Chapter 1

The Structure Layer.

1.1. How to draw elementary geometric shapes in the scene.

The *Structure Layer* permits to build the objects on a scene. Each object can be represented like a combination of one or more elementary geometric shapes (e.g., sphere, plane, cube).

`BabylonJS` permits to build a generic geometric shape by using this format:

```
var mesh = BABYLON.MeshBuilder.CreateXXX(name,{param1 : val1, param2: val2}, scene)
```

The istance produced by any method having this format is named *mesh*.

For example in order to create a spherical mesh you have just to use this code:

```
var sphere = BABYLON.MeshBuilder.CreateSphere("sphere",{diameter : 2}, scene);
```

and you get this result:

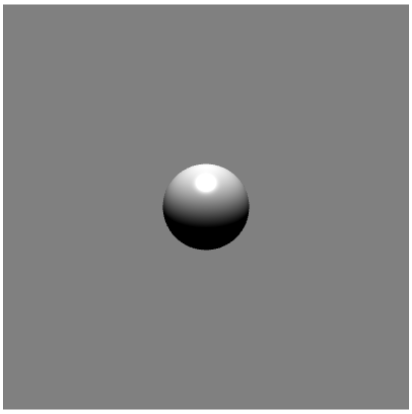


Figure 1.1.

There exists a such method for each common elementary geometric shape, and the number and the type of the parameters vary case by case; *because each parameter has a default value, you have to specificate just the values that you want explicity modify*.

That's it!

You can find all the available elementary geometric shapes and their parameters at this source:

http://doc.babylonjs.com/tutorials/Mesh_ CreateXXX_ Methods_ With_ Options_ Parameter

They are a lot, but the common working principle is just the above explained one[1.1].

1.2. How to transform (rotate, translate or scale) a mesh.

The transformation mechanism of `BabylonJS` is one of the least clear aspects provided by this library.

There are a lot of methods, sometimes redoundant and also not much intuitive.

So in this manual you'll see just a minimal set of these methods, but it is enough for most applications.

You surely know that the transformation operations are realted to a frame.

There are three types of frame:

- the *global world frame*, that consists in the fundamental fixed reference frame.

  If non transformation is applied, then the mesh is put in the origin of this frame, with a default orientation.

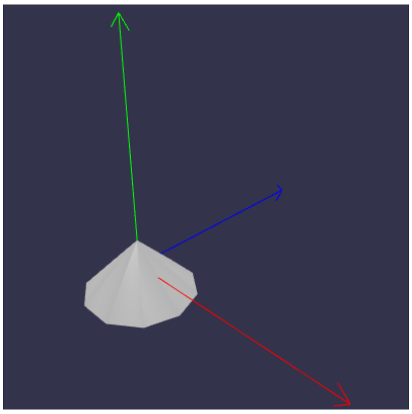


Figure 1.2.

- the *local frames*.

  If to a mesh are applied one or more roto–translations, it will no more stay onto the origin of the *global world frame*.

  The frame such that *(a)* its position is the same of the geometric shape but *(b)* inehirts its orientation from the *global world frame* is named *local world frame*.

  The frame such that both its position and its orientation are the same of the geometric shape is named *local frame*.

1.1. In particular pay attention to the `tessellation` parameter for shapes like the cylinder, the disk, the torus, others: it permits to draw regular polynomial shapes, and not only circular ones.

