Chapter 1

The Appearance Layer.

1.1. Surface color and Lights.

1.1.1. Theoretical background.

1.1.1.1. The *RBG* color algebra

For the definition of the *RBG* color algebra please refer to other sources.

There exist three different operations for this algebra: *(1)* the color multiplication, that consists into $\mathbf{c}^A \times \mathbf{c}^B = \begin{pmatrix} c_r^A c_r^B \\ c_g^A c_g^B \\ c_b^A c_b^B \end{pmatrix}$, *(2)* the scalar multiplication, that consists into $\lambda \cdot \mathbf{c}^A = \begin{pmatrix} \lambda \cdot c_r^A \\ \lambda \cdot c_g^A \\ \lambda \cdot c_b^A \end{pmatrix}$, and *(3)* the color addition that consists into $\mathbf{c}^A + \mathbf{c}^B = \begin{pmatrix} \max\{c_r^A + c_r^B, 1\} \\ \max\{c_g^A + c_g^B, 1\} \\ \max\{c_b^A + c_b^B, 1\} \end{pmatrix}$.

1.1.1.2. Light sources types.

Very typically in *3D* graphics is used the light rays model[1.1].

There are various type of light sources, that differ one from each other for how they radiate the light rays that they produce. Here is an enumeration of the most common light sources:

– *(uniform) Point light source.*
  It is defined by an unique point in the space, and emits light rays in every direction, i.e. its radiation pattern[1.2] is a sphere.

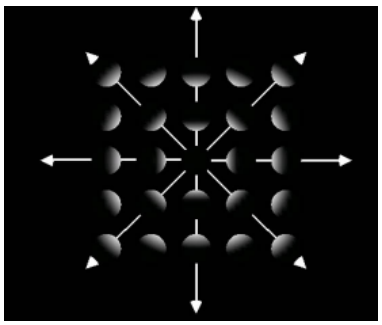


Figure 1.1.
How a point light source emits.

  It is used for example for modelling the light emitted by a light bulb.

– *(uniform) spot ligth source.*
  It is defined by an unique point in the space, and emits light rays in every direction being into a conic volume, i.e. its radiation pattern[1.3] is a cone.

1.1. For more information please refer to:
https://en.wikipedia.org/wiki/Ray_(optics)

1.2. For more information please refer to:
https://en.wikipedia.org/wiki/Radiation_pattern

1.3. For more information please refer to:
https://en.wikipedia.org/wiki/Radiation_pattern

The axis and the opening angle of the cone are named respectively *main direction* and *cutoff angle*.

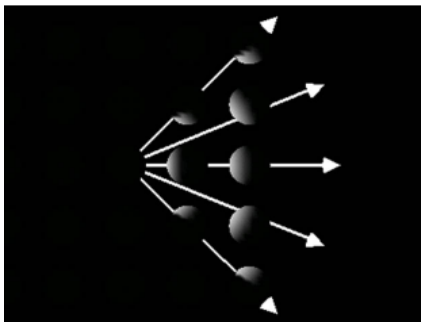


Figure 1.2.
How a spotlight source emits.

It is used for example for modelling the light emitted by a spotlight.

– *Directional light source.*
It is defined like an *uniform point light source at the infinity*. In fact its emitted rays are parallel.

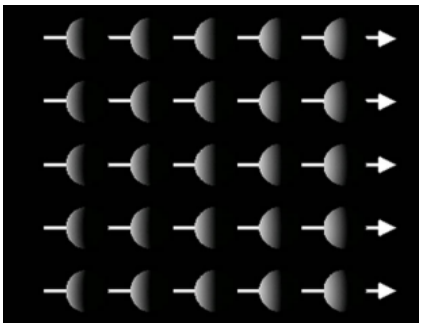


Figure 1.3.
How a directional light source emits.

It is used for example for modelling the light emitted by the sun.

– *Ambient light source.*
It is defined like a set of the ∞^3 point light sources covering the whole free space.

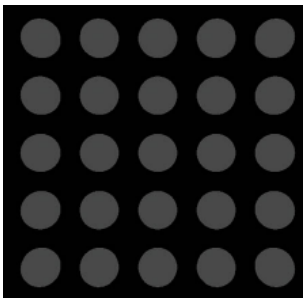


Figure 1.4.
How an ambient light source emits.

It is used for example for modelling the light that, by means diffraction, refraction and reflection mechanisms, reaches the parts of the object that are not directly reached by the rays.

– *Emissive color.*

Furthermore if the material of the object surface is phosphorescent then it emits an own light, *with an unifom intensity in every position and toward each direction*.

Figure 1.5. A green phosphorescent spherical object.

Usually a such object isn't considered a light sources, because its very intense on the surface of the object itself but forfeits very quickly.

1.1.1.3. Ligthing phenomena.

Light reflection and light absorbtion.

When a light ray hits a point of the surface of an object it is reflected along all the free directions.

Actually not all the light energy ray is reflected, in fact a part is absorbed (or, more commonly, trasmitted or refracted) by the material.

The directional distribution of the reflected light energy is shown on the top of the following figure.

It's possible to model this behaviour like the sum of two elementary components: *(1)* a directionally independent uniform component, named the *diffuse reflection* and *(2)* a directionally dependent component symmetrical w.r.t. the specular direction, named the *specular reflection*. These components are shown ont he buttom of the following figure.

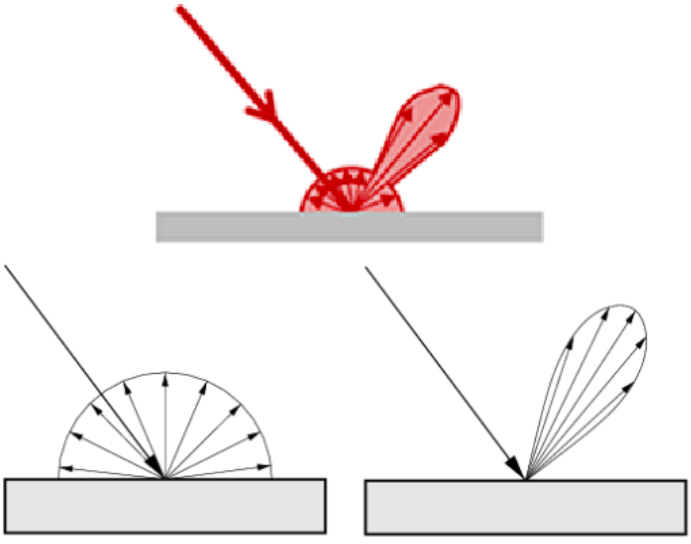


Figure 1.6.

Keep in mind that *this reasoning is valid only for point, spot and directional light sources, whereas for ambient light sources there exists only the diffuse behaviour*.

In fact by definition *each point of a surface can be hitten at most by one light ray for each instance of a such type of sources*, whereas by definition *each point of a surface is hitten by more light rays for each ambient light surce.*

In fact the ambient light is a set of point light sources covering the whole free space, and the resulting reflection is the sum of the contributions of each point light.

It's possible to demosntrate that the omnipresence of the ambient light source causes this result: *the sum of the diffuse plus specular contribution of each point light source produces an only–diffuse reflection behaviour, that is (obviously) different from the singular diffuse reflection contribution.*

As we said above not the whole hitting light energy is reflected by a surface, because a part is absorbed by it. A consequence of this is that *the reflected light rays assume a different color w.r.t. the original one.*

The abosrbing behaviour of a surface material is different w.r.t. the four color channels, namely a generic material absorbs in different ways the red, the green, the blue and the alpha component of the ligth color.

So the reflected ligth color vector $\mathbf{c}^{\text{refl}}$ can be easly computed by knowing *(1)* the hitting light ray color vector $\mathbf{c}^{\text{ligth}}$ and *(2)* the four–dimentional vector $\boldsymbol{\rho}^{\text{abs}}$ composed by the four different absorbing ratios of the material.

In fact in this case we can just do a RGB vector multiplication:

$$\begin{aligned}\mathbf{c}^{\text{refl}} &= \mathbf{c}^{\text{light}} \times \left(\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \boldsymbol{\rho}^{\text{abs}} \right) = \\ &= \mathbf{c}^{\text{light}} \times \boldsymbol{\rho}^{\text{refl}}\end{aligned}$$

Actually a surface material has three different absorbing ratios vectors, and so three different reflection ratios vectors.

