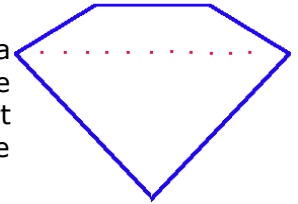


Creating a Diamond with scripting Emma @ 3d-ts-forum / Caligariforum, Nov. 2006, M. Moersner

The knowledge we have now from the introductory scripting tutorial will enable us to create a Diamond, simply by a script. Really simply, is that possible? Sure, it is little work to get there but it in any case it will be worth to follow. This is a good chance to see that modeling in trueSpace can be quite well supported by that little bit of scripting knowledge we already learned so far. At the end you will see the advantage of creating such a Diamond through scripting rather than modeling it manually.

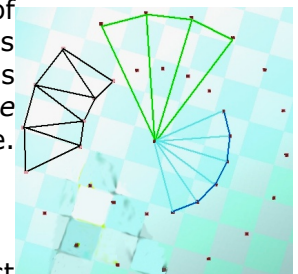
First thing again is to define what you want to do. We want to create a Diamond with a whole lot of facetes. We must be exact so we say that he should have **18** facetes around the outer surrounding *face*. In the upper part we start easy and just will have one row of facetes plus the table facet. So the outline will look as the picture to the right shows.



Now we can already go into analyzing and planing what we will have to do in the script. This is no difference to modelling as there you also have to think about and analyze what you will have to do, which way and with which tools. The difference will be that here you move the vertices with your mind, not with your mouse pointer.

We have the tip at the bottom which will need only one *vertex*. Then we have in the upper half a wide circle where we would have **18** *vertices* so that there will be **18** facetes (or *faces*). Now we reach the upper part, the table facet, which is also surrounded by a circle of **18** *vertices*. Later we can change this to get different cuttings of a Diamond. So this means the circle of *vertices* in the middle will stretch his *faces* down to the tip and up to the table facet. Anything else we need, yes one more *vertex* in the middle of the table facet to create our triangles for the table facetes. This way we can do it with a simple loop.

Let's take a short way and go directly into the code. We have following vertices, **1** for tip, **18** for middle row and **18** for upper row surrounding the table facet plus the one in the middle of the table. The picture to the right shows in the middle the two *vertices* on top of each other, one for the tip and one for the middle of the table. The green lines mark the *faces* reaching from the middle ring towards the tip. The ones marked black are the *faces* reaching from the middle towards the *edge* surrounding the table. The blue ones finally are the *faces* that build the table. So in sum we have now :



- **from tip to table 1 + 18 + 18 + 1 = 38 vertices (numbered 0 to 37)**
- **from tip to table we then have 18+18+18+18 = 72 faces (0 to 71)**

That calls for some loops we already know from our cube script, here we just need some more *vertices* that have to be filled with *faces*. As variables we will use

- **longitude** - number of *vertices* needed for **18** *faces*
- **radius** - radius for ring of *vertices*, we choose the middle ones

This gives us a nice little loop counting **18** cycles. Now there is a modification so that two rows of *vertices* are build and one row position is shifted a little bit so that the *vertices* will stay in the middle between the other rows *vertices*. In the picture above that can be well seen where the green edges of the faces are shown, the upper rows *vertices* (smaller ring) are in the middle.

```
angleStep = 2.0 * Math.PI / longitude; // step width from vertex to next vertex
dist      = height/4 ;                 // distance between table/middle vertices
pip       = longitude ;                 // number of vertices in upper/middle circle
for (i = 0; i < longitude; i++)
{
    dV.x(i)      = Math.cos(angle) * radius; // middle vertices
    dV.x(i + pip) = Math.cos(angle+angleStep/2) * radius*0.6; // table vertices

    dV.y(i)      = Math.sin(angle) * radius; // same for .y
    dV.y(i + pip) = Math.sin(angle+angleStep/2) * radius*0.6;

    dV.z(i)      = dist; // same for .z
    dV.z(i + pip) = 0;

    angle += angleStep ; // increment to next vertex
}
```

The index is different because of the following reasons:

dV.x(i) - the index **i** is counted from **0** to **17** equals **18** steps

dV.x(i + pip) - the index **i + pip** is counted from **18** to **35** steps for second row

To keep the program short we use one loop to count through. The second row has different location so the parameters are a bit different for height and diameter. Of course we could also use a second loop. There are always several way leading towards Rom.

