Chapter 1

The Motion Layer.

1.1. How to make animations.

An animation permits to change the value of an object property dynamically, according to a completely definite mathematical law *having a limited timing domain* $[0, T]$.

`BabylonJS` provides two different ways to make animations, by means the `Animation` class tool and by means the `scene.beforeRender` method.

1.1.1. The `Animation` class technique.

1.1.1.1. Description of the technique.

In `BabylonJS` you can define an animation in four steps:

I. Creating an `Animation` instance, with an instruction having the following scheme:

```
var animation = new BABYLON.Animation(
                                      "animation",
                                      "property",
                                      30,
                                      BABYLON.Animation.ANIMATIONTYPE_XXX,
                                      BABYLON.Animation.ANIMATIONLOOPMODE_XXX
                                      );
```

The first argument is just the name of the instance, the second one is the property name that you want to change, the third one is the maximum frame–per–second admitted (you can consider it fixed to 30).

The fourth one is the `BabylonJS` class related to the property, and the possible suffixes are `_FLOAT`, `_VECTOR2`, `_VECTOR3`, `_QUATERNION`, `_MATRIX` and `_COLOR3`.

The fifth one is a very useful option that permits to choose what to do when the animation ends (i.e. at time $t = T$), in particular you can choose whether *(a)* to remain into the final state, with the suffix `_CONSTANT`, or *(b)* to repeat the animation *in an absolute way – i.e. by overwriting the final state with the initial one, and then repeating the animation periodically –* with the suffix `_CYCLE`, or *(c)* to repeat the animation *in an incremental way – i.e. by not overwriting the final state but by adding to it the effects of each new repetition of the animation –*, with the suffix `_RELATIVE`.

II. Defining the mathematical law.

Actually you doesn't define the whole timing law, but you just specify a set of points, named *keys*; then *`BabylonJS` will automatically build the continuous law by means linear interpolation, i.e. the resulting timing law will be the polygonal chain*[1.1] *connecting the desired keys.*

[1.1]. For more informations please refer to

https://en.wikipedia.org/wiki/Polygonal_ chain

A sample code is the following one:

```
var keys = [];
keys.push({frame: 0, value: new BABYLON.Vector3(0, 0, 0)});
keys.push({frame: 50, value: new BABYLON.Vector3(3, 0, 0)});
keys.push({frame: 100, value: new BABYLON.Vector3(1.5, 0, 0)});
animation.setKeys(keys);
```

You can note that each key is (obviously) a couple of two instances: the `frame` one, that refers to the generic time $t \in [0, T]$, and the `value` one, that refers to the value of the property at the time $t =$ `frame`.

In this case the considered property belongs to the `Vector3` class, and T is is equal to 100.

III. Associating the animation(s) to the desired object.

A sample code is the following one

```
mesh.animations = [animation];
```

You can create more than one animation for the mesh, and you can put all of them into the `mesh.animations` array.

But remember that you can define just an animation per property, and if you'll add two animations referring to the same property only the last added one will be considered.

The `animation` property is inherited by the `Node` class, that is the parent class for all scene objects (e.g., `Mesh`, `Material`, `Texture`, `Light`, `Camera`, `Sprite`, `Scene`). *So you can define an animation for everything.*

IV. Starting the animation(s).

A sample code is the following one

```
scene.beginAnimation(mesh, 0, 100, true);
```

The first argument is the object for which you want to start *synchronously* the associated animations (that you put into its `associations` property at the previous step).

The second and the third ones refer to the `frame` values that have to be considered like $t = 0$ and $t = 100$.

The fouth one enables (or not) the `ANIMATIONLOOPMODE_XXX` option of the `BABYLON.Animation` constructor. If its value is equal to false, at the end of the animation the property will remain to its final state (reached at $t = T$)[1.2].

In the following table the most common properties are enumered:

1.2. That is the same result to put this value equal to `true` and using the `ANIMATIONLOOPMODE_CONSTANT` option above.

Property	**Description**	`BabylonJS` **class suffix**
`mesh.position`	Permits to set the position w.r.t. the *world local frame*. Therefore it can be used by an animation for simulate a translational movement.	`_VECTOR3`
`mesh.rotation`	Permits to set the orientation w.r.t. the *world local frame*. Therefore it can be used by an animation for simulate a rotational movement.	`_VECTOR3`
`mesh.scaling`	Permits to set the scaling factor vectors w.r.t. the *world local frame*. Therefore it can be used by an animation for simulate a scaling transformation.	`_VECTOR3`
`mesh.material.emissiveColor`	Permits to set $\mathbf{c}^{\text{emis}}$	`_COLOR3`
`mesh.material.ambientColor`	Permits to set $\boldsymbol{\rho}^{\text{refl,diff,amb}}$	`_COLOR3`
`mesh.material.diffuseColor`	Permits to set $\boldsymbol{\rho}^{\text{refl,diff}}$	`_COLOR3`
`mesh.material.specularColor`	Permits to set $\boldsymbol{\rho}^{\text{refl,spec}}$	`_COLOR3`
`sprite.position`	Permits to set the position w.r.t. the *world global frame*. Therefore it can be used by an animation for simulate a translational movement.	`_VECTOR3`
`sprite.angle`	Permits to set the orentation (w.r.t. the only degree of freedom permitted). Therefore it can be used by an animation for simulate a rotational movement.	`_FLOAT`
`sprite.width`	Permits to the set the width.	`_FLOAT`
`sprite.height`	Permits to the set the height.	`_FLOAT`
`scene.ambientColor`	Permits to set $\mathbf{c}^{\text{light,amb}}$	`_COLOR3`
`light.diffuse`	Permits to set $\mathbf{c}^{\text{light},\{\text{dir,point,spot}\}}$ when the diffuse reflection is considered	`_COLOR3`
`light.specular`	Permits to set $\mathbf{c}^{\text{light},\{\text{dir,point,spot}\}}$ when the specular reflection is considered	`_COLOR3`
`directionalLight.direction`	Permits to set the directional light direction. Therefore it can be used by an animation for simulate the sun movement.	`_VECTOR3`
`directionalLight.position`	Permits to set the directional light position. Therefore it can be used by an animation for simulate a translational movement.	`_VECTOR3`
`pointLight.position`	Permits to set the point light position. Therefore it can be used by an animation for simulate a translational movement.	`_VECTOR3`
`spotLight.position`	Permits to set the spot light position. Therefore it can be used by an animation for simulate a translational movement.	`_VECTOR3`
`spotLight.direction`	Permits to set the spot light main direction.	`_VECTOR3`
`spotLight.angle`	Permits to set the spot light cut–off angle.	`_FLOAT`