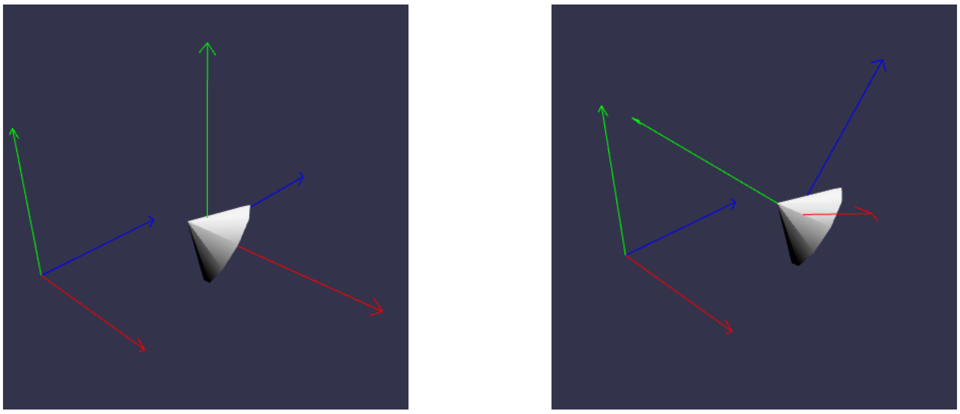


Figure 1.3. *local world frame* (left) and *local frame* (rigth)

In order to transform an object you'll have to follow these (optional) steps *strictly in the following order*[1.2]: *(1)* translate w.r.t. the *global world frame*, *(2)* rotate w.r.t. the *local world frame*, *(3)* translate w.r.t. the *local frame*, *(4)* rotate w.r.t. the *local frame*, *(5)* scale w.r.t. the *local frame*.

You can translate w.r.t. the *global world frame* by setting the direction along which you want translate and the entity of the translation

```
cone.translate(new BABYLON.Vector3(0,1,0), 1.5, BABYLON.Space.WORLD);
```

You can rotate w.r.t. the *local world frame* by setting the direction around which you want rotate and the entity of the rotation

```
cone.rotate(new BABYLON.Vector3(1,0,0), Math.PI/4, BABYLON.Space.WORLD);
```

You can translate w.r.t. the *local frame* by setting the translation vector

```
cone.locallyTranslate(new BABYLON.Vector3(0,1.2,0));
```

You can rotate w.r.t. the *local frame* by setting the direction around which you want rotate and the entity of the rotation

```
cone.rotate(new BABYLON.Vector3(1,0,0), Math.PI, BABYLON.Space.LOCAL);
```

You can scale w.r.t. the *local frame* by just setting the scaling factors

```
cone.scaling = new BABYLON.Vector3(1,3,1);
```

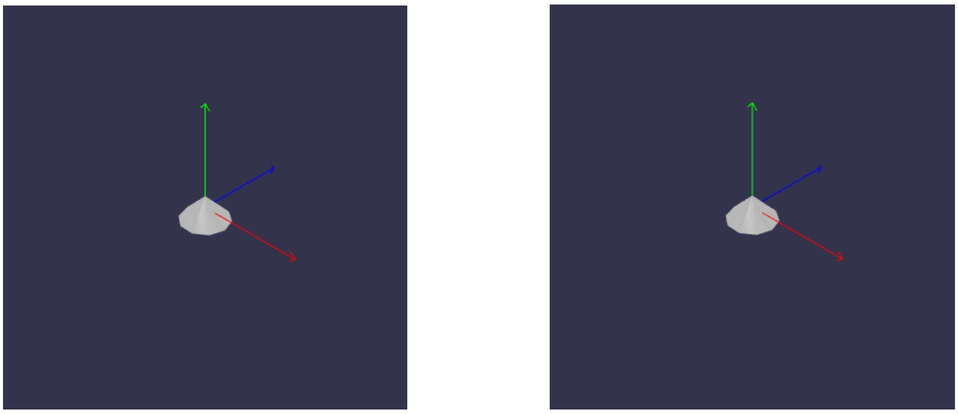
How you can see, there's not a syntactic regularity!

In the following figure the evolution of the transformation is represented[1.3].

1.2. If you want to alter the order you should understand more technical details about the transformation mechanisms in `BabylonJS`.

But this is not necessary, because by following this order it's possible to do everything.