In particular each material has *(a)* a ratios vector for the *diffuse reflection* due to an ambient light source, *(b)* a ratios vector for the *diffuse reflection* due to a point, a spot or a directional light source, and *(3)* a ratios vector for the *specular reflection* (obviously) due to a point, a spot or a directional light source.

Let they be respectively $\boldsymbol{\rho}^{\text{refl,diff,amb}}$, $\boldsymbol{\rho}^{\text{refl,diff}}$ and $\boldsymbol{\rho}^{\text{refl,spec}}$.

Shading.

The *shading* is the mechanism that produces the *chiaroscuro* effect on the surface of the illuminated object, by varying levels of darkness.

Keep in mind that *the shading phenomenon there not exists for ambient light sources, but only for point, spot and directional ones.*

1.1.1.4. The resulting surface color formula.

The the following figure is shown a spherical object *(a)* with an own *emissive color* $\mathbf{c}^{\text{emis}}$, *(b)* illuminated by an ambent light color $\mathbf{c}^{\text{light,amb}}$ and – respectively in the left, the central and the right picture – by a point, a spot and a directional light source having color $\mathbf{c}^{\text{light,\{dir,point,spot\}}}$.

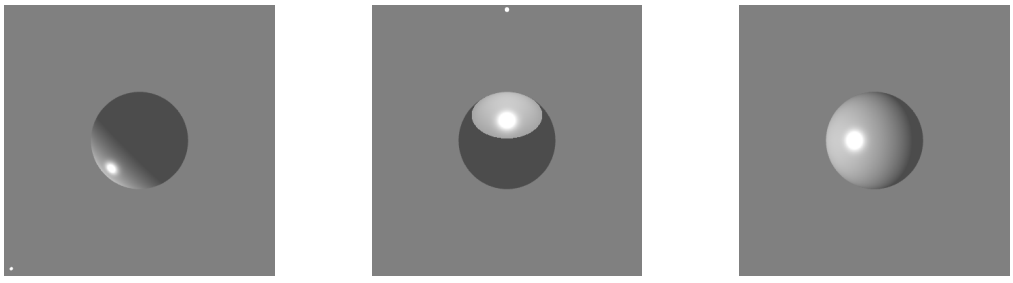


Figure 1.7.

In each picture we can distinguish three different zones, in particular a dark–grey, a middle–grey and a soft–grey zone [1.4].

1.4.

Actually for the second and third zone, but *not also for the first one*, we haven't an homogenous color: this is due to the *shading* phenomenon.

The dark–grey zone represents all the points on the surface that aren't reached by any ray of the point, the spot or the directional light source.

These points are illuminated just by the *ambient light*, that produces only *diffuse reflection*, and by the *emissive color*.

The resulting color for this zone is

$$\mathbf{c}_{\text{zone1}} = \mathbf{c}^{\text{emis}} + (\mathbf{c}^{\text{light,amb}} \times \boldsymbol{\rho}^{\text{refl,diff,amb}})$$

The middle–grey zone represents all the points on the surface such that:

- they are illuminated by the *ambient light* and the *emissive color*
- they are directly reached by a ray of the point, the spot or the directional light source,
- *the light reflected by the surface that arrives to the eye isn't the reflected one around the specular direction*

The resulting color for this zone is

$$\begin{aligned}\mathbf{c}_{\text{zone2}} &= \mathbf{c}^{\text{emis}} + (\mathbf{c}^{\text{light,amb}} \times \boldsymbol{\rho}^{\text{refl,diff,amb}}) + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,diff}}) = \\ &= \mathbf{c}_{\text{zone1}} + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,diff}})\end{aligned}$$

The soft–grey zone represents all the points on the surface such that:

- they are illuminated by the *ambient light* and the *emissive color*
- they are directly reached by a ray of the point, the spot or the directional light source,
- *the light reflected by the surface that arrives to the eye is the reflected one around the specular direction*

The resulting color for this zone is

$$\begin{aligned}\mathbf{c}_{\text{zone3}} &= \mathbf{c}^{\text{emis}} + (\mathbf{c}^{\text{light,amb}} \times \boldsymbol{\rho}^{\text{refl,diff,amb}}) + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,diff}}) + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,spec}}) = \\ &= \mathbf{c}_{\text{zone1}} + \mathbf{c}_{\text{zone2}} + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,spec}})\end{aligned}$$

Even if a generic scene admits the existence of just one *ambient light source* and one *emissive color for object*, there could be more than one point, spot and/or directional light source.

If there were more than one ligth souce of the latter type, you'll get the resulting color by just adding the resulting colors of each light source considered alone, namely you'll use the *superposition principle*[1.5].

But pay attention: as you can see in the previous figure, the zones position changes case by case.

1.1.2. Implementation in `BabylonJS`

1.1.2.1. Implementing the *RBG* color algebra.

`BabylonJS` provides to implement an *RGB* color by the `Color3` class.

This classe performs the three *RGB* operations with the methods `multiply`, `scale` and `add`.

1.1.2.2. Implementing the light sources.

You can create an instance of a point light source by just inserting in your code this instruction:

```
var pointLight = new BABYLON.PointLight(
```

Actually these zones can assume each color you want, like you'll see below.

The choice to use the scale of gray is arbitrary, what is important is that you can distinguish the three zones.

1.5. For more information refer to:

https://en.wikipedia.org/wiki/Superposition_principle

```
                                    "pointLight1",
                                    new BABYLON.Vector3(1, 10, 1),
                                    scene);
```

The second argument is the position of the source w.r.t. the *global world frame*.

You can create an instance of a spot light source by just inserting in your code this instruction:

```
var spotLight = new BABYLON.SpotLight(
                                    "spotLight1",
                                    new BABYLON.Vector3(0, 3, -3),
                                    new BABYLON.Vector3(0, 1, 0)
                                    Math.PI/5,
                                    1,
                                    scene);
```

The second argument is the position of the source w.r.t. the *global world frame*, the third one is the main direction w.r.t. the *global world frame*, the fourth one is the cutoff angle and the fifth one is the falloff exponent[1.6].

You can create an instance of a spot light source by just inserting in your code this instruction:

```
var directionalLight = new BABYLON.DirectionalLight(
                                    "directionalLight1",
                                    new BABYLON.Vector3(0,Math.sqrt(2)/2,Math.sqrt(2)/2),
                                    scene);
```

The second argument is the direction of the light rays emitted by the source w.r.t. the *global world frame*.

The properties `diffuse` and `specular` of each light class represent the value of $\mathbf{c}^{\text{light},\{\text{dir,point,spot}\}}$ when you are considering respectively the *diffuse reflection* and the *specular reflection* [1.7].

A sample code is the following one:

```
light.diffuse = new BABYLON.Color3(1, 1, 1);
light.specular = new BABYLON.Color3(0, 0, 0);
```

In order to change the color $\mathbf{c}^{\text{light,amb}}$ of the ambient source you have just to set the property `ambientColor` of the `scene` instance.

A sample code is the following one:

```
scene.ambientColor = new BABYLON.Color3(1, 1, 1);
```

By default `scene.ambientColor` contains the black color, *that represents de facto the absence of this contribution*.

Because the emissive color depends on the surface material, `BabilonJS` provides the `StandardMaterial` class in order to handle it.

So in order to change $\mathbf{c}^{\text{emis}}$ you have by first *(1)* to create the material and *(2)* to assign it to the object.

1.6. Since we have considered only the uniform spot light source, this parameter will be always set to 1. For more informations please refer to other sources.

1.7. This is a very strange aspect of `BabylonJS`.

In fact by remembering the final color of the so called soft–grey zone,

$$\mathbf{c}_{\text{zone3}} = \mathbf{c}^{\text{emis}} + (\mathbf{c}^{\text{light,amb}} \times \boldsymbol{\rho}^{\text{refl,diff,amb}}) + (\mathbf{c}^{\text{light,}\{\text{dir,point,spot}\}} \times \boldsymbol{\rho}^{\text{refl,diff}}) + (\mathbf{c}^{\text{light,}\{\text{dir,point,spot}\}} \times \boldsymbol{\rho}^{\text{refl,spec}})$$

you can see just one value for the point, spot and directional light color.

