Chapter 1

The User Interaction Layer.

1.1. The camera.

1.1.1. Theoretical background.

The camera permits to the user to see the represented scene. By moving the camera you can change the point of view of the scene and explore it.

To move the camera means to change its psition and orientation. The position of the camera is represented by a particular conventional point named *eye point*, that simulates the position of the viewer's eye[1.1].

The orientation of the camera is represented by two directions, the *look–at direction* and the *up direction*: *(i)* the first one is the direction given by the *eye point* and the so called *look–at* point, that is the point of the scene onto which the camera is focused, namely the target point that will be represed at the center of the canvas, *(ii)* the second one represents the direction of the viewer's head vertical axis, that *is always put onto the plane (a) othogonal to the look–at direction and (b) passing through the eye point*, and therefore represents how much the viewer's head is tilted.

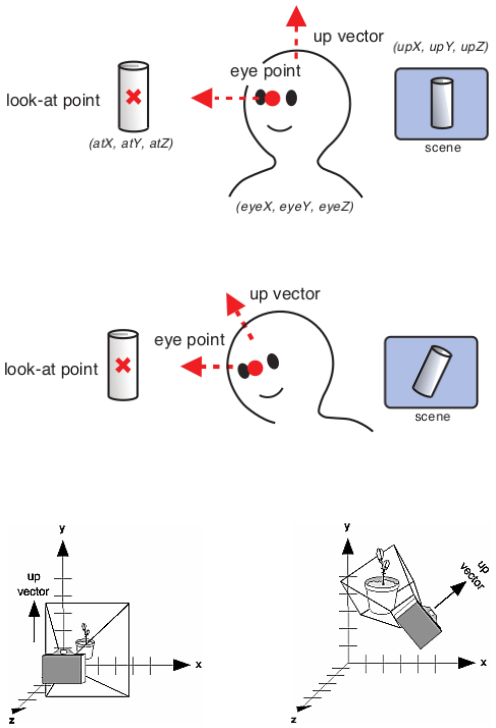

Figure 1.1. Some explicative pictures about the concepts of the *eye poit*, *look–at point*, *look–at direction*, *up direction*.

1.1. yes, the viewer has an only eye!

1.1.2. How to implement a camera in BabylonJS.

BabylonJS provides a lot of types of camera, and the most common ones are listed below.

FreeCamera.

This camera can move itself in each point of the free space.

You can change its position by means the directional arrows and its orientation by using the mouse.

Actually you can instantaneously change the position of the camera just into the plane given by *a)* the *look–at direction* and *b)* the normal direction to both the *up–direction* and the *look–at* one.

You can define this camera by using this instruction:

```
var camera = new BABYLON.FreeCamera(
                                        "FreeCamera",
                                        new BABYLON.Vector3(0, 1, -15),
                                        scene
                                      );
```

The first argument is the name, the second one is the initial *eye position* and the third one is the scene.

You can set the *look–at* position by means this instruction:

```
camera.setTarget(new BABYLON.Vector3(0, 0, 0));
```

The the *up direction* is fixed along the y axis.

ArcRotateCamera.

This camera can move itself in each point just onto the surface of a sphere centered into the *look–at* point.

You can change its position by means the directional arrows, and its orientation will automatically change conveniently.

You can define this camera by using this instruction:

```
var camera = new BABYLON.ArcRotateCamera(
                                           "camera",
                                           1, 0.8, 10,
                                           new BABYLON.Vector3(0, 0, -40),
                                           scene
                                         );
```

The first argument is the name, the following three ones represent the initial *eye position* expressed in spherical coordinates, the fifth one is the *look–at point* and the sixth one is the scene.

You cannot change explicitly the orientation because for this camera *(a)* the *up direction* is fixed *along the direction tangent to the meridian* and *(b)* by definition the *look–at* direction is given by the radius connecting the center with the current position of the camera.

You can find the other (many) types of cameras at the following *URL*:

https://doc.babylonjs.com/tutorials/Cameras

Other than the characteristic instruction of a camera type you have always add these instructions

```
scene.activeCamera = camera;
scene.activeCamera.attachControl(canvas);
```

The former enables the event listeners that permit to the user for changing the pose of the camera by means mouse and/or keyboard (and/or other...), and the latter permits to choose which camera among those you defined you want to use.

1.1.3. How to avoid the `FreeCamera` geometrical intersections.

By definition a `FreeCamera` can be moved freely in the environment.

In order to avoid geometrical intersections between a `FreeCamera` instance and the meshes of the scene you have to add these instructions to the above code:

```
camera.ellipsoid = new BABYLON.Vector3(1, 1, 1);
camera.checkCollisions = true;
meshObstacle1.checkCollisions = true;
meshObstacle2.checkCollisions = true;
```

By setting the `ellipsoid` property you approximate the body of the camera with an ellipsoid having its axes lengths equal to the elements of the assigned `Vector3` instance.