So this way we get the first **18** and the second **18** vertices. Next problem to solve is the shift (also expressed as offset) of the second row. Well, the shift of vertices is done by the value of **angle**. So all necessary is to change this value for the upper row of vertices. This happens simply with **(angle+angleStep/2)** which places them in the middle in between.

To get the two circles in different levels we have to modify the **height** parameter which is done by the variable **dist**. It's simply divided by **4** so we choose level difference to be **¼** or **25%**.

Left now is decreasing the diameter of the upper row of vertices, which happens by changing the coordinate values through *** radius*0.6** instead of *** radius**. So this finally leaves now only the tip and the table vertex to be created.

```
// ... add tip and table 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;

dV.EndWrite();
```

OK, if let us just set the vertices now and watch if it looks like a Diamond shape. The code is most similar to what we did in the first tutorial. We rather move in circles than in squares now.

```
// Diamond creator test code part 1    November 2006 Emma@Caligari-forum/3d-td-forum
function OnComputeOutputs(params)
{ // obtain input params-----
    longitude = params.conValue('longitude');
    radius     = params.conValue('radius');
    height     = params.conValue('height');
    latitude   = params.conValue('latitude');
    nVert      = 2 * longitude + 2;
    nFaces     = 1 ;    // IMPORTANT, dont change unless you create more faces ! ! !
// define vertices-----
    dV = System.CreateD0("Space 3D Package/Vertex Stream Data");
    dV.SetNumVertices(nVert);
    dV.BeginWrite();
    angle     = 0.0;
    angleStep = 2.0 * Math.PI / longitude;
    dist      = height/4 ;
    pip       = longitude ;
// loop creates two rows of vertices in one cycle of length longitude (here 18)
    for (i = 0; i < longitude; i++)
    { dV.x(i)      = Math.cos(angle) * radius;
      dV.x(i + pip) = Math.cos(angle+angleStep/2) * radius*0.6;

      dV.y(i)      = Math.sin(angle) * radius;
      dV.y(i + pip) = Math.sin(angle+angleStep/2) * radius*0.6;

      dV.z(i)      = dist;
      dV.z(i + pip) = 0;

      angle += angleStep ;
    }
// ... add top and bottom vertices-----
    dV.x(longitude*2 ) = 0;
```

```

dV.y(longitude*2 ) = 0;
dV.z(longitude*2 ) = height;

dV.x(longitude*2 + 1) = 0;
dV.y(longitude*2 + 1) = 0;
dV.z(longitude*2 + 1) = 0;

dV.EndWrite();

// ----- define faces longitude = 18   latitude = 2
dF = System.CreateD0("Space 3D Package/Triangle Vertices Stream Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();
dF.i(0) = 0;
dF.j(0) = 35 ;
dF.k(0) = 18;

// create mesh -----
dM = System.CreateD0("Space 3D Package/Mesh Data");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);

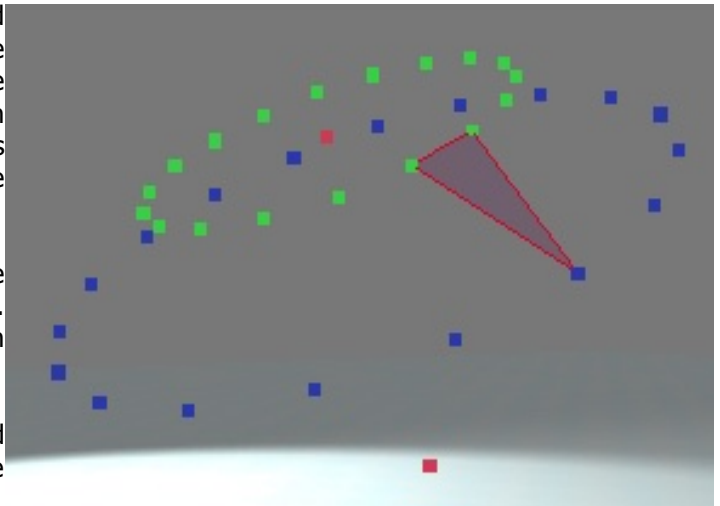
params.conValue('outMesh') = dM;
}

```

In this test program we create next only one *face*. If we tell the program there will be one face, then the necessary parameters have to be set completely and correct, otherwise the program can crash. To have an idea for the positioning we set the face on the vertices **0**, **18** and **35**. We remember, the number of a *vertex* is passed over to the *face* creation, and so there are three numbers necessary to define a *face*. The first ring runs from **0** to **17** (=18 vertices in sum) the second from **18** to **35** (=18 vertices in sum). This will help us for positioning the *faces*.