This is also natural way to model light: the position and the topology of the soft–grey zone depend by the mutual positions and orientations of the light source, the object surface and the viewer's eye, and so the light cannot change its color in dipendence of the viewer's and the object poses!

A sample code is the following one:

```
sphere.material = new BABYLON.StandardMaterial("materialSphere", scene);
```

Then you can handle the value of $\mathbf{c}^{\text{emis}}$ by setting the properties `emissiveColor` and `alpha` of the material.

A sample code is the following one:

```
sphere.material.emissiveColor = new BABYLON.Color3(1, 1, 0);
```

By default `emissiveColor` contains the black color, *that represents de facto the absence of this contribution*.

1.1.2.3. Implementing lighting phenomena.

Implementing light reflection and light absorbtion.

Because in the real world absorbing behaviour of an object depends on its surface material, also for manipulate the values of $\boldsymbol{\rho}^{\text{refl,diff}}$ and $\boldsymbol{\rho}^{\text{refl,spec}}$ you need to use an instance of the `StandardMaterial` class.

In particular the values of $\boldsymbol{\rho}^{\text{refl,diff,amb}}$ $\boldsymbol{\rho}^{\text{refl,diff}}$ and $\boldsymbol{\rho}^{\text{refl,spec}}$ can be put in the `ambientColor`, the `diffuseColor` and the `specularColor` properties of the material.

If you have yet instantiated the material for setting the emissive color, then you have to use that istantance, and not to replace it with a new one (otherwise you'll lose the emissive color, obviously).

A sample code is the following one:

```
// If not yet done for emissive color add also this:
// sphere.material = new BABYLON.StandardMaterial("materialSphere", scene);
sphere.material.ambientColor = new BABYLON.Color3(0.5, 0.5, 0.5);
sphere.material.diffuseColor = new BABYLON.Color3(0.7, 0.7, 0.6);
sphere.material.specularColor = new BABYLON.Color3(0.9, 0.9, 0.8);
```

As you can see, `ambientColor`, `diffuseColor` and `specularColor` represent the values of $\boldsymbol{\rho}^{\text{refl,diff,amb}}$, $\boldsymbol{\rho}^{\text{refl,diff}}$ and $\boldsymbol{\rho}^{\text{refl,spec}}$.

By default these properties contain the white color, *that represents de facto the absence of the absorbing behaviour*.

Implementinng the shading.

The shading effect is made automatically by `BabylonJS`: *you have just to do nothing!*

1.1.2.4. Implementing the resulting surface color formula.

Let's remember the final colors formulas.

$$\mathbf{c}_{\text{zone1}} = \mathbf{c}^{\text{emis}} + (\mathbf{c}^{\text{light,amb}} \times \boldsymbol{\rho}^{\text{refl,diff,amb}})$$

$$\mathbf{c}_{\text{zone2}} = \mathbf{c}_{\text{zone1}} + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,diff}})$$

$$\mathbf{c}_{\text{zone3}} = \mathbf{c}_{\text{zone1}} + \mathbf{c}_{\text{zone2}} + (\mathbf{c}^{\text{light,\{dir,point,spot\}}} \times \boldsymbol{\rho}^{\text{refl,spec}})$$

In the following table we resume how to set all the needed variables.

Theoretical variable.		BabylonJS tool.
$\mathbf{c}^{\text{light,amb}}$		`scene.ambientColor`
$\mathbf{c}^{\text{light,\{dir,point,spot\}}}$	(diffuse reflection considered)	`light.diffuse`
$\mathbf{c}^{\text{light,\{dir,point,spot\}}}$	(specular reflection considered)	`light.specular`
$\mathbf{c}^{\text{emis}}$		`mesh.material.emissiveColor`
$\boldsymbol{\rho}^{\text{refl,diff,amb}}$		`mesh.material.ambientColor`
$\boldsymbol{\rho}^{\text{refl,diff}}$		`mesh.material.diffuseColor`
$\boldsymbol{\rho}^{\text{refl,spec}}$		`mesh.material.specularColor`

Table 1.1.

For computanional aspects by default a `StandardMaterial` instace considers up to four different light sources, and ignores eventual other ones.

You can modify this limit by using the `material.maxsimultaneousLights` property (*do not exaggerate!*).

A point, spot and directional light instances can be deactivated by calling the `setEnabled` method, whereas the emissive color and the ambient light source can be deactivated with their default value (the black color).

1.2. How to use texture images.

1.2.1. How to attach a texture image to a mesh surface.

Obviously `BabylonJS` permits to attach textures to the surfaces of the objects. In particular it provides the following properties in order to do this:

```
mesh.material.emissiveTexture=new BABYLON.Texture("./path/emissiveTexture.png", scene);
mesh.material.ambientTexture=new BABYLON.Texture("./path/ambientTexture.png", scene);
mesh.material.diffuseTexture=new BABYLON.Texture("./path/diffuseTexture.png", scene);
mesh.material.specularTexture=new BABYLON.Texture("./path/specularTexture.png", scene);
```

They are equivalent to the color properties seen above and influence in the same way the resulting color.

However if you are considering the texture then the final color formulas become more complex, because the values of $\mathbf{c}^{\text{emis}}$, $\boldsymbol{\rho}^{\text{refl,diff,amb}}$, $\boldsymbol{\rho}^{\text{refl,diff}}$ and $\boldsymbol{\rho}^{\text{refl,spec}}$ become the *RGB* vector product respectively between `emissiveColor` and `emissiveTexture`, `ambientColor` and `ambientTexture`, `diffuseColor` and `diffuseTexture`, `specularColor` and `specularTexture`.

So *if you want to represent purely a texture you have to put the value of the corresponding color property equals to* `new BABYLON.Vector3(1,1,1)`.

In the following we'll explain how to use textures optimally only for planes and boxes, i.e. the most common cases.

1.2.2. How to use a texture image on a plane.

We want to put the following `640 x 360 px` texture image, saved at the relative location `./resources/PalletTown.png` onto the surface of a plane.

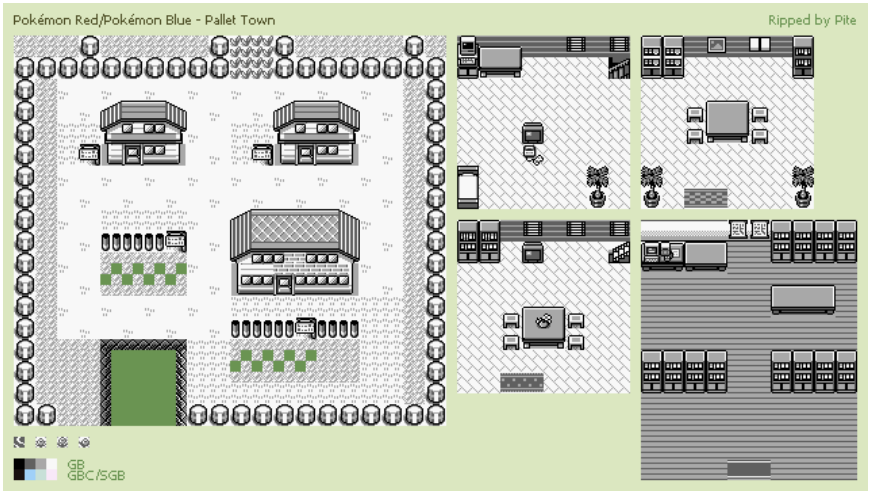


Figure 1.8.

In order to do this we have to build a planar mesh *having the same aspect ratio*[1.8] *of the above texture image* and set conveniently the $\boldsymbol{\rho}^{\mathrm{refl,diff,amb}}$ value, as follows

```
var plane = BABYLON.MeshBuilder.CreatePlane("plane", {width: 6.4, height: 3.6}, scene);
plane.material = new BABYLON.StandardMaterial("material", scene);
plane.material.ambientColor = new BABYLON.Color3(1 , 1,  1);
plane.material.ambientTexture=new BABYLON.Texture("./resources/PalletTown.png", scene);
```

You have to choose the same aspect ratio because *the texture will always cover the entire surface, even if it has to re–scale itself*. The following figure clarifies this aspect.

