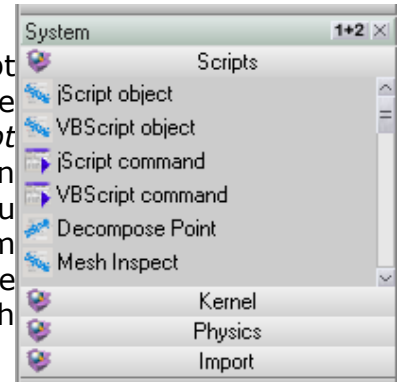


Creating a script object

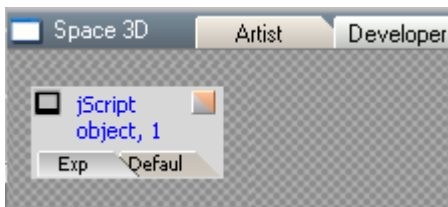
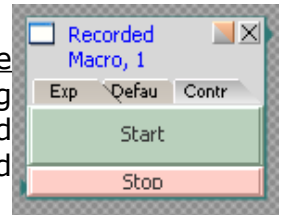
Emma @ 3d-ts-forum / Caligariforum November 2006

M.Moersner

A script object is identified by the word "*object*" on the script editor attributes tab, in the line where it tells you which language is used for scripting. In trueSpace 7 you can code with *VBscript* but also with *jScript*. You can get an empty brick for your own script from the System Library. Depending on the language you want to use you simply drag the corresponding block from System/Scripts into the LinkEditor (LE). You now have the basics to write your own object script that can communicate with trueSpace 7.



The difference between a *script object* and a *script command* should be explained here in short only. With help of the build in macro recording function you can let create a script by trueSpace itself. This will be build with a **Start** and a **Stop** button so the script can be started and stopped manually.

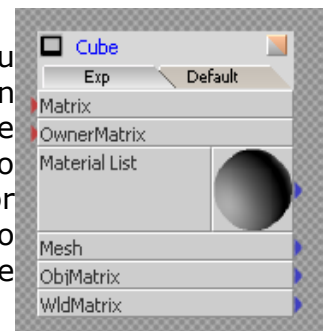


In opposite to a *script command* a *script object* is controlled by an activity. So something must be passed on to the script to initiate it so it starts working. for this we have what is called *attributes* which are passed over to the script through *connectors*. The picture left sided shows a new *jscript* brick that just was draged into the LE. Of course there are in this

moment no *attributes* available that can comunicate through *connectors* to the outside world.

Before writing the script now there should be at least a few thoughts about what has to be done with it and which *attributes* will be necessary to be connected to the outside world of that script.

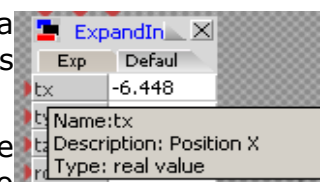
What is an attribute? Well, this can be the name of a cube that you wan't to create, it can be the length of one of the cube edges, or it can be x, y, or z coordinate of the upper right cube corner. It simply can be everything that you need to define anything of that cube in trueSpace or for example anything you need in an animation to manipulate that cube. In the LE a new created cube would look like shown in the right picture.I



Basically an object is displayd in the LE this way so the LE looks clearly arranged. the detailed definitions of the cube are hidden behind the *Matrix*. If you click with your right mouse button on it and select *Expand* you get additionally the left sided picture opened which shows you all the details.

One more remark to the *connectors*, they practically connect the inside of the brick with the outside world to transport information in or out. You can see on the outline of the bricks small red or blue unions. Red means this connector transports information into the brick, blue means that connector translorsts information to the outside world. If you point for a moment with the mouse pointer on such a union there will open a small pop up window giving you some details about that connector.

The picture to the right shows what came up when I pointed on the red connector for tx. The information is quite helpful if you have no idea what kind of data ther is and which formatting is in use for the data transported there.



Data formats can be numbers, characters, logicals (bool) YES/NO or very complex data structures as there are for example behind the different Matrix *connectors* (*Matrix*, *ObjMatrix*, *WldMatrix*). Just a remark to numbers, they can be specified much more in detail as there are integer, floating point and so on.

Then we have arrays that can be seen as kind of a table, either a single row with several cells or a number of row divided into columns. Each cell can be addressed through an index. Contents of these tables can be numbers and characters.

To give a simple example for creating a script I'll take the math example from the developer guide. the multiplication tables will be known by everyone and so understanding of the script will be easier. We will extend though the script a little bit to get a basics calculator à la trueSpace handling the first rules of arithmetics.

Step 1, Define what we want to do:

- we want two values A and B as input
- we want to show the result
- we want to say which first rules of arithmetics + - / * shall be calculated

Step 2, determine which attributes we will need or have :

- wertA
- wertB
- Ergebnis (result)
- Rechenzeichen (arithmetics sign)

Should there be later any need for more attributes, then it is no problem to add them afterwards. The advantage of defining them ahead ahead is, the LE will already prepare necessary definitions in the script so that we don't have to type them in manually. Of course a good programmer always defines with his mind already ahead in theory what he wants to do which way in a program. Good preparation saves lots of debugging time !