1.3. In addition in this figure are represented the reference frames, that should be builded apart.



```
var cone = BABYLON.MeshBuilder.CreateCylinder(
                                       "cone",
                                       {height: 0.5, diameterTop: 0, tessellation: 10},
                                       scene
                                    );
```

```
cone.translate(new BABYLON.Vector3(0,1,0), 1.5, BABYLON.Space.WORLD);
```

```
cone.rotate(new BABYLON.Vector3(1,0,0), Math.PI/4, BABYLON.Space.WORLD);
```

1.3. How to transform (rotate, translate or scale) a hierarchical objects.

You can combine two meshes with a particular parent–child relationship: *the global world frame of the child is the local world frame of the parent.*

This relationship is *one to many*, namely *each mesh can have an only parent but many children*.

From a pratical point of view, this means that *each transformation applied to a parent is automatically applied to its children*, but not viceversa.

A set of two or more meshes such that each mesh is constrained by at least a parent–child relashionship is named hierarchical object.

`BabylonJS` makes really simple to build hierarchical objects, because it permits to set a parent–child relationship by just setting the `parent` property of a mesh:

```
meshChild.parent = meshParent;
```

Pay attentipn to the fact that *even any transformation made to the parent prior to assigning it to children will also be applied to the children*.

You can verify if a mesh is a descendant of an other one by using the method:

```
meshPotentialDescendant.isDescendantOf(meshPotentialAncestor);
```

Pay attention to the fact that a relationship parent–child implies a relationship ancestor–descendant, but not viceversa.

You can get all the descendants of a mesh – in the array format – by using the method:

```
meshAncestor.getDescendants();
```

Pay attention to the fact that a relationship parent–child implies a relationship ancestor–descendant, but not viceversa.

1.4. How to clone a hierarchical object.

`BabylonJS` permits to create a copy of each mash by just using this method:

```
meshClone = meshOrinigal.clone("clone");
```

In order to clone a hierarchical object you have just to clone the ancestor mesh:

```
meshAncestorClone = meshAncestorOriginal.clone("ancestorClone");
```

in fact also their descendants will be automatically cloned.

In order to manipulate them you can extract them by using the `getDescendants()` method onto `meshAncestorClone`.

1.5. How to remove a hierarchical object.

`BabylonJS` permits to remove each mash by just using this method:

```
meshToRemove.dispose();
```

In order to remove a hierarchical object you have just to remove the ancestor mesh:

```
meshAncestorToRemove.dispose();
```

in fact also their descendants will be automatically removed.