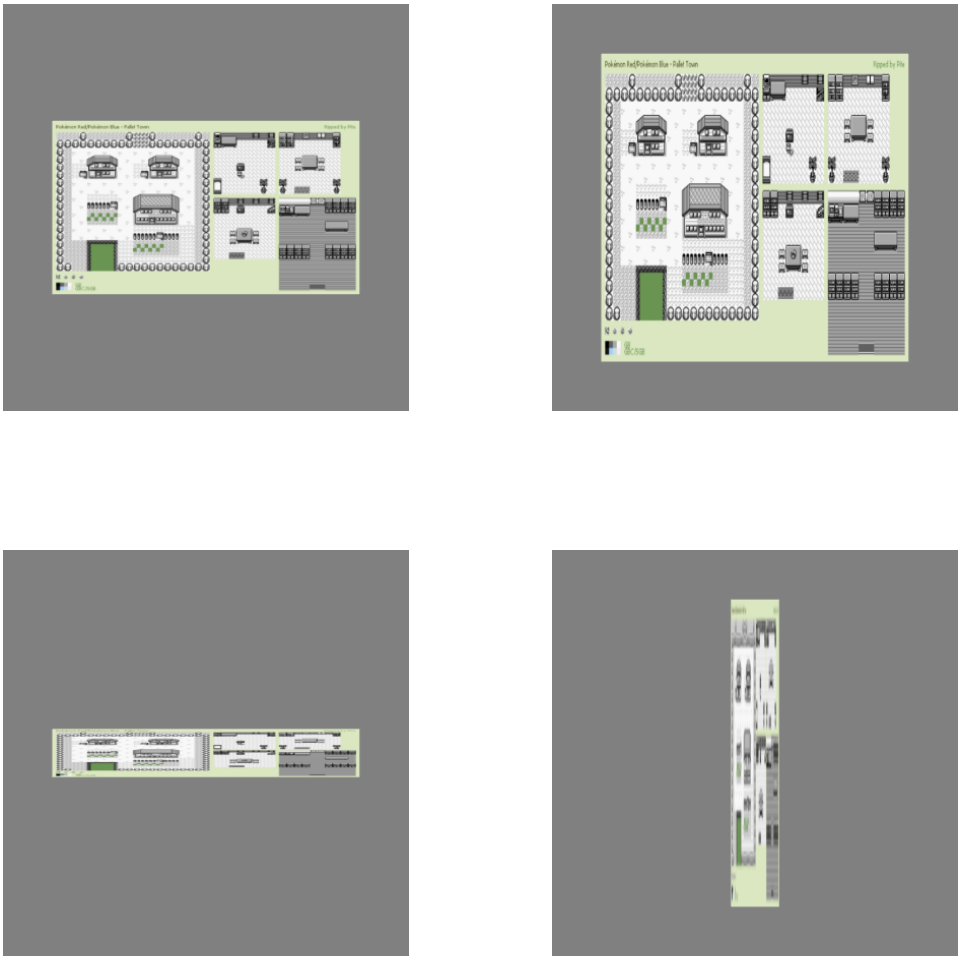


Figure 1.9.
Top left (the result of the above code):
`var plane = BABYLON.MeshBuilder.CreatePlane("plane", {width: 6.4, height: 3.6}, scene);`
Top right:
`var plane = BABYLON.MeshBuilder.CreatePlane("plane", {width: 6.4, height: 6.4}, scene);`
Bottom left:
`var plane = BABYLON.MeshBuilder.CreatePlane("plane", {width: 6.4, height: 1}, scene);`

The problem of the aspect ratio is most critical one about texturing; it is also explained by the

1.8. For more informations please refer to
https://en.wikipedia.org/wiki/Aspect_ratio_(image)

`babylonJS` developers:

> *"In classic texturing, we use 2D images, often pictures that have been shaped specifically to match an object."* (*https://doc.babylonjs.com*, 31 May 2016)

You can put onto the surface also a translated version of the texture image by means the `uOffset` and `vOffset` properties. The following figure clarifies this aspect.

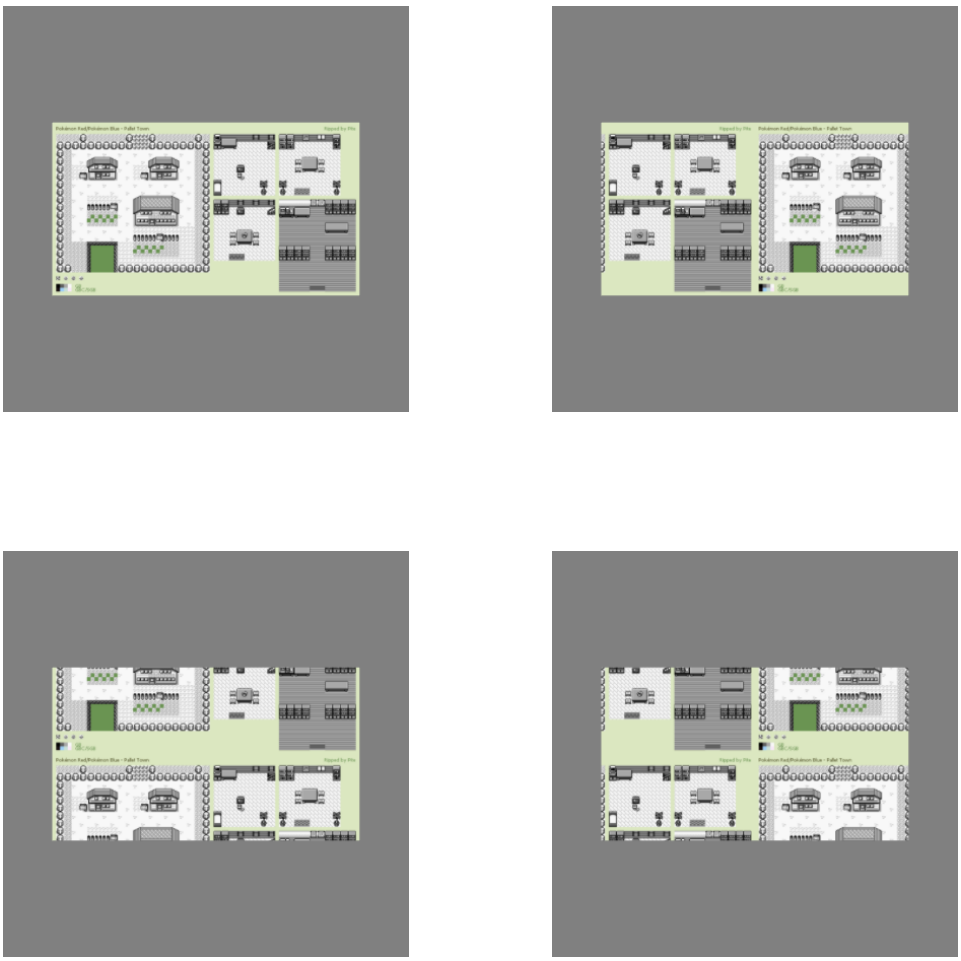


Figure 1.10.
Top left (default behaviour):
`plane.material.ambientTexture.uOffset = 0; plane.material.ambientTexture.vOffset = 0;`
Top right:
`plane.material.ambientTexture.uOffset = 0.5; plane.material.ambientTexture.vOffset = 0;`
Bottom left:
`plane.material.ambientTexture.uOffset = 0 plane.material.ambientTexture.vOffset = 0.5;`
Bottom right ():
`plane.material.ambientTexture.uOffset = 0.5; plane.material.ambientTexture.vOffset = 0.5;`

You can use the `uScale` and `vScale` properties to decide how much copies of the texture image have to be represented on the surface, indipendently along the horizontal and the vertical directions. The following figure clarifies this aspect.