So that script objects in the LE can communicate or interact to each other, trueSpace uses a fundamental method which is also named as handler. Handling solves requests of interactions . In *jscript* you name them also *methods* which are kind of functions written in a program code that does all the work. *OnComputeOutputs* (*params*) is one method behind, that helps moving the contents, equals data, of attributes through the *connectors*. The direction is assigned by the position of the equals sign (=) which is then reflected by the blue or red union on the bricks outside.

```
function OnComputeOutputs(params)
{
  // get the numerical values of A and B
  wertA = params.ConValue ('wertA') ;
  wertB = params.ConValue ('wertB') ;

  // calculate the result
  Ergebnis = wertA + wertB ;

  // show the result
  params.ConValue ('Ergebnis') = Ergebnis ;
}
```

What happens now exactly in this script? *OnComputeOutputs* already shows that attributes are to be moved. The following code line

```
    wertA = params.ConValue ('wertA') ;
```

transports through a connector (*Con*) *wertA* the data (*Value*) to the attribute *wertA* inside the script. Inside the script we rather use variable instead of attribute. The opposite operation is realized by the following code line

```
    params.ConValue ('Ergebnis') = Ergebnis ;
```

So how should we interpret and understand this ? It is really simple, the expression *params.ConValue* ('*Ergebnis*') represents virtually the connector '*Ergebnis*'. So if I say for example *connector = pipifax* then the content of *pipifax* will be moved through the *connector*. If I say *pipifax = connector* then the contents of the *connector* will be moved to the variable *pipifax*. So always the value or content is moved from the right side of the = sign to the left side. A remark here, please notice that in *jscript* each line is closed up with the ; sign.

So now let's put the real stuff into the *jscript* brick waiting in the LE for us. If you didn't already do so just drag and drop the *jscript* brick from System/library into the LE. Well, looks quit unspectacular so far. In the headlines we can read *jscript* which shows that this brick just is able to understand *jscript* code that we will enter. In the next line we find "object". This is typically the way that trueSpace does naming and also typically, if we would drag another such brick into the LE the name would be "jscript object,1", then "jscript object,2" and so on. To give another name we just have to do a left mouse click on the upper left black square, select *Rename* and enter what we want it to be called.

Now we reach the exciting moment, left mouse click on the orange triangle of the *jscript* brick and it will open. The first time done this will always enter the *Attributes* tab. Every open afterwards will start with the *Methods* tab opened. As already mentioned above we will find the expression *object* in the *Language* description. That points to the fact that this is a *script object*, not a *command*, and that trueSpace will now automatically do during execution time all corresponding and necessary handlings between our script and trueSpace silently in the background. Left mouse click on *Add attr* and a window opens so we can enter our first *attribute*.

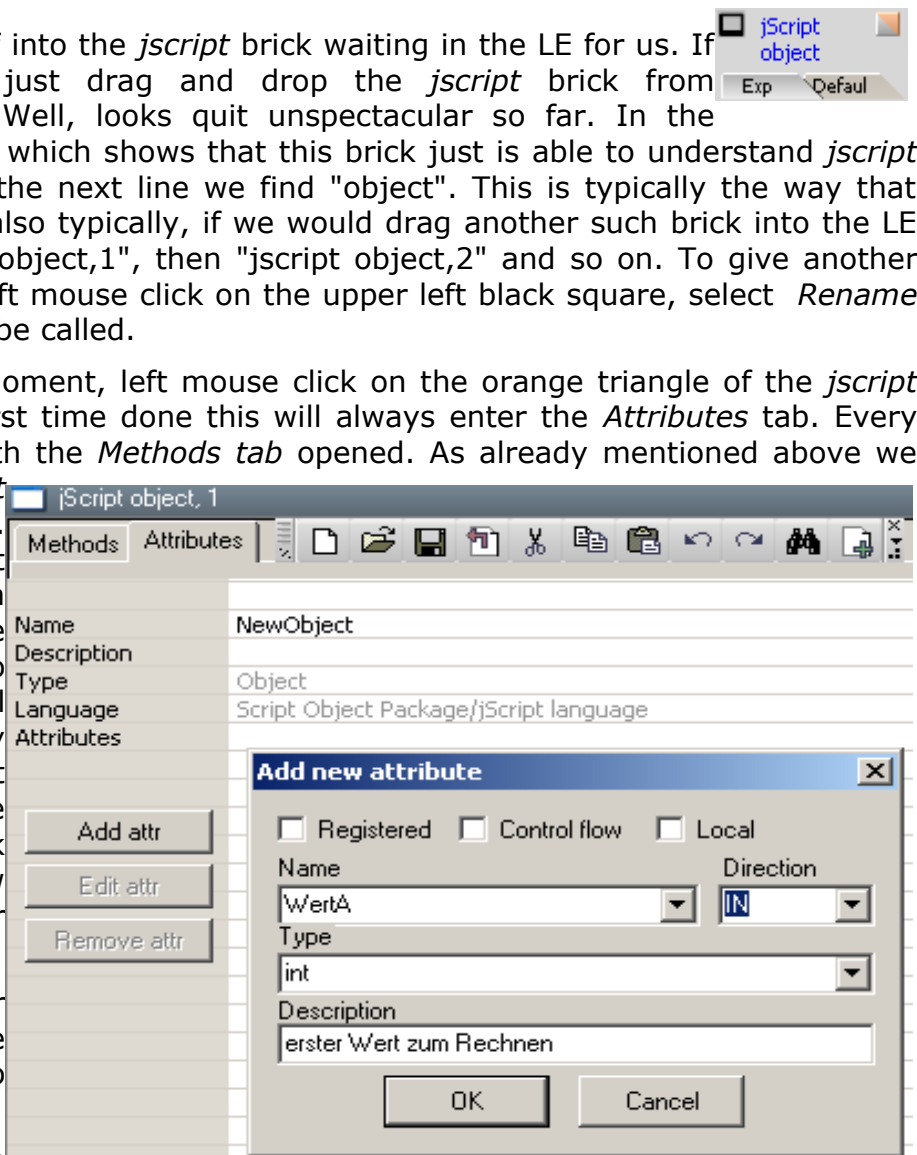
For *Name* we now enter **WertA**. This is an entry value so *Direction* must be set to **IN**.