Table 1.1.

1.1.1.2. How to attach events to the animation.

You can add an event to an animation with the following instruction:

```
animation.addEvent(new BABYLON.AnimationEvent(
                                                50,
                                                function() { /* ... */ },
                                                true
                                             ));
```

The first argument is the frame of the animation at which you want to associate the event; *experimentally we noted that you cannot associated an event to the last frame.*

The second argument is the function that will be executed when the considered frame occurs.

The third argument establishes if the animation has to be executed only once or not *(...)*.

1.1.1.3. Check Point: a Sample Code

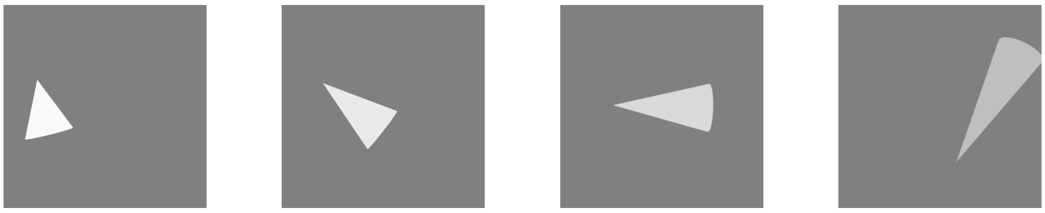
Let's consider the following code.

It applies to a cone four different animations, related to the properties `position`, `rotation`, `scaling` and `material.ambientColor`.

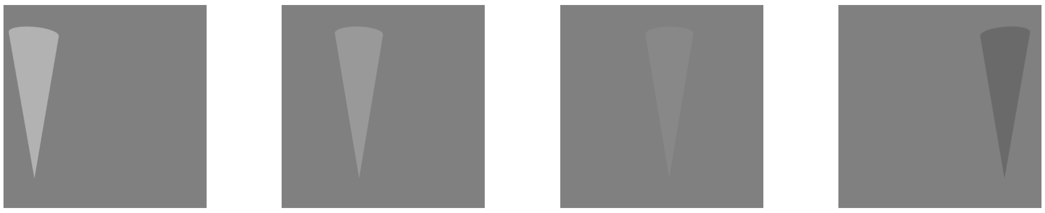
```
// Setting the Structure Layer
var cone = BABYLON.MeshBuilder.CreateCylinder("cone", {height: 2, diameterTop: 0,
diameterBottom: 2}, scene);

// Setting the Appearance Layer
scene.clearColor = new BABYLON.Color3(0.5, 0.5, 0.5);
var lights = setLights();
cone.material = new BABYLON.StandardMaterial("material", scene);

// Setting the Motion Layer
// 1 of 4
var animationTranslation = new BABYLON.Animation(
                                "animationTranslation",
                                "position",
                                30,
                                BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
                                BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE
                        );
var keysTranslation = [];
keysTranslation.push({frame: 0, value: new BABYLON.Vector3(-3, 0, 0)});
keysTranslation.push({frame: 100, value: new BABYLON.Vector3(3, 0, 0)});
animationTranslation.setKeys(keysTranslation);

// 2 of 4
var animationRotation = new BABYLON.Animation(
                                "animationRotation",
                                "rotation",
                                30,
                                BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
                                BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT
                        );
var keysRotation = [];
keysRotation.push({frame: 0, value: new BABYLON.Vector3(0, 0, 0)});
keysRotation.push({frame: 100, value: new BABYLON.Vector3(0, 0, Math.PI)});
animationRotation.setKeys(keysRotation);

// 3 of 4
```

```
var animationScaling = new BABYLON.Animation(
                                "animationScaling",
                                "scaling",
                                30,
                                BABYLON.Animation.ANIMATIONTYPE_VECTOR3,
                                BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT
                        );
var keysScaling = [];
keysScaling.push({frame: 0, value: new BABYLON.Vector3(1, 1, 1)});
keysScaling.push({frame: 100, value: new BABYLON.Vector3(1, 3, 1)});
animationScaling.setKeys(keysScaling);

// 4 of 4
var animationAmbientColor = new BABYLON.Animation(
                                "animationAmbientColor",
                                "material.ambientColor",
                                30,
                                BABYLON.Animation.ANIMATIONTYPE_COLOR3,
                                BABYLON.Animation.ANIMATIONLOOPMODE_RELATIVE
                        );
var keysAmbientColor = [];
keysAmbientColor.push({frame: 0, value: new BABYLON.Color3(1, 1, 1)});
keysAmbientColor.push({frame: 100, value: new BABYLON.Color3(0.7, 0.7, 0.7)});
animationAmbientColor.setKeys(keysAmbientColor);

// go!
cone.animations = [animationTranslation, animationRotation, animationScaling,
animationAmbientColor];
scene.beginAnimation(cone, 0, 100, true);

// Setting the User Interaction Layer
var camera = setCamera();
// ...
```

In the following figure the evolution of the the code is represented.