By setting the `checkCollisions` property equals to `true` *for both the camera and all the meshes that you consider like obstacles* you have done: a camera intersection event handler will be automatically raised when a mesh comes in contact with this ellipsoid, preventing our camera from getting too close to it.

Furthermore you can apply gravity to the camera by means this instruction:

```
camera.applyGravity = true;
```

If you are checking the collisions between the camera and a gound, then the camera will fall onto it and not into the oblivion.

Even if you activated the gravity for the camera, it will be attracted by the gravitational force only when you'll move it for the first time.

1.2. The BabylonJS event management systems.

1.2.1. The *event–action–condition* management system.

1.2.1.1. Description.

We deal with an events management system that permits to react with a suitable *action* when an *event* and a *condition* occur together.

In `BabylonJS` each `mesh` instance has the own events management system, that you can initialize with the following instruction:

```
mesh.actionManager = new BABYLON.ActionManager(scene);
```

When you want to handle an event you have to define (obviously) the terne *event–action–condition*, with a code having this general scheme:

```
var trigger = BABYLON.ActionManager.OnXXXTrigger; // the trigger is an event listener
var condition = new BABYLON.XXXCondition(...);

var action = new BABYLON.XXXAction(
                                    trigger,
                                    ...,
                                    condition
                                   );

mesh.actionManager.registerAction(action);
```

In the following table the most common `trigger` possible classes are listed.

`OnXXXTrigger` class.	Description.
`OnLeftPickTrigger`	Raised when the user clicks (or touches) on a mesh with left button.
`OnRightPickTrigger`	Raised when the user clicks (or touches) on a mesh with right button.
`OnLongPressTrigger`	Raised when the user clicks (or touches) up on a mesh for a long period of time, defined by `BABYLON.ActionManager.LongPressDelay`.
`OnPointerOverTrigger`	Raised when the pointer is over a mesh. *Raised just once.*
`OnPointerOutTrigger`	Raised when the pointer is no more over a mesh. *Raised just once.*

Table 1.1.

In the following table the most common `condition` possible classes are listed.

`XXXCondition` constructor.	Arguments.

`ValueCondition (actionManager, target, propertyPath, value, operator)`	`actionManager` The `ActionManager` instance of the considered mesh.
	`target` the considered mesh instance
	`propertyPath` e.g. `"visibility"` `"material.diffuseColor"`
	`value` The value with which you want to compare the current value of the considered property.
	`operator` The comparing operation: equality, inequalities. You can fix it with one of the following constants: `BABYLON.ValueCondition.IsEqual` `BABYLON.ValueCondition.IsDifferent` `BABYLON.ValueCondition.IsGreater` `BABYLON.ValueCondition.IsLesser`

`PredicateCondition(actionManager, predicate)`	`actionManager` The `ActionManager` instance of the considered mesh. `predicate` A function that has to return a boolean value, that establishes if the condition is verified or not.

Table 1.2.

The other arguments of the generic method `XXXAction` depend on the type of the action. In the following table you'll find the most common actions provided by `BabylonJS`. *Remember that the `condition` argument is optional, and you can omit it if you don't need to it.*

XXXAction constructor.	Description.	Special Arguments.
SwitchBooleanAction (trigger, target, propertyPath, condition)	Used to switch the current value of a boolean property.	propertyPath e.g. "visibility" "material.diffuseColor"
SetValueAction (trigger, target, propertyPath, value, condition)	Used to specify a direct value for a property.	propertyPath e.g. "visibility" "material.diffuseColor"
IncrementalValueAction (trigger, target, propertyPath, value, condition)	Add a specified value to a number property.	propertyPath e.g. "visibility" "material.diffuseColor"
ExecuteCodeAction (trigger, func, condition)	Execute your own code written into the funct function.	
InterpolateValueAction(trigger, target, propertyPath, value, duration, condition)	Create an animation to interpolate the current value of a property to a given target. The considered property type has to be numerical, Color3, Vector3 or Quaternion.	propertyPath e.g. "visibility" "material.diffuseColor" duration the duration of the interpolation.

Table 1.3.

For a complete list of all the possibilitues please refer to the BabylobJS documentation.

Furthermore you can concatenate two or more actions *associated to the same event (i.e. to the same trigger)* with the following instruction:

```
box.actionManager.registerAction(action1).then(action2).then(action3);
```

At the first time the considered event occurs `action1` will be executed, at the second time the considered event occurs `action2` will be executed, at the third time the considered event occurs `action3` will be executed, at the fourth time the considered event occurs `action1` will be executed, and so on...

Finally you can combine two or more actions *even not associated to the same event (i.e. to the same trigger)* with the following instructions:

```
var action6 = new BABYLON.CombineAction(triggerLeft,
                                        [action1, action2, action3, action4, action5],
                                        condition);
box.actionManager.registerAction(action6);
```

De facto the `CombineAction` class use the same sintax of a generic `XXXAction` one but it's used for combining two or more actions togher.