Next one to fill out is *Type* where we select **INT**. That means only whole numbers can be entered, no decimal sign is allowed. In the field *Description* we should now write some explanations which tell us what the defined attribute is staying for. Here it says **first value for calculation** (erster Wert zum Rechnen). So now after clicking on *OK* we can then enter, after clicking *Add attr*, the settings for **WertB**, the second calculation value with the same settings.

Now follows **Ergebnis** (result) where we must set *Direction* to **OUT** because this is passed to the outside. As *Typ* we take **number** for possible decimal point if calculating divisions.

Leaves at the end the attribut **Rechenzeichen** (calculating sign). Will be entered so here we need **IN** again. It is a character, not a number so we have to select for *Typ* the value **string**.

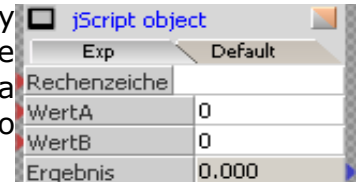
This will finish the setup of our attributes that we already had defined ahead. We now can take a look what the LE created from the data we have given him so far. For this we click on the orange triangle in the upper right of the LE window. this way we return from the **script editor modus** back into our



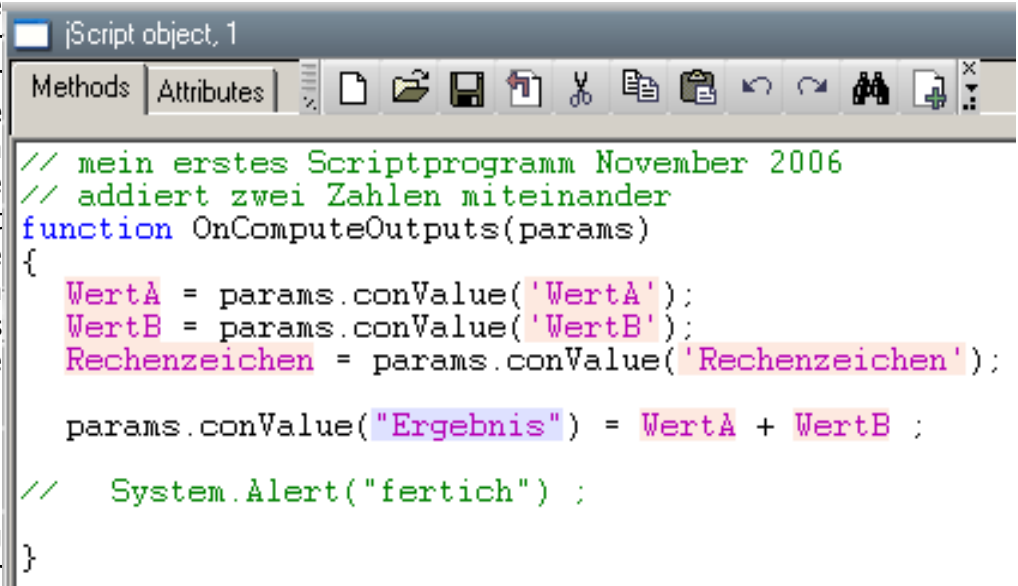
Script Object Package/jScript language
in WertA : int 'erster Wert zum Rechnen'
in WertB : int 'zweiter Wert zum Rechnen'
out Ergebnis : number 'errechnetes Ergebnis'
in Rechenzeichen : string 'Angabe Rechenoperation + - * /'



normal LE window. There we find now our *jscript* brick and by clicking on the **Exp** Tab it should open and look like shown in the right picture. Just by typing in our settings we already got a preprepared brick. Ok, so all we need now is type in the coding so that calculations will be done as we demand them.



So one click again on the orange triangle of the *jscript* brick. From now on we will always enter the Methods tab where program code is entered, Any more attributes necessary, well just click on the **Attributes** tab to enter them. The editor window contains some predefined lines which we will change to the code as you can see it in the window on the right. The explanation what the code means will be explained in the following lines now.



From beginning on you should start using comments in your programs. As more you will write as less you will remember later on what you did, especially as you will change your way of coding the more experienced you will become. Never forget, *there are always several ways leading towards Rome*. Comments should give function explanation, date, author, specifics to be concerned. Comments start with `//` You can also add them behind a command, so you don't need necessarily an extra line for them.

The line `function OnComputeOutputs(params)` takes care that all functions needed now will be available for the program and understood by trueSpace. So we can use code sending calls to trueSpace that transport our parameters through the connectors in either direction and data flow from or to other bricks in the LE will work successfully. There are lots of other ones like for example `function OnDefaultValues(params)` which is almost selfexplanatory as it simply presets the default values for attributes/variables. Without such funtions (or handlers) there would be quite a lot of real complicated program code necessary to do even basic things with trueSpace.

Now follows a `{` character marking the beginning of der script code block. Contained in this block are script commands up to function calls which can stay in their own code block or even be outside somewhere kind of hard coded in trueSpace. The end of the block is shown by the `}` sign.