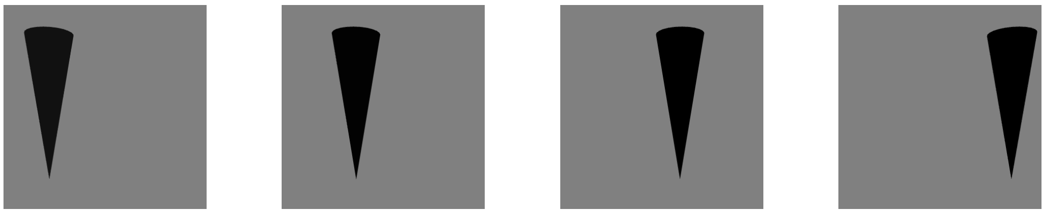


At the first loop all the animation are performed: the cone translates horizontally,
rotates counter–clockwise, scales vertically and changes its color from $\{1, 1, 1\}$ to $\{0.7, 0.7, 0.7\}$.



At the second loop the cone ends to rotate and scale because these animation are set with
the `ANIMATIONLOOPMODE_CONSTANT` option, but restart to trnaslate from the initial position,
because the translation animation is set to the the `ANIMATIONLOOPMODE_CYCLE` option.
The cone becomes increasingly dark, because at this loop its color changes from $\{0.7, 0.7, 0.7\}$ to
$\{0.7^2, 0.7^2, 0.7^2\} = \{0.49, 0.49, 0.49\}$; in fact the color animation is set with
the `ANIMATIONLOOPMODE_RELATIVE`.

At the third loop we have the same processes of the previous one.
In particular the cone becomes increasingly dark, because at this loop its color
changes from $\{0.7^2, 0.7^2, 0.7^2\}$ to$\{0.7^3, 0.7^3, 0.7^3\} = \{0.343, 0.343, 0.343\}$;
in fact the color animation is set with the `ANIMATIONLOOPMODE_RELATIVE` option.



At the third loop we have the same processes of the previous one.
In particular the cone becomes increasingly dark, because at this loop its color
changes from $\{0.7^3, 0.7^3, 0.7^3\}$ to$\{0.7^4, 0.7^4, 0.7^4\} = \{0.2401, 0.2401, 0.2401\}$;
in fact the color animation is set with the `ANIMATIONLOOPMODE_RELATIVE` option.

And so on...

Figure 1.1.
Execution steps of the above code.

1.1.2. The `scene.beforeRender` method technique.

1.1.2.1. Description of the technique.

In order to make animations, `BabylonJS` automatically updates frequently the content of the `canvas`.

Before doing an update it executes the instructions written in the anonymous function `scene.beforeRender`.

Therefore *you can exploit this function in order to make an animation.*

Unlike the previous techniques this one *(a)* permits to execute even methods (so not only to change properties values) but *(b)* hasn't got the powerful tools of the `ANIMATIONLOOPMODE_XXX` option and the keys for defining the timing law.

In order to remedy this problem you need to handle the timing law manually. First of all you have to count manually the frame index by means a global counter variable, let it `frame`.

```
scene.beforeRender = function () {
        // instructions
        frame++;
};
```

In order to execute a function you have just do this:

```
scene.beforeRender = function () {
        if( frame == 50 ){
                functionToExecute();
        }
        frame++;
};
```

You can also execute the three types of animations in this method discussed above, in particular

- a periodical absolute animation can be realized by following this code:

```
scene.beforeRender = function () {
   var f = frame%100;
   if( f>=0 && f<50 ){
      mesh.position = BABYLON.Vector3.Lerp(
                               new BABYLON.Vector3(0,0,0),
                               new BABYLON.Vector3(3,0,0),
                               f/50);
   } else if( f>=50 && f<100 ){
      mesh.position = BABYLON.Vector3.Lerp(
                               new BABYLON.Vector3(3,0,0),
                               new BABYLON.Vector3(0,0,0),
                               (f-50)/50));
   }

   frame++;
};
```

- a periodical relative animation can be realized by following this code:

```
scene.beforeRender = function () {
   var f = frame%100;

   // Preparing the initial and the final value
   var startAmbientColor = new BABYLON.Color3(1,1,1);
   var endAmbientColor = new BABYLON.Color3(0.7,0.7,0.7);
   var actionAmbientColor = function(){
       startAmbientColor = startAmbientColor.multiply(
                                    new BABYLON.Color3(0.7,0.7,0.7));
       endAmbientColor = startAmbientColor.multiply(
                                    new BABYLON.Color3(0.7,0.7,0.7));
   };
   for(var i=0; i<Math.floor(frame/100); i++){
      actionAmbientColor();
   }
```

```
        // Updating the value
        mesh.material.ambientColor = BABYLON.Color3.Lerp(
                                          startAmbientColor,
                                          endAmbientColor,
                                          f/100
                                          );

        frame++;
     };
```

– an animation that preserves its final value can be realized by following this code:

```
     scene.beforeRender = function () {
        var f = frame%100;

        var startAmbientColor = new BABYLON.Color3(1,1,1);
        var endAmbientColor = new BABYLON.Color3(0.7,0.7,0.7);
        var actionAmbientColor = function(){
            startAmbientColor = startAmbientColor.multiply(
                                              new BABYLON.Color3(0.7,0.7,0.7));
            endAmbientColor = startAmbientColor.multiply(
                                              new BABYLON.Color3(0.7,0.7,0.7));
        };

        for(var i=0; i<Math.floor(frame/100); i++){
           actionAmbientColor();
        }
        mesh.material.ambientColor = BABYLON.Color3.Lerp(
                                          startAmbientColor,
                                          endAmbientColor,
                                          f/100
                                          );

        frame++;
     };
```