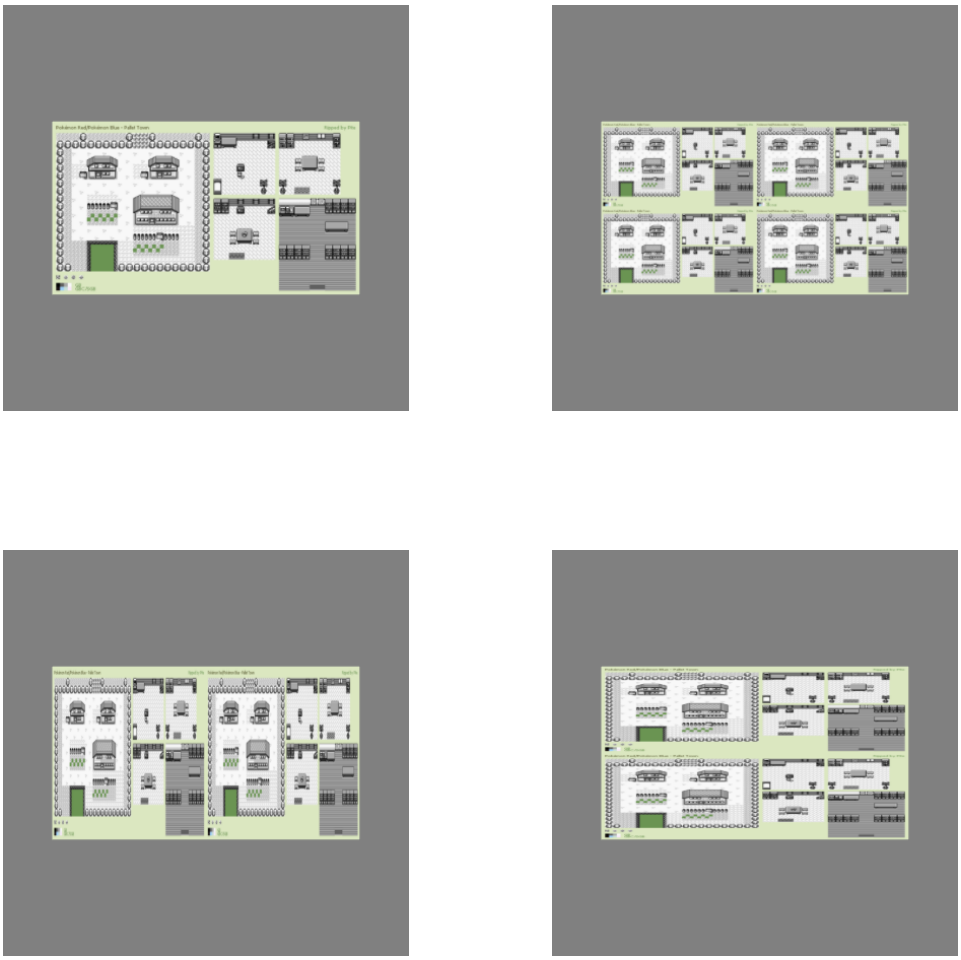


Figure 1.11.
Top left (default behaviour):
`plane.material.ambientTexture.uScale = 1; plane.material.ambientTexture.vScale = 1;`
Top right:
`plane.material.ambientTexture.uScale = 2; plane.material.ambientTexture.vScale = 2;`
Bottom left:
`plane.material.ambientTexture.uScale = 2 plane.material.ambientTexture.vScale = 1;`
Bottom right ():
`plane.material.ambientTexture.uScale = 1; plane.material.ambientTexture.vScale = 2;`

You can put onto the surface also a rotated version of the texture image by means the `wAng` property. The following figure clarifies this aspect.

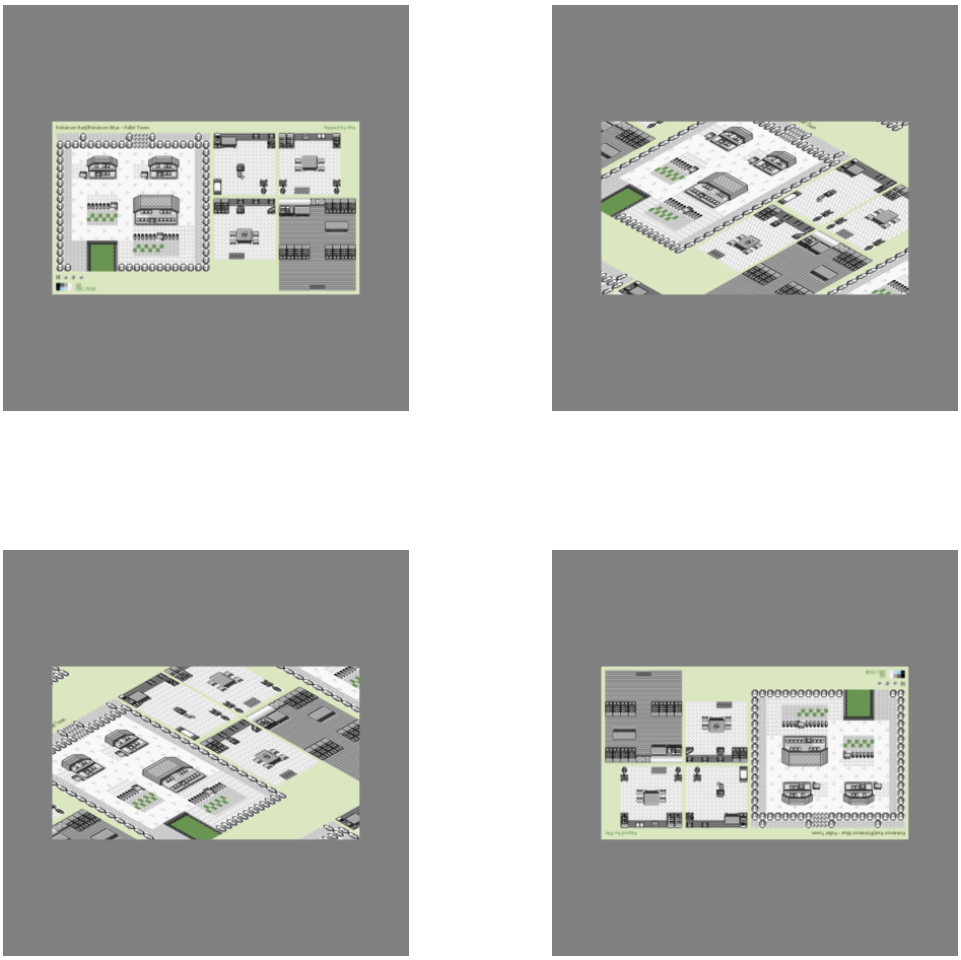


Figure 1.12.
Top left (default behaviour):
`plane.material.ambientTexture.wAng = 0;`
Top right:
`plane.material.ambientTexture.wAng = Math.PI/4;`
Bottom left:
`plane.material.ambientTexture.wAng = -Math.PI/4;`
Bottom right:
`plane.material.ambientTexture.wAng = Math.PI;`

1.2.3. How to use a texture image on a box.

In order to make texturing on a box the most intuitive approach may be to use a separate texture image for each face.

But this approach is not convenient in terms of computation compelxity and so pratically *(A)* only a texture image is used and *(B)* only a suitable rectangular part of it is mapped onto each face.

In order to describe a generic rectangular part of a texture image we need to indicate to `BabylonJS` four numbers: the *normalized* coordinates of the bottom–left and of the top–right vertices of the desired rectangle w.r.t. the bottom–left vertex of the original texture image.

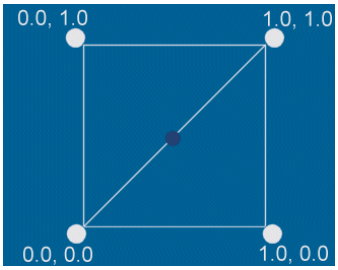


Figure 1.13.
The coordinate system used for describing the coordinates of the bottom–left and of the top–right vertices of the desired rectanglar part of the origin texture.

In particular for mapping a texture on a box in the above sense we need to:

– *indicate to* `BabylonJS` $4 \cdot 6 = 24$ *numbers, organized in an array of* `Babylon.Vector4` *instances having cardinality equals to* 6*: the first element has to refer to the rectangular part related to the front face, the second to the back one, the third to the left one, the fouth to the right one, the fifth to the top one and the sixth to the bottom one;*

– *put this array in the* `faceUV` *argument of the* `BABYLON.MeshBuilder.CreateBox` *constructor.*

For example let consider the following texture image.

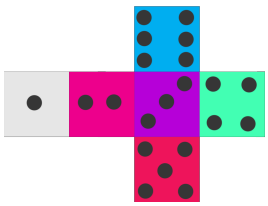


Figure 1.14.

