on the Attributes tab to see the Script Attributes view. This is another view of the attributes on the outside panel. Longitude, radius, and height are shown as Input attributes and outMesh is shown as an Output attribute.



ConeMesh's attributes as seen in the Script Attributes view

Now, click the Methods tab to open the Script Methods view. This will give us a view of the script in ConeMesh as shown in the image below.



The ConeMesh script in the Script Methods view

There are two functions in this script, and these functions are how trueSpace objects interact with the contents of the script. The first, and most important for our purposes, is OnComputeOutputs(). This function is executed any time another object or the system requests the script to provide a value at one of its Output connectors. In the case of ConeMesh there is only one Output attribute, outMesh.

As you might guess, then, this function is executed each time the system requests the script to provide a mesh 'value' for outMesh, This is very convenient, of course, because that is just the time that we would want to create and alter the mesh's geometry. All of the code in this function is devoted to creating a mesh from vertices and faces and then providing that recipe of mesh data to the outMesh connector for the system to use. Take a closer look at the main part of this function, included below.

```

// obtain input params
longitude = params.conValue('longitude');
radius = params.conValue('radius');
height = params.conValue('height');

```

Above, we simply retrieve the values of the Input attributes to use in the script. The sliders control these particular values.

```

nVert = longitude + 2;
nFaces = 2 * longitude;

```

Next, we create two new variables and fill them with a calculation (algorithm). In this case we are determining how many vertices and faces the object will have based on the Input attributes.

```

// define vertices
dV = System.CreateDO("Space 3D Package/Vertex Stream Data");
dV.SetNumVertices(nVert);

```

This creates a special type of data object, designed to hold vertex data, and then fills in the number of vertices calculated earlier by `nVert = longitude + 2;`.

```

// ... add vertices for bottom circle

```



```

angle = 0.0;
angleStep = 2.0 * Math.PI / longitude;

for (i = 0; i < longitude; i++)
{
    dV.x(i) = Math.cos(angle) * radius;
    dV.y(i) = Math.sin(angle) * radius;
    dV.z(i) = 0;

    angle += angleStep ;
}

```

Getting a little more complex now, in this section we have a loop that steps around the outside of a circle adding vertices to the vertex data object created earlier. The loop only executes longitude value number of times; in other words, it only adds as many vertices around the circle as the value Input for longitude.

```

dV.x(longitude) = 0;
dV.y(longitude) = 0;
dV.z(longitude) = height;

dV.x(longitude + 1) = 0;
dV.y(longitude + 1) = 0;
dV.z(longitude + 1) = 0;

```

These six lines add two more vertices to our vertex data object. One at the bottom center and one at the top center as defined by the Input attribute height.

```

// define faces
dF = System.CreateDO("Space 3D Package/Triangle Vertices Stream
Data");
dF.SetNumTripleIndices(nFaces);

```

Now for the faces of the cone, here we create a special type of data object designed to hold face data, which is then told how many faces to expect.

```

// ... define bottom faces
for (i = 0; i < longitude; i++)
{
    dF.i(i) = longitude + 1;
    dF.j(i) = (i + 1) % longitude;
    dF.k(i) = i;
}

```

Next, we now have a loop to add faces to the bottom of the cone. Faces are specified by listing the three vertices, in counter-clockwise order, around the edge of the face. You do not have to understand this to work with the script, but be sure to explore the rest of this chapter and the other reference material, for more information on advanced scripts, and working with special trueSpace data types if you are interested.

```

// ... define upper faces
for (i = 0; i < longitude; i++)
{
    dF.i(i+longitude) = longitude;
    dF.j(i+longitude) = i;
    dF.k(i+longitude) = (i + 1) % longitude;
}

```

Now we add the faces on the upper part of the cone between the top vertex and those vertices around the circle we defined earlier.

```
// create mesh
dM = System.CreateDO( "Space 3D Package/Mesh Data" );
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);
```

With the cone's vertices and faces fully defined, we create another special data object; this one designed to hold mesh data and attach the vertex and face data to it.

```
params.conValue( 'outMesh' ) = dM;
```

We Output the resulting data to outMesh and complete the function. It is a lot to take at a glance but, if you didn't quite understand something the first time, just take another read through the script listing. It sometimes helps if you just think of certain parts of the code as a mysterious black box; give it this value and it gives you a result back. You don't have to know how the math library sine function works; you just have to know that if you give it an angle it will return the sine of that angle!

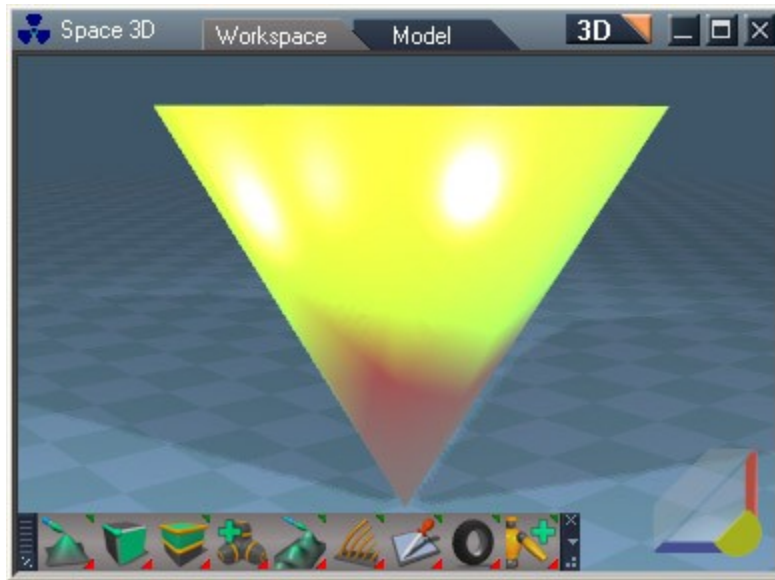
Upright cones are all well and good, but let's see what happens when we tweak some values. Find the section of the script that defines the vertices around the outer circumference of the cone. It should look like this:

```
// ... add vertices for bottom circle
angle = 0.0;
angleStep = 2.0 * Math.PI / longitude;

for (i = 0; i < longitude; i++)
{
    dV.x(i) = Math.cos(angle) * radius;
    dV.y(i) = Math.sin(angle) * radius;
    dV.z(i) = 0;

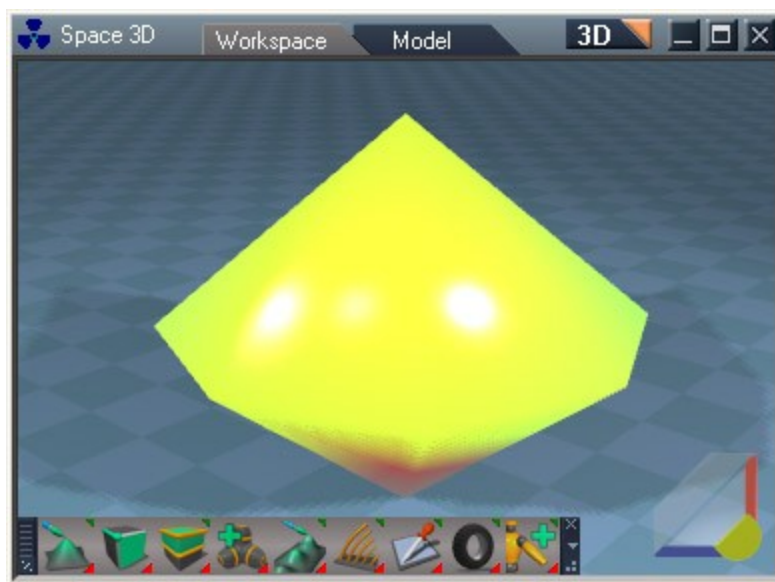
    angle += angleStep ;
}
```

Now change `dV.z(i) = 0;` to `dV.z(i) = height;` and press the Commit icon. You will probably see something like the image below in your Workspace window. All we did here is change the z value of the vertices being created around the circumference to the cone so that they would be created at the top of the cone instead.



Changing just one value can have a major effect on the result

Now, try changing `dV.z(i) = height;` to `dV.z(i) = height / 2;` and press the Commit icon. Again, we see another amazing result for such a small change. Here we created the vertices along the middle of the 'cone' mesh instead of at the bottom or the top.



Now our cone has become a top

Try changing other values to see what happens. If you break the script do not worry – just get a new one from the library and try again. When you feel that you want to move on and learn more about scripting read through the next two sections and begin exploring how Script Commands and Script Objects work!

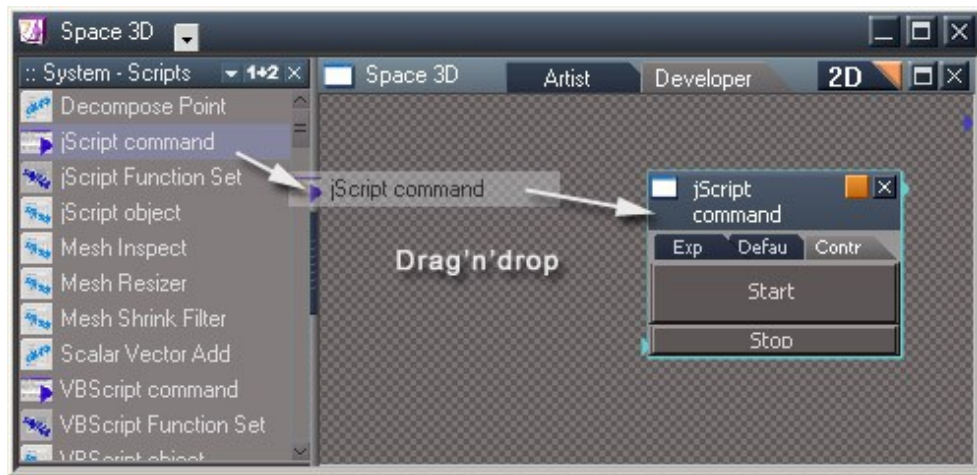
1.2 Script Commands

Script Commands in trueSpace

Script Commands are scripts that are executed *once* for every time the Start button on its interface is pressed, or once for each time it receives a control-flow pulse/signal. Another object may also contain script that effectively presses the start button. You can use a Script Command to create actions that react to a specific event, such as when you push a button or in response to a Control signal from another command type object.

Creating a Script Command Object

You create a Script Command object by dragging and dropping it from the System - Scripts Library into the Link Editor. You may have to revisit your Library Browser if you do not already have the System – Scripts library available in the Stack area. The image below illustrates the scenario after the System – Scripts library was control-dragged into the window and docked at left-hand side.