1.1.2.2. Check Point: a Sample Code

The following code gives the same result of the previous one.

```
   // Setting the Structure Layer
   var cone = BABYLON.MeshBuilder.CreateCylinder("cone", {height: 2, diameterTop: 0,
diameterBottom: 2}, scene);

   // Setting the Appearance Layer
   scene.clearColor = new BABYLON.Color3(0.5, 0.5, 0.5);
   var lights = setLights();

   cone.material = new BABYLON.StandardMaterial("materialCone", scene);

   // Setting the Motion Layer
   scene.beforeRender = function () {

           var f = frame%100;

           // 1 of 4
           if( f>=0 && f<100 ){
           cone.position = BABYLON.Vector3.Lerp(
                                   new BABYLON.Vector3(-3,0,0),
                                   new BABYLON.Vector3(3,0,0),
                                   f/100);
           }

           // 2 of 4
           if( frame < 100 ){
                   cone.rotation = BABYLON.Vector3.Lerp(
                                                new BABYLON.Vector3(0, 0, 0),
                                                new BABYLON.Vector3(0, 0, Math.PI),
                                                frame/100
                                                );
           }

           // 3 of 4
           if( frame < 100 ){
                   cone.scaling = BABYLON.Vector3.Lerp(
                                                new BABYLON.Vector3(1, 1, 1),
                                                new BABYLON.Vector3(1, 3, 1),
                                                frame/100
                                                );
           }

           // 4 of 4
           var startAmbientColor = new BABYLON.Color3(1,1,1);
           var endAmbientColor = new BABYLON.Color3(0.7,0.7,0.7);
           var actionAmbientColor = function(){
           startAmbientColor = startAmbientColor.multiply(
                                                new BABYLON.Color3(0.7,0.7,0.7));
           endAmbientColor = startAmbientColor.multiply(
                                                new BABYLON.Color3(0.7,0.7,0.7));
           };

           for(var i=0; i<Math.floor(frame/100); i++){
                   actionAmbientColor();
           }
           cone.material.ambientColor = BABYLON.Color3.Lerp(
                                   startAmbientColor,
                                   endAmbientColor,
                                   f/100
                                   );

           frame++;
   };

   // Setting the User Interaction Layer
   var camera = setCamera();
   // ...
```

1.1.3. How to synchronize the methods.

Sometimes you would like use both the above methods, in order to exploit the bestt advantages of both for different animation.

You can do it but *you cannot perfectly syncronize the two group of animations, and the error increases linearly over time (at least, currently all the synchronization techniques that I tried have not been successful).*

1.1.4. The geometrical intersection detection mechanisms.

When two or more meshes move and/or scale by following an animation scheme, their volumes could intersect one with the other.

In order to detect and react to this occurrence you can use the `intersectsMesh` method, like it is shown in the following code.

```
if (mesh1.intersectsMesh(mesh2, false)) {
    //...
} else {
    //...
}
```

To avoid costly calculation by checking many details on a mesh, `BabylonJS` creates a *bounding box* around the object, and *tests for intersection between this box and the colliding mesh (i.e.* `mesh2`*).*

Here is an example of a bounding box:

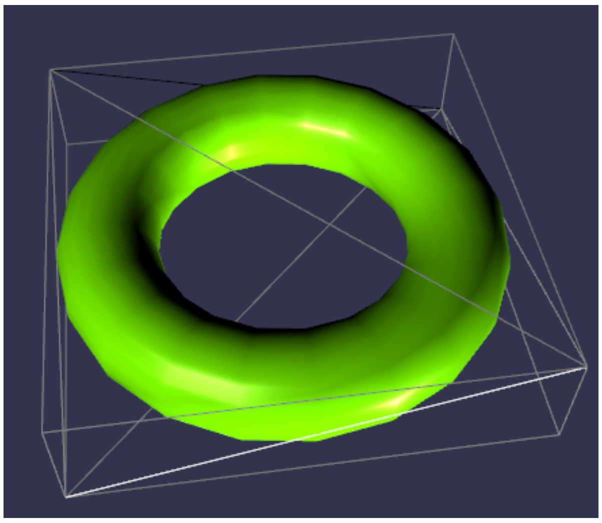


Figure 1.2.

But this *bounding box* can be more or less precise, and that's why we have our second parameter.
In short, if this parameter is set to true (false by default), then the bounding box is closer to the mesh, but it's a more costly calculation.

In order to detect instead if a position in the space is occupied by the volume of a mesh you can use the `intersectsPoint` method, like it is shown in the following code.

```
if (mesh1.intersectsPoint(new BABYLON.Vector3(6, 0, 0))) {
    //...
} else {
    //...
}
```

In this case there aren't options about precision.