In order to build a die, we have just to map onto each face the suitable squared part of this image containing the desider number.

A sample code is the following one:

```
var coordsUV = [
                new BABYLON.Vector4(0.00,0.25,0.25,0.50), // 1 ---> front
                new BABYLON.Vector4(0.25,0.25,0.50,0.50), // 2 ---> back
                new BABYLON.Vector4(0.50,0.25,0.75,0.50), // 3 ---> left
                new BABYLON.Vector4(0.75,0.25,1.00,0.50), // 4 ---> right
```

```
                    new BABYLON.Vector4(0.50,0.00,0.75,0.25), // 5 ---> top
                    new BABYLON.Vector4(0.50,0.50,0.75,0.75)  // 6 ---> button
                   ];
var box = BABYLON.MeshBuilder.CreateBox("box", {size: 3, faceUV: coordsUV}, scene);
box.material = new BABYLON.StandardMaterial("material", scene);
box.material.ambientColor = new BABYLON.Color3(1 , 1,  1);
box.material.ambientTexture = new BABYLON.Texture("./resources/die.png", scene);
```

The result is shown to the following figure.

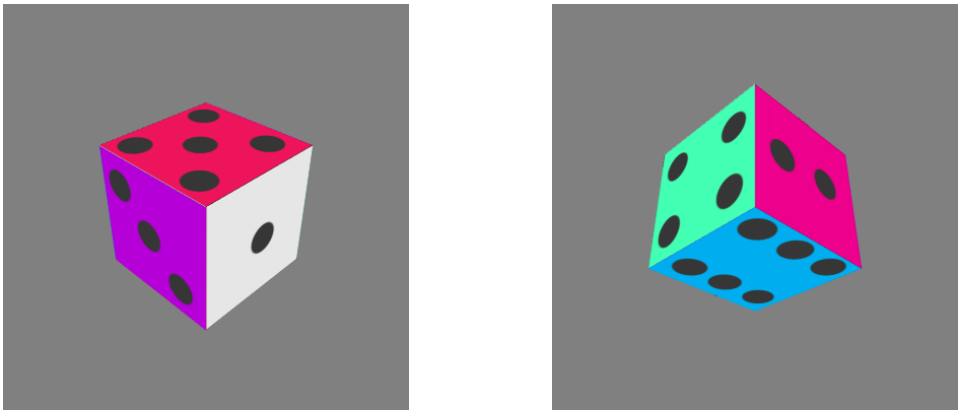
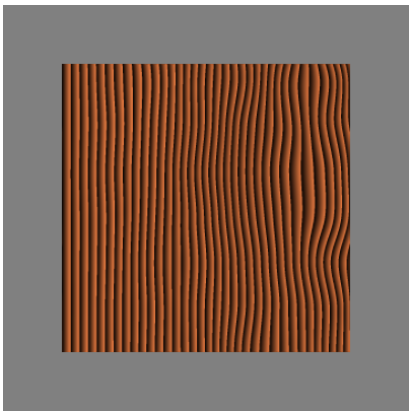


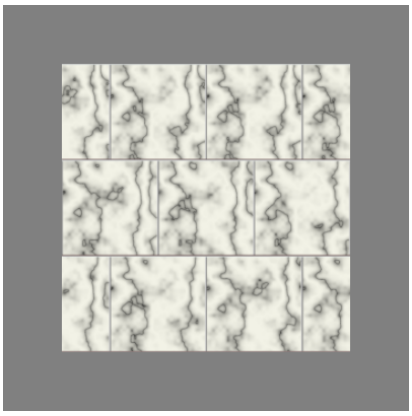
Figure 1.15. Front view (left) and back view (right).

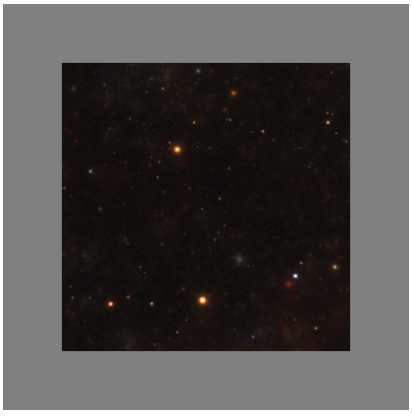
1.2.4. Procedural texture images.

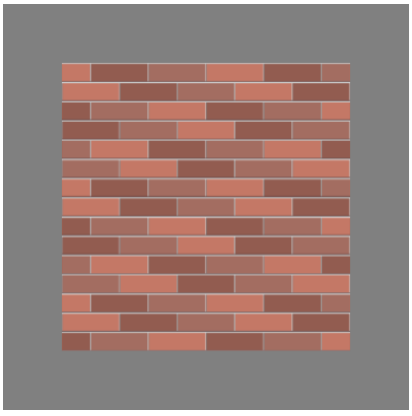
Procedural texture images are texture images written ad hoc for your scene using a fragment shader, and not loaded by the server filesystem.

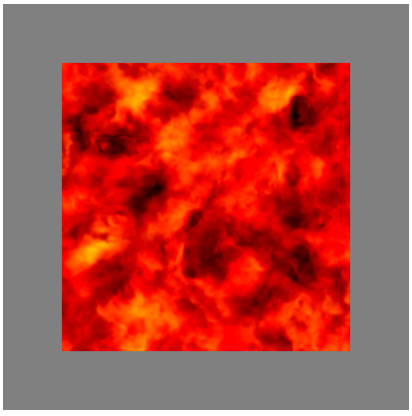
Obviously such images are general purpose, and so have a very simple theme. Currently `BabylonJS` provides the following themes, shown in the following picture: wood, marble, starfield, brick, fire, grass, road and cloud.

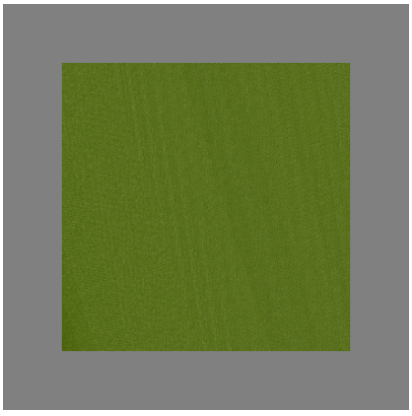


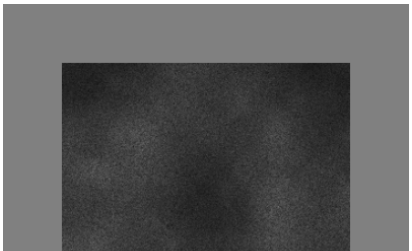


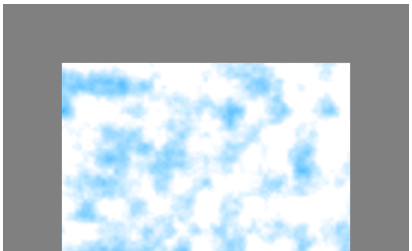












The main advantages of the procedural texture images are: *(1)* the dimention of the image is choosen ad hoc for the surface to apply it, and so there aren't problems about aspect ratio and size; *(2)* the code generating the texture is *GLSL* and not `Javascript`, and so it is executed by the *GPU* and not the *CPU*, gaining an huge performance impact in a positive way.

`BabylonJS` permits to build a generic procedural texture image by using this format:

```
mesh.material.ambientTexture=new BABYLON.XXXProceduralTexture("texture", 1024, scene);
```

For example in order to create a `WoodProceduralTexture` instance and map it onto a plane you have just to use this code:

```
var plane = BABYLON.MeshBuilder.CreatePlane("plane", {size: 5}, scene);
plane.material = new BABYLON.StandardMaterial("material", scene);
plane.material.ambientColor = new BABYLON.Color3(1 , 1,  1);
plane.material.ambientTexture=new BABYLON.WoodProceduralTexture("texture", 1024, scene)
```

and you get the result shown from the top–left image of the previous image.

Furthermore, *the composition of procedural textures process is parametric*, namely you can optionally change the values of special default properties.

Here is an example of setting a parameter for building a wood procedural texture image:

```
plane.material.ambientTexture.woodColor = new BABYLON.Color3(0.50, 0.25, 0);
```

the new result is the following one.