The first three lines of program code just transport the attribute values through the connectors into our script. Remember we said transportation happens from right towards left. On the forth line the calculation `WertA + WertB` is done and then the result is transported towards the left side where we have the attribute `"Ergebnis"` which is led to the outside world through a connector. Notice that all attributes (or vaiables) that go to a connector have a background colour, pink for **IN** and blue violet for **OUT**. i

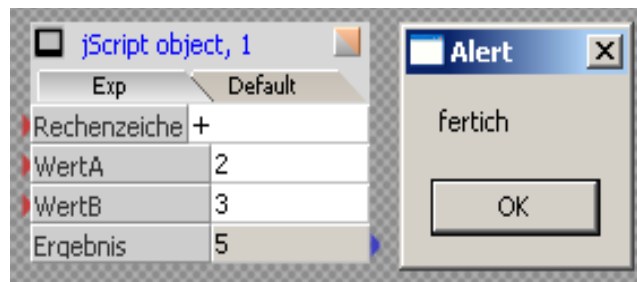
A nice specialty is the command `System.Alert`, which simply prints out a character string in this example `"fertig"` (done). For testing it shows where the script is working at that moment, helpful for long scripts.

Now we should find out if there are any errors in the code syntax. For this we click in the LE menue line on the white square with the red arrow right of the



diskette symbol. If everything is OK nothing changes besides a small message at the bottom saying checked. Is there an error, a window at the bottom pops up with orange background and an error message. The cursor is placed then on the beginning of the code line containing the error. Just correct the error and click on check again.

This check has to be done for every change in code ! You can of course do several changings before clicking the check. As soon as there is no error anymore we can get this



little script on to work now.. For this we simply click on the orange triangle in the upper right of our script brick. The *jscript* brick appears now immediately in the LE

Did you delete the comment signs *//* for the *SystemAlert* code line you will get immediately now the alert as shown above. This shows us that the script is really executed immediately after we closed the editor. Remember what was said above that a *script object* is handled different than a *script command*. An activity is any kind of "happening" this includes of course also change of momentarily status. Jumping back from the script editor into the LE means the LE has taken over control now for our *jscript* brick, thus a change of a status !

One click on OK and we can change the values for A and B and see immediately how the result field changes. Simply entering a value in **WertA**, well nothing happens. As soon as we click now on field **WertB** or hit the return key the value will be taken over by the script. This goes conform to the way you will be familiar when working with entry fields when working in the modeling mode. This has always been a basic functionality of trueSpace in the moment you type you are active, hitting return or changing to other field trueSpace starts activity as you "signaled" you were done. This is now an activity an object needs and so the values are transported through connectors in their corresponding direction.

The **Rechenzeichen** doesn't have any effect till now because there is no script code so that we will have to add this now. Doing this we will learn a new script command but before starting to program we remember, think about what you want to let the program do:

- take over the arithmetics sign (*Rechenzeichen*)
- if **Rechenzeichen** + then add **WertA** to **WertB**
- if **Rechenzeichen** - then subtract **WertB** to **WertA**
- if **Rechenzeichen** * then multiply **WertA** with **WertB**
- if **Rechenzeichen** / divide **WertA** by **WertB**

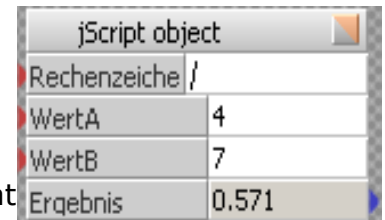
OK, getting **Rechenzeichen** and taking it over already works since the connector to the outside world for this attribute already exists. But first we must check the "if" which is done with the **if** command and leads to what shall be done. Syntax looks as follows:

```
if (Rechenzeichen == "+")
```

The double equal sign (==) stays for **is identical**. A *string* (or text) must always be enclosed inside " ". Practically the value or better described content of **Rechenzeichen** is compared with the plus sign for equality. Other possibilities would be unequal != and for numerical values additionally larger > , smaller < , larger or equal >= , smaller or equal <= .

So the **if** command already opens a large variation of different conditions that can be checked and solved. So missing now is only the output of the corresponding calculation done. This happens with the corresponding assignment: *params.con... = ...formula...*

```
// calculating result on condition of corresponding arithmetics sign(Rechenzeichen)
if (Rechenzeichen == "+") { params.conValue("Ergebnis") = WertA + WertB; }
if (Rechenzeichen == "-") { params.conValue("Ergebnis") = WertA - WertB; }
if (Rechenzeichen == "*") { params.conValue("Ergebnis") = WertA * WertB; }
if (Rechenzeichen == "/")
{
    params.conValue("Ergebnis") = WertA / WertB;
    System.Alert("this was a /"); // if division then alert
}
```



jScript object	
Rechenzeichen	/
WertA	4
WertB	7
Ergebnis	0.571

We divide the **if** command into the condition and the code that has to be executed if condition is true which is enclosed in **{ }**

For the division I change the coding a little bit to give an example for several code lines and also for adding a comment directly in a code line. This should also show that you insert tab fields if you have several code lines in a code block. This gives a structure to the layout of a program which helps recognizing those blocks much easier, very important for debugging situations. All lines included inside the **{ }** signs belong to the same code block.

--< * >--

It is time for an aggregational review for this lead into script programming. So far we should have learned that you can create a **Scriptobjekt** by getting the correct brick from *System Library*. In our example it was for **jScript**, alternatively **VBScript** would have been possible (we will talk about this later).

We now should know that such a brick can transport data in or out with support of connectors, to or from the scripting code. Attributes can be numeric, strings or as we will see later even high complex structures as you will find in Mesh, Matrix and so on.

Then we have done planning of a script ahead, before starting to code. Next step was creating the predefined or preplanned attributes in our *jScript* brick. This was done on the *Attr Tab* with correct definition of *Type* and *Direction* (IN/OUT).