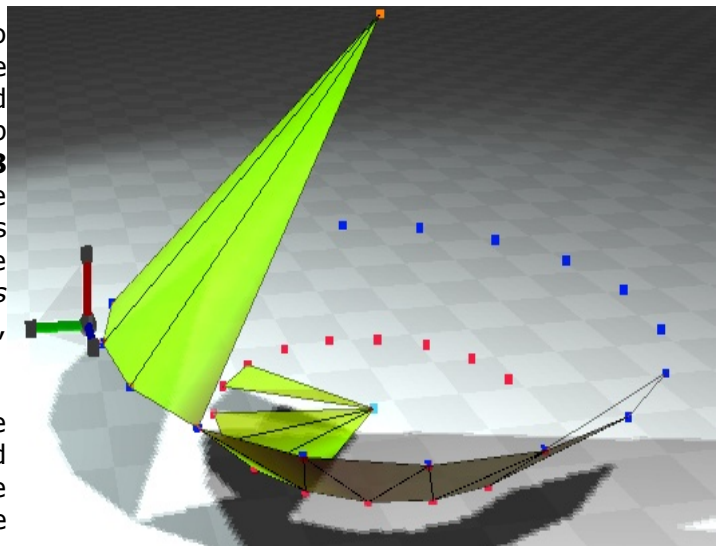
It's a good exercise now to change the three vertex numbers to **0,35,34** and to **1,35,34**. Why, well this way we can see which direction the numbering is running.

The green circle are the vertices that surround the table . The red ones are the one in the middle of the table and the one for the tip.



Ok, doesn't look that bad so it is time now to go on for creating the *faces* for our *facetes*. The picture I show here gives an idea how it should look. There are **18** *facetes* reaching from tip towards the middle ring of *vertices*. Next **18** *vertices* go between middle to upper ring, while the table finishes with the last **18** *faces*. Let us take a last look, is that correct now ? No, there are two rings of *vertices* connected by *faces* and this doubles the *faces* there to **36**. Why, well we work with triangles, not with squares.

So next step to do now will be developing the necessary code for the *faces*. For this it would help printing out a picture where only the *vertices* are shown and mark them with the corresponding number from **0** to **37**. We now will have a much more complicated dependency so construction must be most precisely done.



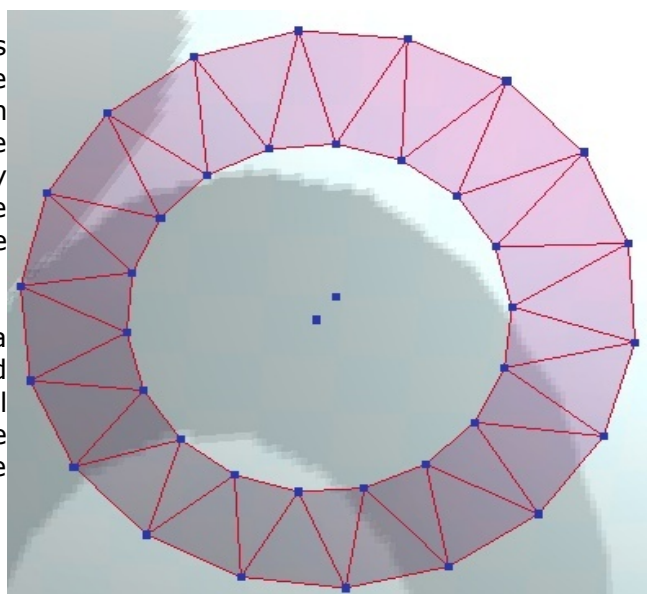
Think now it is time to talk a bit about preventing errors. To do this we must know what we are doing, why we do it, where does what we do influence other things. Let's go back to 1968 when I was programming an old **IBM 1620**. As the machine was reading punched cards one could follow the program by watching the lamps. You could interact by moving some switches on or off, you were able to know what data was in each of the 20.000 core memory positions, you were able to control and change each one of the core memory positions. I donated that core memory and the „console“ of the **IBM 1620** years ago already to a museum, that's definitely history (see picture).