1.1.4.1. Check Point: a Sample Code

Let's consider the following code.
It creates a scene with two boxes *a)* arranged in $\{9, 0, 0\}$ and $\{-9, 0, 0\}$ and *b)* that are moving one towards the other.
The box starting at the right disappears as soon as the intersection between the boxes occurs, but it will reappear when the boxes will be no longer intersected.
Furthermore the box starting at left disappears as soon as the intersection between it and the point $\{6, 0, 0\}$ occurs, but it will reappear when the box and the point will be no longer intersected.

```
// Setting the Structure Layer
var boxRight = BABYLON.MeshBuilder.CreateBox("boxRight", {size: 3}, scene);
boxRight.translate(new BABYLON.Vector3(1,0,0), 9, BABYLON.Space.WORLD);

var boxLeft = BABYLON.MeshBuilder.CreateBox("boxLeft", {size: 3}, scene);
boxLeft.translate(new BABYLON.Vector3(1,0,0), -9, BABYLON.Space.WORLD);

// Setting the Appearance Layer
scene.clearColor = new BABYLON.Color3(0.5, 0.5, 0.5);
var lights = setLights();

// Setting the Motion Layer
scene.beforeRender = function(){
        // Update positions
        boxRight.position = boxRight.position.add(new BABYLON.Vector3(-0.1,0,0));
        boxLeft.position = boxLeft.position.add(new BABYLON.Vector3(0.1,0,0));

        // Geometrical intersections detection
        if (boxRight.intersectsMesh(boxLeft, false)) { // commutative
                boxRight.visibility = 0;
        } else {
                boxRight.visibility = 1;
        }

        if (boxLeft.intersectsPoint(new BABYLON.Vector3(6, 0, 0))){
                boxLeft.visibility = 0;
        } else {
                boxLeft.visibility = 1;
        }
}

// Setting the User Interaction Layer
var camera = setCamera();
```

In the following figure the evolution of the the code is represented.

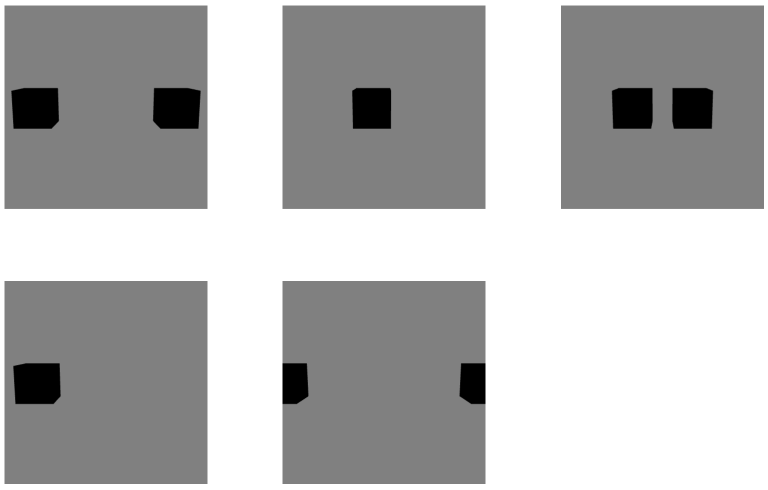


Figure 1.3.
Execution steps of the above code.
From the left to the right: initial configuration, the box starting at the right disappears, the box starting at the right (now at the left) reappears, the box starting at the left disappears, the box starting at the left reappears.

1.2. How to simulate the physical phenomena.

1.2.1. An essential, qualitative theoretical background.

1.2.1.1. The second Newton's law.

The dynamics of a *point–like particle (i)* having fixed mass m and *(ii)* subject to a resultant force $\boldsymbol{f}$ is described by the following law: $\boldsymbol{f}(t) = m\frac{d\boldsymbol{v}(t)}{dt}$.

Therefore *a certain particle (i) having velocity $\boldsymbol{v}_0$ and (ii) subject to a zero resultant force tends to preserve its (constant) velocity*, in fact

$$\begin{aligned} m\frac{d\boldsymbol{v}(t)}{dt} = \mathbf{0} &\Rightarrow \int_{t_0}^{t} \boldsymbol{v}(\tau)d\tau = \int_{t_0}^{t} \mathbf{0}\, d\tau \Rightarrow \\ &\Rightarrow \boldsymbol{v}(t) - \boldsymbol{v}(t_0) = \mathbf{0} \Rightarrow \\ {}_{\boldsymbol{v}(t_0)=\boldsymbol{v}_0} \quad & \boldsymbol{v}(t) = \boldsymbol{v}_0 \end{aligned}$$

By extending this reasoning to a rigid body we can reach an equivalent extended result, according to which *a rigid body (i) having mass m and (ii) subject to a zero resultant force and to a zero resultant torque tends to preserve its (constant) linear and angular velocities, namely tends to preserve its roto–translational (constant) motion.*

1.2.1.2. Collisions.

A *collision* is an event in which two or more bodies exert forces on each other for a relatively short time. In a very general and qualitative way, the consequences of a collision depend on the physical state (positions, velocities, accelerations, etc.), on the form and on the materials of the considered bodies.

In particular the *coefficient of restituzion*[1.3] characterizing the bodies' materials is very relevant parameter needed to model these events.

1.2.1.3. (Nonimpulsive) forces and impulsive forces.

An *impulsive force* is a force *exerting in an infinitesimal time – zero to the limit – with an extremely large intensity – infinite to the limit*. It is very useful to model – for example – strikes and clashes.

A *nonimpulsive force* (commonly just named *force*) is a force exerting in a finite time with a finite, in general varying intensity. It is very useful to model a very large set of phenomena, like the gravitational attraction.

1.2.1.4. Constraints.

Prismatic and revolute joints. Outstanding.

1.2.2. Implementation in `BabylonJS`.

1.2.2.1. The physics engine.

In order to simulate mechanical phenomena into the scene `BabylonJS` permits to integrate a third–part plug–in, named *physics engine*.

Essentially *a physics engine models phenomenta like gravitational attraction, friction, collisions and generic forces (both impulsive and not) exertion, in rispect of some physical constraints that you can define for group of bodies*.