Finally first masterpiece, creating the program code. Since we only wanted some values to move as input or output this all happened inside the function `OnComputeOutputs(params)`. In our last action we extended the code with the conditional blocks for the corresponding arithmetics.

Additionally we learned about the *SystemAlert* command which can be most helpful for status display or debugging of a script. Most important is a well done documentation inside a script, simply done by adding comment lines or comment additions in script code lines, both initiated with the two signs `//`.

Creating a simple 3D object per scripting

We now want to go on a step further and create a simple 3DObject, a cube. First let us look how this is basically created in trueSpace ? This gives us the knowledge we need to influence parameters that can do almost anything with 3D objects through scripting.

Generally a 3DObject needs the following information data so it can be formed:

- **vertex** or plural **vertices** are base of all 3D objects
- connecting lines between the vertex, called **edge** plural **edges**
- areas, bordered by **vertices** and **edges** , called **face** plural **faces**

One can already recognize that there are dependencies. To describe a *face* you need informations about *edges* which form the border for the *face*. The *edges* now need to know the two *vertices* which in fact are their borderline (equals start and ending point)

Of course we work in a three dimensional room with our 3D objects where every point can be described by the coordinates x,y,z and you may have learned in geometry that coordinates respectively point towards width, deepness and height. Here in our viewable 3D scenery they are used as a relative distance to a certain point. This is for the object axis the distance x,y,z to the point 0,0,0 of the scene. Each vertex of an object now uses the distance x,y,z to the object axis. So working in our 3D room we always have to keep in mind which point 0,0,0 we are relatively referenced too.

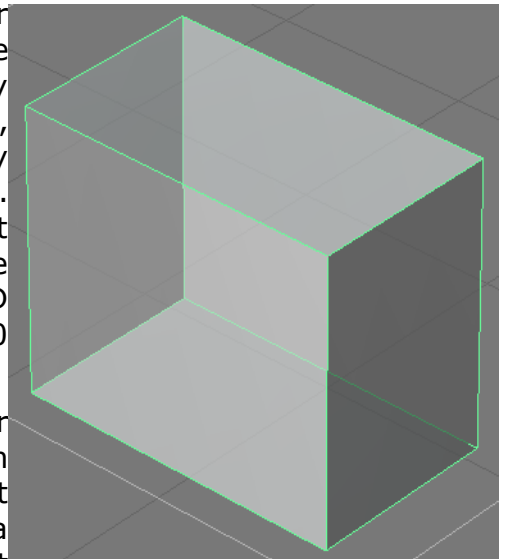
To create a cube we need therefore the coordinates for the 8 *vertices* which will allow to span the *edges* between them and so makes it possible to form the faces. What we get then is kind of a network forming the cube. Such a network is named as **Mesh**, a name that we will meet

quite often when diving deeper into scripting. Let us create a normal cube with the primitives tool in trueSpace and take a look at the cube brick appearing now in the LE. We can find a connector **Mesh**, something for *material* and several connectors for different kind of *Matrix* data. The **Mesh** connector transports a stream of data containing a lot of details that describe the cube. Streams of data are also behind the different *Matrix* connectors and this will meet us again and again when working with truespace, LE and scripting. Getting details out of the data stream is something that trueSpace will support us with through special already existing scripting functions. So there will be no need to become a bit juggling program profi, Caligary put some effort in it so we can work quite comfortable when scripting. A great help will be the basics of structural programming which we will meet there and all we need to do is improve our own knowledge how to tell trueSpace what we want to happen.

Ok, that is the message for us to start with creating a cube simply with a script now. Yes, let's go, but how do we start now ? That's easy, yes, it was already said above, we need to improve our knowledge. Aha, but neither the artist guide nor the developers guide will tell us how to create a simple cube with a script. But it suggest somewhere to use the scripts already included in trueSpace to see how things are done.

What we need is to

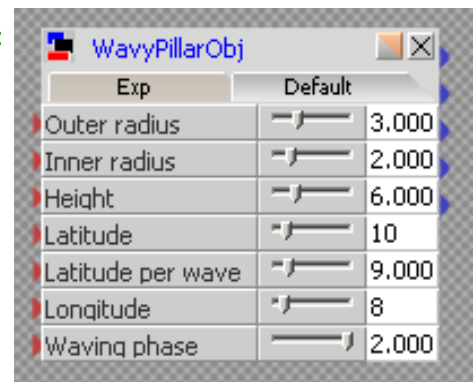
- set up the *vertices*,
- connect the *edges*
- to form the *faces*,
- finally get as result a cube.



Looking around we should notice that the **WavyPillar** script object is at least capable to create a cube if we feed in the right parameters. Oh boy, looking into the script code doesn't help much, looks as if Einstein was enjoying to write down something. That looks very cryptic to script beginners. So let us see what is happening there. For this we will take the original scripting code and try to explain what they did there so we can learn.

After loading the **WavyPillar** we will rename it and save this renamed object. Now enter it with right click on the orange triangle where we will find the *WavyPillarObj*. Entering this we reach the script code that does all these funny things when pushing around the sliders. We will start with the *attributes* as they can be set so that we get a simple cube. There is one difference though, our cube created above did have squares as faces, here we suddenly have triangles. This is correct as every object is created with triangles but on default trueSpace doesn't show us the triangles if we don't set the corresponding setting to do so.

```
function OnComputeOutputs(params) //compute values of all output
connectors
{
    longitude = params.conValue('longitude'); // HORIZONTAL
    radius    = params.conValue('radius');
    height    = params.conValue('height');
    latitude  = params.conValue('latitude'); // VERTICAL
    phase     = params.conValue('phase');
    latPerWave = params.conValue('latPerWave');
    innerRad  = params.conValue('innerRad');
    nVert     = longitude * latitude + 2;
    nFaces    = 2 * longitude * latitude;
}
```



Playing a little around with the sliders we can see different forms but we also can see the influence each parameter does have on the object. So for example *latitude* is set to 10 but we can count only 9 squares formed each of two triangles. But that is correct, *latitude* gives us the number of *vertices* in the vertical direction, and there are really 10 of them. Let us see if that is true for *longitude* too because with that parameter we can form a square so that the object almost looks like a cube. Yes, it says 8 and there are really 8 vertices around the upper plane and so also around the lower plane.