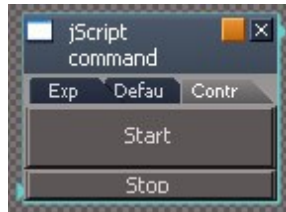


Drag and drop jScript command object into the Link Editor

The Script Command objects are identified by the word “command” after the name of the language that the object handles. The image above shows two types of command objects in the System – Scripts library; jScript command and VBScript command.

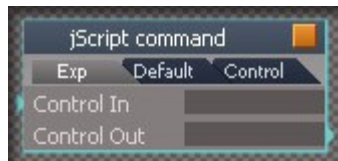
You can place the Script Command object at any level of hierarchy in the Link Editor but you will usually want to place it either within Space 3D or perhaps inside of another object with which you want the script to interact. Script Command objects can be Encapsulated within the Link Editor just like other objects. You may wish to read Chapter 2.9, which references the Link Editor

A JScript Command object as it appears in the trueSpace Link Editor is shown below. The object, by default, shows its Control aspect, which contains buttons for starting and stopping control flow in the link editor. Each time you press the Start button, the script in your JScript Command object will execute once.



A JScript Command object as it appears in the Link Editor; Control Aspect shown

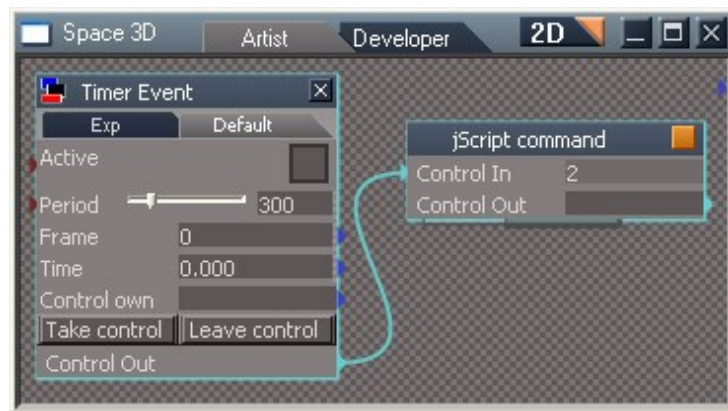
If you change to the Exp aspect you will see two items on the object's panel: an Input connector called *Control In* and an Output connector called *Control Out*. As you add Attributes to your Script Command object additional connectors representing those Attributes will appear on this panel, allowing you to connect them to other objects in the trueSpace Link Editor.



A JScript Command object with Exp aspect

Executing a Script Command

To activate a Script Command object you simply press the Start button on the panel when the object is in the Control aspect. You can also activate a Script Command by providing a control signal from another object connected to the objects Control In connector. This control signal could come from any number of Link Editor activity objects such as the one shown below.

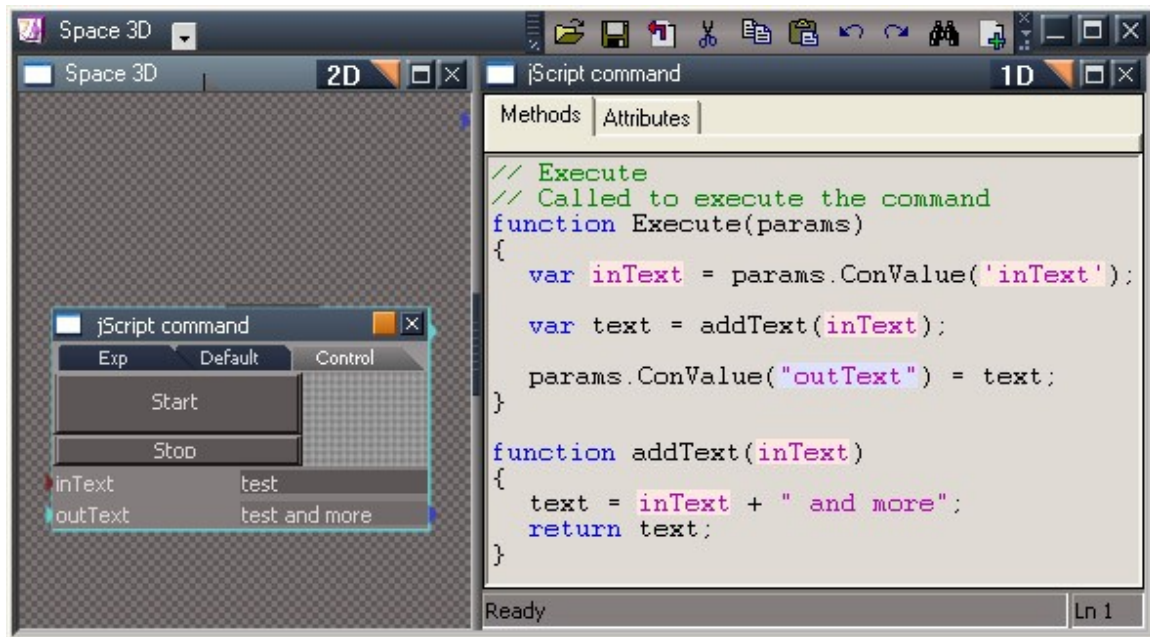


Setting up a JScript Command to execute on a Timer Event

Alternatively you can execute a Script Command with a direct call from another Script Command or from an interface element, such as a button, using the `Activity.Run(<full name of command> command.)`

Writing Script Commands

A trueSpace Script Command script has only one default function, called `Execute()`. This function is called each time the Script Command object receives an activation signal at its Control In connector or when the script is called directly by pressing the object's Start button or via a RunActivity command. You will place most of your script code within this function, though you may create additional functions and call them from within `Execute()`. Note that only script commands within the curly braces following `Execute()` will be carried out by the script interpreter, unless you call another function explicitly. In the illustration below, two functions exist in our script; `Execute` and `addText`.



Calling a function from within another function

In left-hand side we see Link Editor view of the jScript command object. Notice it has an inText Input connector and an outText Output connector. The idea is to type something in the inText field, hit the Start button to initiate the scenario. On the right-hand side, we have the Script Editor showing the two functions. The `Execute` function has the following line:

```
var text = addText(inText);
```

This line is telling the function that another function (`addText`) is to be called and that the value for `inText` is to be passed to the `addText` function. At bottom of Script Editor we see the `addText` function. The `addText` function accepts and is looking for a variable to be passed to it. It takes what it receives, then adds the words " and more" to what it was given. It then "returns" the variable "text" back to the original function that called it. Once control of script is returned to the `Execute` function, it carries on and sends the `outText` Output connector the value of "text" that it received from the `addText` function. The result we see in left-hand side "test and more". In this very simple example we are able to easily "track" the flow of the script as it moves between these functions and finishes with population of the Output connector. It is always beneficial to recreate such an example from scratch in trueSpace. Passing values between functions is not limited to a single value. You may send more than one value separated by a comma and a space character (hit spacebar on keyboard). For example `addText(inText, anotherValue, yetAnotherValue)`.

Forming complex activities from commands

If you want to create a complex project, e.g. create a game or define a special behavior for your scene, you will most likely use commands as 'building blocks' to achieve your goal, and you have to connect these bricks to one big working entity.

We can outline here several of the most frequently used schemes to form final activity:

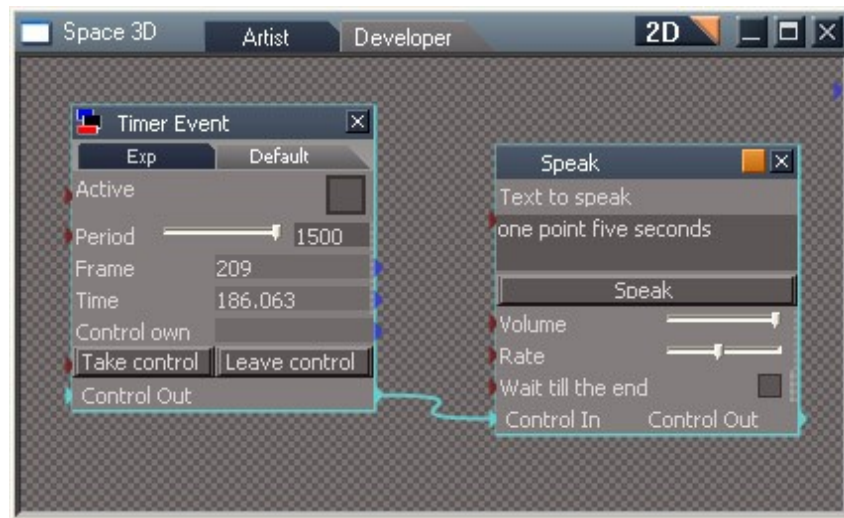
One Shot activity



OneShot activity

One Shot activity is the simplest form. It consists of one or perhaps a few commands (bricks) connected together without loops. You can run it; it performs the required action and stops immediately after it finishes. (Example: MakeTree command connected to the Run Activity command). The single control pulse that the RunActivity command provides down the wire to MakeTree, causes a single run of the code within.

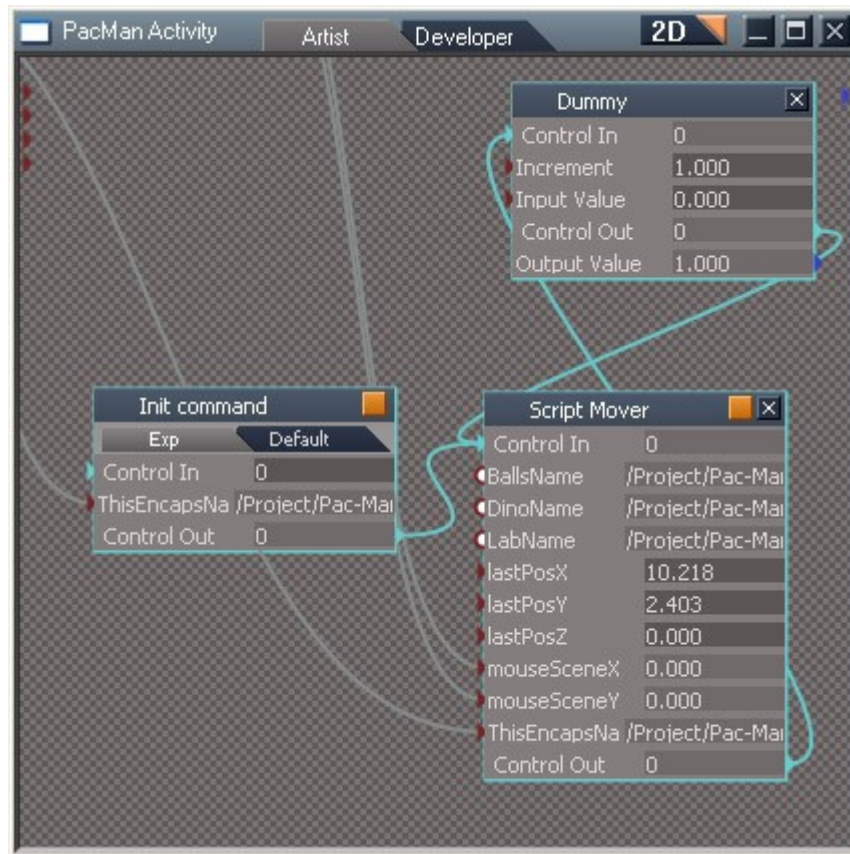
Timer based loop



Timer Event sends constant stream of pulses to the Speak object

The Timer based loop is an example of the next level up in complexity. It is useful when you want to create a loop with exact timing. Look at the example above, which was created by dragging the Speak object from the Activities-Base library, as well as a Timer event component. The Timer Event fires a pulse across the wire to the Speak component for each “Period” of time you decide, until the Timer Event is stopped. The benefit of this scheme is that you can control the speed of execution, by adjusting the timer Period. It is appropriate to create and multiple scripts to the same Timer Event. You are able to choreograph your scripts to work together.