The old programs couldn't have more than **20.000** commands, trueSpace in version 7.11 shows up with **809.000**. But that's not all, we also find libraries (.dll), for example something like *tsBridge.dll*, not even the largest one with size of **3.060.000** bytes. Then we have operating system, graphics driver and of course tons of data for configuration, scenes, material, lights,..... Changing one bit of information can have cross-influencing from trueSpace core, through several libraries, on to the operating system, continue to some driver, yes, this chain of flow could even run back.

Modern object oriented programming „narrows“ a programmers view, only aspects (or attributes) passed on to the next „object box“ are taken in account. Interface connection passing on information is the end of responsibility for the „context“ that is just being programmed. Creating the vertices is so far only a simple list of coordinates, adding faces now will give references for each face to some vertices. Adding material, physics, and ,and ,and will make it more and more complex. I hope it is easy to **see that everything possible should be taken in account to guarantee a minimum risk of errors and so leading to success.**

First we will create now the double ring of faces between middle and upper row of vertices. The picture to the right shows also the two vertices in the middle, on at the tip and the other one in the middle of the table facet. We did already try above setting the parameters for a face. Now we will of course let a loop do the work to create necessary settings for all faces.



But there is one complication the last face closes a circle and therefore it will use the last vertex and the first vertex of the row. For this reason we will start on vertex 1 with our loop and because we have an offset shift of the upper row we will have following order now:

```
dF.i(0) = 1;
dF.j(0) = 18;
dF.k(0) = 19;
```

```
dF.i(1) = 2;
dF.j(1) = 19;
dF.k(1) = 20;
```

```
dF.i(2) = 3;
dF.j(2) = 20;
dF.k(2) = 21;
```

We will put that in a table to make it easier to understand the logic

i	j	k
1	18	19
2	19	20
3	20	21
4	21	22
5	22	23
..	..	..

You can load **Diamon001T.RsObj** to play around with this ---->



That looks nice and will fit well into a programmed loop. We have one variable counting from **1** to **17** (remember from **18** faces we wanted to construct by loop only **17** while the last one, connecting start and end, would be constructed separately) so this will give us a code like this:

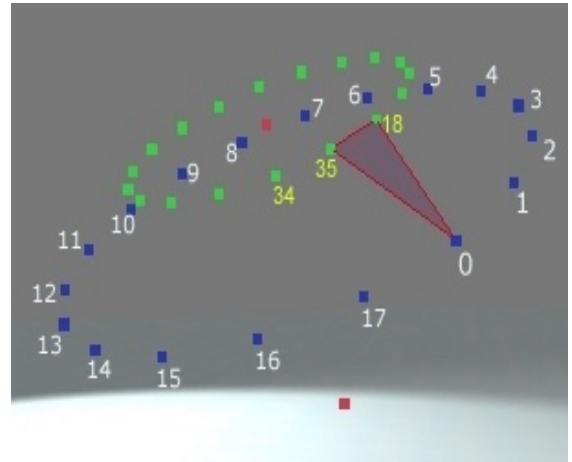
```
// -- define faces, calculated from longitude = number of faces around each circle
```

```
dF = System.CreateD0("Space 3D Package/Triangle Vertices Stream Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();
```

```
// ----starting upper facetes first row-----
```

```
pb = longitude + 1;

for (i = 0; i<pa; i++)
{
    pd      = 1 + i;
    dF.i(i)  = pd;
    dF.j(i)  = i + longitude;
    dF.k(i)  = i + pb;
}
```