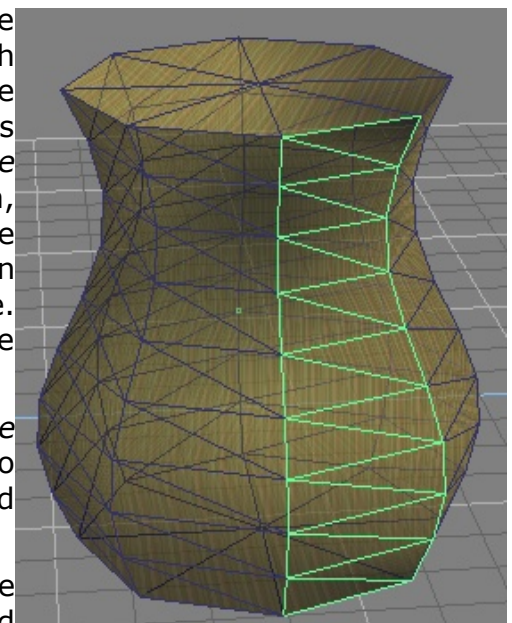
Important for us seems to be two attributes, *longitude* and *latitude* as we will find out now when digging into this. *Longitude* forms a square if we reduce it to 4 and latitude if then reduced to 2 finally will show us a cube.

Anything else we need, yes, if we watched precisely we should notice that the attribute **Outer radius** passed over as **radius** is also necessary as it defines the size of the cube and the **height** which sets the height.

Anyway, since we are already diving into the original script let us see what happens there and afterwards delete everything unneeded for our request of creating a cube.

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

Sounds good, first define the *vertices* as they are the basics for an object, that's exactly what we learned above. Premiere, first time now we will get in contact with one of the genius things Caligari has integrated into script programming, so we don't have to dive into the bits of data streams: **dV = System.CreateDO("Space 3D Package/Vertex Stream Data");**



Behind this is hidden a kind of program sorting the *vertex* information we enter in a correct way so that trueSpace can understand it. For quite a few things as you will see, always starting with **Space 3D Package/** followed by a functionality, we will find such comfortable helpers. Next line **dV.SetNumVertices(nVert);** passes on the number of vertices.

Let's look back where the attributes are listed and it says: **nVert = longitude * latitude + 2** and check if this turns out correct now. The total number of vertices for a cube is **8** and we wanted the height to be one square which would mean **2 vertices** if you look in vertical direction. We defined that *longitude* gives us the number of vertices horizontally which means we have 4 of them while vertically we need 2 vertices defined by *latitude*.

nvert = 4 * 2 + 2

Almost correct, we planned 8 but here the result will be 10. Must be a reason for this so we will have to watch out when continuing to work through the code. Now there is only one line left **dV.BeginWrite();** to be considered. This kind of opens an "ear" that listens what data we will create from now on to define the vertices.

```
// ----- ... add vertices circles
for ( j = 0; j < latitude; j++ )
{
    z          = height * (j / (latitude-1));
    angle      = 0.0;
    angleStep  = 2.0 * Math.PI / longitude;
    currPhase  = phase + j / latPerWave;
    sinVal     = Math.sin(2.0 * Math.PI * currPhase);
    rad       = innerRad + (sinVal + 1.0) * (radius - innerRad) / 2.0;

    for (i = 0; i < longitude; i++)
    {
        dV.x(i + j*longitude) = Math.cos(angle) * rad;
        dV.y(i + j*longitude) = Math.sin(angle) * rad;
        dV.z(i + j*longitude) = z;
        angle                 += angleStep;
    }
}
```

Let us first sort out what we will not need. This is *currPhase*, *sinVal* and *innerRad* as they all are used to create the funny effects which we don't need for our simple cube. So we also can correct *rad* now, which means we can simply use *radius* instead.

```
// ----- ... add vertices circles
for ( j = 0; j < latitude; j++ )
{
    z          = height * (j / (latitude-1));
    angle      = 0.0;
    angleStep  = 2.0 * Math.PI / longitude;

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

But now finally some explanations what is happening there. At first we have a new script comand which performs a loop. The line **for (j = 0; j < latitude; j++)** sets the conditions for the loop while the {...} include the script commands which the loop runs through as often as the conditions say. The loop counter is hidden in the variable **j** which is initiated with the value **0** , the ending condition is **j < latitude**, after each loop step **j++** increments j with 1. Important, but typical is the fact that the counter starts with **0** and not with 1! We will not go into the mathematical formulars that follow, just keep in mind they give the increment from a point 0 for the vertices which are positioned relativ to this point 0. At point 0 we will place our first vertex. Inside this loop we have another loop now.

This next loop runs from **0** to **longitude -1**. Why, well the loop works as long as **i** is smaller but stops immediatley when it has the same value. For our cube we defined *latitude* with the value of 2 and *longitude* with the value of 4. This means first loops runs **2** cycles while the second loop will run **4** cycles. The cycles in this loop now should create the

four *vertices* in the lower plane of the cube and then when the outer cycle runs a second time the four *vertices* in the upper plane will be created. Well, *z* is the coordinate value that