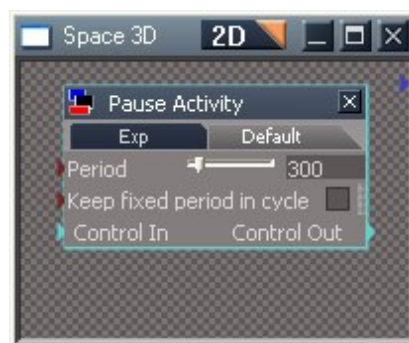
Infinity loop



Connecting commands to form a loop

Alternatively you could form a loop without a timer, by simply forming a loop with commands. In this example (taken from the Scenes-Base library: PacMan), the Script Mover command is connected to Dummy and back to form a loop. The user can stop this infinite loop, by pressing the Stop button on the main control panel of the game. The benefit here is the activity is running at top speed and you can also form complex behaviors with decision blocks, states etc.

Note: To control an execution speed of this kind of loop, or to ensure similar speed on various machines, it is strongly recommended to connect the Pause Activity element to the loop. This way you can create CPU-friendly loops with easily controlled execution speed.



Activity-Base library contains the Pause Activity component

Tutorial: Writing a direction-control Script Command

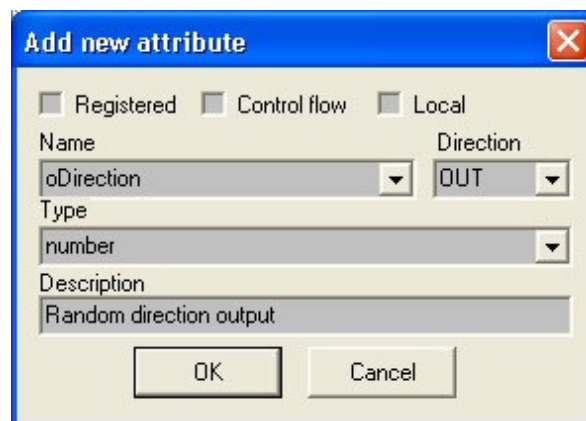
Time to create another script. We will go through the steps required, to create a direction-control Script Command object in trueSpace. This Tutorial will show you how to use a script command to change the orientation of an object, in this case, to turn the needle of a compass to face a random direction when you click a button.

To get started create a new scene and drag-and-drop a new JScript Command object into Space 3D in the Link Editor. You should always name your objects something descriptive. To do this, click on the JScript Command object's drop-down menu box and select Rename Object from the menu. Call it something simple like Random Direction.



Rename the JScript Command

Now we will create an attribute. Left click the Enter icon on the Random Direction object to access the Methods and Attributes windows. Then, click on the Attributes tab to view the Attributes window. Click on the Add Attr button and fill in the fields to match the image below. When you are done press the OK button.

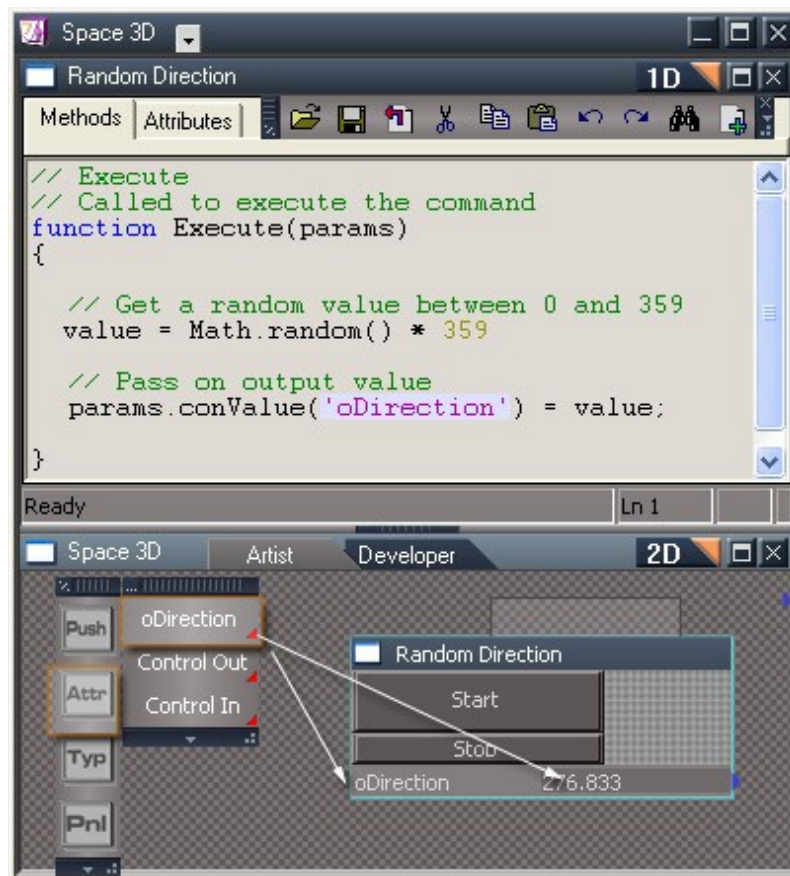


The Add New Attribute dialog box filled in for oDirection

This will create a new attribute called `oDirection`, of the type number (which can represent any real number). Attributes should also have descriptive names, such as direction, location and so on, to ensure that

users of your object can easily tell what they are used for. In this case we also prefaced the name with 'o' to remind us that it is an Output attribute while we are writing the script.

Now, we write a script that will fill oDirection with a value representing an angle from 0 to 359 degrees. To do this, click on the Methods tab to access the Methods window. In the text area type in the following script: trueSpace does most of this for you. You should be able to type in the remainder so you match the script illustrated below.



Adding code and editing the interface

This script simply accesses the JScript Math library to return a pseudorandom number between 0 and 1 and then multiplies that number by 359. This gives us a random number in the range of 0 and 359, which we then send to oDirection using the `params.conValue('oDirection') = value;` statement. So, each time `Execute()` is called oDirection will be filled with a random value ranging from 0 to 359, which just happens to be the number of degrees in a full circle, a handy thing to know of course.

Now, our Random Direction script is completed. Click the Commit icon at the top of the Methods window to tell trueSpace to update the script and check for any errors. If you have entered the script as shown above you should not have any problems. If you do receive an error just check your script against the one above. trueSpace should give you a hint as to where the problem lies.

If your script checks out just click the orange triangle to exit the script and re-enter the Link Editor. Now, change to the Control aspect of your Random Direction object and edit the panel to include the newly created oDirection attribute. The lower half of image above shows toolbars you see when editing the interface. By a right-click on the Random Direction's title-bar, the first toolbar appears. From this first toolbar, select the "Attr" button to show the second toolbar, which contains all the available Input and Output attributes available. Click the oDirection twice and arrange the new visual elements as shown. The right-hand oDirection element was changed to a text field by first selecting it (it sort of highlights blue border), then click the Typ button on first toolbar. You will see yet another toolbar appear with different types of possible ways to show the element. The different types are illustrated below.



Type toolbar to customize your interface elements

After you have customized the interface, drag-and-drop the Compass object into the Link Editor. You can find this object under the Objects Library in the Tutorial Objects panel. Next, connect the oDirection connector to the Direction connector on the Compass object. The final configuration is shown below.



Random Direction connected to Compass in the Link Editor

Assuming you have connected everything correctly you should be able to left-click the Start button on the Random Direction and watch the needle of the Compass object change each time to a random direction.

While this example is not particularly useful in the real world you can easily expand on it to do more useful work within the `Execute()` method. A more-advanced example might be calculating X, Y, and Z values for a spherical orbit using Kepler's formulae (http://en.wikipedia.org/wiki/Kepler's_laws_of_planetary_motion). You might need to add additional Input and Output parameters, but doing this is just a matter of following the steps above.

Tutorial: Creating Advanced Script Commands

We can build on what has been learned in the previous tutorial by creating a Script Command object and an activity loop to make an object perform a slightly more advanced trick – move in a complete circle. To do this we will just use basic trigonometric formulae to calculate points on a circle based on angle and radius, leaving the more complex Keplerian method or orbit calculation as an exercise for the user.

First off, as with the last tutorial, create a new scene and drag-and-drop a new JScript Command object into Space 3D in the Link Editor. Rename it something like Orbit Calculator.

We will use four attributes in our script: `iRadius`, `iAngle`, `oX`, and `oY`. To create these, left-click the Enter icon on the Orbit Calculator object to access the Methods and Attributes windows and click on the

Attributes tab to view the Attributes window. Use the Add Attr button to create entries for each of these attributes as shown below.

```
in iRadius : number 'Radius of orbit'
in iAngle : int 'Current angle along orbit'
out oX : number 'Calculated X position of planet'
out oY : number 'Calculated Y position of planet'
```

Use Add Attribute button

The integer (int in the Add New Attribute dialog) attribute iAngle will be used to Input the current angle of rotation for a planet object. The number attribute iRadius is will control the size of the circle traveled by the planet object. Finally, the two integer Output attributes oX and oY will contain the X and Y values that we will calculate to specify the location of the planet object as it travels around its orbit.