For **dF.i** we simply use the *loop counter + 1*

For **dF.j** we increment longitude with the loop counter

For **dF.k** we create pb and increment that with the loop counter value

Now the loop has all necessary parameters and so we will set correct value of **nfaces** before installing now the loops for *face* creation. Use **Diamon001T.RsObj** simply change nFaces and exchange the manual assignments against following code.

```
.....
nFaces      = 36 ;    // IMPORTANT, dont change unless you create more faces ! ! !
```

```
// ----- define faces, based on value of longitude
```

```
dF = System.CreateD0("Space 3D Package/Triangle Vertices Stream Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();
```

```
// ---- upper facetes first row-----
```

```
pa = longitude - 1;
pb = longitude + 1;
pc = longitude - 2;

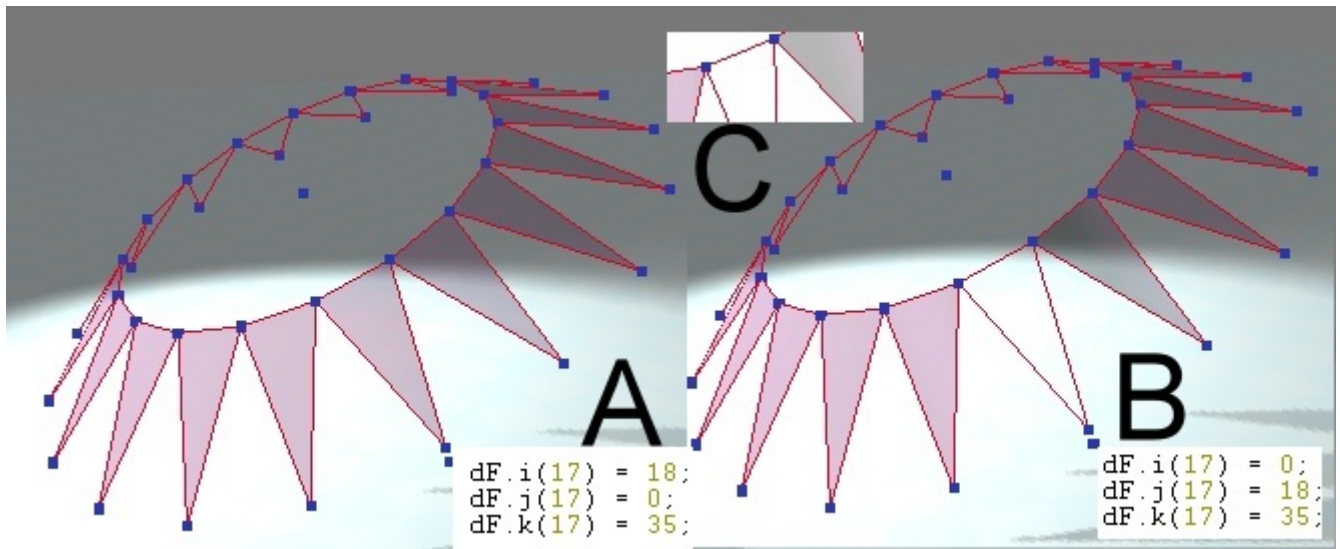
for (i = 0; i<pa; i++)          // ---> per loop start from second end with last face
{
    pd = 1 + i;
    dF.i(i) = pd;
    dF.j(i) = i + longitude;
    dF.k(i) = i + pb;
}
dF.i(pa) = longitude;           // ---> first face connecting first and last vertex
dF.j(pa) = 0;
dF.k(pa) = longitude*2-1;
```

```
// ---- upper facetes second row, points opposite to first row -----
```

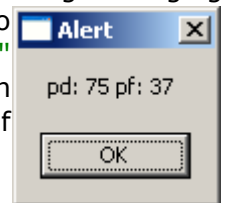
```
pd = 2*longitude-1;

for (i=longitude; i<pd; i++) // ---> per loop start from second end with last face
{
    dF.i(i) = i ;
    dF.j(i) = i - longitude +1 ;
    dF.k(i) = i - longitude ;
}
dF.i(pd) = longitude-1;         // ---> first face connecting first and last vertex
dF.j(pd) = longitude*2-1;
dF.k(pd) = 0;
```

After playing around with this it is a good moment to explain some of the things that may happen.