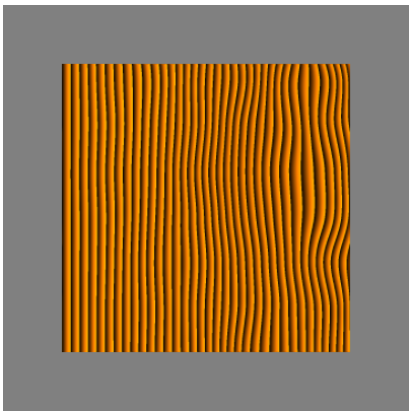


Figure 1.17.

That's it!

Currently (June 2016) there's not a complete documentation on the `Web` about all the classes related to these textures and all their parameters; neverthless each type of procedural texture image corresponds to a `Javascript` file into the folder `./lib/ProceduralTexturesLibrary`, that we have included by means a `php` in the `basic.php` file. So you can extract all informations you need from their code.

1.3. The environment.

1.3.1. How to set the background color.

So far we have just set the background color equals to the middle grey {0.5, 0.5, 0.5} by means the instruction:

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

If you want to change this color you have just to change this *RGB* value with another one related to the desired color.

For example in order to set the yellow color $\{1, 1, 0\}$ you have just to use this code:

```
scene.clearColor = new BABYLON.Color3(1, 1, 0);
```

and you get the following result.

Figure 1.18.

1.3.2. How to use texture images for the background.

This method consists into insert the whole scene into a box such that *(1)* its inner faces are covered by texture images representing a suitable environment, *(2)* is not influenced by local light sources and *(3)* is set at infinity, i.e. doesn't translate when when you change the camera position.

In order to create a sky–box you have just insert the following code.

```
// Creating the sky-box
var skybox = BABYLON.Mesh.CreateBox("skyBox", 500, scene);
skybox.material = new BABYLON.StandardMaterial("skybox_material", scene);

skybox.material.backFaceCulling = false;
skybox.material.disableLighting = true;
skybox.infiniteDistance = true;

skybox.material.reflectionTexture = new BABYLON.CubeTexture("./resources/skybox_folder/
skybox", scene);
skybox.material.reflectionTexture.coordinatesMode = BABYLON.Texture.SKYBOX_MODE;
```

As you can see it's possible to create a skybox in three steps: *(1)* create a big box and its material instances, *(2)* set the three properties of the sky–box (enumered above) and *(3)* load the texture images (and an other thing, don't care about it).

In particular the loading phase depends on the argument of the `CubeTexture` method, that must have the following scheme:

`./relative/path/prefix`

In fact the system will search for 6 texture images *(1)* stored in the folder `./relative/path/` and *(2)* named with the prefix `prefix` and the suffixes `_px.format`, `_nx.format`, `_py.format`, `_ny.format`, `_pz.format` and `_nz.format` respectively for the right, the left, the top, the bottom, the front and the back face[1.9].

For example by using the following texture images (bottom image)[1.10] you get the following result (top image).

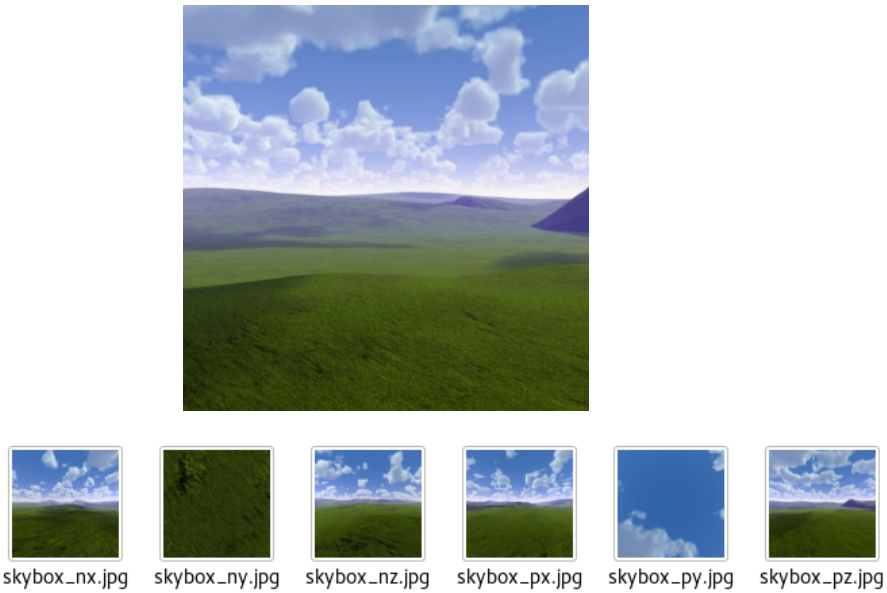


Figure 1.19.

Finally in order to put always the scene forward the skybox remember you have to set the `renderingGroupId` property equal to 1 (or more) for each element of the scene (meshes, particle systems and sprite managers).

A sample code follows:

```
// Creating the scene
var sphere = BABYLON.MeshBuilder.CreateSphere("sphere", {}, scene);
sphere.renderingGroupId = 1;

// Creating the sky-box
var skybox = BABYLON.Mesh.CreateBox("skyBox", 500, scene);
...
```

and here is the related result.

1.9. "n" means "negative" and "p" means "positive"

1.10. You can find a lot of possible sky–box testures at this link

http://3delyvisions.co/skf1.htm

Figure 1.20.

1.4. How to add shadows.

In this section we'll explain how `BabylonJS` implements the *shadow mapping*, a technique that permits to create shadows. Here is an explicatory image about the working principle of the *shadow mapping*[1.11].

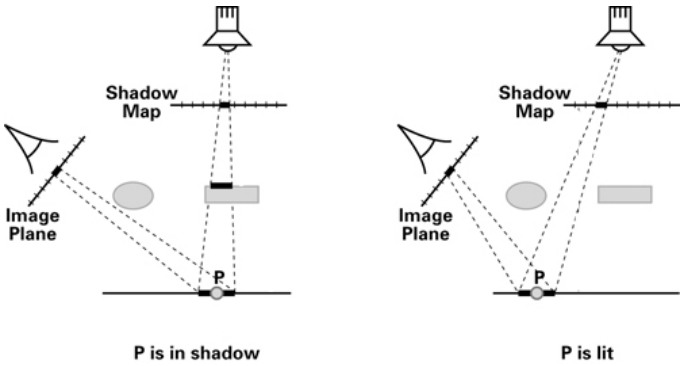


Figure 1.21.

In order to realize the shadows `BabylonJS` need for three informations: *(1)* the light sources instances for which you want enable this effect, *(2)* the meshes instances for which you want produce the shadows and *(3)* the meshes onto which you want project the (possible) shadows.

The first information is passed by menas the following instruction:

```
var shadowGenerator = new BABYLON.ShadowGenerator(1024, light);
```

The second information is passed by means the following instruction:

```
shadowGenerator.getShadowMap().renderList = [meshShadowProducer1,meshShadowProducer2];
```

The third information is passed by means the following instructions:

```
meshOntoWhichProjectShadows1.receiveShadows = true;
meshOntoWhichProjectShadows1.receiveShadows = true;
```

Keep in mind that you need of a `ShadowGenerator` instance for each source light for which you want enable the shadowing effect.

1.11. For more informations please refer to other sources.

For example let's consider a scene in which there are *(1)* two directional light sources, one along the $\left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0\right)$ direction and the $\left(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0\right)$, *(2)* two spherical mesh and one planar one. In the following figure this scene is shown both with both without the shadowing effect.

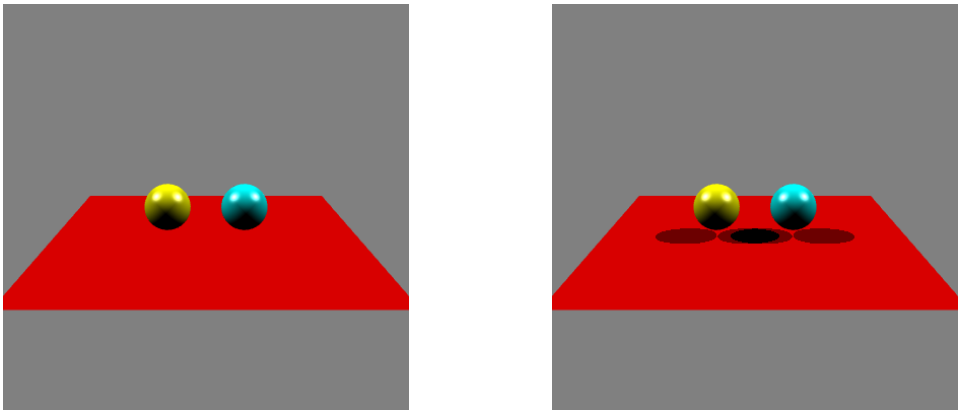


Figure 1.22.