In order to enable the physics engine in `BabylonJS` by first you have to add this instruction:
`scene.enablePhysics(new BABYLON.Vector3(0,-9.81,0), new BABYLON.XXXJSPlugin());`

The first argument is the gravity vector, that in this case is equal to the Earth's one. You can change the scene's gravity – even after you have yet set the gravity above – using the `scene.setGravity()` method.

The second argument permits to choose which physics engine you want to use; currently `BabylonJS` permits to chooce between two different physycs engine, `Oimo.js` and `cannon.js`[1.4], respectively using the prefixes `Oimo` and `Cannon`.

Note.

Here is a `Cannon` *developer's quotation:*

> *"The equations used in game physics are not exact. They usually use iterative solvers that can solve the equations more and more precise depending on how much CPU power you are willing to pay.*
>
> *Also, the equation are only solvable "in one direction", you can't really input a time value and expect the engine to give you the friction constant.*
>
> *It's optimized to only output game object transforms.*
>
> *What you can do is run the simulation many times until you find a value for the friction that matches your time value."* [1.5]

Do you want the body falls faster? Increase the mass value!
Do you want a sliding body decelerates slowlier onto a platform? Decrease the coefficient of friction!
Do you want a body bounces more? Increase the coefficient of restitution!
Do you want to know how much to increase the coefficient of friction of a sliding body in order to stop it in 5.46366 *seconds? This is not the appropriate tool for doing this! You can try empirically different values to get a result that is convenientely likely to the* <u>*exact*</u> *one.*

Nevertheless you can read about the equations ised by `Cannon` at this *URL* (June 2016)
http://www8.cs.umu.se/kurser/5DV058/VT15/lectures/SPOOKlabnotes.pdf

1.2.2.2. The impostors.

A mesh is a geometrical entity. In order to it a physical body and to model it by means mechanichs laws, you need to assign it the following physical properties: the initial linear and angular velocities, the mass, the cofficient of restitution and the coefficient of friction.

1.3. For more informations please refer to:
http://www.science.unitn.it/~fisica1/fisica1/appunti/mecc/appunti/cinematica/urti/node9.html
https://en.wikipedia.org/wiki/Coefficient_of_restitution

1.4. If you want know what are the main differences about them please refer to other sources.

1.5. Source (June 2016):
http://www.html5gamedevs.com/topic/22996-what-are-physical-equations-used-by-cannonjs-and-how-to-map-them-onto-a-babylon-code/

These physical properties are organized into the `PhysicsImpostor` class, that in turn is a property of the considered mesh.

So in order to set these five physical properties to an object you have:

I. firstly to build an instance of `PhysicsImpostor`

```
mesh.physicsImpostor = new BABYLON.PhysicsImpostor(
                                                    mesh,
                                                    BABYLON.PhysicsImpostor.XXXImpostor,
                                                    {},
                                                    scene
                                                    );
```

The first parameter is the considered mesh.

The second parameter has to refer to the type of the considered mesh, for example `SphereImpostor` con be used for spherical meshes, `BoxImpostor` for planar and cubic ones[1.6], `CylinderImpostor` for cyinderical and conical meshes; you can find all the possibilities into the documentation[1.7].

We won't use the third parameter.

The fourth parameter is the scene.

II. and than setting the five values

```
mesh.physicsImpostor.setLinearVelocity(new BABYLON.Vector3(4,0,0));
mesh.physicsImpostor.setAngularVelocity(new BABYLON.Vector3(0,-0.2,0));
mesh.physicsImpostor.setParam("mass", 1);
mesh.physicsImpostor.setParam("friction", 0.1);
mesh.physicsImpostor.setParam("restitution", 0.8);
```

Both the initial linear velocity and the inital angular one are expressed w.r.t. the *local global frame*, where as the mass and the two coefficients are (obviously) scalar quantities.

The impostors mechanism permits to handle automatically the collisions, but you can react conveniently to a collision by means the following instruction:

```
mesh1.physicsImpostor.registerOnPhysicsCollide(
                                                mesh2.physicsImpostor,
                                                function() {
                                                                // e.g., mesh1.visibility = 0.5;
                                                            }
                                                );
```

This operation is commutative, namely you can interchange `mesh1` and `mesh2` without changing the final result.

Pay attention: experimentally we noted that the callback function will be executed more times for each collision, and not just once. So this function has to implement a combinatory logic, and not a sequential one!

1.2.2.3. How to apply additional forces.

Even if the physics engine models automatically the forces related to the gravity, the friction and the collisions, can exert additional forces onto a body with by using the `scene.beforeRender` function:

```
scene.beforeRender = function () {
        // Exerting an additional impulsive force
        if( frame==50 ){
                mesh.physicsImpostor.applyImpulse(
                                new BABYLON.Vector3(0, 10, 0),
                                new BABYLON.Vector3(0, 0, 0)
                                );
        }
        // Exerting an additional non-impulsive force for 10 time units (i.e. frames)
        if( frame>=50 && frame<=60 ){
                mesh.physicsImpostor.applyImpulse(
                                new BABYLON.Vector3(0, 10, 0),
                                new BABYLON.Vector3(0, 0, 0)
```

1.6. Even if there exists `PlaneImpostor`, it doesn't work conveniently.

1.7. Here (june 2016):
http://doc.babylonjs.com/classes/2.4/PhysicsImpostor

```
                                    );
           }

           frame++;
   };
```

The first argument of the `applyImpulse` method is the vector $\boldsymbol{f}$, namely the force itself, expressed w.r.t. the *global world frame*, whereas the second argument represents the coordinates of the point onto which the force is applied, expressed w.r.t. the *global world frame*.

By using thre `mesh.getAbsolutePosition()` method you can get automatically the barycenter coordinates of the body w.r.t. the *global world frame*.