Each time the considered event occurs all the actions, from the `action1` to the `action5`, will be executed; *this is possible even if the action are associated to different triggers because this association is overwritten by the first argument of the* `CombineAction` *constructor.*

1.2.1.2. Check point: two Sample Codes.

Let's consider the following code.

It builds a scene with a box and create five different actions, from `action1` to `action5`.

Actions `action1`, `action3` and `action5` are linked to the `OnLeftPickTrigger` trigger, whereas actions `action2` and `action4` are linked to the `OnRightPickTrigger` one.

All the acions are linke to the condition `box.visibility == true`, that could be omitted because it is always verified.

```
// Setting the Structure Layer
var box = BABYLON.MeshBuilder.CreateBox("box", {size: 1}, scene);

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

```
   box.material = new BABYLON.StandardMaterial("materialBox", scene);
   box.material.ambientColor = new BABYLON.Color3(1,0,0);
   box.visibility = 1;

   // Setting the Motion Layer

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

   box.actionManager = new BABYLON.ActionManager(scene);

   var triggerLeft = BABYLON.ActionManager.OnLeftPickTrigger;
   var triggerRight = BABYLON.ActionManager.OnRightPickTrigger;

   var condition = new BABYLON.ValueCondition(
                                        box.actionManager,
                                        box,
                                        "visibility",
                                        1,
                                        BABYLON.ValueCondition.IsEqual
                                        );

   var action1 = new BABYLON.SwitchBooleanAction(
                                    triggerLeft,
                                    box,
                                    "material.wireframe",
                                    condition
                                    );
   var action2 = new BABYLON.SetValueAction(
                                    triggerRight,
                                    box,
                                    "scaling",
                                    new BABYLON.Vector3(5,5,5),
                                    condition
                                    );
   var action3 = new BABYLON.IncrementValueAction(
                                    triggerLeft,
                                    box,
                                    "position.x",
                                    3,
                                    condition
                                    );
   var action4 = new BABYLON.ExecuteCodeAction(
                                    triggerRight,
                                    function(){ box.rotate(new BABYLON.Vector3(0,0,1),
Math.PI/4, BABYLON.Space.WORLD); },
                                    condition
                                    );
   var action5 = new BABYLON.InterpolateValueAction(
                                    triggerLeft,
                                    box,
                                    "material.ambientColor",
                                    new BABYLON.Color3(0,1,1),
                                    3000,
                                    condition
                                    );
```

Now let's consider these two possible further blocks of code:

–

```
box.actionManager.registerAction(action1).then(action3).then(action5);
```

```
        box.actionManager.registerAction(action2).then(action4);
        // picks chronology: left,right,left,left,right

–
        var action6 = new BABYLON.CombineAction(
                                        triggerLeft,
                                        [action1, action2, action3, action4, action5],
                                        condition
                                        );
        box.actionManager.registerAction(action6);
```

The results gained with these possibilities are shown in the following figures.

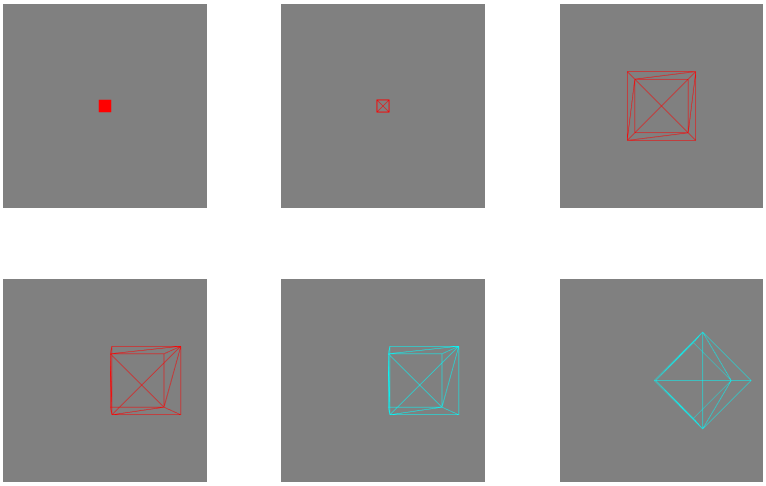

Figure 1.2. The results of the former possibility.

Figure 1.3. Before picking (left) and after picking (right).

Like we said above, even if the actions are initially associated to different triggers – `OnLeftPickTrigger` and `OnRightPickTrigger` – with the `CombineAction` construction their are all associated to the `OnLeftPickTrigger`.

1.2.2. The *observable–observer* management system

An other events management system that provides further events, like `onDispose`*. Outstanding, please refer to http://doc.babylonjs.com/overviews/Observables.*

1.2.3. Some useful properties that can be manipulated with the events handling.

You can play and pause a video by means the properties `material.videoTexture.video.play()` and `material.videoTexture.video.pause()`.