The *Appearance Layer* code for this result is the following one:

```
// Setting the Appearance Layer

// Setting lights
var lights = setLights();
// lights[0] <--- a directional light source along (+Math.sqrt(2)/2,-Math.sqrt(2)/2,0)
// lights[1] <--- a directional light source along (-Math.sqrt(2)/2,-Math.sqrt(2)/2,0)

// Setting materials
sphere1.material = new BABYLON.StandardMaterial("materialSphere1", scene);
sphere1.material.diffuseColor = new BABYLON.Color3(0.8,0.8,0);

sphere2.material = new BABYLON.StandardMaterial("materialSphere2", scene);
sphere2.material.diffuseColor = new BABYLON.Color3(0,0.8,0.8);

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

// Setting shadowing effect
var shadowGenerator1 = new BABYLON.ShadowGenerator(1024, lights[0]);
var shadowGenerator2 = new BABYLON.ShadowGenerator(1024, lights[1]);

shadowGenerator1.getShadowMap().renderList = [sphere1,sphere2];
shadowGenerator2.getShadowMap().renderList = [sphere1,sphere2];

sphere1.receiveShadows = true;
sphere2.receiveShadows = true;
plane.receiveShadows = true;
```

1.5. FURTHER TOOLS AND TECHNIQUES.

1.5.1. The wireframe modality.

By setting the boolean `wireframe` property of an object material

```
sphere.material.wireframe = true;
```

you can obtain the effect shown in the following figure.

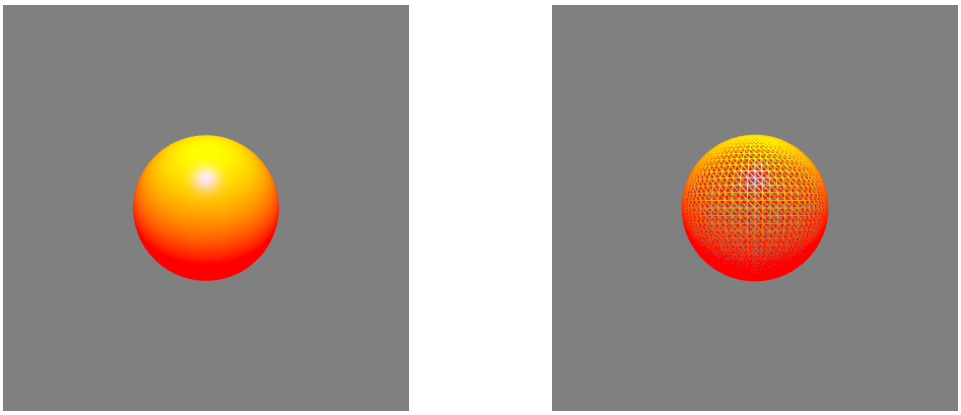


Figure 1.23.

1.5.2. How to set the mesh trasparency.

By setting the numerical `visibility` property of a mesh

```
mesh.visibility = 0.7;
```

you can obtain the effects shown in the following figure.

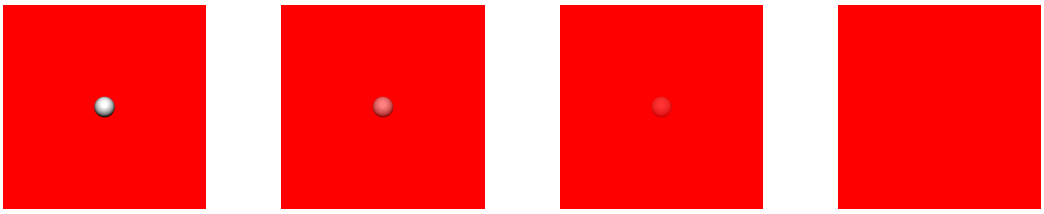


Figure 1.24.
From the left side: `visibility` equals to `1.0`, `0.5`,`0.2` and `0.0`.

1.5.3. How to add audio and video.

`BabylonJS` provides a very simple way to add audio and/or video files, in fact you have just to use the following instructions[1.12]:

```
var screen = BABYLON.MeshBuilder.CreatePlane("screen", {size: 7}, scene);
screen.material = new BABYLON.StandardMaterial("materialScreen", scene);
screen.material.diffuseTexture = new BABYLON.VideoTexture("video",["./path/to/
videoAndOrAudioFile.format"],scene,true);
screen.material.emissiveColor = new BABYLON.Color3(1, 1, 1);
```

1.5.4. How to add sprites.

Sprites are a rudimentary type of video, composed by a set of fotograms (i.e. images) that are shown sequentially onto a platform in order to represents an animated scene.

1.12. Tested only for `.mp4` and `.mp3` files.

Figure 1.25.

`BabylonJS` permits to implement a sprite into three steps:

I. Creating a `SpriteManager` instance.
   A sample code is the following one:

```
var spriteManager = new BABYLON.SpriteManager(
                                                "spriteManager",
                                                "imageURL.format",
                                                1,
                                                64,
                                                scene
                                                );
```

   The first argument is just the name.
   The second one is the relative path of the so called *sprites atlas*: instead of loading each photogram separately, its computationally better to load an only image containing all the photograms, that is named *sprites atlas*. *This is particularly true in this* `Web`*–based context*.
   Into the atlas all the photograms have to be organized in a grid layout, as it is shown in the following figure.

Figure 1.26.

   The fourth argument represents the size of the cell of the above grid, that permits to extract from the atlas all the spirtes correctly. If this quantity is too small or too large you'll get the effects shown in the following figure.

Figure 1.27. Two examples about the first photogram of the above atlas when the cell size is choosen too small (left) or too large (right).

   The third argument will be always set to 1, and the fifth one is the scene instance.

II. Creating a `Sprite` instance.
   In this step you have to write the following instruction:

```
var sprite = new BABYLON.Sprite("sprite", spriteManager);
```

   De facto with this instruction you create the platform onto which the photograms will be put.

You can modify a lot of properties of this platform, like *(a)* `position`, that represents its position w.r.t the *world global frame*, *(b)* `angle` that, *because the platform is always put in front of the viewer, whatever is its pose*, represents the only orientation degree of freedom of the platform itself, *(c)* `width` and `height`, that represents obviously its dimentions.

Here is an example.

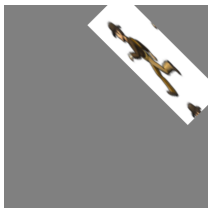


Figure 1.28. A `Sprite` instance having the following properties values: `position = new BABYLON.Vector3(1,1,0)`, `angle = Math.PI/4`, `width = 1` and `height = 3`.

It's woth to note that ***`position`*** *permits to translate,* ***`angle`*** *permits to rotate and* ***`width`*** *and* ***`height`*** *permit to scale the platform.*

III. Enabling the sprite.
At this point you have two possibilities:

– To start the pseudo–video.
In this case you have just to execute the following instructio:

```
sprite.playAnimation(0, 44, true, 100);
```

The first and the second arguments are respectively the indices of the first and of the last photograms of the pseudo–video; *so the pseudo–video execution doesn't include necessarily all the photograms of the atlas.*
The indexing criterion reflects the position of the photograms into the atlas grid, from the top to the bottom and from the left to the right. In this case we have decided to use all the 45 not–void photograms of the atlas chosen above.
The third argument permits to decide if to execute repetedly the pseudo–video or just once.
The fourth argument permits to decide how long (in milliseconds *(verify)*) has to be the residence time of each photogram onto the platform before is replaced by the next one.

– To fix forever onto the platform an only desider photogram.
You can do this by just executing the following instruction:

```
player.cellIndex = 20;
```

1.6. Check Point: A Sample Code

Outstanding, I think it's needed to create more than one sample code.