If you get an effect like shown in B, then the order of the *vertex* assignment to a *face* is incorrect. You see in A that if the order is changed everything is ok. This effect is called face culling. Changing assignment order for the face solves this. Sometimes it is also nice to get some info out of the script, there helps a command like follows: `System.Alert("pd: "+pd+" pf: "+pf);` Inside the parenthesis you can add a text enclosed by " then with +VARIABLE and/or +" TEXT" you can get more info to the outside. But remember, if you do this inside a loop you will have to say OK as often as the loop runs.



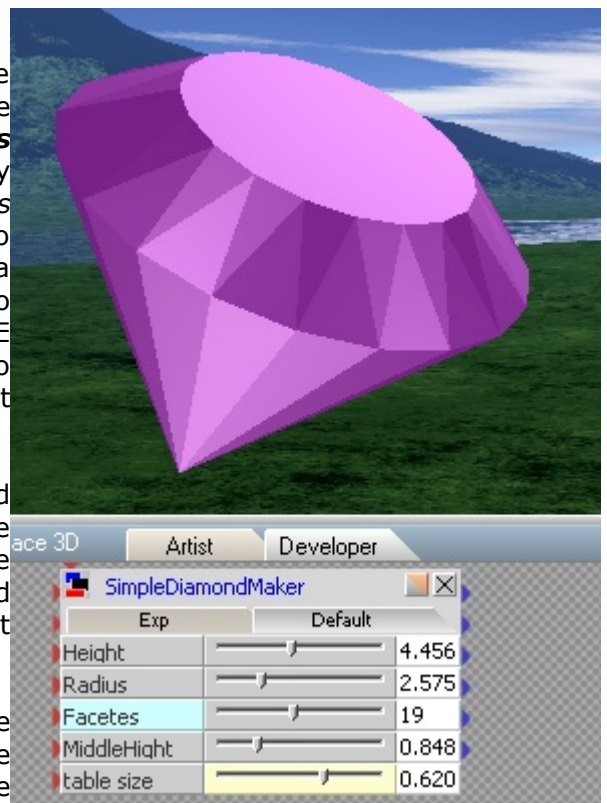
SimpleDiamondMaker

Guess now it is time to do the whole rest, creating the *faces* connected to the tip *vertex* and table *faces*. We also will add some enhancements like making *nFaces* variable depending on longitude now, insert possibility to change the distance between table ring of *vertices* and the lower one and last not least opening a way to change the diameter of the table plate. That is also a good moment to clean up the outlook and rename to **SimpleDiamondMaker**. Changing the outlook in the LE was already explained in another tutorial so I'll just go into describing what connection is between script variables and what we see in the LE block.

First a thought: we want to add some more features and so we need a variable for changing distance between the two *vertices* rows and another one for changing the diameter of the table plate. It's time now also to add some control for preventing errors so this little script could be passed on to others.

For the distance we take the variable **difi** and on the outside world we name it **MiddleHeight**. For the table size I decided to take **damian** as variable and on the outside world I named it **table size**. Good programming

means also documentation and so we also add some comments at the beginning of the program. Good style is also mentioning if others distributed important things to your script. This script is a modification of the **WavyPillar** object which originates from the original Caligari trueSpace 7 installation. There is changed so much in the code that it will not be seen at first glance, but mentioning is also important to point others to the origin which then may give them other ways of possible variations.



- **first we start with some short documentations**

```
function OnComputeOutputs(params)
{
// -----SimpleDiamondMaker---Emma@3D-ts.forum / Caligariforum--Nov. 2006--
//
// modified from original script: WavyPillaObj
//
// longitude - gives the number of facetes around the middle of the table facete
//             middle ring of facetes is twice the # of table facetes
// longitude - is equals # lower ring of facetes
// difi       - is the width factor between upper and middle ring of vertices
//
// longitude - should not go below value of 4
//
// latitude   - for future enhancement if more facete rings will be programmed
// damian     - calculation value for table size
//
// Diamond creator sample with trueSpace 7 scripting
// November 2006 Emma@3d-ts-forum / Caligari-forum
// -----
```

- **our connection to the outside world has changed also a little bit**

```
longitude = params.conValue('longitude');
radius    = params.conValue('radius');
height    = params.conValue('height');
latitude  = params.conValue('latitude');
nVert     = 2 * longitude + 2;
nFaces    = 4 * longitude ;
difi      = params.ConValue('difi');
damian    = params.conValue('damian');
```