Now that we have all of our attributes created it is time to write the script. The basic procedure is simple: get iRadius and iAngle and calculate the planet object's position along the arc of the circle using the built-in JScript Math library functions. The general formula for this is $x = \sin(\text{angle}) * \text{radius}$ and $y = \cos(\text{angle}) * \text{radius}$. The completed script is shown below.

```
// Execute
// Called to execute the command
function Execute(params)
{
    // Get radius and angle
    radius = params.conValue('iRadius');
    angle = params.conValue ('iAngle') * (Math.PI / 180);

    // Calculate new position
    x = Math.sin(angle) * radius;
    y = Math.cos(angle) * radius;

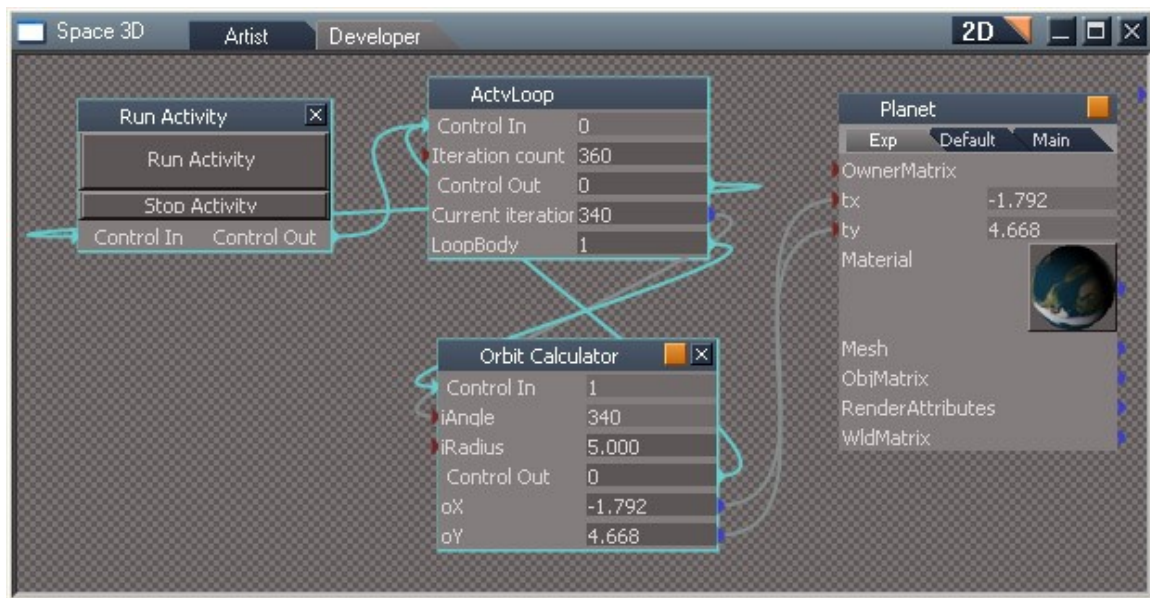
    // Output calculated value
    params.conValue("oX") = x;
    params.conValue("oY") = y;
}
```

One item of note is that we must convert iAngle to radians before we can use it with the JScript Math library functions. To do this we multiply it by π ($\approx 3.1415926\dots$) divided by 180, because there are 2π radians in a circle. The value of π is provided to us via `Math.PI`. After the conversion we can perform our calculation and Output the X and Y coordinates for the position of the planet.

Now, there is a bit more to do before we can see our script in action. In the previous tutorial we use the Start button on the Script Command's panel to execute the script. This time we will use a Run Activity to power our object and we will add an ActvLoop object, both of which can be found in the Activities-Base library.

The ActvLoop object will generate a continuous control signal for a certain number of iterations, specified by the Iteration count attribute. Drag and drop a Run Activity object and an Actv Loop object into Space 3D in the Link Editor.

You will also need to add in the Planet object from the Objects-Tutorial Objects library. Now that you have all four objects placed in the Link Editor you will need to hook up their connectors as shown in the diagram below.



The completed activity loop in the Link Editor

In order to connect the scenario:

- Connect the Current iteration connector on the ActvLoop object to the iAngle connector on the Orbit Calculator object.
- Connect oX and oY on the Script Command to the tx and ty connectors on the Planet object.

Now we turn to control flow.

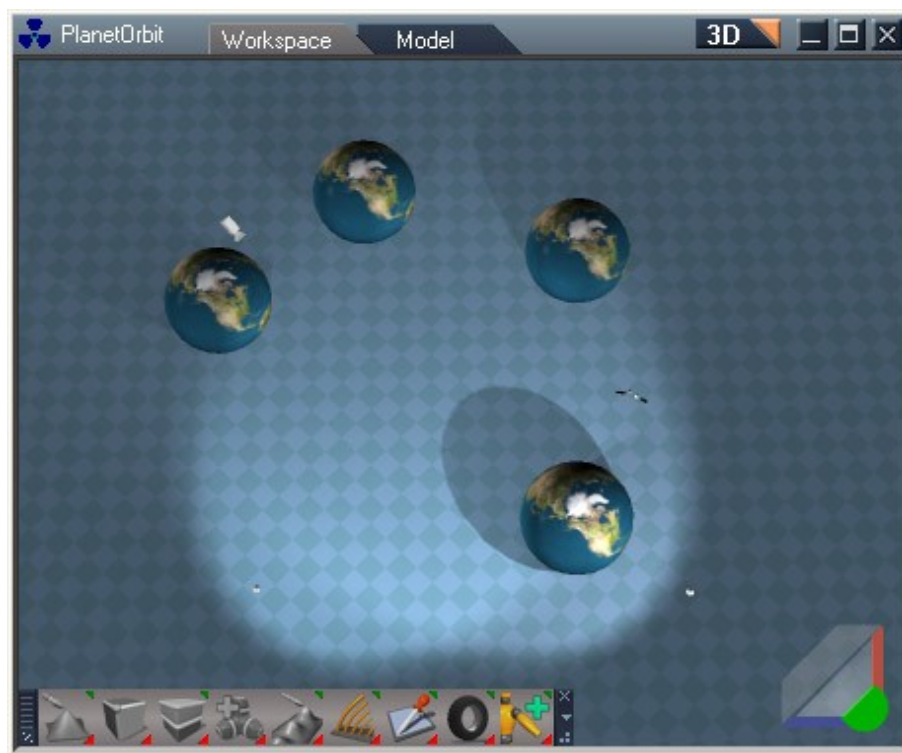
- Connect Control Out on Run Activity to Control In on ActvLoop.
- Connect LoopBody to Control In on your Orbit Calculator object.
- Close the loop by connecting Control Out on Orbit Calculator back into Control In on the ActvLoop object.
- Finally, to make the activity repeat when it is done, connect Control Out on ActvLoop to Control In on Run Activity.

Because of the loop nature of this scenario, these last few steps are required:

- Enter 360 into the Edit control area for Iteration count on the ActvLoop object.
- Enter a small value into the Edit control for iRadius on your Orbit Calculator object (5.0).
 - If you make iRadius too large the planet may be transported outside of your view range.

If you have followed the preceding steps correctly you should be looking at a pretty complex network of blue-green control connections, but do not worry, we will follow the flow step-by-step to see how simple the final product really is:

- First, when you click the Run Activity button on the Run Activity object a control signal is sent to the ActvLoop object, starting the process.
 - ActvLoop starts its iterations at 0 and counts up to 360 each time.
- Now, after receiving the control signal, ActvLoop sends a control signal of its own via LoopBody, which triggers the `Execute()` method of your Orbit Calculator script.
 - When it executes the script requests values for `iRadius` and `iAngle`.
 - Calculates values for `oX` and `oY`, which are sent to the Planet to transform its location.
- When the script is complete it generates its own control signal, which is sent back to ActvLoop starting the process over again.
- Finally, when ActvLoop has completed its 360 iterations it sends a control signal back to Run Activity starting the entire loop over from zero.



Planet orbits in real-time

With this basic loop control apparatus you are armed with the tools to create some pretty sophisticated loop-based animations and simulations.

Tutorial: Script Commands in More Complex Objects

To see how Script Command objects can be used in more complex objects we will take a look at the inner workings of the Landscape generator object. You can find this object in the Objects – Script Objects Library. The icon is shown below, simply drag-and-drop this object into the Link Editor to get started.

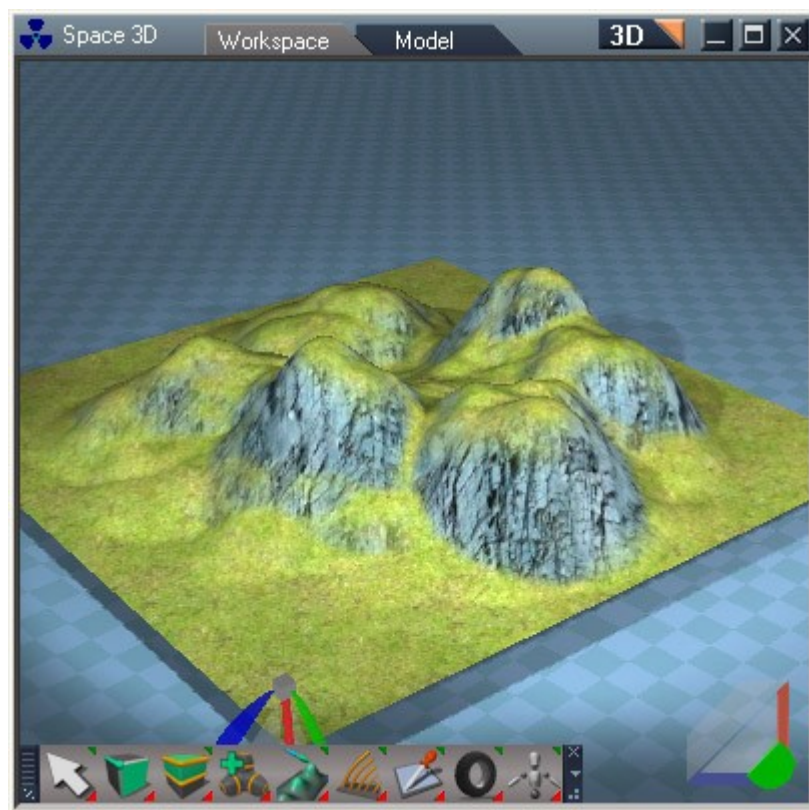


You should see the panel pictured below in the Link Editor. This panel has two buttons, several sliders, and a check box that control the functioning of the object.