```
z          = height * (j / (latitude-1));
angle      = 0.0;
angleStep  = 2.0 * Math.PI / longitude;
```

will influence positioning if upper or lower plane. *angleStep* is the factor needed to position the four vertices of a plane.

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

So here it is happening now what the "ear" must hear. But we still find something unknown which is the definition of **dV**. What does this stay for ? Well, that is the influence of object oriented programming. Looking back we had **dV = System.CreateDO(.....** which is the key to unlock this secret. You can say **dV**. is the reference for the *Vertex Stream Data*. Think of an address where you have a city, let that be **dV** then we have a street, let that be **x** followed now by the house number which could be the numbers inside **x** . Typing in an address we will enter a return to separate the lines. Here we use the dot (.) for separation. So **dV.x** stays for the **x** coordinate of a *vertex* .**y** and **.z** for the rest. Any question left, yes how do I address a single vertex if there are quite a number of them ?

This happens inside the paranthethesis where you find the parameters building the number from 1 to 8 for our cube example. At first outer cycle loop we have $i+j = 0+j$ and second loop we will have $1+j$. You can check this with a calculator by adding there the multiplication with *longitude*. This value inside the paranthesis is called index and a variable of this type, *pp(x)* is called an array. Such an index starts always with 0 and that is the reason why our loops count from 0 to 1 / 0 to 3 than rather from 1 to 2 / 1 to 4

Let us put this now together. We have longitude, latitude, radius and hight to define cube size and number of vertices. Then we get as a reference **dV** which allows us to get access to the "insided" parameters *x,y,z*. Finally we have an **index** with whom we can access each single vertex between a number of vertices.

Now we continue with the code to see what happens next as the edges and faces are still missing.

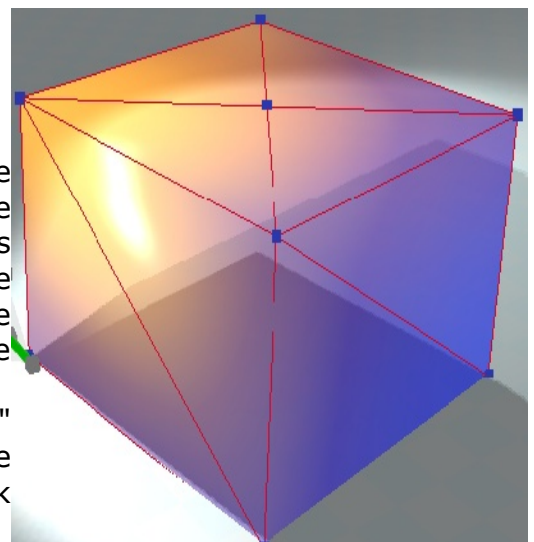
/ ----- ... add top and bottom vertices -----

```
dV.x(nVert - 2) = 0;
dV.y(nVert - 2) = 0;
dV.z(nVert - 2) = 0;

dV.x(nVert - 1) = 0;
dV.y(nVert - 1) = 0;
dV.z(nVert - 1) = height;

dV.EndWrite();
```

No loop, but what happens here ? Well, two more vertices are created ! Aha, that is the reason why there were 10 and not 8 vertices defined. Their coordinates are 0/0/0 at the lower plane level and 0/0/*height* at the upper plane level. This will create a cube looking like the one to the right and the two additional vertices are the ones in the middle of the upper and lower plane. With the command **dV.EndWrite()** we say that the "ears" listening to this stream of vertices data should be closed now. The Caligary helper can now finish it's work to create a package understood by other components.



And here it is, another nice caligari package that should be able to read the above vertice package so that faces can be created out of it. First we have the initialisation and the

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

reference is pointed to the variable **dF** this time. What follows is passing over the number of faces. To get this we must remember that there are always triangles which form the rectangles to then form a model, in our case a cube. OK, we have a cube, that of course has 6 squares or rectangles. Each one of them is build out of two triangles makes in sum 12 of them. Oh boy, didn't we love geometry in school, now we finally know what it was good for. Since there are two more vertices, the upper and lower plane could be build with 4 triangles each. Well, we will try this practically after we are through our theory to see what we really need. Just a remark, trueSpace 7.11 is very sensitive if you give it too many faces, which definitely leads to a program crash and loss of all the work that wasn't saved since last start of the program. So be carefully when experimenting, never enter too many number of faces than possible or save your work before doing it. Next command tells the *Vertice Stream Data* to open the "ears" and listen.

The code now continues with creating the side faces, also with loops. The value **dF.** is here the reference followed by **i,j,k**. What do they stand for, well they are the reference to the three *vertices* necessary to form the triangle of a *face*. Then follows the index value in parenthesis which points to each single face, so in our example this must go from 0 to 11 as we said 12 faces.