ConeMesh
Object
Script Object Package/jScript language
in longitude : int 'No description'
in radius : number 'No description'
in height : number 'No description'
out outMesh : 'Space 3D Package/Mesh Data' 'No description'
in latitude : int
in difi : number 'width calculation factor'
in damian : number 'change table facet'

- **inside the script we do some minor checks to prevent errors**

```
// -----minimum error trace inside the script
if (damian < 0.1) {damian = 0.6;}
if (difi < 0.25) { difi = 0.25; }
if (longitude < 4 ) { longitude = 4; }
```

- **starting with creation of the vertices, depending on value of longitude**

// define vertices, loop creates two rings of vertices according to longitude

```
dV = System.CreateD0("Space 3D Package/Vertex Stream Data");
dV.SetNumVertices(nVert);
dV.BeginWrite();

angle      = 0.0;
angleStep  = 2.0 * Math.PI / longitude;
dist       = height/8 ;
pip        = longitude ;

for (i = 0; i < longitude; i++)
{ dV.x(i)      = Math.cos(angle) * radius;
  dV.x(i + pip) = Math.cos(angle+angleStep/2) * radius*damian;

  dV.y(i)      = Math.sin(angle) * radius;
  dV.y(i + pip) = Math.sin(angle+angleStep/2) * radius*damian;

  dV.z(i)      = difi;
  dV.z(i + pip) = 0;

  angle += angleStep ;
}
```

```

// ... add top (for table facetes) and bottom vertices, -----

dV.x(longitude*2 ) = 0;
dV.y(longitude*2 ) = 0;
dV.z(longitude*2 ) = height;

dV.x(longitude*2 + 1) = 0;
dV.y(longitude*2 + 1) = 0;
dV.z(longitude*2 + 1) = 0;

// vertice creation done so finish creation process

dV.EndWrite();

- starting with creaton of faces calculated

// -- define faces calculated from longitude * 4, must be changed if more than 4 row

dF = System.CreateD0("Space 3D Package/Triangle Vertices Stream Data");
dF.SetNumTripleIndices(nFaces);
dF.BeginWrite();

// ---- upper facetes first row-----

pa = longitude - 1;
pb = longitude + 1;
pc = longitude - 2;

for (i = 0; i<pa; i++)
{
    pd = 1 + i;
    dF.i(i) = pd;
    dF.j(i) = i + longitude;
    dF.k(i) = i + pb;
}

dF.i(pa) = longitude;
dF.j(pa) = 0;
dF.k(pa) = longitude*2-1;

// ---- upper facetes second row, points opposite to first row -----

pd = 2*longitude-1;

for (i=longitude; i<pd; i++)
{
    dF.i(i) = i ;
    dF.j(i) = i - longitude +1 ;
    dF.k(i) = i - longitude ;
}

dF.i(pd) = longitude-1;
dF.j(pd) = longitude*2-1;
dF.k(pd) = 0;

// ---- bottom facetes, from tip vertex towards middle vertices

pa = 2 * longitude;
pb = 3 * longitude - 1;

for (i=pa; i<pb; i++)
{
    dF.i(i) = i - pa ;
    dF.j(i) = i - pa + 1;
}