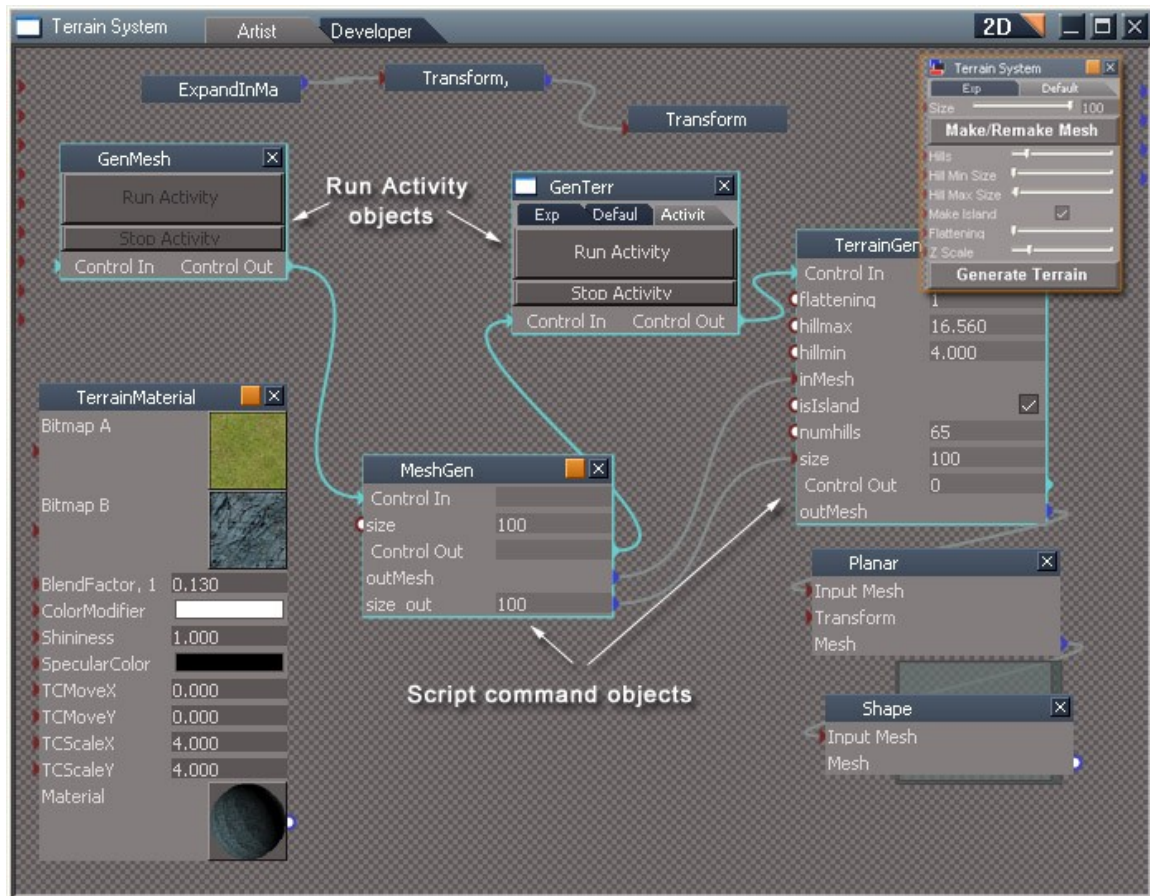
The Terrain System object panel

The image below shows the Workspace Window view of the Terrain System objects with default values. You can see that the object creates a mesh that resembles rolling hills in shape and texture.



The terrain as shown in the Workspace view

So, how does it all work? To find out, enter the Terrain System object by clicking the orange triangle on the upper right-hand side of the panel. Inside you should see several objects including a Euler Transform, a modified LayeredPlastic shader called TerrainMaterial, and a RenderAttributes object. These control the position, orientation, size, and position of the object and are interesting in their own right but the real meat of the object is the collection of objects pictured in the diagram below.



Inside the Terrain System object

Here you see two Run Activity objects, two JScript Command objects, a default UV coordinate mapper object and a Shape object. The flow is actually quite simple. When the user presses the Make/Remake Mesh button on the main object panel (inset on top right of image), a control signal is sent from the GenMesh Run Activity object, which triggers the script in the MeshGen Script Command. This script, shown below, creates a square grid mesh object based on the size parameter, passed in from the slider on the main panel.

```
// Execute(params)
function Execute(params)
{
    // Obtain input params
    size = params.conValue('size');
```



```

// Limit mesh size to 10-100
if (size > 100) size = 100;
if (size < 10) size = 10;

// Get mesh offset
var nHalf = size / 2.0;

// Define vertices
var nVerts = (size + 1) * (size + 1);
dV = System.CreateDO("Space 3D Package/Vertex Stream Data");
dV.SetNumVertices(nVerts);
dV.BeginWrite();

// Define faces
var nFaces = size * size * 2;
dF = System.CreateDO("Space 3D Package/Triangle Vertices Stream
Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();

// Create verts
for (y = 0; y <= size; y++)
{
    for (x = 0; x <= size; x++)
    {
        dV.x(y * (size + 1) + x) = x - nHalf;
        dV.y(y * (size + 1) + x) = y - nHalf;
        dV.z(y * (size + 1) + x) = 0.01;
    }
}

// Create faces
var i = 0;
for (y = 0; y < size; y++)
{
    for (x = 0; x < size ; x++)
    {
        // Create first tri in grid square
        dF.i(i) = (y) * (size + 1) + (x);
        dF.j(i) = (y + 1) * (size + 1) + (x + 1);
        dF.k(i) = (y + 1) * (size + 1) + (x);
        // Create second tri in grid square
        dF.i(i + 1) = (y) * (size + 1) + (x);
        dF.j(i + 1) = (y) * (size + 1) + (x + 1);
        dF.k(i + 1) = (y + 1) * (size + 1) + (x + 1);
        i = i + 2;
    }
}
dV.EndWrite();
dF.EndWrite();

// Create the mesh and Output it
dM = System.CreateDO("Space 3D Package/Mesh Data");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);
params.conValue('outMesh') = dM;

```

```

    // Pass on Output values
    params.conValue('size_out') = size;
}

```

The script in MeshGen retrieves the value passed in for size and checks to make sure that it is between 10 and 100. Then it creates vertex and triangle stream objects (see Scripting reference for more information on these objects and how they work) and then adds the vertices and triangles that make up the mesh. When the information has been assembled these streams are attached to a new mesh data object and passed out to the outMesh connector. The size parameter is also passed on at this time through the size_out Output attribute.

Back out of the script we can see that the control signal is also passed on from the MeshGen Script Command object to the GenTerr Run Activity object, forcing that object to pass on the control signal to activate the script in the TerrGen Script Command object. You can see that this object has a number of attributes, which retrieve their values from sliders on the main object panel. Two attributes, inMesh and size, are retrieved from the MeshGen Script Command object.

When TerrGen receives the control signal the following script is executed.

```

mapData = null;

// Execute(params)
var mapData = null;
var size, numhills, hillmin, hillmax, isIsland, flattening;

// Execute(params)
function Execute(params)
{
    // obtain values of input connectors
    numhills = params.ConValue('numhills');
    size = params.ConValue('size');
    hillmin = params.ConValue('hillmin');
    hillmax = params.ConValue('hillmax');
    isIsland = params.ConValue('isIsland');
    flattening = params.ConValue('flattening');

    // Get input mesh and vertex stream
    dM = params.conValue('inMesh');
    nVert = dM.GetNumVertices();
    dV = dM.GetVertices();
    dV.BeginWrite();

    // Initialize map data array and generate terrain
    mapData = new Array(nVert);
    GenTerrain();

    // Modify z value of vertices based on map data array
    for( i = 0; i < nVert; i++)
    {
        dV.z(i) = 0.1 + mapData[i];
    }
}

```

```

    }

    // Reattach vertex stream and Output the mesh
    dV.EndWrite();
    dM.AttachVerticesStream(dV);
    params.ConValue('outMesh') = dM;
}

```

First off the script defines mapData (and a few others) at the global scope, i.e. outside of any function. This is so that the variable can be used throughout the script (see the <http://msdn2.microsoft.com/en-us/library/bzt2dkta.aspx> for more information about variable scope). When the Execute() function is called the script first obtains all of the data it will act on, including the mesh which was just created by MeshGen. It also uses a couple of the mesh object's methods to determine the number of vertices that are attached to that mesh and to get a copy of those vertices.

Next the script creates an array in the variable mapData, sized so that there is an entry for each vertex in the provided mesh. Then the script calls GenTerrain() to fill that array with terrain data (GenTerrain() and other terrain generation functions are not listed here but you can examine them by opening the Terrain System object and entering the TerrGen Script Command object.)

After generating the data, the script iterates through each vertex in the mesh and changes the height (z coordinate) to the value stored in the mapData array. Finally the new vertex stream is attached to the mesh and it is passed out as outMesh.

Back in the Link Editor the finished, modified mesh has a planar UV mapping applied and is then fed into a Shape object so that it can be rendered by trueSpace. Following all of these steps the Workspace window shows a nicely formed terrain, complete with rocky hillsides and rolling grassy plains.

You may have noticed that the description above did not mention the second button on the main object panel – Generate Terrain. When you press the Make/Remake Mesh button the object runs through the entire sequence above: creating a mesh, assigning data, and Outputting the mesh. When you press the Generate Terrain button the process skips over the mesh creation phase, by sending a control signal to GenTerr. This is so that a new mesh only needs to be created if you want to change the size of the mesh, not each time one of the terrain-specific parameters is changed.

Though it appears complex at first glance, the Terrain System object actually has a simple, understandable flow and can be used as a basic model for a number of different mesh creation tasks with a few simple modifications. Try changing some of the parameters or altering the algorithms used for mesh creation to see what other possibilities exist. The same basic setup could be used, for instance, to create a spherical mesh covered in craters or to create a city-like area covered with buildings. Your imagination is the only limit.

1.3 Script Objects

Script Objects in trueSpace

Script Objects are scripts that create *persistent* objects, reacting to changes by Input and Output parameters as they happen. You should use a Script Object when you want to have a script that regularly interacts with your scene or activity as the values that are provided to it change.

Creating a Script Object

You create a Script Object the same way you create a Script Command object, by dragging and dropping it from the System - Scripts Library into the Link Editor.

A Script Object is identified by the word “object” in the “Type” field of Script Editor/Attribute tab. For instance, if you wish to create a jScript Object, simply left-click on the item named “jScript object” in the System - Scripts Library, left-click-drag that object into the Link Editor.