1.2.2.4. How to implement constraints.

BabylonJS hasn't yet well developed these aspects (June 2016). You have to wait.

1.2.3. Check Point: a Sample Code.

Let's consider the following code.

There are two cylinders, one in an initial rest position and the other with an initial linear velocity having intensity 1.5 along the x axis. Not friction neither resitution are considered.

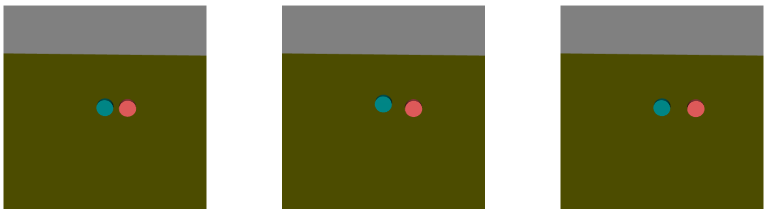
Onto the former cylinder two forces are exerted: at the initial instant `frame==0` an impulsive one is exerted, whereas during the time such that `cylinder1.position.x>10 && cylinder1.position.x < 10.5` an other not impulsive one is exerted.

Both the forces are oriented along the y axis.

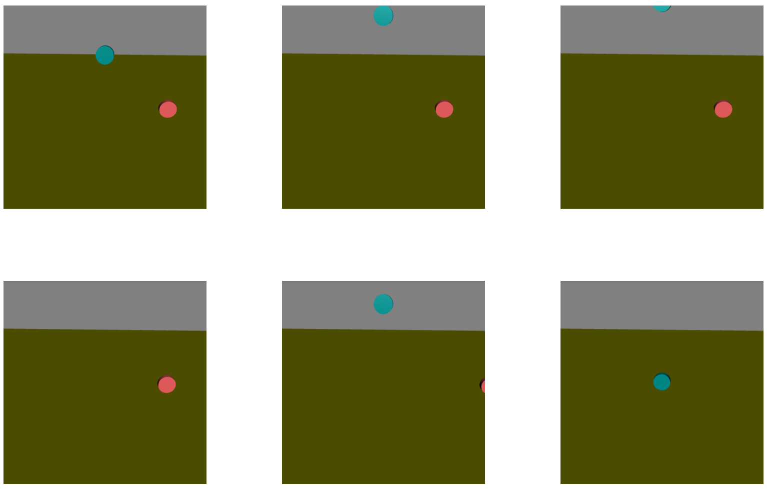
```
// Setting the Structure Layer
var plane = BABYLON.MeshBuilder.CreatePlane("plane", {size: 150}, scene);
plane.translate( new BABYLON.Vector3(0, 1, 0), -1.5, BABYLON.Space.WORLD );
plane.rotate( new BABYLON.Vector3(1, 0, 0), Math.PI/2, BABYLON.Space.WORLD );

var cylinder1 = BABYLON.MeshBuilder.CreateCylinder("cylinder1", {height: 1, diameter:
3, tessellation: 32}, scene);
cylinder1.translate( new BABYLON.Vector3(4,0,0), 1, BABYLON.Space.WORLD);
cylinder1.rotate( new BABYLON.Vector3(1, 0, 0), Math.PI/2, BABYLON.Space.WORLD );

var cylinder2 = BABYLON.MeshBuilder.CreateCylinder("cylinder2", {height: 1, diameter:
3, tessellation: 32}, scene);
cylinder2.translate( new BABYLON.Vector3(0,0,0), 1, BABYLON.Space.WORLD);
cylinder2.rotate( new BABYLON.Vector3(1, 0, 0), Math.PI/2, BABYLON.Space.WORLD );

// Setting the Appearance Layer
scene.clearColor = new BABYLON.Color3(0.5, 0.5, 0.5);
var lights = setLights();

plane.material = new BABYLON.StandardMaterial("materialPlane", scene);
plane.material.diffuseColor = new BABYLON.Color3(0.6 , 0.6,  0);

cylinder1.material = new BABYLON.StandardMaterial("materialCylinder1", scene);
cylinder1.material.diffuseColor = new BABYLON.Color3(1, 0.4, 0.4);

cylinder2.material = new BABYLON.StandardMaterial("materialCylinder2", scene);
cylinder2.material.diffuseColor = new BABYLON.Color3(0, 0.6, 0.6);

// Setting the Motion Layer
scene.enablePhysics(new BABYLON.Vector3(0,-9.81,0), new BABYLON.CannonJSPlugin());

plane.physicsImpostor = new BABYLON.PhysicsImpostor(plane,
BABYLON.PhysicsImpostor.BoxImpostor, {mass: 0, friction: 0, restitution: 0}, scene);

cylinder1.physicsImpostor = new BABYLON.PhysicsImpostor(cylinder1,
BABYLON.PhysicsImpostor.CylinderImpostor, {}, scene);
cylinder1.physicsImpostor.setLinearVelocity(new BABYLON.Vector3(1.5,0,0));
cylinder1.physicsImpostor.setAngularVelocity(new BABYLON.Vector3(0,0,0));
cylinder1.physicsImpostor.setParam("mass", 1);
cylinder1.physicsImpostor.setParam("friction", 0);
```

```
cylinder1.physicsImpostor.setParam("restitution", 0);

cylinder2.physicsImpostor = new BABYLON.PhysicsImpostor(cylinder2,
BABYLON.PhysicsImpostor.CylinderImpostor, {}, scene);
cylinder2.physicsImpostor.setLinearVelocity(new BABYLON.Vector3(0,0,0));
cylinder2.physicsImpostor.setAngularVelocity(new BABYLON.Vector3(0,0,0));
cylinder2.physicsImpostor.setParam("mass", 1);
cylinder2.physicsImpostor.setParam("friction", 0);
cylinder2.physicsImpostor.setParam("restitution", 0);

scene.beforeRender = function () {
        if(frame == 0 || (cylinder1.position.x>10 && cylinder1.position.x < 10.5)){
                cylinder2.physicsImpostor.applyImpulse(
                      new BABYLON.Vector3(0, 4, 0),
                      cylinder2.getAbsolutePosition().add(new BABYLON.Vector3(0, 0, 0))
                );
        }
        frame++;
};

// Setting the Interaction Layer
var camera = setCamera();
// ...

// Setting the Debug Layer (Optional)
// ...
```

In the following figure the evolution of the the code is represented.