```

```

    dF.k(i) = pa;
}

dF.i(pb) = longitude-1;
dF.j(pb) = 0;
dF.k(pb) = pa;

// --- table facetes -----

pa = longitude;           // 18
pb = 1 + longitude;       // 19
pc = 3 * longitude;       // 54
pd = 4 * longitude - 1;   // 71
pf = 2 * longitude -1;    // 35

for (i=pc; i<pd; i++)      // 17
{
    dF.i(i) = i - pc + longitude;
    dF.j(i) = longitude * 2 + 1;
    dF.k(i) = i - pc + longitude + 1;
}

// sample how to give message ----> System.Alert("pe: "+pe+" pf: "+pf);

dF.i(pd) = longitude;
dF.j(pd) = 2 * longitude -1 ;
dF.k(pd) = 2 * longitude+1;

dF.EndWrite();

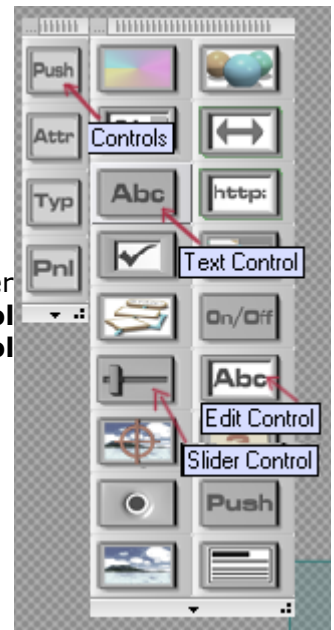
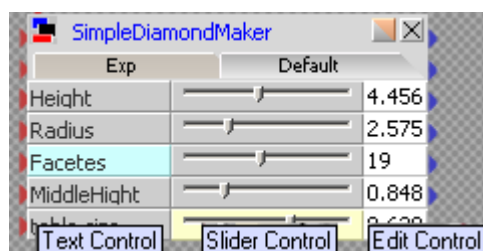
// create mesh, preparation of vertices & faces done, normals programemd later ----

dM = System.CreateD0("Space 3D Package/Mesh Data");
dM.AttachVerticesStream(dV);
dM.AttachTrianglesStream(dF);

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

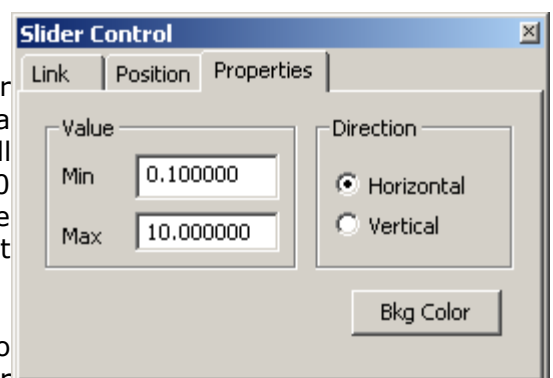
```

What should be done is also checking the LE blocks for preventing user errors. What we have in our **SimpleDiamondCreator** so far **TextControl** for showing names, **SliderControl** for changing values and **EditControl** showing numerical values to us.



In the Properties tab from the **SliderControl** the values for **Min** and **Max** should be correct. If you allow for example a value of **Min**=0 for Facetes, your script will crash. For all sliders control the values. Allowing a Radius Value of 10000 as Max for example will make it very hard to fine adjust the radius since even a small move of the slider will have great increase or decrease.

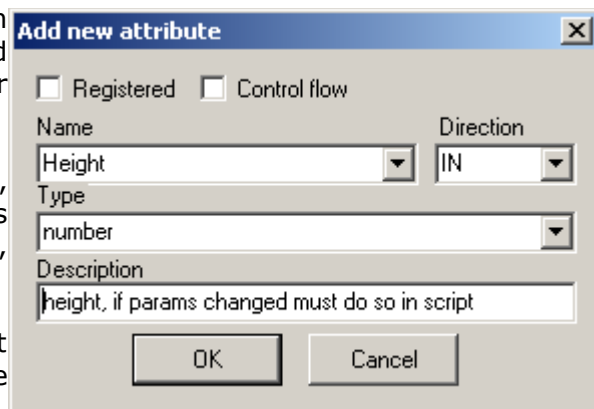
If you have larger values than make the sliders larger to make their usage a little easier. Also think about the order



in the block, having to move the mouse too much makes working more complicated. For this diamond example it is not that important, but will be with larger projects.

Use the **Description** field to give the user information, this helps preventing errors. If y user does changings without following the **Description** infos, he is to blame, not you as the programmer.

Now only give the right material to your diamonds to let them shine in brilliance. Selcet **AutoFacet** and set the **AutofacetAngle** to **0**



So what else to say ? This script will create the mesh for variable kinds of diamonds. Expansion can be to add another circlce of vertices to get more facetes. Or to do the calculation so that you will get ovals instead of round circles. End of imagination is the only border for extensions (and of course some more turoials to leran how to do things).

Next tutorial would cover how to add **normals** since the material between **Model** and **Player** works totally different. In **Player** mode you have to use **normals** to get sharp edges. One help till then can be to export your diamond as .3DS and import it back.

... to be continued

..... some day