As with Script Command objects you can place your Script Object at any level of hierarchy in the Link Editor but you will usually want to place it either within Space 3D or perhaps inside of another object, one with which you want the script to interact. Script Objects can be encapsulated within the Link Editor just like other objects

A JScript Script Object as it appears in the trueSpace Link Editor is shown below. Notice that there are not any items on its panel when the object is created. As you add Attributes to your Script Object additional connectors representing those Attributes will appear on the panel.



A JScript Script Object in the as it appears in the Link Editor

Inside the Script Object

As with a Script Command object you can access the script and attributes of your Script Object by clicking the orange Enter icon in the upper right-hand side of the Script Object in the Link Editor. Once you click the icon the Link Editor will show either the Methods window or the Attributes window.

You can easily switch between these two views by clicking the Methods and Attributes tabs at the top-left of the window. The Methods window is where you will write your script code and the Attributes window is where you create Input and Output attributes for your Script Object to use.

Writing Script Object Scripts

A trueSpace Script Object has one basic method that controls the way the script interacts with other objects in the Link Editor. The method, also called a handler because it handles requests for interaction, is `OnComputeOutputs(params)`. Below you will find a quick overview of this method. Please note that all of the samples in this section are written in JScript.

OnComputeOutputs(params)

This method is called whenever the value of any Output connector is requested by another object in the Link Editor. You can compute values of all Output connectors at once here using Input connector values. You have access to all connector values through the generic *params* parameter.

```
function OnComputeOutputs(params)
{
    var inValue1 = params.ConValue('inValue1');
    var inValue2 = params.ConValue('inValue2');

    params.ConValue('outValue1') = inValue1 + 1;
    params.ConValue('outValue2') = inValue2 + 1;
}
```

Our example Script Object has just two Output values, outValue1 and outValue2, showing that you can define the basic behavior of your object with just a few lines of code!

Advanced Handlers

For advanced users there are more handlers available to define more complex behavior. These advanced handlers include OnSetValue(params), OnDefaultValue(params), and OnCreate(params). Below you will find a quick overview of these methods. Please note that all of the samples in this section are written in JScript.

OnSetValue(params)

This method is called whenever the value of one of the Script Object's Input connectors is changed. You can use this method to react to changing values, perhaps by making sure that the value stays within a certain range. As with OnComputeOutputs() you can access two values from within this method, both provided through the generic params parameter. These are ConName and dataBlock. Below you see a basic implementation of this method.

```
function OnSetValue(params)
{
    conName = params.Param('ConName');
    dataBlock = params.Param('dataBlock');

    switch (conName)
    {
        case 'inValue1' :

            // perform some range checking
            if (dataBlock.ConValue('inValue1') > 10)
                dataBlock.ConValue('inValue1') = 10
            break;

        default :
            break;
    }
}
```

```

    }
}

```

In this example implementation we are checking only one Input attribute/connector, `inValue1`. If `ConName` is `inValue1` then the statements directly under case `'inValue1'` : will execute. In this case we check to see if the new value is greater than 10 and, if so, set it equal to 10 again.

OnDefaultValue(params)

This method is used to set the default values for specific connectors. It is called when the Script Object is first created or whenever the default value for a connector is needed. Two values are provided through the generic `params` parameter. These are `ConName` and `vtData`. A simple implementation of this method is shown below.

```

// OnDefaultValue
// Called to set default value for specific connector
function OnDefaultValue(params)
{
    conName = params.Param( 'ConName' );

    switch (ConName)
    {
        case 'height' :
            params.Param( 'vtData' ) = 3.0;
            break;
        case 'longitude' :
            params.Param( 'vtData' ) = 15;
            break;
        case 'radius' :
            params.Param( 'vtData' ) = 1.5;
            break;
    }
}

```

The parameter `ConName` provides the name of the connector for which a default value is being requested. In this case there are three connectors (`height`, `longitude`, and `radius`) and we have used `switch-case` statements to find out which connector is being requested. Then we set `vtData` equal to our chosen default value.

You only need to implement method in your Script Object if you wish your connectors to have a set of default values when the object is created. The implementation above, for example, provides the basic construction parameters for a cone object. When the Script Object is created `height` will be set to 3, `longitude` will be set to 15, and `radius` will be set to 1.5.

Note: you can also create default values for a complex data object connectors within the `OnDefaultValue` method. See this example of initializing Input color connector to red color value:

```

// OnDefaultValue
// Called to set default value for specific connector
function OnDefaultValue(params)

```



```

{
    conName = params.Param('ConName');
    switch (ConName)
    {
        // set default value for particular connectors like this, if desired:
        case inColor:
            inColor = System.CreateDO("Common Data Package/Color Data");
            inColor.SetRed8(255);
            inColor.SetGreen8(0);
            inColor.SetBlue8(0);
            params.Param('vtData') = inColor;
            break;
    }
}

```

OnCreate(params)

This method is called only once, when the Script Object is created and initialized. The main purpose of the method is to set dependencies for the objects' connectors (for example: to set which connector(s) will depend on another ones(s)).

You can access one value from within this method, provided through the generic params parameter. This is the connector's parameter, as shown in the default implementation of this method below.

```

function OnCreate(params)
{
    connectors = params.Param('connectors');

    // remove following code line to set dependencies manually,
    // e.g. connectors.SetDepend(inConName, outConName)

    connectors.SetDefaultDependency();
}

```

The last code line above (connectors.SetDefaultDependency) sets default dependency rules for all object's connectors – each Output connector will depend on each Input connector. This provide a standard behavior for your object – when you change value of some Input connector, you expect that all Outputs will get recomputed, using new Input.

This behavior is welcomed for overwhelming majority of objects. For some special objects you may need to define your own custom dependency rules – mostly to speed up computation. Imagine an object with many attributes where some Output doesn't depend on all Inputs, only on a subset of them. You can save a lot of computation time by reducing dependencies.

Example – we have an object with 2 Inputs 'a', 'b' and one Output connector 'c'. We will set 'c' Output to depend only from 'a' Input, so changing value of the 'b' doesn't cause the Output get recomputed.

```

function OnCreate(params)
{
    connectors = params.Param('connectors');
    connectors.SetDepend('a', 'c')
}

```

Other advanced handlers

We append quick description for remaining advanced handlers here.

OnInvalidate

Called each time the particular connector's value get invalidated (e.g. due to value change). ConName parameter is name of the invalidated connector.

OnCustomEvent

For some special situations you may need to define custom communication between your objects.

Use RsApp.SendCustomEvent method to send an event and implement OnCustomEvent to handle the event.

OnPostLoad

Called right after object gets de-serialized (loaded).

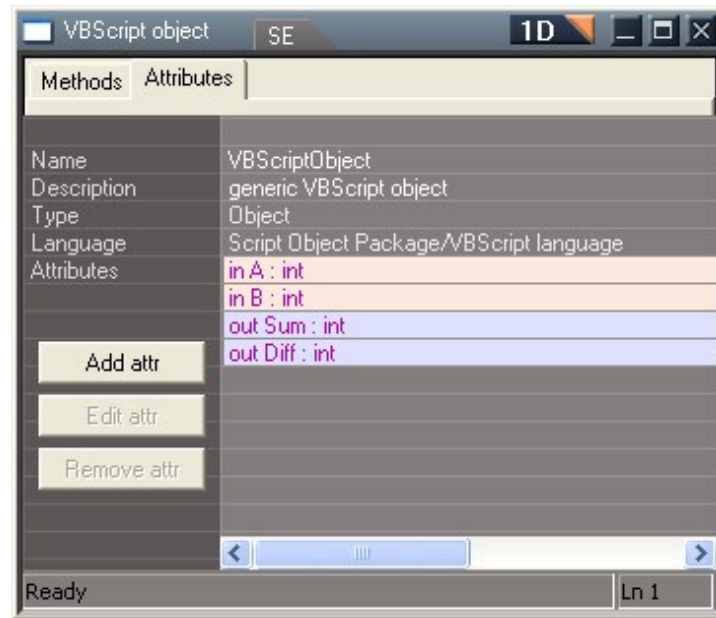
OnSharedSpace_NodeStatusChanged **OnSharedSpace_SpaceStatusChanged**

Shared space advanced event handlers, allowing you e.g. to detect connection lost.

Tutorial: Creating a Simple Math Object

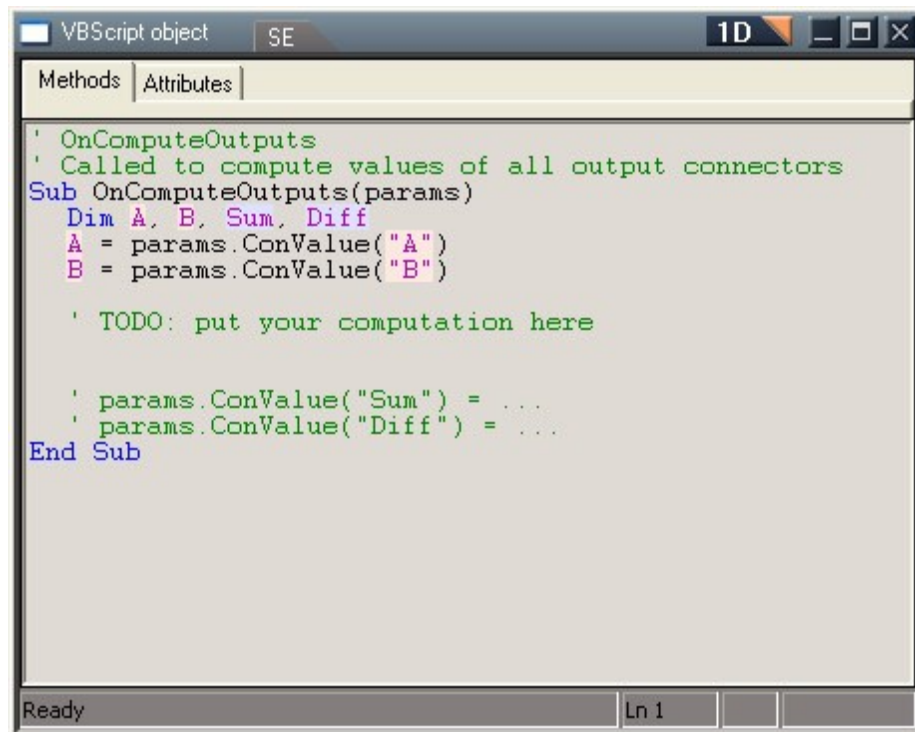
In this tutorial we will create very simple object to perform a basic calculation with only two Inputs and two Outputs. Though this script object is as simple as possible you should be able to see how this example can be applied to much more useful objects. Please note that this example is written in VBScript.