The effects of the inirtial additional impulse (`frame==0`) onto the blue cylinder.



The effects of the additional force exerted onto the blue cylinder during the interval `cylinder1.position.x>10 && cylinder1.position.x < 10.5`.

Figure 1.4.

1.2.4. Other tools and techniques.

1.2.4.1. How to simulate clothes physical motions.

In this section you'll find a function generating a moving cloth, that permits to set many parameters.
You can use it like a black–box cloth generator, without examine its details.

```
   function createCloth( size, texturePath, translationVector, rotationVector,
scalingVector ){

           // Setting the Structure Layer
           var subs = size; // subdivisions
           var cloth = BABYLON.MeshBuilder.CreateGround("cloth",
                   { width: size, height: size, subdivisions: subs-1, updatable: true },
scene);
           cloth.translate( translationVector, 1, BABYLON.Space.WORLD);
           cloth.rotate( rotationVector, 1, BABYLON.Space.WORLD );
           cloth.scaling = scalingVector;
           var positions = cloth.getVerticesData(BABYLON.VertexBuffer.PositionKind);

           var particles = [];
           for (var i=0; i<positions.length; i=i+3) {
                   var particle = BABYLON.MeshBuilder.CreateSphere("particle" + i, {
diameter: 0.1 }, scene);
                   particle.visibility = 0;
                   particle.position.copyFrom(BABYLON.Vector3.FromArray(positions, i));
                   particles.push(particle);
           }

           // Setting the Appearance Layer
           cloth.material = new BABYLON.StandardMaterial("material", scene);
           cloth.material.diffuseTexture = new BABYLON.Texture(texturePath, scene);

           // Setting the Motion Layer
           for(i=0; i<particles.length; i++){
                   particles[i].physicsImpostor = new BABYLON.PhysicsImpostor(
                                                   particles[i],
                                                   BABYLON.PhysicsImpostor.ParticleImpostor,
                                                   { mass: i < subs ? 0 : 1 },
                                                   scene);

                   if (i >= subs) {
                           particles[i].physicsImpostor.addJoint(
                                   particles[i-subs].physicsImpostor,
                                   new BABYLON.DistanceJoint({ maxDistance: size/subs }));
                           if (i % subs) {
                                   particles[i].physicsImpostor.addJoint(
                                   particles[i-1].physicsImpostor,
                                   new BABYLON.DistanceJoint({ maxDistance: size/subs }));
                           }
                   }
           }

           cloth.registerBeforeRender(function () {
                   var positions = [];

                   particles.forEach(function (particle) {
                           positions.push(particle.position.x,
                                          particle.position.y,
                                          particle.position.z);
                   });

                   cloth.updateVerticesData(BABYLON.VertexBuffer.PositionKind, positions);
```

```
        });
}
```

The first parameter permits to set the size of the squared drape.

The second parameter permits to set the texture of the drape.

The other three parameters permit respectively to translate w.r.t. the *world global frame*, rotate w.r.t. the *world local frame* and scale w.r.t. the *local frame* the drape.

Therefore you can let the squared drape rectangular by means a scaling transformation.

Here is a sample result.

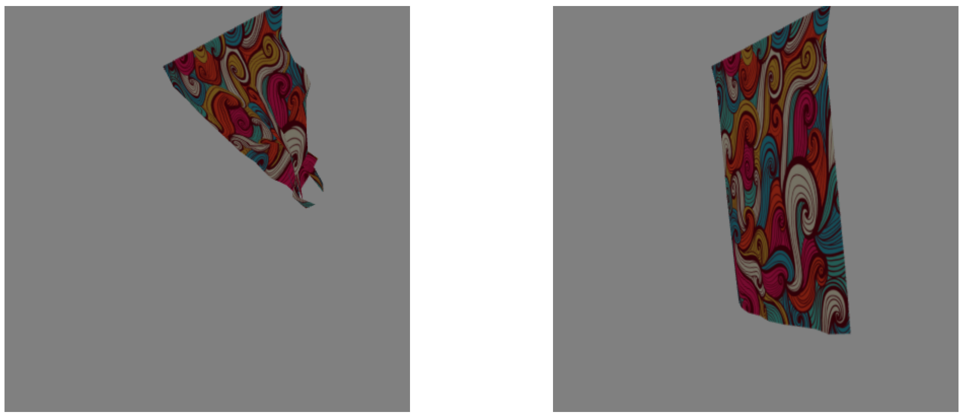


Figure 1.5. `size=10`, `translationVector=new BABYLON.Vector3(-4,8,0)`, `rotationVector=new BABYLON.Vector3(0,Math.PI/8,0)`, `scalingVector=new BABYLON.Vector3(1,1.5,1)`

Pay attention: the cloth is built by means an horizontal plane – a ground –, but you see a vertical plane just because of the gravity. So when you apply a rotation you have to consider that you're rotating an horizontal plane, and not a vertical plane.