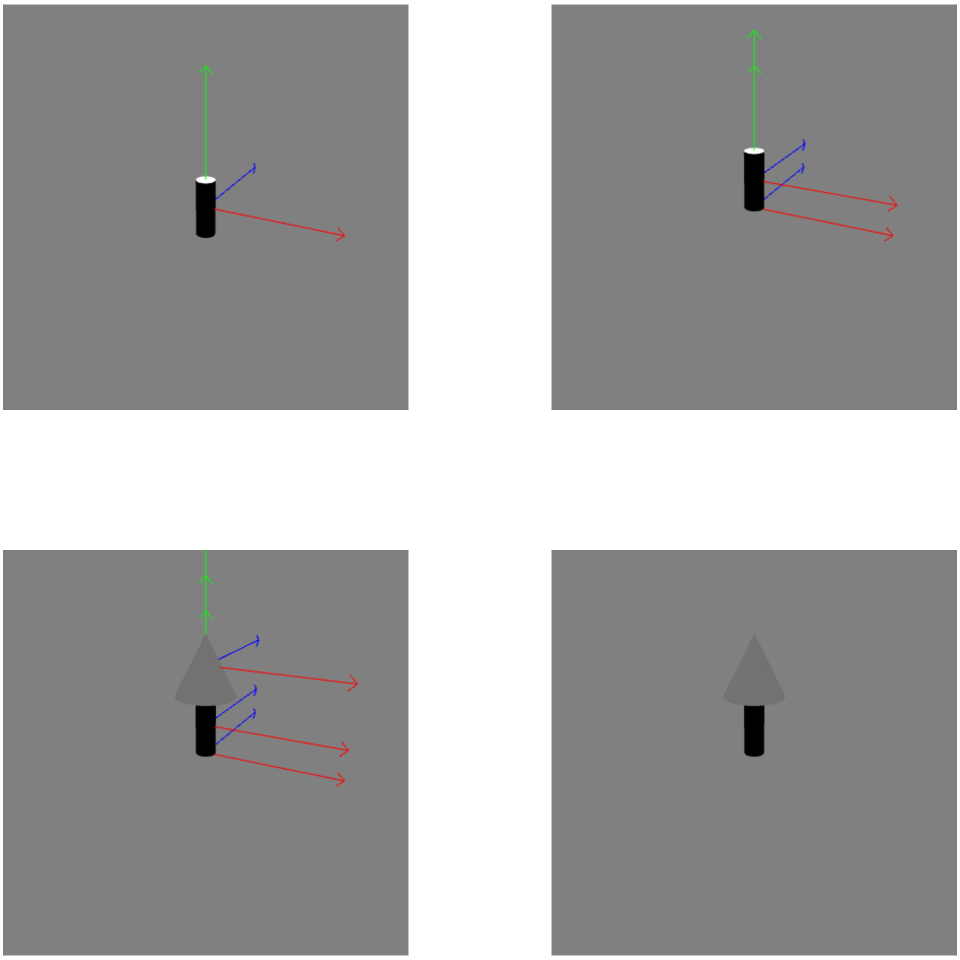
1.6. Check Point: a sample code.

Let's apply these instructions to the *Structure Layer* section of the basical structure presented in the *Prologue*:

```
// Setting the Structure Layer
var createTree = function(){
    var cylinder = BABYLON.MeshBuilder.CreateCylinder(
                                                    "cylinder",
                                                    {height: 3},
                                                     scene
                                                     );
    cylinder.translate(new BABYLON.Vector3(0,1,0), 1.5, BABYLON.Space.WORLD); //1.5=3/2

    var cone = BABYLON.MeshBuilder.CreateCylinder(
                                                    "cone",
                                                     {height: 3,
                                                     diameterTop: 0,
                                                     diameterBottom: 3},
                                                     scene
                                                    );
     cone.parent = cylinder;
     cone.translate(new BABYLON.Vector3(0,1,0), 3, BABYLON.Space.WORLD); //3=3/2+3/2

     return cylinder;
};

var tree = createTree();

var treeClone = tree.clone("clone"); // Recursive clonation
treeClone.translate(new BABYLON.Vector3(1,0,0), 4, BABYLON.Space.WORLD);
treeClone.getDescendants().pop().scaling = new BABYLON.Vector3(1,2,1);

tree.dispose(); // Recursive disposing
```

In the following figure the evolution of the the code is represented[1.4].

1.4. In addition in this figure are represented the reference frames, that should be builded apart.



Execution of `createTree()`.
Top left (*world global frame*).
`var cylinder = BABYLON.MeshBuilder.CreateCylinder("cylinder",{height: 3}, scene);`
Top right (*world global frame* and *cylinder world local frame*).
`cylinder.translate(new BABYLON.Vector3(0,1,0), 1.5, BABYLON.Space.WORLD); // 1.5 = 3/2`
Buttom left (*world global frame*, *cylinder world local frame* and *cone world local frame*).

```
var cone = BABYLON.MeshBuilder.CreateCylinder(
                                                  "cone",
                                                   {height: 3,
                                                   diameterTop: 0,
                                                   diameterBottom: 3},
                                                   scene
                                                  );
     cone.parent = cylinder;
     cone.translate(new BABYLON.Vector3(0,1,0), 3, BABYLON.Space.WORLD); //3=3/2+3/2
```

Buttom right.
The actual result.