First open the System - Scripts library and drag a VBScript Object into the Link Editor. Then enter the object by clicking the orange button on its panel to open the Script Editor. Click the Add Attr button and define two Input attributes (A and B) as integers and two Output attributes Sum and Diff as integers.



The math object will need four attributes

Now switch to the Methods tab and take a look at the script generated for you by trueSpace. You should always try to define all of an object's attributes first before beginning to work on the script to take advantage of the built-in script generator. You can, of course, always add attributes later but it is easier to let trueSpace write a skeleton script for you.

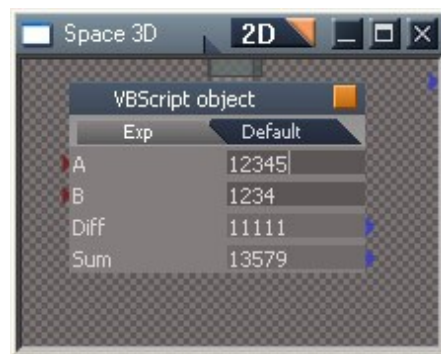


The math object script generated after you add attributes

Now, manually complete the OnComputeOutputs() method to figure the sum and difference of the values at attributes A and B, with these two lines. Hint, if you remove the apostrophe (') before the lines you will only have to replace the ellipses with 'A + B' and 'A - B'

```
params.ConValue("Sum") = A + B  
params.ConValue("Diff") = A - B
```

That's all you need to do to define the Script Object. Now, press the orange exit triangle to return to the Link Editor. Switch the object to the Exp aspect and then change the values of A and B to see the results.

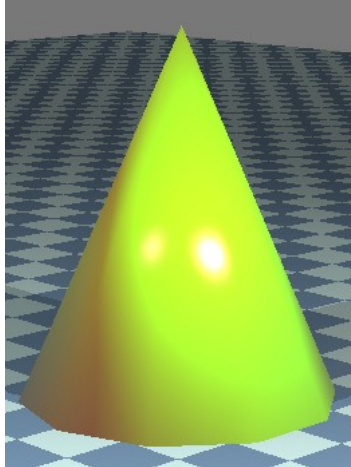


The final math object in the Link Editor

Tutorial: Examining a Simple Script Object

For this tutorial we will take a look at one of the more basic Script Objects, the ConeObj. This object creates a simple cone and provides front panel access to attributes including longitude, height, and radius using slider controls.

The ConeObj can be found in the Objects - Script Objects library. Simply drag the item into Workspace or into the Link Editor. When completed you should see a cone in the Workspace view, as shown in the image below.



The Output of ConeObj as seen in the Workspace view

In the Link Editor the ConeObj panel will appear as in the picture below. Try moving the sliders to see the effect they have on the shape and geometry of the cone in the Workspace window.



ConeObj as seen in the Link Editor

Now, click the orange button at the upper right of the ConeObj's panel window in the Link Editor to enter the object. The ConeObj contains a Script Object called ConeMesh, along with a transform, a material, and a Render Attributes object as shown in the image below.



The interior of ConeObj as seen in the Link Editor

You can see that the ConeMesh script exports three attributes (height, longitude, and radius), and these are the values affected by the sliders on the main object panel. It also exports a mesh to the Shape object so that trueSpace can render the cone mesh.

Entering the ConeMesh script reveals the script, containing implementations of two basic methods, or handlers. The important method for this object is OnComputeOutputs(), shown below:

```
function OnComputeOutputs(params)
{
    // obtain input params
    longitude = params.conValue('longitude');
    radius = params.conValue('radius');
    height = params.conValue('height');

    nVert = longitude + 2;
    nFaces = 2 * longitude;

    // define vertices
    dV = System.CreateDO("Space 3D Package/Vertex Stream Data");
    dV.SetNumVertices(nVert);

    // ... add vertices for bottom circle
    angle = 0.0;
    angleStep = 2.0 * Math.PI / longitude;

    for (i = 0; i < longitude; i++)
    {
        dV.x(i) = Math.cos(angle) * radius;
        dV.y(i) = Math.sin(angle) * radius;
        dV.z(i) = 0;

        angle += angleStep ;
    }

    // ... add top and bottom vertices
    dV.x(longitude) = 0;
    dV.y(longitude) = 0;
    dV.z(longitude) = height;

    dV.x(longitude + 1) = 0;
    dV.y(longitude + 1) = 0;
    dV.z(longitude + 1) = 0;

    // define faces
    dF = System.CreateDO("Space 3D Package/Triangle Vertices Stream Data");
    dF.SetNumTripleIndices(nFaces);

    // ... define bottom faces
    for (i = 0; i < longitude; i++)
    {
        dF.i(i) = longitude + 1;
        dF.j(i) = (i + 1) % longitude;
        dF.k(i) = i;
    }
}
```



```

// ... define upper faces
for (i = 0; i < longitude; i++)
{
    dF.i(i+longitude) = longitude;
    dF.j(i+longitude) = i;
    dF.k(i+longitude) = (i + 1) % longitude;
}

// create mesh
dM = System.CreateDO("Space 3D Package/Mesh Data");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);

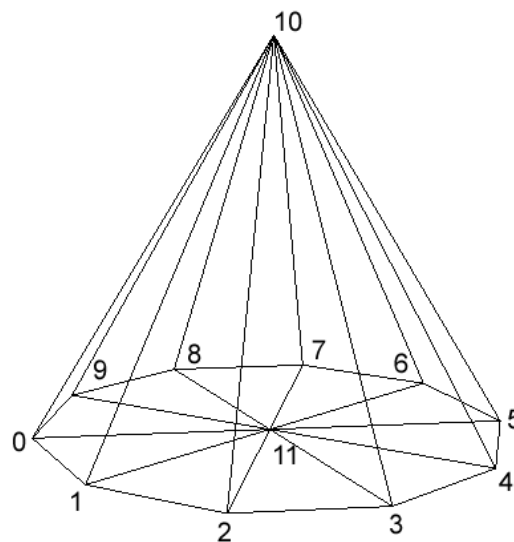
params.conValue('outMesh') = dM;
}

```

While this script may look complex it is actually very straightforward when reviewed piece by piece. Essentially, it reacts to a request for an updated mesh on the outMesh attribute. Once this happens, the script obtains the Input values for longitude, radius, and height and uses those values to construct a mesh representing the cone.

To do this, we must first determine the number of vertices and faces in the mesh. The number of vertices equals the longitude (or number of divisions around the circumference of the cone) plus two (one for the top of the cone and one for the bottom center of the cone). The number of faces is equal to two times the longitude (there are an equal number of faces on the bottom of the cone and along the circumference, between the base and the top).

Then, using a RDVertexStream object, create the vertices for the bottom, using trigonometric functions to place vertices along the circumference of the cone. Then it creates a vertex at the bottom, center of the cone and another at the top-center, located at the height of the cone. The following image shows the order of vertex creation for a 10-sided cone mesh.



Creating vertices and faces to make up the cone mesh

Next, the script defines the faces for the cone; using an RDTriangleStream data object type. First it defines the bottom of the cone, followed by the upper vertical faces. Faces are specified by identifying the indices of three vertices to be used as vertices for the triangle in counter-clockwise order. So, the first face would use vertices 11, 1, and 0, in that order. The second would use 11, 2, and 1.

Finally, the script creates a RDMesh object and attaches the vertices and triangles to it and then Outputs that to the attribute outMesh. Each time a slider on the outer panel is changed the object is requested to update the value of outMesh, causing the script to execute.

You can use this basic structure to create almost any sort of geometry you can think of. For instance, with slightly different formulae you could create a sphere object, or a geosphere.

Advanced implementation notes:

As for Output Mesh attribute that the mesh generator scripts produce, you have 2 possibilities here (both used in existing demos in libraries) – either you define Output mesh connector as registered Mesh attribute (you have to check “Registered” checkbox in the attributes edit dialog), or use simple Mesh attribute – in the later case you have to add “Shape” component to your final object to allow tS-renderers to recognize the mesh. (Refer to the Cone object and Gear object in the Objects - Script objects library to compare both approaches).

1.4 New in trueSpace 7.6

Set of array data objects

There are 5 new array data objects, defined in Common Data Package:

- Universal Array Data
- Int Array Data
- Number (float) Array Data
- String Array Data
- Boolean Array Data

All arrays can be single-dimensional or multidimensional.

The Universal Array can store items of any arbitrary Rosetta-allowed type (i.e. base types like integers, numbers, strings, as well as data objects like Point Data etc).

While the Universal array can handle most everything, for arrays of simple types it is better to use a specialized array, e.g. Int Array Data for integers, or String Array for text strings. Because these simpler array types are predefined to handle specific types of data, they are much more efficient than using the Universal Array Data type. You would want to use the Universal Array Data type when you are mixing data, for instance if you have a series of data that has someone’s name, age, address, height etc, you have a need for an array type that will hold the various types of data. Universal Array is best in such a case.

Single dimensional array example:

```
// Execute
// Called to execute the command
function Execute(params)
{
    a = System.CreateDO("Common Data Package/Universal Array Data");
    a.SetDim(1); // set dimensions for array
    a.Add(1.5);
    a.Add(7);
    a.Add("hallo");
    a.InsertAt(0, 55);
    System.Alert("array size is " + a.GetSize());
}
```

Multi-dimensional array example:

```
// Execute
// Called to execute the command
function Execute(params)
{
    a = System.CreateDO("Common Data Package/Number Array Data");
    a.SetDim(2); // set dimensions for array
    a.SetSize(10, 10);
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 10; j++)
        {
            a.SetAt(i, j, i + j);
        }
    }
    System.Alert("array size is " + a.GetSize());
}
```

Note: methods such as Add, InsertAt, RemoveAt will only work for single-dimensional arrays.

See the “WormEater” demo in Objects – Tutorial Object library, or “Magic Sculpturer” demo in Objects – Script Objects library for real –life examples of arrays in use.

OnGetValues handler can be used instead of OnComputeValues

OnGetValue is part of advanced handlers for script objects. Be sure to delete the OnComputeValues function if you want to use OnGetValue (they are mutually exclusive), which means do not use them together in the same script!

Script caching

Internal script execution model has been changed, allowing you to cache scripts that have already been parsed.

There are several consequences and benefits of the new-cached model:

- Better encapsulation of each script object, in particular script execution now doesn't depend on previously executed scripts (i.e. random state of script engine).
- Easier persistent variables (see below).
- Possibility to perform nested script command calls (see below).
- Possibility, to create script function sets (see below).
- Script execution enjoys a performance increase.