```
/ ----- ... define side faces -----
for (j = 0; j < latitude - 1; j++)
{
    shift = 2 * j * longitude;
    shiftV = j * longitude;
    for (i = 0; i < longitude - 1; i++)
    {
        dF.i(i+shift) = i+shiftV;
        dF.j(i+shift) = i+shiftV+longitude+1;
        dF.k(i+shift) = i+shiftV+longitude;

        dF.i(i+shift+longitude) = i+shiftV;
        dF.j(i+shift+longitude) = i+shiftV+1;
        dF.k(i+shift+longitude) = i+shiftV+longitude+1;
    }
    dF.i(shift+longitude-1) = longitude-1+shiftV;
    dF.j(shift+longitude-1) = shiftV+longitude;
    dF.k(shift+longitude-1) = shiftV+longitude-1+longitude;

    dF.i(shift+longitude+longitude-1) = longitude-1+shiftV;
    dF.j(shift+longitude+longitude-1) = shiftV;
    dF.k(shift+longitude+longitude-1) = shiftV+longitude;
}
}
```

Generally this works a similar way as setting the coordiantes above. Here we get as a result though the number of a vertice which can be from 1 to 8 (or from 1 to 10 if we set the middle vertices in the upper and lower plane) If you take your calculator or your well trained brain and a pencil you will find out that the first face is constructed from the vertices number 0,5,4, second one from vertice number 1,6,5 and so on.

What follows now is the command for closing "ears" **dF.EndWrite();**

creating the mesh data that truespace needs and understands

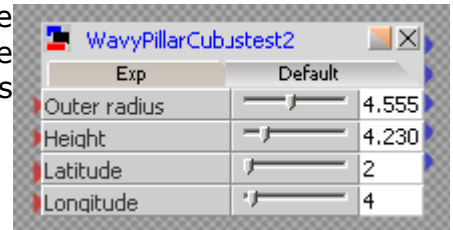
```
// ----- create final mesh -----
Mesh = System.CreateDO("Space 3D Package/Mesh Data");
Mesh.AttachVerticesStream(dV);
Mesh.AttachTrianglesStream(dF);
```

and passing on this data with the help of the connector mechanism to the outside

```
// ----- set output finally -----
params.conValue("Mesh") = Mesh;
```

Guess now it is time to rest a bit and think about all of this and reread the parts still unclear. What we are dealing here with are the very basics and it will be rare occasions one will create a simple cube this way. But working with many of the other available functions will be easier with an understanding what happens in the underground.

Before getting to higher functions we will modify the original **WavyPillar** code a bit to get used to the script editor and its functionings. Therefore I changed the WavyPillar object to our demands, just *radius*, *height*, *latitude* and *longitude*. The settings should be changed so that *vertices* and *edges* are visible. Also set the *opacity*. Then just watch what effects playing around will come up.



```
// WavyPillar code modified for experimenting, phase 1
// Called to compute values of all output connectors
function OnComputeOutputs(params)
{
    // obtain input params needed, only necessary attributes left
    longitude = params.conValue('longitude');
    radius     = params.conValue('radius');
    height     = params.conValue('height');
    latitude   = params.conValue('latitude');
    nVert      = longitude * latitude + 2;
    nFaces     = 2 * longitude * latitude;

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

    // ... add vertices circles , only radius(= Outer radius) needed
    for (j = 0; j < latitude; j++)
    {
        z = height * (j / (latitude-1));
        angle = 0.0;
        angleStep = 2.0 * Math.PI / longitude;

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

    // ... add top and bottom vertices
    dV.x(nVert - 2) = 0;
    dV.y(nVert - 2) = 0;
    dV.z(nVert - 2) = 0;
    dV.x(nVert - 1) = 0;
    dV.y(nVert - 1) = 0;
    dV.z(nVert - 1) = height;
    dV.EndWrite();

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

    // ... define side faces
    for (j = 0; j < latitude - 1; j++)
    {
        shift = 2 * j * longitude;
        shiftV = j * longitude;
        for (i = 0; i < longitude - 1; i++)
        {
            dF.i(i+shift) = i+shiftV;
            dF.j(i+shift) = i+shiftV+longitude+1;
            dF.k(i+shift) = i+shiftV+longitude;
            dF.i(i+shift+longitude) = i+shiftV;
            dF.j(i+shift+longitude) = i+shiftV+1;
            dF.k(i+shift+longitude) = i+shiftV+longitude+1;
        }
        dF.i(shift+longitude-1) = longitude-1+shiftV;
        dF.j(shift+longitude-1) = shiftV+longitude;
        dF.k(shift+longitude-1) = shiftV+longitude-1+longitude;
        dF.i(shift+longitude+longitude-1) = longitude-1+shiftV;
        dF.j(shift+longitude+longitude-1) = shiftV;
        dF.k(shift+longitude+longitude-1) = shiftV+longitude;
    }
}
```



```

    for (i = 0; i < longitude; i++)
    {
        dV.x(i + j*longitude) = Math.cos(angle) * radius;
        dV.y(i + j*longitude) = Math.sin(angle) * radius;
        dV.z(i + j*longitude) = z;
        angle += angleStep;
    }
}
// ... add top and bottom vertices
dV.x(nVert - 2) = 0;
dV.y(nVert - 2) = 0;
dV.z(nVert - 2) = 0;
dV.x(nVert - 1) = 0;
dV.y(nVert - 1) = 0;
dV.z(nVert - 1) = height;
dV.EndWrite();
// ----- define faces
dF = System.CreateDO("Space 3D Package/Triangle Vertices Stream Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();
// -----
// ---> all loop comands deleted
// ... define faces reduced to one single face WATCH OUT !!!
// you can enter any number between 0 and 9 every other value will
// lead to program crash
        dF.i(0) = 4;      // change manually
        dF.j(0) = 5;      // change manually
        dF.k(0) = 8;      // change manually
// -----
        dF.EndWrite();
// create final mesh
Mesh = System.CreateDO("Space 3D Package/Mesh Data");
Mesh.AttachVerticesStream(dV);
Mesh.AttachTrianglesStream(dF);
// set output finally
params.conValue("Mesh") = Mesh;
}

```