Persistent variables support

Script command / object often needs to remember it's internal state somehow. E.g. in PacMan game, Mover script command needs to remember current position and direction of the player in the game loop. Usually, these state variables can be stored in connectors – this is the most safe and recommended way.

But there are situations where this model is not possible or is not effective (elegant), like

- Your script command uses variable of complex type that cannot be stored into connector (e.g. Dictionary data type used in Game 15)
- Your script defines too many internal state variables so storing in connectors is not effective – etc

The simplest way to define persistent variables for script command is:

- Define variable in your script command, outside of the Execute method (e.g. at the beginning of the script):
 - Then you can read or modify the variable in your Execute method, and the system will keep its value between subsequent Execute method calls, just like if the value is stored in the connector.

Example:

```
// this variable is persistent when used from Execute calls
var A = 1;

// Execute
// Called to execute the command
function Execute(params)
{
    A += 1;
    System.Alert( System.ThisName() + " has incremented value A = " + A );
}
```

Important note: the described persistent technique only works for script commands Execute method. To create persistent variables accessible for handlers other than Execute, and for script objects as well, global variables can help here (see below)

Global variables support

Globals are variables, which are persistent within one object / command and can be used to create more complex scripts. Globals can be useful for remembering events like OnInvalidate, OnPostLoad., or to keep

pre-computed data to speed up next computation. You can also store some status variables in globals instead of connectors.

Globals serve for the exact same purpose as Persistent variables described above, but are more universal (works not only in Execute handler but for all script handlers like OnDefaultValue, OnComputeOutputs, OnInvalidate etc). Drawback is couple of extra script lines that need to be used:

Code samples with globals:

```
// set global variable in any script handler
params.Global("invalidated") = true;

// obtain value of previously set global
Num = params.Global("precomputed_vertices_count");

// increment global counter and ensure initialization
if (!params.IsGlobal("counter"))
    params.Global("counter") = 0;
params.Global("counter") += 1;

// erase existing global
Params.DeleteGlobal("counter");

// code snippet - Execute handler sample
function Execute(params)
{
    if (!params.IsGlobal("A"))
        params.Global("A") = 1;
    params.Global("A") += 1;

    System.Alert( System.ThisName() + " has incremented value A = " +
params.Global("A") );
}
```

Script function sets

Script function sets are a new (advanced) type of script, introduced in trueSpace7.5.

A Function set contains a set of functions or variables, which are logically grouped into one container. Unlike script commands or objects, function sets cannot contain any connectors, and cannot be 'executed' directly - so in LE they appear lifeless. Only when they are accessed from another script command will they reveal their power.

Use Node.AccessFnSet, or Node.AccessNearFnSet methods to access and use the fn set from your script command.

Benefits:

- If function set is included in the project (scene), all commands in the project can access it easily (see both commands in the demo).
- General-purpose function sets could be defined and shared between users (e.g. extended math fn set, matrix arithmetic fn set, scene traversing fn set etc).

- Fn set can contain shared piece of code that is used frequently in the particular project. They can speedup development time for larger scripting projects

How to define new script fn set:

- Open System – Scripts library.
- Insert jScript Function Set or VBScript Function Set to the Link Editor.
- Open the object and define script.
- Then define at least one script command that access and uses your function set features.

Example:

```
// This is a jScript Function Set
// Put useful functions, constants, enumerators here and share them
// across your project
// Use Node.AccessFnSet / Node.AccessNearFnSet to get access here from
// your script commands
// Hint: do not use commands like System.ThisOwner, System.ThisName or
// Node.NearValue here, as they refer to caller's command location

// ==== Extended Math Demo Function Set ====

function isPrimeNumber ( a )
{
    for ( i = 2; i < a; i++)
    {
        if ( a % i == 0 )
            return false;
    }
    return true;
}

function Factorial ( a )
{
    if ( a <= 1 )
        return 1;

    var aa = a - 1
    var f = a * Factorial ( aa );
    return f;
}
```

Above-mentioned code snippet defines a sample function set for extended math operations with two functions, IsPrimeNumber and Factorial. Any script command can use the fn set, this is how to do it:

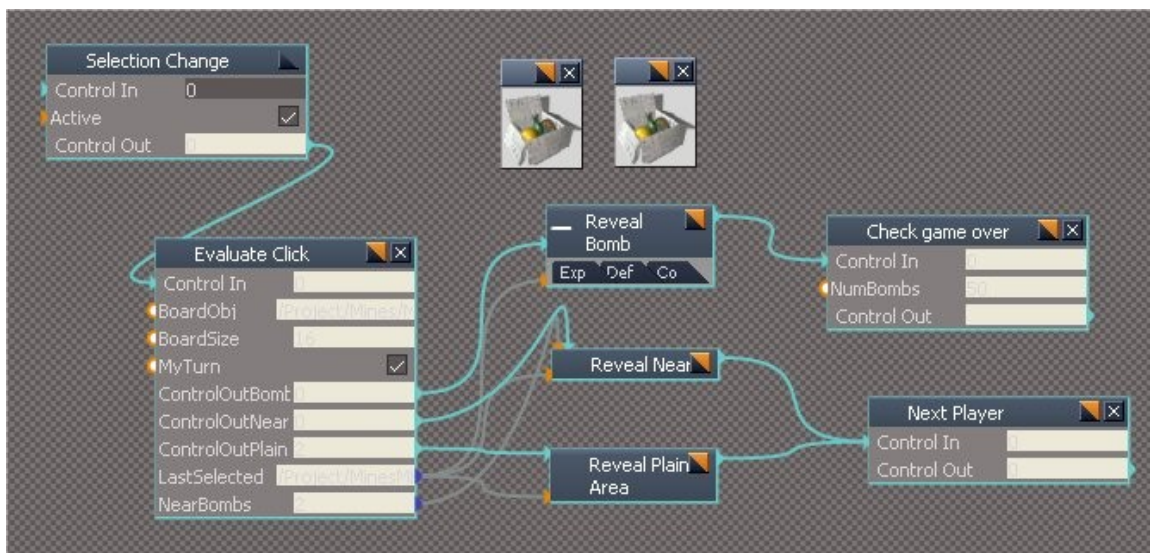
```
fn = Node.AccessNearFnSet( "ExtMath" );
isPrime = fn.isPrimeNumber( a );
```


Note: script error reporting in a script that calls a function set will most likely be referring to the function set. When an error does occur, write down the error line/code reference and check the function set at the specified line number. The error will refer to the script command object that is calling the function set and reference it rather than the function set. It does this because the function set was called from that command script.

Nested script command calls

When creating a larger script project, it usually consists of many script commands – activities, connected with green lines to form the main project execution sequence. One can do many things using this basic model – form loops, forks, decision branches etc. Still, in real life projects, situations can arise when you need to extend this basic schema.

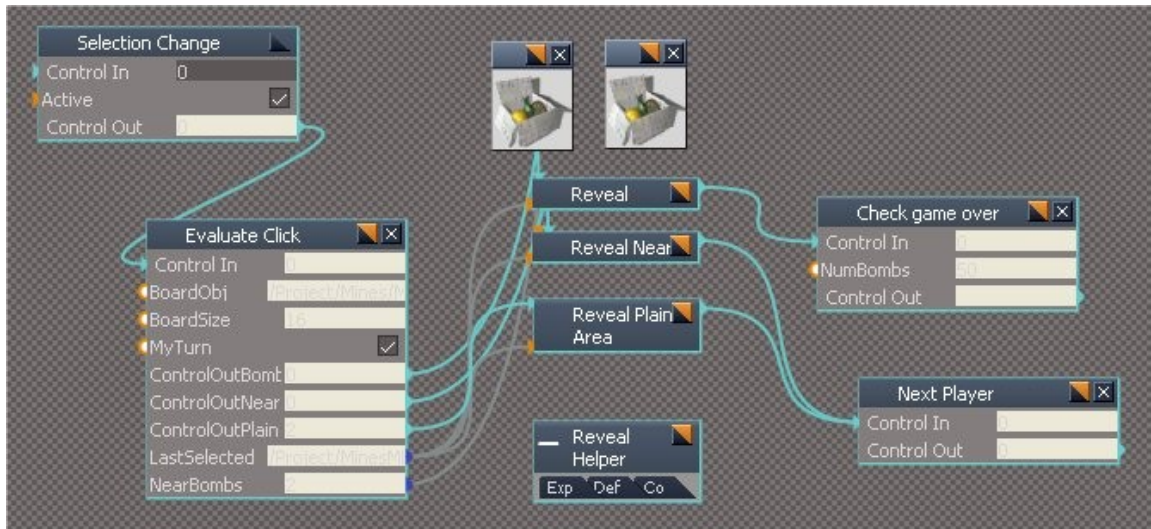
Imagine a command that should be used (called) from different places in the project. In such a scenario, you would not be statically linking with green activity lines. One solution is to create several identical instances of the command and link them when needed – but this leads to duplicate code that can cause problem with further project maintenance and flexibility.



Mines game inner workings

In the above picture (taken from Mines game), there is a “Reveal Near” command, which is used as an alternative to “Reveal Plain Area”. In fact, Reveal plain area procedure needs to use the Reveal Near several times as a part of its processing.

Solution could be to separate shared functionality required by both commands to third command – Reveal Helper – and use this command dynamically from both above mentioned command objects.



Setting up the Reveal Helper

Call a script command from another script command.

- Fill the called command's Input connectors (parameters) dynamically using `Node.Value()` rather than using connection wires.
- Execute the command – using `Activity.Run` or `ScriptObject.Execute`
- Important difference is that `Activity.Run` is asynchronous (delayed) call – it is being executed after the current command finishes. It is correct and a safe way, however the main problem is that you cannot check or use the result of this call in your command (i.e. read the Output connectors of the command).
- You can use the `ScriptObject.Execute`, to do the immediate nested command execute. This allows you to read and use the nested call results like this:

```
var helperCmd = System.ThisOwner() + "/Reveal Helper";
// prepare inputs
Node.Value(helperCmd, 'x') = x;
Node.Value(helperCmd, 'y') = y;
// execute the command
ScriptObject.Execute(helperCmd);
// check the results
revealed = Node.Value(helperCmd, 'revealed');
```

Other notable new and fixed

- `Node.NearValue (obj, con)` method (shorter form of `Node.Value (System.ThisOwner() + "/" + obj, co)`)
- `Node.LookupParentValue (con)` - search parents for given connector value
- `Node.AccessFnSet` - see function sets chapter
- `Node.AccessNearFnSet` - uses relative name of the fn set instead of full name
- Fixed occasional nonsense error messages on script changes commit
- Fixed `OnDefaultValue` handler for script commands

